ITU VLSI LABS

Very Large Scale Integration II - VLSI II

Basic Pipelining

Berna Örs Yalçın

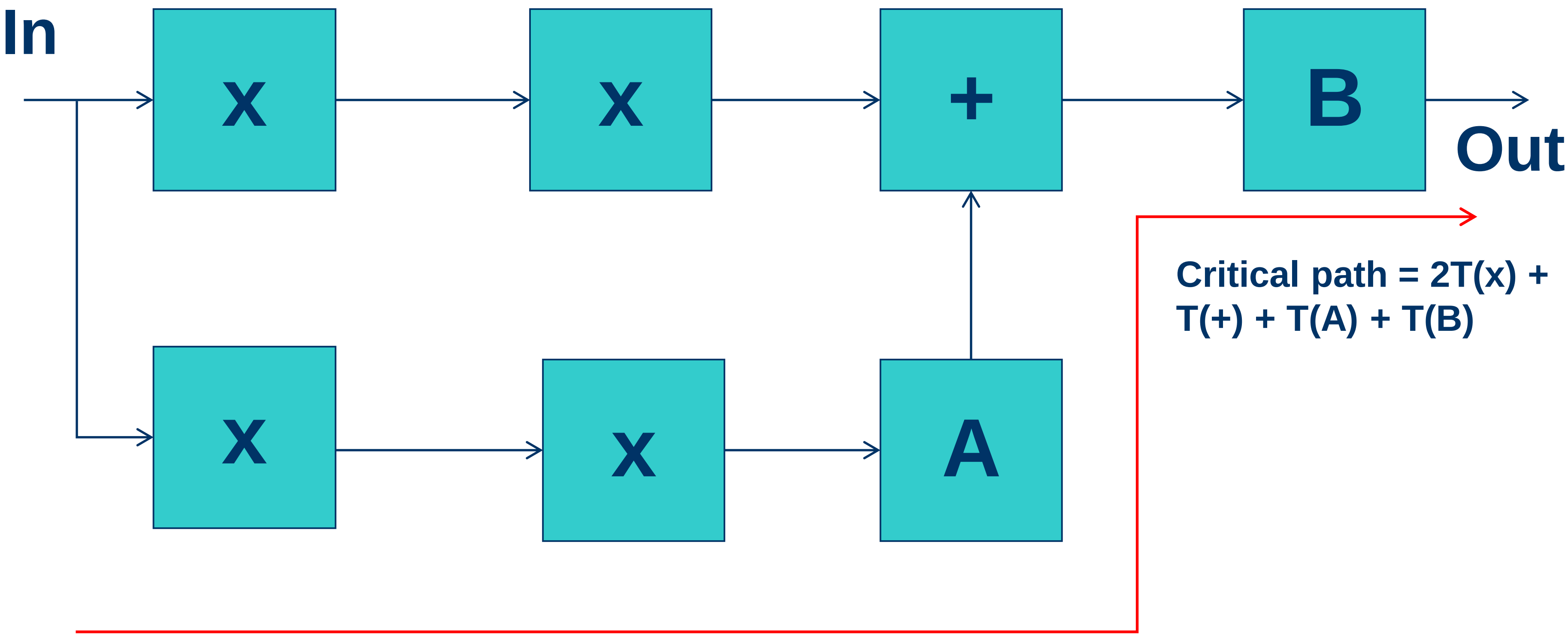
ITU VLSI Laboratories

Istanbul Technical University

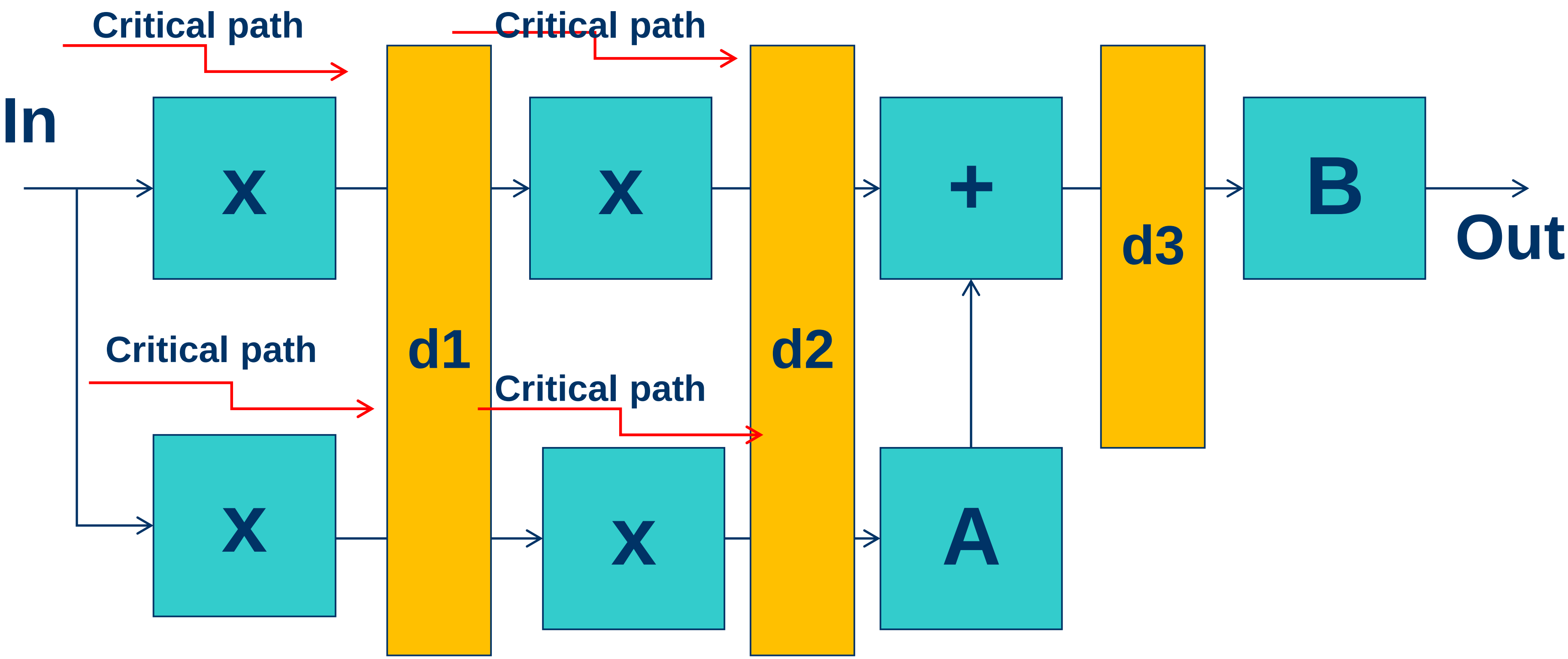
Basic Pipelining

- Today's topics:
  - Pipeline: General Concept
  - Pipelineing in Processors
  - Pipeline Hazards
    - Structural Hazards
    - Data Hazards
    - Control Hazards

General Concept

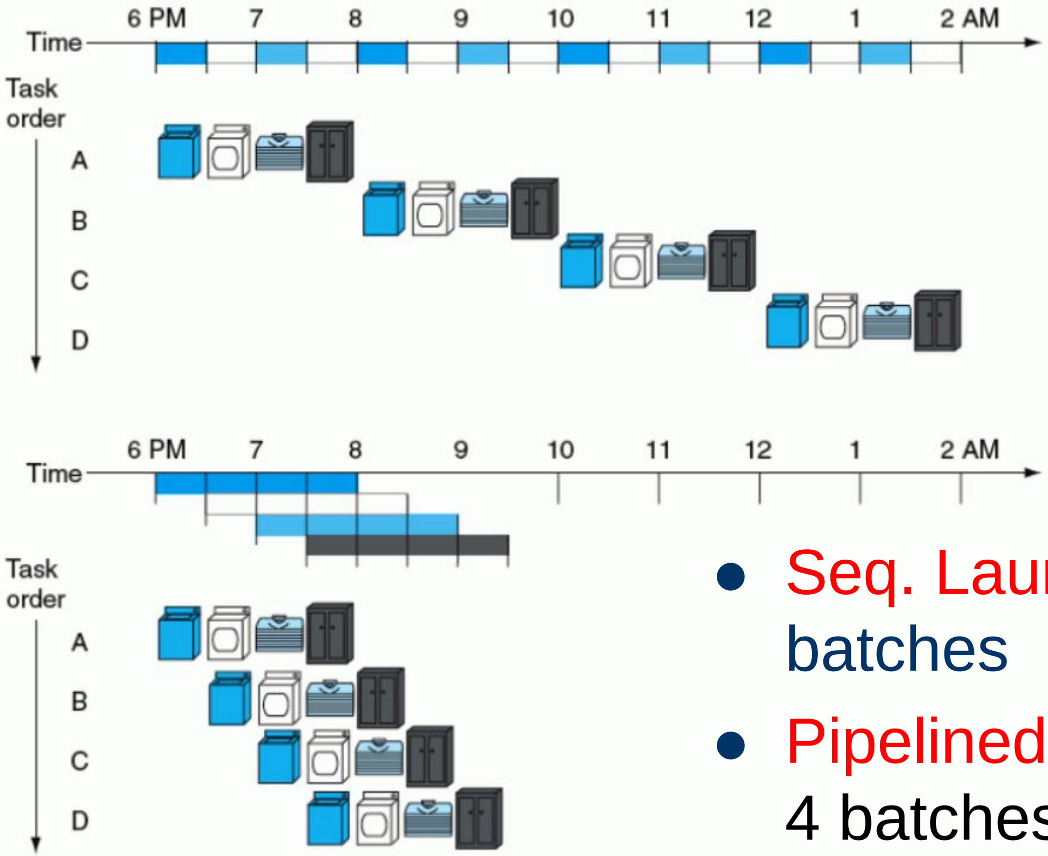


General Concept



Assuming T(x) > T(A) + T(+) and T(x) > T(B) , New critical path is T(x)

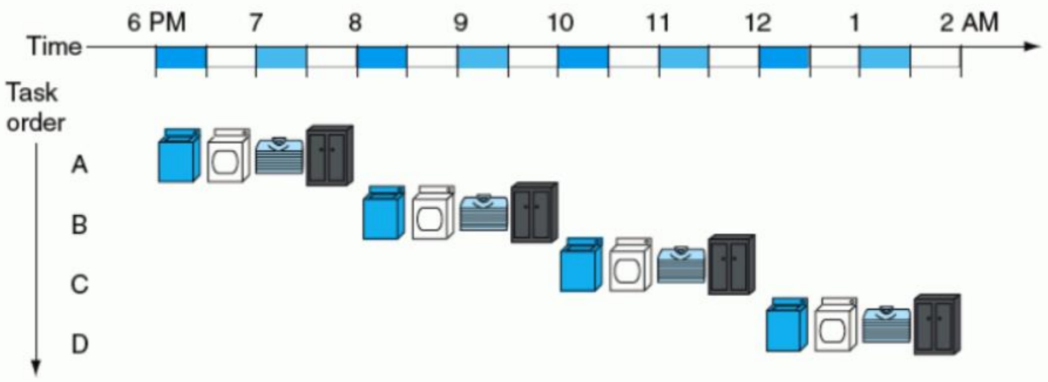
General Concept: An Analogy



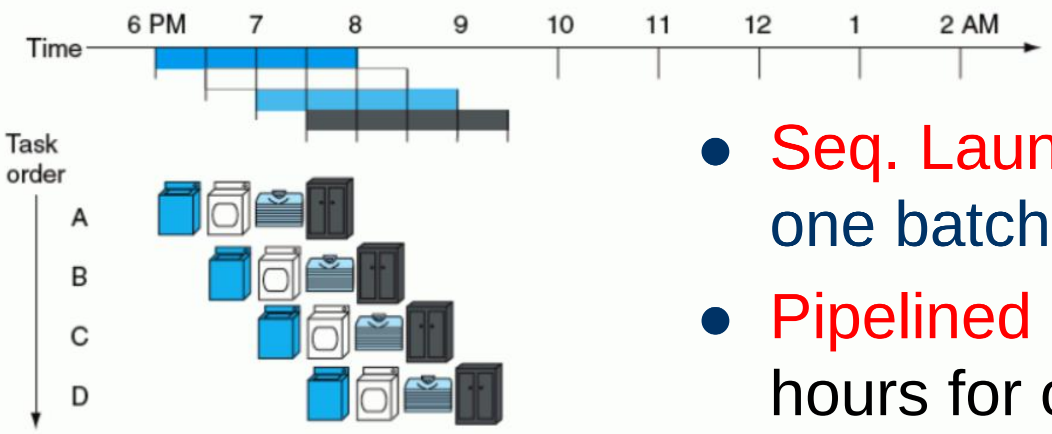
- Time for handling one batch of clothes is not changed
- But we are doing jobs in PARALELL

- Seq. Laundry = 8 hours for 4 batches
- Pipelined Laundry = 3.5 hours for 4 batches!

General Concept: An Analogy



- 8/3.5 = 2.3 times speedup
- Potential 4 times increase

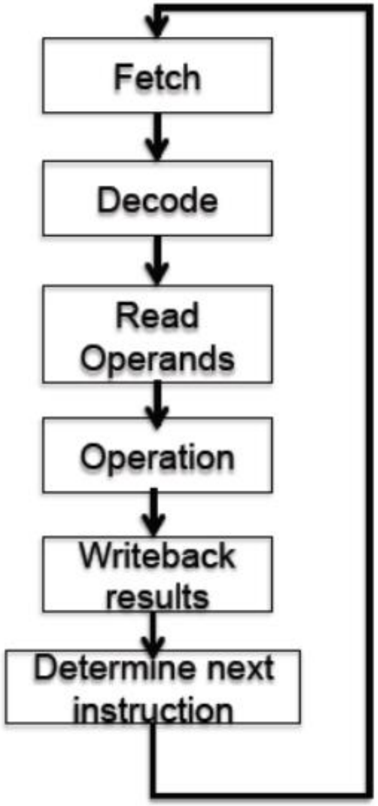


- Seq. Laundry = Adds 2 hours for one batch
- Pipelined Laundry = Adds 0.5 hours for one batch

General Concept

- Benefits?
  - Increase in clock frequency
  - Data for each stage will be held at the stage’s register line, this will allow us to exploit a form of paralellism – (increase throughput)

Pipelineing in Processors: Basic execution loop



Instruction execution is not a monolithic action. There are multiple micro-actions involved…

- Five stages, one step per stage

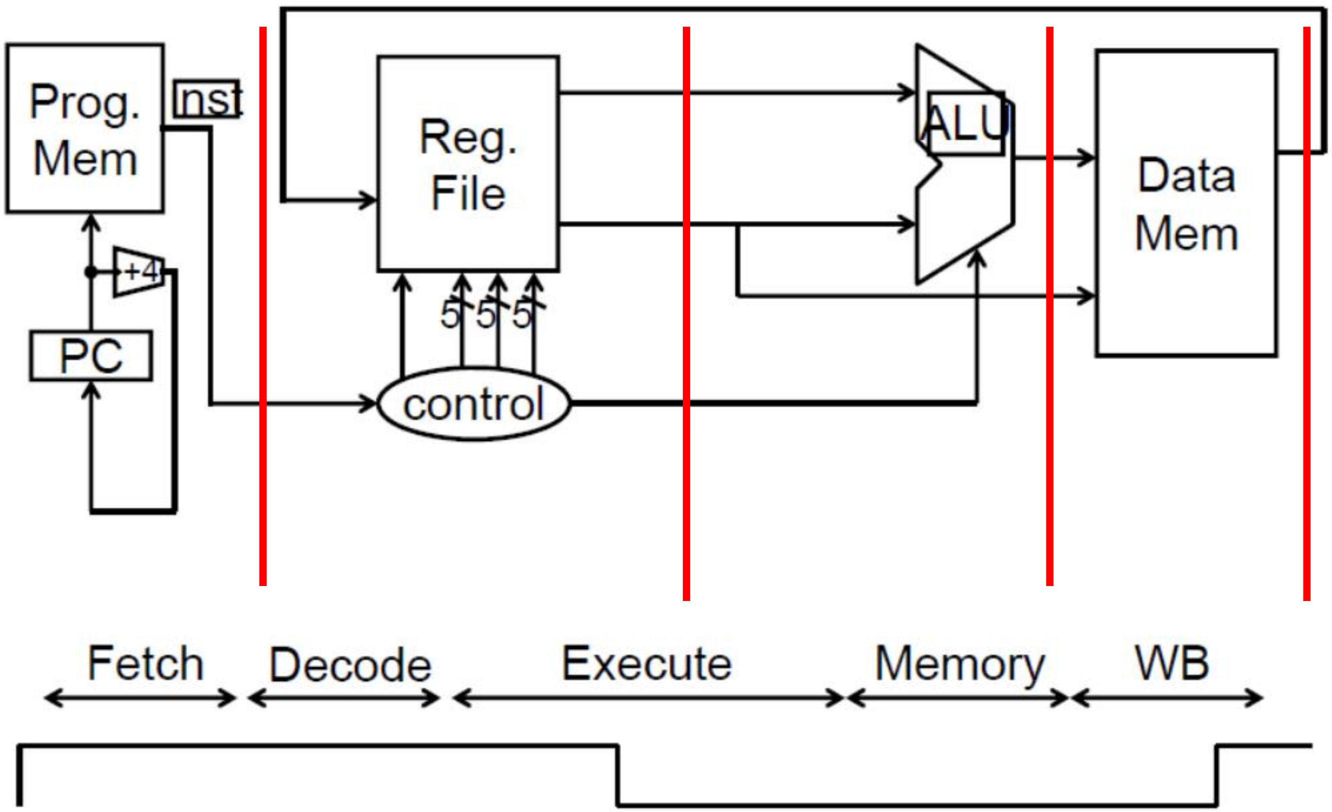
1. IF: Instruction fetch from memory

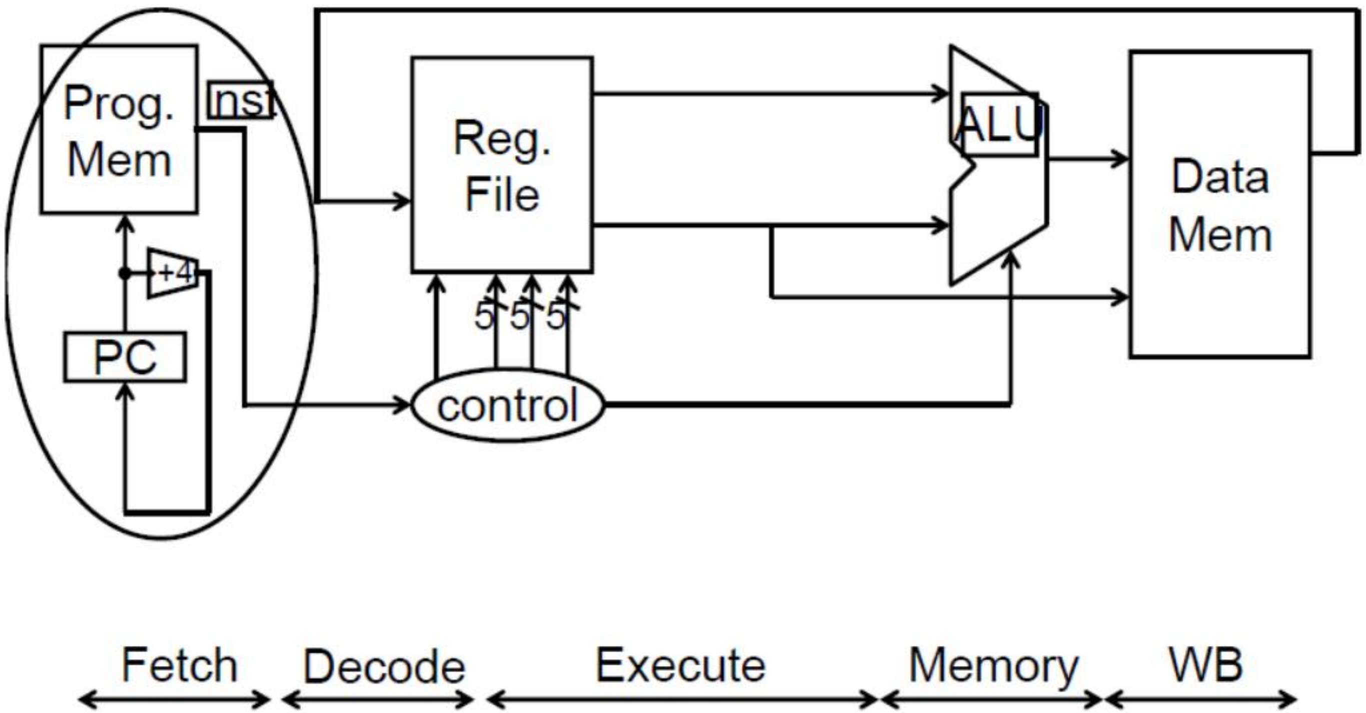
2. ID: Instruction decode & register read

3. EX: Execute operation or calculate address

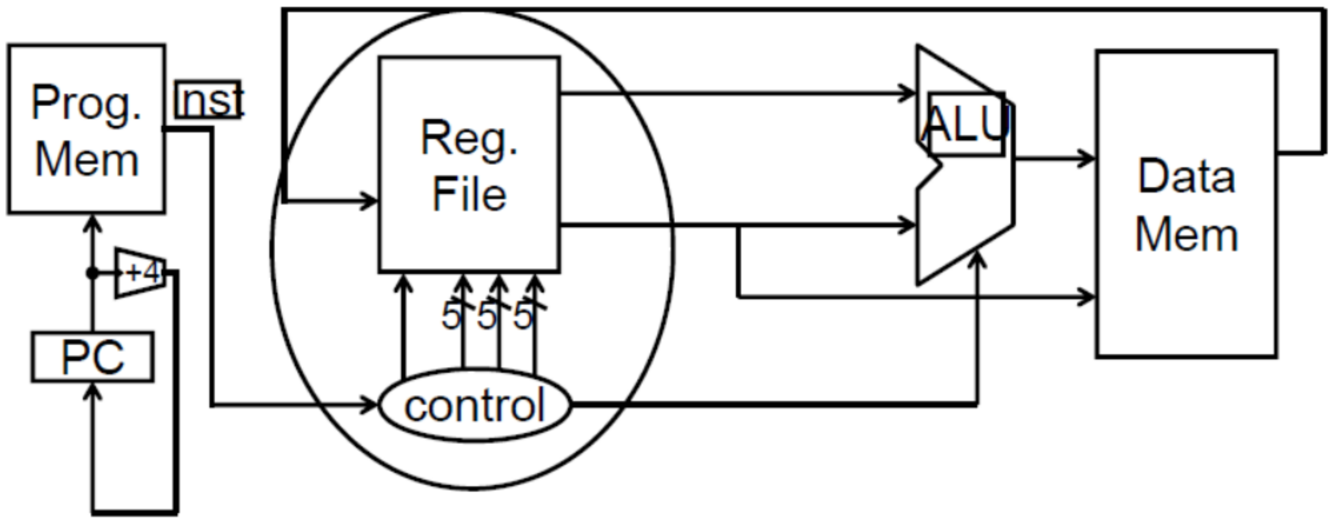
4. MEM: Access memory operand

5. WB: Write result back to register


Prog.
Mem
Inst
+4
PC
Reg.
File
5 5 5
control
ALU
Data
Mem
Fetch
Decode
Execute
Memory
WB

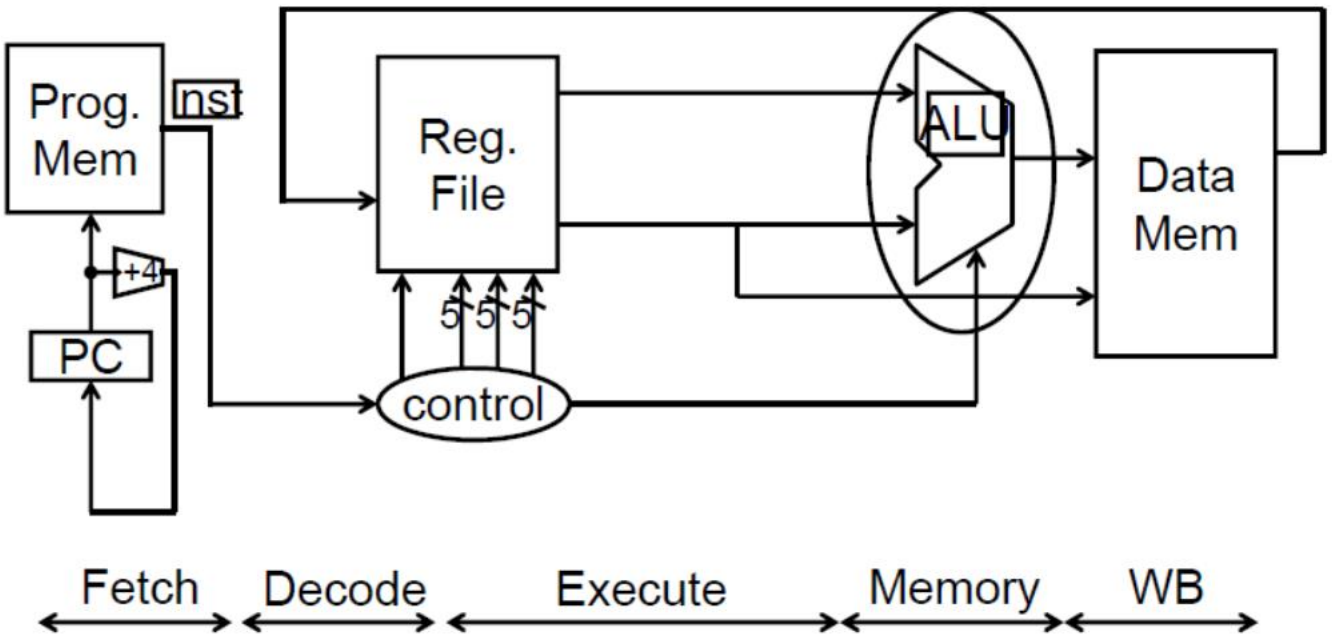


Fetch 32-bit instruction from memory
Increment PC = PC + 4



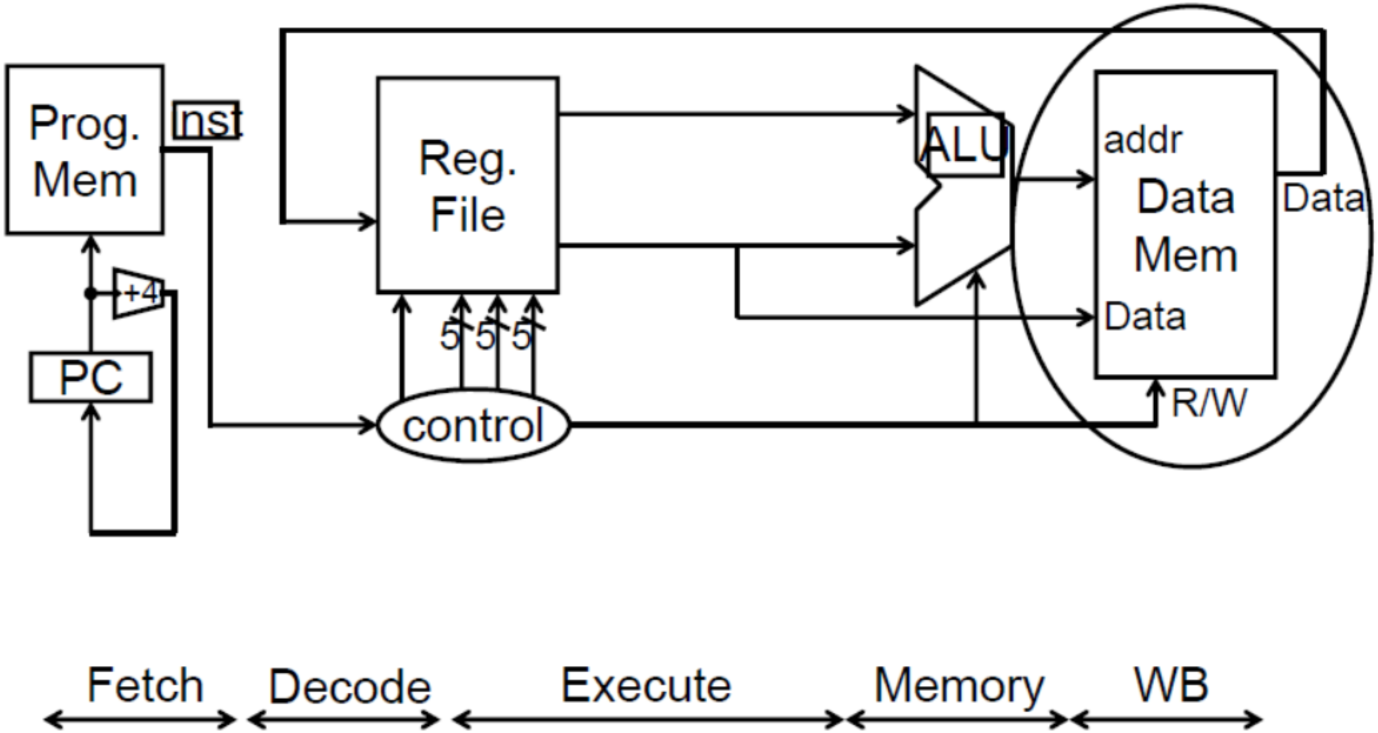
Fetch | Decode | Execute | Memory | WB

Gather data from the instruction
Read opcode; determine instruction type, field lengths
Read in data from register file
(0, 1, or 2 reads for `jump`, `addi`, or `add`, respectively)

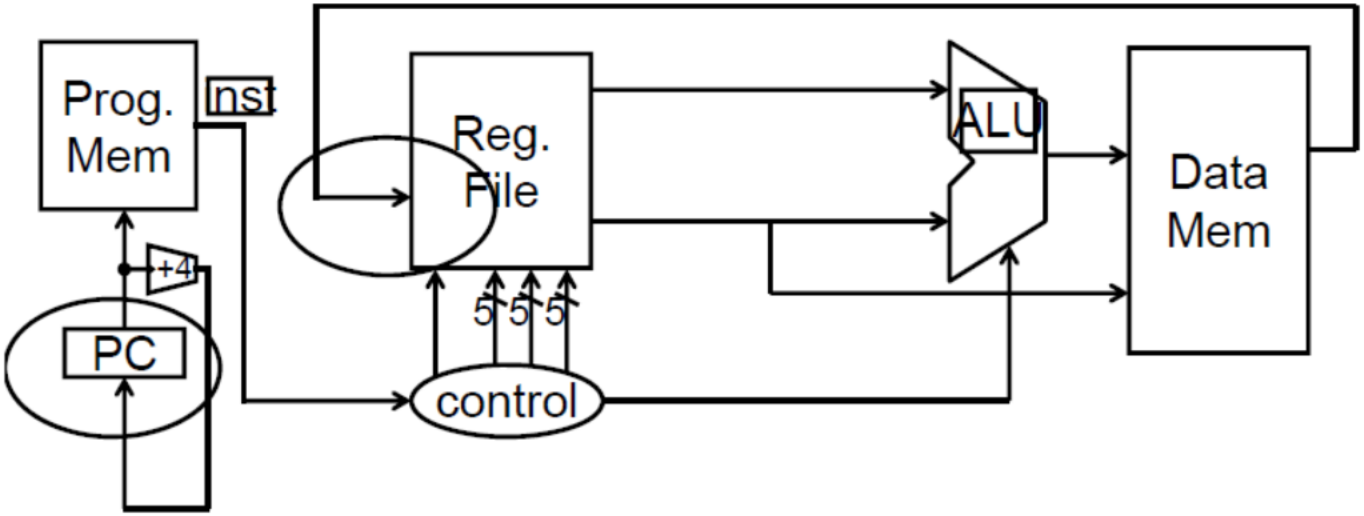


Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)

Load/Store? lw x2, x3, 32 → Compute address



Used by load and store instructions only
Other instructions will skip this stage



Fetch | Decode | Execute | Memory | WB

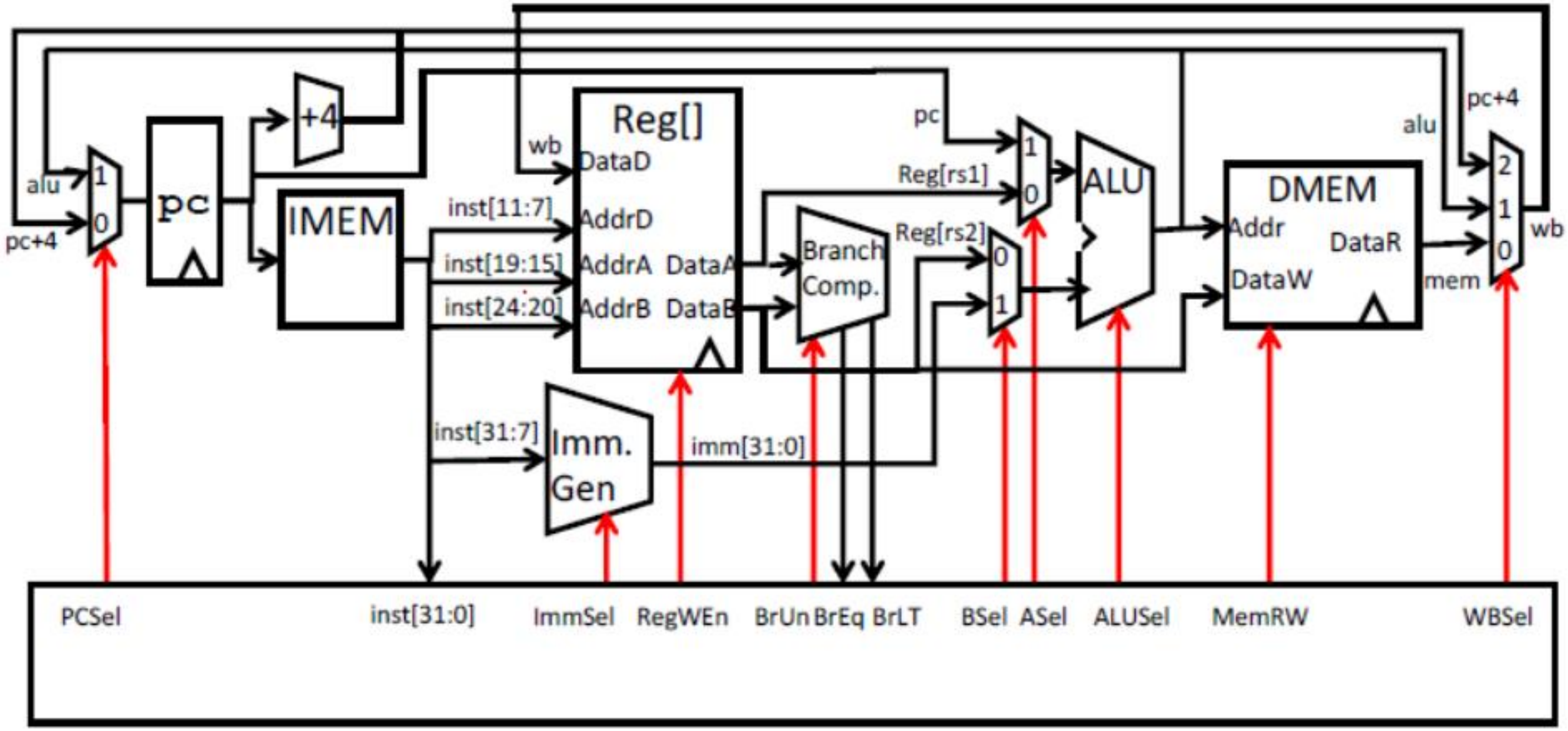
Write to register file

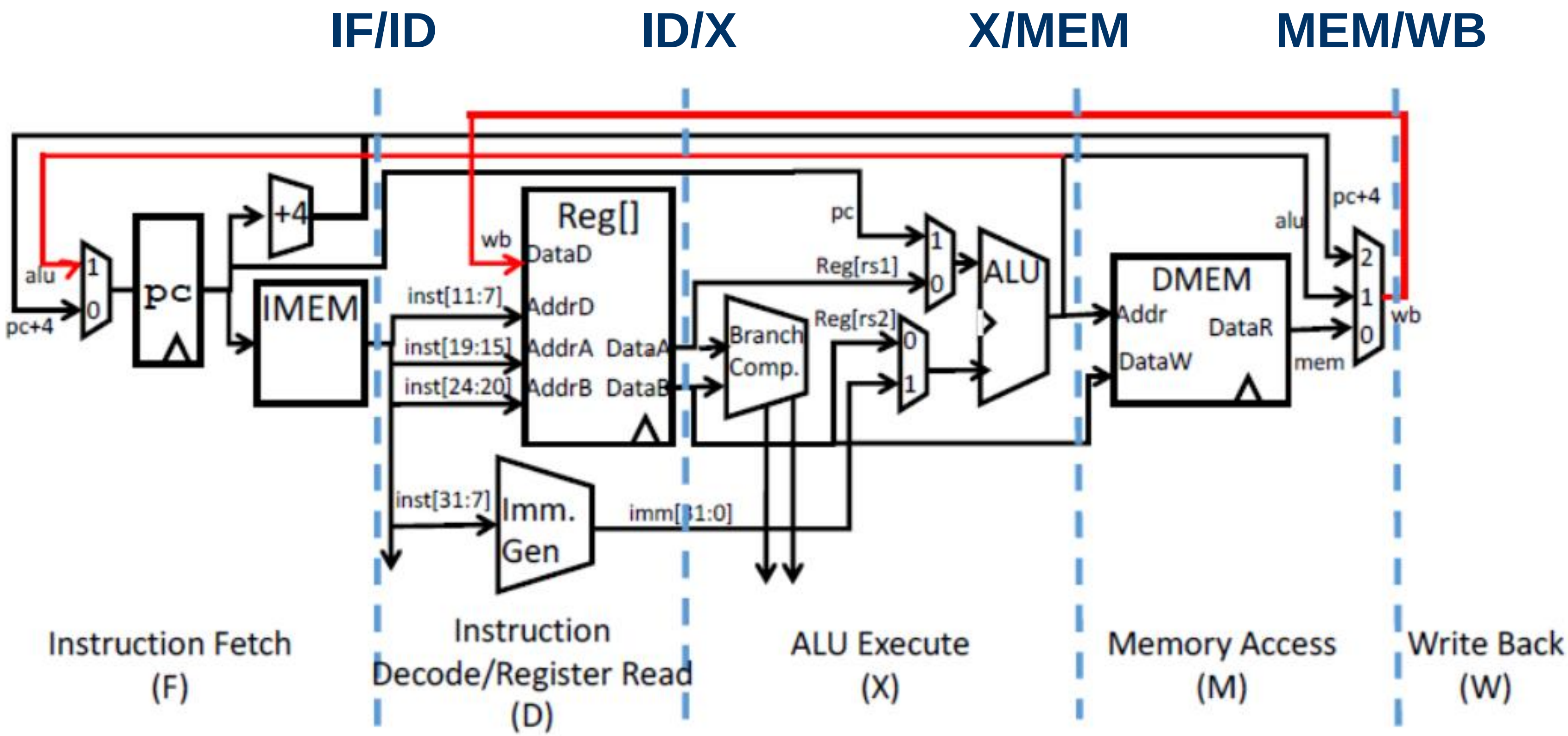
- For arithmetic ops, logic, shift, etc, load. What about stores?

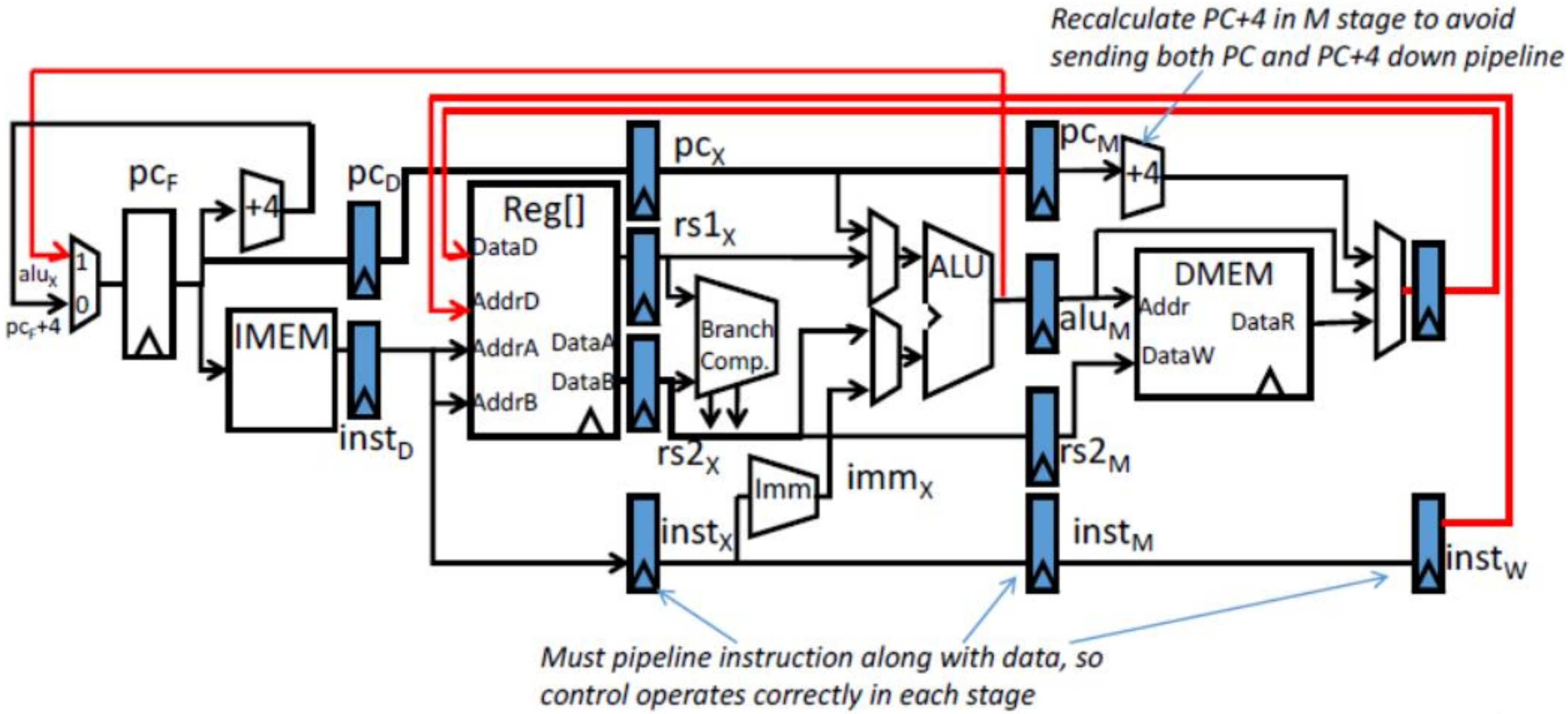
Update PC

- For branches, jumps

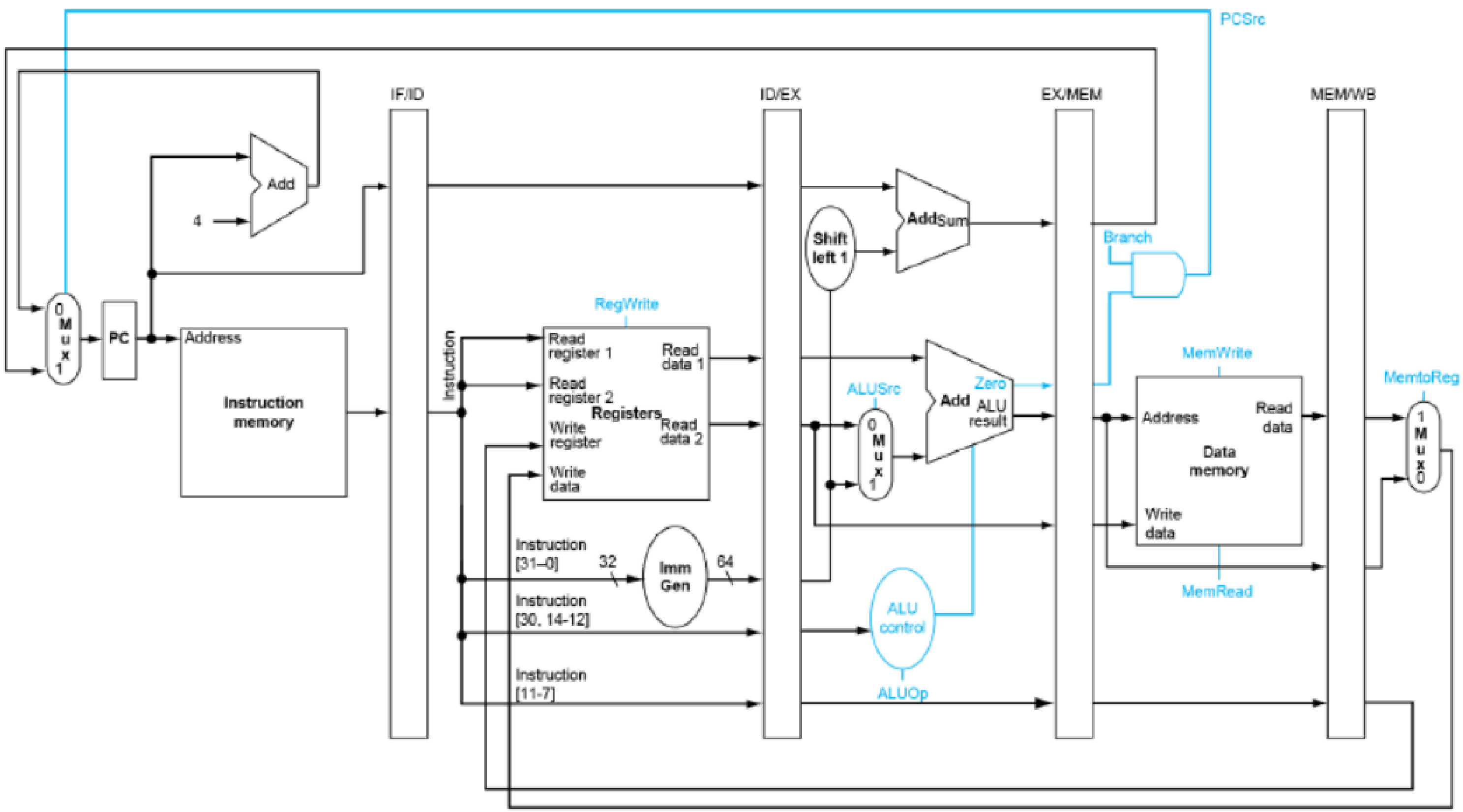
How about in RISC-V?



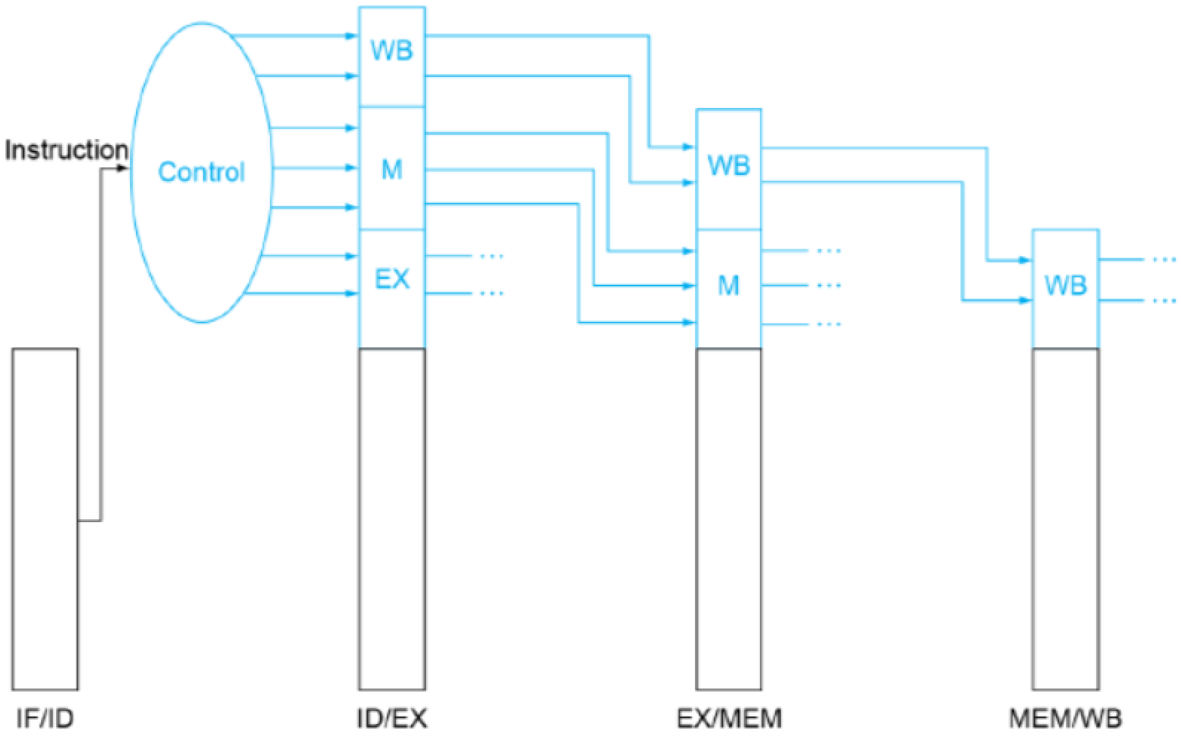

IF/ID
ID/X
X/MEM
MEM/WB
alu
pc+4
1
0
pc
+4
IMEM
wb
Reg[]
DataD
inst[11:7]
AddrD
inst[19:15]
AddrA
DataA
inst[24:20]
AddrB
DataB
inst[31:7]
Imm. Gen
imm[31:0]
Branch Comp.
pc
Reg[rs1]
Reg[rs2]
ALU
DMEM
Addr
DataR
DataW
alu
pc+4
mem
2
wb
Instruction Fetch (F)
Instruction Decode/Register Read (D)
ALU Execute (X)
Memory Access (M)
Write Back (W)


Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline
pc_F
+4
pc_D
alu_X
pc_F+4
1
0
IMEM
$inst_D$
Reg[]
DataD
AddrD
AddrA
AddrB
DataA
DataB
pc_X
$rs1_X$
$rs2_X$
$inst_X$
Branch Comp.
Imm
imm_X
ALU
pc_M
+4
alu_M
$rs2_M$
$inst_M$
DMEM
Addr
DataR
DataW
$inst_W$
Must pipeline instruction along with data, so control operates correctly in each stage

Pipelined Control



Pipelined Control



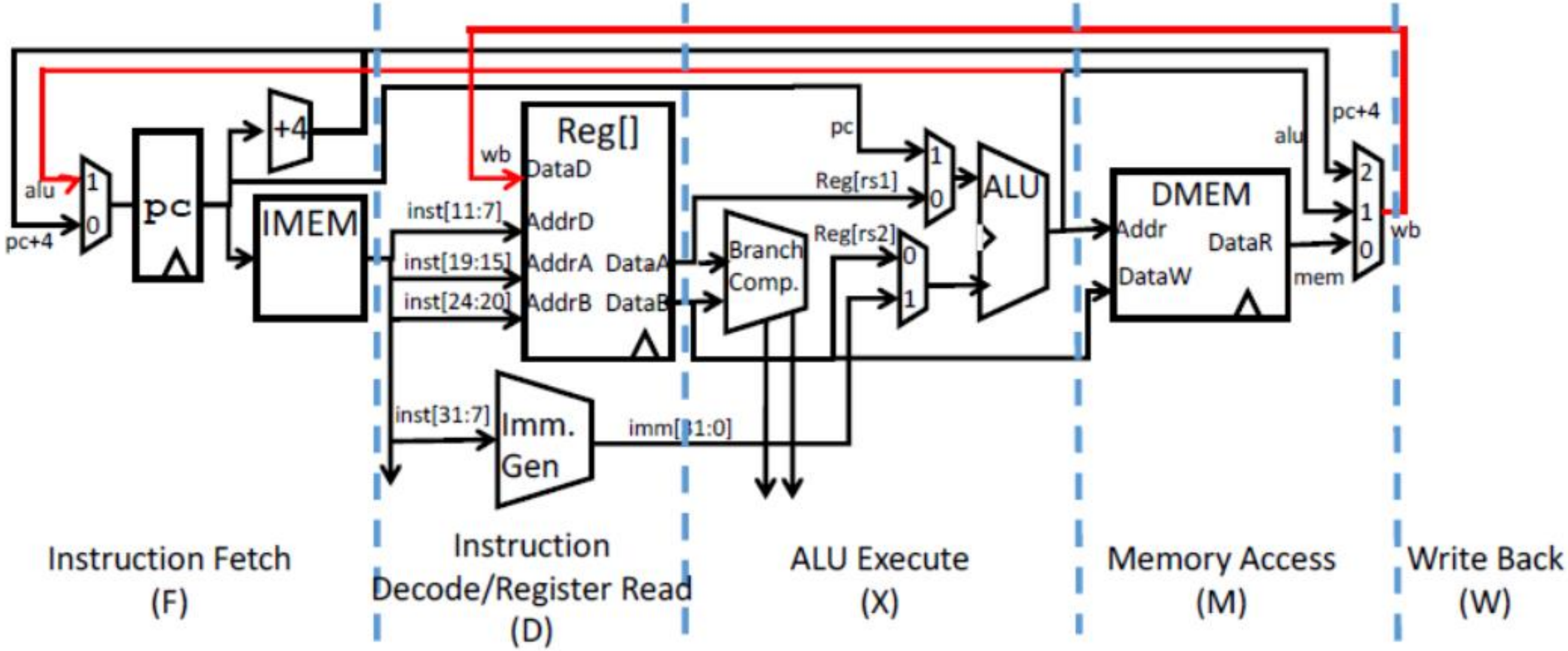
Pipelining in Processors

- Pipelining allows us to execute parts of multiple instructions at the same time using the same hardware!
  - – This is known as *instruction level parallelism (ILP)*

Assume time for stages is;

- *100ps for register read or write*
- *200ps for other stages*

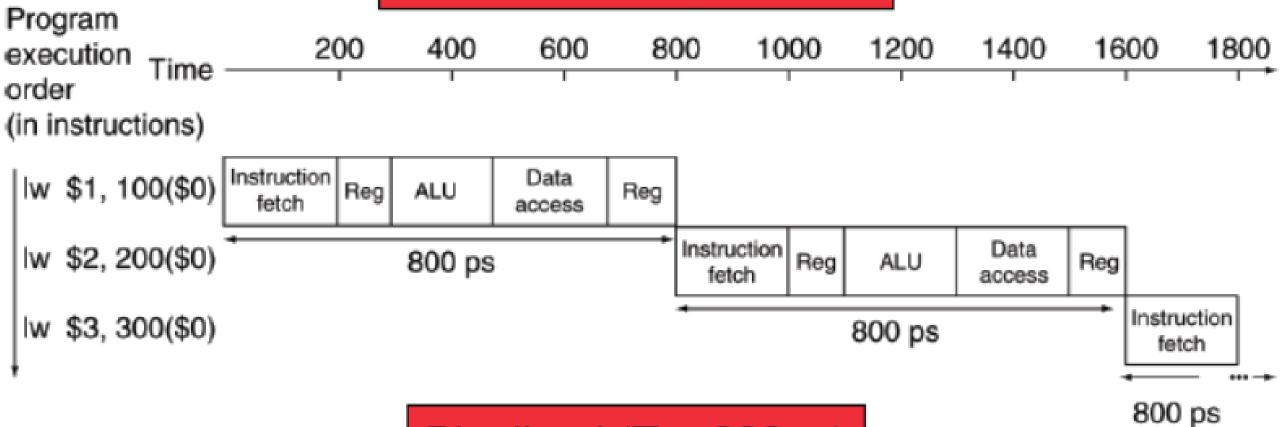
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps



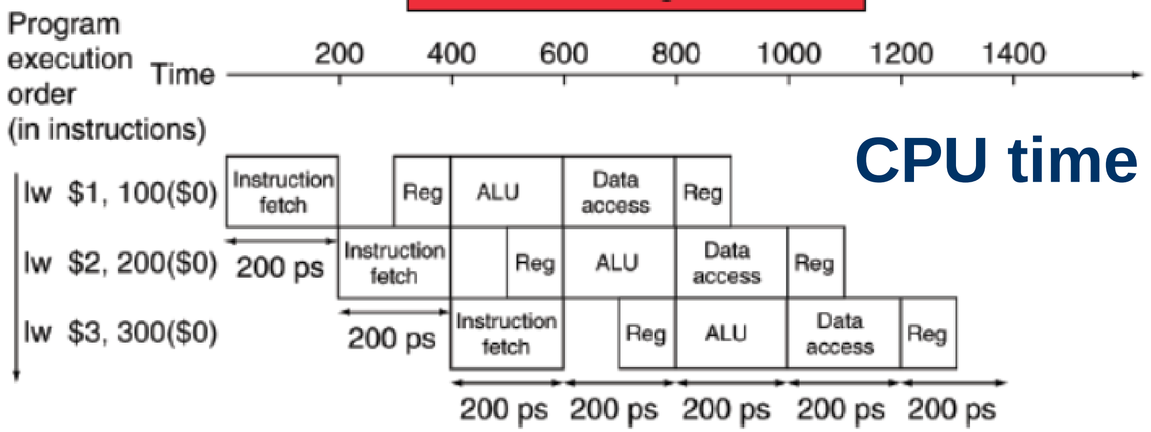
Operation times?

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Single-cycle (T_c= 800ps)

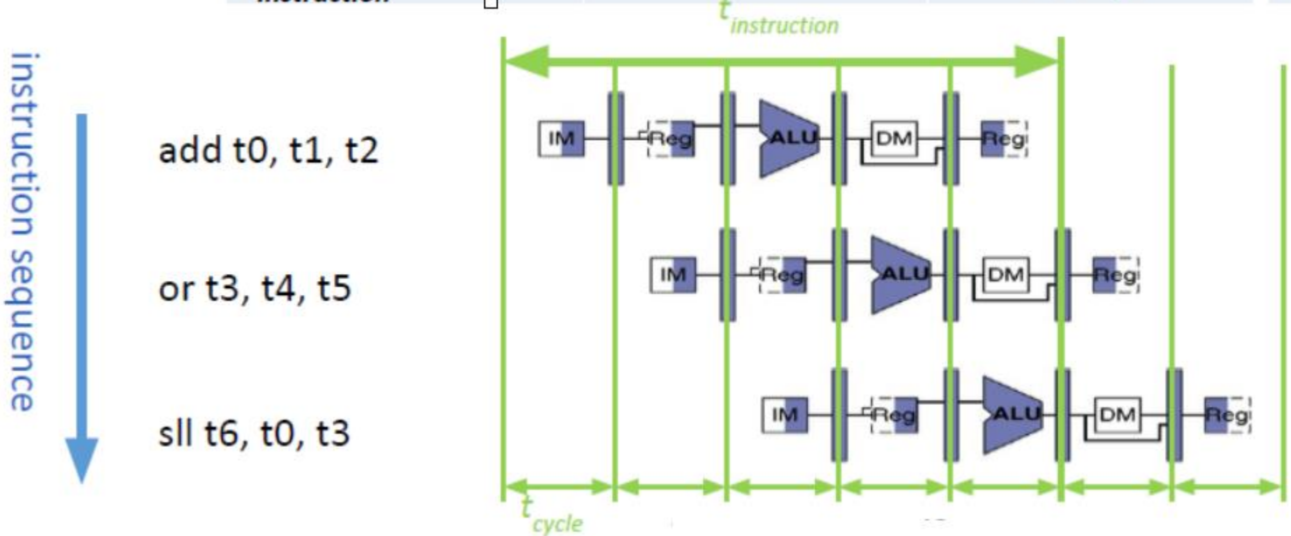


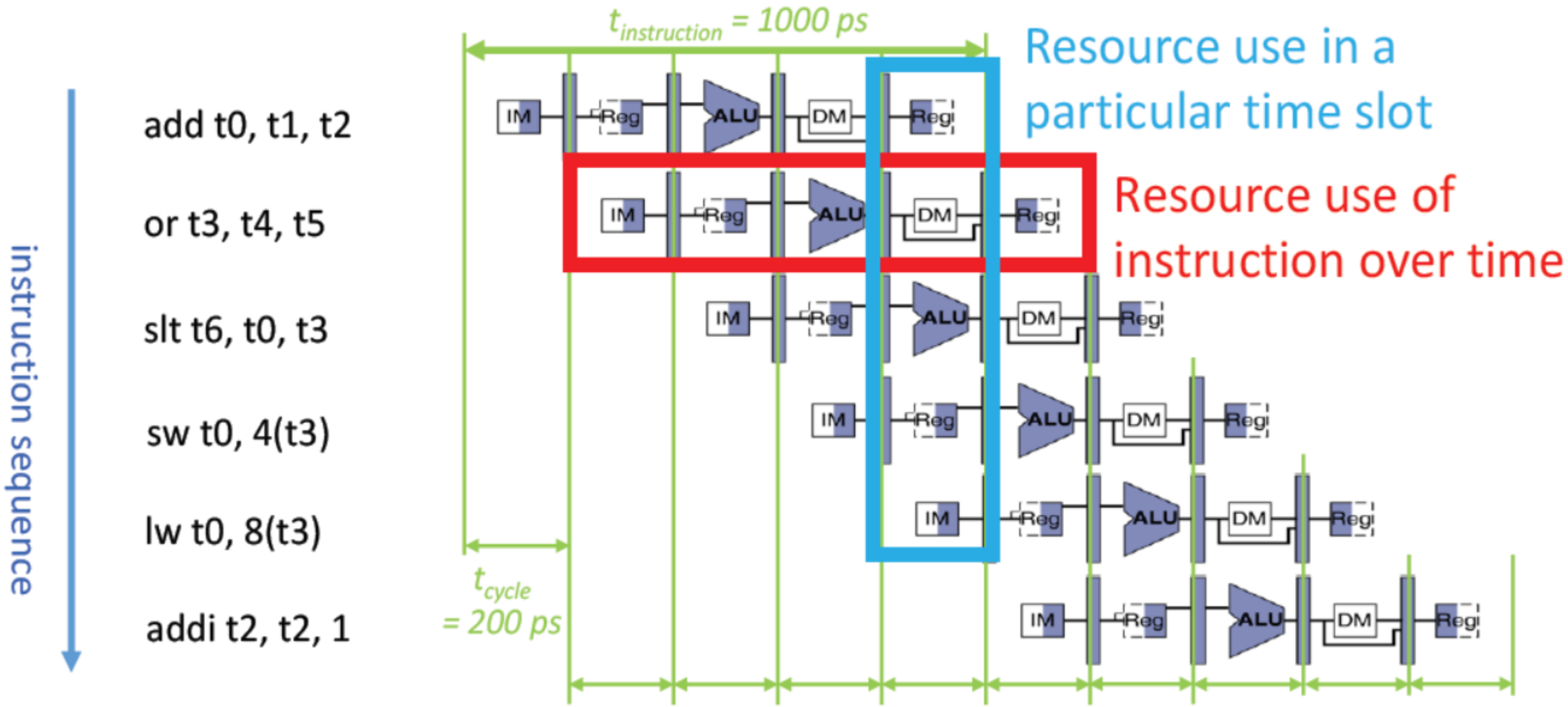
Pipelined (T_c= 200ps)



CPU time = CPI x CC x IC

Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch	IM	200 ps	200 ps
Reg Read	Reg	100 ps	200 ps
ALU	ALU	200 ps	200 ps
Memory	DM	200 ps	200 ps
Register Write	Reg	100 ps	200 ps
$t_{instruction}$	IM Reg ALU DM Reg	**800 ps**	**1000 ps**




$t_{instruction}$ = 1000 ps
Resource use in a particular time slot
Resource use of instruction over time
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
t_{cycle} = 200 ps
IM
Reg
ALU
DM
Reg

Pipelineing in Processors: Performance

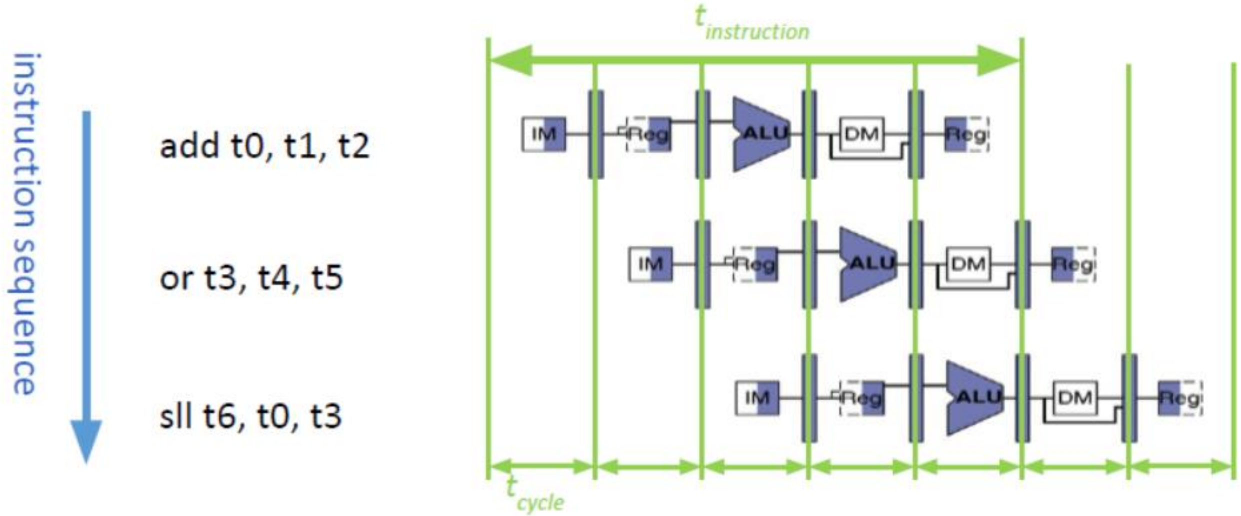
- Pipeline speedup

  If all stages are balanced (i.e., all take the same time)

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

  If not balanced, speedup is less

- Speedup due to <u>increased throughput</u>
- *Time for each instruction <u>does not decrease</u>* <u>(CPI = 1)</u>



	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
Relative speed	1 x	4 x

Consider 1,000,000 instructions

Pipelining in Processors: RISC-V

- 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

Hazards

Situations that prevent starting the next instruction in the next cycle

- Structure hazards

  A required resource is busy

- Data hazard

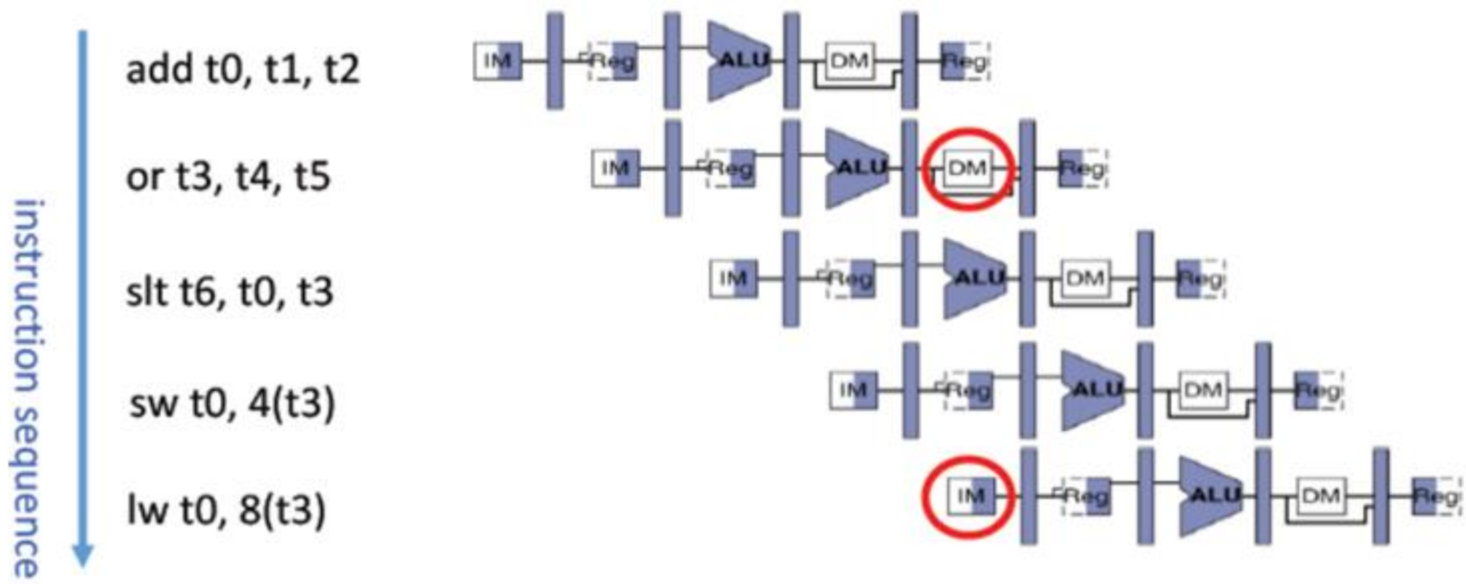
  Need to wait for previous instruction to complete its data read/write

- Control hazard

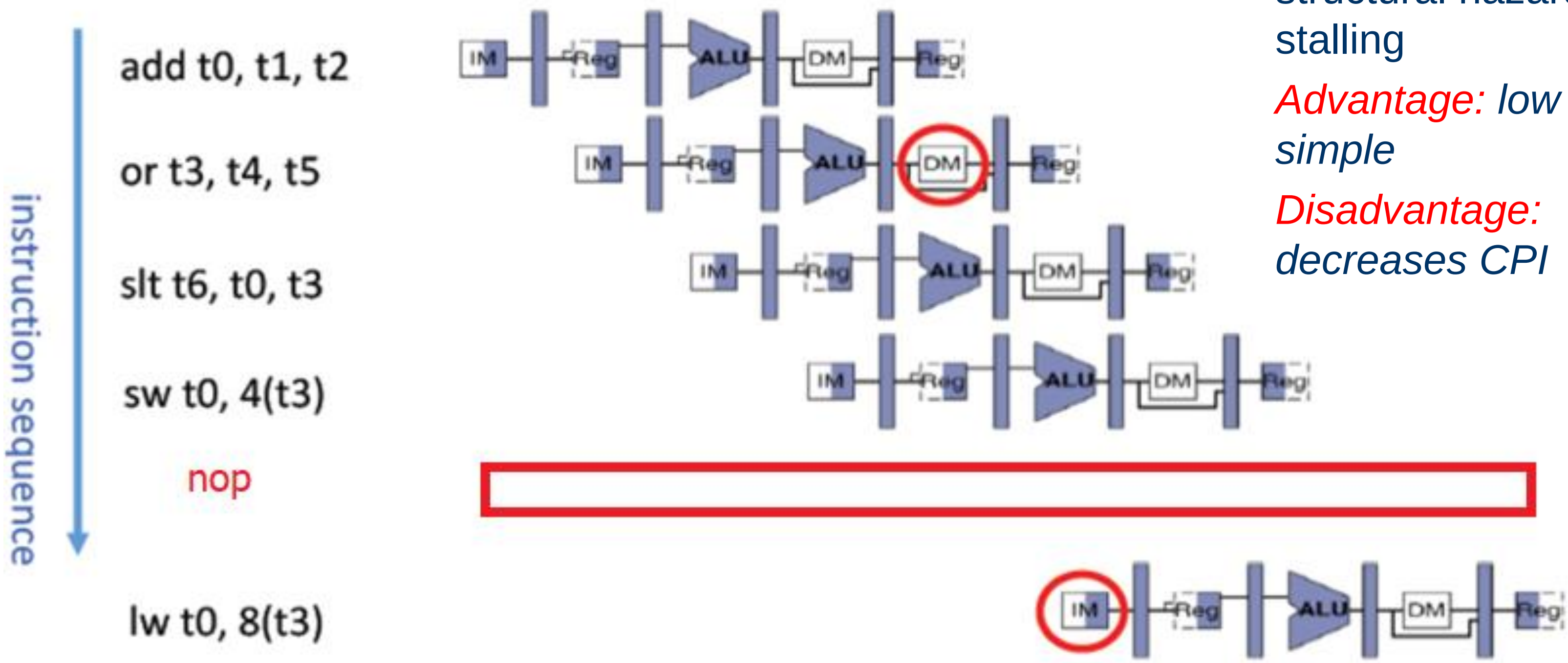
  Deciding on control action depends on previous instruction

Hazards: Structural Hazards

- Two different instructions need same h/w resource in same cycle (e.g., loads/stores use the same cache port as fetch



Hazards: Structural Hazards



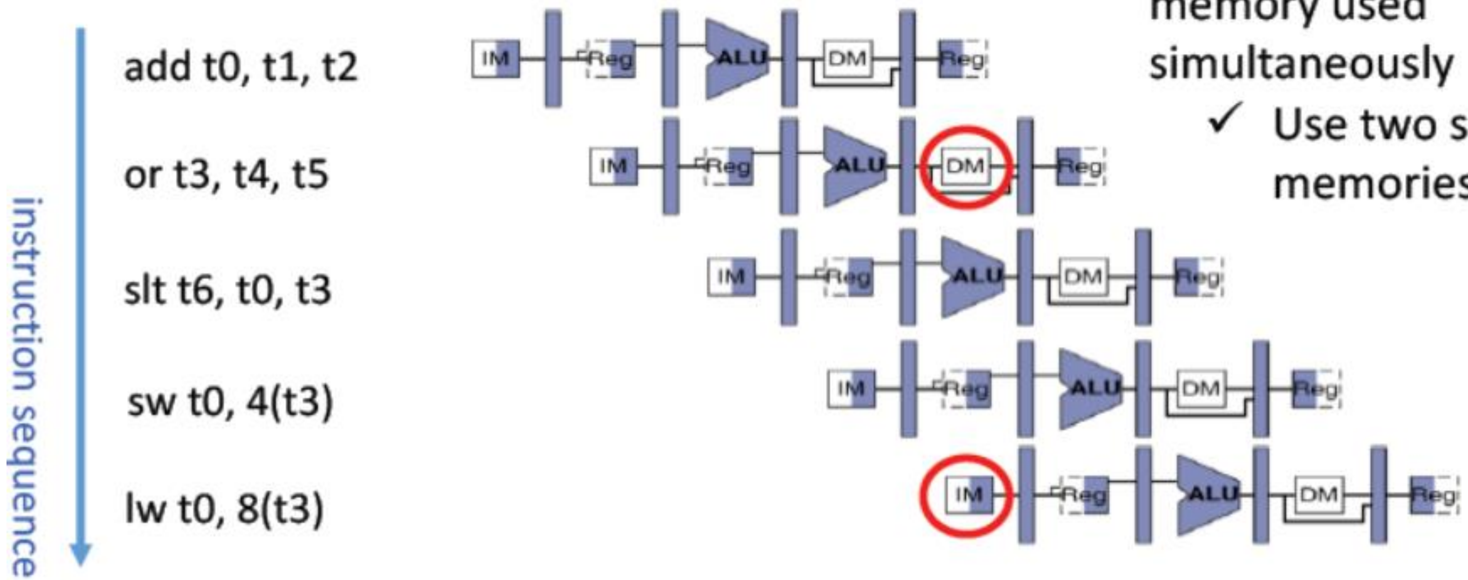
- **Option#1:** Fix structural hazard by stalling

  Advantage: low cost, simple

  Disadvantage: decreases CPI

Hazards: Structural Hazards

- Better solution…



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

Hazards: Structural Hazards

- Conflict for use of a resource

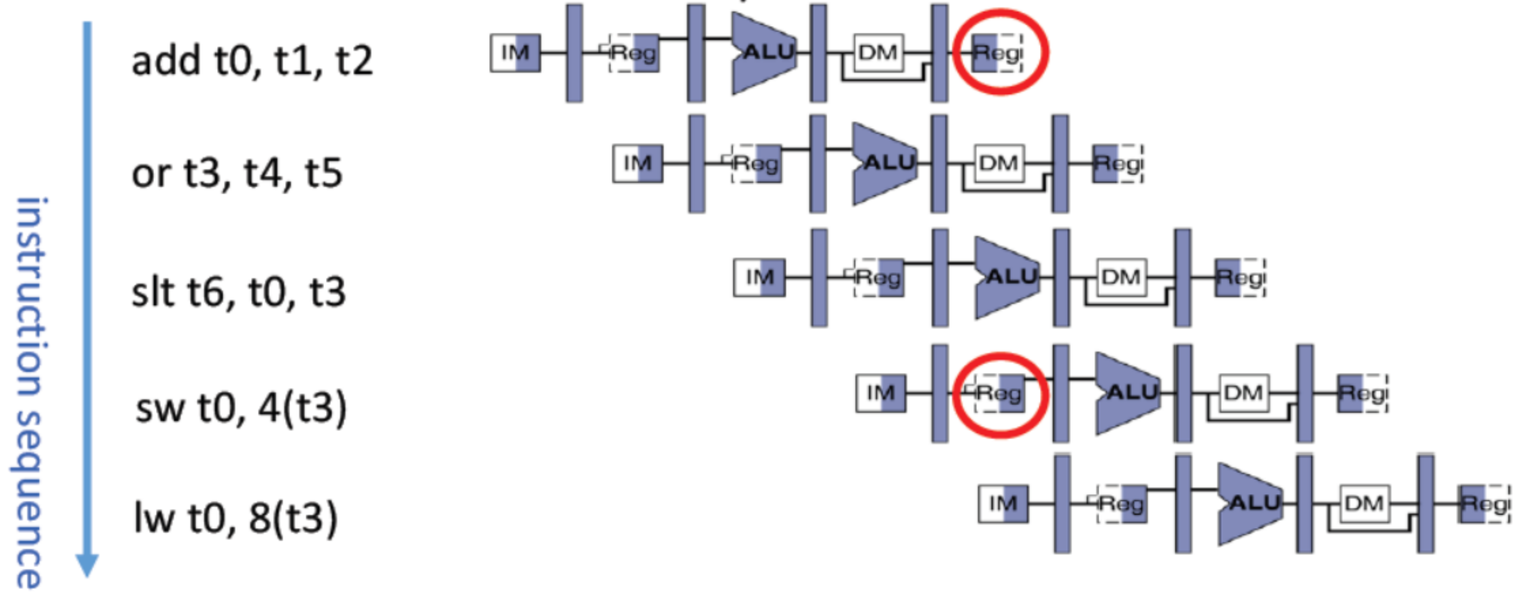
  In MIPS-style pipeline with a single memory *Load/store requires data access.*

  Instruction fetch would have to stall for that cycle. Would cause a pipeline "bubble"
- Hence, pipelined datapaths require <u>separate instruction/data memories</u> (and/or caches)

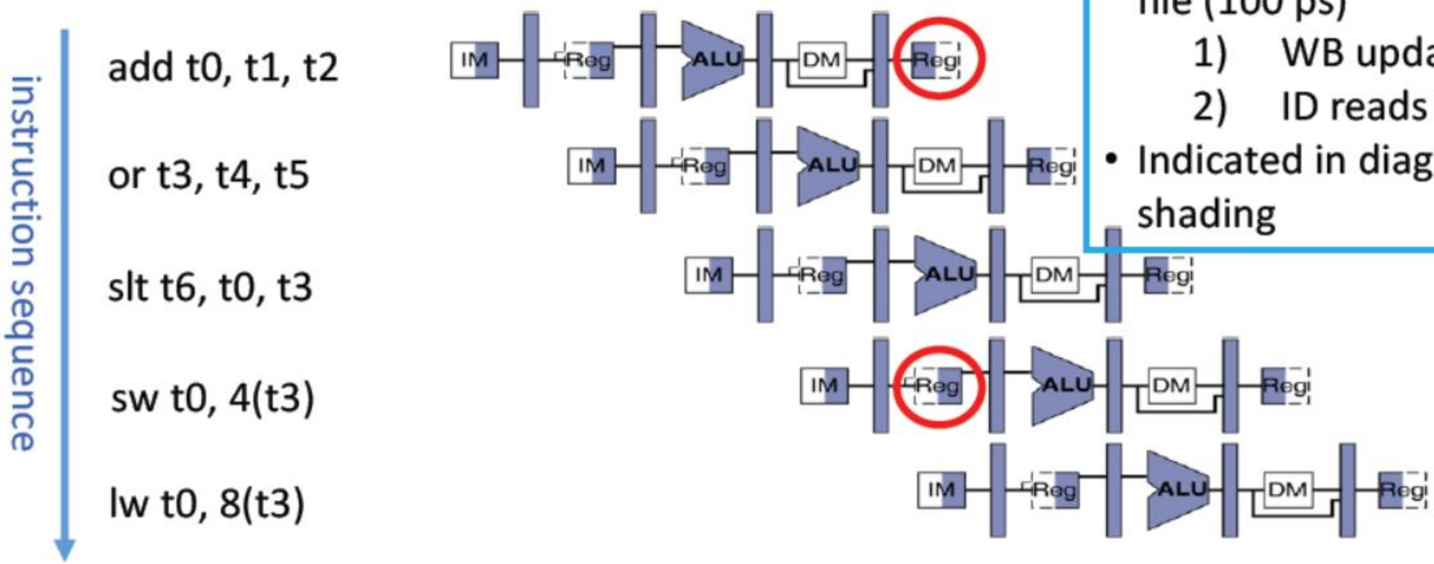
ITU VLSI LABS

Data Hazards: Register Access

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



Data Hazards: Register Access



- Exploit high speed of register file (100 ps)
  1) WB updates value
  2) ID reads new value
- Indicated in diagram by shading

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

Hazards: Structural Hazards

- Avoiding structural hazards

Option #1: Replicate the contended resource

Advantage: good performance

Disadvantage: increased area, slower (interconnect delay)

Use for cheap, divisible, or highly-contended resources

Option #2: pipeline the contended resource

Advantage: good performance, low area

Disadvantage: sometimes complex (e.g., RAM)

Hazards: Structural Hazards

Option #3: Design ISA/pipeline to reduce structural hazards

1: Instructions use a given resource at most once

2: Instructions use a given resource in same pipeline stage

That's the solution in RISC

Structural Hazards: Summary & In RISC

- 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

Hazards: R-Type Data Hazards

- Read-after-write (RAW): Two different instructions use the same storage location
- Problem: **sub reads R1 before add has written it**

add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3

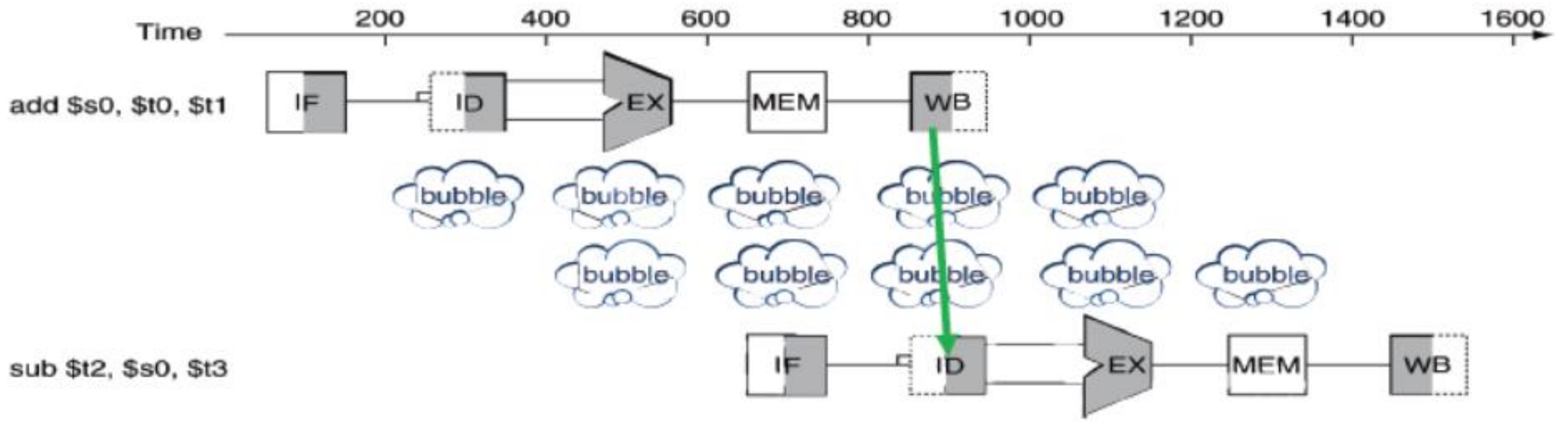
Read-After-Write (RAW)

“add” result (R1) will be written back to the registers in fifth cycle,

But “sub” will require R1 before that!

Hazards: R-Type Data Hazards

- One solution is “stalling”



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

Data Hazards: R-Type Data Hazards

- 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      <- Stall!
sw   t3, 12(t0)
lw   t4, 8(t0)
add  t5, t1, t4      <- Stall!
sw   t5, 16(t0)
```

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

Instruction Reordering

Hazards: R-Type Data Hazards

DEALING WITH RAW STALLS

- Why wait until WB stage?

  Data available at end of EX/MEM stage

  Bypass (aka "forward") data directly to input of EX

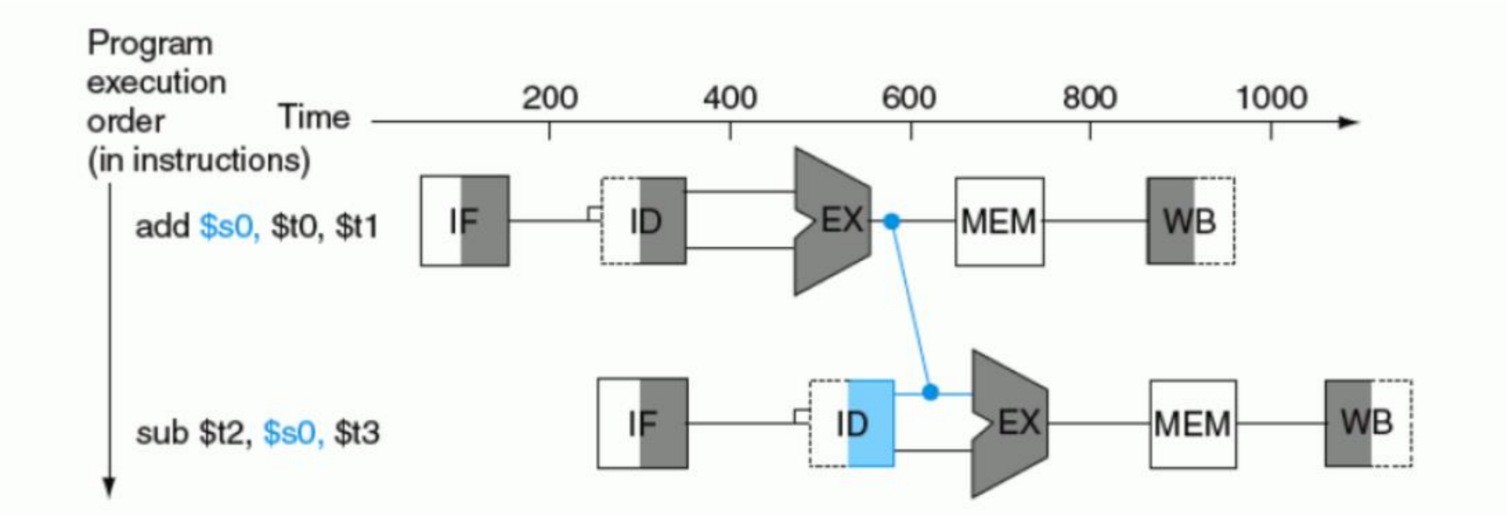
  Advantage: very effective at reducing/avoiding stalls

  in practice, a large fraction of input operands are bypassed

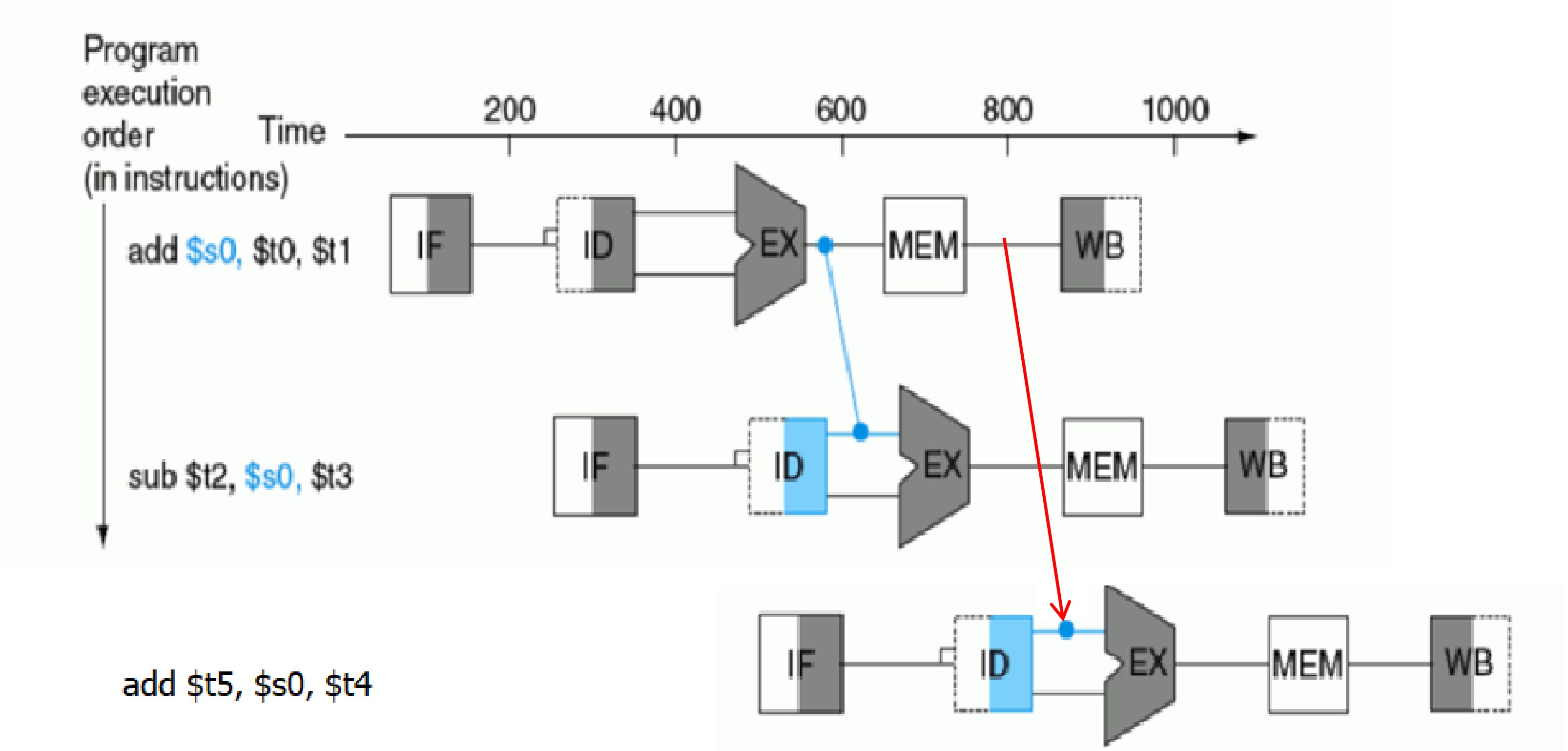
  Disadvantage: Adds complexity (extra decision logic)

Data Hazards: Bypassing

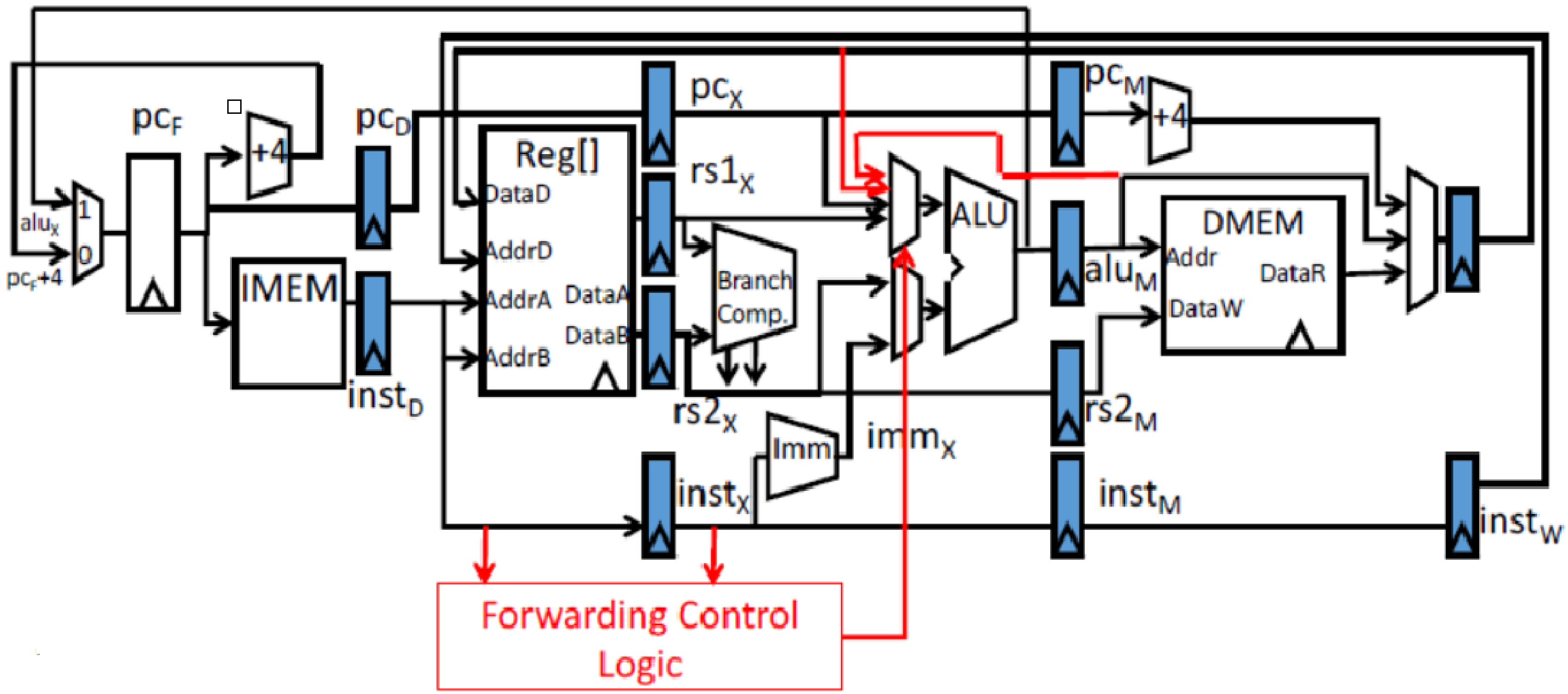
- Take values within pipeline stages, rather than taking from registers
- Requires extra connections in datapath
- Also called as "forwarding"



Data Hazards: Bypassing

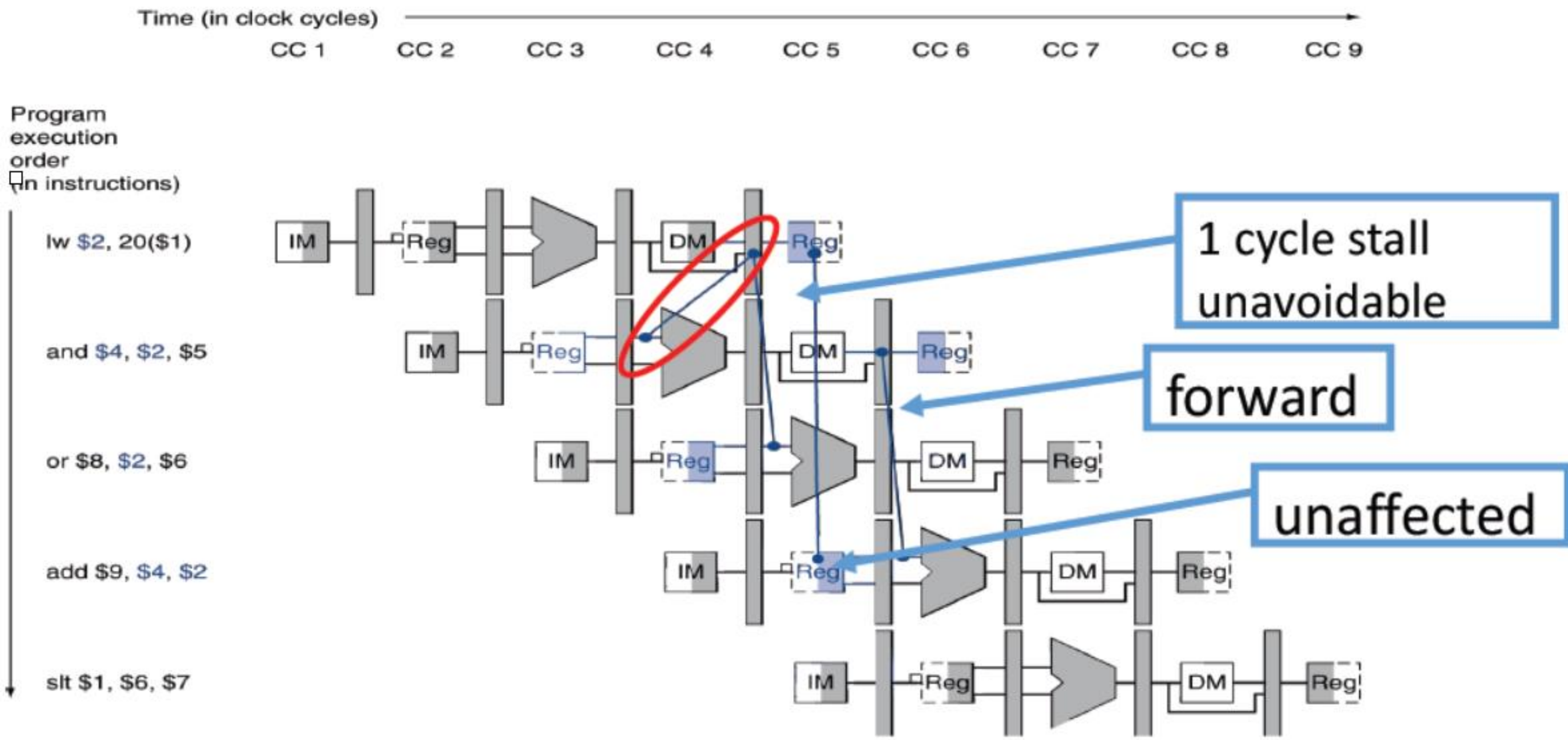


Data Hazards: Bypassing



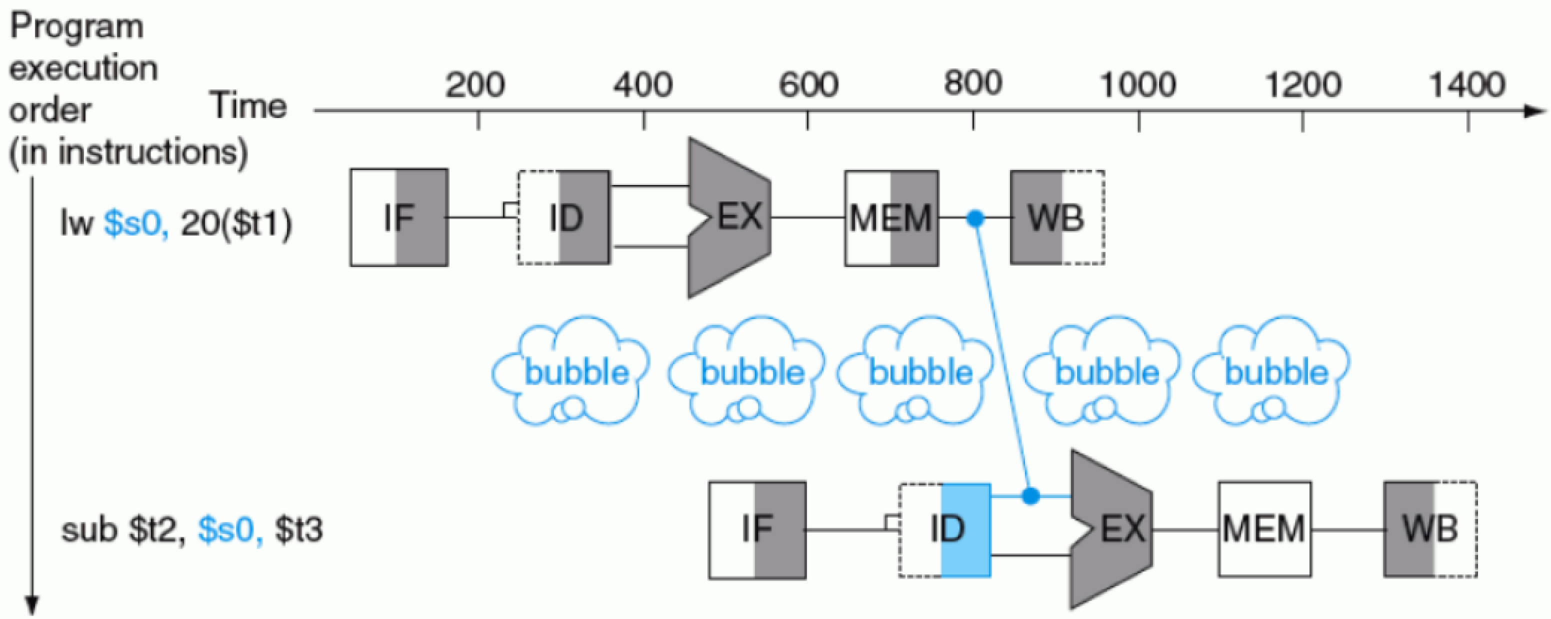
ITU VLSI LABS

Data Hazards: Load-Use Hazards



Data Hazards: Load-Use Hazards

Stalling + Bypassing is a solution



Hazards: Control Hazards

When an instruction affects *which instruction executes next*

Naive solution: stall until outcome is available (end of EX)

Advantage: simple

Disadvantage: low performance (2 cycles here, longer in general), e.g. 15% branches * 2 cycle stall ⇒ 30% CPI increase!

store R4,0(R5)
bne R2,R3,loop
sub R1,R6,R3

	1	2	3	4	5	6	7	8	9
store R4,0(R5)	F	D	X	M	W				
bne R2,R3,loop		F	D	**X**	M	W			
??			**c***	**c***	**F**	D	X	M	W

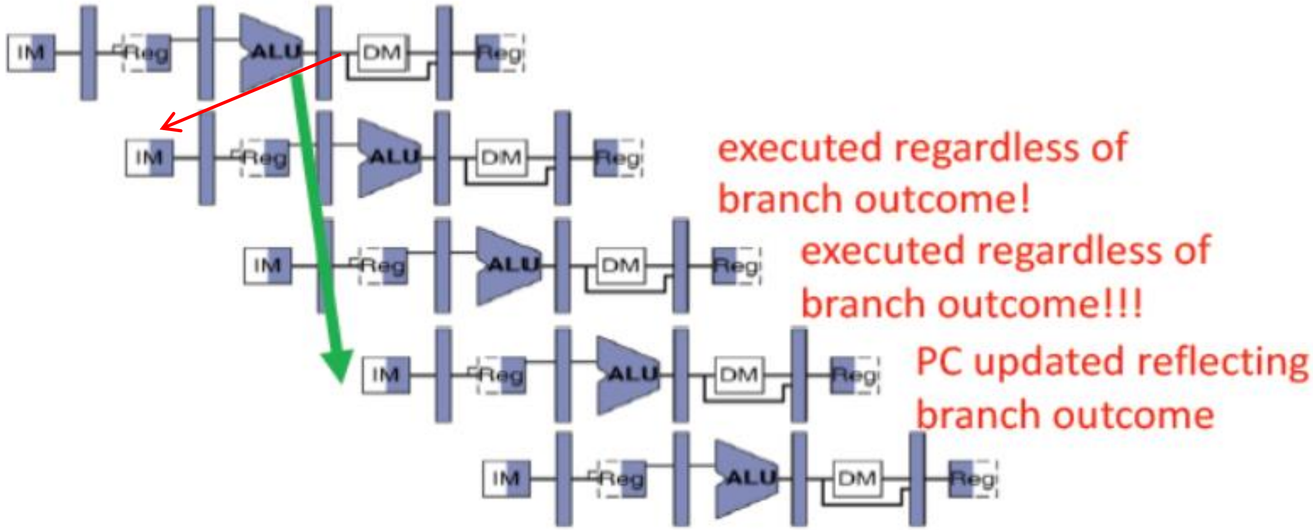
beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

xor t5, t1, s0

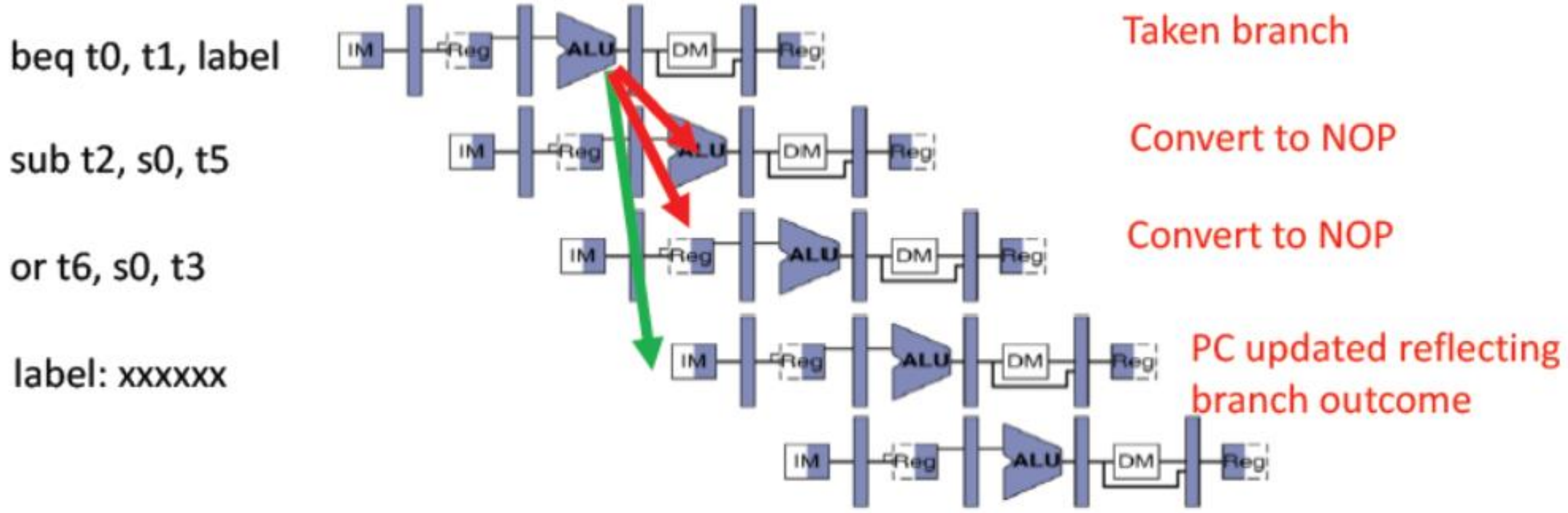
sw s0, 8(t3)



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

Hazards: Control Hazards

- Kill insturctions after branch is taken



Hazards: Control Hazards

- Branch prediction?
  - Static branch prediction
    - Based on typical branch behavior
    - Example: loop and if-statement branches
      - Predict backward branches taken
      - Predict forward branches not taken
  - Dynamic branch prediction
    - Hardware measures actual branch behavior
      - e.g., record recent history of each branch
    - Assume future behavior will continue the trend
      - When wrong, stall while re-fetching, and update history

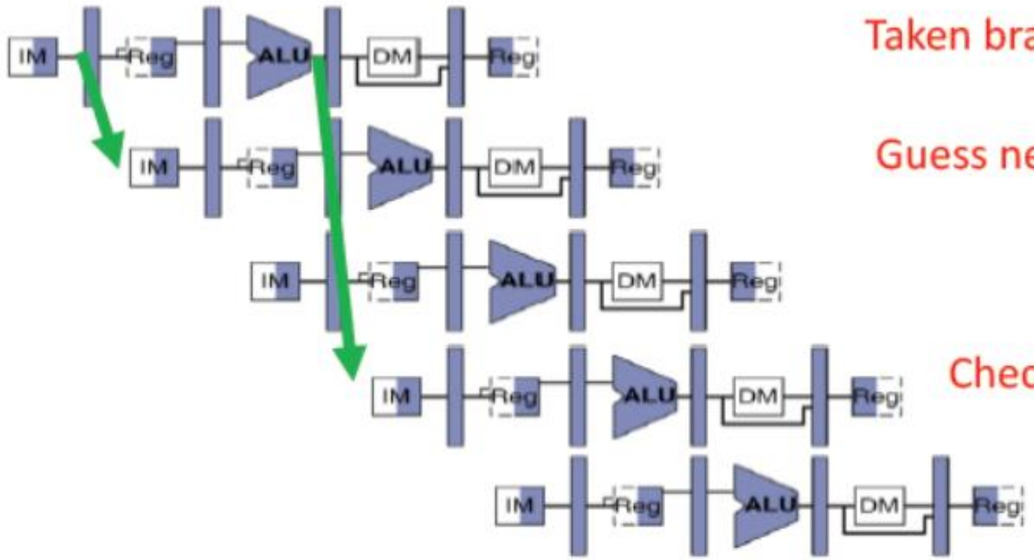
ITU VLSI LABS

- 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

beq t0, t1, label

label: .....

.....



Taken branch

Guess next PC!

Check guess correct

Hazards: Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch

- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Hazards: Control Hazards

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay