(Basic) Processor Pipeline

Nima Honarmand

Generic Instruction Life Cycle

- *Logical* steps in processing an instruction:
  - Instruction Fetch (**IF_STEP**)
  - Instruction Decode (**ID_STEP**)
  - Operand Fetch (**OF_STEP**)
    - Might be from registers or memory
  - Execute (**EX_STEP**)
    - Perform computation on the operands
  - Result Store or Write Back (**RS_STEP**)
    - Write the execution results back to registers or memory
- ISA determines what needs to be done in each step for each instruction
- Micro-architecture determines how HW implements steps

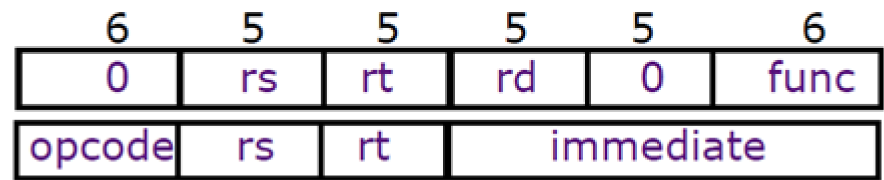
Datapath vs. Control Logic

- ***Datapath*** is the collection of HW components and their connection in a processor
  - Determines the *static* structure of processor
  - E.g., inst/data caches, register file, ALU(s), lots of multiplexers, etc.
- ***Control logic*** determines the dynamic flow of data between the components, e.g.,
  - the control lines of MUXes and ALU
  - read/write controls of caches and register files
  - enable/disable controls of flip-flops
- Micro-architecture = Datapath + control logic

Example: MIPS Instruction Set

- In MIPS, all instructions are 32 bits

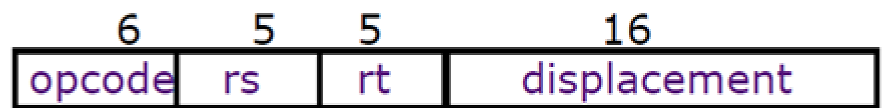
ALU



rd ← (rs) func (rt)

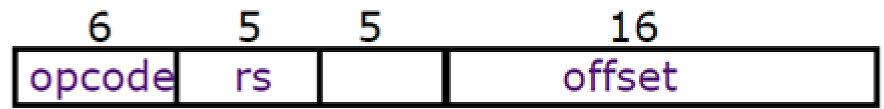
rt ← (rs) op immediate

Mem

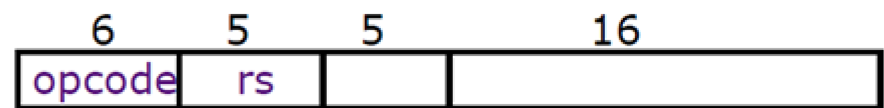


M[(rs) + displacement]

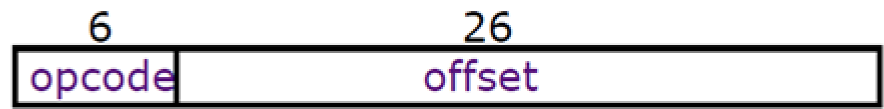
Control Flow



BEQZ, BNEZ

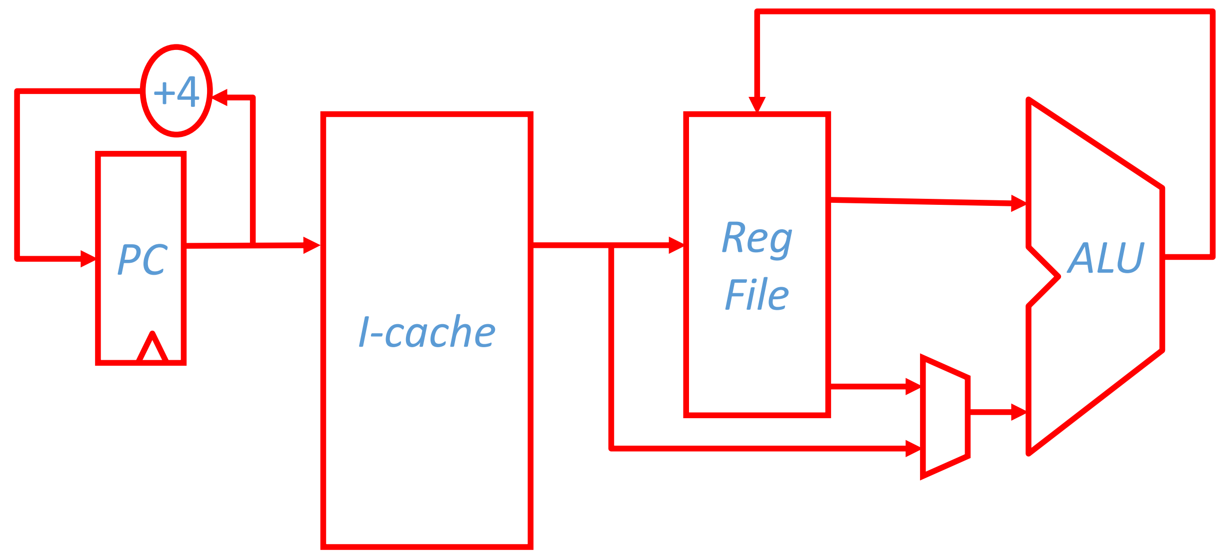


JR, JALR



J, JAL

Building a Simple MIPS Datapath (1)



ALU

6	5	5	5	5	6
0	rs	rt	rd	0	func
opcode	rs	rt	immediate		

rd ← (rs) func (rt)

rt ← (rs) op immediate

Building a Simple MIPS Datapath (2)

PC

+4

I-cache

Reg File

ALU

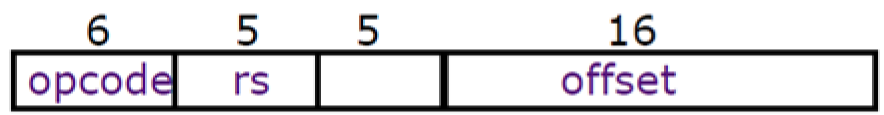
D-cache

Mem

Building a Simple MIPS Datapath (3)

+4
PC
I-cache
Reg File
+
ALU
D-cache

Control Flow



BEQZ, BNEZ

Building a Simple MIPS Datapath (4)

PC

+4

I-cache

Reg File

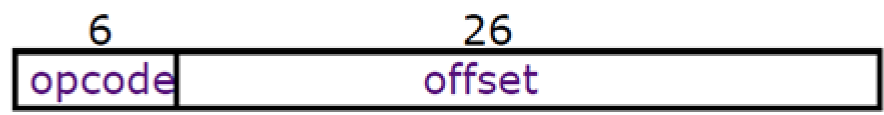
+

ALU

D-cache

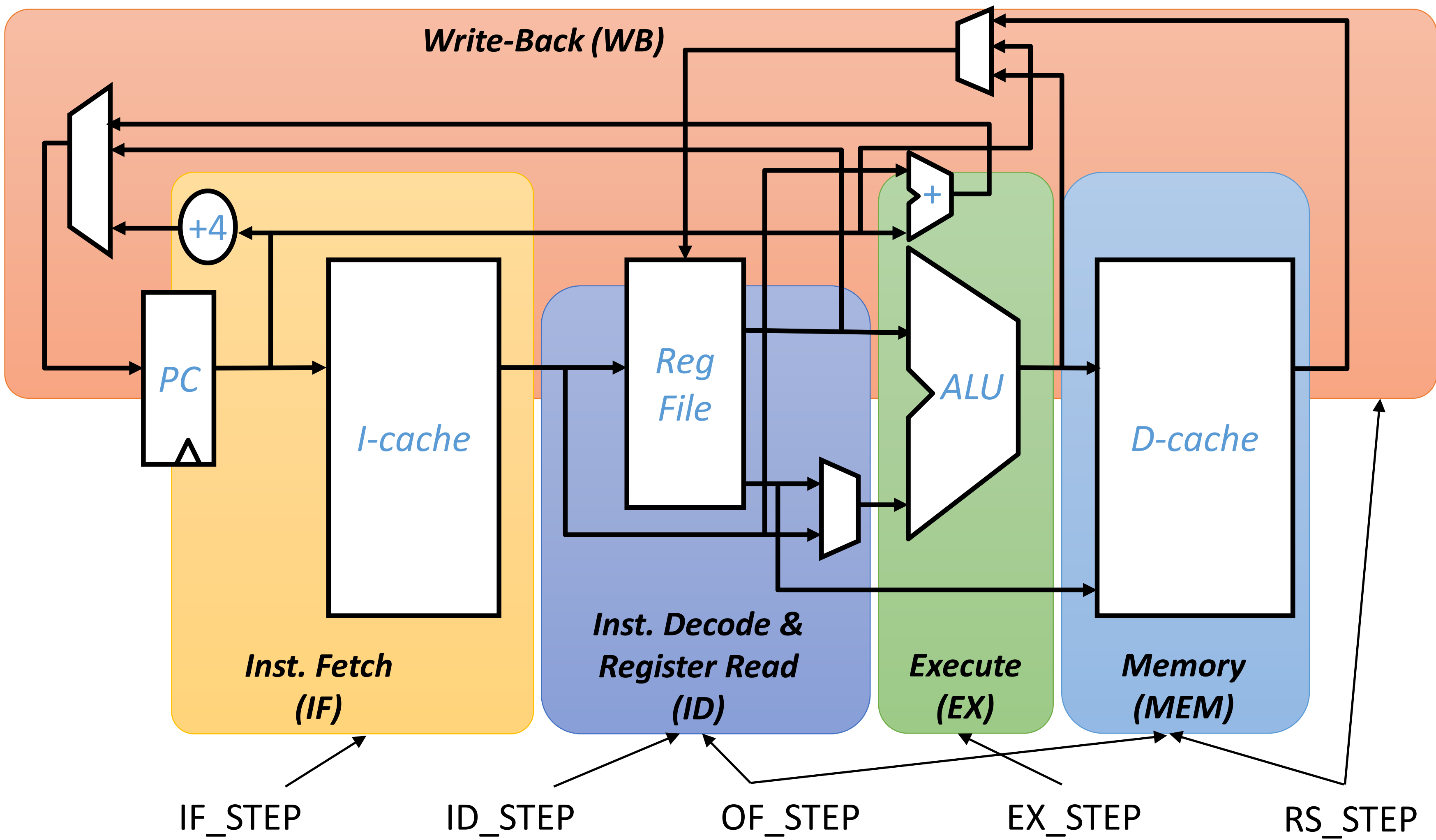
Control Flow

JR, JALR



J, JAL

Our Final MIPS Datapath



Datapath steps need not directly map to logical steps!

What about the Control Logic?

- Datapath is only half the micro-architecture
  - Control logic is the other half
- There are different possibilities for implementing the control logic of our simple MIPS datapath, including
  - Single cycle operation
  - Multi-cycle operation
  - Pipelined operation

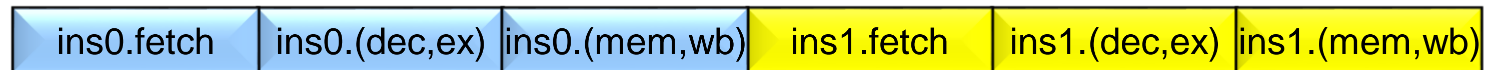
Single Cycle Operation

Single-cycle	ins0.(fetch,dec,ex,mem,wb)	ins1.(fetch,dec,ex,mem,wb)

- Only one instruction is using the datapath at any time
- **Single-cycle** control: all components operate in one, very long, clock cycle
  - At the rising edge of clock, PC gets the new address (new inst); it is the address to I$
  - After some delay, I$ outputs the required word (assuming a hit)
  - After some delay, is decoded and parts of becomes read addresses to register file
  - After some delay, register file outputs the values of the registers
  - After some delay, ALU generates its output and branch-adder generates next inst address; ALU output is the input to D$ (if memory instruction)
  - After some delay, D$ finished its operations (load or store); if load, it generates the output
  - Next inst's cycle: at the rising edge of clock, outputs of ALU or D$ is latched in the register file, and the next-inst address is latched in PC
- This has good IPC (= 1) but very slow clock

Multi-Cycle Operation (1)

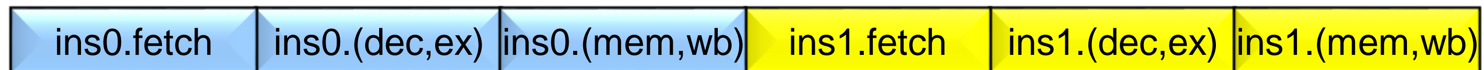
Multi-cycle



- Again, Only one instruction is using datapath at any time
- Perform each subset of the previous steps in a different clock cycle
  - First cycle:
    - At the rising edge of clock, PC gets new value, activates I$;
    - I$ generates the instruction word (assuming a hit)
  - Second cycle:
    - At the rising edge of clock, inst word is latched into a temporary register which becomes input to control logic and register file
    - output of register file is fed to ALU
    - ALU generates its output
    - Branch-adder generates its output

Multi-Cycle Operation (2)

Multi-cycle

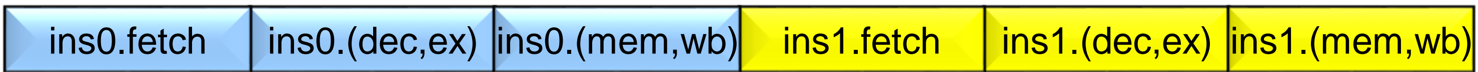


- Third cycle:
  - At the rising edge of clock, ALU output is latched into a temporary register and becomes input to D$
  - D$ performs the operation (assuming a hit)
- Next instruction's first cycle:
  - ALU or D$ output is stored in register file
  - Next-inst address is latched into PC

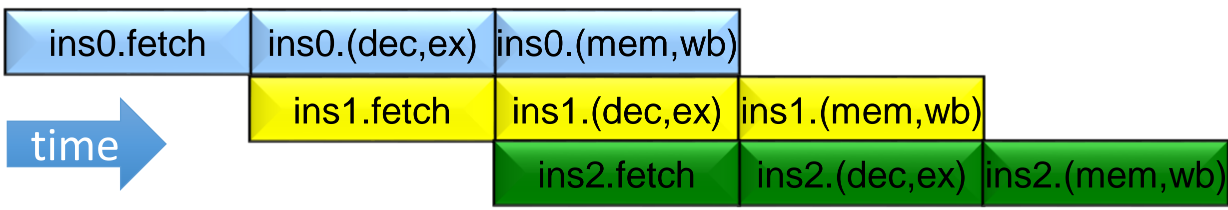
- This has bad IPC (= 0.33) but faster clock
- Can we have both low IPC and short clock period?
  - Yes, through pipelining

Pipelined Operation

Multi-cycle



Pipelined



- Start with multi-cycle design
- When insn0 goes from stage 1 to stage 2, insn1 starts stage 1
- Doable as long as different stages use distinct resources
  - This is the case in our datapath
- Each instruction passes through all stages, but instructions enter and leave at faster ***rate***

Style	Ideal IPC	Cycle Time (1/freq)
Single-cycle	1	Long
Multi-cycle	< 1	Short
Pipelined	**1**	**Short**

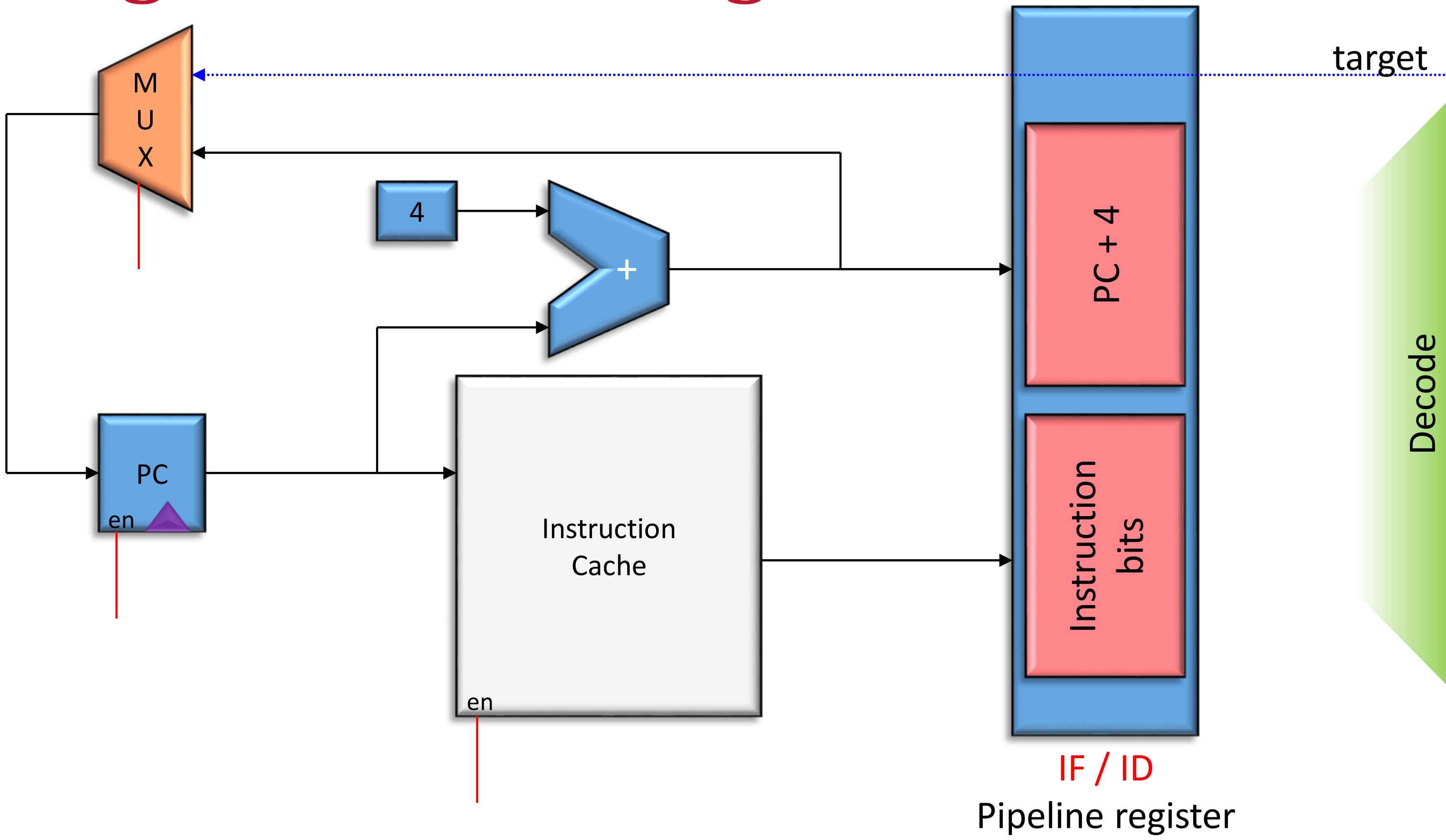
Pipeline can have as many insns ***in flight*** as there are stages

5-Stage MIPS Pipelined Datapath

Stage 1: Fetch

- Fetch an instruction from instruction cache every cycle
  - Use PC to index instruction cache
  - Increment PC (assume no branches for now)
- Write state to the pipeline register **IF/ID**
  - The next stage will read this pipeline register

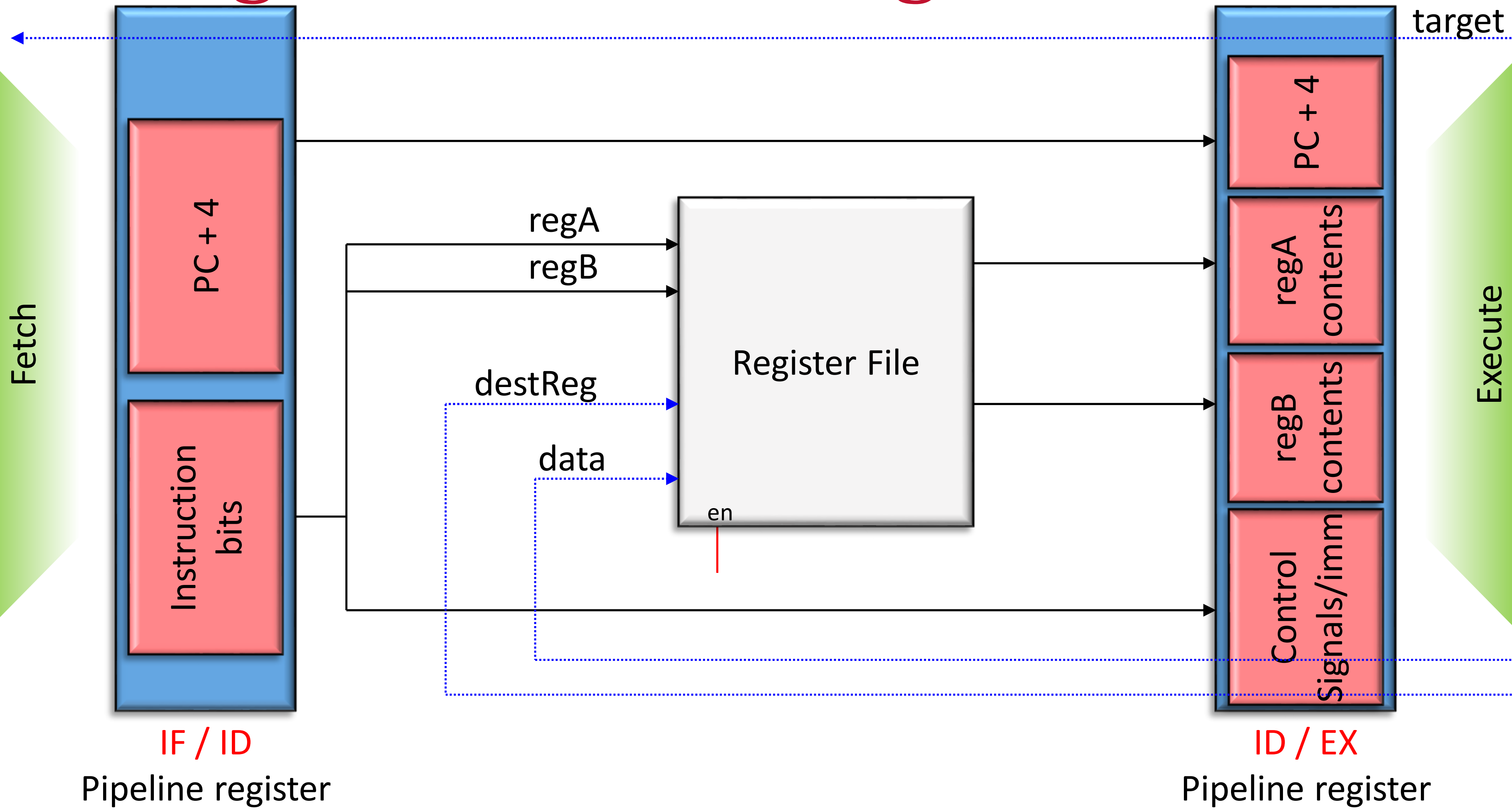
Stage 1: Fetch Diagram



Stage 2: Decode

- Decodes opcode bits
  - Set up Control signals for later stages
- Read input operands from register file
  - Specified by decoded instruction bits
- Write state to the pipeline register **ID/EX**
  - Opcode
  - Register contents, immediate operand
  - PC+4 (even though decode didn't use it)
  - Control signals (from insn) for opcode and destReg

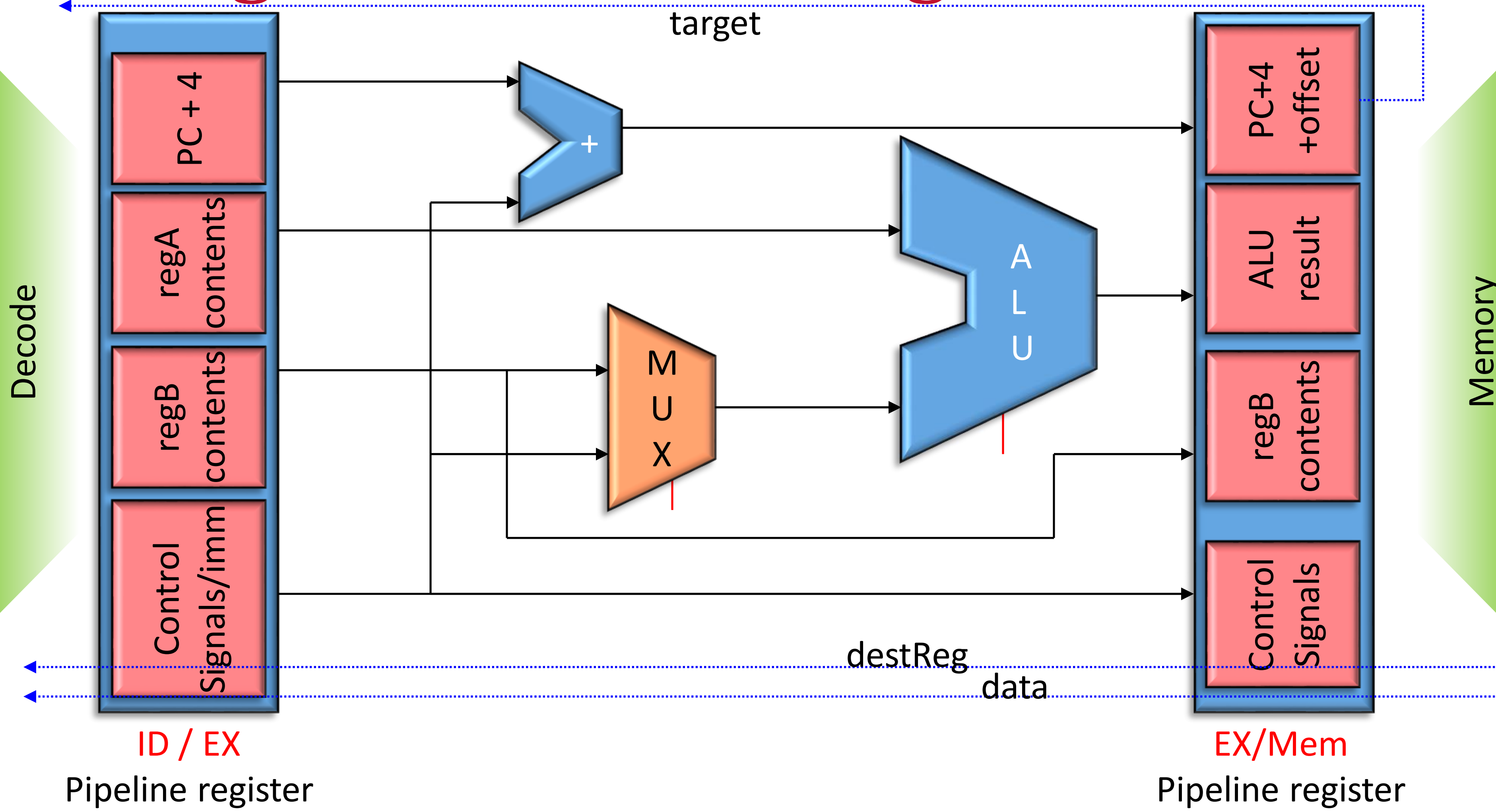
Stage 2: Decode Diagram



Stage 3: Execute

- Perform ALU operations
  - Calculate result of instruction
    - Control signals select operation
    - Contents of regA used as one input
    - Either regB or constant offset (imm from insn) used as second input
  - Calculate PC-relative branch target
    - PC+4+(constant offset)
- Write state to the pipeline register **EX/Mem**
  - ALU result, contents of regB, and PC+4+offset
  - Control signals (from insn) for opcode and destReg

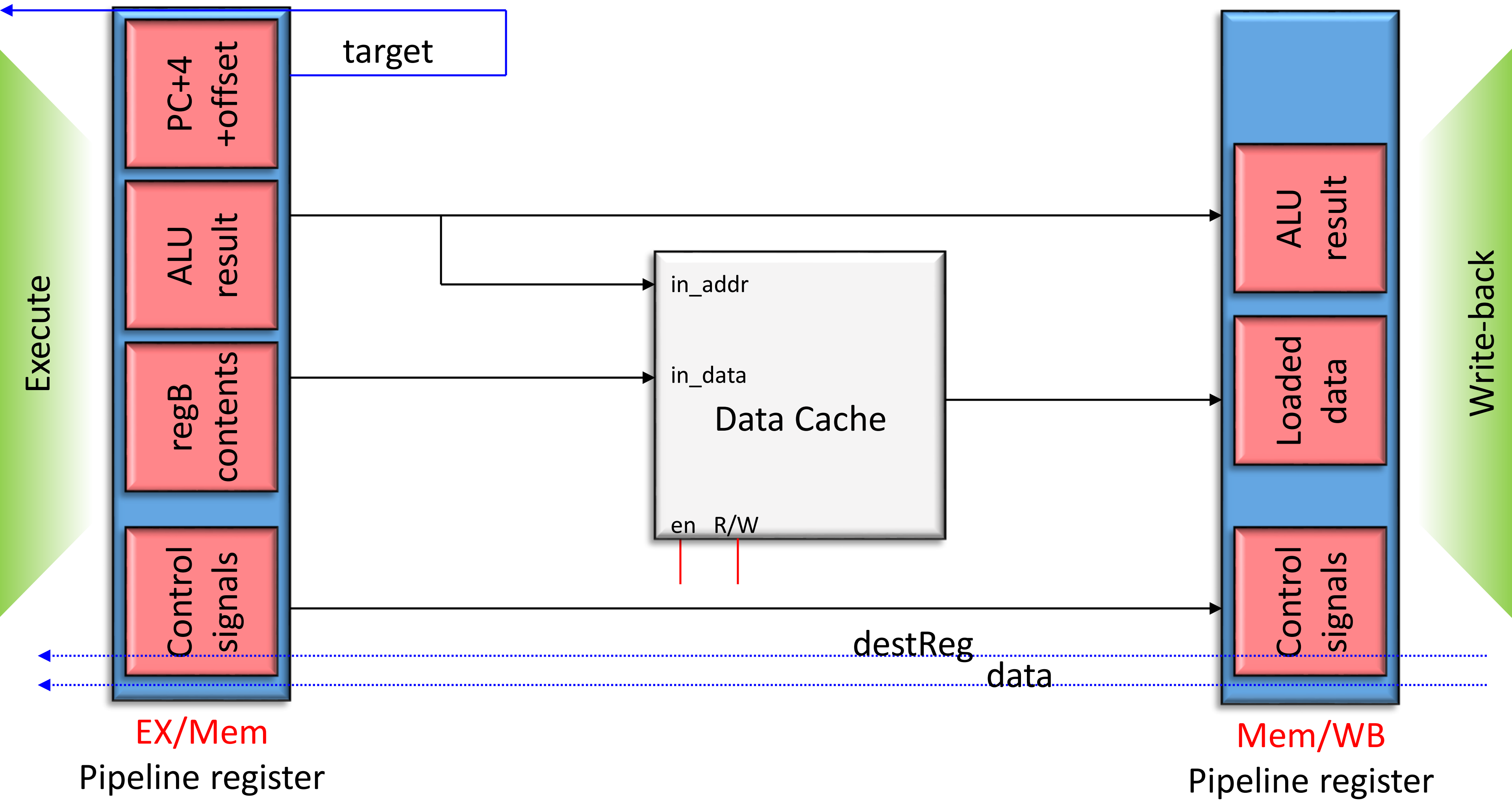
Stage 3: Execute Diagram



Stage 4: Memory

- Perform data cache access
  - ALU result contains address for LD or ST
  - Opcode bits control R/W and enable signals
- Write state to the pipeline register **Mem/WB**
  - ALU result and Loaded data
  - Control signals (from insn) for opcode and destReg

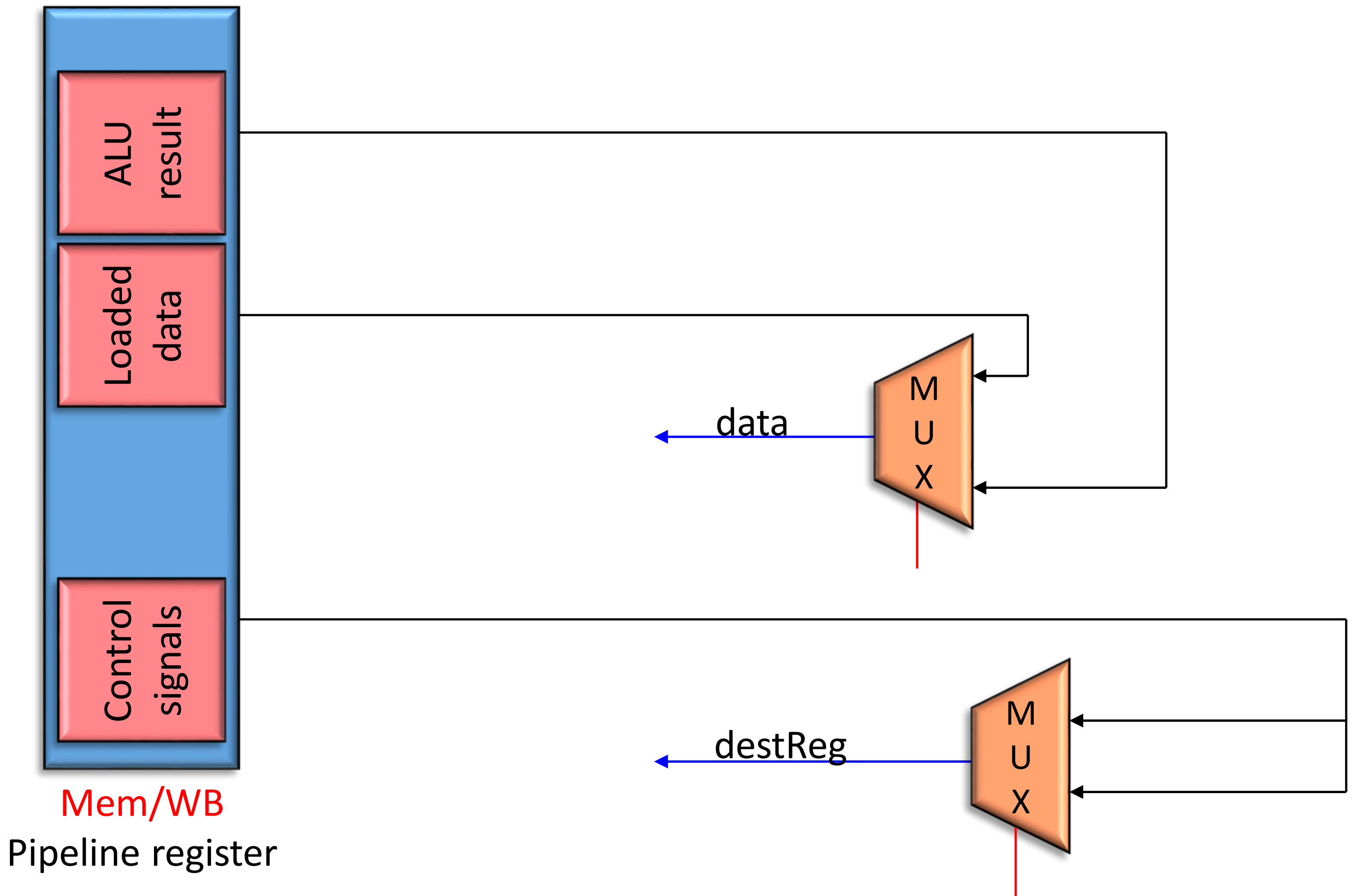
Stage 4: Memory Diagram



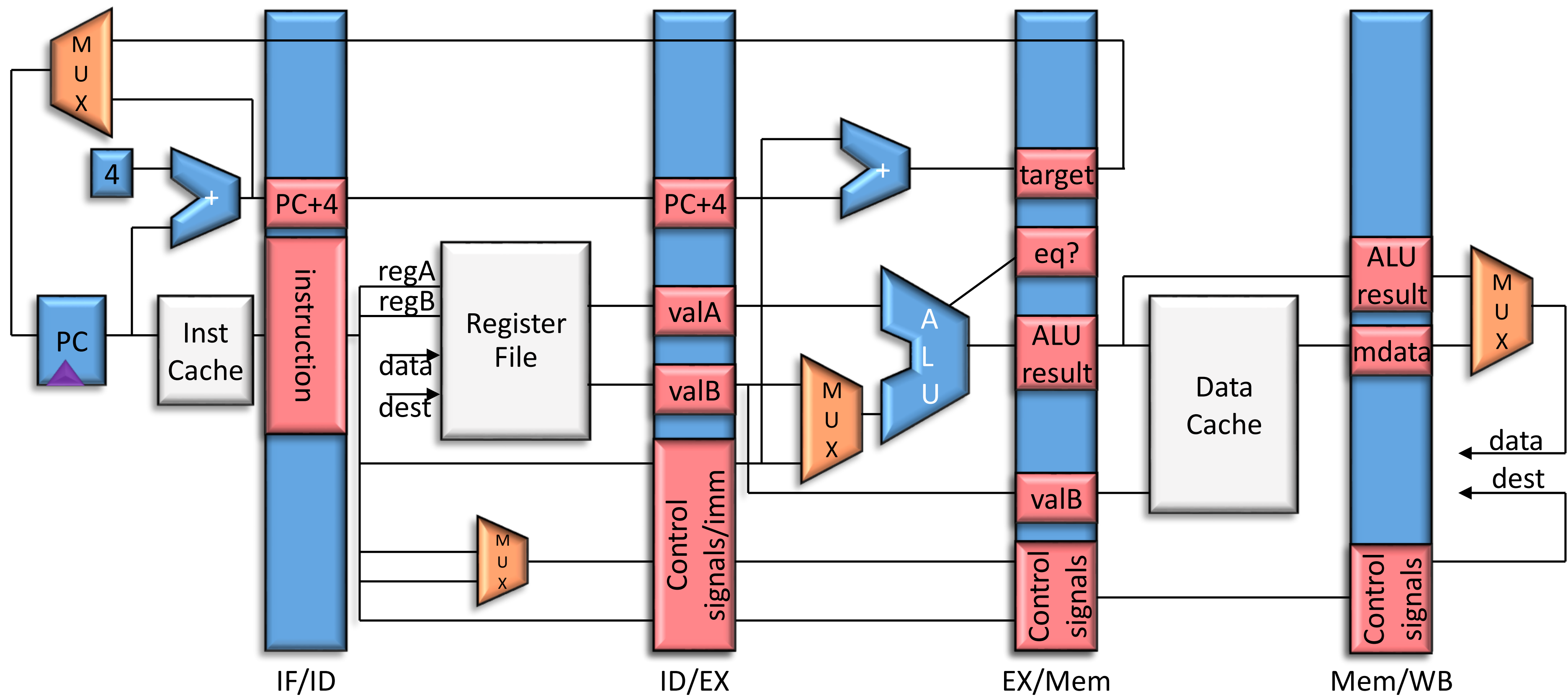
Stage 5: Write-back

- Writing result to register file (if required)
  - Write Loaded data to destReg for LD
  - Write ALU result to destReg for ALU insn
  - Opcode bits control register write enable signal

Stage 5: Write-back Diagram



Putting It All Together



Pipelining Issues

Pipeline Hazards

- A **pipeline hazard** is any condition that disrupts the normal flow of instructions in the pipeline
- Three types of pipeline hazards
  1) **Structural hazards**: required resource is busy
  2) **Data hazards**: need to wait for previous instruction to complete its data read/write
  3) **Control hazards**: deciding on control flow depends on previous instruction

Structural Hazard (1)

- Conflict for use of a resource
  - When multiple instructions need the same resource at the same time
- E.g., in MIPS pipeline with a single cache
  - Load/store requires data access
  - Instruction fetch would have to stall for that cycle
- Hence, pipelined datapaths require separate instruction/data caches to avoid this structural hazard

Structural Hazard (2)

- Another example: if the register file could only do either read or write (but not both) in one cycle
  - ID and WB stages would conflict
- Solution: allow reads and writes in same cycle
- E.g., perform the write at rising edge of the clock and the read at the falling edge
- Why not the other way around?
  - Because, in our MIPS pipeline, reads come from younger instructions and writes older inst.
  - If they both access the same register, younger inst. should read the result of the older inst.

Instruction Dependencies (1)

- Instruction dependencies are root causes of data and control hazards

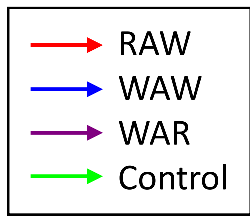
1) Data Dependence
   - **Read-After-Write** (**RAW**) (the only true dependence)
     - Read must wait until earlier write finishes
   - **Anti-Dependence** (**WAR**)
     - Write must wait until earlier read finishes (avoid clobbering)
   - **Output Dependence** (**WAW**)
     - Earlier write can't overwrite later write

2) Control Dependence (a.k.a., Procedural Dependence)
   - Branch condition and target address must be known before future instructions can be executed

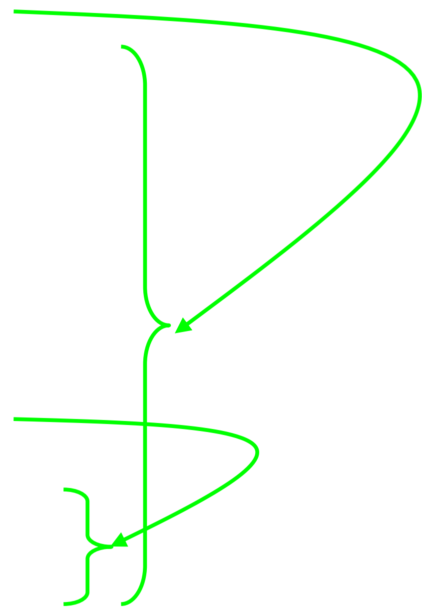
Instruction Dependencies (2)

From Quicksort:

```
for (; (j < high) && (array[j] < array[low]); ++j);
```



	bge	j,	high,	L_2
	mul	$15,	j,	4
	addu	$24,	array,	$15
	lw	$25,	0($24)	
	mul	$13,	low,	4
	addu	$14,	array,	$13
	lw	$15,	0($14)	
	bge	$25,	$15,	L_2
L_1:				
	addu	j,	j,	1
	. . .			
L_2:				
	addu	$11,	$11,	-1
	. . .			



Real code has lots of dependencies

Hardware Dependency Analysis

- Pipeline must handle
  - Register Data Dependencies (same register)
    - RAW, WAW, WAR
  - Memory Data Dependencies (same/overlapping locations)
    - RAW, WAW, WAR
  - Control Dependencies

Pipeline: Steady State

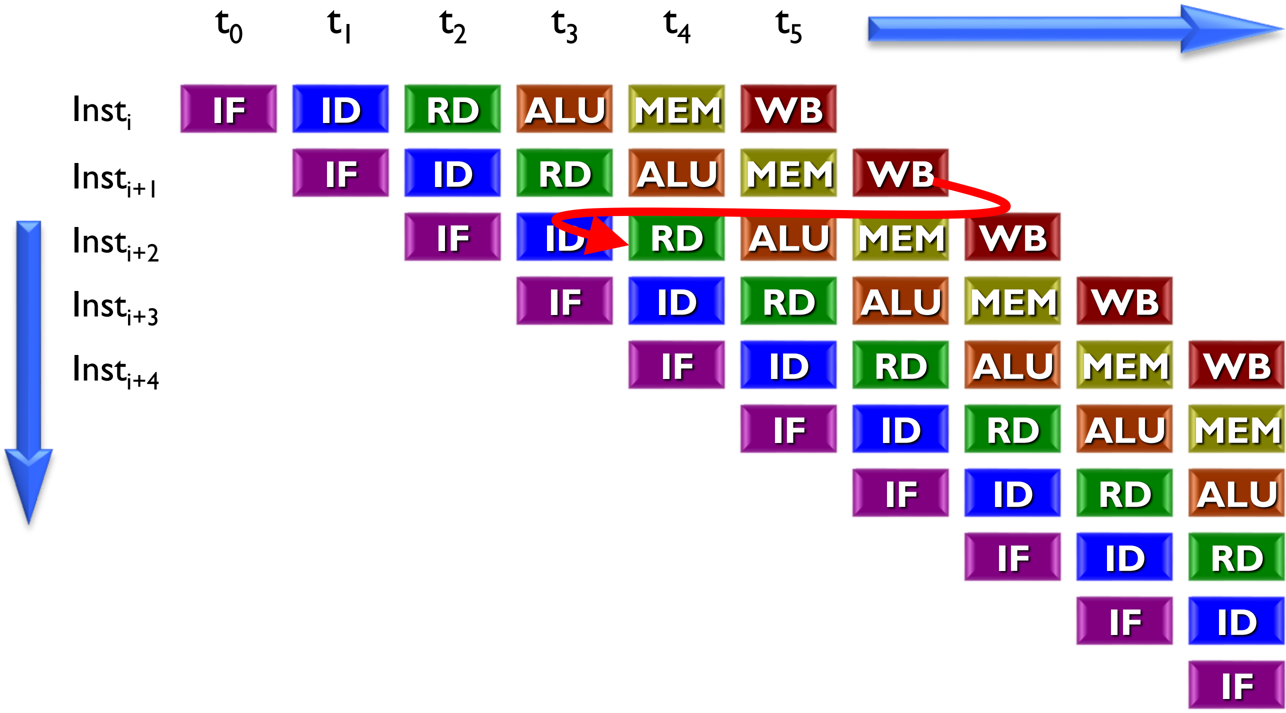
	t_0	t_1	t_2	t_3	t_4	t_5				
$Inst_i$	IF	ID	RD	ALU	MEM	WB				
$Inst_{i+1}$		IF	ID	RD	ALU	MEM	WB			
$Inst_{i+2}$			IF	ID	RD	ALU	MEM	WB		
$Inst_{i+3}$				IF	ID	RD	ALU	MEM	WB	
$Inst_{i+4}$					IF	ID	RD	ALU	MEM	WB
						IF	ID	RD	ALU	MEM
							IF	ID	RD	ALU
								IF	ID	RD
									IF	ID
										IF

Data Hazards

- Caused by data dependencies between instruction
- Necessary conditions in linear pipeline
  - **WAR**: write stage earlier than read stage
    - Is this possible in our pipeline?

  - **WAW**: write stage earlier than write stage
    - Is this possible in our pipeline?
  - **RAW**: read stage earlier than write stage
    - Is this possible in our pipeline?
- If conditions not met, hazards won’t happen
- Check pipeline for both register and memory

Problem: Data Hazard



- Only RAW is possible in our case
  - and only for registers (not memory)

How to Detect Data Hazard (1)

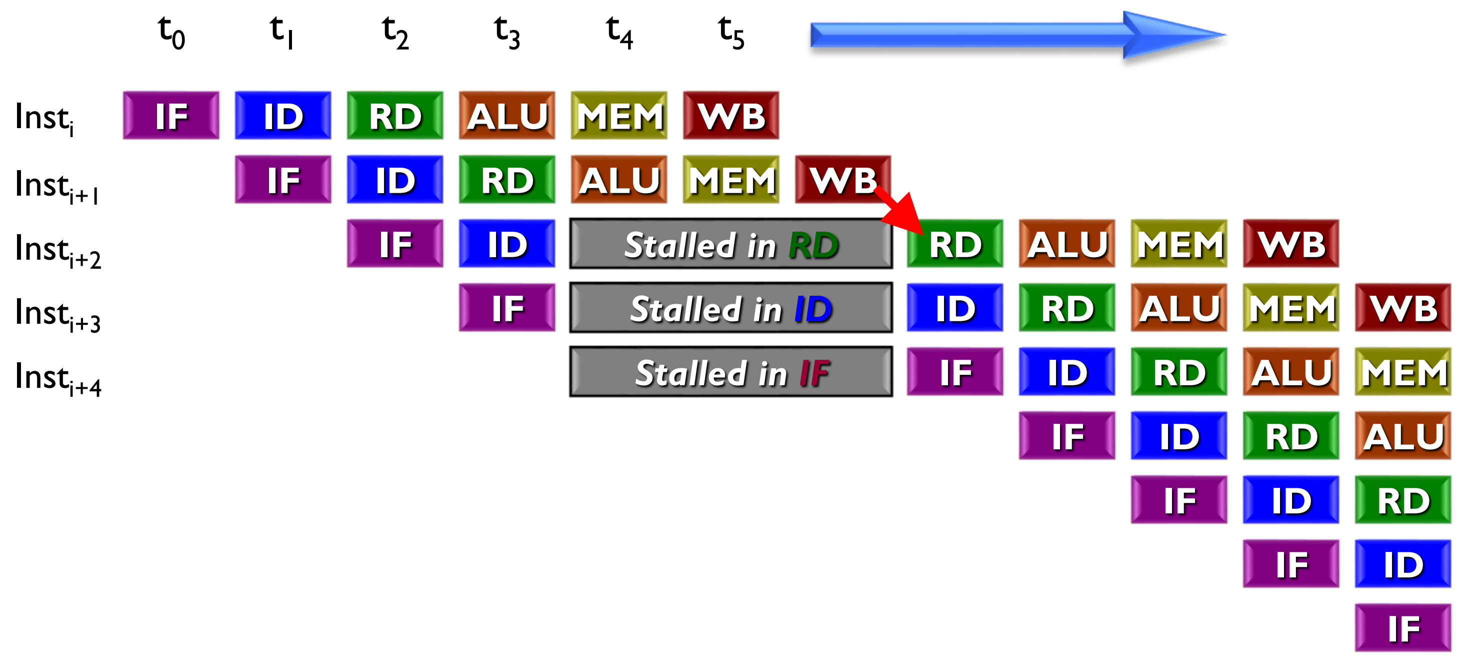
- Compare read-register specifiers for newer instructions with write-register specifiers for older instructions
- E.g., in this 6-stage pipeline, to detect if there is a RAW dependence between inst in RD stage and an older inst:
  - **1a.** ID/RD.RegisterRs == RD/ALU.RegisterRd
  - **1b.** ID/RD.RegisterRt == RD/ALU.RegisterRd
  - **2a.** ID/RD.RegisterRs == ALU/MEM.RegisterRd
  - **2b.** ID/RD.RegisterRt == ALU/MEM.RegisterRd
  - **3a.** ID/RD.RegisterRs == MEM/WB.RegisterRd
  - **3b.** ID/RD.RegisterRt == MEM/WB.RegisterRd

  Dependency to inst in ALU stage (1a, 1b)
  Dependency to inst in MEM stage (2a, 2b)
  Dependency to inst in WB stage (3a, 3b)
- Should also check that the older instruction is going to write to the register. E.g., in case **1**, should also check for
  - RD/ALU.RegWrite && (RD/ALU.RegisterRd != 0)

How to Detect Data Hazard (2)

- If there are multiple dependences with older instructions, determine the “youngest” of the older instruction with which we have a dependency
  - That’s the dependency we should resolve
- In the previous example, inst in ALU is thr youngest of older instructions, so case **1** takes precedence over others

Solution 1: Stall on Data Hazard (1)

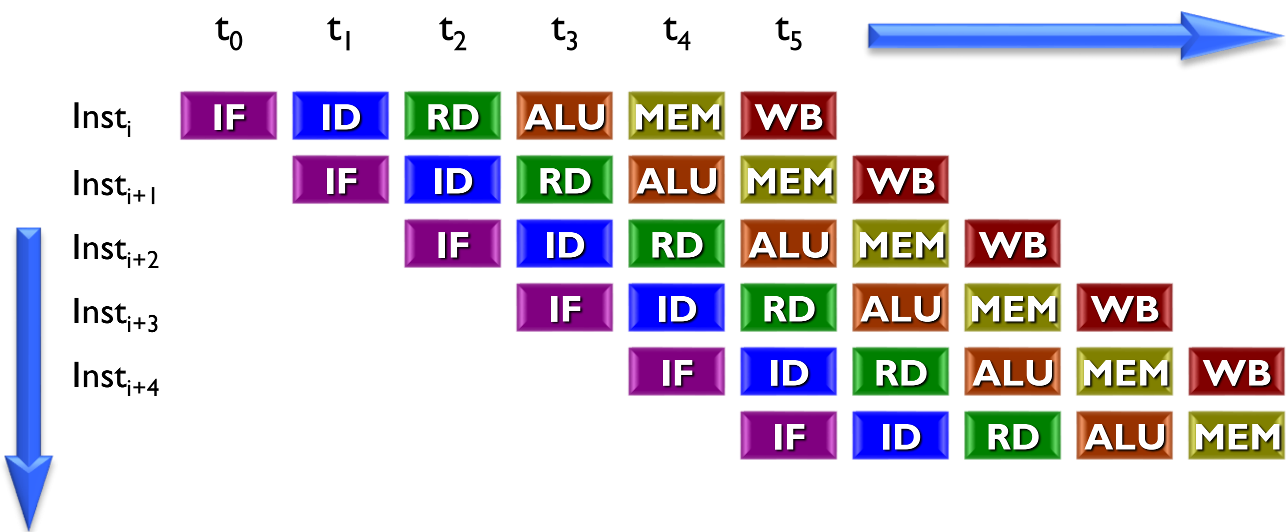


- Dependent instruction moves to RD, and stays there until dependency is resolved
- E.g., if $inst_{i+2}$ depends on $inst_{i+1}$, $inst_{i+2}$ has to stall for 3 cycles
  - So do instructions following $inst_{i+2}$

Solution 1: Stall on Data Hazard (2)

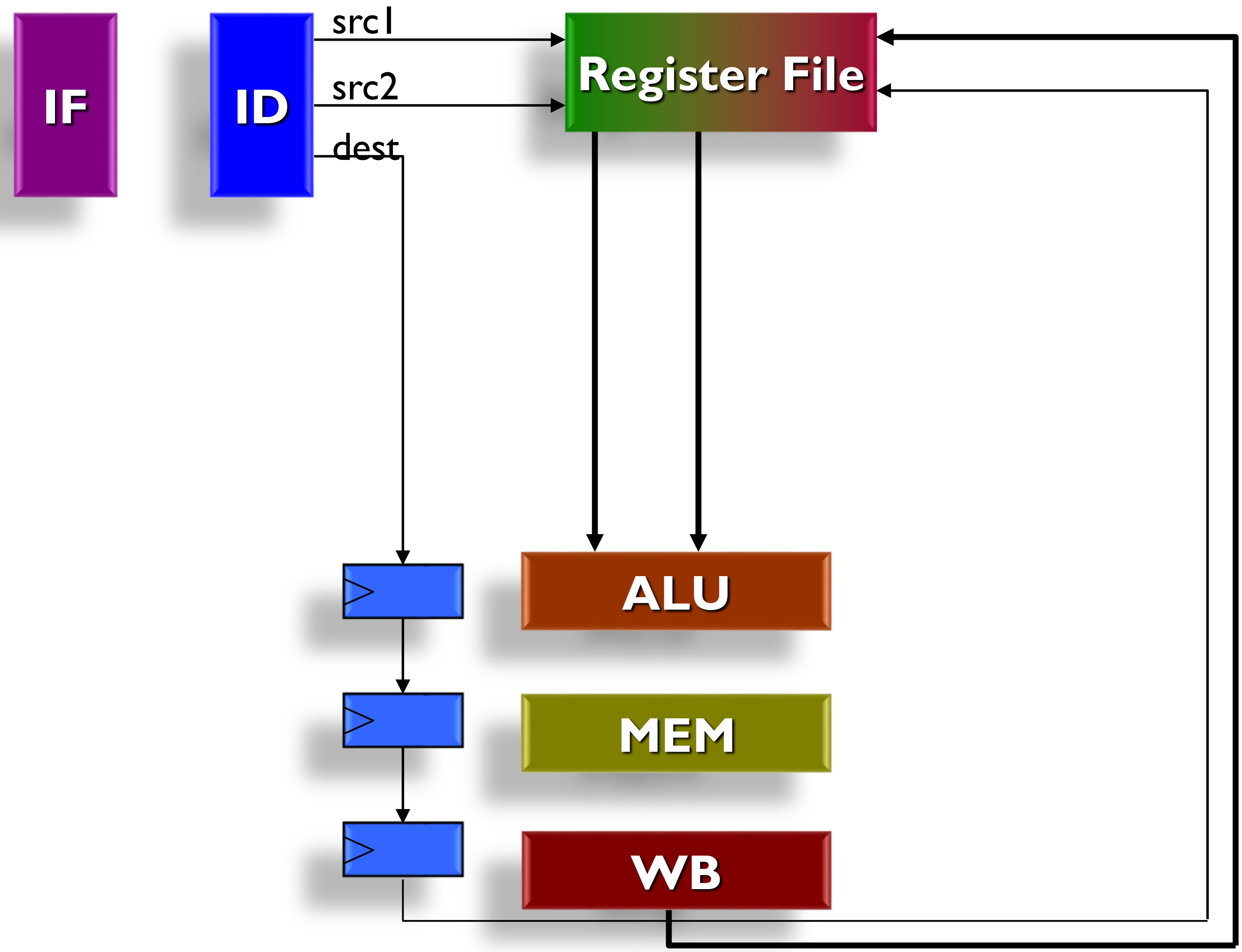
- Instructions in IF, ID and RD stay
  - ID/RD and IF/ID pipeline registers not updated
- For stages after RD, send **no-op** down pipeline (called a **bubble**)
  - **bubble**: state of pipeline registers that would correspond to a no-op instruction occupying that stage

Solution 2: Forwarding Paths (1)

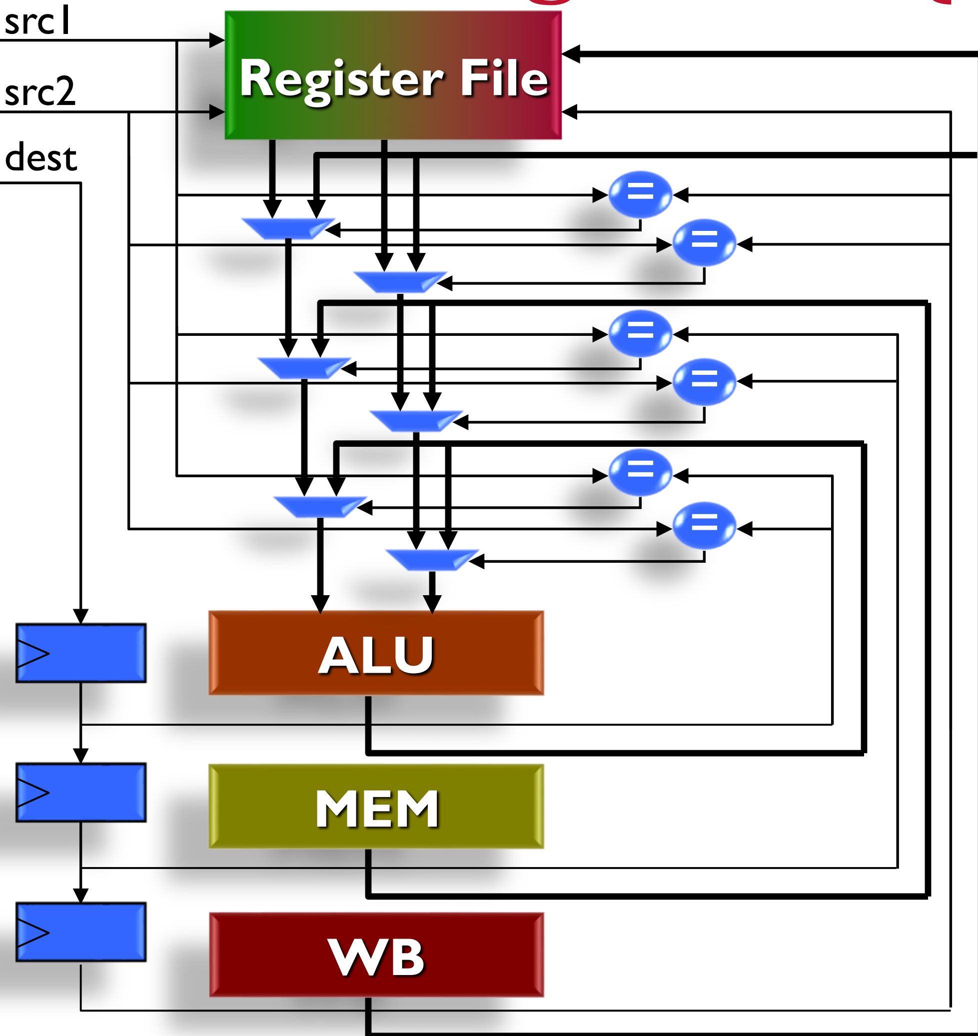


- Idea: avoid stalling by forwarding older inst results to younger ones before they are written to RF.

Solution 2: Forwarding Paths (2)

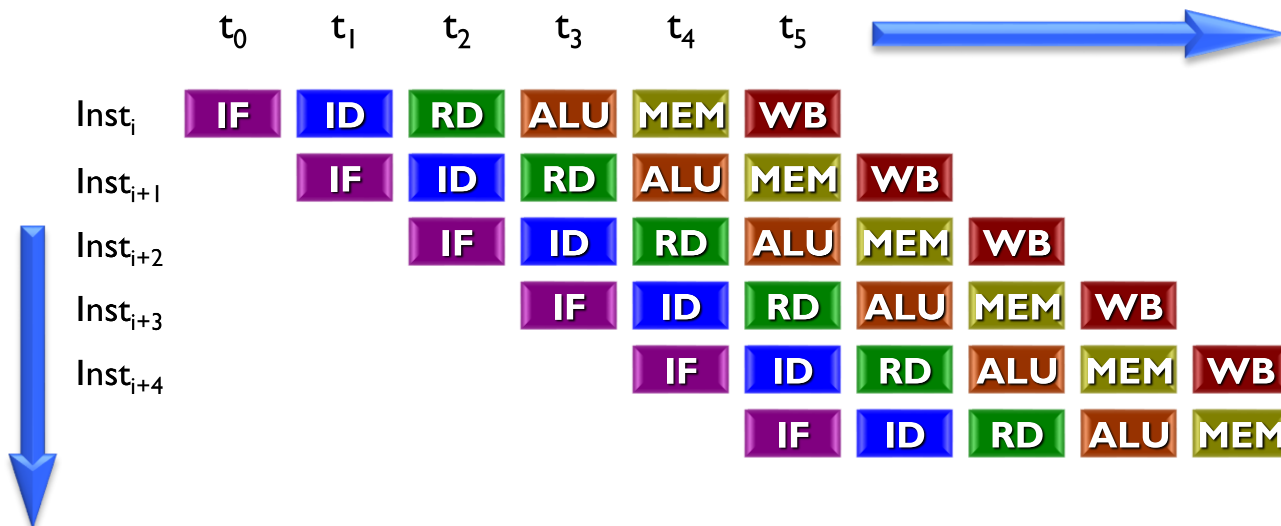


Solution 2: Forwarding Paths (3)



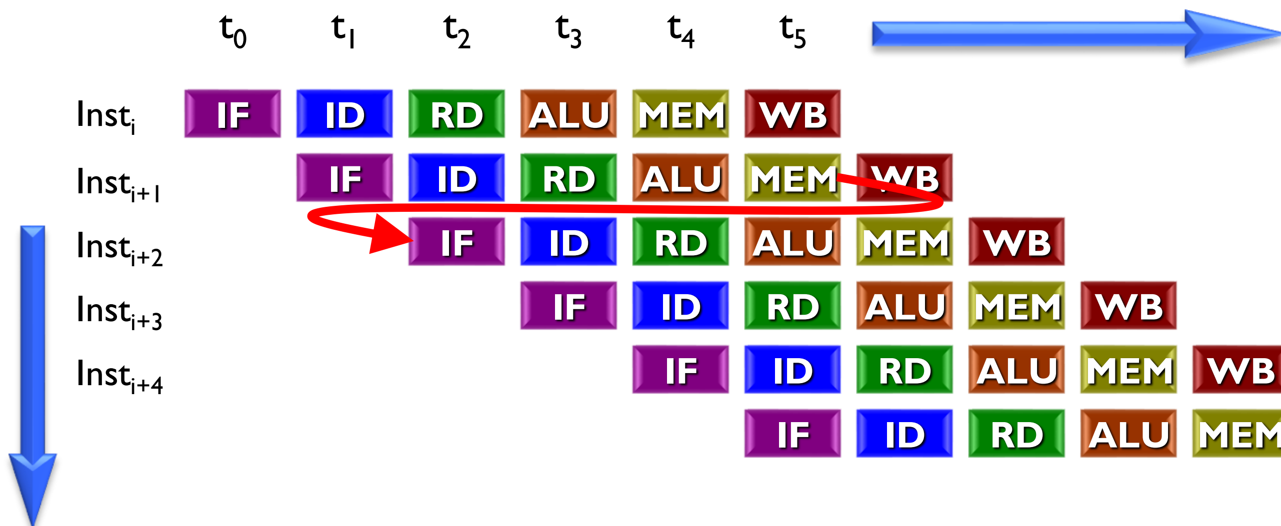
Deeper pipelines in general require additional forwarding paths

Solution 2: Forwarding Paths (4)



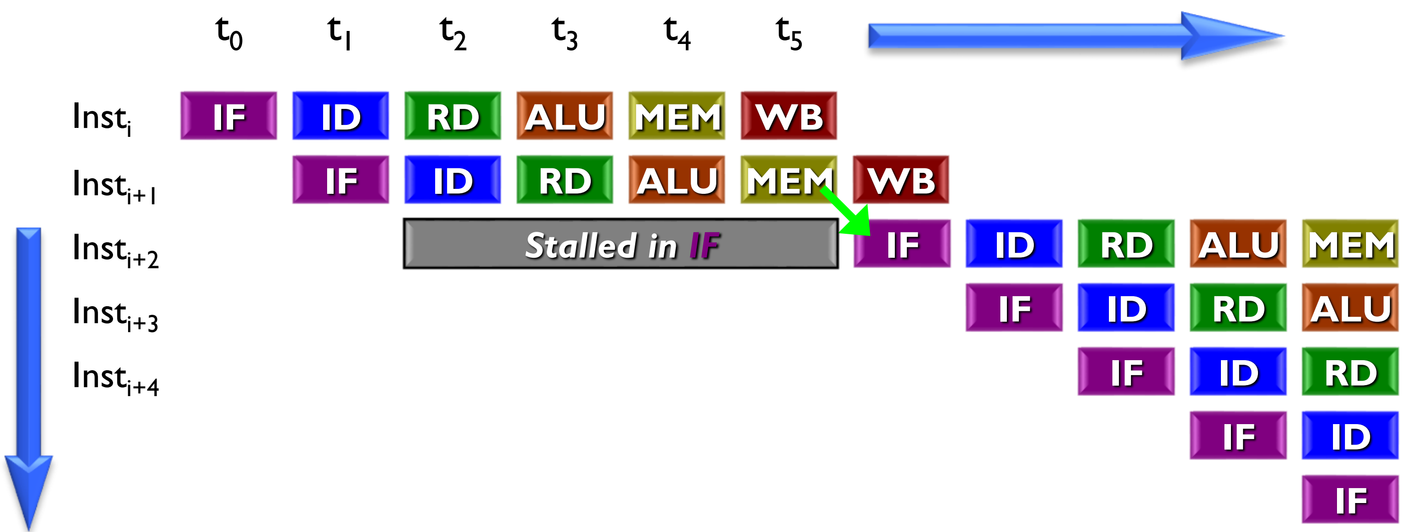
- Sometimes, forwarding is not enough and some stalling is needed
- E.g., if $inst_{i+2}$ depends on $inst_{i+1}$, and $inst_{i+1}$ is a load, $inst_{i+2}$ has to be stalled for at least one cycle until $inst_{i+1}$ accesses the data cache
  - Then, we can forward the result to $inst_{i+2}$

Problem: Control Hazard



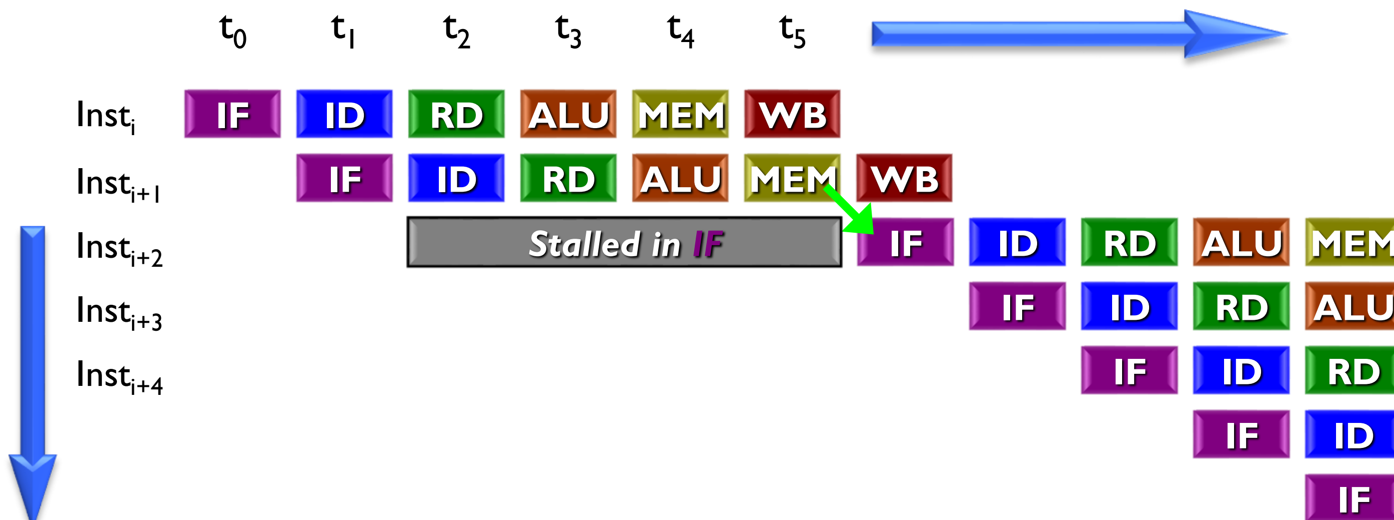
- Assume $inst_{i+1}$ is a branch
- We won't know the address of $inst_{i+2}$ until $inst_{i+1}$ (branch instruction) writes to PC
- Assume the branch outcome and target is calculated at the ALU stage, but is written back to PC during the MEM stage
  - Similar to our 5-stage MIPS pipeline

Solution 1: Stall on Control Hazard



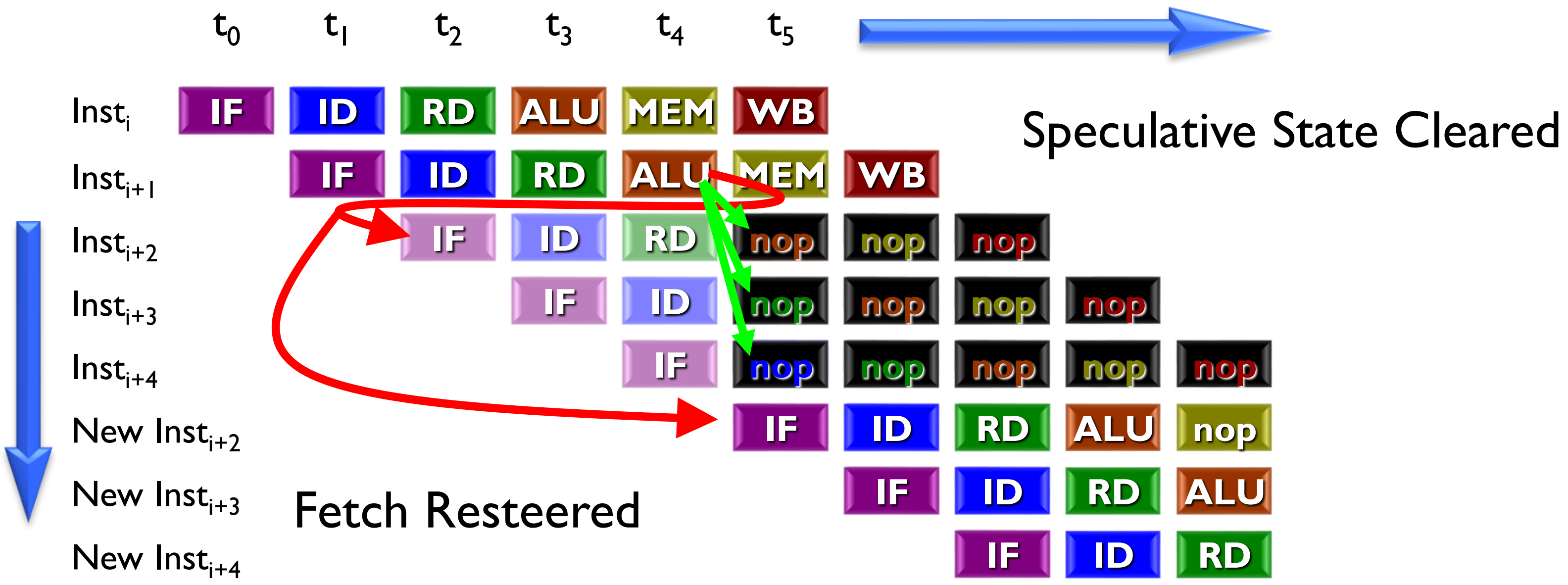
- Stop fetching until branch outcome is known
  - Send no-ops down the pipe
- Easy to implement
  - Requires simple pre-decoding in IF to know if $inst_{i+1}$ is a branch
- Performs poorly
  - On out of ~6 instructions are branches
  - Each branch takes 4 cycles to resolve
  - CPI = 1 + 4 x 1/6 = 1.67 (best case (lower bound))

Solution 1: Stall on Control Hazard



- Stop fetching until branch outcome is known
- Easy to implement
  - Requires simple pre-decoding in IF to know if insti+1 is a branch
  - Send no-ops down the pipe
- Performs poorly
  - 1 out of ~6 instructions are branches
  - Each branch takes 4 cycles to resolve
  - CPI = 1 + 4 x 1/6 = 1.67 (best case (lower bound))

Solution 2: Prediction for Control Hazards



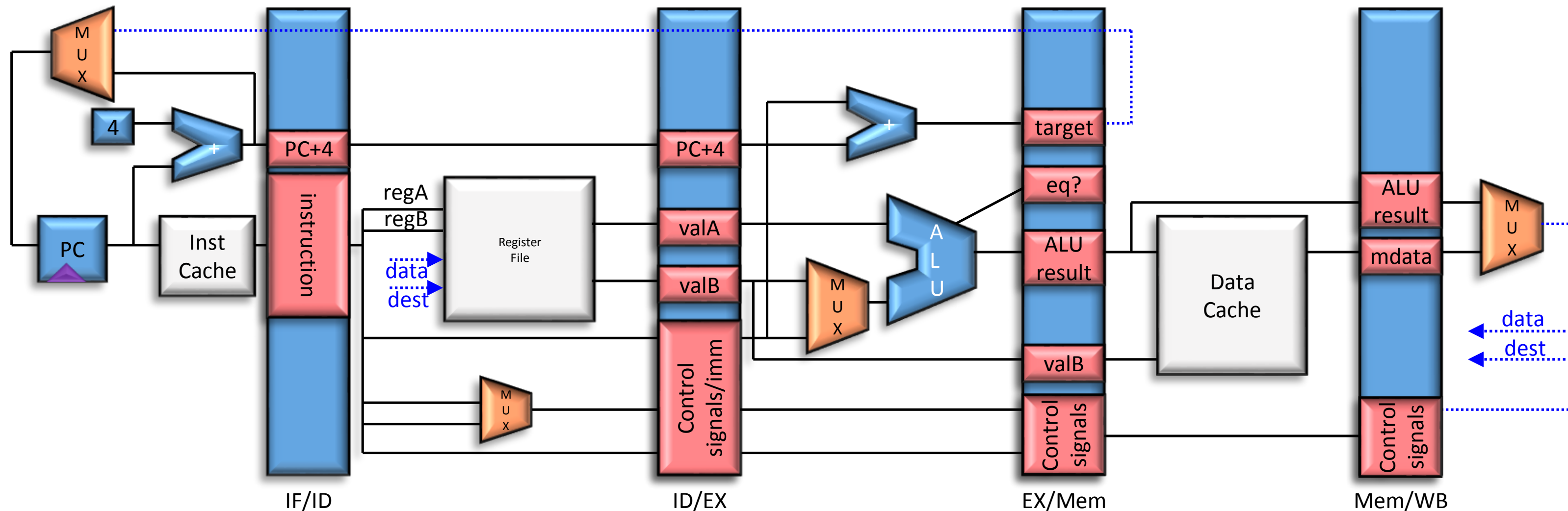
- Predict branch not taken
  - Send sequential instructions down pipeline
- We would know the branch outcome the end of ALU
  - If incorrect prediction, kill “speculative” instructions (turn them into no-ops by setting pipeline registers)
  - Fetch from branch target
- **Important:** “Speculative” instructions cannot perform memory and RF writes
  - No problem in this pipeline
  - Because MEM and WB stages of speculative instructions come after ALU stage of branch

Solution 3: Delay Slots for Control Hazards

- Another option: **delayed branches**
  - # of delay slots (*ds*) : less-than-or-equal-to # stages between IF and where the branch is resolved
    - 3 (IF to ALU) in our example
  - Always execute following *ds* instructions regardless of branch outcome
  - Compiler should put useful instruction there, otherwise no-op insts
- Has lost popularity but lingers for compatibility reasons
  - Just a stopgap (one cycle, one instruction)
  - In superscalar processors, delay slot just gets in the way

Legacy from old RISC ISAs

Hazards & Backward-Going Lines in Pipeline



- In a linear pipeline, all structural, data and control hazards manifest as backward-going lines in the pipeline design
- You can use them to double-check your identification of possible control hazards in your pipeline