Pipeline: Exceptions

Portions of these slides are derived from:
Dave Patterson © UCB

Exceptions

- Exceptions definition: "*unexpected change in control flow*"
- Another form of control hazard.

For example:

```
add R1, R2, R1;    causing an arithmetic overflow
sw   R3, 400(R1);
add R5,  R1, R2;
```

Invalid r1 contaminates other registers or memory locations!

Types of Exceptions

Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O device request	Input/output interruption	Device interrupt	Exception (L0 to L7 autovector)	Vectored interrupt
Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction)— on Macintosh	Interrupt (INT instruction)
Tracing instruction execution	Not applicable	Exception (trace fault)	Exception (trace)	Interrupt (single-step trap)
Breakpoint	Not applicable	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
Integer arithmetic overflow or underflow; FP trap	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
Page fault (not in main memory)	Not applicable (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt (page fault)
Misaligned memory accesses	Program interruption (specification exception)	Not applicable	Exception (address error)	Not applicable
Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/reserved fault)	Exception (illegal instruction or break-point/unimplemented instruction)	Interrupt (invalid opcode)
Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	Not applicable
Power failure	Machine-check interruption	Urgent interrupt	Not applicable	Nonmaskable interrupt

Exception classification

- ***Synchronous vs Asynchronous*** - If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is synchronous. Otherwise asynchronous.
  - Except for hardware malfunctions, asynchronous events are caused by devices external to the CPU and memory.
  - Asynchronous events usually are easier to handled because asynchronous events can be handled after the completion of the current instruction.

Exception classification

- ***User requested versus coerced***- If the user task directly asks for it, it is a user-requested event.
  - User-requested exceptions are not really exceptions
    - They can be handled after the instruction has completed
  - Coerced exceptions are caused by some hardware event that is not under the control of user program
    - Harder to implement, not predictable

Exception classification

- ***Within instruction versus between instructions*** - whether the event (in the middle of execution of an instruction) does not allow the instruction completion (usually synchronous) or whether it allows instruction completion.
  - It is harder to implement synchronous exceptions within instructions, since the instruction must be stopped and restarted
  - Asynchronous exceptions within instructions arise from hardware malfunction and terminate program

- Resume versus terminate – if the program's execution always stops after the exception, it is a terminate event
  - It is easier to implement exception that terminate program

Exception classification

Exception type	Synchronous vs. asynchronous	User request vs. coerced	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Between	Resume
Invoke operating system	Synchronous	User request	Between	Resume
Tracing instruction execution	Synchronous	User request	Between	Resume
Breakpoint	Synchronous	User request	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	Within	Resume
Page fault	Synchronous	Coerced	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	Within	Resume
Memory protection violations	Synchronous	Coerced	Within	Resume
Using undefined instructions	Synchronous	Coerced	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Within	Terminate
Power failure	Asynchronous	Coerced	Within	Terminate

Exceptions in Simple five-stage pipeline

- Due to the overlapping of instruction execution, multiple interrupts can occur in the same clock cycle. Sources of interrupt in the MIPS are as follows:

- Instruction Fetch, & Memory stages
  - Page fault on instruction/data fetch
  - Misaligned memory access
  - Memory-protection violation
- Instruction Decode stage
  - Undefined/illegal opcode
- Execution stage
  - Arithmetic exception
- Write-Back stage
  - *No exceptions!*

Two simple cases

- Example 1. I/0 device interrupt
  - When the interrupt occurs, the current instruction completes, and the current context is saved. After the interrupt is serviced, the context is restored, and the execution resumes from the next instruction.

- Example 2. Page fault / arithmetic overflow
  - The current instruction cannot be completed. So it is aborted, the exception is handled, and the execution resumes from the instruction causing the exception.

Precise exception

- **Precise exception** preserve the model that instructions execute in program-generated order, one at a time
  - If an exception occurs, the processor can recover from it
- To implement precise interrupts, the interrupt handler needs to create the illusion of sequential instruction execution.
- To implement a precise exception,
  - Complete all instructions before the faulting instruction
  - Undo all instructions after the interrupting instruction, and
  - Restart from the faulting instruction.
- Otherwise, the interrupt becomes imprecise

What happens during a precise exception

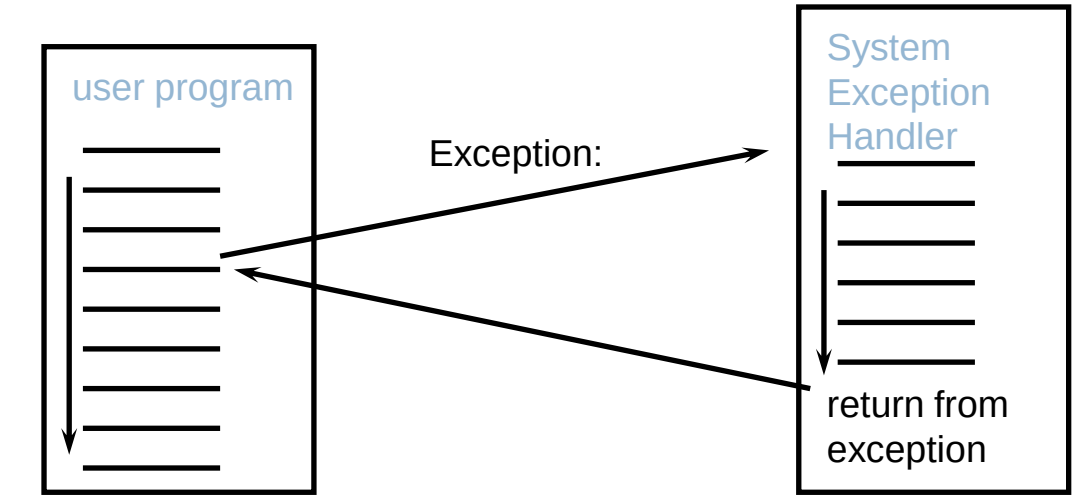
In The Hardware

- The pipeline has to
  1) stop executing the offending instruction in midstream,
  2) let all preceding instructions complete,
  3) flush all succeeding instructions,
  4) set a register to show the cause of the exception,
  5) save the address of the offending instruction, and
  6) then jump to a prearranged address (the address of the exception handler code)

In The Software

- The software (OS) looks at the cause of the exception and "deals" with it
- OS could kill the program

Exceptions



normal control flow:
sequential, jumps, branches, calls, returns

Exception = non-programmed control transfer

- system takes action to handle the exception
  - must record the address of the offending instruction
  - record any other information necessary to return afterwards
- returns control to user
- must save & restore user state

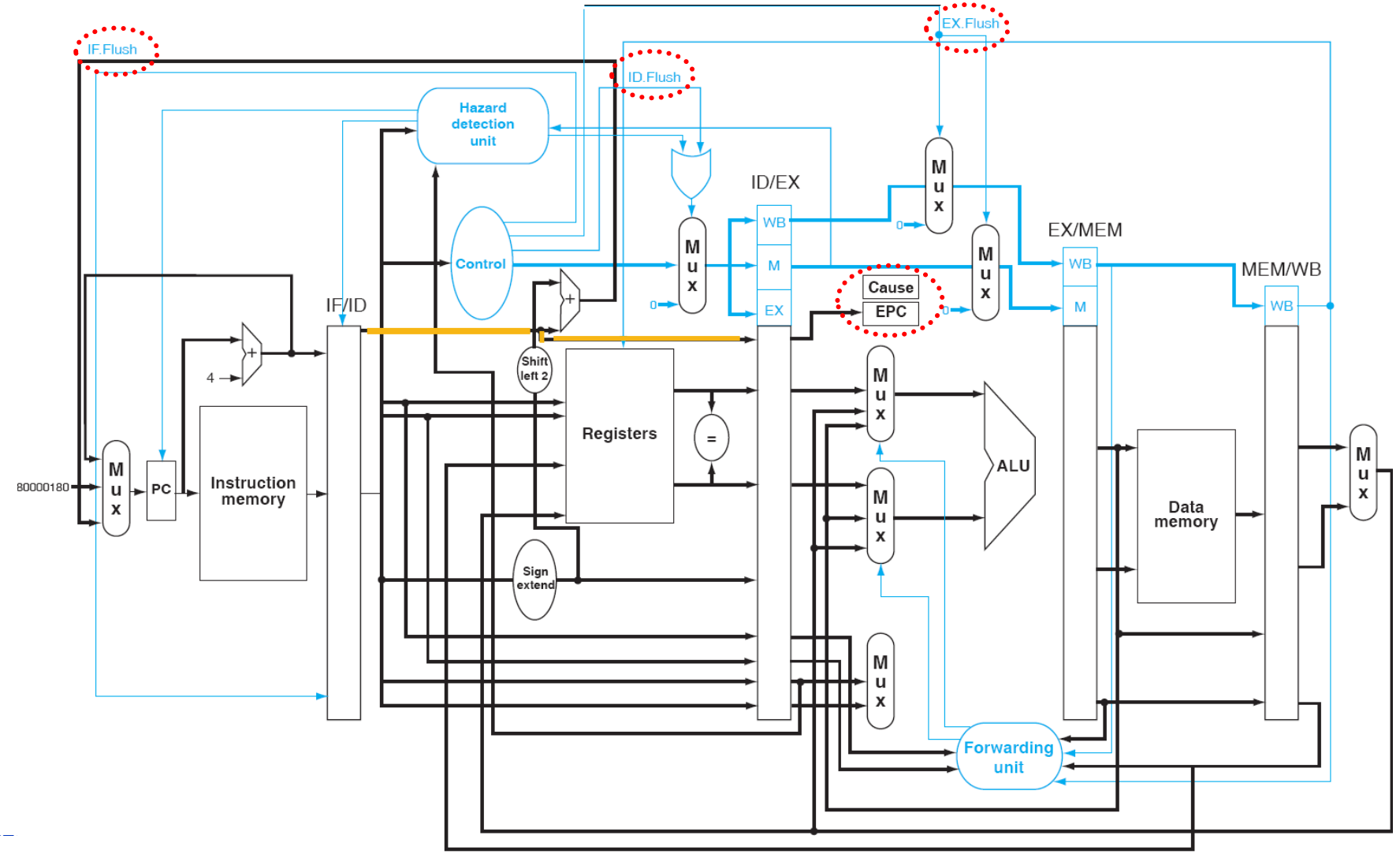
Additions to MIPS ISA to support Exceptions

- EPC (Exceptional Program Counter)
  - A 32-bit register
  - Hold the address of the offending instruction
- Cause
  - A 32-bit register in MIPS (some bits are unused currently.)
  - Record the cause of the exception
- Status - interrupt mask and enable bits and determines what exceptions can occur.
- Control signals to write EPC , Cause, and Status
- Be able to write exception address into PC, increase mux set PC to exception address (MIPS uses 8000 00180$_{hex}$).
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor); PC = PC – 4

- What else?

flush all succeeding instructions in pipeline

Additions to MIPS ISA to support Exceptions



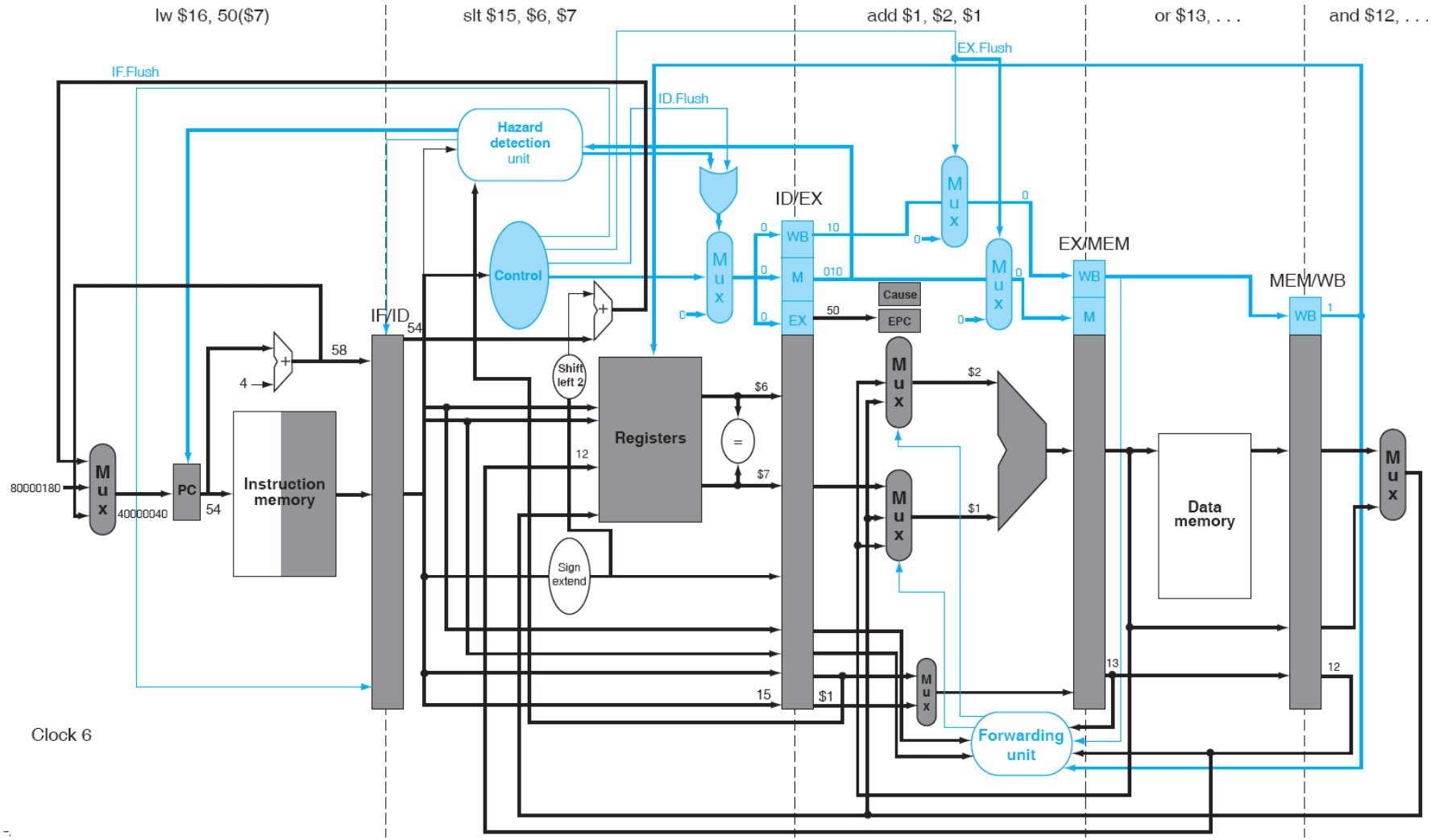
Exceptions Example

```
40hex   sub   $11,  $2,   $4
44hex   and   $12,  $2,   $5
48hex   or    $13,  $2,   $6
4Chex   add   $1,   $2,   $1; // arithmetic overflow
50hex   stl   $15,  %6,   $7
54hex   lw    $16,  50($7)
```

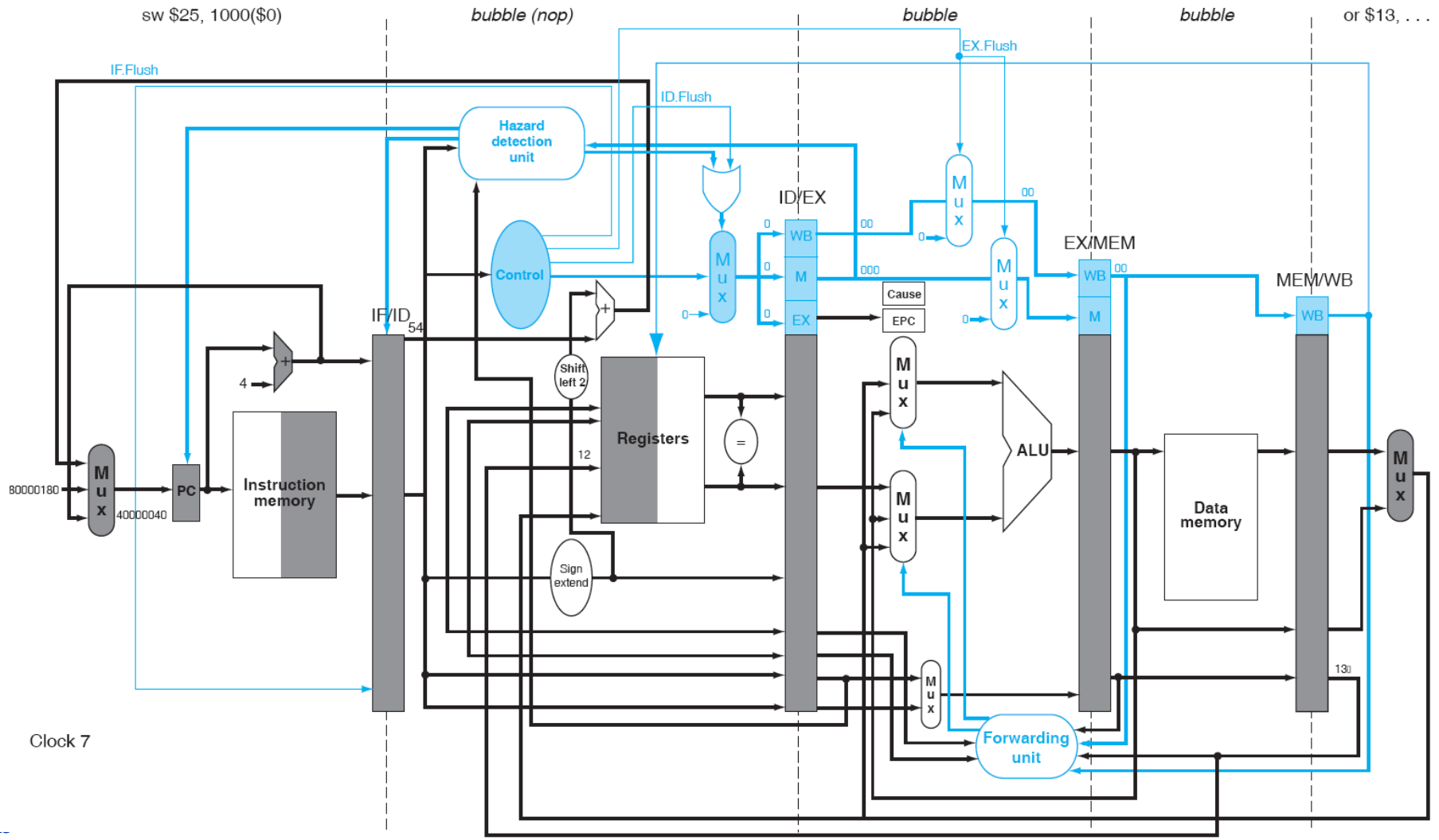
Exception handling program:

```
40000040hex   sw   $25,   1000($0)
40000044hex   sw   $12,   1004($0)
```

Exceptions Example



Exceptions Example



How Pipelines Complicate Interrupt Handling

Simultaneous interrupts

- 2 stages cause an interrupt at the same time
- a solution: handle them in program order
- still precise

Precise exception

- Precise exception is difficult to implement for
  - Pipelined processors
    - Because exceptions can be generated out of order in different pipeline stages

Example of difficult cases

LW r4, X	**F**	**D**	**X**	**M***	**W**	
ADD r1, r2, r3		**F***	**D**	**X**	**M**	**W**

- **The Problem** The second instruction interrupts first!
  - If the second instruction is restarted first, and then the first instruction is restarted, then second instruction is executed twice!
- **A Solution** Let the hardware post interrupts for each instruction.
  - When instruction enters the W stage, check the interrupt flags, and handle the flags in instruction order.

How Pipelines Complicate Interrupt Handling

- **Interrupts out of order wrt sequential instruction execution**
  - subsequent instruction causes an interrupt before a previous instruction
  - interrupts still must be handled in program order for precise interrupts
- A **solution**: interrupt handled before the write stage
  - interrupt recorded in a per-instruction bit vector which flows with it down the pipeline
  - interrupts for instruction in write stage are handled before it changes any state
  - restart all instructions in the pipeline

How Pipelines Complicate Interrupt Handling

Multicycle operations in separate pipelines

- some types
  - floating point operations
  - integer multiply & divide
- can cause **imprecise interrupts** because operations don't necessarily complete in program-generated order
- example:

divf	**FP exception**
multf	**not done**
add	**completed**

  - cannot restart interrupting & subsequent instructions because some have completed
  - a completed instruction may have overwritten one its source operands

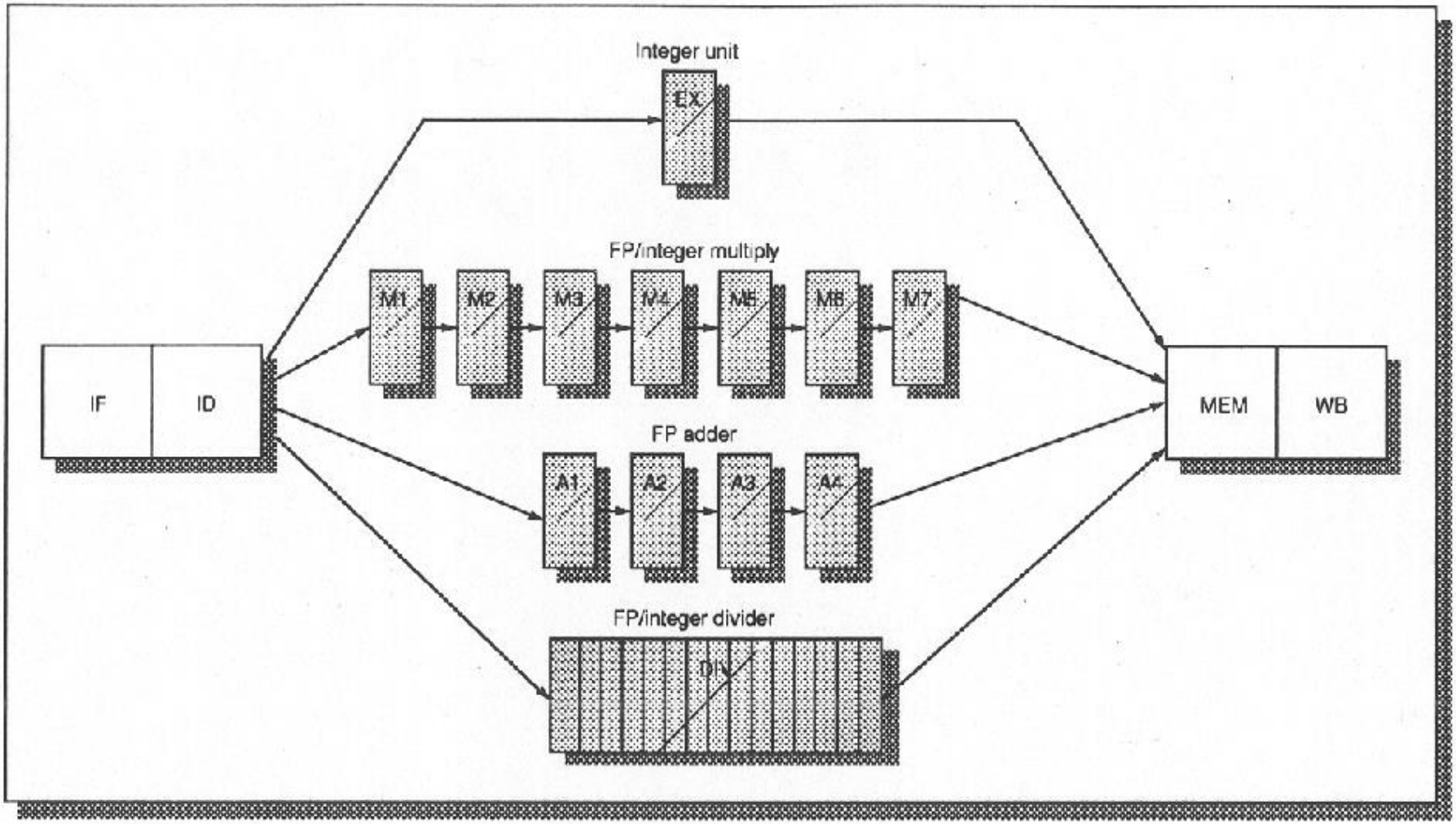


FIGURE 3.44 A pipeline that supports multiple outstanding FP operations.

How Pipelines Complicate Interrupt Handling

Dealing with **imprecise interrupts**

- ignore the imprecise interrupt (IBM 360)
- precise mode/imprecise mode (Alphas?)
  trade-off: correctness vs. performance
- buffer writes until instructions complete
  - future file or shadow register; today called **register renaming**
  - history file (VAX)
    costs: additional registers & bypass logic to FUs

Rename registers

- Rename registers form a pool of registers that can be temporarily used to store results until the instruction is "committed". These can be useful in implementing precise interrupts.
  - Instruction 1 (completed)
  - Instruction 2 (completed)
  - Instruction 3 (generates an interrupt)
  - Instruction 4 (completed, with result in rename register)
  - Instruction 5 (not executed yet)
- The result of instruction 4 will be written into a rename register first. It is written into the final destination (i.e committed or graduated) after all previous instruction have completed their executions.