# Colt: An Experiment in Wormhole Run-Time Reconfiguration

Ray A. Bittner, Jr., Peter M. Athanas and Mark D. Musgrove

Virginia Polytechnic Institute and State University
The Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia 24061-0111

## ABSTRACT

Wormhole Run-Time Reconfiguration (RTR) is an attempt to create an refined computing paradigm for high performance computational tasks. By combining concepts from Field Programmable Gate Array (FPGA) technologies with data flow computing, the Colt/Stallion architecture achieves high utilization of hardware resources, and facilitates rapid run-time reconfiguration. Targeted mainly at DSP-type operations, the Colt integrated circuit – a prototype wormhole RTR device – compares favorably to contemporary DSP alternatives in terms of silicon area consumed per unit computation and in computing performance. Although emphasis has been placed on signal processing applications, general purpose computation has not been overlooked. Colt is a prototype that defines an architecture not only at the chip level but also in terms of an overall system design. As this system is realized, the concept of wormhole RTR will be applied to numerical computation and DSP applications including those common to image processing, communications systems, digital filters, acoustic processing, real-time control systems and simulation acceleration.

**Keywords**: Configurable Computing, FPGA, DSP, High Performance Computing, Data Flow, VLSI

## 1. INTRODUCTION

Many applications to which CCMs have been successfully applied tend to have three common properties: they make use of deep computational pipelines, are inherently data flow in nature, and adroitly exploit concurrency. Furthermore, in situations where the hardware resource requirements for an application exceed the available CCM resources, it becomes necessary to rapidly reconfigure (portions of) the platform to effectively time-share the hardware resources. Applications such as genomic sequence scanning,[1] FFT calculations,[2,3] text searching[4] and computer vision[5] consist of highly concurrent, spatially localized algorithms characterized by large amounts of data that must be processed in real-time. In order to further the capabilities of configurable computing machinery, it is only natural to capitalize on the strengths exhibited in contemporary platforms, and use these strengths to derive new architectures and computing paradigms. An intuitive path to pursue would be to reinforce the realization of deeply pipelined data flow[6] computations, while enhancing rapid run-time reconfiguration capabilities. Indeed, progress has been made in writing compilers to accomplish this.[7]

An introduction to the developmental background of CCMs can be found in a paper by Acosta, et. al.[8] The architectures of most contemporary CCMs, and the systems based upon them, have been implemented using RAM-based FPGAs. FPGA properties, such as single bit computational cells, global control structures, and loosely controllable routing resources prove troublesome when realizing efficient computational structures. Further, the amount of overhead required to configure many contemporary FPGAs can be excessive, which adversely impacts not only the density of computing elements in the circuit silicon, but also the reconfiguration time of these chips. Since faster reconfiguration time can translate directly into faster computation, the relatively slow reconfiguration times of many FPGAs is a hindrance to competitive computing speeds. Also, the speed of operation for designs mapped to most of these devices is slow compared to ASIC solutions, making comparisons less than favorable. Worse yet, the maximum attainable clock rate varies greatly with the design being implemented. Such variations can be disastrous for a real-time system and can prove vexatious for any rapidly reconfigurable system that is designed to maintain a processing schedule (such as a deeply pipelined system).

The Colt/Stallion architecture attempts to overcome many of these limitations while maintaining the computational flexibility of FPGA-based computing. The purpose of this paper is to describe the basic Colt/Stallion architecture and, to some extent, justify this approach. The architecture offers alternative concepts in the design of CCMs, including computation, communication and reconfiguration.

## 2. DATA FLOW IMPLEMENTATION

Large scale implementation of tagged-token dynamic data flow machines is expensive in terms of hardware and communications overhead. For many problems, the computational advantages of a data flow architecture can be achieved using a (locally) static direct communication graph. This is the approach taken in the Colt/Stallion architecture. By directly connecting functional units and guaranteeing that only one operand exists on an arc at a given time, the implemented data flow graph is reduced to a set of interacting pipelines. In this regard, the architecture resembles a Pipenet as discussed by Hwang, et. al.[9] Hwang describes the types of algorithms and looping structures that can be executed and the theoretical speedups that are achievable. [10] Colt extends these capabilities by providing special provisions for conditional execution, as well as a novel pipeline configuration scheme.

The hardware for such a system should consist of flexible functional units with fairly diverse interconnection abilities to allow subsections, or *chunks*, of the data flow graph to be directly mapped onto the system and to facilitate rapid swapping of chunks with little overhead to the active computation. Each chunk is a piece of the overall data flow graph that can fit within the limitations of currently available hardware resources. Operands arriving on arcs entering a particular chunk are queued by the system until the programming process is completed. Once the chunk has been configured onto the architecture, the queues feed operands into the pipelines, producing new results on arcs leaving the chunk. These are either fed directly into the next chunk, if it has already been configured, or are queued until part of the system has been configured with the next chunk. Thus, chunks can be dynamically programmed into the system at run-time, paging in and out as necessary much as an operating system pages sections of programs.

## 3. WORD VS. BIT ORIENTED APPROACH

The Colt/Stallion architecture is targeted towards DSP tasks. As such, most computation will be performed on word-wide operands as opposed to bit-oriented operations. To take advantage of this, the Colt has been designed with chainable 16-bit data paths and processing units throughout the chip. Whereas other FPGA type processors use a single bit data path, the bus wide grouping of bits in the data path of the Colt requires less control circuitry and switching to steer operands around the chip. The advantages of this are three fold. First, less silicon area is needed for routing than in a bit oriented design (routing is extremely resource-intensive in silicon). Second, less power is consumed for circuitry used to route the signals. Third, there is a higher predictability of propagation times for signals, making it possible to better pipeline designs. Following the same philosophy, the functional units on the Colt process operands which have sizes consisting of multiples of 16-bits; performing the same operation on all bits in a word. Since the functional units are larger, and are capable of a 16-bit computation every clock cycle, fewer units are needed than would be required in a bit-oriented design. Thus, higher speed operation can be achieved because fewer switches must be traversed to proceed from computation to computation. Further, the locality of the bits within a word allows for the construction of faster structures for computations that require high speed communication between neighboring bits. Addition, subtraction and shifts all exhibit this requirement.

Perhaps the greatest gains in using a word oriented approach lie in the number of configuration bits required. Since the same operation is performed over the entire 16-bit word, only 1/16 the number of configuration bits are required to specify the operation compared to a bit oriented architecture. Again, the gains from this are many fold. Fewer storage elements are required to store configuration bits; saving silicon area, power and routing resources n

## 4. ARCHITECTURAL OVERVIEW

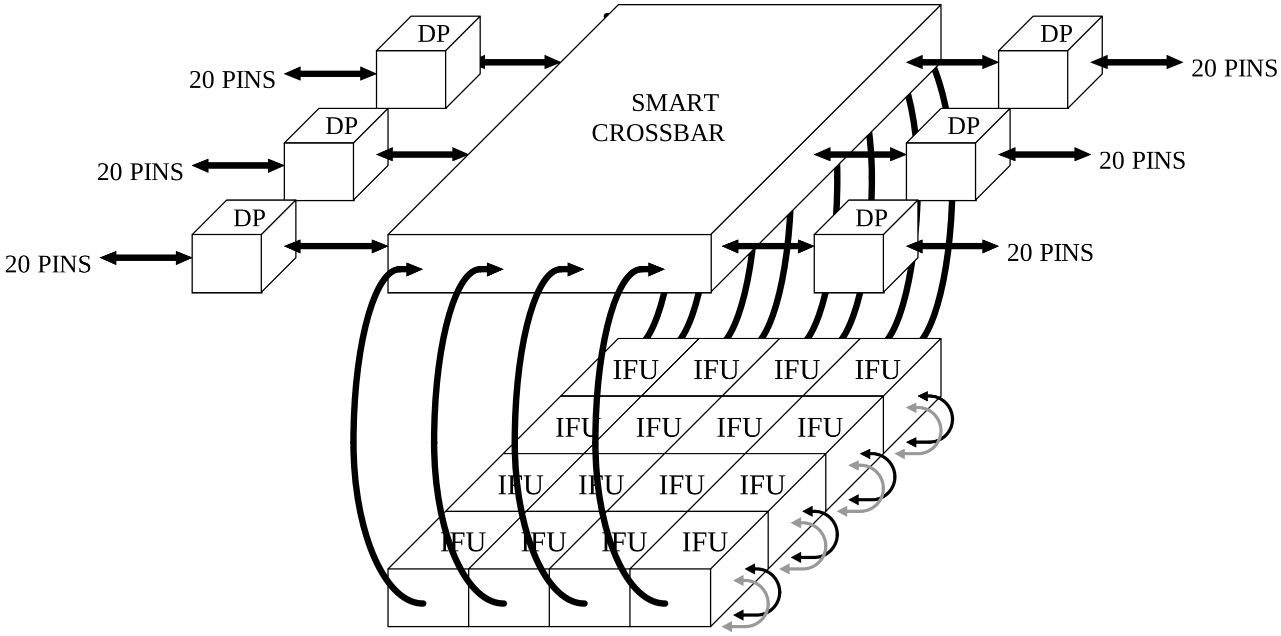

**Figure 1 - A diagram of the Colt architecture illustrating the major interacting blocks.**

As shown in Figure 1, there are four main subsystems in the Colt: data ports (DPs), the crossbar, the multiplier and the mesh. The mesh is further subdivided into Interconnected Functional Units (IFUs) and then Functional Units (FUs). With the assumption that computation will be performed at the word level rather than the bit level, the more traditional array of single-bit computational cells has been foregone in favor of a system that is more oriented toward configuring deep pipelines of integral numbers of word sized operands around the chip. While this could have been accomplished using a classic mesh architecture, the run-time reconfiguration of data flow graphs should benefit from the added connectivity present in the Colt. For example, some units of the Colt may be processing data while others are being configured with the next chunk. Ideally, the configuration of the next chunk would complete just before the inputs to that chunk were ready. The Von Neumann-like configuration/execution cycle would continue until the final outputs from the data flow graph have been computed. During the process however, hardware resources could become fragmented as different graph topologies are loaded onto the chip. In a pure mesh, a good planning strategy is to carefully predetermine the size and shape of all chunks that may need to coexist within the hardware resources at a given time for different possible execution scenarios. This has been done with contemporary FPGAs, such as the Altera FLEX and, to an extent, the Xilinx XC6200. In a diverse application this would inevitably lead to internal fragmentation within the allocation blocks of the mesh since each chunk would have different resource requirements. By using the crossbar and skip bus resources of the Colt, this situation can be avoided since it is possible to reach any part of the chip once the data has reached the crossbar. Thus, the processing resources needn't be allocated in blocks or subsections. Rather, different chunks can coexist in the mesh, in almost any random pattern. The exact resource allocation distribution can even be determined at run-time.

### 4.1 Data port

In the Colt chip prototype, there are six bi-directional data ports. Each is 20 pins wide, consisting of 16 bi-directional pins for operands and four bi-directional pins for stream flow control. Each port supports both read and write operations as well as five modes of operation; two of which are strictly used for programming. The three data modes differ mainly in the operation of the port when it is in input mode. These modes were created with possible styles of usage in mind. One style of use would be to view a Colt design as a relatively static pipeline for dedicated use in real-time systems. In this case, data could be fed into the chip at the acquisition rate and processed with a guaranteed latency. No complex flow control is necessary as only simple conditional execution is required and no reconfiguration is required. *Raw Mode* is designed for this task. It accepts data unconditionally and merely injects it directly into the chip. Another style of use would be in a more complex system where run-time reconfiguration occurs frequently and flow control would be required to prevent data loss during the programming latency. Such a system may still have relatively simple conditional execution and so requires only simple queuing of the data operands. This style of use is addressed by *Synchronization Mode*, which helps align operand data as it enters so that the operands are aligned when two intersecting pipelines meet within the chip. Operand acceptance is synchronized with any subset of the other data ports on the chip. The third style of use would involve run-time reconfiguration coupled with complex looping structures and conditional execution. The corresponding data port mode operates much as the second with the additional constraint that it allows only one data operand to enter each of the data ports in the subset at a given time. No new data is allowed to enter until valid results have left the chip. This has been dubbed *Loop Mode*.

### 4.2 Crossbar

The on-chip crossbar network is the primary means of deep pipeline construction. In the Colt, the crossbar is organized as a 12-input, 16-output intelligent network supporting 16-bit data paths. A single 16-bit input arrives from each of the six data ports and the bottom of each column in the computational mesh. Two additional inputs come from the high and low words of the output of the dedicated multiplier. Single outputs go to each data port and double outputs go to the top of each column of the mesh as well as to the multiplier. The crossbar provides nearly full connectivity from any input to any output. The only exception to this being that direct connections from one data port to another are not supported. Such connections would be a waste of chip resources in that all pipelines that consume I/O resources should perform some type of computation before leaving the chip. The construction of the crossbar is specialized to support the Wormhole RTR concept, but the details of implementation are beyond the scope of this paper.

### 4.3 Mesh

In the prototype chip, the computational mesh consists of a 4x4 array of IFU cells. While it would have been preferable to attach each IFU to the crossbar individually, the increase in crossbar size using contemporary planar VLSI technology was prohibitive. The top of each column of the mesh receives two inputs from the crossbar. One output from the bottom of each column is fed back into the crossbar. In most applications, some degree of data reduction will be performed within the mesh as pipelines interact, thus justifying the reduced output connectivity. Each IFU has four nearest neighbor connections. IFUs along the top of the mesh receive their northern local inputs and skip bus connections from the crossbar. First row IFU northern local outputs are left unconnected. The eastern and western edges of the mesh are connected as if they were nearest

### 4.4 Functional Unit

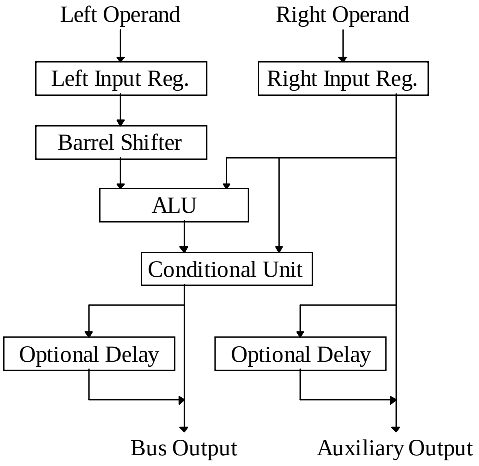

**Figure 3 - Functional unit (FU).**

The IFU is the reconfigurable computational core of the Colt. It consists of the FU surrounded by the control and buses necessary to provide the nearest neighbor and skip bus connectivity. The FU has 16-bit left and right input registers, each of which can load an operand from any of the four nearest neighbor connections or from any of the four skip bus segments connected to the IFU. The left input register feeds into a shifter than can conditionally shift by 1, 2, 3 or 4 bits to the left or 1 bit to the right. The shifter facilitates the implementation of multipliers and floating point arithmetic (to be discussed later). The output of the shifter and the right operand feed into a programmable ALU. The ALU is based on a Propagate, Generate, Result structure as found in Mead and Conway.[11] It can be programmed to perform any binary operation of two bits between the two words. Also, there is a carry path that can be used to allow the unit to perform addition, subtraction, negation, shifting, etc. The output of the ALU and the right operand then feed into the conditional unit which can be used to select between these two inputs based on the value of the conditional flag. Finally, the output of the conditional unit passes through an optional output delay before being release to the rest of the chip on the four nearest neighbor connections.

The output delay is used to help synchronize the execution path lengths of different pipelines so that the correct pairs of operands are matched at the right times where pipelines meet. Normally, a result is produced every clock cycle within the IFU, but when the output delay is enabled a two clock cycle latency is incurred. Again, an architecture with smaller computational elements could afford to sacrifice units for this purpose, but in a larger grained system such as Colt it is more cost effective in terms of area to include some provisions to avoid the loss.

An auxiliary output from the FU merely forwards the value of the right input register. This capability is valuable for floating point arithmetic and assorted signal processing computations. Each of the four skip bus segments attached to the IFU can independently select direction to be either an input or an output. If the output direction is chosen, the value sent can be either the normal FU output, the auxiliary output, the value of the compass opposite skip bus segment or the value of the compass right skip bus segment.

Three flags also enter and leave the IFU, each of which has similar routing capabilities to the normal data paths within the mesh. These flags are used for the carry in and carry out of the ALU, the shift in and shift out of the shifter and the input and generated conditional flag. The conditional flag can be generated from several sources including sign bits, carry out and the lower bits of the right operand for use in controlling the shifter. By using this one bit signal to affect the data path and computation within the FU, conditional execution can be controlled both by that FU and by other FUs. This includes the action of the shifter and the selection chosen by the conditional unit.

Also, special provisions have been made within the FU to accommodate multiplication. As documented by Magenheimer, et. al.,[12] multiplication by a constant can be reduced to optimal shift/add trees. Multiplication by five, for example, can be expressed as 4x + x, which is merely a shift left by two followed by an addition. The structure of the FU allows this operation to be performed in a single clock pulse. Note that if a series of values must all be multiplied by the same value, then a pipeline can be reconfigured at run-time to multiply by that constant with very little overhead. Indeed, since each FU is capable of performing a shift and add in a single clock pulse, multiple multipliers could all be configured on the Colt simultaneously. The exact number of multipliers possible would be dependent on the multiplicative constants desired. The extreme on the prototype chip would be the realization of 16 multipliers in full operation simultaneously within the mesh plus the dedicated multiplier for a total of 17 chip wide.

Further, floating point arithmetic has been considered in the construction of the FU. Input sources for the generation of the conditional flag have been included to aid in operations such as mantissa alignment and normalization. These can be used to control a conditional shift of the mantissa depending on the value of an exponent loaded into the right input register or to renormalize a mantissa after an addition operation. The same conditional flag can be forwarded to another FU that would then increment or decrement the corresponding exponent accordingly.

### 4.5 Multiplier

Inevitably, circumstances will arise when the multiplicative constant will need to change too quickly to justify the overhead of reconfiguring an FU to perform the operation. For these occasions, such as when performing a dot product, it was deemed necessary to incorporate a distribution of integer multipliers. In the Colt prototype, a single integer multiplier is made available via direct connection to the crossbar. While a multiplier could have been constructed from the mesh of FUs, the relative size of such a multiplier compared to the size of a dedicated version, added to the frequency of multiplications in DSP type operations drove the decision to include this unit on the Colt. The pipelined multiplier accepts 16-bit operands and produces a 32-bit result in two clock periods. Since the multiplier inputs and outputs are directly connected to the crossbar, the results can be quickly routed to any part of the chip for further processing.

## 5. CONDITIONAL EXECUTION

Conditional execution can be achieved on Colt using several different mechanisms. The appropriate mechanism or set of mechanisms to use depends on the characteristics of the branch in the data flow graph being processed.

### 5.1 Chunk Swapping (Blocked Conditionals)

In the data flow graph implementation of a typical IF-THEN style conditional statement, operands are directed to both alternative paths of execution. Results are calculated for both execution paths and then the correct result, as determined by the condition, is chosen. If the graph to be executed is such that all operands are exclusively processed on one side or the other of the conditional then the unused side of the conditional graph need never be configured onto the chip. Chunk boundaries can be drawn at the beginning and end of both sides of the conditional and only the side that is to be executed would be programmed into the chip at run-time (i.e. partial run-time reconfiguration). An example of such a situation would be two alternative formulas to be applied to a data set. The chip would only need to be configured with the formula to be applied in a given situation.

### 5.2 Valid Bit (Interleaved Conditionals)

The valid bit is the most basic means of handling stream discontinuities. This is a flag bit that is passed along as a seventeenth bit with every operand on the chip. If this bit is set the data is considered valid and if the bit is not set the data is considered invalid and worthless. When the valid bit passes through an FU the valid bit of the result can be calculated using a number of different means. The most basic is to logically AND the valid bits of the two input operands and use this as the valid bit of the result. The multiplier always uses this method. Another option in the FU is to use the conditional flag as the valid bit of the calculated result. There are other options that are included as conveniences for certain processing tasks.

The valid bit of one of the input operands can be used as the conditional flag for a given FU and so it can control the selection function of the conditional unit. This makes it possible to feed two data streams containing valid and invalid data into an FU and have it multiplex the two together to form a stream consisting entirely of valid data. This function can then be combined with other FUs to evaluate both sides of a conditional, in true data flow fashion, flag data on the appropriate side as being invalid and then merge the two sides of the conditional again to form a single data stream of valid results that can be used for further processing.

The valid bit is also used to implement flow control on the Colt. The Transmit pin on the data ports is used to flag valid and invalid data entering and leaving the chip. Because the pipeline through the chip can be almost arbitrarily long it is difficult to halt the entire pipeline in a single clock period. A global control mechanism could be used for this purpose, but instead the valid bit is used as it is more in keeping with a distributed control philosophy. Whenever a data port isn’t injecting valid operands into the system, it sends data items that are flagged as being invalid. This may happen because the system hasn’t begun filling the pipeline or because there is a gap in a data stream that has caused the synchronization mechanism to block the flow of data. In any event the invalid data is clocked through the pipeline forming a “bubble” of invalid operands and results in the system. When the invalid data eventually reaches an output data port it is flagged as invalid to the external memory and so is not stored.

### 5.3 Loop Mode (Iterative Conditionals)

There are circumstances under which neither of the previous two methods would suffice for conditional execution in a deeply pipelined system. Such situations are found in data-dependent iterative and recursive function evaluation. Calculation of a factorial would be an appropriate example. In these situations, the problem of unknown execution time challenges the Colt programmer. The pipeline cannot be stalled to await final evaluation of a given function. Therefore, new operands that enter the system could either be discarded or would need to be injected into a running calculation; thus destroying the results. To combat this problem, loop mode has been implemented using the data ports. This mode allows a single valid operand to enter each input data port and then blocks all other data from entering the chip. A loop may then be constructed within the chip and the single set of valid operands may circulate through it as many times as necessary. When processing is completed a calculation can flag the final result as being valid and allow it to leave the chip through an output data port. When this occurs the output data port sends a signal to all input data ports so that each allows another valid operand to enter the chip and begin the looping process. The process continues for as long as necessary. This is not an optimal solution to the problem; however, it was an adequate compromise given that this situation was not expected to occur often in the types of operations to be performed by Colt. Further, a more elegant solution would required significantly more hardware to implement, and so the practicality of creating such a solution was questioned.

## 6. PERFORMANCE CHARACTERISTICS

Performance results of the Colt to-date are all based on digital and analog simulations as the chip is going through the final phases of layout and has been submitted for fabrication to MOSIS. A 3-metal, 0.8 μm CMOS process is being used on a 7 mm square die with 132 pins. Under SPICE simulations, the chip operates at 50 MHz, achieving a collective I/O bandwidth of 4.8 billion data bits per second through the six 16-bit data ports. In terms of computational power, the pipelined multiplier produces a result every clock cycle for a total of 50 million 32-bit results per second. The computational power of the mesh is harder to quantify since each FU can perform several operations in a single clock period including an ALU operation, a shift and a multiplexing operation. The exact subset of the functions used for a given application would vary, but a range can be established of one to three operations per clock pulse. This gives a raw computational power of 800-2400 million 16-bit operations per second. In the extreme case where all 16 FUs could be used to perform multiplication, a performance of 850 million multiplications per second could be achieved. These numbers are considered to establish a lower bound for this type of computation as Colt is merely a test bed for concepts that will reach their full fruition in the full scale, 0.5 μm version: Stallion.

## 7. CONCLUSIONS

Colt is the result of a compromise between proof-of-concept testing of the various subsystems and layout practices and the production of a chip with useful computational power. After learning the lessons that can only come from the full implementation of a real world system, a second chip, called Stallion, will be produced which will feature enhancements beyond the Colt prototype. Countless lessons have already been learned from the implementation of Colt.[13] These include issues with layout, simulation, synthesis, management and chip planning. Nonetheless, the prototype Colt system is expected to provide the necessary computational boost for a number of experimental wireless communication applications currently under investigation at Virginia Tech. [14] This work was funded by DARPA grant J-FBI-94-219.

## 8. REFERENCES

[1] E. Lemoine and D. Merceron, "Run Time Reconfigurations of FPGA for Scanning Genomic DataBases," *IEEE Symposium on FPGAs for Custom Conputing Machines*, P. Athanas and K. Pocek, eds., pp. 90-98, IEEE Computer Society Press, Los Alamitos, California, 1995.

[2] N. Shirazi, P. Athanas and L. Abbott, "Implementation of a 2-D Fast Fourier Transform on a FPGA-Based Custom Computing Machine," *Fifth International Workshop of Field Programmable Logic and Applications*, Lecture Notes in Computer Science 975, pp. 282-292, Springer-Verlag, Oxford, England, September, 1995.

[3] S. Guccione and M. Gonzalez, “FFT on reconfigurable hardware,” *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, J. Schewel, eds., pp. 30-41, SPIE - The International Society for Optical Engineering, Bellingham, Washington, 1995.

[4] D. Pryor, M. Thistle and N. Shirazi, “Text Searching on Splash 2,” *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, D. Buell, and K. Pocek, eds., Napa, California, April 1993.

[5] W. King, T. Drayer, R. Conners and P. Araman, “Using MORPH in an Industrial Machine Vision System,” *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, J. Arnold and K. Pocek, eds., Napa, California, April 1996.

[6] A. Veen, “Dataflow Machine Architecture,” *ACM Computing Surveys*, Volume 18, Number 4, pp. 365-396, December 1986.

[7] J. Peterson, R. O’Connor and P. Athanas, “Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures,” *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, California, April 1996.

[8] E. Acosta, V. Bove, Jr., J. Watlington and R. Yu, “Reconfigurable Processor for a Data-Flow Video Processing System,” *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, J. Schewel, eds., pp. 83-91, SPIE - The International Society for Optical Engineering, Bellingham, Washington, 1995.

[9] K. Hwang, *Advanced Computer Architecture*, pp. 442-446, McGraw-Hill, 1993.

[10] K. Hwang and Z. Xu, “Multipipeline Networking for Compound Vector Processing,” *IEEE Transactions on Computers*, pp. 33-47, Institute of Electrical and Electronics Engineers, New York, 1988.

[11] C. Mead and L. Conway, *Introduction to VLSI Systems*, pp. 150-154, Addison-Wesley, 1980.

[12] D. Magenheimer, L. Peters, K. Pettis and D. Zuras, “Integer Multiplication and Division on the HP Precision Architecture,” *IEEE Transactions On Computers*, Vol. 37, No. 8, pp. 980-990, August 1988.

[13] R. Bittner, “Development and VLSI Implementation of a High Speed Data Flow DSP Computing System,” Ph.D. Dissertation, Bradley Department of Electrical and Computer Engineering, Virginia Tech, 1996, work in progress.

[14] R. Bittner and P. Athanas, “Stream-Based Processing Using Wormhole Run-time Reconfiguration,” in preparation.