5.6 Exceptions

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.)

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make fast. One of the hardest parts of control is implementing **exceptions** and **interrupts**—events other than branches or jumps that change the normal flow of instruction execution. An exception is an unexpected event from within the processor; arithmetic overflow is an example of an exception. An interrupt is an event that also causes an unexpected change in control flow but comes from outside of the processor. Interrupts are used by I/O devices to communicate with the processor, as we will see in Chapter 8.

Many architectures and authors do not distinguish between interrupts and exceptions, often using the older name *interrupt* to refer to both types of events. We follow the MIPS convention, using 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. The Intel IA-32 architecture uses the word *interrupt* for all these events.

Interrupts were initially created to handle unexpected events like arithmetic overflow and to signal requests for service from I/O devices. The same basic mechanism was extended to handle internally generated exceptions as well. Here are some examples showing whether the situation is generated internally by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	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 Chapter 7, when we discuss memory hierarchies, and in Chapter 8, when we discuss I/O, and we better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting two 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 machine, 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 a complicated implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

How Exceptions Are Handled

The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow. The basic action that the machine must perform when an exception occurs is to save the address of the offending instruction in the exception program counter (EPC) 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 an overflow, 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 EPC to determine where to restart the execution

of the program. In Chapter 7, 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 MIPS architecture is to include a status register (called the *Cause register)*, which holds a field that indicates the reason for the exception.

vectored interrupt An interrupt for which the address to which control is transferred is determined by the cause of 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. For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	$C000\ 0000_{hex}$
Arithmetic overflow	$C000\ 0020_{hex}$

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or 8 instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

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 the finite state machine. Let's assume that we are implementing the exception system used in the MIPS architecture. (Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to the datapath:

- *EPC:* A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- *Cause:* A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused. Assume that the low-order bit of this register encodes the two possible exception sources mentioned above: undefined instruction = 0 and arithmetic overflow = 1.

We will need to add two control signals to cause the EPC and Cause registers to be written; call these *EPCWrite* and *CauseWrite*. In addition, we will need a 1-bit control signal to set the low-order bit of the Cause register appropriately; call this signal *IntCause*. Finally, we will need to be able to write the *exception address*, which is the operating system entry point for exception handling, into the PC; in

the MIPS architecture, this address is 8000 0180_{hex}. (The SPIM simulator for MIPS uses 8000 0080_{hex}.) Currently, the PC is fed from the output of a three-way multiplexor, which is controlled by the signal PCSource (see Figure 5.28 on page 323). We can change this to a four-way multiplexor, with additional input wired to the constant value 8000 0180_{hex}. Then PCSource can be set to 11_{two} to select this value to be written into the PC.

Because the PC is incremented during the first cycle of every instruction, we cannot just write the value of the PC into the EPC, since the value in the PC will be the instruction address plus 4. However, we can use the ALU to subtract 4 from the PC and write the output into the EPC. This requires no additional control signals or paths, since we can use the ALU to subtract, and the constant 4 is already a selectable ALU input. The data write port of the EPC, therefore, is connected to the ALU output. Figure 5.39 shows the multicycle datapath with these additions needed for implementing exceptions.

Using the datapath of Figure 5.39, the action to be taken for each different type of exception can be handled in one state apiece. In each case, the state sets the Cause register, computes and saves the original PC into the EPC, and writes the exception address into the PC. Thus, to handle the two exception types we are considering, we will need to add only the two states, but before we add them we must determine how to check for exceptions, since these checks will control the arcs to the new states.

How Control Checks for Exceptions

Now we have to design a method to detect these exceptions and to transfer control to the appropriate state in the exception states. Figure 5.40 shows the two new states (10 and 11) as well as their connection to the rest of the finite state control. Each of the two possible exceptions is detected differently:

- *Undefined instruction:* This is detected when no next state is defined from state 1 for the op value. We handle this exception by defining the next-state value for all op values other than `lw`, `sw`, 0 (R-type), `j`, and `beq` as state 10. We show this by symbolically using *other* to indicate that the op field does not match any of the opcodes that label arcs out of state 1 to the new state 10, which is used for this exception.
- *Arithmetic overflow:* The ALU, designed in Appendix B, included logic to detect overflow, and a signal called *Overflow* is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state (state 11) for state 7, as shown in Figure 5.40.

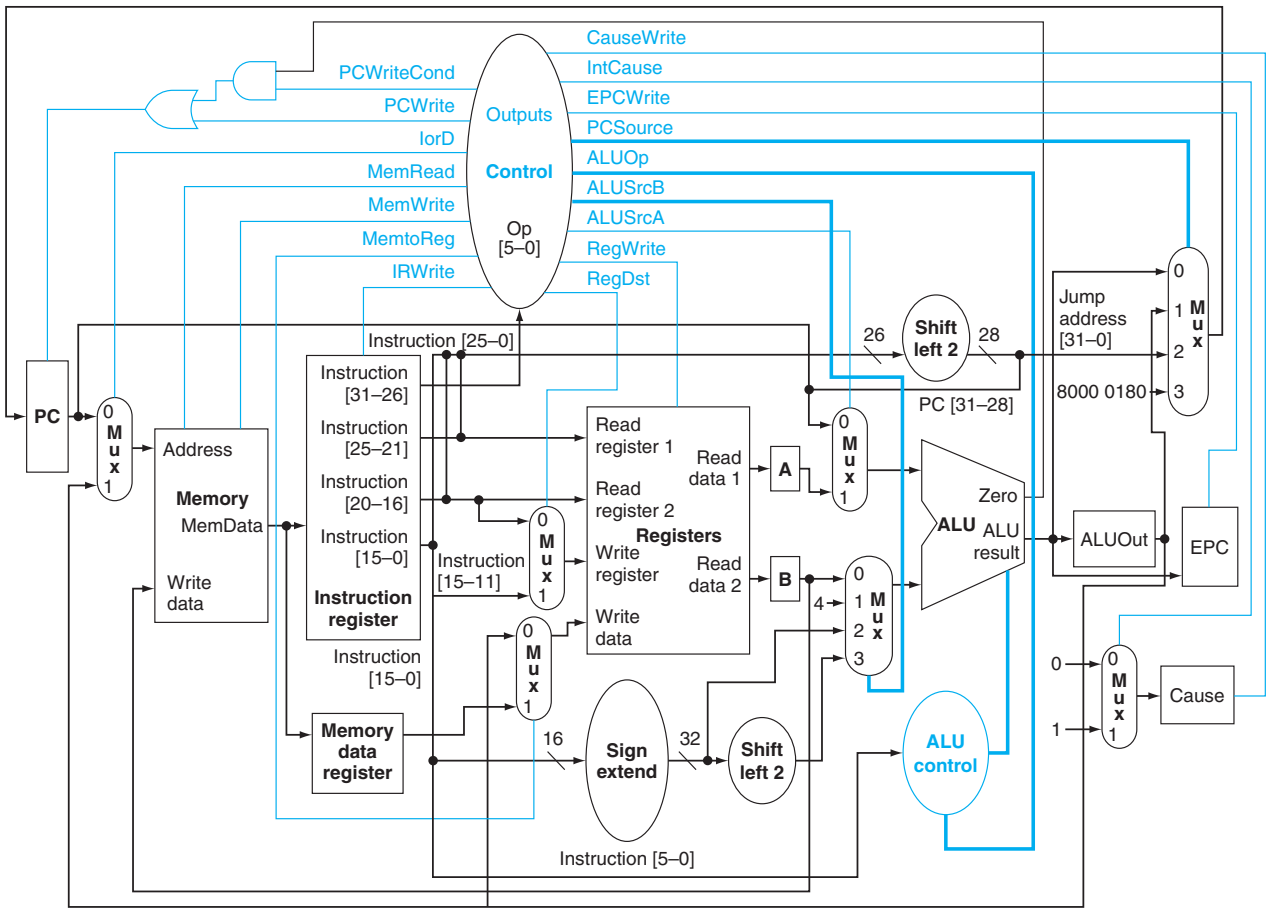

FIGURE 5.39 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers. For simplicity, this figure does not show the ALU overflow signal, which would need to be stored in a one-bit register and delivered as an additional input to the control unit (see Figure 5.40 to see how it is used).

Figure 5.40 represents a complete specification of the control for this MIPS subset with two types of exceptions. Remember that the challenge in designing the control of a real machine is to handle the variety of different interactions between instructions and other exception-causing events in such a way that the control logic remains both small and fast. The complex interactions that are possible are what make the control unit the most challenging aspect of hardware design.

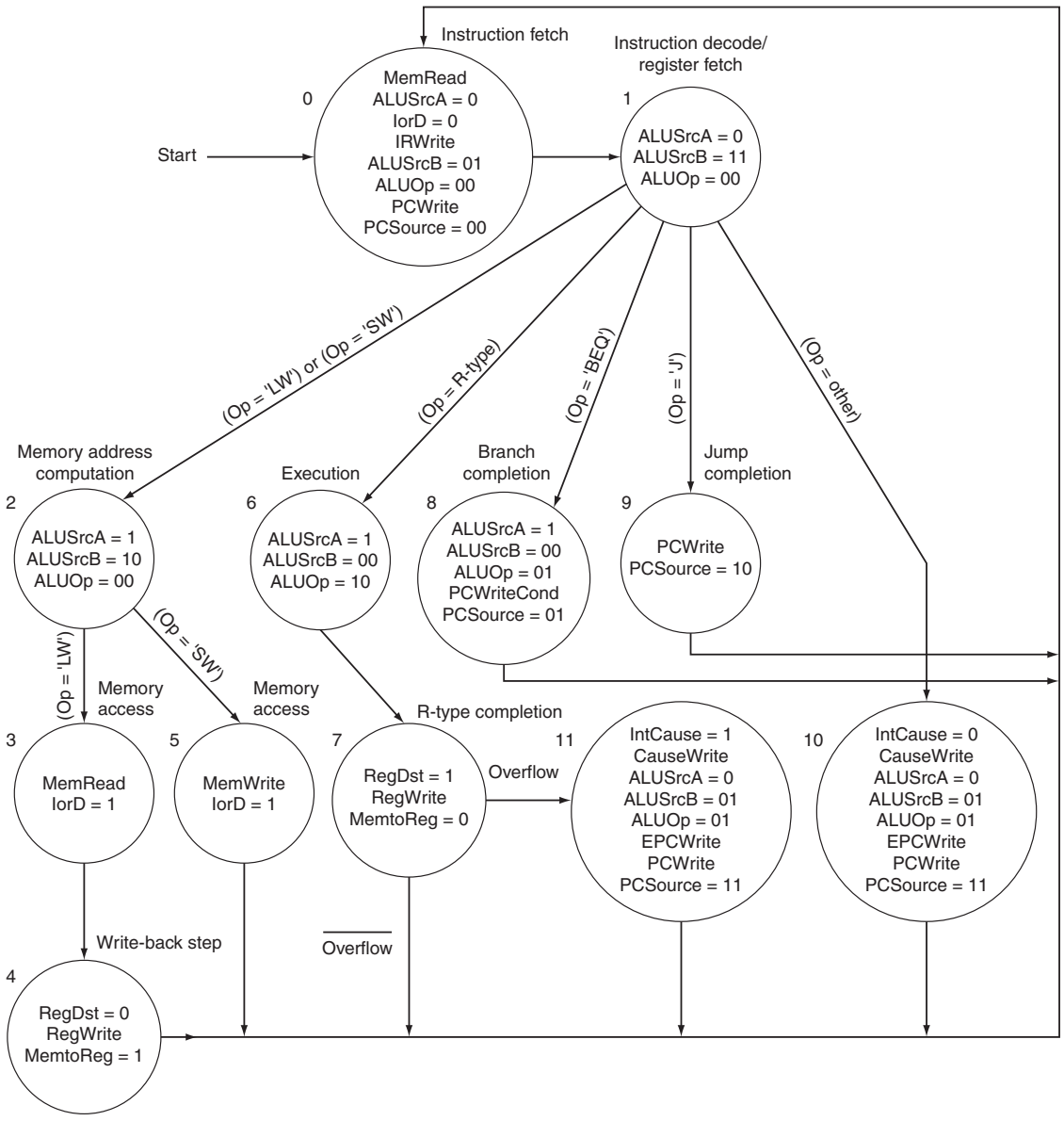

FIGURE 5.40 This shows the finite state machine with the additions to handle exception detection. States 10 and 11 are the new states that generate the appropriate control for exceptions. The branch out of state 1 labeled (*Op = other*) indicates the next state when the input does not match the opcode of any of `lw`, `sw`, `0` (R-type), `j`, or `beq`. The branch out of state 7 labeled *Overflow* indicates the action to be taken when the ALU signals an overflow.