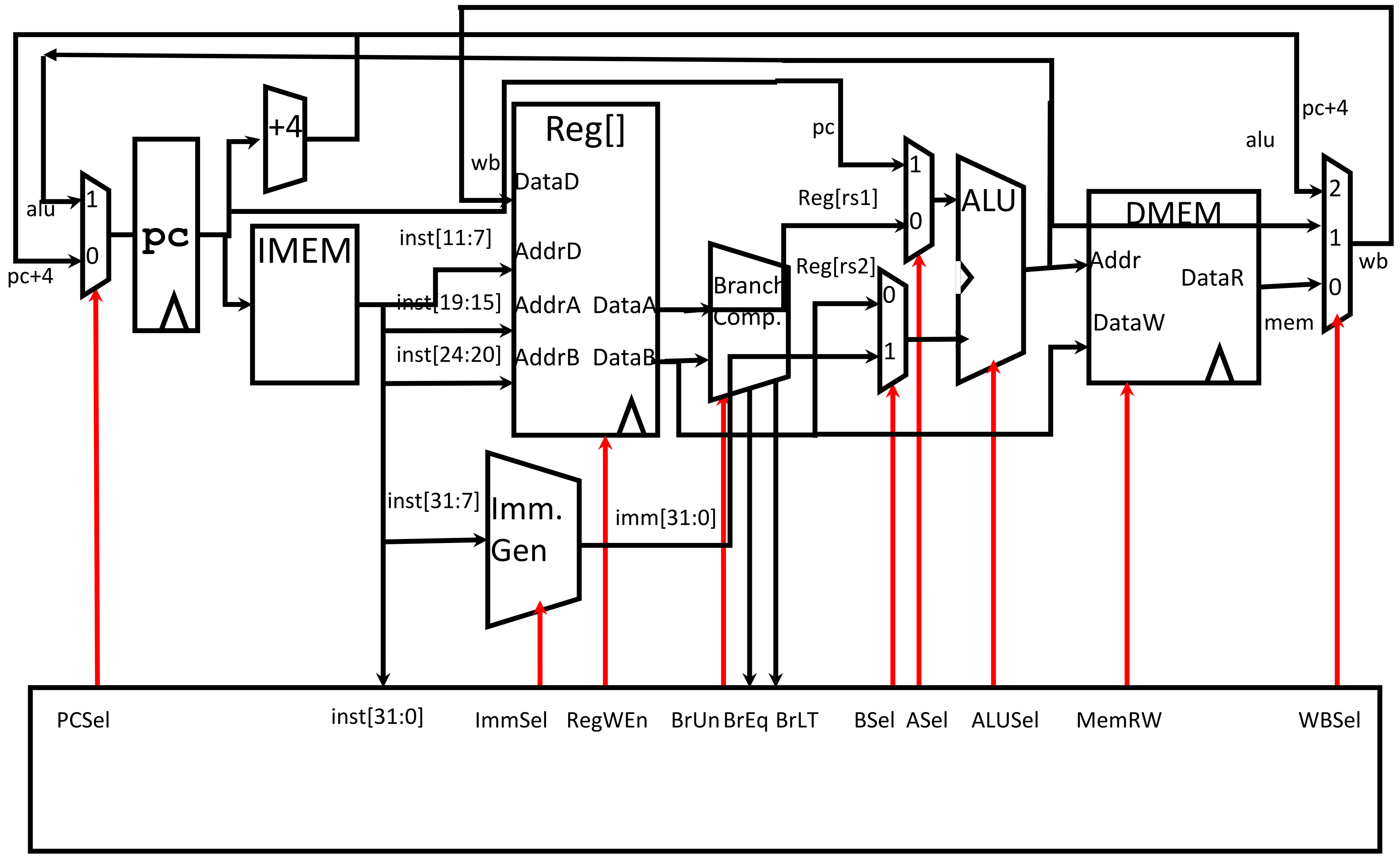
CPU Control
Pipelines and Hazards

CMPT 295 Week 11

Control Signals

- Control signals are how we get the same hardware to behave differently and produce different instructions
- For every instruction, all control signals are set to one of their possible values (Not always 0 or 1!) or an indeterminate (*) value indicating the control signal doesn't affect the instruction's execution
- Each control signal has a sub-circuit based on ~nine bits from the instruction format:
  - Upper 5 func7 bits (lower 2 are the same for all instructions)
  - All func3 bits
  - "2nd" upper opcode bit (others are the same for all instructions)

Control Signals: ADD



ADD: Control Signals

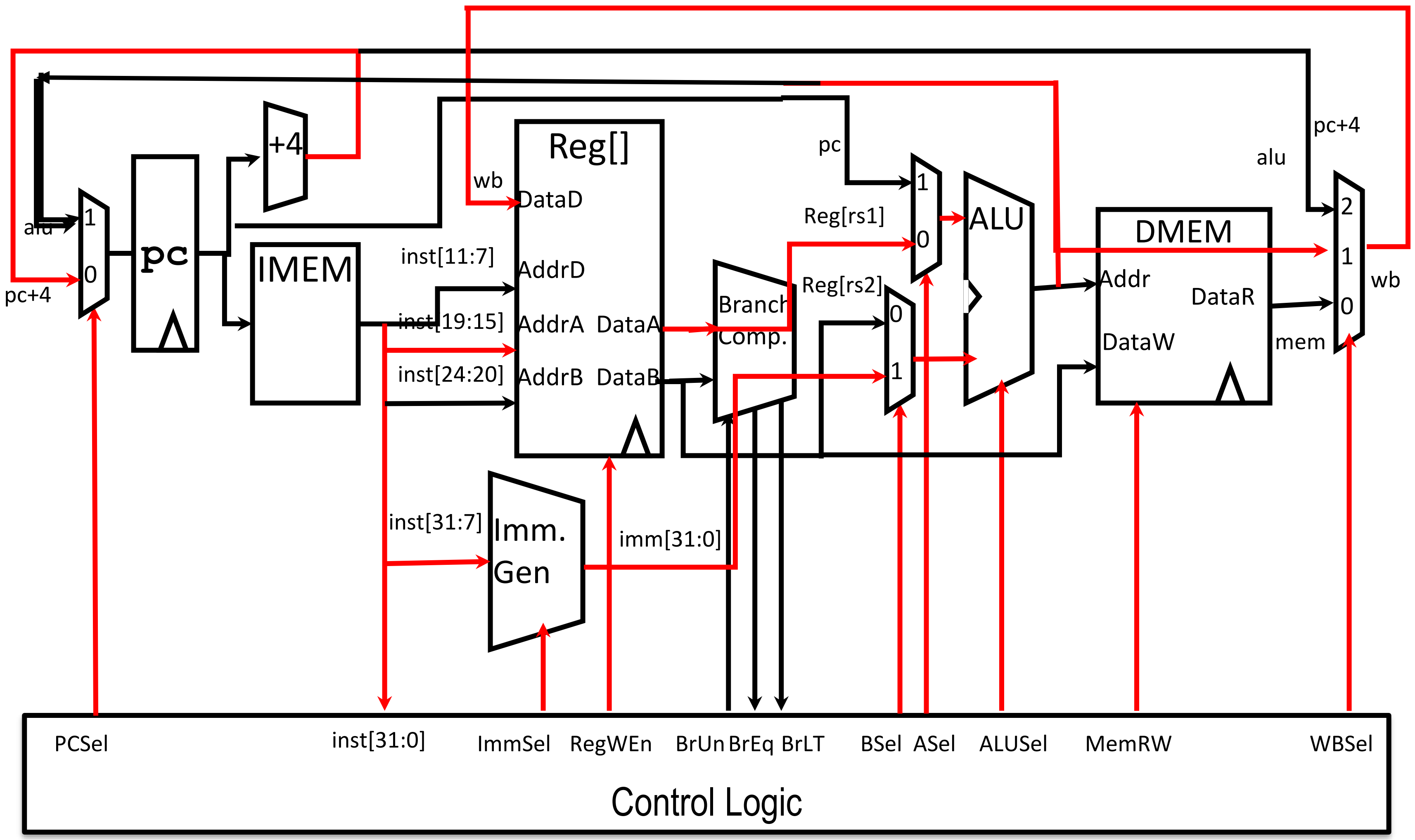
Here are the signals and values we've compiled for our ADD instruction:

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
`add`	*	*	+4	*	*	Reg	Reg	Add	Read	1 (Y)	ALU

(green = left 3 cols = control INPUTS)

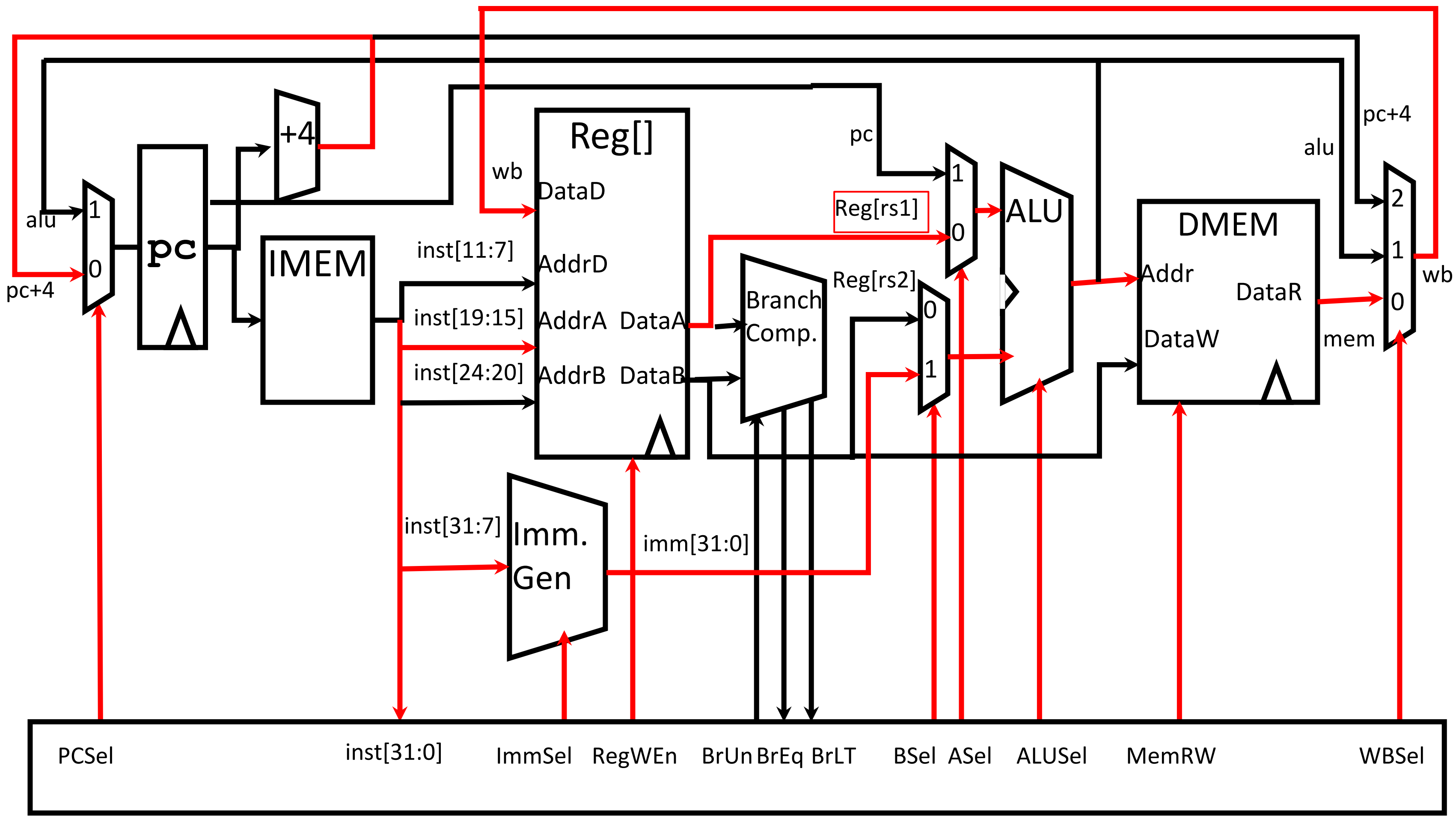
(orange = right 9 cols = control OUTPUTS)

addi datapath



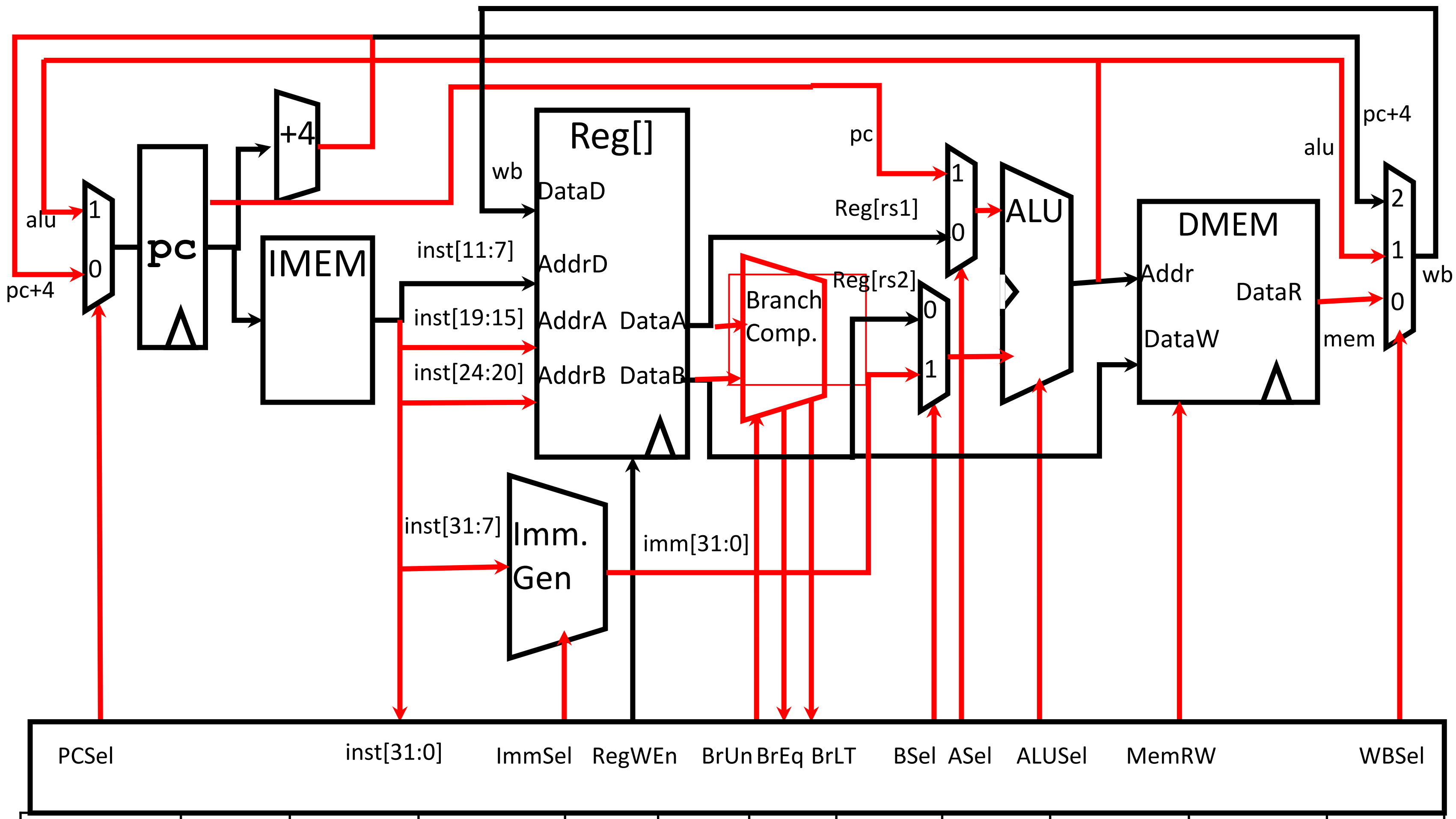
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br LT	Br Eq	BSel	ASel	ALUSel	MemRW	WBSel
addi	+4	I	1	*	*	*	Imm	Reg	Add	Read	ALU

lw datapath



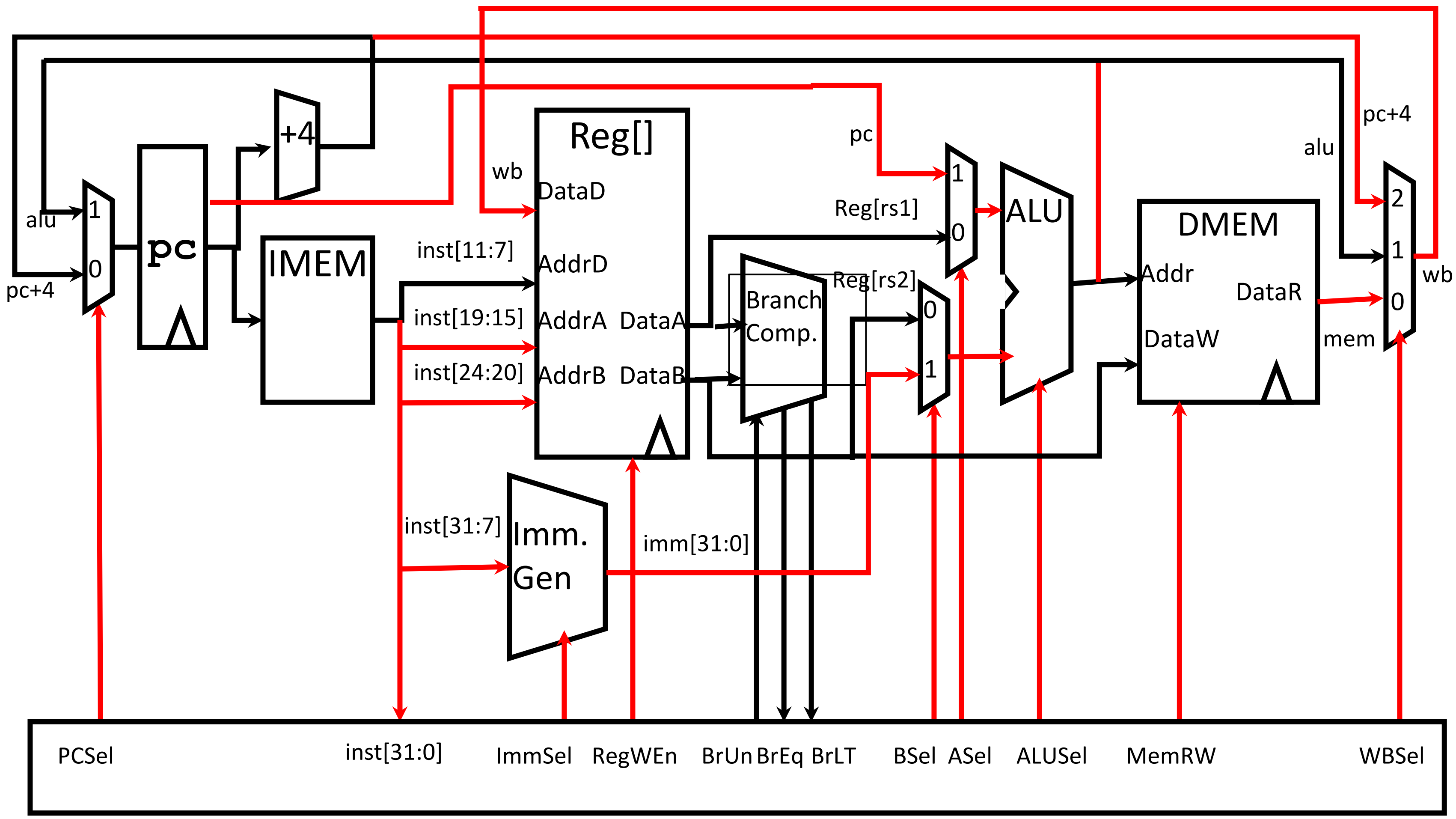
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	Br LT	BSel	ASel	ALUSel	MemRW	WBSel
`lw`	+4	I	1	*	*	*	Imm	Reg	Add	Read	Mem

Br datapath



Inst[31:0]	PCSel	ImmSel	RegWEn	BrUn	BrEq	BrLT	BSel	ASel	ALUSel	MemRW	WBSel
beq	+4	B	0	*	0	*	Imm	PC	Add	Read	*
beq	ALU	B	0	*	1	*	Imm	PC	Add	Read	*

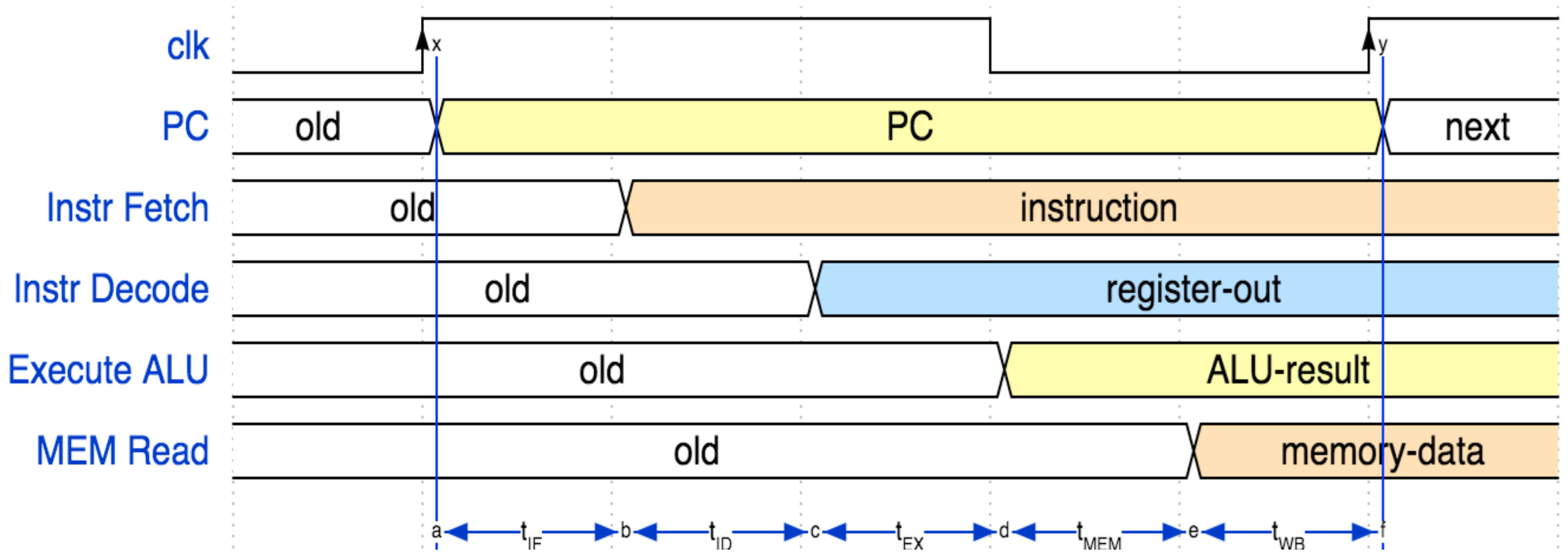
jal datapath



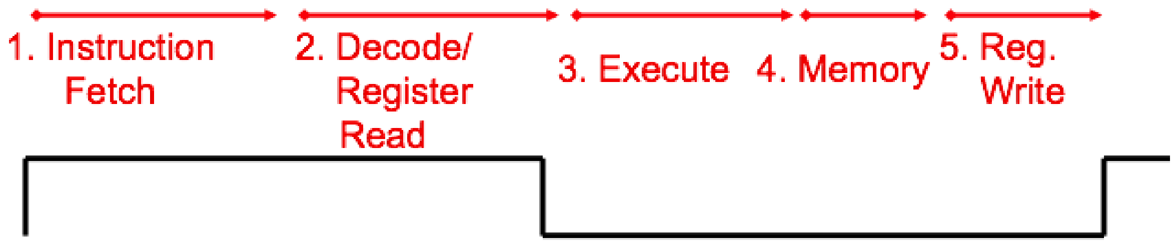
Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	BrLT	BSel	ASel	ALUSel	MemRW	WBSel
jal	ALU	J	1	*	*	*	Imm	PC	Add	Read	PC+4

Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	Br LT	BSel	ASel	ALUSel	MemRW	WBSel
`add`	+4	*	1 (Y)	*	*	*	Reg	Reg	Add	Read	ALU
`sub`	+4	*	1	*	*	*	Reg	Reg	Sub	Read	ALU
`(R-R Op)`	+4	*	1	*	*	*	Reg	Reg	*(Op)*	Read	ALU
`addi`	+4	I	1	*	*	*	Imm	Reg	Add	Read	ALU
`lw`	+4	I	1	*	*	*	Imm	Reg	Add	Read	Mem
`sw`	+4	S	0 (N)	*	*	*	Imm	Reg	Add	Write	*
`beq`	+4	B	0	*	0	*	Imm	PC	Add	Read	*
`beq`	ALU	B	0	*	1	*	Imm	PC	Add	Read	*
`bne`	ALU	B	0	*	0	*	Imm	PC	Add	Read	*
`bne`	+4	B	0	*	1	*	Imm	PC	Add	Read	*
`blt`	ALU	B	0	0	*	1	Imm	PC	Add	Read	*
`bltu`	ALU	B	0	1	*	1	Imm	PC	Add	Read	*
`jalr`	ALU	I	1	*	*	*	Imm	Reg	Add	Read	PC+4
`jal`	ALU	J	1	*	*	*	Imm	PC	Add	Read	PC+4
`auipc`	+4	U	1	*	*	*	Imm	PC	Add	Read	ALU

Instruction Timing



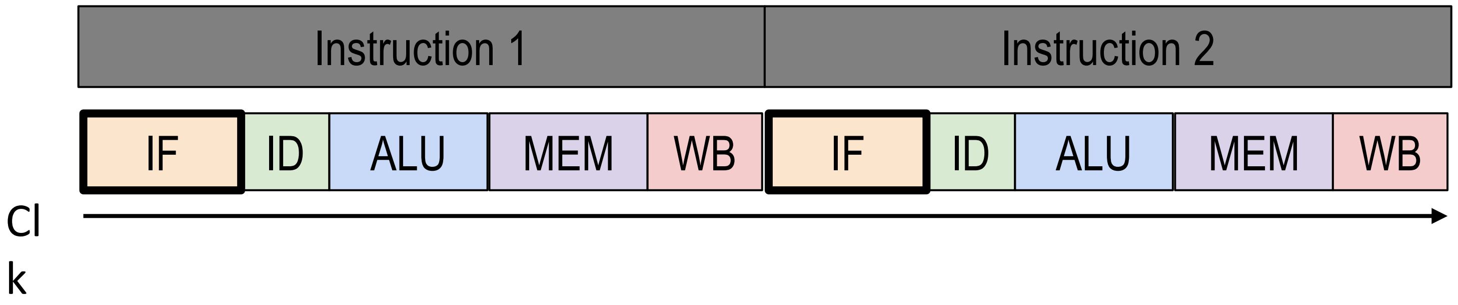
IF	ID	EX	MEM	WB	Total
IMEM	Reg Read	ALU	DMEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	800 ps



Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

- Maximum clock frequency
  - f_{max} = 1/800ps = 1.25 GHz
- Most blocks idle most of the time! ex. "IF" active every 600ps



“Iron Law” of Processor Performance

$$\boxed{\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}}$$

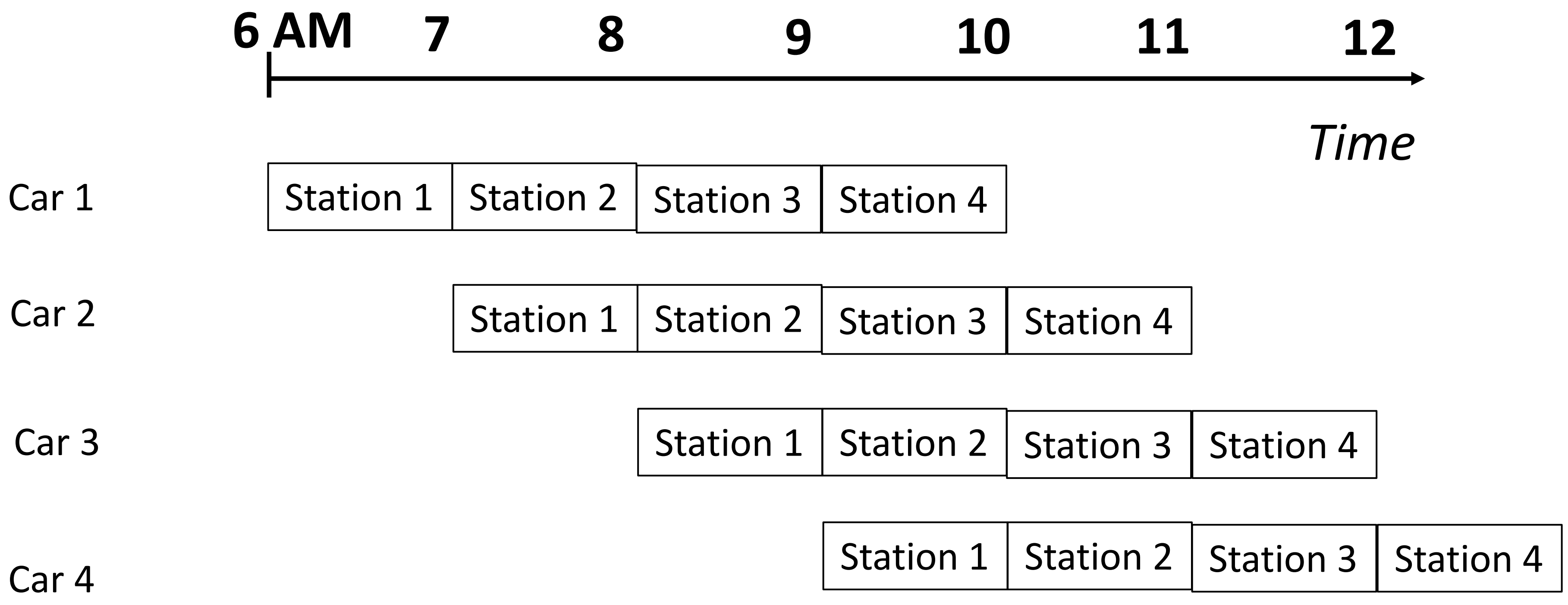
Speed Trade-off Example

- For some task (e.g. image compression) …

	Processor A	Processor B
# Instructions	1 Million	1.5 Million
Average CPI	2.5	1
Clock rate *f*	2.5 GHz	2 GHz
Execution time	1 ms	0.75 ms

Processor B is faster for this task, despite executing more instructions and having a lower clock rate! Why? Each instruction is less complex! (~2.5 B instructions = 1 A instruction)

Pipelined Car Assembly Line



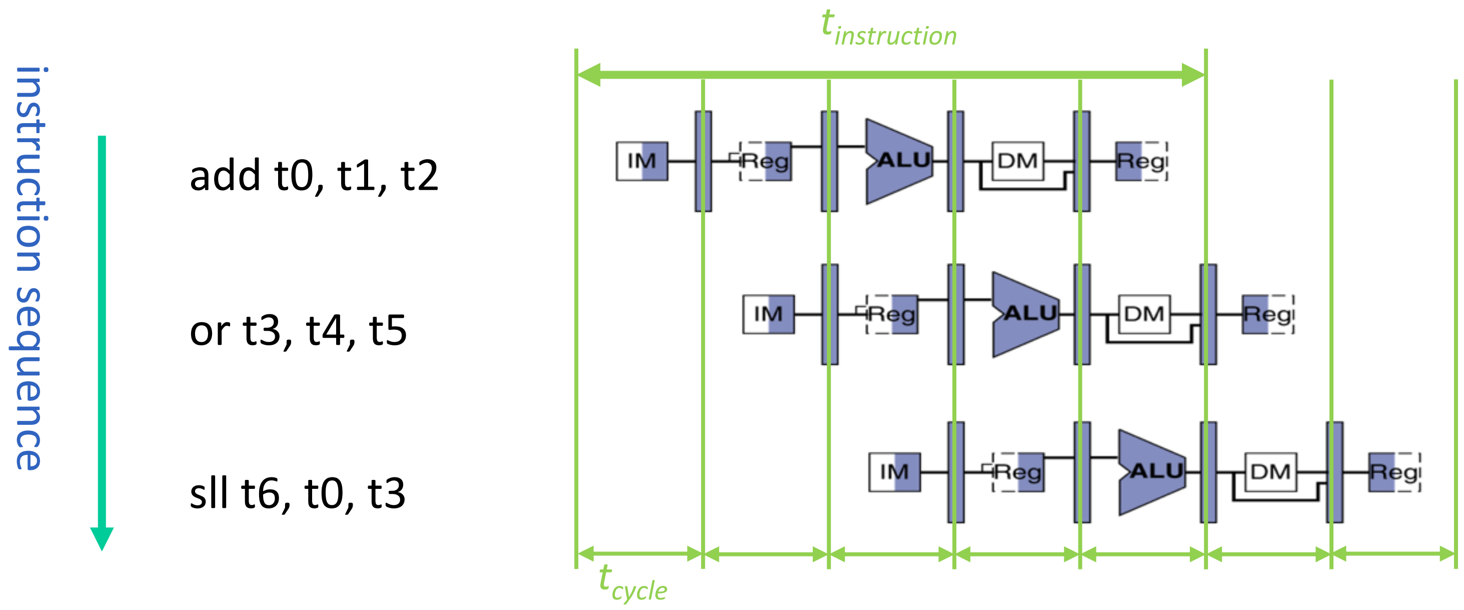
- Pipelined Car assembly takes 7 hours for 4 cars

➢ 1 car finishes every hour (after the car, which takes 4 hours)

Pipelining Lessons

- Pipelining doesn’t decrease *latency* of single task; it increases *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources
- Potential speedup ~ number of pipeline stages
- Speedup reduced by time to *fill* and *drain* the pipeline:
  16 hours/7 hours which gives 2.3X speedup v. potential 4X in this example

Pipelining with RISC-V



	Single Cycle	Pipelining
Timing	t_{step} = 100 … 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz

RISC-V Pipeline

instruction sequence

add t0, t1, t2

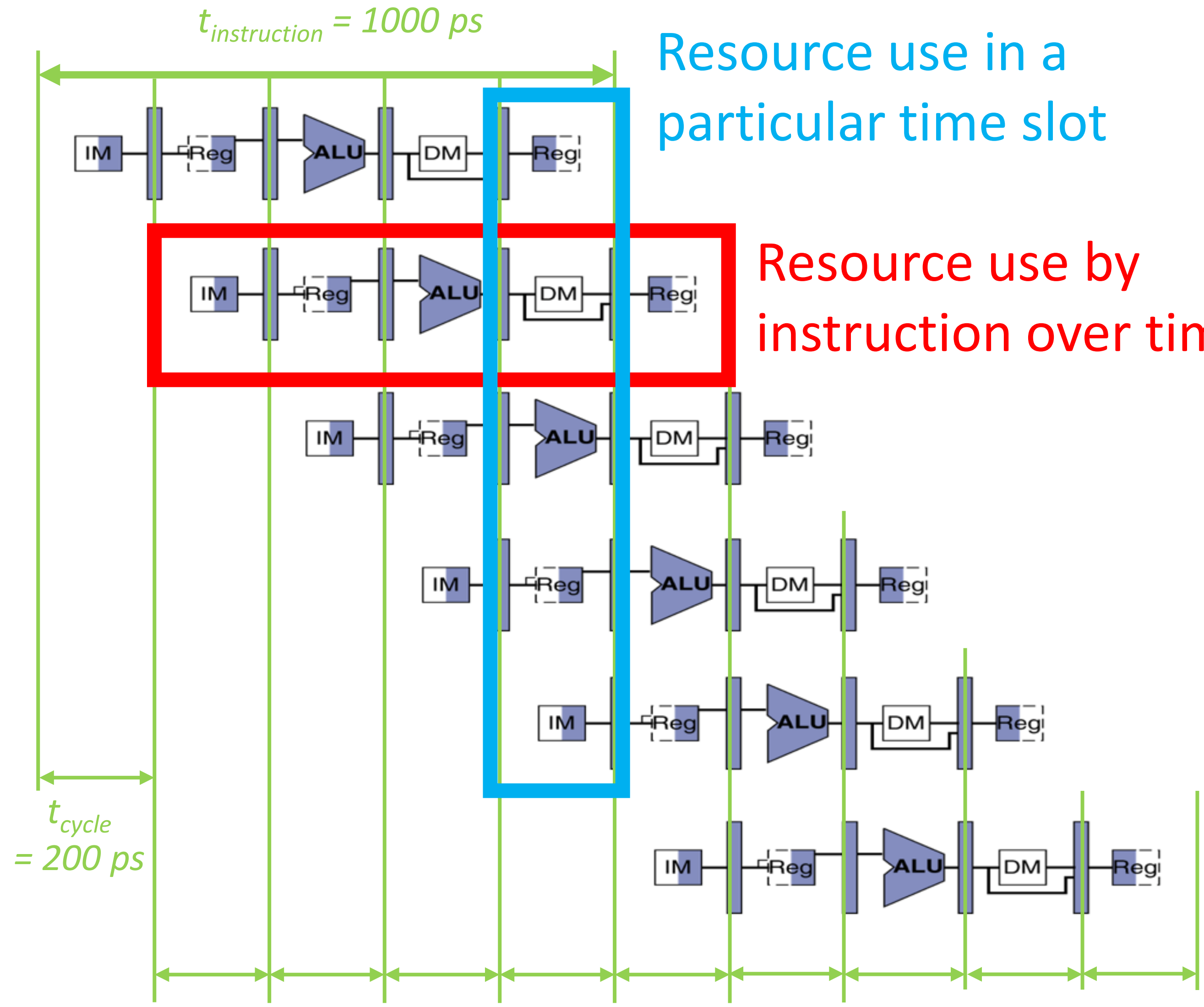
or t3, t4, t5

slt t6, t0, t3

sw t0, 4(t3)

lw t0, 8(t3)

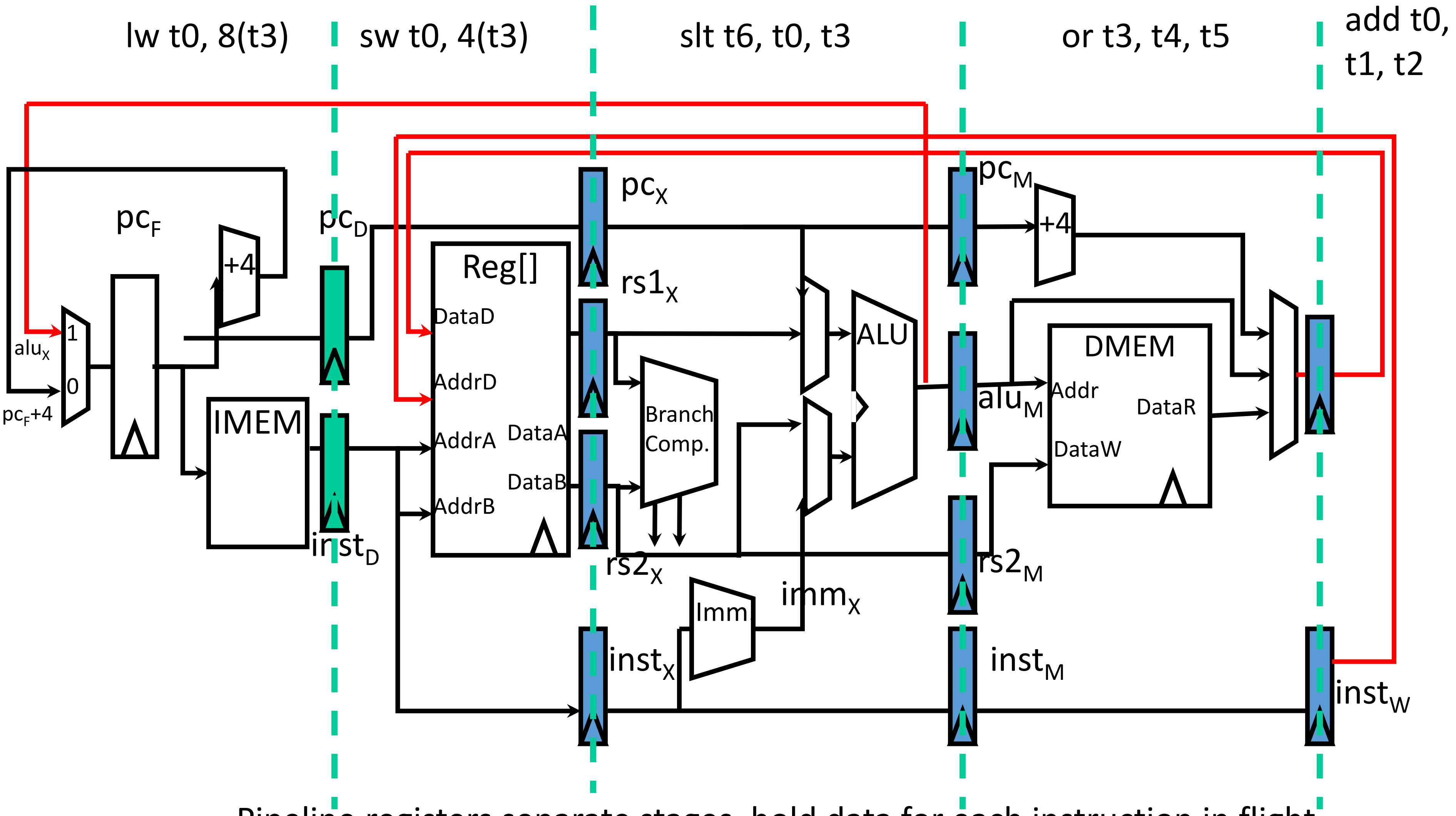
addi t2, t2, 1



Resource use in a particular time slot

Resource use by instruction over time

Each stage operates on different instruction



Pipeline registers separate stages, hold data for each instruction in flight

RISC-V Pipeline Example

Address	Inst | Cycle	0	1	2	3	4	5	6	7
0x00	add a1,a2,a3	IF	ID	EX	MEM	WB			
0x04	addi a4,a5,0x2f7		IF	ID	EX	MEM	WB		
0x08	sub s4,s0,s3			IF	ID	EX	MEM	WB	
0x0C	or s1,s2,s5				IF	ID	EX	MEM	WB

Instruction Level Parallelism (ILP)

- Pipelining allows us to execute parts of multiple instructions at the same time using the same hardware!
  - This is known as *instruction level parallelism*
- Later: Other types of parallelism
  - DLP: same operation on lots of data (SIMD)
  - TLP: executing multiple threads "simultaneously" (OpenMP)

Question: Assume the stage times shown below. Suppose we *remove loads and stores* from our ISA. Consider going from a single-cycle implementation to a **4-stage** pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.25x slower.
2) The ***throughput*** will be 3x faster.

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

No mem access

throughput:

(IF+ID+EX+WB) = 600 →

(4*max_stage)/4 = 200

old/new = 600/200 = 3x faster

Question: Assume the stage times shown below. Suppose we *remove loads and stores* from our ISA. Consider going from a single-cycle implementation to a **4-stage** pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.25x slower.
2) The ***throughput*** will be 3x faster.

	1	**2**
(A)	**F**	**F**
(B)	**F**	**T**
(C)	**T**	**F**
(D)	**T**	**T**

No mem access! Latency:
IF+ID+EX+WB = 600 →
4*max_stage = 800
old/new = 600/800 = negative speedup!
800/600 = 1.33x slower!

Question: Assume the stage times shown below. Suppose we *remove loads and stores* from our ISA. Consider going from a single-cycle implementation to a **4-stage** pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.25x slower.
2) The *throughput* will be 3x faster.

	1	2
(A)	F	F
[✓ (B)]	F	T
(C)	T	F
(D)	T	T

Agenda

Hazards Ahead!

- RISC-V Pipeline
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

Pipeline Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) Structural hazard

- A required resource is busy (e.g. needed in multiple stages)

2) Data hazard

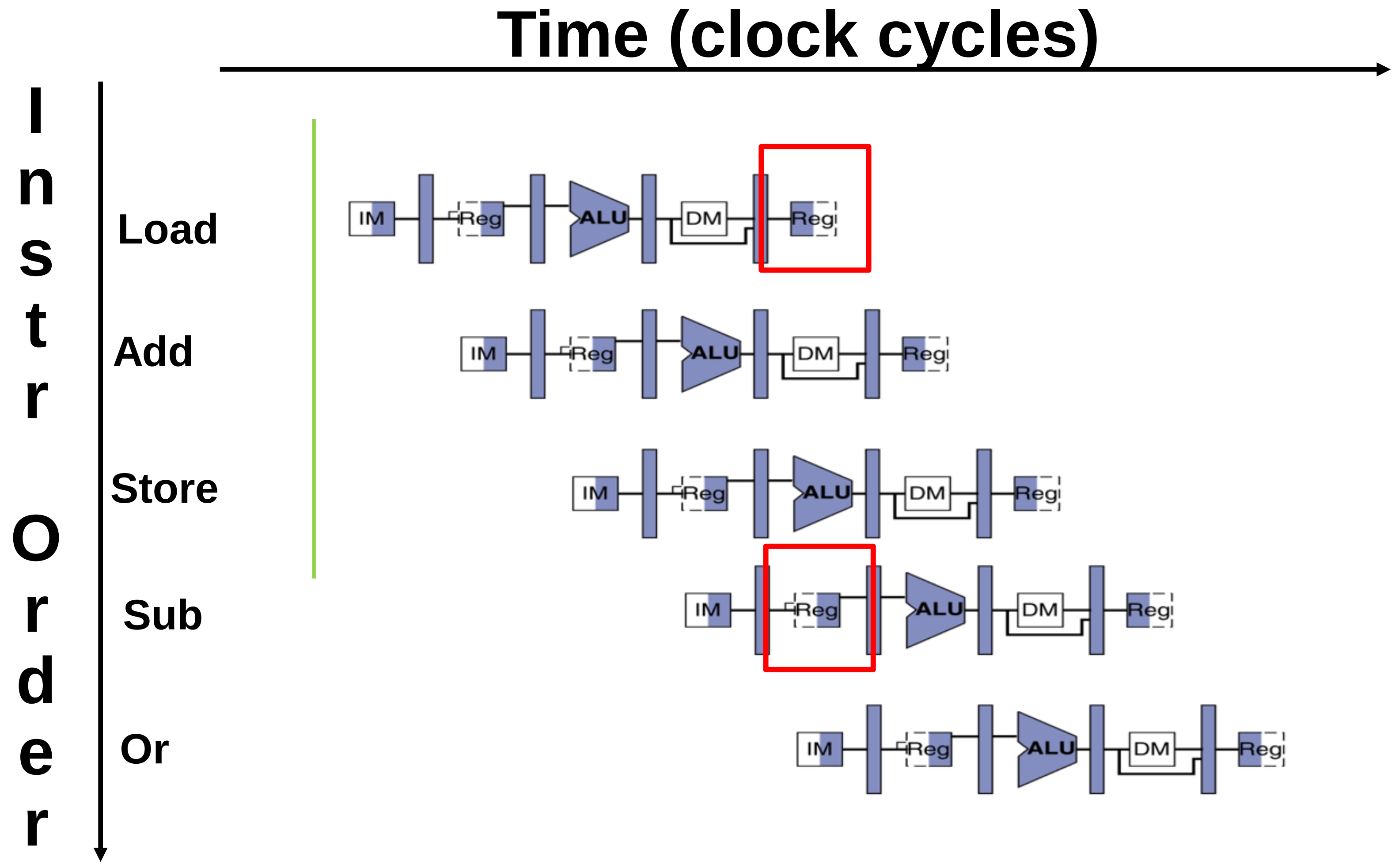
- Data dependency between instructions
- Need to wait for previous instruction to complete its data write

3) Control hazard

- Flow of execution depends on previous instruction

Structural Hazard: Regfile!

- RegFile: Used in ID and WB!

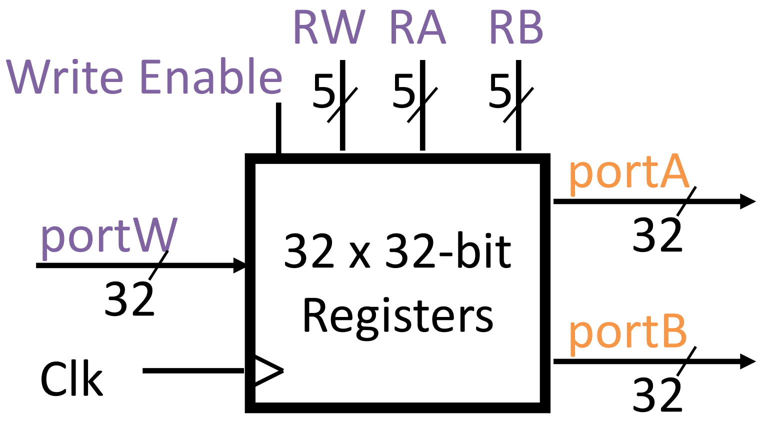


RISC-V Pipeline: Regfile Structural Hazard

Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	addi a0, zero, 5	IF	ID	EX	MM	WB						
0x04	addi a1, a4, 5		IF	ID	EX	MM	WB					
0x08	addi a2, a5, 5			IF	ID	EX	MM	WB				
0x0C	addi a3, a6, 5				IF	ID	ID	EX	MM	WB		

Regfile Structural Hazards

- Each instruction:
  - Can read up to two operands in decode stage
  - Can write one value in writeback stage
- Avoid structural hazard by having separate "ports"
  - Two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously



Regfile Structural Hazards

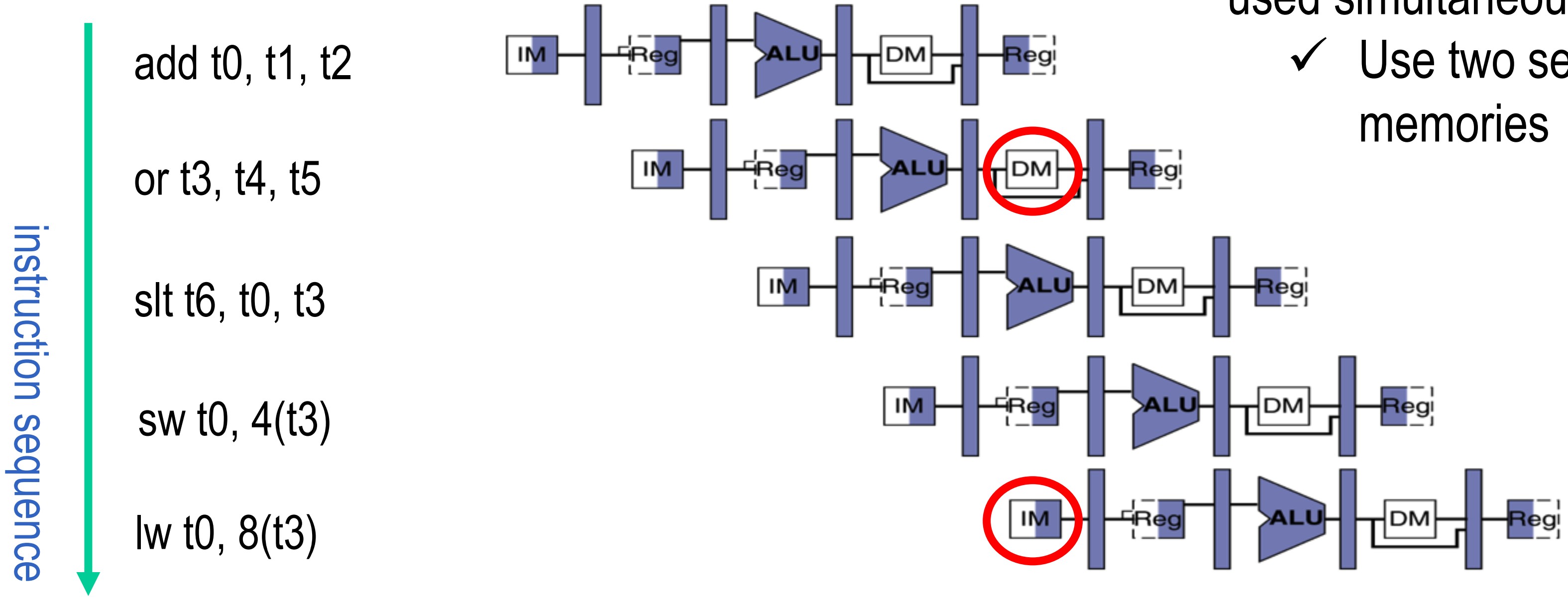
- Two *alternate* solutions:
  1) Build RegFile with independent read and write ports (assignment); good for single-stage
  2) Double Pumping: split RegFile access in two! Prepare to write during 1st half, write on *falling* edge, read during 2nd half of each clock cycle
     - Will save us a cycle later...
     - Possible because RegFile access is *VERY* fast (takes less than half the time of ALU stage)
- **Conclusion:** Read and Write to registers during same clock cycle is okay

Regfile Structural Hazard: 2 Rd+1Wr Ports

Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	addi a0, zero, 5	IF	ID	EX	MM	WB						
0x04	addi a1, a4, 5		IF	ID	EX	MM	WB					
0x08	addi a2, a5, 5			IF	ID	EX	MM	WB				
0x0C	addi a3, a6, 5				IF	ID	EX	MM	WB			

Structural Hazard: Memory Access

- Instruction and data memory used simultaneously
  - ✓ Use two separate memories



Structural Hazards – Summary

- Conflict for use of a resource
- In RISC-V pipeline with a single memory unit
  - Load/store requires data access
  - Without separate memory units, instruction fetch would have to *stall* for that cycle
    - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data memory units
  - Or separate instruction/data caches
- RISC ISAs (including RISC-V) designed to avoid structural hazards
  - e.g. at most one memory access/instruction

2. Data Hazards (1/2)

- Consider the following sequence of instructions:

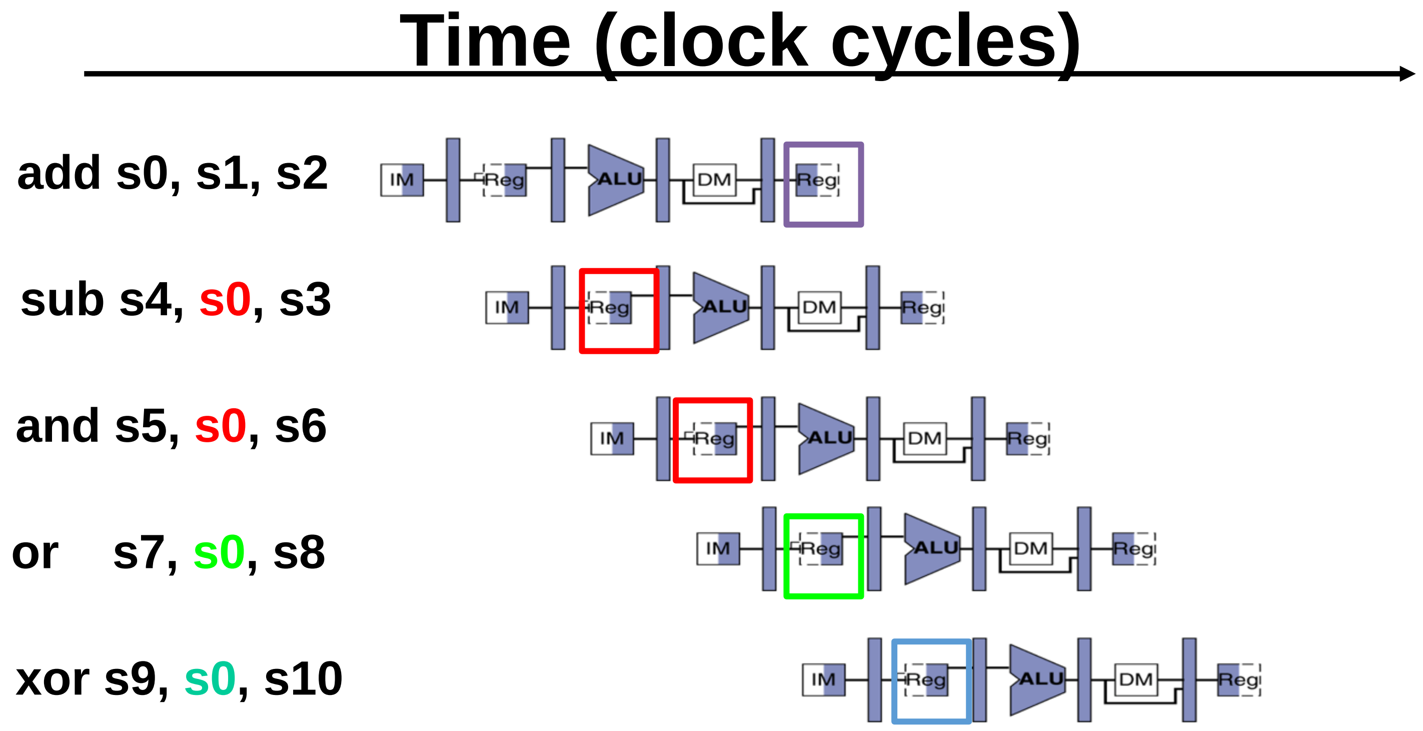
```
add  s0, s1, s2
sub  s4, s0, s3
and  s5, s0, s6
or   s7, s0, s8
```

$$\texttt{xor}\quad \underbrace{\texttt{s9,}}_{\text{Stored during WB}}\ \underbrace{\texttt{s0, s10}}_{\text{Read during ID}}$$

2. Data Hazards (2/2)

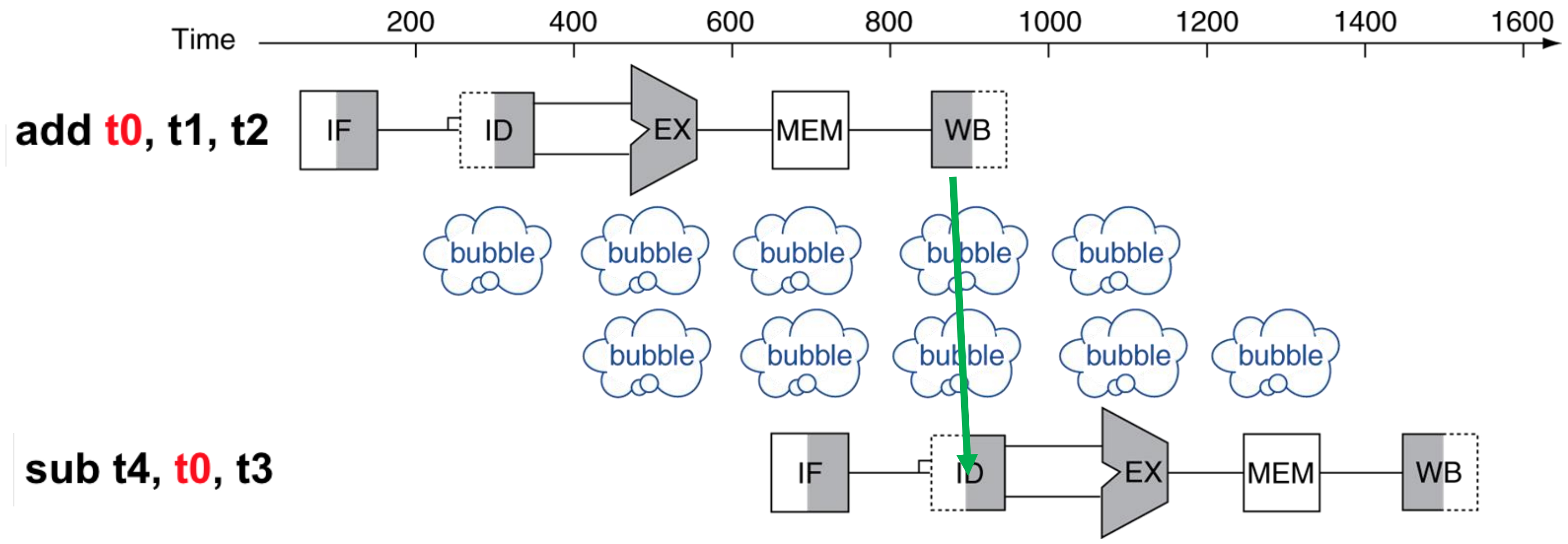
Identifying data hazards:

- Where is data **WRITTEN**?
- Where is data **READ**?
- Does the WRITE happen AFTER the READ?



Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction
  - add s0, s1, s2
    sub s4, s0, s3



- Bubble:
  - effectively NOP: affected pipeline stages do “nothing” (add x0 x0 x0)

Data Hazard

Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	add s0, s1, s2	IF	ID	EX	MM	WB						
0x04	sub s4, s0, s3		IF	ID	-	-	EX	MM	WB			
0x08	and s5, s0, s6			IF	IF	IF	ID	EX	MM	WB		
0x0C	or s7, s0, s8						IF	ID	EX	MM	WB	

Data Hazard Solution: Forwarding

- Forward result as soon as it is available, even though it's not stored in RegFile yet

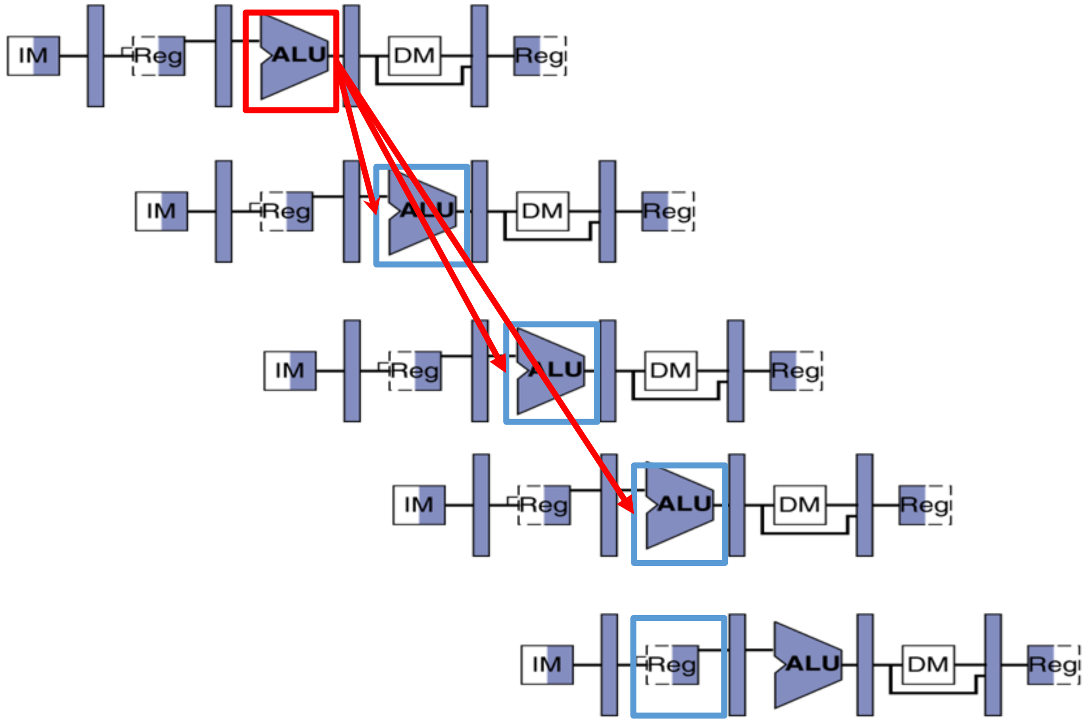
add s0, s1, s2

sub s4, s0, s3

and s5, s0, s6

or s7, s0, s8

xor s9, s0, s10



Forwarding: get operand from pipeline stage, rather than register file

Data Hazard with Forwarding

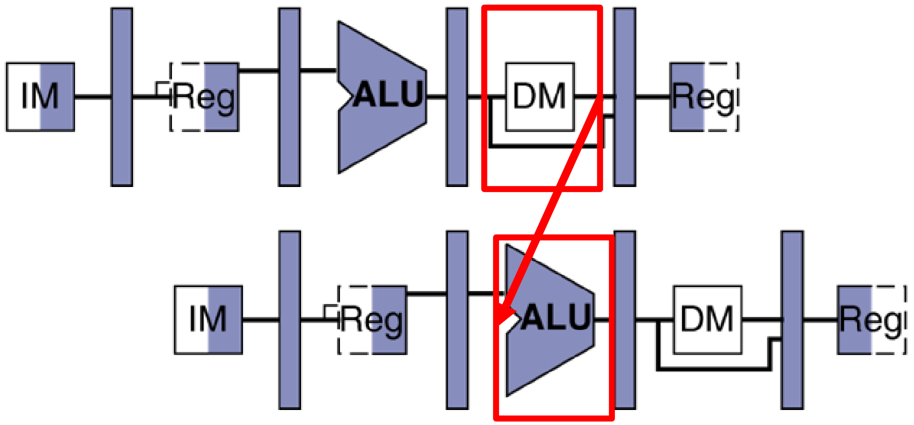
Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	add s0, s1, s2	IF	ID	EX	MM	WB						
0x04	sub s4, s0, s3		IF	ID	EX	MM	WB					
0x08	and s5, s0, s6			IF	ID	EX	MM	WB				
0x0C	or s7, s0, s8				IF	ID	EX	MM	WB			

Data Hazard: Loads (1/2)

- **Recall:** Dataflow backwards in time are hazards

lw t0, 0(t1)

sub t3, t0, t2



- Can't solve all cases with forwarding
  - Must *stall* instruction dependent on load (sub), then forward after the load is done (more hardware)

Data Hazard: Loads (2/2)

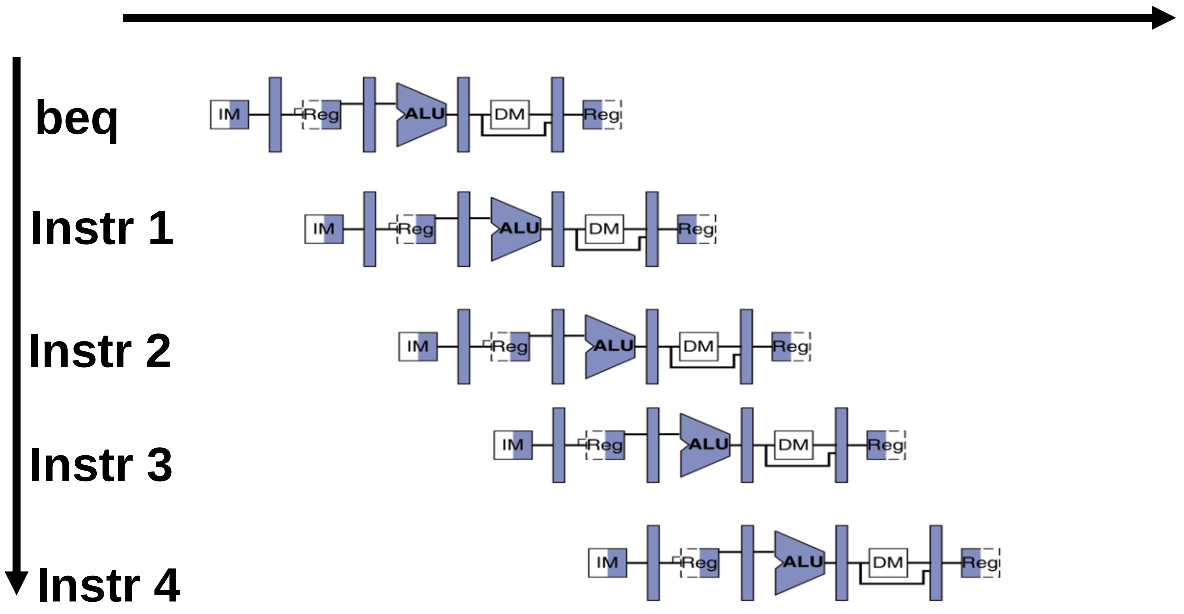
- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware will stall for one cycle
  - Equivalent to inserting an explicit **`nop`** in the slot
    - except the latter uses more code space
  - Performance loss
- **Idea:** Let the compiler/assembler put an unrelated instruction in that slot → no stall!

3. Control Hazards

- Branch (`beq`, `bne`, `...`) determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Result isn't known until end of execute
- **Simple Solution:** Stall *or flush* on *every* branch until we have the new PC value
  - How long must we stall?

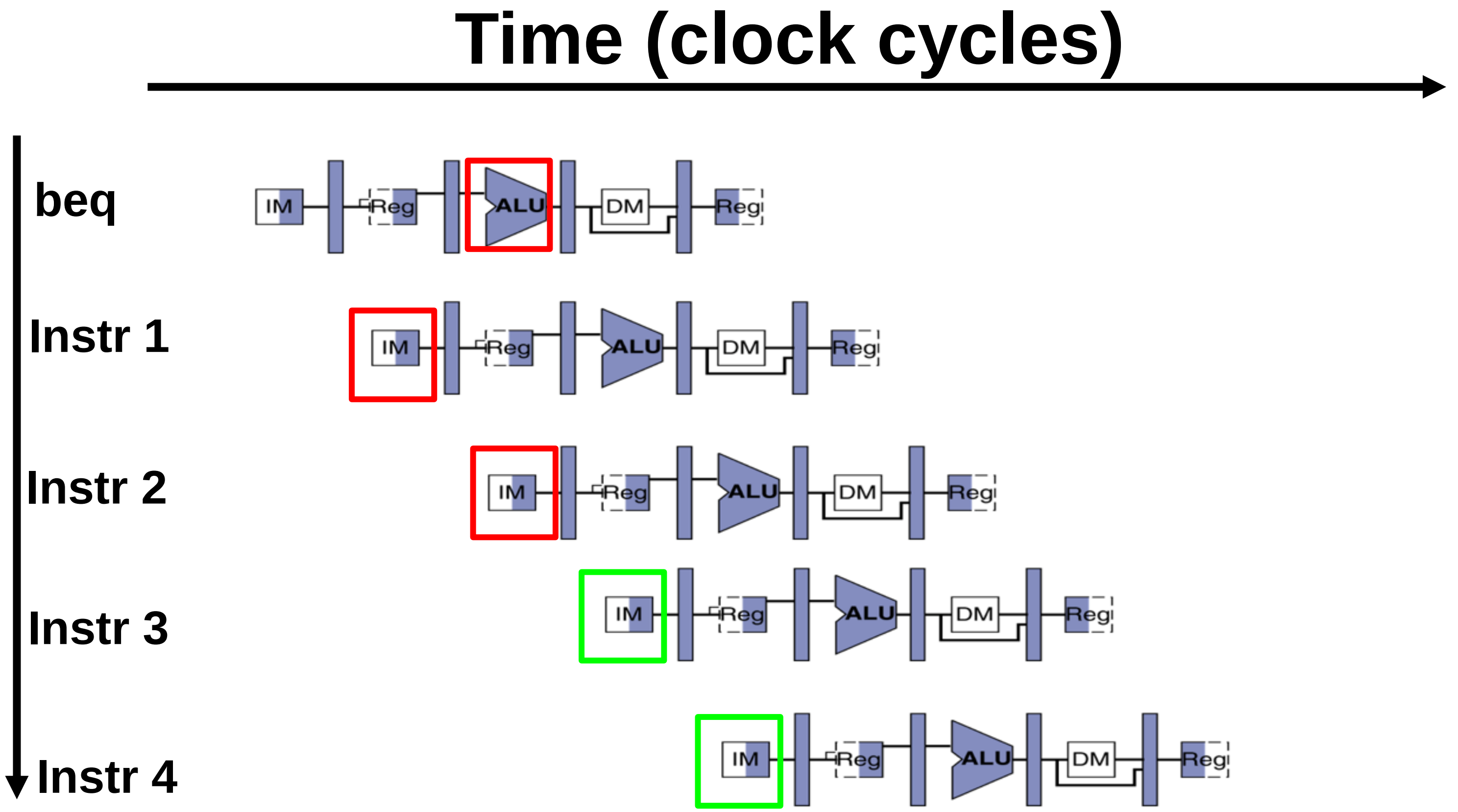
- How many instructions after **beq** are affected by the control hazard?

A) 1
B) 2
C) 3
D) 4
E) 5

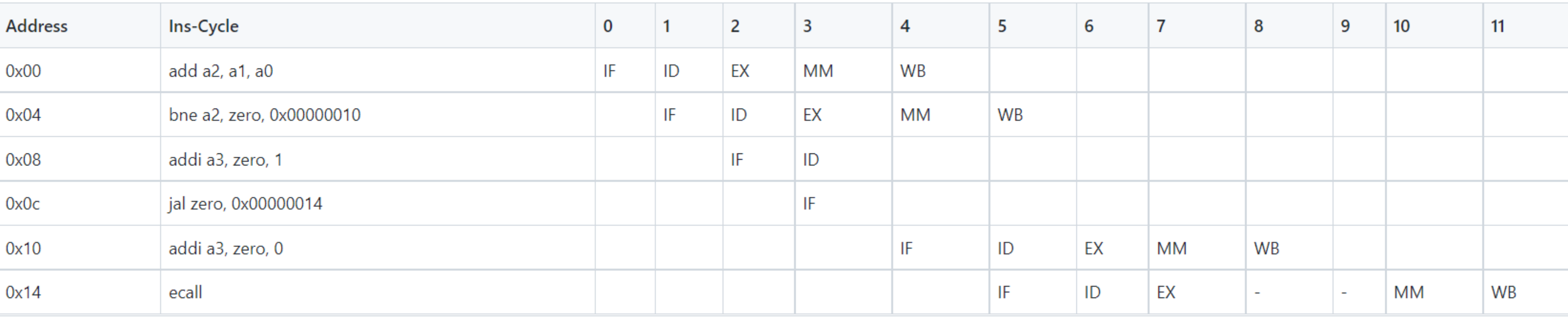


Branch Stall

- How many bubbles required for branch?

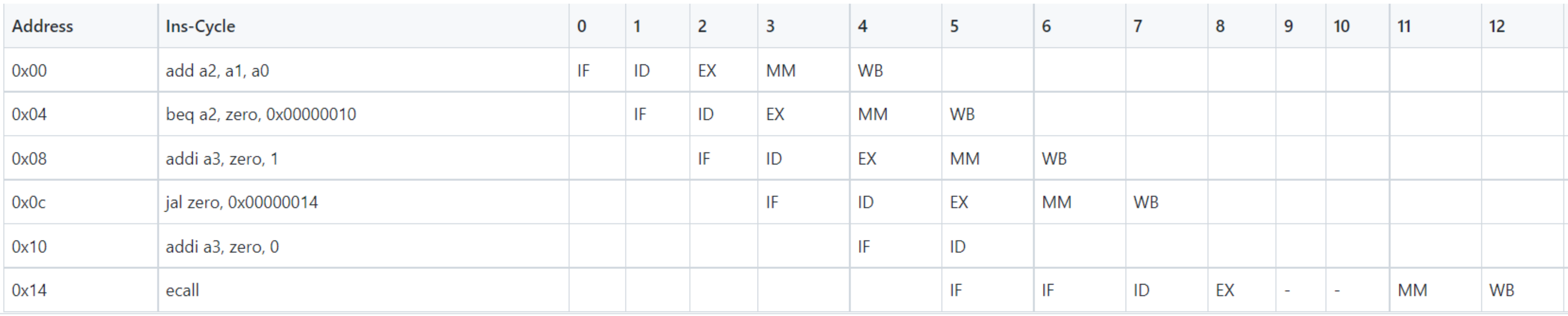


Taken Branch & ecall



Address	Ins-Cycle	0	1	2	3	4	5	6	7	8	9	10	11
0x00	add a2, a1, a0	IF	ID	EX	MM	WB							
0x04	bne a2, zero, 0x00000010		IF	ID	EX	MM	WB						
0x08	addi a3, zero, 1			IF	ID								
0x0c	jal zero, 0x00000014				IF								
0x10	addi a3, zero, 0					IF	ID	EX	MM	WB			
0x14	ecall						IF	ID	EX	-	-	MM	WB

Not-Taken Branch



Address	Ins-Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12
0x00	add a2, a1, a0	IF	ID	EX	MM	WB								
0x04	beq a2, zero, 0x00000010		IF	ID	EX	MM	WB							
0x08	addi a3, zero, 1			IF	ID	EX	MM	WB						
0x0c	jal zero, 0x00000014				IF	ID	EX	MM	WB					
0x10	addi a3, zero, 0					IF	ID							
0x14	ecall						IF	IF	ID	EX	-	-	MM	WB

3. Control Hazard: Branching

- **RISC-V Solution:** *Branch Prediction* – guess outcome of a branch, fix afterwards if necessary
  - Must cancel (*flush*) all instructions in pipeline that depended on guess that was wrong
  - How many instructions do we end up flushing?

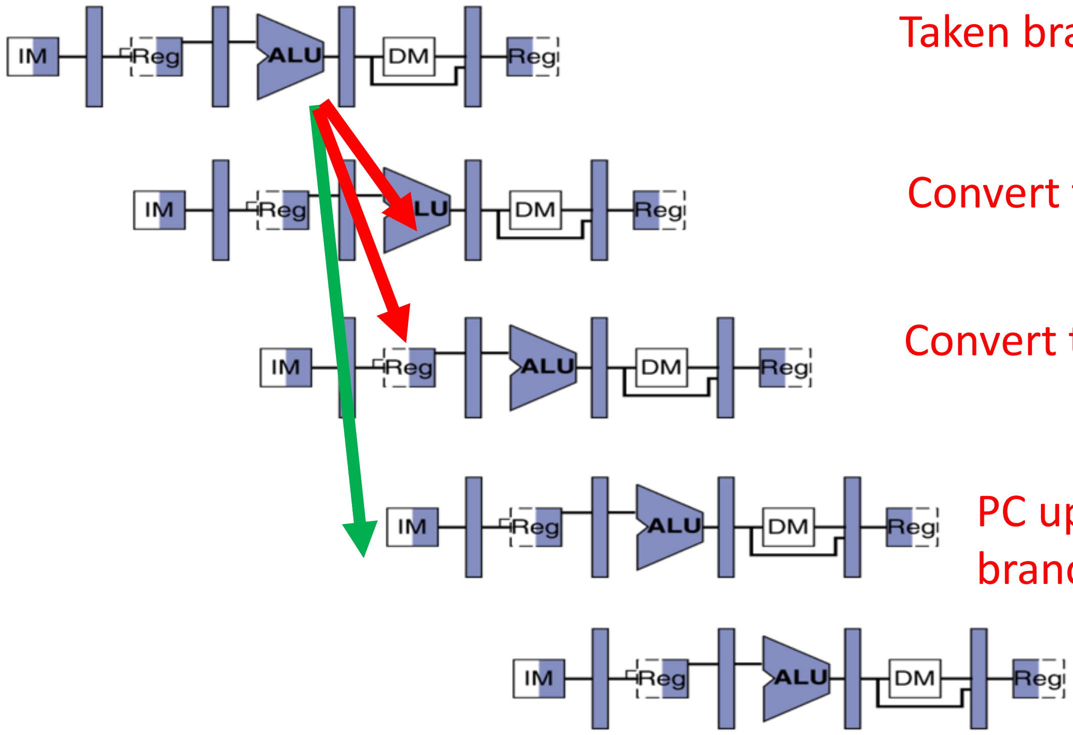
Clear Instructions after Branch if Taken

beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

label: xxxxxx



Taken branch

Convert to NOP

Convert to NOP

PC updated reflecting branch outcome

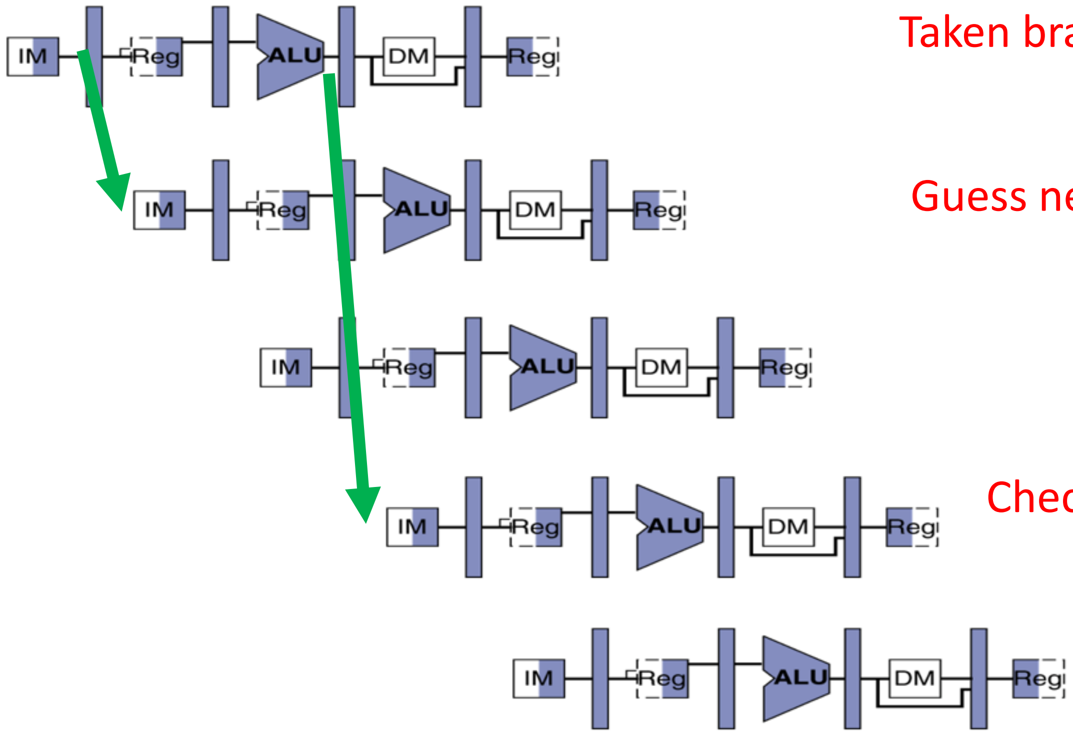
Two instructions are affected by an incorrect branch, just like we'd have to insert two NOP's/stalls in the pipeline to wait on the correct value!

Branch Prediction

beq t0, t1, label

label: .....

.....



Taken branch

Guess next PC!

Check guess correct

In the correct case, we don't have any stalls/NOP's at all!

Prediction, if done correctly, is better on average than stalling