Performance and Pipelining

CS 3410, Spring 2014

Computer Science
Cornell University

See P&H Chapter: 1.6, 4.5-4.6

Announcements

HW 1

Quite long. Do not wait till the end.

PA 1 design doc

Critical to do this, else PA 1 will be hard

HW 1 review session

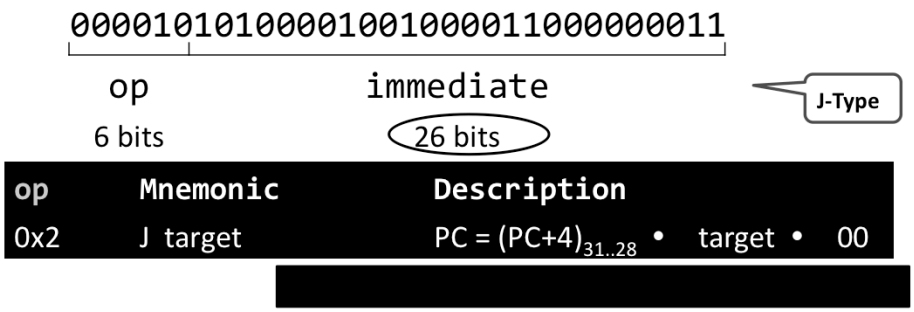
Fri (2/21) and Sun (2/23). 7:30pm.

Location: Olin 165

Prelim 1 review session

Next Fri and Sun. 7:30pm. Location: TBA

Control Flow: Absolute Jump



op	Mnemonic	Description
0x2	J target	PC = $(PC+4)_{31..28}$ • target • 00

Absolute addressing for jumps

$(PC+4)_{31..28}$ will be the same

- Jump from 0x30000000 to 0x20000000?
  - But: Jumps from 0x2FFFFFFc to 0x3xxxxxxx are possible, but not reverse
- Trade-off: out-of-region jumps vs. 32-bit instruction encoding

MIPS Quirk:

- jump targets computed using *already incremented* PC

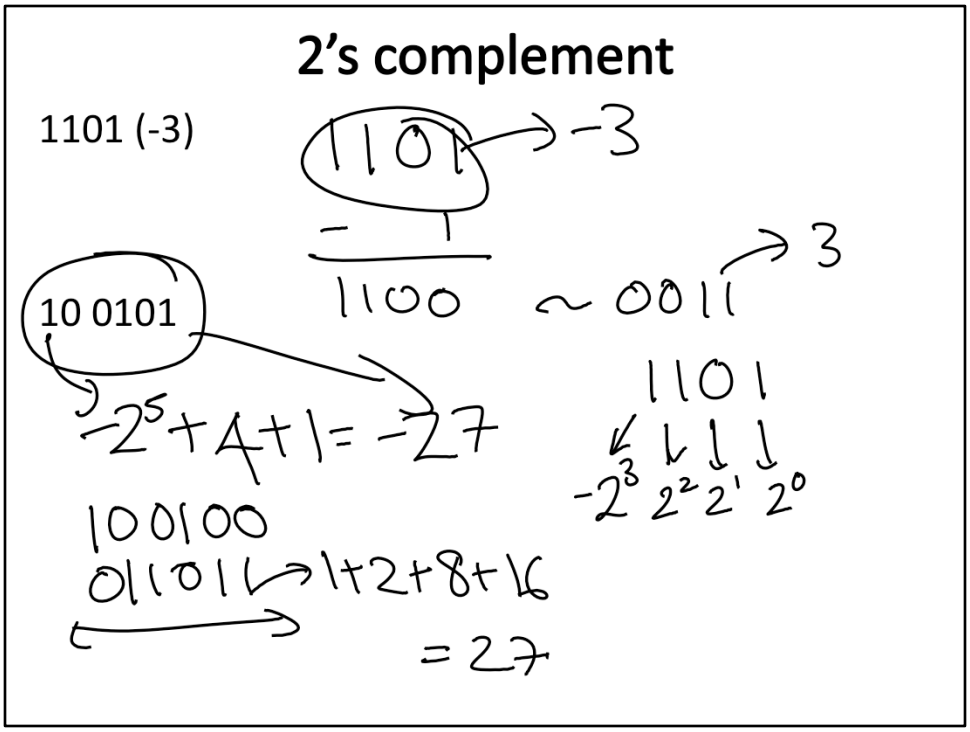
Where • is used to concatenate
0011 0000 0000 0000 0000 0000 0000 0000 (28)
0010 0000 0000 0000 0000 0000 0000 0000 (28)

PC: no explicit

Two's Complement

Non-negatives (as usual):	Negatives (two's complement: flip then add 1):	
+0 = 0000	flip = 1111	-0 = 0000
+1 = 0001	flip = 1110	-1 = 1111
+2 = 0010	flip = 1101	-2 = 1110
+3 = 0011	flip = 1100	-3 = 1101
+4 = 0100	flip = 1011	-4 = 1100
+5 = 0101	flip = 1010	-5 = 1011
+6 = 0110	flip = 1001	-6 = 1010
+7 = 0111	flip = 1000	-7 = 1001
	flip = 0111	-8 = 1000
+8 = 1000		

choose -8 so we have a sign bit
+0 = -0
wraps from +7 to -8
asymmetric: no +8
Range of values with n bits goes from unsigned: 0 to 2^n – 1
For signed: 2^(n-1)-1 to -2^n



Take 1101.
Subtract 1: 1100, flip bits 0011 which is 3. Therefore 1101 represents -3

MSB x (-2^) + all the other bits evaluated as usual
-8 + 4 + 1 = -8 + 5 = -3

MSB x (-2^5) + all the other bits evaluated as usual

Try another example
-32 + 5 = -27
Subtract 1: 100100, flip bits 011011. This is 16 + 8 + 3 = 27

MSB x (-2^5) + all the other bits evaluated as usual

Goals for today

Performance

- What is performance?
- How to get it?

Pipelining

Performance

Complex question

- How fast is the processor?
- How fast your application runs?
- How quickly does it respond to you?
- How fast can you process a big batch of jobs?
- How much power does your machine use?

Measures of Performance

Clock speed

- 1 MHz, 10^6 Hz: cycle is 1 microsecond (10^{-6})
- 1 Ghz, 10^9 Hz: cycle is 1 nanosecond (10^{-9})
- 1 Thz, 10^{12} Hz: cycle is 1 picosecond (10^{-12})

Instruction/application performance

- MIPs (Millions of instructions per second)
- FLOPs (Floating point instructions per second)
  - GPUs: GeForce GTX Titan (2,688 cores, 4.5 Tera flops, 7.1 billion transistors, 42 Gigapixel/sec fill rate, 288 GB/sec)
- Benchmarks (SPEC)

Peta: 10^(-15)
Exa: 10^(-18)
Zotta: 10^(-21)
Yotta: 10^(-24)

Benchmarks like SPEC are used to compare across architectures

Measures of Performance

Latency

- How long to finish my program
  - Response time, elapsed time, wall clock time
  - CPU time: user and system time

Throughput

- How much work finished per unit time

Ideal: Want high throughput, low latency

… also, low power, cheap ($$) etc.

How to make the computer faster?

Decrease latency

Critical Path

- Longest path determining the minimum time needed for an operation
- Determines minimum length of cycle, maximum clock frequency

Optimize for delay on the critical path

- Parallelism (like carry look ahead adder)
- Pipelining
- Both

Is the the AND path or the 32 bit adder path that is going to determine your performance in your ALU from Lab1?

Critical path is what determines what is the slowest path through the logic. And therefore, it determines the minimum length of the cycle. That in turn determines the maximum clock frequency.
For example if the critical path is 1 nanosecond, the clock frequency is at most 1 GHz.

Latency: Optimize Delay on Critical Path

E.g. Adder performance

32 Bit Adder Design	Space	Time
Ripple Carry	≈ 300 gates	≈ 64 gate delays
2-Way Carry-Skip	≈ 360 gates	≈ 35 gate delays
3-Way Carry-Skip	≈ 500 gates	≈ 22 gate delays
4-Way Carry-Skip	≈ 600 gates	≈ 18 gate delays
2-Way Look-Ahead	≈ 550 gates	≈ 16 gate delays
Split Look-Ahead	≈ 800 gates	≈ 10 gate delays
Full Look-Ahead	≈ 1200 gates	≈ 5 gate delays

Multi-Cycle Instructions

But what to do when operations take diff. times?

E.g: Assume:

- load/store: 100 ns ← 10 MHz
- arithmetic: 50 ns ← 20 MHz
- branches: 33 ns ← 30 MHz

ms = 10^{-3} second
us = 10^{-6} seconds
ns = 10^{-9} seconds

Single-Cycle CPU

10 MHz (100 ns cycle) with

– 1 cycle per instruction

100ns = 10MHz; 50ns = 20MHz; 33ns = 30 MHz

Multi-Cycle Instructions

Multiple cycles to complete a single instruction

E.g: Assume:

- load/store: 100 ns ← 10 MHz
- arithmetic: 50 ns ← 20 MHz
- branches: 33 ns ← 30 MHz

ms = 10^{-3} second
us = 10^{-6} seconds
ns = 10^{-9} seconds

Multi-Cycle CPU

30 MHz (33 ns cycle) with

- 3 cycles per load/store
- 2 cycles per arithmetic
- 1 cycle per branch

100ns = 10MHz; 50ns = 20MHz; 33ns = 30 MHz

Cycles Per Instruction (CPI)

Instruction mix for some program P, assume:

- 25% load/store (3 cycles / instruction)
- 60% arithmetic (2 cycles / instruction)
- 15% branches (1 cycle / instruction)

Multi-Cycle performance for program P:

3 * .25 + 2 * .60 + 1 * .15 = 2.1

average *cycles per instruction* (CPI) = 2.1

Multi-Cycle @ 30 MHz ← 30M cycles/sec ÷2.0 cycles/instr = 15 MIPS

vs

Single-Cycle @ 10 MHz → 10 MIPS

MIPS = millions of instructions per second

0.25 x 3 + 0.6 x 2 + 0.1 x 1 = 0.75 + 1.2 + .15 = 2.1

Total Time

CPU Time = # Instructions x CPI x Clock Cycle Time

Say for a program with 400k instructions, 30 MHz:

Time = 400k x 2.1 x 33 ns = 27 millisecs

$$I \times \frac{\text{cycles}}{\text{instr}} \times \text{time cycle}$$

Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

Instruction mix (for P):

- 25% load/store, CPI = 3
- 60% arithmetic, CPI = 2
- 15% branches, CPI = 1

7

So the goal is to make it run at 30 MIPs.

CPI = (.25 x 3 + .6 x 2 + .15 x 1)/1 = 2.1

MIPS = 30 MHz/2.1 = 14.28 MIPS. Call it 15 MIPS

Want to double It to 28.56

Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

Instruction mix (for P):

- 25% load/store, CPI = 3
- 60% arithmetic, CPI = 2
- 15% branches, CPI = 1

0.25×3+
0.6×1+
0.15×1

First lets try CPI of 1 for arithmetic.
Is that 2x faster overall? No
How much does it improve performance?

0.75+0.6+0.15
= 1.5

So the goal is to make it run at approximately 30 MIPs.

Original CPI = (.25 x 3 + .6 x 2 + .15 x 1)/1 = 2.1
MIPS = 30 MHz/2.1 = 14.28 MIPS. Call it 15 MIPS

Say you drop the CPI for the arithmetic operation to 1. Will that double it? No.
.25 x 3 + .6 + .15 = 1.5
30 MHz/1.5 = 20MIPS

Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

2.1 CPI → 2.1/2

1.05

Instruction mix (for P):

- 25% load/store, CPI = 3
- 60% arithmetic, CPI = 2
- 15% branches, CPI = 1

3 x 0.25 + x x 0.6 + 0.15 x 1 = 1.05

x = 0.25

But we want to half our CPI. Let the new arithmetic operation have a CPI of x.

3 x .25 + x * 0.6 + .15 = 1.05
.75 + .15 + x * 0.6 = 1.05
x = 0.25

That's a big improvement you need!

Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

Instruction mix (for P):

- 25% load/store, CPI = 3
- 60% arithmetic, CPI = 2
- 15% branches, CPI = 1

To double performance CPI has to go from 2 to 0.25

Amdahl's Law

Amdahl's Law

Execution time after improvement =

$$\frac{\text{execution time affected by improvement}}{\text{amount of improvement}} + \text{execution time unaffected}$$

Or: Speedup is limited by popularity of improved feature

Corollary: Make the common case fast

Caveat: Law of diminishing returns

Consider our GPU example with 2k cores.

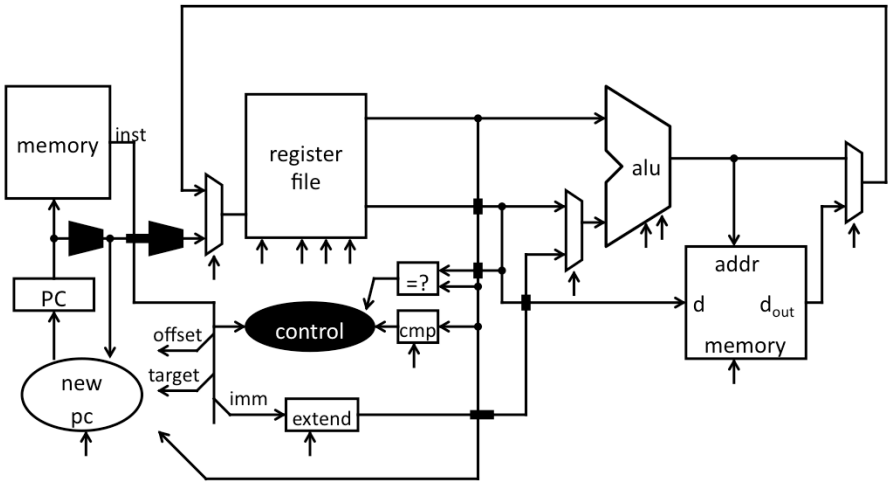
Say we have a program that takes 2000 seconds to run: 200 seconds is the start up time (reading data), and 1800 is the "main" algorithm.! This doesn't seem so bad. 200/2000 = 10% only of startup and 90% of the program is in the slow algorithm.

We want to speed it up by running on a GPU with 2000 cores! Ideally we would get 2000x speedup and the program will run in 1 second.

But when we port it to the GPU, we can only improve the "main" algorithm which is highly parallelizable. You can improve the 1800 seconds down to < 1 second say, because you can fully parallelize the algorithm on 2000 cores.

But still the time for the whole program is 201 seconds. So you threw 2000 cores at the problem, but your speedup is 2000/201 which is approximately 10x. So with 2000 cores you only got 10x speedup. Amdahl's law expresses that "unfortunate" relation.

Review: Single cycle processor



Review: Single Cycle Processor

Advantages

- Single cycle per instruction make logic and clock simple

Disadvantages

- Since instructions take different time to finish, memory and functional unit are not efficiently utilized
- Cycle time is the longest delay
  - Load instruction
- Best possible CPI is 1 (actually < 1 w parallelism)
  - However, lower MIPS and longer clock period (lower clock frequency); hence, lower performance

Review: Multi Cycle Processor

Advantages

- Better MIPS and smaller clock period (higher clock frequency)
- Hence, better performance than Single Cycle processor

Disadvantages

- Higher CPI than single cycle processor

Pipelining: Want better Performance

- want small CPI (close to 1) with high MIPS and short clock period (high clock frequency)

Improving Performance

Parallelism

Pipelining

Both!

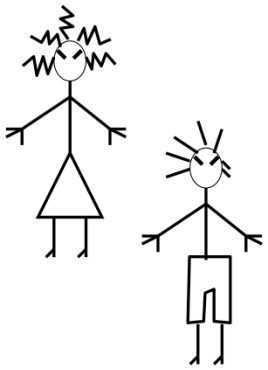
Single Cycle vs Pipelined Processor

See: P&H Chapter 4.5

The Kids

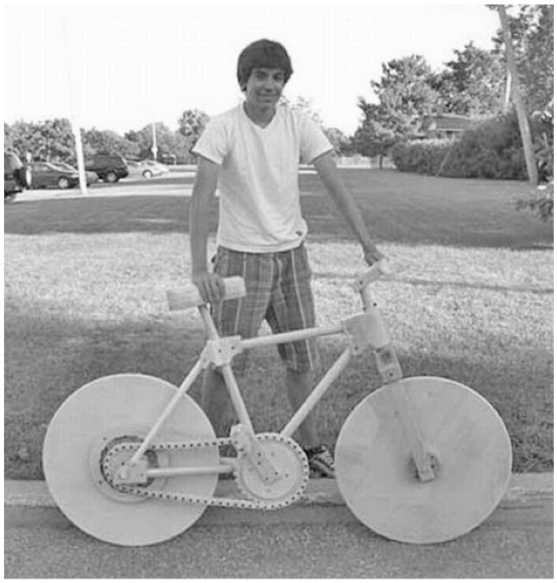
Alice

Bob



They don’t always get along…

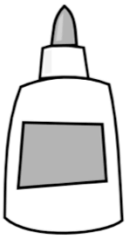
The Bicycle



The Materials

Saw

Drill

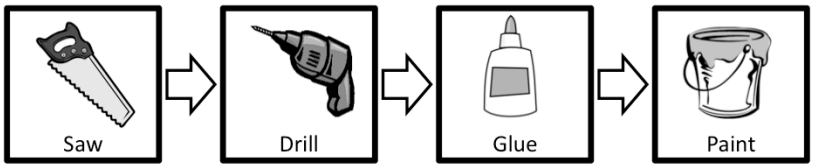


Glue

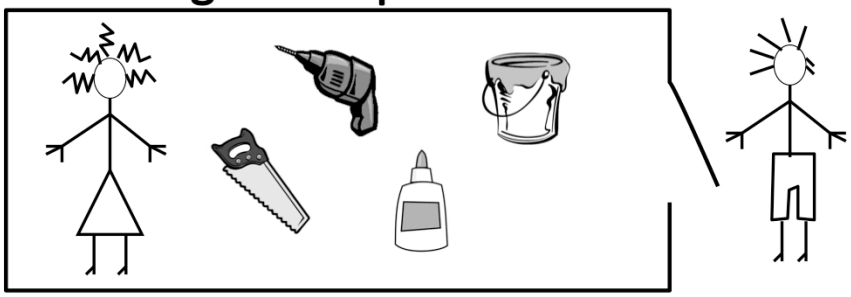
Paint

The Instructions

N pieces, each built following same sequence:



Design 1: Sequential Schedule

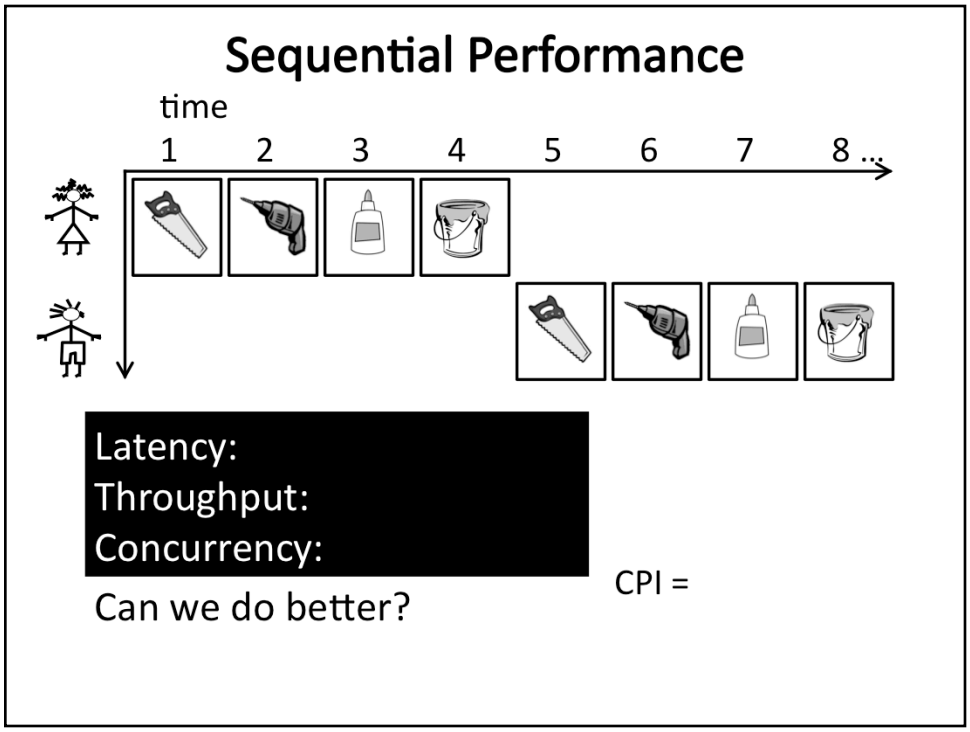


Alice owns the room

Bob can enter when Alice is finished

Repeat for remaining tasks

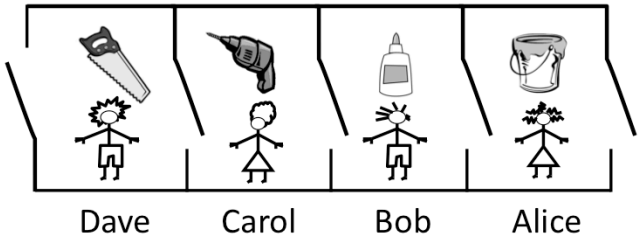
No possibility for conflicts



Latency = 4
CPI = 4 (here the instruction is the construction of the bike)
Throughput = 2 bikes in 8 secs. So 1 task in 4 secs. So ¼ throughput
Concurrency: 0

Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

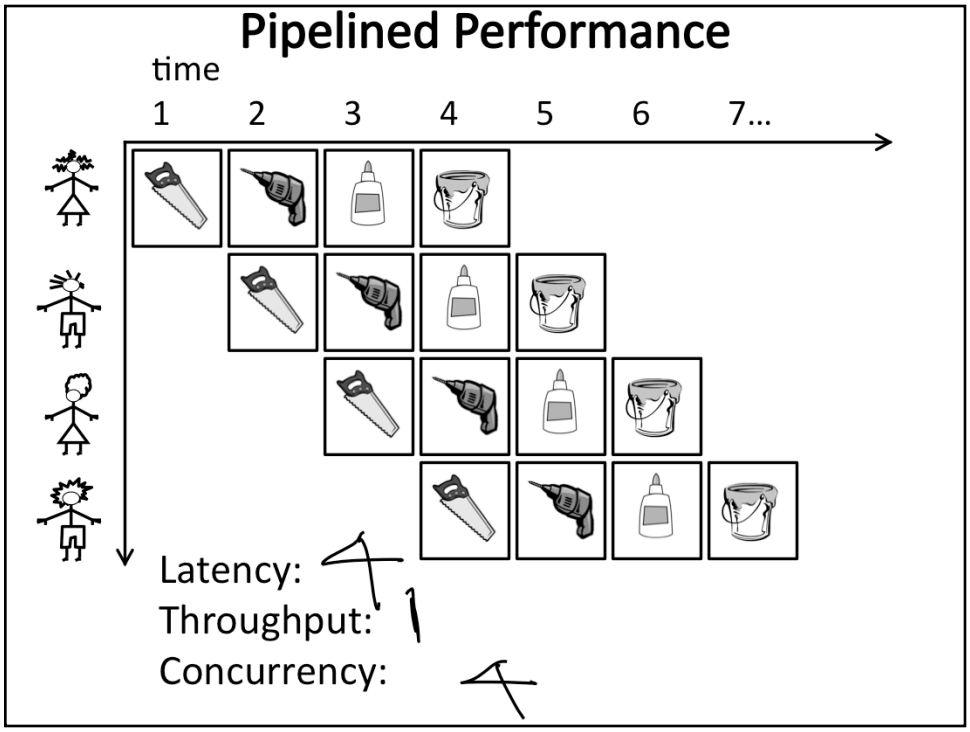


One person owns a stage at a time

4 stages

4 people working simultaneously

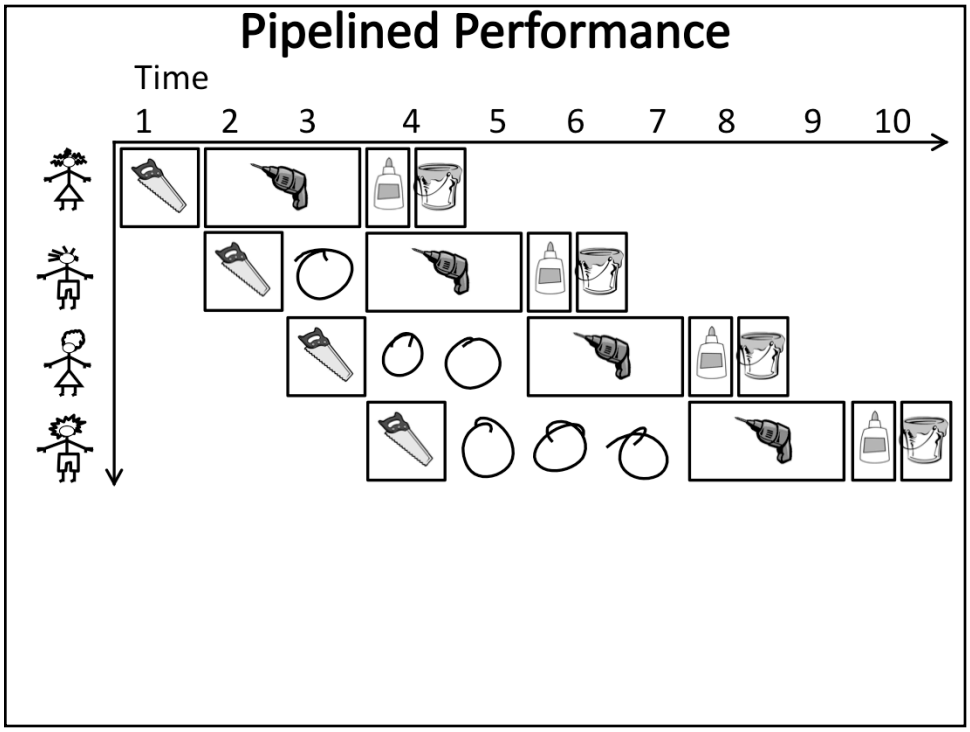
Everyone moves right in lockstep



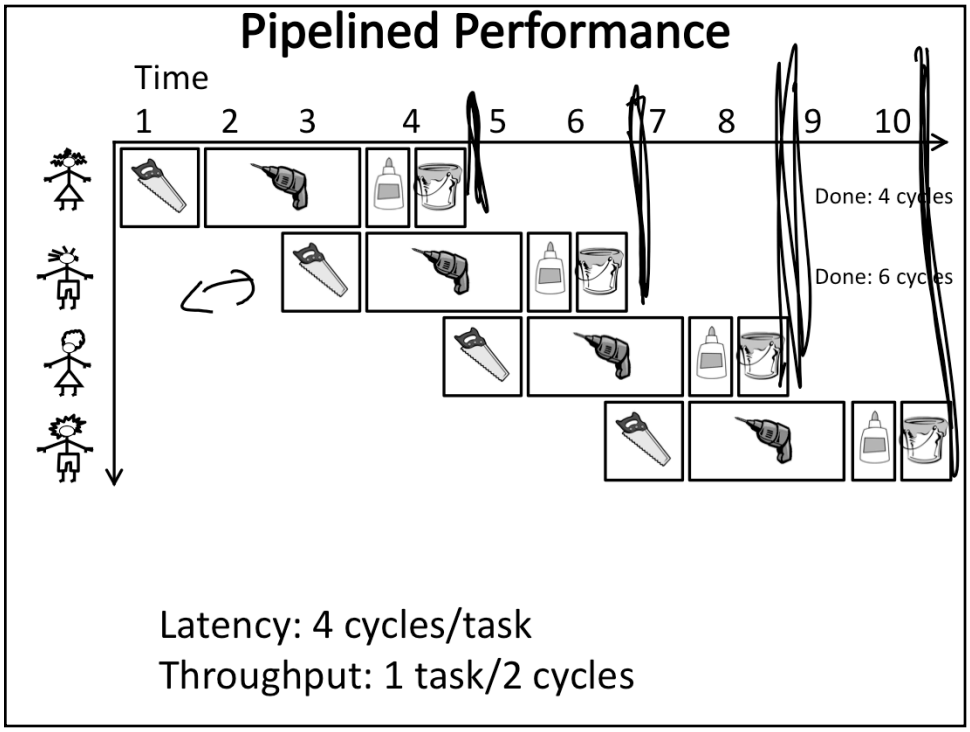
Delay: 4 cycles / task

Throughput: 1 task / cycle (huge improvement) after your initial startup of filling the pipeline.

Parallelism: 4 concurrent tasks



Now what if drilling is twice as long but the gluing and paint are ½ each.



So total latency is still the same: 4

CPI: 4 cycles / task

Throughput: 1 task / 2 cycle, First task out at cycle 4, second at 6, third at 8, fourth at 10. So ½. Not 1!

Lessons

Principle:

Throughput increased by parallel execution

Balanced pipeline very important

Else slowest stage dominates performance

Pipelining:

- Identify *pipeline stages*
- Isolate stages from each other
- Resolve pipeline *hazards* (next lecture)

MIPs designed for pipelining

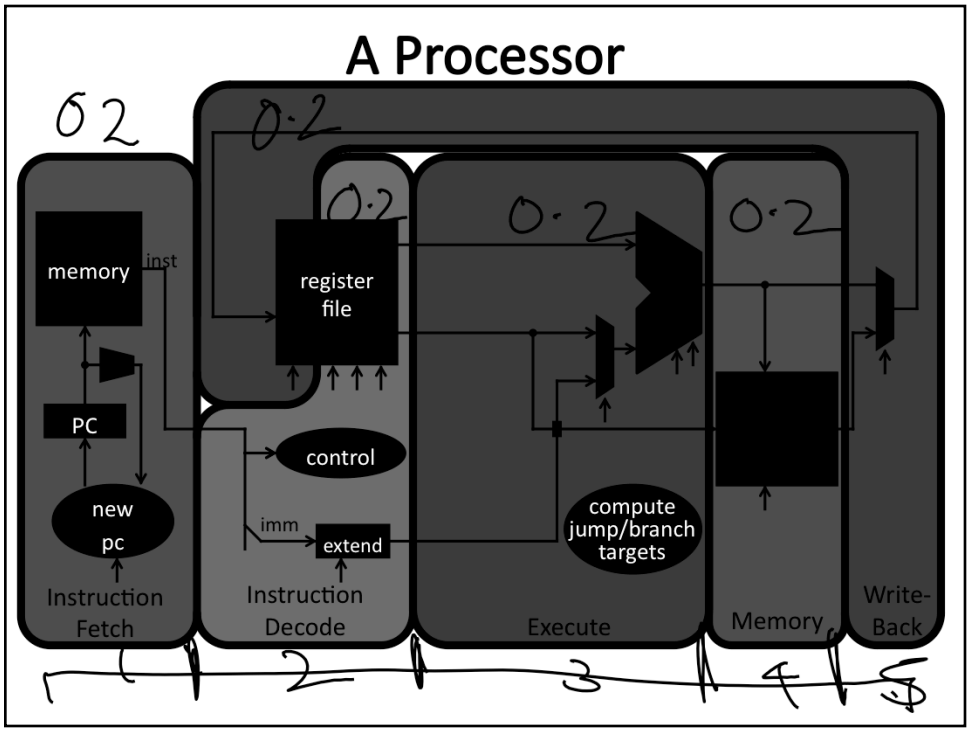
- Instructions same length
  - 32 bits, easy to fetch and then decode
- 3 types of instruction formats
  - Easy to route bits between stages
  - Can read a register source before even knowing what the instruction is
- Memory access through lw and sw only
  - Access memory after ALU

Basic Pipeline

Five stage "RISC" load-store architecture

1. Instruction fetch (IF)
   - get instruction from memory, increment PC
2. Instruction Decode (ID)
   - translate opcode into control signals and read registers
3. Execute (EX)
   - perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   - access memory if needed
5. Writeback (WB)
   - update register file

This is simpler than the MIPS, but we're using it to get the concepts across – everything you see here applies to MIPS, but we have to deal w/ fewer bits in these examples (that's why I like them)



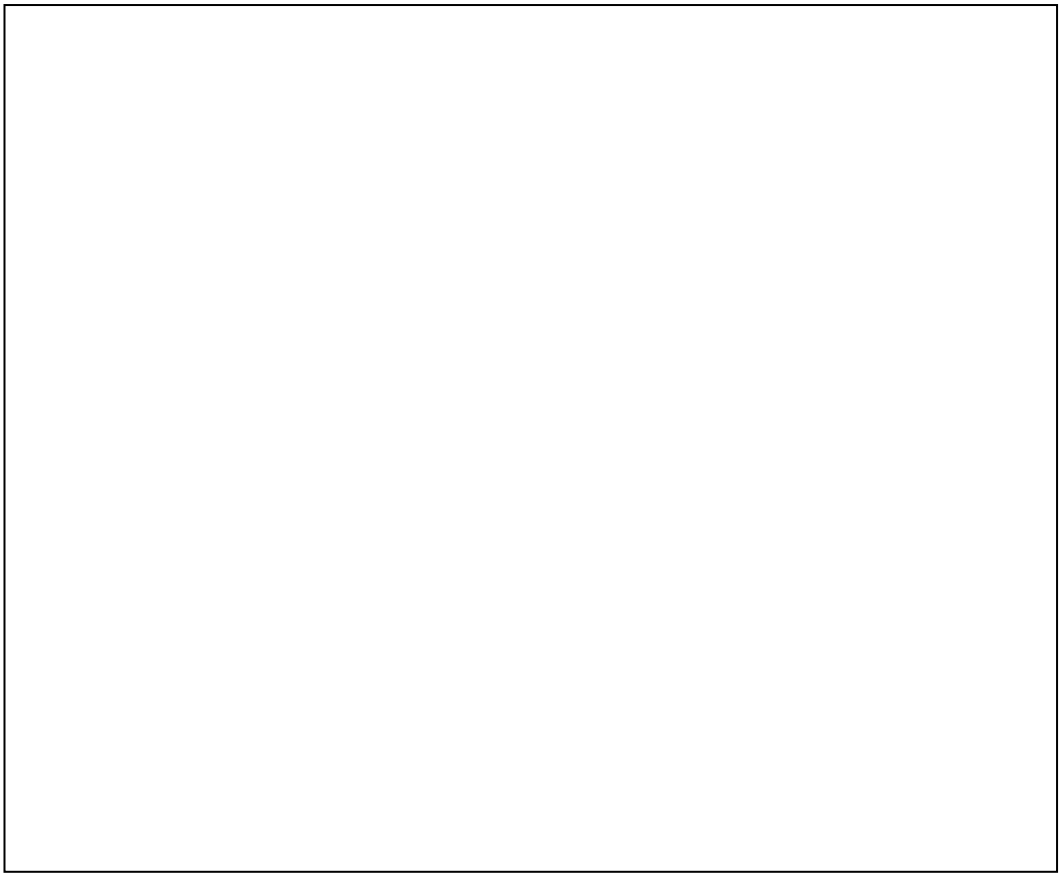
What does that do to a clock cycle. It is the time for 1 stage. So 5 times faster in this case (ASSUMING all stages are approximately equal sized)
Left to right flow except for the write-back phase and the branch targets that can change the PC. Otherwise left to right.

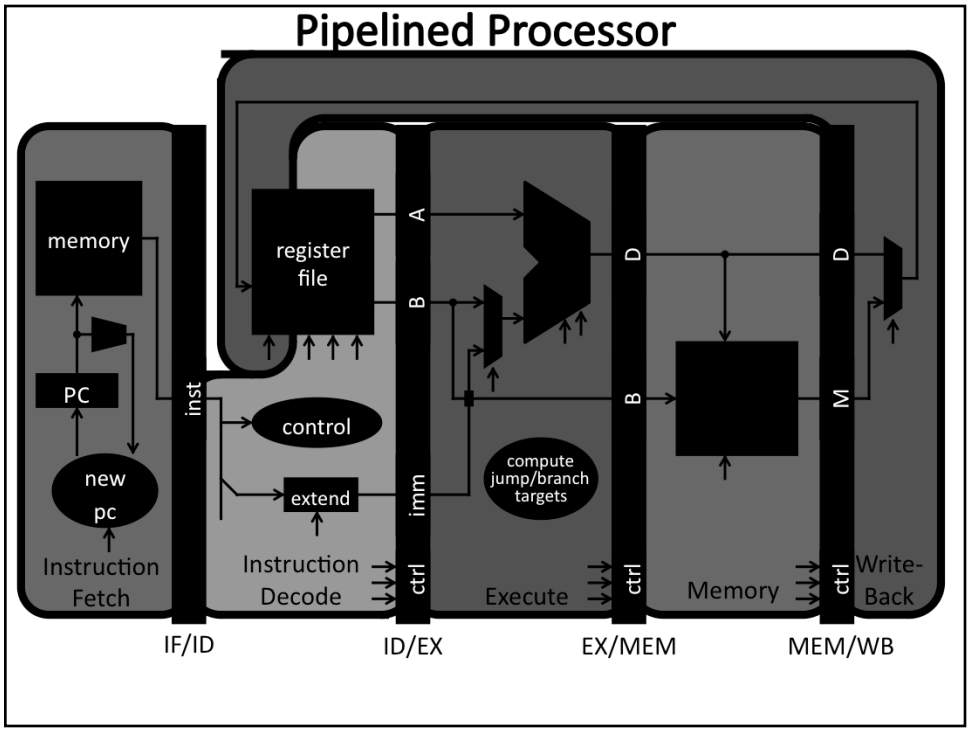
Principles of Pipelined Implementation

Break instructions across multiple clock cycles (five, in this case)

Design a separate stage for the execution performed during each clock cycle

Add pipeline registers (flip-flops) to isolate signals between different stages




Pipelined Processor
memory
register file
PC
new pc
inst
control
extend
A
B
imm
ctrl
D
B
ctrl
compute jump/branch targets
D
M
ctrl
Instruction Fetch
Instruction Decode
Execute
Memory
Write-Back
IF/ID
ID/EX
EX/MEM
MEM/WB

IF

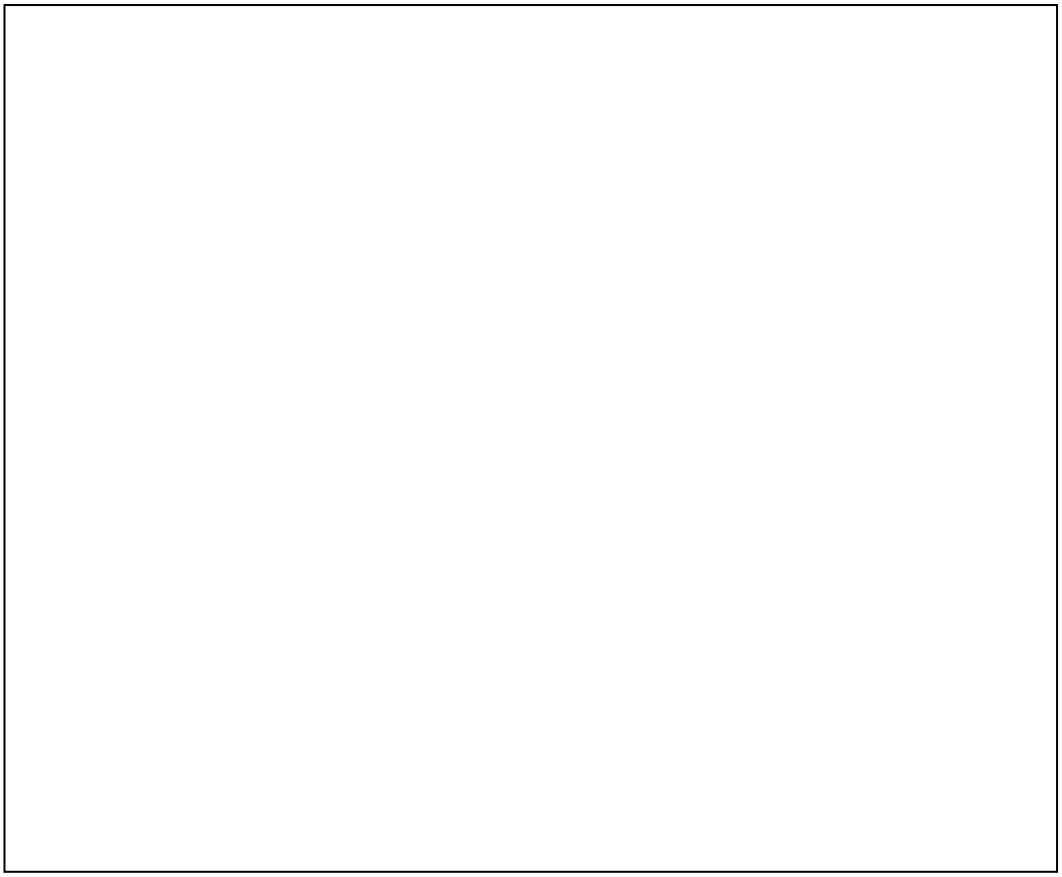
Stage 1: Instruction Fetch

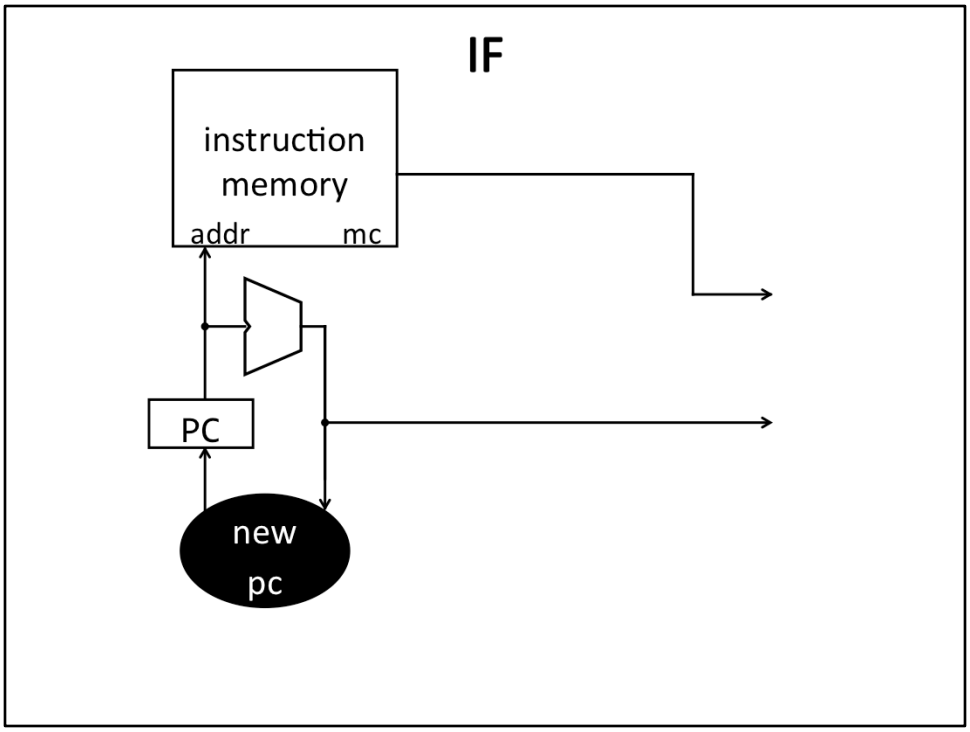
Fetch a new instruction every cycle

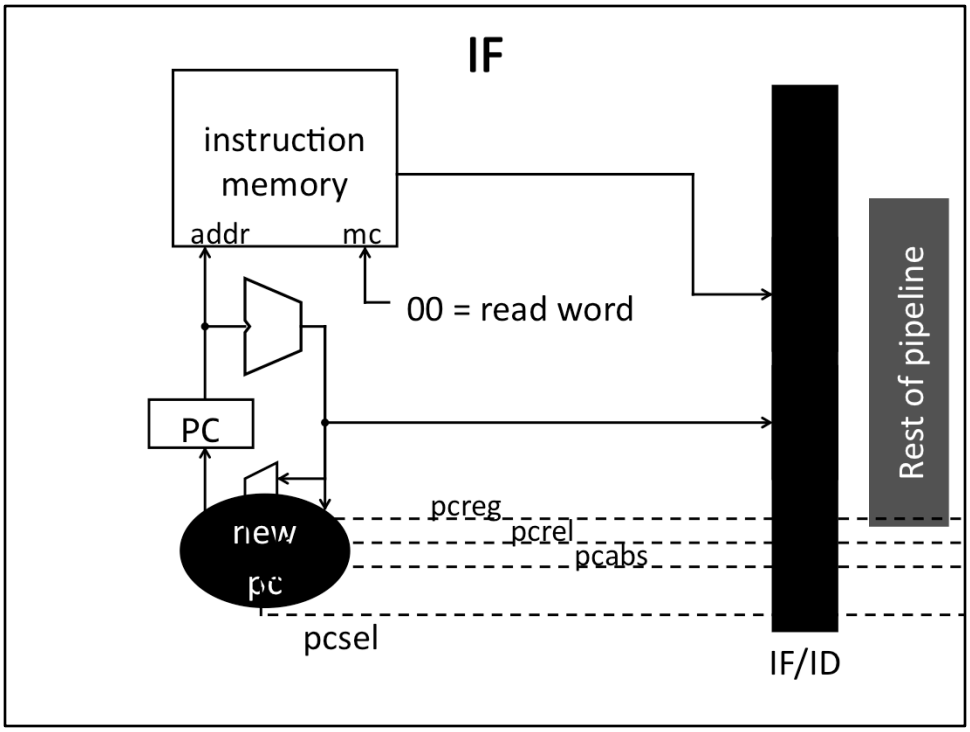
- Current PC is index to instruction memory
- Increment the PC at end of cycle (assume no branches for now)

Write values of interest to pipeline register (IF/ID)

- Instruction bits (for later decoding)
- PC+4 (for later computing branch targets)




IF
instruction
memory
addr
mc
PC
new
pc


IF
instruction memory
addr
mc
00 = read word
PC
new pc
pcreg
pcrel
pcabs
pcsel
IF/ID
Rest of pipeline

ID

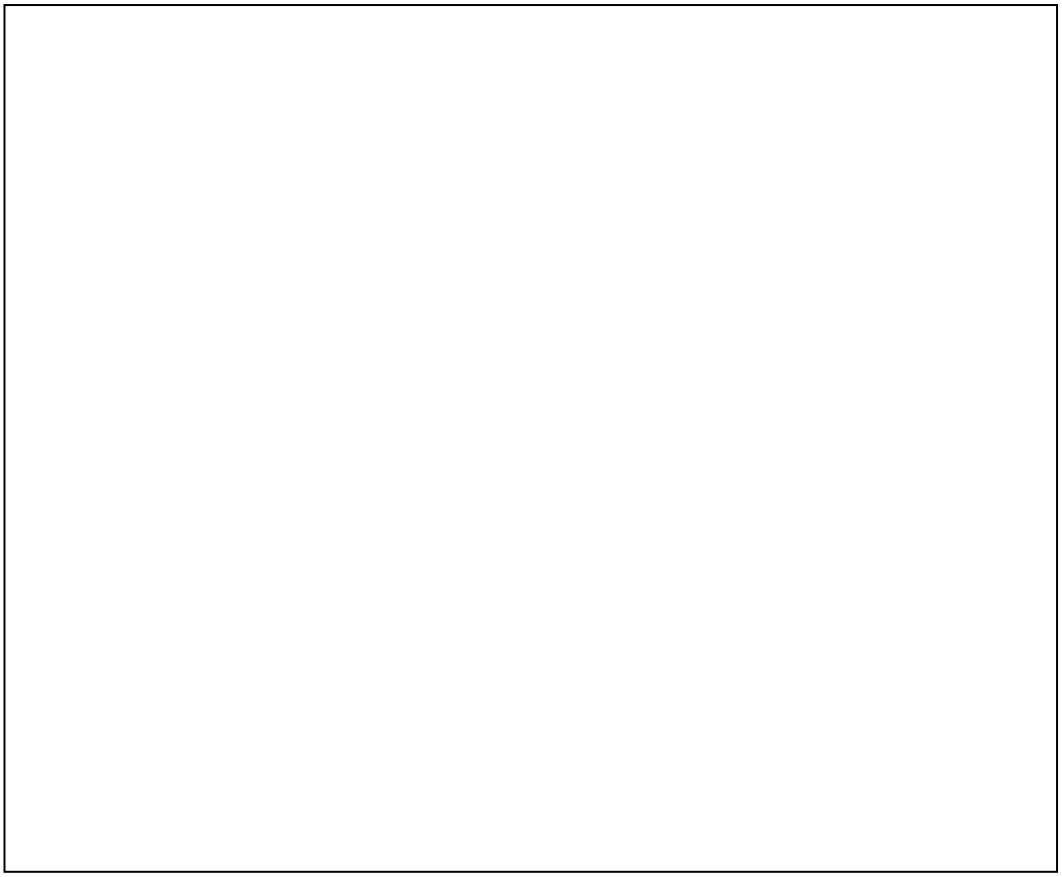
Stage 2: Instruction Decode

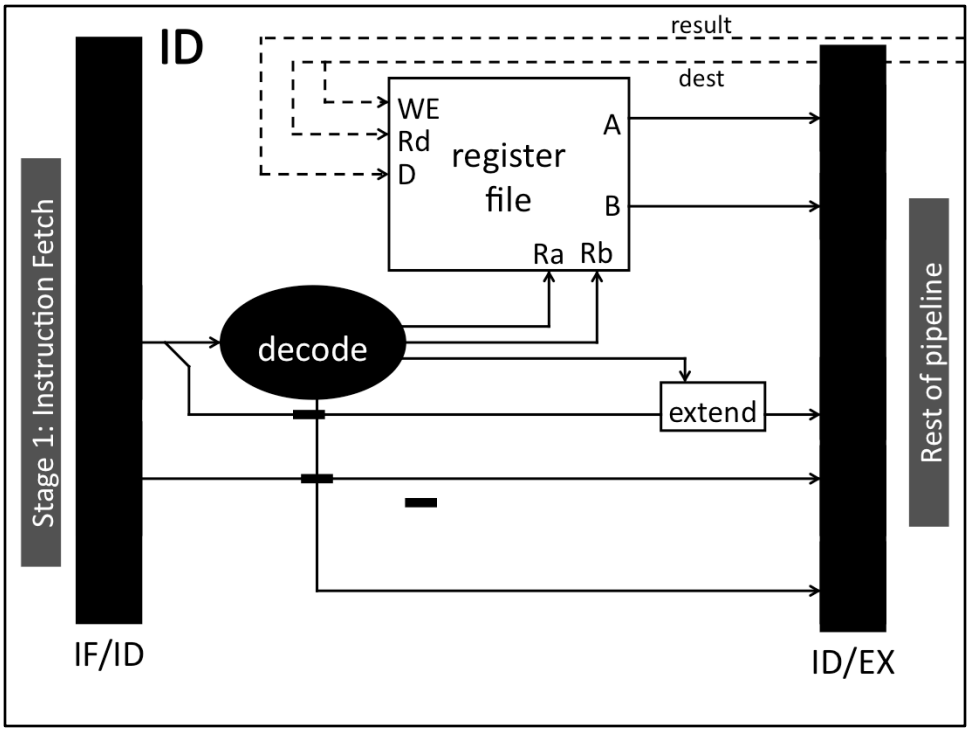
On every cycle:

- Read IF/ID pipeline register to get instruction bits
- Decode instruction, generate control signals
- Read from register file

Write values of interest to pipeline register (ID/EX)

- Control information, Rd index, immediates, offsets, …
- Contents of Ra, Rb
- PC+4 (for computing branch targets later)





Early decode: decode all instr in ID, pass control signals to later stages
Late decode: decode some instr in ID, pass instr so each stage computes its own control signals

EX

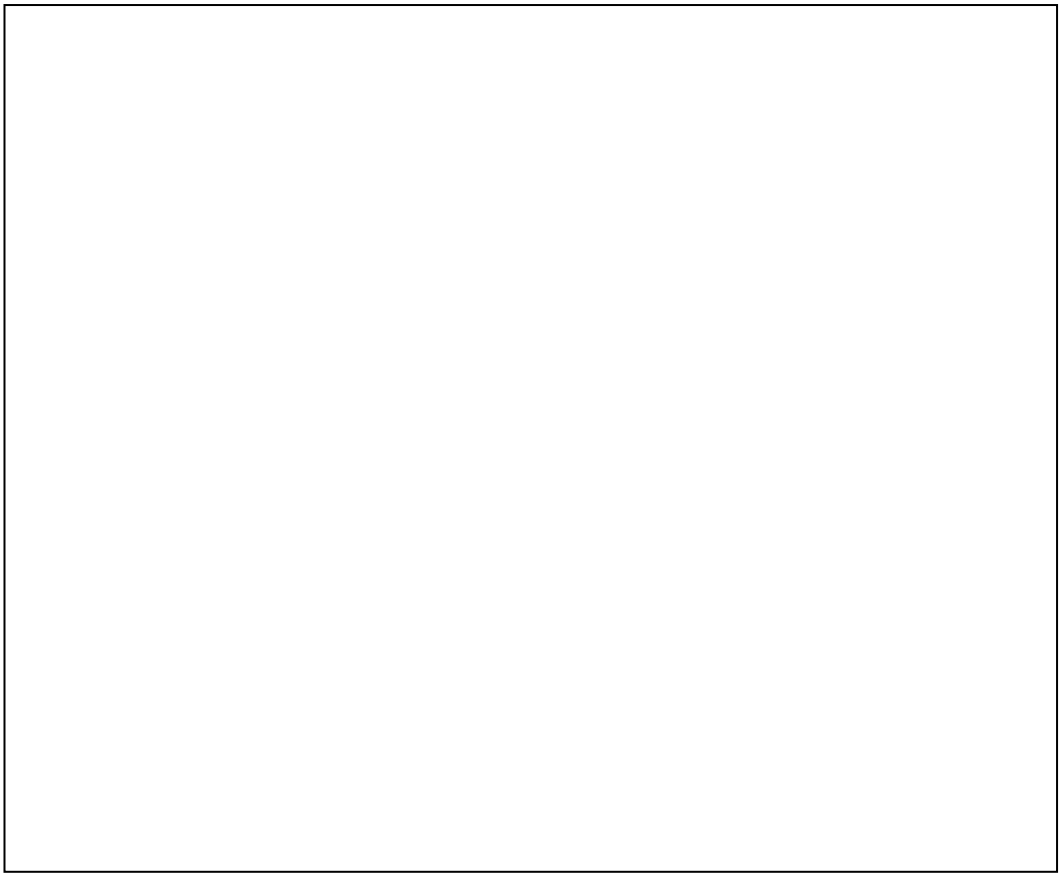
Stage 3: Execute

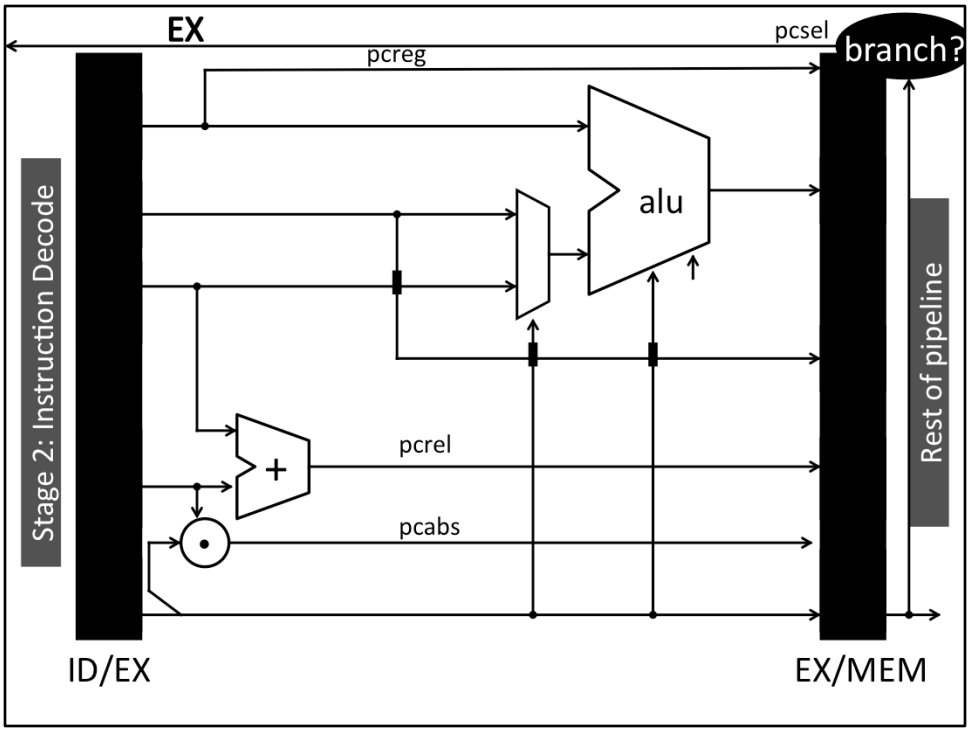
On every cycle:

- Read ID/EX pipeline register to get values and control bits
- Perform ALU operation
- Compute targets (PC+4+offset, etc.) *in case* this is a branch
- Decide if jump/branch should be taken

Write values of interest to pipeline register (EX/MEM)

- Control information, Rd index, …
- Result of ALU operation
- Value *in case* this is a memory store instruction




EX
pcsel
branch?
pcreg
alu
Stage 2: Instruction Decode
Rest of pipeline
+
pcrel
pcabs
ID/EX
EX/MEM

MEM

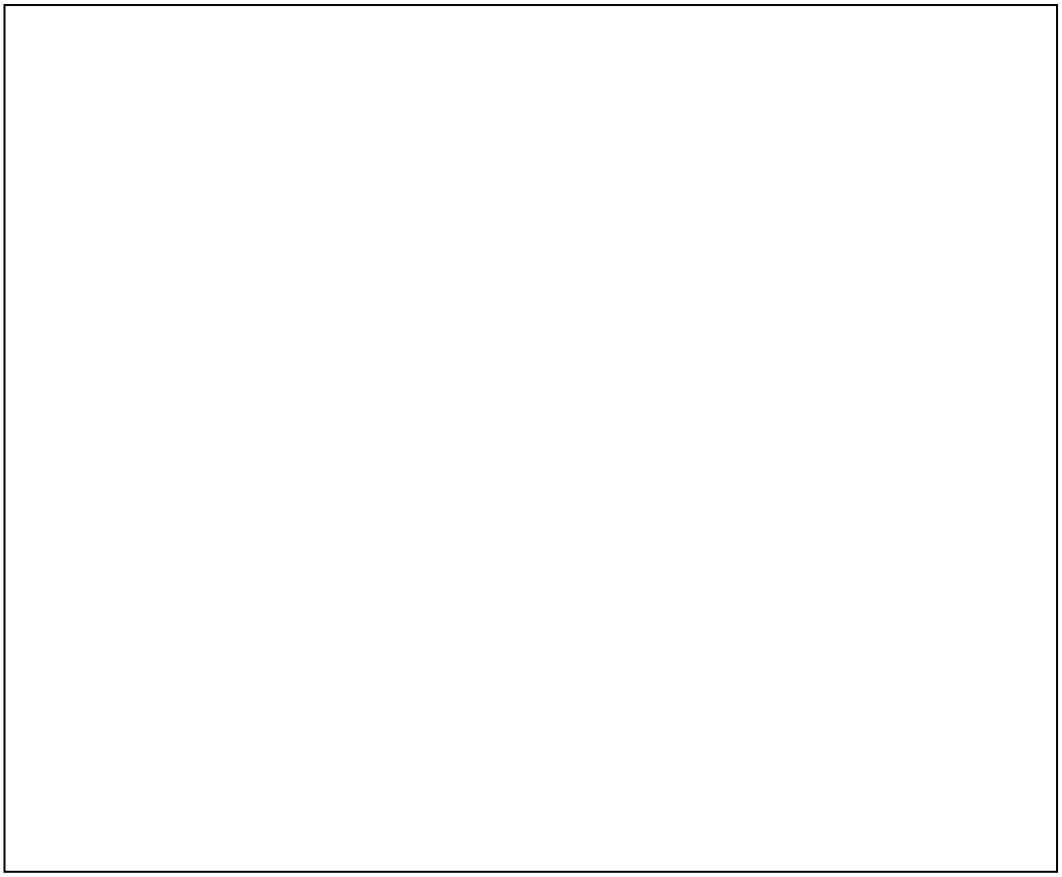
Stage 4: Memory

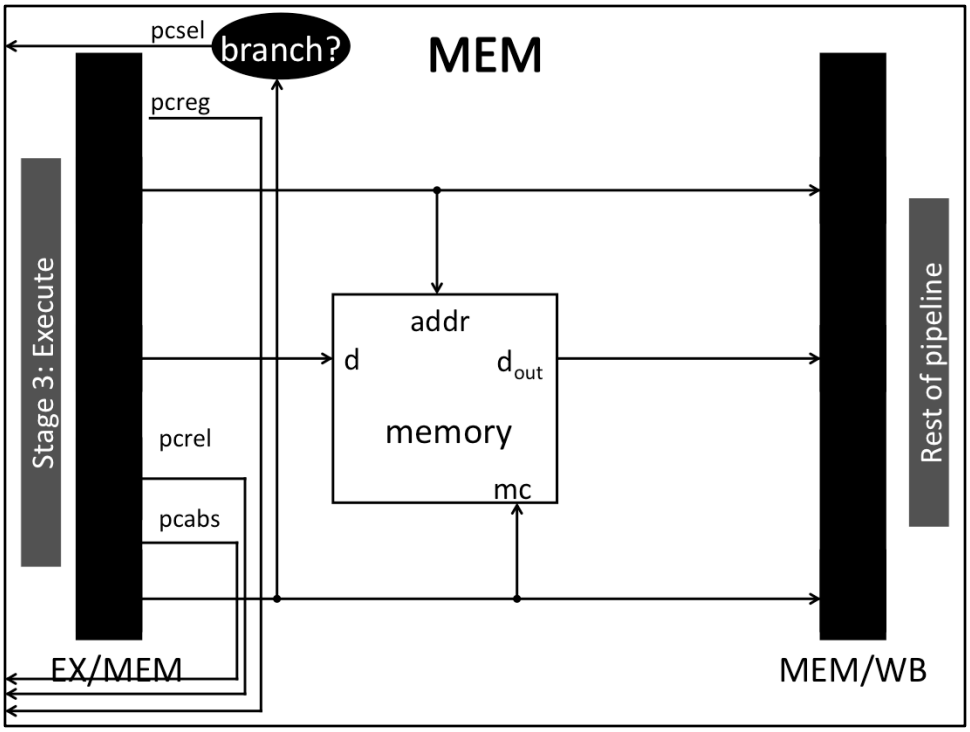
On every cycle:

- Read EX/MEM pipeline register to get values and control bits
- Perform memory load/store if needed
  - address is ALU result

Write values of interest to pipeline register (MEM/WB)

- Control information, Rd index, …
- Result of memory operation
- Pass result of ALU operation



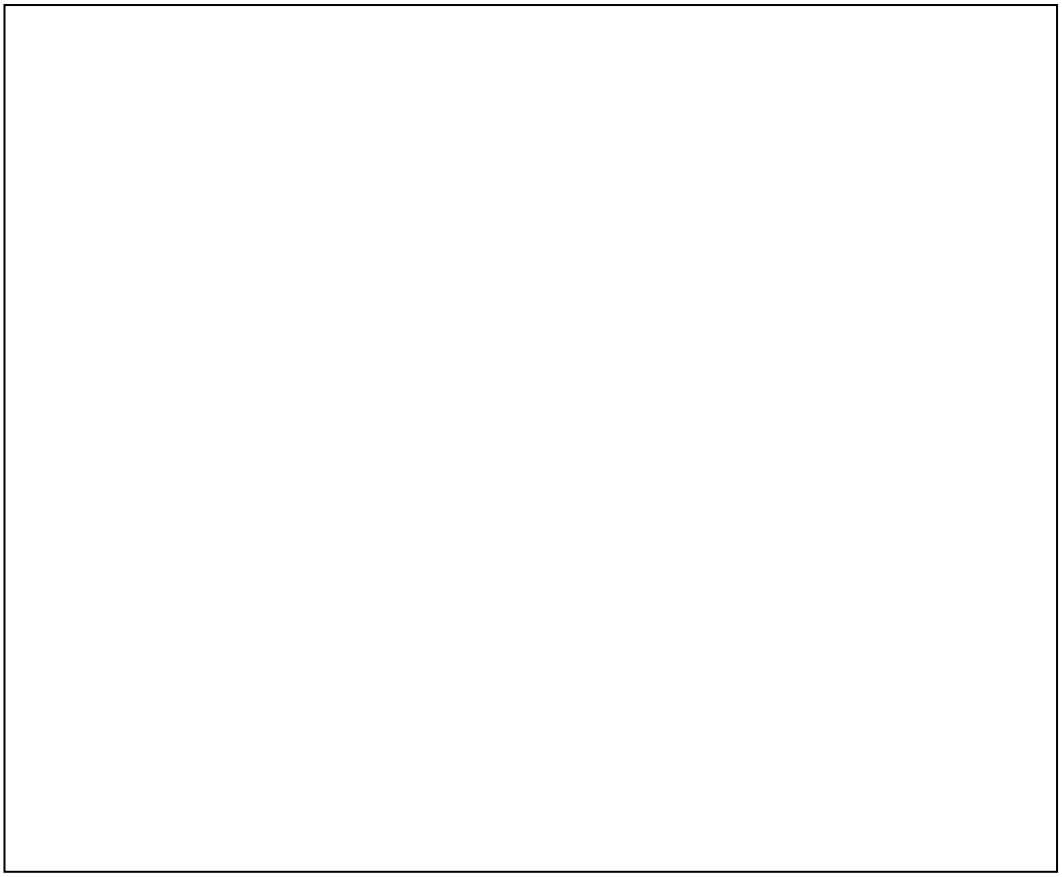

pcsel
branch?
MEM
pcreg
Stage 3: Execute
addr
d
d_{out}
memory
mc
pcrel
pcabs
Rest of pipeline
EX/MEM
MEM/WB

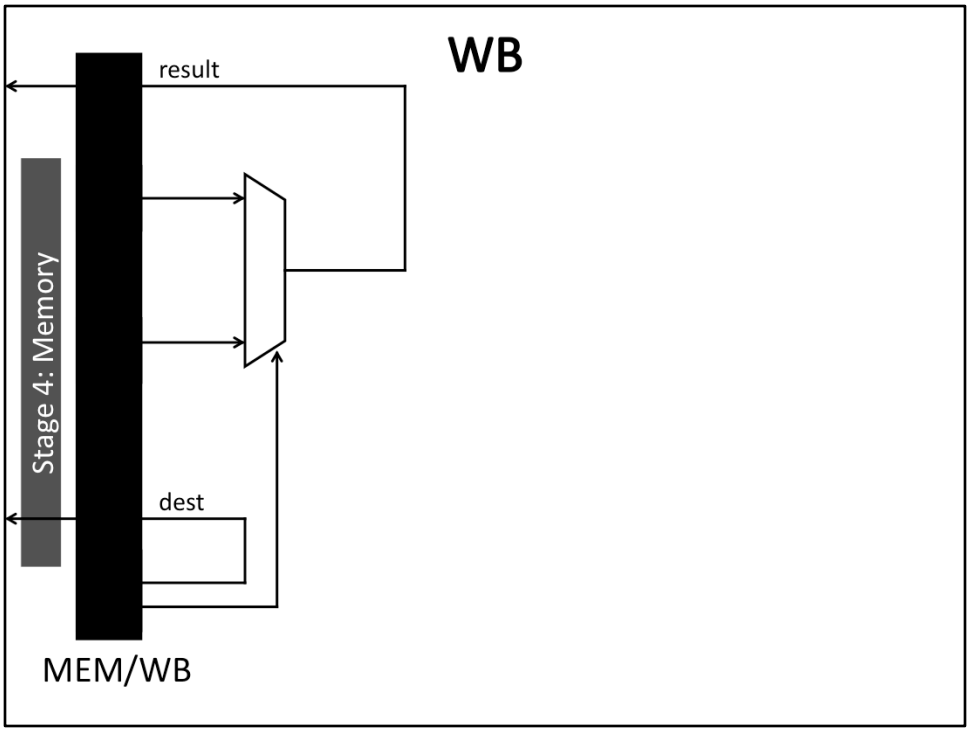
WB

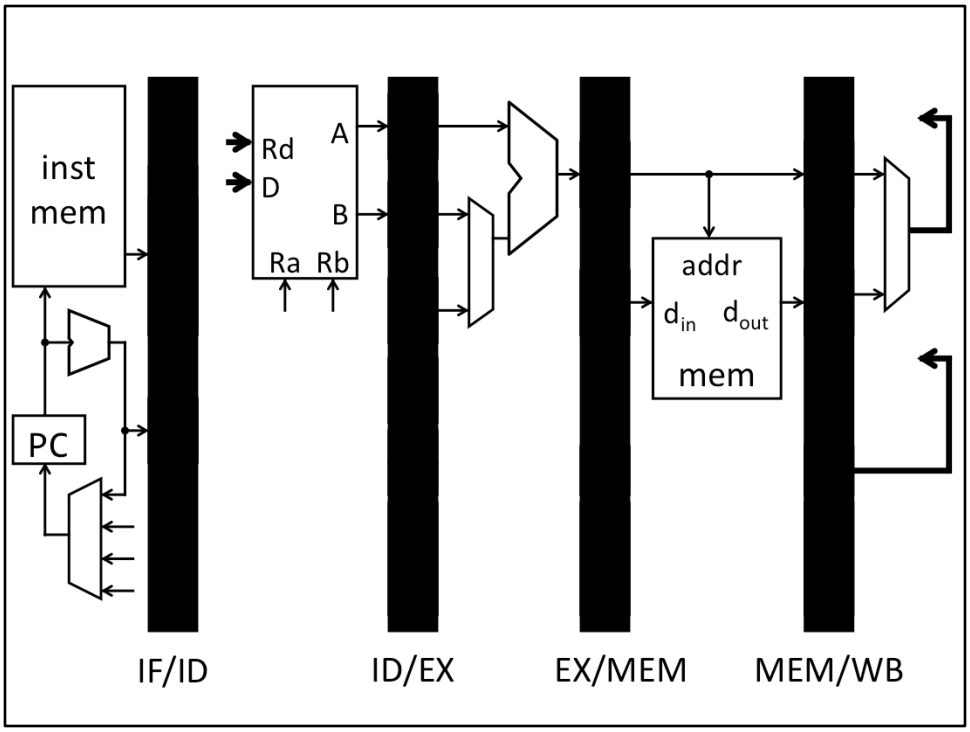
Stage 5: Write-back

On every cycle:

- Read MEM/WB pipeline register to get values and control bits
- Select value and write to register file




WB
result
Stage 4: Memory
dest
MEM/WB


inst
mem
PC
Rd
D
Ra
Rb
A
B
addr
d_{in}
d_{out}
mem
IF/ID
ID/EX
EX/MEM
MEM/WB

Pipelining Recap

Powerful technique for masking latencies

- Logically, instructions execute one at a time
- Physically, instructions execute in parallel
  - Instruction level parallelism

Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)