UNIVERSITY OF CALIFORNIA
SANTA CRUZ

FLOATING-POINT NETWORK NODE FOR MULTI-UNIVERSITY RESEARCH NETWORK

A thesis submitted in partial satisfaction of the
requirements for the degree of

Master of Science

in

COMPUTER ENGINEERING

by

Pranav Kumar Natesh

September 2012

Table of Contents

List of Figures

List of Tables

Abstract

Floating-point Network Node for Multi-University Research Network

by

Pranav Kumar Natesh

Floating-point operations are complex, they are both power and area intensive. The performance of floating-point operations can have a significant impact on the overall performance of a processor. This thesis explores the design, verification and testing of a synchronous pipelined out-of-order IEEE-754 compliant floating-point unit (FPU) for the Santa Cruz Out of Order RISC Engine (SCOORE).

SCOORE is a superscalar out-of-order SparcV8 processor and is currently under development in the micro-architecture Santa Cruz (MASC) Laboratory. The processor is planned to be taped out in the 28nm GLOBALFOUNDRIES process as a part of the Multi University Research Network. Our FPU provides hardware support for floating-point addition, subtraction, multiplication, division, square-root arithmetic operations for single as well as double precision. The design is modeled using the Verilog-2001 HDL and implemented using the STM 90nm process.

The three major design units described in this work are a traditional Flop-based pipelined FPU; a latch-based pipelined FPU utilizing the *retry* mechanism; and the FPU ring network node consisting of the FPU, the programmable built-in self test (BIST) and the demultiplexer (DMUX). The design meets the target operating frequency goal of 1.4 GHz on the TSMC 90nm process.

Acknowledgments

I would like to thank my advisor, Professor Jose Renau, for his invaluable insight, advice and patience to help make this work a reality. I would also like to thank all the members of the MASC and SCOORE group.

Most importantly, I would like to thank my parents, Natesh Mani and Gayathri Natesh, and my sister, Aishwarya. I will forever be thankful for your endless suport and encouragement.

Chapter 1

Introduction

1.1 Motivation

Fixed-point numbers are used extensively in computers, but their limited range make them inadequate for many scientific and engineering applications.

Floating-point numbers are of great importance, because they give us the ability to represent a large range of numbers using very few bits. Floating-point numbers are especially useful to the scientific community. Extremely large quantities such as the inter-planetary distances or very tiny quantities such as the size of an electron are impractical to represent in a fixed-point format, but are feasible in the floating-point format. To put it in more realistic terms, 50 bits can represent inter-planetary distances with an error of a few microns [1].

Floating-point arithmetic is being used for 3-D graphics and automotive control, among other applications. Modern mobile processors are also fitted with FPUs for their performance benefits.

1.2 SCOORE

SCOORE, the Santa Cruz Out-Of-Order RISC Engine, is a high-performance out-of-order SPARCV8 processor currently under implementation by the MASC research lab at UCSC. SCOORE has several novel aspects; it is an implementation of the SPARCV8 Instruction set Architecture (ISA). It is 4-issue superscalar and has a 12-stage pipeline. In comparison with current Intel and AMD processors, SCOORE has a larger issue logic, re-order buffer (ROB), and register file size.

The computational heavy-lifting is done by the compute engine. The compute engine consists of four architectural pieces, which perform different tasks. The A-unit is responsible for the arithmetic operations. The B-unit takes care of the branching operations. The C-unit also called the complex unit, handles the floating-point instructions, and lastly the M-unit deals with load/store operations. The FPU, the focus of this thesis, lies in the C-unit.

SCOORE supports two types of ISAs, namely the SPARCV8 [2] and ARM, and internally these instructions are converted to SCOORE micro-operations (μOPS). Thus the instructions are decoded to SCOORE μOPs before the FPU operates on them.

ASIC TARGET

SCOORE utilizes a shadow process technology during the design process, namely the 90nm STM and the 45nm TSMC. The 28nm GLOBALFOUNDARIES (GF) process being our final tapeout technology. The 90nm ASIC target clock speed is 1.4 GHz and the estimated 45nm ASIC target clock speed is 3.0 GHz with Synopsys Design Compiler using a 20% clock uncertainty and typical libraries. Our final 28nm tapeout target frequency is estimated between 2.5 GHz and 3.0 GHz. The tool flow is shown in figure 1.1. The tools DC shell and RC Cadence are used for design synthesis, SOC Cadence is used for all back-end activities such as floorplanning, place and route, power and clock distribution. Lastly DRC mentor is being used for design rule checking.

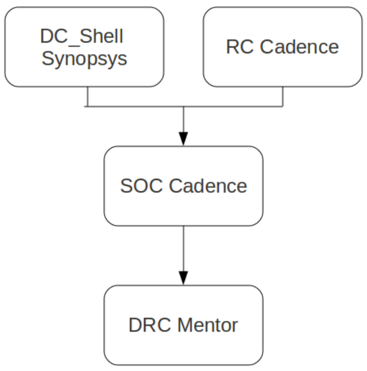


Figure 1.1: ASIC Design Flow

1.3 MURN

Multi-University Research Network (MURN) is an academic project lead by MASC group at University of California, Santa Cruz. MURN is a collaborative effort between GF, UCSC and UCSD. The goal of the project is to tapeout a heterogeneous chiplet on the GF 28nm Super Low-Power (SLP) process.

There are two tapeouts planned for the project. An early tapeout with certain blocks on a ring-network and a final tapeout of the finished design. The rationale behind the early tapeout is to work out all the issues in the backend flow of the design. Early tapeouts are also known to have a positive impact on the overall design time of a project [3]. The high-level view of the first tapeout is shown in figure 1.2.

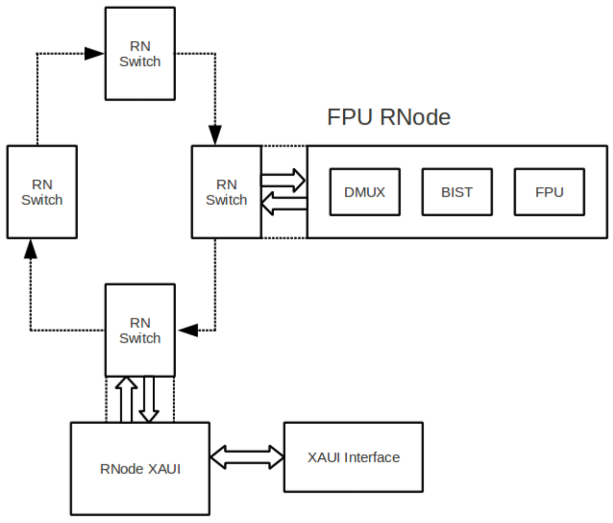


Figure 1.2: Architectural view of the ring network.

Each design block is self-contained within a network node. All the nodes in the chip are inter-connected in a ring topology. This ring also contains a communication network node, which provides a high-speed interface to the ring-network to the mother-board on which the chip is sitting. The protocol used is XAUI, which is an extension of the XGMII (10 Gigabit Ethernet protocol). Alongside the MURN chiplet, a daughter FPGA is also present on the mother board.

The daughter FPGA is used for chip bring-up as well as running experiments on the chip.

Among various blocks, the FPU is going to be a part of the first tapeout. The FPU along with a programmable testing unit is given the name floating point network node or *FPNode*.

1.4 Floating-point Numbers

Many applications including scientific and DSP require inputs that are rational in nature i.e. numbers that have fractional components. One method is to reserve certain bits for the whole part of the number and other bits for the fractional part. The disadvantage of this method is that we have a limited range of the numbers that can be represented. Based on how the bits are reserved, we sacrifice either the precision of the integer part or the fractional part. In real-world applications, the requirement on the precision of the fractional component also varies, for example, in miles we do not care about the fourth decimal value when representing distance between cities, but this value becomes important to us when we talk about the share price of a stock.

The answer to this problem is the use of floating-point numbers, where a large range of numbers can be represented with varying amounts of precision. Higher precision for smaller number which is usually the desired behaviour.

The FPU presented in this work is compliant with the IEEE 754-1985 standard [4], which is the most widely used standard for all floating-point computations. The last major update to the standard was in 2008.

1.5 IEEE 754-1985 standard

The standard defines a method to represent numbers in the floating-point format, which includes special values like zeros, denormalized numbers, infinities and NaNs (Not a Number), and it defines operations to be performed on floating point operands. Lastly, it also sets rules on how to handle special scenarios such as divide-by-zero conditions or dealing with irrational numbers. The goal of a unified standard is to give programmers the power to create more robust and predictable systems [5].

Representation

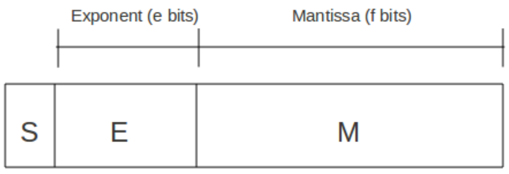


Figure 1.3: IEEE floating-point bit field representation.

As shown in figure 1.3, a floating-point number conforming with the IEEE standard is made up of a sign bit *S*, an exponent field *E* and a mantissa field *M*. The value of the floating-point number is given by the

$$(-1)^S * M * 2^{E-Bias} \tag{1.1}$$

The number of bits assigned for the exponent and mantissa fields are based on the precision level. There are four formats defined by IEEE 754-1985 namely single precision, double precision, single-extended precision and double-extended precision. Our FPU supports both single and double precision floating-point operations as shown in figure 1.4.

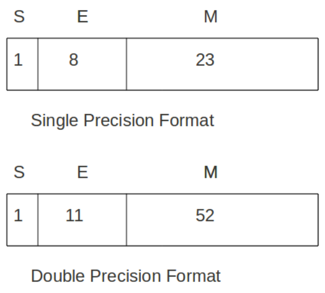


Figure 1.4: IEEE Floating-point Formats.

The sign bit *S* represents whether the number is positive or negative, a value of 0 represents a positive number. The exponent value *E* is stored in a biased format, this implies that the value stored is offset from the actual value by a constant known as the bias. The value of the bias is dependent on the number of bits in the exponent field. The mantissa field is used to represent the fractional component of the number. If the exponent field is neither zero nor the largest representable exponent the value is considered to be *normalized.* A normalized number puts the radix point after the first non-zero digit. In a base 2 representation the only possible non-zero digit is 1. Thus we can assume a leading digit of 1 and not store it explicitly. There are cases when the numbers are extremely small and where the most significant bit is 0. This is a *denormalized* representation.

IEEE-754 also allows the representation of certain special values. Not-a-Number is used for a result of an illegal operation, for example imaginary numbers like $\sqrt{-1}$. The values +∞ and -∞ are used to represent values that are out-of-range. In both of these scenarios the exponent field is set to all one's. The mantissa is set to zero for infinite values and to non-zero for NaNs.

Chapter 2

FPU Design and Implementation

This chapter describes the design and implementation of the FPU of the SCOORE processor. The initial section of this chapter provides an overall architectural view as well as various performance metrics. Sections 2.2 2.3 2.4 2.5 include explanations of the individual floating-point operation pipelines, providing design details as well as their implemented algorithms. Finally, we present the implementation details of the design.

2.1 Overview

The FPU is completely pipelined with each operation having its own pipeline and operating on double-precision floating-point numbers. All the operations share some common stages like the denormalization, normalization, rounding and result queueing. All operation pipelines can take in one instruction every cycle with the exception of the divide and of the square root units that have a multi-cycle iterative operation.

Figure 2.1 shows a top-level architectural view of the FPU. It depicts internal structures as well as the unit's interface with the *Complex Unit*. The design has three major phases starting with the de-normalization phase followed by the execute phase, and finally the post-normalize and rounding phase. Most of the operations have different pipelines due to the difference in their operation algorithms, but they still share certain stages such as the de-normalization stage and the rounding stage. Table 2.1 describes the input and output interfaces of the design.

We now describe the major stages in the floating-point unit.

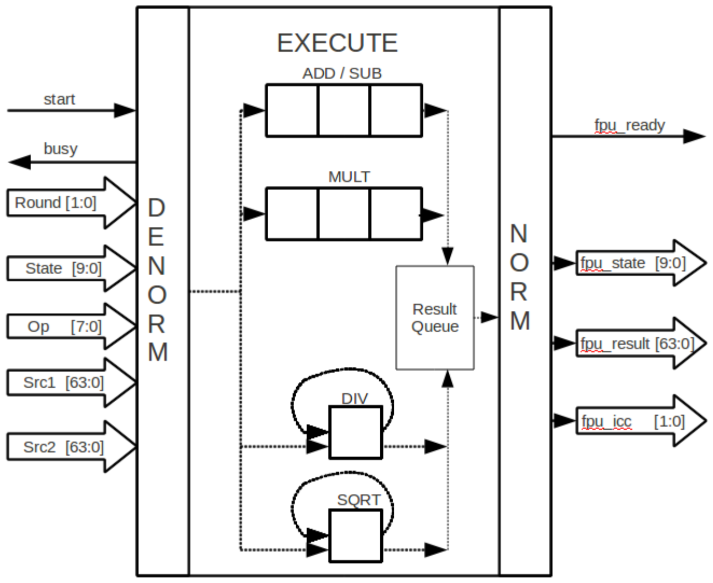


Figure 2.1: FPU top level architectural view

2.1.1 De-normalization Phase

The de-normalization phase is responsible for unpacking the input operands. Certain hardware logic required prior to the execute phase is also performed in the de-normalization phase such as mantissa swapping for the addition/subtraction operation. It has a latency of one clock cycle. Only one operation is issued in-order for execution each cycle. Another block which operates in parallel with the denorm phase is the Operation Pre-decode block that pre-decodes the opcode of the current instruction and converts it to a one-hot bit vector. This bit vector is used by the following stages, thus optimizing on the decoding logic in the pipeline.

Port	Type	Width	Description
clk, reset	I, I	1,1	
start	I	1	This is to indicates a valid instruction at the FPU input.
round	I	2	Rounding mode of the current instruction. 0 - Round to nearest even; 1 - Round to zero; 2 - Round to positive infinity; 3 - Round to negative infinity.
op	I	6	Used for the OPCODE of the instruction.
state, fpu_state	I, O	10, 10	Every FPU instruction has an associated 10-bit destination address. This address is passed along the data-path as is.
src1, src2	I, I	64, 64	These are used to provide two double precision sources for the instruction. The lower 32-bit word is used for single precision numbers. Only SRC2 is used for single input operations.
fpu_result	O	64	Data result of the operation.
fpu_ready	O	1	Indicates the validity of the result data.
fpu_icc	O	3	Integer or Floating-point Condition Codes depending on the OPCODE of the result.
busy	O	1	Indicates that the FPU is not ready to accept any new instructions. Only applicable in the flop-based FPU.

Table 2.1: FPU Interface

2.1.2 Execute Phase

The execute phase performs the out-of-order execution of the various floating-point operations. The addition and multiply operations are designed in a pipelined fashion and have a latency of 3 clock cycles each. Divide and square root utilize iterative algorithms for their execution, and use a multi-cycle data-path; they have a latency of 64 clock cycles. The completed instructions pass through a *Result Queue* to avoid contention and to ensure only one instruction is finished per cycle.

2.1.3 Post-Normalization

The result generated by the execution phase may or may not be normalized. The post-normalization block ensures the result is normalized by shifting the mantissa and adjusting exponent appropriately. This functionality is provided prior to being sent to the result queue.

2.1.4 Rounding

The FPU supports four rounding modes, i.e. round to nearest, round to 0, round to $+\infty$ and round to $-\infty$. The value of the RD field of the floating-point state register (FSR) for each instruction is sent to the FPU [2].

2.2 Floating-point Addition

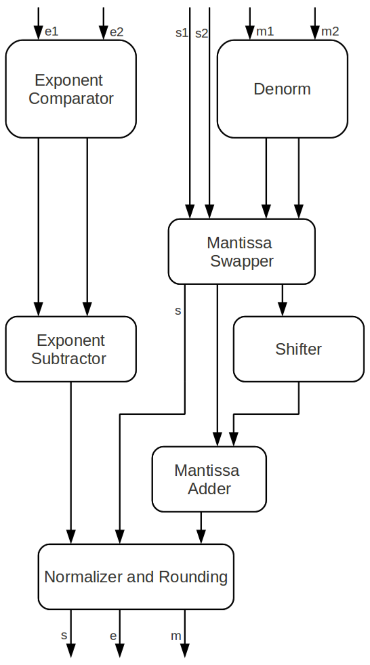


Figure 2.2: Floating-point addition pipeline

Floating point addition is a simple algorithmic process that needs to be completed in sequential order [1]. The steps involved in this process are illustrated in figure 2.2. The subtraction operation also utilizes the same hardware except that the sign bit is inverted at the denormalize phase.

Denormalization

The operands are initially unpacked to get the mantissa, exponent and sign bits. The operation may be an addition or a subtraction based on the value of the sign bit as well as the opcode of the instruction.

Mantissa Swapper

The addition algorithm works by shifting the smaller mantissa by the difference of the two exponent values. Thus, the operands may need to be swapped before sending them to the shifter depending on their magnitudes.

Shifter

The exponent value of the smaller operand is adjusted to match the value of the larger operand, this is done by shifting the mantissa bits appropriately.

Mantissa Adder

This is a simple process where the mantissa values are added together to generate the mantissa of the result.

Normalizer and Rounding

Since both operands are aligned to the larger operand's exponent value, the result needs to be normalized back to have the largest mantissa bit as 1. This is done by detecting the number of zeros starting from the *msb* and left-shifting the operand and adjusting the exponent value. The result also needs to be rounded based on the rounding mode set for that instruction.

The SCOORE addition implementation is a five-stage pipeline i.e. has a latency of 5 cycles, where the first stage is utilized for operand unpacking, exponent comparison and mantissa swapping. The next two stages are used for mantissa shifting as well as detecting which mantissa is greater, and essentially determines how the sign bit *S* is set. The next stage is used for the addition operation. The last stage is used for normalization and rounding.

2.3 Floating-point Multiplication

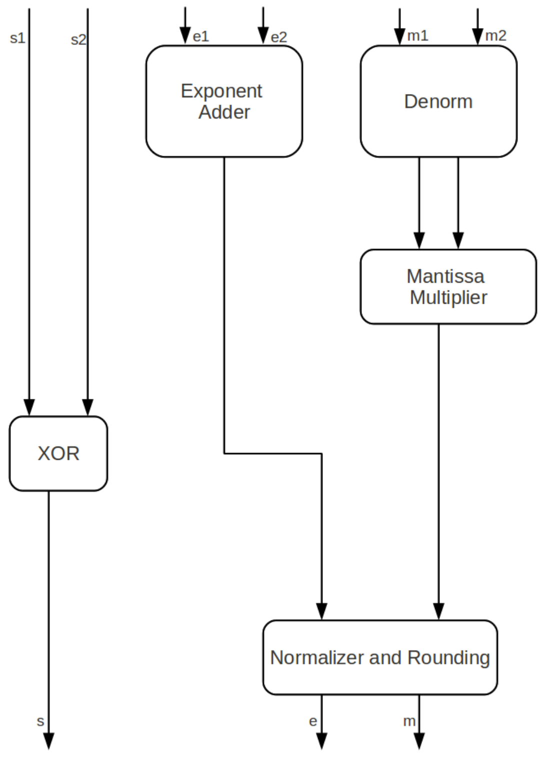


Figure 2.3: Floating-point Multiplication Pipeline

The mantissa multiplier is the most complex component of the multiply pipeline. Since mantissa multiplier functions similarly to an integer multiplier, it is used in the FPU of SCOORE to perform signed and unsigned integer multiplications. The first step in the multiply pipeline is similar to that of addition, we unpack the operands into exponent, mantissa and sign fields. Much like all FP pipelines, FP multiplication consists of two major data paths, the exponent and the mantissa fields. To generate the resultant exponent, we need to perform addition on the biased exponents, ensuring that the bias is subtracted from the result. The sign bit of the result is obtained by an XOR of the two sign fields of the input operands. The remaining and the most logic intensive step is to perform the mantissa multiplication. The multiplication of two 52-bit operands yields a result of 104 bits. Internally, both single and double precision are treated the same, for single precision number we truncate appropriately after obtaining the result.

The basic steps involved in the mantissa multiplication are partial product generation and partial product addition [6]. The floating-point multiplier of SCOORE has a latency of three cycles for execution and normalization of the result. In the first stage, the partial products are generated and in the next two stages, the partial products are added.

After obtaining the result, it might be necessary to normalize the result by shifting the mantissa and adjusting the exponent accordingly. The NaN detection and propagation logic is similar to that of the division unit, thus we have common logic that is being used by both the blocks. We observe that NaN detection and propagation can be completed in one cycle but extra flops are added to maintain synchronization with the mantissa multiplier pipeline. All the multiplication operations are completed in-order.

2.4 Floating-point Division

To perform a floating-point divide operation, a logical XOR operation is performed on the sign fields; the exponent fields are subtracted and similar to the floating-point multiply, the complexity remains in the division of the mantissa fields, which is done in a similar fashion to an integer division.

There are two major classes of division algorithms, *Digit Recurrence* which applies a subtractive recursive approach and *Functional Iteration* which applies a multiplicative approximation method [7]. The subtractive method selected for SCOORE FPU, provides for a simpler implementation and produces a constant number of bits per iteration [8].

There are four identical dividers used within the FPU, each taking 64 cycles to perform a divide operation. This gives us the ability to handle four in-flight division operations. A divider selector unit selects, which divider to use, and selects each divider in a sequential order. The FPU will generate a busy signal if all the dividers are busy. Each divider has a multi-cycle data path used to evaluate the mantissa values. A ready signal will be generated when the computation is finished. In the regular flop-based implementation, a FIFO was used to buffer the results. As multiple units can finish at the same cycle a result selector is used to make sure there are no losses due to contention.

Parallel to the four mantissa dividers, we also have combination logic to detect the propagates NaNs in the design. The NaN detection logic evaluates in one cycle, but contrary to the multiply pipeline, instructions can retire out-of-order.

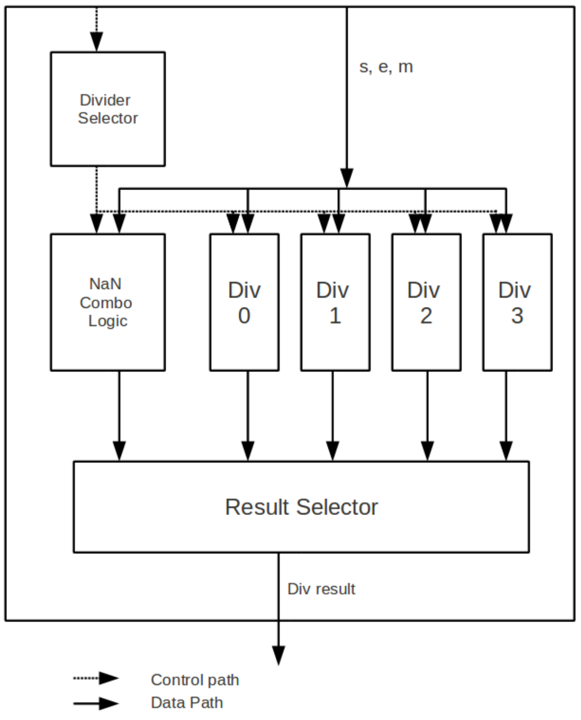


Figure 2.4: Floating-point division unit

2.5 Floating-point Square Root

There are two major classes of algorithms for mantissa square root calculation, *Digit Recurrence* and *Multiplicative*, depending on the basic step-by-step method used [8]. In the digit recurrence method, the convergence is linear and one cycle is needed to terminate the step. On the other hand, in the multiplicative, the convergence is quadratic and each iteration requires floating-point multiplications, and the rounding process is more complex. The multiplicative method requires changes to the multiplier which can make it a slower process [1].

In our design we have implemented the recurrence method which takes 64 cycles to perform the operation for double precision numbers and 32 cycles for single precision. Square root being a unary operation only the SRC2 is needed.

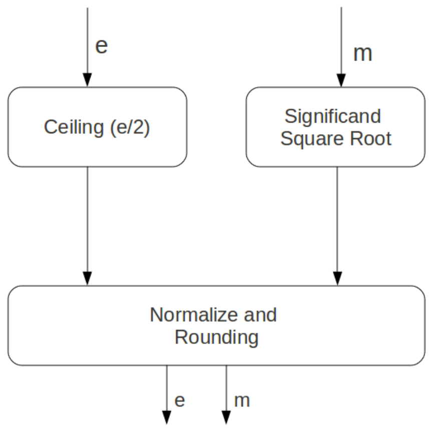


Figure 2.5: Floating-point Square Root Architecture

2.6 Non-arithmetic Operations

The floating-point unit also supports compare operations. These operations are treated internally as floating-point subtract operations and relevant condition-codes are generated based on the result. As per IEEE-754, the four condition codes generated are zero, less than, greater than and unordered (when one or more of the operands is a NaN).

2.7 Rounding

Arithmetic operations on floating-point numbers lead to results that cannot be represented in the given amount of precision, thus the results needs to be rounded.

2.7.1 Round Toward Zero

This mode of rounding is achieved by truncating the result to the required amount of precision, this has an effect of getting the value closer to zero. For example, the number 0.6677 is truncated to 0.667, if three decimal places are available; and 0.66, if two decimal places are available.

2.7.2 Round Towards $+\infty$

The result is rounded to $+\infty$ regardless of the value to the right of the number of available digits. For example, the result 1.345 is rounded to 1.35 and similarly -1.345 is rounded to -1.34 for three digits of precision.

2.7.3 Round Towards $-\infty$

This method of rounding is similar to the round to $+\infty$ except that now we round to $-\infty$. For example, the result 1.345 is rounded to 1.34 and similarly -1.345 is rounded to -1.35 for three digits of precision.

2.7.4 Round To Nearest Even

This is the mostly commonly used rounding mode, this specifies that for scenarios where the number can be rounded in two legitimate ways, pick the closest even number. For example 1.5 rounds to 2.0 but 4.5 rounds to 4.0 for two digits of precision.

2.8 Latch Based Design

In this implementation of the FPU all the flip-flops in the design were replaced by specialized latch-based structures. These structures called stages are currently under research in the MASC laboratory. Even though they do have a bigger area impact they provide the means to ask a sequential element to hold state. Stages work on a retry protocol and can be thought of as a FIFO of size 1. The underlying hand-shaking mechanism is shown in figure 2.6. Each data input has a corresponding valid signal. When an output data is issued with an invalid signal, the data is discarded and it is neither processed nor stored. Each module has two retries associated, one retry signal is received from next stage in the pipeline and retry which is an output to the previous stage. The retry from the next stage implies that current stage needs to hold its state until the retry is de-asserted. A retry output to the previous stage implies that the current stage cannot accept any new data and requests that the data be sent again. The retry stage contains a handshaking system based on the valid signal that simply controls the contention in the communication line.

The following figure 2.7 shows the basic interface of the retry stages. The internal structure of this unit is beyond the scope of this thesis. The timing diagram 2.8 shows the operation of the stage unit. In the first scenario 2.8(a) there are no retries applied to the stage, therefore it acts like a regular flip-flop, and we see the data being propagated to the output as expected. In the next scenario 2.8(b) the *retry_in* input to the stage is asserted for two cycles, concurrently there is also a data input to the stage, this data is buffered since the next stage is not ready to accept the data. The stage asserts *retry_out* during the next cycle, at this point the input to the stage must be held until the *retry_out* is de-asserted. After the *retry_in* is removed we see the buffered data on the stage output.

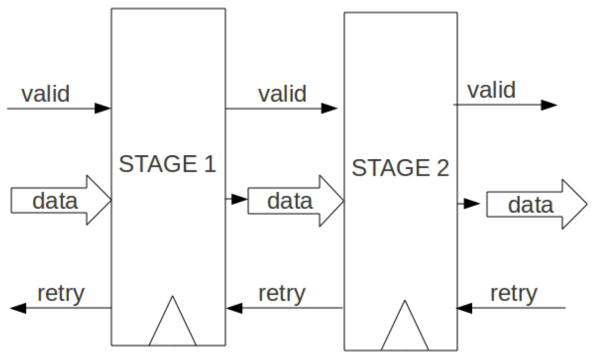


Figure 2.6: Retry handshaking

The replacement of the flip-flops leads to a simpler architecture as well as reduction in certain buffer logic like FIFOs, because an entire pipeline can be put in ”retry” state without the worry of losing data. The retry protocol also simplifies parallel paths we can now have parallel pipelines of different lengths without the need to balance them. Certain control logic

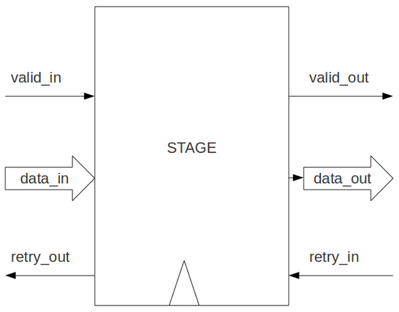


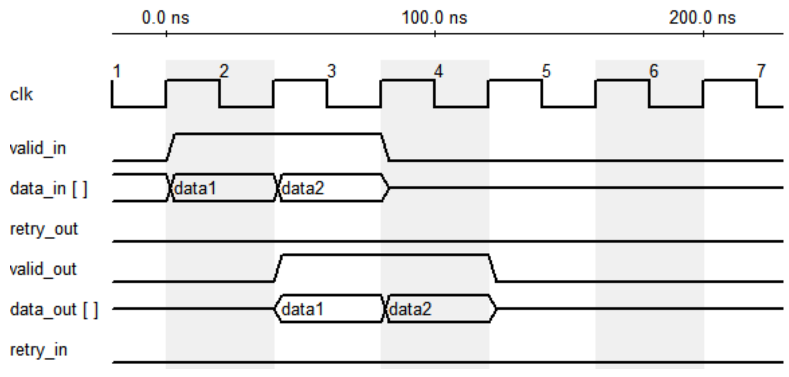
Figure 2.7: Stage interface

also needed to be re-vamped since all the data-paths in the design did not have a standardized valid signal, which is needed by the stages. The process of conversion of the design was found to be non-trivial, and the method adopted was to convert the design one operation at a time. Special points of interests were the de-normalize and result-queue phases. In the denormalize phase, a large flop was used to feed data to the different operation pipelines, in this new scheme this flop was replaced by individual retry stages. All the operation pipelines feed the result-queue, there can be a scenario where more than one operation pipeline is ready with the result on the same cycle, to avoid contention a priority scheme was added to the retry logic. The highest priority given to the FP addition pipeline, followed by the multiplication, FP square root, and lastly division.

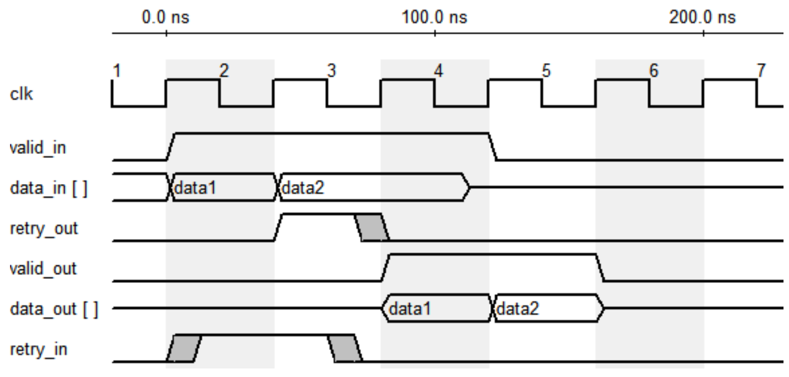
2.9 Performance Evaluation

The table 2.2 reports the IPC results of multi-cycle FP operations. Fully pipelined operations such as floating-point addition (F_ADD), floating-point subtraction (F_SUB), floating-point multiplication (F_MUL), and integer multiplication (I_MUL) achieve an IPC of 1 as expected.

To gauge the performance of the FPU under realistic application scenarios, we wanted to study the performance of the FPU under different SPECfp [9] benchmarks. The SESC [10] simulator was used to study four benchmarks to profile the floating-point operations performed.



(a) Stage Operation Without Retries



(b) Stage Operation With Retries

Figure 2.8: Stage Timing Diagram

The benchmarks swim, applu, wupwise and mgrid were used. Using these numbers we generate instruction mixes for the FPU and supply them as stimuli to the FPU. The operands and the sequence of operations used for these evaluations were randomized and about 1 million instructions per test-suite was used to gather results. All the evaluations assume that the test-bench does not assert any retries. The performance results are shown in table 2.2.

2.10 Implementation Results

The floating-point unit was synthesized using the Synopsys DC compiler using the CORE90GPLVT library under normal operating conditions. Table 2.3 details the synthesis results of the FPU. The design was constrained for a frequency of 1.4 Ghz.

Operation / Benchmark	IPC Achieved
FP64_DIV	0.076
FP32_DIV	0.066
FP64_SQRT	0.032
FP32_SQRT	0.016
I32_DIV	0.062
SWIM Benchmark	0.913
APPLU Benchmark	0.905
WUPWISE Benchmark	0.976
MGRID Benchmark	0.999

Table 2.2: Performance evaluation

Parameter	Synthesis Results
Time period (ns)	0.71
Maximum Frequency(GHz)	1.41
Non-combinational Area (μm^2)	31.75
Combinational Area (μm^2)	36057.57
Total Area (μm^2)	36089.32

Table 2.3: Synthesis results

Using the 90nm TSMC libraries the leakage power of the FPU was estimated at 2.33 mW and average dynamic power consumption was estimated at 7.37 μW. The major critical paths in the design were found to be the partial product generation in the multiplication unit and square-root unit. The multiplier was brought out of the critical path by adding another pipeline stage to the unit. The timing of the square root unit was improved by moving some the computation out of the multi-cycle data-path. The design will finally be synthesized using the 28nm GLOBALFOUDRIES libraries. The exact timing, area and power numbers will be obtained at that time.

Chapter 3

FPU Ring Node

3.1 Testing Unit

The floating-point unit is going to be a part of the first MURN tapeout. Functional testing of the unit could not have been achieved via the network interface. This is due to the fact that the target frequency of the final SCOORE design is going to be close to 1.4 Ghz, going off-chip for every test vector would be far too slow to ensure the design is functional at-speed. To overcome this issue a testing unit was designed which runs functional tests independent of the network interface. The purpose of this unit is to buffer the test vectors as programmed by the off-chip device and run the FPU test autonomously.

The main elements of this unit are a DMUX which decodes packets from the ring network, and a unit which can store and drive the test independently called the programmable testing unit or PBIST.

3.1.1 DMUX

This unit directly talks to the network switch. All data exchanged between the network node and network ring is done via the DMUX. Each packet received at the DMUX interface contains an opcode field as well as a data field (see figure 3.1). These opcodes are decoded and passed to the PBIST unit as control signals. Once the test results are ready, the DMUX will ship the packets on the network.

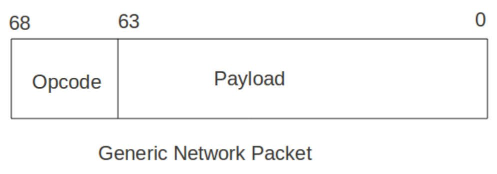


Figure 3.1: Network Packet Description

3.1.2 PBIST

The PBIST buffers the test vectors and once instructed by the operation network packet, performs the test sequence while maintaining all the control signals as expected by the design under test (DUT), for example when the FPU asserts a busy signal, the testing unit must hold its inputs to the FPU. The test sequence can contain any of the operations supported by the FPU. The unit acts like the transactor and the scoreboard for the test being performed. The PBIST in essence resembles a memory built-in self test or BIST, which performs march algorithms to detect manufacturing defects in memories and can be programmed externally. The difference lies in the fact that this testing unit is used to check for functional correctness. The figure 3.2 shows the inputs and outputs of the block.

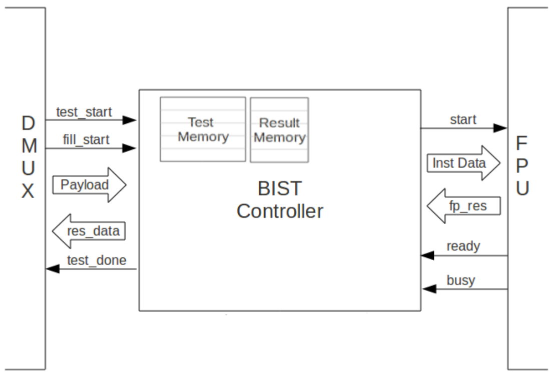


Figure 3.2: Programmable BIST

The PBIST uses SRAMs to buffer the input test vectors as well as the outputs generated by the DUT. The testing unit consists of seven banks of 32 x 128 single port SRAMs used as test and result memory, all the memories have a one cycle read/write latency. Five of the banks are used to store the input tests and the two banks are used to buffer the results generated by the FPU.

The PBIST has an internal state machine used drive the test. If the network nOde is not busy and the test data has been consumed by the FPU, the memory pointers will be updated to the next instruction, thus all the instructions are driven sequentially until all the test vectors have been passed to FPU. At any point the FPU can assert a retry signal to the testing unit at which point the state machine waits until the de-assertion of the retry signal and only then will the next instruction be passed to the FPU. The retry protocol requires the testing unit to hold the input valid signal until the data is consumed. The FPU asserts a ready signal once it has the result of an operation ready. Since the testing unit is always ready to accept these results it never asserts a retry to the FPU output stage. Only one instruction is retired by the FPU per cycle, but results can finish out-of-order. To store the result in-order, the state information of the FPU result is used to generate the index for the memory write operation. Therefore the result memory is kept in the order instruction issue.

Currently the SRAM depth has been limited to 128, thus as many instructions are given to the FPU, if a smaller test is needed, NOPs will need to be added at the end of the instruction sequence. A configuration register is set by the network which determines the number of iterations to be performed. The configured number in this register instructs the testing unit to repeat the testing sequence for n iterations. After the first iteration every consecutive iteration results is compared against the first iteration and an error signal will be raised if there is any inconsistency.

One interesting testing feature that can be implemented is to randomly assert retries at the FPU output to check for possible retry related issues in the design. The high-level design of the testing is reusable. In order to adapt the testing unit, the DMUX logic can be used as is, but the PBIST logic and the memories used within the design will need to be modified based on the requirements of the DUT block, for example the A-unit of the SCOORE design can be tested in a similar manner.

3.2 RNode

Every design node of the MURN ring network is called an RNode (as seen in figure 1.2). The designer of the RNode instantiates a ring switch which is shared by all RNode blocks. This switch provides an interface to the ring. Every network packet has an address bit field, which dictates which ring switch is the destination of the packet. The floating-point ring node consists of the switch, DMUX, PBIST and the FPU.

Figure 1.2 details how various RNode connect to the MURN network. Table 3.1 describes the various operations available on the testing unit. The following section walks through the step-by-step data-flow in the testing unit.

Opcode	Payload	Description
FILL	Discarded	This opcode is sent before the test vectors are given to the programmable BIST.
STOP_FILL	Discarded	This opcode is sent after all the test vectors are sent.
TEST	RAM index + data	This opcode tells the testing unit that the payload contains a test vector
ITER	Iteration count (2-1023)	This is used to set the iteration count for the test, the default is 2
GO	Discarded	Once all the SRAMS are programmed, this command indicates to the testing unit that the test can start. The results are automatically sent out once the test is finished.

Table 3.1: FP RNode Opcodes

3.2.1 Testing Unit Flow

The steps involved in programming the testing unit and performing the tests on the floating-point is detailed in the following section.

- Send a network packet with the FILL opcode.
- 32-bit test vectors are sent in sequential order with the opcode TEST.
  - Payload[31-0] Represents actual test data

    - Payload[41-32] Provides the row number for the buffer memory.
    - Payload[47-31] Provides the bank number for the buffer memory.
        * Bank0 $\rightarrow$ SRC1 lo
        * Bank1 $\rightarrow$ SRC1 hi
        * Bank2 $\rightarrow$ SRC2 lo
        * Bank3 $\rightarrow$ SRC2 hi
        * Bank4[5:0] $\rightarrow$ fp_op
        * Bank4[7:6] $\rightarrow$ rounding

- Once all the test vectors are completely initialized, a network packet with the STOP_FILL opcode is given.
- A network packet with a GO opcode instructs the testing unit to initiate the test.
- Once the testing is complete, the unit will automatically send out the test results sequentially.
- The testing unit is designed to send out 32 bits of data in the payload of the network packet at a time. Each FPU instruction generates a 64-bit result, thus each output is broken down to two network packets.
- The first packet received refers to the lower 32-bits of the first FPU instruction.
- The second packet received refers to the upper 32 bits of the first FP instruction and similarly for the rest of the instructions. Thus for a 128-bit test set, there will be 256 network packets seen at the network node output.

3.2.2 BIST Network Fields

The network node talks to ring network on two occasions, firstly when the test vectors are being programmed and secondly when the test is complete and the packetized result data is being sent out. The Figure 3.3 illustrates the two packets types. In the first scenario, starting from the highest order field, the network opcode field is programmed with the FILL opcode. The next two bit fields are programmed with the bank and instruction number respectively, the bank number can be thought of as the index of the packets that make up one FPU instruction.

The bank number ranges from zero to four, for the FPU network node. The lowest order word is meant for the actual test data.

In the second scenario, the lower 32-bit field is used to hold the result data. Each FPU instruction result is packetized into two network packets of 32-bits. The last representation in figure 3.3 shows the construction of one FPU instruction using five network packets.

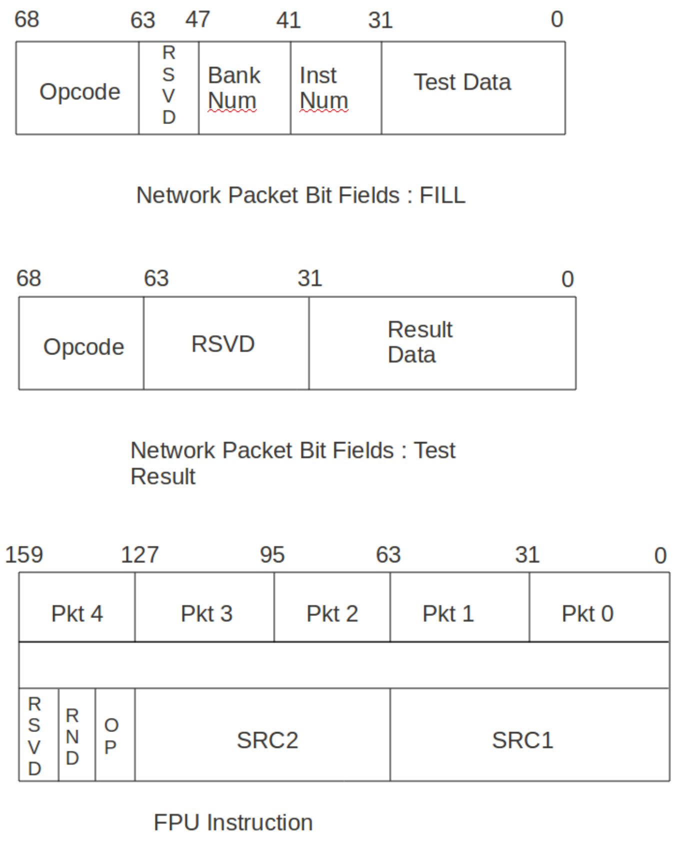


Figure 3.3: FPU Network Node Packet Bit Fields.

Chapter 4

Pre-Silicon Verification

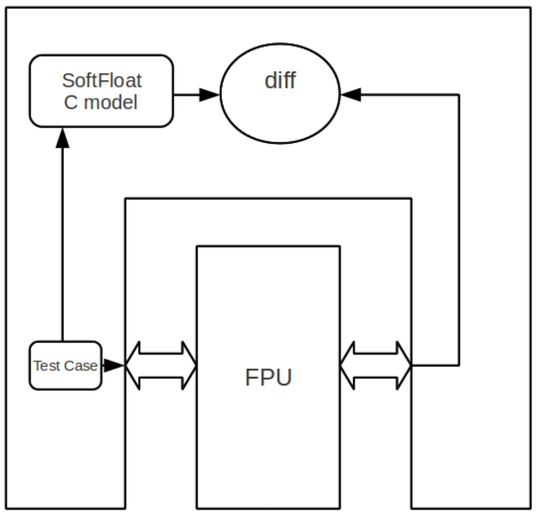


Figure 4.1: Testing Harness for the Floating-point Unit.

4.1 Verificaton of the Floating-point Unit

The validation of floating-point units is known to be a complex process. The infamous floating-point divide bug cost Intel millions of dollars as it was detected only after it had been released to the customers [11]. Intel currently conducts formal methods to verify its FPUs [12].

All SCOORE design blocks use a uniform testing scheme. All the test scenarios are written in C/C++ and the interface with the RTL code with a programming language interface (PLI) function. At every clock cycle we assert the DUT inputs using a set function and the DUT outputs are monitored using a check function.

The *SoftFloat* [13] test suite provided by John Hauser is used as the stimulus for the FPU. *TestFloat*, also part of the SoftFloat suite, provides a range of randomized inputs vectors, giving special attention to the corner cases used to check for IEEE compliance. A test scenario is built using these vectors and randomized operations. The test vectors are used drive the DUT as well as the C model, which computes the expected results. These expected results are stored in data structures in the testbench with the state ID used as a index. As we observe the outputs generated by the FPU, the state ID of the results is used to fetch the pre-computed results, which are then compared. A testbench error is generated if they are not equal.

The testing mechanism is shown in figure 4.1. The major challenges in testing of the floating-point is to ensure the algorithms were well defined in verilog as well the design RTL compliance with the IEE-754 standard. Table 4.1 lists all the tests that have been implemented for the FPU.

In our testbench we have included checks to ensure all the instructions have been been retired only once and within an allowable range of delay based on the operation and the retries applied on the design. Even though the default test performs randomized operations, certain switches can be provided to test for certain operations. We also have the ability to instrument operation mixes which is used for our performance evaluation. Our scheme has been successful in catching much of the design implementation and compliance.

The second implementation of the design uses latch-based stages instead of flip-flops. The only change in the testing infrastructure was to introduce a retry stage like interface to the DUT inputs and outputs. The legacy busy signal is replaced by out_retry. To ensure the DUT is tested under all retry conditions. The output or normalize stage of the FPU is stressed to a maximum of eleven continuous and randomized retires. The floating-point design is also

Test Name	Description
float64_add	Double Precision Floating-point Addition Test
float64_sub	Double Precision Floating-point Subtraction Test
float64_mul	Double Precision Floating-point Multiplication Test
float64_div	Double Precision Floating-point Division Test
float64_sqrt	Double Precision Floating-point Square Root Test
float32_add	Single Precision Floating-point Addition Test
float32_sub	Single Precision Floating-point Subtraction Test
float32_mul	Single Precision Floating-point Multiplication Test
float32_div	Single Precision Floating-point Division Test
float32_sqrt	Single Precision Floating-point Square Root Test
umul	32-bit Unsigned Integer Multiplication Test
smul	32-bit Signed Integer Multiplication Test
udiv	32-bit Unsigned Integer Divide Test
sdiv	32-bit Signed Integer Divide Test
random	The default test, random mix of all floating-point and integer operations that can be performed on the FPU

Table 4.1: List of implemented tests

stressed by forcing all the pipeline stages using retry to assert the retry to the previous stage randomly via the stage cell. This helped uncovering a lot of bugs in the retry logic and is found to be a very effective method to test designs using retry as an interface.

4.2 Verification of the Floating-point Network Node

The block-level testing of the FPU network node is done at the network switch interface. The testing is done in a step-wise fashion as described in the section 3.2.1. Firstly, the FPU network node is programmed with test vectors and then instructed to conduct the test independently using the GO opcode given to the DMUX. Once the GO opcode is given the testing unit does not communicate with the network interface. The test cannot be halted until the test has finished. After the test completes the result vectors are received at the network switch interface in the form of network packets, which will then be compared against pre-computed values.

Chapter 5

Conclusions

A floating-point network node is designed for the first tapeout of the MURN project. The design is synthesized using the TSMC90 process and achieves an operating frequency of 1.4 GHz. Using the 28nm GLOBALFOUNDRIES flow, we expect the frequency to be around the 2.3 GHz mark. The floating-point unit was verified for functional and IEEE [4] compliance using the software model made available by John Hauser [13].

The floating-point unit standalone has an area estimate of 36057.57 μm^2. The testing unit adds an area overhead of 4865.09 μm^2, which does not affect the operating frequency of the design unit.

Appendix A

Running Tests

The SCOORE project uses a unified Ruby Make infrastructure for running all tests. The following are examples of running tests on the command line for the SCOORE FPU.

```
rake test:fpu_tb
rake test:fpu_tb args="+test=float64_add"
rake test:fpu_tb args="+test=float32_mul +debug=on"
```

Figure A.1: FPU testing

The following command line executes the test for the FPU RNode.

```
rake test:rnode_fpu_tb
```

Figure A.2: FPU RNode testing

Bibliography

[1] M. Ercegovac and T. Lang. Digital arithmetic. 2003.

[2] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[3] C. Ma. DFM - an industry paradigm shift. 30-oct. 2, 2003.

[4] IEEE Standards Board. IEEE standards for binary floating-point arithmetic. Technical Report ANSI/IEEE Std. 754-1985, Institute of Electrical and Electronics Engineers, 1985.

[5] C. Severance. An interview with the old man of floating-point.

[6] R. Yu and G. Zyner. 167 MHz radix-4 floating point multiplier. Washington, DC, USA, 1995. IEEE Computer Society.

[7] S.F. Obermann and M.J. Flynn. Division algorithms and implementations. *Computers, IEEE Transactions on*, (8), aug 1997.

[8] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Comput. Surv.*, September 1996.

[9] The Standard Performance Evaluation Corporation. http://www.specbench.org, Dec 1996.

[10] J. Renau, F. Basilio, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[11] Vaughan P. Anatomy of the Pentium bug. Springer-Verlag, 1995.

[12] B. Bentley. Validating the Intel(r) Pentium(r) 4 microprocessor. 2001.

[13] J. Hauser. The softfloat and testfloat packages.