Fred: An Architecture for a Self-Timed Decoupled Computer

William F. Richardson
Computer Science Department
University of Utah
Salt Lake City, UT 84112
willrich@cs.utah.edu

Erik Brunvand
Computer Science Department
University of Utah
Salt Lake City, UT 84112
elb@cs.utah.edu

Abstract

Decoupled computer architectures provide an effective means of exploiting instruction level parallelism. Self-timed micropipeline systems are inherently decoupled due to the elastic nature of the basic FIFO structure, and may be ideally suited for constructing decoupled computer architectures. Fred is a self-timed decoupled, pipelined computer architecture based on micropipelines. We present the architecture of Fred, with specific details on a micropipelined implementation that includes support for multiple functional units and out-of-order instruction completion due to the self-timed decoupling.

1. Introduction

As computer systems have grown in size and complexity, the difficulty in synchronizing the system components has also grown. 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, while 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 signals proceeding in the correct sequence. However, these circuits can be very difficult to design and debug without some additional structure to help the designer deal with the complexity. While there are many different asynchronous methodologies, one of the simplest to design, test, and debug is the self-timed micropipeline approach described by Sutherland [19], which avoids clock-related timing problems by enforcing a simple communication protocol between circuit elements. This is quite different from traditional synchronous signaling conventions where signal events occur at specific times and must 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.

Experience has shown the difficulty of writing parallel programs, yet most sequential programs have an (arguably) significant amount of instruction-level parallelism [13,23][1]. One way of exploiting this parallelism is by decoupling the memory access portion of an instruction stream from the execution portion [7,24,5]. By performing the two operations independently, peaks and valleys in each may be smoothed, resulting in an overall performance gain.

Although 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 even more serious problems.

The micropipeline approach is based on simple, self-timed, elastic, 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

1. Nicolau claims there is lots of parallelism available. Wall claims there's some, but not much.

increased in this type of system organization, without the overhead of a global controller keeping track of the state of the decoupled components. This should allow increased performance due to the increased decoupling and potentially faster local control of the components, however it also means that exception handling must be considered carefully. Because each of the elements is running at its own rate, and data are possibly being transmitted through FIFO queues when the exception is signaled, care must be taken to make sure that the machine can process an exception in a functionally precise way without losing state that might be in the process of being modified by a different component.

Fred[2] is a self-timed decoupled, pipelined processor architecture based on micropipelines. We present the architecture of Fred, with specific details on a micropipelined implementation that includes support for out-of-order instruction completion due to the decoupling, and a model for functionally precise exception processing.

2. Asynchronous Processors

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 70's at Washington University [3] and the self-timed dataflow machines built at the University of Utah in the late 70's [4].

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. Some recent processor projects include the following:

2.1 The CalTech Asynchronous Microprocessor

The first asynchronous VLSI processor was built by Alain Martin's group at CalTech [11]. It is completely asynchronous, using (mostly) delay-insensitive circuits and dual-rail data encoding. 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. It has been fabricated both in CMOS and GaAs [20].

2.2 The NSR

The NSR (Non-Synchronous RISC) processor [2,15] is structured as a five-stage pipeline where each pipe stage operates concurrently and communicates over self-timed data channels in the style of micropipelines. 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 built using FPGAs. It is pipelined and decoupled, but doesn't handle exceptions. It is a simple 16-bit processor with only sixteen instructions, since it was built partially as an exercise in using FPGAs for rapid prototyping of self-timed circuits [1].

2.3 The Amulet

A group at Manchester has built a self-timed micropipelined VLSI implementation of the ARM processor [6] which is an extremely power-efficient commercial microprocessor. The Amulet is a real processor in the sense that it mimics the behavior of an existing commercial processor and it handles simple exceptions. It is more deeply pipelined than the synchronous ARM, but it is not decoupled (although it does allow instruction prefetching), and its precise exception model is a simple one. The Amulet has been designed and fabricated. The performance of the first-generation design is within a factor of two of the commercial version [14]. Future versions of Amulet are expected to close this gap.

2.4 The Counterflow Pipeline Processor

This is an innovative architecture proposed by a group at Sun Microsystems Labs [18]. 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. 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 has been simulated. The potential of this architecture is intriguing, but still unknown.

3. The Fred Architecture

The Fred architecture is based roughly on the NSR

2. Fred is not an acronym, and it doesn't mean anything. It's just a name, like "SPARC" or "Alpha."

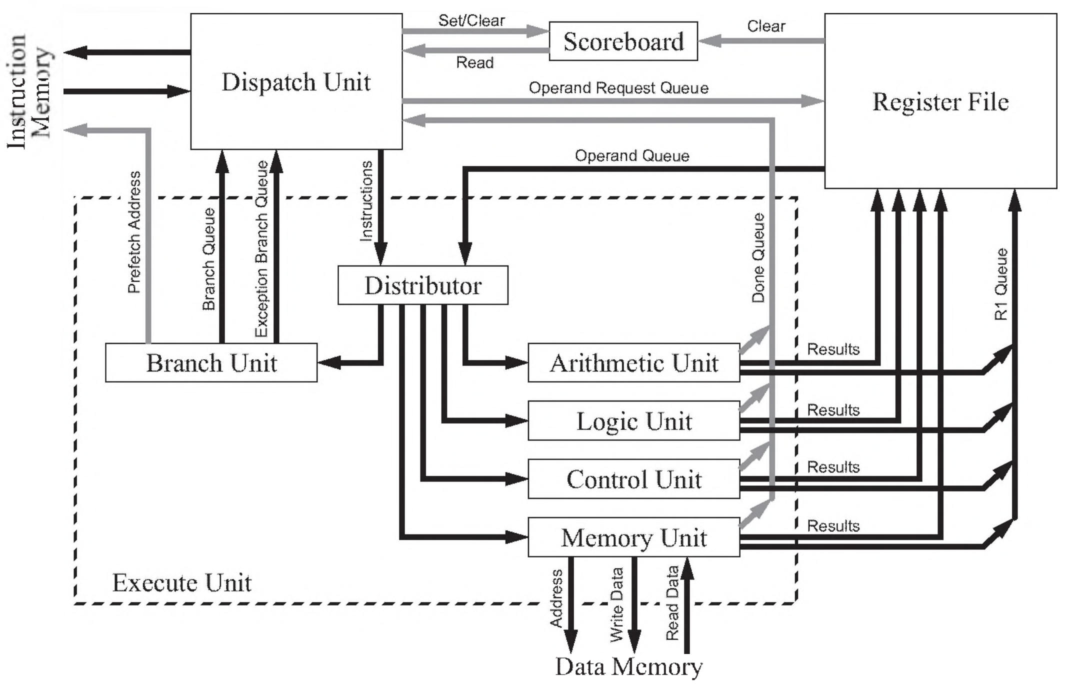

Figure 1. Fred block diagram
Black lines are primary data paths; gray lines are control paths.
All data and control paths are pipelined queues.

architecture developed at the University of Utah [2,15]. As such it consists of several decoupled independent processes connected by FIFO queues of various lengths, an approach which we believe offers a number of advantages over a clocked synchronous organization. The Fred architecture specifies the instruction set and the general layout and behavior of the processor. Other 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. The details of the exception handling mechanism is not specified by the architecture, but some means must be provided.

A prototype of Fred has been implemented in a detailed VHDL model. Figure 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 1, may be pipelined to any desired depth without affecting the results of the computation. Because Fred uses self-timed micropipelines [19] 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, the corresponding resource contention is avoided.

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 and thus the instruction that uses that circuit finishes faster.

The VHDL version chooses particular implementations for each of the main pieces of Fred. For example, the Dispatch unit is organized so as to dynamically reorder instructions for issue, allowing instructions to be issued out of order, and to complete in yet a different order. This is of particular interest in a self-timed processor where the multiple functional units might take varying amounts of time to compute a result. An individual functional unit might even take different amounts of time to compute a result based on the data, which will lead naturally to out of order instruction completion. The VHDL prototype is fully operational, and includes a functionally precise exception model [16]. The timing and configuration parameters can be adjusted for each component of the design.

4. Instruction Set

Choosing an instruction set for a RISC processor can be a complex task [9,8,10]. Rather than attempt to design a new instruction set from scratch, an instruction set from an existing commercial RISC processor was adapted. Much of the Fred instruction set is taken directly from the Motorola 88100 instruction set [12]. However, Fred does not implement all the 88100 instructions, and several of Fred's instructions do not correspond to any instructions of the 88100. The instructions, and the functional units that execute them, are shown in Figure 2.

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

Figure 2. Fred instruction set

5. Instruction Dispatch

Instruction Dispatch 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. Instructions are issued as soon as all dependencies are satisfied, without further regard to program order. Because different functional units may take different amounts of time to complete, individual instructions may complete in a different order than which they were issued.

Deadlocking the processor is theoretically possible. Because both the R1 Queue and Branch Queue (see below) 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 take an exception before an instruction sequence is issued that would result in deadlock. Obviously, there is no way to handle such an exception except by aborting the current user program. Deadlock is only possible due to programmer error, and Fred can detect and abort the illegal instruction sequence before it takes effect.

5.1 The Instruction Window

An Instruction Window (IW) is used to buffer incoming instructions and to track the status of issued instructions [22]. A register scoreboard is used to avoid all data hazards. 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. The scoreboard is also maintained in the Dispatch unit, and is cleared when results arrive at the Register File.

Instructions are issued from the IW when all their data dependencies are satisfied (including WAW dependencies). Issuing an instruction does not remove it from the IW. Instead, instructions are removed from the IW only after they have completed successfully. 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 IW. The status is usually an indication that the instruction completed successfully, but is also used to report exceptions. 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. When an instruction is unsuccessful, it returns an exception status to the IW, which then begins exception processing. Instructions which can never cause exceptions do not have to report their status, and can be removed from the IW when they are dispatched. Because instructions may complete out-of-order, recoverable exceptions can cause unforeseen WAW hazards. The Instruction Window contains enough information to resolve these issues.

The Dispatch unit uses the Instruction Window and scoreboard to determine when to issue new instructions to the rest of the machine. When instruction issue occurs, the required operands are requested from the Register File (possibly through a FIFO), and the instruction is issued to the Execute unit (also possibly through a FIFO).

5.2 Exceptions

The exception model seen by the programmer is not that of a single point where the exception occurred. Instead, the Instruction Window holds a set of instructions which were in progress when the exception occurred. The hardware guarantees that this set (unless empty) will consist only of instructions which either faulted or which had been fetched but not yet issued when the exception occurred. The instructions in this set are a subset of a sequential portion of the dynamic program instructions, where the missing elements are those instructions which completed successfully out of order, and which do not need to be re-issued. Because the total state of the processor is not available at one known time (such as on a clock tick), the details of the exception handling are somewhat complicated, but no more so than for a synchronous processor that is deeply pipelined and may issue or complete instructions out of order. This is described in more detail elsewhere [16].

6. 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 a data memory pipeline similar to that used in the WM machine [24]. 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*. Data from any of the functional units may be queued into the R1 Queue, and loads from memory can also be 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. This is similar to having as many load delay slots as desired and allowed by the program structure. Note also that 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 which write to the R1 Queue are forced to complete in-order, to provide deterministic behavior.

7. Branch Decoupling

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, we gain the ability to prefetch instructions effectively. 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.

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, as shown in Figure 3. Because the **doit** is consumed entirely within the Dispatch Unit, it will take effect as soon as the branch target data is 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 **doit** instruction does not have to be explicitly specified. To prevent extra instruction fetches, the **doit** instruction can be encoded implicitly by a single bit available in the opcode of other instructions. The **doit** is implicit in Figure 3B. Figure 4 shows an example, based on the code in Figure 3B. Note that instructions may continue to be issued out-of-order, even with respect to the delay slot instructions. Note also that the **doit** may be consumed independently of the instruction which encodes it.

```
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.d      r2,r9,2
```

B. Reordered, with implicit doit

Figure 3. Two ways of ordering the same program segment

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 4. Branch prefetching in the IW

This two-part 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. It also allows other interesting behaviors such as achieving the effect of loop unrolling without increasing code size. This can be accomplished by computing several branch targets at one time and putting them in the branch queue before executing the loop code[3].

8. Independent Functional Units

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. 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 shown in Figure 2. The Distributor and its associated functional units collectively make up the Execute unit.

The Memory unit is treated as just another functional unit. The only difference is that the Memory unit sometimes produces data that is written to the data memory rather than the Register File.

Each of the functional units may produce results that are written back to the register file directly, or which are made available through the R1 Queue. In addition to reusing a result within single functional unit, in many processors a result may be forwarded directly to another functional unit without passing through a register, so that pipeline delays involved in writing to the register file are avoided. Forwarding results between independent functional units requires either a common shared bus as in Tomasulo's algorithm [21], or dedicated data paths as used in the DEC Alpha [17] and other high-performance processors. Fred does not forward results directly between functional units, because of the complexity involved. However, reusing the last result of a computation within a single functional unit is certainly possible. Trace data suggests that such reuse may provide a measurable performance increase, but it is highly dependent on the compiler technology.

9. Register File

The Register File responds to requests from the Dispatch unit for operands which it delivers through a FIFO to the Execute unit. These operands are paired with instructions and passed to the appropriate functional unit. Because the operands are requested in the same order as instructions are issued, there is no matching required to determine which operands should be paired with which instructions. They emerge from the FIFO queues in the correct sequence.

3. This is not true loop unrolling since the registers are not recolored, but it could be useful.

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 as described earlier, but there is no actual register associated with the R1 Queue. Instead, the Dispatch unit clears the scoreboard bit for register r1 when the producing instruction completes successfully.

10. Results

Several benchmarks have been run 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 Figure 5.

Program name	Dynamic instruction count	Description
ackermann	1660	recursion
cat	7109	copy stdin to stdout, for "cat.c"
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

Figure 5. Benchmark programs

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 a custom post-processor. 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. This is entirely due to the fact that the compiler is not targeted specifically for Fred, and has nothing to do with any shortcomings of the Fred architecture.

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. 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 queues for maximum performance on a given benchmark.

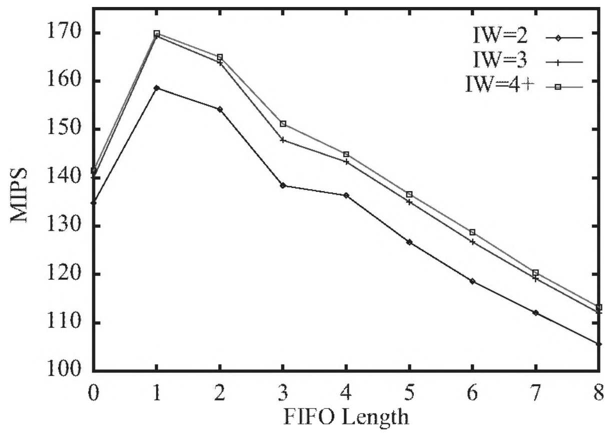

Figure 6. Average performance vs. IW size

The average performance is more dependent on the length of the FIFO queues than on the size of the Instruction Window. There was no appreciable difference in performance for IW sizes greater than 3 slots. Figure 6 shows the relationship between 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, it only affects performance in terms of saving state during exception handling.

10.1 Instruction Window Usage

Figure 7 shows how the average IW usage varied with queue length and IW size. With longer queue lengths the time needed for each issued instruction to complete is longer, 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 2.5 slots. 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

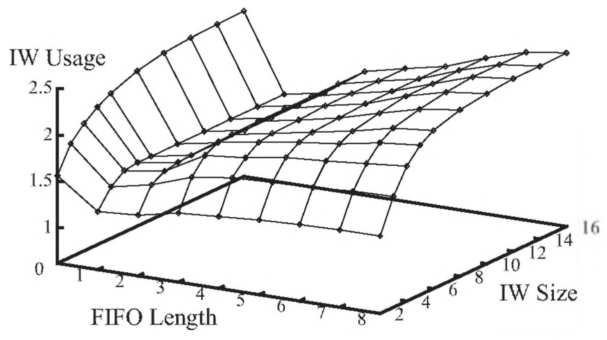

Figure 7. Average IW slot usage

is no storage in the queues, there is essentially no pipelining except for those instructions which can be sent to separate functional units.

10.2 Instruction Completion

Only those instructions which might possibly fail must report their completion to the Dispatch Unit. This enables a significant speedup in performance, since there is less communication with the Instruction Window. Instructions which will *always* complete successfully may be removed from the IW as soon as they have dispatched, providing a corresponding decrease in the average IW usage. Figure 8 and Figure 9 tabulate the differences for an optimal queue length of 1 and an IW size of 4 slots. On average, intelligent completion increases the performance by about 12%, while decreasing IW usage by about 20%.

Benchmark	MIPS			Percent reporting
	Forced	Optional	Increase	
ackermann	151.82	173.45	14.2%	24.1%
cat	152.77	178.18	16.6%	13.3%
cbubble	146.80	164.34	11.9%	31.8%
cquick	152.22	173.49	14.0%	25.2%
ctowers	157.19	176.96	12.6%	34.9%
dhry	148.87	164.69	10.6%	42.1%
fact	143.65	167.12	16.3%	9.9%
grep	146.48	168.27	14.9%	20.1%
heapsort	152.54	172.82	13.3%	28.0%
mergesort	148.14	168.91	14.0%	24.2%
mod	148.87	165.66	11.3%	36.0%
muldiv	152.08	170.53	12.1%	33.9%
pi	145.58	163.43	12.3%	29.3%
queens	148.40	171.19	15.4%	16.6%
average	**149.67**	**169.93**	**13.5%**	**26.4%**

Figure 8. Completion signalling and performance

Benchmark	IW slot usage with forced completion	IW slot usage with optional completion	Reduction
ackermann	2.00	1.42	29.0%
cat	1.82	1.15	36.8%
cbubble	2.11	1.71	19.0%
cquick	2.17	1.65	24.0%
ctowers	2.20	1.70	22.7%
dhry	2.09	1.71	18.2%
fact	1.80	1.16	35.6%
grep	1.89	1.35	28.6%
heapsort	2.08	1.63	21.6%
mergesort	2.04	1.55	24.0%
mod	2.24	1.82	18.8%
muldiv	2.20	1.70	22.7%
pi	2.21	1.77	19.9%
queens	1.93	1.39	28.0%
average	**2.06**	**1.56**	**24.9%**

Figure 9. Completion signalling and IW slot usage

10.3 Branch Decoupling

As mentioned earlier, Fred's decoupled branch mechanism allows for a variable number of delay slots but the compiler used for the benchmarks generates code for the Motorola 88100 processor, a synchronous RISC processor which has only a single delay slot. This allows only 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 a basic block. Despite these handicaps, the average number of useful delay slot instructions 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**, and would also benefit from such a compiler. The dynamic separation results are shown in Figure 10.

11. Conclusions

The current prototype of Fred is in the form of a detailed VHDL model. This model is completely functional including the out-of-order instruction completion and functionally precise exceptions. Benchmark results seem to bear out the premise that a self-timed implementation is a natural match for decoupled computer architectures. The ability to allow different parts of the machine to proceed at their own rate and the natural use of self-timed FIFO queues enhances the decoupling due to the architec-

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**

Figure 10. Dynamic branch/doit separation

ture. As general processor designs (both synchronous and asynchronous) grow more complex and the degree of concurrency and decoupling increases, the features and techniques found in the Fred architecture—functionally precise interrupts, decoupled branches, intelligent prefetching, decoupled memory access, etc.—may gain in importance.

12. References

[1] Erik Brunvand. Using FPGAs to prototype a self-timed computer. In *International Workshop on Field Programmable Logic and Applications*, Vienna University of Technology, September 1992.

[2] Erik Brunvand. The NSR processor. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 428–435, Maui, Hawaii, January 1993.

[3] Wesley A. Clark and Charles A. Molnar. Macromodular system design. Technical Report 23, Computer Systems Laboratory, Washington University, April 1973.

[4] A.L. Davis. The architecture and system method for DDM1: A recursively structured data-driven machine. In *5th Annual Symposium on Computer Architecture*, April 1978.

[5] Matthew Farrens, Pius Ng, and Phil Nico. A comparison of superscalar and decoupled access/execute architectures. In *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, Austin, Texas, December 1993. IEEE,ACM.

[6] 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.

[7] 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*, pages 20–27. IEEE Computer Society, June 1985.

[8] Thomas R. Gross, John L. Hennessy, Stephen A. Przybylski, and Christopher Rowen. Measurement and evaluation of the MIPS architecture and processor. *ACM Transactions on Computer Systems*, 6(3):229–257, August 1988.

[9] John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross, and John Gill. Hardware/software tradeoffs for increased performance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM, April 1982.

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

[11] Alain Martin, Steven Burns, T.K. Lee, Drazen Borkovic, and Pieter Hazewindus. The design of an asynchronous microprocessor. In *Proc. CalTech Conference on VLSI*, 1989.

[12] Motorola. *MC88100 RISC Microprocessor User's Manual*. Prentice Hall, Englewood Cliffs, New Jersey 07632, second edition, 1990.

[13] Alexandru Nicolau and Joseph A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, C-33(11):110–118, November 1984.

[14] Nigel Charles 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**.

[15] William F. Richardson and Erik 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**.

[16] William F. Richardson and Erik Brunvand. Precise exception handling for a self-timed processor. In *1995 International Conference on Computer Design: VLSI in Computers & Processors*, pages 32–37, Los Alamitos, CA, October 1995. IEEE Computer Society Press.

[17] James E. Smith and Shlomo Weiss. Powerpc 601 and alpha 21064: A tale of two RISCs. *IEEE Computer*, 27(6):46–58, June 1994.

[18] Robert F. Sproull and Ivan E. Sutherland. Counterflow pipeline processor architecture. Technical Report 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**.

[19] Ivan Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[20] José A. Tierno, Alain J. Martin, Drazen Borkovic, and Tak Kwan Lee. A 100-MIPS GaAs asynchronous microprocessor. *IEEE Design & Test of Computers*, 11(2):43–49, Summer 1994.

[21] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[22] H. C. Torng and Martin Day. Interrupt handling for out-of-order execution processors. *IEEE Transactions on Computers*, 42(1):122–127, January 1993.

[23] David 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**.

[24] Wm. A. Wulf. The WM computer architecture. *Computer Architecture News*, 16(1), March 1988.