Universität Bremen

RISC-V Pipelining

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck

- Problem
- Algorithm
- Program
- Instruction Set Architecture
- **Microarchitecture**
- Logic
- Digital Circuits
- Analog Circuits
- Devices
- Physics

Announcements

- There is a tutorial tomorrow


Turn this car into a convertible.

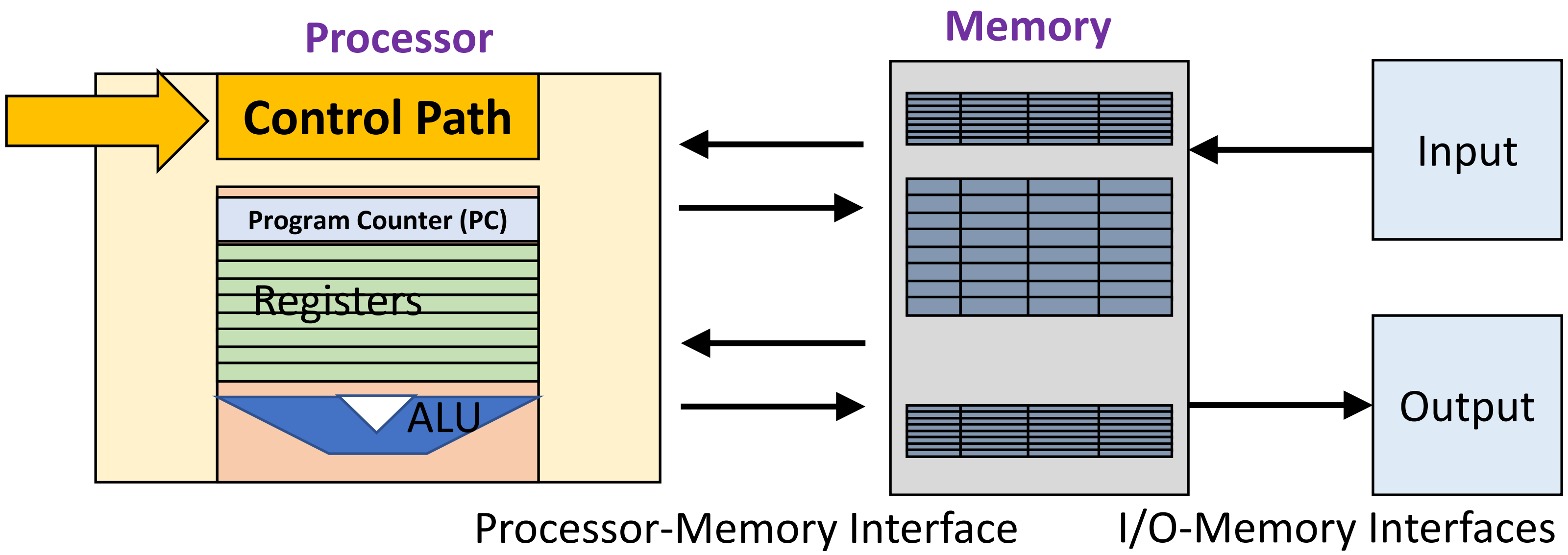
Today’s Agenda

- Implementing Controller
  - Logic-based implementation
  - ROM-based implementation
- Drawbacks of a Single-Cycle Processor
- An Overview of Pipelining
- Pipelined Datapath
  - Fetch, Decode, Execution, Memory, Write-Back
- Pipelined Control
- Pipeline Hazards
  - Structural Hazards
  - Data Hazards
  - Control Hazards

Controller

- We have designed a complete datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - We still need to design it

Assembly Variables – Registers



Control Logic Truth Table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUN	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
r-r op	*	*	+4	*	*	Reg	Reg	(Op)	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4

Control Realization Options

- ROM
  - “Read-Only Memory”
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions
  - Popular when designing control logic manually
- Combinatorial Logic
  - Today, chip designers use logic synthesis tools to convert truth tables to networks of gates

RV32I, A Nine-bit ISA!

- Instruction type encoded using only 9 bits:
  - inst[6:2]
  - inst[14:12]
  - inst[30]

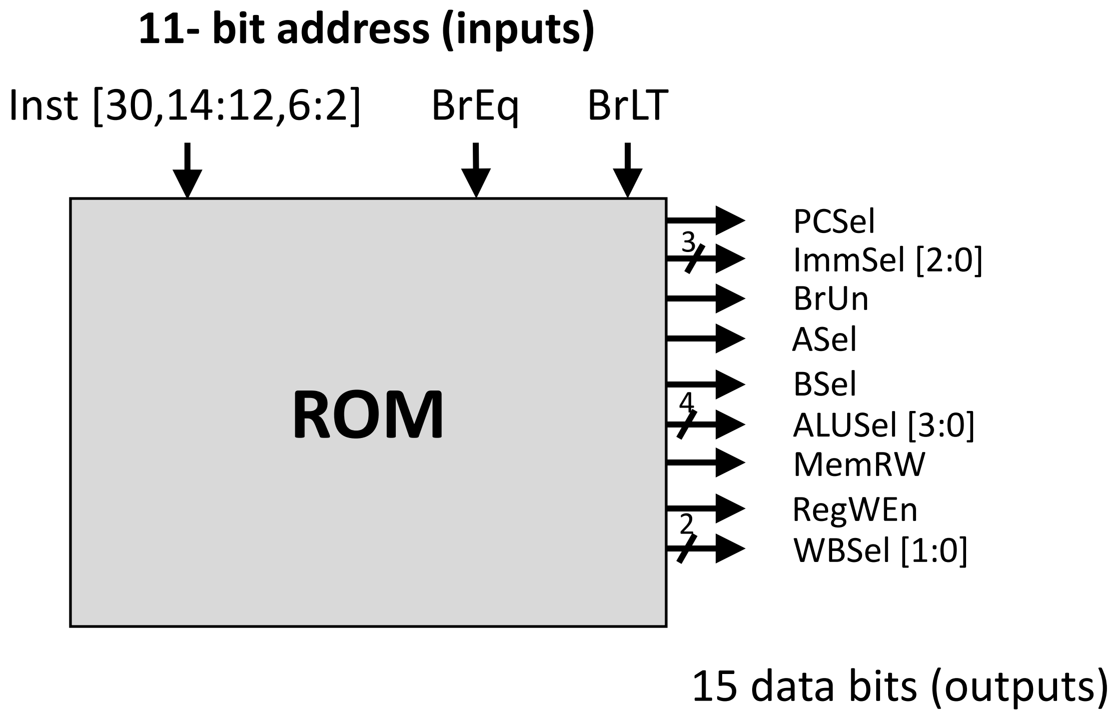
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

https://five-embeddev.com/riscv-isa-manual/latest/instr-table.html

ROM-based Control

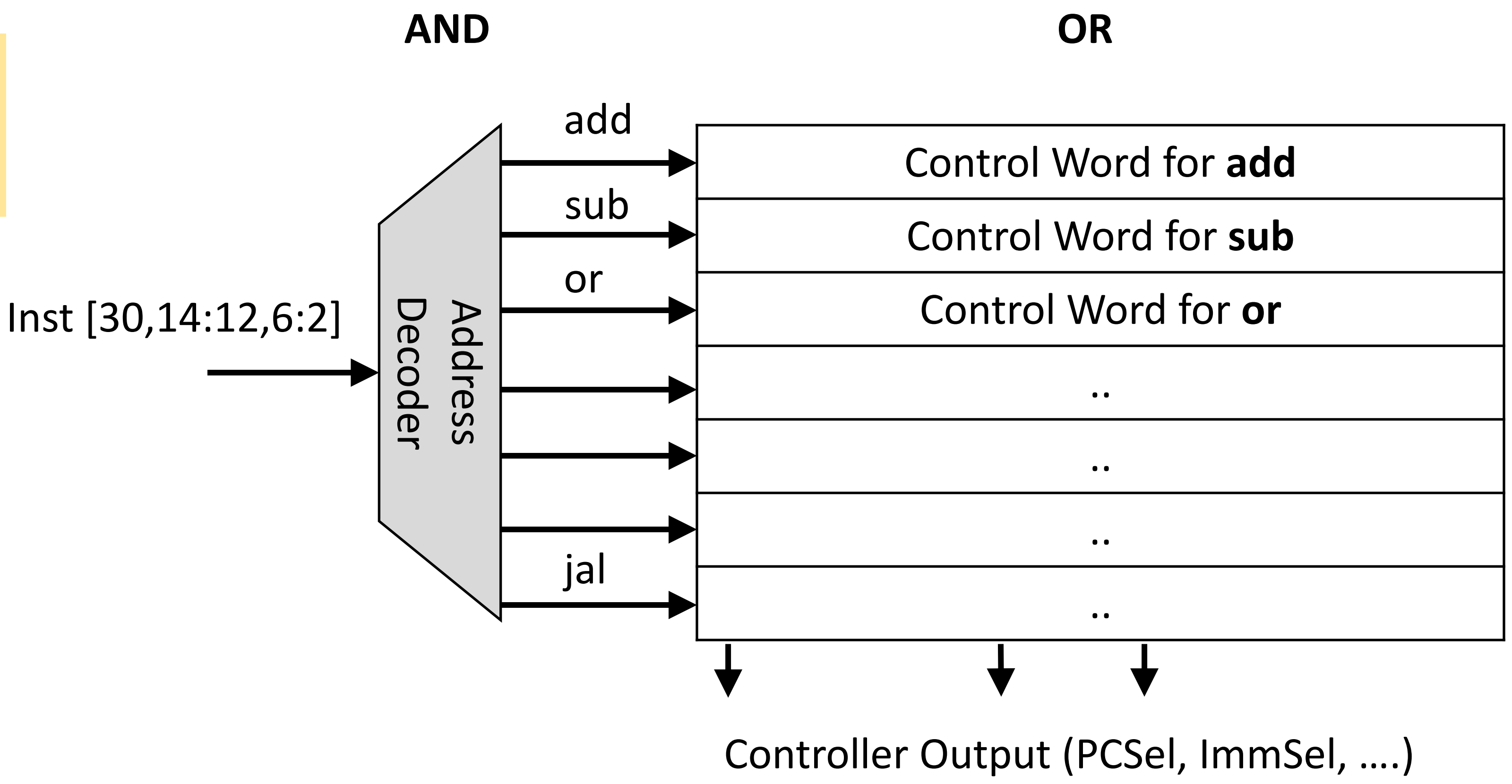
- Any logical transformation of N input bits to M output bits can be accomplished by a look-up table with 2^N entries (indexed by the input bits) of M bits each.

- look-up table can be simpler (less error-prone) and friendlier



ROM Controller Implementation

For each binary address input pattern, exactly one of the wordlines (horizontal) is activated by the address-decoder.



Control Logic to Decode add

inst[30] inst[14:12] inst[6:2]

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

add $= \overline{i[30]}\ \&\ \overline{i[14]}\ \&\ \overline{i[13]}\ \&\ \overline{i[12]}\ \&\ \text{R-type}$

R-type $= \overline{i[6]}\ \&\ i[5]\ \&\ i[4]\ \&\ \overline{i[3]}\ \&\ \overline{i[2]}\ \&\ \text{RV32I}$

RV32I $= i[1]\ \&\ i[0]$

Universität Bremen

RISC-V Pipelining

Instruction Timing

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

Maximum clock frequency $\mathbf{f}_{\mathbf{max}}$ = 1/800ps = 1.25 GHz

Performance Measures

- “Our” Single-cycle RISC-V CPU executes instructions at 1.25 GHz
  - 1 instruction every 800 ps
- Can we improve its performance?
  - What do we mean with this statement?
  - Not so obvious:
    - Quicker response time, so one job finishes faster?
    - More jobs per unit time (e.g. web server returning pages, spoken words recognized)?
    - Longer battery life?

Universität Bremen

Public Transport Analogy

	Sports Car	Bus
Passenger Capacity	2	50
Travel speed	200 KM/H	50 KM/H
50 KM trip (assume they return instantaneously)		
Travel time	15 mins	60 mins
Time for 100 passengers	750 mins (50 trips, 2 persons each)	120 mins (2 trips. 50 persons each)

Transportation	Computer
Trip time	Program execution time, e.g., display update
Time for 100 passengers	Throughput, e.g., number of server requests handled per hour

Processor Performance

CPI = Cycles Per Instruction

- $\frac{Time}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}$

Determined by
- Task
- Algorithm, e.g., O(N)
- Programming language
- Compiler
- ISA

Determined by
- ISA
- Micoarchitecture, e.g., single cycle RISC-V design, CPI = 1

Determined by
- Microarchitecture (critical path)
- Technology
- Power budget (low voltage reduces transistor speed)
- 1/Frequency

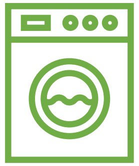
Speed tradeoff example
For some task (e.g., image processing) ….

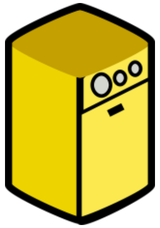
Processor B is faster for this task, despite executing more instructions and having a slower clock rate!

	Processor A	Processor B
# Instructions	1 Million	1.5 Million
Average CPI	2.5	1
Clock rate (Frequency)	2.5 GHz	2 GHz
Execution time	1 ms	0.75 ms

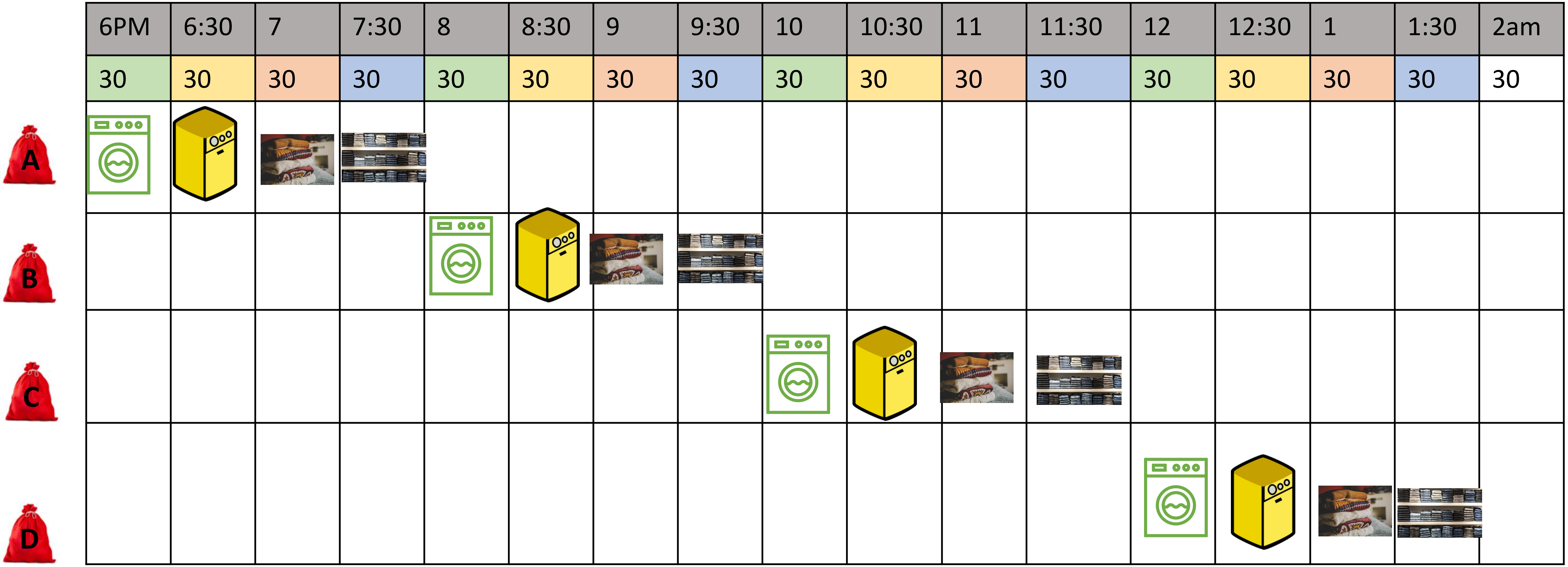
Laundry Day

- Adam, Benjamin, Caroline, Dan each have one load of clothes to wash, dry, fold, and put away
- Washer takes 30 minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- “Stasher” takes 30 minutes to put clothes into drawers





Sequential Laundry



Sequential laundry takes 8 hours for 4 loads!

Pipelined Laundry

6PM	6:30	7	7:30	8	8:30	9	9:30	10
30	30	30	30	30	30	30	30	30

- What happens **sequentially**?
- What happens **simultaneously**?

Pipelined laundry takes 3.5 hours for 4 loads!

Sequential Laundry

6PM	6:30	7	7:30	8	8:30	9	9:30	10
30	30	30	30	30	30	30	30	30

- Pipelining doesn’t help **latency** of single task, it helps **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number of pipe stages**
- Time to “**fill**” pipeline and time to “**drain**” it reduces speedup: 2.3X v. 4X in this example

Pipelined laundry takes 3.5 hours for 4 loads!

Sequential Laundry

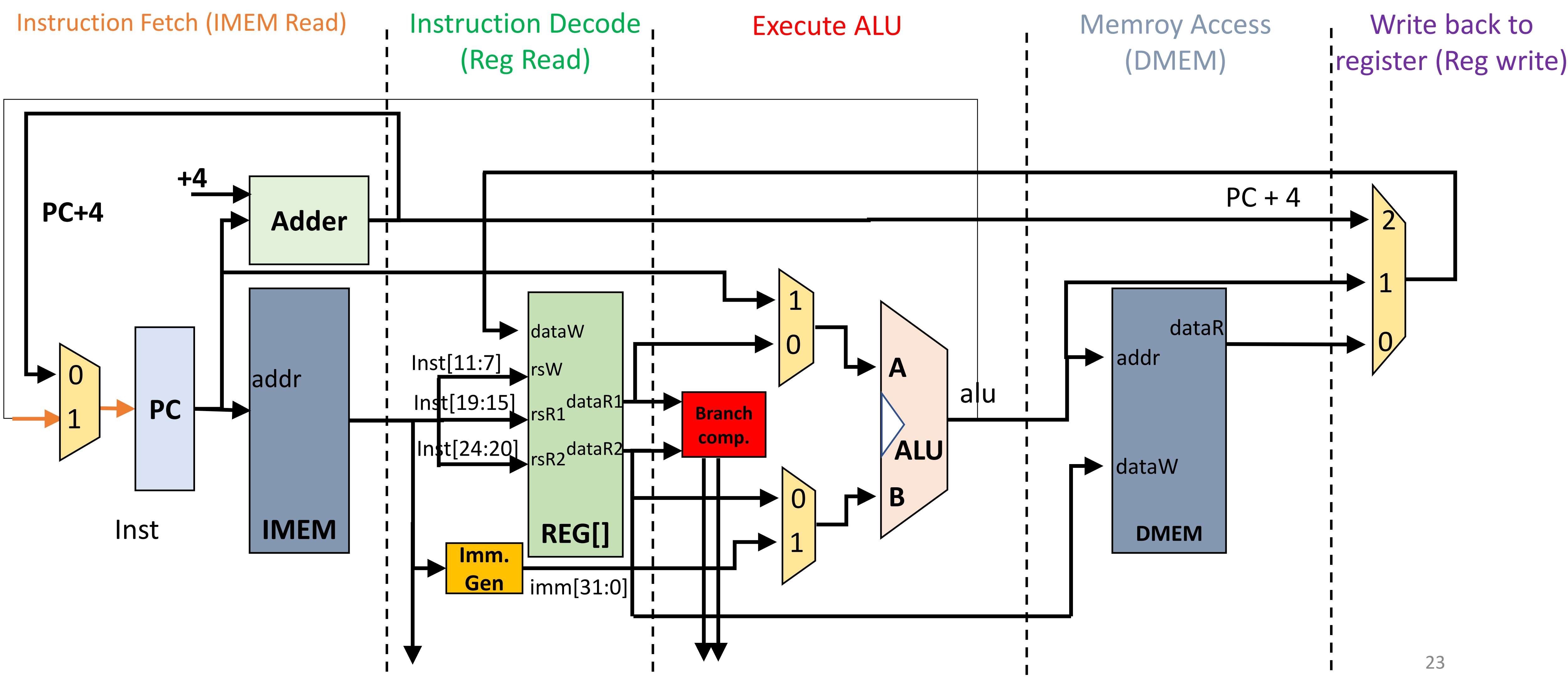
6PM	6:30	7	7:30	8	8:30	9	9:30	10
20	30	30	20	30	30	30	30	30

- Suppose:
  - new Washer takes 20 minutes
  - new Stasher takes 20 minutes.
- How much faster is pipeline?
- Pipeline rate limited by **slowest** pipeline stage
- Unbalanced lengths of pipe stages reduce speedup

Pipelined laundry takes 3.5 hours for 4 loads!

Universität Bremen

Single-cycle RV32I Datapath



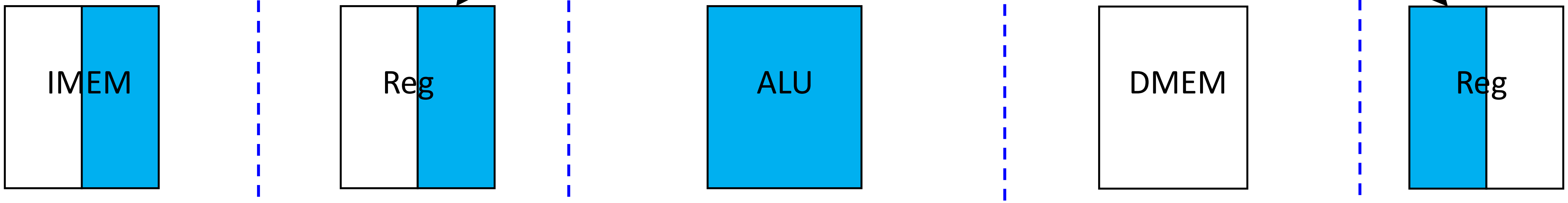
Universität Bremen

Symbolic Representation of 5 Stages

Instruction Fetch (IMEM Read)	Instruction Decode (Reg Read)	Execute ALU	Memroy Access (DMEM)	Write back to register (Reg write)

add x1, x2,x3

Instruction Decode, Write back stages respectively read and write the same hardware element, Reg (RegFile).

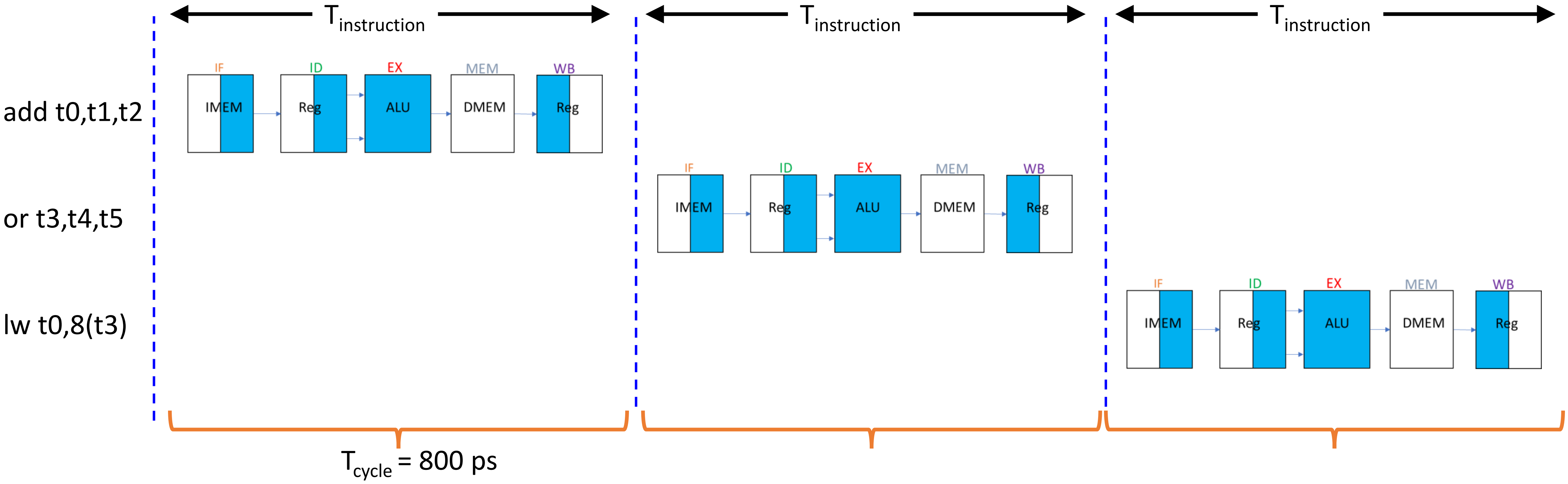


200 ps	**100 ps**	200 ps	200 ps	**100 ps**

- Shading indicates the usage of element
- Shading on left → element is written in that stage
- Shading on right → element is read in that stage

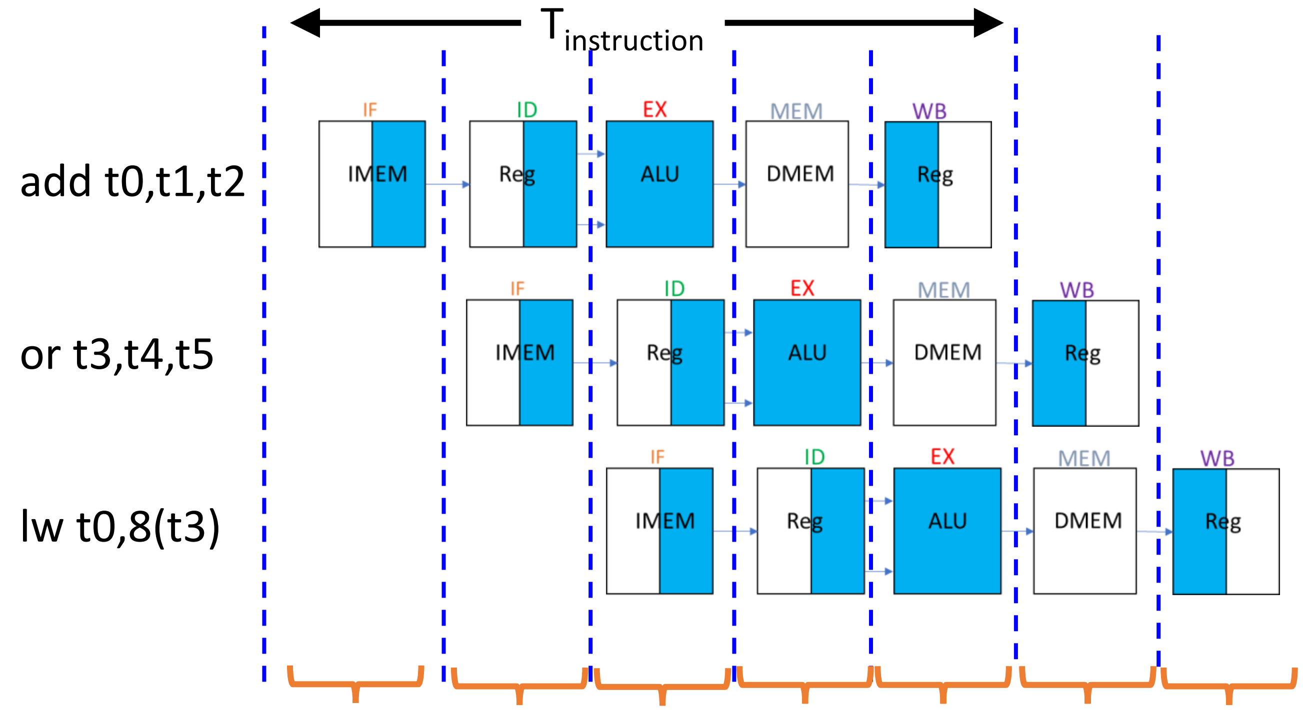
“Sequential” RISC-V Datapath

In a **single-cycle CPU**, only one instruction can access any resources in one clock cycle, in sequence



“Pipelined” RISC-V Datapath

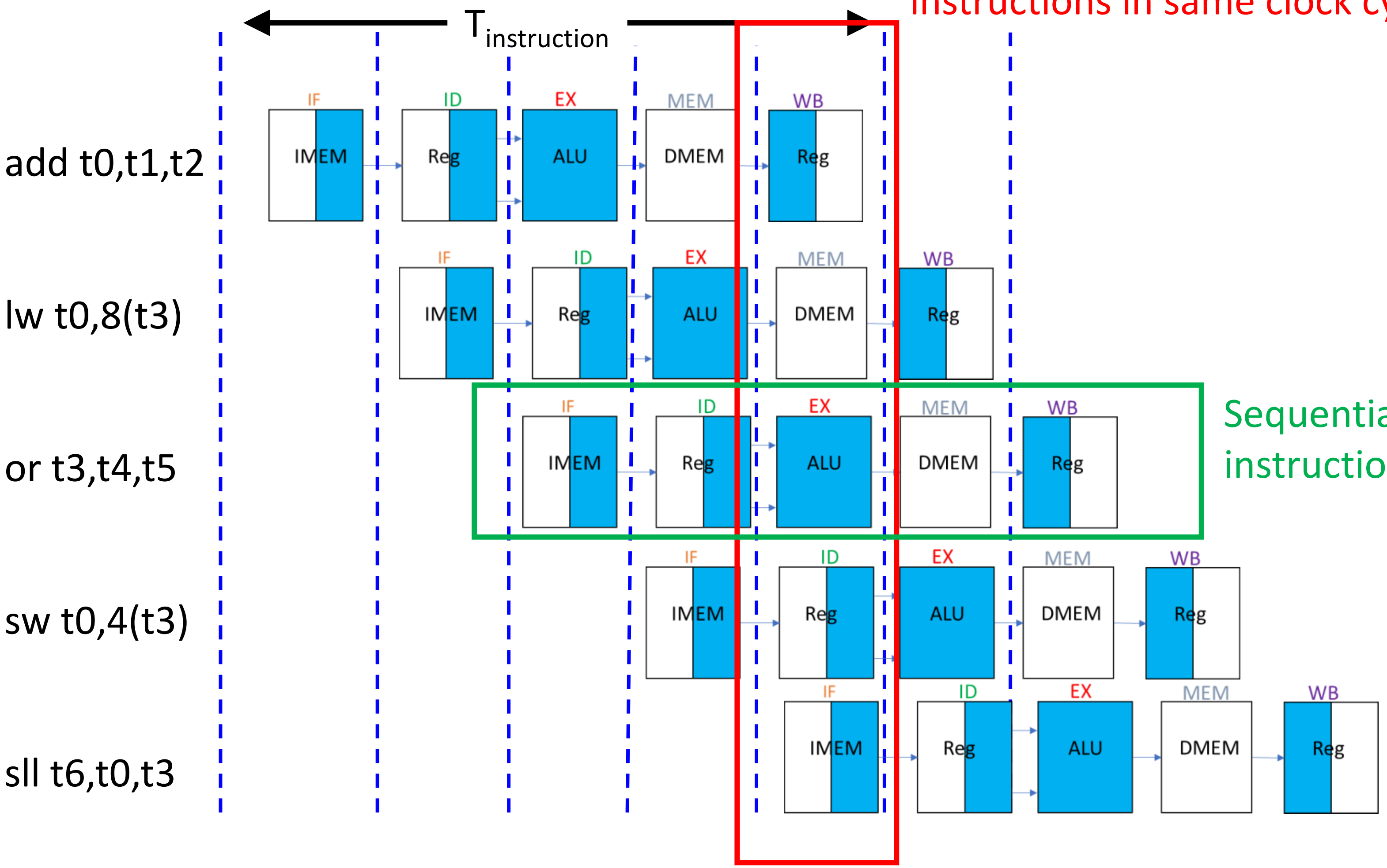
In a **pipelined CPU**, multiple instructions access resources in one clock cycle



T_{cycle} = 200 ps

What Happens Sequentially? Simultaneously?

Simultaneous: Resource use by multiple instructions in same clock cycle.



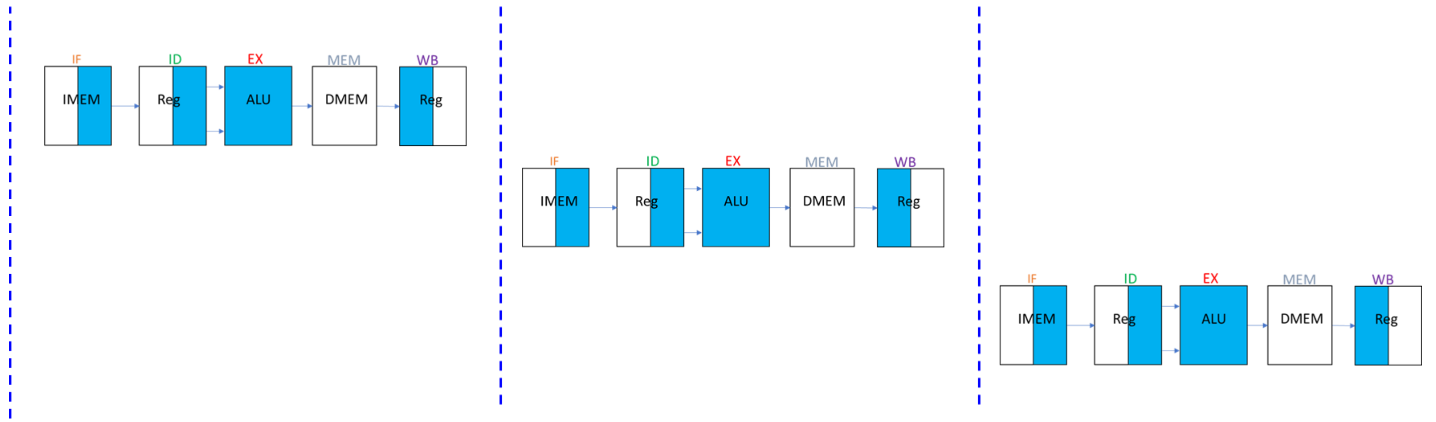
Sequential: Resource use by same instruction over time (multiple clock cycles)

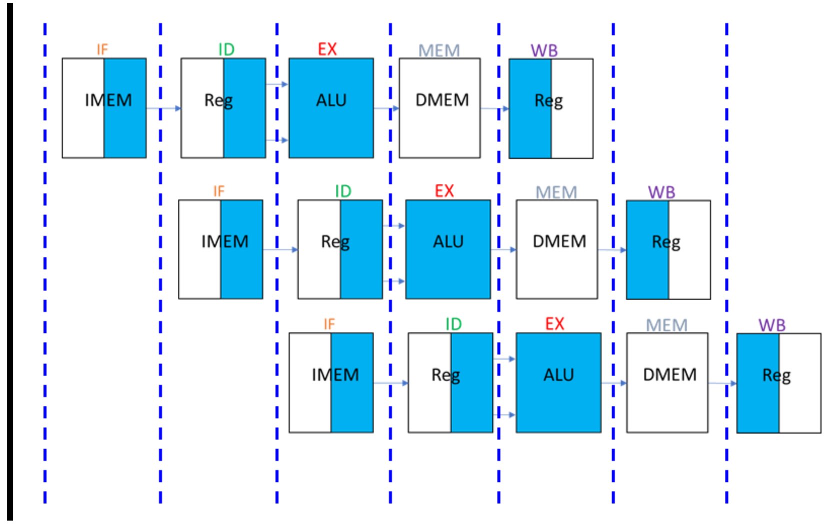
Performance: Latency and Throughput

add t0,t1,t2

or t3,t4,t5

lw t0,8(t3)





1/throughput

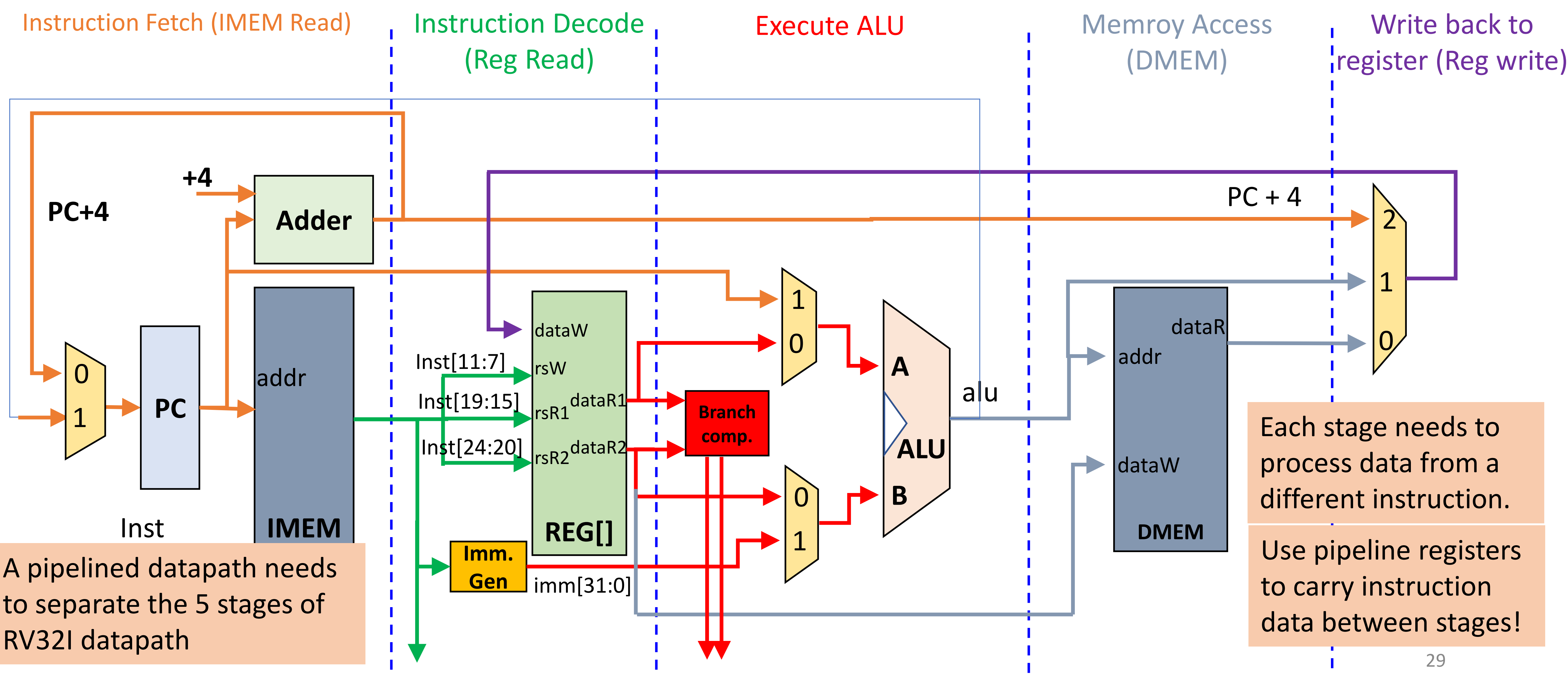
$$\frac{Time}{Program} = \frac{Instructions}{Program} \times \boxed{\frac{Cycles}{Instruction} \times \frac{Time}{Cycle}}$$

CPI, inverse of (# instrs executed/cycle)

Processor throughput = #instructions / time

	Single-cycle	Pipelined
Timing of each stage	T_{stage} = 200,100,200,200,100 ps (Reg access stages ID, WB only 100 ps)	T_{stage} =200 ps All stages same length
Instruction time (Latency)	$T_{instruction}$ = 800 ps	$T_{instruction}$ = 5 x T_{cycle} = 1000ps
Clock cycle time, T_{cycle} Clock rate, F_s = 1 / T_{cycle}	T_{cycle} = $T_{instruction}$ = 800 ps F_s = 1/ 800 ps = 1.25 GHz	T_{cycle} = T_{stage} = 200 ps F_s = 1/ 200 ps = 5 GHz
CPI (Cycle Per Instruction)	~1 (ideal)	~1 (ideal), < 1, actual
Relative throughput gain	1x	4x

Constructing a Pipelined RV32I Datapath

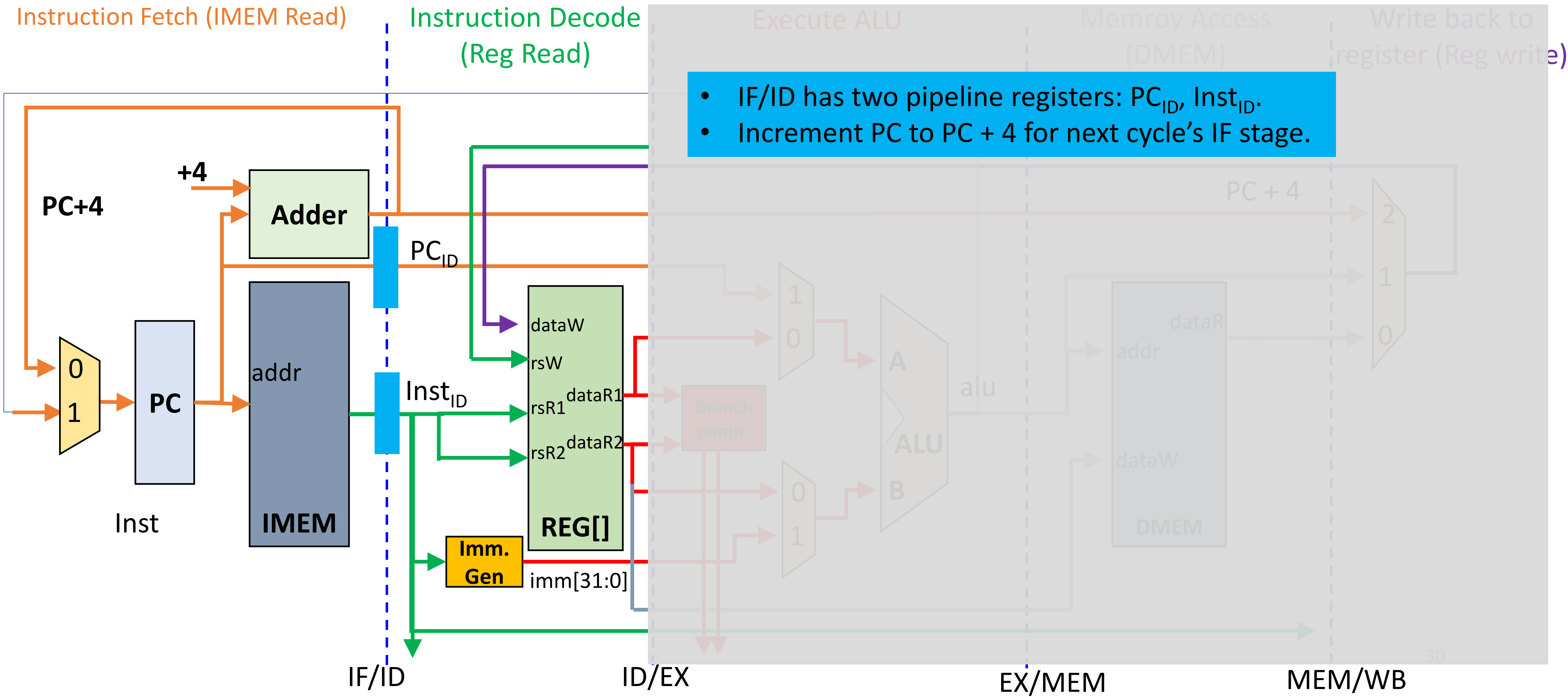


A pipelined datapath needs to separate the 5 stages of RV32I datapath

Each stage needs to process data from a different instruction.

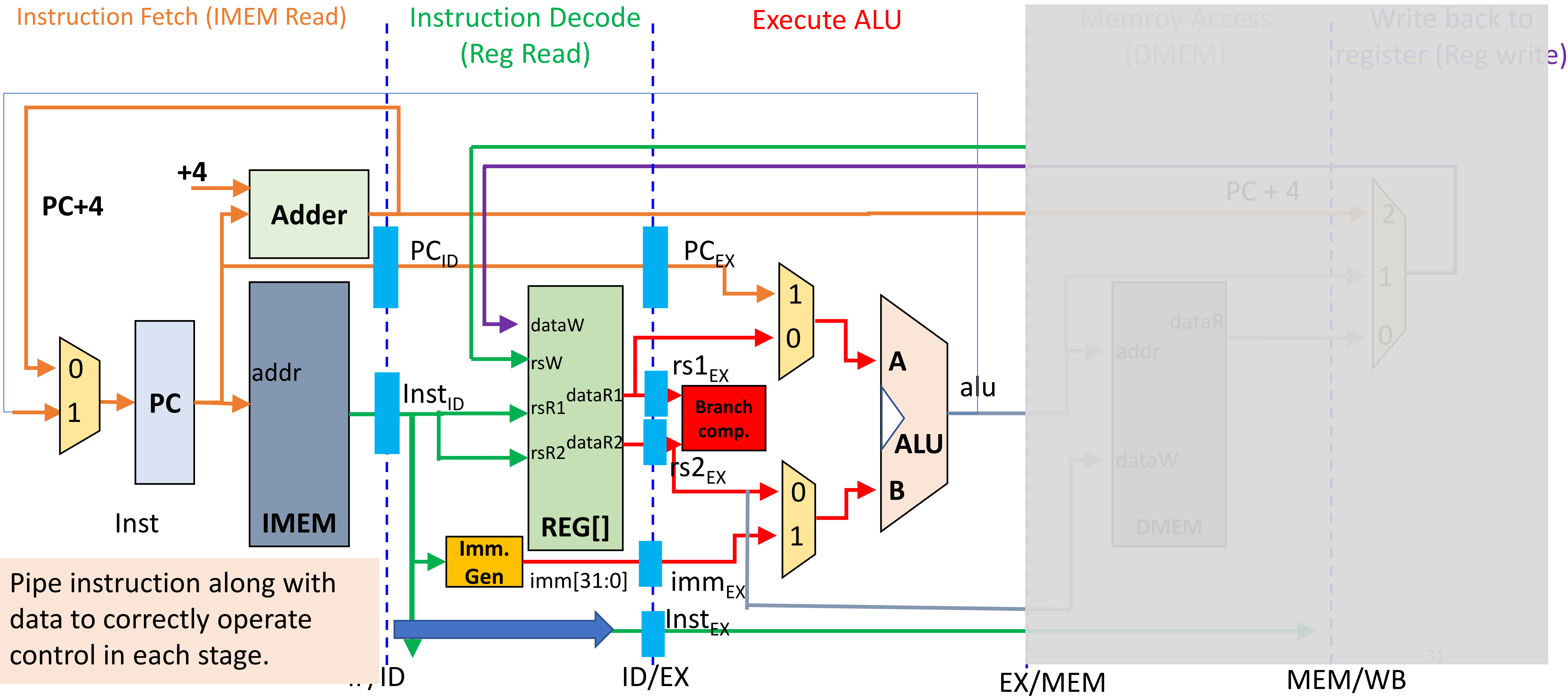
Use pipeline registers to carry instruction data between stages!

IF/ID Pipeline Registers



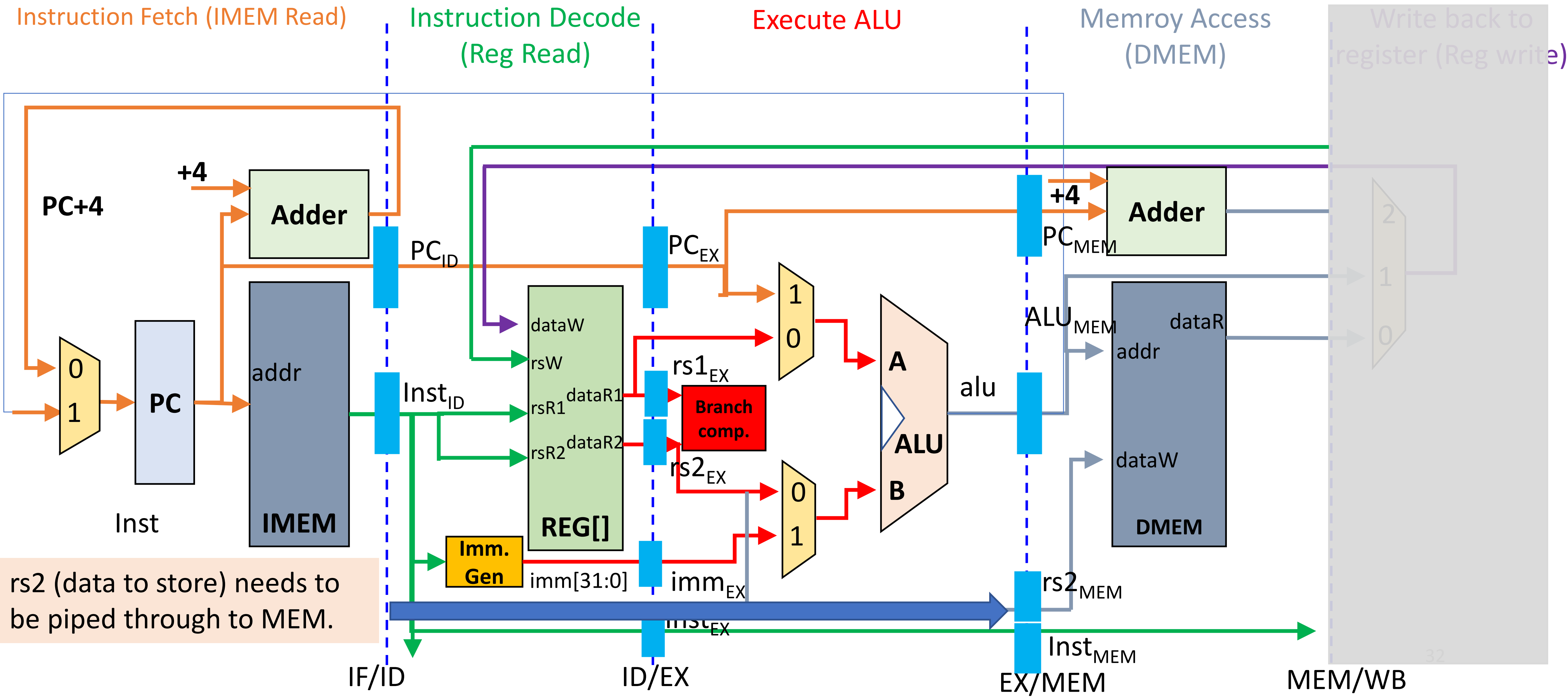
- IF/ID has two pipeline registers: PC_{ID}, $Inst_{ID}$.
- Increment PC to PC + 4 for next cycle's IF stage.

ID/EX Pipeline Registers



Pipe instruction along with data to correctly operate control in each stage.

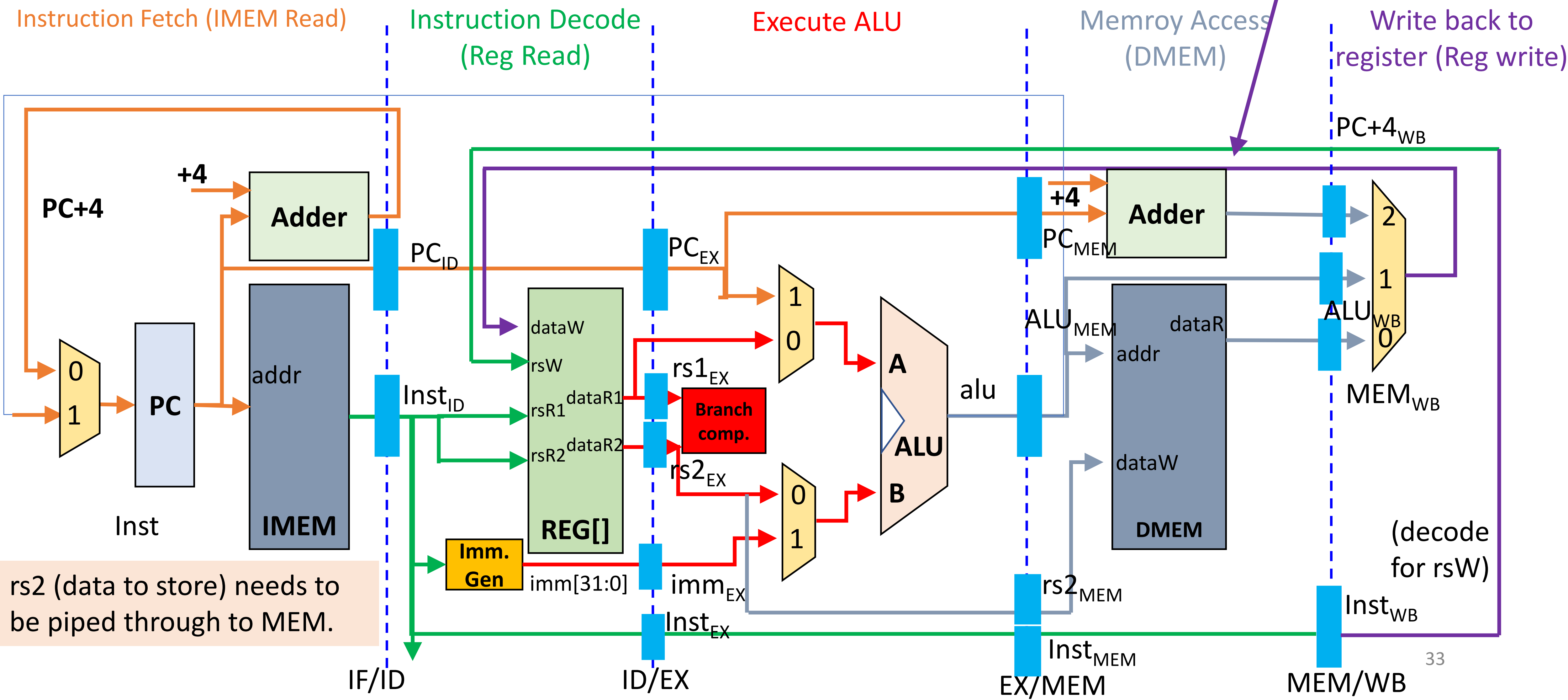
EX/MEM Pipeline Registers



rs2 (data to store) needs to be piped through to MEM.

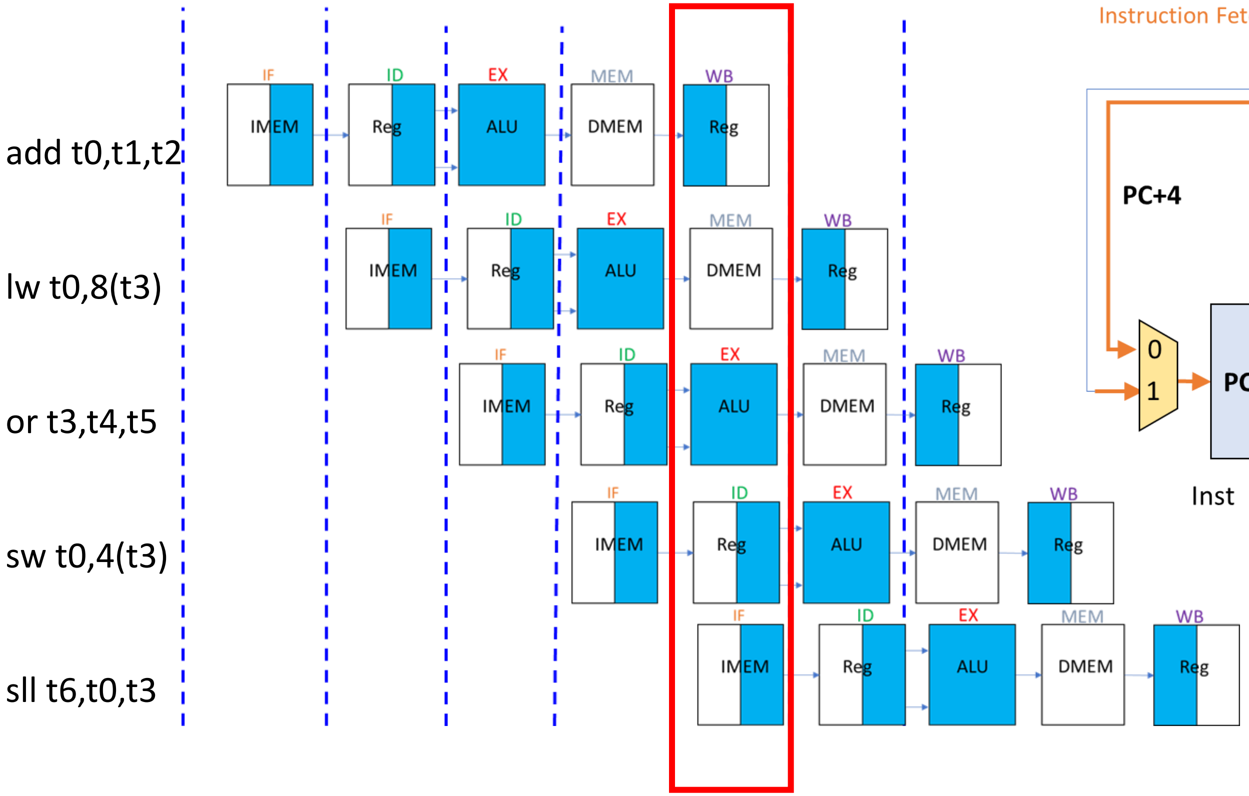
MEM/WB Pipeline Registers

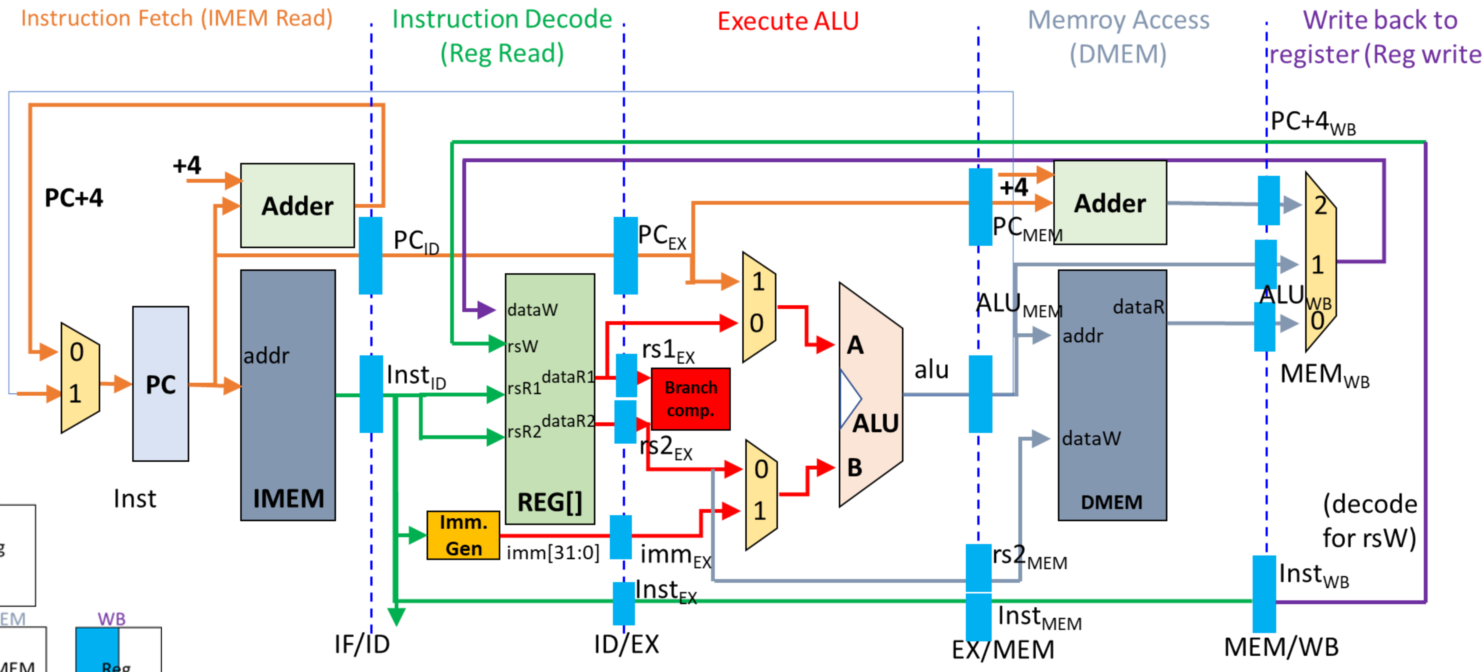
Recalculate PC+4 to avoid sending both PC, PC+4 down pipeline.



rs2 (data to store) needs to be piped through to MEM.

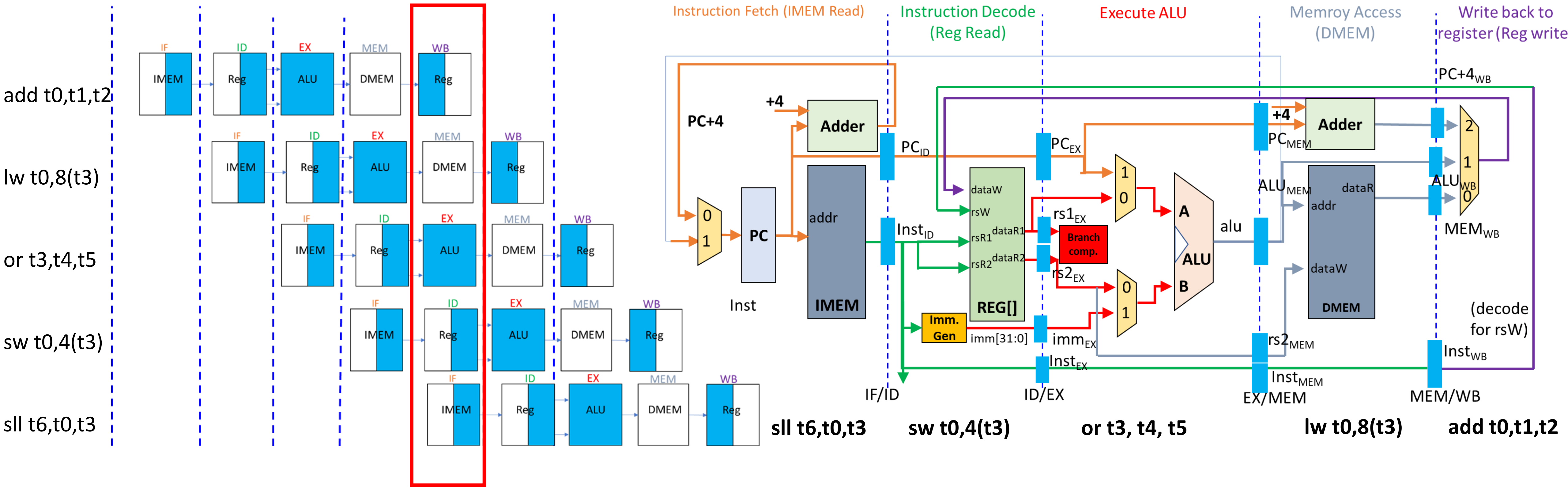
Instruction Placement





Suppose the CPU is currently executing this clock cycle.

Instruction Placement cont ...



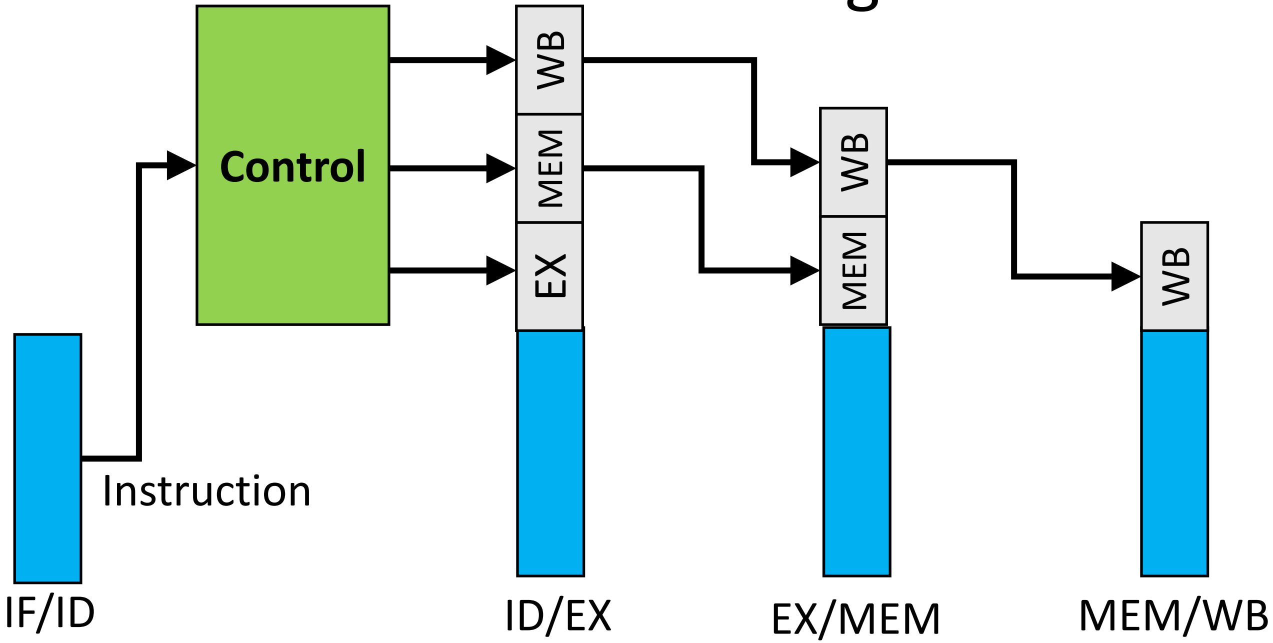
Suppose the CPU is currently executing this clock cycle.

The leftmost stage (IF) contains the most recent instruction. On the next clock cycle, pipeline registers carry the instruction/data to the next stage (ID).

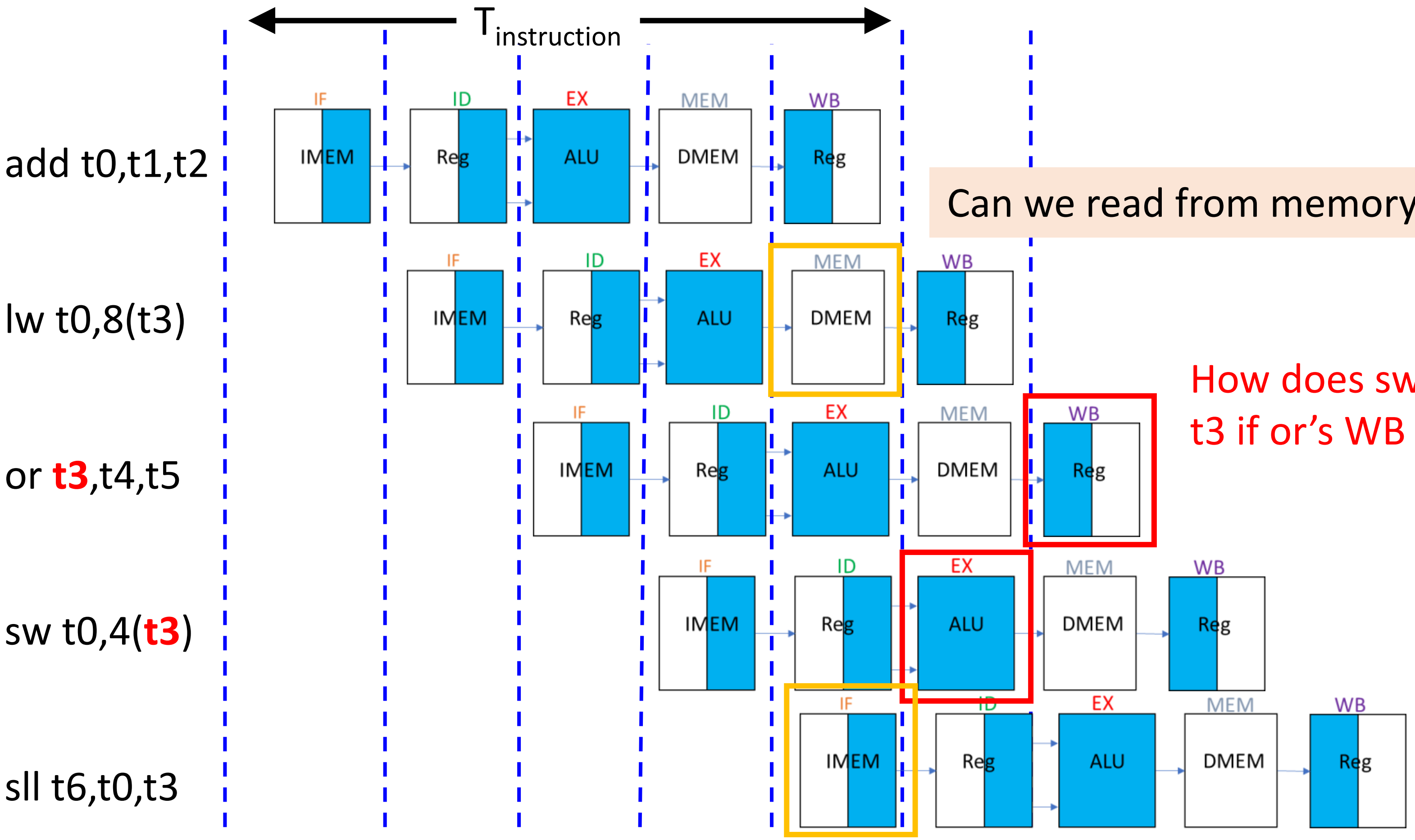
Control is also Pipelined

- Control signals are derived from the instruction
  - Like in the single-cycle CPU, control is usually computed during instruction decode (ID).

- Control information for later stages is stored in pipeline registers



Pipeline Hazards!



Can we read from memory twice in the same clock cycle?

How does sw's EX stage get the right value of t3 if or's WB stage hasn't executed yet??

Three Type of Pipeline Hazards

- A hazard is a situation in which a planned instruction cannot execute in the proper clock cycle.
- Structural hazard
  - Hardware does not support access across multiple instructions in the same cycle.
- Data hazard
  - Instructions have data dependency.
  - Need to wait for previous instruction to complete its data read/write.
- Control hazard
  - Flow of execution depends on previous instruction.

Structural Hazard

- Structural hazard
  - Hardware does not support access across multiple instructions in the same cycle.
- Occurs when multiple instructions compete for access to a single physical resource
- Solution 1 – Inefficient
  - Instructions take turns using the resource.
  - Some instructions **stall** while the resource is busy.
- Solution 2 – Add more hardware
  - Can always solve structural hazards by adding more HW.
  - In our current CPU, structural hazards are not an issue.

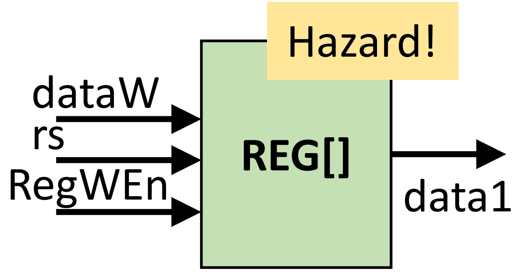
The RV32I ISA datapath avoids structural hazards via its hardware requirements on RegFile and Memory.

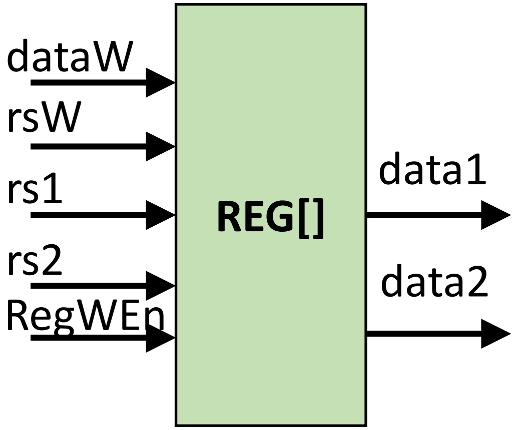
Required RegFile Avoids Structural Hazards

Universität Bremen

- Each RV32I instruction
  - Reads up to 2 operands in ID (decode) stage; and
  - Writes up to 1 operand in WB (writeback) stage.
- Structural Hazard can occur if RegFile HW does not support simultaneous Read/Write
- RV32I's required RegFile design works
  - Two independent read ports, one independent write port.
  - Three accesses (2 read, 1 write) can happen in the same cycle.

add t0,t1,t2

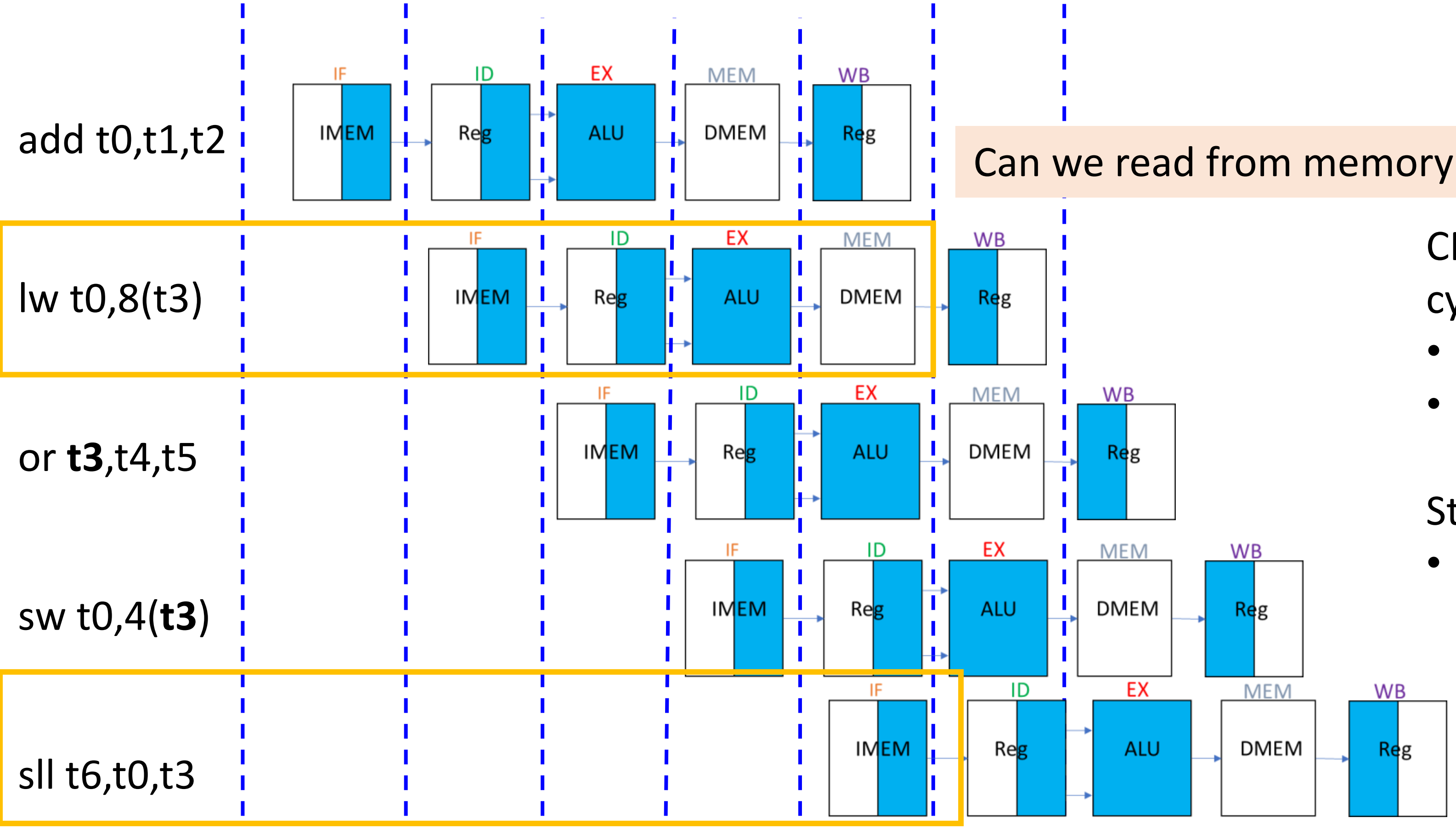




Separate IMEM, DMEM Memories

Universität Bremen

RV32I's required separation of IMEM and DMEM works



Can we read from memory twice in the same clock cycle?

CPU can read the memory twice in the same cycle

- IF: Instruction memory (IMEM)
- MEM: data memory (DMEM)

Structural Hazard if IMEM and DMEM same HW

- Without separate memories, instruction fetch would have to stall for a cycle.

Universität Bremen

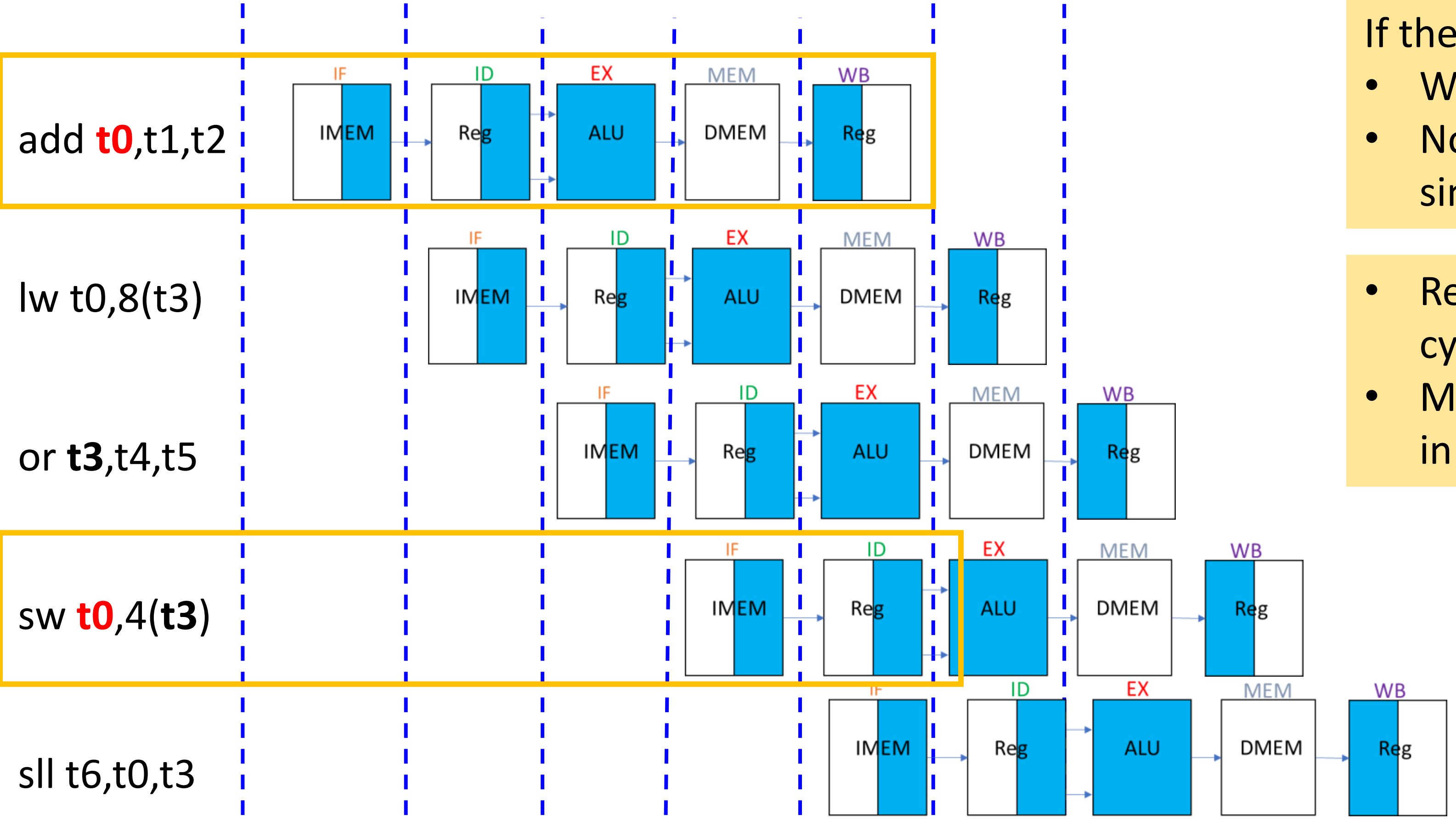
Data Hazard

- Data hazard
  - Instructions have data dependency.
  - Need to wait for previous instruction to complete its data read/write.
- Occurs when an instruction reads a register before a previous instruction has finished writing to the register
- Three cases to consider
  - Register access
  - ALU Result
  - Load data hazard

Example:
add **x19**, x0 , x1
sub x2 , **x19**, x3

Sidenote: Compilers can help!

Data Hazard 1 – Register Access

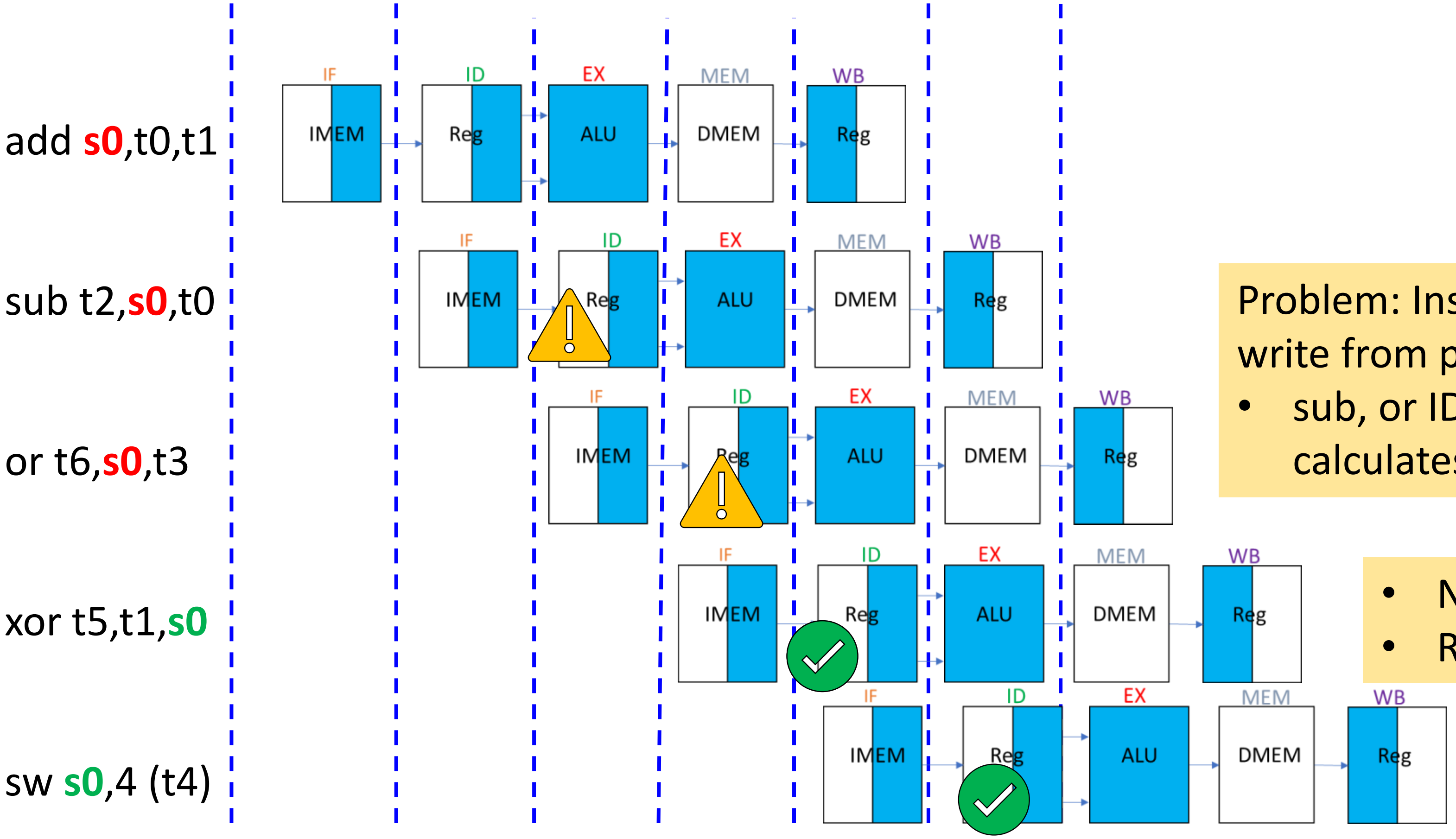


If the same register is written and read in one cycle

- WB must write value before ID reads new value.
- Not structural hazard! Separate ports allow simultaneous R/W.

- RegFile HW should write-then-read in same cycle
- Might not always be possible to write-then-read in same cycle, e.g., in high-frequency designs

Data Hazard 2 – ALU Result

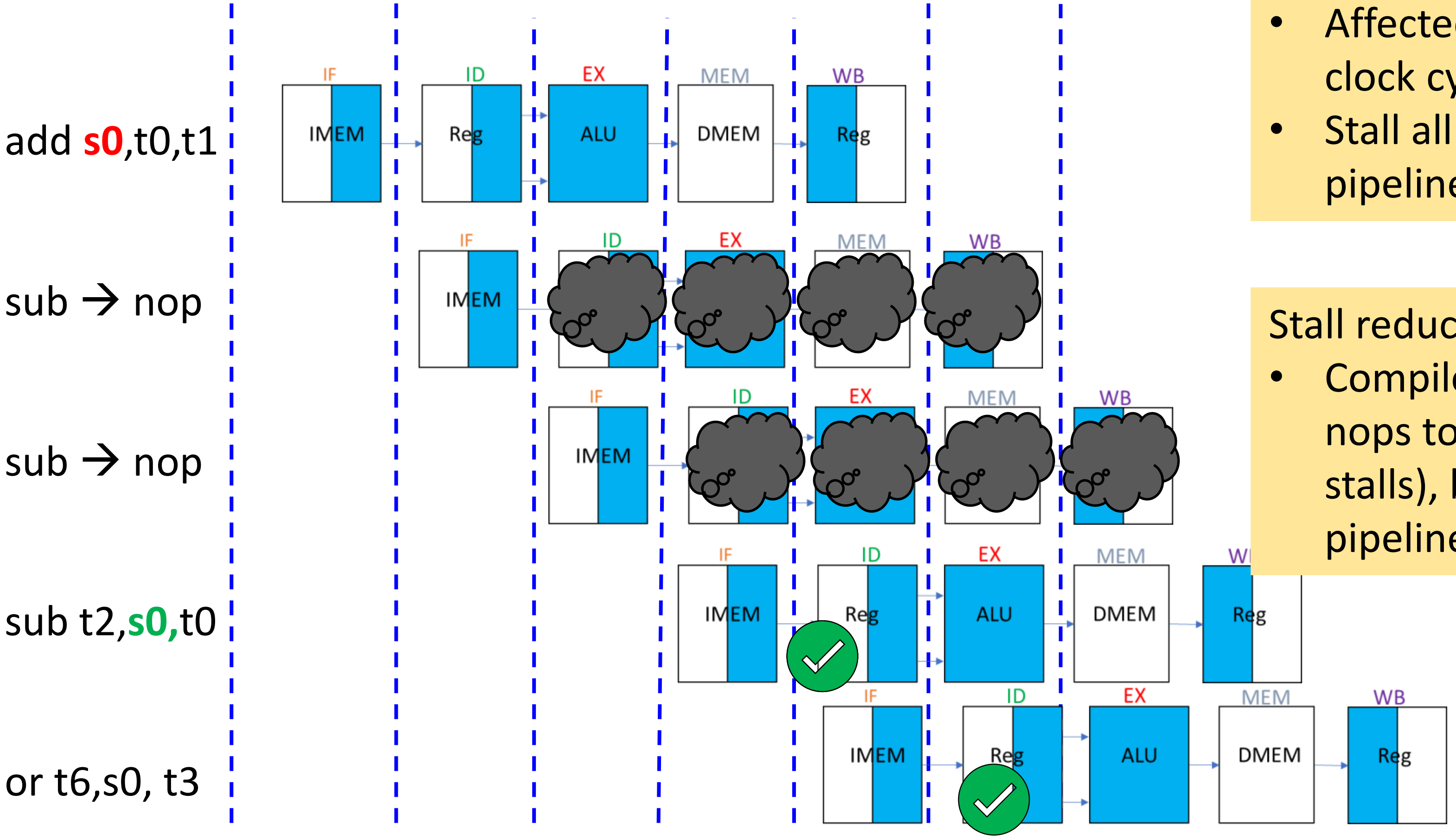


Problem: Instruction depends on WB's RegFile write from previous instruction

- sub, or ID-stage reads old value of s0 and calculates wrong result.

- Note: **xor** gets right value;
- RegFile is write-then-read.

ALU Solution 1 - Stalling



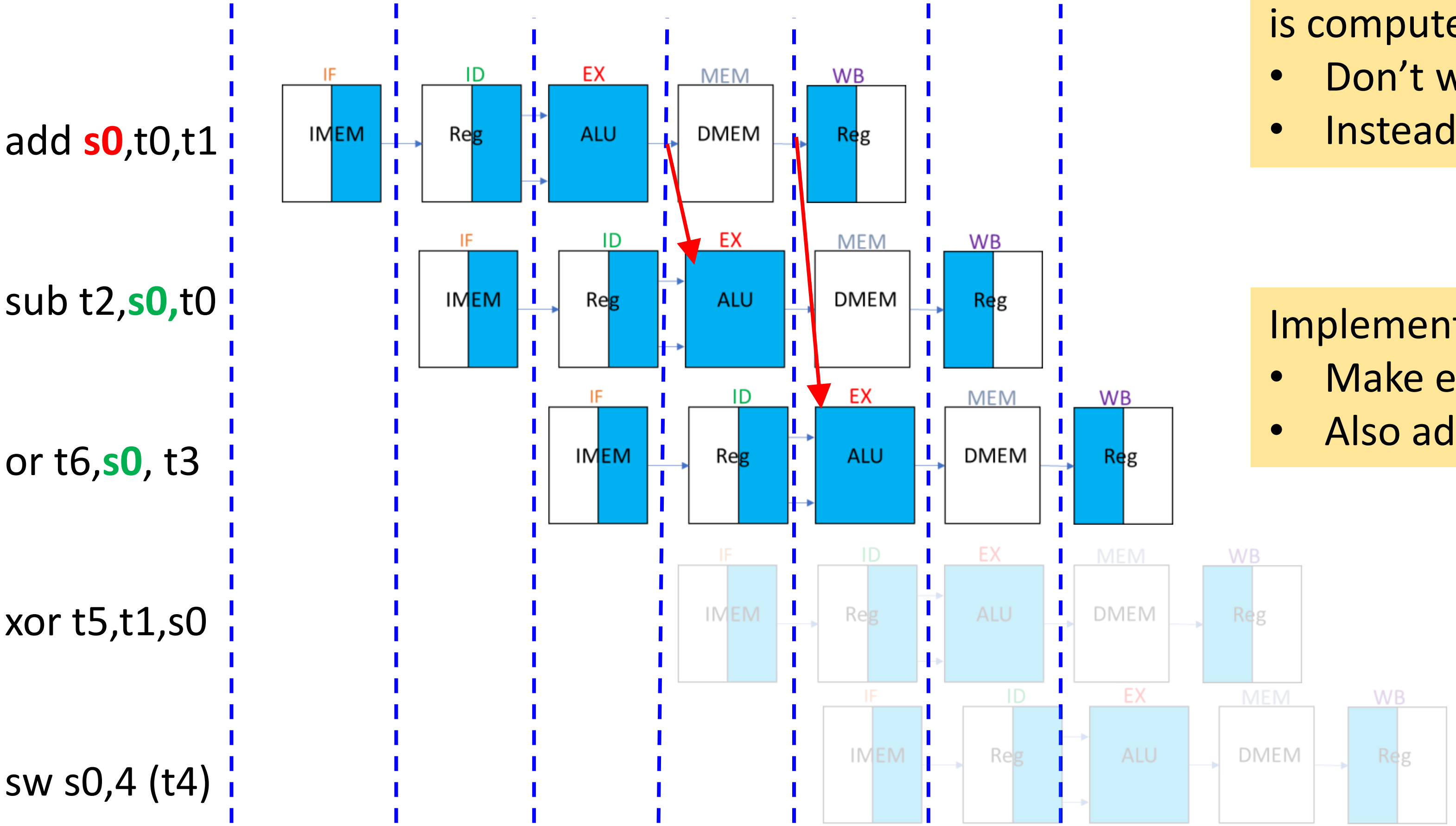
Bubble to effectively nop

- Affected pipeline stages do nothing during clock cycles.
- Stall all stages by preventing PC, IF/ID pipeline register from writing

Stall reduces performance

- Compiler could rearrange code/insert nops to avoid hazards (and therefore stalls), but this requires knowledge of the pipeline structure.

ALU Solution 2 – Forwarding



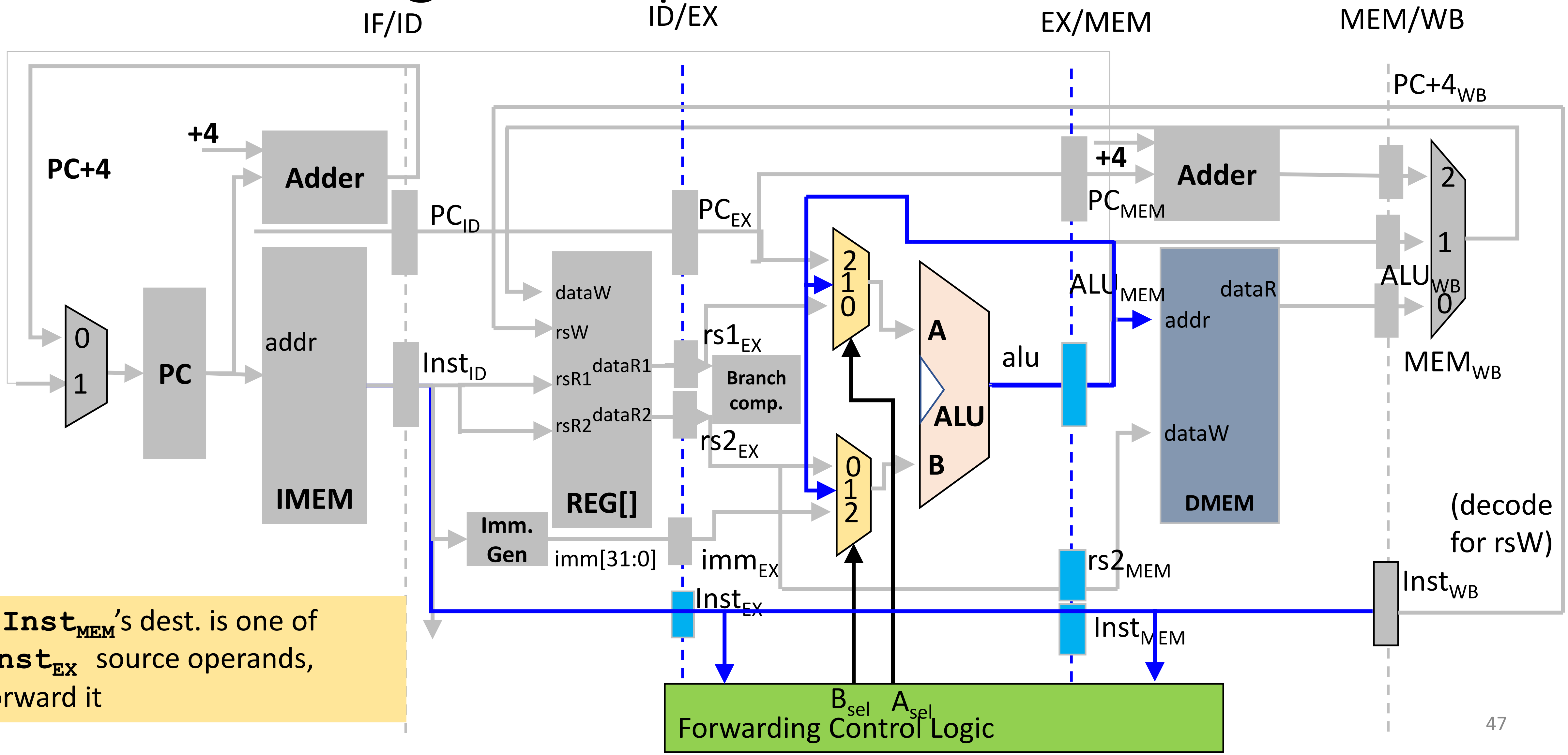
Forwarding aka bypassing uses the result when it is computed

- Don't wait for value to be stored into RegFile.
- Instead, grab operand from the pipeline stage

Implementation

- Make extra connections in the datapath.
- Also add forwarding control logic

Forwarding EX output



If $\texttt{Inst}_{\texttt{MEM}}$'s dest. is one of $\texttt{Inst}_{\texttt{EX}}$ source operands, forward it

Forwarding Cannot Fix all Data Hazards

Forward EX stage output to input of EX stage on next clock cycle.

Forward MEM stage output to input of EX stage on next clock cycle.

Forwarding Cannot Fix all Data Hazards

Forward EX stage output to input of EX stage on next clock cycle.

MEM stage (lw)’s output needed as EX stage (or)’s input in the same clock cycle

Forward MEM stage output to input of EX stage on next clock cycle.

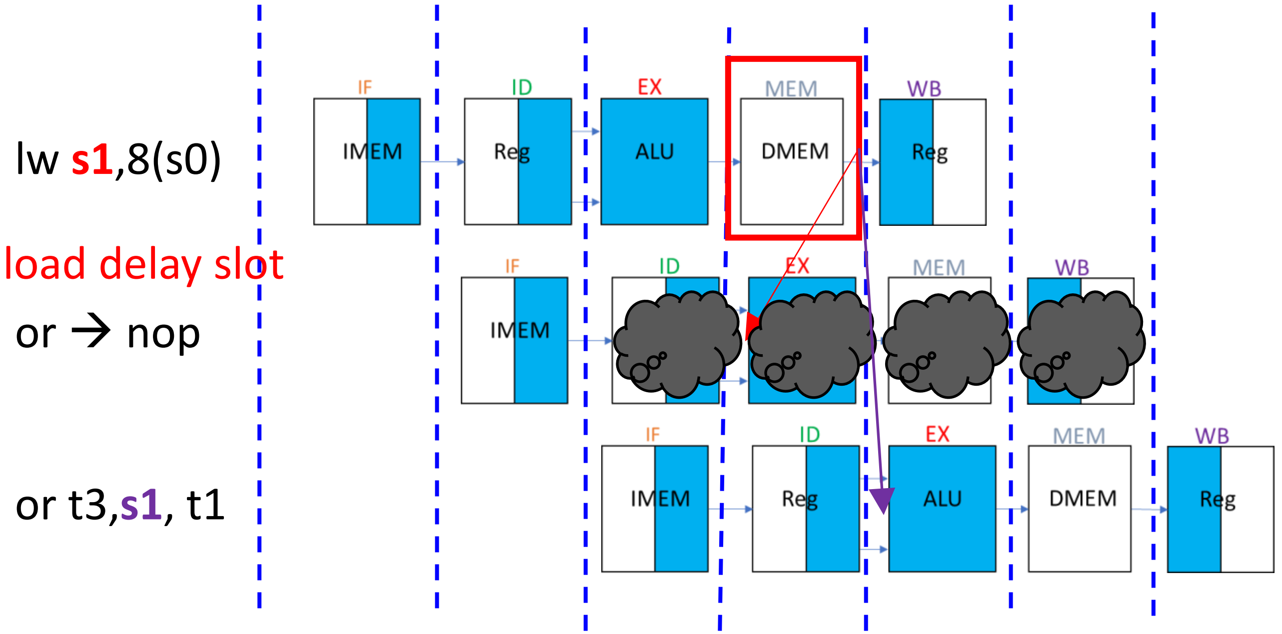
Loads can result in an unavoidable pipeline stall.

Data Hazard 3 – Loads

The instruction after a load is called the **load delay slot**

If this instruction uses the result of load:

- The hardware must stall for one cycle (plus forwarding).
- This results in performance loss!



MEM stage (lw)'s output needed as EX stage (or)'s input in the same clock cycle.

Forwarding sends data to the next clock cycle. Cannot go backwards in time!

Solution – Code Scheduling

- Fix this hazard at code compilation stage
  - In the delay slot, put an instruction unrelated to the load result.
  - No performance loss!

C Code

A[3] = A[0] + A[1];
A[4] = A[0] + A[2];

Code scheduling: With knowledge of the underlying CPU pipeline, the compiler reorders code to improve performance

Simple compilation
(9 cycles for 7 instructions)

Stall & forward! (+1 cycle)

(+1 cycle)

```
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)
```

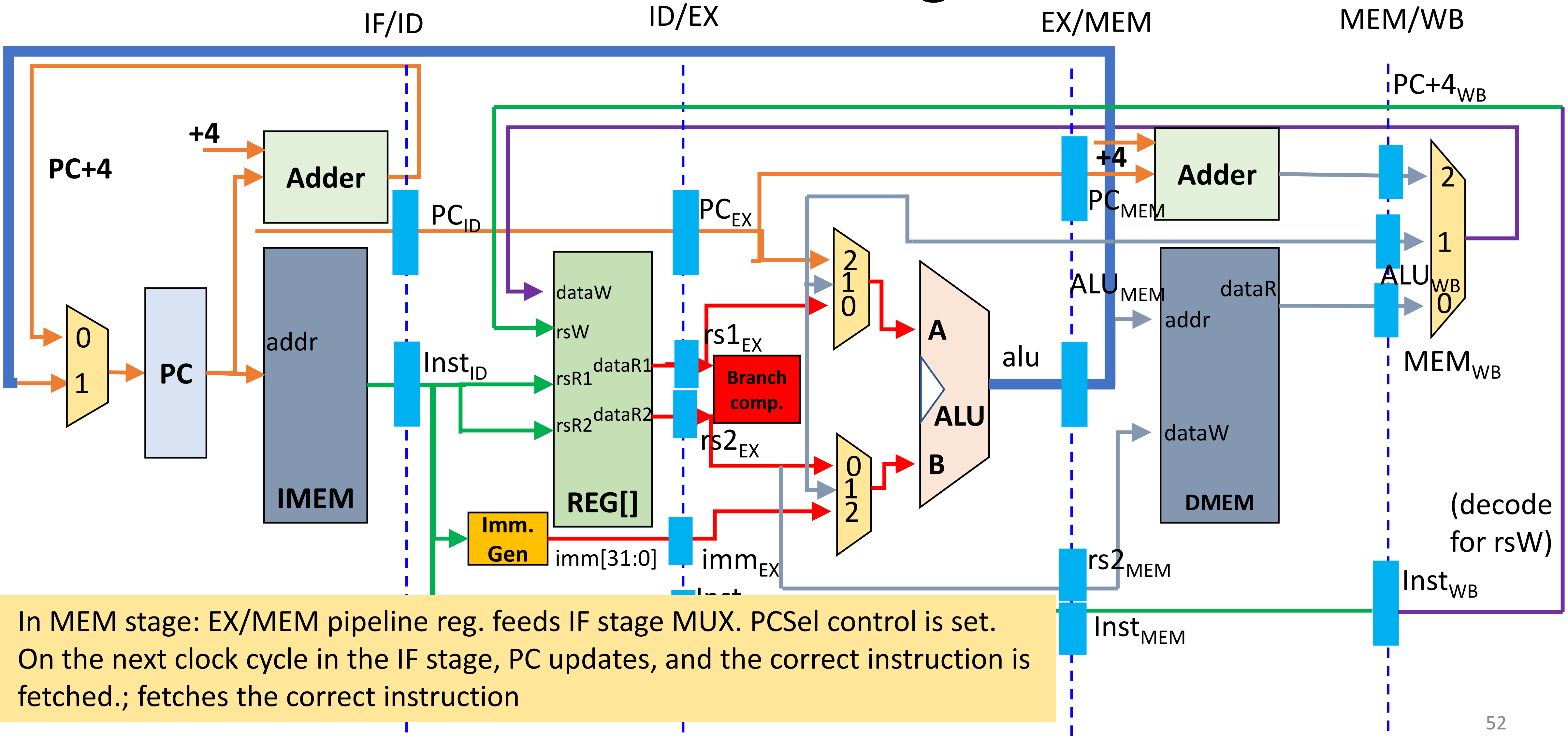
Alternative
(7 cycles)

```
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)
```

Forward!
(+0 cycle)

Universität Bremen

Branch Results at MEM Stage



- In MEM stage: EX/MEM pipeline reg. feeds IF stage MUX. PCSel control is set.
- On the next clock cycle in the IF stage, PC updates, and the correct instruction is fetched.; fetches the correct instruction

Control Hazard – Conditional Branches

Control hazards occur when the instruction fetched may not be the one needed, e.g., if beq branch is taken ….

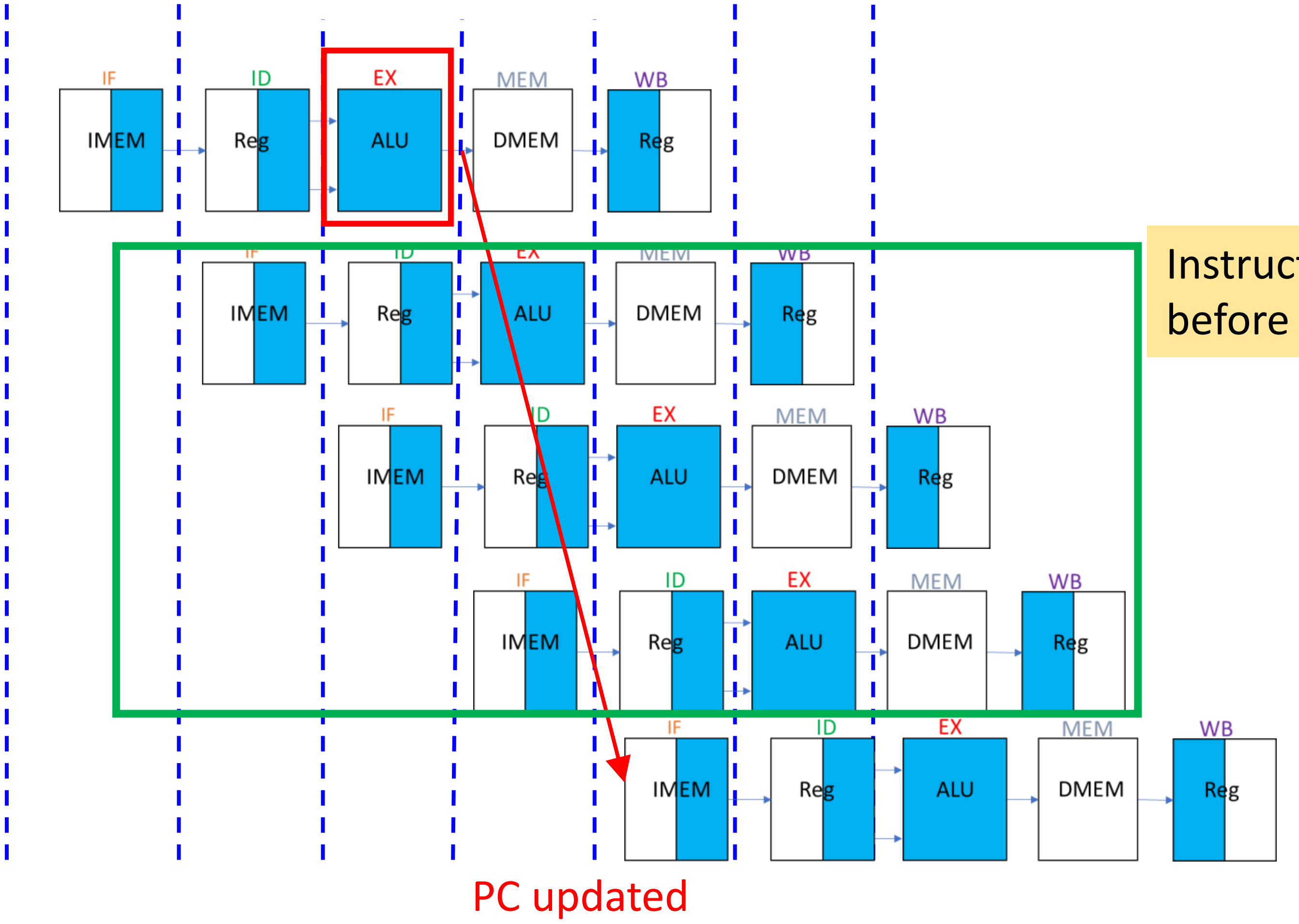
0x40 beq t0,t1,Label

0x44 sub t2,s0,t0

0x48 or t6,s0,t3

0x4c xor t5,t1,s0

0x70 sw s0,8(t3) #Label



Instruction execution starts before branch outcome is known!

Correct instructions starts executing

Kill Instructions After Branch (If Taken)

Control hazards occur when the instruction fetched may not be the one needed, e.g., if beq branch is taken ....

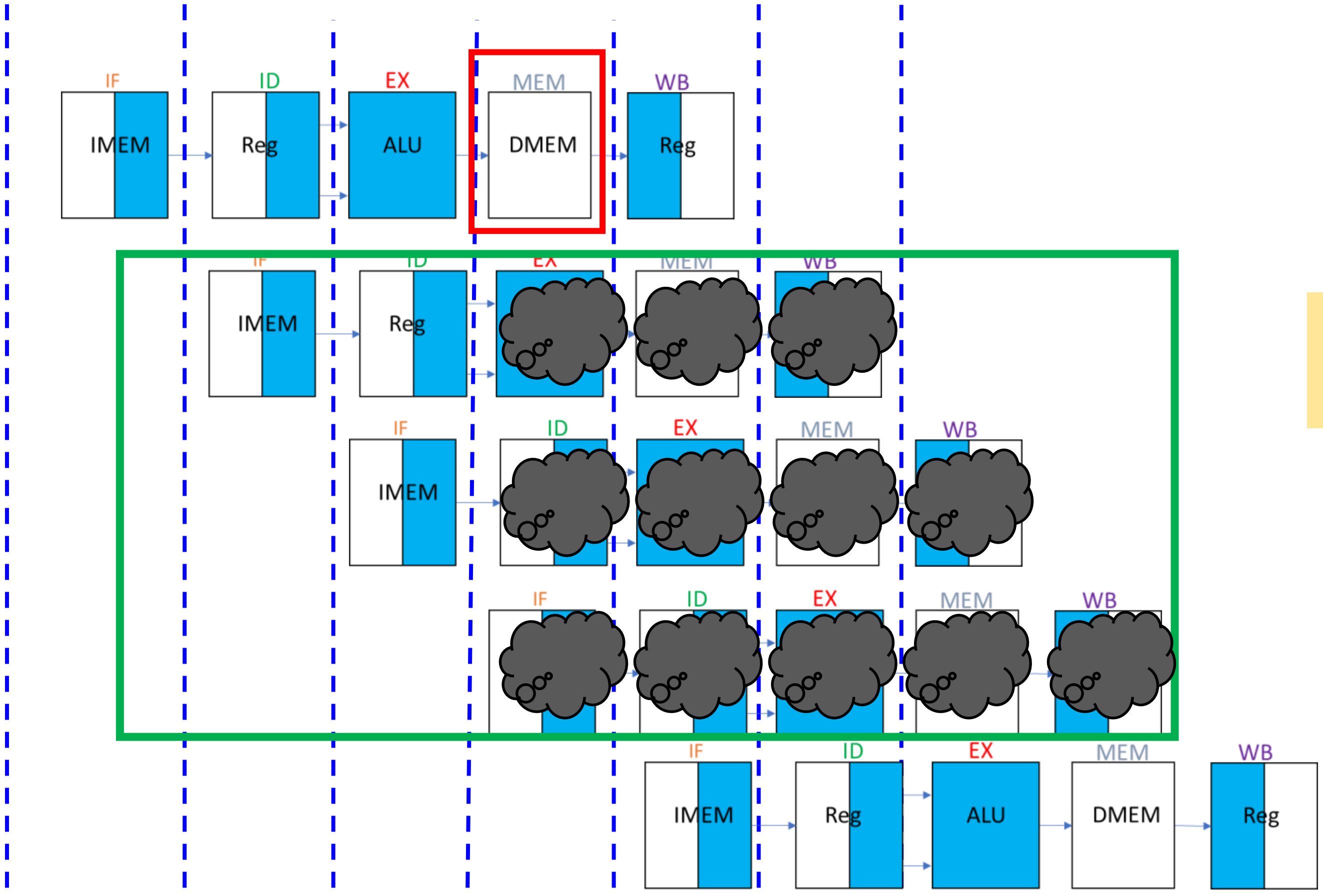
0x40 beq t0,t1,Label

0x44 sub t2,s0,t0

0x48 or t6,s0,t3

0x4c xor t5,t1,s0

0x70 sw s0,8(t3) #Label



PC updated

Flush pipeline by converting incorrect instructions to nops

PC updated, correct instruction loaded

Branch Prediction to Reduce Penalties

- Every branch taken in the simple RV32I pipeline costs 3 clock cycles
  - Note if branch is not taken, then pipeline is not stalled; the correct instructions are correctly fetched sequentially after the branch instruction.

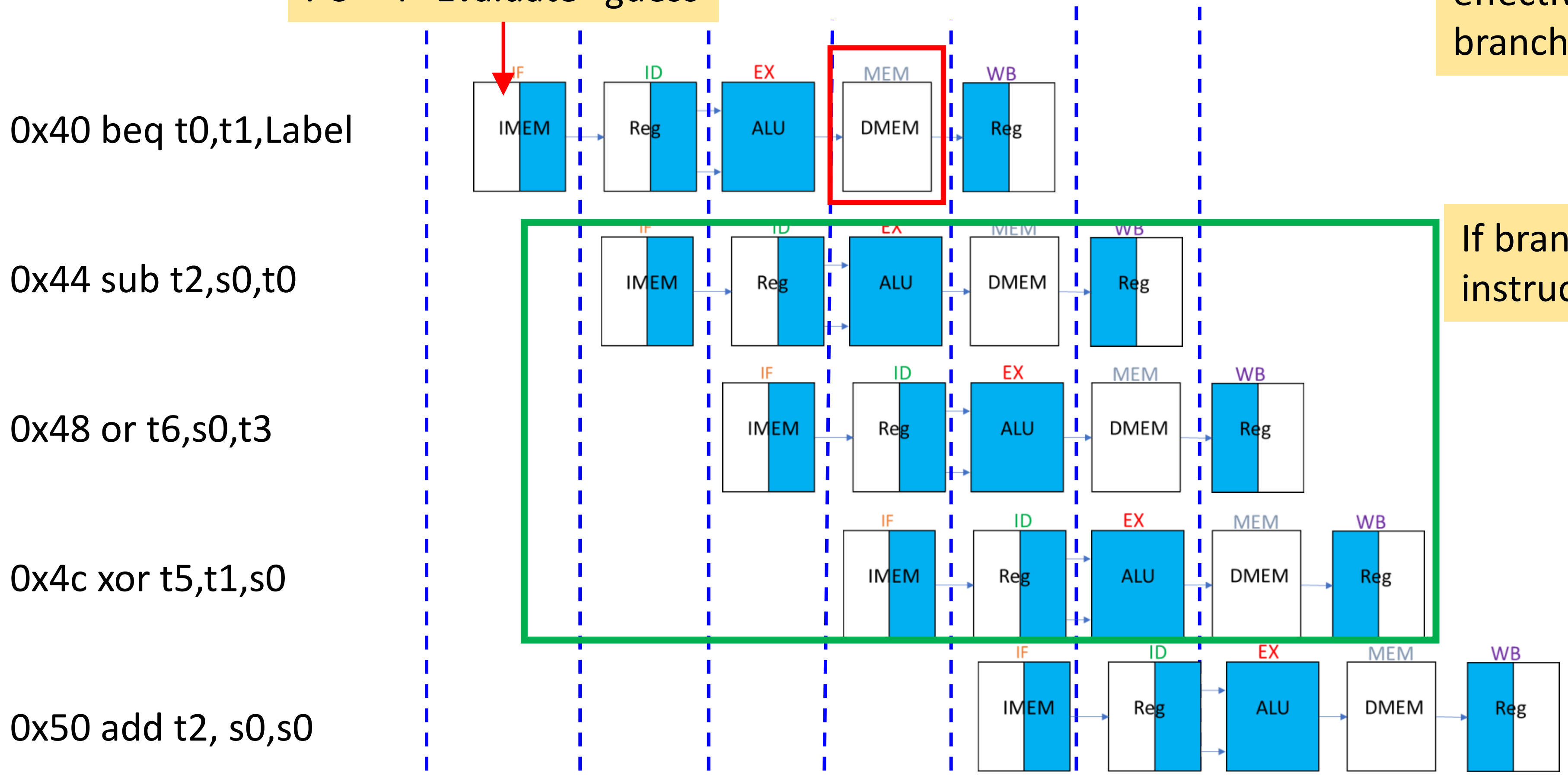
- We can improve the CPU performance on average through branch prediction
  - Early in the pipeline, guess which way branches will go.
  - Flush pipeline if branch prediction was incorrect

Naïve Predictor – Don’t Take Branch

“Guess” next PC to be PC + 4 “Evaluate” guess

Evaluate guess

The simple RV32I pipeline effectively always predicts that branch is not taken.



If branch not taken, correct instructions are already executed

Penalty for incorrect prediction is still 3 clock cycles! Branch prediction tries to improve average performance.

- Happy holidays!

RISC-V Pipelining

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck

Universität Bremen

- Problem
- Algorithm
- Program
- Instruction Set Architecture
- **Microarchitecture**
- Logic
- Digital Circuits
- Analog Circuits
- Devices
- Physics

Literature

- D. Patterson, J. Hennessy: Computer Organization and Design RISC-V Edition – The Hardware Software Interface, Elsevier, 2020.
- S. L. Harris, D. Harris: Digital Design and Computer Architecture, RISC-V Edition, Elsevier, 2021.
- ARM University program: Introduction-to-Computer-Architecture-Education.
- J. Teich, C. Haubelt: Digitale Hardware/Software-Systeme – Synthese und Optimierung, Springer Verlag, 2. Auflage, 2007.
- G. De Micheli: Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.
- Dan Garcia, Lisa Yan : Great Ideas in Computer Architecture (Machine Structures), CS 61C at UC Berkeley.
- G. D. Hachtel, F. Somenzi: Logic Synthesis and Verification Algorithms, Kluwer, 1996.

Literature

- H. Bähring, J. Dunkel, G. Rademacher: Mikrorechner-Systeme: Mikroprozessoren, Speicher, Peripherie, Springer-Verlag, 2. Auflage, 1994.
- J. P. Hayes: Computer Architecture and Organization, McGraw-Hill, 1998.
- M. G. Arnold: Verilog Digital Computer Design: Algorithms to Hardware, Prentice Hall, 1998.
- A. Tanenbaum: Structured Computer Organization, Prentice Hall, 5th Edition, 2006.
- D. A. Patterson, J. L. Hennessy: Computer Organization and Design - The Hardware/Software-Interface, Morgan Kaufmann, 3. Auflage, 2007.
- J. L. Hennessy, D. A. Patterson: Computer Architecture - A Quantitative Approach, Morgan Kaufmann, 4. Auflage, 2007.
- H. Kaeslin: Digital Integrated Circuit Design, From VLSI to CMOS Fabrication, Cambridge University Press, 2008.
- A. Biere, D. Kroening, G. Weissenbacher, C. M. Wintersteiger: Digitaltechnik – Eine praxisnahe Einführung, Springer-Verlag, 2008.