4.9 Exceptions

> “ To make a computer with automatic program-interruption facilities behave [sequentially] was not an easy matter, because the number of instructions in various stages of processing when an interrupt signal occurs may be large.
> *Fred Brooks, Jr., Planning a Computer System: Project Stretch, 1962.*

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the toughest part to make fast. One of the demanding tasks of control is implementing *exceptions* and *interrupts*—events other than branches that change the normal flow of instruction execution. They were initially created to handle unexpected events from within the processor, like floating-point overflow. The same basic mechanism was extended for I/O devices to communicate with the processor, as we will see in COD Chapter 5 (Large and fast: Exploiting memory hierarchy).

Exception: Also called ***interrupt***. An unscheduled event that disrupts program execution; used to detect overflow.

Interrupt: An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

Many architectures and authors do not distinguish between interrupts and exceptions, often using either name to refer to both types of events. For example, the Intel x86 uses interrupt. We use the term *exception* to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal or external; we use the term *interrupt* only when the event is externally caused. Here are examples showing whether the situation is internally generated by the processor or externally generated and the name that ARM uses:

Type of event	From where?	ARMv8 terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Floating-point arithmetic overflow or underflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Many of the requirements to support exceptions come from the specific situation that causes an exception to occur. Accordingly, we will return to this topic in COD Chapter 5 (Large and fast: Exploiting memory hierarchy), when we will better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting types of exceptions that arise from the portions of the instruction set and implementation that we have already discussed.

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a processor, which determines the clock cycle time and thus performance. Without proper attention to exceptions during design of the control unit, attempts to add exceptions to an intricate implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

PARTICIPATION ACTIVITY 4.9.1: Exceptions and interrupts.

1) According to ARM convention, the term interrupt refers to an unscheduled event caused by an external source.
   - ○ True
   - ○ False

2) Exception handling is not an essential feature of processor's control unit.
   - ○ True
   - ○ False

How exceptions are handled in the LEGv8 architecture

The types of exceptions that our current implementation can generate are execution of an undefined instruction, floating-point overflow and underflow, or a hardware malfunction. We'll assume a hardware malfunction occurs during the instruction `ADD X1, X2, X1` as the example exception in the next few pages. The basic action that the processor must perform when an exception occurs is to save the address of the unfortunate instruction in the *exception link register* (ELR) and then transfer control to the operating system at some specified address.

The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to a malfunction, or stopping the execution of the program and reporting an error. After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the ELR to determine where to restart the execution of the program. In COD Chapter 5 (Large and fast: Exploiting memory hierarchy), we will look more closely at the issue of restarting the execution.

For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception. The method used in the LEGv8 architecture is to include a register (called the *Exception Syndrome Register* or *ESR*), which holds a field that indicates the reason for the exception.

A second method is to use *vectored interrupts*. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception, possibly added to a base register that points to memory range for vectored interrupts. For example, we might define the following exception vector addresses to accommodate these exception types:

Exception type	Exception vector address to be added to a Vector Table Base Register
Unknown Reason	$00\ 0000_{two}$
Floating-point arithmetic exception	$10\ 1100_{two}$
System Error (hardware malfunction)	$10\ 1111_{two}$

Vectored interrupt: An interrupt for which the address to which control is transferred is determined by the cause of the exception.

The operating system knows the reason for the exception by the address at which it is initiated. When the exception is not vectored, as in LEGv8, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause. For architectures with vectored exceptions, the addresses might be separated by, say, 32 bytes or eight instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence.

We can perform the processing required for exceptions by adding a few extra registers and control signals to our basic implementation and by slightly extending control. Let's assume that we are implementing the exception system with the single interrupt entry point being the address 0000 0000 1C09 0000_{hex}. (Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to our current LEGv8 implementation:

- *ELR:* A 64-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- *ESR:* A register used to record the cause of the exception. In the LEGv8 architecture, this register is 32 bits, although some bits are currently unused. Assume there is a field that encodes the three possible exception sources mentioned above, with 8 representing an undefined instruction, 10 representing arithmetic overflow or underflow, and 12 representing hardware malfunction.

PARTICIPATION ACTIVITY 4.9.2: LEGv8 exception handling.

1) When an exception occurs in LEGv8, the processor first saves the address of the offending instruction in the ________.

Check Show answer

2) In LEGv8, the ________ register stores the cause of an exception and communicates that information to the operating system for exception handling.

Check Show answer

3) For a vectored interrupt, the cause of an exception determines the ________ that control is transferred to.

Check Show answer

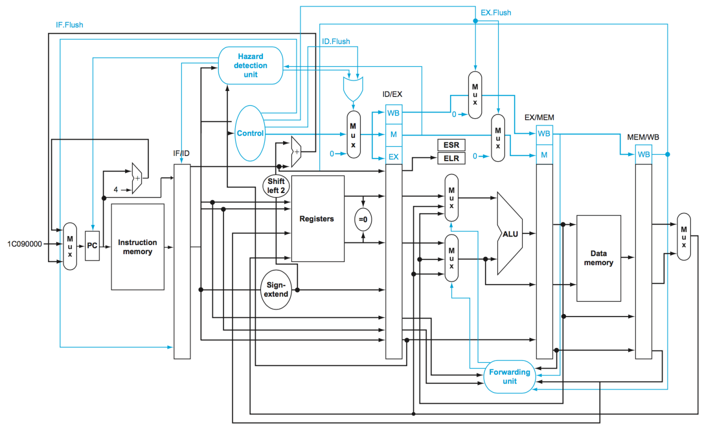
Exceptions in a pipelined implementation

A pipelined implementation treats exceptions as another form of control hazard. For example, suppose there is a hardware malfunction in an add instruction. Just as we did for the taken branch in the previous section, we must flush the instructions that follow the `ADD` instruction from the pipeline and begin fetching instructions from the new address. We will use the same mechanism we used for taken branches, but this time the exception causes the deasserting of control lines.

When we dealt with branch misprediction, we saw how to flush the instruction in the IF stage by turning it into a `nop`. To flush instructions in the ID stage, we use the multiplexer already in the ID stage that zeros control signals for stalls. A new control signal, called ID.Flush, is ORed with the stall signal from the hazard detection unit to flush during ID. To flush the instruction in the EX phase, we use a new signal called EX.Flush to cause new multiplexers to zero the control lines. To start fetching instructions from location 0000 0000 1C09 0000_{hex}, which we are using as the LEGv8 exception address, we simply add an additional input to the PC multiplexer that sends 0000 0000 1C09 0000_{hex} to the PC. The figure below shows these changes.

Figure 4.9.1: The datapath with controls to handle exceptions (COD Figure 4.64).

The key additions include a new input with the value 0000 0000 1C09 0000_{hex} in the multiplexer that supplies the new PC value; an ESR register to record the cause of the exception; and an ELR register to save the address of the instruction that caused the exception. The 0000 0000 1C09 0000_{hex} input to the multiplexer is the initial address to begin fetching instructions in the event of an exception.



This example points out a problem with exceptions: if we do not stop execution in the middle of the instruction, the programmer will not be able to see the original value of register `X1` because it will be clobbered as the Destination register of the `ADD` instruction. If we assume the exception is detected during the EX stage, we can use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage. Many exceptions require that we eventually complete the instruction that caused the exception as if it executed normally. The easiest way to do this is to flush the instruction and restart it from the beginning after the exception is handled.

The final step is to save the address of the offending instruction in the *exception link register* (ELR). The figure above shows a stylized version of the datapath, including the branch hardware and necessary accommodations to handle exceptions.

Example 4.9.1: Exception in a pipelined computer.

Given this instruction sequence,

40_{hex} `SUB  X11, X2, X4`
44_{hex} `AND  X12, X2, X5`
48_{hex} `ORR  X13, X2, X6`
$4C_{hex}$ `ADD  X1,  X2, X1`
50_{hex} `SUB  X15, X6, X7`
54_{hex} `LDUR X16, [X7,#100]`
`. . .`

assume the instructions to be invoked on an exception begin like this:

$1C090000_{hex}$ `STUR    X26, [X0,#1000]`
$1C090004_{hex}$ `STUR    X27, [X0,#1008]`
`. . .`

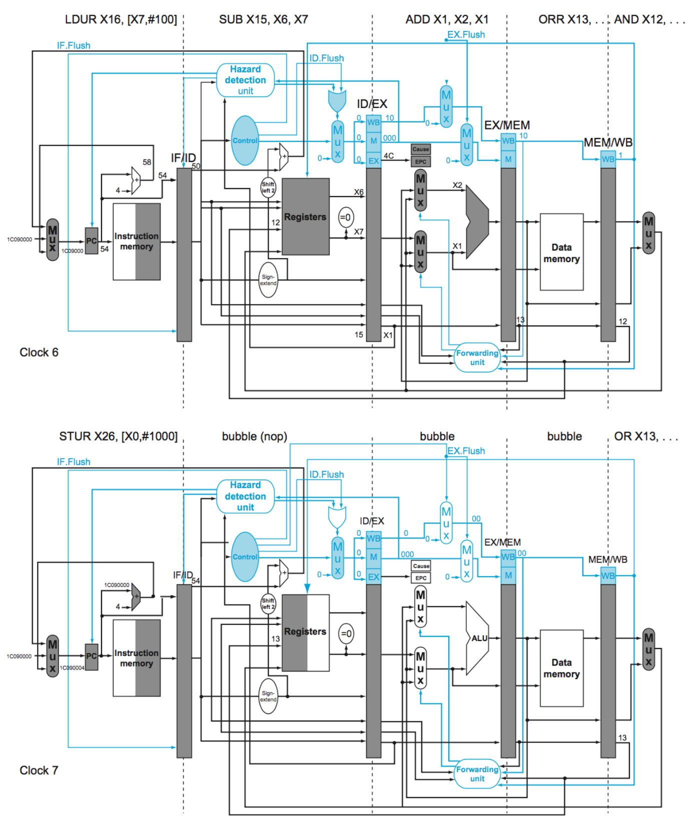
Show what happens in the pipeline if an exception occurs in the `ADD` instruction.

Answer

The figure below shows the events, starting with the add instruction in the EX stage. Assume the hardware malfunction is detected during that phase, and 0000 0000 1C09 0000_{hex} is forced into the PC. Clock cycle 7 shows that the `ADD` and following instructions are flushed, and the first instruction of the exception code is fetched. Note that the address of the instruction *following* the `ADD` is saved: $4C_{hex}$ + 4 = 50_{hex}.

Figure 4.9.2: The result of an exception due to hardware malfunction in the add instruction (COD Figure 4.65).

The exception is detected during the EX stage of clock 6, saving the address following the add in the ELR register (4C + 4 = 50_{hex}). It causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the `ADD`. Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine —`STUR X26, [X0,#1000]`—from instruction location 0000 0000 1C09 0000_{hex}. Note that the `AND` and `ORR` instructions, which are prior to the `ADD`, still complete.



We mentioned several examples of exceptions, and we will see others in COD Chapter 5 (Large and fast: Exploiting memory hierarchy). With five instructions active in any clock cycle, the challenge is to associate an exception with the appropriate instruction. Moreover, multiple exceptions can occur simultaneously in a single clock cycle. The solution is to prioritize the exceptions so that it is easy to determine which is serviced first. In LEGv8 implementations, the hardware sorts exceptions so that the earliest instruction is interrupted.

I/O device requests and hardware malfunctions are not associated with a specific instruction, so the implementation has some flexibility as to when to interrupt the pipeline. Hence, the mechanism used for other exceptions works just fine.

The ELR captures the address of the interrupted instructions, and the ESR records the highest priority exception in a clock cycle if more than one exception occurs.

Hardware/Software Interface

The hardware and the operating system must work in conjunction so that exceptions behave as you would expect. The hardware contract is normally to stop the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then branch to a prearranged address. The operating system contract is to look at the cause of the exception and act appropriately. For an undefined instruction or hardware failure, the operating system normally kills the program and returns an indicator of the reason. For an I/O device request or an operating system service call, the operating system saves the state of the program, performs the desired task, and, at some point in the future, restores the program to continue execution. In the case of I/O device requests, we may often choose to run another task before resuming the task that requested the I/O, since that task may often not be able to proceed until the I/O is complete. Exceptions are why the ability to save and restore the state of any task is critical. One of the most important and frequent uses of exceptions is handling page faults and TLB exceptions; COD Chapter 5 (Large and fast: Exploiting memory hierarchy) describes these exceptions and their handling in more detail.

Elaboration

The difficulty of always associating the proper exception with the correct instruction in pipelined computers has led some computer designers to relax this requirement in noncritical cases. Such processors are said to have imprecise interrupts or imprecise exceptions. In the example above, PC would normally have 58_{hex} at the start of the clock cycle after the exception is detected, even though the offending instruction is at address $4C_{hex}$. A processor with imprecise exceptions might put 58_{hex} into ELR and leave it up to the operating system to determine which instruction caused the problem. LEGv8 and the vast majority of computers today support precise interrupts or precise exceptions. One reason is designers of a deeper pipeline processor might be tempted to record a different value in the ELR, which would create headaches for the OS. To prevent them, the deeper pipeline would likely be required to record the same PC that would have been recorded in the five-stage pipeline. It is simpler for everyone to just record the PC of the faulting instruction instead. (Another reason is to support virtual memory, which we shall see in COD Chapter 5 (Large and fast: Exploiting memory hierarchy).)

Imprecise interrupt: Also called ***imprecise exception***. Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.

Precise interrupt: Also called ***precise exception***. An interrupt or exception that is always associated with the correct instruction in pipelined computers.

PARTICIPATION ACTIVITY 4.9.3: Exceptions in pipelined computers.

1) In a pipeline implementation, offending arithmetic overflow instructions are detected in the ________ stage of the pipeline to prevent the results from being written to the ________ stage.

- ○ IF, ID
- ○ EX, MEM
- ○ EX, WB

2) In the majority of LEGv8 implementations, multiple thrown exceptions are interrupted ________.

- ○ according to which instruction causes the largest exception
- ○ according to which offending instruction is earliest
- ○ randomly

3) A(n) ________ is always associated with an exact instruction in pipelined computers.

- ○ precise interrupt
- ○ imprecise interrupt

Elaboration

We show that LEGv8 uses the exception entry address 0000 0000 1C09 0000_{hex}, which is based on the ARMv8 Model Architecture. ARMv8 can have different exception entry addresses, depending on the platform.

Elaboration

The LEGv8 architecture has three levels of exception, each with their own ELR and ESR registers, as we'll see in COD Chapter 5 (Large and fast: Exploiting memory hierarchy).

PARTICIPATION ACTIVITY 4.9.4: Check yourself: Detecting Exceptions.

A five-stage pipeline (IF, ID, EX, MEM, WB) executes the following instruction sequence:

```
XXX X1, X2, X1  // undefined instruction
SUB X1, X2, X1  // hardware error
```

1) Which exception should be recognized first in the above sequence?

- ○ undefined instruction
- ○ hardware error

Provide feedback on this section