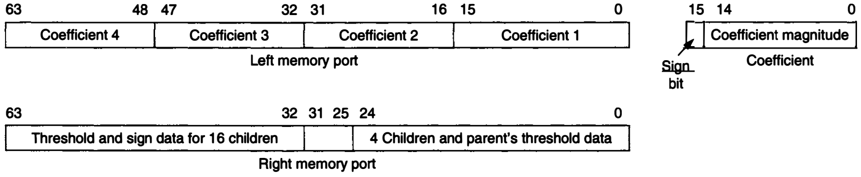


FIGURE 27.15 ■ Data passed to the SPIHT coder to calculate a single block.

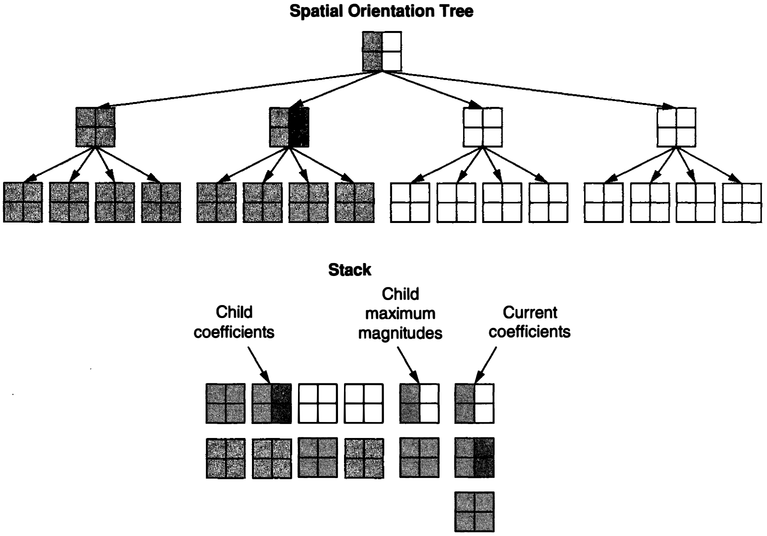


FIGURE 27.16 ■ A depth-first search of the spatial orientation trees.

The second block in the second level is now complete, and its maximum magnitude can now be calculated, shown as the dark gray block in the stack's highest level. In addition, the 16 child coefficients in the lowest level were saved and are available. There are no child values for the lowest level since there are no children.

Another benefit of scanning the image in a depth-first search order is that Morton Scan Ordering is naturally realized within each level, although it is intermixed between levels. By writing data from each level to a separate area of memory and later reading the data from the highest wavelet level to the lowest, the Morton

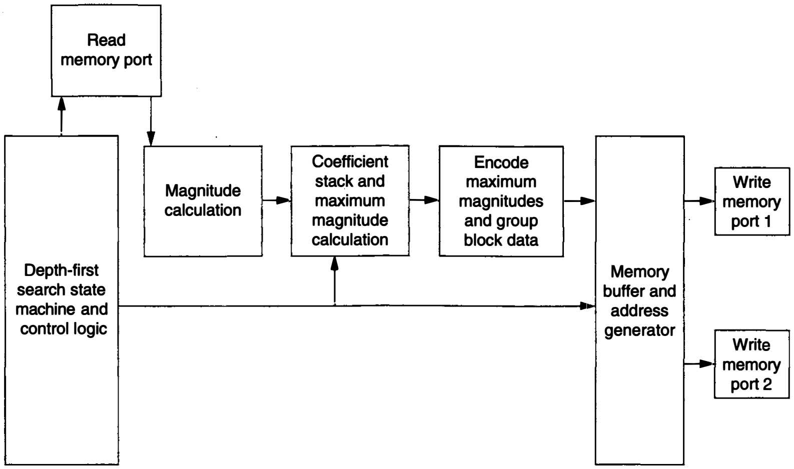

Read memory port
Depth-first search state machine and control logic
Magnitude calculation
Coefficient stack and maximum magnitude calculation
Encode maximum magnitudes and group block data
Memory buffer and address generator
Write memory port 1
Write memory port 2

FIGURE 27.17 ■ A block diagram of the SPIHT maximum magnitude phase.

Scan Ordering is naturally realized. A block diagram of the maximum magnitude phase is provided in Figure 27.17. Since two pixels are read together and the image is scanned only once, the runtime of this phase is half a clock cycle per pixel. Because the maximum magnitude phase computes in less time than the wavelet phase, the throughput of the overall system is not affected.

27.4.5 The SPIHT Coding Phase

The final SPIHT coding phase performs the Fixed Order SPIHT encoding in parallel, based on the data from the maximum magnitude phase. Coefficient blocks are read from the highest wavelet level to the lowest. As information is loaded from memory it is shifted from the variable fixed-point representation to a common fixed-point representation for every wavelet level. Once each block has been adjusted to an identical numerical representation, the parallel version of SPIHT is used to calculate what information each block will contribute to each bit plane.

The information is grouped and counted before being added to three separate variable FIFOs for each bit plane. The data that the variable FIFO components receive range in size from 0 to 37 bits, and the variable FIFOs arrange the block data into regular sized 32-bit words for memory access. Care is also taken to stall the algorithm if any of the variable FIFOs becomes too full.

Data from each buffer is output to a fixed location in memory and the number of bits in each bitstream is output as well. Given that data is added dynamically to each bitstream, there needs to be a dynamic scheduler to select which buffer should be written to memory. Since there are a large number of FIFOs that all require a BlockRAM, the FIFOs are spread across the FPGA, and some type of staging is required to prevent a signal from traveling too far. The scheduler selects which FIFO to read based on both how full a FIFO is and when it was last accessed.

Our studies showed that the LSP bitstream is roughly the same size of the LIP and LIS streams combined. Because of this the LSP bitstreams transfer more data to memory than the other two lists. In our design the LIP and LIS bitstreams share a memory port while the LSP stream writes to a separate memory port. Since a 2 × 2 block of coefficients is processed every clock cycle, the design takes one-quarter of a clock cycle per pixel, which is far less than the three-quarters of a clock cycle per pixel for the DWT. The block diagram for the SPIHT coding phase is given in Figure 27.18.

With 22 total bit planes to calculate, the design involves 66 individual data grouping and variable FIFO blocks. Although none consume a significant amount of FPGA resources individually, 66 blocks do. The entire design required 160 percent of the resources in a Virtex 2000E, and would not fit in the target system. However, by removing the lower bit planes, less FPGA resources are needed, and the architecture can easily be adjusted to fit the FPGA being used. Depending on the size of the final bitstream required, the FPGA size used in the SPIHT phase can be varied to handle the number of intermediate bitstreams generated.

Removing lower bit planes is possible since the final bitstream transmits data from the highest bit plane to the lowest. In our design the lower 9-bit planes

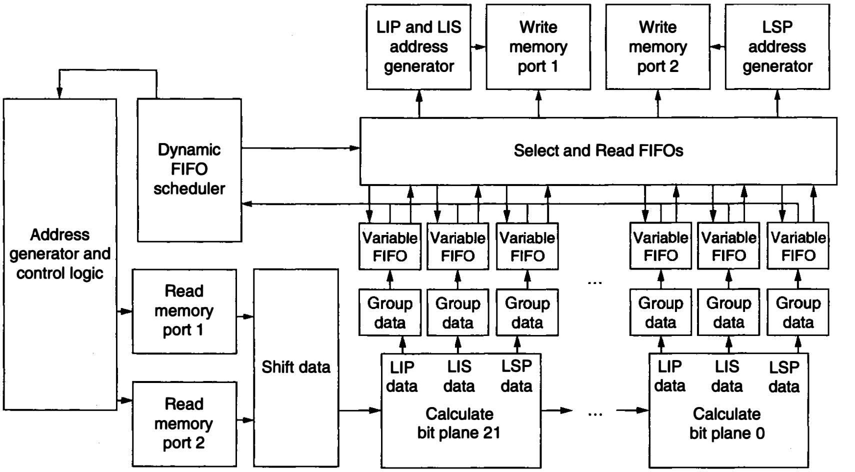


FIGURE 27.18 ■ A block diagram of the SPIHT coding phase.

were eliminated. Yet, without these lower planes, bitrates of up to 6 bpp can still be achieved. We found the constraint to be acceptable because we are interested in high compression ratios using low bitrates, and 6 bpp is practically a lossless signal. Since SPIHT is optimized for lower bitrates, the ability to calculate higher bitrates was not considered necessary. Alternatively, the use of a larger FPGA would alleviate the size constraint.

27.5 DESIGN RESULTS

The system was designed using VHDL with models provided by Annapolis Micro Systems to access the PCI bus and memory ports. Simulations for debugging purposes were carried out with ModelSim EE 5.4e from Mentor Graphics. Synplify 6.2 from Synplicity was used to compile the VHDL code and generate a netlist. The Xilinx Foundation Series 3.1i tool set was used to place and route the design. Lastly, the `peutil.exe` utility from Annapolis Micro Systems generated the FPGA configuration streams.

Table 27.3 shows the speed and runtime specifications of the final architecture. All performance numbers are measured results from the actual hardware implementation. Each phase computes on separate memory blocks, which can operate at different clock rates. The design can process any square image where the dimensions are a power of 2: 16 × 16, 32 × 32, up to 1024 × 1024.

Since the WildStar board is connected to the host computer by a relatively slow PCI bus, the throughput of the entire system we built is constrained by the throughput of the PCI bus. However, since the study is on how image compression routines could be implemented on a satellite, such a system would be designed differently, and would not contain a reconfigurable board connected to some host platform though a PCI bus. Instead, the image compression routines would be inserted directly into the data path and the data transfer times would not be the bottleneck of the system. For this reason we analyzed the throughput of just the SPIHT compression engine and analyzed how quickly the FPGAs can process the images.

The throughput of the system was constrained by the discrete wavelet transform at 100 MPixels/sec. One method to increase this rate is to compute more rows in parallel. If the available memory ports accessed 128 bits of data instead of the 64 bits with our WildStar board, the number of clock cycles per pixel could be reduced by half and the throughput could double.

TABLE 27.3 ■ Performance numbers

Phase	Clock cycles per 512 × 512 image	Clock cycles per pixel	Clock rate	Throughput	FPGA area (%)
Wavelet	182465	3/4	75 MHz	100 MPixels/sec	62
Magnitude	131132	1/2	73 MHz	146 MPixels/sec	34
SPIHT	65793	1/4	56 MHz	224 MPixels/sec	98

Assuming the original image consists of 8 bpp, images are processed at a rate of 800 Mbits/sec.

The entire throughput of the architecture is less than one clock cycle for every pixel, which is lower than parallel versions of the DWT. Parallel versions of the DWT used complex scheduling to compute multiple wavelet levels simultaneously, which left limited resources to process multiple rows at a time. Given more resources though, they would obtain higher data rates than our architecture by processing multiple rows simultaneously. In the future, a DWT architecture other than the one we implemented could be selected for additional speed improvements.

We compared our results to the original software version of SPIHT provided on the SPIHT web site [15]. The comparison was made without arithmetic coding since our hardware implementation does not perform any arithmetic coding on the final bitstream. Additionally, in our testing on sample NASA images, arithmetic coding added little to overall compression rates and thus was dropped [11]. An IBM RS/6000 Model 270 workstation was used for the comparison, and we used a combination of standard image compression benchmark images and satellite images from NASA's web site. The software version of SPIHT compressed a 512×512 image in 1.101 seconds on average without including disk access. The wavelet phase, which constrains the hardware implementation, computes in 2.48 milliseconds, yielding a speedup of 443 times for the SPIHT engine. In addition, by creating a parallel implementation of the wavelet phase, further improvements to the runtimes of the SPIHT engine are possible.

While this is the speedup we will obtain if the data transfer times are not a factor, the design may be used to speed up SPIHT on a general-purpose processor. On such a system the time to read and write data must be included as well. Our WildStar board is connected to the host processor over a PCI bus, which writes images in 13 milliseconds and reads the final datastream in 20.75 milliseconds. Even with the data transfer delay, the total speedup still yields an improvement of 31.4 times.

Both the magnitude and SPIHT phases yield higher throughputs than the wavelet phase, even though they operate at lower clock rates. The reason for the higher throughputs is that both of these phases need fewer clock cycles per pixel to compute an image. The magnitude phase takes half a clock cycle per pixel and the SPIHT phase requires just a quarter. The fact that the SPIHT phase computes in less than one clock cycle per pixel, let alone a quarter, is a striking result considering that the original SPIHT algorithm is very sequential in nature and had to consider each pixel in an image multiple times per bit plane.

27.6 SUMMARY AND FUTURE WORK

In this chapter we demonstrated a viable image compression routine on a reconfigurable platform. We showed how by analyzing the range of data processed by each section of the algorithm, it is advantageous to create optimized memory structures as with our variable fixed-point work. Doing so minimizes memory usages and yields efficient data transfers. Here each bit transferred between memory and the processor board directly impacted the final results. In addition, our Fixed Order SPIHT modifications illustrate how by making slight adjustments to an existing algorithm, it is possible to dramatically increase the performance in a custom hardware implementation and simultaneously yield essentially identical results. With Fixed Order SPIHT the throughput of the system increased by over an order of magnitude while still matching the original algorithm's PSNR curve.

This SPIHT work was part of a development effort funded by NASA.

References

[1] V. R. Algazi, R. R. Estes. Analysis-based coding of image transform and subband coefficients. *Applications of Digital Image Processing XVIII, SPIE Proceedings* 2564, 1995.

[2] Annapolis Microsystems. *WildStar Reference Manual*, Annapolis Microsystems, 2000.

[3] A. Benkrid, D. Crookes, K. Benkrid. Design and implementation of generic 2D biorthogonal discrete wavelet transform on an FPGA. *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[4] M. Carraeu. Hubble Servicing Mission: Hubble is fitted with a new "eye." *http://www.chron.com/content/interactive/space/missions/sts-103/hubble/archive/931207.html*, December 7, 1993.

[5] C. M. Chakrabarti, M. Vishwanath. Efficient realization of the discrete and continuous wavelet transforms: From single chip implementations to mappings in SIMD array computers. *IEEE Transactions on Signal Processing* 43, March 1995.

[6] C. M. Chakrabarti, M. Vishwanath, R. M. Owens. Architectures for wavelet transforms: A survey. *Journal of VLSI Signal Processing* 14, 1996.

[7] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*, MIT Press, 1997.

[8] T. W. Fry. *Hyper Spectral Image Compression on Reconfigurable Platforms*, Master's thesis, University of Washington, Seattle, 2001.

[9] R. C. Gonzalez, R. E. Woods. *Digital Image Processing*, Addison-Wesley, 1993.

[10] A. Graps. An introduction to wavelets. *IEEE Computational Science and Engineering* 2(2), 1995.

[11] T. Owen, S. Hauck. *Arithmetic Compression on SPITH Encoded Images*, Technical report UWEETR-2002-2007, Department of Electrical Engineering, University of Washington, Seattle, 2002.

[12] K. K. Parhi, T. Nishitani. VLSI architectures for discrete wavelet transforms. *IEEE Transactions on VLSI Systems* 1(2), 1993.

[13] J. Ritter, P. Molitor. A pipelined architecture for partitioned DWT based lossy image compression using FPGAs. *ACM/SIGDA Ninth International Symposium on Field-Programmable Gate Arrays*, February 2001.

[14] A. Said, W. A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology* 6, June 1996.

[15] A. Said, W. A. Pearlman. SPIHT image compression: Properties of the method. *http://www.cipr.rpi.edu/research/SPIHT/spiht1.html.*

[16] H. Sava, M. Fleury, A. C. Downton, A. Clark. Parallel pipeline implementations of wavelet transforms. *IEEE Proceedings Part 1 (Vision, Image and Signal Processing)* 144(6), 1997.

[17] J. M. Shapiro. Embedded image coding using zero trees of wavelet coefficients. *IEEE Transactions on Signal Processing* 41(12), 1993.
[18] W. Sweldens. The Lifting Scheme: A new philosophy in biorthogonal wavelet constructions. *Wavelet Applications in Signal and Image Processing* 3, 1995.
[19] NASA. TERRA: The EOS flagship. The EOS Data and Information System (EOSDIS). *http://terra.nasa.gov/Brochure/Sect_5-1.html.*
[20] C. Valens. A really friendly guide to wavelets. *http://perso.wanadoo.fr/polyvalens/clemens/wavelets/wavelets.html.*
[21] M. Vishwanath, R. M. Owens, M. J. Irwin. VLSI architectures for the discrete wavelet transform. *IEEE Transactions on Circuits and Systems, Part II*, May 1995.
[22] Xilinx, Inc. *The Programmable Logic Data Book*, Xilinx, Inc., 2000.
[23] Xilinx, Inc. *Serial Distributed Arithmetic FIR Filter*, Xilinx, Inc., 1998.

CHAPTER 28

AUTOMATIC TARGET RECOGNITION SYSTEMS ON RECONFIGURABLE DEVICES

Young H. Cho
Open Acceleration Systems Research

An Automatic Target Recognition (ATR) system analyzes a digital image or video sequence to locate and identify all objects of a certain class. There are several ways to implement ATR systems, and the right one is dependent, in large part, on the operating environment and the signal source. In this chapter we focus on the implementations of reconfigurable ATR designs based on the algorithms from Sandia National Laboratories (SNL) for the U.S. Department of Defense Joint STARS airborne radar imaging platform. STARS is similar to an aircraft AWACS system, but detects ground targets.

ATR in Synthetic Aperture Radar (SAR) imagery requires tremendous processing throughput. In this application, data come from high-bandwidth sensors, and the processing is time critical. On the other hand, there is limited space and power for processing the data in the sensor platforms. One way to meet the high computational requirement is to build custom circuits as an ASIC. However, very high nonrecurring engineering (NRE) costs for low-volume ASICs, and often evolving algorithms, limit the feasibility of using custom hardware. Therefore, reconfigurable devices can play a prominent role in meeting the challenges with greater flexibility and lower costs.

This chapter is organized as follows: Section 28.1 describes a highly parallelizable Automatic Target Recognition (ATR) algorithm. The system based on it is implemented using a mix of software and hardware processing, where the most computationally demanding tasks are accelerated using field-programmable gate arrays (FPGAs). We present two high-performance implementations that exercise the FPGA's benefits. Section 28.2 describes the system that automatically builds algorithm-specific and resource-efficient "hardwired" accelerators. It relies on the dynamic reconfiguration feature of FPGAs to obtain high performance using limited logic resources.

The system in Section 28.3 is based on an architecture that does not require frequent reconfiguration. The architecture is modular, easily scalable, and highly tuned for the ATR application. These application-specific processors are automatically generated based on application and environment parameters. In Section 28.4 we compare the implementations to discuss the benefits and the trade-offs of designing ATR systems using FPGAs. In Section 28.5, we draw our conclusions on FPGA-based ATR system design.

28.1 AUTOMATIC TARGET RECOGNITION ALGORITHMS

Sandia real-time SAR ATR systems use a hierarchy of algorithms to reduce the processing demands for SAR images in order to yield a high probability of detection (PD) and a low false alarm rate (FAR).

28.1.1 Focus of Attention

As shown in Figure 28.1, the first step in the SNL algorithm is a Focus of Attention (FOA) algorithm that runs over a downsampled version of the entire image to find regions of interest that are of approximately the right size and brightness. These regions are then extracted and processed by an indexing stage to further reduce the datastream, which includes target hypotheses, orientation estimations, and target center locations. The surviving hypotheses have the full resolution data sent to an identification executive that schedules multiple identification algorithms and then fuses their results.

The FOA stage identifies interesting image areas called "chips." Then it composes a list of targets suspected to be in a chip. Having access to range and altitude information, the FOA algorithm also determines the elevation for the chip, without having to identify the target first. It then tasks the next stage with evaluating the likelihood that the suspected targets are actually in the given image chip and exactly where.

28.1.2 Second-level Detection

The next stage of the algorithm, called Second Level Detection (SLD), takes the extracted imagery (an image chip), matches it against a list of provided target

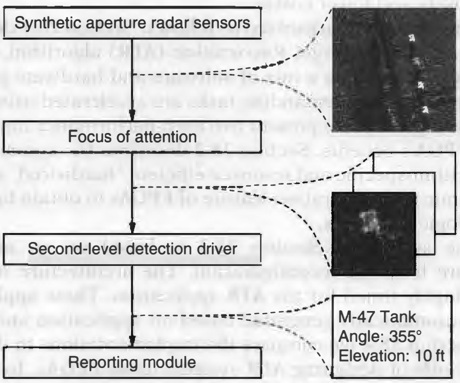


FIGURE 28.1 ■ The Sandia Automatic Target Recognition algorithm.

hypotheses, and returns the hit information for each image chip consisting of the best two orientation matches and other relevant information.

The system has a database of target models. For each target, and for each of its three different elevations, 72 templates are defined corresponding to its all-around views. The orientations of adjacent views are separated by 5 degrees.

SLD is a binary silhouette matcher that has a bright mask and a surround mask that are mutually exclusive. Each template is composed of several parameters along with a "bright mask" and a "surround mask," where the former defines the image pixels that should be bright for a match, and the latter defines the ones that should not. The bright and surround masks are 32×32 bitmaps, each with about 100 asserted bits. "Bright" is defined relative to a dynamic threshold.

On receiving tasks from the FOA, the SLD unit compares all of the stored templates for this target and elevation and the applicable orientations with the image chip, and computes the level of matching (the "hit quality"). The two hits with the highest quality are reported to the SLD driver as the most likely candidates to include targets. For each hit, the template index number, the exact position of the hit in the search area, and the hit quality are provided. After receiving this information, the SLD driver reports it to the ATR system.

The purpose of the first step in the SLD algorithm, called the shape sum, is to distinguish the target from its surrounding background. This consists of adaptively estimating the illumination for each position in the search area, assuming that the target is at that orientation and location. If the energy is too little or too much, no further processing for that position for that template match is required. Hence, for each mask position in the search area, a specific threshold value is computed as in equation 28.1.

$$SM_{x,y} = \sum_{u=0}^{31} \sum_{v=0}^{31} B_{u,v} M_{x+u,y+v} \tag{28.1}$$

$$TH_{x,y} = \frac{SM_{x,y}}{BC} - Bias \tag{28.2}$$

The next step in the algorithm distinguishes the target from the background by thresholding each image pixel with respect to the threshold of the current mask position, as computed before. The same pixel may be above the threshold for some mask positions but below it for others. This threshold calculation determines the actual bright and surround pixel for each position. As shown in equation 28.2, it consists of dividing the shape sum by the number of pixels in the bright mask and subtracting a template-specific *Bias* constant.

As shown in equation 28.3, the pixel values under the bright mask that are greater than or equal to the threshold are counted; if this count exceeds the minimal bright sum, the processing continues. On the other hand, the pixel values under the surround mask that are less than the threshold are counted to calculate the surround sum as shown in equation 28.4. If this count exceeds the minimal surround sum, it is declared a hit.

$$BS_{x,y} = \sum_{u=0}^{31}\sum_{v=0}^{31} B_{u,v}\left[M_{x+u,y+v} \geq TH_{x,y}\right] \tag{28.3}$$

$$SS_{x,y} = \sum_{u=0}^{31}\sum_{v=0}^{31} S_{u,v}\left[M_{x+u,y+v} < TH_{x,y}\right] \tag{28.4}$$

Once the position of the hit is determined, we can calculate its quality by taking the average of bright and surround pixels that were correct, as shown in equation 28.5. This quality value is sent back to the driver with the position to determine the two best targets.

$$Q_{x,y} = \frac{1}{2}\left(\frac{BS_{x,y}}{BC} + \frac{SS_{x,y}}{SC}\right) \tag{28.5}$$

28.2 DYNAMICALLY RECONFIGURABLE DESIGNS

FPGAs can be reconfigured to perform multiple functions with the same logic resources by providing a number of corresponding configuration bit files. This ability allows us to develop dynamically reconfigurable designs. In this section, we present an ATR system implementation of UCLA's Mojave project that uses an FPGA's dynamic reconfigurability.

28.2.1 Algorithm Modifications

As described previously, the current Sandia system uses 64 × 64 pixel chips and 32 × 32 pixel templates. However, the Mojave system uses chip sizes of 128 × 128 pixels and template sizes of 8 × 8 pixels. It uses different chip and template sizes in order to map into existing FPGA devices that are relatively small. A single template moves through a single chip to yield 14,641 (121 × 121) image correlation results. Assuming that each output can be represented with 6 bits, the 87,846 bits are produced by the system.

There is also a divide step in the Sandia algorithm that follows the shape sum operation and guides the selection of threshold bin for the chip. This system does not implement the divide, mainly because it is expensive relative to available FPGA resources for the design platform.

28.2.2 Image Correlation Circuit

FPGAs offer an extremely attractive solution to the correlation problem. First of all, the operations being performed occur directly at the bit level and are dominated by shifts and adds, making them easy to map into the hardware provided by the FPGA. This contrasts, for example, with multiply-intensive algorithms that would make relatively poor utilization of FPGA resources. More important, the sparse nature of the templates can be utilized to achieve a far more efficient implementation in the FPGA than could be realized in a general-purpose correlation device. This can be illustrated using the example of the simple template shown in Figure 28.2.

In the example template shown in the figure, only 5 of the 20 pixels are asserted. At any given relative offset between the template and the chip, the correlation output is the sum of the 5 binary pixels in the chip that match the asserted bits in the template. The template can therefore be implemented in the FPGA as a simple multiple-port adder. The chip pixel values can be stored in flip-flops and are shifted to the right by one flip-flop with each clock cycle. Though correlation of a large image with a small mask is often understood conceptually in terms of the mask being scanned across the image, in this case the opposite is occurring—the template is hardwired into the FPGA while the image pixels are clocked past it.

Another important opportunity for increased efficiency lies in the potential to combine multiple templates on a single FPGA. The simplest way to do this is to spatially partition the FPGA into several smaller blocks, each of which handles the logic for a single template. Alternatively, we can try to identify templates that have some topological commonality and can therefore share parts of their adder trees. This is illustrated in Figure 28.3, which shows two templates sharing several pixels that can be mapped using a set of adder trees to leverage this overlap.

A potential advantage FPGAs have over ASICs is that they can be dynamically optimized at the gate level to exploit template characteristics. For our application, a programmable ASIC design would need to provide large general-purpose adder trees to handle the worst-case condition of summing all possible template bits, as shown in Figure 28.4. In constrast, an FPGA exploits the sparse nature of the templates and constructs only the small adder trees required. Additionally, FPGAs can optimize the design based on other application-specific characteristics.

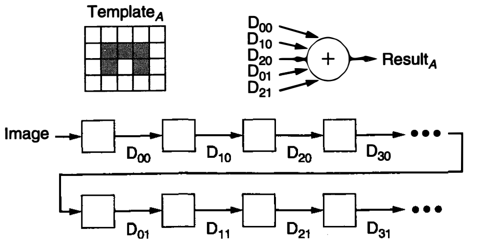


FIGURE 28.2 ■ An example template and a corresponding register chain with an adder tree.

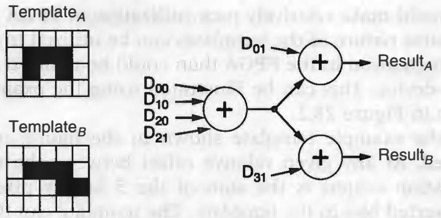


FIGURE 28.3 ■ Common hardware shared between two templates.

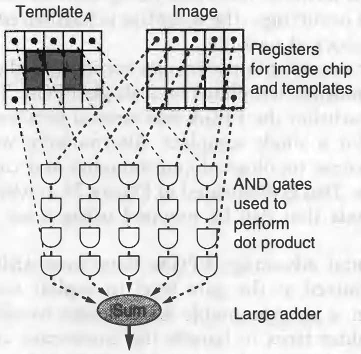

Template$_A$
Image
Registers for image chip and templates
AND gates used to perform dot product
Sum
Large adder

FIGURE 28.4 ■ The ASIC version of the equivalent function.

28.2.3 Performance Analysis

Using a template-specific adder tree achieves significant reduction in routing complexity over a general correlation device, which must include logic to support arbitrary templates. The extent of this reduction is inversely proportional to the fraction of asserted pixels in the template. While this complexity reduction is important, alone it is not sufficient to lead to efficient implementations on FPGAs. The number of D-flip-flops required for storing the data points can cause inefficiencies in the design. Implementing these on the FPGA using the usual flip-flop–based shift registers is inefficient.

This problem can be resolved by collapsing the long strings of image pixels—those not being actively correlated against a template—into shift registers, which can be implemented very efficiently on some lookup table (LUT)–based FPGAs. For example, LUTs in the Xilinx XC4000 library can be used as shift registers that delay data by some predetermined number of clock cycles. Each 16×1-bit

LUT can implement an element that is effectively a 16-bit shift register in which the internal bits cannot be accessed. A flip-flop is also needed at the output of each RAM to act as a buffer and synchronizer. A single control circuit is used to control the stepping of the address lines and the timely assertion of the write-enable and output-enable signals for all RAM-based shift register elements. This is a small price to pay for the savings in configurable logic block (CLB) usage relative to a brute-force implementation using flip-flops.

In contrast, the 256-pixel template images, like those shown in Figure 28.5, can be stored easily using flip-flop–based registers. This is because sufficient flip-flops are available to do this, and the adder tree structures do not consume them. Also, using standard flip-flop–based shift registers for image pixels in the template simplifies the mapping process by allowing every pixel to be accessed. New templates can be implemented by simply connecting the template pixels of concern to the inputs of the adder tree structures. This leads to significant simplification of automated template-mapping tools.

The resources used by the two components of target correlation—namely, storage of active pixels on the FPGA and implementation of the adder tree corresponding to the templates—are independent of each other. The resources used by the pixel storage are determined by the template size and are independent of the number of templates being implemented. Adding templates involves adding new adder tree structures and hence increases the number of function generators being used. The total number of templates on an FPGA is bounded by the number of usable function generators.

The experimental results suggest that in practice we can expect to fit 6 to 10 surround templates having a higher number of overlapping pixels onto a 13,000-gate FPGA. However, intelligent grouping of compatible templates is important. Because the bright templates are less populated than the surround templates, we estimate that 15 to 20 of them can be mapped onto the same FPGA.

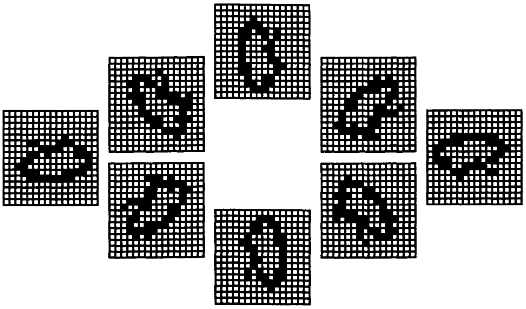


FIGURE 28.5 ■ Example of eight rotation templates of a SAR 16 × 16 bitmap image.

28.2.4 Template Partitioning

To minimize the number of FPGA reconfigurations necessary to correlate a given target image against the entire set of templates, it is necessary to maximize the number of templates in every configuration of the FPGA. To accomplish this optimization goal, we want to partition the set of templates into groups that can share adder trees so that fewer resources are used per template. The set of templates may number in the thousands, and the goal may be to place 10 to 20 of them per configuration; thus, exhaustive enumeration of all of the possible groupings is not an option. Instead, we use a heuristic method that furnishes a good, although perhaps suboptimal, solution.

Correlation between two templates can establish the number of pixels in common, and it is a good starting point for comparing and selecting templates. However, some extra analysis, beyond iterative correlations on the template set, is necessary. For example, a template with many pixels correlates well with several smaller templates, perhaps even completely subsuming them, but the smaller templates may not correlate with each other and involve no redundant computations. There are two possible solutions to this. The first is to ensure that any template added to an existing group is approximately the same size as the templates already in it. The second is to compute the number of additions required each time a new template is brought in—effectively recomputing the adder tree each time.

Recomputing the entire adder tree is computationally expensive and not a good method of partitioning a set of templates into subsets. However, one of the heuristics used in deciding whether or not to include a template in a newly formed partition is to determine the number of new terms that its inclusion would create in the partition's adder tree. The assumption is that more terms would result in a significant number of new additions, resulting in a wider and deeper adder tree. Thus, by keeping to a minimum the number of new terms created, newly added templates do not increase the number of additions by a significant amount.

Using C++, we have created a design tool to implement the partitioning process that uses an iterative approach to partitioning templates. Templates that compare well to a chosen "base" template (usually selected by largest area) are removed from the main template set and placed in a separate partition. This process is repeated until all templates are partitioned. After the partitions have been selected, the tool computes the adder tree for each partition.

Figure 28.6 shows the creation of an adder tree from the templates in a partition. Within each partition, the templates are searched for shared subsets of pixels. Called *terms*, these subsets can be automatically added together, leading to a template description that uses terms instead of pixels.

The most common addition of two terms is chosen to be grouped together, to form a new term that can be used by the templates. In this way, each template is rebuilt by combining terms in such a way that the most redundant additions are shared between templates; the final result is terms that compute entire templates. For the sample templates shown in Figure 28.6, 39 additions would be required to compute the correlations for all 5 in a naive approach. However,

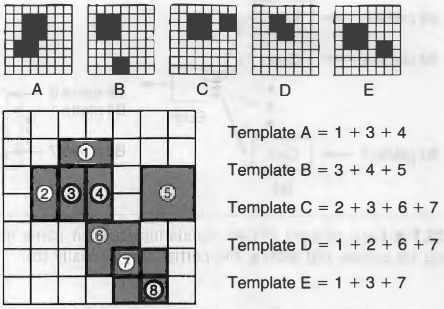


FIGURE 28.6 ■ Example of template grouping and rewritten as sums of terms.

after combining the templates through the process just described, only 17 additions are required.

28.2.5 Implementation Method

For a configurable computing system, the problem of dividing hardware and software is particularly interesting because it is both a hardware and a software issue. Consider the two methods for performing addition shown in Figure 28.7. Method A, a straightforward parallel implementation requiring several FPGAs, has several drawbacks. First, the outputs from several FPGAs converge at the addition operation, which may create a severe I/O bottleneck. Second, the system is not scalable—if it requires more precision, and therefore more bit planes, more FPGAs must be added.

Method B in Figure 28.7 illustrates our approach. Each bit plane is correlated individually and then added to the previous results in temporary storage. It is completely scalable to any image or template precision, and it can implement all correlation, normalization, and peak detection routines required for ATR. One drawback of method B is the cost and power required for the resulting wide temporary SRAM. Another possible drawback is the extra execution time required to run ATR correlations in serial. The ratio of performance to number of FPGAs is roughly equivalent for the two methods, and the performance gap can be closed simply by using more of the smaller method B boards.

The approach of a reconfigurable FPGA connected to an intermediate memory allows us a fairly complicated flow of control. For example, the sum calculation in ATR tends to be more difficult than the image–template correlation. Thus, we may want a program that performs two sum operations and forwards the results to a single correlation.

Reconfigurations for 10K-gate FPGAs are typically around 20 kB in length. Reconfiguring every 20 milliseconds gives a reconfiguration bandwidth of approximately 1 MB per FPGA per second. Coupled with the complexity of the

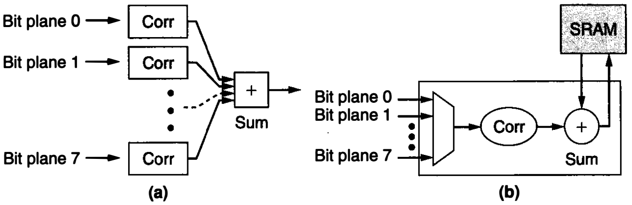


FIGURE 28.7 ■ Each of eight FPGAs correlating each bit plane of the template (a). A single FPGA correlating bit planes and adding the partial sums serially (b).

flow control, this reconfiguration bandwidth can be handled by placing a small microcontroller and configuration RAM next to every FPGA. The microcontroller permits complicated flow of control, and since it addresses the configuration RAM, it frees up valuable I/O on the FPGA. The microcontroller is also important for debugging, which is a major issue in configurable systems because the many different hardware configurations can make it difficult to isolate problems.

The principal components include a "dynamic" FPGA, which is reconfigured on the fly and performs most of the computing functions, and a "static" FPGA, which is configured only once and performs control and some computational functions. The EPROM holds configuration bitstreams, and the SRAM holds the input image data (e.g., the chip). Because the correlation operation involves the application of a small target template to a large chip, a first in, first out (FIFO) is needed to hold the pixels being wrapped around to the next row of the template mask. The templates used in this implementation are of size 8 × 8, whereas the correlation image is 128 × 128. Each configuration of the dynamic FPGA implements a total of four template pairs (four bright templates and four surround templates).

The large amount of sum in the algorithm can be performed in parallel. This requires a total of *D* clock cycles, where *D* is each pixel's depth of representation. Once the sum results are obtained, the correlation outputs are produced at the rate of 1 per clock cycle. Parallelism cannot be as directly exploited in this step because different pixels are asserted for different templates. However, in the limit of very large FPGAs the number of clock cycles to compute the correlation is upper-bounded by the number of possible thresholds, as opposed to the number of templates.

28.3 RECONFIGURABLE STATIC DESIGN

Although the idea of reusing reconfigurable hardware to dynamically perform different functions is unique to FPGAs, the main weaknesses of dynamic FPGA reconfiguration are the lengthy time and additional resources required for FPGA reconfiguration and design compilation. Although reconfiguration time has improved dramatically over the years, any time spent on reconfiguration is time that could be used to process more data.

Unlike the dynamic reconfigurable architecture describe in the previous section, we describe another efficient FPGA design that does not require complete design reconfiguration. However, like the previous system, it uses a number of parameters to design a highly pipelined custom design to maximize utilization of the design space to exploit the parallelism in the algorithm.

28.3.1 Design-specific Parameters

To verify our understanding of the algorithm, we first implemented a software simulator and ran it on a sample dataset. Our simulations reproduced the expected results. Over time this algorithm simulator became a full hardware simulator and verifier. It also allowed us to investigate various design options before implementing them in hardware.

The dataset includes 2 targets, each with 72 templates for 5-degree orientation intervals. In total, then, we have 144 bright masks and 144 surround masks, each a 32×32 bitmap. The dataset also includes 16 image chips, each with 64×64 pixels at 1 byte per pixel. Given a template and an image, there are 441 matrix correlations that must take place for each mask. This corresponds to 21 search rows, each 21 positions wide. The total number of search row correlations for the sample data and templates is thus 48,384. The behavior of the simulator on the sample dataset revealed a number of algorithm-specific characteristics. Because the design architecture was developed for reconfigurable devices, these characteristics are incorporated to tune the hardware engine for the best cost and performance.

28.3.2 Order of Correlation Tasks

Correlation tasks for threshold calculation (equation 28.2), bright sum (equation 28.3), and surround sum (equation 28.4) are very closely related. Valid results for all three must exist in order to calculate the quality of the hit, so invalid results from any one of them make other calculations unnecessary.

For the data samples, about 60 percent of the surround sums and 40 percent of the threshold results were invalid, while all of the bright sum results were valid. The low rejection rate by bright sum is the result of the threshold being computed using only the bright mask, regardless of the surround mask. The threshold is computed by the same pixels used for computing bright sum, so we find that, for a typical dataset, checking for invalid surround sums before the other calculations drastically reduces the total number of calculations needed.

Zero mask rows

Each mask has 32 rows. However, many have all-zero rows that can be skipped. By storing with each template a pointer to its first nonzero row we can skip directly to that row "for free." Embedded all-zero rows are also skipped.

The simulation tools showed that, for our template set, this optimization significantly reduces the total computational requirements. For the sample template set, there are total of 4608 bitmap rows to use in the correlation tasks. Out of 4608 bright rows, only 2206 are nonzero, and out of 4608 surround rows, 2815 are nonzero. Since the bright mask is used for both threshold and bright sum calculations, and the surround mask is used once, skipping the zero rows reduces the number of row operations from 13,824 to 7227, which produces a savings of about 52 percent.

It is also possible to reduce the computation by skipping zero columns. However, as will be described in following section, the FPGA implementation works on an entire search row concurrently. Hence, skipping rows reduces time but skipping columns reduces the number of active elements that work in parallel, yielding no savings.

28.3.3 Reconfigurable Image Correlator

Although it is possible to reconfigure FPGAs dynamically, the time spent on context switching and reconfiguration could be used instead to process data on a register-based static design. For this reason, minimizing reconfiguration time during computation is essential in effective FPGA use. Nevertheless, when we use FPGAs as compute engines, reconfiguration allows the hardware to take on a large range of task parameters.

The SLD tasks represented in equations 28.1, 28.3, and 28.4 are image correlation calculations on sliding template masks with radar images. To explain our design strategies, we examine each equation by applying the algorithm on a small dataset consisting of a 6 × 6 pixel image, a 3 × 3 mask bitmap, and a 4 × 4 result matrix.

For this dataset, the shape sum calculation for a mask requires multiplying all 9 mask bits with the corresponding image pixels and summing them to find 1 of 16 results. To build an efficient circuit for the sum equations 28.3 and 28.4, we write out the subset of both equations as shown in Table 28.1. By expanding the summation equations, we expose opportunities for hardware to optimize the calculations. First, the same B_{uv} is used to calculate the nth term of all of the shape sum results. Thus, when the summation calculations are done in parallel, the B_{uv} coefficient can be broadcast to all of the units that calculate each result. Second, the image data in the nth term of the SM_{xy} is in the $(n+1)$th term of SM_{xy-1}, except when v returns to 0, the image pixel is located in the subsequent row. This is useful in implementing the pipeline datapath for the image pixels through the parallel summation units.

TABLE 28.1 ■ Expanded sum equations 28.3 and 28.4

Term	1	2	3	4	5	6	7	8	9
u	0	0	0	1	1	1	2	2	2
v	0	1	2	0	1	2	0	1	2
$SM_{00} =$	$B_{00}M_{00}+$	$B_{01}M_{01}+$	$B_{02}M_{02}+$	$B_{10}M_{10}+$	$B_{11}M_{11}+$	$B_{12}M_{12}+$	$B_{20}M_{20}+$	$B_{21}M_{21}+$	$B_{22}M_{22}$
$SM_{01} =$	$B_{00}M_{01}+$	$B_{01}M_{02}+$	$B_{02}M_{03}+$	$B_{10}M_{11}+$	$B_{11}M_{12}+$	$B_{12}M_{13}+$	$B_{20}M_{21}+$	$B_{21}M_{22}+$	$B_{22}M_{23}$
$SM_{02} =$	$B_{00}M_{02}+$	$B_{01}M_{03}+$	$B_{02}M_{04}+$	$B_{10}M_{12}+$	$B_{11}M_{13}+$	$B_{12}M_{14}+$	$B_{20}M_{22}+$	$B_{21}M_{23}+$	$B_{22}M_{24}$
$SM_{03} =$	$B_{00}M_{03}+$	$B_{01}M_{04}+$	$B_{02}M_{05}+$	$B_{10}M_{13}+$	$B_{11}M_{14}+$	$B_{12}M_{15}+$	$B_{20}M_{23}+$	$B_{21}M_{24}+$	$B_{22}M_{25}$

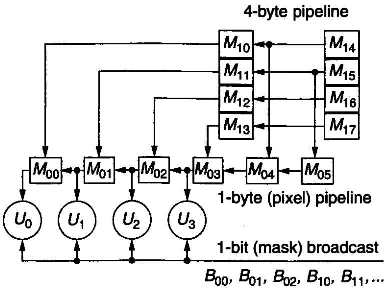


FIGURE 28.8 ■ A systolic image array pipeline.

Based on the characteristics of the expanded equations, we can build a systolic computation unit as in Figure 28.8. To save time while changing the rows of pixels, the pixel pipeline can either operate as a pipeline or be directly loaded from another set of registers. At every clock cycle, each U_y unit performs one operation, v is incremented modulo 3, and the pixel pipeline shifts by one stage (U_1 to U_0, U_2 to $U_1, \ldots$). When v returns to 0, u is incremented modulo 3, and the pixel pipeline is loaded with the entire $(u+x)$th row of the image. When u returns to 0, the results are offloaded from the U_y stage, their accumulators are cleared, and x is incremented modulo 4. When x returns to 0, this computing task is completed.

The initial loading of the image pixel pipeline is from the image word pipeline, which is word wide and so four times faster than the image pixel pipeline. This speed advantage guarantees that the pipeline will be ready with the next image row data when u returns to 0.

28.3.4 Application-specific Computation Unit

Developing different FPGA mappings for equations 28.1, 28.3, and 28.4 in parallel processing unit is one way to implement the design. At the end of each stage, the FPGA device is reconfigured with the optimal structure for the next task. As appealing as this may sound, current FPGA devices have typical reconfiguration times of tens of milliseconds, during which the reconfiguring logic cannot be used for computation.

As presented in Section 28.3, each set of template configurations also has to be designed and compiled before any computation can take place. This can be a time-consuming procedure that does not allow dynamic template sets to be immediately used in the system.

Fortunately, we can rely on the fact that FPGAs can be tuned to target-specific applications. From the equations, we derived one compact structure, shown in Figure 28.9, that can efficiently perform all ATR tasks. Since the target ATR

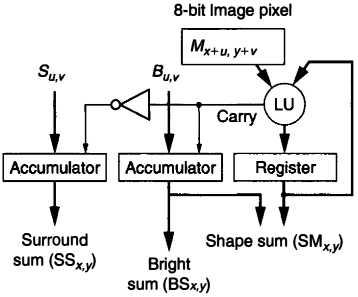


FIGURE 28.9 ■ Computation logic for equations 28.1, 28.3, and 28.4.

system can be seen as "embarrassingly parallel," the performance of the FPGA design is linearly scalable to the number of the application-specific units.

28.4 ATR IMPLEMENTATIONS

In this section we present the implementation results of two reconfigurable Sandia ATR systems, researched and developed on different reconfigurable platforms. Both designs leverage the unique characteristics of reconfigurable devices to accelerate ATR algorithms while making efficient use of available resources. Therefore, they both outperformed existing software as well as custom ASIC solutions. By analyzing the results of the reconfigurable solutions, we examine design trade-offs in cost and performance.

28.4.1 A Dynamically Reconfigurable System

All of the component technologies described in this chapter have been designed, implemented, tested, and debugged using the Mojave board shown in Figure 28.10. This section discusses various performance aspects of the complete system, from abstract template sets through application-specific CAD tools and finally down to the embedded processor and dynamic FPGA. The current hardware is connected to a video camera rather than a SAR data source, though this is only necessary for testing and early evaluation.

The results presented here are based on routing circuits to two devices: the Xilinx 4013PG223-4 FPGA and the Xilinx 4036. Xilinx rates the capacity of these parts as 13K and 36K equivalent gates.

Table 28.2 presents data on the effectiveness of the template-partitioning phase. Twelve templates were considered for this comparison: in one case they were randomly divided into three partitions; in the other, the CAD tool was used to guide the process. The randomly selected partitions required 33 percent more CLBs than those produced by the intelligent partitioning tool. These numbers

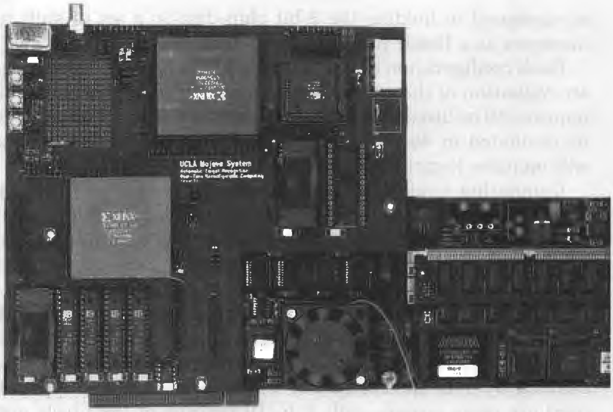


FIGURE 28.10 ■ Photograph of second-generation Mojave ATR system.

Table 28.2 ■ Comparison of scored and random partitioning on an Xilinx 4036

Random grouping CLB count	Initial partitioning CLB count
1961	1491
1959	1449
1958	1487

Table 28.3 ■ Comparison of resources used for the dynamic and static FPGAs

	Flip-flops	Function generators	I/O pins
Dynamic FPGA	532	939	54
Support FPGA	196	217	96
Available	1536	1536	192

account for the hardware requirements of the entire design, including the control hardware that is common to all designs as well as the template-specific adder trees. Relative savings in the adder trees alone are higher.

Table 28.3 lists the overall resources used for both FPGAs in the system, the dynamic devices used for correlation, and the static support device used to implement system control features. Because the image source is a standard video camera rather than a SAR sensor, the surround template is the complement of the bright template, resulting in more hardware than would be required for true SAR templates. The majority of the flip-flops in the dynamic FPGA are assigned to holding the 8-bit chip data in a set of shift registers. This load increases as a linear function of the template size.

Each configuration of the dynamic FPGA requires 16 milliseconds to complete an evaluation of the entire chip for four template pairs. The Xilinx 4013PG223-4 requires 30 milliseconds for reconfiguration. Thus, a total of 4 template pairs can be evaluated in 46 milliseconds, or 84 template pairs per second. This timing will increase logarithmically with the template size.

Comparing configurable machines with traditional ASIC solutions is necessary but complicated. Clearly, for almost any application, a bank of ASICs could be designed that used the same techniques as the multiple configurations of the FPGA and would likely achieve higher performance and consume less power. The principal advantage of configurable computing is that a single FPGA may act as many ASICs without the cost of manufacturing each device. If the comparison is restricted to a single IC (e.g., a single FPGA against a single ASIC of similar size), relative performance becomes a function of the hardware savings enabled by data specificity. For example, in the ATR application the templates used are quite sparse—only 5 to 10 percent of the pixels are important in the computation—which translates directly into a hardware savings that is much more difficult to realize in an ASIC. Further savings in the ATR application are possible by leveraging topological similarities across templates. Again, this is an advantage that ASICs cannot easily exploit.

If the power and speed advantages of ASICs over FPGAs are estimated at a factor of 10, the configurable computing approach achieves a factor of improvement anywhere from 2 and 10 (depending on sparseness and topological properties) for the ATR application.

28.4.2 A Statically Reconfigurable System

The FPGA nodes developed by Myricom integrate reconfigurable computing with a 2-level multicomputer to promote flexibility of programmable computational components in a highly scalable network architecture. The Myricom FPGA nodes and its motherboard are shown in Figure 28.11. The daughter nodes are 2-level multicomputers whose first level provides the general-purpose infrastructure of the Myrinet network using the LANai RISC microprocessor. The FPGA functions as a second-level processor responsible for application-specific tasks.

The host is a SparcStation IPX running SunOS 4.1.3 with a Myrinet interface board having a 512K memory. The FPGA node—consisting of Lucent Technologies' ORCA FPGA 40K and Myricom's LANai 4.1 running in 3.3 V at 40 MHz—communicates with the host through an 8-port Myrinet switch.

Without additional optimization, static implementation of the complete ATR algorithm on one FPGA node processes more than 900 templates per second. Each template requires about 450,000 iterations of 1-bit conditional accumulate for the complete shape sum calculation. The threshold calculation requires one division followed by subtraction. The bright and surround sum compares all the image pixels against the threshold results. Next, 1-bit conditional accumulate is

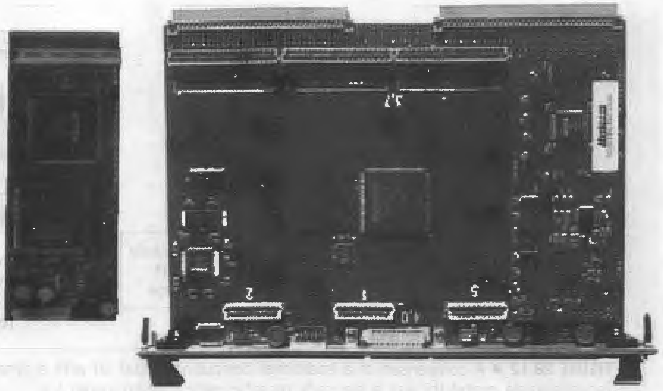


FIGURE 28.11 ■ A Myrinet 8-port switch motherboard with Myricom ORCA FPGA daughter nodes. Four FPGA nodes can be plugged into a single motherboard.

executed for each sum. And then the quality values are calculated using two divides, an add, and a multiply.

Given that 1-bit conditional accumulate, subtract, divide, multiply, and 8-bit compare are one operation each, the total number of 8-bit operations to process one 32 × 32 template over a 64 × 64 image is approximately 3.1 million. Each FPGA node executes over 2.8 billion 8-bit operations per second (GOPS).

After the simulations, we found that the sparseness of the actual templates reduced their average valid rows to approximately one-half the number of total template rows. This optimization was implemented to increase the throughput by 40 percent. Further simulations revealed more room for improvements, such as dividing the shape sum in the FPGA, transposing narrow template masks, and skipping invalid threshold lines. Although these optimizations were not implemented in the FPGA, the simulation results indicated an additional 94 percent increase in throughput. Implementing all optimizations would yield a result equivalent to about a 7.75 GOPS correlator.

28.4.3 Reconfigurable Computing Models

The increased performance of configurable systems comes with several costs. These include the time and bandwidth required for reconfiguration, the memory and I/O required for intermediate results, and the additional hardware required for efficient implementation and debugging. Minimizing these costs requires innovative approaches to system design.

Figure 28.12 illustrates the fundamental difference between a traditional computing model and the two reconfigurable computing architectures discussed in this chapter. The traditional processor receives simple operands from data

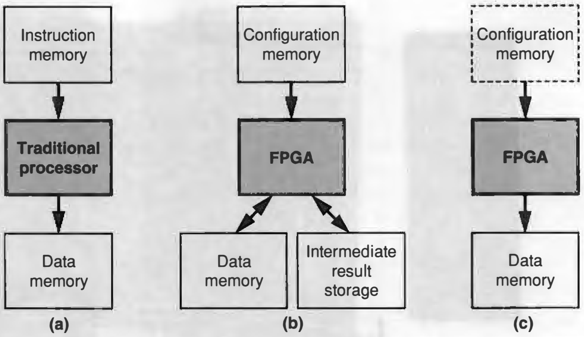


FIGURE 28.12 ■ A comparison of a traditional computing model (a) with a dynamically reconfigurable model (b) and a statically reconfigurable custom model (c).

memory, performs a simple operation in the program, and returns the result to data memory. Similarly, dynamic computing uses a small number of rapidly reconfiguring FPGAs tightly coupled to an intermediate result memory, data memory, and configuration memory. A reconfigurable custom computer is similar to a fixed ASIC device in that, usually, only one highly tuned design is configured on the FPGA—there is no need to reconfigure to perform a needed function.

In most cases, a custom ASIC performs far better than a traditional processor. However, traditional processors continue to be used for their programmability. FPGAs attempts to bridge the gap between custom ASICs and software by allowing designers to build custom hardware using programmable firmware. Therefore, unlike in pure ASIC designs, configuration memory is used to program the reconfigurable hardware as instructions in a traditional processor would dictate the functionality of a program. Unlike software, once the FPGA is configured, it can function just like a custom device.

As shown in previous sections, an ATR was implemented in an FPGA using two different methods. The first implementation uses the dynamic computer model, where parts of the entire algorithm are dynamically configured to produce the final results. The second design uses simulation results to produce a highly tuned fixed design in the FPGA that does not require more than a single reconfiguration. Because of algorithm modifications made to the first design, there is no clear way to compare the two designs. However, looking deeper, we find that there is not a drastic difference in the subcomponents or the algorithm; in fact, the number of required operations for the algorithm in either design should be the same.

The adders make up the critical path of both designs. Because both designs are reconfigurable, we expect the adders used to have approximately the same performance as long as pipelining is done properly. Clever use of adders in the static design allows it to execute more than one calculation simultaneously. However, it is possible to make similar use of the hardware to increase performance in the dynamic design.

The first design optimizes the use of adders to skip all unnecessary calculations, also making each configuration completely custom. The second design has to be more general to allow some programmability. Therefore, depending on the template data, not all of the adders may be in use at all times. If all of the templates for the first design can be mapped onto a single FPGA, the first method results in more resource efficiency than the second. The detrimental effect of idle adders in the static design becomes increasingly more prominent as template bitmap rows grow more sparse.

On the other hand, if the templates do not all fit in a single FPGA, the first method adds a relatively large overhead because of reconfiguration latency. Unfortunately, the customized method of the second design works against making the design smaller. Every bit in the template maps to a port of the adder engine, so the total size of the design is proportional to the number of total bits in all of the templates. Therefore, as the number of templates increases, the total design size must also increase. Ultimately, the design must be divided into several smaller configurations that are dynamically reconfigured to share a single device.

From these results, we observe the strengths and weaknesses of dynamic reconfiguration in such applications. Dynamic reconfiguration allows a large custom design to successfully run in a smaller FPGA device. The trade-off is significant time overhead in the system.

28.5 SUMMARY

Like many streaming image correlation algorithms, the Sandia ATR system discussed in this chapter can be efficiently implemented on an FPGA. Because of the high degree of parallelism in the algorithm, designers can take full advantage of parallel processing in hardware while linearly scaling total throughput with available hardware resources. In this chapter we presented two different ways of implementing such a system.

The first system employs a dynamic computing model to effectively implement a large custom design using a smaller reconfigurable device. To fit, high-performance custom designs can be divided into subcomponents, which can then share a single FPGA to execute parts of the algorithm at a high speed. For the ATR algorithm, this process produced a resource-efficient design that exceeded the performance of previous custom ASIC-based systems.

The second system is based on a more generic architecture highly tuned for a given set of templates. Through extensive simulations, many parameters of the algorithm are tuned to efficiently process the incoming data. With algorithm-specific optimizations, the throughput of the system increased threefold from an initial naive implementation. Because of the highly pipelined structure of the design, the maximum clock frequency is more than three times that of the dynamic computer design. Furthermore, a larger FPGA on the platform allowed the generic processing architecture to duplicate the specifications of the original algorithm. Therefore, the raw performance of the static design was faster than the dynamically reconfigurable system.

Although the second system is a static design, it is best suited for reconfigurable platforms because of its highly tuned parameters. Since this system is reconfigurable, it is conceivable that the dynamic computational model can be applied on top of it. Thus, the highly tuned design may be implemented efficiently, even on a device with enough resources for only a fraction of the entire design.

Acknowledgments I would like to acknowledge Professor William H. Mangione-Smith for permission to integrate publications on the Mojave project into this chapter.

References

[1] P. M. Athanas, H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer* 26, 1993.

[2] J. G. Eldredge, B. L. Hutchings. Run-time reconfiguration: A method for enhancing the functional density of SRAM-based FPGAs. *Journal of VLSI Signal Processing* 12, 1996.

[3] J. Villasenor, W. H. Mangione-Smith. Configurable computing. *Scientific American* 276, 1997.

[4] E. Mirsky, A. DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, 1996.

[5] R. Razdan, M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 172–180, 1994.

[6] G. Estrin. Organization of computer systems—the fixed plus variable structure computer. *Proceedings of the Western Joint Computer Conference*, 1960.

[7] M. Shand, J. Vuillemin. Fast implementations of RSA cryptography. *Proceedings of the Symposium on Computer Arithmetic*, 1993.

[8] K. W. Tse, T. I. Yuk, S. S. Chan. Implementation of the data encryption standard algorithm with FPGAs. *Proceedings of International Symposium on Field-Programmable Logic and Applications*, 1993.

[9] J. Leonard, W. H. Mangione-Smith. A case study of partially evaluated hardware circuits: Key-specific DES. *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications* 1304:151–160, 1997.

[10] P. M. Athanas, A. L. Abbott. Real-time image processing on a custom computing platform. *IEEE Computer* 28, 1995.

[11] J. G. Eldredge, B. L. Hutchings. Density enhancement of a neural network using FPGAs and run-time reconfiguration. *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, 1994.

[12] J. G. Eldredge, B. L. Hutchings. RRANN: The run-time reconfiguration artificial neural network. *Proceedings of the Custom Integrated Circuits Conference*, 1994.

[13] B. Schoner, C. Jones, J. Villasenor. Issues in wireless coding using run-time-reconfigurable FPGAs. *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, 1995.

[14] C. Chou, S. Mohanakrishnan, J. B. Evans. FPGA implementation of digital filters. *Proceedings of the Fourth International Conference on Signal Processing Applications and Technology*, pp. 80–88, 1993.

[15] G. Estrin, B. Bussell, R. Turn, J. Bibb. Parallel processing in a restructurable computer system. *IEEE Transactions on Electronic Computers* EC-12(5):747–755, December 1963.

[16] G. Estrin, R. Turn. Automatic assignment of computations in a variable structure computer system. *IEEE Transactions on Electronic Computers* EC-12(6):755–773, December 1963.

[17] M. J. Wirthlin, B. L. Hutchings. Improving functional density through run-time constant propagation. *Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays*, 1997.

[18] P. Lee, M. Leone. Optimizing ML with run-time code generation. *Proceedings of Programming Language Design and Implementation*, 1996.

[19] D. R. Engler, T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Proceedings of the Sixth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1994.

[20] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, Ph.D. thesis, Columbia University, Department of Computer Science, 1992.

[21] W. H. Mangione-Smith, B. Hutchings. Configurable computing: The road ahead. *Proceedings of the Reconfigurable Architectures Workshop*, 1997.

[22] P. Bertin, H. Touati. PAM programming environments: Practice and experience. *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 1994.

[23] Y. H. Cho. Optimized automatic target recognition algorithm on scalable Myrinet/field programmable array nodes. *Thirty-fourth IEEE Asilomar Conference on Signals, Systems, and Computers*, October 2000.

[24] K. N. Chia, H. J. Kim, S. Lansing, W. H. Mangione-Smith, J. Villasenor. High-performance automatic target recognition through data-specific very large scale integration. *IEEE Transactions on Very Large Scale Integration Systems* 6(3), 1998.

[25] J. Villasenor, B. Schoner, K. N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, W. H. Mangione-Smith. Configurable computing solutions for automatic target recognition. *Proceedings of the IEEE International Symposium on FPGAs for Custom Computing Machines*, April 1996.

[26] R. Sivilotti, Y. Cho, D. Cohen, W. Su, B. Bray. Scalable network based FPGA accelerators for an automatic target recognition application. *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[27] R. Sivilotti, Y. Cho, W. Su, D. Cohen. *Scalable, Network-connected, Reconfigurable, Hardware Accelerators for an Automatic-Target-Recognition Application*, Myricom technical report, May 1998.

[28] R. Sivilotti, Y. Cho, W. Su, D. Cohen. *Myricom's FPGA-based Approach to ATR/SLD*, DARPA ACS PI meeting slide presentation, November 1997.

[29] R. Sivilotti, Y. Cho, W. Su, D. Cohen. Production-quality, LANai-4-based quad-FPGA-node VME boards. *http://www.myri.com/research/darpa/97a-fpga.html*, October 1997.

[30] C. L. Seitz, Tactical network and multicomputer technology. *http://www.myri.com/research/darpa/index.html*, March 1997, July 1997, August 1998.

[31] C. L. Seitz. Two-level-multicomputer project: Summary. *http://www.myri.com/research/darpa/96summary.html*, July 1996.

[32] W. C. Athas, L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer* 21, 1988.

[33] M. Shand, J. Vullemin. Fast implementations of RSA cryptography. *Proceedings of 11th Symposium on Computer Arithmetic*, 1993.

[34] J. G. Eldredge, B. L. Hutchings. RRANN: The run-time reconfiguration artificial neural network. *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1994.

[35] Xilinx, Inc. *RAM-based Shift Register v9.0, LogiCORE Datasheet*, Xilinx, Inc., July 13, 2006.

CHAPTER 29

Boolean Satisfiability: Creating Solvers Optimized for Specific Problem Instances

Peixin Zhong
Department of Electrical and Computer Engineering
Michigan State University
Margaret Martonosi, Sharad Malik
Department of Electrical Engineering
Princeton University

Boolean satisfiability (SAT) is a classic NP-complete problem with a broad range of applications. There have been many projects that use reconfigurable computing to solve it. This chapter presents a review of the subject with emphasis on a particular approach that employs a backtrack search algorithm and generates solver hardware according to the problem instance. This approach utilizes the reconfigurability and fine-grained parallelism provided by FPGAs.

The chapter is organized as follows: Section 29.1 is an introduction to the SAT formulation and applications. Section 29.2 describes the algorithms to solve the SAT problem. Sections 29.3 and 29.4 describe in detail two SAT solvers that use reconfigurable computing, and Section 29.5 provides a broader discussion.

29.1 BOOLEAN SATISFIABILITY BASICS

The Boolean satisfiability problem is well known in computer science [1]. Given a Boolean formula, the goal is to find an assignment to the variables so that the formula evaluates to true or 1 (it satisfies the formula), or to prove that such an assignment does not exist (the formula is not satisfiable). It has many applications, including theorem proving [5], automatic test pattern generation [2], and formal verification [3,4].

29.1.1 Problem Formulation

The Boolean formula in an SAT problem is typically represented in conjunctive normal form (CNF), also known as product-of-sums. Each sum of literals is called a clause. A literal is either a variable or the negation of a variable, denoted with a negation symbol or a bar (such as $\neg v_1$ or $\bar{v}_1$). Equations 29.1 and 29.2 are examples of simple CNFs.

$$(v_1+v_2+v_3)(\bar{v}_1+v_2+v_3)(v_1+v_2+\bar{v}_3)(\bar{v}_2+\bar{v}_3) \tag{29.1}$$

or

$$(v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_2 \vee \neg v_3) \qquad (29.2)$$

Each sum term, such as $(v_1 + v_2 + v_3)$, is a clause. In the clause, v_1 or $\neg v_1$ is called a literal. It can be easily tested that $v_1 = 1, v_2 = 1, v_3 = 0$ is a solution to the problem.

The SAT clauses represent implication relationships between variables. To satisfy the CNF, each clause should be satisfied (i.e., at least one literal in each clause should be 1). For a given partial assignment, if only one literal in a clause is not assigned but all others are assigned to 0, the unassigned literal is implied to be 1 to satisfy the clause. The first clause in equation 29.1 contains three possible implications. If $v_1 = 0$ and $v_2 = 0, v_3$ is implied to be 1, denoted as $\neg v_1 \neg v_2 \supset v_3$. Similarly, $v_1 = 0$ and $v_3 = 0$ imply $v_2 = 1$, and $v_2 = 0$ and $v_3 = 0$ imply $v_1 = 1$. Such implications can be used to construct powerful logic expressions. They are also the key to SAT-solving algorithms.

29.1.2 SAT Applications

The many applications of SAT include test pattern generation [2] and model checking [3, 4]. The logic relations of a digital circuit can also be represented in SAT CNF. Each logic gate is represented by a group of clauses, with each signal represented by a variable with two possible values, 1 or 0. A circuit is represented by a conjunction of clauses representing all gates in the circuit. What follows is the transformation from simple gates to clauses:

AND gate, $z <= ab$, maps to $(a + \neg z)(b + \neg z)(\neg a + \neg b + z)$
NAND gate, $z <= \neg(ab)$, maps to $(a + z)(b + z)(\neg a + \neg b + \neg z)$
OR gate, $z <= a + b$, maps to $(\neg a + z)(\neg b + z)(a + b + \neg z)$
NOR gate, $z <= \neg(a + b)$, maps to $(\neg a + \neg z)(\neg b + \neg z)(a + b + z)$
XOR gate, $z <= a \oplus b$, maps to $(\neg a + \neg b + \neg z)(\neg a + b + z)(a + \neg b + z)(a + b + \neg z)$
Buffer gate, $z <= a$, maps to $(\neg a + z)(a + \neg z)$
Inverter gate, $z <= \neg a$, maps to $(a + z)(\neg a + \neg z)$

SAT can be used in test pattern generation or to verify the equivalence of two combinational circuits. The circuit construction is shown in Figure 29.1. In equivalence checking, the two representations of the circuit are fed with the same primary inputs signals, and the corresponding primary outputs feed into an exclusive-or (XOR) gate. If an assignment of primary inputs can be found such that any of the XOR gates has 1 as an output, the circuits are different. If no such assignment can be found, the circuits are functionally identical.

For test pattern generation, instead of using two representations of one circuit, we use two copies of the same circuit. However, one copy has a fault introduced into the design, which we can detect by searching for some pattern of inputs. In this case any input pattern that can generate a 1 on an XOR output is a test for that fault. If no such assignment is possible, that fault is untestable.

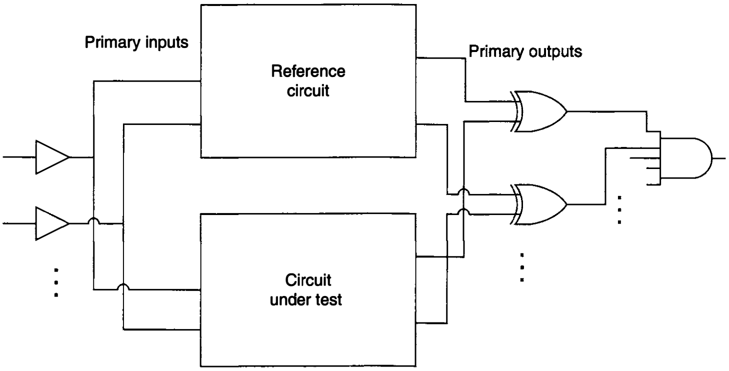


FIGURE 29.1 ■ Test pattern generation.

29.2 SAT-SOLVING ALGORITHMS

29.2.1 Basic Backtrack Algorithm

There are many algorithms to solve the SAT problem. They can be divided into two categories: complete and incomplete. A complete algorithm guarantees either to find a solution on termination or to prove that there is no solution. Complete algorithms typically employ a methodical search of the variable assignment space. For hard problems, the runtime may well exceed acceptable levels. An incomplete algorithm does not guarantee to find the solution and typically involves greedy or randomized search [22]. It can often find a solution of an easy problem very quickly, but if it fails to do so within a given time, it does not prove that no solution exists. Many applications require a complete algorithm to provide a definite answer, so this chapter concentrates on such algorithms for SAT.

An early SAT algorithm was proposed by Davis and Putnam [5]. Like theirs, most complete SAT algorithms are based on backtrack search [6–9], which is similar to depth-first search in traversing a tree. The pseudo-code of the basic algorithm, shown in Figure 29.2, starts with an empty variable partial assignment (i.e., every variable value is assumed to be unknown, or free). The search level is increased by branching—that is, assigning a value for a free variable. The algorithm checks if the incremented partial assignment can be part of a solution. If not, we say a conflict is detected. If there is no conflict, the algorithm will choose another free variable and branch on it; if a conflict is detected, it will backtrack to the most recently assigned variable and choose the opposite value. All decisions made after that backtrack point will be undone.

```
Solve_SAT()
{
  assign all variables to unknown;
  while (true) {
    if (implications force an unknown variable to a specific value)
      set that variable to that specific value;
    if (the current assignment has a conflict) {
      undo all implications and branches up to most recent untoggled branch;
      if (all branches undone)
        return No_solution;
      toggle value assigned to the variable of last untoggled branch;
    }
    if (no unassigned variables remain)
      return Solved;
    } else {
      start new branch by assigning a value to the next free variable;
    }
  }
}
```

FIGURE 29.2 ■ The basic backtrack algorithm to solve SAT.

The algorithm has two possible terminating conditions. If all variables values are known and the formula is satisfied, a solution is found. If all branches fail to find a solution and the algorithm must backtrack beyond the first branch variable, there is no solution and the formula is unsatisfiable.

The key to the efficiency of the backtrack algorithm is effectively pruning the search space. Early detection of a conflict assignment avoids useless searches along this branch. The following are some basic rules and techniques used in the algorithm. At each stage of the search, a variable can have one of three possible values: 1, 0, and free (unassigned).

1. If at least one literal of a clause evaluates to 1, this clause is satisfied. There is no need to check other literals in the clause.
2. If all literals of a clause evaluate to 0, the partial assignment is a conflict and cannot be part of the solution.
3. If only one literal of a clause is free and all other literals evaluate to 0, the free literal is implied to be 1. This is called unit resolution or implication. Implication is a powerful mechanism because it can deduce implied values of variables not yet branched on. However, it can create another case of conflict if a variable is implied by two clauses to be of opposite values.
4. If all of the literals of a free variable in the as yet unsatisfied clauses are all of the same polarity (i.e., inverted or not inverted), a value can be chosen for this variable that safely satisfies these clauses.
5. Because the variable ordering of branches has a large impact on the efficiency of the algorithm, different dynamic or static ordering schemes have been investigated. A simple heuristic orders the variables based on the number of clauses they appear in. A variable with the most appearances

often has more influence than others. Therefore, branching on it early typically prunes the search space more quickly.

A basic algorithm can use a static variable ordering. It can also use a fixed branching scheme, such as always branching with value 1, in which, after each branch or backtrack, implication is checked exhaustively. This basic algorithm corresponds to the reconfigurable SAT solver described in Section 29.2.

29.2.2 Improving the Backtrack Algorithm

Among the advanced features explored to further improve the efficiency of the backtrack search algorithm [6, 7], an effective one is learning based on conflict analysis. With the search algorithm moving back and forth by branching and backtracking, similar spaces are explored many times. Consider a problem, as in equation 29.3, where some of the clauses are

$$(\neg v_i + v_j + v_k)(\neg v_i + v_j + \neg v_k)(\neg v_i + \neg v_j + v_k)(\neg v_i + \neg v_j + \neg v_k)\ldots\ldots \qquad (29.3)$$

The variable v_i is branched to be 1, and many other variables may have been tested before v_j is branched on. When v_j is branched on and 1 is tested, a conflict on v_k is detected. Then v_j is switched to 0, which again causes a conflict. Thus, the algorithm will backtrack to the previous branch variable. However, switching variable assignments other than v_i will not help. The algorithm may reenter the same region many times before it backtracks to v_i. Conflict analysis would be helpful in this situation.

A new variable value is implied by the value choices of all other literals in this clause being 0. Each literal has obtained its value either from branch decisions or from earlier implications. Therefore, we can create a transitive implication graph where an implied variable is ultimately implied by a set of branch decisions. A conflict is detected when a variable is implied to be of opposite values. It can be identified by backtracking the implication graph to identify the complete set of branch assignments that led to it. This set of decisions is responsible for the conflict.

In the example just given, the first conflict is caused by $v_i = 1$ and $v_j = 1$. A new clause can be derived as $(\neg v_i + \neg v_j)$. This is a redundant clause that can be added to the formula without changing the solution. It can also be viewed as applying the following consensus theorem to clauses 3 and 4 in equation 29.4:

$$(x + y)(\neg x + z) = (x + y)(\neg x + z)(y + z) \qquad (29.4)$$

With the conflict on $v_j = 1$ detected, it can be interpreted as v_j is implied to be 0. In this case, it is implied by $v_i = 1$. Another round of implication will render a conflict because of the first two clauses in the original formula. From the second conflict, a new clause can be derived as $(\neg v_i + v_j)$. Combined with the conflict analysis result of the previous conflict, the resulting clause is $(\neg v_i)$, which dictates $v_i = 0$.

The algorithm should instead directly backtrack to v_i, in what is called nonchronological backtracking by Marques-Silva and Sakallah [6]. The new clause can be added to the problem and thus help prune the future search space.

This example is extremely simple, but the principle is applicable to all conflicts and can reduce runtime by several orders of magnitude on many problems. For example, for the AIM200 group of problems, GRASP takes 10.8 seconds, whereas many other SAT solvers take more than 10,000 seconds. However, because of the heuristic nature of the algorithms, they show different performance characteristics with different problems.

Learning also has its trade-offs. Every conflict will generate one redundant clause, and storage will explode if every such clause is recorded permanently. Heuristics for discarding long or unused redundant clauses can keep the storage size manageable and still achieve significant speedup.

29.3 A RECONFIGURABLE SAT SOLVER GENERATED ACCORDING TO AN SAT INSTANCE

This section presents an example of generating an SAT solver according to the SAT instance [10–12]. That is, instead of creating a generic, hardware SAT solver, we generate a new configuration for the reconfigurable computing machine for each SAT equation being solved.

29.3.1 Problem Analysis

A hard SAT problem can take a very long time to solve, limiting the application of the formula and the solvers' powerful formalism. Therefore, we will look at the use of reconfigurable computing techniques to accelerate SAT solutions. For this it is necessary to compare the relative merit of FPGAs and CPUs and look at the characteristics of SAT algorithms to identify an efficient solution.

FPGAs allow the full customization of control and datapaths. In particular, they make it efficient to perform bit-level operations. Also, by allocating more computing resources for bottleneck operations, they can provide massive parallelism and deep pipelining for suitable applications. However, FPGA clock rates are lower than those for microprocessors of the same technology generation, so raw chip performance may suffer.

Two opportunities for parallel processing in the SAT algorithm stand out, one of which is the parallelism in the vast search space. For a problem with n variables, there are 2^n possible assignments (though with the backtrack algorithm pruning the search space, that number is actually much smaller). It is possible to split at the branch choices and allocate each subspace to its own processor. However, because the search space is typically unbalanced, such parallelization requires rebalancing the load and this would be very complex to implement in hardware. Another source of potential performance gain is implication and conflict checking. Whenever a new value is assigned to a variable, all clauses containing the variable should be checked for implication and conflict. New implied values will trigger further checking and implication. Additionally, the variables are Boolean and suitable for low-level processing by logic circuits, and thus implication and conflict checking are good candidates for hardware acceleration. It has also been confirmed through software profiling that implication and conflict checking take up the majority of computing time.

The basic backtrack search includes branch, implication, and backtrack functions, which are relatively simple and can be implemented with finite-state machines. Many projects implement a full SAT solver on one or multiple FPGAs. The next section describes one of them.

29.3.2 Implementing a Basic Backtrack Algorithm with Reconfigurable Hardware

Since implication and conflict checking are time-consuming processes, they are good candidates for hardware acceleration. Checking all clauses in parallel is one approach enabled by reconfigurable computing techniques. The circuit used for such parallel checking is presented as follows.

During the search, a variable can take one of three possible values: unknown, 1 (true), and 0 (false). A 2-bit encoding, denoted $(v, \bar{v})$, is used for the three variable values because it can conveniently represent them: (0, 0) is an unknown (free) variable; (1, 0) is value 1; and (0, 1) is value 0. The fourth combination, (1, 1), is used for conflict. The 2-bit encoding can be easily used for implication as well. For example, a clause with three literals $(v_i + \neg v_j + v_k)$ represents three possible implications that can be expressed with the 2-bit encoding as logical assignments, as shown in equation 29.5:

$$\begin{aligned} v_i &<= v_j \bar{v}_k \\ \bar{v}_j &<= \bar{v}_i \bar{v}_k \\ v_k &<= \bar{v}_i v_j \end{aligned} \tag{29.5}$$

When a literal appears in multiple clauses, its value is 1 if any one of the clauses implies it to be 1. The general form can be written as

$$\begin{aligned} v_{inew} &<= \sum_{\substack{\textit{each clause } v_i\\ \textit{appears in}}} \Big(\prod_{\substack{\textit{each uninverted}\\ \textit{literal } v_k}} \bar{v}_k \prod_{\substack{\textit{each inverted}\\ \textit{literal } \neg v_l}} v_l \Big) \\ \bar{v}_{inew} &<= \sum_{\substack{\textit{each clause } \bar{v}_i\\ \textit{appears in}}} \Big(\prod_{\substack{\textit{each uninverted}\\ \textit{literal } v_k}} \bar{v}_k \prod_{\substack{\textit{each inverted}\\ \textit{literal } \neg v_l}} v_l \Big) \end{aligned}$$

The summation $\sum$ is a logic OR over the set of clauses in which the implied literal appears. The production $\prod$ is a logic AND over all other literals in the clause. Note that the literal in the formula is inverted from the one in the clause, meaning that the implication is effective if and only if all other literals are known to be 0. With this formula, a complete CNF can be converted to circuits that evaluate all possible implications in parallel.

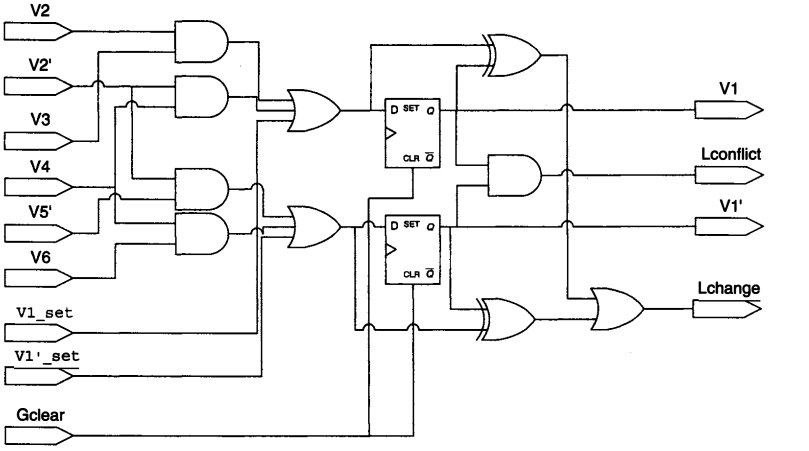


FIGURE 29.3 ■ The implication circuit for one variable, V1.

The implication circuit for V1, shown in Figure 29.3, corresponds to the partial CNF of $(v_1 + \neg v_2 + \neg v_3)(v_1 + v_2 + \neg v_4)(\neg v_1 + v_2 + v_5)(\neg v_1 + \neg v_4 + \neg v_6)$, and is directly derived from the implication equation. A variable may assume a value because of either a branch decision or implication. An OR gate adds the assigned value. Since a newly implied variable may take part in generating new implications, registering the newly implied values allows implication to propagate one level in each clock cycle and avoids combinational cycles. To determine when implications have settled, an XOR gate checks the difference between the current and the next value. An AND gate checks if both literals of a variable are assigned to 1. If such a situation exists, the conflict (also called contradiction) signal is raised.

The other part of the algorithm is the control for the backtrack search. A distributed control architecture is used, with each finite-state machine (FSM) controlling one variable. Using a predetermined variable ordering, the architecture can be implemented by a linear array of communicating FSMs, as shown in Figure 29.4. Other than a few global signals, each FSM communicates only with the two neighboring FSMs. During the SAT-solving process, only one variable is active in terms of branching and backtracking. Its active status is represented by an active token. Two wires connect each pair of FSMs to pass the active token back and forth. Only one variable is the owner of the token at any given time.

In addition to the basic clock and reset signals, there are three global control signals. `Gconflict` is asserted when a conflict is detected. It is the wide OR function of all local conflicts, `Lconflict`. A local conflict is asserted when both

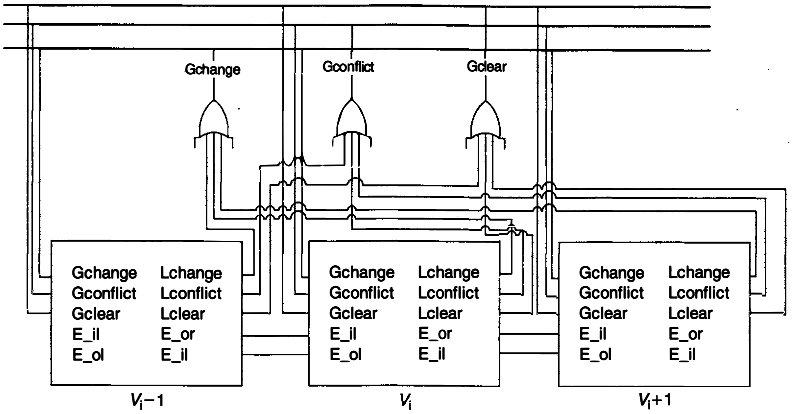


FIGURE 29.4 ■ The global topology for a basic SAT solver circuit.

v_i and $\bar{v}_i$ are assigned or implied to be 1. Gchange is asserted when any variable has changed value. It is the wide OR function of all local changes, Lchange. A local change is asserted when v_{inew} is different from v_i or when $\bar{v}_{inew}$ is different from $\bar{v}_i$. Gclear tells each state machine to clear the implied values. It is issued when the algorithm needs to backtrack and erase earlier implications.

With the external interface defined, each FSM should hold the assigned value, the implied value, and its state of backtrack search. The state machine is designed as registers for the implied value and an FSM combining the assigned value and state in the backtrack search. The state diagram of the latter FSM, shown in Figure 29.5, contains five states:

- *Idle:* This is the initial state, in which the internal variable value is (0, 0). The FSM will stay in the idle state unless it has received the active token from its neighbor through branching or backtracking. When the token is received, if this variable already has an implied value, there is no need to branch, and the FSM will simply pass the token to the next variable at the next clock. If this variable has no implied value and the token has been passed from the left, it will branch and choose the branch value as 1 (the active 1 state).
- *Active 1:* This state is the result of branching from the idle state, in which the variable value is chosen to be 1. The new value will be available for implication and conflict checking. The FSM will keep the token until there is no more change or until a conflict is detected. In the case of no conflict, it will pass the token to the right and will transition to the

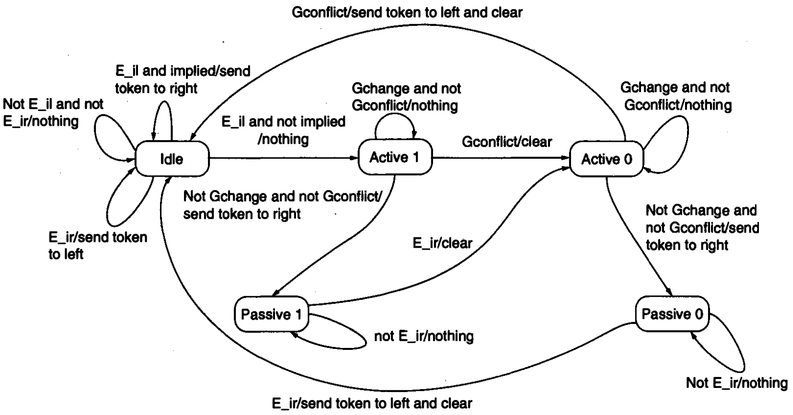

Gconflict/send token to left and clear
E_il and implied/send token to right
Not E_il and not E_ir/nothing
Gchange and not Gconflict/nothing
Gchange and not Gconflict/nothing
E_il and not implied /nothing
Gconflict/clear
Idle
Active 1
Active 0
Not Gchange and not Gconflict/ send token to right
E_ir/send token to left
E_ir/clear
Not Gchange and not Gconflict/send token to right
Passive 1
Passive 0
not E_ir/nothing
Not E_ir/nothing
E_ir/send token to left and clear

FIGURE 29.5 ■ The FSM associated with one variable.

passive 1 state. If a conflict is detected, it will transition to the active 0 state and restart the implication and conflict checking.

- *Active 0:* This state is the result of a conflict in the active 1 state or of the token being passed to passive 1 by backtracking. The variable value is set to 0. Implication and conflict are checked. If there is no conflict, the FSM passes the token to the right and transitions to passive 0. If there is a conflict, it will transition to the idle state and pass the token to the left.
- *Passive 1:* This state is the result of branching further from active 1. If the FSM receives a token from the right because of backtracking, it will transition to active 0.
- *Passive 0:* This state is the result of branching further from active 0. If the FSM receives a token from the right because of backtracking, it will transition to idle and pass the token to the left.

With these FSMs logically forming a linear chain, the branching of the algorithm corresponds to passing the token to the right and performing implications during the process. When a conflict is detected, backtracking is needed. Backtrack switches a value from 1 to 0. If it is already 0, the token is passed to the left. Whenever a conflict is detected, all of the implied values are cleared by the global clear signal and reset to free. The termination condition is easy to test: If the token is passed to the left of the first variable, the problem is unsatisfiable; if it is passed to the right of the last variable, a solution has been found. In addition to the regular problem-solving mode, the linear chain of variables can also be configured as a shift register. When a solution is found, it can be shifted out as a bitstream.

At the time of the design of this SAT solver (1997–1998), a single FPGA chip provided a very limited number of logic gates, and so for typical problems a multi-FPGA solution was needed. The algorithm was implemented on an IKOS (now part of Mentor Graphics) VirtualLogic SLI Emulator, which contained one to six FPGA boards, each containing 64 Xilinx XC4013E FPGA chips to form an 8 × 8 mesh. Thus, it provided the logic capacity to handle a midsize to large SAT problem. While the FPGA itself could support a clock rate of about 20 MHz, the Ikos system used a time-multiplexing I/O scheme called VirtualWire to overcome the pin limitation (see Section 6.4). Thus, the system clock rate was reduced to the 1-MHz range. An HP logic analyzer/function generator was connected to provide the initial input signal and collect the result.

To provide perspective, in 1992 the mainstream FPGA XC4013E had 1368 logic cells. In 2006, the large XC4VLX200 FPGA had 200,448 logic cells (i.e., about 146 times the logic capacity), which was more than what two big Ikos boards could provide.

To solve an SAT problem on this platform, the following steps are needed:

1. *Generate VHDL*. A software tool written in C++ reads in the problem CNF file and generates the VHDL code that models the SAT solver circuit. The FSM is manually coded in VHDL and reused for each SAT problem.
2. *Compile the FPGA*. The VHDL is compiled to bitstream files for programming the FPGAs. For a single FPGA implementation, this can be done by the FPGA tools. For the Ikos emulator, in contrast, this process takes three steps: (1) the design is synthesized into a netlist and partitioned to multiple FPGAs by the IKOS tool; (2) the partitioned netlist is generated; and (3) the netlist is compiled by Xilinx tools into bitstream files. The main function of the Xilinx tools is placement and routing.
3. *Configure the FPGA*. The bitstream is downloaded to the FPGA board, and the FPGA is configured with these files.
4. *Run the problem solver in the FPGA and load the result*. The logic analyzer/function generator creates the initial signals to start the computation. When the problem is solved, the solution is shifted out, where it can be captured by a logic analyzer.

The runtime performance of the FPGA SAT accelerator is shown in Figure 29.6 as a histogram of speedup ratios. This test was carried out in 1998 using the problem set from the DIMACS SAT challenge benchmark. The software runtime basis was obtained by running GRASP with parameter settings close to those of the basic backtrack algorithm. GRASP was run on a Sun 5 workstation with a 110-MHz processor and 64 MB of RAM. The hardware performance was normalized to a 1.33-MHz system clock rate, which is representative of implementations on the IKOS emulator. In the figure, the x-axis is the ratio of software solver runtime to reconfigurable hardware runtime. It does not include the compilation time and the time to configure the FPGAs.

As we see from Figure 29.6, the result indicates that even though the reconfigurable solution has a clock rate 82 times slower than that of the microprocessor-based system, it can still achieve 20 times or greater speedup

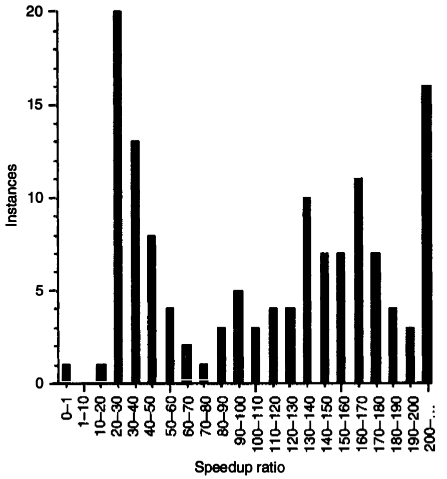


FIGURE 29.6 ▪ A performance comparision of the FPGA SAT accelerator and the software version implementing the same algorithm as hardware.

for many problems. It should be noted that the comparison is based on runtime alone. The reconfigurable approach suffers from compilation overhead, which in 1998 required hours to perform logic synthesis and placement and route for the FPGAs. Current FPGA tools can perform such compilation within a minute. Ways to ameliorate compilation issues will be discussed in later sections.

For an understanding of the speedup results, Table 29.1 shows the speedup ratios for different problems. The average number of clause evaluations per cycle serves as a rough measure of the utilization of parallelism. It is defined as the number of clauses that contain at least one literal from the variables newly assigned in the previous clock cycle. There is a correlation between parallelism in clause evaluation and speedup ratio. Another factor in the speedup is that custom hardware effectively reduces a complex operation into single-cycle implication.

29.3.3 Implementing an Improved Backtrack Algorithm with Reconfigurable Hardware

The example in the previous section shows the performance benefit of reconfigurable computing. However, the hardware solution was implemented with the

TABLE 29.1 ■ Speedup ratios for different problems

Problem	Number of clauses	Average clause evaluations/cycle	Clock rate (MHz)	Speedup ratio
aim-50-2_0-yes1-2	100	7.1	1.78	44.5
aim-100-2_0-yes1-4	200	8.4	0.95	20.9
aim-200-6_0-yes1-1	1200	62.3	0.92	101
dubois20	160	8.0	1.78	13.9
hole7	204	18.3	1.78	44.5
hole8	297	21.9	1.78	45.6
hole9	415	25.9	1.57	40.2
hole10	561	30.1	1.48	41.4
ii8a2	800	15.8	1.07	923
par-8-1-c	254	29.4	1.57	174
par-16-1-c	1264	60.4	0.99	153
pret60_40	160	8.5	2.05	39
ssa0432-003	1027	11.0	0.95	24.7

basic backtrack algorithm, and improvements to the algorithm have brought thousands of times speedup in the software solution. The following example shows a more sophisticated backtrack algorithm with reconfigurable computing. As demonstrated by GRASP, conflict analysis helps identify the true reasons for conflict. Nonchronological backtracking and learning based on the analysis can greatly improve search efficiency.

Knowing that the hardware can perform fast implication checking, an alternative to conflict analysis-based backtracking was developed through trial assignments. When a conflict is detected, there are two possible scenarios regarding the most recently assigned variable. In the first, the variable has just been assigned by branching—it will be assigned the alternative value and tested. In the second, the variable has been assigned to an alternative value because of previous conflicts, so backtracking is needed. GRASP shows that conflict analysis can identify the reasons for conflict and may backtrack multiple levels, saving search time.

In the reconfigurable hardware approach, trial backtrack is performed. The algorithm moves back one decision level at a time and flips the assigned variable. Unlike a real backtrack, the most recent assignment is not turned to unknown. Instead, two implication/conflict tests are run for both value 0 and value 1. If both lead to conflict, we can trial-backtrack another level. If either case leads to no conflict, we have seen the real backtrack destination and the search reverts to regular search mode. This leads to much improved performance, with the only drawback being an increase in finite-state machine complexity.

Figure 29.7 is a diagram of the state machine for this enhanced algorithm. It is an extension of the basic backtrack algorithm, but with nine states instead of five.

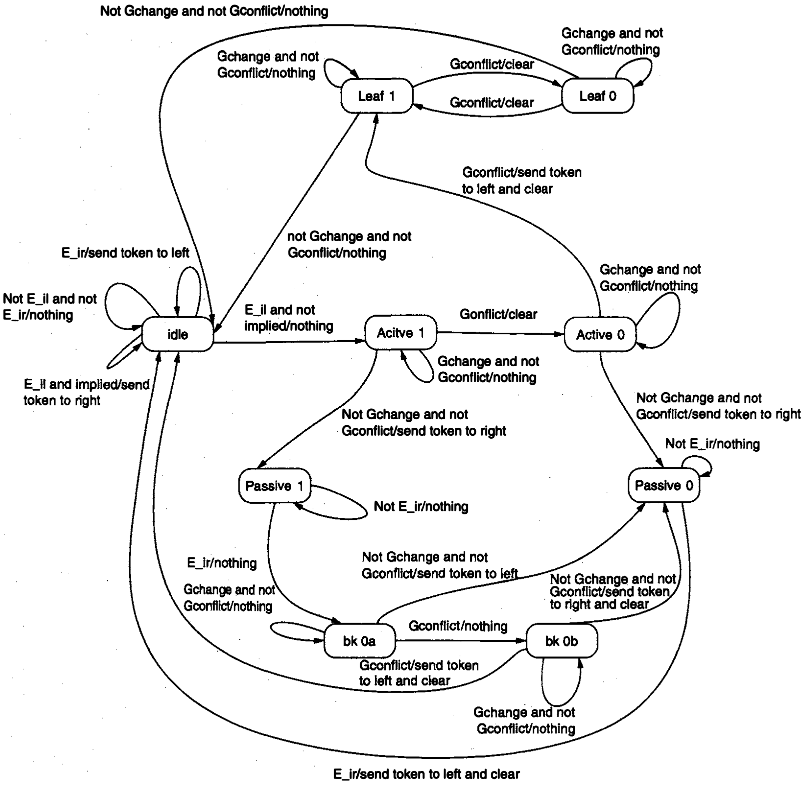

Not Gchange and not Gconflict/nothing
Gchange and not Gconflict/nothing
Gchange and not Gconflict/nothing
Gconflict/clear
Leaf 1
Gconflict/clear
Leaf 0
Gconflict/send token to left and clear
not Gchange and not Gconflict/nothing
E_ir/send token to left
Gchange and not Gconflict/nothing
Not E_il and not E_ir/nothing
E_il and not implied/nothing
Gonflict/clear
idle
Acitve 1
Active 0
Gchange and not Gconflict/nothing
E_il and implied/send token to right
Not Gchange and not Gconflict/send token to right
Not Gchange and not Gconflict/send token to right
Not E_ir/nothing
Passive 1
Passive 0
Not E_ir/nothing
E_ir/nothing
Not Gchange and not Gconflict/send token to left
Gchange and not Gconflict/nothing
Not Gchange and not Gconflict/send token to right and clear
Gconflict/nothing
bk 0a
bk 0b
Gconflict/send token to left and clear
Gchange and not Gconflict/nothing
E_ir/send token to left and clear

FIGURE 29.7 ■ A state diagram of the improved algorithm.

- *Idle:* This is the state before branch; it is also the state if the value is already determined by implication.
- *Active 1:* This is the state after branch on value 1.
- *Active 0:* This is the state after backtrack on the branched value 1. When a conflict is detected, instead of a simple backtrack, a new phase of testing is added. It passes the token to the left and transitions to leaf 1.
- *Passive 1:* The variable value is 1 because of branching, and active control has been passed to the right in branching.
- *Passive 0:* The variable value is 0 because of backtracking, and active control has been passed to the right in branching.

- *Leaf 1:* Leaves 1 and 0 are testing states after conflict is detected with value 0. If the testing settles with no conflict, we have found the most recent branch assignment that contributes to the conflict. The FSM will backtrack directly to that variable. If a conflict is detected, it will try a 0 value in the leaf 0 state.
- *Leaf 0:* This is also a testing state. If the testing settles with no conflict, we have found the most recent branch assignment that contributes to the conflict. The FSM will backtrack directly to that variable. If a conflict is detected, it will switch to 1 and continue the testing.
- *bk0a:* This state works in coordination with the leaf 0 state. It is reached through testing backtrack to the passive 1 state. If the test results in no conflict, this variable is the backtrack target.
- *bk0b:* This state works in coordination with the leaf 1 state. If the test results in no conflict, this variable is the backtrack target. If the conflict persists, FSM passes the token to the left and returns to idle.

29.4 A DIFFERENT APPROACH TO REDUCE COMPILATION TIME AND IMPROVE ALGORITHM EFFICIENCY

A practical issue in creating an FPGA-based SAT solver circuit optimized to a specific problem instance is the time needed to generate the circuit. While the VHDL for the solver circuit can be generated in less than a second, the process of FPGA compilation is quite long. It can take at least 10 to 20 minutes to compile the mapping for a single FPGA. FPGA hardware and software have improved to the point that a compilation may take a few minutes; however, compilation time still cannot be ignored. In the next section we describe an SAT solver with reduced compilation time and a further improved algorithm.

29.4.1 System Architecture

The solution described in the previous section directly maps the SAT formula into an SAT solver circuit. It does, however, have limitations:

- The circuit design does not take into account any physical design issues. The implication circuit includes connections between state machines that may be placed far away from each other. There are also wide OR gates that generate global control signals. The solver requires massive routing resources, and the system clock rate is low.
- The circuit is a complex netlist with little locality, and it takes a long time to compile into FPGA configurations.
- The solver implements the basic backtrack search algorithm. Although an improved nonchronological backtracking was implemented, the architecture does not support learning.

To deal with these issues, we developed a follow-on SAT solver with lessons learned from the previous design [13, 14]. The following characteristics of the new design address the previous design's shortcomings:

- Structural regularity is a high priority. A regular structure allows easier physical design. Specially designed processing elements allow regular placement and distributed processing. Overall, modular approaches can improve clock speed and allow fast circuit generation.
- Shared-wire global signaling is used to distribute data across the system. For example, a pipelined ring-style bus replaces the random interconnects. The bus allows a faster clock rate, a low pin count between chips, and a regular structure.
- The algorithm control is separated from the parallel data processing in the architecture. This allows the development of sophisticated control algorithms.
- Algorithm improvements have been implemented. In addition to implication, the circuit is capable of conflict analysis. Therefore, nonchronological backtracking and learning can be implemented.

The core of the new design is an optimized pipelined bus system, in which the bus width can be customized according to the hardware resources. The bus includes both control and data bits. The control bits notify the processing elements of actions to take; the data bits utilize a fixed sequence to encode the variable values. The system uses the same 2-bit encoding for variable values. Thus, a width of 32 data bits supports 16 variables. Also, the variables are encoded with a fixed order. For example, if at clock t the variables are v_1 through v_{16}, then, at $t+1$, the variables are v_{17} through v_{32}. In n clock cycles, $w*n$ variables pass through a stage, where $2w$ is the bit width of the data bus. The bus only propagates the variable value. There is no need to propagate the variable identification because it is inferred from the sequence. At each stage, the data bit may be OR'ed with a local signal, allowing it to be set to 1.

Figure 29.8 shows the global topology. The bus width is 40 bits, with 32 bits for data and 8 bits for control. Figure 29.9 shows one stage of the bus. The value is accessible to the PE as Vi_in. The propagated value can be set or reset through the signals Vi_set and Vi_reset_n. The main control block is the core of the algorithm control. It maintains an internal copy of the variable states and controls the backtrack algorithm.

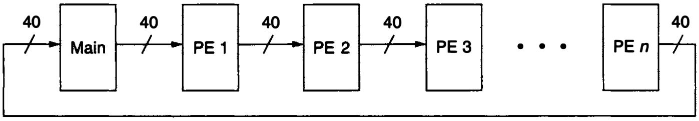


FIGURE 29.8 ■ The global topology for processing and communication in the new SAT architecture, with improved conflict analysis and nonchronological backtracking.

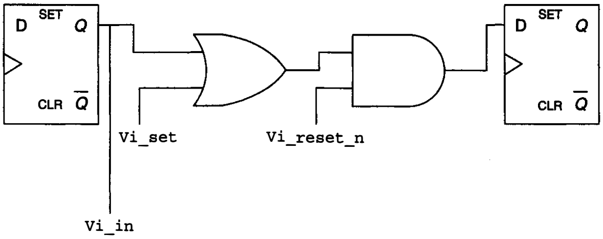


FIGURE 29.9 ■ One stage of the pipelined bus.

Multiclause modules can be placed in one processing element (PE). The total number of PEs depends on the total number of clauses in the CNF and the number of clauses per PE. Each PE contains a resettable counter to count the sequence of variables. The clause modules use the counter to identify variables on the bus.

A clause module holds the data corresponding to one clause. To simplify the hardware design, a 3-SAT formula is assumed (i.e., each clause has at most three literals). This assumption does not lose generality, because any SAT formula can be transformed into a 3-SAT formula in polynomial time by introducing new variables and breaking up long clauses. Each clause module has the following functions:

- *Implication.* Each clause should check for implication and put implied values onto the bus.
- *Conflict analysis.* This is the reversal of the implication process. Given an implied variable, the module finds the variables that lead to the implication.
- *Storage and interface.* The module interfaces with the bus, taking commands and variable values from it. It also sends new values and flags for value updates to the bus. It needs to store the values of variables related to the clause as well as the implication information.

Clause modules have three basic states: reset, implication, and analysis. The reset state will reset variables to (0, 0) if the corresponding value on the bus is (0, 0) and the state bus dictates reset. It is used during backtrack to undo the decisions and implications made after the backtrack point. Implication uses the same algorithm defined in the previous section. However, because the variable value is propagated on the bus, the clause module should also hold variable values locally. The data latching takes place when the PE counter matches the count stored in the module. The implied value is also stored locally until the correct bit passes through. The module will update the bus value at that moment. An internal flag denotes the implied value. It will be used in the analysis phase.

The analysis phase is the reverse of implication. The goal is to find the list of branch decisions that are transitive predecessors. This can be easily obtained if the history is stored. When the clause module is in analysis mode, it will be idle if it has not generated an implication. If it has generated an implication, it will check if the implied literal is asserted on the bus during analysis. If so, the module will reset this literal on the bus and set the complement of other literals in the clause. In this way it signals to the units that generated the values of these other literals. For example, in the clause $(v_i + v_j + \neg v_k)$, if v_i is implied, the implying predecessors are $v_j = 0$ and $v_k = 1$. These variables may in turn be implied by other variables.

The main control unit handles flow control and decision making. It has the following major states and functions:

- *Branch.* Branch chooses the next free variable and assigns a value to it. Using a fixed variable order and always choosing 1 simplifies the function. A priority encoder can quickly select the first row with a free variable and assign it to 1. The branch state is associated with the first round of broadcasting the variable values. The next state is implication.
- *Implication.* The controller checks for conflicts, in which case it performs conflict analysis. Alternatively, if in two cycles of data movement no new values have been found, all iterative implications have settled. It then performs the next round of branching.
- *Conflict analysis.* This step identifies the variable assignments leading to the conflict. The control bus shows the analysis state. The conflict variable is set to (1, 1), while all other variables are set to (0, 0). When a clause that implied a variable currently asserted on the bus is found, that implied literal is reset to 0 and the implying literals are all set to 1.

When a conflict arises from a branch, a list of variable assignments contributing to it can be collected through conflict analysis. The current branching variable is considered to be implied by this set of literals. The implication is stored in the main control unit and can be expressed as a redundant clause. For example, if assignments $v_i = 1, v_j = 1, v_k = 1, v_l = 1$ lead to conflict, the new clause is $(\neg v_i + \neg v_j + \neg v_k + \neg v_l)$. If v_l is the current branch variable, it is implied to be 0 by this new clause. Conceptually, the new value is not a branch decision. Rather, it is forced to be the opposite value because of the recent conflict. It is a redundant implication not explicitly visible from the original formula. Adding the new clause to the database is a learning process that has been used in modern SAT solvers to prune future search space. Such learning can be carried out in hardware by reserving some FPGAs for this purpose and generating new compilations during runtime.

29.4.2 Performance

The performance of the new design is shown in Table 29.2. It should be noted that the table lists the cycle counts, but the clock rates of the two designs are different. The new design has a regular structure, and communication is pipelined. It is therefore easy to achieve a much higher clock rate. Based on the same Xilinx XC4000 FPGAs, the earlier design, implemented on the IKOS

TABLE 29.2 ■ Performance comparison

Problem	Acceleration of new design without added clauses	Acceleration of new design with added clauses
aim-50_2_0-yes1-2	33.00	65.87
aim-200-6_0-yes1-1	1.32	3.66
aim-50-1_6-no-1	8.10	487.19
aim-50-2_0-no-1	4.95	2449.26
aim-50-2_0-no-4	13.89	1121.68
aim-100-1_6-yes1-1	20.57	4354.04
aim-100-3_4-yes1-4	2.81	10.58
hole7	4.63	4.63
hole8	3.95	3.95
hole9	3.46	3.46
par8-1-c	5.03	5.03
par16-1-c	1.29	1.29
pret60_40	4.05	2154.23
ssa0432-003	0.65	2.04

Note: The comparison is based on normalized speedup against the old design, assuming 20 x clock speed improvement in the new design.

Logic Emulator, achieved a 1- to 2-MHz clock rate. The new design could easily achieve a 20-MHz clock rate in 1998. In 2006, the achievable clock rate was in the range of 200 MHz. This shows that the new design will likely achieve better performance even without added clauses. Still, added clauses can bring dramatic improvement in many problems.

29.4.3 Implementation Issues

One of the objectives of the new design is to reduce compilation time by exploiting its regular structure. However, typical FPGA tools use simulated annealing or similar algorithms to place the components. They are not capable of utilizing the regular structure automatically, and so a regular structure will not yield faster compilation times. It is necessary to bypass the automated tool and directly generate the system layout.

JBits is a tool set that allows direct programming of Xilinx FPGAs. It is an application programming interface (API) to the Xilinx configuration bitstream file that permits Java applications to dynamically modify Xilinx XC4000EX/XL bitstream configurations quickly.

A two-step approach can take advantage of the JBits tool and effectively reduce compilation time. The first step is to create a generic SAT solver template mapped to the FPGAs. The second step is customization to modify the configuration according to a specific problem instance. For each instance, only the second step is needed to compile the SAT solver. It can be performed quickly if the number of changes is small.

The architecture described in the previous section is used with additional constraints to minimize the customization. At each pipelined stage of the bus, multiple clause modules are connected to the bus. By limiting the problem formulation to 3-SAT, all clause modules are the same. The only difference is the variable identification of these three variables and the bus connection. The variable identification is expressed as a constant that can be programmed as a ROM that feeds a comparator. The connection to the bus also depends on the variable identity and polarity.

The points where a clause module wire interconnects with the bus wire should be programmed in the second step. Another simple constraint, that each bus wire connect to no more than one clause module, can be met with a simple greedy assignment algorithm.

The complete methodology to create an SAT solver is as follows:

1. *Design of a single clause module.* An SAT clause module is designed in VHDL. The synthesized netlist is further optimized manually. The design is expressed by schematic capture, which provides a more direct correspondence between design and implementation.
2. *Placement and routing of the module in a bounding box.* Placement constraints/floorplanning sets the bounding box of the clause module. The Xilinx tool automatically places and routes within the bounding box.
3. *Manual improvement.* The Xilinx EPIC tool provides a graphical user interface to manually edit the placement and routing on the FPGA.
4. *Solver generation.* With the bounding box constraints, a sample SAT solver is generated. Additional manual editing creates a regular layout.
5. *Template extraction.* The JBits tool reads the configuration bitstream and identifies the modification points.
6. *Java generator.* The SAT solver generator is created in Java with the JBits library and templates.
7. *Instance-specific bitstream.* The SAT solver generator is run with the problem instance, and the bitstream files are created.
8. *Load/run.* The programming is loaded to the FPGAs and the solver is run.

Only steps 7 and 8 are needed for each problem instance. For this reason, the compilation time is reduced from hours to merely seconds compared to the logic emulator implementation.

The target implementation is the Xilinx XC4036EX FPGA. Each FPGA contains 36 × 36 CLBs, and each clause module takes 4 × 16 CLBs. Sixteen clauses are placed in each FPGA. Each FPGA forms a stage of the pipeline, and multiple FPGAs can form a ring. The Sun Java 1.1.7 tool is used to compile and run the Java program. The host computer is an Intel Pentium Pro running Microsoft NT 4.0. The CPU clock rate is 200 MHz, and the main memory is 128 MB.

Table 29.3 shows the performance comparison, with times given in seconds. The Old Hardware and New Hardware columns include the time to create the FPGA mapping (CAD) and the time to find the solution on the hardware engine (HW). Numbers in parentheses are speedups as compared to the GRASP software.

Table 29.3 ■ Performance comparison between the standard GRASP software and two versions of the hardware SAT solver

	GRASP	**Old hardware**			**New hardware**		
Problem	**SW**	**CAD**	**HW**	**Total**	**CAD**	**HW**	**Total**
a50-2_0-y1-2	0.05	10783	0.0011 (45x)	10783	1.9	0.0004 (125x)	19 (<1x)
a100-2_0-y1-4	894	89530	42 (21x)	89572	2.4	9.7 (92x)	12.1 (74x)
a200-6_0-y1-1	128	>100K	1.35 (94x)	>100K	7.9	0.89 (144x)	8.8 (14x)
dubois20	986	11377	70.8 (14x)	11447	2.3	8.44 (117x)	10.7 (92x)
par8-1-c	0.02	12834	0.000011 (1818x)	12834	2.7	0.000035 (571x)	2.7 (<1x)
par16-1-c	202	83191	1.3 (155x)	83192	9.4	2.2 (92x)	11.6 (17x)
pret60_40	705	12396	18 (39x)	12414	2.3	9 (78x)	11.3 (62x)
Geometric Mean			75.6x	<1x		(134x)	(4.14x) (27.6x speedup problems only)

29.5 DISCUSSION

Many groups have demonstrated that reconfigurable computing, compared to software, can achieve speedups of about 100 times in solving SAT problems. The main reasons are massive parallelism and fine-grained operation due to customized hardware. Software/hardware solutions have been explored to reduce hardware complexity and allow larger problems to be solved. A recent survey of these systems is presented by Skliarova and Ferrari [15].

In each of the software/hardware systems, the massive computation to find unit resolutions/implications and conflicts is the target of hardware acceleration. However, there are several differences among these SAT solvers:

- *Algorithms.* The base algorithms are different. Several of them are based on backtracking similar to that of GRASP. Some use a full variable assignment and employ flipping during the search. Some use matrix representations.
- *Logic engine implementation.* Different styles are used to implement the massively parallel engine. Some use circuit translation, where the SAT formula is translated into logical circuits. This means that the FPGA configuration must be compiled for each problem instance, which is slow. Alternatively, the formula is translated into memory, often distributed into small blocks, which can avoid the compilation time.
- *HW/SW organization.* Some implementations are all hardware, where the entire solver is mapped onto one or multiple FPGAs. Some implementations are SW/HW, in which part of the problem is handled by software.

While there has been significant progress in reconfigurable SAT solvers, we do not see them replacing software solvers in real applications for several reasons:

- *The need for flexibility.* The SAT problem is NP-complete—that is, the worst case is assumed to be exponential to the problem size. However, sophisticated heuristics make many large problems solvable in practice. Modern software SAT solvers typically contain many heuristics and allow the user to choose different heuristic combinations to tackle especially hard problems. Reconfigurable solvers generally have only a few heuristics, and there is little flexibility on which ones to use.
- *Algorithm efficiency.* Most reconfigurable SAT solvers have algorithm efficiencies similar to that of the basic backtrack algorithm with some simple heuristics. In the meantime, software algorithms have made significant efficiency gains. More elaborate analysis, such as conflict analysis, leads to more efficient backtracking and learning. Learning can improve SAT solver speed by several orders of magnitude. Reconfigurable SAT solvers generally lag in algorithm sophistication.
- *The scalability of hardware.* The implementations of reconfigurable SAT solvers are generally limited to moderate-size problems. However, large problems are more likely to benefit from hardware acceleration.

Many projects have designed Boolean satisfiability solvers with reconfigurable computing. These projects demonstrate the performance potential of these solvers through fine-grained custom hardware and massively parallel processing. Significant progress has been made in software algorithms as well, and recently, reconfigurable computing solutions have not kept up in incorporating these innovations. This is partly because the tools for reconfigurable computing are not yet mature.

Future research may result in a breakthrough by studying these issues:

- *Hardware/software solution.* The complex algorithms are difficult to implement and verify in hardware. It is more efficient to partition the problem and allocate only the massively parallel portion to the reconfigurable hardware. With microprocessors embedded in FPGAs, such as Xilinx Virtex-II Pro and Virtex-4, communication between the processor and the FPGA is greatly improved. The proliferation of multicore processors and high-bandwidth interconnects enables the exploitation of parallelism at different levels with heterogeneous processing technologies.
- *System-level design and synthesis methodologies.* Models of computation that preserve concurrency can be mapped to heterogeneous multicore architectures. The designer can decide the trade-off between parallelism and hardware usage. FPGA-based fabrics provide the massive parallelism and low-level customization, while other components, such as embedded processor or controller, can be chosen for their desirable characteristics.
- *Distribution of data and customization of hardware.* Mapping SAT formulas to FPGA circuits generates random routing and requires long compilation times. Mapping problem instances into distributed memory

blocks can solve the time issue but it forces some degree of sequential access. Learning from the design of content addressable memory may lead to hardware architectures better able to solve SAT and other Boolean problems.

- Simultaneous exploration of multiple states. Creating an algorithm that can efficiently explore multiple states in the assignment space simultaneously will allow the utilization of large amounts of computing resources. A simplified approach is to simultaneously run the search on multiple machines with different heuristics. However, efficient utilization of learning across different searches remains an open problem.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[2] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design* 11, January 1992.

[3] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic model checking without BDDs. *Proceedings of the Workshop on Tools and Algorithms for Analysis and Construction of Systems (TACAS)* 1579, LNCS, 1999.

[4] A. Gupta, M. Ganai, C. Wang, Z. Yang, P. Ashar. Learning from BDDs in SAT-based bounded model checking. *Proceedings of the Design Automation Conference*, 2003.

[5] M. Davis, H. Putnam. A computing procedure for quantification theory. *Journal of the ACM* 7, 1960.

[6] J. P. Marques-Silva, K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5), May 1999.

[7] R. J. Bayardo Jr., R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. *Proceedings of the 14th International Conference on Artificial Intelligence*, 1997.

[8] E. Goldberg, Y. Novikov. BerkMin: A fast and robust SAT-solver. *Design, Automation and Test in Europe*, 2002.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an efficient SAT solver. *Proceedings of the 38th Design Automation Conference*, 2001.

[10] P. Zhong, M. Martonosi, P. Ashar, S. Malik. Using configurable computing to accelerate Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18(6), June 1999.

[11] P. Zhong, P. Ashar, S. Malik, M. Martonosi. Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean satisfiability. *Proceedings of the 35th Design and Automation Conference*, June 1998.

[12] P. Zhong, M. Martonosi, P. Ashar, S. Malik. Accelerating Boolean satisfiability with configurable hardware. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[13] P. Zhong, M. Martonosi, P. Ashar, S. Malik. Solving Boolean satisfiability with dynamic hardware configurations. *Proceedings of the Eighth International Workshop on Field-Programmable Logic and Applications: From FPGAs to Computing Paradigms*, August–September 1998.

[14] P. Zhong, M. Martonosi, P. Ashar. FPGA-based SAT solver architecture with near-zero synthesis and layout overhead. *IEE Proceedings on Computer and Digital Techniques* 147(3), May 2000.

[15] I. Skliarova, A. B. Ferrari. Reconfigurable hardware SAT solvers: A survey of systems. *IEEE Transactions on Computers* 53(11), November 2004.

[16] M. Yokoo, T. Suyama, H. Sawada. Solving satisfiability problems using field-programmable gate arrays: First results. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, 1996.

[17] T. Suyama, M. Yokoo, H. Sawada, A. Nagoya. Solving satisfiability problems using reconfigurable computing. *IEEE Transactions on VLSI Systems* 9(1), 2001.

[18] T. Suyama, M. Yokoo, A. Nagoya. Solving satisfiability problems on FPGAs using experimental unit propagation. *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, 1999.

[19] T. Suyama, M. Yokoo, H. Sawada. Solving satisfiability problems using logic synthesis and reconfigurable hardware. *Proceedings of the 31st Hawaii International Conference on System Sciences* 7, 1998.

[20] J. de Sousa, J. P. Marques-Silva, M. Abramovici. A configware/software approach to SAT solving. *Proceedings of the Ninth IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2001.

[21] I. Skliarova, A. B. Ferrari. A software/reconfigurable hardware SAT solver. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12(4), April 2004.

[22] J. Gu. Local search for satisfiability (SAT) problem. *IEEE Transactions on Systems, Man, and Cybernetics* 23(4), July 1993.

[23] H. Zhang, M. Stickel. An efficient algorithm for unit-propagation. *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, 1996.

[24] H. Zhang. SATO: An efficient propositional prover. *Proceedings of the International Conference on Automated Deduction*, 1997.

[25] L. Zhang, S. Malik. The quest for efficient Boolean satisfiability solvers. *Proceedings of the Eighth International Conference on Computer-Aided Deduction*; *Proceedings of 14th Conference on Computer-Aided Verification*, July 2002.

[26] L. Zhang, S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. *DATE2003*, March 2003.

[27] F. A. Aloul, A. Ramani, I. L. Markov, K. A. Sakallah. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22(9), September 2003.

[28] P. T. Darga, M. H. Liffiton, K. A. Sakallah, I. L. Markov. Exploiting structure in symmetry detection for CNF. *Proceedings of the 41st IEEE/ACM Design Automation Conference*, 2004.

[29] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, I. L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor. *Proceedings of the 41st IEEE/ACM Design Automation Conference, 2004.*

CHAPTER 30

MULTI-FPGA SYSTEMS: LOGIC EMULATION

Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst

Application specific integrated circuit (ASIC) verification has been an important and commercially successful application of field-programmable gate arrays (FPGAs) for over a decade. By mapping the logic of a new chip design onto a system of FPGAs, logic emulation systems provide a high-speed simulation of the design under development. As FPGA technology has matured and FPGA logic ca,pacity has grown, the use of FPGAs for functional logic emulation has increased. Contemporary emulation systems often include a sizable number of FPGA and memory devices organized in topologies that allow for efficient logic evaluation and inter-FPGA communication.

Although the hardware architecture of an emulator plays an important role in defining its effectiveness, system usability is often most closely tied to an emulator's compilation environment. To successfully map a complete ASIC design to an emulation system, emulators require optimized compilation steps that effectively distribute design logic across available FPGA resources and coordinate intra-FPGA computation and inter-FPGA communication.

To illustrate contemporary approaches to FPGA-based logic emulation, we profile here the hardware and software systems of a commercial FPGA-based emulator. We show that, although off-the-shelf FPGAs have been used effectively in a number of commercial logic emulators, several issues related to FPGA compile time, design debugging, and emulator host interfacing must be addressed to maintain their commercial viability.

30.1 BACKGROUND

Research in reconfigurable computing has been active for well over a decade, but the widespread commercial use of FPGAs as computing devices has been limited. A notable commercial success story for reconfigurable computing has been the use of FPGAs in ASIC logic verification. Over the past decade, the number of transistors that can be integrated into application-specific devices has grown exponentially with Moore's Law, leading to an increased need to verify design functionality prior to device fabrication. Currently, it is estimated that 60 to 80 percent of ASIC design time is spent performing verification [29], primarily because of the high nonrecurring engineering (NRE) cost associated with ASIC fabrication. The flexibility, parallelism, and reprogrammability of FPGAs make them an ideal platform for verifying, prior to fabrication, the functionality of ASIC designs. The availability of automatic FPGA mapping tools, such as those described in Chapters 13, 14, and 17, have streamlined the design conversion process, making the path from ASIC design to FPGA implementation more straightforward.

FPGA-based logic verification is often used to augment or replace microprocessor-based simulation of register transfer level (RTL) or gate-level designs. The primary source of emulation speed improvement versus simulation is the parallel implementation of circuit logic in the FPGA. While the amount of logic evaluated per clock cycle in a microprocessor-based simulator is constrained by a limited number of ALUs (typically four or five at most), the number of per-cycle FPGA operations per emulation system is constrained only by the available amount of total FPGA resources. This increase in logic evaluation capacity comes at a cost. Unlike its simulation counterparts, FPGA-based emulation can provide only functional verification for designs. Because the fundamental technology used to implement the emulated logic differs from the source ASIC technology, postlayout timing information cannot be replicated. As a result, FPGA-based emulators support only cycle-accurate logic evaluation that is synchronized to design clock edges of the emulated design. Additionally, circuit debugging for emulation systems is often more complex than debugging with simulators. The sequential nature of simulation-based verification facilitates debugging and logic tracing. Logic analysis in a parallel verification environment requires the use of specialized hardware resources and debugging tools.

FPGA-based emulators take on a variety of forms, ranging from single-device systems to commercial emulation systems that include hundreds of devices. Although specific system implementations vary, most FPGA-based logic emulators contain a tightly connected collection of FPGA devices. These systems can be distinguished by their component FPGA and memory devices, interconnection topology, design-mapping software, and external interfaces. The system topology defines the positions of FPGAs and inter-FPGA communication resources. The need for multiple devices to emulate many ASIC designs is due to the cost of FPGA reconfigurability. Because the silicon area overhead of FPGA versus ASIC technology has been measured to be about 40x [15], FPGA programming technology requires that an ASIC logic design be partitioned across multiple FPGA devices to achieve the necessary device logic capacity.

For most emulators, there is a strong association between the physical architecture of the FPGA system and the compiler used to map user designs to the emulator. Like the intra-FPGA mapping flow outlined in Chapters 13, 14, and 17, emulation mapping for multi-FPGA emulators requires a series of complex and interrelated algorithms. As we will see later in this section, emulation system compilation is complicated by the variety of design features in contemporary ASICs. These features include multiple asynchronous clock domains, multiported memories, and testing and debugging interfaces, which are playing an increasingly important role. In assessing modern emulation, the interfaces between emulators, simulators, logic analyzers, and prototype systems must be considered. It will be shown that, in the future of FPGA-based logic emulation, both design compilation and testing interfaces will play a critical role.

To illustrate the complexity of contemporary FPGA-based emulation, the hardware, compilation, and testing components of a VirtuaLogic VLE-2M emulation system from Mentor Graphics [21] will be profiled. This commercially successful system demonstrates not only the benefits of FPGA-based emulation, but also some of its limitations.

30.2 USES OF LOGIC EMULATION SYSTEMS

Logic emulation systems are typically used in one of two verification scenarios: (1) as a physical replacement for an ASIC in a target system, or (2) as a simulation accelerator. The ASIC replacement approach requires the use of a physical connection between the emulator and the target system. As shown in Figure 30.1, one end of the connection typically plugs into connectors on the emulation system that are interfaced to selected FPGA I/O pins. The other end of the connection plugs into the location on the target system that would normally hold the package of the emulated device. This emulation pod typically has the same pin configuration as the emulated device package. The use of in-circuit emulation allows for complete target system verification, including the emulated design and surrounding interfaces and peripherals. Although many times the target system is forced to operate at clock speeds of 0.5 to 5 MHz, a substantial amount of system functionality can generally be evaluated via in-circuit emulation. An attached logic analyzer is often used to probe specific design signals.

An alternative to in-circuit emulation is coverification (sometimes called cosimulation). In this mode of operation, the logic emulator works in concert with a host workstation to verify an emulated design without the use of

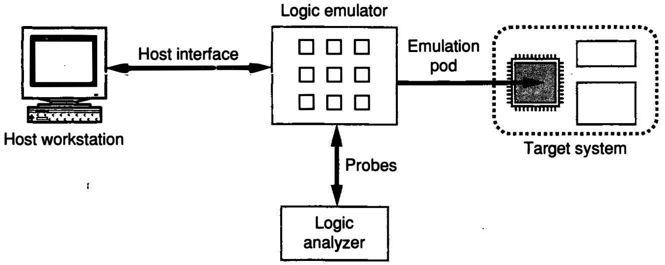


FIGURE 30.1 ■ A typical configuration of a logic emulation system.

a physical target system. Typically, the host workstation (Figure 30.1) performs the simulation of target system components and provides inputs to the emulated design via a host interface such as a backplane bus or cable. Design outputs are returned to the host workstation via the same path. In most cases, only the most time-consuming portion of the design under test is mapped to the emulator. The rest is simulated on the companion processor located in the host workstation. Coverification is often used to concurrently verify software components running on both the processor in the host workstation and in the emulated design.

In contrast to simulation, the use of in-circuit emulation and coverification allows for exhaustive prefabrication functional testing [3]. Typically, logic emulation can provide about five to six orders of magnitude speedup versus simulation for a logic design [2, 14]. Numerous commercial ASIC projects have used coverification to confirm the functionality of end applications with billions of test vectors prior to chip fabrication [3]. The speed of in-circuit emulation often allows for complete software system design verification as soon as a functionally specified ASIC design is complete. In the case of microprocessor design, a significant fraction of the emulated processor's software system can be tested long before processor fabrication, ensuring the functionality of both hardware and software. For example, Unix was successfully booted on an emulated M68060 microprocessor in about two hours [14]. This value represents a 40,000 times speedup over RTL simulation for the same processor operation.

30.3 TYPES OF LOGIC EMULATION SYSTEMS

For many designers of small ASICs, a large, expensive multi-FPGA emulation system may be unnecessary because one large FPGA and some associated external memory may be sufficient to implement the entire ASIC design.

30.3.1 Single-FPGA Emulation

The use of a single FPGA simplifies emulation system mapping because design partitioning and inter-FPGA routing are unneeded. Often, an unmodified RTL description of the ASIC design can be resynthesized for the FPGA with the use of an alternate synthesis library. Standard FPGA compilation tools are then used to complete the design mapping. As shown in Figure 30.2, the FPGA used for prototyping is typically mounted on a custom board that receives design inputs either from a target system where the completed ASIC design eventually will be located or from a workstation that provides input test vectors via a download cable. Additional interfaces are usually provided to allow for connections to a power supply and a logic analyzer. Since most FPGAs used for prototyping are SRAM based, resources must be provided to store and download the configuration bitstream to the FPGA at power-up.

As the logic capacity of FPGAs grows, it may appear that an increasing number of ASIC designs could be prototyped using a single FPGA. However, since both FPGA and ASIC gate counts follow the same VLSI process trends,

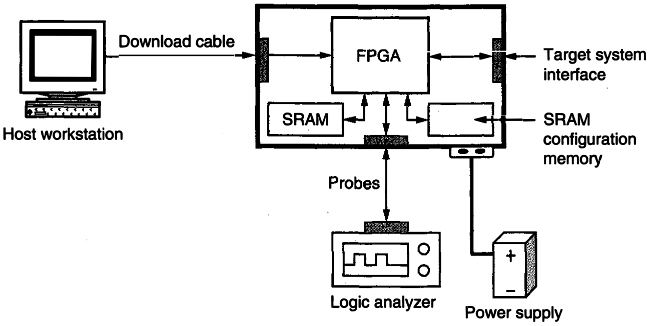


FIGURE 30.2 ■ An example of a single-FPGA logic emulation system.

it is likely that most ASIC designs will continue to require multiple FPGAs for verification.

30.3.2 Multi-FPGA Emulation

Contemporary multi-FPGA emulation systems are complex verification platforms containing hundreds of FPGA and memory chips, high-speed interfaces to target systems, hosts, logic analyzers, and support for interactive debugging [11]. Since their initial commercial introduction in 1988, these systems have evolved into important functional verification platforms [7]. Typical systems include multiple boards each containing tens of FPGA devices interconnected in a fixed topology. Interboard communication is performed via fixed connections or a backplane bus. Because of the need to communicate signals between FPGAs, the typical frequency of an emulated design is in the range of 0.5 to 5 MHz.

Two distinguishing characteristics of a multi-FPGA logic emulator are the topology used to interconnect FPGAs and the approach used to communicate interpartition logic signals between them. Before addressing the issues of topology, two possible approaches for assigning logical signals to inter-FPGA wires will be analyzed.

Consider the mapping of a simple circuit shown in Figure 30.3(a) to two FPGAs as shown in Figure 30.3(b). For this circuit, two interpartition signals (x and y) exist. One approach to mapping these signals to inter-FPGA wires is to dedicate them to inter-FPGA wires A and B, respectively, as shown in Figure 30.3(b). This *dedicated-wire* mapping preserves the original structure of the circuit and does not require the inclusion of any additional logic. In contrast, the mapping shown in Figure 30.3(c), adds pipeline flip-flops and a multiplexer to interpartition signals so that inter-FPGA wire A can be shared. From the figure it can be seen that wire A is multiplexed to transport both x and y. This *multiplexed-wire* approach allows for more efficient use of FPGA pins and inter-FPGA wires, at the cost of additional FPGA logic and flip-flops. However, in most emulation systems I/O pins are a more precious resource than logic and flip-flops.

Both dedicated-wire and multiplexed-wire FPGA-based emulators are commercially available. Dedicated-wire systems include the SystemRealizer [24] and Mercury [25] families from Cadence; multiplexed-wire systems include Cadence Xcite [36] and the Mentor Graphics VirtuaLogic [21] and VStation [22] families. For dedicated-wire systems, design logic partitions must meet both the pin and gate count requirements of the target FPGAs. In virtually all cases, the FPGAs are pin limited, constraining the amount of logic and associated I/O that can be assigned to each FPGA. Rent's Rule [17], an empirical relationship that quantifies the growth of pin requirements as logic capacity increases, indicates that this problem is likely to get worse as FPGA logic capacity increases. As a result,

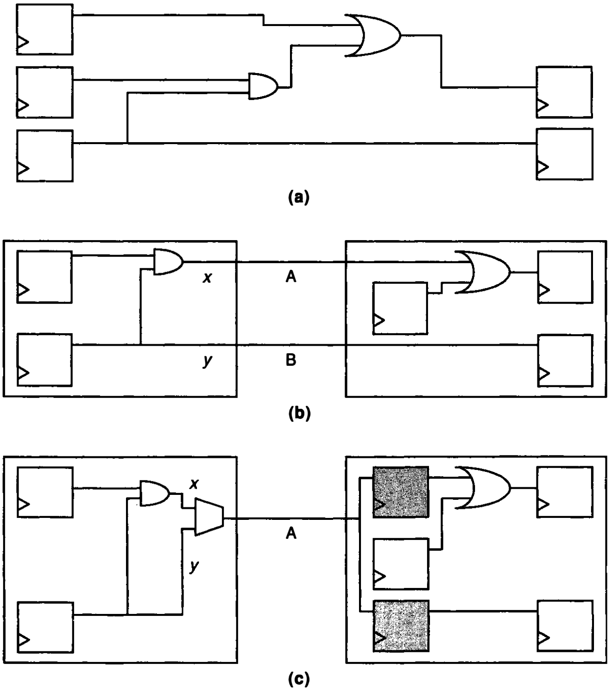


FIGURE 30.3 ■ The mapping of a simple circuit (a) by dedicated-wire(b) and multiplexed-wire (c) assignment.

the time-multiplexed use of pin resources is prevalent in contemporary emulation systems.

A series of topologies for FPGA interconnection have been investigated for both dedicated-wire and multiplexed-wire emulators. A number of early commercial dedicated-wire emulation systems organized FPGAs primarily in a near-neighbor or low-dimensional mesh topology, as illustrated in Figure 30.4(a). Although these topologies are easy to build, their lack of routing flexibility complicates design partitioning. Since many interpartition connections may not have direct FPGA-to-FPGA connections, one or more FPGAs are required to provide through-hop connectivity. Not only does this make the timing along interpartition connections unpredictable, but scarce FPGA pin resources must be dedicated to through-hop connections. As a result, direct-connect dedicated-wire systems are now used only for emulation systems with a very small number of FPGAs (typically four or less) [4]. These systems often allow direct connections between all FPGAs, eliminating the need for through-hops.

In an attempt to provide predictable FPGA delay and eliminate the need for through-hops, a series of emulation systems were developed that use specialized crossbar devices called field-programmable interconnect chips (FPICs) in addition to FPGAs [7]. These systems route most or all inter-FPGA connections through the FPICs so that the length of each inter-FPGA path is predictable. For basic systems, such as the one shown in Figure 30.4(b), some of each FPGA's I/O pins are dedicated to bidirectional connections on each FPIC device forming a crossbar. As a result, any inter-FPGA connection can be made by passing through a single FPIC, leading to predictable timing. Multiple levels of FPIC interconnect allow for system scaling to hundreds of FPGAs. The delay for each individual path is predictable because the FPIC's timing is predictable, although the number of FPICs traversed by different inter-FPGA paths may vary.

Most multiplexed-wire systems use meshes with primarily near-neighbor connectivity [7, 34]. Inter-FPGA paths are pipelined, so each path has a predictable

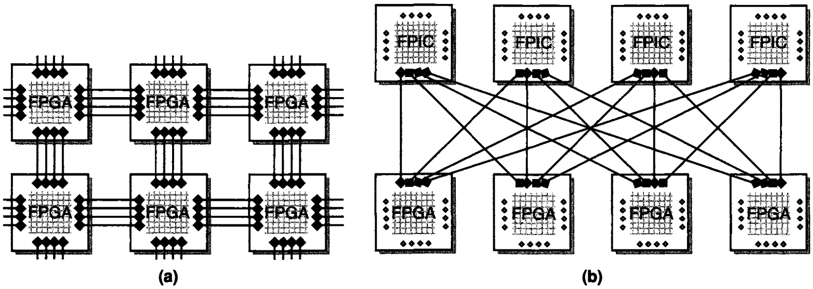


FIGURE 30.4 ■ Example FPGA-based logic emulator topologies: (a) mesh; (b) crossbar. *Source:* Adapted from Hauck [7].

delay, which is a multiple of the system clock frequency. Additionally, inter-FPGA routing congestion is overcome by the reuse of inter-FPGA routing resources, eliminating the restrictions created by through-hops. Although some multiplexed-wire systems that use partial or full crossbars (FPICs) have been proposed [19], the need for these expensive devices in time-multiplexed systems is unclear.

30.3.3 Design-mapping Overview

Several key issues drive the use of logic emulation systems. For most emulation products, system ease of use and resource utilization are important factors in system design. The translation of designs from ASIC netlist to multi-FPGA implementation must be fully or nearly automatic. These ease-of-use issues require sophisticated multi-FPGA computer-aided design approaches to process netlists in addition to the per-FPGA processing for numerous individual FPGAs.

A high-level flow for multi-FPGA logic emulation similar to the flow outlined by Hauck and Agarwal [8] is shown in Figure 30.5. It starts with a circuit description that is specified at the behavioral or register transfer level. Design translation, which typically includes logic synthesis, converts the high-level netlist to a gate-level structural equivalent. Following design translation, design logic is partitioned into pieces that will fit within the logic resources of individual FPGA devices. Partitioning is often performed to minimize required inter-FPGA interconnect, control system-wide critical path delay, and localize memory access. For some systems, partitioning must be performed so that inter-FPGA routing restrictions in terms of available FPGA pin count and system topology are considered. If the logic emulator contains memory chips that are external to the FPGA, design memory must be partitioned across memory resources to meet memory chip capacity constraints.

Partitioned design logic and memory structures are subsequently assigned to specific system devices via global placement. For some systems, swap-based placement algorithms, which are similar to the FPGA placement approaches described in Chapter 14, are used. A placement cost metric based on distance and delay is often iteratively used to judge placement quality. Partitioning and placement are sometimes combined into a single step to concurrently optimize interpartition bandwidth and inter-FPGA signal delay and distance [8]. The communication of interpartition signals between FPGAs is determined based on routing algorithms. For most multi-FPGA emulators, routing involves the determination of the shortest feasible path between FPGAs using available board interconnect resources for each inter-FPGA signal [2]. Topology constraints often require these signals to pass through intermediate (through-hop) FPGAs.

The last mapping step in logic emulation involves the individual compilation of the FPGAs. Multi-FPGA emulation systems have a number of constraints that can lead to less-than-efficient FPGA use. The FPGA compilation step may require hundreds of individual compiles. If even one design partition fails to successfully map to its target FPGA, the emulation flow shown in Figure 30.5 must be restarted from the design partitioning step. As a result, design partitions are often sized conservatively to ensure successful compilation.

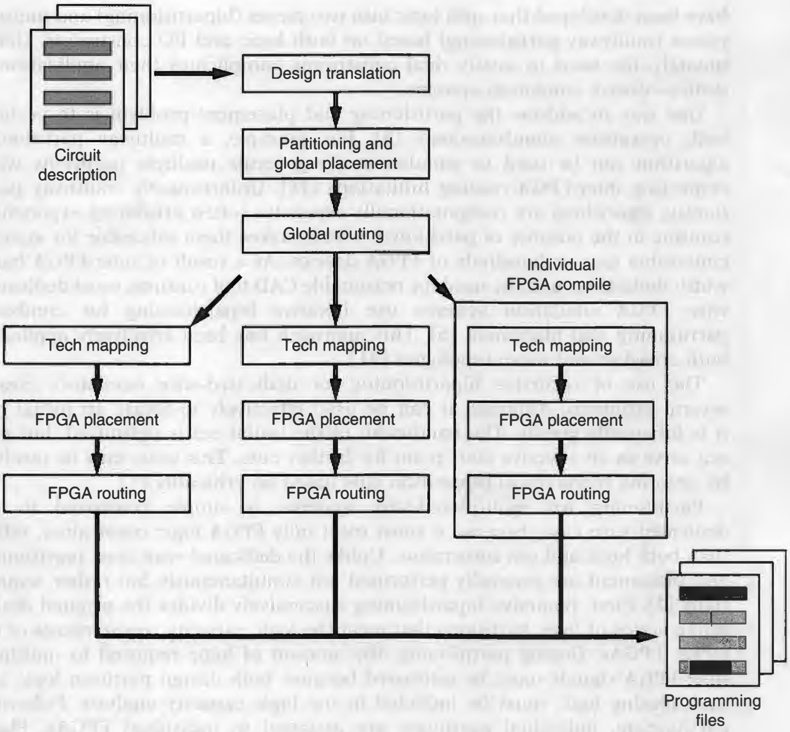


FIGURE 30.5 ■ A typical multi-FPGA emulator mapping flow. *Source:* Adapted from Hauck and Agarwal [8].

Although the steps just described define the high-level mapping flow for FPGA-based logic emulators, the specific partitioning, placement, and routing approaches used by individual emulators are heavily influenced by the approach used to communicate intermediate data signals between FPGAs. Although similar, dedicated-wire and multiplexed-wire emulators require specialized partitioning, placement, and routing algorithms.

30.3.4 Multi-FPGA Partitioning and Placement Approaches

Design partitioning and placement play an important role in system performance for dedicated-wire FPGA-based logic emulators. Because FPGA pins are such critical resources for these systems, the primary goal of partitioning is to minimize communication between partitions. A large number of algorithms have been developed that split logic into two pieces (bipartitioning) and multiple pieces (multiway partitioning) based on both logic and I/O constraints. Unfortunately, the need to satisfy dual constraints complicates their application to dedicated-wire emulation systems.

One way to address the partitioning and placement problem is to perform both operations simultaneously [8]. For example, a multiway partitioning algorithm can be used to simultaneously generate multiple partitions while respecting inter-FPGA routing limitations [28]. Unfortunately, multiway partitioning algorithms are computationally expensive (often exhibiting exponential runtime in the number of partitions), which makes them infeasible for systems containing tens or hundreds of FPGA devices. As a result of inter-FPGA bandwidth limitations and the need for reasonable CAD tool runtime, most dedicated-wire FPGA emulation systems use iterative bipartitioning for combined partitioning and placement [6]. This approach has been effectively applied to both crossbar and mesh topologies [31].

The use of recursive bipartitioning for dedicated-wire emulators creates several problems. Although it can be used effectively to locate an initial cut, it is inherently greedy. The bandwidth of the initial cut is optimized, but may not serve as an effective start point for further cuts. This issue may be resolved by ordering hierarchical bipartition cuts based on criticality [5].

Partitioning for multiplexed-wire systems is simple compared to the dedicated-wire case, because it must meet only FPGA logic constraints, rather than both logic and pin constraints. Unlike the dedicated-wire case, partitioning and placement are generally performed not simultaneously but rather sequentially [2]. First, recursive bipartitioning successively divides the original design into a series of logic partitions that meet the logic capacity requirements of the target FPGAs. During partitioning, the amount of logic required to multiplex inter-FPGA signals must be estimated because both design partition logic and multiplexing logic must be included in the logic capacity analysis. Following partitioning, individual partitions are assigned to individual FPGAs. Placement typically attempts to minimize system-wide communication by minimizing inter-FPGA distance, particularly on critical paths. To fully explore placement choices, simulated annealing is frequently used for multi-FPGA placement [2].

30.3.5 Multi-FPGA Routing Approaches

The global routing step determines which FPGAs are used to route inter-FPGA signals. Inter-FPGA routes may directly connect source and destination FPGAs, or intermediate through-hops may be necessary. Global routing algorithms typically attempt to minimize distance and inter-FPGA routing resource usage while ensuring that no routing resources are overused.

The routing problem for dedicated-wire systems is similar to the intra-FPGA routing problem described in Chapter 17. In dedicated-wire systems, the amount of available inter-FPGA wiring is fixed, possibly leading to infeasible or inefficient routes if an effective routing algorithm is not employed. Groups of wires between FPGAs are considered a communication channel, and inter-FPGA routing channels can be represented as a directed channel graph. As seen in Figure 30.6, for a direct-connect topology, the edge weight in the channel graph represents the number of physical wires in the channel [8]. Prior to routing, the channel graph for the system topology in Figure 30.6(a) can be represented as in Figure 30.6(b).

As routing is performed, inter-FPGA connections are assigned to wires, reducing the available capacity in each channel. A variant of maze routing [18] is typically used to assign inter-FPGA signals to specific system wires. Like the maze-routing algorithms used for intra-FPGA connections, multiple router iterations are often necessary. The maze-routing algorithm works by selecting a wire and finding the shortest feasible path from its source to its destination partition. Multiple iterations involving rip-up may be necessary to complete all routes.

The example mapping in Figure 30.7 provides an overview of the use of channel graph representation. Following the assignment of logical signals from the mapped design in Figure 30.7(a) to inter-FPGA wires, the channel availability is modified to take used wires into account. The effects of this assignment are shown in Figure 30.7(b), where the modified channels are shown with dashed lines.

For multiplexed-wire systems, both intra-FPGA computation and inter-FPGA communication are synchronized by a global system clock. This clock provides control over the sequence of events in the time-multiplexed system. Because many combinational evaluations and signal transfers occur in a single design (emulation) clock cycle, the system clock must operate at a faster speed than that of the design clock of the emulated design. Thus, routing in multiplexed-wire

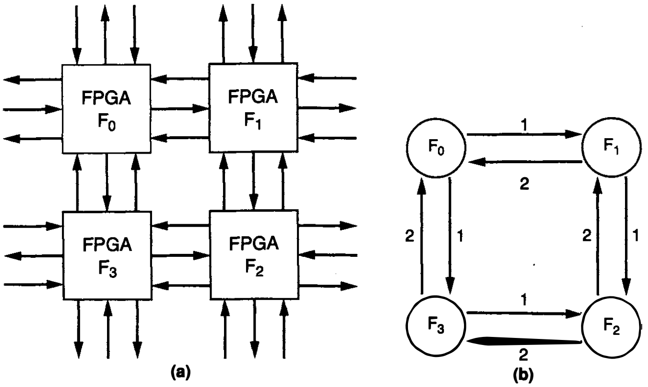


FIGURE 30.6 ■ (a) A multi-FPGA interconnection and (b) the associated channel graph for dedicated-wire routing. *Source:* Adapted from Hauck and Agarwal [8].

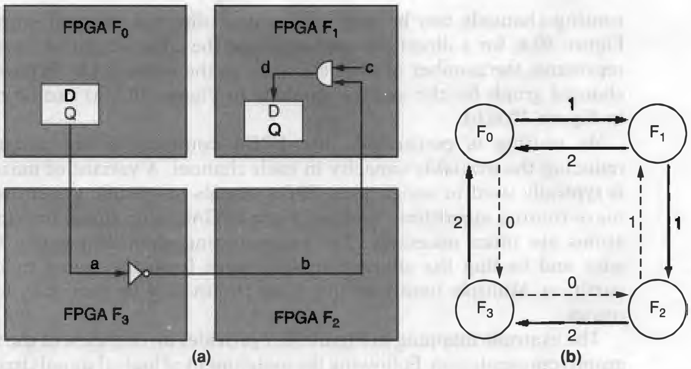


FIGURE 30.7 ■ Assignment of logic signals to inter-FPGA wires in a dedicated-wire system (a), and the resultant mapping (b).

systems assigns each interpartition wire a source–destination path schedule in both time and space.

Routing for multiplexed-wire systems generally requires two routing steps to connect an inter-FPGA signal: the determination of a feasible path between FPGAs and the scheduling of multiplexed signal transport along the path [2]. Initially, a path between source and destination FPGAs is determined using a shortest-path algorithm. Unlike dedicated-wire routing, the utilization of wires in the channel is less restrictive because a different signal may be assigned to each wire on each clock cycle. Following path selection, a data signal can be transmitted along an inter-FPGA path as soon as it is assigned a valid logic value by the flip-flop or logic gate that drives it. To complete the transmission, the signal is assigned to a series of inter-FPGA wires along the path until it reaches the destination FPGA. One clock cycle of the system clock is allowed for each inter-FPGA hop along the path. Because inter-FPGA paths are synchronized at FPGA boundaries with pipeline flip-flops, long combinational paths are effectively broken into a series of discrete timesteps. A number of scheduling algorithms that perform the assignment of interpartition signals to inter-FPGA wires have been developed [2, 32].

The result of routing using multiplexed wires is illustrated in the following example taken from Tessier and Jana [34]. In Figure 30.8, the circuit shown in Figure 30.7(a) has once again been partitioned onto FPGAs interconnected using the direct-connect FPGA topology shown in Figure 30.6(a). Each inter-FPGA signal can travel only between two FPGAs during each system clock cycle. In the figure, pipeline flip-flops, which have been added to allow multiplexed communication on each path, are shaded. Circuit communication and computation in terms of system clock cycles can be determined by evaluating the critical path from signal *a* to signal *d*, as shown in Figure 30.9. In both Figures 30.8 and 30.9, system clock cycles are labeled V_1 through V_5. In Figure 30.8, communication delays

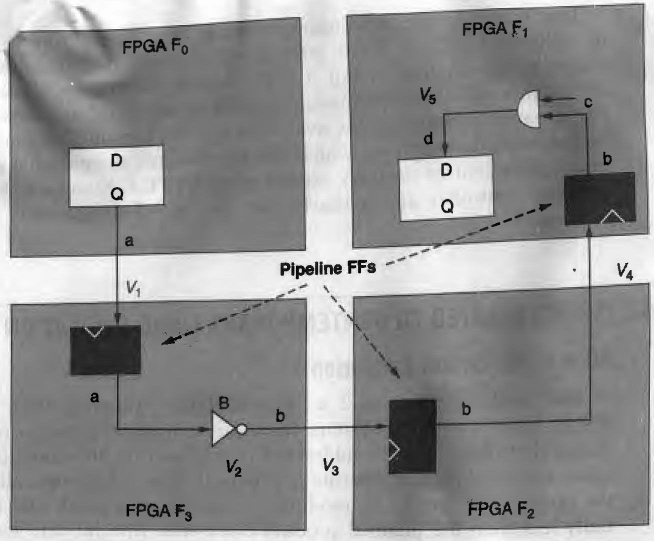


FIGURE 30.8 ■ Circuit mapping to FPGAs for a multiplexed-wire system.

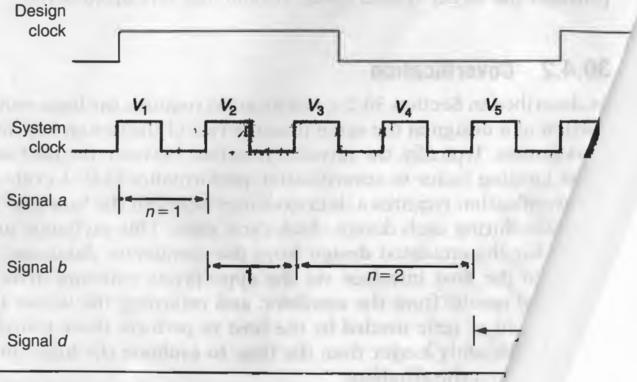


FIGURE 30.9 ■ The design clock cycle for the circuit mapping shown ir
labeled *n* indicate a communication delay of *n* system clock cycles.

are listed, with *n* equal to the number of system clock
munication. Combinational evaluations are listed, wit
system cycle V_5, signal *d* is latched into a design flip-
clock cycle.

The schedule for this example does not depend on
of individual signals. Each interpartition signal is transmi binary
design cycle, whether or not it has changed. Alternative, dyna value
approaches, which only transmit changed signals, have also been p ing each
For dynamic scheduling, the availability of the communication reso uling
be determined at *runtime*, which can significantly increase the amoun ing
munication control circuitry needed in each FPGA. Kwon and Kyung [1
a global controller and a shared bus to control dynamically scheduled
movement.

30.4 ISSUES RELATED TO CONTEMPORARY LOGIC EMULATION

30.4.1 In-circuit Emulation

As discussed in Section 30.2, a logic emulation system is often used to replace design logic in a target system. In-circuit emulation presents a series of chal-
nges that often must be addressed by the user of the emulation system [11].
e emulated designs operate at relatively slow clock rates, all or a portion of
get system must be modified to operate at a clock rate that is substan-
s than the planned product clock rate. Special care must be taken to
actions such as DRAM refresh and device phase-locked loop activity
sely affected. The clock for the target system must be interfaced to
control emulator logic evaluation. In some cases, the emulator
t system clock, simplifying synchronization.

coverification requires the logic emulator to verify a
e time the rest of the design is simulated on a host
cal interface between the host and the emulator
ion performance [12]. A cycle-based approach
ange between the host and the FPGA-based
le edge. This exchange includes collating
simulation database, transferring the
priate software driver, collecting the
urning the values to the simulator.
form these transfer operations is
te the logic for a single design

Figure 30.8. Spans
cycles required for com-
h a number (e.g., 1). After
flop, completing the design

en introduced as a way
interfacing, the host-based
independently for a number
of data that must be trans-
Transaction-based interfacing often

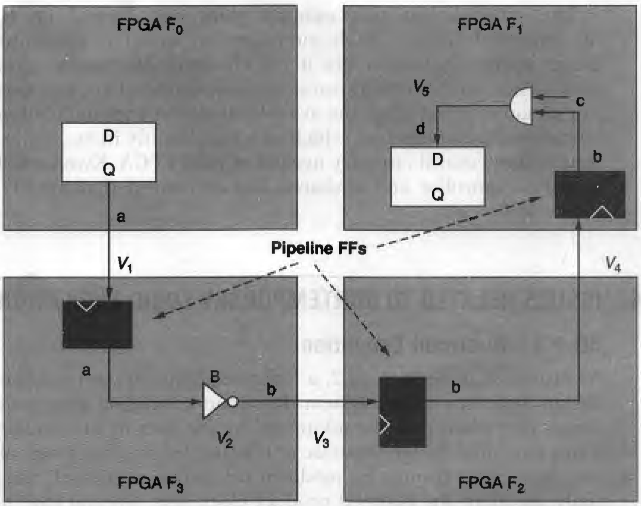


FIGURE 30.8 ■ Circuit mapping to FPGAs for a multiplexed-wire system.

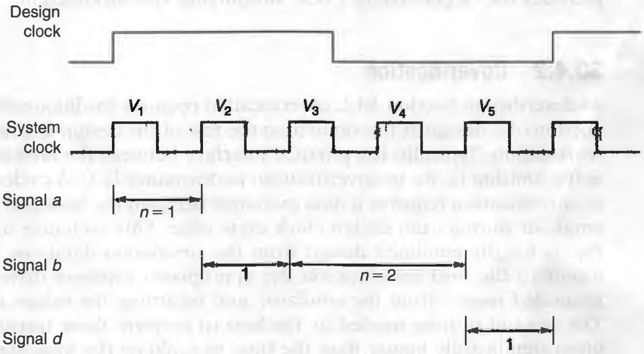


FIGURE 30.9 ■ The design clock cycle for the circuit mapping shown in Figure 30.8. Spans labeled *n* indicate a communication delay of *n* system clock cycles.

are listed, with *n* equal to the number of system clock cycles required for communication. Combinational evaluations are listed, with a number (e.g., 1). After system cycle V_5, signal *d* is latched into a design flip-flop, completing the design clock cycle.

The schedule for this example does not depend on the binary value of individual signals. Each interpartition signal is transmitted during each design cycle, whether or not it has changed. Alternative, dynamic scheduling approaches, which only transmit changed signals, have also been proposed [16]. For dynamic scheduling, the availability of the communication resources must be determined at *runtime*, which can significantly increase the amount of communication control circuitry needed in each FPGA. Kwon and Kyung [16] used a global controller and a shared bus to control dynamically scheduled data movement.

30.4 ISSUES RELATED TO CONTEMPORARY LOGIC EMULATION

30.4.1 In-circuit Emulation

As discussed in Section 30.2, a logic emulation system is often used to replace design logic in a target system. In-circuit emulation presents a series of challenges that often must be addressed by the user of the emulation system [11]. Since emulated designs operate at relatively slow clock rates, all or a portion of the target system must be modified to operate at a clock rate that is substantially less than the planned product clock rate. Special care must be taken to ensure that actions such as DRAM refresh and device phase-locked loop activity are not adversely affected. The clock for the target system must be interfaced to the emulator to control emulator logic evaluation. In some cases, the emulator provides the target system clock, simplifying synchronization.

30.4.2 Coverification

As described in Section 30.2, coverification requires the logic emulator to verify a portion of a design at the same time the rest of the design is simulated on a host workstation. Typically, the physical interface between the host and the emulator is the limiting factor to coverification performance [12]. A cycle-based approach to coverification requires a data exchange between the host and the FPGA-based emulator during each design clock cycle edge. This exchange includes collating inputs for the emulated design from the simulation database, transferring the inputs to the host interface via the appropriate software driver, collecting the generated results from the emulator, and returning the values to the simulator. The amount of time needed by the host to perform these transfer operations is often significantly longer than the time to evaluate the logic for a single design clock cycle on the emulator.

Transaction-based host–emulator interfacing has been introduced as a way to reduce interface time [12]. In transaction-based interfacing, the host-based simulator and FPGA-based emulator operate independently for a number of design clock cycles, limiting the amount of data that must be transferred across the host–emulator interface. Transaction-based interfacing often works best for stream-based computations where dependencies between the simulated and emulated designs are minimal, allowing independent operation [27]. A detailed example of transaction-based coverification will be presented in Section 30.7.

For coverification environments, the simulation performed on the host workstation can take a variety of forms. Most commonly, an RTL or behavioral representation of a system component written in a hardware description language is simulated with a commercial HDL simulation tool. Following preliminary verification, some simulated components may then be synthesized and mapped to the logic emulator. Alternately, a software version of the simulated system components (typically in C/C++) may be used [27].

30.4.3 Logic Analysis

Logic analysis, the capturing of signal state around specific events of interest, plays an important role in FPGA-based logic emulation for both in-circuit emulation and coverification. Unlike processor-based logic simulation, which stores intermediate logic signals in a centralized memory, intermediate signals in FPGA-based emulation are physically distributed throughout the emulation system. As a result, for emulation the signal set of interest usually must be selected prior to compilation so that probing circuitry can be added to the design under test. The data collected by this circuitry can then be connected to an external logic analyzer or sent back to the host workstation for display. In some cases, combinational signals can be reconstructed from saved design flip-flop values via simulation once emulation is complete [20]. Signal reconstruction allows for a significant reduction in the amount of probe circuitry required within the logic emulator, and limits the amount of signal data transferred from the emulator after each design clock cycle.

Because of their cycle-accurate operation, logic analysis for FPGA-based emulators has several additional, unique characteristics:

- FPGA-based emulators can only perform functional verification, so only combinational and flip-flop values captured on design clock edges accurately indicate design behavior.
- If the set of design signals selected for probing is changed, one or more FPGAs may need to be recompiled to implement the change.
- Logic analysis for a design can be triggered by prespecified logic conditions in the design. This triggering circuitry can be added to the design under test.

Logic emulators can be used to evaluate millions of design clock cycles, so there often has to be a trade-off between the number of probes and the number of consecutive clock cycles probing is performed. If emulation can be stopped, intermediate probe values can be offloaded to the host workstation or to a disk. Emulation can then be restarted [20].

30.5 THE NEED FOR FAST FPGA MAPPING

Commercially available FPGAs are optimized to provide good performance and mapping efficiency to a wide range of user designs. As seen in Chapter 1, contemporary off-the-shelf FPGAs offer a diverse and flexible routing network to reach this goal. To achieve modest to high logic resource utilization (e.g., greater than 75 percent lookup table [LUT] usage) and high design performance, an FPGA's mapping tools must perform a detailed evaluation of FPGA placement and routing choices, typically requiring 30 minutes to several hours of compile time per device. As a result, most FPGA-based logic emulators suffer from long compile times, which is a major limitation to their widespread deployment. The presence in an emulator of hundreds of FPGAs with significant compile times can considerably delay the debug, redesign, and retest cycle for a design under test. As noted in Chapter 20, several research projects have investigated accelerated FPGA mapping to solve this problem.

There are several reasons why fast FPGA design mapping for logic emulation is important:

1. The sheer number of FPGAs needed for logic emulation necessitates fast compilation. If compilation can be accelerated by an order of magnitude, so too, roughly, can the turnaround time from design change to emulator implementation. For many systems, faster design turnaround time can make a substantial difference in emulator usability, especially early in the design cycle when design errors are more prevalent.

2. A fast mapping is useful for determining if all logic partitions will fit within emulation system FPGA devices. If any partition fails to map into the emulator, the entire emulation mapping flow typically must be restarted from scratch.

3. Because multiplexed-wire emulation systems require the use of a synchronous global clock to coordinate computation and communication, the overall system clock speed is dependent on the slowest FPGA. A fast evaluation of achievable clock speed is therefore important. A fast mapping helps identify if the partitions are likely to meet the emulator's target system clock speed.

4. The inclusion of probes, which are frequently changed, necessitates a fast design compilation turnaround. Changes generally affect only a small number of FPGAs, which usually can be recompiled quickly.

Of the emulation system mapping steps shown in Figure 30.5, the individual FPGA compiles collectively require over 90 percent of the total compilation time. However, unlike the other steps, individual FPGA compiles can be easily distributed to multiple PCs and workstations for parallel compilation [9]. A centralized server is used to control distribution of the compiles to the client workstations, collect the resulting FPGA configuration bitstreams, and verify that all compilation constraints have been met.

It will be difficult to significantly accelerate compilation for FPGAs with existing commercial architectures without a substantial increase in the ratio of routing resources to logic resources per device or improved parallel mapping approaches for individual FPGAs. Fundamentally, FPGA placement and routing are dedicated resource assignment problems, and the search for a mapping solution is accelerated only through additional available resources or a parallel search. Although compile times for logic emulation can be significantly reduced by underpopulating commercial FPGA device logic in emulators, the hardware cost involved is prohibitive. Therefore, parallel FPGA placement and routing offer the most promise in improving compile times for existing FPGA architectures.

In many ways, FPGA compilation for a partition of an emulated design under test is more difficult than FPGA compilation for a single-chip design specifically created for an FPGA. All FPGA compiles for logic emulators must be performed with constrained pin assignments because inter-FPGA channel assignments are determined prior to individual FPGA compilation. Forced pin assignments make designs more difficult to map and require extended FPGA compilation times. Since partitions were not specifically designed for an FPGA, performance or utilization issues may sometimes arise during mapping.

30.6 CASE STUDY: THE VIRTUALOGIC VLE EMULATION SYSTEM

To illustrate many of the issues in logic emulation, we consider the VirtuaLogic VLE emulator from Mentor Graphics [9]. This system represents one point in a spectrum of similar FPGA-based emulation systems from Mentor Graphics, including the Avatar and the VStation [23]. The following analysis illustrates the basic approaches used by this family for system architecture, design compilation, external system interfacing, and coverification.

30.6.1 The VirtuaLogic VLE Emulation System Structure

Figure 30.10 illustrates the components of the VLE emulation system hardware, including its interfaces to a host workstation and target system [9]. The system chassis, shown on the right, can contain up to six multi-FPGA array boards, which emulate the logic and memory of a design under test. Two array boards are shown in the configuration in the figure. Each board contains 64 Xilinx XC4036XL FPGAs, arranged in an 8 × 8 array, and 32 32K × 32 single-port synchronous SRAM chips. As shown in Figure 30.11, each FPGA connects to its four nearest neighbors in both horizontal and vertical directions and to FPGAs two hops away in the horizontal and vertical directions. A single memory device is shared between each pair of FPGAs. Direct connections between each FPGA and the six I/O connectors on the array board provide an interface for in-circuit emulation connections, logic analysis, and host interfacing. As shown in Figure 30.10, these connectors are located at the front of each board.

The FPGA array boards connect to a passive backplane in the system chassis to create a scalable system. Each FPGA has direct connections through the backplane to FPGAs on other array boards. All intra-FPGA computation and inter-FPGA communication throughout the system is coordinated via a global

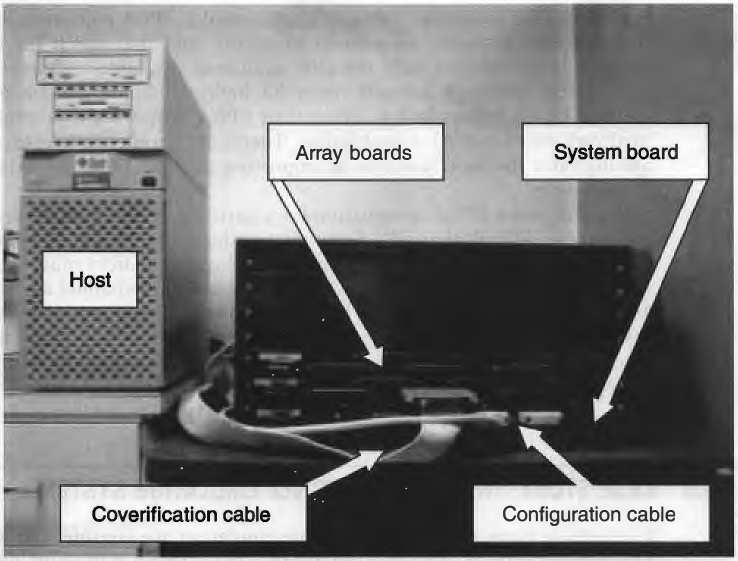


FIGURE 30.10 ■ A VirtuaLogic VLE-2M logic emulation system with two array boards.

system clock. The system board in the emulator controls the configuration of array board FPGAs and coordinates the distribution of the global system clock. Configuration bitstreams are loaded into the system board from the host workstation via an SCSI-2 cable.

30.6.2 The VirtuaLogic Emulation Software Flow

The emulation mapping flow for the VirtuaLogic VLE system follows the flow outlined earlier in this section. During design translation, an RTL netlist is converted to a gate-level design through the use of RTL synthesis. The mapped netlist is then partitioned into pieces appropriate for the logic capacity of each FPGA using algorithms that attempt to minimize bandwidth and encapsulate critical design paths within individual FPGAs.

Partitioning is performed so that the logic capacity of the FPGA is considered while partitioning to minimize bandwidth [1, 8]. For the multiplexed-wire VLE system, the number of logic gates required per partition can be represented as

$$G \geq G_P + c^*P$$

where G is the number of available gates in the FPGA, G_P is the number of user design logic gates in the partition, c is a constant representing the amount of

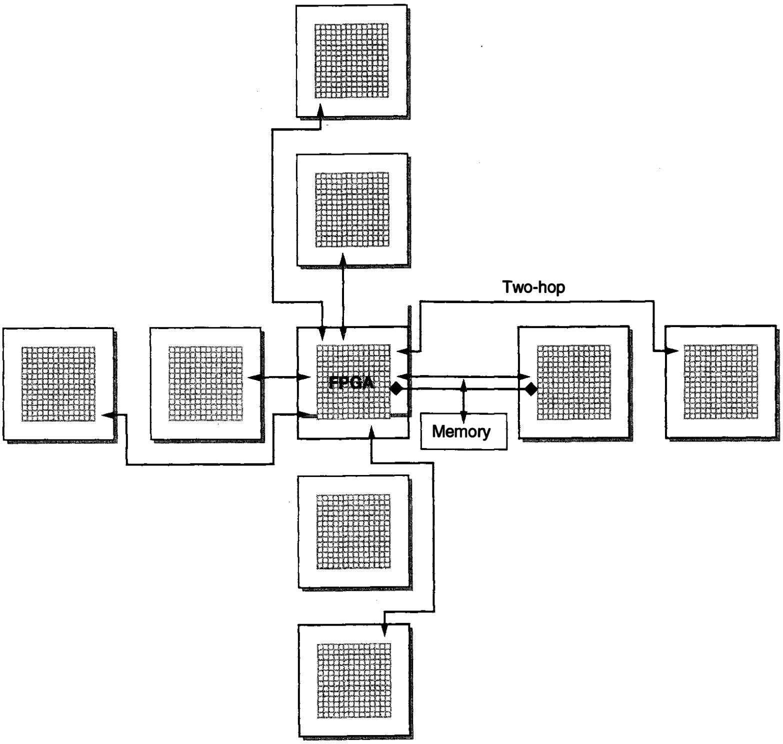


FIGURE 30.11 ■ The array board connections for an FPGA in the VLE logic emulation system.

logic required to multiplex a pin, and P is the number of I/O signals associated with the partition.

Design partitions assigned to an FPGA have a required gate count that is less than G. The partitioning process for the VLE system starts with an initial assignment of logic to partitions. Iterative mincut swapping is then performed to reduce the amount of I/O needed by each partition (the value P in the equation). Not only does this optimization reduce the amount of subsequent pin multiplexing for I/Os, but the amount of required logic per device is also reduced because G depends on P [8]. Partitions for this emulation system are subsequently placed using a simulated annealing placement algorithm [30]. In general, placement is performed to minimize the overall distance of inter-FPGA connections assuming that all connections will be scheduled along shortest paths. The logic partition to FPGA assignment formulation is similar to the one used to place clusters inside an island-style FPGA.

A distinctive aspect of the VLE system is the statically scheduled routing approach used to make connections between signal sources and destinations. The approach used by the VirtuaLogic compiler follows that described in Section 30.4 [8, 34]. All intra-FPGA computation and inter-FPGA communication is synchronized to the global system clock cycle so that multiple system clock cycles are required to complete an emulation clock cycle. A signal may be routed between FPGAs on a specific system clock cycle once it is known to be valid for the current emulation cycle based on signal dependencies. The following steps are then taken to perform the statically scheduled routing of the signal between a source FPGA s_f and a destination FPGA d_f [34]:

1. The shortest feasible path P_{sd} between FPGAs s_f and d_f in terms of inter-FPGA channels is determined.
2. The send time T_s of the signal is determined. This is the system clock time slot at which the signal leaves s_f.
3. The signal arrives at FPGA d_f at the arrival time T_a of the signal. The arrival time is defined as $T_a = T_s + n$, where n is the number of FPGA chip boundaries (hops) between source FPGA s_f and destination FPGA d_f.

To illustrate the use of T_s and T_a, the schedule of the circuit shown in Figure 30.8 can be augmented to include send and arrival times. The communication schedule, including T_s and T_a values, is shown in Figure 30.12. Note that in Figure 30.8 signal b passes unchanged through FPGA F_2 on the path from

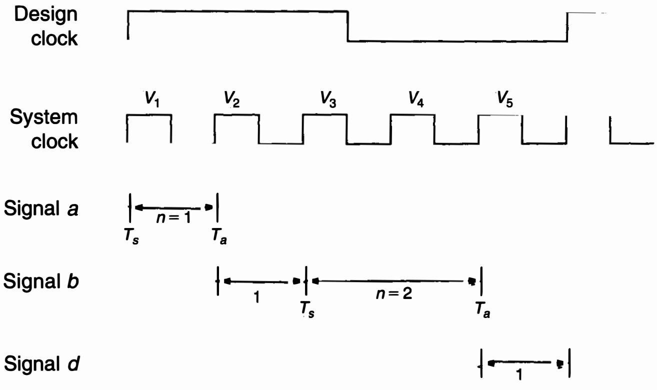


FIGURE 30.12 ■ The design clock cycle for the circuit mapping shown in Figure 30.8, including send times T_s and arrive times T_a.

FPGA F_3 to FPGA F_1. This through-hop is necessary given the lack of a direct FPGA F_3 to FPGA F_1 connection.

After each interpartition signal is scheduled for communication, the chosen schedule is implemented by synthesizing multiplexers, registers, and state machines that are added to the circuit partition for each FPGA. The resulting circuits are then applied to standard Xilinx Foundation design-mapping tools [37].

Most ASIC designs that are targeted for emulation contain complex logic and memory structures that require specialized processing outside the standard emulation mapping flow. For VLE systems, specialized mapping techniques have been developed to map complex design memories to emulation system memory chips [1], to map designs that contain multiple asynchronous design clocks [13], and to incrementally map design changes [34]. The algorithms created to address these mapping issues are important keys to system usability.

30.6.3 Multiported Memory Mapping

In a VLE system, multiple accesses to a 32K × 32 synchronous single-ported SRAM can be scheduled within a design (emulation) cycle to emulate the behavior of a multiport RAM. For example, Figure 30.13(a) shows a user-specified dual-port memory with two read ports and a single write port. During an emulation cycle access that requires reads from both read ports, both reads can be performed in sequence from the single-ported SRAM chip. As shown in Figure 30.13(b), a state machine can be used to sequence the application of the addresses to the single-ported SRAMs, and the storage of the read data in the output registers.

The VirtuaLogic compiler determines the schedule for data accesses in conjunction with routing address, data, and control signals to the on-board physical memory devices. Although not shown in the Figure 30.13, for data wider than the width of the physical memory, memory accesses can be made by sequentially accessing consecutive memory locations. For example, a read of a 128-bit value requires four system clock cycles. Dependency relationships for multiported RAMs (e.g., read-after-write) can be handled via the sequential scheduling of RAM accesses.

30.6.4 Design Mapping with Multiple Asynchronous Clocks

In Section 30.4 it was shown that for multiplexed-wire systems both intra-FPGA computation and inter-FPGA communication are coordinated to a global system clock. Because multiple system clock cycles are required to perform computation and communication for a single emulation clock cycle, a fixed relationship must exist between the clocks. Many contemporary ASIC designs contain multiple design clocks that operate asynchronously to each other. While synchronization between a system clock and a single design clock can be addressed by rising design clock edges that delineate functional evaluations, deriving a relationship between multiple asynchronous design clocks and a system clock is more difficult.

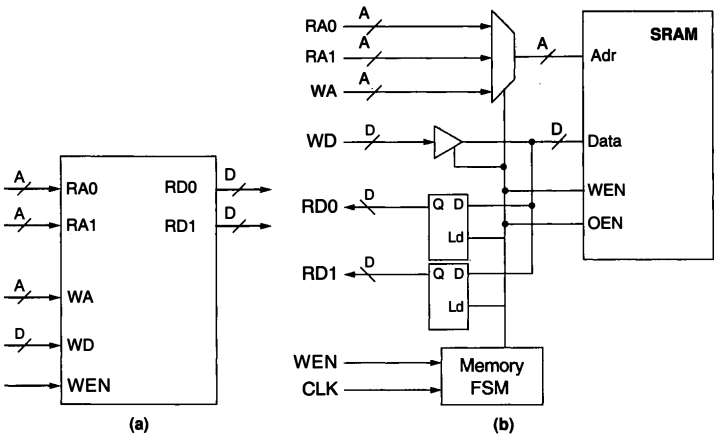


FIGURE 30.13 ■ A mapping of a multiported design memory to a single-ported emulator memory: (a) parallel-accessed multiport memory; (b) sequentially accessed single-port multiplexed memory. *Source:* Adapted from Agarwal [1].

In the circuitry shown in Figure 30.14, taken from Kudlugi and Tessier [13], the asynchronous clocks CLK1 and CLK2 drive state elements. It can be seen that signal N5 is a multidomain signal because it changes value and is sampled as a result of both CLK1 and CLK2 clock transitions. Now consider a situation where the circuit in Figure 30.14 is partitioned so the multidomain signal N5 must be transported from FPGA 1 to FPGA 4 as shown in Figure 30.15. In a multi-FPGA VLE system, the physical wires that connect FPGAs are grouped into unidirectional channels, where each physical wire is capable of carrying multiple signals that belong to the same emulation clock domain (e.g., CLK1 or CLK2).

Signal routing may include several intermediate FPGA hops. To simplify scheduling, logical signals assigned to the same inter-FPGA wire must be associated with the same clock domain. For designs with multidomain signals, this restriction requires that each multidomain signal be logically split into separate single-domain versions prior to transport. These single-domain values are then transmitted separately along separate physical channel links and combined at the destination to support multidomain behavior. Unfortunately, this approach of separately routing copies of the same signal along different links can lead to scheduling problems because each copy may arrive at the destination at different system clock cycles.

This issue is best illustrated through an example. As shown in Figure 30.15, communication for each asynchronous clock domain takes place over a different

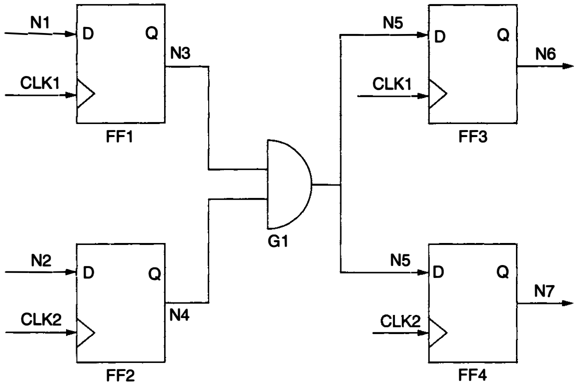


FIGURE 30.14 ■ A circuit that requires clocks from multiple asynchronous clock domains.

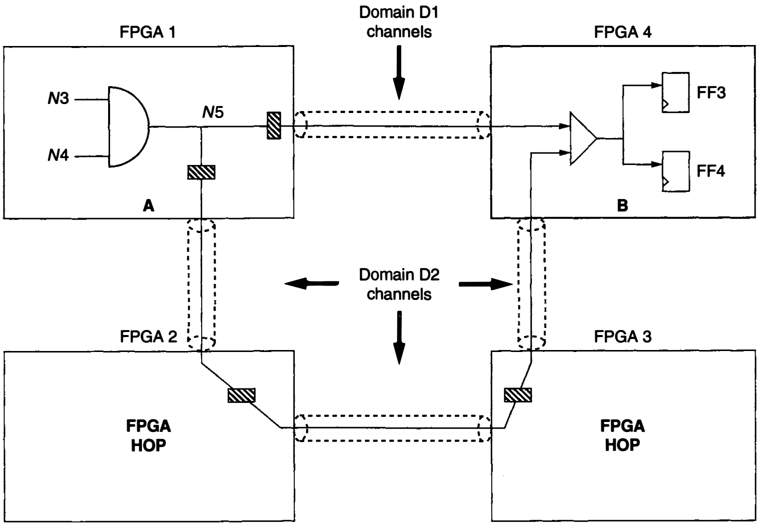


FIGURE 30.15 ■ An example of multidomain signal transport. *Source:* Adapted from Kudlugi and Tessier [13].

set of inter-FPGA channels. In the case of N5, paths using both domain 1 (D1) and domain 2 (D2) channels are needed to transport N5 between FPGA 1 and FPGA 2. The disjoint nature of multiple routing paths for the same logical signal can lead to differing arrival times for the copies of signal N5 at the destination FPGA. If both copies of signal N5 leave FPGA 1 at the same time, the D1 version of the signal will arrive at FPGA 2 two system clock cycles before the D2 version. This arrival order can lead to an incorrect logic evaluation if an attempt is made to use the D1 version of the signal before the D2 version arrives.

A requirement in transporting multidomain signals is to ensure that causality of events is guaranteed irrespective of routing delays. Causality can be preserved by ensuring that the length of the route for each domain from the source to the destination requires exactly the same number of system clock cycles. This can be accomplished by requiring the scheduler to use the same number of system clock cycles to communicate versions of the same signal to a destination FPGA. In Figure 30.16, for example, the scheduler must determine a path from FPGA 1 to FPGA 2 of length 3 for domain D1, since this is the path length of the domain D2 version. Each path now contains three pipeline flip-flops. The determination of the specific schedule may require several scheduling iterations because the length of the longest path is not known until each path is initially scheduled.

The scheduler used by the VirtuaLogic compiler takes multidomain paths into account and can handle designs with any number of asynchronous clock

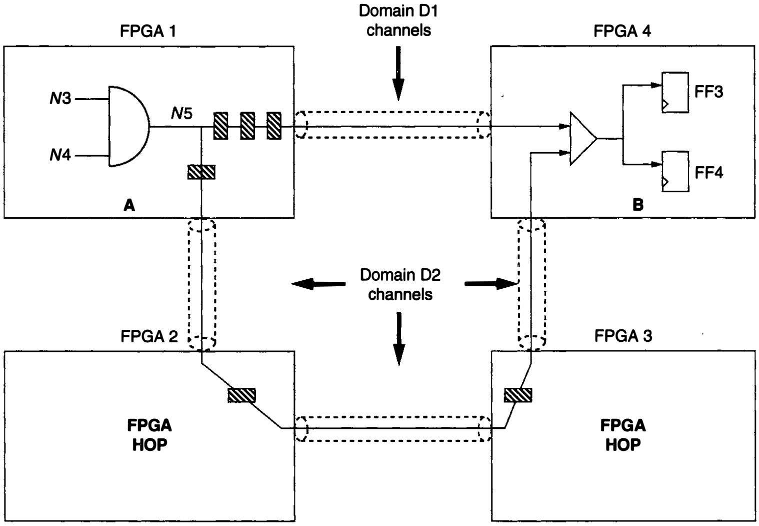


FIGURE 30.16 ■ A retimed version of the multidomain signal transport shown in Figure 30.15.

domains. The mapping of this multidomain logic to the emulator takes place automatically. The asynchronous design clock signals may be interfaced to the emulator from outside the system through the system board.

30.6.5 Incremental Compilation of Designs

The need for incremental design support in VLE systems is a result of recent interest in core-based design and system-on-a-chip integration. Most ASIC verification flows involve numerous iterations of design test, debug, and recompilation. As modifications are evaluated and errors are identified, the original design is subjected to a series of minor modifications. Often, a change may be isolated to a component that was originally spread across two or more FPGAs in the emulator. If emulator recompilation can be limited primarily to those FPGAs that contain logic affected by the change, the compilation process can be greatly accelerated. The ability to support design changes in a small set of FPGAs is crucial to avoid the need to recompile all FPGAs in the system from scratch. In addition to providing fast design turnaround, the resulting emulation performance of the incrementally compiled design should be the same or close to the same as the performance of the original design mapping [34].

The use of scheduling for VirtuaLogic inter-FPGA routing facilitates the management of incremental design compilation. A series of steps are required to address changes in the design and map them to the FPGA-based emulator [34]:

1. *Netlist comparison.* The first step in the incremental compilation process is to identify the logic and interconnect associated with the initial design that is no longer in the modified design. Subsequently, the logic and interconnect added to the initial design to create the modified design are identified. Logic removed from the initial design was assigned to a set of FPGAs as a result of initial design mapping. These modified FPGAs provide a possible destination for added logic.

2. *Incremental path identification.* In the VLE system, individual FPGAs may serve as through-hop steps for intermediate routes. Thus, even if a given FPGA does not contain logic that has changed, these FPGAs will require recompilation if they are used as through-hops for the modified logic. To limit compile time, the number of unmodified FPGAs selected to perform through-hop routing should be minimized.

3. *Incremental partitioning*. Once the modified and required through-hop FPGAs have been identified, newly added design logic can be partitioned onto them subject to processor logic and memory capacity constraints.

4. *Incremental routing*. Following incremental partitioning, routing is performed to create a path for the added design signals connecting the modified FPGAs. Because FPGAs surrounding the modified FPGAs are unaltered, this incremental routing must be performed using board-level routing resources that have not been consumed by unchanged design routes. Feasible shortest paths between FPGAs are evaluated and then incremental scheduling is used to form a communication pipeline.

The most important part of incremental compilation for multiplexed-wire systems is the scheduling of added signals onto available inter-FPGA wires (incremental routing). In some cases, portions of previously routed inter-FPGA links may need to be rerouted as a result of changed logic depth and dependency. Consider the circuit shown in Figure 30.17, taken from Tessier and Jana [34]. The circuit is the same as the one assigned to FPGAs in Figure 30.8 except that the OR gate *F* and signals *e* and *f* have been added. One potential incremental mapping for the modified circuit appears in Figure 30.18. A design clock

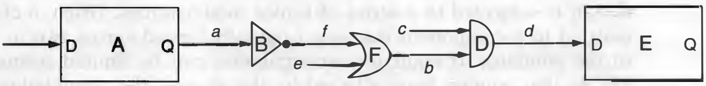


FIGURE 30.17 ■ A modified version of the circuit assigned to FPGAs in Figure 30.8. *Source:* Adapted from Tessier and Jana [34].

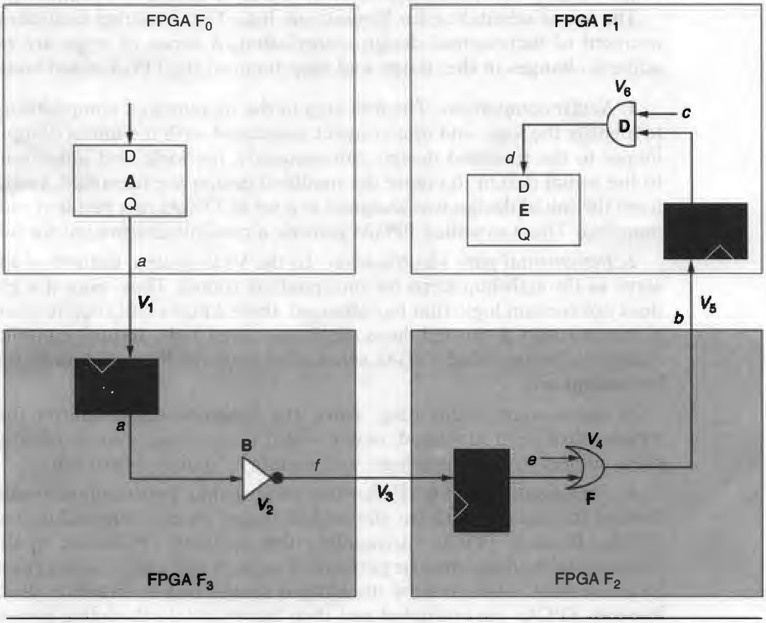


FIGURE 30.18 ■ An incremental mapping of the circuit shown in Figure 30.17.

cycle associated with the scheduled route of the circuit mapping in Figure 30.17 is shown in Figure 30.19.

When these waveforms are compared to the waveforms in Figure 30.12, it can be seen that an extra cycle of combinational delay has been added because of the OR gate evaluation in FPGA F_2, extending the number of system clock cycles needed to evaluate the design. Closer examination of the two sets of waveforms indicates that although signal *b* was previously routed between FPGA F_2 and FPGA F_1 in the initial design, it will have to be rerouted for the modified mapping. For the initial design, signal *b* has been routed between FPGA F_2 and FPGA F_1 on system clock cycle V_4. As a result of the mapping shown in Figure 30.18, signal *b* cannot be routed until system clock cycle V_5 because of combinational dependencies. This results in a need to recompile both FPGA F_2 to transmit the signal on cycle V_5 and FPGA F_1 to receive the value on system clock cycle V_5.

After dependencies are determined, the new links are scheduled for communication using the VirtuaLogic compiler two-step routing approach described earlier. Only added interpartition signals are routed; previously routed signals that are unchanged are left in place. Incremental routing of added signals may lead to an emulation system performance loss. For example, the waveforms shown in Figure 30.19 represent the schedule of the incrementally modified design shown in Figure 30.17. The new schedule requires six system clock cycles to complete a design clock cycle as opposed to the five required for the original design. Although not shown in Figure 30.19, a global control signal distributed to all FPGAs indicates the end of the design clock cycle. Following recompilation, this signal can be asserted every six rather than five system clock cycles. This requires FPGAs

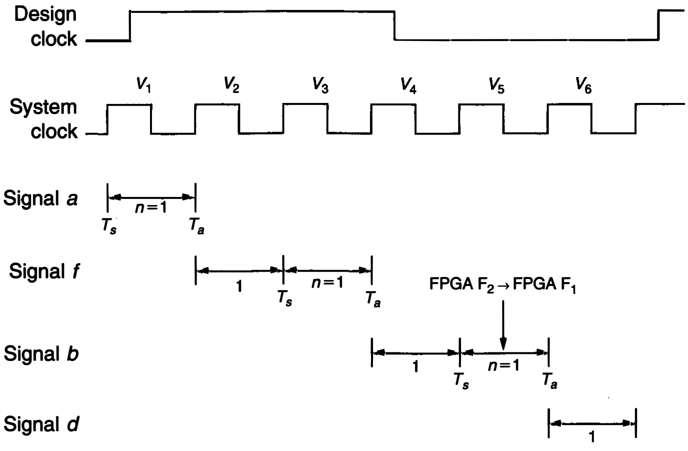


FIGURE 30.19 ■ The design clock cycle for the incremental mapping shown in Figure 30.18.

that were not recompiled to hold data values for an extra system clock cycle while the recompiled FPGAs complete computation. All results are then clocked into design flip-flops system-wide after six clock cycles by the design clock.

30.6.6 VLE Interfaces for Coverification

The VLE system has a number of interfaces to support both in-circuit emulation and coverification. For in-circuit emulation, an emulation pod can be interfaced to one of six connectors on each of the array boards sho\.n in Figure 30.10. These signals are directly connected to FPGAs and drive/receive I/O signals on the emulated design. Tuned clock cables are used to control clocking both on the target system and in the emulator when the emulator has completed evaluation for an emulation clock. To permit in-circuit emulation the target system must be slowed to accommodate the 0.5- to 2-MHz design emulation rate.

In addition to support for in-circuit emulation, the VLE emulator has significant support for a variety of coverification modes. This support is primarily provided through a series of software interfaces created at the host workstation and on the emulator. These interfaces allow the emulator to be used in a variety of coverification scenarios [9]. Designers initiate ASIC verification by representing the ASIC using a high-level language such as C or SystemC (a C-compatible language that represents the concurrency and clocking associated with hardware implementations). As a design matures, portions of it are migrated to hardware. Inputs and outputs to the portion of the design on the emulator are interfaced to the emulator via an application programming interface (API).

The transfer, execution, and collection of results using the emulator can be represented as shown in Figure 30.20. This implementation of coverification is performed with a series of components. The software test environment interacts with an application adapter—that is, an interface to a series of library-based drivers that packetize the data and prepare it for transfer via a PCI-based board. The use of library-based drivers allows for communication at functional, bus-cycle-accurate, and cycle-accurate levels [27].

An interface circuit is required at the destination to reassemble data for subsequent use as input to the design. A transactor accepts the reassembled data, generates an emulation clock for use with the design under test, and coordinates per-cycle data transfer to and from the design. Generally, the interface circuit and transactor are created in RTL and added to the design. VLE systems use the transaction-based approach described earlier in this section. Transactions contain both data and synchronization information. A single transaction results in multiple verification cycles of work being performed by the emulator. The transaction can be as simple as a memory read or as complex as the transfer of an entire structured packet through a channel. To support coverification, the host for the VLE emulator contains an SPCI (Springtime PCI) card [27]. This custom PCI card implements the physical layer of transaction-based interfacing between the host and the emulator via a cable.

The transaction application protocol interface (TAPI) forms the application adapter for the VLE system [27]. TAPI consists of a library of C functions. The

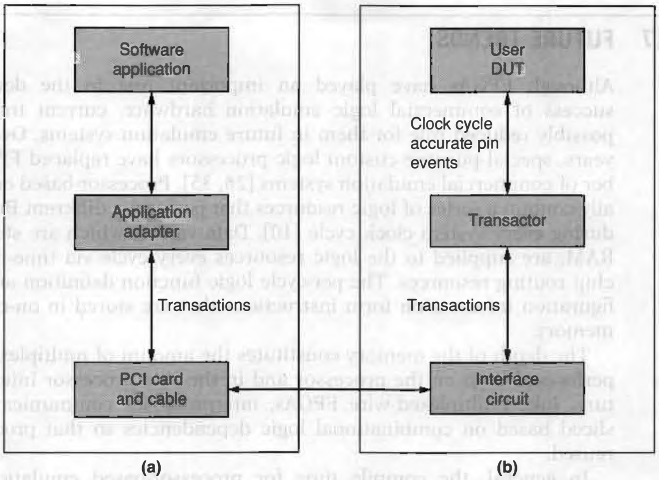


FIGURE 30.20 ■ The coverification flow between the workstation (a) and the emulator (b).

adapter is a utility package that converts raw signals into transactions by making calls to the C function library. It supports a verification environment that allows a C model to interact with an RTL model running on the emulator. The transfer of data across the host–emulator cable can be aided by buffering data in memory and transferring it as a block. This approach is preferable to the individual transfer of values from discrete memory locations in a file. Data buffering in arrays can be implemented in the same C modules that contain the TAPI driver calls for the emulator.

For the VLE system, the emulator system clock speed is set to 30 MHz. The same six multi-FPGA board connectors used for interfacing to an in-circuit emulation pod can also be used as an interface for coverification. The remaining connectors on the multi-FPGA boards can allow for direct access to logic analyzers for signal probing.

30.6.7 Parallel FPGA Compilation for the VLE System

Given the number of FPGAs in the VLE system, parallel compilation of the individual devices is a necessity. An FPGA compile server is used to distribute the numerous Xilinx XC4036XL compiles out to a number of available workstations that can perform the needed operations [9]. Unfinished compiles are held in a queue until compilation resources become available. Following design compilation, configuration bitstream information and status reports are returned to the server for subsequent transfer to the emulation system.

30.7 FUTURE TRENDS

Although FPGAs have played an important role in the development and success of commercial logic emulation hardware, current trends indicate a possibly reduced role for them in future emulation systems. Over the past few years, special-purpose custom logic processors have replaced FPGAs in a number of commercial emulation systems [26, 35]. Processor-based emulators generally contain a series of logic resources that perform a different Boolean function during every system clock cycle [10]. Data values, which are stored in on-chip RAM, are supplied to the logic resources every cycle via time-multiplexed on-chip routing resources. The per-cycle logic function definition and routing configuration information form instructions that are stored in on-chip instruction memory.

The depth of the memory constitutes the amount of multiplexing that can be performed both on the processor and in the interprocessor interconnect structure. Like multiplexed-wire FPGAs, interprocessor communications are time-sliced based on combinational logic dependencies so that processor pins are reused.

In general, the compile time for processor-based emulation is very fast compared to FPGA-based emulation. This disparity is a result of the assignment of intra-FPGA (processor) logic to interconnect resources. In multiplexed- and dedicated-wire emulation systems, internal FPGA logic and interconnect are *dedicated* to specific design resources. This has three implications:

1. For long combinational paths, each logic block and intra-FPGA wire is used only a small fraction of the time, effectively limiting system efficiency.
2. The dedicated assignment of signals to intra-FPGA wires is a problem of limited resource allocation. To significantly reduce compile time, a substantial increase in routing resources is needed relative to available logic to make FPGA routing linear time (a value of at least 20 percent is reported by Swartz et al. [33]). According to Rent's Rule, this disparity is likely to become worse as designs and FPGAs increase in size.
3. Because FPGA routers are unpredictable, it is impossible to determine both whether a device will route and what the per-FPGA (and hence global system) performance will be until all FPGAs have been successfully mapped.

In contrast, in processor-based emulator hardware, internal logic and routing structures are time-multiplexed. As a result, simpler routing structures with fixed memory to processor delays for all intra-processor paths are set. This, too, has implications:

1. Logic and interconnect resources are multiplexed over time to increase resource use efficiency per clock cycle.
2. The assignment of both inter- and intra-FPGA resources is a scheduling problem. Unlike search-based FPGA routing, scheduling algorithms

typically can be performed quickly and have runtimes largely proportional to circuit combinational depth.

3. The global system clock period is fixed by the architecture of the device, not by individual designs.

Specialized logic processors have other potential benefits. Specialized circuitry for signal probing and coverification transactions do not have to be fashioned out of generic FPGA logic, but rather can be customized to limit silicon overhead and optimize speed.

FPGA-based emulators do have some advantages. In some cases, they may provide more parallelism for certain designs that have shallow combinational depth. Rather than multiplexing logic resources, FPGAs can perform all logic operations simultaneously. The use of specialized logic processors in emulation introduces additional overhead for the emulation system provider. Because FPGAs typically use the newest silicon fabrication processes, specialized logic processors are likely to be at least one silicon generation behind the state of the art. Additionally, mapping tools for the logic processors must be developed and maintained by the emulation company rather than by the FPGA vendor. Recent trends indicate that despite these issues, the benefits of orders of magnitude faster compile time are driving emulation vendors in the direction of special-purpose logic processors.

Several developments in the design of FPGAs may swing this trend back in their favor. Recent FPGAs provide high-speed I/Os such as low-voltage differential signaling (LVDS) that support rapid I/O multiplexing. Additionally, the introduction of fixed cores, such as multipliers and microprocessors, may provide faster mapping and higher performance for emulation once they are integrated in the emulator compilation flow.

30.8 SUMMARY

FPGA-based logic emulation is a distinct example of a commercially successful reconfigurable computing application. A key aspect of its success has been the development of sophisticated software systems that can seamlessly map a large ASIC design to hundreds of FPGAs with minimal or no designer intervention. An important characteristic of most multi-FPGA emulators is the scheduling of both intra-FPGA computation and inter-FPGA communication in concert with a global system clock. The use of scheduling overcomes limited FPGA pin resources and takes advantage of signal dependencies, so that only portions of a design are active at a given time. Contemporary multi-FPGA logic emulators are used as both physical replacements in circuit and as coverification engines to accelerate design simulation. These supporting environments have advanced in recent years to include multiple asynchronous clock domains and support for incremental design changes.

Extended compile times are quickly becoming a dominant issue for FPGA-based emulators, and have motivated the development of fast FPGA compile approaches. Although emulation systems with custom-designed logic processors have been developed, recent FPGA trends and faster compile approaches may spur renewed interest in FPGA-based emulation.

References

[1] A. Agarwal. *VirtualWires: A Technology for Massive Multi-FPGA Systems*, Mentor Graphics Corp., 2002.

[2] J. Babb, R. Tessier, M. Dahl, S. Hanano, D. Hoki, A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16(6), June 1997.

[3] M. Butts. Future directions of dynamically reprogrammable systems. *IEEE Custom Integrated Circuits Conference*, May 1995.

[4] C. Chang, K. Kuusilinna, B. Richards, R. Broderson. Implementation of BEE: A real-time, large-scale hardware emulation engine. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2003.

[5] S. Hauck, G. Borriello. Logic partition orderings for multi-FPGA systems. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1995.

[6] S. Hauck, G. Borriello. An evaluation of bipartitioning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16(8), August 1997.

[7] S. Hauck. The role of FPGAs in reprogrammable systems. *Proceedings of the IEEE* 86(4), April 1998.

[8] S. Hauck, A. Agarwal. *Software Technologies for Reconfigurable Systems*, Technical report, Department of ECE, Northwestern University, 1996.

[9] IKOS Systems. *VirtuaLogic VLE Emulation System Manual*, 2001.

[10] D. Jones, D. Lewis. A time-multiplexed FPGA architecture for logic emulation. *IEEE Custom Integrated Circuits Conference*, May 1995.

[11] H. Krupnova, G. Saucier. FPGA-based emulation: Industrial and custom prototyping solutions. *International Conference on Field-Programmable Logic and Applications*, August 2000.

[12] M. Kudlugi, S. Hassoun, C. Selvidge, D. Pryor. A transaction-based unified simulation/emulation architecture for functional verification. *ACM/IEEE Design Automation Conference*, June 2001.

[13] M. Kudlugi, R. Tessier. Static scheduling and multidomain circuits for fast functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21(11), November 2002.

[14] J. Kumar. Prototyping the M68060 for concurrent verification. *IEEE Design and Test of Computers* 24(1), January 1997.

[15] I. Kuon, J. Rose. Measuring the gap between FPGAs and ASICs. *International Symposium on Field-Programmable Gate Arrays*, February 2006.

[16] Y. Kwon, C. Kyung. Performance-driven event-based synchronization for multi-FPGA simulation accelerator with event time-multiplexing bus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24(9), September 2005.

[17] B. Landman, R. Russo. On a pin versus block relationship for partitioning of logic graphs. *IEEE Transactions on Computers* C20(12), December 1971.

[18] C. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers* EC-10(2), September 1961.

[19] J. Li, C.-K Cheng. Routability improvement using dynamic interconnect architecture. *IEEE Workshop on FPGA-Based Custom Computing Machines*, April 1995.

[20] J. Marantz. Enhanced visibility and performance in functional verification by reconstruction. *ACM/IEEE Design Automation Conference*, June 1998.

[21] Mentor Graphics Corp. *VirtuaLogic Datasheet*, 2002.

[22] Mentor Graphics Corp. *VStation Datasheet*, 2004.

[23] Mentor Graphics. Emulation products web page: *http://www.mentor.com/emulation*, April 2006.

[24] Quickturn Design Systems. *System Realizer Data Sheet*, 1998.

[25] Quickturn Design Systems. *Mercury Data Sheet*, 1999.

[26] Quickturn Design Systems. *Cobalt Systems User Guide*, 2001.

[27] R. Ramaswamy, R. Tessier. The integration of SystemC and hardware-assisted verification. *International Conference on Field-Programmable Logic and Applications*, September 2002.

[28] K. Roy-Neogi, C. Sechen. Multiple FPGA partitioning with performance optimization. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1995.

[29] M. Santarini. ASIC prototyping: Make versus buy. *EDN*, November 21, 2005.

[30] K. Shahookar, P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys* 23(1), June 1991.

[31] G. Snider, P. Kuekes, W. Culbertson, R. Carter, A. Berger, R. Amerson. The Teramac configurable compute engine. *International Conference on Field-Programmable Logic and Applications*, August 1995.

[32] H. Su, Y. Lin. A phase assignment method for virtual-wire-based hardware emulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16(7), July 1997.

[33] J. S. Swartz, V. Betz, J. Rose. A fast routability-driven router for FPGAs. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1998.

[34] R. Tessier, S. Jana. Incremental compilation for parallel verification systems. *IEEE Transactions on VLSI Systems* 10(5), October 2002.

[35] Tharas Systems. *Tharas Hammer Product Brief*, 2002.

[36] P. Tseng. Reconfigured engines REV simulation. *EE Times*, July 10, 2000.

[37] Xilinx, Inc. *Xilinx Foundation Tools User Guide*, 2002.

CHAPTER 31

THE IMPLICATIONS OF FLOATING POINT FOR FPGAS

Keith D. Underwood, K. Scott Hemmert
Sandia National Laboratories

FPGA-based computing has a long history of accelerating assorted types of computations in integer and fixed-point arithmetic. Until recently, however, applications based on floating-point arithmetic have been a relative rarity. This stems from early work [6, 12, 13] that indicated that IEEE-754 standard [11] floating point was a poor match for field-programmable gate array (FPGA) technology. This led directly to numerous efforts that created libraries using specialized floating-point formats [1, 3, 7], where the width of the exponent and the width of the mantissa could be specified. Unfortunately, many scientific applications require compliance with the IEEE standard. While seemingly an arbitrary requirement, it is driven by several factors. Foremost, some scientific applications have data with high dynamic ranges and high precision requirements. A good example is a typical linear solver that needs high precision to guarantee convergence of the algorithm. Second, application developers rely on the portability of their applications and the reproducibility of their results. Put another way, it is difficult to trust results that differ on every platform that runs them.

Fortunately, recent work indicates that FPGAs are viable competitors in IEEE-compliant floating-point arithmetic [14], and there has been an explosion of interest in mapping floating-point kernels to FPGA platforms [2, 5, 8–10, 15, 17, 18]. However, while FPGAs are now capable of implementing floating-point applications, the use of floating point in FPGAs still requires a great deal of care. This chapter introduces the IEEE floating-point standard and discusses implementations of compliant floating-point units for FPGAs. Section 31.2 contains case studies of three floating-point application kernels and their implementation on FPGAs.

31.1 WHY IS FLOATING POINT DIFFICULT?

Floating-point arithmetic is fundamentally different from typical integer or fixed-point arithmetic. Where integer and fixed-point values are typically stored in 2's

Note: Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

complement, floating-point numbers are typically stored in signed-magnitude format. Floating-point numbers also add an exponent field to control the position of the decimal point in the value. The most widely used floating-point format is the IEEE-754 standard. As an example, the IEEE double-precision floating-point format is shown in Figure 31.1. The mantissa (fraction part) is 52 bits, the exponent is 11 bits, and the sign is a single bit. A simple picture, however, cannot tell the full story of the complexities of the IEEE format.

First, as the figure suggests, the exponent in the IEEE format is maintained in *biased* notation. That is, rather than being in a signed-magnitude or 2's complement format, a *bias* is added to the true exponent to store it. For double precision, the bias is 1023 (approximately half the range). This means that an exponent of -1022 is stored as a 1. The second complication in the format is the use of an *implied 1*. An implied 1 means that the stored number is maintained in a normalized format such that there is a 1 immediately to the left of the decimal and the decimal is immediately to the left of the stored value. This allows the format to have an extra bit of precision without having to store it. Thus, the value can be extracted as shown in equation 31.1.

$$(-1)^S \times 2^{exp-bias} \times 1.mantissa \tag{31.1}$$

The format, as discussed so far, would have a major shortcoming. The number 0 would be impossible to represent. Since humanity has had the use of 0 for a few millennia now, the format inventors thought it best to include it by reserving a special value. They also saw fit to include representations for ∞, −∞, and not-a-number (NaN), which is used as the result of meaningless operations (e.g., ∞ × 0). The reserved special values are summarized in Table 31.1.

As the table implies, both positive and negative 0 are possible (0 and 1 for the sign bit, respectively) as are positive and negative infinity. Several values require that the maximum possible value be loaded into the exponent field (i.e., all bits are set to 1 in the field). Finally, there is a set of values known as denormals.

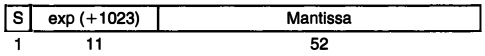


FIGURE 31.1 ■ IEEE double-precision floating-point format.

TABLE 31.1 ■ Special values in the IEEE-754 format

Special value	Sign	Exponent	Mantissa
Zero	0/1	0	0
∞	0	MAX	0
−∞	1	MAX	0
NaN	0/1	MAX	nonzero
Denormal	0/1	0	nonzero

Denormals are a special form of IEEE floating-point numbers that provide a small amount of extra precision as the result of an operation approaches underflow. Unlike most IEEE floating-point numbers, they do not include the implied 1. Instead, they have an exponent of 0, keep the decimal immediately to the left of the stored value, and allow the first 1 to fall anywhere in the stored value. Denormals are particularly useful for code such as: if `(x!= y) z = 1/(x - y)`. This code should never cause an exception, but without denormal support it can easily cause a divide by 0 when x and y are small enough and close enough that the format cannot represent the difference. Floating-point hardware within a microprocessor typically implements denormals with an exception that then computes the value via software. However, in an FPGA-based implementation, to support full IEEE floating point we must generally add denormal support into the hardware itself. Thus, for denormal numbers, the value is extracted as in equation 31.2.

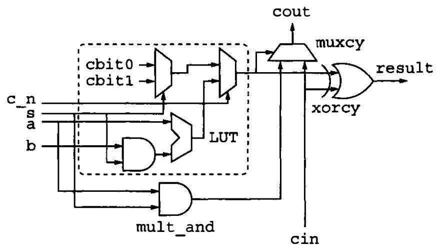
$$(-1)^S \times 2^{exp-bias} \times 0.mantissa \tag{31.2}$$

31.1.1 General Implementation Considerations

To produce the smallest, fastest circuits, it is necessary to efficiently use the structure of the FPGA. This comes up in two areas: (1) It is necessary to fully utilize every lookup table (LUT), whenever possible and (2) it is advantageous to provide an optimized layout for each unit. The floating-point units presented here have been written using JHDL—a structural design tool that provides a clean mechanism for mapping and relationally placing logic.

The units were optimized by identifying opportunities to combine logic into the LUT architecture of the FPGA. This can be challenging, particularly for operations that use the carry-chain logic. However, the special values in the IEEE format make it vital that carry-chain and other logic be mixed. For example, there are many instances where the output of the exponent logic is either the result of an arithmetic operation or a constant. For FPGA architectures, such as the Xilinx Virtex family, it is possible to map the arithmetic operation and the constant generation into the same LUT (along with its associated carry logic).

Take, for example, the `passAddOrConstant` circuit. It has four possible outputs: $a+b$, a, $c0$, or $c1$, where a and b are variables and $c0$ and $c1$ are constants. The inputs to the circuit are a, b, s, and c_n. When $c\_n = 0$, the output is one of the two constants, which is selected by the s input. Otherwise, the result is $a+b$ when $s = 1$ and a when $s = 0$. The logic used for each bit is shown in Figure 31.2(a). The circuit is only possible because of the `mult_and` added in the Virtex family of FPGAs. `mult_and` was originally intended for use in multipliers built from logic, but it enables many other useful optimizations. The same basic logic can also create a `passSubOrConstant`, and if the AND gate before the arithmetic operation is left off, the circuit is simply an `addOrConstant` or `subOrConstant`. These circuits are used to reduce the amount of logic and the logic delay required to compute the exponents. The JHDL code used to generate each bit of this circuit is shown in Figure 31.2(b). Note that all the logic is



(a)

```
// Produce the constant bit. The Xilinx tools believe
// that gnd and vcc are inputs to the LUT, so we can't
// use them. Instead, use c_n, which will be 0
// when the constant is selected.
Wire cbit0 = ((c0 >> i) & 1) == 1 ? not(c_n) : c_n;
Wire cbit1 = ((c1 >> i) & 1) == 1 ? not(c_n) : c_n;
Wire constant_result = mux(cbit0,cbit1,s);

// Generate the sum bit.
Wire sum = mux(constant_result,
          xor(a.gw(i),and(s,b.gw(i))),c_n);

// Map all the above logic in a single LUT
Cell x = map(c_n,s,a.gw(i),b.gw(i),s_partial);
place(x,0,virtex ? maxrow - i/2 : i/2);

Wire mult_and_out = wire(1,"mult_and_out" +i);
x = new mult_and(this,c_n,a.gw(i),mult_and_out);
place(x,0,virtex ? maxrow - i/2 : i/2);

x = new muxcy(this,mult_and_out,cin,s_partial,cout);
place(x,0,i/2);

x = new xorcy(this,s_partial, cin, output.gw(i));
place(x,0,i/2);
```

(b)

FIGURE 31.2 ■ Logic (a) and JHDL code (b) for the i^{th} bit of the `passAddOrConstant`.

first mapped into LUTs using the `map` function, then relationally placed, using the `place` function. The same `place` function is used to relationally place the lower-level blocks at each level of hierarchy. The overall unit is placed into a rectangular area so that it can be easily tiled in a design (see the descriptions of the adder and multiplier in Sections 31.1.2 and 31.1.3).

In addition to concerns about efficiently using the LUT and providing good placement directives, there are concerns about where to pipeline the units. The major concern that largely determined the pipelining of the units presented here involves the carry-chain logic. In the Virtex family, the times to initalize and finalize the carry chain are large relative to the per-bit propagation time on the carry chain. Thus, it is necessary to avoid having cascaded carry chains in the same stage. In most cases, this constraint determines the stage mapping.

31.1.2 Adder Implementation

The most noticeable difference between integer operations and floating-point operations is in the implementation of the adder. A 64-bit registered integer adder requires 64 4-LUTs, 64 flip-flops, and the associated carry-chain logic. It can be packed into 32 *slices* in a Xilinx Virtex-4[1] or similar family. In stark contrast, a 64-bit floating-point adder requires hundreds of 4-LUTs, hundreds of flip-flops, and nearly 700 *slices*. The core of the differences can be seen in Figure 31.3(a).

The fundamental problem is that two numbers of the form

$$(-1)^{S0} \times 2^{exp0-bias} \times 1.mantissa0 \tag{31.3}$$

and

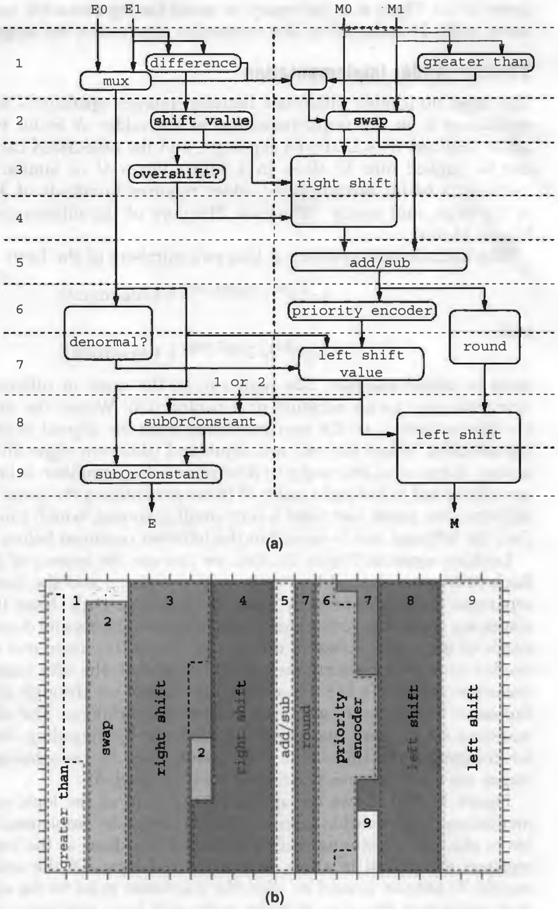
$$(-1)^{S1} \times 2^{exp1-bias} \times 1.mantissa1 \tag{31.4}$$

must be added together. The signs can be the same or different, so the actual operation may be an addition or a subtraction. Worse, the exponents can differ (dramatically), so the two mantissas must be aligned before the operation can proceed. When the two are combined (different signs and different exponents), it becomes necessary to determine which number is larger so that they are subtracted in the right order. If the exponents are the same but the signs are different, the result can yield a very small mantissa, which must be normalized (i.e., the leftmost one is moved to the leftmost position) before it can be stored.

Looking again at Figure 31.3(a), we can see the impact of the extra format. Each horizontal dashed line represents a register, and the vertical dashed line separates the exponent path from the mantissa path. Note that the first two stages are spent inspecting and preparing the numbers and determining whether either of the inputs is one of the special values. The third and fourth stages are needed to align the mantissas, and it is not until the fifth stage that the actual *operation* occurs. In the exponent path, stages six through nine clean up the exponent to handle a variety of exception conditions. The sixth and seventh mantissa stages have two parallel paths: one for rounding the result and one for computing the shift value if the result must be renormalized. The last two stages are used to renormalize the result (if needed).

Figure 31.3(b) shows the approximate layout of the logic used in an implementation of the floating-point adder. For the adder implementation, it is possible to place all pipelining registers in the same slices as the logic, though some registers are placed in slices with unrelated logic. Of the total area, approximately 39 percent is used to align the mantissas prior to the actual add or subtract operation; this area includes right-shift logic and swap logic. These operations would be required for any floating-point format; however, the left-shift on the backend is only required because of the existence of the implicit 1 in the format. This case arises during a loss of precision when two numbers with

[1] A slice is two 4-LUTs, two flip-flops, and the associated carry-chain logic in this generation.



(a)

(b)

FIGURE 31.3 ■ Adder block (a) and adder layout (b) diagrams.

identical, or very close, exponents are subtracted and require normalization. The normalization logic, including a priority encoder to locate the first 1, uses another 39 percent of the logic. For comparison, the actual add and round logic consumes only 9 percent of the area.

31.1.3 Multiplier Implementation

The relationship between a floating-point multiplication and a fixed-point multiplication is a little more unusual. A fixed-point multiplier grows with the square of the width of the input. At the core of a floating-point multiplier is a fixed-point multiplier that multiplies the mantissas. Since the mantissa is significantly narrower than the floating-point number, a 64-bit fixed-point multiplier actually has a much larger core operation than a 64-bit floating-point multiplier because the floating-point multiplier only has to multiply two 53-bit mantissas. It does, however, have a lot of other work to do that more than makes up for the difference.

Floating-point multiplication starts with two numbers:

$$(-1)^{S0} \times 2^{exp0-bias} \times 1.mantissa0 \tag{31.5}$$

and

$$(-1)^{S1} \times 2^{exp1-bias} \times 1.mantissa1 \tag{31.6}$$

that produce the result:

$$(-1)^{(S0 \oplus S1)} \times 2^{(exp0-bias)+(exp1-bias)} \times 1.mantissa0 \times 1.mantissa1 \tag{31.7}$$

Conceptually, the dataflow shown in Figure 31.4(a) is quite simple. The first three stages unpack the IEEE format looking for special cases and preparing a possible denormal mantissa for the multiplier core. Stages F4 through F6 operate concurrently with the multiplier core and compute the resulting exponent and determine whether the result is denormal. The four backend stages provide shifting for creating denormal numbers, rounding, and normalization, which includes adjusting the exponent when required.

Figure 31.4(b) gives the approximate layout of the logic for the front- and backends of the multiplier. The multiplier core (not shown in the figure) uses nine 17 × 17 multiplier blocks plus additional logic to sum the partial products to create a 53 × 53 multiplier core. The logic used in the core is about 40 percent of the total multiplier logic. Unlike the adder, it is not possible to place all of the required pipelining registers in slices used by the logic. The black regions in Figure 31.4(b) are either unused or used by pipelining registers.

The logic required to support the IEEE format is nontrivial. Support for denormals consumes 40 percent of the multiplier area and includes logic to gather information about the mantissa, swap the mantissa, and shift the mantissa. Thus, supporting denormals requires approximately the same amount of logic resources as the multiplier core. An additional 7 percent of the area is used for rounding and normalization to put the number back into the IEEE format.

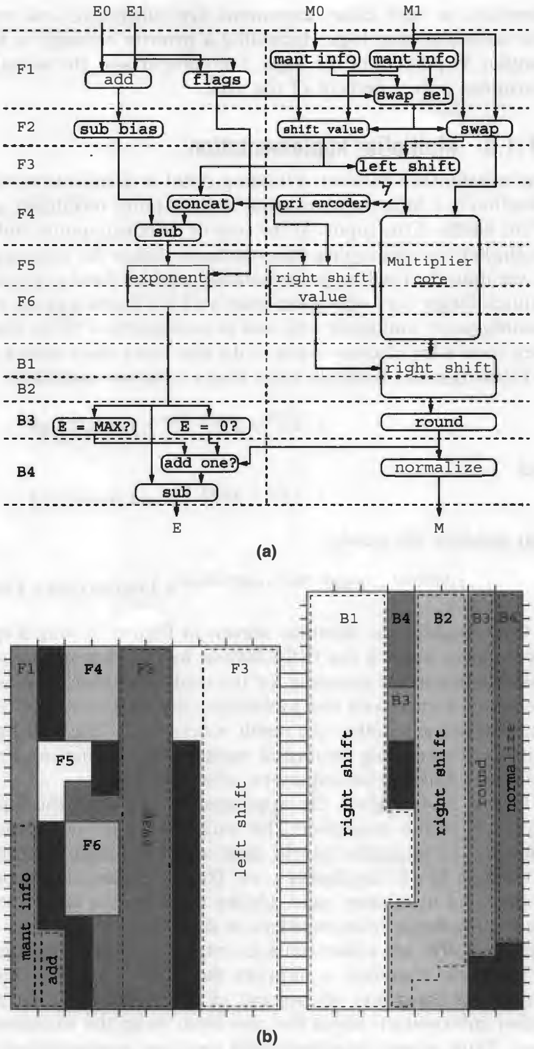


FIGURE 31.4 ■ Multiplier block (a) and multiplier layout (b) diagrams.

31.2 FLOATING-POINT APPLICATION CASE STUDIES

Floating-point applications that are appropriate to map to FPGAs differ dramatically from integer applications that are typically mapped to FPGAs. The differences can be understood by realizing that a single floating-point operation can easily consist of 30 integer operations; thus, where a 2005-era FPGA can easily implement 1000 integer operations, it is more likely that it can only implement 32 double-precision floating-point operations. Furthermore, floating-point operations are much higher latency than corresponding integer operations, which significantly affects designs.

This section considers three kernel operations implemented with double-precision floating point to demonstrate three important considerations when using floating point operations on FPGAs. The first operation is matrix multiply, which demonstrates the FPGA's ability to exploit high degrees of parallelism and to programmably manage local storage to significantly reduce the amount of external RAM bandwidth needed. The second kernel is a vector dot product, which highlights the ability of the FPGA to provide large amounts of RAM bandwidth; plus it highlights limitations introduced by the high latency of the floating-point units. The third kernel is the fast Fourier transform (FFT), which can find similar advantages in mitigating the need for memory bandwidth as the matrix multiply, but has similar limitations from the latency of the floating-point units to the dot product.

31.2.1 Matrix Multiply

The standard matrix multiply (the `DGEMM` BLAS routine) is defined as:

$$\mathbf{C}_{ij} += \sum_{k=0}^{N-1} \mathbf{A}_{ik}\mathbf{B}_{kj} \tag{31.8}$$

The operation multiplies two matrices and adds it to a third (in place). Conceptually, this means performing the dot product of a single row of A with a single column of B and adding the result to a single point of C. Each dot product is completely independent, which means there are N^2 independent dot products. In practice, neither microprocessors nor FPGAs implement it this way because of the nature of modern memory hierarchies. In all modern systems (including FPGAs), main memory is "far away" and there is one or more caches significantly "closer."

The primary performance characteristic of matrix multiply is that it does $O(N^3)$ operations on $O(N^2)$ data. Thus, for every data item loaded from memory, it should be hypothetically possible to do $O(N)$ operations. Performing matrix multiplication as a series of independent dot products would throw away this advantage; thus, all matrix multiply implementations attempt to exploit some form of locality within the cache structure.

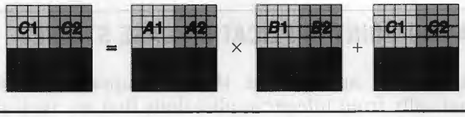


FIGURE 31.5 ■ Block decomposition of a matrix multiply.

FPGA implementation

To understand an FPGA implementation of matrix multiply, it helps to first understand how it is done on a microprocessor. To exploit (or rather compensate for) the nature of modern memory hierarchies, the typical approach to matrix multiplication on a microprocessor breaks the matrices into smaller $S \times S$ blocks [16]. A given block from each matrix is loaded into the processor, a matrix multiply is performed on the block, and partial results are stored. An example for an 8×8 matrix multiply is shown in Figure 31.5. Each matrix is broken into four regions that are 4×4. A row of these blocks is then multiplied by a column of these blocks to create a 4×4 block of the result; thus, $C1 = A1 * B1 + A2 * B3 + C1$. In the process, the partial result (a 4×4 block) is updated two times (although typically in local storage or cache).

The same approach can be used on FPGAs. After all, FPGAs and microprocessors are similar in that they have a small amount of local memory with high bandwidth and a large amount of external, slower memory. FPGAs differ, however, in that they have a drastically large number of floating-point units that should be kept fully utilized. Whereas microprocessors must supply inputs to two functional units per cycle, FPGAs must supply inputs to 32 functional units (in a 2005 FPGA).

A matrix multiply can be decomposed into a series of multiply–accumulate (MACC) operations that multiply the individual elements of a row with elements of a column and accumulate the result into one element of the final matrix. The MACC unit has a multiplier, an adder, and a feedback path. In an FPGA, 16 MACC units are operating concurrently. Unfortunately, the latency of the adder is very high (10 cycles). This means that we must keep at least 10 concurrent operations ($row \times column$ operations) in progress at all times to hide the latency of the adder. In a perfect world, each unit could work on a block of the matrix, with the concurrent operations happening on the independent row–column dot product in that block. Unfortunately, this would require far more internal memory than is available in typical FPGAs.

To exploit the parallelism available in FPGAs without exhausting the limited internal memory, we can further decompose the view of the problem. A simple way to view one block-level matrix multiplication is as a collection of S matrix–vector multiplications. As such, significantly more parallelism is obvious. Figure 31.6 shows an FPGA-based implementation that first decomposes the problem into blocks and then distributes portions of the work to multiply the two blocks as matrix–vector multiplications.

]

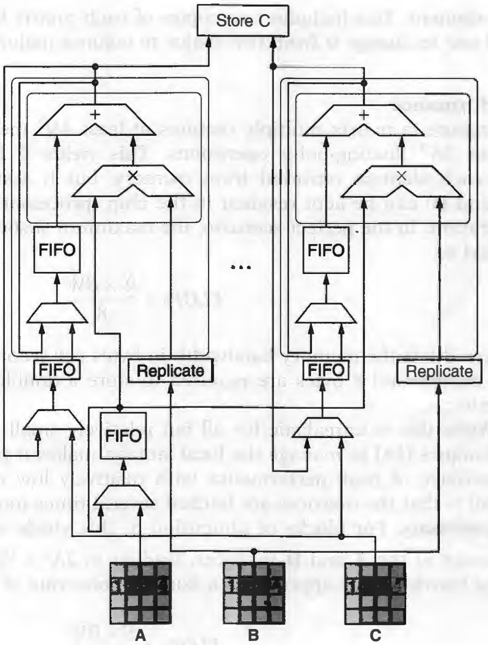


FIGURE 31.6 ■ Matrix multiply implementation.

To perform the full matrix multiplication, each matrix is decomposed into $S \times S$ blocks. In Figure 31.6, S is 4, but in practice, S is typically set large enough to cover the adder's latency (currently 10 cycles). Blocks of **B** are broken into m columns, where m is the number of MACC units (m is assumed to be 4 in the figure); thus, independent columns of a block of **B** go to each MACC unit. All the blocks of **A** are *broadcast* to all MACC units. Thus, in Figure 31.6, one column of block **B** is multplied by all four rows from the **A** block. This requires that four copies (in the general case S copies) of the **B** block be made by the replicate unit. This creates the concurrency needed to cover the latency of the adder.

Matrix **C** is managed similarly. A block of **C** is loaded and distributed in the same *order* as the block of **B**, but there is no need to replicate it. In addition, taking the example from Figure 31.5, two **A** blocks and two **B** blocks are needed for each **C** block. Thus, $A1$, $B1$, and $C1$ are loaded and used to create an intermediate product $C1-2$ that is used as the C block when $A2$ and $B2$ are multiplied. Overall, this requires no more than $6S^2$ elements of storage at 8 bytes per element. This includes two copies of each matrix block—one to operate on and one to change it from row-major to column-major order.

Performance

By nature, a matrix multiply requires at least $4N^2$ memory accesses[2] and performs $2N^3$ floating-point operations. This yields $\frac{N}{2}$ floating-point operations for each element retrieved from memory, but it assumes that two matrices (**A** and **B**) can be kept resident in the chip (processor or FPGA) for the entire operation. In the perfect scenario, the maximum sustainable floating-point rate would be

$$FLOPs = \frac{\frac{N}{2} \times BW}{8} \tag{31.9}$$

where BW is the memory bandwidth in bytes per second, N is the dimension of the matrix, and 8 bytes are required to store a double-precision floating-point number.

While this is unrealistic for all but relatively small matrices, using blocking techniques [16] to manage the local storage makes it possible to sustain a high percentage of peak performance with relatively low memory bandwidth. The result is that the matrices are fetched several times more than would otherwise be necessary. For blocks of dimension S, this yields a factor of $\frac{N}{S}$ increase in accesses to the **A** and **B** matrices, leading to $2N^2 + \frac{2N^3}{S}$ memory accesses. For large matrices, this approaches a floating-point rate of

$$FLOPs = \frac{S \times BW}{8} \tag{31.10}$$

This is shown in Figure 31.7(a) as MFLOP/s versus MB/s on a log–log graph. Delineations that map memory bandwidth needs to the generation of FPGAs are provided for clarity, based on earlier work [14, 15].

A slightly different perspective is presented in Figure 31.7(b) where the total amount of on-chip memory needed to sustain peak performance is graphed. What is notable about these graphs is the relatively small amount of memory and relatively small amount of memory bandwidth needed to sustain peak performance on FPGAs. This stands in stark contrast to modern microprocessors (2005 era) that only sustain 85 to 90 percent of peak performance on a matrix multiply using several times as much on-chip memory and off-chip memory bandwidth. This is a product of the ability of the FPGA to directly manage local storage and to separate data prefetching from computation.

We can also compare performance over time using data from 2004 [see 14,15]. Table 31.2 shows parts used for comparison. The performance of FPGAs gained rapidly on microprocessors during this era, as shown in Figure 31.8.

[2] This assumes square matrices and includes retrieving three matrices and storing one matrix.

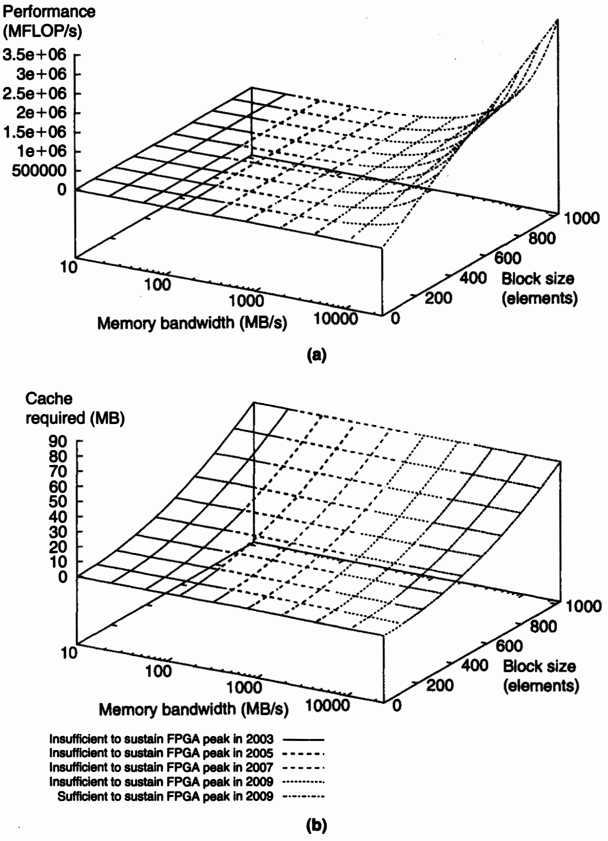


FIGURE 31.7 ■ Maximum achievable performance versus memory bandwidth and block size (a); on-chip memory needed versus memory bandwidth and block size (b).

31.2.2 Dot Product

The standard vector dot product (the DDOT BLAS routine) is the sum of the pairwise products of two vectors, or

$$p = \sum_{i=0}^{N-1} \mathbf{x}_i \mathbf{y}_i \tag{31.11}$$

TABLE 31.2 ■ Parts used for performance comparison

Year	FPGA	CPU
1997	XC4085XLA-09	Pentium 266 MHz
1999	Virtex 1000-5	
2000	Virtex-E 3200-7	Athlon 1.2 GHz
2001	Virtex-II 6000-5	
2003	Virtex-II Pro 100-6	Pentium-4 3.2 GHz

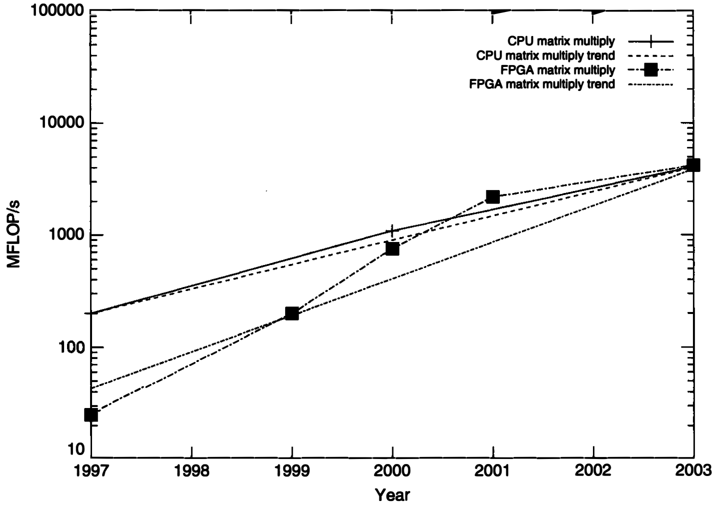


FIGURE 31.8 ■ Matrix multiply performance of FPGAs and microprocessors from 1997–2003.

which requires 2*N* memory accesses to perform 2*N* floating-point operations. This means that a double-precision floating-point number (8 bytes) must be fetched from memory for every floating-point operation that will be done. Modern processors are not built with this type of balance between memory bandwidth and floating-point capability. A processor capable of providing five GFLOP/s may only have 6.4 GB/s of memory bandwidth. Streaming problems (like this one) provide FPGAs an opportunity to excel—processors have a fixed-memory bandwidth that is configured based on a balance between the requirements for various markets and the cost of providing that bandwidth. In contrast, each board containing an FPGA can decide how many FPGA pins are used for memory bandwidth, including dedicating almost all available user pins to memory connections.

FPGA implementation

Although the potential for increased memory bandwidth on an FPGA gives it a distinct advantage, it also faces significant challenges imposed by the large number of functional units and the high latency of the units. Like many BLAS routines, DDOT is based on multiply–accumulate operations; however, it differs from many BLAS routines in that it exposes a relatively limited amount of parallelism. Where a DGEMM operation computes N^2 independent results and a DGEMV operation computes N independent results, a DDOT operation produces a single number as the final result. This means than any partial products must be reduced through a long, slow pipeline. The nature of the problem is best realized through a comparison to microprocessors.

Current microprocessors typically have a floating-point pipeline depth of four to six cycles for the functional unit running at 2 GHz or more. Obviously, we would not want every addition to depend on the previous addition, so the microprocessor can easily keep six running sums in progress and then reduce those sums to one result. This leads to several pipeline stalls in the final reduction, but the total time is a small number of nanoseconds. In contrast, FPGAs differ in three dramatic ways:

- The adder pipeline is deeper.
- Multiple MACC units are required to fully utilize high bandwidth memory.
- The clock rate is lower.

A modern FPGA would have tens of functional units with a pipeline depth of 10-cycles running at approximately 300 MHz. Assuming 16 adders with a pipeline depth of 10 cycles means that there must be 160 concurrent summations. This is impossible for short vectors and challenging even for longer vectors. Furthermore, the process of reducing these partial sums to a single result is slow and cumbersome.

To achieve reasonable performance, additional control logic is required inside and outside the multiply–add and MACC units. First, a multiplier bypass multiplexer (labeled MB) is required in the multiply–add (Figure 31.9(b)) to reuse the adder for portions of the final summation. Second, the adder has a 10-cycle latency; thus, the MACC must perform 10 concurrent operations to keep the adder pipeline filled. This requires a second feedback path (with associated control) through the FP multiplexer in the MACC (Figure 31.9(c)) to sum the 10 results. The added logic is shown with dashed lines in Figure 31.9(b) and (c).

Performance

If we work from the memory bandwidth as the typical limiting factor, the maximum sustainable floating-point rate is

$$FLOPs = \frac{BW}{8} \tag{31.12}$$

where BW is the memory bandwidth in bytes per second and 8 bytes are required to store a floating-point number. This is graphed in Figure 31.10(a)

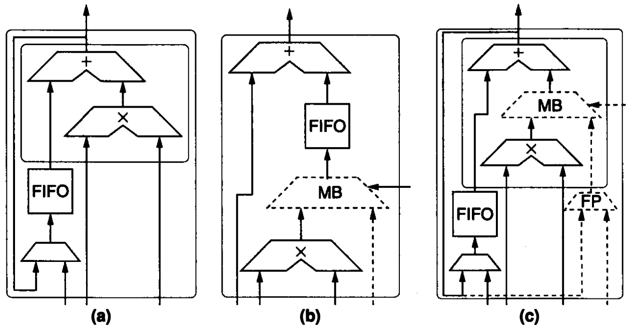


FIGURE 31.9 ■ A standard multiply–accumulate (a); a modified multiply–add for the dot product (b); a modified multiply–accumulate for the dot product (c).

on a log–log graph. Like Figure 31.8, Figure 31.10(b) compares performance projections for both FPGAs and microprocessors [14, 15]. In this case, however, the FPGA shows a much more dramatic advantage over a microprocessor. This is because large FPGAs provide sufficient I/O resources to obtain much higher memory bandwidths than commodity microprocessors offer. Since this is a memory bandwidth-limited problem, the platform with the most memory bandwidth wins.

The other notable feature of Figure 31.10(b) is that it is somewhat more crowded than the matrix–multiply comparison. This is because FPGAs face a second challenge in implementing the dot product operation: the latency of the floating-point unit. Thus, the size of the vector has a much greater impact on sustained performance on the FPGA than the microprocessor. The top FPGA line represents a scenario whereby the FPGA achieves 90 percent of its peak performance, but this requires a nearly 6000-element vector.[3] The second FPGA line shows the FPGA achieving 50 percent of peak performance by using an 800-element vector. Despite this hefty penalty, the FPGA still has a remarkable advantage (4× in 2003) over the microprocessor.

31.2.3 Fast Fourier Transform

The fast Fourier transform (FFT) is a reduced-complexity implementation of the discrete Fourier transform (DFT), which takes as input N complex numbers and returns as output N complex numbers where each of the outputs is determined by the following equation:

[3] Earlier work by Underwood and Hemmert [15] specified a 7500-element vector, but the floating-point unit latency has been optimized since then.

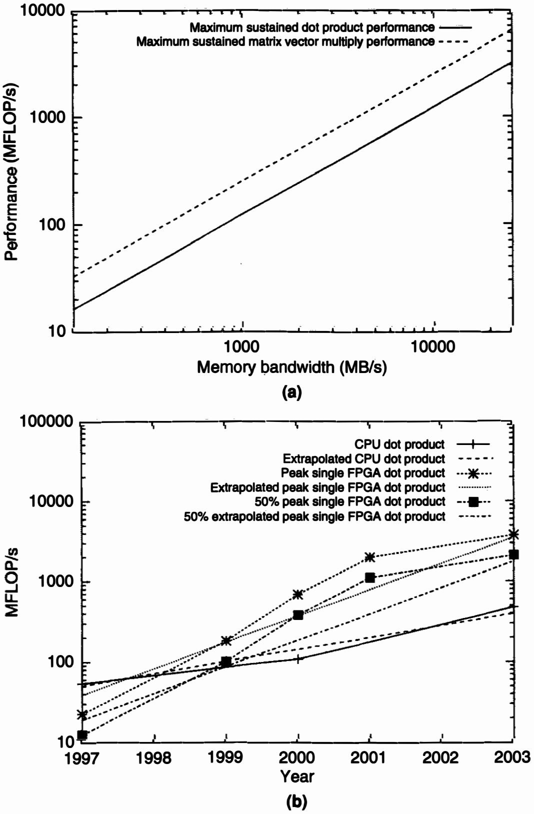


FIGURE 31.10 ■ Maximum achievable performance versus memory bandwidth (a) and dot product performance on FPGAs and microprocessors from 1997–2003 (b).

$$Y[j] = \sum_{k=0}^{N-1} X[k]W_N^{jk} \tag{31.13}$$

where $W_N^{jk} = e^{\frac{-i2\pi jk}{N}}$.

The FFT exploits symmetries in the DFT and is implemented in stages, where each stage combines r items to create r outputs. The value r is known as the *radix*. For the implementation discussed here, $r = 2$ (radix-2). For the radix-2

FFT, each stage operates pairwise on the data, although there are different formulations of the algorithm that determine how the data are combined. These operations are commonly referred to as *butterflies* and in the formulation used in this example, each pairwise operation is identical and consists of one complex multiply and two complex adds. This is shown graphically in Figure 31.11(a).

Even after selecting the formulation that gives the structure of the butterfly, there is some flexibility in the structure of the stages. The basic stage structures are shown in Figure 31.12. Both structures require data reordering, either on

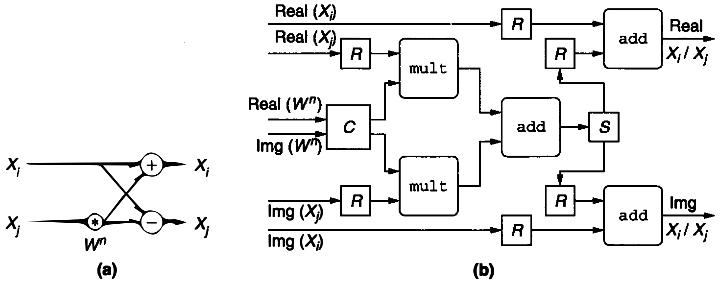


FIGURE 31.11 ■ Basic butterfly operation (a) and basic butterfly datapath (b). The component *S* is a switch that directs inputs to alternate outputs. The components marked as *R* replicate the input once and *C* is a crossover to facilitate the complex multiply.

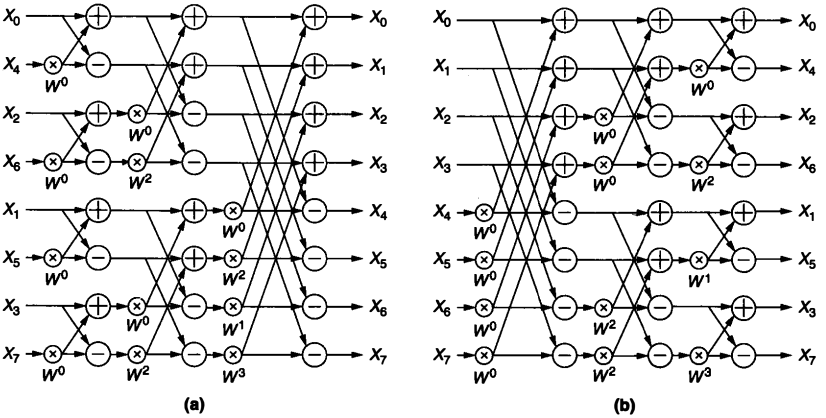


FIGURE 31.12 ■ Variations of the 8-point, radix-2 FFTs with reordered inputs (a) and reordered outputs (b).

the frontend or backend, and produce the identical set of computations (though in different orders). This example uses the ordering shown in Figure 31.12(b), because this structure provides an increasing number of independent datasets as the computation progresses. This approach is easier for implementations that use units in parallel to process data within a single stage since all interunit communication can reside at the front of the pipeline.

FPGA implementation

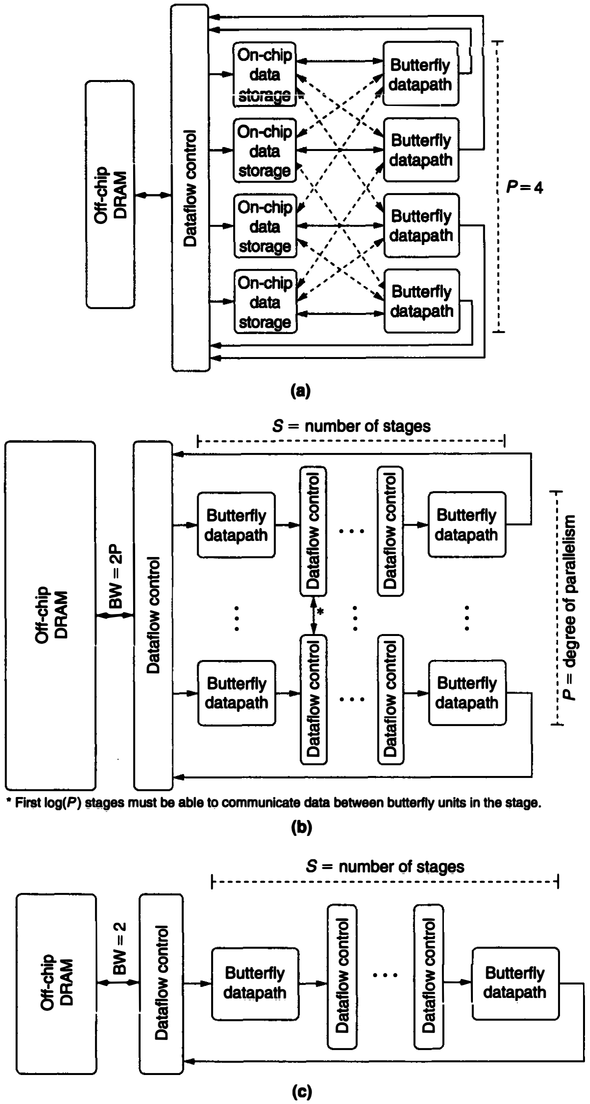
The butterfly computation requires four multiplications and six additions to implement one complex multiply and two complex adds. The hardware presented here uses two double-precision multiplies and three double-precision adds (see Figure 31.11(b)). Each floating-point unit is used twice for each set of inputs, which results in an average throughput of one data item per clock cycle. Although it is possible to design a datapath that accepts two data items per clock cycle, this design was chosen because it matches the available bandwidth of internal RAM blocks in the target architecture and because it provides the greatest flexibility when scaling the parallelism of the final implementation.

Parallelism in the FFT computation can be exploited in two ways: (1) pipelined units, or parallelism in the stages (S), and (2) parallel units, or parallelism (P) within a stage. Three architectures, which exploit the two types of parallelism to differing degrees, are explored.

Parallel architecture The *parallel implementation* exploits only parallelism within a stage (P). This is shown in Figure 31.13(a). In this implementation, data are read from external memory, processed iteratively, and written back to external memory. Each of the butterfly units operates on a subset of the data and is able to work independently of the other units for a large part of the computation (the datasets are completely independent after $log_2(P)$ stages).

The advantages of this architecture are that the utilization of the units is high because the pipeline depth is short. The parallel version can also take advantage of higher-memory bandwidths. The disadvantages of this architecture as implemented are that it requires a large amount of internal memory and it requires a parallelism that is a power of 2. This second restriction is important because it can limit the number of butterfly units that can be used. For example, if six butterfly units fit in an FPGA, the parallel architecture is still only able to use four.

Pipelined architecture At the other extreme, one butterfly unit can be dedicated to each of the stages of the FFT in a pipelined fashion, as illustrated in Figure 31.13(c). Data is read from memory and passed through a series of butterfly units before being written back to memory. Data delays and permutations are needed between each of the stages and between the pipelined FFT unit and DRAM memory. When the number of stages, S, that can be implemented in the FPGA is less than the number of stages needed by the FFT ($log_2(N)$), then $\frac{log_2(N)}{S}$ passes to memory are needed, with the final pass using a subset, R, of the stages. For each pass to memory, data must be read and written in a particular permutation to optimize the delay and storage requirements in the pipeline.



* First log(P) stages must be able to communicate data between butterfly units in the stage.

FIGURE 31.13 ■ Three architectures: (a) parallel, (b) parallel–pipelined, and (c) pipelined for exploiting parallelism in the FFT—from using all parallelism within a single stage to using all parallelism in the stages.

The pipelined architecture works well when streaming a large number of small FFTs. This is because the architecture gets good performance with minimal memory bandwidth requirements. Another benefit of this architecture is that it can take advantage of parallelism at a finer granularity than the parallel version (i.e., it can use a nonpower of 2 number of processors). However, there are some major disadvantages to this architecture. First, for single FFTs, the unit utilization is low because of the depth of the overall pipeline. Second, it is unable to take advantage of higher-memory bandwidth. Last, the buffer space required between stages for data reordering grow as 2^S, where S is the number of stages in the circuit. For a large number of stages, the memory required for buffering can easily exceed available on-chip memory.

Parallel–pipelined architecture Figure 31.13(b) is a cross between the two previous architectures. Data moves from external memory, through a set of P parallel pipelines—each with S stages—and back to external memory. The first $log_2(P)$ stages must have additional data exchange circuits (for the first pass through the pipeline) because these stages have data dependencies between the pipelines. This approach leverages the ability of the pipelined architecture to reduce bandwidth demands and the ability of the parallel architecture to tolerate shorter input vectors (as well as a wider variety of vector lengths) than the pure pipelined approach. In contrast, the parallel–pipelined hybrid has a higher bandwidth demand than the purely pipelined approach and less tolerance of short vectors than the parallel approach.

Performance

In evaluating the performance of the FFT, the floating-point operation count that is typically used is $5Nlog_2(N)$; there are $log_2(N)$ stages that each contain $5N$ computations (four multiplies and six additions for each pair of data). To determine performance, it is necessary to know how long it will take the FPGA to compute the FFT. For the parallel version, the number of cycles required to complete the FFT is given by the following equation:

$$T = \frac{32N}{BW} + BL + (\frac{N}{P} + BL)(log_2(N) - 2) \qquad (31.14)$$

The first term of equation 31.14 is the time to read and then write N items based on the memory bandwidth, BW, in bytes per cycle. The usable bandwidth is limited to the number of units, P. The second term is the latency of passing through the butterfly units during the read from memory. The third term is the time to perform the iterations—using P butterfly units of latency BL for $log_2(N) - 2$ iterations, assuming that the first and last iterations are performed as part of reading and writing the data.

The pipelined and parallel–pipelined architectures share the same equation for determining the number of clock cycles required to complete the operation. The only difference is that the pipelined architecture is limited to a bandwidth (*BW*) of 2. The number of cycles to compute the FFT for these architectures is

$$T = P(S) \times \left\lfloor \frac{log_2(N)}{S} \right\rfloor + P(R) \tag{31.15}$$

$$P(J) = BL \times J + I(J) + \frac{2N}{BW} + (B-1) \times 2^J \tag{31.16}$$

$$I(K) = \sum_{i=0}^{K-1} B \times 2^i \approx B \times 2^K \tag{31.17}$$

$$R = log_2(N) \bmod S \tag{31.18}$$

Each pass, *P*(*J*), through *J* butterfly stages (each having a latency of *BL*) requires the time shown in equation 31.16.

Data dependencies between the stages introduce a delay that doubles at each stage, and create a total interstage delay given by *I*(*K*). Using standard DRAM memories introduces a penalty associated with the burst length (*B*) required to maintain full memory bandwidth to both the interstage delay and a backend reordering time. The time to retrieve the data from memory and write them back is defined by $\frac{2N}{BW}$. The final term represents the final pass through a subset of the stages, *R*, with the corresponding delays.

The preceding equations point to the fact that the *best* implementation for the FFT depends on many factors: memory bandwidth, size of the FFT, and size of the FPGA. The performance (in FLOPs per cycle) for a single FFT of the different FPGA architectures on a Xilinx Virtex-II Pro (a late 2005 part) are shown in Figure 31.14(a). For single short vectors, the parallel architecture provides the best performance. This is because of the high utilization of the floating-point units. For longer FFTs, all three units provide good performance, though the pipelined version requires less external memory bandwidth. Figure 31.14(b) shows that the FPGA implementations (running at 160 MHz) compare favorably to microprocessors for large FFTs.

31.3 SUMMARY

Implementing floating-point arithmetic on FPGAs requires significant effort. Supporting the IEEE-754 standard poses particularly unique challenges, but much of the effort is expended in coping with the interaction between exponent logic and mantissa logic. Great care is required to minimize the latency through the unit without significantly decreasing clock rate by having two dependent carry chains in a single pipeline stage. Even with effort, floating-point operations are significantly bigger and have significantly deeper pipelines than their fixed-point counterparts. This adds additional challenges to the design of applications.

Although FPGAs can now deliver impressive performance on double-precision floating-point operations, it requires a very different mind-set from working with fixed-point arithmetic. Increased operation latency leads to a need to find more parallelism to exploit in paths with the cyclic data dependencies typical of

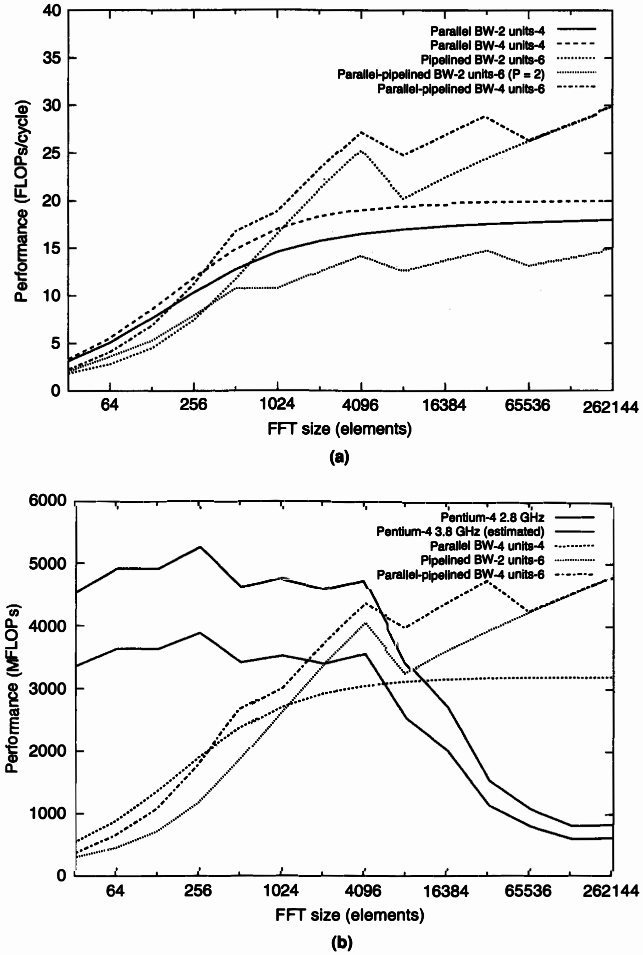


FIGURE 31.14 ■ A comparison of performance for different FFT architectures in FLOPs per cycle (a) and a comparison of FFT implementation on FPGAs and CPUs (b).

iterative solutions. Simultaneously, the increased size of a single operation reduces the portion of a given dataflow graph that can be implemented directly and pushes a designer toward more iterative solutions. The dot product is an excellent example because it is forced to reuse adders to compute a summation that would typically be done as a tree of adders in a fixed-point solution. The result is that only longer vectors make sense.[4]

Even simple feedforward paths incur a penalty from the high latency of the floating-point units. The FFT provides an example whereby the latency of a single butterfly path can approach the length of a short vector. Thus, if the FFT implementation in an FPGA is not used for a long FFT or a series of short FFTs, it cannot offer competitive performance.

There are, however, floating-point kernels that offer abundant parallelism for the FPGAs to exploit. Matrix–multiply (DGEMM), for example, is an N^3 operation with minimal data dependencies. Similar things can be said about LU solvers, which form the basis of the traditional Linpack benchmark [4]. Three-dimensional FFTs are another example in which hundreds of one-dimensional FFTs can be carried out simultaneously.

References

[1] P. Belanovic, M. Leeser. A library of parameterized floating-point modules and their use. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2002.

[2] M. deLorimier, A. DeHon. Floating point sparse matrix-vector multiply for FPGAs. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2005.

[3] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, D. Poirier. A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2002.

[4] J. J. Dongarra. The linpack benchmark: An explanation. *First International Conference on Supercomputing*, June 1987.

[5] Y. Dou, S. Vassiliadis, G. Kuzmanov, G. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2005.

[6] B. Fagin, C. Renard. Field-programmable gate arrays and floating point arithmetic. *IEEE Transactions on VLSI* 2(3), 1994.

[7] A. A. Gaar, W. Luk, P. Y. Cheung, N. Shirazi, J. Hwang. Automating customisation of floating-point designs. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2002.

[8] G. Govindu, S. Choi, V. K. Prasanna, V. Daga, S. Gangadharpalli, V. Sridhar. A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs. *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW)*, April 2004.

[9] G. Govindu, L. Zhuo, S. Choi, P. Gundala, V. K. Prasanna. Area and power performance analysis of a floating-point based application on FPGAs. *Proceedings of the Seventh Annual Workshop on High-Performance Embedded Computing*, September 2003.

[10] K. S. Hemmert, K. D. Underwood. An analysis of the double-precision floating-point FFT on FPGAs. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005.

[4] A long series of short vectors can also be made to work using an appropriate architecture.

[11] IEEE Standards Board. *IEEE Standard for Binary Floating-Point Arithmetic*. Technical Report ANSI/IEEE Std. 754-1985, The Institute of Electrical and Electronics Engineers, 1985.

[12] L. Louca, T. A. Cook, W. H. Johnson. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[13] N. Shirazi, A. Walters, P. Athanas. Quantitative analysis of floating-point arithmetic on FPGA based custom computing machines. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[14] K. D. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2004.

[15] K. D. Underwood, K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[16] R. C. Whaley, A. Petitet, J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27(1–2), 2001.

[17] L. Zhuo, V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. *18th International Parallel and Distributed Processing Symposium*, April 2004.

[18] L. Zhuo, V. K. Prasanna. Sparse matrix–vector multiplication on FPGAs. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2005.

CHAPTER 32

FINITE DIFFERENCE TIME DOMAIN: A CASE STUDY USING FPGAS

Wang Chen, Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University

This chapter presents a reconfigurable hardware accelerator that implements the FDTD method. We first present background, including applications of the FDTD method. We then provide analysis and design details of the FPGA accelerator for FDTD.

32.1 THE FDTD METHOD

Modeling electromagnetic behavior has become a requirement for key technologies such as cellular phones, mobile computing, lasers, and photonic circuits. The finite-difference time-domain (FDTD) method, which provides a direct, time domain solution to Maxwell's equations in differential form with relatively good accuracy and flexibility, has become a powerful method for solving a wide variety of electromagnetic problems [1–3]. The main drawback to FDTD is its high computational complexity.

32.1.1 Background

The discovery of Maxwell's equations was one of the outstanding achievements of nineteenth-century science. The equations give a unified and complete theory for understanding electromagnetic (EM) wave phenomena. Solving Maxwell's equations is an important method for investigating the propagation, radiation, and scattering of EM waves.

The FDTD method, first introduced by Yee in 1966 [4], is a way to solve Maxwell's equations. The differential form of these equations and constitutive relations can be written as follows:

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} - \sigma_m \vec{H} - \vec{M} \tag{32.1}$$

$$\nabla \times \vec{H} = \frac{\partial \vec{D}}{\partial t} + \sigma_e \vec{E} + \vec{J} \tag{32.2}$$

$$\nabla \cdot \vec{D} = \rho_e; \quad \nabla \cdot \vec{B} = \rho_m \tag{32.3}$$

$$\vec{D} = \epsilon \vec{E}; \qquad \vec{B} = \mu \vec{H} \tag{32.4}$$

In equations 32.1 through 32.4, the following symbols are used:

$\vec{E}$: electric field
$\vec{D}$: electric flux density
$\vec{J}$: electric current density
ϵ: electrical permittivity
σ_e: electric conductivity

$\vec{H}$: magnetic field
$\vec{B}$: magnetic flux density
$\vec{M}$: equivalent magnetic current density
μ: magnetic permeability
σ_m: equivalent magnetic conductivity

First, the FDTD method replaces $\vec{D}$ and $\vec{B}$ in equations 32.1 and 32.2 with $\vec{E}$ and $\vec{H}$ according to the constitutive relations in equation 32.4, which yields Maxwell's curl equation.

$$\mu \frac{\partial \vec{H}}{\partial t} = -\nabla \times \vec{E} - \sigma_m \vec{H} - \vec{M}; \quad \epsilon \frac{\partial \vec{E}}{\partial t} = \nabla \times \vec{H} - \sigma_e \vec{E} - \vec{J} \tag{32.5}$$

All of the curl operators are then written in differential form and replaced by partial derivative operators, as shown in equation 32.6, with the $\vec{E}$ and $\vec{H}$ vectors separated into three vectors in three dimensions (i.e., $\vec{E}$ is separated into E_x, E_y, E_z, and $\vec{H}$ is separated into H_x, H_y, H_z):

$$curl\vec{F} = \nabla \times \vec{F} = \hat{x}\left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}\right) + \hat{y}\left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x}\right) + \hat{z}\left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y}\right) \tag{32.6}$$

We then can rewrite Maxwell's curl equations into six equations in differential form in rectangular coordinates.

$$\mu \frac{\partial H_x}{\partial t} = \frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} - \sigma_m H_x - M_x; \quad \epsilon \frac{\partial E_x}{\partial t} = \frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma_e E_x - J_x \tag{32.7}$$

$$\mu \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z} - \sigma_m H_y - M_y; \quad \epsilon \frac{\partial E_y}{\partial t} = \frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - \sigma_e E_y - J_y \tag{32.8}$$

$$\mu \frac{\partial H_z}{\partial t} = \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - \sigma_m H_z - M_z; \quad \epsilon \frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma_e E_z - J_z \tag{32.9}$$

Second, in preparation for "discretizing" the model in the next step, the model size, unit size, and unit timestep must be determined. The FDTD method establishes a model space, which is the physical region where Maxwell's equations are solved or the simulation is performed. The model space is then discretized to a number of cells, and the time duration, t, is discretized to a number of timesteps. The unit cell size should be small enough to ensure the accuracy of the result, but large enough to minimize the number of cells in order to save computation resources.

Although half of the EM wavelength is an upper bound of the cell size by the Nyquist sampling theorem, the cell size is often set to less than one-tenth of the EM wavelength for better results [1]. The model size depends on the number of cells in the model space, which is usually inversely proportional to the size of the unit cell. The unit timestep is calculated by following the Courant condition, which states that it must be less than the time the EM wave spends traveling to the adjacent unit cell. For a ground-penetrating radar example, assuming a central frequency of 1.25 GHz, the central wave length is 0.24 m. We set the unit cell size to 0.012 m, which is one-twentieth of the central wave length, for good simulation quality. The timestep can be set to 0.02 ns, which meets the Courant condition.

Every cell in the model space has its associated electric and magnetic fields. The material type of each cell is specified by its permittivity ϵ, permeability μ, and conductivity σ. The three-dimensional grid shown in Figure 32.1, the "Yee cell" [4], is helpful for understanding the discretized EM model space. The Yee cell is a small cube that can be treated as a single cell picked from the discretized model space; Δx, Δy, and Δz are the three dimensions of this cube. We use (i,j,k) to denote the point whose real coordinate is $(i\Delta x, j\Delta y, k\Delta z)$ in the model space. Instead of placing the E and H components in the center of each cell, the E and H field components are interlaced so that every E component is surrounded by four circulating H components and every H component is surrounded by four circulating E components.

Maxwell's equations in rectangular coordinates—equations 32.7 through 32.9—can be clearly illustrated by Yee's cell. For example, the H_x component located at point $(i, j+\frac{1}{2},\ k+\frac{1}{2})$ is surrounded by four circulating E components, two E_y components, and two E_z components, matching equation 32.7, which states that the H_x component increases directly in response to a curl of E components in the x direction. Similarly, the E_x component increases directly in response to the curl of the H components, as shown in Figure 32.2, also matching equation 32.7. We represent an electric component E_z at the discretized three-dimensional coordinate $(i\Delta x, j\Delta y, (k+\frac{1}{2})\Delta z)$ as $E_z|_{i,j,k+\frac{1}{2}}$, and when the

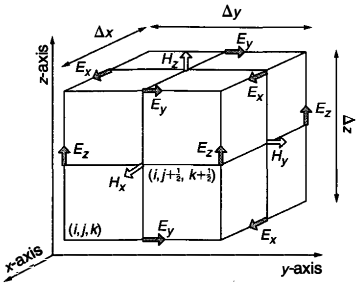


FIGURE 32.1 ■ The geometrical representation of the three-dimensional Yee cell.

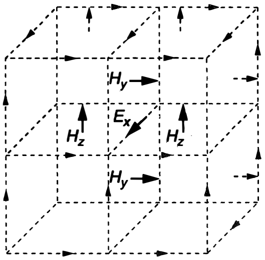


FIGURE 32.2 ■ Example of electric and magnetic components on a 4-cell grid.

current time is in the discretized N^{th} timestep, we denote the same component as $E_z|^N_{i,j,k+\frac{1}{2}}$.

Third, all of the partial derivative operators in equations 32.7 through 32.9 are replaced by their central difference approximations, as illustrated in equation 32.10. The second-order part of the Taylor series expansion is discarded to keep the algorithm simple and reduce the computational cost. Also, the variable without partial derivative can be approximated by time averaging, as shown in equation 32.11, which has a similar structure to the central difference approximation.

$$\frac{\partial f(u_o)}{\partial u} = \frac{f(u_o + \Delta u) - f(u_o - \Delta u)}{2\Delta u} + O[(\Delta u)^2] \tag{32.10}$$

$$f(u_o) = \frac{f(u_o + \Delta u) + f(u_o - \Delta u)}{2} \tag{32.11}$$

For example, equation 32.7 is changed to

$$\mu\frac{H_x(t_0 + \frac{\Delta t}{2}) - H_x(t_0 - \frac{\Delta t}{2})}{\Delta t} = \frac{E_y(z_0 + \frac{\Delta z}{2}) - E_y(z_0 - \frac{\Delta z}{2})}{\Delta z} \tag{32.12}$$
$$- \frac{E_z(y_0 + \frac{\Delta y}{2}) - E_z(y_0 - \frac{\Delta y}{2})}{\Delta y} - \sigma_m \frac{H_x(t_0 + \frac{\Delta t}{2}) + H_x(t_0 - \frac{\Delta t}{2})}{2} - M_x$$

After these modifications, the FDTD method turns Maxwell's equations into a set of linear equations from which we can calculate the electric and magnetic fields in every cell in the model space. We call these equations *the electric and magnetic field updating algorithms*. Six field-updating algorithms form the basis of the FDTD method. For example, the field-updating algorithm for the H_x

component, derived from equation 32.12 or equation 32.7, is given by

$$\left(\frac{\mu}{\Delta t}+\frac{\sigma_m}{2}\right)H_x\Big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{N+\frac{1}{2}}=\left(\frac{\mu}{\Delta t}-\frac{\sigma_m}{2}\right)H_x\Big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{N-\frac{1}{2}} \\ +\frac{1}{\Delta z}\left[E_y\Big|_{i,j+\frac{1}{2},k+1}^{N}-E_y\Big|_{i,j+\frac{1}{2},k}^{N}\right]-\frac{1}{\Delta y}\left[E_z\Big|_{i,j+1,k+\frac{1}{2}}^{N}\right. \\ \left.-E_z\Big|_{i,j,k+\frac{1}{2}}^{N}\right]-M_x\Big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{N}. \tag{32.13}$$

32.1.2 The FDTD Algorithm

The FDTD algorithm, whose flow diagram is shown in Figure 32.3, is based on these equations. It first establishes the model space and specifies the material properties and the excitation source. The source can be a point source, a plane wave, an electric field, or another option depending on the application. The algorithm then runs through the electric and magnetic updating algorithms on every cell in the model space and loops through every timestep. The output of the FDTD algorithm can be any electric or magnetic field data from any cell in any timestep.

The electric and magnetic fields depend on each other. As we can see from equation 32.13, the current timestep's magnetic field depends on the electric fields in the surrounding cells. Similarly, the current timestep's electric field depends on the magnetic fields in the surrounding cells. Because of this dependence between the electric and magnetic fields, we cannot update them

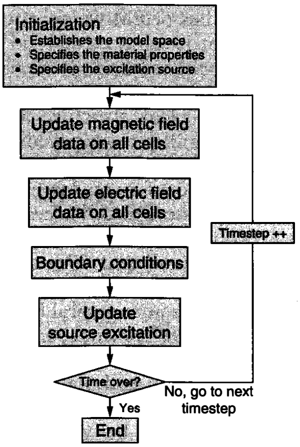

Initialization
• Establishes the model space
• Specifies the material properties
• Specifies the excitation source
Update magnetic field data on all cells
Update electric field data on all cells
Timestep ++
Boundary conditions
Update source excitation
Time over?
No, go to next timestep
Yes
End

FIGURE 32.3 ■ The flow diagram of the FDTD algorithm.

in parallel. So the FDTD algorithm updates the electric and magnetic fields in an interlaced manner, timestep by timestep, until the job finishes. First, all the magnetic fields in all cells in the model space are updated; next, all the electric fields in all cells, then the source excitation and boundary conditions, are given to the model space; finally the algorithm goes to the next timestep and starts from the magnetic fields again.

The boundary condition computation consists of special algorithms to deal with the unit cells located on the boundary of the model space. The preceding electric and magnetic updating algorithms work accurately in the interior of the model space; however, because the cells on the boundary do not have the adjacent cells needed, the algorithm does not work properly and, as a result, there are algorithm-introduced reflections on the boundary. Special techniques, called absorbing boundary conditions (ABC), are necessary to deal with boundary cells, to prevent nonphysical reflections from outgoing waves, and to simulate the extension of the model space to infinity. The development of efficient ABCs is very important for the FDTD method.

The perfect matched layers (PMLs) ABC [5] sets the outer boundary of the model space to an absorbing material medium layer, which absorbs most of the impinging wave and has low reflection for most incidence angles. The UPML (uniaxial PML) ABC [3]—a modification of PML—uses a generalized formulation on the entire FDTD model space that integrates the boundary condition and electric field updating algorithms, simplifies the FDTD algorithm, and makes a good model for hardware datapath design. Although UPML introduces extra computation and memory consumption, the quality of the uniaxial PML is especially good for dispersive media, which is useful in solving many realistic problems (e.g., the dispersive soil found in modeling ground-penetrating radar and medical studies of EM waves' effects on dispersive human tissue).

The FDTD algorithm is an accurate and successful modeling technique for computational electromagnetics. It is flexible, allowing the user to model a wide variety of EM materials and environments on most scales. It is also easy to understand, with its clear structure and direct time domain calculation. However, FDTD is data and computationally intense. It needs to visit all the cells in every step of the calculation, forcing a large working set. The amount of data in the FDTD model space can be very large for large model sizes, creating a heavy burden on both memory storage and access. The computation is also intense for each cell in the FDTD model space, including updating six electric and magnetic fields and the boundary conditions. This complexity makes the FDTD algorithm run slowly on a single processor—modeling an electromagnetic problem using the FDTD method can easily require several hours. Without powerful computational resources, FDTD models are too time consuming to be implemented on a single computer node. Accelerating FDTD with inexpensive and compact hardware will greatly expand its application and popularity, which is the purpose of an FPGA implementation.

The FDTD algorithm can be viewed as a cellular automata (CA) (see Section 5.2.5). A cellular automaton is a discrete model that consists of an infinite or finite grid of cells, where the state of every cell at discrete time t is a function of the states of a finite number of neighborhood cells at discrete time $t-1$. Every cell has the same rule for updating. The updating algorithm loops through the whole discrete model and then goes to the next discrete time $t+1$. The FDTD algorithm exactly fits the definition of a CA. First, it creates a discrete model space, discretizing both physical space and time with a uniform grid. Second, every cell in the model space follows the same rule (six uniform updating algorithms) for updating the electric and magnetic fields. Finally, the calculation loops though cells to simulate the phenomenon of the whole model space through time. A hardware implementation of the FDTD method is thus a template hardware design for most CA problems.

32.1.3 FDTD Applications

The FDTD method is an important tool for investigating the propagation, radiation, and scattering of EM waves. Before the 1990s the cost of solving Maxwell's equations directly was high and most of the related research was for military–defense purposes. For example, engineers used huge parallel supercomputing arrays to model the radar wave reflection of airplanes by solving Maxwell's equations, trying to develop an airplane with a low radar cross-section [6]. The difficult task of solving Maxwell's equations has had more economical solutions since 1990 with the development of fast computing resources applied to the FDTD method. Now FDTD has spread to many areas, including discrete scattering analysis, antenna and radar design [3], EM wave phenomena analysis on multilayer circuit boards [6], subsurface sensing and ground-penetrating radar (GPR) detection [7, 8], studies of EM wave phenomena in the human body, and the study of breast cancer detection using EM waves [9, 10]. We apply our FDTD solution to landmine detection using GPR, breast cancer detection, and spiral antenna modeling.

Ground-penetrating radar

The FDTD method has been used to simulate GPR applications for buried landmine detection [7, 8]. A three-dimensional FDTD model, as shown in Figure 32.4, simulates the wave propagation and scatter response of three-dimensional GPR geometries with realistic dispersive soil along with air, metal, and dielectric media. The UPML ABC produces good results for this application. The three-dimensional model has been validated by experiments performed with a commercially available GPR system and realistic soil.

Breast cancer detection

Because of the large difference in electromagnetic properties between malignant tumor tissue and normal fatty breast tissue, microwave breast cancer detection has attracted much interest because it may overcome some of the shortcomings of X-ray detection. Accurate computational modeling of microwaves in human tissue with the FDTD method is promising for breast cancer detection research. Researchers built a three-dimensional model of the human breast [9, 10], shown in Figure 32.5, that includes a semi-ellipsoid geometric representation of the

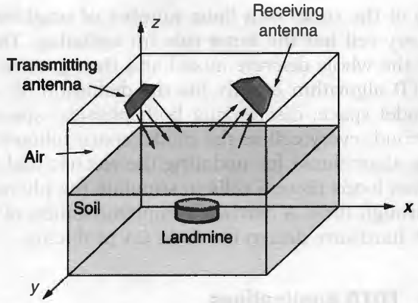


FIGURE 32.4 ■ A three-dimensional FDTD application of landmine detection using GPR.

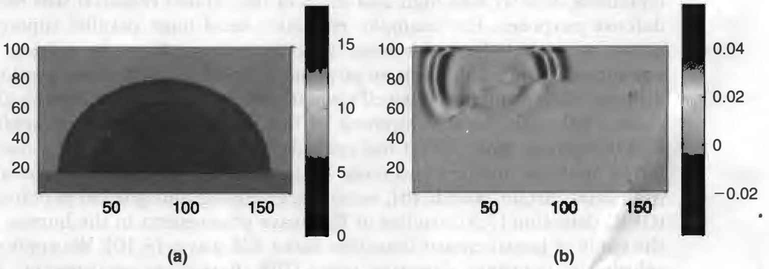


FIGURE 32.5 ■ Three-dimensional FDTD application of microwave breast cancer detection: (a) geometry map; (b) simulated model space.

breast and a planar chest wall. The modeling is in the range of 30 MHz to 20 GHz, and the UPML ABC is implemented.

Spiral antenna model

The spiral antenna is a popular frequency-independent antenna. As shown in Figure 32.6, we use the FDTD method to simulate the radiation of the Archimedean spiral antenna as an example of its application to antenna design.

Clearly, FDTD is a powerful tool that can be used in many different applications. However, its data-intense and computationally intense properties make it run slowly on a single processor.

The reconfigurable hardware implementation of the FDTD method can greatly accelerate the running speed of the algorithm and maintain its accuracy and flexibility. For example, the breast cancer detection FDTD algorithm running on a single processor may require hours, while the hardware implementation delivers results in minutes, enabling a medical device that

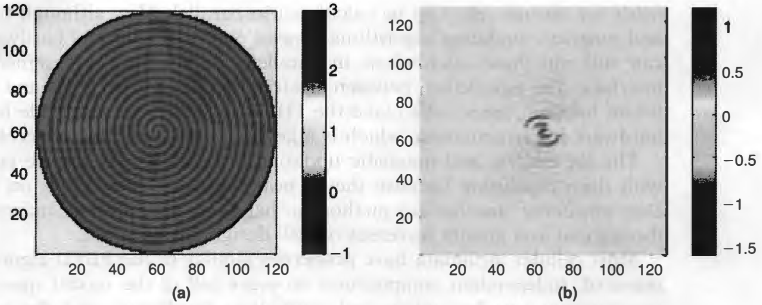


FIGURE 32.6 ■ (a) The floorplan of the spiral antenna model; (b) an FDTD-simulated two-dimensional space.

delivers an answer during the examination. With the help of faster computing technology, the FDTD method will be applied to more research areas and applications.

32.1.4 The Advantages of FDTD on an FPGA

Compared to software running on a general-purpose processor, the advantages of an FPGA implementation are evident—faster speed, smaller size, lower power consumption; the last two advantages are significant, especially compared to a large computer cluster.

Compared to an ASIC finite-difference time-domain design, the FDTD field-programmable gate array (FPGA) implementation has the advantage of flexibility while accelerating the algorithm. The FDTD method models a wide variety of electromagnetic problems that are difficult to cover with a single hardware design. With an FPGA, a designer can modify the model size, the materials, and the parameters and even introduce new updating algorithms and boundary conditions easily. While the ASIC may outperform the FPGA as to speed, size, and power, the reconfigurable property of an FPGA makes it more suitable for the FDTD algorithm.

We can achieve fast computation in an FPGA for finite-difference time-domain because FDTD has properties that make it very suitable for hardware implementation. These properties are its favorable structure for pipelining and parallelism and its constrained data ranges, which are good for fixed-point representation. They make the FDTD method especially suitable for FPGA implementation.

Parallelism and deep pipelining

The FDTD algorithm repeats the same electric and magnetic updating algorithms on every cell of the model space. These calculations are independent between each cell. As long as there are adequate hardware resources, the fields for several cells can be calculated in parallel. Also, although the electric and magnetic updating algorithms depend on each other, the hardware design can still run these calculations in parallel with a carefully designed memory interface. The parallelism between electrical and magnetic fields and the parallelism between space cells make the FDTD algorithm very suitable for parallel hardware implementation, which is a key method for hardware acceleration.

The six electric and magnetic updating algorithms can also be constructed with deep pipelining because they repeat the same calculation on each cell. *Deep pipelining*, another key method for hardware acceleration, maximizes data throughput and greatly increases overall design performance.

Most cellular automata have properties similar to the FDTD algorithm with repeated, independent computation on every cell of the model space. The CA computation can be constructed with deep pipelining, and the parallelism between discrete cells is the same as that available in any CA problem.

Fixed-point arithmetic

Floating-point representation provides high resolution and large dynamic range, but it can be costly. In hardware design, floating point uses slower arithmetic components and consumes more area. In contrast, fixed-point components have much faster speed and occupy less area. In applications where data resolution and dynamic range can be constrained, such as the FDTD algorithm, fixed-point arithmetic can provide similar precision and much faster speed than floating-point arithmetic.

The majority of the data in the FDTD algorithm is the six EM field variables and nine intermediate field variables for each cell in the model space. Since all the calculations in the FDTD method are linear, we can maintain the EM field data at a certain level of magnitude by normalizing the incoming source field magnitude. For example, if the source fields are between -1 and 1, all the EM field variables are between -1 and 1. In rare cases, we simulate the model space with a focus lens to magnify the EM field data. In this case we can estimate the EM data range and still keep the variables between -1 and 1 by normalizing the source field. Since all the EM field variables can be controlled in a fixed range, a fixed-point representation can be used for better performance with a relatively low error rate.

The uniaxial PML FDTD algorithm must be optimized for fixed-point representation. Several parameters in the algorithm have a much different order of magnitude than the EM fields. They may not be representable in fixed point directly or may result in a large error when quantized. Additional error can arise from arithmetic calculations with these parameters in fixed point. These errors can be canceled by making a few changes to the original FDTD algorithm. For example, very large and small coefficients can be multiplied together to create a medium-value coefficient to be used in the new equation. The modification has no effect on the result of the algorithm.

Careful analysis is important for fixed-point quantization to avoid errors. For normalized EM field values that range between -1 and 1, the data tends to be accurate to a relative error of 0.5 percent. The resolution of the fixed-point representation is determined by its data bit width. The longer the bit width, the higher the resolution, so the smaller the error. However, longer bit-width data uses more hardware resources. After careful study of the FDTD algorithm and representative data, we can pick a suitable bit-width with relatively small error (see also Chapter 23).

In conclusion, the FDTD algorithm is very suitable for hardware implementation. The FPGA implementation of the finite-difference time-domain method will empower many FDTD applications in medical, military, and other areas by providing fast, small, low-power, and inexpensive implementations. Many cellular automata, which share similar properties, are also suitable for FPGA hardware implementation. The FDTD hardware design we present in the next section is a good example of hardware implementations for CA.

32.2 FDTD HARDWARE DESIGN CASE STUDY

The FDTD algorithm has a clear structure for hardware design. For each cell in the model space, it reads the electric and magnetic data out of the memories, passes them through the updating algorithms, and writes the results back to the memories. The algorithm repeats this processing until it completes the model space; then it goes to the next timestep and does the same calculations again.

It is easy to separate any hardware design into datapath, memory interface, and control logic. For FDTD, the datapath implements all the electric and magnetic updating algorithms; the memory interface controls data reading, writing, and caching; and the control logic uses a finite-state machine (FSM) to control the progress of the whole design. However, because of its complexity, an efficient hardware implementation of FDTD is not straightforward. The FDTD algorithm is data intense. The electric and magnetic updating algorithms interface a lot with the input and output memories, which creates a heavy burden on the memory interface and data bandwidth. Also, the EM field dataset for the whole model space can be very large for a large model size (a 100×100×100 model may require 60 MB of memory space), meaning that local FPGA memory is insufficient to contain the entire problem.

The FDTD algorithm is also computationally intense. Every EM field has its own updating algorithms and boundary conditions. A special interlaced mechanism is used between the electric and magnetic updating algorithms, making them depend on each other. Many problems arise when considering the pipelining and parallelism of the datapaths. The FDTD algorithm is complex enough to reach the resource limits of most advanced FPGAs available on the market. Consideration of fixed-point quantization and resource performance trade-offs is very important for efficient hardware design.

One of the main purposes of a hardware implementation is to achieve better performance. To implement the FDTD algorithm on an FPGA efficiently, we need to consider the following:

- Determining the right precision for fixed-point representation (Section 32.2.2)
- Determining the memory hierarchy and designing the memory interface and cache module (see Memory hierarchy and memory interface subsection of Section 32.2.3)
- Determining the pipelining and parallelism by considering the trade-off between resources and performance (see Pipelining and parallelism subsection of Section 32.2.3)

It is important to analyze the data structures, algorithm structure, hardware architecture, and resource limits before design of hardware implementation. This section introduces a target reconfigurable platform, the WildStar-II Pro FPGA board, and lists its detailed specifications. Then we choose the suitable fixed-point representation by analyzing the quantization error of a fixed-point FDTD algorithm and the hardware resource limits. Then we go through the problems in the FDTD hardware implementation and provide detailed solutions and analyses. By carefully considering the trade-offs between hardware resources and performance, we can design the FDTD accelerator with the memory interface, pipelining, and parallelism optimal to the current FPGA computing board.

32.2.1 The WildStar-II Pro FPGA Computing Board

The FPGA board used here is a WildStar-II Pro/PCI reconfigurable FPGA computing board from Annapolis Micro Systems [12]. Its main features are summarized in Table 32.1; a block diagram of this board is shown in Figure 32.7. There are two Xilinx Virtex-II Pro FPGAs, each with 328 embedded 18×18 signed multipliers and 328×18-Kb BlockRAMs.

The embedded multipliers are much faster than a multiplier component implemented with reconfigurable logic, so it is best to use them if possible. The BlockRAMs are the fastest memory the designer can use in an FPGA design, operating as fast as 200+ MHz on the Virtex-II Pro chip. Critical data interchange and interfacing can be programmed using the BlockRAMs. A pair consisting of an embedded multiplier and a BlockRAM shares the same data and address buses in the Xilinx Virtex-II architecture, so once the embedded multiplier is used, we cannot use its

TABLE 32.1 ■ The main features of the WildStar-II Pro FPGA board

FPGA chips	Two Xilinx Virtex-II Pro XC2V70 FPGAs (33,088 slices, 328 embedded multipliers, and 5904 Kb BlockRAM)
Memory ports	Twelve DDRII SRAM ports totaling 54 MBytes (6×4.5 MBytes for each FPGA chip)
Memory bandwidth	Eleven GB/s memory bandwidth (6×72 bits for each FPGA chip)
PCI interface	133 MHz/64-bit PCI-X up to 1.03 GB/s

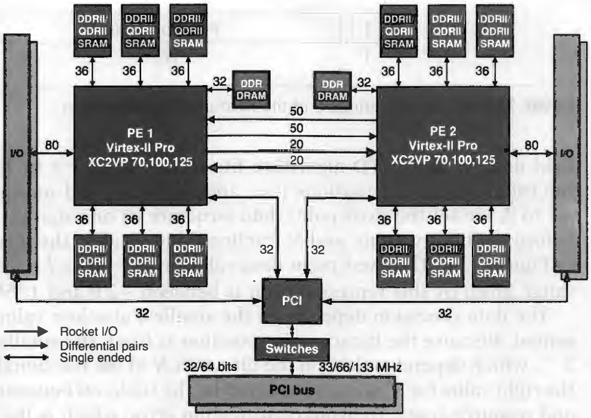


FIGURE 32.7 ■ A block diagram of the WildStar-II Pro FPGA board.

corresponding BlockRAM, and vice versa. Thus, the sum of the total number of embedded multipliers and BlockRAMs used must be less than 328.

Each FPGA is connected to six independent onboard memories, which are 1-M×36-bit DDRII SRAM that have 72-bit data bandwidth and speeds up to 200 MHz. The size of each SRAM is 36 Mbits, or 4.5 MBytes, so the total SRAM attached to each FPGA is 27 MBytes. The WildStar-II Pro board is connected to the desktop computer via a PCI-X interface, with a DMA data transfer rate up to 1 GB/s between the host PC and the FPGA.

The WildStar-II Pro is a typical commercial off-the-shelf (COTS) FPGA computing board, which is widely available and easy to set up. These boards normally contain one or two FPGA chips. Each FPGA chip may be connected to several onboard memories consisting of SRAM or DRAM. The computing boards are often PCI boards for a desktop computer or PCMCIA cards for a laptop. Data and control signals can be transferred between the FPGA computing board and the host PC via either standard PCI transfer or fast DMA transfer. The FDTD hardware design is based on the WildStar-II Pro board but can be easily modified for other COTS FPGA boards.

32.2.2 Data Analysis and Fixed-point Quantization

Because of its limited data range and favorable algorithm properties, the FDTD method is suitable for fixed-point arithmetic (see Section 32.1.4). To use fixed-point representation with the algorithm, we need to first decide its representation and the right data precision.

For simplicity, we use a 2's complement fixed-point representation that has a fixed number of digits before and after the binary point. Because the EM

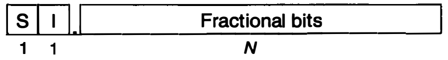


FIGURE 32.8 ■ The data structure of the fixed-point representation.

field data in the FDTD algorithm fits in the range −1 to 1, and the results of the intermediate calculations (i.e., add, subtract, and multiply) fit in the range −2 to 2, we set the fixed-point data structure as one sign bit S, one integer bit I before the binary point, and N fractional bits F_i after the binary point, as shown in Figure 32.8. The fixed-point data value is $V = -S \cdot 2 + I + \frac{1}{2^N} \sum_{i=0}^{N-1} 2^i F_i$. The data range given by this representation is between −2.0 and 1.999.

The data precision depends on the smallest absolute value that can be represented. Because the binary point position is fixed, the smallest absolute value is 2^{-N}, which depends solely on the bit width N of the fractional part. To determine the right value for N, we need to consider the trade-off between quantization error and resource costs. To avoid quantization error, which is the difference between the fixed-point and corresponding floating-point data, a longer data bit width is preferable. However, longer data bit widths require larger and slower arithmetic components and put more burden on memory bandwidth and data storage. The problem is how to pick the optimal data bit width such that the fixed-point FDTD algorithm generates acceptable quantization error and consumes a reasonable amount of hardware resources.

To determine this, we wrote the FDTD algorithm in C code both in double-precision floating-point and fixed-point arithmetic and compared the results. Fixed-point representation is simulated by long integers in C, which have a 32-bit maximum bit width. We used two long integer variables to represent one fixed-point datum up to 64 bits. Based on this representation, we created add, subtract, and multiply components for each fixed-point bit width. The C code simulates the fixed-point arithmetic and produces results that are exactly the same as the hardware output. Thus, this C code also can be used for hardware results verification.

By comparing floating-point and the corresponding fixed-point data results for the same model space, we can calculate the relative error, defined in equation 32.14, over the time period that the algorithm runs.

$$\textit{Relative error} = \frac{|\textit{floating-point data} - \textit{fixed-point data}|}{|\textit{floating-point data}|} \qquad (32.14)$$

We studied the following six experimental FDTD models to investigate quantization errors:

- The two-dimensional and three-dimensional soil media–based GPR landmine detection models
- The two-dimensional and three-dimensional human tissue media–based tumor detection models
- The two-dimensional and three-dimensional spiral antenna models

TABLE 32.2 ■ Detailed specifications of the experimental FDTD models

	2D landmine detection	3D landmine detection	2D breast detection	3D breast detection	2D spiral antenna	3D spiral antenna
Size	150×100	50×50×50	240×140	80×60×40	120×120	120×120×25
Time duration	2000	2000	2000	2000	2000	2000
Source	Plane wave		Point source		Point source	
Media	Soil, air, dielectric		Human tissue, dielectric		Metal, air, dielectric	

TABLE 32.3 ■ Relative error between fixed-point and floating-point representation

		Timestep (%)					Average
Bit width	Field	400	600	1000	1400	1600	across timestep (%)
29	E_x	9.187	3.503	0.280	0.182	0.558	2.742
	H_y	12.440	0.124	1.431	0.244	0.264	2.901
	H_z	2.706	1.925	0.472	0.200	0.235	1.108
31	E_x	3.861	0.941	0.058	0.032	0.110	1.001
	H_y	3.681	0.025	0.295	0.042	0.001	0.809
	H_z	1.905	0.461	0.105	0.039	0.046	0.511
33	E_x	2.155	0.209	0.016	0.010	0.031	0.484
	H_y	2.101	0.007	0.077	0.012	0.014	0.442
	H_z	1.479	0.120	0.029	0.010	0.013	0.330
35	E_x	1.729	0.063	0.004	0.002	0.008	0.361
	H_y	1.420	0.002	0.021	0.003	0.004	0.290
	H_z	1.314	0.030	0.007	0.003	0.003	0.271

The specifications of these models are listed in Table 32.2. For all of them, we studied the average relative errors between the floating-point and the fixed-point results. This section analyzes the GPR model results. The other model spaces are similar.

Table 32.3 shows average relative errors for the fractional data bit-width range from 29 to 35 bits in the two-dimensional GPR landmine detection model. E_x, H_y, and H_z are electric and magnetic field data. The relative errors are plotted in Figure 32.9. Those of both electric and magnetic field data decrease as bit widths increase. However, the rate of decrease slows as the bit widths increase. Considering both the relative error and the bit-width cost, a 33-bit fractional part is a good choice for the trade-off between data precision and hardware resources. The average absolute error for this representation is on the order of 10^{-8} for magnetic field data and on the order of 10^{-6} for electric field data; the average relative error is about 0.3 to 0.5 percent. Thus, this representation satisfies the accuracy requirement that the relative error is less than 0.5 percent.

In addition to quantization error analysis, we need to consider the resource limits of the real hardware device in determining the fixed-point data bit width. The FDTD model space will be stored in the onboard SRAMs on the WildStar-II

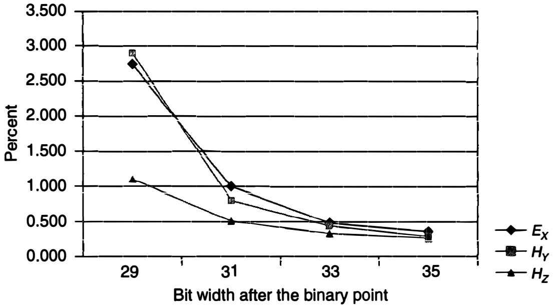


FIGURE 32.9 ■ The relative error between fixed-point and floating-point arithmetic for different bit widths.

Pro FPGA board. The SRAM memory chip we used has size 512K × 36 bit. The data is stored in the memory in units of 36 bits. Any data more than 36 bits wide will take two memory units. To keep the memory interface working efficiently, we want to set the data bit width less than or equal to 36 bits.

The embedded multiplier provided on the Xilinx Virtex II-Pro FPGA chip, an 18×18-bit 2's complement signed multiplier, is much faster than the multiplier component implemented by normal reconfigurable logic. Four embedded multipliers can form a 35×35-bit signed multiplier. However, to construct a 36×36-bit signed multiplier, nine embedded multipliers are needed. Because the number of multipliers is limited and very useful in the FDTD algorithm, it is uneconomical to use a 36×36 multiplier or 36-bit data. A data bit width of 35 bits is more efficient for the embedded multiplier. Because the fixed-point quantization error analysis performed in the last section also recommends a data bit width of 35, we choose 35 bits of data as the fixed-point data structure based on both quantization error and resource limits.

32.2.3 Hardware Implementation

After choosing the fixed-point data representation, we then study two very important problems in the FDTD hardware implementation: memory interfacing and pipelining and parallelism.

Memory hierarchy and memory interface

Because the EM field data is proportional to the number of cells in the FDTD model space, the dataset can be very large. Every cell in the FDTD model space has 6 EM field data and 9 intermediate field data for the UPML computation, adding up to 15 field data. An FDTD model space may have millions of cells, require hundreds of megabytes of memory space, and easily exceed the limits of the memory available inside the FPGA chip. Therefore, the data must be stored in larger memories, which are normally slower than the fast on-chip memories, outside the FPGA chip.

The data stored in the slower memories needs to be transferred to the processing core in the FPGA. The processing core is composed of six electric and magnetic updating algorithms, which require very large amounts of input data. In the worst case, three electric updating algorithms require 36 input data and three magnetic ones require 18, adding up to 54 input data for each dispersive UPML FDTD cell. In other words, to make sure that the processing core works at full speed, we need to transfer 54 input data from off-chip memory to the FPGA for each cell. The data transfer puts a heavy burden on the interface between the off-chip memories and the FPGA design. To provide the necessary data at the right time and to optimize the efficiency of the memory interface, we need to determine how to organize the memory resources efficiently by considering the size, speed, and interface bandwidth of each memory resource.

There are three levels of memory hierarchy, based on the WildStar-II Pro/PCI FPGA computing board:

- The fast and wide data-width on-chip memory (BlockRAM) integrated on the FPGA chip
- The fast but limited data-width onboard memory located on the FPGA computing board
- The slow memory for the FPGA to access located in the host PC

BlockRAMs are programmable memories that are integrated inside modern FPGA chips. A Xilinx Virtex-II Pro XC2V70 FPGA contains 328 BlockRAMs, 18 Kb each, with a maximum data width of 36 bits. They can be implemented as small memory blocks or cascaded to form large memory blocks. They also can be programmed to be different depths and widths to fit the hardware design and data structures. They are fast memory units in terms of latency, with only one clock cycle delay for clock cycles up to 200 MHz.

Although BlockRAMs are fast and flexible memory resources, there is much less BlockRAM available compared to off-chip memory. So normally we do not fit the entire model space's data into BlockRAMs. Instead, they are used to build cache modules that read from and write to off-chip memories continuously and feed data to the processing core. What's more, the BlockRAMs are true dual-ported RAM units, and a group of BlockRAMs can provide a very wide data width to the processing core when aggregated together. For example, 54 BlockRAMs on the input side can provide a 54×36-bit data width every clock cycle, which allows the FDTD processing core to run at full speed. The *data width* is the number of bits that can be transferred in one clock cycle. Along with clock frequency, data width determines the data transfer speed (bandwidth) of the memory interface.

Onboard memories, which directly communicate with the FPGA chip, are relatively slower than BlockRAMs in terms of latency, but they are usually much larger in size, varying from megabytes to hundreds of megabytes. The interface between the memory chips and the FPGA chips follows the read/write cycles of the specific memory chips, which are normally single-ported data access with limited data transfer width. Because of the heavy data access required by the FDTD algorithm, the onboard memory bandwidth is very important to the performance of the FDTD design.

As discussed before, the six electric and magnetic updating algorithms need 54 input data for each FDTD cell, which is around 54×36-bit×100 MHz = 194 Gb/s—far beyond the onboard memory bandwidth of typical FPGA boards. The input data of a single cell have to be transferred to the updating algorithms in several clock cycles, while the updating algorithms can calculate results with a throughput of one cell per clock cycle. So, the onboard memory data transfer bandwidth is the bottleneck of the FDTD design. Memory bandwidth is an important specification in choosing the FPGA computing board for a finite-difference time-domain implementation. To solve this bottleneck, we introduce the managed-cache module that is explained in the next subsection.

The memories in the host PC can be accessed by the FPGA via the PCI or other interfaces. These interfaces are normally slower than the two memory interfaces we have discussed, so we treat the memories in the host PC as the slowest memory, no matter what the actual speed. This memory can be used for data initialization at design startup and data retrieval at the end. At the start of processing, the model space data are loaded from the host PC to the onboard memory and loaded back to memory in the PC at the end of the design. If the onboard memory is not big enough to hold the whole model space, the memory in the host PC will be the primary memory and the data need to be transferred to and from the onboard memory throughout the entire calculation, slowing down the whole design. The size of onboard memories is thus another critical specification in choosing an FPGA computing board.

The memory hierarchy and memory interface structure used in this design is shown in Figure 32.10. We use one FPGA and six onboard memories on the WildStar-II FPGA board. The FDTD field data stored in the onboard memories are sent to the electric and magnetic field–processing cores for calculation via

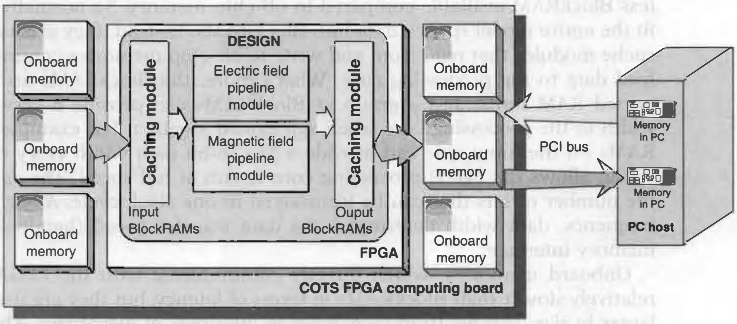


FIGURE 32.10 ■ A structural diagram of the memory interface.

the caching modules built using the BlockRAMs on the FPGA chip. The 3-level memory hierarchy formed from the host PC, the onboard memories, and the BlockRAM caching modules ensure that the electric and magnetic field updating algorithms work at optimal speed.

As shown in Figure 32.10, the BlockRAM caching modules are split into two parts: input and output. The six onboard memories, which are used to store EM field data, are split into two parts also. The entire FDTD model space of the previous timestep is stored in the input onboard memories, and the calculation results, which comprises the data in the current timestep, will be stored in the output onboard memories. In the next timestep, the role of the onboard memories is swapped. The original output onboard memories, which store the current timestep's data, will be connected to the input caching module and the original input onboard memories will be connected to the output module to store the next timestep's result.

The separation of the input and output onboard memories eliminates the need for simultaneous read/write access to the same memory. Because the onboard memories are single ported, shifting between reading and writing to the same memory will create overhead and greatly reduce the speed of the design. By separating input and output memory, we can read from and write to the onboard memories at the same clock cycle, and continue reading and writing a group of data on every clock cycle. So, although the separation of the memory interface does not change the memory bandwidth, the data-transfer rate of the memory interface is increased. Also, the separation makes the structure of the memory interface clearer and the swapping mechanism avoids the extra effort of transferring data from output memories to input memories at the end of every timestep. This swapping of input and output memories is a common hardware design technique to increase throughput.

Managed-cache module

As introduced in the previous section, onboard memory data bandwidth is limited on the FPGA computing board, so the EM field data cannot be transferred to the FPGA fast enough to allow the processing core to run at full speed. To solve this memory transfer bottleneck, we need to introduce the managed-cache module, which is an important part of the memory interface design.

Memory transfer bottleneck Although the FDTD processing core requires a large amount of input data, the input data for each cell are the EM field data in their nearest-neighbor cells. For two cells located near each other in the FDTD model space, some of the nearest-neighbor cells are the same. The cache module between the onboard memories and the hardware processing cores is designed to avoid reading the same data multiple times from onboard memories.

All of the input data for each cell are from their near neighbors, which means the data are located in a small cubic window around the current cell. If the managed-cache module is designed to be larger than this cubic window, when we calculate the fields of the next cell, the processing core can get all the necessary input data from the cache module. Among the input data, only a little is new, so we only need to fetch the new data from the onboard memories every clock cycle, which greatly reduces the data-transfer burden. At the same time, some of the old data becomes obsolete. In the managed-cache module, we can replace the obsolete data with the new data fetched from onboard memory.

Ideally, we keep the processing core running at full speed so that it calculates one cell's EM data per clock cycle. The managed-cache module needs to be designed to provide all the necessary input data for the processing core, while fetching only one new cell's data from onboard memory every clock cycle. Since every UPML FDTD cell has 15 field data and the processing core needs up to 54 field data inputs, an ideal managed-cache module will fetch 15 field data from onboard memory every clock cycle and provide a data width of 54 field data to the processing core, solving the memory bandwidth bottleneck problem by reducing the number of fetches to 15 every clock cycle, which is 15×36-bit×100 MHz = 54 Gb/s. This rate can be supported by the WildStar-II Pro FPGA computing board. We explain how to realize this ideal cache module in the next two subsections.

Dataflow and processing core optimization To simplify the explanation of how to optimize the dataflow and how to optimize the processing core, we start from a two-dimensional FDTD algorithm, which can be directly reduced from the three-dimensional FDTD algorithm by considering only one plane in the three-dimensional model. The two-dimensional algorithm updates three EM field data instead of six, handling much less data transfer and calculation, but it keeps the same algorithm structure and datapath. For a two-dimensional model plane of size $N \times N$, we assume that each N cell row is a basic processing unit. Calculating one row of data means updating all EM field data for this row.

The cache modules separate the whole dataflow of the FDTD design into three processes: (1) READ from the input onboard memory and store to the input cache module; (2) read from the input cache module, CALCULATE, and write the result to the output cache module; (3) read from the output cache module and WRITE to the output onboard memory. These three processes can be run in parallel since the cache module can be read from and written to at the same time (i.e., because the cache modules are built from dual-ported BlockRAMs). The parallelism of READ, CALCULATE, and WRITE means that the FDTD design can, at the same time, READ one row of data, CALCULATE the previous loaded row, and WRITE out the results of the row before that. We can understand this as systemwide pipelining in the dataflow. Each process is a pipeline stage. Rows of data are pushed into this 3-stage pipeline, one at a time. Compared to running the three processes serially, this optimized dataflow structure increases the throughput by a factor of 3.

For a two-dimensional plane of size $N \times N$, a simple 2-row cache module (size $2 \times N$) realizes the READ/CALCULATE/WRITE pipelining. As shown in Figure 32.11, the data can be READ from input onboard memory and stored in the second input cache row while the CALCULATE process works on the previously loaded data in the first input cache row. The result is stored in the first output cache

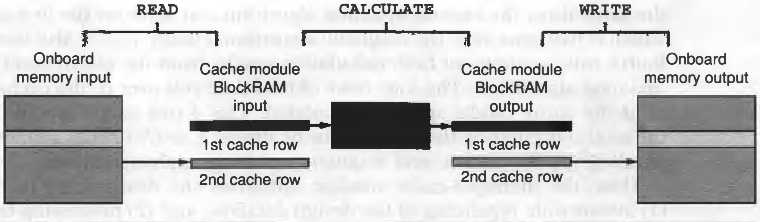


FIGURE 32.11 ■ A structural diagram of the simple 2-row cache module.

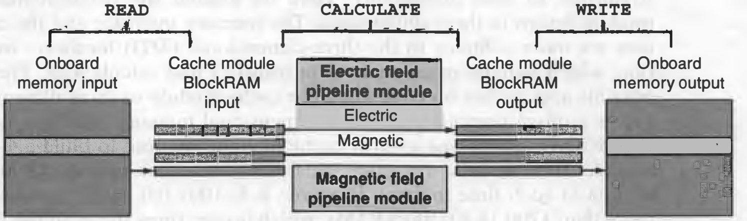

READ
CALCULATE
WRITE
Onboard memory input
Cache module BlockRAM input
Electric field pipeline module
Cache module BlockRAM output
Onboard memory output
Electric
Magnetic
Magnetic field pipeline module

FIGURE 32.12 ■ A structural diagram of the two-dimensional managed-cache module.

row while the previous row's result is read from the second output cache row and WRITTEN to output onboard memory. This cache module structure can be applied to other CA designs.

Furthermore, for FDTD implementation the managed-cache module enables parallel implementation of the electric and magnetic updating algorithms in the implementation of the processing core. Because of the data dependency of the electric updating algorithm on the magnetic updating algorithm—the former needs the current result of the latter—we cannot directly update the M-field and E-field in parallel until we introduce two extra rows in the managed-cache module (see Section 32.1.2). Why two extra rows?

The electric updating algorithm needs to have newly updated magnetic data in the current cell and newly updated magnetic data in the cell below as inputs. So, the electric updating algorithm needs to wait until the magnetic updating algorithm finishes two rows of computation. As long as the cache has two extra rows to save the newly calculated magnetic data, we can run the magnetic updating algorithms two rows ahead of the electric updating algorithms and partially overlap their computation. This is illustrated in Figure 32.12.

For a two-dimensional model space of size $N \times N$, the managed-cache module stores four rows ($4 \times N$) of field data. While the READ process is working on the fourth cache row, the magnetic updating algorithm can work on the data in the third row, which was just read from the memories by the last READ. At the same time, the electric updating algorithm can work on the first cache row, which is two rows after the magnetic algorithm. Finally, WRITE also works on the fourth row, sending out both calculation results from the electric and magnetic updating algorithms. The four rows of field data roll over in the cache modules until the entire model space is calculated. This 4-row cache module improves the total computation time by a factor of almost 2, or $(N+2)/(2N+2)$, by partially parallelizing the electric and magnetic updating implementations.

Thus, the managed-cache module optimizes the design here in two ways: (1) systemwide pipelining of the design dataflow, and (2) processing-level parallelism of the electric and magnetic updating algorithms.

Expansion to three dimensions Here we expand the two-dimensional cache module design to three dimensions. The memory interface and the cache modules are more complex in the three-dimensional FDTD hardware implementation, which handles many more data transfers and calculations. There are two possible approaches for upgrading the cache module to three dimensional. The first is a direct upgrade of the two-dimensional memory interface, as shown in Figure 32.13. Instead of a 4-row cache module, we need to build a 4-slice cache module. Here we READ one slice, CALCULATE one slice, and WRITE out one slice of data at each time interval. However, a 4×100×100 cache module consumes more than 1200 18-Kb BlockRAMs, which is over three times all the BlockRAMs on the targeted Virtex-II Pro XC2V70. This approach is not feasible for large three-dimensional model spaces.

The second approach reduces the size of the cache module to 4×3 rows of field data by cutting the model space into slices and then into rows. As shown in Figure 32.14, the cache module reads three rows of field data at each time interval, goes through the current vertical slice until it finishes, and then goes to the next vertical slice in the model space. Instead of a 4-slice cache module, we only need to build a 4×3 row cache module. This method minimizes BlockRAM consumption; however, it sacrifices overall design speed to achieve larger model space compatibility. We READ three rows of data at each time interval to CALCULATE only one row of results. This is because we need the current row

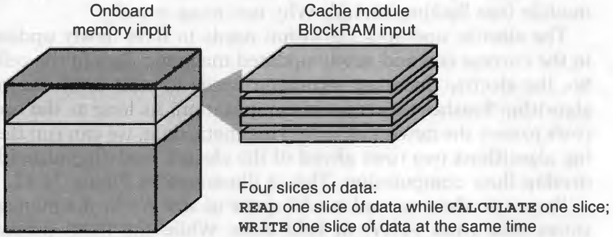

Onboard memory input
Cache module BlockRAM input
Four slices of data:
READ one slice of data while CALCULATE one slice;
WRITE one slice of data at the same time

FIGURE 32.13 ■ A structural diagram of the 4-slice caching design.

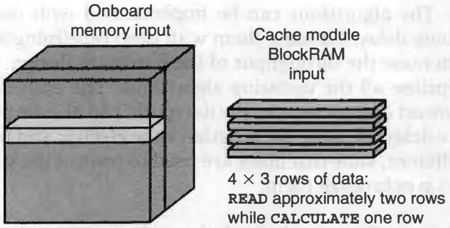


FIGURE 32.14 ■ A structural diagram of the 4×3 row caching module.

and adjacent two rows of data to calculate the current row's results. Because only one row of results is calculated from the field-updating pipelines, the READ process is longer than the CALCULATE and WRITE processes. At this point the other two processes need to wait for the READ process.

This waiting process slows down hardware design. Fortunately, we do not need to READ all three rows (45 data per cell) to start processing since the field-updating algorithm only needs part of the data in adjacent rows. We only need to READ approximately two rows of data (36 data per cell), CALCULATE one row, and WRITE one row at each time interval. Due to the limited number of BlockRAMs, the second approach is more practical. From the preceding analysis of the managed-cache modules, we conclude that the efficiency of the memory interface plays a key role in the performance of the complete FPGA design. The speed and manner in which the memory interface handles the input data often limits the speed of the entire design.

Pipelining and parallelism

Given an efficient memory interface and proper fixed-point data representations, the designer next needs to adjust the architecture and optimize design performance by considering pipelining and parallelism.

As discussed before, we can implement the electric and magnetic updating algorithms in parallel with the correct cache structure. We can also implement the three key processes—READ, CALCULATE, and WRITE—in parallel by separating the input and output memory interfaces and building dual-ported cache modules. In hardware design, parallelism translates to faster speed; however, it also "costs" more in hardware resources. The FDTD algorithm is large enough to reach the resource limits of the most advanced FPGAs on the market. One of the important problems in FDTD hardware design is determining the design architecture by considering the trade-offs between resources and performance. The hardware resource limit of each FPGA chip and computing board is different. The resource–performance trade-off analysis here is based on the targeted WildStar-II Pro FPGA computing board.

Pipelining The FDTD algorithm repeats the same electric and magnetic updating algorithms, which are independent of each other, on every cell of the model space. The algorithms can be implemented with complex combinational logic with long delay. Building them with deep pipelining helps reduce the clock cycle and increase the throughput of the hardware design. Because of the advantages, we pipeline all the updating algorithms. The embedded multipliers, which are the slowest components in the datapath, can also be pipelined to several stages to reduce delay. Because the lengths of the electric and magnetic updating pipelines are different, state machines are used to control the start and end of the pipelines and to synchronize them.

Parallelism Because the updating calculations on every cell in the FDTD model space are independent of each other, as long as there are adequate hardware resources, the computation of two or more FDTD cells can be implemented in parallel. However (see Section 32.2.3), the bandwidth of the memory interface is the bottleneck of the FDTD hardware design. The memory data width here is 3×72 bits, which can transfer six 35-bit field data inputs at each clock cycle. This memory bandwidth needs 6 clock cycles to prepare one cell's 36 input data when using the 4×3 row cache module. Can this memory interface handle the increased parallelism?

Running two cells in parallel actually saves memory bandwidth per cell. As shown in Figure 32.15, two adjacent FDTD cells share a portion of their nearest-neighbor cells. For each single cell, we need to read three rows of data (36 field data per cell) from the onboard memories, which is when running two cells in parallel, we only need to read four rows of data, or 24 data per cell. Because the bottleneck of the design is the memory bandwidth, the 2-cell parallelism mechanism improves the performance of the whole design. We can use the ratio between input data and result data as a metric to measure the efficiency of the memory interface. After implementing 2-cell parallelism, the input–result ratio decreases from 6:1 to 4:1.

Running two cells in parallel creates an extra burden on the cache size and the calculation pipelines, however. The cache module needs to hold 4×4 rows of data at the same time instead of 3×4 rows. Fortunately, the Virtex-II Pro XC2V70 FPGA has adequate BlockRAMs for the 4×4 row cache, but there is no space for increasing the cache beyond this, which is why we choose not to run three cells in parallel, even though this would further save memory bandwidth per cell and improve the input–result ratio.

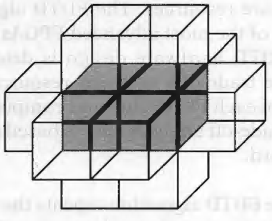


FIGURE 32.15 ■ Running two cells in parallel.

Also, the Virtex-II Pro FPGA XC2V70 does not have enough reconfigurable logic to implement all the updating pipelines in parallel. Instead, because the memory interface takes four clock cycles to transfer enough input data for one cell's calculation, the number of parallel updating pipelines can be reduced. The calculation core can run several updating algorithms serially in one updating pipeline, taking more than one clock cycle to finish the calculation for one cell. The serial calculation reduces the level of parallelism, saves reconfigurable logic, and still maintains the performance of the hardware design.

Two hardware implementations The preceding input–result ratio is calculated based on the input data needed for the uniaxial PML FDTD algorithm. This algorithm treats the whole model space as UPML cells and provides a uniform structure for both the UPML cells and the non-UPML center cells, as shown in Figure 32.16. However, the UPML FDTD algorithm requires nine extra field data for each cell in the model space, which adds overhead to the memory interface. The cells in the center of the model space that are not located in the UPML layer can be calculated by the normal FDTD algorithm, which has only six field data for each cell. Small modifications to the UPML updating pipelines can make the new updating pipelines work on both the UPML cells and non-UPML center cells.

Therefore, we can save memory bandwidth and memory space on the center cells by combining the UPML and center cell algorithms in the hardware design. The input–result ratio of a center cell is 3:1 and will be 2:1 after applying 2-cell parallelism. For the normal model space, where half the cells are center cells and the other half are UPML cells, the overall input–result ratio will decrease to approximately (4:1 + 2:1)/2 = 3:1, raising the performance of the hardware design.

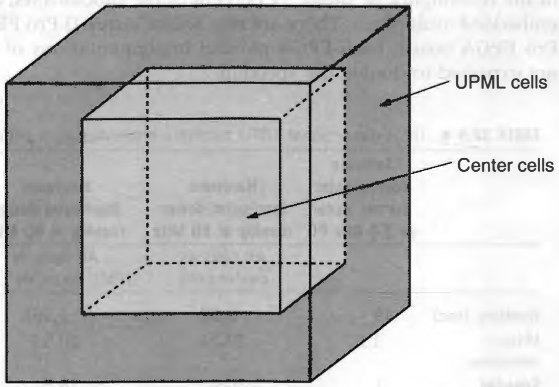


FIGURE 32.16 ■ Uniaxial PML boundary condition cells and non-uniaxial PML center cells in the model space.

We have two hardware implementations for the uniaxial PML FDTD algorithm. The first implementation treats the whole model space as UPML cells, with a simpler design structure and an input–result ratio of 4:1. The second implementation, which includes center cell and UPML cell calculations, has a more complex memory interface and better performance (the input–result ratio depends on the number of center cells and UPML cells).

The analysis of resources and performance trade-offs here is based on the WildStar-II Pro FPGA computing board. For other FPGA devices, the analysis is similar. A wider onboard memory data width, which can ease the memory bottleneck, will raise the design performance proportionally. A bigger FPGA chip, which can hold larger cache modules and more updating pipelines, will speed up the hardware design by calculating more cells in parallel.

32.2.4 Performance Results

A comparison of performance results for three-dimensional FDTD software and hardware implementations is shown in Table 32.4. The sample model is a 50×50×50 three-dimensional uniaxial PML FDTD algorithm model with 500 timesteps of FDTD iteration. The fixed-point FDTD hardware design, which treats all cells as UPML boundary cells, runs at 90 MHz on the WildStar-II Pro FPGA board. The UPML FDTD FPGA implementation is 16 times faster than the floating-point Fortran software implementation running on a 3.0-GHz PC. Hardware times are measured on the board and include the time to transfer data between the FPGA board and the host PC at the start and end of computation. The hardware design speedup can increase to 25 times with the implementation that combines the center and UPML region. The Virtex-II Pro XC2V70 FPGA chip is almost fully utilized because the FDTD hardware design occupies 99 percent of the reconfigurable slices, 51 percent of the BlockRAMs, and 46 percent of the embedded multipliers. There are two Xilinx Virtex-II Pro FPGAs on a WildStar-II Pro FPGA board. Dual-FPGA parallel implementations of the FDTD algorithm are expected to double the speedup.

TABLE 32.4 ■ Three-dimensional FDTD hardware implementation performance results

	Software floating-point Fortran code on 3.0 GHz PC	**Hardware fixed-point design running at 90 MHz**	**Hardware fixed-point design running at 90 MHz**	**Hardware fixed-point design running at 90 MHz**
		All cells as center cells	All cells as UPML boundary cells	Combined center and UPML region
Runtime (sec)	49	1.59	2.985	1.89
Million nodes/sec	1.27	39.31	20.93	33.07
Speedup	1	30.9	16.5	25.9

32.3 SUMMARY

Implementing the FDTD algorithm in hardware greatly increases its computational speed. The speedup is due to three major factors: fixed-point representation, custom memory interface design, and pipelining and parallelism. FDTD is a data-intense algorithm; the bottleneck of the hardware design is its memory interface. With the limited bandwidth between the FPGA and data memories, a carefully designed custom memory interface allows for full utilization of the memory bandwidth and greatly improves performance. The FDTD algorithm is also a computationally intense algorithm; by considering the trade-offs between resources and performance, we implement as much pipelining and parallelism as possible to speed up the design.

The FDTD algorithm is also a cellular automata, sharing a similar algorithmic structure with many other CA problems. The hardware design techniques and memory interface architecture presented in this chapter can be applied to a wide range of other CA problems to achieve speedup on an FPGA and to provide fast, small, low-power, and inexpensive implementations.

References

[1] K. S. Kunz, R. J. Luebbers. *The Finite Difference Time Domain Method for Electromagnetics*, CRC Press, 1993.

[2] A. Taflove, S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 2nd ed., Artech House, 2000.

[3] A. Taflove. *Advances in Computational Electrodynamics: The Finite-Difference Time-Domain Method*, Artech House, 1998.

[4] K. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation* 16, 1966.

[5] J. P. Berenger. Three-dimensional perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics* 127, 1996.

[6] A. Taflove. Reinventing electromagnetics: Emerging applications for FD–TD computation. *IEEE Computational Science and Engineering* 2(4), 1995.

[7] B. Yang, C. Rappaport. Response of realistic soil for GPR applications with two-dimensional FDTD. *IEEE Transactions on Geoscience and Remote Sensing*, June 2001.

[8] P. Kosmas, Y. Wang, C. Rappaport. Three-dimensional FDTD model for GPR detection of objects buried in realistic dispersive soil. *SPIE Proceedings* 4742, April 2002.

[9] P. Kosmas, C. Rappaport. Modeling with the FDTD method for microwave breast cancer detection. *IEEE Transactions on Microwave Theory and Technology* 52(8), 2004.

[10] P. Kosmas, C. Rappaport. Use of the FDTD method for time reversal: Application to microwave breast cancer detection. *SPIE Proceedings Computational Imaginary* 5299, 2004.

[11] Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2004.

[12] Annapolis Micro Systems. *WildStar-II Hardware Reference Manual*, 2004.

CHAPTER 33

EVOLVABLE FPGAs

Andres Upegui, Eduardo Sanchez
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

One of the main advantages of living beings over engineered computing systems is their capacity to adapt. While computers are tied to a fixed architecture predefined at design time, the human brain exhibits an impressive structural plasticity whereby interconnections are constantly being reinforced or destroyed according to environmental interactions. This and other comparisons between computers and living beings have given rise to what we know today as bioinspired hardware design.

Evolvable hardware is a bioinspired technique that has enjoyed impressive growth during the last decade. In 1993 Higuchi et al. and de Garis proposed an analogy between living beings and programmable hardware devices [1,2]: In both cases specification of the system is by means of a finite string of symbols. In the case of living beings, DNA determines how the organism develops into its final phenotypic representation; in programmable hardware devices, a configuration bitstream drives behavior. This parallel suggests the utilization of so-called *evolutionary algorithms* in the design of hardware systems.

33.1 THE POE MODEL OF BIOINSPIRED DESIGN METHODOLOGIES

Living organisms, from microscopic bacteria to giant sequoias, including animals such as butterflies and humans, have successfully survived on Earth for millions of years. If we had to propose but one key to explain this success, it certainly would be *adaptation*. In contrast with nature, adaptation has been very elusive to human technology. The model examples of adaptive systems are not among human's creations but among nature's—natural organisms show a striking capacity to adapt to changing circumstances, thus ensuring their continued functionality.

During the last few years, computer scientists, inspired by certain biological processes, have given birth to domains such as artificial neural networks and evolutionary computation.

Living organisms are complex systems exhibiting a range of desirable characteristics, such as evolution, adaptation, and fault tolerance, which have proved difficult to realize using traditional engineering methodologies. Such systems are characterized by a genetic program—the genome—that guides their development, their functioning, and their death. If one considers life on Earth from its very beginning, the following three levels of organization can be distinguished [3].

Phylogeny: The first level is the temporal evolution of the genetic program, the hallmark of which is the evolution of species, or *phylogeny*. The multiplication of living organisms is based on the reproduction of the program, subject to an extremely low error rate at the individual level to ensure that the species of the offspring remains unchanged. Mutation (asexual reproduction) or mutation with recombination (sexual reproduction) gives rise to new organisms. The phylogenetic mechanisms are fundamentally nondeterministic, with the mutation and recombination rate providing a major source of diversity. This diversity is indispensable for the survival of living species, for their continuous adaptation to a changing environment, and for the appearance of new species.

Ontogeny: This level constitutes the developmental process of multicellular organisms. The successive divisions of the mother cell, the zygote, into newly formed cells, each possessing a copy of the original genome, is followed by a specialization of the daughter cells in accordance with their surroundings (i.e., their position within the ensemble). This latter phase is known as cellular differentiation. The ontogenetic process is essentially deterministic: An error in a single base within the genome can provoke an ontogenetic sequence that results in notable, possibly lethal, malformations.

Epigenesis: The ontogenetic program is limited in the amount of information it can store, rendering the complete specification of the organism impossible. A well-known example is the human brain, whose some 10^{10} neurons and 10^{14} connections are far too many to be completely specified in the 4-character genome with a length of approximately 3×10^9. Therefore, when a certain level of complexity is reached, there must emerge a different process that permits the individual to integrate its vast quantity of interactions with the outside world. This is known as *epigenesis*, which primarily includes the nervous, immune, and endocrine systems. These systems are characterized by a basic structure that is entirely defined by the genome (the innate part), which is then subjected to modification through the individual's lifetime interactions with the environment (the acquired part). The epigenetic processes can be grouped under the heading of *learning* systems.

Analogous to nature, the space of bio-inspired hardware systems can be partitioned along the phylogenic, ontogenic, and epigenetic axes; we refer to this as the POE model [3, 4]. The distinction between the axes cannot be easily drawn where nature is concerned. We therefore define each axis within the model's framework as follows:

- The phylogenetic axis involves *evolution*.
- The ontogenetic axis involves the *development* of a single individual from its own genetic material, essentially without environmental interactions.
- The epigenetic axis involves *learning* through environmental interactions that take place after the individual is formed.

As an example, consider the following three paradigms, whose hardware implementations can be positioned along the POE axes:

- *P*—evolutionary algorithms are the simplified artificial counterpart of phylogeny.
- *O*—self-replicating and self-repairing cellular automata are based on the concept of ontogeny, where a single mother cell gives rise through multiple divisions to a multicellular organism.
- *E*—artificial neural networks embody the epigenetic process, where the system's synaptic weights and perhaps topological structure change through interactions with the environment.

The domains collectively referred to as soft computing [5] often involve the solution of ill-defined problems coupled with the need for continual adaptation or evolution. The paradigms listed yield impressive results, frequently rivaling those of traditional methods.

We will talk about the phylogenetic axis of hardware bio-inspired systems, most known as evolvable hardware (EHW). The scope of EHW covers diverse areas ranging from analog circuits to antenna design, but this chapter focuses on evolution of digital circuits using reconfigurable computing devices, more precisely, field-programmable gate arrays (FPGAs).

33.2 ARTIFICIAL EVOLUTION

The idea of applying the biological principle of natural evolution to artificial systems, introduced more than three decades ago, has seen impressive growth in the past few years. Usually grouped under the term *evolutionary algorithms* (EAs) or *evolutionary computation*, we find the domains of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming [6–9].

33.2.1 Genetic Algorithms

As a generic example of artificial evolution, we consider genetic algorithms (GAs) [10]. As illustrated in Figure 33.1, a GA is an iterative procedure applied to a constant-size population of individuals. Each individual represents a possible

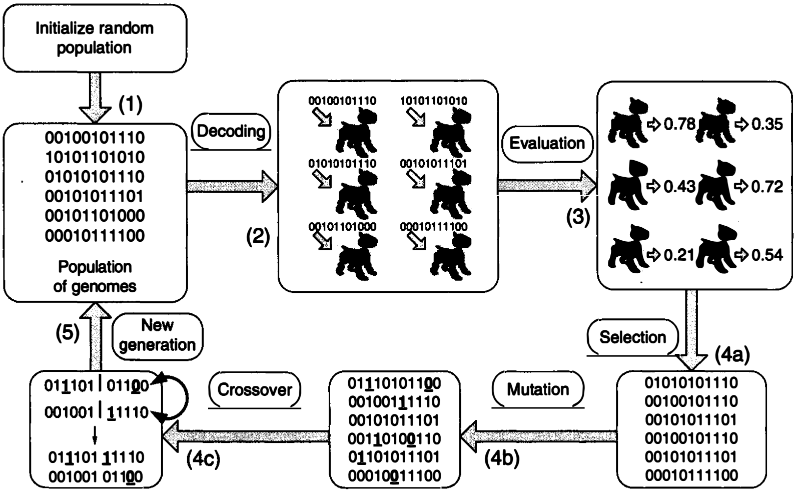


FIGURE 33.1 ■ A genetic algorithm.

solution to the given problem, and eventually one is chosen as the searched solution.

Each individual is coded by a finite string of symbols from a given alphabet, known as the *genome*. Each genome gives rise to the individual's *phenotype*, which constitutes the actual solution (a program or a circuit) to the problem at hand (e.g., a robot controller for the example in Figure 33.1). The individual receives a score (better known as fitness) depending on the performance exhibited during its evaluation. The process from the genome to a fitness value can be seen as an n-dimensional function (where n is the genome size), and the set of all possible solutions can be seen as an n-dimensional *search space*.

A GA can be summarized in the following steps:

1. *Initialization:* Create an initial population of individuals by defining a set of genomes in a random or heuristic manner.
2. *Decoding:* Generate the phenotypes for the individuals in the current population by decoding (mapping) the genotypes.
3. *Fitness evaluation:* Evaluate individuals according to some predefined quality criterion, referred to as *fitness* or *fitness function*.
4. *Genetic operators:* Apply genetically inspired operators to the current population:
    (a) *Selection:* Individuals are selected into a mating pool for reproduction according to their fitness. With stochastic or deterministic

selection mechanisms, the fittest individuals have more chances to transmit their genetic material to the next generation.

(b) *Mutation:* The genome is randomly changed; and

(c) *Crossover:* Two genomes are selected to be split and swapped at a random position.

5. If a predefined convergence condition has not been met, go back to step 2 to evaluate a new generation. Otherwise, deliver the best individual evaluated.

The basic components of GAs are always the same: a population of individuals, a decoding mechanism from a genotype to a phenotype, a fitness evaluation, genetic operators, and an iterative process. However, GAs allow variants: There exist several methods for defining each of the steps just listed. By running a large enough number of generations, the GA should eventually find an acceptable solution (i.e., one with high fitness).

EAs can be considered as a family of stochastic global optimization algorithms, mainly differing from their deterministic counterparts [11] by the lower knowledge of the problem they require and by the absence of mathematical proofs of convergence due to their stochastic nature. For highly nonlinear search spaces, EAs have exhibited faster convergence than deterministic methods, given their population-based approach. In most cases, the applications solved by EAs can also be tackled with deterministic optimization methods.

EAs are very common, having been successfully applied to numerous problems from domains as diverse as optimization, circuit design, disease diagnosis assistance, precision agriculture, self-organizing systems, automatic programming, machine learning, economics, immune systems, ecology, population genetics, studies of evolution and learning, and social systems [9].

33.3 EVOLVABLE HARDWARE

In the case of humans, adaptation due to evolution comes about through modifications in our DNA (deoxyribonucleic acid), which constitutes the encoding of every living being on Earth. DNA is a double-stranded molecule composed of two sugar-phosphate chains linked together by pairs of the bases adenine, cytocine, guanine, and thymine, constituting a string of symbols from a quaternary alphabet (A, C, G, T). Similarly, reconfigurable logic devices are configured by a string of symbols (the configuration bitstream) from a binary alphabet (0, 1). This string determines the function implemented by each of the programmable components and the connectionism of each of the switch matrices.

With this description, a rough analogy arises naturally between DNA and a configuration bitstream and between a living being and a circuit (Figure 33.2). In both cases there is a mapping from a string representation to an entity that will perform one or more actions: growing, moving, reproducing, and so forth, for living beings; computing a function for circuits.

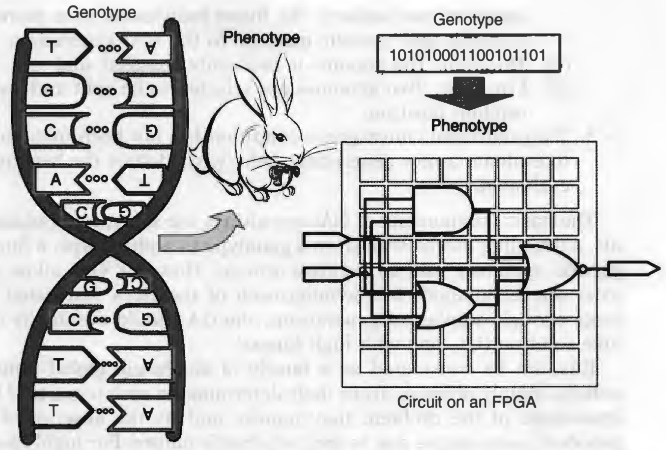


FIGURE 33.2 ■ The analogy between living beings and digital circuits.

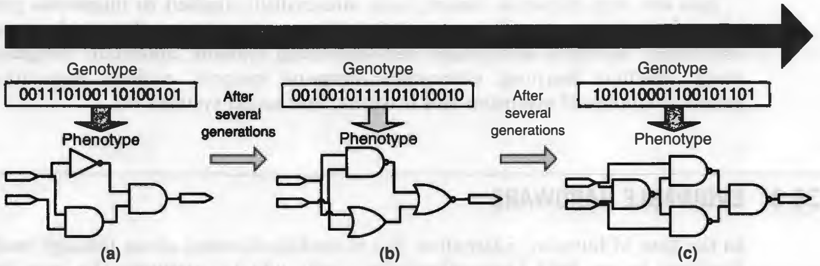


FIGURE 33.3 ■ The evolutionary design of digital circuits: (a) intial random circuit, (b) intermediate circuit, and (c) final circuit.

This analogy between living beings and digital circuits suggests the possibility of applying the principles of artificial evolution to circuit design (Figure 33.3). Designing analog and digital electrical circuits is, by tradition, a hard engineering task vulnerable to human error, and for large circuits the optimality of a solution cannot be guaranteed. Design automation has become a challenge for tool designers, and given the increasing complexity of circuits, higher abstraction levels are needed. Evolvable hardware arises as a promising solution to this problem: From a given behavior specification of a circuit, an EA will search for a bitstream describing a circuit that satisfies it.

If we carefully examine the EHW work carried out to date, it becomes evident that it mostly involves the application of EAs to the synthesis of digital systems [12–23]. From this perspective, EHW is simply a subdomain of artificial evolution, where the final goal is the synthesis of an electronic circuit. The work of Koza [8], which includes the application of genetic programming to the evolution of a 3-variable multiplexer and a 2-bit adder, may be considered an early precursor along this line. It should be noted that, in Koza's time, the main goal was to demonstrate the capabilities of the genetic programming methodology rather than to design actual circuits. We argue that the term *evolutionary circuit design* would be more descriptive of such work than the term *evolvable hardware* [24]. For now, we will stay with the latter (popular) term; however, we will return to the issue of definitions in Section 33.4.

Taken as a design methodology, EHW offers a major advantage over classical methods. The designer's job is reduced to constructing the evolutionary setup, which involves specifying the circuit requirements, the basic elements, a decoding mechanism, and the testing scheme used to assign fitness (this last phase is often the most difficult). If the setup has been well designed, evolution may then (automatically) generate the desired circuit. Currently, most evolved digital designs are suboptimal with respect to traditional methodologies; however, improved results are regularly demonstrated.

There are two critical questions to ask when setting up a system to be evolved: how to map a phenotype from a genotype and how to compute the fitness of a circuit. The answers to these questions are critical and can make the difference between a successful and an unsuccessful evolution.

33.3.1 Genome Encoding

In examining the EHW work carried out to date, we can derive a classification of current EHW in accordance with genome encoding (i.e., the circuit description) and the calculation of a circuit's fitness.

High-level languages

Using a high-level functional language to encode the evolving population implies an additional step to obtain the final circuit implementation: The chosen individual must be synthesized. Koza's evolved solution [8] was a program that described the (desired) multiplexer or adder rather than an interconnection diagram of logic elements (the actual hardware representation). Mermoud et al. [25] used fuzzy rules as evolvable components, and Murakawa et al. [26] and Upegui et al. [27] proposed the evolution of artificial neural network topologies at the neuron and layer levels. Hemmi et al. [28] used a high-level HDL to represent the genomes. Koza et al. [29] used the rewriting operator, in addition to crossover and mutation, to form a hierarchical structure.

Low-level languages

The idea of directly incorporating the bit string representing the configuration of a programmable circuit within the genome was presented early on by Atmar [30] and more recently by Higuchi et al. [1] and de Garis [2]. As a first step, a set of basic logic gates must be chosen (e.g., AND, OR, and NOT) and suitably codified, along with the interconnections between gates, to produce the genome encoding. For example, Higuchi et al. [31] used a low-level bit-string representation of the system's logic diagram to describe small-scale programmable array logics (PALs), where the circuit is restricted to a logic sum of products. The limitations of PAL circuits have been overcome to a large extent by the introduction of FPGAs, as used initially by Thompson [32,33] and later by a number of research groups.

The use of a low-level circuit description that requires no further transformation is an important step forward because it potentially enabled the placing of the genome directly into the actual circuit and thus paved the way toward true EHW (we will elaborate on this in Section 33.4). However, FPGAs presented two major problems: (1) The genome's length was on the order of tens of thousands of bits, rendering evolution practically impossible using current technology, and (2) within the circuit space, consisting of all representable circuits, many circuits were invalid.

With the introduction of the Xilinx XC6200 [34] family of FPGAs, these problems were reduced. As with previous FPGA families, there was a direct correspondence between the bit string of a cell and the actual logic circuit; however, because the XC6200 was completely multiplexer based, the result was always a viable system with no short circuits. Moreover, as opposed to previous FPGAs where the entire system had to be configured, the XC6200 family permitted the separate configuration of each cell, which was markedly faster and more flexible. Thompson [32] employed this feature to reduce the genome's size, although he did not introduce real-time, partial system reconfigurations. Unfortunately, the XC6200 was discontinued after a few years; however, the results achieved by directly evolving its bitstream led to increased visibility for the EHW community and made possible the growth of this research field.

Fitness calculation

Note the following with regard to calculations for fitness with evolvable hardware.

- *Off-chip*. The use of a high-level language for genome representation means that we have to transform the encoded system to evaluate its fitness. This is usually carried out by simulation, and only the final solution found by evolution is actually implemented in hardware.
- *On-chip*. As noted previously, the low-level genome representation enables a direct configuration (and reconfiguration) of the circuit, which leads to the possibility of using real hardware during the evolutionary process. An example of *on-chip fitness calculation* is presented in the next section in the form of an intrinsic evolvable system.

33.4 EVOLVABLE HARDWARE: A TAXONOMY

In EHW, the phylogenetic axis admits four qualitative subdivisions of evolution (Figure 33.4) according to the level of bio-inspiration: extrinsic, intrinsic, complete, and open ended.

33.4.1 Extrinsic Evolution

At the bottom of this axis, we find what is in essence *evolutionary circuit design*, where all operations are carried out in software, and the resulting solution may be loaded onto a real circuit. Though a potentially useful design methodology, this falls completely within the realm of traditional evolutionary techniques. This category is also widely known as *extrinsic* EHW.

Extrinsic EHW has typically targeted the synthesis of circuits—that is, from a desired behavior specification, an EA finds a schematic of a circuit implementing a function that satisfies the specification [29]. This category supports different levels of abstraction, allowing to evolve logical gates, arithmetic operations, more complex functional blocks, or HDL code; however, it is not suited for evolving circuits at the bitstream level. Evolution has also been used in other extrinsic aspects of circuit design such as placement and routing [35, 36] and scheduling and allocation [37].

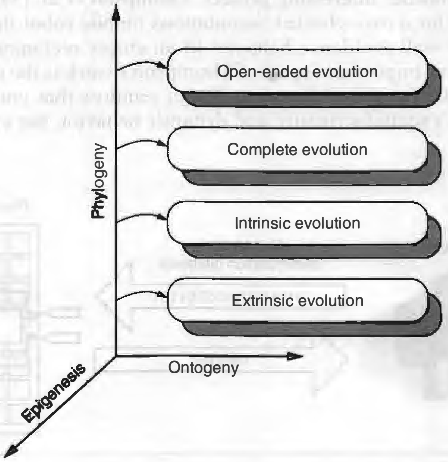


FIGURE 33.4 ■ The divisions of phylogenetic hardware.

33.4.2 Intrinsic Evolution

Moving upward along the axis, we find research in which a real circuit is used during the evolutionary process for fitness computation, although most operations are still carried out offline, in software, as depicted in Figure 33.5.

The very first intrinsic evolution was reported by Thompson [32]. He evolved a section of an XC6216 FPGA, consisting of 10×10 cells (the full array size was 64 × 64), to discriminate between square waves of 1 kHz and 10 kHz presented as inputs. His complete system setup is depicted in Figure 33.6 (see Thompson [33]). From a PC, he configured the FPGA with a configuration bitstream generated by a GA, which used a genome of 1800 bits (18 configuration bits per cell) to represent a possible circuit. Then the individual's fitness was automatically evaluated as follows:

1. The tone generator, driven by the PC, presented five bursts each of both waves (1 kHz and 10 kHz) to the circuit. The analog integrator was reset before the generation of each burst, and it then integrated the circuit's output during the presentation of the burst.
2. Back in the PC, the individual's fitness was computed by a function aiming to maximize the difference between the average output voltages when presenting both waves.
3. After running the experiment for 2 to 3 weeks, during which 5000 generations of 50 individuals were evaluated, the resulting circuit achieved successful discrimination of the waves. However, the perfect desired behavior was obtained around generation 4100.

In another interesting project, Thompson et al. [38] evolved a hardware controller for a two-wheeled autonomous mobile robot that was required to display simple wall avoidance behavior in an empty rectangular arena.

A very important aspect of Thompson's work is the unconstrained use of hardware. Conventional (human) design requires that constraints be applied to the circuit's spatial structure and dynamic behavior, but evolution can do away with

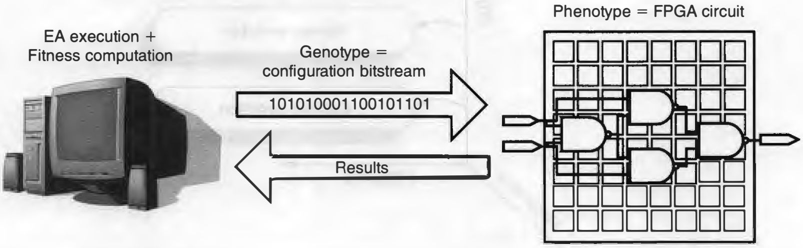


FIGURE 33.5 ■ Intrinsic evolution.

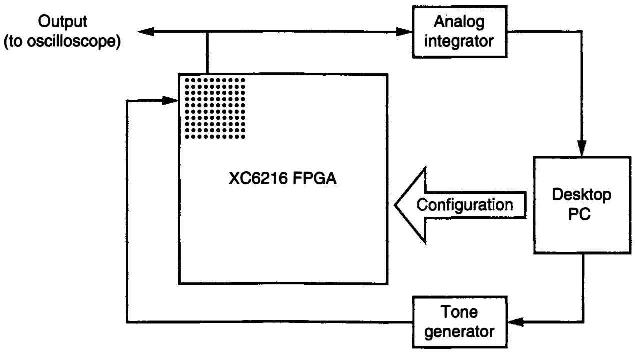


FIGURE 33.6 ■ Adrian Thompson's intrinsic evolvable system setup.

these. The circuits evolved by Thompson [33, 38] and Ly and Mowchenko [37] had no enforced spatial structure (e.g., limitations on recurrent connections), no impositions upon modularity, and no dynamic constraints such as a synchronizing clock or handshaking between modules. Unconstrained circuit design can better exploit the dynamics of the circuit supporting it; however, such circuits exhibit two main drawbacks. One is the impossibility of reproducing a solution: The same bitstream does not behave in the same manner in two different devices. The other is the circuit's high sensitivity to external conditions: Slight temperature changes can modify its behavior.

Two more examples from this subdivision of the phylogenetic axis are the works of Murakawa et al. [39] and Iwata et al. [40]. One of the major obstacles these researchers hoped to overcome was large genome size (defining the FPGA's full configuration). They suggested two solutions:

1. Variable-length chromosome GAs (VGA), where the genome does not directly represent the configuration bit string but rather codifies the possible logical operations and interconnections [40].
2. Evolution at the function level, where the basic units are not elementary logic gates (e.g., AND, OR, and NOT) but rather higher-level functions (e.g., sine-wave generator, multiplier) [39].

Because no such commercial FPGA currently exists, Murakawa and Iwata and their colleagues proposed a novel architecture, dubbed F^{2}PGA (function-based FPGA).

It is important to note that while experiments of the above type have been referred to by some as intrinsic evolution, they have a prominent extrinsic aspect because the population is stored in an external computer, which also controls the evolutionary process.

33.4.3 Complete Evolution

Still further along the phylogenetic axis, we find systems in which all operations (selection, crossover, mutation), as well as fitness evaluation, are carried out intrinsically, in hardware (Figure 33.7). This category, called *complete* evolution by Haddow and Tufte [41], has as its main motivation attaining adaptive systems that are able to accomplish difficult tasks, possibly involving real-time behavior in a complex, dynamic environment. The major aspect missing here, compared with biological evolution, is that the evolution is not open ended (i.e., there is a predefined goal and no dynamic environment to speak of).

Within the category of complete evolution, we find two subdivisions: *centralized* and *population oriented.*

Centralized evolution

The main characteristic of centralized evolution is the existence of a single evolvable circuit and a single evolvable algorithm computation (Figure 33.7(a)). With this approach an on-chip genetic machine, a hardwired EA, is implemented. The approach also comprises implementations where the EA is executed in an on-chip processor. Centralized evolution holds special interest because it greatly enhances the autonomy of the circuit, allowing the EHW to adapt to a changing environment during its lifetime. Implementations of EAs in general-purpose processors, in spite of their lower performance compared to their fully hardwired counterparts, exhibit several important advantages that permit them to benefit from a more general framework: They provide a more user-friendly interface for implementing chromosome manipulations, fitness evaluations, and memory access; they support easier algorithm upgrades; and they enhance the possibilities of immediately using the evolving circuit for useful computations.

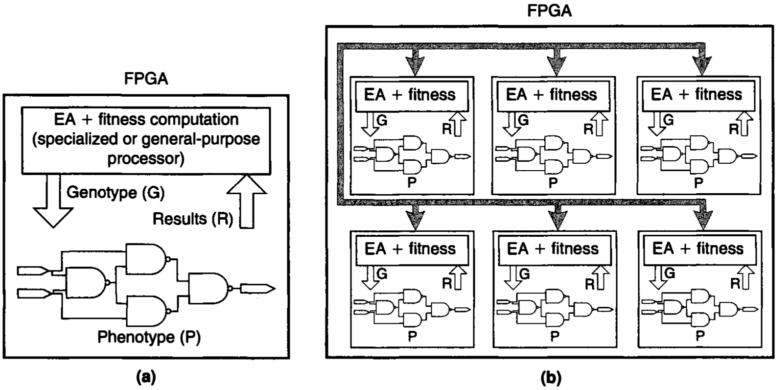

FPGA
EA + fitness computation (specialized or general-purpose processor)
Genotype (G)
Results (R)
Phenotype (P)
(a)
FPGA
EA + fitness
G
R
P
(b)

FIGURE 33.7 ■ Complete evolution: centralized (a) and population oriented (b).

One example of a self-reconfigurable platform that performs online and on-chip evolution is that of Upegui and Sanchez [42, 43]. Their standalone platform consists of a MicroBlaze processor with memory access control, ICAP (internal configuration access port) access, and a reconfigurable evolvable section, as depicted in Figure 33.8. The full system, implemented in a Virtex-II FPGA, runs an EA on the MicroBlaze processor, reads a section of the configuration bitstream through the ICAP, modifies the bitstream according to the genome currently evaluated in the MicroBlaze, sends back the bitstream though the ICAP for partially reconfiguring the FPGA, and evaluates the fitness of the current individual by interacting with the reconfigurable evolvable section through the standard OPB bus. Upegui and Sanchez [42] evolve nonuniform cellular rules and FPGA lookup table (LUT) configurations with fixed interconnectivity. In Upegui and Sanchez [43], Boolean networks are evolved as well, but in this case the interconnectivity is not fixed, so the system topology is also driven by the evolutionary algorithm.

Other interesting experiments were carried out by Haddow and Tufte [41] in which a hardware implementation of a GA, the "GA pipeline," evolves a robot controller. Glette and Torresen [44] report the implementation of a GA on an embedded PowerPC processor in a Virtex-II Pro FPGA that evolves a circuit in the same FPGA.

Population-oriented evolution

A hardware implementation of the *full population*, not only of one individual (as was the case in previous categories), is the distinctive feature of the population-oriented approach (Figure 33.7(b)). A significant example is the work of Goeke et al. [45], where an evolving cellular system was implemented in which evolution takes place completely on-chip. This system is based on the cellular automata model—a discrete dynamic system that performs computations in

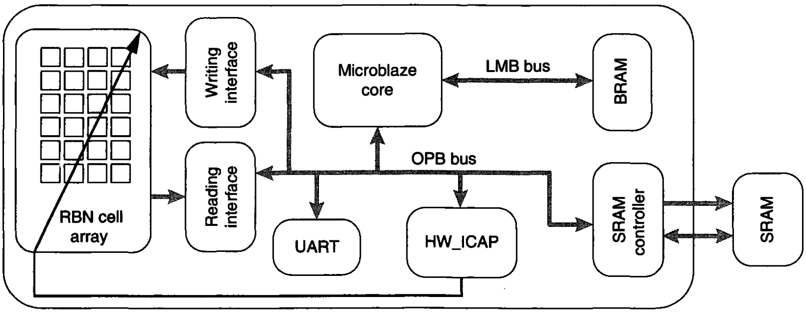


FIGURE 33.8 ■ The setup of a complete and centralized self-reconfigurable evolvable platform.

a distributed fashion on a spatially extended grid. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete timesteps according to a *local, identical* interaction rule [46]. The *state* of a cell at the next timestep is determined by the current state of a surrounding neighborhood of cells. This transition is usually specified in the form of a *rule table*, which delineates the cell's next state for each possible neighborhood configuration. The cellular array (grid) is n-dimensional, where typically $n = 1, 2, 3$. Nonuniform cellular automata have also been considered in which the local update rule need not be identical for all grid cells [47].

Based on the *cellular programming* EA of Sipper [47], Goeke et al. [45] implemented an evolving, one-dimensional, nonuniform cellular automaton. The main feature of the cellular programming algorithm is the fact that genetic operators are computed in a distributed way: Each automaton modifies its own rule based on its own and its neighbors' fitness. Each of the system's 56 binary-state cells contains a genome that represents its rule table. These genomes are initialized at random and then are subjected to evolution.

The environment imposed on the system specifies the resolution of a global synchronization task: On presentation of a random initial configuration of cellular states, the system must reach, after a bounded number of timesteps, a configuration for which the states of the cells oscillate between all zeros and all ones on successive timesteps. This may be compared to a swarm of fireflies, thousands of which may flash on and off in unison, having started from totally uncoordinated flickerings. Each insect has its own rhythm, which changes only through local interactions with its neighbors'. Because of the local connectivity of the system, this global behavior, which involves the entire grid, makes for a difficult task. Nonetheless, applying the evolutionary process of Sipper [47], the system evolves (i.e., the genomes change) such that the task is completed.

The evolving cellular system described here exhibits complete on-chip evolution in that all operations are performed in hardware in a distributed population-based manner with no reference to an external computer.

33.4.4 Open-ended Evolution

The last subdivision, situated at the top of the phylogenetic axis, involves a population of hardware entities evolving in an open-ended environment. When the fitness criterion is imposed by the user in accordance with the task to be performed (currently the rule with artificial evolution techniques), we attain a form of guided, or directed, evolution. This is to be contrasted with the open-ended evolution that occurs in nature, which admits no externally imposed fitness criterion but rather an implicit, emergent, dynamic one (which can arguably be summed up as reproducibility). Open-ended undirected evolution is the only form of evolution known to produce such devices as eyes, wings, and nervous systems and to give rise to the formation of species. Undirectedness may have to be applied to artificial evolution if we want to observe the emergence of completely novel systems.

We argue that only open-ended evolution can be truly considered EHW, which is still an elusive goal at present. We point out that a more correct term would probably be *evolving hardware*. A natural application area for such systems is the field of autonomous robots—that is, machines capable of operating in unknown environments without human intervention [48]. Specifically, collective robotics exhibits a population of individuals interacting in a common environment, in which they can learn to cooperate or to compete for achieving their goals [49]. In their interactions the individuals exhibit a high level of emergence as a first step to open endedness. Modular robotics, a subtype of collective robotics, also offers a promising open-ended real environment.

A modular robotic platform well suited for evolving distributed hardware is YaMoR. This is a modular robot composed of mechanically homogeneous modules [50], each of which contains an FPGA-based system that allows wireless FPGA configuration and on-board self-reconfiguration. Another interesting example is what we call Hard-Tierra. This involves the hardware implementation (e.g., FPGA circuits) of the Tierra "world," which consists of an open-ended environment of evolving computer programs [51]. Hard-Tierra is important because it demonstrates that open-endedness does not necessarily imply a real, biological environment.

33.5 EVOLVABLE HARDWARE DIGITAL PLATFORMS

The hardware substrate that supports evolution is one of the most important initial decisions to make when evolving hardware. The hardware architecture is closely related to the type of solution being evolved. Hardware platforms usually have a cellular structure composed of uniform or nonuniform components. In some cases, we can evolve the components' functionality; in others, the connectivity; or sometimes both, with the most powerful ones. FPGAs fit well into this third category because they are composed of configurable logic elements interconnected by configurable switch matrices. FPGA configuration is contained in a configuration bitstream, which holds every function and switch position to be configured for implementing a given design. Current FPGAs allow the processing of partial bitstreams, reconfiguring just a sector of the FPGA while the remaining logic stays the same.

When evolving a circuit on an FPGA, we consider the logic cell as the basic element. The logic cells' configuration and their interconnectivity are defined by the evolution. However, this implies a huge search space to explore and can prevent the EA from finding a solution. A common technique to constrain the search space is to define a basic block as a set of logic cells. In this way each basic block can be an artificial neuron, a fuzzy rule, or a more complex cell in general. Another option is to constrain the connectionism, using layered architectures, to a certain neighborhood, or by just defining it as fixed.

The most basic requirement when evolving hardware is to have a set of high- or low-level evolvable components and a hardware substrate supporting them.

These evolvable components are the basic elements from which the evolved circuits will be built (transistors, logic gates, arithmetic functions, functional cells, etc.), and the evolvable substrate must be a flexible hardware platform that allows arbitrary configurations mapped from a genome. FPGAs constitute the perfect hardware substrate, given their connectivity and functional flexibility. The evolvable substrate can be implemented using one of two main techniques: (1) exploiting the flexibility provided by the FPGA's configuration logic and (2) building a virtual flexible substrate on top of the logic.

In the first approach the configuration bitstream of the FPGA is directly generated. In this way, we can make better use of FPGA resources—logic functions are directly mapped into the FPGAs LUTs, and connections are directly mapped to routing switch matrices and multiplexers—but the penalty is very low-level circuit descriptions [33, 38, 52]. In the second approach a virtual reconfigurable circuit is built on top of the actual circuit [53]. In this way the designer can also define the configuration bitstream and determine which features of the circuit to evolve. This approach has been widely used by several groups, as it produces enhanced flexibility and ease of implementation. The penalty here is the cost of an inefficient use of logic resources [25, 27, 42, 45, 53–60].

Different custom chips have been proposed for this purpose with very interesting results: The main interest in proposing an architecture is that commercial FPGAs are designed for general-purpose applications, so they do not necessarily fit the requirements for evolvable architectures. For example, commercial devices may have illegal configurations that cause short circuits; this is reasonable for standard FPGA users who rely on the CAD flow to create the design, but it can be disastrous for genetically evolved bitstreams. Custom evolvable chips generally provide dynamic and partial reconfiguration, contain multi-context configuration memories, and can be configured with arbitrary bitstreams. However, although the custom chips are better suited to EHW applications, the commodity devices benefit from economies of scale and access to more advanced fabrication processes.

Different chips and platforms have been developed to provide the flexibility necessary for evolving analog, digital, and mixed circuits; some of them have been designed specifically for EHW, while for others EHW is just another application field. Among them we find different levels of granularity, different types of reconfiguration including dynamic and static reconfigurations, and the possibility of loading partial configuration bitstreams, and the utilization of context memories.

33.5.1 Xilinx XC6200 Family

The obsolete Xilinx XC6200 family [61] deserves a special mention in a discussion of EHW platforms. For several years, the XC6200 family constituted the perfect platform for intrinsic EHW, because it made possible downloading any arbitrary bitstream without risking contention given its multiplexer-based connection architecture. It also allowed dynamic reconfiguration, making it more flexible for adaptive algorithms in a general sense. The results reported by Thompson [32, 33, 38, 62], discussed previously, are a very good example of the XC6200's potential for evolving circuits.

The XC6200 represents an important initial stepping-stone in the EHW field. It has also been used for implementing several types of applications, among them cooperative robot controllers [63], sorting networks [64], and image-processing algorithms [65].

33.5.2 Evolution on Commercial FPGAs

After the XC6200 disappeared, many research groups turned to the Xilinx XC4000 family. However, these FPGAs had an important drawback for evolving hardware: They were not partially reconfigurable, and no arbitrary bitstreams were allowed. When the Virtex FPGAs appeared, they exhibited two well-appreciated features for the EHW community: partial and dynamic reconfiguration. However, not all the evolution-friendly features from the XC6200 were kept. Specifically, the connection mechanism does not support arbitrary bitstreams, making these FPGAs susceptible to damage by internal short circuits.

Recent work on evolvable circuits in commercial FPGAs has focused on the Virtex and Virtex-II architectures from Xilinx [66] and will extend its focus to Virtex-4 in the near future. Two main approaches have been used for evolving Virtex circuits: using virtual reconfigurable circuits [67] and partially reconfiguring the FPGA.

Virtual reconfiguration

Two solutions were used in order to replace the obsolete XC6200 family: implementing an ASIC evolvable circuit (only achievable by some privileged groups, summarized in Section 33.5.3) and building a reconfigurable circuit on top of another reconfigurable circuit (i.e., a virtual reconfigurable device [53]). The concept of a virtual reconfigurable circuit is depicted in Figure 33.9, where a reconfigurable neuron cell constitutes the device's basic logic cell.

In the beginning, the most intuitive method was to reconstruct the XC6200 architecture. At the University of York, a virtual XC6200 CLB was implemented in Virtex FPGAs [68, 69]. Slorach and Sharman [54] also used virtual XC6200 cells in the Xilinx XC4010 and Altera EPF6010A, evolving configuration bitstreams that configured not the FPGA itself but the virtual XC6200 CLBs. Afterward, other research groups developed different reconfigurable architectures with enhanced features, several of which had the goals of flexibility and easy reconfiguration [54–59, 70–72]. For example, Sekanina and Drabek [70] developed a virtual reconfigurable cell called a *functional block* (FB) and used an array of FBs for image compression. Durbeck and Macias [71] implemented an 8 × 8 cell matrix using a Xilinx Spartan-2 FPGA.

With this approach came the possibility of designing any desired reconfigurable fabric. In most cases the architecture consists of a fine-grained cellular array in which a general-purpose evolvable architecture is proposed. However,

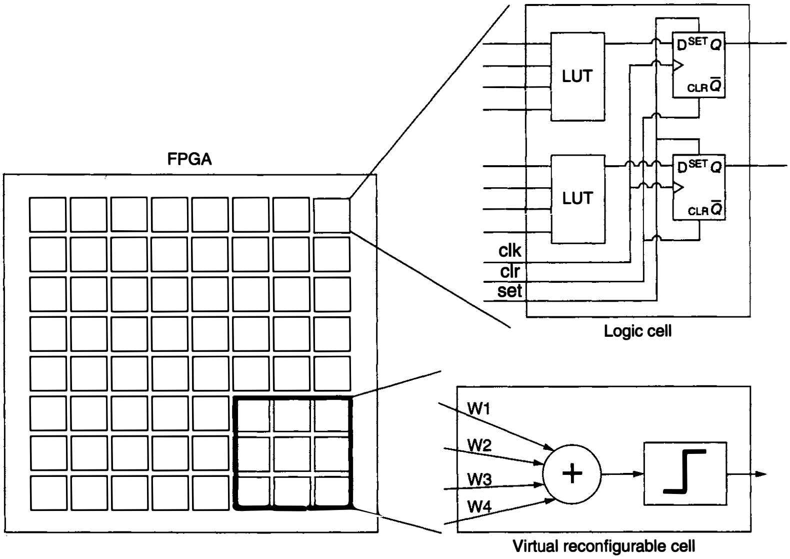


FIGURE 33.9 ■ A virtual reconfigurable circuit with a reconfigurable neuron.

problem-oriented reconfigurable fabrics can use coarser-grained architectures, where a reduced set of features is evolved.

Dynamic partial reconfiguration

In addition to the Xilinx XC6200, other commercial platforms have been partially reconfigured for evolving circuits, with the main focus on the Xilinx Virtex families. However, there are two main issues in evolving circuits by partially reconfiguring Virtex architectures. The first is the size of their configuration bitstreams, which implies a huge search space for the EA. The second is the generation of invalid bitstreams—that is, bitstreams that cause internal contentions. Different solutions to these problems have been suggested.

Haddow and Tufte proposed a two-dimensional array of Sblocks [72], each containing a flip-flop, a 5-input LUT, and some routing resources. Sblocks provide a reduced configurability compared to Virtex cells in order to reduce the search space size and to guarantee contention-free configurations. Even though the Sblock array is virtually reconfigurable, the functionality is reconfigured by partially reconfiguring a Virtex FPGA. Haddow and Tufte used a partial bitstream for reconfiguring only the LUT contents.

At the University of York, JBits [73] has been used for evolving circuits. JBits is a Java API for describing circuits and manipulating configuration bitstreams. It allows safe generation of partial bitstreams, permitting the modification of internal modules in the FPGA design. At York, LUT contents have been mapped from a genome for evolving simple combinatorial functions [74], fault tolerance circuits [69], and robot controllers for obstacle avoidance [75]. Also using JBits, Levi and Guccione from Xilinx developed a tool called GeneticFPGA [76], which translates a configuration bitstream from a chromosome, making it easy to generate legal bitstreams.

Even though JBits provides interesting features for EHW, it has several limitations, such as the impossibility of running on an embedded platform (for on-chip evolution), dependence on supported FPGA families and supported boards, incompatibility with other hardware description languages (HDLs), and limited support from Xilinx, mainly reflected in insufficient documentation.

Several ways to overcome these limitations have been proposed at the EPFL. Upegui and Sanchez [52] summarize three techniques for EHW by partially reconfiguring Virtex and Virtex-II families dynamically, without using JBits. The first is a coarse-grained high-level solution based on the modular partial reconfiguration flow proposed by Xilinx [77]. It defines large evolvable functions, implemented as modules, that are well suited for architecture exploration [27].

The second and third techniques are fine-grained low-level solutions. In both of the cases, hard-macros are used to define an evolvable component. Then by placing the hard-macros they modify, the bitstream partially reconfigures components of the hard macros. The second technique uses the difference-based partial reconfiguration flow proposed by Xilinx [77]. The third technique directly manipulates the bitstream in a manner similar to the XC6200, by adding some constraints (only LUT and multiplexer configuration modifications are allowed). These techniques are well suited for fine-tuning. With the difference-based approach, Mermoud et al. [25] report the intrinsic evolution of a fuzzy classifier; and with the bitstream manipulation, they report a complete evolution of cellular automata [42] and Boolean networks [43].

33.5.3 Custom Evolvable FPGAs

One of the more recent evolvable chips is the POEtic tissue [78, 79], a computational substrate optimized for the implementation of digital systems inspired by the POE model presented in the introduction to this chapter. The POEtic tissue is a self-contained, flexible physical substrate designed (1) to interact with the environment through spatially distributed sensors and actuators; (2) to develop and adapt its functionality through a process of evolution, growth, and learning to a dynamic and partially unpredictable environment; and (3) to self-repair parts damaged by aging or environmental factors in order to remain viable and retain the same functionality.

The POEtic tissue is composed of a two-dimensional array of POEtic cells, each designed as a 3-layer structure following the three axes of bio-inspiration (Figure 33.10):

- The phylogenetic layer acts on a cell's genetic material. It can be used to find and select the genes of the cells for the genotype layer, which is conceptually the simplest of the three tissue layers as it is mainly a memory containing the genetic information of the organism.
- Ontogeny concerns the development of the individual and thus the mapping or configuration layer of the cell, which implements cellular differentiation and growth. In addition, it has an impact on the system as a whole for self-repair. The configuration layer selects which gene will be expressed depending on a user-defined differentiation algorithm.
- The epigenetic axis modifies the behavior of the organism during its operation and is therefore best applied to the phenotype, which is probably the most application-dependent layer. If the final application is a neural network, the phenotype layer will consist of an artificial neuron.

A key aspect of the applicability of the POEtic tissue, in addition to its architecture, is its reconfigurability. A molecule can be partially reconfigured by an on-chip microprocessor or by neighbor molecules. For EHW, this feature is

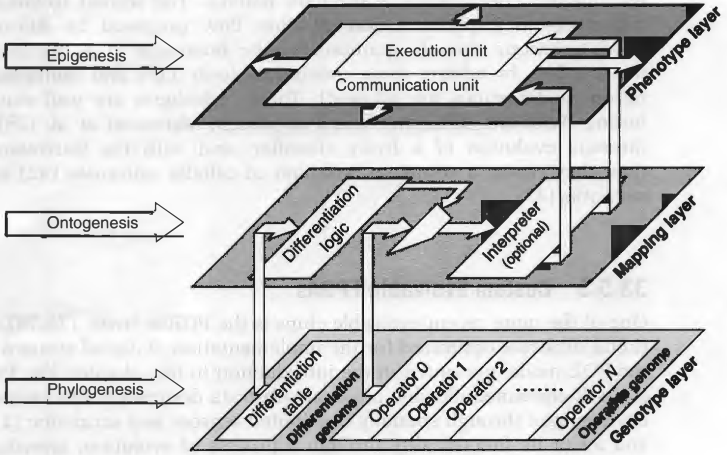


FIGURE 33.10 ■ The organizational layers of the POEtic cell.

very important in terms of execution time. Because only two clock cycles are needed for a write, and three words of 32 bits define a complete molecule, the configuration of the entire array (or a part of it) is very fast. In comparison with commercial FPGAs, such as the Virtex-II, in which at least a full configuration frame must be sent each time, reconfiguration takes place in parallel, allowing a huge speedup.

A distinctive feature of the POEtic tissue is its two-dimensional array of routing units that implement a dynamic routing algorithm [80]. It is used for intercellular communication, allowing the tissue to dynamically create paths between cells. The dynamic routing can be performed by a distributed algorithm [80] or by the on-chip processor.

Another very important circuit is the evolvable LSI chip developed by Higuchi's group [81]. It includes a GA unit and has the ability to process two chromosomes in parallel. Higuchi's group is famous for the large number of applications implemented in their chips [82, 83]. They have implemented an adaptive prosthetic hand controller [84, 85] that can adapt to the user's electromyographic signals in less than 10 minutes with a much more compact circuit than required with a neural network (before that, the user had to adapt to the hand instead of the hand to the user, requiring more than a month of training). They have also evolved data compressors for electrophotographic printing [86, 87], often attaining compression ratios twice those obtained with international standard compression algorithms such as Lempel-Ziv, JBIG, and JBIG2. It must be noted that Higuchi's applications often finish as part of a commercial product. Other interesting applications implemented by the same group include robot navigation controllers [88] and low-power integrated circuits [89].

This chapter focused primarily on evolution for digital devices; however, several platforms have been proposed for analog and mixed-signal circuit evolution. At the Jet Propulsion Laboratory of the California Institute of Technology, a field-programmable transistor array (FPTA) [90] has been developed that is the basis of the Standalone Board-level Evolvable System (SABLES) [91]. Layzell [92] proposed the evolvable motherboard: a diagonal matrix of analog switches connected to up to six plug-in daughter boards, which contain the desired basic elements for evolution.

33.6 CONCLUSIONS AND FUTURE DIRECTIONS

EHW has been shown to be effective at finding solutions [82, 83] for real-world applications. Additionally, some solutions have proven to perform better than their engineered counterparts [83, 89, 93]. On the other hand, EHW generally performs poorly, as a system-level solution: Microprocessor architectures, for example, are not among evolution results. As a matter of fact, evolution works better when the target is a complex cellular architecture: cellular automata, neural networks, or gate arrays.

If we look at the EHW work carried so far, we find many common characteristics spanning most current systems that often differ from biological evolution (this difference is not necessarily disparaging):

- Evolution pursues a predefined goal: The design of an electronic circuit is subject to precise specifications. On finding the desired circuit, the evolutionary process terminates.
- The population has no material existence. At best, in what has been called intrinsic and complete evolution, there is one circuit available onto which individuals from the population are loaded *one at a time* to evaluate their fitness.
- The absence of a real population in which individuals coexist simultaneously entails notable difficulties in the realization of interactions between "organisms." This usually results in a completely independent fitness calculation, contrary to nature, which exhibits a coevolutionary scenario.
- The different phases of evolution are carried out sequentially, controlled by a central unit.

These limitations suggest that the simple application of EAs to hardware design is not enough and that future research in EHW must not be limited to exploration of architectures and substrates; there is also much to do at the algorithmic level. Human-made adaptable systems are still far from exhibiting an adaptation comparable to living beings, and even though we have yet to attain circuits of equivalent complexity, limitations are not just a matter of magnitude. Only by modeling together the three axes of life (phylogeny, ontogeny, and epigenesis) will we be able to build systems featuring naturelike adaptation.

Future trends in nanotechnology are also guiding us toward "Avogadro computers"—that is, massively parallel devices with 10^{23} transistors. What to do with such huge number of transistors, and how to use, interconnect, and program them, goes beyond present engineering knowledge; however, EHW architectures and algorithms arise as a promising solution for dealing with the design complexity of these machines.

In this chapter we focused on evolving silicon circuits, which constitute the main developments achieved by the EHW community. However, other types of substrates have been evolved that extend the domain and represent new directions for evolvable hardware. For example, NASA researchers have been working on evolving antennas for space missions [94, 95]. Miller and Downing are currently working on evolving liquid crystals (LC) [96]—by applying electric fields mapped from a genome, they modify the LC molecular alignment to implement a desired function. Molecular circuit design is another promising evolvable substrate. Masiero et al. [97] report the use of a GA for tuning component parameters in a molecular circuit. Quantum circuit synthesis, too, is a potential field for EHW [98], given that designing circuits in such a substrate will require new design paradigms.

References

[1] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, T. Furuya. Evolving hardware with genetic learning: A first step towards building a Darwin Machine. From animals to animals 2. *Proceedings of the International Conference on Simulation of Adaptive Behavior*, 1993.

[2] H. de Garis. Evolvable hardware: Genetic programming of a Darwin Machine. *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, 1993.

[3] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Perez-Uribe, A. Stauffer. Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. *Evolvable Systems: From Biology to Hardware, LNCS* 1259, 1997.

[4] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Perez-Uribe, A. Stauffer. A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Transactions on Evolutionary Computation* 1(1), 1997.

[5] S. Mitra, Y. Hayashi. Neuro-fuzzy rule generation: Survey in soft computing framework. *IEEE Transactions on Neural Networks* 11(3), 2000.

[6] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, 1996.

[7] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, 2nd ed., IEEE Press, 2000.

[8] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[9] M. Mitchell. *An Introduction to Genetic Algorithms*, MIT Press, 1996.

[10] M. D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*, MIT Press, 1999.

[11] J. Pinter. *Global Optimization in Action (Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications)*, Kluwer Academic Press, 1996.

[12] E. Sanchez, M. Tomassini. Towards evolvable hardware. *LNCS* 1062. Springer-Verlag, 1996.

[13] Y. Liu. Evolvable systems: from biology to hardware. *Proceedings of the Fourth International Conference, ICES*, October 2001.

[14] A. M. Tyrrell, P. C. Haddow, J. Torresen. Evolvable systems: From biology to hardware. *Proceedings of the 5th International Conference, LNCS*, March 2003.

[15] J. M. Moreno, J. Madrenas, J. Cosp. Evolvable systems: From biology to hardware. *Proceedings of the Sixth International Conference, ICES 2005*, September 2005.

[16] T. Higuchi, M. Iwata, W. Liu. Evolvable systems: From biology to hardware. *Proceedings of the First International Conference*, October 7–8, 1996. *LNCS* 1259, Heidelberg: Springer-Verlag, 1997.

[17] M. Sipper, D. Mange, A. Pérez-Uribe. Evolvable systems: From biology to hardware. *Proceedings of the Second International Conference*, September, *LNCS* 1478, Heidelberg: Springer, 1998.

[18] J. Miller. Evolvable systems: From biology to hardware. *Proceedings of the Third International Conference, ICES 2000*, April 17–19, 2000. *LNCS* 1801, Heidelberg: Springer, 2000.

[19] A. Stoica, D. Keymeulen, J. D. Lohn. *Proceedings of the First NASA/DOD Workshop on Evolvable Hardware*, July. IEEE Computer Society, 1999.

[20] A. Stoica, J. D. Lohn, R. Katz, D. Keymeulen, R. Zebulum. *Proceedings of the 2002 NASA/DOD Conference on Evolvable Hardware,* July. IEEE Computer Society, 2002.

[21] J. D. Lohn, R. Zebulum, J. Steincamp, D. Keymeulen, A. Stoica, M. Ferguson. *Proceedings of the 2003 NASA/DOD Conference on Evolvable Hardware,* July. IEEE Computer Society, 2003.

[22] R. Zebulum, D. Gwaltney, G. Hornby, D. Keymeulen, J. D. Lohn. A. Stoica. *Proceedings of the 2004 NASA/DOD Conference on Evolvable Hardware,* July 2004. IEEE Computer Society.

[23] J. D. Lohn, D. Gwaltney, G. Hornby, R. Zebulum, D. Keymeulen. A. Stoica. *Proceedings of the 2005 NASA/DOD Conference on Evolvable Hardware,* June 2005. IEEE Computer Society.

[24] X. Yao, T. Higuchi. Promises and challenges of evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 29(1), 1999.

[25] G. Mermoud, A. Upegui, C. A. Pena. E. Sanchez. A dynamically-reconfigurable FPGA platform for evolving fuzzy systems. *Computational Intelligence and Bioinspired Systems, LNCS* 3512, 2005.

[26] M. Murakawa, S. Yoshizawa, I. Kajitani, X. Yao, N. Kajihara, M. Iwata, T. Higuchi. The GRD chip: Genetic reconfiguration of DSPs for neural network processing. *IEEE Transactions on Computers* 48(6), 1999.

[27] A. Upegui, C. A. Peña-Reyes, E. Sanchez. An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and Microsystems* 29(5), 2005.

[28] H. Hemmi, J. Mizoguchi, K. Shimohara. Development and evolution of hardware behaviors. *Towards Evolvable Hardware, LNCS* 1062, 1996.

[29] J. R. Koza, F. H. Bennett, D. Andre, M. A. Keane. Synthesis of topology and sizing of analog electrical circuits by means of genetic programming. *Computer Methods in Applied Mechanics and Engineering* 186(2), 2000.

[30] J. W. Atmar. Speculation on the Evolution of Intelligence and Its Possible Realization in Machine Form, Ph.D. dissertation, New Mexico State University, Las Cruces, 1976.

[31] T. Higuchi, M. Iwata, I. Kajitani, H. Iba, Y. Hirao, F. T. Furuya, B. Manderick. Evolvable hardware and its application to pattern recognition and fault-tolerant systems. *Towards Evolvable Hardware, LNCS* 1062, 1996.

[32] A. Thompson. Silicon evolution. *Proceedings of Genetic Programming,* J. R. Koza et al. (eds.), MIT Press, 1996.

[33] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. *Evolvable Systems: From Biology to Hardware, LNCS* 1259, 1997.

[34] Xilinx, Inc. *The Programmable Logic Data Book*, 1996.

[35] G. K. Venayagamoorthy, V. G. Gudise. Swarm intelligence for digital circuits implementation on field-programmable gate array platforms. *Proceedings of the 2004 NASA/DOD Conference on Evolvable Hardware,* July 2004.

[36] B. C. Kahne. *A Genetic Algorithm-Based Place-and-Route Compiler for a Run-time Reconfigurable Computing System,* Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1997.

[37] T. A. Ly, J. T. Mowchenko. Applying simulated evolution to high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 12(3), 1993.

[38] A. Thompson, I. Harvey, P. Husbands. Unconstrained evolution and hard consequences. *Towards Evolvable Hardware, LNCS,* 1996.

[39] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, T. Higuchi. Hardware evolution at function level. *Parallel Problem Solving from Nature (PPSN IV), LNCS* 1141, 1996.

[40] M. Iwata, I. Kajitani, H. Yamada, H. Iba, T. Higuchi. A pattern recognition system using evolvable hardware. *Parallel Problem Solving from Nature (PPSN IV), LNCS* 1141, 1996.

[41] P. Haddow, G. Tufte. Evolving a robot controller in hardware. *Proceedings of the Norwegian Computer Science Conference*, 1999.

[42] A. Upegui, E. Sanchez. On-chip and on-line self-reconfigurable adaptable platform: The non-uniform cellular automata case. *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

[43] A. Upegui, E. Sanchez. Evolving hardware with self-reconfigurable connectivity in Xilinx FPGAs. *Proceedings of the First NASA/ESA Conference on Adaptive Hardware and Systems*, 2006.

[44] K. Glette, J. Torresen. A flexible on-chip evolution system implemented on a Xilinx Virtex-II Pro device. *Evolvable Systems: From Biology to Hardware, LNCS* 3637, 2005.

[45] M. Goeke, M. Sipper, D. Mange, A. Stauffer, E. Sanchez, M. Tomassini. Online autonomous evolware. *Evolvable Systems: From Biology to Hardware, LNCS* 1259, 1997.

[46] T. Toffoli, N. Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press Series in Scientific Computation, 1987.

[47] M. Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*, Springer, 1997.

[48] R. A. Brooks. New approaches to robotics. *Science* 253, 1991.

[49] Y. U. Cao, A. S. Fukunaga, A. B. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots* 4(1), 1997.

[50] R. Moeckel, C. Jaquier, K. Drapel, E. Dittrich, A. Upegui, A. Ijspeert. YaMoR and Bluemove: An autonomous modular robot with Bluetooth interface for exploring adaptive locomotion. *Proceedings of the 8th International Conference on Climbing and Walking Robots (CLAWAR)*, 2005.

[51] T. S. Ray. An approach to the synthesis of life. *Artificial Life II, SFI Studies in the Sciences of Complexity* 10, 1992.

[52] A. Upegui, E. Sanchez. Evolving hardware by dynamically reconfiguring Xilinx FPGAs. *Evolvable Systems: From Biology to Hardware, LNCS* 3637, 2005.

[53] L. Sekanina. *Evolvable Components: From Theory to Hardware Implementations*, Springer, 2004.

[54] C. Slorach, K. Sharman. The design and implementation of custom architectures for evolvable hardware using off-the-shelf programmable devices. *Evolvable Systems: From Biology to Hardware, LNCS*, 2000.

[55] Y. Zhang, S. Smith, A. Tyrrell. Digital circuit design using intrinsic evolvable hardware. *Proceedings of the 2004 NASA/DOD Conference on Evolvable Hardware*, July 2004.

[56] L. Sekanina, S. Friedl. On routine implementation of virtual evolvable devices using COMBO6. *Proceedings of the 2004 NASA/DOD Conference on Evolvable Hardware*, July 2004.

[57] K. Vinger, J. Torresen. Implementing evolution of FIR-filters efficiently in an FPGA. *Proceedings of the 2003 NASA/DOD Conference on Evolvable Hardware*, July 2003.

[58] L. Sekanina. Towards evolvable IP cores for FPGAs. *Proceedings of the 2003 NASA/DOD Conference on Evolvable Hardware*, July 2003.

[59] P. C. Haddow, G. Tufte. An evolvable hardware FPGA for adaptive hardware. *Proceedings of the 2000 Congress on Evolutionary Computation*, 2000.

[60] M. Sipper, M. Goeke, D. Mange, A. Stauffer, E. Sanchez, M. Tomassini. The firefly machine: Online evolware. *Proceedings of the IEEE International Conference on Evolutionary Computation*, 1997.

[61] Xilinx, Inc. *The XC6200 Data Sheet v.1.7*, 1996.

[62] A. Thompson, P. Layzell. Evolution of robustness in an electronics design. *Evolvable Systems: From Biology to Hardware, LNCS* 1801, 2000.

[63] D.-W. Lee, C.-B. Ban, K.-B. Sim, H.-S. Seok, L. Kwang-Ju, B.-T. Zhang. Behavior evolution of autonomous mobile robot using genetic programming based on evolvable hardware. *Proceeding of the 2000 IEEE International Conference on Systems, Man, Cybernetics*, 2000.

[64] J. R. Koza, F. H. Bennett, J. Hutchings, S. L. Bade, M. A. Keane, D. Andre. Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate arrays. *Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence*, 1997.

[65] J. Dumoulin, J. A. Foster, J. F. Frenzel, S. McGrew. Special purpose image convolution with evolvable hardware. *Real-World Applications of Evolutionary Computing, EvoWorkshops 2000, LNCS*, 2000.

[66] Xilinx, Inc. *Virtex-II Platform FPGA User Guide* (*www.xilinx.com*), March 2005.

[67] L. Sekanina. Virtual reconfigurable circuits for real-world applications of evolvable hardware. *Evolvable Systems: From Biology to Hardware, LNCS* 2606, 2003.

[68] G. Hollingworth, S. Smith, A. Tyrrell. Safe intrinsic evolution of Virtex devices. *Proceedings of the Second NASA/DoD Workshop on Evolvable Hardware*, 2000.

[69] R. O. Canham, A. Tyrrell. Evolved fault tolerance in evolvable hardware. *Proceedings of the Congress on Evolutionary Computation*, 2002.

[70] L. Sekanina, V. Drabek. The concept of pseudo evolvable hardware. *Proceedings of the IFAC Workshop on Programmable Devices and Systems*, 2000.

[71] L. Durbeck, N. J. Macias. Defect-tolerant, fine-grained parallel testing of a cell matrix. *Proceedings of SPIE ITCom* 4867, 2002.

[72] P. Haddow, G. Tufte. Bridging the genotype-phenotype mapping for digital FPGAs. *Proceedings of the Third NASA/DoD Workshop on Evolvable Hardware*, 2001.

[73] S. A. Guccione, D. Levi, P. Sundararajan. JBits: A Java-based interface for reconfigurable computing. *Proceedings of the Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*, 1999.

[74] G. Hollingworth, S. Smith, A. Tyrrell. The intrinsic evolution of Virtex devices through Internet reconfigurable logic. *Evolvable Systems: From Biology to Hardware, LNCS* 1801, 2000.

[75] A. M. Tyrrell, R. A. Krohling, Y. Zhou. Evolutionary algorithm for the promotion of evolvable hardware. *IEE Proceedings—Computers and Digital Techniques* 151(4), 2004.

[76] D. Levi, S. A. Guccione. Genetic FPGA: Evolving stable circuits on mainstream FPGA devices. *Proceedings of the First NASA/DOD Workshop on Evolvable Hardware*, 1999.

[77] Xilinx, Inc. *XAPP 290: Two Flows for Partial Reconfiguration: Module Based or Difference Based* (*www.xilinx.com*), September 2004.

[78] Y. Thoma, E. Sanchez. A reconfigurable chip for evolvable hardware. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2004.

[79] Y. Thoma, G. Tempesti, E. Sanchez, J.M.M. Arostegui. POEtic: An electronic tissue for bio-inspired cellular applications. *Biosystems* 76(1–3), 2004.

[80] Y. Thoma, E. Sanchez, J.M.M. Arostegui, G. Tempesti. A dynamic routing algorithm for a bio-inspired reconfigurable circuit. *Proceedings of the International Conference on Field-Programmable Logic and Applications* 2778, 2003.

[81] M. Iwata, I. Kajitani, Y. Liu, N. Kajihara, T. Higuchi. Implementation of a gate-level evolvable hardware chip. *Evolvable Systems: From Biology to Hardware, LNCS* 2210, 2001.

[82] T. Higuchi, M. Iwata, H. Sakanashi, E. Takahashi, M. Murakawa, I. Kajitani. Dynamic adaptive devices and their applications. *Bulletin of the Electrotechnical Laboratory, Special Issue: RWC Research Toward Realization of Real World Intelligence* 64(4/5), 2000.

[83] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, M. Salami, N. Kajihara, N. Otsu. Real-world applications of analog and digital evolvable hardware. *IEEE Transactions on Evolutionary Computation* 3(3), 1999.

[84] I. Kajitani, M. Iwata, M. Harada, T. Higuchi. A myoelectric controlled prosthetic hand with an evolvable hardware LSI chip. *Technology and Disability, Special Issue: Advances in the Control of Prosthetic Arms* 15(2), 2003.

[85] I. Kajitani, T. Hoshino, N. Kajihara, M. Iwata, T. Higuchi. An evolvable hardware chip and its application as a multi-function prosthetic hand controller. *Proceedings of the 16th National Conference on Artificial Intelligence,* 1999.

[86] H. Sakanashi, M. Iwata, T. Higuchi. Evolvable hardware for lossless compression of very high resolution bi-level images. *IEE Proceedings—Computers and Digital Techniques* 151(4), 2004.

[87] H. Sakanashi, M. Iwata, D. Keymulen, M. Murakawa, I. Kajitani, M. Tanaka, T. Higuchi. Evolvable hardware chips and their applications. *Proceedings of the International Conference on Systems, Man, and Cybernetics,* 1999.

[88] D. Keymeulen, M. Iwata, Y. Kuniyoshi, T. Higuchi. Online evolution for a self-adapting robotic navigation system using evolvable hardware. *Artificial Life* 4, 1998.

[89] E. Takahashi, M. Murakawa, Y. Kasai, T. Higuchi. Power dissipation reductions with genetic algorithms. *Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware,* 2003.

[90] A. Stoica, R. Zebulum, D. Keymeulen, R. Tawel, T. Daud, A. Thakoor. Reconfigurable VLSI architectures for evolvable hardware: From experimental field-programmable transistor arrays to evolution-oriented chips. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9(1), 2001.

[91] A. Stoica, R. Zebulum, M. Ferguson, D. Keymeulen, V. Duong. Evolving circuits in seconds: Experiments with a stand-alone board-level evolvable system. *Proceedings of the 2002 NASA/DOD Conference on Evolvable Hardware,* July 2002.

[92] P. Layzell. A new research tool for intrinsic hardware evolution. *Evolvable Systems: From Biology to Hardware, LNCS,* 1998.

[93] L. Sekanina, R. Ruzicka. Easily testable image operators: The class of circuits where evolution beats engineers. *Proceedings of the 2003 NASA/DOD Conference on Evolvable Hardware,* July 2003.

[94] J. Lohn, J. Crawford, A. Globus, G. Hornby, W. Kraus, G. Larchev, A. Pryor, D. Srivastava. Evolvable systems for space applications. *Proceedings of the International Conference on Space Mission Challenges for Information Technology,* 2003.

[95] J. Lohn, D. Linden, G. Hornby, W. Kraus, A. Rodriguez-Arroyo. Evolutionary design of an X-band antenna for NASA's space technology 5 mission. *Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware*, 2003.

[96] J. F. Miller, K. Downing. Evolution in materio: Looking beyond the silicon box. *Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware*, 2002.

[97] L. P. Masiero, M. Pacheco, C. R. Hall, C. Santini. Molecular circuit design. *Proceedings of the 2005 NASA/DOD Conference on Evolvable Hardware*. June–July, 2005.

[98] L. Spector, H. Barnum, H. J. Bernstein, N. Swamy. Quantum computing applications of genetic programming. *Advances in Genetic Programming*, MIT Press, 1999.

CHAPTER 34

NETWORK PACKET PROCESSING IN RECONFIGURABLE HARDWARE

John W. Lockwood
Washington University in St. Louis and Stanford University

This chapter will show, through an example, how networking systems have been built with reconfigurable hardware. It will describe how data can be switched, routed, buffered, processed, scanned, and filtered over networks using field-programmable gate arrays (FPGAs).

The chapter begins by describing the mechanisms by which Internet packets are segmented into frames and cells for transmission across a network. Internet Protocol (IP) wrappers are introduced, and it is shown how they simplify the implementation of large packet-processing systems. Next, a framework for building modular systems that implement Internet firewalls and intrusion prevention systems is presented. The chapter continues with a detailed explanation of how Bloom filters can scan streams of data for fixed strings and how finite automata can be used to scan for regular expressions.

Case studies are provided that show how deep packet inspection systems are implemented in reconfigurable hardware. One circuit detects the spread of worms and viruses across an Internet link. Another circuit analyzes the semantics of the text in traffic flows to determine which language is used within attached documents. A hardware-accelerated version of the popular SNORT intrusion detection system is illustrated, and it is shown how the FPGA hardware works with the software on a host to analyze packets.

34.1 NETWORKING WITH RECONFIGURABLE HARDWARE

34.1.1 The Motivation for Building Networks with Reconfigurable Hardware

Although modern microprocessors continue to improve their performance, they are not improving as fast as the rate at which data flows over Internet connections. As the limits of Moore's Law are reached, alternative computational methods are needed to route, process, filter, and transform Internet datastreams.

Networking systems created with reconfigurable hardware are flexible and easily modified to provide new functionality. Reconfigurable hardware enables features on networking platforms to be implemented in ways that are quite different from current platform implementations. It allows new modular components to be created and then dynamically installed in remote networksystems.

By processing network packets in hardware rather than in software, networking applications do not suffer the performance penalty caused by sequential data processing.

The Internet evolves as new protocols, features, and capabilities are added to the routers that implement the underlying network. Protocols, such as IP version 6 (IPv6), allow more devices to be individually addressed. Added features, such as per-flow queuing, allow voice and video to be reliably delivered in real time. Firewalls and intrusion prevention systems (IPSs) enhance Internet security.

Network platforms have been built to route network traffic, filter packets, and queue data in reprogrammable hardware. With reconfigurable hardware, networking platform operation can change over time as packet-processing algorithms and protocols evolve. With FPGAs, all features of the packet-processing system are configurable down to the logic gates. These systems enable new services to deploy and operate at the rate of the highest-speed backbone links.

34.1.2 Hardware and Software for Packet Processing

For their packet-processing operations, today's fastest routers use network processing elements implemented in custom silicon or in application-specific integrated circuits (ASICs). As shown in Figure 34.1, network processing elements reside between the line card where packets are transmitted and received and the Gigabit/second rate switch fabric that interconnects ports. They contain hundreds to thousands of parallel logic circuits and finite-state machines that are optimized to route, filter, queue, and/or process Internet datagrams in hardware.

Several platform types have been developed, many of which use standard microprocessors such as the Intel Pentium, AMD Athlon, or Motorola/IBM PowerPC. Others use ASICs from vendors such as Agere, Intel, Motorola, Cavium, Broadcom and Vitesse. Although software-based systems have outstanding flexibility, their packet processing is limited because of the sequential nature of their instruction execution. ASICs and custom silicon networking chips have high performance, but they offer little flexibility as measured by their ability to reprogram. Figure 34.2 illustrates the trade-offs between flexibility and performance.

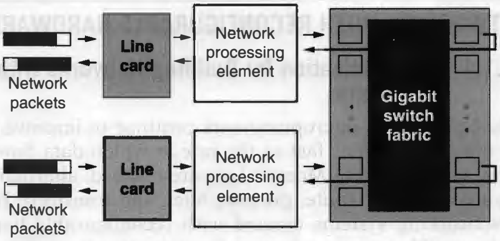


FIGURE 34.1 ▪ A reconfigurable network processing element located between a line card and switch fabric.

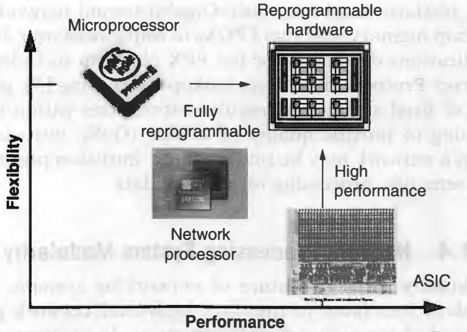


FIGURE 34.2 ■ Flexibility and performance trade-offs for networking systems that use microprocessors, network processors, ASICs, and reprogrammable hardware.

34.1.3 Network Data Processing with FPGAs

Reconfigurable hardware devices share the performance advantage of ASICs because they can implement parallel logic functions in hardware. However, they also share the flexibility of microprocessors and network processors because they can be dynamically reconfigured.

Using FPGAs for high-performance asynchronous transfer mode (ATM) networking was explored during the development of the Illinois Pular-based Optical Interconnect (iPOINT) testbed. In this project, an ATM switch with FPGAs [2] was developed and an advanced queuing module was implemented that provided per-flow queuing functionality in FPGA hardware. The FPGAs were used to implement the datapath of the switch and to control the state machines that buffered the ATM cells as they arrived on each switch port of the switch. The lookup tables (LUTs) in the FPGA fabric were used to build the multiplexers that switched the data between the ports. Finally, combinational logic was used to implement the state machines that controlled how packets were written to and read from SRAM [3].

FPGAs have also proven effective for implementation of bit-intensive function networking, such as forward error correction (FEC), and for boosting the performance of networking protocols [4]. The bitwise processing function maps well into the fine-grained logic on an FPGA. On-chip LUTs are used to encode data patterns as symbols with redundant bits of information. When the symbols are decoded, the redundant bits allow the receiver to reconstruct the data even with a few bits in error. Reconfigurable logic allows algorithms that use varying amounts and types of error correction to be programmed on-chip.

Through the development of the Field-Programmable Port Extender (FPX) platform [1], it was demonstrated that high-performance network packet-processing systems implemented with FPGAs are both useful and practical. The

FPX platform used two multi-Gigabit/second network interfaces, four banks of off-chip memory, and two FPGAs to implement over 30 networking applications. Applications developed for the FPX platform included modules that performed Internet Protocol IP address lookup for routing [7]; payload scanning for detection of fixed strings and regular expressions within the body of a packet; data queuing to provide quality of service (QoS); intrusion detection to determine when a network may be under attack; intrusion prevention to halt such attacks; and semantic processing of network data.

34.1.4 Network Processing System Modularity

Modularity is a key feature of networking systems. Network developers need standard interfaces to interface high-level network processing components to the underlying network infrastructure. In systems with reconfigurable hardware, modules can be implemented in regions of an FPGA and bound by a well-defined interface to the datapath and to external memory. Multiple modular data-processing components can be integrated to compose systems. Memory interfaces can connect logic to off-chip memory in order to buffer data and hold large lookup tables LUTs.

For the FPX platform modules, data was received and transmitted via a series of ATM cells carried over a 32-bit-wide Utopia interface. ATM cells contained 48 bytes of payload data and 4 bytes of a header that included a virtual path identifier (VPI) and a virtual circuit identifier (VCI). Each ATM cell also included an 8-bit checksum that covered the ATM cell header. Larger IP datagrams were sent between modules using layered protocol wrappers that segmented and reassembled multiple cells into ATM adaptation layer 5 (AAL5) frames. These frames contained data from a series of ATM cells and a 32-bit checksum at the end that covered all bytes of the payload. Segmentation and reassembly of cells into frames were performed to transfer packets over the network.

The FPX platform (Figure 34.3) stored and loaded data from two types of off-chip memory. Two interfaces supported transfer of 36-bit-wide data to and from an on-chip SRAM. SDRAM interfaces provided 64-bit-wide interfaces to multiple banks of high-capacity, off-chip memory. In the implementation of the IP lookup module, the off-chip SRAM was used to store data structures for IP lookup, while the SDRAM was used to buffer packets. The lower latency of SRAM access was important for the implementation of lookup functions where there was a data dependency for the result; the larger capacity of the SDRAM was beneficial for reducing the cost of storing bulk data, including buffering dataflows.

A switch was implemented using the reprogrammable application device (RAD) FPGA logic that allowed traffic to be routed to extensible modules. Layered protocol wrappers performed the segmentation and reassembly of AAL5 frames so that full packets could be processed by the FPGA hardware. To reprogram the RAD FPGA that contained the extensible modules, configuration and control logic was implemented on the network interface device (NID) FPGA.

The FPX platform was integrated into the Washington University Gigabit Switch (WUGS) to process packets as they passed into and out of the networking

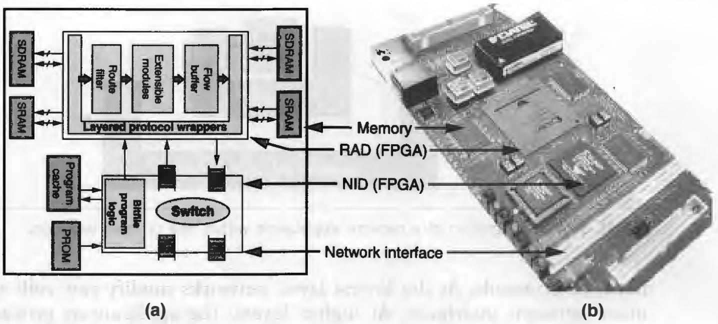


FIGURE 34.3 ■ A block diagram and a physical implementation of the FPX platform.

ports of a scalable network switch. The WUGS switching platform provided a backplane for transferring ATM cells between ports. By adding the FPX between the line cards and the switch fabric, the system was able to analyze, process, route, and filter IP packets as they flowed through the system. OC-3 to OC-48 line cards were used to directly send and receive ATM cells, while Gigabit Ethernet line cards were used to segment frames into multiple ATM cells and reassemble them. After data passed through the FPX, they were forwarded to the switch fabric, where cells were forwarded to other FPX modules in the chassis based on their VPI and VCI values.

34.2 NETWORK PROTOCOL PROCESSING

The Open Systems Interconnection (OSI) Reference Model defines how multiple layers can be used to transport data over a computer network. OSI divides the functions of a protocol into a series of layers, each of which has two properties: (1) It uses only the functions of the layer below, and (2) it exports functionality only to the layer above. A system that implements protocol behavior consisting of a series of these layers is known as a protocol stack. Protocol stacks can be implemented in hardware, in software, or in a mixture of the two (typically, only the lower layers are implemented in hardware; the higher layers, in software). This logical separation makes reasoning about the behavior of protocol stacks much easier and allows their design to be elaborate but highly reliable. Each layer performs services for the next highest layer and makes requests for the next lowest layer [5].

For real systems that process Internet data, the OSI model is not directly implemented but instead serves as a reference for implementation of the real protocols. Layers are important for processing IP data, however, because they permit application-processing modules to abstract details of the lower-layer

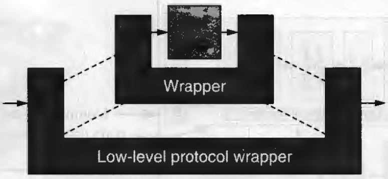


FIGURE 34.4 ■ Integration of a network application within one or more wrappers.

network protocols. At the lowest layer, networks modify raw cells of data that move between interfaces. At higher layers, the applications process variable-length frames or IP packets. To send and receive data at the user level, a network application may transmit directly or receive user datagram protocol (UDP) messages by instantiating all wrappers and sending data from a network application down through a series of wrappers [6] (see Figure 34.4).

34.2.1 Internet Protocol Wrappers

Hundreds of millions of computers deployed throughout the world communicate over the Internet. Traffic from these machines is concentrated to flow over a smaller number of routers that forward traffic through the Internet core. Currently, Internet backbones operate over communication links ranging in speed from OC-3 (155 Mbps) to OC-768 (40 Gbps). Fast links that process small packets have the ability to process millions of IP packets per second.

A library of layered protocol wrappers (see Figure 34.5) was developed to process Internet packets in reconfigurable hardware. Collectively, the wrappers simplified and streamlined the implementation of high-level networking functions by abstracting the operation of lower-level packet-processing functions. The library infrastructure was synthesized into FPGA logic and integrated into an FPX network platform. At the lowest levels, the library processes ATM cells. Complete frames of data are segmented and reassembled using ATM adaptation layer 5 (AAL5), over which IP messages are then transported.

When only a single message needs to be transmitted, the UDP can send one packet over the Internet. UDP encapsulates a variable-length message into an IP packet and allows the system to specify source and destination port numbers that identify from which application on a machine the data was sent and to which application it should be delivered. UDP/IP also provides a checksum to ensure the integrity of the data. Using the FPX protocol-processing library, this checksum is automatically computed, using FPGA hardware, as the sum over the payload bytes of the message.

34.2.2 TCP Wrappers

Over 85 percent of all traffic on the Internet today uses the Transmission Control Protocol (TCP). TCP is stream oriented and guarantees delivery of data with

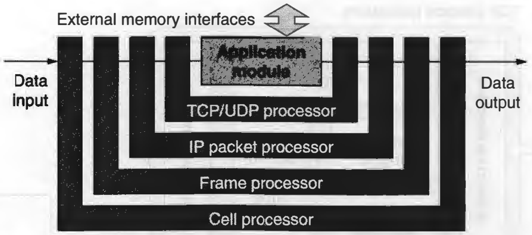


FIGURE 34.5 ■ Implementation of layered protocol wrappers on the FPX platform.

an ordered byte flow. Processing TCP dataflows in the middle of the network is extremely difficult because network packets can be dropped, duplicated, and reordered. Packet sequences observed within the interior of the network may be different from packets received and processed at the connection endpoints. The complexities associated with tracking the state of end systems and reconstructing byte sequences based on observed traffic are significant.

A TCP processing circuit was developed that handles the complexities associated with flow classification and TCP stream reassembly. It provided the FPGA logic with a view of network traffic flow data through a simple client interface. The TCP wrapper enabled other high-performance data-processing subsystems to operate on TCP network content without needing to implement their own state-tracking operations. The TCP module used a state store to track the status of each TCP/IP flow and, using a hash function, assigned a unique flow number to each session [8].

Figure 34.6 is a block diagram of the TCP processor. Internet packets arrive as frames of data to the input state machine of the TCP processing engine. The input state machine forwards the frames to a first in, first out (FIFO) that buffers the packet; a checksum engine that computes and verifies the correctness of the TCP checksum; and a flow classifier that computes a flow identifier (flow ID) using a hash over fields in the packet header.

The flow ID is passed to the state store manager that retrieves the state associated with the particular flow. Results are written to the control and state FIFO, and the state store is updated with the current flow state. The output state machine reads data from the frame and control FIFO buffers and passes data to the packet-routing engine. Most traffic flows through the content-scanning engines, which scan the data. Packet retransmissions bypass these engines and go directly to the flow-blocking module.

Data returning from the content-scanning engines also goes to the flow-blocking module. This stage updates the per-flow state store with application-specific state information. If a content-scanning engine indicates that it has a need to block a flow, the flow-blocking module can enforce this rule by comparing the packet's sequence number with the sequence numbers for which flow blocking should take place. If the packet meets the blocking criteria, the

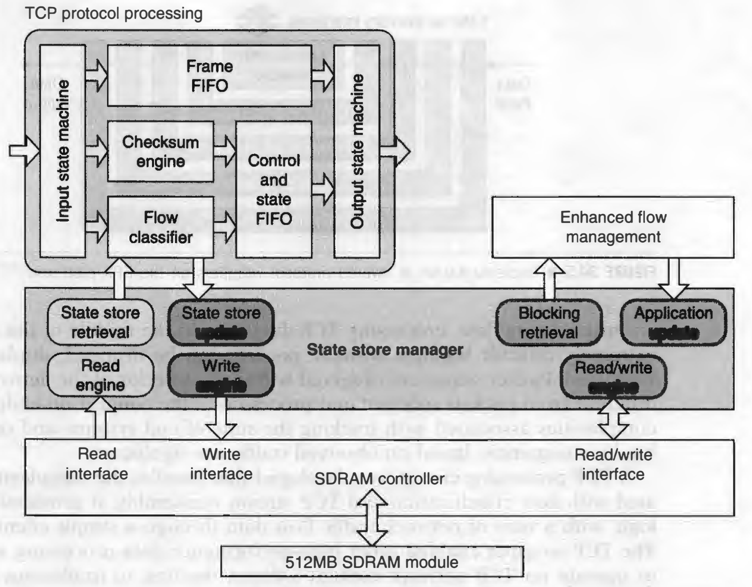


FIGURE 34.6 ■ A block diagram of the TCP processor.

flow-blocking module drops it from the network. Any remaining packets go to the outbound protocol wrapper.

The state store manager processes requests to read and write flow state records. It also handles all interactions with SDRAM memory and caches recently accessed flow state information. The SDRAM controller exposes three memory access interfaces: a read/write, a write-only, and a read-only. The controller prioritizes requests in that order, with the read/write interface having the highest priority.

34.2.3 Payload-processing Modules

Many network applications have a common requirement for string matching in the payload of packets or flows. Once the data being transported over the network has been reconstructed using the IP and TCP modules, it can be examined in the payload. For example, the presence of a string of bytes (or a signature) can identify the presence of a media file, an attachment, or a security exploit. Well-known Internet worms, such as Nimda, Code Red, and Slammer, propagate by sending malicious executable programs identifiable by certain byte sequences in payloads [14]. Because the location (or offset) of such strings and their length are unknown, such applications must be able to detect strings of different lengths starting at arbitrary packet payload locations.

Packet inspection applications, when deployed at router ports, must operate at wire speeds. As network rates increase, the implementation of packet monitors that process data at Gigabit/second line rates has become increasingly difficult. Thus, the growth in network traffic has motivated specialized packet- and payload-processing modules in hardware.

34.2.4 Payload Processing with Regular Expression Scanning

A regular expression (RE) is a pattern that describes a set of strings. The basic building blocks for these patterns consist of individual characters, such as {`a`, `b`, and `c`}. These characters can be combined with meta-characters, such as: {`*`, `|`, and `?`}, to form regular expressions with wildcards. For two regular expressions, `r1` and `r2`, rules define that `r1*` matches any string composed of zero or more occurrences of `r1`; `r1?` matches any string composed of zero or one occurrence of `r1`;`r1|r2` matches any string composed of `r1` or `r2`; and `r1r2` matches any string composed of `r1` concatenated with `r2`. For instance, `a` is an RE that denotes the singleton set {`a`}, while `a|b` denotes the set {`a, b`} and `a*` denotes the infinite set {null, `a, aa, aaa, . . .`}. REs can be identified using nondeterministic finite automata (NFA).

Research on RE matching in hardware has been performed by Sidhu and Prasanna [16] and Franklin et al. [17]. Sidhu and Prasanna were primarily concerned with minimizing the time and space required to construct NFAs. They ran their NFA construction algorithm in hardware as opposed to software. Franklin et al. followed with an analysis of this approach for the large set of expressions found in a SNORT database [18].

The search function FPgrep was implemented by Moscola et al. to search packet payloads for substrings that belong to the language defined by the RE [15]. When FPgrep matched a substring in a packet, it transmitted information about the packet to a monitoring host system. The information sent for network intrusion detection functions specified the content found and the sender's and receiver's IP addresses. The search ran in linear time (proportional to packet size), $O(n)$ (where n was the number of bytes in a packet), and in constant space. That is, there was never a need to examine a character more than once and the amount of hardware was proportional to the size of the RE. Approximately one flip-flop was required per character.

A streaming content editor, FPsed, was implemented as a module on the FPX platform. The FPsed module selectively replaced content in packet payloads. String replacement for an RE is not as straightforward or efficient as searching. It requires that the machine do more than simply determine the presence of matching substrings in a record—it must also determine the position of the first and last character of all complete substrings that are matched by it. It is this requirement that makes RE search and replace more complicated and less efficient than a simple search. Searching for the complete substring is logical when the goal is to replace it.

Consider the replacement of every occurrence of a certain hexadecimal string associated with a computer virus, 3n*4n*5n*B, with the text Virus Pattern Detected. For the sake of brevity, the previous expression uses *n* as shorthand for any hexadecimal character (i.e., 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F). For the input string 3172F34435B6B7B8, the substring can be replaced from the point where the machine starts running, 34, to the point where the substring is accepted, just before B6 (i.e., substring 34435B). However, this would allow a portion of the virus to remain in the content stream. In most situations, it is preferable to replace complete substrings; here the complete substring match starts with 31 and includes everything to just before B8 (i.e., the substring 3172F34435B6B7B).

34.2.5 Payload Scanning with Bloom Filters

A hash table is one of the most attractive choices for quick lookups. Hash tables require only constant time, $O(1)$, average memory accesses per lookup. Because of their versatile applicability in network packet processing, it is useful to implement these hashing functions in hardware [19, 20].

Bloom filters can detect strings of characters that appear in streaming data moving at very high data rates. A Bloom filter is a data structure that stores a set of signatures compactly by computing multiple hash functions on each member of the set. It queries a database of strings to check for the membership of a particular string. The answer to this query can be false positive but never false negative. The average computation time to perform a query remains constant so long as the sizes of the hash tables scale linearly with the number of strings they store. Because each table entry stores only a hashed version of the content, the amount of storage required by the Bloom filter for each string is independent of its length.

34.3 INTRUSION DETECTION AND PREVENTION

Existing firewalls that examine only the packet headers do little to protect against many types of attack. Multiple new worms transport their malicious software, or *malware*, over trusted services and cannot be detected without examining the payload. Intrusion detection systems (IDSs) perform deep scanning of the payload to detect malware, but do nothing to impede the attack because they only operate passively. An intrusion prevention system (IPS), on the other hand, can intervene and stop malware from spreading. The configuration of a network intrusion prevention system is shown in Figure 34.7.

One problem with software-based IDSs is that they cannot keep pace with the high volume of traffic that transits high-speed networks. Existing systems that implement IPS functions in software limit the bandwidth of the network and delay the end-to-end connection.

A reconfigurable system that can keep pace with high-speed network traffic has been developed. It scans data quickly, reconfigures to search for new attack

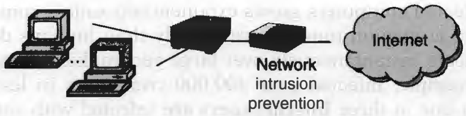


FIGURE 34.7 ■ Configuration of an in-line network IPS situated between two hosts attached to a router and to the Internet.

patterns, and takes immediate action when attacks occur. By processing the content of Internet traffic in real time within an extensible network, data that contains computer viruses or Internet worms can be detected and prevented. By adding only a few filtering devices at key network aggregation points, Internet worms and computer viruses can be quarantined to the subnets where they were introduced.

A complete system has been designed and implemented that scans the full payload of packets to route, block, and track the packets in the flow based on their content. The result is an intelligent gateway that provides Internet worm and virus protection in both local and wide area networks.

Network intrusion detection and prevention systems search for predefined virus or worm signatures in network traffic flows (see Section 34.2.3). Such signatures can be loaded into the system manually by an operator or automatically by a signature detection system. (Note that *string* is synonymous with *signature* throughout the chapter.)

Once a signature is found, an intrusion detection and prevention system (IDPS) can use it to block traffic containing infected data from spreading throughout a network. To perform this operation on a high-speed network, the signature scanning and data blocking must operate quickly. Comparing a variety of systems running the SNORT rule-based NID sensor reveals that most general-purpose computer systems are inadequate as NID sensor platforms even for moderate-speed networks. Factors such as microprocessor, operating system, main memory bandwidth, and latency limit the performance that an NIDS sensor platform can achieve [22].

34.3.1 Worm and Virus Protection

Computer virus and Internet worm attacks are pervasive, aggravating, and expensive, both in terms of lost productivity and consumption of network bandwidth. Attacks by Nimba, Code Red, Slammer, SoBig.F, and MSBlast have infected computers globally, clogged large computer networks, and degraded corporate productivity. It can take weeks to months for information technology professionals to sanitize infected computers in a network after an outbreak [24].

In the same way that a human virus spreads among people coming in contact with each other, computer viruses and Internet worms spread when computers communicate electronically [25]. Once a few systems are compromised, they infect other machines, which in turn quickly spread the infection throughout a network. As is the case with the spread of a contagious disease, the number of infected computers grows exponentially unless contained. Computer systems spread contagion much more quickly than humans do because they can communicate instantaneously over large geographical distances. The Blaster worm, for example, infected over 400,000 computers in less than five days. In fact, about one in three Internet users are infected with some type of virus or worm every year.

Malware can propagate as a computer virus, an Internet worm, or a hybrid of both. Viruses spread when a computer user downloads unsafe software, opens a malicious attachment, or exchanges infected computer programs over a network. An Internet worm spreads over the network automatically when malware exploits one or more vulnerabilities in an operating system, a web server, a database application, or an email exchange system.

Malware can appear as a virus embedded in software that a user has downloaded. It can also take the form of a Trojan that is embedded in what appears to be benign freeware. Alternatively, it can spread as content attached to an email message, as content downloadable from a web site, or in files transferred over peer-to-peer systems. Modern attacks typically use multiple mechanisms to execute. Malware, for example, can spoof messages that lure users to submit personal financial information to cloaked servers. In the future, malware is likely to spread much faster and cause much more damage.

Today, most anti-virus solutions run in software on end systems. To ensure that an entire network is secure from known attacks, integrated systems were developed that can perform multiple network processing functions.

34.3.2 An Integrated Header, Payload, and Queuing System

An integrated system that incorporated the payload-scanning function, a ternary content addressable memory (TCAM) for header matching, and a flow buffer and queue manager for packet storage was implemented [13]. It is shown as a block diagram in Figure 34.8.

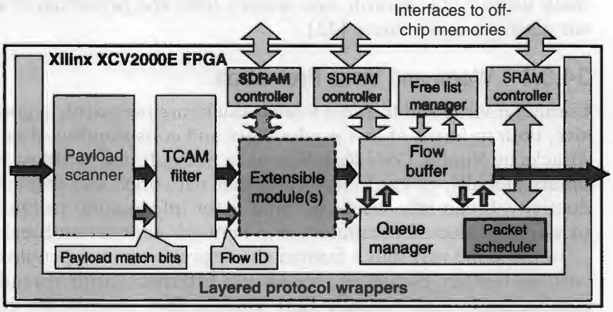


FIGURE 34.8 ■ Complete on-chip networking header and payload processing integrated with a flow buffer and a queue manager.

SNORT is a lightweight NID sensor that can filter packets based on predefined rules over packet headers and payloads [18]. With the TCP option enabled, SNORT matches strings that appear anywhere within traffic flows. Each SNORT rule operates first on the packet header to verify that the packet is from a source or to a destination network address and/or port of interest. If the packet matches a certain header rule, its payload is scanned against a set of predefined patterns associated with that rule. Matching of one or multiple patterns implies a complete match of a rule, and further action can be taken on either the packet or the TCP flow.

To provide complete detection of all known attacks, an intrusion system must process all packets. Several thousand patterns appeared in the version 2.2 rule set for SNORT. SNORT's rule database continually expands as new threats are observed. As the number of headers and signatures to match increases, the CPU on a PC running SNORT becomes overloaded and not all packets are processed. A SNORT intrusion filter for TCP (SIFT) was implemented in reconfigurable hardware and is illustrated in Figure 34.9. SIFT data entered the system via the TCP de-serialize wrapper. Control signals marked specific locations in the

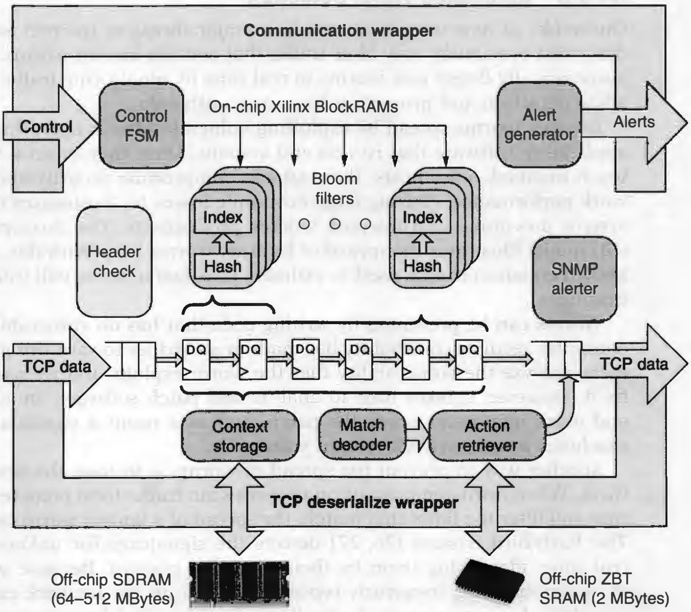


FIGURE 34.9 ■ A block diagram of SIFT.

packet that included the starts of the IP header, the TCP header, and the payload. The value of the header was sent to a header check component to determine if the packet matches a header-only rule. The payload was sent through an 8-stage pipeline where each byte offset is searched for signatures by Bloom filters. If a match was detected, the match decoder determines the string identifier (ID), which was next sent to the action retriever to determine what to do with the packet. Suspect packets were forwarded to software for further inspection. Those that had no match were not inspected further; those that did need additional processing were sent to the outgoing side of the TCP de-serialized wrapper.

To match payloads, SIFT used Bloom filters to allow signatures to be incrementally programmed into hardware. Signatures could be added or deleted via messages embedded in UDP control packets. These packets were sent through the communication wrapper to a control finite-state machine (FSM). In turn, the FSM set the appropriate bits in BlockRAM memories on the FPGA to add the signature to the Bloom filter. To achieve high throughput, four engines ran in parallel [21].

34.3.3 Automated Worm Detection

Outbreaks of new worms constitute a major threat to Internet security. IDPSs described previously only filter traffic that contain known worms. Systems that automatically detect new worms in real time by monitoring traffic on a network allow detection and protection from new outbreaks.

Internet worms spread by exploiting vulnerabilities in operating systems and application software that run on end systems. Once they infect a machine, they use it to attack other hosts; these attacks compromise security and degrade network performance, causing large economic losses for businesses resulting from system downtime and lowered worker productivity. The Susceptible/Infective (SI) model illustrates the spread of Internet worms [25]. With this model, a well-known equation can be used to estimate how fast a worm will infect vulnerable machines.

Worms can be prevented by writing code that has no vulnerabilities, and the computer security community has made great strides toward this goal. Programmers analyze the vulnerability that the worm exploits and release a "patch" to fix it. However, it takes time to analyze and patch software. In addition, many end users may never apply the patch, and as a result a significant number of machines in the network remain vulnerable.

Another way to prevent the spread of worms is to have the network contain them. When intrusion prevention systems scan traffic for a predetermined signature and filter the flows that match, the spread of a known worm can be blocked. The EarlyBird System [26, 27] detects the signatures for unknown worms in real time, identifying them by their repeating content. Because worms consist of malicious code, frequently repeated content on the network can be a useful warning of worm activity. Large flows are identified by computing a hash of packet content in combination with a destination port.

A hardware-accelerated worm detection circuit implemented in reconfigurable hardware draws from two ideas presented in the EarlyBird system [23]. To detect commonly occurring content, a hash is computed over 10-byte windows of streaming data. The hash value is used to identify a counter in a vector that is instructed to increment by one. At periodic intervals (called timeouts), the counts in each of the vectors are decremented by the average number of arrivals due to normal traffic. When a counter reaches a predetermined threshold, an alert is generated and its value is reset to zero.

For the implementation of the circuit on an FPGA, the count vector was implemented by configuring dual-ported, on-chip BlockRAMs as an array of memory locations. Each memory afforded one read operation and one write operation every clock cycle, which allowed a 3-stage pipeline to be implemented that reads, increments, and writes memory every clock cycle. Because the signature changes every clock cycle and because every occurrence of every signature must be counted, the dual-ported memories allow the occurrence count to be written back while another count is being read.

When an on-chip counter crosses the threshold, the corresponding signature is hashed to a table in off-chip SRAM. The next time the same string causes the counter to exceed the threshold, it is hashed to the same location in SRAM and the two strings are compared. If they are the same, it is determined that the match is not a false positive and the counter is incremented. If they are different, the contents of the string stored in SRAM is overwritten with the value of the new string and the count is reset.

On receiving confirmation from the SRAM analyzer that a signature frequently occurs, a UDP control packet is sent to an external computer. The packet contains the offending signature, which is the string of bytes by which the hash was computed. The computer, in turn, programs other IDS/IDP systems to filter traffic that contains this signature.

34.4 SEMANTIC PROCESSING

Next-generation networks route and forward data based on the semantics of the data within documents. Rather than assigning arbitrary headers to packets, routers use the meaning of the text itself to determine the packet routing.

34.4.1 Language Identification

As of 2004, nearly two-thirds of the world's Internet users spoke a non-English native language [29], and nearly one-third of the pages available on the Internet were written in a non-English language [29, 30]. As the rate at which data is transferred over the Internet increases, the rapid identification of languages becomes an increasingly difficult problem. A system capable of quickly identifying the primary language or languages used in documents can be useful as a preprocessor for document classification and translation services. It can also be used as a mechanism for language-based document routing.

A hardware-accelerated algorithm was designed to automatically identify the primary languages used in documents transferred over the Internet [28]. The module was implemented in hardware on the FPX platform. Referred to as Hardware-Accelerated Identification of Languages (HAIL), this complete system identified the primary languages used in content transferred over TCP/IP networks. It operated on streaming data at a rate of 2.4 Gigabits/second using FPGA hardware. This level of performance far outstripped software algorithms running on microprocessors.

Several methods have been shown to be effective for the classification of document characteristics based on principles from linguistics and artificial intelligence. Some methods used dictionary-building techniques [31], while others used Markov Models, trigram frequency vectors [32], and/or *n*-gram–based text categorization [33, 34]. Although these methods are capable of achieving high degrees of accuracy, most require floating-point mathematics, large amounts of memory, and/or generous amounts of processing time.

HAIL uses *n*-grams to determine the language of a document. These are sequential patterns of exactly *n* characters that are found in written documents, and when they are used as indicators of language, the primary language or languages of a document can be reliably determined. HAIL can use any *n*-gram length, although experiments have shown that *n*-grams of length 3 (trigrams) and length 4 (tetragrams) provide the most accurate results.

Before processing data with HAIL, the target system is trained with information on languages. Training is performed by scanning a set of documents in the languages of interest. When an *n*-gram appears significantly more frequently in the documents of one language than in any other, it is associated with that language. After training has established which *n*-grams best correspond to particular languages, memory modules on the hardware platform implementing HAIL have to be programmed. Memory is populated by using a hash to map each *n*-gram to a particular memory location. The memory location that corresponds to a particular *n*-gram is labeled with the associated language. Once data processing begins, the *n*-grams are sampled from the datastream and used as addresses into memory to discern the language associated with the *n*-gram. The final language is determined by the statistics of the words that appear in each language.

34.4.2 Semantic Processing of TCP Data

Within the intelligence community, there is a need to search through massive amounts of multilingual documents that are encoded using different character sets. It has been shown that computational linguistics and text-processing techniques are effective for sorting through large information sets, extracting relevant documents, and discovering new concepts [33]. There is a problem, however, in that the computational complexity of the text-processing algorithms is such that the document ingest rate is too slow to keep up with the high rate of information flow [34].

To overcome this problem, a system using FPGA hardware was developed for accelerated concept discovery and classification algorithms [35, 36].

Circuits were implemented as reconfigurable hardware modules that dramatically increased data ingest rates. It was found that text analysis algorithms that perform "bag of words" processing were widely used and appropriate for many types of computational linguistics tasks. To investigate the utility of hardware-accelerated text analysis algorithms, a reconfigurable FPGA-based semantic-processing system was developed. The hardware tested a variety of target problems involving concept classification, concept discovery, and language identification [36].

A blend of high-speed network devices and reconfigurable hardware was used to rapidly ingest and process data [35]. Data were received from the network as text or HTML documents and carried over standard TCP/IP packets. The TCP processor decoded the packets that contained the document in one or more TCP/IP input flows. Every word (baseword) in the document was analyzed for its semantic meaning. All words in each document were then counted to determine their frequency of occurrence. A document vector was generated that characterized the document content. It was then scored against a set of vectors that represented known or emerging concepts. Thresholds were used to determine if content could be classified as existing or if a new cluster should be formed.

Figure 34.10 diagrams the dataflow of the semantic-processing system. The FPGAs enabled streaming, computationally intensive semantic-processing functions to be performed in constant time. They performed all of the

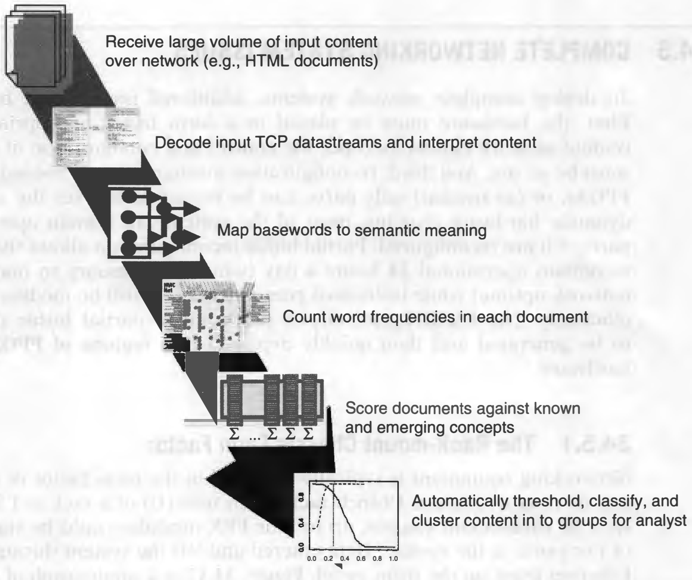

Receive large volume of input content over network (e.g., HTML documents)
Decode input TCP datastreams and interpret content
Map basewords to semantic meaning
Count word frequencies in each document
Score documents against known and emerging concepts
Σ ... Σ Σ Σ
Automatically threshold, classify, and cluster content in to groups for analyst
0.0 0.2 0.4 0.6 0.8 1.0

FIGURE 34.10 ■ Dataflow for the semantic processing system.

data-processing functions for the system shown in the figure except for threshold and classification (which were performed and displayed on a computer console). By using FPGAs to implement all parts of the text processing, the entire system could be dynamically reconfigured to allow variations of algorithms to be evaluated for their content classification or concept-clustering ability. Massive volumes of data were streamed through the system, and the system's precision, recall, throughput, and latency were measured [36].

The RAD circuits on the FPX (shown in Figure 34.3) were used to implement the TCP processor, the baseword module, the count module, the score module, and the report module. All were implemented as modular hardware components on individual FPX platforms connected in a vertical stack. The high-speed network interfaces allowed the FPX platforms to communicate intermediate results of processing to other modules in the system and to send reports to software running on a computer outside the system using standard IP datagrams. Multiple copies of the FPX platform were stacked on each other to implement network intrusion detection and network intrusion prevention. Figure 34.11 is a photograph displaying how five FPX cards were stacked to implement the semantic processing system. Additional modules were added to tag tokens in a context-free grammar [37].

34.5 COMPLETE NETWORKING SYSTEM ISSUES

To deploy complete network systems, additional issues must be considered. First, the hardware must be placed in a form factor appropriate for use in remote network closets. Second, the control and configuration of the hardware must be secure. And third, reconfiguration mechanisms are needed so that entire FPGAs, or (as needed) only parts, can be reconfigured over the network. With dynamic hardware plug-ins, most of the system can remain operational while parts of it are reconfigured. Partial bitfile reconfiguration allows the system itself to remain operational 24 hours a day (which is necessary to maintain a good network uptime) while individual components can still be modified quickly and efficiently. The PARBIT tool allows precompiled partial bitfile configurations to be generated and then quickly deployed into regions of FPGA networking hardware.

34.5.1 The Rack-mount Chassis Form Factor

Networking equipment is typically deployed in the form factor of a chassis that can be mounted into a 19-inch rack. Each unit (U) of a rack is 1.75 inches tall. In a 3U rack-mount chassis, up to four FPX modules could be stacked on each of two ports in the system. Data entered and left the system through the Gigabit Ethernet ports on the front panel. Figure 34.12 is a photograph of FPX modules integrated in a rack-mount chassis.

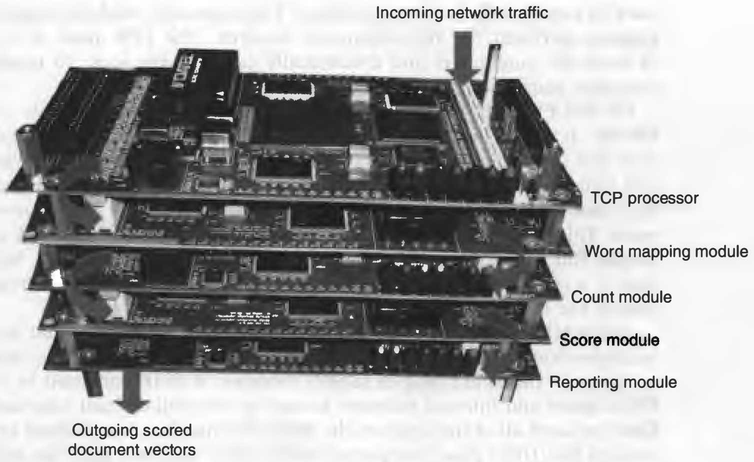


FIGURE 34.11 ■ A stack of the FPX modules implemented the semantic processing system.

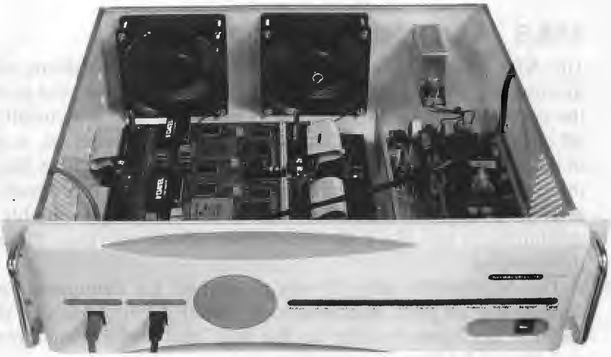


FIGURE 34.12 ■ FPX modules integrated in a rack-mount chassis.

34.5.2 Network Control and Configuration

Reconfigurable hardware circuits perform a variety of functions in the networking system. Some parts of the system implemented the infrastructure while others implemented the dynamically reconfigurable logic. Static circuits are used to switch cells between modules. The extensible modules implemented as plug-ins perform the reconfigurable features. The FPX used a combination of statically configured and dynamically configurable logic to implement the complete platform.

On the FPX, the NID was statically configured using a bitfile stored in a PROM. It controlled how data was routed between network modules. and included switching modules that forwarded traffic flows based on virtual paths and circuits found in the ATM cell headers. The NID also contained the logic that enabled other hardware modules to be dynamically loaded over the network. This logic implemented a circuit that used a reliable network protocol to receive full and partial bitfiles over the network. The NID, in turn, buffered this data in a configuration cache and streamed the bitstream into the programming port of the attached FPGA.

The RAD on the FPX was a Xilinx VirtexE-2000E FPGA that received the configuration data and performed application-specific functions implemented as dynamic hardware plug-in (DHP) modules. A DHP consisted of a region of FPGA gates and internal memory bound by the well-defined interface. For bitfiles that used all of the logic on the RAD, the interface was defined by user constraints file (UCF) pins. For partial bitfiles that used less than the entire FPGA, a standard on-chip interface was developed to transmit and receive packet data between modules. A full or partial bitfile was built using standard CAD tools [11].

34.5.3 A Reconfiguration Mechanism

The NID allowed modules created for the FPX platform to be remotely and dynamically loaded into the RAD. This bitstream was sent over the network into the configuration cache, which was implemented by a circuit that controlled an off-chip SRAM. Once a full or partial bitfile was received, a command was sent to the NID to initiate the RAD reconfiguration. On a Xilinx Virtex, the SelectMAP interface loaded a new bitstream into the FPGA. To reprogram the RAD, the NID read the configuration memory and wrote a preprogrammable number of configuration bytes into the RAD FPGA's SelectMAP interface. Figure 34.13 illustrates this process.

The NCHARGE API [9] was developed for debugging, programming, and configuring an FPX. Specifically, it included commands to check the status
an FPX, configure routing on the NID, and perform memory updates and
and partial RAD reprogramming.

NCHARGE provided a mechanism for applications to define their ow
control interface. Control cells were transmitted by NCHARGE an
by control cell processors (CCPs) on the RAD or NID. To config
the traffic flowing through the system, NCHARGE sent control
mands that modified routing tables on the Gigabit switch c
check the status of the FPX, NCHARGE sent a control cel
FPX, the NID updated fields in the cell, and the softwar
response.

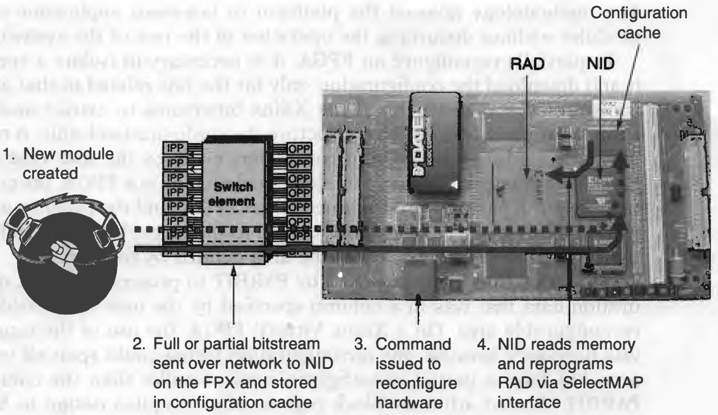

Configuration cache
RAD
NID
1 New module created
IPP
IPP
IPP
IPP
IPP
IPP
IPP
IPP
Switch element
OPP
OPP
OPP
OPP
OPP
OPP
OPP
OPP
2. Full or partial bitstream sent over network to NID on the FPX and stored in configuration cache
3. Command issued to reconfigure hardware
4. NID reads memory and reprograms RAD via SelectMAP interface

FIGURE 34.13 ■ Remote reconfiguration of the FPX platform.

34.5.4 Dynamic Hardware Plug-ins

Use of runtime reconfiguration in networking systems enables developers of hardware packet-processing applications to achieve a capability similar to that of the dynamically linked libraries (DLLs) used in software applications. Just as a DLL is a software module that can be attached to or removed from a running program as an application demands, DHPs can be loaded into or removed from a running FPGA without disturbing other circuits operating in it. The ability to change the hardware feature set in a running system is particularly useful in packet-processing applications such as firewalls and routers where it is not desirable to suspend the network operation during reprogramming.

A practical system for implementing DHPs was implemented on the FPX and provided sufficient resources for networking, well-defined interfaces to hardware, a complete design methodology, scripts that ran physical implementation tools to place and route logic, and tools that allowed selective reconfiguration of portions of the bitstream. These five elements were analogous to an operating system platform, application programming interface, modular programming methodology, compiler, and linker needed to implement DLLs in the software domain.

34.5.5 Partial Bitfile Generation

Tools and a design methodology were developed to support partial runtime reconfiguration of DHP modules on the FPX platform. The PARBIT tool was developed to transform and restructure bitstreams created by standard computer-aided design tools into partial bitstreams that programmed DHPs.

The methodology allowed the platform to hot-swap application-specific DHP modules without disturbing the operation of the rest of the system [12].

To partially reconfigure an FPGA, it is necessary to isolate a specific area in it and download the configuration only for the bits related to that area. PARBIT transformed and restructured the Xilinx bitstreams to extract and merge data from the bitfile's regions. To restructure the configuration bitfile, it read the original bitfile, a target bitfile, and parameters given by the user that specified the block coordinates of the logic implemented on a source FPGA, the coordinates of the area for a partially programmed target FPGA, and the programming options. After reading these data, PARBIT copied to the target bitstream only the part of the original bitstream related to the area defined by the user.

The target bitstream was used by PARBIT to preserve the part of the configuration data that was in a column specified by the user but outside the partial reconfigurable area. On a Xilinx VirtexE FPGA, the use of the target bitstream was necessary because one reconfiguration frame could span all rows of a column but have a partial reconfigurable area smaller than the column's height. PARBIT allowed arbitrary block regions of a compiled design to be retargeted into any similarly sized region of an FPGA.

To relocate blocks from the original bitfile, a user defined the start and end columns and rows for the block in the original design. Then the user defined where to put this block in a target bitfile of the same device type. The tool generated the partial bitfile containing the area selected by the user (from the original bitfile). This data was used to reconfigure the target device. The configuration bits for the top and bottom input/output blocks (IOBs) from the target device did not change after the partial bitfile was loaded. Those for the columns from the original and target bitfile were merged according to the rows defined by the user.

34.5.6 Control Channel Security

For devices deployed remotely on the Internet, security of the control channel is critical. Remote systems need to be safe from both passive and active network attacks by malicious users. In passive attacks, malicious users glean information by monitoring the system. In active attacks, they attempt to change the system's behavior or paralyze it. Access control mechanisms have been developed to protect remotely configured systems from unauthorized use.

Common attacks include passive eavesdropping, active tampering, replay, and denial of service (DoS). For a passive eavesdropping attack, a malicious user taps the network to copy and analyze its traffic. If the attacker can see clear text control and configuration information, he or she may discover how to control and configure the system. In an active tampering attack, an unauthorized user attempts to gain control of the remote system by issuing bogus control packets. For a replay attack, a malicious user passively captures legitimate traffic and then attempts to change the operation of the system by resending the captured traffic at a later time. For an active DoS attack, the user paralyzes the system by overloading the network with massive amounts of traffic.

Remotely configurable network systems can be made safe by mechanisms that ensure confidentiality of data, provide authentication of the administrator, and guarantee integrity of the messaging. By encrypting messages with the Advanced Encryption Standard (AES) or other secure encryption algorithms, data confidentiality can be protected. With digital signatures generated by public key algorithms, the administrator of the system can be authenticated to guarantee that no one else attempts to modify its operation. The integrity of messages can be ensured by verifying that exactly what is transmitted by the administrator is received by the system. Use of a message authentication code (MAC) can assure users that data are not modified and that no additional control messages are inserted.

The Internet Protocol Security (IPSec) standard provides a mechanism to secure communications across the Internet. Many companies, such as Cisco, have implemented IPSec capability in their networking products. To secure a remotely reconfigurable FPGA, an IPSec in transport mode was designed for a Xilinx Virtex-II Pro FPGA [10]. Security policies at network access points defined who could gain access and under what conditions access was granted. Encryption keys and hash keys remained secret using the security services previously described. The Internet key exchange (IKE) protocol negotiated and exchanged shared secrets between communication entities.

34.6 SUMMARY

As the limits of processor clock scaling are reached, systems that route, process, filter, and transform Internet data scale better in reconfigurable hardware than in software alone. Networking platforms created with FPGA hardware are both fast and flexible. The FPX platform was used to implement over 30 core networking functions.

The combination of Gigabit network interfaces, parallel banks of SRAM and SDRAM, and a large array of reconfigurable logic on the FPX platform enabled it to perform a wide range of networking applications. Modules and protocol wrappers created in reconfigurable hardware were developed on the FPX and provided functionality similar to the procedures and DLLs in software for network processing. Reconfiguration of the modules over the network proved to be as effective for remotely loading new functionality on the FPX as the reprogramming of software on remote PCs.

By using IP wrappers, the FPX platform provided the ability to process ATM cells, AAL5 frames, IP packets, UDP datagrams, and/or TCP/IP flows. Parallel finite automata engines proved useful in detecting regular expressions in packet payloads and TCP traffic flows. Bloom filters that performed parallel hash lookups also proved to be effective for detecting fixed strings in packets and TCP flows. A complete IDS system was implemented that performed a large subset of SNORT using a combination of protocol-processing wrappers, IP header matching circuits, and Bloom filter payload-scanning circuits. A worm and virus detection and blocking system was built using an FPX that demonstrated its utility in providing Internet security.

Reconfigurable hardware holds great promise for new types of networking applications. A language detection circuit was demonstrated that routed traffic based on the language used in a document. A semantic-processing circuit was demonstrated that allowed documents to be classified based on their topic.

Going forward, reconfigurable hardware is becoming the technology of choice for future networking systems. Reconfigurable hardware is the key feature of a new platform, called the NetFPGA. This open platform enables switching and routing of network packets on Gigabit Ethernet links. Because the NetFPGA has many of the same resources as the FPX, it can implement most of the features first prototyped on the FPX [38, 39].

References

[1] J. W. Lockwood. Evolvable Internet hardware platforms. *NASA/DoD Workshop on Evolvable Hardware*, July 2001.

[2] J. W. Lockwood, H. Duan, J. M. Morikuni, S. M. Kang, S. Akkineni, R. H. Campbell. Scalable optoelectronic ATM networks: The iPOINT fully functional testbed. *IEEE Journal of Lightwave Technology*, June 1995.

[3] H. Duan, J. W. Lockwood, S. M. Kang, J. D. Will. A high-performance OC-12/OC-48 queue design prototype for input-buffered ATM switches. *IEEE Infocom '97*, April 1997.

[4] W. Marcus, I. Hadzic, A. McAuley, J. Smith. Protocol boosters: Applying programmability to network infrastructures. *IEEE Communications Magazine* 36(10), 1998.

[5] Wikipedia. OSI model. *http://wikipedia.org/wiki/OSI_model*, July 2006.

[6] F. Braun, J. W. Lockwood, M. Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable hardware. *IEEE Micro* 22(3), February 2002.

[7] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, D. B. Parlour. Scalable IP lookup for Internet routers. *IEEE Journal on Selected Areas in Communications* 21(4), May 2003.

[8] D. Schuehler, J. W. Lockwood. A modular system for FPGA-based TCP flow processing in high-speed networks. *Proceedings of the 14th International Conference on Field-Programmable Logic and Applications*, August 2004.

[9] T. S. Sproull, J. W. Lockwood, D. E. Taylor. Control and configuration software for a reconfigurable networking hardware platform. *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[10] J. Lu, J. W. Lockwood. IPSec implementation on Xilinx Virtex-II Pro FPGA and its application. *Reconfigurable Architectures Workshop*, April 2005.

[11] E. D. Horta, J. W. Lockwood, D. E. Taylor, D. Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. *Design Automation Conference*, June 2002.

[12] E. Horta, J. W. Lockwood. Automated method to generate bitstream intellectual property cores for Virtex FPGAs. *Proceedings of the 14th International Conference on Field-Programmable Logic and Applications*, August 2004.

[13] J. W. Lockwood, C. Neely, C. Zuver, D. Lim. Automated tools to implement and test Internet systems in reconfigurable hardware. *SIGCOMM Computer Communications Review* 33(3), July 2003.

[14] J. W. Lockwood, J. Moscola, D. Reddick, M. Kulig, T. Brooks. Application of hardware accelerated extensible network nodes for Internet worm and virus protection. *International Working Conference on Active Networks*, December 2003.

[15] J. Moscola, J. W. Lockwood, R. P. Loui, M. Pachos. Implementation of a content-scanning module for an Internet firewall. *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[16] R. Sidhu, V. K. Prasanna. Fast regular expression matching using FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[17] R. Franklin, D. Carver, B. L. Hutchings. Assisting network intrusion detection with reconfigurable hardware. *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[18] M. Roesch. Snort: Lightweight intrusion detection for networks. *Proceedings of the 13th Administration Conference, LISA*, November 1999.

[19] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, J. W. Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro* 24(1), January 2004.

[20] H. Song, S. Dharmapurikar, J. Turner, J. W. Lockwood. Fast hash table lookup using extended Bloom filter: An aid to network processing. *ACM SIGCOMM*, August 2005.

[21] M. Attig, J. W. Lockwood. SIFT: SNORT intrusion filter for TCP. *IEEE Symposium on High Performance Interconnects (Hot Interconnects-13)*, August 2005.

[22] L. Schaelicke, T. Slabach, B. Moore, C. Freeland. Characterizing the performance of network intrusion detection sensors. *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection*, September 2003.

[23] B. Madhusudan, J. W. Lockwood. A hardware-accelerated system for real-time worm detection. *IEEE Micro* 25(1), January 2005.

[24] D. Moore, C. Shannon, G. Voelker, S. Savage. Internet quarantine: Requirements for containing self-propagating code. *IEEE INFOCOM*, 2002.

[25] S. Staniford, V. Paxson, N. Weaver. How to own the Internet in your spare time. *Usenix Security Symposium*, August 2002.

[26] S. Singh, C. Estan, G. Varghese, S. Savage. *The Earlybird System for the Real-time Detection of Unknown Worms*, Technical report CS2003-0761, University of California, San Diego, Department of Computer Science, 2003.

[27] C. Estan, G. Varghese. New directions in traffic measurement and accounting. *ACM SIGCOMM*, August 2002.

[28] C. M. Kastner, G. A. Covington, A. A. Levine, J. W. Lockwood. HAIL: A hardware-accelerated algorithm for language identification. *Proceedings of the 15th Annual Conference on Field-Programmable Logic and Applications*, August 2005.

[29] Global Reach. Global Internet statistics by language. *http://www.glreach.com/globstats/index.php3*, December 2004.

[30] Global Reach. Global Internet statistics: Sources and references. *http://www.glreach.com/globstats/refs.php3*, December 2004.

[31] R. Paulsen, M. Martino. *Word Counting Natural Language Determination*, U.S. Patent 6,704,698, 1996.

[32] J. Schmitt. *Trigram-based Method of Language Identification*, U.S. Patent 5,062,143, 1990.

[33] M. Damashek. *Method of Retrieving Documents that Concern the Same Topic*, U.S. Patent 5,418,951, 1994.

[34] J. B. Sharkey, D. Weishar, J. W. Lookwood, R. Loui, R. Rohwer, J. Byrnes, K. Pattipati, D. Cousins, M. Nicolletti, S. Eick. Information processing at very

high-speed data ingestion rates. In *Emergent Information Technologies and Enabling Policies for Counter Terrosiom,* edited by R. Popp and J. Yin. IEEE Press/Wiley, 2006.

[35] J. W. Lockwood, S. G. Eick, D. J. Weishar, R. Loui, J. Moscola, C. Kastner, A. Levine, M. Attig. Transformation algorithms for datastreams. *IEEE Aerospace Conference,* March 2005.

[36] J. W. Lockwood, S. G. Eick, J. Mauger, J. Byrnes, R. Loui, A. Levine, D. J. Weishar, A. Ratner. Hardware accelerated algorithms for semantic processing of document streams. *IEEE Aerospace Conference,* March 2006.

[37] Y. H. Cho, J. Moscola, J. W. Lockwood. Context-free grammar based token tagger in reconfigurable devices. *Proceedings of the International Workshop on Data Engineering,* April 2006.

[38] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, J. Luo. NetFPGA—An open platform for Gigabit-rate network switching and routing. *IEEE International Conference on Microelectronic Systems Education (MSE2007),* June 2007.

[39] J. Luo, J. Pettit, M. Casado, N. McKeown, J. W. Lockwood. Prototyping fast, simple, secure switches for ethane. *IEEE Symposium on High-Performance Interconnects (Hot Interconnects-15),* August 2007.

CHAPTER 35

ACTIVE PAGES: MEMORY-CENTRIC COMPUTATION

Diana Franklin
Department of Computer Science
California Polytechnic State University

Although field-programmable gate arrays (FPGAs) excel at tailoring the computation and interconnect to an application's needs, we can go one step further. In many applications, regardless of the speed of the computation, memory performance always will be the limiting factor. This problem, referred to as the *memory wall*, is broken up into two parts—memory latency and bandwidth. For large-scale data-parallel applications, the computation can be moved to memory. This allows for both parallel computation and increased bandwidth. The replication of small computation units provides parallelism, and the sum of their memory ports provides increased bandwidth. Because they are located in memory, there is no shared-bus resource to serialize communication.

One such system, Active Pages, places computation with each page of DRAM. It is unique in that it targets the commodity DRAM market. This decision has both advantages and disadvantages. One advantage is that it supports both data streaming and general-purpose computation, and the computational resources scale automatically with memory allocation. One disadvantage is that, to keep costs low, there is no additional interconnect, and parallelism is only at the page level.

Many of the characteristics of Active Pages are present in any memory-centric system. This case study explores several characteristics of the Active Pages design. It begins, in Section 35.1, with an overview of the Active Pages architecture and programming model. Section 35.2 shows the performance potential of a scalable, memory-centric design. Section 35.3 then looks at how this scaling of computational resources, but not the interconnect resources, affects the asymptotic properties of several algorithms. Finally, Sections 35.4 and 35.5, explore the parallelism properties and the defect tolerance provided by the Active Pages design. Active Pages is just one of many projects in this realm, and Section 35.6 presents related work, followed by some conclusions in Section 35.7.

35.1 ACTIVE PAGES

This section gives a brief description of the Active Pages system. We present three aspects of the design: the hardware design, the interface between Active

Pages and the Central Processor, and the programming model that arises naturally from the design and interface.

35.1.1 DRAM Hardware Design

High-density DRAMs are divided into subarrays, complete with row and column decoders, to minimize column capacitance and decrease power consumption [1]. The proposed Active Pages implementation exploits this natural structure, treating each subarray as an Active Page. As shown in Figure 35.1, a small computational unit and cache—a *Page Processor* and *Page Cache*—are embedded next to each subarray to implement Active Page functions [2]. Using commodity 1-Gb DRAM technology as a target [3], we expect subarray size to be 512 KB and the embedded processing to consume less than 31 percent of the chip area.

To minimize DRAM modification and reduce hardware overhead, the Active Pages implementations do not provide hardware support for communication between Active Pages. If two Active Pages need to share data, the Central Processor reads the data from one and writes to the other. The disadvantage of this *process-mediated approach* is that interpage communication must be infrequent to maintain performance with a single processor.

35.1.2 Hardware Interface

To interface with the Central Processor, Active Pages leverage conventional page-based memory mechanisms to "virtualize" hardware for memory-based computation. Computations for each page can be suspended, restarted, and even swapped to disk. Computations for several pages can be multiplexed on a single embedded processing element.

Further, Active Pages use the same interface as conventional memory systems. Active Pages data are modified with conventional memory reads and writes; Active Pages functions are invoked through memory-mapped writes. Synchronization is accomplished through user-defined memory locations.

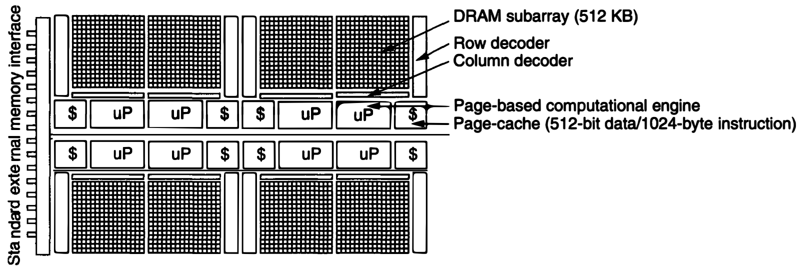

Standard external memory interface
$
uP
uP
$
$
uP
uP
$
$
uP
uP
$
$
uP
uP
$
DRAM subarray (512 KB)
Row decoder
Column decoder
Page-based computational engine
Page-cache (512-bit data/1024-byte instruction)

FIGURE 35.1 ■ The Active Pages architecture (8 pages).

35.1.3 Programming Model

The programming model of Active Pages was determined by several design decisions. First, communication between Active Pages and the Central Processor is accomplished through traditional reads and writes, allowing the Central Processor to operate on Active Pages data just as it does on any other data. Second, Active Pages were intended for commodity DRAM systems, which may be running general-purpose applications. Thus, we could not assume a traditional data parallel, streaming model. Third, there is no interconnect between Active Pages processors. The model needs to limit the Pages to their own data, with no knowledge of neighboring cells. Finally, each Active Page has computation associated with it. This is a direct association of data with computation. For these two reasons, the model of computation here is object-oriented programming.

To program a Page Processor, the programmer creates an object in C++. The choice of C++ is not critical; it is used because it has no runtime system associated with it and has well-defined interfaces for object manipulation. The 512 KB allocated to each Page Processor is divided between code, stack, and data. These 512 KB, larger-than-typical operating systems' virtual pages are referred to as superpages. The code must fit within the code segment, and the data size of the object is padded appropriately.

The operating system (OS) is responsible for allocating Active Pages memory and loading the code into the correct region. The Page Processor begins on activation, first performing any initialization similarly to a C++ object constructor, and then polling a variable waiting for an invocation of a function. To maintain pin compatibility, all Active Pages functions are designed to use conventional reads and writes. The Central Processor invokes Active Pages functions by writing the parameters into appropriate places in the Active Pages memory. The Central Processor then changes the `Running` variable, on which the Page Processor is polling, indicating which function to execute next.

When the Page Processor has completed the function, it resets the looping variable (`Running`) and waits for the next invocation. Figure 35.2 shows the object declaration and implementation for execution on a Page Processor for LCS. More details on the LCS algorithm can be found in Section 35.3.3. In the LCS algorithm, the application requires only a single function, so the event loop is not actually necessary. It is shown, however, to illustrate how an application with many functions would use the Central Processor to invoke functions on the Page Processors. The main function run on the Central Processor is not shown. The Central Processor can poll the `Running` variable to determine whether a Page Processor has completed a particular function.

35.2 PERFORMANCE RESULTS

Now that we have an idea of what the Active Pages architecture looks like and how it is programmed, this section presents performance results for several applications using a simulated Active Pages system. A more detailed study can be found in Oskin et al. [4].

```
Class LCS{
 //int    CodeAndStack[8192];    // added by compiler
 public:
   int Running, Data[WIDTH-1][LENGTH-1];
   char X[WIDTH], Y[LENGTH];
   LCS(){ Running = AP_WAIT; }
   void Start();
   void DoLCS();
} ;
void LCS::DoLCS() {
 int i, j;
 for(i=1;i<LENGTH;i++) // row 0, column 0 initialized by Central Processor
    for(j=1;j<WIDTH;j++) {
       if (X[i] == Y[j])
           Data[i][j] = Data[i-1][j-1] + 1;
       else if (Data[i-1][j] > Data[i][j-1])
           Data[i][j] = Data[i-1][j];
       else
           Data[i][j] = Data[i][j-1];
 }
}
void LCS::Start() {
   volatile int *act = &(Running);
   while(*act != AP_STOP) {
      while(*act == AP_WAIT) ;  // wait for Central Processor
      switch (*act) {
       case(AP_LCS):
          DoLCS(Val);
          *act = AP_WAIT;  // it is done
          break;
     }
   }
}
```

FIGURE 35.2 ■ A code example of an Active Pages object. Each Page Processor initializes its own space on allocation using the constructor. The Central Processor starts the Page Processor by writing to the `Running` variable. When the call is finished, the Page Processor sets `Running` back to `AP_WAIT`.

To estimate the performance of Active Pages configurations, each Active Pages function was hand-coded in a high-level circuit-description language, such as VHDL (see Chapter 6 and [5]), and synthesized to an Altera 10K FPGA. The mapping was carried out all the way to placed and routed designs [6].

To demonstrate effective partitioning of applications between the Central Processor and Active Pages, we chose a range of applications representing both memory- and processor-centric partitioning. Table 35.1 summarizes the attributes of these applications.

35.2.1 Speedup over Conventional Systems

To evaluate performance of the Active Pages memory system, each application was executed on a range of problem sizes. The speedup of the applications

TABLE 35.1 ■ Summary of the partitioning of applications between the Central Processor and Active Pages

Name	Application	Central Processor computation	Active Pages computation
		Memory-centric applications	
Array	C++ standard template library array class	C++ code using array class cross-page moves	Array insert, delete, and find
Database	Address database	Initiates queries summarizes results	Searches unindexed data
Median	Median filter for images	Image I/O	Median of neighboring pixels
Dynamic program	Protein sequence matching	Backtracking	Compute MINs and fills table
		Processor-centric applications	
Matrix	Matrix multiply for Simplex and finite element	Floating-point multiplies	Index comparison and data gathering and scattering
MPEG-MMX	MPEG decoder using MMX instructions	MMX dispatch Discrete cosine transform	MMX instructions

running on an Active Pages memory system compared to a conventional memory system is shown in Figure 35.3. Each application was run on a range of problem sizes, given in terms of number of Active Pages (512-KB superpages). The following are two primary observations about this graph.

First, the performance results qualitatively scale as expected. This shows the advantage of memory-centric computation. We observe that most applications show little growth in speedup as data size grows within the subpage region (below one page). In this region, Active Pages applications have little parallelism to offset activation costs. When leaving this region, however, we enter the scalable region and see that performance on all applications grows as data size increases. Four applications—database, MMX, matrix-simplex, matrix-boeing, and median-filtering—also reach the saturated region. Here, Active Pages performance is limited by the progress of the Central Processor. This limitation may be because of either too much work for a given-speed Central Processor or too much data travelling between the Central Processor and Active Pages across the memory bus. Performance can actually decrease as coordination costs dominate performance. Given a large enough problem size, all applications would eventually reach the saturated region.

Second, we see that the array-delete primitive performs poorly in the subpage region. This is because of the difference between the FPGA implementation and the instruction set used to implement the Central Processor. The Central Processor's instruction set is especially well suited for the array-delete primitive. Thus, unless there is sufficient parallelism to justify using Active Pages, it is faster to use the Central Processor. So, for small deletes, we use only the Central Processor. This benchmark was a combination of small deletes and large deletes.

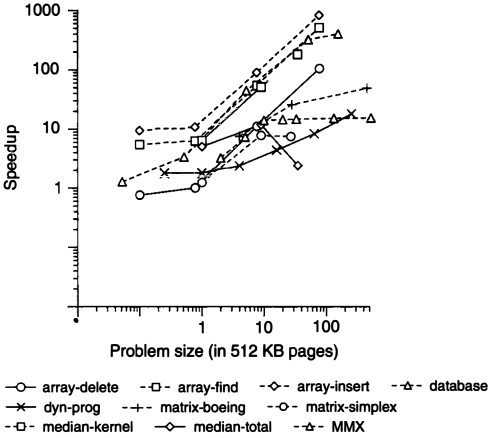


FIGURE 35.3 ■ Active Pages speedup as problem size varies.

As problem size grows, and the Central Processor is used for both the coordination of large deletes and the complete execution of small deletes, the Central Processor becomes the limiting factor in performance and the performance gets closer to that of the uniprocessor. This shows an interesting tradeoff between the FPGA and the Central Processor. Some computations, though not many, will perform better on the Central Processor. If this coincides with a part of the application that does not require parallelism, then the advantage of the memory-centric FPGA implementation will be reduced.

35.2.2 Processor–Memory Nonoverlap

The saturated region of Active Pages performance emphasizes the importance of partitioning applications to efficiently use the Central Processor in a system. For processor-centric applications, this dependence is obvious. The goal is to keep the Central Processor computing by providing a steady stream of useful data from the memory system. For memory-centric partitions, however, the Central Processor is still a vital resource. Active Pages cannot compute without activation and interpage communication, both provided by the Central Processor.

As data size grows in an Active Pages application, so does the load on the Central Processor. We measure the remaining capacity of a Central Processor to handle this load with a metric, *processor–memory nonoverlap* time. Nonoverlap is the time the Central Processor spends waiting for the memory system and can be used to estimate the boundary between the scalable and saturated regions of application performance.

The relative percentage of time the Central Processor is stalled, waiting for memory system computation, is shown in Figure 35.4. As described in

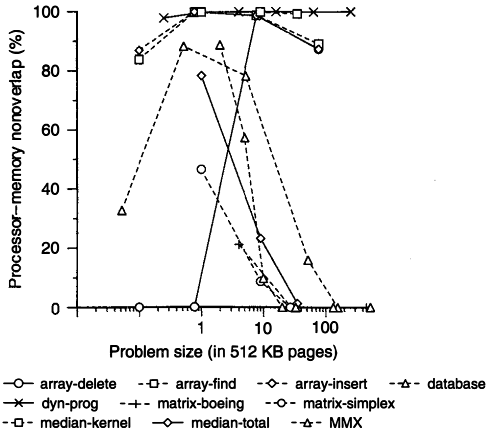


FIGURE 35.4 ■ The percent of cycles that the Central Processor is stalled on Active Pages as problem size varies.

the previous section, the applications that reached the saturated region of speedup were database, matrix-simplex, matrix-boeing, and median-filtering. As Figure 35.4 shows, these applications also reach a point of complete processor–memory overlap.

We also observe that for the array primitives and the dynamic programming application, the nonoverlap percentage remains relatively high. These applications are largely memory-centric with very little Central Processor activity. In fact, the array primitives operate asynchronously to the end of the application and are artificially forced into synchronous operation for this study. This means that an application can use the array-insert and array-delete primitives with only the cost of Active Pages function invocation. Modulo dependencies on the array, the time spent by the memory system shifting data, can be overlapped with operations outside of the STL array class. This overlap occurs in a natural way with no additional effort required by the programmer who uses the Active Pages STL array class. Opportunities for overlapping execution of data structure operations with data structure usage are intriguing and are being investigated further.

The dynamic programming example maintains a very high processor–memory nonoverlap; however, preliminary results indicate that processor-mediated communication required by the Active Pages memory system eventually dominates performance. This occurs for extremely large problems that are well beyond the range of problem sizes presented in this study. Dedicating more resources to the interconnect increases the range of problems that Active Pages can help solve.

35.2.3 Summary

Memory-centric computation provides a scalable source of performance for large-scale applications. Active Pages provides a large number of simple, reconfigurable computational elements that can achieve speedups up to 1000 times faster than conventional systems. Systems with rich interconnects have the potential for scalable gains on an even wider range of applications.

35.3 ALGORITHMIC COMPLEXITY

Although the simulated results show great promise, to truly understand how Active Pages improves runtimes as problem sizes grow, we need to explore asymptotic properties of algorithms in conventional systems as well as Active Pages systems [7]. For this study, we use a set of kernels whose asymptotic properties are well known in algorithmic literature.

While it is unrealistic to expect the number of processors in a conventional multiprocessor to scale arbitrarily, the amount of DRAM in a system is expected to scale with problem size for a majority of problems. With Active Pages DRAMs, computational hardware also scales. This scaling provides parallelism that can improve asymptotic performance. Table 35.2 gives a preview of such gains for a variety of algorithms. Note that Active Pages execution times rely on the optimal page size given in the table. In practice, we expect Active Pages hardware to support a small range of page sizes designed to support target applications and problem sizes.

The challenge in the analysis is to take communication costs into account. In any system, the interconnect will affect the asymptotic properties of the performance as the problem scales. Active Pages, in particular, requires careful consideration of the communication between Page Processors as well as between the Central Processor and the Page Processors. The partitioned computations and restricted communication model here differ substantially from traditional parallel models such as PRAM [8]. This section presents an analysis of each algorithm that considers these issues. These analyses are also validated with simulation results.

TABLE 35.2 ■ Algorithmic complexity (summary)

		Execution time	
Application	**Conventional**	**within Active Pages**	**Page size**
Array insert	$O(n)$	$O(\sqrt{n})$	$O(\sqrt{n})$
2D LCS	$O(n^2)$	$O(n\sqrt{n})$	$O(n)$
3D LCS	$O(n^3)$	$O(n^{7/3})$	$O(n^2)$
All-pairs shortest path	$O(n^3)$	$O(n^{7/3})$	$O(n^{4/3})$
Sorting	$O(n \cdot \log_2(n))$	$O(n \cdot \log_2(\log_2(n)))$	$O(n/z)$ where $n = z \cdot e^z$
Volume rendering	$O(n^3)$	$O(n^{5/2})$	$O(n^{3/2})$

35.3.1 Algorithms

Active Pages can dramatically improve the performance of many algorithms. This section maps several common algorithms to an Active Pages system and analyzes performance gains. Figure 35.5 introduces the notation used here. With these conventions, we analyze the worst-case execution time of the algorithms: insertion of an element into a linear array of elements, longest common subsequence of two- and three-dimensional sequences using a dynamic programming formulation, all-pairs shortest path using a dynamic programming formulation, sorting of a linear array of elements, and volume rendering using ray-tracing and linear absorption coefficients [7,9].

Each analysis is provided by first presenting a general model for the algorithm's execution time. Next, various model-specific parameters are assumed to be constants. After this simplification, the derivative of execution time with respect to page size is used to find an optimal page size. This page size is then substituted back into the model, and execution time is expressed again as a function of problem size.

These results are then validated with a high-level simulator. The simulator models Active Pages execution using parameters based on execution of the cycle-level simulator. The parameters used are given in Table 35.3. *Typical* parameters correspond to the target architecture studied here and often exhibit better performance than a purely asymptotic analysis would suggest. *Asymptotic* parameters emphasize the dominant terms in asymptotic performance while remaining within realistic problem sizes. These exaggerated parameters are used to validate the more conservative analyses.

Table 35.3 summarizes the parameters used in the high-level simulator. T_a is the amount of time required by the processor to invoke a function on a memory-based

n is the size of the input.

p is the number of data elements in an Active Page.

q is a problem-specific function of p that is used for most algorithms to define p. For instance, for dynamic programming algorithms where a two-dimensional result set is generated, it is convenient to describe p as equal to $p=q^2$.

k is a function of the number of Active Pages used for the problem—usually $k=n/q$.

FIGURE 35.5 ■ The notation used for algorithmic analysis.

TABLE 35.3 ■ Summary of simulation parameters

Parameter	APSP*	Sort	Array insert	LCS*	LCS3	Render
Activation time (T_a)	100/0	0	2058	100/100	100	100
Central Processor per-page processing time (T_p)	–	1	387	–	–	5
Page processing per-element processing time (T_c)	10/10	1	2	–	10	10
Fixed communication overhead (T_{sa})	1/1	–	–	10/10	1	–
Per-element communication cost (T_{sb})	1/1	–	–	1/100	1	–

* Typical/asymptotic.

processor. This includes setup, argument passing, and invocation. This constant is per page. T_p is the amount of time required by the processor to complete execution of an algorithm associated with a particular page. Generally, the "focus" of execution traverses from the Central Processor to the Active Pages and then back again. This may proceed many times and involve overlap throughout the execution of the algorithm. However, for the analysis presented here the focus is on a single set of transitions from host to memory and back. Hence, T_p is the time spent by the Central Processor per page when completing the Central Processor portion of the computation for that page. T_c is the amount of time required by the memory-based processing element to compute its portion of the algorithm for a single data item within the page. For instance, on a conventional processor and memory system, an $O(n)$ algorithm requires some time, T_c, to compute the solution for each element; hence, the execution time is described as $T = T_c \cdot n$. T_{sa} is the amount of time that corresponds to the "fixed overhead" associated with each interpage communication. Inter–Active Pages communication is a necessarily expensive process, and this constant quantifies the relatively large fixed overhead associated with each such communication request. T_{sb} is the amount of time, per data item, associated with an interpage communication. Not all algorithms use interpage communication, and some use portions of T_a or T_p to perform such communication as part of activation and postprocessing, respectively.

This short section can present detailed analysis only of the array and LCS applications. We refer the reader to a technical report by Oskin et al. for the full set of analyses and results [9].

35.3.2 Array-Insert

The analysis begins with a simple array library. Specifically, we examine an insertion operation performed on an array of elements arranged in a linear fashion. A conventional system requires $O(n)$ execution to complete this task. In an Active Pages memory system, we partition these n elements into k pages, with each Active Page managing n/k elements. To insert an element at position j within the array, each Active Page from the page containing j up to the last page of the array shifts the elements up by one to make room for the new element. These shifts proceed in parallel, however, since each Active Page operates independently. Note, though, that some form of communication between pages is required to migrate elements across page boundaries. This communication is grouped within the activation portion of each Active Page. The algorithm can be expressed as shown in Figure 35.6.

```
for j=1 to k
   communicate the last element of
     page j to page j+1
   activate page j informing it to
     shift elements upward
```

FIGURE 35.6 ■ The array-insert algorithm.

The analysis begins with $s(i)$, the nonoverlap (*stall*) time for page i. The nonoverlap time, discussed in Section 35.2.2, is the amount of time spent by the processor waiting for the Active Pages memory system to finish. Essentially, this algorithm (and many other Active Pages algorithms) proceeds by having the Central Processor set up and activate memory-based processing, then wait for a page to complete computing. After the memory-based computation section is complete, the processor can return to finish its section of the computation. It turns out that quantifying how much a processor stalls while waiting for memory-based computation to complete, for traditionally linear algorithms, is an important and measurable quantity that can be used to tune applications to achieve maximum performance. We use it to quantify execution time.

Three functions—$T\hat{a}, T\hat{p}$, and $T\hat{c}$—are used to quantify portions of the execution time. These are expressed as functions because several linear-based algorithms can be mapped to an execution time analysis similar to that presented here. The functions correspond to activation time, host processor postexecution time, and per-page memory-based computation time, respectively.

For array insertion, these are essentially constant functions; hence, $Tc(i) = T_c, Ta(i) = T_a$, and $Tp(i) = T_p$. Figure 35.7 shows the timing of the array-insert operation (or any other linear-based function) on the Active Pages system using Ta, Tc, and Tp. Next, note that $\sum_{i=1}^{k} s(i) \leq T_c \cdot p$ allows us to simplify execution time and take the derivative of T with respect to p. This gives us a new expression for T given the optimal value for p:

$$
\begin{aligned}
T &= \sum_{i=1}^{k} \left[T_a + T_p + s(i)\right] = k\left(T_a + T_p\right) + \sum_{i=1}^{k} s(i) \\
&\leq k\left(T_a + T_p\right) + T_c \cdot p = \frac{n}{p}\left(T_a + T_p\right) + T_c \cdot p \\
\frac{dT}{dp} &= \frac{-n}{p^2}\left(T_a + T_p\right) + T_c \Rightarrow p = \sqrt{\frac{n\left(T_a + T_p\right)}{T_c}} \\
T_{opt} &= \frac{n}{p}\left(T_a + T_p\right) + T_c \cdot p = 2 \cdot \sqrt{n \cdot \left(T_a + T_p\right) \cdot T_c} = O(\sqrt{n}) \qquad (35.1) \\
T &= \sum_{i=1}^{k} \left[Ta(i) + Tp(i) + s(i)\right] \\
s(i) &= \max \begin{cases} 0 \\ s'(i) \end{cases} \\
s'(i) &= Tc(i) - \left(\sum_{j=i+1}^{k} Ta(j) + \sum_{j=1}^{i-1} \left(Tp(j) + s(j)\right) \right)
\end{aligned}
$$

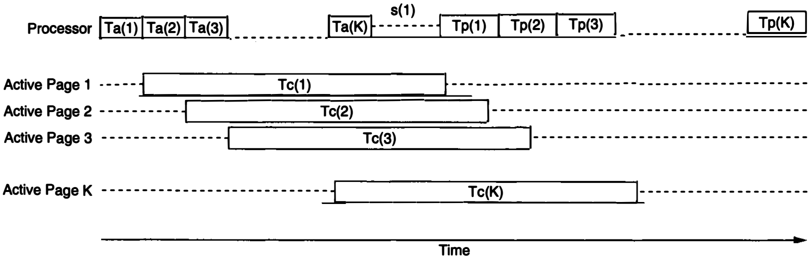


FIGURE 35.7 ■ An array-insert operation demonstrating processor and Active Page computations.

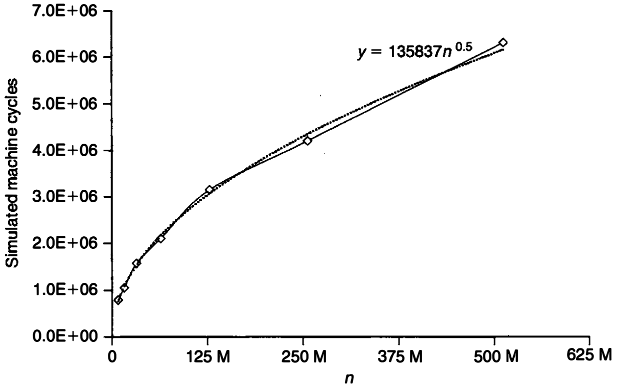


FIGURE 35.8 ■ Simulation results for the array-insert operation.

This analysis makes the conservative assumption that computation proceeds in serializable steps. First, all pages are activated; then all pages compute; finally, all pages finish and the processor performs some minimal postpage computation for each page. In reality, there is substantial overlap of these functions, and only during asymptotic performance is this serializing behavior observed. During practical application of this algorithm, the dominant term is $T_C \cdot p$, and execution time is held relatively constant. This behavior is observed until the point at which the number of pages times the activation and postpage processing per page starts to significantly approach $T_C \cdot p$. Figure 35.8 depicts simulated application performance versus problem size. As can be seen from the graph, simulated performance follows an $O(\sqrt{n})$ growth curve, as predicted by the analytical model here.

35.3.3 LCS (Two-dimensional Dynamic Programming)

Moving to a more complex algorithm, we examine a dynamic programming formulation for computing the longest common subsequence in a protein. The conventional execution time of this algorithm is $O(n^2)$. Figure 35.9 outlines the algorithm. For a more in-depth discussion of the LCS algorithm with fine-grained parallel execution in a systolic model, see Hoang [10].

Parallel execution of this algorithm proceeds in "wave-fronts," as depicted in Figure 35.10. Once the first subproblem is solved and the results have been dispatched, two other problems can immediately start computing, and when they are done, three other Active Pages can start their computation in parallel. The processor is responsible for activating a wave-front. When processor-mediated communication is used, the wave-front is uneven, with certain pages of the computation executing slightly ahead of other pages. This is because of the overlapping nature of Active Pages computation and processor activity. In the model of computation here, this overlap is very important to performance, and we take advantage of it to lower overall execution time. Also note that the subproblem solution that an Active Page will make available consists only of the items on two edges of the page.

For this problem we assume the following constants. T_c is the time required by the Active Pages processor to compute the result of a single item of the LCS computation. T_{sa} is the fixed overhead cost associated with an interpage communication. T_{sb} is the cost to transfer items between pages on a per-item basis.

```
partition x and y into k segments
divide the computation into x/q and y/q smaller computations
initialize page (i,j) with the corresponding component i of string x
                   and with component j of string y.
let page (i, j) perform the conventional LCS algorithm after subproblems
                 (i, j-1), (i-1, j), and (i-1, j-1) have been solved.
page (i,j) dispatches results to neighboring subproblems.
```

FIGURE 35.9 ■ The two-dimensional LCS algorithm.

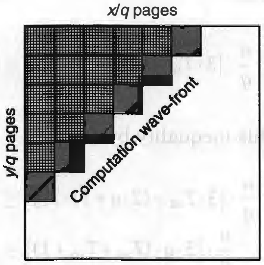


FIGURE 35.10 ■ Parallel execution of two-dimensional LCS on Active Pages.

Further, since the dynamic programming model dictates that the number of items in a page be quadratic in terms of the length of sequence x and the length of sequence y, we define the page size p to be equal to q^2, where q is a variable.

This makes the reasonable analytical assumption that x and y are of similar lengths. We can express application execution time as

$$T < 2 \cdot \sum_{i=1}^{j} \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] + 2 \cdot \sum_{i=j+1}^{n/q} i \cdot \left[3 \cdot T_{sa} + (2 \cdot q + 1) \cdot T_{sb}\right] \qquad (35.2)$$

where j represents the particular wave-front in which the overall execution switches from being bounded by computation to being bounded by communication. Focusing on the first half of the computation-bound area, each wave-front has an ever-increasing cost of communication. This is because more Active Pages are involved in each wave-front.

At first, the communication is hidden by computation, but eventually the cost of communicating the required data between wave-fronts exceeds the cost of computation for the wave-front. At this point, the algorithm crosses over from being bounded by computation to being bounded by communication; thus, computation completely overlaps with communication. We denote the wave-front where this occurs as j. This chapter presents an analysis that achieves a better theoretical upper-bound than the conventional sequential solution. Based on particular protein sequence sizes, computer-assisted analysis can reveal the ideal j and q, which minimize the execution time of this algorithm, thus tailoring the behavior of Active Pages in terms of the given problem size. The simulation results show that computer-calculated ideal page sizes entail even a slightly better performance than the theoretical analysis. As will be seen, this is because of a simplification in the analysis.

Suppose we force $j \geq n/q$. This implies that the algorithm will never become bounded by communication resources. We can do this by carefully selecting q and then demonstrating that this q does indeed force $j \geq n/q$. To find a q that satisfies these conditions, we require that the communication always weighs less than computation:

$$\frac{n}{q} \cdot \left[3 \cdot T_{sa} + (2 \cdot q + 1) \cdot T_{sb}\right] \leq \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \qquad (35.3)$$

Then simplify this inequality by:

$$\begin{aligned} \frac{n}{q} \cdot \left[3 \cdot T_{sa} + (2 \cdot q + 1) \cdot T_{sb}\right] &\leq \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \\ \frac{n}{q} \cdot \left[3 \cdot q \cdot (T_{sa} + T_{sb} + 1)\right] &\leq \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \qquad (35.4) \\ T_c \cdot q^2 &\leq \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \end{aligned}$$

This simplification will not lead to an absolute lower-bound on execution time, but it does present a tractable alternative that can be used to find an "ideal" q:

$$q \geq \sqrt{n} \cdot \sqrt{\frac{3 \cdot (T_{sa} + T_{sb} + 1)}{T_c}} = \alpha \cdot \sqrt{n} \tag{35.5}$$

Then use this q to drop j from the equation, since the algorithm will never be bound by communication:

$$\begin{aligned} T &< 2 \cdot \sum_{i=1}^{n/q} \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] = 2 \cdot \frac{n}{q} \cdot \left[T_c \cdot q^2 + T_{sa} + q \cdot T_{sb}\right] \\ &= 2 \cdot \frac{\sqrt{n}}{\alpha} \cdot \left[T_c \cdot n \cdot \alpha^2 + T_{sa} + \sqrt{n} \cdot T_{sb} + \alpha\right] = O(n\sqrt{n}) \end{aligned} \tag{35.6}$$

While $O(n\sqrt{n})$ is a loose upper-bound, it is faster than the conventional runtime of $O(n^2)$. The simulation results concurred with the findings and suggested a slightly better than $O(n\sqrt{n})$ lower worst-case execution bound.

Figure 35.11 depicts simulated performance of the LCS algorithm; two curves are shown. The first curve depicts the predicted performance of $O(n\sqrt{n})$ (using *asymptotic* parameters from Table 35.3). The second curve predicts a more realistic performance of $O(n^{4/3})$ (using *typical* parameters). The discrepancy is because of communication performance. If communication were more expensive, then the ideal page size would shift away from communication requirements and toward increased computational requirements, amplifying that term in the execution time expression. This in turn would reveal the asymptotic order of the LCS algorithm.

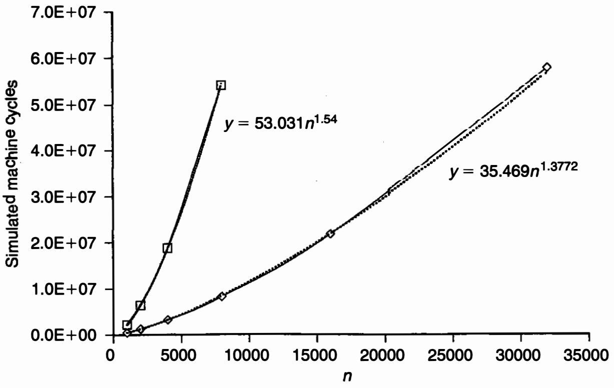


FIGURE 35.11 ■ Simulation results for the two-dimensional LCS.

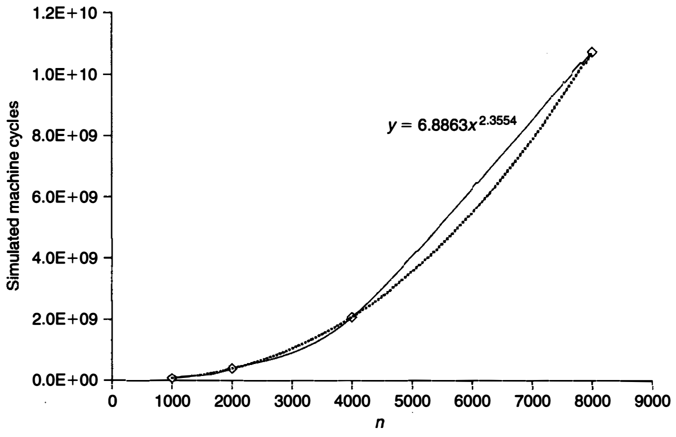


FIGURE 35.12 ■ Simulation results for the three-dimensional LCS.

A more realistic depiction of application performance follows an $O(n^{4/3})$ trend. A similar analysis predicts performance of $O(n^{7/3})$ for three-dimensional LCS. Figure 35.12 shows that the simulated performance for three-dimensional LCS closely matches this prediction.

35.3.4 Summary

We can see that with a memory-centric architecture such as Active Pages, in which the computation scales with the communication, the asymptotic complexity can be reduced. We also see that it is a much more complex equation than one might think. The overhead of the Active Pages, the delay of any communication, and the page size need to be taken into account. Two algorithms, along with validated simulations, have been presented to show their new asymptotic properties. We have found that the inexpensive parallelism provided by page-based intelligent memories can have a significant affect on asymptotic performance. We have also found the optimal page sizes that are required to maximize performance.

35.4 EXPLORING PARALLELISM

In any memory-centric system, we must decide the proper balance between memory resources and computation power. To save money, we could share a single computational element with twice as much memory. Allowing sharing can potentially even out the computational requirements of two processing elements because their needs may not always be identical.

This section looks at virtualizing the computational logic across superpages in the Active Pages chip. Virtualization is accomplished by time-slicing a VLIW processor (see VLIW datapath control subsection of Section 5.2.2) across one to eight Active Pages. We refer to this time-slicing as the *multiplexing* of the computational logic. This study presents an analysis of multiplexing and its effects on performance in a multiprocess environment. In addition, it looks at how varying individual processor widths affects performance. By combining these approaches, we demonstrate that multiplexing is a more effective technique for reducing logic area requirements than reducing individual Page Processor performance.

In this study, we chose to use VLIW computational elements rather than an FPGA so that we could explore the trade-off between instruction-level parallelism and task-level parallelism. The results hold for FPGAs as well. From a high level, it is merely the trade-off between smaller dedicated resources per memory segment and shared resources between memory segments. The study is cleaner when using processor width rather than FPGA area.

35.4.1 Speedup over Conventional

We begin with the raw speedups of a commodity workload that is used for this study. Because the focus is on multi-programmed systems, we are using a slightly different workload than before.

Figure 35.13 depicts application speedup when applications use an Active Pages memory system. Speedup is measured in terms of wall-clock time for the application in a conventional memory system divided by its wall-clock time

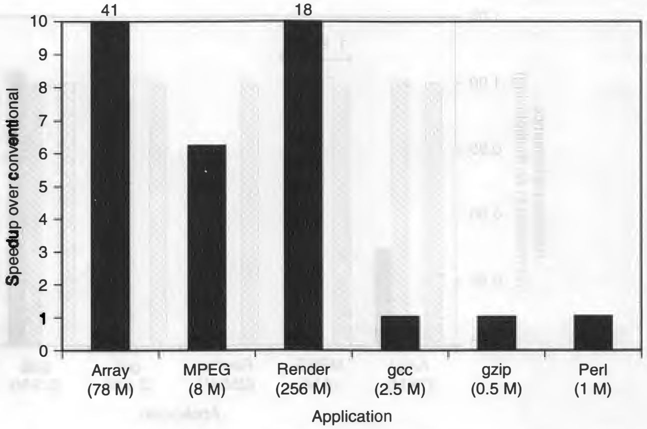


FIGURE 35.13 ■ Speedup over conventional.

using an Active Pages memory system. We observe that Active Pages applications continue to show substantial speedups when executed in a multiprocess environment. That is, even when many independent applications are executed at once, the applications experience speedup.

35.4.2 Multiplexing Performance

We continue by exploring how much performance degradation occurs as resources are shared between Active Pages. Figure 35.14 depicts relative application performance as the degree of multiplexing is increased. We normalize the results to a configuration with no multiplexing, where a one-to-one relationship exists between 4-wide VLIW processors and DRAM subarrays. Multiplexing factors of two, four, and eight make up the remaining data points. Note that hardware multiplexing of eight incurs no more than a 17 percent performance penalty, and a multiplexing factor of four incurs no more than a 6 percent performance penalty for all Active Page applications in the workload.

35.4.3 Processor Width Performance

It is promising that with a 4-wide VLIW, performance does not degrade substantially, as it is shared between Active Pages. Is this because the VLIW processor is not being used efficiently? We now examine the inherent instruction-level parallelism (ILP) in our applications. Figure 35.15 depicts relative application performance as VLIW processor width is varied. Here, processor widths of one, two, four, and eight were evaluated. We observe that half of the applications show a 20 to 80 percent increase in performance from increasing processor width, but the other half do not. It should be noted that MPEG suffers adverse

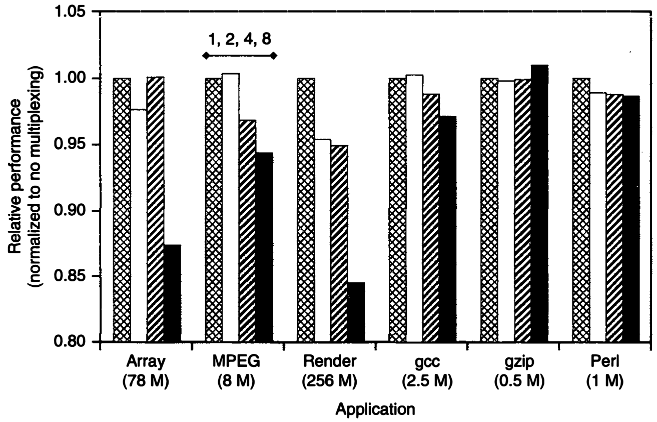


FIGURE 35.14 ■ Performance versus hardware multiplexing.

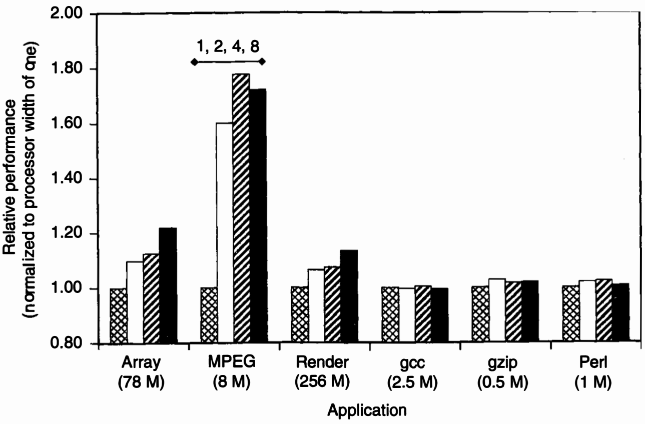


FIGURE 35.15 ■ Performance of multiplexing versus VLIW processor width.

cache effects with a VLIW width of eight, thus lowering performance relative to a 4-wide VLIW. We note that the largest performance gains because of VLIW processor width are achieved with processor widths of two and four, and not with eight.

35.4.4 Processor Width versus Multiplexing

Taking another look at Figure 35.15, we find that the Active Pages applications do not have the static instruction-level parallelism to use much beyond a 4-wide VLIW processor. In addition, Figure 35.14 shows that degradation because of multiplexing is superlinear, suggesting that too much coarse-grained parallelism exists within the application workloads to substantially multiplex processor resources.

An experiment designed to compare these two forms of parallelism is depicted in Figure 35.16. Here we compare an Active Pages device using a single-issue processor with no multiplexing against a device using a 2-wide VLIW with two-way multiplexing, a 4-wide VLIW with four-way multiplexing, and an 8-wide VLIW with eight-way multiplexing.

In the Active Pages applications, a 2-wide VLIW with two-way multiplexing shows a performance gain. This implies that the gain from the increased ILP outweighs the reduced coarse-grained parallelism. Because several conventional applications are active in the workloads, this makes sense because many of the pages do not need the page processors. A 4-wide VLIW with four-way multiplexing is the best configuration studied. Hence, we use this configuration in the remainder of this study.

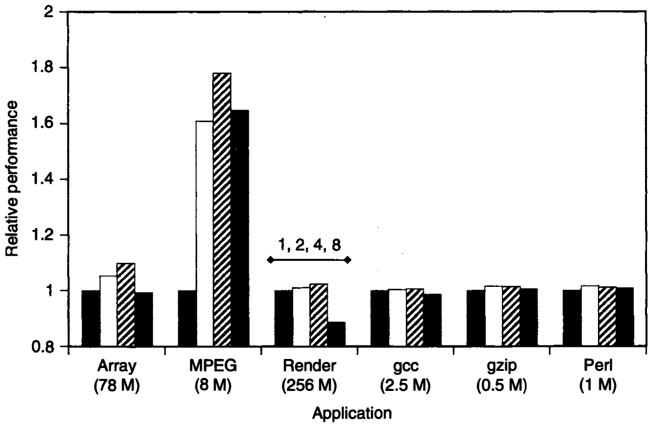


FIGURE 35.16 ■ Performance versus processor width.

To describe why multiplexing performs well in a multiprocess environment, we identify three key factors: nonactive memory, Active Pages processing time, and partitioning.

Nonactive memory
This helps mask the performance degradation because of multiplexing. By definition, all pages of memory in a conventional application require no computation in memory. Some pages in an Active Pages application also require no memory computation.

Active Pages processing time
This is the amount of time spent by the Active Pages computing without main processor intervention. The time varies with Page Processor performance. Simple data manipulations are easily offloaded to the memory system. This leads to longer per-page computation times, most notably MPEG, with Active Pages processing time on the order of seconds.

The combination of low Active Pages processing times and context switching in the Central Processor hides the effects of multiplexing in the memory system. In the absence of multiplexed Active Pages, when the main processor switches to another process, the Active Pages associated with the previous process quickly finish their work and stall until the process regains control of the Central Processor. Multiplexing allows efficient utilization of Page Processors by context-switching them to another Active Pages process when they would otherwise be idle.

In an environment with Active Pages processing times longer than a Central Processor time slice, such as those observed in MPEG, we would expect multiplexing to degrade performance. Within this study, however, degradation is minimal due to the relatively low memory requirements of MPEG and the effects of conventional memory (without computational capability).

Partitioning

This is the process of dividing an application into work done in Active Pages and work done in the Central Processor. As long as the main processor can keep up with the Active Pages, an application is *scalable* and will exhibit linear speedup as its dataset grows and more Active Pages are used. Once the main processor becomes *saturated* with work, however, performance will no longer increase as more Active Pages are used.

We find that multiprocess environments change the position at which an application transitions from scalable to saturated. Multiprocessing time slices the Central Processor, which may be viewed as artificially slowing down the processor from the perspective of a single process. This will shift the scalable-saturated point toward smaller problem sizes. We may use multiplexing to reverse this shift. Essentially, multiplexing slows down the Active Pages computation, shifting the scalable-saturated point back toward larger problem sizes.

Because of the preceding properties of multi-programming environments, we observe that multiplexing is an efficient mechanism for reducing logic area requirements in an Active Pages memory device. A four-way multiplexed 4-wide VLIW Active Pages device is estimated to require 12 percent of the available chip area for computational logic while still providing substantial performance gains. This estimate is based on the reduced logic area coupled with a 20 percent logic area increase because of additional interconnect requirements.

35.4.5 Summary

This study has looked at a promising method for reducing the computational logic area requirements of an Active Pages memory device. Such an approach could be exploited by any memory-centric device. By multiplexing the computational logic among one to four Active Pages, hardware cost can be reduced by four times with little performance impact in a multiprogrammed environment. Further, we find that it is more important to have fewer, faster computational logic elements that are time-shared across pages than more abundant, slower ones available for direct computation at each page. With a 4-wide VLIW processor multiplexed with every four Active Pages, computational logic area can be reduced to 12 percent of total chip area in a gigabit DRAM.

35.5 DEFECT TOLERANCE

The previous section explored the parallelism trade-offs gained by sharing computational units between pages. This section focuses on another major factor in cost: manufacturing defects. DRAM architectures use redundant cells to tolerate defects, dramatically increasing chip yields and reducing cost. Embedded processors, however, do not have an analogous unit of redundancy. While multiplexing several Active Pages with one embedded processor reduces chip area, multiplexing each group of pages with two processors allows each group to tolerate a processor defect. This *associativity* requires some additional interconnect, but tolerance to randomly distributed processor defects increases from 33 percent to more than 50 percent.

In this section, we use associativity to increase the defect tolerance of an Active Pages system. The focus is on manufacturing defects that render embedded processors inoperative. The goal is to provide some degree of processor redundancy under the assumption that memory cells already have their own redundancy techniques.

Instead of four Active Pages sharing one 4-wide VLIW processor, we allow eight pages to share two processors. We study the effect of randomly distributed processor defects on this associative system. If a group suffers two defects, the operating system will only map conventional pages to that group (pages with no computation).

The performance degradation because of randomly distributed processor defects is depicted in Figure 35.17. We note that up to a 50-percent defect rate is tolerated. Increasing the defect rate to 60 percent decreased the number of functional Active Pages below that required by the workload without page swapping. Virtualizing Active Pages to disk was studied by Oskin et al. [11], and a similar mechanism can be used to further increase defect tolerance.

Associativity creates an increased tolerance to defects. The benefits are straightforward. Two processors must fail instead of one in order to disable any Active Pages. If 50 percent of embedded processors fail in the test system, we see that with two-way associativity up to 75 percent of the memory will still be available for Active Pages use.

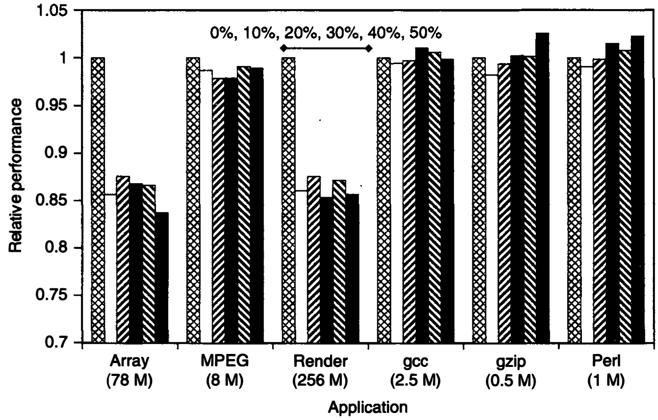


FIGURE 35.17 ■ Performance versus random processor defects.

Second, not all of the system memory is required to be "active" at the same time. This allows the OS to map around defect areas and use fully defective functional groups for conventional applications. Further, the workloads do not require the full 512 MB available to the system, and the unutilized memory is available to map into defective regions. The OS can tolerate some defects without associativity by taking advantage of underutilization and conventional applications.

As noted in this section, multiplexing, associativity, and clever OS resource allocation can map around manufacturing defects with only a 20 percent performance penalty with 50 percent random logic defects. An Active Pages–aware OS can be defect tolerant and allow a lower-cost system to be developed by increasing manufacturing chip yield. These incremental costs make Active Pages an attractive memory-based computation model, though the same principles would hold for FPGA-based systems (see Chapter 37).

35.6 RELATED WORK

DRAM densities have made intelligent memory attractive as commodity components. Intelligent memory, however, was proposed well before the current commodity thrust. The SWIM project [12] combined reconfigurable logic and memory to perform fast protocol computations. The J-Machine integrated processor, memory, and network router in a single chip to form building blocks for a fine-grained multiprocessor [13]. The RAW [14], MORPH [15], and RaPiD [16] projects continue to explore the use of reconfigurable technology to exploit parallelism. The RAW project, in particular, has also examined issues of processor width, dynamically trading off ILP and speculation. The HPAM project [17] takes a hierarchical approach to intelligent memory.

The project that is most similar to Active Pages is FlexRAM [18], which targeted general-purpose computation. The goal was to find computation that could take advantage of the bandwidth provided within a DRAM chip. FlexRAM proposed a hierarchical solution with simple computational elements within each page and a more complex processor for each DRAM. This allowed communication to be handled by an on-chip processor rather than the Central Processor. This had the disadvantage of adding pins to commodity DRAM packaging.

Several other projects explored placing processors in DRAM for more massively parallel computation. IRAM [19] solved this problem by placing a single-vector processor in DRAM. For applications amenable to vectorization, this is an excellent match between a high, bandwidth memory and a processing element. Notre Dame's PIM [20] project uses SIMD functional units to consume the extra bandwidth. DIVA [21] has the most sophisticated design, allowing for a kernel to run on the PIM processors. It also features a dedicated PIM communication network, allowing for communication between PIM processors without host processor intervention. Currently, there is a single computational element in each DRAM.

The Impulse project [22] has similar goals to Active Pages but focuses on adding address manipulation functions to the memory controller. Its applications, such as gather-scatter for multiplying a sparse matrix by a dense vector, are also enhanced by more efficiently feeding the microprocessor with data. All the Active Pages applications, however, require some small computations that cannot be supported without more generalized computation in the memory system than Impulse provides.

35.7 SUMMARY

This chapter presented the enormous potential for memory-centric computation, along with several issues specific to the Active Pages DRAM environment. The potential for all memory-centric designs is the bandwidth between memory and the nearest computational unit. The challenge, just as in Active Pages, is how to communicate between units. As the ratio of memory to processing units decreases, the total bandwidth increases, but the communication needs increase. This different balance between computation and communication can affect the asymptotic properties of algorithms.

The barriers for intelligent memory, in particular, are the need for explicit parallel programming and the buy-in by manufacturers to put it in commodity production to lower the price. DIVA is working on a migration path for this technology. The advent of multicore commodity processors pushes the field in two directions. First, it provides performance improvements in multi-programmed environments without the need for parallel programming. This hurts the case for intelligent memory. The prevalence of parallel processors on the market, however, increases the utility of parallel programming so that this may not be such a rare skill in the future. If parallel programming becomes commonplace, then intelligent memory will be poised for success in the commodity market.

Acknowledgments Like any large-scale project, Active Pages was the work of several people over several years. Fred Chong and Mark Oskin were the driving force behind the project. Matt Farrens provided valuable advice. Several graduate and undergraduate students contributed to the project, including Justin Hensley, Lucian Vlad-Lita, Tim Sherwood, Ravishankar Rao, Aneet Chopra, Paul Sultana, and Jennifer Hollfelder.

References

[1] K. Itoh et al. Limitations and challenges of multigigabit DRAM chip design. *IEEE Journal of Solid-State Circuits* 32(5), 1997.

[2] M. Oskin, J. Hensley, D. Keen, F. T. Chong, M. K. Farrens, A. Chopra. Exploiting ILP in page-based intelligent memory. *International Symposium on Microarchitecture*, 1999.

[3] Semiconductor Industry Association. The national technology roadmap for semiconductors. *http://www.sematech.org/public/roadmap/*, 1994.

[4] M. Oskin, F. T. Chong, T. Sherwood. Active pages: A computation model for intelligent memory. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[5] P. Ashenden. *The Designer's Guide to VHDL*, 2nd ed., Morgan Kaufmann, 2002.

[6] Altera Corporation. *FLEX 10K Embedded Programmable Logic Family*, May 1998.

[7] M. Oskin, L. V. Lita, F. T. Chong, J. Hensley, D. K. Franklin. Algorithmic complexity with page-based intelligent memory. *Parallel Processing Letters* 10(1), 2000.

[8] A. Kautonen, V. Leppnen, M. Penttonen. PRAM model. *http//www.cs.joensuu.fi/pages/penttonen/parallel/pram.pram.html.*

[9] M. Oskin, L.-V. Lita, F. T. Chong, J. Hensley, D. K. Franklin. Algorithmic Complexity with Page-Based Intelligent Memory. Technical Report CS-01-00, Department of Computer Science, University of California, Davis, February 2000.

[10] D. T. Hoang. Searching genetic database on Splash 2. In D. Buell, J. Arnold, W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996.

[11] M. Oskin, F. T. Chong, T. Sherwood. ActiveOS: Virtualizing intelligent memory. *Proceedings of the IEEE International Conference on Computer Design*, 1999.

[12] A. Asthana, M. Cravatts, P. Krzyzanowski. Design of an active memory system for network applications. *International Workshop on Memory Technology, Design and Testing*, IEEE Computer Society Press, 1994.

[13] M. Noakes, D. Wallach, W. Dally. The J-Machine multicomputer: An architectural evaluation. *Proceedings of the 20th Annual ACM International Symposium on Computer Architecture*, May 1993.

[14] W. Lee. Space-time scheduling of instruction-level parallelism on a Raw machine. *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[15] A. A. Chien, R. K. Gupta. MORPH: A system architecture for robust high performance using customization. *Frontiers*, 1996.

[16] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. *Symposium on FPGAs for Custom Computing Machines*, April 1997.

[17] Z. Miled, R. Eigenmann, J. Fortes, V. Taylor. Hierarchical processors-and-memory architecture for high performance computing. *Sixth Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[18] Y. Kang, M. Huang, S. Yoon, Z. Ge, D. K. Franklin, V. Lam, P. Pattnaik, J. Torrellas. FlexRAM: An advanced intelligent memory system. *International Conference on Computer Design*, October 1999.

[19] D. Patterson. Microprocessors in 2020. *Scientific American*, September 1995.

[20] P. M. Kogge, T. Sunaga, E. A. E. Retter. Combined DRAM and logic chip for massively parallel applications. *16th IEEE Conference on Advanced Research in VLSI*, 1995.

[21] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, G. Daglikoca. Architecture: The architecture of the DIVA processing-in-memory chip. *International Conference on Supercomputing*, 2002.

[22] J. Carter, et al. Impulse: Building a smarter memory controller. *Proceedings of the International Symposium on High-Performance Computer Architecture*, January 1999.

PART VI

THEORETICAL UNDERPINNINGS AND FUTURE DIRECTIONS

Parts I through V addressed what reconfigurable architectures look like (Part I), how we can develop reconfigurable solutions (Parts I, II, IV, V), and, by example, where reconfigurable solutions can be particularly beneficial (Part V). In this, the final part of the book, we examine why reconfigurable architectures are beneficial and we gain insight into the areas where the benefits of reconfigurable solutions lie. We also observe technology trends and examine why reconfigurable architectures may become increasingly important over time. To support and ground these discussions, the following chapters delve into the technology basis from which we build these architectures, and their alternatives, and discuss physical issues including area, defects, faults, and manufacturing trends.

Chapter 36 constructs a simplified model of the architectural design space in which postfabrication programmable architectures (e.g., processors, FPGAs, VLIWs, SIMD arrays) are built. Using this model, the chapter illustrates the trade-offs inherent in different architectures and the impact these trade-offs have on the architectures' efficiency in implementing various applications. This simple analysis illuminates the appropriate roles for processors and FPGAs, underscores how we can use FPGAs efficiently, and suggests why, as component capacities continue to grow, reconfigurable architectures may be important for carrying out an ever-enlarging set of high-throughput tasks.

Chapters 37 and 38 explore how continued feature size scaling will influence the design of integrated circuits. As device feature sizes approach the atomic scale, our traditional techniques, abstractions, and solutions may no longer be appropriate. Manufacturing at the atomic scale demands higher regularity and produces less controlled structures. At the same time, physical imperfections (e.g., defects, faults, wear) occur at significantly higher rates. Postfabrication configurability appears to be an essential tool for dealing with these atomic-scale effects. This, too, suggests the growing importance of reconfigurable architectures for future technologies.

Chapter 37 addresses defect and fault tolerance. It shows how configurable designs can accommodate defects and suggests in what directions our design and usage paradigms should evolve in order to deal with increasing defect rates. The chapter also examines how transient faults will affect future configurable systems.

Chapter 38 further explores the impact of technologies in which feature sizes are measured in single-digit atomic widths. It reviews emerging atomic-scale technologies and shows how they can be assembled into a complete reconfigurable architecture.

CHAPTER 36

THEORETICAL UNDERPINNINGS

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

Throughout this book there are examples for which reconfigurable designs offer superior performance to processor-based solutions. The reconfigurable implementation is typically orders of magnitude faster than the processor-based system. Even when we normalize the performance advantage to the number of components used in the solution, or to the number of square millimeters of silicon in the same process technology, we often see the reconfigurable solution providing one to two orders of magnitude higher computational capacity per square millimeter. These observations raise questions about reconfigurable computing systems.

- Why do we see this greater computational capacity per unit area?
- How can we predict when reconfigurable systems can deliver significantly higher performance than processor-based implementations?
- What does this tell us about how we should engineer reconfigurable designs?

This computational density advantage is not an accident. It occurs for real, structural reasons resulting from where silicon is allocated in reconfigurable architectures. Field-programmable gate arrays (FPGAs) and reconfigurable architectures organize their instructions differently from processors, making different trade-offs between instruction and computational density. Processors give up raw computational capacity for the ability to support large and irregular computations robustly, while FPGAs give up the ability to switch rapidly among diverse tasks to maximize available compute density and spatial parallelism. This chapter develops a simple model of programmable devices and uses it to illustrate the gross design space, which includes processors and FPGAs, the trade-offs each makes, and the consequences of those trade-offs.

36.1 GENERAL COMPUTATIONAL ARRAY MODEL

Let us start by focusing exclusively on a capabilities viewpoint, ignoring, for the moment, costs. *What would be good to have for a general-purpose programmable computing architecture?*

The most general and flexible programmable architecture we might build would have:

- Computational operators (e.g., programmable gates) that compute an output bit from some number of input bits
- Full, bit-level interconnect among computational operators
- Local data storage for each bit operator
- The ability to issue a unique instruction to each bit-level computational operator on every cycle; this instruction should indicate:
  - Which computational function the operator should perform on each cycle
  - Where the inputs for the operator should come from, including both spatially from any other operator and temporally from local memory
  - Where the output of the operator on this cycle should go into local memory

Figure 36.1 shows a diagram of this architecture. For this simple model, we assume that all the programmable blocks are identical. We call the instruction that controls each programmable block (including interconnect and memory, as just summarized) a *primitive instruction*, or *pinst* for short (see Figure 36.2). With an array of N blocks, the full instruction word issued on every cycle to control the computational array is the composition of N pinsts.

This array provides a computational capacity of N-bit operations (*bitops*) on each cycle. We have great flexibility in using this array since every bitop can have a unique pinst on every cycle. So, if we need to process an irregular collection of operations, such as a 17-bit add, an 8-bit subtract, a 13-bit exclusive-or (XOR), the next state evaluation on a 23-state finite-state machine (FSM), and a 5-bit shift left by 3, we can direct each bitop independently to keep all bitops performing exactly the operations needed for the computation. Further, if the following cycle needs a very different set of operations, such as a 9-bit multiply by the constant 27, a 12-bit AND, the next state evaluation on a 23-state FSM,

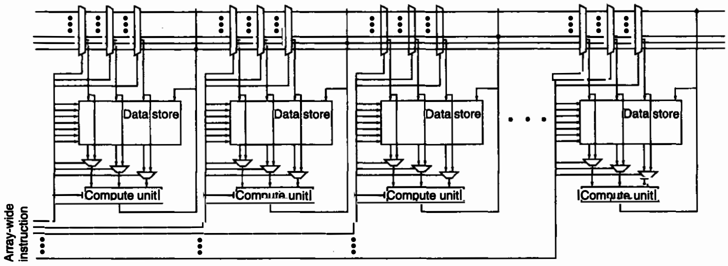


FIGURE 36.1 ■ The general computational array model.

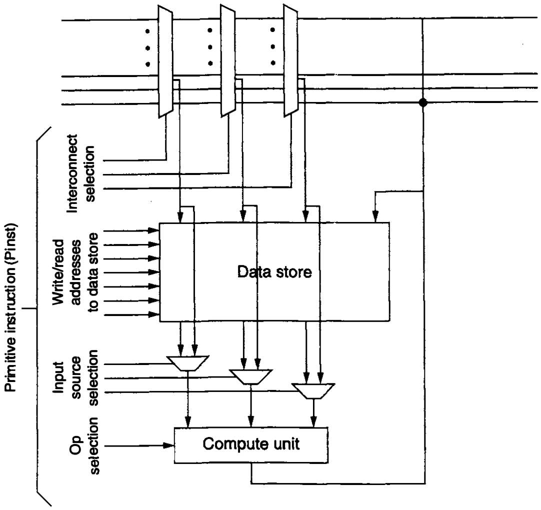


FIGURE 36.2 ■ Primitive instruction (pinst) for programmable bitops.

and an 11-bit shift right by 2, we can issue the next array-wide instruction to control the computational array accordingly.

We get to use all the bitops all the time. Mapping designs to this array is simply a matter of scheduling the bit-level computational needs onto the N-bit operations provided by the array. With this full ability to control the cycle-by-cycle operation of each bitop independently, scheduling is relatively easy. (Strictly speaking, optimal scheduling remains NP-hard, but it can be approximated within a factor of 2 of optimal using a variant of Johnson's Algorithm [1].) So, why is it that we do not have a popular architecture that provides this model?

36.2 IMPLICATIONS OF THE GENERAL MODEL

From a purely logical standpoint, we cannot fault the general computational array model. However, we must implement any architecture in a physical computational medium (e.g., out of a number of discrete vacuum tubes or transistors, on a silicon die, ultimately out of molecules and atoms). To support the architecture, we must commit physical resources. Those resources have a cost in terms of area, delay, and energy. The general computational array model turns out to be extravagant—so much so that we are generally willing to compromise its power to build more practical architectures.

This section illustrates two ways in which the instruction organization of the general model is unreasonably expensive. The focus here is on silicon VLSI implementations, and we discuss the sizes and areas of components in VLSI. To make the discussion general, resource areas are measured in terms of technology-normalized units. In particular, we will measure widths in units of F—the minimum feature size in a VLSI process; as a consequence, areas are measured in units of F^2. VLSI technologies are normally named by their minimum feature size, so when we talk about a 45 nm technology, we are talking about a technology with $F = 45$ nm. Ideally, when we scale from a larger technology to a smaller technology, everything scales as F. Features 900 nm wide in a 90 nm technology are $10F$ wide and should become 450 nm wide in a 45 nm technology. Features do not always scale perfectly linearly like this, but they scale close enough for illustrative purposes. Details and estimates on how the industry expects silicon technology to scale are summarized by the ITRS [2]; the industry collaborates to produce an updated or revised version of this document annually.

36.2.1 Instruction Distribution

This section starts by considering the resource implications of delivering a separate pinst to every bitop. We assume the following:

- The bitops are arranged in a dense $\sqrt{N} \times \sqrt{N}$ array (see Figure 36.3).
- The area required for each bitop, including compute, storage, and interconnect, is $A_{bop} = 250{,}000F^2$; we further assume that the bit operator itself is laid out as a square $500F$ on a side. This size assumes that the interconnect has also been designed in a more restrictive way than the most general model (see Section 36.1), perhaps resembling something closer to traditional FPGA interconnect capabilities.
- The metal pitch available for distributing an instruction bit is $W_{metal} = 4F$. The minimum pitch possible in a given technology is $2F$ because we need to leave one feature size worth of space between features so that they do not short together. The smallest feature sizes tend to be polysilicon for transistor gate widths, with metal pitches being a little wider. A modern VLSI process has many metal layers, and the ones higher in the stack (farther from the silicon base) tend to be wider.
- We have one complete horizontal metal layer and one complete vertical metal layer available to distribute instructions. As noted, modern VLSI processes generally have many metal layers; for example, an $F = 65$ nm process might have 11 metal layers. Some of the layers will be needed for local wiring in the cell, some for power and clock distribution, and some for interconnect. Dedicating two complete metal layers to instruction distribution is extravagant even with 11 metal layers.
- Each pinst requires $I_{bits} = 64$ to specify its instruction. This may seem small if we think about how many bits are required per 4-LUT in an FPGA, or large if you think about 32-bit processor instructions. Encoded densely, FPGA configurations could be much smaller [3]. The capabilities of the pinst might be closer to two processor instructions than one.

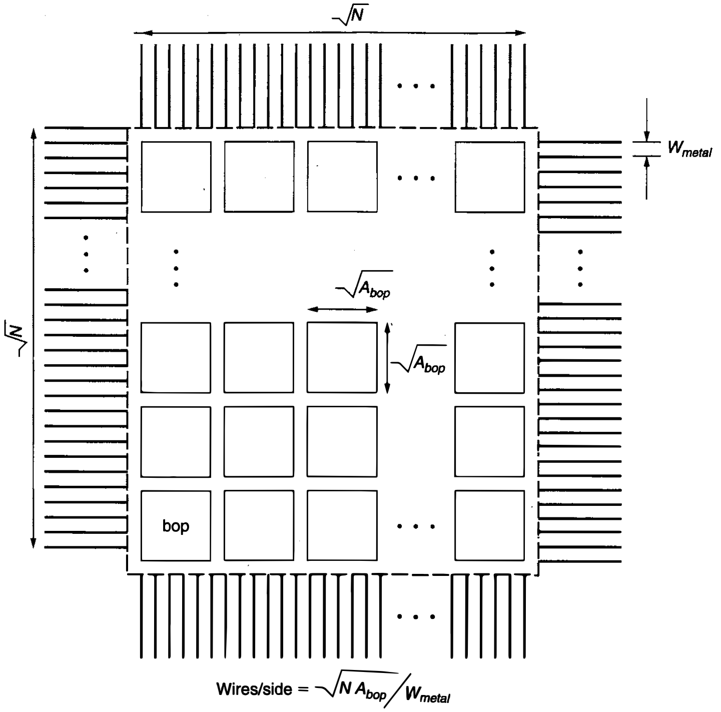


FIGURE 36.3 ■ Wiring for instruction distribution.

As we will see, the preceding assumptions only affect the particular quantitative conclusion we reach. The qualitative effect remains even if we assume two or four times as many metal layers, half the metal pitch, more compact instruction encodings, or larger bitop cell sizes.

If the instructions must all come into the computational array, then the total wiring capacity available for instruction distribution is equal to the perimeter of the array.

$$A_{side}(N) = \sqrt{N} \times \sqrt{A_{bop}} \tag{36.1}$$

$$L_{perimeter}(N) = 4 \times A_{side}(N) \tag{36.2}$$

Note that the two metal layers allow the connections on the top and bottom layers to cross over each other to reach into the array. However, if the lower layer is completely dense, we will have trouble making connections between the upper layer and the bit operations (i.e., we need to reserve space for vias through the lower level). To keep the math simple, general, and illustrative, we will not model that effect, which will only tend to make the problem more severe than the simple model indicates.

To feed the N-bit operators into the array, we need:

$$I_{total\_bits}(N) = N \times I_{bits} \tag{36.3}$$

$$L_{instr\_dist}(N) = W_{metal} \times I_{total\_bits}(N) \tag{36.4}$$

For the distribution to be viable, we need:

$$L_{perimeter}(N) > L_{instr\_dist}(N) \tag{36.5}$$

Substituting into the previous equations, this results in:

$$4 \times \sqrt{N} \times \sqrt{A_{bop}} > W_{metal} \times N \times I_{bits} \tag{36.6}$$

$$\frac{4 \times \sqrt{A_{bop}}}{W_{metal} \times I_{bits}} > \sqrt{N} \tag{36.7}$$

$$N < \left(\frac{4 \times \sqrt{A_{bop}}}{W_{metal} \times I_{bits}}\right)^2 \tag{36.8}$$

Using the preceding assumptions:

$$N < \left(\frac{4 \times 500F}{4F \times 64}\right)^2 = 61 \tag{36.9}$$

This says that we cannot afford to feed more than about 60 bit-processing units without saturating available instruction distribution bandwidth. If we want to support more bit-processing elements, we must increase the perimeter and effectively make the bitops larger. Rearranging equation 36.6 with A_{bop} as the variable:

$$\sqrt{A_{bop}(N)} > \frac{W_{metal} \times \sqrt{N} \times I_{bits}}{4} \tag{36.10}$$

$$A_{bop}(N) = \left(\frac{W_{metal} \times \sqrt{N} \times I_{bits}}{4}\right)^2 \tag{36.11}$$

$$A_{bop}(N) = 4096 \times NF^2 \tag{36.12}$$

That is, the area of each bitop needs to grow linearly with N, meaning that the array area is actually growing quadratically with N.

Equivalently, we can recognize this effect as a difference between the growth rate of the area and the perimeter. If we assume the bitop area is constant, then the total area in the array is growing linearly in the number of bitops. However, the perimeter of the array is only growing as the square root of the array area. So it is not surprising that we reach a point where the array's need for instructions, which is also growing linearly with bitops, exceeds the ability to feed instructions into the array that grows only as the square root of the number of bitops in it. The particular assumptions used for this example starkly illustrate that this effect is already an issue for very small arrays. You can substitute your favorite assumptions about instruction bits, metal pitch, metal layers, or bit-operator area, but the qualitative conclusion remains as follows:

> *If we support this model, either we are limited in the size of the arrays we can build, or instruction distribution wiring ends up dominating all other resources and forces us to scale only as the square root of the area we spend on the computational array.*

36.2.2 Instruction Storage

The previous section illustrated that instruction distribution from outside the computational array is not scalable to large computations. Alternately, consider storing the instructions inside the array. In particular, each bitop could include an instruction memory that holds its instruction (see Figure 36.4). We would

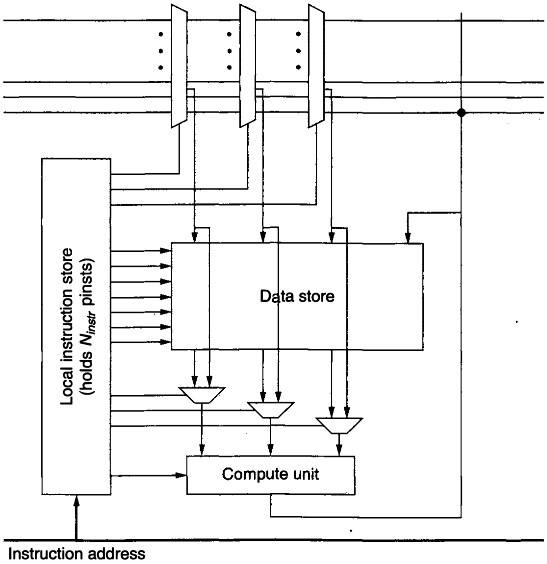


FIGURE 36.4 ■ A bitop with local instruction memory.

then only need to broadcast an address into the array, and each bitop could translate that through the instruction memory to its instruction. Even a 64-bit address is small compared to $L_{perimeter}(1)$, so this solution does not challenge wiring capacity. However, it does raise the question of how large the instruction memory should be to begin to approximate the general model.

In any case, storing the instructions requires area. So we should assess the cost of storing these instructions. Assume that the instruction memory lives in SRAM, and that the area of an SRAM cell to hold an instruction bit is $A_{bit} = 200F^2$. This means that the area per instruction is:

$$A_{pinst} = A_{bit} \times I_{bits} \tag{36.13}$$

$$A_{pinst} = 200F^2 \times 64 = 12{,}800F^2 \tag{36.14}$$

The total area per bitop is now:

$$A_{bitop\_w\_imem} = A_{bop} + N_{instrs} \times A_{pinst} \tag{36.15}$$

$$A_{bitop\_w\_imem} = 250{,}000F^2 + N_{instrs} \times 12{,}800F^2 \tag{36.16}$$

Equation 36.16 now tells a very interesting story. The area required to store a single instruction is small compared to the area required for compute and interconnect in the bit operator (one-twentieth the area). If we store 20 instructions locally, we place half of the area into instruction memory. When we store 200 instructions locally, the instruction memory area ends up dominating (i.e., is 10 times the size of) the area required for computation. That is, given fixed area, the design with 200 instructions will only fit one-tenth the number of bitops as the design with a single local instruction.

Unless we can limit the number of different, array-wide instructions we need to issue, the instruction memory needed to approximate the general model will end up dominating the computational area. Taken together with the result on instruction distribution, these examples illustrate why the general model is not typically supported:

> *To support the general model, instruction resources would dominate all other resources, forcing limited computational density.*

We are left with the choice of either accepting very low computational density or looking for compromises in the general model that will allow us to avoid the huge instruction expense it implies.

36.3 INDUCED ARCHITECTURAL MODELS

If the general model was viable, we would not have the varied set of computer architectures that exist. That is, computer architectures arise because (1) the general model is too expensive, and (2) there is structure in typical computational tasks that permits more economical implementations. Having identified that it is unreasonable to support the general computational array model, we ask: Which structure exists in typical computations that can be exploited to provide a more economical implementation?

36.3.1 Fixed Instructions (FPGA)

If the instructions never change, we do not need to distribute them into the computational array, nor do we need to allocate instruction memory area to store more than a single instruction. We still allow each bitop a pinst, so each can perform a unique operation; however, we do not allow the pinst to change from cycle to cycle. Unchanging instructions is an extreme form of temporal locality, where computation remains the same over time. This allows us to build large arrays and keep the computation dense. If we need to, or can arrange to, perform the same computation on every cycle, then we use the array efficiently. This restriction on the general model effectively gives us an FPGA or spatially reconfigurable architecture. In Chapter 5, Section 5.2, we saw many system architectures that illustrate how we might organize computation to enhance this kind of structure.

36.3.2 Shared Instructions (SIMD Processors)

Another structure common to applications is SIMD datapaths (see Single program, multiple data subsection of Section 5.2.4)—that is, it is common for us to identify sequences of bit-level operations that are the same across a number of data bits. The most common case is word-wide operations, such as multibit adds or bitwise logical operations (e.g., OR, AND, XOR). At a higher level, we would perform a number of identical word-wide operations on different data (e.g., performing a component-wise multiplication on the elements of two arrays as part of a dot product). Here we perform the same operation across many bitops. Rather than providing a unique instruction for each bitop, we can arrange to share a single instruction across a large number of bit operators, amortizing the instruction distribution or storage expense.

In the extreme, we would distribute a single instruction to all the bitops in the array. This is the opposite of the simplification used in the FPGA. Here, all bitops in the array must perform the same operation on a given cycle, but this operation may change from cycle to cycle.

We can view conventional, word-wide processors as exploiting this idea. A processor instruction typically only tells the datapath to do one homogeneous thing—that is, the processor instruction asks every bit in the arithmetic logic unit (ALU) bit slice to perform the same computation (e.g., perform a full adder bit, perform an XOR, perform a shift). For example, a 32-bit processor datapath could perform many more operations if each individual bit slice of the ALU could operate independently; instead, ALUs are constrained to operate in SIMD fashion to keep the cycle-by-cycle instruction size small.

In the general computational array model, we saw that the instruction memory took up the same area as the computation when we stored only 20 instructions in the array (equation 36.16). If we instead share each instruction across

$W_{simd} = 32$ bitops to form a SIMD datapath, it takes 625 instructions for the instruction memory to reach parity with the computation—that is:

$$A_{bitop\_w\_imem}(W_{simd}, N_{instrs}) = A_{bop} + \left(\frac{N_{instrs}}{W_{simd}}\right) \times A_{pinst} \qquad (36.17)$$

$$A_{bitop\_w\_imem}\,(N_{instrs}, 32) = 250,000F^2 + N_{instrs} \times 400F^2 \qquad (36.18)$$

From these illustrations, we can see how the more familiar FPGA and processor architectures fall out as simplifications of the general computational array model that exploits different kinds of structures that exist in typical computations.

36.4 MODELING ARCHITECTURAL SPACE

The demonstrations in Sections 36.2 and 36.3 highlight the fact that choices about instruction architecture can have a first-order impact on the area, and hence density, of programmable computing components. We can take this a step farther and build models of the density, and ultimately relative efficiency, of architectural design points.

Table 36.1 summarizes where some familiar architectures fall in the (W_{simd}, N_{isntr}) architectural space. Nonetheless, remember that we are using a deliberately simple model and that many other effects and issues are associated with each architecture, some of which are mentioned in Section 36.4.3.

36.4.1 Raw Density from Architecture

Using equation 36.17, we can plot the relative densities of each bit operator as a function of the local instruction memory, N_{instr}, and the SIMD instruction width, W_{simd}. Figure 36.5 shows plots of the computational density for the instruction memory from 1 to 16,384 and the instruction width from 1 to 1024. Here, note that peak densities vary over three orders of magnitude. As we increase instruction depth (N_{instr}), we shift area into instructions rather than compute, often significantly reducing computational density. Wide-word architectures can reduce the memory costs at a particular instruction depth, but there also may be significant computational density reductions as instruction depth grows.

TABLE 36.1 ■ Placement of sample architectures in (W_{simd}, N_{instr}) space

Architecture	W_{simd}	N_{instr}	Reference
FPGA	1	1	
GARP fabric	2	4	Chapter 2, Section 2.1.1
KiloCore256	8	16	Chapter 2, Section 2.1.2
MIPS-X	32	512	[4]
IA-64 (Montecito)	64	200,000	[5]
Cell SPU	128	65,536	[6]

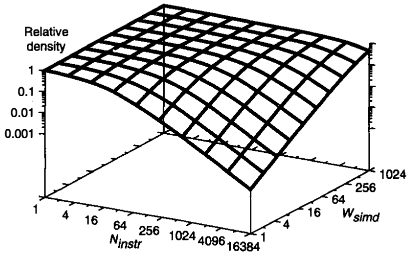


FIGURE 36.5 ■ Relative peak computational density from the model (normalized to the density of $N_{instr} = 1$, $W_{simd} = 1024$ design points).

36.4.2 Efficiency

The previous section showed peak raw densities achievable at various architectural points. If peak raw density was all that mattered, we would build SIMD designs with shallow instruction memories, as Figure 36.5 illustrates. However, it is seldom the case today that we can keep the millions of SIMD bit-processing elements we might be able to put on a die performing useful computations. When we cannot match the structure assumed by the architecture, the yield is only a fraction of the potential density—that is, another architecture, perhaps one with lower peak density, often can deliver more net density to the application. In particular, the architectural point whose structure assumptions exactly match the application will deliver the highest net density on that application. This leads to an interesting set of questions:

- How does the efficiency of an architecture fall off as it becomes mismatched to the structure of the application?
- How does the net density compare between various matched and mismatched architectures?

Since there is a model for the area of architectural design points in the (W_{simd}, N_{instr}) design space (equation 36.17), we can use that to measure efficiency. In particular, it is possible to measure the efficiency of an architecture design point (Arch(W_{simd}, N_{instr})) processing applications with a particular structure (App(W_{app}, L_{path})) as the ratio of the area of the architecture that exactly matches the application structure to the area of the point being evaluated:

$$\text{Efficiency}\left(\text{Arch}(W_{simd}, N_{instr}), \text{App}\left(W_{app}, L_{path}\right)\right) = \frac{\mathit{Area}\left(\text{Arch}\left(W_{app}, L_{path}\right), \text{App}\left(W_{app}, L_{path}\right)\right)}{\mathit{Area}\left(\text{Arch}(W_{simd}, N_{instr}), \text{App}\left(W_{app}, L_{path}\right)\right)} \quad (36.19)$$

TABLE 36.2 ■ Sample applications in the (W_{app}, L_{path}) space

Application	W_{app}	$L_{critpath}$	L_{path}	Comments
Conway's Game of "Life"	1	1	1	Bit-level CA [7]
Error correcting codes	1	1	1–10,000	At memory interface, need one per cycle; on audio-rate, real-time data can be low throughput
Entropy coding	1	1–10	1–10,000	(similar to previous)
Video processing of pixel data	8	1–6	12	1024×1024 at 30 frames per second on a 500 MHz cycle can afford approximately 12 cycles per pixel
CD audio	16	1–10	10,000	44 kHz real-time vs. 500 MHz cycle
SPIHT image compression	16	10	10+	Chapter 27
FDTD	35	1–5	1–5	Chapter 32

To characterize the structure of the architecture separately from the structure of the application, equation 36.19 keeps W_{simd} and N_{instr} as parameters characterizing the architecture and adds the dual parameters W_{app} and L_{path} to characterize the application structure. W_{app} is simply the natural SIMD datapath width of the application, while L_{path} is the path length of the application (see the Mismatch in N_{instr} subsection).

For illustrative purposes, Table 36.2 summarizes where several applications appear in the (W_{app}, L_{path}) space. The area of the mismatched design is always larger, so the efficiency metric in equation 36.19 effectively tells us how much lower the mismatched point's net density is than the matched point's net density.

To develop the intuition and keep the explanation simple, we stay with the assumption that applications have homogeneous structure (i.e., single-characteristic W_{app} and L_{path}). One of the reasons we are interested in how well an architecture deals with different, mismatched structures is that a real application will typically contain heterogeneity in the structure it exhibits.

Mismatch in W_{simd}

What happens when the application width W_{app} is mismatched to the architectural width W_{simd}?

- $W_{simd} > W_{app}$: Here we do not have as fine-grained control of the bit operators as the application requires. Consequently, bitops go unused. In particular, we will actually need a larger array so that we match the instruction control needs of the application. For example, if $W_{app} = 5$ and $W_{simd} = 8$, then three bitops in every architectural SIMD datapath will go idle. To satisfy the application requirements, we end up needing $\frac{W_{simd}}{W_{app}} = \frac{8}{5} = 1.6$ times as many physical bitops as the application actually requires.

- $W_{simd} < W_{app}$: There are two effects that can work to make implementations in this architecture larger than the optimally matched architecture:
  1. We have finer-grained control, but may still need more physical bit operators because of granularity problems. For example, when $W_{app} = 8$ and $W_{simd} = 5$, we need $\left\lceil \frac{W_{app}}{W_{simd}} \right\rceil = 2$ groups of W_{simd} bitops to cover each application group, or $\left\lceil \frac{8}{5} \right\rceil \times W_{arch} = 10$ bitops, of which only $W_{app} = 8$ are doing useful work.
  2. Since we have more control than necessary for the application, the area of each bitop is larger than necessary in order to accommodate additional instruction memory; this extra instruction memory holds redundant information. Continuing the $W_{app} = 8$ and $W_{simd} = 5$ example, each bit operator effectively pays for $\frac{W_{app}}{W_{simd}} = \frac{8}{5} = 1.6$ times as many instructions as necessary for the application.

Assuming that instruction storage depth is matched to application path length ($N_{instr} = L_{path}$) to focus on the width mismatch, we can show this in an area model as:

$$
\begin{aligned}
&\mathit{Area}\left(\mathrm{Arch}\left(W_{simd}, L_{path}\right), \mathrm{App}\left(W_{app}, L_{path}\right)\right) \\
&= \left(\frac{W_{simd}}{W_{app}}\right) \times \left\lceil \frac{W_{app}}{W_{simd}} \right\rceil \times A_{bitop\_w\_imem}\left(W_{simd}, L_{path}\right) \qquad (36.20) \\
&= \left(\frac{W_{simd}}{W_{app}}\right) \times \left\lceil \frac{W_{app}}{W_{simd}} \right\rceil \times \left(A_{bop} + \left(\frac{L_{path}}{W_{simd}}\right) \times A_{pinst}\right)
\end{aligned}
$$

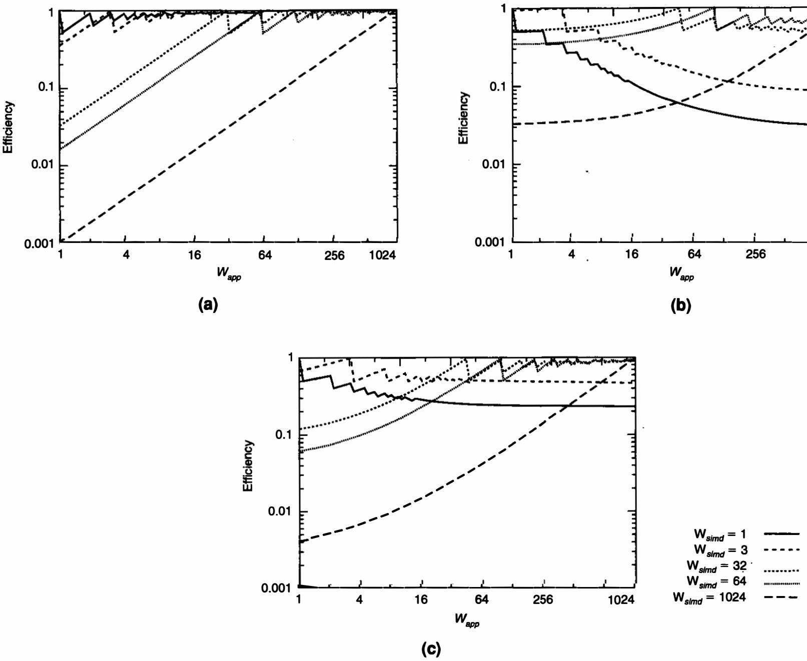
This allows us to compute the efficiency of the mismatched SIMD datapath width at a matched L_{path} as:

$$
\begin{aligned}
&\mathrm{Efficiency}\ \left[L_{path}\right]\left(W_{simd}, W_{app}\right) \\
&= \frac{\left(A_{bop} + \left(\frac{L_{path}}{W_{app}}\right) \times A_{pinst}\right)}{\left(\frac{W_{simd}}{W_{app}}\right) \times \left\lceil \frac{W_{app}}{W_{simd}} \right\rceil \times \left(A_{bop} + \left(\frac{L_{path}}{W_{simd}}\right) \times A_{pinst}\right)} \qquad (36.21)
\end{aligned}
$$

Figure 36.6 shows plots of the efficiency from equation 36.21 versus W_{app} for a collection of W_{simd}'s and L_{path}'s. Perhaps more significant than the large density range shown in Figure 36.5, we see that SIMD width mismatches can cost us orders of magnitude in net density delivered to an application. Interestingly, we see some SIMD width selections that do not show orders of magnitude efficiency losses (e.g., $W_{simd} = 1$ for $L_{path} = 1$, $W_{simd} = 3$ for $L_{path} = 64$, $W_{simd} = 32$ for $L_{path} = 640$). These robust points occur when the instruction area is equal to the compute and interconnect area. That is:

$$A_{bop} = \left(\frac{L_{path}}{W_{simd}} \right) \times A_{pinst} \qquad (36.22)$$

In these cases, half the area is allocated to storing instructions and half to compute. For illustration, consider the $L_{path} = 640$ and $W_{simd} = 32$ case. Here, if we are processing $W_{app} = 1$ data, then we use only one-thirty-second of the compute



(a)

(b)

(c)

FIGURE 36.6 ■ Efficiency as a function of W_{app} for various L_{path} values: (a) $L_{path} = 1$, (b) $L_{path} = 640$, and (c) $L_{path} = 64$.

area. However, we are able to use all the memory area; a matched architecture can, at most, be half the size of this design point since it still requires the 640 instructions, even if they drive a smaller datapath. At the opposite extreme, if $W_{app} = 16,384$, we can use all the compute operators but we underutilize the instructions. Here, a matched architecture could have used a factor of 512 lower instruction area; however, since half the area is in compute, the matched architecture is, at best, only half the size of this robust point.

It should be clear that this observation holds for any choice of W_{app} when the area is allocated evenly between compute and instruction memory. In contrast, if we make $W_{simd} = 1$ for this $L_{path} = 640$ case, then 97 percent of the area goes into memory; if this $W_{simd} = 1$ architecture now has a task with $W_{app} = 16,384$, it is much larger (at least 33 times larger) than a design with matched width, which can put significantly less area into instruction memory.

If we can design to a single application width, or a small range of widths, it is best to select a matched width, or the width that provides the highest average efficiency over the range. However, if we don't have tight bounds on the application width, these robust points show how we can select organizations that remain fairly efficient for any application width.

Mismatch in N_{instr}

A similar phenomenon occurs when N_{instr} does not match the structure of the application. First, we need to understand L_{path}—the application demand for N_{instr}. In particular, let us consider an inner loop in a kernel or the computation required for each invocation of a transform operator (see Transform or object subsection of Section 5.1.2). To compute each inner loop iteration, or each operator invocation, we need to evaluate a set of N_{ops} bitops. In general, there may be a set of cyclic sequential dependencies, or a critical path, of depth $L_{critpath}$ among the bitops in the computation that prevent us from starting the next iteration of the loop or invocation of the operator until the $L_{critpath}$ array cycles have completed. For example, consider the loop body of a saturated accumulation:

$$y[i] = \max(\min(x[i] + y[i-1], 255), 0)$$

Before performing the next addition to compute $y[i+1]$ from $y[i]$, we must complete the computation of $y[i]$, including both the addition and the selection of maximum or minimum bound limits (see Figure 36.7).[1] Assume the following:

- The addition requires a path length of six sequential bitops.
- The comparisons can be performed in parallel.
- Each comparison requires a path of three sequential bitops.
- The final selection requires a single bitop.

The critical path $L_{critpath}$ is 10 for this computation. With a path length of $L_{critpath}$, we can schedule the N_{ops} required to evaluate the application into $L_{critpath}$ cycles

[1] With care, this actually can be avoided using sophisticated transformations [8].

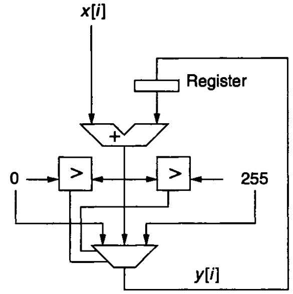


FIGURE 36.7 ■ Saturated accumulator cyclic dependency.

on the array without slowing down the application, the sequentially dependent paths guarantee that it will always take at least $L_{critpath}$ cycles to perform the operation.

The application may not actually demand that the computation be performed every $L_{critpath}$ cycle. Perhaps the data throughput is lower and new samples, $x[i]$, are arriving every 20 ns while the array cycle time is 1 ns. Here, evaluating with $L_{critpath} = 10$ leaves the array sitting idle for 10 cycles before the next input sample is available to compute. Consequently, it would be possible to schedule to $L_{path} = 20 > L_{critpath}$ and cut the number of bitops needed by at least a factor of 2. In this way, the loop or transform body is efficiently implemented by scheduling the computations onto a minimum number of bitops in a period of L_{path} cycles, with each operator potentially getting a unique instruction on each cycle $N_{instr} = L_{path}$. For examples, see Table 36.2, which summarizes the throughput L_{path} required in a few applications.

Now consider the two mismatched cases:

- $N_{instr} > L_{path}$: In this case, by scheduling the computation into L_{path} cycles, $(N_{instr} - L_{path})$ instruction memory slots in each bitop go unused. The matched architecture is smaller because it does not spend area on these unused instruction memories. In the aforementioned saturated accumulation, if $L_{path} = 20$ and an array with $N_{instr} = 100$ is used, then 80 instruction slots go unused.
- $N_{instr} < L_{path}$: In this case, we cannot necessarily reuse each bit operator in L_{path} in different ways on each of the L_{path} cycles. Since we can only use each operator in N_{instr} ways, to solve the entire problem we may need a total of $\left\lceil \frac{L_{path}}{N_{instr}} \right\rceil$ times as many bitops to perform the computation. Continuing with the example, if $N_{instr} = 5$ and there is an $L_{path} = 20$, we may need four times as many bitops as the optimally matched architecture. The total amount of memory is the same between these cases; however, an $N_{instr} = 5$ architecture pays for four times as many

compute blocks (A_{bop}). There is also a granularity effect here; for example, we still need four times as many bitops even when $N_{instr} = 6$.

Assuming that the datapath width is matched ($W_{simd} = W_{app}$), allows us to focus on the instruction mismatch; we can show this in an area model as:

$$
\begin{aligned}
&Area\left(\text{Arch}\left(W_{app}, N_{instr}\right), \text{App}\left(W_{app}, L_{path}\right)\right) \\
&\quad = \left\lceil \frac{L_{path}}{N_{instr}} \right\rceil \times A_{bitop\_w\_imem}\left(W_{app}, N_{instr}\right) \\
&\quad = \left\lceil \frac{L_{path}}{N_{instr}} \right\rceil \times \left(A_{bop} + \left(\frac{N_{instr}}{W_{app}}\right) \times A_{pinst}\right)
\end{aligned}
\qquad (36.23)
$$

This allows us to compute the efficiency of the mismatched instruction store at a matched W_{app} as:

$$
\begin{aligned}
&\text{Efficiency}\left[W_{app}\right]\left(N_{instr}, L_{path}\right) \\
&\quad = \frac{\left(A_{bop} + \left(\frac{L_{path}}{W_{app}}\right) \times A_{pinst}\right)}{\left\lceil \frac{L_{path}}{N_{instr}} \right\rceil \times \left(A_{bop} + \left(\frac{N_{instr}}{W_{app}}\right) \times A_{pinst}\right)}
\end{aligned}
\qquad (36.24)
$$

Figure 36.8 plots the efficiency from equation 36.24 versus L_{path} for a collection of N_{instrs}'s and W_{apps}'s. Again, note that instruction store mismatches can cost orders of magnitude in net density. We also see robust points here where the net density remains within 50 percent of the matched architecture. The effect is the same as for datapath width mismatch (see previous section), and the efficient points are governed by an analogous equation:

$$
A_{bop} = \left(\frac{N_{instr}}{W_{app}}\right) \times A_{pinst} \qquad (36.25)
$$

For any of these robust points, at the minimum value, $L_{path} = 1$, we are using all the compute area and only a fraction of the instruction memory area, so an optimally matched architecture could, at best, be implemented in half the area. Similarly, for arbitrarily large L_{path}, if $N_{instr} < L_{path}$, all the instruction memory area is used to hold instructions, but this may leave the compute area idle most of the time. Here, again, with only 50 percent of the area in compute, the design is, at most, twice the size of an optimally matched architecture with less area allocated to computation. In contrast, if we put 90 percent of the area into compute, then we could end up wasting 90 percent of the area in scenarios where $L_{path} >> N_{instr}$; matched architectures can be an order of magnitude smaller in such cases. Similarly, if 90 percent of the area is put into instruction memory, we can end up wasting almost 90 percent of the area when L_{path} is small.

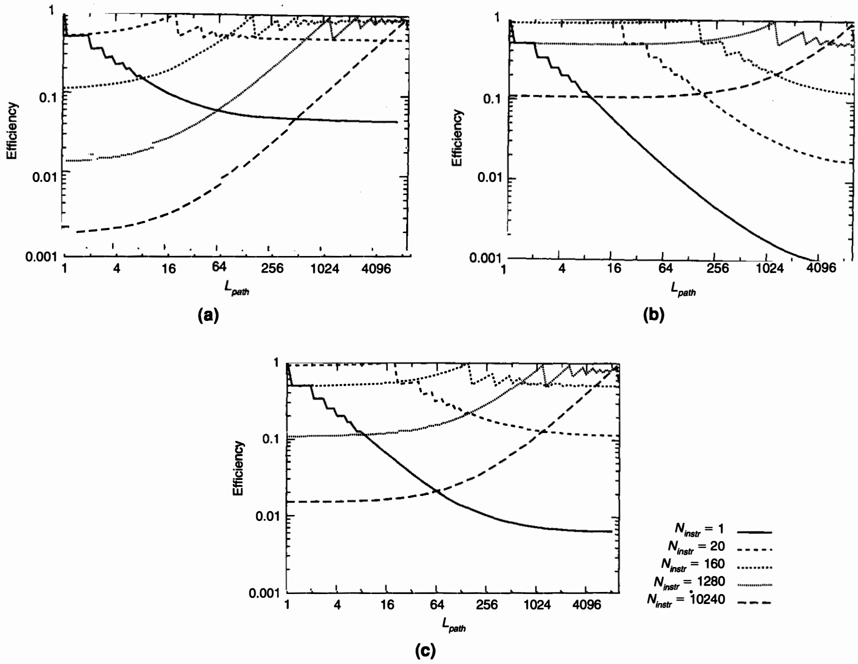


FIGURE 36.8 ■ Efficiency as a function of L_{path} for various W_{app} values: (a) W_{app} = 1, (b) W_{app} = 64, and (c) W_{app} = 8.

Composite effects

Combining the effects of SIMD width mismatch and local instruction storage mismatch, we get the total efficiency:

$$\text{Efficiency}\ \left(\text{Arch}(W_{simd}, N_{instr}), \text{App}\left(W_{app}, L_{path}\right)\right) = \frac{\left(A_{bop} + \left(\frac{L_{path}}{W_{app}}\right) \times A_{pinst}\right)}{\left(\frac{W_{simd}}{W_{app}}\right) \times \left\lceil \frac{W_{app}}{W_{simd}} \right\rceil \times \left\lceil \frac{L_{path}}{N_{instr}} \right\rceil \times \left(A_{bop} + \left(\frac{N_{instr}}{W_{simd}}\right) \times A_{pinst}\right)} \qquad (36.26)$$

Unfortunately, if both the SIMD width and the local instruction storage mismatch, it is not possible to pick a robust point as we did in previous sections.

Returning to equations 36.22 and 36.25, we note that the robust points occur when we can match the instruction storage area, $\left(\frac{N_{instr}}{W_{simd}}\right) \times A_{pinst}$, and the computation and interconnect area, A_{bop}. However, when both W_{app} and L_{path} vary, even when the area is matched, we can have cases where the allocation of width versus storage size within that area can prevent us from using the computational units efficiently.

Efficiency of processors and FPGAs

The previous section suggests that we will not find an architectural point in this (W_{simd}, N_{instr}) design space that is efficient across a wide range of application structures. To understand where processors and FPGAs are efficient, we can use the composite efficiency relation (equation 36.26) and estimate how efficient they each can be across a portion of the design space (see Figure 36.9). Here the FPGA is naturally modeled with $N_{instr} = 1$ and $W_{simd} = 1$. We model a processor as $W_{simd} = 64$ and $N_{instr} = 16{,}384$.

Figure 36.9 shows starkly that the FPGA and processor are both designed for different points in the application space. Notice that each can be less than 1 percent efficient in some portions of the space. Further, we note that *in the places where the processor is very inefficient (< 1 percent), the FPGA is highly efficient; the reverse is true as well.* This effect, coupled with the heterogeneous nature of applications, explains why it is often useful to have reconfigurable systems that mix FPGA or reconfigurable fabrics along with processors (e.g., Instruction augmentation subsection of Section 5.2.2 and Chapter 26).

36.4.3 Caveats

As noted in the introduction to this chapter, we are deliberately using a simple model to illustrate key effects in instruction organization. There are many other application structural opportunities and architectural variables that can also have a large effect on resource balance and efficiency, including interconnect richness (e.g., [9]) and organization, data storage and memory hierarchy capacities, bandwidth and latencies, threads of control, dynamic instruction selection, and integration of hardware functional units (e.g., multipliers [10,11]

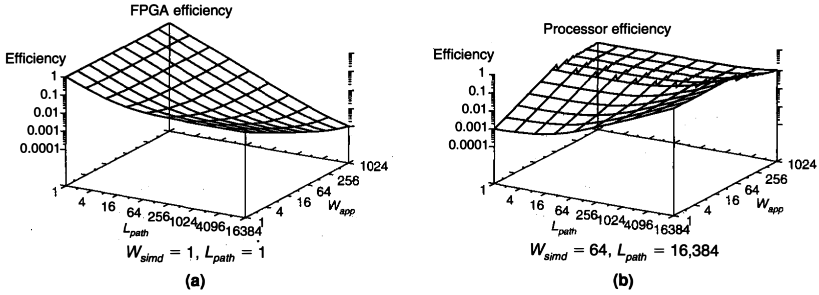


FIGURE 36.9 ■ Efficiency of FPGA-like (a) and processor-like (b) designs across both L_{path} and W_{app}.

and floating-point units [12]). In processors, the SIMD control of ALUs is coupled with fast logic to support carries in arithmetic (e.g., [13]), which serves to reduce $L_{critpath}$; FPGAs also employ fast cascade structures for similar reasons (e.g., [14], Chapter 1) but do not tie them to SIMD datapaths. Nonetheless, the simple model shows that these instruction organization decisions can have a significant impact on computational density, and it illustrates why FPGAs can be more efficient than processors for important classes of applications.

36.5 IMPLICATIONS

36.5.1 Density of Computation versus Description

From this model, we can clearly see a trade-off between computational density and instruction density. Equation 36.16 illustrates that the instruction store area for a single bitop can be an order of magnitude smaller than the computation to support it. This means an $N_{instr} = 1$ design stores instructions an order of magnitude less densely than an $N_{instr} = 200$ design, and an $N_{instr} = 200$ design packs computation an order of magnitude less densely than an $N_{instr} = 1$ design.

When the goal is to simply pack a large, irregular computation into a small area, we are best off focusing on instruction density; this minimizes the area for the implementation, at the expense of lower performance. When the goal is to perform the computation at high throughput, designs with high computational density allow us to meet the throughput with the least area.

36.5.2 Historical Appropriateness

When we first started building programmable integrated circuits, the premium for describing large computations was high. The capacity on a single integrated circuit was very low when they were built with $F = 3\mu m$ technology. In the mid-1980s, with $N_{instr} = 1$ and $W_{simd} = 1$, we could put only 64 bitops on a die [15], limiting computations to those that could be described by 64 instructions. At roughly the same time, one could put $N_{instr} = 512$ instructions on the die along with 32 bitops controlled in an SIMD fashion by a single pinst on each cycle ($W_{simd} = 32$) [4]. The struggle at this point in history was to fit an entire computational kernel onto a single die, and the deep instruction, word-wide processor design could begin to fit interesting kernels while the FPGA designs could fit only the most trivial computations.

By 2005, however, with $F \leq 0.1\mu m$, the landscape had changed. Moore's Law process scaling has given us more than a 10,000-fold increase in capacity per integrated circuit. Modern processors, still built with ever-deeper memories, have large enough instruction stores to contain large applications. At the same time, FPGAs hold hundreds of thousands of active bitops. Even kernels with thousands of 64-bit-wide operations can fit spatially on the FPGA and exploit the higher computational density.

The question with today's silicon is less "Can we get the application to fit on the die?" and more "How do we turn the available die area into performance?" Consequently, as we continue to scale feature sizes, the fraction of tasks where high instruction density remains the premium is shrinking, while the fraction where the application fits on the die and high computational density offers a benefit is increasing.

36.5.3 Reconfigurable Applications

Understanding why FPGAs can be efficient and where they are most efficient (e.g., Figure 36.9) provides additional insight into where we should use FPGAs and how to fully exploit their strengths. Certainly, if the task has low throughput requirements (i.e., large L_{path}), then FPGAs are often not an efficient implementation. The FPGA is efficient when we operate at minimum path length, preferably $L_{path} = 1$, where we are performing the same operation over and over and keeping all the bitops active during the operation. For FPGAs with a variable clock cycle, we want to keep the cycle time to the minimum, maximizing the reuse rate of each operation. This underscores why retiming operations such as pipelining and *C*-slow (see Chapter 18) are important for optimizing FPGA efficiency, as well as behavioral transformations that reduce $L_{critpath}$.

When L_{path} is large simply because of a low throughput demand, we can often turn the SIMD structure, W_{app}, into additional operation regularity. In particular, when $W_{app} > 1$, that is an indication that a number of bit-level operators do perform the same operation. By moving this regularity into time rather than space, we can reduce the number of unique instruction combinations needed and hence reduce the N_{instr} required. For example, if $W_{app} = 16$ and $L_{path} >> L_{critpath}$, we can implement the SIMD datapath bit serially so that the necessary instruction storage depth is a factor of 16 smaller ($N'_{instr} \approx \frac{L_{path}}{W_{app}}$). As shown in Figure 36.10, this can increase the FPGA's domain of efficiency.

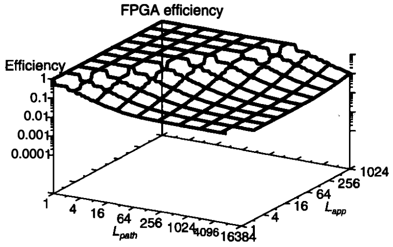


FIGURE 36.10 ■ FPGA efficiency when datapath regularity can be used to increase temporal regularity.

References

[1] D. S. Hochbaum, ed. *Approximation Algorithms for NP-Hard Problems,* PWS Publishing, 1997.

[2] International technology roadmap for semiconductors. *http://www.itrs.net/Links/2005ITRS/Home2005.htm*, 2005.

[3] A. DeHon. Entropy, counting, and programmable interconnect. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, ACM/SIGDA, 1996.

[4] M. Horowitz, J. Hennessy, P. Chow, G. Gulak, J. Acken, A. Agarwal, C.-Y. Chu, S. McFarling, S. Przybylski, S. Richardson, A. Salz, R. Simoni, D. Stark, P. Steenkiste, S. Tjiang, M. Wing. A 32b microprocessor with on-chip 2 Kbyte instruction cache. *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, IEEE, 1987.

[5] C. McNairy, R. Bhatia. Montecito: A dual-core, dual-thread Titanium processor. *IEEE Micro* 25(2), 2005.

[6] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki. Synergistic processing in cells multicore architecture. *IEEE Micro* 26(2), 2006.

[7] M. Gardner. The fantastic combinations of John Conway's new solitaire game "Life." *Scientific American* 223, 1970.

[8] K. Papadantonakis, N. Kapre, S. Chan, A. DeHon. Pipelining saturated accumulation. *Proceedings of the International Conference on Field-Programmable Technology*, 2005.

[9] A. DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1999.

[10] A. DeHon. The density advantage of configurable computing. *IEEE Computer* 33(4), 2000.

[11] I. Kuon, J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26(2), 2007.

[12] M. J. Beauchamp, S. Hauck, K. D. Underwood, K. S. Hemmert. Embedded floating-point units in FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2006.

[13] R. P. Brent, H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers* 31(3), 1982.

[14] S. Hauck, M. M. Hosler, T. W. Fry. High-performance carry chains for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8(2), 2000.

[15] W. S. Carter, K. Duong, R. H. Freeman, H.-C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, S. L. Sze. A user programmable reconfigurable logic array. *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1986.

CHAPTER 37

DEFECT AND FAULT TOLERANCE

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

As device size F continues to shrink, it approaches the scale of individual atoms and molecules. In 2007, 65-nm integrated circuits are in volume production for processors and field-programmable gate arrays (FPGAs). With atom spacing in a silicon lattice around 0.5 nm, $F = 65$-nm drawn features are a little more than 100 atoms wide. Key features, such as gate lengths, are effectively half or a third this size. Continued geometric scaling (e.g., reducing the feature size by a factor of 2 every six years) will take us to the realm where feature sizes are measured in single-digit atoms sometime in the next couple of decades.

Very small feature sizes will have several effects on integrated circuits, including:

- *Increased defect rates:* Smaller devices and wires made of fewer atoms and bonds are less likely to be "good enough" to function properly.
- *Increased device variation:* When dimensions are a few atoms wide, the addition, absence, or exact position of each atom has a significant affect on device parameters.
- *Increased change in device parameters during operational lifetime:* With only a few atoms making up the width of wires or devices, small changes have large impacts on performance, and the likelihood of a complete failure grows. The fragility of small devices reduces traditional opportunities to overstress them as a means of forcing weak devices to fail before the component is integrated into an end system. This means many weak devices will only turn into defects during operation.
- *Increased single die capacity:* Smaller devices allow integration of more devices per die. Thus, not only do we have devices that are more likely to fail, but there also are more of them, meaning more chances that some device on the die will fail.
- *Increased susceptibility to transient upsets:* Smaller nodes use less charge to hold state or configuration data, making them more susceptible to upset by noise, including ionizing particles, thermal noise, and shot noise. Coupled with the greater capacity, which means more nodes that can be upset, dies will have significantly increased upset rates.

Accommodating and exploiting these effects will demand an increasing role for postfabrication configurable architectures. Nonetheless, some usage paradigms will need to shift to fully exploit the potential benefits of reconfigurable architectures at the atomic scale.

This chapter reviews defect tolerance approaches and points out how the configurability available in reconfigurable architectures is a key tool for coping with defects. It also touches briefly on lifetime and transient faults and their impact on configurable designs.

37.1 DEFECTS AND FAULTS

A *defect* is a persistent error in a component. Because defects are persistent, we can test for defect occurrences and record their locations. We contrast defects with transient faults that may produce the wrong value on one or a few cycles but do not continue to corrupt calculations. For the sake of simple discussion here, we classify any persistent problem that causes the circuitry to work incorrectly for some inputs and environments as defects. Defects are often modeled as stuck-at-1, stuck-at-0, or shorted nodes. They can also be nodes that are excessively slow, such that they compute correctly but not in a timely fashion, or excessively leaky, such that they do not hold their value properly. A large number of physical effects and causes may lead to these manifestations, including broken wires, shorts or bridging between nodes that should be distinct, excessive or inadequate doping in a device, poor contacts between materials or features, or excessive variation in device size.

A *transient fault* is a temporary error in a circuit result. Transient faults can occur at random times. A transient fault may cause a gate output or node to take on the incorrect value on some cycle of operation. Examples of transient faults include ionizing particles (e.g., α-particles), thermal noise, and shot noise.

37.2 DEFECT TOLERANCE

37.2.1 Basic Idea

An FPGA or reconfigurable array is a set of identical (programmable) bit-processing operators with postfabrication configurable interconnect. When a device failure renders a bitop or an interconnect segment unusable, we can configure the computation to avoid the failing bitop or segment (see Figure 37.1). If the bitop is part of a larger SIMD word (Chapter 36, Section 36.3.2) or other structure that does not allow its independent use, we may be forced to avoid the entire structure. In any case, as long as all the resources on the reconfigurable array are not being used, we can substitute good resources for the bad ones. As defect rates increase, this suggests a need to strategically reserve spare resources on the die so that we can guarantee there are enough good resources to compensate for the unusable elements.

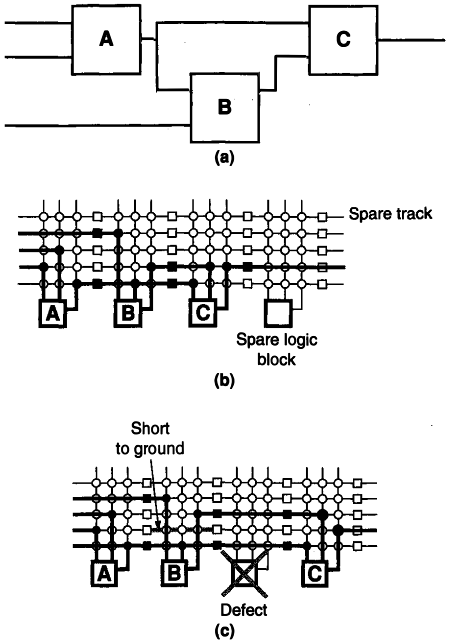


FIGURE 37.1 ■ Configuring computation to avoid defective elements in a reconfigurable array: (a) logical computation graph, (b) mapping to a defect-free array with spare, and (c) mapping to an array with defects.

This basic strategy of (1) provisioning spare resources, (2) identifying and avoiding bad resources, and (3) substituting spare resources for bad resources is well developed for data storage. DRAM and SRAM dies include spare rows and columns and substitute the spare rows and/or columns for defective rows and columns (e.g., see [1,2]). Magnetic data storage (e.g., hard disk) routinely has bad sectors; the operating system (OS) maps the bad sectors and takes care not to allocate data to those sectors. These two forms of storage actually illustrate two models for dealing with defects:

1. *Perfect component:* In the perfect component model, the component has to look perfect; that is, we require every address visible to the user to perform correctly. The spare resources are added beyond those required to deliver the promised memory capacity and are substituted out behind the scenes so that users never see that there are defective elements in the component. DRAM and SRAM components are the traditional example of the perfect component model.

2. *Defect map:* The defect map model allows elements to be bad. We expose these defects to higher levels of software, typically the OS, which is responsible for tracking where the defects occur and avoiding them. Magnetic disks are a familiar example of the defect map model—we permit sectors to be bad and format the disk to avoid them.

37.2.2 Substitutable Resources

Some defects will be catastrophic for the entire component. While a reconfigurable array is composed largely of repeated copies of identical instances, the device infrastructure is typically unique; defects in this infrastructure may not be repairable by substitution. Common infrastructures include power and ground distribution, clocking, and configuration loading or instruction distribution. It is useful to separate the resources in the component into *nonrepairable* and *repairable* resources. Then we can quantify the fraction of resources that are nonrepairable.

We can minimize the impact of nonrepairable resources either by reducing the fraction of things that cannot be repaired or by increasing the reliability of the constituent devices in the nonrepairable structures. Many of the infrastructure items, such as power and ground networks, are built with larger devices, wires, and feature sizes. As such, they are less susceptible to the failures that impact small features. Memory components (e.g., DRAMs) also have distinct repairable and nonrepairable components; they typically use coarser feature sizes for the nonrepairable infrastructure. Memory designs only use the smallest features for the dense memory array, where row and column sparing can be used to repair defects. In FPGAs, it may be reasonable to provide spares for some of the traditional infrastructure items to reduce the size of the nonrepairable region. For example, modern FPGAs already include multiple clock generators and configurable clock trees; as such, it becomes feasible to repair defective clock generators or portions of the clock tree by substitution. We simply need to guarantee that there are sufficient alternative resources to use instead of the defective elements.

For any design there will be a minimum *substitutable unit* that defines the granularity of substitution. For example, in a memory array we cannot substitute out individual RAM cells. Rather, with a technique like row sparing, the substitutable unit is an entire row. In the simplest sparing schemes, a defect anywhere within a substitutable unit may force the discard of the entire element. Consequently, the granularity of substitution can play a big role in the viable yield of a component (see the Perfect yield subsection that follows). Section 37.2.5 examines more sophisticated sparing schemes that relax this constraint.

37.2.3 Yield

This section reviews simple calculations for the yield of components and substitutable units. We assume uniform device defect rates and independent, random failure (i.e., identical, independently distributed—iid). Using these simple models, we can illustrate the kinds of calculations involved and build intuition on the major trends.

Perfect yield
A component with no substitutable units will be nondefective only if all the devices in the unit are not defective. Similarly, in the simplest models each substitutable unit is nondefective only when all of its constituent devices are not defective. If we have a device defect probability P_d and if a unit contains N devices, the probability that the entire component or unit is nondefective is:

$$P_{defect-free}(N, P_d) = (1 - P_d)^N \tag{37.1}$$

We can expand this as a binomial:

$$P_{defect-free}(N, P_d) = \sum_i \left(\binom{N}{i} (-P_d)^i \right) = 1 - N \cdot P_d + \binom{N}{2} (P_d)^2 - \ldots \tag{37.2}$$

If $N \times P_d << 1$, then we observe that each successive power of P_d is much smaller than the previous term. We can approximate this yield as:

$$P_{defect-free}(N, P_d) \approx 1 - N \cdot P_d \tag{37.3}$$

This tells us we have a substitutable unit defect rate, P_{sd}, or a component defect rate, roughly equal to the product of the number of devices and the device defect rate:

$$P_{sd}(N, P_d) \approx N \cdot P_d \tag{37.4}$$

This simple equation indicates several things:

- For today's large chips with $N > 10^9$ devices, the defect rate P_d must be below 10^{-10} to expect 90 percent or greater chip yield.
- To maintain constant yield ($P_{defect-free}$) for a chip as N scales, we must continually decrease P_d at the same rate. For example, a 10× increase in device count, N, must be accompanied by a 10× decrease in per-device defect rate.
- As noted in this chapter's introduction, we expect the opposite effect for atomic-scale devices; smaller devices mean a higher likelihood of defects. This exacerbates the challenge of increasing device counts.
- At the same defect rate, P_d, a finer-grained substitutable unit (e.g., an individual LUT or bitop) will have a higher unit yield rate than a coarser-grained unit (e.g., a cluster of 10 LUTs, such as an Altera LAB (Section 1.5.1) or an SIMD collection of 32 bitops). Alternatively, if one reasons about defect rates of the substitutable units, a defect rate of $P_{sd} = 0.05$ for a coarse-grained block corresponds to a much lower device defect rate, P_d, than the same P_{sd} for a fine-grained substitutable unit.

- To keep substitutable unit yield rates at some high value, we must decrease unit size, N, as P_d increases. For example, if we design for a $P_{sd} = 10^{-4}$ and the device defect rate doubles, we need to cut the substitutable block size in half to achieve the same block yield; this suggests a trend toward fine-grained resource sparing as defect rates increase (e.g., see Fine-grained Pterm matching subsection of Section 37.2.5 and Section 38.6).

Yield with sparing

We can significantly increase overall yield by providing spares so that there is no need to demand that every substitutable unit be nondefective. Assume for now that all substitutable units are interchangeable. The probability that we will have exactly i nondefective substitutable units is:

$$P_{yield}(N,i) = \left(\binom{N}{i}(P_{sd})^i(1-P_{sd})^{N-i}\right) \tag{37.5}$$

That is, there are $\binom{N}{i}$ ways to select i nondefective blocks from N total blocks, and the yield probability of each case is $(P_{sd})^i(1-P_{or})^{N-i}$. An ensemble with at least M items is obtained whenever M or more items yield, so the ensemble yield is actually the cumulative distribution function, as follows:

$$P_{yield}(N,M) = \sum_{M\le i\le N}\left(\binom{N}{i}(P_{sd})^i(1-P_{sd})^{N-i}\right) \tag{37.6}$$

As an example, consider an Island-style FPGA cluster (see Figure 37.2) composed of 10 LUTs (e.g., Altera LAB, Chapter 1). Assume that each LUT, along with its associated interconnect and configuration, is a substitutable unit and that the LUTs are interchangeable. Further, assume $P_{sd} = 10^{-4}$. The probability of yielding all 10 LUTs is:

$$P_{yield}(10,10) = \left(10^{-4}\right)^{10}\left(1-10^{-4}\right)^{0} \approx 0.9990005 \tag{37.7}$$

Now, if we add two spare lookup tables, the probability of yielding at least 10 LUTs is:

$$\begin{aligned} P_{yield}(12,10) &= \left(10^{-4}\right)^{12}\left(1-10^{-4}\right)^{0} + 12\left(10^{-4}\right)^{11}\left(1-10^{-4}\right)^{1} \\ &\quad + \frac{12\cdot 11}{2}\left(10^{-4}\right)^{10}\left(1-10^{-4}\right)^{2} \\ &= 0.99880065978 + 0.0011986806598 + 0.0000006593402969 \\ &\approx 0.9999999998 > 1-10^{-9} \end{aligned} \tag{37.8}$$

Without the spares, a component with only 1000 such clusters would be difficult to yield. With the spares, components with 1,000,000 such clusters yield more than 99.9 percent of the time.

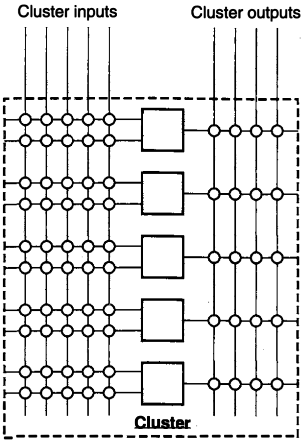


FIGURE 37.2 ■ An island-style FPGA cluster with five interchangeable 2-LUTs.

The assumption that all substitutable units are interchangeable is not directly applicable to logic blocks in an FPGA since their location strongly impacts the interconnections available to other logic block positions. Nonetheless, the sparing yield is illustrative of the trends even when considering interconnect requirements.

To minimize the required spares, it would be preferable to have fewer large pools of mostly interchangeable resources rather than many smaller pools of interchangeable resources. This results from Bernoulli's Law of Large Numbers (the Central Limit Theorem) effects [3, 4], where the variance of a sum of random variables decreases as the number of variables increases. For a more detailed development of the impact of the Law of Large Numbers on defect yield statistics and strategies see DeHon [5].

37.2.4 Defect Tolerance through Sparing

To exploit substitution, we need to locate the defects and then avoid them. Both testing (see next subsection) and avoidance could require considerable time for each individual device. This section reviews several design approaches, including approaches that exploit full mapping (see the Global sparing subsection) to minimize defect tolerance overhead, approaches that avoid any extra mapping (see the Perfect component model subsection), and approaches that require only minimal, local component-specific mapping (see the Local sparing subsection).

Testing

Traditional acceptance testing for FPGAs (e.g., [6]) attempts to validate that the FPGA is defect free. Locating the position of any defect is generally not important if any chip with defects is discarded. Identifying the location of all defects is more difficult and potentially more time consuming. Recent work on group testing [7–9] has demonstrated that it is possible to identify most of the nondefective resources on a chip with N substitutable components in time proportional to $\sqrt{N}$.

In group testing, substitutable blocks are configured together and given a self-test computation to perform. If the group comes back with the correct result, this is evidence that everything in the group is good. Conversely, if the result is wrong, this is evidence that something in the group may be bad. By arranging multiple tests where substitutable blocks participate in different groups (e.g., one test set groups blocks around rows while another groups them along columns), it is possible to identify which substitutable units are causing the failures.

For example, if there is only one failure in each of two groupings, and the failing groups in each grouping contain a single, common unit, this is strong evidence that the common unit is defective while the rest of the substitutable units are good. As the failure rates increase such that multiple elements in each group fail in a grouping, it can be more challenging to precisely identify failing components with a small number of groupings. As a result, some group testing is conservative, marking some good components as potential defects; this is a trade-off that may be worthwhile to keep testing time down to a manageably low level as defect rates increase.

In both group testing and normal FPGA acceptance testing, array regularity and homogeneity make it possible to run tests in parallel for all substitutable units on the component. Consequently, testing time does not need to scale as the number of substitutable units, N. If the test infrastructure is reliable, group tests can run completely independently. However, if we rely on the configurable logic itself to manage tests and route results to the test manager, it may be necessary to validate portions of the array before continuing with later tests. In such cases, testing can be performed as a parallel wave from a core test manager, testing the entire two-dimensional device in time proportional to the square root of the number of substitutable units (e.g., [8]).

Global sparing

A defect map approach coupled with component-specific mapping imposes low overhead for defect tolerance. Given a complete map of the defects, we perform a component-specific design mapping to avoid the defects. Defective substitutable units are marked as bad, and scheduling, placement, and routing are performed to avoid these resources. An annealing placer (Chapter 14) can mark the physical location of the defective units as invalid or expensive and penalize any attempts to assign computations to them. Similarly, a router (Chapter 17) can mark defective wires and switches as "in use" or very costly so that they are avoided. The Teramac custom-computing machine tolerated a 10 percent defect rate in logic cells ($P_{sd_{logic}} = 0.10$) and a 3 percent defect rate in on-chip interconnect ($P_{sd_{interconnect}} = 0.03$) using group testing and component-specific mapping [7].

With place-and-route times sometimes running into hours or days, the component-specific mapping approach achieves low overhead for defect tolerance at the expense of longer mapping times. As introduced in Chapter 20, there are several techniques we could employ to reduce this mapping time, including:

- Tuning architectures to facilitate faster mapping by overprovisioning resources and using simple architectures that admit simple mapping; the Plasma chip—an FPGA-like component, which was the basis of the Teramac architecture—takes this approach and was highlighted in Chapter 20.
- Trading mapping quality in order to reduce mapping time.
- Using hardware to accelerate placement and routing (also illustrated in Sections 9.4.2 and 9.4.3).

Perfect component model

To avoid the cost of component-specific mapping, an alternate technique to use is the perfect component model (Section 37.2.1). Here, the goal is to use the defect map to preconfigure the allocation of spares so that the component looks to the user like a perfect component. Like row or column sparing in memory, entire rows or columns may be the substitutable units. Since reconfigurable arrays, unlike memories, have communication lines between blocks, row or column sparing is much more expensive to support than in memories. All interconnect lines must be longer, and consequently slower, to allow configuration to reach across defective rows or columns. The interconnect architecture must be designed such that this stretching across a defective row is possible, which can be difficult in interconnects with many short wires (see Figure 37.3).

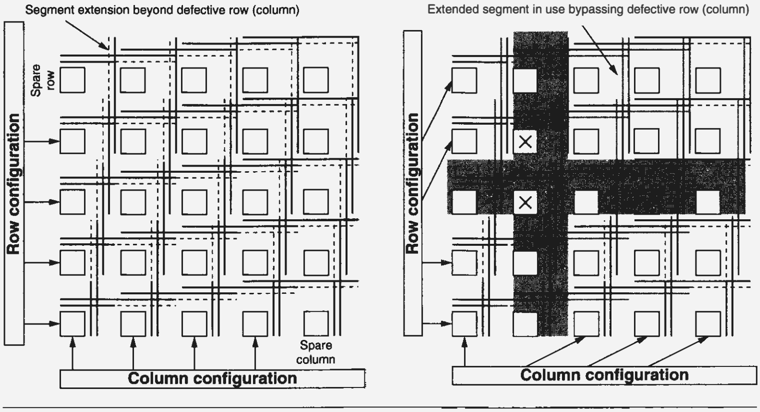

Segment extension beyond defective row (column)
Spare row
Row configuration
Spare column
Column configuration
Extended segment in use bypassing defective row (column)
Row configuration
Column configuration

FIGURE 37.3 ■ Arrays designed to support row and column sparing.

A row of FPGA logic blocks is a much coarser substitutable unit than a memory row. FPGAs from Altera have used this kind of sparing to improve component yield [10,11], including the Apex 20KE series.

Local sparing

With appropriate architecture or stylized design methodology, it is possible to avoid the need to fully remap the user design to accommodate the defect map. The idea here is to guarantee that it is possible to locally transform the design to avoid defects. For example, in cases where all the LUTs in a cluster are interchangeable, if we provision spares within each cluster as illustrated earlier in the Yield with sparing subsection of Section 37.2.3, it is simply a matter of locally reassigning the functions to LUTs to avoid the defective LUTs.

For regular arrays, Lach et al. [12] show how to support local interchange at a higher level without demanding that the LUTs exist in a locally interchangeable cluster. Consider a $k \times k$ tile in the regular array. Reserve s spares within each $k \times k$ tile so that we only populate $(k^2 - s)$ LUTs in each such region. We can now compute placements for the $(k^2 - s)$ LUTs for each of the possible combinations of s defects. In the simplest case, $s = 1$, we precalculate k^2 placements for each region (e.g., see Figure 37.4). Once we have a defect map, as long as each region has fewer than s errors, we simply assemble the entire configuration by selecting an appropriate configuration for each tile.

When a routing channel provides full crossbar connectivity, similarly, it may be possible to locally swap interconnect assignments. However, typical FPGA routing architectures do not use fully populated switching; as a result, interconnect sparing is not a local change. Yu and Lemieux [13,14] show that FPGA switchboxes can be augmented to allow local sparing at the expense of 10 to 50 percent of area overhead. The key idea is to add flexibility to each switchbox that allows a route to shift one (or more) wire track(s) up or down; this allows routes to be locally redirected around broken tracks or switches and then restored to their normal track (see Figure 37.5).

To accommodate a particular defect rate and yield target, local interchange will require more spares than global mapping (see the Global sparing subsection). Consider any of the local strategies discussed in this section where we allocate one spare in each local interchange region (e.g., cluster, tile, or channel). If there are two defects in one such region, the component will not be repairable. However, the component may well have adequate spares; they are just assigned to different interchange regions. With the same number of resources, a global remapping would be able to accommodate the design. Consequently, to achieve the same yield rate as the global scheme, the local scheme always has to allocate more spares. This is another consequence of the Law of Large Numbers (see the Yield with sparing subsection):

> *The more locally we try to contain replacement, the higher variance we must accommodate, and the larger overhead we pay to guarantee adequate yield.*

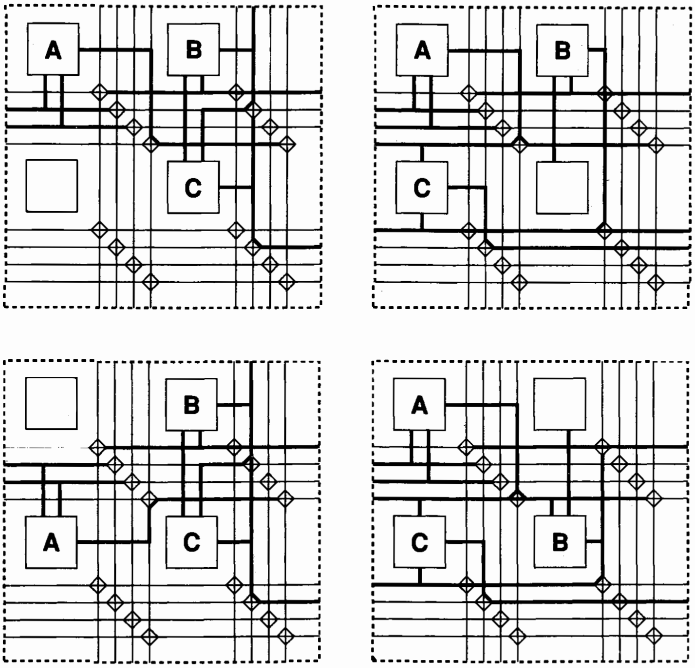


FIGURE 37.4 ■ Four placements of a three-gate subgraph on a 2 × 2 tile.

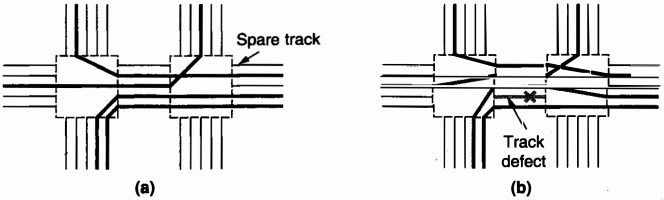


FIGURE 37.5 ■ Added switchbox flexibility allows local routing around interconnect defects: (a) defect free with spare and (b) configuration avoiding defective track.

37.2.5 Defect Tolerance with Matching

In the simple sparing case (Section 37.2.4), we test to see whether each substitutable unit is defect free. Substitutable units with defects are then avoided. This works well for low-defect rates such that P_{sd} remains low. However, it can also be highly conservative. In particular, not all capabilities of the substitutable unit are always needed. A configuration of the substitutable unit that avoids the particular defect may still work correctly. Examples where we may not need to use all the devices inside a substitutable unit include the following:

- A typical FPGA logic block, logic element, or slice includes an optional flip-flop and carry-chain logic. Many of the logic blocks in the user's design leave the flip-flop or carry chain unused. Consequently, these "defective" blocks may still be usable, just for a subset of the logical blocks in the user's design.
- When the substitutable unit is a collection of W_{simd} bitops, a defect in one of the bitops leaves the unit imperfect. However, the unit may work fine on smaller data. For example, maybe a $W_{simd} = 8$ substitutable unit has a defect in bit position 5. If the application requires some computations on $W_{app} = 4$ bit data elements, the defective 8-bit unit may still perform adequately to support 4 bitops.
- A product term (Pterm) in a programmable logic array (PLA) or programmable array logic (PAL) is typically a substitutable unit. Each Pterm can be configured to compute the AND of any of the inputs to the array (see Figure 37.6). However, all the Pterms configured in the array will never need to be connected to all the inputs. Consequently, defects that prevent a Pterm from connecting to a subset of the inputs may not inhibit it from being configured to implement some of the Pterms required to configure the user's logic.

Instead of discarding substitutable units with defects, we characterize their capabilities. Then, for each logical configuration of the substitutable unit

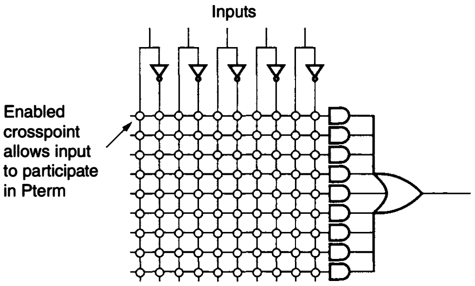


FIGURE 37.6 ▪ A PAL OR-term with a collection of substitutable Pterm inputs.

demanded by the user's application, we can identify the set of (potentially defective) substitutable units capable of supporting the required configuration. Our mapping then needs to ensure that assignments of logical configurations to physical substitutable units obey the compatibility requirements.

Matching formulation

To support the use of partially defective units as substitutable elements, we can formulate the mapping between logical configurations and substitutable units as a bipartite matching problem. For simplicity and exposition, it is assumed that all the substitutable units are interchangeable. This is likely to be an accurate assumption for LUTs in a cluster or Pterms in a PAL or PLA, but it is not an accurate assumption for clusters in a two-dimensional FPGA routing array. Nonetheless, this assumption allows precise formulation of the simplest version of the problem.

We start by creating two sets of nodes. One set, $R = \{r_0, r_1, r_2 \ldots\}$, represents the physical substitutable resources. The second set, $L = \{l_0, l_1, l_2 \ldots\}$, represents the logic computations from the user's design that must be mapped to these substitutable units. We add a link (l_i, r_j) if-and-only-if logical configuration l_i can be supported by physical resource r_j. This results in a bipartite graph, with L being one side of the graph and R being the other. What we want to find is a *complete matching* between nodes in L and nodes in R—that is, we want every $l_i \in L$ to be matched with exactly one node $r_j \in R$, and every node $r_j \in R$ to be matched with at most one node $l_i \in L$.

We can optimally compute the *maximal matching* between L and R in polynomial time using the Ford–Fulkerson maximum flow algorithm [15] with time complexity $O(|V| \cdot |E|)$ or a Hopcroft–Karp algorithm [16] with time complexity $O\left(\sqrt{|V|} \cdot |E|\right)$. In the graph, $|V| = |L| + |R|$ and $|E| = O(|L| \cdot |R|)$. Since there must be at least as many resources as logical configurations, $|L| \leq |R|$, the Hopcroft–Karp algorithm is thus $O\left(|R|^{2.5}\right)$; for local sparing schemes, $|R|$ might be reasonably in the 10 to 100 range, meaning that the matching problem is neither large nor growing with array size. If the maximal matching fails to be a complete matching (i.e., assign each l_i to a unique match in r_i), we know that it is not possible to support the design on a particular set of defective resources.

Fine-grained Pterm matching

Naeimi and DeHon use this matching to assign logical Pterms to physical nanowires in a nanoPLA (Chapter 38, Section 38.6) [17, 18]. Before considering defects, all the Pterm nanowires in the PLA are freely interchangeable. Each nanowire that implements a Pterm has a programmable diode between the input nanowires and the nanowire itself. If the diode is programmed into an off state, it disconnects the input from the nanowire Pterm. If the diode is in the on state, it connects the input to the nanowire, allowing it to participate in the AND that the Pterm is computing.

The most common defect anticipated in this technology is that the programmable diode is stuck in an off state—that is, it cannot be programmed into a valid on state. Consequently, a Pterm nanowire with a stuck-off diode at a particular input location cannot be programmed to include that input in the AND it is performing.

A typical PLA will have 100 inputs, meaning each product-term nanowire is connected to 100 programmable diodes. A plausible failure rate for the product-term diodes is 5% ($P_d = 0.05$). If we demanded that each Pterm be defect free in order to use it, the yield of product terms would be:

$$P_{nwpterm}(100, 0.05) = (1 - 0.05)^{100} \approx 0.006 \quad (37.9)$$

However, since none of the product terms use all 100 inputs, the probability that a particular Pterm nanowire can support a logical Pterm is much higher. For example, if the Pterm only uses 10 inputs, then the probability that a particular Pterm nanowire can support it is:

$$P_{nwpterm}(10, 0.05) = (1 - 0.05)^{10} \approx 0.599 \quad (37.10)$$

Further, typical arrays will have 100 product-term nanowires. This suggests that, on average, this Pterm will be compatible with roughly 60 of the Pterm nanowires in the array—that is, the l_i for this Pterm will end up with compatibility edges to 60 r_j's in the bipartite matching graph described before.

As a result, DeHon and Naeimi [18] were able to demonstrate that we can tolerate stuck-off diode defects at $P_d = 0.05$ with no allocated spare nanowires. In other words, we can have $|L|$ as large as $|R|$ and, in practice, always find a complete matching for every PLA. This is true even though the probability of a perfect nanowire is below 1 percent (equation 37.9), suggesting that most arrays of 100 nanowires contain no perfect Pterm nanowires.

This strategy follows the defect map model and does demand component-specific mapping. Nonetheless, the required mapping is local (see the Local sparing section) and can be fast. Naeimi and DeHon [17] demonstrate the results quoted previously using a greedy, linear-time assignment algorithm rather than the slower, optimal algorithm. Further, if it is possible to test the compatibility of each Pterm as part of the trial assignment, it is not necessary to know the defect map prior to mapping.

FPGA component level

It is also possible to apply this matching idea at the component level. Here, the substitutable unit is an entire FPGA component. Unused resources will be switches, wires, and LUTs that are not used by a specific user design. Certainly, if the specific design does not fill the logic blocks in the component, there will be unused logic blocks whose failure may be irrelevant to the proper functioning of the design. Even if the specific design uses all the logic blocks, it will not use all the wires or all the features of every logic block. So, as long as the defects in the component do not intersect with the resources used by an particular FPGA configuration, the FPGA can perfectly support the configuration.

Xilinx's EasyPath series is one manifestation of this idea. At a reduced cost compared to perfect FPGAs, Xilinx sells FPGAs that are only guaranteed to work with a particular user design, or a particular set of user designs. The user provides their designs, and Xilinx checks to see whether any of their defective devices will successfully implement those designs. Here, Xilinx's resource set, R, is the nonperfect FPGAs that do not have defects in the nonrepairable portion of the logic. The logical set, L, is the set of customer designs destined for Easy-Path. Xilinx effectively performs the matching and then supplies each customer with FPGA components compatible with their respective designs.

Hyder and Wawrzynek [19] demonstrate that the same idea can be exploited in board-level FPGA systems. Here, their resource set, R, is the set of FPGAs on a particular board with multiple FPGAs. Their logical set is the set of FPGA configurations intended for the board. If all the FPGAs on the board were interchangeable, this would also reduce to the previous simple matching problem. However, in practice, the FPGAs on a board typically have different connections. This provides an additional set of topological constraints that must be considered along with resource compatibility during assignment. Rather than creating and maintaining a full defect map of each FPGA in the system, they also use application-specific testing (e.g., Tahoori [20]) to determine whether a particular FPGA configuration is compatible with a specific component on the FPGA board.

37.3 TRANSIENT FAULT TOLERANCE

Recall that transient faults are randomly occurring, temporary deviations from the correct circuit behavior. It is not possible to test for transient faults and configure around them as we did with defects. The impact of a transient fault depends on the structure of the logic and the location of the transient fault. The fault may be masked (hidden by downstream gates that are not currently sensitive to this input), may simply affect the circuit output temporarily, or may corrupt state so that the effect of the transient error persists in the computation long after the fault has occurred. Examples include the following:

- If both inputs to an OR gate should be 1, but one of the inputs is erroneously 0, the output of the OR gate will still have the correct value.
- If the transient fault impacts the combinational output from a circuit, only the output on that cycle is affected; subsequent output cycles will be correct until another transient fault occurs.
- If the transient fault results in the circuit incorrectly calculating the next state transition in a finite-state machine (FSM), the computation may proceed in the incorrect state for an indefinite period of time.

To deal with the general case where transient faults impact the observable behavior of the computation, we must be able to prevent the errors from propagating into critical state or to observable outputs from the computation. This demands that we add or exploit some form of redundancy in the calculation to detect or correct errors as they occur. This section reviews two general approaches to transient fault tolerance: feedforward correction (Section 37.3.1) and rollback error recovery (Section 37.3.2).

37.3.1 Feedforward Correction

One common strategy to tolerate transient faults is to provide adequate redundancy to correct any errors that occur. This allows the computation to continue without interruption. The simplest example of this redundancy is replication. That is, we arrange to perform the intended computation *R* times and vote on the result, using the majority result as the value allowed to update state or to be sent to the output. The smallest example uses $R = 3$ and is known as triple modular redundancy (TMR) (see Figure 37.7). In general, for there to be a clear majority, *R* must be odd, and a system with *R* replicas can tolerate at least $\frac{R-1}{2}$ simultaneous transient faults. We can perform the multiple calculations either in space, by concurrently placing *R* copies of the computation on the reconfigurable array, or in time, by performing the computation multiple times on the same datapath.

In the simple design in Figure 37.7, a failure in the voter may still corrupt the computation. This can be treated similarly to nonrepairable area in defect-tolerance schemes:

- If the computation is large compared to the voter, the probability of voter failure may be sufficiently small so that it is acceptable.
- The voter can be implemented in a more reliable technology, such as a coarser-grained feature size.
- The voter can be replicated as well. For example, von Neumann [21] and Pippenger [22] showed that one can tolerate high transient fault rates (up to 0.4 percent) using a gate-level TMR scheme with replicated voters.

TMR strategies have been applied to Xilinx's Virtex series [23]. Rollins et al. [24] evaluate various TMR schemes on Virtex components, including strategies with replicated voters and replicated clock distribution.

A key design choice in modular redundancy schemes is the granularity at which voting occurs. At the coarsest grain, the entire computational circuit could be the unit of replication and voting. At the opposite extreme, we can replicate and vote individual gates as the Von Neumann design suggests. The appropriate choice will balance area overhead and fault rate. From an area

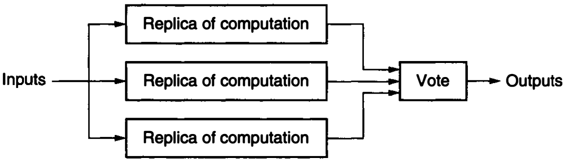


FIGURE 37.7 ■ A simple TMR design.

overhead standpoint, we would prefer to vote on large blocks; this allows the area of the voters to be amortized across large logic blocks so that the total area grows roughly as the replication factor, R. From an area overhead standpoint, we also want to keep R low. From a reliability standpoint, we want to make it sufficiently unlikely that more than $\frac{R-1}{2}$ replicas are corrupted by transient errors in a single cycle. Similar to defects (equation 37.4), the failure rate of a computation, and hence a replica, scales with the number of devices in the computation and the transient fault rate per device; consequently, we want to scale the unit of replication down as fault rate increases to achieve a target reliability with low R.

Memory

A common form of feedforward correction is in use today in memories. Memories have traditionally been the most fault-sensitive portions of components because: (1) A value in a memory may not be updated for a large number of cycles; as such, memories integrate faults over many cycles. (2) Memories are optimized for density; as such, they often have low capacitance and drive strength, making them more susceptible to errors.

We could simply replicate memories, storing each value in R memories or memory slots and voting the results. However, over the years information theory research has developed clever encoding schemes that are much more efficient for protecting groups of data bits than simple replication [25,26]. For example, DRAMs used in main memory applications generally tolerate a single-bit fault in a 64-bit data-word using a 72-bit error correcting code. Like the nonrepairable area in DRAMs, the error correcting circuitry in memories is generally built from coarser technology than the RAM memory array and is assumed to be fault free.

37.3.2 Rollback Error Recovery

An alternative technique to feedforward correction is to simply detect when errors occur and repeat the computation when an error is detected. We can detect errors with less redundancy than we need to correct errors (e.g., two copies of a computation are sufficient to detect a single error, while three are required for correction); consequently, detection schemes generally require lower overhead than feedforward correction schemes. If fault rates are low, it is uncommon for errors to occur in the logic. In most cycles, no errors occur and the normal computation proceeds uninterrupted. In the uncommon case in which a transient fault does occur, we stop processing and repeat the computation in time without additional hardware. With reasonably low transient-fault rates, it is highly unlikely that repeated computation will also be in error; in any case, detection guards against errors in the repeated computation as well.

To be viable, the rollback technique demands that the application tolerate stalls in computation during rollback. This is easily accommodated in streaming models (Chapter 5, Section 5.1.3) that exploit data-presence signaling (see Data presence subsection of Section 5.2.1) to tolerate variable timing for operator implementations. When detection and rollback are performed on an operator level, stream buffers between operator datapaths can isolate and minimize the performance impact of rollback.

Detection

To detect errors we use some form of redundancy. Again, this can be either temporal or spatial redundancy.

To minimize the performance impact, we can employ a *concurrent-error detection* (CED) technique—that is, in parallel with the normal logic, we compute some additional function or property of the output (see Figure 37.8). We continuously check consistency between the logical output and this concurrent calculation. If the concurrent calculation ever disagrees with the base computation, this means there is an error in the logic.

In the simplest case, the parallel function could be a duplicate copy of the intended logic (see Figure 37.8(b)). Checking then consists of verifying that the two computations obtained the equivalent results. However, it is often possible to avoid recomputing the entire function and, instead, compute a property of the output, such as its parity (see Figure 37.8(c)) [27].

The choice of detection granularity is based on the same basic considerations discussed before for feedforward replica granularity. Larger blocks can amortize out comparison overhead but will increase block error rates and hence the rate of rollback. For a given fault rate, we reduce comparison block granularity until the rollback rate is sufficiently low so that it has little impact on system throughput.

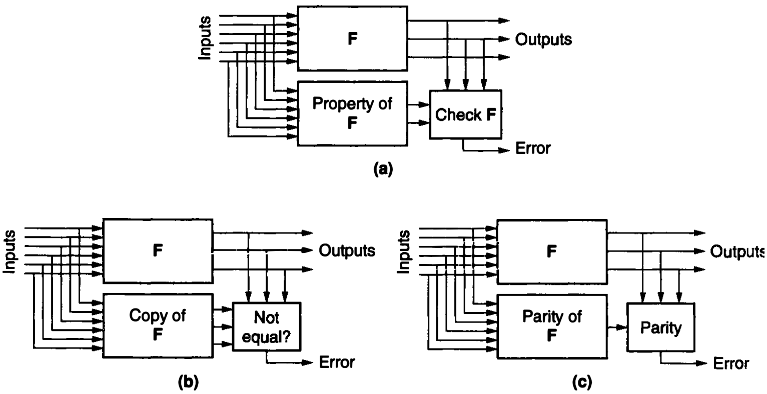


FIGURE 37.8 ▪ A concurrent error-detection strategy and options: (a) generic formulation, (b) duplication, and (c) parity.

Recovery

When we do detect an error, it is necessary to repeat the computation. This typically means making sure to preserve the inputs to a computation until we can be certain that we have reliably produced a correct result. Conceptually, we read inputs and current state, calculate outputs, detect errors, then produce outputs and save state if no errors are detected. In practice, we often want to pipeline this computation so that we detect errors from a previous cycle while the computation continues, and we may not save state to a reliable storage on every calculation. However, even in sequential cases, it may be more efficient to perform a sequence of computations between error checks.

A common idiom is to periodically store, or *snapshot*, state to reliable memory, store inputs as they arrive into reliable memory, perform a series of data computations, and store results to reliable memory. If no errors are detected between snapshots, then we continue to compute with the new state and discard the inputs used to produce it. If errors are detected, we discard the new state, restore the old state, and rerun the computation using the inputs stored in reliable memory. As noted earlier in the Memory subsection, we have particularly compact techniques for storing data reliably in fault-prone memories; this efficient protection of memories allows rollback recovery techniques to be robust and efficient.

In streaming systems, we already have FIFO streams of data between operators. We can exploit these memories to support rollback and retry. Rather than discarding the data as soon as the operator reads it, we keep it in the FIFO but advance the head pointer past it. If the operator needs to rollback, we effectively reset the head pointer in the FIFO to recover the data for reexecution. When an output is correctly produced and stored in an output FIFO, we can then discard the associated inputs from the input FIFOs. For operators that have bounded depth from input to output, we typically know that we can discard an input set for every output produced.

Communications

Data transmission between two distant points, especially when it involves crossing between chips and computers, is highly susceptible to external noise (e.g., crosstalk from nearby wires, power supply noise, clock jitter, interference from RF devices). As such, for a long time we have protected communication channels with redundancy. As with memories, we simply need to reliably deliver the data sent to the destination.

Unlike memories, we do not necessarily need to guarantee that the correct data can be recovered from the potentially corrupted data that arrive at the destination. When the data are corrupted in transmission, it suffices to detect the error. The sender holds onto a copy of the data until the receiver indicates they have been successfully received. When an error is detected, the sender can retransmit the data. The detection and retransmission are effectively a rollback technique.

When the error rates on the communication link are low, such that error detection is the uncommon event, this allows data to be protected with low overhead error-detecting codes, or *checksums*, instead of more expensive error correcting codes. The Transmission Control Protocol (TCP) used for communication across the Internet includes packet checksums and retransmission when data fail to arrive error free at the intended destination [28].

37.4 LIFETIME DEFECTS

Over the lifetime of a component, the physical device will change and degrade, potentially introducing new defects into the device. Individual atomic bonds may break or metal may migrate, increasing the resistance of the path or even breaking a connection completely. Device characteristics may shift because of hot-carrier injection (e.g., [29,30]), NBTI (e.g., [31]), or even accumulated radiation doses (e.g., [32, 33]). These effects become more acute as feature sizes shrink. To maintain correct operation, we must detect the errors (Section 37.4.1) and repair them (Section 37.4.2) during the lifetime of the component.

37.4.1 Detection

One way to detect lifetime failures is to periodically retest the device—that is, we stop normal operation, run a testing routine (see the Testing subsection in Section 37.2.4), then resume normal operation if there are no errors. It can be an application-specific test, determining whether the FPGA can still support the user's mapping [20], or an application-independent test of the FPGA substrate. Application-specific tests have the advantage of both being more compact and ignoring new defects that do not impact the current design. Substrate tests may require additional computation to determine whether the newly defective devices will impact the design. While two consecutive, successful tests generally mean that the computation between these two points was correct, the component may begin producing errors at any time inside the interval between tests and the error will not be detected until the next test is run.

Testing can also be interleaved more directly with operation. In partially reconfigurable components (see Section 4.2.3), it is possible to reconfigure portions of a component while the rest of the component continues operating. This allows the reservation of a fraction of the component for testing. If we then arrange to change the specific portions of the component assigned to testing and operation over time, we can incrementally test the entire component without completely pulling it out of service (e.g., [34,35]).

In some scenarios, the component may need to stall operation during the partial reconfiguration, but the component only needs to stall for the reconfiguration period and not the entire testing period. When the total partial reconfiguration time is significantly shorter than the testing time, this can reduce the fraction of cycles the application must be removed from normal operation. This still means that we may not detect the presence of a new defect until long after it occurred and started corrupting data.

If it is necessary to detect an error immediately, we must employ one of the fault tolerance techniques reviewed in Section 37.3. CED (see the Detection subsection in Section 37.3.2) can identify an error as soon as it occurs and stall computation. TMR (Section 37.3.1) can continue correct operation if only a single replica is affected; the TMR scheme can be augmented to signal higher-level control mechanisms when the voters detect disagreement.

37.4.2 Repair

Once a new error has occurred, we can repeat global (see the Global sparing subsection in Section 37.2.4) or local mapping (see the Local sparing subsection in Section 37.2.4) to avoid the new error. However, since the new defect map is most likely to differ from the old defect map by only one or a few defects, it is often easier and faster to incrementally repair the configuration. In local mapping schemes, we only need to perform local remapping in the interchangeable region(s) where the new defect(s) have occurred. This may mean that we only need to move LUTs in a single cluster, wires in channel, or remap a single tile. Even in global schemes the incremental work required may be modest. Lakamraju and Tessier [36] show that incrementally rerouting connections severed by new lifetime defects can be orders of magnitude faster than performing a complete reroute from scratch.

A rollback scheme (Section 37.3.2) can stall execution during the repair. A replicated, feedforward scheme (Section 37.3.1) with partial reconfiguration may be able to continue operating on the functional replicas while the newly defective replica is being repaired.

Lifetime repair strategies depend on the ability to perform defect mapping and reconfiguration. Consequently, the perfect component model cannot support lifetime repair. Even if the component retains spare redundancy, redundancy and remapping mechanisms are not exposed to the user for in-field use.

37.5 CONFIGURATION UPSETS

Many reconfigurable components, such as FPGAs, rely on volatile memory cells to hold their configuration, typically static memory cells (e.g., SRAM). Dynamic memory cells have long had to cope with upsets from ionizing particles (e.g., α-particles). As the feature sizes shrink, even static RAM cells can be upset by ionizing particles (e.g., Harel et al. [37]). In storage applications, we can typically cope with memory soft errors using error correcting codes (see the Memory subsection in Section 37.3.1) so that bit upsets can be detected and corrected However, in reconfigurable components, we use the memory cells directly and continuously as configuration bits to define logic and interconnect. Upsets of these configuration memories will change, and potentially corrupt, the logic operation.

Unfortunately, although memories can amortize the cost of a large error correction unit across a deep memory, FPGA configurations are shallow (i.e., $N_{instr} = 1$); an error correction scheme similar to DRAM memories would end up being as large as or larger than the configuration memory it protects. Data and projections from Quinn and Graham [38] suggest that ionizing radiation upsets can be a real concern for current, large FPGA-based systems and will be an ongoing concern even for modest systems as capacity continues to increase.

Because these are transient upsets of configuration memories, they can be corrected simply by reloading the correct bitstream once we detect that the bitstream has been corrupted. Logic corruption can be detected using any of the strategies described earlier for lifetime defects (Section 37.4.1). Alternatively, we can check the bitstream directly for errors. That is, we can compute a checksum for the correct bitstream, read the bitstream back periodically, compute the checksum of the readback bitstream, and compare it to the intended bitstream checksum to detect when errors have occurred. When an error has occurred, the bitstream can be reloaded [38,39]. Like interleaved testing, bitstream readback introduces a latency, which can be seconds long, between configuration corruption and correction. If the application can tolerate infrequent corruption, this may be acceptable.

Asadi and Tahoori [40] detail a rollback scheme for tolerating configuration upsets. Pratt et al. [41] use TMR and partial TMR schemes to tolerate configuration upsets; their partial TMR scheme uses less area than a full TMR scheme in cases where it is acceptable for the outputs to be erroneous for a number of cycles as long as the state is protected so that the results return to the correct values when the configuration is repaired.

37.6 OUTLOOK

The regularity in reconfigurable arrays, coupled with the resource configurability they already possess, allow these architectures to tolerate defects. As features shrink and defect rates increase, all devices, including ASICs, are likely to need some level of regularity and configurability; this will be one factor that serves to narrow the density and cost gap between FPGAs and ASICs. Further, at increased defect rates, it will likely make sense to ship components with defects and defect maps. Since each component will be different, some form of component-specific mapping will be necessary.

Transient upsets and lifetime defects further suggest that we should continuously monitor the computation to detect errors. To tolerate lifetime defects, repair will become part of the support system for components throughout their operational lifetime. Increasing defect rates further drive us toward architectures with finer-grained substitutable units. FPGAs are already fairly fine grained, with each bit-processing operator potentially serving as a substitutable unit, but finer-grained architectures that substitute individual wires, Pterms, or LUTs may be necessary to exploit the most aggressive technologies.

References

[1] S. E. Schuster. Multiple word/bit line redundancy for semiconductor memories. *IEEE Journal of Solid State Circuits* 13(5), 1978.

[2] B. Keeth, R. J. Baker. *DRAM Circuit Design: A Tutorial.* Microelectronic Systems, IEEE Press, 2001.

[3] J. Bernoulli. *Ars Conjectandi.* Impensis thurnisiorum, fratrum, Basel, Switzerland, 1713.

[4] A. W. Drake. *Fundamentals of Applied Probability Theory*, McGraw-Hill, 1988.

[5] A. DeHon. Law of large numbers system design. *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, S. K. Shukla, R. I. Bahar (eds.), Kluwer Academic, 2004.

[6] W. K. Huang, F. J. Meyer, X.-T. Chen, F. Lombardi. Testing configurable LUT-based FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6(2), 1998.

[7] W. B. Culbertson, R. Amerson, R. Carter, P. Kuekes, G. Snider. Defect tolerance on the TERAMAC custom computer. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[8] M. Mishra, S. C. Goldstein. Defect tolerance at the end of the roadmap. *Proceedings of the International Test Conference (ITC)*, 2003.

[9] M. Mishra, S. C. Goldstein. Defect tolerance at the end of the roadmap. *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, S. K. Shukla, R. I. Bahar (Eds.), Kluwer Academic, 2004.

[10] R. G. Cliff, R. Raman, S. T. Reddy. Programmable logic devices with spare circuits for replacement of defects. U.S. Patent number 5,434,514, July 18, 1995.

[11] C. McClintock, A. L. Lee, R. G. Cliff. Redundancy circuitry for logic circuits. U.S. Patent number 6,034,536, March 7, 2000.

[12] J. Lach, W. H. Mangione-Smith, M. Potkonjak. Low overhead fault-tolerant FPGA systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26(2), 1998.

[13] A. J. Yu, G. G. Lemieux. Defect-tolerant FPGA switch block and connection block with fine-grain redundancy for yield enhancement. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2005.

[14] A. J. Yu, G. G. Lemieux. FPGA defect tolerance: Impact of granularity. *Proceedings of the International Conference on Field-Programmable Technology*, 2005.

[15] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[16] J. E. Hopcroft, R. M. Karp. An $n^{2.5}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing* 2(4), 1973.

[17] H. Naeimi, A. DeHon. A greedy algorithm for tolerating defective crosspoints in nanoPLA design. *Proceedings of the International Conference on Field-Programmable Technology*, IEEE, 2004.

[18] A. DeHon, H. Naeimi. Seven strategies for tolerating highly defective fabrication. *IEEE Design and Test of Computers* 22(4), 2005.

[19] Z. Hyder, J. Wawrzynek. Defect tolerance in multiple-FPGA systems. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2005.

[20] M. B. Tahoori. Application-dependent testing of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14(9), 2006.

[21] J. von Neumann. Probabilistic logic and the synthesis of reliable organisms from unreliable components. *Automata Studies* C. Shannon, J. McCarthy (ed.), Princeton University Press, 1956.

[22] N. Pippenger. Developments in "the synthesis of reliable organisms from unreliable components." *Proceedings of the Symposia of Pure Mathematics* 50, 1990.

[23] C. Carmichael. *Triple Module Redundancy Design Techniques for Virtex FPGAs*. San Jose, 2006 (XAPP 197—*http://www.xilinx.com/bvdocs/appnotes/xapp197.pdf*).

[24] N. Rollins, M. Wirthlin, P. Graham, M. Caffrey. Evaluating TMR techniques in the presence of single event upsets. *Proceedings of the International Conference on Military and Aerospace Programmable*, 2003.

[25] G. C. Clark Jr., J. B. Cain. *Error-Correction Coding for Digital Communications*, Plenum Press, 1981.

[26] R. J. McEliece. *The Theory of Information and Coding*, Cambridge University Press, 2002.

[27] S. Mitra, E. J. McCluskey. Which concurrent error detection scheme to choose? *Proceedings of the International Test Conference*, 2000.

[28] J. Postel (ed.). Transmission Control Protocol—DARPA Internet Program Protocol Specification, RFC 793, Information Sciences Institute, University of Southern California, Marina del Rey, 1981.

[29] E. Takeda, N. Suzuki, T. Hagiwara. Device performance degradation to hot-carrier injection at energies below the Si-SiO2 energy barrier. *Proceedings of the International Electron Devices Meeting*, 1983.

[30] S.-H. Renn, C. Raynaud, J.-L. Pelloie, F. Balestra. A thorough investigation of the degradation induced by hot-carrier injection in deep submicron N- and P-channel partially and fully depleted unibond and SIMOX MOSFETs. *IEEE Transactions on Electron Devices* 45(10), 1998.

[31] D. K. Schroder, J. A. Babcock. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing, *Journal of Applied Physics* 94(1), 2003.

[32] J. Osborn, R. Lacoe, D. Mayer, G. Yabiku. Total dose hardness of three commercial CMOS microelectronics foundries. *Proceedings of the European Conference on Radiation and Its Effects on Components and Systems*, 1997.

[33] C. Brothers, R. Pugh, P. Duggan, J. Chavez, D. Schepis, D. Yee, S. Wu. Total-dose and SEU characterization of 0.25 micron CMOS/SOI integrated circuit memory technologies. *IEEE Transactions on Nuclear Science* 44(6) 1997.

[34] J. Emmert, C. Stroud, B. Skaggs, M. Abramovici. Dynamic fault tolerance in FPGAs via partial reconfiguration. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[35] S. K. Sinha, P. M. Kamarchik, S. C. Goldstein. Tunable fault tolerance for runtime reconfigurable architectures. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[36] V. Lakamraju, R. Tessier. Tolerating operational faults in cluster-based FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2000.

[37] S. Harel, J. Maiz, M. Alavi, K. Mistry, S. Walsta, C. Dai Impact of CMOS process scaling and SOI on the soft error rates of logic processes. *Proceedings of Symposium on VLSI Digest of Technology Papers*, 2001.

[38] H. Quinn, P. Graham. Terrestrial-based radiation upsets: A cautionary tale. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[39] C. Carmichael, M. Caffrey, A. Salazar. *Correcting Single-Event Upsets Through Virtex Partial Configuration*. Xilinx, Inc., San Jose, 2000 (XAPP 216—*http://www.xilinx.com/bvdocs/appnotes/xapp216.pdf*).

[40] G.-H. Asadi, M. B. Tahoori. Soft error mitigation for SRAM-based FPGAs. *Proceedings of the VLSI Test Symposium*, 2005.

[41] B. Pratt, M. Caffrey, P. Graham, K. Morgan, M. Wirthlin. Improving FPGA design robustness with partial TMR. *Proceedings of the IEEE International Reliability Physics Symposium*, 2006.

CHAPTER 38

RECONFIGURABLE COMPUTING AND NANOSCALE ARCHITECTURE

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

For roughly four decades integrated circuits have been patterned top down with optical lithography, and feature sizes, *F*, have shrunk in a predictable, geometric fashion. With feature sizes now far below optical wavelengths (c.f. 400 nm violet light and 65 nm feature sizes) and approaching atomic lattice spacings (c.f. 65 nm feature sizes and 0.5 nm silicon lattice), it becomes more difficult and more expensive to pattern arbitrary features.

At the same time, fundamental advances in synthetic chemistry allow the assembly of structures made of a small and precise number of atoms, providing an alternate, bottom-up approach to constructing nanometer-scale devices. Rather than relying on ever-finer precision and control of lithography, bottom-up techniques exploit physical phenomena (e.g., molecular dimensions, film thicknesses composed of a precise number of atomic layers, nanoparticles constructed by self-limiting chemical processes) to directly define key feature sizes at the nanometer scale. Bottom-up fabrication gives us access to smaller feature sizes and promises more economical construction of atomic-scale devices and wires.

Both bottom-up structure synthesis and extreme subwavelength top-down lithography can produce small feature sizes only for very regular topologies. In optical lithography, regular interference patterns can produce regular structures with finer resolution than arbitrary topologies [1]. Bottom-up syntheses are limited to regular structures amenable to physical self-assembly.

Further, as noted in Chapter 37, construction at this scale, whether by top-down or bottom-up fabrication, exhibit high defect rates. High defect rates also drive increasing demand for regularity to support resource substitution.

At the same time, new technologies offer configurable switchpoints that can fit in the space of a nanoscale wire crossing (Section 38.2.3). The switches are much smaller than current SRAM configurable switches and can reduce the cost of reconfigurable architectures relative to ASICs. Smaller configurable switchpoints are particularly fortuitous because they make fine-grained configurability for defect tolerance viable.

High demand for regularity and fine-grained defect tolerance coupled with less expensive configurations increase the importance of reconfigurable architectures. Reconfigurable architectures can accommodate the requirements of these atomic-scale technologies and exploit the density benefits they offer. Nonetheless, to fully accommodate and exploit these cost shifts, reconfigurable architectures continue to evolve.

This chapter reviews proposals for nanoscale configurable architectures that address the demands and opportunities of atomic-scale, bottom-up fabrication. It focuses on the nanoPLA architecture (see Section 38.6 and DeHon [2]), which has been specifically designed to exploit nanowires (Section 38.2.1) as the key building block. Despite the concrete focus on nanowires, many of the design solutions employed by the nanoPLA are applicable to other atomic-scale technologies. The chapter also briefly reviews nanoscale architectures (Section 38.7), which offer alternative solutions to key challenges in atomic-scale design.

38.1 TRENDS IN LITHOGRAPHIC SCALING

In the conventional, top-down lithographic model, we define a minimum, lithographically imageable feature size (i.e., half pitch, *F*) and build devices that are multiples of this imageable feature size. Within the limits of this feature size, VLSI layout can perfectly specify the size of features and their location relative to each other in three dimensions—both in the two-dimensional plane of each lithographic layer and with adequate registration between layers. This gives complete flexibility in the layout of circuit structures as long as we adhere to the minimum imageable and repeatable feature size rules.

Two simplifying assumptions effectively made this possible: (1) Feature size was large compared to atoms, and (2) feature size was large compared to the wavelength of light used for imaging. With micron feature sizes, features were thousands of atoms wide and multiple optical wavelengths. As long as the two assumptions held, we did not need to worry about the discreteness of atoms nor the limits of optical lithography.

Today, however, we have long since passed the point where optical wavelengths are large compared to feature sizes, and we are rapidly approaching the point where feature sizes are measured in single-digit atom widths. We have made the transition to optical lithography below visible light (e.g., 193 nm wavelengths) and subwavelength imaging. Phase shift masking exploits interference of multiple light sources with different phases in order to define feature sizes finer than the wavelength of the source. This has allowed continued feature size scaling but increases the complexity and, hence, the cost of lithographic imaging.

Topology in the regions surrounding a pattern now impacts the fidelity of reproduction of the circuit or interconnect, creating the demand for optical proximity correction. As a result, we see an increase both in the complexity of lithographic mask generation and in the number of masks required. Region-based topology effects also limit the structures we can build. Because of both limitations in patterning and limitations in the analysis of region-based patterning effects, even in "full-custom" designs, we are driven to compose functions from a small palette of regular structures.

Rock's Law is a well-known rule of thumb in the semiconductor industry that suggests that semiconductor processing equipment costs increase geometrically as feature sizes shrink geometrically. One version of Rock's Law estimates that the cost of a semiconductor fabrication plant doubles every four years. Fabrication plants for the 90 nm generation were reported to cost $2 to 3 billion.

The increasing cost comes from several sources, including the following:

- *Increasing demand for accuracy:* Alignment of features must scale with feature sizes.
- *Increasing demand for purity:* Smaller features mean that even smaller foreign particles (e.g., dust and debris) must be eliminated to prevent defects.
- *Increasing demand for device yield:* As noted in Chapter 37 (see Perfect yield, Section 37.2.3), to keep component yield constant, the per-device defect rate, P_d, must decrease as more devices are integrated onto each component.
- *Increasing processing steps:* More metal layers plus increasingly complex masks for optical resolution enhancement (described before) demand more equipment and processing.

It is already the case that few manufacturers can afford the capital investment required to develop and deploy the most advanced fabrication plants. Rising fabrication costs continue to raise the bar, forcing consolidation and centralization in integrated circuit manufacturing.

Starting at around 90 nm feature sizes, the mask cost per component typically exceeds $1 million. This rising cost comes from the effects previously noted: more masks per component and greater complexity per mask. Coupled with rising component design and verification complexity, this raises the nonrecurring engineering (NRE) costs per chip design.

The economics of rising NRE ultimately lead to fewer unique designs. That is, if we hope to keep NRE costs to a small fraction—for example 10 percent—of the potential revenue for a chip, the market must be at least 10 times the NRE cost. With total NRE costs typically requiring tens of millions of dollars for 90 nm designs, each chip needs a revenue potential in the hundreds of millions of dollars to be viable. The bar continues to rise with NRE costs, decreasing the number of unique designs that the industry can support. This decrease in unique designs creates an increasing demand for differentiation after fabrication (i.e., reconfigurability).

38.2 BOTTOM-UP TECHNOLOGY

In contrast, bottom-up synthesis techniques give us a way to build devices and wires without relying on masks and lithography to define their atomic-scale features. They potentially provide an alternative path to device construction that may provide access to these atomic-scale features more economically than traditional lithography.

This section briefly reviews the bottom-up technology building blocks exploited by the nanoPLA, including nanowires (Section 38.2.1), ordered assembly of nanowires (Section 38.2.2), and programmable crosspoints (Section 38.2.3). These technologies are sufficient for constructing and understanding the basic nanoPLA design. For a roundup of additional nanoscale wire and crosspoint technologies, see the appendix in DeHon's 2005 article [2].

38.2.1 Nanowires

Chemists and material scientists are now regularly producing semiconducting and metallic wires that are nanometers in diameter and microns long using bottom-up synthesis techniques. To bootstrap the process and define the smallest dimensions, self-limiting chemical processes (e.g., Tan et al. [3]) can be used to produce nanoparticles of controlled diameter. From these nanoparticle seed catalysts, we can grow nanowires with diameters down to 3 nm [4]. The nanowire self-assembles into a crystalline lattice similar to planar silicon; however, growth is only enabled in the vicinity of the nanoparticle's catalyst. As a result, catalyst size defines the diameter of the grown nanowires [5]. Nanowires can be grown to millimeters in length [6], although it is more typical to work with nanowires tens of microns long [7].

Bottom-up synthesis techniques also allow the definition of atomic-scale features within a single nanowire. Using timed growth, features such as composition of different materials and different doping levels can be grown along the axis of the nanowire [8–10]. This effectively allows the placement of device features into nanowires, such as a field effect gateable region in the middle of an otherwise ungateable wire (see Figure 38.1). Further, radial shells of different materials can be grown around nanowires with controlled thickness using timed growth [11, 12] or atomic-layer deposition [13, 14] (see Figure 38.2). These shells can be used to force the spacing between device and wire features, to act as dielectrics for field effect gating, or to build devices integrating heterogeneous materials with atomic-scale dimensions.

After a nanowire has been grown, it can be converted into a metal–silicon compound with lower resistance. For example, by coating select regions of

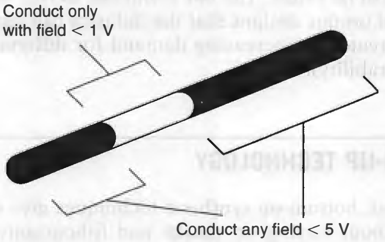


FIGURE 38.1 ■ An axial doping profile. By varying doping along the axis of the nanowire, selectively gateable regions can be integrated into the nanowire.

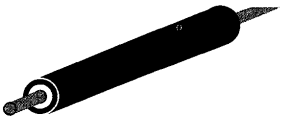


FIGURE 38.2 ■ A radial doping profile.

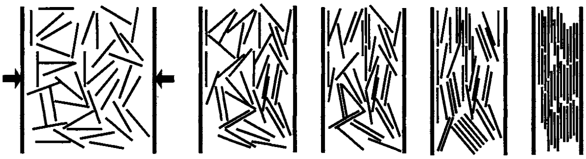


FIGURE 38.3 ■ The Langmuir–Blodgett alignment of nanowires.

the nanowire with nickle and annealing, we can form a nickle–silicide (NiSi) nanowire [15]. The NiSi resistivity is much lower than the resistivity of heavily doped bulk silicon. Since nanowires have a very small cross-sectional area, this conversion is very important to keep the resistance, and hence the delay, of nanowires low. Further, this conversion is particularly important in reducing contact resistance between nanowires and lithographic-scale power supplies.

38.2.2 Nanowire Assembly

Langmuir–Blodgett (LB) flow techniques can be used to align a set of nanowires into a single orientation, close-pack them, and transfer them onto a surface [16, 17] (see Figure 38.3). The resulting wires are all parallel, but their ends may not be aligned. By using wires with an oxide sheath around the conducting core, the wires can be packed tightly without shorting together. The oxide sheath defines the spacing between conductors and can, optionally, be etched away after assembly. The LB step can be rotated and repeated so that we get multiple layers of nanowires [16, 18], such as crossed nanowires for building a wired-OR plane (Section 38.4.1).

38.2.3 Crosspoints

Many technologies have been demonstrated for nonvolatile, switched crosspoints. Common features include the following:

- Resistance that changes significantly between on and off states
- Ability to be made rectifying (i.e., to act as diodes)
- Ability to turn the device on or off by applying a voltage differential across the junction
- Ability to be placed within the area of a crossed nanowire junction

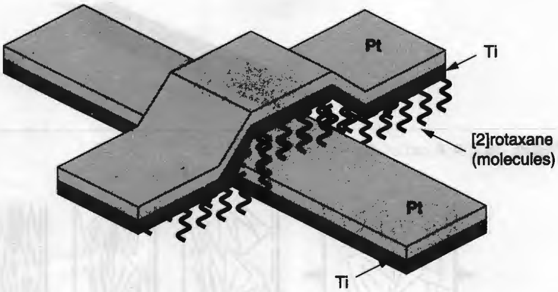


FIGURE 38.4 ■ Switchable molecules sandwiched between nanoscale wires.

Chen et al. [19, 20] demonstrate a nanoscale Ti/Pt-[2]rotaxane-Ti/Pt sandwich (see Figure 38.4), which exhibits hysteresis and nonvolatile state storage showing an order of magnitude resistance difference between on and off states. The state of these devices can be switched at ±2 V and read at ±0.2 V. The basic hysteretic molecular memory effect is not unique to the [2]rotaxane, and the junction resistance is continuously tunable [21]. The exact nature of the physical phenomena involved is the subject of active investigation. LB techniques also can be used to place the switchable molecules between crossed nanowires (e.g., Collier et al. [22], Brown et al. [23]).

In conventional VLSI, the area of an SRAM-based programmable crosspoint switch is much larger than the area of a wire crossing. A typical CMOS switch might be $600F^2$ [24], compared to a $3F \times 3F$ bottom-level metal wire crossing, making the crosspoint more than 60 times the area of the wire crossing. Consequently, the nanoscale crosspoints offer an additional device size reduction beyond that implied by the smaller nanowire feature sizes. This particular device size benefit reduces the overhead for configurability associated with programmable architectures (e.g., FPGAs, PLAs) in this technology, compared to conventional CMOS.

38.3 CHALLENGES

Although the techniques reviewed in the previous section provide the ability to create very small feature sizes using the basic physical properties of materials to define dimensions, they also bring with them a number of challenges that any nanoscale architecture must address, including the following:

- *Required regularity in assembly and architecture:* These techniques do not allow the construction of arbitrary topologies; the assembly techniques limit us to regular arrays and crossbars of nanowires.

- *Lack of correlation in features:* The correlation between features is limited. It is possible to have correlated features within a nanowire, but only in a single nanowire; we cannot control which nanowire is placed next to which other nanowire or how they are aligned.
- *Differentiation:* If all the nanowires in a regular crossbar assembly behaved identically (e.g., were gated by the same inputs or were diode-connected to the same inputs), we would not get a benefit out of the nanoscale pitch. It is necessary to differentiate the function performed by the individual nanowires in order to exploit the benefits of their nanoscale pitch.
- *Signal restoration:* The diode crosspoints described in the previous section are typically nonrestoring; consequently, it is necessary to provide signal restoration for diode logic stages.
- *Defect tolerance:* We expect a high rate of defects in nanowires and crosspoints. Nanowires may break or make poor contacts. Crosspoints may have poor contact to the nanowires or contain too few molecules to be switched into a low-resistance state.

38.4 NANOWIRE CIRCUITS

It is possible to build a number of key circuits from the nanoscale building blocks introduced in the previous section, including a diode-based wired-OR logic array (Section 38.4.1) and a restoring nanoscale inverter (Section 38.4.2).

38.4.1 Wired-OR Diode Logic Array

The primary configurable structure we can build is a set of tight-pitched, crossed nanowires. With a programmable diode crosspoint at each nanowire intersection, this crossed nanowire array can serve as a programmable OR-plane. Assuming the diodes point from columns to rows (see Figure 38.5), each row output nanowire serves as a wired-OR for all of the inputs programmed into the low-resistance state. In the figure, programmed on crosspoints are shown in black; off crosspoints are shown in gray. Bold lines represent a nanowire pulled high, while gray lines remain low. Output nanowires are shown bold starting at the diode that pulls them high to illustrate current flow; the entire output nanowire would be pulled high in actual operation. Separate circuitry, not shown, is responsible for pulling wires low or precharging them low so that an output remains low when no inputs can pull it high.

Consider a single-row nanowire, and assume for the moment that there is a way to pull a nondriven nanowire down to ground. If any of the column nanowires that cross this row nanowire are connected with low-resistance crosspoint junctions and are driven to a high voltage level, the current into the column nanowire will be able to flow into the row nanowire and charge it up to a higher voltage value (see O1, O3, O4, and O5 in Figure 38.5). However, if none of

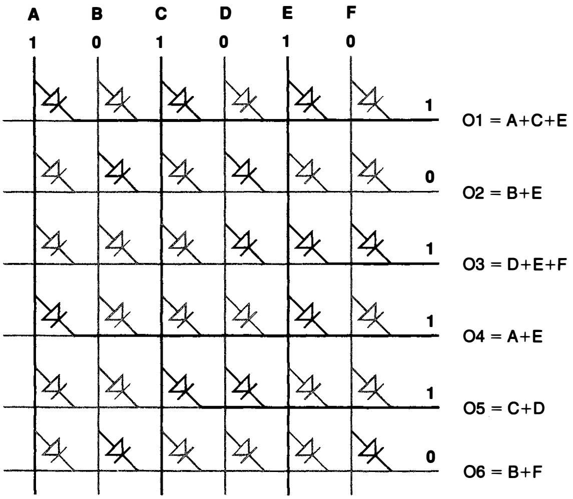


FIGURE 38.5 ■ The wired-OR plane operation.

the connected column nanowires is high, the row nanowire will remain low (see O2 and O6 in the figure). Consequently, the row nanowire effectively computes the OR of its programmed inputs.

The output nanowires do pull their current directly off the inputs and may not be driven as high as the input voltage. Consequently, these outputs will require restoration (Section 38.4.2).

A special use of the wired-OR programmable array is for interconnect. That is, if we restrict ourselves to connecting a *single* row wire to each column wire, the crosspoint array can serve as a crossbar switch. This allows any input (column) to be routed to any output (row) (see Figure 38.6). This structure is useful for postfabrication programmable routing to connect logic functions and to avoid defective resources. In the figure, programmed on crosspoints are shown in black; off crosspoints are shown in gray. This means that the crossbar shown in the figure is programmed to connect A→T, B→Q, C→V, D→S, E→U, and F→R.

38.4.2 Restoration

As noted in Section 38.4.1, the programmable, wired-OR logic is passive and nonrestoring, drawing current from the input. Further, OR logic is not universal. To build a good, composable logic family, we need to be able to isolate inputs from output loads, restore signal strength and current drive, and invert signals.

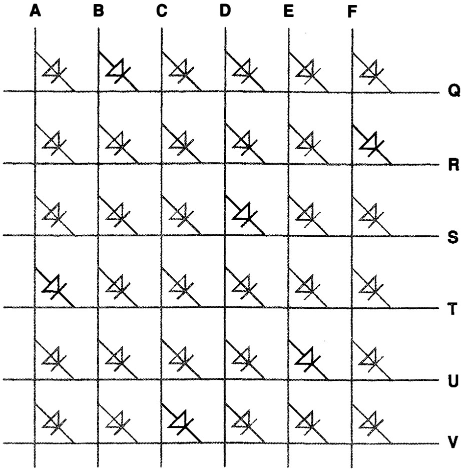


FIGURE 38.6 ■ An example crossbar routing configuration.

Fortunately, nanowires can be field effect controlled. This provides the potential to build gates that behave like field effect transistors (FETs) for restoration. However, to realize them, we must find ways to create the appropriate gate topology within regular assembly constraints (Section 38.5).

If two nanowires are separated by an insulator, perhaps using an oxide core shell, we can use the field from one nanowire to control the other nanowire. Figure 38.7 shows an inverter built using this basic idea. The horizontal nanowire serves as the input and the vertical nanowire as the output. This gives a voltage transfer equation of

$$V_{out} = V_{high} \left(\frac{R_{pd}}{R_{pd} + R_{fet}(\text{Input}) + R_{pu}} \right) \tag{38.1}$$

For the sake of illustration, the vertical nanowire has a lightly doped P-type depletion-mode region at the input crossing that forms a FET controlled by the input voltage (R_{fet}(Input)). Consequently, a low voltage on the input nanowire allows conduction through the vertical nanowire ($R_{fet} = R_{\text{on-fet}}$ is small), and a high input depletes the carriers from the vertical nanowire and prevents conduction ($R_{fet} = R_{\text{off-fet}}$ is large). As a result, a low input allows the nanowire to conduct and pull the output region of the vertical nanowire up to a high voltage. A high input prevents conduction and the output region remains low. A second crossed region on the nanowire is used for the pulldown (R_{pd}). This region can be used as a gate for predischarging the output so that the inverter is pulled low

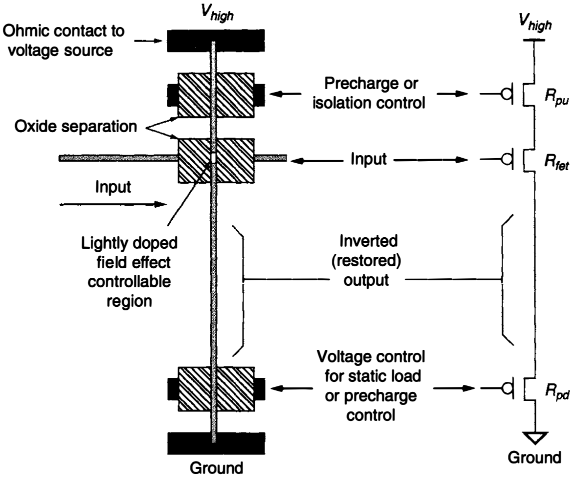

V_{high}
Ohmic contact to voltage source
Oxide separation
Input
Lightly doped field effect controllable region
Ground
Precharge or isolation control
Input
Inverted (restored) output
Voltage control for static load or precharge control
V_{high}
R_{pu}
R_{fet}
R_{pd}
Ground

FIGURE 38.7 ■ A nanowire inverter.

before the input is applied, then left high to disconnect the pulldown voltage during evaluation. Alternatively, it can be used as a static load for PMOS-like ratioed logic. By swapping the location of the high- and low-power supplies, this same arrangement can be used to buffer rather than invert the input.

Note that the gate only loads the input capacitively. Consequently, the output current is isolated from the input current at this inverter or buffer. Further, nanowire field effect gating has sufficient nonlinearity so that this gate provides gain to restore logic signal levels [25].

38.5 STATISTICAL ASSEMBLY

One challenge posed by regular structures, such as tight-pitch nanowire crossbars, is differentiation. If all the wires are the same and are fabricated at a pitch smaller than we can build arbitrary topologies lithographically, how can we selectively address a single nanowire? If we had enough control to produce arbitrary patterns at the nanometer scale, we could build a decoder (see Figure 38.8) to provide pitch-matching between this scale and the scale at which we could define arbitrary topologies.

The trick is to build the decoder statistically. That is, differentiate the nanowires by giving each one an address, randomly select the nanowires that go into each array, and carefully engineer the statistics to guarantee a high

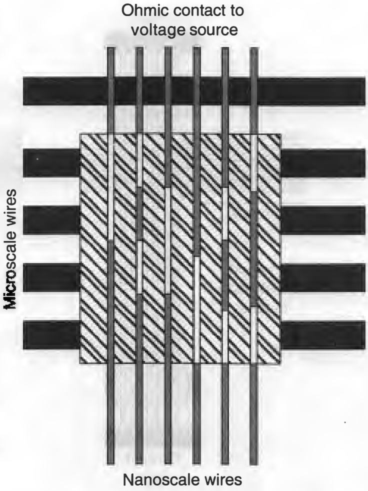


FIGURE 38.8 ■ A decoder for addressing individual nanowires assembled at nanoscale pitch.

probability that there will be a unique address associated with each nanowire in each nanowire array. We can use axial doping to integrate the address into each nanowire [26].

If we pick the address space sparsely enough, Law of Large Numbers statistics can guarantee unique addressability of the nanowires. For example, if we select 10 nanowires out of a large pool with 10^6 different nanowire types, we get a unique set of nanowires more than 99.99 percent of the time. In general, we can guarantee more than 99 percent probability of uniqueness of N nanowires using only $100N^2$ addresses [26]. By allowing a few duplications, the address space can be much smaller [27].

Statistical selection of coded nanowires can also be used to assemble nanoscale wires for restoration [2]. As shown in Figure 38.9(a), if coded nanowires can be perfectly placed in an array, we can build the restoration circuit shown in Section 38.4.2 (Figure 38.7) and arrange them to restore the outputs of a wired-OR array. However, the bottom-up techniques that can assemble these tight-pitch feature sizes cannot order or place individual nanowires and cannot provide correlation between nanowires. As shown in Figure 38.9(b), statistical alignment and placement of the restoration nanowires can be used to construct the restoration array. Here, not every input will be restored, but the Law of Large Numbers guarantees that we can restore a reliably predictable fraction of the inputs. For further details, see DeHon [2, 27].

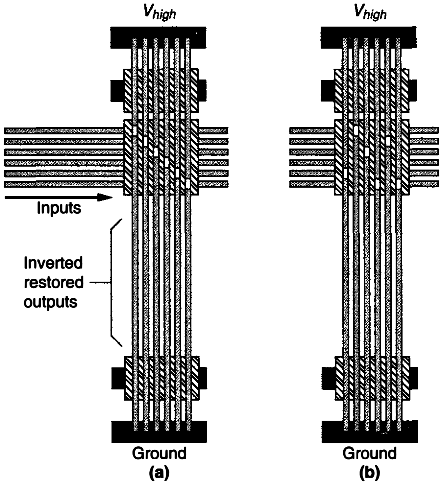


FIGURE 38.9 ■ A restoration array: (a) ideal and (b) stochastic.

38.6 NANOPLA ARCHITECTURE

With these building blocks we can assemble a complete reconfigurable architecture. This section starts by describing the PLA-based logic block (Section 38.6.1), then shows how PLAs are connected together into an array of interconnected logic blocks (Section 38.6.2). It also notes that nanoscale memories can be integrated with this array (Section 38.6.3), reviews the defect tolerance approach for this architecture (Section 38.6.4), describes how designs are mapped to nanoPLA designs (Section 38.6.5), and highlights the density benefits offered by the technology (Section 38.6.6).

38.6.1 Basic Logic Block

The nanoPLA architecture combines the wired-OR plane, the stochastically assembled restoration array, and the stochastic address decoder to build a simple, regular PLA array (see Figure 38.10). The stochastic decoder described in Section 38.5 allows individual nanowires to be addressed from the lithographic scale for testing and programming (see Figures 38.11 and 38.12). The output of the programmable, wired-OR plane is restored via a restoration plane using field effect gating of the crossed nanowire set as described in Section 38.5 and shown in Figure 38.9.

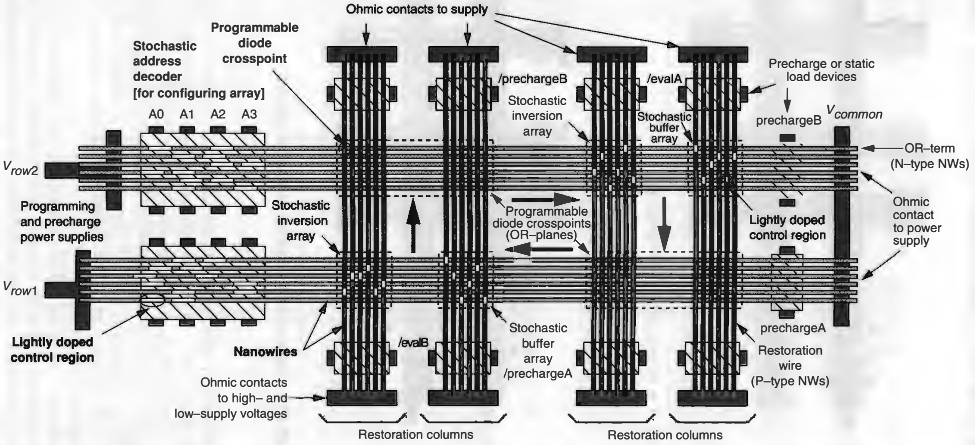

Ohmic contacts to supply
Programmable diode crosspoint
Stochastic address decoder [for configuring array]
A0
A1
A2
A3
/prechargeB
/evalA
Precharge or static load devices
Stochastic inversion array
Stochastic buffer array
prechargeB
V_{common}
V_{row2}
OR–term (N–type NWs)
Ohmic contact to power supply
Programming and precharge power supplies
Stochastic inversion array
Programmable diode crosspoints (OR–planes)
Lightly doped control region
V_{row1}
Lightly doped control region
Nanowires
/evalB
Stochastic buffer array /prechargeA
prechargeA
Restoration wire (P–type NWs)
Ohmic contacts to high– and low–supply voltages
Restoration columns
Restoration columns

FIGURE 38.10 ■ A simple nanoPLA block.

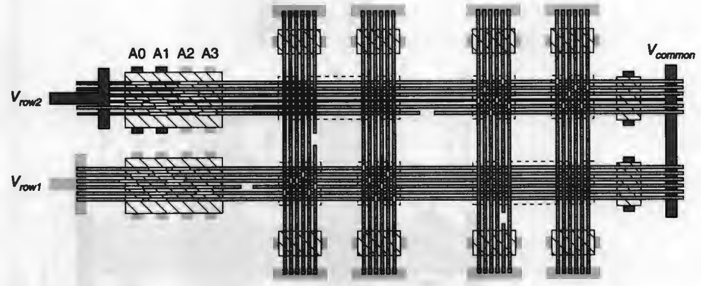


FIGURE 38.11 ■ Addressing a single nanowire.

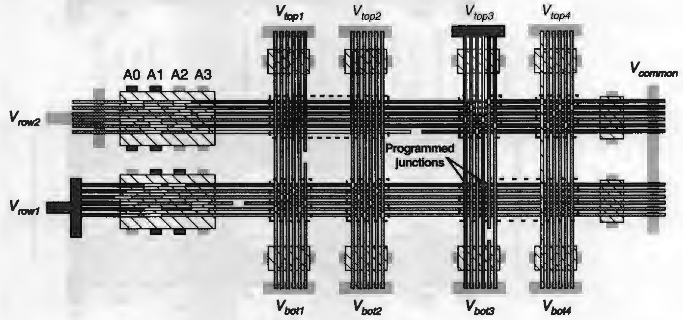


FIGURE 38.12 ■ Programming a nanowire–nanowire crosspoint.

As shown in Figure 38.11, an address is applied on the lithographic-scale address lines (A0 ... A3). The applied address (1100) allows conduction through only a single nanowire. By monitoring the voltage at the common lithographic node at the far end of the nanowire (V_{common}), it is possible to determine whether the address is present and whether the wire is functional (e.g., not broken). By monitoring the timing of the signal on V_{common}, we may be able to determine the resistance of the nanowire.

As shown in Figure 38.12, addresses are applied to the lithographic-scale address lines of both the top and bottom planes to select individual nanowires in each plane. We use the stochastic restoration columns to turn the corner between the top plane and the restoration inputs to the bottom plane. Note that since column 3 is an inverting column, we arrange for the single, selected signal on the top plane to be a low value. Since the stochastic assembly resulted in two restoration wires for this input, both nanowire inputs are activated. As a result, we place the designated voltage across the two marked crosspoints to turn on the crosspoint junctions between the restored inputs and the selected nanowire in the bottom plane.

The restoration planes can provide inversion such that the pair of planes serve as a programmable NOR. The two back-to-back NOR planes can be viewed as a traditional AND–OR PLA with suitable application of DeMorgan's Law. A second set of restoration wires provides buffered, noninverted inputs to the next wired-OR plane; in this manner, each plane gets the true and complement version of each logical signal just as is normally provided at the inputs to a VLSI PLA. Microscale field effect gates (e.g., /evalA and /evalB) control when nanowire logic can evaluate, allowing the use of a familiar 2-phase clocking discipline. As such, the PLA cycle shown in Figure 38.10 can directly implement an FSM. Programmable crosspoints can be used to personalize the array, avoid defective wires and crosspoints (Section 38.6.4), and implement a deterministic function despite fabrication defects and stochastic assembly.

38.6.2 Interconnect Architecture

To construct larger components using the previously described structures, we can build an array of nanoPLA blocks, where each block drives outputs that cross the input (wired) regions of many other blocks (Figure 38.13) [2, 28]. This allows the construction of modest-size PLAs (e.g., 100 Pterms), which are efficient for logic mapping and keep the nanowire runs short (e.g., 10 μm) in order to increase yield and avoid the high resistance of long nanowires. The nanoPLA blocks provide logic units, signal switching, and signal buffering for long wire runs. With an appropriate overlap topology, such nanoPLAs can support Manhattan (orthogonal X–Y) routing similar to conventional, island-style FPGA architectures (Chapter 1).

By stacking additional layers of nanowires, the structure can be extended vertically into the third dimension [29]. Programmable and gateable junctions between adjacent nanowire layers allow routing up and down the nanowire stack. This provides a path to continue scaling logic density when nanowire diameters can shrink no further.

The resulting nanoPLA structure is simple and very regular. Its high-density features are built entirely from tight-pitched nanowire arrays. All the nanowire array features are defined using bottom-up techniques. The overlap topology between nanowires is carefully arranged so that the output of a function (e.g., wired-OR, restoration, routing) is a segment of a nanowire that then crosses the active or input portion of another function. Regions (e.g., wired-OR, restoration) are differentiated at a lithographic scale. Small-scale differentiation features are built into the nanowires and statistically populated (e.g., addressing, restoration).

In the nanoPLA, the wired-OR planes combine the roles of switchbox, connection box, and logic block into one unified logic and switching plane. The wired-OR plane naturally provides the logic block in a nanoPLA block. It also serves

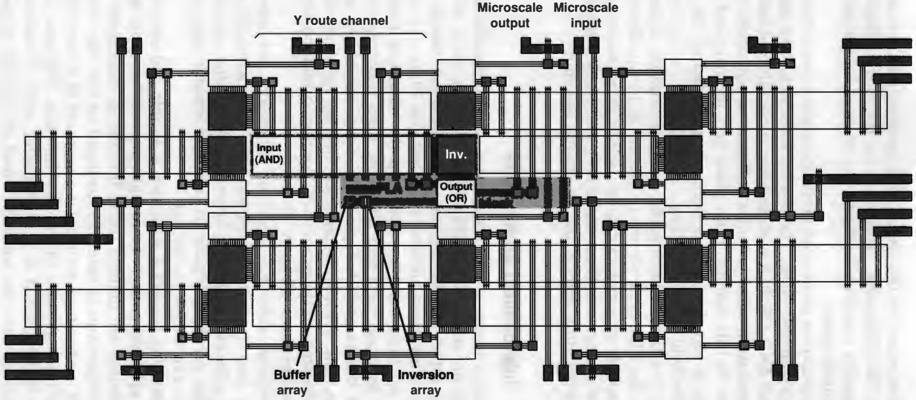


FIGURE 38.13 ■ nanoPLA block tiling with edge I/O to lithographic scale.

to select inputs from the routing channel that participate in the logic. Signals that must be rebuffered or switched through a block are also routed through the same wired-OR plane. Since the configurable switchpoints fit within the space of a nanowire crossing, the wired-OR plane (hence the interconnect switching) can be fully populated unlike traditional FPGA switch blocks that have a very limited population to reduce their area requirements.

38.6.3 Memories

The same basic crosspoints and nanowire crossbar used for the wired-OR plane (Section 38.4.1) can also serve as the core of a memory bank. An address decoder similar to the one used for programming the wired-OR array (see Section 38.5 and Figure 38.8) supports read/write operations on the memory core [26, 30]. Unique, random addresses can be used to configure deterministic memory addresses, avoiding defective memory rows and columns [31]. A full-component architecture would interleave these memory blocks with the nanoPLA logic blocks similar to the way memory blocks are embedded in conventional FPGAs (Chapter 1).

38.6.4 Defect Tolerance

Nanowires in each wired-OR plane and interconnect channel are locally substitutable (see the Local sparing subsection in Section 37.2.4). The full population of the wired-OR crossbar planes guarantees this is true even for the interconnect channels. We provision spare nanowires based on their defect rate, as suggested in the Yield with sparing subsection of Section 37.2.3. For each array, we test for functional wires as illustrated in Section 38.6.1. Logical Pterms are assigned to nanowires using the matching approach described in the Fine-grained Pterm matching subsection of Section 37.2.5. For a detailed description of nanoPLA defect tolerance, see DeHon and Naeimi [32].

38.6.5 Design Mapping

Logic-level designs can be mapped to the nanoPLA. The logic and physical mapping for the nanoPLA uses similar techniques to those introduced in Part III. Starting from a logic netlist, technology mapping can be performed using PLAmap (see Section 13.3.4) to generate two-level clusters for each nanoPLA block, which can then be placed using an annealing-based placer (Chapter 14). Routing is performed with a PathFinder-based router (Chapter 17). Because of the full population of the switchboxes, the nanoPLA router need only perform global routing. Since nanoPLA blocks provide both logic and routing, the router must also account for the logic assigned to each nanoPLA block when determining congestion. As noted before, at design loadtime, logical Pterms are assigned to specific nanowires using a greedy matching approach (see the Fine-grained Pterm matching subsection of Section 37.2.5).

38.6.6 Density Benefits

Despite statistical assembly, lithographic overheads for nanowire addressing, and high defect rates, small feature sizes, and compact crosspoints can offer a significant density advantage compared to lithographic FPGAs. When mapping the Toronto 20 benchmark suite [33] to 10-nm full-pitch nanowires (e.g., 5-nm-diameter nanowires with 5-nm spacing between nanowires), we typically see two orders of magnitude greater density than with defect-free 22-nm lithographic FPGAs [2]. As noted earlier, areal density can be further increased by using additional layers of nanowires [29].

38.7 NANOSCALE DESIGN ALTERNATIVES

Several architectures have been proposed for nanoscale logic. A large number are also based on regular crossbar arrays and look similar to the nanoPLA at a gross level (see Table 38.1). Like the nanoPLA, all these schemes employ fine-grained configurability to tolerate defects. Within these architectures there are different ways to address the key challenges (Section 38.3). These architectures enrich the palette of available component solutions, increasing the likelihood of assembling a complementary set of technology and design elements to practically realize nanoscale configurable logic.

38.7.1 Imprint Lithography

In the concrete technology described in Section 38.2, seeded nanowire growth was used to obtain small feature sizes and LB flow to assemble them into parallel arrays. Another emerging technique for producing regular, nanoscale structures (e.g., a set of parallel, tight-pitched wires) is imprint lithography. The masks for imprint lithography can be generated using bottom-up techniques.

TABLE 38.1 ■ A comparison of nano-electronic programmable logic designs

Component element	HP/UCLA crossbar architecture	CMU nanoFabric	nanoPLA	Stony Brook CMOL	Hewlett-Packard FPNI
Crosspoint technology	Programmable diode	Programmable diode	Programmable diode	Programmable diode	Programmable diode
Nanowire technology	Nano-imprint lithography	Nanopore templates	Catalyst nanowires	Nano-imprint lithography	Nano-imprint lithography
Logic implementation	Nanoscale wired-OR	Nanoscale wired-OR	Nanoscale wired-OR	Nanoscale wired-OR	Lithoscale (N)AND2
CMOS↔Nanowire interface	Random particles	–	Coded nanowires	Crossbar tilt	Crossbar tilt
Restoration	CMOS	RTD latch	nanowire FET	CMOS	CMOS
References	[34, 35, 36]	[37, 38]	[28, 39]	[40]	[41]

In one scheme, timed vertical growth or atomic-layer deposition on planar semiconductors is used to define nanometer-scale layers of differentially etchable materials. Cut orthogonally, the vertical cross-section can be etched to produce a comblike structure where the teeth, as well as the spacing between them, are single-digit nanometers wide (e.g., 8 nm). The resulting structure can serve as a pattern for nanoscale imprint lithography [42,43] to produce a set of tight-pitched, parallel lines. That is, the long parallel lines resulting from the differential etch can be stamped into a resist mask [43], which is then etched to produce a pattern in a polymer or coated with metal to directly transfer metallic lines to a substrate [42]. These techniques can produce regular nanostructures but cannot produce arbitrary topologies.

38.7.2 Interfacing

When nanowires are fabricated together using imprint lithography, it is not possible to uniquely construct and code nanowires as exploited for addressing in the nanoPLA (Section 38.5). Williams and Kuekes [36] propose the first randomized decoder scheme for differentiating nanoscale wires and interfacing between lithographic and nanoscale feature sizes. They use a physical process to randomly deposit metal particles between the lithographic-scale address lines and the nanoscale wires. A nanowire is controllable by an address wire only if it has a metal particle bridging it to the address line. Unlike the nanowire-coding scheme where addresses are selected from a carefully chosen address space and grown into each nanowire (Section 38.5), in this scheme the address on each nanowire is randomly generated. As a result, this scheme requires 2 to 2.5 times as many address wires as the statistically assembled nanowire-coding scheme.

Alternately, Strukov and Likharev [40, 44] observe that it should be possible to directly connect each long crossbar nanowire by a nanovia to lithographic-scale circuitry that exists below the nanoscale circuits. The *nanovia* is a semiconductor pin spaced at lithographic distances and grown with a taper to a nanoscale tip for interfacing with individual nanowires. An array of these pins (e.g., Jensen [45]) can provide nanovia interfaces.

The key idea is to pitch-match the lithographically spaced nanovia pins with the nanoscale pitch nanowires and guarantee that there is space in the CMOS below the nanoscale circuitry for the CMOS restoration and programming circuits. Note of the following:

- Nanoscale wires can be angled relative to the CMOS circuitry to match the pitch of the CMOS nanovias to the nanoscale wires. Figure 38.14 shows this tilt interfacing to a single nanowire array layer. Nanovias that connect to the CMOS are arranged in a square array with side $2\beta F_{CMOS}$, where F_{CMOS} is the half-pitch of the CMOS subsystem, and β is a dimensionless factor larger than 1 that depends on CMOS cell complexity. The nanowire crossbar is turned by an angle $\alpha = \arcsin(F_{nano}/\beta F_{CMOS})$ relative to the CMOS pin array, where F_{nano} is the nanowire half-pitch.

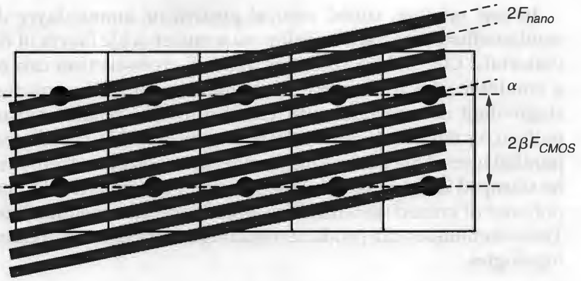


FIGURE 38.14 ■ Nanoscale and CMOS pitch matching via tilt.

- If sufficiently long nanowires are used, the area per nanowire can be as large as each CMOS cell (e.g., restoration buffer and programming transistors). For example, if we use 10 μm nanowires at 10 nm pitch, each nanowire occupies 10^5 nm^2; each such nanowire could have its own 300 nm × 300 nm CMOS cell ($\beta \approx 3$ for $F_{CMOS} = 45$ nm) and keep the CMOS area contained below the nanowire area.

For detailed development of this interface scheme, see Likharev and Strukov [44]. Hewlett-Packard employs a variant of the tilt scheme for their field-programmable nanowire interconnect (FPNI) architecture [41].

38.7.3 Restoration

Enabled by the array-tilt scheme that allows each nanowire to be directly connected to CMOS circuitry, the hybrid semiconductor–molecular electronics (CMOL) and FPNI nanoscale array designs use lithographic-scale CMOS buffers to perform signal restoration and inversion. CMOS buffers with large feature sizes will be larger than nanowire FETs and have less variation. The FPNI scheme uses nanoscale configurability only to provide programmable interconnect, using a nonconfigurable 2-input CMOS NAND/AND gate for logic.

Alternatively, it may be possible to build latches that provide gain and isolation from 2-terminal molecular devices [38]. Specifically, molecules that serve as resonant-tunneling diodes (RTDs) or negative differential resistors have been synthesized [46, 47]. These devices are characterized by a region of negative resistance in their IV-curve. The CMU nanoFabric design shows how to build and integrate latches based on RTD devices. The latches draw their power from the clock and provide restoration and isolation.

38.8 SUMMARY

Between highly regular structures and high defect rates, atomic-scale design appears to demand postfabrication configurability. This chapter shows how configurable architectures can accommodate the extreme regularity required. It further shows that configurable architectures can tolerate extremely limited control during the fabrication process by exploiting large-scale assembly statistics. Consequently, we obtain a path to denser logic using building blocks roughly 10 atoms wide, as well as a path to continued integration in the third dimension.

Spatially configurable design styles become even more important when all substrates are configurable at their base level. We can always configure sequential processors on top of these nanoscale substrates when tasks are irregular and low throughput (see Chapter 36 and the Processor subsection of Section 5.2.2). However, when tasks can be factored into regular subtasks, direct spatial implementation on the configurable substrate will be more efficient, reducing both runtime and energy consumption.

References

[1] S. R. J. Brueck. There are no fundamental limits to optical lithography. *International Trends in Applied Optics*, SPIE Press, 2002.

[2] A. DeHon. Nanowire-based programmable architectures. *ACM Journal on Emerging Technologies in Computing Systems* 1(2), 2005.

[3] Y. Tan, X. Dai, Y. Li, D. Zhu. Preparation of gold, platinum, palladium and silver nanoparticles by the reduction of their salts with a weak reductant–potassium bitartrate. *Journal of Material Chemistry* 13, 2003.

[4] Y. Wu, Y. Cui, L. Huynh, C. J. Barrelet, D. C. Bell, C. M. Lieber. Controlled growth and structures of molecular-scale silicon nanowires. *Nanoletters* 4(3), 2004.

[5] Y. Cui, L. J. Lauhon, M. S. Gudiksen, J. Wang, C. M. Lieber. Diameter-controlled synthesis of single crystal silicon nanowires. *Applied Physics Letters* 78(15), 2001.

[6] B. Zheng, Y. Wu, P. Yang, J. Liu. Synthesis of ultra-long and highly-oriented silicon oxide nanowires from alloy liquid. *Advanced Materials* 14, 2002.

[7] M. S. Gudiksen, J. Wang, C. M. Lieber. Synthetic control of the diameter and length of semiconductor nanowires. *Journal of Physical Chemistry B* 105, 2001.

[8] M. S. Gudiksen, L. J. Lauhon, J. Wang, D. C. Smith, C. M. Lieber. Growth of nanowire superlattice structures for nanoscale photonics and electronics. *Nature* 415, 2002.

[9] Y. Wu, R. Fan, P. Yang. Block-by-block growth of single-crystalline Si/SiGe superlattice nanowires. *Nanoletters* 2(2), 2002.

[10] M. T. Björk, B. J. Ohlsson, T. Sass, A. I. Persson, C. Thelander, M. H. Magnusson, K. Depper, L. R. Wallenberg, L. Samuelson. One-dimensional steeplechase for electrons realized. *Nanoletters* 2(2), 2002.

[11] L. J. Lauhon, M. S. Gudiksen, D. Wang, C. M. Lieber. Epitaxial core-shell and core-multi-shell nanowire heterostructures. *Nature* 420, 2002.

[12] M. Law, J. Goldberger, P. Yang., Semiconductor nanowires and nanotubes. *Annual Review of Material Science* 34, 2004.

[13] M. Ritala. Advanced ALE processes of amorphous and polycrystalline films. *Applied Surface Science* 112, 1997.

[14] M. Ritala, K. Kukli, A. Rahtu, P. I. Räisänen, M. Leskelä, T. Sajavaara, J. Keinonen. Atomic layer deposition of oxide thin films with metal alkoxides as oxygen sources. *Science* 288, 2000.

[15] Y. Wu, J. Xiang, C. Yang, W. Lu, C. M. Lieber. Single-crystal metallic nanowires and metal/semiconductor nanowire heterostructures. *Nature* 430, 2004.

[16] Y. Huang, X. Duan, Q. Wei, C. M. Lieber. Directed assembly of one-dimensional nanostructures into functional networks. *Science* 291, 2001.

[17] D. Whang, S. Jin, C. M. Lieber. Nanolithography using hierarchically assembled nanowire masks. *Nanoletters* 3(7), 2003.

[18] D. Whang, S. Jin, Y. Wu, C. M. Lieber. Large-scale hierarchical organization of nanowire arrays for integrated nanosystems. *Nanoletters* 3(9), 2003.

[19] Y. Chen, D. A. A. Ohlberg, X. Li, D. R. Stewart, R. S. Williams, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, D. L. Olynick, E. Anderson. Nanoscale molecular-switch devices fabricated by imprint lithography. *Applied Physics Letters* 82(10), 2003.

[20] Y. Chen, G.-Y. Jung, D. A. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, R. S. Williams. Nanoscale molecular-switch crossbar circuits. *Nanotechnology* 14, 2003.

[21] D. R. Stewart, D. A. A. Ohlberg, P. A. Beck, Y. Chen, R. S. Williams, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart. Molecule-independent electrical switching in Pt/organic monolayer/Ti devices. *Nanoletters* 4(1), 2004.

[22] C. Collier, G. Mattersteig, E. Wong, Y. Luo, K. Beverly, J. Sampaio, F. Raymo, J. Stoddart, J. Heath. A [2]catenane-based solid state reconfigurable switch. *Science* 289, 2000.

[23] C. L. Brown, U. Jonas, J. A. Preece, H. Ringsdorf, M. Seitz, J. F. Stoddart. Introduction of [2]catenanes into Langmuir films and Langmuir–Blodgett multilayers: A possible strategy for molecular information storage materials. *Langmuir* 16(4), 2000.

[24] A. DeHon. Reconfigurable Architectures for General-Purpose Computing. AI Technical Report 1586, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1996.

[25] A. DeHon. Array-based architecture for FET-based, nanoscale electronics. *IEEE Transactions on Nanotechnology* 2(1), 2003.

[26] A. DeHon, P. Lincoln, J. Savage. Stochastic assembly of sublithographic nanoscale interfaces. *IEEE Transactions on Nanotechnology* 2(3), 2003.

[27] A. DeHon. Law of Large Numbers system design. In *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, Kluwer Academic, 2004.

[28] A. DeHon. Design of programmable interconnect for sublithographic programmable logic arrays. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2005.

[29] B. Gojman, R. Rubin, C. Pilotto, T. Tanamoto, A. DeHon. 3D nanowire-based programmable logic. *Proceedings of the International Conference on Nano-Networks* 2006.

[30] A. DeHon, S. C. Goldstein, P. J. Kuekes, P. Lincoln. Non-photolithographic nanoscale memory density prospects. *IEEE Transactions on Nanotechnology* 4(2), 2005.

[31] A. DeHon. Deterministic addressing of nanoscale devices assembled at sublithographic pitches. *IEEE Transactions on Nanotechnology* 4(6), 2005.

[32] A. DeHon, H. Naeimi. Seven strategies for tolerating highly defective fabrication. *IEEE Design and Test of Computers* 22(4), 2005.

[33] V. Betz, J. Rose. FPGA Place-and-Route Challenge. *http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html*, 1999.

[34] J. R. Heath, P. J. Kuekes, G. S. Snider, R. S. Williams. A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science* 280(5370), 1998.

[35] Y. Luo, P. Collier, J. O. Jeppesen, K. A. Nielsen, E. Delonno, G. Ho, J. Perkins, H.-R. Tseng, T. Yamamoto, J. F. Stoddart, J. R. Heath. Two-dimensional molecular electronics circuits. *ChemPhysChem* 3(6), 2002.

[36] S. Williams, P. Kuekes. Demultiplexer for a molecular wire crossbar network. U.S. Patent number 6,256,767, July 3, 2001.

[37] S. C. Goldstein, M. Budiu. NanoFabrics: Spatial computing using molecular electronics. *Proceedings of the International Symposium on Computer Architecture* 178–189, 2001.

[38] S. C. Goldstein, D. Rosewater. Digital logic using molecular electronics. *ISSCC Digest of Technical Papers*, IEEE, 2002.

[39] A. DeHon, M. J. Wilson. Nanowire-based sublithographic programmable logic arrays. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2004.

[40] D. B. Strukov, K. K. Likharev. CMOL FPGA: A reconfigurable architecture for hybrid digital circuits with two-terminal nanodevices. *Nanotechnology* 16(6), 2005.

[41] G. S. Snider, R. S. Williams. Nano/CMOS architectures using a field-programmable nanowire interconnect. *Nanotechnology* 18(3), 2007.

[42] N. A. Melosh, A. Boukai, F. Diana, B. Gerardot, A. Badolato, P. M. Petroff, J. R. Heath. Ultra high-density nanowire lattices and circuits. *Science* 300, 2003.

[43] M. D. Austin, H. Ge, W. Wu, M. Li, Z. Yu, D. Wasserman, S. A. Lyon, S. Y. Chou. Fabrication of 5 nm linewidth and 14 nm pitch features by nanoimprint lithography. *Applied Physics Letters* 84(26), 2004.

[44] K. K. Likharev, D. B. Strukov. CMOL: Devices, circuits, and architectures. In *Introducing Molecular Electronics*, Springer, 2005.

[45] K. L. Jensen. Field emitter arrays for plasma and microwave source applications. *Physics of Plasmas* 6(5), 1999.

[46] J. Chen, M. Reed, A. Rawlett, J. Tour. Large on-off ratios and negative differential resistance in a molecular electronic device. *Science* 286, 1999.

[47] J. Chen, W. Wang, M. A. Reed, M. Rawlett, D. W. Price, J. M. Tour. Room-temperature negative differential resistance in nanoscale molecular junctions. *Applied Physics Letters* 77, 2000.

INDEX

RECONFIGURABLE COMPUTING: THE THEORY AND PRACTICE OF FPGA-BASED COMPUTATION

EDITED BY SCOTT HAUCK AND ANDRÉ DEHON

THE MORGAN KAUFMANN SERIES IN SYSTEMS ON SILICON

SERIES EDITOR WAYNE WOLF

In the two decades since field-programmable gate arrays (FPGAs) were introduced, they have radically changed the way digital logic is designed and deployed. By marrying the high performance of custom VLSI chips with the flexibility of microprocessors, FPGAs have made possible entirely new types of applications. From full-chip logic verification to radar and image-processing tasks, FPGA-based solutions are often the most efficient way to perform some of today's most challenging tasks.

To make the most of this unique combination of performance and flexibility, designers must understand hardware, software, and FPGA-based application development. This book will teach designers all these issues, enabling them to exploit the vast opportunities possible with reconfigurable logic. The book includes:

- Introduction to FPGA chips and computing boards, including current devices, reconfigurable computing-specific chips, and fast reconfiguration systems.
- Models and languages for coding reconfigurable computing applications.
- CAD flows for automatically mapping to reconfigurable systems.
- Application development and optimization techniques critical for achieving high-quality FPGA-based designs.
- Nine in-depth case studies of important FPGA applications.
- Simple models for understanding the source of the FPGA benefits and the outlook for reconfigurable systems as Moore's Law scaling continues.

This book provides a jumping on point for students and engineers from both hardware and software backgrounds to the challenges and opportunities associated with reconfigurable computing.

COMPUTING/
COMPUTER
HARDWARE
AND DESIGN

NEW

ISBN 978-93-80931-86-9

9 789380 931869

ELSEVIER

MK®
MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
www.mkp.com