Hazards

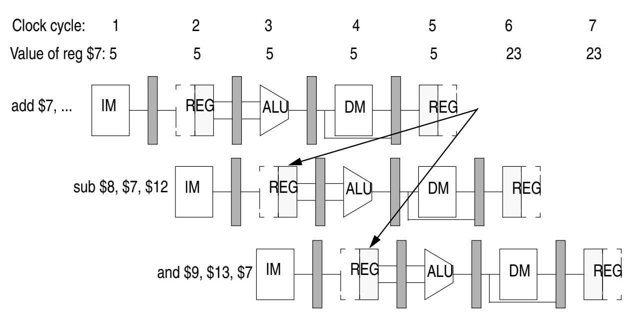
Introduction

- Pipelining up until now has been “ideal”
- In real life, though, we might not be able to fill the pipeline because of hazards:
  - Data hazards. For example, the result of an operation is needed before it is computed:

```
add    $7, $12, $15      # put result in $7
sub    $8, $7, $12       # use $7
and    $9, $13, $7       # use $7 again
```

  - Note that there is no dependency for $12, b/c it is used only as a source register.
  - Control hazards. If we take the branch, then the instructions were fetched after the branch (which are now in the pipe) are the wrong ones.

Data Hazards



- The arrow represents a dependency. Arrows that go backwards are trouble.

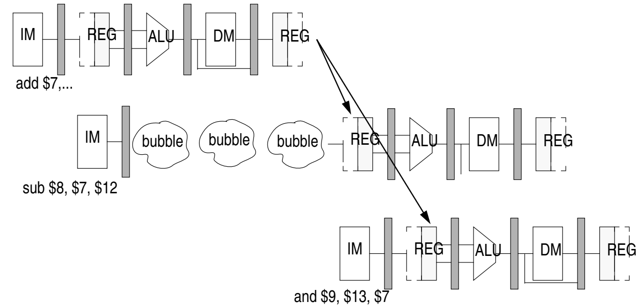
Detecting Data Dependencies

- Dependencies: Given two instructions, *i* and *j* (*i* occurs before *j*).
- We say a *dependence* exists between *i* and *j* if *j* reads the result produced by *i*, and there is no instruction *k* which occurs between *i* and *j* and that produces the same result as *i*.
- We call a data dependence a *hazard* when an instruction tries to read a register in stage 2 (ID) and this register will be written by a previous instruction that has not yet completed stage 5 (WB).
- This is sometimes called a read-after-write hazard.
- What kind of instructions can create data dependences?
- Modern microprocessors have several ALUs, floating point units that take longer than integer units, etc which give rise to other kinds of data hazards.

Resolving Data Hazards

- There are several options:
  - Build a hazard detection unit, which stalls the pipeline until the hazard has passed. It does this by inserting "bubbles" (essentially nops) in the pipeline. This costs hardware.
  - *Forwarding.* Forward the result as an ALU source.
  - *Software (static) scheduling.* Leave it up to the compiler. It must schedule instructions to avoid hazards. Often it won't be able to, so it will issue no-ops (an instruction that does nothing) instead. This is the cheapest (in terms of hardware) solution.
  - *Hardware (dynamic) scheduling.* Build special hardware that schedules instructions dynamiclly.

Hazard Detection and Stalling



- Note that the hazard costs us 3 cycles...

Detecting Hazards

- Between instruction i+1 and instruction i (3 bubbles):
  - ID/EX.WriteReg == IF/ID read-register 1 or 2 (in fact, it is slightly more complex b/c write-register can be rd or rt depending on the instruction)
- Between instruction i+2 and i (2 bubbles):
  - EX/MEM.WriteReg == IF/ID read-register 1 or 2
- Between instruction i+3 and i (1 bubble):
  - MEM/WB.WriteReg == IF/ID read-register 1 or 2
- Note that stalls stop instructions in the ID stage. Therefore, we must stop fetching new instructions, or else we would clobber the PC and the IF/ID register. So we need control lines to:
  - Create bubbles. This can be done by setting all control lines that are passed from ID to 0, hence creating a nop.
  - Prevent new instruction fetches. This should be done for as many cycles as there are bubbles.

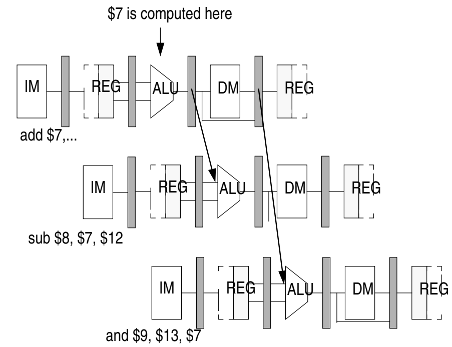
Improvements

- Our stalling scheme is very conservative, and there are a few improvements we can make.
- Is the RegWrite control bit asserted (this determines whether we're really dealing with an R-type or load instruction)?
- Build a better register file. Currently, we assume that the register file will not produce the correct result if a given register is both read and written in the same cycle. Doing this would eliminate hazards in the WB stage.

Forwarding

- Inserting bubbles is a pessimistic solution, since data that is written during the writeback stage is often computed much earlier:
  - At the end of the EX stage for arithmetic instructions
  - At the end of the MEM stage for a load.
- So why not *forward* the result of the computation (or load) directly to the input of the ALU if it is required there?
- Forwarding is sometimes called *bypassing*.
- Note that for reasons related to *interrupts* or *exceptions*, we do not want the *state* of the process (i.e. the registers), to be modified until the last stage.

Forwarding Example



- There is no need to wait until WB, because we've already computed the value required.

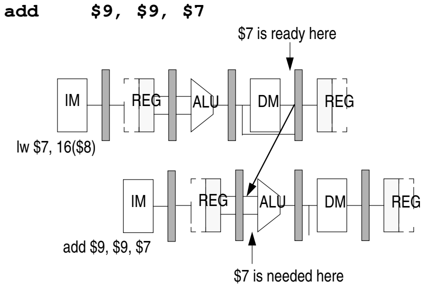
Implementing Forwarding

- Change the data path so that data can be read from either the EX/MEM or MEM/WB registers and be forwarded to one of the ALU inputs.
- This requires logic to detect forwarding:
  - We can do this at stage 3 (EX) of instruction i to forward to stage 2 (ID) of instruction i+1
  - We can do this at stage 4 (MEM) of instruction i to forward to stage 2 (ID) of instruction i+2.
- It also requires additional inputs to the muxes over the ALU inputs (inputs can now come from ID/EX, EX/MEM, or MEM/WB pipe registers).

The Trouble with Loads

- What if we have a load followed by an arithmetic operation which needs the result of the load:

```
lw      $7, 16($8)
add     $9, $9, $7
```



- We're busy fetching the data while it is needed in the EX stage.

Loads

- Forwarding cannot save the day in the face of a dependent instruction which immediately follows a load.
- The only solution is to insert a bubble after loads if the next operation is dependent, so we still need a hazard detection unit.
- Good compilers will attempt to schedule instructions in the “load delay slot” so as to avoid these kinds of stalls.

Scheduling

- Other important approaches include scheduling the instructions to avoid hazards, in hardware or software.
- This is particularly important for processors which have multiple or very deep pipelines (most modern processors).
- Dependences force a partial ordering on the instruction stream.

```
lw      $t2, 0($t0)       # 1
add     $t5, $t2, $t3     # 2
sub     $t3, $t1, $t8     # 3
mult    $t7, $t8, $t8     # 4
addi    $t5, $t7, 16      # 5
```

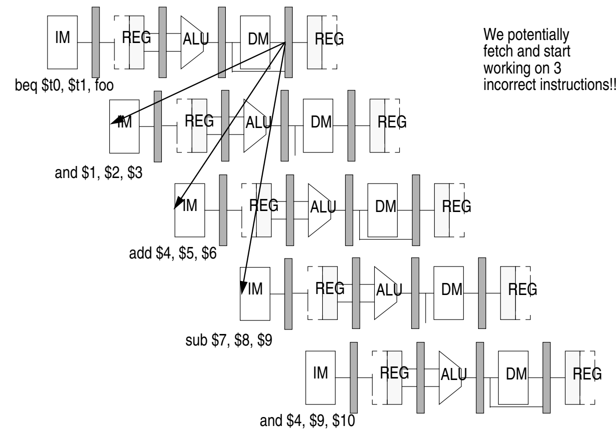
- Three kinds of dependence: data (read-after-write), anti-dependence (write-after-read), output (write-after-write).
- Above: data dependences (1->2); anti-depencences (2->3, 4->5); output (2->5).
- How can we reorder these instructions to do better?

Control Hazards

- Pipelining and branching just don’t get along...
- The transfer of control, via jumps, returns, and taken branches cause control hazards.
- The branch instruction decides whether to branch in the MEM stage. In other words, if the branch is taken, the PC isn’t updated to the proper address until the end of the MEM stage.
- By this time, however, we’ve already entered 3 instructions into the pipeline that were the *wrong* ones!

```
        beq     $t0, $t1, foo  # assume $t0==$t1
        and     $1, $2, $3
        add     $4, $5, $6
        sub     $7, $8, $9
foo:    add     $4, $9, $10
```


Example



Resolving Control Hazards

- Detecting one is easy: just look at the opcode!
- At least 4 possibilities:
  - *Always stall*. Stall as soon as we see a branch. This costs a bit of control hardware and 3 cycles for every branch.
  - *Assume branch not taken*. Just go ahead and start executing the next instructions, but find a way to *flush* those instructions if the branch was taken. This costs more control hardware and 3 cycles *only* if the branch is taken.
  - *Delayed branches*. Change the semantics of your branch instruction to force the compiler/assembler to deal with the problem.
  - *Branch prediction*. Try to guess whether the branch will be taken or not and do the right thing. Be ready to flush the pipeline in case you were wrong...

Assume Branch Not Taken

- We need to be able to flush the pipeline in case the branch actually was taken.
- If the branch is taken:
  - For the IF stage, we zero out the instruction field in IF/ID register.
  - For the ID stage, since this is where we determine control, we just set all control lines to zero, creating the effect of a nop.
  - For the EX stage, we use an extra mux to zero out the result of the ALU.
- This approach costs additional control hardware and costs cycles only when the branch is taken.
- A rule of thumb says that forward branches are taken 60% of the time, and backward branches (as in loops) are taken 85% of the time.

Delayed Branches

- Change the semantics (meaning) of your branch instruction so that they won't have effect until N (where N is the branch delay) cycles later.
- This means that the N instructions after the branch will be executed *regardless* of the branch outcome. These are called *delay slots*.
- This forces the programmer/compiler/assembler to deal with the problem, by requiring them to fill the N delay slots.
- This costs the hardware nothing, since it is the compilers job to assure that correct instructions (or nops) are scheduled in the delay slots.
- Good compilers can usually fill 1 or 2 slots.
- MIPS branches are delayed (1 slot) and compilers can fill around 70% of the slots.

Branch Prediction

- Develop some hardware to tell you the chances that you will actually take the branch or not (a history table, for example).
- Given this information, make a prediction (taken or untaken) and start executing instructins speculatively.
- If you're wrong, you still have to flush the pipeline.
- Note that assuming branch-not-taken is just a special case of branch prediction (where you always predict not taken).
- Branch prediction should do better than assuming not taken, but you pay the price in additional hardware.
- Branch prediction should do better than delayed branches, assuming you predict right more often than the compiler can fill the delay slot with interesting work (not a nop).

Exceptions

- Historical definitions:
  - An *exception* is an unexpected event from within the processor (such as arithmetic overflow).
  - An *interrupt* is an unexpected event from outside of the processor (such as IO requests).
- MIPS doesn't distinguish the source of the event, and calls both of the above *exceptions*.
- Kinds:
  - IO device request (external)
  - System call (internal)
  - Arithmetic overflow (internal)
  - Undefined instruction (internal)
  - Hardware malfunctions (either)
- Note that we can view system calls as exceptions!

How to Handle Exceptions

- We must save the program counter of the offending instruction in the EPC (Exception PC), and then transfer control to the operating system.
- The OS can then take appropriate action (provide an IO service for the program, kill the program, etc). If it chooses to restart the program, it can jump back to the EPC.
- How does the OS know what kind of exception? MIPS includes a *cause* register.
- In hardware, the cause is saved into the cause register, the PC is saved in EPC, and control transfers to a predefined address in the kernel (0x4000 0040).
- Exceptions are hard to deal with because we have several instructions in the pipeline.
- Suppose we get an arithmetic overflow (in the EX stage). We need to be sure to let the downstream instructions finish, while flushing the upstream instructions.

The Truth

- The MIPS R2000/3000 pipelined implementation is pretty close to the one we've discussed in class, but modern machines use much more complex implementations:
- Multiple pipelines: *superscalar*:
  - Trend: exploit instruction level parallelism (ILP) by working on multiple instructions simultaneously. This reduces CPI.
  - Many modern machines issue up to 4 instructions at once.
  - Challenge: statically or dynamically scheduling instructions to extract maximal ILP while keeping cycle time low
- Deep pipelines: *superpipelined*:
  - Trend: Reduce cycle time
  - Modern pipelines often have 8 or more stages.
  - Challenge: longer branch and load delays (often leading to higher CPI), more forwarding required, scheduling is also important

Summary

- Pipelining improves performance by increasing throughput (instructions/time) not latency (time/instruction).
- We examined the classic 5 stage pipeline (IF, ID, EX, MEM, WB)
- Data and control hazards place limits on the speedups we can achieve through pipelining.
  - Data hazards can be avoided by stalling or forwarding (unless it is a load!). Forwarding is more efficient.
  - The performance of branches can be improved through delayed branches or branch prediction.
- Compilers must understand the pipeline to extract maximum performance through scheduling. The ISA is no longer a perfect abstraction.