Control Hazards

- Branches (conditional, unconditional, call-return)
- Interrupts: asynchronous event (e.g., I/O)
  - Occurrence of an interrupt checked at every cycle
  - If an interrupt has been raised, don’t fetch next instruction, flush the pipe, handle the interrupt (see later in the quarter)
- Exceptions (e.g., arithmetic overflow, page fault etc.)
  - Program and data dependent (repeatable), hence “synchronous”

Exceptions

- Occur “within” an instruction, for example:
  - During IF: page fault (see later)
  - During ID: illegal opcode
  - During EX: division by 0
  - During MEM: page fault; protection violation
- Handling exceptions
  - A pipeline is *restartable* if the exception can be handled and the program restarted w/o affecting execution

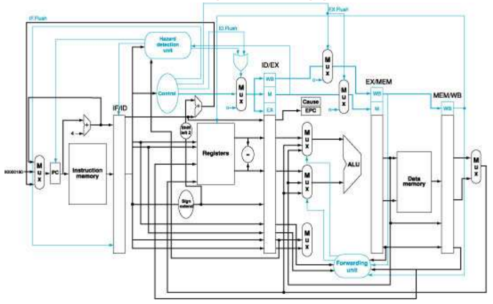
Precise exceptions

- If exception at instruction *i* then
  - Instructions *i-1, i-2 etc* complete normally (flush the pipe)
  - Instructions *i+1, i+2 etc.* already in the pipeline will be “no-oped” and will be restarted from scratch after the exception has been handled
- Handling precise exceptions: Basic idea
  - Force a trap instruction on the next IF (i.e., transfer of control to a known location in the O.S.)
  - Turn off writes for all instructions *i* and following
  - When the target of the trap instruction receives control, it saves the PC of the instruction having the exception
  - After the exception has been handled, an instruction “return from trap” will restore the PC.

Exception Handling

- When an exception occurs
  - Address (PC) of offending instruction saved in Exception Program Counter (a register not visible to ISA).
    - In MIPS should save PC – 4
  - Transfer control to OS
- OS handling of the exception. Two methods
  - Register the cause of the exception in a status register which is part of the state of the process
  - Transfer to a specific routine tailored for the cause of the exception; this is called vectored interrupts

Exception Support



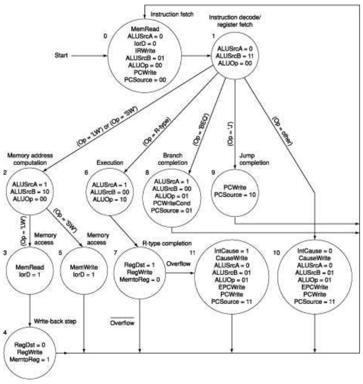
Exception Handling (continued)

- OS saves the state of the process (registers etc.)
- OS “clears” the exception
  - Can decide to abort the program
  - Can “correct” the exception
  - Can perform useful functions (e.g., I/O interrupt, syscall etc.)
- Return to the running process
  - Restores state
  - Restores PC

Precise exceptions (continued)

- Relatively simple for integer pipeline
  - All current machines have precise exceptions for integer and load-store operations
- Can lead to loss of performance for pipes with multiple cycles execution stage

Exception Control



Integer pipeline (RISC) precise exceptions

- Recall that exceptions can occur in all stages but WB
- Exceptions must be treated in *instruction order*
  - Instruction *i* starts at time *t*
  - Exception in MEM stage at time *t + 3* (treat it first)
  - Instruction *i* + 1 starts at time *t +1*
  - Exception in IF stage at time *t* + *1* (occurs earlier but treat in 2nd)

Treating exceptions in order

- Use pipeline registers
  - Status vector of possible exceptions carried along with the instruction.
  - Once an exception is posted, no writing (no change of state; easy in integer pipeline -- just prevent store in memory)
  - When an instruction leaves MEM stage, check for exception

Difficulties in less RISCy environments

- Due to instruction set ("long" instructions")
  - String instructions (but use of general registers to keep state)
  - Instructions that change state before last stage (e.g., autoincrement mode in Vax and *update addressing* in Power PC) and these changes are needed to complete inst. (require ability to back up)
- Condition codes (another way to handle branches)
  - Must remember when last changed