Architectural Considerations in a Self-Timed Processor Design

William F. Richardson

CSTD-96-001

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

February 12, 1996

Abstract

There are fundamental differences in the structure of asynchronous and synchronous processors, and the problems of each approach require innovative solutions. This work explores some of the ways in which the structure of a specific design is affected by an asynchronous paradigm. The Fred architecture presented here is an example of such a design approach. The self-timed design philosophy directly results in a powerful and flexible architecture which exhibits significant savings in design effort and circuit complexity. Some of the architectural constraints discovered in the course of the research have simple yet unconventional solutions, which in turn provide additional benefits beyond their immediate application. Further, when an asynchronous philosophy is incorporated at every stage of the design, the microarchitecture is more closely linked to the basic structures of the self-timed circuits themselves, and the resulting processor is quite surprising in its simplicity and elegance.

ARCHITECTURAL CONSIDERATIONS IN A SELF-TIMED PROCESSOR DESIGN

by

William F. Richardson

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

March 1996

ABSTRACT

There are fundamental differences in the structure of asynchronous and synchronous processors, and the problems of each approach require innovative solutions. This work explores some of the ways in which the structure of a specific design is affected by an asynchronous paradigm. The Fred architecture presented here is an example of such a design approach. The self-timed design philosophy directly results in a powerful and flexible architecture which exhibits significant savings in design effort and circuit complexity. Some of the architectural constraints discovered in the course of the research have simple yet unconventional solutions, which in turn provide additional benefits beyond their immediate application. Further, when an asynchronous philosophy is incorporated at every stage of the design, the microarchitecture is more closely linked to the basic structures of the self-timed circuits themselves, and the resulting processor is quite surprising in its simplicity and elegance.

To Sarah.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ACKNOWLEDGMENTS

My thanks to John Bunda (**bunda@centtech.com**) and Aimee Severson (**aimee@gecko.ee.byu.edu**) for providing benchmark source code and to Carl Bruggeman (**bruggema@cs.indiana.edu**) for suggesting the implicit **doit**. Thanks also to Steve Nowick (Columbia University) and Steve Furber (University of Manchester) for their helpful comments.

I would also like to thank those people in the University of Utah Department of Computer Science who provided advice, guidance, assistance, and comic relief: Lüli Josephson, Nick Michell, Prabhakar Kudva, Colleen Hoopes, Mary Rawlinson, Raelynn Potts, Al Davis, David Hanscom, Ganesh Gopalakrishnan, Kent Smith, Rory Cejka, Jim de St. Germain, Scott Brown, Jimmy Miklavcic, Marshall Soares, John Kawai, and many others.

I am extremely grateful to Erik Brunvand, without whom I would still be in the graphics group, with several years left to go. Advisors as pleasant and supportive as Erik are rare indeed.

Finally, I am most appreciative of my wife Sarah, who cheerfully left family, friends, career, and warm weather to support me in my quest. Without her efforts, none of this would have been possible.

CHAPTER 1

INTRODUCTION

As computer systems continue to grow in size and complexity, the difficulty in coordinating the activity of the system components also grows. A major cause of this problem lies in the traditional synchronous design style in which all the system components are synchronized to a global clock signal. For example, simply distributing the clock signal throughout a large synchronous system can be a major source of complication. Clock skew is a serious concern in a large system and is becoming significant even within a single chip. At the chip level, more and more of the power budget is being used to distribute the clock signal, and designing the clock distribution network can take a significant portion of the design time. These symptoms have led to an increased interest in *asynchronous* designs. General asynchronous circuits do not use a global clock for synchronization, but instead rely on the behavior and arrangement of the circuit elements to keep the control signals proceeding in the correct sequence. In general these circuits are very difficult to design and debug without some additional structure to help the designer deal with the complexity.

1.1. Self-Timed Circuits

Traditional clocked synchronous systems are an example of one particular structure applied to circuit design to facilitate design and debugging. Important signals are latched into various registers on a particular edge of a special clock signal. Between clock signals information flows between the latches and must be stable at the input to the latches before the next clock signal. This structure allows the designer to rely on data values being asserted at a particular time in relation to this global clock signal.

Asynchronous circuits provide a different approach. In an asynchronous system, events are restricted to a particular *sequence*. The time at which these events occur is a separate issue. For correct functioning, it is important only that the correct order of events be maintained within the circuit.

Self-timed circuits [35] are a subset of the broad class of asynchronous circuits, which applies a different type of structure to circuit design. Rather than let signals flow through the circuit whenever they are able as with an unstructured asynchronous circuit or require that the entire system be synchronized to a single global timing signal as with clocked systems, self-timed circuits avoid clock-related timing problems by enforcing a simple communication protocol between circuit elements. This is quite different from traditional synchronous signaling conventions in which signal events occur at specific times and may remain asserted for specific time intervals. In self-timed systems it is important only that the correct *sequence* of signals be maintained. The timing of these signals is an issue of performance that can be handled separately.

1.2. Communication Protocol

Self-timed protocols are often defined in terms of a pair of signals that 1) request an action and 2) acknowledge that the requested action has been completed. One module, the sender, sends a *request* event to another module, the receiver. Once the receiver has completed the requested action, it sends an *acknowledge* event back to the sender to complete the transaction.

This procedure defines the operation of the modules by following the common idea of passing a token of some sort back and forth between two participants. A single token is owned by the sending module, and to issue a request event the sender passes that token to the receiver. When the receiver is finished with its processing, it produces an acknowledge event by passing the token back to the sender. The sequence of events in a communication transaction is called the protocol. In this case the protocol is simply for request and acknowledge events to alternate, although in general a protocol may be much more complicated and involve many interface signals.

1.3. Motivation

A self-timed paradigm offers several potential advantages beyond the savings in design effort which result from eliminating the global clock distribution circuits. Because of their request/acknowledge communication protocol, self-timed circuits separate timing from functionality, which leads to an increase in composability. Systems may be constructed by connecting components and subsystems based only on their functionality without having to consider their timing requirements. Incremental improvements in speed or functionality are possible by replacing individual subsystems with newer designs without changing or retiming the system as a whole. The systems are robust since subsystems continue to operate over a wide range of process variations, voltage differences, or temperature changes. Because self-timed systems signal completion as soon as they are able, self-timed pipelined systems tend to display average case behavior, as opposed to the worst-case behavior typical of traditional synchronous systems. Additionally, self-timed systems do not incur the power overhead of distributing a free running clock across the entire system, and since the circuit elements make signal transitions only when actually doing work or communicating, large systems can show greatly decreased power dissipation in some technologies, especially during quiescence.

In fairness, there are some potential disadvantages as well. Self-timed circuits often exhibit an increase in circuit size, an increase in the number of wires connecting parts of a system, possible performance penalties due to the larger circuits, and a marked difference in design and test procedures from those used in standard synchronous circuits. However the potential advantages of self-timed circuits are analogous to those evinced by object-oriented programming languages, in which the advantages of using encapsulated software objects without individually tailoring each instantiation outweigh the disadvantages of increased code size and minor performance degradation.

1.4. Thesis Objectives

There are fundamental differences in the structure of asynchronous and synchronous processors, and the problems of each design require innovative solutions. The focus of this work is not to demonstrate the advantages of self-timed processors over equivalent synchronous implementations but to explore some of the ways in which the structure of a specific design is affected by an asynchronous paradigm. It is certainly possible to implement a conventional microprocessor design using self-timed circuits. However, when the asynchronous philosophy is incorporated at every stage of the design, the microarchitecture is more closely linked to the basic structures of the self-timed circuits themselves, and the resulting design can be quite surprising in its simplicity and elegance. The Fred architecture is an example of such a design approach, which is described in the remainder of this document. The self-timed design philosophy directly results in a powerful and flexible architecture which exhibits significant savings in design effort and circuit complexity.

1.5. Thesis Structure

Chapter 2 contains an overview of related work in asynchronous processor design. Several self-timed microprocessor designs are examined, and the overall state of the current research is summarized.

Chapter 3 describes the micropipeline design methodology and the circuit components which were used in this research. Micropipelines are a form of self-timed circuits which are particularly well-suited to an incremental design approach.

The major architectural features of the self-timed Fred processor are related in Chapter 4. A brief discussion of the instruction set, general structure, and other salient points is presented here.

Chapter 5 is devoted to the exception handling mechanism for the Fred processor. Exception conditions can impose unusual difficulties on a self-timed design. This chapter describes a solution to the particular problems found in the Fred design.

Specific details of Fred's implementation as a the VHDL simulation model are related in depth in Chapter 6. Design decisions and interface requirements are explained. The internal workings of the various functional units of the processor are described.

Chapter 7 describes and interprets experimental results. A suite of 14 test programs was used to measure performance and behavioral characteristics of the Fred processor under a variety of configurations.

Chapter 8 discusses possibilities for future investigation. Future work may take place to realize a physical or commercial implementation or simply to explore further the design space made possible by the Fred architecture.

Chapter 9 describes the conclusions that have been drawn from this research, examines some of the more innovating features of the Fred architecture, and discusses how the findings may be applied to synchronous systems.

CHAPTER 2

RELATED WORK

In spite of the possible advantages, there have been very few asynchronous processors reported in the literature. Early work in asynchronous computer architecture includes the Macromodule project during the early 70s at Washington University [6] and the self-timed dataflow machines built at the University of Utah in the late 70s [7].

Although these projects were successful in many ways, asynchronous processor design did not progress much, perhaps because the circuit concepts were a little too far ahead of the available technology. With the advent of easily available custom ASIC technology, either as VLSI or FPGAs, asynchronous processor design is beginning to attract renewed attention.

2.1. CalTech's Asynchronous Microprocessor

The first asynchronous VLSI processor was built by Alain Martin's group at CalTech [24,41]. It is completely asynchronous, using (mostly) delay-insensitive circuits and dual-rail data encoding. The processor was designed as a set of concurrent programs written in a CSP-like language, then compiled into a circuit through a series of program transformations. It has been fabricated in both CMOS and GaAs and was found to work well. As a proof-of-concept vehicle, it was a great success. The goal was not to implement an innovative architecture but to show that self-timed circuits in general, and circuits generated from program descriptions in particular, could be used to build a circuit as complex as a processor. The processor as implemented has a small 16-bit instruction set, uses a simple two-stage fetch-execute pipeline, is not decoupled, and does not handle exceptions.

2.2. NSR

Erik Brunvand's group at the University of Utah built a small 16-bit pipelined RISC processor [4,32] using self-timed techniques. The NSR processor is a general purpose processor structured as a collection of self-timed units that operate concurrently and communicate over bundled data channels in the style of micropipelines. These units correspond to standard synchronous pipeline stages such as Instruction Fetch, Instruction Decode, Execute, Memory Interface, and Register File, but each operates concurrently as a separate self-timed process. In addition to being internally self-timed, the units are decoupled through self-timed FIFO queues between each of the units which allows a high degree of overlap in instruction execution. Branches, jumps, and memory accesses are also decoupled through the use of additional FIFO queues which can hide the execution latency of these instructions. The NSR was tested and found to work well. It is pipelined and decoupled, but does not handle exceptions. It is a very simple processor with only 16 instructions, since it was built partially as an exercise in using FPGAs for rapid prototyping of self-timed circuits [3].

2.3. Amulet

A group at Manchester has built a self-timed micropipelined VLSI implementation of the ARM processor [17] which is an extremely power-efficient commercial microprocessor. The Amulet is more deeply pipelined than the synchronous ARM, but it is not decoupled although it allows for instruction prefetching. Its precise exception model is a simple one, since its single ALU causes all instructions to issue and complete sequentially. The Amulet has been designed and fabricated. The performance of the first-generation design is within a factor of two of the commercial version [29]. Future versions of Amulet are expected to close this gap.

2.4. STRiP

The STRiP processor was developed at Stanford University [9]. Although the designers refer to it as "self-timed," the STRiP processor is not an asynchronous machine in the sense used in this document. Instead it is a synchronous machine which dynamically alters its clock period on a cycle-by-cycle basis. This dynamic clocking method sequences the pipelined functional units in lock-step but adjusts each clock period to match the instructions currently in progress. This processor has been simulated but not built.

2.5. Counterflow Pipeline Processor

The Counterflow Pipeline Processor (CFPP) is an innovative architecture proposed by a group at Sun Microsystems Labs [38]. It derives its name from its fundamental feature, that instructions and results flow in opposite directions in a pipeline and interact as they pass. The nature of the Counterflow Pipeline is such that it supports in a very natural way a form of hardware register renaming, extensive data forwarding, and speculative execution across control flow changes. It should also be able to support exception processing.

A self-timed micropipeline-style implementation of the CFPP has been proposed that takes advantage of the local control and geometric regularity of the micropipeline and should enable fast and efficient VLSI layout. The CFPP is deeply pipelined and partially decoupled, with memory accesses launched and completed at different stages in the pipeline. It can handle exceptions, and a self-timed implementation which mimics a commercial RISC processor's instruction set is under development. The potential of this architecture is intriguing but still unknown.

2.6. FAM

The FAM is a 32-bit, fully asynchronous microprocessor design from the University of Tokyo [5]. It contains a four-stage pipeline with a central controller and uses four-phase handshaking. It uses asynchronous circuits to implement an otherwise fairly conventional RISC processor, which does not take advantage of the micropipeline approach. The FAM has been designed and simulated for a 0.5μm CMOS technology but has not been built.

2.7. TITAC

Researchers at the Tokyo Institute of Technology have designed and built TITAC, a simple 8-bit asynchronous microprocessor [27]. It uses dual rail encoding for the data paths and has been built using 1µm CMOS gate array technology. The architecture uses a single-ALU von Neumann design and is only slightly pipelined. It is not decoupled and does not handle interrupts.

2.8. Hades

A group at the University of Hertfordshire have proposed a superscalar asynchronous processor design named Hades [12]. It has a simple RISC instruction set and resembles a conventional synchronous processor in many ways. It issues instructions in sequential order but allows out-of-order completion. Explicit forwarding is done between functional units under the direction of a central scoreboard mechanism.

2.9. SCALP

SCALP is a superscalar pipelined asynchronous processor design, developed at the University of Manchester [13]. It was designed to maximize code density and parallelism and to minimize power consumption. Multiple queues are used to forward results between functional units. SCALP does not use a register file, but instead each instruction indicates which of several functional units should receive its computed result. The processor has been simulated with a gate-level VHDL model but has not been built.

2.10. Fred

To date, a number of self-timed architectures have been proposed. Most of these are either small processors designed to explore some specific aspect of self-timed circuits such as formal verification or synthesis, or they are larger processors modeled closely on existing synchronous designs. The Fred architecture presented in this document attempts to address some of the more fundamental issues related to a large-scale processor built with self-timed design approach. For instance, of the processors just described, only the CFPP and the AMULET specifically address the issue of exception handling.

Fred is a self-timed, decoupled, concurrent, pipelined computer architecture.[1] It dynamically reorders instructions to issue out of order and allows out-of-order instruction completion. It handles exceptions and interrupts. Several features of the Fred architecture are directly related to its self-timed design, such as the decoupled branch mechanism and exception model. Early versions of the Fred architecture have been discussed elsewhere [33,34]. Detailed descriptions of the most recent implementation are contained in the following chapters.

---

[1]Fred is not an acronym, and it does not mean anything. It is just a name, like “Pentium” or “Alpha.”

CHAPTER 3

MICROPIPELINES

Although self-timed circuits can be designed in a variety of ways, all of the circuits described in this document use two-phase transition signaling for control and a bundled protocol for data paths. Two-phase transition signaling uses *transitions* on signal wires to communicate the request and acknowledge events between circuit modules. Only the transitions are meaningful; a transition from low to high is the same as a transition from high to low and the particular high or low state of each wire is not important.

A bundled data path uses a single set of control wires to indicate the validity of a *bundle* of data wires. This requires that the data bundle and the control wires be constructed such that the value on the data bundle is stable at the receiver before a signal appears on the control wire. Bundled protocols are a compromise to complete self-timing because they impose this timing constraint, called the *bundling constraint,* which must be met between communicating modules. Two modules connected with a bundled data path are shown in Figure 3.1, and a timing diagram showing the sequence of the signal transitions using two-phase transition signaling is shown in Figure 3.2. For details on this and other self-timed communication protocols, the reader is referred to Seitz [35].

3.1. Control Modules

Control circuits for transition signalling may be built from the set of simple building blocks shown in Figure 3.3. The *XOR* gate provides the OR (merge) function for two transition signals, since a transition event on either the first input OR the second input will produce a transition on the output. For correct functioning, two input events may not arrive so close to each other that the

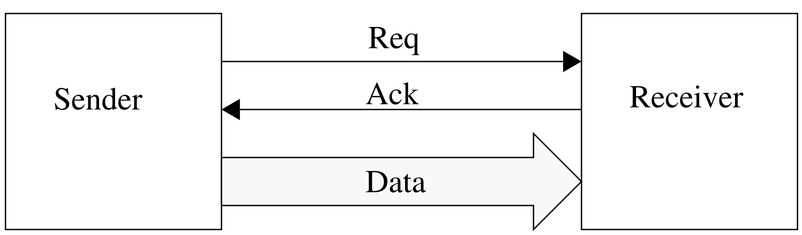

Figure 3.1 A bundled data interface

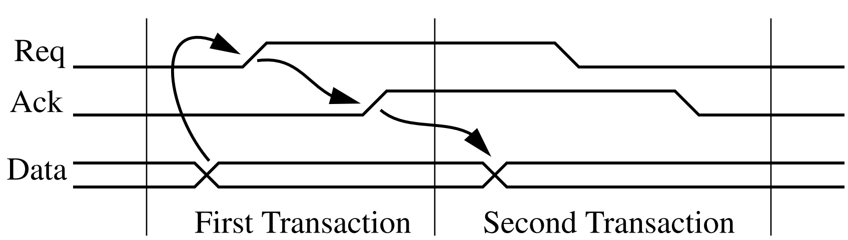

Figure 3.2 Two-phase bundled transition signalling

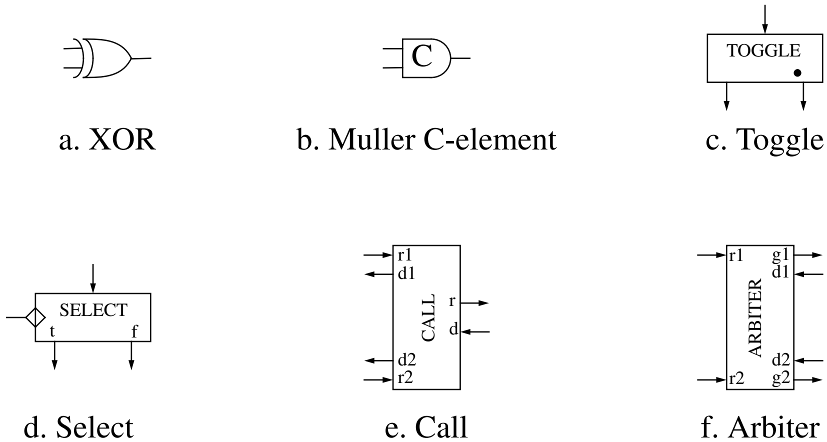

Figure 3.3 Self-timed event logic elements

two output transitions cannot be reliably distinguished. In most cases, the input events are mutually exclusive by design, but additional control modules are available to ensure that this requirement is met, as will be seen shortly.

The *Muller C-element* provides the logical AND (rendezvous) function for transition events. Although drawn as a standard AND gate with the letter “C” inside, this gate is not purely combinational but contains some internal state. When both inputs are at the same level, the output is also at that level. However, the output does not change until both inputs have changed.

The *toggle* module alternates incoming transition events between two outputs. Following some master clear signal, the first incoming transition is passed to the output with the dot, the second transition is passed to the output without the dot, the third to the dot output again, and so forth.

The *select* module is similar to the toggle, but instead of simply alternating, the output is selected according to the level value of the diamond input when the input transition arrives. The select input is subject to the bundling constraint, in that it must be stable before the input transition arrives.

The *call* module allows two self-timed modules to share the resources of a third. An incoming transition event on either of the request (R) inputs is passed to the single R output. The done (acknowledge) transition is then returned to the corresponding D output, to complete the handshake. This circuit operates correctly only when one handshake cycle has completed before the other begins.

Finally, the *arbiter* can be used to provide mutual exclusion between two unrelated systems. Incoming request (R) transitions produce grant (G) output transitions, but only one grant is made available at a time. If two request transitions arrive close together, one grant output transition is issued, and the other G output transition event is delayed until the first has been acknowledged by a transition on its D (done) input. When used together with the call module, it allows two independent processes to share a common resource in a mutually exclusive manner. Because the arbiter module must provide stable outputs under metastable conditions, the grant output events are not generated until any internal metastability has been resolved.

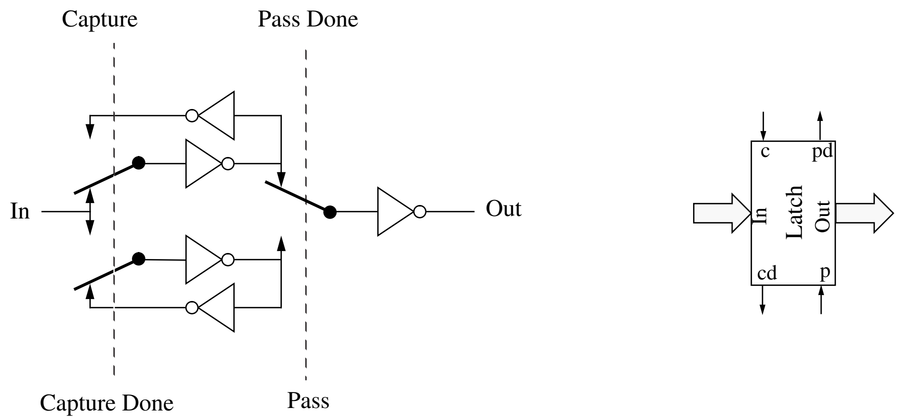

Figure 3.4 A transition-controlled latch

3.2. Storage Elements

The building blocks above are sufficient for handling control signals under a transition signalled protocol, but to construct a complete self-timed logic system some storage elements are required. A transition controlled latch can be used to provide state storage. Figure 3.4 shows how such a latch might be constructed.

The latch is normally transparent. When a transition occurs on the Capture input, the latch holds its current value. When a transition occurs on the Pass input, the latch becomes transparent again. The Capture Done and Pass Done outputs provide suitably delayed, amplified versions of the Capture and Pass inputs, indicating that the latch has performed the required action.

Although this latch works well, its physical implementation can be somewhat complex, slow, or bulky. With some care, standard gated latches can also be used for storage [18,29].

3.3. Micropipelines

The two-phase bundled data protocol offers a number of advantages in its simplicity and composability. Using this protocol, Ivan Sutherland described an elegant methodology for constructing self-timed systems, known as *micropipelines* [39]. Micropipelines are self-timed, event driven, elastic pipelines composed of the elements just described, that may or may not contain processing between the pipe stages. If no processing is done between the pipe stages, the micropipeline reduces to a simple first-in first-out (FIFO) buffer.

A block diagram of a generic micropipeline is shown in Figure 3.5. It consists of three parts: a control network consisting of one C-element per micropipeline stage, an event-controlled latch in each stage, and possibly some combinational logic between the stages. The logic can signal its own completion (Stage One), or it can be simulated with a known delay (Stage Two). If no processing is present between the stages, the pipeline becomes a simple FIFO (Stage Three).

The C-elements control the action of the micropipeline. One input of each C-element is inverted. Thus, assuming that all the control signals start low, the leftmost C-element will produce a transition to the leftmost latch when the incoming request (Req In) line first makes a transition from low to high. The acknowledge from the latch will produce a similar request through the next C-element to the right. Meanwhile, the leftmost C-element will not produce another request to the leftmost latch until there are transitions both on Req In (signaling that there are more data to be accepted) and the Capture Done from the next latch to the right (signaling that the next stage has finished with the current data). Each pipe stage acts as a concurrent process that will accept new data provided that the previous stage has data to give and the next stage is finished with the data currently held. The Fred processor design is based on this micropipeline approach.

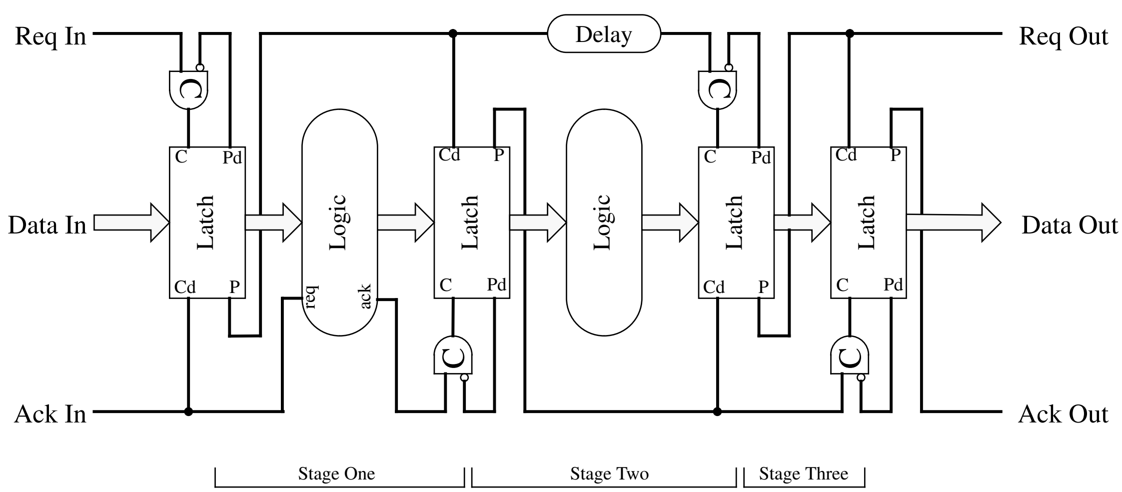

Figure 3.5 A micropipeline FIFO buffer

CHAPTER 4

A SELF-TIMED ARCHITECTURE

Experience has shown the difficulty of writing parallel programs, yet most sequential programs contain an (arguably) significant amount of instruction-level parallelism [28,43].[2] The most commercially successful approach to exploiting this parallelism is the *superscalar* architecture, which executes two or more independent instructions in parallel. Superscalar implementations provide dynamically scheduled instruction issue, often combined with out-of-order completion.

Another effective approach lies in decoupling the memory access portion of an instruction stream from the execution portion [19,45]. The two operations are scheduled statically but are allowed to move in and out of phase dynamically. In this manner, peaks and valleys in each may be smoothed for an overall performance gain. It has been shown [15] that the decoupled approach can provide a limited version of register renaming, out-of-order completion, and dynamic loop unrolling, resulting in performance which equals or surpasses a superscalar implementation. Additionally, the decoupled approach requires significantly less hardware support.

Although most decoupled architectures have been proposed and built using a traditional synchronous design style, a self-timed approach seems to offer many advantages. Typically the independent components of the machine are decoupled through a FIFO queue of some sort. As long as the machine components are all subject to the same system clock, connecting the components through the FIFOs is subject to only the usual problems of clock skew and distribution. If, however, the components are running at different rates or on separate clocks, the FIFO must serve as a synchronizing element and thus presents more serious problems.

Self-timed implementation therefore seems to be a natural match for decoupled computer architectures. The micropipeline approach is based on simple, elastic, self-timed FIFO queues, which suggests that decoupled computer architectures may be implemented much more easily in a self-timed micropipeline form than with a clocked design. Because the FIFOs are self-timed, synchronization of the decoupled elements is handled naturally as a part of the FIFO communication. The elastic nature of a micropipeline FIFO allows the decoupled units to run at data-dependent speeds, producing or consuming data as fast as possible for the given program and data. Because the data are passed around in self-timed FIFO queues and the decoupled processing elements are running at their own rate, the degree of decoupling is increased in this type of system organization without the overhead of a global controller to keep track of the state of the decoupled components.

4.1. Overview of Fred

The Fred architecture is roughly based on the NSR architecture developed at the University of Utah [32,4]. As such it consists of several decoupled independent processes connected by FIFO queues of various lengths, an approach which could potentially offer a number of advantages over a clocked synchronous organization. Unlike the NSR, Fred provides features necessary and desirable in any microprocessor used to build a modern general purpose computer system, including

---

[2]Nicolau claims there is lots of parallelism available. Wall claims there is some, but not much.

wider data paths and memory addressing, a large register file, exception handling, and security protection.

Multiple independent functional units allow several instructions to be in progress at a given time. Because the machine organization is self-timed, the functional units may take as long or short a time as necessary to complete their function. One of the performance advantages of a self-timed organization is directly related to this ability to finish an instruction as soon as possible, without waiting for the next discrete clock cycle. It also allows the machine to be upgraded incrementally by replacing functional units with higher performance circuits after the machine is built with no global consequences or retiming. The performance benefits of the improved circuits are realized by having the acknowledgment produced more quickly; thus the instruction that uses that circuit finishes faster.

The basic Fred architecture suggests the instruction set and the general layout and behavior of the processor. Extensions to the Fred architecture may be made. New instructions may be added, and additional functional units may be incorporated. The existing functional units may be rearranged, combined, or replaced. Often, references are made to 32-bit words, operations, and memory addressing, but this is only for convenience. If more bits are needed for operations such as double-precision floating point, the implementation can be revised either to make all internal data paths and registers wider or to transfer the necessary operands with multiple transactions. The details of the exception handling mechanism are not specified by the architecture, but some means must be provided.

The architecture just described merely provides a framework for further investigation. The remainder of this document will describe aspects of a specific implementation of the Fred architecture, and these details will be addressed. A solution to the exception handling problem will be discussed in Chapter 5.

4.2. VHDL Model

A prototype of Fred has been implemented as a detailed VHDL model to investigate the performance[3] and behavior of the Fred architecture under varying conditions. Figure 4.1 shows the overall organization. Each box in the figure is a self-timed process communicating via dedicated data paths rather than buses. Each of these data paths (shown as wires in Figure 4.1) may be pipelined to any desired depth without affecting the results of the computation. Because Fred uses self-timed micropipelines in which pipeline stages communicate locally only with neighboring stages in order to pass data, there is no extra control circuitry involved in adding additional pipeline stages. Because buses are not used, their corresponding resource contention is avoided.

The VHDL model chooses particular implementations for each of the main pieces of Fred. For example, the Dispatch Unit is organized so as to dynamically schedule instruction issue and to allow out-of-order completion. This is of particular interest in a self-timed processor where the multiple functional units might take varying amounts of time to compute their results, thus leading naturally to out-of-order instruction dispatch and/or completion. An individual functional unit might even take different amounts of time to compute a result, depending on the input data. The VHDL prototype is fully operational in all aspects, including dynamic scheduling, out-of-order instruction completion, and a functionally precise exception model. The timing and configuration parameters can be adjusted for each component of the design.

---

[3]Fred's performance is obviously measured in "Fhlintstones."

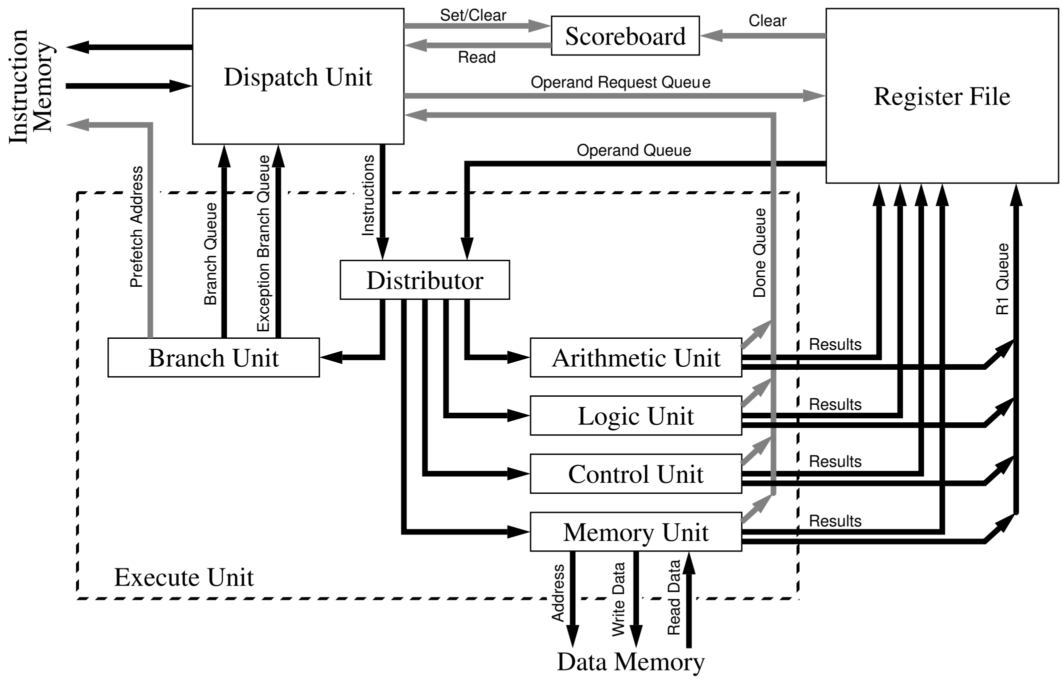

Figure 4.1 Fred block diagram

The Powerview environment from Viewlogic provides a way of combining gate-level circuit designs with VHDL models for a composite model of digital systems. The simulator is developed using this tool, which lets the simulation run in a graphical environment in which signals and processes can be traced, stopped, restarted, and plotted. However, the original Viewlogic VHDL implementation provided only a small subset of the IEEE VHDL standard, which restricts the ease with which the models are developed and modified.[4] Later VHDL releases from Viewlogic implement most of the IEEE standard language.

The major functional units of Fred are written in behavioral VHDL, whereas the connecting FIFOs and glue logic are specified using discrete circuit components. This does not affect the functionality or correctness of the model to any great degree, and it allows the model to be developed with a reasonable amount of time and effort. For example, well-known implementations of integer multiplication circuits exist. It is much easier to say

```
C <= A * B after delay_time;
```

than to spend several days developing an equivalent implementation with discrete logic. The delay times for each operation were carefully chosen to mimic typical gate delays found in existing processor designs. Most of the self-timed circuit component library used for the glue logic was developed by Erik Brunvand [2] for use with Actel FPGAs.

---

[4]A strong hypobaric forcing function is inherent to Viewlogic's VHDL implementation.

4.3. Instruction Set

Choosing an instruction set for a RISC processor can be a complex task [21,20,23]. Rather than attempt to design a new instruction set from scratch, much of the Fred instruction set was taken directly from the Motorola 88100 instruction set [26]. However, Fred does not implement all of the 88100 instructions, and several of Fred's instructions do not correspond to any instructions of the 88100. Fred's instruction format is triadic, where most instructions specify two source registers as operands and one destination register for the result. The destination register is specified first. For example, the instruction **`sub r2,r3,r4`** would load register **`r2`** with the value of register **`r3`** minus the value of register **`r4`**, while the instruction **`ld r2,r3,r4`** would load register **`r2`** with the memory at location **`r3`**+**`r4`**. However, the instruction **`st r2,r3,r4`** would write the value of register **`r2`** to the memory at location **`r3`**+**`r4`**. A brief summary of the Fred instructions is shown in Table 4.1. A complete opcode listing is found in Appendix A.

4.4. Decoupled Operations

Two of Fred's FIFO queues are of particular interest, as they may be used for decoupling specific operations through software control. The R1 Queue allows data transfers to be decoupled, whereas the Branch Queue provides efficient prefetching for changes in program flow.

Deadlocking the processor is theoretically possible. Because both the R1 Queue and Branch Queue are filled and emptied via two separate instructions, it is possible to issue an incorrect number of these instructions so that the producer/consumer relationship of the queues is violated. Fred's dispatch logic will detect these cases and force an exception before an instruction sequence is issued that would result in deadlock.

4.4.1. R1 Queue

There are 32 general registers in the Fred architecture. Registers **`r2`** through **`r31`** are normal general-purpose registers, but **`r0`** and **`r1`** have special meaning. Register **`r0`** may be used as the destination of an instruction but will always contain zero. Register **`r1`** is not really a register at all but provides read access to the R1 Queue, a data pipeline similar to that used in the WM machine [45]. Specifying **`r1`** as the destination of an instruction inserts the result into the pipeline. Each use of **`r1`** as a source for an instruction retrieves one word from the R1 Queue. For example, the instruction **`add r2,r1,r1`** would fetch two words from the R1 Queue, add them together, and place the sum in register **`r2`**. Likewise, assuming that sequential access to register **`r1`** would result in values *A*, *B*, and *C*, the instruction **`st r1,r1,r1`** would write the value *C* into memory location *A*+*B*. The program receives different information each time it performs a read access on register **`r1`**, thus achieving a form of register renaming directly in the R1 Queue. Instructions that utilize the R1 Queue are forced to issue and complete in program order to provide deterministic behavior.

Although data from any of the functional units may be queued into the R1 Queue, loads from memory are most likely to benefit from being so queued. It may be possible to subsume some of the memory latency by queuing loaded data in the R1 Queue in advance of its use. Such access is not always required but is possible when desired.

The R1 Queue is not necessarily an integral part of the Fred architecture. The early development of the Fred architecture was influenced by previous work on the NSR processor [4,32], which completely decouples all data memory operations by means of special pipelines. Although

Table 4.1 Fred instruction set

Mnemonic	Function
add	Arithmetic sum
addu	Arithmetic sum (unsigned)
and	Logical AND
bb0	Branch if a particular bit is clear
bb1	Branch if a particular bit is set
beq	Branch if a register is equal to zero
bge	Branch if a register is greater than or equal to zero
bgt	Branch if a register is greater than zero
ble	Branch if a register is less than or equal to zero
blt	Branch if a register is less than zero
bne	Branch if a register is not equal to zero
br	Branch unconditionally
clr	Clear a range of bits from a register value
cmp	Compute a set of result bits, based on a comparison
div	Arithmetic division
divu	Arithmetic division (unsigned)
doit	Consume a previously computed branch target
ext	Extract a sign-extended bitfield from a register
extu	Extract an unsigned bitfield from a register
ff0	Find the first zero bit in a register
ff1	Find the first one bit in a register
getcr	Read a control register
ld	Load data from memory
lda	Compute an effective memory address
ldbr	Place a specific target value into the Branch Queue
mak	Load a specified pattern into a bit field
mask	Immediate form of logical AND, with upper 16 bits cleared
mul	Arithmetic multiplication
mvbr	Remove a target from the Branch Queue
mvpc	Compute a given offset from the Program Counter
or	Logical OR
putcr	Write a control register
rot	Logically rotate bits in a register
rte	Return from exception
set	Set a range of bits from a register value
st	Store data to memory (this has three source operands)
sub	Arithmetic difference
subu	Arithmetic difference (unsigned)
sync	Synchronize the processor
trap	Invoke a software exception
xmem	Atomically exchange a register value with a memory location
xor	Logical exclusive OR

detailed investigation of the effectiveness of such a technique is beyond the scope of this research, the additional complexity involved in handling exceptions with the R1 Queue is intriguing enough to warrant its inclusion in the Fred architecture. The exception model is discussed in Chapter 5.

4.4.2. Branch Queue

Fred's branch instructions are also decoupled. Each branch operation is broken into an *address generating* part and a *sequence change* part. The instructions for both absolute and relative branches compute a 32-bit value which will replace the program counter if the branch is taken, but the branch is not taken immediately. Instead, the branch target value is computed by the Branch Unit and passed through the Branch Queue back to the Dispatch Unit, along with a condition bit indicating whether the branch should be taken or not. These data are consumed by the Dispatch Unit when a subsequent **doit** instruction is encountered, and the branch is either taken or not taken at that time. Although this action is similar to the synchronous concept of *squashing* instructions, Fred does not convert the **doit** instructions into NO-OPs but instead removes them completely from the main processor pipeline.

Any number of instructions (including zero) may be placed between the branch target computation and the **doit** instruction. From the programmer's view, these instructions do not have to be common to both branches nor must they be undone if the branch goes in an unexpected way. The only requirement for these instructions is that they not be needed to determine the direction of the branch. The branch instruction can be placed in the current block as soon as it is possible to compute the direction, whereas the **doit** instruction should come only when the branch must be taken. Since the direction and target of the branch is known as soon as the first part is complete, the maximum possible time is available for prefetching the new instruction stream. To prevent unnecessary code expansion, setting bit 31 in the opcode of an instruction indicates that an implicit **doit** instruction should immediately follow the current instruction, removing the need to explicitly encode one. This is indicated by appending a "**.d**" suffix to the instruction mnemonic.

4.4.3. Prefetching

If the branch target computation and the corresponding **doit** instruction are separated by several instructions, it provides an opportunity for accurate and early prefetching. When a branch instruction is executed, the branch target value is computed by the Branch Unit and passed through the Branch Queue back to the Dispatch Unit, along with a condition bit indicating whether the branch should be taken or not. These data remain in the Branch Queue and are not used or consumed until the corresponding **doit** is encountered. There are two ways in which this branch decoupling allows effective prefetching. First, the **doit** instruction does not have to be consumed in program order but instead can be executed as soon as the branch data reaches the head of the Branch Queue, provided that the **doit** has been recognized by the Dispatch Unit. This reduces some of the branch latency, because instructions from the new instruction stream can then be requested as soon as possible. However, in order to fetch instructions from the new stream the **doit** instruction must be fetched by the Dispatch Unit, and if there are several instructions between the branch and the **doit,** all of those instructions must be fetched by the Dispatch Unit before the **doit** can be seen.

A second opportunity for reducing branch latency takes place in the Branch Unit. Even though the **doit** can be consumed out of order, it does not alleviate the latency involved in taking a branch to a location which is not in the instruction cache. However, the direction and destination of the branch are known as soon as they are computed by the Branch Unit. By passing this information off-chip to an intelligent cache controller, it is possible to preload the appropriate instructions.

Although the detailed design of an external cache system for the Fred architecture is beyond the current scope of this research, some speculation is useful. In addition to the standard *normal* cache implementation there could be a *preload* cache, containing a small number of *preload* cache lines reserved specifically for loading nonsequential instructions. The cache controller would fill the normal cache one line at a time, allowing effective cache operation as long as the instruction stream consists of sequential instructions. When the Branch Unit computes a new target address, it would pass it off-chip to the cache controller. The cache controller can then use the new target address to fill the preload cache lines with the instructions from this target. When the **doit** is consumed, Fred's normal instruction fetch process would request instructions from the new target. The cache controller would then copy the normal cache lines from the preload cache lines, instead of loading them from memory.

There are several facets to this mechanism:

1. Taken branches are more interesting, since nontaken branches simply continue the sequential instruction stream. However, both types could benefit from the preload cache.
2. The preload cache would have to be loaded in parallel with the normal cache, since the normal cache might also need to load lines before the **doit** is consumed.
3. The preload cache can simply give up in the event of memory faults and let the normal cache handle those in sequence.
4. If the target address is already in the normal cache, there is no need to use the preload cache at all.
5. If the preload cache is invalid (either it has detected a fault, or it is not being used for nontaken branches), the normal cache just loads the new target from memory. If the preload cache has not finished loading when the normal cache needs it, the normal cache should wait rather than reissuing the same load requests.
6. The preload cache probably only needs to contain one or two lines, since the normal cache will take over as soon as the first preload line is used.
7. The preload cache does not need any replacement strategy. Because Fred does not issue speculative branches, only one branch/**doit** pair will appear in the Dispatch Unit at a given time. Once the **doit** is consumed, the preload cache is copied into the normal cache and invalidated.

4.5. Execute Unit

There are five independent functional units in the prototype implementation of Fred: Logic, Arithmetic, Memory, Branch, Control. Each functional unit is responsible for a particular type of instruction, as shown in Table 4.2. These functional units plus the Distributor collectively make up the Execute Unit. The Distributor is responsible for routing instructions to their proper functional unit. It takes incoming instructions and operands, matches them up where needed, and routes instructions to appropriate functional units. It also provides the zero value for **`r0`** accesses. Instructions pass through the Distributor sequentially but may complete in any order because the pipelines are self-timed and the functional units themselves may take more or less time to execute a given instruction.

Each of the functional units may produce results that are written back to the register file directly or that reenter the register file through the R1 Queue. In addition, last result reuse may take place in each functional unit in a manner similar to that found in synchronous processors. The only difference is that in a synchronous processor the reused data will stay latched in the functional unit only until the following clock tick. In a self-timed processor, data remain latched until overwritten or invalidated. The validity of the last result latches is maintained by the Dispatch Unit with no feedback from the functional units involved. There is currently no provision for forwarding results between functional units.

The Memory Unit is treated as just another functional unit. The only difference is that the Memory Unit can produce results that are written to the data memory rather than the Register File.

4.6. Instruction Dispatch

The Dispatch Unit is, in some sense, the main control unit for the Fred processor. It is responsible for keeping track of the Program Counter, fetching new instructions, issuing instructions to the rest of the processor, and monitoring the instruction stream to watch for data hazards. Instructions are fetched and issued to the rest of the machine as quickly as possible. Because each functional unit may take a different amount of time to complete, instructions may complete in a different order from which they were issued.

A register scoreboard is used to avoid all data hazards. The scoreboard is set by the Dispatch Unit and is cleared when results arrive at the Register File. Instructions will not be dispatched until all data hazards are resolved.

An Instruction Window (IW) is used to buffer incoming instructions and to track the status of issued instructions [42]. The IW is a set of internal registers located in the Dispatch Unit which tracks the state of all current instructions. Each *slot* in the IW contains information about each instruction such as its opcode, address, current status, and various other parameters. As each instruction is fetched, it is placed into the IW. New instructions may continue to be added to the IW independently, as long as there is room for them. Justification for the IW will be found in Chapter 5, with implementation details in Chapter 6.

4.7. Register File

The Register File provides operands through a FIFO to the Execute Unit, in response to requests from the Dispatch Unit. These operands are paired with instructions by the Distributor

Table 4.2 Fred instruction set execution

Dispatch Unit	Execute Unit				
	Logic	Arithmetic	Memory	Branch	Control
doit, rte, sync, trap, *illegal*	and, clr, ext, extu, ff0, ff1, mak, mask, or, rot, set, xor	add, addu, cmp, div, divu, mul, sub, subu	ld, lda, st, xmem	blt, ble, bne, beq, bge, bgt, bb0, bb1, br, ldbr	getcr, mvbr, mvpc, putcr

and passed to the appropriate functional unit. Because operands are requested concurrently with instruction issue, there is no matching required to determine which operands should be paired with which instructions. Operands and instructions emerge from the FIFO queues in the correct sequence.

On the incoming side, the Register File accepts results from each functional unit that produces data. These results are accepted independently from each functional unit and are not multiplexed onto a common bus. Data hazards are prevented by the scoreboard and the Dispatch Unit, which will not issue an instruction until all its data dependencies are satisfied, so there will never be conflicts for a register destination. The Register File clears the associated scoreboard bit when results arrive at a particular register. Instruction results may also be written into the R1 Queue, but there is no actual register associated with it. Instead, the Dispatch Unit (not the Register File) clears the scoreboard bit for register **`r1`** when the producing instruction completes successfully. This bit is needed only to insure deterministic results.

CHAPTER 5

EXCEPTIONS

Exceptions are unforeseen events which require the processor to stop what it is doing and deal with some unexpected problem. There are three general causes for exceptions: software traps (including illegal opcodes), external interrupts, and process exceptions (such as memory faults). When exceptions occur, it is necessary for the processor to temporarily stop executing its current instruction stream and handle whatever conditions caused the exception. Often, once the exception has been dealt with, the processor must be able to resume as though no exception had occurred.

Precise exception models allow the programmer to view the processor state as though the exception occurred at a point exactly between two instructions, such that all instructions before that point have completed while all those after have not yet started. The present implementation of Fred uses a *functionally precise* model, which is not quite the same. The motivation for and explanation of this exception model are presented in this chapter.

5.1. Exception Requirements

In a heavily pipelined architecture, where instructions execute concurrently and possibly out of order, identifying a precise point for exception handling can be costly. Several methods have been developed to deal with this problem [10,36,37,44]. However, clocked systems have the advantage that the state of the processor is available to the processor's control logic at every clock cycle. In a self-timed processor like Fred, this is not the case. One characteristic of a self-timed system is that while the completion of a task is reported through a handshake of some sort, the actual completion time for that event is not particularly well-defined with respect to any global signal such as a clock. Although this may provide advantages in achieving average-case performance or simplifying modular composition [8], it makes exception processing difficult. Much of the state of the Fred processor is contained in the pipelines, but it is problematic to determine exactly how many items are in a particular pipeline at a given moment in time.

This problem has been addressed in part by the AMULET group at the University of Manchester [17,29], who have built a self-timed implementation of the ARM. However, its precise exception model is a simple one since its single ALU causes all instructions to issue and complete sequentially. Fred's decoupled concurrent architecture requires a more general solution.

5.1.1. The Instruction Window

To resolve the uncertainty regarding instruction status, Fred uses an IW, similar to that described by Torng and Day [42], to fetch and dispatch instructions. The IW is a set of internal registers located in the Dispatch Unit, which tracks the state of all current instructions. Each *slot* in the IW contains information about each instruction, such as its opcode, its address, its current status, and various other parameters. As each instruction is fetched, it is placed into the IW. New instructions may continue to be added to the IW independently, as long as there is room for them.

Instructions are scheduled dynamically and issued from the IW when all their data dependencies are satisfied. Each issued instruction is assigned a tag which uniquely distinguishes it from all other current instructions. When an instruction completes, it uses this tag to report its status to back to the Dispatch Unit. The status is usually an indication that the instruction completed successfully, but when an instruction is unsuccessful it returns an exception status to the Dispatch Unit, which then initiates exception processing. Instructions are removed from the IW only after they have completed successfully. Instructions which can never cause exceptions (such as **`xor r2,r3,r4`**) do not have to report their status and can be removed from the IW as soon as they are dispatched. The instruction completion signals are returned to the Dispatch Unit through a FIFO queue, so that the completion signal does not become a performance bottleneck.

5.1.2. Data Hazards

Data hazards are handled by the Dispatch Unit. RAW and WAR hazards are resolved by using a simple register scoreboard. When an instruction is dispatched, the Dispatch Unit marks the destination register as in use by setting the register's scoreboard flag. When the result of the instruction arrives at the register, the Register File clears the scoreboard flag for that register. The Dispatch Unit will not request operands from the Register File unless the scoreboard bit for each source register is clear, indicating that the register holds valid data.

WAW hazards are handled in the same way. An instruction will not be dispatched unless its destination register is available for writing. Instructions that write to the same destination register complete sequentially, since the second instruction will not be dispatched until the results of the first instruction arrive at the destination register and its scoreboard bit is cleared.

For instructions that write to the R1 Queue, the scoreboard bit for register **`r1`** is cleared by the Dispatch Unit when the result has been placed into the R1 Queue, as indicated by the instruction's completion status. Instructions signal completion as soon as the functional unit which processes them has generated a valid result, even though that result may not yet have reached its final destination. This allows faster sequential access to the R1 Queue, allows exceptions to be recognized earlier, and enables successful instructions to be removed from the IW sooner so that more instructions may be fetched. This early completion signaling has no effect on data hazards.

5.1.3. Out-of-Order Completion

In Torng and Day's design, provision was made to reduce interrupt latency by aborting issued instructions which would take a long time to complete [42]. In a self-timed processor there is no way to tell how soon an instruction will complete, since there are no clock cycles by which to measure progress. Instead, when an exception occurs, all outstanding instructions are allowed to either complete or fault before handling the exception.

Because instructions may complete out of order, recoverable exceptions can cause unforeseen WAW hazards. To handle the exceptions properly it is necessary for a faulting instruction to save its original operands as part of the IW status. This code fragment illustrates the problem:

```
ld   r2,r3,r4
add  r4,r5,r6
```

The instructions are issued in order. The load instruction uses sources **`r3`** and **`r4`** to compute the effective address. The add instruction then modifies register **`r4`**. This is fine, unless the load faults after the add has completed. The load cannot simply be reissued, since the original value of **`r4`** has been overwritten. Saving the operands as part of the load instruction's status allows the software to emulate the operation of the load instruction once the fault has been resolved.

It might be possible to abort some instructions involving iterative processing (such as multiply or divide) when exceptions occur. Unfortunately, matters are worse when instructions can be aborted, because all aborted instructions need some way to recover their original operands. This could be done via a history buffer or future buffer or by storing the original operands as part of the IW slot. By not aborting issued instructions, only those instructions that fault need to report their operands back to the IW as part of their status. This reduces the complexity required of the Dispatch Unit and the Register File, at the expense of widening the data path needed to report instruction status. Some alternatives are discussed in Section 6.6.

5.1.4. Memory Unit

In most cases, waiting for outstanding instructions to complete before handling exceptions does not increase the latency by a significant amount and, in fact, may reduce the latency when compared with the time needed to save aborted instructions as part of the processor state. The instructions that could make a big difference are those involving the Memory Unit. The Fred architecture does not specify a particular external memory system, but it can be assumed to include a multilevel cache system with both fast and slow memory. The interface to the external memory uses a standard self-timed request/acknowledge handshake when dispatching loads or stores. Bundled with the acknowledgment is a memory status signal used to indicate exception conditions such as write-protection violations, page faults, cache misses, and so forth. This status signal can allow the processor to take an exception in the event of page faults or even cache misses. Because decoupling is provided in the FIFO queues, split transaction memory accesses are not currently implemented.

When a memory access instruction faults, it returns the fault type and operands to the Dispatch Unit as part of its completion status. All issued instructions are allowed to complete or fault, and those that finish successfully are removed from the IW before exception processing begins. The exception-handling software can then repair the cause of the exception and emulate the memory operation, based on the operands saved in the IW. Program execution can then resume.

5.1.5. Branch Queue and R1 Queue

These two queues provide particularly difficult challenges to an exception mechanism. For example, in the case of the Branch Queue, a problem arises if the **doit** is consumed out of order in this code fragment:

```
br   _foo
add  r2,r3,r4
ld   r5,r6,r7
or   r8,r9,r10
doit
```

If the load instruction faults after the **doit** has been consumed, a traditional precise exception model would require that the state of the processor be reset to the point where the first two instructions have completed, while the next three have not been issued. In order to attain this goal, two possible approaches may be used. Either the instructions following the **ld** must wait until it has completed successfully before issuing, or some mechanism may be used to “undo” their effect when the load faults. Neither of these options is acceptable.

Stalling the following instructions is not a desirable option, since the vast majority of load instructions will *not* fault. It would impose a needless performance penalty to delay issuing instructions just to avoid a very unlikely condition.

Adding some mechanism to “undo” out-of-order instructions is the approach typically taken in synchronous processor designs. The IW is ideally suited to containing instruction information, and some sort of history buffer or reorder buffer [36] can be used to restore any register contents which were changed as a result of an out-of-order instruction. Unfortunately, this does not address all the problems. If the **doit** has completed, it means that a branch target has been removed from the Branch Queue. Because this queue is implemented as a self-timed FIFO, undoing the **doit** instruction would require putting the consumed branch target back into the queue from the wrong end. Although theoretically possible, a practical implementation is very difficult to achieve. Also, not only must items be forced back into the queue, but other items may need to be removed to undo the effect of any branch instructions which followed the faulty load.

The R1 Queue suffers from the same problems and restrictions. An out-of-order instruction that must be undone may have obtained operands from the R1 Queue. Those operands must somehow be saved in order for the R1 Queue to be reversed if necessary.

To further complicate matters, undoing instructions means that although instructions may complete out of order, they may only be removed from the IW in sequential order. This places arbitrary restrictions on how much reordering is allowed, based solely on the size of the IW and the Register File’s history buffer. Fred avoids these constraints by using a different exception model and never reversing the operation of any successful instruction. This is described briefly below. Additional details are found in Chapter 6.

5.1.6. Exception Software

When exception processing begins, the processor state includes the IW contents, the address from which the next instruction will be fetched, the Register File, and the contents of the R1 Queue and Branch Queue. Once all outstanding instructions have completed or faulted, the IW is copied to a set of Shadow IW registers visible to the programmer, then cleared. Since all successful instructions are removed from the IW when they complete, the Shadow IW contains only faulty and nonissued instructions.

This Shadow IW provides a “*functionally precise*” exception point. The exception model seen by the programmer is not that of a single point where the exception occurred. Instead, there is a “window” (hence the name) of instructions which were in progress. The hardware guarantees that this window will consist only of instructions which either faulted or had not yet issued when the exception occurred. The instructions in the Shadow IW comprise a subset of a portion of the sequential instructions of the program. The missing elements are instructions which completed successfully out of order and which do not need to be reissued.

Any instructions that have completed successfully need no further consideration. All operands for those instructions are valid, or they would not have been issued. All destination registers will have the correct values for the same reason. Any items consumed from the Branch Queue or R1 Queue have been used correctly, and neither of these queues needs to be reversed.

To allow additional exceptions or to perform a context switch, the exception software must save the state of the processor. All of the state can be obtained via control registers, except for the contents of the R1 Queue and the Branch Queue, which are not automatically flushed. However, there are control registers which keep a count of the number of items in these two queues. Instructions exist which can be used to manually flush and reload these queues. The other queues do not need special attention.

Although the R1 Queue can wait for software to save and restore its contents, the Branch Queue is needed to branch to the exception-handling code. Rather than try to flush this queue in hardware, an additional queue, the Exception Branch Queue, is used for flow control until the Branch Queue contents have been saved. The usage of this queue is controlled by a mode bit in a control register, set by the hardware when exception processing begins. Additional exceptions cannot be taken while the Exception Branch Queue is in use, because there is no way to save or recover the processor state.

Once the exception condition has been handled, the original state of the processor must be restored. Faulty instructions must be emulated in software and removed from the Shadow IW. Nonissued instructions are left in the Shadow IW. The Branch Queue and R1 Queue are reloaded (if necessary). The **rte** instruction will restore the IW from the Shadow IW, reenable exceptions, and resume fetching instructions and issuing them from the IW.

5.2. Exception Example

Figure 5.1 shows a section of a program as it may appear in the IW. At this point, the second instruction can't issue until the top instruction completes (because of the dependency on **`r2`**), and the rest must issue sequentially due to the antidependency chain. Figure 5.2 shows the state soon after the **`r2`** dependency is satisfied. The top instruction has completed and been removed, several additional instructions have been issued (one has completed), and two new instructions have been fetched and placed into the IW.

If the load instruction faults, exception processing will take place. Figure 5.3 shows the state

Tag	Status	Instruction
1	Issued	add **r2**,r2,r2
2		add r1,**r2**,**r2**
3		xor r2,r3,r3
4		mul r3,r4,r4
5		and r4,r5,r5
6		ld r5,r6,r6
7		add r6,r7,r7

Figure 5.1 IW with data dependency

Tag	Status	Instruction
2	Issued	add r1,r2,r2
3	Complete	xor r2,r3,r3
4	Issued	mul r3,r4,r4
5	Issued	and r4,r5,r5
6	Issued	ld r5,r6,r6
7		add r6,r7,r7
8		xor r7,r8,r8
9		add r8,r9,r9

Figure 5.2 IW data dependency resolved

Tag	Status	Instruction
4	Issued	mul r3,r4,r4
5	Complete	and r4,r5,r5
6	**Page Fault**	ld r5,**r6**,**r6**
7	Issued	add **r6**,r7,r7
8	Complete	xor r7,r8,r8
9	Issued	add r8,r9,r9
10		mul r9,r10,r10

Figure 5.3 IW with exception condition

Tag	Status	Instruction
4	**Overflow**	mul r3,r4,r4
6	**Page Fault**	ld r5,r6,r6
10		mul r9,r10,r10

Figure 5.4 IW ready for exception handling

of the IW when the fault is reported. Several instructions have already completed and been removed, while others are still pending. In particular the add instruction with tag 7 will modify register **`r6`**, which was used by the faulty load instruction. Figure 5.4 shows the state of the IW once all outstanding instructions have either completed or faulted. There is more than one faulty instruction now in the IW, and the IW only contains faulty and nonissued instructions, since all completed instructions have been removed. In this case, the first faulty instruction in the IW is *not* the instruction that first signaled an exception. In addition, the add instruction which modified **`r6`** has completed successfully, so the current value of **`r6`** cannot be used to reissue the load instruction. Not shown are the operands for the faulty instructions, which are included in the reported status. The cause of the first exception is saved in a special control register, since external interrupts are not associated with any particular instruction and could not otherwise be handled. Both faulty instructions can be dealt with before returning from the exception. There is no need to handle only one fault condition at a time.

The Fred processor has the ability to dynamically schedule instructions. This was not demonstrated here, since it only adds to the complexity of the example without changing the overall behavior. The out-of-order completion and its effect on the IW state is functionally identical for instructions issued out of order.

CHAPTER 6

IMPLEMENTATION DETAILS

This chapter discusses the particular details needed to understand fully the Fred architecture. Each functional unit of the VHDL implementation is written as an independent behavioral model, with only one independent process included in each model. The functionality and data structures of the VHDL prototype implementation are explained. Higher-level architectural concepts are explained as they arise, in the context of the underlying VHDL implementation.

6.1. Instruction Window

The IW is contained in the Dispatch Unit, and is used to buffer incoming instructions, to track the status of issued instructions, and to control the overall state of the processor when exceptions are detected. Each slot in the IW contains information about each instruction such as its opcode, address, current status, and various other parameters. As each instruction is fetched, it is placed into the IW. New instructions may continue to be added to the IW independently, as long as there is room for them.

6.1.1. IW Contents

The information contained in each IW slot is shown in Table 6.1. The VALID bit simply indicates whether or not the IW slot contains an instruction, and the ISSUED bit indicates whether the instruction has been dispatched or not. Each dispatched instruction has a TAG, which uniquely distinguishes it from all other currently executing instructions. There need to be only as many unique tags as there are slots in the IW. For decoding and fault recovery, the original ADDRESS and OPCODE of each instruction are required. There are some instructions which may require special treatment or affect the operation of the IW, and the WAT and SINGLE bits distinguish these instructions. The STATUS field is used to indicate the cause of failure for instructions which fault, and the ARG1 and ARG2 fields are used to recover from the fault when possible.

Table 6.1 IW slot size

Field	Bits	Meaning
VALID	1	this slot is filled
ISSUED	1	instruction has been dispatched
TAG	4	dispatched instruction tag
ADDRESS	30	instruction address
OPCODE	32	instruction opcode
WAT	1	can be issued only at the top of the IW
SINGLE	1	inhibits further instruction prefetching
STATUS	8	completion status
ARG1	32	used for fault recovery
ARG2	32	used for fault recovery

6.1.2. Special Instructions

Instructions are dispatched from the IW in a dynamically scheduled order. This causes some problems for some instruction sequences which must take place in a sequential order. Accordingly, the WAT and SINGLE flags shown in Table 6.1 are used to handle these special cases. The WAT ("Wait At Top") flag indicates that the instruction can only be dispatched from the top of the IW and that no other instructions can be issued until it completes, thus forcing internal synchronization. The SINGLE flag inhibits further instructions from being fetched into the IW until the flagged instruction completes. This ensures that any instructions which might affect the Program Counter will be executed before the Program Counter is used again. Because of this, implicit **doit**s may not be attached to any SINGLE instructions.

There are four instructions for which these flags are applicable, shown in Table 6.2. The **sync** instruction will only be executed when it is at the top of the IW and will not allow any additional instructions to issue until it is complete. This ensures that all previous instructions have completed successfully and that nothing else is in progress. The **sync.x** instruction does exactly the same thing but additionally requires a request/acknowledge handshake to be executed off-chip, so that synchronization with other processors may be achieved.

The **doit** instruction is never actually placed into an IW slot but simply sets a flag in the Dispatch Unit when it is fetched, indicating that a **doit** is pending. Once this flag is set, no additional instructions can be fetched until the pending **doit** operation has completed and the flag cleared. The **doit** will be consumed, as soon as the target data are available at the head of the Branch Queue. The single status ensures that the new PC value is always set to the correct value to use for fetching instructions. It might be possible to fetch instructions speculatively past a **doit**, but issuing instructions speculatively is much more difficult. The current implementation of Fred does no speculative operations.

The **putcr** instruction writes to control registers. It is important that this instruction both synchronize the machine and inhibit instruction fetch, because the value written to a particular control register may affect the operation of subsequent instructions or may change the address space between user-mode and supervisor-mode. Because of these hazards, the **putcr** instruction should be the only instruction in the IW when it is dispatched. The **rte** instruction has the same requirements for exactly the same reasons. This instruction is used to return from exception handling and affects all control registers as well as the entire IW. Any instructions which may be affected by control registers should report their status to ensure that they have completed before the control registers are modified.

Table 6.2 Instruction flags

Instruction	WAT	SINGLE
sync	Yes	No
doit	No	Yes
putcr	Yes	Yes
rte	Yes	Yes

6.2. Dispatch Unit

The Dispatch Unit takes action in response to several requests. Because many of the data structures internal to the Dispatch Unit (such as the IW) are shared among various tasks, arbitration is used to control access to these resources. Although at first glance this may seem to impose an unnecessary bottleneck, the alternative to a single arbiter is to individually arbitrate, lock, and release every shared resource as needed, which could well increase the complexity to a point that the overall performance suffers. This decision is best saved for a particular hardware implementation, in which the performance of the arbiter circuitry can be more accurately weighed against other circuit elements.

After each event to which the Dispatch Unit responds, the Dispatch Unit attempts to request a new instruction. A new instruction is not requested if there is a fault pending, if instruction fetch has been temporarily inhibited by the presence of certain synchronizing instructions in the IW, or if all slots in the IW are filled. The Dispatch Unit does not wait for the acknowledgment of the instruction request, since that acknowledgment is handled independently as an initiating event.

6.2.1. Control Register Access Events

All control registers are maintained in the Dispatch Unit. Requests to read and write them must always be handled, but do not need to be arbitrated. Shared data resources must normally be arbitrated to prevent simultaneous reads and writes, but the control registers are only modified when starting and ending exception processing (when all other processor activity has stopped) or by **putcr** instructions (which serialize the machine). Reading a shared resource like the control registers would normally require a four-phase handshake (request, grant, read, release), but since the register modification is controlled by other means, this handshaking is not necessary.

6.2.2. Instruction Arrival Events

All self-timed functional units require some initial event before they can take any action. The initial event for the Fred processor is the sending of the first request for an instruction to the external memory system. This event is generated by the Dispatch Unit in response to the release of the global clear signal. Following that initial request, the Dispatch Unit takes on a purely reactive role. The arrival of program instructions at the Dispatch Unit can be considered to be a self-timed “Request” event in the sense that they are events which cause further action to be taken. The arrival of a new instruction can be considered as an externally imposed event.

When a new instruction arrives, the Dispatch Unit adds it to the IW. Every instruction requires one slot in the IW, but any instruction may also include an implicit **doit** (indicated by bit 31 of the opcode), which will set the internal DOIT flag. Explicit **doit** instructions simply set this flag and are discarded without being added to the IW.

Bundled with the acknowledgment from the external memory is a memory status signal, which may indicate certain types of memory fault, such as read protection or page faults. If the memory status indicates that some sort of fault is associated with the instruction fetch, the instruction is marked as faulty, and exception processing is initiated.

Assuming no faults, the new instruction is inserted at the bottom of the IW. Very little decoding is needed now, since the instruction can be decoded when it is dispatched. In a hardware implementation, decoding theoretically could be done while the instruction is in the IW but not yet ready to be dispatched. The only decoding needed at this point is to recognize the SINGLE instructions, which temporarily inhibit further instruction fetch.

6.2.3. Instruction Dispatch Events

Because it is self-timed, the Dispatch Unit is not active all the time. Only upon receiving an external event does it begin to perform some action. One important action is to dispatch instructions for execution by the rest of the processor. There are several reasons why the Dispatch Unit might be unable to dispatch an instruction at a given time, such as a scoreboard conflict or a full queue. The Dispatch Unit continually monitors certain of its external interfaces for any changing condition which might allow instruction dispatch. When such events occur, an internal dispatch request is generated.

6.2.3.1 Dynamic scheduling

When the dispatch request is received, the Dispatch Unit attempts to dispatch an instruction from the IW. There is no guarantee that such is possible, since the dispatch request indicates only that some unspecified conditions have changed since the last attempt. The IW is examined for any instruction which can be issued. If an instruction is invalid (because it is an illegal opcode, does not have the correct permissions, or would cause deadlock), a flag is set to indicate a pending fault and nothing more is done.

Instructions are dynamically scheduled, meaning that they do not have to issue sequentially in program order. Any instruction which can satisfy its dependencies is able to issue, although preference is given to in-order dispatch when more than one instruction is ready. The rules governing out-of-order issues are not complex [1]. Briefly, an instruction cannot be issued if any of the following statements is true:

1. The destination register is used as a destination in a prior nonissued instruction.
2. The destination register is used as an operand in a prior nonissued instruction.
3. An operand register is used as a destination in a prior nonissued instruction.

If all these statements are false, the instruction can potentially be issued. The scoreboard must still be checked to see if the registers are available, since it is possible to have an instruction complete (and be removed from the IW) in advance of the scoreboard becoming clear.

There are additional constraints for some instruction classes. Arithmetic instructions which access the carry flag must issue sequentially. Branch instructions must issue in program order but can be removed from the IW as soon as they are issued. Because the effective addresses for memory operations are computed only in the Memory Unit, load operations can safely be reordered, but stores may not be reordered with respect to any other load or store. It is possible for the Memory Unit to further reorder memory operations once address aliases can be detected. This further reordering has not been implemented.

Finally, there are special rules for instructions using the R1 Queue. Conceivably, rules 2 and 3 above do not apply to register **`r1`**, since each access to this register results in a different value and the source and destination operations are decoupled. However it is much more difficult to detect deadlock conditions when these rules are relaxed (it becomes legal to place instructions which consume R1 Queue data before those which produce the data). The current Fred implementation enforces these rules for register **`r1`** as well.

Assuming an instruction passes the tests above and the scoreboard indicates no register conflicts involving either destination or source, the Operand Request Queue and Instruction Queue are probed to see if they are ready to accept more data. In the case of a **doit** or **mvbr** instruction, the Branch Queue is probed to see if data are available. A unique tag is assigned to the instruction, any operands are requested, the scoreboard is updated to mark the destination register as invalid, and the instruction is dispatched to the Execute Unit. If the instruction is one that cannot fault, it is immediately removed from the IW at this time.

6.2.3.2 IW update

The IW must be updated with all changes to the state of the issued instruction, and there are several internal counters which are updated as well. For instructions that use the Branch Queue or the R1 Queue, the current count of items in the queues is updated. A separate count is used for pending operations as well as for completed operations. For example, it is possible that an instruction which uses the R1 Queue for a source operand may be issued immediately because there are enough operands in the R1 Queue to satisfy its requirements, or it may be aborted if there are not enough operands. On the other hand, a prior instruction which places the operands in the R1 Queue may simply not yet have finished. If this prior instruction succeeds, all is well, but it if fails, issuing the second instruction would deadlock the processor. The Dispatch Unit will wait for pending results before dispatching a new instruction under this sort of condition.

A few instructions are not dispatched to the rest of the processor but are executed and consumed entirely within the Dispatch Unit. Except for the status codes, **trap** instructions are identical to illegal opcodes and simply initiate exception handling. The **sync** and **sync.x** instructions allow the processor to synchronize its state internally or in conjunction with some external hardware, respectively.

The **rte** instruction performs the return-from-exception processing. It clears the IW and reloads it from the Shadow IW registers, restores the Program Counter and certain other control registers from their saved values, and finally continues fetching instructions and dispatching from the new IW.

The **doit** instruction is consumed entirely from within the Dispatch Unit also, but with one important difference. The **doit** normally inhibits further instruction fetch, so only one **doit** can appear in the IW at a time. As mentioned earlier, no IW slot is used for the **doit** instruction, but a DOIT flag is set instead. As soon as the branch target data appears at the head of the Branch Queue, the **doit** can be consumed and the Program Counter updated. This does not affect normal program operation in any way, but it allows the processor to fetch instructions sooner, thus increasing the performance.

6.2.3.3 Last result reuse

Each time a functional unit sends a result to a destination register other than **`r0`** or **`r1`**, it keeps an internal copy of the result. If a second instruction dispatched to the same functional unit can use this latched result, it does not need to wait for the result to be placed into the Register File and the scoreboard bit cleared. Instead, the Dispatch Unit can issue the second instruction immediately, indicating that it should obtain its operand directly from the functional unit instead of waiting to request it from the Register File.

The Dispatch Unit is responsible for remembering which functional unit has generated and latched the most recent value for every register. When a second functional unit produces a different value for the same register, the Dispatch Unit will note that the latched value in the first functional unit is no longer valid. Last-result-reuse only works within a single functional unit. Currently no provision for forwarding data between functional units exists, except by means of the Register File.

Two additional constraints exist. First, if the instruction generating a result which will be reused faults, then a second instruction cannot reuse the result. Since instructions which never fault are removed from the IW as soon as they are dispatched, this means that an instruction destined for a functional unit can reuse the previous result only if the instruction which generated the result is not in the IW.

The second constraint is that reuse can only be done when source operands are reused, and the destination register is not involved. This is unfortunate but unavoidable without greatly increasing the scoreboard complexity. This code illustrates a common case:

```
or   r2,r0,0x4321
or.u r2,r2,0x8765
```

This type of code sequence is often used to load a known 32-bit value, such as an address, into a register. Because both instructions go to the same functional unit, use the same destination register, and never fault, it is conceivable that the second instruction could be issued immediately after the first. Unfortunately, this is not the case, as illustrated by this code segment:

```
or   r2,r0,0x4321
or.u r2,r2,0x8765
addu r4,r2,1
```

The first instruction sets the scoreboard bit for register **r2** and is dispatched. The second instruction follows it immediately and also sets the same scoreboard bit. The first instruction finishes and sends its result to the Register File. As soon as the Register File gets the value for register **r2**, it clears the scoreboard bit. At this point, the third instruction requests the value for **r2**, and is dispatched. Finally, the second instruction finishes and sends its **r2** result to the Register File. The value of **r2** used as an operand for the third instruction is incorrect because the scoreboard was cleared prematurely.

The solution to this problem is to implement the scoreboard with a set of counters for each register instead of only a single bit. Set operations would increment the count, whereas clear operations decrement the count. The register value would be considered valid only when the scoreboard count is zero. This would greatly increase the size and complexity of the scoreboard unit. Additional dispatch logic would also be required to ensure that the scoreboard counter bits do not overflow due to too many last-result-reuse instructions being dispatched at once.

The AMULET processor [17] uses a scoreboarding mechanism which has a similar effect. The destination register of all pending instructions is encoded in unary form in the dispatch FIFO, and the logical OR of each bit column indicates that a register is in use. Because some instructions may be removed from Fred's IW as soon as they have issued, this technique is not directly applicable to Fred.

6.2.4. Instruction Completion Events

Because instructions may complete out of order there must be some mechanism for detecting when instructions complete and whether they have done so successfully. As each instruction completes, it reports its tag and status to the Dispatch Unit. Normally, the status indicates successful completion and the corresponding instruction is removed from the IW. If the instruction reports a faulty status, it is not removed from the IW. Instead the IW fields dealing with faulty instructions (*status*, *arg1*, *arg2*) are updated, and a pending fault is flagged. Because the result of the faulting instruction will never appear at the Register File, the Dispatch Unit must clear the scoreboard entry for the destination register. This will not cause data discrepancies, since subsequent instructions will not be issued when faults are pending.

If the destination register of the completing instruction was **r1**, the Dispatch Unit must clear the scoreboard entry for that register, and update the internal counts used to track the number of pending operations which affect the R1 Queue. Instructions that write to the R1 Queue report their completion as soon as the result is enqueued. This allows sequential writes into the R1 Queue to complete in order, as quickly as possible.

Not all instructions need to report their status. Many of Fred's instructions are incapable of faulting once they have been issued (for example, **`or r2,r3,r4`**). Instructions which can never fault do not have to report their status and are removed from the IW as soon as they are dispatched. In particular, the following conditions apply:

1. Instructions that might fault as a result of execution must report completion. This handles faults such as memory errors, arithmetic overflow, and so forth.
2. Instructions that write into the R1 Queue must report completion, in order to force serialization. Otherwise, the order of results placed into the R1 Queue is indeterminate.
3. Instructions that might affect the Program Counter must report completion. These instructions are **putcr**, **rte**, and **doit**, but only **putcr** actually completes outside the Dispatch Unit. The others are consumed entirely by the Dispatch Unit and so do not report completion as such.

The lack of the completion signal for every instruction results in a significant reduction in the number IW slots used (Section 7.6) but has no adverse effect on exception handling or program correctness. Any data hazards are resolved by the scoreboard, which relies solely on the arrival of results, not on the completion signal.

6.2.5. External Interrupt Events

External interrupts also signal a pending fault. There is no particular instruction associated with the interrupt, and the external event arbitration ensures that the interrupt is recognized only between instructions. Control registers are used to hold the status information for the interrupt, and a pending fault is signalled. Additional logic external to the Dispatch Unit ensures that additional interrupt events will not be signalled until the current one has been handled or until interrupts are reenabled through software.

6.3. Execute Unit

The Execute Unit is composed of multiple functional units, which may be further subdivided into additional subunits (although that has not been done with the current implementation). Each of these functional units has a particular group of instructions for which it is responsible. They are described in detail below.

6.3.1. Branch Unit

Flow control instructions are significantly affected by the degree of decoupling in Fred. By decoupling the branch instructions into an *address generating* part and a *sequence change* part, the ability to prefetch instructions effectively is gained. Fred does not require any special external memory system, but it can provide prefetching information which may be used by an intelligent cache or prefetch unit. This information is generated by the Branch Unit when branch target addresses are computed and is always correct.

6.3.1.1 Decoupling

The instructions for both absolute and relative branches compute a 32-bit value which will replace the program counter if the branch is taken, but the branch is not taken immediately. Instead, the branch target value is computed by the Branch Unit and passed back to the Dispatch Unit, along with a condition bit indicating whether the branch should be taken or not. These data are consumed by the Dispatch Unit when a subsequent **doit** instruction is encountered, and the branch is either taken or not taken at that time. Although this action is similar to the synchronous concept of *squashing* instructions, Fred does not convert the **doit** instructions into NO-OPs but instead removes them completely from the main processor pipeline.

Any number of instructions (including zero) may be placed between the branch target computation and the **doit** instruction. From the programmer's view, these instructions do not have to be common to both branches, nor must they be undone if the branch goes in an unexpected way. The only requirement for these instructions is that they not be needed to determine the direction of the branch. The branch instruction can be placed in the current block as soon as it is possible to compute the direction. The **doit** instruction should come only when the branch must be taken, allowing maximum time for instruction prefetching. Figure 6.1 shows two ways of ordering the same instructions. Because the **doit** is consumed entirely within the Dispatch Unit, it can take effect out of order, as soon as the branch target data are available, allowing instructions past the branch point to be loaded into the IW before the prior instructions have completed (or even issued). This lets the IW act as an instruction prefetch buffer, but it is always correct, never speculative. The explicit **doit** is not actually placed into the IW but is shown here for clarity. To avoid extra instruction fetches, the **doit** instruction can be implicitly inserted into the instruction stream by setting a bit available in the opcode of any other instruction.

This two-phase branch model allows for a variable number of "delay slots" by allowing an arbitrary number of instructions to be executed between the computation of the branch target and its use. Figure 6.2 shows an example based on the reordered code in Figure 6.1. In this case, the **doit** is encoded implicitly by setting a bit in another instruction. In either event, the **doit** does not actually take up an IW slot. Instructions may continue to be issued out of order, even with respect to the delay slot instructions. Also, the **doit** may be consumed independently of the instruction which encodes it.

The decoupled branch model also allows other interesting behaviors, such as computing several branch targets at one time and putting them in the branch queue before executing some loop code. This trick eliminates the explicit branch computation each time through the loop, providing

```
loop:
      addu     r3,r3,3
      mul      r9,r2,r3
      addu     r2,r9,2
      subu     r8,r8,1
      bgt      r8,loop
      doit
```

a. Simple ordering

```
loop:
      subu     r8,r8,1
      bgt      r8,loop
      addu     r3,r3,3
      mul      r9,r2,r3
      addu     r2,r9,2
      doit
```

b. Reordered

Figure 6.1 Two ways of ordering the same program segment a version of loop unrolling which does not require code expansion. This is not true loop unrolling since the registers are not recolored, but it could be very useful.

6.3.1.2 Branch queues

There are two branch queues which may be used to route the branch target back to the Dispatch Unit. The Exception Branch Queue is used during exception handling, since the regular Branch Queue may contain data which must not be consumed or lost while switching to the exception handling routine. Because branch instructions do not fault, the Dispatch Unit removes them from the IW as soon as they have been dispatched. The current branch queue in use is passed to the Branch Unit along with the branch instruction, so that the correct queue will be used for the resulting target data. Otherwise, an exception occurring after the dispatch of the branch instruction but before its completion could cause the target data to be placed in the wrong queue.

This is only a potential problem because the branch instructions never fault. Memory instructions can safely use the global value of the Supervisor mode bit to determine the correct address space. Because memory instructions may fault, the Dispatch Unit will not begin exception handling until all outstanding memory instructions have completed, and so the Supervisor mode bit will always be correct as seen by the Memory Unit.

6.3.2. Arithmetic and Logic Units

These two functional units handle the logical and integer arithmetic instructions for 32-bit quantities. The Logic Unit is not particularly interesting, since it just implements the standard bit operations, and cannot generate any faults. A barrel shifter probably would be used implement the bitfield instructions. The Arithmetic Unit can generate faults, due to either an arithmetic overflow or an integer divide by zero condition. The current implementation of Fred has only one each of these units, but other units could be added.

As with most RISC processors, Fred does not have a Flag Register to hold condition bits. Any test conditions are generated explicitly by the **cmp** instruction, and the results are placed in a general register. However, a carry flag is required for arithmetic operations. This carry flag is maintained internally by the Arithmetic Unit and is not accessible directly. All arithmetic instructions must explicitly request to read or write the carry flag. In order to save, set, or clear the carry flag,

Tag	Status	Instruction	Loop #
1	Issued	subu r8,r8,1	1
2	-	bgt r8,loop	1
3	Issued	addu r3,r3,3	1
4	-	mul r9,r2,r3	1
5	-	addu**.d** r2,r9,2	1

a. Branch target not yet available

Tag	Status	Instruction	Loop #
4	Issued	mul r9,r2,r3	1
5	-	addu r2,r9,2	1
6	Issued	subu r8,r8,1	2
7	-	bgt r8,loop	2
8	Issued	addu r3,r3,3	2
9	-	mul r9,r2,r3	2
10	-	addu**.d** r2,r9,r2	2

b. Branch target consumed

Figure 6.2 Branch prefetching in the IW

Table 6.3 Carry flag manipulation

Operation	Instruction sequence
Set Carry	`sub.o r0,r0,r0`
Clear Carry	`add.o r0,r0,r0`
Read Carry	`add.i r2,r0,r0`

specific arithmetic instructions are used, as shown in Table 6.3.

Fred's architectural description allows additional functional units to be implemented. Because the carry flag is currently maintained within a single Arithmetic Unit, adding additional Arithmetic Units requires special consideration. If a single carry flag is to be shared among these units, any instructions which use the carry flag must be executed in sequence. If any instruction could be executed on any unit, this would require additional hardware in the form of a global register for the carry flag.

As an alternative, instead of sharing a single carry flag among several Arithmetic Units or forcing all carry flag instructions to use a single unit, each Arithmetic Unit could have its own carry flag. There is enough room in the opcode space to allow each arithmetic instruction which uses the carry flag to specify the individual Arithmetic Unit that should be used to execute it, up to a maximum of eight units. In this fashion, multiple threads of carry flag use could be in progress simultaneously. A modification to the compiler algorithms would be required to schedule this sort of instruction use appropriately.

6.3.3. Control Unit

The Control Unit handles those few instructions that do not fit anywhere else. All access to Fred's control registers takes place in this unit. In addition, the Control Unit handles the **mvbr** and **mvpc** instructions. The **mvbr** instruction provides a way of removing branch target data from the Branch Queue without affecting the program counter and is most commonly used by exception handling routines. The result of the instruction is a 30-bit word address and a 1-bit condition code indicating whether the branch should be taken. The inverse operation (adding to the Branch Queue) is done by the **ldbr** instruction, handled by the Branch Unit. Because both of these instructions affect the Branch Queue, they are subject to the same deadlock-avoidance controls as normal branch and **doit** instructions.

The **mvpc** instruction is Fred's version of a subroutine call. Calling a subroutine requires that the return address be somehow saved. In many processors this is done automatically by the hardware, placing the return address either on the stack or in a dedicated register. Placing the return address on the stack is generally accepted to imply a significant penalty, since it requires a memory access even when calling a leaf subroutine. However, placing the return address into a dedicated registers adds an unacceptable amount of extra complexity to the Fred architecture.

A dedicated register could be used to hold the return address. If the register is a control register that might be acceptable, but because of the way that control registers are accessed it would mean that additional resource locking would be required (Section 6.2.1). If a general register were used, the question of when to write the return address is a difficult one. The return address is known when the original branch instruction is used to generate the target address for the subroutine, but it cannot be written into a register at that time, since the instructions between the branch

```
br       _main               ; go to program entry point
or.u     r31,r0,0xFFF0       ; initialize stack pointer
mvpc.d   r28,_exit           ; return directly to exit()
```

Figure 6.4 Forcing cleanup after a subroutine call

and the **doit** may make use of that register. If the subroutine call is not taken, then the register use is blocked unnecessarily until the **doit** is consumed. If the return address is written when the **doit** is executed, then early execution of the **doit** instruction must be prevented to avoid the same problem. Also to be considered is where to store the return address until it can be written, since the address is known only by the Dispatch Unit, not the Branch Unit. Additionally, what to do when more than one branch target is in the Branch Queue is an even bigger problem. In short, it is a mess.

Instead of all this, the **mvpc** instruction is used. This instruction adds a 16-bit signed offset to the current PC value (passed from the Dispatch Unit as part of the instruction), and places the result in the destination register. A typical subroutine call sequence is shown in Figure 6.3. The PC offset allows for flexibility in the value of the return address. For example, it may be used to make cleanup code conditional on actually calling (or not calling) the subroutine, as in Figure 6.4.

6.3.4. Memory Unit

Currently, there is only one Memory Unit, which handles all data memory access. Although many memory instructions may be enqueued, the Memory Unit currently handles only one complete operation at a time, which makes hazard avoidance implicitly simple. If a memory operation faults, the status code returned to the Dispatch Unit indicates the reason for the fault. In addition, the operands needed to emulate the faulting instruction in software are also returned. For a **ld** instruction, only the effective address is needed. For **st** and **xmem** instructions both the address and the data to be written are required.

The memory unit can access either supervisor or user memory space, specified by an additional output line. Under normal conditions the global value of the Supervisor mode bit (in control register 0) is used to determine which space to use. Some supervisor-mode instruction can explicitly override this bit to access user space. There is no danger that the Supervisor mode bit will be incorrect for a memory operation, since that bit is only changed by the Dispatch Unit after all outstanding instructions have completed. Because memory instructions might fault, they will always report their status to the Dispatch Unit; thus their completion status will be required.

Most of the functional units which make up the Execute Unit do not bother to compute any results when the destination register is register **`r0`**. This register is hardwired to a zero value, so

```
br       _printf             ; compute target
or.u     r2,r0,hi16(_string) ; string address is 1st argunent
or       r2,r2,lo16(_string)
add      r3,r12,r13          ; pass 2nd argument
mvpc.d   r28,.+4             ; save return address and doit
```

Figure 6.3 Subroutine call sequence

writing to it is simply a waste of time and effort, which in a self-timed system may be important. However, although the Memory Unit will not send any results to register **`r0`**, it still performs the requested memory operation. This allows software to touch a cache line by reading a memory value into register **`r0`**.

6.4. Register File

The Register File holds the values of the general purpose registers. Operands are provided to instructions as requested by the Dispatch Unit and results from completing instructions are collected.

6.4.1. Results

The results from completing instructions arrive through individual queues directly from each functional unit to the Register File. Each result is tagged with its destination, so that the result may be routed into the correct register. Because the registers are scoreboarded by the Dispatch Unit, there can never be multiple results destined for the same register, so there is no contention involved in writing to a particular register. Combining results into a single queue would add unnecessary contention, so there are separate queues from each functional unit.

6.4.2. Scoreboard

As results are received, a signal is sent from the Register File to the Scoreboard to clear the flag for the destination register, thus allowing the Dispatch Unit to issue additional instructions which may be waiting on the result. The Scoreboard is shared between the Dispatch Unit and the Register File, so its use must be arbitrated. The Dispatch Unit must first lock the Scoreboard in order to read it, but writing to the Scoreboard is a single operation, arbitrated with a dedicated circuit. Unlike most of Fred's circuitry, the Dispatch Unit's interface to the Scoreboard cannot be pipelined. If the Dispatch Unit sets a bit in the Scoreboard and then tries to read it, a pipelined interface might allow these two operations to reach the Scoreboard out of order, and the incorrect value would be read. The "clear" signal from the Register File can be pipelined, but there is unlikely to be much point to doing so.

6.4.3. Sources

Source operands are provided to the Execute Unit by the Register File, in response to requests from the Dispatch Unit. Requests are made only when the register contents are valid, so there is no need for the Register File to examine the Scoreboard before placing the requested register contents on the Operand Queue. The Register File will remove items from the R1 Queue and place them into the correct field of the Operand Queue but otherwise has no interaction with the R1 Queue.

6.5. Exception Software

When exceptions occur, software exception handling routines are invoked to determine the cause and, if possible, to fix the problem so that program execution can continue. Before the exception handler is invoked, the processor saves just enough of the state so that merely executing the handler will not destroy crucial information. The remainder of the processor state must be saved and managed by the exception software. The exception handlers are executed in supervisor mode.

6.5.1. Exception Vectors

Fred uses a vector table to select the appropriate exception handling routine. Each exception type or trap instruction has an 8-bit number used to indicate the memory address where the entry point to the exception handing routine begins. The addresses are memory locations **`0x00000000`** through **`0x00000400`**, corresponding to exception types 0 through 255. There is only one memory word available for each vector, but that is enough to branch to the actual handler entry. The exception types corresponding to each vector are shown in Table 6.4. Vectors 0-31 are dedicated to hardware exceptions, although not all have been assigned. Vectors 32-127 are reserved for supervisor-mode traps, whereas vectors 128-255 may be used by user-mode traps. Although user programs may issue trap instructions to vectors 128-255, the exception handler will be executed in supervisor mode, so the exception handler should not be written by the user. If a user program attempts a trap to vectors 0-127, the Illegal Instruction vector is taken instead.

6.5.2. Control Registers

Two sets of control registers accessible by supervisor code (no control register access is available in user mode) provide a means of accessing and changing the state of the processor. The primary control registers are detailed in Table 6.5. In addition to these registers, the Shadow IW is accessible by control registers **`c100-c179`**. There are 16 slots in the IW for the Fred prototype implementation. Each slot in the Shadow IW is accessible via five control registers, for a total of 80 control registers. The meaning of each set of five registers is shown in Table 6.6. Each set of registers is repeated for each slot in the Shadow IW.

6.5.2.1 Exception Control Register

The Exception Control Register (ECR) affects the operation of the processor. Each bit within

Table 6.4 Exception vectors

Vector	Exception Type
0	Power-On Reset
1	Instruction Memory Fault
2	Data Memory Fault
3	Illegal Instruction
4	Unrecoverable Error
5	Interrupt
6	Deadlock
7	Integer Overflow
8	Integer Divide
9-31	future system exceptions
32-127	supervisor traps
128-255	user traps

this register enables or disables a certain behavior, as shown in Table 6.7. The Imem and Dmem bits refer to the 4-bit memory status signals returned from external memory operations. These may be used to accept or mask page faults, protection violations, or other memory problems. The exact cause of the fault should be obtained from the external memory system. For the Fred prototype only one of these bits is used, to indicate accesses to nonexistent memory.

Table 6.5 Primary control registers

Register	Name	Usage
0	Exception Control	Flags which affect exception operation
1	Saved Exception Control	Value of ECR before the exception
2	Fault Status	Indicates exception cause
3	Fault Address	Address of faulting instruction
4	Next PC	Address of next instruction
5	Branch Queue Count	Number of items in the Branch Queue
6	R1 Queue Count	Number of items in the R1 Queue
7	IW Count	Number of full slots in the IW
8	IW Fault Count	Number of faulty instructions in the IW
9-12	c9-c12	Temporary storage for supervisor mode

Table 6.6 Shadow IW control registers

Slot Register	Name	Usage
0	Status	Current instruction status
1	Address	Instruction address
2	Opcode	Instruction opcode
3	Arg1	For exception recovery
4	Arg2	For exception recovery

Table 6.7 ECR bits

Bit Field	Usage
31-12	unused
11-8	Dmem fault enable
7	DOIT flag
6	Exception enable
5	Interrupt enable
4	Supervisor mode
3-0	Imem fault enable

Table 6.8 Instruction and exception status codes

Major Value	Meaning
0	No status
1	Completed successfully
4	Imem fault
8	Dmem fault
9	Misaligned data access
12	Unimplemented opcode
13	User mode violation
16	Unrecoverable error
20	External interrupt
21	Sync.x aborted externally
24	Deadlock on Branch Queue empty
25	Deadlock on Branch Queue full
26	Deadlock on R1 Queue empty
27	Deadlock on R1 Queue full
28	Integer overflow
32	Integer divide error

6.5.2.2 Status codes

The status codes found in the Shadow IW registers and in the Fault Status register use the same set of values to encode both the instruction status and the cause of a particular exception. For exception handling, the status values consist of a 16-bit major value indicating which exception vector to take and a 16-bit minor value providing further details concerning the exact fault condition (usually dependent on the functional unit). Only bits 7-0 for each field are actually returned from the functional units, but bit 8 of the major status will be set if the vector was called as the result of a **trap** instruction. The implemented major values are shown in Table 6.8.

The minor values for Dmem faults and misaligned data accesses indicate the type (load, store, xmem) and size (byte, halfword, word) of the operation attempted. The minor values for integer overflow indicate the operation type (add, subtract, divide). Other status codes do not have specific minor values. The minor status information could be obtained from the instruction opcode, but providing it here eliminates the need for disassembly by the exception handler.

6.5.3. Example Code

The simplest exception handling routine just eliminates the offending instruction and returns. This means that the correct operation of the program in question is not guaranteed, but the program can at least continue running. Figure 6.5 shows an exception handler that performs this operation.

When the **rte** is executed, the Shadow IW contents are copied directly into the IW. Control register **`c7`** indicates how many slots to fill, but there is no compression of invalid or empty slots so all the specified slots must contain valid instructions. Because the exception handler needs to remove the faulty instructions from the Shadow IW while leaving the nonissued instructions, one

```
__stompem:
      putcr    c9,r2          ; free up some registers to use
      putcr    c10,r3

      getcr    r2,c8          ; how many faults are there?
      or       r3,r0,102      ; point to first opcode
      beq.d    r2,Lsdone      ; done if no more faults
Lsloop:
      subu     r2,r2,1        ; one less fault
      bne      r2,Lsloop      ; do more if there are more
      putcr    r3,r0          ; put no-op in Shadow IW slot
      addu.d   r3,r3,5        ; point to next slot and go

Lsdone:
      getcr    r2,c9          ; restore registers
      getcr    r3,c10
      rte                     ; return from exceptions
```

Figure 6.5 A quick and dirty exception handler

possibility is to copy the Shadow IW to memory and reload only those slots which are still valid. This is a fairly slow process, so instead of removing a faulty instruction from the Shadow IW, its opcode field is replaced with the value zero. This value is the opcode of the **`and r0,r0,0`** instruction, which has no side effects and does not fault. It is much quicker to issue this instruction upon returning from the exception handler than it is to modify the Shadow IW to remove a faulty instruction.

It is possible to have more than one faulty instruction in the Shadow IW, so the exception handler must continue to examine Shadow IW slots until all faulty instructions have been replaced. It is also possible that there are *no* faulty instructions (in the case of an interrupt), so that eventuality must be considered as well. A more complex exception handling example, which does a context switch between two running processes, is shown in Appendix B.

6.6. Circuit Complexity

The design of Fred is relatively straightforward. Most of the complexity is in the Dispatch Unit. Data hazards are resolved with a simple 31-bit scoreboard. Normal structural hazards do not require any additional scoreboarding or control, since the flexibility of the micropipelines eliminates the need for inserting artificial pipeline stalls. To prevent deadlock, the Dispatch Unit must be aware of the number of items in the R1 Queue and Branch Queue, but this can be done with a simple shift register.

The size of the IW may have the largest impact on the circuit complexity, especially since each IW slot requires a significant number of bits for exception processing. Table 6.1 shows the bits needed for each of the IW slots in the current VHDL implementation of Fred. The number of slots in the IW is arbitrary, but obviously will have some effect on performance (see Section 7.2). A variety of options exist which could reduce the complexity or size of the processor circuitry.

6.6.1. Control Registers

The amount of circuitry needed could be reduced significantly by eliminating the Shadow IW. Saving the entire IW in a set of control registers at exception time roughly doubles the number of transistors needed to implement the IW. Eliminating the Shadow IW would be an important goal for a physical implementation.

This can be done by revising the dispatch logic such that the IW is entirely disabled while exceptions are being handled. The control registers used to access the Shadow IW would actually access the IW itself. This means that instructions are not tracked in any way and, therefore, must *not* cause exceptions. This is not an unreasonable requirement for an exception-handling routine.

A second side effect is that the R1 Queue must not be accessed while the IW is disabled. To prevent deadlock and WAW hazards, the Dispatch Unit uses the IW to keep track of the number of items in the R1 Queue and to scoreboard register **`r1`**. If the IW is disabled, this cannot be done correctly. Therefore, a reentrant exception handler routine would consist of four stages: 1) save the IW state while the IW is disabled, 2) reenable the IW and save the R1 Queue and Branch Queue contents, 3) reenable the Branch Queue to possibly allow nested exceptions, and 4) continue with normal exception processing.

6.6.2. IW Slots

Another contributor to the size of the IW is the number of bits needed for each slot, with the operands of faulty instructions making up nearly half of the total. One alternative is to add some form of history buffer to maintain the original register values. However, doing so would complicate the completion logic without necessarily reducing the size of the processor. Additionally, this would require reversing operations on the R1 Queue, which is *extremely* difficult (see Section 5.1.5).

Another alternative is to change the way in which data dependencies are detected. Currently, because faulty instructions return their operands to the IW, a simple scoreboard model is all that is needed to resolve register antidependencies. Exceptions can violate these dependencies, requiring the original operands to be recovered. If the dispatch logic were revised such that instructions could issue while always avoiding WAW hazards, then the register file contents would be sufficient to reissue faulty instructions. This would also allow instructions to be aborted if desired. By eliminating WAW hazards, the instruction operands would no longer be needed in the IW, reducing its size by nearly 50%.

However, a potentially significant drawback would be the possible reduction in program efficiency. The flexibility of the dynamic scheduling mechanism would be decreased, since dependencies on issued as well as nonissued instructions would have to be considered when searching the IW for instructions to issue. The degree of parallelism in most programs is not great [43], yet it is enough that some pipelining is possible. With WAW-safe dispatch, no two concurrent instructions can use the same registers for either source or destination. It is questionable whether typical programs have enough parallelism to maintain performance under these conditions.

6.7. Software Tools

To obtain meaningful information about the performance issue affecting the Fred architecture, it was necessary to be able to simulate accurately programs running on a prototype implementation. Several software tools were developed to assist with this process.

6.7.1. Assembler

Rather than write an assembler from scratch, it was decided to use the GNU assembler [11]. The GNU assembler offers several advantages: it is available for free, it works, it is fairly robust, and it is relatively portable. It has only one major disadvantage: there is no documentation on exactly how to port it to a new architecture. Details of the porting process can be found in Appendix D.1. Because the Viewlogic VHDL simulation engine is only installed locally for Sun SPARC workstations, some shortcuts were taken which imply that the assembler will only work for that same architecture.

6.7.2. Linker

Unfortunately, the assembler by itself cannot do everything. It is correct as far as it goes, but the linker must be able to handle Fred's 16-bit immediate values correctly. An attempt was made to use the SPARC linker without modification, but it was unsuccessful. It was necessary to port a linker for Fred also. Details of porting the GNU linker are found in Appendix D.2.

6.7.3. Compiler

As mentioned earlier, Fred's instruction set is very close to that of the Motorola 88100. Two C compilers for that architecture were available on the local 88100 machines: the GNU compiler (version 2.4.5) and the native Green Hills Software compiler (version 1.8.5m16). There was no need to port a separate compiler for Fred, since these compilers produce assembly code that is very nearly what is needed by Fred.

Neither compiler is especially intelligent. Even with all optimization turned on, only a few of the available registers were used, memory references were common, and several obvious optimizations were not performed. There is clearly lots of room for improvement here.

6.7.4. Translator

There are enough differences between the 88100 instruction set and the Fred instruction set that an additional step was needed to translate the assembly code between the two machines. C programs can be compiled into 88100 assembly language, but the result must then be translated into Fred's assembly language before running the assembler. A Perl script was written to handle this translation.

Most of the translation was straightforward. Some instructions emitted by the 88100 compilers are not implemented in Fred but can be easily emulated by other instructions, and the branch and subroutine call instructions can be decomposed into the appropriate branch/**doit** pair. As an optional additional step, the newly decoupled branch instructions can be relocated by a simple peephole optimization scheme built into the translator.

One hardware convention could potentially cause trouble. On the 88100, register **`r1`** is a normal general purpose register except during subroutine calls (when the return address is automatically written to it), whereas on Fred register **`r1`** is the access port to the R1 Queue and cannot be used as a general purpose register. Fortunately, the 88100 ABI [25] indicates that 88100 register **`r28`** is reserved for future expansion and should never be used by a compiler, so it was possible to replace all 88100 references to **`r1`** with Fred references to **`r28`**, leaving Fred's **`r1`** register available for the R1 Queue.

6.7.5. Other Utilities

Several miscellaneous utilities make life slightly easier. A disassembler was developed which will either disassemble object files or prompt for opcodes from **`stdin`**, depending on the command-line arguments. It can produce output in assembly language format or in a form used by Viewlogic's ViewSim program to initialize simulated memory models. The latter form is used to load programs into the simulator.

Since the disassembler uses the data structures defined by the assembler in **`gas/config/tc-fred.c`**, if any changes are made to the assembler, the disassembler must be changed also. This was a common occurrence in the early phase of the design process, since the opcode table changed occasionally to reflect the evolving design, so some Perl scripts were written to automate the process. A **`peek`** program to display the headers of an object file was also written.

6.8. Simulator

Discrete circuit components are used to model the glue logic and FIFO controls of the Fred simulation model, whereas the major functional units are written in behavioral VHDL. In order to make the simulation results as meaningful as possible, the delay times for each behavioral process should mimic closely the delay times of the actual circuitry needed to implement the same function. Accordingly, for each step in the behavioral models, the number of gate delays to accomplish the needed functionality was estimated. ViewSim's default gate delay time for discrete components is 0.1ns, so the overall delay time was scaled by this value. The delays used in the Fred model are shown in Table 6.9. The schematics and VHDL code are available elsewhere [31].

Table 6.9 Simulation delay times

Functional Unit	Operation	Delay time (nsecs)
Dispatch	add to IW	0.5
	search IW	0.3
	decode instruction	1.0
	retire instruction	0.4
	save Shadow IW	1.0
	getcr	0.4
	putcr	0.5
	doit	0.3
	rte	1.0
	sync.x	0.3
	sync.x abort	0.4
Arithmetic	add	0.5
	sub	0.6
	cmp	0.6
	div	18.0
	mul	3.0
Logic	boolean operations	0.1
	barrel shifter	0.5
	ff0, ff1	0.5
Control	mvpc	0.4
Branch	absolute	0.1
	relative	0.2
	conditional absolute	0.4
	conditional relative	0.5
Memory	decode operation	0.6
External memory	Imem response	2.0
	Dmem response	2.0

CHAPTER 7

FINDINGS

This chapter describes the results obtained from running several benchmarks through the Fred simulator. Although the benchmarks are not particularly large, representative results may still be obtained because every signal transition is timed. The benchmarks used are shown in Table 7.1. Detailed instruction and performance breakdowns are tabulated in Appendix C.

All of the benchmarks are written in C. The code was compiled for the Motorola 88100 using either the GNU C compiler (v. 2.4.5) or the Green Hills compiler (v. 1.8.5m16) and then translated into Fred's assembly language using the program described in Section 6.7.4. All possible optimization flags were used, to little effect. Both compilers produced very poor code, using only a few of the available registers, making many memory references, and leaving many obvious optimizations undone.

Two major parameters of the Fred simulator were varied, and each of the 14 benchmarks was executed under each configuration. First, the number of IW slots was varied between 2 and 16. At least one IW slot is needed to hold an instruction prior to dispatch, but with only one slot, there is no provision for out-of-order dispatch or completion, and this is not very interesting.

Second, the number of latch stages in each FIFO queue was varied from 0 to 8. With zero stages, there is no storage in the FIFO queue at all, and each request/acknowledge pair between functional units is directly connected. Although there are many FIFO queues in the Fred processor, they were not varied independently since general performance trends were of more interest than tweaking the queue sizes for maximum performance on a given benchmark.

In order to gather information regarding exception handling, each benchmark was subjected to several interrupt signals at pseudo-random intervals. The addition of the interrupts affected the benchmark performance by less than 0.5% on average but was useful in measuring exception performance.

Table 7.1 Benchmark programs

Program name	Dynamic instruction count	Description
ackermann	1660	recursion test for m=2, n=6
cat	7109	copy stdin to stdout, for "cat.c" source
cbubble	13300	bubble sort on 50 integers
cquick	5680	quicksort on 50 integers
ctowers	3095	towers of Hanoii, 4 rings
dhry	1710	dhrystone v. 2.1, 3 loops
fact	2858	10 factorial, computed 5 times
grep	13668	search for "printf" in cat.c source
heapsort	2465	heapsort on 16 integers
mergesort	1857	mergesort on 16 integers
mod	4582	test of 10 modulo operations
muldiv	1669	test of multiply and divide
pi	13883	compute 10 digits of π
queens	8181	solve 5 queens problem

7.1. Average Performance

The average performance is more dependent on the length of the FIFO queues than on the size of the IW. There was no appreciable difference in performance for IW sizes greater than 3 slots. Figure 7.1 shows the relationship between average performance and queue length for various IW sizes. Because the Dispatch Unit searches the IW for executable instructions in a parallel manner, the main factor affecting performance is the time it takes to complete an instruction. As long as the IW is large enough to issue instructions efficiently, the IW size only affects performance in terms of saving state during exception handling. The best performance was seen for a FIFO length of one and an IW size of four or more slots. For greater FIFO lengths, the instructions spend more time in the queues than they do in execution.

7.2. IW Slot Usage

Figure 7.2 shows the average IW slot usage for all configurations. With longer queue lengths the time needed for each issued instruction to complete is greater, giving more time for the IW to be loaded with instructions, so the usage increases. As the number of IW slots increased, the average IW usage also went up, but this is to be expected since there are more slots available. Regardless of the configuration, the average IW usage is still no greater than 3.5 slots, and for a FIFO length of one, the average IW usage is never above 2.3. Figure 7.3 shows the same data in a different manner. The relatively high usage seen when the queue length is zero is due to the inability to dispatch more than one instruction at a time. Because there is no storage in the queues, there is essentially no pipelining except for those instructions which can be sent to separate functional units.

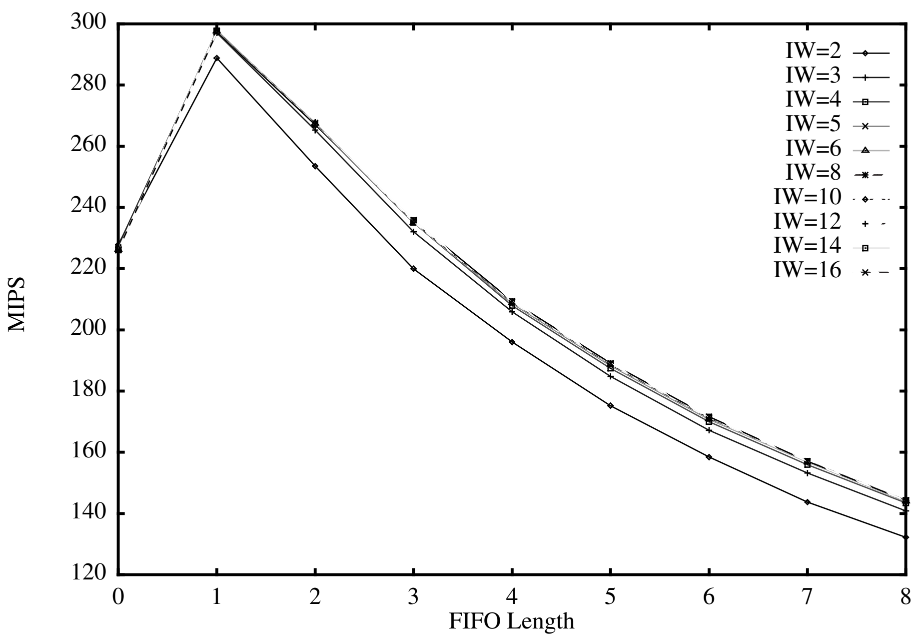

Figure 7.1 Average performance vs. IW size

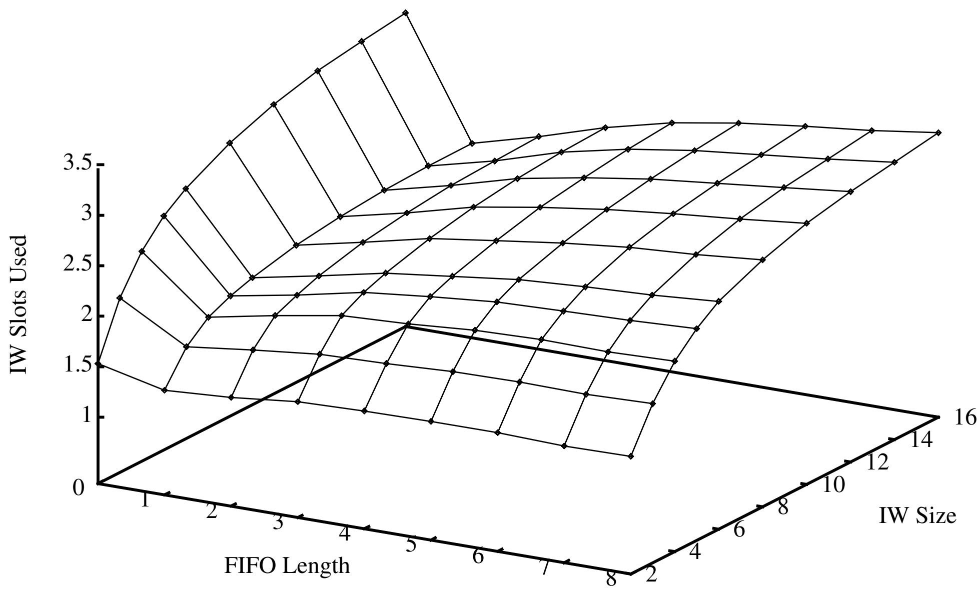

Figure 7.2 Average IW usage

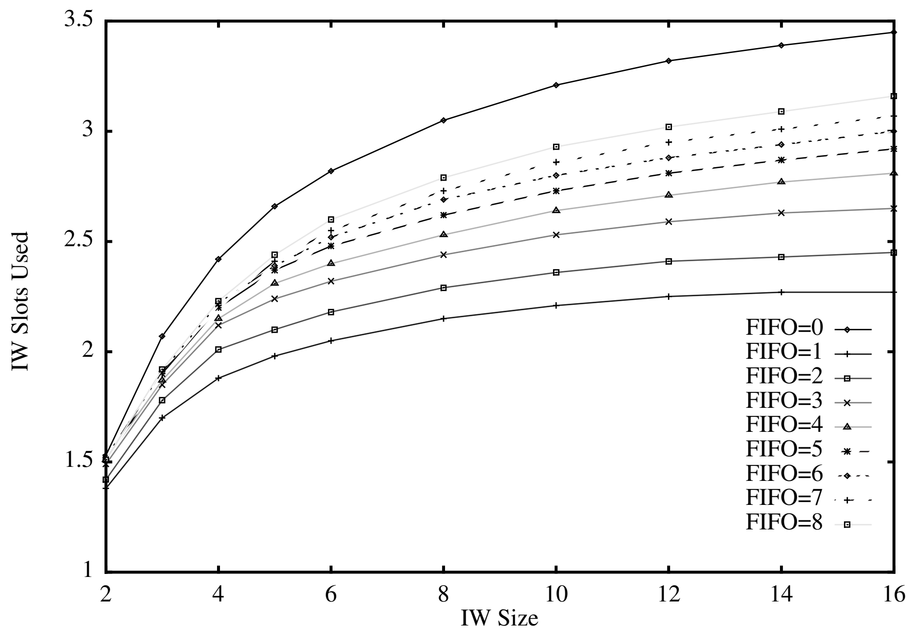

Figure 7.3 Average IW usage

7.3. Decoupled Branches

Fred's decoupled branch mechanism allows for a variable number of "delay slots." The compilers used for the benchmarks generate code for the Motorola 88100 processor, a synchronous RISC processor which has only a single delay slot. This allows at most one instruction to be placed between the branch instruction and the first instruction at the target address. The instructions generated by the 88100 compiler are translated into Fred's instruction set, and a very simple peephole optimization is performed to separate the branch and **doit** instructions as far as possible within each basic block. Typical branch separation results are shown in Table 7.2, as obtained from execution traces. Dynamic scheduling may alter the traces for different configurations. On average, the number of useful delay slots is greater than one. With a compiler targeted specifically for Fred, the separation should be much greater. The time available for instruction prefetching is directly related to the separation between the branch target calculation and the **doit**.

The average branch prefetch time is shown in Figure 7.4. The prefetch time is the time difference between the computation of the branch target data by the Branch Unit and the recognition of a **doit** instruction by the Dispatch Unit. A positive time value indicates time available to prefetch the target address before it is needed by the Dispatch Unit. A strongly negative value usually means that the decoupling between the branch computation and the **doit** is too low. For example, the instruction **`br.d foo`** would always have a negative prefetch time, since the **doit** is recognized by the Dispatch Unit before the branch instruction is even dispatched.

Both the IW size and the FIFO length affect the branch prefetch time. Increasing the FIFO length adds to the time needed for a branch instruction to move from the Dispatch Unit to the Branch Unit. Since the **doit** is recognized by the Dispatch Unit as soon as it is fetched and is unaffected by the FIFO length, the increased time to execute the branch instruction can only decrease the overall prefetch time.

Table 7.2 Dynamic branch separation

Benchmark	Separation
ackermann	1.52
cat	1.82
cbubble	0.81
cquick	1.59
ctowers	1.67
dhry	1.56
fact	0.84
grep	1.14
heapsort	1.54
mergesort	1.14
mod	2.03
muldiv	2.66
pi	1.89
queens	0.88
average	**1.51**

The size of the IW affects the prefetch time also. Although Fred can dynamically reorder instructions, if several instructions may be dispatched, preference is given towards program order. A larger IW means that more instructions must be examined (and possibly dispatched) before the branch instruction is considered for issue. Again, since the **doit** is recognized immediately, this additional time can only decrease the prefetch time. An interesting subject for future investigation would be to alter the dispatch logic to give greater preference to branch instructions.

Figure 7.5 and Figure 7.6 show a further breakdown of the branch prefetch times. Nontaken branches evince significantly poorer prefetch times than taken branches. This is primarily an artifact of the compiler. Most taken branches occur at the bottom of a basic block, and postprocessing of the assembly output can provide greater decoupling for these cases. Roughly 70% of the dynamic branch instructions are taken branches, so this factor outweighs the nontaken case. Additionally, nontaken branches by definition will execute the next sequential instruction, so prefetching for nontaken branches does not actually require advance knowledge of the branch target.

Finally, the figures seem to indicate that the longest prefetch time is for an IW size of two. This does not mean that this configuration provides the best performance, since other factors come into play. For example, a longer prefetch time can be obtained by slowing down the processor, but that does not result in an overall performance increase. Figure 7.1 indicates that the best performance is for an IW size of at least four.

7.4. Dynamic Scheduling

The degree to which instructions are issued out of program order is shown in Figure 7.7. Obviously, as the size of the IW increases more instructions are available to choose from, so the number of instructions which can issue out of order increases. This does not have a marked effect for IW sizes greater than four, since at that point the degree of parallelism available to the pro-

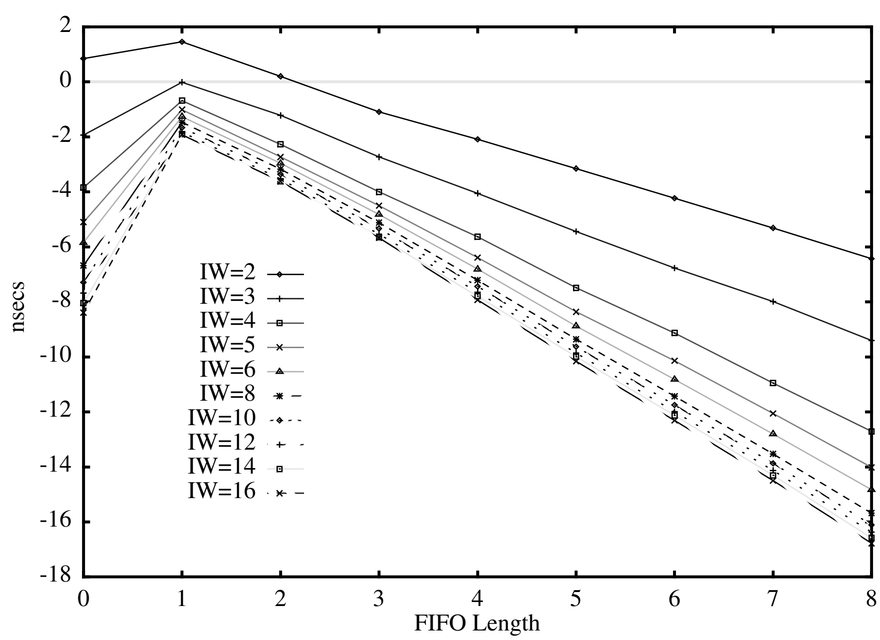

Figure 7.4 Average branch prefetch time

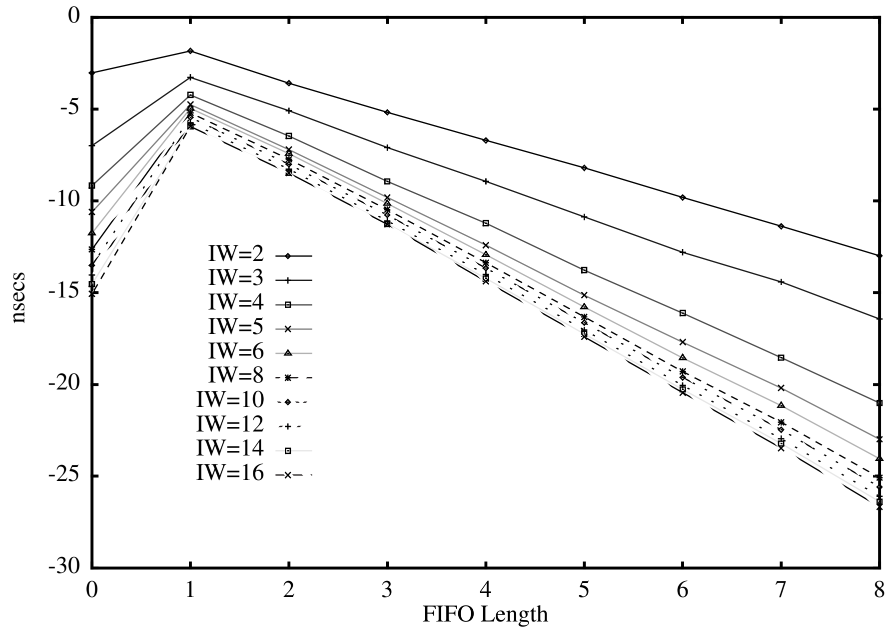

Figure 7.5 Branch prefetch times for nontaken branches

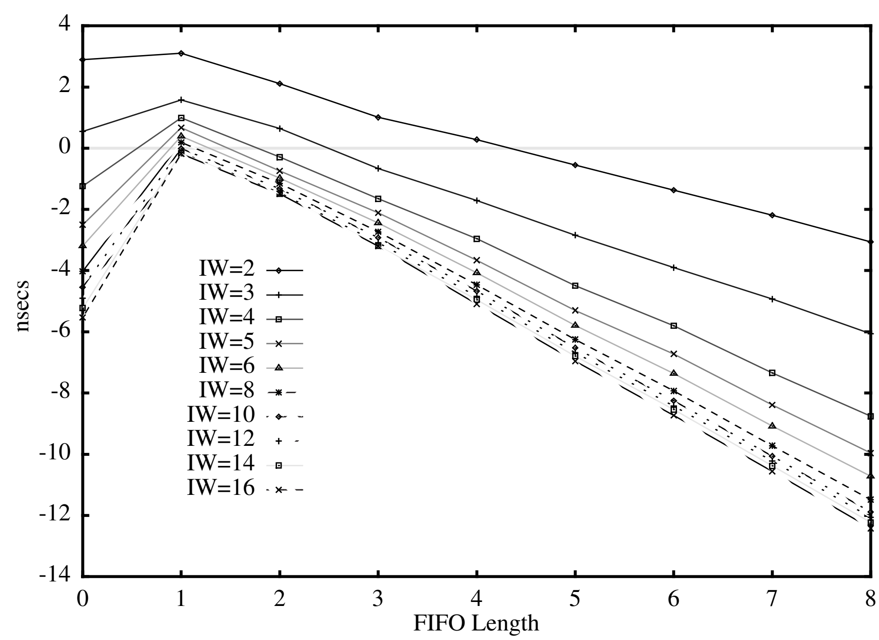

Figure 7.6 Branch prefetch times for taken branches

gram limits the dynamic scheduling. The FIFO length has a more noticeable effect. Longer FIFOs mean that instructions take longer to execute, so that instructions stay in the IW for a longer time. Fred normally gives precedence to in-order dispatch, but as each instruction takes longer to execute and signal completion, the slower dependency resolution causes out-of-order dispatch to become more frequent. Regardless of configuration, the average out-of-order distance is between one and two instructions. Figure 7.8 shows the effect of out-of-order dispatch for a FIFO length of one. The difference between allowing out-of-order dispatch and forcing sequential dispatch is only around half a percent in terms of overall average performance, primarily due to the relatively slow instruction fetch time, which tends to limit overall performance. Surprisingly, the performance is slightly less when out-of-order dispatch is enabled. Dispatching out of order requires increased activity by the Dispatch Unit, which must arbitrate for access to the IW and scoreboard. This extra arbitration is most likely the primary cause of the phenomenon.

7.5. Last Result Reuse

The evaluation of the last-result-reuse policy uncovered some unexpected results. Figure 7.9 shows the average percentage of instructions for which last-result-reuse is possible. The slight but gradual decline in last-result-reuse as FIFO length increases is a side-effect of the dynamic instruction scheduling. The average out-of-order distance is not very great, but as FIFO lengths increase, more time is available to fetch instructions, dependencies take longer to resolve, and more instructions are issued out of order. Operand reuse (by definition) occurs when a register value is latched in a specific functional unit and is then reused as a source operand by a following instruction within the same functional unit. The compiler tends to implement each high-level program statement with several instructions which use the same functional unit. If there is a greater

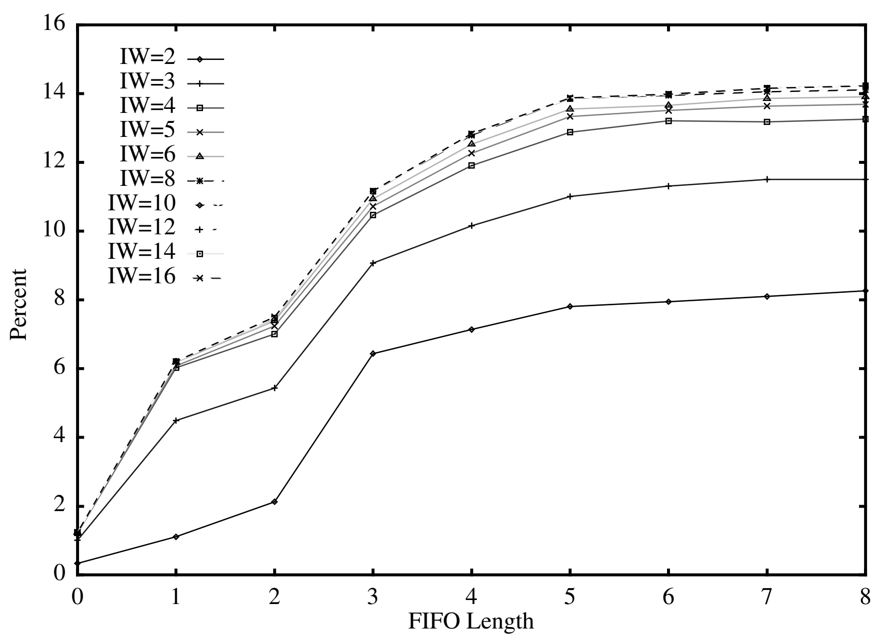

Figure 7.7 Percentage of instructions issued out of order

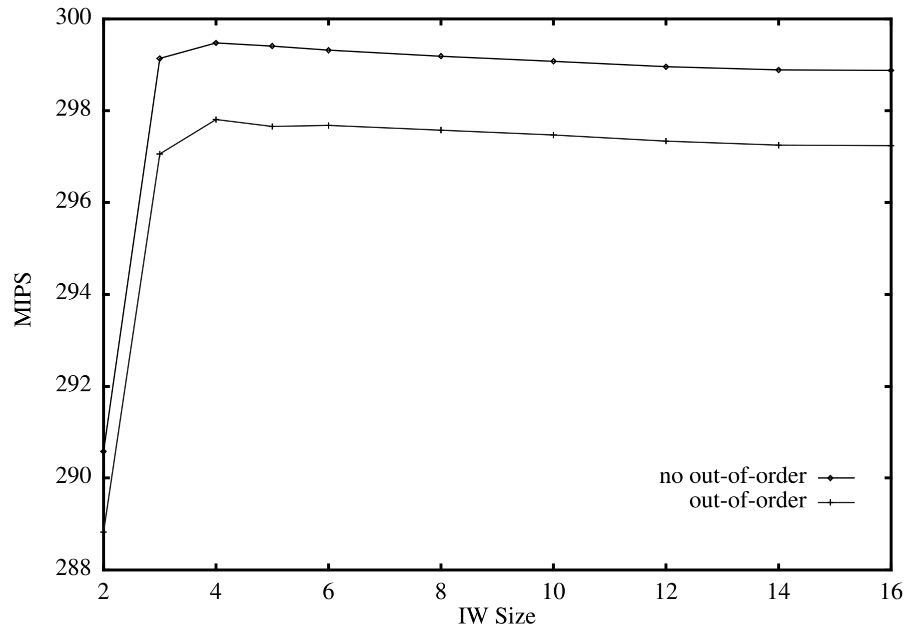

Figure 7.8 Effect of out-of-order dispatch on performance

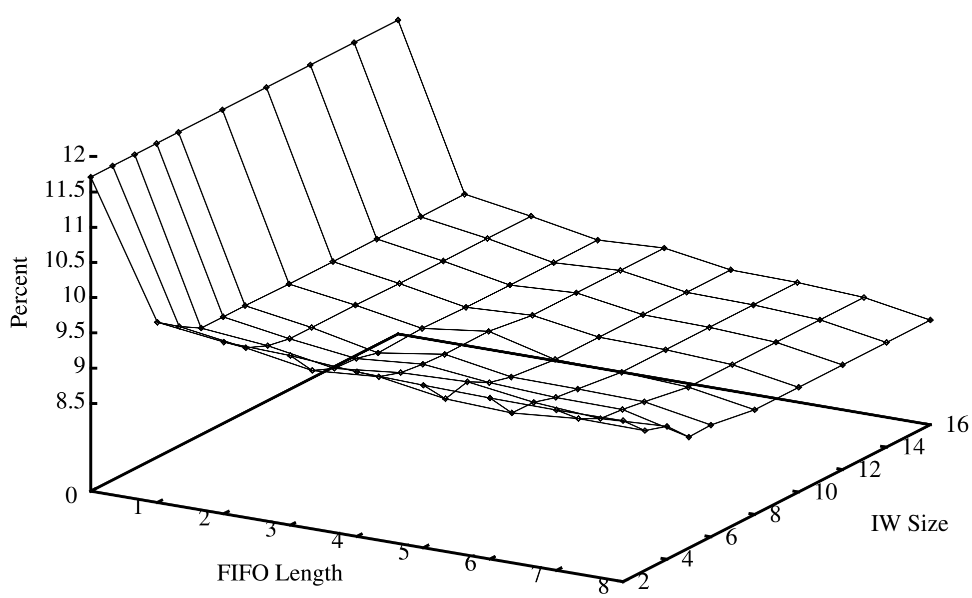

Figure 7.9 Instructions for which last-result-reuse is possible

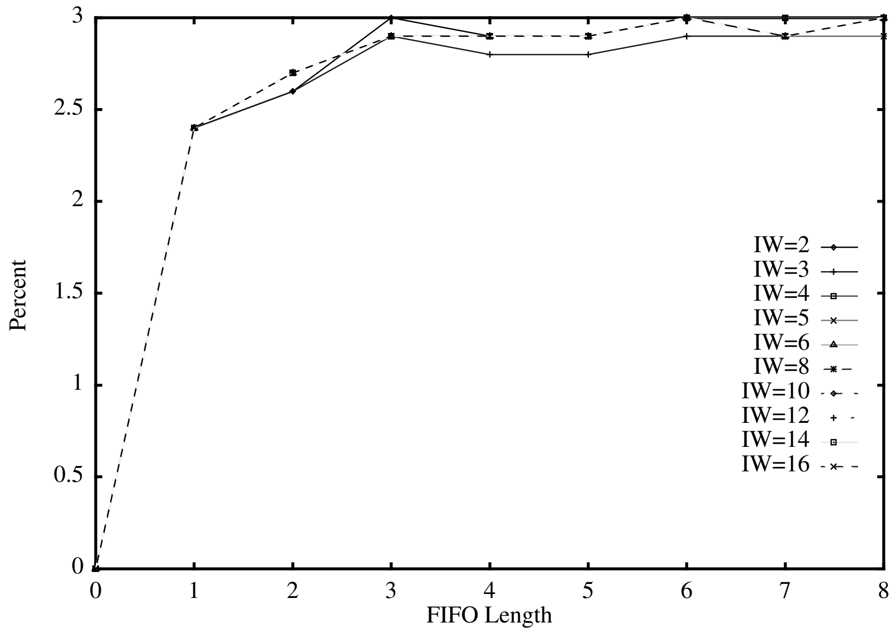

Figure 7.10 Instructions for which last-result-reuse helps

opportunity to issue out of order, it is more likely to issue instructions from different program statements, which use the same functional unit to compute different register values. This replaces the register value computed from the first instruction, so that it is no longer available for reuse by the next one. This code provides an example:

```
add  r2,r3,r4
add  r5,r2,42
sub  r15,r19,r20
```

All three of these instructions must signal completion. For short FIFO lengths, instructions dispatch and complete quickly, so that most of the time they are issued in sequential order. After the first instruction completes, the second one can issue and reuse the previous result, but because of the dependency on **r2**, the second instruction must wait for the first to complete before it can issue. If the first instruction is slow to complete, the third instruction may issue before the second one. In this case, the result left in the Arithmetic Unit is **r3**, not **r2**, and when the second instruction issues, it cannot reuse the result.

Another interesting result is how little difference last-result-reuse makes to the overall performance. Figure 7.10 shows the percentage of instructions which can issue early due to last-result-reuse. No more than 3% of the instructions benefit from last result reuse, since in most cases the results are written into the Register File more quickly than the Dispatch Unit can issue the next instruction.

7.6. Completion Signal

Not every issued instruction must report its completion to the Dispatch Unit. This enables a slight speedup in performance, since there is less communication with the IW. Because nonreporting instructions can be removed from the IW as soon as they are dispatched, there is a corresponding decrease in the average IW usage. Table 7.3 and Table 7.4 tabulate the differences for a FIFO length of one and an IW size of four slots.

Table 7.5 shows the performance when all completion signalling is disabled. Completion must be enabled to detect faulty instructions, so this information is of limited use. However, if the non-faulting constraint suggested for exception handlers in Section 6.6.1 can be met, it may be possible to disable all completion while executing certain sequences of supervisor code. This could provide a measurable increase in the exception handling performance.

Table 7.3 IW slot size

Field	Bits	Meaning
VALID	1	this slot is filled
ISSUED	1	instruction has been dispatched
TAG	4	dispatched instruction tag
ADDRESS	30	instruction address
OPCODE	32	instruction opcode
WAT	1	can be issued only at the top of the IW
SINGLE	1	inhibits further instruction prefetching
STATUS	8	completion status
ARG1	32	used for fault recovery
ARG2	32	used for fault recovery

Table 7.4 Completion signalling and IW usage

Benchmark	IW usage with forced completion	IW usage with optional completion	Reduction
ackermann	2.39	1.65	31.0%
cat	2.56	1.63	36.3%
cbubble	2.41	2.03	15.8%
cquick	2.65	2.03	23.4%
ctowers	2.80	2.21	21.1%
dhry	2.68	2.27	15.3%
fact	2.02	1.24	38.6%
grep	2.15	1.57	27.0%
heapsort	2.49	1.91	23.3%
mergesort	2.33	1.75	24.9%
mod	2.63	2.19	16.7%
muldiv	2.76	2.27	17.8%
pi	2.60	2.10	19.2%
queens	2.29	1.47	35.8%
average	**2.48**	**1.88**	**24.7%**

Table 7.5 Performance with no completion signalling

Benchmark	MIPS with optional completion	MIPS with no completion	Percent increase
ackermann	340.91	349.87	2.6%
cat	349.73	349.90	0.0%
cbubble	255.19	275.58	8.0%
cquick	287.81	291.52	1.3%
ctowers	303.45	329.69	8.6%
dhry	276.43	303.12	9.7%
fact	323.96	327.34	1.0%
grep	287.30	288.91	0.6%
heapsort	299.11	304.42	1.8%
mergesort	297.68	316.48	6.3%
mod	288.87	302.17	4.6%
muldiv	290.62	310.54	6.9%
pi	252.61	269.36	6.6%
queens	315.60	320.96	1.7%
average	**297.81**	**309.99**	**4.3%**

7.7. Exceptions

There are several factors affecting exception handling. The latency is the time between when an exception condition is first detected by the Dispatch Unit and when exception processing begins by copying the IW into the Shadow IW before branching to the exception vector. The latency can vary, since exception handling cannot begin until all outstanding instructions have completed. However, the time for those instructions to complete is strongly dependent on the length of the FIFO queues, as seen in Figure 7.11.

The actual exception latency varies from essentially zero time when there are no outstanding instructions to a much longer period when there are several slower instructions pending. The average maximum measured latency for all configurations is shown in Figure 7.12, but this is not necessarily representative of anything in particular, since there is no theoretical upper bound on the response time for self-timed systems. In practice, statistical methods may be used to predict response time, but there is no hard limit.

As might be expected, the number of IW slots in use when exceptions are taken is very similar to the average IW usage seen in Figure 7.2. However, the number of IW slots reported in use for exception handling does not include outstanding instructions, since they complete before exception handling begins and only faulty and nonissued instructions are seen in the IW during exception conditions. The number of IW slots in use when exception handling begins is 10 to 20% less than the average for nonexception conditions.

A related item of interest is the time needed to perform a context switch. Saving the register contents and program counter values takes a constant amount of time, but the items in the IW, the Branch Queue, and the R1 Queue must also be saved, and the number of these items may vary. A test program which performs context switches between two running processes is listed in

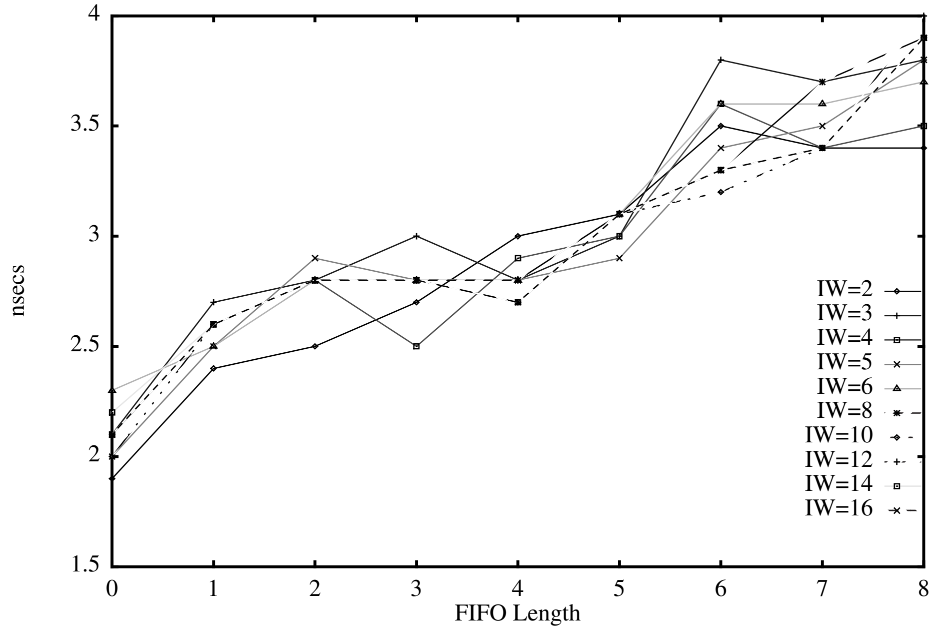

Figure 7.11 Average exception latency

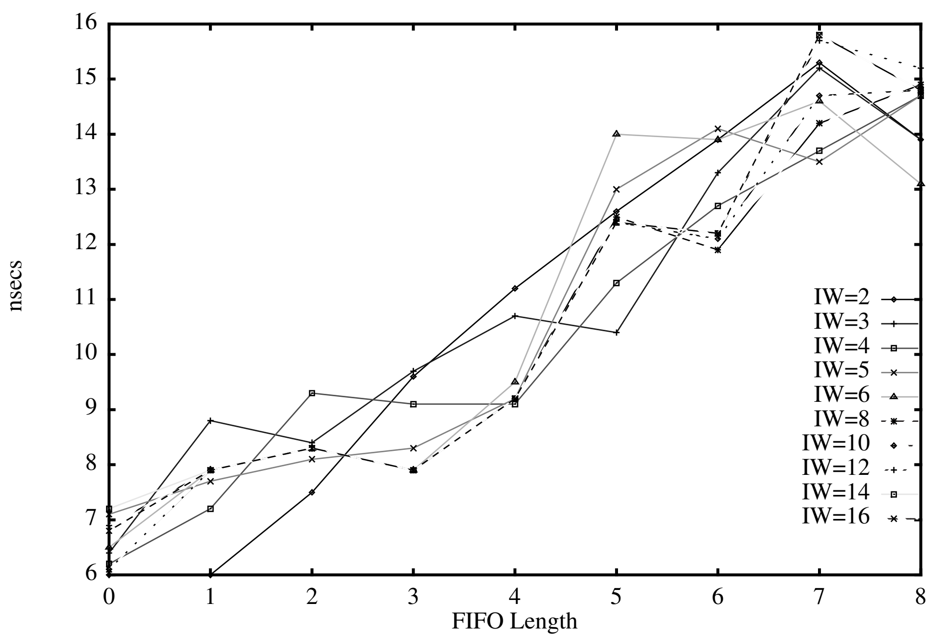

Figure 7.12 Maximum observed exception latency

Appendix B. Under the optimum configuration (IW size is four, queue length is one), the program performs a complete context switch in 940 nsecs using 220 instructions on average.

7.8. Real Performance

For Fred, the major factor affecting performance is the response speed of instruction memory. In the Fred model, there is no pipelining of memory accesses, so each instruction is individually requested and acknowledged before the following instruction is requested. To provide a baseline for comparison with the simulation results, the benchmarks were executed on an existing Motorola 88100 UNIX workstation. Unlike Fred, the 88100 is able to prefetch sequential instructions at a rate of one per clock and only suffers a pipeline stall when a branch is taken. Taking this difference in memory accesses into account, and scaling the clock frequency and estimated gate delays of the 88100 to match those assumed in the Fred simulator (Table 6.9) provides an estimated average performance measurement of the 88100 that is within 15% of the results reported for Fred. Closer estimates are not possible with any degree of accuracy, due to the myriad implementation assumptions which must be made. The rough estimate is encouraging, since it seems to indicate that the Fred architecture is not unreasonably handicapped in terms of performance. Chapter 8 suggests several ways in which the Fred architecture and performance may be increased.

CHAPTER 8

FUTURE WORK

A number of aspects of the Fred architecture would benefit from further investigation. Some of these issues would certainly be subjects for detailed analysis if Fred were to be built as a commercial microprocessor. However, commercial processors typically require the design efforts of dozens of people, often over a period of several years, and the scale of such effort precludes its investigation here. Other aspects could well be fruitful topics for additional research by one or two persons.

8.1. Instruction Issue

Currently Fred fetches and issues only one instruction at a time, yet there is no conceivable reason why the instruction fetch path cannot be several words wide. There is also no reason not to attempt superscalar instruction issue, although this would require substantial modification to the dispatch logic and the Register File.

Modification of the register scoreboard could also result in better last-result-reuse characteristics, as mentioned in Section 6.2.3.3. This also would require modification of the dispatch logic.

Trap instructions are increasingly used to perform user-specific functions [40]. Fred could well benefit from a faster means of invoking user traps. It may be possible to restructure the trap instructions into something similar to the branch/**doit** framework, so that a trap just loads a target address into the Exception Branch Queue, which is used at some later point in the program without causing the processor to stall first.

8.2. Result Forwarding

Although Fred currently implements last-result-reuse within a single functional unit (with limited benefit), forwarding of results between functional units has not been attempted. Forwarding in an asynchronous architecture requires explicit synchronization between functional units and is often difficult. Some research in this area has been done by others [12,13].

8.3. Floating Point Arithmetic

Fred does not implement any floating point arithmetic, mainly because it adds additional complexity without contributing significantly to the major issues under investigation, such as exception handling and IW dispatch logic. Obviously any commercial implementation would require floating-point capability.

8.4. Preloading Cache

An instruction cache which could take advantage of the prefetching information made possible by the decoupled branch mechanism is certainly worth further investigation. The design, behavior, and architectural features of a preloading cache like that conjectured in Section 4.4.3 offers much scope for additional research. The potential benefits of such an inquiry are substantial.

8.5. Speculative Prefetching

Essentially, speculative prefetching involves assuming a direction for a branch before it is taken, so as to hopefully avoid the penalty involved in waiting for the branch to be resolved before fetching new instructions. This can take place at two levels. If a preload cache is used as mentioned above, there is no additional logic needed to recover from an incorrect guess, since the preload cache is simply invalidated as soon as the assumed branch direction is known to be incorrect.

The second level involves speculative prefetching into the IW. This might be useful, but it would require some additional bits to distinguish speculative instructions from those which are known to be correct. Additional hardware is also needed to flush the speculative instructions from the IW for incorrect guesses.

8.6. Speculative Execution

Speculative execution is a logical extension of speculative prefetching, in which the prefetched instructions are executed before it is known whether or not they are valid. Speculative execution adds several hardware requirements to the architecture. First, some form of history or reorder buffer is needed to provide a sequential state, consistent with the in-order completion of instructions, because it must use a fully precise exception model in order to go back to the branch point and take the other choice if the speculation went the wrong way.

In turn, this means that *every* instruction must be removed from the IW in sequential order. To do so, every instruction must report its completion status. This carries no penalty for a synchronous design, but there *is* a penalty for an asynchronous design, since every transaction consumes power and takes time. Additionally, by not requiring a completion signal from every instruction, the IW size may be reduced.

Fred's Branch Queue cannot be run speculatively without a *lot* of trouble. For example, consuming a **doit** before an instruction faults is a big problem because it is extremely difficult to restore the consumed branch target data to the Branch Queue. However, stalling on every **doit** just in case there is a fault is not a good solution. In the benchmark tests, roughly 12% of **doits** immediately follow an instruction which could potentially fault, and around 17% of all basic blocks contain at least one potentially faulty instruction in the delay slots.

Fred intentionally does not issue speculative instructions. The entire point of the functionally precise exception model described in Chapter 5 is that the order of completion does not matter. Requiring a traditional precise exception model means that order of completion *does* matter, at least in the order in which instructions are retired from the IW. The decoupled memory access should allow for enough prefetching time to obviate the need for speculative execution [15]. Evaluation of a synchronous processor design using an IW indicates that speculative execution does not always provide a performance increase [10]. Nonetheless, this could be an interesting topic to investigate further.

8.7. Memory Unit

Fred's Memory Unit is a very simple one. Memory interfaces for commercial microprocessors contain a number of optimizations from which Fred would benefit. Dynamic reordering, split transactions, burst mode transfers, and other such techniques may be worth considering.

8.8. Compiler

This is probably the area requiring the most investigation. As with any other RISC design, Fred's performance is greatly dependent on the associated compiler technology, yet developing compilers for an asynchronous architecture is even more difficult. A number of factors present additional problems.

First, predictive scheduling becomes necessary. In a synchronous processor (assuming perfect memory) every instruction takes a known integral number of clock cycles to execute, which makes scheduling instructions in a noncompeting order much simpler. In a self-timed processor, not only can each instruction take a different amount of time, but the actual time may vary depending on the data, environmental conditions, and other arbitrary factors. Add to this the fact that the processor may dynamically reorder instructions at run-time, and instruction scheduling becomes extremely complicated.

In a clocked architecture, reusing registers between successive instructions poses little or no penalty, since register bypass can be used between pipeline stages to ensure that the register value is readily available on the next clock cycle. With Fred, because the execution time and even the relative order of parallel instructions is indeterminate, register coloring may be much more important. It may be that performance can be enhanced by continually renaming registers to avoid artificial dependencies between successive instructions or functional units.

There is also the question of Fred's decoupled branch structure. Determining how far to separate the branch/**doit** pair under specific conditions and developing means by which the separation can be increased are areas worthy of investigation. The **mvpc** instruction is an additional artifact of the decoupled memory access which should be considered.

Further speculation on the details of compiler technology is beyond the expertise of the author.

8.9. R1 Queue

Closely related to the issues involving the Branch Queue is the R1 Queue. This queue is an architectural feature inherited from the original NSR architecture, which was retained because it provides additional challenges to developing a robust exception model. Aside from simple tests to ensure its correct functioning, it has been little used. Much research could be done to determine the advantages and disadvantages of this feature. Some potential advantages have been mentioned elsewhere—register renaming and hiding memory latency, for example. Obviously, the usefulness of the R1 Queue depends largely on its utilization by the compiler.

Additional configurations of the R1 Queue are possible. Rather than allowing all functional units to write to the R1 Queue, it may be possible to restrict its operation to just the Memory Unit with beneficial results. Alternatively, it may be possible to increase the speed of access to the R1 Queue by relaxing the requirement that all instructions which write to **`r1`** report their completion. This would remove the guarantee of in-order access, but this behavior could be placed under program control to be enabled under certain safe conditions.

CHAPTER 9

CONCLUSIONS

A self-timed approach offers several advantages over traditional synchronous design methods. Composability of circuit modules is increased, so that systems may be constructed by connecting components based only on functionality. Incremental improvements to systems are made possible in the same way. Self-timed systems exhibit average-case behavior, as opposed to the worst-case behavior required by a clocked system. Power consumption is reduced, since idle circuits are completely inactive.

These advantages are inherent in the self-timed circuit elements and are available to any architecture which uses them. However, when the asynchronous philosophy is incorporated at every stage of the design, the architecture is more closely linked to the basic structures of the self-timed circuits themselves.

In architectural designs, there are fundamental differences in the structure of asynchronous and synchronous processors. The problems encountered in asynchronous designs differ from those in synchronous designs and require innovative solutions. Several features of the Fred architecture were developed as solutions to such problems. The self-timed design philosophy directly results in a powerful and flexible architecture which exhibits significant savings in design effort and circuit complexity.

9.1. Decoupling

In order to achieve a reasonable performance level, Fred utilizes both pipelining and concurrency. Self-timed circuits (especially micropipelines) are ideally suited to the pipeline aspect, since there is no centralized control logic needed to govern the movement of data through the FIFOs, and as soon as an instruction is issued, no further control is required. This greatly simplifies the processor control logic, since there is no need to explicitly control the progression of each datum through each stage of the pipelines. The concurrency aspect is also simplified. No control logic is required to implement multiple pipelines other than arbitration at the fork and join points, which is done locally with self-timed control elements (Section 3.1).

Much has been written in favor of a decoupled branch mechanism [19,45,30,15,14,16]. The performance of Fred lends support to this view, especially so when the test conditions are considered. The benchmark code used in testing was compiled for a synchronous, nondecoupled architecture by a relatively poor compiler and then subjected to the most simple peephole optimization after compilation. Despite these handicaps, the average number of delay slots between instructions was around 1.5. This argues strongly in favor of the branch decoupling technique.

9.1.1. Delay Slots

Fred's branch decoupling method differs from other decoupled architectures in two significant ways. First, the distance between the branch instruction and the corresponding **doit** is completely variable, whereas in most synchronous decoupled machines a count of the instructions to place in the delay slot is encoded in the original branch instruction. Although the concept of delay slots is still valid, the self-timed nature of Fred renders it of less importance. Instructions are dispatched as soon as possible, not in accordance with some arbitrary time signal such as a clock.

Regardless, fixing the number of delay slots to use would seriously interfere with Fred's ability to reorder instructions dynamically. The **doit** instruction can be issued out of order, just like any other instruction. This would not be possible with a hard-coded delay slot count.

9.1.2. Multiple Branch Targets

A second difference lies in Fred's ability to place more than one branch target into the Branch Queue. A possible use of this ability is to mimic loop unrolling, but without any code expansion, as mentioned in Section 6.3.1.1. Complete utilization of this feature will most likely require the development of a better compiler. Nonetheless, this feature offers some unique opportunities.

9.2. Exceptions

The major innovation of the Fred architecture, which makes possible many additional features, is the functionally precise exception model. A *precise* exception model allow the programmer to view the processor state as though the exception occurred at a point exactly between two instructions, such that all instructions before that point have completed while all those after have not yet started. Instead, Fred's *functionally precise* model simply presents a snapshot of the IW, in which some instructions have faulted, some have not yet issued, and any nominally sequential instructions which are not present have completed successfully out of order. This was discussed in Chapter 5 and Chapter 6.

9.2.1. IW Size

With a precise exception model, even though instructions can issue and complete out of order, they still must retire from the IW in sequential order so that the precise exception model may be maintained. To enable that to happen, every instruction would have to report its completion, even if it could never fault. This imposes an artificial constraint on the behavior and size of the IW, since the size of the IW directly limits the maximum distance between out-of-order instructions. This constraint is imposed solely to handle exceptions, which typically happen at widely spaced intervals. The IW would have to be significantly larger, both to hold the increased number of instructions which must be tracked and to allow sufficient history for out-of-order completion to be possible. For example, the Hewlett-Packard PA-8000 uses a precise exception model and has a 56-entry Instruction Reorder Buffer, which serves the purpose of an IW [22] with out-of-order completion and in-order retirement.

In contrast, Fred's functionally precise exception model allows instructions to retire in any order, in many cases as soon as the instructions issue. The IW must track all issued instructions which might fault only until they have completed successfully. Instructions which will never fault can be removed from the IW immediately after dispatch. Once an instruction has completed, it can be removed from the IW. There is no need to retire the instructions in any particular order, and except for dependency chains, the order of any two instructions is of no importance for either dispatch or completion.

This means that the primary factor governing the size of the IW is that it should be large enough to issue instructions efficiently. Section 7.2 indicates that for the current implementation, an IW of only four slots is sufficient, and that larger sizes have no effect on performance. This small IW size is a direct consequence of the exception model.

The register file is also much simpler under Fred's exception model. With in-order instruction retirement, a history or reorder buffer must be used to ensure that register values are also retired (or made permanent) in order and to restore the original in-order values if an exception is taken. For Fred, no such buffer is needed.

9.2.2. Fast Completion

In a self-timed system, every data transaction requires a request/acknowledge handshake, which uses power and takes some time. Additionally, reporting instruction completion implies a possible contention for access to the IW by several functional units, which must be arbitrated, requiring additional time and power. Eliminating the requirement that every instruction report its completion allows much of the power penalty to be avoided, while also reducing the size of the IW needed. Section 7.6 shows the effect of this improvement. As just mentioned, a precise exception model would require every instruction to report its completion.

9.2.3. Instruction Reissue

With a precise exception model, all out-of-order instructions are discarded when the exception occurs. Assuming that the exception is recoverable, the program must resume at the exception point, thus reissuing those out-of-order instructions. The time and power used in executing these out-of-order instructions the first time are completely wasted. This is not necessarily a major concern, but it could have a measurable effect on performance if exceptions occur frequently (as often happens with certain functional languages).

In contrast, Fred's exception model never reissues any completed instruction. Because of this, the result of every completed instruction is always valid. This means that any instruction currently executing when an exception occurs does not have to be aborted. In fact, any executing instruction that can never fault will not even be recorded in the IW, so the exception handling routine may actually begin before the instruction finishes. If the exception handler needs to access the destination register value associated with that instruction, the scoreboard will ensure that the value becomes valid before it is used. In other words, the Register File does not require any special attention to ensure its correctness during exception handling. The normal scoreboarding mechanisms are sufficient, and exception processing can begin immediately, without waiting for the Register File to become quiescent.

9.2.4. Multiple Faults

Under certain circumstances there may be more than one faulty instruction in the IW when exceptions are detected. For example, this segment of code shows such an occurrence:

```
ld   r2,r3,r4
add  r5,r6,r7
sub  r8,r9,r10
```

These instructions do not have any mutual dependencies, so they may all be executing at once. However, if the address for the load instruction would cause a page fault and the arithmetic instructions would overflow, the IW could indicate that all three instructions have faulted when the exception handler is invoked. Under Fred's exception model, all three faulty instructions can be addressed with one exception invocation. With a precise exception model, each would require a separate exception, discarding any out-of-order instructions each time.

9.2.5. Circuit Complexity

Because the functionally precise exception model does not reverse any state of the processor, the complexity is much less than otherwise might be expected. The IW can be much smaller, since it does not need to worry about the order of instruction completion and does not even need to track nonfaulting instructions. Likewise, there is no need to track or reverse the contents of any registers. Because of this, the hardware needed to implement Fred's exception model may be much less complex than that for a precise model. In a physical implementation, this means that less time is needed to design the exception circuitry and that more silicon area is available to be used for other purposes. Of course, fast design time and reduced complexity (due to the lack of clock circuitry) are two of the often-quoted advantages of asynchronous circuits in general [8].

9.3. Applicability to Synchronous Systems

The elastic nature of a micropipeline FIFO allows Fred's decoupled units to run at data-dependent speeds; producing or consuming data as fast as possible for the given program and data. Obviously, this behavior is inherent in a self-timed organization and cannot be easily attained in a clocked system.

Otherwise, the major difference between synchronous architectures and Fred lies in how the IW is used to handle exceptions. Fred's self-timed implementation reduces the size of the IW, removes the need for register history and state reversal, and thereby eliminates all the circuitry normally used for those functions. Even though the functionally precise exception model was developed for Fred because traditional precise models will not work (Chapter 5), this technique is not solely limited to self-timed systems.

The functionally precise exception model can be directly applied to synchronous processor design at the expense of eliminating speculative execution. Some form of decoupled branch mechanism would most likely be needed to maintain the original performance level normally seen when speculative execution is present. It is conjectured that by making improvements in compiler technology and by utilizing prefetching caches as suggested in Section 4.4.3, a functionally precise exception model may well become the method of choice in future processors, whether synchronous or asynchronous.

APPENDIX A

OPCODES

Opcode Bit Values	Instruction	
000000 ddddd aaaaa iiiiii iiiii iiiii	and	rd,ra,imm16
000001 ddddd aaaaa iiiiii iiiii iiiii	and.u	rd,ra,imm16
000010 ddddd aaaaa iiiiii iiiii iiiii	mask	rd,ra,imm16
000011 ddddd aaaaa iiiiii iiiii iiiii	mask.u	rd,ra,imm16
000100 ddddd aaaaa iiiiii iiiii iiiii	or	rd,ra,imm16
000101 ddddd aaaaa iiiiii iiiii iiiii	or.u	rd,ra,imm16
000110 ddddd aaaaa iiiiii iiiii iiiii	xor	rd,ra,imm16
000111 ddddd aaaaa iiiiii iiiii iiiii	xor.u	rd,ra,imm16
001000 ddddd aaaaa iiiiii iiiii iiiii	add	rd,ra,imm16
001001 ddddd aaaaa iiiiii iiiii iiiii	addu	rd,ra,imm16
001010 ddddd aaaaa iiiiii iiiii iiiii	div	rd,ra,imm16
001011 ddddd aaaaa iiiiii iiiii iiiii	divu	rd,ra,imm16
001100 ddddd aaaaa iiiiii iiiii iiiii	mul	rd,ra,imm16
001101 ddddd aaaaa iiiiii iiiii iiiii	cmp	rd,ra,imm16
001110 ddddd aaaaa iiiiii iiiii iiiii	sub	rd,ra,imm16
001111 ddddd aaaaa iiiiii iiiii iiiii	subu	rd,ra,imm16
010000 nnnnn aaaaa iiiiii iiiii iiiii	bb0	n,ra,imm16
010001 nnnnn aaaaa iiiiii iiiii iiiii	bb1	n,ra,imm16
010010 --001 aaaaa iiiiii iiiii iiiii	bgt	ra,imm16
010010 --010 aaaaa iiiiii iiiii iiiii	beq	ra,imm16
010010 --011 aaaaa iiiiii iiiii iiiii	bge	ra,imm16
010010 --100 aaaaa iiiiii iiiii iiiii	blt	ra,imm16
010010 --101 aaaaa iiiiii iiiii iiiii	bne	ra,imm16
010010 --110 aaaaa iiiiii iiiii iiiii	ble	ra,imm16
010011 iiiii iiiii iiiiii iiiii iiiii	br	imm26
010100 ddddd ----- iiiiii iiiii iiiii	mvpc	rd,imm16
010101 ----- ----- ------ --iii iiiii	trap	imm8
010110 sssss aaaaa iiiiii iiiii iiiii	xmem	rs,ra,imm16
010111 ddddd aaaaa 000000 ----- bbbbb	and	rd,ra,rb
010111 ddddd aaaaa 000001 ----- bbbbb	and.c	rd,ra,rb
010111 ----- ----- 000010 ----- -----	UNUSED	
010111 ----- ----- 000011 ----- -----	UNUSED	
010111 ddddd aaaaa 000100 ----- bbbbb	or	rd,ra,rb
010111 ddddd aaaaa 000101 ----- bbbbb	or.c	rd,ra,rb
010111 ddddd aaaaa 000110 ----- bbbbb	xor	rd,ra,rb
010111 ddddd aaaaa 000111 ----- bbbbb	xor.c	rd,ra,rb
010111 ddddd aaaaa 001000 00--- bbbbb	add	rd,ra,rb
010111 ddddd aaaaa 001000 01--- bbbbb	add.o	rd,ra,rb
010111 ddddd aaaaa 001000 10--- bbbbb	add.i	rd,ra,rb
010111 ddddd aaaaa 001000 11--- bbbbb	add.io	rd,ra,rb
010111 ddddd aaaaa 001001 00--- bbbbb	addu	rd,ra,rb
010111 ddddd aaaaa 001001 01--- bbbbb	addu.o	rd,ra,rb
010111 ddddd aaaaa 001001 10--- bbbbb	addu.i	rd,ra,rb
010111 ddddd aaaaa 001001 11--- bbbbb	addu.io	rd,ra,rb
010111 ddddd aaaaa 001010 ----- bbbbb	div	rd,ra,rb
010111 ddddd aaaaa 001011 ----- bbbbb	divu	rd,ra,rb
010111 ddddd aaaaa 001100 ----- bbbbb	mul	rd,ra,rb
010111 ddddd aaaaa 001101 ----- bbbbb	cmp	rd,ra,rb

Opcode Bit Values	Instruction	
010111 ddddd aaaaa 001110 00--- bbbbb	sub	rd,ra,rb
010111 ddddd aaaaa 001110 01--- bbbbb	sub.o	rd,ra,rb
010111 ddddd aaaaa 001110 10--- bbbbb	sub.i	rd,ra,rb
010111 ddddd aaaaa 001110 11--- bbbbb	sub.io	rd,ra,rb
010111 ddddd aaaaa 001111 00--- bbbbb	subu	rd,ra,rb
010111 ddddd aaaaa 001111 01--- bbbbb	subu.o	rd,ra,rb
010111 ddddd aaaaa 001111 10--- bbbbb	subu.i	rd,ra,rb
010111 ddddd aaaaa 001111 11--- bbbbb	subu.io	rd,ra,rb
010111 nnnnn aaaaa 010000 ----- bbbbb	bb0	n,ra,rb
010111 nnnnn aaaaa 010001 ----- bbbbb	bb1	n,ra,rb
010111 --001 aaaaa 010010 ----- bbbbb	bgt	ra,rb
010111 --010 aaaaa 010010 ----- bbbbb	beq	ra,rb
010111 --011 aaaaa 010010 ----- bbbbb	bge	ra,rb
010111 --100 aaaaa 010010 ----- bbbbb	blt	ra,rb
010111 --101 aaaaa 010010 ----- bbbbb	bne	ra,rb
010111 --110 aaaaa 010010 ----- bbbbb	ble	ra,rb
010111 ----- ----- 010011 ----- bbbbb	br	rb
010111 ----- ----- 010100 ----- -----	UNUSED	
010111 ----- ----- 010101 ----- -----	UNUSED	
010111 sssss aaaaa 010110 0---0 bbbbb	xmem	rs,ra,rb
010111 sssss aaaaa 010110 1---0 bbbbb	xmem.usr	rs,ra,rb
010111 sssss aaaaa 010110 0---1 bbbbb	xmem	rs,ra[rb]
010111 sssss aaaaa 010110 1---1 bbbbb	xmem.usr	rs,ra[rb]
010111 ----- ----- 010111 ----- -----	doit	
010111 ddddd aaaaa 011000 000-0 bbbbb	ld	rd,ra,rb
010111 ddddd aaaaa 011000 00100 bbbbb	ld.bu	rd,ra,rb
010111 ddddd aaaaa 011000 00110 bbbbb	ld.b	rd,ra,rb
010111 ddddd aaaaa 011000 01000 bbbbb	ld.hu	rd,ra,rb
010111 ddddd aaaaa 011000 01010 bbbbb	ld.h	rd,ra,rb
010111 ddddd aaaaa 011000 100-0 bbbbb	ld.usr	rd,ra,rb
010111 ddddd aaaaa 011000 10100 bbbbb	ld.bu.usr	rd,ra,rb
010111 ddddd aaaaa 011000 10110 bbbbb	ld.b.usr	rd,ra,rb
010111 ddddd aaaaa 011000 11000 bbbbb	ld.hu.usr	rd,ra,rb
010111 ddddd aaaaa 011000 11010 bbbbb	ld.h.usr	rd,ra,rb
010111 ddddd aaaaa 011000 000-1 bbbbb	ld	rd,ra[rb]
010111 ddddd aaaaa 011000 00101 bbbbb	ld.bu	rd,ra[rb]
010111 ddddd aaaaa 011000 00111 bbbbb	ld.b	rd,ra[rb]
010111 ddddd aaaaa 011000 01001 bbbbb	ld.hu	rd,ra[rb]
010111 ddddd aaaaa 011000 01011 bbbbb	ld.h	rd,ra[rb]
010111 ddddd aaaaa 011000 100-1 bbbbb	ld.usr	rd,ra[rb]
010111 ddddd aaaaa 011000 10101 bbbbb	ld.bu.usr	rd,ra[rb]
010111 ddddd aaaaa 011000 10111 bbbbb	ld.b.usr	rd,ra[rb]
010111 ddddd aaaaa 011000 11001 bbbbb	ld.hu.usr	rd,ra[rb]
010111 ddddd aaaaa 011000 11011 bbbbb	ld.h.usr	rd,ra[rb]
010111 ddddd aaaaa 011001 ----- bbbbb	lda	rd,ra[rb]
010111 ddddd aaaaa 011010 ----- bbbbb	lda.h	rd,ra[rb]
010111 ----- ----- 011011 ----- -----	UNUSED	
010111 sssss aaaaa 011100 000-0 bbbbb	st	rs,ra,rb
010111 sssss aaaaa 011100 001-0 bbbbb	st.b	rs,ra,rb
010111 sssss aaaaa 011100 010-0 bbbbb	st.h	rs,ra,rb
010111 sssss aaaaa 011100 100-0 bbbbb	st.usr	rs,ra,rb
010111 sssss aaaaa 011100 101-0 bbbbb	st.b.usr	rs,ra,rb
010111 sssss aaaaa 011100 110-0 bbbbb	st.h.usr	rs,ra,rb
010111 sssss aaaaa 011100 000-1 bbbbb	st	rs,ra[rb]
010111 sssss aaaaa 011100 001-1 bbbbb	st.b	rs,ra[rb]

Opcode Bit Values	Instruction	
010111 sssss aaaaa 011100 010-1 bbbbb	st.h	rs,ra[rb]
010111 sssss aaaaa 011100 100-1 bbbbb	st.usr	rs,ra[rb]
010111 sssss aaaaa 011100 101-1 bbbbb	st.b.usr	rs,ra[rb]
010111 sssss aaaaa 011100 110-1 bbbbb	st.h.usr	rs,ra[rb]
010111 ----- ----- 011101 ----- -----	UNUSED	
010111 ----- ----- 011110 ----- -----	UNUSED	
010111 ----- ----- 011111 ----- -----	UNUSED	
010111 ddddd aaaaa 100000 ----- bbbbb	clr	rd,ra,rb
010111 ddddd aaaaa 100001 ----- bbbbb	set	rd,ra,rb
010111 ddddd aaaaa 100010 ----- bbbbb	ext	rd,ra,rb
010111 ddddd aaaaa 100011 ----- bbbbb	extu	rd,ra,rb
010111 ddddd aaaaa 100100 ----- bbbbb	mak	rd,ra,rb
010111 ddddd aaaaa 100101 ----- bbbbb	rot	rd,ra,rb
010111 ddddd ----- 100110 ----- bbbbb	ff0	rd,rb
010111 ddddd ----- 100111 ----- bbbbb	ff1	rd,rb
010111 ddddd aaaaa 101000 wwwww ooooo	clr	rd,ra,w5<o5>
010111 ddddd aaaaa 101001 wwwww ooooo	set	rd,ra,w5<o5>
010111 ddddd aaaaa 101010 wwwww ooooo	ext	rd,ra,w5<o5>
010111 ddddd aaaaa 101011 wwwww ooooo	extu	rd,ra,w5<o5>
010111 ddddd aaaaa 101100 wwwww ooooo	mak	rd,ra,w5<o5>
010111 ddddd aaaaa 101101 ----- ooooo	rot	rd,ra,<o5>
010111 ----- ----- 101110 ----- -----	UNUSED	
010111 ----- ----- 101111 ----- -----	UNUSED	
010111 ----- ----- 110000 ----- -----	rte	
010111 ----- ----- 110001 ----- -----	UNUSED	
010111 ----- ----- 110010 ----- -----	UNUSED	
010111 ----- ----- 110011 ----- -----	UNUSED	
010111 ----- ----- 110100 ----- -----	sync	
010111 ----- ----- 110101 ----- -----	sync.x	
010111 ddddd ----- 110110 ----- -----	mvbr	rd
010111 ----- aaaaa 110111 ----- -----	ldbr	ra
010111 ----- ----- 111000 ----- -----	UNUSED	
010111 ----- ----- 111001 ----- -----	UNUSED	
010111 ----- ----- 111010 ----- -----	UNUSED	
010111 ----- ----- 111011 ----- -----	UNUSED	
010111 ddddd ----- 111100 ccccc ccccc	getcr	rd,cr
010111 ddddd ----- 111101 ----- bbbbb	getcr	rd,rb
010111 ----- aaaaa 111110 ccccc ccccc	putcr	cr,ra
010111 ----- aaaaa 111111 ----- bbbbb	putcr	rb,ra
011000 ddddd aaaaa iiiiii iiiii iiiii	ld.bu	rd,ra,imm16
011001 ddddd aaaaa iiiiii iiiii iiiii	ld.b	rd,ra,imm16
011010 ddddd aaaaa iiiiii iiiii iiiii	ld.hu	rd,ra,imm16
011011 ddddd aaaaa iiiiii iiiii iiiii	ld.h	rd,ra,imm16
011100 ddddd aaaaa iiiiii iiiii iiiii	ld	rd,ra,imm16
011101 sssss aaaaa iiiiii iiiii iiiii	st.b	rs,ra,imm16
011110 sssss aaaaa iiiiii iiiii iiiii	st.h	rs,ra,imm16
011111 sssss aaaaa iiiiii iiiii iiiii	st	rs,ra,imm16
100000 ddddd aaaaa iiiiii iiiii iiiii	and.d	rd,ra,imm16
100001 ddddd aaaaa iiiiii iiiii iiiii	and.u.d	rd,ra,imm16
100010 ddddd aaaaa iiiiii iiiii iiiii	mask.d	rd,ra,imm16
100011 ddddd aaaaa iiiiii iiiii iiiii	mask.u.d	rd,ra,imm16
100100 ddddd aaaaa iiiiii iiiii iiiii	or.d	rd,ra,imm16
100101 ddddd aaaaa iiiiii iiiii iiiii	or.u.d	rd,ra,imm16
100110 ddddd aaaaa iiiiii iiiii iiiii	xor.d	rd,ra,imm16
100111 ddddd aaaaa iiiiii iiiii iiiii	xor.u.d	rd,ra,imm16

Opcode Bit Values	Instruction	
101000 ddddd aaaaa iiiiii iiiii iiiii	**add.d**	**rd,ra,imm16**
101001 ddddd aaaaa iiiiii iiiii iiiii	**addu.d**	**rd,ra,imm16**
101010 ddddd aaaaa iiiiii iiiii iiiii	**div.d**	**rd,ra,imm16**
101011 ddddd aaaaa iiiiii iiiii iiiii	**divu.d**	**rd,ra,imm16**
101100 ddddd aaaaa iiiiii iiiii iiiii	**mul.d**	**rd,ra,imm16**
101101 ddddd aaaaa iiiiii iiiii iiiii	**cmp.d**	**rd,ra,imm16**
101110 ddddd aaaaa iiiiii iiiii iiiii	**sub.d**	**rd,ra,imm16**
101111 ddddd aaaaa iiiiii iiiii iiiii	**subu.d**	**rd,ra,imm16**
110000 nnnnn aaaaa iiiiii iiiii iiiii	**bb0.d**	**n,ra,imm16**
110001 nnnnn aaaaa iiiiii iiiii iiiii	**bb1.d**	**n,ra,imm16**
110010 --001 aaaaa iiiiii iiiii iiiii	**bgt.d**	**ra,imm16**
110010 --010 aaaaa iiiiii iiiii iiiii	**beq.d**	**ra,imm16**
110010 --011 aaaaa iiiiii iiiii iiiii	**bge.d**	**ra,imm16**
110010 --100 aaaaa iiiiii iiiii iiiii	**blt.d**	**ra,imm16**
110010 --101 aaaaa iiiiii iiiii iiiii	**bne.d**	**ra,imm16**
110010 --110 aaaaa iiiiii iiiii iiiii	**ble.d**	**ra,imm16**
110011 iiiii iiiii iiiiii iiiii iiiii	**br.d**	**imm26**
110100 ddddd ----- iiiiii iiiii iiiii	**mvpc.d**	**rd,imm16**
110101 ----- ----- ------ --iii iiiii	**trap.d**	**imm8**
110110 sssss aaaaa iiiiii iiiii iiiii	**xmem.d**	**rs,ra,imm16**
110111 ddddd aaaaa 000000 ----- bbbbb	**and.d**	**rd,ra,rb**
110111 ddddd aaaaa 000001 ----- bbbbb	**and.c.d**	**rd,ra,rb**
110111 ----- ----- 000010 ----- -----	UNUSED	
110111 ----- ----- 000011 ----- -----	UNUSED	
110111 ddddd aaaaa 000100 ----- bbbbb	**or.d**	**rd,ra,rb**
110111 ddddd aaaaa 000101 ----- bbbbb	**or.c.d**	**rd,ra,rb**
110111 ddddd aaaaa 000110 ----- bbbbb	**xor.d**	**rd,ra,rb**
110111 ddddd aaaaa 000111 ----- bbbbb	**xor.c.d**	**rd,ra,rb**
110111 ddddd aaaaa 001000 00--- bbbbb	**add.d**	**rd,ra,rb**
110111 ddddd aaaaa 001000 01--- bbbbb	**add.o.d**	**rd,ra,rb**
110111 ddddd aaaaa 001000 10--- bbbbb	**add.i.d**	**rd,ra,rb**
110111 ddddd aaaaa 001000 11--- bbbbb	**add.io.d**	**rd,ra,rb**
110111 ddddd aaaaa 001001 00--- bbbbb	**addu.d**	**rd,ra,rb**
110111 ddddd aaaaa 001001 01--- bbbbb	**addu.o.d**	**rd,ra,rb**
110111 ddddd aaaaa 001001 10--- bbbbb	**addu.i.d**	**rd,ra,rb**
110111 ddddd aaaaa 001001 11--- bbbbb	**addu.io.d**	**rd,ra,rb**
110111 ddddd aaaaa 001010 ----- bbbbb	**div.d**	**rd,ra,rb**
110111 ddddd aaaaa 001011 ----- bbbbb	**divu.d**	**rd,ra,rb**
110111 ddddd aaaaa 001100 ----- bbbbb	**mul.d**	**rd,ra,rb**
110111 ddddd aaaaa 001101 ----- bbbbb	**cmp.d**	**rd,ra,rb**
110111 ddddd aaaaa 001110 00--- bbbbb	**sub.d**	**rd,ra,rb**
110111 ddddd aaaaa 001110 01--- bbbbb	**sub.o.d**	**rd,ra,rb**
110111 ddddd aaaaa 001110 10--- bbbbb	**sub.i.d**	**rd,ra,rb**
110111 ddddd aaaaa 001110 11--- bbbbb	**sub.io.d**	**rd,ra,rb**
110111 ddddd aaaaa 001111 00--- bbbbb	**subu.d**	**rd,ra,rb**
110111 ddddd aaaaa 001111 01--- bbbbb	**subu.o.d**	**rd,ra,rb**
110111 ddddd aaaaa 001111 10--- bbbbb	**subu.i.d**	**rd,ra,rb**
110111 ddddd aaaaa 001111 11--- bbbbb	**subu.io.d**	**rd,ra,rb**
110111 nnnnn aaaaa 010000 ----- bbbbb	**bb0.d**	**n,ra,rb**
110111 nnnnn aaaaa 010001 ----- bbbbb	**bb1.d**	**n,ra,rb**
110111 --001 aaaaa 010010 ----- bbbbb	**bgt.d**	**ra,rb**
110111 --010 aaaaa 010010 ----- bbbbb	**beq.d**	**ra,rb**
110111 --011 aaaaa 010010 ----- bbbbb	**bge.d**	**ra,rb**
110111 --100 aaaaa 010010 ----- bbbbb	**blt.d**	**ra,rb**
110111 --101 aaaaa 010010 ----- bbbbb	**bne.d**	**ra,rb**

Opcode Bit Values	Instruction
`110111 --110 aaaaa 010010 ----- bbbbb`	`ble.d        ra,rb`
`110111 ----- ----- 010011 ----- bbbbb`	`br.d         rb`
`110111 ----- ----- 010100 ----- -----`	UNUSED
`110111 ----- ----- 010101 ----- -----`	UNUSED
`110111 sssss aaaaa 010110 0---0 bbbbb`	`xmem.d       rs,ra,rb`
`110111 sssss aaaaa 010110 1---0 bbbbb`	`xmem.usr.d   rs,ra,rb`
`110111 sssss aaaaa 010110 0---1 bbbbb`	`xmem.d       rs,ra[rb]`
`110111 sssss aaaaa 010110 1---1 bbbbb`	`xmem.usr.d   rs,ra[rb]`
`110111 ----- ----- 010111 ----- -----`	UNUSED
`110111 ddddd aaaaa 011000 000-0 bbbbb`	`ld.d         rd,ra,rb`
`110111 ddddd aaaaa 011000 00100 bbbbb`	`ld.bu.d      rd,ra,rb`
`110111 ddddd aaaaa 011000 00110 bbbbb`	`ld.b.d       rd,ra,rb`
`110111 ddddd aaaaa 011000 01000 bbbbb`	`ld.hu.d      rd,ra,rb`
`110111 ddddd aaaaa 011000 01010 bbbbb`	`ld.h.d       rd,ra,rb`
`110111 ddddd aaaaa 011000 100-0 bbbbb`	`ld.usr.d     rd,ra,rb`
`110111 ddddd aaaaa 011000 10100 bbbbb`	`ld.bu.usr.d  rd,ra,rb`
`110111 ddddd aaaaa 011000 10110 bbbbb`	`ld.b.usr.d   rd,ra,rb`
`110111 ddddd aaaaa 011000 11000 bbbbb`	`ld.hu.usr.d  rd,ra,rb`
`110111 ddddd aaaaa 011000 11010 bbbbb`	`ld.h.usr.d   rd,ra,rb`
`110111 ddddd aaaaa 011000 000-1 bbbbb`	`ld.d         rd,ra[rb]`
`110111 ddddd aaaaa 011000 00101 bbbbb`	`ld.bu.d      rd,ra[rb]`
`110111 ddddd aaaaa 011000 00111 bbbbb`	`ld.b.d       rd,ra[rb]`
`110111 ddddd aaaaa 011000 01001 bbbbb`	`ld.hu.d      rd,ra[rb]`
`110111 ddddd aaaaa 011000 01011 bbbbb`	`ld.h.d       rd,ra[rb]`
`110111 ddddd aaaaa 011000 100-1 bbbbb`	`ld.usr.d     rd,ra[rb]`
`110111 ddddd aaaaa 011000 10101 bbbbb`	`ld.bu.usr.d  rd,ra[rb]`
`110111 ddddd aaaaa 011000 10111 bbbbb`	`ld.b.usr.d   rd,ra[rb]`
`110111 ddddd aaaaa 011000 11001 bbbbb`	`ld.hu.usr.d  rd,ra[rb]`
`110111 ddddd aaaaa 011000 11011 bbbbb`	`ld.h.usr.d   rd,ra[rb]`
`110111 ddddd aaaaa 011001 ----- bbbbb`	`lda.d        rd,ra[rb]`
`110111 ddddd aaaaa 011010 ----- bbbbb`	`lda.h.d      rd,ra[rb]`
`110111 ----- ----- 011011 ----- -----`	UNUSED
`110111 sssss aaaaa 011100 000-0 bbbbb`	`st.d         rs,ra,rb`
`110111 sssss aaaaa 011100 001-0 bbbbb`	`st.b.d       rs,ra,rb`
`110111 sssss aaaaa 011100 010-0 bbbbb`	`st.h.d       rs,ra,rb`
`110111 sssss aaaaa 011100 100-0 bbbbb`	`st.usr.d     rs,ra,rb`
`110111 sssss aaaaa 011100 101-0 bbbbb`	`st.b.usr.d   rs,ra,rb`
`110111 sssss aaaaa 011100 110-0 bbbbb`	`st.h.usr.d   rs,ra,rb`
`110111 sssss aaaaa 011100 000-1 bbbbb`	`st.d         rs,ra[rb]`
`110111 sssss aaaaa 011100 001-1 bbbbb`	`st.b.d       rs,ra[rb]`
`110111 sssss aaaaa 011100 010-1 bbbbb`	`st.h.d       rs,ra[rb]`
`110111 sssss aaaaa 011100 100-1 bbbbb`	`st.usr.d     rs,ra[rb]`
`110111 sssss aaaaa 011100 101-1 bbbbb`	`st.b.usr.d   rs,ra[rb]`
`110111 sssss aaaaa 011100 110-1 bbbbb`	`st.h.usr.d   rs,ra[rb]`
`110111 ----- ----- 011101 ----- -----`	UNUSED
`110111 ----- ----- 011110 ----- -----`	UNUSED
`110111 ----- ----- 011111 ----- -----`	UNUSED
`110111 ddddd aaaaa 100000 ----- bbbbb`	`clr.d        rd,ra,rb`
`110111 ddddd aaaaa 100001 ----- bbbbb`	`set.d        rd,ra,rb`
`110111 ddddd aaaaa 100010 ----- bbbbb`	`ext.d        rd,ra,rb`
`110111 ddddd aaaaa 100011 ----- bbbbb`	`extu.d       rd,ra,rb`
`110111 ddddd aaaaa 100100 ----- bbbbb`	`mak.d        rd,ra,rb`
`110111 ddddd aaaaa 100101 ----- bbbbb`	`rot.d        rd,ra,rb`
`110111 ddddd ----- 100110 ----- bbbbb`	`ff0.d        rd,rb`
`110111 ddddd ----- 100111 ----- bbbbb`	`ff1.d        rd,rb`

Opcode Bit Values	Instruction
110111 ddddd aaaaa 101000 wwwww ooooo	clr.d rd,ra,w5<o5>
110111 ddddd aaaaa 101001 wwwww ooooo	set.d rd,ra,w5<o5>
110111 ddddd aaaaa 101010 wwwww ooooo	ext.d rd,ra,w5<o5>
110111 ddddd aaaaa 101011 wwwww ooooo	extu.d rd,ra,w5<o5>
110111 ddddd aaaaa 101100 wwwww ooooo	mak.d rd,ra,w5<o5>
110111 ddddd aaaaa 101101 ----- ooooo	rot.d rd,ra,<o5>
110111 ----- ----- 101110 ----- -----	UNUSED
110111 ----- ----- 101111 ----- -----	UNUSED
110111 ----- ----- 110000 ----- -----	UNUSED
110111 ----- ----- 110001 ----- -----	UNUSED
110111 ----- ----- 110010 ----- -----	UNUSED
110111 ----- ----- 110011 ----- -----	UNUSED
110111 ----- ----- 110100 ----- -----	sync.d
110111 ----- ----- 110101 ----- -----	sync.x.d
110111 ddddd ----- 110110 ----- -----	mvbr.d rd
110111 ----- aaaaa 110111 ----- -----	ldbr.d ra
110111 ----- ----- 111000 ----- -----	UNUSED
110111 ----- ----- 111001 ----- -----	UNUSED
110111 ----- ----- 111010 ----- -----	UNUSED
110111 ----- ----- 111011 ----- -----	UNUSED
110111 ddddd ----- 111100 ccccc ccccc	getcr.d rd,cr
110111 ddddd ----- 111101 ----- bbbbb	getcr.d rd,rb
110111 ----- ----- 111110 ----- -----	UNUSED
110111 ----- ----- 111111 ----- -----	UNUSED
111000 ddddd aaaaa iiiiii iiiii iiiii	ld.bu.d rd,ra,imm16
111001 ddddd aaaaa iiiiii iiiii iiiii	ld.b.d rd,ra,imm16
111010 ddddd aaaaa iiiiii iiiii iiiii	ld.hu.d rd,ra,imm16
111011 ddddd aaaaa iiiiii iiiii iiiii	ld.h.d rd,ra,imm16
111100 ddddd aaaaa iiiiii iiiii iiiii	ld.d rd,ra,imm16
111101 sssss aaaaa iiiiii iiiii iiiii	st.b.d rs,ra,imm16
111110 sssss aaaaa iiiiii iiiii iiiii	st.h.d rs,ra,imm16
111111 sssss aaaaa iiiiii iiiii iiiii	st.d rs,ra,imm16

APPENDIX B

CONTEXT SWITCHING TEST

```
;;;
;;; This is a context switching test.  There are two processes, which alternate
;;; with each interrupt. When one of the processes finishes, they both stop.
;;;

.text

_stringA:
.ascii  "String A, value is %d\n\000"
.align  4


_stringB:
.ascii "Foo foo! %d\n\000"
.align  4


.global _main
_main:

        or.u    r2,r0,0x9000            ; generate an interrupt to start it off
        st      r0,r0,r2

;;;
;;; Here's the first process.
;;;

_main0:

        br      _printf                 ; gonna call printf
        or.u    r2,r0,hi16(_stringA)    ; to print a string
        or.u    r3,r0,0xA5A5            ; and a constant
        or      r2,r2,lo16(_stringA)
        or      r3,r3,0xA5A5
        mvpc.d  r28,_sync               ; call printf and quit when returning


;;;
;;; Here's the second process.
;;;


_main1:
        br      _printf                 ; gonna call printf
        or.u    r2,r0,hi16(_stringB)    ; to print a string
        or.u    r3,r0,0xB7B7            ; and a constant
        or      r2,r2,lo16(_stringB)
        or      r3,r3,0xB7B7
        mvpc.d  r28,_sync               ; call printf and quit when returning



;;;
;;; This is the interrupt handler. It should toggle between the two programs
;;; above. I have to initialize the second one beforehand.
;;;

.global __handle5
__handle5:

        ;;
        ;; Save the current state.
        ;;
        ;; c12 = original value of r28
        ;; c11 = base address to save state (unchanged)
        ;; c10 = modified
        ;; c9  = set to indicate what to save (unchanged):
        ;;          general registers and control registers are always saved
```

```
        ;;          if bit 0 is set, Shadow IW registers are saved
        ;;          if bit 1 is set, the Branch Queue is saved and then enabled
        ;;          if bit 2 is set, the R1 Queue is saved


        br      __savem                 ; gonna call the save routine
        putcr   c12,r28                 ; save original r28 value
        or      r28,r0,7                ; I want save everything
        putcr   c9,r28
        or.u    r28,r0,hi16(_where)     ; where do I keep the address?
        ld      r28,r28,lo16(_where)    ; where do I want the data to go?
        putcr   c11,r28
        mvpc.d   r28,.+4                ; call the save routine


        ;; Change the state address to the other process for next time

        or.u    r2,r0,hi16(_where)
        ld      r5,r2,lo16(_where)      ; r5 is where I just saved the state

        or      r3,r0,lo16(_store0)
        or.u    r3,r3,hi16(_store0)     ; point r3 at state0

        cmp     r4,r3,r5                ; did I just save it in state0?

        bb1.d   eq,r4,_justdid0         ; branch if I did


        ;; No, I just saved it in _state1

        br      _doit
        st.d    r3,r2,lo16(_where)      ; so next time use state0


_justdid0:                              ; yes, just wrote state0

        or      r3,r0,lo16(_store1)
        or.u    r3,r3,hi16(_store1)     ; point r3 at state1

        st      r3,r2,lo16(_where)      ; next time use state1

_doit:

        ;; Load other state now. The address is in r3
        ;;
        ;; c12 = modified
        ;; c11 = base address to restore state (unchanged)
        ;; c10 = modified
        ;; c9  = set to indicate what to save (unchanged):
        ;;          general registers and control registers are always saved
        ;;          if bit 0 is set, Shadow IW registers are saved
        ;;          if bit 1 is set, the Branch Queue is saved and then enabled
        ;;          if bit 2 is set, the R1 Queue is saved
        ;;
        ;; Return convention:
        ;;
        ;; c12 = original value of r28, which must be restored by caller.
        ;;

        br      __loadem                ; gonna call the load routine
        putcr   c11,r3                  ; point to place to get state
        mvpc.d  r28,.+4                 ; call the reload routine

        or.u    r28,r0,0x9000           ; generate another interrupt
        st      r0,r0,r28

        getcr   r28,c12                 ; restore r28 value
        rte



;;;
;;; Here's the storage space to use for each context.  The first one is blank,
;;; since it's going to hold the main process.  The second one has to be
;;; pre-initialized to start the second process.
```

```
;;;

.data

_where:                                ; address of where to save state
.long   _store0


_store0:

.long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; r0 - r15
.long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; r16 - r31
.long   0,0,0,0,0,0,0,0                 ; c1 - c8
.long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; Branch Queue contents
.long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; R1 Queue contents
.long   0,0,0,0,0                       ; Shadow IW slots
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0


_store1:

.long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; r0 - r15
.long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0x0000c000 ; r16 - r31

.long   112                             ; c1
.long   0                               ; c2
.long   0                               ; c3
.long   _main1                          ; c4
.long   0                               ; c5
.long   0                               ; c6
.long   0                               ; c7
.long   0                               ; c8

.long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; Branch Queue contents
.long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; R1 Queue contents
.long   0,0,0,0,0                       ; Shadow IW slots
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
.long   0,0,0,0,0
```

APPENDIX C

DYNAMIC INSTRUCTION PERCENTAGES

Instruction class	Benchmark						
	ackermann	cat	cbubble	cquick	ctowers	dhry	fact
arithmetic	**29.6%**	**12.5%**	**43.4%**	**43.2%**	**19.0%**	**21.8%**	**13.6%**
mul	0.0%	0.0%	0.8%	1.8%	0.0%	0.5%	0.0%
div	0.0%	0.0%	0.4%	0.9%	0.0%	0.2%	0.0%
other	29.6%	12.5%	42.3%	40.5%	19.0%	21.1%	13.6%
logic	**14.7%**	**31.2%**	**2.8%**	**10.0%**	**29.6%**	**15.6%**	**36.1%**
barrel shifter	1.6%	6.2%	0.0%	1.4%	4.8%	2.8%	26.9%
ff0, ff1	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
boolean	13.1%	25.0%	2.8%	8.5%	24.7%	12.7%	9.2%
control	**3.7%**	**12.5%**	**0.4%**	**1.7%**	**3.0%**	**3.7%**	**3.5%**
mvpc	3.7%	12.5%	0.4%	1.7%	3.0%	3.7%	3.5%
getcr	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
putcr	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mvbr	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
branch	**28.2%**	**31.2%**	**21.6%**	**18.3%**	**13.4%**	**22.9%**	**37.8%**
relative	24.3%	18.7%	21.2%	16.5%	10.3%	19.2%	34.2%
indirect	3.9%	12.5%	0.4%	1.7%	3.1%	3.7%	3.6%
taken	20.4%	25.0%	16.2%	12.8%	8.9%	13.7%	28.6%
not taken	7.8%	6.2%	5.4%	5.5%	4.5%	9.2%	9.2%
ldbr	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
memory	**23.8%**	**12.6%**	**31.7%**	**26.8%**	**35.0%**	**36.1%**	**9.0%**
loads	*11.6%*	*6.3%*	*20.8%*	*16.0%*	*18.9%*	*21.4%*	*4.4%*
word	8.1%	6.3%	20.8%	15.9%	17.2%	12.6%	4.0%
half	0.0%	0.0%	0.0%	0.0%	0.0%	0.3%	0.0%
byte	3.4%	0.0%	0.0%	0.1%	1.7%	8.5%	0.5%
stores	*12.2%*	*6.3%*	*10.5%*	*8.0%*	*16.1%*	*14.0%*	*4.6%*
word	9.3%	6.3%	10.5%	8.0%	14.6%	13.0%	4.3%
half	0.0%	0.0%	0.0%	0.0%	0.0%	0.3%	0.0%
byte	3.0%	0.0%	0.0%	0.1%	1.5%	0.6%	0.3%
lda	*0.0%*	*0.0%*	*0.4%*	*2.8%*	*0.0%*	*0.7%*	*0.0%*

Instruction class	Benchmark						
	grep	heapsort	mergesort	mod	muldiv	pi	queens
arithmetic	**24.5%**	**20.6%**	**33.6%**	**24.3%**	**22.9%**	**42.5%**	**39.4%**
mul	0.0%	0.0%	0.0%	0.3%	1.2%	2.3%	0.0%
div	0.0%	0.0%	0.0%	0.4%	1.2%	4.6%	0.0%
other	24.5%	20.6%	33.6%	23.5%	20.5%	35.6%	39.4%
logic	**24.9%**	**30.5%**	**16.5%**	**18.4%**	**23.0%**	**8.3%**	**12.1%**
barrel shifter	0.0%	4.8%	6.7%	8.0%	6.6%	2.4%	0.1%
ff0, ff1	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
boolean	24.9%	25.8%	9.7%	10.4%	16.4%	5.9%	12.0%
control	**3.5%**	**3.2%**	**1.6%**	**0.8%**	**2.5%**	**0.5%**	**0.6%**
mvpc	3.5%	3.2%	1.6%	0.8%	2.5%	0.5%	0.6%
getcr	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
putcr	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mvbr	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
branch	**27.4%**	**17.9%**	**24.3%**	**20.9%**	**17.7%**	**21.8%**	**29.4%**
relative	23.9%	14.7%	22.7%	17.3%	13.4%	21.3%	28.8%
indirect	3.5%	3.2%	1.6%	3.6%	4.3%	0.5%	0.6%
taken	17.0%	10.8%	18.4%	10.9%	13.4%	18.9%	16.6%
not taken	10.4%	7.1%	5.9%	10.0%	4.3%	2.9%	12.7%
ldbr	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
memory	**19.8%**	**27.8%**	**24.1%**	**35.6%**	**33.9%**	**26.8%**	**18.6%**
loads	*14.8%*	*18.5%*	*12.4%*	*20.5%*	*15.2%*	*16.3%*	*11.6%*
word	7.8%	18.4%	12.2%	7.0%	6.8%	10.8%	11.3%
half	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
byte	7.0%	0.2%	0.2%	13.5%	8.4%	5.5%	0.3%
stores	*4.8%*	*9.2%*	*11.4%*	*15.1%*	*17.6%*	*8.2%*	*4.4%*
word	1.0%	9.1%	11.3%	5.5%	12.2%	3.4%	4.0%
half	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
byte	3.9%	0.1%	0.2%	9.6%	5.4%	4.8%	0.3%
lda	*0.1%*	*0.0%*	*0.2%*	*0.0%*	*1.2%*	*2.3%*	*2.6%*

APPENDIX D

PORTING GNU SOFTWARE

The publicly available GNU assembler and linker were ported to the Fred architecture. There is little documentation included with the tools on how to do this, so the empirical procedures are explained here.

D.1. Porting the GNU Assembler

The assembler source was obtained as **`gas-2.0.tar.gz`** from the GNU archives at **`prep.ai.mit.edu`**. Since the Fred VHDL simulator was written to run on a Sun SPARCstation, it was decided that the simplest method of porting the assembler would be to treat Fred like a new processor in the Sun SPARC family and use as much as possible of the existing configuration for the SPARC, including the **`a.out.h`** header files and object formats.

The **`gas`** source tree was examined and FRED keywords added everywhere there were SPARC keywords. All the machine-specific files were duplicated and renamed Fred, until Makefiles for the assembler could be created by specifying either

```
configure sun4 --target=sun4
```

or

```
configure sun4 --target=fred
```

The changes required are shown in Table D.1 and Table D.2.

Table D.1 Changes to GNU assembler source files

Source file	Changes required
`gas/config/aout_gnu.h`	Added TC_FRED macros in all places where there were TC_SPARC macros.
`gas/config/tc-sparc.c`	This contained a bug in the way in which the BFD_RELOC_32 relocation types are handled. A problem was fixed in the **`tc-fred.c`** file, but it is unknown if the GNU people intended to fix this, since there may be some subtle reason why it is the way it is.
`gas/read.c`	Added TC_FRED macros in all places where there were TC_SPARC macros.
`bfd/configure.in`	The line **`fred-*-*)     bfd_target=sparc-aout`** was added to declare Fred as part of the Sun processor family.
`config.sub`	The line **`basic_machine=fred-sun`** was added to declare Fred as part of the Sun processor family.

Table D.2 Additions to GNU assembler source files

Source File	Contents
`gas/config/tc-fred.h`	This contains **`#defines`** for object file formats, relocation fixups, and other items which are needed to build and link. This is where the TC_FRED macro is defined.
`gas/config/tc-fred.c`	This is the actual parser for the assembly mnemonics. It's pretty ugly-looking spaghetti code, with several global variables that are modified as side-effects to functions outside this file. The parser works by matching substrings from the input line with the expected or allowed tokens specified for each instruction. When it has matched all the tokens, the opcode is complete. If there is a mismatch, it restarts with the next entry in the opcode table, until all possibilities for that instruction have been tried.

Within the main parser function **`fred_ip()`** in the file **`gas/config/tc-fred.c`** there are four string pointers which are used heavily by the parser. These pointers are described in Table D.3.

D.2. Porting the GNU Linker

Unfortunately, the assembler by itself cannot do everything. It is correct as far as it goes, but the linker must be able to handle Fred's 16-bit immediate values correctly. An attempt was made to fool the existing SPARC linker into correctly linking Fred object code. The SPARC linker has RELOC_LO10 and RELOC_HI22 types for splitting 32-bit immediate values into the lower 10-bit part and the upper 22-bit part. It also has a 30-bit PC-relative branch instruction, instead of Fred's 26-bit branch. Since Fred does not need some of the SPARC relocation types, the existing SPARC values for RELOC_WDISP30, RELOC_WDISP22, RELOC_HI22, and RELOC_LO10 were simply reused in both the assembler and linker to supply Fred-specific functions.

Unfortunately, both RELOC_LO16 and RELOC_HI16 types were needed to convert immedi-

Table D.3 Important parser variables

Variable	Usage
`char *str`	Points to the current input line.
`char *s`	Points to the operands within the input line. It is updated as operands are consumed.
`char *argsStart`	Tracks the original start of the input operands so that **`*s`** can be reset if needed.
`char *args`	Tracks the **`args`** field of the **`fred_opcode`** structure, moving along as each token is recognized.

ate values into equal-sized 16-bit chunks. If only a RELOC_16 is used in the assembler, then a 32-bit address is truncated into the lower 16-bits and placed at the upper 16 bits of the instruction. It may be possible to fiddle with the index field to put it in the right place, but it cannot be forced to use the upper 16 bits for the RELOC_HI16. Likewise, RELOC_OFF16 and RELOC_OFF26 types are needed to handle Fred's 16- and 26-bit branch offsets. It was necessary to port a linker for Fred also.

`binutils-1.9` was obtained from **`prep.ai.mit.edu`**, to get the source for **`ld`**. The source was placed into its own directory and modified to deal with Fred's relocation types. It was only necessary to change the **`reloc_target_bitsize`** and **`reloc_target_rightshift`** values in **`ld.c`**. These variables are documented in the source code, but basically they determine how many bits are used for the data in question and how many bits to shift it before using it.

The linker's default output format was modified to produce "O MAGIC" executable files, which makes the text and data segments contiguous. This was done to prevent the segments from being aligned on separate page boundaries, so that the simulator memory could be smaller. In the SPARC core image, the first page is always skipped, so that references through a NULL pointer can be detected. In the Fred core image, this first page is used to hold the exception branch table.

The SPARC version of the linker (and therefore Fred's version too) uses Sun's **`a.out.h`** header file to define the relocation types with an enum. This probably means that the assembler and linker must be compiled on SPARC machines. To make everything portable, it should only be necessary to modify the relocation type definitions (in **`config/aout_gnu.h`** for the assembler), but it would have been difficult and was not necessary, so it was not done.

REFERENCES

[1] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An instruction issuing approach to enhancing performance in multiple functional unit processors," *IEEE Transactions on Computers*, vol. C-35, pp. 815–828, September 1986.

[2] E. Brunvand, "A cell set for self-timed design using actel FPGAs," Technical Report UUCS–91–013, University of Utah, 1991.

[3] E. Brunvand, "Using FPGAs to prototype a self-timed computer," in *International Workshop on Field Programmable Logic and Applications*, (Vienna University of Technology), September 1992.

[4] E. Brunvand, "The NSR processor," in *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, (Maui, Hawaii), pp. 428–435, January 1993.

[5] K.-R. Cho, K. Okura, and K. Asada, "Design of a 32-bit fully asynchronous microprocessor (FAM)," in *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, (Washington, D.C.), pp. 1500–1503, 1992.

[6] W. A. Clark and C. A. Molnar, "Macromodular system design," Tech. Rep. 23, Computer Systems Laboratory, Washington University, April 1973.

[7] A. Davis, "The architecture and system method for DDM1: A recursively structured data-driven machine," in *5th Annual Symposium on Computer Architecture*, (Palo Alto, CA), pp. 210–215, April 1978.

[8] A. L. Davis, "Asynchronous advantages often cited and NOT often cited." Distributed at the Async94 Conference, Salt Lake City, Utah., November 1994.

[9] M. E. Dean, "STRiP: A self-timed RISC processor," Technical Report CSL-TR-92-543, Computer Systems Laboratory, Stanford University, Stanford, CA 94305–4055, July 1992.

[10] H. Dwyer and H. C. Torng, "An out-of-order superscalar processor with speculative execution and fast, precise interrupts," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, (Portland, OR), pp. 272–281, December 1992.

[11] D. Elsner, J. Fenlason, and friends, *Using* **`as`** *– The GNU Assembler*. Free Software Foundation, Inc., 675 Massachusetts Avenue, Cambridge, MA 02139, 1.13 ed., November 1992.

[12] C. J. Elston, D. B. Christianson, P. A. Findlay, and G. B. Steven, "Hades—towards the design of an asynchronous superscalar processor," in *Second Working Conference on Asynchronous Design Methodologies*, (London, UK), pp. 200–209, May 1995.

[13] P. B. Endecott, *SCALP: A Superscalar Asynchronous Low-Power Processor*. PhD thesis, University of Manchester, 1995. **`ftp://ftp.cs.man.ac.uk/pub/amulet/theses/endecott_phd.ps.gz`**.

[14] M. K. Farrens and A. R. Pleszkun, "Implementation of the PIPE processor," *IEEE Computer*, pp. 65–70, January 1991.

[15] M. Farrens, P. Ng, and P. Nico, "A comparison of superscalar and decoupled access/execute architectures," in *Proceedings of the 26th Annual ACM/IEEE International Sympo-*

sium on Microarchitecture, (Austin, Texas), IEEE, ACM, December 1993.

[16] M. K. Farrens and A. R. Pleszkun, "Improving performance of small on-chip instruction caches," in *14th Annual International Symposium on Computer Architecture*, (Pittsburgh, PA), pp. 234–241, ACM, 1987.

[17] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods, "A micropipelined ARM," in *Proceedings of the VII Banff Workshop: Asynchronous Hardware Design*, (Banff, Canada), August 1993.

[18] S. Furber, "Computing without clocks," in *Proceedings of the VII Banff Workshop: Asynchronous Hardware Design*, (Banff, Canada), August 1993.

[19] J. R. Goodman, J. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: A VLSI decoupled architecture," in *12th Annual International Symposium on Computer Architecture*, (Boston, MA), pp. 20–27, IEEE Computer Society, June 1985.

[20] T. R. Gross, J. L. Hennessy, S. A. Przybylski, and C. Rowen, "Measurement and evaluation of the MIPS architecture and processor," *ACM Transactions on Computer Systems*, vol. 6, pp. 229–257, August 1988.

[21] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, and J. Gill, "Hardware/software tradeoffs for increased performance," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, (Palo Alto, CA), pp. 2–11, ACM, April 1982.

[22] D. Hunt, "Advanced performance features of the 64-bit PA-8000," in *Proceedings of COMPCON'95*, (San Francisco, CA), pp. 123–128, 1995.

[23] M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*. MIT Press, 1985.

[24] A. Martin, S. Burns, T. Lee, D. Borkovic, and P. Hazewindus, "The design of an asynchronous microprocessor," in *Proceedings of the CalTech Conference on VLSI*, (Pasadena, CA), 1989.

[25] Motorola, *System V Application Binary Interface: Motorola 88000 Processor Supplement*. Unix Press, 1988.

[26] Motorola, *MC88100 RISC Microprocessor User's Manual*. Englewood Cliffs, New Jersey 07632: Prentice Hall, 2nd ed., 1990.

[27] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura, "TITAC: Design of quasi-delay-insensitive microprocessor," *IEEE Design & Test of Computers*, vol. 11, pp. 50–63, Summer 1994.

[28] A. Nicolau and J. A. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Transactions on Computers*, vol. C-33, pp. 110–118, November 1984.

[29] N. C. Paver, *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, University of Manchester, 1994. **http://www.cs.man.ac.uk/amulet/publications/thesis/paver94_phd.html**.

[30] A. R. Pleszkun, J. R. Goodman, W.-C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. B. Schechter, "WISQ: A restartable architecture using queues," in *1987 Symposium on Com-*

puter Architecture, (Pittsburgh, PA), pp. 290–299, ACM, June 1987.

[31] W. F. Richardson, “The Fred VHDL model,” Technical Report UUCS–95–021, University of Utah, November 1995. **ftp://ftp.cs.utah.edu/techreports/1995/UUCS-95-021.ps.Z**.

[32] W. F. Richardson and E. Brunvand, “The NSR processor prototype,” Technical Report UUCS–92–029, University of Utah, August 1992. **ftp://ftp.cs.utah.edu/techreports/1992/UUCS-92-029.ps.Z**.

[33] W. F. Richardson and E. Brunvand, “Fred: An architecture for a self-timed decoupled computer,” Technical Report UUCS–95–008, University of Utah, May 1995. **ftp://ftp.cs.utah.edu/techreports/1995/UUCS-95-008.ps.Z**.

[34] W. F. Richardson and E. Brunvand, “Precise exception handling for a self-timed processor,” in *1995 International Conference on Computer Design: VLSI in Computers & Processors*, (Los Alamitos, CA), pp. 32–37, IEEE Computer Society Press, October 1995.

[35] C. L. Seitz, “System timing,” in *Mead and Conway, Introduction to VLSI Systems*, ch. 7, Addison-Wesley, 1980.

[36] J. E. Smith and A. R. Pleszkun, “Implementing precise interrupts in pipelined processors,” *IEEE Transactions on Computers*, vol. 37, pp. 562–573, May 1988.

[37] G. S. Sohi, “Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers,” *IEEE Transactions on Computers*, vol. 39, pp. 349–359, March 1990.

[38] R. F. Sproull and I. E. Sutherland, “Counterflow pipeline processor architecture,” Tech. Rep. SMLI TR-94-25, Sun Microsystems Laboratories, Inc., M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, April 1994. **http://www.sun.com/smli/technical-reports/1994/smli_tr-94-25.ps**.

[39] I. Sutherland, “Micropipelines,” *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[40] C. A. Thekkath and H. M. Levy, “Hardware and software support for efficient exception handling,” in *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA), pp. 110–119, ACM Press, October 1994.

[41] J. A. Tierno, A. J. Martin, D. Borkovic, and T. K. Lee, “An asynchronous microprocessor in gallium arsenide,” Technical Report CS–TR–93–38, California Institute of Technology, Department of Computer Science, Pasadena, CA 91125, 1993.

[42] H. C. Torng and M. Day, “Interrupt handling for out-of-order execution processors,” *IEEE Transactions on Computers*, vol. 42, pp. 122–127, January 1993.

[43] D. W. Wall, “Limits of instruction-level parallelism,” WRL Technical Note TN-15, Digital Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA 94301, December 1990. **ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TN-15.ps**.

[44] C.-J. Wang and F. Emnett, “Area and performance comparison of pipelined RISC processors implementing different precise interrupt methods,” in *1993 International Conference on Computer Design*, (Cambridge, MA), pp. 102–105, IEEE Computer Society Press,

October 1993.

[45] W. A. Wulf, "The WM computer architecture," *Computer Architecture News*, vol. 16, March 1988.