

EE 044252: Digital Systems and Computer Structure
Spring 2018

Lecture 13: Pipeline









EE 044252: Digital Systems and Computer Structure

Topic	wk	Lectures	Tutorials	Workshop	Simulation
Arch	1	Intro. RISC-V architecture	Numbers. Codes		
Comb	2	Switching algebra & functions	Assembly programming		
	3	Combinational logic	Logic minimization	Combinational	
	4	Arithmetic. Memory	Gates		Combinational
Seq	5	Finite state machines	Logic		
	6	Sync FSM	Flip flops, FSM timing	Sequential	Sequential
	7	FSM equiv, scan, pipeline	FSM synthesis		
	8	Serial comm, RISC-V functions	Serial comm, pipeline		
μ Arch	9	RISC-V instruction formats	Function call		
	10	RISC-V single cycle	Single cycle RISC-V		
	11	Multi-cycle RISC-V	Multi-cycle RISC-V		
	12	Interrupts	Microcode, interrupts		Multi-cycle
	13	Pipeline RISC-V	Depend. in pipeline RISC-V		

Agenda

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

Pipelining with RISC-V

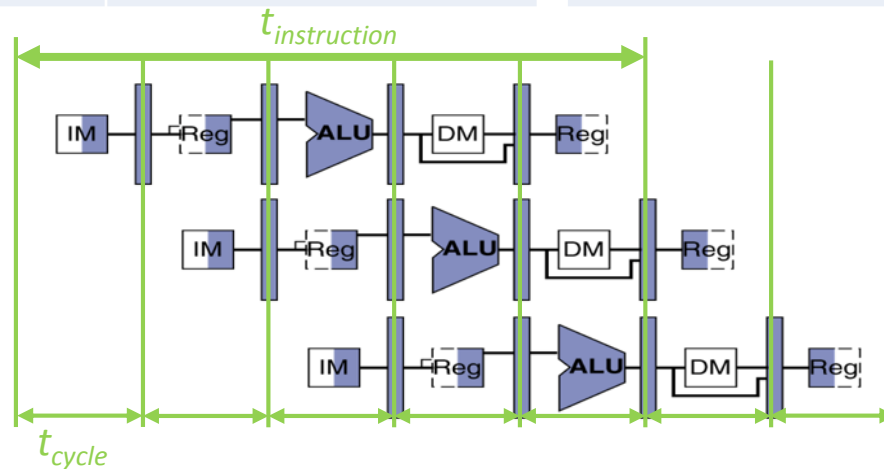
Phase	Pictogram	t_{step} Single Cycle	t_{cycle} Multi-Cycle	t_{cycle} Pipelined
Instruction Fetch		200 ps	200 ps	200 ps
Reg Read		100 ps	200 ps	200 ps
ALU		200 ps	200 ps	200 ps
Memory		200 ps	200 ps	200 ps
Register Write		100 ps	200 ps	200 ps
$t_{instruction}$		800 ps	1000 ps	1000 ps

instruction sequence
↓

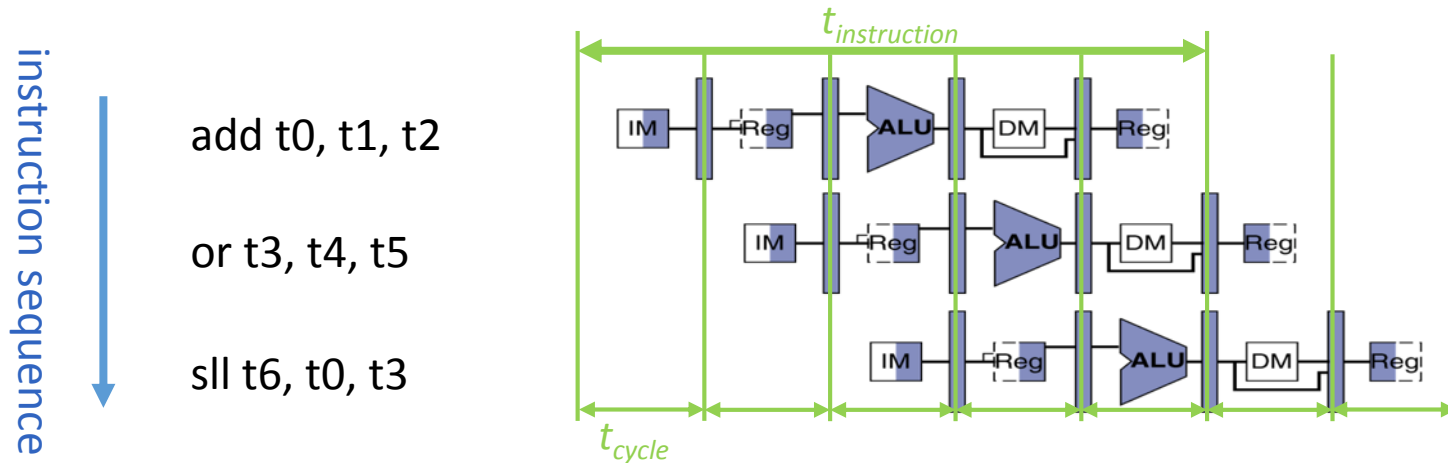
add t0, t1, t2

or t3, t4, t5

sll t6, t0, t3

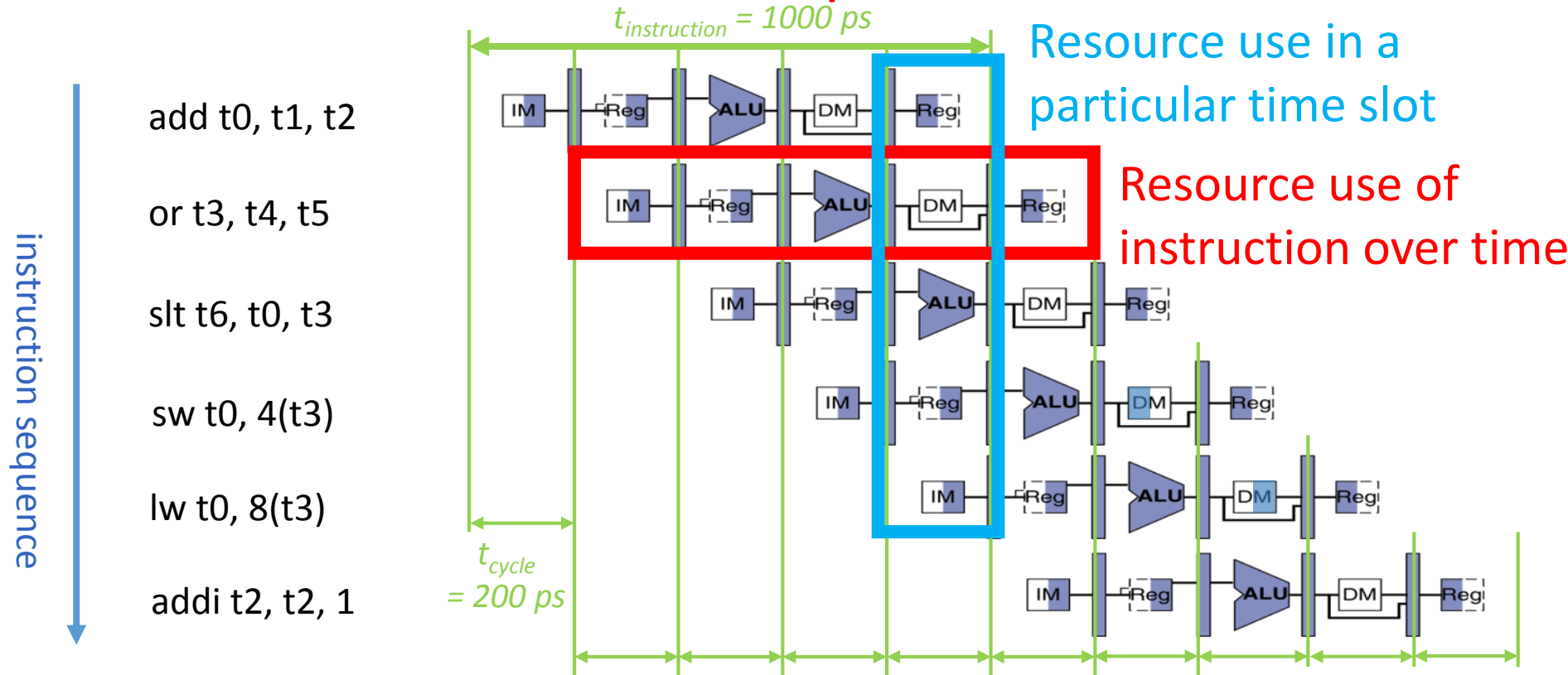


Pipelining with RISC-V

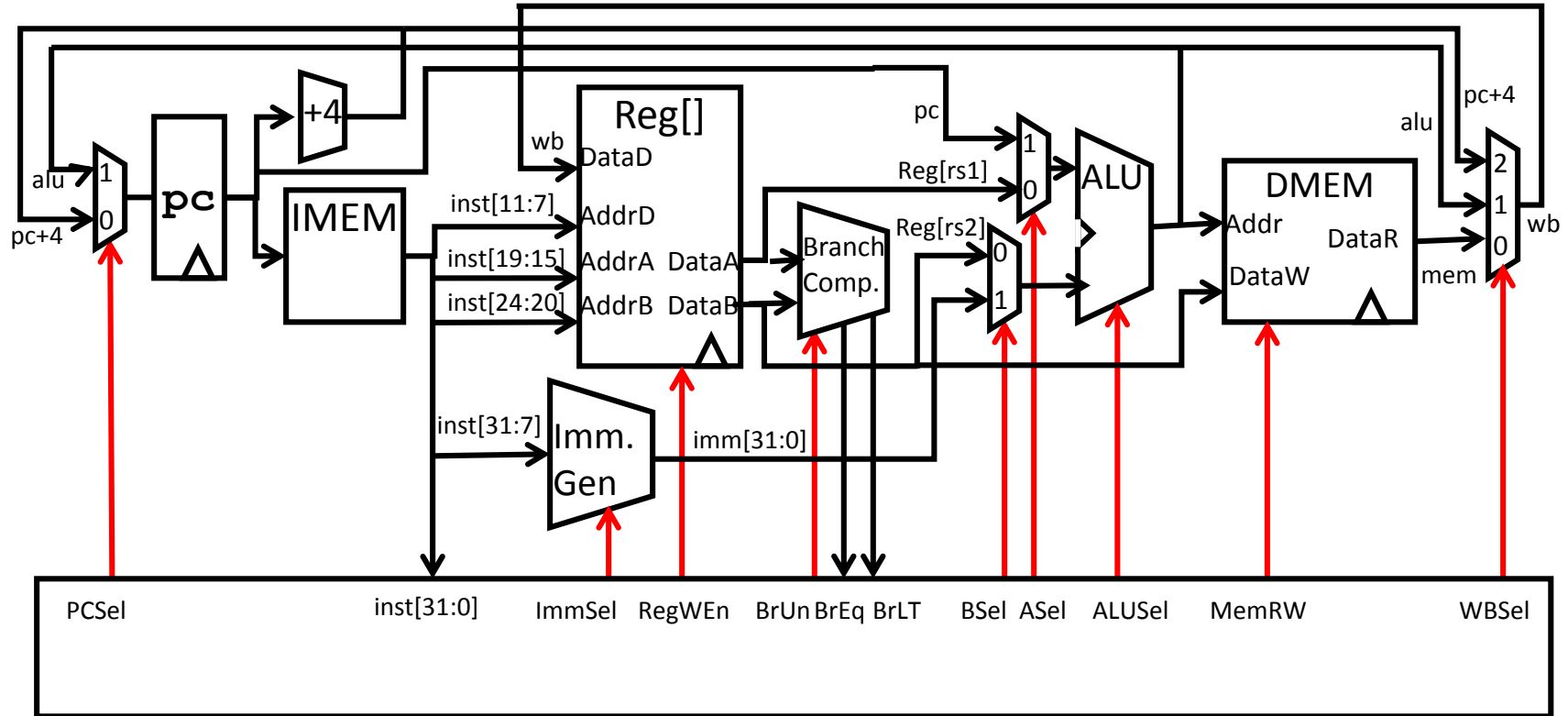


	Single Cycle	Multi-Cycle	Pipeline
Timing	$t_{step} = 100 \dots 200 \text{ ps}$	$t_{cycle} = 200 \text{ ps}$	$t_{cycle} = 200 \text{ ps}$
	Register access only 100 ps	All cycles same length Not all cycle needed	All cycles same length
Instruction time, $t_{instruction}$	$= t_{cycle} = 800 \text{ ps}$	Average (~4 cycles) 800 ps	1,000 ps
Clock rate, f_s	$1/800 \text{ ps} = 1.25 \text{ GHz}$	$1/200 \text{ ps} = 5 \text{ GHz}$	$1/200 \text{ ps} = 5 \text{ GHz}$
Relative throughput	1 x	1x	4x

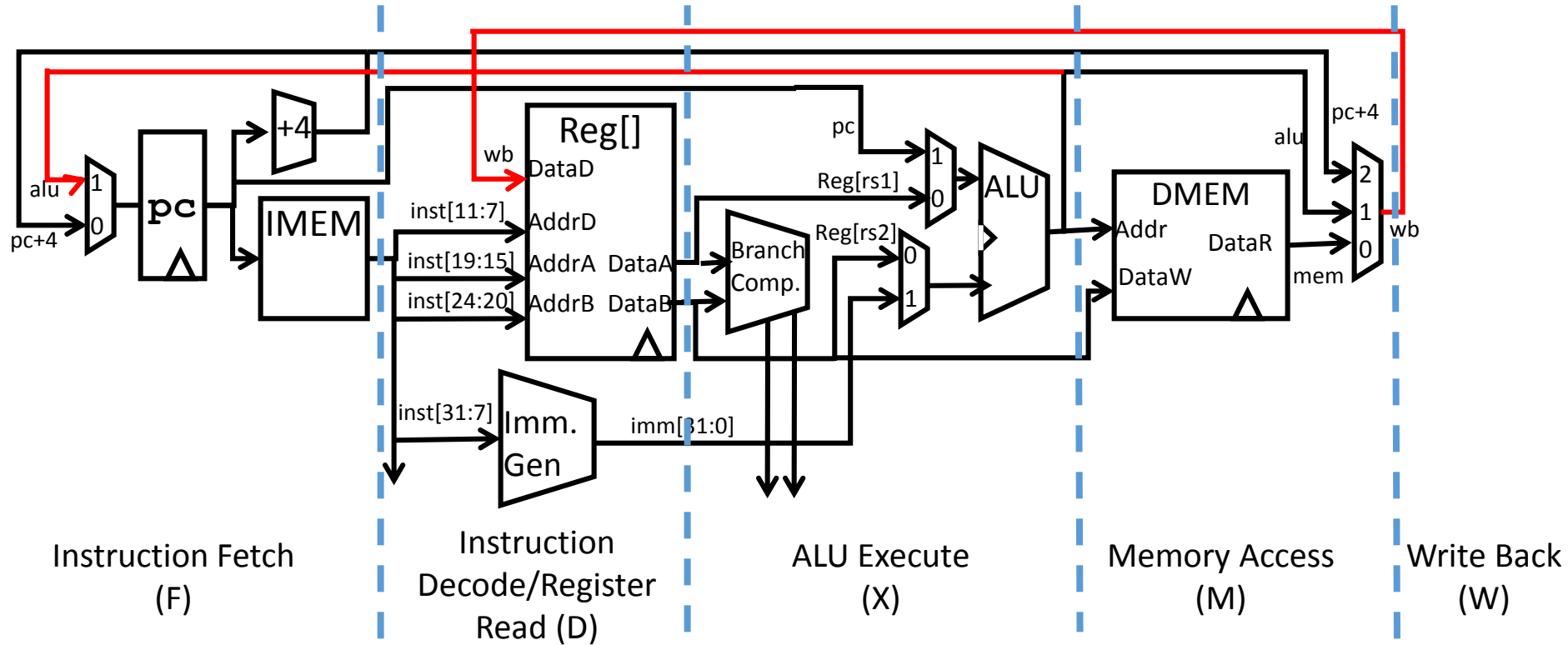
RISC-V Pipeline



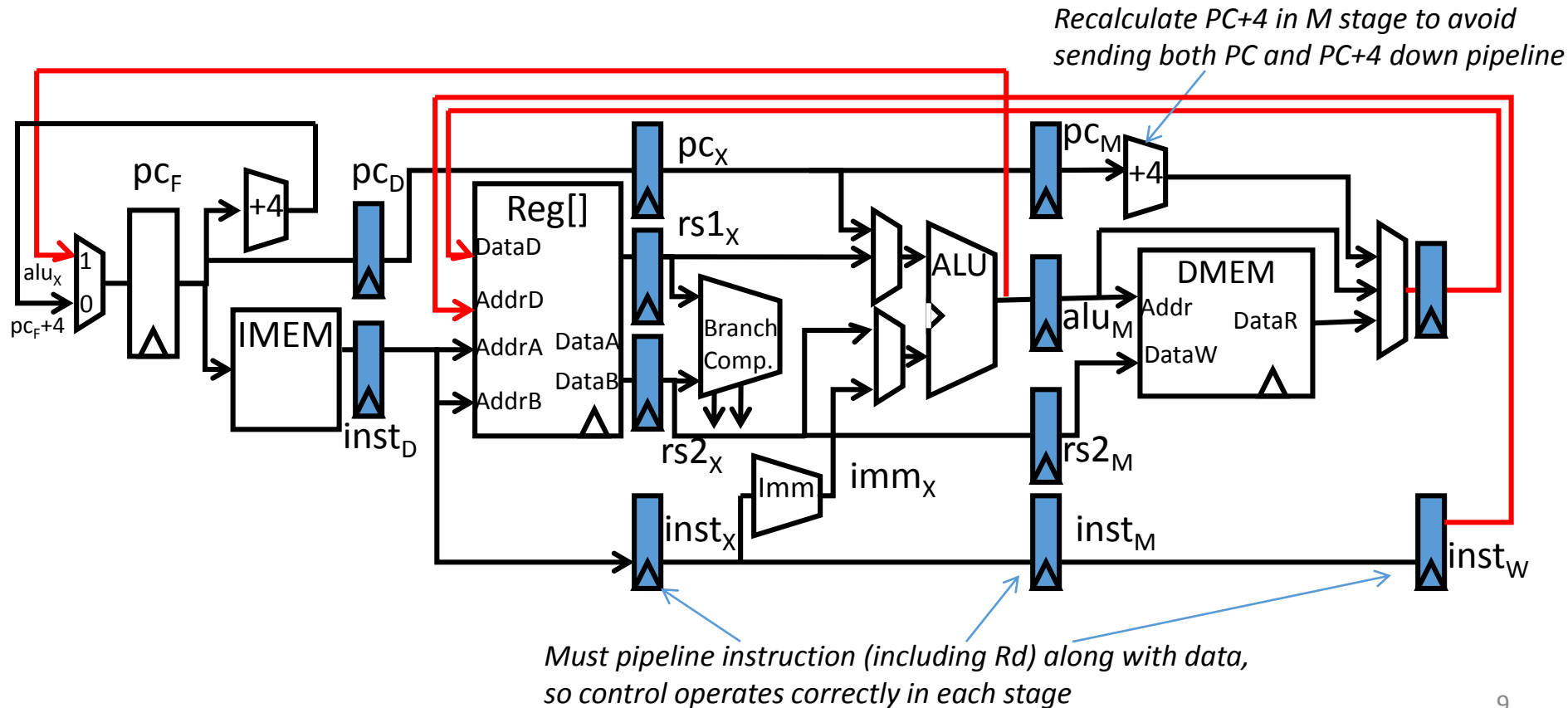
Single-Cycle RISC-V RV32I Datapath



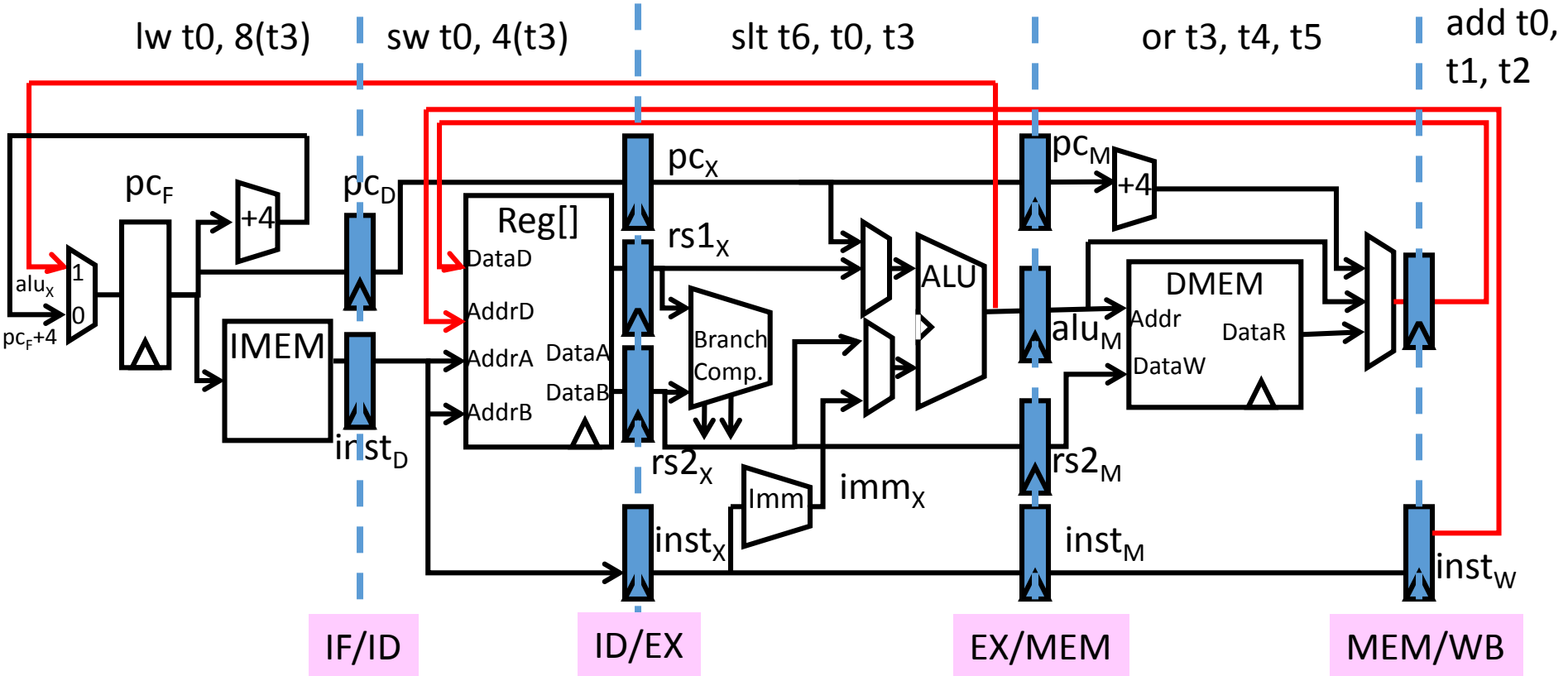
Pipelining RISC-V RV32I Datapath



Pipelined RISC-V RV32I Datapath



Each stage operates on different instruction



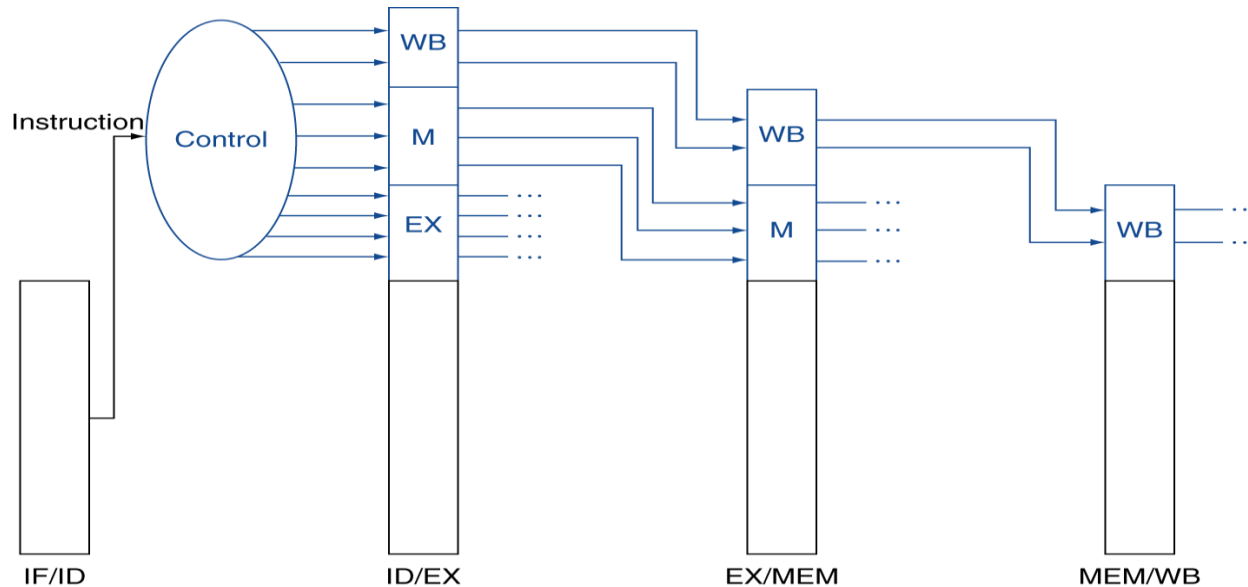
Pipeline registers separate stages, hold data for each instruction in flight

Agenda

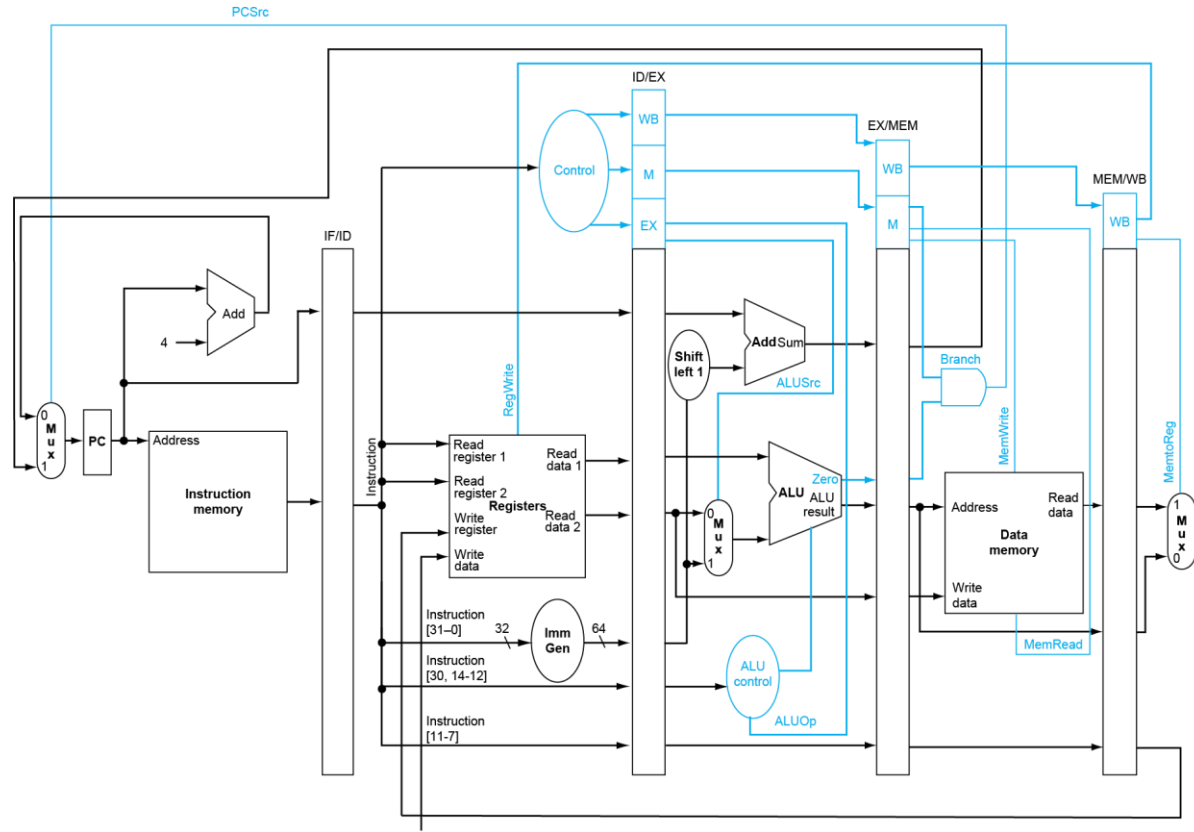
- RISC-V Pipeline
- **Pipeline Control**
- Hazards
 - Structural
 - Data
 - R-type instructions
 - Load
 - Control
- Superscalar processors

Pipelined Control

- Control signals derived from instruction
 - As in single-cycle implementation
 - Information is stored in pipeline registers for use by later stages



Pipelined Control



Hazards Ahead



Agenda

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

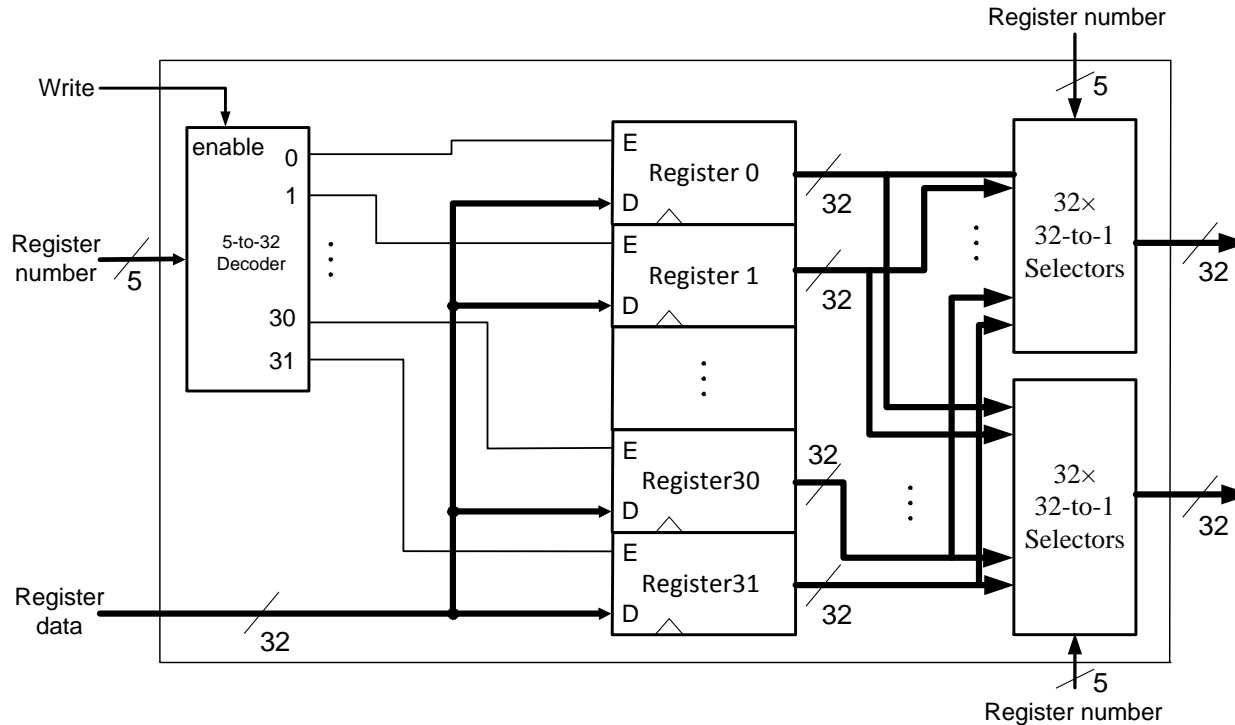
Structural Hazard

- **Problem:** Two or more instructions in the pipeline compete for access to a single physical resource
- **Solution 1:** Instructions take it in turns to use resource, some instructions have to stall
- **Solution 2:** Add more hardware to machine
 - Can always solve a structural hazard by adding more hardware

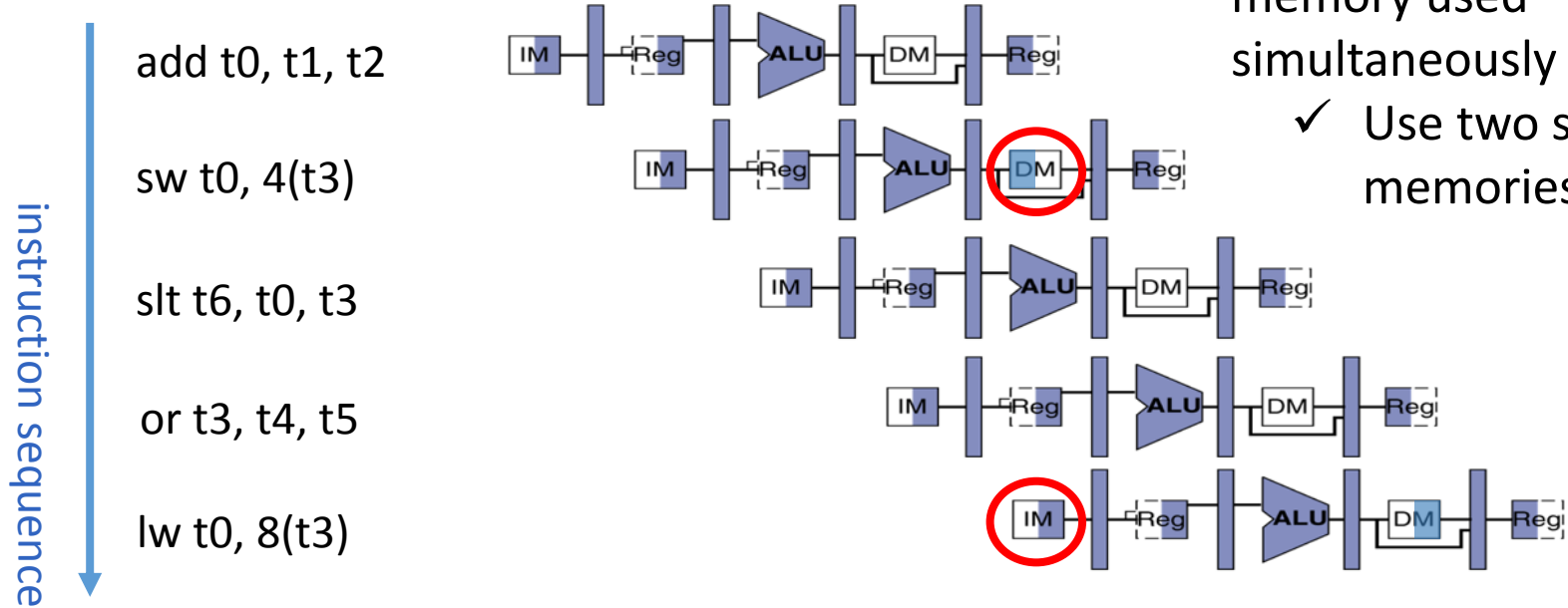
Regfile Structural Hazards

- 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

RISC-V (RV32) Register File

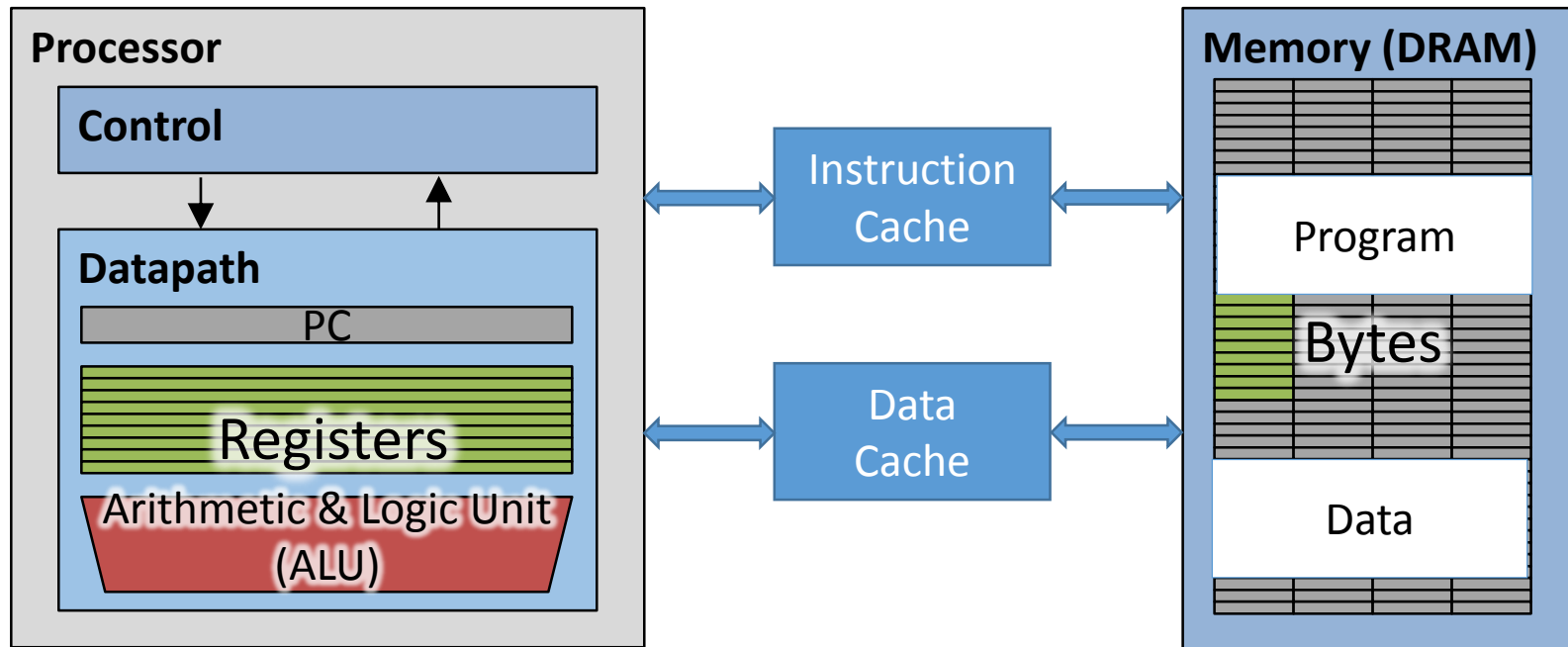


Structural Hazard: Memory Access



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

Instruction and Data Caches



Caches: small and fast “buffer” memories

Structural Hazards – Summary

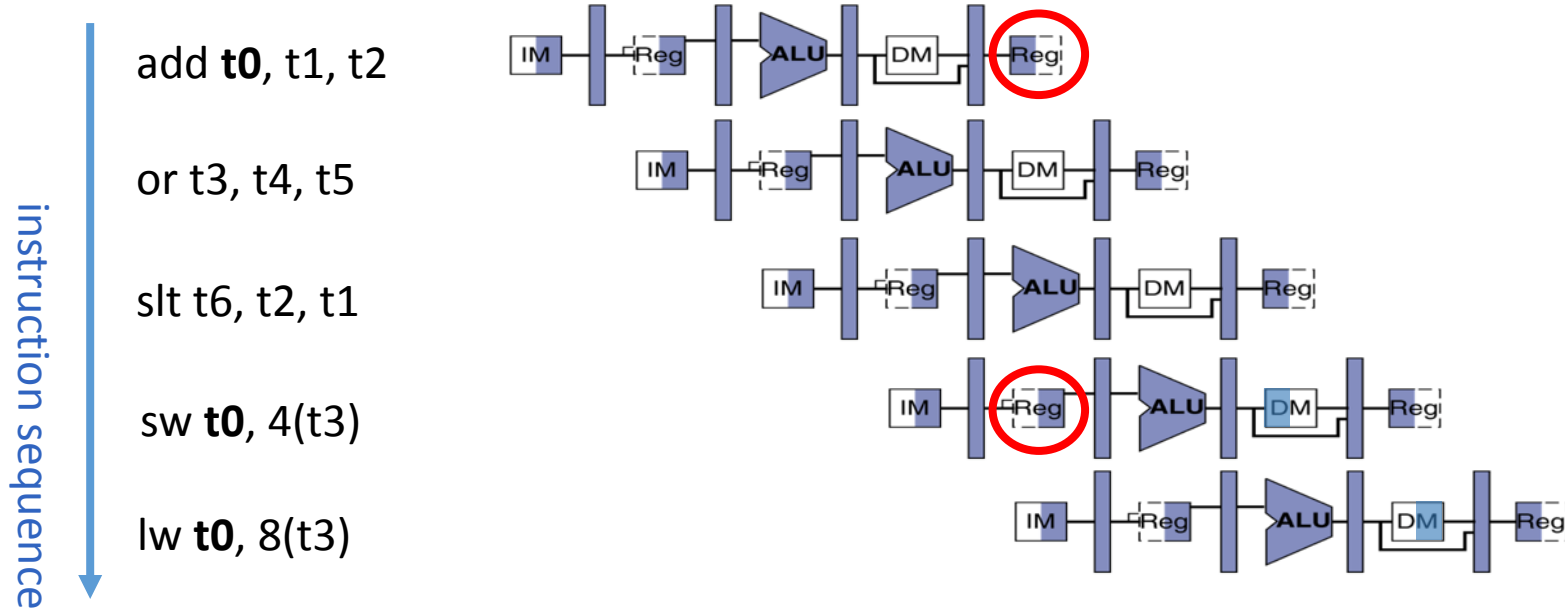
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
 - Load/store requires data access
 - Without separate memories, instruction fetch would have to *stall* for that cycle
 - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data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

Agenda

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

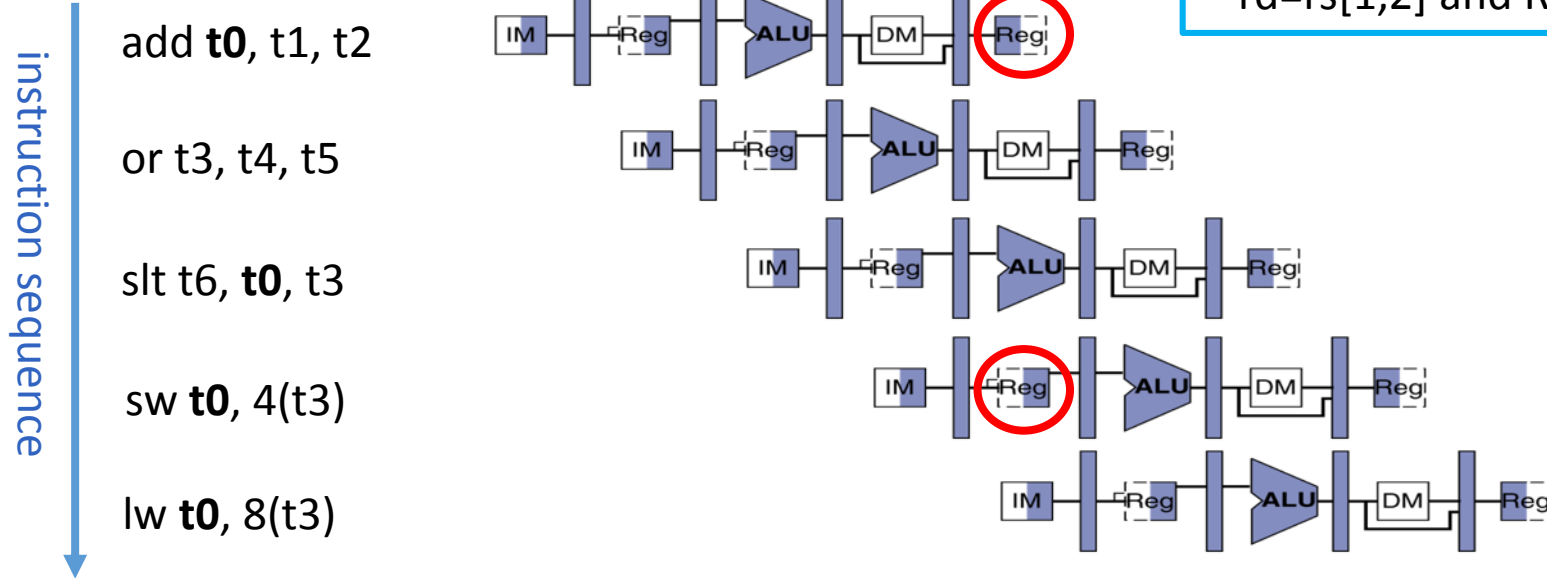
Data Hazard: Register Access

- Separate ports, but what if write to same value as read?
- Does **sw** in the example fetch the old or new value?



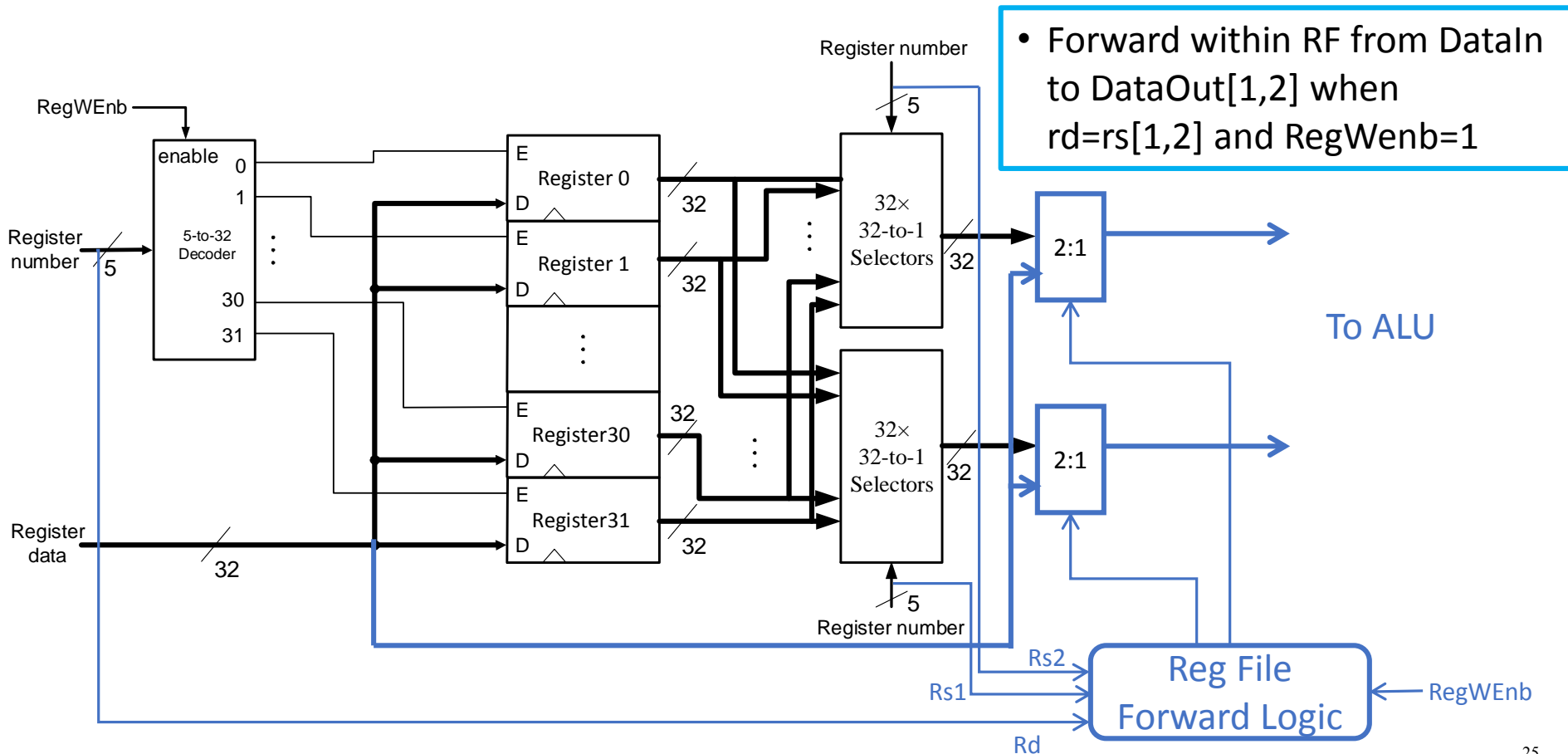
Register Access Policy

- Forward within RF from DataIn to DataOut[1,2] when $rd=rs[1,2]$ and $RegWenb=1$

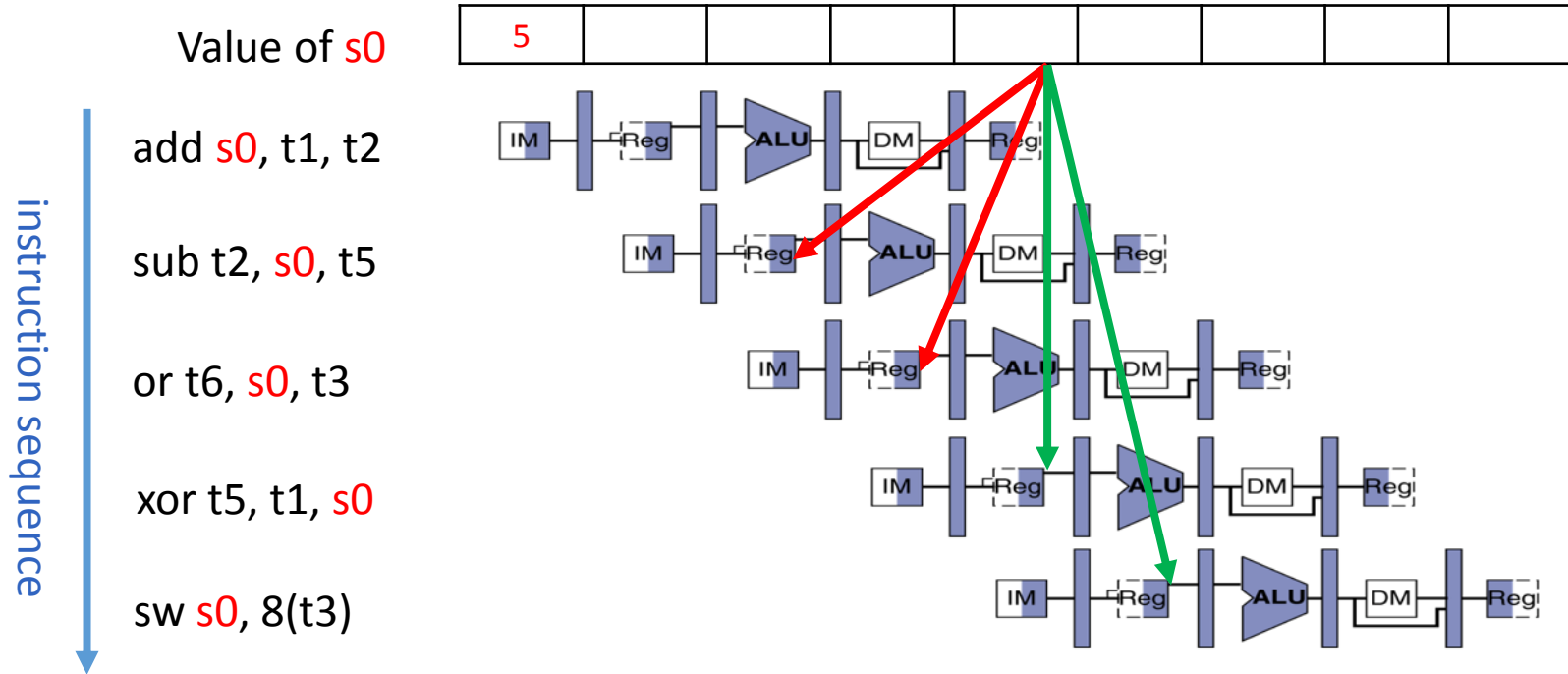


Might not always be possible to write then read in same cycle, especially in high-frequency designs. Check assumptions in any question.

RISC-V (RV32) Register File



Data Hazard: ALU Result

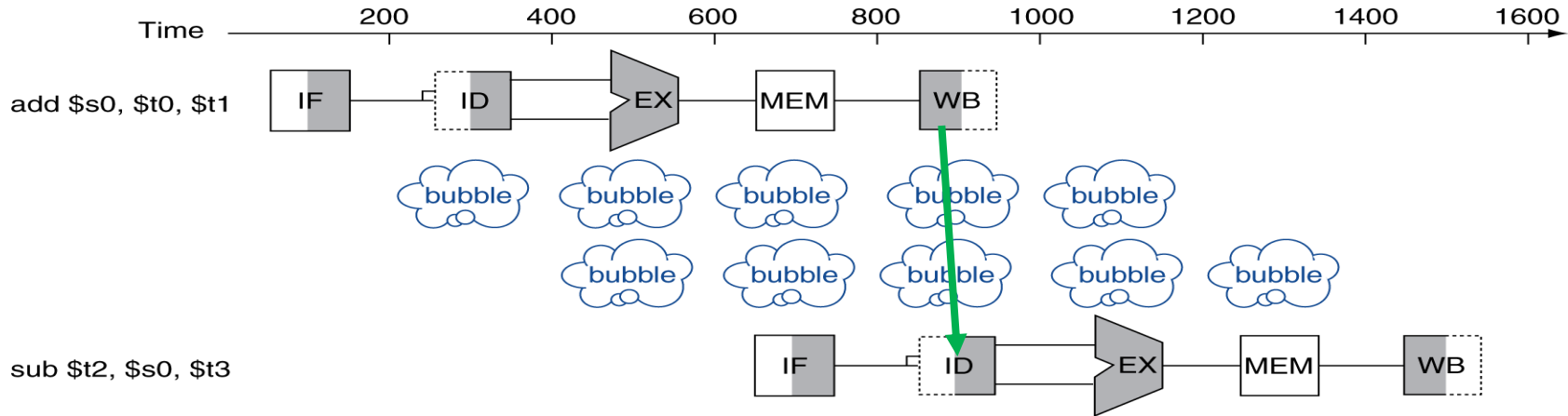


Without some fix, **sub** and **or** will calculate wrong result!

Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction

– add *s0*, t0, t1
 sub t2, *s0*, t3

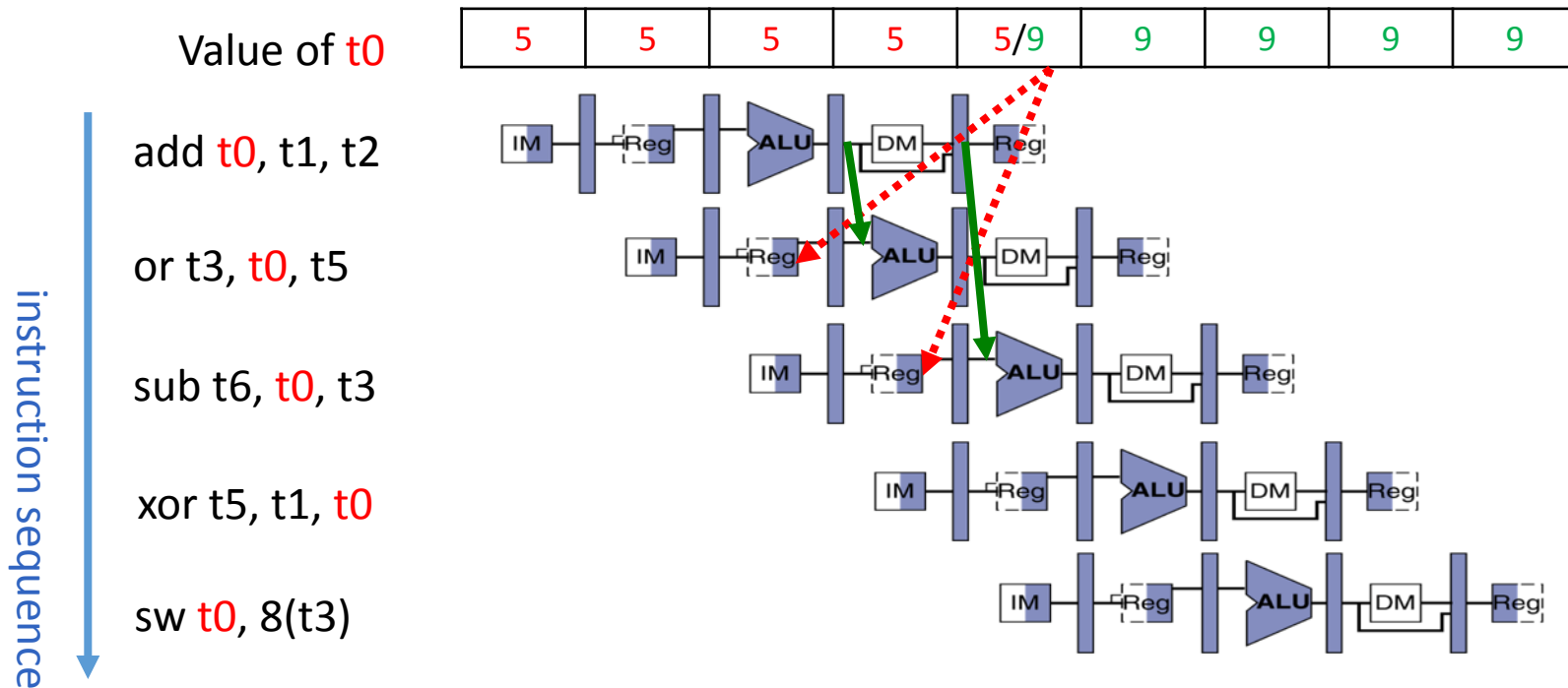


- Bubble:
 - effectively NOP: affected pipeline stages do “nothing”

Stalls and Performance

- Stalls reduce performance
 - But stalls are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
 - Requires knowledge of the pipeline structure

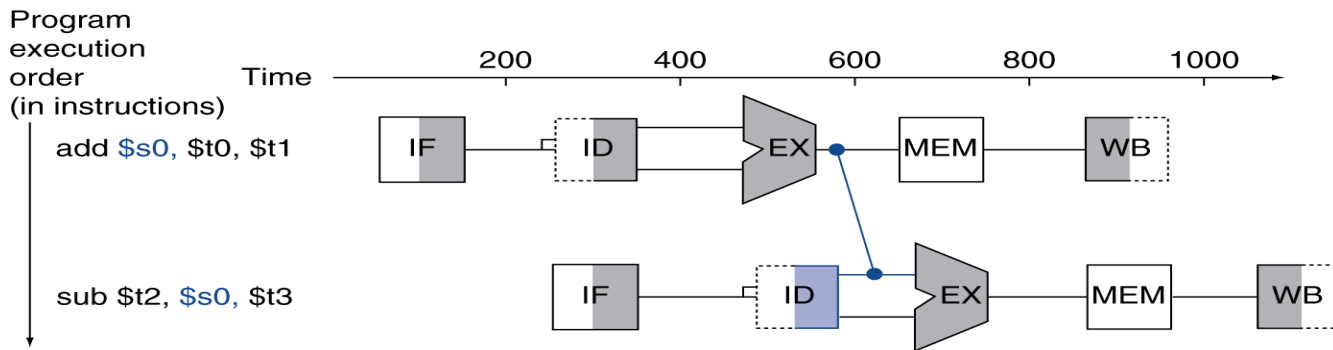
Solution 2: Forwarding



**Forwarding: grab operand from pipeline stage,
rather than register file**

Forwarding (aka Bypassing)

- Use result when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath



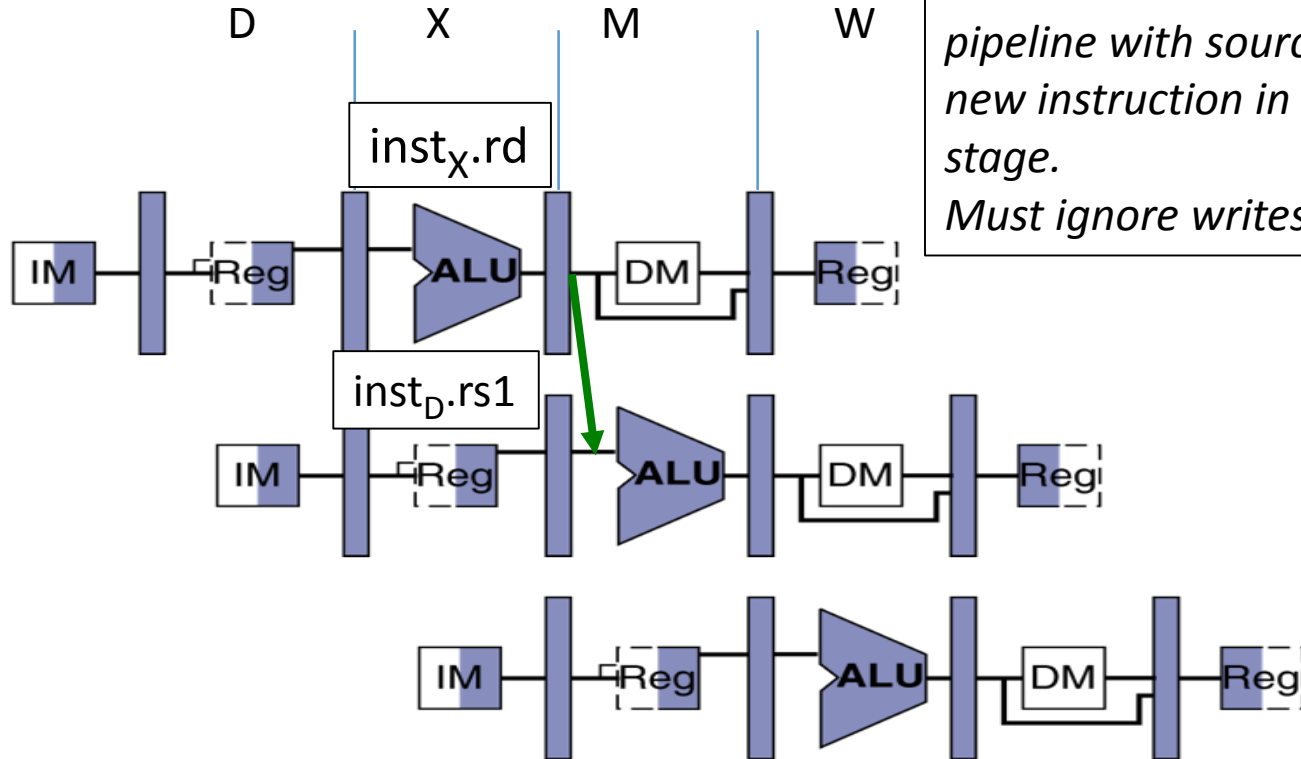
Detect Need for Forwarding (example)

*Compare destination of
older instructions in
pipeline with sources of
new instruction in decode
stage.
Must ignore writes to x0!*

add t0, t1, t2

or t3, t0, t5

sub t6, t0, t3



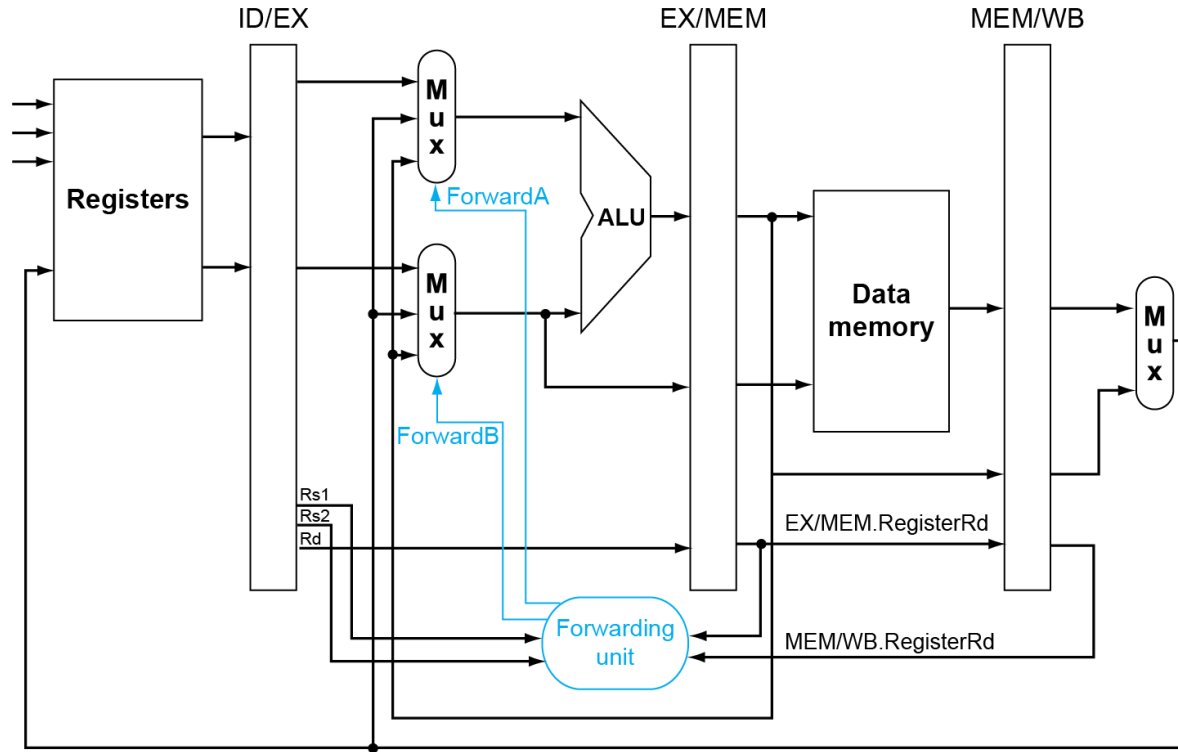
Data Hazards in ALU Instructions

- Consider this sequence:

```
sub    x2, x1, x3
and    x12, x2, x5
or     x13, x6, x2
add    x14, x2, x2
sd     x15, 100(x2)
```

- We can resolve hazards with forwarding
 - How do we detect when to forward?

Forwarding Paths



Detecting the Need to Forward

- Pass register numbers along pipeline
 - e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
 - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when
 - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
 - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
 - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
 - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

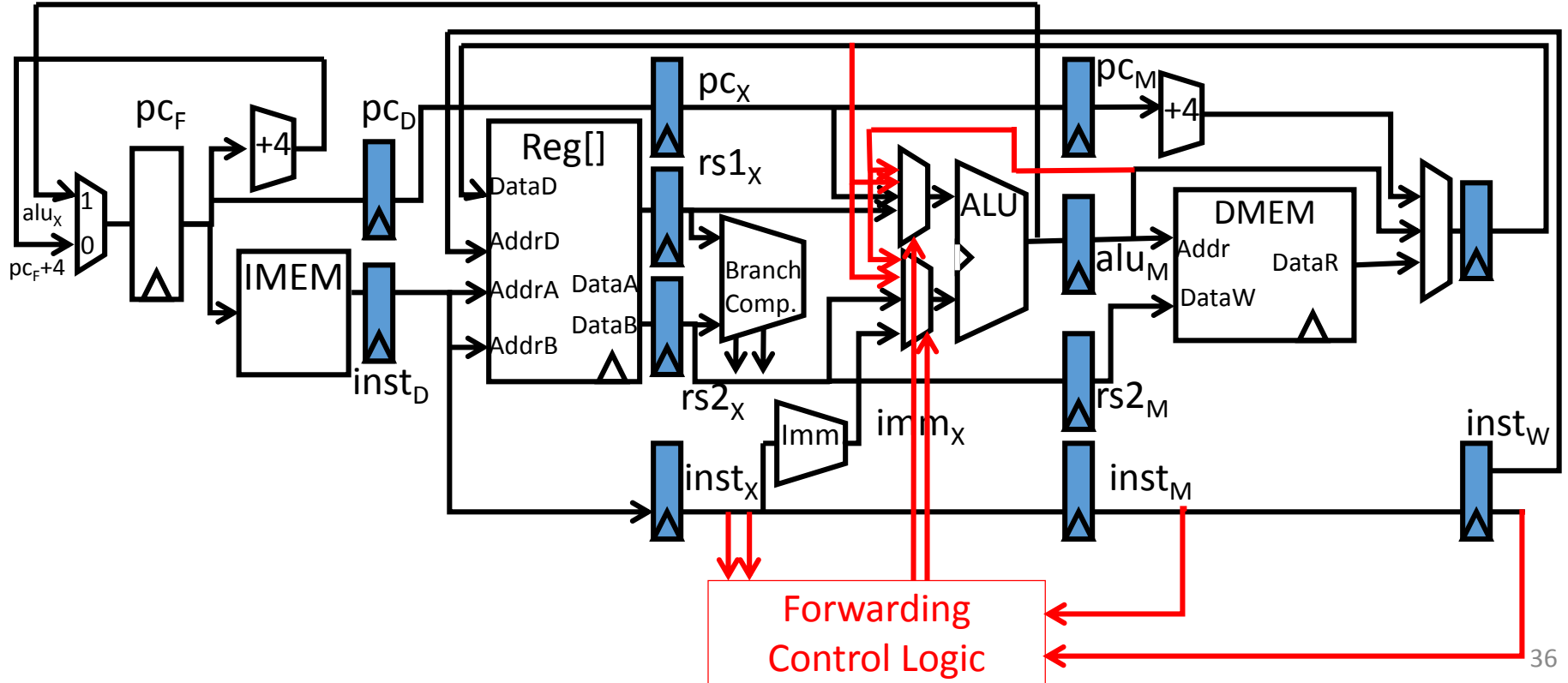
Fwd from
EX/MEM
pipeline reg

Fwd from
MEM/WB
pipeline reg

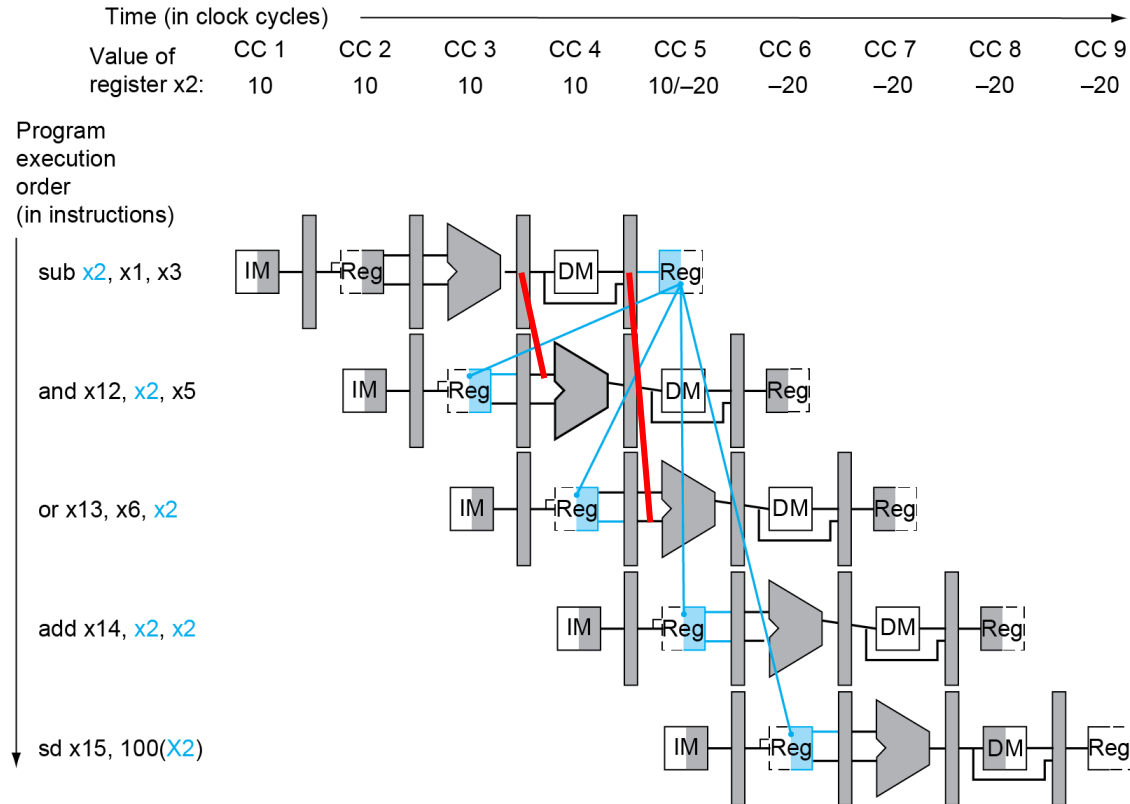
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
 - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not x0
 - EX/MEM.RegisterRd \neq 0,
MEM/WB.RegisterRd \neq 0

Forwarding Paths



Dependencies & Forwarding



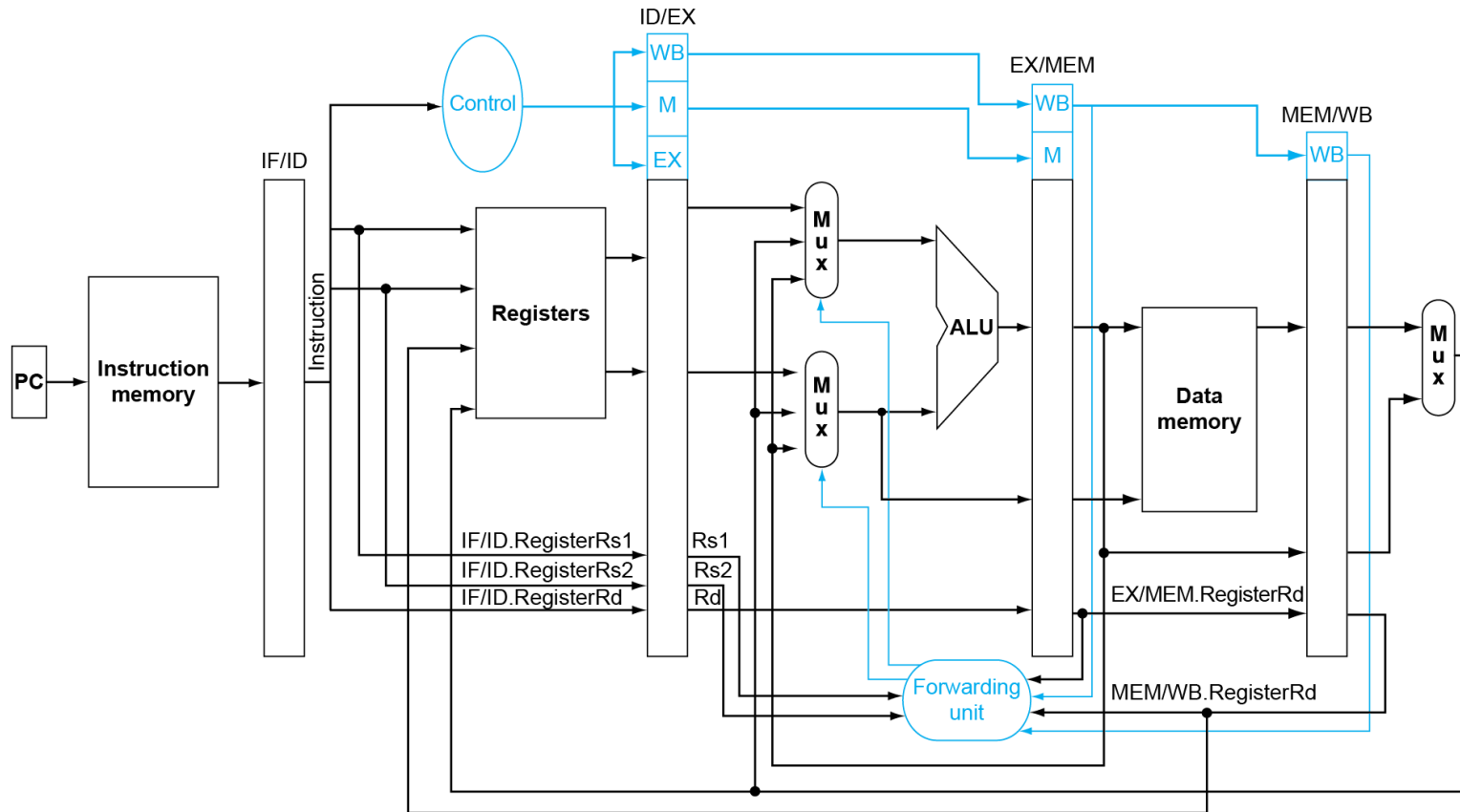
Double Data Hazard

- Consider the sequence:
 - add x1, x1, x2
 - add x1, x1, x3
 - add x1, x1, x4
- Both hazards occur
 - Want to use the most recent
- Revise MEM hazard condition
 - Only fwd if EX hazard condition isn't true

Revised Forwarding Condition

- MEM hazard
 - if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd \neq 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd \neq ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
 - if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd \neq 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd \neq ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

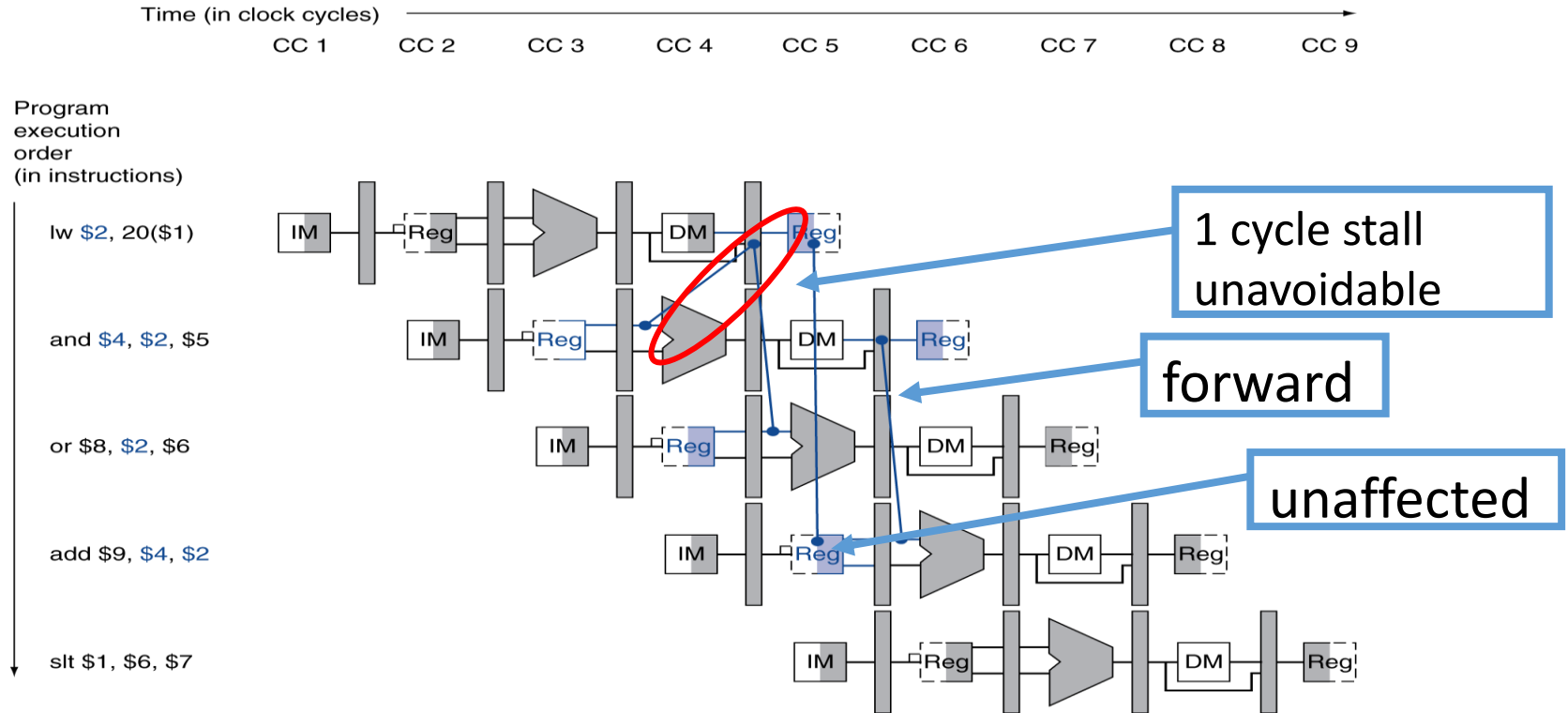
Datapath with Forwarding



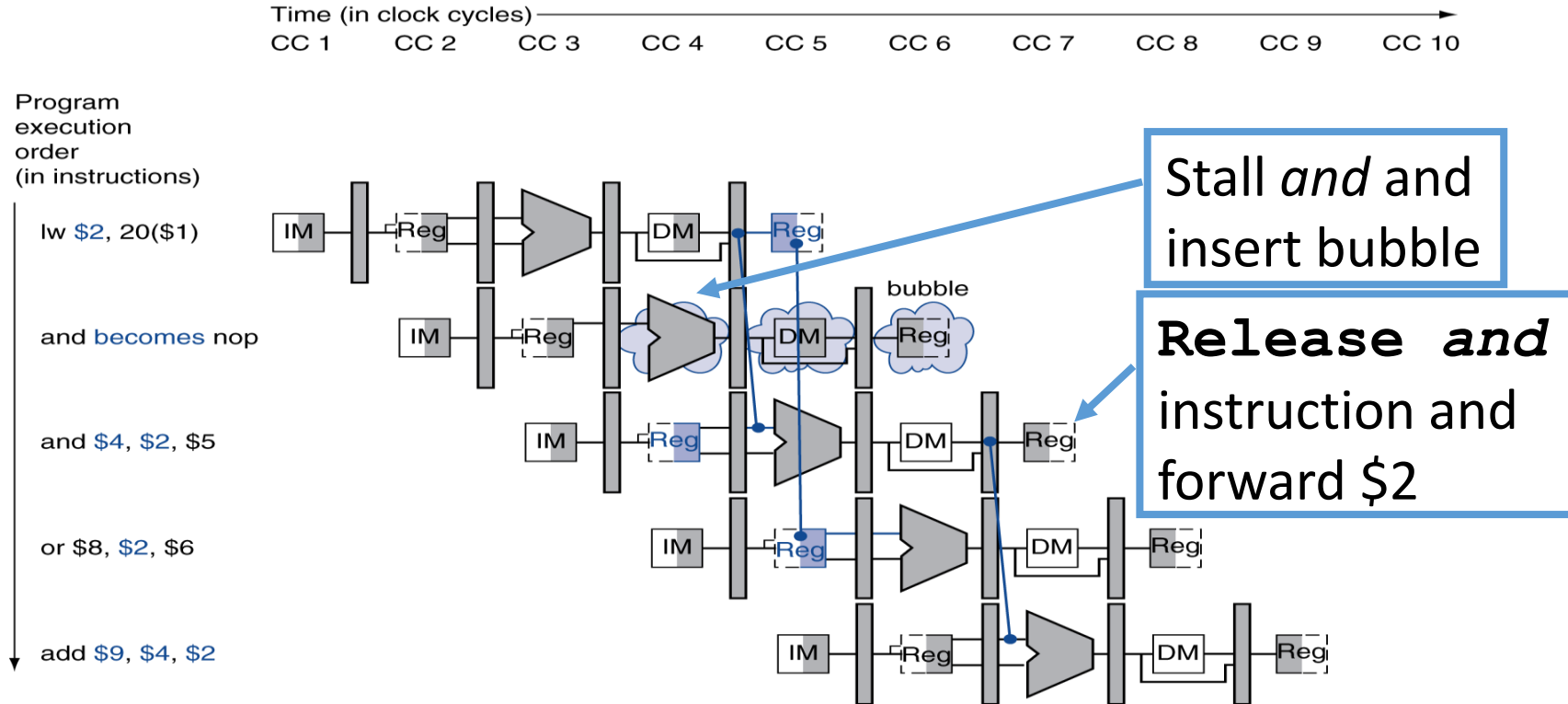
Agenda

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

Load Data Hazard



Stall Pipeline



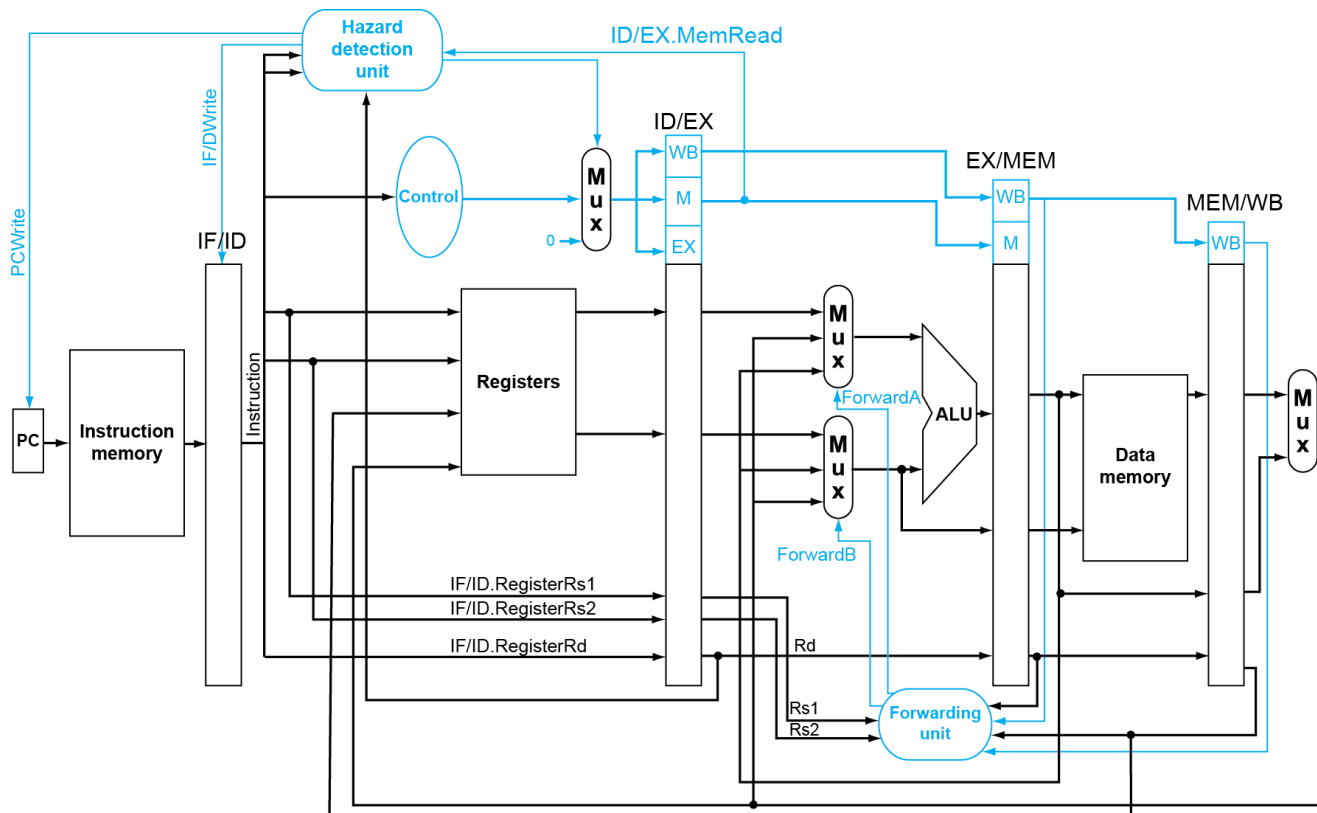
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
 - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
 - ID/EX.MemRead and
((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
(ID/EX.RegisterRd = IF/ID.RegisterRs1))
- If detected, stall and insert bubble

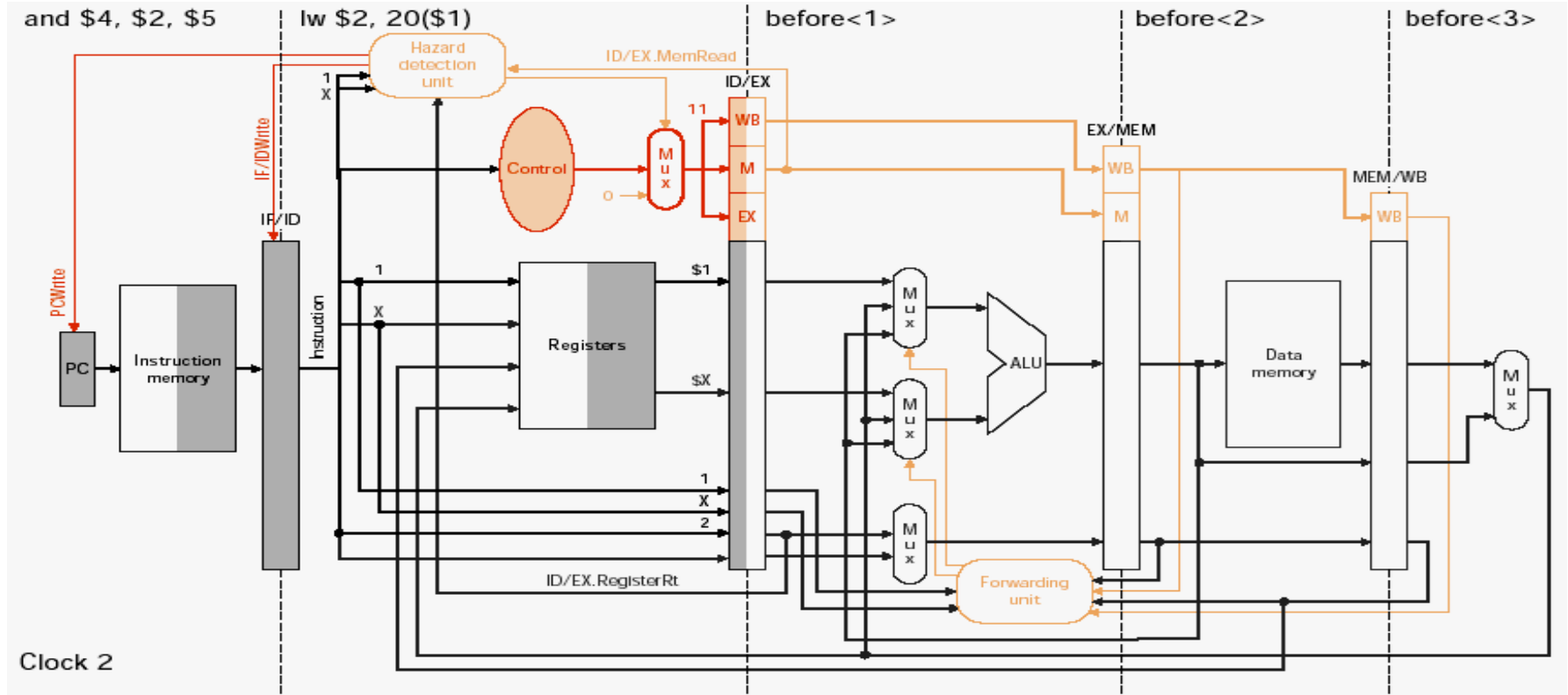
How to Stall the Pipeline

- Force control values in ID/EX register to 0
 - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
 - Using instruction is decoded again
 - Following instruction is fetched again
 - 1-cycle stall allows MEM to read data for 1d
 - Can subsequently forward to EX stage

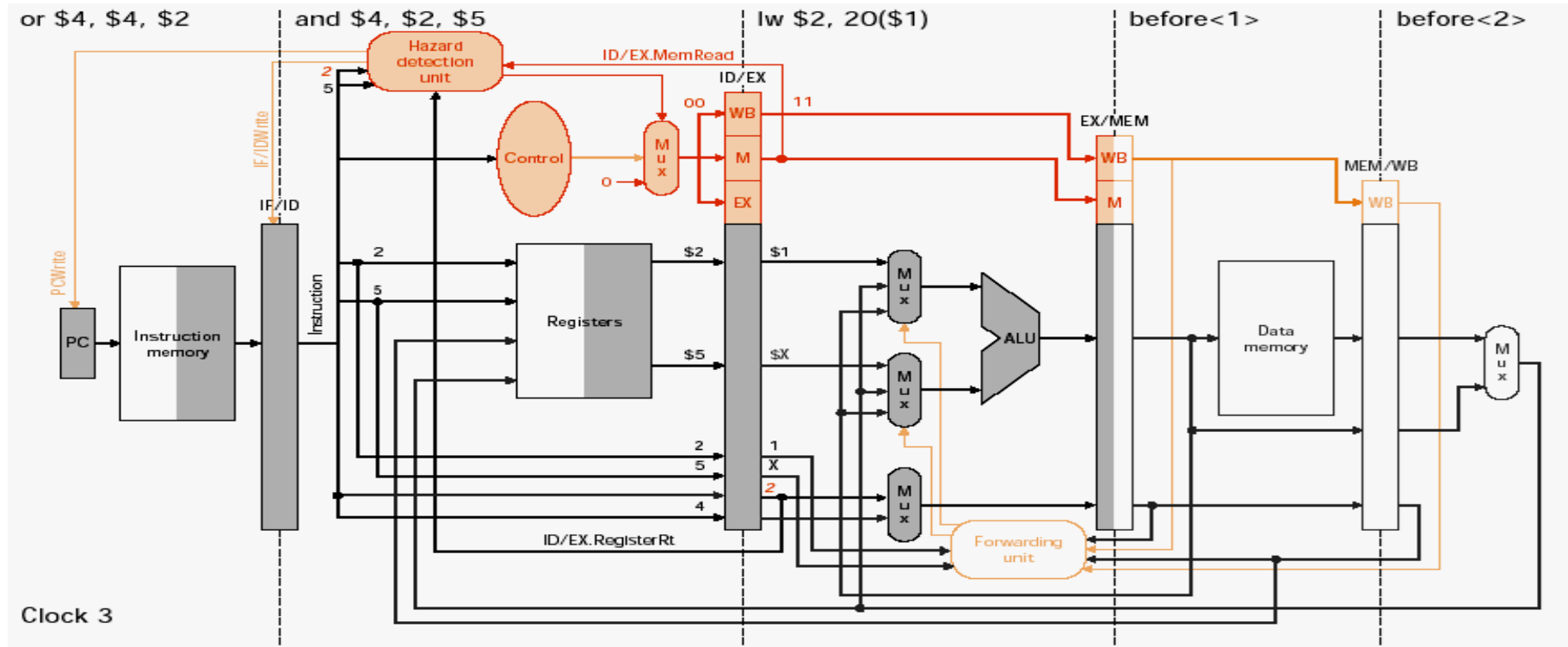
Datapath with Hazard Detection



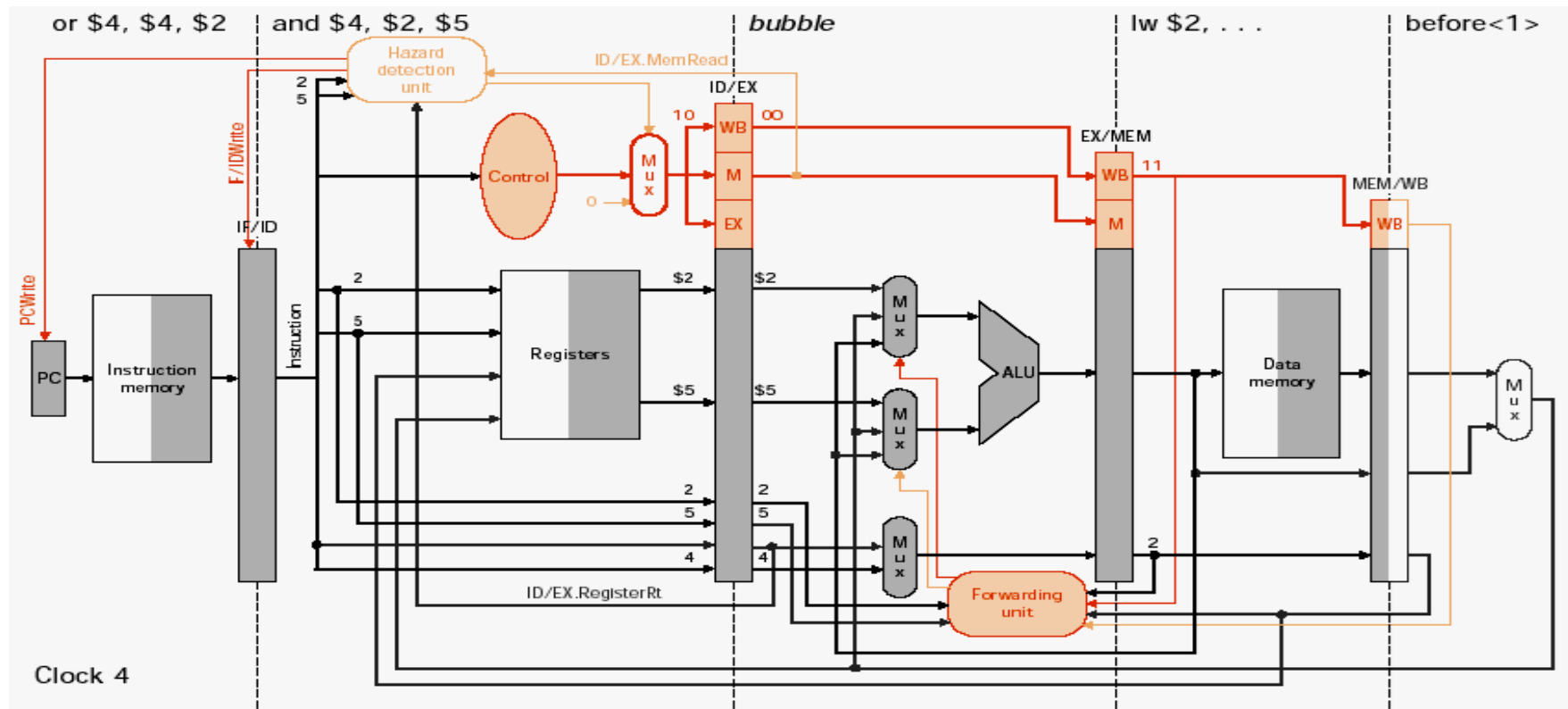
Example: Load is Decoded (1/4)



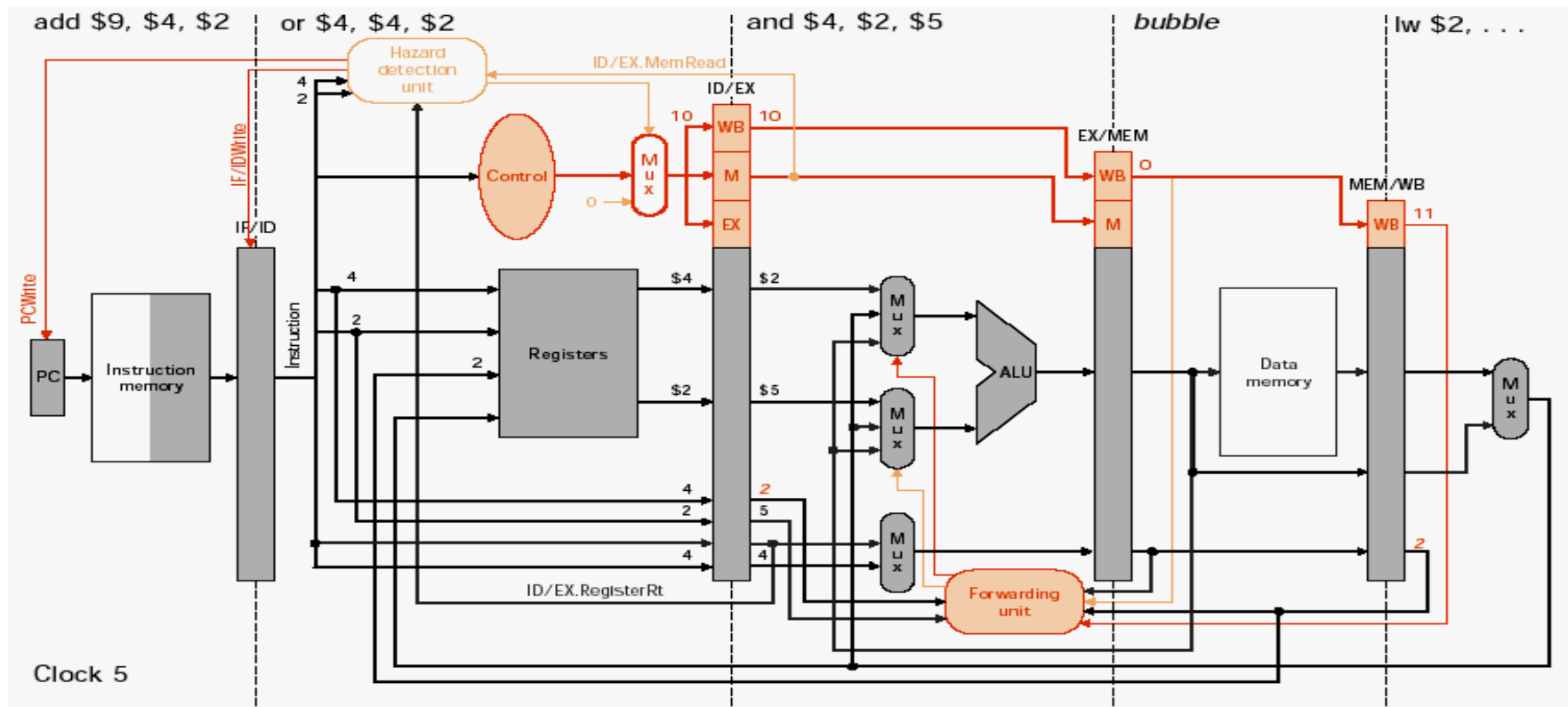
Example: Load Hazard is Detected (2/4)



Example: Bubble Inserted (3/4)



Example: Data Gets Forwarded (4/4)



1w Data Hazard

- 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:
 - Put unrelated instruction into load delay slot
 - No performance loss!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction!
- RISC-V code for $D=A+B$; $E=A+C$;

Original Order:

```
lw    t1, 0(t0)
lw    t2, 4(t0)
add   t3, t1, t2
sw    t3, 12(t0)
lw    t4, 8(t0)
add   t5, t1, t4
sw    t5, 16(t0)
```

Stall!

Stall!

13 cycles

Alternative:

```
lw    t1, 0(t0)
lw    t2, 4(t0)
lw    t4, 8(t0)
add   t3, t1, t2
sw    t3, 12(t0)
add   t5, t1, t4
sw    t5, 16(t0)
```

11 cycles

Agenda

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

Control Hazards

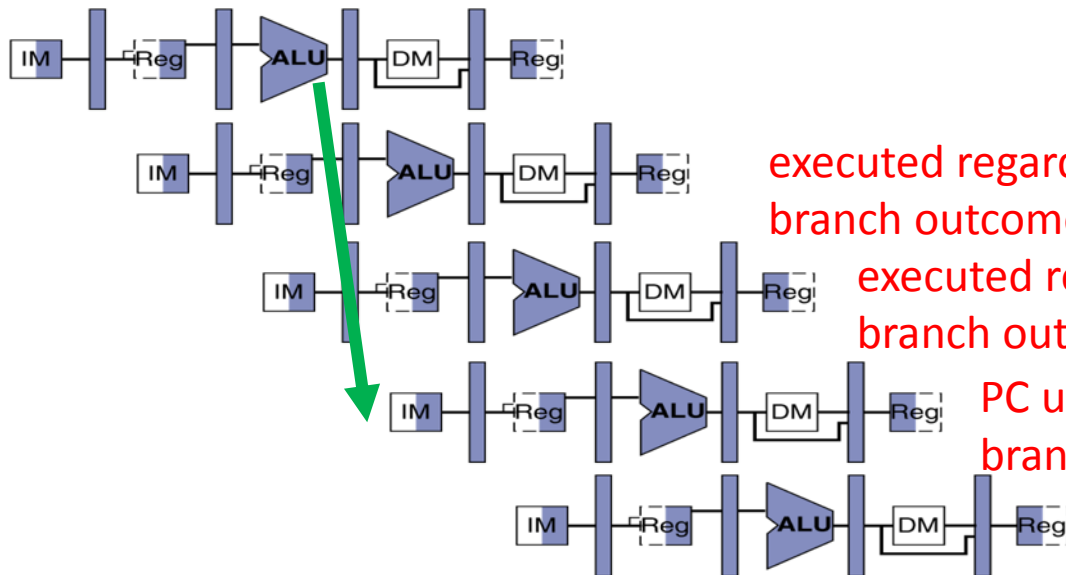
beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

xor t5, t1, s0

sw s0, 8(t3)



Observation

- If branch not taken, then instructions fetched sequentially after branch are correct
- If branch or jump is taken, then need to **flush** incorrect instructions from pipeline by converting to NOPs

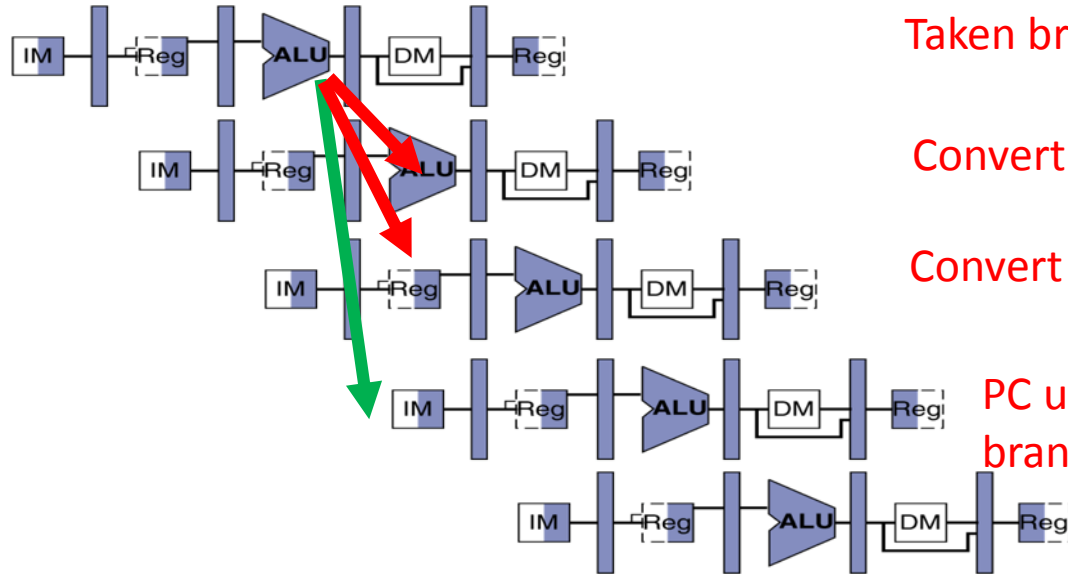
Kill Instructions after Branch if Taken

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

Reducing Branch Penalties

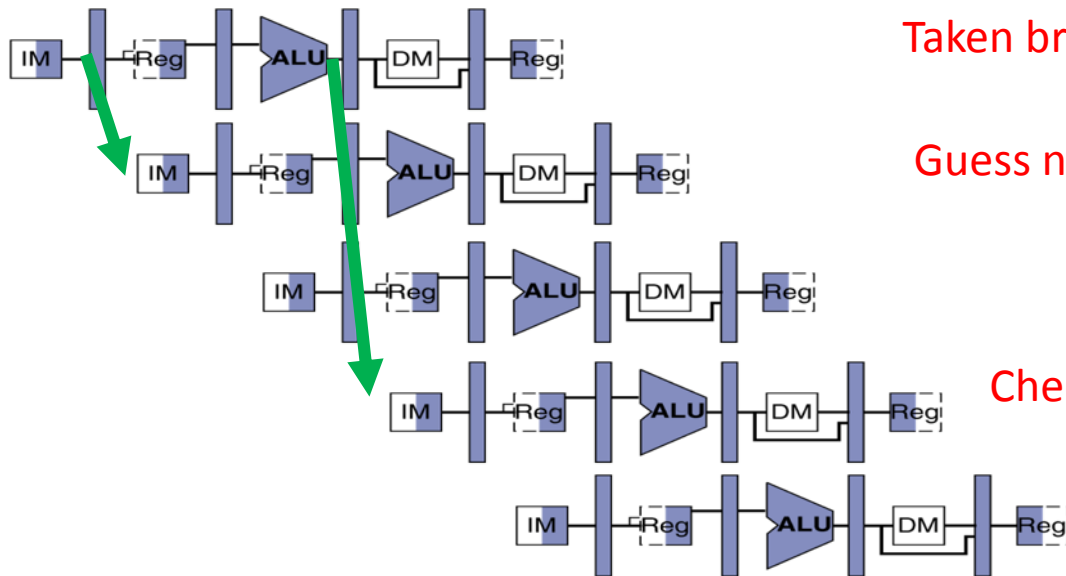
- Every taken branch in simple pipeline costs 2 dead cycles
- To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
- Only flush pipeline if branch prediction was incorrect

Branch Prediction

beq t0, t1, label

label:

.....



Taken branch

Guess next PC!

Check guess correct

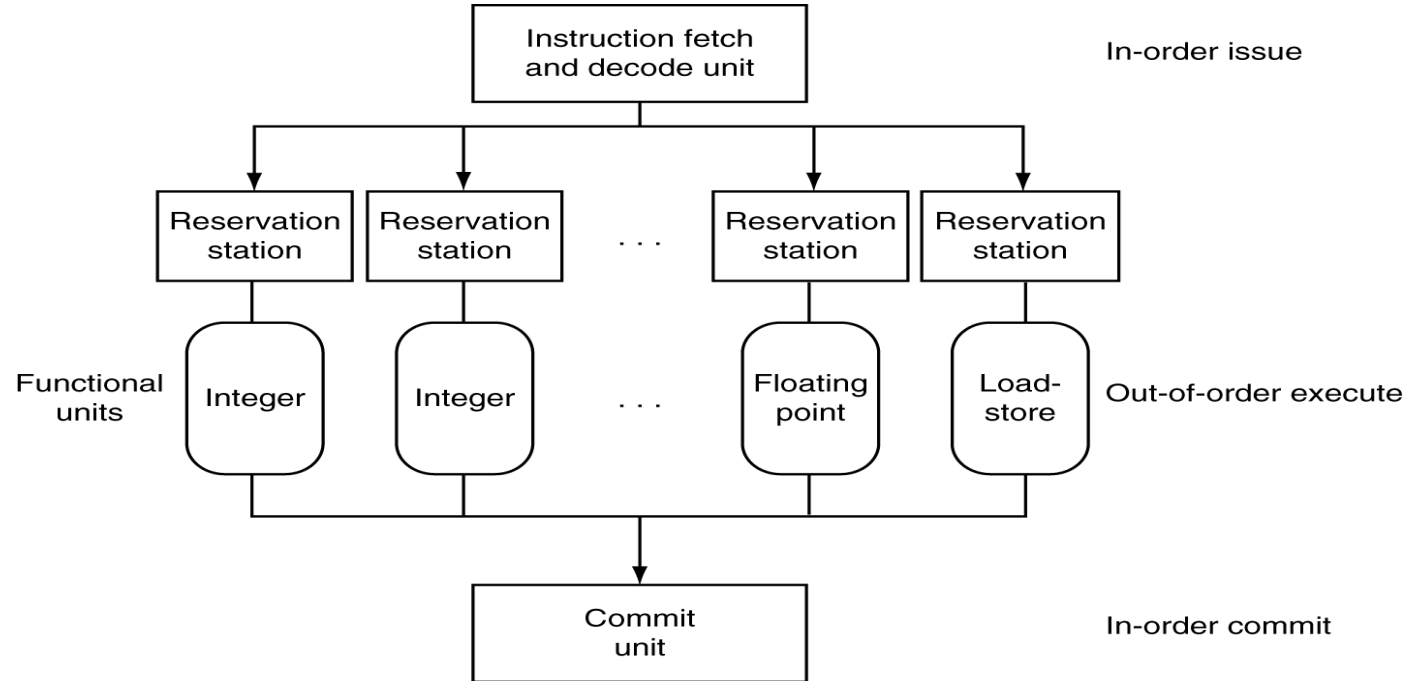
Agenda

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

Increasing Processor Performance

1. Clock rate
 - Limited by technology and power dissipation
2. Pipelining
 - “Overlap” instruction execution
 - Deeper pipeline: 5 \Rightarrow 10 \Rightarrow 15 stages
 - Less work per stage \rightarrow shorter clock cycle
 - But more potential for hazards ($CPI > 1$)
3. Multi-issue “super-scalar” processor
 - Multiple execution units (ALUs)
 - Several instructions executed simultaneously
 - $CPI < 1$ (ideally)

Superscalar Processor

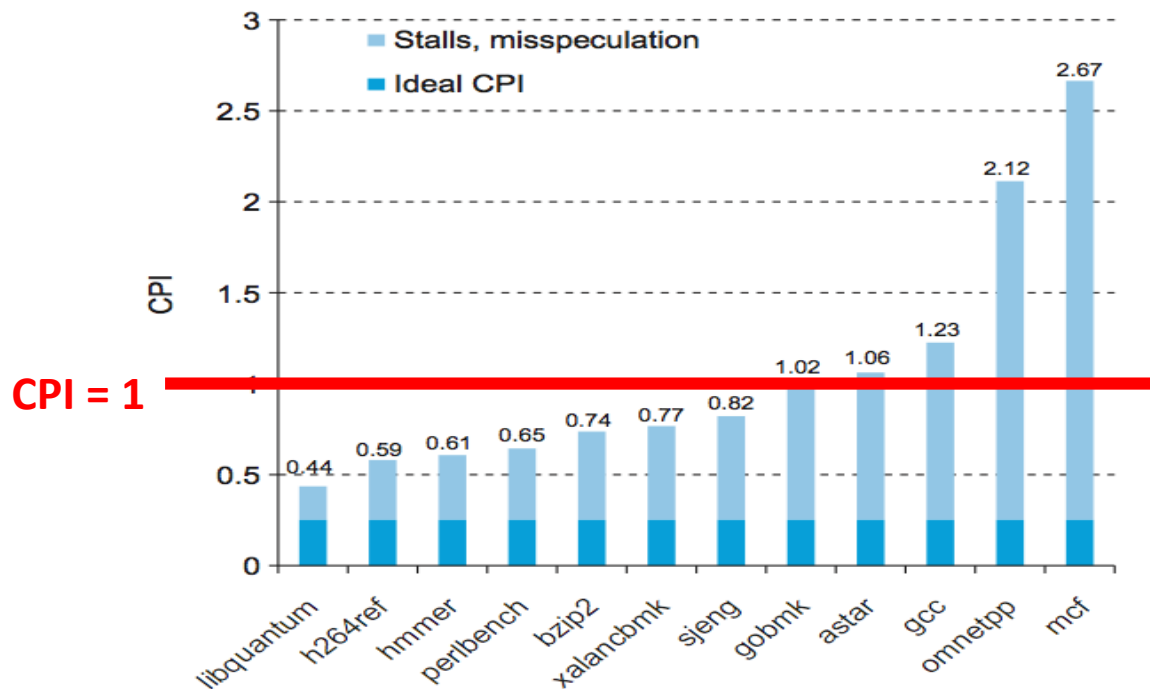


P&H p. 340

Superscalar Processor

- Multiple issue “superscalar”
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - $CPI < 1$, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak $CPI = 0.25$, peak $IPC = 4$
 - Dependencies reduce this in practice
- “Out-of-Order” execution
 - Reorder instructions dynamically in hardware to reduce impact of hazards
- *Technion 046267/234267 discuss these techniques!*

Benchmark: CPI of Intel Core i7



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

P&H p. 350

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
 - All instructions are 32-bits
 - Easy to fetch and decode in one cycle
 - Versus x86: 1- to 15-byte instructions
 - Few and regular instruction formats
 - Decode and read registers in one step
 - Load/store addressing
 - Calculate address in 3rd stage, access memory in 4th stage
 - Alignment of memory operands
 - Memory access takes only one cycle

In Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions
- All pipeline stages have same duration
 - Choose partition that accommodates this constraint
- Hazards potentially limit performance
 - Maximizing performance requires programmer/compiler assistance
 - E.g. Load and Branch delay slots
- Superscalar processors use multiple execution units for additional instruction level parallelism
 - Performance benefit highly code dependent