Hazards

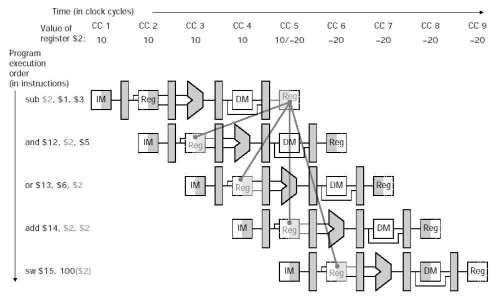
Introduction

- Recall that pipelining so far has been "ideal"
- In reality, we have to deal with these issues:
  - *Data hazards*:

```
add     $7, $12, $15
sub     $8, $7, $12
and     $9, $13, $7
```

  - We can show dependencies in diagrams like the one below. Arrows that go backwards in time are bad news!
  - *Control hazards*: if we take a branch, the instructions immediately following are the wrong ones to have fetched...

Data Hazards



Defining Dependencies/Hazards

- Dependencies: Given two instructions *m* and *n* (*m* occurs before *n*)
- A *dependence* exists between *m* and *n* if *n* reads the result produced by *m*, and there is no instruction *k* which occurs between *m* and *n* that produces the same result as *m*.
- We call the dependence a *hazard* when an instruction tries to read a value in the ID stage that will be written by a prior instruction that has not yet completed WB.
- Hazards are defined with respect to a particular implementation.
- Read-after-write hazards/dependencies.

Strategies for Resolving Hazards

- *Stalling*: detect the hazard, and then inject bubbles (nops) into the pipeline until the hazard clears. Not a great idea. If we stalled for 3 cycles to let each R-type instruction finish, our performance would be terrible...
- *Forwarding*: pass results to the ALU from different stages in the pipe.
- *Static scheduling*: make the compiler avoid generating code that gives rise to hazards. If it can't do so, it must insert nops.
- *Dynamic scheduling*: Build hardware that can reorder instructions (at run time) to avoid hazards. If it can't do so, it injects nops.

Detecting Hazards

- Between instruction i+1 and i (insert 3 bubbles):
  - ID/EX.WriteReg == IF/ID read-register 1 or 2
- Between instruction i+2 and i (insert 2 bubbles):
  - EX/MEM.WriteReg == IF/ID read register 1 or 2
- Between instruction i+3 and i (insert 1 bubble):
  - MEM/WB.WriteReg == IF/ID read register 1 or 2

Stalling

- Once we have detected a hazard, we can stall the pipeline.
- Stalls stop instructions in the ID stage. We must stop fetching new instructions (to avoid clobbering PC and IF/ID register). Control lines:
  - Create bubbles. Done by setting all control lines from ID to zero. Creating a nop.
  - Prevent new instruction fetches (or PC updates).
- Our scheme is conservative:
  - Is the RegWrite control bit asserted (if not, we're off the hook).
  - Build a better Register file. If a register is both read and written on the same cycle, we currently get the *old* value. We can improve the register file to provide the right data in this case. This takes care of the case between WB and Decode.

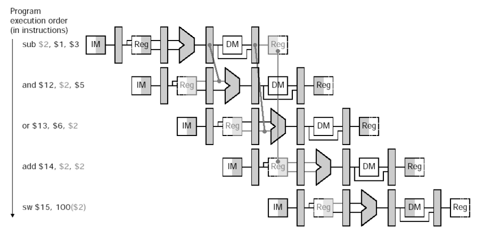
Forwarding

- Stalling is pessimistic, because we often do have the right values available (they just haven't been written yet).
  - At the end of the EX stage for R-type instructions
  - At the end of the MEM stage for Load instructions.
- Why not forward the result of the computation (or load) directly to the input of the ALU?
- (Note, we still want to force the writeback to occur in the final stage, due to issues related to exceptions.)

Forwarding Example

Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM :	X	X	X	–20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	–20	X	X	X	X

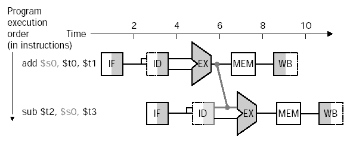


Implementing Forwarding

- Change datapath so that data can be read either from the EX/MEM or MEM/WB registers and be forwarded to one of the ALU inputs.
- Hazard logic for EX stage:
  - if EX/MEM.RegWrite AND EX/MEM.DestReg == ID/EX.rs
  - if EX/MEM.RegWrite AND EX/MEM.DestReg == ID.EX.rt
- Hazard logic for MEM stage:
  - if MEM/WB.RegWrite AND EX/MEM.DestReg != ID/EX.rs AND MEM/WB.DestReg == ID/EX.rs
  - if MEM/WB.RegWrite AND EX/MEM.DestReg != ID/EX.rt AND MEM/WB.DestReg == ID/EX.rt

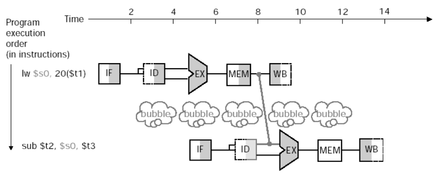
The Trouble with Loads

- Forwarding can't save the day with loads (if they're followed immediately by a dependent R-type instruction).

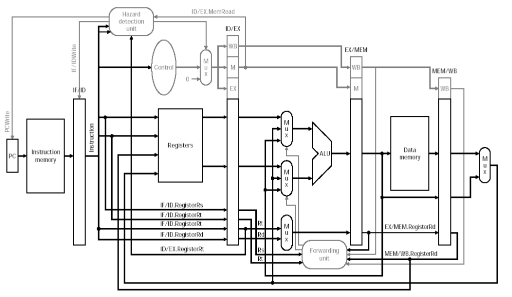


Loads

- Must insert a bubble after the load in this case, so we still need a hazard detection unit.
- Good compilers will try to schedule non-dependent instructions after the load...



Datapath with Forwarding & Hazard Detection



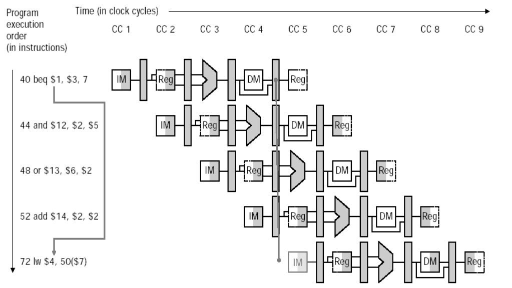
5/21/2002 166

Control Hazards

- The transfer of control, via jumps and branches causes this class of hazard.
- The branch instruction decides in the EX stage if the branch will be taken. It doesn't update the PC until the end of the MEM stage.
- Hence, we will have erroneously fetched 3 instructions (if the branch is taken).

5/21/2002 167

Control Hazards in Pictures



5/21/2002 168

Resolving the Hazard

- Detecting it is easy: just look at the opcode!
- At least 4 strategies:
  - *Always stall*: Stall as soon as we see a branch. This costs a tiny bit of control hardware and 3 cycles for each branch!
  - *Assume branch not taken*: Just go ahead and start executing the subsequent instructions. If the branch is taken, then flush the improper instructions. Costs more hardware, but 3 cycles only if the branch is taken.
  - *Delayed branches*: Change the semantics of the branch instruction -- force the compiler/assembler to deal with the problem.
  - *Branch prediction*: Try to guess. Still have to be prepared to flush the pipeline if you were wrong!

5/21/2002 169

Assume Not Taken

- Need to flush instructions from the pipe if the branch is eventually taken.
- First, move all branch activity to the EX stage. This means we'll have to flush two instructions (the one in ID and the one just fetched by IF).
  - For IF: we zero out the instruction field in the IF/ID register.
  - For ID: just set all control lines to zero, creating a NOP.
- Rule of thumb:
  - Forward branches are taken 60% of the time.
  - Backward branches (loops) are taken 90% of the time.
- Performance?

5/21/2002 170

Delayed Branches

- Change the meaning of branches: they don't have effect until N cycles later. (N is the *branch delay*.)
- The N instructions after the branch will be executed regardless of the branch outcome.
- Costs zero hardware! (The compiler/assembler must insert nops after the branch if they can't put other instructions there.)
- Good compilers can fill 1 or 2 slots, but more is hard.
- (Real) MIPS branches are delayed by one cycle. Compilers fill 70% of the delay slots.
- Performance?

5/21/2002 171

Prediction

- Build a history table and use it to make a guess.
- If you're wrong, you still have to flush the pipeline.
- Note that you still can't get rid of the delay entirely. Why? Because it will take you at least a cycle to look up the branch target. The earliest you can get the new target is in the ID stage, so you still may have to flush an instruction...
- Predict-not-taken is just a special case of branch prediction
- Even pretty simple branch prediction units are correct 90% of the time.
- Performance compared to other schemes?

Exceptions

- Historical definitions:
  - *Exceptions*: an unexpected event from within the processor (divide-by-zero)
  - *Interrupt*: an unexpected event from outside the processor (such as I/O request)
- MIPS calls them all exceptions.
- Kinds:
  - I/O device request (external)
  - System call (internal)
  - Undefined instruction (internal)
  - Hardware malfunction (either)
  - Breakpoint (internal)

Handling Exceptions

- Save the program counter of the offending instruction: EPC.
- Set up the cause register (what kind of exception)
- Transfer control to the OS (to a fixed address).
- The OS then takes appropriate action:
  - Illegal instruction: kill the program
  - I/O device: handle the request
  - Timer: switch to another program
  - Breakpoint: switch to the debugger, etc.
- Suppose we get a divide-by-zero in EX. We need to be sure to let the downstream instructions complete, while flushing the upstream instructions.