Integrating a floating point unit into the AT&T Hobbit™ microprocessor

Paul T. Holler
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
Holler, Paul T., "Integrating a floating point unit into the AT&T Hobbit™ microprocessor" (1993). Theses and Dissertations. Paper 214.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.
AUTHOR: Holler, Paul T.

TITLE: Integrating a Floating Point Unit Into the AT&T Hobbit TM Microprocessor

DATE: October 10, 1993
Integrating a Floating Point Unit
into the

AT&T Hobbit™ Microprocessor

by

Paul T. Holler

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Electrical Engineering
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

9/28/93
Date

Thesis advisor

EECS Department Chairperson
Table of Contents

Abstract ................................................................................................................................. 1

1. Introduction ..................................................................................................................... 2
   1.1. Hobbit History ........................................................................................................... 2
   1.2. Organization of thesis .............................................................................................. 3

2. Advanced Microprocessor Architecture Techniques .................................................... 4
   2.1. RISC vs. CISC ........................................................................................................... 5
   2.2. Pipelines ................................................................................................................... 8
   2.3. Pipeline Hazards ...................................................................................................... 10
       2.3.1. Structural Hazards .......................................................................................... 11
       2.3.2. Data Hazards .................................................................................................. 11
       2.3.3. Control Hazards ............................................................................................ 13
           2.3.3.1. Conditional Branches ............................................................................. 13
           2.3.3.2. Interrupts ............................................................................................... 14
   2.4. Advanced Architectures ........................................................................................... 15
   2.5. Survey of Superscalar Processors ............................................................................ 18
       2.5.1. Pentium ............................................................................................................ 18
           2.5.1.1. Pipeline Details ....................................................................................... 19
           2.5.1.2. Speculative Execution - Precise Interrupts ............................................ 20
       2.5.2. Metaflow Lightning ......................................................................................... 20
           2.5.2.1. Pipeline Details ....................................................................................... 21
           2.5.2.2. Speculative Execution ............................................................................. 25
           2.5.2.3. Precise Interrupts .................................................................................... 26
       2.5.3. Motorola 88110 ............................................................................................... 26
           2.5.3.1. Pipeline Details ....................................................................................... 27
### 3. Floating Point Units

#### 3.1. Data Types

#### 3.2. Required Operations

#### 3.3. Instruction Execution

#### 3.4. Performance

#### 3.5. Exceptions

### 4. Current Hobbit Architecture

#### 4.1. Instruction Set Architecture

##### 4.1.1. Instruction Categories

##### 4.1.2. Variable Length Instructions

##### 4.1.3. Accumulator

##### 4.1.4. Addressing Modes

##### 4.1.5. Flow Control

##### 4.1.6. Stack Cache (Register Allocation)

##### 4.1.7. Subroutine Calls

#### 4.2. Implementation

##### 4.2.1. I/O

##### 4.2.2. Prefetch Buffer

##### 4.2.3. Prefetch/Decode Unit

##### 4.2.4. Decoded Instruction Cache (DINC)

##### 4.2.5. Execution Unit

##### 4.2.6. Stack Cache (SC)

#### 4.3. Execution Unit Details
4.3.1. IR Stage .......................................................... 50
4.3.2. OR Stage .......................................................... 52
4.3.3. RR Stage .......................................................... 53

5. Proposed Modifications ..................................................... 55

5.1. Architecture of Floating Point Unit ....................................... 55
  5.1.1. Data Types ....................................................... 56
  5.1.2. Instruction Set Extensions ....................................... 56
  5.1.3. Organization of FPU ............................................. 58
  5.1.4. Accumulators .................................................... 61
  5.1.5. Memory Requirements for Integration of FPU ...................... 62

5.2. Integration .................................................................... 63
  5.2.1. Prefetch Decode Unit ............................................. 64
  5.2.2. Decoded Instruction Cache ....................................... 65
    5.2.2.1. Issue .......................................................... 65
  5.2.3. Execution Unit Modifications ..................................... 67
    5.2.3.1. Address Generation/Operand Fetch (IR) ............... 68
    5.2.3.2. Execution (OR) .............................................. 70
    5.2.3.3. Retiring Instructions (RR) ............................... 71
  5.2.4. Speculative Execution ........................................... 72
    5.2.4.1. Exceptions - Changes in Flow ........................... 72

5.3. Supporting Memory Organization - Stack Cache Modifications .... 73

6. Conclusions and Future Work .............................................. 75

6.1. Effect of the Proposed Modifications ................................... 75
  6.1.1. Area ............................................................... 75
  6.1.2. Power .............................................................. 76
  6.1.3. Performance ...................................................... 76
6.1.4. Accuracy ........................................................................ 77

6.2. Summary ............................................................................ 78

7. References ............................................................................. 81

Appendix A ............................................................................... 83

Vita .......................................................................................... 85
List of Tables

Table 1: Characteristics of RISC and CISC processors ........................................ 6
Table 2: Instruction Set Comparison Intel 486 vs. SPARC .................................... 7
Table 3: Parameters for IEEE floating point ......................................................... 32
Table 4: Floating Point Unit Performance ............................................................. 35
Table 5: Hobbit Instruction Set .............................................................................. 38
Table 6: Hobbit Addressing Modes ...................................................................... 39
Table 7: Call Procedure ......................................................................................... 42
Table 8: Floating Point Instructions (both double and single precision) ............ 57
Table 9: Performance of Proposed Floating Point Unit ......................................... 59
Table 10: Accumulator Mapping on Stack ............................................................. 62
Table 11: Pipeline sequence for C=(*A++ *B++)+C ........................................... 80
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Pipeline with a Single Functional Unit</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Pipeline with Multiple Functional Units</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Metaflow Architecture</td>
<td>21</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Data Structure for the DRIS</td>
<td>22</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Pipeline for the Motorola 88110</td>
<td>27</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Hobbit Block Diagram</td>
<td>44</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Execution Unit Pipeline</td>
<td>50</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Execution Unit Implementation</td>
<td>51</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Possible Floating Point Unit Organization</td>
<td>60</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Issue - Intelligent FIFO to Functional Blocks</td>
<td>66</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Destination File</td>
<td>71</td>
</tr>
</tbody>
</table>
Abstract

The Hobbit™ microprocessor is a high performance, low power, low cost microprocessor. The intended market is personal communicators where communications, handwriting recognition and general purpose computing are all important. High performance, high integration, low power and low cost all contribute to the success of personal communicators. The Hobbit microprocessor has a single integer processing unit which is optimized for the execution of C language programs. Integration of more processing units in the processor should increase performance with a corresponding increase in area, cost and power.

The topic of this thesis is the addition of a floating point execution unit to the Hobbit processor. The requirements of such an execution unit and a method for integrating it is presented. All modifications proposed keep the basic architecture of Hobbit processor intact. Applying previous techniques is not straightforward because of the unique architecture and philosophies of the Hobbit processor.

This paper describes hazards that arise because of the pipelined nature of processors. It also discusses several techniques available for improving processor performance including both superpipelined and superscalar architectures. Several proposed and commercial processors are analyzed to gain insight on how to modify the Hobbit processor.
Chapter 1: Introduction

AT&T's latest entry into the microprocessor field is the Hobbit™ Microprocessor. This processor is currently targeted for low power-high performance applications. The processor performs only integer operations with software emulation of floating point operations. The topic of this paper is to propose a new implementation with a floating point unit (FPU) fully integrated into the processor. Architectural organizations and implementations that will enhance performance will be suggested and analyzed. Modern and historical techniques will be studied in an effort to determine how to meld the FPU while achieving the best possible results.

The current metric that is used to evaluate the performance of Hobbit is MIPS (Millions of Instruction per Second) per Watt. This metric is useful for determining the relative performance for low power systems. In line with this benchmark, any changes made to Hobbit must contribute to throughput without substantially decreasing the MIPS per Watt criterion. Therefore, exotic or performance-at-all-costs solutions will not be allowed. In addition, any proposed changes should allow backward compatibility and not require binary recompilation to run existing code.

1.1 Hobbit History

The Hobbit Microprocessor has its origin in the CRISP architecture [3,4,5,6] designed at AT&T Bell Labs in the 1980's. CRISP, which stands for "C Rationalized Instruction Set Machine", arose out of work started in the mid 1970's to design and build a processor optimized toward the fast execution of C language programs. The first CMOS version of the design began in 1983 with silicon available in 1986. Since then a number of modifications have been made to the architecture yielding the present day Hobbit microprocessor. The
architecture of *Hobbit* was not constrained to be a CISC (Complex Instruction Set Computer) or a RISC (Reduced Instruction Set Computer) processor. Features that led to high performance of C programs were included. Those that were not effective or were detrimental were not included. Like RISC, the instruction set was kept small so as to ease implementation of the pipeline. This allows high peak processing rates. Like CISC, the instructions were encoded with variable lengths to increase code density. Unlike current high performance processors, *Hobbit* was designed to work well with no external cache, but yet use only a single piece of silicon. Register allocation was designed to be transparent easing the burden on the compiler design and reducing call procedure overhead. Finally, the architecture is scalable without resorting to new instructions when the number of registers is increased.

### 1.2 Organization of thesis

This paper focuses on applying advanced architectural techniques which can be used to add a hypothetical floating point unit to the *Hobbit* processor. Chapter 2 discusses such techniques in the context of historical and contemporary processors. It explains the options available in improving the performance of processors. Background information pertaining to CISC and RISC processors is also contained in Chapter 2.

Chapter 3 covers the implementation of a floating point unit. Topics include data types, instructions and exception handling along with a survey of performance for several processors. The current architecture of the *Hobbit* microprocessor is analyzed in the next chapter.

Proposed modifications to the architecture is discussed in Chapter 5, including a possible implementation of a floating point unit and a method of integrating it into the *Hobbit* processor. Chapter 6 contains conclusions and suggestions for future work.
Chapter 2: Advanced Microprocessor Architecture Techniques

Microprocessors are generally classified into two broad categories: RISC (Reduced Instruction Set Computers) and CISC (Complex Instruction Set Computers). Although most research, development and new design has focused on RISC processors (PowerPC, MIPS, SPARC, ALPHA, to name a few), CISC processors have maintained their ground in terms of performance (Pentium, MC68000s, among others). The underlying nature of the microprocessor architecture can ease or compound the ability to seamlessly add functional units or the ability to improve performance without simply relying on processing advances. It is for this reason that a digression into the RISC and CISC strategies is needed.

It is also useful to review some basic tenets of processor design. One such fundamental is the pipeline. Along with performance benefits from pipeline execution come potential pitfalls. Many microprocessors are including more functional units onto the chip to supplement the traditional single integer ALU operations. As additional functional units are integrated more hazards arise.

The basic architecture of the processor decides which techniques can be employed when adding functional units. However, an overview of modern architectures, such as superpipeline, superscalar and VLIW give insight into techniques that may be useful.

While the design details of the floating point are not going to be discussed in this paper, it is critical to know some of the fundamentals since timing and I/O are important in integrating such a unit into the processor. Therefore, a section on floating point units is included.
A brief discussion of current microprocessors, their architecture and the techniques employed to attain high performance while avoiding hazards is presented in the last section.

2.1 RISC vs. CISC

CISC processors have evolved from the simple processors of the early 70's. Physical limitations on die size, transistor geometry and complexity kept the hardware simple. Instruction sets were very limited and a good compiler was needed to transform the high level source code down to efficient machine code. As the physical constraints on processor design eased and the hardware implementations became more efficient, the ability to execute more complex instructions grew. This made the task of translating source code to machine code much simpler. This proliferation of instructions, data formats and addressing modes exact a toll on the processor. To keep from exploding the size of a compiled program and to conserve memory bandwidth, instructions had to be variable in length. Simple instructions with no arguments can be as small as 1 byte, whereas instructions with full 32 bit operands might require more than 3 words. To make matters more complex, the portion of the instruction containing the opcode (operation code) and addressing modes can be from 1 to several bytes.

Of course the instruction set influences the organization greatly. The complex instructions mean that the execution unit must internally execute small programs, called microcode, which break the instruction down into smaller pieces. Architects of RISC processors realize that all of this flexibility comes at a price. The designs are much more complex and did not necessarily perform much better than "simple" solutions which put the burden on the compiler.
System performance is typically determined by:

\[ t_{\text{exec}} = t_{\text{cycle}} \cdot N \cdot CPI \]

where \( t_{\text{exec}} \) is the total execution time of the program, \( t_{\text{cycle}} \) is the clock period of the processor, \( N \) is the number of instructions to be executed and \( CPI \) (cycles per instruction) is the average number of cycles needed to execute an instruction.

CISC implementations try to minimize the number of instructions\((N)\). But since the processors are more complex, the number of cycles per instruction\((CPI)\) is usually higher. RISC architectures tend to improve \( CPI \) at the expense of more instructions. Although a few RISC processors run faster than CISC, CISC is very close in clock rate to the current RISC machines. Table 1 summarizes the characteristics of the “typical” RISC and CISC machines.

<table>
<thead>
<tr>
<th>Field</th>
<th>RISC</th>
<th>CISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Instructions</td>
<td>under 100</td>
<td>over 200</td>
</tr>
<tr>
<td>Number of Addressing modes</td>
<td>1-2</td>
<td>5-20</td>
</tr>
<tr>
<td>Instruction Length</td>
<td>Fixed</td>
<td>variable</td>
</tr>
<tr>
<td>Instruction Formats</td>
<td>1-2</td>
<td>3+</td>
</tr>
<tr>
<td>CPI</td>
<td>near 1</td>
<td>3-10</td>
</tr>
<tr>
<td>Memory access</td>
<td>load/store only</td>
<td>most CPU operations</td>
</tr>
<tr>
<td>Registers</td>
<td>32+</td>
<td>2-16</td>
</tr>
<tr>
<td>Control Unit</td>
<td>Hardwired</td>
<td>Microcode</td>
</tr>
</tbody>
</table>

The number of instructions, addressing modes, instruction length and formats directly affects the ease and efficiency of fetching and decoding instructions. RISC processors with their limited number of instructions and formats make it easier to fetch the next
instruction. CISC processors need to determine the length of the current instruction before adding it to the current program counter (PC) to get the address of the next instruction. The program counter is a register which contains the address of the current instruction. Table 2 shows the extent of the difference between the Intel 486 and SPARC instruction set. The impact on modern architecture techniques will be seen later.

The method of memory access also differs between the two classes. RISC processors have a large number of registers inside the processor and operate on the load/store model. Data must be explicitly loaded into the processor before it can be manipulated or moved. Writing data back to main memory must also be done explicitly via a store instruction. This simplifies the internal architecture of the processor allowing CPI to be high. A problem arises in this case if simple moves are used often. When this happens rather than executing a single instruction, MOVE, the processor must execute a load and then a store. The expectation is that this would not happen very often.

<table>
<thead>
<tr>
<th>Instruction Set Category</th>
<th>Intel 486</th>
<th>SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Transfer (Load-store)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flag Manipulation</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Logical, Shift, Rotate</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>String Manipulation</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Bit Manipulation</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Conditional &amp; Unconditional Transfer (Branch)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Loop</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Interrupts</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Table 2: Instruction Set Comparison Intel 486 vs. SPARC (Continued)

<table>
<thead>
<tr>
<th>Instruction Set Category</th>
<th>Intel 486</th>
<th>SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLL (high level lang)</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Protection Support</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Processor Control</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

2.2 Pipelines

All modern day microprocessors, the *Hobbit* processor included, break the execution of each instruction into several phases. This process, known as pipelining, can vary from a few stages to many stages.[8,9] The idea is to overlap instruction execution, so that the processor can be executing portions of several instructions concurrently. This greatly improves the performance of the processor.

Figure 1 Pipeline with a Single Functional Unit

A typical method of portioning the work is shown in Figure 1. Here an instruction is fetched in the Instruction Fetch (IF) stage and then handed to the Instruction Decode/Operand Fetch (ID) stage. After the instruction is accepted by the ID stage, the IF stage is free to fetch another instruction. The procedure continues in a similar fashion with the ID stage feeding the Execute stage (EX) which in turn feeds the Memory Access (MA) stage which in turn transfers the result to the write back (WB) stage.

Limitations in performance can arise when one stage takes longer to complete than the others or when a pass through a functional unit takes multiple cycles. Most stages can
operate equally well for all types of instructions. For instance, the IF and ID stages don't become much more complicated when handling different classes of instructions. However, the functional unit which does the computation in the execute stage is generally efficient at doing one class of operations. Historically, processors performed only integer operations in the microprocessor and floating point operations were done in a coprocessor. This is because it was not effective to implement both integer and floating point operations in one functional unit.

**Figure 2 Pipeline with Multiple Functional Units**

The solution is to use multiple functional units as shown in Figure 1. These are common place on most high performance microprocessors. The functional units can be uniform or nonuniform. Uniform processors duplicate entire functional units so that the functional units are identical. More often the architects use nonuniform approaches where the functional units are different and each solves a certain task. For example, integer operations may execute in one functional unit and floating point in another. It is not unusual, however, to find multiple integer units within a processor. The integer units can be symmetric (identical) or asymmetric with one including more functionality than the other.

The functional unit is single stage of the instruction pipeline. However, this does not mean that operations must execute in a single cycle. Integer ALUs in RISC machines complete most operations in one cycle. However, floating point units generally take more
than one cycle. The delay from inputting an instruction until the result is available is called the latency. Having a latency greater than one cycle does not constrain the ability to insert new instructions while the previous instructions are still under execution. The floating point units in high performance processors are themselves usually pipelined so that new instructions can enter the execute stage before previous operations have completed. The rate at which new instructions can be entered is called the issue rate.

2.3 Pipeline Hazards

There are 3 classes of hazards that occur in pipelined machines: structural, data and control hazards. These hazards will decrease the performance of the processor if they cannot be avoided by causing the pipeline to stall while waiting for the hazard to clear. All of these can be addressed through careful architectural considerations.

Structural conflicts arise in processors if functional units are not fully pipelined such that a new instruction cannot be issued to the execution unit every cycle. They also occur in processors which can simultaneously issue two instructions if both instructions need to execute simultaneously on the same functional unit and there is only one unit available. The solutions are to fully pipeline every execution unit, or to provide a queue for each unit, so that a new instruction can be sent every cycle and to provide the correct mix of functional unit.

Data conflicts arise due to the nature of the instruction stream and the pipeline. Read After Write (RAW) hazards occur when a subsequent instruction tries to read a register or memory location before a prior instruction updated it. Write After Read (WAR) hazards happen when a subsequent instruction writes a register before a prior instruction had the ability to read it. Write After Write (WAW) hazards happen when a subsequent instruction writes to a register before a prior instruction has the ability to write to it. The last two hazards are typical results of not issuing and completing instructions in program order.
Control hazards arise from interrupts and branches. Due to the delay through the pipeline both of these are difficult to manage efficiently. These problems are exacerbated if instructions are executed or complete out-of-order.

Much of the current work in advanced microprocessors involves adapting the work done in the early 1960's by CDC and IBM.

2.3.1 Structural Hazards

Structural hazards are easy to detect. If the particular unit or units that can execute the next instruction to be issued are all busy, a hazard has arisen. This can cause the pipeline to stall unless instructions are allowed to execute (issue) out-of-order. With out-of-order execution, the instruction that is stalled because of the structural hazard can be bypassed by a subsequent instruction that can find an available functional unit. This requires a window of decoded instructions which are available for execution. Control hazards are then compounded because the issues do not fetch operands in strict program order. This also creates more data hazards. However, performance can be greatly improved by keeping the functional units running.

2.3.2 Data Hazards

Hazards can be avoided by software or detected by hardware. As an example, early designs by MIPS, Inc. [22, 23] did not provide for hardware hazard detection. The compiler was responsible for filling load delay slots or inserting instructions that perform no operation (nops) so that no hazards occurred. In most microprocessors, hardware does the data hazard detection. Detection of RAW hazards is straightforward on machines with a single nonpipelined functional unit. A simple comparison of the destination address (register or memory) and the operand address will determine the presence of a hazard. The solu-
tions are also straightforward. One solution, not used often is to simply interlock the processor so that the read is delayed until the result that is needed is written back to memory. Another option is to do forwarding or bypassing where the output of the execute or write-back stage can be fed to the operand fetch stage.

In processors with only one functional unit which is fully pipelined with a latency more than one cycle, the detection is still relatively easy, but the operand fetch stage must be delayed until the execution is completed. The WAR and WAW hazards can not arise in single functional unit processors if all of the updates to memory and registers occur only in stages after the execute stage.

On processors with multiple functional units or architectures where write backs to registers or cache can occur before a subsequent instruction can access the memory location, the situation is much more difficult. The RAW hazards can still be detected and solved in a method similar to the earlier scheme, but more hardware will be required. It is possible for instructions to complete out-of-order on processors with multiple functional units if the units have different latencies. The results of this is that data can be written back to memory in the wrong order. This can cause a write after write (WAW) hazard if both instructions have the same destination. In addition, if the instruction which was issued first caused an exception, or was interrupted while an instruction that issued later has already completed, the state of the machine would be invalid. An anti-dependency, Write After Read (WAR), hazard may also occur with out-of-order execution. A write might occur to a memory location or register, before a previous read is executed. In this case the read gets new data rather than old data.

All three data hazards were addressed and solutions were implemented in mainframes in the early 60s. The CDC 6600 originated the idea of a central scoreboard to detect and either redirect operands or stall the issue of instructions.[17] Tomasulo’s work on the IBM 360/91 originated the concept of decentralized control with multiple reservation sta-
tions per functional unit for the IBM 360/91 and register renaming[18]. Many of the newer processors on the market have adapted techniques from both the CDC and IBM research.

2.3.3 Control Hazards

Control hazards in a pipelined processor can be divided into two categories: conditional branches and interrupts. Both are greatly complicated by the use of out-of-order completion and execution. Machine states must be retained for both classes.

2.3.3.1 Conditional Branches

There are several ways of handling branches. In the simplest cases, compilers can schedule useful instructions in the slot following the branch. But this is difficult when the pipelines become deep and may be inefficient on a multiple issue machine. Compilers would need to find several instructions to execute in the delay slot. If the branch was the result of a load, the pipeline might need to stall quite a while.

Another method is to predict the direction of the branch early in the pipeline. This can be as simple as a hint from the compiler contained in the instruction which indicates which way the branch is likely to go. Branch history tables can be used to base the prediction on past history. Then the predicted path is loaded in the pipeline so that when the decision is made, the more likely instructions are waiting in the pipeline. This is referred to as prefetching instructions.

A more likely path to follow is speculative execution. If we know the likely path the branch will take we can keep following it. Since instructions are executing and completing out-of-order, it may very well happen that we are many instructions down the path before the branch can be evaluated. Depending on the mechanism the results or the executions may or may not be written back to the register file. If not this is called speculative issue. If they are written back this is called speculative execution.
This, of course, further complicates the hardware. If the prediction was wrong, then the instructions must be somehow reversed and the register file restored to its original condition. A write buffer can be used to retire or commit instructions after the result of the branch is known.

Speculative execution can greatly increase performance if the prediction is reliable and the recovery from misprediction is reasonable.

2.3.3.2 Interrupts

Interrupts can be sent synchronously or asynchronously to the processor. In the asynchronous case, the exact instruction which was executing when the interrupt occurred is of little importance. However with synchronous interrupts, an instruction may be responsible for causing the interrupt. In this case, the ideal situation is to have the machine restored to the state it was in just prior to the interrupting instruction.

There are two methods of handling interrupt: precise and imprecise. In an imprecise method, the exact state of the machine can not be stored. The best that could be achieved was to reset the processor to some state which would allow the processor to restart. This was state of the art in the early 60's. For example the CDC 6600 completed all issued instructions, stored them and then copied a restart state in to the processor.

With the advent of virtual memory, the IEEE floating point standard (exceptions) and other techniques, being able to reset machine to the state before the offending instruction became very important. This is called a precise interrupt. A complication in out-of-order completion/execution architectures is that the hardware must ensure that all instructions before the offending one complete before the interrupt is serviced.

In addition restoring the state of the machine is not easy. The problem is similar to that involved in recovering from mispredicted branches. At least four methods of handling interrupts have been considered.
Buffer-based

The execution results are stored in instruction stream order through the use of a history file or future file. When the interrupt occurs the history file can be used to correct errors. A future file is used to store results temporarily. Results are then written back when no interrupt can affect that result.

Guarantee-not-interrupted

Instructions only proceed to execute if no previous uncompleted instruction can cause an interrupt. This could cause stalls and degrade performance if functional units with long latency can not predict or detect interrupt early.

Checkpoint repair

This method stores the machine state at intervals during execution. Then the stored machine state is restored after the interrupt. This would be costly and complex.

Weakly precise interrupt

The interrupt is handled as being somewhat precise. Enough information is provided by the hardware that the interrupt handler can determine the exact sequence of instructions that caused the exception and can restart the program.

The schemes most widely discussed are buffer-based solutions, although some use guarantee-not-interrupted. They have much in common with recovering from mispredicted branches and so are implemented with not much additional overhead.

2.4 Advanced Architectures

There currently exist 4 major architectures available for microprocessors which may be capable of exploiting the parallelism found in a single instruction stream. Not all are suited for general purpose tasks that many of today’s commercial processors are expected to perform.
Superpipelining is, more simply stated, pipelining and refers to machines that have more than the typical 4 to 6 pipeline stages found in a RISC processor. Often this class of machine has 8 or more stages. Superpipelined machines exploits the concept of temporal parallelism (overlapping multiple instructions on the same hardware concurrently) by having a more deeply pipelined machine. The functional unit is fully pipelined to minimize stalls. Since more stages are used each operation is broken into smaller pieces with each piece being less complicated and less time consuming. Therefore clock speed can be increased and so the performance increases. Superpipelined processors are categorized as following the SISD (Single Instruction Single Data) model. A drawback is that often the branch misprediction penalty is much higher than other machines.

Vector Processors can be compared to SIMD (Single Instruction Multiple Data) machines where there are multiple execution units which execute the same instruction on multiple data. But rather than having multiple execution units, a single pipelined arithmetic unit runs the same operation repeatedly at very high speed. This is very appealing for scientific applications which rely heavily on matrix operations. However, general purpose applications primarily use scalar operations and vector processors are not better than other processors for scalar operations. Use of this architecture for general purpose machines runs counter to Amdahl’s advice, since that which the architecture does best is not used very often.

Superscalar machines are characterized by the ability to issue multiple instructions from a single instruction stream and execute them simultaneously. These machines can exploit parallelism between instructions by executing instructions that would normally be issued in sequence in parallel. At a macroscopic level, this would appear as a MIMD (Multiple Instruction, Multiple Data) machine with a serial instruction stream as the input. Since there are multiple instructions operating on multiple data concurrently, one source
calls its processor a Single Instruction stream/Multiple instruction Pipeline (SIMP) machine [7].

When viewed at a high level perspective very long instruction word (VLIW) processors resemble multiprocessors operating in a MIMD environment. They can be considered a derivative of superscalar machines with their multiple execution units. The difference is that rather than fetching multiple instructions, the instructions for all processors or execution units in a VLIW is contained in one very long instruction. The execution units operate in lock step mode. Even if an execution unit is not needed or otherwise unusable the instruction word would need to include a NOP or something similar for that processor.

The issues that are used to select which base architecture to start from are performance, device technology, code density, compatibility and compiler technology. For general purpose code, it seems apparent that vector processors would not achieve the same performance as the other architectures. Device technology might favor the VLIW and superscalar approaches over the superpipelined approach since device size is decreasing more rapidly than device speed is increasing. From a code density perspective, VLIW has a distinct disadvantage in that instruction units which are necessarily idle still require an instruction. For compatibility only the superscalar and superpipelined architectures offer the ability to reuse binaries from a current RISC or CISC instruction set architecture. VLIW and vector processors would need recompiled code at a minimum. Finally, compiler technology would most likely favor processors which required the minimum amount of scheduling assistance. The VLIW processor requires the compiler to find all possible parallelism. This is a distinct disadvantage. Superpipelined machines required compilers not much different than those that exist today, except for consideration of delayed branches. Superscalar machines make a spectrum of demands on the compiler. At one end,
the compiler must find all parallelism, similar to the VLIW. At the other, the superscalars
do the scheduling themselves limiting the burden on the compiler.

The resulting architecture decision favors the superscalar model, although a few
companies are following the superpipeline approach. An example of a current superpipe-
lined architecture is the MIPS R4000 which has 8 stage pipeline which takes 4 cycles to
complete. Several superscalars are currently commercially available. It has been estimated
that in the mid-90s they will be commonplace.

2.5 Survey of Superscalar Processors

Several examples of both RISC and CISC superscalar microprocessors will be
analyzed. Intel’s Pentium processor [12,10,11] is one of the newest CISC superscalars on
the market. The first implementation of the Metaflow architecture [15] was Lightning, a
superscalar RISC processor running the SPARC instruction set. This processor set never
became a viable product because of its complexity, but the concepts used were very
advanced and pushed the envelope of knowledge. A third architecture, the Motorola 88110
[16], also a superscalar RISC, used less complex means to achieve its high performance.

2.5.1 Pentium

The Pentium is a superscalar processor with two asymmetric pipelines. The U
pipeline, which receives precedence, can execute both integer and floating point instruc-
tions. The V pipeline can only execute simple integer instructions and the “floating point
register exchange contents” instruction. Although there are multiple pipelines, instructions
must be issued in order and complete in order.
2.5.1.1 Pipeline Details

The Pentium uses two differing length pipelines for integer and floating point instruction. Integer instructions follow the following sequence: Prefetch (PF), First Decode (D1), Second Decode (D2), Execute (E) and Write back (WB). Floating point instructions follow an eight stage pipeline are: PF, D1, D2, Operand Fetch (OF), First Execute (X1), Second Execute(X2), Write float (WF) and Error Reporting(ER).

The first two stages are common to both the U and V functional units. The functional split occurs after the first decode stage. Each unit has its own second decode stage.

Prefetch and D1

The PF stage accesses the instruction cache and can read two successive lines. The instructions are aligned and decoded so that the second instruction can be identified. A maximum of two instructions can be issued in parallel from the D1 stage.

Dependency checking is done by the D1 stage to ensure that no hazards occur. To achieve the maximum issue of two instructions, both instructions must be simple instructions not requiring microcode. An instruction is issued to the U pipeline first. If the second instruction cannot execute in the V pipeline, it must wait until the next cycle to be issued to the U pipe. RAW and WAW hazards are avoided by ensuring the source and destination registers of the instruction issued to the V pipe do not match the destination register of the U pipe. If they do match, the second instruction can not be issued in this cycle. Control hazards are handled by not allowing an instruction to enter the V pipe if a jump is issued to the U pipe.

Remainder of Pipeline

Instructions are executed in lock step fashion. If an instruction in one pipeline takes multiple cycles to move through a particular phase, the other instruction in that phase of the other pipeline is delayed until both operations complete. Then the instructions/
results move to the next stage together. The processor is fully pipelined so that as results leave the WF stage, they can be bypassed to stages prior to the first execute stage.

2.5.1.2 Speculative Execution - Precise Interrupts

Pentium does no speculative execution. Because instructions issue, execute and complete in order, the condition codes for branching are always set before a jump. There is therefore no uncertainty when a jump is issued. Similarly, the Pentium easily handles precise interrupts. All that must be done on an interrupt is to guarantee completion of the instructions beyond the execute stage in the pipe.

However, a penalty is paid for the simplicity. The compiler must be relied upon to keep up the two issue rate. Techniques such as loop unrolling are used to keep instructions flowing through both pipes without breaks in execution. The drawback is that code expansion occurs and the I/O bandwidth increases.

2.5.2 Metaflow Lightning

The Metaflow architecture uses hardware rather than compilers to find inherent parallelism in the instruction stream. Instructions are issued in sequence but executed and completed out-of-order. In addition, the processor performs speculative executions and handles interrupt precisely. Many new innovations were made in this architecture to allow such complexity.

The architecture is not geared, at least at a high level, to any particular instruction set. The first implementation is Lightning, a superscalar processor which executes the SPARC instruction set. It contains six execution units: one branch (in the I cache module), 2 integer ALUs and a memory address ALU (in the integer unit), 1 floating point adder
and 1 floating point multiplier (both in the floating point unit). Lightning consists of four
ASICSs and external cache.

2.5.2.1 Pipeline Details

The Metaflow has a 6 stage pipeline: fetch, issue, schedule, execute, update and
retire. All stages have an equal delay of one clock cycle, except some execution units. Fig-

Figure 3 Metaflow Architecture

![Diagram of Metaflow Architecture]

- Instruction Cache
- Fetch (up to 4)
- Issue Queue
  - Issue (up to 4)
  - Instruction n+m
  - DRIS - Data Structure
    - Data Structure
      - Automatic Renaming
      - out-of-order Execution
      - out-of-order Completion
      - Speculative Execution
- Instruction n+2
- Instruction n+1
- Instruction n
- Retire (up to 8)
- Register File
  - Multi-ported
- Schedule
- Instructions w/operands and IDs
- Execute
- Results and IDs
Figure 3 shows the logical flow of instructions and data. Pipeline stages are shown with arrows, while data/instruction storage is shown as blocks. The machine is decoupled so that instructions can move from stage to stage in varying quantities.

WAR and WAW hazards are handled by the issue stage which does register renaming. RAW hazards are handled jointly by the issue, schedule, execute, update and retire stages. The retire stage implements a future file to assist in maintaining a coherent register file on interrupts and conditional branches.

The soul of the Metaflow machine is the DRIS (deferred-scheduling, register-renaming instruction shelf). As its name implies, the DRIS shelves all instructions before execution and performs register renaming. The DRIS is implemented as a dataflow content-addressable FIFO (DCAF). The DCAF can hold a maximum of 64 instructions. This large number of instructions means that the scheduling window can be quite wide allowing very efficient scheduling. The data structure associated with each instruction is shown in Figure 4. The fields in the structure are filled at various times in pipeline. Each of the six pipeline steps will be discussed in detail as they relate to the DRIS and the ability to support out-of-order execution.

### Figure 4 Data Structure for the DRIS

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand 1</td>
<td>Color ID</td>
</tr>
<tr>
<td>Operand 1</td>
<td>Locked Reg Num ID</td>
</tr>
<tr>
<td>Operand 2</td>
<td>Locked Reg Num ID</td>
</tr>
<tr>
<td>Result</td>
<td>Latest Reg Num Value</td>
</tr>
<tr>
<td>Status &amp; PC</td>
<td>Dispatched? Inst Class Executed? PC</td>
</tr>
</tbody>
</table>
Fetch

Instructions are fetched from the 4 way set associative instruction cache. Each line contains eight instructions. To prevent the possibility of pipeline starvation, the fetch is self aligning with line crossing allowed. Every cycle 4 instructions can be fetched.

Issue

At this stage, all four instructions can be issued in a single cycle providing there is room in the DRIS. There can be at most 1 branch instruction and 3 integer/floating point operations issued. The issue process consists of allocating a unique ID to each instruction, register renaming, dependency checking and program counter maintenance.

Instructions are given IDs in strict issue order. The ID given is the index of the entry in the DCAF. The color bit is used to assist in dating instructions. Every time the issue process wraps around the DCAF, the color bit is toggled. Comparing two instructions is simply a matter of inverting the comparison sense. For example if instruction A has an ID of 0 and B has an ID of 1, if both color bits are the same B is younger. However, if the color bits were different A is younger.

Register renaming is similar to Tomasulo’s algorithm, where the new name of the register is the reservation station. In this case the ID of the instruction is the new name of the register. The original register name is stored in the DRIS destination-reg num field so that at retire time, the correct register can be written. In this manner whenever a new write occurs to a register, a WAW hazard is avoided without stalling. Since the ID is the new register name, when the instruction is retired the ID or register alias is once again free. So there are as many aliases available for renaming as there are entries in the DRIS.

For each operand of every instruction being issued, WAR hazard avoidance for previously issued instructions is handled by checking the DRIS to see if any entries would write to the source registers of the operands. This is done by accessing the DRIS on the
destination-reg num. If one or more is found with a match, the latest, or youngest, instruction that does so is the one that will generate, or has generated, the value for the operand.

If no such instruction is found, then the value is already in the register file and the operand is not locked. Correspondingly, the operand n-locked bit will be clear and the source register is written into the operand n-reg num field. If a source instruction is found, its ID (index) is written in the operand n-ID field. If the instruction has completed execution, the executed bit will have been set and the operand is not locked or prevented from being used. As before the locked bit is cleared. If the source instruction has not been set, the operand value is not yet available so the locked bit is set.

If indeed another instruction is found to write that register, the latest bit must be cleared. The instruction being issued always has its latest bit set, since it is always the last to write that register. This avoids any WAR hazards caused by having a previous instructions result be the source rather than the newer instructions result.

Since four instructions can be issued concurrently, there must be some dependency checking within those new instructions to ensure that no hazards exist. Since IDs are given in strict issue order this is not too complicated.

The last step in the issue phase involves cleanup. The program counter values for the instructions are calculated. In order to provide for precise interrupts and branching, the original program counter for each instruction must be known. The instruction class is stored so that the scheduler knows which execution unit can execute the instruction. The dispatch and executed bit are cleared.

Schedule

This stage selects the instructions to be executed. All instructions whose operands are unlocked are eligible for execution. In considering which instructions should be sent, the scheduler looks at the instruction class to determine the execution unit needed and age of the instructions. The oldest instructions with available executions get executed first.
Operands are retrieved from the DRIS, the register file or the bypassing registers as required. They are sent along with the opcode of the instruction and its ID for execution.

Execution

The execution is performed and results are generated. The ID arrives at the output of the execution unit at the same time as the results. Memory access instructions take two phases but are not discussed in this paper.

Update

The results are written back into the content field of the instruction identified by the ID value and the execute bit is set. In addition the DRIS is checked for any operands that may need the result. This is detected by matching the operand n-ID field. If there is a match, the locked bit is cleared.

Retire

This is the last stage in the pipeline. It is here that the results finally get written to the register file. At each cycle, instructions are retired that meet the following requirements:

- Execution is complete.
- All previous commands have been or are currently being retired.
- There was not an interrupt or other error caused by this instruction
- The register file has sufficient write ports. If not, the oldest are retired first.

2.5.2.2 Speculative Execution

Branches are treated just like ordinary instructions when it comes to out-of-order execution with one exception. If the operand of the branch is not locked, the branch can execute immediately. However, if the operand (condition code) is locked, the branch is shelved along with the predicted branch direction. Instructions that set condition codes unlock operands at Update time just like normal instructions would. The scheduling unit selects the oldest unlocked operand. If the branch proceeds in the same direction as pre-
dicted, no action must be taken. If not, the results of the erroneous branch must be repaired.

Since the update unit maintains a future file rather than a history file, recovering from a bad branch is as simple as removing those entries from the DRIS.

2.5.2.3 Precise Interrupts

If an interrupt was generated, the processor continues executing instructions in the DRIS until all instructions before the offending one have been retired. That means that the register file and the memory are both coherent. The retire stage supplies the program counter value for the

All instructions in the DRIS are then flushed, since they are not needed and should not have been loaded.

2.5.3 Motorola 88110

The Motorola 88110 is a 2 issue superscalar RISC microprocessor. It is the follow-on to the 88100. The instruction set architecture was from the 88100 with extensions for improved performance in the integer, floating point, and graphics units. The 88110 is upward compatible with the 88100 so that existing binaries can run unaltered.

The 88110 is not nearly as sophisticated as the Metaflow architecture when viewing out-of-order execution. Instructions are issued in order, with at most two being issued at one time. There are no instruction shelves or reservation stations except for the load/store and branch units. However, there are 10 execution units and several have longer latency than others thereby creating some false hazards. Scoreboarding is used to prevent both true and false hazards, but no register renaming is done. Unlike the CDC 6600, there are no reservation stations at each functional unit, so if a hazard is detected, instruction issue stalls. The execution units are bypassed so that as soon as an instruction completes
execution and the result is available it can be immediately forwarded to a new functional unit.

2.5.3.1 Pipeline Details

The pipeline is similar to a normal RISC machine in that there are only 4 phases: fetch, decode, execute and write-back. A difference is that internally, the clock is doubled so that some stages take less time than others. The timing is aligned so that there is no inherent blocking. As you can see in Figure 4, the fetch and execute stages take 2 minor cycles while the write-back and decode take only one minor cycle. This was possible because of the simplicity of the decode and write-back stages.

**Fetch**

The boundary is variable and line crossing is allowed in order to maximize the flow of instructions into the decode logic and to allow the compiler to spend its time in more important tasks such as scheduling. Two instructions are always presented to the decoder for issuing. In addition, two instructions are taken from the branch target instruc-
tion cache. This helps reduce the branch latency by having the alternate instructions readily available in the case of a mispredicted branch.

**Decode/Issue**

The decode stage does the issuing of instructions to the execution units and is responsible for checking for data hazards. It attempts to send both instructions received from the fetch stage every cycle. The issue unit is symmetrical so that either instruction can be sent to any execution unit. If the first instruction cannot be issued, neither can the second and issue stalls. If the first instruction can be sent, but the second can't, the second will become the first instruction the next cycle and the empty place will be filled by the fetcher.

As previously noted, a scoreboard is used to avoid data hazards. However, there are no reservation stations for instructions as was the case with the CDC 6600, except for the load/store and branch units. Therefore all data dependencies, excepting loads, stores and branches, are resolved in the decode unit. The lack of sophisticated control logic makes this stage operate quickly hence the ability to run in this stage for only one-half a cycle.

RAW hazards are avoided by checking the source operand register bits in the scoreboard to see if the data is available, while WAW and WAR hazards are avoided by checking the destination register bit scoreboard to see if another previously issued instruction will write to it (the register is locked). After the operands are checked and found to be available, the execution can proceed. The corresponding scoreboard bit for the destination register is set to block any other writes to that register. If a hazard is detected the pipeline stalls.

The designers felt that the penalty for blocking on WAW and WAR was not very significant, so they avoided the complexity of register renaming. However, there were still problems with the scoreboard and the issue logic. If the second instruction depends on the
result of the first, only the first could be issued and the second would need to wait. The 88110 relies on the compiler to do static scheduling and remove these types of dependencies. Since the 88100 did not have wide success, it was felt that there would be little resistance to requiring a recompilation to get better performance.

Perhaps, the most notable unit in the processor is the load/store. It is in this unit that instructions are not executed in order. If the issue logic sees a load or store, it immediately sends it to the load/store unit unless there is no room. The load unit is a simple 4 deep FIFO, while the store unit is comprised by three reservation stations. The stores proceed when the operands are available thereby eliminating write hazards. The loads proceed as soon as the data is available from the data cache.

The cache interface was designed to be lockup free so that loads and stores could “pass” each other if the cache has a miss. If a load access resulted in a miss, a store could be performed while waiting for the load to complete. A similar argument holds for stores. If a store results in a miss, a load can access the cache while waiting for the store to complete. This serves to reduce the latency and more efficiently utilize the memory system.

RAW hazards due to load and store crossing are prevented by ensuring that a load gets the most recent data, be it from the store reservation station or the data cache.

The branch unit also has a reservation station to prevent the processor from stalling while waiting for a branch instruction operand. Details of the branch strategy will be discussed later.

**Execute & Write-Back**

All execution units either complete in one cycle or are pipelined to accept a new instruction every cycle. Good balance in functional units greatly reduces the likelihood of a stall due to structural hazards. All units are bypassed so that results of the execution can be routed directly to another execution unit, overcoming a shortcoming of the CDC6600.
2.5.3.2 Speculative Execution

As was mentioned earlier, branch instructions can be placed in a reservation station if the operand is not available. Since the branch could be executed later, the machine dallies in speculative execution. Note that there is only one reservation station so further branches would need to wait until the previous branch has been resolved.

Branch prediction is handled statically with the compiler providing different opcodes depending on the likely path of execution. A 32 entry branch target instruction cache (TIC) is used to quickly provide the instructions for the taken path. The static bit determines which set of instructions should be executed.

While the branch is waiting at the reservation station, loads are handled differently than normal. Misses to the data cache from the load unit go unresolved. Stores can not write to the cache or the data bus. This makes it easier to recover from mistakes, since neither cache or main memory change their state during a speculatively executing branch.

If the correct path is taken, the branch is discarded and execution continues. If the prediction is wrong, the wrong path would have been taken and the machine must undo all changes to the registers and proceed down the correct path. This is performed by using a history buffer.

2.5.3.3 Precise Interrupts

Like the Metaflow, the 88110 also handles interrupts precisely. For synchronous interrupts, all instructions before the one which generated the interrupt are completed, and the machine state is restored through use of the history buffer. For asynchronous interrupts, all uncompleted instructions are aborted and once again the history buffer is used to fix up the register file.
2.5.4 Summary of Survey

Of the three that were analyzed, only the Pentium appears destined for success. The Motorola was a technical success. The Metaflow never became a commercial product. It is interesting, because both the Motorola and the Pentium are only 2 issue machines. And both issue instructions in-order. The relative simplicity of the superscalar implementation in Pentium and its high performance bode well for other CISC processors.
Chapter 3: Floating Point Units

The integration of a floating point unit into a processor requires a basic understanding of the operation of a floating point unit (FPU). An important factor in the design of the data path and I/O to the FPU is the types of data which will be supported. Addition of the FPU and its ability to generate exceptions/interrupts also has implications on the control path. The IEEE floating point standard 754 [19] specifies data formats, rounding, operations, exceptions, and traps. Most processors support this standard. For code to be portable, new processor FPUs should conform to this standard.

3.1 Data Types

Several precisions of floating point numbers are defined by the IEEE standard. All use a mantissa with an implied leading 1, an exponent and a sign bit. The mantissa is a signed magnitude number as opposed to 2's complement, which is how integers are represented. Therefore 1 and -1 are represented with the same mantissa but differing sign bits. Table 3 lists the various precisions that are IEEE standard value.

<table>
<thead>
<tr>
<th>Precision</th>
<th>Total Bits</th>
<th>Mantissa Bits</th>
<th>Exponent Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>32</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>Single Extended</td>
<td>≥43</td>
<td>≥32</td>
<td>≥11</td>
</tr>
<tr>
<td>Double</td>
<td>64</td>
<td>53</td>
<td>11</td>
</tr>
<tr>
<td>Double Extended</td>
<td>≥79</td>
<td>≥64</td>
<td>≥15</td>
</tr>
</tbody>
</table>

a. Implied 1 accounts for 1 bit. Actual stored bits is Mantissa Bits -1.
Some processors do not conform to the IEEE standard at internal stages of the FPU. This is typically done to increase the precision of the intermediate result, while minimizing the data storage requirements. The AT&T's 32 bit floating point digital signal processor, the DSP3210 [14], is an example of one such processor. It supports single precision numbers in its instruction set but uses a modified representation internal to the multiply accumulate unit. The mantissa is expanded by 8 bits for increased precision. When stored back in memory, the value is then either rounded or truncated to 32 bits.

3.2 Required Operations

To comply with the IEEE specification, several operations are required for floating point operands. They include arithmetic, square root, conversion and comparison operations. Many CISC processors optionally implement transcendental functions. Arithmetic operations include: addition, subtraction, multiplication, division, and remainder. Comparison and conversions are generally performed by the addition/subtraction unit (sometimes called the floating point ALU). In many designs, the integer multiply instruction also uses the hardware multipliers in the floating point unit.

In addition to the IEEE required operations, many new processors are incorporating multiply accumulate instructions. This conceptually involves three operands. (The third may be an implicit register similar to the Hobbit's processor accumulator as discussed in Chapter 4). The first two operands are multiplied and the result is added to the third operand. This is precisely what Digital Signal Processors do best. This requires modifying the FPU architecture so that the result of a multiply can directly feed the adder with no other instructions being issued. The typical higher level language expression of this operation is:

\[ C = (A^{++} \cdot B^{++}) + C \]
Although the operation is straightforward, the implementation on a RISC processor might take several instructions. An example coding (assuming all operands are already in registers and indirect addressing is supported) might take three instructions.

1. MultAcc A, B, C
2. Inc A
3. Inc B

On a traditional CISC processor, one complex instruction would execute the entire instruction including the post increments.

3.3 Instruction Execution

Several steps are required to execute the floating point instructions. Multiplication is the simplest and involves the following steps;

1. Add exponents, multiply mantissas.
2. Normalize result by shifting mantissa and modifying exponents.
3. Round the result.
4. Renormalize, adjust exponent.

Addition is slightly more complicated and entails the following:

1. Denormalize smaller number by modifying exponent to match larger number and shifting mantissa.
2. Add mantissas.
3. Normalize, adjust exponent.
4. Round
5. Renormalize, adjust exponent.

Pipelines are used to ease the performance burden on the hardware. The steps involved are often combined to lower latency. In some machines operations that are duplicated (such as normalize) is implemented as a single piece of hardware. The hardware is
then reused. The drawback with this is that throughput suffers. Instructions cannot be issued at a rate of one per cycle. For more complex operations, like divide and square root, this decrease in throughput is acceptable, but in simple operations, like addition/subtraction and multiplication, this is unacceptable.

In many processors, the different operations in the floating point unit(s) are broken into separate units so that if the multiply pipe is stalled due to a divide in progress, a floating point add or compare can still be issued.

3.4 Performance

Most processors have optimized their FPUs so that addition, subtraction and multiplication can be issued to the pipeline at a rate of one per cycle. Table 4 lists the floating point performance for a variety of current processors. Although latencies vary up to 6

<table>
<thead>
<tr>
<th>Processor</th>
<th>Operation</th>
<th>Cycle Time/Frequency</th>
<th>Throughput/Latency</th>
<th>Time Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>Add/Sub/Mult</td>
<td>66MHz</td>
<td>1/3</td>
<td>45ns</td>
</tr>
<tr>
<td>AT&amp;T DSP3210</td>
<td>ALU/Multiply</td>
<td>16.6MHz(^a)</td>
<td>1/1</td>
<td>60ns</td>
</tr>
<tr>
<td>Motorola 88110</td>
<td>ALU/Multiply</td>
<td>50Mhz</td>
<td>1/3</td>
<td>60ns</td>
</tr>
<tr>
<td></td>
<td>Compare</td>
<td></td>
<td>1/1</td>
<td>20ns</td>
</tr>
<tr>
<td>HP PA7100 [13]</td>
<td>ALU</td>
<td>99MHz</td>
<td>1/2</td>
<td>50ns</td>
</tr>
<tr>
<td></td>
<td>Multiple</td>
<td></td>
<td>1/2</td>
<td>50ns</td>
</tr>
<tr>
<td>AMD 29050</td>
<td>Add/Sub</td>
<td>40MHz</td>
<td>1/3-4</td>
<td>75-100ns</td>
</tr>
<tr>
<td></td>
<td>Mult (single precision)</td>
<td></td>
<td>1/3</td>
<td>75ns</td>
</tr>
<tr>
<td></td>
<td>Mult (double precision)</td>
<td></td>
<td>4/6</td>
<td>150ns</td>
</tr>
</tbody>
</table>

\(^a\) Actual clock rate is 66MHz, but a new instruction is issued every 4 cycles.
cycles, the total time in the pipeline is fairly constant at about 50-60ns. A caveat is that none of the speeds quoted are for processors running at 3V, therefore these values for latency must be derated before being used in comparison with the performance of the FPU proposed in Chapter 5.

The AMD 29050 [20,23] processor supports IEEE compliant floating point. However, lower latency through the pipeline can be achieved by using "fast floating point" mode. In this mode, denormal numbers are not supported. This increases performance at the expense of accuracy.

3.5 Exceptions

To comply with IEEE standards, the floating point unit must recognize five exceptions, invalid operation (0/0), divide by zero, overflow, underflow, and inexact (rounding). Detection of these conditions must set a sticky flag which must remain set until explicitly cleared. Trapping on exceptions, if enabled, should cause an interrupt. The interrupts must be handled in a precise manner.

Some processors allow instructions to be issued and complete out-of-order. They typically allow speculative execution. Mispredicted branches require the processor state to be reset to the state before the first incorrect instruction. On such processors, exceptions from the FPU are handled in a similar fashion. On other machines, where strict order is adhered to (instructions issue and complete in order) the pipeline might need to stall while waiting to ensure that a floating point instruction does not issue an exception (guarantee-not-interrupted). On processors like these, it is important to determine early in the pipeline whether the instruction will cause a fault.
Chapter 4: Current *Hobbit* Architecture

The *Hobbit* architecture is a registerless, 2 1/2 operand memory to memory architecture with variable length instructions. For architectural consistency there are few registers accessible by the users. Rather than using registers, local high speed memory is implemented with a stack cache which blends in with the memory-to-memory architecture. The instruction set architecture of the *Hobbit* microprocessor will be discussed very briefly in the next section, followed by details on the operation of the stack cache and how it is integrated into the architecture. An overview of the *Hobbit* processor's organization is presented with discussion about the major functional blocks. Finally, the execution unit pipeline is thoroughly discussed.

The information contained in this chapter is discussed in detail in [1,2,3,4,5,6]. The purpose of this discussion is to provide needed background for the following chapters.

4.1 Instruction Set Architecture

The *Hobbit* instruction set is small but efficient just like RISC processors. Instructions are either niladic (no operands), monadic (one operand) or dyadic (two operands) requiring that operands be explicitly stated. There are no triadic forms. In dyadic instructions, the destination is implicitly the left operand. However, an accumulator can be specified as the destination for a subset of instructions by selecting a variant of the basic instruction. Only a few carefully selected address modes are used.

4.1.1 Instruction Categories

There are about 46 instructions fitting into 8 different categories (See Table 5). This is similar in extent and size to the SPARC instruction set, and much smaller than the
Table 5: Hobbit Instruction Set

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>ADD[3], ADDI, DIV[3], MUL[3], REM[3], SUB[3], UDIV, UREM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>CMPEQ, CMPGT, CMPHI</td>
</tr>
<tr>
<td>Logical</td>
<td>AND[3], ANDI, OR[3], ORI, XOR[3]</td>
</tr>
<tr>
<td>Move</td>
<td>DQM, LDRAA, MOVE, MOVEA</td>
</tr>
<tr>
<td>Program Control</td>
<td>CALL, ENTER, CATCH, RETURN, CRET, POPN, KCALL, KRET</td>
</tr>
<tr>
<td></td>
<td>TESC, TESTV, JMP, JMP(F</td>
</tr>
<tr>
<td>Shift</td>
<td>SHL[3], SHR[3], USJR[3]</td>
</tr>
<tr>
<td>Other</td>
<td>CLRE, CPU, FLUSH(D,DCE,I,P,PBE,PTE), NOP</td>
</tr>
<tr>
<td>Tagged</td>
<td>TADD, TSUB</td>
</tr>
</tbody>
</table>

486 instruction set which includes rotate, string, bit and array instructions. Assuming the data is available on chip and the addressing modes are not indirect, most instructions execute in a single cycle. The exceptions are the multiply, divide, remainder, move and program control instructions. Some branches, both unconditional and conditional, can execute in 0 cycles!

4.1.2 Variable Length Instructions

One of the goals for the Hobbit architects was high code density, like CISC processors. The benefits are smaller program size, lower I/O bandwidth requirements and lower power consumption. This goal was attained by allowing variable length instructions in concert with carefully selected address modes such as: PC relative and stack relative addressing modes. These addressing modes will be discussed later. Instructions can be 1, 3 or 5 parcels long with the length of the instruction being encoded in the first parcel. This simplifies the implementation of the decode logic which must determine the instruction length in order to locate the next instruction.
4.1.3 Accumulator

The *Hobbit* processor is a two address machine where the source value is operated into the destination address/value. Calculations into a third address, known as the accumulator, are allowed with several of the arithmetic, logic and shift instructions. Consistent with the concept of no programmer visible register, the accumulator is not a separate register but rather a defined location on the stack. In that manner, the accumulator can be accessed like any other memory location. The dedicated location for the accumulator is the current value of the stack pointer (SP) + 4. This helps alleviate a potential shortcoming of dyadic machines compared with triadic machines.

4.1.4 Addressing Modes

A limited but sufficient number of addressing modes are allowed. This is also similar to RISC processors. They were designed to contribute to the high code density and fast decoding and processing. There are four addressing modes accessible to most instructions, two available for branching and one special purpose mode. They are summarized in the Table 6.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>embedded constants; sign or 0 extended</td>
<td>5,16 or 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 for KCALL</td>
</tr>
<tr>
<td>Absolute</td>
<td>data at an absolute address</td>
<td>16 or 32</td>
</tr>
<tr>
<td>Stack Offset</td>
<td>Calculated as SP</td>
<td>offset- - can be used to access in/out argument and local variables stored on the stack. If offset is negative, data will not be on the stack and cache coherency is not maintained.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 for RETURN</td>
</tr>
</tbody>
</table>
Table 6: Hobbit Addressing Modes (Continued)

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Offset Indirect</td>
<td>Similar to above except that address of data is contained in stack offset address.</td>
<td>16,32</td>
</tr>
<tr>
<td>Absolute Indirect</td>
<td>The data at the operand address is the target address. Used for JMP, CALL and LDRAA</td>
<td></td>
</tr>
<tr>
<td>PC Relative</td>
<td>Adds offset to PC to get target address. Used for JMP, CALL and LDRAA</td>
<td>10,32</td>
</tr>
<tr>
<td>Register</td>
<td>Only readable by user, writable by kernel</td>
<td></td>
</tr>
</tbody>
</table>

4.1.5 Flow Control

The flow control is handled slightly differently on the Hobbit processor than on other machines. Every instruction can be a branch to the next instruction or to an alternate instruction. There are 3 categories of program flow control: unconditional jumps, conditional jumps, and subroutine/procedure calls. Unconditional and conditional jumps can be folded with the preceding instruction allowing them to execute in 0 cycles. Conditional branching also feature static branch prediction.

Conditional branches are taken based on the setting of a flag in the processor status word. This flag is set by instructions which compare two operands (CMPEQ, CMPGT, CMPHI) and instructions which copy the carry (TESTC) or overflow (TESTV) bits in the process status word (PSW). (Only arithmetic instructions set the carry and overflow bits.) Branches can occur if either the bit is set or cleared. Static software based branch prediction is used. For efficient pipeline execution, the flag setting operation should be separated from the branch by at least 3 instructions which do not require off chip access. The effi-
cient subroutine and procedure calls will be discussed in detail in the next section describing the stack cache.

4.1.6 Stack Cache (Register Allocation)

Most microprocessors utilize on chip registers to provide the operands at a rate to keep the functional units busy. Caches are not usually used because registers are much faster. A cache typically needs to do tag comparisons, check valid bits and multiplex the outputs, whereas a register can be directly accessed. Another benefit to registers is that they always hit whereas caches always miss the first time and may cause a write back to memory, if write through is not supported. If the write policy is write through, all writes are written back to main memory and the cache. In write back, only the cache is updated. The main memory is only updated when a modified line in the cache is being overwritten with data from a different address. But a cache with no tags, valid bits or output multiplexors can be made to operate quick enough to compete with registers with the significant benefit of a much denser layout.

Rather than use registers, the Hobbit microprocessor implements the user data stack on chip as a dedicated cache. In most programming languages, the stack is used to keep track of subroutine calls, local variables and argument lists. In Hobbit, an efficient calling procedure was designed to make maximum use of the stack.

The size of the stack cache on Hobbit is 256 bytes. This is equivalent to 64 word length registers, however, since the stack is byte addressable, the stack can be viewed as storage for 256 bytes of information. (For ease of implementation, parcels and words must be aligned on half word and word boundaries, respectively.) Compared to SPARC register windows, Hobbit can be viewed as having 4 standard size “windows”. However, there is no notion of a standard size window on Hobbit. All “windows” are custom sized. The size of the
stack can be different from one implementation to the next since there is no "register" definition in the instruction set. There are two drawbacks in having large sized caches. The first is that context switching would require the entire stack to be flushed back to memory. The second is that code density decreases for addresses being accessed that are far from the Stack Pointer due to the maximum stack offset of 64 bytes in stack relative addressing mode. (If stack offsets are less than 64 bytes, a one parcel instruction format can be used. If stack offsets are greater than or equal to 64 bytes the 3 parcel format would be needed.)

4.1.7 Subroutine Calls

Researchers found that nearly one out of every 20 instructions in C language programs is a either a procedure call or a return. Since the architectures was evaluated based on quick execution of C programs, it is imperative that the subroutine call and return processes be efficient. A modification of traditional calling procedures was defined which can complete in as few as 4 cycles as described in Table 7.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Prelog.</td>
<td>Calculate outgoing arguments onto the stack. The stack pointer is not modified.</td>
</tr>
<tr>
<td>1</td>
<td>CALL</td>
<td>Store the PC of the instruction at which to resume execution on a RETURN on the stack at the empty position which is pointed to by the Stack Pointer(SP). Execution then continues at the target of the CALL.</td>
</tr>
<tr>
<td>2</td>
<td>ENTER</td>
<td>Adjusting the stack pointer(SP) to allocate enough space for the incoming variables, local variables, temporaries, and the largest possible outgoing variable list for the routine. This guarantees that enough room is available on the stack to store all of the necessary variables without overflowing the stack. If there is enough room, no further operation needs to be done. If not, the CPU moves data from the stack back to memory.</td>
</tr>
<tr>
<td>3</td>
<td>RETURN</td>
<td>De-allocates space on the stack by modifying the SP and continues execution at the PC originally stored by the call.</td>
</tr>
</tbody>
</table>
Table 7: Call Procedure (Continued)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>CATCH</td>
<td>Guarantees that at least as much space is actually in the stack as specified in the instruction.</td>
</tr>
</tbody>
</table>

Note: The object of a CALL must be an ENTER and the object of a RETURN must be a CATCH.

The stack cache is a circular buffer with built in range checking. Two internal registers, the Maximum Stack Pointer (MSP) and the Stack Pointer (SP) are used to manage the cache. These are 30 bit registers which store the highest address of data currently in the stack cache (MSP) and the lowest address of data on the stack cache (SP).

The ENTER and CATCH work with the MSP and SP to guarantee that enough room is on the stack. If, on an Enter, the MSP - SP exceeds the number of storage locations in the stack cache, enough data must be flushed back to memory to fit the new “window”. In the event that more space is needed than is present in the entire stack cache, the entire cache is flushed and only the locations nearest the SP are kept on chip. When returning, the CATCH does not know how much, if any, of the cache was flushed. It must tell the processor how much of its allocated space should be on the stack. The processor then tries to fill that portion of the cache.

Supporting the stack cache concept is the SP relative addressing mode. The offset value is always considered a positive value and is added to the current value of the SP during instruction fetch from the decoded instruction cache. Since the value of the SP only changes during ENTER and RETURN, there is no danger that the SP will change and invalidate the calculated addresses. Automatic range checking is done to determine if the resulting address is in the stack cache or in main memory.
4.2 Implementation

The *Hobbit* microprocessor is composed of 3 separate caches and 4 functional blocks as shown in Figure 6. The major blocks in the typical instruction flow are: I/O Unit, Prefetch Buffer (PFB), Prefetch/Decode Unit (PDU), Decoded Instruction Cache (DINC), Execution Unit (EU), and Stack Cache (SC). A Memory Management Unit provides address translation services to the other functional blocks. The I/O unit interfaces the
processor with the outside world. The prefetch buffer stores instructions in their encoded form. The prefetch decode unit takes instructions from the prefetch buffer, decodes, expands and places them in the Decoded Instruction Cache. The decoded instruction cache acts as a buffer or impedance matcher between the PDU and the EU. The Execution Unit grabs instructions from either the Decoded Instruction Cache or the Prefetch Decode Unit. It gets data from the Stack Cache or I/O unit. The results are either written to the stack cache or back to memory.

The Hobbit architecture is organized as shown in Figure 6 with several buses running between functional blocks and caches. A 32 bit bus sends addresses from the execution or prefetch decode unit to the I/O unit. This is used while prefetching and demand fetching of instructions or while reading-writing data off chip. The 32 bit data in bus is used to place data in the stack cache or in the execution unit or to place instructions in the prefetch buffer. The 32 bit data out bus is used to send results from the execution unit to the stack cache or off chip. The processor uses a 4 phase clock and level sensitive latches to decrease power consumption and increase flexibility.

4.2.1 I/O

The I/O unit interfaces the Hobbit core with the outside world. It is responsible for sending and retrieving instructions and transferring data to/from the external memory sub-system. It also generates control signals and accepts and prioritizes interrupts. Although Hobbit was designed such that it could make use of 0 wait state memories, economic and power consumption limitations require that Hobbit also support a much slower memory subsystem. Multiple CPU configurations are possible to allow redundancy and self checking. The clocks can be stopped by asserting a stop clock pin to guarantee low standby power.
A severe constraint on Hobbit processor and for many processors is the speed of the I/O system. This is not due to Hobbit processor but rather to the slower memory devices used in the system. Any modifications that alleviate this bottleneck will have great impact on the performance of Hobbit.

4.2.2 Prefetch Buffer

The prefetch buffer is more commonly called an instruction cache. In its current implementation, the prefetch buffer is 3Kbytes and organized as a 3 way set associative cache with a line size of 4 words or 8 parcels (16 bytes). A simple random replacement policy is used. The cache does support writes from either the data bus or the execution unit so that self modifying code may be run even when the instructions are stored in ROM. The prefetch buffer can deliver two words of encoded instructions every cycle to the prefetch decode unit. Instructions are stored in their encoded form because of the ease of decode and the benefit of more dense code.

4.2.3 Prefetch/Decode Unit

The role of the Prefetch Decode Unit (PDU) is to fetch instructions from off chip and then to align, decode and expand them. The PDU sends requests to the PFB and I/O frame and receives double words from both units. It receives control information from the EU and delivers 192 bit decoded instructions to the decoded instruction cache. A dedicated TLB is used to convert the virtual address requests to physical addresses.

Once the PDU is started, it can operate autonomously by following the instruction path. It can follow branches, using static branch prediction, if the target of the branch is encoded in the instruction or the address mode is PC relative. It only stops when it detects a branch with an indirect target or when the EU informs the PDU to start fetching from a
different address (because of misprediction or return from subroutine). If the instruction is not in the PFB, the PDU is capable of prefetching the instructions from off chip by sending a request to the I/O unit.

There are two modes of prefetching: aggressive and demand. In aggressive prefetching, the PDU will ask the I/O for a quad word whenever there is a miss in the PFB. In demand prefetching, the PDU can not itself make an I/O request. It can only do so if the EU requests an instruction that is not in the PFB.

Because of the variable instruction length, aligning is more complicated on Hobbit than on traditional RISC processors. However, the limited number of lengths (1, 3 and 5 parcels) and the intelligent encoding scheme allow the decoding to be simpler than on CISC processors. When an instruction is needed, the PDU takes a double word from the PFB or IO and aligns the instruction. The length of the instruction, 1, 3 or 5 parcels, is contained in two bits of the first parcel. This makes calculation of the address of the next instruction straight forward. A queue is used to align the parcels as they arrive from the prefetch buffer.

The decode block of the PDU expands the instruction to 6 words: left operand, right operand, PC (address) for this instruction, Next PC, Alternate Next PC, and control field. Both operands are sign extended 32 bit values. The address of the next instruction following this instruction, Next PC, is computed based on the length of the current instruction. And, in the event of a branch, the address of the instruction for the taken branch is computed, if possible.

Since every instruction has a next address and alternate next address field, every instruction can be a branch. This leads to the unique Hobbit technique of Branch Folding. This entails looking at the instruction after the current instruction to see if it is a branch. If it is, the Next PC and Alt-Next PC are substituted for the Next PC and Alt-NextPC of this instruction. In execution, this allows branches to execute in 0 cycle.
4.2.4 Decoded Instruction Cache (DINC)

The Decoded Instruction Cache is a direct mapped cache with 32 entries. Each entry has a tag indicating its PC. The direct mapped cache allows only one entry per set and an instruction can only be mapped to one set in the cache. This simplifies the operation of the DINC requiring only one tag compare per cycle. (Decoded instruction caches are described in detail in [21].) The DINC can accept a decoded instruction from the PDU and send a different instruction to the execution unit in the same cycle. This can be sustained at a rate of one input and output per cycle. If the DINC misses, it can be bypassed allowing the PDU to send instructions directly to the EU.

4.2.5 Execution Unit

The execution unit consists of three stages: IR (instruction register), OR (operand register) and RR (result register). Instructions from the DINC, or PDU, are latched into the IR stage. The IR stage then determines where the data, if any, for the instruction is located. The data could come from the stack cache, a subsequent pipeline stage of the execution unit, (through bypassing/forwarding), or off chip. When the data is available, it is latched into the OR stage and execution begins. The execution unit is not fully pipelined. That is to say that multiple cycle operations require the instruction to sit in the OR stage until execution is completed and no other instructions can be introduced. When execution is completed, the result is passed to the RR stage. From there the result is sent to the stack cache or off chip and possibly to the prefetch buffer (encoded instruction cache).

4.2.6 Stack Cache (SC)

The stack cache has two read ports (one for each operand), and one write port. It can do both reads and the write in a single cycle. From a circuit standpoint, the stack cache is
easier to implement than a generic data cache, since no tags are necessary. The stack cache looks like a traditional SRAM (static RAM) with its high density cells. Since address calculations are taken care of elsewhere, the addressing is simply the modulo-N address of the operand, where N is the number of words in the cache. It has advantage over register files because of the smaller area required in SRAM designs. The limitations of the stack cache come to light when compared to truly multi-ported register files. In order to do two reads from the stack cache, two separate copies of the data are kept requiring two separate caches. The single write occurs in both halves to the same address. If modifications of the architecture required more read ports it would just mean adding more area to duplicate the cache again. However, adding a write port would be much more difficult. The stack cache would need to be redesigned to allow two writes per memory cycle. This gives rise to the constraint of not adding a write port to the stack cache.

4.3 Execution Unit Details

The execution unit accepts decoded instructions from the DINC or PDU, fetches operand, executes the instructions and sends results back to the stack cache, or memory through the I/O block. It operates independently of the PDU except when a needed instruction is not found in the DINC. The execution unit employs a three stage pipeline (see Figure 7) which is slightly different than the pipeline of most machines since instruction fetch and decode operations have already been done. To keep the architecture clean only a few dedicated purpose registers are maintained by the EU. A detailed diagram of the implementation of the execution unit is shown in Figure 8.

There are a few registers which serve dedicated purposes in the execution unit. Table A-1 lists these registers and their purpose.
4.3.1 IR Stage

The task of the IR stage is to resolve indirection for the left and right operand and to request for data from the SC or memory for the next stage of the pipeline. Immediate data requires no generation and can be passed directly to the OR stage and so requires no action to be done. If the operand is an absolute address, the address is latched and a request is made for the data value. If the address is stack relative, the address is resolved by adding the offset to SP.

Indirect data access is performed by going through the IR twice. The first time through, the contents of the original address is passed to the stack or off chip. The address contained at that location is then relatched in the appropriate IR latch. This works for indirect operand values as well as indirect jumps, where the target address is returned to the next-PC field. The pipeline is fully bypassed/forwarded eliminating all read after write hazards. Data is fed back from the execute stage (OR) or write-back stage (RR) to the IR stage in the event of an indirect memory access where the pointer has just been updated but not yet written back to memory. If the destination address from any other stage matches the source...
Figure 8 Execution Unit Implementation
of the indirect address, the result address from that stage is forwarded before being written to the cache or memory.

The IR stage also controls the fetching of instructions for the EU. The branch information, PC, next PC, and alternate next PC is available from the DINC or the PDU. The alternate next PC field is latched directly. The next PC of the IR can come from the alternate next-PC field of any of the EU stages, the next PC field from the DINC/PDU, the stack cache or the data bus (indirect addressing). The next PC address which is latched by the IR is then used by the DINC to access the next instruction. The PC for the IR stage comes from the next PC field of the IR which was computed during the previous instruction cycle. This allows tracking of instructions for interrupt purposes.

Although most instructions execute in one cycle, those that take longer will stall the pipeline. In addition, the IR stage may be busy resolving an indirection in which case, the fetching from the DINC or PDU will be stalled.

4.3.2 OR Stage

The operand register holds the actual operands values to be used in the ALU or execute phase. The operands can come from various sources. If the values in the IR stage are immediate, they flow through unmodified to the OR stage. If the IR stage holds addresses which hit in the stack, the values come from the stack cache. If they miss they must come from memory unless they can be forwarded from a subsequent stage.

The destination address is also computed and latched by selecting either the SP +4 (accumulator - 2 1/2 address instructions), the SP (calls), or the left operand address as stored in the IR stage. The need to bypass from subsequent stages is determined by the comparison of the destination address with the addresses originally latched in the IR stage. The destination and operand address, alignment and word size must match if forwarding is going to be
allowed. If not, the OR is stalled until the results can be correctly read from memory later. (An example is writing to a byte but needing to read a word at the same address).

The OR stage may take more than one cycle if the data is not in the stack cache, in which case the IR stage can not move on to the next operation. If the instruction being executed is a divide or require I/O, the entire pipeline is stalled. In this case the IR can not resolve an indirection while the OR waits for an operand to be loaded.

4.3.3 RR Stage

The result register contains the result of the ALU operation. The ALU operation first takes the operands from the OR stage, aligns them and sign extends them (if needed). ALU operations always occur on 32 bit words. After the ALU operation is complete, the result is then properly aligned before being latched in the RR stage. The result is then written off chip or to the stack in the next stage.

Branching is handled by examining the flag in the PSW. If the branch is unconditional or the predicted condition is met (the flag is set/reset matching the prediction) no change is necessary since the prefetching followed the correct path. If the prediction is incorrect, the alternate PC is loaded into the next PC register, and all instructions in the IR/OR stage are invalidated.
This is a blank page.
Chapter 5: Proposed Modifications

Folding a floating point unit in to the existing architecture requires several steps. First the precision of data to be supported must be defined. Next, the instruction set must be extended to include floating point operations. Prior to integration, a functional layout of the FPU is required and some performance assumptions must be made. All of these steps are interrelated. For instance, definition of precision, dictates data path width which affects integration into the processor and FPU performance.

There are also limitations as to what can be modified in the existing architecture which serve as constraints to the solution. As was noted earlier, the Hobbit processor has no general purpose and very few special purpose user accessible registers. Maintaining this philosophy is a major stumbling block. An analysis of the stack cache revealed that only one write port is allowed, yet peak performance may require more than one instruction be retired per cycle.

The elegance and efficiency of the Hobbit microprocessor organization also complicate the introduction of a new functional unit. The variable length instructions complicate the ability to fetch more than one instruction per cycle. Although this can be achieved with additional hardware, the decoding time might be too inefficient. Finally, size and cost are important factors which limit the solution space.

5.1 Architecture of Floating Point Unit

The design of the FPU can be broken down into three components which are visible to the user. A decision must be made as to what data types will be supported. The instruction set must be expanded to comply with the IEEE standard. The basic organiza-
tion of the floating point unit needs to be discussed to understand the integration into the overall architecture. Finally, assumptions about performance must be made.

5.1.1 Data Types

Single and double floating point data types will be supported. Because of the 32 bit wide data paths, and 32 bit wide stack cache, single precision arithmetic can be handled with ease. Access to the operands is supported without stalling the pipeline if the data is in the stack or in the pipeline. For communications type algorithms, single precision arithmetic is sufficient as evidenced in the use of 32/40 bit representation in AT&T's DSP3210. For applications that may need increased precision at the cost of throughput double precision is also supported. However, double precision would require two accesses from the stack or I/O unit for each operand. Although the IEEE standard encourages supporting single extended formats, the increase in instruction format complexity to implement a third format is not reasonable. If the user wants better than single precision computations, he/she must use double precision.

5.1.2 Instruction Set Extensions

In keeping with the philosophy and ease of implementation of the reduced instruction set of the Hobbit microprocessor, a small but sufficient number of floating point instructions need to be added. These instructions must cover arithmetic, convert, and compare operations as required. No user visible registers or floating point register files should be used. However, all arithmetic operations should allow an implicit accumulator as the destination. This requirement is modeled after the current integer instructions. In addition, a separate multiply accumulate instruction must be supported which uses the accumulator as both input and output. This yields high dot product performance which is used heavily in communications.
processes. (The number of accumulators that are required is a function of pipeline latency and will be discussed later). Since the current integer multiply implemented in the EU uses successive addition, the FPU multiply sub-unit should also provide fast single word integer multiplication. Table 8 shows the instruction set extensions. Note that there are no instructions to move single or double precision floating point values. Since there are no registers, only memory locations, there is no need for special move instructions.

<table>
<thead>
<tr>
<th>Class</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>FADD,FSUB</td>
<td>Implied accumulator as destination also supported</td>
</tr>
<tr>
<td></td>
<td>FMUL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FDIV,FREM,FSQRT</td>
<td></td>
</tr>
<tr>
<td>Multiply-Acc</td>
<td>FMAC</td>
<td>Multiple accumulate, implied accumulator addend and destination</td>
</tr>
<tr>
<td>Conversions</td>
<td>F(i,s,d)TO(i,s,d)</td>
<td>Convert between integer, single precision and double precision floating</td>
</tr>
<tr>
<td>Comparison</td>
<td>FEQ,FGT,FGE,FUN</td>
<td>Equal, greater than, greater than or equal, and unordered</td>
</tr>
<tr>
<td>Test Exception</td>
<td>TESTFV</td>
<td>Test for Overflow</td>
</tr>
<tr>
<td></td>
<td>TESTFU</td>
<td>Test for Underflow</td>
</tr>
<tr>
<td></td>
<td>TESTFD</td>
<td>Test for Divide by Zero</td>
</tr>
<tr>
<td></td>
<td>TESTFI</td>
<td>Test for invalid</td>
</tr>
<tr>
<td></td>
<td>TESTFX</td>
<td>Test for inexact</td>
</tr>
<tr>
<td>Branch</td>
<td>FJMPT,FJMPE</td>
<td>Branch on floating point flag</td>
</tr>
</tbody>
</table>

A new dedicated register must be defined for control/monitoring of floating point operations. The floating point status word (FPSW) is a read/write register which contains bits pertaining to rounding mode, enabling/disabling of traps/interrupts on exceptions, sticky bits to record or mask floating point exceptions. It also includes a branching flag for use by
the floating point branch instruction. Five new instructions must be added to test the stored exception conditions in the FPSW (similar to the TESTC and TESTV integer based operations). And just like those instructions, testing clears the associated bit in the FPSW. It is important for performance reasons to have a separate branching flag for the floating point unit. It allows instructions in the two different functional units to execute out-of-order with respect to each other.

Comparison instructions will use the floating point ALU and should set the flag in the FPSW for use in branch control. Because branches can be taken on the flag condition being set or cleared, these four comparison instructions should be sufficient. Other instructions can be added which would increase performance at minimal cost, but these should provide a sufficient set.

5.1.3 Organization of FPU

The FPU will consist of two major sub-units: adder(ALU) and multiplier. They must be connected so that the output of the multiplier can be fed directly to the adder in support of multiply accumulate instructions. A prime consideration is the amount of hardware to be used in the FPU. The *Hobbit* microprocessor is a low cost-high performance processor. Accordingly, silicon area must be used judiciously. For communications application, single precision arithmetic is satisfactory. Double precision multiplication is not a necessity. A single precision floating point/single word integer multiplier (32x32) can save area and allow higher clock frequencies than a single cycle double precision or larger unit (54x54). (The current EU does not have a hardware multiplier for integer values). The performance degradation for going through the multiplier four times for double precision is insignificant compared to the memory access time for double precision numbers.
Figure 9 shows a possible organization of the FPU. Implementing the floating point multiply accumulate instructions is different from the integer or other floating point instructions where the accumulator is implicitly an output and explicitly can be an input. The floating point multiply accumulate instructions implicitly use the accumulator as an input and an output. To avoid a performance decrease, this requires three read ports: left operand, right operand and accumulator. There is only one output of the floating point unit which is rounded according to the IEEE standard. An option is offered similar to the AMD 29050 [20] which allows a fast add mode(subtract). The decrease in latency is achieved by not supporting denormals as the IEEE standard dictates.

For maximum precision, output from the multiply sub-unit to the ALU is wider than the required 54 bits for double precision. This makes the internal calculations more accurate than would have been achieved using rounding. The ALU unit is fully bypassed so that the add, subtract or multiply accumulate can be issued every instruction, or every other instruction (except when supporting denormals) even with dependent data in consecutive instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Single Precision</th>
<th>Double Precision&lt;sup&gt;a&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Throughput</td>
</tr>
<tr>
<td>Add/Sub</td>
<td>3/4</td>
<td>1</td>
</tr>
<tr>
<td>Multiply</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MultAcc</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

<sup>a</sup> Assumes double precision numbers can be delivered in one cycle. This can not occur so minimum throughput is 2 cycles.
Figure 9  Possible Floating Point Unit Organization

OR Stage

LEFT EXP  \rightarrow  ADJUST EXP  \rightarrow  13 BITS  \rightarrow  109 BITS  \rightarrow  109 BIT ADDER  \rightarrow  SINGLE/DOUBLE ENHANCED PRECISION

RIGHT EXP  \rightarrow  ADJUST EXP  \rightarrow  32x32 MULTI  \rightarrow  65 BITS  \rightarrow  109 BIT ADDER  \rightarrow  SINGLE/DOUBLE ENHANCED PRECISION

LEFT MANT  \rightarrow  65 MANTISSA BITS  \rightarrow  MULT_OUT  \rightarrow  ROUND/NORMALIZE

RIGHT MANT

ACC MUX

ACC MUX

RR Stage

IEEE SINGLE/DOUBLE PRECISION
Table 9 lists the performance of the sample FPU. The current Hobbit microprocessor operating frequency of 30MHz at 5V or 20MHz at 3V. At those rates, the latency of the pipelines ranges from 66-133ns (5V) to 100-200ns (3V). This compares reasonably with the computation times presented earlier. Of prime importance is that the issue rate for the FPU is one instruction per cycle for peak performance in single precision mode. Notice also that the latency for the add operation is much longer than for the multiply. This creates some complications because adds and multiplications cannot be interleaved without stalls due to the combined use of the round unit. However, the FPU can still achieve a peak throughput of one instruction per cycle.

5.1.4 Accumulators

The minimum requirements to support the extended instruction set is one accumulator that can store both single precision and double precision values. This can be accomplished in one double word space. Performance considerations may dictate multiple accumulators. The reference FPU implementation requires only one accumulator for sustained single cycle throughput of addition/subtraction or multiply-accumulate instructions. However, the multiplication sub-unit has a latency of two cycles with no internal forwarding. To achieve the same sustained single cycle throughput for single precision, two accumulators are needed. A compromise is to stall through an interlock if a dependency was found and two accumulators could not be provided. For double precision, the throughput of the multiply operation is already 4 cycles per operation. A delay of an additional cycle when using the accumulator might be acceptable.

The accumulator(s) must be accessible in both single precision and double precision formats. However, it is assumed that single precision and double precision accumula-
tors are not needed at the same time and so the same memory location can be used to store either single precision or double precision values.

For integer operations, the accumulator is defined as SP + 4. It is logical to assign the first floating point accumulator to location SP+8. Since this is a double word aligned address, it can support both a single and double precision value (as discussed earlier double precision values must be double word aligned to minimize address calculation overhead). The second single precision accumulator can be defined as SP+12, or the upper half of the double precision accumulator.

The Stack Pointer is aligned on quad words boundaries. This assignment is very efficient, since the accumulator space does not cross quad word boundaries. Table 10 illustrates the location of the accumulators in the stack frame.

<table>
<thead>
<tr>
<th>Stack Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP+12</td>
<td>Single FP Acc2</td>
</tr>
<tr>
<td></td>
<td>Double FP Acc1 hi</td>
</tr>
<tr>
<td>SP+8</td>
<td>Single FP Acc1</td>
</tr>
<tr>
<td></td>
<td>Double FP Acc1 lo</td>
</tr>
<tr>
<td>SP+4</td>
<td>Integer accumulator</td>
</tr>
<tr>
<td>SP</td>
<td>Empty - Storage for PC on next call</td>
</tr>
</tbody>
</table>

5.1.5 Memory Requirements for Integration of FPU

The floating point unit needs up to three operands every cycle. It can output a maximum of one operand per cycle. Accordingly, the stack cache must be modified to allow a minimum of three read ports. As was discussed earlier, this is a relatively straightforward task and does not involve adding a write port. When executing double precision instructions,
two reads to consecutive memory locations will be used rather than increasing internal intra-block data paths to 64 bits.

5.2 Integration

Integrating the reference FPU can be done in two ways. The FPU can be tightly meshed with the existing EU. Instructions can be single issued and subsequently sent down either the floating point pipeline or the integer pipeline. This is conceptually and hardware-wise the simplest solution. Very little extra logic, beyond the floating point unit, needs to be added. However, this might not be the best performance option. Based on the earlier code segment for implementing a dot product operation, it would take three instructions and three cycles for each multiply accumulate. At the current Hobbit processor rate of 20MHz, this translates to 13.3 MFLOPs (million floating point operations per second). Many commercial processors can easily exceed that rate. The AT&T DSP3210 can execute 25 MFLOPs.

Although the single issue machine would be much simpler, the performance would not be on par with the new processors which are typically superscalar. The superscalar alternative is to allow the FPU to operate in parallel and concurrent with the existing EU. In this case, instructions would need to be dual issued. Along with dual issue, dual retirement is required. Under this scenario, the 20MHz processor could deliver 20 MFLOPs.

This leads to the premise that the FPU and EU must operate in tandem. Several areas of the processor must be addressed to allow superscalar (pseudo-superscalar) operation. The Prefetch Decode Unit must be capable of issuing more than one instruction per cycle to the decoded instruction cache (DINC). The DINC must be able to accept multiple instructions and send an instruction to each functional unit every cycle. The functional units must be able to retire two instructions simultaneously. In addition some scoreboarding must
be done to ensure that no data hazards occur because of common operand/destination pairs between the two functional units.

5.2.1 Prefetch Decode Unit

Because of the variable width instructions, branch folding and queue logic involved and prefetch buffer line width, it is difficult to fetch multiple instructions at the same time. Currently, a double word (4 parcels) is presented to the PDU every cycle. Because instructions are not required to be aligned on word, double word or quad word, it may not always be possible to find two complete instructions in the queue and input double word. Therefore, two instructions can not be issued every cycle. However, it should be possible to send one or two instructions to the DINC every cycle, depending on the instruction encoding.

A modification must be made in the branch folding logic. Branches that test the flag in the PSW, so called integer branches (JMPT, JMPF), can only be folded with instructions destined for the integer unit. Branches that test the flag in the FPSW, so called floating point branches (FJMPT, FJMPF), can only be folded with instructions destined for the integer unit. The unconditional branch can be folded with any instruction. The instructions would be sent along with a timestamp to the decoded instruction cache. The POU would still need to prefetch instructions along the instruction stream and the timestamp should reflect that order. The maximum size, $B$, of the timestamp is given by

$$B = \log_2 N + 1$$

where $N$ is the number of instructions that may exist in the processor (DINC, EU, FPU) at any time.
5.2.2 Decoded Instruction Cache

In this cache, the instructions have already been decoded and contain the PC of the current instruction, the next PC, an alternate next PC, a left operand and right operand. Reading two instructions from this cache in one cycle as it is currently implemented would be difficult. This process would involve doing a read of a requested PC while simultaneously doing a tag compare. If the comparison was valid, the next PC value for that line would be taken and the process would repeat for the second instruction. These operations must be done in series.

An alternative is to use an intelligent FIFO with two read ports and one write port. The Prefetch Decode Unit would place instructions in the FIFO in order, one per cycle, and attach a timestamp on them. The FPU and EU could both read an instruction in the same cycle. A restriction is imposed that the instructions issue in order from the FIFO. This makes maintenance of the instruction stream much simpler.

5.2.2.1 Issue

The timestamp associated with each entry is included in the set of data given to the FPU and the EU. Every cycle the FPU and EU would either request a new instruction (ready) or tell the FIFO that it is busy. The timestamp of the currently executing or stalled instruction is also presented. The FIFO looks at the timestamp to determine which unit knows the true next-PC. The request with the most recent timestamp correctly knows which is the next instruction. Since branching can occur in either unit, there is an override so that one can demand precedence. If they both demand precedence, the unit with the older timestamp wins. The FIFO then tries to send two instructions. If the first instruction can not be issued because the desired functional unit is busy, no instructions are issued. If the first instruction can be
issued and the second uses the other functional unit and it is not busy both instructions are sent. Figure 10 shows the signalling for issue logic between FIFO and functional units.

### Figure 10 Issue - Intelligent FIFO to Functional Blocks

<table>
<thead>
<tr>
<th>Intelligent Decoded Instruction FIFO replaces Decoded Instruction Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction N+1 - can only issue if Instruction N issues and no structural hazard</td>
</tr>
<tr>
<td>Instruction N - must issue if no structural hazard</td>
</tr>
</tbody>
</table>

Floating Point Unit

Integer Unit (Old EU)

The hardware is a little more complex than the cache, since there are two read ports. The FIFO entries must have tags with the current PC just like the current cache. The difficulty involved is reading the FIFO. The read of the next instructions in the FIFO can occur in parallel with the tag compare of all entries in the FIFO. If the tags match, the instructions are ready to go. However, if the tag compare fails, a new read will be required at the FIFO entry where the tag match was found, if any.

In the normal case where the PDU prefetched along the correct path, this can run issue instructions much faster than a dual ported cache. In the case where the PDU erred in following the instruction stream this strategy would not be any slower than a dual ported cache solution.
5.2.3 Execution Unit Modifications

As was mentioned before, two widely used mechanisms for increasing throughput while avoiding hazards, are register renaming and scoreboardin. In the Hobbit architecture there are no registers to rename. To be sure, an aliasing scheme could be conceived, but it is not likely to be practical considering the 32 bit length of the addresses.

Habits, although handled with relative ease on the original Hobbit architecture, are much more complex when using dual issue or multiple functional units with single issue. A first concern is that the technique used to bypass all stages would require an incredible growth in the number of 32 bit data paths which would need to be routed. In the simplest case there are two points in each functional unit where feedback is possible (output of ALU/FPU and retirement register (RR)). When routed to the ten different places the forwarded data might need to go, complete bypassing yields a possible 40 paths. Each multiplexor at the OR/IR stage would require at least two more input busses, causing some multiplexors to have as many as 8 inputs.

Beyond the increased wiring complexity comes all of the hazards associated with superscalar machines which were discussed earlier. Because of the two pipes and varying latencies, more diligence must be given to detection. In addition, the performance cost of stalling the pipeline versus the increase area and power used by additional logic must be considered. Furthermore, the interaction between the two functional units must be determined.

Because the two functional units pipelines are fairly shallow, only a few instructions can exist at any one time. A small destination file can be used to tract the instructions in the pipeline. A difficulty encountered with the Hobbit's architecture is that the address identifiers are very long (32 bits), much longer than most processors register addresses (4-
6 bits). The key to arriving at a reasonable solution is to minimize the number of wide comparisons needed.

A destination file is proposed to assist in the hazard detection and forwarding. It has several entries: ID, destination, value, valid, most recent. The ID is the tag that was originally given to the instruction by the prefetch decode unit. The destination is the destination address of the instruction. The value is the result of the instruction and the valid bit indicates that execution is complete and is ready to be written back. The most recent bit indicates that, if there are multiple instructions that write a destination, this instruction is the most recent.

5.2.3.1 Address Generation/Operand Fetch (IR)

Intra-stage hazards can arise in the IR stage because of the indirection of operands and destinations. Consider the following code sequence when executed with the proposed issue strategy of issuing two instructions simultaneously.

```
PC. MULT *A,B /* where A contains the address 0xFF00 */
PC+1. ADD 0xFF00,1 /* address 0xFF00 = address 0xFF00+1 */
```

The multiply instruction will use the floating point pipe, while the increment will use the integer pipe. In theory, both instructions can be issued at the same time. In the first cycle, the multiply instruction moves into the IR and sends out the address A while the increment instruction sends out the address 0x00FF. In the next cycle, the indirection gets resolved and the IR sends out the address contained in A, 0xFF00, while the increment instruction has moved to the OR stage and latched the data from memory. Without further correction, the increment instruction will result in the address 0xFF00 being incremented instead of it being multiplied by B and then incremented. A solution is to delay the increment instruction so
that it also spends two cycles in the IR stage. Then a data dependency check can be made between the instructions in the IR stage.

Consider the case where the instructions are in the opposite order and no destination is indirect but rather the operand of the second instruction is an indirect.

```plaintext
PC.  ADD 0xFF00,1 /* address 0xFF00 = address 0xFF00+1 */
PC+1. MULT B,*A /* where A and B are integers, and A=0xFF00 */
```

In this case, the resolution of address A to a final data value must be completed after the add has completed.

Consider a final case where an operand in the first instruction is directly used by the second instruction.

```plaintext
PC.  ADD A,l /* address 0xFF00 = address 0xFF00+1 */
PC+1. MULT B,*A /* where A and B are integers, and A=0xFF00 */
```

In this case, the first instruction must complete before the resolution of *A begins.

The IR stage must be modified to handle intra-instruction dependencies because of these problems with indirection. Some simple rules can ensure hazards are detected and appropriately handled. If the older instruction has an indirect as its destination, it must be resolved first. If the younger instruction has an indirect for one of its operands, it must be resolved last. After resolving the addresses, if required, intradependency checks are done to ensure that one instruction is not dependent on the other. Otherwise, the instructions can be executed independently.

After these hazards have been noted, the IR stage starts its work. Addresses latched in the IR registers are compared against the destination file to see if any destination address matches its source operands. This takes a total of five comparisons for each entry in the destination file, two for the integer unit and three for the floating point unit. If one
matches, that one is the source for the operand. The operand is then latched into the OR stage when it becomes available, unless it is an indirection which requires another loop. If more than one matches, then the most recently written one is used. For this purpose, the register file must have at least 5 read ports, one for each IR/OR data latch. The input to the IR/OR latch can come from the ALU output, the destination file, or the more typical sources: stack cache, memory, instruction cache, or an optional data cache. Operands are locked into the OR stage as soon as available. If the operand is not immediately available, but rather it is in execution, the IR will stall on that operand while waiting for completion of execution. On every cycle the IR will again check the destination file, but since the instruction/destination that sourced the operand is known, the lookup can occur with the ID tag rather than the full address.

When the completely resolved destination is known, it is then written into the file along with its ID. Concurrently, a comparison is made to determine if any instruction in the destination file has the same address. If there is a match, the current one is marked as most recent. In that manner, multiple matches, like those discussed above are much easier to resolve. No comparison of timestamps or IDs is needed. If no space is available in the destination file for new entries, instruction issue is stalled until there is space. Meanwhile, the instruction is held at the OR stage and any subsequent instructions are stalled.

5.2.3.2 Execution (OR)

Execution can begin when all operands are ready and sitting in the OR stage. When an instruction is completed, the result is stored in the destination file. (The data can be forwarded back to the execute stage or the address generation stage.) The key used to write to the destination file is the ID tag. The valid flag is set to indicate that the contents of that field are valid.
5.2.3.3 Retiring Instructions (RR)

The small destination file that was discussed in the previous section must now be expanded to ensure that instructions are retired in order. (see Figure 11) The additional entries contain the significant bits from the PSW and FPSW, PC and resolved next PC. The inputs come from the ALU or FPU. This register file replaces the RR stage in the previous design. Entry into the register file may be done out of order. Feedback of the proper data is the responsibility of the IR stage as discussed earlier.

Results leave the retirement registers in strict PC order. There are two ports. One is dedicated to accumulator locations on the stack cache. The other is for writing to other locations on the stack, memory, or a potential data cache. In this manner, two instructions can be retired in one cycle.
5.2.4 Speculative Execution

Speculative execution does not occur in this architecture because instructions always execute in order within a functional unit and all conditional branches are classified as to which functional unit should execute them. The Hobbit processor branches on the flag in the PSW for integer instructions which is set by TESTC, TESTV, CMPEQ, CMPGT, and CMPHI and on the FPSW flag for floating point instructions which is set by TESTFV, TESTFU, TESTFD, TESTFI, TESTFX, FEQ, FGT, FGE, and FUN. (Arithmetic instructions do not set the branch flag. See Chapter 4.) The integer compare instructions set the flag based on compares done in the integer unit and the integer test instructions are affected by previously executed instructions that run only in the integer unit. A similar statement is true for the floating point unit.

Since within functional units the instructions are executed in order and between functional units the instructions move through the pipeline in unison, there is no possible data or control hazard concerning those condition codes.

5.2.4.1 Exceptions - Changes in Flow

Because the PC of the instructions and the next or alternate-next PC are available, it is easy to assure that changes in flow do not corrupt the results. In addition, exceptions are handled cleanly. The destination file keeps the state of the machine as it changes with each instruction so that it can be restored. For instructions which execute in the integer pipe, flags and bits that it can set are stored as their values at the end of the execution, while flags and bits it can not change are stored as their values prior to that cycle's start. Therefore, as instructions are retired in order, a copy of the machines state at that point in the strictly sequential instruction stream are known. Because of this procedure exceptions are handled precisely.
5.3 Supporting Memory Organization - Stack Cache Modifications

An original constraint was that the stack cache could not have multiple write ports. However, as mentioned previously, the accumulators and any other single location on the stack cache can be updated at the same time. This is necessary to achieve issue rates near or better than one per cycle. This apparent contradiction can be managed if the bottom three words just above the stack pointer, (SP+4 through SP+15) are actually kept in a register file. Whenever a read or write is addressed in that range, the register file with words (1-3) or bytes (4-15) responds. The remainder of the cache remains untouched. The original data for the three words nearest the SP is still kept in the stack cache. However, any modification to the register file will invalidate that data.

The quad word register file would require 5 read ports and 2 write ports. The stack cache would need 4 read ports in addition to the single write port. This optimizes the performance while minimizing the area used by the stack.

A difficulty is encountered when a procedure call is taken and the stack pointer is adjusted. The contents of the register locations must be transferred to the appropriate location on the stack. This can increase the call overhead by 3 cycles, which reduces the benefit of the fast calling sequence. The additional cycles needed to update the cache can be reduced by keeping track of which accumulators changed before the procedure call. Only those accumulators that changed must have the contents of the register file transferred to the stack cache. A similar problem occurs on a return from the call. The values in the accumulators must be restored. However, tracking which accumulator locations were modified can be used to determine which registers need to be restored.

This is a lot of overhead, but to allow maximum throughput, while maintaining the credo of NO user/programmer visible registers, it is required. Of course, a typical RISC processor would require that the general purpose register be save on procedure calls. In this light, the penalty from a call is not unusual.
Another option is to change the stack cache into a circular addressable register file with multiple write ports. While it is impractical to complete 2 writes in series to a cache (memory cells), it is not impractical to use a multi-ported register file. The area of the register file would be much larger than a single 4 ported cache. But the total area for the proposed implementation, including control logic, 4 ported stack cache and the three word register file, might be comparable to the area for a true multi-ported register file. In addition, the multi-ported register file would not incur the previously discussed overhead during a procedure call.

The benefit of both solutions is that they both conform to the current architectural model of a circular stack. The added cost of multiple ports is simply required when issuing multiple instructions.
Chapter 6: Conclusions and Future Work

The quality of the modifications is determined by several factors. Quantitative results determine how well the processor meets its goals, while qualitative metrics determine the extensibility and relative success in achieving the desired goals of integrating the FPU.

6.1 Effect of the Proposed Modifications

The effectiveness of the proposed changes is measured by the increased silicon area and power consumption compared to the increased performance. The accuracy and repeatability of the calculations is also a topic for evaluation.

6.1.1 Area

A key factor in evaluating the design is the area required to implement this new architecture. As a point of reference, the existing Hobbit microprocessor includes 419,000 transistors in 92 mm². The floating point unit is a requirement, so the area incurred by that block is not considered in evaluating the new organization of the processor. The major changes to the processor to support the new floating point unit are the expansion of the stack cache, modification of the PDU and DINC, and addition of the destination file.

The dual ported stack cache currently occupies less than 2% of the chip area. This would need to be doubled to a four ported cache and the multi-ported accumulator register file must be added. This would most likely require a 3-4% increase in area over the existing chip area. The PDU logic needs to be modified to allow 1 to 2 instructions to be placed in the DINC every cycle. This might take only 0.5% or less of the existing chip area. The
DINC is also less than 2% of the current chip area and modification might expand this area by a factor of 2. The addition of the destination file will likely take a maximum of 1-2%.

The total area cost for using the proposed architectural and organizational changes is less than 8-10%. Beyond the area for each block is the increased intrablock routing. This must be added to the previous estimate.

6.1.2 Power

The power consumption of the processor would definitely increase with all of these modifications. Studies need to be done to determine if it is feasible to shut down functional units when not needed. A concern in low power systems is that power might be wasted when speculatively executing instructions that are invalidated because of delayed mispredicted branches. However, the depth of speculative execution is minimal because of the depth of the pipeline, the presence of only two functional units each with a single reservation station, and the in-order issue. Therefore, the amount of power wasted is likely to be small and is a fair trade-off for the increased performance.

6.1.3 Performance

Peak performance as measured by an unrolled multiply accumulate with postfixes of the source addresses is shown in Table 11. It is possible to complete one multiply accumulate instruction every other cycle, assuming the data for the multiply accumulate is in the stack. Both the multiply and the first increment can be issued at the same time because one is an integer instruction and the other is a floating point instruction. The destination of the first instruction is not an indirect, the operands in the second instructions are also not indirects and the operands for the second instruction do not depend on the results of the first instruction. This allows both instructions to start indirect resolution and operand fetch
in the beginning of cycle 1. By the end of that cycle, the operand for the increment instruction is latched in the OR stage of the integer pipeline. Meanwhile the first instruction is still undergoing address resolution. At the start of cycle 2, the operands for the multiply-accumulate instruction are fetched and the increment for the second instruction has begun. At the end of cycle 2, the operands for the multiply-accumulate are latched and the increment is complete. The FPU starts its operation in cycle 3. Meanwhile, the results of instruction 2 are written to the destination file. The results must remain in the destination file until the FPU results are written to the destination file, then both results can be retired.

While the first two instructions are moving through the pipeline, the intelligent FIFO is looking to issue more instructions. At the end of cycle 1, the operand for the increment A instruction is latched and the IR stage is available. This allows a new instruction, increment B, to be latched in the IR stage. The next instruction in the stream is another floating point multiply accumulate. However, that instruction can not be issued because the IR stage is still busy resolving the indirection of the original multiply accumulate. The FMAC instruction can be issued in cycle 3 when the original indirection has been resolved.

Later in the pipeline the ability to forward results is shown. At the end of cycle 7, the rounded result of the first multiply accumulate is fed to the input of the add input of the second multiply accumulate instruction in cycle 8.

6.1.4 Accuracy

There is potential shortcoming with the current architecture of the floating point unit and the method of integration is present in this implementation. Single precision floating point calculations are inherently inaccurate because of the small number of mantissa bits. As was mentioned before, several processors, the proposed Hobbit FPU included, maintain extended precisions within the FPU while executing multiply accumulate
instructions. In the example unrolled loop which was presented, the accumulator is not internally forwarded. It is always rounded and stored in the destination file before being forward as an input to the adder for subsequent instructions. This will occur whenever one or more instructions separate two multiply accumulate instructions.

This causes two problems. The first is that the same instruction sequence of two multiply accumulates (MAC), which use the same accumulator, can result in two different results if one executes the MAC in consecutive time slots and the other executes them with one or more cycles separating them. This problem can be avoided by requiring that different accumulators be used when using consecutive MAC instructions.

It is likely that in most code sequences, the multiply accumulate will have some delay between them (to do pointer indexing and other control functions). This gives rise to the other problem which is that the single precision accuracy may not be sufficient. If it is not, a modification must be made so that the extended precision can somehow be recorded in the destination file and the accumulator space on the stack cache. In this manner, higher precision results can be obtained. The difficulty is in maintaining rounded and extended results at the same memory location.

6.2 Summary

This topic of this thesis was the integration of a floating point unit into the Hobbit microprocessor. The goals were to extend the architecture in such a way as to be consistent with the current programming model and organization. Important metrics were area, power and performance. After discussing several advanced processor strategies, it was decided that a superscalar model could be designed which would be consistent with the current design styles in the microprocessor arena.

The proposed modifications includes separate floating point and integer functional units. A maximum of two instructions can be issued in-order. Within a functional unit
instructions are executed in-order execution. However, between functional units, executions can occur out-of-order (primarily due to delays in accessing operands). Each functional unit has a single "reservation station", where an instruction can sit while awaiting operands or execution. A destination file was proposed to eliminate data hazards and to handle interrupt precisely. Speculative branching is allowed to the extent that while an instruction containing a branch is executing but not yet completed in one functional unit, the other functional unit can continue along the predicted path.

The modifications to the Hobbit architecture do not result in a deviation from the current programming model except for the inclusion of floating point instructions. This means that the processor will be binary compatible with the original. Existing programs do not use the new floating point instructions. Only integer multiplication instructions would execute in the floating point unit. Therefore there will be no significant increase in the performance of existing binary programs unless they heavily use the multiply instruction.

Future studies must be performed to thoroughly evaluate the extensibility of the new architecture. As mentioned previously, a severe performance limitation in the multiple accumulate loop is off chip data access. It may be required to add a data cache or some other fast memory to store coefficients and data for the floating point unit. Another topic for further research is the required accuracy of the multiply accumulate instruction for DSP type programs. A software model of the architecture remains to be written and tested on real code to ensure the viability of this proposal. Finally, the existing compiler needs to be modified to support the new instructions.
<table>
<thead>
<tr>
<th>#</th>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FMAC1</td>
<td>abc₁</td>
<td>Fetch abc₁</td>
<td>Resolve abc₁</td>
<td>Mul Man</td>
<td>Sum PP</td>
<td>Norm/ Denorm</td>
<td>Add</td>
<td>Round c₁</td>
<td>Avail</td>
<td>Write</td>
<td>Retire c₁</td>
</tr>
<tr>
<td>2</td>
<td>INC a</td>
<td>a</td>
<td>Fetch a</td>
<td>Inc a</td>
<td>Write a</td>
<td>Hold in destination file for previous instruction</td>
<td>Retire a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>INC b</td>
<td>b</td>
<td>Fetch b</td>
<td>Inc b</td>
<td>Write b</td>
<td>Hold in destination file for previous instruction</td>
<td>Retire b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>FMAC1</td>
<td>abc₁</td>
<td>Fetch abc₁</td>
<td>Resolve abc₁</td>
<td>Mul Man</td>
<td>Sum PP</td>
<td>Norm/ Denorm</td>
<td>Add</td>
<td>Round c₁</td>
<td>Avail</td>
<td>Write</td>
<td>Retire c₁</td>
</tr>
<tr>
<td>5</td>
<td>INC a</td>
<td>a</td>
<td>Fetch a</td>
<td>Inc a</td>
<td>Write a</td>
<td>Hold in destination file for previous instruction</td>
<td>Retire a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>INC b</td>
<td>b</td>
<td>Fetch b</td>
<td>Inc b</td>
<td>Write b</td>
<td>Hold in destination file for previous instruction</td>
<td>Retire b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>FMAC1</td>
<td>abc₁</td>
<td>Fetch abc₁</td>
<td>Resolve abc₁</td>
<td>Mul Man</td>
<td>Sum PP</td>
<td>Norm/ Denorm</td>
<td>Add</td>
<td>Round</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>INC a</td>
<td>a</td>
<td>Fetch a</td>
<td>Inc a</td>
<td>Write a</td>
<td>Hold for previous instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>INC b</td>
<td>b</td>
<td>Fetch b</td>
<td>Inc b</td>
<td>Write b</td>
<td>Hold in destination file for previous instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>FMAC2</td>
<td>abc₂</td>
<td>Fetch abc₂</td>
<td>Resolve abc₂</td>
<td>Mul Man</td>
<td>Sum PP</td>
<td>Norm/ Denorm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend**

- **IR Latch Point**
- **OR Latch Point**
- **Write to Dest File**
- **Retire out of Dest**
Chapter 7: References


### Table A-1: Registers in Execution Unit In the *Hobbit* Microprocessor

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Fields/Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>Configuration Register</td>
<td>Timer1,2 Configuration - defines which event to count.</td>
</tr>
<tr>
<td></td>
<td>Precache Configuration</td>
<td>Prefetch Mode - aggressive/demand</td>
</tr>
<tr>
<td></td>
<td>Cache enables</td>
<td>Cache enables - Prefetch, Instruction, Stack</td>
</tr>
<tr>
<td></td>
<td>Kernel Data Endian</td>
<td>Kernel Data Endian - big or little</td>
</tr>
<tr>
<td></td>
<td>PC extension</td>
<td></td>
</tr>
<tr>
<td>PSW</td>
<td>Processor Status Register</td>
<td>Address Mode - Physical/Virtual</td>
</tr>
<tr>
<td></td>
<td>User Endian - data big/little endian mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interrupt Priority</td>
<td>Interrupt Priority - mask interrupts</td>
</tr>
<tr>
<td></td>
<td>Enter Guard - Set on enter if stack not flushed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Execution Level - user/kernel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Current Stack Pointer - Stack or Interrupt Stack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Trace Basic Block</td>
<td>Trace Basic Block - force exceptions on change in program flow (jump, call, return)</td>
</tr>
<tr>
<td></td>
<td>Trace Instruction</td>
<td>Trace Instruction - forces exception after next instruction</td>
</tr>
<tr>
<td></td>
<td>Overflow - set if overflow on signed arithmetic operations, cleared if no overflow on signed arithmetic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Carry - set if carry out during unsigned arithmetic or borrow on SUB, cleared if no carry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flag - used for conditional branches.</td>
<td></td>
</tr>
<tr>
<td>MSP</td>
<td>Maximum Stack Pointer</td>
<td>Address of highest stack location stored on chip.</td>
</tr>
<tr>
<td>SP</td>
<td>Stack Pointer</td>
<td>Address of the top of the stack</td>
</tr>
<tr>
<td>SHAD</td>
<td>Shadow Register</td>
<td>Copy of the current stack Pointer</td>
</tr>
<tr>
<td>ISP</td>
<td>Interrupt Stack Pointer</td>
<td>Address of interrupt stack. Used as the stack pointer when the current stack pointer bit of the PSW is 0.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Fields/Details</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>STB</td>
<td>Segment Table Base</td>
<td>Virtual address mode - pointer to the start of segment table used in address translation</td>
</tr>
<tr>
<td>TIMER1&amp; 2</td>
<td>Timer1,2</td>
<td>Used to count events as defined in configuration register.</td>
</tr>
<tr>
<td>VB</td>
<td>Vector Base</td>
<td>Base address for vector-table for use in exception and interrupt processing</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
<td>Address of instruction currently being executed.</td>
</tr>
<tr>
<td>ID</td>
<td>Jtag ID Register</td>
<td>Used by test access port (TAP)</td>
</tr>
<tr>
<td>Fault</td>
<td>Fault Register</td>
<td>Address of instruction which caused current exception</td>
</tr>
</tbody>
</table>
Vita

Paul T. Holler was born in Allentown, Pennsylvania on November 15th, 1960 to Paul T. Holler, Sr. and Philomena B. Holler. He received his Bachelor of Science degree in Electrical Engineering from Drexel University in 1983. He is presently employed at AT&T Bell Laboratories as a member of the technical staff doing silicon design in the Microprocessor Design group. His current areas of research interest are microprocessor design and clock circuit design. Previously, he was employed at AT&T Microelectronics where his area of expertise was in product engineering and the test and debugging of VLSI and application specific integrated circuits.