users can download either compiled JAR files of the JHDL system, or they can download and build JHDL from sources themselves. Documentation on the JHDL system is provided as well.

References

[1] P. Bellows, B. L. Hutchings. JHDL—An HDL for reconfigurable systems. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1998.

[2] P. Bertin, D. Roncin, J. Vuillemin. Programmable active memories: A performance assessment. In G. Borriello, C. Ebeling (eds.). *Research on Integrated Systems: Proceedings of the 1993 Symposium*, 1993.

[3] P. Bertin, H. Touati. PAM programming environments: Practice and experience. In D. A. Buell, K. L. Pocek (eds.). *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[4] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[5] P. Graham, B. Nelson, B. Hutchings. Instrumenting bitstreams for debugging FPGA circuits. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[6] P. S. Graham. *Logical Hardware Debuggers for FPGA-Based Systems*, Ph.D. thesis, Brigham Young University, 2001.

[7] K. S. Hemmert, J. L. Tripp, B. L. Hutchings, P. A. Jackson. Source level debugger for the Sea Cucumber synthesizing compiler. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[8] B. L. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting. A CAD suite for high-performance FPGA design. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1999.

[9] C. Iseli, E. Sanchez. A C++ compiler for FPGA custom execution units synthesis. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[10] W. J. Landaker, M. J. Wirthlin, B. L. Hutchings. Multitasking hardware on the SLAAC1-V reconfigurable computing system. *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, September 2002.

[11] J. L. Tripp, P. A. Jackson, B. L. Hutchings. Sea Cucumber: A synthesizing compiler for FPGAs. *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, September 2002.

[12] T. Wheeler, P. Graham, B. Nelson, B. Hutchings. Using design-level scan to improve FPGA design observability and controllability for functional verification. *Proceedings of the 11th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, August/September 2001.

PART III

MAPPING DESIGNS TO RECONFIGURABLE PLATFORMS

The chapters that follow cover the key mapping steps unique to field-programmable gate arrays (FPGAs) and reconfigurable targets. These steps include technology mapping to the primitive FPGA programmable gates (Chapter 13), placement of these gates (Chapters 14 through 16), routing of the interconnect between gates (Chapter 17), retiming of registers in the design (Chapter 18), and bitstream generation (Chapter 19). A final chapter summarizes a number of approaches to accelerating various stages of the mapping process (Chapter 20).

Placement is a difficult mapping problem, but is critical to the performance of the resulting reconfigurable design. As a result, it can be very slow, limiting the rate of the edit–compile–debug loop for reconfigurable application development, and the designs it produces may have longer cycle times than we would like. For these reasons, in addition to the general-purpose algorithms for placement covered in Chapter 14, algorithms that are highly optimized to exploit the regularity of datapaths are discussed in Chapter 15, and constructive approaches to layout are treated in Chapter 16. These more specialized approaches can significantly reduce placement runtime and often deliver placements that allow faster design operation.

As Chapters 13 through 20 demonstrate, there is a well-developed set of approaches and tools for programming reconfigurable applications. However, the tools are always slower than we might like them to be, especially as FPGA capacities continue to grow with Moore's Law. Moreover, the designs they produce are often too large or too slow, and the level at which we must program them is often lower than optimal. These deficiencies present ample opportunities for innovation and improvement in software support for reconfigurable systems.

For the designer who works on reconfiguration issues, the following chapters provide a look under the covers at the tools used to map designs and at the problems they must solve. It is important to understand which problems the tools are and are not solving and how well they can be expected to work. An understanding of the mapping flow and algorithms often helps the designer appreciate why tools may not produce

the quality of results expected and how the design could be optimized to obtain better results. Similarly, understanding the problems that the tools are solving helps the designer understand the trade-offs associated with higher- or lower-level designs and how to mix and match design levels to obtain the desired quality of results with minimal effort.

For the tool or software developer, this part covers the key steps in a traditional tool flow and summarizes the key algorithms used to map reconfigurable designs. With this knowledge the developer can rapidly assimilate conventional approaches and options and thus prepare to explore opportunities to improve quality of results, reduce tool time, or increase automation and raise the configurable design's level of abstraction.

CHAPTER 13

TECHNOLOGY MAPPING

Jason Cong
Department of Computer Science
California NanoSystems Institute
University of California–Los Angeles

Peichen Pan
Magma Design Automation, Inc.

Technology mapping is an essential step in an field-programmable gate array (FPGA) design flow. It is the process of converting a network of technology-independent logic gates into a network comprising logic cells on the target FPGA device. Technology mapping has a significant impact on the quality of the final FPGA implementation.

Technology-mapping algorithms have been proposed for optimizing area [29, 36, 58, 65], timing [9, 12, 13, 19, 21, 37, 58], power [2, 8, 34, 45, 52, 71], and routability [3, 67]. Mapping algorithms can be classified into those for general networks [13, 16] and those for special ones such as treelike networks [35, 36]. Algorithms for special networks may be applied to general ones through partitioning, with a possible reduction in solution quality.

Technology-mapping algorithms can be *structural* or *functional*. A structural mapping algorithm does not modify the input network other than to duplicate logic [12, 13]. It reduces technology mapping to a covering problem in which the technology-independent logic gates in the input network are covered with logic cones such that each cone can be implemented using one logic cell—for example, a K-input lookup table (K-LUT)—for LUT-based FPGAs. Figure 13.1 is an example of structural mapping. The logic gates in the original network (a) are covered with three logic cones, each with at most three inputs, as indicated (b). Note that node i is included in two cones and will be duplicated. The corresponding mapping solution (c) comprises three 3-LUTs.

A functional mapping algorithm, on the other hand, treats technology mapping in its general form as a problem of Boolean transformation/decomposition of the input network into a set of interconnected logic cells [15, 48, 58, 60]. It mixes Boolean optimization with covering. Functional mapping algorithms tend to be time consuming, which limits their use to small designs or to small portions of a design.

Note: This work is partially supported by the National Science Foundation under grant number CCF 0530261.

Recent advances in technology mapping try to combine mapping with other steps in the design flow. Such integrated mapping algorithms have the potential to explore a larger solution space than is possible with just technology mapping and thus have the potential to arrive at mapping solutions with better quality. For example, algorithms have been proposed to combine logic synthesis with covering to overcome the limitations of pure structural mapping [11, 22, 57].

13.1 STRUCTURAL MAPPING ALGORITHMS

Technology mapping is part of a logic synthesis flow, which typically consists of three steps. First, the initial network is optimized using technology-independent optimization techniques such as node extraction/substitution and don't-care optimization [33]. Second, the optimized network is decomposed into one consisting of 2-input gates plus inverters (that is, the network is *2-bounded*) to increase flexibility in mapping [12, 36]. Third, the actual mapping takes place, with the goal of covering the 2-bounded network with *K*-LUTs while optimizing one or more objectives. In the remaining discussion, we assume that the input network is 2-bounded.

A logic network can be represented as a graph where the nodes represent logic gates, primary inputs (PIs), and primary outputs (POs). The edges represent the interconnects or wires. A *cut* of a node v is a set of nodes in the input network such that every path from the primary inputs or sequential element outputs to v contains at least one node in the set. A *K-cut* is a cut with at most K nodes. For example, $\{a, b, z\}$ is a 3-cut for the node y in the network in Figure 13.1(a). Given a K-cut for v, we can obtain a K-LUT for v by collapsing the gates in the logic cone between the nodes in the cut, v, including v itself. For the 3-cut $\{a, b, z\}$ for y, the 3-LUT for y in Figure 13.1(c) is derived from the corresponding cone indicated for y in Figure 13.1(b).

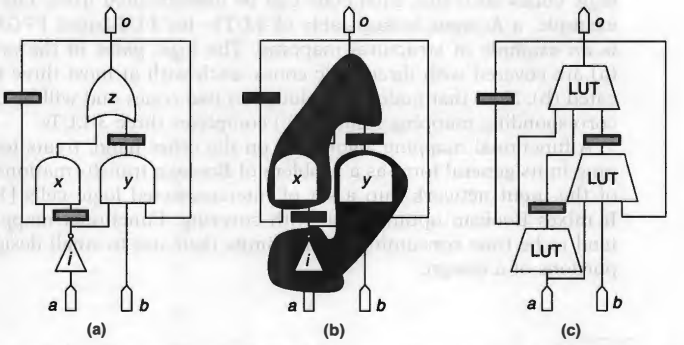


FIGURE 13.1 ■ Structural technology mapping: (a) original network, (b) covering, and (c) mapping solution.

Most structural mapping algorithms are based on the dynamic programming technique. They typically consist of the following steps:

1. Cut generation/enumeration
2. Cut ranking
3. Cut selection
4. Final mapping solution generation

Cut generation obtains one or more cuts that will be used to generate LUTs; it is discussed in the next section. Cut ranking evaluates the cuts obtained in cut generation to see how good they are based on the mapping objectives. It assigns a label or cost to each cut by visiting the nodes in a topological order from PIs to POs. Cut selection picks a cut with the best label for each node and is typically done in reverse topological order from POs to PIs. Cut ranking and selection may be carried out multiple times to refine solution quality.

After the final cut selection, a mapping solution is generated using the selected cuts. In this step, the nodes are visited in the reverse topological order, starting from POs and going back to PIs. At each node, a cut with the best label is selected and the corresponding LUT is added to the solution. Next, the nodes that drive the LUT are visited. This process is repeated until only PIs are left. At that point, a complete mapping solution is obtained.

13.1.1 Cut Generation

Early mapping algorithms combine cut generation and selection to determine one or a few "good" cuts for each node. The most successful example is the FlowMap algorithm, which finds a single cut with optimal mapping depth at each node via max-flow computation [16]. It computes the optimal mapping depth of each node in a topological order from PIs to POs, and at each node uses a max-flow formulation to test whether that node can have the same optimal mapping depth as the maximum depth of its input nodes. If not, the depth is set to one greater than the input nodes' maximum depth. It is shown that these are the only two possible mapping depths. The FlowMap algorithm was the first polynomial time algorithm to find a depth-optimal mapping solution for LUT-based FPGAs.

In practice, K, the number of inputs of the LUTs, is a small constant typically ranging between 3 and 6. It becomes practical to enumerate all K-cuts for each node. With all cuts available, we have additional flexibility in selecting cuts to optimize the mapping solution.

Cuts can be generated by a traversal of the nodes in a combinational network (or the combinational portion of a sequential network) from PIs to POs in a topological order [29, 67]. Let $\Phi(v)$ denote the set of all K-cuts for a node v. For a PI, $\Phi(v)$ contains only the trivial cut consisting of the node itself, that is, $\Phi(v) = \{\{v\}\}$. For a non-PI node v with two fanin nodes, u_1 and u_2, $\Phi(v)$ can be computed by merging the sets of cuts of u_1 and u_2 as follows:

$$\Phi(v) = \{\{v\} \cup \{c_1 \cup c_2 | c_1 \in \Phi(u_1),\ c_2 \in \Phi(u_2),\ |c_1 \cup c_2|\} <= K\} \tag{13.1}$$

In other words, the set of cuts of v is obtained by the pairwise union of the cuts of its fanin nodes and then the elimination of those cuts with more than K nodes. Note that the trivial cut is added to the set. This is necessary so the nodes driven by v can include v in their cuts.

13.1.2 Area-oriented Mapping

For LUT mapping, the area of a mapping solution can be measured by the total number of LUTs. It has been shown that finding an area-optimal mapping solution is NP-hard [35]. Therefore, it is unlikely that there is an accurate way to rank cuts for area. The difficulty of precise area estimation is mainly due to the existence of multiple fanout nodes. In fact, for treelike networks, area-optimal mapping solutions can be determined in polynomial time [35].

Cong et al. [29] proposed the concept of *effective area* as a way to rank and select cuts for area. A similar concept, *area flow*, was later proposed by Manohararajah et al. [55]. Intuition regarding effective area is to distribute the area for a multi-fanout node to its fanout nodes so that logic sharing and reconvergence can be considered during area cost propagation. Effective areas are computed in a topological order from PIs to POs. The effective area $a(v)$ of a PI node v is set to zero. Equation 13.2 is used to compute the effective area of a cut:

$$a(c) = \left(\Sigma_{u \in c} \left[a(u)/|output(u)|\right]\right) + A_c \tag{13.2}$$

where A_c is the area of the LUT corresponding to the cut c. The area cost of a non-PI node can then be set to the minimum effective area of its cuts: $a(v) = \min\{a(c) | \forall u \in \Phi(v)\}$.

It should be pointed out that effective area may not account for the situation where the node may be duplicated in a mapping solution. In the example shown in Figure 13.2, with $K = 3$, the LUT for w is introduced solely for the LUT for v. However, in effective area computation, only one-half is counted for v, and as a result the LUT for w is undercounted. In this example, the sum of effective area of the POs is 2.5 whereas the mapping solution has three LUTs. In general, effective area is a lower bound of the actual area.

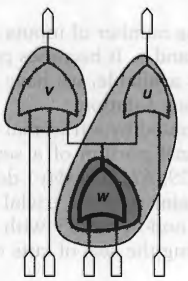


FIGURE 13.2 ■ Inaccuracy in effective area.

The PRAETOR algorithm [29] is an area-oriented mapping algorithm that ranks cuts using effective area. It further improves the basic mapping framework with a number of area reduction techniques. One such technique is to encourage the use of common subcuts. A cut for a fanout of a node v induces a cut for v (perhaps the trivial cut consisting of v itself). If two fanouts of v induce different cuts for v, the most likely result will be an area increase due to the need to duplicate v and possibly some of its predecessor nodes. To alleviate this problem, PRAETOR sorts and selects cuts with the same effective area in a predetermined order to avoid arbitrary selection. It assigns an integer ID to each node and then sorts all cuts with the same effective area according to the lexicographic order based on the IDs of the nodes in the cuts. The first cut with minimum effective area for each node is selected.

Another area reduction technique introduced in PRAETOR is to carry out cut selection twice. The nodes with LUTs selected in the first pass are declared nonduplicable and can only be covered by LUTs for themselves in the second pass. This encourages selection of cuts with less duplication. As an example, suppose that in the first pass of cut selection, the mapping solution shown in Figure 13.3(a), with four LUTs, is selected. In the second pass, the LUT containing v and u_1 is excluded from consideration for u_1. This exclusion will also encourage the selection of the cut that results in the LUT containing a for u_1. As a result, the mapping algorithm generates, in the second pass, the mapping solution in Figure 13.3(b), with only three LUTs. Experimental results show that PRAETOR can significantly improve area over previous algorithms.

The IMap algorithm proposed by Manohararajah et al. [55] is another mapping algorithm targeting area optimization. It introduced two enhancements: (1) iteration between cut ranking and cut selection multiple times, and (2) adjustment of the area costs between successive iterations using history information. In the effective area formula (equation 13.2), the fanout count of u in the initial network, $|output(u)|$, is used to estimate the fanout count of the LUT rooted at u in the mapping solution. In the IMap algorithm between iterations, the fanout

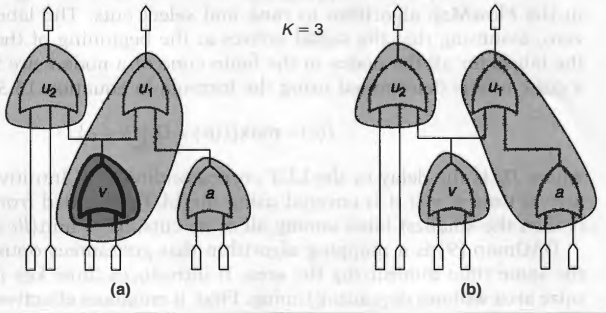


FIGURE 13.3 ■ Effect of excluding cuts across nonduplicable nodes: (a) initial mapping solution, and (b) improved solution with better area.

count estimation is updated by using a weighted combination of the estimated and the real fanout counts in previous iterations. As a result, equation 13.2 becomes $a(c) = \left(\Sigma_{u \in c}\left[a(u)/estimated\_fc(u)\right]\right) + A_c$, where $estimated\_fc(u)$ denotes the estimated fanout count for the current iteration.

Ling et al. [54] proposed a mixed structural and functional area-mapping algorithm that starts with a mapping solution (e.g., generated by a structural mapping algorithm). The key idea is a Boolean satisfiability (SAT) formulation for the problem of mapping a small circuit with up to ten inputs into the smallest possible number of LUTs. The algorithm iteratively selects a small logic cone to remap to fewer LUTs using an SAT solver. It is shown that for some highly structured (albeit small) designs, area can be improved significantly.

Most area optimization techniques are heuristic. A natural question is how close the mapping solutions obtained using existing mapping algorithms are from optimal. Cong and Minkovich [24] constructed a set of designs with known optimal area-mapping solutions, called LEKO (logic synthesis examples with known optimal bounds) examples, and tried existing academic algorithms and commercial tools on them. The average gap from optimal varied from 5 to 23 percent. From LEKO examples, they further derived LEKU (logic synthesis examples with known upper bounds) examples that require both logic optimization and mapping. Existing algorithms perform poorly on LEKU examples, with an average optimality gap of more than 70 times. This indicates that more research is needed in area-oriented mapping and optimization.

13.1.3 Performance-driven Mapping

The FlowMap algorithm and its derivatives can find a mapping solution with optimal depth. Recent advances in delay mapping focus on achieving the best performance with minimal area.

Exact layout information is not available during technology mapping in a typical FPGA design flow. Mapping algorithms usually ignore routing delays and try to optimize the total cell delays on the longest combinational paths in the mapping solution.

Most delay optimal mapping algorithms use the labeling scheme introduced in the FlowMap algorithm to rank and select cuts. The label of a PI is set to zero, assuming that the signal arrives at the beginning of the clock edge. After the labels for all the nodes in the fanin cone of a node v are found, the label of a cut c of v is determined using the formula in equation 13.3:

$$l(c) = \max\{l(u) + D_c | \forall u \in c\} \tag{13.3}$$

where D_c is the delay of the LUT corresponding to c. Intuitively, $l(c)$ is the best arrival time at v if it is covered using the LUT generated from c. The label of v is then the smallest label among all of its cuts: $l(v) = \min\{l(c) | \forall c \in \Phi(v)\}$.

DAOmap [9] is a mapping algorithm that guarantees optimal delay while at the same time minimizing the area. It introduces three key techniques to optimize area without degrading timing. First, it enhances effective area computation to make it better avoid node duplication. Second, it applies area optimization

techniques on noncritical paths. Last, it uses an iterative cut selection procedure to explore and perturb the solution space to improve solution quality.

DAOmap first picks cuts with the minimum label for each node. From those, it then picks one with minimum effective area. Furthermore, when there is positive slack, which is the difference between required time and arrival time at a node, it picks a cut with as small an area cost as possible under the condition that the timing increase does not exceed the slack.

Recognizing the heuristic nature of effective area computation, DAOmap also employs the technique of multiple passes of cut selection. Moreover, it adjusts area costs based on input sharing to encourage using nodes that have already been contained in selected cuts. This reduces the chance that a newly picked cut cuts into the interior of existing LUTs. Between successive iterations of cut selection, DAOmap also adjusts area cost to encourage selecting cuts containing nodes with a large number of fanouts in previous iterations. There are a few other secondary techniques used in DAOmap. The interested reader is referred to Chen and Cong [9] for details.

Based on the results reported, DAOmap can improve the area by about 13 percent on a large set of academic and industrial designs while maintaining optimal depths. It is also many times faster than previous mapping algorithms based on max-flow computation, mainly because of efficient implementation of cut enumeration.

A recent delay optimal mapping algorithm introduced several techniques to improve area while preserving performance [57]. Like DAOmap, this algorithm goes through several passes of cut selection, with each pass selecting cuts with better areas among the cuts that do not degrade timing. It is also based on the concept of effective area (or area flow). However, it does cut selection from PIs to POs instead of from POs to PIs, as in most other algorithms. With this processing order, the algorithm tries to use timing slacks on nodes close to PIs to reduce area cost. This is based on the observation that logic is typically denser when close to PIs, so slack relaxation is more effective for nodes closer to PIs. Experimental data shows 7 percent better area over DAOmap for the same optimal depths.

13.1.4 Power-aware Mapping

Power has become a major concern for FPGAs [51, 68]. Dynamic power dissipation in FPGAs results from charging and discharging capacitances. It is determined by the switching activities and the load capacitance of the LUT outputs and can be captured by equation 13.4:

$$P = \frac{1}{2}\Sigma_v C_v \cdot f_v \cdot V^2 \tag{13.4}$$

where C_v is the output load capacitance of node v, f_v is the switching activity of node v, and V is the supply voltage. Given a fixed supply voltage, power consumption in a mapped netlist is determined by switching activities and load capacitance of the LUT outputs.

Because technology mapping for power is NP-hard [34], a number of heuristic algorithms have been proposed. Most power-aware mapping algorithms try to reduce switching activities by hiding nodes with high switching activities inside LUTs, hence leaving LUTs with small output-switching activities in the mapped netlist.

Anderson and Najm [2] proposed a mapping algorithm to reduce switching activities and minimize logic duplication. Logic duplication is necessary to optimize timing and area, but can potentially increase power consumption. The algorithm uses the following power-aware cost function to rank cuts: $Cost(c) = l(c) + \beta \cdot P(c) + \gamma \cdot R(c)$, where $l(c)$ is the depth label of the cut c as given in equation 13.3 and $P(c)$ and $R(c)$ are the power and replication costs of the cut, respectively. The weighting factors β and γ can be used to bias the three cost terms. Anderson and Najm suggest a very small β to get a depth-optimal mapping solution with minimal power.

Power cost $P(c)$ is defined in such a way that it encourages absorbing high-activity connections inside LUTs. The replication cost tries to discourage logic duplication on timing noncritical paths. Power savings of over 14 percent were reported over timing-oriented mapping algorithms when both targeted optimal depths. When the mapping depth was relaxed by one level over optimal, additional power reduction of about 8 percent for 4-LUTs and 10 percent for 5-LUTs was reported.

One serious limitation of the power-based ranking in Anderson and Najm [2] is that it cannot account for multiple fanouts and reconvergence, which are common in most practical designs. Chen et al. [8] proposed a low-power technology-mapping algorithm based on an improved power-aware ranking in equation 13.5:

$$P(c) = (\Sigma_{u \in c}[P(u)/|output(u)|]) + U_c \tag{13.5}$$

where U_c is a cost function that tries to capture power contributed by the cut c itself. Experimental results show that this algorithm outperforms previous power-aware mapping algorithms. It has also been extended to handle dual supply voltage FPGA architectures.

13.2 INTEGRATED MAPPING ALGORITHMS

Technology mapping is a step in the middle of an FPGA design flow. Technology-independent optimization is carried out before mapping; placement is carried out after. Sequential optimization such as retiming can be carried out before or after mapping. A separate approach can miss the best overall solutions even if we can solve each individual step optimally. In the section that follows we discuss mapping algorithms that combine mapping with other steps in the design flow.

13.2.1 Simultaneous Logic Synthesis, Mapping

Technology-independent Boolean optimizations carried out prior to technology mapping can significantly impact the mapping solution. During technology-independent optimization, we have the freedom to change the network structures,

but accurate estimation of their impact on mapping is not available. During technology mapping, we can achieve optimal or close to optimal solutions using the algorithms discussed in Section 13.1. However, we are stuck with a fixed network. It is desirable to capture the interactions between logic optimization and mapping to arrive at a solution with better quality.

Lossless synthesis has been proposed by Mishchenko et al. [57] as a way to consider technology-independent optimization during mapping. It is based on the concept of *choice networks*, which is similar to the concept of mapping graphs [11, 49]. A choice network contains choice nodes that encode functionally equivalent but structurally different alternatives. The algorithm operates on a simple yet powerful data structure called *AIG*, which is a network of AND2 and INV gates. A combination of SAT and simulation techniques is used to detect functionally equivalent points in different networks and compress them to form one choice network.

Figure 13.4 illustrates the construction of a network with choices from two equivalent networks with different structures. The nodes x_1 and x_2 in the two networks are functionally equivalent. They are combined in an equivalence class in the choice network, and an arbitrary member (x_1 in this case) is set as the class representative. Note that p does not lead to a choice because its implementation is structurally the same in both networks. Similarly, o does not lead to a choice node.

Rather than try to come up with one "good" optimized network before mapping, the algorithm proposed by Mishchenko et al. [57] accumulates choices by combining intermediate networks seen during logic synthesis to generate a network with many choices. In a sense, it does not make judgments on the goodness of the intermediate networks but defers that decision to the mapping phase, when the best combination of these choices is selected. In the final mapping solution, different sections may come from different intermediate networks. For example, the timing-critical sections of the final mapping solution may come

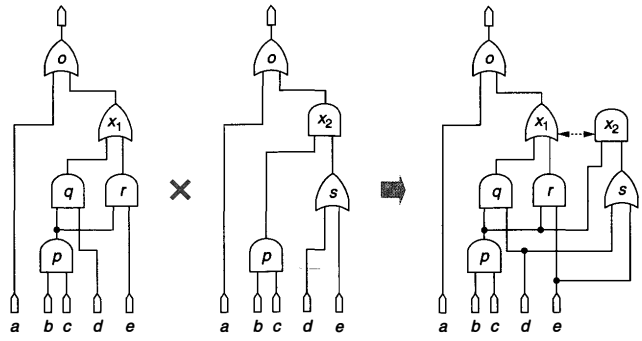


FIGURE 13.4 ■ Combining networks to create a choice network.

from networks optimized for timing, while the timing noncritical sections of the final mapping solution may come from networks optimized for area.

For mapping on choice networks, cut generation and cut ranking are extended to choice nodes. For example, the set of cuts of a choice node is simply the union of the sets of cuts of all of that node's fanin nodes. Similarly, the label of a choice node is the smallest one among the labels of its fanin nodes. The rest of the approach is similar to a conventional mapping algorithm. Results reported by Mishchenko et al. [57] show that both timing and area can be improved by over 7 percent on a set of benchmark designs compared to applying mapping to just one "optimized" network.

13.2.2 Integrated Retiming, Mapping

Retiming (discussed in Chapter 18) is an optimization technique that relocates flip-flops (FFs) in a network while preserving functionality of the network [50]. Retiming can shift FF boundaries and change the timing. If retiming is applied after mapping, mapping may optimize the wrong paths because the critical paths seen during mapping may not be critical after the FFs are repositioned. On the other hand, if retiming is applied before mapping, it will be carried out using less accurate timing information because it is applied to an unmapped network. In either approach, the impact of retiming on cut generation cannot be accounted for.

The network in Figure 13.5(a) is derived from the design in Figure 13.1(a) by retiming the FFs at the outputs of y and i to their inputs. After the retiming, all gates can be covered with one 3-LUT, as indicated in (a). The corresponding mapping solution is shown in (b). This mapping solution is obviously better than the one in Figure 13.1(c) in both area and timing.

Pan and Liu [63] proposed a polynomial time-mapping algorithm that can find a solution with the best cycle time in the combined solution space of

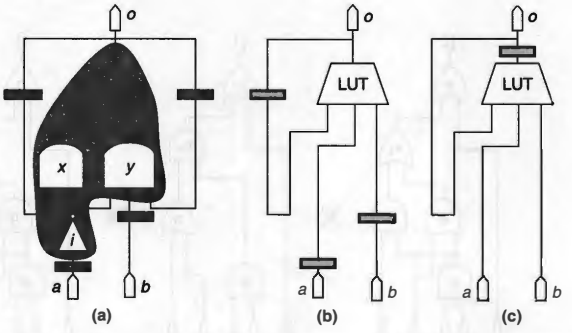


FIGURE 13.5 ■ Retiming, mapping: (a) retiming and covering, (b) mapping solution, and (c) retimed solution.

retiming and mapping. In other words, the solution obtained is the best among all possible ways of retiming and mapping a network. Improved algorithms were later proposed that significantly reduce runtime while preserving the optimality of the final mapping solution [25, 27]. These algorithms, like the FlowMap algorithm, are all based on max-flow computation.

A cut enumeration-based algorithm for integrated retiming and mapping was proposed by Pan and Lin [61]. In it, cut generation is extended to go across FF boundaries to generate sequential cuts. In a network with FFs, a gate may go through zero or more FFs in addition to logic gates before reaching gate v. To capture this information, an element in a cut for a node v is represented as a pair consisting of the driving node u and the number of FFs d on the paths from u to v, denoted by u^d. Note that one node may reach another node through paths with different FF counts. In that case, the node will appear in the cut multiple times with different values of d. For example, for the cone in Figure 13.5(a), the corresponding cut is $\{z^1, a^1, b^1\}$. Pan and Lin [61] suggested an iterative procedure to determine the sequential cuts for all nodes.

To consider retiming effect, the concept of labels is extended using sequential arrival times [62, 63]. The label of a cut c is now defined as follows:

$$l(c) = \max\{l(u) - d \cdot \phi + D_c | \forall u^d \in c\} \tag{13.6}$$

where ϕ is the target cycle time and D_c is the delay of the LUT corresponding to c. The combinationl cut formula (equation 13.3) can be viewed as a special case of equation 13.6 when $d = 0$. As in combinational mapping algorithms, the label of a gate v is the minimum of the labels of its cuts: $l(v) = \min\{l(c) | \forall c \in \Phi(v)\}$. The label of each PI is zero, and the label for each PO is that of its driver.

Pan and Lin's algorithm finds the labels for cuts and nodes through successive approximation by going through the nodes in the initial network in passes. After the labels for all nodes are computed and the target cycle time is determined to be achievable, the next step is to generate a mapping solution. As in the combinational case, a mapped network is constructed starting from POs and going backward. At each node v, the algorithm selects one of the cuts that realize the node's label and then moves on to select a cut for u if u^d is in the cut selected for v. On the interconnection from u to v, d FFs are inserted. To obtain the final mapping solution with a cycle time of ϕ, the algorithm retimes the LUT for each non-PI/PO node v by $\lceil l(v)/\phi \rceil - 1$. For the initial network in Figure 13.1(a), the final mapping solution with optimal cycle time generated by the algorithm is shown in Figure 13.5(c). Experimental results show that the algorithm is very efficient and consistently produces mapping solutions with better performance than combinational depth optimal mapping followed by optimal retiming.

13.2.3 Placement-driven Mapping

One drawback of the conventional mapping flow is the lack of accurate timing information on interconnects. Most algorithms use logic depth to measure timing. However, optimal-depth mapping solutions may not always be good

after placement. To overcome this problem, we need to combine mapping with placement so that mapping can see more accurate interconnect information.

A number of algorithms try to carry out placement and mapping simultaneously [3, 6, 53, 59, 69]. For example, the MIS-pga algorithm of Murgai et al. [59] performs iterative logic optimization and placement. Chen et al. [6] proposed an algorithm that tightly couples technology mapping and placement by mapping each cell and placing it at the same time. In practice, such integrated approaches suffer a serious limitation: Because of the complexity of the combined problem, simple mapping, placement techniques are employed. As a result, the benefit of the combined approach is diminished.

Another approach is to iterate between mapping and placement (or placement refinement). Here, the design is first mapped and placed. Then the netlist is back-annotated and remapped under the given placement. This process can be repeated until a satisfactory solution is found. Figure 13.6 outlines the major steps in the iterative mapping and placement algorithm proposed by Lin et al. [53]. The key step is placement-driven remapping. The remapping step may make the placement illegal—for example, it may place more than one cell at the same location. If this happens, the placement needs to be legalized and refined.

Lin et al.'s algorithm [53] uses table lookup to estimate interconnect delays based on placement locations. Given two locations, it looks up the estimated delay in a prestored table for the wiring between the two locations. This is more accurate and realistic than the "fixed" interconnect delays used in earlier layout-based mapping algorithms [56, 72].

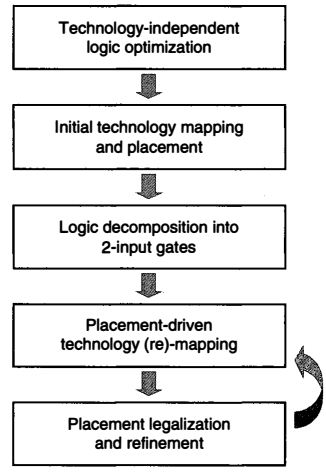


FIGURE 13.6 ■ Iterative mapping, placement.

One difficulty in placement-driven mapping is that the placement may not become legal because of cell overlaps. Another is that timing predicted in the labeling phase may be unrealizable because of congestion in the new mapping solution. Congestion means that many LUTs are assigned to a small region, which requires many cell relocations to legalize the placement, which in turn, perturbs the placement and eventually the timing. To overcome this problem, the algorithm employs an iterative process with multiple passes of cut selection. Each pass uses the cell congestion information gathered during previous iterations to guide the mapping decisions. Several techniques have been proposed to relieve congestion. One is a hierarchical area control scheme to evaluate the local congestion cost, in which the chip is divided into bins with different granularities. Area increase is tallied in bins, and penalty costs are given to bins with area overflows.

Once a mapping solution is generated, the algorithm invokes timing-driven legalization that moves overlapping cells to empty locations in their neighborhood based on the timing slack available to the cells. Finally, a simulated annealing-based placement refinement phase is carried out to improve performance. Experimental results show that the algorithm can improve timing by more than 12 percent, with minimal area penalty due to remapping.

13.3 MAPPING ALGORITHMS FOR HETEROGENEOUS RESOURCES

Up to this point, we have assumed that all logic cells are LUTs with a uniform input size K. In reality, commercial FPGA architectures contain heterogeneous resources (e.g., LUTs of different input sizes, embedded memory, and PLA-like logic cells). We briefly summarize mapping algorithms that target or take advantage of such architectural features.

13.3.1 Mapping to LUTs of Different Input Sizes

There are a number of commercial FPGA architectures that support LUTs with multiple input sizes on the same device. Mapping algorithms have been proposed to optimize area [29, 39, 40, 43] and timing [30, 32].

In the special case of tree networks, Korupolu et al. [43] presented a polynomial area optimal algorithm. For general networks, the PRAETOR algorithm discussed in Section 13.1.2 can be applied to these architectures by assigning different area costs for LUTs with different input sizes.

For timing optimization, the algorithm proposed by Cong and Xu [30] is an extension of FlowMap. Like FlowMap, it is also based on flow computation and can be cast in the cut enumeration framework. Assume that there are two types of LUTs with input sizes K_1 and K_2, and delays d_1 and d_2, where $K_1 < K_2$, $d_1 < d_2$. We can enumerate all K_2-cuts. When labeling a cut, we can set its delay to d_1 or d_2 depending on its size. With this simple modification, an algorithm for homogeneous LUT architectures can be used for architectures with different LUT sizes.

When there are resource bounds on available LUTs of different sizes, the mapping problem becomes NP-hard. Assuming that there can be at most r K_2-LUTs, a heuristic algorithm was proposed that starts out by finding a mapping solution without considering resource bounds [31]. If the current mapping solution meets the resource bound, it stops. If not, it increases d_2, the delay of K_2-LUTs, and solves the unconstrained version again, which should lead to another mapping solution with a decreased number of K_2-LUTs. This process is repeated until the resource bound is met.

13.3.2 Mapping to Complex Logic Blocks

FPGA devices typically contain additional logic that, together with LUTs, can form complex *programmable logic blocks* (PLBs). PLBs can implement complex logic functions. Figure 13.7 shows two PLBs that consist of LUTs and logic gates and can implement functions of up to nine inputs.

A simple approach to PLB mapping is to map the initial network to the constituent cells inside the PLBs. For example, for a device with the PLB in Figure 13.7(a), we can first map the initial network to 3-LUTs and 4-LUTs. Afterwards, the LUTs are clustered to obtain a network of PLBs. Such a two-step approach is obviously suboptimal.

Recent approaches try to map directly to PLBs [13, 23, 47, 65]. The cut enumeration framework can still be used after enhancements. Because a PLB can have more inputs than a typical LUT, a node may have too many cuts. Intelligent cut pruning, using techniques such as those proposed by Chatterjee et al. [5] and Ling et al. [54], is necessary to avoid long runtime and memory explosion. Unlike in the case of LUTs, a PLB has limited functional capability in that it cannot implement all of the functions of its inputs. For example, the PLB in Figure 13.7(b) can implement all functions of up to five inputs, but it can only implement some of the functions with six inputs. An essential step in PLB mapping is Boolean matching, which, given a cut, decides if the corresponding logic cone can be implemented by a PLB.

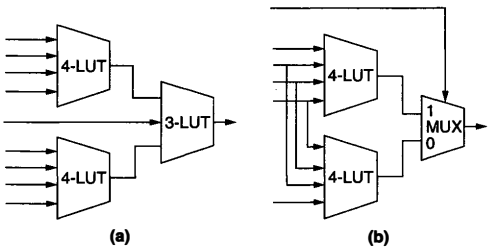


FIGURE 13.7 ■ Two PLB examples.

Algorithms for Boolean matching for PLBs can be classified into two categories: decomposition based [13, 23] and satisfiability (SAT) based [25, 54, 65]. Decomposition-based Boolean matching tries to decompose the input function according to the structure of the target PLB using functional decomposition. Cong and Hwang [23] proposed matching procedures for a wide variety of common PLBs.

A drawback of decomposition-based Boolean matching is that each PLB needs a specialized matching procedure. Decomposition-based Boolean matching can also be slow and memory intensive because of extensive use of BDD operations. On the other hand, SAT-based Boolean matching encodes the function, the target PLB, and their matching in a Boolean expression in conjunctive normal form (CNF). Then it leverages an efficient SAT solver (e.g., the one proposed by Moskewicz et al. [58]) to check whether the PLB can be configured to implement the function. The size of the CNF expression can have significant impact on the runtime of an SAT-based matching algorithm. An improved SAT formulation with smaller expressions was proposed recently by Cong and Minkovich [25].

13.3.3 Mapping Logic to Embedded Memory Blocks

On-chip memory has become a common feature of high-performance FPGAs. Dedicated embedded memory blocks (EMBs) can be used to improve clock frequencies and lower costs for large designs that require memory. If a design does not need all the available EMBs, unused ones can be employed to implement logic, which essentially turns them into large multi-input multi-output LUTs.

EMBs usually have configurable widths and depths, so they can be used to implement functions with different numbers of inputs/outputs. For example, a 2K-bit memory with configurations 2048×1, 1024×2, and 512×4 can be used to implement an 11-input/1-output, 10-input/2-output, or 9-input/4-output logic function, respectively.

Mapping logic to EMBs is typically done as a postprocessing step after LUT mapping. These algorithms start with an optimized LUT-mapping solution and then pack groups of LUTs into unused EMBs [26, 70]. The SMAP algorithm [70] maps one EMB at a time. It begins by selecting a seed node. A fanin cone of the seed node is generated by finding a d-feasible cut that covers as many nodes as possible, where d is the bit width of the address line of the target EMB. Because d is considerably large, flow-based cut generation is used. After the cone is generated, the output selection process selects signals to be the EMB outputs. Output selection tries to select a set of signals so that the resulting EMB can eliminate as many LUTs as possible. This is done by assigning each node a score that reflects the number of eliminated nodes if the node is selected. The w highest-scoring nodes are selected as the EMB outputs, where w is the number of outputs of the target EMB.

The selection of the seed node is critical for this method. The algorithm tests each candidate node and selects the one that leads to the maximum number of

eliminated LUTs. Heuristics were introduced to consider EMBs with different configurations and to preserve timing.

Another algorithm, EMB_Pack, proposed by Cong and Xu [26], takes a slightly different approach. It finds the logic to map to EMBs altogether instead of one at a time, as in SMAP, which can potentially find better mapping.

13.3.4 Mapping to Macrocells

Complex programmable logic devices (CPLDs) are a class of programmable logic devices that are more coarse grained than typical FPGAs. Each CPLD logic cell (called Pterm block) is essentially a programmable logic array (PLA) that consists of a set of product terms (Pterms) with multiple outputs. A Pterm block can be characterized by a 3-tuple (k,m,p) where k is the number of inputs, p is the number of outputs, and m is the number of Pterms for the block. The input size k is typically much larger than that of FPGA logic cells.

Relatively speaking, there is much less mapping work reported for CPLDs. A fast heuristic partition method for PLA-based structures was presented by Hasan et al. [38]. The DDMap algorithm [42] adapts a LUT mapper for CPLD mapping. It uses wide cuts to form big LUTs and decomposes the big LUTs into Pterms allowed in the target CPLD. Packing is used to form multi-output Pterm cells. An area-oriented mapping algorithm was proposed for CPLDs by Anderson and Brown [1]. Cong et al. [20] investigated an FPGA architecture consisting of single-output Pterm blocks, and proposed a timing-oriented mapping algorithm.

PLAmap is a timing-oriented mapping algorithm for CPLDs [7]. Like the LUT mapping algorithms discussed earlier, it has a labeling phase and a mapping phase. In the labeling phase, it tries to find the minimal mapping depth for each node using a logic cell $(k,m,1)$—that is, a single-output Pterm block, assuming that each logic cell has one unit delay. The labeling procedure is based on Lawler et al.'s clustering algorithm [46]. Let l be the largest label of the nodes in the fanin cone of a node. The algorithm forms a cluster for the node by grouping it with all nodes in its fanin cone with the label l. If the cluster can be implemented by a $(k,m,1)$ cell, the node is assigned the label l; otherwise, the node gets the label $l+1$ with a cluster consisting of the node itself. Note that this is a heuristic in that the label may not be the best because of the so-called *nonmonotone property* [7]. The mapping phase is done in reverse topological order from the POs. The algorithm tries to merge the clusters generated in the labeling phase to form (k, m, p) cells whenever possible. Cluster merging is done in such a way that duplication is minimized and the labels of the POs do not exceed the performance target. Experimental results show that PLAmap outperforms commercial tools and other algorithms with no (or a very small) area penalty.

Pterm blocks or macrocells are suitable for implementing wide-fanin, low-density logic, such as finite-state machines. They can potentially complement fine-grained LUTs to improve both performance and utilization. Device architectures with a mixture of LUTs and Pterm blocks or macrocells have been suggested to take advantage of different types of logic cells. Technology mapping algorithms have been proposed for such hybrid architectures [41, 42, 44].

13.4 SUMMARY

This chapter discussed technology mapping algorithms for FPGAs. Emphasis was placed on state-of-the-art algorithms that have been, or most likely will be, reduced to practice. We discussed mapping algorithms for different objectives, such as area, timing, and power, as well as mapping algorithms that take advantage of heterogeneous resources in modern FPGA devices.

FPGA technology mapping has been and continues to be a subject of active research. A general trend is to integrate technology mapping with other steps in the FPGA design flow to improve the quality of final implementations (e.g., combining mapping and clustering [10]).

As semiconductor technologies advance, new FPGA architecture features are being introduced to improve area utilization, performance, and power consumption. For example, architectures have been introduced or proposed that use large LUTs (much larger than traditional 4-/5-LUTs) or multiple supply voltages. New mapping techniques are being developed to take advantage of these architecture features.

References

[1] J. H. Anderson, S. D. Brown. Technology mapping for large complex PLDs. *ACM/IEEE Design Automation Conference*, 1998.

[2] J. H. Anderson, F. N. Najm. Power-aware technology mapping for LUT-based FPGAs. *IEEE International Conference on Field-Programmable Technology*, 2002.

[3] N. Bhat, D. D. Hill. Routable technology mapping for LUT FPGAs. *IEEE International Conference on Computer Design*, 1992.

[4] S. C. Chang, M. Marek-Sodowska, T. Hwang. Technology mapping for LUT FPGA based on decomposition of binary decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(10), October 1996.

[5] S. Chatterjee, A. Mishchenko, R. Brayton. Factor cuts. *International Conference on Computer-Aided Design*, 2006

[6] C. Chen, Y. Tsay, Y. Hwang, T. Wu, Y. Lin. Combining technology mapping, placement for delay-optimization in FPGA designs. *International Conference on Computer-Aided Design*, 1993.

[7] D. Chen, J. Cong, M. Ercegovac, Z. Huang. Performance-driven mapping for CPLD architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22(10), October 2003.

[8] D. Chen, J. Cong, F. Li, L. He. Low-power technology mapping for FPGA architectures with dual supply voltages. *International Symposium on Field-Programmable Gate Arrays*, February 2004.

[9] D. Chen., J. Cong. DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs. *International Conference on Computer-Aided Design*, 2004.

[10] D. Chen, J. Cong, J. Lin. Optimal simultaneous mapping, clustering for FPGA delay optimization. *ACM/IEEE Design Automation Conference*, 2006.

[11] G. Chen, J. Cong. Simultaneous logic decomposition with technology mapping in FPGA designs. *International Symposium on Field-Programmable Gate Arrays*, 2001.

[12] K. C. Chen, J. Cong, Y. Ding, A. B. Kahng, P. Trajmar. DAGmap: Graph-based FPGA technology mapping for delay optimization. *IEEE Design and Test of Computers* 9(3), September 1992.
[13] M. Chikodikar, S. Laddha, A. Sirasao. A technology mapper for Xilinx FPGAs. *Tenth International Conference on VLSI Design*, January 1997.
[14] J. Cong, Y. Ding. An optimal technology-mapping algorithm for delay optimization in lookup table–based FPGA designs. *International Conference on Computer-Aided Design*, November 1992.
[15] J. Cong, Y. Ding. Beyond the combinatorial limit in depth minimization for LUT-based FPGA designs. *International Conference on Computer-Aided Design*, 1993.
[16] J. Cong, Y. Ding. FlowMap: An Optimal technology-mapping algorithm for delay optimization in lookup table–based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13(1), January 1994.
[17] J. Cong, Y. Ding. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Transactions on VLSI Systems* 2(2), 1994.
[18] J. Cong, Y. Ding. Combinational logic synthesis for LUT-based field-programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems* 1(2), April 1996.
[19] J. Cong, Y. Ding. T. Gao, K. C. Chen. LUT-base, FPGA technology mapping under arbitrary net-delay model. *Computers and Graphics* 18(4), 1994.
[20] J. Cong, H. Huang, X. Yuan. Technology mapping and architecture evaluation for k/m-macrocell-based FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, January 2005.
[21] J. Cong, Y. Hwang. Simultaneous depth and area minimization in LUT-based FPGA mapping. *International Symposium on Field-Programmable Gate Arrays*, February 1995.
[22] J. Cong, Y. Hwang. Structural gate decomposition for depth-optimal technology mapping in LUT-based FPGA design. *Design Automation Conference*, 1996.
[23] J. Cong, Y. Hwang. Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20(9), 2001.
[24] J. Cong, K. Minkovich. Optimality study of logic synthesis for LUT-based FPGAs. *International Symposium on Field-Programmable Gate Arrays*, February 2006.
[25] J. Cong, K. Minkovich. Improved SAT-based Boolean matching using implicants for LUT-based FPGAs. *International Symposium on Field-Programmable Gate Arrays*, February 2007.
[26] J. Cong, S. Xu. Technology mapping for FPGAs with embedded memory blocks. *International Symposium on Field-Programmable Gate Arrays*, 1998.
[27] J. Cong, C Wu. FPGA Synthesis with retiming and pipelining for clock period minimization of sequential circuits, *Design Automation Conference*, 1997.
[28] J. Cong, C Wu. Optimal FPGA mapping, retiming with efficient initial state computation. *Design Automation Conference*, 1998.
[29] J. Cong, C. Wu, Y. Ding. Cut ranking and pruning: Enabling a general, efficient FPGA mapping solution. *International Symposium on Field-Programmable Gate Arrays*, February 1999.
[30] J. Cong, S. Xu. Delay-optimal technology mapping for FPGAs with heterogeneous LUTs. *Design Automation Conference*, 1998.
[31] J. Cong, S. Xu. Delay-oriented technology mapping for heterogeneous FPGAs with bounded resources. *International Conference on Computer-Aided Design*, 1998.

[32] J. Cong, S. Xu. Performance-driven technology mapping for heterogeneous FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19(11), November 2000.
[33] G. De Micheli. *Synthesis: Optimization of Digital Circuits*, McGraw-Hill, 1994.
[34] A. H. Farrahi, M. Sarrafzadeh. FPGA technology mapping for power minimization. *International Workshop on Field-Programmable Logic and Applications*, 1994.
[35] A. Farrahi, M. Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13(11), November 1994.
[36] R. J. Francis et al. Chortle-CRF: Fast technology mapping for lookup table–based FPGAs. *Design Automation Conference*, 1991.
[37] R. J. Francis, J. Rose, Z. Vranesic. Technology mapping for lookup table–based FPGAs for performance. *International Conference on Computer-Aided Design*, November 1991.
[38] Z. Hasan, D. Harrison, M. Ciesielski. A fast partition method for PLA-based FPGAs. *IEEE Design and Test of Computers*, December 1992.
[39] J. He, J. Rose. Technology mapping for heterogeneous FPGAs. *International Symposium on Field-Programmable Gate Arrays*, 1994.
[40] M. Inuani, J. Saul. Resynthesis in technology mapping for heterogeneous FPGAs. *International Conference on Computer-Aided Design*, 1998.
[41] A. Kaviani, S. Brown. Technology-mapping issues for an FPGA with lookup tables, PLA-like blocks. *International Symposium on Field-Programmable Gate Arrays*, 2000.
[42] J. L. Kouloheris. *Empirical Study of the Effect of Cell Granularity on FPGA Density, Performance*, Ph.D. thesis, Stanford University, 1993.
[43] M. R. Korupolu, K. K. Lee, D. F. Wong. Exact tree-based FPGA technology mapping for logic blocks with independent LUTs. *Design Automation Conference*, 1998.
[44] S. Krishnamoorthy, R. Tessier. Technology-mapping algorithms for Hybrid FPGAs containing lookup tables, PLAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22(5), May 2003.
[45] J. Lamoureux, S. J. E. Wilton. On the interaction between power-aware FPGA CAD algorithms. *IEEE International Conference on Computer-Aided Design*, November 2003.
[46] E. L. Lawler, K. N. Levitt, J. Turner. Module clustering to minimize delay in digital networks. *Transactions on Computers* 18(1), 1969.
[47] K. Lee, D. Wong. An exact tree-based, structural technology-mapping algorithm for configurable logic blocks in FPGAs. *International Conference on Computer-Aided Design*, 1999.
[48] C. Legl, B. Wurth, K. Eckl. A Boolean approach to performance-directed technology mapping for LUT-based FPGA designs. *Design Automation Conference*, June 1996.
[49] E. Lehman, Y. Watanabe, J. Grodstein, H. Harkness. Logic decomposition during technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16(8), 1997.
[50] C. E. Leiserson, J. B. Saxe. Retiming synchronous circuitry. *Algorithmica* 6, 1991.
[51] F. Li, D. Chen, L. He, J. Cong. Architecture evaluation for power-efficient FPGAs. *International Symposium on Field-Programmable Gate Arrays*, February 2003.
[52] H. Li, S. Katkoori, W. K. Mak. Power minimization algorithms for LUT-based FPGA technology mapping. *ACM Transactions on Design Automation of Electronic Systems* 9(1), January 2004.

[53] J. Lin, A. Jagannathan, J. Cong. Placement-driven technology mapping for LUT-based FPGAs. *International Symposium on Field-Programmable Gate Arrays*, February 2003.
[54] A. Ling, D. Singh, S. Brown. FPGA technology mapping: A study of optimality. *Design Automation Conference*, 2005.
[55] V. Manohararajah, S. D. Brown, Z. G. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. *International Workshop on Logic Synthesis*, 2004.
[56] A. Mathur, C. L. Liu. Performance-driven technology mapping for lookup table–based FPGAs using the general delay model. *International Workshop on Field-Programmable Gate Arrays*, February 1994.
[57] A. Mishchenko, S. Chatterjee, R. Brayton. Improvements to technology mapping for LUT-based FPGAs. *International Symposium on Field-Programmable Gate Arrays*, 2006.
[58] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference*, 2001.
[59] R. Murgai et al. Improved logic synthesis algorithms for table lookup architectures. *International Conference on Computer-Aided Design*, November 1991.
[60] R. Murgai et al. Performance directed synthesis for table lookup programmable gate arrays. *International Conference on Computer-Aided Design*, November 1991.
[61] P. Pan, C. C. Lin. A new retiming-based technology-mapping algorithm for LUT-based FPGAs. *International Symposium on Field-Programmable Gate Arrays*, 1998.
[62] P. Pan, C. L. Liu. Technology mapping of sequential circuits for LUT-based FPGAs for performance. *International Symposium on Field-Programmable Gate Arrays*, 1996.
[63] P. Pan, C. L. Liu. Optimal clock period FPGA technology mapping for sequential circuits. *Design Automation Conference*, June 1996.
[64] P. Pan, C. L. Liu. Optimal clock period FPGA technology mapping for sequential circuits. *ACM Transactions on Design Automation of Electronic Systems* 3(3), 1998.
[65] S. Safarpour, A. Veneris, G. Baeckler, R. Yuan. Efficient SAT-based Boolean matching for FPGA technology mapping. *Design Automation Conference*, July 2006.
[66] P. Sawkar, D. Thomas. Technology mapping for table lookup–based field-programmable gate arrays. *ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, February 1992.
[67] M. Schlag, J. Kong, P. K. Chan. Routability-driven technology mapping for lookup table–based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13(1), 1994.
[68] L. Shang, A. Kaviani, K. Bathala. Dynamic power consumption in Virtex-II FPGA family. *International Symposium on Field-Programmable Gate Arrays*, February 2002.
[69] N. Togawa, M. Sato, T. Ohtsuki. Maple: A simultaneous technology mapping, placement, and global routing algorithm for field-programmable gate arrays. *International Conference on Computer-Aided Design*, 1994.
[70] S. Wilton. SMAP: Heterogeneous technology mapping for area reduction in FPGAs with embedded memory arrays. *International Symposium on Field-Programmable Gate Arrays*, 1998.
[71] Z. H. Wang, E. C. Liu, J. Lai, T. C. Wang. Power minimization in LUT-based FPGA technology mapping. *Asia South Pacific Design Automation Conference*, 2001.
[72] H. Yang, D. F. Wong. Edge-map: Optimal performance-driven technology mapping for iterative LUT-based FPGA designs. *International Conference on Computer-Aided Design*, November 1994.

FPGA PLACEMENT

One thing that stands out in this book's contents: While most individual steps in the compilation flow are covered in a single chapter, placement is covered in three—Chapters 14 through 16. Placement is actually just the problem of assigning specific logic computations to individual logic blocks in the architecture, so why does it merit a longer treatment than, say, FPGA routing? There are at least two reasons.

One reason is historical: Until relatively recently, the placement problem was small enough that structured approaches were possible. These included hand placement, which produced higher-quality results than automatic placement. In contrast, for a problem such as routing, FPGA routers were very fast and efficient, and thus hand-routing was almost never done.

A second reason is that fundamentally different approaches can be taken to solve the placement problem. Do we view the design as an unstructured pile of gates to be scattered across the FPGA's surface, or is there an inherent structure that can be leveraged? And, if we use the computation's structure to drive the placement process, how do we handle portions of the computation, such as control, that likely do not have such an easily determined structure?

These considerations have given rise to several ways of performing FPGA placement, which are represented by the three chapters that follow. In Chapter 14 we consider general-purpose FPGA placement. Such systems, using complex optimization techniques, treat the designer's circuit as essentially an unstructured collection of gates. These are packed together into logic blocks and placed in the array, guided almost exclusively by the design's local connectivity information. Higher-level information, such as the design hierarchy or the regularity in multibit operations, is largely ignored. Thus, these techniques can handle any possible placement problem. Moreover, they serve as a good starting point, as other approaches that rely on more structure in the netlist generally do not work for unstructured designs, and so there must always be some way for unstructured netlists to be processed.

Chapter 15 considers datapath placement. Most designs for an FPGA consist of a large, highly structured datapath and a small, unstructured control system. The datapath is built from multibit function units, such as adders and multipliers, where the computation is fairly similar for each bit of the operands. Datapath-oriented placers can automatically leverage this information to improve the resulting placement quality.

An alternative to fully automatic placement, whether for random logic or for datapaths, is to provide ways for the user to guide the placement process. For example, the user generally knows what portions of the design should be kept

together, where the critical paths are, and how these critical paths should be laid out. Chapter 16 considers such systems, in which placement is more a user-guided process than a fully automated algorithm. Whereas the size of modern FPGA designs, and the increasing quality of placers, is making this approach less attractive over time, constructive placement of critical subsystems is still a valid alternative.

CHAPTER 14

PLACEMENT FOR GENERAL-PURPOSE FPGAS

Vaughn Betz
Altera Corporation

Placement follows technology mapping in the CAD flow and chooses a location for each block in a circuit. This chapter describes "general-purpose" placement approaches; these techniques can be used with any circuit targeting the commercial field-programmable gate arrays (FPGAs) in widespread use today. After defining the placement problem and optimization goals, the chapter describes the clustering algorithms that are frequently used in conjunction with placement tools. Three different classes of placement algorithms are then detailed: simulated annealing, partition based, and analytic. The chapter concludes with suggestions for further reading and open challenges in FPGA placement.

14.1 THE FPGA PLACEMENT PROBLEM

An FPGA placement algorithm takes two basic inputs: (1) a netlist specifying the functional blocks to be implemented and the connections between them, and (2) a device map indicating which functional unit can be placed at each location. The algorithm selects a legal location for each block such that the circuit wiring is optimized. Figure 14.1 illustrates the FPGA placement problem. Both the legality constraints and the optimization metric (what constitutes a "good" arrangement of functional blocks) depend on the FPGA architecture being targeted.

A good placement is extremely important for FPGA designs—without a high-quality placement, a circuit generally cannot be successfully routed. Even if the circuit does route, a poor placement will still lead to a lower maximum operating speed and increased power consumption. At the same time, finding a good placement for a circuit is a challenging problem. A large commercial FPGA contains approximately 500,000 functional blocks, leading to approximately 500,000! possible placements. Exhaustive evaluation of the placement solution space is therefore impossible. Furthermore, placement is a computationally hard problem, so there are no known algorithms that produce optimal results in practical central processing unit (CPU) time. Consequently, the development of fast and effective heuristic placement algorithms is a very important research area.

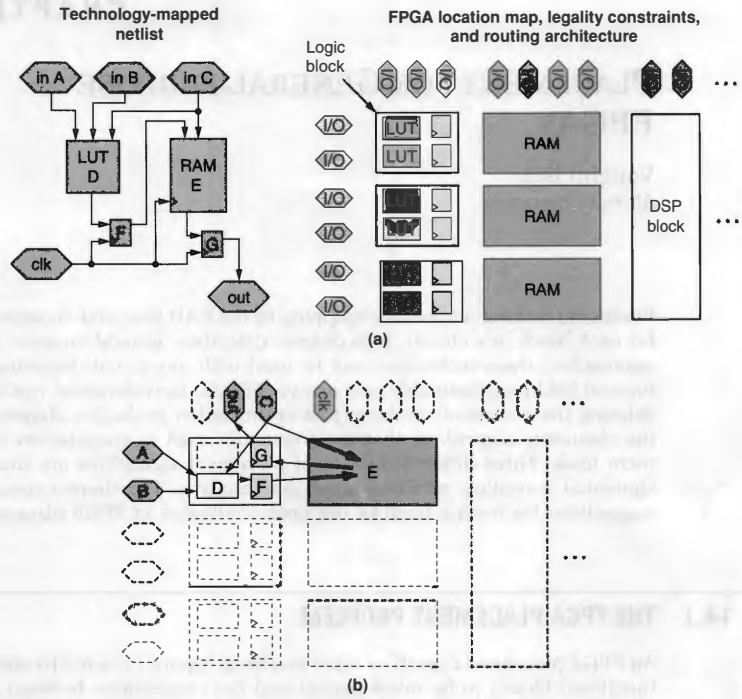


FIGURE 14.1 ■ Placement overview: (a) inputs to the placement algorithm, and (b) placement algorithm output—the location of each block.

14.1.1 Device Legality Constraints

The fact that all resources are prefabricated in an FPGA leads to a variety of placement legality constraints:

- A legal placement must place a functional block only in a location on the chip that can accommodate it. For example, a RAM block must be placed in a RAM location, and a lookup table (LUT) must be placed in a LUT location.
- Usually there are legality constraints on groups of functional blocks. In Altera's Stratix-II FPGAs, for example, a *logic block* contains 16 LUTs and 16 registers [1]. However, there are limits on the number of clock signals, clock enable signals, and routing inputs to the logic block. Consequently, not every grouping of 16 LUTs and 16 registers constitutes

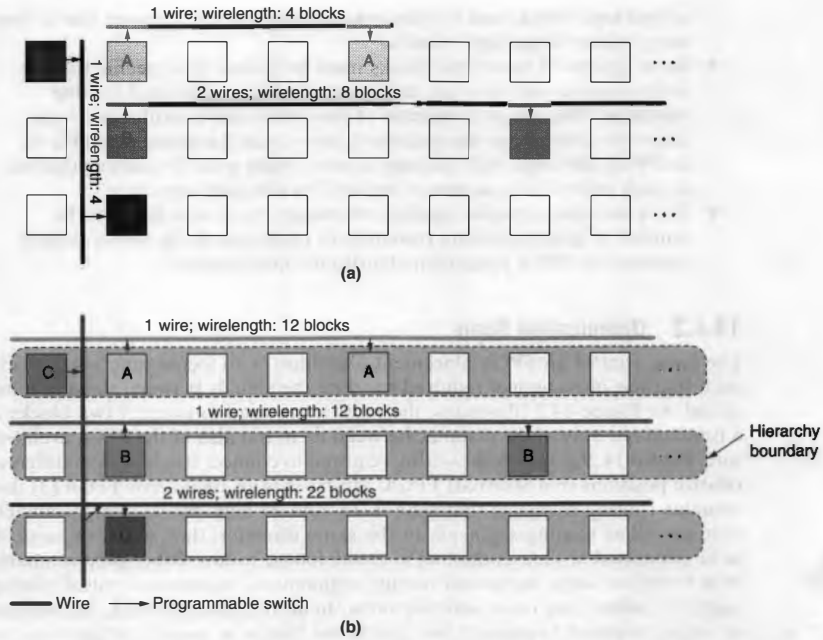


FIGURE 14.2 ■ Influence of the routing architecture on wirelength for a given placement: (a) sample routings on a Stratix-II FPGA (island style), and (b) sample routings on an APEX FPGA (hierarchical).

minimize not only the total wiring required by the design but also the amount of *routing congestion*. Routing congestion occurs when the interconnect demand approaches or exceeds the fabricated wiring capacity in some part of the FPGA.

In addition to optimizing for routability, timing-driven algorithms use timing analysis [5] to identify critical paths and/or connections and to optimize the delay of those connections. Since most delays in an FPGA are due to the programmable interconnect, timing-driven placement can achieve a large improvement in circuit speed over routability-driven approaches.

Some recent FPGA placement algorithms attempt to minimize power consumption as well.

14.1.3 Designer Placement Directives

Commercial FPGA placement tools allow designers to control the placement of some or all of the design logic at various levels of abstraction. Obeying the placement directives specified by a designer while still choosing good locations

a legal logic block, and the placement algorithm must ensure that it does not produce illegal logic blocks.

- Some groups of functional blocks must be placed in a specific relative orientation so that they can make use of special, dedicated routing resources. The simplest example of this constraint is arithmetic logic cells—in order to use the dedicated carry-chain hardware available in an FPGA, the logic cells forming a carry chain must be placed adjacent to each other in the sequence required by the carry structure.
- There are other detailed legality constraints, such as a limit on the number of global clocking resources in each area of the device, which commercial FPGA placement algorithms must respect.[1]

14.1.2 Optimization Goals

The basic goal of an FPGA placement algorithm is to locate functional blocks such that the interconnect required to route the signals between them is minimized. As Figure 14.2 illustrates, the routing required to connect two blocks is a function not only of the distance between them but also of the FPGA architecture. Figure 14.2(a) shows the wiring required to connect two blocks in different relative positions in a Stratix-II FPGA. Stratix-II is an *island-style* FPGA [3] that contains routing segments that span 4, 16, and 24 logic blocks. Programmable switches allow routing segments in the same direction (horizontal or vertical) to be connected at their endpoints to create longer routes. Other programmable switches allow some horizontal routing segments to connect to vertical routing segments where they cross and vice versa. In an island-style FPGA, the amount of wiring required to connect two functional blocks is roughly proportional to the Manhattan distance between them.

Figure 14.2(b) shows that the wiring required by the same placements in an FPGA with a *hierarchical* routing architecture (in this case the Altera APEX family [4]) is quite different. For hierarchical FPGAs, the amount of wiring required to connect two functional blocks is proportional to the number of levels of the routing hierarchy that must be traversed to connect them. Note that even the ranking of placement choices is different between APEX and Stratix-II—in Stratix-II placements, *A* and *C* are best, while in APEX placements, *A* and *B* are best. Clearly FPGA placement algorithms must have a model of the routing architecture they target in order to achieve good results.

FPGA placement tools can broadly be divided into *routability-driven* and *timing-driven* algorithms. Routability-driven algorithms try to create a placement that minimizes the total interconnect required, as this increases the probability of successfully routing the design. Since FPGA interconnect is prefabricated, the amount of interconnect in each region of a device is fixed, and a placement that requires more interconnect in a device region than that region contains cannot be routed. Consequently, some routability-driven placement algorithms

[1] Researchers wishing to target their CAD tools to industrial FPGAs can obtain a full list of the legality constraints in Altera FPGAs from the Quartus University Interface Program [2].

for the unconstrained and partially constrained blocks is a challenging problem, but one on which little has been published.

Figure 14.3 illustrates the common types of placement directives. The most restrictive specifies the *exact location* of a block. Typical uses of this directive are to lock down the design I/Os at the locations required by the circuit board or to lock down the elements of a performance-critical intellectual property (IP) core. A less restrictive directive forces blocks to go into a specific two-dimensional area, or *fixed region*. This directive allows a designer to guide the placement tool to a good high-level floorplan while still allowing automatic optimization of the placement details. One can specify the *relative location* of several blocks, but let the placement tool choose exactly where to locate the block group. This directive is useful for library components where a designer knows a good placement of the component blocks relative to each other. A *floating region* specifies that some logic should be placed within a tight region but that the placement tool can choose where that region should be on the device.

One must take care when specifying placement directives, as fixing portions of the placement ineffectively will reduce result quality versus a fully automatic placement. Modern placement tools produce high-quality results, and generally

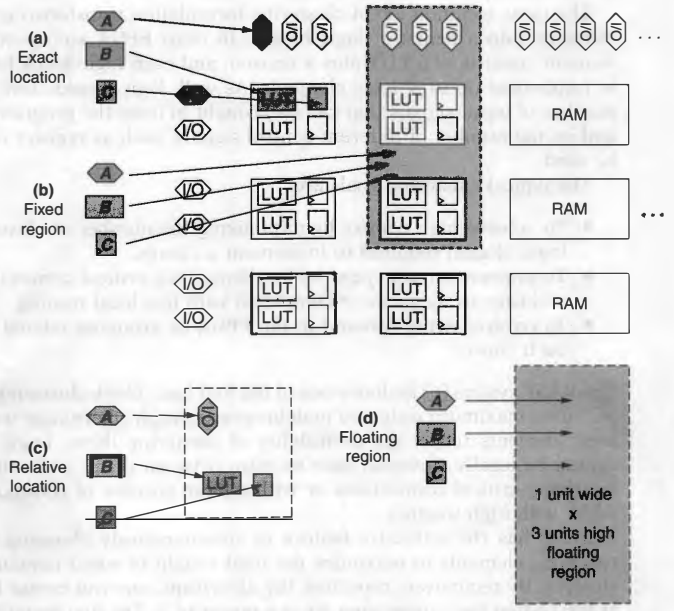


FIGURE 14.3 ■ Placement directives, ordered from most to least restrictive: (a) exact location, (b) fixed region, (c) relative location, and (d) floating region.

it is very difficult for a designer to specify placement directives on irregular logic that lead to a better solution than the placement tool would find without guidance. Placement directives have more value for regular structures, since humans are better than conventional CAD tools at recognizing regular logic patterns and matching them to a highly optimized regular placement. For examples of the use of placement directives, see Chapter 16.

14.2 CLUSTERING

A common companion to FPGA placement algorithms is a bottom-up clustering step that runs before the main placement algorithm to group-related circuit elements together into clusters. Clustering reduces the number of blocks to place, improving the runtime of the main placement algorithm. In addition, one normally chooses a cluster size that corresponds to a natural boundary in the FPGA architecture, such as a logic block. This allows the clustering algorithm to deal with many of the device legality constraints by ensuring that each cluster forms a legal logic (or RAM or DSP) block, and it simplifies legality checking for the main placement algorithm.

The most common FPGA clustering formulation transforms a netlist of logic elements into a netlist of logic blocks. In most FPGA architectures each logic element consists of a LUT plus a register, and each logic block has the capacity to implement up to N logic elements. As well, logic blocks have a limit on the number of input signals that can be brought in from the programmable routing and on the number of different control signals, such as register clocks, that can be used.

The typical clustering goals are:

- To achieve high density by minimizing the number of clusters (i.e., logic blocks) required to implement a circuit.
- To improve circuit speed by localizing time-critical connections within a cluster so they can be completed with fast local routing.
- To reduce wiring demand in the FPGA by grouping related logic in each cluster.

The RASP system [6] includes one of the first logic block clustering algorithms. It performs maximum weighted matching on a graph where edge weights between logic elements reflect the desirability of clustering them. Logic elements that cannot be legally clustered have no edge between them, while those connected by timing-critical connections or with a large number of common signals have edges with high weights.

RASP has the attractive feature of simultaneously choosing all clusters of two logic elements to maximize the total weight of edges contained within the clusters. By recursively repeating the algorithm, one can create larger clusters, at least when the cluster capacity is a power of 2. The first matching produces a netlist of size-2 clusters; a matching on the size-2 cluster netlist produces size-4

clusters, and so on. The RASP clustering algorithm has a high computational complexity of $O(n^3)$, where n is the number of logic elements in the circuit. This prevents it from scaling to large problems.

The VPack algorithm [3] takes the opposite approach to that of RASP—it creates one cluster of the desired size (e.g., seven logic elements) before moving on to create the next cluster. VPack first chooses a *seed* logic element for a new cluster and then greedily packs the logic element with the highest *attraction* to the current cluster until no more can be legally added. The attraction function is the number of nets that connect to both the logic element in question and the current cluster. VPack has a computational complexity of $O(k_{\max}n)$ where $k_{\max}$ is the maximum fanout of any net in the design, so it scales well to large problems.

Many algorithms that use the same basic procedure as VPack, but different attraction functions, have been published. The T-VPack algorithm by Marquardt et al. [3,7] is a timing-driven enhancement of VPack where the attraction function for a logic element, L, to cluster C becomes

$$Attraction(L) = 0.75 \cdot \sum_{j \in conn(L,C)} criticality(j) + 0.25 \cdot \frac{|Nets(L) \cap Nets(C)|}{MaxNets} \tag{14.1}$$

The first term in equation 14.1 gives higher attraction to logic elements that are connected to the current cluster by timing-critical connections, while the second term is taken from VPack and favors grouping together logic elements with many common signals. To find the criticality of each connection, a timing analysis is performed with a simple delay model to determine each connection's timing *slack*. The slack of a connection [5] is defined as the amount of delay that can be added to that connection before some path through it limits the circuit speed. The *criticality* of a connection, j, is then given by

$$criticality(j) = 1 - \frac{slack(j)}{D_{\max}} \tag{14.2}$$

where $D_{\max}$ is the delay of the longest path in the circuit. Connections on the critical path (i.e., with no timing slack) have a criticality of 1, while connections with a large amount of slack have a criticality near 0.

Somewhat surprisingly, T-VPack improves not only circuit speed over VPack but also reduces the amount of programmable routing required between clusters. By absorbing more connections within clusters, T-VPack is able to capture more nets entirely within a cluster, which reduces wiring demand between logic blocks.

The iRAC [8] clustering algorithm uses an attraction function that favors the absorption of small nets within a cluster:

$$Attraction(L, C) = \sum_{i \in Nets(L) \cap Nets(C)} k(i, L, C) \cdot \frac{[1 + pins\_in\_cluster(i, C)]}{|pins(i)|}$$

$$k(i, L, C) = \begin{cases} 10, \text{ if adding } L \text{ to } C \text{ would absorb net } i \text{ within } C \\ 1, \text{ otherwise} \end{cases} \tag{14.3}$$

The attraction function (equation 14.3) weights nets more heavily with a small number of terminals outside the cluster, and also gives a ten-times attraction bonus to any net that would be immediately absorbed by adding block L to the cluster. By reducing the number of nets to be routed between logic blocks, iRAC achieves an improvement in routability over T-VPack.

Lamoureaux and Wilton [9] have developed a power-aware enhancement of T-VPack. They modify equation 14.1 by adding a power minimization term that weights each connection from block L to cluster C by its *switching activity*. The switching activity of a signal is the number of times it is expected to change state per second. The power minimization term favors the absorption of nets that frequently switch logic states, resulting in lower capacitance for these nets and lower overall dynamic power.

14.3 SIMULATED ANNEALING FOR PLACEMENT

Simulated annealing is the most widely used placement algorithm for FPGAs. It mimics the annealing procedure by which strong metal alloys are created—initially blocks can move fairly freely, but as the *temperature* drops they gradually freeze into a high-quality placement [10].

Figure 14.4 shows the basic flow of simulated annealing for placement. First an initial placement is generated. This initial placement is generally of low quality, and is often created simply by assigning each block to the first legal location found. The placement is then iteratively improved by proposing and evaluating placement perturbations, or *moves*. A placement perturbation is proposed by a *move generator*, generally by moving a small number of blocks to new locations. A *cost function* is used to evaluate the impact of each proposed move.

Moves that reduce cost are always accepted, or committed to the placement, while those that increase cost are accepted with probability

$$e^{-\frac{\Delta Cost}{T}}$$

where T is the current *temperature*. This function ensures that moves that increase the cost by an amount that is small compared to the current temperature are likely to be accepted, while moves that increase the cost by an amount much larger than the current temperature are not. Accepting some moves that increase the cost helps escape local minima and produces a higher-quality final placement. At the start of the anneal, temperature is high; it gradually decreases according to the *annealing schedule*. This schedule also controls how many moves are performed between temperature updates and when the placement is considered sufficiently optimized that the anneal should end.

Two key strengths of simulated annealing that make it well suited to FPGA placement are:

1. One can enforce all the legality constraints imposed by the FPGA architecture fairly directly. The two basic techniques are to forbid the creation of illegal placements in the move generator or to add a penalty cost to illegal placements.

```
P = InitialPlacement ();
T = InitialTemperature ();

while (ExitCriterion () == False) {
    while (InnerLoopCriterion () == False) { /* One temperature */

        P_new = PerturbPlacementViaMove (P);
        ΔCost = Cost (P_new) - Cost (P);
        r = random (0,1);
        if (r < e^(-ΔCost/T)) {
            P = P_new ; /* Accept move */
        }
    }  /* End one temperature */
    T = UpdateTemp (T);
}
```

FIGURE 14.4 ■ Pseudo-code of a generic simulated annealing placement algorithm. (*Source*: Adapted from [13].)

2. By creating an appropriate cost function, one can directly model the impact of the FPGA routing architecture on circuit delay and routing congestion.

14.3.1 VPR and Related Annealing Algorithms

VPR [3,11,12] is a popular timing-driven simulated annealing placement tool. It is usually used in conjunction with T-VPack, or a similar clustering algorithm, that preclusters the logic elements into legal logic blocks. One of VPR's main features is that it can automatically adapt to different FPGA architectures so long as they employ island-style routing.

VPR's annealing schedule is based on parameters computed during placement rather than on fixed starting and ending temperatures and a fixed cooling rate. This adaptive annealing schedule generates high-quality results across a wide range of design sizes, FPGA architectures, and cost functions, making it preferable to more "hardcoded" schedules. VPR sets the *InitialTemperature* to 20 times the cost change of the average move, and the *ExitCriterion* is met when the temperature is less than 0.5 percent of the cost divided by the number of nets in the circuit. The fraction of moves that are accepted at each temperature, α, is monitored throughout the anneal.

Lam and Delosme [14] showed that simulated annealing makes the largest improvements to a placement when α is near 44 percent. Consequently, VPR rapidly decreases the temperature when α is significantly above or below 44 percent and slowly decreases it when α is near 44 percent in order to spend the majority of the annealing time in the most productive range. The move generator used by VPR to find placement perturbations also varies as the anneal progresses in order to keep α near 44 percent. When a block is picked for a move, its new proposed location will always be within a window with a Manhattan radius of *range limit* blocks. Initially, the range limit is the size of the entire chip, allowing a block to move anywhere in the device in one move.

As the anneal progresses, the range limit shrinks so that the moves proposed are smaller local improvements, since these are the most likely moves to be

accepted as the placement converges to an increasingly high-quality solution. More specifically, whenever the temperature is updated in Figure 14.4, VPR also updates the range limit according to

$$range\_limit\ (new) = range\_limit\ (old) \cdot (1 - 0.44 - \alpha) \tag{14.4}$$

VPR's cost function [12] also has some ability to adapt to different FPGA architectures:

$$\begin{aligned} Cost = (1-\lambda) \sum_{i \in All\ Nets} q(i) \left[\frac{bb_x(i)}{C_{av,x}(i)} + \frac{bb_y(i)}{C_{av,y}(i)} \right] \\ + \lambda \sum_{j \in All\ Connections} Criticality(j) \cdot Delay(j) \end{aligned} \tag{14.5}$$

The first term in equation 14.5 causes the placement algorithm to optimize an estimate of the routed wirelength, normalized to the average wiring capacity in each region of the FPGA. The wirelength needed to route each net i is estimated as the bounding box span (bb_x and bb_y) in each direction, multiplied by a fanout-based correction factor, $q(i)$. As Figure 14.5(a) illustrates, the bounding box of a net is simply the smallest rectangle that encloses all the net terminals. Figure 14.5(b) shows that for higher fanout nets, the bounding box span underpredicts the wiring needed. For the eight-terminal net shown, the sum of bb_x and bb_y is 10 units, but even a best-case routing requires 11 units of wire. $q(i)$ is 1 for two- and three-terminal nets and slowly increases with net terminal count to compensate for this underprediction [16].

The corrected bounding box span is a reasonable estimate of the routed wirelength for an island-style FPGA that contains at least some short wiring segments that span only a few logic blocks. Most recent commercial FPGAs, including the Altera Stratix and Xilinx Virtex [15] families, meet this condition. Equation 14.5 does not contain a good estimate of wirelength for other FPGA types, such as hierarchical FPGAs, so this cost function would not perform well with them.

Some FPGAs have differing amounts of routing available in the vertical direction compared to the horizontal direction, or in different regions of the chip. For example, a Stratix-II FPGA has 1.6 times as much horizontal as vertical routing, and some routing is not available over the large 576-kbit RAM blocks. Therefore, the routing capacity is not uniform everywhere in the device. In such cases, it is beneficial to move wiring demand to the more routing-rich direction or regions. Accordingly, the cost function of equation 14.5 scales the estimated wiring in each direction by the average routing capacity over the net bounding box in that direction. Figure 14.5(a) shows an example computation.

The second term in equation 14.5 optimizes timing by favoring placements in which timing-critical connections have the potential to be routed with low delay. To evaluate the second term quickly, VPR needs to be able to rapidly estimate the delay of a connection. It makes use of the fact that the delay between two points in an island-style FPGA is primarily a function of the distance between them. Before placement begins, VPR precomputes a table of best-case routing

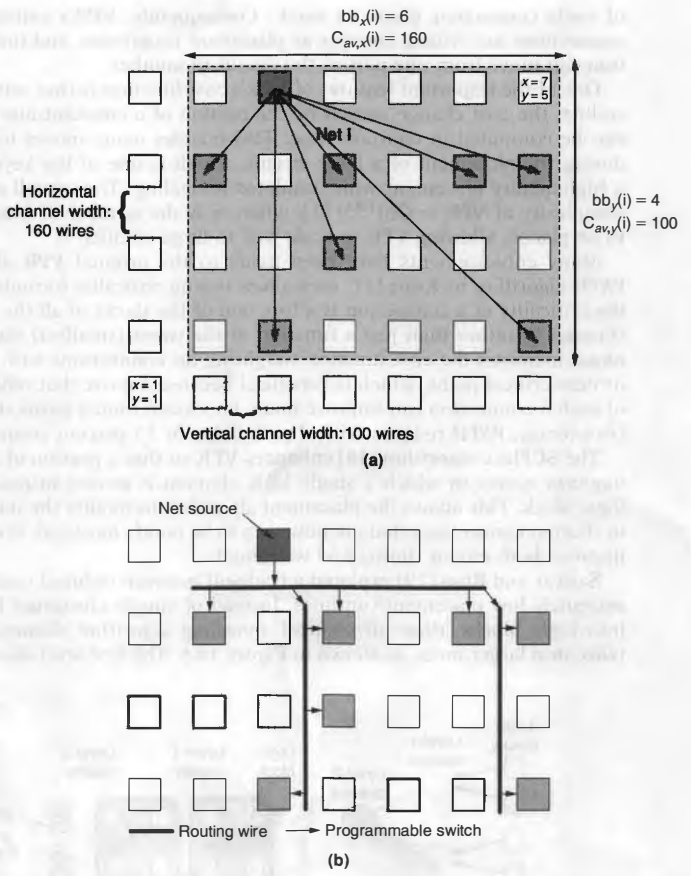


FIGURE 14.5 ■ An example wirelength cost computation: (a) net bounding box and average channel capacity; (b) best-case routing, with a wirelength of 11.

delays for every possible distance between pairs of points. The delay table entries are computed by invoking a router with each possible $(\Delta x, \Delta x)$—the router finds the fastest path between the two endpoints.

Periodically (generally once per temperature) VPR computes the delay of every connection given the current placement and then performs a timing analysis to find each connection's slack. Equation 14.2 computes the criticality

of each connection given its slack. Consequently, VPR's estimate of which connections are critical changes as placement progresses, and timing optimization can move from one part of the circuit to another.

One of the important features of VPR's cost function is that, with appropriate coding, the cost change caused by the motion of a constant number of blocks can be computed in constant time. This enables many moves to be evaluated during the placement of a large circuit, which is one of the keys to obtaining a high-quality placement with simulated annealing. The overall computational complexity of VPR is $O(n^{1.33})$ [3], where n is the number of functional blocks to be placed, allowing VPR to scale well to large circuits.

Many enhancements have been made to the original VPR algorithm. The PATH algorithm by Kong [17] uses a new timing criticality formulation in which the criticality of a connection is a function of the slacks of all the paths passing through it, rather than just a function of the worst (smallest) slack. This technique increases the cost function weighting on connections with many critical or near-critical paths, which is beneficial because a move that reduces the delay of such a connection can improve many important timing paths simultaneously. On average, PATH reduces critical path delay by 15 percent compared to VPR.

The SCPlace algorithm [18] enhances VPR so that a portion of the moves are *fragment moves* in which a single logic element is moved instead of an entire logic block. This allows the placement algorithm to modify the initial clustering to shorten connections that are now seen to be poorly localized. Fragment moves improve both circuit timing and wirelength.

Sankar and Rose [19] explored a trade-off between reduced result quality and extremely low placement runtimes. Instead of simply clustering logic elements into logic blocks, their *hierarchical annealing* algorithm clusters logic blocks twice into larger units, as shown in Figure 14.6. The first-level clustering creates

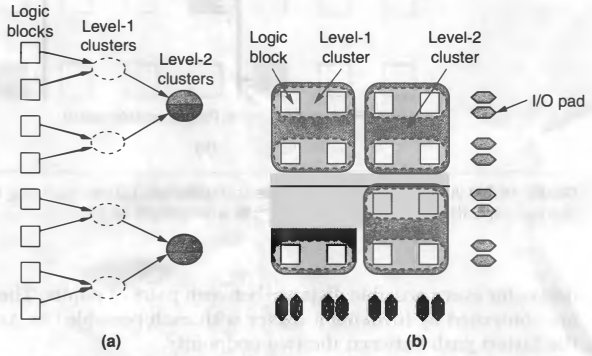


FIGURE 14.6 ■ An overview of hierarchical annealing: (a) multilevel clustering, and (b) placement of large clusters followed by unclustering and placement refinement.

clusters that each contain approximately 64 logic blocks, and the second-level clustering groups four level-1 clusters into each level-2 cluster. Placement of a netlist of level-2 clusters is very fast because there are relatively few blocks to place. To make placement of the level-2 clusters even faster, Sankar and Rose [19] use a greedy (temperature = 0 anneal) iterative improvement algorithm, seeded with a fast constructive (instead of random) placement. Once placement of the level-2 clusters is complete, a level-1 initial placement is created by locating each level-1 cluster inside the boundary of the level-2 cluster that contained it.

The placement of level-1 clusters is refined by a temperature-0 anneal. The clusters are then replaced by their constituent logic blocks and the placement of each logic block is fine-tuned with a *low-temperature* anneal. The initial temperature for this anneal is selected so that only moves that reduce cost or increase it a small amount are allowed; consequently, the initial placement solution has a large impact on the final placement. For very fast CPU times this algorithm significantly outperforms VPR in achieved wirelength, but it lags behind VPR for longer permissible CPU times.

Lamoureaux and Wilton [9] modified VPR's cost function by adding a third term, *PowerCost*, to equation 14.5.

$$PowerCost = \sum_{i \in AllNets} q(i)\left[bb_x(i) + bb_y(i)\right] \cdot Activity(i) \qquad (14.6)$$

where *Activity(i)* is the average number of times net i transitions per second. This additional cost term reduces circuit power by focusing more effort on localizing rapidly transitioning nets.

14.3.2 Simultaneous Placement and Routing with Annealing

Instead of relying on fast heuristics to estimate placement routability and timing, some algorithms use a router to obtain a partial or complete routing for each placement proposed during the anneal. These algorithms can directly extract wiring usage, congestion, and timing from the circuit routing, so their cost functions can be very detailed. Another of their advantages is that one can develop a placement algorithm that automatically adapts to a wider class of FPGA architectures, since fewer (or ideally no) assumptions about the device-routing architecture need to be incorporated into the cost function. The disadvantage of using a router in the cost function is CPU time. Evaluating the cost change after each move is very CPU intensive, making it difficult to evaluate enough moves to obtain high-quality placements for large circuits in a reasonable time.

PROXI [20] is a timing-driven FPGA placement algorithm that uses a router to compute its cost function. The PROXI cost function is a weighted sum of the number of unrouted nets and the delay of the circuit critical path. After each placement perturbation, PROXI rips up all of the nets connected to blocks that have moved and reroutes them via a fast, directed-search maze router [21].

To improve CPU time, PROXI allows the maze router to explore only a small portion of the routing fabric at high temperatures—if no unblocked routing path is found quickly, the net is left unrouted. At lower temperatures, the placement is of higher quality and the router is allowed to explore a larger portion of the routing fabric. After each net is rerouted, the critical path is recomputed incrementally. PROXI produces high-quality results, but requires high CPU time.

Independence [22] is an FPGA placement tool that can effectively target a wide variety of FPGA routing architectures. It is purely routability-driven, and its cost function monitors both the amount of wiring used by the placement and the routing congestion:

$$Cost = \sum_{i \in Nets} RoutingResources(i) + \lambda \sum_{k \in RoutingResources} \max(Occupancy(k) - Capacity(k), 0) \tag{14.7}$$

The λ parameter in equation 14.7 is a heuristic weighting factor. Independence uses the PathFinder routing algorithm [23] to find new routes for all affected nets after each move. Instead of leaving nets unrouted when there is no unblocked path, PathFinder allows *wire congestion* by routing two nets on the same routing resource. Such a routing is not legal; however, by summing the overuse of all the routing resources in the FPGA, Independence can directly monitor the amount of routing congestion implicit in the current placement. The Independence cost function monitors not only routing congestion but also the total wirelength used by the router to create a smoother cost function that is easier for the annealer to optimize. Independence produces high-quality results on a wide variety of FPGA architectures, including both island style and hierarchical, but it requires very high CPU time.

14.4 PARTITION-BASED PLACEMENT

Another popular placement approach recursively partitions the circuit netlist and assigns each partition to a different physical region in the FPGA. Usually each partitioning step divides a previous (larger) partition into two pieces, or *bipartitions* the component, although some algorithms perform *multiway partitioning* to produce a larger number of circuit partitions in each step. Partitioning algorithms attempt to minimize the number of nets that are cut, or that cross, between partitions. Since each partition of the circuit will be assigned to a different region of the FPGA, partition-based placement minimizes the number of nets leaving each region and hence indirectly optimizes the amount of wiring required by the design. Partition-based placement can leverage the availability of high-quality, CPU-efficient partitioning algorithms, making this approach scalable to large problems. However, for some FPGA architectures, partition-based placement suffers from the disadvantage that it does not directly optimize the circuit timing or the amount of routing required by the placement.

Hierarchical FPGAs are good candidates for partition-based placement, since their routing architectures create natural partitioning cut lines. Hutton et al. [24] describe a commercial placement algorithm for the Altera Apex 20K family that recursively partitions the circuit along the cut lines formed by the routing hierarchy, as shown in Figure 14.7. This algorithm is made timing-driven by heavily weighting connections with low slack during each partitioning phase and by partitioning to minimize weighted cut size. This encourages partitioning solutions in which timing-critical connections can be routed using the fast routing available within the lower levels of the routing hierarchy. To improve the prediction of the critical path, the delay estimate for each connection is a function of (1) the number of hierarchy boundaries the net must traverse because of the known partition cuts at the higher levels of the routing hierarchy, and (2) statistical estimates of how many hierarchy boundaries the connection will cross at future partitioning steps.

Recursive partitioning has also been used for placement in island-style FPGAs. ALTOR [25] was originally developed for standard cell circuits, but was adapted to FPGAs and widely used in FPGA research. Figure 14.8 shows the sequence of cut lines used by ALTOR to target an island-style FPGA—note that the sequence is quite different from that used with a hierarchical FPGA. In an island-style FPGA, blocks separated by a short Manhattan distance can be connected with a small amount of routing. Consequently, the cut lines are designed to divide the FPGA into ever-shrinking squares—the fewer signals that must leave each square, the less interconnect required.

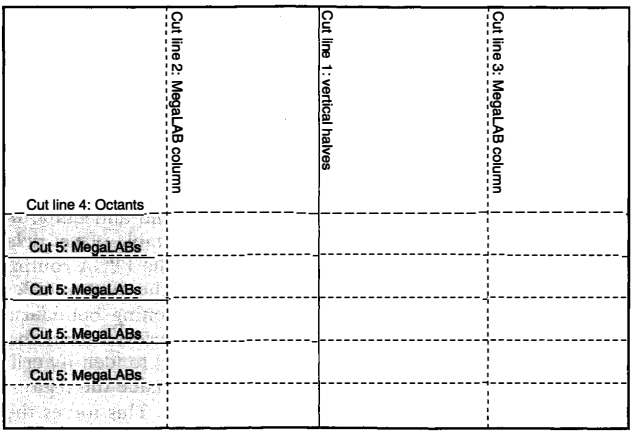


FIGURE 14.7 ■ The partitioning sequence for the APEX 20K FPGA.

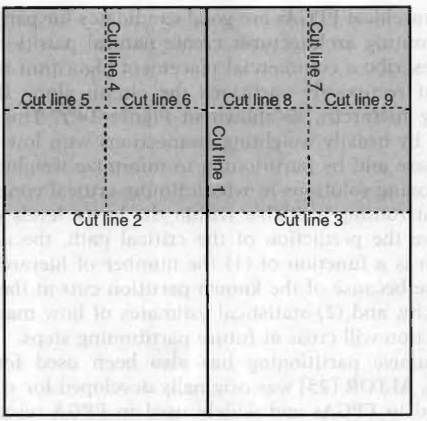


FIGURE 14.8 ■ The partitioning sequence for an island-style FPGA.

ALTOR's first cut line divides the chip into two halves vertically. The second cut line divides the left half of the circuit into upper left and lower left quarters. The third cut line divides the right half of the circuit in the same way. When partitioning along the third cut line, ALTOR uses *terminal propagation* [26] from the left half of the chip, which is already partitioned into an upper and lower quarter, to bias the partitioning of the right half. For example, the net shown in Figure 14.9 has one terminal in the right half of the chip and one terminal in the upper left corner. During partitioning along cut line 3, this net is considered to have a fixed terminal in the upper partition, which will bias the partitioner to keep the free terminal of this net in the partition above cut line 3. Terminal propagation reduces final wirelength by optimizing the placement of the terminals of nets that have been cut in some partitioning step.

Maidee et al. [27] developed a timing-driven placement algorithm for island-style FPGAs that employs both partitioning and annealing. Before partitioning begins, the VPR router is used to generate a table of net delay versus distance spanned by the net that takes into account the FPGA routing architecture. As partitioning proceeds, the algorithm records the minimum length each net can achieve given the current number of partitioning boundaries it crosses. The delay corresponding to each net's span is retrieved from the net delay versus span table, and a timing analysis is performed to identify critical connections.

Timing-critical connections to terminals outside the region being partitioned act as anchor points during each partitioning. This forces the other end of the connection to be allocated to the partition that allows the critical connection to be short. Once partitioning has proceeded to the point that each region contains only a few cells, any overfilled regions are legalized with a greedy movement

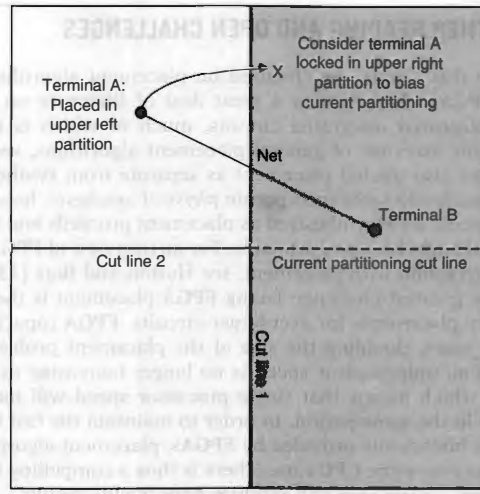


FIGURE 14.9 ■ An example of terminal propagation.

heuristic. Finally, the VPR annealing algorithm is invoked with a low starting temperature to "fine-tune" the placement. This fine-tuning step allows blocks to move anywhere in the device, so early placement decisions made by the partitioner, when little information about the critical paths or the final wirelength of each net was available, can be reversed. This algorithm achieves wirelength and speed results comparable to those of a full VPR anneal, with significantly reduced CPU time.

14.5 ANALYTIC PLACEMENT

Analytic algorithms are based on creating a smooth function of a placement that approximates routed wirelength. Efficient numerical techniques are used to find the global minimum of this function; if the function approximates wirelength well, this solution is a placement with good wirelength. However, this global minimum is usually an illegal placement, so constraints and heuristics must be applied to guide the algorithm to a legal solution.

While analytic placement approaches are popular for ASICs, few exist for FPGAs, likely due to the more difficult FPGA placement legality constraints. The Negotiated Analytic Placement (NAP) algorithm from Chan and Schlag [28] targets FPGAs and has several novel features, including some that make it suitable for implementation on multiple processors in parallel.

14.6 FURTHER READING AND OPEN CHALLENGES

While this chapter has focused on placement algorithms specifically designed for FPGAs, there is also a great deal of literature on placement for custom-manufactured integrated circuits, much of which is relevant to FPGAs. For a recent overview of general placement algorithms, see Cong et al. [29]. This chapter also treated placement as separate from synthesis. Recent commercial and academic tools incorporate *physical synthesis*, however, where portions of the circuit are resynthesized as placement proceeds and more information about critical paths becomes available. For an overview of FPGA physical synthesis and its interaction with placement, see Hutton and Betz [13].

The greatest challenge facing FPGA placement is the need to produce high-quality placements for ever-larger circuits. FPGA capacity doubles every two to three years, doubling the size of the placement problem at the same rate. In addition, uniprocessor speed is no longer increasing as quickly as it did in the past, which means that single processor speed will increase by less than two times in the same period. In order to maintain the fast time to market and ease of use historically provided by FPGAs, placement algorithms cannot be allowed to take ever more CPU time. There is thus a compelling need for algorithms that are very scalable yet still produce high-quality results.

The roadmap for future microprocessors indicates that the number of independent processors, or cores, on a single chip will increase rapidly in the coming years. Consequently, most engineers will have parallel computers on their desktops. Part of the solution to the problem of keeping FPGA placement times reasonable may be to find techniques and algorithms to exploit parallel processing without sacrificing result quality.

References

[1] D. Lewis, E. Ahmed, G. Baeckler. The Stratix-II routing and logic architecture. *Proceedings of the 13th ACM International Symposium on Field-Programmable Gate Arrays*, 2005.

[2] The Quartus University Interface Program (*www.altera.com/education/univ/research/unv-quip.html*).

[3] V. Betz., J. Rose, A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer, February 1999.

[4] R. Cliff, et al. A next generation architecture optimized for high density system level integration. *Proceedings of the 21st IEEE Custom Integrated Circuits Conference*, 1999.

[5] R. Hitchcock, G. Smith, D. Cheng. Timing analysis of computer hardware. *IBM Journal of Research and Development*, January 1983.

[6] J. Cong, J. Peck, Y. Ding. RASP: A general logic synthesis system for SRAM-based FPGAs. *Proceedings of the Fifth International Symposium on Field-Programmable Gate Arrays*, 1996.

[7] A. Marquardt, V. Betz, J. Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. *Proceedings of the Seventh International Symposium on Field-Programmable Gate Arrays*, 1999.

[8] A. Singh, M. Marek-Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2002.
[9] J. Lamoureaux, S. Wilton. On the interaction between power-aware FPGA CAD algorithms. *Proceedings of the International Symposium on Computer-Aided Design*, 2003.
[10] S. Kirkpatrick, C. Gelatt, M. Vecchi. Optimization by simulated annealing. *Science* 2(20), May 1983.
[11] V. Betz, J. Rose. VPR: A new packing, placement and routing tool for FPGA research. *Proceedings of the Seventh International Conference on Field-Programmable Logic and Applications*, 1997.
[12] A. Marquardt, V. Betz, J. Rose. Timing-driven placement for FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2000.
[13] M. Hutton, V. Betz. *Electronic Design Automation for Integrated Circuits Handbook*, Taylor and Francis, eds. (Chapter 13), CRC Press, 2006.
[14] J. Lam, J. Delosme. Performance of a new annealing schedule. *Design Automation Conference*, 1988.
[15] *Virtex Family Datasheet* (*www.xilinx.com*).
[16] C. Cheng. RISA: Accurate and efficient placement routability modeling. *Proceedings of the International Conference on Computer-Aided Design*, 1994.
[17] T. Kong. A novel net weighting algorithm for timing-driven placement. *Proceedings of the International Conference on Computer-Aided Design*, 2002.
[18] G. Chen, J. Cong. Simultaneous timing driven clustering and placement for FPGAs. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2004.
[19] Y. Sankar, J. Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1999.
[20] S. K. Nag, R. A. Rutenbar. Performance-driven simultaneous placement and routing for FPGAs. *IEEE Transactions on Computer-Aided Design*, June 1998.
[21] Y. C. Lee. An algorithm for path connections and applications. *IRE Transactions on Electronic Computing*, September 1961.
[22] A. Sharma, C. Ebeling, S. Hauck. Architecture-adaptive routability-driven placement for FPGAs. *Proceedings of the International Symposium on Field-Programmable Logic and Applications*, 2005.
[23] L. McMurchie, C. Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs. *Proceedings of the Fifth International Symposium on Field-Programmable Gate Arrays*, 1995.
[24] M. Hutton, K. Adibsamii, A. Leaver. Adaptive delay estimation for partitioning-driven PLD placement. *IEEE Transactions on VLSI* 11(1), February 2003.
[25] J. Rose, W. Snelgrove, Z. Vranesic. ALTOR: An automatic standard cell layout program. *Proceedings of the Canadian Conference on VLSI*, January 1985.
[26] A. Dunlop, B. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, January 1985.
[27] M. Maidee, C. Ababei, K. Bazargan. Fast timing-driven partitioning-based placement for island style field-programmable gate arrays. *Design Automation Conference*, 2003.
[28] P. Chan, M. Schlag. Parallel placement for field-programmable gate arrays. *Proceedings of the 11th International Symposium on Field-Programmable Gate Arrays*, 2003.
[29] J. Cong, J. Shinnerl, M. Xie, T. Kong, X. Yuan. Large-scale circuit placement. *ACM Transactions on Design Automation of Electronic Systems*, April 2005.

CHAPTER 15

DATAPATH COMPOSITION

Andreas Koch
Department of Computer Science
Embedded Systems and Applications Group
Technische Universität of Darmstadt, Germany

As shown in Chapter 14, a wide variety of algorithms can be employed for placing arbitrary netlists on various reconfigurable fabrics. To achieve this generality, the input netlists are treated as random collections of primitive elements (gates, lookup tables [LUTs], flip-flops) and interconnections. These approaches do not attempt to exploit any kind of structure that might be present in their input circuits. Many practically relevant circuits, however, do exhibit regularities in their composition (e.g., by following a classical bit-sliced design). Since the days of manual full-custom ASIC design ("polygon pushing"), regularity in circuit *structure* has been exploited with great success to derive a corresponding regular circuit *layout*—for example, by abutment of replicated bit-slice layouts.

This chapter describes the application of this idea to efficient layout of regular bit-sliced datapaths on reconfigurable fabrics. It will begin by considering how to characterize, extract, and preserve regularities at different abstraction levels. The next steps describe the datapath composition tool flow and address issues such as mapping dataflow operators to hardware units and arranging these in an abutting regular layout. We will also cover how quality can be improved even further by judiciously dissolving regularity boundaries in parts of the datapath performing cross-boundary optimization, and finally reregularizing the optimized circuit.

15.1 FUNDAMENTALS

With the increasing use of reconfigurable devices as core processing units in adaptive computer systems, the architecture and implementation of high-performance compute units on reconfigurable fabrics becomes ever more important. A *datapath* is one architectural style of realizing a given computation (Figure 15.1(a)) in hardware. It is often described as the number of interconnected *operators* in the form of a dataflow graph (DFG) or control dataflow graph (CDFG), shown in Figure 15.1(b). The execution of the operators is orchestrated by a supervising *controller* (Figure 15.1(c)). The controller is generally not considered part of the datapath, but together the datapath and controller form a *compute unit*. For purposes of this discussion, we will assume that we are processing a CDFG but will concentrate on its dataflow part.

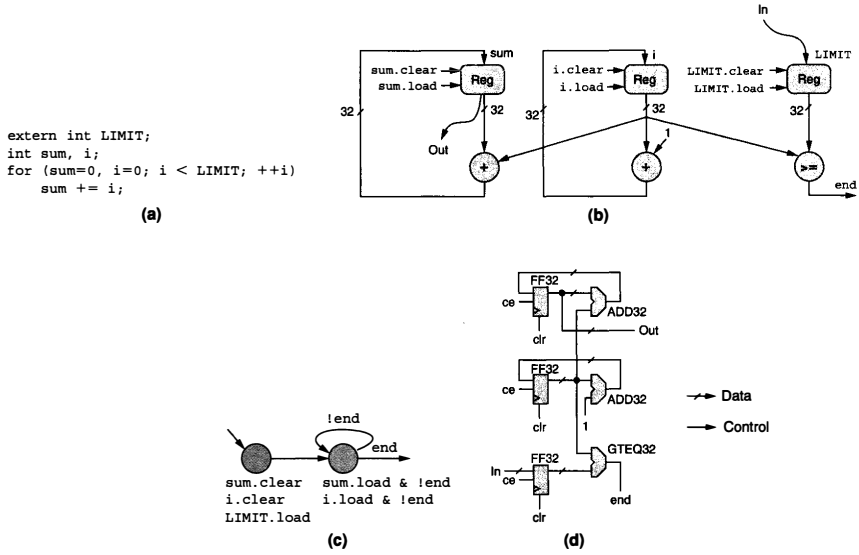


FIGURE 15.1 ■ From computation to realization.

The datapath is created in hardware by mapping the CDFG operators to *hardware operators*, or HWOPs (see Figure 15.1(d)). Generally, HWOPs have multibit *data* inputs and outputs for the operand(s) and result(s) (e.g., ADD32 HWOPs). Some may also have *control* inputs (e.g., the load and clear signals of the FF32 HWOPs) or outputs (e.g., for indicating certain conditions such as the GTEQ32 output). These control signals are generally much narrower than bused data signals, often only a single bit wide. In some cases, an HWOP is available in several different *implementations*, all having the same function but differing, for example, in their area/speed characteristics or layout shape.

15.1.1 Regularity

The multibit-wide HWOPs are often assembled by repeatedly instantiating and interconnecting narrower template circuits in an adjacent fashion until the specific HWOP's desired bit width is reached (Figure 15.2). These template circuits will be called *master slices* here, while their instances are generally referred to as *bit slices*. We will further extend this terminology to call areas where the same master slice has been instantiated a number of times a *zone*, and a sequence of zones is termed a *stack*. Together, these concepts describe an HWOP as a *regular* circuit.

Such a structure has a natural direction of *dataflow* (horizontally in the case of Figure 15.2). When processing word-wide data, the individual bits of the

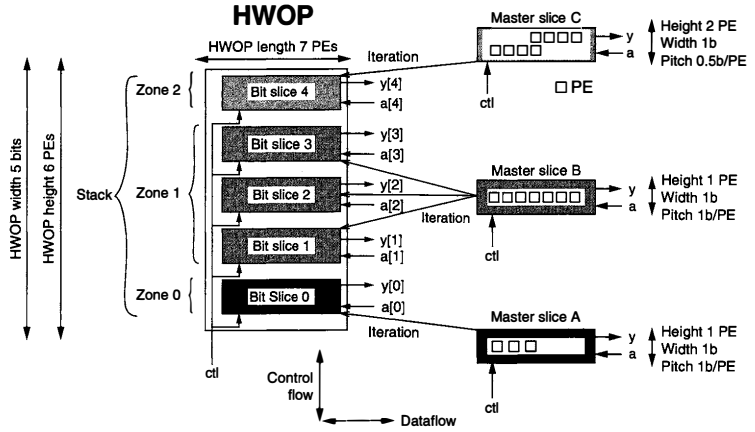


FIGURE 15.2 ■ Regular HWOP structure.

words are arranged orthogonally to the direction of dataflow (in the figure, vertically). With few exceptions (e.g., bus-wide logic gates), the position of individual bits is not arbitrary but follows an ordering from least significant (LSB) to most significant (MSB). For example, stacking ripple-carry full-adder bit slices generally has the first slice process the LSB and the last slice process the MSB. Ports on the master slice (e.g., a, y) do not have a bit significance of their own. Only after instantiating the masters as bit slices can the significance be derived from their iteration number (e.g., port a on the bottommost slice will have a significance of 0; the one above that, 1, etc.).

For describing the characteristics of elements such as HWOPs, bit slices, and master slices, four quantities are useful. Any of these elements may process multiple bits from a single word, with the logical *width* being the largest number of such bits. *Height* and *length* refer to the bounding box of the element layout on the target device. They are specified in device-dependent units, such as processing elements (PEs), cells, configurable logic blocks (CLBs), and the like. The *pitch* of a master slice is the width divided by the height—essentially, the number of output bits per unit height. To reduce interconnect lengths, all HWOPs in the datapath should have the same pitch and the LSBs of all data nets should be vertically aligned.

Regularity in datapaths does not appear just in the replicated logic elements but also in commonly occurring interconnect patterns (Figure 15.3):

Data nets are generally multibit buses that carry operands and results between HWOPs, where they are connected to data ports (e.g., `op1`, `op2`). Each signal in the bus has an associated bit significance and generally connects to the HWOP at a data port with the same significance. Shifts and permutations occur only rarely [23].

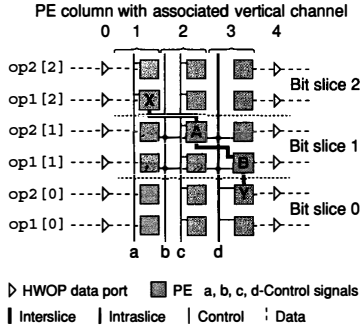


FIGURE 15.3 ■ Regular interconnection patterns.

Control nets are generally narrower, often only a single bit wide. In general, they connect an HWOP to a controller but not to another HWOP in the datapath. Control signals attach to the HWOP at control ports. In many cases, a control signal connects to the same control port in all bit slices of a zone. With our assumption of horizontal dataflow, in the following discussion control signals are assumed to run vertically.

Interslice nets run between separate bit slices in the same HWOP, thus vertically crossing slice boundaries (e.g., B–Y, A–X). Most commonly, they connect neighboring bit slices, but these may have different master slices, particularly near the top and bottom of a stack. An example of an interslice net is the carry net running between full-adder bit slices.

Intraslice nets connect individual logic elements within a bit slice (e.g., A–B). Since the internals of a bit slice are considered random logic, these nets do not follow specific interconnection patterns.

An example of a unified representation for both block and interconnect regularity, the Abstract Physical Model (APM), is proposed by Ye and De Micheli [22].

15.1.2 Datapath Layout

With these concepts in place, we can now consider the anatomy of our compute unit in greater detail (Figure 15.4(a)). The datapath will have a *regular* area, where pitch-matched HWOPs with a common direction of increasing bit significance process horizontal, LSB-aligned dataflows. Outside this area, HWOPs may contain *irregular* parts (e.g., carry initialization, overflow detection, or, for complex sequential HWOPs, even local controllers). The global controller for the compute unit is also placed outside the regular area. Generally, control nets are routed vertically across the regular area. This chapter does not address the handling of the controller, but concentrates on the datapath

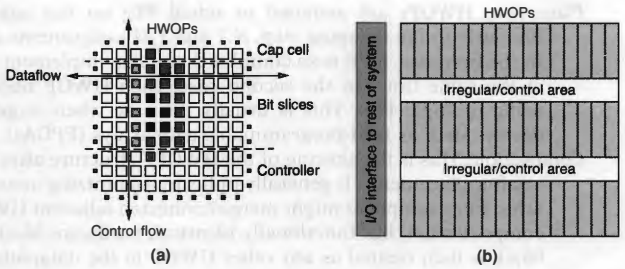


FIGURE 15.4 ■ Common datapath layouts: (a) classical linear and (b) multistripe.

instead. The controller can be placed via techniques such as those presented in Chapter 14.

Given these constraints, the best arrangement for minimizing interconnect lengths and delays for a small number of HWOPs will generally be linear. This approach has been exploited by devices like Garp [4], which realize such topologies directly in their chip architecture. However, once the number of HWOPs grows, the datapath generally needs to be wrapped into multiple stripes of HWOPs (Figure 15.4(b)).

15.2 TOOL FLOW OVERVIEW

Multiple steps are required to actually compose the datapath from individual HWOPs. These steps can be broadly grouped into the following categories:

Module generation: The HWOPs are often realized by procedural descriptions in the form of module generators (see Section 15.4). Thus, at some point in the flow other tools will interact with the library of module generators either to retrieve data *about* appropriately parametrized module instances or (later in the process) to generate the actual netlists. Often these netlists are already annotated with module-local relative placement information.

Mapping: The operators in the computation are mapped from the CDFG to the HWOPs realizing them in hardware. Beyond a straight 1:1 mapping, this can be performed in 1:*M* (if an operator requires multiple HWOPs) or *N*:1 fashion (if multiple operators can be combined into the same HWOP). The mapping calculated here need not be final, but can be altered in later flow steps. In some cases, the mapping step can also choose among multiple different HWOP implementations for an operator. This is sometimes called the *module selection* step.

Placement: HWOPs are assigned to actual PEs on the target device fabric. Similarly to the mapping step, *N*:1 and 1:*M* assignments are possible here. In the first case, a PE is so complex that it can implement multiple HWOPs at the same time. In the second case, each HWOP needs to be realized using multiple PEs. This is usually the case when targeting fine-grained devices such as field-programmable gate arrays (FPGAs).

Compaction: This is the altering of the HWOPs' structure after mapping (before or after placement). It generally indicates optimizing across HWOP boundaries. For example, it might merge connected adjacent HWOPs into a more compact/faster, but functionally identical, hardware block. This optimized block is then treated as any other HWOP in the datapath.

Not all of the flows discussed next perform all of these steps, and their execution order can vary. Additionally, some steps may be repeated.

Certain combinations are also possible. For example, in some flows placement and the mapping of operators to HWOPs occur simultaneously. For coarse-grained targets, operators can be mapped to HWOPs that are placeable in the same PE. For fine-grained devices, HWOP implementations can be selected whose layouts fit together with minimal area.

15.3 THE IMPACT OF DEVICE ARCHITECTURE

The tool flow required for creating a datapath on a reconfigurable fabric of PEs is highly dependent on the target device architecture. For coarse-grained target devices, the operators of the computation can often be mapped to PEs in a one-to-one fashion. On a fine-grained device, the operators have to be assembled from individual PEs.

Bit-sliced is not the only way to realize HWOPs. They may as well be completely irregular internally, or they may be monolithic (Figure 15.5). In both cases, many of the optimizations described in Section 15.7 that affect the internal structure of HWOPs are not be applicable. However, the techniques for processing multiple HWOPs at the datapath level (Section 15.6) remain relevant.

If the reconfigurable fabric has a linear or a two-dimensional matrix structure (Figure 15.6(a–c)), this can be exploited to efficiently map the regular

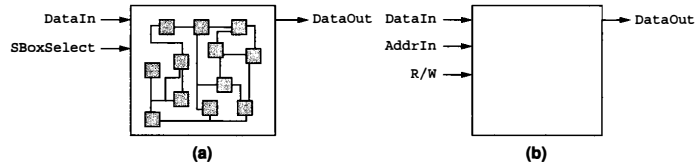


FIGURE 15.5 ■ Non-bit-sliced HWOPs: (a) irregular and (b) monolithic.

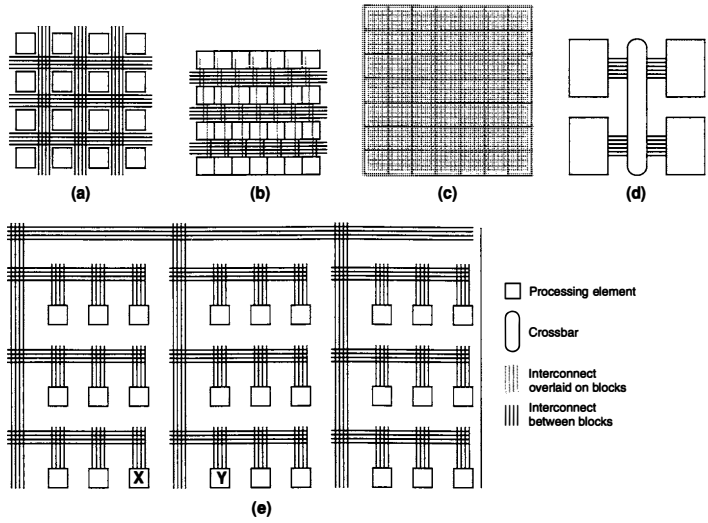


FIGURE 15.6 ■ Reconfigurable fabric architectures: (a) symmetrical array, (b) row-based, (c) sea-of-gates, (d) hierarchical PLD, and (e) hierarchical FPGA.

datapath structure to a corresponding regular geometric layout. For other kinds of target devices—for example, those having fully hierarchical structures (d–e in Figure 15.6)—algorithms optimizing for geometric arrangement are unsuitable, because geometrically adjacent blocks on the device might not actually be neighbors in the interconnect network (Figure 15.6(e), PEs `X` and `Y`). While other techniques such as hierarchical partitioning and clustering [19] could be used instead, they no longer attempt to take advantage of the datapath regularity.

15.3.1 Architecture Irregularities

Even in seemingly regular fabrics, irregularities often occur at the detail level. Consider, for example, the logic block structure of the Xilinx XC4000 FPGA (Figure 15.7). The base architecture of this device is a symmetrical array of CLBs, each of which contains two 4-LUTs and registers. However, each CLB also provides an additional 3-LUT. While very useful (e.g., for the efficient implementation of 4-input multiplexers or 5-input functions within a single CLB), the 3-LUT impedes the regularity in that it is no longer possible to realize two instances of a master slice that uses the 3-LUT within a single CLB. Also, when using the 3-LUT it is no longer possible to employ the registers in the CLB independently from the 4-LUTs: Only one of the registers can be directly connected to a CLB external port (`DIN`); the other one is not reachable from the outside.

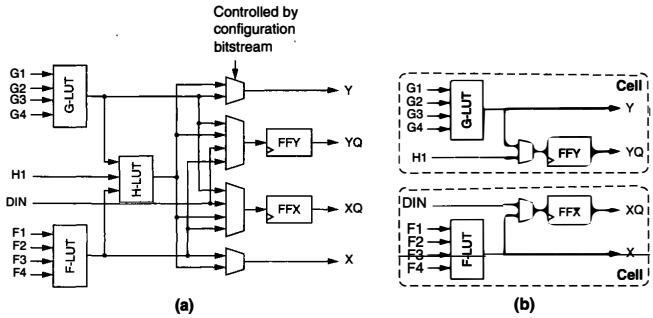


FIGURE 15.7 ■ Regularizing an existing device architecture: (a) the real structure of the Xilinx XC4000 CLB and (b) the simplified regular structure.

These irregularities can be alleviated by disregarding the 3-LUT for regular logic, using it solely to make the other register accessible via the `H1` port. As a result, each CLB can now be used to implement two fully regular bit slices, with the registers accessible both from inside and outside the bit slice.

Interconnect features also have an effect on datapath placement style. The physical direction of bit significances on the fabric is sometimes dictated by the running order of fast carry wires, which, on most devices is fixed. Also, high fanout control signals (e.g., the select signal of wide multiplexers) can be distributed across an entire HWOP by special long-distance interconnects. For example, on the Xilinx Virtex series of chips, so-called vertical long lines connect to all PEs on both sides of a vertical routing channel and are thus ideally suited for control routing. As will be shown in the following section, tool flows for datapaths can take advantage of all these features for efficient layout.

15.4 THE INTERFACE TO MODULE GENERATORS

As in many hardware design flows, individual hardware cells (in our case, the circuits used as HWOPs), are retrieved from a library. Instead of static cells, however, a more flexible approach uses procedural module generators to tailor these circuits to fit current requirements. For example, a multiplier might have eight pipeline stages in one context and only four in another, matching it to the latency/clock speed of the rest of the datapath. No longer a passive collection of cell descriptions, the library now becomes *active*: It accepts a set of constraints from another part of the flow and delivers a matching circuit.

The very flexibility of these parametrized generators complicates their integration with the rest of the tool flow: Other tools need not only the circuit description in the form of a (possibly preplaced) netlist but also data *about* this specific instance. Different tools are interested in different aspects of the

circuit. This plethora of cell *views*, combined with the sheer volume of the design space covered by each parametrized generator, precludes a simple enumeration of all alternatives. Thus, the traditional static library data files, holding tables of delays, bounding boxes, and the like, for a set of fixed parameter values, become impractical.

The Flexible API for Module-based Environments (FLAME) [11] is one approach to overcoming these difficulties. It consists of three major components: (1) the communications interface between the generator library and the other flow tools, (2) the design data model, and (3) the library specification.

A reference realization of a FLAME-based generator library exists in the form of the Generic Library for Adaptive Computing Environments (GLACE) [14]. This package has successfully been used in the COMRADE compiler [7], which compiles C into hybrid hardware/software applications for adaptive computer systems. GLACE uses a Java-based FLAME implementation, but could be called from other languages using the Java Native Interface (JNI).

15.4.1 The Flow Interface

The communications infrastructure and API provided by the FLAME Manager (Figure 15.8) replace static library files with an active function call–based interface. Clients in the main design flow can thus enter into a dialog with the module libraries and retrieve data specific to the actual parameter values of the current instance. In GLACE, the client queries accepted by the FLAME Manager are forwarded to the circuit generation code [6], resulting in the retrieval of circuit characteristics, or the creation of actual netlists.

15.4.2 The Data Model

The information exchanged in this manner just described is represented using the FLAME design data model. This model is partitioned into a number of task-specific views: A frontend compiler might request a "behavior" view to determine which functions are available for a given target technology. Later on, it could

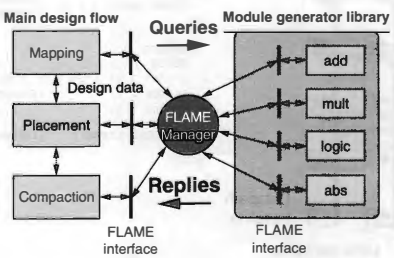


FIGURE 15.8 ■ FLAME system overview.

query for a "synthesis" view to retrieve area and timing characteristics for a specific module instance. Additional views include "topology" for layout shapes and port pitch, and "netlist," "placed," and "mapped" views describing the circuit itself. For the latter, standard formats such as EDIF are encapsulated inside the FLAME messages.

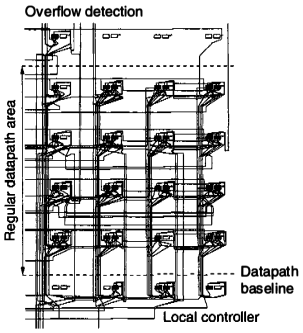
15.4.3 The Library Specification

The FLAME library specification describes a set of behaviors and interfaces. One or more of these can be attached to a hardware cell to precisely define its function for automatic use by another tool. For example, the cell of a runtime controllable adder/subtractor might have both the addition and subtraction behaviors attached. The interface carefully distinguishes between the logical (e.g., the operands of the adder) and the physical perspective (e.g., clock ports and clock enable signals). Furthermore, a FLAME interface extends beyond port specifications such as width and data type to the control characteristics of the cell. This can cover "start" and "done" signals as well as mode switches (e.g., alternating between addition and subtraction). By considering all of these aspects, another tool can choose the cell most applicable to a given task and automatically drive it correctly from the central datapath controller.

15.4.4 The Intra-module Layout

For efficiency, most module generators create circuits whose internal PEs have already been preplaced. In this case, the module generators and the datapath placement tools must agree on a set of common layout conventions. Otherwise, the regular target layout described in Section 15.1.1 will not be achievable.

Figure 15.9 shows such a regular layout, along with the FLAME description of its topology, using an unsigned 8-bit multiplier from GLACE as an example.




```
(TECHNOLOGY "Xilinx" "Virtex" "XCV50PQ240I" "-4"
 (STATUS QUERYOK "technology ok. area unit is 'CLB's...")

  (MATRIX                  ] Target device has matrix architecture

  (SHAPE                   ]
   ("CLB" (RECT 4 6 1))    ] Layout is a single 4x6 CLB rectangle
  )                        ] extending 1 unit below baseline

  (PORTLOC
   (PORTS
    ( ("a" 7 0) ("b" 7 0) ("start" 0 0) ("out" 7 0) ("done" 0 0) )
    (PITCH 2 1)            ] Port pitch for busses is 2 bits per CLB
    (COORD 0 0)

    (FOLDING LINEAR)       ] Layout is not folded
   )
  )
 )
```

FIGURE 15.9 ■ Module topology and FLAME reply.

The layout has the LSBs of the operand and result data busses aligned at a common baseline. This sequential HWOP has two irregular components, which are placed below and above the regular datapath region. For that reason, in order to preserve regularity within the stack, we had to leave extra space on the top and/or bottom to accommodate any irregularities (such as overflow detection, sign handling, etc.). All buses are spaced with a pitch of 2 bits per CLB of layout height.

15.5 THE MAPPING

Mapping techniques can be distinguished by whether they map in *N*:1 fashion (i.e., *multiple* CDFG operators into a single HWOP) or map (at least initially) in 1:1 fashion.

15.5.1 1:1 Mapping

Here each CDFG operator is considered individually. However, trade-off decisions can still occur with regard to the different HWOP alternatives for it:

Area/delay trade-offs can be performed to allow the selection of smaller but slower HWOPs for operations that are not on the critical path of the computation.

Topology matching can be performed to match the heights of the HWOPs across the datapath (Figure 15.10(a)). This can be necessary when a few HWOPs in the datapath are significantly wider than the rest (e.g., 64-bit modules in

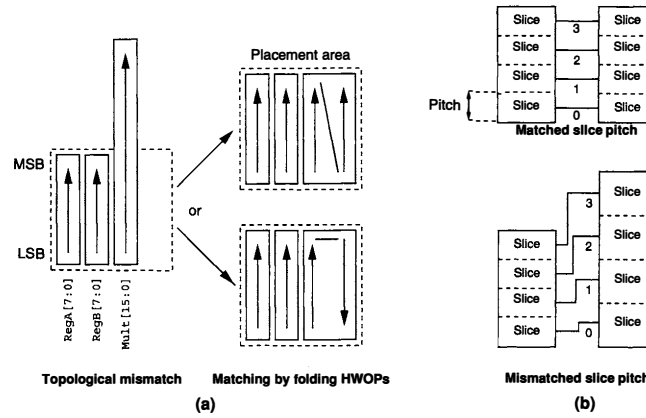


FIGURE 15.10 ■ Topology and pitch matching.

a mostly 32-bit datapath). Here regularity can be traded for area efficiency by selecting implementations for these modules that have been folded, doubling the length but halving the height.

Pitch matching occurs if modules in the library are available only with a limited number of pitch values. The goal here is to compose the datapath with the least number of pitch mismatches (Figure 15.10(b)).

Various techniques can be employed to solve these optimization problems. Since in general no single *best* solution exists for complex cases, it is practical to use an algorithm that can generate sets of good (Pareto-optimal) solutions. The SDI system [10] used a genetic algorithm in the floorplanning step to perform these calculations.

However, this approach is only applicable if a very flexible module library exists that actually gives the optimization heuristics some leeway to operate. This was the case with the PARAMOG library used in SDI, but the effort to implement this degree of flexibility is significant. More current module libraries, such as GLACE, often provide a smaller variety of implementations (generally just one) for each operator, allowing the replacement of complex heuristics with just a few simple rules for pitch and topology matching.

15.5.2 *N*:1 Mapping

In this approach, multiple operators can be mapped to a single HWOP, often using a tree-covering approach. The initial CDFG is split into a forest of trees (Figure 15.11) using techniques that splitt at multi-fanout nodes (between `B` and `D,F`) and possibly partially duplicate the operator cones rooted at the multi-fanout node (duplicating `A` into `A,A'`). While this limited approach no longer optimally solves the *graph*-covering problem, it is necessary in order to avoid the NP-completeness of computing the latter.

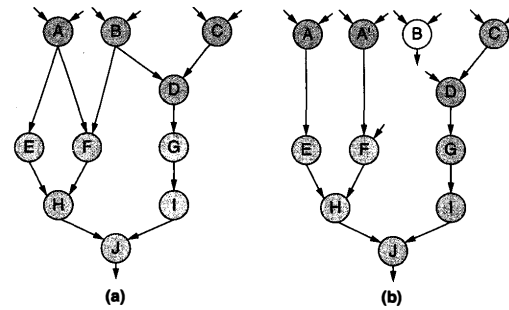


FIGURE 15.11 ■ Conversion of CDFG to a forest of trees: (a) input dataflow graph and (b) forest of dataflow trees.

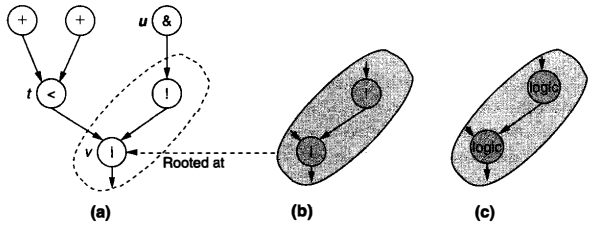


FIGURE 15.12 ■ Covering operator trees using patterns: (a) the dataflow tree, (b) the HWOP pattern P, and (c) HWOP equivalence class pattern C.

GAMA [3] employs a linear time algorithm using dynamic programming to cover the operator trees with HWOPs (Figure 15.12). This algorithm, which has its origin in the code generation steps of compilers, treats the operator(s) realizable by each HWOP as a *pattern*. Patterns are described as productions in a grammar, from which a code generator-generator creates the actual tree-covering code.

For each operator tree, the covering proceeds from the leaf nodes toward the root, applying all matching patterns that can be locally rooted at the currently examined node (`v` in the example, roots pattern `P`). A cost function computing delay and area characteristics determines the desirability of using the current pattern at this point. It is based on the cost of the currently tried pattern plus the previously computed costs (dynamic programming) of the fanin nodes to the pattern (`u, v` in the example). The "best" pattern covering each node/subtree is then selected using heuristics that either do a straight area minimization or attempt to additionally minimize delays. This best solution is then stored in the local root node, and the covering proceeds to the next node. Once the tree's root node has been matched with a best pattern, the final covering can be retrieved by starting with the root pattern and then processing the current pattern's fanin nodes. At each of these fanin nodes, the best pattern selection stored there is retrieved. This phase of the algorithm thus works recursively toward the leaves.

The algorithm has some limitations that *must* be worked around:

- First, tree covering in this fashion relies on the principle of optimality, where the combination of optimal solutions to subproblems leads to an optimal solution of the entire problem. This is indeed achievable when optimizing for minimal area. However, when attempting to minimize delays the timing criticality of operators can vary depending on later covering decisions. Thus, at the time of decision the criticality of the current node is not known.

  To mitigate this issue, GAMA attempts to estimate the criticality using an initial purely delay-oriented covering pass. Then the final covering proceeds in an area-minimizing fashion until the currently accumulated

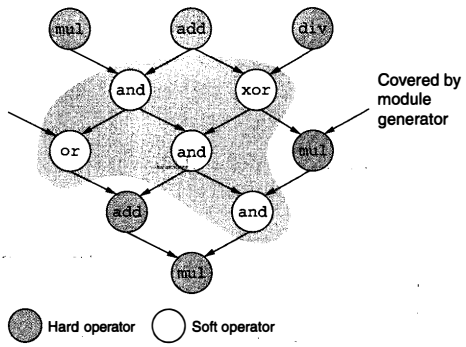


FIGURE 15.13 ■ Subgraph covering with flexible generator.

delay at a node exceeds its estimate. At this stage, the cost function is switched from area to delay minimization.

- Second, the runtime of the algorithm depends linearly on the number of patterns in the grammar (which equal different modules in the library). When the PEs of the target device are very flexible (e.g., LUT based), they can implement a wide spectrum of CDFG primitive operators (e.g., AND, OR, INV, ADD, SUB, combinations ...). Without further refinement to the approach, a straight description of this flexibility in the grammar will lead to an explosion in the number of rules. However, in practice, many operators are *equivalent* for mapping purposes. For example, all 2-input logic operators map in exactly the same way in all patterns in which they occur. This fact can be exploited by defining equivalence classes for all operators (e.g., logic, additive) and then defining the grammar rules in terms of these classes (C in Figure 15.13). Combined with the factoring out of common subpatterns, this significantly reduces the complexity of the grammar.

15.5.3 The Combined Approach

A completely different approach maps some operators in a 1:1 fashion and others in an *N*:1 fashion. This combination employs powerful module generators that can generate regular modules covering entire subgraphs of the CDFG. As an example, the LogicGen tool [20] can handle arbitrary multibit logical expressions, including shifts and permutations, with optional registering of the outputs. It extracts a regular structure from the input operators and synthesizes logic-optimized bit slices using SIS [16], which are then preplaced in a regular layout. To apply LogicGen, the CDFG is searched for the largest subgraphs of plain logic modules. Each of these clusters is then handed to the tool in its entirety, allowing it to exploit reconvergent fanouts, factorization, and the

like. All operators in the cluster are thus covered by a single, LogicGen-created HWOP. Operators that are not amenable to traditional logic optimizations, such as arithmetic and memories that are usually implemented on device-specific blocks, are then mapped into corresponding HWOPs in a 1:1 manner by dedicated module generators.

15.6 PLACEMENT

The HWOPs resulting from mapping have to be placed on the device fabric. This can happen either during mapping or in a separate step afterward. Placement approaches can be classified into three groups according to the nature of the generated placement (see Figure 15.14). Purely linear techniques create a one-dimensional arrangement of HWOPs in a single stripe. Others compute a placement consisting of multiple stripes, which is sometimes referred to as 1.5 dimensional or constrained two dimensional. A last group of algorithms generates arbitrary two dimensional arrangements, an approach closely related to the classical floorplanning or macro-module scenarios in ASIC tool flows.

15.6.1 Linear Placement

An example of linear placement, GAMA [3], performs a one-dimensional placement simultaneously with the mapping step (see Figure 15.15(a)). It assumes that the external I/Os to the datapath are located on only one side of the stripe (at the right in the figure). The roots of all subtrees are placed toward this I/O side, with the root of the entire HWOP tree directly adjacent to the I/Os (`op3` in the figure). Furthermore, the HWOPs within a subtree are all placed contiguously, which means that (at least initially) HWOPs from different subtrees (here `op1` and `op2`) will not be intermingled in the placement. The placement algorithm thus consists of recursively deciding in which linear order to place the fanin HWOPs of a node.

Note that the placement order *does* affect the routing delay between different HWOPs (Figures 15.15(b) and (c)). The timing estimates calculated in this fashion are used in the cost function guiding the mapping (covering the trees

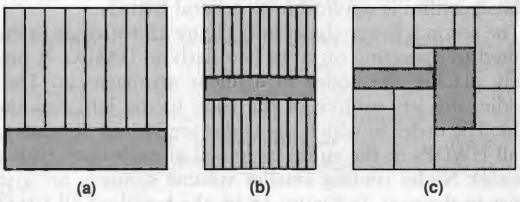


FIGURE 15.14 ■ Placements styles: (a) linear, (b) constrained two dimensional, and (c) full two dimensional.

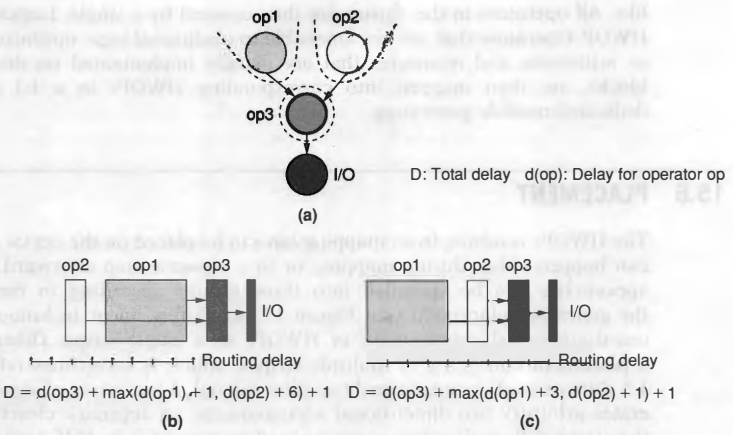


FIGURE 15.15 ■ Simultaneous tree covering and placement.

with HWOP patterns). The different trees of the forest (into which the CDFG has already been split) are placed in the stripe using a greedy algorithm that aims to place critical path trees close to each other. After this purely constructive initial placement, a greedy clustering algorithm can move HWOPs globally, across subtree and tree boundaries, in a further attempt to reduce routing delays. In practice, however, the quality gains achievable using this simple cleanup pass are negligible.

The techniques proposed by Ababei and Bazargan [1] are an example of a separate postmapping linear placement step, which employs two core algorithms to quickly determine linear placements in polynomial time. The first, shown in Figure 15.16(a), tries to heuristically compute a minimum bandwidth/minimum wirelength placement by transforming a matrix representation of the input circuit into band form and reflecting the transformation steps in HWOP swaps. This algorithm is applicable to general CDFGs.

The second, faster algorithm (Figure 15.16(b)) gives even better results, but is limited to operating on trees (similarly to GAMA). It proceeds topdown, recursively placing the nodes in a linear arrangement. The root is placed in the middle; the left subtree of the root, to the left; and the right subtree, to the right. The order in which the nodes are visited depends on the summed lengths of all HWOPs in the subtrees rooted at each node (this is called the *volume* of a node): Nodes rooting smaller volume subtrees are visited first, placing them closer to the root. In Figure 15.16, the length of all HWOPs is assumed to be 1.

In a refinement, Ababei and Bazargan [1] then extend the techniques for partial reconfiguration: A sequence of CDFGs is arranged so that previously placed

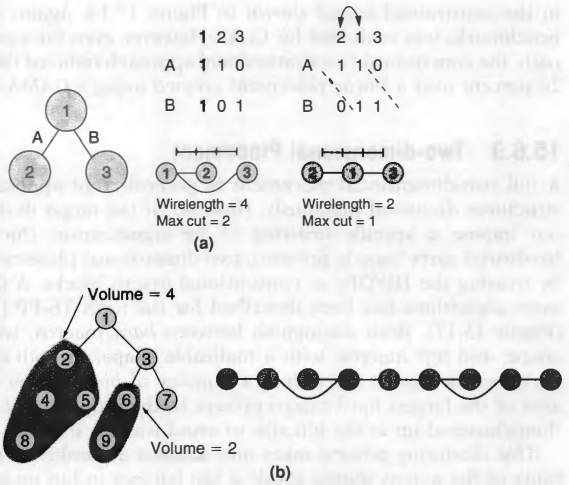


FIGURE 15.16 ■ Postmapping linear placement.

HWOPs and their interconnect can be reused in succeeding configurations, thus reducing the amount of configuration data. In the (albeit limited) experiments, up to 74 percent of HWOPs and 36 percent of inter-HWOP connectivity could be reused between configurations. However, with increased reuse, the delays and wirelengths began to deteriorate over independent placements (without reuse).

Other techniques that have been applied to compose linear stripes of HWOPs are spectral partitioning [13], genetic algorithms [10], and quadratic placement [22]. In the last case, it was determined that the quadratic placement needed to be postprocessed for by computing the optimal arrangement of HWOPs in a small window (less than or equal to five HWOPs long) using exact methods (e.g., exhaustive search, branch/bound). The process is then repeated, sliding the window across the stripe, until no further improvement can be realized.

15.6.2 Constrained Two-dimensional Placement

With the focus on linear datapath structures, published work on constrained two-dimensional or 1.5-dimensional datapath placement is sparse. Some limited results are reported by Thorns [18]: The CLAP tool first performs a clustering procedure similar to that in VPack [2] to determine the HWOPs to fit into each stripe. Then the horizontal arrangement of HWOPs inside a stripe, as well as the vertical and horizontal arrangements of entire stripes, is optimized using different moves in an adaptive simulated annealing algorithm [2], resulting

in the constrained layout shown in Figure 15.14. Again, only a limited set of benchmarks was evaluated for CLAP. However, even for a small 28-module datapath, the constrained two-dimensional approach reduced the delay by more than 20 percent over a linear placement created using a GAMA-like technique.

15.6.3 Two-dimensional Placement

A full two-dimensional placement is generally not applicable to the datapath structures discussed previously. However, if the target device architecture does not impose a specific ordering of bit significances (for example, when no hardwired carry logic is present), two-dimensional placement can be performed by treating the HWOPs as conventional macro blocks. A family of such placement algorithms has been described for the tools TS-FP [5] and Frontier [17] (Figure 15.17). Both distinguish between *hard* macros, with fixed rectangular shape, and *soft* macros, with a malleable shape. In both cases, the algorithms partition the device fabric into a number of *bins*, whose size depends on the area of the largest hard macro present in the input circuit. Smaller macros are then clustered up to the bin size to avoid wasting intrabin area.

This clustering process takes into account a number of factors: the compatibility of the macro shapes inside a bin (shapes in bin must geometrically fit in the bin bounding box), the relative size of the cluster compared to the entire circuit, the relative size of the blocks in the cluster, and the connectivity of the macros in the cluster. If, after clustering, the number of clusters exceeds the number of available bins, the size of the bins is increased and the clustering process is repeated. The clusters are then assigned to individual bins using standard placement techniques.

Intrabin placement is now performed constructively. TS-FP places hard macros from right to left by abutment, leaving the left side of the bin free for

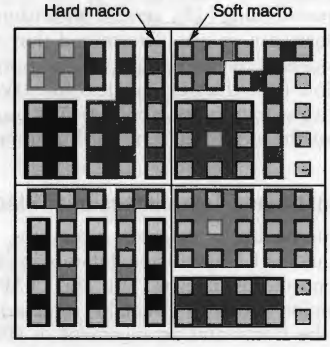


FIGURE 15.17 ■ Bin-based two-dimensional HWOP floorplanning.

soft macros. Frontier (shown in Figure 15.17) spreads the hard macros horizontally across the entire length of a bin, leaving the unused space *between* them for the soft macros. These are then placed in the free regions. TS-FP performs a geometrical minimax matching, reshaping the logic of each soft macro to fit into available space while attempting to keep the macros' initial internal placement intact. Frontier uses a simpler approach, laying a snakelike pattern across the free space, filled by sequentially selecting from the soft macro an unassigned PE that leads to the minimal overall wirelength. To improve routability, Frontier additionally employs a final low-temperature annealing pass for the PEs in the soft macros. These are allowed to move across macro and bin boundaries. The annealing start temperature is set sufficiently high to allow perturbation of the layout but low enough to ensure that the basic bin structure is kept intact.

15.7 COMPACTION

In a 1:1 mapping of simple CDFG operators (for example, trivial logic gates) to HWOPs, the PEs inside an HWOP are often not used to their full capacity. This inefficiency is worse when coarse-grained PEs are being targeted, and it accumulates across all HWOPs implementing simple operators. Figure 15.18 shows an example of this in which the functionality of a 2-input multiplexer described using simple logic HWOPs requires three PEs—even though it would completely fit in a single PE.

Compaction dissolves the boundaries of selected HWOPs and optimizes their contents as a whole, resulting in the creation of a new super-HWOP that realizes all of the original functions in a smaller/faster fashion. The procedure can generally be split into four phases:

1. Select the HWOPs to merge and compact.
2. Analyze regularity across the selected HWOPs to derive new master slices.
3. Optimize the newly discovered master slices.

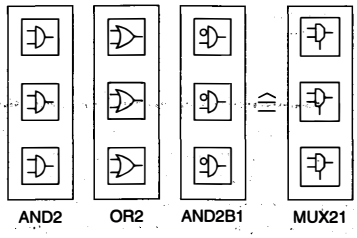


FIGURE 15.18 ■ Wasted space in the layout of very simple HWOPs.

4. Construct the super-HWOP by instantiating and placing the optimized master slices according to the regular inter-HWOP structure discovered previously.

15.7.1 Selecting HWOPs for Compaction

Two approaches have been proposed for selecting candidate HWOPs for compaction. Early work, such as the Structured Design Implementation (SDI) approach [8–10], aimed to keep a precomputed one-dimensional placement intact and so only considered connected *neighboring* HWOPs to compact. However, more recent research [21, 23] shows that better area efficiency is achievable by selecting candidates purely based on their connectivity, independent of any placement.

Additionally, depending on the actual optimization procedures to be performed on the selected candidates certain HWOPs, despite being connected and adjacently placed, might later be unsuitable for compaction. For commonly used optimization methods, this category generally includes HWOPs exploiting target device–specific features such as hardwired carry chains or fixed-function blocks (e.g., multipliers or memory blocks). Thus, their enclosing HWOPs are exempt from compaction.

15.7.2 Regularity Analysis

Since compaction is a regularity-preserving transformation, regularity aspects have to be considered both in its preparation and while it is taking place. Although methods exist to determine regular patterns in arbitrary circuits [12,15], it is much more efficient to keep track of this data from the moment of HWOP circuit generation. The method developed by Ye and colleagues [21,23] requires knowledge of the netlists at the bit slice level. SDI, supported by the powerful PARAMOG module generator library, goes beyond that by explicitly describing both regularity (in the model described in Section 15.1.1) and hierarchy (using master slice/bit slice relationships).

Based on the detailed data, SDI can consider more complicated structures for regular compaction. Figure 15.19 shows how it can isolate two new master slices and their instances from the HWOPs `ALU` and `LSHR` under compaction, even though the number of bit slices between these HWOPS differs. The inter-HWOP regularity consists of a 2-zone stack. The top zone holds a single instance of a newly discovered master slice, which consists of the original master slices `ALU4`, `TOPDWN`, and `DWN`. The second zone has two instances of a new master slice, which consists of `ALU4` and two instances `DWN`. Ye and colleagues' technique [23] would not attempt to merge these two HWOPs, as it can only compact HWOPs with the same number of bit slices.

15.7.3 Optimization Techniques

The core of compaction lies in the intermodule optimizations applied to the super-HWOP constructed by merging the original HWOPs. Here, Ye and

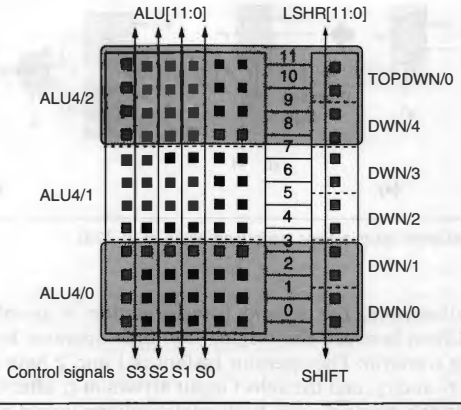


FIGURE 15.19 ■ Extracting inter-HWOP regularity.

colleagues' approach [23] performs two additional steps compared to SDI: word-wide transformations that affect entire HWOPs followed by exploiting the context (external signals) of the HWOPs under compaction. The main processing step of both SDI and the system of Ye et al. [23], however, consists of applying traditional logic synthesis and optimization algorithms at the bit slice level.

Word-level optimization

Word-level optimizations, which in Ye and colleagues' approach [23] were performed manually, alter the datapath from the structure described in the original CDFG. Two of the transformations are shown in Figure 15.20. The first, shown in Figure 15.20(a), tentatively collapses trees of multiplexers into a single wide multiplexer, modifying the select logic appropriately. If this replacement requires more area than the original version, the original version is retained. This transformation cannot be performed by optimizing at the slice level, because the multiplexer select logic is not part of the regular area holding the bit slices.

The second transformation, shown in Figure 15.20(b), is called operation reordering. It attempts to reduce area by restructuring individual multiplexers. A subcircuit, in which a multiplexer selects a single result from multiple identical operator instances, is turned into a form where multiple multiplexers select from a set of inputs feeding a single operator instance. Under the assumption that a multiplexer is smaller than the operator, this reduces area. Note, however, that this is not always the case: In many fine-grained architectures that combine LUTs and arithmetic carry logic within a logic block, both multiplexers and adders/subtractors may occupy the same number of logic blocks.

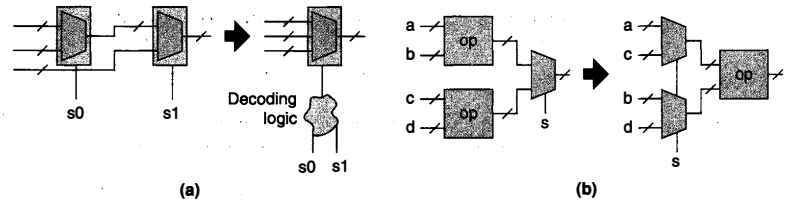


FIGURE 15.20 ■ Word-level optimizations performed by Ye et al. [23].

Furthermore, the second transformation is problematical in that it loses parallelism between the original multiple operator instances. Consider the following scenario: The operator instances 1 and 2 have data-dependent execution times t_1 and t_2, and the select input arrives at t_s after the operands of the operators. In the original case, both computations would be speculatively performed in parallel. The delay of the entire structure is then $\max(t_s, t_1)$ if the result of the first operator is selected, and $\max(t_s, t_2)$ otherwise. In effect, the delay of the select input hides part of the operator delay. In the reordered form, the operator can begin computation only *after* the select input has become valid, leading to total delays of $t_s + t_1$ and $t_s + t_2$, respectively.

The ramifications of such a transformation can be appraised to their full extent only when *building* the CDFG in the first place—for example, when considering instruction-level parallelism in a hardware compiler. At the same time, the multiplexer tree collapsing could also be performed, dispensing with a special optimization pass later in the design flow. Instead, the CDFG would contain generic multiplexer operator nodes with a varying number of inputs. During the mapping step, the module library would determine the best realization for each operator, also considering global issues such as the criticality of their signal paths.

Context-sensitive optimization

The tool flow designed by Ye and colleagues [23] then performs an additional suite of optimizations that also considers the super-HWOP in the *context* of the surrounding datapath (Figure 15.21). To this end, it partitions the super-HWOP into m-bit-wide *superslices*, each of which may thus consist of multiple bit slices. Next, the external ports of each superslice are examined for certain connectivity patterns and the presence of constant values. The actual optimizations are then performed in this superslice-specific context.

Constant inputs are absorbed for each of the superslices (Figure 15.21(a)). Similarly, nets that connect slice inputs directly to outputs are also pulled into the slice (Figure 15.21(b)). Multiple slice inputs all sourced by the same external signal are replaced by a single input that fans out to the original internal sinks (Figure 15.21(c)).

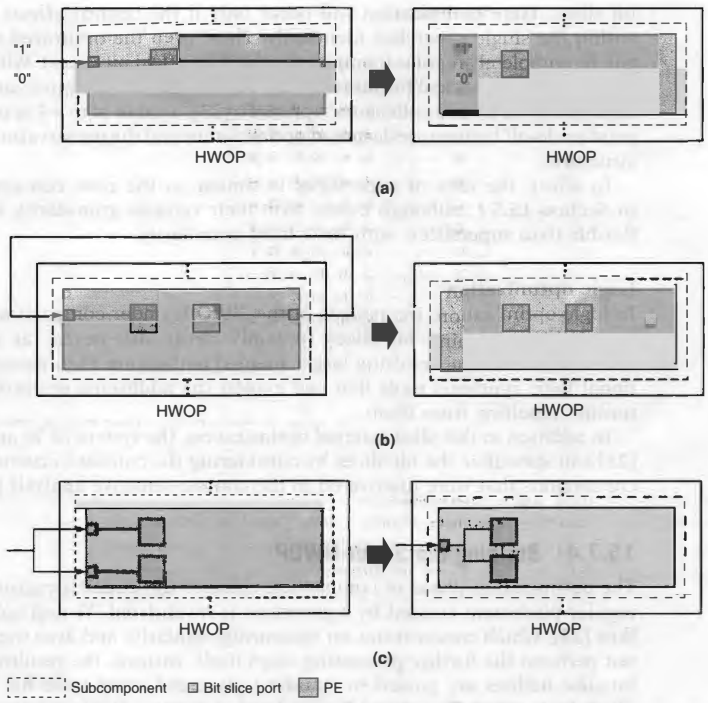


FIGURE 15.21 ■ Context-sensitive optimizations performed by Ye et al. [23].

These transformations occur *only* if all bit slices within a superslice have *identical* context (e.g., all bit slice input ports `a` within a superslice have the constant value 0 applied from the outside). Otherwise, the superslice is left unchanged.

The quantity m is thus a control for the internal regularity of the super-HWOP. With $m = 1$, the super-HWOP is partitioned into *width* superslices, each consisting only of a single 1-bit-wide bit slice. Each of these narrow superslices is thus affected by only very limited context: A superslice's single bit slice can be perfectly matched to its context (e.g., allowing the absorption of even irregular constant input patterns into each slice) in the super-HWOP. However, while allowing a large degree of optimization, this setting of $m = 1$ potentially introduces significant irregularity into the optimized super-HWOP (it may end up consisting of completely different bit slices). At the other extreme, with $m = width$, the super-HWOP is covered by a single superslice containing m 1-bit-wide

bit slices. Here optimization will occur only if the context affects *all* bit slices within the single superslice identically. Thus, even the optimized super-HWOP will be completely regular (composed only of identical bit slices). With the context required to be identical for more bit slices, however, fewer optimization opportunities arise. In Ye and colleagues' approach [23], a value of $m = 4$ is suggested as a good trade-off between widespread optimization and the preservation of a regular structure.

In effect, the idea of superslices is similar to the *zone* concept introduced in Section 15.5.1, although zones, with their variable granularity, remain more flexible than superslices, with their fixed granularity.

Logic optimization

In logic optimization, the netlists of the HWOPs under compaction are merged into HWOP-spanning bit slices (possibly newly discovered, as discussed in Section 15.7.2). The resulting larger merged netlists are then passed to conventional logic synthesis tools that can exploit the additional optimization opportunities resulting from them.

In addition to this slice-internal optimization, the system of Ye and colleagues [23] can specialize the bit slices by considering the constant *external* inputs and connections that were discovered in the context-sensitive analysis pass.

15.7.4 Building the Super-HWOP

The optimization phase of compaction changes the circuit structure. Thus, any regular placement created by a generator is invalidated. Ye and colleagues' tool flow [23], which concentrates on measuring regularity and area overheads, does not perform the further processing steps itself. Instead, the resulting optimized bit-slice netlists are passed to standard place-and-route tools for further handling. In contrast, Structured Design Implementation (SDI), additionally aiming at delay minimization, attempts to restore a regular placement for the optimized super-HWOP. This *micro-placement* step, shown in Figure 15.22, exploits regularity by operating at the master slice level. The results are then automatically replicated across the entire super-HWOP according to its zone structure.

Microplacement operates on cells (LUT and FF blocks), and proceeds in two phases:

1. The placement of cells horizontally, grouped into columns (Figure 15.22(a)). This is performed across all master slices, ensuring that cells sharing a control net are located adjacently to a vertical routing channel. Such an arrangement allows the efficient routing of high-fanout control nets on vertical long lines. Analogously, cells on interslice nets are horizontally aligned to allow short-distance routing. The remaining cells are placed in a timing-driven fashion, using estimates for the as yet unknown vertical position. This placement phase optimizes the super-HWOP in the geometric context of the datapath by constraining the master slice I/O ports to the appropriate sides of the layout.

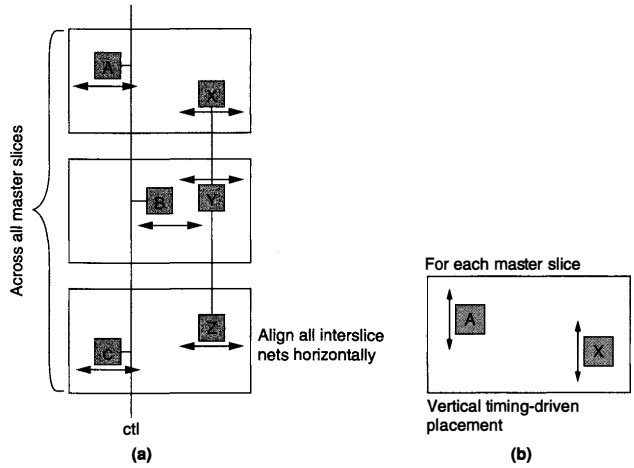


FIGURE 15.22 ■ Horizontal and vertical microplacement to restore regularity to compacted super-HWOP.

2. The placement of cells within the columns vertically (Figure 15.22(b)). This step looks across master slice boundaries only initially when performing a timing analysis on the entire super-HWOP. After annotating the timing criticalities calculated in this manner on the master slice ports, each master slice is placed independently in a purely timing-driven fashion. The timing model used here models the intricacies of the target device routing network and leads to measurably better results than simple Manhattan distances.

Since the microplacement results are replicated according to the regular structure previously determined for the super-HWOP, it is advantageous to employ high-quality algorithms. To this end, SDI uses a combination of well-converging heuristics and exact integer linear programming (ILP)-based methods. The latter are feasible because of the separation of the placement problem into horizontal and vertical phases, and the relatively small circuit size of the master slices (compared to the entire super-HWOP).

15.7.5 Discussion

Implementing a circuit in a regular bit-sliced fashion is generally associated with some area overhead compared to synthesizing/optimizing the circuit in an irregular flat manner. The reason is that the bit-slice boundaries prevent the exploitation of cross-slice optimization opportunities. The system devised by Ye and colleagues [23], with its additional interslice optimizations, observed area overheads of between 0 percent and 7.4 percent for superslice granularity

values of $m = 1$ (fully irregular) and $m = 32$ (fully regular with a width of 32 bits), respectively. For SDI, which lacks these optimizations, area increases of up to 17 percent were observed over the flat solution. However, by scrupulously maintaining a regular structure, SDI was able to reduce the total delay in the circuit by up to 33 percent over the flat implementation. A combination of the interslice optimizations of Ye and colleagues [23] with the microplacement of SDI appears to be promising to achieve further gains.

15.8 SUMMARY AND FUTURE WORK

This chapter presented an overview of some of the many issues to consider when realizing datapaths on reconfigurable logic devices. The aspect of *regularity* is a crucial one and must be considered both at the level of the target device architecture and during the operation of the EDA tools. Module generators are an efficient means to actually create the circuits making up the datapath. However, in addition they must offer sufficient metadata to the rest of the tool flow as a base for effective transformation and optimization steps.

With increasing requirements on datapath performance, tool flows and algorithms must keep up with improvements in device architectures. All of the techniques described here have the potential for further refinement. Refinement opportunities include module generators that better support specialization, floorplanning with constrained two-dimensional placement, and a compaction technique in which the best of these refinements is combined.

References

[1] C. Ababei, K. Bazargan. Non-contiguous linear placement for reconfigurable fabrics. *Proceedings of the of the Reconfigurable Architectures Workshop*, 2004.

[2] V. Betz, J. Rose, A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer, 1999.

[3] T. J. Callahan, P. Chong, A. DeHon, J. Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. *Proceedings of the of the International Symposium on Field-Programmable Gate Arrays*, 1998.

[4] T. Callahan, R. Hauser, J. Wawrzynek. The GARP architecture and C compiler. *IEEE Computer* 33(4), 2000.

[5] J. M. Emmert, D. Bhati. A methodology for fast FPGA floorplanning. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1999.

[6] B. Hutchings, P. Bellows. J. Hawkins, S. Hemmert. A CAD suite for high-performance FPGA design. *Proceedings of the of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[7] N. Kasprzyk, A. Koch. High-level-language compilation for reconfigurable computers. *Proceedings of the International Conference on Reconfigurable Communication-centric SoCs*, 2005.

[8] A. Koch. Module compaction in FPGA-based regular datapaths. *Proceedings of the Design Automation Conference*, 1996.

[9] A. Koch. Structured design implementation—A strategy for implementing regular datapaths on FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1996.
[10] A. Koch. *Regular Datapaths on Field-Programmable Gate Arrays*. CS doctoral thesis, technical, University of Braunschweig, 1997.
[11] A. Koch. On tool integration in high-performance FPGA design flows. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 1999.
[12] T. Kutzschebauch, L. Stok. Regularity-driven logic synthesis. *Proceedings of the International Conference on Computer-Aided Design*, 2000
[13] J. Li, J. Lillis, L. T. Liu, C. K. Cheng. New spectral linear placement and clustering approach. *Proceedings of the Design Automation Conference*, 1996.
[14] T. Neumann, A. Koch. A generic library for adaptive computing environments. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2001.
[15] R. Nijssen, J. Jess. Two-dimensional datapath regularity extraction. *Proceedings of the ACM SIGDA Physical Design Workshop*, 1996.
[16] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, et al. SIS: A system for sequential circuit synthesis. *EECS Memorandum No. UCB/ERL M92/41*, University of California, Berkeley, 1992.
[17] R. Tessier. Frontier: A fast placement system for FPGAs. *Proceedings of the International Conference on VLSI*, 1999.
[18] F. Thorns. *CLAP—Clustering and placement*. Diploma thesis, Technical University of Braunschweig, 2003.
[19] C. C. Vi, D. Lewis. Area-speed trade-offs for hierarchical field-programmable gate arrays. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1996.
[20] C. Wewetzer. *A Universal Generator for Logic Circuits on FPGAs*. Diploma thesis, Technical University of Braunschweig, 2005.
[21] A. G. Ye. *Field-Programmable Gate Array Architectures and Algorithms Optimized for Implementing Datapath Circuits*. Doctoral thesis, University of Toronto, 2004.
[22] T. T. Ye, G. De Micheli. Data path placement with regularity. *Proceedings of the International Conference on Computer-Aided Design*, 2000.
[23] A. G. Ye, J. Rose, D. Lewis. Synthesizing datapath circuits for FPGAs with emphasis on area minimization. *Proceedings of the International Conference on Field-Programmable Technology*, 2002.
[24] A. G. Ye, J. Rose. Measuring and utilizing the correlation between signal connectivity and signal positioning for FPGAs containing multibit building blocks. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2005.

CHAPTER 16

SPECIFYING CIRCUIT LAYOUT ON FPGAS

Satnam Singh
Programming Principles and Tools Group
Microsoft Research Cambridge

Typically, the layout of a circuit implemented on a field-programmable gate array (FPGA) is computed automatically by vendor design tools. This computation often results in an acceptable mapping of logical wires in the design onto actual physical routing resources on the FPGA that meets the designer's performance requirements. Instead of relying on automated tools, however, a designer could try to use an FPGA by explicitly stating the configuration of individual logic blocks and explicitly specifying the routing between them. One almost never needs to program an FPGA at this basic and raw level, and often the proprietary nature of programming information makes it difficult or impossible to take this approach. Still, the FPGA design flow provides a powerful set of abstractions that allow a designer to think in terms of structural circuit netlists, which can be automatically converted into programming information for FPGAs. Structural netlists are abstracted further by the synthesis flow, which allows designers to think of circuit functions in an algorithmic or sequential manner.

16.1 THE PROBLEM

Although it is just about tractable for humans to explicitly specify the layout of some mapped circuits on an FPGA, explicitly specifying the routing is extremely difficult because of the complex nature of the wiring resources. A screen snapshot of some of these resources on a Xilinx FPGA is shown in Figure 16.1. As one can see there are simply too many wires and interconnection options for a human to economically make routing decisions. However, providing layout hints or even explicit layout for only the logic blocks is a reasonable approach, because designers often have good intuition about a desirable layout but little intuition about how to use the underlying routing resources. By specifying some aspects of the layout, the tools can produce a faster circuit than is possible with purely automatic approaches [4]. The ability to specify layout helps with other operations like dynamic reconfiguration [3].

A design that contains a mixture of manually and automatically placed blocks is shown in Figure 16.2. The rectangular block is the core of the Xilinx MicroBlaze soft processor, which is designed with explicit layout specification for each gate. The other blocks are components, such as the system bus and peripherals,

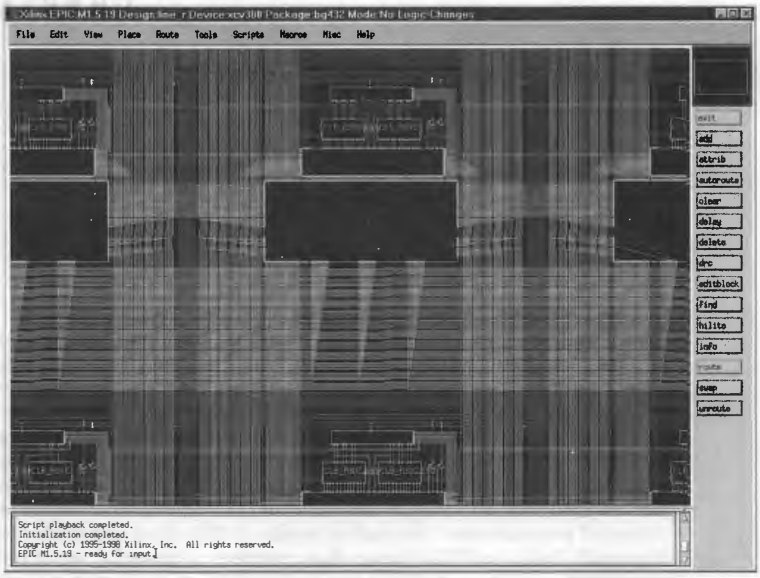


FIGURE 16.1 ■ FPGA routing resources.

that are designed without explicit layout specification—the placer automatically decides where to put these gates. Many of Xilinx's Core Generator IP core blocks are designed with explicit layout information. By giving a good layout for a circuit, one can indirectly control performance by influencing wiring that contributes to the critical path. Also, by providing user-specified placement information for small blocks that will be reused for many designs, the upfront design effort can be worthwhile.

An automatic placement algorithm can often find an acceptable placement that meets the design requirements for speed, area, power, and so forth. However, when such an algorithm cannot find a good placement—or any placement at all—there is often little the designer can do. In these situations it would be desirable either to allow the designer to influence the placement by adding extra information or to allow her to partly or completely specify the layout of her circuit. For circuits that need very high performance or that need to be very compact, often only a user-specified layout can achieve the required results. For example, the design shown in Figure 16.3 has been automatically placed and routed without any user-specified layout information. The same design can be augmented with user-specified layout information to produce the layout shown in Figure 16.4, which performs approximately 30 percent faster.

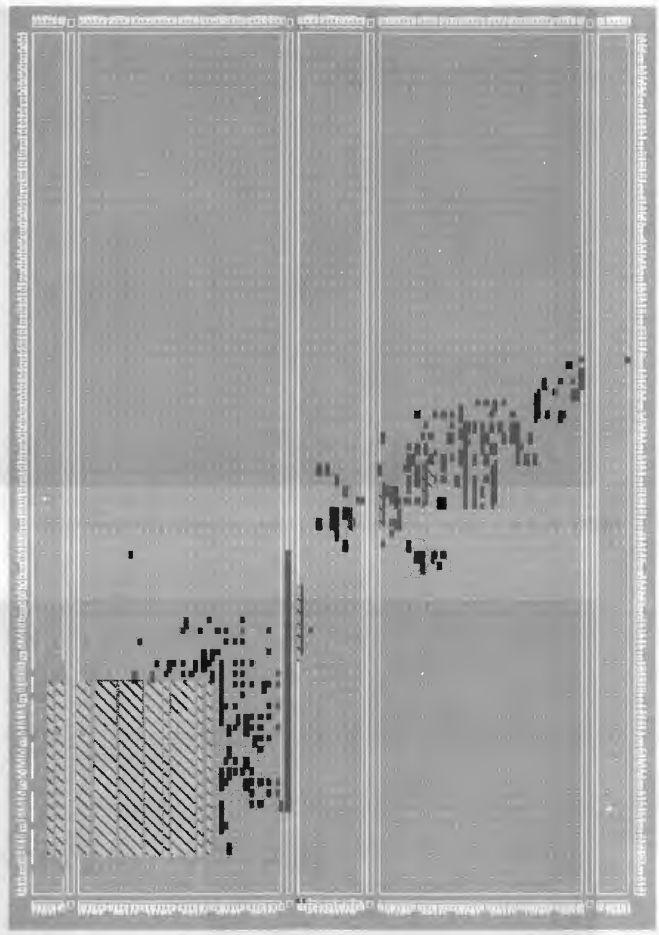


FIGURE 16.2 ■ An example of manually and automatically placed blocks.

Providing explicit layout information can also reduce the runtime of FPGA implementation tools, mainly because of the reduction in work for the automatic router. This is particularly important for uses of reconfigurable computing that create custom circuit designs for each problem instance, when placement and routing tool runtimes are part of the system's execution time (see Chapter 5).

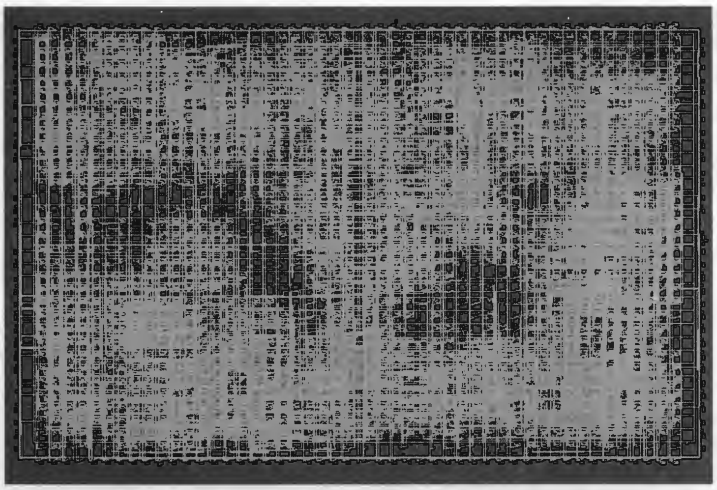


FIGURE 16.3 ■ A design with no explicit layout (automatic place and route).

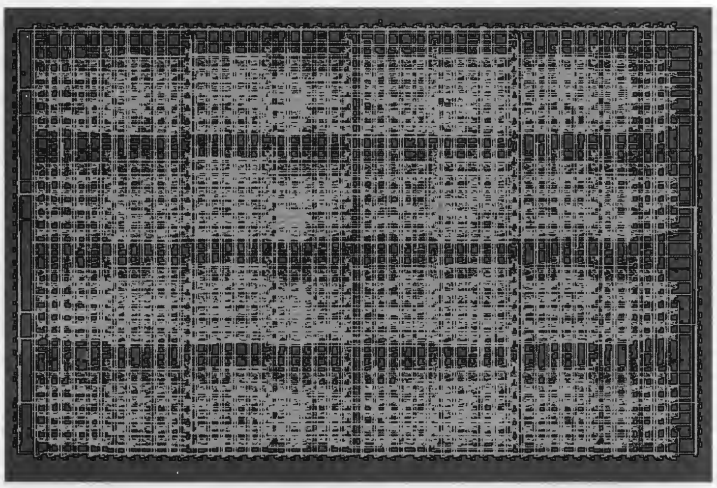


FIGURE 16.4 ■ A design with totally explicit layout.

An important reason for explicitly specifying absolute or relative layout is to support runtime reconfiguration, which is much easier to perform if the system knows the shape and location of circuits to be swapped in and out or updated in place.

This chapter reviews various techniques for specifying the layout of circuits for FPGAs. We illustrate our examples using Xilinx's FPGA technology, which provides an accessible mechanism for specifying circuit layout.

16.2 EXPLICIT CARTESIAN LAYOUT SPECIFICATION

Explicit Cartesian layout specification involves specifying the location of some or all logic elements using a two-dimensional coordinate system. One form of explicit layout involves giving an absolute location for each gate in the mapped netlist. This approach is not common because it does not permit the specification of reusable layouts, which can be replicated throughout the FPGA, and such descriptions may be unnecessarily specific to a particular FPGA chip or family. A more common approach is the relative layout specification.

Xilinx's placement tools can take user-specified layout information either as absolute or as relative locations. Relative locations identify the bottom left corner of a block of logic. Blocks may be placed relative to each other in a hierarchical fashion.

The layout of a gate or block is achieved by attaching a special attribute called `LOC` for absolute layouts and `RLOC` for relative layouts. The VHDL code in Figure 16.5 illustrates the design of a 1-bit adder in which two of the gates have their relative layout explicitly specified.

In the figure, the attribute mechanism of VHDL is used to attach a relative layout attribute to two instances: one for an `xor` gate and the other for an `or` gate. The RLOC attribute specifies the relative location of the CLB that will be used to realize a given gate. One may further specify the specific lookup table (LUT) within the CLB or omit this specification to allow the placer to make the choice.

```
architecture structural of adder is
   signal xor1_out, and1_out, and2_out, or1_out : std_logic;
   attribute RLOC of xor1 is "X2Y5" ;
   attribute RLOC of or1 is "X3Y4" ;
begin
   xor1: xorg port map (in1  =>a, in2  => b, out1 => xor1_out);
   xor2: xorg port map (in1 => xor1_out, in2 => cin, out1 => sum);
   and1: andg port map (in1 => a, in2 => b, out1  => and1_out);
   or1: org port map (in1 => a, in2 => b, out1  => or1_out);
   and2: andg port map (in1 => cin, in2 => or1_out, out1  => and2_out);
   or2: org port map (in1 => and1_out, in2 => and2_out, out1  => cout);
end structural;
```

FIGURE 16.5 ■ An example of explicit layout in VHDL.

Explicit layout works well for small circuits that are not parameterized and for VHDL and Verilog descriptions that do not make use of statements like `for . . . generate`. In parameterized circuits, layout specifications become quite complex, with location specifications becoming difficult to comprehend layout calculation expressions. Because layout specifications are string attributes, one has the extra complexity of performing integer index calculations and then converting them into their string representation. This is often too tedious to be practical. The difficulty of working with explicit Cartesian layout specifications has led to the development of various systems to specify layout at a higher level of abstraction.

16.3 ALGEBRAIC LAYOUT SPECIFICATION

Algebraic layout specification typically does not involve Cartesian coordinates. Instead, one specifies the geometric relationship between one circuit and another. These specifications (or constraints) are gathered together, and a *deterministic* layout can then be *calculated*. Techniques such as this have been shown to work for parameterized circuits, circuits with irregular layouts, and recursively defined circuit layouts. Such descriptions are also slightly less tightly coupled to a specific FPGA architecture or family. In this section we describe how algebraic layout specifications work in the Lava system [1]. Several other systems are based on similar principles.

Lava is based on the concept of *circuit combinators*, which are calculations that take circuits as inputs and deliver a circuit as a result; essentially, they are procedures that compute on circuit descriptions. One important design decision in Lava is the coupling of the description of circuit behavior and that of circuit layout by using circuit combinators that compose both behavior and layout. This works well when the circuit layout description can use the same patterns as those of the circuit behavior. When this is not the case, one can directly use Cartesian coordinates.

One important combinator is the serial composition combinator. This combinator, written as an infix operator >->, takes two circuits `R` and `S` as arguments and delivers a circuit comprising `R` with its output connected to the input of `S`. Furthermore, `R` is laid out to the left of `S`, which matches a left-to-right dataflow.

Figure 16.6 shows the composition of an AND2 and an INV gate. Each gate or circuit starts life in its own coordinate system. The basic gates each have a height and width of one unit. The serial composition combinator sees that the circuit on the left has a width of one and then translates the circuit on the right by one unit. These algebraic descriptions can be arbitrarily nested. When the system needs to produce a VHDL or EDIF netlist, the algebraic specifications are computed and a netlist that contains RLOCs is automatically generated.

Notice, now that layout has been combined with behavior, that there is a need for several kinds of serial composition combinators. Those for right-to-left (<-<), bottom-to-top (^), and top-to-bottom (V) layout are all supported by Lava.

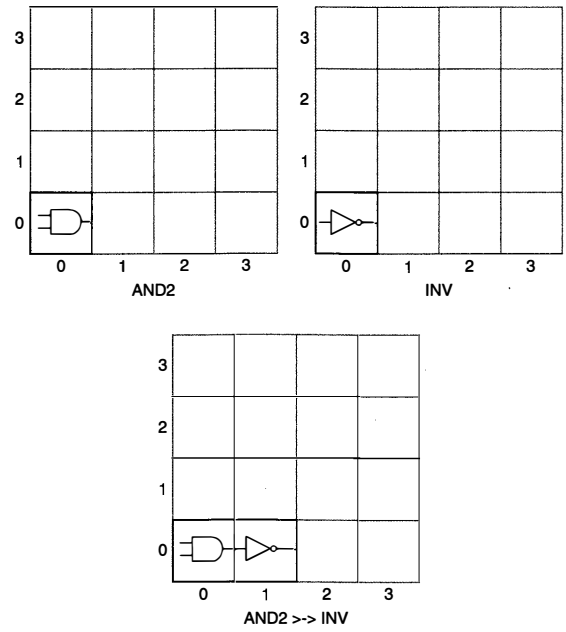


FIGURE 16.6 ■ Layout calculation.

Figure 16.7 shows the layout produced by the Lava circuit expression AND2 >-> FD clk, which serially composes an AND2 gate with an FD component (a flip-flop).

In the Xilinx device, a LUT–flip-flop pair is called a *slice*. AND2 and a flip-flop (FD) each have a width and height of one unit, or slice, causing the FD flip-flop to be mapped to a slice to the right of the slice containing the function generator for the AND2 gate. Such a process is very inefficient. To allow circuits to be composed but mapped to the same location we can use the serial overlay operator, written as >|>. This is illustrated on the right side of Figure 16.7 and shows both the AND2 gate and the FD flip-flop mapped to the same location.

The circuit tiles presented so far have only one-dimensional dataflow. Four-sided tiles allow us to specify dataflow horizontally and vertically. Rather than introduce a new basic tile, a 4-sided tile can be represented in terms of a 2-sided tile. This is done by considering the 4-sided tile as a function that maps a pair of input values to a pair of output values. Each element of each pair corresponds to a face of the tile, as shown in Figure 16.8. We can now define a below combinator, which places one tile below another (`r` below `s` is shown in the middle of the

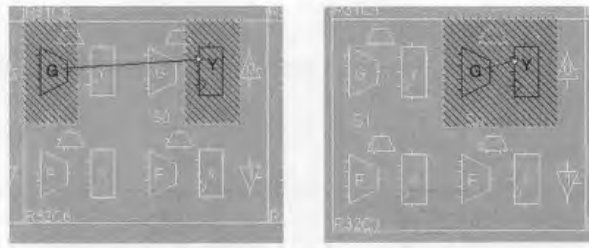


FIGURE 16.7 ■ The overlay combinator: (a) AND2 >-> FD clk; (b) AND2 >|> FD clk.

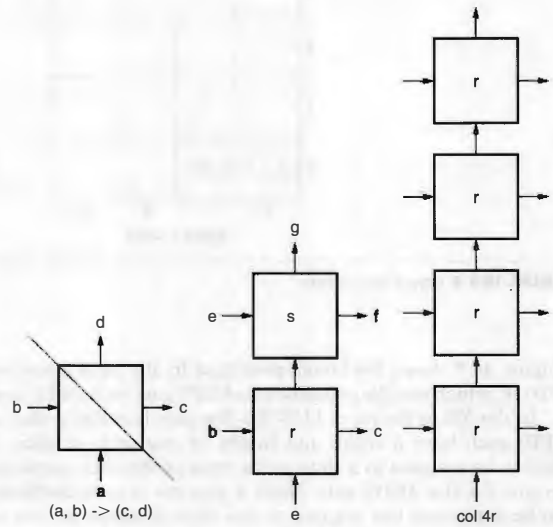


FIGURE 16.8 ■ Four-sided tiles.

figure). The col combinator replicates a tile vertically (`col 4 r` is shown on the right of the figure).

A concrete example of the col combinator is shown in Figure 16.9. The col combinator acts on a 1-bit adder circuit that takes a pair as input (the carry-in [`cin`] and another pair of values to be added) and delivers a pair as its output

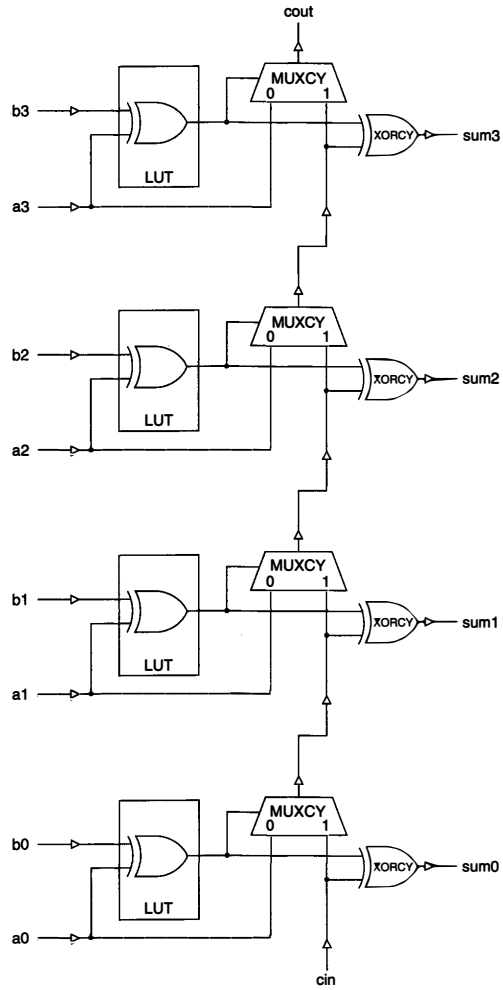


FIGURE 16.9 ■ A col 4 1-bit adder.

(the sum and the carry-out [`cout`]). It will connect the carry-out of each stage to the carry-in of the next stage. Furthermore, it will vertically stack the 1-bit adders.

The actual FPGA layout produced for `col 8 oneBitAdder` is shown in Figure 16.10. In this case the automatic placement tools would have produced the same layout because the carry chain would have constrained a vertical alignment for the circuit. Through combinations of these regular abutment techniques, very complex but regular circuits can be efficiently created.

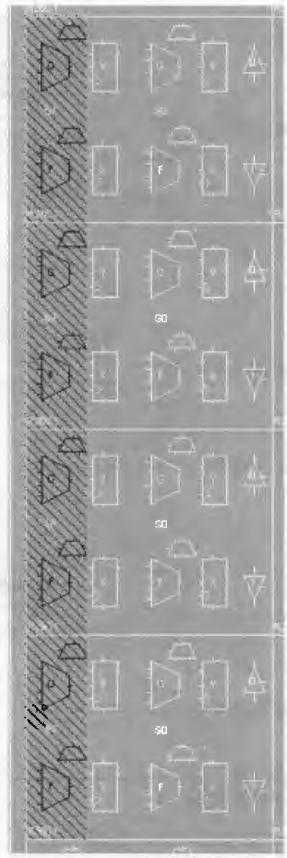


FIGURE 16.10 ■ FPGA layout of `col 8 oneBitAdder`.

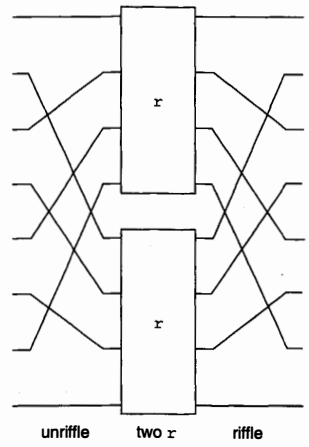


FIGURE 16.14 ■ The `ilv` combinator.

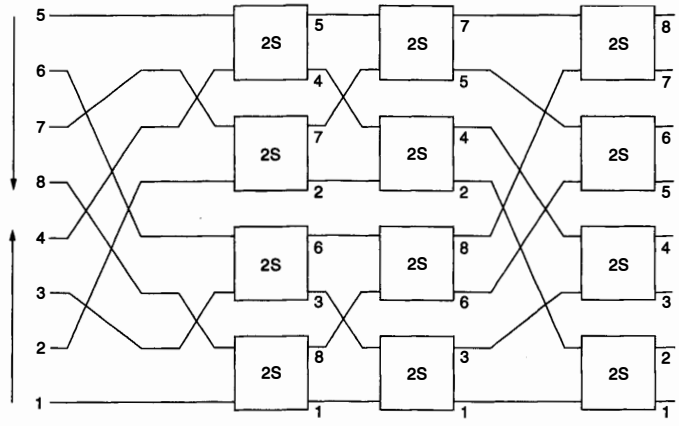


FIGURE 16.15 ■ A bitonic merger.

specifies the *layout* of the merger circuit using algebraic layout specifications. This circuit is a bitonic merger that can merge its inputs as long as one half of the input is increasing in the opposite order from the other half, as shown in the figure.

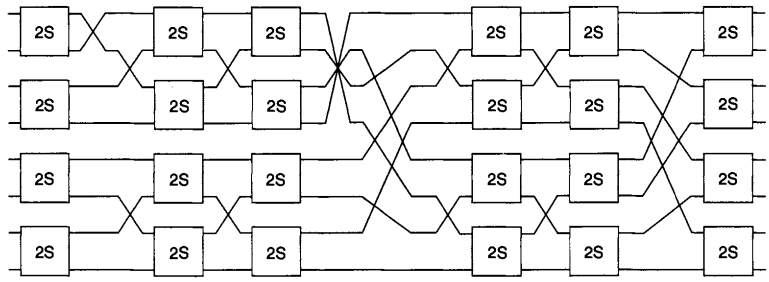


FIGURE 16.16 ■ Sorter recursion and layout for 8 inputs.

Now that we have our merger, we can recursively unfold the pictorial specification in of the sorter layout to produce the design and layout in Figure 16.16 (for 8 inputs). This layout can be specified using the following combinators:

```
sortB cmp 1 = cmp
sorB cmp n
= two (sortB cmp (n-1)) >->
  pair >-> snD reverse >-> unpair >->
  butterfly cmp n
```

In the figure the description uses two subsorters to produce a bitonic input for a merger (shown on the right).

The 8-input description can be evaluated to produce an EDIF or VHDL netlist containing RLOC specifications for every gate. The FPGA layout of a degree-5 sorter (32 inputs) with 16-bit numbers is shown in Figure 16.17 on a Xilinx Virtex-II device. The resulting netlist is the same but with the layout information removed. It is shown in Figure 16.18. The netlist with the layout information leads to an implementation that is approximately 50 percent faster, and a 64-input sorter leads to a 75 percent speed improvement.

The case study just outlined shows how a complicated and recursive layout can be described in a feasible manner using algebraic layout combinators rather than explicit Cartesian coordinates.

16.4 LAYOUT VERIFICATION FOR PARAMETERIZED DESIGNS

A common problem with parameterized layout descriptions (especially those based on Cartesian coordinates) is that designer errors can produce bad layouts that cannot be realized on the target FPGAs—for example, the layout specification may try to map too many logic gates into the same location. Such errors

16.3.1 Case Study: Batcher's Bitonic Sorter

This section presents the layout specification of a high-speed parallel sorter that would have been difficult to lay out using explicit Cartesian coordinates. We show how to build complex structures incrementally by composing the layout of subcomponents using simple operators. The use of hierarchy achieves complex layout structures that would have been difficult or tedious to produce otherwise and impossible to produce in a compositional manner.

The objective is to build a parallel sorter from a parallel merger, as shown in Figure 16.11. A parallel merger takes two sublists of numbers where each sublist is sorted and produces a completely sorted list of numbers as its output. All inputs and outputs are shifted in, in parallel rather than serially. Furthermore, for performance reasons the sorter should have the same floorplan as shown in the figure.

This parallel sorter uses a two-sorter as its building block, which is shown fully placed in Figure 16.12. This circuit has left-to-right dataflow. Although the >=> combinator is also a serial composition combinator, it does not have any layout semantics because it is used to compose wiring circuits (which are not subject to layout directives).

The two-sorter in Figure 16.12 has been carefully designed to have a rectangular footprint because we will want to tile many of these circuits together vertically and horizontally to produce a compact and high-performance sorter network.

Another important combinator we will use in our sorter design is the two-combinator, which makes two copies of a circuit `r`, one of which works on the bottom half of the input and the other on the top half of the input, as illustrated in Figure 16.13. Furthermore, the second copy of `r` should be placed vertically on top of the first copy. The two combinator can be defined as

```
two r = halve >-> par [r,r] >-> unhalve
```

which says halve the input, use two copies of `r` in parallel (stacked vertically) on the halved input, and then take the result and unhalve it.

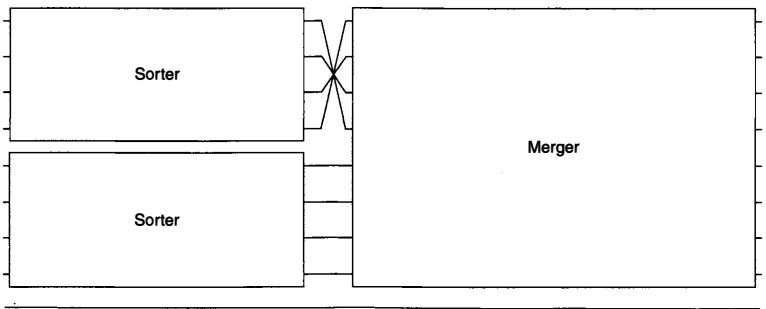


FIGURE 16.11 ■ The recursive structure of a sorter.

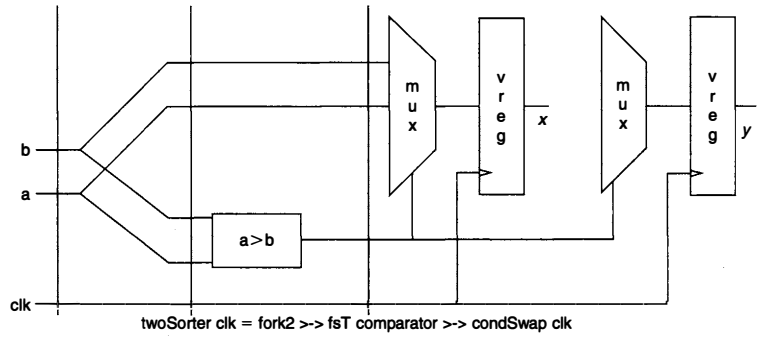


FIGURE 16.12 ■ Two-sorter layout and behavior specification.

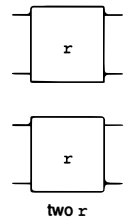


FIGURE 16.13 ■ The two-combinator.

Interleave (`ilv`) is another combining form that uses two copies of the same circuit. This combinator has the property that the bottom circuit processes the inputs at even positions and the top circuit processes the inputs at odd positions. It can be defined as

```
ilv r = unriffle >-> two r >-> riffle
```

An instance of `ilv r` for an 8-input bus is shown in Figure 16.14. The related `evens` combinator chops the input list into pairs and then applies copies of the same circuit to each input.

Given these ingredients, we can give a recursive description of a parallel merger butterfly circuit:

```
bfly r 1 = r
bfly r n = ilv (bfly r (n-1)) >-> evens r
```

A bitonic merger of degree 3 is shown in Figure 16.15, which not only describes how to compose the behavior of elements to form a merger circuit, but also

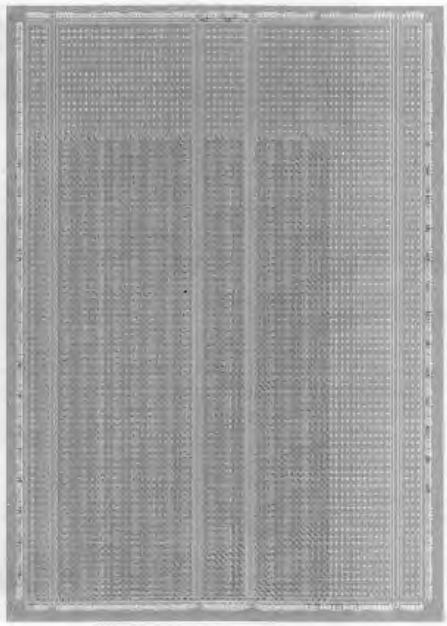


FIGURE 16.17 ■ The sorter FPGA layout (32 16-bit inputs).

make the production of IP cores that rely on layout very difficult and time consuming.

For a nonparameterized design, this is not much of an issue: The developer can check if the design maps, places, and routes. However, for a parameterized design it is usually impractical to check every possible combination of parameters to ensure that each one leads to a valid layout. A recent, interesting approach for layout verification involves theorem provers to statically analyze and formally verify that a design is free of layout errors. This is the approach taken by Pell [2] in his Quartz declarative block composition system, which uses a special hardware description notation that can be formally analyzed with the Isabelle theorem prover. The Quartz system works on algebraic layout combinators similar to those presented in the previous section.

The Quartz system verifies layout correctness by checking for validity, containment, and intersection. Validity ensures that the size function of a block always evaluates to a positive result. Containment ensures that for all parameter values all subblocks stay within the bounding box of the overall circuit. The intersection property checks for badly overlapping blocks.

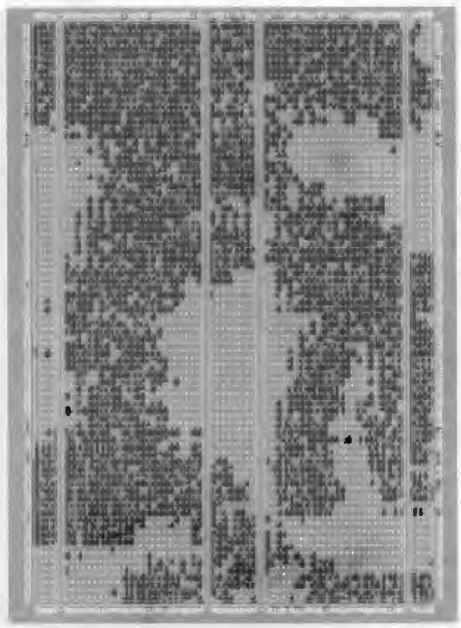


FIGURE 16.18 ■ The sorter with layout information removed.

16.5 SUMMARY

User specification of the layout of circuits for FPGAs is sometimes necessary to meet performance requirements, to reduce area, or to facilitate dynamic reconfiguration. While a user-defined layout is impractical for many complete designs because of complexity or time-to-market constraints, optimizing the most critical blocks of a circuit can have significant benefits, especially for reusable IP blocks and vendor libraries.

Some vendor tools provide the ability to specify the layout of gates or composite blocks through either absolute or relative Cartesian coordinates. However, these tools are tedious to use and error prone, particularly for parameterized circuits. Various systems have adopted algebraic layout specifications that use geometric relationships between blocks instead of coordinate values. Such descriptions work well for irregular and recursive layouts, as demonstrated by the recursive parallel sorter in this chapter. However, one may still specify illegal layouts for parameterized circuits, and no satisfactory technique exists for

finding them. A promising approach is the use of theorem provers to statically analyze algebraic layout descriptions to ensure that they have no layout errors for any given permutation of parameters.

References

[1] P. Bjesse, K. Claessen, M. Sheeran, S. Singh. Lava: Hardware design in Haskell. *International Conference on Functional Programming (ICFP)*, Springer-Verlag, 1998.

[2] O. Pell. Verification of FPGA layout generators in higher order logic. *Journal of Automated Reasoning* 37(1–2), August 2006.

[3] P. J. Roxby, S. Singh. Rapid construction of partial configuration datastreams from high level constructs using JBits. *Field Programmable Logic (FPL)*, Springer-Verlag, 2001.

[4] S. Singh. Death of the RLOC. *Field-Programmable Custom Computing Machines (FCCM)*, April 2000.

CHAPTER 17

PATHFINDER: A NEGOTIATION-BASED, PERFORMANCE-DRIVEN ROUTER FOR FPGAS

Larry McMurchie
Synplicity Corporation
Carl Ebeling
Department of Computer Science and Engineering
University of Washington

Routing is a crucial step in the mapping of circuits to field-programmable gate arrays (FPGAs). For large circuits that utilize many FPGA resources, it can be very difficult and time consuming to successfully route all of the signals. Additionally, the performance of the mapped circuit depends on routing critical and near-critical paths with minimum interconnect delays. One disadvantage of FPGAs is that they are slower than their ASIC counterparts, so it is important to squeeze out every possible nanosecond of delay in the routing.

The first goal, a complete routing of all signals, is difficult to achieve in FPGAs because of the hard constraints on routing resources. Unlike ASICs and printed circuit boards (PCBs), FPGAs have a fixed amount of interconnect. The usual approach in placement is to minimize the wiring resources anticipated for routing signals. Although this reduces the overall demand for resources, signals inevitably compete for the same resources during routing. The challenge is to find a way to allocate resources so that all signals can be routed. The second goal, minimizing delay, requires the use of minimum-delay routes for signals, which can be expensive in terms of routing resources, especially for high-fanout signals. Thus, the solution to the entire routing problem requires the simultaneous solution of two interacting and often competing subproblems.

Early solutions to the FPGA routing problem were based on the considerable literature on routing in the context of ASICs and gate arrays. The problem of routing FPGAs bears a considerable resemblance to the problem of global routing for custom integrated circuit design, where signals are assigned to channels. However, the two problems differ in several fundamental respects. First, routing resources in FPGAs are discrete and scarce while they are relatively continuous in custom integrated circuits (ICs). For this reason FPGAs require an integrated approach using both global and detailed routing. A second difference is that global routing for custom ICs is based on an undirected graph embedded in Cartesian space (i.e., a two-dimensional grid). In FPGAs the switches are often directional, and the routing resources connect arbitrary (but fixed) locations,

requiring a directed graph that may not be embedded in Cartesian space. Both of these distinctions are important, as they prevent direct application of much of the previous work in routing.

By far, the most common approach to global routing of custom ICs is a shortest-path algorithm with obstacle avoidance. By itself, this technique usually yields many unroutable nets that must be rerouted by hand. A plethora of rip-up and retry approaches have been proposed to remedy this deficiency [1–3]. The basic problem with rip-up and retry is that the success of a route is dependent not just on the choice of nets to reroute but also on the order in which rerouting is done. Delay is usually factored into the standard rip-up and retry approach by ordering the nets to be routed such that critical nets are routed most directly [4–6].

To make the FPGA routing problem tractable, nearly all of the routing schemes in the literature incorporate features of the underlying architecture. Palczewski [7] describes a maze router with rip-up and reroute targeting the Xilinx 4000 series. In this work the structure of the plane-parallel switchbox in the 4000 series is exploited in conjunction with an A* search. Brown et al. [4] employ an architecture model consisting of channels, switchboxes, connection matrices, and logic blocks. A global router balances channel densities and a detailed router generates families of explicit paths within channels to resolve congestion. These approaches, as well as others, obtain some of their success by exploiting the features of a particular architecture model. The problem is that new architectures become constrained by the restrictions of such existing routing algorithms.

17.1 THE HISTORY OF PATHFINDER

PathFinder was used initially in the development of the Triptych FPGA architecture [8–10]. In fact, Triptych, with its heavy reliance on effective placement and routing tools, was a catalyst for the development of the PathFinder algorithm—a perfect example of "necessity being the mother of invention." As part of an FPGA architecture exploration tool called Emerald [11], PathFinder was also employed in the development of an FPGA under development by IBM in the mid-1990s. This was particularly appropriate because PathFinder is inherently architecture independent. That experience showed that PathFinder was indeed an improvement over other FPGA routers available at the time.

The PathFinder algorithm was adopted and carefully implemented by Betz and Rose in the very popular versatile place and route (VPR) FPGA tool suite [12, 13], which has been widely used for academic and industry research. The Toronto place-and-route challenge [14] was established as a way to compare different FPGA placement and routing algorithms. Since the contest was established in 1997, the champion has been either VPR's implementation of PathFinder or SC-PathFinder, implemented at the University of California–Santa Cruz. Although companies are reluctant to divulge the details of their design tools, it is clear that some version of the PathFinder algorithm is currently used by virtually all commercial FPGA routers.

17.2 THE PATHFINDER ALGORITHM

17.2.1 The Circuit Graph Model

One of the key features of PathFinder is its architecture independence, which derives from the use of a simple underlying graph representation of FPGA architectures. This model allows PathFinder to be adapted to virtually any architecture and thus used to explore new architectures with very little startup cost. Once an architecture has been decided on, PathFinder can be specialized to it for improved results and performance.

The routing resources in an FPGA and their connections are represented by the directed graph $G = (V, E)$. The set of vertices V corresponds to the electrical nodes or wires in the FPGA architecture, and the edges E correspond to the switches that connect these nodes. An example of this graph model is shown in Figure 17.1 for a version of the Triptych FPGA cell. Note that devices are represented only implicitly by the wires connected to their terminals. That is, routing from one device terminal to another is routing between the wires connected to those terminals.

Associated with each node n in the architecture is a base cost b_n that represents the relative cost of using that node. This cost is typically proportional to the length of the wire, although other measures like capacitance or number of fanins and fanouts are also possible. Each node also has a delay d_n, which may or may not be the same as b_n.

Given a signal i in a circuit mapped onto the FPGA, the signal net N_i is the set of terminals, including the source terminal s_i and sinks t_{ij}. N_i forms a subset of V. A solution to the routing problem for signal i is the directed routing tree RT_i embedded in G and connecting the source s_i to all of its sinks t_{ij}.

17.2.2 A Negotiated Congestion Router

We assume that the reader is familiar with Djikstra's shortest-path graph algorithm [15–17], which is at the core of many routing algorithms. Note that in our formulation costs are associated with nodes, not edges. This changes the basic

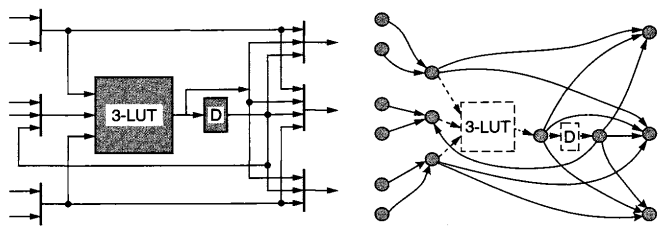


FIGURE 17.1 ■ The circuit for a Triptych FPGA cell is represented in PathFinder by the graph at the right.

shortest-path algorithm only slightly by redefining the cost of a path from node n_i to node n_j as the sum of the node costs along the path, including the starting and ending nodes.

Routing algorithms differ primarily in the cost function applied to the routing resources and in how individual applications of the shortest-path algorithm are used to successfully route all the signals of a netlist onto the graph representing the architecture. We ignore the issue of fanout in our initial presentation and assume that each signal is a simple route from source to a single sink.

A naive routing algorithm proceeds by applying the shortest-path algorithm to each signal in order, with the cost of a node defined as

$$c_n = b_n \tag{17.1}$$

Resources already used by previous routes are not available to later routes. It is clear that the order in which signals are routed is crucial, as later routes have many fewer available routing resources. Some algorithms perform rip-up and retry when later routes cannot find a path. Selected early routes that are blocking are ripped up and rerouted later—in essence, adaptively changing the order in which signals are routed.

The very simple example in Figure 17.2 shows how this naive algorithm can fail. There are three signals, 1, 2, and 3, to be routed from the sources S_1, S_2, and S_3 to their respective sinks D_1, D_2, and D_3. The ovals represent partial paths through one or more nodes, annotated with the associated costs. Ignoring congestion, the minimum-cost path for each signal would use node B. If the naive obstacle avoidance routing scheme is used, the order in which the signals are routed becomes crucial: Routing in the order 1, 2, 3 fails, and the minimum-cost routing solution will be found only when starting with signal 2.

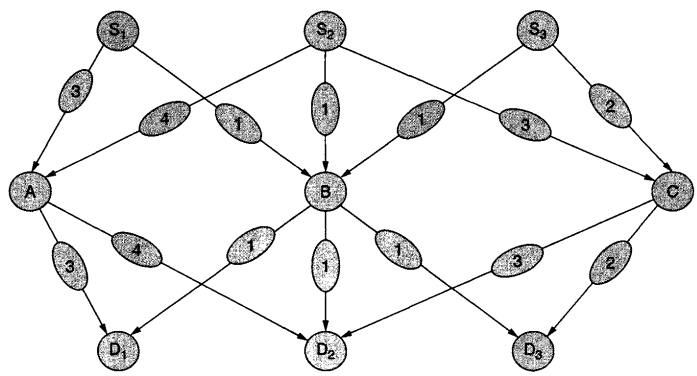


FIGURE 17.2 ■ First-order congestion.

This problem can be solved by introducing negotiated congestion avoidance, first suggested by Nair [18] by extending the cost of using a given node n in a route to

$$c_n = b_n \cdot p_n \tag{17.2}$$

where b_n is the base cost of using n, and p_n is a function of the number of other signals presently using n (p_n is often called the "present-sharing" term). Note that in the naive router, $p_n = 1$ if no other signals are using n, and infinity otherwise. In the negotiated congestion algorithm, p_n is set initially to 1 and all signals are routed. This allows each signal to be routed as if no other signals were present. The cost of sharing is then increased, and all nets are ripped up and rerouted in turn. This iterative process continues, with the cost of sharing increasing at each iteration until all signals have been successfully routed. The idea is that the cost of a congested node will increase and that signals that have other alternatives will eventually find other paths, leaving the node to the signal that needs it most. p_n is a function of the iteration i and the number of signals sharing a node k. The definition of p_n is a key tuning parameter of PathFinder.

The negotiated congestion avoidance algorithm solves the problem of Figure 17.2. During the first iteration, p_n is initialized to 1, and consequently no penalty is imposed for the use of n regardless of how many signals occupy it. Thus, in the first iteration all three signals share B. When the sharing function p_n increases sufficiently, signal 1 will find that a route through node A gives a lower cost than a route through the congested node B. During an even later iteration signal 3 will find that a route through node C gives a lower cost than that through B. This scheme of negotiation for routing resources depends on a relatively gradual increase in the cost of sharing nodes. If the increase is too abrupt, signals may be forced to take high-cost routes that lead to other congestion. Just as in the standard rip-up and retry scheme, the ordering becomes important.

While iterative negotiated congestion routing with the cost function of equation 17.2 can optimally route simple "first-order" routing problems like that in Figure 17.2, it fails on more complex "second-order" routing problems like that shown in Figure 17.3. Again we need to route three signals, one from each source to the corresponding sink. Let us first consider this example from the standpoint of obstacle avoidance with rip-up and retry. Assume that we start with the routing order (1, 2, 3). Signal 1 routes through node B, and signals 2 and 3 share node C. For rip-up and retry to succeed, both signals 1 and 2 would have to be rerouted, with signal 2 rerouted first. Because signal 1 does not use a congested node, determining that it needs to be rerouted is in general difficult.

This second-order congestion problem cannot be solved using p_n alone. Signal 2 will never choose node B because the present sharing costs for nodes B and C are the same, with B used by signal 1 and C used by signal 3. Since the path through C is cheaper, it is always chosen. PathFinder solves this by extending the cost function with a "history" term, h_n:

$$c_n = (b_n + h_n) \cdot p_n \tag{17.3}$$

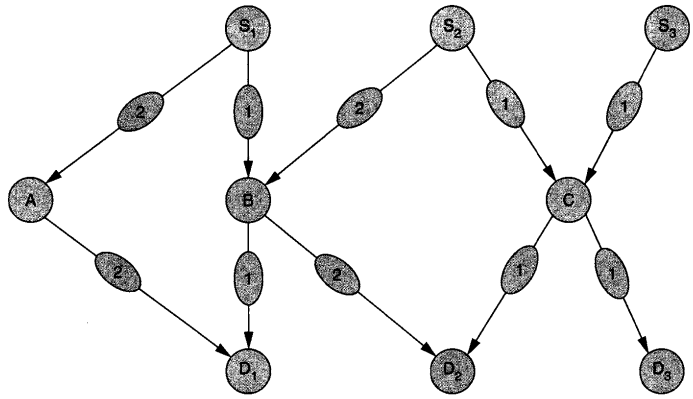


FIGURE 17.3 ■ Second-order congestion.

Unlike p_n, h_n "remembers" the congestion that has occurred on node n during previous routing iterations. That is, the history term is updated after each routing iteration; any node shared by multiple signals has its history term increased by some amount. The effect of h_n is to permanently increase the cost of using congested nodes so that routes through other nodes are attempted. Without this term, as soon as signals stop sharing a node, its cost drops to the base cost and it again becomes attractive. This leads to oscillations where signals switch back and forth between nodes but never resolve the congestion problem. The addition of the history term is a key difference between PathFinder and Nair's routing algorithm [18].

The term h_n allows the problem in Figure 17.3 to be routed successfully. On each iteration that node C is shared, h_n is increased slightly. When signal 2 switches to using node B, the cost of node C remains elevated. Now the history cost of node B rises because it is shared by signals 1 and 2. Eventually signal 1 will route through node A. Note that, depending on the base costs and how p_n and h_n are defined, signal 2 may switch back and forth between nodes B and C several times before the history costs of both are sufficiently high to force signal 1 onto node A.

The history term h_n is updated whenever a node n has shared signals. The size of δ_h, the amount by which h_n is increased, and how this depends on k, the number of sharing signals, are tunable parameters. If δ_h is too small, many iterations may be required to resolve the congestion; if it is too large, some solutions may not be found. Additionally, the relationship between p_n and h_n is very important. For example, it can be important to give the history term a chance to solve congestion before forcing the issue with p_n.

The details of the Negotiated Congestion algorithm are given in Figure 17.4. The while loop at line 2 executes the routing iterations until a solution has been

```
iteration ← 0                                                    1
While shared resources exist                                     2
    Iteration ← iteration + 1                                    3
    Loop over all signals i (signal router)                      4
        Rip up routing tree RT_i                                 5
        RT_i ← s_i                                               6
        Loop until all sinks t_ij have been found                7
            Initialize priority queue PQ to RT_i at cost 0       8
            Loop until new t_ij is found                         9
                Remove lowest cost node m from PQ                10
                Loop over fanouts n of node m                    11
                    Add n to PQ at cost P_im + c_n               12
                end loop                                         13
            end loop                                             14
            Loop over nodes n in path t_ij to s_i (backtrace)    15
                Update c_n                                       16
                Add n to RT_i                                    17
            end loop                                             18
        end loop                                                 19
    end loop                                                     20
    Loop over all nodes n_i shared by multiple signals           21
        h_i ← h_i + δ(k)                                         22
    end loop                                                     23
end while                                                        24
```

FIGURE 17.4 ■ Negotiated Congestion algorithm.

found. The signal router loop at line 4 iterates over all signals in the netlist, ripping up and rerouting the nets one at a time. The routing tree RT_i is the set of nodes used to route signal i. To reroute a signal, the routing tree is reset to be just the signal's source.

The priority queue is used to implement the breadth-first search of Djikstra's algorithm. At each iteration of the loop of line 9, the lowest-cost node is taken from the priority queue. It is generally best to order the nodes with the same cost according to when they were inserted into the queue, with the newest nodes being extracted first. The cost used when inserting a new node in the priority queue at line 12 is

$$P_{im} + c_n \tag{17.4}$$

where P_{im} is the cost of the current partial path from the source, and c_n is the cost of using node n.

A signal is routed one sink at a time using Djikstra's breadth-first algorithm. When the search finds a sink, the nodes on the path from the source to it are added to RT_i. This is done by back-tracing the search path to the source. The search is then restarted with the priority queue being initialized with all the nodes already in RT_i. In this way, all the nodes on routes to previously found sinks are used as potential sources for routes to subsequent sinks. This algorithm for constructing the routing tree is similar to Prim's algorithm for determining a minimum spanning tree over an undirected graph, and it is identical to one

suggested by Takahashi and Matsuyama [19] for constructing a tree embedded in an undirected graph. The quality of the points chosen by the algorithm is an open question for directed graphs; however, finding optimum (or even near-optimum) points is not essential for the router to be successful in adjusting costs to eliminate congestion.

The VPR router [12] reduces the cost of reinitializing the priority queue for each fanout by observing that for large-fanout nets, most of the paths found in searching for the previous fanout remain valid, especially if the segment added to the routing tree is relatively small. Thus, the search continues from the previous state after the new segment has been added to the routing tree. Because of the way Djikstra's algorithm ignores nodes after they have been visited once, this optimization must be implemented carefully to avoid expensive routing trees for high-fanout nets. Other algorithms for forming the fanout tree are possible. For example, there are times when routing to the most distant sink first results in a better routing tree.

At the end of each iteration, the history cost of each node shared by multiple signals is updated. The δ added to the history cost is generally a function of k, the number of signals sharing the node.

17.2.3 The Negotiated Congestion/Delay Router

To introduce delay into the Negotiated Congestion algorithm, we redefine the cost of using node n when routing a signal from s_i to t_{ij} as

$$C_n = A_{ij} d_n + (1 - A_{ij}) c_n \tag{17.5}$$

where c_n is defined in equation 17.3 and A_{ij} is the slack ratio:

$$A_{ij} = D_{ij} / D_{\max} \tag{17.6}$$

where D_{ij} is the delay of the longest delay (register–register) path containing the signal segment (s_i, t_{ij}), and $D_{\max}$ is the maximum delay over all paths (i.e., the critical-path delay). Thus, $0 < A_{ij} \leq 1$. (This standard definition of slack ratio is easily extended to include circuit inputs and outputs with timing constraints as well as circuits with multiple clocks.)

Because path delay is made up of both device and wire delay, and the router can only control the wire delay, a more accurate formulation for A_{ij} is

$$A_{ij} = (D_{ij} - Ddev_{ij}) / (D_{\max} - Ddev_{ij}) \tag{17.7}$$

where $Ddev_{ij}$ is the path delay from node i to node j attributable to devices, and $D_{ij} - Ddev_{ij}$ is thus the wire delay on the path from node i to node j. With equation 17.7, paths with the same path delay but greater wire delay pay more attention to delay and less to congestion.

The first term of equation 17.5 is the delay-sensitive term; the second term is congestion sensitive. Equations 17.5, 17.6, and 17.7 are the keys to providing the appropriate mix of minimum-cost and minimum-delay trees. If a particular source/sink pair lies on the critical-path, then $A_{ij} = 1$ and the cost of node n

is just the delay term; hence a minimum-delay route is used and congestion is ignored. In practice, A_{ij} is limited to a maximum value such as 0.9 or 0.95 so that congestion is not completely ignored. If a source/sink pair belongs to a path whose delay is much smaller than the critical-path, then A_{ij} is small and the congestion term dominates, resulting in a route that avoids congestion at the expense of extra delay.

To accommodate delay, the basic Negotiated Congestion algorithm of Figure 17.4 is changed as follows. For the first iteration, all A_{ij} are initialized to 1 and minimum-delay routes are found for every signal. This yields the smallest possible critical-path delay. All A_{ij} are recomputed after every routing iteration using the critical-path delay and the delays incurred by signals on that iteration.

The sinks of each signal are now routed in decreasing A_{ij} order. This allows the most timing-constrained sinks to determine the coarse structure of the routing tree with no interference from less constrained paths.

The priority queue (line 8 in Figure 17.4) is initialized by inserting each node of RT_i with the cost $A_{ij} \sum_k d_k$, where the n_k are nodes on the path from the source n_i to node n_j. This initializes the nodes already in the partial routing tree with the weighted path delay from the source.

The router completes when no more shared resources exist. Note that by recalculating all A_{ij}, we have kept a tight rein on the critical-path. Over the course of the routing iterations, the critical-path increases only to the extent required to resolve congestion. This approach is fundamentally different from other schemes [4, 5] that attempt to resolve congestion first and then reduce delay by rerouting critical nets.

The PathFinder algorithm is particularly powerful for asymmetric architectures that have a range of slow and fast wires. By making the slower wires lower cost, the negotiation algorithm automatically assigns critical signals to the fast wires as needed and noncritical signals to the slow wires.

17.2.4 Applying A* to PathFinder

Djikstra's shortest-path algorithm performs an expensive breadth-first search of the graph. This search has an $O(n^2)$ running time for two-dimensional circuit structures, where n is the length of the path. The A* heuristic [20] is a technique that uses additional information about the cost of paths in the graph to bound the size of the search. The cost of a partial path becomes the cost of the partial path plus the estimated cost from the end of the partial path to the destination. If this estimated cost is a lower bound on the actual cost, then the search will provide an optimal solution. If the estimated cost is accurate, then the search becomes a depth-first search with $O(n)$ running time.

In applying A* to PathFinder, both the cost and the delay of paths in the graph must be estimated. We modify equation 17.4 as follows:

$$C_n = P_{im} + A_{ij}(d_n + Dest_{nj}) + (1 - A_{ij})(c_n + Cest_{nj}) \tag{17.8}$$

where $Dest_{nj}$ and $Cest_{nj}$ are the estimated delay and cost, respectively, of the minimum-delay route from n to sink j.

To use the A* heuristic, the router must know the destination in order to determine the estimated cost. Instead of letting the breadth-first router find the closest destination when there are multiple fanouts, the path length estimates are used to sort the fanouts from closest to furthest and the routing is performed in this order.

In many FPGAs, such as those that are standard island style, the cost and delay of routes can be estimated based on the locations of the source and destination using the geometry of the layout. A more general and accurate method is to use the shortest-path algorithm to create a complete "distance table" that contains the cost estimate of the minimum-delay route from every node to all potential sinks. This is only feasible, however, for relatively small architectures or for coarse-grained architectures that have many fewer nodes than fine-grained FPGAs. To reduce the table size, clustering can be used and estimates stored for the cost/delay between clusters [21]. If the cost/delay between two clusters is taken as the minimum cost/delay between any two nodes in the two clusters, it represents a true lower bound. Clustering has been reported to reduce the size of the distance table by a factor of 100 while slowing the search only by a factor of 2 [21].

In the early iterations of PathFinder, when sharing is ignored, the full advantage of A* is obtained. That is, if the cost/delay estimates are accurate, a depth-first search is achieved. As the cost of sharing rises, however, the cost estimates, which do not include the sharing costs, become less and less accurate and the search becomes less efficient.

In experiments with PathFinder and A*, Swartz et al. [22] used a multiplicative direction factor α to inflate the path estimate. In effect, α determines how aggressively the router drives toward the target sink. An α of 1.0 corresponds to true A* and is guaranteed to find the shortest source/sink connection. Swartz et al. determined that an α of 1.5 gave the best results for large circuits, with no measurable degradation in the quality of the resulting routing. However, note that the cost function had only a congestion term and no delay term. Tessier also experimented with accelerating routing with even more aggressive use of the A* search [23, 24].

17.3 ENHANCEMENTS AND EXTENSIONS TO PATHFINDER

Many research papers have discussed extensions and optimizations of the PathFinder algorithm. First and foremost is the work by Betz and Rose on VPR [12], which for the past eight years has been a widely used vehicle for academic and industrial research into FPGA architectures and CAD. We discuss here some of the more salient ideas that have been applied to PathFinder.

17.3.1 Incremental Rerouting

A common optimization suggested in the original PathFinder paper [8] is to limit the rip-up and rerouting of signals in an iteration only to those that use shared resources. Intuitively, this reduces the amount of "wasted" effort that

goes into rerouting signals that always take the same path. The argument is that if a signal does not use a shared resource, it will take the same path as it did before, because history costs can only rise and thus no other path can become cheaper. This argument fails where p_n becomes smaller as sharing signals reroute around a congested node. Experience shows that this optimization increases the number of routing iterations, but reduces the total running time substantially, with negligible impact on the quality of the solution found.

17.3.2 The Cost Function

There are many ways to tune PathFinder for specific architectures or to achieve specific goals. Many variations of the cost function have been described that change how the three cost terms b_n, p_n, and h_n are computed and combined. The essential feature of the cost function is that h_n is a function of the history of the congestion of the node and that p_n is a function of the current congestion. The rates at which h_n and p_n increase can be tuned; increasing them quickly, for example, decreases the number of iterations required but also decreases the quality of the solution. The history term may include a decay function on the assumption that the more recent history is more valid than the distant past. This is particularly important when PathFinder is used in an integrated place-and-route tool [21, 25].

The PathFinder cost function can also be modified to include both short-path and long-path delay terms [26]. For long paths, delay is minimized by using the PathFinder cost function. For short paths, however, the cost function is changed to find a path with a target delay, not the minimum delay. This changes the underlying shortest-path problem considerably and requires an accurate "look-ahead" function that predicts the remaining delay to the destination so that the router can opportunistically add the appropriate extra delay.

17.3.3 Resource Cost

Determining the base cost of routing resources is harder than it appears. The shortest-path algorithm attempts to minimize the total cost of a solution, so minimizing the cost should also minimize congestion. The typical cost function used by routers is the length of the wire, which is a good heuristic for typical architectures where the number of available wires is inversely proportional to their individual lengths. A better heuristic is to base the cost of a wire on the expected routing demand for it. This can be approximated by routing a set of placed benchmarks onto an architecture and measuring wire by wire the routing demand. Another method is to perform a large number of random routes using a typical Rent's wirelength distribution through the architecture and again measuring the overall use of each wire. In this formulation, wire costs are initialized to 1, raised à la PathFinder according to wire usage, and converge to some constant value.

Delay is an approximation that is often used for cost as it is typically closely related to wirelength and relative demand. It also simplifies the cost function for the integrated congestion and delay router.

17.3.4 The Relationship of PathFinder to Lagrangian Relaxation

The PathFinder algorithm is very similar to Lagrangian relaxation for finding an optimal routing subject to congestion and delay constraints [27–29]. In Lagrangian relaxation, the constraints are relaxed by multiplying them by a vector of Lagrangian multipliers and adding them to the objective function to be minimized. The solution to a Lagrangian formulation with a specific set of Lagrangian multipliers provides an approximate solution to the original minimization problem. An iterative procedure that modifies the Lagrangian multipliers is used to find increasingly better solutions. A subgradient method is used to update the multipliers. Intuitively, the multipliers are increased or decreased depending on the extent to which the corresponding constraint is satisfied.

A Lagrangian relaxation method proceeds somewhat differently from the PathFinder algorithm. The multipliers operate much like PathFinder's history term, but there is no corresponding present-sharing term p_n. While the history term is monotonically nondecreasing, the Lagrangian multipliers can both increase and decrease depending on how well the corresponding constraint is satisfied. The amount by which the multipliers are adjusted in Lagrangian relaxation is also decreased with each iteration.

17.3.5 Circuit Graph Extensions

The simple circuit graph model is very general, but there are some specific circuit structures that require extensions. This section describes some solutions for these.

Symmetric device inputs

Lookup tables (LUTs) are the prime example of FPGA devices whose pins are "permutable." That is, the inputs to a LUT can be swapped arbitrarily by permuting the table's contents. Other devices like adders also have symmetric inputs. In the simple graph model, a signal is routed to a specific input terminal and there is no way to specify a route to one of a set of terminals.

Symmetric inputs are easily accommodated in the graph model by adding "pseudo-multiplexers" on the inputs of the LUT. These are shown as dashed nodes at the top of Figure 17.5. Signal sinks can be arbitrarily assigned to the LUT inputs and routed in the usual way. After the routing solution has been found, the pseudo-multiplexers are removed and implemented "virtually" by permuting the LUT table contents appropriately. In the example of Figure 17.5, the signals a, b, and c are routed to the LUT inputs A, B, and C, respectively, using the pseudo-multiplexers as shown with bold lines. This routing is then used to permute the LUT inputs as shown on the right by modifying the LUT contents.

De-multiplexers

A de-multiplexer is a device that can connect its input to at most one of several outputs. Each output connection is represented as an edge in the circuit graph shown in Figure 17.6. Wire fanout, of course, is not constrained, and there is no way in the graph model to specify a constraint on the number of fanouts that can be used. This case is handled by a special counter that counts the number of the edges that are used. If more than one edge is being used, the

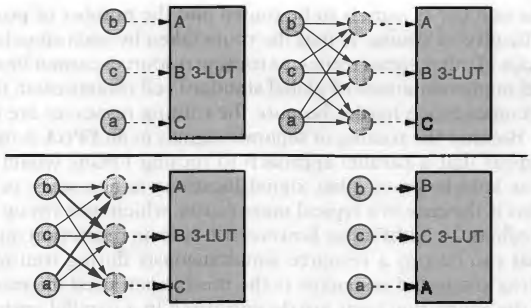


FIGURE 17.5 ■ Symmetric device inputs are handled by inserting pseudo-multiplexers.

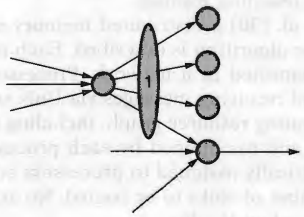


FIGURE 17.6 ■ De-multiplexers are handled by negotiating for the fanouts of the de-multiplexer.

de-multiplexer is being shared in much the same way that wires can be shared by signals. A PathFinder cost function can be applied with both a sharing and a history component so that the single fanout used is determined by means of negotiation.

Bidirectional switches

Edges in the graph model, which represent connections, are directional. This models multiplexer-based architectures directly. Transistors that are often used to construct configurable interconnects are bidirectional. These bidirectional switches simply translate to two directional edges in the graph. The router uses at most one of the edges, which induces a logical direction on the switch. That is, when a switch is turned on in a configuration, it is being driven by an output from one side to the other.

17.4 PARALLEL PATHFINDER

A typical large FPGA design has many thousands of signals. If separate signals could be routed in parallel, the degree of parallelism would be limited only by

the number of signals to be routed and the number of processors available. The difficulty, of course, is that the route taken by each signal depends on the knowledge of other signal routes, as routing resources cannot be shared. Although parallel implementations of global standard cell routers exist, the problem for FPGAs becomes much harder because the routing resources are discrete and fixed.

Because the routing of separate signals in an FPGA is tightly coupled, it might appear that a parallel approach to routing FPGAs would not be possible given that knowledge of other signal locations is necessary to find a feasible route. This is the case in a typical maze router, which uses rip-up and reroute to resolve conflicts. In PathFinder, however, there is no restriction on the number of signals that can occupy a resource simultaneously during routing. Instead, the cost of using congested resources is the mechanism used to resolve resource conflicts. If the congestion costs are decentralized in a parallel environment, the concerns are how and when they will be updated and whether the update method will be acceptable in terms of the number of processors effectively utilized and the quality of the resulting routing.

In Chan et al. [30] a distributed memory multiprocessor implementation of the PathFinder algorithm is described. Each processor has a private local memory and is connected in a network. Processors communicate with each other by sending and receiving messages via Unix socket communication. A complete copy of the routing resource graph, including first- and second-order congestion costs, is kept and maintained by each processor. The signals in a netlist to be routed are statically assigned to processors such that each processor has about the same number of sinks to be routed. No attempt is made to assign signals to processors based on locality.

Processors route signals asynchronously and thus communicate updated congestion costs asynchronously. There is no guarantee of the order or the timing of the arrival of such congestion cost updates, resulting in a source of indeterminism. Processors are allowed to proceed to successive iterations without waiting for others, although a limit of a few iterations of separation is generally employed.

It is conceded that, because of latency, this parallel routing algorithm may not converge. Imagine a scenario in which two signals being routed by two different processors vie for the same resource. Message latency or merely concurrency may cause the two signals to oscillate between routing iterations, because each processor knows where the other processor's signal was in the last iteration but not in the current one. Such cases generally occur during the last iterations of a route. At that point, Chan and colleagues [30] reduce the multiprocessor implementation to a single-processor implementation in order to resolve the congestion.

This parallel implementation was tested on a set of benchmarks ranging from 118 to 1542 signal nets on the Xilinx 4000 architecture. Speedups ranged from 1.6 to 2.2 times for two processors and 2.3 to 3.8 times for four processors. For nearly all benchmarks, no additional speedups are obtained for more than four processors. The performance of the benchmarks (in terms of delay or clock rate) was shown to vary minimally with increasing numbers of processors.

This initial implementation of a parallel form of PathFinder is significant in that it demonstrates appreciable speedups while employing a rather simple computational framework. Because of the inherent approximations of congestion cost and its gradual increase, PathFinder exhibits good qualities for parallelism in a framework where congestion costs are communicated asynchronously, as they become available. It may result (as shown by Chan et al. [30]) in an increased number of iterations to converge, but is able to employ more multiple loosely connected processors to good advantage.

17.5 OTHER APPLICATIONS OF THE PATHFINDER ALGORITHM

PathFinder has been used to incrementally reroute signals around faults in cluster-based FPGAs [31]. This rerouting uses the accumulated history costs acquired by the initial routing to quickly find a new routing solution when nodes and edges in the circuit graph have been removed because of faults.

QuickRoute [32] extends PathFinder to handle pipelined routing structures. The key idea in QuickRoute is to change Djikstra's shortest-path algorithm to allow nodes to be visited more than once, by paths with different latencies. This causes many more overlapping paths to be explored, but the negotiated congestion avoidance of PathFinder still performs well.

Several groups have applied PathFinder to the problem of scheduling the communication in computing graphs to coarse-grained architectures or multiprocessors [33–35]. In this application of PathFinder, the routing becomes a space–time problem.

17.6 SUMMARY

The widespread use of PathFinder by commercial FPGA routers and university research efforts alike is a testimonial to its robustness.

Several key facets of the algorithm make it attractive. However, its primary advantage is the iterative nature of resolving congestion, using both current as well as historical resource use in the formulation of the cost function. By very gradually increasing cost due to both usages, the routing search space is thoroughly explored. Routing with other objective functions, delay in particular, is easily integrated into the cost function. A primary feature implicit in PathFinder (that distinguishes it from previous efforts) is the allowance of nonphysically feasible intermediate states—for example, shared resources—while converging to a physically feasible final state. Finally, by being grounded in a directed graph representation, PathFinder is very adaptable to changing FPGA architectures as well as other problems that can be abstracted to a directed graph.

In the future we see the routing problem as being an increasingly dominant hurdle in the use of FPGAs with millions of resources. To reduce the runtime, more investigation will be required to effectively parallelize PathFinder, making

use of additional computational resources. Given the growing focus on other objectives such as power consumption, it is likely that we will see experimentation with other cost function formulations as well.

Acknowledgments We wish to thank Gaetano Borriello for initial discussions about routing when PathFinder was being applied to the Triptych architecture, and Steven Yee for his help in constructing detailed descriptions of the Xilinx architectures. We also thank Pak Chan and Martine Schlag for sharing the results on parallel PathFinder.

References

[1] W. A. Dees, R. J. Smith. Performance of interconnection rip-up and reroute strategies. *Design Automation Conference*, 1981.

[2] R. Linsker. An iterative-improvement penalty-function-driven wire routing system. *IBM J. Res. Development* 28(5), 1984.

[3] J. Cohn, D. Garrod, R. Rutenbar, L. Carley. Koan/anagram II: New tools for device-level analog placement and routing. *IEEE Journal of Solid-State Circuits* 26(3), 1991.

[4] S. Brown, J. Rose, Z. Vranesic. A detailed router for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11(5), 1992.

[5] J. Frankle. Iterative and adaptive slack allocation for performance-driven layout and FPGA routing. *Design Automation Conference*, 1992.

[6] M. J. Alexander, J. P. Cohoon, J. L. Ganley, G. Robins. An architecture-independent approach to FPGA routing based on multi-weighted graphs. *Proceedings of the Conference on European Design Automation*, 1994.

[7] M. Palczewski. Plane parallel a maze router and its application to FPGAs. *Design Automation Conference*, 1992.

[8] L. McMurchie, C. Ebeling. A negotiation-based performance-driven router for FPGAs. *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays Aided Design*, 1995.

[9] G. Borriello, C. Ebeling, S. Hauck, S. Burns. The triptych FPGA architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3(4), 1995.

[10] C. Ebeling, L. McMurchie, S. Hauck, S. Burns. Placement and routing tools for the triptych FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3(4), 1995.

[11] D. C. Cronquist, L. McMurchie. Emerald: An architecture-driven tool compiler for FPGAs. *Proceedings of the Fourth ACM International Symposium on Field-Programmable Gate Arrays*, 1996.

[12] V. Betz, J. Rose. VPR: A new packing, placement and routing tool for FPGA research. *Proceedings of the Seventh International Workshop on Field-Programmable Logic and Applications*. Springer-Verlag, 1997.

[13] V. Betz, J. Rose, A. Marquardt. *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic, 1999.

[14] V. Betz. The FPGA place-and-route challenge (*www.eecg.toronto.edu/vaughn/challenge/challenge.html*).

[15] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik* 1(1), December 1959.

[16] E. Moore. The shortest path through a maze. *International Symposium on the Theory of Switching*, April 1959.
[17] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers* 10, September 1961.
[18] R. Nair. A simple yet effective technique for global wiring. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6(2), 1987.
[19] H. Takahashi, A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Math. Japonica* 24(6), 1980.
[20] P. Hart, N. Nilsson, B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 1968.
[21] A. Sharma. *Place and Route Techniques for FPGA Architecture Advancement*, Ph.D. thesis, University of Washington, 2005.
[22] J. S. Swartz, V. Betz, J. Rose. A fast routability-driven router for FPGAs. *Proceedings of the ACM/SIGDA Ssixth International Symposium on Field-Programmable Gate Arrays*, 1998.
[23] R. G. Tessier. Negotiated A* routing for FPGAs. *Fifth Canadian Workshop on Field-Programmable Logic*, 1998.
[24] R. G. Tessier. *Fast Place and Route Approaches for FPGAs*, Ph.D. thesis, MIT, 1999.
[25] A. Sharma, S. Hauck, C. Ebeling. Architecture-adaptive routability-driven placement for FPGAs. *International Conference on Field-Programmable Logic and Applications*, 2005.
[26] R. Fung, V. Betz, W. Chow. Simultaneous short-path and long-path timing optimization for FPGAs. *IEEE/ACM International Conference on Computer Aided Design*, 2004.
[27] S. Lee, Y. Cheon, M. D. F. Wong. A min-cost flow based detailed router for FPGAs. *International Conference on Computer-Aided Design*, 2003.
[28] S. Lee, M. Wong. Timing-driven routing for FPGAs based on Lagrangian relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22(4), 2003.
[29] M. M. Ozdal, M. D. F. Wong. Simultaneous escape routing and layer assignment for dense PCBs. *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, 2004.
[30] P. K. Chan, M. D. F. Schlag, C. Ebeling, L. McMurchie. Distributed-memory parallel routing for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19(8), August 2000.
[31] V. Lakamraju, R. Tessier. Tolerating operational faults in cluster-based FPGAs. *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field-Programmable Gate Arrays*, 2000.
[32] S. Li, C. Ebeling. QuickRoute: A fast routing algorithm for pipelined architectures. *IEEE International Conference on Field-Programmable Technology*, 2004.
[33] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *Design, Automation and Test in Europe*, 2003.
[34] J. Cook, L. Baugh, D. Gottlieb, N. Carter. Mapping computation kernels to clustered programmable reconfigurable processors. *IEEE International Conference on Field-Programmable Technology*, 2003.
[35] L.-Y. Lin, C.-Y. Wang, P.-J. Huang, C.-C. Chou, J.-Y. Jou. Communication-driven task binding for multiprocessor with latency insensitive network-on-chip. *Proceedings of the 2005 Conference on Asia South Pacific Design Automation*, 2005.

CHAPTER 18

RETIMING, REPIPELINING, AND *C*-SLOW RETIMING

Nicholas Weaver
International Computer Science Institute

Although pipelining is a huge benefit in field-programmable gate array (FPGA) designs, and may be required on some FPGA fabrics [5, 10, 12], it is often difficult for a designer to manage and balance pipeline stages and to insert the necessary delays to meet design requirements.

Leiserson et al. [4] were the first to propose *retiming*, an automatic process to relocate pipeline stages to balance a design. Their algorithm, in $O(n^2 lg(n))$ time, can rebalance a design so that the critical path is optimally pipelined. In addition, two modifications, *repipelining* and *C-slow retiming*, can add additional pipeline stages to a design to further improve the critical path.

The key idea is simple: If the number of registers around every cycle in the design does not change, the end-to-end symantics do not change. Thus, retiming attempts to solve two primary constraints: All paths longer than the desired critical path are registered, and the number of registers around every cycle is unchanged.

This optimization is useful for conventional FPGAs but absolutely essential for fixed-frequency FPGA architectures, which are devices that contain large numbers of registers and are designed to operate at a fixed, but very high, frequency, often by pipelining the interconnect as well as the computation.

To meet the array's fixed frequency, a design must ensure that every path is properly registered. Repipelining or *C*-slow retiming enables a design to be transformed to meet this constraint. Without automated repipelining or *C*-slow retiming, the designer must manually ensure that all pipeline constraints are met by the design.

Retiming operates by determining an optimal placement for existing registers, while repipelining and *C*-slowing add registers before the retiming process begins. After retiming, the design should be optimally (or near-optimally) balanced, with no pipeline stage requiring significantly more time than any other stage.

Section 18.1 describes the basic retiming operation and the retiming algorithm and its semantics. Then Section 18.2 discusses repipelining and *C*-slowing: two different techniques for adding registers. Repipelining improves feedforward designs by adding additional pipelining stages, while *C*-slowing creates

an interleaved design by replacing every register with a sequence of C registers. Both of these transformations increase throughput but also increase latency.

Section 18.3 surveys the various implementations, beginning with Leiserson's original algorithm and concluding with both academic and commercial tools. Section 18.4 discusses implementing retiming for fixed-frequency arrays. Unlike general FPGAs, fixed-frequency FPGAs require retiming in order to match user designs with architectural constraints. Finally, Section 18.5 discusses an interesting side effect of *C*-slowing: the creation of interleaved, multi-threaded architectures. We conclude in Section 18.6 with a discussion of the reasons that retiming is not a ubiquitous optimization in FPGA tool flows.

18.1 RETIMING: CONCEPTS, ALGORITHM, AND RESTRICTIONS

The goal of retiming is to move the pipeline registers in a design into the optimal position. Figure 18.1 shows a trivial example. In this design, the nodes represent logic delays (a), with the inputs and outputs passing through mandatory, fixed registers. The critical path is 5, and the input and output registers cannot be moved. Figure 18.1(b) shows the same graph after retiming. The critical path is reduced from 5 to 4, but the I/O semantics have not changed, as three cycles are still required for a datum to proceed from input to output.

As can be seen, the initial design has a critical path of 5 between the internal register and the output. If the internal register could be moved forward, the critical path would be shortened to 4. However, the feedback loop would then be incorrect. Thus, in addition to moving the register forward, another register would need to be added to the feedback loop, resulting in the final design.

Additionally, even if the last node is removed, it could never have a critical path lower than 4 because of the feedback loop. There is no mechanism that can reduce the critical path of a single-cycle feedback loop by moving registers: Only additional registers can speed such a design.

Retiming's objective is to automate this process: For a graph representing a circuit, with combinational delays as nodes and integer weights on the edges, find a new assignment of edge weights that meets a targeted critical path or fail if the critical path cannot be met. Leiserson's retiming algorithm is guaranteed to find such an assignment, if it exists, that both minimizes the critical path and ensures that around every loop in the design the number of registers always remains the same. It is this second constraint, ensuring that all feedback loops

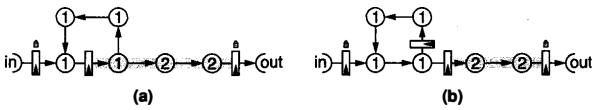


FIGURE 18.1 ■ A small graph before retiming (a) and the same graph after retiming (b).

TABLE 18.1 ■ The constraint system used by the retiming procsess

Condition normal edge from $u \rightarrow v$	Constraint $r(u) - r(v) \leq w(e)$
Edge from $u \rightarrow v$ must be registered	$r(u) - r(v) \leq w(e) - 1$
Edge from $u \rightarrow v$ can never be registered	$r(u) - r(v) \leq 0$ and $r(v) - r(u) \leq 0$
Critical paths must be registered	$r(u) - r(v) \leq W(u, v) - 1$ for all u, v such that $D(u, v) > P$

are unchanged, which ensures that retiming doesn't change the semantics of the circuit. In Table 18.1, $r(u)$ is the lag computed for each node (which is used to determine the final number of registers on each edge), $w(e)$ is the initial number of registers on an edge, $W(u,v)$ is the minimum number of registers between u and v, and $D(u,v)$ is the critical path between u and v.

Leiserson's algorithm takes the graph as input and then adds an additional node representing the external world, with appropriate edges added to account for all I/Os. This additional node is necessary to ensure that the circuit's global I/O semantics are unchanged by retiming.

Two matrices are then calculated, W and D, that represent the number of registers and critical path between every pair of nodes in the graph. These matrices are necessary because retiming operates by ensuring that at least one register exists on every path that is longer than the critical path in the design.

Each node also has a lag value r that is calculated by the algorithm and used to change the number of registers that will be placed on any given edge. Conventional retiming does not change the design semantics: All input and output timings remain unchanged while minor design constraints are imposed on the use of FPGA features. More details and formal proofs of correctness can be found in Leiserson's original paper [4].

The algorithm works as follows:

1. Start with the circuit as a directed graph. Every node represents a computational element, with each element having a computational delay. Each edge can have zero or more registers as a weight w. Add an additional dummy node with 0 delay, with an edge from every output and to every input. This additional node is to ensure that from every input to every output the number of registers is unchanged and therefore the data input to output timing is unaffected.

2. Calculate W and D. D is the critical path for every node to every other node, and W is the initial number of registers along this path. This requires solving the all-pairs shortest-path problem, of which the optimal algorithm, by Dijkstra, requires $O(n^2 lg(n))$ time. This dominates the asymptotic running time of the algorithm.

3. Choose a target critical path and create the constraints, as summarized in Table 18.1. Each node has a lag value r, which will eventially specify the *change* in the number of registers between each node. Initialize all nodes to have a lag of 0.

4. Since all constraints are pairwise integer inequalities, the Bellman–Ford constraint solver is guaranteed to find a solution if one exists or to terminate if not. The Bellman–Ford algorithm performs N iterations (N = the number of constraints to solve). In each iteration, every constraint is examined. If a constraint is already satisified, nothing happens. Otherwise, $r(u)$ or $r(v)$ is decremented to meet the particular constraint. Once an iteration occurs where no values change, the algorithm has found a solution. If there is no solution, after N iterations the algorithm terminates with a failure.

5. If the constraint solver fails to find a solution, or a tighter critical path is desired, choose a new critical path and return to step 3.

6. With the final set of constraints, a new set of registers is constructed for each edge, $w' \cdot w'(e) = w(e) - r(u) + r(v)$.

A graphical example of the algorithm's results is shown in Figure 18.1. The initial graph has a critical path of 5, which is clearly nonoptimal. After retiming, the graph has a critical path of 4, but the I/O semantics have not changed, as any input will still require three cycles to affect the output. To determine whether a critical path P can be achieved, the retiming algorithm creates a series of constraints to calculate the lag on each node (Table 18.1).

The primary constraints ensure correctness: No edge will have a negative number of registers, while every cycle will always contain the original number of registers. All I/O passes through the intermediate node, ensuring that input and output timings do not change. These constraints can be modified so that a particular line will contain no registers, or a mandatory minimum number of registers, to meet architectural constraints without changing the complexity of the equations. But it is the final constraint, that all critical paths above a predetermined delay P are registered, that gives this optimization its effectiveness.

If the constraint system has a solution, the new lag assignments for all nodes will allocate registers properly to meet the critical path P. But if there is no solution, there cannot be an assignment of registers that meets P. Thus, the common usage is to find the minimum P where the constraints are all met.

In general, multiple constraint-solving attempts are made to search for the minimum critical path P. The constraints for P are the final retimed design. There are two ways to speed up this process. First, if the Bellman–Ford algorithm can find a solution, it usually converges very quickly. Thus, if there is no solution that satisfies P, it is usually effective to abandon the Bellman–Ford algorithm early after $0.1N$ iterations rather than N iterations. This seems to have no impact on the quality of results, yet it can greatly speed up searching for the minimum P that can be satisfied in the design.

A second optimization is to use the last computed set of constraints as a starting point. In conventional retiming, the Bellman–Ford process is invoked multiple times to find the lowest satisfiable critical path. In contrast, fixed-frequency repipelining or C-slow retiming uses Bellman–Ford to discover the minimum number of additional registers needed to satisfy the constraints. In both cases,

keeping the last failed or successful solution in the data structure provides a starting point that can significantly speed up the process if a solution exists.

Retiming in this way imposes only minimal design limitations: Because it applies only to synchronous circuits, there can be no asynchronous resets or similar elements. A synchronous global reset imposes too many constraints to allow effective retiming. Local synchronous resets and enables only produce small, self loops that have no effect on the correct operation of the algorithm.

Most other design features can be accommodated simply by adding appropriate constraints. For example, an FPGA with a tristate bus cannot have registers placed on this bus. A constraint that says that all edges crossing the bus can never be registered ($r(u) - r(v) \leq 0$ and $r(v) - r(u) \leq 0$) ensures this. Likewise, an embedded memory with a mandatory output flip-flop can have a constraint ($r(u) - r(v) \leq w(e) - 1$) that ensures that at least one register is placed on this output.

Memories themselves can be retimed similarly to any other element in the design, with dual-ported memories treated as a single node for retiming purposes. Memories that are synthesized with a negative clock edge (to create the design illusion of asynchronicity) can be either unchanged or switched to operate on the positive edge with constraints to mandate the placement of registers.

Some FPGA designs have registers with predefined initial values. If retiming is allowed to move these registers, the proper initial values must be calculated such that the circuit still produces the same behavior.

In an ASIC model, all flip-flops start in an undefined state, and the designer must create a small state machine in order to reset the design. FPGAs, however, have all flip-flops start in a known, user-defined state, and when a dedicated global reset is applied the flip-flops are reset to it. This has serious implications in retiming.

If the decision is made to utilize the ASIC model, retiming is free to safely ignore initial conditions because explicit reset logic in state machines will still operate correctly—this is reflected in the I/O semantics. However, without the ability to violate the initial conditions with an ASIC-style model, retiming quality often suffers as additional logic is required or limits are placed on where flip-flops may be moved in a design.

In practice, performing retiming with initial conditions is NP-hard. Cong and Wu [3] have developed an algorithm that computes initial states by restricting the design to forward retiming only so that it propagates the information and registers forward throughout the computation. This is because solving initial states for all registers moved forward is straightforward, but backward movement is NP hard as it reduces to satisfiability.

Additionally, global set/reset imposes a huge constraint on retiming. An asynchronous set/reset can never be retimed (retiming cannot modify an asynchronous circut) while a synchronous set/reset just imposes too high a fanout.

An important question is how to deal with multiple clocks. If the interfaces between the clock domains are registered by clocks from both domains, it is a simple process to retime the domains separately, with mandatory registers

TABLE 18.2 ■ The results of retiming four benchmarks

Benchmark	Unretimed	Automatically retimed
AES core	48 MHz	47 MHz
Smith/Waterman	43 MHz	40 MHz
Synthetic datapath	51 MHz	54 MHz
LEON processor	23 MHz	25 MHz

on the domain crossings—the constraints placed on the I/Os ensure correct and consistent timing through the interface. Yet without this design constraint, retiming across multiple clock domains is very hard, and there does not appear to be any clean automatic solution.

Table 18.2 shows the results for a particular retiming tool [13]—the Xilinx Virtex family of FPGAs—on four benchmark circuits: an AES core, a Smith/Waterman systolic cell, a synthetic microprocessor datapath, and the LEON-I synthesized SPARC core. This tool does not use a perfectly accurate delay model and has to place registers after retiming, so it sometimes creates slightly suboptimal results.

The biggest problem with retiming is that it is of limited benefit to a well-balanced design. As mentioned earlier, if the clock cycle is defined by a single-cycle feedback loop, retiming can never improve the design, as moving the register around the feedback loop produces no effect.

Thus, for example, the Smith–Waterman example in Table 18.2 does not benefit from retiming. The Smith–Waterman benchmark design consists of a series of repeated identical systolic cells that implement the Smith–Waterman sequence alignment algorithm. The cells each contain a single-cycle feedback loop, which cannot be optimized. The AES encryption algorithm also consists of a single-cycle feedback loop. In this case, the initial design used a negative-edge Block-RAM to implement the S-boxes, which the retiming tool converted to a positive edge memory with a "must register" constraint.

Nevertheless, retiming can still be a benefit if the design consists of multiple feedback loops (such as the synthetic microprocessor datapath or the LEON SPARC–compatible microprocessor core) or an initially unbalanced pipeline. Still, for well-designed circuits, even complex ones, retiming is often only a slight benefit, as engineers have considerable experience designing reasonably optimized feedback loops.

The key benefit to retiming occurs when more registers can be added to the design along the critical path. We will discuss two techniques, repipelining and C-slow retiming, which first add a large number of registers that general retiming can then move into the optimal location.

18.2 REPIPELINING AND C-SLOW RETIMING

The biggest limitation of retiming is that it simply cannot improve a design beyond the design-dependent limit produced by an optimal placement of

registers along the critical path. As mentioned earlier, if the critical path is defined by a single-cycle feedback loop, retiming will completely fail as an optimization. Likewise, if a design is already well balanced, changing the register placement produces no improvement. As was seen in the four reasonably optimized benchmarks (refer to Table 18.2), this is often the case.

Repipelining and *C*-slow retiming are tranformations designed to add registers in a predictible matter that a designer can account for, which retiming can then move to optimize the design. Repipelining adds registers to the beginning or end of the design, changing the pipeline latency but no other semantics. *C*-slow retiming creates an interleaved design by replacing every register with a sequence of *C* registers.

18.2.1 Repipelining

Repipelining is a minor extension to retiming that can increase the clock frequency for feedforward computations at the cost of additional latency through more pipeline registers. Unlike *C*-slow retiming, repipelining is only beneficial when a computation's critical path contains no feedback loops.

Feedforward computations, those that contain no feedback loops, are commonly seen in DSP kernels and other tasks. For example, the discrete cosine transform (DCT), the fast Fourier transform (FFT), and finite impulse response filters (FIRs) can all be constructed as feedforward pipelines.

Repipelining is derived from retiming in one of two ways, both of which create semantically equivalent results. The first involves adding additional pipeline stages to the start of the computation and allowing retiming to rebalance the delays and create an absolute number of additional stages. The second involves decoupling the inputs and outputs to allow the retimer to add additional pipelining. Although these techniques operate in slightly different ways, they both provide extra registers for the retimer to then move and they produce roughly equivalent results.

If the designer wishes to add *P* pipeline stages to a design, all inputs simply have *P* delays added before retiming proceeds. Because retiming will develop an optimum placement for the resulting design, the new design contains *P* additional pipeline stages that are scattered throughout the computation. If a CAD tool supports retiming but not repipelining, the designer can simply add the registers to the input of the design manually and let the tool determine the optimum placement.

Another option is to simply remove the cycle between all outputs and inputs, with additional constraints to ensure that all outputs share an output lag, with all inputs sharing a different input lag. This way, the inputs and outputs are all synchronized but retiming can add an arbitrary number of additional pipeline registers between them. To place a limit on these registers, an additional constraint must be added to ensure that for a single I/O pair no more than *P* pipeline registers are added. Depending on the other constraints in the retiming process, this may add fewer than *P* additional pipeline stages, but will never add more than *P*.

Repipelining adds additional cycles of latency to the design, but otherwise retains the rest of the circuit's behavoir. Thus, it produces the same results and the same relative timing on the outputs (e.g., if input *B* is supposed to be presented three cycles after input *A*, or output *C* is produced two cycles after output *D*, these relative timings remain unchanged). It is only the data-in to data-out timing that is affected.

Unfortunately, repipelining can only improve feedforward designs or designs where the feedback loop is not on the critical path. If performance is limited by a feedback loop, repipelining offers no benefit over normal retiming.

Repipelining is designed to improve throughput, but will almost always make overall latency worse. Although the increased pipelining will boost the clock rate (and thus reduce some of the delay from unbalanced clocked paths), the delay from additional flip-flops on the input-to-output paths typically overwhelms this improvement and the resulting design will take longer to produce a result for an individual input.

This is a fundamental trade-off in repipelining and *C*-slow retiming. While ordinary retiming improves both latency and throughput, repipelining and *C*-slow retiming generally improve throughput at the cost of additional latency due to the additional pipeline stages required.

18.2.2 *C*-slow Retiming

Unlike repipelining, *C*-slow retiming can enhance designs that contain feedback loops. *C*-slowing enhances retiming simply by replacing every register with a sequence of *C* separate registers before retiming occurs; the resulting design operates on *C* distinct execution tasks. Because all registers are duplicated, the computation proceeds in a round-robin fashion, as illustrated in Figure 18.2.

In this example, which is 2-slow, the design interleaves between two computations. On the first clock cycle, it accepts the first input for the first stream of execution. On the second clock cycle, it accepts the first input for the second stream, and on the third it accepts the second input for the first stream. Because of the interleaved nature of the design, the two streams of execution will *never* interfere. On odd clock cycles, the first stream of execution accepts input; on even clock cycles, the second stream accepts input.

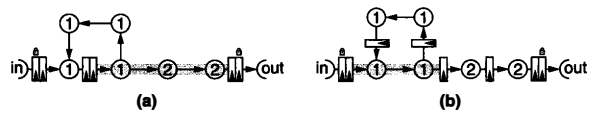


FIGURE 18.2 ■ The example from Figure 18.1, converted to 2-slow operation (a). The critical path remains unchanged, but the design now operates on two independent streams in a round-robin fashion. The design retimed (b). By taking advantage of the extra flip-flops, the critical path has been reduced from 5 to 2.

The easiest way to utilize a C-slowed block is to simply multiplex and de-multiplex C separate datastreams. However, a more sophisticated interface may be desired depending on the application (as described in Section 18.5).

One possible interface is to register all inputs and outputs of a C-slowed block. Because of the additional edges retiming creates to track I/Os and to ensure a consistent interface, every stream of execution presents all outputs at the same time, with all inputs registered on the next cycle. If part of the design is C-slowed, but all parts operate on the same clock, the result can be retimed as a complete whole and still preserve all other semantics.

One way to think of C-slowing is as a threaded design, with an overall system clock and with each stream having a "stream clock" of 1/C—each stream is completely independent. However, C-slowing imposes some more significant FPGA design constraints, as summarized in Table 18.3. Register clock enables and resets must be expressed as logic features, since each independent thread must have an independent reset or enable. Thus, they can remain features in the design but cannot be implemented by current FPGAs using native enables and resets. Other specialized features, such as Xilinx SRL16s (a mode where a LUT is used as a 16-bit shift register), cannot be utilized in a C-slow design for the same reason.

One important challenge is how to properly C-slow memory blocks. In cases where the C-slowed design is used to support N independent computations, one needs the illusion that each stream of execution is completely independent and unchanged. To create this illusion, the memory capacity must be increased by a factor of C, with additional address lines driven by a thread counter. This ensures that each stream of execution enjoys a completely separate memory space.

For dual-ported memories, this potentially enables a greater freedom in retiming: The two ports can have different lags as long as the difference in lag is less than C. After retiming, the difference is added to the appropriate port's thread counter, which ensures that each stream of execution will read and write to both ports in order while enabling slightly more freedom for retiming to proceed.

C-slowing normally guarantees that all streams view independent memories. However, a designer may desire shared memory common to all streams. Such

TABLE 18.3 ■ The effects of various FPGA features on retiming, repipelining, and C-slowing

FPGA feature	Effect on retiming	Effect on repipelining	Effect on C-slowing
Asynchronous global set/reset	Forbidden	Forbidden	Forbidden
Synchronous global set/reset	Effectively forbidden	Effectively forbidden	Forbidden
Asynchronous local set/reset	Forbidden	Forbidden	Forbidden
Synchronous local set/reset	Allowed	Allowed	Express as logic
Clock enables	Allowed	Allowed	Express as logic
Tristate buffers	Allowed	Allowed	Allowed
Memories	Allowed	Allowed	Increase size
SRL16	Allowed	Allowed	Express as logic
Multiple clock domains	Design restrictions	Design restrictions	Design restrictions

memories could be embedded in a design, but the designer would need to consider how multiple streams would affect the semantics and would need to notify any automatic tool to treat the memory in a special manner. Beyond this, there are no other semantic effects imposed by *C*-slow retiming.

C-slowing significantly improves throughput, but it can only apply to tasks where there are at least *C* independent threads of execution and where throughput is the primary goal. The reason is that *C*-slowing makes the latency substantially worse. This trade-off brings up a fundimental observation: Latency is a property of the design and computational fabric whereas throughput is a property derived from cost. Both repipelining and *C*-slow retiming can be applied only when there is sufficient task-level parallelism, in the form of either a feed-forward pipeline (repipelining) or independent tasks (*C*-slowing).

Table 18.4 shows the difference that *C*-slowing can make in four designs. While the retiming tool alone was unable to improve the AES or Smith Waterman designs, *C*-slowing substantially increased throughput, improving the clock rate by 80–95 percent! However, latency for individual tasks was made worse, resulting in significantly slower clock rates for individual tasks.

Latency can be improved only up to a given point for a design through conventional retiming. Once the latency limit is met, no amount of optimization, save a major redesign or an improvement in the FPGA fabric, has any effect. This often appears in cryptographic contexts, where feedback mode–based encryption (such as CFB) requires the complete processing of each block before the next can be processed.

In contrast, throughput is actually a part of a throughput/cost metric: throughput/area, throughput/dollar, or throughput/joule. This is because independent task throughput can be added via replication, creating independent modules that perform the same function, as well as *C*-slowing. When sufficient parallelism exists, and costs are not constrained, simply throwing more resources at the problem is sufficient to improve the design to meet desired goals.

One open question on *C*-slowing is its effect in a low-power environment. Higher throughput, achieved through high-speed clocking, naturally increases the power consumption of a design, just as replicating units for higher throughput increases power consumption. In both cases, if lower power is desired, the higher-throughput design can be modified to save power by reducing the clock rate and operating voltage.

Unlike the replicated case, the question of whether a *C*-slowed design would offer power savings if both frequency and voltage were reduced is highly design

TABLE 18.4 ■ The effect of *C*-slowing on four benchmarks

Benchmark	Initial clock	*C*-factor	*C*-slow clock	Stream clock
AES encryption	48 MHz	4-slow	87 MHz	21 MHz
Smith/Waterman	43 MHz	3-slow	84 MHz	28 MHz
Synthetic datapath	51 MHz	3-slow	91 MHz	30 MHz
LEON processor core	23 MHz	2-slow	46 MHz	23 MHz

and usage dependent. Although the finer pipelining allows the frequency and the voltage to be scaled back to a significant degree while maintaining throughput, the activity factor of each signal may now be considerably higher. Because each of the *C* streams of execution is completely independent, it is safe to assume that every wire will probably have a significantly higher activity factor that increases power consumption.

Whether the initial design before *C*-slowing has a comparable activity factor is highly input and design dependent. If the initial design's activity factor is low, *C*-slowing will significantly increase power consumption. But if that factor is high, *C*-slowing will not increase it. Thus, although the *C*-slowing transformation may have a minor affect on worst-case power (and can even result in significant savings through voltage scaling), the impact on average-case power may be substantial.

18.3 IMPLEMENTATIONS OF RETIMING

Three significant academic retiming tools have been developed for FPGAs. The first, by Cong and Wu [3], combines retiming with technology mapping. This approach enables retiming to occur before placement without adding undue constraints on the placer, because the retimed registers are packed with their associated logic. The disadvantage is a lack of precision, as delays can only be crudely estimated before placement. This tool is unsuitable for significant *C*-slowing, which creates significantly more registers that can pose problems with logic packing and placement.

The second tool, developed by Singh and Brown [6], combines retiming with placement, operating by modifying the placement algorithm to be aware that retiming is occurring and then modifying the retiming portion to enable permutation of the placement as retiming proceeds. Singh and Brown demonstrate how the combination of placement and retiming performs significantly better than retiming either before or after placement.

The simplified FPGA model used by Singh and Brown has a logic block where the flip-flop cannot be used independently of the LUT, constraining the ability of postplacement retiming to allocate new registers. Thus, the need to permute the placement to allocate registers is significantly exacerbated in their target architecture.

The third tool, developed by Weaver et al. [13], performs retiming after placement but before routing, taking advantage of the (mostly) independent register operation available on Xilinx FPGAs. (It would not apply to most Altera FPGAs.) It too also supports *C*-slowing.

Some commercial HDL synthesis tools, notably the Synopsys FPGA compiler [9] and Synplify [8], also support retiming. Because this retiming occurs fairly early in the mapping and optimization processes, it suffers from a lack of precision regarding placement and routing delays. The Amplify tool [10] can produce a higher-quality retiming because it contains placement information. Since these

tools attempt to maintain the FPGA model of initial conditions, both on startup and in the face of a global reset signal, considerable logic is added to the design.

18.4 RETIMING ON FIXED-FREQUENCY FPGAs

Fixed-frequency FPGAs differ from conventional FPGAs in that they have an intrinsic clock rate and commonly include pipelined interconnect and other design features to enable very high-speed operations. However, this fixed frequency demands a design modification to support the pipeline stages it requires.

Retiming for fixed-frequency FPGAs, unlike that for their conventional counterparts, does not require the creation of a global critical path constraint, as simply ensuring that all local requirements are met guarantees that the final design meets the architecture's required delay constraints. Instead, retiming attempts to solve these local constraints by ensuring that every path through the interconnect meets the delay requirements inherent in the FPGA. Once these local constraints are met, the final design will operate at the FPGA's intrinsic clock frequency.

Because there are no longer any global constraints, the W and D matrices are not created. A fixed-frequency FPGA does not require the global constraints, so having only to solve a set of local constraints requires linear, not quadratic, memory and $O(n^2)$, rather than $O(n^2 lg(n))$, execution time. This speeds the process considerably.

Additionally, only a single invocation of the constraint solver is necessary to determine whether the current level of pipelining can meet the constraints imposed by the target architecture. Unfortunately, most designs do not possess sufficient pipelining to meet these constraints, instead requiring a significant level of repipelining or C-slow retiming to do so. The level necessary can be discovered in two ways.

The first approach is simply to allow the user to specify a desired level of repipelining or C-slowing. The retiming system then adds the specified number of delays and attempts to solve the system. If a solution is discovered, it is used. Otherwise, the user is notified that the design must be repipelined or retimed to a greater degree to meet the array's clock cycle. The second approach requires searching to find the minimal level of repipelining or C-slowing necessary to meet the constraints. Although this necessitates multiple iterations of the constraint solver, fixed-frequency retiming only requires local constraints. Without having to check the global constraints, this process proceeds quickly. The resulting level of repipelining or C-slowing is then reported to the user.

Fixed-frequency FPGAs require retiming considerably later in the tool flow. It is impossible to create a valid retiming until routing delays are known. Since the constraints required invariably depend on placement, the final retiming process must occur afterwards. Some arrays, such as HSRA [10], have deterministic routing structures that enable retiming to be performed either before or after routing. Other interconnect structures, such as SFRA [12], lack deterministic routing and require that retiming be performed only after routing.

Finally, the fact that fixed-frequency arrays may use considerably more pipelining than conventional arrays makes retiming registers a significant architectural feature. Because these delay chains [10], either on inputs or on outputs, are programmable, the array can implement longer ones. A common occurrence after aggressive C-slow retiming is a design with several signals requiring considerable delay. Therefore, dedicated resources to implement these features are effectively required to create a viable fixed-frequency FPGA.

18.5 C-SLOWING AS MULTI-THREADING

There have been numerous multi-threaded architecture designs, but all share a common theme: increasing system throughput by enabling multiple streams of execution, or threads, to operate simultaneously. These architectures generally fall into four classes: context switching always without bypassing (HEP [7] and Tera [2]), context switching on event (Intel IXP) [14], interleaved multi-threaded, and symmetric multi-threaded (SMT) [11]. The ideal goal of all of them is to increase system throughput by operating on multiple streams of execution.

The general concept of C-slow retiming can be applied to highly complex designs, including microprocessors. Unlike a simple FIR filter bank or an encryption algorithm, it is not a simple matter of inserting registers and balancing delays. Nevertheless, the changes necessary are comparatively small and the benefits substantial: producing a simple, statically scheduled, higher clock rate, multi-threaded architecture that is semantically equivalent to an interleaved-multi-threaded architecture, alternating between a fixed number of threads in a round-robin fashion to create the illusion of a multiprocessor system.

C-slowing requires three minor architectural changes: enlarging and modifying the register file and TLB, replacing the cache and memory interface, and slightly modifying the interrupt semantics. Beyond that, it is simply a matter of replacing every pipeline register in both the control logic and the datapath with C registers and then moving the registers to balance the delays, as is traditional in the C-slow retiming transformation and can be performed by an automatic tool. The resulting design, as expected, has full multi-threaded semantics and improved throughput because of a significantly higher clock rate. Figure 18.3 shows how this transformation can operate.

The biggest complications in C-slowing a microprocessor are selecting the implementation semantics for the various memories through the design. The first type keeps the traditional C-slow semantics of complete independence, where each thread sees a completely independent view, usually by duplication. This applies to the register file and most of the state registers in the system. This occurs automatically if C-slowing is performed by a tool, because it represents the normal semantics for C-slowed memory.

The second is completely shared memory, where every thread sees the same memory, such as the caches and main memory of the system. Most such memories exist in the non-C-slowed portion and so are unaffected by an automatic tool.

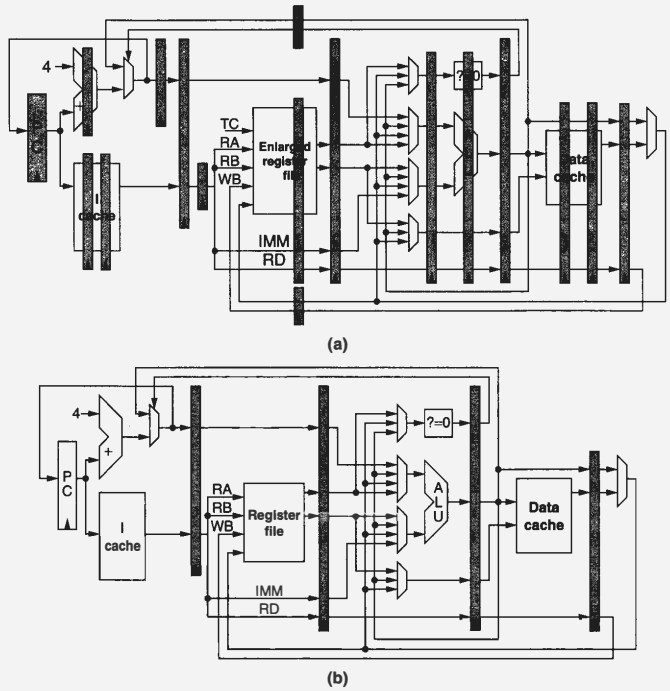


FIGURE 18.3 ■ A traditional five-stage microprocessor pipeline, and its conversion to 3-slow operation.

The third is dynamically shared, where a hardware thread ID or a software thread context ID is tagged to each entry, with only the valid tags used. This breaks the automatic *C*-slow semantics and is best employed for branch predictors and similar caches. Such memories need to be constructed manually, but offer potential efficiency advantages as they do not need to increase in size. Because they cannot be constructed automatically they may be subject to interference or synergistic effects between threads.

The biggest architectural changes are to the register file: It needs to be increased by a factor of *C*, with a hardware thread counter to select which group of registers is being accessed. Now each thread will see an independent set of registers, with all reads and writes for the different threads going to separate memory locations. Apart from the thread selection and natural enlargement, the only piece remaining is to pipeline the register access. If necessary, the

C independently accessed sections can be banked so that the register file can operate at a higher clock frequency.

Naturally, this linearly increases the size of the register file, but pipelining the new larger file is not difficult since each thread accesses a disjoint register set, allowing staggered access to the banks if desired. This matches the automatic memory transformations that C-slowing creates: increasing the size and ensuring that each task has an independent view of memory.

To maintain the illusion that the different threads are running on completely different processors, it is important that each thread have an independent translation of memory. The easiest solution is to apply the same transformations to the TLB that were applied to the register file: increasing the size by C, with each thread accessing its own set, and pipelining access. Again, this is the natural result of applying the C-slow semantics from an automatic tool.

The other option is to tag each TLB entry. The interference effect may be significant if the associativity or size of the TLB is low. In such a case, and considering the generally small size of most TLBs, increasing the size (although perhaps by less than a factor of C) is advisable. Software thread ID tags are preferable to hardware ID tags because they reduce the cost of context switching if a shared TLB is used and may also provide some synergistic effects. In either case, a shared TLB requires interlocking between TLB writes to prevent synchronization bugs.

If the caches are physically addressed, it is simply a matter of pipelining access to improve throughput without splitting memory. Because of the interlocked execution of the threads and the pipelined nature of the modified caches, no additional coherency mechanisms are required except to interlock any existing test-and-set or atomic read/write instructions between the threads to ensure that each instruction has time to be completed.

Such cache modifications occur outside the C-slow semantics, suggesting that the cache needs to be changed manually. This means that the cache and memory controller must be manually updated to support pipelined access from the distinct threads, and must exist outside of the C-slowed core itself.

Unfortunately, virtually addressed caches are significantly more complicated: They require that each tag include thread ownership (to prevent one thread from viewing another's version of memory) and that a record of virtual-to-physical mappings be maintained to ensure coherency between threads. These complications suggest that a physically addressed cache would be superior when C-slowing a microprocessor to produce a simple multi-threaded design. A virtually addressed cache is one of the few structures that do not have a natural C-slow representation or that can easily exist outside a C-slowed core.

The rest of the machine state registers, being both loaded and read, are automatically separated by the C-slow transformation. This ensures that each thread will have a completely independent set of machine registers. Combined with the distinct registers and TLB tagging, each thread will see an independent processor.

The only other portion that needs to be changed is the interrupt semantics. Just as the rest of the control logic is pipelined, with control registers duplicated,

the same transformations need to be applied to the interrupt logic. Thus, every external interrupt is interpreted by the rules corresponding to *every* virtual processor running in the pipeline. Yet, since the control registers are duplicated, the OS can enforce policies where different interrupts are handled by different execution streams. Similarly, internally driven interrupts (such as traps or watchdog timers), when *C*-slowed, are independent between threads, as *C*-slowing ensures that each thread sees only its own interrupts.

In this way, the OS can ensure that one virtual thread receives one set of externally sourced interrupts while another receives a different set. This also suggests that interrupts be presented to all threads of execution, enabling each thread (or even multiple threads) to service the appropriate interrupt.

The resulting design has full multi-threaded semantics, with each of *C* threads being independent. Because *C*-slowing can improve the clock rate (by two times in the case of the LEON benchmark), this can easily and substantially improve the throughput of a very complex design.

18.6 WHY ISN'T RETIMING UBIQUITOUS?

An interesting question is why retiming is not heavily used in FPGA tool flows. Although some FPGA vendors [1] and CAD vendors [8] support retiming, it is not universally available, and even when it is, it is usually optional.

There are three major factors that limit the general adoption of retiming: It interacts poorly with many critical FPGA features; it can only optimize poor implementations yet is not a substitute for good implementation; and it is computationally intensive.

As mentioned earlier, retiming does not work well with initial conditions or global resets—features that FPGA designers have traditionally relied on. Likewise, BlockRAMs, hardware clock eEnables, and other features can pin registers, limiting the ability of a retiming tool to move them. For these reasons, many FPGA designs *cannot* be effectively retimed.

A related observation is that retiming helps only poor designs and, moreover, only fixes one common deficiency of a poor design, not all of them. Additionally, if the designer has enough savvy to work around the limitations of retiming, he will probably produce a naturally well-balanced design.

Finally, although retiming is a polynomial time algorithm, its still superlinear. As designs continue to grow in size, $O(n^2 lg(n))$ can still be too long for many uses. This is especially problematic as the Moore's Law scaling for FPGAs is currently greater than that for single-threaded microprocessors.

References

[1] Altera Quartus II eda (*http://www.altera.com/*).

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith. The Tera computer system. *Proceedings of the 1990 International Conference on Supercomputing*, 1990.

[3] J. Cong, C. Wu. Optimal FPGA mapping and retiming with efficient initial state computation. *Design Automation Conference*, 1998.
[4] C. Leiserson, F. Rose, J. Saxe. Optimizing synchronous circuitry by retiming. *Third Caltech Conference On VLSI*, March 1993.
[5] H. Schmit. Incremental reconfiguration for pipelined applications. *Proceedings of the IEEE Symposium on Field-Programmable Gate Arrays for Custom Computing Machines*, April 1997.
[6] D. P. Singh, S. D. Brown. Integrated retiming and placement for field-programmable gate arrays. *Tenth ACM International Symposium on Field-Programmable Gate Arrays*, 2002.
[7] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. Advances in laser scanning technology. *SPIE Proceedings* 298, Society for Photo-Optical Instrumentation Engineers, 1981.
[8] Synplify pro (*http://www.synplicity.com//products//synplifypro//index.html*).
[9] Synopsys, Inc. Synopsis FPGA Compiler II (*http://www.synopsys.com*).
[10] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, A. DeHon. HSRA: High-speed, hierarchical synchronous reconfigurable array. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 1999.
[11] D. M. Tullsen, S. J. Eggers, H. M. Levy. Simultaneous multi-threading: Maximizing on-chip parallelism. *Proceedings 22nd Annual International Symposium on Computer Architecture*, June 1995.
[12] N. Weaver, J. Hauser, J. Wawrzynek. The SFRA: A corner-turn FPGA architecture. *Twelfth International Symposium on Field-Programmable Gate Arrays*, 2004.
[13] N. Weaver, Y. Markovskiy, Y. Patel, J. Wawrzynek. Postplacement *C*-slow retiming for the Xilinx-Virtex FPGA. *Eleventh ACM International Symposium on Field-Programmable Gate Arrays*, 2003.
[14] Intel Corporation. The Intel IXP network processor. *Intel Technology Journal* 6(3), August 2002.

CHAPTER 19

CONFIGURATION BITSTREAM GENERATION

Steven A. Guccione
Cmpware, Inc.

While a reconfigurable logic device shares some of the characteristics of a fixed hardware device and some of a programmable instruction set processor, the details of the underlying architecture and how it is programmed are what distinguish these machines. Both a reconfigurable logic device and an instruction set processor are programmable by "software," but the internal organization and use of this software are quite different. In an instruction set processor, the programming is a set of binary codes that are incrementally fed into the device during operation. These codes actually carry out a form of reconfiguration inside the processor. The arithmetic and logic unit(s) (ALU) is configured to perform a requested function and various control multiplexers (MUXes) that control the internal flow of data are set. In the instruction set machine, these hardware components are relatively small and fixed and the system is reconfigured on a cycle-by-cycle basis. The processor itself changes its internal logic and routing on every cycle based on the input of these binary codes.

In a processor, the binary codes—the processor's machine language—are fairly rigid and correspond to sequential "instructions." The sequence of these instructions to implement a program is often generated by some higher-level automatic tool such as a high-level language (HLL) compiler from a language such as Java, C, or C++. But they may, in reality, come from any source. What is important is that the collection of binary data fits this rigid format. The collection of binary data goes by many names, most typically an "executable" file or even more generally a "binary program."

A reconfigurable logic device, or field-programmable gate array (FPGA), is based on a very different structure than that of an instruction set machine. It is composed of a two-dimensional array of programmable logic elements joined together by some programmable interconnection network. The most significant difference between FPGA and the instruction set architecture is that the FPGA is typically intended to be programmed as a complete unit, with the various internal components acting together in parallel. While the structure of its binary programming (or configuration) data is every bit as rigid as that of an instruction set processor, the data are used spatially rather than sequentially.

In other words, the binary data used to program the reconfigurable logic device are loaded into the device's internal units before the device is placed

in its operating mode, and typically, no changes are made to the data while the device is operating. There are some significant exceptions to this rule: The configuration data may in fact be changed while a device is operational, but this is somewhat akin to "self-modifying code" in instruction set architectures. This is a very powerful technique, but carries with it significant challenges.

The collection of binary data used to program the reconfigurable logic device is most commonly referred to as a "bitstream," although this is somewhat misleading because the data are no more bit oriented than that of an instruction set processor and there is generally no "streaming." While in an instruction set processor the configuration data are in fact continuously streamed into the internal units, they are typically loaded into the reconfigurable logic device only once during an initial setup phase. For historical reasons, the somewhat undescriptive "bitstream" has become the standard term.

As much as the binary instruction set interface describes and defines the architecture and functionality of the instruction set machine, the structure of the reconfigurable logic configuration data bitstream defines the architecture and functionality of the FPGA. Its format, however, currently suffers from a somewhat interesting handicap. While the format of the programming data of instruction set architectures is freely published, this is almost never the case with reconfigurable logic devices. Almost all of them that are sold by major manufacturers are based on a "closed" bitstream architecture.

The underlying structure of the data in the configuration bitstream is regarded by these companies as a trade secret for reasons that are historical and not entirely clear. In the early days of reconfigurable logic devices, the underlying architecture was also a trade secret, so publishing the configuration bitstream format would have given too many clues about it. It is presumed that this was to keep competitors from taking ideas about an architecture, or perhaps even "cloning" it and providing a hardware-compatible device. It also may have reassured nervous FPGA users that, if the bitstream format was a secret, then presumably their logic designs would be difficult to reverse-engineer.

While theft and cloning of device hardware do not appear to be a potential problem today, bitstream formats are still, perhaps out of habit alone, treated as trade secrets by the major manufacturers. This is a shame because it prohibits interesting experimentation with new tools and techniques by third parties. But this is perhaps only of interest to a very small number of people. The vast majority of users of commercial reconfigurable logic devices are happy to use the vendor-supplied tools and have little or no interest in the device's internal structure as long as the logic design functions as specified. However, for those interested in the architecture of reconfigurable logic devices, trade secrecy is an important subject.

While exact examples from popular industry devices are not possible because of this secrecy, much is publicly known about the underlying architectures, the general way a bitstream is generated, and how it operates when loaded into a device.

19.1 THE BITSTREAM

The bitstream spatially represents the configuration data of a large collection of small, relatively simple hardware components. Thus, we can identify these components and discuss the ways in which the bitstream is used to produce a working digital circuit in a reconfigurable logic device. Although there is really no limit to the types of units possible in a reconfigurable logic device, two basic structures make up the microarchitecture of most modern FPGAs. These are the lookup table (LUT) and the switch box.

The LUT is essentially a very small memory element, typically with 16 bits of bit-oriented storage. Some early FPGAs used smaller 8-bit LUTs, and other more exotic architectures used non-LUT structures. In general, however, the vast majority of commercial FPGA devices sold over the last decade use the 16-bit LUT as a primary logic building block.

The functionality of LUTs is very simple. Binary data are loaded into them to produce some Boolean function. In the case of the 16-bit LUT, there are four inputs, which can produce any arbitrary 4-input Boolean logic function. For instance, to provide the AND function of all four inputs, each bit in the memory except the bit at address `A(1,1,1,1)` is loaded with a binary `0` and the `A(1,1,1,1)` bit is loaded with a `1`. The address inputs of the LUT are used as the inputs to the logic function, with the output of the LUT providing the output of the logic function. Figure 19.1 illustrates this mapping of a 2-input LUT to a 2-input AND gate.

While the LUTs provide the logic for the circuit, the switch boxes provide the interconnection. These switch boxes are typically made up of multiplexers in various regular configurations. These multiplexers are controlled by bits of memory that select the inputs and send them to the multiplexer's outputs. Figure 19.2 shows a typical configurable interconnect element constructed using a multiplexer.

The multiplexer inputs in Figure 19.2 are controlled by two memory elements that are set during configuration. They select which input value is sent to the output. By connectiong large numbers of elements of this type, an interconnection

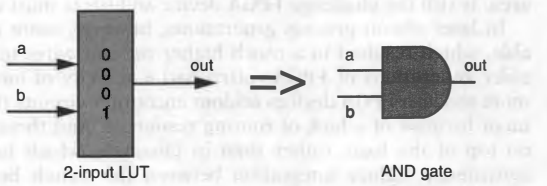


FIGURE 19.1 ■ A 2-input LUT configured as an AND gate.

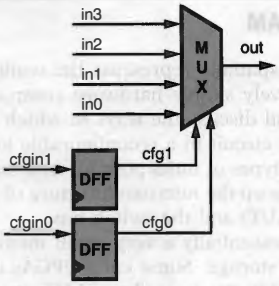


FIGURE 19.2 ■ A configurable 4-input multiplexer used in routing.

network of the kind typically used to construct modern reconfigurable logic devices can be made.

In various topologies, the ouputs of the multiplexers in the switch boxes feed the address inputs of the LUTs; the outptus of the LUTs, in turn, feed the inputs of the switch box multiplexers. This provides a basic reprogrammable architecture capable of producing arbitrary logic functions, as well as the ability to interconnect these functions in a variety of ways. How complex a circuit a given reconfigurable logic device can implement is based on both the number of LUTs and the size and complexity of the interconnection fabric.

In fact, the topology of the interconnect fabric and the implementation of the switch boxes is perhaps the defining characteristic of an FPGA architecture. Older FPGAs had a limited silicon area and few metal layers to supply wires. For this reason, the LUTs were typically "islands" of logic, with the interconnect wires running in the "channels" between them. Where these channels intersected were the switch boxes. How many wires to use and how to configure the switch boxes were the main work of the FPGA architect. Balancing the cost of more wires with the needs of typical digital circuit was important to making a cost-effective device that would be commercially successful. Covering as many potential circuit designs as possible at as high a speed as possible, but with the smallest silicon area, is still the challenge FPGA device architects must confront.

In later silicon process generations, however, more metal layers were available, which resulted in a much higher ratio of wires to logic in FPGAs. Where older generations of FPGAs often had a scarcity of interconnection resources, more modern FPGA devices seldom encounter circuits they are unable to implement because of a lack of routing resources. And these wires now tend to run on top of the logic rather than in channels, which has led to higher circuit densities, a tighter integration between the switch boxes and the logic, and faster interconnect.

The configuration bitstream data for the routing are essentially the multiplexer inputs in these switch boxes. The memory for these MUX inputs tends

to be individual memory elements such as flip-flops scattered around the device as needed, establishing the basic bitstream for the FPGA: the LUT data plus the bits to control the routing multiplexers.

While the multiplexer and switch boxes are the basic elements of modern FPGA devices, many other components are possible. One of the more popular is a configurable input/output block, or IOB. An IOB is typically connected to the end of one of the wires in the routing system on one side and to a physical device pin on the other. It is then configured to define the type of pin used by this device: either input or output. More complex IOBs can configure pin voltages and even parameters such as capacitance, and some even provide higher-level support for various serial communication protocols. Much like switch boxes, the configuration bitstream data for the IOBs are some collection of bits used to set flip-flops within them to select these features.

In addition to IOBs, other, more special-purpose units have turned up in later generations of FPGA devices. Two prominent examples are block memory and multiplier units. Block memory (BlockRAM) is simply relatively large RAM units that are usually on the order of 1K bits but can be implemented in any number of ways. The actual data bits may be part of the bitstream, which initializes the BlockRAM upon power-up. To reduce the size of the bitstream, however, this data may be absent and internal circuitry may be required to reset and initialize the BlockRAM.

In addition to the internal data, the BlockRAM is typically interfaced to the switch boxes in various ways. Its location and interfacing to the interconnection network is a major architectural decision in modern reconfigurable logic device design.

Because the multiplication function has become more popular in FPGA designs and because FPGAs are so inefficient at implementing such circuits, the addition of hardwired multiplier units into modern FPGA devices has been increasing. These units typically have no internal state or configuration, but are interfaced to the interconnection network in a manner similar to the BlockRAM interface. As with the BlockRAM, where to locate these resources and how many to include are major architectural decisions that can have a large impact on the size and efficiency of modern FPGAs.

Many other features also find control bits in the FPGA bitstream. Some of these are global control related to configuration and reconfiguration; others are ID codes and error-checking information such as cyclic redundancy check codes. How these features are implented is very architecture dependent and can vary widely from device family to device family. One common feature is basic control for bit-level storage elements, often in the form of flip-flops on the LUT output. Various control bits often set circuit parameters such as the flip-flop type (D, JK, T) or the clock edge trigger type (rising or falling edge). The ability to chage the flip-flop into a transparent D-type latch is also a popular option. Each of these bits also contributes to the configuration data, with one set of flip-flop configuration settings per LUT being typical.

Finally, while the items just discussed are the major standard units used to construct modern FPGA devices and define the configuration bitstream, there

TABLE 19.1 ■ Configuration bitstream sizes

Year	Device	Bits
1986	XC2018	18 Kbits
1988	XC3090	64 Kbits
1990	XC4013	248 Kbits
1994	XC4025	422 Kbits
1996	XC4028	668 Kbits
1998	XCV1000	6.1 Mbits
2000	XCV3200	16 Mbits
2003	XC2V8000	29 Mbits

is no limit to the types of circuits and configurations possible. For example, an interest in analog FPGAs has resulted in unique architectures to perform analog signal processing. Also, some coarser-grained reconfigurable logic devices have moved up in granularity from LUTs to ALUs, and these devices have somewhat different bitstream structures. Other architectures have gone in the other direction toward extremely fine-grained architectures. One notable device, the Xilinx XC6200, has a logic cell that is essentially a 2-input multiplexer. The balance of routing and logic in these devices has made them less attractive than coarser-grained devices, but they have not been reevaluated in the context of the denser routing available with newer multilayer metal processes and so may yet have some promise.

As FPGA devices themselves have grown, so has the size of the configuration bitstreams. In fact, bitstream size can be a reasonable gauge of the size and complexity of the underlying device, which can be useful because it is a single number that is readily available. Table 19.1 gives some representative sizes of various bitstreams from members of the Xilinx family of FPGAs and the approximate dates they were introduced.

19.2 DOWNLOADING MECHANISMS

The FPGA configuration bitstream is typically saved externally in a nonvolatile memory such as an EPROM. The data are usually loaded into the device shortly after the initial power-up sequence, most often bit-serially. (This loading mechanism may be the reason that many engineers perceive the configuration data as a "stream of bits.") The reason for serial loading is primarily one of cost and convenience. Since there is usually no particular hurry in loading the FPGA configuration data on power-up, using a single physical device pin for this data is the simplest, cheapest approach. Once the data are fully loaded, this pin may even be put into service as a standard I/O pin, thus preventing the configuration downloading mechanism from consuming valuable I/O resources on the device.

A serial configuration download is the norm, but some FPGA devices have a parallel download mode that typically permits the use of eight I/O pins to

download configuration data in parallel. This may be helpful for designs that use an 8-bit memory device and for applications where reprogramming is common and speed is important—often the case when an FPGA is controlled by a host processor in a coprocessor arrangement. As with the serial approach, the pins may be returned to regular I/O duty once downloading is complete.

One place where such high-bandwidth configuration is useful is in the device test in the factory. Testing FPGA devices after manufacture can be a very expensive task, mostly because of time spent attached to the test equipment. Thus, decreasing the configuration download time by a factor of eight may result in the FPGA manufacturer requiring substantially fewer pieces of test equipment, which can result in a significant cost savings during manufacture. Anecdotal evidence suggests that high-speed download is driven mostly by increased test efficiency and not by any customer requirements related to runtime reconfiguration.

One type of device that is based on nonvolatile memory bears mention here. Rather than using RAM and flip-flops as the internal logic and control, commercially available devices from companies such as Actel use nonvolatile Flash-style internal configuration memory. These devices are programmed once and do not require reloading of configuration data on power-up, which can be important in systems that must be powered-up quickly. Such devices also tend to be more resistant to soft errors that can occur in volatile RAM devices. This makes them especially popular in harsh environments such as space and military applications.

19.3 SOFTWARE TO GENERATE CONFIGURATION DATA

The software used to generate configuration bitstream data for FPGA devices is perhaps some of the most complex available. It usually consists of many layers of functionality and can run on the largest workstations for hours or even days to produce the output for a single design. While the details of this software are beyond the scope of this chapter, some of the way the software generates this bitstream will be briefly discussed in this section.

The top-level input to the FPGA design software is most often a hardware description language (HDL) or a graphical circuit design created with a schematic capture package. This representation is usually then translated into a low-level description more closely related to the implementation technology. A common choice for this intermediate format is EDIF (Electronic Design Interchange Format). This translation is fairly generic and such tools are widely available from a variety of software vendors.

The EDIF description is still not suitable for directly programming the reconfigurable logic device. In the typical FPGA, the underlying circuit must be "mapped" onto the array of LUTs and switch boxes. While the actual implementation may vary, the two basic processes for getting such abstract circuit descriptions into a physical representation of FPGA configuration data are placement/routing and mapping. Figure 19.3 shows the basic flow of this process.

Mapping refers to taking general logic descriptions and converting them into the bits used to fill in a LUT. This is sometimes referred to as "packing," because

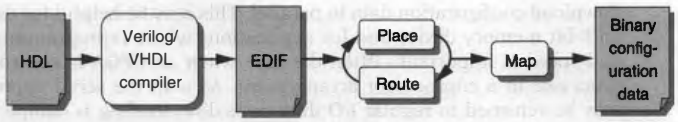


FIGURE 19.3 ■ The tool flow for producing the configuration bitstream.

several small logic gates are often "packed" into a single LUT. There is also a notion of placement that decides which LUT should receive the data, but this may also be considered a part of the mapping process.

Once the values for the LUTs have been decided, software can begin to decide how to interconnect the LUTs in a process called "routing." There are many algorithms of varying sophistication to perform routing, and factors such as circuit timing may be taken into account in the process. The result of the routing procedure is eventually used to supply the configuration data for the switch boxes.

Of course, this description is highly simplified, and mapping and routing can take place in various interleaved phases and can be optimized in a wide variety of ways. Still, this is the essential process used to produce the configuration bitstream. Finally, data for configuring the IOBs are typically input in some form that is aware of the particular package being used for the FPGA device. Once all of this data have been defined and collected, they can be written out to a single file containing the configuration bitstream.

As mentioned, FPGA configuration bitstream formats have almost always been proprietary. For this reason, the only tools available to perform bitstream generation tasks have been those supplied by the device manufacturer. The one notable exception is the Xilinx XC6200, which had an "open" bitstream. One of the XC6200's software tools was an application program interface (API) that permitted users to create configuration data or to even directly alter the configuration of an XC6200 in operation mode. Some of this technology was transferred to more mainstream Xilinx FPGAs and is available from Xilinx as a toolkit called JBits.

JBits is a Java API into the configuration bitstream for the XC4000 and Virtex device families. With JBits, the actual values on LUTs and switch box settings, as well as all other microarchitectural components, could be directly programmed. While the control data could be used to produce a traditional bitstream file, they could also be accessed directly and changed dynamically. The JBits API not only permitted dynamic reconfiguration of the FPGA but also permitted third-party tools to be built for these devices for the first time. JBits was very popular with researchers and users with exotic design requirements, but it never achieved popular use as a mainstream tool, although many of its related toolkit components, including the debug tool and partial reconfiguration support, have found their way into more mainstream software.

19.4 SUMMARY

While the generation of bitstream data to configure an FPGA device is a very common activity, there has been very little information available on the details of either the configuration bitstream or the underlying FPGA architecture. Thus, the FPGA can best be viewed as a collection of microarchitecture components, chiefly LUTs and switch boxes. These components are configured by writing data to the LUT values and to control memories associated with the switch boxes. Setting these bits to various values results in custom digital circuits.

A variety of tools and techniques are used to program reconfigurable logic devices, but all must eventually produce the relatively small configuration "bitstream" data the devices require. This data is in as rigid a format as any binary execution data for a microprocessor, but this format is typically proprietary and unpublished. While direct examination of actual commercial bitstream data is largely impossible, the general structure and the microarchitecture components configured by this data can be examined, at least in the abstract.

References

[1] Xilinx, Inc. *Virtex Data Sheet*, Xilinx, Inc., 1998.

[2] S. A. Guccione, D. Levi, P. Sundararajan. JBits: A Java-based interface for reconfigurable computing. *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, Laurel, MD, September 1999.

[3] E. Lechner, S. A. Guccione. The Java environment for reconfigurable computing. *Proceedings of the Seventh International Workshop on Field-Programmable Logic and Applications*, September 1997.

[4] Xilinx, Inc. *XAPP151: Virtex Series Configuration Architecture User Guide (version 1.7)*, (*http://direct.xilinx.com/bvdocs/appnotes/xapp151.pdf*), October 20, 2004.

[5] P. Alfke. *FPGA Configuration Guidelines (version 1.1)* (*http://direct.xilinx.com/bvdocs/appnotes/xapp090.pdf*), November 24, 1997.

[6] Xilinx, Inc. *XC6200 Field-Programmable Gate Arrays*, Xilinx, Inc., 1997.

[7] V. Betz, J. Rose. VPR: A new packing, placement, and routing tool for FPGA research. *Proceedings of the Seventh International Workshop on Field-Programmable Logic and Applications*, September 1997.

[8] Xilinx, Inc. *JBits 2.8 SDK for Virtex*, Xilinx Inc., 1999.

CHAPTER 20

FAST COMPILATION TECHNIQUES

Ken Eguro, Scott Hauck
Department of Electrical Engineering
University of Washington

Most users rely on sophisticated CAD tools to implement their circuits on field-programmable gate arrays (FPGAs). Unfortunately, since each of these tools must perform reasonably complex optimization, the entire process can take a long time. Although fairly slow compilation is fine for the majority of current FPGA users, there are many situations that demand more efficient techniques. Looking into the future, we see that faster CAD tools will become necessary for many different reasons.

FPGA scaling. Modern reconfigurable devices have a much larger capacity compared to those from even a few years ago, and this trend is expected to continue. To handle the dramatic increase in problem size, while maintaining current usability and compilation times, smarter and more efficient techniques are required.

Hardware prototyping and logic emulation systems. These are very large multi-FPGA systems used for design verification during the development of other complex hardware devices such as next-generation processors. They present a challenging CAD problem both because of the sheer number of FPGAs in the system and because the compilation time for the design is part of the user's debug cycle. That is, the CAD tool time directly affects the usability of the system as a whole.

Instance-specific design. Instance-specific designs are applications where a given circuit can only solve one particular occurrence of a problem. Because of this, every individual hardware implementation must be created and mapped as the problems are presented. Thus, the true solution time for any specific example includes the netlist compilation time.

Runtime netlist compilation. Reconfigurable computing systems are often constructed with an FPGA or an array of FPGAs alongside a conventional processor. Multiple programs could be running in the system simultaneously, each potentially sharing the reconfigurable fabric. In some of the most aggressive systems, portions of a program are individually mapped to the FPGA while the instructions are in flight. This creates a need for almost real-time compilation techniques.

For each of these systems, the runtime of the CAD tools is a clear concern. In this chapter, we consider each scenario and cover techniques to accelerate the

various steps in the mapping flow. These techniques range from fairly cost-neutral optimizations that speed the CAD flow without greatly impacting circuit quality to more aggressive optimizations that can significantly accelerate compilation time but also appreciably degrade mapping quality.

FPGA scaling

The mere scaling of VLSI technology itself has created part of the burden for conventional FPGA CAD tools. Fulfilling Moore's Law, improvements in lithography and manufacturing techniques have radically increased the capabilities of integrated circuits over the last four decades. Of course, just as these advancements have increased the performance of desktop computers, they have increased the logic capacity of FPGAs. Correspondingly, the size of desired applications has also increased. Because of this simultaneous scaling across the industry, reconfigurable devices and their applications become physically larger at approximately the same rate that general-purpose processors become faster.

Unfortunately, this does not mean that the time required to compile a modern FPGA design on a modern processor stays the same. Over a particular period of time, desktop computers and compute servers will become twice as fast and, concurrently, FPGA architectures and user circuits will double in size. Since the complexity of many classical design compilation techniques scale super-linearly with problem size, however, the relative runtime for mapping contemporary applications using contemporary machines will naturally rise.

To continue to provide reasonable design compilation time across multiple FPGA generations, changes must be made to prevent a gap between available computational power and netlist compilation complexity. However, although application engineers depend on compilation times of at most a few hours to meet fast production timelines, they also have expectations about the usable logic block density and achievable clock frequency for their applications. Thus, any algorithmic improvements or architectural changes made to speed up the mapping process cannot come at the cost of dramatically increased critical-path timing or reduced mapping density.

Hardware prototyping and logic emulation systems

The issue of nonscalable compilation is even more obvious in large prototyping or logic emulation systems. These devices integrate multiple FPGAs into a single system, harnessing tens to thousands. As Chapter 30 discusses in more detail, the fundamental size of typical circuits on these architectures suggests fast mapping techniques. However, even more critical, the compilation time of the netlists themselves may become a limiting factor in the basic usefulness of the entire system.

Hardware prototyping is often employed for many reasons. One of the greatest advantages of hardware emulation over software simulation is its extremely fast validation time. During the design and debug cycle of hardware development, hundreds of thousands of test vectors may be applied to ensure that a given implementation complies with design specifications. Although an FPGA-based prototyping system cannot be expected to achieve anywhere near the clock rate of the dedicated final product, the sheer volume of tests that need to be performed

every time a change is made to the system makes software simulation too slow to have inside the engineering design loop. That said, software simulation code can easily accommodate design updates and, more important, the changes have a predictable compilation time of minutes to hours, not hours to days. Still, since reconfigurable logic emulation systems maintain such a runtime advantage over software simulation, prototyping designers are willing to exchange some of the classical FPGA metrics of implementation quality, critical-path timing, and logical density for faster and more predictable compilation time.

Instance-specific design

Similar to logic emulation systems, the netlist compilation time of instance-specific circuits can greatly affect the overall value of an FPGA-based implementation. For example, although Boolean satisfiability is NP-complete, the massive parallelism offered by reconfigurable fabrics can often solve these problems extremely quickly—potentially on the order of milliseconds (see Chapter 29). Unfortunately, these FPGA implementations are equation-specific, so the time required to solve any given SAT problem is not determined by the vanishingly short runtime of the actual mapped circuit running on a reconfigurable device, but instead is dominated by the compilation time required to obtain the programming bitstream in the first place—potentially on the order of hours.

Because of this reliance on netlist compilation, the Boolean satisfiability problem differs strongly from more traditional reconfigurable computing applications for two reasons.

First, if we disregard compilation time, FPGA-based SAT solvers can obtain two to three orders of magnitude better performance than software-based solutions. Thus, the critical path and, by extension, the overall quality of the mapping in the classical sense are virtually irrelevant. As long as compilation results in *any* valid mapping, the vast majority of the performance benefit will be maintained. While some effort is required to reliably produce routable circuits, we can make huge concessions in terms of circuit quality in the name of speeding compilation. Mappings that are quickly produced, but possibly slow, will still drastically improve the overall solution runtime.

Second, features of the SAT problem itself suggest that application-specific approaches might be worthwhile. For example, because SAT solvers typically have very structured forms, fast SAT-specific CAD tools can be created. One possibility is the use of preplaced and prerouted SAT-specialized macros that simply need to be assembled together to create the overall system. To extend the concept of application-specialized tuning to its logical end, architectural changes can even be made to the reconfigurable fabric itself to make the device particularly amenable to simple, fast mapping techniques. That said, the large engineering effort this would involve must be weighed against the possible benefits.

Runtime netlist compilation

All reconfigurable computing systems have a certain amount of overhead that eats away at their performance benefit. Although kernel execution might be blindingly fast once started on the reconfigurable logic, its overall benefit is limited by the

need to profile operations, transfer data, and configure or reconfigure the FPGA. Reconfigurable computing systems that use dynamically compiled applications have the additional burden of runtime netlist compilation. These systems only map application kernels to the hardware during actual system execution, in the hope that runtime data, such as system loads, resource availability, and execution profiles, can improve the resultant speedups provided by the hardware. Their almost real-time requirements demand the absolutely fastest compilation techniques. Thus, even more so than instance-specific designs, these systems are only concerned with compilation speed.

Mapping stages

When evaluating mapping techniques for high-speed circuit compilation, we have to remember that the individual tools are part of a larger system. Therefore, any quality degradation in an early stage may not only limit the performance of the final mapping, but also make subsequent compilation problems more difficult. If these later mapping phases are more difficult, they may require a longer runtime, overwhelming the speedups achieved in earlier steps. For example, a poor-quality placement obtained very quickly will likely make the routing problem harder. Since we are interested in reducing the runtime of the compilation phase as a whole, we must ensure that we do not simply trade placement runtime for routing runtime. We may even run the risk of increasing total compilation time, since a very poor placement might be impossible to route, necessitating an additional placement and routing attempt.

Although logic synthesis, technology mapping, and logic block packing are considered absolutely necessary parts of a modern, general-use FPGA compiler flow, the majority of research into fast compilation has been focused on efficient placement and routing techniques. Not only do the placement and routing phases make up a large portion of the overall mapping runtime, in some cases the other steps can be considered either unsuitable or unnecessary to accelerate. Sometimes high-level synthesis and technology mapping may be unnecessary because designs are assumed to be implemented in low-level languages, or it is assumed that they can be performed offline and thus outside the task's critical path. Furthermore, although logic synthesis and technology mapping can be very difficult problems by themselves, they are also common to all hardware CAD tools—not just FPGA-based technologies. On the other hand, placement and routing tools for reconfigurable devices have to deal with architectural restrictions not present in conventional standard cell tools, and thus generally must be accelerated with unique approaches.

20.1 ACCELERATING CLASSICAL TECHNIQUES

An obvious starting point to improve the runtime of netlist compilation is to make minor algorithmic changes to accelerate the classical techniques already in use. For example, simulated annealing placement has some obvious parameters that can be changed to reduce overall runtime. The initial annealing temperature

can be lowered, the freezing point can be increased, the cooling schedule can be accelerated, or the number of moves per iteration can be reduced. These approaches all tend to speed up the annealing, but at some cost to placement quality.

20.1.1 Accelerating Simulated Annealing

Because of the adaptive nature of modern simulated annealing temperature schemes, any changes made to the structure of the cooling schedule itself can have unreliable runtime behavior. Not only have the settings of initial and final temperatures been carefully selected to thoroughly explore the solution space, changing these values may dramatically affect final placement quality while still not guaranteeing satisfactorily shorter runtime.

As described in Chapter 14, VPR updates the current temperature based on the fraction of moves accepted out of those attempted during a given iteration. Thus, decreasing the initial temperature cuts off the phase in which sweeping changes can easily occur early in the annealing. Simply starting the system at a lower initial temperature may cause the annealing to compensate by lingering longer at moderately high temperatures. Similarly, modifying the cooling schedule to migrate toward freezing faster fundamentally goes against the basic premise of simulated annealing itself. This will have an unpredictable, and likely undesirable, effect on solution quality.

It is generally accepted that the most predictable way to scale simulated annealing effort is by manipulating the number of moves attempted per temperature iteration. For example, in VPR the number of moves in a given iteration is always based on the size of the input netlist: $O(n^{1.33})$. The annealing effort is simply adjusted by scaling up or down the multiplicative constant portion of this value. In VPR, the "fast" placement option simply divides the default value by 10, which in testing indeed reduces the overall placement time by a factor of 10 while affecting final circuit quality by less than 10 percent [3]. Furthermore, as shown by Mulpuri and Hauck [12], simply changing the number of moves per iteration allows a continuous and relatively predictable spectrum of placement effort versus placement quality results.

Haldar and colleagues [11] exploited a very similar phenomenon to reduce mapping time by distributing the simulated annealing effort across multiple processors. In the strictest sense, simulated annealing is very difficult to parallelize because it attempts sequential changes to a given placement in order to slowly improve the overall wirelength. To be most faithful to this process while attempting multiple changes simultaneously, different processors must try non-overlapping changes to the system; otherwise, multiple processors may try to move the same block to two different locations or two different blocks to the same location. Not only is this type of coordination typically very difficult to enforce, it also generally requires a large amount of communication between processors. Since all processors begin each move operating on the same placement, they all must communicate any changes that are made after each step. However, a slightly less faithful but far simpler approach can take advantage of

the idea that reducing the number of moves attempted per temperature iteration can gracefully reduce runtime.

In this case, all of the processors agree upon a single placement to begin a temperature iteration. At this point, though, each processor performs simulated annealing independently of the others. To reduce the overall runtime, given N processors, each only attempts $1/N$ of the originally intended moves per iteration. At the end of the iteration, the placements discovered by all of the processors are compared and the best one is broadcasted to the rest for use during the next iteration. This greatly reduces the communication overhead and produces nearly linear speedup for two to four processors while reducing placement quality by only 10 to 25 percent [11].

Wrighton and DeHon [19] also parallelized the simulated annealing process, but approached the problem in a completely different manner. In this case, instead of attempting to develop parallel software, they actually configure an FPGA to find its own placement for a netlist. They divide a large array into distinct processing elements that will each keep track of one node in a small netlist. In their testing, the logic required to trace the inputs and outputs of a single LUT required approximately 400 LUTs. Because every processing element represents the logic held at a single location in the array, a large emulation system consisting of approximately 400 FPGAs can place a netlist for one device at a time, or one large FPGA can place a netlist requiring approximately 1/400 of the array.

Each processing element is responsible for keeping track of both the block in the netlist currently mapped to that location and the position of the sinks of the net sourced by this block. During a given timestep, each processing element determines the wirelength of its output net by evaluating the location of all of its sinks; the entire system is then perturbed in parallel by allowing each location to negotiate a possible swap with its neighbors. Just as in conventional simulated annealing, good moves are always accepted and bad moves are accepted with a probability dependent on the annealing temperature and how much worse the move makes the system as a whole. Similarly, although swaps can only be made one nearest neighbor to another, any block can eventually migrate to any other location in the array through multiple swaps. The system avoids having two blocks attempt to occupy the same location by always negotiating swaps pairwise.

As shown in Figure 20.1, a block negotiates a swap with each of its neighbors in turn. Phases 1 and 2 may swap blocks to the left or right, while phases 3 and 4 may swap with a neighbor above or below.

We should note that although very similar to the classical simulated annealing model, this arrangement does not necessarily calculate placement cost in the same way. The net bounding box calculated at each timestep cannot take into account the potential simultaneous movement of all the other blocks to which it is connected. That said, whatever inaccuracies might be introduced by this computation difference are relatively small.

Of much greater importance is the problem caused by communication bandwidth. It is possible that in a given timestep every processing element decides to swap with its neighbor. If this is the case, the location of all sinks will change.

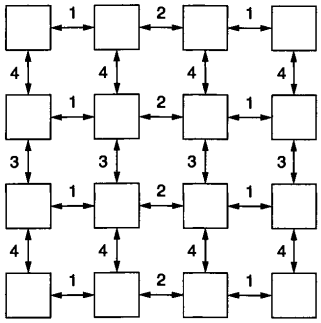


FIGURE 20.1 ■ Swap negotiation in hardware-assisted placement. (*Source*: Based on an illustration in Wrighton and DeHon [19]).

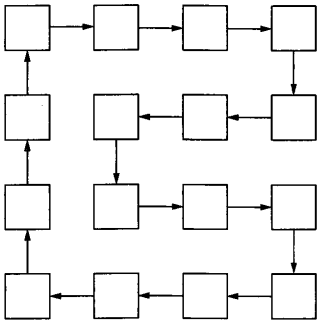


FIGURE 20.2 ■ Location update chain. (*Source*: Based on an illustration in Wrighton and DeHon [19]).

To keep completely consistent recordkeeping with conventional simulated annealing, this requires each processing element to notify its nets' sources of the block's new location. Of course, this creates a huge communication overhead. However, this can be avoided if the processing elements are allowed to calculate wirelength based on stale location information.

As shown in Figure 20.2, instead of a huge broadcast each time a block is relocated, position information marches through the system in a linear fashion. As blocks are moved during the annealing process, new positions for each one are communicated to other blocks via a dedicated location update chain. Thus, if the system has N processing elements, it might take N clock cycles before all relevant processing elements see the new placement of that block. Since the

processing elements are still calculating further moves, this means up to N cycles of stale data. Because of these inaccuracies, compared with a fast VPR run, this hardware-based simulated annealing system generally requires 36 percent more routing tracks to implement the same circuits. However, it also is three to four orders of magnitude faster.

As mentioned earlier, classical simulated annealing techniques have been very carefully tuned to produce high-quality placements. Most of the methodologies we have covered to accelerate simulated annealing rely on reducing the number of moves attempted. Thus, while they can produce reasonable placements quickly for current circuits, they do not necessarily perform well for all applications.

Mulpuri and Hauck [12] demonstrated that, while we may be able to reduce the number of moves per temperature iteration by a factor of 10 with little effect on routability, if we continue to reduce the placement effort, the quality of the placement drops off severely. The conclusion to be drawn is that, acceleration approaches, although reasonable for dealing with FPGA scaling in the short term, are not a permanent solution. Applying them on increasing netlist and device sizes will eventually lead to worse and worse placements, and, furthermore, they simply do not have the capability to produce useable placements quickly enough for either runtime netlist compilation or most instance-specific circuits.

On the other hand, hardware-assisted simulated annealing seems far more promising. Although this technique introduces some inaccuracy in cost calculation because of both simultaneously negotiated moves and stale location information, the effect of these factors is relatively predictable. The error introduced by simultaneous moves will always be relatively small because all swaps are performed between nearest neighbors. Also, the error introduced by stale location information scales linearly with netlist size. This means not only that such information will likely cause the placement quality to degrade gracefully but also that we can reduce this inaccuracy relatively easily by adding additional update paths, perhaps even a bidirectional communication network that quickly informs both forward and backward neighbors of a moved element. Since we hope that the majority of nets will cover a relatively small area, this should considerably reduce inaccurate cost calculation due to stale location information.

These trade-offs make hardware-assisted annealing an interesting possibility. Although it may impose a significant quality cost, that cost may not grow with increased system capacity, and it may be one of the only approaches that provide the drastic speedups necessary for both runtime netlist compilation and instance-specific circuits. This may make it of particular interest for future nanotechnology systems (see Chapter 38).

20.1.2 Accelerating PathFinder

Just as in placement, minor alterations can be made to classical routing algorithms to improve their runtime. Some extremely simple modifications may speed routing without affecting overall quality, or they may reduce routability in a graceful and predictable manner. Swartz et al. [15] suggest sorting the nets to be routed in order of decreasing fanout instead of simply arbitrarily. Although

high fanout nets generally make up a small fraction of a circuit, they typically monopolize a large portion of the routing runtime. By routing these comparatively difficult nets first in a given iteration, they may be presented with the lowest congestion cost and thus take the most direct and easily found paths. Lower fanout nets tend to be more localized, so they can deal with congestion more easily and their search time is comparatively smaller. This tends to speed overall routing, but since no changes are made to the actual search algorithm, it is not expected to affect routability.

Conversely, Swartz et al. [15] also suggest scaling present sharing and history costs more quickly between routing iterations. As discussed in Chapter 17, PathFinder gradually increases the cost of using congested nodes to discourage sharing over multiple iterations. Increasing present sharing and history costs more aggressively emphasizes removing congestion over route exploration. This may potentially decrease achievable routability, but the system may converge on a legal routing more quickly.

One of the most effective changes that can be made to conventional Dijkstra-based routing approaches is limiting the expansion of the search. Ignoring congestion, in most island-style FPGAs it is unnecessary for a given net to use routing resources outside the bounding box formed by its terminals. Of course, congestion must be resolved to obtain a feasible mapping, but given the routing-rich nature of modern reconfigurable devices, and assuming that routing is performed on a reasonable placement, the area formed by a net's bounding box is most likely to be used.

However, traditional Dijkstra's searches expand from the source of a net evenly in all directions. Given that the source of a 2-terminal net must lie on the edge of the bounding box, this is obviously wasteful since, again ignoring congestion costs, the search essentially progresses as concentric rings—most of which lie in the incorrect direction for finding the sink. As shown in Figure 20.3, it is unlikely that a useful route will require such a meandering path. If we would like to find

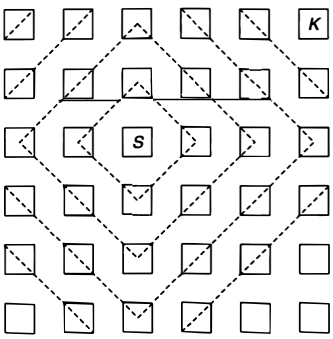


FIGURE 20.3 ■ A conventional routing search wave.

a route between blocks S and K, it is most likely that we will be able to find a direct route between them. Thus, we should direct the majority of our efforts upward and to the right before exploring downward or to the left. As described in Chapter 17, this is the motivation for adding A^* enhancements to the PathFinder algorithm. However, this concept can be taken even further by formally preventing searches from extending very far beyond the net's bounding box.

According to Betz et al. [3], a reasonable fixed limitation can prevent an exploration from visiting routing channels more than three steps outside of a net's bounding box. Although this technique may degrade routability under conditions of very high congestion, such situations may not be encountered. An architecture might have sufficient resources so that high-stress routing situations are never created, particularly in scenarios where the user is willing to reduce the amount of logic mapped to an FPGA to improve compilation runtimes.

Slightly more difficult to manage is the case of multi-terminal nets. Although the scope of a multisink search as a whole may be limited by the net's bounding box, this only alleviates one source of typically unnecessary exploration. PathFinder generally sorts the sinks of a multi-terminal net by Manhattan distance. However, each time a sink is discovered, the search for the next sink is restarted based on the entire routing tree found up to that point. As shown in Figure 20.4, this creates a wide search ring that is explored and reexplored each time a new sink is discovered, which is particularly problematic for high-fanout nets.

If we consider the new sink and the closest portion of the existing routing tree to be almost a 2-terminal net by itself, we can further reduce the amount of extraneous exploration. Swartz et al. [15] suggest splitting the bounding box of multi-terminal nets into gridlike bins. As shown in Figure 20.5, after a sink is found, a new search is launched for the next furthest sink, but explorations are only started from the portion of the routing tree contained in the bin closest to the new target. In our example, after a route to $K1$ is found, only the portion

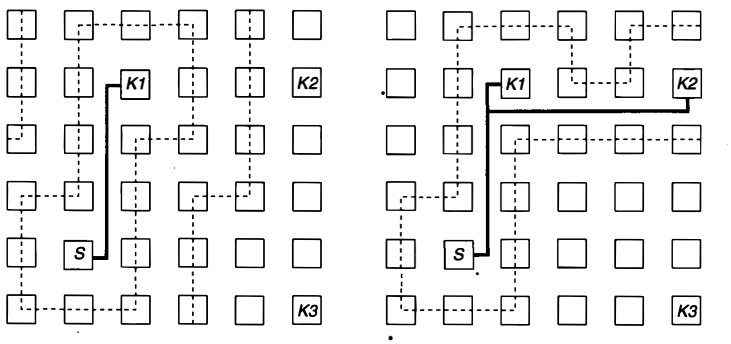


FIGURE 20.4 ■ PathFinder exploration and multi-terminal nets.

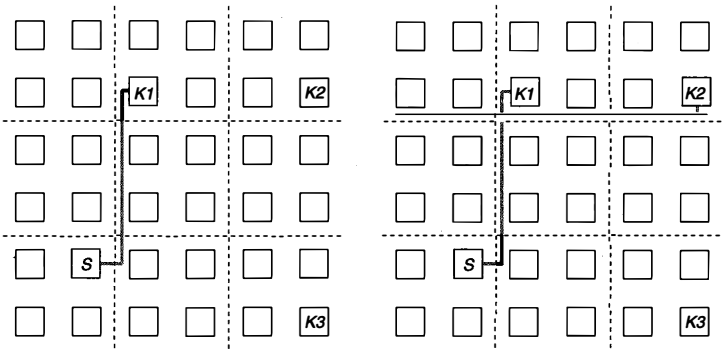


FIGURE 20.5 ■ Multi-terminal nets and region segmentation.

of the existing path in the topmost bin is used to launch a search for $K2$. The process of restricting the initialization of the search is repeated to find a route to $K3$. This may result in slightly longer branches, but, again, it is not an issue in low-stress routing situations.

Although potentially very effective, all of these techniques only attempt to improve the time required to route a single net. As described in Chapter 17, however, the PathFinder algorithm is relatively amenable to parallel processing. Chan et al. [7] showed that we can simply split the nets of a given circuit among multiple processors and allow each to route its nets mostly independently of the others. Similarly to what happens in parallel simulated annealing, complete faithfulness to the original PathFinder algorithm requires a large amount of communication bandwidth. This is because we have no guarantees that one processor will not attempt to route a signal on the same wire as another processor during a given iteration unless they are in constant communication with each other. However, because PathFinder already has a mechanism to discourage the overuse of routing resources between different nets over multiple iterations, such continuous communication is unnecessary. We can allow multiple processors to operate independently of one another for an entire routing iteration.

When all processors have routed all of their nets, we can simply determine which nodes were accidentally shared by different processors and increase their present sharing and history costs appropriately. Just as it discourages sharing between nets in classical single-processor PathFinder, this gradually discourages sharing between different processors over multiple iterations. We are using the built-in conflict-resolution mechanism in a slightly different way, but this allows us to reduce the communication overhead considerably. That said, after we have resolved the large-scale congestion in the system, the last few routing iterations likely must be performed on a single processor using conventional PathFinder.

Overall, these techniques are extremely effective on modern FPGAs. Most of today's reconfigurable architectures include a wealth of routing resources that are sufficient for a wide range of applications. Because of this, all of these approaches to accelerating PathFinder-style routing produce good results. Ordering of nets, fast growth of present sharing and history costs, and limiting the scope of exploration to net bounding boxes are common in modern FPGA routing tools. Unfortunately, however, they are still not fast enough for the most demanding applications such as runtime netlist compilation. Even the parallel technique outlined here has an unavoidable serial component. Thus, while such techniques may be adequate to produce results for next-generation FPGAs or hardware prototyping systems, they must be much faster if we are to make runtime netlist compilation practical.

20.2 ALTERNATIVE ALGORITHMS

Although classical mapping techniques have proven that they can achieve high-quality results, there is a limit to their acceleration through conventional means if we want to maintain acceptable quality for many applications. For example, in the case of placement the number of moves attempted in the inner loop of simulated annealing can only be reduced to a certain point before solution quality is no longer acceptable. While the runtime on a single processor can be cut by a factor of 10 with relatively little change in terms of routability or critical-path timing, even such modest degradation may not meet the most demanding design constraints. Furthermore, as discussed earlier, attempting to scale this technique beyond the $10x$ point generally results in markedly lower quality because the algorithm simply does not have sufficient time to adequately explore the solution space. To achieve further runtime improvements without resorting to potentially complex parallel implementations and without abandoning solution quality, we must make fundamental algorithmic changes.

20.2.1 Multiphase Solutions

One of the most popular ways to accelerate placement is to break the process into multiple phases, each handled by a different algorithm. Although many techniques use this method, a common thread among them all is that large-scale optimization is performed first by a fast but relatively imprecise algorithm. Slower, more accurate algorithms are reserved for local, small-scale refinement as a secondary step. A good example of this approach is shown in papers such as that by Xu and Kalid [20]. Here, the authors use a quadratic technique to obtain a rough placement and then work toward a better solution with a short simulated annealing phase.

In quadratic placement, the connections between blocks in the netlist are converted into linear equations, any valid solution to which indicates the position of each block. A good placement solution is found by solving the matrix equations while attempting to minimize another function: the sum of the squared

wirelength for each net. Unfortunately, one of the problems with this approach is that, in order for the equations to be solved quickly, they must be unconstrained. Thus, the placements found directly from the quadratic solver will likely have many blocks that overlap.

Xu and Kalid [20] identify these overlapping cells and, over multiple iterations, slowly add equations that force them to move apart. This is a comparatively fast process, but the additional placement legalization factors are added somewhat arbitrarily. Thus, although the quadratic placement might have gotten all of the blocks in roughly the correct area, there is still quite a bit of room for wirelength and timing improvements.

In contrast, while simulated annealing produces very good results, much of the runtime is devoted to simply making sense of a random initial placement. By combining the two approaches, and starting a low-temperature annealing only after we obtain a reasonable initial placement from the quadratic solver phase, we can drastically reduce runtime and still maintain the majority of the solution quality. Similar approaches can substitute force-directed placement for large-scale optimization or completely greedy optimization for small-scale improvement [12].

Another way to quickly obtain relatively high-quality initial placements is with partitioning-based approaches. As mentioned in Chapter 14, although recursive bipartitioning can be performed very quickly, reducing the number of signals cut by the partitions is not necessarily the same thing as minimizing wirelength or critical path delay. A similar but more sophisticated method is also discussed in Chapter 14. In hierarchical placement, as described by Sankar and Rose [13], the logical resources of a reconfigurable architecture are roughly divided into K separate regions. Multiple clustering steps then assign the netlist blocks into groups of approximately the correct size for the K logical areas. At this point, the clusters themselves can be moved around via annealing, assuming that all of the blocks in a cluster are at the center of the region.

This annealing can be performed very quickly since the number of clusters is relatively small compared to the number of logic blocks in the netlist. We can obtain a relatively good logic block-level placement by taking the cluster-level placement and decomposing it. Here, we can take each cluster in turn and arbitrarily place every block somewhere within the region assigned to it earlier. This initial placement can then be refined with a low-temperature annealing.

Purely mechanical clustering techniques are not the only way to group related logic together and obtain rough placements very quickly. In fact, the initial design specification itself holds valuable information concerning how the circuit is constructed and how it might best be laid out. Unfortunately, this knowledge is typically lost in the conventional tool flow. Regardless of whether they are using a high-level or low-level hardware description language, the organizational methods of humans naturally form top-level designs by connecting multiple large modules together. These large modules are, in turn, also created from lower-level modules. However, information about the overall design organization is generally not passed down through logical synthesis and technology mapping tools.

Packing, placement, and routing are typically performed on a completely flattened netlist of basic logic blocks. However, as suggested in works by Gehring and Ludwig and colleagues [10] and Callahan et al. [6], for example, for most applications this innate hierarchy can suggest which pieces are heavily interconnected and should be kept close together during the mapping process. Furthermore, information about multiple instances of the same module can be used to speed the physical design process.

The datapath-oriented methodology described in Chapter 15 uses a closely related concept to help design highly structured computations. In datapath composition, the entire CAD toolflow, from initial algorithm specification to floorplanning to placement, is centered on building coarse-grained objects that have obvious, simple relationships to one another. The entire computation is built from regular, snap-together tiles that can be arranged in essentially the same order in which they appear in the input dataflow graph. Although many applications simply do not fit the restrictive nature of the datapath computation model, applications that can be implemented in this way benefit greatly from the highly regular structures these tools create.

There may not be as much regularity in most applications, but we can still use organizational information to accelerate both placement and routing. At the very least, such information provides some top-level hints to reasonable clustering boundaries and can be used to roughly floorplan large designs. In some sense, this is exactly the aim of hierarchical placement, although it attempts to accomplish this without any a priori knowledge. Extending this idea, for very large systems we can use these natural boundaries to create multiple, more or less independent top-level placement problems. Even if we place each of the large system-level modules serially on a single processor, it is likely that, because of nonlinear growth in problem complexity, the total runtime will still be smaller than if we had performed one large, unified placement.

We can also employ implicit organizational information on a smaller scale in a bottom-up fashion. For example, many modern FPGAs contain dedicated fast carry-chain logic between neighboring cells. To use these structures, however, the cells must be placed in consecutive vertical logic block locations. If we were to begin with a random initial placement for a multibit adder, we would probably not find the optimal single-column placement despite the fact that, based on higher-level information, the best organization is obvious. Such very common operations can be identified and then preplaced and routed with known good solutions. These blocks then become hard macros. Less common or larger calculations can be identified and turned into soft macros. As suggested by projects such as Tessier's [17], using the high-level knowledge of macros within a hierarchical-style placement tool can improve runtime by a factor of up to 50 without affecting solution quality.

Still, while macro identification can significantly improve placement runtime, its effect on routing runtime is likely negligible. Soft macros still need to be routed because each instance may be of a different shape. Furthermore, although hard macros do not need to be repeatedly routed, and may be relatively common, their nets represent a small portion of the overall runtime because

they are typically short and are simple to route. Rather, to substantially improve routing runtime we need to address the nets that consume the largest portion of the computational effort—high-fanout nets. As discussed earlier, multi-terminal nets present a host of problems for routers such as PathFinder. In many circuits, the routing time for one or two extremely high-fanout nets can be a significant portion of the overall routing runtime. However, this effort might be unnecessary since, even though these nets are ripped up and rerouted in every iteration, they go nearly everywhere within their bounding box. This means that virtually all legal routing scenarios will create a relatively even distribution of traffic within this region and none are markedly better than any other. For this reason, we can easily route these high-fanout nets once at the beginning of the routing phase and then exclude them from following a conventional PathFinder run without seriously affecting overall routability. At the very least, if we do not want to put these nets completely outside the control of PathFinder congestion resolution, we can rip up and reroute them less frequently, perhaps every other or every third iteration.

Regardless of how the placement and routing problem is divided into simpler subproblems, multiphase approaches are the most promising way to deal with the issues associated with FPGA technology scaling. Of course, when possible it is best to gather implicit hierarchical information directly from the source hardware description language specification. This not only allows us to create both hard and soft macros very easily, but gives strong hints regarding how large designs might be floorplanned. That said, we may not have information regarding high-level module organization. In these cases we can fall back on hierarchical or partitioning placement techniques to make subsequent annealing problems much more manageable. All of these placement methodologies scale very well, and they represent algorithms that can solve the most pressing issues presented by growing reconfigurable devices and netlists.

When applicable, constructive techniques, such as the datapath-oriented methodology described in Chapter 15, or macro-based approaches can be very useful for mapping hardware prototyping systems and instance-specific circuits. These methodologies naturally produce reasonable placements very quickly. Because hardware emulation systems and instance-specific circuits do not necessarily need optimal area or timing results, these techniques often produce placements that can be used directly without the need for subsequent refinement steps.

20.2.2 Incremental Place and Route

Incremental placement and routing techniques attempt to reduce compilation time by combining and extending the same ideas exploited by multiphase compilation approaches: (1) begin with a known reasonable placement and (2) avoid ripping up and rerouting as many nets as possible.

In many situations, multiple similar versions of a given circuit might be placed and routed several times. In the case of hardware emulation, for example, it is unlikely that large portions of the circuit will change between consecutive

designs. Far more likely is that small bug fixes or local modifications will be made to specific portions of the circuit, leaving the vast majority of the design completely unchanged. Incremental placement and routing methodologies identify those portions of a circuit that have not changed from a previous mapping and attempt to integrate the changed portions in the least disruptive manner. This allows successive design updates to be compiled very quickly and minimizes the likelihood of dramatic changes to the characteristics of the resultant mapping.

The key to incremental mapping techniques is to modify an existing placement as little as possible while still finding good locations for newly introduced parts. The largest hurdle to this is merely finding a legal placement for all new blocks. If the changes reduce the overall size of the resulting circuit, any new logic blocks can simply fit into the void left by the old section. However, if the overall design becomes larger, the mapping process is more complex. Although the extra blocks can simply be dropped into any available location on the chip, this will probably result in poor timing and routability. Thus, incremental mapping techniques generally use simple algorithms to slightly move blocks and make vacant locations migrate toward the modified sections of the circuit.

The most basic approaches, such as those described by Choy et al. [4], determine where the closest empty logic block locations are and then simply slide intervening blocks toward these vacancies to create space where it is needed. Singh and Brown [14] use a slightly more sophisticated approach that employs a stochastic hill-climbing methodology, similar to a restricted simulated annealing run. This algorithm takes into account where additional resources are needed, the estimated critical path of the circuit, and the estimated required wirelength. In this way, logic blocks along noncritical paths will preferentially be moved to make room for the added logic.

Incremental techniques not only speed up the placement process, but can accelerate routing as well. Because so much of the placement is not disturbed, the nets associated with those logic blocks do not necessarily have to be rerouted. Initially, the algorithm can attempt to route only the nets associated with new or moved logic blocks. If this fails, or produces unacceptable timing results, the algorithm can slowly rip up nets that travel through congested or heavily used areas and try again. Either way, it will likely need to reroute only a very small portion of the overall circuit.

Unfortunately, there are many situations in which we do not have the prior information necessary to use incremental mapping techniques. For example, the very first compilation of a netlist must be performed from scratch. Furthermore, it is a good idea to periodically perform a complete placement and routing run, because applying multiple local piecework changes, one on top of another, can eventually lead to disappointing global results. However, as mentioned earlier, incremental compilation is ideal for hardware prototyping systems because they are typically updated very frequently with minor changes. This behavior also occurs in many other development scenarios, which is why incremental compilation is a common technique to accelerate the engineering/debugging design loop.

However, there are some situations in which it is very difficult to apply incremental approaches. For example, these techniques rely on the ability to determine what portions of a circuit do or do not change between design revisions. Not only can merely finding these similarities be a difficult problem, we must also be able to carefully control how high-level synthesis, technology mapping, and logic block packing are performed. These portions of the mapping process must be aware when incremental placement and routing is going to be attempted, and when major changes have been made to the netlist and placement and routing should be attempted from scratch.

20.3 EFFECT OF ARCHITECTURE

Although we have considered many algorithmic changes that can improve compilation runtime, we should also consider the underlying reasons that the FPGA mapping problem is so difficult. Compared to standard cell designs, FPGAs are much more restrictive because the logic and routing are fixed. Technology mapping must target the lookup tables (LUTs) and small computational cores available on a given device, placement must deliver a legal arrangement that coincides with the array of provided logic blocks, and routing must contend with a fixed topology of communication resources.

For these reasons, the underlying architecture of a reconfigurable device strongly affects the complexity of design compilation. For example, routing on a device that had an infinite number of extremely fast and flexible wires in the communication network would be easy. Every signal could simply take its shortest preferred path, and routing could be performed in a single Dijkstra's pass. Furthermore, placement would also be obvious on such an architecture since even a completely arbitrary arrangement could meet design constraints. Granted, real-world physical limitations prevent us from developing such a perfect device, but we can reduce the necessary CAD effort with smart architectural design that emphasizes ease of compilation—potentially even over logic capacity and clock speed.

The Plasma architecture [2] is a good example of designing an FPGA explicitly for simple mapping. Plasma was developed as part of the Teramac project [1]—an extremely large reconfigurable computing system slated to contain hundreds or thousands of individual FPGAs. Even given that a large design would be separated into smaller pieces that could be mapped onto individual FPGAs, contemporary commercial reconfigurable devices required tens of minutes to complete placement and routing for each chip. To further compound this issue, even after placement was completed once, there was no guarantee that all of the signals could be successfully routed, so the entire process might have to be repeated. This meant that a design that utilized thousands of conventional FPGAs could require days or weeks of overall compilation time. For the Teramac system to be useful in applications such as hardware prototyping, in which design changes might be made on a daily or even hourly basis, mapping had to be orders of

magnitude faster. Thus, the Plasma FPGA architecture was designed explicitly with fast mapping in mind.

Although Plasma differed from contemporary commercial FPGAs in several key ways, its most important distinction was high connectivity. Plasma was built from 6-input, 2-output logic blocks connected hierarchically by two levels of crossbars. As seen in Figure 20.6, logic blocks are separated into groups of 16 that are connected by a full crossbar that spans half the width of the chip. These groups are then connected to other groups by a central partial crossbar. The central vertical lines span a quarter of the height of the array, but have the capability to be connected together to span the entire distance. Since full crossbars would

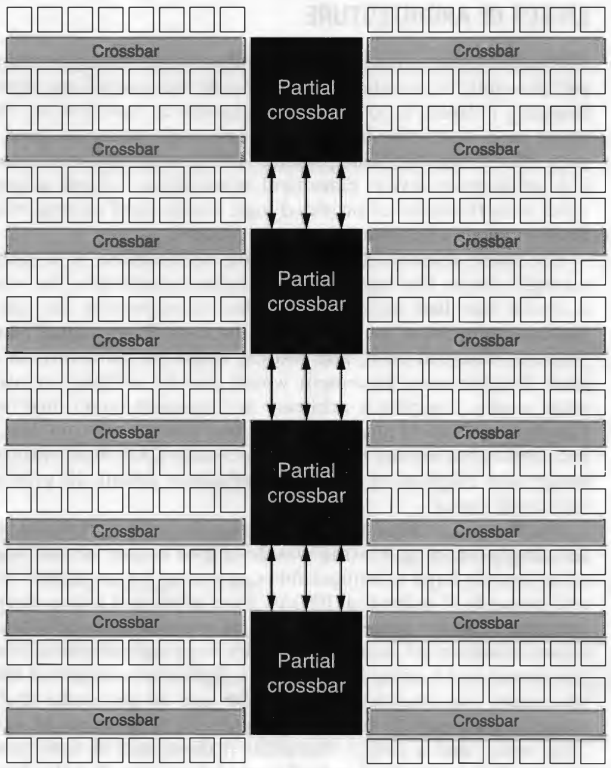


FIGURE 20.6 ■ The Plasma interconnect network.

have been prohibitively large, the developers used empirical testing to determine what level of connectivity was typically used in representational benchmarks. In addition to high internal connectivity, Plasma also contained an unusually large number of off-chip I/O pins.

Although this extremely dense routing fabric consumed 90 percent of the overall area, and its reliance on very long wires reduced the maximum operating frequency considerably, placement and routing could reliably be performed on the order of seconds on existing workstations. Given Teramac's target applications, the dramatic increase in compilation speed and the extremely consistent place and route success rate was considered to be more important than logical density or execution clock frequency.

Of course, not all applications can make such an extreme trade-off between ease of compilation and general usability metrics. However, manipulating the architecture of an FPGA does not necessarily require dramatically altering the characteristics of the device. For example, it is possible to make small changes to the interconnect to make routing simpler. One possibility is using a track domain architecture, which restricts the structure of the switch boxes in an island-style FPGA.

As shown in Figure 20.7, the connectivity of an architecture's switch boxes can affect routability. While each wire in both the top and bottom switch boxes have the same number of fanouts, the top switch box allows tracks to switch wire domains, eventually migrating to any track through multiple switch points.

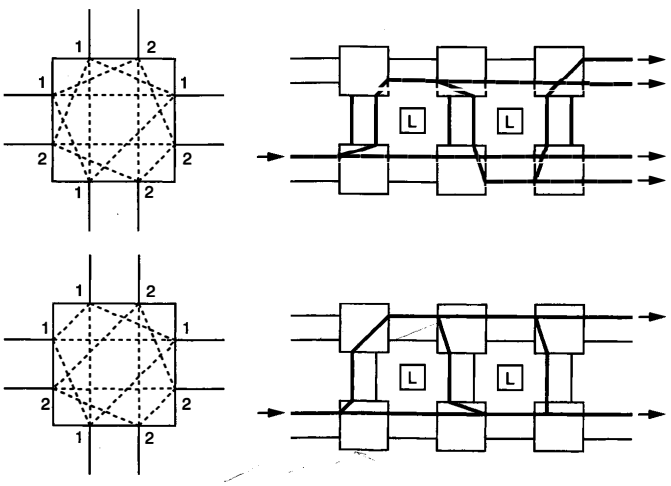


FIGURE 20.7 ■ Switch box style and routability.

This allows a signal coming in on one wire on the left of the top architecture to reach all four wires exiting the right. However, the symmetric switch box shown on the bottom does not allow tracks to switch wire domains and forces a signal to travel along a single class of wire. This means that a signal coming in from the left of the bottom architecture can only reach two of the four wires exiting to the right. Although this may reduce the flexibility of the routing fabric somewhat [18], potentially requiring more wires to achieve the same level of routability [8], this effect is relatively minor.

Even though we may need to increase the channel width of our architecture because of the restrictive nature of track domain switch boxes, routing on this type of FPGA can be dramatically faster than on more flexible systems. As shown by Cabral et al. [5], since the routing resources on track domain FPGAs are split into M different classes of wire, routing becomes a parallel problem. First, N processors are each assigned a small number of track domains from a given architecture. Then the nets from a circuit placed onto the architecture are simply split into N groups. Because each track domain is isolated from every other due to the nature of the architecture, each processor can perform normal PathFinder routing without fear that the paths found by one processor will interfere with the paths found by another. When a processor cannot route a signal on its allotted routing resources, it is given an additional unassigned track domain. Although load balancing between processors and track domains is somewhat of a problem, this technique has shown linear or even super-linear speedup with a very small penalty to routability. In this case, Cabral and colleagues [5] were able to solve the problems encountered by the parallel routing approaches that were discussed earlier by modifying the architecture itself.

Another way to modify the physical FPGA to speed routing is by offering specialized hardware to allow the device to route its own circuits. Although similar to the approach discussed earlier in which simulated annealing is implemented on a generic FPGA to accelerate the placement of its own circuits, DeHon et al. [9] suggest that by modifying the actual switch points internal to an FPGA, we can create a specialized FPGA that can assist a host processor to perform PathFinder-like routing by performing its own Dijkstra searches. In this type of architecture, the switch points have additional hardware that gives them the ability to remember the inputs and outputs currently being used when the FPGA is put into a special compilation time-only "routing search" mode.

After the placement of a given circuit is found, we configure the FPGA to perform routing on itself. This begins by clearing the occupancy markers on all of the switch points. During the routing phase, the host processor requests that each net in turn drive a signal from its source, which helps discover a path to each of its sinks. Every time this signal encounters a switching element, the switch allows the signal to propagate though unallocated resources but prevents it from continuing along occupied segments. In this way, the device explores all possible paths virtually instantaneously. When a route is found between the source and a sink, the switch point occupancy markers along this path are updated to reflect the "taken" status of these resources. When a route cannot be found for a given net, because all of the legal paths have been occupied

by earlier nets, the system simply victimizes a random previously routed path and rips it up until the blocked net can successfully route. Nets are continuously routed and ripped up in this round-robin fashion until all nets have been routed. Although this approach does not have the same sophistication as PathFinder, the experiments by DeHon and colleagues [9] show that hardware-assisted routing can obtain extremely similar track counts (only 1 to 2 additional tracks) with 4 to 6 orders of magnitude speedup in terms of runtime on the largest benchmarks.

Of course, modifying an FPGA architecture can involve a great deal of engineering effort. For example, while hardware-assisted routing is one of the only approaches that is fast enough to make runtime netlist compilation feasible, it involves completely redesigning the communication network. That said, not all of our architecture modifications need to be that drastic. For example, commercial FPGA manufacturers have already made modifications to their architectures that accelerate routing. As mentioned earlier, commercial FPGAs offer a resource-rich, flexible routing fabric to support a wide range of applications. Their high bandwidth and connectivity naturally make the routing problem simpler and much faster to solve. Following this logic, it seems natural that FPGAs might switch to track domain architectures in the future. While such devices require only minor layout changes that slightly affect overall system routability, they enable very simple parallel routing algorithms to be used. This becomes more and more important as reconfigurable devices scale and as multi-threaded and multicore processors gain popularity.

20.4 SUMMARY

In this chapter we explored many techniques to accelerate FPGA placement and routing. Ultimately, all of them have restrictions, benefits, and drawbacks. This means that our applications, architectures, and design constraints must dictate which methodologies can and should be used. Several of the approaches do not provide acceptable runtime given problem constraints, while some may not offer sufficient implementation quality. Some techniques may not scale adequately to address our issues, while we may not have the necessary information to use others.

FPGA scaling. Although classical block-level simulated annealing techniques have been the cornerstone of FPGA CAD tools for decades, these methodologies must eventually be replaced. Hierarchical and macro-based techniques seem to scale much more gracefully while preserving the large-scale characteristics of high-quality simulated annealing. On the other hand, routing will likely depend on PathFinder and other negotiated congestion techniques for quite some time. That said, for compilation time to keep pace given newer and larger devices, FPGA developers need to make some architectural changes that simplify the routing problem. Track domain

systems seem to be a natural solution given that modern desktops and workstations offer multiple types of parallel processing resources.

Hardware prototyping and logic emulation systems. While these systems benefit greatly from incremental mapping techniques, they still require fast place and route algorithms when compilation needs to be performed from scratch. Hardware-assisted placement seems an obvious choice that can take full advantage of the multichip arrays present in these large devices. Furthermore, since optimal critical-path timing is not essential and application source code is generally available to provide hierarchical information, datapath and macro-based approaches can be very effective.

Instance-specific designs. Datapath and macro-based approaches are even more important to instance-specific circuits because they cannot take advantage of many other techniques. However, the limited scope of these problems and the dramatic speedup made possible by these systems also make specialized architectures attractive. While the overhead imposed by architectures such as Plasma may not be practical for most commercial devices, these drawbacks are far less important to instance-specific circuits given the significant CAD tool benefits.

Runtime netlist compilation. Reconfigurable computing systems that require runtime netlist compilation present an incredibly demanding real-time compilation problem. Correspondingly, these systems require the most aggressive architectural approaches to make this possible. Radical system-wide modifications that provide huge amounts of routing resources significantly simplify the placement problem. However, just providing more bandwidth does not necessarily accelerate the routing process. These systems need to provide communication channels that either do not need to be negotiated or, through hardware-assisted routing, can automatically negotiate their own connections. An open question is whether the advantages of runtime netlist compilation are worth the attendant costs and complexities they introduce.

References

[1] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider. Teramac—configurable custom computing. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[2] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider, L. Albertson. Plasma: An FPGA for million gate systems. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 1996.

[3] V. Betz, J. Rose, A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic, 1999.

[4] C. Choy, T. Cheung, K. Wong. Incremental layout placement modification algorithm. *IEEE Transactions on Computer-Aided Design* 15(4), April 1996.

[5] L. Cabral, J. Aude, N. Maculan. TDR: A distributed-memory parallel routing algorithm for FPGAs. *Proceedings of International Conference on Field-Programmable Logic and Applications*, 2002.

[6] T. Callahan, P. Chong, A. Dehon, J. Wawrynek. Fast module mapping and placement for datapaths in FPGAs. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 1998.

[7] P. Chan, M.D.F. Schlag, C. Ebeling. Distributed-memory parallel routing for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design* 19(8), August 2000.

[8] Y. Chang, D. F. Wong, C. K. Wong. Universal switch modules for FPGA design. *ACM Transactions on Design Automation of Electronic Systems* 1(1), January 1996.

[9] A. DeHon, R. Huang, J. Wawrzynek. Hardware-assisted fast routing. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 2002.

[10] S. Gehring, S. Ludwig. Fast integrated tools for circuit design with FPGAs. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 1998.

[11] M. Haldar, M. A. Nayak, A. Choudhary, P. Banerjee. Parallel algorithms for FPGA placement. *Proceedings of the Great Lakes Symposium on VLSI*, 2000.

[12] C. Mulpuri, S. Hauck. Runtime and quality trade-offs in FPGA placement and routing. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 2001.

[13] Y. Sankar, J. Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays*, 1999.

[14] D. Singh, S. Brown. Incremental placement for layout-driven optimizations on FPGAs. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2002.

[15] J. Swartz, V. Betz, J. Rose. A fast routability-driven router for FPGAs. *Proceedings of the ACM Symposium on Field-Programmable Gate Arrays*, 1998.

[16] R. Tessier. Negotiated A* routing for FPGAs. *Proceedings of the Canadian Workshop on Field-Programmable Devices*, 1998.

[17] R. Tessier. Fast placement approaches for FPGAs. *Transactions on Design Automation of Electronic Systems* 7(2), April 2002.

[18] S. Wilton. *Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*, Ph.D. thesis, University of Toronto, 1997.

[19] M. Wrighton, A. DeHon. Hardware-assisted simulated annealing with application for fast FPGA placement. *Proceedings of the ACM Symposium on Field-Programmable Gate Arrays*, 2003.

[20] Y. Xu, M.A.S. Kalid. QPF: Efficient quadratic placement for FPGAs. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2005.

PART IV

APPLICATION DEVELOPMENT

Creating an efficient FPGA-based computation is similar to creating any other hardware. A designer carefully optimizes his or her computation to the needs of the underlying technology, exploiting the parallelism available while meeting resource and performance constraints. These designs are typically written in a hardware description language (HDL), such as Verilog, and CAD tools are then used to create the final implementation.

Field-programmable gate arrays (FPGAs) do have unique constraints and opportunities that must be understood in order for this technology to be employed most effectively. The resource mix is fixed, and the devices are never quite fast enough or have high enough capacity for what we want to do. However, because the chips are reprogrammable we can change the system in response to bugs or functionality upgrades, or even change the computation as it executes.

Because of the unique restrictions and opportunities inherent in FPGAs, a set of approaches to application development have proven critical to exploiting these devices to the fullest. Many of them are covered in the chapters that follow. Although not every FPGA-based application will use each of the approaches, a true FPGA expert will make them all part of his or her repertoire.

Some of the most challenging questions in the design process come at the very beginning of a new project: Are FPGAs a good match for the application? If so, what problems must be considered and overcome? Will runtime reconfiguration be part of the solution? Will fixed- or floating-point computation be used? Chapter 21 focuses on this level of design, covering the important issues that arise when we first consider an application and the problems that must be avoided or solved. It also offers a quick overview of application development. Chapters 22 through 26 delve into individual concerns in more detail.

FPGAs are unique in their potential to be more efficient than even ASICs for some types of problems: Because the circuit design is completely programmable, we can create a custom circuit not just for a given problem but for a specific problem *instance*. Imagine, for example, that we are creating an engine for solving Boolean equations (e.g., a SAT solver, discussed in Chapter 29 in Part V). In an ASIC design, we

would create a generic engine capable of handling any possible Boolean equation because each use of the chip would be for a different equation. In an FPGA-based system, the equation can be folded into the circuit mapping itself, creating a custom FPGA mapping optimized to solving that Boolean equation and no other. As long as there is a CPU available to dynamically create a new FPGA bitstream each time a new Boolean equation must be solved, a much more aggressively optimized design can be created. However, because this means that the time to create the new mapping is part of system execution, fast mapping algorithms are often the key (Chapter 20). This concept of instance-specific circuits is covered in Chapter 22.

In most cases, the time to create a completely new mapping in response to a specific problem instance is too long. Indeed, if it takes longer to create the custom circuit than for a generic circuit to solve the problem, the generic circuit is the better choice. However, more restricted versions of this style of optimization are still valuable. Consider a simple FIR filter, which involves multiplication of an incoming datastream with a set of constant coefficients. We could use a completely generic multiplier to handle the constant * variable computation. However, the bits of the constant are known in advance, so many parts of this multiplication can be simplified out. Multipliers, for example, generally compute a set of partial products—the result of multiplying one input with a single bit of the other input. These partial products are then added together. If the constant coefficient provided that single bit for a partial product, we can know at mapping creation time whether that partial product will be 0 or equal to the variable input—no hardware is necessary to create it. Also, in cases where the partial product is a 0, we no longer need to add it into the final result. In general, the use of constant inputs to a computation can significantly improve most metrics in FPGA mapping quality. These techniques, called constant propagation and partial evaluation, are covered in Chapter 22.

Number formats in FPGAs are another significant concern. For microprocessor-based systems we are used to treating everything as a 64-bit integer or an IEEE-format floating-point value. Because the underlying hardware is hardcoded to efficiently support these specific number formats, any other format is unlikely to be useful. However, in an FPGA we custom create the datapath. Thus, using a 64-bit adder on values that are at most 18 bits in length is wasteful because each bit position consumes one or more lookup tables (LUTs) in the device.

For this reason, an FPGA designer will carefully consider the required wordlength of the numbers in the system, hoping to shave off some bits of precision and thus reduce the hardware requirements of the design.

Fractional values, such as π or fractions of a second, are more problematic. In many cases, we can use a fixed-point format. We might use numbers in the range of $0 \ldots 31$ to represent the values from 0 to $\frac{31}{32}$ in steps of $\frac{1}{32}$ by just remembering that the number is actually scaled by a factor of 32. Techniques for addressing each of the concerns just mentioned are treated in Chapter 23.

Sometimes these optimizations simply are not possible, particularly for signals that require a high dynamic range (i.e., they must represent both very large and very small values simultaneously), so we need to use a floating-point format. This means that each operation will consume significantly more resources than its integer or fixed-point alternatives will. Chapter 31 in Part V covers floating-point operations on FPGAs in detail.

Once the number format is decided, it is important to determine how best to perform the actual computation. For many applications, particularly those from signal processing, the computation will involve a large number of constant coefficient multiplications and subsequent addition operations, such as in finite impulse response (FIR) filters. While these can be carried out in the normal, parallel adders and multipliers from standard hardware design, the LUT-based logic of an FPGA allows an even more efficient implementation. By converting to a bit–serial dataflow and storing the appropriate combination of constants into the LUTs in the FPGA, the multiply–accumulate operation can be compressed to a small table lookup and an addition. This technique, called distributed arithmetic, is covered in Chapter 24. It is capable of providing very efficient FPGA-based implementations of important classes of digital signal processing (DSP) and similar operations.

Complex mathematical operations such as sine, cosine, division, and square root, though less common than multiply–add, are still important in many applications. In some cases they can be handled by table lookup, with a table of precomputed results stored in memories inside the FPGA or in attached chips. However, as the size of the operand(s) for these functions grows, the size of the memory explodes, limiting this technique's effectiveness. A particularly efficient alternative in FPGA logic is the CORDIC algorithm. By the careful creation of an iterative circuit, FPGAs can efficiently compute many of these complex functions. The full details of the CORDIC algorithm, and its implementation in FPGAs, are covered in Chapter 25.

A final concern is the coupling of both FPGAs and central processing units (CPUs). In early systems, FPGAs were often deployed together with microprocessors or microcontrollers, either by placing an FPGA card in a host PC or by placing both resources on a single circuit board. With modern FPGAs, which can contain complete microprocessors

(either by mapping their logic into LUTs or embedding a complete microprocessor into the chip's silicon layout), the coupling of CPUs and FPGAs is even more attractive. The key driver is the relative advantages of each technology. FPGAs can provide very high performance for streaming applications with a lot of data parallelism—if we have to apply the same repetitive transformation to a large amount of data, an FPGA's performance is generally very high. However, for more sequential operations FPGAs are a poor choice. Sometimes long sequences of operations, with little or no opportunity for parallelism, come up in the control of the overall system. Also, exceptional cases do occur and must be handled—for example, the failure of a component, using denormal numbers in floating point, or interfacing to command-based peripherals. In each case a CPU is a much better choice for those portions of a computation. As a result, for many computations the best answer is to use the FPGA for the data-parallel kernels and a CPU for all the other operations. This process of segmenting a complete computation into software/CPU portions and hardware/FPGA portions is the focus of Chapter 26.

CHAPTER 21

IMPLEMENTING APPLICATIONS WITH FPGAS

Brad L. Hutchings, Brent E. Nelson
Department of Electrical and Computer Engineering
Brigham Young University

Developers can choose various devices when implementing electronic systems: field-programmable gate arrays (FPGAs), microprocessors, and other standard products such as ASSPs, and custom chips or application-specific integrated circuits (ASICs). This chapter discusses how FPGAs compare to other digital devices, outlines the considerations that will help designers to determine when FPGAs are appropriate for a specific application, and presents implementation strategies that exploit features specific to FPGAs.

The chapter is divided into four major sections. Section 21.1 discusses the strengths and weaknesses of FPGAs, relative to other available devices. Section 21.2 suggests when FPGA devices are suitable choices for specific applications/algorithms, based upon their I/O and computation requirements. Section 21.3 discusses general implementation strategies appropriate for FPGA devices. Then Section 21.4 discusses FPGA-specific arithmetic design techniques.

21.1 STRENGTHS AND WEAKNESSES OF FPGAs

Developers can choose from three general classes of devices when implementing an algorithm or application: microprocessor, FPGA, or ASIC (for simplicity, ASSPs are not considered here). This section provides a brief summary of the advantages and disadvantages of these devices in terms of time to market, cost, development time, power consumption, and debug and verification.

21.1.1 Time to Market

Time to market is often touted as one of the FPGA's biggest strengths, at least relative to ASICs. With an ASIC, from specification to product requires (at least): (1) design, (2) verification, (3) fabrication, (4) packaging, and (5) device test. In addition, software development requires access to the ASIC device (or an emulation of such) before it can be verified and completed. As immediately available standard devices, FPGAs have already been fabricated, packaged, and tested by the vendor, thereby eliminating at least four months from time to market.

More difficult to quantify but perhaps more important are: (1) refabrications (respins) caused by either errors in the design or late changes to the specification, due to a change in an evolving standard, for example, and (2) software development schedules that depend on access to the ASIC. Both of these items impact product production schedules; a respin can easily consume an additional four months, and early access to hardware can greatly accelerate software development and debug, particularly for the embedded software that communicates directly with the device.

In light of these considerations, a conservative estimate of the time-to-market advantage of FPGAs relative to ASICs is 6 to 12 months. Such a reduction is significant; in consumer electronics markets, many products have only a 24-month lifecycle.

21.1.2 Cost

Per device, FPGAs can be much less expensive than ASICs, especially in lower volumes, because the nonrecurring costs of FPGA fabrication are borne by many users. However, because of their reprogrammability, FPGAs require much more silicon area to implement equivalent functionality. Thus, at the highest volumes possible in consumer electronics, FPGA device cost will eventually exceed ASIC device cost.

21.1.3 Development Time

FPGA application development is most often approached as hardware design: applications are described in Verilog or VHDL, simulated to determine correctness, and synthesized using commercial logic synthesis tools. Commercial tools are available that synthesize behavioral programs written in sequential languages such as C to FPGAs. However, in most cases, much better performance and higher densities are achieved using HDLs, because they allow the user to directly describe and exploit the intrinsic parallelism available in an application. Exploiting application parallelism is the single best way to achieve high FPGA performance. However, designing highly parallel implementations of applications in HDLs requires significantly more development effort than software development with conventional sequential programming languages such as Java or C++.

21.1.4 Power Consumption

FPGAs consume more power than ASICs simply because programmability requires many more transistors, relative to a customized integrated circuit (IC). FPGAs may consume more or less power than a microprocessor or digital signal processor (DSP), depending on the application.

21.1.5 Debug and Verification

FPGAs are developed with standard hardware design techniques and tools. Coded in VHDL or Verilog and synthesized, FPGA designs can be debugged

in simulators just as typical ASIC designs are. However, many designers verify their designs directly, by downloading them into an FPGA and testing them in a system. With this approach the application can be tested at speed (a million times faster than simulation) in the actual operating environment, where it is exposed to real-world conditions. If thorough, this testing provides a stronger form of functional verification than simulation. However, debugging applications in an FPGA can be difficult because vendor tools provide much less observability and controllability than, for example, an hardware description language (HDL) simulator.

21.1.6 FPGAs and Microprocessors

As discussed previously, FPGAs are most often contrasted with custom ASICs. However, if a programmable solution is dictated because of changing application requirements or other factors, it is important to study the application carefully to determine if it is possible to meet performance requirements with a programmable processor—microprocessor or DSP. Code development for programmable processors requires much less effort than that required for FPGAs or ASICs, because developing software with sequential languages such as C or Java is much less taxing than writing parallel descriptions with Verilog or VHDL. Moreover, the coding and debugging environments for programmable processors are far richer than their HDL counterparts. Microprocessors are also generally much less expensive than FPGAs. If the microprocessor can meet application requirements (performance, power, etc.), it is almost always the best choice.

In general, FPGAs are well suited to applications that demand extremely high performance and reprogrammability, for interfacing components that communicate with many other devices (so-called glue-logic) and for implementing hardware systems at volumes that make their economies of scale feasible. They are less well suited to products that will be produced at the highest possible volumes or for systems that must run at the lowest possible power.

21.2 APPLICATION CHARACTERISTICS AND PERFORMANCE

Application performance is largely determined by the computational and I/O requirements of the system. Computational requirements dictate how much hardware parallelism can be used to increase performance. I/O system limitations and requirements determine how much performance can actually be exploited from the parallel hardware.

21.2.1 Computational Characteristics and Performance

FPGAs can outperform today's processors only by exploiting massive amounts of parallelism. Their technology has always suffered from a significant clock-rate disadvantage; FPGA clock rates have always been slower than CPU clock rates by about a factor of 10. This remains true today, with clock rates for FPGAs

limited to about 300 to 350 MHz and CPUs operating at approximately 3 GHz. As a result, FPGAs must perform at least 10 times the computational work per cycle to perform on par with processors. To be a compelling alternative, an FPGA-based solution should exceed the performance of a processor-based solution by 5 to 10 times and hence must actually perform 50 to 100 times the computational work per clock cycle. This kind of performance is feasible only if the target application exhibits a corresponding amount of exploitable parallelism.

The guideline of 5 to 10 times is suggested for two main reasons. First of all, prior to actual implementation, it is difficult or impossible to foresee the impact of various system and I/O issues on eventual performance. In our experience, 5 times can quickly become 2 times or less as various system and algorithmic issues arise during implementation. Second, application development for FPGAs is much more difficult than conventional software development. For that reason, the additional development effort must be carefully weighed against the potential performance advantages. A guideline of 5 to 10 times provides some insurance that any FPGA-specific performance advantages will not completely vanish during the implementation phase.

Ultimately, the intrinsic characteristics of the application place an upper bound on FPGA performance. They determine how much raw parallelism exists, how exploitable it is, and how fast the clock can operate. A review of the literature [3–6, 11, 16, 19–21, 23, 26, 28] shows that the application characteristics that have the most impact on application performance are: data parallelism, amenability to pipelining, data element size and arithmetic complexity, and simple control requirements.

Data parallelism

Large datasets with few or no data dependencies are ideal for FPGA implementation for two reasons: (1) They enable high performance because many computations can occur concurrently, and (2) they allow operations to be extensively rescheduled. As previously mentioned, concurrency is extremely important because FPGA applications must be able to achieve 50 to 100 times the operations per clock cycle of a microprocessor to be competitive. The ability to reschedule computations is also important because it makes it feasible to tailor the circuit design to FPGA hardware and achieve higher performance. For example, computations can be scheduled to maximize data reuse to increase performance and reduce memory bandwidth requirements. Image-processing algorithms with their attendant data parallelism have been among the highest-performing algorithms mapped to FPGA devices.

Data element size and arithmetic complexity

Data element size and arithmetic complexity are important because they strongly influence circuit size and speed. For applications with large amounts of exploitable parallelism, the upper limit on this parallelism is often determined by how many operations can be performed concurrently on the FPGA device. Larger data elements and greater arithmetic complexity lead to larger

and fewer computational elements and less parallelism. Moreover, larger and more complex circuits exhibit more delay that slows clock rate and impacts performance. Not surprisingly, representing data with the fewest possible bits and performing computation with the simplest operators generally lead to the highest performance. Designing high-performance applications in FPGAs almost always involves a precision/performance trade-off.

Pipelining
Pipelining is essential to achieving high performance in FPGAs. Because FPGA performance is limited primarily by interconnect delay, pipelining (inserting registers on long circuit pathways) is an essential way to improve clock rate (and therefore throughput) at the cost of latency. In addition, pipelining allows computational operations to be overlapped in time and leads to more parallelism in the implementation. Generally speaking, because pipelining is used extensively throughout FPGA-based designs, applications must be able to tolerate some latency (via pipelining) to be suitable candidates for FPGA implementation.

Simple control requirements
FPGAs achieve the highest performance if all operations can be statically scheduled as much as possible (this is true of many technologies). Put simply, it takes time to make decisions and decision-making circuitry is often on the critical path for many algorithms. Replacing runtime decision circuitry with static control eliminates circuitry and speeds up execution. It makes it much easier to construct circuit pipelines that are heavily utilized with few or no pipeline bubbles. In addition, statically scheduled controllers require less circuitry, making room for more datapath operators, for example. In general, datasets with few or no dependencies often have simple control requirements.

21.2.2 I/O and Performance

As mentioned previously, FPGA clock rates are at least one order of magnitude slower than those of CPUs. Thus, significant parallelism (either data parallelism or pipelining) is required for an FPGA to be an attractive alternative to a CPU. However, I/O performance is just as important: Data must be transmitted at rates that can keep all of the parallel hardware busy.

Algorithms can be loosely grouped into two categories: I/O bound and compute bound [17, 18]. At the simplest level, if the number of I/O operations is equal to or greater than the number of calculations in the computation, the computation is said to be I/O bound. To increase its performance requires an increase in memory bandwidth—doing more computation in parallel will have no effect. Conversely, if the number of computations is greater than the number of I/O operations, computational parallelism may provide a speedup.

A simple example of this, provided by Kung [18], is matrix–matrix multiplication. The total number of I/Os in the computation, for *n-by-n* matrices, is $3n^2$—each matrix must be read and the product written back. The total number of computations to be done, however, is n^3. Thus, this computation is

compute bound. In contrast, matrix–matrix addition requires $3n^2$ I/Os and $3n^2$ calculations and is thus I/O bound. Another way to see this is to note that each source element read from memory in a matrix–matrix multiplication is used n times and each result is produced using n multiply–accumulate operations. In matrix–matrix addition, each element fetched from memory is used only once and each result is produced from only a single addition.

Carefully coordinating data transfer, I/O movement, and computation order is crucial to achieving enough parallelism to provide effective speedup. The entire field of systolic array design is based on the concepts of (1) arranging the I/O and computation in a compute-bound application so that each data element fetched from memory is reused multiple times, and (2) keeping many processing elements busy operating in parallel on that data.

FPGAs offer a wide variety of memory elements that can be used to coordinate I/O and computation: flip-flops to provide single-bit storage (10,000s of bits); LUT-based RAM to provide many small blocks of randomly distributed memory (100,000s of bits); and larger RAM or ROM memories (1,000,000s of bits). Some vendors' FPGAs contain multiple sizes of random access memories, and these memories are often easily configured into special-purpose structures such as dynamic-length shift registers, content-addressable memories (CAMs), and so forth. In addition to these types of on-chip memory, most FPGA platforms provide off-chip memory as well.

Increasing the I/O bandwidth to memory is usually critical in harnessing the parallelism inherent in a computation. That is, after some point, further multiplying the number of processing elements (PEs) in a design (to increase parallelism) usually requires a corresponding increase in I/O. This additional I/O can often be provided by the many on-chip memories in a typical modern FPGA. The work of Graham and Nelson [8] describes a series of early experiments to map time-delay SONAR beam forming to an FPGA platform where memory bandwidth was the limiting factor in design speedup. While the data to be processed were an infinite stream of large data blocks, many of the other data structures in the computation were not large (e.g., coefficients, delay values). In this computation, it was not *the total amount of memory* that limited the speedup but rather the *number of memory ports available*. Thus, the use of multiple small memories in parallel were able to provide the needed bandwidth.

The availability of many small memories in today's FPGAs further supports the idea of trading off computation for table lookup. Conventional FPGA fabrics are based on a foundation of 4-input LUTs; in addition, larger on-chip memories can be used to support larger lookup structures. Because the memories already exist on chip, unlike in ASIC technology, using them adds no additional cost to the system. A common approach in FPGA-based design, therefore, is to evaluate which parts of the system's computations might lend themselves to table lookup and use the available RAM blocks for these lookups.

In summary, the performance of FPGA-based applications is largely determined by how much exploitable parallelism is available, and by the ability of the system to provide data to keep the parallel hardware operational.

21.3 GENERAL IMPLEMENTATION STRATEGIES FOR FPGA-BASED SYSTEMS

In contrast with other programmable technologies such as microprocessors or DSPs, FPGAs provide an extremely rich and complex set of implementation alternatives. Designers have complete control over arithmetic schemes and number representation and can, for example, trade precision for performance. In addition, reprogrammable, SRAM-based FPGAs can be configured any number of times to provide additional implementation flexibility for further tailoring the implementation to lower cost and make better use of the device.

There are two general configuration strategies for FPGAs: configure-once, where the application consists of a single configuration that is downloaded for the duration of the application's operation, and runtime reconfiguration (RTR), where the application consists of multiple configurations that are "swapped" in and out as the application operates [14].

21.3.1 Configure-once

Configure-once (during operation) is the simplest and most common way to implement applications with reconfigurable logic. The distinctive feature of configure-once applications is that they consist of a single system-wide configuration. Prior to operation, the FPGAs comprising the reconfigurable resource are loaded with their respective configurations. Once operation commences, they remain in this configuration until the application completes. This approach is very similar to using an ASIC for application acceleration. From the application point of view, it matters little whether the hardware used to accelerate the application is an FPGA or a custom ASIC because it remains constant throughout its operation.

The configure-once approach can also be applied to reconfigurable applications to achieve significant acceleration. There are classes of applications, for example, where the input data varies but remains constant for hours, days, or longer. In some cases, data-specific optimizations can be applied to the application circuitry and lead to dramatic speedup. Of course, when the data changes, the circuit-specific optimizations need to be reapplied and the bitstream regenerated. Applications of this sort consist of two elements: (1) the FPGA and system hardware, and (2) an application-specific compiler that regenerates the bitstream whenever the application-specific data changes. This approach has been used, for example, to accelerate SNORT, a popular packet filter used to improve network security [13]. SNORT data consists of regular expressions that detect malicious packets by their content. It is relatively static, and new regular expressions are occasionally added as new attacks are detected. The application-specific compiler translates these regular expressions into FPGA hardware that matches packets many times faster than software SNORT. When new regular expressions are added to the SNORT database, the compiler is rerun and a new configuration is created and downloaded to the FPGA.

21.3.2 Runtime Reconfiguration

Whereas configure-once applications statically allocate logic for the duration of an application, RTR applications use a dynamic allocation scheme that re-allocates hardware at runtime. Each application consists of *multiple* configurations per FPGA, with each one implementing some fraction of it. Whereas a configure-once application configures the FPGA once before execution, an RTR application typically reconfigures it many times during the normal operation.

There are two basic approaches that can be used to implement RTR applications: *global* and *local* (sometimes referred to as partial configuration in the literature). Both techniques use multiple configurations for a single application, and both reconfigure the FPGA during application execution. The principal difference between the two is the way the dynamic hardware is allocated.

Global RTR

Global RTR allocates *all* (FPGA) hardware resources in each configuration step. More specifically, global RTR applications are divided into distinct temporal phases, with each phase implemented as a single system-wide configuration that occupies all system FPGA resources. At runtime, the application steps through each phase by loading all of the system FPGAs with the appropriate configuration data associated with a given phase.

Local RTR

Local RTR takes an even more flexible approach to reconfiguration than does global RTR. As the name implies, these applications *locally* (or selectively) reconfigure subsets of the logic as they execute. Local RTR applications may configure any percentage of the reconfigurable resources at any time, individual FPGAs may be configured, or even single FPGA devices may themselves be partially reconfigured on demand. This flexibility allows hardware resources to be tailored to the runtime profile of the application with finer granularity than that possible with global RTR. Whereas global RTR approaches implement the execution process by loading relatively large, global application partitions, local RTR applications need load only the necessary functionality at each point in time. This can reduce the amount of time spent downloading configurations and can lead to a more efficient runtime hardware allocation.

The organization of local RTR applications is based more on a *functional* division of labor than the phased partitioning used by global RTR applications. Typically, local RTR applications are implemented by functionally partitioning an application into a set of fine-grained operations. These operations need not be temporally exclusive—many of them may be active at one time. This is in direct contrast to global RTR, where only one configuration (per FPGA) may be active at any given time. Still, with local RTR it is important to organize the operations such that idle circuitry is eliminated or greatly reduced. Each operation is implemented as a distinct circuit module, and these circuit modules are then downloaded to the FPGAs as necessary during operation. Note that, unlike global RTR, several of these operations may be loaded simultaneously, and each may consume any portion of the system FPGA resources.

RTR applications

Runtime Reconfigured Artificial Neural Network (RRANN) is an early example of a global RTR application [7]. RRANN divided the back-propagation algorithm (used to train neural networks) into three temporally exclusive configurations that were loaded into the FPGA in rapid succession during operation. It demonstrated a 500 percent increase in density by eliminating idle circuitry in individual algorithm phases.

RRANN was followed up with RRANN-2 [9], an application using local RTR. Like RRANN, the algorithm was still divided into three distinct phases. However, unlike the earlier version, the phases were carefully designed so that they shared common circuitry, which was placed and routed into identical physical locations for each phase. Initially, only the first configuration was loaded; thereafter, the common circuitry remained resident and only circuit differences were loaded during operation. This reduced configuration overhead by 25 percent over the global RTR approach.

The Dynamic Instruction Set Computer (DISC) [29] used local RTR to create a sequential control processor with a very small fixed core that remained resident at all times. This resident core was augmented by circuit modules that were dynamically loaded as required by the application. DISC was used to implement an image-processing application that consisted of various filtering operations. At runtime, the circuit modules were loaded as necessary. Although the application used all of the filtering circuit modules, it did not require all of them to be loaded simultaneously. Thus, DISC loaded circuit modules on demand as required. Only a few active circuit modules were ever resident at any time, allowing the application to fit in a much smaller device than possible with global RTR.

21.3.3 Summary of Implementation Issues

Of the two general implementation techniques, configure-once is the simplest and is best supported by commercially available tool flows. This is not surprising, as all FPGA CAD tools are derivations of conventional ASIC CAD flows. While the two RTR implementation approaches (local and global) can provide significant performance and capacity advantages, they are much more challenging to employ, primarily because of a lack of specific tool support.

The designer's primary task when implementing global RTR applications is to temporally divide the application into roughly equal-size partitions to efficiently use reconfigurable resources. This is largely a manual process—although the academic community has produced some partitioning tools, no commercial offerings are currently available. The main disadvantage of global RTR is the need for equal-size partitions. If it is not possible to evenly partition the application, inefficient use of FPGA resources will result.

The main advantage of local RTR over global RTR is that it uses fine-grained functional operators that may make more efficient use of FPGA resources. This is important for applications that are not easily divided into equal-size temporally exclusive circuit partitions. However, partitioning a local RTR design may require an inordinate amount of designer effort. For example, unlike global

RTR, where circuit interfaces typically remain fixed between configurations, local RTR allows these interfaces to change with each configuration. When circuit configurations become small enough for multiple configurations to fit into a single device, the designer needs to ensure that all configurations will *interface* correctly one with another. Moreover, the designer may have to ensure not only structural compliance but *physical* compliance as well. That is, when the designer creates circuit configurations that do not occupy an entire FPGA, he or she will have to ensure that the physical footprint of each is compatible with that of others that may be loaded concurrently.

21.4 IMPLEMENTING ARITHMETIC IN FPGAs

Almost since their invention, FPGAs have employed dedicated circuitry to accelerate arithmetic computation. In earlier devices, dedicated circuitry sped up the propagation of carry signals for ripple-carry, full-adder blocks. Later devices added dedicated multipliers, DSP function blocks, and more complex fixed-function circuitry. The presence of such dedicated circuitry can dramatically improve arithmetic performance, but also restricts designers to a very small subset of choices when implementing arithmetic.

Well-known approaches such as carry-look-ahead, carry-save, signed-digit, and so on, generally do not apply to FPGAs. Though these techniques are commonly used to create very high-performance arithmetic blocks in custom ICs, they are not competitive when applied to FPGAs simply because they cannot access the faster, dedicated circuitry and must be constructed using slower, general-purpose user logic. Instead, FPGA designers accelerate arithmetic in one of two ways with FPGAs: (1) using dedicated blocks if they fit the needs of the application, and (2) avoiding the computation entirely, if possible. Designers apply the second option by, for example, replacing full-blown floating-point computation with simpler, though not equivalent, fixed-point, or block floating-point, computations. In some cases, they can eliminate multiplication entirely with constant propagation. Of course, the feasibility of replacing slower, complex functions with simpler, faster ones is application dependent.

21.4.1 Fixed-point Number Representation and Arithmetic

A fixed-point number representation is simply an integer representation with an implied binary point, usually in 2's complement format to enable the representation of both positive and negative values. A common way of describing the structure of a fixed-point number is to use a tuple: n, m, where n is the number of bits to the left of the binary point and m is the number of bits to the right. A 16.0 format would thus be a standard 16-bit integer; a 3.2 format fixed-point number would have a total of 5 bits with 3 to the left of the implied binary point and 2 to the right. A range of numbers from +1 to −1A is common in digital signal-processing applications. Such a representation might be of the

form 1.9, where the largest number is 0.111111111 = 0.99810 and the smallest is 1.000000000 = -1_{10}. As can be seen, fixed-point arithmetic exactly follows the rules learned in grade school, where lining up the implied binary point is required for performing addition or subtraction.

When designing with fixed-point values, one must keep track of the number format on each wire; such bookkeeping is one of the design costs associated with fixed-point design. At any point in a computation, either truncation or rounding can be used to reduce the number of bits to the right of the binary point, the effect being to simply reduce the precision with which the number is represented.

21.4.2 Floating-point Arithmetic

Floating-point arithmetic overcomes many of the challenges of fixed-point arithmetic but at increased circuit cost and possibly reduced precision. The most common format for a floating-point number is of the form *seeeeeffffff*, where *s* is a sign bit, *eeeee* is an exponent, and *ffffff* is the mantissa. In the IEEE standard for single-precision floating point, the number of exponent bits is 8 and the number of mantissa bits is 23, but nonstandard sizes and formats have also been used in FPGA work [2, 24].

IEEE reserves various combinations of exponent and mantissa to represent special values: zero, not a number (NAN), infinity (+8 and −8), and so on. It supports denormalized numbers (no leading implied 1 in the mantissa) and flags them using a special exponent value. Finally, the IEEE specification describes four rounding modes. Because supporting all special case number representations and rounding modes in hardware can be very expensive, FPGA-based floating-point support often omits some of them in the interest of reducing complexity and increasing performance.

For a given number of bits, floating point provides extended *range* to a computation at the expense of *accuracy*. An IEEE single-precision floating-point number allocates 23 bits to the mantissa, giving an effective mantissa of only 24 bits when the implied 1 is considered. The advantage of floating point is that its exponent allows for the representation of numbers across a broad range (IEEE normalized single-precision values range from $\approx \pm 3 \times 10^{38}$ to $\approx \pm 1 \times 10^{-38}$). Conversely, while a 32-bit fixed-point representation (1.31 format) has a range of only −1 to ≈+1, it can represent some values within that range much more accurately than a floating-point format can—for example, numbers close to +1 such as 0.11111111111111111111111111111111. However, for numbers very close to +0, the fixed-point representation would have many leading zeroes, and thus would have *less* precision than the competing floating-point representation.

An important characteristic of floating point is its auto-scaling behavior. After every floating-point operation, the result is normalized and the exponent adjusted accordingly. No work on the part of the designer is required in this respect (although significant hardware resources are used). Thus, it is useful in cases where the range of intermediate values cannot be bounded by the designer and therefore where fixed point is unsuitable.

The use of floating point in FPGA-based design has been the topic of much research over the past decade. Early papers, such as Ligon and colleagues [15] and Shirazi et al. [24], focused on the cost of floating point and demonstrated that small floating-point formats as well as single-precision formats could be eventually implemented using FPGA technology. Later work, such as that by Bellows and Hutchings [1] and Roesler and Nelson [22], demonstrated novel ways of leveraging FPGA-specific features to more efficiently implement floating-point modules. Finally, Underwood [27] argued that the capabilities of FPGA-based platforms for performing floating point would eventually surpass those of standard computing systems.

All of the research just mentioned contains size and performance estimates for floating-point modules on FPGAs at the time they were published. Clever design techniques and growing FPGA densities and clock rates continually combine to produce smaller, faster floating-point circuits on FPGAs. At the time of this writing, floating-point module libraries are available from a number of sources, both commercial and academic.

21.4.3 Block Floating Point

Block floating point (BFP) is an alternative to fixed-point and floating-point arithmetic that allows entire blocks of data to share a single exponent. Fixed-point arithmetic is then performed on a block of data with periodic rescaling of its data values. A typical use of block floating point is as follows:

1. The largest value in a block of data is located, a corresponding exponent is chosen, and that value's fractional part is normalized to that exponent.
2. The mantissas of all other values in the block are adjusted to use the same exponent as that largest value.
3. The exponent is dropped and fixed-point arithmetic proceeds on the resulting values in the data block.
4. As the computation proceeds, renormalization of the entire block of data occurs—after every individual computation, only when a value overflows, or after a succession of computations.

The key is that BFP allows for growth in the range of values in the data block while retaining the low cost of fixed-point computations. Block floating point has found extensive use in fast Fourier transform (FFT) computations where an input block (such as from an A/D converter) may have a limited range of values, the data is processed in stages, and stage boundaries provide natural renormalization locations.

21.4.4 Constant Folding and Data-oriented Specialization

As mentioned Section 21.3.2, when the data for a computation changes, an FPGA can be readily reconfigured to take advantage of that change. As a simple example of data folding, consider the operation: $a =?b$, where a and b are 4-bit

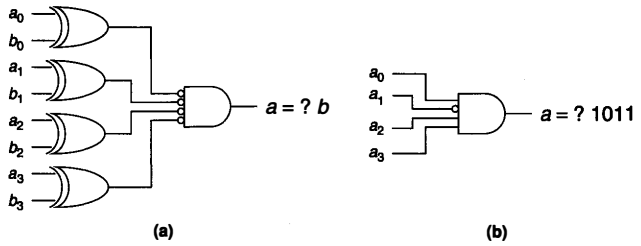


FIGURE 21.1 ■ Two comparator implementations: (a) with and (b) without constant folding.

numbers. Figure 21.1 shows two implementations of a comparator. On the left (a) is a conventional comparator; on the right (b) is a comparator that may be used when b is known ($b = 1011$). Implementation (a) requires three 4-LUTs to implement while implementation (b) requires just one. Such logic-level constant folding is usually performed by synthesis tools.

A more complex example is given by Wirthlin [30], who proposed a method for creating constant coefficient multipliers. When one constant to a multiplier was known, a custom multiplier consuming far fewer resources than a general multiplier could usually be created. Wirthlin's manipulations [30], going far beyond what logic optimization performed, created a custom structure for a given multiplier instance based on specific characteristics of the constant.

Hemmert et al. [10] offer an even more complex example in which a pipeline of image morphology processing stages was created, each of which could perform one image morphology step (e.g., one iteration in an erosion operation). The LUT contents in each pipeline stage controlled the stage's operation; thus, reconfiguring a stage required modifying only LUT programming. A compiler was then created to convert programs, written in a special image morphology language, into the data required to customize each pipeline stage's operation.

When a new image morphology program was compiled, a new bitstream for the FPGA could be created in a second or two (by directly modifying the original bitstream) and reconfigured onto the platform. This provided a way to create a custom computing solution on a per-program basis with turnarounds on the order of a few seconds. In each case, the original morphology program that was compiled provided the constant data that was folded into the design.

Additional examples in the literature show the power of constant folding. However, its use typically requires specialized CAD support. Slade and Nelson [25] argue that a fundamentally different approach to CAD for FPGAs is the solution to providing generalized support for such data-specific specialization. They advocate the use of JHDL [1, 12] to provide deployment time support for data-specific modifications to an operating FPGA-based system.

In summary, FPGAs provide architectural features that can accelerate simple arithmetic operations such as fixed-point addition and multiplication.

Floating-point operations can be accelerated using block floating point or by reducing the number of bits to represent floating-point values. Finally, constants can be propagated into arithmetic circuits to reduce circuit area and accelerate arithmetic performance.

21.5 SUMMARY

FPGAs provide a flexible, high-performance, and reprogrammable means for implementing a variety of electronic applications. Because of their reprogrammability, they are well suited to applications that require some form of direct reprogrammability, and to situations where reprogrammability can be used indirectly to increase reuse and thereby reduce device cost or count. FPGAs achieve the highest performance when the application can be implemented as many parallel hardware units operating in parallel, and where the aggregate I/O requirements for these parallel units can be reasonably met by the overall system. Most FPGA applications are described using HDLs because HDL tools and synthesis software are mature and well developed, and because, for now, they provide the best means for describing applications in a highly parallel manner.

Once FPGAs are determined to be a suitable choice, there are several ways to tailor the system design to exploit their reprogrammability by reconfiguring them at runtime or by compiling specific, temporary application-specific data into the FPGA circuitry. Performance can be further enhanced by crafting arithmetic circuitry to work around FPGA limitations and to exploit the FPGA's special arithmetic features. Finally, FPGAs provide additional debug and verification methods that are not available in ASICs and that enable debug and verification to occur in a system and at speed.

In summary, FPGAs combine the advantages and disadvantages of microprocessors and ASICs. On the positive side, they can provide high performance that is achievable only with custom hardware, they are reprogrammable, and they can be purchased in volume as a fully tested, standard product. On the negative side, they remain largely inaccessible to the software community; moreover, high-performance application development requires hardware design and the use of standard synthesis tools and Verilog or VHDL.

References

[1] P. Bellows, B. L. Hutchings. JHDL—An HDL for reconfigurable systems. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1998.

[2] B. Catanzaro, B. Nelson. Higher radix floating-point representations for FPGA-based arithmetic. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 2005.

[3] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, G. Snider. Exploring architectures for volume visualization on the Teramac custom computer. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[4] A. Dandalis, V. K. Prasanna. Fast parallel implementation of DFT using configurable devices. Field-programmable logic: Smart applications, new paradigms, and compilers. *Proceedings 6th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, 1997.

[5] C. H. Dick, F. Harris. FIR filtering with FPGAs using quadrature sigma-delta modulation encoding. Field-programmable logic: Smart applications, new paradigms, and compilers. *Proceedings 6th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag 1996.

[6] C. Dick. Computing the discrete Fourier transform on FPGA-based systolic arrays. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1996.

[7] J. G. Eldredge, B. L. Hutchings. Density enhancement of a neural network using FPGAs and runtime reconfiguration. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[8] P. Graham, B. Nelson. FPGA-based sonar processing. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1998.

[9] J. D. Hadley, B. L. Hutchings. Design methodologies for partially reconfigured systems. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[10] S. Hemmert, B. Hutchings, A. Malvi. An application-specific compiler for high-speed binary image morphology. *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[11] R. Hudson, D. Lehn, P. Athanas. A runtime reconfigurable engine for image interpolation. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, April 1998.

[12] B. L. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting. A CAD suite for high-performance FPGA design. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1999.

[13] B. L. Hutchings, R. Franklin, D. Carver. Assisting network intrusion detection with reconfigurable hardware. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, April 2002.

[14] B. L. Hutchings, M. J. Wirthlin. Implementation approaches for reconfigurable logic applications. *Field-Programmable Logic and Applications*, August 1995.

[15] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, K. D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[16] W. E. King, T. H. Drayer, R. W. Conners, P. Araman. Using MORPH in an industrial machine vision system. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[17] H. T. Kung. Why Systolic Architectures? *IEEE Computer* 15(1), 1982.

[18] S. Y. Kung. *VLSI Array Processors*, Prentice-Hall, 1988.

[19] T. Moeller, D. R. Martinez. Field-programmable gate array based radar front-end digital signal processing. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1999.

[20] G. Panneerselvam, P. J. W. Graumann, L. E. Turner. Implementation of fast Fourier transforms and discrete cosine transforms in FPGAs. *Fifth International Workshop on Field-Programmable Logic and Applications*, September 1995.

[21] R. J. Petersen. *An Assessment of the Suitability of Reconfigurable Systems for Digital Signal Processing*, Master's thesis, Brigham Young University, 1995.

[22] E. Roesler, B. Nelson. Novel optimizations for hardware floating-point units in a modern FPGA architecture. *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications*, August 2002.
[23] N. Shirazi, P. M. Athanas, A. L. Abbott. Implementation of a 2D fast Fourier transform on an FPGA-based custom computing machine. *Fifth International Workshop on Field-Programmable Logic and Applications*, September 1995.
[24] N. Shirazi, A. Walters, P. Athanas. Quantitative analysis of floating point arithmetic on FPGA-based custom computing machines. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.
[25] A. Slade, B. Nelson. Reconfigurable computing application frameworks. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.
[26] L. E. Turner, P. J. W. Graumann, S. G. Gibb. Bit-serial FIR filters with CSD coefficients for FPGAs. *Fifth International Workshop on Field-Programmable Logic and Applications*, September 1995.
[27] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. *Proceedings of the ACM/SIGDA 12th International Symposium on Field-Programmable Gate Arrays*, 2004.
[28] J. E. Vuillemin. On computing power. Programming languages and system architectures. *Lecture Notes in Computer Science*, vol. 781, Springer-Verlag, 1994.
[29] M. J. Wirthlin, B. L. Hutchings (eds). A dynamic instruction set computer. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.
[30] M. J. Wirthlin. Constant coefficient multiplication using look-up tables. *Journal of VLSI Signal Processing* 36, 2004.

CHAPTER 22

INSTANCE-SPECIFIC DESIGN

Oliver Pell, Wayne Luk
Department of Computing
Imperial College, London

This chapter covers instance-specific design, an optimization technique involving effective exploitation of information specific to an instance of a generic design description. Here we introduce different types of instance-specific designs with examples. We then describe partial evaluation, a systematic method for producing instance-specific designs that can be automated. Our treatment covers the application of partial evaluation to hardware design in general, and to field-programmable gate arrays (FPGAs) in particular.

22.1 INSTANCE-SPECIFIC DESIGN

FPGAs are an effective way to implement designs in computationally intensive datapath-orientated applications such as cryptography, digital signal processing, and network processing. The main alternative implementation technologies in these application areas are general-purpose processors, digital signal processors, and application-specific integrated circuits (ASICs).

ASICs are integrated circuits designed to implement a single application directly in fixed hardware. Because they are specialized to a single application, they can be very efficient, with reduced resource usage and power consumption over processor-based software implementations. Reconfigurable logic offers similar advantages over general-purpose processors. However, the overhead of providing general-purpose logic and routing resources means that FPGA-based systems typically provide lower density and performance than ASICs. Still, reconfigurable logic can provide a level of specialization beyond what is possible for an ASIC: optimizing circuits not just for a particular problem but for a particular *instance* of it. For example, an encryption application can create custom FPGA mappings every time a new password is given, allowing any password to be supported yet providing very highly optimized circuitry.

The basic concept of instance-specific design is to optimize a circuit for a particular computation. This can allow a reduction in area and/or an increase in processing speed by sacrificing the flexibility of the circuit. It is important to distinguish between the FPGA itself, which is inherently flexible and can be reconfigured to suit any application by loading a new bitstream, and the current configuration of the chip, which may have a certain level of flexibility in processing its inputs.

One common way of achieving instance-specific designs automatically is constant folding (Section 22.2.3), which involves propagating static input values through a circuit to eliminate unnecessary logic. Thus, in our encryption example, an exclusive-or (XOR) gate with one input driven by a password bit can be replaced with a wire or an inverter because the value of that bit is known for each specific password.

To produce an instance-specific design, one first needs a means of providing a particular instance for a given design. In the previous encryption example, if all the passwords are known at design time, an instance-specific design specialized for each password can be produced, say by constant propagation followed by the usual tools such as placement (Chapter 14), routing (Chapter 17), and bitstream generation (Chapter 19).

At runtime, a processor is often used to control the configuration of the FPGA by the appropriate bitstream at the right moment to support a particular password. However, if the passwords are known only at runtime, then the designer has to decide whether the benefits of having instance-specific designs outweigh the time to produce them, since, for instance, current place and route tools often take a long time to complete and their use is usually not recommended at runtime. Fortunately for some applications, differences between instances are so small that they can be generated realistically using runtime partial evaluation (Section 22.2).

The ability to implement specialized designs, while at the same time providing flexibility by allowing different specialized designs to be loaded onto a device, can make reconfigurable logic more effective at implementing some applications than what is possible with ASICs. For other applications, performance improvements from optimizing designs to a particular problem instance can help shift the price/performance ratio away from ASICs and toward FPGAs. Specializing a Data Encryption Standard (DES) crypto-processor, for example, can save 60 percent in area, while replacing general multipliers with constant coefficient versions can save area and lead to speedups of two to four times. Instance-specific designs can also consume lower power. Bit-width optimization of digital filters, for example, has been shown to reduce power consumption by up to 98 percent [2].

Changing an instance-specific design at runtime is generally much slower than changing the inputs of a general circuit, because a new (or partial) configuration must be loaded. Because this may take many tens or hundreds of milliseconds, it is important to carefully choose how a design is specialized.

22.1.1 Taxonomy

Types of instance-specific optimizations

We can divide the different approaches to optimizing a design for a particular problem instance into three main categories. Table 22.1 lists some examples of the different categories used.

Constant folding Constant folding is the process of eliminating unnecessary logic that computes functions with some inputs that never change or that

TABLE 22.1 ■ Examples of the uses of instance-specific designs

	Purpose	Example use	Impact
Constant folding	Optimize logic for static inputs	Key-specific DES	60% area reduction
Function adaptation	Optimize for desired quality of result	Accuracy-guaranteed bit-width optimization [4]	26% area reduction, 12% latency reduction
Architecture adaptation	Achieve a specified performance, area, or power target	Custom instruction processors [3]	72% decrease in runtime for 3% more area

change only rarely. This logic can be specialized to increase performance and reduce area. Examples of circuits that can benefit from constant folding will be seen later, and a more detailed description of the technique can be found in Section 22.2.3.

Function adaptation Function adaptation is the process of altering a circuit's function to achieve a specific quality of result. Typically this involves varying the number of bits used to represent data values or switching between floating-point and fixed-point arithmetic functions. It can also involve adding or removing parts of processing units that affect accuracy—for example, adding or removing stages from a CORDIC circuit. Word-length optimization can be treated automatically (Chapter 23), modifying a circuit's area to meet particular accuracy constraints.

Architecture adaptation Architecture adaptation alters the way in which a circuit computes a result while keeping the overall function the same. This can entail introducing additional parallelism to increase speed, serializing existing parallel processing units to save area, or refining processing capabilities to exploit some expected characteristics of the input data. Custom instruction processors (see Figure 22.4 later) are one example of the latter type of architecture adaptation.

22.1.2 Approaches

Instance-specific circuits can be produced either by specializing a general-purpose circuit or by starting directly from a "template" that must be instantiated for a particular problem instance before use, as shown in Figure 22.1. Specialization has the advantage that it can often be performed automatically, using techniques such as partial evaluation (Section 22.2). The template approach probably requires the manual design of a template circuit substantially different from the general-purpose architecture, but it can possibly provide a greater level of optimization than what is possible through specializing a general-purpose circuit. It can also offer the advantage that the hardware compilation process may need to be

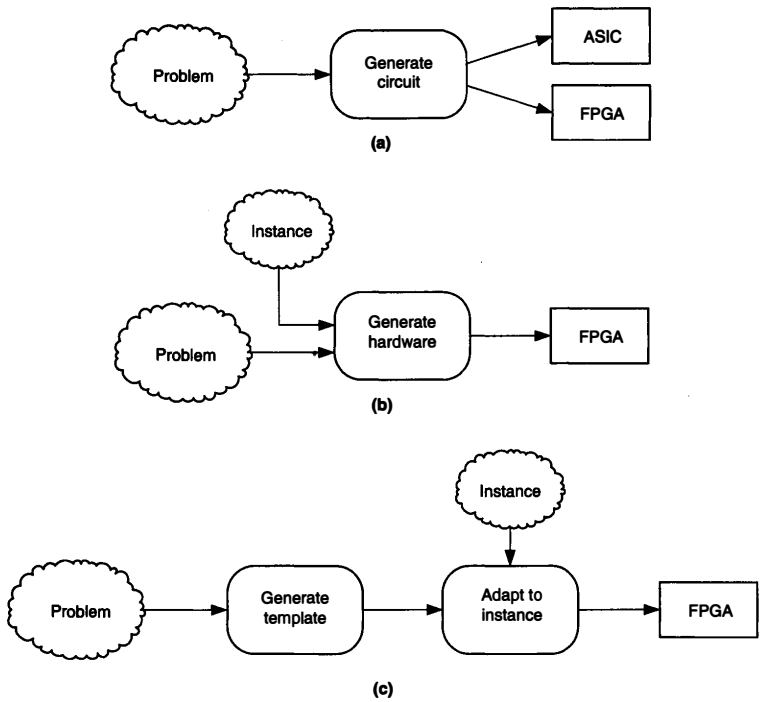


FIGURE 22.1 ■ General-purpose hardware (a) can be implemented using FPGAs or ASICs. Instance information (b) can be incorporated at hardware generation to produce a specialized circuit. "Template" hardware (c) can be generated and then instantiated for particular problem instances. The reason for the differences between (b) and (c) are that, in (b) the time-consuming process of hardware compilation must be executed for each instance while in (c) hardware compilation may only need to be run once, after which the final circuit bitstream can be amended.

executed only once, with instance-specific information being annotated directly into the bitstream.

In both cases, one or more instance-specific designs will be produced that can be converted into bitstreams through the FPGA design flow (see chapters in Part III). The appropriate bitstream can then be used to configure an FPGA, usually under the control of a general-purpose processor; during the reconfiguration process the FPGA will usually not be able to process data, although some partially reconfigurable devices can support the reconfiguration of some of its resources, while some of its other resources stay operational.

22.1.3 Examples of Instance-specific Designs

The benefits of instance-specific design can be illustrated by considering a few examples of its use. In this section we present three examples of specialization by constant folding into an existing design, and two examples of architecture adaptation.

Constant coefficient multipliers

If using standard logic cells, multipliers are relatively expensive to implement on FPGAs. A standard combinational multiplier ANDs each bit of input B with all bits of input A (to perform the multiply by 0/1); an adder is then used to sum together the partial products. When one coefficient of the multiplication is constant, however, the required area can be reduced dramatically. The AND functions are unnecessary because multiplying by a fixed 0 or 1 is trivial, and the adders can be eliminated for bits of B that are 0 (and thus have a partial product of 0). Constant coefficient multiplication is a useful operation in many signal-processing applications.

Finite impulse response (FIR) filters contain a set of multiply–add cells that multiply the value of the input signal across a number of cycles with filter coefficients and then sum these values. The multiplier coefficients are properties of the filter and do not change with the input data, but only need adjusting when different filter properties are required. Thus, the generic multipliers in a FIR filter circuit can often be replaced by smaller constant coefficient multipliers. (see Figure 22.2).

Another application that requires multipliers with constant coefficients is conversion from RGB to YUV video signals. This is a matrix multiplication operation where one matrix is constant, allowing specialized multipliers to be used.

Key-specific crypto-processors

Cryptographic algorithms are often designed for efficient implementation in both hardware and software. Block ciphers, such as DES and its successor Advanced Encryption Standard (AES), have regular algorithmic structures consisting of simple operations, such as XOR and bit permutation, that are efficiently implemented in hardware.

The DES algorithm consists of 16 "rounds," or processing stages, that can be pipelined for parallel operation. Blocks of 64-bit data are input to the array along with a 56-bit key and processed through each round, with the same key required to decrypt the data at the other end of the communication channel. A single DES round is illustrated in Figure 22.3.

In typical operation it is likely that a crypto-processor is used to process large blocks of data with the same key—for example, when transferring data between a single sender and receiver in a network or encrypting a large file to be saved to disk. It is therefore expected that, in contrast to the data input, the key value will change very slowly.

The shaded area of Figure 22.3 is key generator circuitry that generates the round key from the master key and then uses it as an input to a set of 2-input XOR functions across the data bits.

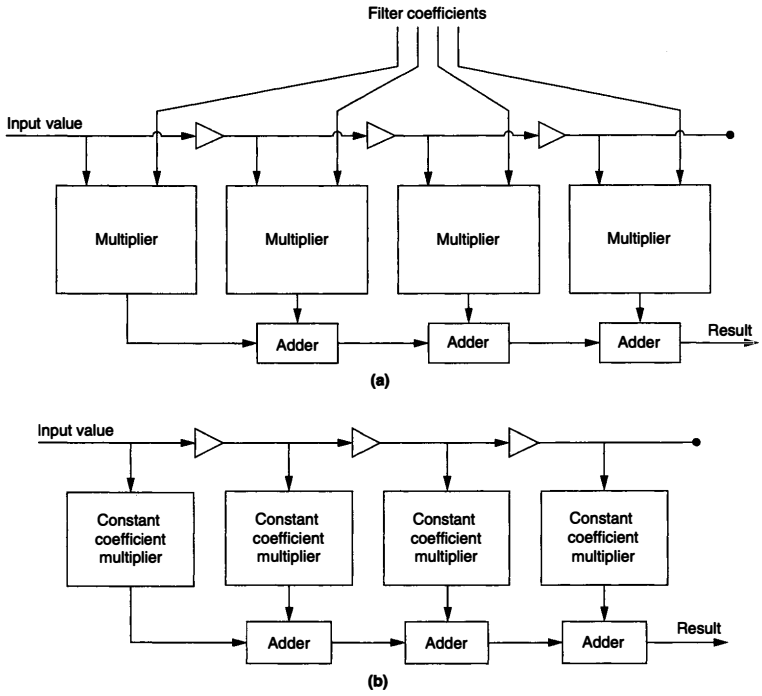


FIGURE 22.2 ■ FIR filters utilizing (a) general multipliers with variable filter coefficients and (b) instance-specific multipliers specialized to filter coefficients.

When the key value is known, the key generation circuitry can be eliminated and the XOR functions replaced with either wires or inverters [5]. In fact, these inverters can be merged into the substitution stage, eliminating the inverter logic as well [11]. Key-specific crypto-processors can exhibit much higher throughput than general versions, even outperforming ASIC implementations. Area savings are also significant—a relatively simple specialization of a placed DES description can yield area savings of 60 percent when implemented on a Xilinx Virtex FPGA [9].

Network intrusion detection

Network Intrusion Detection Systems (NIDS) perform deep packet inspection on network packets to identify malicious attacks. Normally, these systems are implemented in software, but on high-speed networks software alone is often unable to process all traffic at the full data rate.

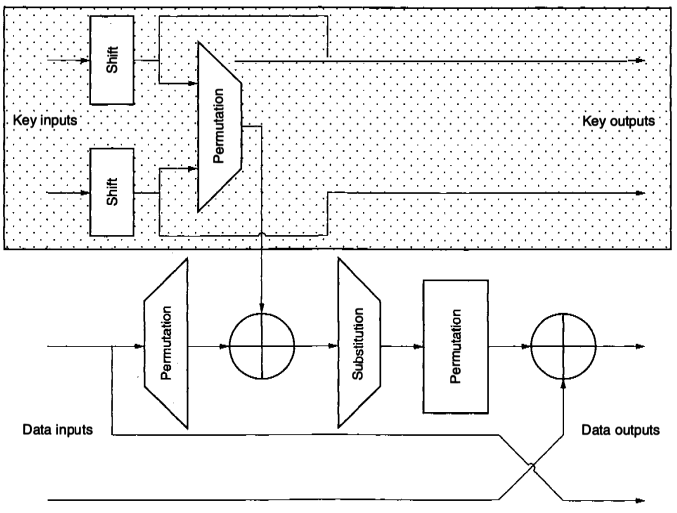


FIGURE 22.3 ■ A single round of a DES circuit. The shaded area contains key expansion circuitry that can be eliminated in a key-specific DES circuit, allowing the XOR function to be optimized.

The SNORT open source NDIS (see *http://www.snort.org*) uses a rule-based language to detect abnormal network activities. It contains thousands of rules, more than 80 percent of which contain signatures that must be matched against packet contents. Eighty percent of the CPU time for SNORT is consumed by this string-matching task [6]. String matching can be done efficiently in hardware and in particular can be easily optimized for particular search strings. While network data might be expected to arrive at high speed, the rule set changes much more slowly, so string-matching circuitry on FPGAs can be customized to match particular signatures. Section 22.2.5 illustrates in more detail how an instance-specific pattern matcher can be constructed. Further information about instance-specific designs for SAT solving applications can be found in Chapter 29.

Customizable instruction processors

General-purpose instruction processors are very flexible computational devices. Application-specific instruction processors, in contrast, have been customized to perform particularly well in a particular application area. This is a form of architecture adaptation that can improve performance for particular problem instances while maintaining the flexibility of the overall system.

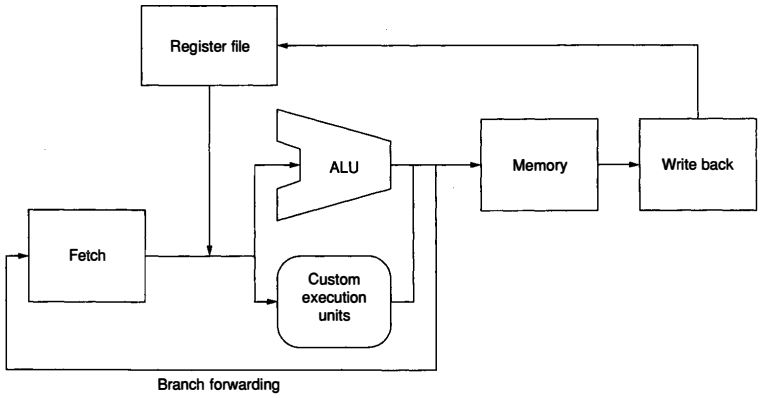


FIGURE 22.4 ■ A simplified architecture of a custom instruction processor. The standard arithmetic and logic operations are augmented by custom execution units that can accelerate particular applications.

Figure 22.4 illustrates the architecture of a simple custom instruction processor that has standard arithmetic and logic functions implemented by a standard ALU. These functions can be supported by additional custom execution units to accelerate particular applications. The automatic identification of instructions that can benefit from the custom execution units is a topic of active research [1]. Further information about partitioning sequential and parallel programs for software and hardware execution can be found in Chapter 26.

22.2 PARTIAL EVALUATION

Partial evaluation is a process that automates specialization in software or hardware. In both cases the motivation is the same: to produce a design that runs faster than the original. In software, partial evaluation can be thought of as a combination of constant folding, loop unrolling, function inlining, and interprocedural analyses; in hardware, constant folding is mainly used as an optimization method.

Partial evaluation is accomplished by detecting fragments of hardware that depend exclusively on variables with fixed values and then optimizing the hardware logic to reduce its area or even eliminate it totally from the design by precomputing the result.

22.2.1 Motivation

Partial evaluation can simplify logic, and thus reduce area and increase performance. Figure 22.5 illustrates its impact on a 2-input XOR function. When both inputs are dynamic, the logical function must be implemented; however, when one input is known, a partial evaluator can simplify the circuit. If one input is fixed high, the XOR functions as an inverter and so can be replaced by a 1-input NOT gate; if the input is fixed low, the XOR serves as a wire and the logic can be completely eliminated.

Constant folding propagates constants through a circuit and can substantially simplify logic functions. This can both reduce area (by allowing functions to be implemented using fewer LUTs) and increase performance (by reducing the number of logic levels between registers).

In this chapter we highlight two related uses of partial evaluation for circuits. The first, at the beginning of Section 22.2.4, optimizes generic circuit descriptions for improved performance. That is, circuits are described using clear and easily maintainable but nonoptimal design patterns, which are then automatically optimized during synthesis. The second, in the middle of Section 22.2.4, specializes general circuits when some inputs are static, such as constant coefficient arithmetic.

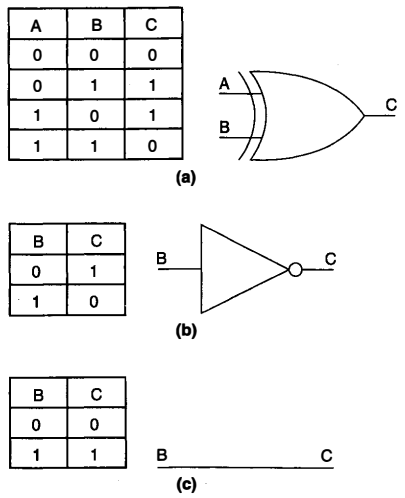


FIGURE 22.5 ■ Partial evaluation of an XOR gate. (a) A 2-input XOR function can be specialized, when input A is to become static: (b) an inverter when A is true or (c) a wire when A is false.

22.2.2 Process of Specialization

Consider a general circuit C producing output R, whose inputs are partitioned into two sets S and D.

$$R = C(S,D)$$

This circuit can be specialized for a particular set of S inputs such that it computes the same result for all possible inputs D:

$$R = C_{S=X}(D)$$

A partial evaluator is an algorithm that, when supplied with values for the set of inputs S and the circuit C, produces a specialized circuit $C_{S=X}$.

$$C_{S=X} = \mathbf{P}(C,S,X)$$

where S is the set of static inputs that are known at compile time, and D is the set of dynamic inputs. The importance of partial evaluation is that the specialized circuit computes precisely the same result as the original circuit, though it may require less hardware to do so.

Relating this framework to the XOR gate example, $R = \text{XOR}(A,B)$, with $S = \{A\}$ and $D = \{B\}$, the two possible simplified functions can be described as

$$R = \text{XOR}_{A=X}(B)$$

for the two possible values of A.

$$\text{XOR}_{A=0} = \text{P}(\text{XOR},A,0) = \text{NOT}(B)$$
$$\text{XOR}_{A=1} = \text{P}(\text{XOR},A,1) = B$$

22.2.3 Partial Evaluation in Practice

Constant folding in logical expressions

Partial evaluation of logic is well understood and has been used to simplify circuit logic for many years. Figure 22.6 gives a simple partial evaluation function, P(S)[[X]], for optimizing Boolean logic expressions expressed using *not*, *and*, and *or* connectives. The function is parameterized by a set S of pairs mapping static variables to their values and a Boolean expression X represented as a tree.

The function is defined recursively on the structure of Boolean expressions. Cases (1), (2), and (3) are base conditions, indicating that partial evaluation of the Boolean constants True and False always has no effect, and partial evaluation of a variable a returns either the constant value of that variable (if it is contained within the static inputs) or the variable name if it is not static (i.e., remains dynamic).

Case (4) defines partial evaluation of a single-input *not* function. If the subexpression evaluates to logical truth or falsity, this is inverted by the conditional

```
(1) P(S)[[True]]     = True

(2) P(S)[[ False ]]  = False

(3) P(S)[[ a ]]      = if a ∈ dom(S) then P(S)[[ S(a) ]] else a

(4) P(S)[[ ¬ x ]]    = Let y = P(S)[[ x ]]
                       If y == True then False
                       Else if y == False then True
                       Else ¬ y

(5) P(S)[[ x & y ]]  = Let x' = P(S)[[ x ]]
                       Let y' = P(S)[[ y ]]
                       if(x' == False || y' == False) then False
                       Else if x' == True then y'
                       Else if y' == True Then x'
                       Else x' & y'

(6) P(S)[[ x+y ]]    = Let x' = P(S)[[ x ]]
                       Let y' = P(S)[[ y ]]
                       If(x' == True || y' == True) then True
                       Else if x' == False then y'
                       Else if y' == False then x'
                       Else x+y
```

FIGURE 22.6 ■ A partial evaluation algorithm for simplifying Boolean logic expressions.

check. Otherwise, the partially evaluated subexpression is returned with the *not* operation.

Cases (5) and (6) define partial evaluation of 2-input *and* and *or* functions. The process is the same: Simplify the subexpressions, precompute the function result if possible, and, if not, return the function with simplified arguments.

As an example, consider the application of this algorithm to the simplification of the XOR function in Figure 22.5. XOR can be described in terms of basic Boolean operators as

$$a \text{ xor } b = (a \& \neg b) + (\neg a \ \& \ b)$$

Partially evaluating when a is asserted, the function is executed:

$$\text{(i)} \quad P(\{a \rightarrow \text{True}\})[[(a \ \& \ \neg b) + (\neg a \ \& \ b)]]$$

Case (6) for simplifying logical-or is used, and the two subexpressions are partially evaluated separately:

$$\text{(ii)} \quad P(\{a \rightarrow \text{True}\})[[a \& \neg b]]$$
$$\text{(iii)} \quad P(\{a \rightarrow \text{True}\})[[\neg a \& b]]$$

Both (ii) and (iii) are partially evaluated by the case for logical-and. For (ii) the two subexpressions are first evaluated as

$$\text{(iv)} \quad P(\{a \rightarrow \text{True}\})[[a]] = \text{True}$$
$$\text{(v)} \quad P(\{a \rightarrow \text{True}\})[[\neg b]] = \neg b$$

In (iv), the variable a is within the static inputs S and thus is simplified to True, while $\neg b$ is unchanged because it does not contain a. The results from partially evaluating (iii) are similar:

$$\text{(vi)} \quad P(\{a \rightarrow \text{True}\})[[\neg a]] = P(\{a \rightarrow \text{True}\})[[\neg \text{True}]] = \text{False}$$
$$\text{(vii)} \quad P(\{a \rightarrow \text{True}\})[[b]] = b$$

Equipped with the simplified subexpressions, the expression $a \,\&\, \neg b$ is simplified to $\neg b$ and the expression $\neg a \,\&\, b$ is simplified to False. At the top level this gives a logical-or: $\neg b + \text{False}$:

$$\text{(viii)} \quad P(\{a \rightarrow \text{True}\})[[\neg b + \text{False}]] = \neg b$$

The XOR function reduces to a single inverter; if supplied with $\{a \rightarrow \text{False}\}$ the partial evaluation function instead returns just b, indicating the simple wire. This is consistent with the truth tables in Figure 22.5.

The partial evaluation function just given is quite simple and does not capture all possible optimizations. For example, the logic function $a + \neg a$ always evaluates to True, regardless of the value of a; however, this expression will not be simplified by this function.

Unnecessary logic removal

Another optimization that can be carried out during partial evaluation is removal of dead logic in a design, which does not affect any output and thus is unnecessary. This is a very important optimization because it allows generic hardware blocks computing many functions to be used in designs, with unused functions pruned during synthesis.

As an algorithmic process, logic removal is quite simple and can be formulated in a number of different ways. One of the simplest is to identify each gate whose output is unconnected and eliminate it. By recursively applying this rule we can eliminate acyclic dead logic.

22.2.4 Partial Evaluation of a Multiplier

Optimizing a simple description

Figure 22.7 shows a shift–add circuit designed for a Xilinx architecture to compute the 3-bit multiplication of two 3-bit inputs. This circuit appears semiregular, with x and y inputs propagating horizontally and vertically through a triangular array of processing cells. Each processing cell has common features; however, it contains slightly different logic depending on its position in the array.

Creating and maintaining a circuit description that contains and correctly connects the different types of cell is quite complicated. A simpler approach is to exploit the regularity to describe the circuit as an array of a single type of cell that is then partially evaluated during synthesis to produce the circuit in Figure 22.7.

The general cell of the multiplier can be described as shown in Figure 22.8. This cell implements a multiplication operation for 1 bit of x and 1 bit of y,

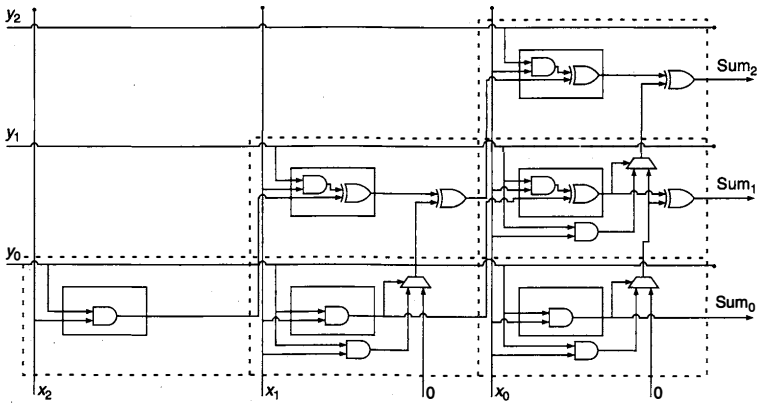


FIGURE 22.7 ■ A shift–add multiplier circuit that takes two 3-bit inputs and produces a 3-bit output.

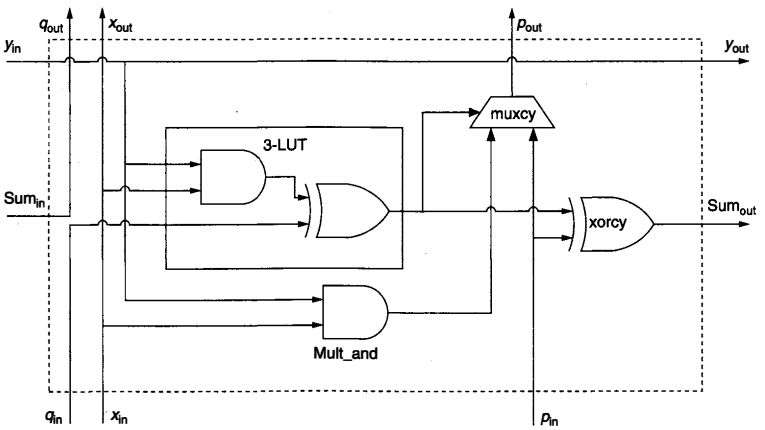


FIGURE 22.8 ■ This cell design can be replicated in a grid arrangement to create a multiplier.

producing sum and carry-out bits, and can be arranged in a grid to generate a multiplication circuit identical in function to that shown in Figure 22.8. These cells can be implemented densely on Xilinx architectures by using the specialized `mult_and`, `xorcy`, and `muxcy` components in each slice.

Partial evaluation can automatically produce the optimized multiplication circuitry from the initial regular description. The four components within each cell each have their own logical formula. In the case of `mult_and`, `xorcy`, and `muxcy`, no simplification is possible unless we can totally eliminate these functions, because these are fixed resources on the device, compared with the LUT, which can flexibly implement any 4-input function.

The logic of the standard cell can be represented as

$$\begin{aligned} LUT_{\text{out}} = (Y_{\text{in}} \ \& \ X_{\text{in}}) \text{xor } Q_{\text{in}} &= (\neg\,(Y_{\text{in}} \ \& \ X_{\text{in}}) \ \& \ Q_{\text{in}}) + ((Y_{\text{in}} \ \& \ X_{\text{in}}) \ \& \neg Q_{\text{in}}) \\ AND_{\text{out}} &= (Y_{\text{in}} \ \& \ X_{\text{in}}) \\ P_{\text{out}} &= (LUT_{\text{out}} \ \& \ P_{\text{in}}) + (\neg\, LUT_{\text{out}} \ \& \ AND_{\text{out}}) \\ SUM_{\text{out}} &= (\neg\, LUT_{\text{out}} \ \& \ P_{\text{in}}) + (LUT_{\text{out}} \ \& \ \neg\, P_{\text{in}}) \end{aligned}$$

This logic can be simplified by two operations: removing unconnected logic and constant folding to optimize the logic that remains. Removal of disconnected logic transforms the grid into the triangular array, while constant folding can be performed by the partial evaluation function introduced in Figure 22.6.

For example, for the cells along the bottom in Figure 22.8, inputs Q_{in} and P_{in} are all zero. This allows the LUT contents to be optimized by

$$\begin{aligned} LUT_{\text{out}}' = P(\{Q_{\text{in}} \rightarrow \text{False}, SUM_{\text{in}} \rightarrow \text{False}, P_{\text{in}} \rightarrow \text{False}\}) \\ [[(\neg\,(Y_{\text{in}} \ \& \ X_{\text{in}}) \ \& \ Q_{\text{in}}) + ((Y_{\text{in}} \ \& \ X_{\text{in}}) \ \& \neg\, Q_{\text{in}})]] = (Y_{\text{in}} \ \& \ X_{\text{in}}) \end{aligned}$$

The function attempts to partially evaluate both branches of the OR expression. On the left branch, $\neg(Y_{\text{in}} \ \& \ X_{\text{in}})$ cannot be further optimized and so is left intact; however, Q_{in} is known to be false, so the entire left branch must be false and thus is eliminated. On the right branch, $\neg Q_{\text{in}}$ is evaluated to true and eliminated from the expression, leaving $(Y_{\text{in}} \ \& \ X_{\text{in}})$ as the simplified function for the LUT contents.

AND_{out} cannot be simplified because both Y_{in} and X_{in} are unknown. Neither can P_{out} because, although it can be partially optimized (because P_{in} is false), it is a fixed component available on the FPGA that cannot be simplified. Partial evaluation of SUM_{out} does succeed in eliminating logic:

$$\begin{aligned} SUM_{\text{out}}' = P(\{Q_{\text{in}} \rightarrow \text{False}, SUM_{\text{in}} \rightarrow \text{False}, P_{\text{in}} \rightarrow \text{False}\}) \\ [[(\neg\, LUT_{\text{out}} \ \& \ P_{\text{in}}) + (LUT_{\text{out}} \ \& \ \neg P_{\text{in}})]] = LUT_{\text{out}} \end{aligned}$$

The result of this partial evaluation is that the bottom cells of the multiplier are optimized to remove the unnecessary `xorcy` component and to simplify the 3-input LUT function into a basic 2-input AND function.

Functional specialization for constant inputs

If some of the input values to the multiplication circuit are known statically, we can apply constant folding to eliminate further logic. For example, assume that x_1 is static and always zero. Partially evaluating the cell logic under the new assumption that $\{X_{\text{in}} \rightarrow \text{False}\}$ we find that the entire cell can be eliminated and replaced with pure routing. The simplified cell is shown in Figure 22.9.

Because a single bit of the x input is shared with an entire column of the multiplier, this specialized cell can be used for the full column, replacing all the logic with routing, as shown in Figure 22.10; this arrangement in turn allows optimizations to be applied to the second LUT in the final column to eliminate the XOR function (not shown in the figure so that the routing can be seen).

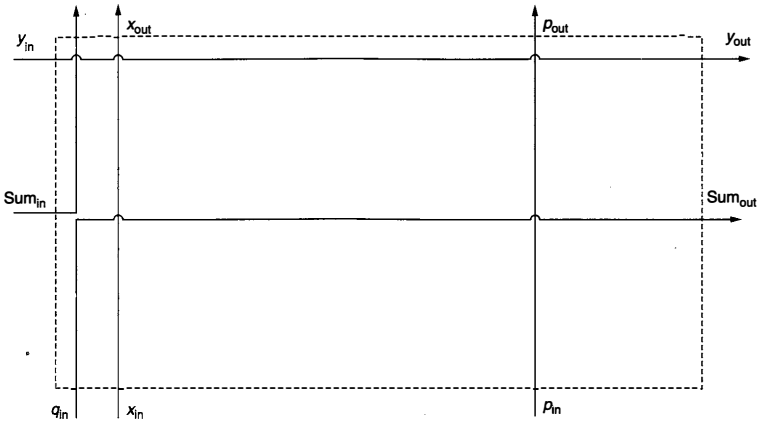


FIGURE 22.9 ■ The impact of partial evaluation on multiplier cell logic when X_{in} = False.

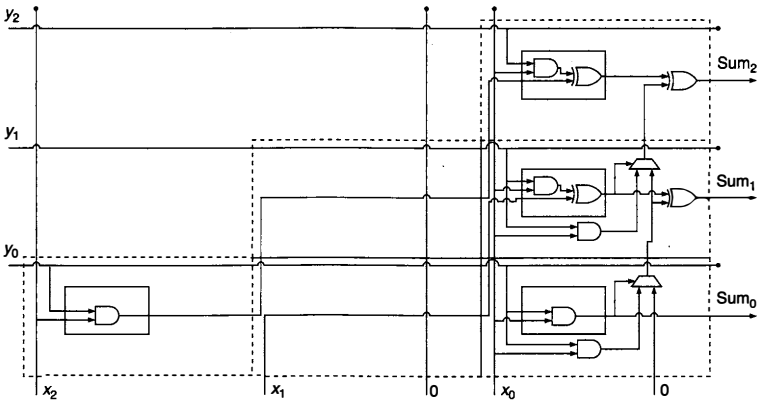


FIGURE 22.10 ■ Multiplier circuit specialized by eliminating the center column when x_i is always zero.

When an x value is known to be true, partial evaluation can still carry out some optimizations. However, it does not offer the significant advantages that result when x is false. The LUT can again be optimized to a 2-input function and the `mult_and` component can be eliminated. This is not very significant, however—the `mult_and` component is already present on the device, so no area is saved, and it is utilized in parallel with the (slower) LUT so there is also no performance gain.

Geometric specialization

High-performance FPGA designs often include layout information to produce good placements with low routing delays (see Chapter 17). Specialization of placed designs may lead to nonoptimal results if the placement is not updated to reflect eliminated logic. Automatic placement is not affected, since partial evaluation is usually carried out at the synthesis stage prior to placement and routing. However, when hand-placed designs are specialized, the effect can be to introduce unnecessary delays by failing to compact components. These gaps can also prevent effective use of freed logic because it is fragmented among other components. To ensure a good placement of specialized designs it is necessary to optimize placement information, compacting the circuit. This can be achieved in a framework that allows partial evaluation prior to placement position generation [8] or by describing circuit layouts in a way that adapts when the circuit is specialized [12].

22.2.5 Partial Evaluation at Runtime

Pattern matching is a relatively simple operation that can be performed efficiently in hardware. It is useful in a range of fields but is of particular interest in networking for inspecting the contents of data packets.

Figure 22.11 illustrates a simple general pattern matcher made up of a repeating bit-level matcher cell. Each cell contains a pattern and a mask value, which can be loaded separately from the data to be matched. Input data is streamed in 1 bit per cycle; if the mask value for a particular bit position is set, the cell for that position checks the current data value against the bit pattern.

The pattern matcher requires one LUT and three registers for each bit in the data pattern. However, it is likely that the pattern and mask values will change much more slowly than the data input, so it is reasonable to investigate the potential for partial evaluation to optimize this circuit for fixed patterns.

When the pattern and mask are fixed, the registers storing their values can be eliminated and the logic in the LUTs can be optimized. Figure 22.12 shows how the pattern matcher can be optimized for a pattern of "10X1" (the third pattern bit is a "don't care," as specified by the mask of "1101"). This circuit uses fewer registers and three LUTs rather than four. The significance of this particular way of optimizing is that the pattern matcher's structure has mostly been maintained and thus this specialization can be carried out at runtime.

Changes to the mask require routing changes—complex, though far from impossible at runtime; however, the pattern to be matched can be changed merely by updating the LUT contents.

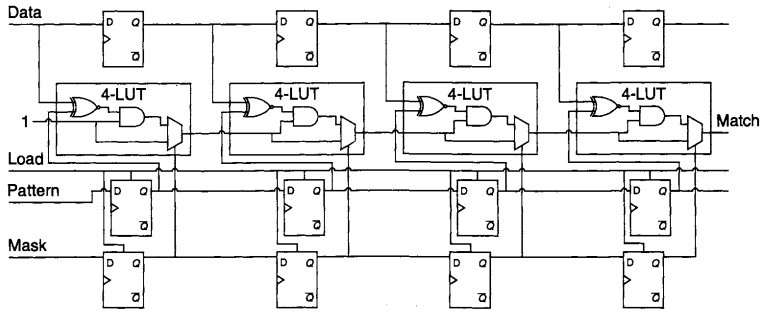


FIGURE 22.11 ■ A general bit-level pattern matcher, shown for 4-bit patterns. The pattern matcher circuit is controlled by a pattern and a mask, which can be loaded by asserting the load signal. If the mask bit is set for a particular position, the matcher will attempt to detect a match between the pattern bit and the data bit.

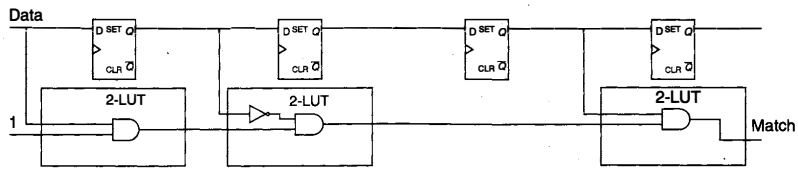


FIGURE 22.12 ■ An instance-specific pattern matcher optimized for a mask of `1101` and pattern of `10X1` requires only three LUTs and four registers.

22.2.6 FPGA-specific Concerns

LUT mapping

Recall the pattern matcher example from the previous section, where we showed one partial evaluation of the circuit for a particular pattern. In this case partial evaluation significantly simplified the contents of each LUT, from a 4-input function to a much simpler 2-input function.

It is important that, in contrast to ASICs, there is often no performance advantage to be gained by reducing the complexity of logic functions in an FPGA unless the number of LUTs required to implement those functions is reduced. The propagation delay of a LUT is independent of the function it implements; thus, there is no gain in reducing a 4-input function to a 2-input function within the same LUT (although it does allow routing resources to be freed for other uses).

For runtime specialization, it may be desirable to maintain much of the original circuit structure. However, when partial evaluation is carried out at compile time it should be performed before logic is mapped to LUTs, giving more scope for improvements in circuit area and performance. Figure 22.13 shows that the

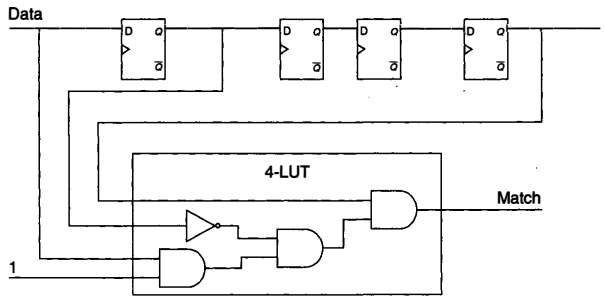


FIGURE 22.13 ■ The instance-specific pattern matcher from Figure 22.12 can be implemented using a single 4-LUT rather than three 2-LUTs.

specialized pattern matcher can indeed be implemented using one 4-LUT rather than three 2-LUTs, with higher performance and lower area requirements than the version partially evaluated at runtime.

In fact, the static 1-input can also be eliminated from this LUT; however, it has been left to indicate that this LUT structure can be used as part of a chain in a larger pattern matcher.

Static resources

As alluded to in the multiplier example, the existence of specific resources on an FPGA in addition to LUTs, such as carry chain logic, poses a problem for automatic partial evaluation algorithms. Not only can this logic not be simplified (for example, the `xorcy` gate cannot be replaced with an inverter), in some cases it cannot be eliminated at all because of routing constraints (carry signals must propagate through `muxcy` multiplexers, for example, regardless of necessity).

Furthermore, it is often important to maintain use of the dedicated carry chain, even though significantly simpler logic could perhaps be generated after partial evaluation, because the carry chain is designed to propagate carry signals very quickly—and much faster than the general routing fabric.

Verification of runtime specialization

Dynamic specialization at runtime poses additional verification problems over and above verification of an original design. While a circuit may have been verified through extensive simulation or formal methods prior to synthesis, when it is specialized at runtime it is possible for new errors to be introduced.

To avoid this it is necessary to ensure that the algorithms that apply partial evaluation at runtime have themselves been verified. Formal proof is an appropriate methodology for this problem, since it is necessary to check a generic property of the algorithm applied to all circuits rather than any particular specialization operation.

Although formal verification has been applied to partial evaluation algorithms for specialization of FPGA circuits [7, 14], it remains a relatively unexplored area.

22.3 SUMMARY

This chapter described instance-specific design, which offers the opportunity to exploit the reconfigurable nature of FPGAs to improve performance by tailoring circuits to particular problem instances. It can be broadly categorized into three techniques: constant folding, which can be applied when some inputs are static; function adaptation, which alters the function of circuitry to produce a certain quality of result; and architecture adaptation, in which the circuit architecture is adapted without affecting its functional behavior.

The level of automation that can be applied varies among these approaches. Constant folding can often be carried out automatically using partial evaluation techniques. Function adaptation can be performed by varying bit widths and arithmetic methods in parameterized IP cores. Tools, such as Quartz (for low-level design) [12] or ASC (for stream architectures) [10], can produce highly parameterized circuit cores where design parameters can be traded off against each other to achieve the desired requirements in area, speed, and power consumption. Architecture adaptation, such as adding additional processing units to instruction processors, is typically much less automated. The designer must create separate implementations of the different architectures, optimizing each of them somewhat independently.

References

[1] K. Atasu, R. Dimond, O. Mencer, W. Luk, C. Özturan, G. Dündar. Optimizing instruction-set extensible processors under data bandwidth constraints. *Proceedings of Design, Automation and Test in Europe Conference*, 2007.

[2] G. A. Constantinides. Perturbation analysis for word-length optimization. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[3] R. Dimond, O. Mencer, W. Luk. Application-specific customisation of multithreaded soft processors. *IEE Proceedings on Computers and Digital Techniques*, May 2006.

[4] D. Lee, A. Abdul Gaffar, R.C.C. Cheung, O. Mencer, W. Luk, G. A. Constantinides. Accuracy guaranteed bit-width optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October 2006.

[5] J. Leonard, W. Magione-Smith. A case study of partially evaluated hardware circuits: Key-specific DES. *Proceedings of the International Workshop on Field-Programmable Logic and Applications*, 1997.

[6] E. P. Markatos, S. Antonatos, M. Polychronakis, K. G. Anagnostakis. Exclusion-based signature matching for intrusion detection. *Proceedings of IASTED International Conference on Communication and Computer Networks*, 2002.

[7] S. McKeever, W. Luk. Provably-correct hardware compilation tools based on pass separation techniques. *Formal Aspects of Computing*, June 2006.
[8] S. McKeever, W. Luk, A. Derbyshire. Towards verifying parametrised hardware libraries with relative placement information. *Proceedings of the 36th IEEE Hawaii International Conference on System Sciences*, 2003.
[9] S. McKeever, W. Luk, A. Derbyshire. Compiling hardware descriptions with relative placement information for parameterised libraries. *Proceedings of International Conference on Formal Methods in Computer-Aided Design*, LNCS 2517, 2002.
[10] O. Mencer. ASC: A stream compiler for computing with FPGAs. *IEEE Transactions on Computer-Aided Design*, August 2006.
[11] C. Patterson. High performance DES encryption in Virtex FPGAs using JBits. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
[12] O. Pell, W. Luk. Compiling higher-order polymorphic hardware descriptions into parametrised VHDL libraries with flexible placement information. *Proceedings of the International Workshop on Field-Programmable Logic and Applications*, 2006.
[13] O. Pell, W. Luk. Quartz: A framework for correct and efficient reconfigurable design. *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, 2005.
[14] K. W. Susanto, T. Melham. Formally analyzed dynamic synthesis of hardware. *Journal of Supercomputing* 19(1), 2001.

CHAPTER 23

PRECISION ANALYSIS FOR FIXED-POINT COMPUTATION

George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College, London

Many values in a computation are naturally represented by integers, which have very efficient hardware implementations; basic operations are relatively cheap, and they map well to an FPGA's underlying hardware. However, some computations naturally result in fractional values, that is, numbers where part or all of the value are less than 1—for example, 0.25, 3.25, and π—or that are so large that representation as integers is too costly—for example, 10^{120}. Handling these values is a significant concern because the hardware necessary to compute on *scaled* values can be significant in speed, power consumption, and area.

In arithmetic for reconfigurable computing designs, it is common to employ fixed point instead of floating point to represent scaled values. This chapter explores the reason for this design decision and the associated analysis that must be performed in order to choose an appropriate fixed-point representation for a particular design. Since designs for reconfigurable logic can be customized for particular applications, it is appropriate to fit the number system to the underlying application properties.

23.1 FIXED-POINT NUMBER SYSTEM

In general-purpose computing, floating-point representations are most commonly used for the representation of numbers containing fractional components. The floating-point representations standardized by the IEEE [22] have several advantages, the foremost being portability across different computational platforms.

In general, we may consider a floating-point number $X[t]$ at time t as made up of two components: a signed mantissa $M[t]$ and a signed exponent $E[t]$ (see equation 23.1). Within this representation, the ratio of the largest positive value of X to the smallest positive value of X varies exponentially with the exponent $E[t]$ and hence doubly exponentially with the number of bits used to store the exponent. As a result, it is possible to store a wide dynamic range with only a few bits of exponent, while the mantissa maintains the precision of the

representation across that range by dividing the corresponding interval for each exponent into equally spaced representable values.

$$X[t] = M[t] \cdot 2^{E[t]} \tag{23.1}$$

However, the flexibility of the floating-point number system comes at a price. Addition or subtraction of two floating-point numbers requires the alignment of radix ("decimal") points, typically resulting in a large, slow, and power-hungry barrel shifter. In a general-purpose computer, this is a minor concern compared to the need to easily support a wide range of applications. This is why processors designed for general-purpose computing typically have a built-in floating-point unit.

In embedded applications, where power consumption and silicon area are of significant concern, the fixed-point alternative is more often used [24]. We can consider fixed point as a degenerate case of floating point, where the exponent is fixed and cannot vary with time (i.e., $E[t] = E$). The fixing of the exponent eliminates the need for a variable alignment and thus the need for a barrel shifter in addition and subtraction. In fact, basic mathematical operations on fixed-point values are essentially identical to those on integer values. However, compared to floating point, the dynamic range of the representation is reduced because the range of representable values varies only singly exponentially with the number of bits used to represent the mantissa.

When implementing arithmetic in reconfigurable logic, the fixed-point number system becomes even more attractive. If a low-area fixed-point implementation can be achieved, space on the device can be freed for other logic. Moreover, the absence of hardware support for barrel shifters in current-generation reconfigurable logic devices results in an even higher area and power overhead compared to that in fully custom or ASIC technologies.

23.1.1 Multiple-wordlength Paradigm

For simplicity we will restrict ourselves to 2's complement representations, although the techniques presented in this chapter apply similarly to most other common representations. Also, we will use dataflow graphs, also known as signal flow graphs in the digital signal processing (DSP) community, as a simple underlying model of computation [12]. In a dataflow graph, each atomic computation is represented by a vertex $v \in V$, and dataflow between these nodes is represented by a set of directed edges $S \subseteq V \times V$. To be consistent with the terminology used in the signal-processing community, we will refer to an element of S as a *signal*; the terms *signal* and *variable* are used interchangeably.

The multiple-wordlength paradigm is a design approach that tries to fit the precision of each part of a datapath to the precision requirements of the algorithm [8]. It can be best introduced by comparison to more traditional fixed-point and floating-point implementations. Each 2's complement signal $j \in S$ in a multiple-wordlength implementation of a dataflow graph (V, S) has two parameters n_j and p_j, as illustrated in Figure 23.1(a). The parameter n_j represents the

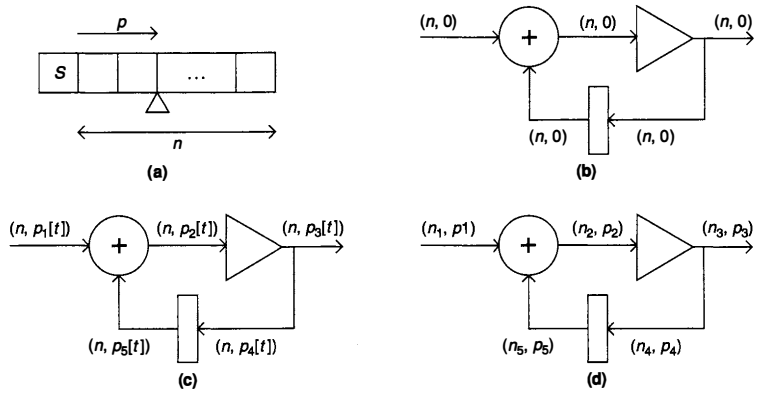


FIGURE 23.1 ■ The multiple-wordlength paradigm: (a) signal parameters ("s" indicates a sign bit); (b) fixed point; (c) floating point; (d) multiple wordlength. The triangle represents a constant coefficient multiplication, or "gain"; the rectangle represents a register, or unit sample delay.

number of bits in the representation of the signal (excluding the sign bit, by convention), and the parameter p_j represents the displacement of the binary point from the least significant bit (LSB) side of the sign bit toward the LSB. Note that there are no restrictions on p_j; the binary point could lie outside the number representation (i.e., $p_j < 0$ or $p_j > n_j$).

A simple fixed-point implementation is illustrated in Figure 23.1(b). Each signal j in this dataflow graph representing a recursive DSP algorithm is annotated with a tuple (n_j, p_j) representing the wordlength scaling of the signal. In this implementation, all signals have the same wordlength and scaling, although shift operations are often incorporated in fixed-point designs in order to provide an element of scaling control [25]. Figure 23.1(c) shows a standard floating-point implementation, where the scaling of each signal is a function of time.

A single systemwide wordlength is common to both fixed and floating point. This is a result of historical implementation on single, or multiple, predesigned arithmetic units. In FPGAs the situation is quite different. Different operations are generally computed in different hardware resources, and each of these computations can be built to any size desired. Such freedom points to an alternative implementation style, shown in Figure 23.1(d). This multiple-wordlength implementation style inherits the speed, area, and power advantages of traditional fixed-point implementations, since the computation is fixed point with respect to each individual computational unit. However, by potentially allowing each signal in the original specification to be encoded by binary words with different scaling and wordlength, the degrees of freedom in design are significantly increased.

23.1.2 Optimization for Multiple Wordlength

Now that we have established the possibility of using multiple scalings and wordlengths for different variables, two questions arise: How can we optimize the scalings and wordlengths in a design to match the computation being performed, and what are the potential benefits from doing so? For FPGA-based implementation, the benefits have been shown to be significant: Area savings of up to 45 percent [8] and 80 percent [15] have been reported compared to the use of a single wordlength across the entire circuit. The main substance of this chapter is to describe suitable scaling and wordlength optimization procedures to achieve such savings.

Section 23.2 shows that we can determine the appropriate scaling for a signal from an estimation of its peak value over time. One of two main techniques—simulation based and analytical—is then introduced to perform this peak estimation. While an analytical approach provides a tight bound on the peak signal value, it is limited to computations exhibiting certain mathematical properties. For computations outside this class, an analytical technique tends to be pessimistic, and so simulation-based methods are commonly used.

Section 23.3 focuses on determining the wordlength for each signal in the computation. The fundamental issue is that, because of roundoff or truncation, the wordlength of different signals in the system can have different impacts on both the implementation area and the error observed at the computation output. Thus, any wordlength optimization system needs to perform a balancing act between these two factors when allocating wordlength to signals. The goal of the work presented in this section is to allocate wordlength so as to minimize the area of the resulting circuit while maintaining an acceptable computational accuracy at the output of the circuit.

23.2 PEAK VALUE ESTIMATION

The physical representation of an intermediate result in a bit-parallel implementation of an algorithm consists of a finite set of bits, usually encoded using 2's complement representation. To make efficient use of the resources, it is essential to select an appropriate *scaling* for each signal. Such a scaling should ensure that the representation is not overly wasteful in catering to rare or impossibly large values and that overflow errors, which lead to low arithmetic quality, do not occur often.

To determine an appropriate scaling, it is necessary to determine the peak value that each signal can reach. Given a peak value P, a power-of-two scaling p is selected with $p = \lfloor \log_2 P \rfloor + 1$, since power-of-two multiplication is free in a hardware implementation.

For some algorithms, it is possible to estimate the peak value that each signal could reach using analytic means. In the next section, such techniques for two different classes of system are discussed. The alternative, to use simulation to determine the peak signal value, is described in the following section.

Also discussed are some hybrid techniques that aim to combine the advantages of both approaches.

23.2.1 Analytic Peak Estimation

If the DSP algorithm under consideration is a linear time-invariant system, it is possible to find a tight analytic bound on the peak value reachable by every signal in it. This is the problem addressed in the section immediately following. If, on the other hand, the system is nonlinear or time varying, such an approach cannot be used. If the algorithm is nonrecursive—that is, the dataflow graph does not contain any feedback loops—data range propagation may be used to determine an analytic bound on the peak value of each signal. However, this approach, described in the next section, cannot be guaranteed to produce a tight bound.

Linear time-invariant systems

A linear time-invariant (LTI) system is one that obeys the distinct properties of linearity and time invariance. A *linear system* is one that obeys superposition—that is, if its output is the sequence $y_1[t]$ in response to input $x_1[t]$, and is $y_2[t]$ in response to input $x_2[t]$, then it will be $\alpha y_1[t] + \beta y_2[t]$ in response to input $\alpha x_1[t] + \beta x_2[t]$. A *time-invariant* system is one that, given the input $x[t]$ and the corresponding output $y[t]$, will provide output $y[t - t_0]$ a given input $x[t - t_0]$. In other words, shifting the input sequence in time merely shifts the output sequence by the same amount.

From a practical perspective, any computation made entirely of addition, constant coefficient multiplication, and delay operations is guaranteed to be LTI. This class of algorithms, while restricted, is extremely important; it contains all the fundamental building blocks of DSP, such as finite impulse response (FIR) and infinite impulse response (IIR) filters, together with transformations such as the discrete cosine transform (DCT), the fast Fourier transform (FFT), and many color–space conversions.

The remainder of this section assumes a basic knowledge of digital signal processing, in particular the z-transform and transfer functions. For the unfamiliar reader, Mitra [32] provides an excellent introduction. Readers unconcerned with the mechanics of peak estimation for LTI systems may simply take it as read that for such systems it is possible to obtain tight analytic bounds on peak signal values.

Transfer function calculation The analytical scaling rules derived in this section rely on a knowledge of system transfer functions. A transfer function of a discrete-time LTI system between any given I/O pair is defined to be the z-transform of the sequence produced at that output, in response to a unit impulse at that input [32]; these transfer functions may be expressed as the ratio of two polynomials in z^{-1}. The transfer function from each primary input to each signal must be calculated for signal-scaling purposes. This section considers the practical problem of transfer function calculation from a dataflow graph.

Given a dataflow graph $G(V, S)$, let $V_I \subseteq V$ be the set of input nodes, $V_O \subseteq V$ be the set of output nodes, and $V_D \subseteq V$ be the set of unit sample delay nodes. For signal scaling, a matrix of transfer functions $\boldsymbol{H}(z)$ is required, with elements $h_{iv}(z)$ for $i \in V_I$ and $v \in V$ representing the transfer function from the primary input i to the output of node v.

Calculation of transfer functions for nonrecursive systems is a simple task, leading to a matrix of polynomials in z^{-1}; a straightforward algorithm is presented by Constantinides et al. [12]. For recursive systems, it is necessary to identify a subset $V_c \subseteq V$ of nodes whose outputs correspond to a system *state*. In this context, a state set consists of a set of nodes that, if removed from the dataflow graph, would break all feedback loops. Once such a state set has been identified, transfer functions can easily be expressed in terms of the outputs of these nodes using algorithms suitable for nonrecursive computations.

Let $\boldsymbol{S}(z)$ be a z-domain matrix representing the transfer function from each input signal to the output of each of these state nodes. The transfer functions from each input to each state node output may be expressed as in equation 23.2, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are matrices of polynomials in z^{-1}. Each of these matrices represents a z-domain relationship once the feedback has been broken at the outputs of state nodes. $\boldsymbol{A}(z)$ represents the transfer functions between state nodes and state nodes, and $\boldsymbol{B}(z)$ represents the transfer functions between primary inputs and state nodes.

$$\boldsymbol{S}(z) = \boldsymbol{A}\boldsymbol{S}(z) + \boldsymbol{B}(z) \tag{23.2}$$

$$\boldsymbol{H}(z) = \boldsymbol{C}\boldsymbol{S}(z) + \boldsymbol{D}(z) \tag{23.3}$$

The matrices $\boldsymbol{C}(z)$ and $\boldsymbol{D}(z)$ are also matrices of polynomials in z^{-1}. $\boldsymbol{C}(z)$ represents the z-domain relationship between state node outputs and the outputs of all nodes. $\boldsymbol{D}(z)$ represents the z-domain relationship between primary inputs and the outputs of all nodes.

It is clear that $\boldsymbol{S}(z)$ may be expressed as a matrix of rational functions (equation 23.4), where $\boldsymbol{I}$ is the identity matrix of appropriate size. This allows the transfer function matrix $\boldsymbol{H}(z)$ to be calculated directly from equation 23.3.

$$\boldsymbol{S}(z) = (\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{B} \tag{23.4}$$

Example Consider the simple dataflow graph from Section 23.1.1, shown in Figure 23.1. Clearly, removal of any one of the four internal nodes (adder, gain, delay, or the signal branch) from it will break the feedback loop. Let us arbitrarily choose the adder node as a state node and choose the gain coefficient to be 0.1. The polynomial matrices $\boldsymbol{A}(z)$ to $\boldsymbol{D}(z)$ may then be calculated (equation 23.5).

$$\begin{aligned} \boldsymbol{A}(z) &= 0.1z^{-1} \\ \boldsymbol{B}(z) &= 1 \\ \boldsymbol{C}(z) &= [0\ 1\ 0.1\ 0.1\ 0.1\ 0.1z^{-1}]^T \\ \boldsymbol{D}(z) &= [1\ 0\ 0\ 0\ 0\ 0]^T \end{aligned} \tag{23.5}$$

Calculation of $\boldsymbol{S}(z)$ may then proceed following equation 23.4, yielding equation 23.6. Finally, the matrix $\boldsymbol{H}(z)$ can be constructed following equation 23.3, giving equation 23.7.

$$\boldsymbol{S}(z) = 1/(1-0.1z^{-1}) \tag{23.6}$$

$$\boldsymbol{H}(z) = [1\ 1/(1-0.1z^{-1})\ 0.1/(1-0.1z^{-1})\ 0.1/(1-0.1z^{-1})\ 0.1/(1-0.1z^{-1})\ 0.1z^{-1}/(1-0.1z^{-1})]^T \tag{23.7}$$

The runtime of this algorithm grows significantly with the number of *state* signals $|V_c|$, and so selecting a small set of state signals is important. A simple approach is to select all of the delay elements in a circuit, assuming that it has no combinational cycles. Alternatively, techniques such as Levy and Low's [30] can be employed.

Scaling with transfer functions To produce the smallest fixed-point implementation, it is desirable to utilize as much as possible of the full dynamic range provided by each internal signal representation. The first step of the optimization process is therefore to choose the smallest possible value of p_j for each signal $j \in S$ in order to guarantee no overflow.

Consider a dataflow graph $G(V, S)$, annotated with wordlengths $\boldsymbol{n}$ and scalings $\boldsymbol{p}$. Recall that $V_I \subseteq V$ denotes the set of input nodes, and let us say that each such node reaches peak signal values of $\pm M_i (M_i > 0)$ for $i \in V_I$. Let $\boldsymbol{H}(z)$ be the scaling transfer function matrix defined before, with the associated impulse response matrix $h[t]$ related to the transfer function matrix through the component-wise inverse z-transform. Then the worst-case peak value P_j reached by any signal $j \in S$ is given by maximizing the well-known convolution sum (equation 23.8) [32], where $x_i[t]$ is the value of the input $i \in V_I$ at time index t.

Solving this maximization problem provides the input sequence given in equation 23.9, and allowing $N_{ij} \to \infty$ leads to the peak response at signal j given in equation 23.10. Here sgn( ) is the signum function (equation 23.11).

$$P_j = \pm \sum_{i \in V_I} \max_{x_i[t']} \left(\sum_{t=0}^{N_{ij}-1} x_i[t'-t]\, h_{ij}[t] \right) \tag{23.8}$$

$$x_i[t] = M_i \operatorname{sgn}\left(h_{ij}\lfloor N_{ij} - t - 1 \rfloor\right) \tag{23.9}$$

$$P_j = \sum_{i \in V_I} M_i \sum_{t=0}^{\infty} |h_{ij}[t]| \tag{23.10}$$

$$\operatorname{sgn}(x) = \begin{cases} 1, & x \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{23.11}$$

This worst-case approach leads to the concept of l_1 scaling, defined in the following paragraphs.

The l_1-norm of a transfer function $H(z)$ is given by equation 23.12, where $Z^{-1}\{\ \}$ denotes the inverse z-transform.

$$l_1\{H(z)\} = \sum_{t=0}^{\infty} Z^{-1}\{H(z)\}[t] \tag{23.12}$$

A dataflow graph $G(V, S)$ annotated with wordlengths $\boldsymbol{n}$ and scalings $\boldsymbol{p}$ is said to be l_1-scaled} if equation 23.13 holds for all signals $j \in S$.

$$p_j = \left\lfloor \log_2 \left(\sum_{i \in V_I} M_i l_1\{h_{ij}(z)\} \right) \right\rfloor + 1 \tag{23.13}$$

The important point about an l_1-scaled algorithm is that the scalings used are *optimal* in the following sense. If any scaling is reduced lower than its value from equation 23.13, it is possible for overflow to result on that variable. If any scaling is increased beyond its value from equation 23.13, the area of the resulting implementation increases or stays the same without any matching improvement in arithmetic quality observable at the algorithm outputs.

Data range propagation

If the algorithm under consideration is not linear or time invariant, one mechanism for estimating the peak value reached by each signal is to consider the propagation of data ranges through the computation graph. This is generally possible only for nonrecursive algorithms.

Forward propagation A naive way of approaching this problem is to examine the binary-point position that "naturally" results from each hardware operator. Such an approach, illustrated here, is an option in the Xilinx System Generator tool [20].

In the dataflow graph shown in Figure 23.2, if we consider that each input has a range $(-1, 1)$, then we require a binary-point location of $p = 0$ at each input. Let us consider each of the adders in turn. Adder `a1` adds two inputs with $p = 0$ and therefore produces an output with $p = \max(0, 0) + 1 = 1$. Adder `a2` adds one input with $p = 0$ and one with $p = 1$, and therefore produces an output with $p = \max(0, 1) + 1 = 2$. Similarly, the output of `a3` has $p = 3$, and the output of `a4` has $p = 4$. While we have successfully determined a binary-point location for each signal that will not lead to overflow, the disadvantage of this approach

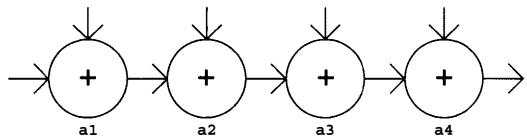


FIGURE 23.2 ■ A dataflow graph representing a string of additions.

should be clear. The range of values reachable by the system output is actually $5^*(-1,1) = (-5,5)$, so $p = 3$ is sufficient; $p = 4$ is an overkill of one MSB.

A solution to this problem that has been used in practice is to propagate data ranges rather than binary-point locations [4, 40]. To understand this approach in practice, let us apply the technique to the example of Figure 23.2. The output of adder `a1` is a subset of $(-2,2)$ and thus is assigned $p = 1$; the output of adder `a2` is a subset of $(-3,3)$ and is thus assigned $p = 2$; the output of adder `a3` is a subset of $(-4,4)$ and is thus assigned $p = 3$; and the output of adder `a4` is a subset of $(-5,5)$ and is thus also assigned $p = 3$. For this simple example, the problem of peak value detection has been solved to optimality.

However, such a tight solution is not always possible with data range propagation. Under circumstances where the dataflow graph contains one or more branches (fork nodes), which later reconverge, such a "local" approach to range propagation can be overly pessimistic. As an example, consider the computation graph representing a constant coefficient multiplication on complex numbers shown in Figure 23.3.

In the figure, each signal has been labeled with a propagated range, assuming that the primary inputs have range $(-0.6, 0.6)$. Under this approach, both outputs require $p = 2$. However, such ranges are overly pessimistic. The upper output in Figure 23.3 has the value $y_1 = 2.1x_1 - 1.8(x_1 + x_2) = 0.3x_1 - 1.8x_2$. Thus, its range can also be calculated as $0.3(-0.6, 0.6) - 1.8(-0.6, 0.6) = (-1.26, 1.26)$. A similar calculation for the lower output provides a range of $(-1.2, 1.2)$. By examining the global system behavior, we can therefore see that in reality $p = 1$ is sufficient for both outputs.

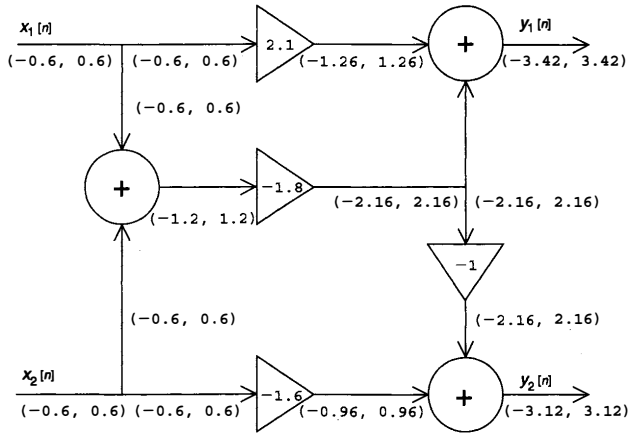


FIGURE 23.3 ■ Range propagation through a complex constant coefficient multiplier. Triangles represent (real) constant coefficient multiplication.

Note that the analytic scheme described previously for linear time-invariant systems would calculate the tighter bound in this case.

In summary, range propagation techniques may provide larger bounds on signal values than are absolutely necessary. This problem is seen *in extremis* with recursive computation graphs. In these cases, it is generally impossible to use range propagation to place a finite bound on signal values, even in cases when such a finite bound can analytically be shown to exist. Under these circumstances, it is standard practice to use some form of simulation to estimate the peak value of signals.

23.2.2 Simulation-based Peak Estimation

A completely different approach to peak estimation is to use simulation—that is, to actually run the algorithm with one or more provided input datasets and measure the peak values reached by each signal.

In its simplest form, the simulation approach consists of measuring the peak signal value P_j reached by a signal $j \in S$ and then setting $p = \lfloor \log_2 kP_j \rfloor + 1$, where $k > 1$ is a user-supplied "safety factor" (typically 2 to 4). Thus, it is ensured that no overflow will occur so long as the signal value does not exceed kP_j when excited by a different input sequence. Particular care must therefore be taken to select an appropriate test sequence.

Kim et al. [25] extend the simulation approach by considering more complex forms of the safety factor. In particular, it is possible to extract information from the simulation relating to the class of probability density function followed by each signal. A histogram of the data values for each signal is built, and from it the distribution is classified as unimodal or multimodal, symmetric or nonsymmetric, and zero mean or nonzero mean. Different forms of safety factor are applied in each case.

Simulation approaches are appropriate for nonlinear or time-varying systems, for which data range propagation, described in Section 23.1.2, provides overly pessimistic results (such as for recursive systems). The main drawback of simulation-based approaches is the significant dependence on the input dataset used for simulation; moreover, usually no general guidelines can be given for how to select an appropriate input. These approaches can, of course, be combined with the analytical techniques of Section 23.2.1 [13].

There has been some recent work [34] aiming to put the derivation of safety factors on a sound theoretical footing by using the statistical theory of extreme value distributions [26]. It is known that the distribution of the sum of a large number of statistically independent identically distributed (i.i.d.) random variables approaches the Gaussian distribution (the Central Limit Theorem). What is less well known is that the (scaled) maximum value of a large number of i.i.d. variables also approaches one of three possible distributions, no matter the distribution of the variables themselves. These are the Gumbel, Fréchet, and Weibull distributions [26]. Using this property, and making an assumption on the type of distribution converged to (Özer and colleagues [34] assume Gumbel), provides a statistically sound way of estimating the safety factor required for a given arbitrarily small probability of overflow.

23.2.3 Summary of Peak Estimation

The optimization of a bit-parallel fixed-point datapath can be split into the two problems of determining an appropriate scaling and determining an appropriate wordlength for each signal. We have discussed the first of these two problems in detail. It has been shown that in the case of LTI systems, tight analytic bounds can be placed on the scaling required. Analytic scaling is also possible for non-LTI systems, at the cost of tightness in the bound—disastrously so in the case of recursive systems. The alternative to the analytical approach is the use of simulation on trusted input datasets; some progress has recently been made on the issue of statistically sound simulation-based peak determination.

23.3 WORDLENGTH OPTIMIZATION

Once a scaling has been determined, it is necessary to find an appropriate wordlength for each signal. While optimizing the scaling usually improves circuit quality without changing circuit functionality (assuming no overflows occur), wordlength optimization trades circuit quality (area, delay, power) for result accuracy. The major problem in wordlength optimization is to determine the error at system outputs for a given set of wordlengths and scalings of all internal variables. We will call this problem *error estimation*. Once a technique for error estimation has been selected, the wordlength selection problem reduces to utilizing the known area and error models within a constrained optimization setting: Find the minimum area implementation satisfying certain constraints on arithmetic error at each system output.

The majority of this section is taken up with the problem of error estimation (Section 23.3.1). Following on from this discussion, the problem of area modeling is addressed. Optimization techniques suitable for solving the wordlength determination problem are introduced (Section 23.3.2), with some discussion of the problem's inherent computational complexity.

23.3.1 Error Estimation and Area Models

Traditionally, much of the research on estimating the effects of truncation and roundoff noise in fixed-point systems has focused on DSP uniprocessors. This leads to certain constraints and assumptions on quantization errors—for example, that the wordlength of all signals is the same, that quantization is performed after multiplication, and that the wordlength before quantization is much greater than that following it [36]. The multiple-wordlength paradigm allows a more general design space to be explored, free from these constraints.

The effect of using finite register length in fixed-point systems has been studied for some time. Oppenheim and Weinstein [36] and Liu [29] lay down standard models for quantization errors and error propagation through LTI systems based on a linearization of signal truncation or rounding. Error signals, assumed to be uniformly distributed, uncorrelated with each other and

with themselves over time, are added whenever a truncation occurs. This approximate model has served very well because quantization error power is dramatically affected by wordlength in a uniform wordlength structure, decreasing at approximately 6 dB per bit. This means that it is not necessary to have highly accurate models of quantization error power in order to predict the required signal width [35]. In a multiple-wordlength circuit, the implementation error power may be adjusted much more finely, and so the resulting implementation tends to be more sensitive to errors in estimation. This has led to a simple refinement of the model, which will be discussed soon.

The most generally applicable method for error estimation is simulation: Simulate the system with a given "representative" input and measure the deviation at the system outputs when compared to an accurate simulation (usually "accurate" means IEEE double-precision floating point [22]). Indeed, this is the approach taken by several systems [6, 27]. Unfortunately, simulation suffers from several drawbacks, some of which correspond to the equivalent simulation drawbacks discussed in Section 23.2, and some of which are peculiar to the error estimation problem.

First, there is the problem of dependence on the chosen "representative" input dataset. Second, there is the problem of speed: Simulation runs can take a significant amount of time, and during an optimization procedure a large number of simulation runs may be needed. Third, even the "accurate" simulation will have errors induced by finite wordlength effects that, depending on the system, may not be negligible.

We will be using signal-to-noise ratio (SNR), sometimes referred to as signal-to-quantization-noise ratio (SQNR), as a generally accepted metric for measuring the quality of a fixed-point algorithm implementation [32] (although other measures, such as maximum instantaneous error, exist). Conceptually, the output sequence at each system output resulting from a particular finite-precision implementation can be subtracted from the equivalent sequence resulting from an infinite-precision implementation. The difference is known as the *fixed-point error*.

The ratio of the output power (i.e., the sum of squared signal values) resulting from an infinite precision implementation to the fixed-point error power of a specific implementation defines the SNR. For the purposes of this chapter, the signal power at each output is fixed because it is determined by a combination of the input signal statistics and the dataflow graph $G(V, S)$. To explore different implementations of the dataflow graph, it is therefore sufficient to concentrate on noise estimation, which is the subject of this section.

The approach taken to wordlength optimization should depend on the mathematical properties of the system under investigation. After briefly considering simulation-based estimation, we will examine analytic or semi-analytic techniques that may be applied to certain classes of system. Next we will describe one such method, which may be used to obtain high-quality results for linear time-invariant algorithms. Then we will generalize this approach to nonlinear systems containing only differentiable nonlinear components.

Simulation-based methods

Simulation-based methods for wordlength optimization were first established at Seoul National University, and some of them have been integrated into the Signal Processing Worksystem of Cadence.

In Kim et al. [25] and Kum and Sung [27], the search space is reduced by grouping together all variables involved in a multiply–add operation and optimizing them as a single-wordlength "block." Within each block, the Oppenheim model of quantization noise is applied [35].

Although simulation is almost certainly the most widespread mechanism for estimating the impact of a given choice of wordlength, it suffers from the drawbacks discussed earlier. Indeed, the dependence of the result on the input dataset, while widely acknowledged, is rarely considered in depth. The class of algorithm for which simulation forms a suitable mechanism has also remained unclear. Recently, Alippi [1] proposed an analytical framework within which the question of simulation input dependence can be addressed. A mechanism for understanding the perturbation of Lebesgue-measurable functions, an extremely wide class of algorithmic behavior, has been proposed that uses the theory of randomized algorithms. The essential contribution of this work, for the purposes of fixed-point analysis, has been to demonstrate that simulation is an appropriate mechanism for analyzing fixed-point error. Moreover, Alippi [1] provides a theoretically sound guideline on the number of simulations required in order to be confident, to within a certain probability, that the SNR is within a given limit (alternative signal quality metrics are also Lebesgue measurable and hence can be used as well).

An analytic technique for linear time-invariant systems

We will first address error estimation for LTI systems. An appropriate noise model for truncation of LSBs is described in the subsection that follows. It is then shown that the noise injected through truncation can be analytically propagated through the system in order to measure the effect of such noise on system outputs.

Noise model A common assumption in DSP design is that signal quantization (rounding or truncation) occurs only after a multiplication or multiply–accumulate operation. This corresponds to a uniprocessor viewpoint, where the result of an n-bit signal multiplied by an n-bit coefficient needs to be stored in an n-bit register. The result of such a multiplication is an $n' = 2n$-bit word, which must therefore be quantized down to n bits. Considering signal truncation, the least area-expensive method of quantization [18], the lowest value of the truncation error in 2's complement with $p = 0$, is $2^{-n'} - 2^{-n} \approx -2^{-n}$, and the highest value is 0 (2's complement truncation error is always nonpositive).

It has been observed that values between these values tend to be equally likely to occur in practice, so long as the $2n$-bit signal has sufficient dynamic range [29, 36]. This observation leads to the formulation of a uniform distribution model [36] for the noise of variance $\sigma^2 = 2^{-2n}/12$ for the standard normalization of $p = 0$. It has also been observed that, under the same conditions, the

spectrum of such errors tends to be white because there is little correlation between low-order bits over time even if there is a correlation between high-order bits. Similarly, different truncations occurring at different points within the implementation structure tend to be uncorrelated.

When considering a multiple-wordlength implementation, or truncation at different points within the datapath, some researchers have opted to carry the uniform distribution model over to the new implementation style [25]. However, there are associated inaccuracies involved in such an approach [7]. First, quantizations from n' bits to n bits, where $n' \approx n$, will suffer in accuracy because of the discretization of the error probability density function; for example, if $p = 0$, $n' = 2$, $n = 1$, then the only possible error values are 0 and $-1/4$. Second, in such cases the lower bound on error can no longer be simplified in the preceding manner because $2^{-n'} - 2^{-n} \approx -2^{-n}$ no longer holds.

These two issues may be resolved by considering a discrete probability distribution for the injected error signal. For 2's complement arithmetic, the truncation error injection signal $e[t]$ caused by truncation from (n', p) to (n, p) is bounded by equation 23.14.

$$-2^p\left(2^{-n} - 2^{-n'}\right) \le e[t] \le 0 \tag{23.14}$$

It is assumed that each possible value of $e[t]$ has equal probability, as discussed earlier. For 2's complement truncation, there is nonzero mean $E\{e[t]\}$ (equation 23.15) and variance σ_e^2 (equation 23.16).

$$E\{e[t]\} = -\frac{1}{2^{n'-n}} \sum_{i=0}^{2^{n'-n}-1} i \cdot 2^{p-n} = -2^{p-1}\left(2^{-n} - 2^{-n'}\right) \tag{23.15}$$

$$\sigma_e^2 = \frac{1}{2^{n'-n}} \sum_{i=0}^{2^{n'-n}-1} \left(i \cdot 2^{p-n'}\right)^2 - E^2\{e[t]\} = \frac{1}{12} 2^{2p}\left(2^{-2n} - 2^{-2n'}\right) \tag{23.16}$$

Note that for $n_1 \gg n_2$ and $p = 0$, equation 23.16 simplifies to $\sigma_e^2 \approx 1/12\ \ 2^{-2n}$, which is the well-known predicted error variance of Oppenheim and Schafer [35] for a model with continuous probability density function.

Noise propagation and power estimation If it is our aim to optimize the wordlengths used in a design, then it is important to be able to predict the arithmetic quality observable at the design outputs. Given a set of wordlengths and scalings, it is possible to use the truncation model described in the previous section to predict the variance of each injection input. For each signal $j \in S$, a straightforward application of equation 23.16 may be used, with n_1 equal to the "natural" full-precision wordlength produced by the source component, $n_2 = n_j$, and $p = p_j$.

By constructing noise sources in this manner for the entire dataflow graph, a set $F = \{(\sigma_p^2, \boldsymbol{R}_p)\}$ of injection input variances σ_p^2, and their associated transfer function to each primary output $\boldsymbol{R}_p(z)$, can be constructed. From this set it is possible to predict the nature of the noise appearing at the system primary

outputs, which is the quality metric of importance to the user. Since the noise sources have a white spectrum and are uncorrelated with each other, it is possible to use L_2 scaling to predict the noise power at the system outputs. The L_2 norm of a transfer function $H(z)$ is defined in equation 23.17, where Z^{-1} denotes the inverse z-transform. It can be shown that the noise variance E_k at output k is given by equation 23.18.

$$L_2\{H(z)\} = \left(\sum_{n=0}^{\infty} \left| Z^{-1}\{H(z)\}[n] \right|^2 \right)^{1/2} \tag{23.17}$$

$$E_k = \sum_{(\sigma^2, R) \in F} \sigma^2 {L_2}^2 \{R_k\} \tag{23.18}$$

A hybrid approach for nonlinear differentiable systems

With some modification, some of the results from the preceding section can be carried over to the more general class of nonlinear time-varying systems containing only differentiable nonlinearities. In this section we address one possible approach to this problem, deriving from the type of small-signal analysis typically used in analogue electronics [12, 38].

Perturbation analysis To make some of the analytical results on error sensitivity for LTI systems applicable to nonlinear systems, the first step is to linearize these systems. The assumption is made that the quantization errors induced by rounding or truncation are sufficiently small not to affect the system's macroscopic behavior. Under such circumstances, each system component can be locally linearized or replaced by its "small-signal equivalent" [38] in order to determine the output behavior under a given rounding scheme.

We will consider one such n-input component, the differentiable function $Y[t] = f(X_1[t], X_2[t], \ldots, X_n[t])$, where t is a time index. If we denote by $x_i[t]$ a small perturbation on variable $X_i[t]$, then a first-order Taylor approximation for the induced perturbation $y[t]$ on $Y[t]$ is given by equation 23.19.

$$y[t] \approx x_1[t] \left. \frac{\partial f}{\partial X_1} \right|_t + \ldots + x_n[t] \left. \frac{\partial f}{\partial X_n} \right|_t \tag{23.19}$$

Note that this approximation is linear in each x_i but that the coefficients may vary with time index t because, in general, $\partial f / \partial X_1$ is a function of $X_1, X_2, \ldots, X_n$. Thus, by applying such an approximation, we have produced a linear time-varying small-signal model for a nonlinear time-invariant component. Such an analysis is readily extended to a time-varying component by expressing $Y[t] = f(t, X_1[t], X_2[t], \ldots, X_n[t])$.

The linearity of the resulting model allows us to predict the error at system outputs due to *any* linear scaling of a small perturbation of signal $j \in S$ analytically, given the simulation-obtained error from a *single* such perturbation instance at j, which can be obtained by a single simulation run. Thus, this method can be considered to be a hybrid analytic/simulation error analysis [15].

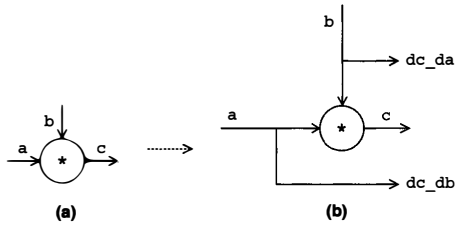


FIGURE 23.4 ■ A local graph transformation to insert derivative monitors: (a) multiplier node; (b) with derivative monitors.

Derivative monitors To construct the small-signal model, we must first evaluate the differential coefficients of the Taylor series model for nonlinear components.

In general, methods must be introduced to calculate the differential of each nonlinear node type. This is performed by applying a graph transformation to the dataflow graph, introducing the necessary extra nodes and outputs to do this calculation.

The general multiplier is the only nonlinear component considered explicitly in this section, although the approach is general; the graph transformation for multipliers is illustrated in Figure 23.4. Since $f(X_1, X_2) = X_1X_2$, $\partial f/\partial X_1 = X_2$ and $\partial f/\partial X_2 = X_1$.

After insertion of the monitors (`dc_da` and `dc_db`, which capture the derivatives of `c` with respect to `a` and `b`, respectively), a simulation may be performed to write the derivatives to appropriate data files to be used by the linearization process, which is described next.

Linearization Our aim is to construct a small-signal model, which can be simulated to determine the sensitivity to rounding errors. Once we have obtained the derivative monitors, the construction of the small-signal model may proceed, again through graph transformation. All linear components (adder, constant coefficient multiplier, fork, delay, primary input, primary output) remain unchanged as a result of the linearization process. Each nonlinear component is replaced by its first-order Taylor model. Additional primary inputs are added to the dataflow graph to read the Taylor coefficients from the derivative monitor files created by the previous large-signal simulation.

As an example, the Taylor expansion transformation for the multiplier node is illustrated in Figure 23.5. The inputs `dc_da` and `dc_db` are themselves time-varying sequences, derived from the previous step of the procedure. Note that the graph portion of Figure 23.5(b) still contains multiplier "nonlinear" components, although one input of each multiplier node is now external to the model. This absence of feedback ensures linearity, although not time invariance.

Noise injection In Section 23.3.1, L_2 scaling was used to analytically estimate the noise variance at a system output through scaling of the (analytically

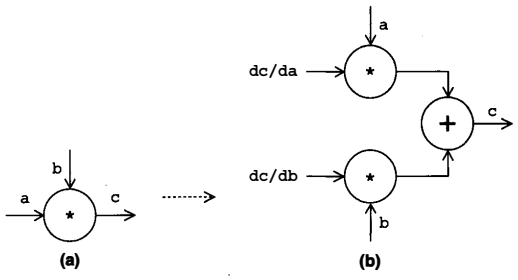


FIGURE 23.5 ■ A local graph transformation to produce a small-signal model: (a) multiplier node; (b) first-order Taylor model.

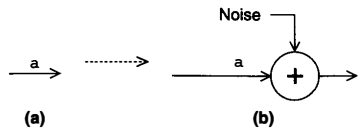


FIGURE 23.6 ■ A local graph transformation to inject perturbations: (a) original signal; (b) with noise injection.

derived) noise variance injected at each point of quantization. Such a purely analytic technique can be used only for LTI systems. In this section we discuss an extension of the approach for nonlinear systems.

Because the small-signal model is linear, if an output exhibits variance V when excited by an error of variance σ^2 injected into a given signal, then the output will exhibit variance αV when excited by a signal of variance $\alpha\sigma^2$ injected into the same signal ($\alpha \geq 0$). Herein lies the strength of the proposed linearization procedure: If the output response to a noise of known variance can be determined *once only* through simulation, this response can be scaled with analytically derived coefficients in order to estimate the response to any rounding or truncation scheme.

Thus, the next step of the procedure is to transform the graph through the introduction of an additional adder node, and associated signals, and then simulate the graph with a known noise. In our case, to simulate truncation of a 2's complement signal, the noise is independent and identically distributed with a uniform distribution over the range $[-2\sqrt{3}, 0]$, chosen to have unit variance ($1/12(2\sqrt{3})^2 = 1$), in this way making the measured output response an unscaled "sensitivity" measure. The graph transformation of inserting a noise injection is shown in Figure 23.6. One of these transformations is applied to a distinct copy of the linearized graph for each signal in the dataflow graph,

after which zeros are propagated from the *original* primary inputs, to finalize the small-signal model. This is a special case of constant propagation [2] that leads to significantly faster simulation results for nontrivial dataflow graphs.

The entire process is illustrated for a simple dataflow graph in Figure 23.7. The original graph is shown in (a). The perturbation analysis will be performed for the signals marked (*) and (**). After inserting derivative monitors

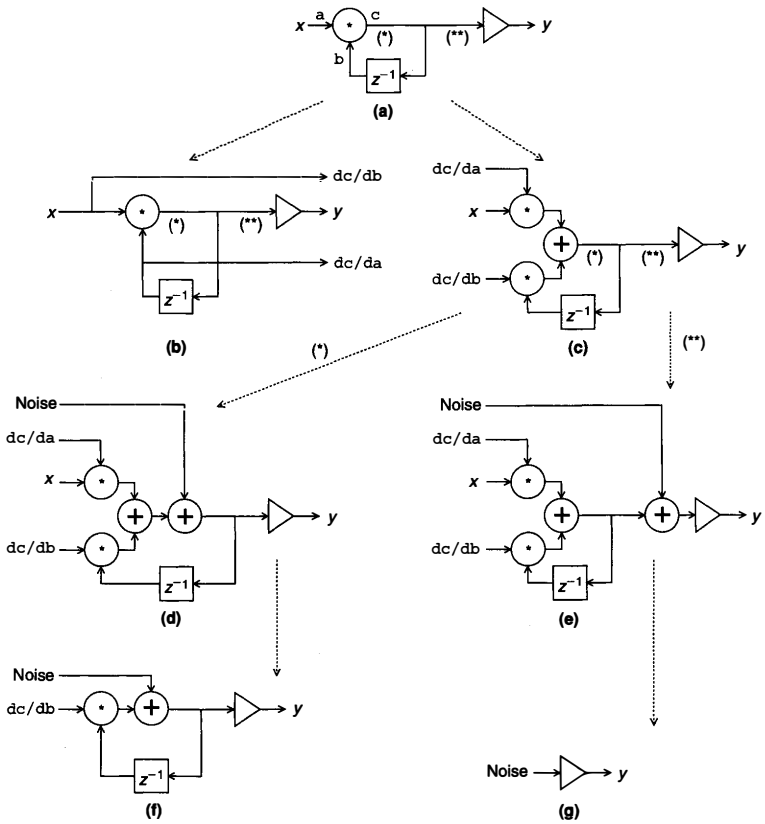


FIGURE 23.7 ■ An example of perturbation analysis: (a) original dataflow graph; (b) transformed dataflow graph; (c) linearized dataflow graph; (d) variant for (*) signal; (e) variant for (**) signal; (f) simplified graph for (*) signal; (g) simplified graph for (**) signal.

for nonlinear components, the transformed DFG is shown in (b). The linearized DFG is shown in (c), and its two variants for the signals (*) and (**) are illustrated in (d) and (e), respectively. Finally, the corresponding simplified DFGs after zero propagation are shown in (f) and (g), respectively.

High-level area models

To implement a multiple-wordlength system, component libraries must be available to support multiple-wordlength arithmetic. These libraries can then be instantiated by the synthesis system and must be modeled in terms of area consumption to provide the wordlength optimization procedure with a cost metric.

Integer arithmetic libraries are available from FPGA vendors (e.g., Xilinx Coregen or Altera LPM macros). Parameterizable macros for standard arithmetic functions operating on integer arithmetic form the basis of the multiple-wordlength libraries synthesized to by wordlength optimization tools such as Right-Size [15] and Synoptix [8]. Blocks from each of these vendors may have slightly different cost parameters, but the general approach described in this section is applicable across all of them. Example external interfaces of multiple-wordlength library blocks for constant coefficient multipliers (`gain`) and adders (`add`) written in VHDL are shown in Listing 23.1 [23].

Listing 23.1 ■ Constant coefficient multipliers (`gain`) and adders (`add`) written in VHDL.

```
ENTITY gain IS
 GENERIC( INWIDTH, OUTWIDTH, NULLMSBS, COEFWIDTH : INTEGER;
          COEF : std_logic_vector( COEFWIDTH downto 0 ) );
 PORT( data : IN std_logic_vector( INWIDTH downto 0 );
       result : OUT std_logic_vector( OUTWIDTH downto 0 ) );
END gain;

ENTITY add IS
 GENERIC( AWIDTH, BWIDTH, BSHL, OUTWIDTH, NULLMSBS : INTEGER );
 PORT( dataa : IN std_logic_vector( AWIDTH downto 0 );
       datab : IN std_logic_vector( BWIDTH downto 0 );
       result : OUT std_logic_vector( OUTWIDTH downto 0 ) );
END add;
```

As well as an individually parameterizable wordlength for each input and output port, each library block has a `NULLMSBS` parameter that indicates how many most significant bits (MSBs) of the operation result are to be ignored (the converse of sign extension). Thus, each operation result can be considered to be made up of zero or more MSBs that are ignored, followed by one or more data bits, followed by zero or more LSBs that may be truncated depending on the `OUTWIDTH` parameter. For the adder library block, there is an additional `BSHL` generic that accounts for the alignment necessary for addition operands. `BSHL` represents the number of bits by which the `datab` input must be conceptually shifted left to align it with the `dataa` input. Note that, because this is fixed-point arithmetic, there is no physical shifting involved; the data is simply aligned in a

skewed manner, as shown in Figure 23.8. Note, too, that `dataa` and `datab` are permuted as necessary to ensure that `BSHL` is always nonnegative.

In the figure, (a) shows that the MSB of input `b` protrudes beyond that of input `a` and that all the output bits are drawn from the core integer addition of the overlap. Figure 23.8(b) shows that the MSB of input `a` protrudes beyond that of input `b` and that all output bits are drawn from the core integer addition of the overlap. Figure 23.8(c) shows that the MSB of input `b` protrudes beyond that of input `a` but that some of the output bits are drawn from the LSB overhang of input `a` and are thus produced "free." Figure 23.8(d) shows that the MSB of input `a` protrudes beyond that of input `b` but that some of the output bits

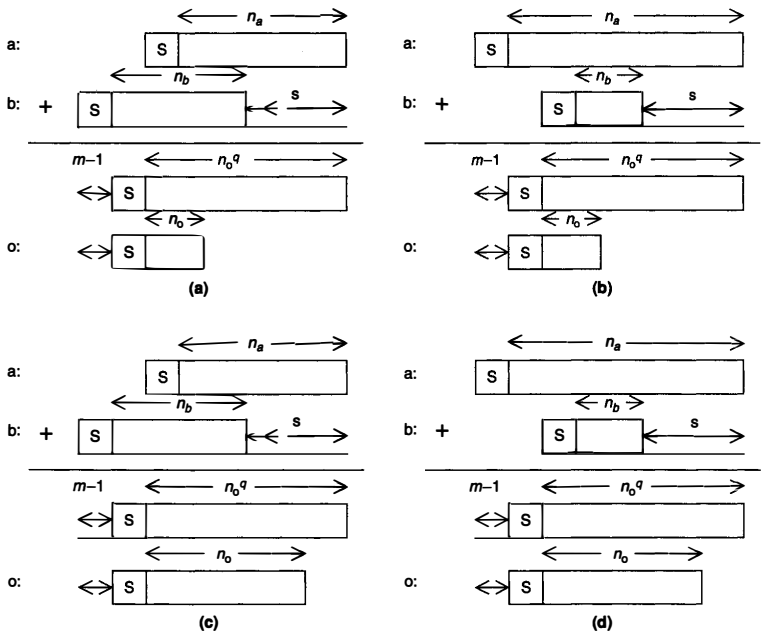


FIGURE 23.8 ■ Four multiple-wordlength adder formats arising in practice: (a) MSB of input `b` protruding beyond MSB of input `a`; (b) MSB of input `a` protruding beyond MSB of input `b`; (c) MSB of input `b` protruding beyond MSB of input `a`, with "free" output bits; (d) MSB input `a` protruding beyond MSB of input `b`, with "free" output bits. (`s` denotes the value of the BSHL generic; `m` denotes the value of the `NULLMSBS` generic.)

are drawn from the LSB overhang of input a and are thus produced "free." In each case, the upper result shows the "error-free" wordlength n_o^q without further truncation, whereas the lower result shows the wordlength n_o after potential further truncation.

Each of the library block parameters has an impact on the area resources consumed by the overall system implementation. It is generally assumed when constructing a cost model that each operator in the dataflow graph will map to a separate hardware resource and that the area cost of wiring is negligible [17]. These assumptions (relaxed by Constantinides et al. [12]) simplify the construction of an area cost model. It is sufficient to estimate separately the area consumed by each computation node and then sum the resulting estimates. In reality, of course, logic synthesis, performed after wordlength optimization, is likely to result in some logic optimization between the boundaries of two connected library elements. This may result in lower area than estimated, but experience shows that these deviations from the area model are small.

The area model for a multiple-wordlength adder is reasonably straightforward. A ripple-carry architecture is used [21] since FPGAs provide good support for fast ripple-carry implementations. The only area-consuming component is the core (integer) adder constructed from the vendor library. This adder has a width of max(AWIDTH − BSHL, BWIDTH) − NULLMSBS + 2 bits. Depending on the FPGA architecture in question, each bit may not consume the same area; however, because some bits are required for the result port whereas others may be needed only for carry propagation, their sum outputs remain unconnected and therefore the sum circuitry is optimized away by logic synthesis. The cost model thus has two parameters k_1 and k_2, corresponding to the area cost of a sum-and-carry full adder and to the area cost of a carry-only full adder, respectively. The area of an adder is expressed in equation 23.20.

$$\begin{aligned} &A_{\text{add}}(\texttt{AWIDTH, BWIDTH, BSHL, NULLMSBS, OUTWIDTH}) \\ &\quad = k_1(\texttt{OUTWIDTH} + 1) + k_2(\max(\texttt{AWIDTHBSHL, BWIDTH}) \\ &\qquad - \texttt{NULLMSBS} - \texttt{OUTWIDTH} + 1) \end{aligned} \tag{23.20}$$

Area estimation for general multipliers can proceed in a similarly straightforward way. However, the equivalent problem for constant coefficient multipliers is significantly more problematic. A constant coefficient multiplier is typically implemented as a series of additions through a recoding scheme such as the classic Booth technique [3]. This implementation style causes the area consumption to be highly dependent on the coefficient value. In addition, the exact implementation scheme used by the vendor integer arithmetic libraries is known only to the vendor.

A simple area model has been proposed (equation 23.21) and the coefficient values k_3 and k_4 have been determined through the synthesis of several hundred multipliers of different coefficient values and widths [12]. The model has then been fitted to this data using a least-squares approach. Note that the model does not account for NULLMSBS because, for a properly scaled coefficient,

NULLMSBS ≤ 1 for a constant coefficient multiplier and therefore has little impact on area consumption.

$$A_{\text{gain}}(\text{INWIDTH, OUTWIDTH, COEFWIDTH}) = k_3\,\text{COEFWIDTH}(\text{INWIDTH} + 1) + k_4(\text{INWIDTH} + \text{COEFWIDTH} - \text{OUTWIDTH}) \quad (23.21)$$

More detailed area models for components are discussed by Chang and Hauck [14].

23.3.2 Search Techniques

A heuristic search procedure

Because the wordlength optimization problem is NP-hard [16], several heuristic approaches have been developed to find feasible wordlength vectors having small, though not necessarily optimal, area consumption. An example heuristic is shown in Listing 23.2. After performing binary-point estimation using the techniques of Section 23.2, the algorithm determines the minimum uniform wordlength satisfying all error constraints. The design at this stage corresponds to a standard uniform wordlength design with implicit power-of-two scaling, such as may be used for an optimized uniprocessor-based implementation. Each wordlength is then scaled up by a factor $k > 1$, which represents a bound on the largest value that any wordlength in the final design may reach (in the Synoptix implementation of this algorithm [8], $k = 2$ has been used).

The resulting structure forms a starting point from which one signal wordlength is reduced by one bit on each iteration. The signal wordlength to reduce is decided in each iteration by reducing each wordlength in turn until it violates an output noise constraint (Listing 23.2). At this point there is likely to have been some pay-off in reduced area, and the signal whose wordlength reduction provided the largest pay-off is chosen. Each signal's wordlength is explored using a binary search.

Listing 23.2 ■ Algorithm wordlength falling.

```
Input: A Dataflow Graph G(V,S) and binary-point vector p.
Output: An optimized wordlength vector n.
begin
  Let the elements of S be denoted as S = {j1, j2, ..., j|S|}
  Determine u, the minimum uniform wordlength satisfying error
    criteria
  Set n ← 1ku
  do
    currentcost ← AREA(n)
    foreach ji∈S do
      bestmin ← currentcost
      Set w to the smallest positive value where the error criteria
       are satisfied for wordlength [n1 ... ni-1 w ni+1 ... n|S|]
      Set minval ← AREA([n1 ... ni-1 w ni+1 ... n|S|])
      if minval < bestmin, set bestsig ← i and bestmin ← minval
    end foreach
```

```
    if bestmin < currentcost
      n_bestsig ← n_bestsig − 1
  while bestmin < currentcost
end
```

Alternative search procedures

The algorithm described in Section 23.3.1 is a heuristic; it does not guarantee to produce the optimum area cost for a given set of error constraints. A technique to discover the true optimum-wordlength vectors has also been proposed [10] that uses integer linear programming (ILP) to model the constraint space and objective functions. This technique was able to demonstrate that the heuristic from Section 23.1.1 provides good-quality results for the small benchmark problems addressed by both approaches. Like all NP-hard problems [16], however, finding the optimum solution becomes computationally infeasible for large problem sizes. The methodology of Constantinides et al. [10] is applicable only for very small practical problems and is thus more of a theoretical than practical interest.

Several other heuristic search procedures have been proposed in the literature, and we will review some of the more interesting ones (further comparisons are made in the brief survey by Cantin et al. [6]).

An approach used by Kum and Sung [27] is based on the intuition that the error observable at a system output reduces monotonically with each wordlength in that system. This is a plausible conjecture, but is not always the case. Indeed, it was shown independently by Constantinides [9] and Lehtinen and Renfors [31] that this conjecture may be violated in practical situations. Nevertheless, if we accept it for the moment, a natural search procedure becomes apparent. We may divide the search into two phases. In the first phase, the system is simulated with all but one variable having a very large precision (e.g., double precision floating point). In this way, we can find the point at which the output constraints are violated because of quantization on this variable alone. Repeating this for all variables provides, under the conjecture, a lower bound on each element of the wordlength vector. The second phase of the algorithm is invoked if the constraints are violated when these lower bounds are used as the wordlength vector. In this case, the precision of all variables is increased by an equal number of bits until the constraints are satisfied. A variation on the second phase is to exhaustively explore all possibilities above this lower bound, until the constraints are satisfied [27].

The common meta-heuristics of simulated annealing and genetic algorithms have been used for this problem—for example, by Chang and Hauck [14]—(using a linear combination of area and error as an objective function [28,40]). While there are practical advantages to using tried-and-tested meta-heuristics for combinatorial problems, the smooth nature of the constraints and objectives, as outlined previously, means that it is likely that better results can be obtained within a fixed computation time budget by using application-specific heuristic techniques.

23.4 SUMMARY

This chapter introduced the fundamental problems of designing optimized fixed-point arithmetic circuits in custom hardware, including FPGA devices. The fixed-point number system is of widespread interest in the FPGA community because of the highly efficient arithmetic implementations possible when compared to what can be achieved with floating-point arithmetic. However, much more than with floating point, working with fixed point requires designers to have a good grasp of the numerical robustness issues involved with their designs. Performing such design by hand is tedious and error prone, which has motivated the development of automatic procedures, some of which have been described in this chapter.

The freedom in custom hardware to use multiple wordlengths in a design creates the possibility of shaping the circuit datapath to the requirements of the algorithm, leading to low-area, high-speed, and low-power implementations. This emerging paradigm throws up a new challenge, however: wordlength optimization.

This chapter demonstrated that wordlength determination can be considered as a constrained optimization, and suitable models were presented for FPGA-based bit-parallel implementations, together with signal-to-noise ratio of linear time-invariant and differentiable nonlinear time-varying systems. In each case, we described at least one error estimation procedure in depth and discussed related procedures and their advantages and disadvantages.

We will now consider some fruitful avenues for further research in this field, broken down into MSB-side optimization, error modeling, and search procedures.

The work discussed in Section 23.2 either avoids overflow completely (e.g., l_1-scaling) or reduces the probability of overflow to an arbitrary level (e.g., extreme value theory) without considering the effect of overflow on signal-to-noise ratio or other accuracy metrics. In algorithms where the worst-case variable range is much larger than the average-case range, it may make sense to save area by allowing rare overflow and its consequent reduction in arithmetic accuracy. This problem was discussed by Constantinides et al. [11] using a simple model of the error induced by overflow, based on approximating all signals by Gaussian random variables. The results achieved were weakened, however, by an inability of the proposed method to accurately estimate the correlations between overflow errors at different points within the algorithm. Further work could provide much stronger bounds.

The analytical error-modeling approaches discussed in Section 23.3.1 can adequately deal with linear time-invariant systems or with time-varying systems containing only differentiable nonlinearities. This still leaves open the problem of adequately modeling systems containing nondifferentiable nonlinearities. This is a serious omission, as it includes any algorithm containing conditionally executed statements, where the condition is a logical expression containing variables generated by the algorithm itself (in the case where the variables

are external inputs, this can be viewed as a time-varying differentiable system). Further work incorporating the results from the analysis of nonlinear dynamical systems is likely to shed new light here.

Both heuristic and optimal search procedures were discussed in Section 23.3.2. One of the limitations of the optimal approach from Constantinides et al. [10] is that is has relied on coercing inherently nonlinear constraints into a linear form, resulting in a large ILP problem. Branch-and-bound, or other combinatorial search procedures, on top of bounding procedures from the more general field of nonlinear mathematical programming may be able to provide optimal results for significantly larger problems. Further effort is also called for in the development of heuristic search procedures. None of the heuristics presented thus far can guarantee a bounded distance to optimality, although under certain error metrics the wordlength optimization problem is approximatible in this sense. It would be useful to concentrate efforts on heuristics that do provide these guarantees.

It is my belief that, apart from a practical design problem, the problem of wordlength optimization has much to offer in terms of understanding the numerical properties of algorithms. The earliest contributions to this subject can be traced back to two giants of computing, Alan Turing [39] and John von Neumann [33]. At the time, IEEE standard floating point was nonexistent, and it was necessary to carefully design the architecture around the algorithm. FPGA-based computing has reopened this method of design by giving an unprecedented degree of freedom in the implementation of numerical algorithms.

References

[1] C. Alippi. Randomized algorithms: A system-level poly-time analysis of robust computation. *IEEE Transactions on Computers* 51(7), 2002.

[2] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[3] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal Mechanical Applications of Mathematics* 4(2), 1951.

[4] A. Benedetti, P. Perona. Bit-width optimization for configurable DSPs by multi-interval analysis. *Proceedings of the 34th Asilomar Conference on Signals, Systems and Computers*, 2000.

[5] M.-A. Cantin, Y. Savaria, P. Lavoie. An automatic word length determination method. *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2001.

[6] M.-A. Cantin, Y. Savaria, P. Lavoie. A comparison of automatic word length optimization procedures. *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2002.

[7] G. A. Constantinides, P. Y. K. Cheung, W. Luk. Truncation noise in fixed-point SFGs. *IEE Electronics Letters* 35(23), November 1999.

[8] G. A. Constantinides, P. Y. K. Cheung, W. Luk. The multiple wordlength paradigm. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April–May 2001.

[9] G. A. Constantinides. *High-level Synthesis and Wordlength Optimization for Digital Signal Processing Systems*, Ph.D. thesis, University of London, 2001.
[10] G. A. Constantinides, P. Y. K. Cheung, W. Luk. Optimum wordlength allocation. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.
[11] G. A. Constantinides, P. Y. K. Cheung, W. Luk. Synthesis of saturation arithmetic architectures. *ACM Transactions on Design Automation of Electronic Systems* 8(3), 2003.
[12] G. A. Constantinides, P. Y. K. Cheung, W. Luk. *Synthesis and Optimization of DSP Algorithms*, Kluwer Academic, 2004.
[13] M. Chang, S. Hauck. Precis: A design-time precision analysis tool. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.
[14] M. Chang, S. Hauck. Automated least-significant bit datapath optimization for FPGAs. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.
[15] G. A. Constantinides. wordlength optimization for differentiable nonlinear systems. *ACM Transactions on Design Automation for Electronic Systems*, January 2006.
[16] G. A. Constantinides, G. J. Woeginger. The complexity of multiple wordlength assignment. *Applied Mathematics Letters* 15, 2002.
[17] G. DeMicheli. *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
[18] P. D. Fiore. Lazy rounding. *Proceedings of the IEEE Workshop on Signal Processing Systems*, 1998.
[19] C. Fang, T. Chen, R. Rutenbar. Floating-point error analysis based on affine arithmetic. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2003.
[20] J. Hwang, B. Milne, N. Shirazi, J. Stroomer. System level tools for DSP in FPGAs. In R. Woods and G. Brebner, eds., *Processing Field Programmable Logic*, Springer-Verlag, 2001.
[21] K. Hwang. *Computer Arithmetic: Principles, Architecture and Design*, Wiley, 1979.
[22] *IEEE Standard for Binary Floating-point Arithmetic* (ANSI/IEEE Standard 991), 1986.
[23] *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis* (IEEE Standard 1076.6), 1999.
[24] C. Inacio, D. Ombres. The DSP decision: Fixed point or floating? *IEEE Spectrum* 33(9), September 1996.
[25] S. Kim, K. Kum, W. Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems II* 45(11), November 1998.
[26] S. Kotz, S. Nadarajah. *Extreme Value Distributions: Theory and Applications*, Imperial College Press, 2000.
[27] K.-I. Kum, W. Sung. Combined wordlength optimization and high-level synthesis of digital signal processing systems. *IEEE Transactions on Computer-Aided Design* 20(8), August 2001.
[28] D.-U. Lee, A. Gaffar, R. Cheung, O. Mencer, W. Luk, G. A. Constantinides. Accuracy guaranteed bit-width optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006.
[29] B. Liu. Effect of finite word length on the accuracy of digital filters—A review. *IEEE Transactions on Circuit Theory* 18(6), 1971.
[30] H. Levy, D. W. Low. A contraction algorithm for finding small cycle cutsets. *Journal of Algorithms* 9, 1988.

[31] V. Lehtinen, M. Renfors. Truncation noise analysis of noise shaping DSP systems with application to CIC decimators. *Proceedings of the European Signal Processing Conference*, 2002.
[32] S. K. Mitra. *Digital Signal Processing*, McGraw-Hill, 1998.
[33] J. von Neumann, H. H. Goldstine. Numerical inverting of matrices of high order. *Bulletin of the American Mathematics Society* 53, 1947.
[34] E. Özer, A. Nisbet, D. Gregg. Stochastic bit-width approximation using extreme value theory for customizable processors. *Proceedings of the International Conference on Compiler Construction*, 2004.
[35] A. V. Oppenheim, R. W. Schafer. *Digital Signal Processing*, Prentice-Hall, 1975.
[36] A. V. Oppenheim, C. J. Weinstein. Effects of finite register length in digital filtering and the fast fourier transform. *IEEE Proceedings* 60(8), 1972.
[37] W. Sung, K. Kum. Simulation-based wordlength optimization method for fixed-point digital signal processing systems. *IEEE Transactions on Signal Processing* 43(12), December 1995.
[38] A. S. Sedra, K. C. Smith. *Microelectronic Circuits*, Saunders, 1991.
[39] A. Turing. Rounding-off errors in matrix processes. *Quarterly Journal of Mechanics* 1, 1948.
[40] S. A. Wadekar, A. C. Parker. Accuracy sensitive wordlength selection for algorithm optimization. *Proceedings of the International Conference on Computer Design*, October 1998.

CHAPTER 24

DISTRIBUTED ARITHMETIC

Rajeevan Amirtharajah
Department of Electrical and Computer Engineering
University of California–Davis

Distributed arithmetic (DA) [1, 2] is a computation algorithm that performs multiplication using precomputed lookup tables (LUTs) instead of logic. It is well suited to implementation on homogeneous field-programmable gate arrays (FPGAs) because of its high utilization of the available LUTs. It may also have advantages for modern heterogeneous FPGAs that contain built-in multipliers because it is area efficient for implementing long digital filters. DA targets the sum-of-products (or vector dot product) operation, and many digital signal processing (DSP) tasks such as filter implementation, matrix multiplication, and frequency transformation can be reduced to one or more sum-of-products computations.

24.1 THEORY

The theory behind DA is based on reorganizing the vector dot product operation around the binary representation of the vector elements [2]. Suppose that X is the vector of input samples and A is a constant vector of filter coefficients, corresponding to the taps of a finite impulse response (FIR) filter. Vectors X and A each consist of M elements X_k and A_k. The dot product y of X and A (corresponding to the convolution of X with the FIR impulse response) can be written as

$$y = \sum_{k=0}^{M-1} A_k X_k \tag{24.1}$$

We can represent each element of the input sample vector X in N-bit 2's complement notation. Then equation 24.1 can be expressed as

$$y = \sum_{k=0}^{M-1} A_k \left[-b_{k(N-1)} 2^{N-1} + \sum_{n=0}^{N-2} b_{kn} 2^n \right] \tag{24.2}$$

where $b_{k(N-1)}$ is the sign bit of the input sample X_k in N-bit 2's complement notation, and b_{kn} is the nth bit of input sample X_k. The possible values of b_{ki}

are either 0 or 1. Equation 24.2 can be further rearranged into equation 24.3 by multiplying out the factors and changing the order of the summation:

$$y = -\sum_{k=0}^{M-1} A_k b_{k(N-1)} 2^{N-1} + \sum_{n=0}^{N-2} \left[\sum_{k=0}^{M-1} A_k b_{kn} \right] 2^n = Z_{sign} + Z_{n1} \tag{24.3}$$

Consider each term in the brackets of the second summation in equation 24.3, labeled Z_{n0} in the following:

$$Z_{n0} = \sum_{k=0}^{M-1} A_k b_{kn} \tag{24.4}$$

where term Z_{n0} has 2^M possible values because b_{kn} is either 1 or 0. Therefore, each summation term $A_k b_{kn}$ can have the value of either A_k or 0. Instead of using a multiplier to compute any of these 2^M possible values whenever necessary, we can precompute them and store them in a LUT with depth 2^M. The contents of the LUT are then addressed directly by the bit-serial input data, $[b_{0n}, b_{1n}, b_{2n}, \ldots b_{Mn}]$, corresponding to the nth bits of each element X_k of input vector X. Multiplication by the factor 2^n in equation 24.3 can be realized by a shifter and the addressed LUT contents shifted and accumulated to form term Z_{n1} in $(N-1)$ cycles.

The sign term Z_{sign} can be handled in the same way with additional circuitry to implement subtraction; it takes one additional clock cycle. The final result y is formed after N cycles. Note that, if the filter length is greater than the bit width of the input data (i.e., $M > N$), DA computes the final result in fewer cycles than an implementation using a single multiply–accumulate functional unit. However, because the size of the LUT grows exponentially in the number of vector elements (2^M), most practical implementations use multiple LUTs and adders to combine partial dot products into the final result.

24.2 DA IMPLEMENTATION

A simple DA implementation is shown in Figure 24.1. It requires a 16-bit shift register for the input vector, a 16-entry LUT, an adder/subtractor, and an accumulator (`Result`) for the output. The `x2` operation is handled purely by wiring. This unit is a direct implementation of the DA algorithm described in the preceding section, and it is capable of computing the dot product of a 4-element vector X and a constant 4-element vector A.

In the figure the four 4-bit-wide elements of X are fed into the address decoder in most significant bit (MSB) first order to select the appropriate LUT row contents. The selected content is added with the left-shifted version of the previous `RESULT` value to form the current `RESULT` value. `Ts` is the sign bit timing signal that controls the add/subtract operation; when `Ts` is high, the current LUT content is subtracted from the left-shifted version of the previous result. The final vector dot product is obtained in four cycles. Shifting in the bit vector

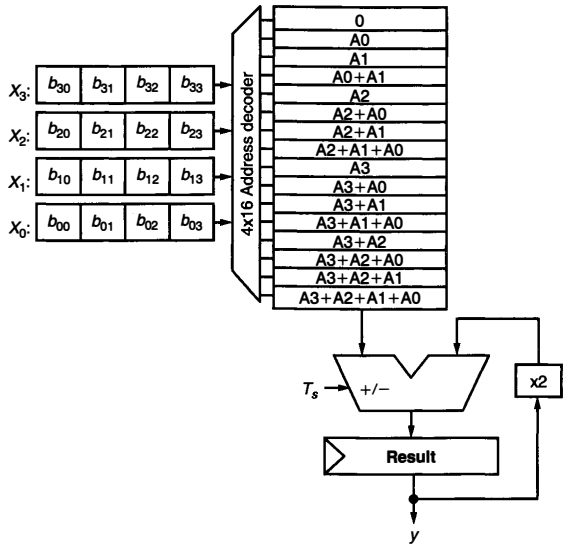


FIGURE 24.1 ■ A simple implementation of distributed arithmetic.

least significant bit (LSB) first also produces the correct final value and has the advantage of eliminating long carry propagations when accumulating the intermediate results.

The only modifications to Figure 24.1 required for this alternative are to reverse the bits of vector X_{in}, the shift register, and replace the left shift by 1 bit and the right shift by 1 bit. Various other modifications to this structure are possible. For example, the input sample shift register can be serial in/serial out or parallel in/serial out depending on the application.

LUT size can be a determining factor in the total hardware cost of a DA implementation. It is possible to modify the structure in Figure 24.1 to reduce the table size by a factor of 2. To achieve this reduction, consider a different representation of the input data samples X_k:

$$X_k = \frac{1}{2}[X_k - (-X_k)] \tag{24.5}$$

The 2's complement representation of the negative of X_k can be expressed as

$$-X_k = -\bar{b}_{k(N-1)}2^{N-1} + \sum_{n=0}^{N-2} \bar{b}_{kn}2^n + 1 \tag{24.6}$$

where each bit of X_k has been complemented and a 1 has been added to the complemented bits. Plugging equation 24.6 into equation 24.5 yields

$$X_k = \frac{1}{2}\left[-\left(b_{k(N-1)} - \bar{b}_{k(N-1)}\right)2^{N-1} + \sum_{n=0}^{N-2}\left(b_{kn} - \bar{b}_{kn}\right)2^n - 1\right] \tag{24.7}$$

Each difference term $\left(b_{kn} - \bar{b}_{kn}\right)$(for $n = 0$ to $N-1$) in equation 24.7 can take on values of +1 or −1. This alternate representation for X_k is convenient because, in the resulting summation for the dot product, each linear combination of A_k has a corresponding negative linear combination. Only one of these combinations needs to be stored in the LUT, with the negative being applied during operation using the subtractor. Substituting equation 24.7 into equation 24.1 and rearranging terms yields the following new expression for the result of the dot product y:

$$y = \sum_{n=0}^{N-1} Q(b_n) + Q(0) \tag{24.8}$$

where

$$Q(b_n) = \frac{1}{2}\sum_{k=0}^{M-1} A_k\left(b_{kn} - \bar{b}_{kn}\right)2^n, \quad n \neq N-1 \tag{24.9a}$$

$$Q(b_{N-1}) = -\frac{1}{2}\sum_{k=0}^{M-1} A_k\left(b_{k(N-1)} - \bar{b}_{k(N-1)}\right)2^{N-1}, \quad n = N-1 \tag{24.9b}$$

$$Q(0) = -\frac{1}{2}\sum_{k=0}^{M-1} A_k \tag{24.9c}$$

Note that the expressions for $Q(b_n)$ and $Q(b_{N-1})$ have 2^{M-1} possible magnitudes, with signs determined by the input bits, and that the computation of y requires an additional register to hold the constant term $Q(0)$. This leads to the reduced DA memory implementation shown in Figure 24.2, where the exclusive-or (XOR) gates are required to recode the addresses to access the appropriate LUT row and to control the timing of the sign bit into the adder/subtractor. The XOR gates, the initial condition register for $Q(0)$, and a 2-input multiplexer are the only additional hardware required to reduce the memory size by a factor of 2.

The implementations in both Figures 24.1 and 24.2 require N clock cycles to compute the final result, although additional cycles may be needed to match the throughput of the DA unit to other functional units in the system for a particular application. In Section 24.3 we will discuss mapping these basic structures onto FPGA fabrics. We will address the issue of performance improvement (by reducing the number of required clock cycles and increasing the clock frequency) in Section 24.4.

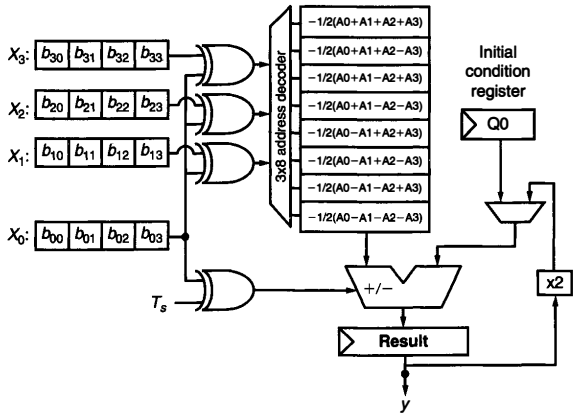


FIGURE 24.2 ■ Reduced DA memory implementation.

24.3 MAPPING DA ONTO FPGAs

Consider mapping a 16-tap FIR filter ($M = 16$) operating on 16-bit data ($N = 16$) onto an FPGA fabric based on 4-input LUTs. As discussed earlier, DA's primary drawback is that the size of the LUTs grows exponentially in the number of filter coefficients (or filter taps). If we want to use 16-bit data to represent the precomputed values, we need $16 \times 2^{16} = 1$ Mbit of memory. To limit this growth, long filters can be partitioned into several smaller DA units whose outputs are then combined using a tree of 2-input adders, as shown in Figure 24.3. This partitions the 16 filter taps A_0 to A_{15} among four DA units, each of which incorporates N 1-bit-wide 4-input LUTs.

The partitioning is chosen to correspond to the LUT size of the individual logic elements or CLBs. If the filter taps are symmetric (which they often are for typical signal-processing applications), the memory size can be reduced by a further factor of 2 by summing the appropriate elements of the input vector X_k using serial addition and using the bits of the resulting sum to address the LUTs. In addition to the serial adder hardware, this memory reduction comes at the expense of an additional clock cycle of latency before the final result is valid.

As CMOS technology has scaled and the complexity of individual CLBs has increased with succeeding FPGA generations, the hardware cost of implementing our example filter has shrunk dramatically. Based on an early implementation of an 8-tap, 8-bit filter using DA on a Xilinx 3042 FPGA [3], our example would consume approximately 120 CLBs, including control logic, even using the

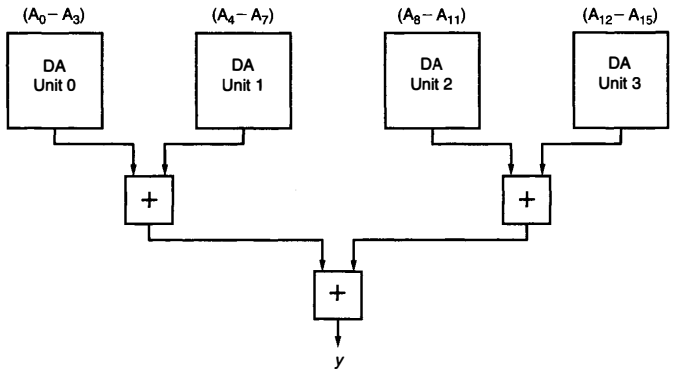


FIGURE 24.3 ■ A 16-tap FIR filter mapped onto multiple DA units.

symmetry of the filter coefficients to reduce the memory requirements. This would consume roughly the entire FPGA chip. Resource usage would be dominated by the input shift registers (60 CLBs) since this older FPGA architecture only allowed the local CLB flip-flops to be used in a shift configuration.

In contrast, a recent FPGA architecture encompasses four logic "slices" in each CLB, where two slices each roughly correspond to an entire CLB in the older architecture [6]. Because LUTs in Xilinx Spartan-3E FPGAs can be configured as 16×1 shift registers, the number of CLB resources to implement the data memory for DA is drastically reduced. Each logic slice also contains carry propagation logic for efficient implementation of adder circuits, which can be used to increase the speed of DA computation, as will be shown later. Implementing the example filter on a Spartan-3E FPGA requires approximately 113 slices, corresponding to 29 CLBs. This is under 12 percent of the total number of slices available in the smallest member of the 3S100E FPGA family.

Further enhancements to the architecture building blocks may allow for more efficient DA implementation in the future. For example, the potential of heterogeneous or coarse-grained FPGAs to support DA more efficiently by incorporating small adders and accumulators directly in the CLB is currently being explored [7].

24.4 IMPROVING DA PERFORMANCE

Two approaches can be taken to improve DA performance on an FPGA platform. First, the design can be modified to reduce the number of cycles required to compute the final result. Second, the cycle time can be decreased by reducing

the number of logic stages in the critical path of the computation. Examples of both approaches will be discussed in this section.

A simple approach to speeding up DA computation is to recognize that multiple bits from each input vector element X_k can be used to address multiple LUTs in each clock cycle (because addition is associative, we can perform the sum in equation 24.3 using any combination of partial sums that is convenient). This leads to an architecture like the one shown in Figure 24.4, which uses 2 bits of the input data vector elements at a time. The LUTs are identical because they contain the same linear combinations of filter coefficients A_k. The LUT outputs must be scaled by the correct exponent of 2 to maintain the significance of the bits added to the accumulated result (the x2 unit in Figure 24.4). Only two cycles are required to compute the result y for this implementation, instead of four cycles for the implementation in Figure 24.2. For longer bit-width input data, this idea can be extended to using more bits at a time.

The modification just described provides the benefit of a linear decrease in the number of clock cycles at the expense of a linear increase in LUT memory size. In addition, the number of inputs and the bit width of the adder/subtractor must increase. Mapping this approach onto an FPGA involves a trade-off between the routing resources consumed and the speed of the computation, as the input data bit vectors must be divided into subwords and distributed to multiple CLBs. In addition, multiple LUT outputs must be accumulated at a single destination to form the result, which consumes further routing.

Following a derivation similar to that presented by White [2], we can analyze this trade-off quantitatively. Suppose that we are implementing an M-tap filter

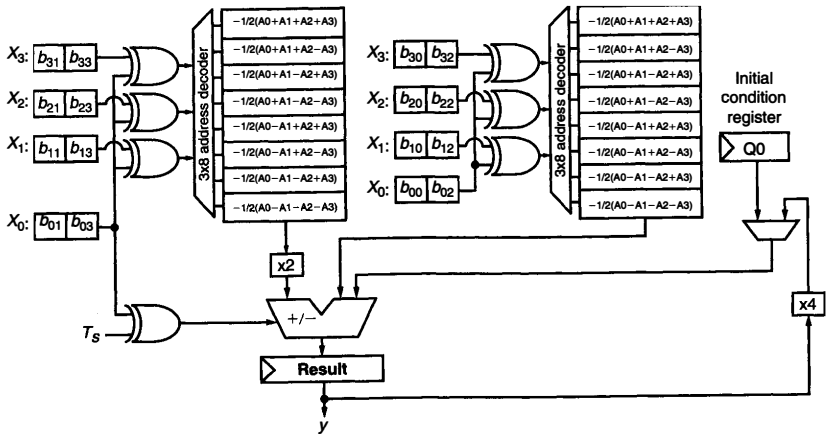


FIGURE 24.4 ■ Two-bit-at-a-time reduced memory DA implementation.

using an N-bit number representation and that the computation is proceeding L bits at a time. Further suppose that the LUT data is W bits wide. Computing the result requires that, in each cycle, MN bits are shifted in and WL bits are read out, and N/L clock cycles must pass. The number of wires N_W is therefore

$$N_W = \frac{MN}{N/L} + WL = (M+W)L \tag{24.10a}$$

If we define the relative importance of minimizing routing resources to minimizing latency as the ratio r, then

$$r = \frac{N/L}{N_W} = \frac{N}{(M+W)L^2} \tag{24.10b}$$

and we can find the L that satisfies our design criterion of relative importance r:

$$L = \left\lceil \sqrt{\frac{N}{r(M+W)}} \right\rceil \tag{24.10c}$$

Now suppose that an application demands low latency and that routing resources are not too tightly constrained; then, for $r = 2$, 32-bit input data ($N = 32$), a 4-tap FIR filter ($M = 4$), and 4-bit LUT data ($W = 4$); this yields $L = 2$. The desired DA implementation takes the input data 2 bits at a time to address the LUTs, completing a dot product computation in 16 cycles.

In addition to exploiting parallelism to speed up the DA computation, it is possible to employ various levels of pipelining. As we saw in Figure 24.1, the critical path involves decoding the address presented by the data shift registers, accessing the row from the LUT, and propagating the carry through the adder/subtractor while meeting the setup time constraints for the accumulator. If the implementation spans multiple CLBs, there is a potentially significant interconnect delay in this critical path in addition to the combinational logic delay. An obvious way to pipeline the simple implementation is to make the LUT synchronous and latch the outputs before they are fed to the adder/subtractor.

An alternative approach is to use carry save addition to reduce the carry propagation chain in the critical path [8]. The key modification to Figure 24.1 is to use a different structure for the adder/subtractor and to perform the computation in LSB first order. Instead of using a carry propagate adder to accumulate the entire result in one clock cycle, the adder/subtractor is pipelined at the bit level and the sum and carry outputs are stored in flip-flops at each cycle. Each full adder takes one input bit from the LUT output and one from the sum output of the next most significant full adder, automatically accounting for the `x2` scaling required in Figure 24.1. Assuming that the accumulator is wider than N bits, after N clock cycles the least significant N bits of the final result are stored in the LSBs of the accumulator while the remaining MSBs require one more carry propagating addition to produce the final result. This operation adds one extra clock cycle to the latency of the DA computation.

Most modern FPGA fabrics have dedicated paths for high-speed carry propagation. Given that most DA designs require accumulators with not too many more than N bits, the final carry propagation is typically not the critical path for the entire computation. The throughput is determined by the speed of the carry save addition in the accumulator.

Although using carry save addition at the single-bit level results in the greatest speed improvement, it is also the most resource intensive in terms of logic slices and CLBs. A speed versus area trade-off can be achieved by partitioning the adder/subtractor into multiple subcircuits, each of which propagates a carry across p bits ($p = 1$ in the example just described). Speedup factors of at least 1.5 have been observed over the traditional design shown in Figure 24.1 [8].

24.5 AN APPLICATION OF DA ON AN FPGA

In addition to FIR filters, a common DA application on FPGAs is acceleration of frequency transformations such as the discrete cosine transform (DCT), which is a critical component of the MPEG video compression and JPEG image compression standards. The two-dimensional DCT can be implemented as two one-dimensional DCTs and a matrix transposition. Each DCT can be implemented as a matrix–vector multiplication, which is easy to implement on an FPGA using DA because it can be decomposed into a set of vector dot products.

In one example, using DA instead of multiply–accumulate for the DCT resulted in a factor of 2.4 reduction in area for the FPGA implementation (on a Xilinx XC6200 FPGA) [9]. Using DA and pipelining of the routing to improve the algorithm performance, this implementation was fast enough to process VGA resolution images (640 × 480 pixels) at 25 frames per second—approximately four times faster than a full software implementation running on a microprocessor. The entire two-dimensional DCT consumed a 64 × 78 array of logic blocks on the chip (about 30 percent of the total FPGA area) and the DA portions of the DCT consumed 3648 logic blocks, or about 70 percent of the two-dimensional DCT total. The average utilization of each logic block for the DA components was 61 percent. This high level of utilization was a result of careful floorplanning in addition to DA's inherent suitability to FPGA implementation.

References

[1] Xilinx, Inc. *The Role of Distributed Arithmetic in FPGA-based Signal Processing*, Xilinx, Inc. (*http://www.xilinx.com/appnotes/theory1.pdf*), January 2006.

[2] S. A. White. Applications of distributed arithmetic to digital signal processing: A tutorial review. IEEE *ASSP Magazine* 6(3), July 1989.

[3] L. Mintzer. FIR filters with field-programmable gate arrays. *Journal of VLSI Signal Processing* 6, 1993.

[4] G. Roslin. A guide to using field-programmable gate arrays (FPGAs) for application-specific digital signal processing performance. Xilinx white paper, 1995.

[5] W. Wolf. *FPGA-based System Design* (Modern Semiconductor Design Series), Prentice-Hall, 2004.
[6] Xilinx, Inc. *Spartan-3E FPGA Family: Complete Data Sheet*, DS312 (v2.0) (*http://www.xilinx.com*), November 2005.
[7] B. Calhoun, F. Honore, A. Chandrakasan. A leakage reduction methodology for distributed MTCMOS. *IEEE Journal of Solid-State Circuits* 39(5), May 2004.
[8] R. Grover, W. Shang, Q. Li. A faster distributed arithmetic architecture for FPGAs. *Proceedings of the 10th ACM International Symposium on Field-Programmable Gate Arrays*, February 2002.
[9] R. Woods, D. Trainor, J.-P. Heron. Applying an XC6200 to real-time image processing. *IEEE Design & Test of Computers* 15(1), January/March 1998.

CHAPTER 25

CORDIC ARCHITECTURES FOR FPGA COMPUTING

Chris Dick
Advanced Systems Technology Group
DSP Division of Xilinx, Inc.

Because field-programmable gate arrays (FPGAs) are often used for realizing complex mathematical calculations, the FPGA designer is in need of a set of math libraries to support such implementations. The literature is rich with algorithmic options for evaluating the type of math functions (e.g., sine, cosine, sinh, cosh, arctangent, atan2, logarithms) that are typically found in a math library for general-purpose and DSP processors. The enormous flexibility of the FPGA coupled with the vast suite of algorithmic options for computing math functions can make the development of an FPGA math library a challenging task.

Common approaches to evaluating math functions include polynomial approximation-based techniques [13] and Newton-style iterations [13], to name a couple. One of the most useful and flexible approaches available to the hardware designer for developing high-performance computing hardware is the CORDIC (COordinate Rotation DIgital Computer) algorithm.

CORDIC is unparalleled in its ability to encapsulate a diversity of math functions in one basic set of iterations. It can be viewed as the Swiss Army Knife, so to speak, of arithmetic—that is, a single hardware architecture, with very minimal control overhead, having the ability to compute sine, cosine, cosh, sinh, atan2, square root, and polar-to-rectangular and rectangular-to-polar conversions, to name only a few functions.

It is in coordinate transformations that the algorithm comes into its own. In both, multi-operand input and multi-element output vectors are involved. There are a plethora of alternatives for realizing, say, division in an FPGA, and most of the CORDIC alternatives provide good hardware efficiency. However, the algorithm remains unrivaled when it comes to processing multi-element I/O vectors, as is the case when converting from Cartesian to polar coordinates or vice versa. CORDIC falls into the class of shift-and-add algorithms—it is a multiplierless method dominated by additions. FPGAs are very efficient at realizing arbitrary precision adders, and so the CORDIC algorithm is in many ways a natural fit for course-grained FPGA architectures such as the Xilinx Virtex-4 family of devices [41].

This chapter begins with a brief tutorial overview of the CORDIC algorithm. Because most hardware realizations of CORDIC employ fixed-point arithmetic,

design considerations for quantizing the datapath and selecting a suitable number of iterations are provided. Approaches for architecting FPGA CORDIC processors are then presented. Various options are discussed that highlight the use of FPGA features such as embedded multipliers, embedded multiply–accumulator (MACC) tiles, and logic fabric to deliver hardware realizations that provide various trade-offs between throughput, latency, logic fabric utilization, and numerical accuracy. A brief overview of the System Generator [38] design flow used to produce our implementations is also provided. Design considerations for producing very high throughput (450–500 MHz) implementations in Virtex-4 [41] devices are presented as well.

25.1 CORDIC ALGORITHM

The CORDIC algorithm was first published by Volder [35] in 1959 as a technique for efficiently implementing the trigonometric functions required for real-time aircraft navigation. Since first being published, the method has been extensively analyzed and extended to the point where a very rich set of functions is accessible from the one basic set of equations. The algorithm is dominated by bit shifts and additions and so was an ideal match for early-generation computing technology in which multiplication and division were expensive in terms of computation time and physical resources. Volder essentially presented iterative techniques for performing translations between Cartesian and polar coordinate systems (*vectoring mode*), and a method for realizing a plane rotation (*rotation mode*) using a series of arithmetic shifts and adds.

Since its publication, the CORDIC algorithm has been applied to many different applications and has been used as the cornerstone of the arithmetic engine in many VLSI signal-processing implementations [34]. It has been used extensively for computing various types of transforms, including the fast Fourier transform (FFT) [10,11], the discrete cosine transform [4], and the discrete Hartley transform [3]. And it has found widespread use in realizing various classes of digital filters, including Kalman filters [31], adaptive lattice structures [21], and adaptive nulling [30]. A large body of work has been published on CORDIC-based approaches for implementing various types of linear algebra operations, including singular value decomposition (SVD) [1], Given's rotations [30], and QRD-RLS (recursive least squares) filtering [14].

A brief tutorial style treatment of the basic algorithm is provided here; its FPGA implementation will be discussed in subsequent sections.

25.1.1 Rotation Mode

The CORDIC algorithm has two basic modes: vectoring and rotation. These can be applied in several coordinate systems, including circular, hyperbolic, and linear, to compute various functions such as atan2, sine, cosine, and even division. We begin our treatment by considering the problem of constructing an efficient method to realize a plane rotation of the vector (x_s, y_s) through an angle θ to produce a vector (x_f, y_f), as shown in Figure 25.1.

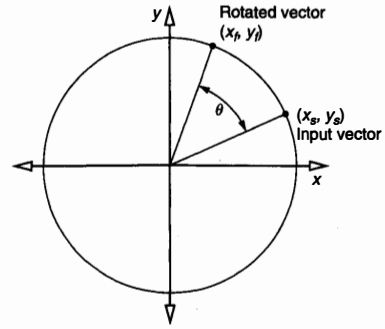


FIGURE 25.1 ■ Plane rotation of the vector (x_s, y_s) through an angle θ.

The rotation is formally captured in matrix form by equation 25.1.

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_s \\ y_s \end{bmatrix} = ROT(\theta) \begin{bmatrix} x_s \\ y_s \end{bmatrix} \tag{25.1}$$

which can be expanded to the set of equations in equation 25.2.

$$\begin{aligned} x_f &= x_s \cos\theta - y_s \sin\theta \\ y_f &= x_s \sin\theta + y_s \cos\theta \end{aligned} \tag{25.2}$$

The development of a simplified approach for producing rotation through the angle θ begins by considering it not as one lumped operation but as the result of a series of smaller rotations, or *micro-rotations*, through the set of angles α_i where

$$\theta = \sum_{i=0}^{\infty} \alpha_i \tag{25.3}$$

The rotation can now be cast as a product of smaller rotations, or

$$ROT(\theta) = \prod_{i}^{\infty} ROT(\alpha_i) \tag{25.4}$$

If these values α_i are carefully chosen, we can provide a very efficient computation structure. Equation 25.2 can be modified to reflect a micro-rotation $ROT(\alpha_i)$, leading to equation 25.5.

$$\begin{aligned} x_{i+1} &= x_i \cos\alpha_i - y_i \sin\alpha_i \\ y_{i+1} &= x_i \sin\alpha_i + y_i \cos\alpha_i \end{aligned} \tag{25.5}$$

where $(x_0, y_0) = (x_s, y_s)$. Factoring permits the equations to be expressed as

$$\begin{aligned} x_{i+1} &= \cos\alpha_i (x_i - y_i \tan\alpha_i) \\ y_{i+1} &= \cos\alpha_i (y_i + x_i \tan\alpha_i) \end{aligned} \tag{25.6}$$

which positions the iterative update as the product of two procedures: a scaling by the $\cos\alpha_i$ term and a similarity transformation, or scaled rotation.

The next significant step that leads to an algorithm that lends itself to an efficient hardware realization is to place restrictions on the values that α_i can take. If

$$\alpha_i = \tan^{-1}\left(\sigma_i 2^{-i}\right) \tag{25.7}$$

where $\sigma_i \in \{-1, +1\}$, then equation 25.6 can be written as

$$\begin{aligned} x_{i+1} &= \cos\alpha_i \left(x_i - \sigma_i y_i 2^{-i}\right) \\ y_{i+1} &= \cos\alpha_i \left(y_i + \sigma_i x_i 2^{-i}\right) \end{aligned} \tag{25.8}$$

The purpose of σ_i will be explained shortly.

With the exception of the scaling term, these equations can be implemented using only additions, subtractions, and shifts. In the set of equations that are typically presented as the CORDIC iterations, and following the lead of Volder [35], the scaling term is usually excluded from the defining equations to produce the modified set of equations

$$\begin{aligned} x_{i+1} &= x_i - \sigma_i y_i 2^{-i} \\ y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \end{aligned} \tag{25.9}$$

To determine the value of these σ_i we introduce a new variable, z (the *angle* variable). The recurrence on z is defined by equation 25.10.

$$z_{i+1} = z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) \tag{25.10}$$

If the z variable is initialized with the desired angle of rotation θ—that is, z_0—it can be driven to 0 by conditionally adding or subtracting terms of the form $\tan^{-1}\left(2^{-i}\right)$ from the state variable z. The conditioning is captured by the term σ_i as a test on the sign of the current state of the angle variable z_i—that is,

$$\sigma_i = \begin{cases} 1 & \text{if } z_i \geq 0 \\ -1 & \text{if } z_i < 0 \end{cases} \tag{25.11}$$

Driving z to 0 is actually an iterative process for decomposing θ into a weighted linear combination of terms of the form $\tan^{-1}\left(2^{-i}\right)$. As z goes to 0, the vector (x_0, y_0) experiences a sequence of micro-rotation extensions that in the limit $n \to \infty$ converge to the coordinates (x_f, y_f).

The complete algorithm is summarized in equation 25.12.

$$\begin{aligned} i &= 0 \\ x_0 &= x_s \\ y_0 &= y_s \\ z_0 &= \theta \\ x_{i+1} &= x_i - \sigma_i y_i 2^{-i} \\ y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\ z_{i+1} &= z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) \\ \sigma_i &= \begin{cases} 1 & \text{if } z_i \geq 0 \\ -1 & \text{if } z_i < 0 \end{cases} \end{aligned} \tag{25.12}$$

which is easily realized in hardware because of the simple nature of the arithmetic required. The only complex function is the $\tan^{-1}$, which can be precomputed and stored in a memory.

Because of the manner in which the updates are directed, this mode of the CORDIC algorithm is sometimes referred to as the *z-reduction mode.* Figure 25.2 shows the signal flow graph for the algorithm. Observe the butterfly-style architecture in the cross-addition update.

25.1.2 Scaling Considerations

Because the scaling term $\cos\alpha_i$ has not been carried over into equation 25.12, the input vector (x_0, y_0) not only undergoes a rotation but also experiences scaling or growth by a factor $1/\cos\alpha_i$ at each iteration. That is,

$$\begin{aligned} R_{i+1} &= K_{c,i} R_i = \frac{1}{\cos\alpha_i} R_i = \left(1 + \sigma_i^2 2^{-2i}\right)^{1/2} R_i \\ &= \left(1 + 2^{-2i}\right)^{1/2} R_i \end{aligned} \tag{25.13}$$

where $R_i = |x_i + jy_i|$ designates the modulus of the vector at iteration i, and the subscript c associates the scaling constant with the *circular* coordinate system.

Figure 25.3 illustrates the growth process at each of the intermediate CORDIC iterations as (x_0, y_0), which is translated to its final location (x_f, y_f). For an infinite number of iterations the scaling factor is

$$K_c = \prod_{i=0}^{\infty} \left(1 + 2^{-2i}\right)^{1/2} \approx 1.6468 \tag{25.14}$$

It should also be noted that, since $\sigma_i \in \{-1, +1\}$, the scaling term is a constant that is independent of the angle of rotation.

As captured by equation 25.4, the angle of rotation θ is decomposed into an infinite number of elemental angles α_i, which implies that an infinite number of iterations is theoretically required. In practice, a finite number of iterations, n, is selected to make the system realizable in software or hardware. Application of n iterations translates (x_0, y_0) to (x_n, y_n) rather than to (x_f, y_f)

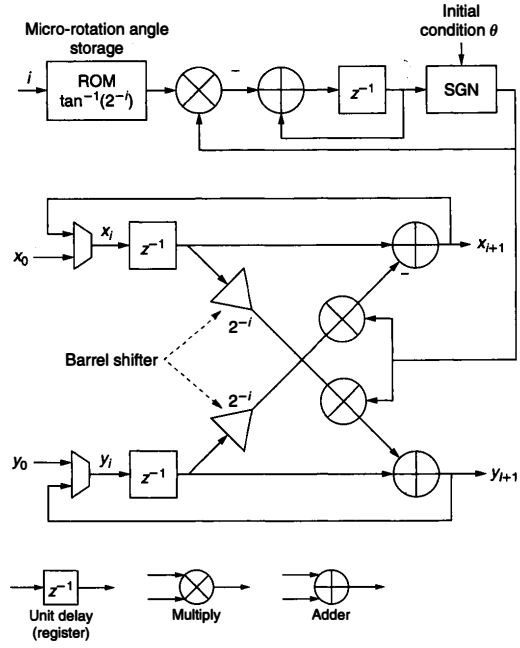


FIGURE 25.2 ■ A signal flow graph for CORDIC vector rotation.

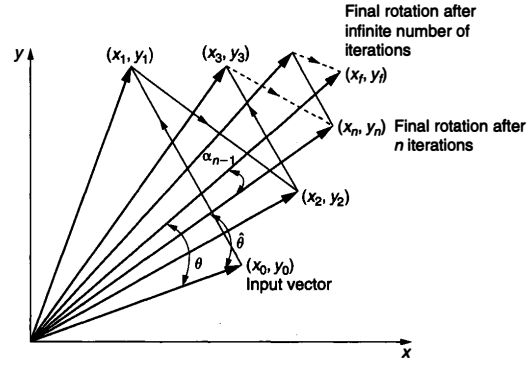


FIGURE 25.3 ■ Each iteration of a CORDIC rotation introduces vector growth by a factor of $\frac{1}{\cos\alpha_i} = \left(1+\sigma_i^2 2^{-2i}\right)^{1/2}$.

as shown in Figure 25.3. The rotation error $|\arg(x_f+jy_f)-\arg(x_n+jy_n)|$ has an upper bound of α_{n-1}, which is the smallest term in the weighted linear expansion of θ.

For an *infinite-precision* arithmetic implementation of the system of equations, each iteration contributes one additional effective fractional bit to the result. Most hardware implementations of the CORDIC algorithm are realized using fixed-point arithmetic, and, as will be discussed soon, the relationship between the number of effective output binary result digits is very different from that of a floating-point realization of the algorithm.

25.1.3 Vectoring Mode

The CORDIC vectoring mode is most commonly used for implementing a conversion from a rectangular to a polar coordinate system. In contrast to rotation mode, where Z is driven to 0, in the vectoring mode the initial vector (x_0, y_0) is rotated until the y component is driven to 0. The modification to the basic algorithm required to accomplish this goal is to direct the iterations using the sign of y_i. As the y variable is reduced, the corresponding angle of rotation is accumulated in the z register. The complete vectoring algorithm is captured by equation 25.15.

$$\begin{aligned} i &= 0 \\ x_0 &= x_s \\ y_0 &= y_s \\ z_0 &= 0 \\ x_{i+1} &= x_i - \sigma_i y_i 2^{-i} \\ y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\ z_{i+1} &= z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) \\ \sigma_i &= \begin{cases} 1 & \text{if } y_i < 0 \\ -1 & \text{if } y_i \geq 0 \end{cases} \end{aligned} \tag{25.15}$$

This CORDIC mode is commonly referred to as *y-reduction* mode.

Figure 25.4 shows the results of a CORDIC vector mode simulation for $\arg(x_s+jy_s) = 7\pi/8$ and $|x_s+jy_s| = 1$. The top plot (a) shows the true angle of the input vector (solid line) overlaid with $\arg(x_i+jy_i), i = 1,\ldots,16$. We note the oscillatory behavior of (x_i, y_i) about the true value of the angle. Overdamped or underdamped behavior will be produced depending on the system initial conditions. The lower plot (b) shows, for this case of initial conditions, how rapidly the algorithm can converge toward the correct solution. In fact, for many practical applications, a *short* CORDIC (small number of iterations) produces acceptable performance.

For example, in a 16-QAM (quadrature amplitude modulation) carrier recovery circuit [29] employing a Costas Loop [23], a 5-iteration CORDIC usually provides adequate performance [12].

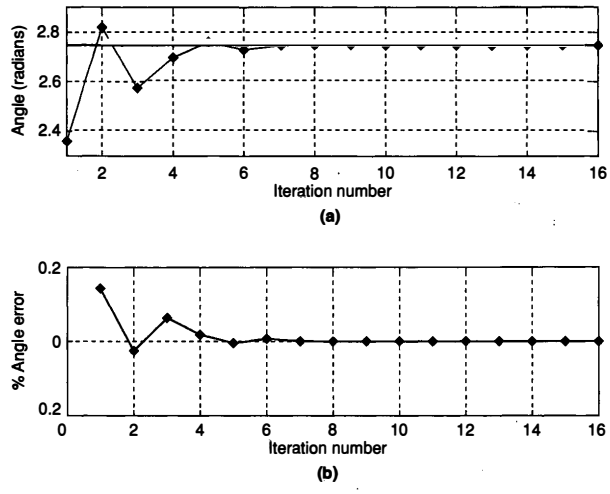


FIGURE 25.4 ■ Convergence of CORDIC vectoring. The top plot (a) shows the true angle of the input vector $\arg(x_s + jy_s)$ (solid line) overlaid with $\arg(x_i + jy_i)$, $i = 1,\ldots,16$. The bottom plot (b) is the percentage angle error as a function of the iteration number.

25.1.4 Multiple Coordinate Systems and a Unified Description

Alternative versions of the CORDIC engine can be defined under the circular, hyperbolic, and linear coordinate systems [13]. These use a computation similar to that of the basic CORDIC algorithm, but can provide additional functions. It is possible to capture the vectoring and rotation modes of the CORDIC algorithm in all three coordinate systems using a single set of unified equations. To do this a new variable, m, is introduced to identify the coordinate system so that

$$m = \begin{cases} +1 & \text{circular coordinates} \\ 0 & \text{linear coordinates} \\ -1 & \text{hyperbolic coordinates} \end{cases} \tag{25.16}$$

The unified micro-rotation is

$$\begin{aligned} x_{i+1} &= x_i - m\sigma_i y_i 2^{-i} \\ y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\ z_{i+1} &= \begin{cases} z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) & \text{if } m = 1 \\ z_i - \sigma_i \tan\mathrm{h}^{-1}\left(2^{-i}\right) & \text{if } m = -1 \\ z_i - \sigma_i \left(2^{-i}\right) & \text{if } m = 0 \end{cases} \end{aligned} \tag{25.17}$$

The scaling factor is $K_{m,i} = \left(1 + m2^{-2i}\right)^{1/2}$.

TABLE 25.1 ■ Functions computed by a CORDIC processor for the circular ($m = 1$), hyperbolic ($m = -1$), and linear ($m = 0$) coordinate systems

Coordinate system	Rotation/vectoring	Initialization	Result vector
1	Rotation	$x_0 = x_s$	$x_n = K_{1,n} \cdot (x_s \cos\theta - y_s \sin\theta)$
		$y_0 = y_s$	$y_n = K_{1,n} \cdot (y_s \cos\theta + x_s \sin\theta)$
		$z_0 = \theta$	$z_n = 0$
		$x_0 = 1/K_{1,n}$	$x_n = \cos\theta$
		$y_0 = 0$	$y_n = \sin\theta$
		$z_0 = \theta$	$z_n = 0$
1	Vectoring	$x_0 = x_s$	$x_n = K_{1,n} \cdot \text{sgn}(x_0) \cdot (\sqrt{x^2 + y^2})$
		$y_0 = y_s$	$y_n = 0$
		$z_0 = \theta$	$z_n = \theta + \tan^{-1}(y_s/x_s)$
0	Rotation	$x_0 = x_s$	$x_n = x_s$
		$y_0 = y_s$	$y_n = y_s + x_s y_s$
		$z_0 = z_s$	$z_n = 0$
0	Vectoring	$x_0 = x_s$	$x_n = x_s$
		$y_0 = y_s$	$y_n = 0$
		$z_0 = z_s$	$z_n = z_s + y_s/x_s$
-1	Rotation	$x_0 = x_s$	$x_n = K_{-1,n} \cdot (x_s \cosh\theta + y_s \sinh\theta)$
		$y_0 = y_s$	$y_n = K_{-1,n} \cdot (y_s \cosh\theta + x_s \sinh\theta)$
		$z_0 = \theta$	$z_n = 0$
		$x_0 = 1/K_{-1,n}$	$x_n = \cosh\theta$
		$y_0 = 0$	$y_n = \sinh\theta$
		$z_0 = \theta$	$z_n = 0$
-1	Vectoring	$x_0 = x_s$	$x_n = K_{-1,n} \cdot \text{sgn}(x_0) \cdot (\sqrt{x^2 - y^2})$
		$y_0 = y_s$	$y_n = 0$
		$z_0 = \theta$	$z_n = \theta + \tanh^{-1}(y_s/x_s)$

TABLE 25.2 ■ CORDIC shift sequences, ranges of covergence, and scale factor bound for circular, linear, and hyperbolic coordinate systems

Coordinate system	Shift sequence	Convergence	Scale factor
m	$s_{m,i}$	θ_{MAX}	K_m $(n \rightarrow \infty)$
1	$0, 1, 2, 3, 4, \ldots, i, \ldots$	≈ 1.74	≈ 1.64676
0	$1, 2, 3, 4, 5, \ldots, i+1, \ldots$	1.0	1.0
-1	$1, 2, 3, 4, 4, 5, \ldots$*	≈ 1.13	≈ 0.83816

* For $m = -1$, the following iterations are repeated: $\{4, 13, 40, 121, \ldots, k, 3k+1, \ldots\}$.

Operating the two modes in the three coordinate systems, in combination with suitable initialization of the algorithm variables, generates a rich set of functions, shown in Table 25.1. Table 25.2 summarizes the shift sequences, maximum angle of convergence θ_{MAX} (elaborated on in a later section), and

scaling function for the three coordinate systems. Note that each system requires slightly different shift sequences (the sequence of i values).

25.1.5 Computational Accuracy

One of the first design requirements for the fixed-point arithmetic implementation of a CORDIC processor is to define the numerical precision requirements of the datapath. This includes defining the numeric representation for the input operands and the processing engine internal registers, in addition to the number of micro-rotations that will be required to achieve a specified numerical quality of result. To guide this process it is useful to have an appreciation for the sources of computation noise in CORDIC arithmetic. While CORDIC processing can be realized with floating-point arithmetic [2,7], we will restrict our discussion to fixed-point arithmetic implementations, as they are the most commonly used numeric type employed in FPGA realizations.

Two primary noise sources are to be considered. One is associated with the weighted and finite linear combination of elemental angles that are used to represent the desired angle of rotation θ; the second source is associated with the rounding of the datapath variables x, y, and z. These noise sources are referred to as the *angle approximation* and the *rounding error*, respectively.

Angle approximation error

In this discussion we assume that all finite-precision quantities are represented using fixed-point 2's complement arithmetic, so the value F of a normalized number u represented using m binary digits $(u_{m-1}u_{u-2}\ldots u_0)$ is

$$F = -u_{m-1} + \sum_{j=0}^{m-2} u_j \cdot 2^{-m+j+1} \tag{25.18}$$

As will be presented next, there is a requirement in the CORDIC algorithm to accommodate bit growth in both the integer and fractional fields of the x and y variables. To accommodate this, the data format is enhanced with an additional G_I and G_F integer and fractional guard bits, respectively, so that a number with $B_I + G_I$ and $B_F + G_F$ bits allocated to the integer and fractional fields s and r, respectively $\left(s_{B_I+G_I-1}s_{B_I+G_I-2}\ldots s_0 r_{B_F+G_F-1}r_{B_F+G_F-2}\ldots r_0\right)$, is expressed as

$$F = -r_{B_I+G_I-1} \cdot 2^{B_I+G_I-1} + \sum_{j=0}^{B_I+G_I-2} s_j \cdot 2^j + \sum_{j=0}^{B_F+G_F-1} r_j \cdot 2^{-(B_F+G_F)+j} \tag{25.19}$$

Figure 25.5 illustrates the extended data format. The integer guard bits are necessary to accommodate the vector growth experienced when operating in circular coordinates. The fractional guard bits are required to support the word growth that occurs in the fractional field of the x and y registers due to the successive arithmetic shift-right operations employed in the iterative updates. It is assumed that the input samples are represented as normalized $(1 \cdot B_F)$ quantities.

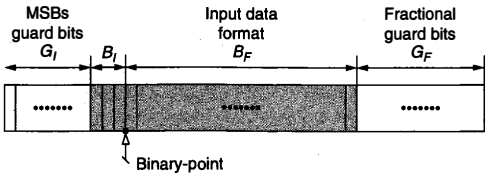


FIGURE 25.5 ■ The fractional fixed-point data format used for internal storage in the quantized CORDIC algorithm.

There are n fixed rotation angles $\alpha_{m,i}$ employed to approximate a desired angle of rotation θ. Neglecting all other error sources, the accuracy of the calculation is governed by the nth and final rotation, which limits the angle approximation error to $\alpha_{m,n-1}$. Because $\alpha_{m,n-1} = \frac{1}{\sqrt{m}}\tan^{-1}\left(\sqrt{m}\cdot 2^{-s_{m,n-1}}\right)$, the angle approximation error can be made arbitrarily small by increasing the number of micro-rotations n. Of course, the number of bits allocated to represent the elemental angles $\alpha_{m,i}$ needs to be sufficient to support the smallest angle $\alpha_{m,n-1}$. The number representation defined in equation 25.19 results in a least significant digit weighting of $2^{-(B_F+G_F)}$. Therefore, $\alpha_{m,n-1} \geq 2^{-(B_F+G_F)}$ must hold in order to represent $\alpha_{m,n-1}$. Approximately $n+1$ iterations are required to generate B_F significant fractional bits.

Datapath rounding error

As discussed earlier, most FPGA realizations of CORDIC processors employ fixed-point arithmetic. The update of the x, y, and z state variables according to equation 25.12 produces a dynamic range expansion, which is ideally supported by precisions that accommodate the worst-case bit growth. The number of additional guard bits beyond the original precision of the input operands can be very large, and carrying these additional bits in the datapath is generally impractical. For example, in the circular mode of operation the number of additional fractional bits required to support a full-precision calculation is determined by the sum of the shift sequence $s_{m,i}$.

If the input operands are presented as a 16.15 value (a 16-bit field width with 15 fractional bits) and 16 micro-rotations are performed, the bit growth for the fractional component of the datapath is $\sum_{i=0}^{15} i = 120$ bits. Thus, the total number of fractional bits required for a full-precision calculation is $120+15 = 135$. While FPGAs certainly provide the capability to support arbitrary precision arithmetic, it would be highly unusual to construct a CORDIC processor with such a wide datapath. In fact, the error in the CORDIC result vector can be maintained to a desired value using far few fractional guard bits, as discussed next.

Rather than by accommodating the bit growth implied in the algorithm, the dynamic range expansion is better handled by rounding the newly computed state variables. Control over wordlength can be achieved using unbiased rounding, simple truncation, or other techniques [26]. True rounding, while the

preferred approach because of the smaller error introduced when compared to truncation, can be the most area consuming because a second addition is potentially required. In some cases, the cost of rounding can be significantly reduced by exploiting the carry-in port of the adders used in the implementation. Truncation is obviously the simplest approach, requiring only the extraction of a bit field from the full-precision value, but it introduces an undesirable positive bias in the final result and an error component that is twice the magnitude of unbiased rounding. Nevertheless, truncation arithmetic is the option most frequently employed in FPGA CORDIC datapath design.

A simple approach to understanding the quantization effects of the CORDIC algorithm was first presented by Walther [36]. A very complete analysis was later published by Hu [16], with further work reported by Park and Cho [28] and Hu and Bass [17].

For many practical applications Walther's method produces acceptable results, and this is the approach we will use to design the FPGA implementations. A brief summary of the method is presented here.

Analysis of the rounding error for the z variable is straightforward because there are no data shifts involved in the state update, as there are with the x and y variables. The rounding error is simply due to the quantization of the rotation angles. The upper bound on the error is then the accumulation of the absolute values of the rounding errors for the quantized angles $\alpha_{m,i}$.

Datapath pruning and its associated quantization effects for the x and y variables is certainly a more challenging analysis than that for the angle variable because the scaling term involved in the cross-addition update. Nevertheless, several extensive treatments have been published. The effects of error propagation in the algorithm were reported by Hu in a Cray Research publication [5] and later extended by Hu and Bass [17]. Walther's treatment takes a slightly simplified approach and assumes that the maximum rounding error for n iterations is the sum of the absolute value of the maximum rounding error associated with each micro-rotation and the subsequent quantization that is performed to control word growth.

The format for the CORDIC variables was shown in Figure 25.5. $B = B_I + B_F + G_F + G_I$ bits are used to for internal storage, with $B_F + G_F$ of these bits assigned to the fractional component of the representation. The maximum error for one iteration is therefore of magnitude $2^{-(B_F+G_F)}$. In the simplified analysis, the rounding error $e(n)$ in the final result, and after all n iterations, is simply n times this quantity, which is $e(n) = n2^{-(B_F+G_F)}$. If B_F accurate fractional bits are required in the result word, the required resolution is $2^{-(B_F-1)}$. If B_F is selected such that $e(n) \leq 2^{-B_F}$, the datapath quantization can effectively be ignored. This implies that $n2^{-(B_F+G_F)} \leq 2^{-B_F}$, which requires $B_F \geq \log_2(n)$. Therefore, $G_F = \lceil \log_2(n) \rceil$ fractional guard bits are required to produce a result that has an accuracy of B_F fractional bits. This simplified treatment of the computation noise is a reasonable approximation that can help guide the definition of the datapath width required to meet a specified numerical fidelity.

Figure 25.6 shows the results of a simulation using different data representations for the x, y, and z variables of a CORDIC vectoring algorithm in circular

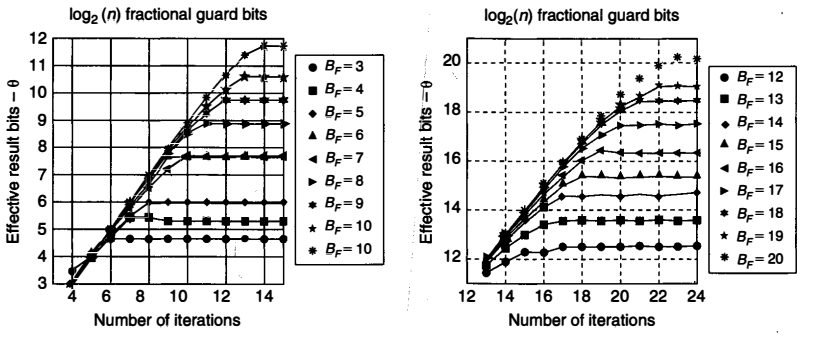


FIGURE 25.6 ■ The effiective number of result bites for a CORDIC vector processor (circular coordinates). The number of fractional guard bites is $G_F = \lceil log_2(n) \rceil$.

coordinates. Unit modulus complex vectors with random angles were generated and projected onto the CORDIC input sample (x_0, y_0). Each sample point in the plot represents the maximum absolute error of the angle estimate resulting from 4000 trials. We note that in all of the simulations the effective number of fractional output bits is matched to the number of fractional bits in the input operand.

The simplified treatment of the rounding noise generated in the update equations is certainly pessimistic and produces a requirement on the number of guard bits that is biased slightly higher than what might typically be required.

Selecting $G_F = \lceil \log_2(n) \rceil$ is certainly a safe, if not a slightly overengineered, choice. In the context of an FPGA realization, an additional bit of precision carried by the variables has almost negligible impact on the area and maximum operating clock frequency of the design.

An additional observation from the plots in Figure 25.6 is that the production of B_F effective output digits requires more iterations than the $B_F + 1$ iterations required for a full floating-point implementation—an additional three iterations are, in general, necessary. The implication of this is that two additional bits must be allocated to represent the elemental angles to provide the angle resolution implied by the adjusted iteration count.

Defining the number of guard bits G_I is very straightforward based on the number of integer bits B_I in the input operands, the coordinate system to be employed (e.g., circular, hyperbolic, or linear), and the mode (vectoring or rotation). For example, if the input data is in standard 2's complement format and bounded by ± 1, then $B_I = 1$. This means that the l^2 norm of the input (x_0, y_0) is $\sqrt{2}$. For the CORDIC vectoring mode, the range extension introduced by the iterations is approximately $K_1 \approx 1.6468$ for any reasonable number of iterations. The maximum that the final value of the x register can assume is approximately $\sqrt{2} \cdot 1.6468 \approx 2.3289$, which requires that $G_I = 2$.

TABLE 25.3 ■ Number of rotations and required CORDIC processor datapath format required to achieve a desired number of effective output bits

Number of effective fractional result bits	Micro-rotations: n	Internal storage data format: x and y	Internal storage data format: z
8	10	(15.12)	(15.14)
12	15	(19.16)	(19.18)
16	19	(24.21)	(24.23)
24	27	(32.29)	(32.31)

Based on this approach, a reasonable procedure for selecting the number of CORDIC micro-rotations and a suitable quantization for the x, y, and z variables, given the effective number of fractional bits required in the output, is the following:

1. Define the number of iterations as $n = B_F + 3$.
2. Select the field width for the x and y variables as $2 + B_I + B_F + \log_2(n)$ for the vectoring mode in circular coordinates—$B_F + \log_2(n)$ of these bits are of course allocated to the fractional component of the register.
3. Select the fractional precision of the angle register z to be $B_F + \log_2(n) + 2$, while maintaining 1 bit for the integer portion of the register.
4. Apply similar reasoning to select n and G_I for the other coordinate systems and modes.

Based on this approach, Table 25.3 shows the number of micro-rotations n and the internal data storage format corresponding to 8, 12, 16, 24, and 32 effective fractional result bits. The notation $(p \cdot q)$ indicates a bit field width of p bits, with q of these bits allocated to the fractional component of the value.

25.2 ARCHITECTURAL DESIGN

There are many hardware architecture options to evaluate when considering FPGA CORDIC datapath implementation. A particular choice is determined by the design specifications of numerical accuracy, throughput, and latency. At the highest level are key architectural decisions on whether a folded [27] or fully parallel [27] pipelined (or nonpipelined) architecture is to be used. At a lower, technology-specific level, FPGA features associated with a particular FPGA family are also a factor in the decision process. For example, later-generation FPGAs such as the Virtex-4 family [41] include an array of arithmetic units called the XtremeDSP Slice [43] (referred to as the DSP48 in the remainder of the chapter).

As discussed later, a CORDIC implementation can be realized that is mostly based on the DSP48 embedded tile. Thus, with this particular family of devices

the designer has a choice of producing an implementation that is completely logic slice based [40] or biased toward the use of DSP48 elements. The process that guides such decisions is elaborated in the next section.

25.3 FPGA IMPLEMENTATION OF CORDIC PROCESSORS

One of the elegant properties of FPGA computing is the ability to construct a compute engine closely tailored to the problem specifications, including processing throughput, latency, and numerical accuracy. Consider, for example, the throughput requirement. At one end of the architecture spectrum, and when modest processing rates are involved, a fully folded [27] implementation, where the same logic is used for all iterations (folding factor = n), is one option. In this case, new operands are delivered, and a new result vector is produced, every n clock cycles. This choice of implementation results in the smallest FPGA footprint at the expense of processing rate. If a high-throughput unit is required, a fully parallel, or completely unfolded implementation (folding factor = 1) that allocates a complete hardware PE to each iteration is appropriate. This will of course result in the largest area, but provides the highest compute rate.

25.3.1 Convergence

One of the design considerations for the CORDIC engine is the region of convergence that needs to be supported by the implementation, as the basic form of the algorithm does not converge for all input coordinates. For the rotation mode, the CORDIC algorithm converges provided that the absolute value of the rotation angle is no larger than $\theta_{MAX} \approx 1.7433$ radians, or approximately 99.88°.

In many applications we need to support input arguments that span all four quadrants of the complex plane—that is, a so-called *full-range* CORDIC. Much published work addresses this requirement [8, 19, 25], and many elegant extensions to the basic set of CORDIC iterations have been produced. Some of them introduce additional iterations and, while maintaining the basic shift-and-add property of the algorithm, result in a significant time or area penalty.

The most straightforward approach for handling the convergence issue in FPGA hardware is to first note that the natural range of convergence extends beyond the angle $\pi/2$. That is, the basic set of equations converges over the interval $[-\pi/2, \pi/2]$. To extend the implementation to converge over $[-\pi, \pi]$, we can simply detect when the input angle extends beyond the first quadrant, map that angle to either the first or fourth quadrants, and make a post-micro-rotation correction to account for the input angle mapping. This architecture is illustrated in Figure 25.7.

The input mapping is particularly simple. Referring to Figure 25.7, if x_0 is negative, the quadrants must be changed by applying $a \pm \pi/2$ ($\pm 90°$) rotation. Whether it is a positive or negative rotation is determined by the sign of y_0. To compensate for the input mapping, an angle rotation is conditionally applied to the micro-rotation engine result z'_n to produce the final output value z_n. Details

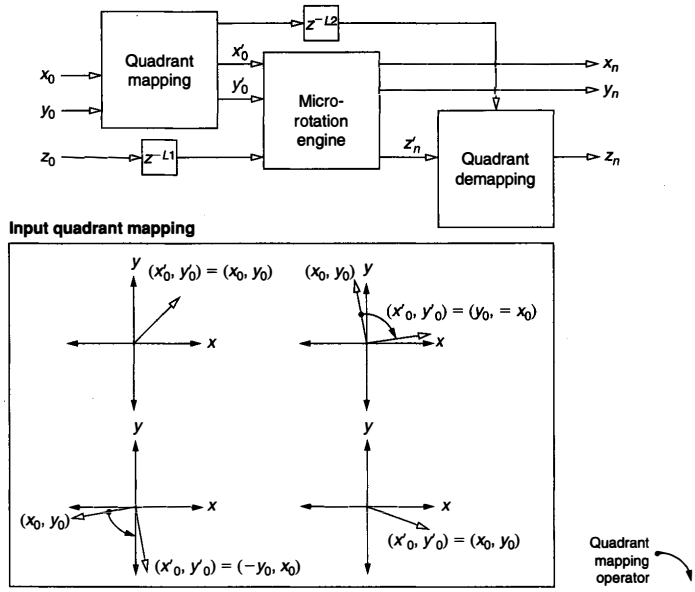


FIGURE 25.7 ■ A full-range CORDIC processor showing input quadrant mapping, micro-rotation engine, and quadrant correction.

of the course angle rotator and matching quadrant correction circuit are shown in Figure 25.8. The area cost for an FPGA implementation of the circuits is modest [40].

25.3.2 Folded CORDIC

The folded CORDIC architecture allocates a single PE to service all of the required micro-rotations. At one architectural extreme a bit-serial implementation employing a single 3-2 full adder, with appropriate control circuitry and state storage, can address all of the required updates for x, y, and z. However, our treatment employs a word-oriented architecture that associates unique functional units (FU) with each of the x, y, and z processing engines, as shown in Figure 25.9.

Multiple mapping options are available when projecting the dependency graph onto an FPGA architecture. In the Xilinx Virtex-4 family [41], one option for supporting the adder/subtractor FUs is to utilize the logic fabric and realize these modules at the cost of one lookup table (LUT) per result digit. So for example, the addition of two 16-bit operands to generate a 17-bit sum requires 17 LUTs. An alternative is to use the 48-bit adder in the DSP48 tile.

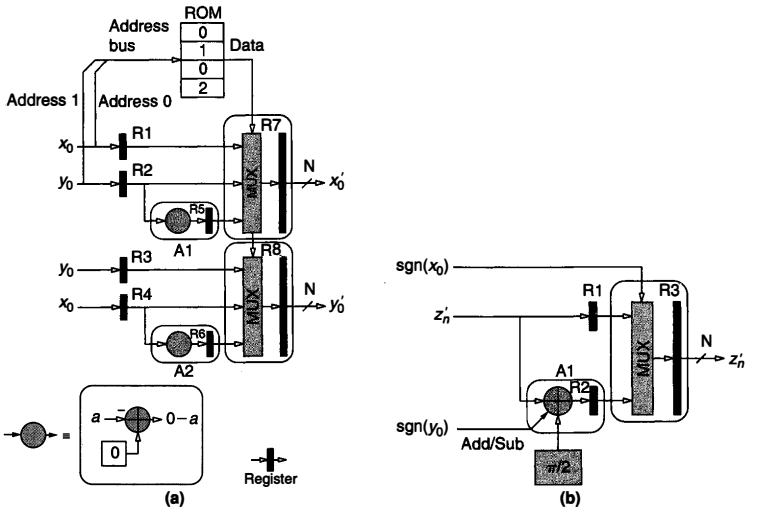


FIGURE 25.8 ■ A course angle rotator preceding a micro-rotation engine for a full-range CORDIC processor (a). A post-micro-rotation quadrant correction circuit (b).

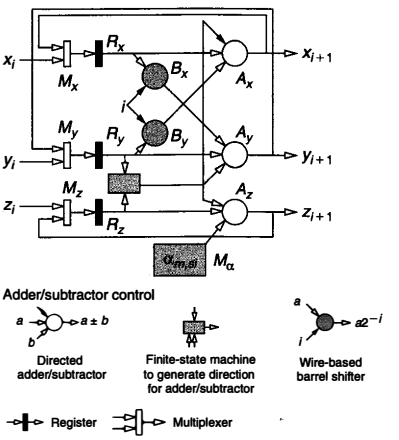


FIGURE 25.9 ■ A folded CORDIC architecture with separate functional units for each of the x, y, and z updates. Only the micro-rotation engine is shown.

There are also several mapping options for the barrel shifter: It can be realized in the logic fabric, with the multiplier in the DSP48 tile, or, for that matter, using an embedded multiplier in any FPGA family that supports this architectural component (e.g., Virtex-II Pro [39] or Spartan-3E [37]).

Consider a fabric-only implementation of a vectoring CORDIC algorithm in circular coordinates. In this case all of the FUs are implemented directly in the logic fabric. The FPGA area, A_F, can be expressed as

$$A_F = 3 \cdot a_{add} + 2 \cdot a_{barrel} + 3 \cdot a_{mux} + a_{LUT} + a_Q + a_{Q-1} \tag{25.20}$$

where a_{add}, a_{barrel}, a_{mux}, a_{LUT}, a_Q, and a_{Q-1} correspond to the area of an adder, barrel shifter, input multiplexer, elementary angle LUT, quadrant input mapper, and output mapper circuits, respectively. The FPGA logic fabric is designed to efficiently support the implementation of arbitrary-precision high-speed adder/subtractors. Each configurable logic block (CLB) [41] includes dedicated circuitry that provides fast carry resolution, with the LUT itself producing the half-sum.

The component that can be costly in terms of area is the barrel shifter. The barrel shifter area cost can be much more significant than the aggregate cost of the adder/subtractors used for updating the x, y, and z variables. For example, in a design that supplies 16 effective result digits, the 2 barrel shifters occupy an aggregate area of 226 LUTs while the adders occupy 74 LUTs in total. Here, the barrel shifters have a footprint approximately three times that of the adders.

The barrel shifter area can be reduced if a multiplier-based barrel shifter is used rather than a purely logic fabric–based implementation. FPGA families such as Spartan-3E [37], Virtex-II Pro [39], and Virtex-4 [40] include an array of embedded multipliers, which are useful for realizing arithmetic shifts. The multiplier accepts 18-bit precision operands and produces a 36-bit result. When used as a barrel shifter, one port of the multiplier is supplied with the input operand that is to experience the arithmetic shift, while the second port accepts the shift value 2^i, where i is the iteration index. In a typical hardware implementation the iteration index rather than the exponentiated value is usually available in the control plane that coordinates the operation of the circuit. The exponentiation can be done via a small LUT implemented using distributed memory [40]. Multiple multiplier primitives can be combined with an adder to form a barrel shifter that can support a wider datapath. For the previous example, multiplier realization of the barrel shifter results in an FPGA footprint that is less than half that of an entirely fabric-based implementation.

The folded CORDIC architecture is a recursive graph, which means that deep pipelining cannot be employed to reduce the critical path. The structure can accept a new set of operands, and produces a result every n clock cycles.

25.3.3 Parallel Linear Array

When throughput is the overriding design consideration, a fully parallel pipelined CORDIC realization is the preferred architecture. With this approach

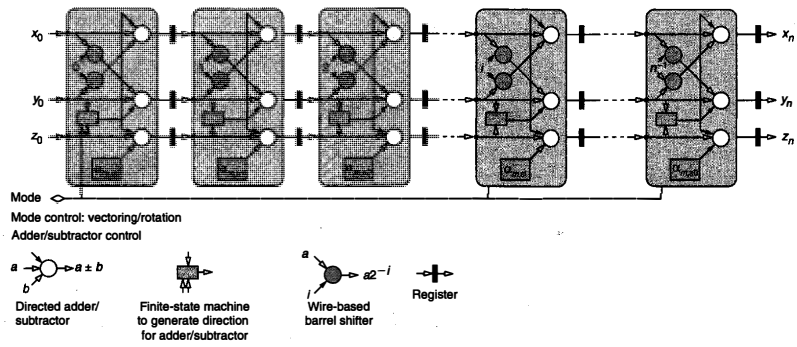


FIGURE 25.10 ■ A programmable parallel pipelined CORDIC array. In a completely unfolded implementation, the barrel shifters are realized as FPGA routing and so consume no resources other than interconnect.

the CORDIC algorithm is completely unrolled and each operation is projected onto a unique hardware resource, as shown in Figure 25.10.

One interesting effect of the unrolling is that the data shifts required in the cross-addition update can be realized as wiring between successive CORDIC processing elements (PEs). Unlike the folded architecture, where either LUTs or embedded multipliers are consumed to realize the barrel shifter, no resources other than interconnect are required to implement the shift in the linear array architecture. The only functional units required for each PE with this approach are three adder/subtractors and a small amount of logic to implement the control circuit that steers the add/subtract FUs. The micro-rotation angle for each PE is encoded as a constant supplied on one arm of the adder/subtractor that performs the angle update—no LUT resources are required for this. Note in Figure 25.10 that the sign bit of the y and z variables is supplied to the control circuit that is local to each processing engine. This permits the architecture to operate in the y- or z-reduction configuration under the control of the Mode input control signal, and thus support vectoring or rotation, respectively.

Figure 25.11(a) shows a comparison of the area functions for the parallel and folded architectures. The folded implementation is entirely fabric based. As expected, the area of the parallel design exhibits modest exponential growth and, for an effective number of result digits greater than 15, occupies more than three times the area of the folded architecture. For the case of 24 effective result digits, the parallel design is larger by a factor of approximately 5. Figure 25.11(b) contrasts the throughput of the two architectures. Naturally, the parallel design has a constant throughput of one CORDIC operation per second for a normalized clock rate of 1, while the throughput for the folded design falls off as the inverse of the number of iterations.

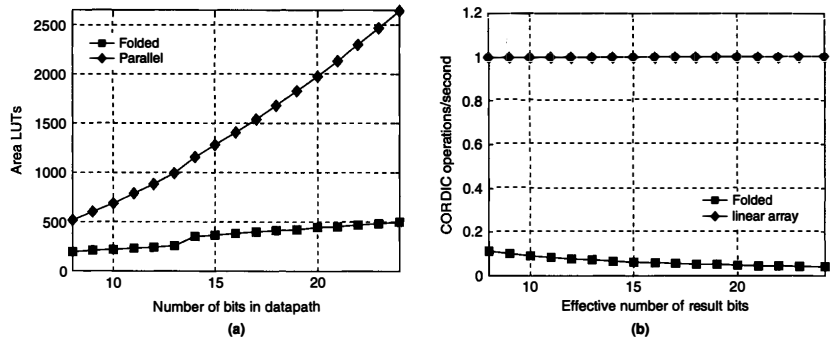


FIGURE 25.11 ■ (a) Comparison of the FPGA resource requirements for folded and linear array CORDIC architectures—circular coordinates. (b) Throughput in rotations/vectoring operations per second for the two architectures. A normalized clock rate of 1 is assumed.

The parallel design has a performance advantage of approximately an order of magnitude for the number of effective result bits great than 10. In an FPGA implementation the advantage is significantly more than this because of the higher clock frequency that can be supported by the linear array compared to the folded processor. With its heavy pipelining, the linear array typically achieves an operating frequency approximately twice that of the folded architecture, so for high-precision calculations—for example, on the order of 24 effective fractional bits or greater—the parallel implementation has a throughput advantage of approximately 50, which is delivered in a footprint that is only five times that of the folded design.

The add/subtract FUs can be realized using the logic fabric or the 48-bit adder that is resident in each DSP48 tile in the Virtex-4 class of FPGAs. The DSP48 [42] is a dynamically configurable embedded processing block that supports over 40 different op-codes, optimized for signal-processing tasks. The logic fabric approach tends to result in an implementation that operates at a lower clock frequency than a fully pipelined version based on the DSP48. The DSP48-based implementations can operate at very high clock frequencies—in the region of 500 MHz in the fastest "–12" speed-grade parts [40]. However, for a datapath precision of up to 36 bits, three DSP48 tiles are required for each CORDIC iteration (see Figures 25.12 and 25.13). For scenarios where throughput is the overarching requirement, these resource requirements are acceptable.

A potential downside to the use of the DSP48 in this application is that the multiplier colocated with the high-precision adder is not available for use by another function if the adder is used by the CORDIC PE. This is because the input and output ports of the block are occupied supporting the addition/subtraction and there is no I/O available to access other functions (such as the multiplier) in the tile.

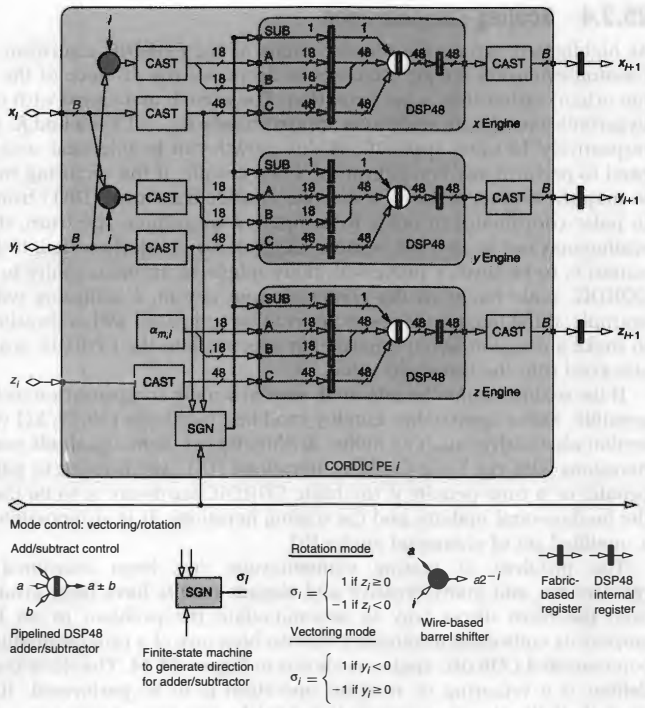


FIGURE 25.12 ■ Processing element i of a Virtex-4 DSP48-based CORDIC processor.

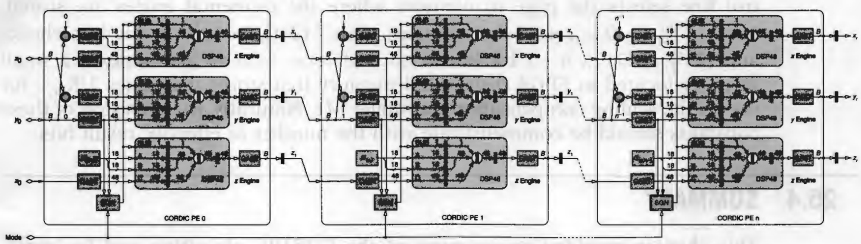


FIGURE 25.13 ■ A programmable parallel pipelined CORDIC array based almost entirely on the Virtex-4 DSP48 embedded tile. Each DSP48 has three levels of pipelining. Additional fabric-based registers are included to pipeline the routing between DSP48 tiles.

25.3.4 Scaling Compensation

As highlighted earlier, the rotation mode of the CORDIC algorithm produces a rotation extension (i.e., it increases or decreases the distance of the point from the origin) rather than a pure rotation. The growth associated with circular and hyperbolic coordinate systems is approximately $K_{1,n} \approx 1.6468$ and $K_{-1,n} \approx 0.8382$, respectively. In some applications this growth can be tolerated, and there is no need to perform any compensation. For example, if the vectoring mode is used to map the output vector of a discrete Fourier transform (DFT) from Cartesian to polar coordinates in order to compute a magnitude spectrum, the CORDIC scaling may not be an issue because all terms are similarly scaled. If the CORDIC output is to be further processed, there might be an opportunity to absorb the CORDIC scale factor in the postprocessing circuit. Continuing with the DFT example, if the magnitude spectrum is to be compared with a threshold in order to make a decision about a particular spectral bin, the CORDIC scaling can be absorbed into the threshold value.

If the scaling cannot be tolerated, several scaling compensation techniques are possible. Some approaches employ modified iterations [20, 32, 33] while others exploit alternatives such as online arithmetic [6]. Some methods merge scaling iterations with the basic CORDIC iterations [15], which result in either an area penalty or a time penalty if the basic CORDIC hardware is to be used for both the fundamental updates and the scaling iterations. It is also possible to employ a modified set of elemental angles [9].

The problem of scaling compensation has been examined by many researchers, and many creative and elegant results have been produced; however, the most direct way to accommodate the problem in an FPGA is to employ its embedded multipliers. The architecture of a programmable and scale-compensated CORDIC engine is shown in Figure 25.14. The `Mode` control signal defines if a vectoring or rotation operation is to be performed. It essentially controls if the iteration update is guided by the sign of the y or z variable for vectoring or rotation, respectively. The `Coordinate_System` signal selects the coordinate system for the processor: circular, hyperbolic, or linear. This control line selects the page in memory where the elemental angles are stored: $\tan^{-1}(2^{-i}), i = 0, \ldots, n-1$ for circular; $\tanh^{-1}(2^{-i}), i = 1, \ldots, n$ for hyperbolic; and $(2^{-i}), i = 0, \ldots, n-1$ for linear. `Coordinate_System` also indexes a small memory located in FPGA distributed memory that stores the values $1/K_{m,n}$ for use by the scaling compensation multiplier $M1$. Naturally, the precision of these constants should be commensurate with the number of effective result bits.

25.4 SUMMARY

This chapter provided an overview of the CORDIC algorithm and its implementation in current-generation FPGAs such as the Xilinx Virtex-4 family. The basic set of CORDIC equations was first reviewed, and the utility of this simple shift-and-add-type algorithm was highlighted by the many functions that can be accessed through it. We also highlighted the fact that, while there are many options for architecting math functions in hardware, the CORDIC approach

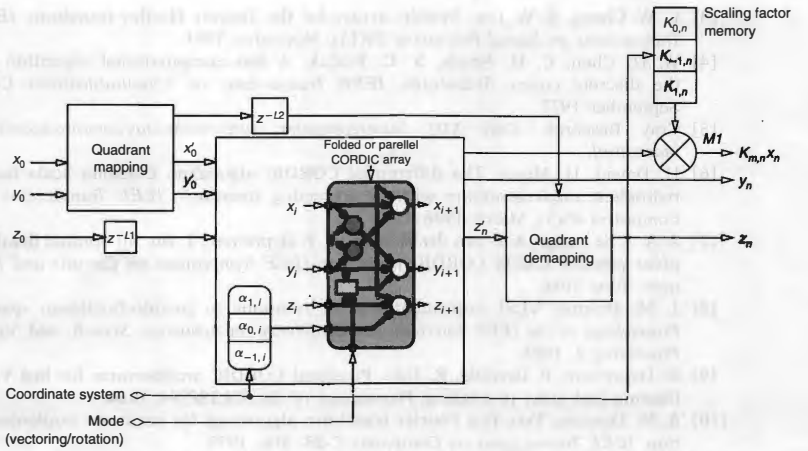


FIGURE 25.14 ■ A programmable CORDIC processor with multiplier-based scaling compensation.

comes into its own when multi-element input and output vectors are involved. The functional requirements of the angle and cross-addition updates make it an excellent match for FPGAs because of the utility and efficiency with which these devices realize addition and subtraction.

Most hardware realizations of the CORDIC algorithm employ fixed-point arithmetic, and this is certainly true of nearly all FPGA implementations. We showed that it is therefore important to understand the effects of quantizing the datapath. While this analysis can be complex [16], for most applications the simplified approach first described by Walther [36] is suitable for most cases and provides excellent results.

The FPGA implementation of a CORDIC processor would appear to be straightforward. However, FPGA-embedded functions such as multipliers and the DSP48 provide opportunities for architectural innovation and for design trade-offs that satisfy design requirements. For example, embedded multipliers can be exchanged for logic fabric with the implementation of the barrel shifter. The wide 48-bit adder in the DSP48 can be used almost as the sole arithmetic building block of a complete fully parallel CORDIC array.

References

[1] J. R. Cavallaro, F. T. Luk. CORDIC arithmetic for an SVD processor. *Journal of Parallel and Distributed Computing* 5, 1988.

[2] J. R. Cavallaro, F. T. Luk. Floating-point CORDIC for matrix computations. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1988.

[3] L. W. Chang, S. W. Lee. Systolic arrays for the discrete Hartley transform. *IEEE Transactions on Signal Processing* 29(11), November 1991.
[4] W. H. Chen, C. H. Smith, S. C. Fralick. A fast computational algorithm for the discrete cosine Transform. *IEEE Transactions on Communications* C-25, September 1977.
[5] Cray Research. *Cray XD1 Supercomputer, http://www.cray.com/products/xd1/index.html.*
[6] H. Dawid, H. Meyer. The differential CORDIC algorithm: Constant scale factor redundant implementation without correcting iterations. *IEEE Transactions on Computers* 45(3), March 1996.
[7] A. A. J. de Lange, A. J. van der Hoeven, E. F. Deprettere, J. Bu. An optimal floating-point pipeline CMOS CORDIC processor. *IEEE Symposium on Circuits and Systems*, June 1988.
[8] J. M. Delsme. VLSI implementation of rotations in pseudo-Euclidean spaces. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* 2, 1983.
[9] E. Deprettere, P. Dewilde, R. Udo. Pipelined CORDIC architectures for fast VLSI filtering and array processing. *Proceedings of the ICASSP'84*, 1984.
[10] A. M. Despain. Very fast Fourier transform algorithms for hardware implementation. *IEEE Transactions on Computers* C-28, May 1979.
[11] A. M. Despain. Fourier transform computers using CORDIC iterations. *IEEE Transactions on Computers* 23, October 1974.
[12] C. Dick, F. Harris, M. Rice. FPGA implementation of carrier phase synchronization for QAM demodulators. *Journal of VLSI Signal Processing, Special Issue on Field-Programmable Logic* (R. Woods, R. Tessier, eds.), Kluwer Academic, January 2004.
[13] D. Ercegovac, T. Lang. *Digital Arithmetic*, Morgan Kaufmann, 2004.
[14] B. Haller, J. Gotze, J. Cavallaro. Efficient implementation of rotation operations for high-performance QRD-RLS filtering. *Proceedings of the International Conference on Application-Specific Systems, Arthictectures and Processors*, July 1997.
[15] G. H. Haviland, A. A. Tuszinsky. A CORDIC arithmetic processor chip. *IEEE Transactions on Computers* c-29(2), February 1980.
[16] Y. H. Hu. The quantization effects of the CORDIC algorithm. *IEEE Transactions on Signal Processing* 40, July 1992.
[17] X. Hu, S. C. Bass. A neglected error source in the CORDIC algorithm. *IEEE International Symposium on Circuits and Systems* 1, May 1993.
[18] X. Hu, S. C. Bass. A neglected error source in the CORDIC algorithm. *Proceedings of the IEEE ISCAS*, 1993.
[19] X. Hu, R. G. Garber, S. C. Bass. Expanding the range of convergence of the CORDIC algorithm. *IEEE Transactions on Computers* 40(1), January 1991.
[20] J. Lee. Constant-factor redundant CORDIC for angle calculation and rotation. *IEEE Transactions on Computers* 41(8), August 1992.
[21] Y. H. Liao, H. E. Liao. CALF: A CORDIC adaptive lattice filter. *IEEE Transactions on Signal Processing* 40(4), April 1992.
[22] Mathworks, The, *http://www.mathworks.com/*.
[23] U. Mengali, A. N. D'Andrea. *Synchronization Techniques for Digital Receivers*, Plenum Press, 1997.
[24] J. Mia, K. K. Parhi, E. F. Deprettere. Pipelined implementation of CORDIC-based QRD-MVDR adaptive beamforming. *IEEE Fourth International Conference on Signal Processing*, October 1998.
[25] J. M. Muller. Discrete basis and computation of elementary functions. *IEEE Transactions on Computers* C-34(9), September 1985.

[26] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.
[27] K. K. Parhi. *VLSI Digital Signal Processing Systems Design and Implementation*, John Wiley, 1999.
[28] S. Y. Park, N. I. Cho. Fixed-point error analysis of CORDIC processor based on the Variance Propagation Formula. *IEEE Transactions on Circuits and Systems* 51(3), March 2004.
[29] J. G. Proakis, M. Salehi. *Communication Systems Engineering*, Prentice-Hall, 1994.
[30] C. M. Rader. VLSI systolic arrays for adaptive nulling. *IEEE Signal Processing Magazine* 13(4), July 1996.
[31] T. Y. Sung, Y. H. Hu. Parallel VLSI implementation of Kalman filter. *IEEE Transactions on Aerospace and Electronic Systems* AES 23(2), March 1987.
[32] N. Takagi. Redundant CORDIC methods with a constant scale factor for sine and cosine computation. *IEEE Transactions on Computers* 40(9), September 1991.
[33] D. H. Timmerman, B. J. Hosticka, B. Rix. A new addition scheme and fast scaling factor compensation methods for CORDIC algorithms. *Integration, the VLSI Journal* (11), 1991.
[34] D. H. Timmerman, B. J. Hosticka, G. Schmidt. A programmable CORDIC chip for digital signal processing applications. *IEEE Journal of Solid-State Circuits* 26(9), September 1991.
[35] J. E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers* 3, September 1959.
[36] J. S. Walther. A unified algorithm for the elementary functions. *AFIPS Spring Joint Computer Conference* 38, 1971.
[37] Xilinx Inc. Spartan-3E Datasheet, *http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category= /Data+Sheets/FPGA+Device+Families/Spartan-3E.*
[38] Xilinx Inc. System Generator for DSP, *http://www.xilinx.com/ise/optional_prod/system_generator.htm.*
[39] Xilinx Inc. Virtex-II Pro Datasheet, *http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?category=Publications/FPGA+Device+Families/Virtex-II+Pro&iLanguageID=1.*
[40] Xilinx Inc. Virtex-4 Datasheet, *http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?sGlobalNavPick=&sSecondaryNavPick=&category=-1210771&iLanguage ID=1.*
[41] Xilinx Inc. Virtex-4 Multi-Platform FPGA, *http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.*
[42] Xilinx Inc. XtremeDSP Design Considerations Guide, *http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/xtremedsp.htm.*
[43] Xilinx Inc. XtremeDSP Slice, *http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/xtremedsp.htm.*

CHAPTER 26

Hardware/Software Partitioning

Frank Vahid, Greg Stitt
Department of Computer Science and Engineering
University of California–Riverside

Field-programmable gate arrays (FPGAs) excel at implementing applications as highly parallel custom circuits, thus yielding fast performance. However, large applications implemented on a microprocessor may be more size efficient and require less designer effort, at the expense of slower performance. In some cases, mapping an entire application to a microprocessor satisfies performance requirements and so is preferred. In other cases, mapping an application entirely to custom circuits on FPGAs may be necessary to meet performance requirements. In many cases, though, the best implementation lies somewhere between these two extremes.

Hardware/software partitioning, illustrated in Figure 26.1, is the process of dividing an application between a microprocessor component ("software") and one or more custom coprocessor components ("hardware") to achieve an implementation that best satisfies requirements of performance, size, designer effort, and other metrics.[1] A custom coprocessor is a processing circuit that is tailor-made to execute critical application computations far faster than if those computations had been executed on a microprocessor.

FPGA technology encourages hardware/software (HW/SW) partitioning by simplifying the job of implementing custom coprocessors, which can be done just by downloading bits onto an FPGA rather than by manufacturing a new integrated circuit or by wiring a printed-circuit board. In fact, new FPGAs even support integration of microprocessors within an FPGA itself, either as separate physical components alongside the FPGA fabric ("hard-core microprocessors") or as circuits mapped onto the FPGA fabric just like any other circuit ("soft-core microprocessors"). High-end computers have also begun integrating microprocessors and FPGAs on boards, allowing application designers to make use of both resources when implementing applications.

Hardware/software partitioning is a hard problem in part because of the large number of possible partitions. In its simplest form, hardware/software partitioning considers an application as comprising a set of *regions* and maps

[1] The terms *software*, to represent microprocessor implementation, and *hardware*, to represent coprocessor implementation, are common and so appear in this chapter. However, when implemented on FPGAs, coprocessors are actually just as "soft" as programs implemented on a microprocessor, with both consisting merely of a sequence of bits downloaded into a physical device, leading to a broader concept of "software."

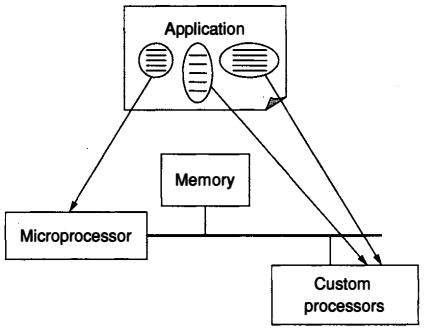


FIGURE 26.1 ■ A diagram of hardware/software partitioning, which divides an application between a microprocessor component ("software") and custom processor components ("hardware").

each region to either software or hardware such that some cost criteria (e.g., performance) is optimized while some constraints (e.g., size) are satisfied.

A *partition* is a complete mapping of every region to either hardware or software. Even in this simple formulation, the number of possible partitions can be enormous. If there are n regions and there are two choices (software or hardware) for each one, then there are 2^n possible partitions. A mere 32 regions yield over 4 billion possibilities. Finding the optimal partition of this simple form is known to be NP-hard in general. Many other factors contribute to making the problem even harder, as will be discussed.

This chapter discusses issues involved in partitioning an application among microprocessor and coprocessor components. It considers two application categories: *sequential programs*, where an application is a program written in a sequential programming language such as C, C++, or Java and where partitioning maps critical functions and/or loops to coprocessors; and *parallel programs*, where an application is a set of concurrently executing tasks and where partitioning maps some of those tasks to coprocessors.

While designers today do mostly manual partitioning, automating the process has been an area of active study since the early 1990s (e.g., [10, 15, 26]) and continues to be intensively researched and developed. For that reason, we will begin the chapter with a discussion of the trend toward automatic partitioning.

26.1 THE TREND TOWARD AUTOMATIC PARTITIONING

Traditionally, designers have manually partitioned applications between microprocessors and custom coprocessors. Manual partitioning was in part necessitated by radically different design flows for microprocessors versus coprocessors. A microprocessor design flow typically involved developing code

in programming languages such as C, C++, or Java. In sharp contrast, a coprocessor design flow may have involved developing cleverly parallelized and/or pipelined datapath circuits, control circuits to sequence data through the datapath, memory circuits to enable rapid data access by the datapath, and then mapping those circuits to a particular ASIC technology. Thus, manual partitioning was necessary because partitioning was done early in the design process, well before a machine-readable or executable description of an application's desired behavior existed. It resulted in specifications for both the software design and the hardware design teams, both of which might then have worked for many months developing their respective implementations.

However, the evolution of synthesis and FPGA technologies is leading toward automated partitioning because the starting point of FPGA design has been elevated to the same level as that for microprocessors, as shown in Figure 26.2.

Current technology enables coprocessors to be realized merely by downloading bits onto an FPGA. Downloading takes just seconds and eliminates the months-long and expensive design step of mapping circuits to an ASIC. Furthermore, synthesis tools have evolved to automatically design coprocessors from high-level descriptions in hardware description languages (HDLs), such as VHDL or Verilog, or even in languages traditionally used to program microprocessors, such as C, C++, or Java. Thus, designers may develop a single machine-readable high-level executable description of an application's desired behavior and then partition that description between microprocessor and coprocessor parts, in a process sometimes called hardware/software codesign. New

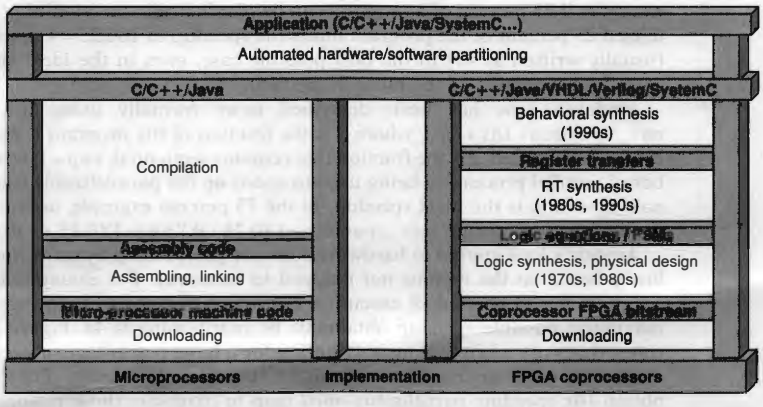


FIGURE 26.2 ■ The codesign ladder: evolution toward automated hardware/software partitioning due to synthesis tools and FPGA technologies enabling a similar design starting point, and similar implementation manner of downloading bits into a prefabricated device.

approaches, such as SystemC [14], which supports HDL concepts using C++, have evolved specifically to support it. With a single behavior description of an application, and automated tools to convert partitioned applications to coprocessors, automating partitioning is a logical next step in tool evolution. Some commercial automated hardware/software partitioning products are just beginning to appear [4,7,21,27].

In the remainder of the chapter, many of the issues discussed relate to both manual and automatic partitioning, while some relate to automatic partitioning alone.

26.2 PARTITIONING OF SEQUENTIAL PROGRAMS

In a sequential program, the regions comprising an application's behavior are defined to execute sequentially rather than concurrently. For example, the semantics of the C programming language are such that its functions execute sequentially (though parallel execution is allowed as long as the results of the computation stay the same). Hardware/software partitioning of a sequential program involves speeding up certain regions by moving them to faster-executing FPGA coprocessors, yielding overall application speedup.

Hardware/software partitioning of sequential programs is governed to a large extent by the well-known Amdahl's Law [1] (described in 1967 by Gene Amdahl of IBM in the context of discussing the limits of parallel architectures for speeding up sequential programs). Informally, Amdahl's Law states that application speedup is limited by the part of the program *not* being parallelized. For example, if 75 percent of a program can be parallelized, the remaining nonparallelized 25 percent of the program limits the speedup to 100/25 = 4 times speedup (usually written as 4*x*) in the best possible case, even in the ideal situation of zero-time execution of the other 75 percent.

Amdahl's Law has been described more formally using the equation $\texttt{max\_speedup} = 1/(s + p/n)$, where p is the fraction of the program execution that can be parallelized; s is the fraction that remains sequential, $s + p = 1$; n is the number of parallel processors being used to speed up the parallelizable fraction; and `max_speedup` is the ideal speedup. In the 75 percent example, assuming that n is very large, we obtain $\texttt{max\_speedup} = 1/(0.25 + 0.75/n) = 1/(0.25 + \sim 0) = 4\text{x}$.

Amdahl's Law applies to hardware/software partitioning by providing speedup limits based on the regions *not* mapped to hardware. For example, if a region accounts for 25 percent of execution but is not mapped to hardware, then the maximum possible speedup obtainable by partitioning is 4x. Figure 26.3 illustrates that only when regions accounting for a large percentage of execution are mapped to hardware might partitioning yield substantial results. For example, to obtain 10x speedup, partitioning *must* map to hardware those regions accounting for *at least* 90 percent of an application's execution time.

Fortunately, most of the execution time for many applications comes from just a few regions. For example, Figure 26.4 shows the average execution time

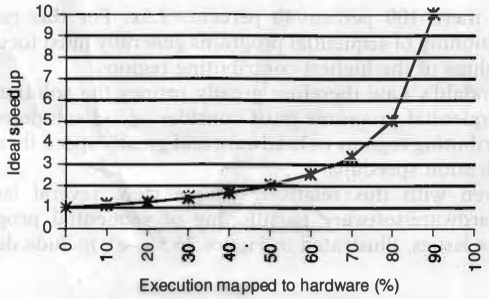


FIGURE 26.3 ■ Hardware/software partitioning speedup following Amdahl's Law.

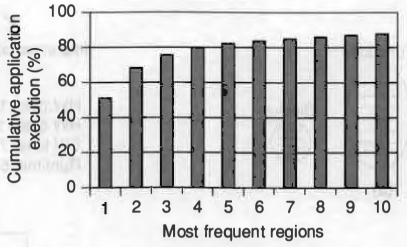


FIGURE 26.4 ■ Ideal speedups achievable by moving regions (loops) to hardware, averaged for a variety of embedded system benchmark suites (MediaBench, Powerstone, and Netbench).

contribution for the first n regions (in this case loops) for several dozen standard embedded system application benchmarks, all sequential programs. Note that the first few regions account for 75 to 80 percent of the execution time. The regions are roughly equal in size following the well-known informal "90–10" rule, which states that that 90 percent of a program's execution time is spent in 10 percent of its code. Thus, hardware/software partitioning of sequential programs generally must sort regions by their execution percentage and then consider moving the highest contributing regions to hardware.

A corollary to Amdahl's Law is that if a region is moved to hardware, its actual speedup limits the remaining possible speedup. For example, consider a region accounting for 80 percent of execution time that, when moved to hardware, runs only 2x faster than in software. Such a situation is equivalent to 40 percent of the region being sped up ideally and the other 40 percent not being sped up at all. With 40 percent not sped up, the ideal speedup obtainable by partitioning of the remaining regions (the other 20 percent) is limited

to a mere 100 percent/40 percent = 2.5x. For this reason, hardware/software partitioning of sequential programs generally must focus on obtaining very large speedups of the highest-contributing regions.

Amdahl's Law therefore greatly prunes the solution space that partitioning of sequential programs must consider—good solutions must move the biggest-contributing regions to hardware and greatly speed them up to yield good overall application speedups.

Even with this relatively simple view, several issues make the problem of hardware/software partitioning of sequential programs quite challenging. Those issues, illustrated in Figure 26.5(a–e), include determining critical region

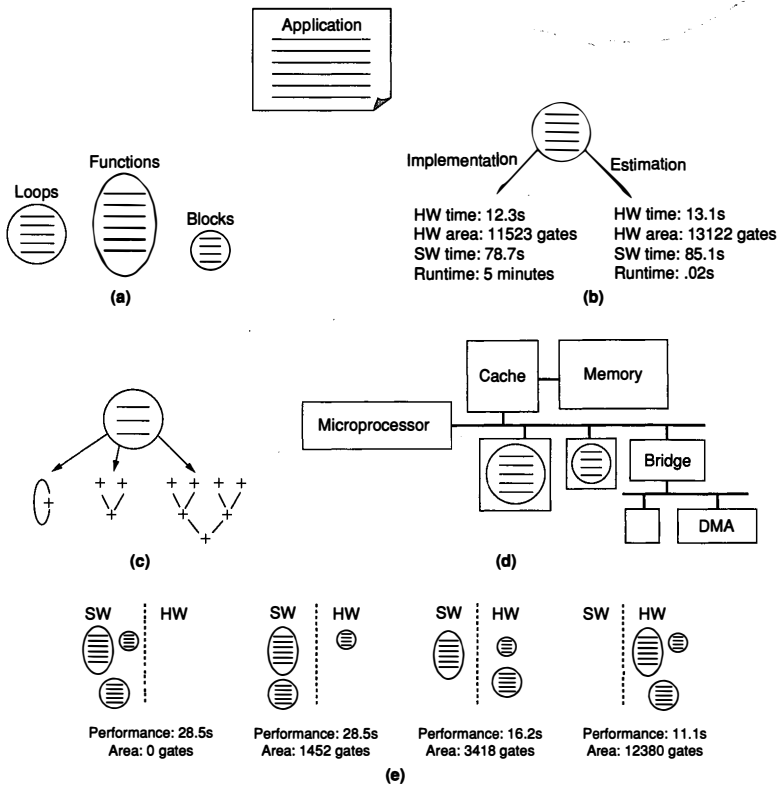


FIGURE 26.5 ■ Hardware/software partitioning: (a) granularity; (b) partition evaluation; (c) alternative region implementations; (d) implementation models; (e) exploration.

granularity (a), evaluating partitions (b), considering multiple alternative implementations of a region (c), determining implementation models (d), and exploring the partitioning solution space (e).

26.2.1 Granularity

Partitioning moves some code regions from a microprocessor to coprocessors. A first issue in defining a partitioning approach is thus to determine the granularity of the regions to be considered. *Granularity* is a measure of the amount of functionality encapsulated by a region, which is illustrated in Figure 26.6.

A key trade-off involves coarse versus fine region granularity [11]. Coarser granularity simplifies partitioning by reducing the number of possible partitions, enables more accurate estimates during partitioning by considering more computations when creating those estimates (and thus reducing inaccuracy when combining multiple estimates for different regions into one), and reduces inter-region communication. On the other hand, finer granularity may expose better

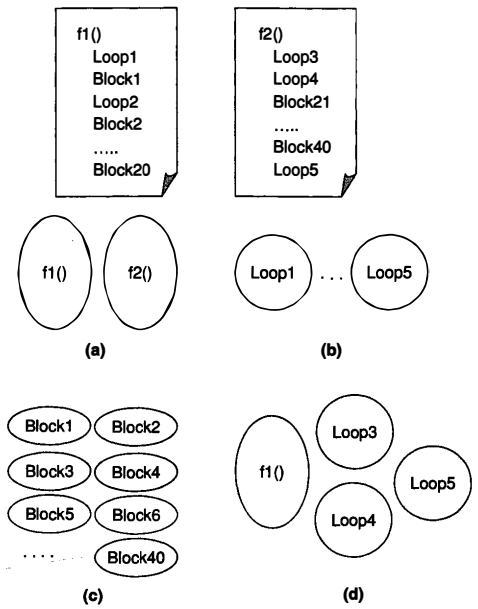


FIGURE 26.6 ■ The region granularities of an application (*top*): (a) functions; (b) loops; (c) blocks; (d) heterogeneous combination. Finer granularities may expose better solutions, at the expense of a more complex partitioning problem and more difficult estimation challenges.

partitions that would not otherwise be possible. Early automated partitioning research considered fine granularities of arithmetic operations or statements, while more recent work typically considers coarser granularities involving basic blocks, loops, or entire functions.

Coarse granularity simplifies the partitioning problem by reducing the number of possible partitions. Take, for example, an application with two 1000-line C functions, like the one shown in Figure 26.6 (*top*), and consider partitioning at the granularity of functions, loops, or basic blocks. The granularity of functions involves only two regions, as shown in Figure 26.6(a), and the granularity of loops involves five regions, as shown Figure 26.6(b). However, the granularity of the basic block may involve many tens or hundreds of regions, as shown in Figure 26.6(c). If partitioning simply chooses between hardware and software, then two regions would yield 2*2 = 4 possible partitions, while just 32 regions would involve 2*2*2*...*2 (32 times) possible partitions, or over four billion.

Coarse granularity also enables more accurate early estimations of a region's performance, size, power, and so forth. For example, an approach using function granularity could individually presynthesize the two previously mentioned functions to FPGAs before partitioning, gathering performance and size data. During partitioning, it could simply estimate that, for the case of partitioning both functions to the FPGA, the two functions' performances would stay the same and their sizes would add. This estimate is not entirely accurate because synthesizing both functions could involve interactions between the function's implementations that would impact performance and size, but it is likely *reasonably* accurate. In contrast, similar presynthesis and performance/size estimates for basic blocks would yield grossly inaccurate values because multiple basic blocks would actually be synthesized into a combined circuit having extensive sharing among the blocks, bearing little resemblance to the individual circuits presynthesized for each block.

However, finer granularity may expose better partitions that otherwise would not be possible. In the two-function example just described, perhaps the best partition would move only half of one function to hardware—an option not possible at the coarse granularity of functions but possible at finer granularities of loops or basic blocks.

Manual partitioning often involves initially considering a "natural" granularity for an application. An application may consist of dozens of functions, but a designer may naturally categorize them into just a few key high-level functions. A data-processing application, for example, may naturally consist of several key high-level functions: acquire, decompress, transform, compress, and transmit. The designer may first try to partition at that natural granularity before considering finer granularities.

Granularity may be restricted to one region type, but can instead be *heterogeneous*, as shown in Figure 26.6(d). For example, in the previous two-function example from Figure 26.6 (*top*), one function may be treated as a region while the other may be broken down so that its loops are each considered as a region. A particular loop may even be broken down so that its basic blocks are individually considered as regions. Thus, for a single application, regions considered

for movement to hardware may include functions, loops, and basic blocks. With heterogeneous granularity, preanalysis of the code may select regions based on execution time and size, breaking down a region with very high execution time or large size.

Furthermore, while granularity can be predetermined statically, it can also be determined *dynamically* during partitioning [16]. Thus, an approach might start with coarse-grained regions and then decompose specific regions deemed to be critical during partitioning.

Granularity need not be restricted to regions defined by the language constructs such as functions or loops, used in the original application description. Transformations, some being well-known compiler transformations, may be applied to significantly change the original description. They include function inlining (replacing a function call with that function's statements), function "exlining" (replacing statements with a function call), function cloning (making multiple copies of a function for use in different places), function specialization (creating versions of a function with constant parameters), loop unrolling (expanding a loop's body to incorporate multiple iterations), loop fusion (merging two loops into one), loop splitting (splitting one loop into two), code hoisting and sinking (moving code out of and into loops), and so on.

26.2.2 Partition Evaluation

The process of finding a good partition is typically iterative, involving consideration and evaluation of certain partitions and then decisions as to which partitions to consider next. *Evaluation* determines a partition's design metric values. A *design metric* is a measure of a partition. Common metrics include performance, size, and power/energy. Other metrics include implementation cost, engineering cost, reliability, maintainability, and so on.

Some design metrics may need to be *optimized*, meaning that partitioning should seek the best possible value of a metric. Other design metrics may be *constrained*, meaning that partitioning must meet some threshold value for a metric. An *objective function* is one that combines multiple metric values into a single number, known as *cost*, which the partitioning may seek to minimize. A partitioning approach must define the metrics and constraints that can be considered, and define or allow a user to define an objective function.

Evaluation can be a complex problem because it must consider several implementation factors in order to obtain accurate design metric values. Among others, these factors include determining the communication time between regions that transfer data (thus requiring knowledge of the communication structure), considering clock cycle lengthening caused by multiple application regions sharing hardware resources (which may introduce multiplexers or longer wires), and the like.

The key trade-off in evaluation involves estimation versus implementation. Estimating design metric values is faster and so enables consideration of more possible partitions. Obtaining the values through implementation is more accurate and thus ensures that partitioning decisions are based on sound evaluations.

Estimation involves some characterization of an application's regions before partitioning and then, during partitioning, quickly combining the characterizations into design metric values. The previous section on granularity discussed how two C function regions could be characterized for hardware by synthesizing each region individually to an FPGA, resulting in a characterization of each region consisting of performance and size data. Then a partition with multiple regions in hardware could be evaluated simply by assuming that each region's performance is the same as the predetermined performance and by adding any hardware-mapped region sizes together to obtain total hardware size. Estimation for software can be done similarly, using compilation rather than synthesis for characterization.

Nevertheless, while estimation typically works well for software [24], the nature of hardware may introduce significant inaccuracy into an estimation approach because multiple regions may actually share hardware resources, thus intertwining their performance and size values [9,18]. Alternatively, implementation as a means of evaluation involves synthesizing actual hardware circuits for a given partition's hardware regions. Such synthesis thus accounts for hardware sharing and other interdependencies among the regions. However, synthesis is time consuming, requiring perhaps tens of seconds, minutes, or even hours, restricting the number of partitions that can be evaluated.

Many approaches exist between the two extremes just described. Estimation can be improved with more extensive characterization, incorporating much more detail than just performance and size. Characterization may, for example, describe what hardware resources a region utilizes, such as two multipliers or 2 Kbytes of RAM. Then estimation can use more complex algorithms to combine region characterizations into actual design metric values, such as that the regions may share resources such as multipliers (possibly introducing multiplexers to carry out such sharing) or RAM. These algorithms yield higher accuracy but are still much faster than synthesis. Alternatively, synthesis approaches can be improved by performing a "rough" rather than a complete synthesis, using faster heuristics rather than slower, but higher-optimizing heuristics, for example.

Evaluation need not be done in a single exploration loop of partitioning, but can be *heterogeneous*. An outer exploration loop may be added to partitioning that is traversed less frequently, with the inner exploration loop considering thousands of partitions (if automated) and using estimation for evaluation, while the outer exploration loop considers only tens of partitions that are evaluated more extensively using synthesis. The inner/outer loop concept can of course be extended to even more loops, with the inner loops examining more partitions evaluated quickly and the outer loops performing increasingly in-depth synthesis on fewer partitions.

Furthermore, evaluation methods can change *dynamically* during partitioning. Early stages in the partitioning process may use fast estimation techniques to map out the solution space and narrow in on particular sections of it, while later stages may utilize more accurate synthesis techniques to fine-tune the solution.

26.2.3 Alternative Region Implementations

Further adding to the partitioning challenge is the fact that a given region may have *alternative region implementations* in hardware rather than just one implementation, as assumed in the previous sections. For example, Figure 26.7 (*top*) shows a particular function that performs 100 multiplications. A fast but large hardware implementation may use 100 multipliers, as shown in Figure 26.7(a). The much smaller but much slower hardware implementation in Figure 26.7(b) uses only 1 multiplier. Numerous implementation alternatives exist between those two extremes, such as having 2 multipliers as in Figure 26.7(c), 10 multipliers, and so on. Furthermore, the function may be implemented in a pipelined or non-pipelined manner. Utilized components may be fast and large (e.g., array-style multipliers or carry-lookahead adders) or small and slow (e.g., shift-and-add multipliers or carry-ripple adders). Many other alternatives exist.

A key trade-off involves deciding how many alternative implementations to consider during partitioning. More alternatives greatly expand the number of possible partitions and thus may possibly lead to improved results. However, they also expand the solution space tremendously. For example, 8 regions each with one hardware implementation yield $2^8 = 256$ possible partitions. If each

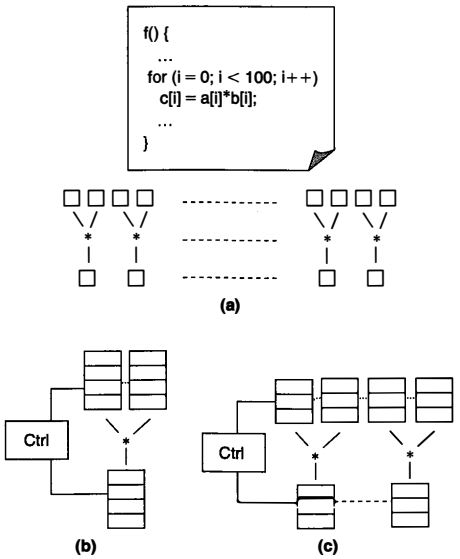


FIGURE 26.7 ■ Alternative region implementations for an original application (*top*) requiring 100 multiplications: (a) 100 multipliers; (b) 1 multiplier; (c) 2 multipliers. Alternative region implementations may have hugely different performances and sizes.

region instead has 4 possible hardware implementations, then it has 5 possible implementations (1 software and 4 hardware implementations), yielding 5^8, or more than 300,000, possible partitions.

Most automated hardware/software partitioning approaches consider one possible hardware implementation per region. Even then, a question exists as to which one to consider for that region: the fastest, the smallest, or some alternative in the middle? Some approaches do consider multiple alternative implementations, perhaps selecting a small number that span the possible space, such as small, medium, and large [5].

As we saw with granularity and evaluation, the number of alternative implementations considered can also be *heterogeneous*. Partitioning may consider only one alternative for particular regions and multiple alternatives for other regions deemed more critical.

Furthermore, as we saw with granularity and evaluation, the number of alternative implementations can change *dynamically* as well. Partitioning may start by considering only a few alternatives per region and then consider more for particular regions as partitioning narrows in on a solution.

Sometimes obtaining alternative implementations of an application region may require the designer to write several versions of it, each leading to one or more alternatives. In fact, a designer may have to write different region versions for software and hardware because a version that executes fast in software may execute slow in hardware, and vice versa. That difference is due to software's fundamental sequential execution model that demands clever sequential algorithms, while hardware's inherently parallel model demands parallelizable algorithms.

26.2.4 Implementation Models

Partitioning moves critical microprocessor software regions to hardware coprocessors. Different *implementation models* define how the coprocessors are integrated with the microprocessor and with one another [6], enlarging the possible solution space for partitioning and greatly impacting performance and size.

One implementation model parameter is whether coprocessor execution and microprocessor execution overlap or are mutually exclusive. In the overlapping model, the microprocessor activates a coprocessor and may then continue to execute concurrently with it (if the data dependencies of the application allow). In the mutually exclusive model, the microprocessor waits idly until the coprocessor finishes, at which time the microprocessor resumes execution.

Figure 26.8(a) illustrates the execution of both models. Overlapping may improve overall performance, but mutual exclusivity simplifies implementation by eliminating issues related to memory contention, cache coherency, and synchronization—the coprocessor may even access cache directly. In many partitioned implementations, the coprocessor executes for only a small fraction of the total application cycles, meaning that overlapping gains little performance improvement. When the microprocessor and coprocessor cycles are closer to

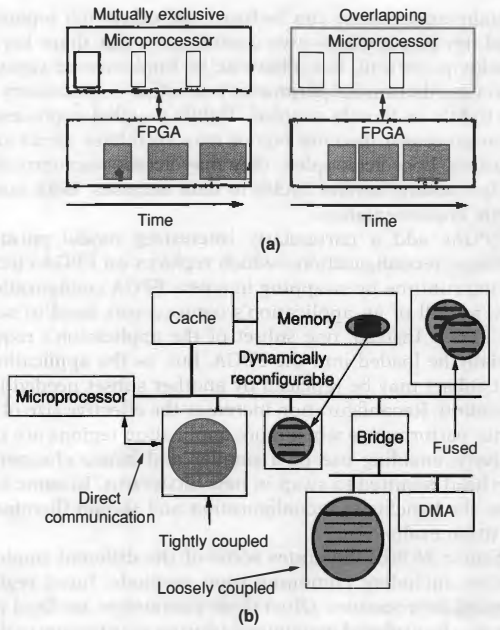


FIGURE 26.8 ■ Implementation models: (a) mutually exclusive and overlapping. (b) implementation model parameters.

being equal, overlapping may improve performance, up to a limit of 2 times, of course. Similarly, the execution of coprocessors relative to one another may be overlapped or mutually exclusive.

A second implementation model parameter involves communication methods. The microprocessor and coprocessors may communicate through memory and share the same data cache, or the microprocessor may communicate directly with the FPGA through memory-mapped registers, queues, fast serial links, or some combination of those mechanisms.

Another implementation model parameter is whether multiple coprocessors are implemented separately or are fused. In a separate coprocessor model, each critical region is synthesized to its own controller and datapath. In a fused model, the critical regions are synthesized into a single controller and datapath. The fused model may reduce size because the hardware resources are shared, but it may result in performance overhead because of a longer critical path as well as the need to run at the slowest clock frequency of all the regions.

Certain coprocessors can be fused and others left separate. Furthermore, fusing need not be complete—two coprocessors can share key components, such as a floating-point unit, but otherwise be implemented separately.

Yet another model parameter is whether coprocessors and the microprocessor are tightly or loosely coupled. Tightly coupled coprocessors may coexist on the microprocessor memory bus or may even have direct access to microprocessor registers. Loosely coupled, they may access microprocessor memory through a bridge, adding several cycles to data accesses. Both couplings can coexist in a single implementation.

FPGAs add a particularly interesting model parameter to partitioning—dynamic reconfiguration—which replaces an FPGA circuit with another circuit during runtime by swapping in a new FPGA configuration bitstream [2]. In this way, not all of an application's coprocessors need to simultaneously coexist in the FPGA. Instead, one subset of the application's required coprocessors may initially be loaded into the FPGA, but, as the application continues to execute, that subset may be replaced by another subset needed later in the application's execution. Reconfiguration increases the effective size of an FPGA, thus enabling better performance when more application regions are partitioned to it or, alternatively, enabling use of a smaller and hence cheaper FPGA with a runtime overhead required to swap in new bitstreams. In some cases, this overhead may limit the benefits of reconfiguration and should therefore be considered during partition evaluation.

Figure 26.8(b) illustrates some of the different implementation model parameters, including communication methods, fused regions, and tightly/loosely coupled coprocessors. Often these parameters are fixed prior to partitioning, but can also be explored dynamically during partitioning to determine the best implementation model for a given application and given constraints.

26.2.5 Exploration

Exploration is the searching of the partition solution space for a good partition. As mentioned before, it is at present mostly a manual task, but automated techniques are beginning to mature. This section discusses automated exploration techniques for various formulations of the partitioning problem.

Simple formulation

A simple and common form of the hardware/software partitioning problem consists of n regions, each having a software runtime value, a hardware runtime value, and a hardware size. It assumes that all values are independent of one another (so if two regions are mapped to hardware, their hardware runtime and size values are unchanged); it assumes that communication times are constant regardless of whether a region is implemented as software or hardware (such as when all regions use the same interface to a shared memory); and it seeks to minimize total application runtime subject to a hardware size constraint (assuming no dynamic reconfiguration).

Although this problem is known to be NP-hard, it can be solved by first mapping it to the well-known *0-1 knapsack problem* [20]. The 0-1 knapsack problem involves a knapsack with a specified weight capacity and a set of items, each with a weight and a profit. The goal is to select which items to place in the knapsack such that the total profit is maximized without violating the weight capacity. For hardware/software partitioning, regions correspond to items, the FPGA size constraint corresponds to the knapsack capacity, an implementation's size corresponds to an item's weight, and the speedup obtained by implementing a region in hardware instead of software corresponds to an item's profit.

Thus, algorithms that solve the 0-1 knapsack problem solve the simple form of the hardware/software partitioning problem. The 0-1 knapsack problem is NP-hard, but efficient optimal algorithms exist for relatively large problem sizes. One of these is a well-known dynamic programming algorithm [12] having runtime complexity of $O(A*n)$, where A is the capacity and n is the number of items. Alternatively, integer linear programming (ILP) [22] may be used. ILP solvers perform extensive solution space pruning to reduce exploration time.

For problems too big for either such optimal technique, heuristics may be utilized. A *heuristic* finds a good, but not necessarily the optimal, solution, while an *algorithm* finds the optimal solution. A common heuristic for the 0-1 knapsack problem is a greedy one. A greedy heuristic starts with an initial solution and then makes changes only if they seem to improve the solution. It sorts each item based on the ratio of profit to weight and then traverses the sorted list, placing an item in the knapsack if it fits and skipping it otherwise, terminating when reaching the knapsack capacity or when all items have been considered. This heuristic has $O(nlgn)$ time complexity, allowing for fast automated partitioning of thousands of regions or feasible manual partitioning of tens of regions. Furthermore, the heuristic has been shown to commonly obtain near-optimal results in the situation when a few items have a high profit to weight ratio. In hardware/software partitioning terms, that situation corresponds to the existence of regions that are responsible for the majority of execution time and require little hardware area, which is often the case.

Formulation with asymmetric communication and greedy/nongreedy automated heuristics

A slightly more complex form of the hardware/software partitioning problem considers cases where communication times between regions change depending on the partitioning, with different required times for communication depending on whether the regions are both in software or both in hardware, or are separated, with one in software and one in hardware. This form of the problem can be mapped to the well-known graph bipartitioning problem.

Graph bipartitioning divides a graph into two sets in order to minimize an objective function. Each graph node has two weights, one for each set. Edges may have three different weights: two weights associated with nodes connected in the same set (one weight for each set) and one for nodes connected between sets. Typically, the objective function is to minimize the sum of all node and edge

weights using the appropriate weights for a given partition. Graph bipartitioning is NP-hard.

ILP approaches may be used for automatically obtaining optimal solutions to the graph bipartitioning problem. Heuristics may be used when ILP is too time consuming. A simple greedy heuristic for graph bipartitioning starts with some initial partition, perhaps random or all software. It then determines the cost improvement of moving each node from its present set to the opposite set and then moves the node yielding the best improvement. The heuristic repeats these steps until no move yielding an improvement is found. Given n nodes, a basic form of such a heuristic has $O(n^2)$ runtime complexity. Techniques to update the existing cost improvement values can reduce the complexity to $O(n)$ in practice [25].

More advanced heuristics seek to overcome what are known as "local minima," accepting solution-worsening moves in the hope that they will eventually lead to an even better solution. For example, Figure 26.9 illustrates a heuristic that accepts some solution-worsening changes to escape a local minimum and eventually reach a better solution. A common situation causing a local minimum involves two items such that moving only one item worsens the solution but moving both improves it.

A well-known category of nongreedy heuristic used in partitioning is known as *group migration* [11], which evolved from an initial heuristic by Kernighan–Lin. Like the previous greedy heuristic, group migration starts with an initial partition and determines the cost improvement of moving each node from its present set to the opposite set. The group migration heuristic then moves the node yielding the best improvement (like the greedy heuristic) or yielding the *least worsening* (including zero cost change) if no improving move exists. Accepting such worsening moves enables local minima to be overcome. Of course, such a heuristic would never terminate, so group migration ensures termination by locking a node after it is moved. Group migration moves each node exactly once in what is referred to as an iteration, and an iteration has complexity of $O(n^2)$ (or $O(n)$ if clever techniques are used to update cost improvements after each

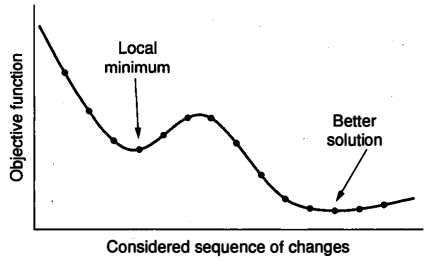


FIGURE 26.9 ■ Solution-worsening moves accepted by a nongreedy heuristic to escape local minima and find better solutions.

move). If an iteration ultimately leads to an improvement, then group migration runs another iteration. In practice, only a few iterations, typically less than five, can be run before no further improvement can be found.

The previous discussions of heuristics ignore the time required by partition evaluation. The heuristics therefore may have even higher runtime complexity unless care is taken to incorporate fast incremental evaluation updates during exploration.

Complex formulations and powerful automated heuristics

Increasingly complex forms of the hardware/software partitioning problem integrate more parameters related to the earlier mentioned issues of exploration—granularity, evaluation, alternative region implementation, and implementation models. For example, the earlier mentioned dynamic granularity modifications, such as decomposing a given region into smaller regions, or even applying transformations to an application such as function inlining, can be applied during partitioning. The partitioning problem can consider different couplings of coprocessors, may also consider coprocessor fusing, and can support dynamic reconfiguration. When one considers the multitude of possible parameters that can be integrated with partitioning, the size of the solution space is mind-boggling. Searching that space for the best solution becomes a tremendous combinatorial optimization challenge, likely requiring long-running search heuristics.

At this point, it may be interesting to note that hardware/software partitioning brings together two previously separate research fields: compilers and CAD (computer-aided design). Compilation techniques tend to emphasize a quick series of *transformations* applied to an application's description. In contrast, CAD techniques tend to emphasize a long-running iterative *search* of enormous solution spaces. One possible reason for these different perspectives is that compilers were generally expected to run quickly, in seconds or at most minutes, because they were part of a design loop in which compilation was applied perhaps dozens or hundreds of times a day as programs were developed. In contrast, CAD optimization techniques were part of a much longer design loop. Running CAD optimization tools for hours or even days was perfectly acceptable because that time was still small compared to the weeks or months required to manufacture chips. Furthermore, the very nature of coprocessor design meant that a designer was extremely interested in high performance, so longer tool runtimes were acceptable if they optimized an implementation.

Hardware/software partitioning merges compilation and synthesis into a single framework. In some cases, compiler-like runtimes of seconds must be achieved. In other cases, CAD-like runtimes of hours may be acceptable. Approaches to partitioning may span that range. Highly complex partitioning formulations will likely require moving away from the fast linear time algorithms and heuristics described earlier and toward longer-running powerful search heuristics.

A popular powerful and general search heuristic is *simulated annealing* [17]. The simulated annealing heuristic starts with a random solution and then randomly makes some change to it, perhaps moving a region between software

and hardware, choosing an alternative implementation for a particular region, decomposing a particular region into finer-grained regions, performing a transformation on the original regions, and so forth, and evaluates the cost (as determined by an objective function) of the new partition obtained from that change. If the change improves the cost, it is accepted (i.e., the change is made). If the change worsens the cost, the seemingly "bad" change is accepted with some probability. The key feature of simulated annealing is that the probability of accepting a seemingly bad move decreases as the approach proceeds, with the pattern of decrease determined by some parameters provided to the annealing process that eventually causes it to narrow in on a good solution. Simulated annealing typically must evaluate many thousands or millions of solutions in order to arrive at a good one and thus requires very fast evaluation methods.

The complexity of simulated annealing is generally dependent on the problem instance. With properly set parameters, it can achieve near-optimal solutions on very large problems in long but acceptable runtimes. Faster machines have made simulated annealing an increasingly acceptable search heuristic for a wider variety of problems—it can complete in just seconds for many problem instances.

The simulated annealing heuristic is known as a neighborhood search heuristic because it makes local changes to an existing solution. Tabu search [13] is an effective method for improving neighborhood search. Meaning "forbidden," Tabu maintains a list of recently seen, Tabu, solutions. When considering a change to an existing solution, it disregards any change that would yield a solution on the Tabu list. This prevents cycling among the same solutions and has been shown to yield improved results in less time. The Tabu list concept can also be applied on a broader scale, maintaining a long-term history of considered solutions in order to increase solution diversity. Tabu search can improve neighborhood search heuristic runtimes during hardware/software partitioning by a factor of 20x [8].

Other issues

Because implementing an application as software generally requires a smaller size and less designer effort, most approaches to exploration start with an all-software implementation and then explore the mapping of critical application regions to hardware. However, in some cases, such as when the application is written specifically for hardware, an approach may start with an all-hardware implementation and then move noncritical application regions to software to reduce hardware size.

Furthermore, when an application is originally written for software implementation, some of its regions may not be suitable for hardware implementation. For example, application regions that utilize recursive function calls, pointer-based data structures, or dynamic memory allocation may not be easy to implement as a hardware circuit. Some research efforts are beginning to address these problems by developing new synthesis techniques that support a wider range of program constructs and behavior. Alternatively, designers sometimes

write (or rewrite) critical regions such that those regions are well suited for circuit implementation.

26.3 PARTITIONING OF PARALLEL PROGRAMS

In parallel programs, the regions that make up an application are defined to execute concurrently, as opposed to sequentially. Such regions are often called *tasks* or *processes*. For some applications, expressing behavior using tasks may result in a more parallel implementation and hence in faster application performance. For example, an MPEG2 decoder may be described as several tasks, such as motion compensation, dequantization, or inverse discrete cosine transform, that can be implemented in a pipelined manner.

Numerous parallel programming models have been considered for hardware/software partitioning, among others, synchronous dataflow, dynamic dataflow, Kahn process networks, and communicating sequential processes.

26.3.1 Differences among Parallel Programming Models

While hardware/software partitioning of parallel programs has many similarities to partitioning for sequential programs, several key differences exist.

Granularity
Partitioning of parallel programs typically treats each task as a region, meaning that the granularity is quite coarse. In some cases, decomposing a task into finer granularity may be considered.

Evaluation
Parallel programs often involve multiple performance constraints, with particular tasks or sets of tasks having unique performance constraints of their own. Furthermore, estimations of performance must consider the scheduling of tasks on processors, which is not an issue for sequential programs because regions in these programs are not concurrent.

Alternative region implementations
Given the coarse granularity of tasks, considering alternative implementations becomes even more important, as the variations among the alternatives can be huge.

Implementation models
Because tasks are inherently concurrent, partitioning of parallel programs typically uses parallel execution models in their implementations, meaning that microprocessors and coprocessors run concurrently rather than mutually exclusively and meaning that coprocessors may be arranged to form high-level pipelines. Partitioning of parallel programs is less likely to consider fusing multiple coprocessors into one because fusing eliminates concurrency.

Parallel program partitioning introduces a new aspect to exploration—scheduling. When mapping multiple tasks to a single microprocessor, partitioning must carry out the additional step of scheduling to determine when each task will execute. Scheduling tasks to meet performance constraints is known as *real-time scheduling* and is a heavily studied problem [3].

Including partitioning during scheduling results in a more complex problem. Such partitioning often considers more than just one microprocessor as well and even different types of microprocessors. It may even consider different numbers and types of memories and different bus structures connecting memories to processors.

Parallel partitioning must also pay more attention to the data storage requirements between processors. Queues may be introduced between processors, the sizes of those queues must be determined, and their implementation (e.g., in shared memory or in separate hardware components) must be decided.

Exploration

More complex issues in the hardware/software partitioning problem—such as scheduling, different granularities, different evaluation methods, alternative region implementations, and different numbers and connections of microprocessors/memories/buses—require more complex solution approaches. Most modern automatic partitioning research considers one or a few extensions to basic hardware/software partitioning and develops custom heuristics to solve the new formulations in fast compiler-like runtimes. However, as more complex forms of partitioning are considered, more powerful search heuristics with longer runtimes, such as simulated annealing or search algorithms tuned to the problem formulation, may be necessary.

26.4 SUMMARY AND DIRECTIONS

Developing an approach for hardware/software partitioning requires the consideration of granularity, evaluation, alternative region implementations, implementation models, exploration, and so forth, and each such issue involves numerous options. The result is a tremendously large partition solution space and a huge variety of approaches to finding good partitions. While much research into automated hardware/software partitioning has occurred over the past decades, most of the problem's more complex formulations have yet to be considered. A key future challenge will be the development of effective partitioning approaches for these increasingly complex formulations.

As FPGAs continue to enter mainstream embedded, desktop, and server computing, incorporating automated hardware/software partitioning into standard software design flows becomes increasingly important. One approach to minimizing the disruption of standard software design flows is to incorporate partitioning as a backend tool that operates on a final binary, allowing continued use of existing programming languages and compilers and supporting the use

of assembly and even object code. Such binary-level partitioning [23] requires powerful decompilation methods to recover high-level regions such as functions and loops. Binary-level partitioning even opens the door for dynamic partitioning, wherein on-chip tools transparently move software regions to FPGA coprocessors, making use of new lean, just-in-time compilers for FPGAs [19].

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967.

[2] J. Burns, A. Donlin, J. Hogg, S. Singh, M. De Wit. A dynamic reconfiguration runtime system. *Proceedings of the Symposium on FPGA-Based Custom Computing Machines*, 1997.

[3] G. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic, 1997.

[4] S. Chappell, C. Sullivan. Handel-C for co-processing and co-design of field programmable system on chip. *Proceedings of Workshop on Reconfigurable Computing and Applications*, 2002.

[5] K. Chatha, R. Vemuri. An iterative algorithm for partitioning, hardware design space exploration and scheduling of hardware-software systems. *Design Automation for Embedded Systems* 5(3–4), 2000.

[6] K. Compton, S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys* 34(2), 2002.

[7] CriticalBlue. *http://www.criticalblue.com.*

[8] P. Eles, Z. Peng, K. Kuchchinski, A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems* 2(1), 1997.

[9] R. Enzler, T. Jeger, D. Cottet, G. Tröster. High-level area and performance estimation of hardware building blocks on FPGAs. *Lecture Notes in Computer Science* 1896, 2000.

[10] R. Ernst, J. Henkel. Hardware-software codesign of embedded controllers based on hardware extraction. *Proceedings of the International Workshop on Hardware/Software Codesign*, 1992.

[11] D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems*, Prentice-Hall, 1994.

[12] P. C Gilmore, R. E Gomory. The theory and computation of knapsack functions. *Operations Research* 14, 1966.

[13] F. Glover. Tabu search, part I. *Operations Research Society of America Journal on Computing* 1, 1989.

[14] T. Grotker, S. Liao, G. Martin, S. Swan. *System Design with System C.* Springer-Verlag, 2002.

[15] R. Gupta, G. De Micheli. System-level synthesis using re-programmable components. *Proceedings of the European Design Automation Conference*, 1992.

[16] J. Henkel, R. Ernst. A hardware/software partitioner using a dynamically determined granularity. *Design Automation Conference*, 1997.

[17] S. Kirkpatrick, C. Gelatt, M. Vecchi. Optimization by simulated annealing. *Science* 220(4598), May 1983.

[18] Y. Li, J. Henkel. A framework for estimation and minimizing energy dissipation of embedded HW/SW systems. *Design Automation Conference*, 1998.
[19] R. Lysecky, G. Stitt, F. Vahid. Warp processors. *Transactions on Design Automation of Electronic Systems* 11(3), 2006.
[20] S. Martello, P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*, Wiley, 1990.
[21] Poseidon Design Systems, Inc. *http://www.poseidon-systems.com/index.htm.*
[22] A. Schrijver. *Theory of Linear and Integer Programming*, Wiley, 1998.
[23] G. Stitt, F. Vahid. New decompilation techniques for binary-level co-processor generation. *Proceedings of the International Conference on Computer-Aided Design*, 2005.
[24] K. Suzuki, A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. *Design Automation Conference*, 1996.
[25] F. Vahid, D. Gajski. Incremental hardware estimation during hardware/software functional partitioning. *IEEE Transactions on VLSI Systems* 3(3), 1995.
[26] F. Vahid, D. Gajski. Specification partitioning for system design. *Design Automation Conference*, 1992.
[27] XPRES Compiler. *http://www.tensilica.com/products/xpres.htm.*

PART V

CASE STUDIES OF FPGA APPLICATIONS

Parts I through IV covered technologies and techniques for creating efficient FPGA-based solutions to important problems. Part V focuses on specific, important field-programmable gate array (FPGA) applications, presenting case studies of interesting uses of reconfigurable technology. While this is by no means an exhaustive survey of all applications done on FPGAs, these chapters do contain several very interesting representative points in this space. They can be read in any order, and can even be interspersed with other chapters of this book.

This introduction should help readers identify the concepts the case studies cover and the chapters each help to illustrate. To understand the case studies, a basic knowledge of FPGAs (Chapter 1), CAD tools (Chapters 6, 13, 14, and 17), and application development (Chapter 21) is required.

Chapter 27 presents a high-performance image compression engine optimized for satellite imagery. This is a streaming signal-processing application (Chapters 5, 8, and 9), a type of computation that typically maps well to reconfigurable devices. In this case, the system saw speedups of approximately 400 times, for which the authors had to optimize the algorithm carefully, considering memory bandwidth (Chapter 21), conversion to fixed point (Chapter 23), and alteration of the algorithm to eliminate sequential dependencies.

Chapter 28 focuses on automatic target recognition, which is the detection of regions of interest in military synthetic aperture radar (SAR) images. Like the compression engine in Chapter 27, this represents a very complex, streaming signal-processing application. It also is one of the most influential applications of runtime-reconfiguration (Chapters 4 and 21), where a large circuit is time-multiplexed onto a single FPGA, enabling it to reuse the same silicon multiple times. This was necessary because the possible targets to be detected were represented by individual custom, instance-specific circuits (Chapter 22), the huge number of which was too large for the available FPGAs.

Chapter 29 discusses Boolean satisfiability (SAT) solving—the determination of whether there is an assignment of values to variables that

makes a given Boolean equation true (satisfied). SAT is a fairly general optimization technique that is useful in, for example, chip testing, formal verification, and even FPGA CAD flows. This work on solving Boolean equations via FPGAs is an interesting application of instance-specific circuitry (Chapter 3) because each equation to be solved was compiled directly into FPGA logic. However, this meant that the runtime of the CAD tools was part of the time needed to solve a given Boolean equation, creating a strong push toward faster CAD algorithms for FPGAs (Chapter 20).

Chapter 30 covers logic emulation—the prototyping of complex integrated circuits on huge boxes filled with FPGAs and programmable interconnect chips. This is one of the most successful applications of multi-FPGA systems (Chapter 3) because the translation of a single ASIC into FPGA logic necessitates hundreds to thousands of FPGAs to provide adequate logic capacity. Fast mapping tools for such systems are also important (Chapter 20).

In Chapter 23 we discussed methods for eliminating (or at least minimizing) the amount of floating-point computation in FPGA designs by converting floating-point operations to fixed point. However, there are situations where floating point is unavoidable. Scientific computing codes often depend on floating-point values, and many users require that the FPGA-based implementation provide *exactly* the same results as those of a processor-based solution. These situations require full floating-point support. In other cases, the high dynamic range of values might make fixed-point computations untenable. Chapter 31 considers the development of a library of floating-point units and their use in applications such as FFTs.

Chapter 32 covers a complex physical simulation application—the finite difference time domain (FDTD) method, which is a way of modeling electromagnetic signals in complex situations that can be very useful in applications such as antenna design and breast cancer detection. The solution involves a large-scale cellular automata (Chapter 5) representation of the space to be modeled and an iterative solver. The key to achieving a high-performance implementation on FPGAs, however, involves conversion to fixed-point arithmetic (Chapter 23), simplification of complex mathematical equations, and careful consideration of the memory bottlenecks in the system (Chapter 21).

Chapter 33 discusses an alternative to traditional design flow for creating FPGA mappings in which the FPGA is allowed to evolve its own configuration. Because the FPGA is reprogrammable, a genetic optimization system can simply load into it random configurations and see how well they function. Those that show promise are retained; those that do

not are removed. Through mutation and breeding, new configurations are created and evaluated in the same way, slowly evolving better and better computations. The hope is that such a system can support important classes of computation with circuits significantly more efficient than standard design flows. This design strategy exploits special features of the FPGA's reprogrammability and flexibility (Chapter 4).

Some of the chapters in this section focus on streaming digital signal processing (DSP) applications. Such applications often benefit from FPGA logic because of their amenability to pipelining and because of the large amount of data parallelism inherent in the computation. Network processing and routing is another such application domain. Chapter 34 considers packet processing, the application of FPGA logic to network filtering, and related tasks. Heavy pipelining of circuits onto the reconfigurable fabric and optimization of custom boards to network processing (Chapter 3) support very high-bandwidth networking. However, because the system retains the flexibility of FPGA logic, new computations and new filtering techniques can be easily accommodated within the system. This ability to incrementally adjust, tune, and invent new circuits provides a valuable capability even in a field as rapidly evolving as network security.

For many applications, memory access to a large set of state, rather than computational, throughput can be the bottleneck. Chapter 35 explores an object-oriented, data-centric model (Chapter 5) based on adding programmable or reprogrammable logic into DRAM memories. The chapter emphasizes custom-reprogrammable chips (Chapter 2) and explores both FPGA and VLIW implementation for the programmable logic. Nevertheless, much of the analysis and techniques employed can also be applied to modern FPGAs with large, on-chip memories.

CHAPTER 27

SPIHT IMAGE COMPRESSION

Thomas W. Fry
Samsung, Global Strategy Group
Scott Hauck
Department of Electrical Engineering
University of Washington

This chapter describes the process of mapping the image compression algorithm SPIHT onto a reconfigurable logic architecture. A discussion of why adaptive logic is required, as opposed to an ASIC, is provided, along with background material on SPIHT. Several discrete wavelet transform hardware architectures are analyzed and evaluated. In addition, two major modifications to the original image compression algorithm, which are required in order to build a reconfigurable hardware implementation, are presented: (1) the storage elements necessary for each wavelet coefficient, and (2) a modification to the original SPIHT algorithm created to parallelize the computation. Also discussed are the effects these modifications have on the final compression results and the trade-offs involved.

The chapter then describes how the updated SPIHT algorithm is mapped onto the Annapolis Microsystems WildStar reconfigurable hardware system. This system is populated with three Virtex-E field-programmable gate array (FPGA) parts and several memory ports. The issues of how the modified algorithm is divided between individual FPGA parts and how data flows through the memories are discussed. Lastly, final results and speedups are presented and evaluated against a comparable microprocessor solution from the time the Annapolis Microsystems WildStar was released.

27.1 BACKGROUND

As NASA deploys each new generation of satellites with more sensors, capturing an ever-larger number of spectral bands, the volume of data being collected begins to outstrip a satellite's ability to transmit data back to Earth. For example, the Terra satellite contains five separate sensors, each collecting up to 36 individual spectral bands. The Tracking and Data Relay Satellite System (TDRSS) ground terminal in White Sands, New Mexico, captures data from these sensors at a limited rate of 150 Mbps [19]. As the number of sensors on a satellite grows and the transmission rates increase, this bandwidth limitation became a driving force for NASA to study methods of compressing images prior to downlinking.

FPGAs are an attractive implementation medium for such a system. Software solutions suffer from performance limitations and power requirements. At the same time, traditional hardware platforms lack the required flexibility needed for postlaunch modifications. After launch, such fixed hardware systems cannot be modified to use newer compression schemes or even to implement bug fixes. In the past, modification of fixed systems in satellites proved to be very expensive [4].

By implementing an image compression kernel in a reconfigurable system, we overcame these shortcomings. Because such a system may be reprogrammed after launch, it does not suffer from conventional hardware's inherit inflexibility. At the same time, the algorithm is computing in custom hardware and can perform at the required processing rates while consuming less power than a traditional software implementation.

This chapter describes the work performed as part of a NASA-sponsored investigation into the design and implementation of a space-bound FPGA-based hyperspectral image compression machine. For this work, the Set Partitioning in Hierarchical Trees (SPIHT) routine was selected as the image compression algorithm. First, we describe the algorithm and discuss the reasons for its selection. Then we describe how the algorithm was optimized for implementation in a specific hardware platform and we present the results.

27.2 SPIHT ALGORITHM

SPIHT is a wavelet-based image compression coder. It first converts an image into its wavelet transform and then transmits information about the wavelet coefficients. The decoder uses the received signal to reconstruct the wavelet and then performs an inverse transform to recover the image. SPIHT was selected because both it and its predecessor, the embedded zerotree wavelet coder, were significant breakthroughs in still-image compression. Both offered significantly improved quality over other image compression techniques such as vector quantization, JPEG, and wavelets combined with quantization, while not requiring training that would have been more difficult to implement in hardware. In short, SPIHT displays exceptional characteristics over several properties all at once [15]:

- Good image quality with a high peak-signal-to-noise ratio (PSNR).
- Fast coding and decoding.
- A fully progressive bitstream.
- Can be used for lossless compression.
- May be combined with error protection (useful in satellite transmissions).
- Ability to code for an exact bitrate or PSNR.

In addition, since the SPIHT algorithm processes an image in two distinct steps—the discrete wavelet transform phase and the coding phase—it provides a natural point at which a hardware implementation may be divided. (The advantage of this property will be seen in Section 27.4.) The rest of this section

describes the basics of wavelets, the discrete wavelet transform, and the SPIHT coding engine.

27.2.1 Wavelets and the Discrete Wavelet Transform

The wavelet transform is a reversible transform on spatial data. The discrete wavelet transform (DWT) is a form appropriate to discrete data, such as the individual points or pixels in an image. DWT runs a high-pass and low-pass filter over the signal in one dimension. This produces a low-pass ("average") version of the data and a high-pass (rapid changes within the average) version. Every other result from each pass is then sampled, yielding two subbands, each of which is one-half the size of the input stream. The result is a new image comprising of a high- and a low-pass subband. These two subbands can be used to fully recover the original image. In the case of a multidimensional signal such as an image, this procedure is repeated in each dimension (Figure 27.1).

The vertical and horizontal transformations break up the image into four distinct subbands. The wavelet coefficients that correspond to the fine details are the LH, HL, and HH subbands. Lower frequencies are represented by the LL subband, which is a low-pass filtered version of the original image [17].

The next wavelet level is calculated by repeating the horizontal and vertical transformations on the LL subband from the previous level. Four new subbands are created from the transformations. The LH, HL, and HH subbands in the next level represent coarser-scale coefficients and the new LL subband is an even smoother version of the original image. It is possible to obtain coarser and coarser scales of the LH, HL, and HH subbands by iteratively repeating the wavelet transformation on the LL subband of each level. Figure 27.2 displays the subband components of an image with three scales of wavelet transformation.

The reverse transformation uses an inverse filter on the final LL subband and the LH, HL, and HH subbands at the same level to recreate the LL subband of the previous level. By iteratively processing each level, the original image may be restored. Figure 27.3 displays a satellite image of San Francisco and its corresponding 3-level DWT. By processing either the wavelet transform or the inverse wavelet transform, these two images may be converted from one into the other and thus may be viewed as equivalent.

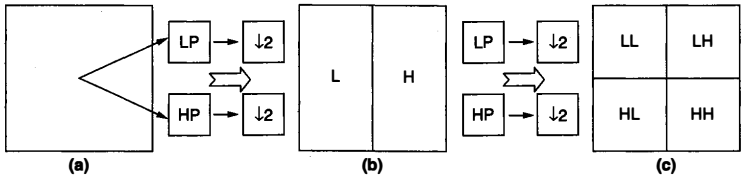


FIGURE 27.1 ■ A 1-level wavelet built by two one-dimensional passes: (a) original image, (b) horizontal pass, and (c) vertical pass.

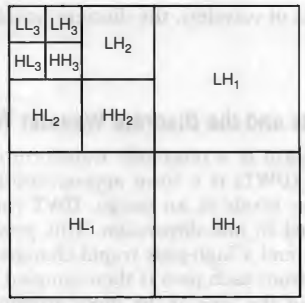


FIGURE 27.2 ■ A 3-level wavelet transform.

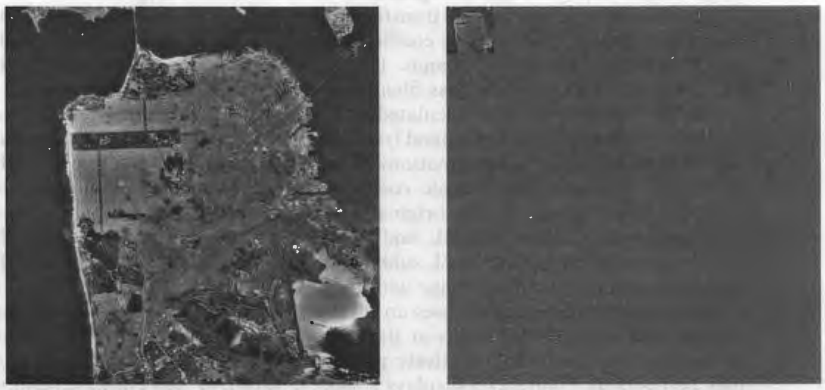


FIGURE 27.3 ■ An image of San Francisco (a) and the resulting 3-level DWT (b).

27.2.2 SPIHT Coding Engine

SPIHT is a method of coding and decoding the wavelet transform of an image. As discussed in the previous section, by coding and transmitting information about the wavelet coefficients, it is possible for a decoder to perform an inverse transformation on the wavelet and reconstruct the original image. A useful property of SPIHT is that the entire wavelet does not need to be transmitted in order to recover the image. Instead, as the decoder receives more information about the original wavelet transform, the inverse transformation yields a better-quality reconstruction (i.e., a higher PSNR) of the original image. SPIHT generates excellent image quality and performance due to three properties of the coding algorithm: partial ordering by coefficient value, taking advantage

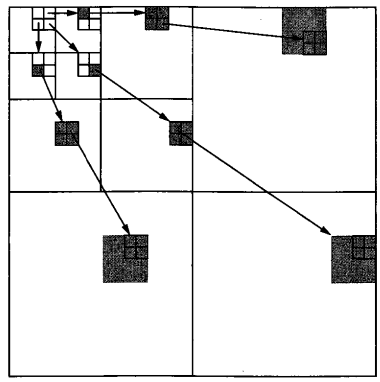


FIGURE 27.4 ■ Spatial orientation trees.

of the redundancies between different wavelet scales, and transmitting data in bit-plane order [14].

Following a wavelet transformation, SPIHT divides the wavelet into *spatial orientation trees* (Figure 27.4). Each node in a tree corresponds to an individual pixel. The offspring of a pixel are the four pixels in the same spatial location of the same subband at the next finer scale of the wavelet. Pixels at the finest scale of the wavelet are the leaves of the tree and have no children. Every pixel is part of a 2×2 block with its adjacent pixels. Blocks are a natural result of the hierarchical trees because every pixel in a block shares the same parent pixel. Also, the upper-left pixel of each 2×2 block at the root of the tree has no children since there are only three subbands at each scale and not four. Figure 27.4 shows how the pyramid is defined. Arrows point to the offspring of an individual pixel and the grayed blocks show all of the descendents for a specific pixel at every scale.

SPIHT codes a wavelet by transmitting information about the significance of a pixel. By stating whether or not a pixel is above some threshold, information about that pixel's value is implied. Furthermore, SPIHT transmits information stating whether a pixel or any of its descendents are above a threshold. If the statement proves false, all of the pixel's descendants are known to be below that threshold level and they do not need to be considered during the rest of the current pass. At the end of each pass, the threshold is divided by two and the algorithm continues. In this manner, information about the most significant bits of the wavelet coefficients will always precede information on lower-order significant bits, which is referred to as *bit-plane ordering*.

Information stating whether or not a pixel is above the current threshold or is being processed at the current threshold is contained in three lists: the *list of insignificant pixels* (LIP), the *list of insignificant sets* (LIS) and the *list of significant pixels* (LSP). The LIP are pixels that are currently being processed

but are not yet above the threshold. The LIS are pixels that are currently being processed but none of their descendents are yet above the current threshold and so they are not being processed. Lastly, the LSP are pixels that were already stated to be above a previous threshold level and whose value at each bit plane is now transmitted.

Figure 27.5 is the algorithm from the original SPIHT paper [14], modified to reflect changes (discussed later in the chapter) referring to 2×2 block information. $S_n(i, j)$ represents if the pixel (i, j) is greater than the current threshold, and $S_n(D(i, j))$ states if any of the pixel's (i, j) descendents are greater than the current threshold.

There are three important concepts to take from the SPIHT algorithm. First, as the encoder sequentially steps through the image, it inserts or deletes pixels from the three lists. All of the information required to keep track of the lists is output to the decoder, allowing the decoder to generate and maintain an identical list order as the encoder. For the decoder to reproduce the steps taken by the encoder we merely need to replace the `output` statements in the encoder's algorithm with `input` for the decoder's algorithm.

Second, the bitstream produced is naturally progressive. A progressive bitstream is one that can be cut off at any point and still be valid. As the decoder steps through the coding algorithm, it gathers finer and finer detail about the original wavelet transform. The decoder can stop at any point and perform an inverse transform with the wavelet coefficients it has currently reconstructed. Progressive bitstreams can also be reduced to an arbitrary size or be cut off during transmission and still produce a valid image. Such a property is very useful in satellite transmissions.

```
1. Initialization: output n = floor[log2(max(i,j){|ci,j|})]; clear the LSP list,
   add the root pixels to the LIP list and root pixels with descendants to LIS.
2. Sorting Pass:
   2.1 for each entry (i,j) in the LIP:
       2.1.1 output Sn(i,j);
       2.1.2 If Sn(i,j) = 1, move (i,j) to the LSP list and output its sign
   2.2 for each entry (i,j) in the LIS:
       2.2.1 If one of the pixels in (i,j)'s block is not in LIP but all are
             in LIS:
                output Sn(all descendants of the current block);
                if none are significant, skip 2.2.2.
       2.2.2 Output Sn(D(i,j))
              if Sn(D(i,j)) = 1, then
                for each of (i,j) immediate children (k,l):
                  output Sn(k,l);
                  add (k,l) to the LIS for the current pass
                  if Sn(k,l) = 1, add (k,l) to the LSP and output its sign
                                  else add (k,l) to the LIP
3. Refinement Pass: for each entry (i,j) in LSP, except ones inserted in the
   current pass, output the n^th most significant bit of (i,j).
4. Quantization-step Update: decrement n by 1 and go to Step 2.
```

FIGURE 27.5 ■ SPIHT coding algorithm.

Third, and the concept that has the largest impact on building a hardware platform, the SPIHT algorithm develops an individual list order to transmit information within each bit plane. This ordering is implicitly created from the threshold information discussed before—the order in which each pixel enters each list determines the transmission order for each image. As a result, each image will transmit wavelet coefficients in an entirely different order. Slightly better PSNRs are achieved with this dynamic ordering of the wavelet coefficients.

The SPIHT algorithm in Figure 27.5, which creates the individual list ordering, is inherently sequential. As a result, SPIHT cannot be significantly parallelized in hardware. This drawback greatly limits the performance of any SPIHT implementation in hardware. To get around this limitation and improve performance, it was necessary to parallelize the SPIHT algorithm and essentially create a new image compression algorithm. These changes and the trade-offs involved are described in Section 27.3.3.

27.3 DESIGN CONSIDERATIONS AND MODIFICATIONS

To fully take advantage of the high performance a custom hardware implementation of SPIHT could yield, the software specifications had to be examined and adjusted where they either performed poorly in hardware or did not make the most of the resources available. Here we review the three major factors taken under consideration while evaluating how to create a hardware implementation of the SPIHT algorithm on an adaptive computing platform.

The first factor was to determine what discrete wavelet transform architecture to use. Section 27.3.1 provides a summary of the DWTs considered, showing how memory and communication requirements helped dictate the structure chosen. Section 27.3.2 describes the fixed-point precision optimization performed for each wavelet coefficient and the final data representation employed. Section 27.3.3 explains how the SPIHT algorithm was altered to vastly speed up the hardware implementation.

27.3.1 Discrete Wavelet Transform Architectures

One of the benefits of the SPIHT algorithm is its use of the discrete wavelet transform, which had existed for several years prior to this work. As a result, numerous studies on how to create a DWT hardware implementation were available for review. Much of this work on DWTs involved parallel platforms to save both memory access and computations [5, 12, 16].

The most basic architecture is the basic folded architecture. The one-dimensional DWT entails demanding computations, which involve significant hardware resources. Since the horizontal and vertical passes use identical finite impulse response (FIR) filters, most two-dimensional DWT architectures implement folding to reuse logic for each dimension [6]. Figure 27.6 illustrates how folded architectures use a one-dimensional DWT to realize a two-dimensional DWT.

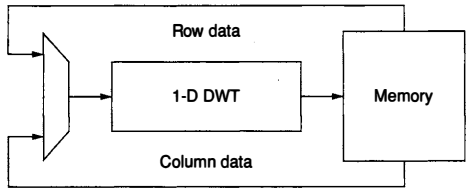


FIGURE 27.6 ■ A folded architecture.

Although the folded architecture saves hardware resources, it suffers from high memory bandwidth. For an $N \times N$ image there are at least $2N^2$ read-and-write cycles for the first wavelet level. Additional levels require rereading previously computed coefficients, further reducing efficiency.

To lower the memory bandwidth requirements needed to compute the DWT, we considered several alternative architectures. The first was the Recursive Pyramid Algorithm (RPA) [21]. RPA takes advantage of the fact that the various wavelet levels run at different clock rates. Each wavelet level requires one-quarter of the time that the previous level needed because at each level the size of the area under computation is reduced by one-half in both the horizontal and vertical dimensions. Thus, it is possible to store previously computed coefficients on-chip and intermix the next level's computations with the current level's. A careful analysis of the runtime yields $(4*N^2)/3$ individual memory load and store operations for an image. However, the algorithm has huge on-chip memory requirements and demands a thorough scheduling process to interleave the various wavelet levels.

Another method to reduce memory accesses is the partitioned DWT, which breaks the image into smaller blocks and computes several scales of the DWT at once for each block [13]. In addition, the algorithm made use of wavelet lifting to reduce the DWT's computational complexity [18]. By partitioning an image into smaller blocks, the amount of on-chip memory storage required was significantly reduced because only the coefficients in the block needed to be stored. This approach was similar to the RPA, except that it computed over sections of the image at a time instead of the entire image at once. Figure 27.7, from Ritter and Molitor [13], illustrates how the partitioned wavelet was constructed.

Unfortunately, the partitioned approach suffers from blocking artifacts along the partition boundaries if the boundaries were treated with reflection.[1] Thus, pixels from neighboring partitions were required to smooth out these boundaries. The number of wavelet levels determined how many pixels beyond a subimage's boundary were needed, since higher wavelet levels represent data

[1] An FIR filter generally computes over several pixels at once and generates a result for the middle pixel. To calculate pixels close to an image's edge, data points are required beyond the edge of the image. Reflection is a method that takes pixels toward the image's edge and copies them beyond the edge of the actual image for calculation purposes.

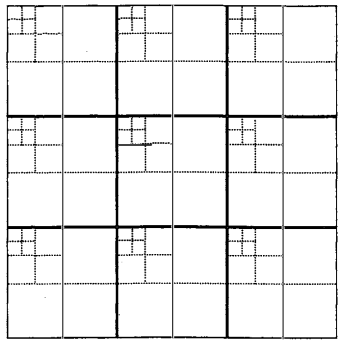


FIGURE 27.7 ■ The partitioned DWT.

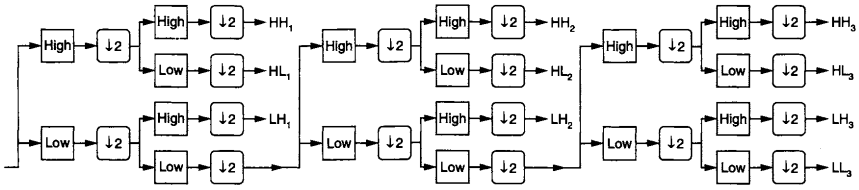


FIGURE 27.8 ■ A generic 2D biorthogonal DWT.

from a larger image region. To compensate for the partition boundaries, the algorithm processed subimages along a single row to eliminate multiple reads in the horizontal direction. Overall data throughputs of up to 152 Mbytes/second were reported with the partitioned DWT.

The last architecture we considered was the generic 2D biorthogonal DWT [3]. Unlike previous designs, the generic 2D biorthogonal DWT did not require FIR filter folding or on-chip memories as the Recursive Pyramid design. Nor did it involve partitioning an image into subimages. Instead, the architecture created separate structures to calculate each wavelet level as data were presented to it, as shown in Figure 27.8. The design sequentially read in the image and computed the four DWT subbands. As the LL_1 subband became available, the coefficients were passed to the next stage, which calculated the next coarser level subbands, and so on.

For larger images that required several individual wavelet scales, the generic 2D biorthogonal DWT architecture consumed a tremendous amount of on-chip resources. With SPIHT, a 1024×1024 pixel image computes seven separate wavelet scales. The proposed architecture would employ 21 individual high- and low-pass FIR filters. Since each wavelet scale processed data at different rates, some control complexity would be inevitable. The advantage of the architecture

was much lower on-chip memory requirements and full utilization of the memory's bandwidth, since each pixel was read and written only once.

To select a DWT, each of the architectures discussed before were reevaluated against our target hardware platform (discussed below). The parallel versions of the DWT saved some memory bandwidth. However, additional resources and more complex scheduling algorithms became necessary. In addition, some of the savings were minimal since each higher wavelet level is one-quarter the size of the previous wavelet level. In a 7-level DWT, the highest 4 levels compute in just 2 percent of the time it takes to compute the first level. Other factors considered were that the more complex DWT architectures simply required more resources than a single Xilinx Virtex 2000E FPGA (our target device) could accommodate, and that enough memory ports were available in our board to read and write four coefficients at a time in parallel.

For these reasons, we did not select a more complex parallel DWT architecture, but instead designed a simple folded architecture that processes one dimension of a single wavelet level at a time. In the architecture created, pixels are read in horizontally from one memory port and written directly to a second memory port. In addition, pixels are written to memory in columns, inverting the image along the 45-degree line. By utilizing the same addressing logic, pixels are again read in horizontally and written vertically. However, since the image was inverted along its diagonal, the second pass will calculate the vertical dimension of the wavelet and restore the image to its original orientation.

Each dimension of the image is reduced by half, and the process iteratively continues for each wavelet level. Finally, the mean of the LL subband is calculated and subtracted from itself. To speed up the DWT, the design reads and writes four rows at a time. Figure 27.9 illustrates the architecture of the DWT phase.

Since every pixel is read and written once and the design processes four rows at a time, for an $N \times N$-size image both dimensions in the lowest wavelet level compute in $2*N^2/4$ clock cycles. Similarly, the next wavelet level processes the image in one-quarter the number of clock cycles as the previous level. With an infinite number of wavelet levels, the image processes in:

$$\sum_{l=1}^{\infty} \frac{2 \cdot N^2}{4^l} = \frac{3}{4} \cdot N^2 \tag{27.1}$$

Thus, the runtime of the DWT engine is bounded by three-quarters of a clock cycle per pixel in the image. This was made possible because the memory ports in the system allowed four pixels to be read and written in a single clock cycle.

It is very important to note that many of the parallel architectures designed to process multiple wavelet levels simultaneously run in more than one clock cycle per image. Also, because of the additional resources required by a parallel implementation, computing multiple rows at once becomes impractical. Given more resources, the parallel architectures discussed previously could process multiple rows at once and yield runtimes lower than three-quarters of a clock cycle per pixel. However, the FPGAs available in the system used, although state of the art at the time, did not have such extensive resources.

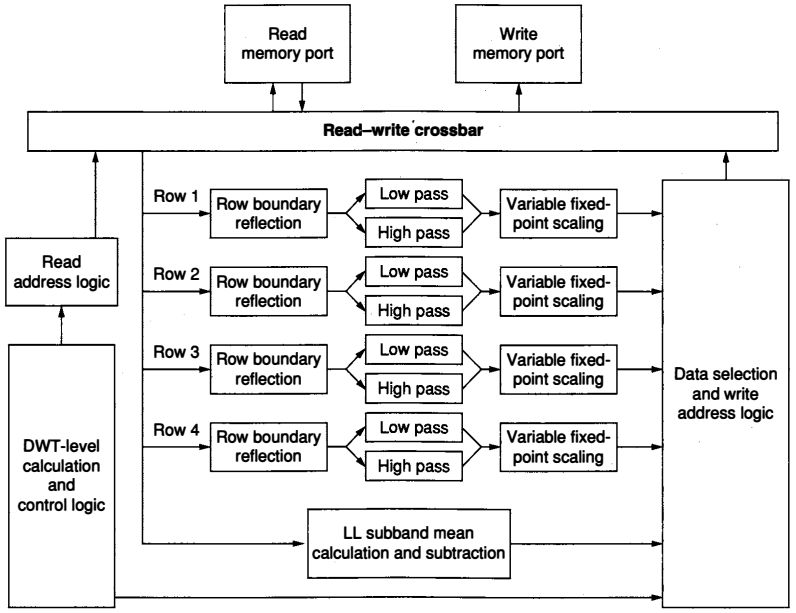


FIGURE 27.9 ■ A discrete wavelet transform architecture.

By keeping the address and control logic simple, there were enough resources on the FPGA to implement 8 distributed arithmetic FIR filters [23] from the Xilinx Core library. The FIR filters required significant FPGA resources, approximately 8 percent of the Virtex 2000E FPGA for each high- and low-pass FIR filter. We chose the distributed arithmetic FIR filters because they calculate a new coefficient every clock cycle, and this contributed to the system being able to process an image in three-quarters of a clock cycle per pixel.

27.3.2 Fixed-point Precision Analysis

The next major consideration was how to represent the wavelet coefficients in hardware. The discrete wavelet transform produces real numbers as the wavelet coefficients, which general-purpose computers realize as floating-point numbers. Traditionally, FPGAs have not employed floating-point numbers for several reasons:

- Floating-point numbers require variable shifts based on the exponential description, and variable shifters perform poorly in FPGAs.

- Floating-point numbers consume enormous hardware resources on a limited-resource FPGA.
- Floating point is often unnecessary for a known dataset.

At each wavelet level of the DWT, coefficients have a fixed range. Therefore, we opted for a fixed-point numerical representation—that is, one where the decimal point's position is predefined. With the decimal point locked at a specific location, each bit contributes a known value to the number, which eliminates the need for variable shifters. However, the DWT's filter bank was unbounded, meaning that the range of possible numbers increases with each additional wavelet level.

We chose to use the FIR filter set from the original SPIHT implementation. An analysis of the coefficients of each filter bank showed that the two-dimensional low-pass FIR filter at most increases the range of possible numbers by a factor of 2.9054. This number is the increase found from both the horizontal and the vertical directions. It represents how much larger a coefficient at the next wavelet level could be if the previous level's input wavelet coefficients were the maximum possible value and the correct sign to create the largest possible filter output. As a result, the coefficients at various wavelet levels require a variable number of bits above the decimal point to cover their possible ranges.

Table 27.1 illustrates the various requirements placed on a numerical representation for each wavelet level. The Factor and Maximum Magnitude columns demonstrate how the range of possible numbers increases with each level for an image starting with 1 byte per pixel. The Maximum Bits column shows the maximum number of bits (with a sign bit) necessary to represent the numeric range at each wavelet level. The Maximum Bits from Data column represents the maximum number of bits required to encode over one hundred sample images obtained from NASA. These numbers were produced via software simulation on this sample dataset.

In practice, the magnitude of the wavelet coefficients does not grow at the maximum theoretical rate. To maximize efficiency, the Maximum Bits from Data values were used to determine what position the most significant bit must stand for. Since the theoretical maximum is not used, an overflow situation may occur.

TABLE 27.1 ■ Fixed-point magnitude calculations

Wavelet level	Factor	Maximum magnitude	Maximum bits	Maximum bits from data
Input image	1	255	8	8
0	2.9054	741	11	11
1	8.4412	2152	13	12
2	24.525	6254	14	13
3	71.253	18170	16	14
4	207.02	52789	17	15
5	601.46	153373	19	16
6	1747.5	445605	20	17

To compensate, the system flags overflow occurrences as an error and truncates the data. However, after examining hundreds of sample images, no instances of overflow occurred, and the data scheme used provided enough space to capture all the required data.

If each wavelet level used the same numerical representation, they would all be required to handle numbers as large as the highest wavelet level to prevent overflow. However, since the lowest wavelet levels never encounter numbers in that range, several bits at these levels would not be used and therefore wasted.

To fully utilize all of the bits for each wavelet coefficient, we introduced the concept of *variable fixed-point* representation. With variable fixed-point we assigned a fixed-point numerical representation for each wavelet level optimized for that level's expected data size. In addition, each representation differed from one another, meaning that we employed a different fixed-point scheme for each wavelet level. Doing so allowed us to optimize both memory storage and I/O at each wavelet level to yield maximum performance.

Once the position of the most significant bit was found for each wavelet level, the number of precision bits needed to accurately represent the wavelet coefficients had to be determined. Our goal was to provide enough bits to fully recover the image and no more. Figure 27.10 displays the average PSNRs for several recovered images from SPIHT using a range of bit widths for each coefficient.

An assignment of 16 bits per coefficient most accurately matched the full-precision floating-point coefficients used in software, up through perfect reconstruction. Previous wavelet designs we looked at focused on bitrates less than 4 bits per pixel (bpp) and did not consider rounding effects on the wavelet transformation for bitrates greater than 4 bpp. These studies found this lower bitrate acceptable for lossy SPIHT compression [3].

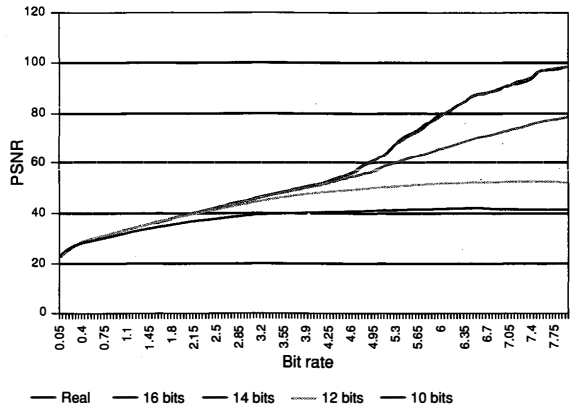


FIGURE 27.10 ■ PSNR versus bitrate for various coefficient sizes.

TABLE 27.2 ■ Final variable fixed-point representation

Wavelet level	Integer bits	Fractional bits
Input image	10	6
0	11	5
1	12	4
2	13	3
3	14	2
4	15	1
5	16	0
6	17	−1

Instead, we chose a numerical representation that retains the equivalent amount of information as a full floating-point number during wavelet transformation. By doing so, it was possible to perfectly reconstruct an image given a high enough bitrate. In other words, we allowed for a lossless implementation. Table 27.2 provides the number of integer and fractional bits allocated for each wavelet level. The number of integer bits also includes 1 extra bit for the sign value. The highest wavelet level's 16 integer bits represent positions 17 to 1, with no bit assigned for the 0 position.

27.3.3 Fixed Order SPIHT

The last major factor we took under consideration was how to parallelize the SPIHT algorithm for use in hardware. As discussed in Section 27.2, SPIHT computes a dynamic ordering of the wavelet coefficients as it progresses. By always adding pixels to the end of the LIP, LIS, and LSP, coefficients most critical to constructing a valid wavelet are generally sent first, while less critical coefficients are placed later in the lists. Such an ordering yields better image quality for bitstreams that end in the middle of a bit plane. The drawback of this ordering is that every image has a unique list order determined by the image's wavelet coefficient values.

By analyzing the SPIHT algorithm, we were able to conclude that the data a block of coefficients contributes to the final SPIHT bitstream is fully determined by the following set of localized information:

- The 2×2 block of coefficients
- Their immediate children
- The maximum magnitude of the four subtrees

As a result, we were able to show that every block of coefficients could be calculated independently and in parallel of one another. We were also able to determine that, if we could parallelize the computation of these coefficients, the final hardware implementation would operate at a much higher throughput. However, we were not able to take advantage of this parallelism because in SPIHT

the order in which a block's data is inserted into the bitstream is not known, since it depends on the image's unique ordering. Only once the order is determined is it possible to produce a valid SPIHT bitstream from the information listed previously.

Unfortunately, the algorithm employed to calculate the SPIHT ordering of coefficients is sequential. The computation steps over the coefficients of the image multiple times within each bit plane and dynamically inserts and removes coefficients from the LIP and LIS lists. Such an algorithm is not parallelizable in hardware. As a result, many of the speedups a custom hardware implementation may produce would be lost. Instead, any hardware implementation we could develop would need to create the lists in an identical manner as the software implementation. This process would require many clock cycles per block of coefficients, which would significantly limit the throughput of any SPIHT implementation in hardware.

To remove this limitation and design a faster system, we created a modification to the original algorithm called *Fixed Order SPIHT*. Fixed Order SPIHT is similar to the SPIHT algorithm shown in Figure 27.5, except that the order of the LIP, LIS, and LSP lists is fixed and known beforehand. Instead of inserting blocks of coefficients at the end of the lists, they are inserted in a predetermined order. For example, block A will always appear before block B, which is always before block C, regardless of the order in which A, B, and C were added to the lists. The order of Fixed Order SPIHT is based upon the Morton scan ordering discussed in Algazi and Estes [1].

Fixed Order SPIHT removed the need to calculate the ordering of coefficients within each bit plane and allowed us to create a fully parallel version of the original SPIHT algorithm. Such a modification increased the throughput of a hardware encoder by more than an order of magnitude at the cost of a slightly lower PSNR within each bit plane. Figure 27.11 outlines the new version of SPIHT we created. The final bitstream generated is precisely the same as the bitstream generated from the original SPIHT algorithm except that data will appear in a different order within each bit plane.

By using the algorithm in Figure 27.11 instead of the original sequential algorithm in Figure 27.8, the final datastream can be computed in one pass through the image instead of multiple passes. In addition, each pixel block is coded in parallel, which yields significantly faster compression times with FPGAs.

The advantage of this method is that at the end of each bit plane, the exact same data will have been transmitted, just in a different order. Thus, at the end of each bit plane the PSNR of Fixed Order SPIHT will match that of the original SPIHT algorithm, as shown in Figure 27.12. Since the length of each bitstream is fairly short within the transmitted datastream, the PSNR curve of Fixed Order SPIHT very closely matches that of the original algorithm. The maximum loss in quality between Fixed Order SPIHT and the original SPIHT algorithm found was 0.2 dB. This is the maximum loss any image in our sample set displayed over any bitrate from 0.05 to 8.00 bpp.

For a more complete discussion on Fixed Order SPIHT, refer to Fry [8].

```
1. Bit-plane calculation: for each 2×2 block of pixels (i,j) in a Morton
   Scan Ordering
   1.1 for each threshold level n from the highest level to the lowest
       1.1.1 if (i,j) is a root and Max((i,j)) >= n
             add all four pixels to the LIP
       1.1.2 if (i,j) is not a root and Max((i,j)) >= previous n
             for each pixel p in the block
               if p < previous n
                 add p to the LIP
               else
                 add p to the LSP
       1.1.3 if (i,j) is not a leaf and Max((i,j) ) >= n
             add all four pixel to the LIS unless (i,j) is a root, then
             just add the three with children
       1.1.4 if all four pixels are in LIS and at least one is not in the LIP
             if at least one pixel will be removed from the LIS at this level
               output a '0' to the LIS stream
             else
               output a '1' to the LIS stream
       1.1.5 for each pixel p in the LIP
              if p >= n
                output a '1' and the sign of p to the LIP stream
                remove p from the LIP and add it to the LSP
              else
                output a '0' to the LIP stream
       1.1.6 for each pixel p in the LIS
              if child max(p) >= n
                output a '1' to the LIS stream
                remove p from the LIS
                for each child (k,l) of p
                  if (k,l) >= n
                    output a '1' and the sign of (k,l) to the LIS stream
                  else
                    output a '0' to the LIS stream
              else
                output a '0' to the LIS stream
       1.1.7 for each pixel p in the LSP
             output the value of p at the bit plane n to the LSP stream
2. Grouping phase: for each threshold level n from the highest level to
   the lowest
   2.1 output the LIP stream at threshold level n to the final data stream
   2.2 output the LIS stream at threshold level n to the final data stream
   2.3 output the LSP stream at threshold level n to the final data stream
```

FIGURE 27.11 ■ Fixed Order SPIHT.

27.4 HARDWARE IMPLEMENTATION

In the following subsections we first describe the target hardware platform that the SPIHT algorithm was mapped onto. Next, we present an overview of the implementation and a detailed description of the three major steps of the

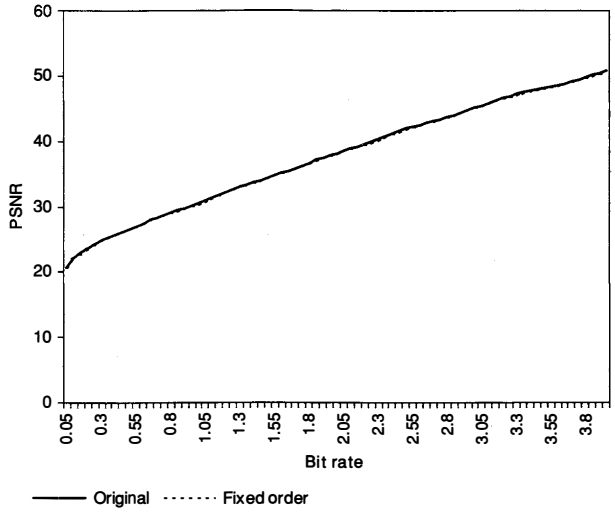


FIGURE 27.12 ■ A comparison of original SPIHT and Fixed Order SPIHT.

computation. A thorough understanding of the target platform is required because it strongly influenced the SPIHT implementation created.

27.4.1 Target Hardware Platform

The target platform was the WildStar FPGA processor board developed by Annapolis Microsystems [2]. Shown in Figure 27.13, it consists of three Xilinx Virtex 2000E FPGAs—PE 0, PE 1, and PE 2—and operates at rates of up to 133 MHz. The board makes available 48 MBytes of memory through 12 individual memory ports, between 32 and 64 bits wide, yielding a throughput of up to 8.5 GBytes/sec. Four shared memory blocks connect the Virtex chips through a crossbar. By switching a crossbar, several MBytes of data are passed between the chips in just a few clock cycles.

The Xilinx Virtex 2000E FPGA allows for 2 million gate designs [22]. For extra on-chip memory, the FPGAs contain 160 asynchronous dual-ported BlockRAMs. Each BlockRAM stores 4096 bits of data and is accessible in 1-, 2-, 4-, 8-, or 16-bit-wide words. Because they are dual ported, the BlockRAMs function well as first in, first outs (FIFOs). A PCI bus connects the board to a host computer.

27.4.2 Design Overview

The architecture constructed consisted of three phases: wavelet transform, maximum magnitude calculation, and Fixed Order SPIHT coding. Each phase

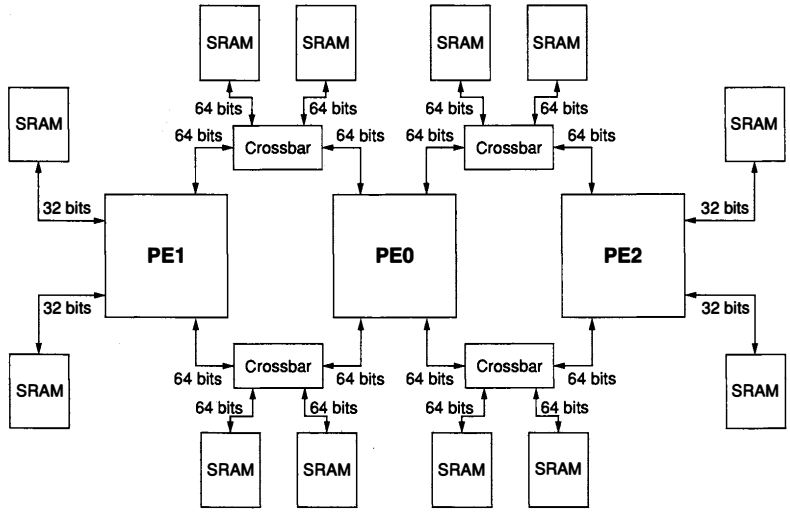


FIGURE 27.13 ■ A block diagram of the Annapolis Microsystems WildStar board.

was implemented in one of the three Virtex chips. By instantiating each phase on a separate chip, separate images could be operated on in parallel. Data was transferred from one phase by the next through the shared memories. The decision on how to break up the phases came naturally from the resources available in each FPGA and the requirements of each section. The DWT and the SPIHT coding phases each required close to the full resources of a single FPGA, and the maximum magnitude phase needed to be completed prior to the SPIHT coding phase. These characteristics of the algorithm and system naturally lead to placing the three phases on the three separate FPGAs.

The architecture was also designed in this manner because once processing in a phase is complete, the crossbar mode could be switched and the data calculated would be accessible to the next chip. By coding a different image in each phase simultaneously, the throughput of the system is determined by the slowest phase, while the latency of the architecture is the sum of the three phases. Figure 27.14 illustrates the architecture of the system.

27.4.3 Discrete Wavelet Transform Phase

As discussed in Section 27.3.1, after implementing each algorithm in hardware we chose a simple folded architecture, which matched the bandwidth, memory, and chip capacities of the target board well. The results of this phase are stored into memory and passed to the maximum magnitude phase.

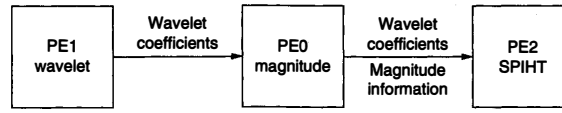


FIGURE 27.14 ■ An overview of the architecture.

27.4.4 Maximum Magnitude Phase

Once the DWT is complete, the next phase prepares and organizes the image into a form easily readable by the parallel version of the SPIHT coder. Specifically, the maximum magnitude phase calculates and rearranges the following information for the next phase:

- The maximum magnitude of each of the four child trees
- The absolute value of the 2×2 block of coefficients
- A sign value for each coefficient in the block
- The threshold level when the block is first inserted into the LIS by its parent
- Threshold and sign data of each of the 16 child coefficients
- Reorder the wavelet coefficients into a Morton Scan Ordering

The SPIHT coding phase shares two 64-bit memory ports with the maximum magnitude phase, allowing it to read 128 bits on each clock cycle. The data just listed can fit into these two memory ports. By doing so on every clock cycle the SPIHT coding phase will be able to read and process an entire block of data. The data that the maximum magnitude phase calculates is shown in Figure 27.15.

To calculate the maximum magnitude of all coefficients below a node in the spatial orientation trees, the image must be scanned in depth-first search order [7]. With a depth-first search, whenever a new coefficient is read and considered, all of its children will have already been read and the maximum coefficient so far is known. On every clock cycle the new coefficient is compared to and updates the current maximum. Because PE 0 (the maximum magnitude phase) uses 32-bit-wide memory ports, it can read half a block at a time.

The state machine, which controls how the spatial orientation trees are traversed, reads one-half of a block as it descends the tree, and the other half as it ascends the tree. By doing so all of the data needed to compute the maximum magnitude for the current block is available as the state machine ascends back up the spatial orientation tree. In addition, the four most recent blocks of each level are saved onto a stack so that all 16 child coefficients are available to the parent block.

Figure 27.16 demonstrates the algorithm. The current block, maximum magnitude for each child, and 16 child coefficients are shown on the stack. Light gray blocks are coefficients previously read and processed. Dark gray blocks are coefficients currently being read. In this example, the state machine has just finished reading the lowest level and has ascended to the second wavelet level.