Automated Target Recognition on SPLASH 2 †

Michael Rencher and Brad L. Hutchings
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT 84602

Abstract

Automated target recognition is an application area that requires special-purpose hardware to achieve reasonable performance. FPGA-based platforms can provide a high level of performance for ATR systems if the implementation can be adapted to the limited FPGA and routing resources of these architectures. This paper discusses a mapping experiment where a linear-systolic implementation of an ATR algorithm is mapped to the SPLASH 2 platform. Simple column-oriented processors were used throughout the design to achieve high performance with limited nearest-neighbor communication. The distributed SPLASH 2 memories are also exploited to achieve a high degree of parallelism. The resulting design is scalable and can be spread across multiple SPLASH 2 boards with a linear increase in performance.

1 Introduction

Automated target recognition (ATR) is a computationally demanding application area that typically requires special-purpose hardware to achieve desirable performance. ASICs are not an option for these systems due to high non-recurring engineering (NRE) costs and because the algorithms are constantly evolving. Existing FPGA-based computing platforms can potentially provide the necessary performance and flexibility for evolving ATR systems; however, mapping applications to these existing platforms can be very challenging because they lack abundant interconnect and FPGA resources. The key to achieving a high-performance implementation of ATR algorithms with existing platforms is to carefully organize the design of the ATR implementation so that it can communicate via the limited interconnect and can be easily partitioned among the FPGA devices.

This paper presents a linear systolic implementation of an existing ATR algorithm on SPLASH 2 that is well-suited to the SPLASH 2 architecture. Inter-FPGA communication is limited and easily accommodated by the SPLASH 2 interconnect. Moreover, the implementation can be scaled across any number of SPLASH 2 boards and achieves high performance with limited resources.

This paper briefly discusses the entire ATR algorithm as developed by Sandia National Labs, and then overviews the design and implementation of the most computationally demanding part of the algorithm: Chunky SLD. The SPLASH 2 implementation is presented in some detail with future directions and possible improvements.

2 Automatic Target Recognition

The goal of a typical ATR system is to analyze a digital representation of a scene and locate/identify objects that are of interest. Although this goal is conceptually simple, ATR systems have extremely demanding I/O and computational requirements: image data are large, can be generated in real-time, and must be processed quickly so that results remain relevant in a dynamic environment. The common use of special-purpose hardware in nearly all high-performance ATR systems is a clear indication of the computational complexity of these systems.

This paper details the implementation of an existing ATR algorithm on SPLASH 2. The algorithm in question was developed at Sandia National Laboratories and was designed to detect partially obscured targets in Synthetic Aperture Radar (SAR) images. It is commonly referred to as *Chunky SLD*, so named for the second step of the algorithm that differentiates this algorithm from others developed at Sandia. This algorithm consists of the following three steps: (1) Focus of Attention (FOA), (2) Second-Level Detection (SLD), and (3) Final Identification (FI). Each of these steps will now be introduced so that the algorithm implementation can be understood in its operating context.

† This work was supported by DARPA/CSTO under contract number DABT63-94-C-0085 under a subcontract to National Semiconductor

0-8186-8159-4/97 $10.00 © 1997 IEEE

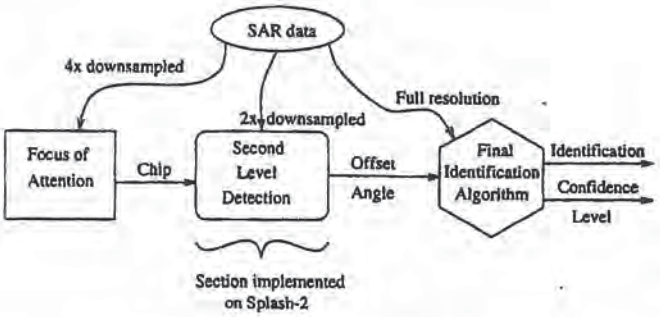


Figure 1: ATR Block Diagram.

2.1 Focus of Attention (FOA)

Focus of attention is the first step of the ATR process and uses image morphology techniques to detect potential targets in SAR data. FOA operates on "down-sampled" SAR images that are approximately 600-1000 pixels on a side. Once FOA detects a potential target, it determines the approximate center of the potential target and creates 2× down-sampled subimages of the original SAR data where each subimage contains a single target centered within the subimage. These subimages are referred to as *chips* and are 128 × 128 pixels.

2.2 Second Level Detection (SLD)

The SLD step processes the chips generated by the FOA step. SLD further restricts the areas of interest by giving the potential targets coordinates and angular orientation. SLD does this by correlating predefined binary templates to the areas of interest. The templates represent different object orientation angles. Templates are oriented between 5 and 10 degrees apart. SLD also uses adaptive threshold levels determined by the overall image intensity.

The algorithm studied in the paper is a variation of SLD called Chunky SLD. Chunky SLD adds a level of complexity to SLD by using more templates to represent objects that have been partially obscured (partially hidden by camouflage or objects overhead). This allows better target recognition at a cost of higher computational requirements. Chunky SLD is discussed in more detail later in this section.

2.3 Final Identification (FI)

The FI algorithm correlates full resolution image data and templates with finer angular resolution (3 to 5 degrees). FI also uses adaptive threshold levels. The output of FI is a location of the target, and confidence level corresponding to the level of correlation between the object and the FI templates.

2.4 The Chunky SLD Algorithm

The general goal of the Chunky-SLD algorithm is to recognize targets that are partially concealed or obscured in some way. To achieve this goal, the designers of this algorithm treat the target as a set of 40 template pairs where each pair of templates is a digital representation of some salient feature of the specific target. If the majority of the template pairs strongly correlate with the image data, then a match of the overall target is assumed. Each pair of templates consists of a *Bright* template and a *Surround* template. The Bright template is a representation of expected reflections directly from surfaces of a salient target feature while the Surround template represents expected absorption in the immediate area surrounding the target feature. Each pair of a Bright and Surround template is referred to as a *chunk*, so called because each pair of templates represents a "chunk" of the overall target. Each set of 40 chunks represents a single target at a specific rotation. There are 72 orientations, each representing a different target orientation and radar incidence angle. Each set of 72 orientations is referred to as a *class* and is the complete set of templates that must be correlated with a chip to detect the presence of a specific target.

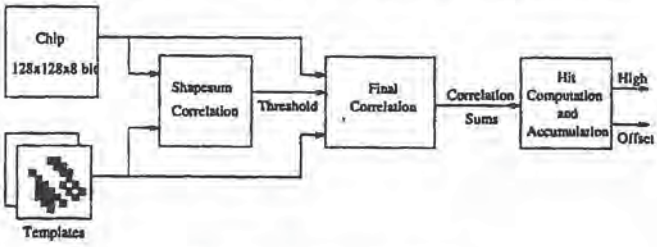


Figure 2: Chunky SLD

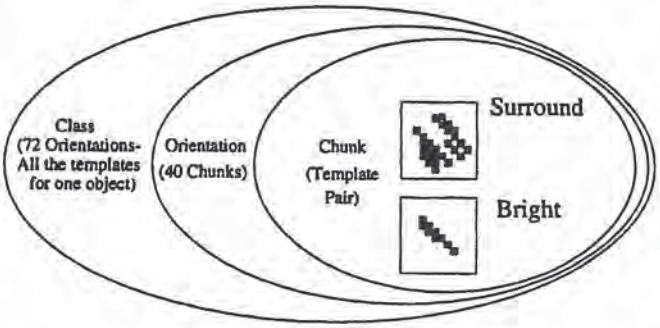


Figure 3: Template Organization

The first step of the Chunky SLD algorithm is to correlate the chip and the Bright template. This correlation value is used to compute a value that will be

used to threshold the incoming chip data, converting the 8-bit chip image into a binary image. The equations describing this process are shown below.

$$Shapesum(x,y) = \sum_{a,b=0}^{15} B_{temp}(a,b) * Chip(x+a, y+b)$$

$$Threshold(x,y) = \frac{Shapesum(x,y)}{Bright\_template\_pixel\_count}$$

The values obtained by correlating the Bright and Surround templates with the binarized chip (B_{sum} and S_{sum}) are checked against minimum values to generate a "hit value" for each offset in the chip. The threshold value is also checked to see if it falls in an acceptable range when generating the hit values.

$$\begin{aligned} &if([T_{max} \geq T \geq T_{min}]\ AND \\ &\quad [B_{sum} \geq B_{min}]\ AND \\ &\quad [S_{sum} \geq S_{min}]) \\ &then \\ &\quad Hit = 1; \\ &else \\ &\quad Hit = 0; \end{aligned} \tag{1}$$

The hit values are accumulated for each offset for a specific orientation (40 chunks). The highest values are used to identify the areas of interest for the final identification step.

2.4.1 Template Characteristics

Template pairs exhibit useful properties: sparseness and mutual exclusivity. The Bright template consists mostly of zeros; only 3 to 10 percent of the template values are '1's and this limits the magnitude of the Shapesum and B_{sum} values. The Bright and Surround templates are also mutually exclusive; that is, if the two templates are overlaid no "on" pixels will overlap. When carefully exploited, both of these properties lead to more compact and higher performance hardware.

3 Other Implementations of Chunky-SLD

As explained the ATR application is computationally demanding. There are (128-15) × (128-15) offsets per chunk × 40 chunks × 72 orientations ≅ 36 × 10^6 hit values to compute per targeted object (or per class, see Figure 3). The computational rate and I/O requirements of this algorithm make it impossible to use current microprocessors. Thus any high-performance implementation of this algorithm will require special-purpose hardware to meet performance goals.

However, custom ASICs are also not an option because the algorithm is constantly evolving and also because commercial-off-the-shelf components (COTS) are often dictated by the ultimate customers of ATR systems. The only remaining options are to construct the system with commercially available fixed-function devices such as correlaters, multipliers, etc., or to use programmable logic, e.g., FPGAs [1, 2]. Thus all known implementations of Chunky-SLD use either fixed-function devices or programmable logic.

3.1 Sandia

Current Sandia implementations of ATR are based on commercially available one-bit correlater chips. Sandia designers adapted the basic Chunky-SLD algorithm so they could exploit the capabilities of these components to achieve high performance. Rather than process the Shapesum and then process the final correlation, the two steps were done in parallel. The correlation was done at 8 discrete threshold levels and the Shapesum determined which threshold to use for each offset.

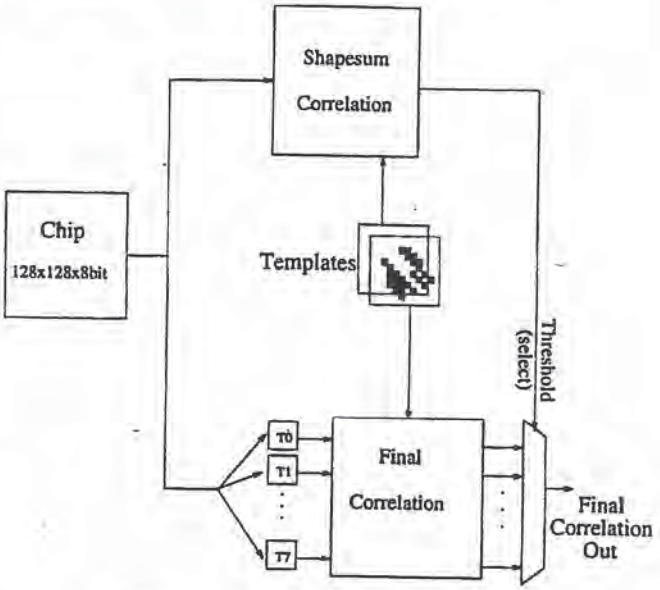


Figure 4: Sandia's Implementation.

3.2 UCLA

A group at UCLA (Headed by John Villasenor) [3] is working on a FPGA based SLD implementation. By doing bit level correlations they are able to do very compact adder trees that take advantage of template sparseness and FPGA on board lookup-table memory capability. Their approach compiles template information directly into the hardware and relies on fast reconfiguration to switch template information. They

also try and take advantage of template overlap by computing the results of multiple correlations simultaneously.

4 Chunky-SLD on Splash 2

On SPLASH 2 a static implementation was done to avoid the overhead of reconfiguring the hardware during execution. In order to reduce the hardware requirements without reducing throughput a deeply pipelined design (~400 cycle latency) was implemented. The Shapesum unit generates the threshold value which is then used to generate the final correlation values. (Note: There is a unique threshold for each offset). By doing this only two final correlations have to be computed per offset (one B_{sum} and one S_{sum}).

The Sandia implementation computes the Shapesum and final correlation in parallel which forces them to compute multiple final correlations. While our implementation does them serially. This allows us to use an exact threshold value. Also only one final correlation needs to be computed because the threshold value is computed before the final correlation begins. The technique used was to look at the correlations by column, compute the partial correlation for that column, and sum up the partial sums for all 16 columns. In this method 16 different column correlations are going on in parallel but only one column of data needs to be available for processing.

4.1 Implementing the Correlation as Column Sums

Figure 5 depicts a simple example that demonstrates a correlation of a 3×3 template with a binary image. Each row in the table represents one clock cycle. The first column is the clock cycle number. Corresponding numbers are found in the *Pixel load order* box at the right. A new pixel is brought in on each clock cycle. The P1, P2, and P3 columns represent the three column processing units needed for a three column template. The last column represents the actual output. Clock cycles 9 through 12 have been expanded to show how data (pixels and partial sums) are passed from column to column and illustrate the data format. Once the pipeline is full, a new correlation value is computed as each column arrives (three pixels/cycles).

Note that valid output comes every three cycles because the template is three rows tall. All processing elements are actively processing 3 pixel values at all times. The SPLASH 2 implementation works just like the example except for the size of the columns (16 pixels instead of three) and the data format (eight-bit instead of one-bit).

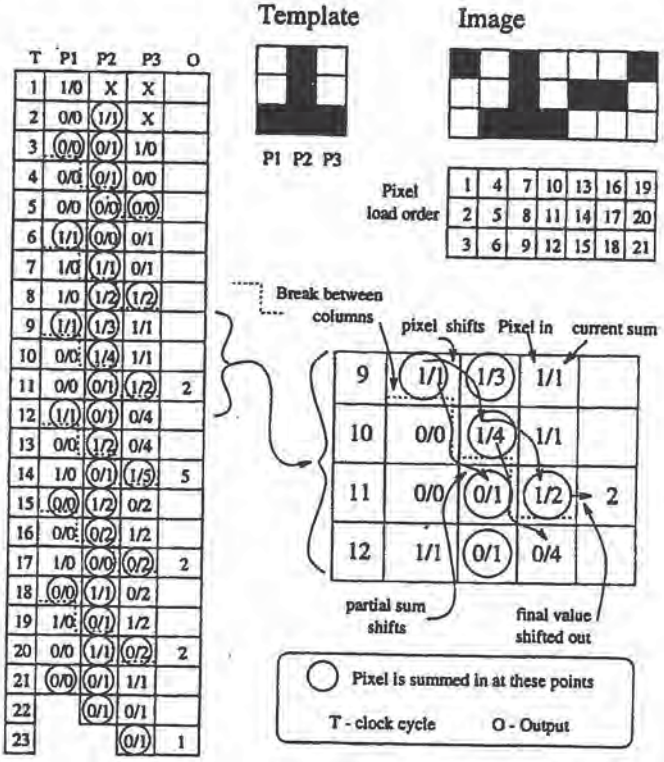


Figure 5: Example Column Sum.

4.2 Platforms/Implementations

Chunky SLD was implemented on the SPLASH 2 board. SPLASH 2 has shown itself to be a useful platform and has had numerous applications mapped to it [4, 5, 6, 7, 8, 9, 10, 11]. The implementation was done in VHDL and simulated/synthesized using Synopsis. All place and route was done automatically using Xilinx place and route tools.

One of the goals of the implementation was to run the system so that it consumed a pixel per cycle. This means that each cycle all processing elements (PE) need to be able to process a new pixel. This implementation follows Sandia National Labs algorithms (not implementation) as closely as possible (see Section 2.4).

The SPLASH 2 board was developed by SRC (Supercomputing Research Center Institute for Defense Analyses) [12]. The SPLASH 2 board is a linear systolic array of processing elements (FPGAs), each with their own memory.

4.2.1 Splash 2 Hardware

From a conceptual point of view, the SPLASH 2 system consists of a linear array of processing elements. This

makes SPLASH 2 a good candidate for linear-systolic applications with limited neighbor-to-neighbor interconnect. Because of limited routing resources SPLASH 2 has difficulty implementing multi-chip systems that are not linear systolic, though they are possible [8].

The actual SPLASH 2 platform consists of a board with 16 Xilinx 4010 chips (plus one for control) arranged in a linear systolic array. Each chip has a limited 36-bit connection to its two nearest neighbors. Each Xilinx 4010 is connected to a 512 kbyte memory

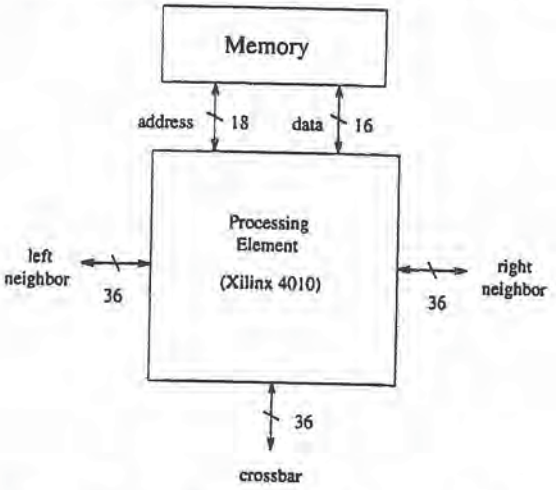


Figure 6: Single Processing Element of SPLASH 2.

(16-bit word size). The memory can handle back-to-back reads, or back-to-back writes, but requires one 'dead' (or turn around) cycle when changing from write to read. There is also a crossbar connected to all of the chips that allows some level of random connection between chips. Up to 16 boards can be daisy-chained together to provide a large linear-systolic array of 256 elements.

4.3 ATR Implementation on Splash 2

Similar to the example, the SPLASH 2 implementation processes one pixel at a time and loads them in column order so that the partial sums can be generated and passed from column to column. All template data are stored in the memories adjacent to the FPGAs on the SPLASH 2 boards. Each memory can hold several thousand templates thus making it possible to store all of the templates for a single class (5760) on a single SPLASH 2 board. There is sufficient room in the FPGA design to store a single template. The templates are switched by reading the new template data out of the memory and storing it within the FPGA. However, because this implementation is deeply pipelined, it is necessary to flush all current

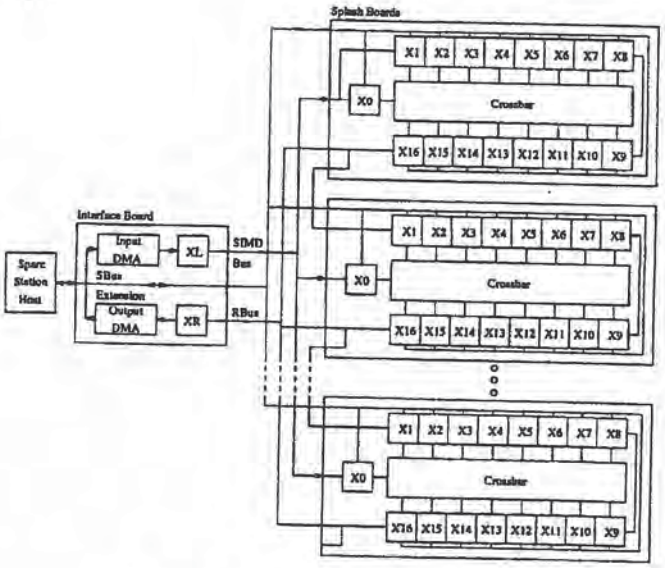


Figure 7: SPLASH 2 platform.

data from the system when switching to a new template. The overhead from the flushing operation is minimal ($\frac{318\ flush\ cycles}{231424\ compute\ cycles} =$) 0.14%.

During each clock cycle, a new pixel arrives at the FPGA. If the template bit corresponding to this pixel is *on* then the incoming pixel is added to the current partial sum. Each 16 clock cycles, this partial sum is then passed on to the next column and a new partial sum is received from the previous column. The last column computes a complete Shapesum every 16 cycles (one column). The final correlation of the Bright and Surround templates with the thresholded chip data works similarly except there are two correlations (one for each template).

Intermediate hit values are stored in a table, referred to as the *hit-table*, in one of the local memories. Each location in the table corresponds to an x-y offset of a chip, the origin of a single correlation. For each offset, if a chunk "hits", then the corresponding location in this table is incremented. Thus the table contains the accumulated hit values for all chunks and all offsets that have been computed to that point.

Hits are computed according to Equation 1. First, each B_{sum} and S_{sum} value is compared to its corresponding minimum value. Second, the threshold value corresponding to each B_{sum} and S_{sum} is checked to see if it is between a certain minimum and maximum value. For reasons of efficiency, the threshold value is actually examined earlier in the process and a zero for the threshold is stored in lookup-table memory if it is out of bounds. This works correctly because if the

threshold is zero, it will cause the B_{sum} to be zero, which will in turn cause the B_{sum} comparison to fail. Otherwise, if all three of these tests come back true then a hit has been found for the corresponding offset (see Equation 1) and the corresponding location in the hit-table is incremented. After the 40 templates are tested against the same chip the two offsets with the highest accumulated hit values are written into memory where the host computer can read them. This is accomplished by examining the hit values during this process and retaining the top two values in special memory locations. These final two hit values (which represent the top two orientations for a specific class) are used in the FI step.

For the SPLASH 2 board, as with most FPGA systems, partitioning is a major issue. The design needed to be modular so that different design modules could be reassigned to different FPGAs as necessary. This is where the column modules were so valuable (see Figure 8).

4.4 Special Features

This implementation has several notable characteristics. They include control distribution, modular design for partitioning and memory utilization.

4.4.1 Distributed Control

The control in this system is distributed throughout the array. Each column module has it's own state machine based control. Module synchronization is achieved by distributing a control token through the pipeline along with the data. When a module receives this signal, it resets its internal state machines and retrieves template data from its local memory. A memory controller resides in each processing element (FPGA) to retrieve template data and give memory access to all other modules.

4.4.2 Modular Design (Design for Partitioning)

Each column in both the Shapesum and final correlation use totally self contained modules that can be easily migrated from processing element to processing element. This was done to simplify the partitioning onto SPLASH 2 [13]. Memory data had to be carefully partitioned as well so that the data could follow the module to which it applied. The regularity of the design was an important concern; it allowed the placement of specific circuit modules to be dictated by the requirements of the algorithm and not by the limited interconnect of the platform. There are 16 identical modules in the Shapesum and 16 more identical modules in the final correlation. Along with these there is a divide, a hit accumulator and 2 delay modules (see Figure 8).

4.4.3 Memory Usage

The memories in SPLASH 2 serve several purposes. The template information is stored in them. They are used to implement video shift registers that correct for the latency incurred during threshold computation. These shift registers require that two memories be used in tandem because every clock cycle a new pixel (8 bits) had to be written to memory and a delayed pixel had to be read from memory. The bandwidth of one memory was such that it can handle two pixels (load and store) every three cycles. Thus one memory would delay two pixels and skip two pixels, while the other memory would delay the two pixels that the first memory skipped and skip the two pixels that the first memory delayed. The divide unit and the final result including the accumulated hit values are also stored in memory.

4.5 Performance Metrics

There are many metrics that could be used to measure the value of this implementation. This section is devoted to discussing some of these metrics.

4.5.1 Performance

This implementation runs at a frequency of 19 MHz using a test template that tests all column modules. `Xdelay` (a Xilinx timing tool) reports a guaranteed frequency of 13.2 MHz. Designs that will run in the 10 to 20 MHz range are typical [8, 10].

Using the above frequency a single system could process one orientation every .487 seconds (.701 seconds using a 13.2 MHz clock).

$$\frac{128\ col \times (128-15)\ row \times 16\frac{pix}{col}}{19\ MHz} \times 40\frac{Chunks}{Orient} = (12\ ms/Chunk) \times 40\frac{Chunks}{Orientation} = \boxed{487\text{ ms @ 19 MHz}}$$

or

$$\frac{128\ col \times (128-15)\ row \times 16\frac{pix}{col}}{13.2\ MHz} \times 40\frac{Chunks}{Orient} = (17.5\ ms/Chunk) \times 40\frac{Chunks}{Orientation} = \boxed{701\text{ ms @ 13.2 MHz}}$$

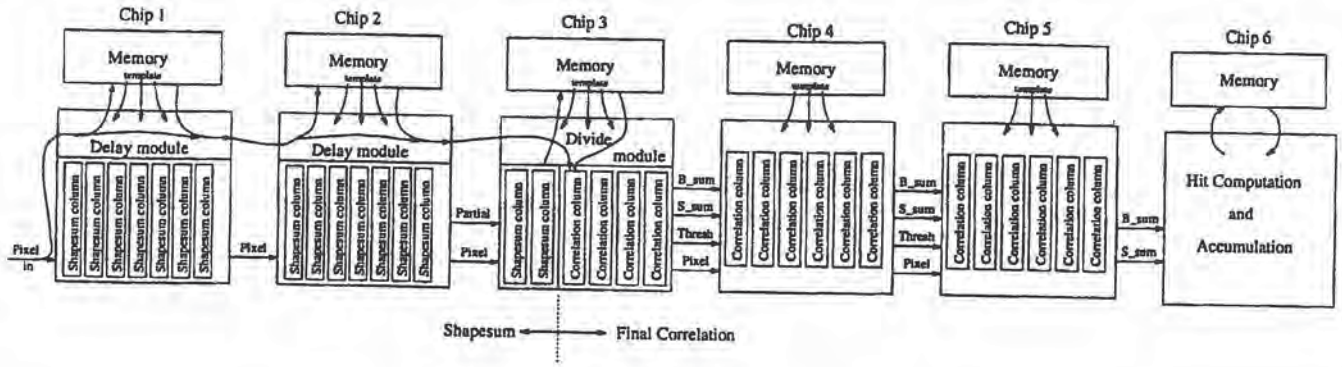


Figure 8: SPLASH 2 Implementation.

Using a brute force approach it takes 59 seconds to compute the same data on an HP 770 (110 MHz). That is two orders of magnitude slower. On one SPLASH 2 board two processing units can be installed yielding the ability to process 1 orientation (40 chunks) every .244 seconds (.350 seconds at 13.2 MHz).

4.5.2 Multi-board Splash 2

If a 16-board system were used then 256 processing elements would be available. this would make 42 processing units (256 PEs ÷ 6 PE per PU) and 42× the throughput. Thus one orientation can be processed every 11.6 ms (16.7 ms at 13.2 MHz). This is equal to covering 86 orientation per second (60 orientations per second at 13.2 MHz).

4.5.3 Memory Bandwidth

The distributed memory and aggregate memory bandwidth is what makes this implementation possible. Because the memory is distributed to each chip this implementation can be extended to a multi-board system quite easily. Consider that a processing unit consists of 6 FPGAs and their corresponding memories. Each processing unit performs 10 reads and 13 writes each 16 clock cycles. The SPLASH 2 memories require 3 clock cycles to perform a read and a write for a total available bandwidth of 2 reads and 2 writes every cycle ($\frac{6\ memories}{3\ cycles}$). Thus a processing unit consumes approximately 1/2 of the available memory bandwidth. If there were only two or three memories available for the six-chip system this implementation would not be possible.

4.5.4 Memory stored templates

The decision to use statically configured hardware and memory-stored templates was made early in the design process. Another approach would be to use hard-coded templates, e.g., hardware with the template data compiled in as constants. A hard-coded template implementation could possibly use less hardware by removing unused accumulators and template reading logic, however it would require the hardware to be reconfigured during run-time in order to switch templates. This would take significantly longer than the memory read done in the memory-stored template version. The SPLASH 2 system requires approximately 17 ms per chip to configure. The system would have to be reconfigured each time a new template was used. In order to process one orientation the system would have to be configured 40 different times. This would adversely impact system throughput. In fact even if the system could be reconfigured in one cycle the performance wouldn't improve because the time spent reading templates from memory is hidden while the pipeline is initially filling (during the flush cycles). There are other advantages as well. Design time is simplified because new hardware bit streams don't have to be generated and tested to use new templates. Re-compilation may improve hardware utilization, however that would come at a cost of repartitioning difficulty.

5 Future work

In the future, additional related research will be conducted on possible improvements to the implementation described in this paper. In addition, alternative implementations that are not currently possible with the SPLASH 2 platform are under examination. Finally, work is underway on a bit-serial version of this implementation.

5.1 Splash 2 Improvements

Although SPLASH 2 was suitable for this project, there are many possible architectural improvements that could be made that would increase performance significantly. A faster memory interface that supports a two-cycle read-write operation or perhaps a dual-ported memory interface would provide improved memory bandwidth. A larger processing element would give more resources and make implementations more flexible. Updated VHDL libraries that conform to standard data types would help design portability and simplicity.

If chip-to-chip I/O were increased on SPLASH 2 a full column could be processed each clock cycle. This would require at least 339 I/O pins ([128 pixel bits + 13 bit partial sum + 8 bit threshold + 2 bits of control] × 2 (in and out) + 16-bit memory data + 21-bit memory address = 339 pins). The current chips (Xilinx 4010) has enough logic and routing to process an entire column at one time. If this kind of I/O bandwidth were available [14] the system would consume 2 SPLASH 2 boards but would have 16 times the throughput because 16 times as much data would be processed each clock cycle.

Another option would be to distribute the *chip* image to each FPGA memory and have a wide memory data path. This would require at least 195 I/O pins ([13 bit partial sum + 8 bit threshold + 2 bit control] × 2 (in and out) + 128 bit memory data path + 21 bit memory address = 195) but would have the same throughput.

5.1.1 Interface Improvements

The current implementation assumes that all data are loaded on the SPLASH 2 board before processing commences. A more realistic implementation would stream data to the system as it is produced. Thus, DMA channels or other high-performance I/O hardware, if added to the system, would make performance more sustainable across multiple images.

5.2 Other implementations

There are several ways this algorithm could be implemented. The most obvious way is to to a full parallel version that looks at a full 16×16 pixels at a time. Another is a linear systolic version that looks at one pixel at a time. A linear systolic implementation was designed because SPLASH 2 is organized that way.

This implementation takes a relatively brute-force approach and feeds the entire chip into the system for processing. Whether or not a given pixel actually contributes to the calculation is determined after the image data are already in the pipeline. Other approaches may take a more selective approach and only access image pixels that actually contribute to the calculation. This may reduce overall memory bandwidth requirements and suggest other implementation strategies. An implementation that only examined pixels that will be used could potentially give a performance increase when correlating the Bright template because the majority of pixels in the template represent chip locations that do not contribute to the computation. However, such an approach would require a complicated control scheme and may not map well onto SPLASH 2.

Here at BYU there is ongoing work targeting 3 different FPGA architectures, each looking to better understand the hardware and ATR algorithms. These include a bit-serial version targeted at National Semiconductors Clay FPGA and a Xilinx 6200. Another project is looking at a full parallel implementation targeting an Altera Flex-10K part.

6 Conclusion

SPLASH 2 is organized as a linear array of FPGA chips each connected to a separate memory. Because of this, applications can only make effective use of SPLASH 2 resources if (1) individual circuit modules are small enough to easily fit into the limited resources available within a single FPGA (Xilinx 4010) and (2) circuit modules can be grouped together in such a way that the interconnect requirements between modules on separate chips will fit in the limited chip-to-chip interconnect available on SPLASH 2.

Both of these requirements were met in this application by carefully organizing all correlations (both Shapesum and final correlation) as independent column-oriented processing elements (PEs). Each PE occupied only a small amount of FPGA resources thereby making it possible to place several PEs into a single FPGA. PEs that are integrated on a single FPGA communicate via the available on-chip FPGA routing; PEs that are located on different FPGAs can easily communicate via the limited inter-chip routing on SPLASH 2.

This repetitive, column-oriented organization also eases the programming of SPLASH 2 considerably. Much of the VHDL code was reused and because the fourth and fifth chips are identical (because of the linear column layout, see Figure 8) it was possible to down-load the same bit-stream into both chips.

This column organization presents a good tradeoff between a fully parallel design and a design that fits within the restrictions of the SPLASH 2 architecture. Each column operates in parallel computing a column

sum that when complete is passed onto its neighboring column. While it is possible to achieve higher levels of parallelism, it is likely that such designs will require denser interconnect than SPLASH 2 is capable of providing (see Section 5.1). By limiting communication to only adjacent columns, significant parallelism is still achieved with only moderate amounts of interconnect.

This implementation can also take advantage of SPLASH 2's extensibility giving it even higher performance capability. Simply by passing the pixel data from chip 6 (see Figure 8) to the next chip on the SPLASH 2 board and configuring the next 6 chips in the array with the same bit stream as the first 6 chips the throughput on the system would be doubled. This extensibility can also be expanded to multiple boards giving a linear increase in system throughput.

References

[1] S. Trimberger. A reprogrammable gate array and applications. *Proceedings of the IEEE*, pages 1030–1041, July 1993.

[2] National Semiconductor. *Configurable Logic Array (CLAy)*. National Semicondutor, 1993.

[3] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79, Napa, CA, April 1996.

[4] A. L. Abbott, P. M. Athanas, L. Chen, and R. L. Elliott. Finding lines and building pyramids with Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 155–161, Napa, CA, April 1994.

[5] J. M. Arnold. The Splash 2 software environment. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 88–93, Napa, CA, April 1993.

[6] P. Athanas and A. Abbott. Image processing on a custom computing platform. In R. Hartenstein and M. Z. Servit, editors, *Field-Programmable Logic: Architectures, Synthesis and Applications. 4th International Workshop on Field-Programmable Logic and Applications*, pages 156–167, Prague, Czech Republic, September 1994. Springer-Verlag.

[7] M. Gokhale and B. Schott. Data parallel C on a reconfigurable logic array. *Journal of Supercomputing*, 9(3):291–313, 1994.

[8] P. Graham and B. Nelson. A hardware genetic algorithm for the travelling salesman problem on SPLASH 2. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 352–361, Oxford, England, August 1995. Springer.

[9] D. T. Hoang. Searching genetic databases on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Napa, CA, April 1993.

[10] D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–177, Napa, CA, April 1993.

[11] N. Ratha, A. Jain, and D. Rover. Convolution on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 204–213, Napa, CA, April 1995.

[12] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–324, June 1992.

[13] Benjamin W. Wah. Systolic arrays–from concept to implementation. *IEEE Journal Computer*, 20(7):12–17, July 1987.

[14] Altera. *Altera–Embedded Programmable Logic Family–Flex 10K Data Sheet*. Altera, 1995.

Attachment 2B

Automated target recognition on SPLASH 2

2 Author(s)

M. Rencher ; B.L. Hutchings

View All Authors

Abstract:
Automated target recognition is an application area that requires special-purpose hardware to achieve reasonable performance. FPGA-based platforms can provide a high level of performance for ATR systems if the implementation can be adapted to the limited FPGA and routing resources of these architectures. The paper discusses a mapping experiment where a linear-systolic implementation of an ATR algorithm is mapped to the SPLASH 2 platform. Simple column oriented processors were used throughout the design to achieve high performance with limited nearest neighbor communication. The distributed SPLASH 2 memories are also exploited to achieve a high degree of parallelism. The resulting design is scalable and can be spread across multiple SPLASH 2 boards with a linear increase in performance.

Published in: Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)

Date of Conference: 16-18 April 1997

Date Added to IEEE *Xplore*: 06 August 2002

Print ISBN: 0-8186-8159-4

INSPEC Accession Number: 5731389

DOI: 10.1109/FPGA.1997.624619

Publisher: IEEE

Conference Location: Napa Valley, CA, USA, USA

Download PDF

Download Citation

View References

Email

Print

Request Permissions

Alerts

Authors

References

Citations

Keywords

Related Articles

Back to Top

IEEE Account

» Change Username/Password

» Update Address

Purchase Details

» Payment Options

» Order History

» View Purchased Documents

Profile Information

» Communications Preferences

» Profession and Education

» Technical Interests

Need Help?

» **US & Canada:** +1 800 678 4333

» **Worldwide:** +1 732 981 0060

» Contact & Support

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

Automated Target Recognition on SPLASH 2 †

Michael Rencher and Brad L. Hutchings
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT 84602

Abstract

Automated target recognition is an application area that requires special-purpose hardware to achieve reasonable performance. FPGA-based platforms can provide a high level of performance for ATR systems if the implementation can be adapted to the limited FPGA and routing resources of these architectures. This paper discusses a mapping experiment where a linear-systolic implementation of an ATR algorithm is mapped to the SPLASH 2 platform. Simple column-oriented processors were used throughout the design to achieve high performance with limited nearest-neighbor communication. The distributed SPLASH 2 memories are also exploited to achieve a high degree of parallelism. The resulting design is scalable and can be spread across multiple SPLASH 2 boards with a linear increase in performance.

1 Introduction

Automated target recognition (ATR) is a computationally demanding application area that typically requires special-purpose hardware to achieve desirable performance. ASICs are not an option for these systems due to high non-recurring engineering (NRE) costs and because the algorithms are constantly evolving. Existing FPGA-based computing platforms can potentially provide the necessary performance and flexibility for evolving ATR systems; however, mapping applications to these existing platforms can be very challenging because they lack abundant interconnect and FPGA resources. The key to achieving a high-performance implementation of ATR algorithms with existing platforms is to carefully organize the design of the ATR implementation so that it can communicate via the limited interconnect and can be easily partitioned among the FPGA devices.

This paper presents a linear systolic implementation of an existing ATR algorithm on SPLASH 2 that is well-suited to the SPLASH 2 architecture. Inter-FPGA communication is limited and easily accommodated by the SPLASH 2 interconnect. Moreover, the implementation can be scaled across any number of SPLASH 2 boards and achieves high performance with limited resources.

This paper briefly discusses the entire ATR algorithm as developed by Sandia National Labs, and then overviews the design and implementation of the most computationally demanding part of the algorithm: Chunky SLD. The SPLASH 2 implementation is presented in some detail with future directions and possible improvements.

2 Automatic Target Recognition

The goal of a typical ATR system is to analyze a digital representation of a scene and locate/identify objects that are of interest. Although this goal is conceptually simple, ATR systems have extremely demanding I/O and computational requirements: image data are large, can be generated in real-time, and must be processed quickly so that results remain relevant in a dynamic environment. The common use of special-purpose hardware in nearly all high-performance ATR systems is a clear indication of the computational complexity of these systems.

This paper details the implementation of an existing ATR algorithm on SPLASH 2. The algorithm in question was developed at Sandia National Laboratories and was designed to detect partially obscured targets in Synthetic Aperture Radar (SAR) images. It is commonly referred to as *Chunky SLD*, so named for the second step of the algorithm that differentiates this algorithm from others developed at Sandia. This algorithm consists of the following three steps: (1) Focus of Attention (FOA), (2) Second-Level Detection (SLD), and (3) Final Identification (FI). Each of these steps will now be introduced so that the algorithm implementation can be understood in its operating context.

† This work was supported by DARPA/CSTO under contract number DABT63-94-C-0085 under a subcontract to National Semiconductor

0-8186-8159-4/97 $10.00 © 1997 IEEE

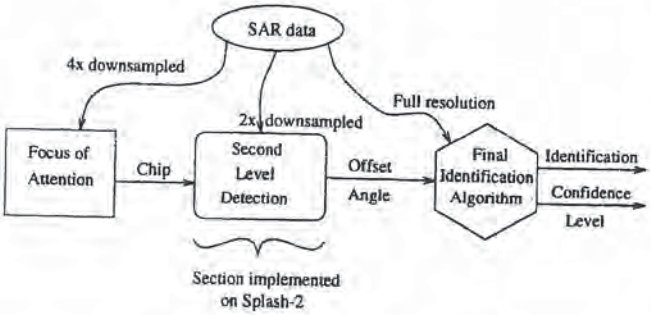


Figure 1: ATR Block Diagram.

2.1 Focus of Attention (FOA)

Focus of attention is the first step of the ATR process and uses image morphology techniques to detect potential targets in SAR data. FOA operates on "down-sampled" SAR images that are approximately 600-1000 pixels on a side. Once FOA detects a potential target, it determines the approximate center of the potential target and creates 2× down-sampled subimages of the original SAR data where each subimage contains a single target centered within the subimage. These subimages are referred to as *chips* and are 128 × 128 pixels.

2.2 Second Level Detection (SLD)

The SLD step processes the chips generated by the FOA step. SLD further restricts the areas of interest by giving the potential targets coordinates and angular orientation. SLD does this by correlating predefined binary templates to the areas of interest. The templates represent different object orientation angles. Templates are oriented between 5 and 10 degrees apart. SLD also uses adaptive threshold levels determined by the overall image intensity.

The algorithm studied in the paper is a variation of SLD called Chunky SLD. Chunky SLD adds a level of complexity to SLD by using more templates to represent objects that have been partially obscured (partially hidden by camouflage or objects overhead). This allows better target recognition at a cost of higher computational requirements. Chunky SLD is discussed in more detail later in this section.

2.3 Final Identification (FI)

The FI algorithm correlates full resolution image data and templates with finer angular resolution (3 to 5 degrees). FI also uses adaptive threshold levels. The output of FI is a location of the target, and confidence level corresponding to the level of correlation between the object and the FI templates.

2.4 The Chunky SLD Algorithm

The general goal of the Chunky-SLD algorithm is to recognize targets that are partially concealed or obscured in some way. To achieve this goal, the designers of this algorithm treat the target as a set of 40 template pairs where each pair of templates is a digital representation of some salient feature of the specific target. If the majority of the template pairs strongly correlate with the image data, then a match of the overall target is assumed. Each pair of templates consists of a *Bright* template and a *Surround* template. The Bright template is a representation of expected reflections directly from surfaces of a salient target feature while the Surround template represents expected absorption in the immediate area surrounding the target feature. Each pair of a Bright and Surround template is referred to as a *chunk*, so called because each pair of templates represents a "chunk" of the overall target. Each set of 40 chunks represents a single target at a specific rotation. There are 72 orientations, each representing a different target orientation and radar incidence angle. Each set of 72 orientations is referred to as a *class* and is the complete set of templates that must be correlated with a chip to detect the presence of a specific target.

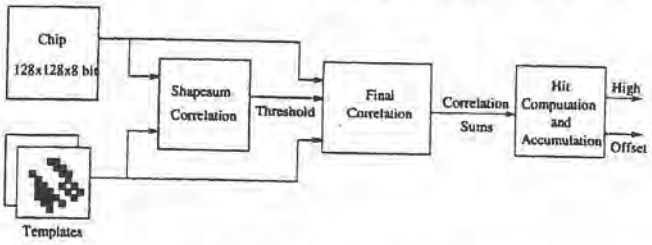


Figure 2: Chunky SLD

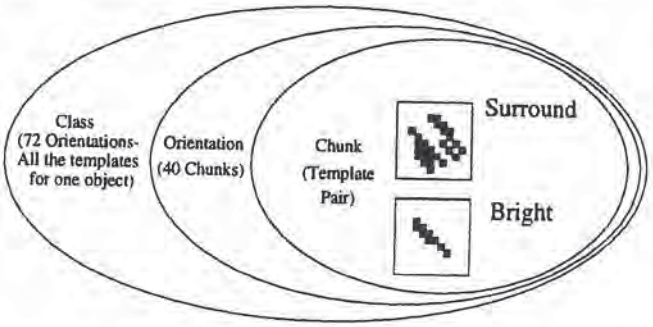


Figure 3: Template Organization

The first step of the Chunky SLD algorithm is to correlate the chip and the Bright template. This correlation value is used to compute a value that will be

used to threshold the incoming chip data, converting the 8-bit chip image into a binary image. The equations describing this process are shown below.

$$Shapesum(x,y) = \sum_{a,b=0}^{15} B_{temp}(a,b) * Chip(x+a, y+b)$$

$$Threshold(x,y) = \frac{Shapesum(x,y)}{Bright\_template\_pixel\_count}$$

The values obtained by correlating the Bright and Surround templates with the binarized chip (B_{sum} and S_{sum}) are checked against minimum values to generate a "hit value" for each offset in the chip. The threshold value is also checked to see if it falls in an acceptable range when generating the hit values.

$$\begin{aligned} &if([T_{max} \geq T \geq T_{min}]\ AND \\ &\quad [B_{sum} \geq B_{min}]\ AND \\ &\quad [S_{sum} \geq S_{min}]) \\ &then \\ &\quad Hit = 1; \\ &else \\ &\quad Hit = 0; \end{aligned} \tag{1}$$

The hit values are accumulated for each offset for a specific orientation (40 chunks). The highest values are used to identify the areas of interest for the final identification step.

2.4.1 Template Characteristics

Template pairs exhibit useful properties: sparseness and mutual exclusivity. The Bright template consists mostly of zeros; only 3 to 10 percent of the template values are '1's and this limits the magnitude of the Shapesum and B_{sum} values. The Bright and Surround templates are also mutually exclusive; that is, if the two templates are overlaid no "on" pixels will overlap. When carefully exploited, both of these properties lead to more compact and higher performance hardware.

3 Other Implementations of Chunky-SLD

As explained the ATR application is computationally demanding. There are (128-15) × (128-15) offsets per chunk × 40 chunks × 72 orientations ≅ 36 × 10^6 hit values to compute per targeted object (or per class, see Figure 3). The computational rate and I/O requirements of this algorithm make it impossible to use current microprocessors. Thus any high-performance implementation of this algorithm will require special-purpose hardware to meet performance goals.

However, custom ASICs are also not an option because the algorithm is constantly evolving and also because commercial-off-the-shelf components (COTS) are often dictated by the ultimate customers of ATR systems. The only remaining options are to construct the system with commercially available fixed-function devices such as correlaters, multipliers, etc., or to use programmable logic, e.g., FPGAs [1, 2]. Thus all known implementations of Chunky-SLD use either fixed-function devices or programmable logic.

3.1 Sandia

Current Sandia implementations of ATR are based on commercially available one-bit correlater chips. Sandia designers adapted the basic Chunky-SLD algorithm so they could exploit the capabilities of these components to achieve high performance. Rather than process the Shapesum and then process the final correlation, the two steps were done in parallel. The correlation was done at 8 discrete threshold levels and the Shapesum determined which threshold to use for each offset.

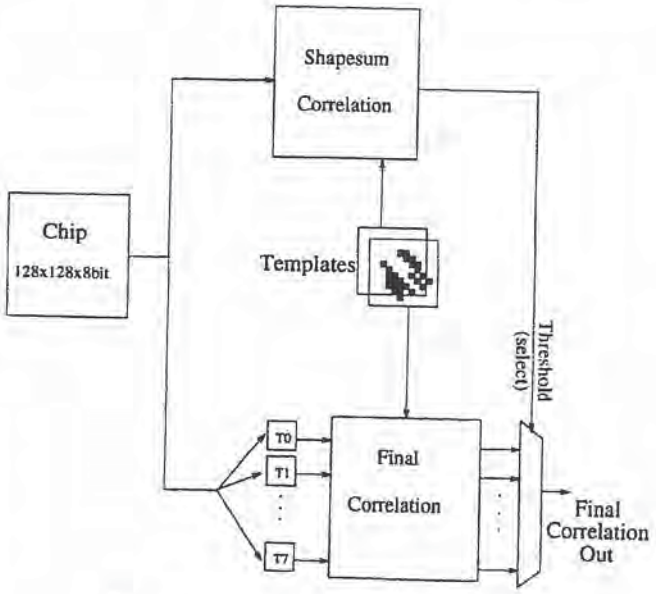


Figure 4: Sandia's Implementation.

3.2 UCLA

A group at UCLA (Headed by John Villasenor) [3] is working on a FPGA based SLD implementation. By doing bit level correlations they are able to do very compact adder trees that take advantage of template sparseness and FPGA on board lookup-table memory capability. Their approach compiles template information directly into the hardware and relies on fast reconfiguration to switch template information. They

also try and take advantage of template overlap by computing the results of multiple correlations simultaneously.

4 Chunky-SLD on Splash 2

On SPLASH 2 a static implementation was done to avoid the overhead of reconfiguring the hardware during execution. In order to reduce the hardware requirements without reducing throughput a deeply pipelined design (~400 cycle latency) was implemented. The Shapesum unit generates the threshold value which is then used to generate the final correlation values. (Note: There is a unique threshold for each offset). By doing this only two final correlations have to be computed per offset (one B_{sum} and one S_{sum}).

The Sandia implementation computes the Shapesum and final correlation in parallel which forces them to compute multiple final correlations. While our implementation does them serially. This allows us to use an exact threshold value. Also only one final correlation needs to be computed because the threshold value is computed before the final correlation begins. The technique used was to look at the correlations by column, compute the partial correlation for that column, and sum up the partial sums for all 16 columns. In this method 16 different column correlations are going on in parallel but only one column of data needs to be available for processing.

4.1 Implementing the Correlation as Column Sums

Figure 5 depicts a simple example that demonstrates a correlation of a 3×3 template with a binary image. Each row in the table represents one clock cycle. The first column is the clock cycle number. Corresponding numbers are found in the *Pixel load order* box at the right. A new pixel is brought in on each clock cycle. The P1, P2, and P3 columns represent the three column processing units needed for a three column template. The last column represents the actual output. Clock cycles 9 through 12 have been expanded to show how data (pixels and partial sums) are passed from column to column and illustrate the data format. Once the pipeline is full, a new correlation value is computed as each column arrives (three pixels/cycles).

Note that valid output comes every three cycles because the template is three rows tall. All processing elements are actively processing 3 pixel values at all times. The SPLASH 2 implementation works just like the example except for the size of the columns (16 pixels instead of three) and the data format (eight-bit instead of one-bit).

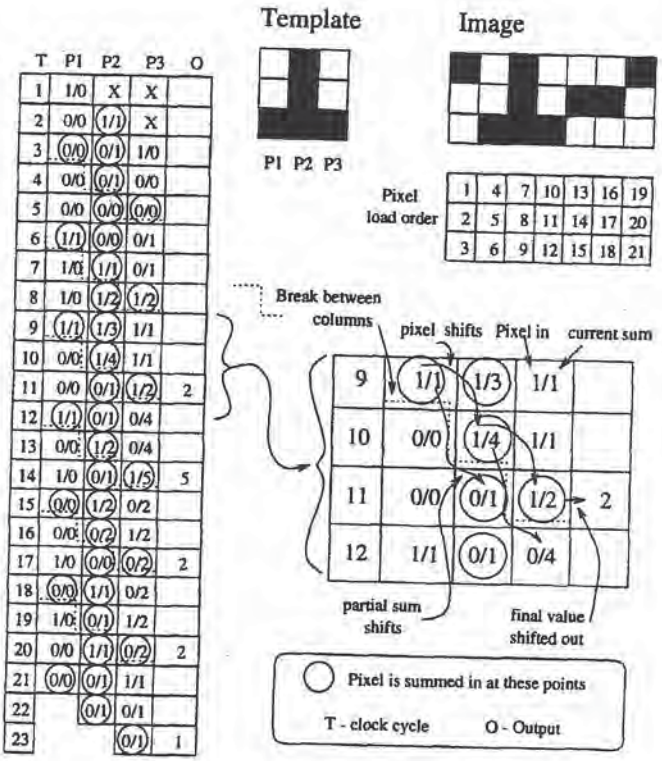


Figure 5: Example Column Sum.

4.2 Platforms/Implementations

Chunky SLD was implemented on the SPLASH 2 board. SPLASH 2 has shown itself to be a useful platform and has had numerous applications mapped to it [4, 5, 6, 7, 8, 9, 10, 11]. The implementation was done in VHDL and simulated/synthesized using Synopsis. All place and route was done automatically using Xilinx place and route tools.

One of the goals of the implementation was to run the system so that it consumed a pixel per cycle. This means that each cycle all processing elements (PE) need to be able to process a new pixel. This implementation follows Sandia National Labs algorithms (not implementation) as closely as possible (see Section 2.4).

The SPLASH 2 board was developed by SRC (Supercomputing Research Center Institute for Defense Analyses) [12]. The SPLASH 2 board is a linear systolic array of processing elements (FPGAs), each with their own memory.

4.2.1 Splash 2 Hardware

From a conceptual point of view, the SPLASH 2 system consists of a linear array of processing elements. This

makes SPLASH 2 a good candidate for linear-systolic applications with limited neighbor-to-neighbor interconnect. Because of limited routing resources SPLASH 2 has difficulty implementing multi-chip systems that are not linear systolic, though they are possible [8].

The actual SPLASH 2 platform consists of a board with 16 Xilinx 4010 chips (plus one for control) arranged in a linear systolic array. Each chip has a limited 36-bit connection to its two nearest neighbors. Each Xilinx 4010 is connected to a 512 kbyte memory

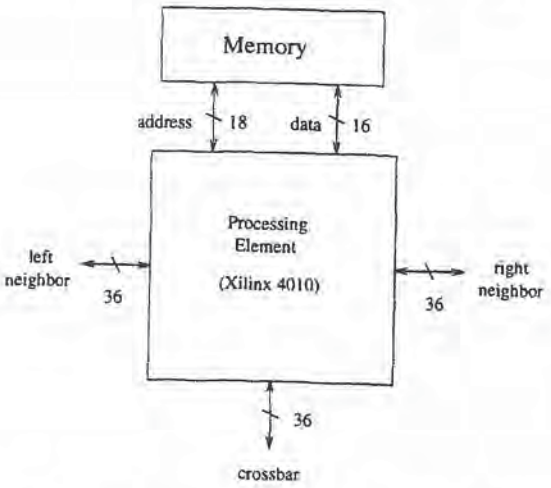


Figure 6: Single Processing Element of SPLASH 2.

(16-bit word size). The memory can handle back-to-back reads, or back-to-back writes, but requires one 'dead' (or turn around) cycle when changing from write to read. There is also a crossbar connected to all of the chips that allows some level of random connection between chips. Up to 16 boards can be daisy-chained together to provide a large linear-systolic array of 256 elements.

4.3 ATR Implementation on Splash 2

Similar to the example, the SPLASH 2 implementation processes one pixel at a time and loads them in column order so that the partial sums can be generated and passed from column to column. All template data are stored in the memories adjacent to the FPGAs on the SPLASH 2 boards. Each memory can hold several thousand templates thus making it possible to store all of the templates for a single class (5760) on a single SPLASH 2 board. There is sufficient room in the FPGA design to store a single template. The templates are switched by reading the new template data out of the memory and storing it within the FPGA. However, because this implementation is deeply pipelined, it is necessary to flush all current

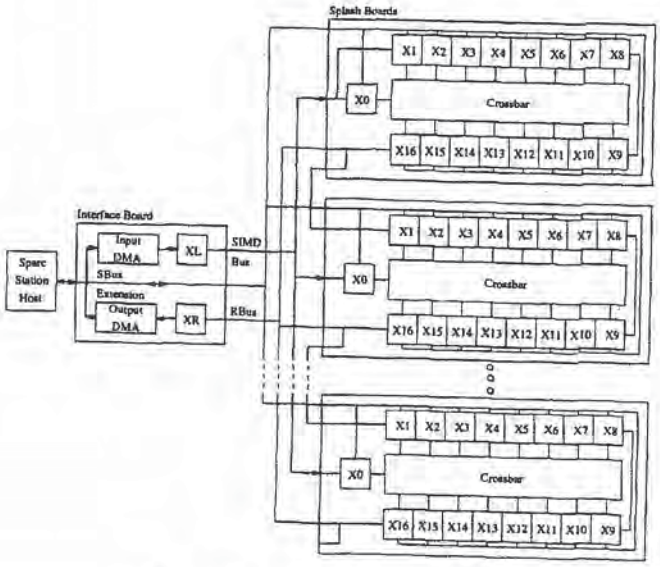


Figure 7: SPLASH 2 platform.

data from the system when switching to a new template. The overhead from the flushing operation is minimal ($\frac{318\ flush\ cycles}{231424\ compute\ cycles} =$) 0.14%.

During each clock cycle, a new pixel arrives at the FPGA. If the template bit corresponding to this pixel is *on* then the incoming pixel is added to the current partial sum. Each 16 clock cycles, this partial sum is then passed on to the next column and a new partial sum is received from the previous column. The last column computes a complete Shapesum every 16 cycles (one column). The final correlation of the Bright and Surround templates with the thresholded chip data works similarly except there are two correlations (one for each template).

Intermediate hit values are stored in a table, referred to as the *hit-table*, in one of the local memories. Each location in the table corresponds to an x-y offset of a chip, the origin of a single correlation. For each offset, if a chunk "hits", then the corresponding location in this table is incremented. Thus the table contains the accumulated hit values for all chunks and all offsets that have been computed to that point.

Hits are computed according to Equation 1. First, each B_{sum} and S_{sum} value is compared to its corresponding minimum value. Second, the threshold value corresponding to each B_{sum} and S_{sum} is checked to see if it is between a certain minimum and maximum value. For reasons of efficiency, the threshold value is actually examined earlier in the process and a zero for the threshold is stored in lookup-table memory if it is out of bounds. This works correctly because if the

threshold is zero, it will cause the B_{sum} to be zero, which will in turn cause the B_{sum} comparison to fail. Otherwise, if all three of these tests come back true then a hit has been found for the corresponding offset (see Equation 1) and the corresponding location in the hit-table is incremented. After the 40 templates are tested against the same chip the two offsets with the highest accumulated hit values are written into memory where the host computer can read them. This is accomplished by examining the hit values during this process and retaining the top two values in special memory locations. These final two hit values (which represent the top two orientations for a specific class) are used in the FI step.

For the SPLASH 2 board, as with most FPGA systems, partitioning is a major issue. The design needed to be modular so that different design modules could be reassigned to different FPGAs as necessary. This is where the column modules were so valuable (see Figure 8).

4.4 Special Features

This implementation has several notable characteristics. They include control distribution, modular design for partitioning and memory utilization.

4.4.1 Distributed Control

The control in this system is distributed throughout the array. Each column module has it's own state machine based control. Module synchronization is achieved by distributing a control token through the pipeline along with the data. When a module receives this signal, it resets its internal state machines and retrieves template data from its local memory. A memory controller resides in each processing element (FPGA) to retrieve template data and give memory access to all other modules.

4.4.2 Modular Design (Design for Partitioning)

Each column in both the Shapesum and final correlation use totally self contained modules that can be easily migrated from processing element to processing element. This was done to simplify the partitioning onto SPLASH 2 [13]. Memory data had to be carefully partitioned as well so that the data could follow the module to which it applied. The regularity of the design was an important concern; it allowed the placement of specific circuit modules to be dictated by the requirements of the algorithm and not by the limited interconnect of the platform. There are 16 identical modules in the Shapesum and 16 more identical modules in the final correlation. Along with these there is a divide, a hit accumulator and 2 delay modules (see Figure 8).

4.4.3 Memory Usage

The memories in SPLASH 2 serve several purposes. The template information is stored in them. They are used to implement video shift registers that correct for the latency incurred during threshold computation. These shift registers require that two memories be used in tandem because every clock cycle a new pixel (8 bits) had to be written to memory and a delayed pixel had to be read from memory. The bandwidth of one memory was such that it can handle two pixels (load and store) every three cycles. Thus one memory would delay two pixels and skip two pixels, while the other memory would delay the two pixels that the first memory skipped and skip the two pixels that the first memory delayed. The divide unit and the final result including the accumulated hit values are also stored in memory.

4.5 Performance Metrics

There are many metrics that could be used to measure the value of this implementation. This section is devoted to discussing some of these metrics.

4.5.1 Performance

This implementation runs at a frequency of 19 MHz using a test template that tests all column modules. `Xdelay` (a Xilinx timing tool) reports a guaranteed frequency of 13.2 MHz. Designs that will run in the 10 to 20 MHz range are typical [8, 10].

Using the above frequency a single system could process one orientation every .487 seconds (.701 seconds using a 13.2 MHz clock).

$$\frac{128\ col \times (128-15)\ row \times 16\frac{pix}{col}}{19\ MHz} \times 40\frac{Chunks}{Orient} = (12\ ms/Chunk) \times 40\frac{Chunks}{Orientation} = \boxed{487\text{ ms @ 19 MHz}}$$

or

$$\frac{128\ col \times (128-15)\ row \times 16\frac{pix}{col}}{13.2\ MHz} \times 40\frac{Chunks}{Orient} = (17.5\ ms/Chunk) \times 40\frac{Chunks}{Orientation} = \boxed{701\text{ ms @ 13.2 MHz}}$$

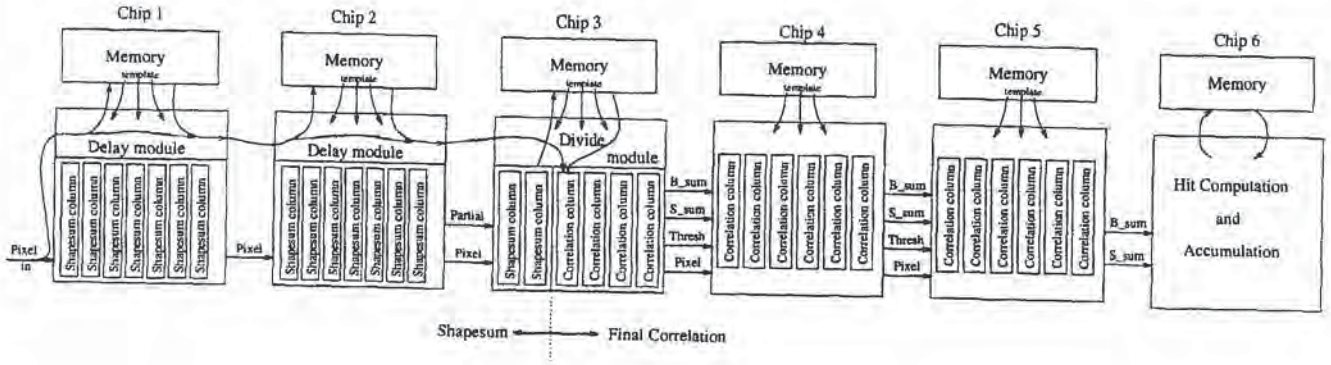


Figure 8: SPLASH 2 Implementation.

Using a brute force approach it takes 59 seconds to compute the same data on an HP 770 (110 MHz). That is two orders of magnitude slower. On one SPLASH 2 board two processing units can be installed yielding the ability to process 1 orientation (40 chunks) every .244 seconds (.350 seconds at 13.2 MHz).

4.5.2 Multi-board Splash 2

If a 16-board system were used then 256 processing elements would be available. this would make 42 processing units (256 PEs ÷ 6 PE per PU) and 42× the throughput. Thus one orientation can be processed every 11.6 ms (16.7 ms at 13.2 MHz). This is equal to covering 86 orientation per second (60 orientations per second at 13.2 MHz).

4.5.3 Memory Bandwidth

The distributed memory and aggregate memory bandwidth is what makes this implementation possible. Because the memory is distributed to each chip this implementation can be extended to a multi-board system quite easily. Consider that a processing unit consists of 6 FPGAs and their corresponding memories. Each processing unit performs 10 reads and 13 writes each 16 clock cycles. The SPLASH 2 memories require 3 clock cycles to perform a read and a write for a total available bandwidth of 2 reads and 2 writes every cycle ($\frac{6\ memories}{3\ cycles}$). Thus a processing unit consumes approximately 1/2 of the available memory bandwidth. If there were only two or three memories available for the six-chip system this implementation would not be possible.

4.5.4 Memory stored templates

The decision to use statically configured hardware and memory-stored templates was made early in the design process. Another approach would be to use hard-coded templates, e.g., hardware with the template data compiled in as constants. A hard-coded template implementation could possibly use less hardware by removing unused accumulators and template reading logic, however it would require the hardware to be reconfigured during run-time in order to switch templates. This would take significantly longer than the memory read done in the memory-stored template version. The SPLASH 2 system requires approximately 17 ms per chip to configure. The system would have to be reconfigured each time a new template was used. In order to process one orientation the system would have to be configured 40 different times. This would adversely impact system throughput. In fact even if the system could be reconfigured in one cycle the performance wouldn't improve because the time spent reading templates from memory is hidden while the pipeline is initially filling (during the flush cycles). There are other advantages as well. Design time is simplified because new hardware bit streams don't have to be generated and tested to use new templates. Re-compilation may improve hardware utilization, however that would come at a cost of repartitioning difficulty.

5 Future work

In the future, additional related research will be conducted on possible improvements to the implementation described in this paper. In addition, alternative implementations that are not currently possible with the SPLASH 2 platform are under examination. Finally, work is underway on a bit-serial version of this implementation.

5.1 Splash 2 Improvements

Although SPLASH 2 was suitable for this project, there are many possible architectural improvements that could be made that would increase performance significantly. A faster memory interface that supports a two-cycle read-write operation or perhaps a dual-ported memory interface would provide improved memory bandwidth. A larger processing element would give more resources and make implementations more flexible. Updated VHDL libraries that conform to standard data types would help design portability and simplicity.

If chip-to-chip I/O were increased on SPLASH 2 a full column could be processed each clock cycle. This would require at least 339 I/O pins ([128 pixel bits + 13 bit partial sum + 8 bit threshold + 2 bits of control] × 2 (in and out) + 16-bit memory data + 21-bit memory address = 339 pins). The current chips (Xilinx 4010) has enough logic and routing to process an entire column at one time. If this kind of I/O bandwidth were available [14] the system would consume 2 SPLASH 2 boards but would have 16 times the throughput because 16 times as much data would be processed each clock cycle.

Another option would be to distribute the *chip* image to each FPGA memory and have a wide memory data path. This would require at least 195 I/O pins ([13 bit partial sum + 8 bit threshold + 2 bit control] × 2 (in and out) + 128 bit memory data path + 21 bit memory address = 195) but would have the same throughput.

5.1.1 Interface Improvements

The current implementation assumes that all data are loaded on the SPLASH 2 board before processing commences. A more realistic implementation would stream data to the system as it is produced. Thus, DMA channels or other high-performance I/O hardware, if added to the system, would make performance more sustainable across multiple images.

5.2 Other implementations

There are several ways this algorithm could be implemented. The most obvious way is to to a full parallel version that looks at a full 16×16 pixels at a time. Another is a linear systolic version that looks at one pixel at a time. A linear systolic implementation was designed because SPLASH 2 is organized that way.

This implementation takes a relatively brute-force approach and feeds the entire chip into the system for processing. Whether or not a given pixel actually contributes to the calculation is determined after the image data are already in the pipeline. Other approaches may take a more selective approach and only access image pixels that actually contribute to the calculation. This may reduce overall memory bandwidth requirements and suggest other implementation strategies. An implementation that only examined pixels that will be used could potentially give a performance increase when correlating the Bright template because the majority of pixels in the template represent chip locations that do not contribute to the computation. However, such an approach would require a complicated control scheme and may not map well onto SPLASH 2.

Here at BYU there is ongoing work targeting 3 different FPGA architectures, each looking to better understand the hardware and ATR algorithms. These include a bit-serial version targeted at National Semiconductors Clay FPGA and a Xilinx 6200. Another project is looking at a full parallel implementation targeting an Altera Flex-10K part.

6 Conclusion

SPLASH 2 is organized as a linear array of FPGA chips each connected to a separate memory. Because of this, applications can only make effective use of SPLASH 2 resources if (1) individual circuit modules are small enough to easily fit into the limited resources available within a single FPGA (Xilinx 4010) and (2) circuit modules can be grouped together in such a way that the interconnect requirements between modules on separate chips will fit in the limited chip-to-chip interconnect available on SPLASH 2.

Both of these requirements were met in this application by carefully organizing all correlations (both Shapesum and final correlation) as independent column-oriented processing elements (PEs). Each PE occupied only a small amount of FPGA resources thereby making it possible to place several PEs into a single FPGA. PEs that are integrated on a single FPGA communicate via the available on-chip FPGA routing; PEs that are located on different FPGAs can easily communicate via the limited inter-chip routing on SPLASH 2.

This repetitive, column-oriented organization also eases the programming of SPLASH 2 considerably. Much of the VHDL code was reused and because the fourth and fifth chips are identical (because of the linear column layout, see Figure 8) it was possible to down-load the same bit-stream into both chips.

This column organization presents a good tradeoff between a fully parallel design and a design that fits within the restrictions of the SPLASH 2 architecture. Each column operates in parallel computing a column

sum that when complete is passed onto its neighboring column. While it is possible to achieve higher levels of parallelism, it is likely that such designs will require denser interconnect than SPLASH 2 is capable of providing (see Section 5.1). By limiting communication to only adjacent columns, significant parallelism is still achieved with only moderate amounts of interconnect.

This implementation can also take advantage of SPLASH 2's extensibility giving it even higher performance capability. Simply by passing the pixel data from chip 6 (see Figure 8) to the next chip on the SPLASH 2 board and configuring the next 6 chips in the array with the same bit stream as the first 6 chips the throughput on the system would be doubled. This extensibility can also be expanded to multiple boards giving a linear increase in system throughput.

References

[1] S. Trimberger. A reprogrammable gate array and applications. *Proceedings of the IEEE*, pages 1030–1041, July 1993.

[2] National Semiconductor. *Configurable Logic Array (CLAy).* National Semicondutor, 1993.

[3] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79, Napa, CA, April 1996.

[4] A. L. Abbott, P. M. Athanas, L. Chen, and R. L. Elliott. Finding lines and building pyramids with Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 155–161, Napa, CA, April 1994.

[5] J. M. Arnold. The Splash 2 software environment. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 88–93, Napa, CA, April 1993.

[6] P. Athanas and A. Abbott. Image processing on a custom computing platform. In R. Hartenstein and M. Z. Servit, editors, *Field-Programmable Logic: Architectures, Synthesis and Applications. 4th International Workshop on Field-Programmable Logic and Applications*, pages 156–167, Prague, Czech Republic, September 1994. Springer-Verlag.

[7] M. Gokhale and B. Schott. Data parallel C on a reconfigurable logic array. *Journal of Supercomputing*, 9(3):291–313, 1994.

[8] P. Graham and B. Nelson. A hardware genetic algorithm for the travelling salesman problem on SPLASH 2. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 352–361, Oxford, England, August 1995. Springer.

[9] D. T. Hoang. Searching genetic databases on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Napa, CA, April 1993.

[10] D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–177, Napa, CA, April 1993.

[11] N. Ratha, A. Jain, and D. Rover. Convolution on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 204–213, Napa, CA, April 1995.

[12] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–324, June 1992.

[13] Benjamin W. Wah. Systolic arrays–from concept to implementation. *IEEE Journal Computer*, 20(7):12–17, July 1987.

[14] Altera. *Altera–Embedded Programmable Logic Family–Flex 10K Data Sheet.* Altera, 1995.

Attachment 2C

« Return to Search Results

Cite/Export Print E-mail Share Permalink

Add to list Add tags Write a review Rate this item: 1 2 3 4 5

Proceedings, the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, April 16-18, 1997, Napa Valley, California

Author: Kenneth L Pocek; Jeffrey M Arnold; IEEE Computer Society. Technical Committee on Computer Architecture.

Publisher: Los Alamitos, Calif. : IEEE Computer Society Press, ©1997.

Edition/Format: Print book : Conference publication : English View all editions and formats

Summary: This symposium on computer hardware and design and testing is aimed at computing professionals. It looks at device architecture, communication applications, performance, software tools, and image Read more...

Rating: (not yet rated) 0 with reviews - Be the first.

Subjects
Field programmable gate arrays -- Congresses.
Computer engineering -- Congresses.
Computer engineering.
View all subjects

More like this Similar Items

Get a Copy

Find a copy in the library

Find a copy online

Links to this item

IEEE Xplore

Find a copy in the library

Enter your location: boston spa Find libraries
Submit a complete postal address for best results.

Displaying libraries 1-6 out of 207 for all 8 editions (Boston Spa, Wetherby LS23, UK)

Show libraries holding just this edition

« First ‹ Prev 1 2 3 Next › Last »

	Library	Held formats	Distance	
1.	The British Library, On Demand **The British Library, Document Supply, Boston Spa** Wetherby, West Yorkshire, LS23 7BQ United	Book	2 miles MAP IT	Library info Add to favorites

Kingdom

2.	**University of Sheffield** **Sheffield University** Sheffield, S10 2TN United Kingdom	Book	37 miles MAP IT	Library info Ask a librarian Add to favorites
3.	**University of Liverpool, Sydney Jones Library** Liverpool, L69 3DA United Kingdom	Book	75 miles MAP IT	Library info Ask a librarian Add to favorites
4.	**University of Gloucestershire** **Francis Close Hall Campus Library, Oxstalls Campus Library, Park Campus Library** Cheltenham, GL50 2RH United Kingdom	Book	143 miles MAP IT	Library info Ask a librarian Add to favorites
5.	**University of Essex** **The Albert Sloman Library** Colchester, CO4 3SQ United Kingdom	Book	170 miles MAP IT	Library info Add to favorites
6.	**London Metropolitan University** London, N7 8DB United Kingdom	Book	171 miles MAP IT	Library info Search at this library Add to favorites

« First ‹ Prev 1 2 3 Next › Last »

Details

Genre/Form:	Conference papers and proceedings Congresses
Material Type:	Conference publication, Internet resource
Document Type:	Book, Internet Resource
All Authors / Contributors:	Kenneth L Pocek; Jeffrey M Arnold; IEEE Computer Society. Technical Committee on Computer Architecture. Find more information about: Kenneth L Pocek Go
ISBN:	0818681594 9780818681592 0818681608 9780818681608 0818681616 9780818681615
OCLC Number:	37949175
Notes:	"IEEE Computer Society order number PR08159"--Title page verso. "IEEE order plan catalog number 97TB100186"--Title page verso.
Description:	x, 250 pages : illustrations ; 28 cm
Other Titles:	FPGAs for Custom Computing Machines, 1997, proceedings, The 5th Annual IEEE Symposium on. FCCM'97 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Fifth Annual IEEE Symposium on Field-Programmable Custom Computing Machines IEEE Symposium on FPGAs for Custom Computing Machines FPGAs for Custom Computing Machines
Responsibility:	sponsored by the IEEE Computer Society, IEEE Computer Society Technical Committee on Computer Architecture ; [edited by Kenneth L. Pocek and Jeffrey Arnold].

Reviews

User-contributed reviews

Add a review and share your thoughts with other readers. Be the first.

Tags

Add tags for "Proceedings, the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, April 16-18, 1997, Napa Valley, California". Be the first.

Similar Items

Related Subjects: (4)

- Field programmable gate arrays -- Congresses.
- Computer engineering -- Congresses.
- Computer engineering.
- Field programmable gate arrays.

Linked Data

Attachment 2D

Close this window to return to the catalogue

EXPLORE THE BRITISH LIBRARY

Item Details

FMT	BK
LDR	caa a22003133u 4500
001	006410603
008	971111s1997 xxu || 100 ||eng
020	|a 0818681594 (pbk)
020	|a 0818681608 (casebound)
020	|a 0818681616 (microfiche)
040	|a Uk |c Uk
CNF	|a Field-programmable custom computing machines (Annual symposium) |n (5th : |d 1997 Apr : |c Napa Valley, CA)
24514	|a The 5th Annual IEEE symposium on field-programmable custom computing machines.
260	|b IEEE Computer Society Press, |c 1997.
336	|a text |2 rdacontent
337	|a unmediated |2 rdamedia
338	|a volume |2 rdacarrier
653	|a FPGAs
653	|a custom computing machines
653	|a IEEE
653	|a computer architecture
653	|a FCCM
653	|a field-programmable custom computing machines
7001	|a Pocek, Kenneth L., |e Ed.
7001	|a Arnold, Jeffrey, |e Ed.
7102	|a IEEE Computer Society. |b Technical Committee on Computer Architecture.
945	|a IEEE symposium on FPGAs for custom computing machines, ISSN 1082-3409 ; 5th
85241	|a British Library |b DSC |j 4363.086450 |i 5th 1997
SYS	006410603

P1

The 5th Annual IEEE symposium on field-programmable custom computing machines.

IEEE Computer Society Press, 1997.

Title: The 5th Annual IEEE symposium on field-programmable custom computing machines.

Contributor: Kenneth L. Pocek Ed.;
Jeffrey Arnold Ed.;
IEEE Computer Society. Technical Committee on Computer Architecture.

Subjects: FPGAs; custom computing machines; IEEE; computer architecture; FCCM; field-programmable custom computing machines

Rights: Terms governing use: Current copyright fee: GBP23.57

Publication Details: IEEE Computer Society Press, 1997.

Language: English

Identifier: ISBN 0818681594 (pbk); System number: 006410603; ISBN 0818681608 (casebound); System number: 006410603; ISBN 0818681616 (microfiche); System number: 006410603

Shelfmark(s): Document Supply 4363.086450 5th 1997

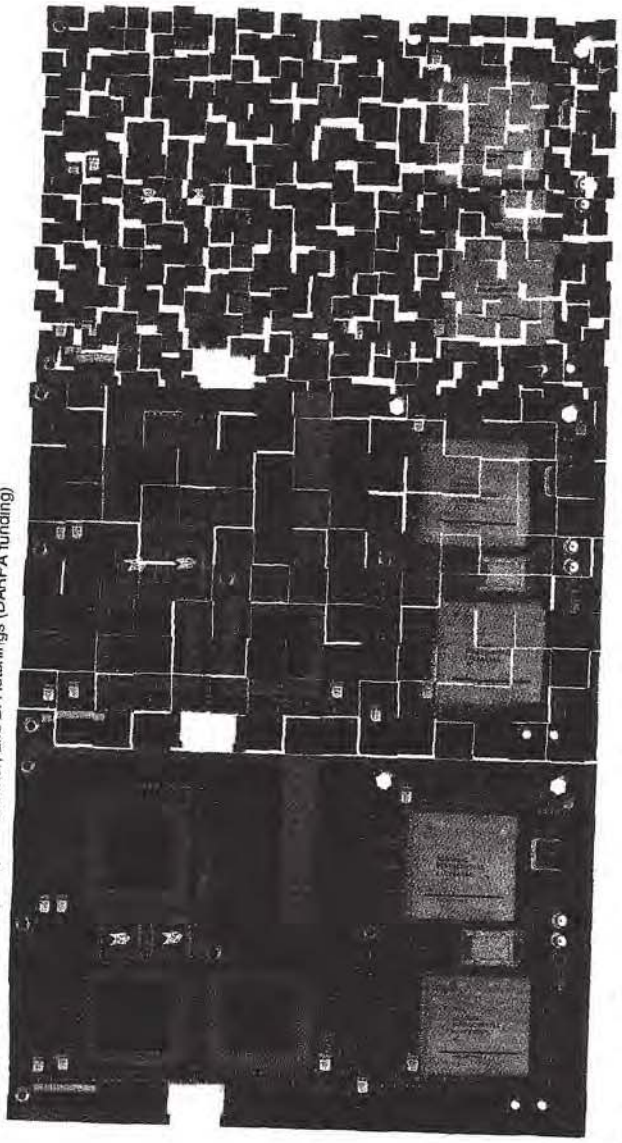
UIN: BLL01006410603

P1

Attachment 2E

John Villasenor and Brad Hutchings

The Flexibility of CONFIGURABLE COMPUTING



Board courtesy of Hea Joung Kim, J. Villasenor, and B. Hutchings (DARPA funding)

Providing the Hardware for Data-Intensive Real-Time Processing

One of the most fundamental tradeoffs in the design of computing devices involves the balance between flexibility and efficiency. At one end of the spectrum are application-specific integrated circuits (ASICs), which contain dedicated circuitry optimized to a particular set of tasks. ASICs have the advantages of low power dissipation and high clock speeds, but suffer the drawback of being able to perform only the tasks for which they were designed, and thus become obsolete if the task is changed. Because ASICs are highly specialized to a given application, they typically achieve the highest possible performance at the lowest silicon cost. However, this specialization comes at the cost of flexibility and ASICs are only useful for the one task for which they were designed. At the other end of the spectrum are programmable processors such as microprocessors or DSPs. Programmable processors implement a limited and *fixed* set of arithmetic and control operations that can be organized and sequenced to implement any arbitrary computation. However, in cases where the native processor operations are not well suited to the task at hand, or in cases where massive amounts of parallelism can be exploited, programmable processors are inefficient and deliver relatively poor performance.

There has been growing recent interest in configurable computing, which can be viewed as a hybrid between ASICs and programmable processors. Configurable computing machines are implemented with *programmable logic*: flexible hardware that can be programmed or, more correctly, structured to fit the natural organization and dataflow of a computation. Unlike programmable processors where computations must be implemented as some sequence of available operations, configurable com-

IEEE SIGNAL PROCESSING MAGAZINE
1053-5888/98/$10.00©1998IEEE

puting machines allow a much more direct and natural approach in which the hardware can be structured to directly implement the native operations required by the *application* and also organized to exploit the concurrency inherent in the computation. These systems can "track" the computational requirements imposed by changes in the processing task or data.

The enabling device for configurable computing is the field-programmable gate array (FPGA). FPGAs have typically been used for prototyping of ASIC designs and as low nonrecurring engineering (NRE) cost parts in applications where the cost of designing and fabricating an ASIC is not justified. FPGAs have not traditionally been viewed as effective computing devices in their own right, mostly because of their higher power requirements, low clock speed, and relatively long (tens of milliseconds) programming time. However, the combination of technology advances and some recent demonstrations of FPGA-based computing machines has led to a re-evaluation of FPGAs as efficient computing devices. It is now clear that for applications characterized by deeply pipelined, highly parallel, and integer arithmetic processing, configurable computing machines can outperform alternative solutions by up to an order of magnitude for metrics such as cost and computation speed. Computational tasks having some or all of these characteristics include pattern matching, image processing, target recognition, cryptography, and some database tasks. Because there are many other applications that require computations that FPGAs do not perform as well, configurable computing today is a niche solution that appears unlikely to make significant inroads into the general-purpose computing applications currently dominated by microprocessors. However, the advantages of configurable computing for these niche applications are quite compelling. The combination in a single device of dedicated hardware and rapid, submillisecond-scale reprogrammability constitutes an exciting and promising development whose implications are only just beginning to be explored.

In this article we begin with a brief tutorial on FPGAs that describes the most common FPGA architectures and how these architectures are used to support computation, memory access, and data flow. We then present FPGAs as computing machines and focus on devices that are reconfigured during run time. Ongoing research involving FPGAs as well as future directions are also discussed.

Field-Programmable Gate Arrays

Architecture

FPGAs share with ASICS the capability to support application-specific circuitry, with the key difference that FPGA circuits are programmed by means of a bitstream that completely specifies the logical functions and connectivity to be implemented. Because FPGAs are static-random-access-memory (SRAM) devices, they can be reprogrammed as often as desired, thereby allowing the silicon resources in a single device to be time shared across a wide range of functions. The flexibility of programmable hardware inevitably comes at a cost in efficiency relative to an ASIC. Therefore, while there is some truth to the statement that FPGAs combine the advantages of ASICs with the flexibility of software-programmable processors, the caveat must be added that an FPGA never achieves the power, clock rate, or die size that could be realized in a full custom chip optimized for a particular task.

Although there is a wide range of architectural approaches that are used in FPGAs, all FPGAs consist of an array of logical units distributed across a grid of programmable interconnect, or routing. For a given total die size, the basic design issues facing an FPGA manufacturer are much the same as those involved in the design of any other computing device. These issues include the amount of resources to devote to routing as opposed to computation; the amount, granularity, and distribution of on-chip memory; and the quantity and accessibility of I/O.

Broadly speaking, FPGAs can be classified as either "coarse-grained," meaning that they possess a smaller number of relatively powerful logical units, or "fine-grained," which refers to a larger number of very elementary logic units. Most FPGAs in current use are coarse-grained, and have logical units that are based on look-up tables (LUTs). For example, the Xilinx 4000 series family of FPGAs has logical units (referred to by Xilinx as "configurable logic blocks") that each contain two 4-input, 1-output LUTs, one 3-input, 1-output LUT, two flip flops, and several multiplexors to select from among the lookup table and flip-flop inputs and outputs. One Xilinx 4000 series configurable logic block has the capacity to implement a two-bit adder or a nine-bit parity checker. By contrast, the Xilinx 6200 series provides an example of a device that lies at the other end of the granularity spectrum. Each logical unit in the 6200 consists of two 4-to-1 multiplexors (though not all of the inputs are independent), three 2-to-1 multiplexors and a flip-flop, and can implement any one- or two-input function as well as a single bit of storage.

To describe an FPGA in terms of its gate count can be broadly useful but also deceptive.

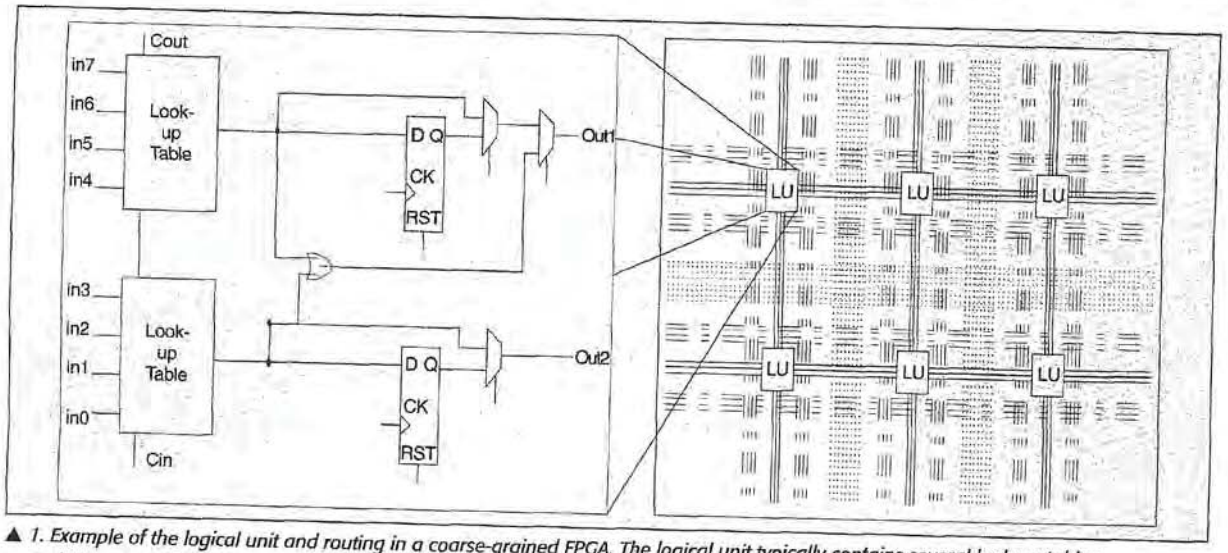
Coarse-grained FPGA architectures also provide some form of distributed memory in addition to logic resources. For example, each LUT in a Xilinx 4K-series FPGA can be used to implement a 16×1 RAM/ROM. This makes it possible to trade logic for memory and vice

versa. FPGAs manufactured by Altera provide dedicated RAM blocks on the 10K-series that can each function as either a 2048×1, 1024×2, 512×4, or 256×8 RAM. Altera RAMs are larger than the Xilinx LUT-based RAMs and are more efficient for implementing larger memories but may go unused in designs that don't need them. It is anticipated that vendors of fine-grained FPGAs will eventually introduce devices that provide RAM blocks, but there are no current fine-grained FPGAs that provide distributed RAM resources (though there is storage capacity in the flip flops that are part of each logical unit in a fine-grained FPGA).

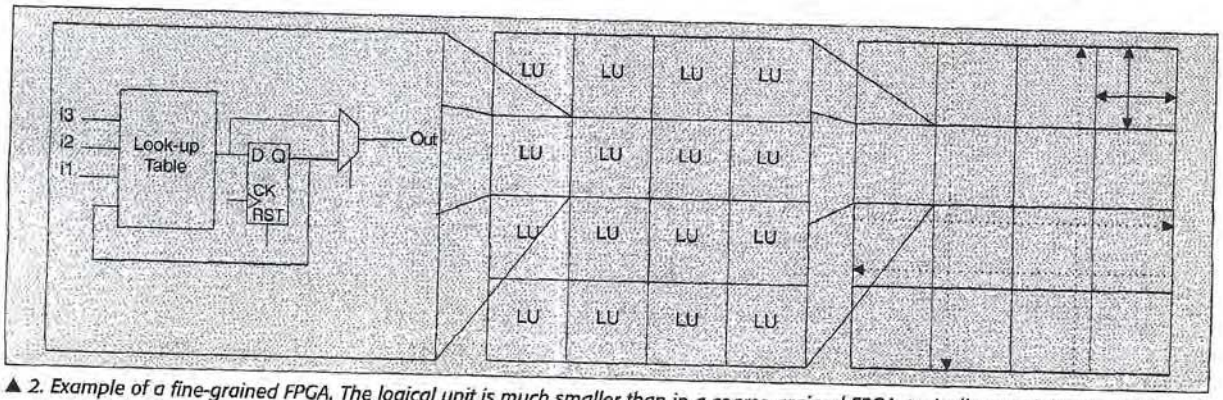
LUTs, memory blocks, and flip-flops are all interconnected via programmable routing. Programmable routing consists of fixed-length metal segments that are interconnected with programmable switches, e.g., a single MOSFET. Metal segments are of various lengths that support nearest-neighbor interconnect, global interconnect, and several lengths in between. Connections are made by programming the switches that interconnect the metal segments between the desired pins. The additional resistance and capacitance introduced by the programmable routing network is one of the primary performance bottlenecks in FPGAs. Current-generation FPGAs are typically clocked at between 10 and 40 MHz, though certain types of algorithms can be optimized to run as fast as 250 MHz [23]. Figures 1 and 2 illustrate the routing and logical unit of a coarse-grained and fine-grained FPGA, respectively.

Logic Capacity

FPGA capacity is often expressed in terms of logic gates, where one logic gate is understood to be a two-input NAND gate. As in other types of computing, in which the pressure placed on manufacturers to characterize performance using a single number has led to oversimplification, to describe an FPGA in terms of its gate count can be broadly useful but also deceptive. As is evident from the



▲ *1. Example of the logical unit and routing in a coarse-grained FPGA. The logical unit typically contains several look-up tables, multiplexors, and flip-flops. Connectivity between logical units is hierarchical, including routing between adjacent logical units (solid lines), among logical units in a local neighborhood (dashed lines), and routing to connect widely separated logical units (dotted lines).*



▲ *2. Example of a fine-grained FPGA. The logical unit is much smaller than in a coarse-grained FPGA, typically containing the equivalent of a single, small look-up table and a memory element. Routing permits a high degree of connectivity among logical units in a local neighborhood as well as among widely separated logical units.*

contents of the logical units as described above, the logical functions that are realized in a given FPGA configuration will be highly dependent on computational demands and algorithm-mapping strategies associated with the application. In addition, within any given configuration, some logical units will be extremely efficiently utilized while others will receive little or no utilization. Another complication is that for LUT-based FPGAs, logical elements can be used for logic, memory, or a combination of both, thereby making it necessary to distinguish between logic gates and gates dedicated to storage.

For experienced users of FPGAs, a more meaningful way to characterize capacity is to use the number of vendor-specific logical units. Since vendors tend to scale parts by increasing the array size as opposed to increasing the complexity of the logical units that are elements in the array, a designer with experience using a part with 1,024 logical units will readily understand what it means if the vendor announces an FPGA with 1,600 logical units. In the absence of experience with FPGA designs, using logical units as a size metric becomes less meaningful. As a rough rule of thumb, each logical unit in Xilinx and Altera FPGAs corresponds to between 20 and 50 gates of logic. For example, the Xilinx 4062, which is an array of 48 by 48 logical units, will support, depending on the computations needed, designs having between 40K and 130K equivalent logic gates. Table 1 provides a summary of the principal architecture features for a wide range of presently available FPGAs.

FPGA-Based Computing Platforms

Since at least 1989 [7], FPGA-based computing platforms have demonstrated the potential for achieving extremely high performance for many tasks such as image filtering [36], convolution [5,7], morphology [13], feature extraction [1], and object tracking [32,39]. The potential is real; prototype systems have been constructed and applications developed that achieve performance that is an order of magnitude faster than conventional approaches [3].

Although a wide variety of FPGA-based computing systems have been constructed and reported in the literature, we focus here on two platforms that stand out as good examples of what can be achieved with FPGA-based computing platforms: DECPeRLE, which was built by researchers at Digital Equipment Corporation (DEC), and SPLASH-2, which was built at the Supercomputer Research Center/Institute for Defense Analysis. These two platforms are successful examples of FPGA-based systems: each had clearly defined architectural goals, integrated design and debug tools, and achieved significant speedup for several important applications.

Programmable Active Memory and the DECPeRLE Systems

Programmable active memory (PAM) is designed to act as a tightly coupled, general-purpose configurable hardware co-processor. The main idea—as with many FPGA-based systems—is to map the computationally intense parts of an application to the FPGAs and execute the remaining code on a host workstation. Designed and developed at the DEC Paris Research Laboratory (PRL), PAM can be conceptually viewed as a large array of bit-level functional units called programmable active memory cells. In practice PAMs are implemented with Xilinx 3K- and 4K-series FPGAs in a variety of configurations: DECPeRLE-0 (first generation), DECPeRLE-1 (second-generation), and Pamette, a smaller PCI-based system. DECPeRLe-1 was a sec-

Table 1. Comparison of resources in common FPGA architectures.

FPGA	Logical Unit Name	Routing Hierarchy	Memory	Largest Part (available)
Xilinx 4000 Series	Configurable logic block (CLB): Two 4-input LUTs with fast carry chain; One 3-input LUT; 2 Flip-flops	5-level routing: single (adjacent CLBs), double (2 CLBs), QUAD (4 CLBs), long (entire length of array) and global lines	Each CLB can be used as 32-bits of RAM	4012EXV: 64 x 64 CLB-array, 448 User I/O
Altera Flex 10K Series	Logic element (LE): 8 LEs make up 1 logic array block (LAB): One 4-input LUT with fast-carry chain; 1 flip-flop OR gate	3-level routing: Local (LE in one LAB), fast-track row and Column (between LABs), and global	Embedded array blocks (EABs), each contain 2K bits. From 3 to 12 EABs per chip depending on size	EPF10K250, has 1520 LABs, 470 max user I/O

ond-generation system and was the most successful of the DEC PAM systems; to the best of our knowledge more applications have been reported for DECPeRLE-1 than for any other single platform.

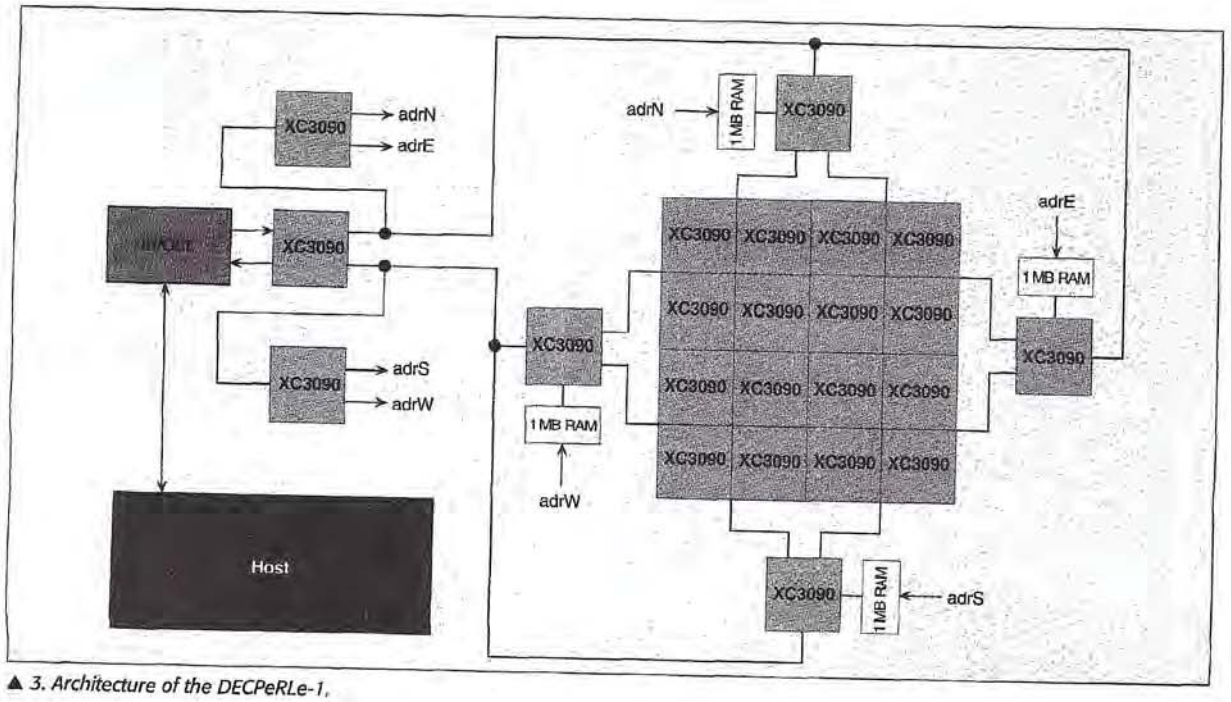
DECPeRLe-1 consists of 23 Xilinx 3090 FPGAs: 16 in a tightly coupled array of 4×4 FPGAs with the remaining FPGAs providing interfaces to RAM and a host machine, which is typically a DEC Alpha workstation [7]. In addition, a programmable clock generator and additional static logic is provided to manage the host bus interface and download process. Figure 3 shows the architecture of the DECPeRLe-1. As shown, the core array is connected to four additional 3090s, one on each side of the array, which are, in turn, connected to 1 MB (32-bits wide) of static RAM. Three additional FPGAs are used to interface the core to two FIFOs, which are connected to the host through a turbo-channel interface.

Although PAMs can be programmed with commercial CAD software, the PAM group developed custom design software for use with DECPeRLe. Referred to as PeRLe-1, the design tool is based on C++ and is a structural/physical design tool for capturing synchronous circuit designs. Standard C++ syntax is used to describe combinational logic. This is combined with a *NET* type and a register primitive, thus making it possible to describe any synchronous circuit with PeRLe-1.

Designers can also annotate their circuits with placement information that controls the final floorplan of the resulting circuit. Application development proceeds by first manually dividing the overall application into individual FPGA circuits. Each circuit is then described with a single PeRLe-1 program that, when compiled, linked to a technology library, and executed, generates a native netlist (Xilinx Netlist Format (XNF)) that can then be processed by Xilinx back-end software to route the design and to finally create the downloadable bitstream.

PeRLE-1 is a relatively low-level design tool that allows designers to finely craft high-performance designs. Unlike verilog-hardware-decription-language (VHDL) synthesis environments where structural implementations can be inferred from higher-level behavioral descriptions, PeRLe-1 requires the designer to describe a circuit strictly in structural terms and designers using PeRLe-1 often manually place significant portions of the circuitry to achieve the highest possible performance. The designers of PeRLe-1 consciously chose this design path because the synthesis tools available at the time when PeRLE-1 was under development were often inefficient for high-performance designs. This was a reasonable tradeoff since one of the goals of the PAM project was to outperform supercomputers.

DECPeRLE-1 also provides a rich run-time library and set of debugging tools that allows design debugging to take place on the actual hardware. The run-time environment supports single-step and various tracing modes that simulate for a single clock cycle and then retrieve FPGA internal flip-flop state data for further analysis. Signal values are accessible via the variable names that denoted them in the PeRLe-1 program as long as they are registered in flip-flops. In addition, a graphical tool, **showRB**, is available for analyzing readback traces. It can graphically playback the stored state traces as a simple animation at about 100 Hz [45].



▲ *3. Architecture of the DECPeRLe-1.*

DECPeRLe-1 achieved significant speedup for a wide range of application areas including RSA cryptography, molecular biology, heat and Laplace equations, neural networks, and image classification [45]. In most cases, reported applications were the fastest known implementations at that time (including ASICs and supercomputers) and typically achieved a 10x speedup over other existing solutions.

SPLASH-2

SPLASH-2 is another example of a well-known FPGA-based computing platform. Created by researchers at the Supercomputer Research Center (SRC), SPLASH-2 was specifically designed to support high-performance linear systolic applications. SPLASH-2 was also a second-generation system that was constructed to overcome I/O limitations and the lack of inter-FPGA communication that limited the general application of the SPLASH-1, which preceded it [4].

The SPLASH-2 system connects Xilinx 4010 FPGAs in a linear systolic array. In addition to the linear systolic data-path, a global broadcast data-path and user-defined crossbar data path are available for custom communication networks. The system scales well for linear-systolic applications because the array can be easily extended simply by adding additional boards. Each of the FPGAs is connected to a local 256K 16-bit memory that is also mapped into the address space of the host processor. FIFOs are placed at the input and output sides of the systolic pipeline to maintain high throughput and buffer data to the host processor, which is typically a Sun SPARC-II. Connections between neighboring FPGAs and the crossbar are 36-bits wide, enough to support 32-bit data and 4-bits of tag, and useful in pipelined applications. Figure 4 shows the organization of the SPLASH II system.

In contrast with DECPeRLe-1 and SPLASH-1, SPLASH-2 uses commercial VHDL synthesis tools for application development. A VHDL model of the complete system is provided that includes all board-level interconnect, memories, and the host interface. This body of VHDL code serves not only as the simulation environment for SPLASH-2 designs but also as the definitive documentation for the entire platform. Applications are developed by creating a single VHDL design for each of the FPGAs that will be used in the system. Designers typically simulate these designs in the context of the system VHDL model to verify that the circuitry will operate correctly when downloaded to the actual hardware. Typically, only the initial simulations are performed in the simulation environment; once the designer is satisfied that the design is close to being correct, the VHDL is typically synthesized and downloaded for further verification via the run-time debugging tools. In practice, the hardware almost always operates as predicted by simulations. When differences occur, it is typically caused by a bug in the synthesis tool or by the use of signal initializers, which are ignored by the synthesis tool.

A SPLASH-2 application is developed by first manually partitioning the design into FPGA-sized pieces and then coding the VHDL that describes the circuit that goes into each FPGA. This code is verified by simulation in the SPLASH-2 environment with all of the other user FPGA designs. Each VHDL file is then synthesized separately, placed and routed with the Xilinx backend software, and finally downloaded to the SPLASH-2 hardware. Breaking the design into FPGA-sized pieces is one of the challenges faced when designing for SPLASH-2 because it is impossible to know if a design will fit in an FPGA until it is synthesized and placed and routed. Thus, partitioning is typically an iterative affair with designers attempting 2-3 times to converge on a design that meets the needs of the application and fits within SPLASH-2 resources.

Reconfiguration costs time because the logic being reconfigured is unavailable for processing while the new bitstream is being downloaded.

Although the commercial CAD tools used with SPLASH-2 do not make as efficient use of silicon as the PeRLe-1 tools, they still allow design to proceed rapidly. They are also based on a commonly-used HDL language, thus giving users a choice of commercial simulation and synthesis environments. Although not recommended, it is possible to floorplan postsynthesis designs if absolutely necessary, but this tends to be extremely unwieldy with current synthesis tools.

SPLASH-2 also provides an excellent run-time environment that is similar to a typical software debugging tool with support for reading back flip-flop state information, single stepping, setting clock frequency, etc. Debugging is performed within T2, a TCL-based, textual debugging tool. After download and during execution, signal values that are registered in flip-flops are accessible via the VHDL signal names that were used in the original VHDL text.

SPLASH-1 and SPLASH-2 are perhaps best known for their high-performance implementation of the genetic edit-distance application. This application achieved over two orders of magnitude of speedup over existing supercomputer implementations [24, 28]. Peter Athanas of Virginia Polytechnic University modified SPLASH-2 so that it would accept data directly from a video camera and reported significant speedups for a wide variety of

image-processing algorithms [1, 5]. In addition, Graham and Nelson of Brigham Young reported significant speedup of a genetic-programming algorithm [19, 20].

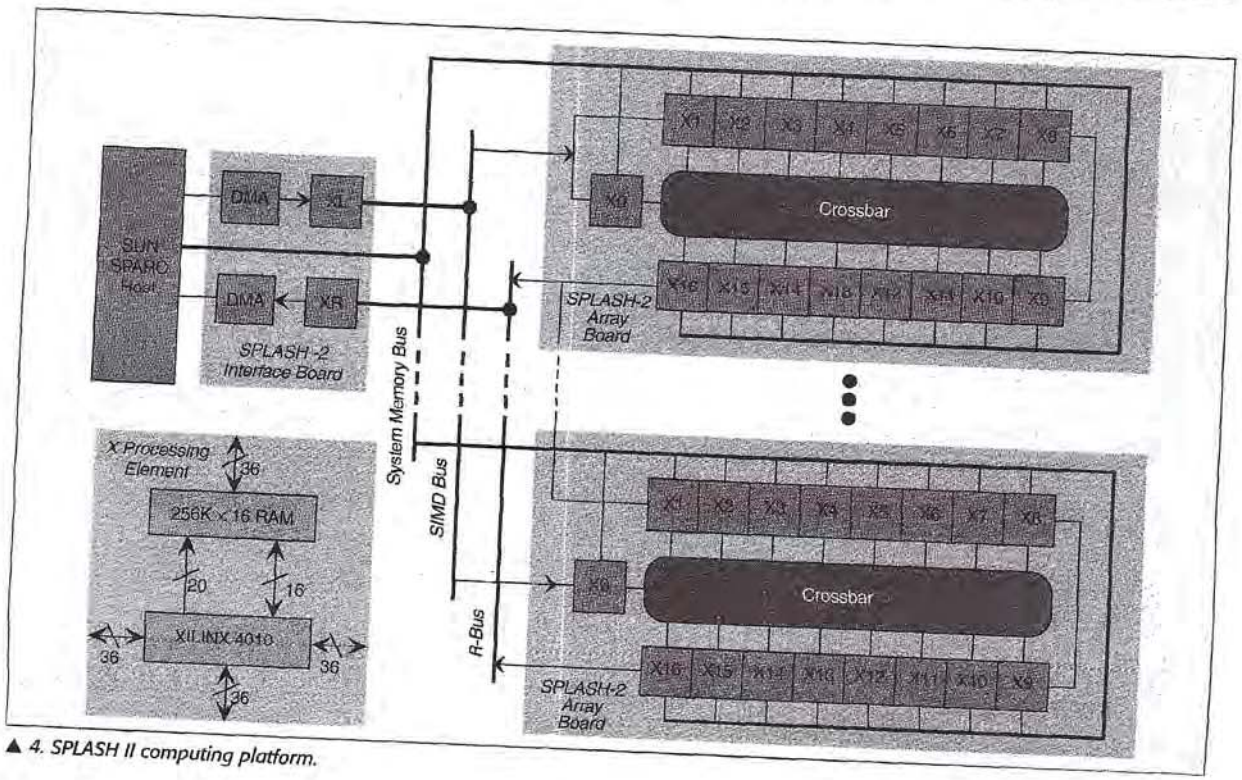
Configurable Computing: Run-Time Reconfiguration

The key attribute distinguishing many of the recent and ongoing efforts in FPGA-based computing from early systems like SPLASH and DECPeRLE is the use of FPGA reconfiguration during computation as a means to further boost performance. While the early systems achieved high performance by focusing on efficient mapping of algorithms into FPGAs, they were capable of only limited reconfiguration during execution. Recent systems specifically target run-time reconfiguration (RTR) as a means to enhance performance. This enables silicon to be time shared across multiple tasks.

Reconfiguration is, of course, not free, and one of the most interesting questions in configurable computing concerns the costs and benefits of RTR for different algorithms, FPGAs, and FPGA platforms. The basic tradeoffs in RTR are relatively straightforward. Reconfiguration costs time because the logic being reconfigured is unavailable for processing while the new bitstream is being downloaded. It also costs power since configuration bitstreams tend to be large (approximately a megabit for the largest currently available FPGAs) and involve a correspondingly large number of pin transitions when loaded onto an FPGA. A few devices, most notably the Xilinx 6200 series and the CLAy chip from National Semiconductor Corporation, permit partial reconfiguration in which a selected region of the logic can be configured without altering the state of the remainder of the FPGA. More often, however, FPGAs allow only a full reconfiguration.

It is only in the last five years or so that FPGAs have supported reconfiguration at speeds fast enough to permit RTR without intolerable overheads. Most FPGAs on the market today can be configured (depending on the device and the size) in between 1 and 50 milliseconds. There have been several proof-of-concept demonstrations of RTR in recent years, and this has stirred interest in the computing community as well as awareness by FPGA vendors of the benefits of fast reconfiguration. FPGA manufacturers, in turn, have made reconfiguration speed a design issue in the newer and next-generation devices, with the result that configuration times of a few milliseconds or faster are likely to become common within a few years.

A full evaluation and understanding of RTR will have to wait several years until the infrastructure of design tools, devices, and machines matures to the point where its potential can be fully realized, and even then it will remain somewhat of a moving target as technology continues to evolve. To date, there have been very few attempts to quantify the tradeoffs in run-time reconfigured systems. It is relatively easy to build a system in which an FPGA is reconfigured multiple times during computation. It is much harder to perform a quantitative evalua-



▲ 4. SPLASH II computing platform.

tion of the advantages, or lack thereof, that this enables relative to systems that perform the same task using other computational approaches. One of us (Hutchings) [49] has performed the first quantitative comparison of RTR for a range of applications. A few other researchers are also studying RTR, including Lysaght [30], Brebner [8], Luk [40], Singh [9], and Schmitt [37]. We expect that many more such studies will become available in the published literature in the next few years. To illustrate the nature and tradeoffs involved in RTR in more detail, we present two examples of systems below.

System Example 1: DISC

The first system example is DISC (Dynamic Instruction Set Computer), which is a research project started in 1994 at the Configurable Computing laboratory at Brigham Young University [46, 47]. In this ongoing experiment, researchers at BYU are experimenting with the following:

- Systematic approaches that ease the design and implementation of RTR applications
- New architecture configurations that integrate conventional processors and configurable-computing devices such as FPGAs
- Library-based techniques that promote the reuse of previously designed circuit modules to achieve both rapid application development and fast compilation

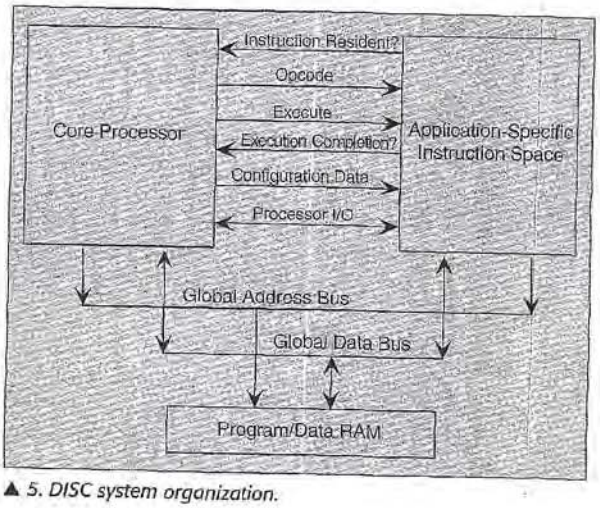
These three important research areas were motivated by BYU's past experiences with RTR applications. A systematic design framework is essential for RTR applications because these applications typically consist of many different configurations; there must be some standard means for these configurations to communicate results both with each other and to the outside world. Processor-FPGA interfaces are important because many of the basic housekeeping tasks that are unwieldy for FPGAs are much easier and efficient to implement on simple, conventional processors. Finally, a library-based approach is important because it makes it possible to *reuse* past efforts: high-performance, FPGA circuits require substantial human design effort, and a library-based approach makes it possible to exploit these efforts across many applications while also reducing compiler complexity and compile time.

Conceptually, DISC is a general-purpose "core" processor augmented with configurable hardware for performing application-specific tasks. Each application-specific function is treated as a custom instruction that augments the standard, general-purpose instruction set found in conventional programmable processors. For example, a DISC designed for image-processing applications might have a custom instruction that implements application specific circuitry for high-speed convolution. DISC overcomes the inherent problems of conventional application-specific hardware with a novel approach made possible by configurable logic: treat application-specific hardware (instructions) as distinct circuit modules that are stored as FPGA configurations in off-chip memory; at run-time, load and execute these circuit modules only as demanded by the application. Stored off-chip, these circuit modules only occupy valuable silicon resources while performing useful work and as soon as they are no longer necessary, they can be "swapped out" and replaced with other application-specific instruction modules. In addition, this library of application-specific circuit modules can be extended and individual circuit modules can be modified and redesigned over time to suit the needs of ongoing applications.

DISC System Architecture and Operation

The DISC architecture consists of two basic sections: a *static* core processor and a *dynamic* application-specific instruction space, which is slaved to the core processor. The basic organization is shown in Fig. 5. The main purpose of the core processor is control: it manages the application-specific instruction space and is responsible for loading circuit modules, initiating the execution of application-specific instructions, and managing I/O between the instruction space and the core processor. In most cases the core processor is static, i.e., implemented with circuitry that remains structurally unchanged throughout operation, and can be implemented with either a microprocessor or a statically configured FPGA—the FPGA in turn implementing either a simple state machine or programmable processor.

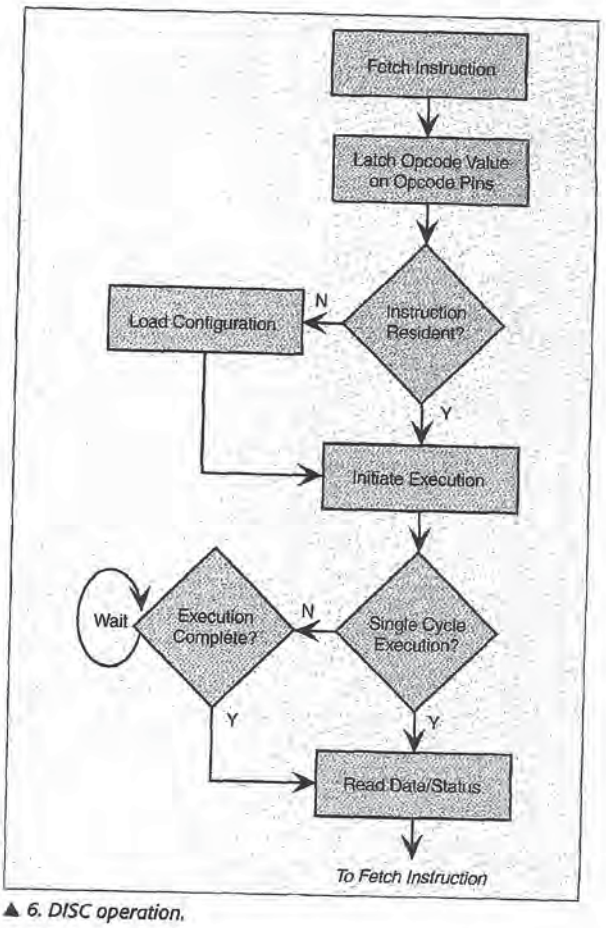
The application-specific instruction space is an undedicated array of reconfigurable logic; this is typically implemented with reconfigurable FPGA(s). Configurations that implement application-specific circuitry are loaded into this instruction space and executed as requested by the core processor. The instruction space is capable of storing several instructions (circuit modules) simultaneously and these instructions individually respond to initiation signals from the core processor. The control and I/O signals that form the interface between



▲ 5. DISC system organization.

the core and the instruction space are also shown in Fig. 5. As shown in the figure, the core processor and instruction space share a common address and data bus. This particular organization requires mutually exclusive operation: either the core processor is actively fetching instructions or data from memory, or the core processor is idle, waiting for some application-specific instruction to complete. However, this is not a requirement but rather a limitation of the current implementation.

In current implementations the core processor is a stripped-down programmable processor that can be programmed via a modified version of the lcc compiler [10,16]. A DISC program is then just a linear list of instructions that is assembled into machine code and stored in program memory for later fetching by the processor. The core processor fetches instructions from the program memory in a conventional manner and then processes them as shown in Fig. 6. Once the instruction is fetched, it is decoded and the opcode is latched onto the opcode pins (see Fig. 5 for these signal connections). The instruction-space reads the opcode value and quickly searches to see if the required circuit module is currently resident. If the required instruction is not resident, the instruction space requests the core to load the correct configuration. Once the load is complete (or if the circuit module was already resident), the core initiates the execution of the instruction module. The core processor then remains idle until the custom instruction has completed execution.



▲ 6. DISC operation.

Hardware Relocation and Internal Architecture of Instruction Space

The internal architecture of the instruction space is organized to support a concept referred to as relocatable hardware. This organization provides the necessary circuitry that supports the execution of any custom instruction of any size at any physical location within the instruction space. The basic organization consists of several static elements: a global controller and three bundles of wires (control, memory address, memory data) that are collectively referred to as the "context," as shown in Fig. 7(a). The context provides a means for custom instruction modules to communicate with external memory and perhaps other devices, and to the global controller—independent of the physical location they occupy while executing. Memory address and data signals are used by custom instructions to directly access the system memory as shown in Fig. 5.

Execution and control of custom instructions is coordinated by the global controller via the global control signals. The opcode (supplied by the core processor) is broadcast to all resident instruction modules via the global control signals; each custom instruction is then responsible for decoding this signal and asserting a global *resident* signal (another one of the global control signals). If no custom-instruction module asserts the resident signal, the global controller requests a configuration load from the core processor. Once the custom instruction is resident, and if the execution signal is asserted, the custom instruction begins execution and asserts the completion signal when finished. Supporting variable-size instructions is important: custom instructions vary greatly in their hardware needs and allowing their size to vary as dictated by their hardware requirements ensures efficient use of the instruction space. In Fig. 7(b), the instruction space is shown with two instructions resident: one general-purpose (add) and one custom (histogram). Note that while both instructions span the entire width of the instruction space, they vary in size in the vertical direction due to their hardware requirements.

Hardware relocation refers to dynamic physical relocation of circuit modules during system operation and is an important part of the DISC system that dramatically improves efficiency. Because circuit modules are dynamically loaded into the instruction space by the core processor as dictated by the user program; it is not possible to know a priori where a custom instruction will be loaded during operation. Moreover, as custom instructions may be loaded, executed, removed, and reloaded over time, commonly used custom instructions will typically need to be loaded into different physical locations at different times, depending upon where free space is available in the instruction space. The process for loading a

new custom instruction is: 1) find an unused portion of the instruction space, 2) process (relocate) the configuration data such that it can be placed in the unused portion, and 3) load the relocated custom instruction into the instruction space. If no free space exists, resident instructions are replaced according to a "least recently used" policy.

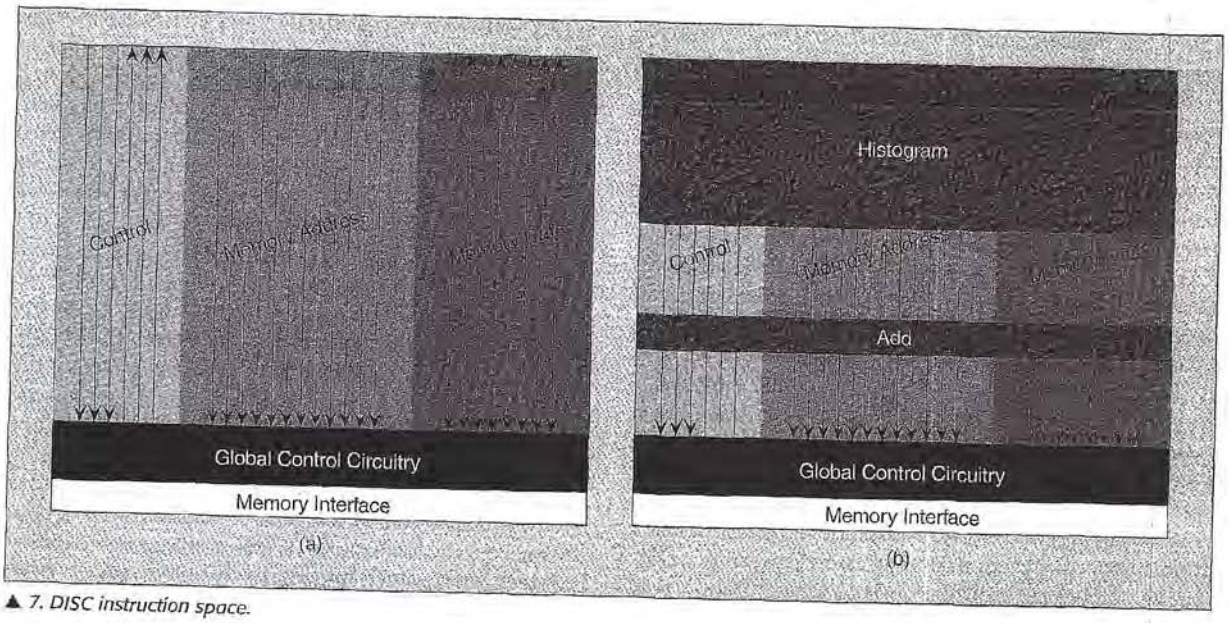
DISC Implementation

The system was implemented on the CLAyFUN board, an ISA prototyping board supplied by National Semiconductor. Consisting of two user-programmable CLAy FPGAs and memory, this board was housed in a conventional PC that provided disk storage for custom-instruction configurations and other general-purpose OS functions. One user-programmable CLAy FPGA was statically configured and implemented the core processor; the other was dynamically configured and implemented the instruction space. The CLAy device implementing the instruction space was partially configurable, thus making it possible to configure only those FPGA locations that would be occupied by the newly loaded custom instruction. Partial configuration was essential for this project as it allows custom instruction modules to be selectively loaded without disturbing the context of address, data, and control wires, global state, and other custom instructions currently resident. (At the time, partial configuration was supported only by the National CLAy and Atmel FPGAs. Xilinx has since introduced the XC6200, another device supporting partial configuration.) In addition, partial configuration directly reduces configuration time by reducing the number of configuration bits that are loaded into the FPGA.

Because FPGA resources were extremely limited, only a bare-bones processor with a single 16-bit accumulator was possible. Because of the core processor's limited performance, the actual process of bitstream relocation was handled by the 486 CPU in the PC enclosure; however, all of the functions that directly contributed to data processing were performed by the core processor. Although limited, the core processor was completely programmable in "C" using a modified version of lcc [16] and a custom assembler developed for this project [10]. In the assembler, application-specific instructions are accessed just as any other native machine instruction. In "C" code, these instructions are accessed like inline function calls simply by prepending a special prefix, **native-**, to the custom instruction identifier. The modified "C" compiler processes these calls in a special manner, replacing these custom function calls with native custom instructions in the generated assembly code.

A library of general-purpose and application-specific instructions are available for application development. The general-purpose instructions are a subset of those found on conventional microprocessors: add, subtract, shift, compare, etc. Custom instructions focused on image-processing applications and consisted of: mean filter, threshold, image subtract, morphological operations, etc. Early in the project the decision was made to implement nearly *all* instructions as circuit modules that are executed in the instruction space. Thus, general-purpose instructions compete for space in the instruction space and get swapped out when necessary to provide room for other instructions, either general-purpose or application-specific. Although this approach to general-purpose instructions actually *reduces* performance for general-purpose operations, it met the BYU research goals at the time.

To test the efficacy of the application-specific approach, a line-thinning (skeletonization) application was developed that used many of the custom im-



▲ 7. DISC instruction space.

age-processing instructions. Although high-performance was not the focus of this early project, the laboratory prototype still achieved good performance for the line-thinning application: a speedup of nearly an order of magnitude over a software implementation running on a 66 MHz DX-2 [48]. Note that these performance numbers include all configuration overhead and are for a substantially suboptimal system. The application required active paging of both the custom and general-purpose instructions—many of the general-purpose instructions were loaded more than once and to more than one location in the instruction space. In addition to its performance advantages, DISC also demonstrated significant resource savings. The accumulated FPGA resource requirements for all of the executed instructions would require three FPGA devices if statically configured. With DISC, idle hardware could be immediately reclaimed by writing over idle circuitry with new custom instructions as requested by the core processor.

System Example 2: Template-Based Automatic Target Recognition

The second system example is from the area of automatic target recognition (ATR). ATR is among the most demanding real-time computational problems, and one which maps particularly well to configurable computing platforms. ATR is the application focus of the DARPA-sponsored UCLA Mojave project in configurable computing. The challenge addressed by an ATR system is conceptually simple—to analyze a digitally represented input image or video sequence in order to automatically locate and identify all objects within the scene of interest to the observer. Since there are many types of imaging devices and many algorithmic choices available to a designer, there are clearly a large number of possible ways to implement an ATR system. The Mojave project uses ATR algorithms developed by Sandia National Labs for the U.S. Department of Defense Joint STARS airborne radar-imaging platform.

The processing used in ATR is illustrated in simplified synthetic aperture radar (SAR) images consisting of 8-bit pixels and measuring several thousand pixels on a side, and they are generated in real time by a radar imager. Images are input to a focus-of-attention processor that identifies a set of regions of interest, each of which contains a potential target. These regions of interest, referred to here as candidate images, must then be correlated with a very large number of target templates. Target templates are binary; e.g., each pixel is represented using one bit. The correlation results are output to a peak detector that identifies the template and relative offset at which the peak correlation value occurs. The correlation of candidate images with templates is an important computational bottleneck in the system, involving data rates and computational requirements that exceed by several orders of magnitude the processing load most of the other steps in the algorithm (though it should be noted that the focus-of-attention step shown in Fig. 8 is also very computationally intensive). While the precise system parameters vary with implementation, the work under way at UCLA uses candidate image sizes of 128 by 128 and template sizes of 32 by 32.

Figure 9 illustrates the correlation operation targeted for FPGA implementation in more detail. Target templates are binary (e.g., 1 bit per pixel) and occur in pairs, one member of which is called the bright template and contains pixels from which a strong radar return is expected, and the other member of which is the surround template and identifies pixels where strong radar absorption is expected. The templates tend to be sparsely populated, with only a relatively small percentage of the pixels set to 1. This property is important in obtaining high performance in FPGA implementations. The first step of the correlation is known as a shapesum calculation, in which the 8-bit candidate image is correlated with the bright template, providing for every pixel in the image a number that is used for local gain control. The second step is the actual correlation, which is performed on a binary version of the image that is obtained via thresholding. The shapesum value is used to select which threshold to apply to generate the binary image.

What is less clear is the extent to which configurable computing techniques will become useful in more general computing environments.

Run-time reconfigured FPGAs offer an extremely attractive solution to the ATR correlation problem. First, 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. In addition, the sparse nature of the templates can be leveraged to achieve a far more efficient implementation in the FPGA than could be realized in a general purpose correlator. This can be illustrated using the example of the simple template shown in Fig. 10.

In this example template, only five of the 24 pixels are "on." At any given relative offset between the template and image, the correlation output is the sum of the five binary pixels in the image that lie immediately above the "on" pixels in the template. The template can therefore be implemented in the FPGA as a simple adder tree as shown in Fig. 10. The image 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