Check for updates

Sequencing Run-Time Reconfigured Hardware with Software*

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

Abstract

Run-Time Reconfigured systems offer additional hardware resources to systems based on reconfigurable FPGAs. These systems, however, are often difficult to build and must tolerate substantial reconfiguration times. A processor based architecture has been built to simplify the development of these systems by providing programmable control of hardware sequencing while retaining the performance of hardware. Configuration overhead of this system is reduced by "caching" hardware on the reconfigurable resource. An image processing application was developed on this system to demonstrate both the performance improvements of custom hardware and the ease of software development.

1 Introduction

The high bandwidth of data and computational load of digital signal processing algorithms generally overwhelm even the highest performance general-purpose processors. Achieving real-time execution rates typically requires custom hardware. SRAM-based Field-Programmable Gate Arrays (FPGAs) are often used to implement this custom hardware because they do not incur non-recurring engineering charges (NREs), are widely available, and can be configured (and reconfigured) to suit a wide range of applications [1, 2].

One major benefit of SRAM FPGAs that is not often exploited is the ability to reconfigure the device *during execution of the application*. This technique, known as Run-Time Reconfiguration (RTR), is used to provide additional hardware resources for systems based on reconfigurable FPGAs[3]. Using RTR, two neural network systems were developed to operate on far fewer FPGAs than their statically configured counterparts[4, 5]. In addition, a wireless video coding system was constructed using fewer FPGA resources by rapidly configuring the FPGA between the image coding stages[6].

Although RTR systems provide additional hardware to FPGA based systems, they are not without problems. Few tools support the steps needed to adequately divide a design into equally sized temporal partitions. Partitioning in time requires the additional complexity of communicating between temporal partitions. In addition, systems employing RTR must tolerate the high time penalty required for configuration at run-time.

The Dynamic Instruction Set Processor (DISC)[3] is a run-time reconfigured processor designed with FPGAs to exploit the advantages of RTR while limiting the disadvantages discussed above. DISC provides a convenient method of sequencing application specific hardware. In addition, DISC allows the caching of hardware modules to reduce the overhead of frequently used modules. As demonstrated in the paper, this system provides both the flexibility and simplicity of conventional software design and the performance of application-specific hardware. A complex image-processing application will demonstrate these advantages.

2 DISC

DISC provides application-specific performance to a simple processor by allowing user-defined application-specific instructions to supplement a conventional instruction set. These user-defined instructions are designed in hardware to exploit the low-level parallelism, custom control, and specialized I/O of custom hardware circuits. Designed such, a hardware instruction module can execute an application-specific function many times faster than a sequential stream of general purpose instructions. Replacing a long series of conventional instructions with an optimized custom-hardware instruction module eliminates the instruction fetch, decode, and other overhead associated with general purpose instructions.

By designing application-specific functions as instruction modules, hardware circuits operate under software control. Controlling application-specific instructions in software allows DISC to retain the programmability of a conventional processor while preserving the performance of custom hardware. Complex sequences of custom instructions are easily described and mixed with conventional general purpose

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

instructions.

DISC removes the hardware limitations of the FPGA by reconfiguring hardware resources as demanded by run-time conditions. Implemented with the partially reconfigurable CLAy^{tm} FPGA[7], DISC instructions are paged in and out by partially reconfiguring the FPGA as demanded by the executing program. FPGA resources can be reused to implement an arbitrary number of performance-enhancing application-specific instructions.

The DISC processor is divided into two distinct regions: the processor core and the custom-instruction space. The processor core handles all instruction sequencing and remains static on the hardware during program execution. The custom instruction space is reserved for hardware instruction modules and is continually configured as run-time execution dictates.

2.1 Processor Core

A simple processor core was designed within the FPGA resources to sequence program instructions, interface with external memory, synchronize instruction swapping, and control inter-module communication. The processor core used on DISC is based on a simple accumulator model. The global accumulator register and several state signals are available to both the processor core and all loaded instruction modules. In addition, the processor core contains addressing control, basic sequencing capability, and dedicated I/O interfacing logic. The static processor provides several static instructions for simple sequencing and internal control.

2.2 Custom Instruction Space

Most FPGA resources required for DISC are reserved for the custom instruction space. Custom instructions are continually configured (and removed) on this instruction space as demanded by the application program. Frequently used custom instructions are kept inside the custom instruction space to eliminate the configuration overhead when reused.

To simplify run-time placement and provide maximum flexibility, DISC allows custom instruction modules to be placed *anywhere* within the FPGA. This flexible placement scheme allows the instruction module's location to be determined at *run-time* and not at design time. Hardware efficiency is maximized by allowing run-time conditions to dictate module placement.

Instruction modules that operate correctly at any location must be designed within a well-defined global context. The interface between the global circuit and the instruction module must appear the *same* at every location. DISC solves this problem by constructing a simple, but regular, global context throughout the FPGA. The global context for DISC is constructed by running global processor signals vertically on the FPGA and spreading these signals along the width of the FPGA as seen in Figure 1.

Instruction modules on DISC are designed under the strict constraints of this global context. As seen in Figure 2, instruction modules are designed horizontally, across the width of the FPGA. The modules lie perpendicular to the communication signals to gain access to each processor signal regardless of their vertical placement. Although each instruction module must span the entire width of the FPGA, each module may consume an arbitrary amount of hardware by varying its height.

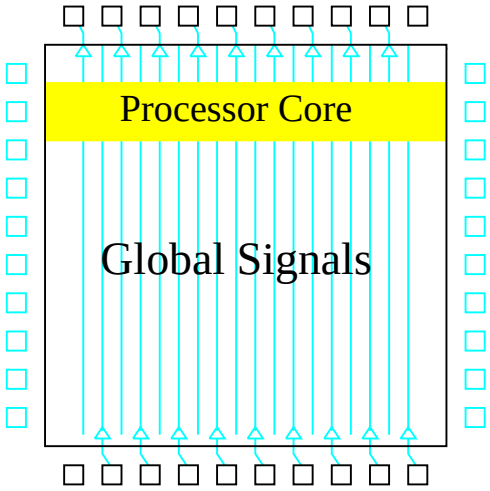

Figure 1: DISC Global Context.

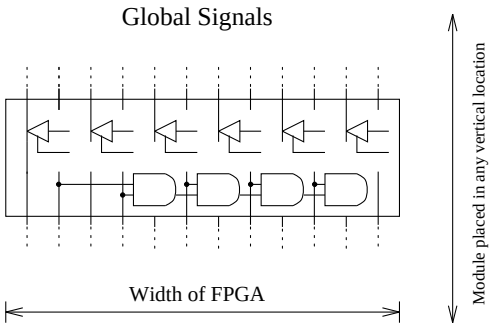

Figure 2: DISC Instruction Module.

3 DISC-II

DISC was constructed and tested on a simple wire-wrapped prototype board. A number of significant design problems made it difficult to use and limited its performance. By constraining the entire design to a single FPGA, few resources were available for both an adequate processor core and unallocated instruction hardware space. The 8-bit processor core provided few built-in instructions and had limited addressing range. In addition, the system board proved unreliable.

A second version of DISC has been built to address many of the problems of the initial "proof-of-concept" system discussed above. The system was developed on a stable FPGA-based board provided by National Semiconductor[8]. The new board offered several ad-

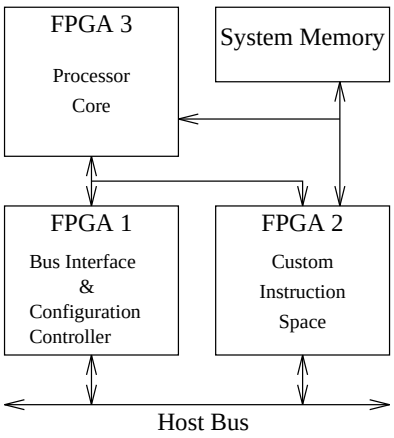

Figure 3: DISC-II System Partitioning.

vantages over the system board used earlier. It provides a dedicated host interface, additional memory, and three partially configurable FPGAs.

The additional FPGAs allowed a more natural partitioning of system functions. Most static resources were moved away from the custom instruction space to maximize dynamic hardware. The FPGAs were partitioned into the three systems as seen in Figure 3.

The Bus Interface, which remains static throughout its operation, actively monitors the processor state and configures modules on the custom instruction space as provided by the host. The Processor Core, which has been separated from the instruction space, sequences the instruction modules as dictated by the attached program memory. With additional FPGA resources, the processor core was improved to take advantage of the 16-bit bus widths found on the board. In addition, extra registers were added to provide more support for function calls, recursion, and high-level language addressing.

4 DISC Design Environment

Designing a DISC application requires both hardware and software development. Application-specific instruction modules are designed in hardware and the instruction modules are sequenced by a program written in software. Those operations of an application that can exploit circuit-level parallelism, custom I/O interfaces, or hardwired control are implemented as custom hardware. Global control, complex control sequences, and complex data structures are usually implemented in software.

4.1 Hardware Design

Application-specific instruction modules are designed under the constraints imposed by the global context. Structural models of the DISC system are available to ensure that user-defined modules conform to global context protocols. Once designed and verified, the instruction is physically mapped to DISC using the device specific mapping tools. Several custom post-processing utilities are used to insure the custom module adheres to global context constraints. Once a custom instruction module has been designed and mapped to the DISC context, it is available to *any* application program.

4.2 Software Development

The software program of a DISC application must organize the application instructions into a specific control sequence. These sequences may range in complexity from a simple assembly program to a complex C program making frequent references to custom instruction modules. Because the instruction set of DISC is never static, an assembler must allow users to continually define and use new instructions in application programs. A retargetable assembler is available for DISC that allows users to add and use new instructions as they are developed[9]. New instructions are defined by specifying the instruction mnemonic, instruction type, opcode, and any necessary control parameters.

The LCC retargetable 'C' compiler was targeted to the standard DISC instruction set[10]. Additional syntax was added to the compiler to support custom instruction calls from within the C language. The compiler allows users to mix high-performance custom instructions with familiar 'C' control sequences. Custom instruction modules are referenced in C by a function call with a `native_` prefix as follows:

```
native_lowpass(a);
```

After compilation, a post-processing step substitutes all `native_` function calls with their corresponding instruction name.

5 Object Thinning on DISC

An image processing instruction library was created for DISC to demonstrate both the performance improvements of custom hardware and the ability to easily sequence DISC custom instructions. Most of these operations obtain significant speed improvements through parallel computation, custom memory addressing, and pipelined arithmetic. Each instruction operates at the DISC execution speed of 8 Mhz. Table 1 lists the available image processing instructions and their associated size (in DISC rows).

With the custom image processing instruction library and high-level language support, an object thinning algorithm was implemented on DISC. This algorithm is divided into the following three operations: pre-filtering, thresholding, and region thinning. The image of Figure 4 was used as the original test image and the software code for the algorithm is as follows:

```
main(){
   image *image1, *image2;
   histogram *hist;
   int thresh;
   int skel;

   image2 = native_lowpass(image1);
   image1 = native_clear(image2);
```

Module	Rows
Image inversion	4
Image move(copy)	4
Image clear	5
Image threshold	7
Histogram generation	8
Dilate and Erode	7
Image difference	7
Median pixel	9
Skeletonization	25
Low-pass filter	28
Edge detection filter	30
High-pass filter	42

Table 1: DISC Image Processing Instruction Library.

Figure 4: Original Test Image.

```
   hist = native_histogram(image2);

   thresh = peakthresh(hist);

   image1 = native_threshold(thresh,image2);

   for (skel = 0; skel != 0;) {
      skel  = native_skeleton1(image1,image2);
      skel += native_skeleton2(image2,image1);
   }
}
```

5.1 Pre-filtering

The DISC object thinning algorithm is intended to operate on high-contrast gray-scale images such as news print, handwriting, and lettering. The original image is filtered through a simple low-pass filter to remove high-frequency noise that may introduce unwanted regions.

The `LOWPASS` instruction from the image processing library performs a suitable low-pass filter. Figure 5 displays the result of an image filtered through the

Figure 5: Low-pass Pre-filtering.

`LOWPASS` instruction. C code to execute the filter appears as follows:

```
      .
   image2 = native_lowpass(image1);
      .
```

5.2 Thresholding

Once the high-frequency noise of the image has been removed, it is ready for the next operation: thresholding. Thresholding algorithms reduce image information by converting a gray-scale image into a simple binary image. With good thresholding algorithms, objects of interest are placed in the foreground while all other image information is placed in the background. The most straight-forward method of thresholding an image compares each pixel value in the image with a predetermined threshold value. Pixels with intensity greater than the threshold value are placed in the foreground, while those less than the threshold are placed in the background.

Although this process of thresholding is relatively simple, obtaining a good threshold value can be troublesome. The process of choosing a threshold value begins by obtaining a histogram of the input image. In some cases, finding the mean or median pixel of the histogram is sufficient. These simple approaches, however, are inadequate for high contrast images with a dominant foreground or background. A more effective technique for these types of images involves detecting the peaks of the histogram. A surprisingly good threshold value can be obtained by calculating the midpoint between the foreground peak and the background peak.

By using instructions from the image processing library, most of the thresholding process can be completed with high-speed custom hardware. To build a histogram table from the input image, the `CLEAR` and `HISTOGRAM` instructions are used. The `CLEAR` instruction initializes the histogram memory and the `HISTOGRAM` instruction efficiently builds the image histogram. There is no custom instruction to determine histogram peaks or find the mid-peak threshold

Figure 6: Threshold Image.

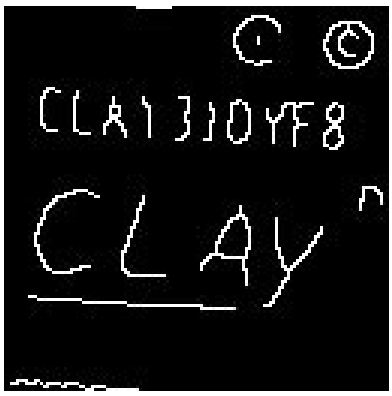

Figure 7: Image Skeleton.

value. To compute the threshold value using the peak method will require conventional software code written as a subroutine in 'C' or assembly. The `THRESHOLD` instruction is also available to convert the source image into the binary image using the threshold value determined in the software subroutine. The 'C' source code to complete thresholding of the input image is as follows:

```
  .
image2 = native_lowpass(image1);
image1 = native_clear(image2);
hist = native_histogram(image2);

thresh = peakthresh(hist);

image1 = native_threshold(thresh,image2);
  .
```

Figure 6 displays the resulting thresholded image.

5.3 Object Thinning

Object thinning algorithms reduce the redundant shape information from an image to simplify the processes of object recognition. The simplified images obtained from the algorithms represent the shape of the original objects with single-pixel lines. The single-pixel lines representing object shapes are often called the "skeleton".

Object thinning algorithms usually involve successive removal of the outer edges of an object until only the skeleton of the object remains. This thinning processes is similar to the erosion operation that uniformly erodes the outer edges of image objects. Unlike erosion, however, object thinning must never completely destroy an object or disconnect object regions. The algorithm will selectively remove outer edges of objects with the following constraints[11]:

- connected object regions must thin to connected line structures,
- approximate end-line locations should be maintained,
- the skeleton should be one pixel wide,
- the skeleton should lie at the center of the object cross section,
- the skeleton must contain the same number of connected regions as the original image.

There are many algorithms available for object thinning that trade skeleton quality for algorithm speed. The Zhang-Suen thinning algorithm[12] was chosen because of its acceptable results and its suitability for hardware. This algorithm determines whether a foreground pixel can removed by evaluating its neighborhood.

To skeletonize the binary image using the Zhang-Suen algorithm, a custom instruction module, `skeletonize` (25 rows in DISC), was designed. Each execution of the instruction module completes one pass of the thinning algorithm and indicates whether the operation is complete. To effectively complete the thinning algorithm, the instruction module must execute until it no longer modifies the image. The resulting thinned image is seen in Figure 7. The code used to completely skeletonize an image on DISC is as follows:

```
for (skel = 0; skel != 0;) {
   skel  = native_skeleton1(image1,image2);
   skel += native_skeleton2(image2,image1);
}
```

5.4 Run-Time Execution

Executing all steps of the object thinning requires five custom image processing instructions (`LOWPASS`, `CLEAR`, `HISTOGRAM`, `THRESHOLD`, and `SKELETON`) and the three instructions necessary to support C constructs (`SHIFT`, `COMPARE`, and `ADD`). These instructions consume 83 rows of custom logic, or almost two times the available static hardware for custom instructions. Since all instructions can not fit on the processor at the same time, instructions must be swapped to make room as the algorithm proceeds.

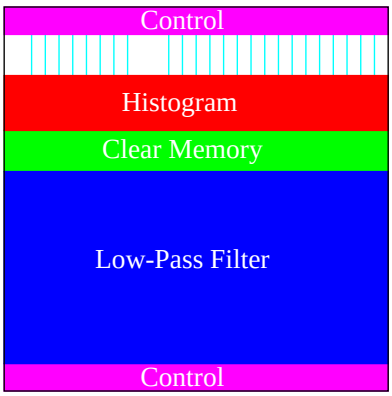

Figure 8: DISC State After Executing `HISTOGRAM`.

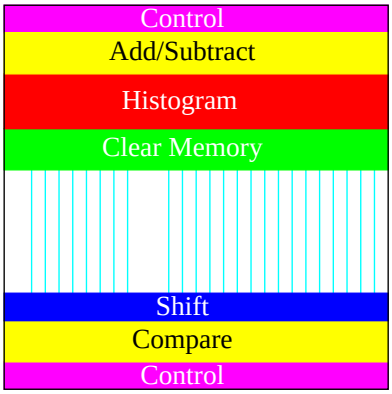

Figure 9: DISC State After Executing C Code.

Beginning with a clean instruction space, the custom instruction modules `LOWPASS`, `CLEAR` and `HISTOGRAM` are configured and executed on DISC. After executing these steps, the custom instruction space appears as seen in Figure 8.

Once the histogram is complete, the software subroutine to compute the threshold value is executed. The C code uses several custom instructions not built into the global controller. Three of these instructions are needed for the sub-routine: `ADD`, `COMPARE`, and, `SHIFT`. Since there is not enough room on the FPGA for all three modules, DISC must make room by removing older modules. At this stage, the `LOWPASS` module is the least recently used module and is removed. Figure 9 displays this transition.

With the computed threshold value, the image is converted into a binary image with the `THRESHOLD` custom instruction. With ample space in hardware to place the instruction, no modules are removed. After thresholding the image, the image is thinned with the skeletonization instruction. At this stage, the `SKELETON` module will not fit within the available hardware and the two oldest modules `CLEAR` and `HISTOGRAM` are removed to make room. With the `SKELETON` module in place, it is executed iteratively until object thinning is complete. Figure 10 displays this last transition.

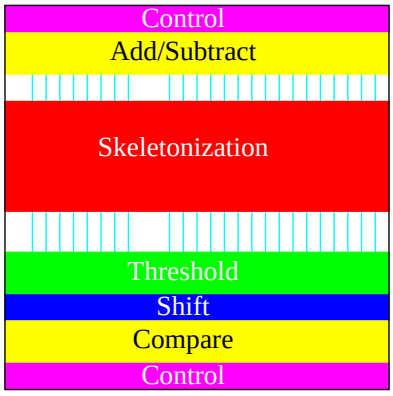

Figure 10: DISC Executing the `SKELETONIZATION` Instruction.

6 Results

For purposes of comparison, two versions of the object-thinning algorithm were implemented: one on DISC and the other on a 66 Mhz 486 PC. In all cases, DISC outperformed the PC by a significant margin as shown in the Table 2.

IMAGE	486	DISC	Speedup
Silk screen	2.17s	.29s	6.5
Block letter	3.90s	.53s	6.4
News print	9.90s	.89s	10.1

Table 2: DISC and PC Execution Results.

Even though DISC provided performance improvements over the host machine, much of the execution time is spent configuring and moving the custom instructions. With the silk screen image, over 25% of execution time is spent configuring DISC and handling instruction moving. Note that DISC is running at a clock rate approximately 1/8 (8 Mhz) that of the microprocessor.

7 Conclusion

This paper demonstrated several advantages of providing software sequencing support for run-time reconfigured hardware. First, RTR applications are implemented as a mix of software ('C') and hardware (custom instruction modules). This adds a great deal of flexibility to the development process. Designers can implement customized hardware modules where necessary to achieve high performance and then use software, with its faster design cycle, to implement com-

plex control and functions that are not performance sensitive. Second, hardware modules are reusable. Because all hardware modules are organized around the same global context, they can easily be reused across any number of different applications without the need for place and route. Large libraries of custom instructions can easily be created and these are immediately available simply by referencing them by name in a 'C' program. The global context and run-time environment of DISC automatically take care of all other concerns. Finally, the development of RTR applications is simplified. FPGA hardware resources are easily reused - any unused DISC resources are automatically reclaimed as necessary through the demand-driven module replacement process. In addition, the extensibility of DISC allows the object thinning application to be combined with more complex image processing algorithms. With little more than additional high-level programming, the object thinning application can be extended to a complete object recognition system.

References

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

[2] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In G. Borriello and C. Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 88–102, 1993.

[3] B. L. Hutchings and M. J. Wirthlin. Implementation approaches for reconfigurable logic applications. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 419–428, Oxford, England, August 1995. Springer.

[4] J. G. Eldredge and B. L. Hutchings. Density enhancement of a neural network using FPGAs and run-time reconfiguration. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, Napa, CA, April 1994.

[5] J. D. Hadley and B. L. Hutchings. Design methodologies for partially reconfigured systems. In P. Athanas and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 78–84, Napa, CA, April 1995.

[6] B. Schoner, C. Jones, and J. Villasenor. Issues in wireless coding using run-time-reconfigurable FPGAs. In P. Athanas and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 85–89, Napa, CA, April 1995.

[7] National Semiconductor. *Configurable Logic Array (CLAy) Data Sheet*, December 1993.

[8] C. R. Rupp. *CLAyFUN^{tm} Reference Manual*. National Semiconductor, September 1994. Version 2.00.

[9] D. A. Clark. *DASM Reference Manual*. Brigham Young University, Provo, Utah, 1995.

[10] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Publishing, 1995.

[11] J. R. Parker. *Practical Computer Vision Using C*. John Wiley & Sons, Inc., 1994.

[12] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM*, 27(3):236–239, 1984.