Proceedings of the 2001 Conference on Advanced Research in VLSI

Precise Exceptions in Asynchronous Processors

Rajit Manohar, Mika Nyström, Alain J. Martin*

Abstract

The presence of precise exceptions in a processor leads to complications in its design. Some recent processor architectures have sacrificed this requirement for performance reasons at the cost of software complexity. We present an implementation strategy for precise exceptions in asynchronous processors that does not block the instruction fetch when exceptions do not occur; the cost of the exception handling mechanism is only encountered when an exception occurs during execution—an infrequent event.

1: Introduction

Ordinarily, a processor executes a sequence of instructions without interruption. Conceptually the instructions are executed one after another, with some instructions that modify the control flow. However, this stream of execution can be interrupted in two different ways: by interrupts—external asynchronous events that are typically generated by various I/O devices—and by exceptions.

Exceptions are used to handle uncommon events that can occur during the execution of otherwise innocuous instructions. Exceptions are used when the events to be handled are so rare that the cost of the exception mechanism is outweighed by the savings resulting from not having to implement in hardware, for every instruction, the behavior when an exceptional condition does occur.

Exceptions are used by a number of software mechanisms. They are used to prevent an executing process from accessing memory that belongs to another process. They are used to prevent a user program from executing certain special, privileged instructions. They are used to begin the execution of special operating system subroutines (traps). They are used when some functionality is implemented partly in hardware and partly in software: hardware page tables (TLB), partial implementations of IEEE 754/854 floating-point arithmetic, etc.

When an exception or interrupt is encountered, a processor aborts the normal instruction sequence by jumping to a predetermined address (an operating-system or architecture-specific address) in memory. This location in memory contains a software routine, the exception handler, that services the exception or interrupt. The hardware is said to implement *precise exceptions* just when the state of the processor seen by the exception handler is the same as the state of the processor *before* execution was attempted of the instruction that caused the exception. As a result, after the exception handler code has executed, the handler can restore the state of the executing program and allow it to continue seamlessly from the point where the exception occurred (if appropriate).

*Mika Nyström and Alain J. Martin are with the Computer Science Department of the California Institute of Technology, Pasadena, CA 91125, U.S.A. Rajit Manohar was with the Computer Science Department of the California Institute of Technology, Pasadena, CA 91125; he is now with the Computer Systems Laboratory in the School of Electrical and Computer Engineering at Cornell University, Ithaca NY 14853, U.S.A.

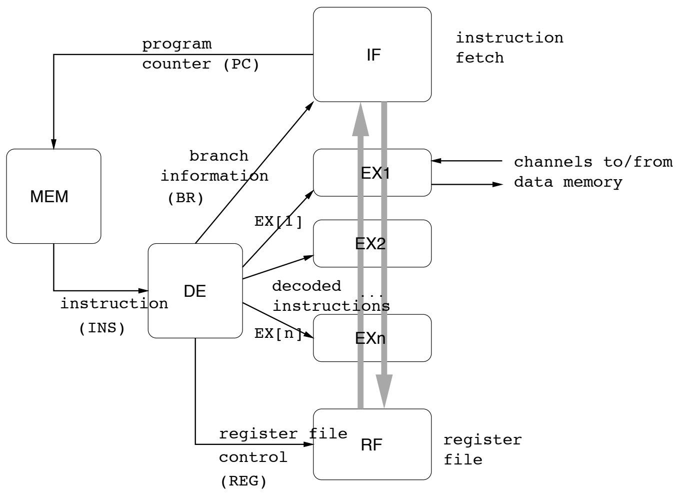

Figure 1. Information flow in a processor without exceptions.

The implementation of such an exception mechanism is complicated by the fact that a modern processor is typically heavily pipelined, and therefore even if a particular instruction has raised an exception, a number of instructions following it may have been partially executed. As a result, some modern high-performance architectures such as the MIPS R8000, DEC Alpha, and Power-2 do not fully implement precise exceptions in hardware.

In this paper, we present a mechanism for the implementation of precise exceptions in asynchronous processors. An important feature of this mechanism is that it permits the presence of a data-dependent number of instructions in the main execution pipeline without the introduction of complex hardware structures. The exception mechanism that we describe was designed as part of the Caltech MiniMIPS asynchronous MIPS processor. The processor was designed between 1995 and 1998, and first silicon was received in early 1999 [5]. The processor was tested and found to be functional.

The paper is organized as follows. In Section 2, we present an overview of a simple pipelined asynchronous processor that does not implement an exception-handling mechanism. The description is based on the MiniMIPS design [5]. In Section 3, we discuss some issues that arise when implementing an exception-handling mechanism in asynchronous architectures and describe the mechanism that we used in the MiniMIPS. We also provide a novel circuit implementation of a particular part of the exception-handling hardware. In Section 4, we extend the design to incorporate external interrupts. Section 5 presents other uses of the exception handling mechanism. Section 6 presents a technique that uses the same exception-handling hardware to implement other non-trivial instructions that result in data hazards or additional design complexity in clocked designs. Section 7 presents related work in clocked and asynchronous design, and Section 8 presents concluding remarks.

2: An Overview of a Processor

In this section, we provide a generic description of an asynchronous processor without interrupts and exceptions. For simplicity, we assume that the processor has a “Harvard architecture,” i.e., the instruction and data memories are not synchronized.

A processor is comprised of a number of “units” (which is the traditional terminology for “process") that communicate with each other. A processor conceptually has a unit that generates the sequence of program counter values and fetches instructions from memory—the "IF" unit, a unit that decodes the instruction stream—the "DE" unit, units that execute the decoded instructions—the "EX" units, and a place on the processor that stores state—the "RF" unit.

The instruction fetch IF generates a program counter value that is sent to the memory. The memory returns an instruction that is sent to the decode DE. DE decodes the instruction and sends the appropriate control information to all the other units: the instruction to be executed is sent to the appropriate execution unit EX_i; information about what state is needed and modified by the instruction is sent to the register file RF; information about control flow is sent to IF. The flow of information is shown in Figure 1. The sequential CHP description of the processor is given below (a brief description of the notation is provided in the appendix):

```
$PROC \equiv$
    *[    $IF :$    $pc := ''next\ \ pc''$;
        $MEM :$    $i := imem[pc]$;
          $DE :$    $id := decode(i)$;
        $EXEC :$   $''read\ \ operands''$;
                   $''execute\ \ instruction''$;
                   $''write\ \ results''$
     ]
```

When this CHP program is decomposed using standard techniques [4], the different parts of the processor shown in Figure 1 can execute concurrently [6]. In particular, the $EXEC$ is decomposed into a number of different execution units EX_i and a register file RF. Once the control information is dispatched to the execution units and the register file by DE, the instruction can execute and asynchronously complete execution. Since multiple execution units are running concurrently, there can be a data-dependent number of instructions executing at any given time. The number of instructions executing in parallel is limited by data-dependencies between instructions, the number of communication channels between the register file and the execution units, the number of execution units, and the instruction fetch bandwidth.

3: Implementing Precise Exceptions

The introduction of exceptions or external interrupts complicates the execution of instructions in a number of ways. When an instruction raises an exception, the exception must be detected and reported to the IF, since the processor must begin execution of the exception handler. In addition, the RF and data memory interface must be notified of the exception so that subsequent instructions do not modify the state of the processor until the exception handler begins execution.

The result of each instruction is modified so that it includes whether the instruction raised an exception. This exception bit is computed by the execution units. The simplest modification to $PROC$ that includes a precise exception-handling mechanism is shown below:

```
$EPROC_0 \equiv$  $e :=$ **false**;
        *[  $IF$ :   [$\neg e \longrightarrow pc :=$ "$next\ \ pc$"
                     [] $e \longrightarrow pc :=$ "$exception\ \ pc$"
                     ];
           $MEM$ :   $i := imem[pc]$;
            $DE$ :   $id := decode(i)$;
          $EXEC$ :  "$read\ \ operands$"; "$execute\ \ instruction$";
                    $e :=$ "$exception\ \ condition$";
            $WB$ :  [$\neg e \longrightarrow$ "$write\ \ results$"
                    [] $e \longrightarrow$ "$set\ \ exception\ \ flags,\ \ save\ \ pc$"
                    ]
         ]
```

The variable e indicates whether or not the executed instruction raised an exception. If e is **false** and remains **false**, the processor $EPROC_0$ is the same as $PROC$. If an instruction raises an exception, e is set to **true**. When e is **true**, the result of the exception-causing instruction does not modify the state of the processor and the instruction fetch IF switches pc to the exception program counter. Instead of writing the results of the instruction in the usual way, the exception-causing instruction sets flags or other status information as specified by the ISA. For the remainder of this section, we take $EPROC_0$ as our reference processor.

A problem with this scheme is that the value of e computed by $EXEC$ affects the next pc value, since it is used by IF. As a result, pipelining this program would not introduce any concurrency between IF and $EXEC$ because IF would have to wait for $EXEC$ to complete before computing the next pc, and $EXEC$ would have to wait for the next pc to be computed before it could receive the decoded instruction.

Since the case e = **true** is rare, we would like to optimize the program so that we break the dependency between IF and $EXEC$ when e = **false**. To do so, we introduce a channel EX that has a one-place buffer that is used to notify IF of the presence of an exception. When an exception occurs, WB inserts a token into this buffer without blocking. The IF polls the state of the buffer and if it finds a token in it, the token is removed and the IF sets the program counter to the exception handler address thereby "detecting" the exception. In the CHP notation, IF uses the probe of channel EX to check if there is a token in the buffer.

Since we do not make assumptions about the speed of buffers, the IF may not immediately notice the presence of a token in the buffer. All we assume is that IF will eventually notice the token and detect the exception. Therefore, $EXEC$ might execute instructions that are invalid—instructions not executed by $EPROC_0$—since the sequence of pc values could have changed. $EXEC$ only executes these invalid instructions after the exception token has been inserted into the EX buffer by WB, and before the exception is detected by IF. We introduce variable va that is set to **true** when the exception is detected by IF (va stands for valid-again; the variable signals the transition from invalid to valid instructions), and variable $valid$ that is set to **false** when the exception token is inserted into the buffer by WB. The processor is therefore executing invalid instructions when $valid$ is set to **false** and va is also **false**.

To eliminate any state change that might occur when the processor executes invalid instructions, we modify WB to a **skip** when $\neg valid \wedge \neg va$ is true. This is the only modification necessary, since all state changes are performed by WB. The resulting program is a correct implementation of $EPROC_0$, and is shown below:

```
$EPROC_1 \equiv$   $valid\uparrow$;
      *[   $IF$ :   [$\neg\overline{EX} \longrightarrow va\downarrow, pc := "next\ \ pc"$
                    | $\overline{EX} \longrightarrow va\uparrow, pc := "exception\ \ pc", EX?$
                    ];
          $MEM$ :   $i := imem[pc]$;
           $DE$ :   $id := decode(i)$;
         $EXEC$ :   $"read\ \ operands"$;
                    $"execute\ \ instruction"$;
                    $e := "exception\ \ condition"$;
           $WB$ :   [$valid \vee va \longrightarrow$ [$\neg e \longrightarrow valid\uparrow, "write\ \ results"$
                                       [] $e \longrightarrow valid\downarrow, EX!, "set\ \ exception\ \ flags,\ \ save\ \ pc"$
                                       ]
                    [] $\neg valid \wedge \neg va \longrightarrow$ **skip**
                    ]
      ]
```

The probe $\overline{EX}$ becomes **true** eventually, causing the IF to detect the presence of the exception token in the buffer. The communication $EX?$ removes the token from the buffer once an exception is detected. WB inserts a token into the buffer by performing an $EX!$ communication when a valid exception is detected.

When exceptions do occur, the exception flags that need to be set are typically not stored in the execution unit that raised the exception. To avoid synchronization across execution units, we can transform $EPROC_1$ so that the update of exception flags is performed by an instruction in the execution pipeline. We call this the “fake exception instruction,” and it corresponds to the instruction that has va set to **true**. When an exception is detected, the program counter pc is set to the address of the instruction before the exception handler. The instruction is fetched and decoded as usual, but since va is set to **true** the pre-exception instruction never executes. The net result is that the exception flags and other status information are updated by the instruction that has va set to **true**. When IF executes next, the program counter is incremented in the usual way, resulting in the correct pc value for the first instruction of the exception handler. The result of these transformations is shown below.

```
$EPROC_2 \equiv$   $valid\uparrow$;
      *[   $IF$ :   [$\neg\overline{EX} \longrightarrow va\downarrow, pc := "next\ \ pc"$
                    | $\overline{EX} \longrightarrow va\uparrow, pc := "pre\_exception\ \ pc", EX?$
                    ];
          $MEM$ :   $i := imem[pc]$;
           $DE$ :   $id := decode(i)$;
         $EXEC$ :   [$\neg va \longrightarrow "read\ \ operands"; \ "execute\ \ instruction"$;
                        $e := "exception\ \ condition"$;
                    [] $va \longrightarrow EINFO?(flags, epc); "set\ \ flags,\ \ save\ \ epc"$
                    ];
           $WB$ :   [$valid \longrightarrow$ [$\neg e \longrightarrow valid\uparrow, "write\ \ results"$
                                  [] $e \longrightarrow valid\downarrow, EX!, EINFO!("exception\ \ flags", pc)$
                                  ]
                    [] $\neg valid \longrightarrow valid := va$
                    ]
      ]
```

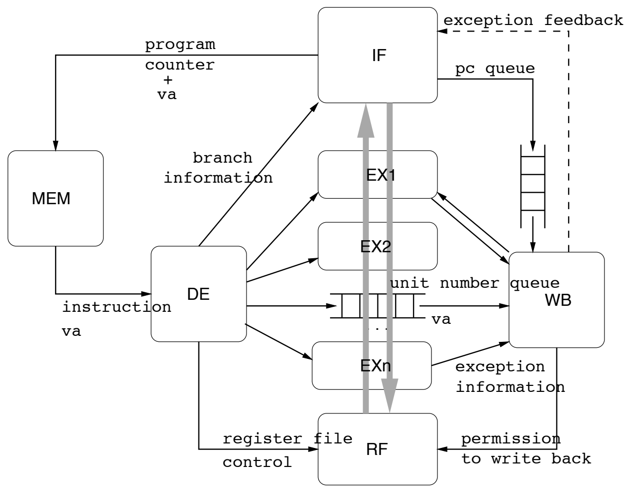

Figure 2. Information flow in a processor with exceptions. EX_1 and RF store state information, and EX_2 cannot generate exceptions.

3.1: Process Decomposition

Next, $EPROC_2$ is decomposed into concurrent processes. Each section of the CHP program $EPROC_2$ is transformed into a pipeline stage that reads the data values it depends on and then computes a result that is passed on to the next stage in the pipeline. For instance, MEM is a process that reads the program counter, fetches the instruction from memory, and sends the instruction to DE, and $EXEC$ is broken down into multiple execution units that execute in parallel with different execution units corresponding to different types of instructions. The pipeline structure that results from this transformation is shown in Figure 2.

Each execution unit computes the exception value e when it executes an instruction. Since the execution units are not synchronized in any way, they may produce their results out of program order. In other words, there are a number of independently executing processes that communicate their e-values to the writeback WB. However, since we must implement precise exceptions, the exception information must be processed in program order.

We introduce a queue that identifies which execution unit is executing the next instruction in program order. This queue is read by WB to determine which execution unit e-value is to be read next. This queue is also a convenient place to store the va bit. The queue is written by DE, since it is responsible for decoding instructions in program order. The pc value used by the writeback is also stored in a separate queue that is connected to IF (which is the process that computes pc values) so that the appropriate information is communicated on the $EINFO$ channel.

In the original CHP, all modifications to the visible state of the processor were performed in WB. When we decompose the CHP into multiple processes, it is convenient to distribute the visible state of the processor among multiple processes. Examples of this include special-purpose registers like `hi` and `lo` in the MIPS ISA that are stored in the multiply/divide unit. Therefore, the writeback WB needs information regarding where the next write is scheduled to take place; this information is also computed by the decode DE and sent to WB by the queue. The parts of the processor where writes occur are modified to read a channel from the writeback that informs them whether writes are permitted to occur or not. Figure 2 shows the modifications to the processor architecture.

3.2: Two Optimizations

In most processors, a large number of instructions are guaranteed not to raise exceptions. When all instructions being executed by one execution unit are guaranteed to terminate normally, we can eliminate the communication between that execution unit and the writeback. This optimization permits the writeback to process an instruction without waiting for any information from the corresponding execution unit. This optimization was used in the asynchronous MiniMIPS processor where the function block and shifter never raise exceptions [5].

When we can quickly (relative to the time taken to execute the instruction completely) determine that an instruction will not raise an exception, reporting this value to the writeback early can improve performance of the processor. This is especially important for instructions that have high execution latency, such as those involved in floating-point arithmetic. The MiniMIPS executes a number of different arithmetic instructions in the adder unit. This unit can raise exceptions in rare cases (instruction traces show that the ratio of instructions that raise exceptions to those that do not is less than 10^{-4}). The unit was optimized so that the latency of exception reporting in the common cases was 40% of the worst-case exception detection latency.

3.3: Evaluating a Probe

The CHP described in the preceding sections can be translated into an asynchronous VLSI implementation using Martin's synthesis method. The only non-standard construct described above is the non-deterministic selection statement in IF. In this section we provide a circuit implementation for a process that can be used to implement this part of the exception mechanism. We can replace the program fragment IF by IF' by the introduction of a process that probes channel EX.

```
$IF'$ :  $E?x$;
         [$\neg x \longrightarrow va\!\downarrow, pc := ''next\ \ pc''$
         ▯$x \longrightarrow va\!\uparrow, pc := ''pre\_exception\ \ pc''$
         ]
```

The process that contains the arbitrated selection statement is shown below.

$$*[[\overline{EX} \longrightarrow E!\textbf{true}, EX? \ \mid \neg\overline{EX} \longrightarrow E!\textbf{false}\ ]]$$

We now provide a circuit implementation for this particular process. Assuming that channels EX and E are both passive, we can write the following handshaking expansion:

```
*[[ $EXi$  $\longrightarrow$  [$Ei$]; $Eto\!\uparrow$; [$\neg Ei$]; $EXo\!\uparrow$; $Eto\!\downarrow$; [$\neg EXi$]; $EXo\!\downarrow$
  | $Ei$  $\longrightarrow$  $Efo\!\uparrow$; [$\neg Ei$]; $Efo\!\downarrow$
 ]]
```

We have eliminated the check for $\neg EXi$ in the handshaking expansion for the second guarded command. The reason we can eliminate this check is that the CMOS implementation of a two-way arbitrated selection statement is weakly fair. Therefore, if EXi is true, the first alternative in the selection will execute eventually. We introduce variables u and v to model the arbitration that is required by the above handshaking expansion.

```
*[[ $EXi$  $\longrightarrow$  $u\!\uparrow$; [$u$]; [$Ei$]; $Eto\!\uparrow$; [$\neg Ei$]; $EXo\!\uparrow$;
                  $Eto\!\downarrow$; [$\neg EXi$]; $u\!\downarrow$; [$\neg u$]; $EXo\!\downarrow$
  | $Ei$  $\longrightarrow$  $v\!\uparrow$; [$v$]; $Efo\!\uparrow$; [$\neg Ei$]; $v\!\downarrow$; [$\neg v$]; $Efo\!\downarrow$
 ]]
```

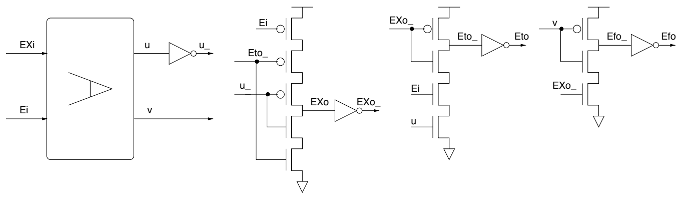

Figure 3. Evaluating the probe of a channel.

We apply process factorization to obtain:

```
*[[ $EXi \longrightarrow u\uparrow$; [$\neg EXi$]; $u\downarrow$
  | $Ei \longrightarrow v\uparrow$; [$\neg Ei$]; $v\downarrow$
 ]]
$\|$
*[[ $u \longrightarrow$ [$Ei$]; $Eto\uparrow$; [$\neg Ei$]; $EXo\uparrow$; $Eto\downarrow$; [$\neg u$]; $EXo\downarrow$
 [] $v \longrightarrow Efo\uparrow$; [$\neg v$]; $Efo\downarrow$
 ]]
```

The first process shown in the decomposition above is an arbiter between EXi and Ei; the compilation of the second process results in the production rules shown below.

$$\begin{array}{rcl} Reset\_ \wedge u \wedge EXo\_ \wedge Ei & \mapsto & Eto\_\downarrow \\ \neg Reset\_ \vee \neg EXo\_ & \mapsto & Eto\_\uparrow \\ Eto\_ & \mapsto & Eto\downarrow \\ \neg Eto\_ & \mapsto & Eto\uparrow \end{array}$$

$$\begin{array}{rcl} Reset\_ \wedge EXo\_ \wedge v & \mapsto & Efo\_\downarrow \\ \neg Reset\_ \vee \neg v & \mapsto & Efo\_\uparrow \\ Efo\_ & \mapsto & Efo\downarrow \\ \neg Efo\_ & \mapsto & Efo\uparrow \end{array}$$

$$\begin{array}{rcl} \neg u\_ \wedge \neg Eto\_ \wedge \neg Ei & \mapsto & EXo\uparrow \\ u\_ \wedge Eto\_ & \mapsto & EXo\downarrow \\ EXo & \mapsto & EXo\_\downarrow \\ \neg EXo & \mapsto & EXo\_\uparrow \end{array}$$

$$\begin{array}{rcl} u & \mapsto & u\_\downarrow \\ \neg u & \mapsto & u\_\uparrow \end{array}$$

The CMOS implementation is shown in Figure 3. For clarity, the reset transistors and staticizers are not shown.

This circuit uses a CMOS arbiter in a non-standard manner. If both Ei and EXi are asserted, and if EXi wins the arbitration, then the circuit completes a handshake on both Ei and EXi *without* raising the v output of the arbiter! In other words, the input Ei is withdrawn from the arbiter even though it loses the arbitration. A careful examination of the handshaking expansion shown above reveals that if u is asserted (EXi wins the arbitration), then the program waits for $\neg Ei$ to be stable before allowing EXi to be deasserted. This property prevents the output of the arbiter from misbehaving.

4: Implementing Interrupts

The mechanism presented in the previous section can be used to handle interrupts as well. The only difference between interrupts and exceptions is that interrupts are external events whereas exceptions are associated with specific instructions. In addition, a processor can disable and enable interrupts by means of kernel mode instructions. In particular, interrupts are disabled by the hardware when the exception handler begins execution to reduce the complexity of the exception handling software.

The simplest modification to the processor that uses precise exceptions is to augment the writeback part of the processor to detect interrupts. Instructions that enable and disable interrupts must communicate with the writeback to enable and disable interrupt checking. Note that the writeback already receives information from various execution units via the exception channel. We use the same communication mechanism to inform the writeback when interrupts are enabled and disabled.

The modified writeback contains a process that repeatedly samples the external interrupt line(s) for the processor. The writeback has a local interrupt enable bit that is used to mask interrupts. This bit is set or cleared by kernel mode instructions. The masked result is used to detect whether an interrupt occurred. The presence of an interrupt is reported using the normal exception mechanism—by performing a communication on the EX channel as before. If the restart address for interrupts differs from that for exceptions, then the EX channel can be augmented to carry information that reports whether the event was an interrupt or exception.

5: Other Uses—Branch Prediction

The exception mechanism that we have presented provides a convenient way to cancel the results of instructions and restart processor execution from a specific address. While we have applied it to the problem that is immediately evident in a pipelined single-scalar asynchronous design, namely exceptions, the mechanism is general. As we have described it, the mechanism is a simple and efficient scheme for the "speculative execution" of possibly exception-raising instructions. The mechanism can be used to do speculative execution for any reason.

5.1: Branch Latency

A vexing problem in the design of a highly pipelined microprocessor is the "branch latency." The high throughput of such a processor makes it impossible to evaluate the branch condition in a branch instruction quickly enough to be able to have the next program counter value ready in time to fetch the next instruction. Indeed, once a program counter value has been generated, the instruction fetch (I-cache) latency may prevent decoding that the corresponding instruction is not a branch in the time available. This can cause program counter generation to become a throughput bottleneck. The amount of time available to decode that the referenced instruction is a branch is, ideally, the time it takes to increment the program counter less the amount of time it takes to distribute the control to select the incremented value or a branch target. In a highly pipelined design, this amount could even be negative!

5.2: Branch Shadow

The "simplest" solution to the branch latency problem is to generate ascending pc values until it has been established that a branch has been fetched. If this is done without any additional thought, a "branch shadow" or "branch delay slot" results. This solution was chosen by the designers of the MIPS ISA, and measurements on compiled code show that the branch shadow is filled with effective instructions about 50% of the time; the rest of the time, the compiler is forced to insert a no-op instruction. This reduces code density and wastes energy, but it simplifies the hardware in a clocked implementation.

5.3: The New Mechanism

We propose to use the exception mechanism to handle the branch latency problem in simple, highly pipelined processors. We introduce a branch prediction mechanism that predicts the next program counter value. The simplest such mechanism is to predict that the branch is not taken; this in effect results in the same fetching behavior as the simple MIPS branch delay slot. However, in this case, because of the writeback cancellation of mis-predicted branches, the depth of the branch delay slot is now *invisible to the programmer.* Also, the depth of the branch delay slot is now, in effect, data-dependent. Code density is clearly better, and energy consumption may be better, depending on the quality of the branch predictor. The drawback when using this mechanism is that in the cases when a delay slot was filled with a useful instruction and the branch was not correctly predicted, an additional instruction has to be cancelled.

Clearly, any branch prediction scheme could be used together with the mechanism we have described. However, the fact that an instruction is a branch may still not be available in time to make the decision for the very next *pc* value without introducing the additional complexity of a branch target buffer. In this case, we can predict that the next instruction is always fetched and use the information that the instruction is a branch as it becomes available to predict a later *pc* value.

The ultimate goal is to generate program counter values that are the most likely, given all information that is available at the time the values are computed. To some extent, the mechanism by which the "most likely" *pc* values are selected is independent of the mechanism used to fetch the corresponding instructions and execute them. The scheme presented in this paper addresses the fetching and execution of the predicted instructions. True to the nature of its asynchronous, communicating-process design, our scheme allows the *pc*-prediction and instruction-fetching and -execution problems largely to be decoupled. Yet, the scheme is efficient and simple to realize at the circuit level.

6: Unpipelining the Processor

The exception mechanism we have presented provides a convenient way to cancel the results of instructions and restart processor execution from a specific address. However, there are some limitations to the use of this mechanism. In this section, we identify these limitations and provide a minor modification of the mechanism that addresses these issues.

Observe that when the instruction that has the valid-again bit set to **true** is being executed, the instructions in the exception handler could have been fetched and decoded by IF, DE, and MEM. However, it is possible that the state changes introduced by the valid-again instruction modify the operation of the MEM and DE units. For instance, the exception handler typically begins execution in kernel mode. The valid-again instruction would modify the mode to kernel mode; however, this affects the accessible address space and the instructions that are allowed to be executed. If the address space check is early in the instruction fetch pipeline, this could lead to erroneous execution. In this particular example, we can avoid the problem by postponing the address space check.

Consider a processor that has a translation lookaside buffer (TLB) for address translation. This translation lookaside buffer is accessed when the statement "$i := imem[pc]$" is executed. When the TLB is modified by special instructions, this modification would interfere with the instruction fetch. A clocked processor typically solves this problem by introducing data hazards around instructions that modify the TLB. We could use a similar approach and introduce an arbiter to enforce mutual exclusion between reads and writes to the TLB. However, this introduces additional circuitry on the commonly used instruction fetch path.

A minor modification to the exception mechanism presented earlier can be used to solve this problem. Suppose we were to implement the TLB write instruction using the exception mechanism. This ensures that the instructions following the TLB write instruction will be discarded ("flushed") by the processor until the exception handler begins execution. However, instead of jumping to the exception handler address, the processor must jump back to the TLB write instruction to resume execution (since this is not a normal, but a "fake" exception). Thus, the exception mechanism flushes the instructions following the TLB write and re-executes the TLB write instruction as the "fake exception instruction" from Section 3; the re-fetched TLB write instruction is marked by the special va bit.

Finally, we introduce a special synchronization channel EX' that is used to block the instruction fetch IF until the re-fetched TLB write instruction is executed. This avoids the mutual exclusion problem, since MEM is waiting for a program counter from IF. Once the TLB write instruction completes, the EX' channel is used to release the instruction fetch and allow it to continue issuing program counter values. The CHP description of these transformations can be found in [3].

To summarize, the mechanism works as follows:

- The TLB write instruction is executed by raising a fake exception;
- The writeback reports this exception to the instruction fetch;
- The instruction fetch detects the exception, and re-issues the fake TLB write instruction and blocks on the EX' channel;
- The fake TLB write instruction executes *in an empty pipeline* and modifies the TLB;
- The completion of the TLB write is reported to the writeback;
- The writeback wakes up the fetch by completing the EX' communication.

This mechanism can be used to implement any infrequent state modification in an empty pipeline, thereby avoiding any additional circuitry on the common execution path.

7: Related Work

In synchronous processors, the clock globally synchronizes all actions, and therefore exception detection is implicitly synchronized with fetching instructions from memory. As a result, synchronous processors implement precise exceptions by allowing a deterministic number of instructions to execute before the exception status of an instruction is checked. The absence of a global clock allows us to break this synchronization. Modern clocked out-of-order processors like the R10000 use a similar mechanism to the one we have just described. The queue between the decode and writeback is similar to the active list in the R10000 [8]. Execution units can directly modify the active list, whereas in our mechanism execution units report their results to the writeback that reads the results in program order. However, the hardware complexity of the R10000 is much higher than the MiniMIPS because of the overhead of implementing out-of-order execution in a clocked processor.

The AMULET is a self-timed clone of the ARM processor [1]. However, it does not have multiple execution units which simplifies the design of an exception handling mechanism. “Fred” is an asynchronous processor with multiple execution units [7]. This processor does not have precise exceptions; it implements a weaker form of exception handling known as functionally precise exceptions. In their implementation, instructions following the one that raised the exception can complete; therefore, the processor keeps track of the instructions that completed so that the exception handler can use this information to recover after an exception. In addition, each instruction saves its inputs so that it can be re-dispatched in the event of an exception [7]. This complicates the exception handling mechanism and increases the software overhead in the exception handler. In addition, the dispatch unit in Fred (which loosely corresponds to IF in our description) receives feedback even when an instruction does not raise an exception, since it keeps track of the status of each instruction. The mechanism proposed here is simpler, more distributed, and implements precise exceptions without synchronizing the IF and the execution units when instructions do not raise exceptions. In addition, the mechanism shown in this paper exercises the arbiter only when an exception does occur. Contrast this with the mechanism in Fred, which requires arbitration even when exceptions do not occur.

8: Conclusion

In this paper, we have described an algorithm for implementing an exception mechanism in pipelined asynchronous processors. Our proposed mechanism is flexible (it allows any number of instructions to be in progress when an exception occurs) and efficient (it incurs a significant performance penalty only when exceptions actually do occur). The mechanism is not restricted to the implementation of “software exceptions”—it can also be used for interrupts and other uncommon events that affect control flow, e.g., branch mispredicts.

Acknowledgments

The research described in this paper was supported in part by the Defense Advanced Research Projects Agency (DARPA) and monitored by the Office of Army Research. Mika Nyström was supported in part by an Okawa Foundation Fellowship and an IBM Research fellowship. We thank the members of the asynchronous VLSI group at Caltech who participated in the design of the MiniMIPS processor for lively and animated discussions.

A: Notation

The notation we use is based on Hoare’s CSP [2]. A full description of the notation and its semantics can be found in [4]. What follows is a short and informal description of the notation we use.

- Assignment: $a := b$. This statement means “assign the value of b to a.” We also write $a\uparrow$ for $a := true$, and $a\downarrow$ for $a := false$.
- Selection: $[G1 \rightarrow S1 \, [] \, ... \, [] \, Gn \rightarrow Sn]$, where Gi’s are boolean expressions (guards) and Si’s are program parts. The execution of this command corresponds to waiting until one of the guards is $true$, and then executing one of the statements with a $true$ guard. The notation

$[G]$ is short-hand for $[G \to \mathit{skip}]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "$|$" instead of "$[]$."

- Repetition: $*[G1 \to S1 \; [] \; ... \; [] \; Gn \to Sn]$. The execution of this command corresponds to choosing one of the $true$ guards and executing the corresponding statement, repeating this until all guards evaluate to false. The notation $*[S]$ is short-hand for $*[true \to S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is $true$ iff a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.

References

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

[2] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–677, 1978

[3] Rajit Manohar. The Impact of Asynchrony on Computer Architecture. Ph.D. thesis, CS-TR-98-12, California Institute of Technology, July 1998.

[4] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.

[5] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. The Design of an Asynchronous MIPS R3000 Processor. *Proceedings of the 17th Conference on Advanced Research in VLSI*. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

[6] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351–373, MIT Press, 1991.

[7] William F. Richardson. Architectural Considerations in a Self-Timed Processor Design. Ph.D. thesis, Department of Computer Science, University of Utah, 1996.

[8] Ken Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, **16**(2):28–40, April 1996.