A Dynamic Instruction Set Computer*

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

Abstract

A Dynamic Instruction Set Computer (DISC) has been developed that supports demand-driven modification of its instruction set. Implemented with partially reconfigurable FPGAs, DISC treats instructions as removable modules paged in and out through partial reconfiguration as demanded by the executing program. Instructions occupy FPGA resources only when needed and FPGA resources can be reused to implement an arbitrary number of performance-enhancing application-specific instructions. DISC further enhances the functional density of FPGAs by physically relocating instruction modules to available FPGA space.

1 Introduction

Developing customized stored-program processors is a convenient design technique that combines the enhanced performance of application-specific circuits with the flexibility of general-purpose programmable processors. Application-specific instruction sets, customized I/O and optimized control can substantially improve the performance of even the simplest programmable processors. FPGAs provide an excellent implementation platform for application specific processors because of the quick development time and simplified design process. In addition, SRAM based FPGAS provide the ability to reconfigure more than one distinct application-specific processor on a single device.

A number of general purpose processors have been developed to show the feasibility of implementing a processor architecture on an FPGA[5, 7, 17]. Several custom processors have successfully demonstrated the advantages of adding specialized hardware to general purpose processor cores. Application areas for these processors include digital audio processing[16], systems of linear equations[17], and statistical physics[12].

One limitation of building customized processors on FPGAs is the lack of hardware resources available for specialized instruction sets. A few hardware-intensive instruction modules can quickly consume all the resources of even the largest FPGAs available today. Reconfiguring an FPGA to replace idle circuitry *during* application execution can provide more hardware resources than is available on a one-time configured FPGA. This technique, known as run-time reconfiguration (RTR), has been shown to increase the functional density of reconfigurable FPGAs[6]. The DISC processor uses RTR to ameliorate FPGA hardware limitations and provide an essentially limitless application-specific instruction set.

Early attempts in modifying a processor instruction set involved a writable control store and generating custom micro-code for each application[14]. The PRISM project extended this idea by augmenting the instruction set of a standard RISC processor with application-specific instructions on a tightly coupled FPGA. Hardware images of these instructions are extracted and compiled from the source code transparent to the user[2]. The WASMII project discusses a more dynamic approach that involves swapping hardware compute configurations in and out of the FPGA resource as demanded by the data-flow token[9].

The DISC processor implements each instruction in the instruction set as an independent circuit module. The individual instruction modules are paged onto the hardware in a demand-driven manner as dictated by the application program. Hardware limitations are eliminated by replacing unused instruction modules with usable instructions at *run-time*. An application running on DISC contains source code, indicating instruction ordering, and a library of application-specific instruction circuit modules.

This paper will begin by describing the techniques used to implement DISC. These include partial reconfiguration, relocatable hardware, and the linear hardware model. The architecture of the DISC processor will be presented along with several example custom instructions. The DISC processing system, including software and hardware platform, will be described. The paper will conclude by presenting results from an algorithm implemented on DISC.

2 Partial FPGA Reconfiguration

DISC takes advantage of partial FPGA configuration to implement dynamic instruction paging. Partial reconfiguration provides the ability to configure a subsection of an FPGA while remaining logic operates unaffected. Although all SRAM-based FPGAs can be reconfigured in-circuit, only the CAL[1], Atmel[3], and National Semiconductor[13] FPGAs support the ability to *partially* reconfigure hardware resources.

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

Although few partially reconfigurable systems have actually been implemented, several have been proposed such as hardware multi-tasking[10], a multi-phase serial communication algorithm[11], a data acquisition system[4], and a self-reconfiguring processor[8]. In addition, caching logic to increase hardware efficiency in standard digital systems has been proposed using partially reconfigurable FPGAs[15].

DISC uses partial configuration to implement custom-instruction caching. Instruction modules are implemented as partial configurations and individually configured on DISC as demanded by the application program. Before initiating execution of a custom-instruction, DISC queries the FPGA for the presence of the custom-instruction configuration. If the custom-instruction is on the FPGA, execution is initiated. Otherwise, program execution pauses while the custom-instruction is configured on the FPGA.

As a typical program executes, custom-instructions are configured onto the FPGA until all available hardware is consumed. When all hardware is used by the custom-instructions, new custom-instruction modules may not be configured on the FPGA until enough existing hardware is removed. By replacing the oldest custom-instruction modules on the FPGA with newer modules, the FPGA serves as a cache of the most-recently used custom-instruction modules.

2.1 Example

The following assembly language source code exemplifies the use of partial configuration on DISC:

```
begin:
       ;instruction INSTA operates on
       ;memory location mem1
   INSTA  mem1
   INSTA  mem2
       ;instruction INSTB operates on
       ;mem3 and mem2
   INSTB  mem3,mem2
       ;"loopback" label defined
loopback:
   INSTC  mem3
       ;instruction CMP compares
       ;mem1 with mem3
   CMP    mem1,mem3
       ;instruction JNE jumps
       ;to loopback if not equal
   JNE    loopback
continue:
  INSTD  mem3
  INSTB  mem2
  INSTE  mem3
end:
```

Once each instruction in the previous program (`INSTA, INSTB, INSTC, CMP, JNE, INSTD, and INSTE`) has been designed as an independent partial configuration, the source code representing the program is loaded into DISC and the processor begins execution. The sequencing of instructions on a small FPGA may execute and configure as follows:

Operation	**Instruction**	
Configure	`INSTA`	Configure `INSTA` on FPGA
Execute	`INSTA`	Execute first `INSTA`
Execute	`INSTA`	Execute second `INSTA`
Configure	`INSTB`	Configure `INSTB` on FPGA
Execute	`INSTB`	Execute first `INSTB`
Configure	`INSTC`	Configure `INSTC` on FPGA
Execute	`INSTC`	Execute first `INSTC`
Execute	`CMP`	Execute `CMP` (always available)
Execute	`JNE`	Execute `JNE` (always available)
(continue looping to `INSTC` until `JNE` fails)		
Remove	`INSTA`	FPGA full, remove oldest module
Configure	`INSTD`	Configure `INSTD`
Execute	`INSTD`	Execute `INSTD`
Execute	`INSTB`	Execute second `INSTB`
Remove	`INSTC`	FPGA full, remove oldest module
Configure	`INSTE`	Configure `INSTE`
Execute	`INSTE`	Execute `INSTE`

In the previous example, it is assumed that the first five instructions (`INSTA, INSTB, INSTC, CMP`, and `JNE`) consume all available space on a single FPGA. Partially configuring the FPGA allows two additional instructions (`INSTD` and `INSTE`) to execute on an otherwise full FPGA.

2.2 Advantages

Partial configuration provides a number of advantages for DISC over conventional configuration methods. First, idle instruction modules can be removed to make room for other usable modules. The ability to *replace* instruction modules in the system at run-time allows the implementation of an instruction set much larger than is possible on a single one-time configured FPGA.

Second, configuration time is substantially reduced. Although the DISC FPGA could be completely configured every time a new instruction is needed, configuration overhead can be dramatically reduced by configuring only the requested instruction. Reducing the size of hardware to configure significantly reduces the configuration bit-stream. Configuration bit-stream reductions for DISC instruction modules fall between $\frac{1}{60}$ and $\frac{1}{3}$ of a complete FPGA configuration. With a significantly smaller bit-stream, the corresponding configuration time is reduced. In an environment of run-time configuration, reducing the configuration time will limit the reconfiguration overhead.

Third, system state can be saved on the FPGA *during* configuration. Conventional configuration techniques prevent the preservation of system state during configuration by destroying the contents of *all* flip-flops. Implementing DISC with conventional configuration methods would require the saving and restoring of system state (program counter, register values, etc.) every time a configuration occurs. To prevent the time-consuming process of saving and restoring

state, DISC implements a global controller that remains on the FPGA at all times.

In summary, partial configuration allows DISC to implement an essentially infinite instruction set in hardware with limited configuration and state-preserving overhead.

3 Relocatable Hardware

The ability to partially configure custom-instruction modules allows DISC to implement an important strategy - relocatable hardware. Relocatable hardware, implemented only in partially configurable FPGAs, provides the ability to *relocate* or make placement decisions of partial configurations at *run-time*. Although not essential for a general purpose processor, it is used on DISC to substantially improve run-time hardware utilization.

Sub-modules in traditional digital systems require a single fixed location in hardware because of strict global and local physical constraints. Because sub-modules in traditional systems are not paged in and out of hardware, a fixed location does not pose any problems and global optimizations can be made on the static circuitry to improve hardware utilization. In a run-time partial reconfigurable system, however, fixed locations for partial configurations can pose serious performance problems.

If DISC modules are designed for a single physical location, instructions in the library will inevitably overlap each other on the hardware. Two overlapping instructions can *never* operate properly on the FPGA at the same time. If two overlapping instructions are used frequently together in an application program, the configuration overhead needed to replace the instructions quickly becomes the system bottleneck. DISC removes these problems by designing each custom-instruction module for *multiple* locations on the FPGA.

The flexibility of multiple locations for DISC custom-instructions significantly improves run-time utilization. Instruction modules are initially configured on the FPGA as close as possible to avoid wasted hardware between modules. Once the hardware space is full, additional instruction modules are placed in locations where older unneeded instruction modules currently lie. Relocatable hardware allows run-time constraints and conditions to dictate instruction module placement for optimal hardware utilization.

Relocatable hardware is implemented by designing custom-instruction modules around a firmly defined global context. A global context provides physical placement positions and a communication network necessary for these modules to operate correctly. The global context partitions the available hardware into an array of potential placement locations for the relocatable instruction modules. The communication network is provided at each placement location to insure adequate communication between the global controller and the instruction modules at *any* location.

In order to design instruction modules that fit within the global context, all instruction modules must be physically independent from each other. The physical layout of any instruction module must have no affect on the physical layout or placement of any other module in the library.

4 Linear Hardware Space

DISC implements relocatable hardware in the form of a linear hardware model. As the name suggests, the model is based on a linear, one-dimensional hardware space. The two-dimensional grid of configurable logic cells are organized as an array of rows: location is specified by vertical location and module size is specified by module height (in rows).

The global context for the linear hardware model consists of a uniform communication network and a global controller. The communication network is constructed by running each global signal vertically across the die and spreading the global signals across the width of the die parallel to each other (see Figure 1).

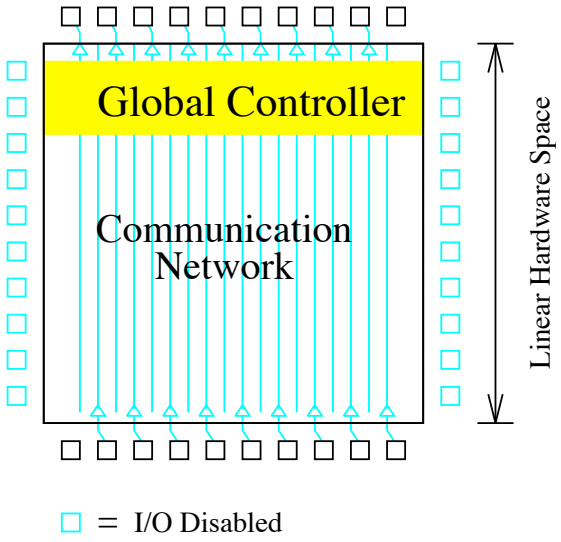

Figure 1: Linear Hardware Space.

The communication network provides access to global resources for all instruction modules and performs intermodule communication. The global controller specifies the communication protocol, controls global resources (such as I/O and global state) and monitors circuit execution. The global controller and the communication network remain in the same location throughout application execution to preserve the global context.

To gain access of all global signals, sub-modules within a linear hardware space are designed horizontally, across the width of the FPGA. The modules lie perpendicular to the global communication signals for full access of *all* global signals regardless of their vertical placement (see Figure 2). Although all sub-modules must span the entire width of the FPGA, each module may consume an arbitrary amount of hardware by varying its height.

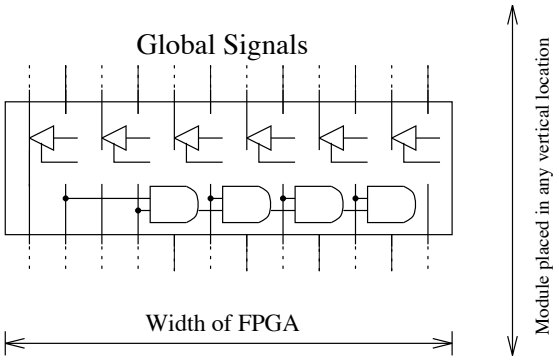

Figure 2: Simplified Custom Instruction Module.

Relocatable circuit modules communicate as established by the global protocol and thus operate properly *at any vertical location*. In a run-time environment, these circuit modules can be relocated as needed to optimize the available hardware space.

5 DISC Architecture

The DISC architecture implements relocatable hardware with the linear hardware model on a single National Semiconductor CLAy31 FPGA coupled to an external RAM. The CLAy31 provides a 56 x 56 array of fine-grain logic cells allowing 56 complete rows in the linear hardware space. A complete processor is made by coupling a global controller to a library of custom-instruction circuit modules (see Figure 3).

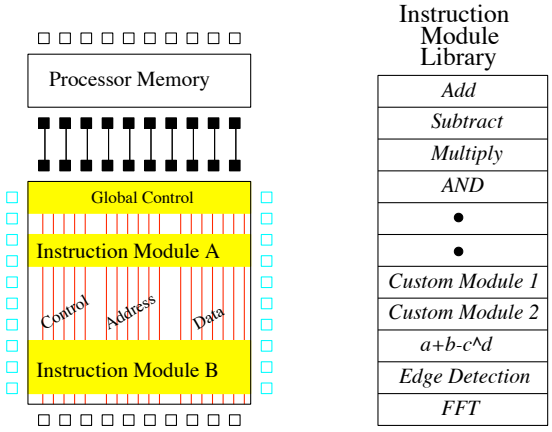

Figure 3: DISC Linear Hardware Space.

5.1 Global Controller

The global controller provides the circuitry for operating and monitoring global resources such as the external RAM, I/O, the internal communication network and global state. The global controller consumes ten complete rows (approximately 1/6 of the chip) leaving 46 rows available for custom-instruction modules. The physical layout of the global controller, estimated at 1007 gates, along with the communication network is seen in Figure 4.

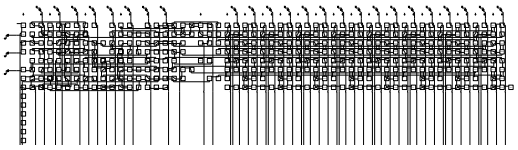

Figure 4: DISC Global Controller Layout.

The architecture of the global controller is seen in Figure 5 and is comprised of the following submodules:

- Data Register (**DR**): stores intermediate results, provides inter-module communication buffering and assists in complex address generation (8 bits),
- Address Register (**AR**): provides standard addressing modes for memory access (16 bits),
- Program Counter (**PC**): provides the sequencing capability of the processor (16 bits),
- Status Register (**SR**): stores internal state of the processor (4 bits),
- Instruction Register (**IR**): stores the opcode of the current instruction (8 bits),
- Global Control Unit (**GCU**): contains the circuitry necessary to preserve communication protocol, sequence through processor states, and interface with I/O.

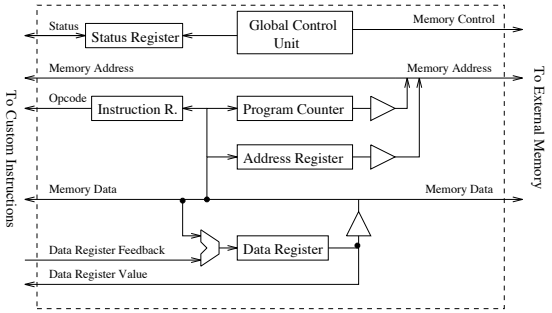

Figure 5: DISC Global Controller Architecture.

The global controller provides a consistent communication interface and standard protocol for all custom-instructions at every vertical location. The global signals available to the custom-instructions include the following:

- Data Register Value: accesses contents of Data Register (8 bits),

- Data Register Feedback: provides new values for Data Register (8 bits),
- Memory Address: allows address generation control by custom-instructions (16 bits),
- Memory Data: allows bi-directional access of memory data by custom-instructions (8 bits),
- Status Signals: provides control capability for custom-instructions (4 bits),
- Instruction Register: provides opcode of current instruction (8 bits).

The global controller is also responsible for sequencing through the instruction cycles for the custom-instruction modules. The following instruction cycles are implemented by the global controller:

- Instruction Fetch (**IF**),
- Operand Fetch (**OF**),
- Halt Processor (**HP**),
- Custom Cycle (**CC**),
- Instruction Execution (**EX**).

The **IF** cycle stores the current program memory into the instruction register and increments the program counter. The **OF** cycle stores the current program byte into the address register and also increments the program counter. The **HP** cycle causes all processor resources to remain idle and is used during configuration. The **CC** cycle is used by complex custom-instruction modules for adding additional cycles and has no affect on global resources. The **EX** cycle loads the value of the data register with the contents of the data register feedback path.

Each instruction in the library operates in one of two possible instruction cycle sequences: **standard** and **custom**. The standard instruction sequence follows a simple three-cycle execution: **IF**, **OF**, and **EX**. Any instruction that completes its computation or function in a single clock cycle, such as basic arithmetic and logic operations, will operate with this sequence.

The custom-instruction sequence offers additional cycles for complex custom-instructions. The custom sequence begins with the following two cycles: **IF** followed by **OF**. The sequence then varies by inserting as many **CC** cycles as necessary to complete a complex application-specific operation. The custom-instruction sequence completes with the **EX** instruction cycle. The custom-instruction module has complete control over the number of **CC** cycles needed for a particular function. Some instructions add as few as one cycle, while others require thousands of cycles for a single operation. Figure 6 displays the two instruction sequences.

The global control unit contains a number of default instructions necessary for controlling global resources. These instructions are used for sequencing, status control, and memory transfer and include the following:

- set carry: sets carry bit in status register,
- clear carry: clears carry bit in status register,
- store data register: store data register in memory,

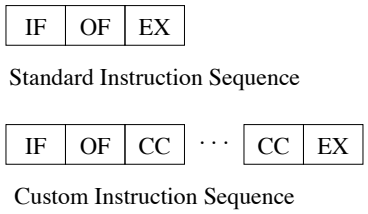

Figure 6: DISC Instruction Sequences.

- load data register: load data register from memory,
- conditional jump: jump with carry not set.

Each of these instructions follow the standard instruction sequence of three cycles. These instructions, coupled with the custom-instruction library designed for a particular application, provide the complete instruction set of the processor. An application can implement an instruction set of *any* size by paging instruction modules in a demand-driven manner from the instruction library.

5.2 Custom-instruction Modules

Custom-instruction modules vary in size and complexity, but each is designed to fit within the global context described above. Specifically, each module contains a decode and a data-path unit. Complex modules contain additional control structures.

The decode unit assigns a specific op-code to the custom instruction and is responsible for acknowledging its presence to the global controller. The decode unit compares the contents of the **IR** for a match against its own opcode during the **OF** cycle. On a positive match the module signals the global controller that the hardware is present and instruction sequencing continues.

The data-path is responsible for providing the proper connections to the global communication network and adhering to the established communication protocol. Instruction modules not executing refrain from sending any signals on the communication channel to prevent the corruption of other operating instructions. The data-path unit provides a new value for the data register during the **EX** stage. Most instructions perform their function by modifying the **DR**.

Several custom-instruction modules of varying size have been implemented on DISC. These vary from a simple single row shifter to a complex edge-detection module of 34 rows. Table 1 shows the current instructions available for DISC. The circuit layout for the Adder/Subtracter module is seen in Figure 7.

6 System Operation

The DISC processor was implemented on a PC-ISA custom board made exclusively for the study. The board includes static bus interface circuitry, two CLAy31 FPGAs, and memory. A configuration controller is implemented on the first FPGA to monitor

Module	Rows	Gates
Shifter	1	50
Comparator	3	155
Add/Subtract	3	153
Addressing Modes	4	447
Masking Operations	5	193
Logical Operators	9	232
Big-Level Operations	9	296
Mean Filter	31	2156
Edge Detector	33	2221

Table 1: Sample Custom Instruction Modules.

Figure 7: DISC Adder/Subtracter Custom Module Layout.

processor execution and request instructions from the host. DISC is implemented on the second FPGA and the application program memory is stored in the adjacent memory (see Figure 8). The board operates under a UNIX-based operating system and is controlled by a host device driver.

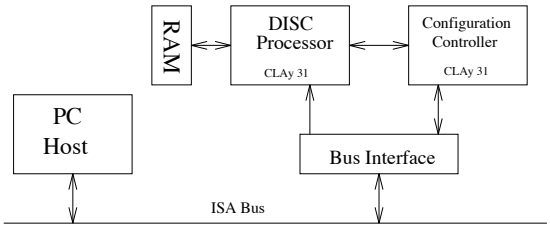

Figure 8: DISC System.

Performance has not been a main consideration as DISC was implemented primarily to study dynamic instruction set modification through partial reconfiguration. As a research tool, the processor is 8 bits and operates at the host bus speed of 7.5 MHz (maximum operating speed calculated at 12 MHz). Processor widths and operating speeds can be increased as device densities increase and tool enhancements become available.

A DISC application is initiated by first, loading the program memory with the target application, and second, configuring the DISC FPGA with the global controller. During execution, the processor validates the presence of each instruction in the hardware. If the instruction requested by the application program does not exist on the hardware, the processor enters a halting state and requests the instruction module from the host.

Upon receiving a request for an instruction module, the host evaluates the current state of the DISC FPGA hardware and chooses a physical location for the requested module. The physical location is chosen based on available FPGA resources and the existence of idle instruction modules. If possible, the instruction module is loaded in an FPGA location not currently occupied by any other instruction module. If no empty hardware locations are available, a simple least-recently-used (LRU) algorithm is used to remove idle hardware. The host modifies the bit-stream of the requested hardware module to reflect the placement changes. The hardware module is then configured on the DISC platform by sending the new configuration to the system. Figure 9 provides a simplified flow chart of DISC instruction execution.

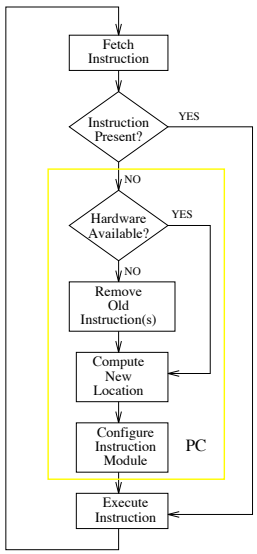

Figure 9: DISC Instruction Execution.

One drawback of partially configuring the device during run-time is the overhead caused by continually reconfiguring instruction modules. The current board configures the DISC processor by sending the configuration bit-stream one bit per bus transfer over the PC-ISA bus. Operating at a maximum transfer rate of 1.5 Mb/sec, the PC-host is capable of configuring one row in 600 us. This represents 4511 processor cycles or 1500 simple instruction executions for each row configured. By removing the current system board and bus limitations, configuration speeds improve by a factor of 64 and operate at the device maximum of 12 MB/sec.

Custom instruction modules should remain resident in the processor for long periods of time to decrease the reconfiguration overhead. In addition, custom instruction modules should provide enough performance improvement over a sequence of general purpose ALU instructions to justify the cost of reconfiguration at run-

time. The following application example will demonstrate this tradeoff.

7 Application Example

A simple image mean filter was developed as both a sequence of general purpose instructions and as an application specific hardware module to demonstrate the performance improvements gained by tailoring the hardware to the application. Both demonstrations calculate the mean value of each pixel in an image, $g(x, y)$, by obtaining an average over a 3x3 neighborhood as follows:

$$g(x,y) = \frac{1}{8} \sum_{m=-1}^{1} \sum_{n=-1}^{1} g(x+m, y+n).$$

A coefficient of $\frac{1}{8}$ was used to simplify the design. The 128 x 64 grey scale image in Figure 10 was used as the test image for both cases.

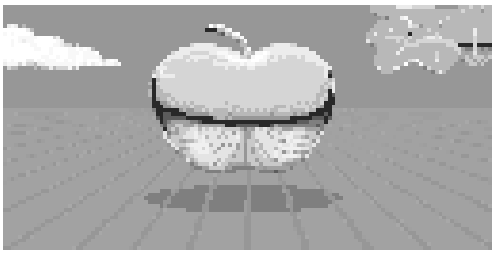

Figure 10: Original Test Image.

7.1 General Purpose Approach

The general purpose approach required four instructions not found in the processor core: add, subtract, shift, and enhanced addressing modes. These additional modules comprised a total of 8 rows, leaving 38 rows free for other custom instruction modules.

Execution of the algorithm centered in the inner loop calculation of the 3x3 neighborhood mean value. Calculating each pixel value involved individually adding each pixel of the neighborhood. Many of the instructions used for this summing operation involved address calculation and pointer manipulations. Computation of each pixel finishes with three shifts for the division by eight.

Complete processing of a pixel required an average 160 instructions or 560 clock cycles. Processing the complete image, including overhead, required 4.59 Mclocks or 610 ms (7.5 MHz).

7.2 Application Specific Approach

The application specific approach significantly improves performance of the algorithm by assuming control of address generation, buffering pixel values, and pipelining the arithmetic. With 31 rows of hardware, the extra registers, arithmetic operators and control logic consume significantly more hardware than the simple instructions used in the general purpose approach.

The MEAN instruction module calculates the average of a 3x3 neighborhood through the use of a sliding window as seen in Figure 11. Each numbered element of the sliding window represents a pixel register in the custom module. Instead of loading the entire window from memory at each pixel, register values are shifted to represent a sliding window (see Figure 12). Only registers 3, 6, and 9 are loaded at each new pixel.

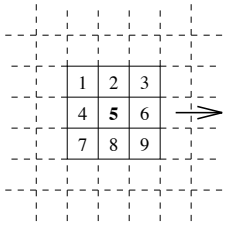

Figure 11: Sliding Pixel Window.

With the window registers loaded, the custom instruction module adds all nine pixel values in parallel with eight custom adders as seen in Figure 12. The division by eight is achieved by shifting the results three bit positions.

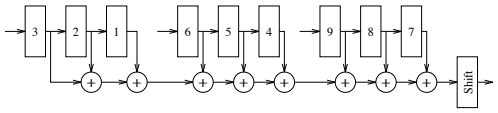

Figure 12: Dataflow of MEAN Instruction Module.

The MEAN instruction requires only 7 clock cycles to evaluate each pixel of the image. The clock cycles are scheduled as follows:

1. Load register 3
2. Load register 6
3. Load register 9
4. Wait (add delay to parallel add)
5. Write results to image memory
6. Calculate new address
7. Shift register window

Reducing the pixel calculation to seven clock cycles and eliminating much of the address calculation overhead reduces the clock count from 4.59M in the general purpose case to 57k for an 80 times speedup. Operating at 7.5 MHz, the image is filtered in 7.6 ms. Figure 13 displays the image filtered with the MEAN custom instruction.

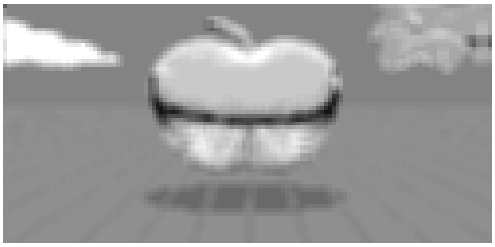

Figure 13: Test Image Filtered Through MEAN Custom Instruction.

7.3 Configuration Overhead

Because the cost of reconfiguring the application-specific instruction module is so high, configuration overhead must be considered when comparing the two approaches. The 31 row MEAN instruction requires an additional 140 kcycles for configuration, raising the total cycle count to 197 kcycles. The MEAN configuration overhead represents 71% of the total operating time. If device configuration speeds are maximized, this configuration overhead is reduced to 16% of the total operating time.

The extra four modules needed for the general purpose approach require only 36 kcycles for configuration. This represents less than 1% of the total operating time. When considering the high-cost of configuration in total operating time, the MEAN filter custom instruction provides a 23 times speedup to the general purpose approach (see Table 2).

	General Purpose	Application Specific
Rows	8	31
Operation Cycles	4.59M	57k
Raw Speedup	1	80
Area*Time	36.7M	1.8M
Configuration Cycles	36k	140k
Total Cycles	4.63M	197k
Actual Speedup	1	23.5

Table 2: Performance Comparison between General Purpose and Application Specific Approaches.

8 Conclusions

The DISC processor successfully demonstrates that application specific processors with arbitrarily large instruction sets can be be constructed on partially reconfigurable FPGAs. The relocatable hardware model improved run-time utilization of FPGA resources and the linear hardware model provided a convenient framework for relocating custom instruction modules. DISC demonstrates the general concept of alleviating density constraints of FPGAs by partially reconfiguring a device at run-time.

Although the techniques of partial configuration, relocatable hardware, and the linear hardware model were implemented as a general purpose processor, they offer similar advantages to other digital architectures. They may enhance the usefulness of FPGA co-processors by providing demand-driven computation. In addition, these techniques may allow FPGA based computing machines to operate in more dynamic environments such as multi-tasking operating systems. Any digital architecture that could benefit from demand-driven hardware may find these techniques useful.

References

[1] Algotronix, Edinburgh, UK. *CAL1024 Preliminary Data Sheet*, 1988.

[2] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, March 1993.

[3] Atmel, San Jose, CA. *Configurable Logic: Design & Application Book*, 1993-1994.

[4] R. Camerota and J. Rosenberg. Data acquisition systems using Cache Logic FPGAs. In *Configurable Logic: Design & Application Book*, pages 7.15–7–18. Atmel, San Jose, CA, 1993-1994.

[5] J. Davidson. FPGA implementation of a reconfigurable microprocessor. In *Proceedings of the IEEE 1993 Custom Integrated Circuits Conference*, pages 3.2.1–3.2.4, 1993.

[6] 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.

[7] B. S. Fagin. Quantitative measurements of FPGA utility in special and general purpose processors. *Journal of VLSI Signal Processing*, 6(2):129–137, August 1993.

[8] P. C. French and R. W. Taylor. A self-reconfiguring processor. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 50–59, Napa, CA, April 1993.

[9] X. P. Ling and H. Amano. WASMII: a data driven computer on a virtual hardware. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 33–42, Napa, CA, April 1993.

[10] P. Lysaght. Dynamically reconfigurable logic in undergraduate projects. In W. Moore and W. Luk, editors, *FPGAs: Proceedings of the 1991 International workshop on field-programmable logic and applications*, Oxford, England, September 1991. Abingdon EE and CS Books.

[11] P. Lysaght and J. Dunlop. Dynamic reconfiguration of FPGAs. In W. Moore and W. Luk, editors, *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications*, pages 82–94, Oxford, England, September 1993.

[12] S. Monaghan and C. P. Cowen. Reconfigurable multi-bit processor for DSP applications in statistical physics. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 103–110, Napa, CA, April 1993.

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

[14] T. G. Rauscher and A. K. Agrawala. Dynamic problem-oriented redefinition of computer architecture via microprogramming. *IEEE Transactions on Computers*, C-27(11):1006–1014, November 1978.

[15] J. Rosenberg. Implementing Cache Logic^{tm} with FPGAs. In *Configurable Logic: Design & Application Book*, pages 7.11–7.14. Atmel, San Jose, CA, 1993-1994.

[16] M. J. Wirthlin, B. L. Hutchings, and K. L. Gilson. The Nano Processor: A low resource reconfigurable processor. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 23–30, Napa, CA, April 1994.

[17] A. Wolfe and J. P. Shen. Flexible processors: a promising application-specific processor design approach. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture - MICRO '21*, pages 30–39, San Diego, CA, November 1988.