

# Floating-Point Modules Targeted for Use with RC Compilation Tools

Clay S. Gloster, Jr., Ph.D., P.E.  
Department of Electrical Engineering  
Howard University  
Washington, DC 20059  
Phone: (202) 806 6628  
cgloster@ howard.edu

Ibrahim Sahin  
Department of Electrical and Computer Engineering  
North Carolina State University  
Raleigh, NC 27695-7914  
Phone: (919) 513 2014  
isahin@eos.ncsu.edu

## ABSTRACT

Reconfigurable Computing (RC) has emerged as a viable computing solution for computationally intensive applications. Several applications have been mapped to RC systems and in most cases, they provided the smallest published execution time. Although RC systems offer significant performance advantages over general-purpose processors, they require more application development time than general-purpose processors. This increased development time of RC systems provides the motivation to develop an optimized module library with an assembly language instruction format interface for use by future RC system compilers. Hence, RC system compilation tools for C++/Java language programs can utilize these modules providing the required infrastructure for an automated RC development system that will reduce development time significantly. In this paper, we present area/performance metrics for 9 different types of floating point (FP) modules that can be utilized to develop complex FP applications. These modules are highly pipelined and optimized for both speed and area. Using these modules, an example application, FP matrix multiplication, is also presented. Our results and experiences show that with these modules, 8-10X speedup over general-purpose processors can be achieved.

**Keywords:** Adaptive Computing, Reconfigurable Computing, Reconfigurable Systems.

## 1. 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.

Although RC systems offer significant performance advantages over general-purpose processors, they have a few disadvantages. RC systems require more application development time than general purpose processors, but significantly less than developing an application specific integrated circuit. Also, RC system designers need to be knowledgeable in the areas of hardware and software system design. Additionally, due to the limited resources available in previous RC systems, applications that required floating-point (FP) operations were either, not mapped to RC systems, or converted to fixed point before developing the RC implementation [7].

In a recent study, we implemented several FP modules in VHDL to perform IEEE floating-point operations [8] including addition, subtraction and multiplication, and mapped them to a XC4044XL FPGA device [9] to create a FP module library. While implementing the FP modules, our goal was to maximize the speed, minimize the hardware resources required, and reduce both the module and the application design and implementation time.

Each module is designed to execute a specific machine language instruction to process a set of FP vectors. Using the modules, RC compilation tools can be developed to automate the RC system design process. These compilation tools can potentially compile applications implemented in C++/Java and produce assembly/machine language instructions that correspond to each module.

In this paper, an example application, FP matrix multiplication, is presented. This application utilizes several modules to demonstrate that larger, more complex, applications can be developed with these modules. Our results and experiences demonstrated that with these modules, application development time is reduced significantly and 8-10X speedup over general-purpose processors can be achieved.

This paper presents implementation details and area/performance metrics for 9 different types of FP modules that can be utilized to develop complex FP applications. These modules are highly pipelined and optimized for both speed and area. 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.

## 2. 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.

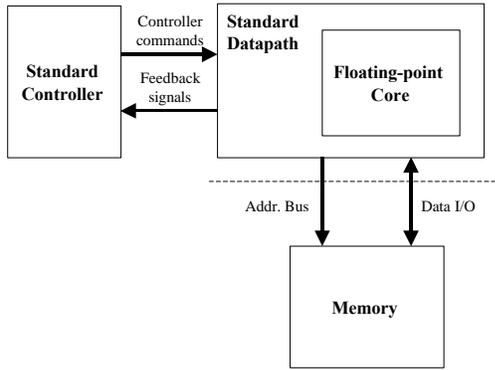


Figure 1: Block diagram of the standard module structure.

Figure 1 shows the block diagram of the standard floating-point module. Each 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, four different controllers and three different datapaths to produce a total of 9 floating-point modules. **(IBRAHIM - LIST THE CORES HERE)** The multiply-accumulate module was used to implement matrix multiplication on 4 FPGA devices.

### 2.1 Module Machine Language and Execution

All modules are designed to execute a specific machine language instructions. Each module instruction corresponds to a single floating-point vector operation. A standard instruction includes three or four operands depending on the type of module used. Figure 2 shows the instruction format for each module. For each 3 operand module instruction, the first operand is the starting address of the input vector, the second is the starting address of the output vector, and the third operand is the size of the input vectors.

For each four operand module instruction, the first two operands are the starting addresses of the two input vectors, the third operand is the starting address of the output vector and the last operand is the size of the vectors. The floating-point accumulator and product modules use the instruction format of Figure 2a. However, these modules produce an output vector with length 1.

All modules were designed for a commercial FPGA board [10] that is readily available in our laboratory. 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 (\$FFFFFFF). The remaining memory that is not used for instructions is used for data.

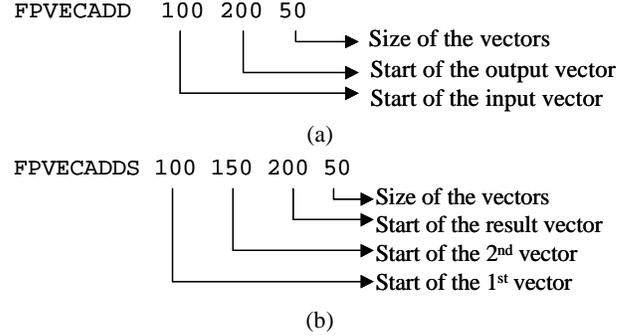


Figure 2: Modules instruction formats. (a) Module instruction for a single input vector module. (b) Module instruction format for a two input vector module and the multiply-accumulate module.

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 (\$FFFFFFF) 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. Using this approach, an output is produced every 4 cycles.

We developed Equation (1) to approximate the total execution time of the modules ( $T_E$ ). In this equation,  $N_F$  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 three and four operand modules developed for vector addition, subtraction, and multiplication, the instruction fetch takes 9 and 10 cycles respectively and pipeline emptying takes 8 cycles. 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 and product modules do not write back to the memory until the end of the module instruction, both are able to read an element of the input vector from the memory every clock cycle. As a result, cores in the accumulator and the product modules are utilized 100% and run almost four times as fast as the other modules. Equation (1) also applies to these modules. Equation (3) shows the execution time when instruction fetch and emptying are negligible.

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

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

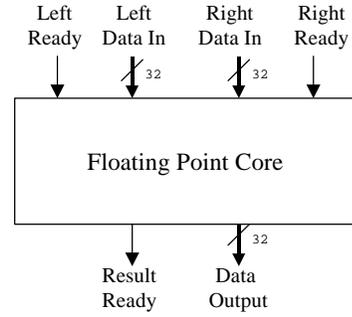
The multiply and accumulate module is able to read two FP vector elements every two (**IBRAHIM – IS THIS RIGHT?**) cycles. The core units are 50% utilized. Equation (4) could be used to estimate execution time for this module.

## 2.2 Core Units

The most important component of a module is the floating-point arithmetic core. 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. By instantiating each unique core unit into a standard module structure, we created a new module for each operation.

Figure 3 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.

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.



**Figure 3 Block diagram of the standard core units.**

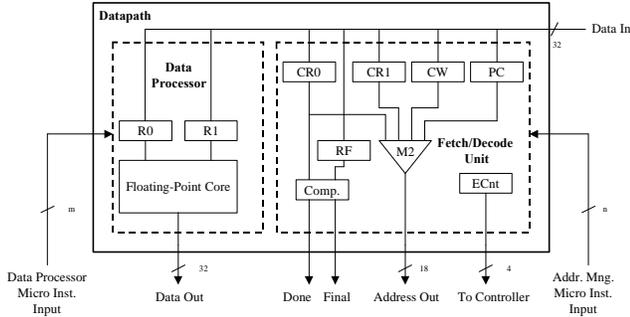
To reduce the hardware requirements and to make the module controller simpler, core units are designed as self-controlled units. Once data is available at both inputs, the core unit starts processing. Results are available at the output of the unit 8 clock cycles later.

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 addition of complex cores into the library. The use of the standard interface control signals makes it is easy to form larger cores by simply linking existing cores together.

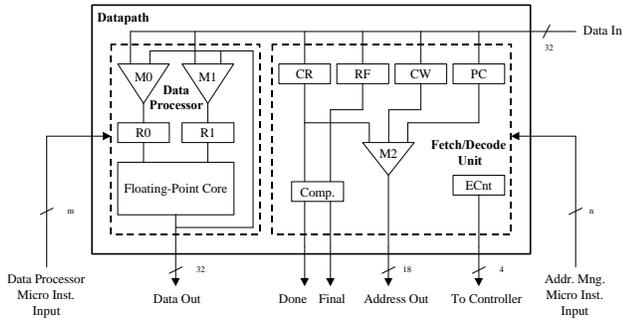
## 2.3 Module Datapath

Three unique datapaths have been developed for this paper. Figure 4 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.



**Figure 4: Block diagram of the standard datapath for two input vector modules.**

The data processor section of the accumulator datapath consists of the core unit, two 32-bit data registers, and two multiplexers. 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 and the product modules, it is necessary to connect the output of the core back to the input of the core. This connection is accomplished with M0 and M1 multiplexers as shown in Figure 5.



**Figure 5: Block diagram of the standard datapath for the accumulator and the product modules.**

The fetch and decode unit includes: four counters, one register, one specialized comparator, and a multiplexer. 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.

## 2.4 The Module Controller

In this paper, four 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 and the product modules and fourth type has been developed for multiply and accumulate module. The controllers for the accumulation, the product and the multiply and accumulate modules are much more complicated than the previous two due the pipeline emptying process of these modules.

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.

## 3. EXPERIMENTAL RESULTS

### 3.1 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.

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, and the multiply and accumulate module requires 79% of an FPGA device. Only one multiplier module or multiply and accumulate module can fit into one FPGA. Therefore, a total of five multipliers or multiply and accumulate modules can be utilized on the board simultaneously.

**Table 1: Device utilization and maximum clock speeds.**

Module Name	CLB Util.	% Util.	Clk. Speed (MHz)
Adder (One Input Vector)	463	28	29.53
Adder (Two Input Vectors)	473	29	30.44
Subtractor (One Input Vector)	464	29	30.08
Subtractor (Two Input Vectors)	476	29	30.64
Multiplier (One Input Vector)	953	59	28.47
Multiplier (Two Input Vectors)	984	61	27.23
Accumulator	432	27	31.43
Product Module	944	59	26.44
Multiply and Accumulate Module	1265	79	25.35

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

Operation Type	Implementation Type					Speed-up (5 modules vs optimized software)
	Software C++	Software Optimized C++	Hardware (1 Module)	Hardware (2 Modules)	Hardware (5 Modules)	
<b>One Input Vector Addition</b>	14.48	8.54	10.80	5.40	2.16	3.95
<b>One Input Vector Subtraction</b>	14.29	7.95	10.80	5.40	2.16	3.68
<b>One Input Vector Multiplication</b>	14.28	7.97	10.80	5.40	2.16	3.69
<b>Two Input Vector Addition</b>	12.67	9.79	10.80	5.40	2.16	4.53
<b>Two Input Vector Subtraction</b>	12.24	9.64	10.80	5.40	2.16	4.46
<b>Two Input Vector Multiplication</b>	12.33	9.52	10.80	5.40	2.16	4.41
<b>Accumulation</b>	7.54	4.89	2.704	1.36	0.54	9.05
<b>Product</b>	7.71	6.21	2.704	1.36	0.54	11.05
<b>Multiply and Accumulate</b>	11.24	8.20	5.432	2.71	1.08	7.59

### 3.2 GPP Versus RC Modules

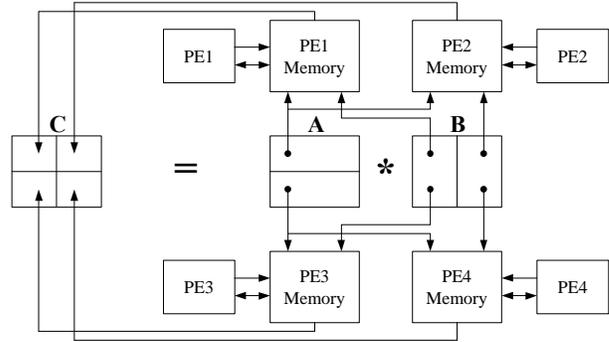
The clock frequencies shown in Table 1 are the values indicated by the design tools. Modules were tested at these speeds and they behaved as expected. However, we over-clocked the modules to 50 MHz, the maximum clock speed supported by the FPGA board. Surprisingly, all the modules worked properly at 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 261,999 floating-point operations were performed by the multiply and accumulate module and 131,000 floating point operations were performed by the other modules. Since all the modules have exactly the same latency, the execution time is identical for similar types of 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 around four times faster than a Pentium II 300 MHz processor. When five accumulator or product modules are utilized, they run 9 and 11 times faster than the optimized software implementations. The accumulator module gains more speedup than the other modules because of 100 % core utilization.

### 4. 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 6. 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 340 x 340.



**Figure 6: Matrix multiplication using modules.**

The matrix multiplication process was implemented in one session. During the 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 session was completed and all elements of the resulting matrix were calculated and stored in each PEs' memory. After the session was completed, the host computer read the results from the PEs' memories and printed it. Since many addresses involved in this matrix multiplication, it is almost impossible to manually generate all module instructions correctly. For that reason, we developed a small tool to generate instructions for the modules. We also implemented a host program to manage all data and instruction transfers between the host computer and the PEs and to manipulate the modules.

#### 4.1 Results of Matrix Multiplication.

Due to the memory limitations of the RC system we use, the maximum matrix size that we can multiply using the modules is 340 x 340. To test the matrix multiplication, using the tool, we generated module instructions for 200 x 200 and 340 x 340 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.

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

Matrix Size	Implementation Type				Speed-up comparing to Regular Software	Speed-up comparing to Optimized Software <sup>2</sup>
	Regular Software (C++)	Optimized Software <sup>1</sup> (C++)	Optimized Software <sup>2</sup> (C++)	Hardware (Modules)		
200 x 200	1046.58	814.27	590.32	71.44	14.65	8.26
340 x 340	9076.06	6585.69	4188.10	411.04	22.08	10.19

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).

## 5. CONCLUSIONS

In this study, we implemented several FP modules in VHDL to perform IEEE floating-point operations, and mapped them to a XC4044XL FPGA device to create a FP module library. The modules are designed to be utilized by RC compilation tools to automate the design process of RC applications and to reduce the design and implementation time, while maintaining enhanced performance. 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. Using these modules, an example application, FP matrix multiplication, is also presented. Our results and experiences demonstrated that with these modules, application development time is reduced significantly and 8-10X speedup over general-purpose processors can be achieved. 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.

## 6. 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.

## 7. 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.