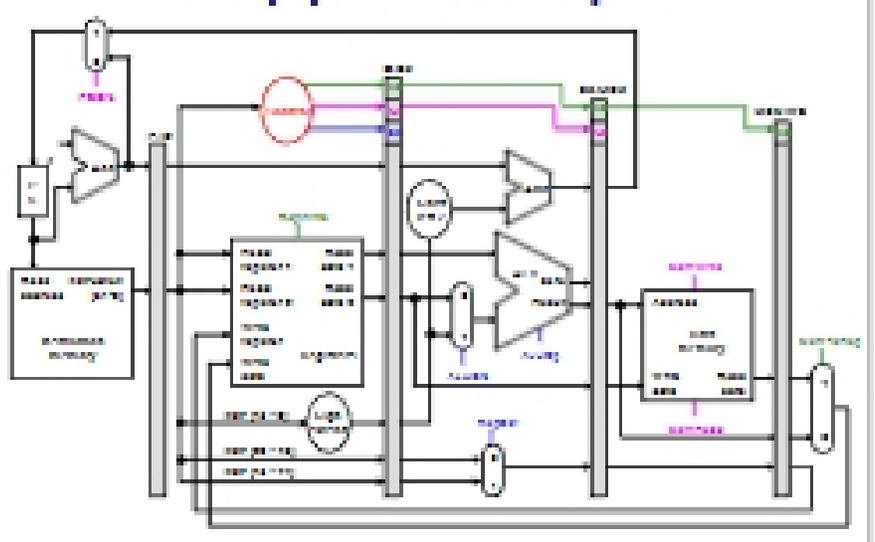
Lecture 13: Pipeline Hazards

(CPEG323: Intro. to Computer System Engineering)

Review

- Pipelining is a BIG idea
- Optimal Pipeline
  - Each instruction requires five stages, and five cycles, to complete.
  - Each stage uses different functional units of the datapath.
  - So we can execute up to five instructions in any clock cycle, with each instruction in a different stage and using different hardware.
- What makes this work well?
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.

The pipelined datapath

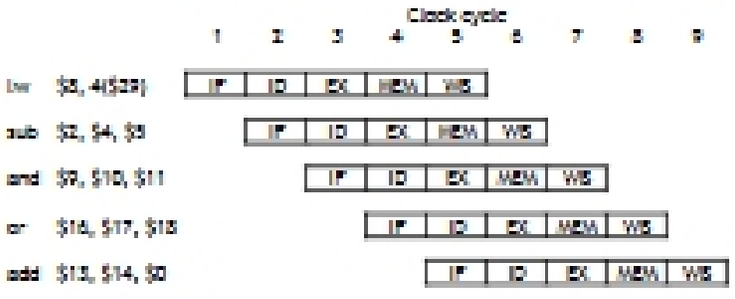


Problems for Pipelining CPUs

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Structural hazards: HW cannot support some combination of instructions
  - Control hazards: Pipelining of branches causes later instruction fetches to wait for the result of the branch
- These might result in pipeline stalls or "bubbles" in the pipeline.

Data Hazards

Pipeline diagram review



- This diagram shows the execution of an ideal code fragment.
  - Each instruction needs a total of five cycles for execution.
  - One instruction begins on every clock cycle for the first five cycles.
  - One instruction completes on each cycle from that time on.

Our examples are too simple

- Here is the example instruction sequence used to illustrate pipelining on the previous page.

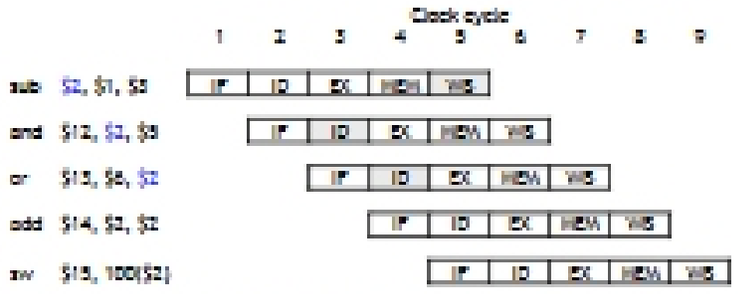
```
lw  $8, 4($29)
sub $2, $4, $5
and $9, $10, $11
or  $16, $17, $18
add $13, $14, $0
```

- The instructions in this example are independent.
  - Each instruction reads and writes completely different registers.
  - Our datapath handles this sequence easily, as we saw last time.
- But most sequences of instructions are *not* independent!

An example with dependencies

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

Data hazards in the pipeline diagram

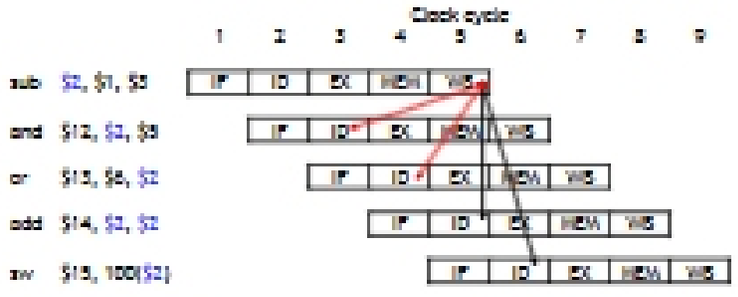


- The SUB instruction does not write to register $2 until clock cycle 5. This causes two data hazards in our current pipelined datapath.
  - The AND needs register $2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of $2, not the new one.
  - Similarly, the OR instruction uses register $2 in cycle 4, again before it's actually updated by SUB.

Things that are okay

- The ADD instruction is okay, because of the register file design.
  - Registers are written at the beginning of a clock cycle.
  - The new value will be available by the end of that cycle.
- The SW is no problem at all, since it reads $2 after the SUB finishes.

Dependency arrows



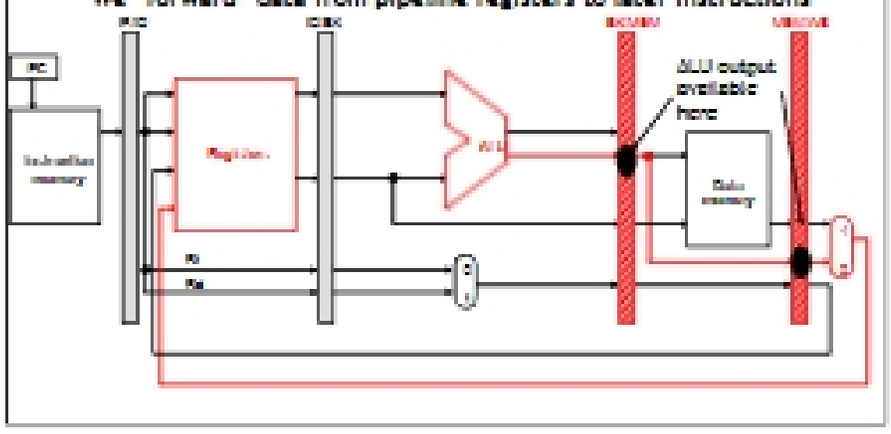
- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register $2 is written.
  - The heads of the arrows show when $2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath.

Bypassing the register file

- The actual result $1 – $3 is computed in clock cycle 3, *before* it's needed in cycles 4 and 5.
- If we could somehow bypass the writeback and register read stages when needed, then we can eliminate these data hazards.
- Essentially, we need to pass the ALU output from SUB directly to the AND and OR instructions, without going through the register file.

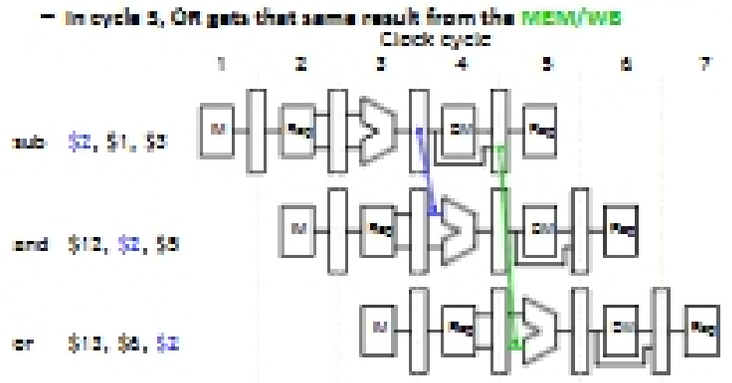
Pipeline Registers to the rescue!

- Pipeline stages communicate through pipeline registers:
  IF/ID ID/EX EX/MEM MEM/WB
- We "forward" data from pipeline registers to later instructions



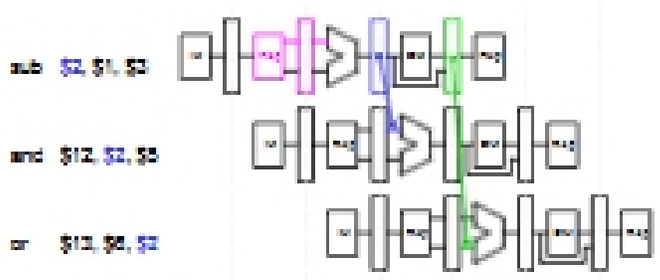
Forwarding

- The actual result $1 - $3 is computed in clock cycle 3, before it is needed in cycles 4 and 5
- We forward that value to later instructions, to prevent data hazards.
  - In clock cycle 4, AND gets the value $1 – $3 from the EX/MEM
  - In cycle 5, OR gets that same result from the MEM/WB

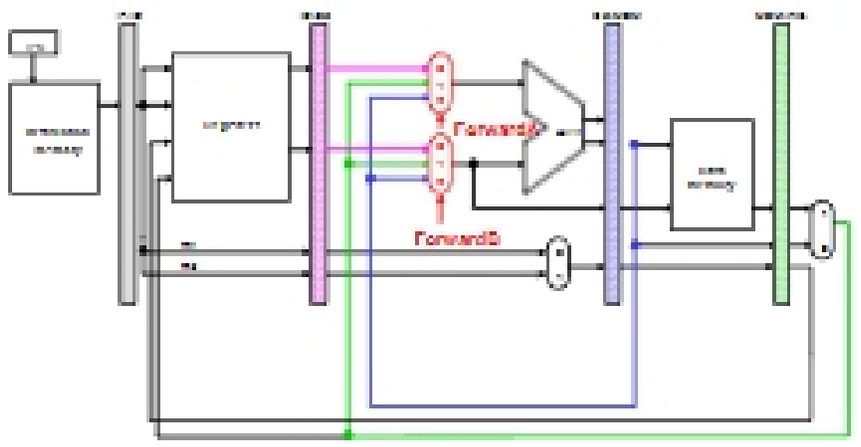


Outline of forwarding hardware

- A forwarding unit selects the correct ALU inputs for the EX stage.
  - No hazard: ALU's operands come from the register file, like normal.
  - Data hazard: operands come from either the EX/MEM or MEM/WB pipeline registers instead.
- The ALU sources will be selected by two new multiplexers, with control signals named ForwardA and ForwardB.

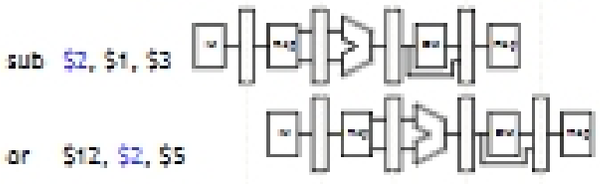


Simplified datapath with forwarding muxes



Detecting EX/MEM Hazards

- In which stage cycle can we detect an impending hazard?



- Answer: cycle 3, when sub is in EX, or is in ID
  - Hazard because: ID/EX.rd == IF/ID.rs
- An EX/MEM hazard occurs between the instruction currently in its EX stage and the previous instruction if:
  1. The previous instruction will write to the register file, *and*
  2. The destination is one of the ALU source registers in the EX stage

17

EX/MEM data hazard equations

- The first ALU source comes from the pipeline register when necessary.

  if (EX/MEM.RegWrite = 1
  and EX/MEM.RegisterRd = ID/EX.RegisterRs)
  then ForwardA = 2

- The second ALU source is similar.

  if (EX/MEM.RegWrite = 1
  and EX/MEM.RegisterRd = ID/EX.RegisterRt)
  then ForwardB = 2