CS M151B: Computer Systems Architecture

Week 5

Single Cycle Datapath... again...

- Ugh...

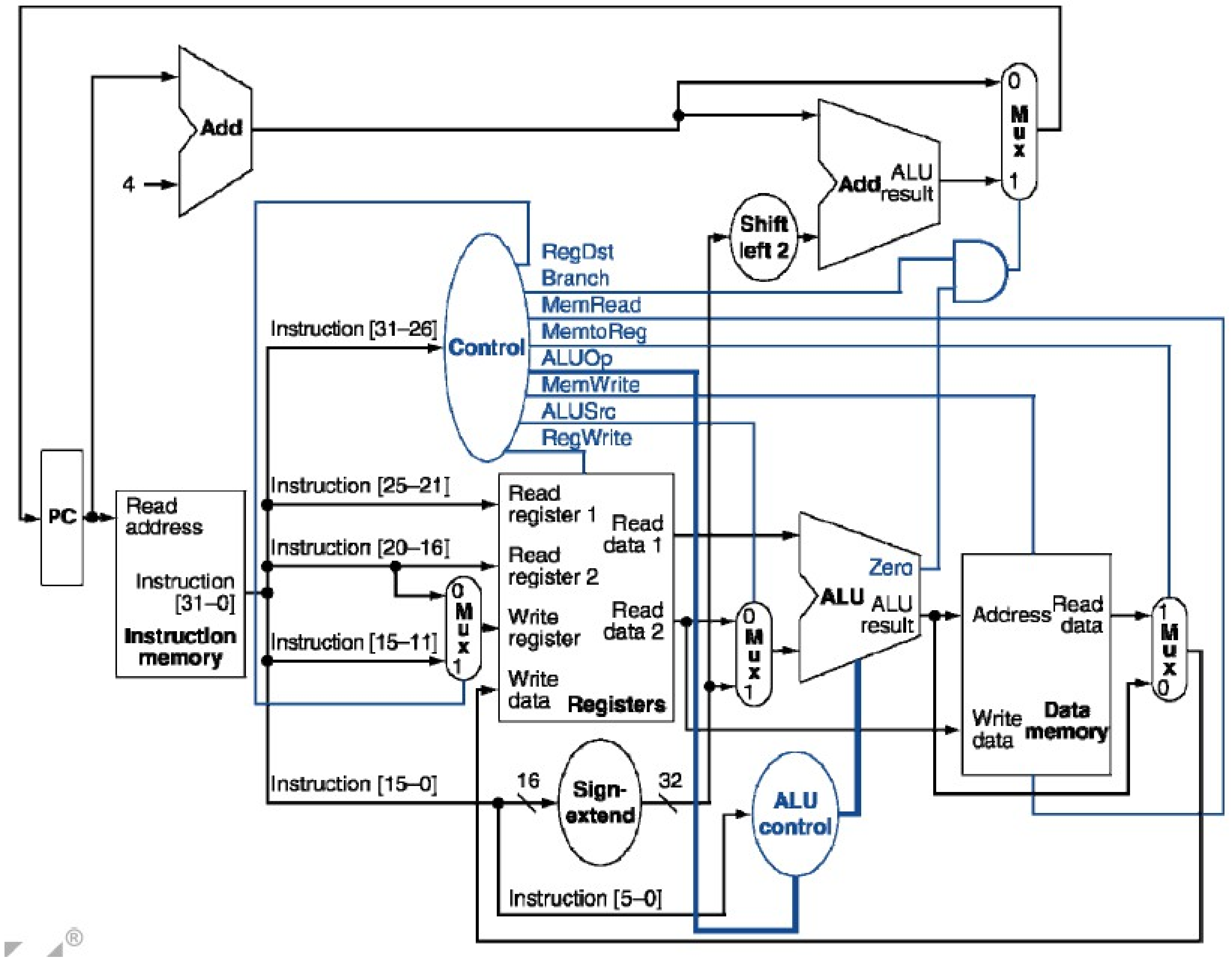
Ex: Extending Datapath for bne

- bne (branch if not equal to)

  ```
  if (R[rs] != R[rt])
    PC = PC + 4 + SE(I)
  else
    PC = PC + 4
  ```

- Definitely an I-Type instruction.
- What connections must be made?

bne



Add
4
Add
ALU result
Shift left 2
0 Mux 1
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
PC
Read address
Instruction [31–0]
Instruction memory
0 Mux 1
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
0 Mux 1
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
1 Mux 0
16
Sign-extend
32
ALU control

Ex: Extending Datapath for bne

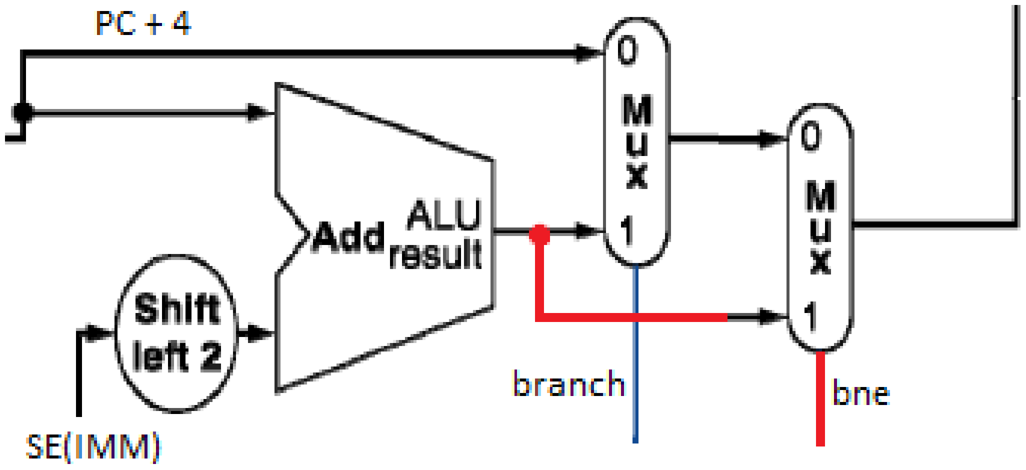
- Unlike some other examples, we don't really need many new connections.
- In fact, beq already defines hardware for choosing for PC = PC + 4 or PC = PC + 4 + SE(IMM)<<2. We can reuse it.
- Like with beq, we can subtract R[rs] and R[rt] and use the Zero signal. However, we'll use ~Zero which will be 1 if they don't equal.

Ex: Extending Datapath for bne

- Like with beq, we also need a bne control signal so that the processor doesn't branch if the Zero flag is coincidentally Zero as a result of an ALU operation.
- Like with beq, we can AND the bne signal and ~Zero to get a signal that will be 1 if the branch should be taken.
- How do choose between this new branch or the previous PC signals?

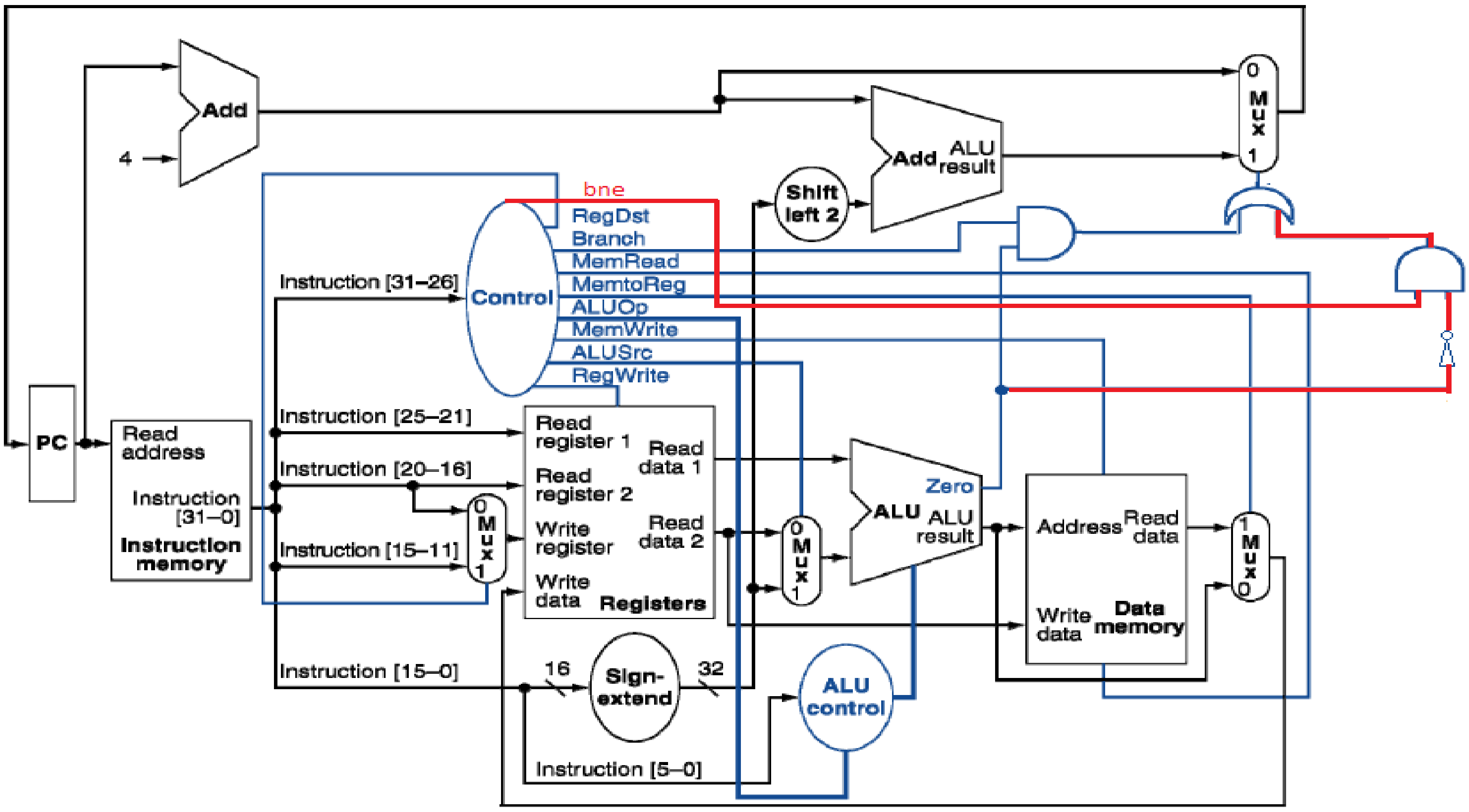
Ex: Extending Datapath for bne

- Our standard solution would be to use a multiplexer. But that will look like this:



- ...and while it works and would be worth full points, it's a little weird.
- Is there another way we can simply indicate to the original multiplexer to be 1?

Ex: Extending Datapath for bne



- Control signals?

Ex: Extending Datapath for bne

- Control signals?
- RegDst: X (not writing to register)
- ALUSrc: 0 (comparing R[Rs] and R[Rt])
- MemtoReg: X (not writing to register)
- RegWrite: 0 (don’t write to register)
- MemRead: 0 (don’t read from memory)
- MemWrite: 0 (don’t write to memory)
- Branch: 0 (don’t branch on equal)
- ALUOp1: 0
- ALUOp2: 1 (Have ALU subtract)
- bne: 1

Ex: Extending Datapath for bne

- Tradeoffs of doing the multiplexer vs. the OR style?

Ex: Extending Datapath for bne

- Tradeoffs of doing the multiplexer vs. the OR style?
  - Multiplexer requires more complicated hardware, but often means that you could leave control signals for multiplexers that come before it as DON'T CARE because they're overridden. In this case however, because the bne is conditional, branch signal must be zero.
  - OR requires simpler hardware, but the branch signal must be 0 rather than “don't care” (otherwise if branch was 1 during a bne, you'd always branch).
  - This is probably trivial enough where it doesn't matter.

The Single Cycle Datapath Strengths

- What is good about this design?

The Single Cycle Datapath Strengths

- What is good about this design?
  - It always takes one cycle to complete an instruction. CPI must then always be 1.
  - Simple to understand relative to other options.
  - Simple to extend.
  - Good stepping stone to understand more complex datapaths.
- What's bad about this design?

The Single Cycle Datapath Weaknesses

- Poor cycle time – If each instruction is expected to complete in one cycle that means the clock time must be long enough to encapsulate instructions with the greatest delay.
- Not so bad in this limited ISA, but not practical when we have to deal with floating operations, division, and etc.

The Single Cycle Datapath Weaknesses

- Recall the Eight Great Ideas in Computer Architecture.
- Does this single cycle datapath inhibit/violate one of these ideas?

The Single Cycle Datapath Weaknesses

- Make the Common Case Fast
- If we can make add or loads take 10 ps, it won't make a difference if the clock time is still 200 ps to account for an expensive operation.

The Single Cycle Datapath Weaknesses

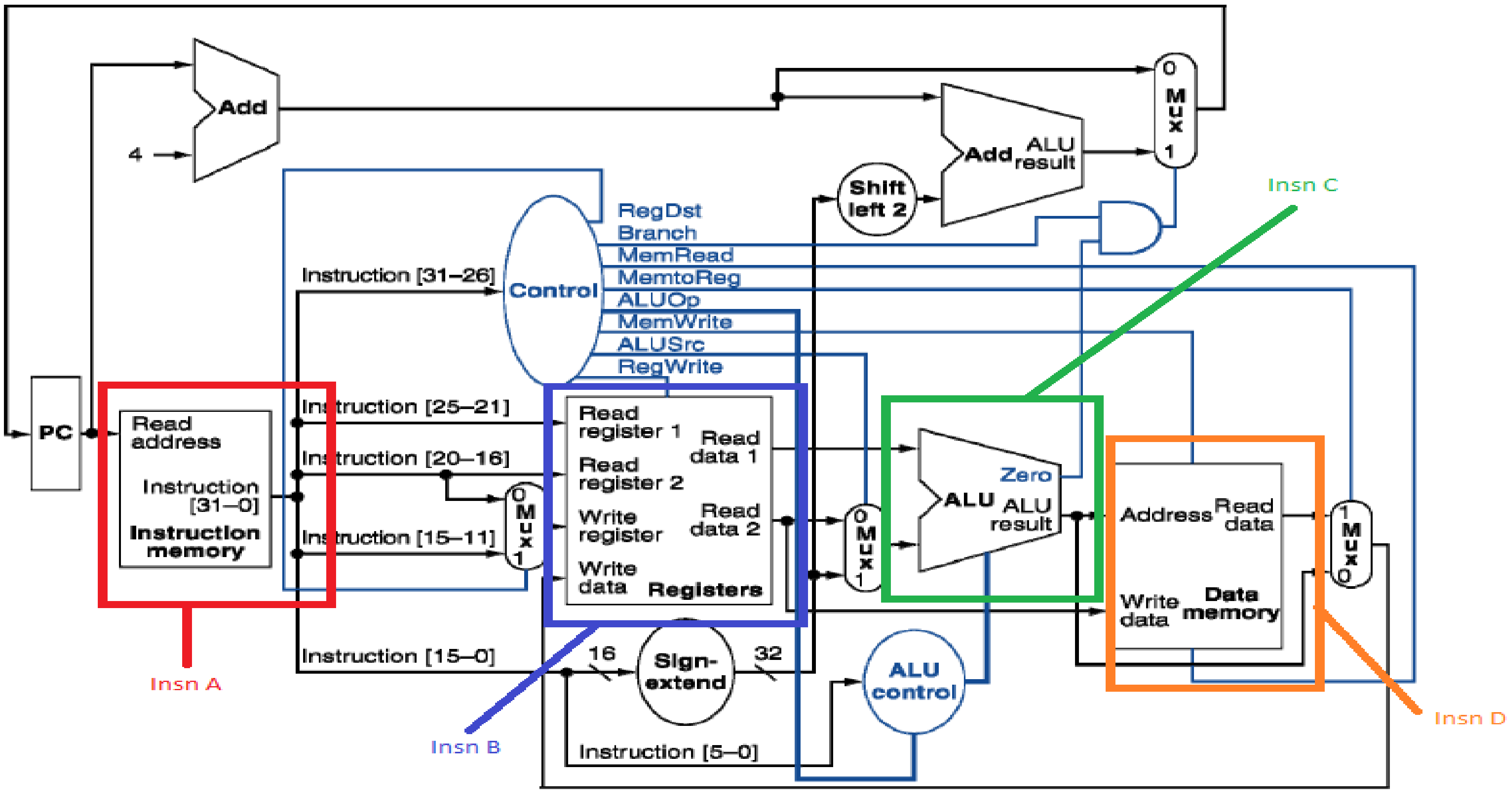
- From the issuing of the instruction, the data goes through the data path sequentially to have a completed result.
- At any one time, only one module is being used.
- This is poor utilization!
- Since we've mastered the single cycle datapath, let's try a different approach.

Alternative

- Let's try an approach where instead of using just one component of datapath, we use different components for different instructions.
- For example, we want something like the following:

Alternative

- We want simultaneous usage of the datapath by different instructions.



Alternative

- To do this, we're forced to split the single datapath into smaller modules that each take a cycle to complete.
- Why?

Alternative

- To do this, we're forced to split the single datapath into smaller modules that each take a cycle to complete.
- Why?
  - If data is simply allowed to flow through the datapath without synchronization, we wouldn't be able to guarantee a stable result at each clock cycle for multiple instructions.

Pipelined Datapath

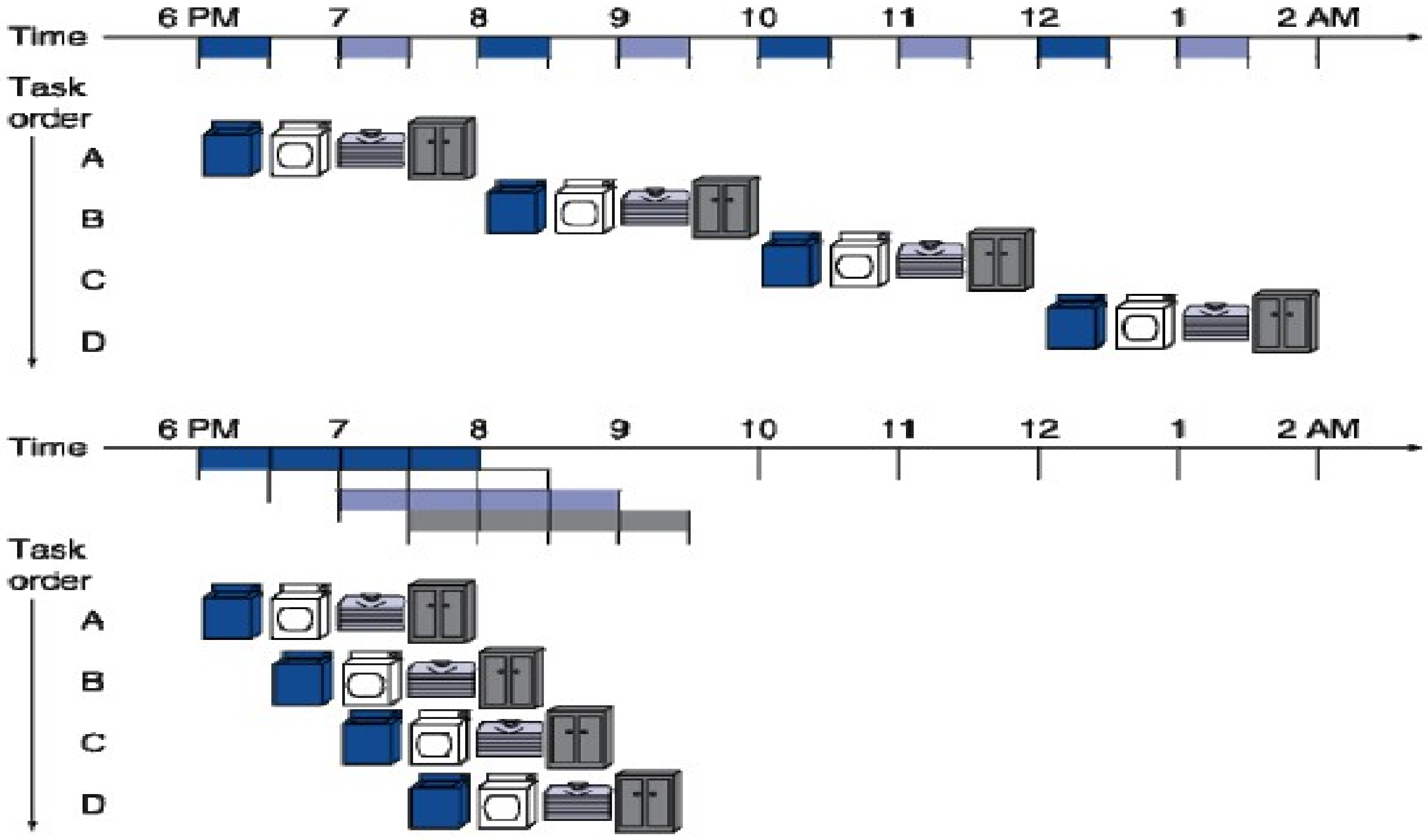
- This means now a single clock cycle only needs to match the latency of the slowest of the sub-components (instruction memory, register file, ALU, data memory).
- The cycle time goes down. Bonus!

Pipeline Principle

- Fundamentally, we want to have a multi-stage pipeline.
- Each instruction goes in one end and passes through each stage sequentially until reaching the end.
- Each stage can be used for different tasks concurrently.
- Hey, that sounds like...

Pipeline Principle

- Yup, it's this picture again.



Pipeline Principle

- Latency: The time it takes a single instruction to complete.
  - The pipelining provides the same (or worse) latency.
- Throughput: The rate at which data is received or tasks are completed. In our instance instructions completed per second or instructions completed per cycle.
  - Pipeline is initially worse in CPI, but approaches the CPI of the non-pipelined cased.

Pipeline Principle

- Assume four pipeline stages and each stage takes a cycle to completed.
- Consider n->infinity instructions issued in sequence.
- Non-pipelined CPI:
  - n cycles / n instructions = 1
- Pipelined CPI:
  - (3 + n) / n = 3/n + n/n = 1 if n->infinity
  - Incredible!

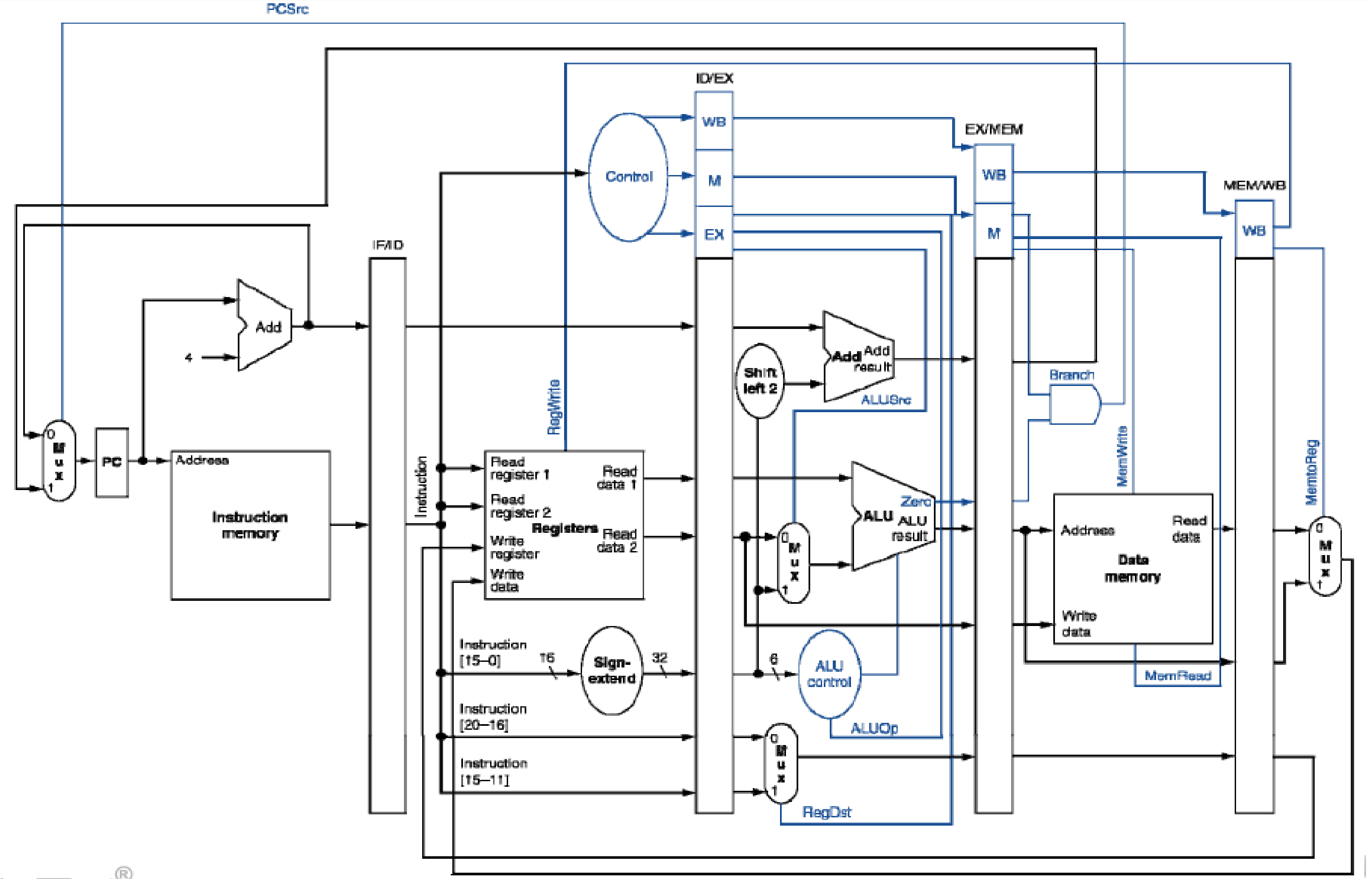
Pipeline Principle

- This allows us to decrease the clock time significantly, while keeping the CPI 1.
- What sorcery! We significantly improved the clock time while not having to an incur an equivalent tradeoff in the CPI.
- Or have we...? More on that later.

Pipeline Principle

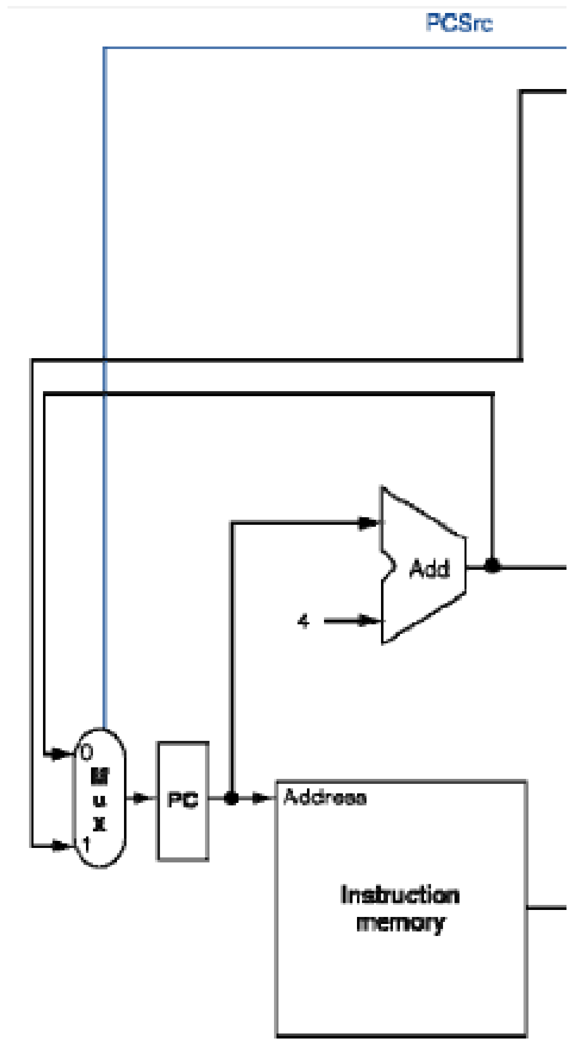
- Some notes:
  - As before clock time has to be tuned to longest latency.
  - This means some modules may still be wasting a permissible amount of time each cycle.
  - ...unless all modules take the same amount of time

Pipelined Datapath



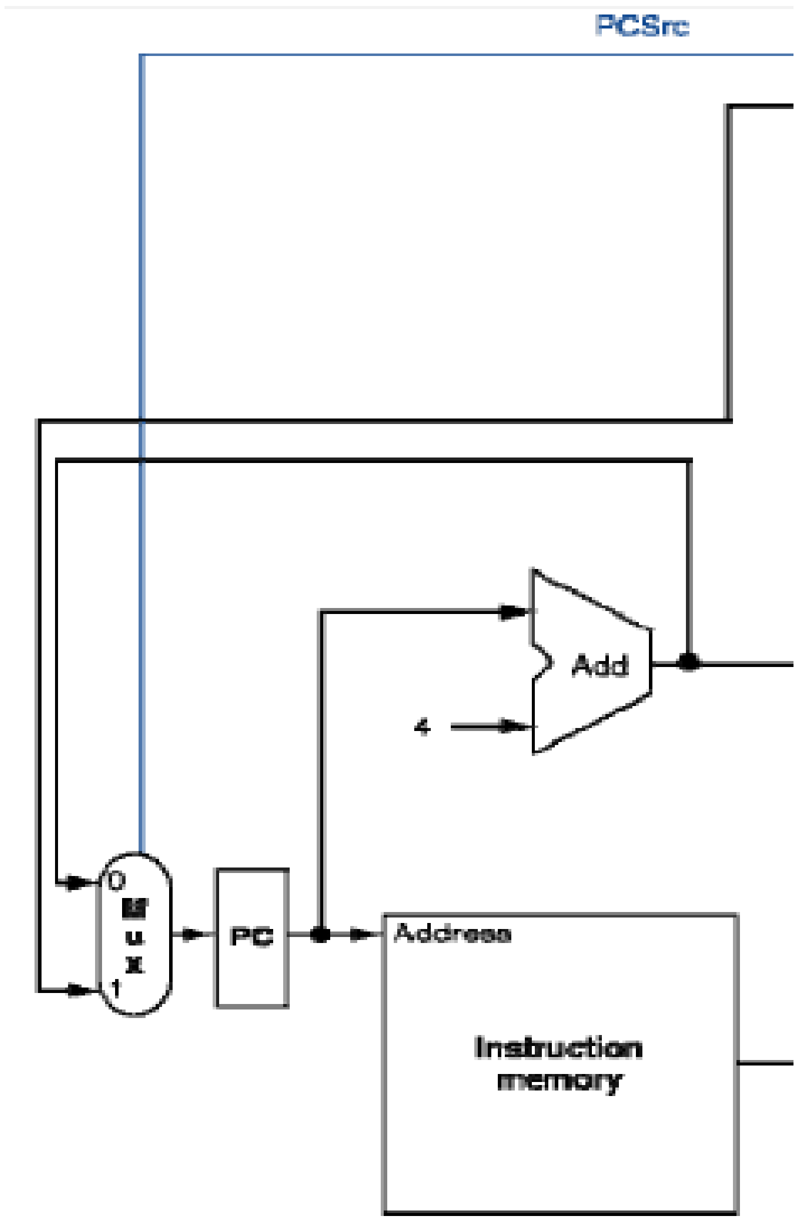
Stage 1: Instruction Fetch

- PC (unconditionally) feeds instruction memory with the address.
- PC + 4 is calculated.
- PC reads from mux the next PC.

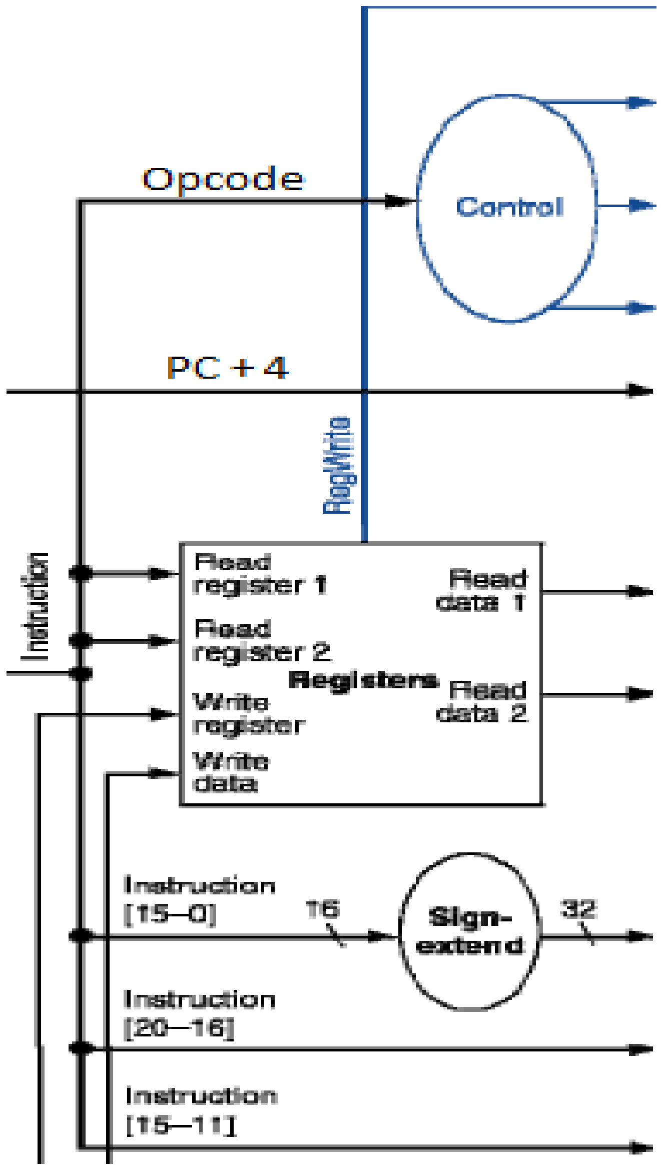


Stage 1: Instruction Fetch

- Technically, this instruction isn't really ready until the end of clock cycle.
- This means this instruction is not responsible for any control signals.
- PCSrc comes from a much later stage (Mem)

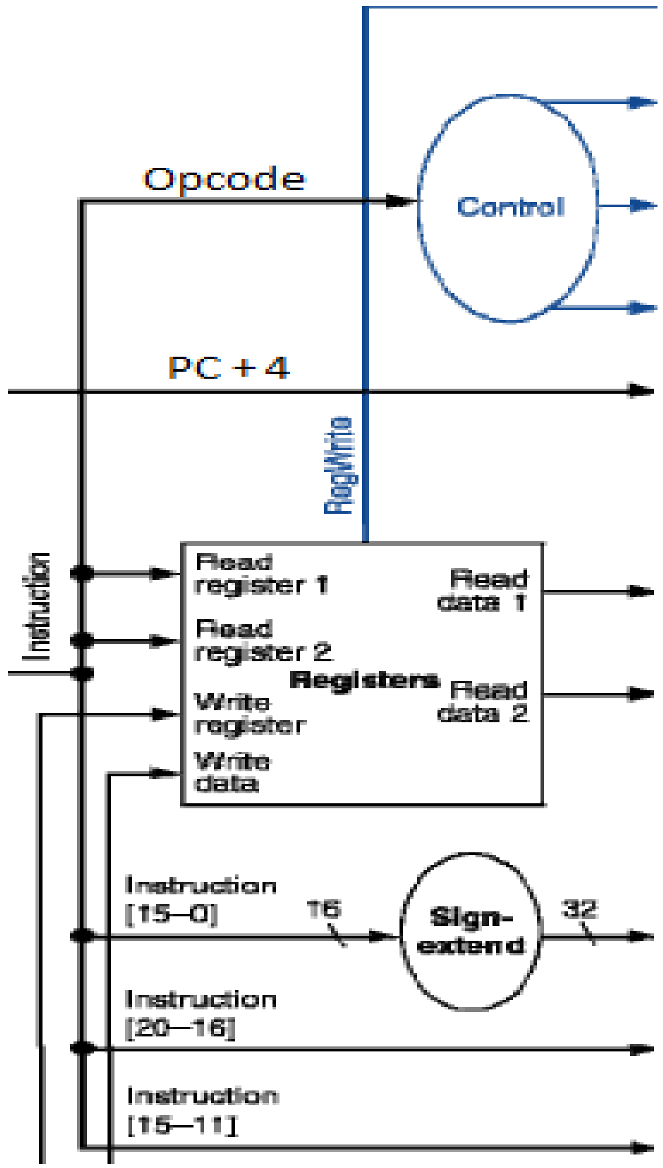


Stage 2: Instruction Decode



- Instruction is taken and broken apart into opcode, registers, etc.
- Opcode is passed to control unit which generates control signals for next stages
- Values are read from registers.

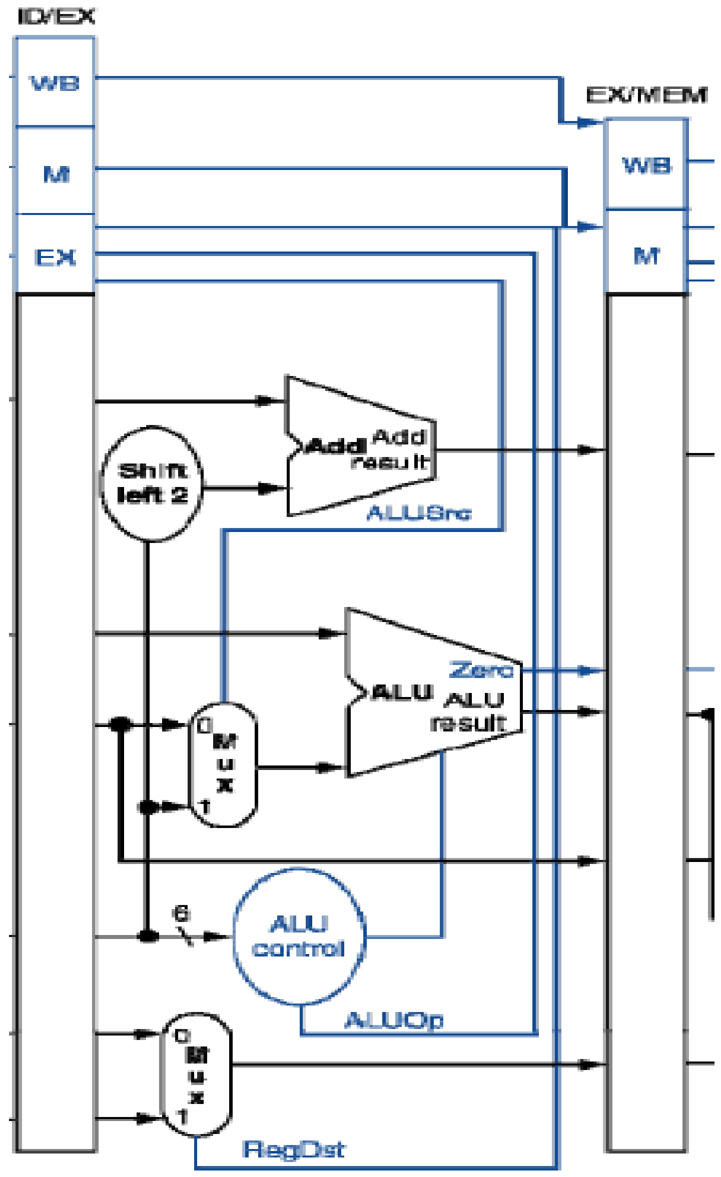
Stage 2: Instruction Decode



- RegWrite is a signal for Stage 5.
- Note that Insn[20-16] (Rt) and Insn[15-11] (Rs) are separately passed to the next stage as well
- If you watched the lectures, you already know why.

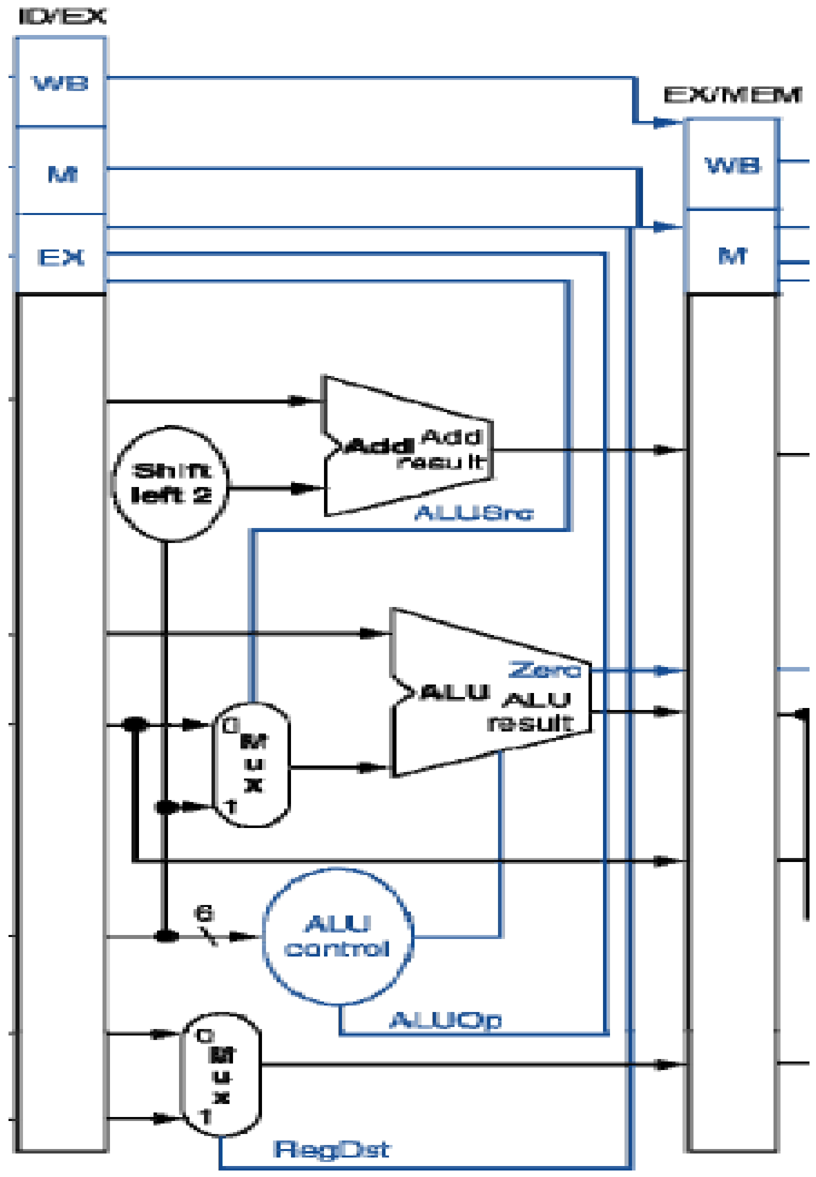
Stage 3: Execution

- Control signals are passed from stage 2. The control signals for Ex are used here (ALUSrc, ALUOp, RegDst)
- The main addition is done here.
- SE(Imm)<<2 + PC + 4 is done here



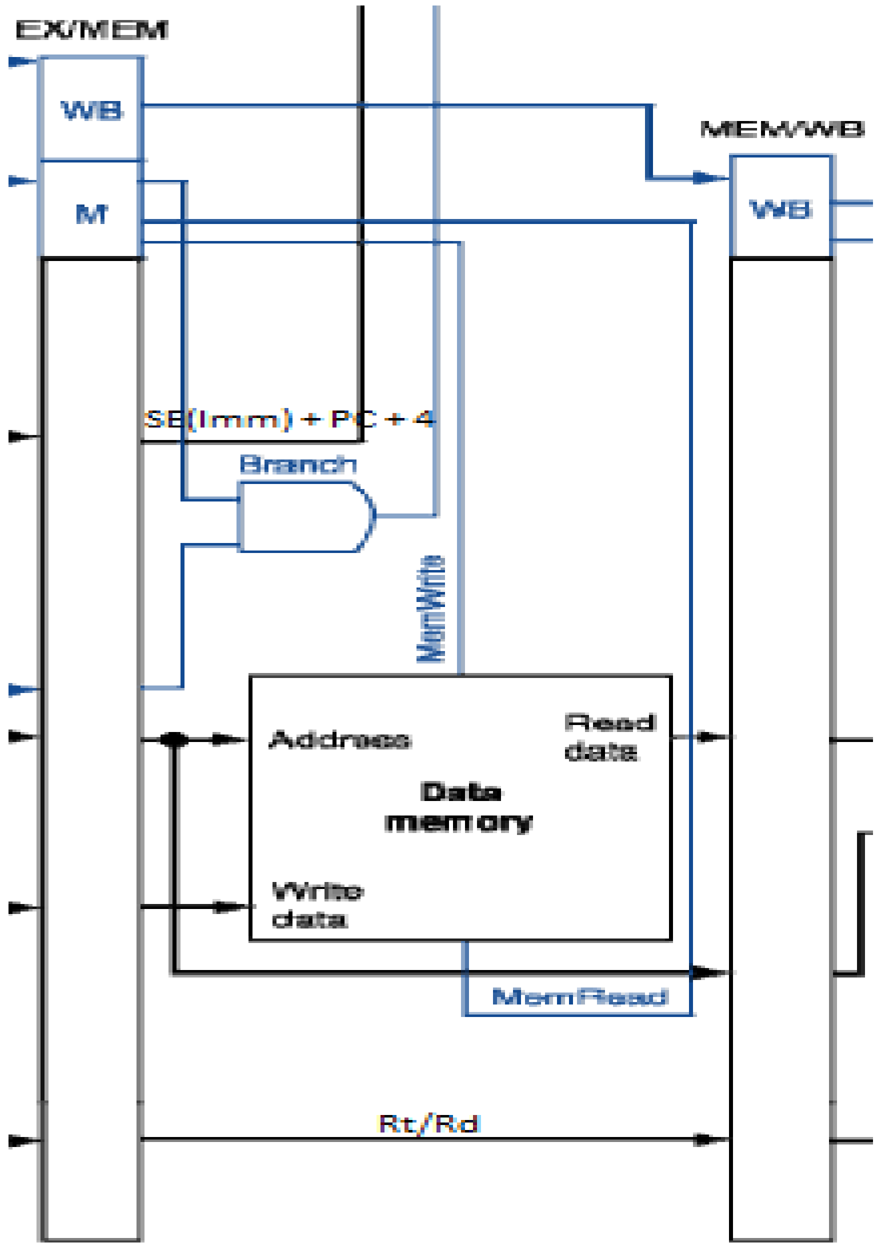
Stage 3: Execution

- Rt and Rs are selected via RegDst here.



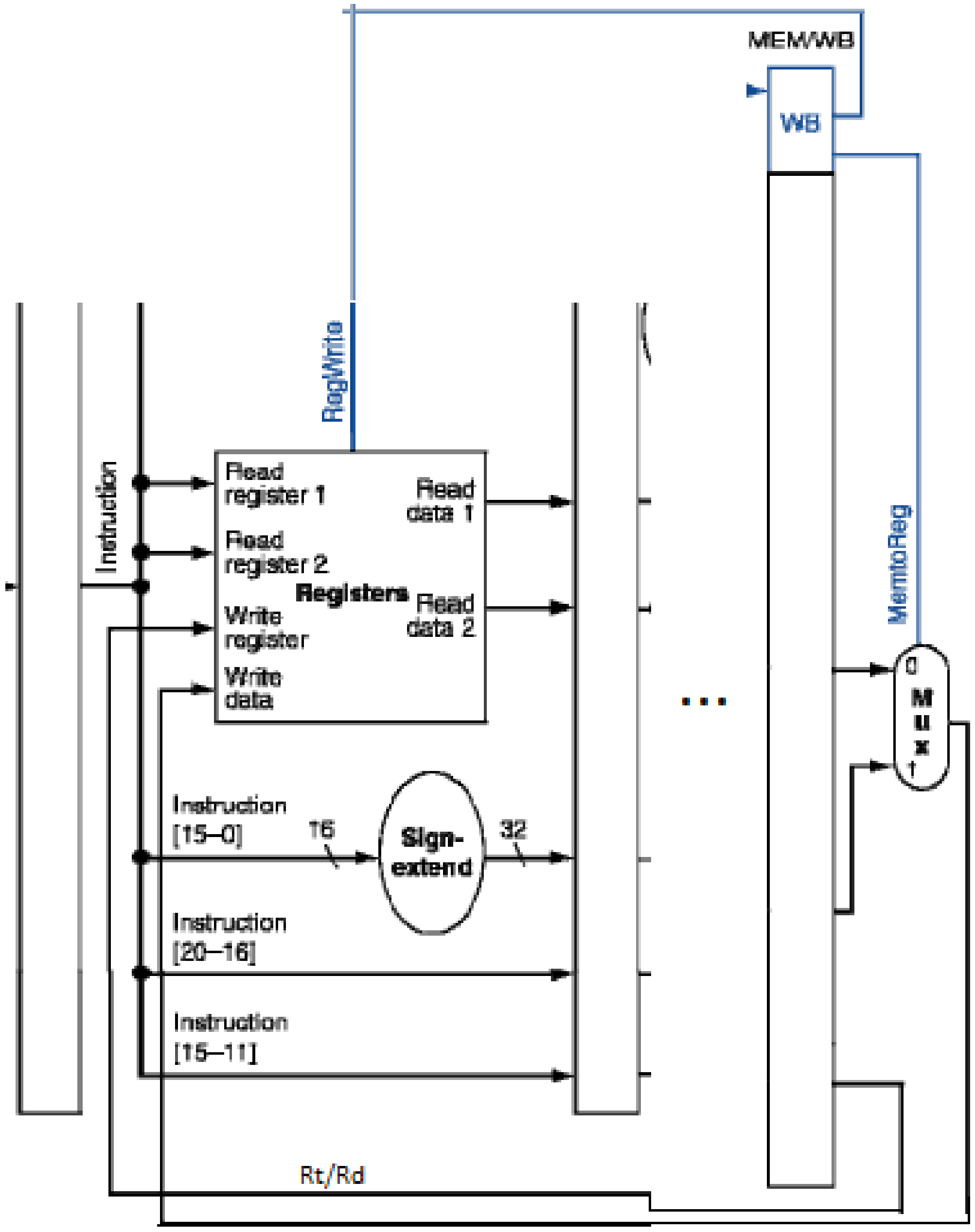
Stage 4: Memory

- This is the stage where the branch value is ready.
- If branch, PC gets the branch value this stage.
- PC will execute branch in next stage
- Either bypass memory or read/write from memory.
- Rt/Rd still hanging out.



Stage 5: Write back

- The Mem stage sends signals back to register file
- If RegWrite, write to Rt/Rd (which finally does something)
- Simultaneously, the register file is reading registers indicated by the instruction that started three cycles afterwards

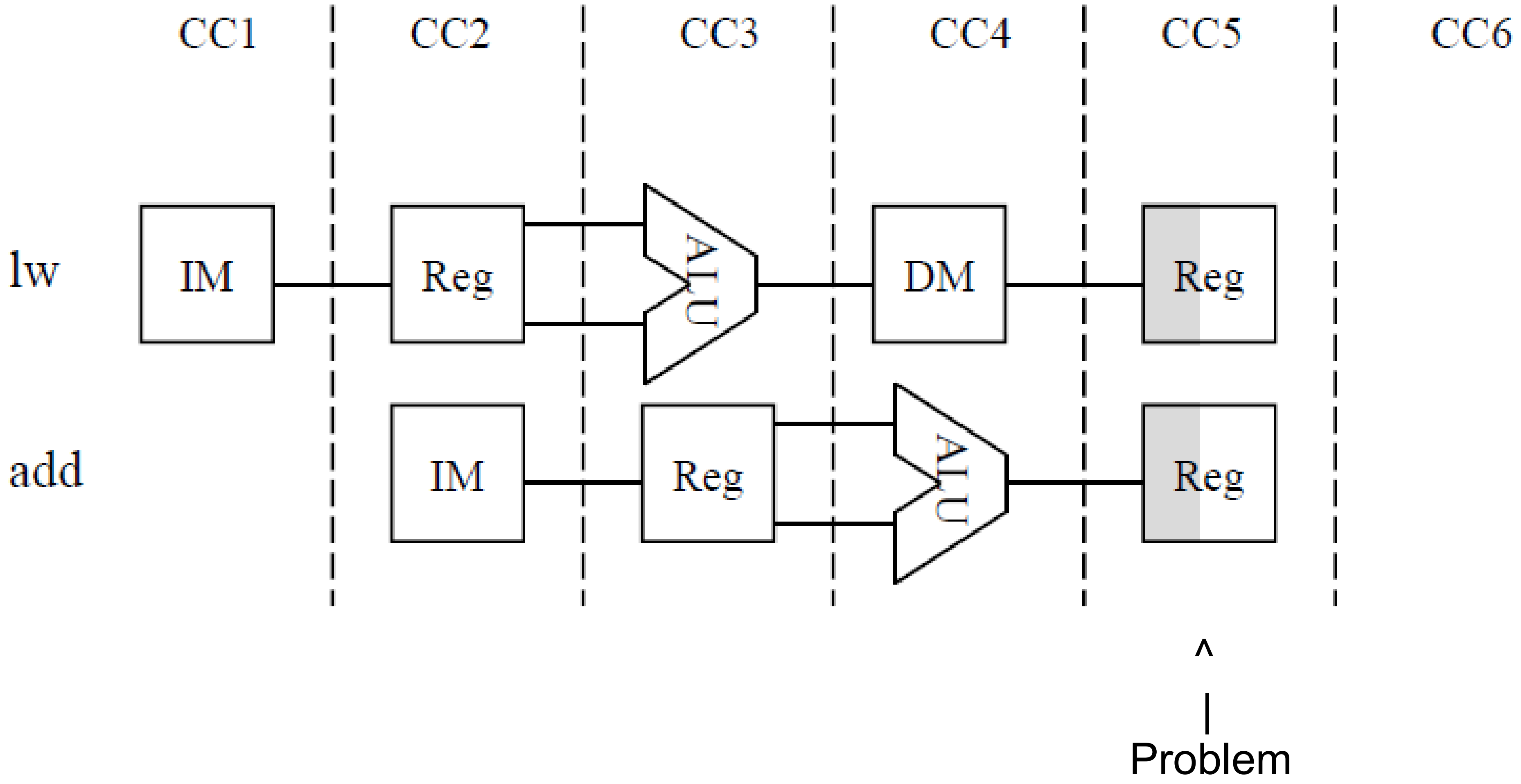


Pipeline Datapath Notes

- For stages that don't use memory, all of the important values are fed to the latch of the EX/WB.
- Can't we just use this stage and direct those outputs to the Register File and save a stage?
- Might this help to make the common case fast?

Pipeline Datapath Notes

- Consider this totally original example that I totally didn't steal from the professor's slides.



Pipeline Datapath Notes

- We could have some sort of datapath system where we conditionally bypass a stage if it's all clear.
- Let's just accept this cost of a wasted stage for simplicity.

Pipeline Datapath Weaknesses?

- What are some weaknesses of the datapath we've seen so far?

Pipeline Datapath Weaknesses?

- What are some weaknesses of the datapath we've seen so far?
  - It won't work...

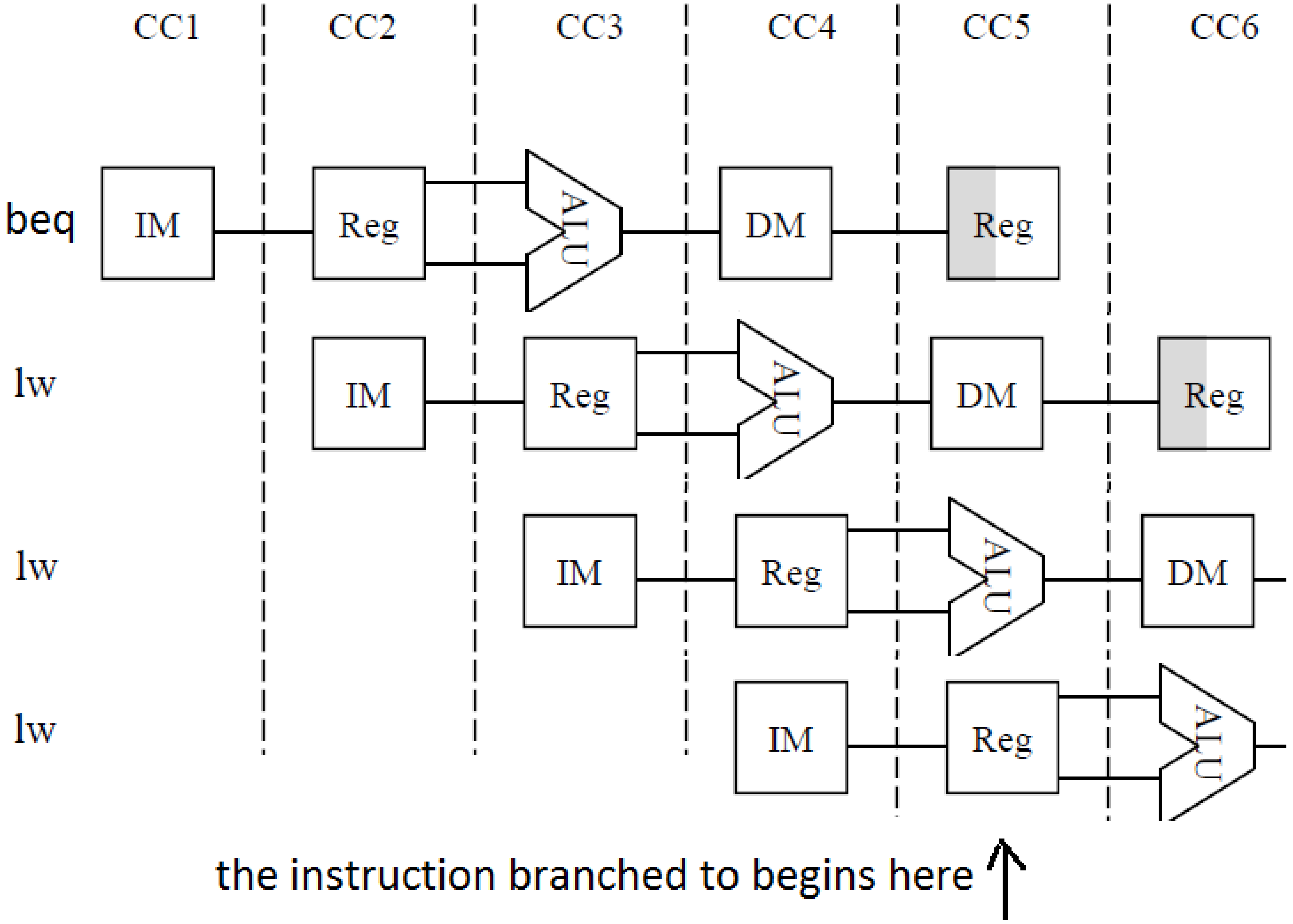
Pipeline Datapath Weaknesses?

- What are some weaknesses of the datapath we've seen so far?
  - It won't work...
  - ...not without some more considerations.
- We've already seen that we had to propagate the Rt and Rd from the ID stage to the WB stage.
- Here's a sneak peek at the fun things you'll see in the future.

Control Hazards: Branching

- Consider the sequence of instructions
- beq $r7 $r8 :LABEL
- lw $r1 ($r2)
- lw $r3 ($r4)
- lw $r5 ($r6)
- LABEL:
- add $r3 $r2 $r1

Control Hazards: Branching



- After we take the branch, we need to invalidate the loads that are about to complete

Data Hazard: Read after Write

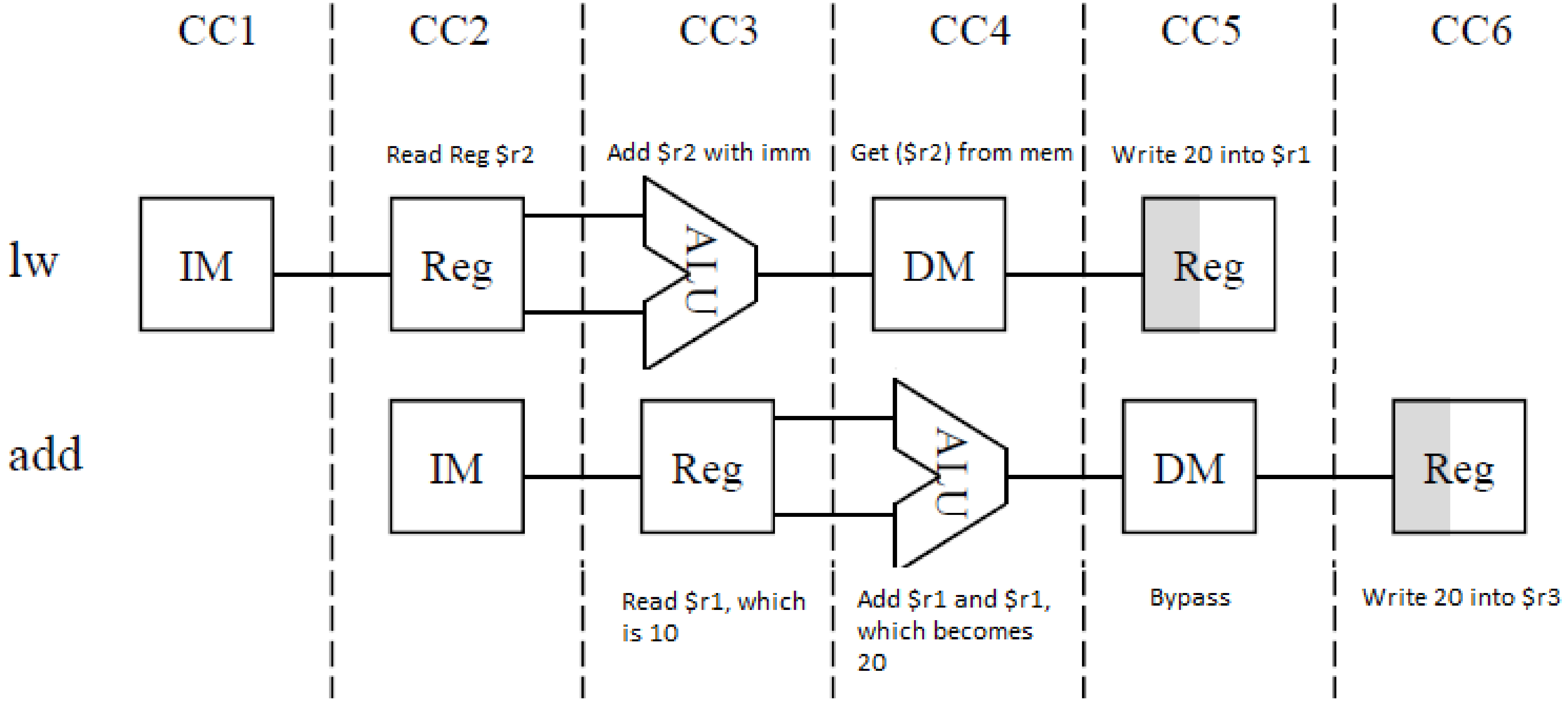
- Consider the following instructions. Before these instructions are executed, $r1 contains the value 10 and M[$r2] contains the value 20.

```
lw $r1 ($r2)
add $r3 $r1 $r1
```

- According to these instructions, we want $r3 to be 40.
- What will it actually be based on this datapath?

Data Hazard: Read after Write

- The add reads from register $r1 before the lw can write to it. We'll have to fix this.



End of

The Fifth Week

-Five Weeks Remain-