Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems

I. Sahin[1], C. S. Gloster[2], and C. Doss[1]

[1]Department of Electrical and Computer Engineering,
North Carolina State University, Raleigh, NC 27695-7914

[2]Department of Electrical Engineering,
Howard University, Washington, DC 20059

Abstract

Reconfigurable Computing (RC) has emerged as a viable computing solution for computationally intensive applications. Applications mapped to RC systems include image processing algorithms, pattern recognition in high-energy physics, and genetic optimization algorithms. Due to the hardware complexity of the floating-point modules and limited resources available in prior RC systems, applications that required floating-point operations were either, not mapped to RC systems, or converted to fixed point before developing the RC implementation. Recent advances in Field Programmable Gate Array (FPGA) technology offer the user more hardware resources on a single FPGA device and thus the greater potential to develop complex RC systems.

In this paper, the feasibility of mapping applications containing floating-point operations to RC systems is presented. Four different types of floating-point modules: vector addition, subtraction, multiplication, and accumulation were modeled using VHDL and mapped to a XC4044XL FPGA device. These modules are highly pipelined and optimized for both speed and area. Using the modules, an example application, floating-point matrix multiplication, was also developed. Our results verify that floating-point applications are feasible and that significant speedup can be obtained when mapping these applications to RC systems.

I. Introduction

Adaptive computing, also known as reconfigurable computing (RC), is a combination of hardware/software data processing platforms that include a general-purpose processor and one or more FPGA devices. These RC systems combine the flexibility of general-purpose processors with the speed of application specific processors [1], [2]. In a typical reconfigurable computer, computationally intensive portions of algorithms are executed on FPGA devices for enhanced performance. A well-designed and utilized adaptive computer could yield 10X to 100X improvement in execution time over conventional general-purpose processor based "software only" computers.

Several applications have been mapped to reconfigurable computers to demonstrate the viability of RC systems. Applications mapped to these systems include image processing algorithms [3], [4], genetic optimization algorithms [5], and pattern recognition [6]. In most cases, the reconfigurable computing system provided the smallest published execution time for these applications.

Each FPGA device contains a finite set of hardware resources. Therefore, not all applications can be efficiently mapped to these systems. This is especially true for applications in which floating-point (FP) arithmetic operations are needed, due to the large amount of resources required by floating-point units. As a result, application developers typically, either avoided implementing these applications in RC systems, or converted the floating-point operations to fixed-point operations to reduce the amount of hardware resources required [7].

Recent advances in FPGA technology have opened new doors for developers. Both size and clock speed of FPGA devices have increased significantly. With today's technology, more than a million logic gates can be implemented on a single FPGA device, and can be clocked at speeds greater than 100 MHz. These improvements give us the opportunity to implement more complex applications, including those that require floating-point arithmetic.

In a recent study, we implemented several IEEE floating-point operations [8] in VHDL, including addition, subtraction and multiplication, and mapped them to a XC4044XL FPGA device [9] to demonstrate the feasibility of floating-point applications in RC systems. While implementing the FP operations, our goal was to maximize the speed, minimize the hardware resources required, and reduce the design time.

For each floating-point operation, we developed a standard core unit. Each core unit is highly pipelined, has the same inputs and outputs, and has the same latency. To facilitate core reuse, we also developed different types of standard module structures, each of which has a standard data path and a standard controller. By instantiating each unique core unit into a standard module structure, we created a new module for each operation.

We used a commercially available [10] RC system to test the modules. Using the modules, an example application, matrix multiplication, was also implemented. Results demonstrated that floating-point applications can be implemented on RC systems, with significant speedup over a general-purpose processor implementation. These systems are also easier to debug since conversions between floating-point and integer formats is not required.

This research was supported in part by NASA grant #NAG5-7942

In this paper, we discuss the implementation details and present FPGA utilization and performance data for using floating-point operations on RC systems. The following section presents detailed information about the core units and the modules. Experimental results and module statistics are presented in Section III. Implementation of a matrix multiplier using our modules is presented in Section IV. The paper concludes with suggestions for future research.

II. FLOATING-POINT MODULES

In this study, we developed several standard components to create different types of modules that are useful for various applications. These component types are floating-point core units, module controllers and module datapaths. These components are standardized in terms of the number of inputs, the number of outputs and module latency, in order to facilitate module interconnection for complex operations. By combining unique core units with a few controllers and datapaths, several different types of modules have been created. Using this approach, the time required to design a new module is reduced significantly. When a new core unit is designed, one simply combines the new core with an off-the-shelf controller and datapath.

Standard Controller
Controller commands
Feedback signals
Standard Datapath
Floating-point Core
Addr. Bus
Data I/O
Memory

Figure 1: Block diagram of the standard module structure.

Figure 1 shows the block diagram of the standard floating-point module. Each floating-point module consists of a standard controller and a standard datapath that interfaces with an external memory. New cores are simply instantiated in a standard datapath resulting in new modules.

This paper presents experimental results using three different core units, three different controllers and three different datapaths to produce a total of 7 floating-point modules (vector addition, subtraction, multiplication, and accumulation). Two of these modules were used to implement matrix multiplication on 4 FPGA devices.

A. *Instruction Formats and Module Execution*

All modules are designed to operate on one or two equal length vectors. Each vector instruction corresponds to a single floating-point vector operation. A standard instruction includes three or four operands depending on the type of module used. For each one input vector module, the first operand of an instruction is the starting address of the input vector, the second operand is the starting address of the output vector. The third operand is the size of the input vectors.

For each two input vector module instruction, the first two operands are the starting addresses of the input vectors, the third operand is the starting address of the output vector and the last operand is the size of the vectors. Figure 2 shows the instruction format for each module. The floating-point accumulation module uses the instruction format of Figure 2a. However, this module produces an output vector with length 1.

The modules were designed using a commercially available FPGA board. This board includes five FPGA devices or Processing Elements (PEs). Each PE has its own dual ported local memory (1M Byte). The host computer and the PE have read and write access to the local memory. The memory space of each module is partitioned into two sections, instruction and data. The instruction memory always starts at memory address $00000 and ends with the HALT instruction ($FFFFFFFF). The remaining memory that is not used for instructions is used for data.

```
FPVECADD  100 200 50
```
Size of the vectors
Start of the output vector
Start of the input vector

(a)

```
FPVECADDS 100 150 200 50
```
Size of the vectors
Start of the result vector
Start of the 2nd vector
Start of the 1st vector

(b)

Figure 2: Modules instruction formats. (a) Module instruction for a single input vector module. (b) Module instruction format for a two input vector modules.

Once a module configuration has been loaded into a PE and the local memory has been initialized by the host computer, the module waits for the reset signal to be asserted. When this occurs, the module reads the first instruction from the memory location $00000. It then begins executing the instruction. When the current instruction has completed, the module reads the next instruction from the instruction memory. This process continues until the module reads a HALT instruction ($FFFFFFFF) from the instruction memory. When this value is read, the module stops and sends an interrupt signal to the host computer.

The modules' execution times can be evaluated given the number of cycles required to process one set of vectors. The memory unit we used has a two clock cycle latency for read operations and a one clock cycles latency for write operations. The vector addition, subtraction and multiplication modules

write results back to the memory between successive read operations. Hence, the optimal memory access schedule for these modules is two read cycles followed by one write cycle producing a result every 3 cycles. We achieved near-optimal performance with our modules since we inserted only one idle state. In this approach, an output is produced every 4 cycles.

Figure 3 shows the memory access schedule for these modules. Equation (1) is used to approximate the total execution time of the modules where T_E is the total execution time, N_F is the number of cycles required to fetch an instruction, N_P is the number of cycles required to process the given vectors, N_E is the number of cycles required to empty the pipelined core, F_M is the module clock rate, and C_{API} is the Application Programming Interface (API) overhead.

$$T_E = \frac{N_F + N_P + N_E}{F_M} + C_{API} \quad (1)$$

$$T_E = \frac{4N}{F_M} + C_{API} \quad (2)$$

For one and two input vector addition, subtraction, and multiplication, the instruction fetch takes 9 and 10 cycles respectively and pipeline emptying takes 8 cycles for both types. Processing takes 4 cycles per pair of vector elements. The constant API overhead depends on the host computer's speed. For large vectors, instruction fetch and pipeline emptying times for addition subtraction, and multiplication are negligible and equation (1) could be rewritten, as equation (2) where N is the length of the vectors.

Since the accumulator unit does not write back to the memory until the end of the module instruction, it is able to read an element of the input vector from the memory every clock cycle. As a result, the core in the accumulator is utilized 100 % and runs almost fours times as fast as the other modules. Figure 4 shows the memory access schedule for the accumulator module. Equation (1) also applies to this module. Equation (3) shows the execution time when instruction fetch and emptying are negligible.

$$T_E = \frac{N}{F_M} + C_{API} \quad (3)$$

B. Core Units

The most important component of a module is the floating-point arithmetic core. Figure 5 shows the block diagram of the standardized core unit. Each core has two 32-bit inputs and one, 32-bit output to accommodate single precision FP numbers. For addition, subtraction and multiplication, different floating-point core units were developed. There is a standard interface definition for the core units to reduce design time. Once a new core unit is designed, it is easy to create a new module by just instantiating the new core unit into the standard module structure.

	$R/\overline{W}$	1	x	1	x	1	x	1	x	1	x	1	x	1	0	1	x	1	0	1	x	x	0	x	x	x	0	x	x	x	0
Mem. Stages	MAB	A_1		A_2		B_1		B_2		C_1		C_2		D_1	A	D_2		E_1	B	E_2			C				D				E
	MDB			A_1		A_2		B_1		B_2		C_1		C_2	A	D_1		D_2	B	E_1		E_2	C				D				E
FP Core Unit Stages	S1						A				B				C				D				E								
	S2							A				B				C				D				E							
	S3								A				B				C				D				E						
	S4									A				B				C				D				E					
	S5										A				B				C				D				E				
	S6											A				B				C				D				E			
	S7												A				B				C				D				E		
	S8													A				B				C				D				E	

Figure 3: Memory access schedule for one and two input vector addition, subtraction, and multiplication modules.

	$R/\overline{W}$	1	1	1	1	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	•	•	•	•	•	•	0
Mem. Stages	MAB	A	B	C	D	E	F	•	•	•	•	X	Y											•	•	•	•	•	•	Re
	MDB			A	B	C	D	E	F	•	•	•	•	X	Y									•	•	•	•	•	•	Re
FP Core Unit Stages	S1				A	B	C	D	E	F	•	•	•	•	X	Y														
	S2					A	B	C	D	E	F					X	Y													
	S3						A	B	C	D	E	F					X	Y												
	S4							A	B	C	D	E	F					X	Y					•	•	•	•	•	•	
	S5								A	B	C	D	E	F					X	Y										
	S6									A	B	C	D	E	F					X	Y									
	S7										A	B	C	D	E	F					X	Y								
	S8											A	B	C	D	E	F	•	•	•	•	X	Y							

N Numbers to accumulate — Emptying the accumulator — Writing result

Figure 4: Memory access schedule for the accumulation module.

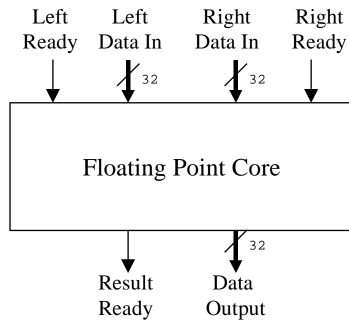

Figure 5: Block diagram of the standard core units.

To improve the maximum clock speed that can be applied to the units, all core units are divided into a standard number of pipeline stages (8). We used a standard number of pipeline stages to alleviate the need to develop a unique controller within each core. However, the main controller can handle cores with arbitrary latencies. While, using pipeline units requires additional registers resulting in an increase in FPGA CLB resources, it provides significant benefit in terms of increased clock speed.

To reduce the hardware requirements and to make the module controller simpler, core units are designed as self-controlled units. Once data is available at both inputs, the core unit starts processing. Results are available at the output of the unit 8 clock cycles later.

This is accomplished with a standard floating–point core I/O interface. Each core has two input signals and one output signal for control and core interconnection. Each time that the module controller reads a floating-point number from the memory, it asserts either the LEFT_READY or RIGHT_READY signal corresponding to the core input that has valid data. When both inputs to the core have valid data and both ready signals are asserted, the core begins the floating-point operation. When the core finishes processing the data, it asserts the RESULT_READY signal. The main controller then stores the result in memory.

Use of the standard interface control signals serves two purposes. The main purpose is to reduce controller complexity and to increase controller flexibility. Hence, a single controller can handle future cores with arbitrary latencies. The controller does not send command signals to each stage of the core. Instead, it uses the interface signals to signal the core that the input data is ready. It also uses the RESULT_READY signal produced by the core to determine when the result is ready. This simplification in the controller saves control states, logic gates, and future application development time. The other purpose is to facilitate the incorporation of complex cores into the system. The use of the standard interface control signals makes it is easy to form larger cores by simply linking existing cores together.

C. Module Datapath

Three unique datapaths have been developed for this paper. Figure 6 shows the block diagram of the datapath for two input vector addition, subtraction, and multiplication. Although the core unit is self-controlled, there are still many parts to control in the datapath. For that reason, as shown in Figure 6, the datapath was partitioned into two sections: the data processor and the fetch/decode unit. The controller generates different micro instructions for each section.

The data processor section of the accumulator datapath consists of the core unit, two 32-bit data registers, and two multiplexors. The registers are used for two purposes. First, they are used for temporary storage. Since we are only able to read one 32-bit value at a time from the memory, the data read from memory is stored in one of these registers. Secondly, since the floating-point core inputs are not registered, we must include registers in the datapath. For the accumulator module, it is necessary to connect the output of the core back to the input of the core. This is accomplished with M0 and M1 multiplexors as shown in Figure 7.

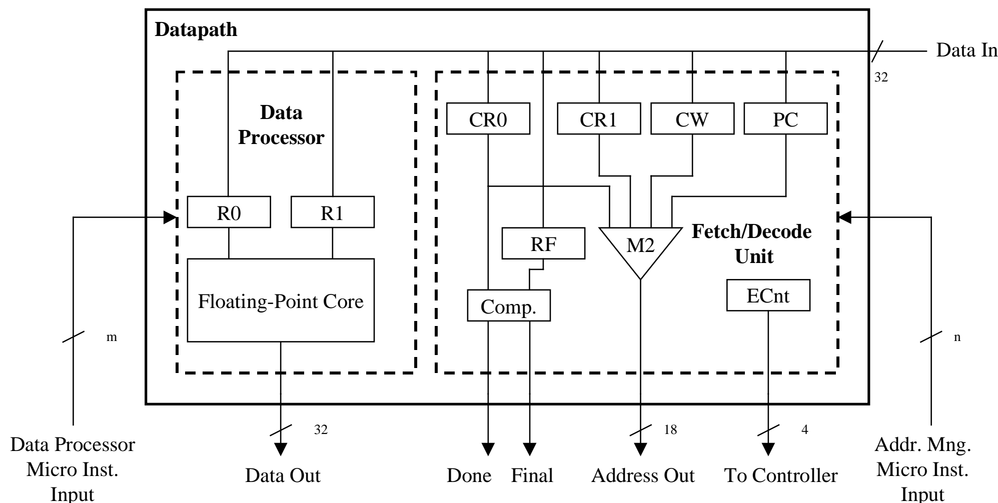

Figure 6: Block diagram of the standard datapath for two input vector modules.

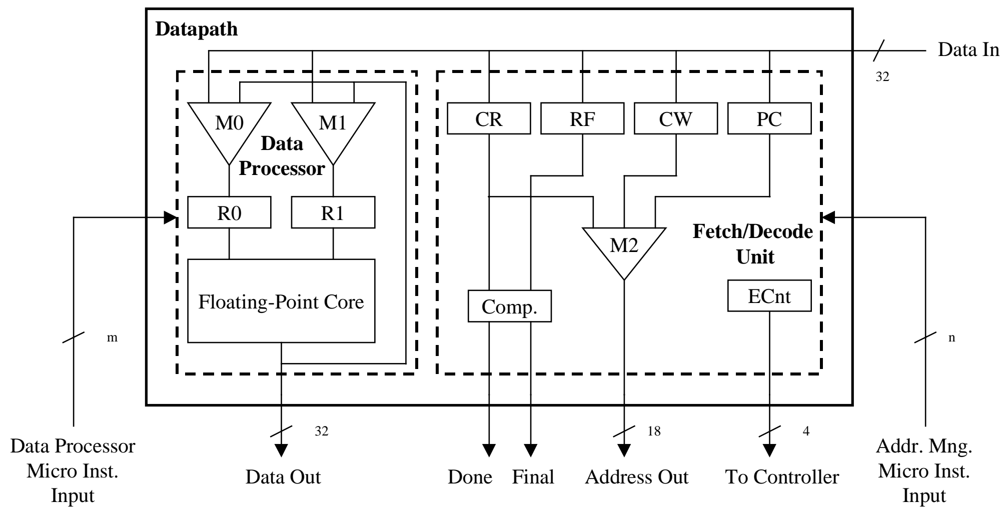

Figure 7: Block diagram of the standard datapath for accumulator.

The fetch and decode unit includes: four counters, one register, one specialized comparator, and a multiplexor. The CR and CW counters are loadable counters and are used for addressing input and output vectors.

PC is used as a program counter to keep track of the module instructions. The E Counter is used for emptying the pipelined core units. After the last set of input data is loaded from the memory, the controller sets this counter equal to the number of cycles required to empty the pipeline. The controller waits until all the remaining data in the core is processed and the results are written back to the memory. The E counter is especially useful while emptying the accumulator core. The RF register is used to store the size of the input vectors.

The specialized comparator produces two signals. The DONE signal is asserted when the module reaches the end of a given set of vectors. The FINAL signal is asserted when all instructions have been processed.

D. *The Module Controller*

In this paper, three unique module controllers are presented. The first controller assumes that elements of the input vector pair are interleaved or stored in consecutive memory locations as follows; A0, B0, A1, B1, A2, B2, ... AN, BN. It is used for one input vector modules. The second controller assumes that the input vectors are separate. The first and the second type of controllers were used to construct vector addition, subtraction, and multiplication modules. The third type of controller has been developed for the accumulator module. This controller is much more complicated than the previous two controllers.

When a pipelined adder is used for vector accumulation the process can be performed in three steps. Step 1: Forward the numbers through the pipeline until the first number appears at the output of the pipeline. Step 2: Accumulate the numbers until the last number is read from the memory. Step 3: Empty the pipeline. The first and the second steps are similar to the addition and multiplication process. The last step requires special handling; therefore, a special module controller has been developed for the accumulator module.

III. Experimental Results

A. *Module Statistics*

Table 1 shows the resulting device utilization and maximum clock speed for each module. These values were collected after module placement and routing was completed for a XC4044XL FPGA device.

Table 1: Device utilization and maximum clock speeds.

Module Name	CLB Util.	% Util.	Clock Speed (MHz)
Adder (One Input Vector)	**463**	**28**	**29.53**
Adder (Two Input Vectors)	**473**	**29**	**30.44**
Adder Core Only	**316**	**20**	**42.90**
Subtractor (One Input Vector)	**464**	**29**	**30.08**
Subtractor (Two Input Vectors)	**476**	**29**	**30.64**
Subtractor Core Only	**316**	**20**	**42.29**
Multiplier (One Input Vector)	**953**	**59**	**28.47**
Multiplier (Two Input Vectors)	**984**	**61**	**27.23**
Multiplier Core Only	**834**	**53**	**34.50**
Accumulator	**432**	**27**	**31.43**

The adder and the subtractor modules use only 28% and 29% of an FPGA device, respectively. This means that three adder or subtractor modules can fit into one FPGA device. On the other hand, since the adder and subtractor cores require only 20% of the device, five cores can fit into one FPGA device. Since the board that we are using has 5 FPGA devices on it, a total of 25 adder or subtractor cores can be utilized on the board. The complete multiplier module requires around 60% of an FPGA device, with the core requiring 53% of an FPGA device. Only one multiplier module can fit into one FPGA. Therefore, a total of five multipliers can be utilized on the board simultaneously.

B. *GPP Versus RC Modules*

The clock frequencies shown in Table 1 are the values indicated by the design tools. Modules were tested at these speeds and they behaved as expected. However, we over-clocked the modules to 50 MHz, the maximum clock speed supported by the FPGA board. Surprisingly, all the modules worked properly at 50 MHz.

Table 2 shows the execution times of the modules, along with the regular C++ implementations running on a Pentium II 300 MHz processor based PC. In these experiments, the modules were clocked at 50 MHz. The length of each input vector was 131,000 requiring 232,000 words of memory for storage. Hence, a total of 131,000 floating-point operations were performed. Since all the modules have exactly the same latency, the execution time is identical for all modules. When only one addition, subtraction or multiplication module utilized, the module runs faster than the regular software implementation but slower than the optimized software implementation. As the number of modules used increases, the execution time of the modules decreases. When five modules are utilized, the modules perform the same number of floating-point operations more than four times faster than a Pentium II 300 MHz processor. When five accumulator modules are utilized, the accumulators run 9 and 14 times faster than regular and optimized software implementations, respectively. The accumulator module gains more speedup than the other modules because of 100 % core utilization.

IV. EXAMPLE APPLICATION: MATRIX MULTIPLICATION USING MODULES

In this study, we also wanted to implement an example application, matrix multiplication, to demonstrate how to use modules to solve larger problems. Matrix multiplication was selected because of its scalability and highly parallel nature [11]. For simplicity we used square matrices. Two input matrices were divided into two equal parts. The first matrix was divided horizontally and the other was divided vertically. Combinations of the halves were assigned to four PEs as shown in figure 7. Each PE was responsible for calculating one quarter of the resulting matrix. The PEs' memory was divided into three sections, instruction, input data and the result data memories. Since each PE has 1 MB of memory the largest square matrices that we can multiply is 98 x 98.

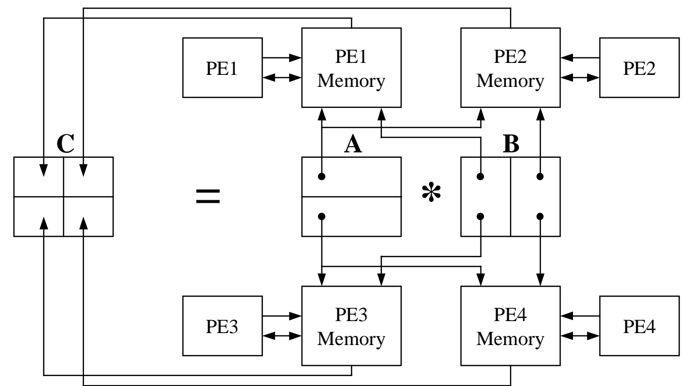

Figure 7: Matrix multiplication using modules.

The matrix multiplication process was implemented in two sessions. In the first session, the input matrix data and the module instructions were stored in the memory units. After that, the PEs were configured with multiplication modules and were started by releasing the reset signal. The host computer waited for the interrupt signals from PEs. When the host computer received interrupt signals from all four PEs, the first session was completed and all inner products of the matrices were stored in each PEs' memory. In the second session, the instruction memories were updated and accumulator instructions were loaded in all four PEs' instruction memories. Subsequently, without loading any input data, PEs were configured with the accumulator module. The result memory section of the first session was used as the input memory for the second session and the accumulator module instructions were generated so that the accumulator module could use the result of the multiplier module. The accumulator modules run and the final results of the matrix multiplication was stored in the input data memory of the first session. After the second session was completed, the host computer read the results from the PEs' memories and printed it. Since many addresses involved in this two-session matrix multiplication, it is almost impossible to manually generate all module instructions correctly.

Table 2: Comparison of module execution time with software implementations.

Operation Type	Implementation Type				
	Software C++	Software Optimized C++	Hardware (1 Module)	Hardware (2 Modules)	Hardware (5 Modules)
One Input Vector Addition	**14.48**	**8.54**	**10.80**	**5.40**	**2.16**
One Input Vector Subtraction	**14.29**	**7.95**	**10.80**	**5.40**	**2.16**
One Input Vector Multiplication	**14.28**	**7.97**	**10.80**	**5.40**	**2.16**
Two Input Vector Addition	**12.67**	**9.79**	**10.80**	**5.40**	**2.16**
Two Input Vector Subtraction	**12.24**	**9.64**	**10.80**	**5.40**	**2.16**
Two Input Vector Multiplication	**12.33**	**9.52**	**10.80**	**5.40**	**2.16**
Accumulation	**7.54**	**4.89**	**2.704**	**1.36**	**0.54**

For that reason, we developed a small tool to generate instructions for the multiplication and the accumulatoin modules. We also implemented a host program to manage all data and instruction transfer between the host computer and the PEs and to manipulate the modules.

A. Results of Matrix Multiplication.

To test the matrix multiplication, using the tool, we generated module instructions for 40 x 40 and 96 x 96 matrices. With the help of the host program, matrix multiplications were performed on a RC system available in our laboratory [10]. Table 3 shows the software and module execution times in millisecond.

In these experiments, the software version was running on a 300 MHz Pentium II based PC and the modules were clocked at 50 MHz. The results showed that, excluding the configuration time, modules performed the matrix multiplication 2 to 3 times faster than the regular software implementation and 0.64 to 1.98 times faster than the optimized software version. The optimized software version runs extremely fast when the matrix size is small because the host computer takes advantage of its cache memory. For small matrices, it is able to hold the entire input and output matrices in the cache memory. From the table one can conclude that as the size of the matrices increase, it is possible to obtain significant speedup using the modules. This result implies that the larger the matrix sizes, the better the modules will run. One disadvantage of this modular matrix multiplier is that the configuration time, which is approximately 130 milliseconds, should be alleviated. To remove the configuration time overhead, a future version of matrix multiplication can be completed in a single session using a multiply-accumulate module (MAM).

Table 3: Comparison of software and module matrix multiplier execution times.

Matrix Size	**Implementation Type**			**Speed-up**
	Software (C++)	**Software** (Optimized C++)	**Hardware** (Ses1 + Ses2 = Total)	
40 x 40	4.40	1.45	1.50+0.74 = 2.24	0.64
96 x 96	67.81	48.35	18.63+5.71 = 24.34	1.98

V. Conclusions

In this study, we investigated the feasibility of using floating-point arithmetic in RC systems and presented the results comparing the system to a general-purpose processor. CLB utilization of the modules demonstrated that applications including floating-point operations could be mapped to current RC systems. Future RC systems will only have increased FPGA resources and hence can accommodate many more floating-point resources. The results indicate that floating-point modules can achieve speedups of a factor of 5 to 14 over a typical desktop computer when the modules are utilized in parallel. As an example, matrix multiplication was implemented using the modules. The matrix multiplication results showed that the modules can achieve greater than 2x speedup over a typical desktop computer. Results of this study will be used in the development of future design automation tools with the goal of facilitating RC system development while maintaining enhanced performance.

Acknowledgments

I would like to thank Dr. Alexander Winser of NCSU, Department of Electrical and Computer Engineering for providing the Annapolis Micro Systems Wildforce RC Board for this study.

References

[1] D. Bhatia, "Reconfigurable computing," *Tenth International Conference on VLSI Design*, pp. 356-359, Jan. 1997.

[2] F. Rincon and L. Teres, "Reconfigurable hardware systems," *1998 International Semiconductor Conference*, Vol.1, pp. 45-54, Oct. 1998.

[3] E. Cerro-Prada, S.M. Charlwood, P.B. and James-Roxby, "Image processing and its applications," *Seventh international conference on image processing and its applications*, Vol.1, pp. 450-454, Jul. 1999.

[4] R.C.D.M. Tavares, C.J.N. Jr. Coelho, A.D.A. Araujo and A.O. Fernandes, "Implementation of an edge detection algorithm in a reconfigurable computing system," *Proceedings of the Eleventh XI Brazilian Symposium on Integrated Circuit Design*, pp. 38-41, Sep. 1998.

[5] P. Graham and B. Nelson, "Genetic Algorithms in Software and in Hardware," *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1996.

[6] H. Hogl, A. Kugel, J. Ludvig, R. Manner, K.H. Noffz, and R. Zoz, "Enable++: A Second Generation FPGA Processor," *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[7] W.B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, K.D. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. Apr. 1998.

[8] IEEE Standard Board, "IEEE Standard for Binary Floating-Point Arithmetic", *ANSI/IEEE Std 754-1985,* Aug. 1985.

[9] Xilinx Data Book 2000, V. 1.7, pp. 6-73, Oct. 1999.

[10] Annapolis Micro Systems Inc., "Wildforce Reference Manual," Revision 3.4, 1997.

[11] K. Li, Y. Pan, and S.Q. Zheng, "Fast and processor efficient parallel matrix multiplication algorithms on a linear array with a reconfigurable pipelined bus system," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 8, pp. 705 – 720, Aug. 1998.