Computing Kernels Implemented with a Wormhole RTR CCM

Ray A. Bittner, Jr. and Peter M. Athanas,
Virginia Tech
Department of Electrical and Computer Engineering
Blacksburg, Virginia 24061-0111

ABSTRACT

The Wormhole Run-Time Reconfiguration (RTR) computing paradigm is a method for creating high performance computational pipelines. The scalability, distributed control and data flow features of the paradigm allow it to fit neatly into the Configurable Computing Machine (CCM) domain. To date, the field has been dominated by large bit-oriented devices whose flexibility can lead to lowered silicon utilization efficiencies. In an effort to raise this efficiency, the Colt CCM has been created based on the Wormhole RTR paradigm. This paper outlines methods of implementation and performance for several common operations using these concepts. They serve as indicators of the diversity of algorithms that can be instantiated through the high-speed run-time reconfiguration that these devices make possible. Particular attention is paid to floating point multiplication. Also discussed is the topic of data dependent computation which would seem to be counter intuitive to the Wormhole RTR paradigm. The paper concludes with a summary of performance of the three computations.

1. INTRODUCTION

The varieties of CCM platforms, the application domains, and the aggregate of CCM practitioners are all growing at a rapid rate. Despite the challenges that contemporary CCM platforms face, namely development software, dynamic resource allocation strategies, and hardware/software coherence, the computational benefits of CCM platforms often prevail over conventional alternatives. Much progress has been made in CCM software; however, software alone cannot overcome all of the challenges mentioned above. It is also necessary to explore alternative architectures and computing paradigms which will not only lead to improvements in computational performance and density, but also facilitate the application development process and system integration. To date, there have been a number of notable endeavors set out to explore various device-level and system-level architectures [1,2,3,4]; yet, since this field is still in its infancy, there are many unexplored avenues that warrant investigation.

Colt is an experimental FPGA-like integrated circuit designed for configurable computing. Unlike

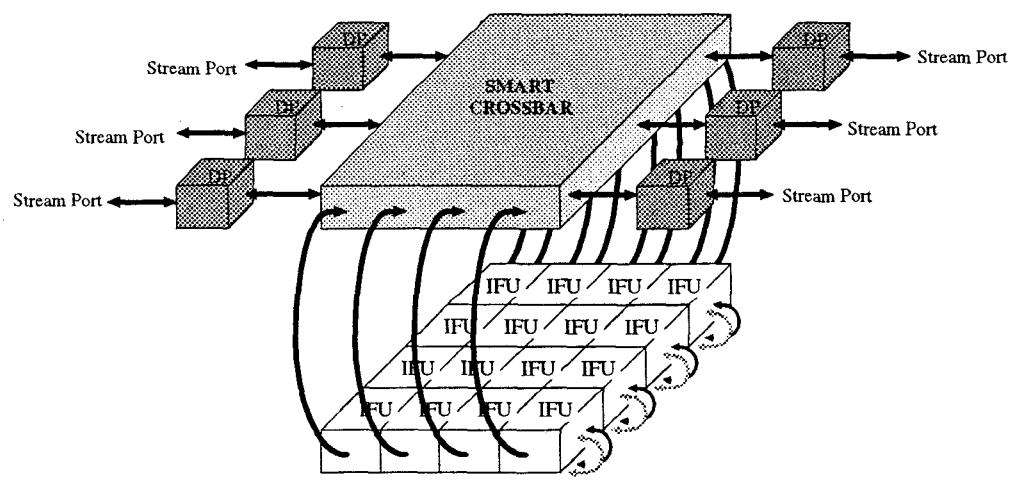

Figure 1: The basic architecture of the Colt CCM, including Interconnected Functional Units (IFUs), Data Ports (DPs) and the Multiplier (MULT).

0-8186-8159-4/97 $10.00 © 1997 IEEE

Authorized licensed use limited to: MICROSOFT. Downloaded on October 8, 2009 at 12:15 from IEEE Xplore. Restrictions apply.

contemporary bit-oriented FPGAs, the Colt CCM features both word-wide and bit-wide pathways, targeted toward digital signal processing applications. As a research endeavor, Colt attempts to explore a number of areas that are not currently possible using commercially available devices. Design attention has be focused on attaining high operational clock rates and shortened reconfiguration times. Although Colt is intended for use in a large, run-time reconfigurable (RTR) environment, this paper illustrates aspects of Colt through the use of simple static applications. By focusing on static applications, the advantages of the word-oriented architecture can be compared independently of the rapid run-time reconfiguration features of the Colt device. Rudimentary computational blocks, such as the ones presented in this paper, coupled with rapid RTR serve as the basic framework for the final computing platform.

2. COLT / STALLION ARCHITECTURE

The Colt CCM is based on the Wormhole Run-time Reconfiguration (RTR) paradigm [5]. Wormhole RTR is a method of creating custom computational pipelines and interconnection networks through a pool of configurable resources in a distributed fashion. These pipelines are constructed using *streams*, -- a sequence of words containing both configuration information and computational data. The streams are injected into the resource pool and then independently steer themselves; programming the functionality of each resource that they encounter. The steering and programming information is contained in the stream header, which is broken up into discrete packets containing programming information for individual units. Each packet is peeled off of the front of the stream by the resource it programs as the stream proceeds through the resource pool. After the header, the remainder of the stream consists of the data to be processed through the pipeline.

Figure 2 illustrates the basic principle of Wormhole RTR. In this figure, a stream is presented to the boundary of allocatable computing resources. The front of the stream (right-hand side) contains the stream header consisting of a variety of packets (*P1* through *P5*). A computational pathway is created as the stream tunnels through the resource pool. Note that the size of the stream header shrinks as the stream progresses through the platform.

The architecture of the Colt CCM (Figure 1) has been designed to support the Wormhole RTR paradigm. It consists of a pool of computational resources including 16 Functional Units (FUs) and a 16-bit x 16-bit unsigned multiplier. A 4x4 cylindrical mesh into which the FUs are embedded exemplifies local routing resources on the chip. A segmented bus, called the Skip Bus, augments the connectivity of the mesh with an additional link between nearest neighbors that can be configured to allow data to flow in either direction between them. These additional segments can be connected so that signals can skip over several underlying FUs in a single clock pulse; allowing them to overcome the inherently planar mesh topology. A 12-input by 16-output crossbar accepts a single output from the bottom of each column of the mesh and provides two inputs into the top of each column. Up to 12 independent streams can simultaneously establish connections through the crossbar by virtue of 156 distributed state machine. Crossbar outputs also route directly into the two 16-bit inputs of the multiplier and the upper and lower 16-bit words of the result have independent paths back into the crossbar. The off-chip I/O resources are represented by six 16-bit bi-directional data ports, each of which has dedicated connections to and from the crossbar.

A stream can be injected into any of the data ports. The first packet is stripped from the header and is used to program the behavior of the data port after configuration. From there, the stream continues to the crossbar, where the second packet is taken from the header and is used to program the output from which the stream will emerge. At that point, the stream can continue to the multiplier, to one of the mesh inputs, or to all of the mesh inputs simultaneously using broadcast mode; the destination is entirely dependent on the configuration desired. A more complete exposition of the Colt/Stallion architecture and computing model can be found in [6].

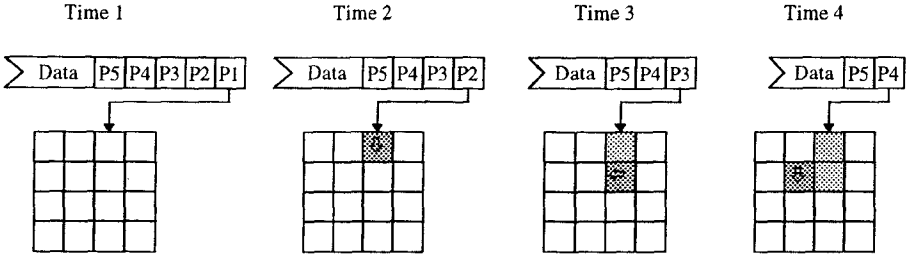

Figure 2 - In this two-dimensional depiction, wormhole run-time reconfiguration uses independently guided streams to configure a resource pool.

Authorized licensed use limited to: MICROSOFT. Downloaded on October 8, 2009 at 12:15 from IEEE Xplore. Restrictions apply.

3. COLT COMPUTATIONAL KERNELS

As mentioned in the introduction, Colt has been designed and optimized specifically to target digital signal processing applications. In this section, a few select example computational structures are presented which are themselves building blocks of higher signal processing functions. The basis of selecting the DSP kernels is familiarity, diversity of computational requirements and diversity of I/O requirements [7]. These structures are represented in their *static* form. It should be noted that one of the key features of Wormhole RTR is the rapid partial reconfiguration capability. These features can only be effectively evaluated in much larger applications in which these kernels are dynamically paged into and out of the platform. The advantages of rapid partial reconfiguration have already be quantified in a number of applications [8,9,10], and for brevity, will not be expounded upon here. Each of these applications fits in a single Colt device with no need for run-time reconfiguration.

This section details the implementation of three different functions on Colt. Starting with a simple example of the implementation of a dot product, the section introduces basic concepts and eccentricities of Colt programming. The implementation of a floating point multiplier is then explored, giving a floating point format that can be easily manipulated using the computing resources of Colt. This example introduces the concept of stream programming and illustrates some of the unique problems that arise due to architectural constraints. The section concludes with an example of conditional execution -- the calculation of an integer factorial.

4 DOT PRODUCT EXAMPLE

The dot product of two vectors is a common kernel found in a variety of signal processing, computer graphics, and system modeling problems. There are many methods of implementing this function on the Colt architecture. Which method is the most optimal choice is dependent on several factors, including the amount of available resources, the length of the vectors, the desired speed of execution, the amount of latency that can be tolerated, the choice of floating or fixed point arithmetic and the number of bits of precision used. For purposes of the first example, fixed-point arithmetic using the fastest execution method possible will be assumed. One possible implementation is shown in Figure 3.

The stream concept is used to the fullest in this implementation. The vector elements are presented sequentially and in order to the Colt data ports. The data values pass from the data ports through the crossbar to a built-in multiplier. The pipelined multiplier requires two cycles to evaluate each product and then passes it on through the crossbar to an FU that has been configured as a summing node. The partial sums are sent through the crossbar to a data port, and from there the partial sums are written to external memory in sequential order. The last value written out will be the final sum. Note that the delta delay for each pipelined stage is given in clock cycles next to each component. There are several operating details that are hidden by this diagram; pertinent issues are identified here, yet refer to [11,12] for a more complete analysis.

In this particular implementation, the stream for each input vector finds its way to the Colt chip boundary. In order for the pipeline calculation to be correct, the a_i's must be matched with the proper b_i's. Depending on the programmed mode of the data ports, different flow control measures can be taken. Raw Mode makes no attempt at synchronizing the inputs to the various ports, while Synchronization and Loop Modes do provide this functionality.

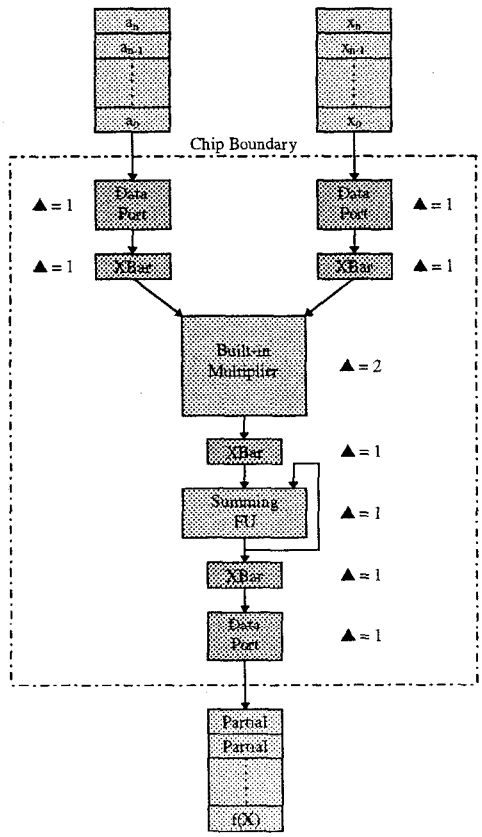

Figure 3: Dot Product Implementation #1.

Authorized licensed use limited to: MICROSOFT. Downloaded on October 8, 2009 at 12:15 from IEEE Xplore. Restrictions apply.

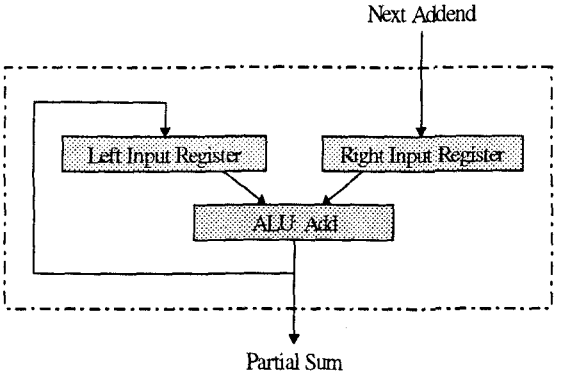

Figure 4: Summing FU Logical Diagram.

Another hidden detail in Figure 3 is the operation of the *Summing FU*. Figure 4 shows a logical depiction of the physical configuration of the *Summing FU*. The *Next Addend* is latched into the *Right Input Register* on each clock pulse. The *Right Input Register* is added to the *Left Input Register* and the new partial sum is latched back into the *Left Input Register*, hence the *Left Input Register* acts as an accumulator. The partial sum is also sent to the primary output of the FU. For extended precision, the accumulation function can be programmed to span multiple FUs. In this particular implementation, the *Left Input Register* must be initialized to zero prior to the start of each dot product computation. In the Colt, this problem can be solved by creative use of the *valid* bit, which accompanies each data item throughout the chip.

Note that in this example, if the vectors are n words long, one output word is produced every n clock pulses (the data is produced at a rate $1/n$ of the rate in which the inputs are supplied). Because of this, it is important to configure Colt to validate only the final summand at the proper time in the output stream. Once again, this can be done with creative use of the valid bit computations, along with the chip data ports. For the sake of brevity, the details of which are left in [11], and only the final configuration is presented here.

5 Floating Point Multiplication Example

Even though it has been targeted primarily to integer computations, Colt will be required to perform floating point operations. Provisions have been made within the Functional Units to ease the burden when floating point mathematics is required. To demonstrate these capabilities, the example of floating point multiplication will be addressed. There are three main steps to consider: mantissa multiplication, addition of exponents and re-normalization.

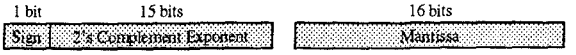

Figure 5: Floating Point Format used in this example.

The floating-point number format demonstrated here (shown in Figure 5) is split into two separate 16-bit words so that they can be easily split into two separate data streams. Floating point representations often use a biased exponent representation instead of a true two's complement form so that an unsigned comparator can be used to determine relative numeric magnitudes; however, signed and unsigned comparators have the same cost on the Colt, so the advantage of biased arithmetic is negated. Further, biasing forces the bias to be added and subtracted during processing, which would require added resources. Another often used floating point technique is the assumption of an implied binary digit 1 to the left of bit 15 of the mantissa. This technique will not be used here for the sake of simplicity. It will be assumed that the mantissa is always normalized so that the leading bit is always a 1. The sign bit for the represented quantity is stored in bit 15 of the exponent.

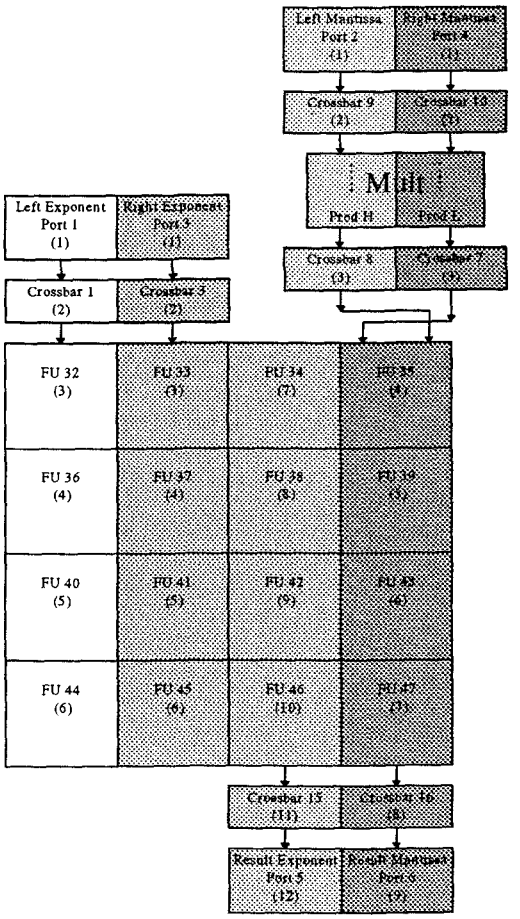

Figure 6: Floating Point Multiplier Programming Method.

Authorized licensed use limited to: MICROSOFT. Downloaded on October 8, 2009 at 12:15 from IEEE Xplore. Restrictions apply.

Four streams are required for the floating-point multiplier and the programming path followed by each of these is shown in Figure 6. Two floating-point values are presented to the Colt chip boundary. The "Left" value is injected through Ports 1 and 2 and the "Right" value is injected through Ports 3 and 4. Two of the streams carry the final result out of Ports 5 and 6. The four streams are color coded for clarification. Each box indicates a unit that must be traversed by a stream. The number in parenthesis on the last line of each box indicates the order in which the stream programs the unit.

proceeds to configure the third column, branching at FU 33.

The full floating point multiplier is shown in Figure 7. Solid lines indicate that a local connection is used. A dotted line indicates that the Skip Bus is used to pass the value. The path taken by the pair of mantissa values is the simplest, and will be the only one examined in detail. The mantissas enter through *Ports 2* and *4* and proceed directly to the multiplier. After two clock cycles, a 32-bit result is produced. The leading bit of the 32-bit mantissa is tested, and if the result is 0, the mantissa is shifted once to the left and the exponent must be

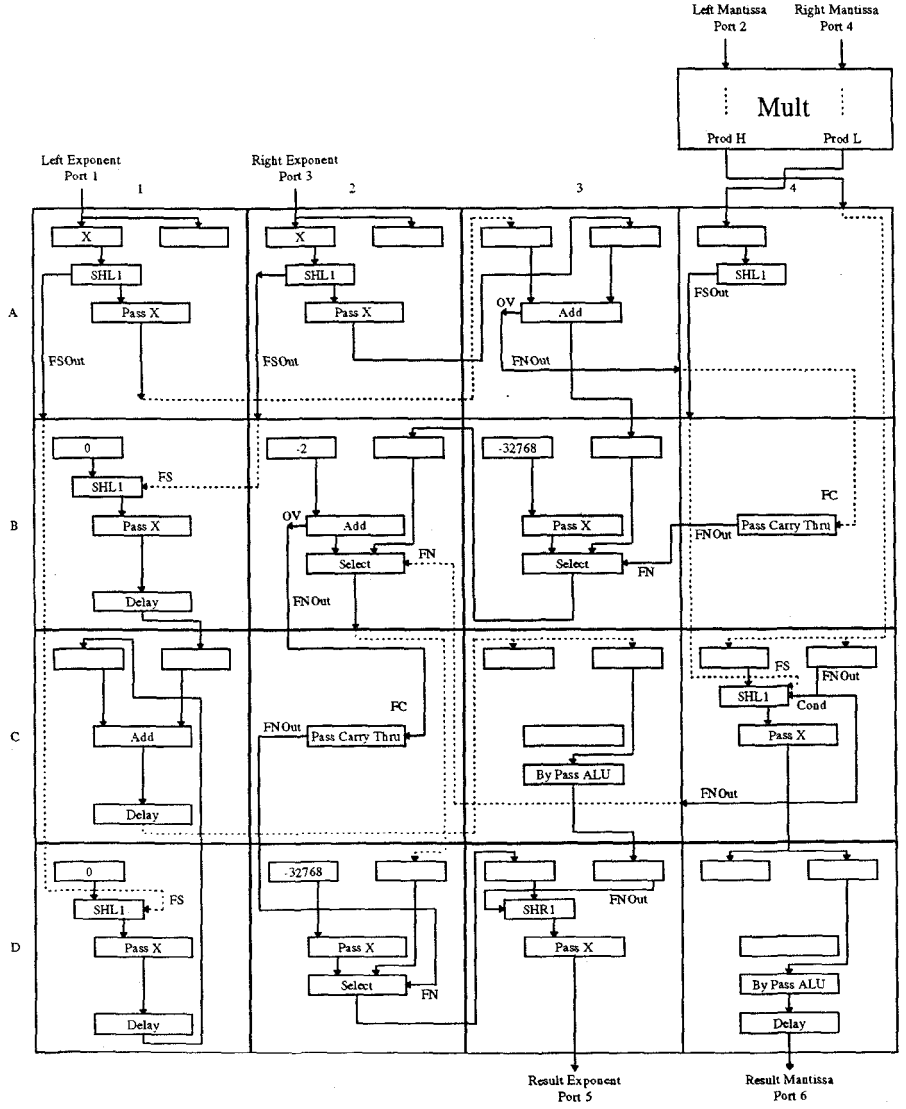

Figure 7: Floating point multiplier structure.

Divergent stream support is also a feature of the Colt. A divergent stream is define to be one that configures two or more different data paths emanating from a single node. This feature is evident in the configuration of the center columns where that stream first programs the second column of the mesh and then

decremented by 1 for proper normalization.

The right hand column of the mesh is configured to perform the re-normalization function (FUs A4, C4 and D4). The low word of the 32-bit result is latched into the *Left Input Register* of FU A4, where it is shifted once to

Authorized licensed use limited to: MICROSOFT. Downloaded on October 8, 2009 at 12:15 from IEEE Xplore. Restrictions apply.

the left. The bit shifted out may need to be shifted into the low bit position of the high word of the 32-bit result if normalization is required. This shifted bit is sent down the *Skip Bus* to FU C4, where it is supplied to the *Barrel Shifter* as the bit to be shifted in if re-normalization is required. The high word of the 32-bit result is latched into both input registers of FU C4 at the same time that the low word is latched into FU A4.

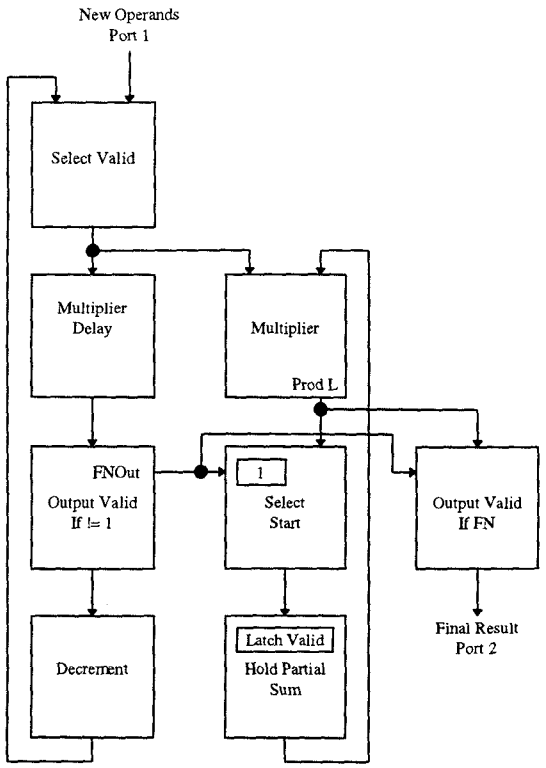

Figure 8: Conceptual Factorial Implementation.

The exponents of the floating-point values take a more convoluted path through the mesh. Each sign bit must first be separated from its exponent so that the sign bits may be XORed together to produce the sign bit of the result. The exponents must be added together and then tested to determine if an arithmetic overflow occurred. If an arithmetic overflow is detected, the number is either too large or too small to be represented and the exponent of the result will be reset to the smallest possible value. After testing for overflow, the exponent may be decremented once more if the mantissa multiplication result requires re-normalization. This, too, is tested for arithmetic underflow, and is adjusted back to the smallest possible exponent if it occurs. Then the sign and exponent of the final result are reunited and the upper word of the final result is sent out through *Port 5*.

6 Factorial Computation

The implementation of conditional execution structures within the Colt architecture can be challenging because of the deep level of understanding required. A full understanding of the valid bit, conditional logic and the data port modes is necessary. In an attempt to clarify the design process, the calculation of a 16-bit integer factorial function will be detailed in this section as an example.

The block level design of the factorial function is shown in Figure 8. The calculation of the factorial function is a data dependent looping problem because the number of multiplicative iterations required is dependent on the input operand. The implementation shown can really be considered as two separate executing processes that must exchange data at timed intervals. For purposes of explanation, all blocks shown in Figure 8 can be considered to incur a single unit delay of latency with the exception of the *Multiplier* and *Multiplier Delay* blocks, which each require two. In the actual implementation shown in Figure 9, the number of cycles of delay varies in order to match the delay constraints required by the hardware to those required by the algorithm. The path equalization process is a key component of design when using Colt, and is arguably the trickiest as well.

The functional blocks show the first process in the left column of the figure. In these blocks, the input operand is decremented until it reaches 1. Because this function has an unknown execution time, a collision free pipeline cannot be constructed. The alternative is to use the data ports in Loop Mode so that they will guarantee that only one valid operand exists in the pipeline at any given time. The *Select Valid* block shown acts as a multiplexer that will switch to forward whichever of the two inputs are flagged as being valid. The pipeline is initialized with only invalid data and then the first operand is injected. While that operand is being processed, it circulates around the loop and the *Select Valid* block will continue selecting it until the computation is complete. The original input value circulates in the loop shown on the left and each time it does so it is decremented by one. When the value is decremented to 1, the *Output Valid If != 1* block will cause the operand to become invalid. When a valid result is written from *Data Port 2*, it will trigger the injection of a new valid operand from *Data Port 1*, and because the previous value is now invalid, the *Select Valid* block will select the new operand and the process begins again. One implementation is shown below in Figure 9.

Authorized licensed use limited to: MICROSOFT. Downloaded on October 8, 2009 at 12:15 from IEEE Xplore. Restrictions apply.

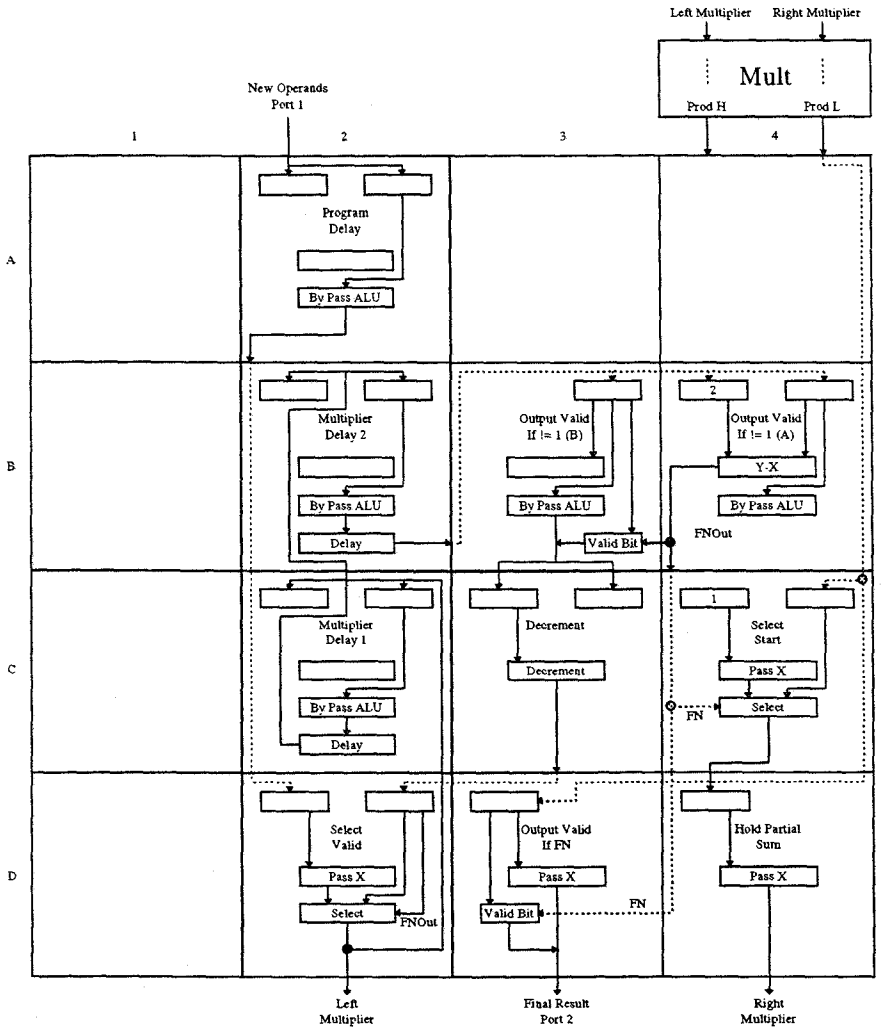

Figure 9: Colt Factorial Implementation.

The multiplicative calculations are performed by the process that executes in the loop shown on the right side of Figure 8. The *Latch Valid* register shown is used to hold the partial product computed in each iteration of the calculation, and is initialized with a valid 1 at configuration time. The *Multiplier* will not output a valid result unless both inputs to it are valid. This condition will only occur when the operand value being decremented circulates through to the output of the *Select Valid* block. At that time, the *Multiplier* will produce a valid result, which will be sent to the *Select Start* and *Output Valid If FN* blocks. The *Select Start* block will always forward the results of the multiplier unless the original operand has been decremented to 1 and the calculation is complete. If the calculation is not finished, the new partial product is forwarded to the *Hold Partial Sum* block and then to the *Multiplier* where the process will begin again when the original operand re-circulates. If the calculation has finished, then the *Select Start* block will forward a valid 1 to the *Hold Partial Sum* block so that it will be initialized for the next calculation. At the same time, the *Output Valid If FN* block will receive the fully computed value and will send it out *Data Port 2*.

7. Conclusion

The applications exemplified in this paper illustrate the diversity of algorithms that can be instantiated through the high-speed run-time reconfiguration. Characteristics of the kernels are provided in Table 1. The Colt chip is a prototype Wormhole RTR device which has been fabricated using a 3-metal 0.5 μm CMOS process through MOSIS on a 6.1 mm x 5.5 mm die.

Authorized licensed use limited to: MICROSOFT. Downloaded on October 8, 2009 at 12:15 from IEEE Xplore. Restrictions apply.

Table 1: Kernel Performance Summary

Kernel	*Die Utilization*	*Configuration Words*	*Configuration Time*	*Performance*
Dot Product	< 30%	37	740 ns	50 Million/N per Sec.
FP Mult.	100%	142	1360 ns	50 Mflops
Factorial	72%	94	1880 ns	Data Dep.

REFERENCES

[1] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider, "Teramac- Configurable Custom Computing," *Proceedings of the Third IEEE Symposium on FPGAs for Custom Computing Machines,* IEEE Computer Society Press, pp. 32-38, April, 1995.

[2] R. Hartenstein, R. Kress, and H. Reinig, "A New FPGA Architecture for Word-Oriented Datapaths," *Proceedings of the 4th International Workshop on Field-Programmable Logic and Applications,* Springer-Verlag, pp. 144-155, September 1994.

[3] I. Page, "Reconfigurable Processor Architectures," *Microprocessors and Microsystems,* vol. 20, no. 3, pp. 185-196, 1996.

[4] E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *the Forth IEEE Symposium on FPGAs for Custom Computing Machines,* IEEE Computer Society Press, pp. 78-84, April, 1995.

[5] R. Bittner, P. Athanas, "Wormhole Run-time Reconfiguration," to appear at the ACM/FPGA Conference, Monterey, California, February 1997.

[6] R. Bittner, M. Musgrove, P. Athanas, "Colt: An Experiment in Rapid Run-time Reconfiguration," in *SPIE Proceedings on High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic,* pp. 187-194, November 1996.

[7] D. Buell, J. Arnold, W. Klienfelder, *Splash 2: FPGAs in a Custom Computing Machine,* IEEE Computer Society Press, 1996.

[8] J. Hadley and B. Hutchings, "Design Methodologies for Partially Reconfigured Systems," *Proceedings of the Third IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 78-84, IEEE Computer Society Press, April, 1995.

[9] E. Lemoine and D. Merceron, "Run-Time Reconfiguration of FPGAs for Scanning Genomic Databases," *Proceedings of the Third IEEE Symposium on FPGAs for Custom Computing Machines,* IEEE Computer Society Press, pp. 90-98,April, 1995.

[10] B. Schoner, C. Jones, and J. Villasenor, "Issues in Wireless Video Coding using Run-time Reconfigurable FPGAs," *Proceedings of the Third IEEE Symposium on FPGAs for Custom Computing Machines,* IEEE Computer Society Press, pp. 85-89, April, 1995.

[11] R. Bittner, "Wormhole Run-Time Reconfiguration: Conceptualization and VLSI Design of a High Performance Computing System," Ph. D dissertation, Virginia Tech, Department of Electrical and Computer Engineering, Blacksburg, Virginia, January 1997.

[12] "Wormhole RTR" http://www.ee.vt.edu/athanas/whrtr

Authorized licensed use limited to: MICROSOFT. Downloaded on October 8, 2009 at 12:15 from IEEE Xplore. Restrictions apply.