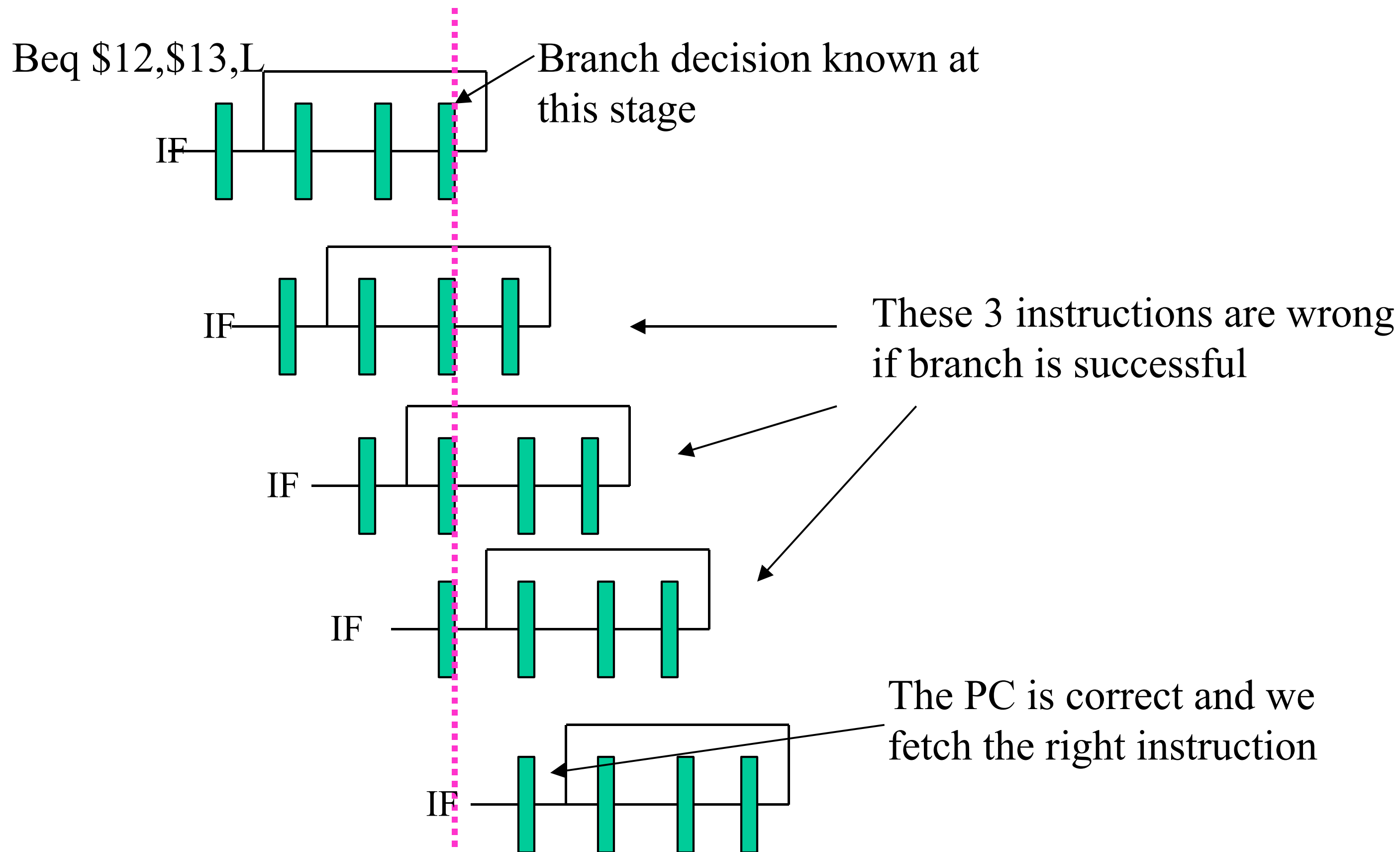
Control hazards

- Pipelining and branching don't get along
- Transfer of control (jumps, procedure call/returns, successful branches) cause control hazards
- When a branch is known to succeed, at the Mem stage (but could be done one stage earlier), there are instructions in the pipeline in stages before Mem that
  - need to be converted into "no-op"
  - and we need to start fetching the correct instructions by using the right PC

Example of control hazard



Resolving control hazards

- Detecting a potential control hazard is easy
  - Look at the opcode
- We must insure that the state of the program is not changed until the outcome of the branch is known. Possibilities are:
  - Stall as soon as opcode is detected (cost 3 bubbles; same type of logic as for the load stall but for 3 cycles instead of 1)
  - Assume that branch won't be taken (cost only if branch is taken; see next slides)
  - Use some predictive techniques

Assume branch not taken strategy

- We have a problem if branch is taken!
- "No-op" the "wrong" instructions
  - Once the new PC is known (in Mem stage)
    - Zero out the instruction field in the IF/ID pipeline register
    - For the instruction in the ID stage, use the signals that were set-up for data dependencies in the Load case
    - For the instruction in the EX stage, zero out the result of the ALU (e.g, make the result register be register $0)

Optimizations

- Move up the result of branch execution
  - Do target address computation in ID stage (like in multiple cycle implementation)
  - Comparing registers is "fast"; can be done in first phase of the clock and setting PC in the second phase.
  - Thus we can reduce stalling time by 1 bubble
- In the book, they reduce it by 2 bubbles but….
  - The organization as shown is slightly flawed (they forgot about extra complications in forwarding ….)

Branch prediction

- Instead of assuming "branch not taken" you can have a table keeping the history of past branches
  - We'll see how to build such tables when we study caches
  - History can be restricted to 2-bit "saturating counters" such that it takes two wrong prediction outcomes before changing your prediction
  - If predicted taken, will need only 1 bubble since PC can be computed during ID stage.
  - There even exists schemes where you can predict and not lose any cycle on predicted taken, of course if the prediction is correct
- Note that if prediction is incorrect, you need to flush the pipe as before

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 Handling (ct'd)

- 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 (cont'd)

- 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

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 on 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

Current trends in microprocessor design

- *Superscalar* processors
  - Several pipelines, e.g., integer pipeline(s), floating-point, load/store unit etc
  - Several instructions are fetched and decoded at once. They can be executed concurrently if there are no hazards
- *Out-of-order execution* (also called dynamically scheduled processors)
  - While some instructions are stalled because of dependencies or other causes (cache misses, see later), other instructions down he stream can still proceed.
  - However results must be stored in program order!

Current trends (ct'd)

- *Speculative execution*
  - Predict the outcome of branches and continue processing with (of course) a recovery mechanism.
  - Because branches occur so often, the branch prediction mechanisms have become very sophisticated
  - Assume that Load/Stores don't conflict (of course need to be able to recover)
- *VLIW (or EPIC)* (Very Long Instruction Word)
  - In "pure VLIW", each pipeline (functional unit) is assigned a task at every cycle. The compiler does it.
  - A little less ambitious: have compiler generate long instructions (e.g., using 3 pipes; cf. Intel IA-64 or Itanium)