Reconfigurable Custom Floating-Point Instructions

Zhanpeng Jin, Richard Neil Pittman, Alessandro Forin
Microsoft Research

August 2009

Technical Report
MSR-TR-2009-157

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Reconfigurable Custom Floating-Point Instructions

Zhanpeng Jin, Richard Neil Pittman, Alessandro Forin
Microsoft Research

ABSTRACT
Multimedia and communication algorithms from the embedded system domain often make extensive use of floating-point arithmetic. Due to the complexity and expense of the floating-point hardware, these algorithms are usually converted to fixed point operations, or implemented using floating-point emulation in software. This study presents the design and implementation of custom floating-point units, leveraging the partial reconfiguration feature of state-of-the-art FPGAs. The custom floating-point units can be dynamically configured, loaded, and executed when needed by software applications. The system is binary compliant with the conventional MIPS architecture and the IEEE-754 standard, and supports most of the floating-point operations and relevant functionalities. Furthermore, we present various customization strategies and construct a set of optimized functional modules to meet different application demands or requirements. Using LINPACK as a floating-point intensive example, we replace a sequence of 25 instructions with a custom instruction, and demonstrate an overall 80x application speedup.

1. INTRODUCTION
Floating-point (F-P) arithmetic, although extremely common in general purpose computing, was rarely used in embedded systems world until recently. While a number of communication and multimedia algorithms are designed and simulated using floating-point arithmetic, the implementation platforms for such algorithms often leave out any hardware floating-point unit in favor of software emulation or float to fixed point conversion due to the complexity and expense of the floating-point hardware (1). Recently, a variety of research efforts considered field-programmable gate arrays (FPGAs) as a means to accelerate floating-point computations using their well-proven flexibility and reconfigurability (2) (3) (4). However, the hardware implementation of an efficient and correct FPU is an extremely difficult, involved and time consuming task. In addition, noticeable difficulties occur when mapping floating-point units to FPGAs due to the inherent complexity of floating-point arithmetic. The increasing demand for application-specific floating-point arithmetic data paths presents the challenge of accommodating several floating-point functional modules in the limited resources available (5) (6). This makes considerations for cost-effectiveness a priority. Many recent studies have explored opportunities to improve the floating-point performance on FPGAs by optimizing the device architecture (7) (8) (5) (9). Despite great efforts on efficient and effective floating-point implementation on FPGAs, none of them implements a complete reconfigurable floating-point unit and actually deploy it to work with a real computer architecture platform.

Different from prior studies, this study not only focuses on the efficient implementation of floating-point units on FPGAs, but also proposes a new dynamically configurable floating-point extension platform, named Custom Floating-Point Unit (CFPU), which is fully compliant with IEEE-754 standard and significantly improves the floating-point intensive computation performance. Independent from the main execution pipeline, the proposed CFPUs can be configured as Extensions to the base processor making up the reconfigurable region of the FPGA. They can be dynamically configured, loaded, and executed when needed by software applications. That is, such CFPUs can be triggered in software by new instructions particularly developed to meet the changing computing needs for various emerging applications. The specific contributions of this project are four-fold:

- Implementing IEEE-754 compliant floating-point functional units (e.g., fadd, fsub, fmul, fdiv, fsqrt, etc.) and deploying them either as MIPS standard instructions or as customized instructions.
- Presenting the floating-point optimization strategies for designs optimized for area, power, and speed; delivering a set of optimized implementation solutions and demonstrating their considerable performance improvements.
- Analyzing the problem characteristics, identifying corresponding performance requirements, and then dynamically constructing and configuring the optimal floating-point functional modules.
- Proposing a multi-mode pipeline flushing mechanism for the specific issues of fetch-again, branch likely flushing, and Extension instructions in the branch delay slot overwriting a taken branch.

2. RELATED WORK
The concept of using reconfigurable logic to improve the performance of floating-point intensive applications (10) is certainly not new but to date there has been little progress towards an actual implementation of this and related concepts in a complete system using dynamically reconfigurable FPGAs. In this section, we will list and present several major advancements in the FPGA-based floating-point architecture.

It is well agreed that floating-point operations are widely used in many areas. However, mapping of the complex floating-point arithmetic still make implementation of floating point on FPGAs difficult. Sahin et al. (4) demonstrate the implementation methodologies of various floating-point arithmetic operations such as addition, subtraction, multiplication, and division using 32-bit IEEE-754 floating-point. This shows that FPGAs are versatile devices for implementing many different applications.

FPGAs have appeared less attractive for floating-point intensive applications because these operations consume a large amount of resources. This motivated Chong et al. (8) to propose flexible, multi-mode embedded FPUs implemented on a single FPGA that can be configured to perform a wide range of tasks or be used to build massively parallel circuits. The floating-point adder and multiplier in their embedded FPU can each be configured to perform one double-precision operation or two single-precision operations in parallel. Moreover, the output of the floating-point multiplier can be internally connected to an input of the floating-point adder to perform a fused multiply-add operation.

Beauchamp et al. (7) present three architectural modifications that make floating-point operations more efficient on FPGAs. The first modification embeds floating-point multiply-add units in an island-style FPGA. The next two modifications target a major component of IEEE compliant floating-point computations: variable length shifters. The first alternative to lookup tables (LUTs) uses a coarse-grained approach for implementing the variable length shifters. The next alternative is a fine-grained approach, which adds a 4:1 multiplexer unit inside a configurable logic block (CLB) in parallel to each 4-LUT. Each of three modifications provides an area and clock rate benefit over traditional approaches with different tradeoffs.

Ho et al. (2) present an architecture for a reconfigurable device that is specifically optimized for floating-point applications. Fine-grained units are used for implementing control logic and bit-oriented operations, while parameterized and reconfigurable word-based coarse-grained units incorporating word-oriented lookup tables and floating-point operations are used to implement data paths. Such usage of multiple granularities offers remarkable advantages in terms of speed, density, and power over more conventional homogenous FPGAs.

Krueger and Seidel (5) present an on-line floating-point adder. In online arithmetic a result is computed as a digit serial output stream from digit serial input streams. The result digits begin to be produced a short delay after the first input digits arrive and before all the input digits have been received. On-line arithmetic proceeds from the most significant digit through the least significant digit. The major challenge for online FP addition particularly comes from the complexity of results that are simultaneously normalized and rounded.

Showing concerns to the excessive resource (or time) requirements for conventional implementations of floating-point operators, Shirazi et al. (9) examine the implementations of various arithmetic operators using two floating-point formats similar to the IEEE-754 standard. Such custom formats, derived for individual applications, are feasible on Custom Computing Machines (CCMs), and can be implemented on a fraction of a single FPGA. Herbordt et al. (11) perform the numeric analysis necessary to use variable-width floating-point units in different portions of the application pipeline. Unfortunately, such a sophisticated numeric analysis cannot yet be performed automatically.

3. FLOATING-POINT FORMAT REPRESENTATION

Floating-point numbers have the advantage of being able to cover a much larger dynamic range compared to fixed-point numbers.

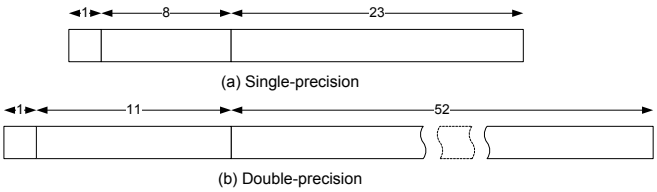

Figure 1 IEEE Floating-Point Numbers

However, correspondingly, they require a much more complex hardware implementation.

The IEEE-754 standard (12) (13) specifies a representation for single and double precision floating-point numbers. It is currently the standard that is used on most computing platforms. Floating-point numbers consist of three parts: sign bit, mantissa, and exponent. In the IEEE-754 format, the mantissa is stored as a fraction (*f*) that is combined with an implied one to form a mantissa (1.*f*). The mantissa is multiplied by the base number (two) to an exponent *e*, as shown in equation (1) and (2) for single and double precision, respectively (7) (14)

$$X = (-1)^{s} \cdot 1.f \cdot 2^{e-127} \quad (1)$$

$$X = (-1)^{s} \cdot 1.f \cdot 2^{e-1023} \quad (2)$$

As shown in Figure 1, a single precision float and double precision float consists of a 32-bit word and a 64-bit word (or 2 32-bit words) respectively. The IEEE standard specifies a sign bit, an 8-bit exponent, and a 23-bit mantissa for a single precision floating-point number. Double precision floating-point has a sign bit, an 11-bit exponent and 52-bit mantissa. The mantissa is normalized to the range [1, 2] as there will always be a leading one on the mantissa. By implying the leading one instead of explicitly specifying it, a single bit of storage can be saved, but it does raise the complexity of floating-point implementations.

Performing arithmetic on represented in this way require data paths including wide shifters, adders and multipliers. These elements can be implemented in different ways with varying levels of efficiency and performance. Our system will provide flexibility in making these tradeoffs.

4. DESIGN

In this and the following sections, we describe the design methodology we used to implement our CFPU. We describe how the CFPU Extensions work on the extensible processor platform, eMIPS [26]. These Extensions can be loaded and executed by software as a reconfigurable module. All floating-point operations are compliant with the IEEE-754 standard, and the floating-point extended instructions are designed to conform to the MIPS instruction format. This section focuses on a high level overview of the CFPU required functionality and how that is realized within the extension architecture, including the instructions we implemented. This includes data we gathered and analyzed from statically and dynamically profiling of floating point applications. The following sections describe the specific floating-point arithmetic algorithms, and discuss how they were implemented using the resources of the FPGA fabric. Examples of floating-point addition and multiplication are presented.

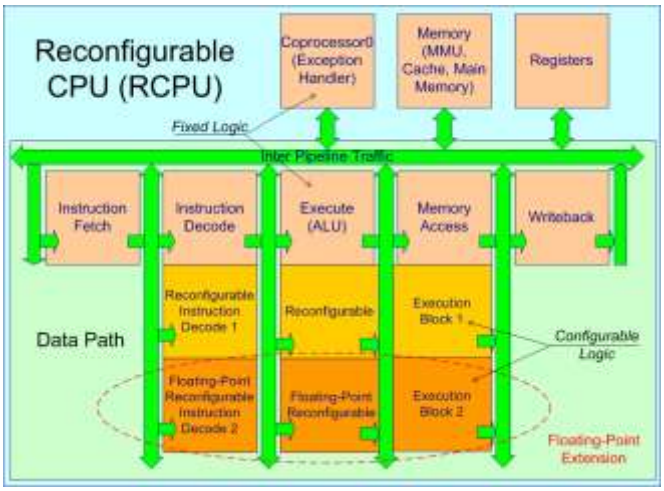

Figure 2 Reconfigurable Processor with a Configurable Floating-Point Extension.

4.1 System Overview

The eMIPS microprocessor is a prototype embedded microprocessor that utilizes an extensible instruction set architecture. It implements a base instruction set based on the MIPS R3000 RISC architecture with enhancements for incorporating reconfigurable logic into its data path. The reconfigurable logic can be used to implement custom logic circuits, called Extensions, for a variety of applications. These Extensions can be loaded into the reconfigurable logic dynamically during runtime in parallel to program execution (15). Figure 2 provides a block diagram of an extensible microprocessor like the eMIPS.

The eMIPS microprocessor prototype is implemented using FPGAs from the Xilinx Virtex IV and V families. The FPGA is partitioned into fixed and reconfigurable regions, as in (15). The fixed region consists of the most basic functions and security sensitive resources of the processor. The Extensions to the base processor make up the reconfigurable region of the FPGA and they are typically used to accelerate the software applications with dedicated circuits. This region is where we load/unload our CFPUs, using dynamic partial reconfiguration. Such Extensions can be triggered by new instructions particularly developed to meet the changing computing needs for various emerging applications. In addition, Extensions can be projected into several versions of optimization to meet the specific performance, area, or power requirements in different cases.

In practice, the most frequently executed instruction basic blocks are given to the hardware designers as specifications so that they can create a corresponding hardware module that implements the semantics of that sequence in the most optimized way (15). The hardware module can also be generated by a C-to-gates flow, or with an assembly to hardware compiler like the MIPS-to-Verilog (M2V) compiler (16) (17).

In our system, the implementation of the Extension and its usage in the application is hidden at the software level. There is no need to change the higher level compiler. Once the basic block is identified, another independent tool scans the assembly binary and looks up the occurrences of the identified basic blocks. The tool encodes the new Extension instruction to match the allocation of registers and immediate values, and inserts such an Extension instruction immediately before the instance of the basic block pattern.

When the application starts up on a platform where the hardware Extension configuration file exists, such Extension is loaded into a free Extension slot in the reconfigurable region of the FPGA. The Extension executes and the corresponding block of code it replaced is skipped. If the platform does not support such Extensions or if the security settings of the platform do not allow it, when the program reaches the newly added Extension instruction, the system simply ignores the extension instruction (i.e., treats it like a NOP) and executes the instructions that follow in the basic block sequence.

Once the Extension is active, it can access all pipeline resources, including the general purpose register file and the system memory. The Extension can read and write data to/from the register file. In practice, the Extension copies the data locally to its own registers for computation and writes back the results. For security purposes the Extension is bound to the virtual memory space of the process for which is being used. For this reason it accesses memory through a memory management unit (MMU) like the standard data path.

In the case of a CFPU, its first goal it is not to implement new instructions but to add support for the floating point instructions that are an optional part of the MIPS specification (18). Implementing single instructions that are embedded in the normal sequential execution stream using the existing system originally designed to optimize large blocks of instructions presented unexpected challenges. The issues encountered are discussed in detail in Section 9. Once this basic goal is met, we can also use CFPUs to implement new, optimized blocks just like with the existing eMIPS technology. This is discussed in Section 4.2 and in Section 8.

4.2 Floating Point Profiling and Optimization

Using a dynamically extensible microprocessor like the eMIPS, allows us the flexibility to add floating point capability at runtime for applications that require it. In addition, it gives us flexibility in how we provide that capability. If we were designing a microprocessor with a static design, we would have to include all the functionality we would ever need (i.e. a fully implemented FPU) at great cost in area and power for the underutilized functions. However, with an extensible architecture we can be more selective and tailor our CFPU to the application. In the event that the applications or operating conditions change the CFPU can be switched out for another. Using application profiling we can determine the precise requirements of the application and its execution pattern for further optimization.

In order to determine the floating point requirement of modern programs, we analyzed and profiled the floating-point instructions of 9 programs in the widely used SPEC CFP2000 benchmark suite (wupwise, swim, mgrid, applu, mesa, art, equake, ammp and apsi) and the Paranoia benchmark. We loaded the programs into a real time platform simulator, called Giano (19). Giano is full system simulator developed at Microsoft Research. Using Giano we can assemble a simulation of our microprocessor system including memory and peripherals and gather data on an application's execution patterns and other data. From Giano we were able to

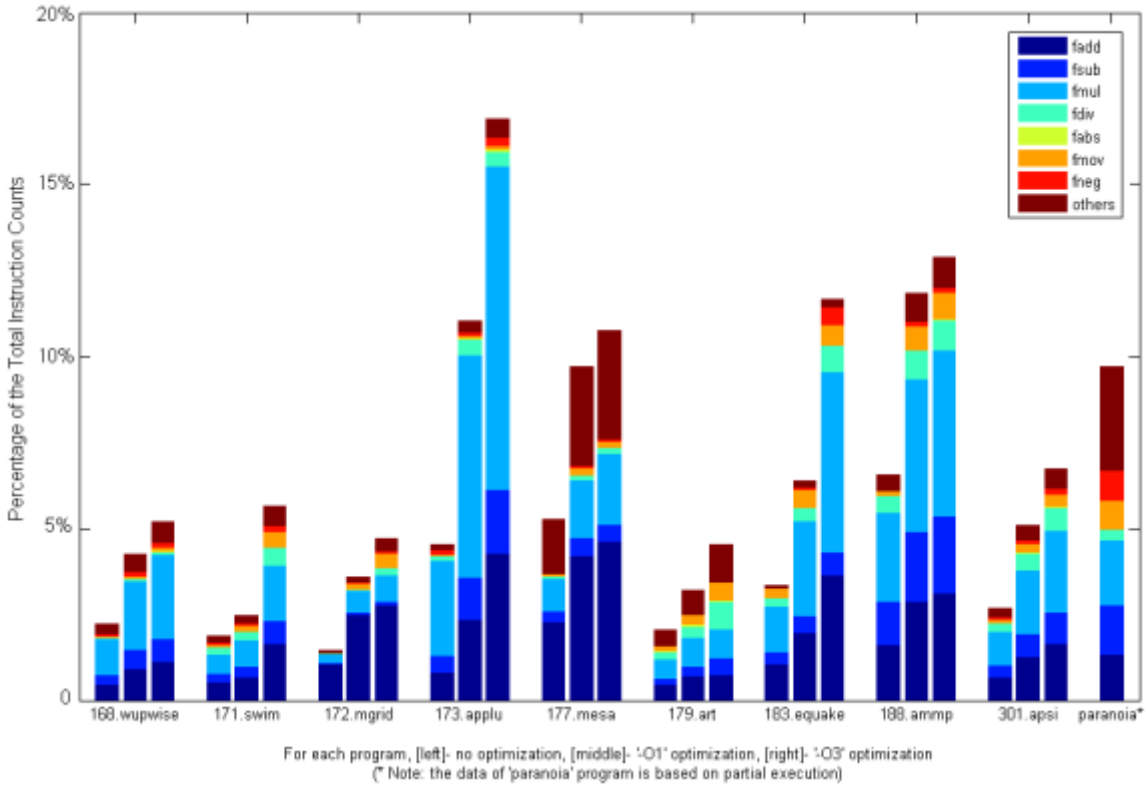

Figure 3 Floating-Point Instruction Profiling Results for SPEC CFP2000 Benchmark Programs

obtain both the static counts of float point instructions that appeared in the application binaries, and the dynamic count of the number of times each type of instruction was executed.

As shown in Figure 3, the floating-point add/sub and multiply operations occupy around 80% of all floating-point instructions present in the programs, making them top candidates for optimization. Several simple floating-point computational operations follow, like floating-point move, absolute, and negate operations. These are also good candidates due to their relatively simple structure, despite their less frequent use. The floating-point division is of concern because of its remarkably long latency (e.g., more than 60 cycles in some modern computer architectures (19)). Based on the profiling results shown in Figure 3, we are able to identify the most significant floating-point operations which account for 95% of all floating-point related computations. The benchmarks were compiled for different levels of optimization. As the optimization level increased, the impact floating point instructions on the overall execution count grew significantly.

Static analysis of the application program tells us what instructions are definitely not needed in the CFPU, and this can already lead to noticeable space savings over a complete, general purpose FPU. Consider, for instance, the case of a program that only uses either single-precision or double precision floats. Double precision implementations require twice the resources of the single precision or more. Eliminating either the single or double precision logic in a CFPU implementation can reduce the area by at least a third or more than two-thirds respectively. Additionally, a program that never uses a square-root and other expensive instructions would be served just as well or better by a CFPU that omitted that function in favor of other optimizations for even better performance.

Dynamic analysis tells us what instructions are actually used, how frequently, and with what expected latency. This leads to three optimization ideas. First, we can save area and leave out of the CFPU the instructions that are compiled but never executed, provided we have a fall-back solution in case they do get executed in a deployed system. In other words, we can create a general-purpose FPU but actually use a CFPU until/unless we get in trouble, at which point we switch to the general-purpose FPU or fall back on software emulation. Secondly, using the latency data we can select the best implementation of the add/sub/etc. modules according to their use in the critical paths of the application program. This leads to better performance, and possibly some area savings. Thirdly, we can identify the sequences of floating-point operations that are most frequently executed together (e.g., in a basic block of code), and realize them in the extension as a single instruction. This leads to large execution speedups, but potentially with large area costs. Ideally, the CFPU will be no larger than the general-purpose FPU, but the special instructions will realize considerable time and energy savings.

4.3 Floating-Point Registers

The MIPS specification allocates a separate register file for use with each floating point unit (18). Figure 4 provides a high level diagram of the MIPS float point register file This register file is

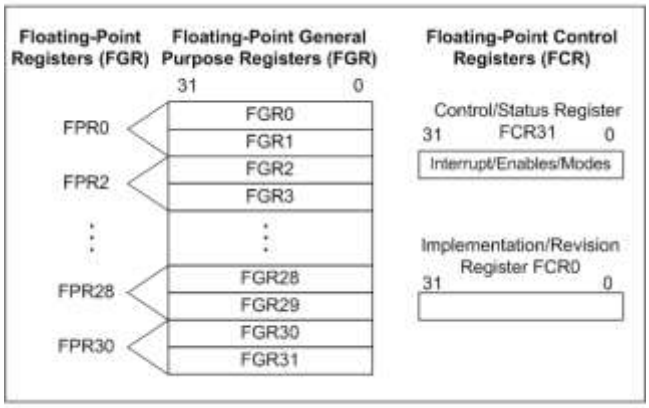

Figure 4 MIPS Floating-Point Registers

included as part of all CFPU Extensions. Within this register file the specification defines three types of registers:

- Floating-Point General Purpose Registers (FGRs)
- Floating-Point Registers (FPRs)
- Floating-Point Control Registers (FCRs)

FGRs are directly addressable, physical registers. There are 32 FGRs, each of which is 32-bits wide and individually accessible through move, load, and store instructions. FPRs are logical registers used in floating-point operations. Each of the 16 FPRs is 64-bits wide and formed by two adjacent FGRs. This logical structure provides the capability to support either single or double precision floating-point operations.

The FGRs and FPRs are implemented using a dual port block ram (BRAM) within the reconfigurable region allocated to the CFPU on the FPGA. One port is used for data related instructions (move, load & store). Another port is available for arithmetic instructions to fetch operand data in parallel, To support both single and double precision floating point operations and support interaction with the 32-bit standard data path, the BRAM is configured with two 64-bit physical ports that are logically mapped to two 64-bit logical ports and two 32-bit logical ports using multiplexors and byte write enables controlled by the address input of each port.

The FCRs store the control information regarding rounding modes, exception handling, and register status. Figure 4 shows only two FCRs because only FCR0 and FCR31 are defined in the specification. All other FCRs are reserved for future expansion. The FCRs include the *Control/Status Register* and the *Implementation/Revision Register*. The Control/Status Register (FCR31) monitors and handles exceptions, holds the result of compare operations, and defines rounding modes for all floating-point operations. The Implementation/Revision Register (FCR0) keeps the version and revision information.

The FCRs are implemented using a dual port BRAM configured to 32-bit wide ports. One port is used for read and writes to support move, load and store instructions. FCR31 contains control bits that are used to control the behavior of the CFPU. The second port of the BRAM has its address set to the position of FCR31 and the bits of the data output connected to the CFPU control logic. If a status bit requires an update from the control logic the data output is masked with the changed status bit before connecting to the data input and the write enable asserted.

Table 1 Implemented Floating-Point Summary

Instructions	Descriptions
ABS.S	Floating-Point Absolute Value
ADD.S	Floating-Point Add
BC1F/BC1FL	Branch on FPU False (Likely*)
BC1T/BC1TL	Branch on FPU True (Likely*)
C.COND.S	Floating-Point Compare
CFC1/CTC1	Move Control Word From/To FPU
CVT.S	Convert to Single Floating-Point Format
CVT.W	Convert to Fixed-Point Format
DIV.S	Floating-Point Divide
LWC1/SWC1	Load/Store Word to/from FPU
MFC1/MTC1	Move from/to FPU
MOV.S	Floating-Point Move
MUL.S	Floating-Point Multiply
NEG.S	Floating-Point Negate
SUB.S	Floating-Point Subtract

(*: "Branch Likely" instructions were introduced to reduce the branch penalty using statically predicted branches since MIPS R4000, due to the longer pipeline.)

4.4 Floating-Point Instructions

Compliant with the MIPS architecture (18), the reconfigurable floating-point modules in the eMIPS architecture support the following floating-point instructions:

- Load and store operations from/to the floating-point registers;
- Moves between floating-point and CPU registers;
- Computational operations including floating-point add, subtract, multiply, divide, and convert instructions;
- Floating-point comparison and branch instructions.

Table 1 provides a summary of the floating-point instructions currently implemented as an Extension in the eMIPS architecture. The programs profiled when determining floating point usage for the most part only utilized single precision floating point. In the few cases we found double precision we were able to replace it with single precision with small modifications to the source. For this reason, we choose to focus on single precision implementations for this study. The focus of this work is the use of reconfiguration to provide greater flexibility. Future development plans do include development of double precision function units.

It is always necessary to move data in and out of the CFPU; therefore all versions of the CFPU will support the instructions for move to (MTC1), move from (MFC1), load (LWC1), and store (SWC1). These instructions are the only means by which data can be sent to the CFPU FGRs and retrieved from it. The move instructions are implemented through the Extension register interface. The move control to (CTC1) and move control from (CFC1) instructions work just like MTC1 and MFC1, except they target the CFPU FCRs. The load and store instructions use the same memory interface used by the data path to fetch instructions and read/write data. They go through a MMU and can result in a TLB miss. In the event this occurs the CFPU is interrupted and the FPU load or store is reported as the causing instruction. After the TLB miss is serviced the exception handler will return to application execution at that instruction and complete the load or store operation. Other reconfigurable processor architectures currently available would encounter difficulty implementing a fully functional FPU in their configurable hardware. Most of these architectures do not give the configurable hardware access to memory, or they use a dedicated direct link to physical memory. This approach is insecure because it does not support virtual memory through an MMU like ours. Cache coherency can also be an issue for specialized links to memory.

During execution, the Extension hardware can modify the current PC of the processor when it is granted access to the pipeline resources. This is used by the Extension to skip basic block code when it is activated, and to perform conditional branches found at the end of basic blocks. The CFPU implements the branch instructions (BC1F, BC1FL, BC1T, BC1TL) using this same infrastructure. Branch instructions allow flow control using only floating point data, without having to perform move instructions back to the data path to use its integer flow control instructions. In practice, software executes first a floating point compare (C.COND.S) instruction to compare the values of two floating point numbers stored in the FGRs. Based on the COND field, the appropriate comparison is performed (less than, greater than, equal, etc) and the C flag in the Control/Status Register (FCR31) is set to true (1) or false (0) accordingly. Then the program executes one of the branch instructions that test the value of the C flag to decide whether to take the branch or not. Floating point branches are similar to the other branches in the MIPS ISA, and include in their semantic a branch delay slot. Handling of branch delay slots from within an extension was an eventuality not considered in the original eMIPS design. This resulted in errors at first, and eventually required some unavoidable modifications of the Extension interface pipeline interlock. More details on this are presented in Section 9.

The absolute value (ABS.S) and negate (NEG,S) instructions are very simple instructions that modify only a single bit in the target FGR, the sign bit. This requires very little hardware to implement and a single cycle in the pipeline. These functions can be realized by emulation in the datapath, using the logic functions AND and XOR respectively. Notice that emulation must keep floating point data in the general purpose registers anyways, so there is really no penalty in this case. For similar reasons, when using a CFPU these instructions should also be supported in the CFPU. There are no performance benefits, but their logic foot print is small and any fall back to the datapath would require moving the data to the datapath and back at greater expense than the hardware.

The arithmetic operations, including add, subtract, multiply and divide are the most complex ones and often get the most attention

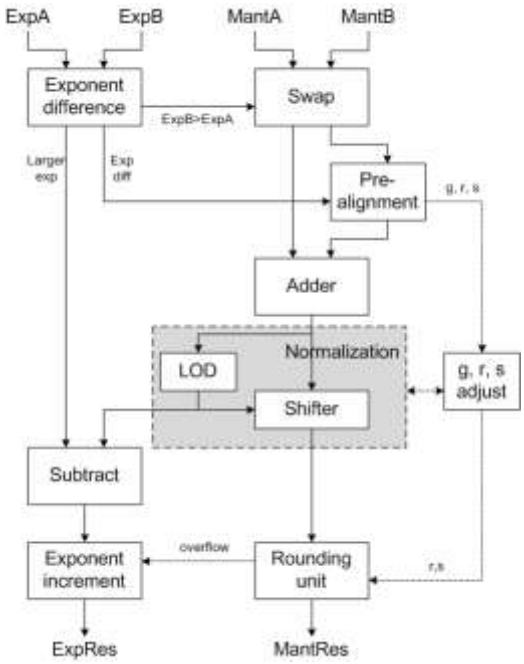

Figure 5 Floating-Point Adder Datapath

during implementation. Since they act on their data stored locally in the CFPU FGRs, the hardware carrying out these operations can run in parallel to normal program execution. In this way, a programmer or compiler can hide the latency of these operations to some degree by sending the operand data to the CFPU if it is not already present and executing the operation without waiting for the results. When the CFPU sees the instruction it can begin execution on its local data and immediately release the datapath to other tasks. When the datapath needs the data, it executes either the move from or store instructions. If the CFPU is finished it will move the data, otherwise at this point the CFPU will stall the data path until execution is complete. Reducing latency of these operations is still important because the amount of parallel work that the datapath can do is often limited and the datapath will likely incur a stall, waiting on the CFPU to finish.

The following sections presents more detailed information on the floating point algorithms and implementation details.

5. FLOATING POINT ARITHMETIC

Floating-point numbers can cover a much larger dynamic range compared to fixed-point numbers (8). However, the disadvantage is that usually floating-point computations are much more complex to implement in hardware. Designing efficient high performance implementations is especially important for frequently used floating-point computational operations, like add, sub, mul, and div. In the following sections, we will briefly introduce the core floating-point algorithms and their implementations for the frequently used floating-point adder, multiplier and divider.

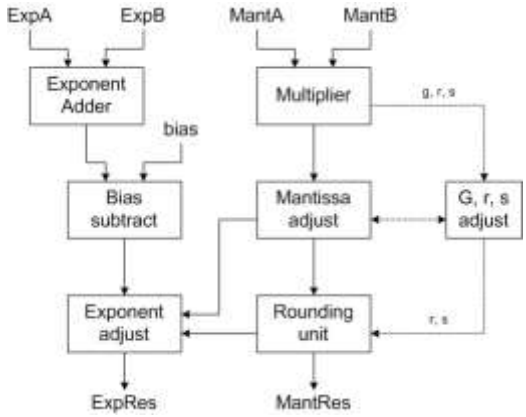

Figure 6 Floating-Point Multiplier Datapath

5.1 Floating-Point Addition

The conventional floating-point addition/subtraction algorithm normally consists of five stages: exponent difference, pre-alignment, mantissa addition, post-normalization and rounding (14). Given two floating-point numbers $X_1 = (s_1, e_1, f_1)$ and $X_2 = (s_2, e_2, f_2)$, where the notation s_i, e_i, and f_i represent the sign field, exponent field, and fraction field of the floating-point number respectively, the algorithm for computing $X_1 + X_2$ is described as follows:

1) Find the exponent difference $d = e_1 - e_2$. If $d < 0$, swap the mantissas. Subtract e_2 from e_1 to calculate the number of positions to right shift f_2. Add the leading bit to the very beginning of mantissas $(1.f_1)$ and $(1.f_2)$.
2) Pre-align the mantissas by shifting $(1.f_2)$ right by d bits.
3) Add or subtract the mantissas to get a tentative result *t*. Set the sign and exponent of the final result to the sign and exponent of the value which is greater.
4) Normalize. If there are leading zeros in the tentative result *t*, shift the result left and decrement the exponent by the number of leading zeros. If there is an overflow, shift right one bit and increment the exponent.
5) Round the mantissa. If there is an overflow, shift right again and increment the exponent.

Figure 5 shows the datapath for a floating-point addition operation. It is known that the pre-alignment and normalization stages require large shifters. The pre-alignment stage requires a right shifter with the size of up to 48-bits for single precision inputs because the parts shifted out need to be stored to calculate the guard, round, and sticky bits for rounding. The normalization stage requires a left shifter equal to the number of mantissa bits plus 1 (to shift in the guard bit), i.e., 25-bit for single precision. The final rounding stage is determined by the rounding mode, the LSB of result mantissa, the round bit and the sticky bit. A "1" will be added into the LSB of the result mantissa if the rounding is required.

5.2 Floating-Point Multiplication

Floating-point multiplications are actually simpler than floating-point addition. Given two floating-point numbers $X_1 = (s_1, e_1, f_1)$ and $X_2 = (s_2, e_2, f_2)$, the multiplication $X_p = X_1 \times X_2$ can be computed as follows:

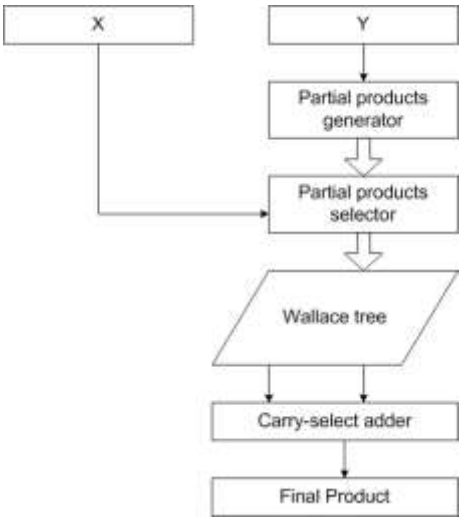

Figure 7 Booth Wallace Multiplier Structure

$$\begin{cases} s_p = s_1 \oplus s_2 \\ e_p = e_1 + e_2 - bias \\ 1.f_p = 1.f_1 \times 1.f_2 \end{cases} \quad (3)$$

Figure 6 shows the datapath for a floating-point multiplier. The result from the multiplier can have at most two bits left of the binary point, in which case the mantissa is shifted right to compensate and the exponent is increased. If the rounding of the mantissa generates an overflow, the mantissa is shifted right by one and the corresponding exponent is incremented.

The key part of the floating-point multiplication is an integral multiplier – 24×24 bits unsigned for single-precision. In our work, we implemented a Radix-4 Booth Encoded (MBE) Wallace (20), as shown in Figure 7. Radix-4 encoding halves the number of partial products, thus reducing the number of levels required in the Wallace tree. For more on Booth encoding, refer to (21) (22). For more details on floating-point arithmetic algorithms, please refer to (23) (24) (25).

5.3 Floating-Point Division

The latency for floating-point addition is typically between two and four cycles, and two to eight cycles for floating-point multiplication. The latency for double precision floating-point division in modern FPUs ranges from eight cycles to more than 60 cycles (19). Designers generally put a lot of effort into the sophisticated design of floating-point addition and multiplication, but usually overlook the division implementation because of its relative infrequency. Unfortunately, it has been shown that ignoring its implementation will result in significant system performance degradation for many applications (26).

The simplest and most widely implemented division algorithm is digit recurrence, with low complexity, small area, and relatively large latency. The digit recurrence algorithm uses subtractive methods to process a fixed number of quotient bits in each iteration. As with the addition and multiplication, the input operands are normalized floating-point numbers compliant with the IEEE-754 standard with an *n*-bit mantissa. The typical division is defined in the following way:

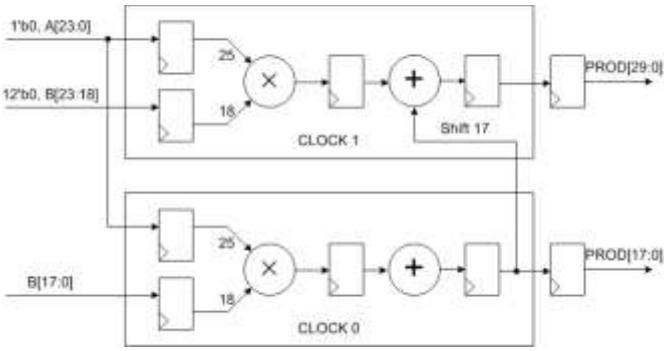

Figure 9 24x24 Two's Complement Multiplier

$$x = q \times d + rem \tag{4}$$

where

$$|rem| < |d| \times ulp \text{ and } \text{sign}(rem) = \text{sign}(x)$$

The *dividend x* and *divisor d* are the input operands. The *quotient q* and, optionally, the *remainder rem* are the results. The *Unit in the Least Position*, or *ulp*, defines the precision of the quotient, where $ulp = r^{-n}$ for n-digit, radix-r fractional results.

The following equations are used in each step of recurrence:

$$rP_0 = dividend(x) \tag{5}$$

$$P_{j+1} = rP_j - q_{j+1} divisor(d) \tag{6}$$

where P_j is the *partial remainder*, or *residual*, at iteration j. Initially, the partial remainder P_0 is set to the value of the dividend. In each iteration, one digit of the quotient is determined by the quotient-digit selection function:

$$q_{j+1} = SEL(rP_j, d) \tag{7}$$

Then, the final *quotient* after k iterations is

$$q = \sum_{j=1}^{k} q_j r^{-j} \tag{8}$$

And the *remainder* is computed from the final residual by:

$$\text{remainder} = \begin{cases} P_n & \text{if } P_n \geq 0 \\ P_n + d & \text{if } P_n < 0 \end{cases} \tag{9}$$

As shown in above equations, the key components within the floating-point division can be summarized in the following steps (19):

1) Determine the quotient digit q_{j+1} for the next iteration using the quotient-digit selection function.
2) Multiply q_{j+1} by the divisor d.
3) Subtract the product of $q_{j+1} \times d$ from the shifted partial remainder $r \times P_j$.

In this study, we only implemented the classic subtractive digit recurrence algorithm for the floating-point division module and demonstrated its functionality with the reconfigurable architecture. Various techniques have been proposed for further increasing division performance, all of which present certain level of trade-offs in the time/area design space. Further details may be found in (23) (24) (26) (19).

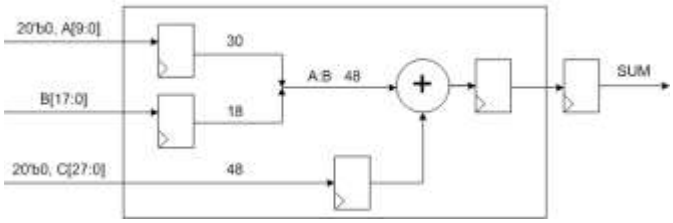

Figure 8 Two Input 28-Bit Addition

6. Xilinx XtremeDSP Modules

For Xilinx Virtex-4 or Virtex-5 series FPGAs, the XtremeDSP Digital Signal Processing DSP48/DSP48E slices have been integrated as new elements, which are referred to as Application Specific Modular Blocks (ASMBL) in the Xilinx literature. The purpose of this model is to deliver off-the-shelf programmable devices with the best mix of logic, memory, I/O, processors, and digital signal processing (27) (28). The DSP48E slice supports many independent functions, including multiply, multiply add, three-input add, barrel shift, bit-wise logic functions. They also can be cascaded to form wide math functions. To achieve the highest resource saving, we will use and reuse as many of these slices as possible. One of the best choices is to fully utilize the versatile capability of DSP48E. The frequently used full-length shifters, addition and subtraction operations, multiplication, as well as division all will be built using DSP48E slices.

6.1 24 x 24 Two's Complement Multiplier

The two's complement multiplier inside the DSP48E slice support two 25-bit × 18-bit, two's complement inputs and produces a 43-bit, two's complement result. For the floating point multiplication we need a 24 x 24 multiply for the mantissas. We can cascade two of the multipliers to achieve the larger product. The DSP facilitates this cascading using an internal 17-bit right-shifted cascaded bus input to the adder/subtractor to right adjust partial products by appropriate bits (27). The signed 24 × 24 multiply may also be implemented using a single DSP48E slice. Both the 24-bit A and B operands are assumed to be signed two's complement numbers and thus corresponding results will be a signed, two's complement, 48-bit number. Since the operand B is beyond the limitation of the internal 25 × 18 multiplier, we need to split the operand B into two parts, which will be sent into the multiplier in two consecutive cycles.

When separating two's complement numbers into two parts, only the most-significant part carries the original sign. The least-significant part will be considered as positive operands and assigned a "forced zero" in the sign position. In this case, we will equally partition the 24-bit operand B into two 12-bit parts. The product of operand A and lower part of operand B will be connected with cascaded bus input which has integrated a 17-bit right shifter and be added with the product of operand A and higher part of operand B in the next cycle. The final product result can be concatenated with parts of those two partial products, as shown in Figure 9. Similarly, you can achieve the same result in one cycle using two DSP slices. Given the finite availability of DSP slices in our target device, we chose the single slice implementation to conserve these resources.

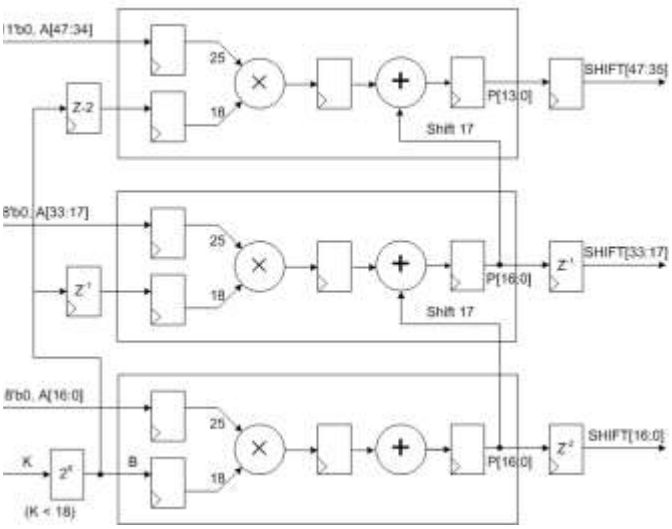

Figure 10 48-Bit Dynamic Shifter

6.2 Two Input 28-Bit Addition

The DSP48E slice can be easily configured as a full 48-bit adder/subtractor, where the 30-bit input A and 18-bit input B are concatenated to form a 48-bit operand and the other operand directly comes from 48-bit wide input C. In our floating-point add/sub operations, either a 28-bit adder or a 28-bit subtractor needs to be used for the aligned mantissas. This smaller operand addition/subtraction in the DSP48E slice requires sign extension all the way up to the 48th bit for the C input C[47], and A input A[29], as shown in Figure 8. By changing the ALUMODE parameter, the specific addition and subtraction operations can be dynamically selected based on the effective operations decided in the pre-alignment stage. This structure can be reused during the different stages of both the floating point addition and multiplication datapaths to conserve resources.

6.3 48-Bit Dynamic Shifter

Using the internal multiplier, the DSP48E slice can behave as logical shifter, arithmetical shifter, or barrel shifter. One of the inputs passing through the 2^K can control the other input to shift to the left by K bits. Meanwhile, the output could be also regarded to be shifted right by 18-K bits. Since the multiplicand and multiplier supported by DSP48E are 25-bit and 18-bit, that is, the input to be shifted, A, can be up to 25 bits wide. However, the shift count value, K, is limited to be a maximum of up to 17 bits.

If A is an 18-bit number, a left shift by K bits is expressed by the lower 18 bits P[17:0] of the multiply result with the shift control value 2^K. Correspondingly, the higher 18 bits P[35:18] of the multiply result will represent the input A shifted by 18-K bits and sign-extended.

In floating-point operations, full-length shifters are excessively used in the pre-normalization and post-normalization stages, and thus become one of performance bottlenecks as well as the most resource consuming components. In the design for area, we implemented all internal shifters using the DSP48E slices. Particularly, a single slice is used with multiple clock cycles creating partial results to be combined at the very end of the computation. Without losing generality, we give an example of the 48-bit full-length shifter used in the post-normalization stage of float-point multiplication, which is also the most complicated shifter used among all single-precision floating-point operations.

As shown in Figure 10, we partition the original 48-bit multiplicand into three sections: A[47:35], A[34:18], and A[17:0], each of which will be sent to the multiplicand port in three consecutive cycles. The output will be shifted right by 17 bits and sent back to the addition block for the next cycle, in order to let the lower part fill the blank space left the higher part during the left shift. Here, we only show the case of shifting by less than 17 bits. For the cases of shifting by larger than 17 bits, they could be expanded and implemented in a similar way. It is shown that, such a full-length 48-bit left shifter could be implemented using a single DSP48E slice in a fully pipelined way. Compared with the traditional way of cascading several DSPs in series, this technique is able to significantly reduce the on-board resource usage at a cost of two extra cycles.

It is worth mentioning that the flexibility of such type of shifter based on DSP48E slice is multifaceted. That is, it is very easy to implement either a right shift or a left shift, as well as either a logical shift or an arithmetic shift. The only thing that is required is to change the ALUMODE or OPMODE parameters in the DSP48E slice. The only exception happens when the shift number K is 17, which will be recognized as a negative number, since the internal multiplier of DSP48E only works for two's compliment operands. In this case, the user needs to set the ALUMODE parameter to compensate for this.

7. Optimization Design Strategies

In order to meet the distinct application-specific demands on custom floating-point functional modules, we propose to explore and integrate multiple versions of the FPU into our custom design platform. In this study, we provide a basic example of implementing CFPUs for speed, area, and power targets, all utilizing the physical resources of FPGAs. It should be mentioned that such explorations only present a case study on how the users' specific demands can be addressed by certain dedicated and optimized designs. There are many more sophisticated algorithmic and systematic optimization methods available for floating-point units, which are beyond the scope of this study and cannot be explicitly addressed in this paper.

7.1 Design for Speed

To achieve the maximum performance in the CFPU design, it is desirable to use all possible pipeline stages within the algorithm. To optimize the functional units for add, subtract, multiply and divide, the datapaths should always be fully pipelined. Each functional unit's pipeline contains three stages: pre normalization, execute and post normalization. The cycle time of each pipeline stage in some cases is dependent on the operands. This is the case in the pre and post normalization stages where operands are shifted into alignment and leading zeros shifted out respectively.

Using this pipeline, the CFPU can have up to three operations in flight at the same time. As in all pipeline architectures, this technique does not necessarily improve the execution time of a single operation however it increases the throughput by a factor roughly equal to the depth of the pipeline. Pipelining prevents most opportunities for hardware reuse, because a pipeline that is filled performs all operations in parallel, on different operands.

For some cases the increased throughput is worth the higher hardware cost.

7.2 Design for Area

To minimize the area consumption, we need to fully utilize the integrated arithmetic modules on the FPGAs, especially the DSP48. We can time multiplex the DSP48E slices, that is, implement more than one function within a single DSP48E slice at different times. Time-multiplexing leads to more compact designs, but typically results in a lower throughput.

In all of the floating point algorithms there are steps that require addition and subtraction operations. The largest of these is the 28 bit add/subtract of the mantissas in the floating point add/subtract. To conserve area, a design can implement one of these DSP adder/subtractors for that step and reuse it during other stages of the algorithm where non trivial additions/subtractions are required like the difference of the exponents.

The largest bottleneck to floating point performance and largest potential hardware cost are the wide shifters used during pre and post normalization. It can be observed that shifting is the same as multiplication or division by a power of two in binary. Using this fact, we can use the same DSP structure for both shifting and performing the multiplication operation by adjusting the number of cycle to multiply, shift and add between two for the multiply and three for the shift.

By reducing the numbers of functional units required by the CFPU we can significantly compact the design. Unfortunately it is not all for free. Time multiplexing the functional units in this way significantly increases the number of steps and cycles required to perform the operations. It also requires more registers to store intermediate results and multiplexers for routing data. Overall the resources utilized by this method are still less than the fully pipelined implementation. Logic utilization statistics from design synthesis are presented in Section 12; they show an overall area reduction of about a fourth.

7.3 Design for Power

In the design-for-area scheme, the highly reused programmable resources have already provided good power consumption efficiency. We can further reduce power consumption at the board level by exploring more aggressive strategies. For example, in the DSP48E module, the USE_MULT attribute selects the usage of the Multiplier. This attribute should be set to NONE to save power when using only the Adder/Logic Unit. In addition, functions implemented in the DSP48E slice use less power than those implemented in the rest of the fabric. Using the cascaded paths within the DSP48E slice instead of fabric routing is another way to reduce power. According to the specification of the Xilinx Virtex-5 FPGA, a multiplier with the M register turned on uses less power than one where the M register is not used (27).

8. FLOATING-POINT RECONFIGURABLE EXTENSIONS

So far we have considered standard instructions and how to best create CFPUs that use them. The focus has been on reducing the number and size of our circuits, either by removing functionality or by optimizing the implementation. Now we explore the packing of larger sets of floating-point and non floating point operations from a basic block into a single CFPU extension instruction.

Table 2 Identified Basic Block Example in the LINPACK Benchmark

```
ENTRY(bb153)
  ExtInstruction(EXT,v1,a1,dimm)
  lwc1    $f0,-16($a2)
  lwc1    $f1,-16($v1)
  lwc1    $f2,-12($v1)
  mul.s   $f0,$f5,$f0
  lwc1    $f3,-8($v1)
  lwc1    $f4,-4($v1)
  add.s   $f1,$f1,$f0
  swc1    $f1,-16($v1)
  lwc1    $f0,-12($a2)
  mul.s   $f0,$f5,$f0
  add.s   $f2,$f2,$f0
  swc1    $f2,-12($v1)
  lwc1    $f0,-8($a2)
  mul.s   $f0,$f5,$f0
  add.s   $f3,$f3,$f0
  swc1    $f3,-8($v1)
  lwc1    $f0,-4($a2)
  mul.s   $f0,$f5,$f0
  add.s   $f4,$f4,$f0
  swc1    $f4,-4($v1)
  slt     $v0,$a1,$a0
  addiu   $v1,$v1,16
  addiu   $a1,$a1,4
  bnez    $v0,bb153
  addiu   $a2,$a2,16
END(bb153)
```

As presented in section 4.1, the eMIPS platform has been designed to use reconfigurable logic to accelerate program execution by implementing frequently used basic blocks in hardware. This has demonstrated performance improvements between 2x and 10x on fixed point general purpose applications in previous work (15). Now that we have implemented float point functional units we have the opportunity to apply the same technique to floating point programs and further increase the flexibility and capabilities of our CFPU model.

Like in section 4.2, we loaded the LINPACK bench program (29) (30), presented in section 11.2, into our Giano simulation of the eMIPS microprocessor. The simulation produced the specification of all the basic blocks found in the program binary and ranked them by the number of times the basic block was executed over the time of the simulation.

Table 2 shows the basic block identified as the most frequently executed block in the LINPACK benchmark program (29) (30). Note that this block is the most frequently used during dynamic

execution rather than the most frequent one found during the static profiling. This basic block implements the multiply-add operations for four pairs of operands stored contiguously in main memory. Depending on the entry values, it might be executed many times. The first instruction shown in Table 2 is the inserted Extension instruction, shown with the register assignments based on the block analysis. Our Extension implements the same operations as the identified basic block, including the conditional branch instruction.

In the basic block of Table 2, the registers A2 and V1 are pointers that refer to the memory that contains two rows in the matrix A. The basic block performs a Gaussian elimination on A by multiplying the elements in the row at A2 by a constant stored in register F5 and adding these to the corresponding elements in the row at V1. The loop is unrolled four times so four such elements are processed per iteration. At the end the pointers are incremented by sixteen, or four data elements, for the next iteration of the loop and the loop count stored in A1 is incremented by four. The loop count is compared with the terminating count given by A0. While A1 is less than A0, the loop continues.

In this basic block we see a loop unrolled four times that reads data from memory, multiplies by a number, then loads another number from memory to add this to it and save that result back to that memory location. The compiler already attempts to deal with the floating point latency by unrolling the loop and interlacing floating point multiplies and adds with the loads and stores.

The Extension derived from this basic block can already improve performance considerably just by eliminating the instruction fetch and the loop control instructions, as we have presented in our previous work. The Extension can further optimize by taking to the next level the interleaving of instructions that the compiler has already attempted. These can be better parallelized by queuing up the memory reads and later the memory stores to occur in parallel with the floating point operations. As it arrives from memory, the data can be fed into the pipeline that performs the multiply and the add, back to back. Then the result is written back to the FPRs at the same time it is queued up to be stored back to memory. Since this basic block loops back on itself or exits the loop at the end, the circuit can restart after all the state values like the pointer values and loop counter are updated, without going back to the datapath. It would also be possible to fully pipeline the entire execution of the loop rather than on a per iteration basis, further reducing the control overhead of the computation. In this way, the entire execution of the loop can be performed in hardware unless interrupted by an exception like a TLB miss or interrupt.

In case of an interrupt, the hardware will stop and escape out, allowing the software to handle the exception. Before the Extension releases the pipeline, the Extension cleans up state by writing back state holding registers such as pointers and loop counters and saving finalized data back to memory and registers. After the exception is handled the program will return to the software version of the basic block in the same logical spot it was when the circuit was interrupted and finish that iteration of the loop. If another iteration is required, it will loop back and return to the hardware to finish unless interrupted again. This mechanism for handling traps and interrupt is unique to eMIPS, and is presented in more detail in (17).

9. MULTI-MODE PIPELINE FLUSHING MECHANISM FOR EXTENSIONS

The eMIPS pipeline interlock for instruction decoding and Extension activation was originally designed to allow (large) sequences of instructions to be replaced by a single Extension instruction. This instruction is inserted into the binary in front of the first instruction of the basic block it replaces. If the Extension is available and active, the hardware module associated with the Extension instruction is activated and after completing execution it changes the PC to the address for the instruction following the last instruction in the basic block. Thus the instructions making up the basic block following the Extension instruction are not actually executed by the datapth (15).

The MIPS microprocessor has a five stage pipeline, of which instruction decode is the second stage. By the time we recognize that instruction I_n is an Extension instruction, the following instruction I_{n+1} has already entered the pipeline. Because the Extension instruction typically overrides it, I_{n+1} will not be executed and we require a flush of the instruction fetch stage. Additionally, the Extension instruction is likely to execute for multiple clocks, stalling all instructions that follow I_{n+1}. In fact, the first instruction that will actually be executed after I_n is selected by the Extension itself, either as a branch or a fall-through after the end of the basic block. For this reason, we designed an automatic pipeline interlock to flush and stall the instruction fetch stage when an Extension is activated (18).

This worked well until we considered implementing the floating point instructions of the MIPS ISA in an Extension. The floating-point instructions can be thought of as an extreme case of a basic block, one with a single instruction in it. The existing eMIPS design therefore presented us with several problems. Clearly, it is inefficient to flush the instruction following the floating point instruction only to fetch it again in the next clock. A considerably more difficult issue involved branches both inside and outside the Extension.

The instruction I_n that follows a branch is fetched by the instruction fetch stage of the pipeline before the branch instruction is decoded and the outcome of the branch is determined. The MIPS designers solved this problem by including the instruction I_{n+1} in the semantic of their branch instructions (the so-called branch delay slot). This semantic defines that the instruction following the branch is executed whether the branch is taken or not. Later on, a new form of branch was introduced (branch likely instructions) where instruction I_{n+1} is flushed if the branch is not taken (18).

A first problem exists if the floating point Extension instruction is located in the branch delay slot of a regular branch. As mentioned above, it is the Extension that selects the address of the next-PC. In the case of the floating point arithmetic instructions, this is the next sequential instruction. If the instruction before the floating point instruction is a taken branch, the next PC should be the branch target not the PC the extension attempts to write to the instruction fetch stage. Second, if the floating point instruction is a floating point branch likely (i.e., Branch on True/False Likely instructions), it would either flush the delay slot if not taken or not flush it if taken. For this reason, the always-flushing or never-flushing configurations are insufficient.

We solved these issues by making a few alterations to the eMIPS pipeline interlock. We added an additional signal from the Extension interface to the static part of the design that allows the Extension to control whether a flush of the instruction fetch stage of the pipeline is necessary. This solves both the fetch-again issue and the branch likely flushing issue. To solve the issue of the Extension in the branch delay slot overwriting a taken branch, we added additional logic to the instruction fetch stage of the pipeline to prevent an Extension from changing the PC immediately following a taken branch. Note that placing a branch in a delay slot has undefined semantics. These considerations illustrate the difficulties of placing an extension instruction in the branch delay slot. Only cases of new or optimized instructions like the FPU instructions will likely execute from within a branch delay slot.

10. TESTING

The testing and verification of floating-point units have long presented a unique challenge in the field of processor verification, due to the highly complex operations involved in floating-point arithmetic, such as rounding and normalization. The largest obstacle to float point verification stems from the vast test space, which includes many corner cases that need to be targeted, and from the intricacies of the implementation.

Mainstream test generation tools, such as Genesys (31) and AVPGEN (32), offer some assistance generating floating-point test patterns. However, their lack of focus and internal knowledge of the floating-point domain render them inadequate for providing a full solution to the floating-point verification problem. Test generation was supplemented by static legacy tests and by large quantities of purely random testing, mostly useless.

In this study, we use SoftFloat (33), a free, high-quality software implementation of the IEC/IEEE Standard for Binary Floating-point arithmetic, to test specific floating-point functional units. All functions dictated by the IEEE-754 Standard are supported except for conversions to and from decimal. SoftFloat fully implements single-precision (32 bits) and double-precision (64 bits) floating-point formats as well as the four most common rounding modes: round to nearest even, round up, round down, and round toward zero. Table 3 presents an example of the test vectors produced by SoftFloat.

We used SoftFloat as the Oracle in an online, automated testing system in conjunction with the Giano simulator (34) and Modelsim. In this setup, the Verilog implementation of the eMIPS microprocessor and the CFPU are running in Modelsim. The board level devices such as memory and serial line etc. are simulated using Giano. Modelsim and Giano communicate using an interface defined the Modelsim Programming Language Interface (PLI). Giano using SoftFloat also generates instructions as well as the expected state of the microprocessor before and after execution of that instruction for testing.

At each step, the Giano-SoftFloat Oracle generates a new instructions and state information and sends all this information to the Verilog model of the FPU running in Modelsim. The Verilog model installs the FPU (register) state required by the test and executes the instruction. The resulting FPU state is compared to the expected one, and any discrepancy is reported as an error. This approach is superior in terms of measurable coverage, boundary cases coverage, unattended operation, orthogonal test generation, and error repeatability for debugging.

Table 3 Floating-Point Test Cases by SoftFloat

537bffbe	Floating-Point Operand 1
4e6c6b5c	Floating-Point Operand 2
000	Floating-Point Operations "000": Add "001": Subtract "010": Multiply "011": Divide "100": Square Root
00	Rounding Modes "00": Round to nearest even "01": Round up "10": Round down "11": Round toward zero
537c3ad9	Expected Floating-Point Result

If a bug is detected, we can use the debugging facilities available in Modelsim to observe the state of the design when the bug occurs and trace back to the cause. If necessary we can repeat these conditions many times to observe the design's behavior as we work out how to correct the bug until it is working as the specification dictates.

After the automated debugging runs a large breadth of random test cases and tests all boundary cases without error, we are satisfied of the behavioral correctness of the design and move to the hardware implementation phase. We compile for the target FPGA and generate a bitstream. As would be expected, some bugs manifested in the hardware that had not been encountered in our simulations. Most of these errors were time related. These were quickly corrected though analysis of the build report and on-chip debugging facilities provided by Xilinx Chipscope. We repeated the validation step in simulation after these modifications to ensure no new bugs were introduced by the modifications.

11. EXPERIMENTAL SETUP

The following section describes our experimental set up for evaluating the performance of the CFPU. We describe the experiments we performed using the LINPACK benchmark program, and the hardware platforms we used for implementing the eMIPS prototype extensible microprocessor.

11.1 eMIPS on the Berkeley Emulation Engine (BEE3)

We conducted our experiments on the Berkeley Emulation Engine, version 3 (BEE3) board, developed in cooperation between MSR and UC Berkeley (35). The BEE3 is a large hardware emulation platform for conducting research on computer architecture. It has four Xilinx Virtex 5 XC5VLX110T-2FF1136 FPGAs. Each FPGA has two channels of 8 GB DDR2, serial line and Ethernet. The FPGAs are linked in a ring configuration.

For the purposes of our experiments we used just one of the four available FPGAs to implement the eMIPS extensible microprocessor system. The eMIPS is a 32-bit processor and therefore can only address 4 GB of 16 GB available to the FPGA. For this reason the two DDR2 channels are truncated to 2 GB each.

The programs run on the system are compiled to software binaries which are downloaded to the system over the serial line. At reset, the system runs a bootloader program stored on-chip in the FPGA block ram (BRAM). The bootloader waits for data to be sent over the serial line. When it receives data, it copies it to the DDR2 memory. When all the program data has arrived in the DDR2, the processor jumps to the DDR2 and begins running the program. This can be done with any MIPS compatible program binary including OS binaries like NetBSD. For the soft float experiment, NetBSD is downloaded to the system. When the system has successfully booted up with NetBSD it uses a network file system over the Ethernet to load the LINPACK benchmark and run it. The hard float version is downloaded over the serial line as a standalone program.

11.2 Linpack Floating Point Benchmark

For our experiments we selected the LINPACK benchmark program to evaluate the floating point performance of our platform. LINPACK is a benchmark used to measure a processor's floating point performance. The benchmark attempts to solve a large N x N system of linear equations represented by:

$$Ax = b$$

where A is an N x N matrix of random floating point values and b is a 1 x N vector of random floating point values. The benchmark attempts to find the vector x using Gaussian elimination and partial pivots assuming the matrix A is not singular. If matrix A is singular the benchmark returns an error. The selection of random values for A makes this unlikely, but the programs checks for this case explicitly anyways. If the benchmark runs to completion, it will perform a number floating operations proportional to N (29) (30):

$$\frac{2}{3}N^3 + 2N^2$$

Even for relatively small matrices this is easily a considerable number of floating point operations, requiring considerable time on slow implementations. We compiled LINPACK using both soft floats (software emulation of floating point operations) using the math library and using hard floats (hardware supported floating point instructions). The soft floats required support of a math library which required us to run the soft floats with NetBSD for OS support. The hard floats version was able to run stand alone on the processor. Over the course of our experiments we varied the size of the matrix, N. from 10 to 1000. We recorded the time required by each version of our CFPU and the soft float program to solve the system of equations.

We chose to compare our performance with the software emulation because this is a common method to allow floating point operations on embedded processors that do not have floating point hardware besides fix point conversion. Fixed point conversion was not considered because it cannot maintain precision and would likely introduce errors running the LINPACK benchmark.

12. RESULTS

We conducted three experiments using the LINPACK Benchmark: 1) in a NetBSD OS environment, using the eMIPS architecture but with purely software emulation of floating-point instructions, 2) in a standalone eMIPS platform with the support

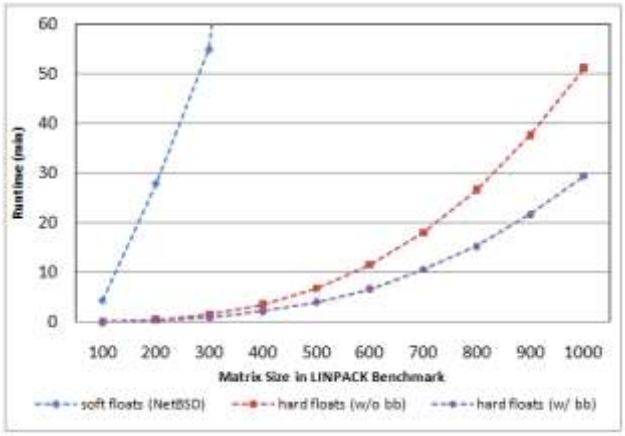

Figure 11 LINPACK Benchmark Execution Time Using Extensible CFPU: Time (min) versus Matrix Size

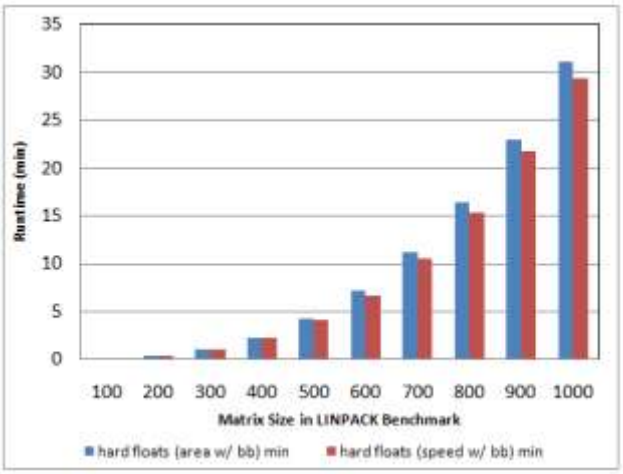

Figure 12 LINPACK Benchmark Execution Time Using Extensible CFPUs: Area Version versus Speed Version

of configurable hardware modules implementing the basic floating-point operations, and 3) same as #2 plus Extension instructions that implement entire basic blocks with floating-point support. The runtime data of this testing is shown in Figure 11, where the matrix size changed regularly from 100 to 1000. When the matrix size is 100, the LINPACK benchmark only needs a few seconds with hard floating-point modules, significantly less than the time needed by the pure soft execution on NetBSD environment (4 minutes). In the other end, when the matrix size is increased to 1000, the pure soft execution requires more than two days (not shown in the figure). With hard floating-point modules, the eMIPS platform is still able to finish the LINPACK program within 1 hour. The best speedup is a constant factor of about 80x.

Figure 11 also illustrates the execution performance when using Extension instructions (marked as "w/ bb" in the figure) against a regular CFPU (marked as "w/o bb" in the figure). It is shown that, with the optimization of even a single basic block, eMIPS already achieves a 1.7X speedup over a regular hard FPU. For instance, the 31.13 min for "hard floats (area w/ bb)" significantly outperforms the 51.20 min for "hard floats (area w/o bb)" when the matrix size is 1000. Considering only one extension was

Table 4 Synthesis Results of the Floating-Point Functional Units

	Slices	Slice FFs	LUTs	DSP48E
Fadd_speed	1091	1126	2013	1
Fadd_area	883	849	1655	1
Fmul_speed	1094	1069	1949	16
Fmul_area	843	776	1618	1
Ext_BB153	2596	2767	4737	2

configured in this study, we could expect that optimizing more basic blocks would create a larger speedup.

The hardware module implementing the basic block duplicates the floating point functional units into a separate data path activated by the extension instruction. This makes it easy to insert and remove the module for different experiments. The resource utilization of the basic block hardware is given in Table 4.

We also studied the performance variations when using different design strategies, such as the designs for area and design for speed. It is shown that (Figure 12) the final performance can be further augmented by utilizing various speed-driven optimization strategies to some extent. For instance, we are able to gain an extra 8% speedup with the speed optimization strategies and the execution time is reduced to 6.71 min comparing to the original 7.21 min need for a matrix size of 600.

We implemented the basic floating-point arithmetic units and extended basic block Extensions using Verilog HDL and we synthesized them with Xilinx ISE 10.1 targeting a Virtex-5 FPGA chip. Resource requirements of each implementation are given by Table 4. Given the addition and multiply units as an example, the additional performance gained by the speed optimized implementation cost approximately 25% more slices or logic resources. The speed implementation also uses 8 times more DSPs to builds its pipeline than the time multiplexing strategy used by the area implementation.

The resource utilization for the basic block hardware given in Table 4 includes the area implementation of the floating point addition and multiplication modules as well as logic for floating point load and store. Future implementations of the CFPU can eliminate most of this hardware cost by reusing functional units in the CFPU to support the standard floating point instructions and only add routing and control logic included in the basic block hardware's control finite state machine.

13. CONCLUSIONS

Motivated by the emerging demands and considerable resource requirements of floating-point applications in the embedded computing space, we introduced the notion of a custom floating-point unit (CFPU), where floating-point operations and any basic block containing floating-point related operations are implemented as a reconfigurable module and dynamically loaded, configured, and executed by software. Compliant with the conventional MIPS architecture and the IEEE-754 standard, the CFPU supports most of the floating-point instructions and operations, as well as the separate floating-point register file. Analysis of the SPEC CFP2000 benchmark suite indicates that floating-point add, subtract, and multiply operations are the most frequent ones. Consequently, we developed multiple implementations of these modules, using different optimization criteria such as area, speed, and latency.

This paper also presents an Extension design example with intensive floating-point operations that is able to replace a basic block of 25 instructions for extra performance and flexibility. Experimental results show that the reconfigurable processor platform with customized floating-point modules provide considerable performance benefits for floating-point computation intensive applications. The speedup over software execution is over 50x, and 80x when using Extension instructions.

We expect the benefits of the current custom floating-point reconfigurable modules to scale accordingly when implemented on the newer FPGAs, since the technology shrink will offer more capability and flexibility for conventional resource-consuming floating-point designs.

Future work includes extending the current work into the double-precision floating-point domain and exploring other ways to improve flexibility, such as an adaptive multi-mode floating-point unit or reconfiguring the computation components within floating-point modules to perform other functions. Furthermore, in our current design, only the classic floating-point arithmetic algorithms were implemented to achieve reasonable on-board resource consumption. In the future, we may exploit more aggressive optimization strategies with state-of-the-art floating-point arithmetic techniques.

14. REFERENCES

1. *Design and Implementation of a Modular and Portable IEEE 754 Compliant Floating-Point Unit.* **Karuri, K., Leupers, R. and Kedia, M.** 2006. Proceedings of Design, Automation and Test in Europe (DATE). pp. 1-6.

2. *Floating-Point FPGA: Architecture and Modeling.* **Ho, C. H., et al.** 2009, to appear in IEEE Transactions on VLSI Systems.

3. *High Performance Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4.* **Karlstrom, P., Ehliar, A. and Liu, D.** 2006. Proceedings of the 24th Norchip Conference. pp. 31-34.

4. *Implementation of Floating-Point Arithmetics using an FPGA.* **Sahin, S., et al.** s.l. : Springer Netherlands, 2007, Mathematical Methods in Eng., pp. 445-453.

5. *Design of an On-Line IEEE Floating-Point Addition Unit for FPGAs.* **Krueger, S. D. and Seidel, P. M.** 2004. Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines. pp. 239-246.

6. *FPGA Floating Point Datapath Compiler.* **Langhammer, M. and Vancourt, T.** 2009. Proceedings of IEEE Symposium of Field-Programmable Custom Computing Machines.

7. *Architectural Modifications to Enhance the Floating-Point Performance of FPGAs.* **Beauchamp, M. J., et al.** 2, 2008, IEEE Transactions on VLSI Systems, Vol. 16, pp. 177-187.

8. *Flexible Multi-Mode Embedded Floating-Point Unit for Field Programmable Gate Arrays.* **Chong, Y. J. and Parameswaran, S.** 2009. Proceedings of the ACM/SIGDA Symposium on Field Programmable Gate Arrays. pp. 171-180.

9. *Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines.* **Shirazi, N., Walters, A. and Athanas, P.** 1995. Proceeding of IEEE Symp. FPGAs for Custom Computing Machines. pp. 155-162.

10. *FPGAs vs. CPUs: Trends in Peak Floating-Point Performance.* **Underwood, K.** 2004. Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA). pp. 171-180.

11. *Single Pass, BLAST-Like, Approximate String Matching on FPGAs.* **Herbordt, M. C., et al.** 2006. Proc. of IEEE Symp. Field-Prog-rammable Custom Computing Machines. pp. 217-226.

12. **IEEE Std 754-1985.** IEEE Standard for Binary Floating-Point Arithmetic. s.l. : IEEE Computer Society, March 21, 1985.

13. **IEEE Std 754-2008.** IEEE Standard for Floating-Point Arithmetic. s.l. : IEEE Computer Society, August 29, 2008.

14. **Hennessy, J. L. and Patterson, D. A.** *Computer Architecture: A Quantitative Approach.* 4th Edition. San Francisco, CA : Morgan Kaufmann, 2006.

15. **Pittman, R. N., Lynch, N. L. and Forin, A.** *eMIPS, A Dynamically Extensible Processor.* MSR-TR-2006-143, Microsoft Research. Redmond, WA : s.n., Code available at http://research.microsoft.com/en-us/projects/emips. 2006.

16. *Hardware Compilation from Machine Code with M2V.* **Meier, K. and Forin, A.** 2008. Proceedings of IEEE Symp. Field-Prog-rammable Custom Computing Machines. pp. 293-295.

17. **Sekar, A. and Forin, A.** *Automatic Generation of Interrupt-Aware Hardware Accelerators with the M2V Compiler.* MSR-TR-2008-110, Microsoft Research. Redmond, WA : s.n., 2008.

18. **Kane, G. and Heinrich, J.** *MIPS RISC Architecture.* Upper Saddle River, NJ : Prentice Hall PTR, 1992.

19. *Division Algorithms and Implementations.* **Oberman, S. F. and Flynn, M. J.** 8, August 1997, IEEE Transactions on Computers, Vol. 46, pp. 833-854.

20. *A Generalized Multibit Recoding of Two's Complement Binary Numbers and Its Proof with Application in Multiplier Implementation.* **Sam, H. and Gupta, A.** 8, 1990, IEEE Trans. on Computers, Vol. 39, pp. 1006-1015.

21. *A signed binary multiplication technique.* **Booth, A. D.** June 1951, Quarterly Journal of Mechanics and Applied Mathematics, Vol. 4, pp. 236-240.

22. *A Suggestion for A Fast Multiplier.* **Wallace, C. S.** 1964, IEEE Trans. on Electronic Computers, Vols. EC-13, pp. 14-17.

23. **Ercegovac, M. D. and Lang, T.** *Digital Arithmetic.* San Francisco, CA : Morgan Kaufmann, 2004.

24. **Flynn, M. J. and Oberman, S. F.** *Advanced Computer Arithmetic Design.* Malden, MA : Wiley-Interscience, 2001.

25. **Overton, M. L.** *Numerical Computing with IEEE Floating Point Arithmetic.* Philadelphia, PA : Society for Industrial and Applied Mathematics (SIAM), 2001.

26. *Design Issues in Division and Other Floating-Point Operations.* **Oberman, S. F. and Flynn, M. J.** 2, February 1997, IEEE Transactions on Computers, Vol. 46, pp. 154-161.

27. **Xilinx Inc.** *Virtex-5 FPGA XtremeDSP Design Considerations User Guide, UG193(v3.3).* 2009.

28. —. *XtremeDSP for Virtex-4 FPGAs User Guide, UG073 (v2.7).* 2008.

29. *The LINPACK Benchmark: An Explanation.* **Dongarra, J.** 1988, Lecture Notes in Computer Science, Vol. 297, pp. 456-474.

30. *A Fine-grained Pipelined Implementation of the LINPACK Benchmark on FPGAs.* **Wu, G., et al.** 2009. Proceedings of IEEE Symp. Field-Programmable Custom Computing Machines.

31. *Industrial experience with test generation languages for processor verification.* **Behm, M., et al.** 2004. Proceedings of ACM/IEEE Design Automation Conference. pp. 36-40.

32. *AVPGEN - A test generator for architecture verification.* **Chandra, A., et al.** 1995, IEEE Transactions on VLSI Systems, Vol. 3, pp. 188-200.

33. **Hauser, J.** *SoftFloat.* Available: http://www.jhauser.us/arithmetic/SoftFloat.html. 2002.

34. **Forin, A., Neekzad, B. and Lynch, N. L.** *Giano: The Two-Headed System Simulator.* MSR-TR-2006-130, Microsoft Research. Redmond, WA : s.n., 2006.

35. **Davis, J. D., Thacker, C. P. and Cheng, C.** *BEE3: Revitalizing Computer Architecture Research.* MSR-TR-2009-45, Microsoft Research. Redmond, WA : s.n., 2009.