LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Adding Fast Interrupts to Superscalar Processors

Computation Structures Group Memo 366
December 14, 1994

Dana S. Henry

Funding for this research is provided in part by the Advanced Research Projects Agency Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland and in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Adding Fast Interrupts to Superscalar Processors

Dana S. Henry
dana@lcs.mit.edu
http://csg-www.lcs.mit.edu:8001/Users/dana/

December 14, 1994

Abstract

The hardware cost of taking an interrupt is increasing as processors become more superscalar. Using FLIP, an aggressively superscalar processor which we have designed and tested in Verilog, we demonstrate that interrupts can be fast and inexpensive. We trace individual signals through FLIP's pipeline stages to show that fast interrupts require negligible new hardware. Except for linkage information, interrupts reuse existing branch mechanisms. An asynchronous interrupt acts as an immediate jump instruction, while a synchronous interrupt acts as a mispredicted branch. Although we concentrate on user-level interrupts, we show that kernel-level interrupts can be handled identically with the addition of protection mode bits to identify the protection mode of every outstanding instruction.

In blending fast interrupts into the superscalar processor, we address two new problems. The first problem arises from fast synchronous interrupts. Because most instructions can cause an interrupt, the processor must be able to revert to its state prior to most instructions, not just mispredicted branches. This ubiquitousness of reversion leads us to design a new renaming data structure. Our renaming data structure can revert to the state prior to any outstanding instruction by updating a single pointer. The entire structure consists of the outstanding renaming bindings plus a simple scan circuit to look up the latest binding. The second problem arises from the interaction of mispredicted branches and asynchronous interrupts. An asynchronous interrupt can sometimes vanish if a branch which dynamically precedes the interrupt handler mispredicts. We offer a simple solution in which the processor remembers an outstanding interrupt and replays the interrupt in case of a preceding misprediction.

1 Introduction

Historically, interrupts have received little attention. The vast majority of the microprocessor market has not had much use for them. In DOS, Windows, and Unix, interrupts happen infrequently. When they do, they typically signal a system event, such as I/O which requires arbitration by the kernel or an error, such as arithmetic overflow which terminates a process anyway.

Several recent developments have undermined the validity of this argument. The success of distributed and parallel processing and the resulting multi-threaded programming environments have dramatically changed the nature and frequency of asynchronous interrupts. Parallel and distributed programs generate frequent asynchronous events in order to communicate. In addition, a wide range of applications rely on synchronous interrupts to signal relatively frequent, non-erroneous events. These include, shared memory emulation [RHL$^+$93], small-kernel operating systems [TR87], transaction support [Rad82], persistent storage management [WK92], distributed virtual memory [LH89], software emulation of instructions [Gol91], and garbage collection [AEL88][Joh90].

Just as the demand for interrupts is rising, today's interrupts are slow and getting slower. Traditionally, interrupts have been slow because of the many layers of software involved in taking an interrupt. Recently, researchers have successfully stripped off these layers and reduced the *software* cost of an interrupt to negligible [TL94]. At the same time, however, the underlying *hardware* latency of transferring control to the interrupt handler has skyrocketed. With potentially hundreds of outstanding instructions, superscalar processors can waste hundreds of issue opportunities draining the processor before even starting the interrupt handler. As the number of outstanding instructions increases, the drainage cost worsens.

We show that the delay in starting an interrupt handler is not necessary. By reusing mechanisms which already exist in the superscalar processor, interrupts can be made fast with minimal additional hardware. We use an aggressively superscalar processor, FLIP, which we have designed and simulated in Verilog, to illustrate fast interrupts. We describe an interrupt implementation which treats interrupts as a routine processor instruction—one that does not break the processor pipeline and one that executes speculatively. FLIP uses standard mechanisms to extract parallelism: renaming and speculative branch execution. We show that both mechanisms can accommodate fast interrupts without deterioration in performance or added hardware cost. In adding fast interrupts, we make only one noticeable change to FLIP's design by reimplementing the renaming data structure. The new structure is both fast and low in VLSI area.

Recall that an interrupt is a transfer of control which takes place when a processor detects one of a predefined set of conditions. These conditions, known as *interrupt events*, are typically defined by the instruction set architecture. Interrupts are necessary because they allow infrequent events to be tested for frequently and because they allow the processor to run user code with the guarantee that the OS will get control back at regular time intervals. Some events are brought about by a specific instruction, such as an add instruction overflowing. These events are called *synchronous* events because the processor must respond to them immediately, before executing the next instruction. Other events are brought about by stimuli from outside of the processor's instruction stream, such as a time-slice timer going off. These events are called *asynchronous* events. In response to any interrupt event, the processor eventually transfers control to a routine customized to handle that type of an event. This routine is called the *interrupt handler*. The process of transferring control is referred to as *taking an interrupt*. Today's state-of-the-art superscalar processors achieve high performance by issuing many instructions, out of order, on every clock cycle. Our goal in designing a superscalar processor with fast interrupts will be to enable the processor to take an interrupt without wasting instruction-issue opportunities.

1.1 The Trouble with Protection Modes

The high cost of interrupts stems from their need to cross protection domains. Traditionally, all interrupt handlers have run in protected, or kernel, mode. When a kernel-mode interrupt event interrupts a processor running a user-mode program, the processor must transition from unprotected to protected mode. During the transition, the processor must insure that outstanding user program instructions do not gain access to protected state and that the kernel interrupt handler's instructions are not denied access to protected state. For instance, an outstanding load instruction in the user program must not be allowed to load a value from protected memory because the processor has begun executing the interrupt handler's instructions and disabled memory protection.

In today's superscalar processors, this guarantee can be costly because these processors operate in only one protection domain at a time. Today's processor is typically busy processing user instructions in user mode when an interrupt arrives. In order to transition to the interrupt handler's kernel mode, the processor must first dispose of all of the outstanding user instructions. Only when the processor is empty can it start fetching the interrupt handler's kernel instructions. Finally, it takes some time for the processor to refill and reach a steady state again in which many instructions run in parallel. From the time when the interrupt is asserted to the time when steady state resumes, the processor may miss hundreds of instruction-issue opportunities.

There are two ways to eliminate this costly delay: user-mode interrupts and mode tagging. User-mode interrupts eliminate the delay because the processor need not cross protection domains to take an interrupt. Since the protection domain remains unchanged on an interrupt, the interrupt handler's instructions can start entering the processor right away. Most types of interrupt events can be handed to the user. Already, on today's machines with only kernel interrupts, some types of interrupts such as message arrival are being reflected back to the user code and incurring the unnecessary overhead of another protection domain transition [Thi93]. [1].

Another way to eliminate the lengthy transition from user to kernel mode is to maintain a *mode bit* with each outstanding instruction specifying the instruction's protection domain. Since protected and unprotected instructions then coexist in the processor, the interrupt handler's instructions can start entering the processor right away. Instead of checking some global mode bit, each instruction checks its own mode bit whenever necessary. For instance, a load instruction may successfully load a value from protected memory because the instruction's mode bit is set to kernel mode. On the same cycle, an add instruction may overflow and raise an overflow interrupt because its mode bit is set to user mode[2].

1.2 Implementing Fast Interrupts

In a processor that executes only one instruction at a time, interrupts are conceptually simple. Figure 1 illustrates the sequence of steps in handling an interrupt. The Figure shows a dynamic sequence of instructions. While the processor is executing Instruction u3, a synchronous interrupt event occurs. The processor aborts Instruction u3 and splices the interrupt handler into the user program in the same way as a procedure call. On the next cycle, the processor jumps to the first

[1] The only interrupt events which *must* be handed directly to the kernel are: the synchronous system call interrupt and the asynchronous timer interrupt.

[2] If its mode bit were set to kernel mode, an overflowing add instruction would typically reset the processor.

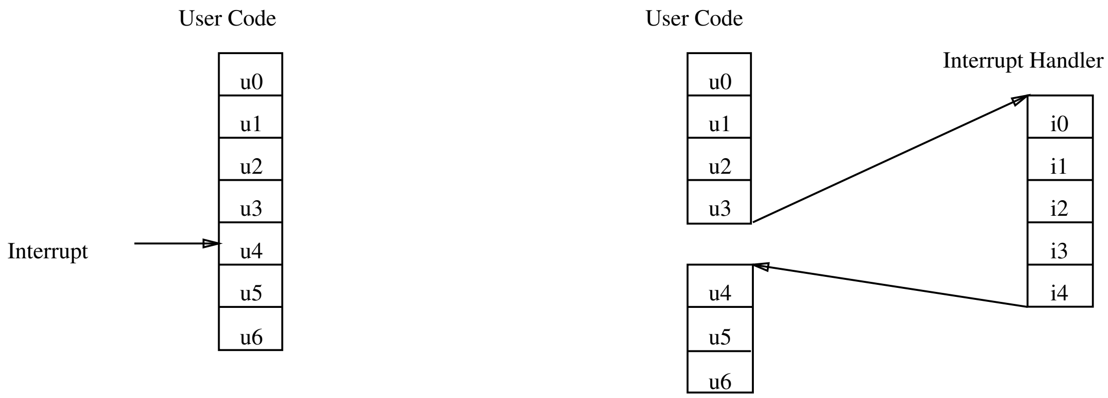

Figure 1: Asynchronous Interrupt.

instruction of the interrupt handler and saves the return address u4 in a linkage register. To prevent another interrupt from destroying the value in the linkage register, the processor at this time also disables further interrupts. The last instruction in the interrupt handler restores the program counter from the linkage register and reenables further interrupts.

At first glance, it is not obvious how to go about implementing interrupts in a superscalar processor. Since instructions are not executed one-at-a-time, the state of the processor can be inconsistent. Using the scenario shown in Figure 1 as an example, Instructions u0 and u1 may not have started executing while Instructions u4, u5, and u6 may have finished executing. To present the interrupt handler with the precise state of the processor after Instruction u3, we may have to complete preceding instructions and undo succeeding ones.

Recovering the precise state without incurring delays is the main topic of this paper. Fortunately, many of the mechanisms necessary to implement fast interrupts are already present in today's superscalar processors. An asynchronous interrupt can be executed using the mechanisms for the existing jump instruction. A synchronous interrupt can be executed using the mechanisms for the existing conditional branch instruction which mispredicted. Although not much new hardware is necessary, interesting problems arise. Existing branch misprediction hardware is optimized to checkpoint the processor's state only at branch instructions. In contrast, almost all instructions can cause a synchronous interrupt. We present an inexpensive new hardware structure which checkpoints the processor on the fly, on every cycle. Another problem arises because speculative user instructions may be coexisting with the instructions of an asynchronous interrupt. Some care is needed to prevent the interrupt from vanishing on a user branch misprediction.

The rest of this paper describes FLIP, a superscalar processor with fast user-mode interrupts, which we designed and simulated in Verilog. Although FLIP's interrupts are user-level, the design is equally applicable to kernel-level interrupts with mode tagging. Section 2 describes FLIP and FLIP's implementation of interrupts and explains the vanishing interrupt problem and its solution. Section 3 describes in greater detail the renaming logic which enables the backing out of computation on any instruction. Section 4 concludes with a discussion of related work and future research directions.

2 Processor Design

Now that we have described interrupts, in general, let us consider our implementation of fast user-level interrupts in the context of FLIP, a superscalar processor which we designed. We reserve detailed discussion of FLIP's renaming table to Section 3. We start in Section 2.1 by explaining the basic pipeline structure of FLIP together with relevant data and control paths. Section 2.2 describes how FLIP implements speculative branching. Section 2.3 shows how FLIP implements synchronous interrupts by reusing its mechanism for mispredicted branches. Section 2.4 shows how FLIP implements asynchronous interrupts and addresses the interaction between asynchronous interrupts and speculative execution, namely the case of the vanishing interrupt.

2.1 FLIP

We start with a description of the pipeline structure of FLIP. FLIP is an aggressively superscalar processor which we designed and simulated in Verilog. FLIP uses a simple RISC instruction set which has been used in our graduate computer architecture class. Much like existing and proposed processors, FLIP extracts parallelism through the use of the Tomasulo algorithm and through speculative branch execution. Although FLIP's methods are not unique, a detailed description of its pipeline structure is missing in literature and is essential to understanding its implementation of interrupts. Some readers may choose to skip to Section 2.3 which describes FLIP's incorporation of synchronous interrupts into its pipeline.

Although FLIP is an aggressively superscalar processor which issues instructions out-of-order and speculates on branch outcomes, FLIP's pipeline closely resembles the traditional RISC pipeline in Figure 2. Recall that a RISC processor executes each instruction through the same series of pipelined stages. An instruction is fetched and decoded. Operands are read from the register file, operation is executed, and results are written back to the register file in successive stages [HP89, chapter 6] [3]. Figure 3 shows FLIP's pipeline. The difference between standard RISC and FLIP, is a pair of additional steps, renaming and buffering, which extract parallelism from the sequential instruction stream.

The process of renaming and buffering, first described by Tomasulo [AST67], eliminates artificial dependencies among instructions and allows ready-to-run instructions to bypass earlier instructions which are not ready. To illustrate how renaming removes artificial dependencies, consider the following code segment:

```
1. r3 <- r1 * r2
2. r5 <- r3 / r4
3. r3 <- r6 + r7
```

Since the arguments of Instruction 3 do not depend on Instruction 1 or 2, we would like to execute Instruction 3 in parallel with instructions 1 and 2. But we cannot because, if Instruction 3 finished before Instruction 2, Instruction 2 might read the wrong value from Register 3. Moreover, if Instruction 3 finished before Instruction 1, the final value of Register 3 would be incorrect. We say that Instruction 3 is anti-dependent on Instruction 2 and output-dependent on Instruction 1 [PW86]

[3] We show only one execute stage, for simplicity. Some RISC processors have more than one.

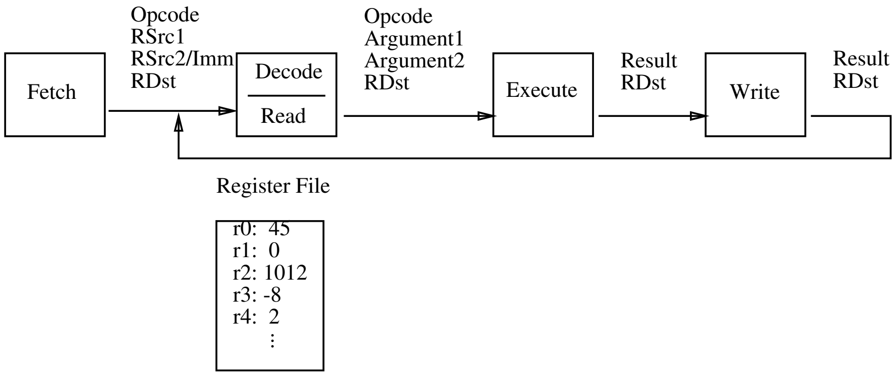

Figure 2: The pipeline of a RISC processor, show for comparison. The labels along each arrow describe the information transfered from one stage to the other. The main data structure, the register file, is outlined.

Artificial dependencies arise because code must frequently reuse the small number of registers defined by the instruction set architecture.

Renaming eliminates this reuse by mapping, or renaming, the small number of registers provided by the instruction set into a much larger number of registers provided by the hardware. For clarity, let us refer to instruction set registers as "registers" and to hardware registers as "tags". As instructions enter the decode/rename stage of FLIP's superscalar pipeline in Figure 3, the processor replaces their register numbers with tag numbers. For example, our original code segment may become:

```
1. t42 <- t15 * t77
2. t9 <- t42 / t56
3. t101 <- t39 + t14
```

Note that Instruction 3 is now independent of 1 and 2. Notice also that Instruction 2 correctly receives the result of Instruction 1 through Tag 42. To correctly assign tags, the decode/rename stage must maintain a mapping table from registers to tags. The mapping table, at the time Instruction 2 enters the decode/rename stage, tells the processor that Register 3 is mapped to Tag 42. The functionality and implementation of the mapping table will be described in detail in Section 3.

Renaming alone will not unleash parallelism, however. Buffering is also necessary. As instructions exit the decode/rename stage, they are free of artificial dependencies, but they are still exiting in the order in which they were fetched. Without further intervention, Instruction 3 must wait its turn behind Instructions 1 and 2. The buffer/read stage eliminates the wait by buffering instructions whose arguments are not ready and allowing subsequent instructions to overtake them.

At the heart of the buffer/read stage is an associative buffer analogous to the template buffer in dataflow machines [Den80, AC86][4]. Figure 4 shows one entry in the buffer which corresponds to

[4]In fact, for arithmetic instructions, the segment of the Tomasulo pipeline from the buffer/read stage to the write stage is identical to a dataflow machine

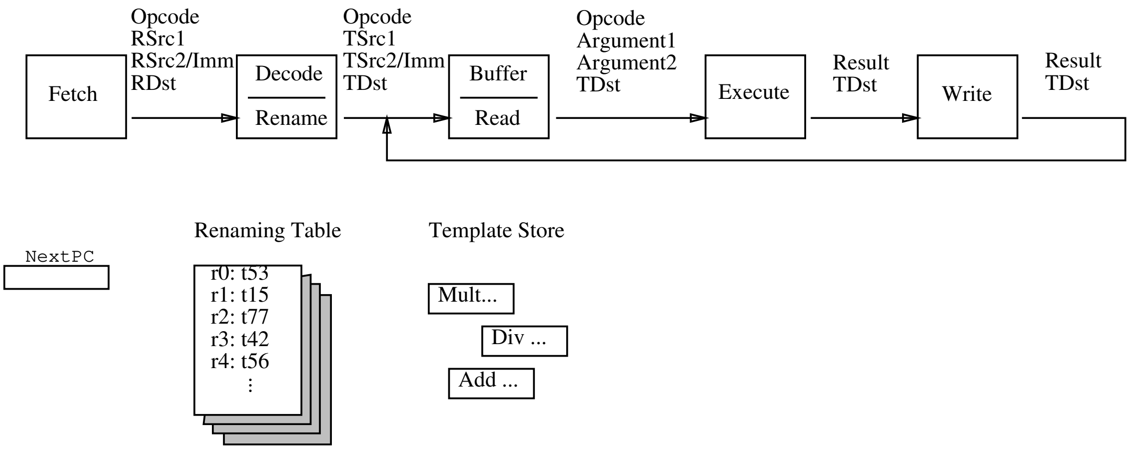

Figure 3: The pipeline of the FLIP processor. The labels along each arrow describe the information transfered from one stage to the other. Main data structures, the renaming table and the template buffer, are outlined.

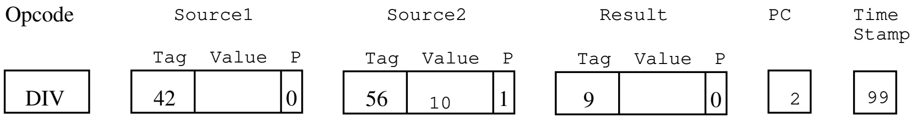

Figure 4: Example of an entry in the template buffer of FLIP.

Instruction 2. The entry contains the decoded, renamed instruction; space for the argument values; space for the result value; and presence bits, `P`, indicating whether arguments and result are present. Two additional fields, the `PC` and the `Time Stamp`, are described later in Section 2.2. In case of Instruction 2 the entry shows that the first argument, the output of Instruction 1, is absent, as is the result. As each instruction enters the buffer/read stage, it associatively searches the result tags of all the templates in order to find its arguments. If any arguments are not present, the instruction is buffered until its missing arguments are associatively written by some preceding instruction in its write stage. In the meantime, succeeding instructions, such as Instruction 3, can precede.

We have shown how FLIP processes arithmetic operations using renaming and buffering, but not how it processes branches and memory operations. Memory operations present two additional kinds of dependencies. First, since they can permanently modify off-chip storage, memory operations cannot be undone. Consequently, memory operations cannot execute until the processor is certain that they will not have to be undone. The processor becomes certain when all preceding instructions which could raise an interrupt or mispredict have completed. The second kind of dependency arises among memory operations to the same address. Any load instruction must receive the value most recently stored to that address. Since memory operations are not relevant to the handling of interrupts, we will not discuss FLIP's mechanisms for sequencing memory operations in this paper.

2.2 Branches in FLIP

Branch instructions, on the other hand, warrant thorough discussion since much of their hardware is reused by synchronous interrupts. Unlike Tomasulo's original algorithm, all modern superscalar processors, including FLIP, speculate on the outcome of branch instructions [5]. When a processor encounters a conditional or indirect branch, it predicts the outcome of the branch and continues executing from the predicted instruction path. The trouble occurs if, later on as the branch resolves, the prediction proves incorrect. At such time, FLIP must somehow restore the processor state to before the misprediction was made and resume fetching from the corrected instruction stream. Since mispredicted branches are relatively common [Smi81, FF92], about one in every one hundred instructions, it is essential that the restoration of state be fast.

We describe how FLIP restores state by referring to Figure 5 which reviews FLIP's pipeline updated with all the signals utilized by branching instructions. First, to be able to restore old state, FLIP maintains two additional pieces of information with each outstanding instruction: the time stamp and the program counter. FLIP time stamps instructions in order to recognize which outstanding instructions precede and which follow the mispredicted branch. The stamp can be generated by a counter in the fetch stage and each instruction stamped as it exits the fetch stage[6] In addition to the time stamp, FLIP passes each instruction's program counter through the pipeline in order to be able to compute and to correct branch addresses (as well as to take synchronous interrupts, ie exceptions).

A speculative branch instruction passes through FLIP's pipeline much like any other instruction. The branch instruction makes its prediction in the decode/rename stage. Based on the prediction, the decode/rename stage may direct the fetch stage to jump to the target address by asserting the `Jump` line and writing the target address to the `JumpPC` lines. The fetch stage jumps to the target address on the next clock. The predicted branch instruction, annotated with the predicted direction of the branch, precedes to the buffer/read stage where it awaits its arguments. Sometime later, when its arguments become available, the branch instruction issues and enters the execute stage. If the execute stage detects a mispredicted branch, it triggers events to undo mispredicted instructions and to resume fetching from the correct arm of the branch. To undo mispredicted instructions, the execute stage asserts the `Revert` line to all pipeline stages and writes the time stamp of the mispredicted branch to the `Stamp` lines. On the next clock, all pipeline stages associatively discard instructions whose time stamp exceeds that of the mispredicted branch and the renaming table reverts to the bindings which were in effect at the time specified by the time stamp. We postpone the discussion of how FLIP reverts its renaming table to Section 3. To resume fetching from the correct arm of the branch, the execute stage asserts the `Jump` line to the fetch stage and writes the corrected target address to the `JumpPC` lines.

To make speculative branch execution more concrete, let us trace the path of a conditional branch instruction through the processor. Consider the branch on zero instruction:

```
10. brz r1 imm5           % jump to PC+5 if the value of r1 is 0
```

[5]Studies show that speculative branch execution is critical to extracting significant instruction-level parallelism from a typical RISC instruction stream [LW92, Wal91].

[6]FLIP does not actually generate time stamps since the ordering of instructions is implicit in their template buffer address. We use time stamps here for simplicity of discussion.

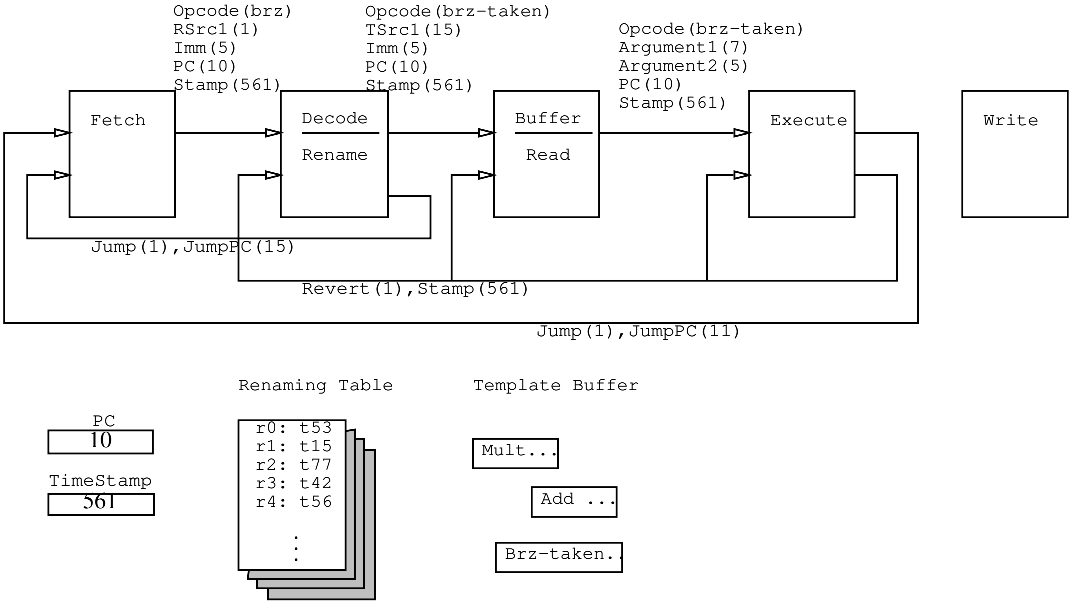

Figure 5: FLIP's logic relevant to branching. Values in parenthesis correspond to a branch-on-zero instruction used as an example in the text. FLIP's state corresponding to each pipeline stage is shown just after the branch-on-zero instruction passes through that stage.

This instruction jumps 5 instructions relative to its PC if the content of Register 1 is zero. Figure 5 shows, in parenthesis, all the values passed along FLIP's pipeline as a consequence of this instruction. When the branch instruction enters the fetch stage, it is assigned a time stamp, in this case 561. In the decode/rename stage, the `brz` instruction makes its prediction. Let us assume that the `brz` instruction predicts that the branch will be taken. It adds 5 to its PC and tells the fetch stage to jump to this address. The decode/rename stage emits the renamed `brz` instruction annotated with its prediction:

```
brz-taken t5 imm5        % if the value of t5 is not 0, revert to PC+1
```

Just like any arithmetic instruction, the `brz-taken` instruction is buffered in a template in the buffer/read stage until its arguments become available. The `brz-taken` instruction may not execute for a long time, while it is waiting for the value of Tag 15. In the meantime, instructions along the predicted instruction stream execute speculatively. If the `brz` instruction eventually finds that the value of Register 1 is *not* zero, it restores the processor back to its state before the branch and resumes fetching from PC+1.

It is possible for a processor to have many speculative branch instructions outstanding at once. If these instructions are correctly predicted, execution simply continues. If, on the other hand, several branch instructions are predicted incorrectly, we must insure that execution eventually reverts to the corrected arm of the earliest mispredicted branch instruction. FLIP's hardware which we have already described insures correct reversion. Consider a scenario in which FLIP is predicting two `brz` instructions to be taken, one with a time stamp 300 and one with 304. Let us assume that both branches are mispredicted. There are two possible scenarios. The `brz` instruction corresponding to the earlier branch, 300, may issue first and anull all instructions whose time stamp exceeds 300, including branch 304. Computation correctly resumes at the corrected arm of the earliest branch. In the second scenario, the later `brz` instruction, 304, issues first. FLIP corrects the later misprediction and continues executing. Note that the earlier branch, 300, remains outstanding since its time stamp is less than that of the later branch. Eventually, the `brz` instruction corresponding to the earlier branch 300 issues, anulls all later instructions and execution again correctly resume at the corrected arm of the earliest branch.

2.3 Synchronous Interrupts in FLIP

Now that we have described the basic design of the FLIP processor, we can turn our attention to interrupts. We show how FLIP implements synchronous interrupts in this section, and asynchronous interrupts in the following section.

Before we delve into FLIP's implementation of interrupts, we must mention the additional processor state which FLIP maintains for the purpose of handling interrupts. FLIP's interrupt related state is similar to that in many modern implementations of interrupts [MIP92, ROS93, Mot93], except that the state is accessible at user-level. A linkage register, `ReturnPC`, resides in the fetch stage and stores the address at which computation resumes once an interrupt handler returns. An interrupt enable bit, `IntEn`, which enables or disables interrupts, resides in the decode/rename stage [7]. Each instruction is stamped with the current value of the `IntEn` bit in the decode/rename stage and it carries its `IntEn` value along as it traverses the pipeline. On a misprediction, the decode/rename stage reverts to the `IntEn` bit of the mispredicted branch instruction. Finally, a base register, `Base`, specifies the beginning address relative to which FLIP computes the starting address of the interrupt handler for each type of an interrupt. This register, which is typically very rarely modified, is not renamed in FLIP.

An instruction which raises a synchronous interrupt behaves much like a branch predicted not taken and later found to be taken. Conceptually, each instruction which could raise an interrupt makes the prediction that it will *not* raise an interrupt. As a result of this prediction, the processor continues fetching from the original instruction stream. Later on, when the instruction executes, it attempts to confirm this prediction. If the confirmation fails, FLIP takes the same steps as it would on a mispredict. In addition, FLIP saves linkage information and disables further interrupts.

Figure 6 shows all the pipeline signals utilized by an instruction which causes a synchronous interrupt. Just like a mispredicted branch, a synchronous interrupt is typically detected in the execute stage [8] The execute stage, on detecting a synchronous interrupt, checks the instruction's interrupt enable bit (`IntEn`). If user-level interrupts are disabled, the execute stage raises a kernel-level interrupt. If user-level interrupts are enabled, FLIP precedes to revert to the state of the interrupting instruction and to take the interrupt. FLIP revert its state on an interrupt using the same mechanism and the same signals as per a mispredicted branch. The `Revert` signal

[7]It may be desirable to have two interrupt enable bits and corresponding linkage registers, one for asynchronous interrupts and one for synchronous ones. This is because an asynchronous interrupt handler, such as a message handler, may wish to disable other messages from interrupting and but still wish to handle user-level exceptions.

[8]The exception is a memory interrupt resulting from an instruction memory access which occurs in the fetch stage and can be handled like an asynchronous interrupt.

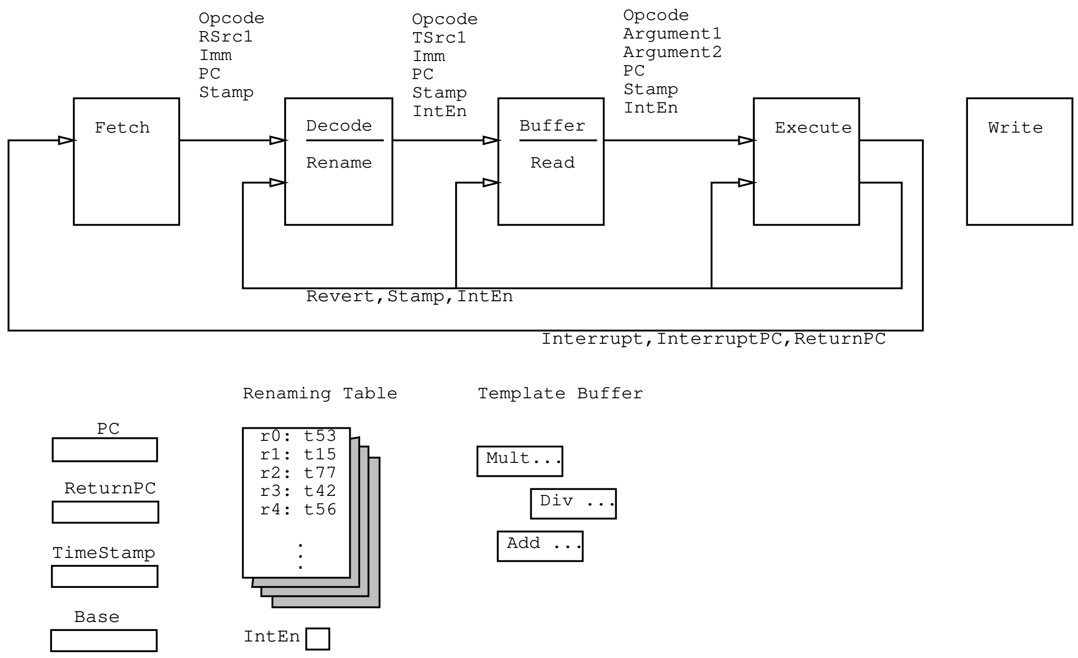

Figure 6: FLIP's pipeline signals and state relevant to synchronous Interrupts.

instructs pipeline stages to anull all instructions whose time stamp exceeds that of the interrupting instruction (`Stamp`) and the decode/rename stage to also revert its renaming table and to disable further interrupts (`IntEn`). The execute stage instructs the fetch stage to take the interrupt by raising the `Interrupt` signal and by sending the fetch stage the starting address of the interrupt handler (`InterruptPC`) and the address of the interrupting instruction (`ReturnPC`). On the following cycle, the fetch stage updates the linkage register with the address of the interrupting instruction and jumps to the interrupt handler.

2.4 Asynchronous Interrupts in FLIP

Now that we have described synchronous interrupts, let us turn our attention to asynchronous interrupts. FLIP takes a very aggressive approach to implementing asynchronous interrupts with some resulting complications. FLIP can start fetching interrupt handler instructions on the very next cycle after an asynchronous interrupt has been asserted. FLIP does not invalidate any outstanding instructions and its pipeline does not waste a single slot. As a consequence, asynchronous interrupts can be very efficient, but also very speculative. There may be many outstanding speculative branches at the time FLIP takes an interrupt. Branches in the fetch and decode/rename stages, may not have even made their prediction yet. Branches further down the pipeline may not have resolved their predictions yet. Even the state of the interrupt enable bit, at the time FLIP takes an interrupt, is speculative.

Ignoring the interaction of branch instructions and interrupts, for now, FLIP's implementation of asynchronous interrupts is quite simple. Figure 7 shows FLIP's pipeline as it relates to asynchronous

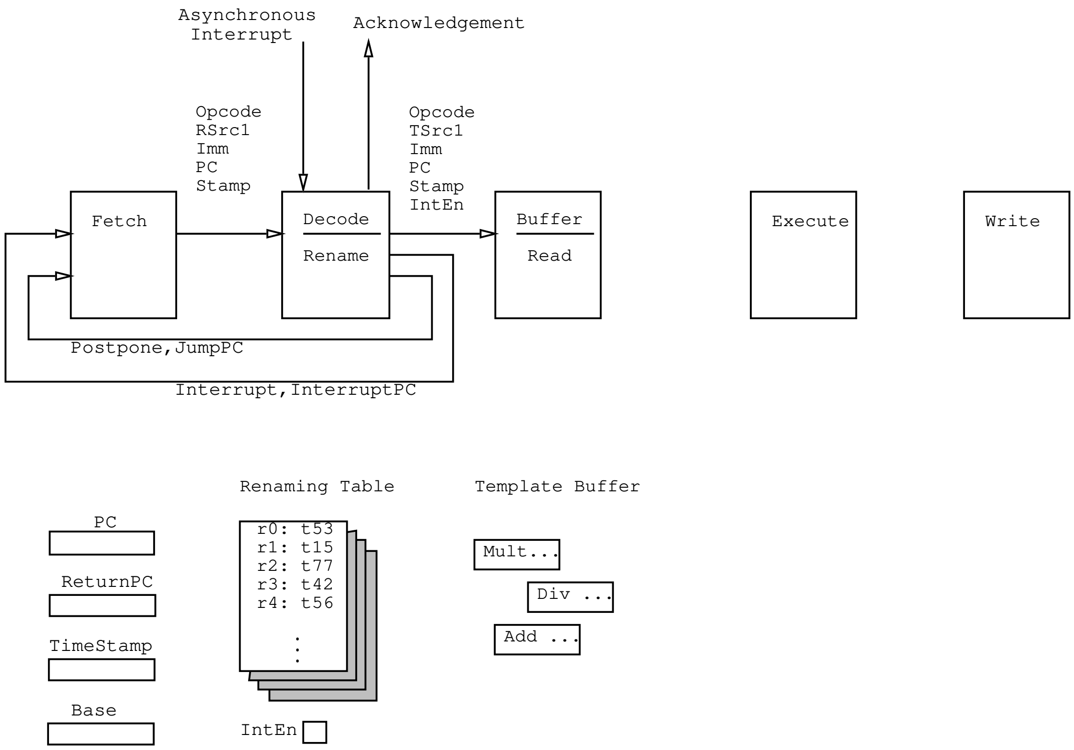

Figure 7: FLIP's pipeline signals and state relevant to asynchronous Interrupts.

interrupts. Asynchronous interrupts are detected and acknowledged by the decode/rename stage. The decode/rename stage detects and acknowledges asynchronous interrupts in parallel with its decoding and renaming of an instruction. If interrupts are enabled when an asynchronous interrupt is detected, the decode/rename stage accepts the interrupt. It informs the fetch stage of the interrupt (`Interrupt`) and sends it the starting address of the corresponding interrupt handler (`InterruptPC`). For reasons which we describe shortly, the decode/rename stage does *not* acknowledge the interrupt at this time. On the next clock cycle, the fetch stage saves its current PC (`PC`) in the interrupt linkage register (`ReturnPC`) and starts fetching instructions from the interrupt handler. The decode/rename stage disables further interrupts on the next cycle, when the first instruction of the interrupt handler reaches it.

As we have already pointed out, the implementation of asynchronous interrupts is complicated by the interaction of interrupts with branch instructions. If any branch instruction which precedes the interrupt handler in FLIP's pipeline were to branch, the processor would resume fetching from the user program and forget about the interrupt handler. Recall, that a branch instruction may branch either in the decode stage as it makes its prediction or in the execute stage as it mispredicts (Figure 5). Let us first consider branch instructions as they make their prediction. Any branch instruction in the fetch or decode/rename stage at the time the decode/rename stage accepts an asynchronous interrupt has yet to make its prediction and branch. To avoid branching back to the user program just as the interrupt handler is beginning to run, FLIP postpones taking either branch until the interrupt returns [9]. The decision to postpone branching is made by the decode/rename stage which instructs the fetch stage to postpone (`Postpone`) the branch instead of jumping to it (`Jump`). To postpone taking the branch, the fetch stage updates the linkage register (`ReturnPC`) with the target address of the branch (`JumpPC`).

Mispredicted branch instructions and, analogously, synchronous interrupts pose a similar problem. If an instruction which precedes the interrupt handler mispredicts, FLIP reverts to its state at the time of the mispredicted instruction. All of the interrupt handler's outstanding instructions disappear and FLIP resumes fetching from the corrected arm of the mispredicted branch. The asynchronous interrupt effectively *vanishes*. To handle a vanished asynchronous interrupt, FLIP simply takes the interrupt again. Recall that the decode/rename stage has not yet acknowledged the interrupt at the time of the misprediction. The decode/rename stage does not acknowledge an asynchronous interrupt until it is certain that the interrupt will not vanish, that is until the first instruction of the interrupt handler commits. When the decode/rename stage detects a misprediction which precedes its unacknowledged asynchronous interrupt, it acts as if the asynchronous interrupt line had just been raised and eventually accepts the interrupt again.

3 Hardware Structures

In the previous section, we have described how FLIP incorporates interrupts into its superscalar pipeline. We have shown that FLIP reuses existing mechanisms and often even existing pipeline signals to implement fast interrupts at almost no hardware cost. The one aspect of FLIP which we have so far neglected is FLIP's data structures: the template buffer and the renaming table. In this section, we describe how FLIP implements these structures and how fast interrupts impact the implementation. We show that both data structures can be implemented efficiently in the presence of fast interrupts.

The way interrupts impact FLIP's data structures is to make checkpointing ubiquitous. As we saw in the previous section, interrupts do not introduce any new operations or signals into the template buffer or the renaming table. An asynchronous interrupt has no effect on either structure. A synchronous interrupt appears to each structure as a mispredicted branch. Since most instructions can raise a synchronous interrupt, however, the set of instructions which can mispredict grows from just branches to almost all instructions. With synchronous interrupts, the processor must be able to checkpoint the state of the renaming table and the template store on every instruction, rather than only on branches.

In the following two sections, we show how ubiquitous checkpointing influences our implementation of FLIP's data structures. We find the standard stack and wrap-around queue structures for implementing the template buffer, as well as our variation, the coalescing-stack structure, to be adequate. The renaming table, on the other hand, requires a new approach. Since our new approach draws on our template buffer implementation, we start by briefly describing the template buffer.

[9] We assume no delay slots, although a similar approach works for branches with delay slots.

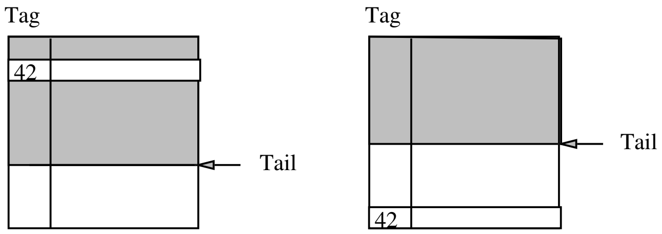

Figure 8: Our coalescing stack implementation of the template buffer before and after a template with the result tag of 42 is deleted. The grey area corresponds to templates which hold valid instructions, the white area to templates which do not.

3.1 The Template Store

We are experimenting with three different implementations of the template buffer in FLIP: a stack, a wrap-around queue, and a coalescing stack. Although a detailed discussion of the three implementations is outside the scope of this paper, the following are their main features. Each structures consists of a buffer of memory in which each entry can hold one template. Templates are stored in successive addresses in the order in which they enter the buffer/read stage. The stack pushes new templates until it overflows, then drains the processor and reinitializes the stack. The wrap-around queue pushes new templates and pops finished ones as they reach the head of the queue. Unlike the two stack implementations, the queue requires an auxiliary register file since the sequence of templates present in the queue at any one time may contain no instruction whose result belongs to a particular register. The coalescing stack pushes new templates at the `Tail` and deletes finished ones from anywhere within the stack. Figure 8 shows how the coalescing stack deletes a template from within the stack by moving it to the very bottom of the buffer and sliding all succeeding templates up by one. The newly deleted template and its result tag are ready to be reused later.

Note that the three implementations of the template buffer are not affected by ubiquitous checkpointing. Since they store templates in their temporal order, all three structures implicitly checkpoint on every instruction. Reverting to an older state on a misprediction, amounts to freeing all templates which spatially succeed the mispredicted template. Referring to Figure 8, reverting on a misprediction simply corresponds to moving the `Tail` pointer up to the mispredicted template.

3.2 The Renaming Table

In case of the renaming table, on the other hand, ubiquitous checkpointing significantly impacts our design. A simple implementation of the renaming table which checkpoints on branches may not easily extend to checkpoint on all instructions. Consider, for instance, an implementation in which the renaming table maintains shadow copies of itself. This is the approach taken by the HPSm processor [HP86, UHN+92]. To checkpoint on branches, a FLIP implementation with a 20 template template buffer might have four shadow renaming tables, for instance. In the hopefully rare occasion when the decode/rename stage encounters a fifth speculative branch, it stalls the

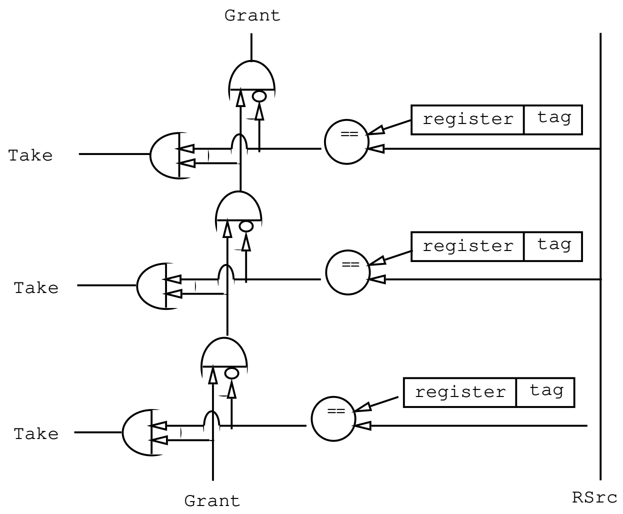

Figure 9: The circuit which implements the `lookup` operation on FLIP's sequence of register-tag bindings.

processor until the oldest speculative branch resolves. We somewhat hinted at this implementation in our pipeline diagrams in which the renaming table is shadowed by several older copies. The shadowing implementation becomes much more costly in the presence of fast interrupts. Since potentially almost any instruction can mispredict, the renaming table must maintain a shadow copy of itself for almost every outstanding instruction.

FLIP takes a drastically different approach to implementing the renaming table. It eliminates the renaming table altogether. Instead of a renaming table, FLIP maintains the register-tag binding of every outstanding instruction in its template store entry. FLIP adds, deletes, and anulls register bindings at the same time as it adds, deletes, and anulls the instruction's template. The only additional operation required of the renaming table is the *lookup* operation. As a new instruction enters the decode/rename stage, FLIP must look up the current binding of the instruction's argument registers. Since the bindings are stored in a time ordered sequence, looking up the current binding amounts to finding the most recent occurrence of the register.

A simple scan circuit of Figure 9 can find the most recent binding of a register. The figure shows a column of register-tag bindings in temporal order, with the most recent binding at the bottom. The `RSrc` line broadcasts the argument's register number to all the bindings. The `Grant` tells each successive binding whether any preceding binding has already matched. The `Take` line reports the most recent match. The scan circuit initiates its search by raising the `Grant` line to the most recent binding, as specified by the `Tail` pointer. This scan circuit is linear in time and area with the number of outstanding instructions. Logarithmic versions also exist [CLR90, Chapter 29]. FLIP maintains two such scan circuits, one for each source register of the instruction passing through the decode/rename stage. In general, a superscalar processor which decodes n instructions at a time would require $2n$ scan circuits.

The scan circuit of Figure 9, as well as its logarithmic version, require very little hardware. All that is required is several logical gates plus a register-tag binding for every outstanding instruction. Because of its economy, the scan circuit implementation of the renaming table may be desirable for many speculating superscalar processor, regardless of their interrupt implementation.

4 Related and Future Work

We have presented the design of FLIP. Before discussing future research directions, let us review the relationship of other work to the FLIP mechanisms. Most previous research on fast interrupts has either focused on precise synchronous interrupts or on asynchronous message-arrival interrupts.

Smith and Pleszkun [SP88] describe and evaluate methods for adding precise synchronous interrupts to a pipelined processor. Unlike FLIP, their processor issues instructions in order and does not speculate. A reorder table allows their instructions to complete out of order, yet update state in order. Our wrap-around queue implementation of the template buffer is similar to Smith and Pleszkun's reorder buffer. Smith and Pleszkun address the problem of crossing protection domains on kernel-level interrupts by draining the pipeline, whereas FLIP avoids draining the pipeline.

Sohi describes [Soh90] an implementation of synchronous interrupts which resembles FLIP. Sohi's processor, like FLIP, issues instructions out of order. Sohi's template buffer (which he calls the *RUU*) resembles our wrap-around queue implementation. For renaming, Sohi uses a pair of counters per register specifying the number of outstanding instances of that register and the latest instance number. FLIP, in contrast, allocates tags from a pool of free tags and uses a scan-based lookup table to represent the association between registers and tags. Sohi's paper does not describe how to revert the instance counters to their correct state on a synchronous interrupt or mispredicted branch. FLIP's scan-based lookup table can revert quickly and inexpensively to the right processor-state on a synchronous interrupt or a mispredicted branch. Unlike FLIP's design, Sohi's paper does not address the problem of crossing protection domains for kernel-level interrupts.

Research on asynchronous interrupts is driven by the parallel and distributed processing community which tries to reduce the cost of interprocessor communication [HJ92]. In interrupt-driven communication a processor is interrupted right away when a message arrives, avoiding the delays and overheads associated with polling but incurring the cost of an interrupt. Today's multiprocessors typically use off-the-shelf processors and live with the cost of kernel-level interrupts. On these machines, users often go to great lengths to avoid interrupts because they are too expensive [BK94]. The cost of interrupts is projected to be even greater in the next generation of multiprocessors based on off-the-shelf superscalars [KOH+94, AAC92].

Our work uses a model of interrupts in which interrupt handlers run atomically with respect to the user program's thread of computation. Difficulties arise in the treatment of instructions which are outstanding in the processor at the time the processor takes the interrupt. These difficulties are not present in processors in which the interrupt handler runs as a separate thread or in processors which execute only one instruction per thread at a time. The extreme example of such a processor is the dataflow processor, such as Monsoon [Pap91], which can execute a new one-instruction thread on every cycle. Another example is the HEP [Jor83] processor which interleaves instructions from multiple threads and the J-Machine multicomputer in which the state of a supervisor thread and a user thread coexist in the processor [DFK+92]. While such processors are capable of providing efficient interrupts even in the presence of many outstanding instructions, they rely on parallel instruction sets and computing models to generate multiple threads of computation. FLIP, on the other hand, provides lost-cost, fast, user-level interrupts for a commodity serial instruction set.

Adding fast interrupts to aggressively superscalar processors requires only minor changes and brings great potential benefits. We have shown that, in aggressively superscalar processors such as FLIP, interrupts can reuse existing mechanisms for executing branch instructions. The two problems which arise in implementing fast interrupts, the vanishing asynchronous interrupt and ubiquitous checkpointing due to synchronous interrupts, can be resolved by simple hardware solutions. Interrupts are a simple processor mechanism used by an extraordinary variety of applications. We believe that making the cost of interrupts comparable to the cost of branches will enable progress in many fields of computation.

Our goals for future research include a more aggressive implementation of user-level asynchronous interrupts. We plan to investigate a model of the asynchronous interrupt in which the interrupt handler is not atomic with respect to the user program, rather it runs in parallel with the user program providing a natural source of instruction-level parallelism. In our proposed model, the interrupt handler and the user program share processor resources and architectural state, including the register file. Since they share registers, they can synchronize quickly via atomic operations on registers. Since the handler and the program share processor resources, such as the template buffer and the renaming table, we predict that very little new hardware is needed.

References

[AAC92] Boon Seong Ang, Arvind, and Derek Chiou. StarT the next generation: Integrating global caches and dataflow architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture Dataflow Workshop*, Hamilton Island, Australia, May 1992.

[AC86] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988.

[AST67] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.

[BK94] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS '94)*, pages 858–867, Mexico, April 1994.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1st edition, 1990.

[Den80] Jack B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[DFK$^+$92] William J. Dally, J.A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Filer. The message driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '92)*, pages 111–122, Boston, Massachusetts, October 1992. (Also published as SIGARCH Computer Architecture News, Volume 20, Special Issue, October 1992.).

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[HJ92] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '92)*, pages 111–122, Boston, Massachusetts, October 1992. (Also published as SIGARCH Computer Architecture News, Volume 20, Special Issue, October 1992.).

[HP86] Wen-mei Hwu and Yale N. Patt. HPSm, a high performance restricted data flow architecture having minimal functionality. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 297–306, Tokyo, Japan, June 1986.

[HP89] John L. Hennessy and David A Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2929 Campus Drive, San Mateo, CA 94403, 1st edition, 1989.

[Joh90] Douglas Johnson. Trap architectures for Lisp systems. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 79–86, Nice, France, June 1990.

[Jor83] Harry F. Jordan. Performance measurements on HEP - a pipelined MIMD computer. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 207–212, Stockholm, Sweden, June 1983. (Also published as SIGARCH Newsletter, Volume 11, Number 3, June 1983.).

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, April 1994. (Also published as SIGARCH Computer Architecture News, Volume 22, Number 2, April 1994.).

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *The 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 1992. (Also published as ACM SIGARCH Computer Architecture News, Volume 20, Number 2.).

[MIP92] MIPS Computer Systems, Inc., Englewood Cliffs, New Jersey 07632. *MIPS RISC Architecture*, 1992.

[Mot93] Motorola, RISC Microprocessor Division, OE42, Motorola Semiconductor Products Sector, PO Box 20912, Phoenix, Arizona 85036-9938. *PowerPC 601*, 1993.

[Pap91] Gregory M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor*. The MIT Press, Cambridge, MA, 1991.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Rad82] George Radin. The 801 minicomputer. In *Proceedings of the First Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS '82)*, pages 39–47, Palo Alto, California, March 1982. (Also published as SIGARCH Computer Architecture News, Volume 10, Number 2.).

[RHL+93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, California, May 1993. (Published as ACM SIGMETRICS Volume 21, Number 1, June 1993.).

[ROS93] ROSS Technology, Inc., 5316 Hwy. 290 West, Austin, TX 78735. *SPARC RISC User's Guide hyperSPARC Edition*, 1993.

[Smi81] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, June 1981.

[Soh90] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[SP88] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[Thi93] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1264. *CMMD User's Guide*, 1993.

[TL94] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. Technical Report UW-CSE-94-07-05, University of Washington, Department of Computer Science and Engineering, Seattle, WA 98195, July 1994. (To appear in ASPLOS '94.).

[TR87] Avadis Tevanian, Jr. and Richard F. Rashid. MACH: A basis for future UNIX development. Technical Report CMU-CS-87-139, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA 15213, June 1987.

[UHN+92] Gregory A. Uvieghara, Wen-mei W. Hwu, Yoshinobu Nakagome, Deog-Kyoon Jeong, David D. Lee, David A. Hodges, and Yale N. Patt. An experimental single-chip data flow cpu. *IEEE Journal of Solid-State Circuits*, 27(1):17–28, January 1992.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.

[WK92] P. R. Wilson and S. V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 364–377, Dourdan, France, September 1992. The IEEE Computer Society Press.