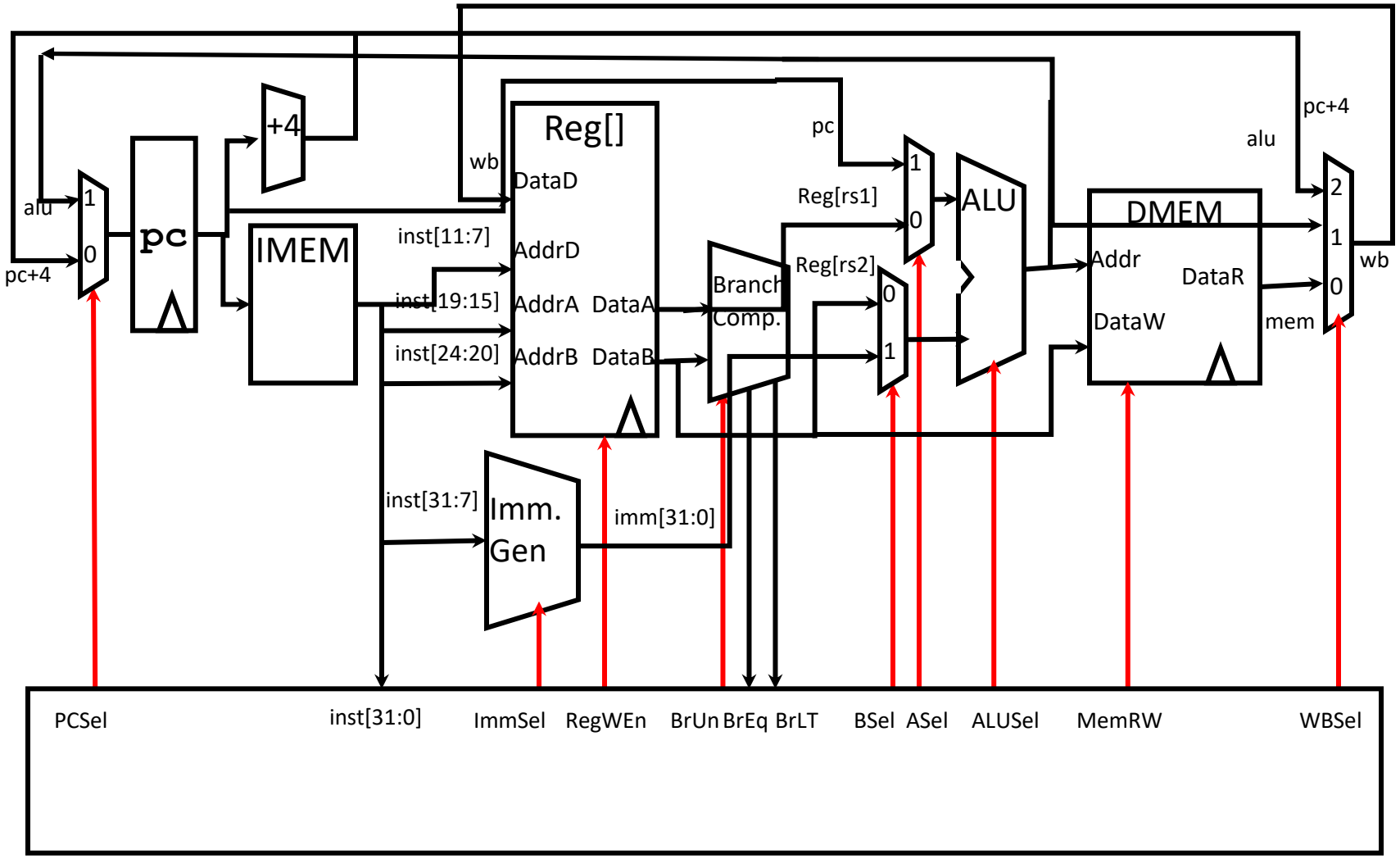
Control Signals: Big picture!

- 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 295 instructions)
  - All func3 bits
  - "2nd" upper opcode bit (others are the same for all 295 instructions)

Control Signals: ADD

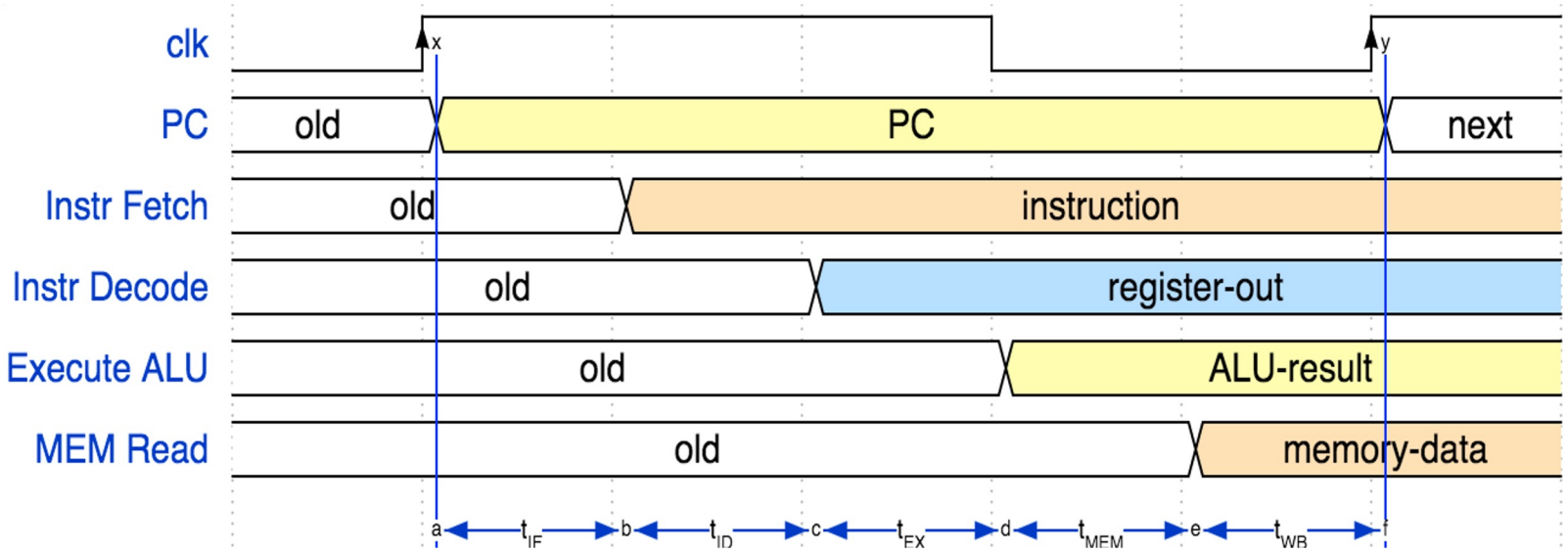


Inst[31:0]	PCSel	ImmSel	RegWEn	Br Un	Br Eq	Br LT	BSel	ASel	ALUSel	MemRW	WBSel
add	+4	*	1 (Y)	*	*	*	Reg	Reg	Add	Read	ALU
sub	+4	*	1	*	*	*	Reg	Reg	Sub	Read	ALU
(R-R Op)	+4	*	1	*	*	*	Reg	Reg	*(Op)*	Read	ALU
addi	+4	I	1	*	*	*	Imm	Reg	Add	Read	ALU
lw	+4	I	1	*	*	*	Imm	Reg	Add	Read	Mem
sw	+4	S	0 (N)	*	*	*	Imm	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

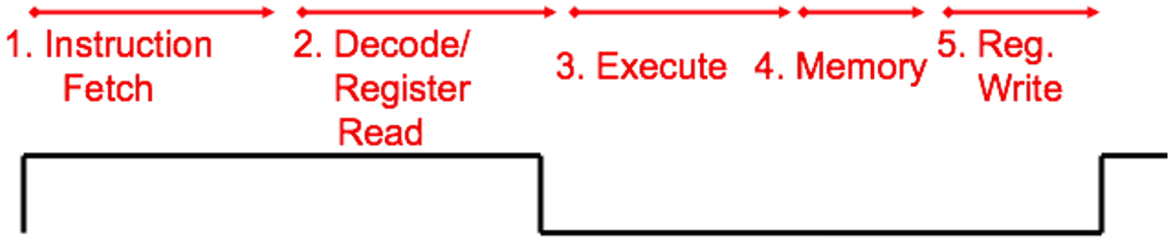
Agenda

- Quick Datapath Review
- Control Implementation
- Performance Analysis
- Pipelined Execution
- Pipelined Datapath

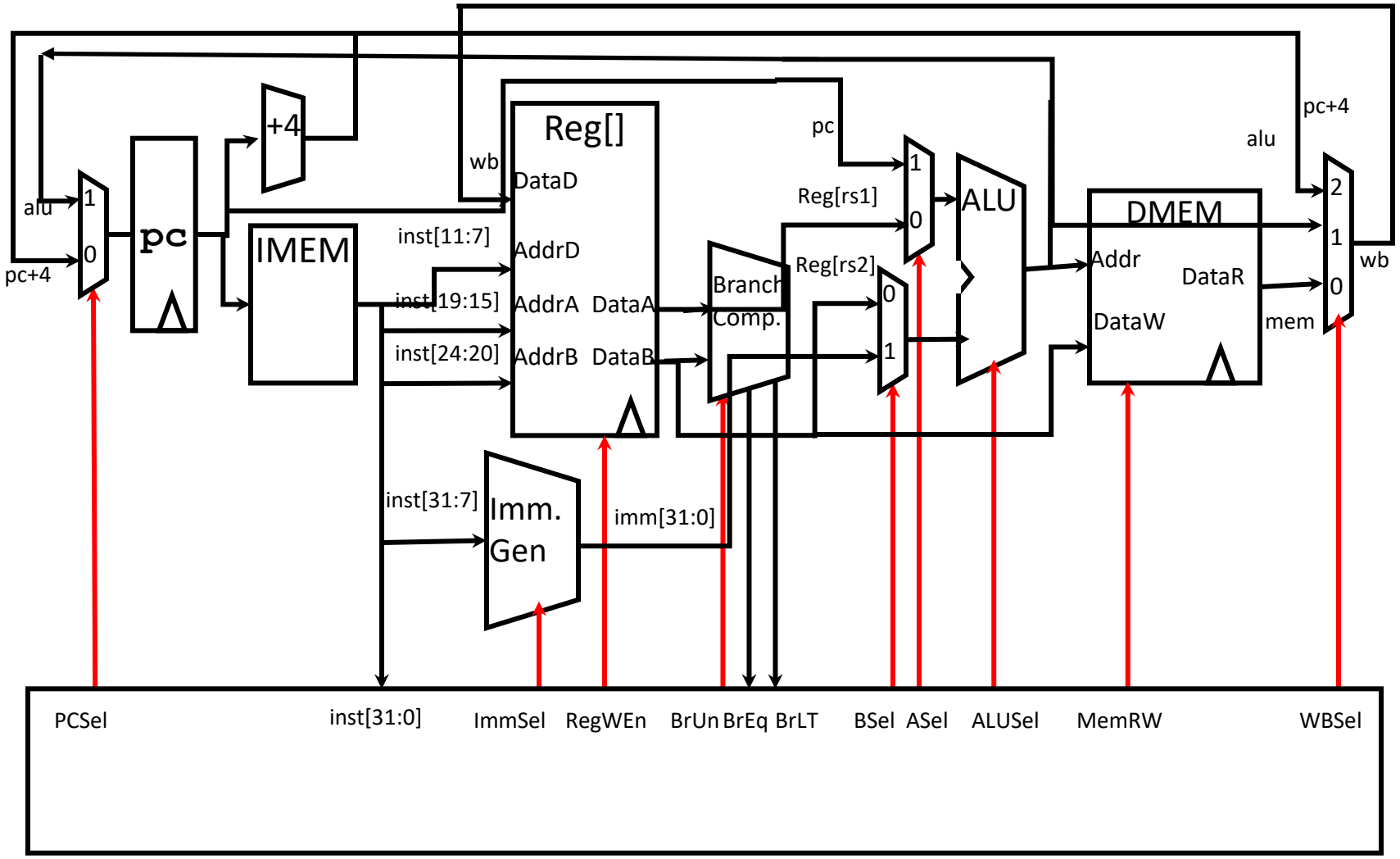
Instruction Timing



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



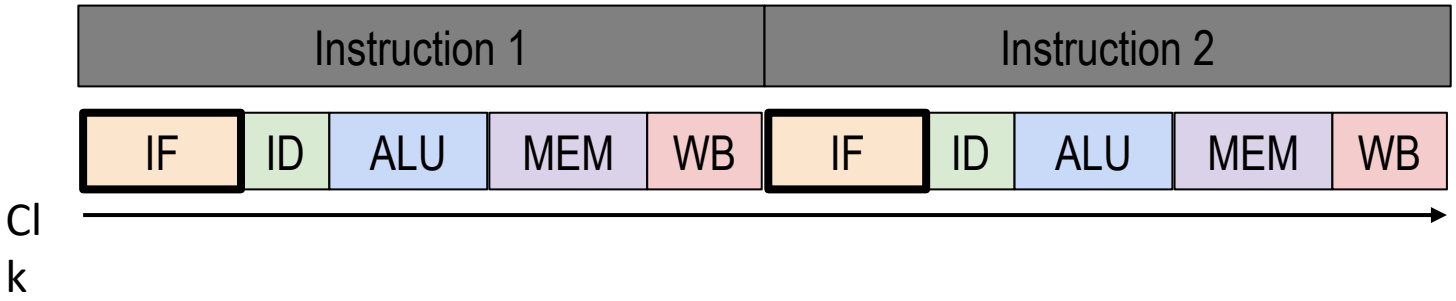
Control Signals: ADD



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



Performance Measures

- In our example, 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?

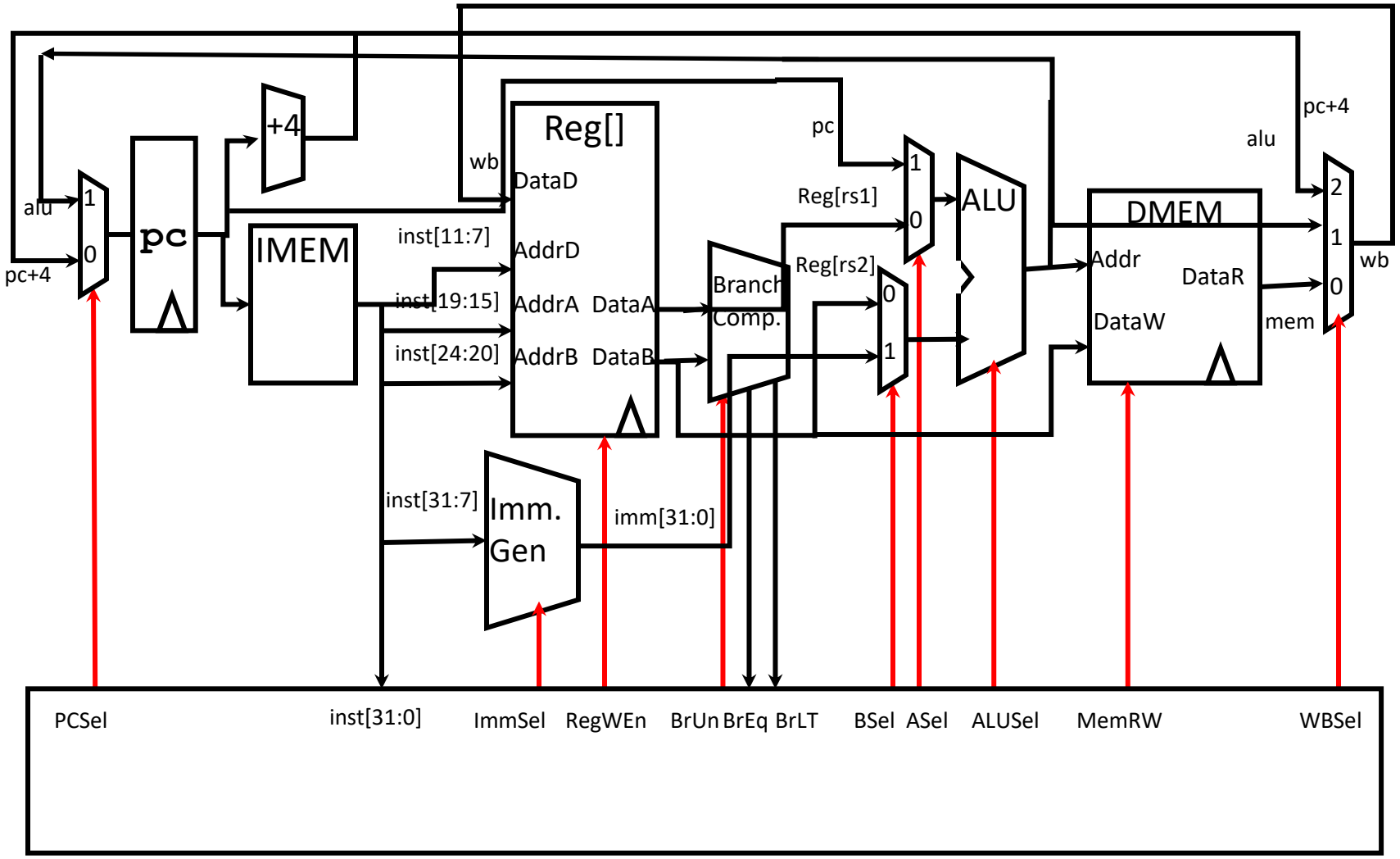
    More jobs per unit time (e.g. web server returning pages)?

    Longer battery life?

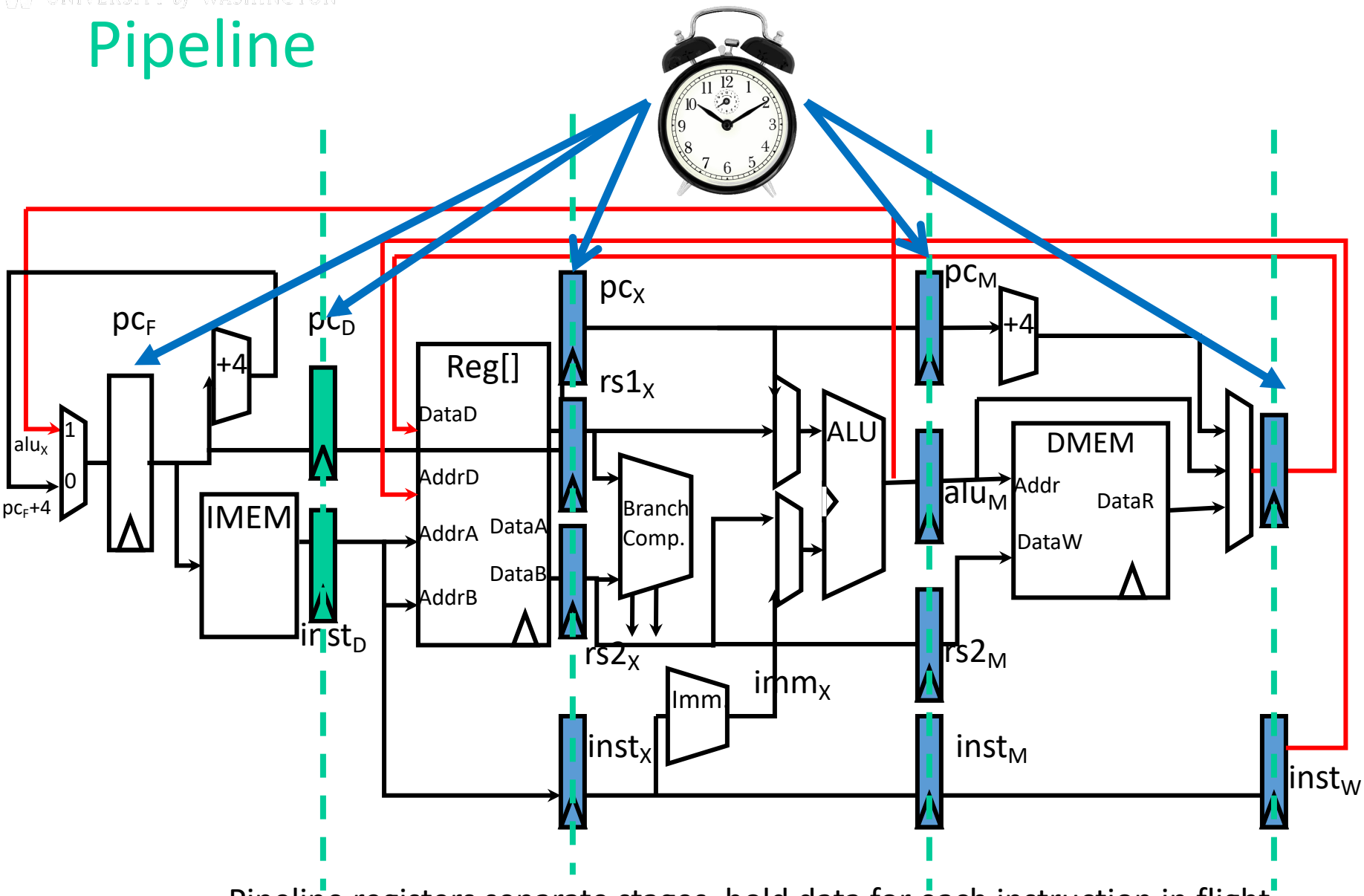
Agenda

- Quick Datapath Review
- Control Implementation
- Administrivia
- Performance Analysis
- Pipelined Execution
- Pipelined Datapath

Control Signals: ADD

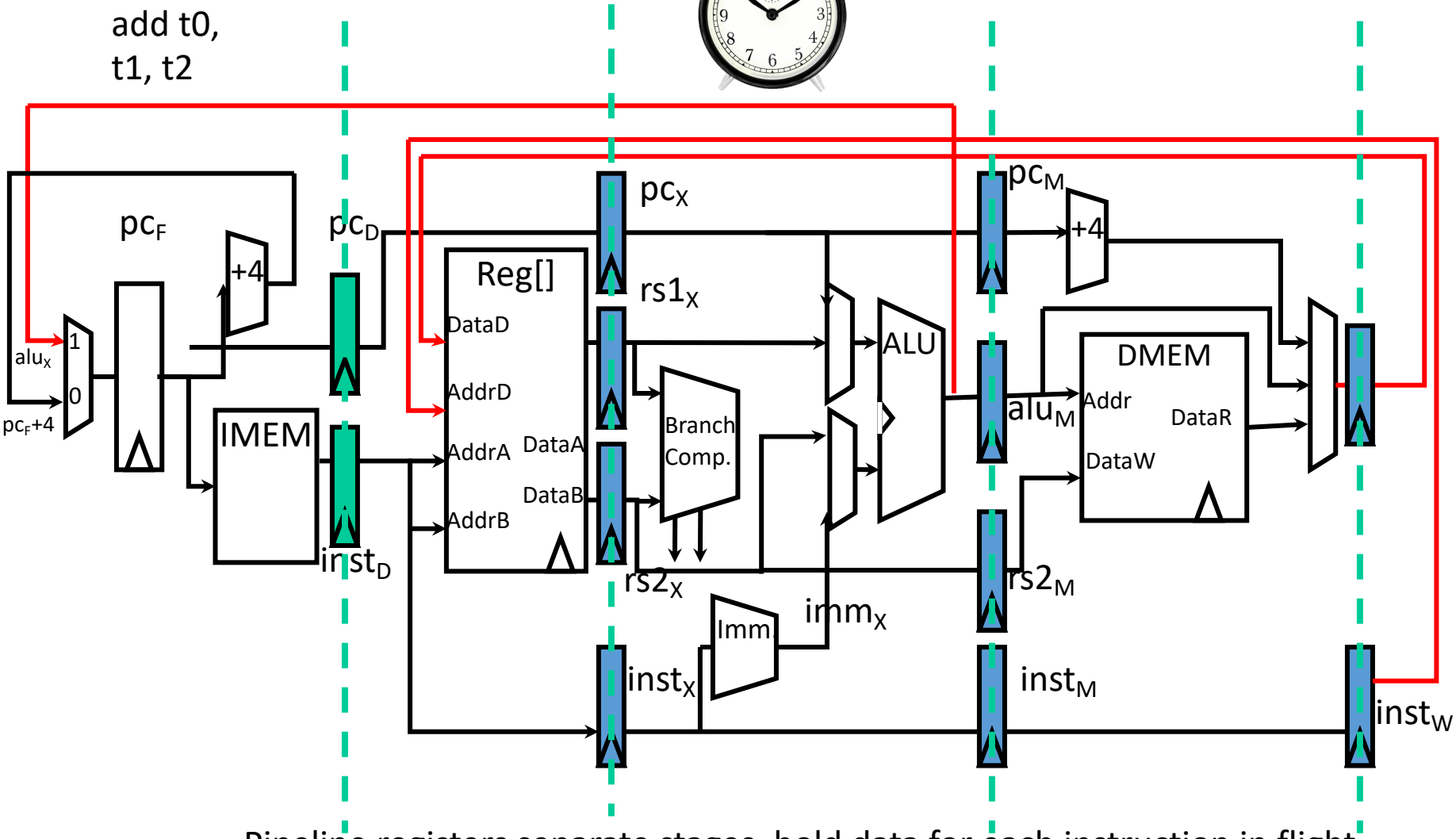


Pipeline



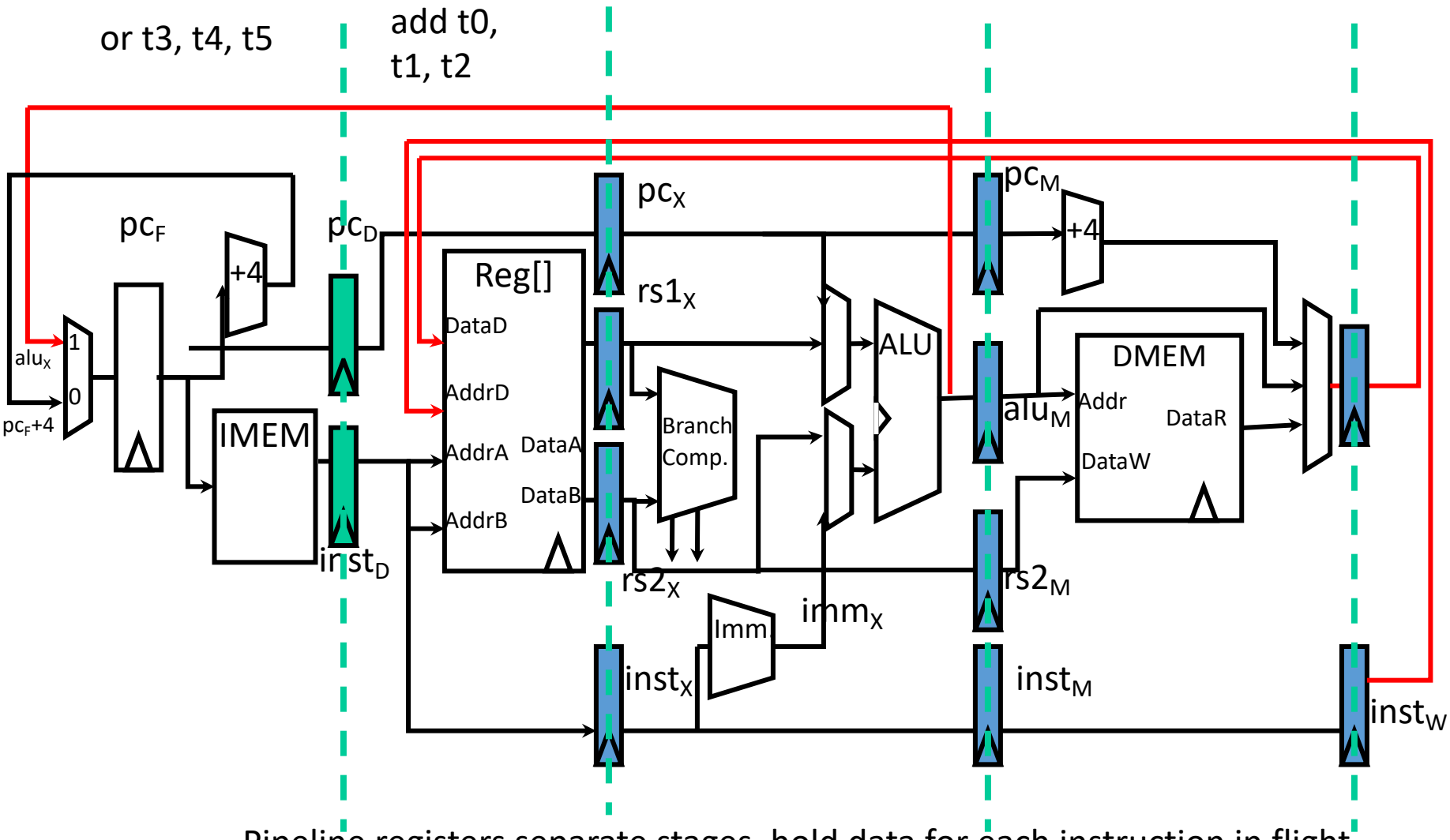
Pipeline registers separate stages, hold data for each instruction in flight

Each stage operates on different instruction



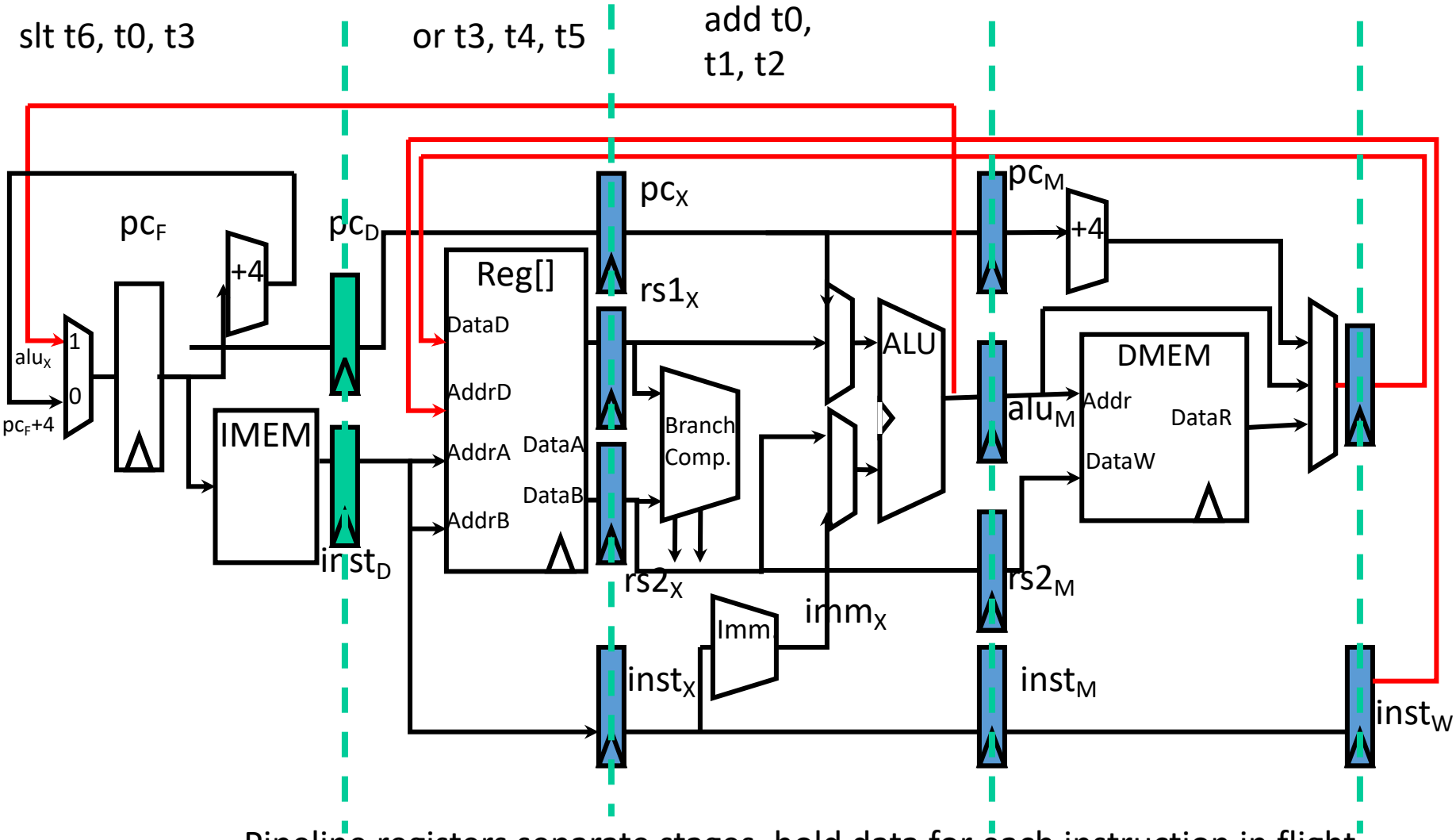
Pipeline registers separate stages, hold data for each instruction in flight

Each stage operates on different instruction



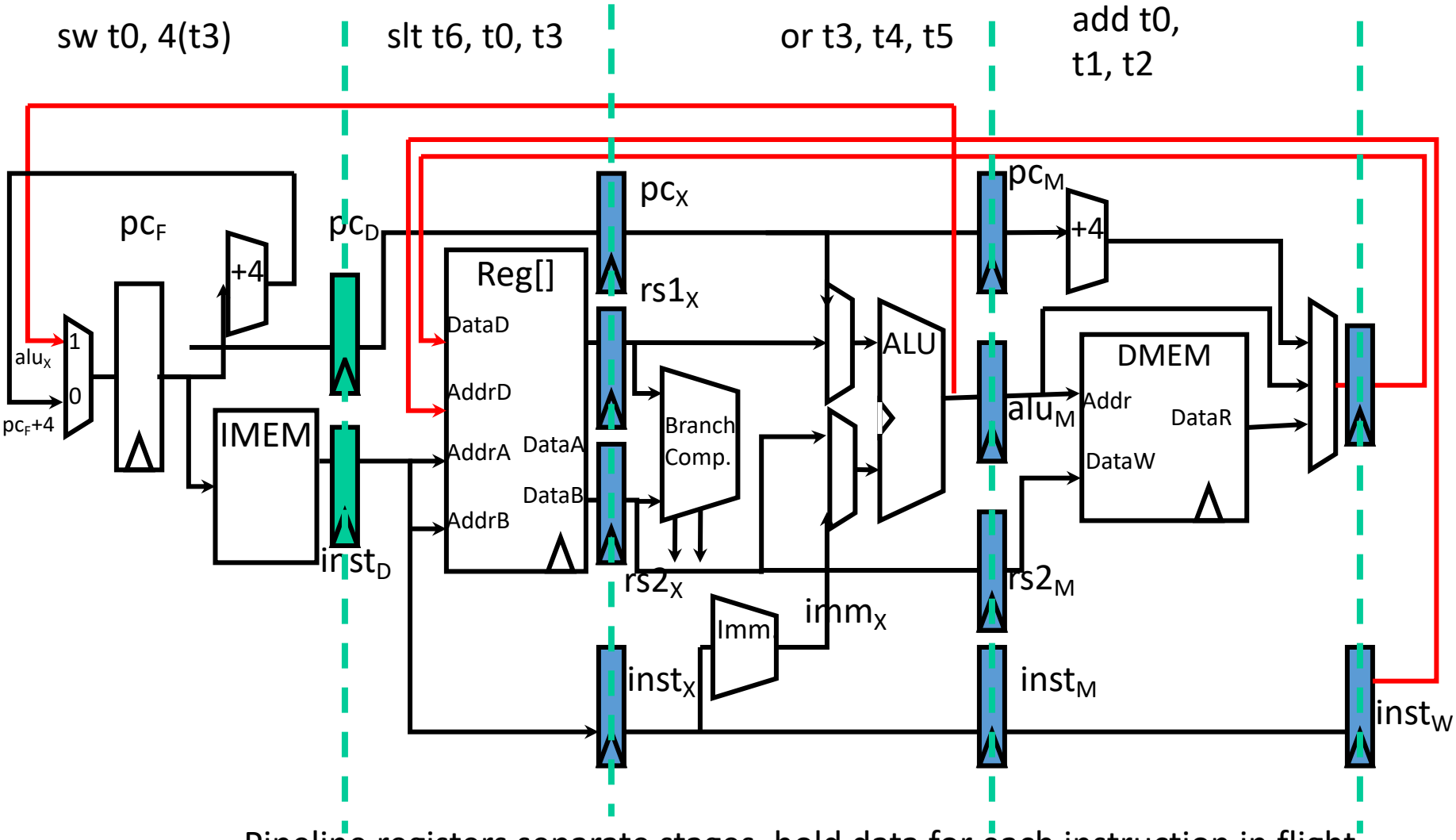
Pipeline registers separate stages, hold data for each instruction in flight

Each stage operates on different instruction



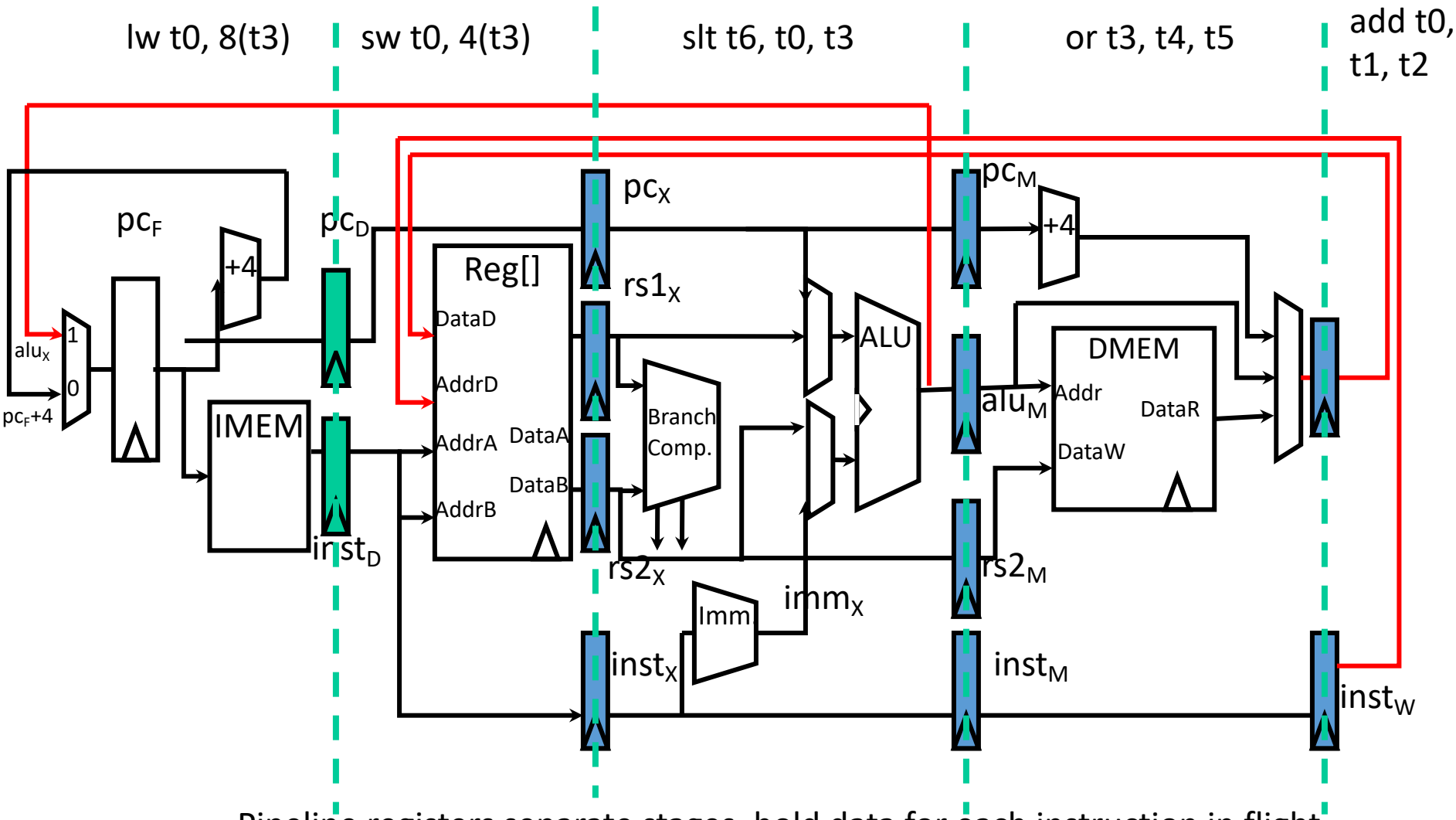
Pipeline registers separate stages, hold data for each instruction in flight

Each stage operates on different instruction



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

Each stage operates on different instruction



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

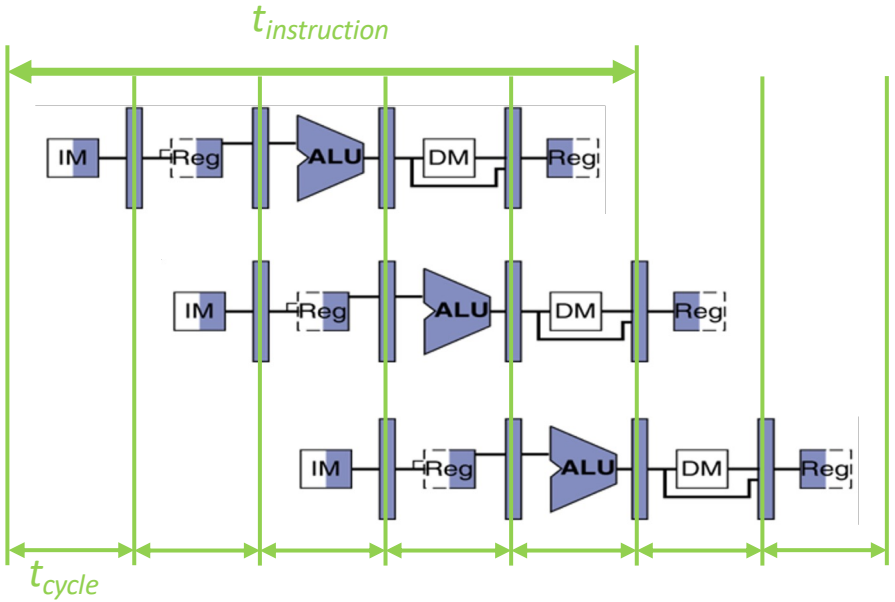
Pipelining with RISC-V

instruction sequence

add t0, t1, t2

or t3, t4, t5

sll t6, t0, t3



Longest pipeline stage

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

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 ?x slower.
2) The *throughput* will be ?x higher.

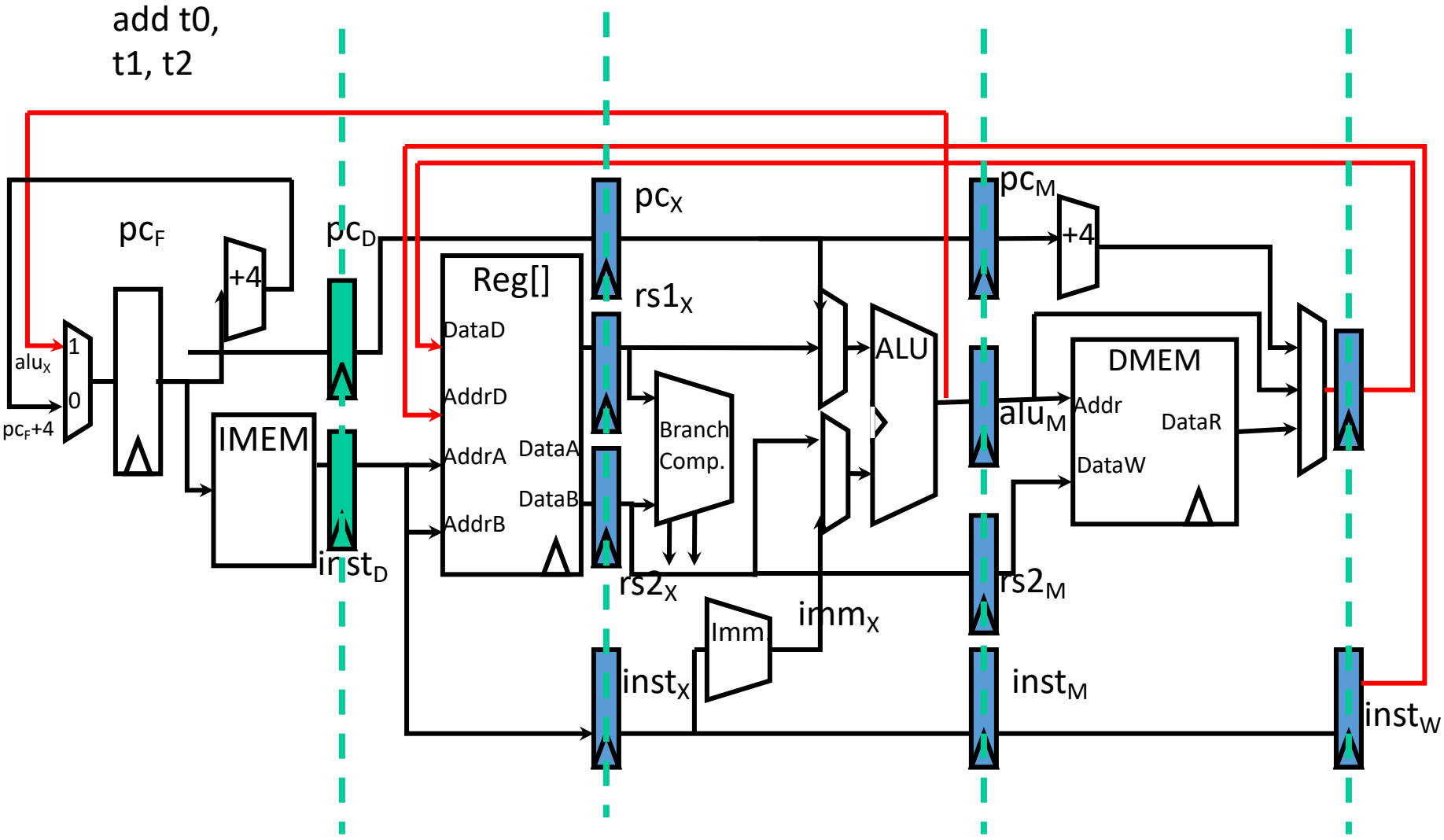
No mem access Throughput:

Old: one inst every (IF+ID+EX+WB) = 600 ps →

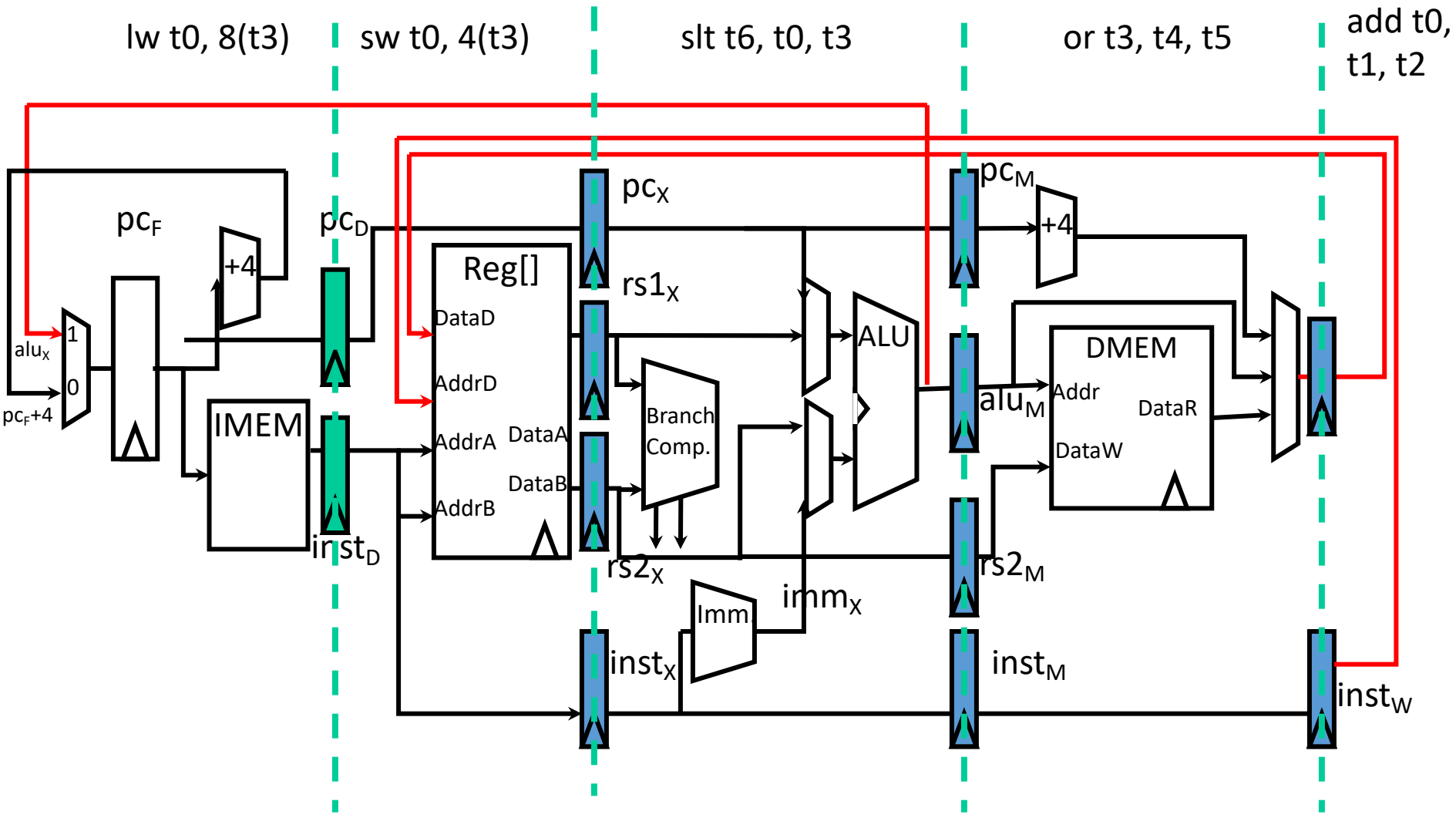
New: one inst every (4*max_stage)/4 = 200 ps

New/Old throughput = (1/200)/(1/600) = 3x higher

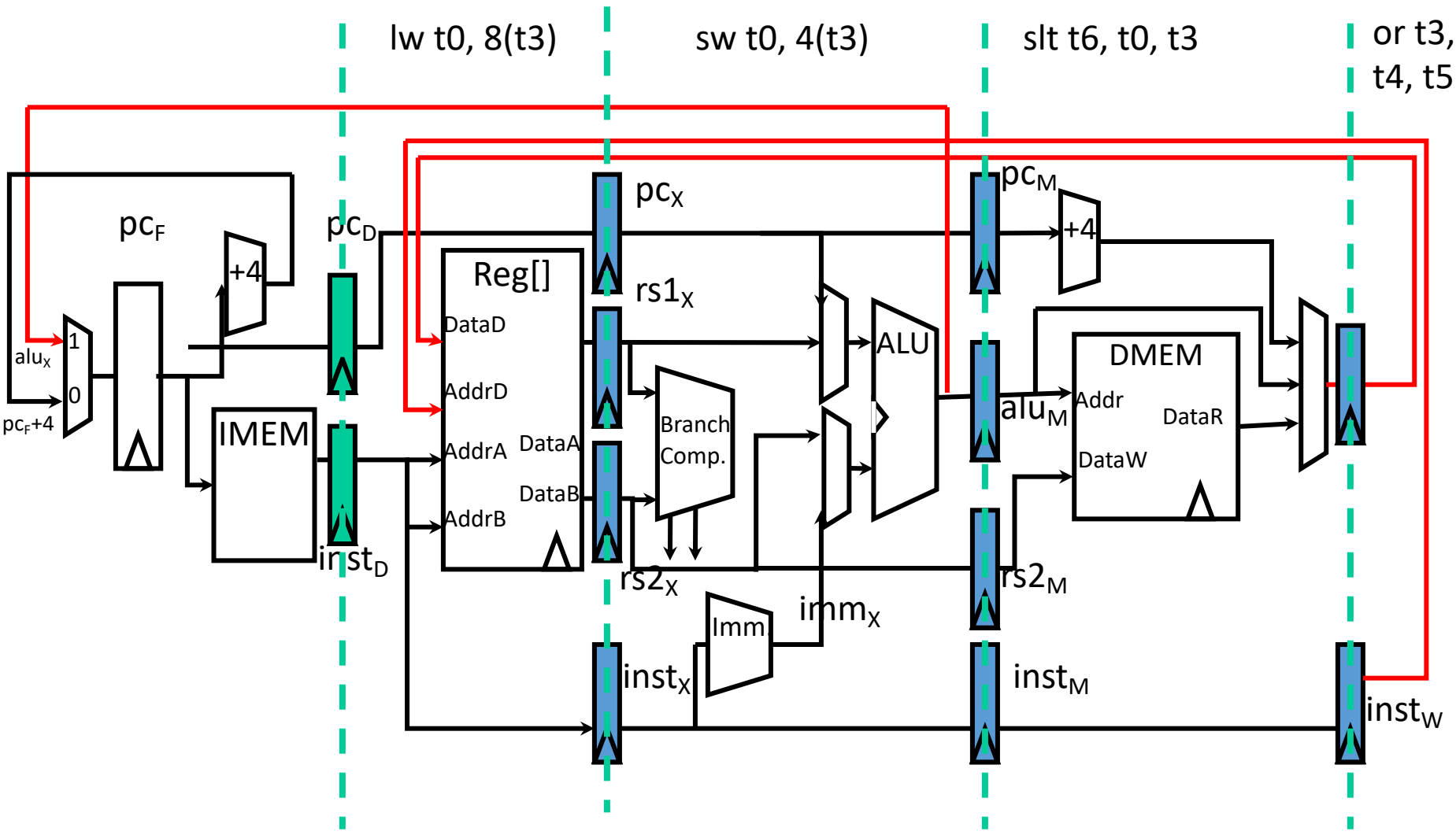
At time t = 0



At time t = 4

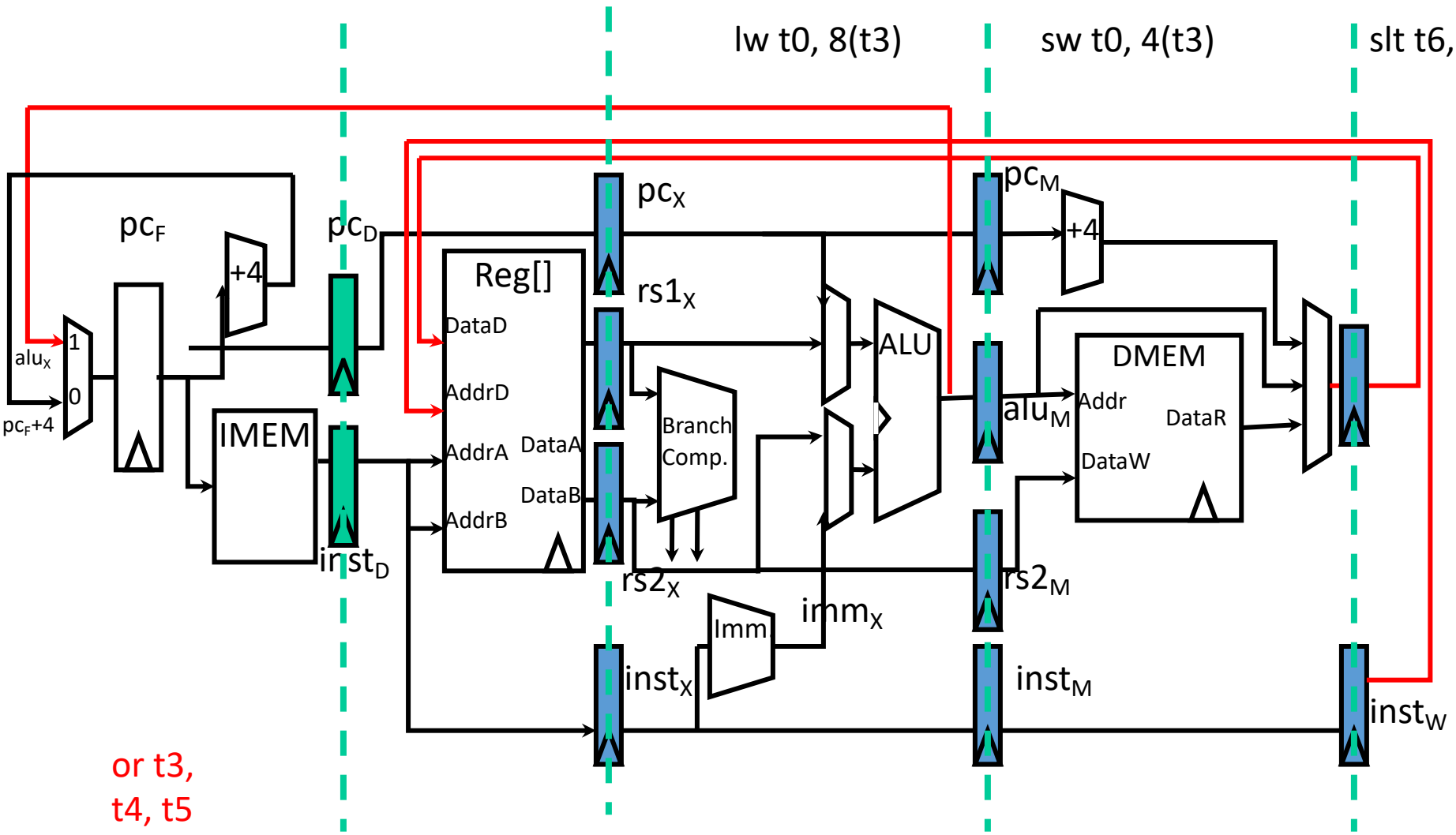


At time t = 5



add t0, t1, t2

At time t = 6



or t3,
t4, t5

add t0,
t1, t2

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 ?x slower.
2) The *throughput* will be 3x higher.

No mem access Latency (per inst):

Old latency: IF+ID+EX+WB = 600 ps

New latency = 4*max_stage = 800 ps (only as fast as slowest stage; faster stage slowed down since common clock).

New/Old = 800/600 = 1.33x slower

"Iron Law" of Processor Performance

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

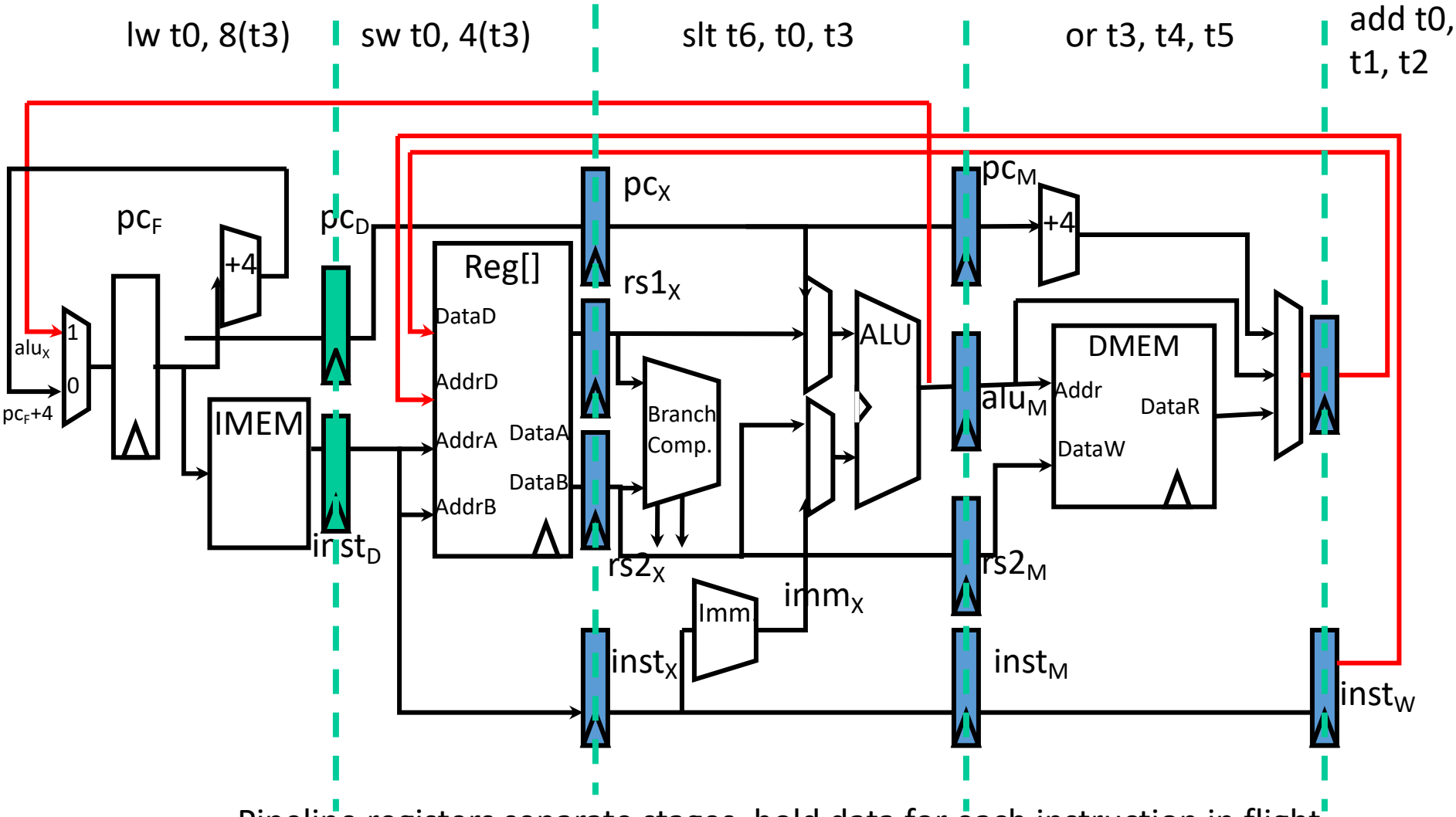
Speed Trade-off Example

- 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)

Each stage operates on different instruction

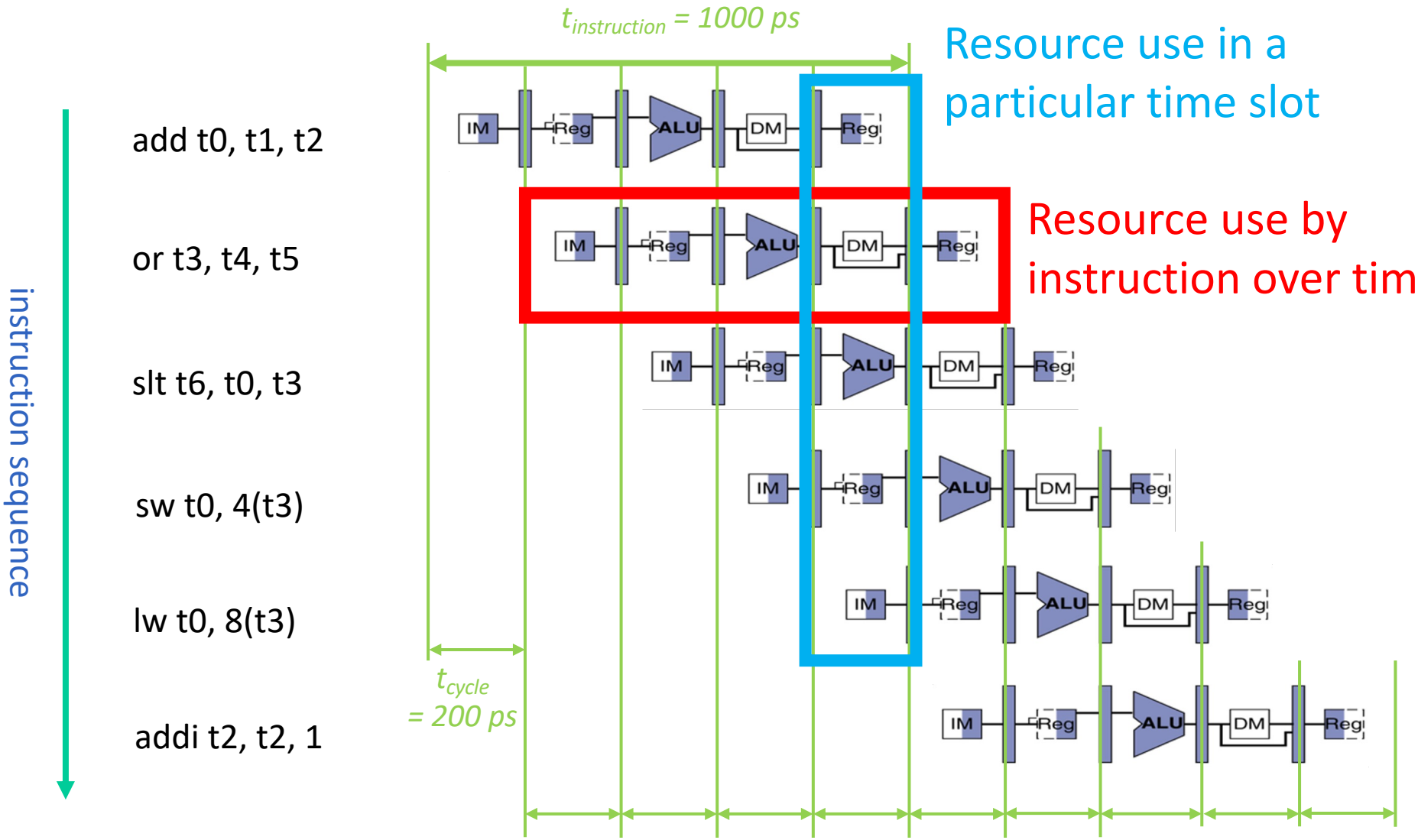


Pipeline registers separate stages, hold data for each instruction in flight

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" (e.g., using OpenMP or pthreads)

RISC-V Pipeline



RISC-V Pipeline Example

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

Agenda

Hazards Ahead!

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

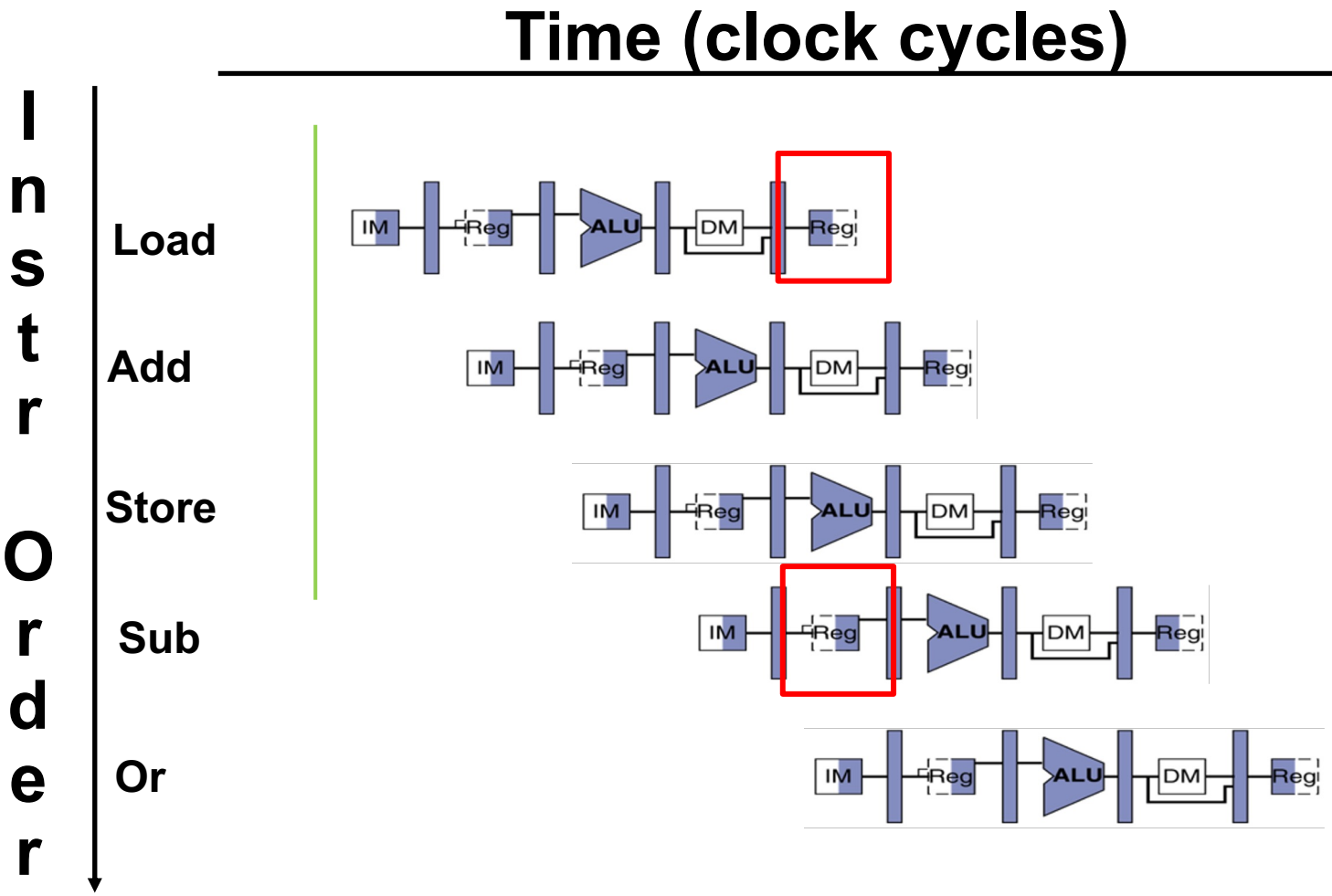
Pipeline Hazards

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

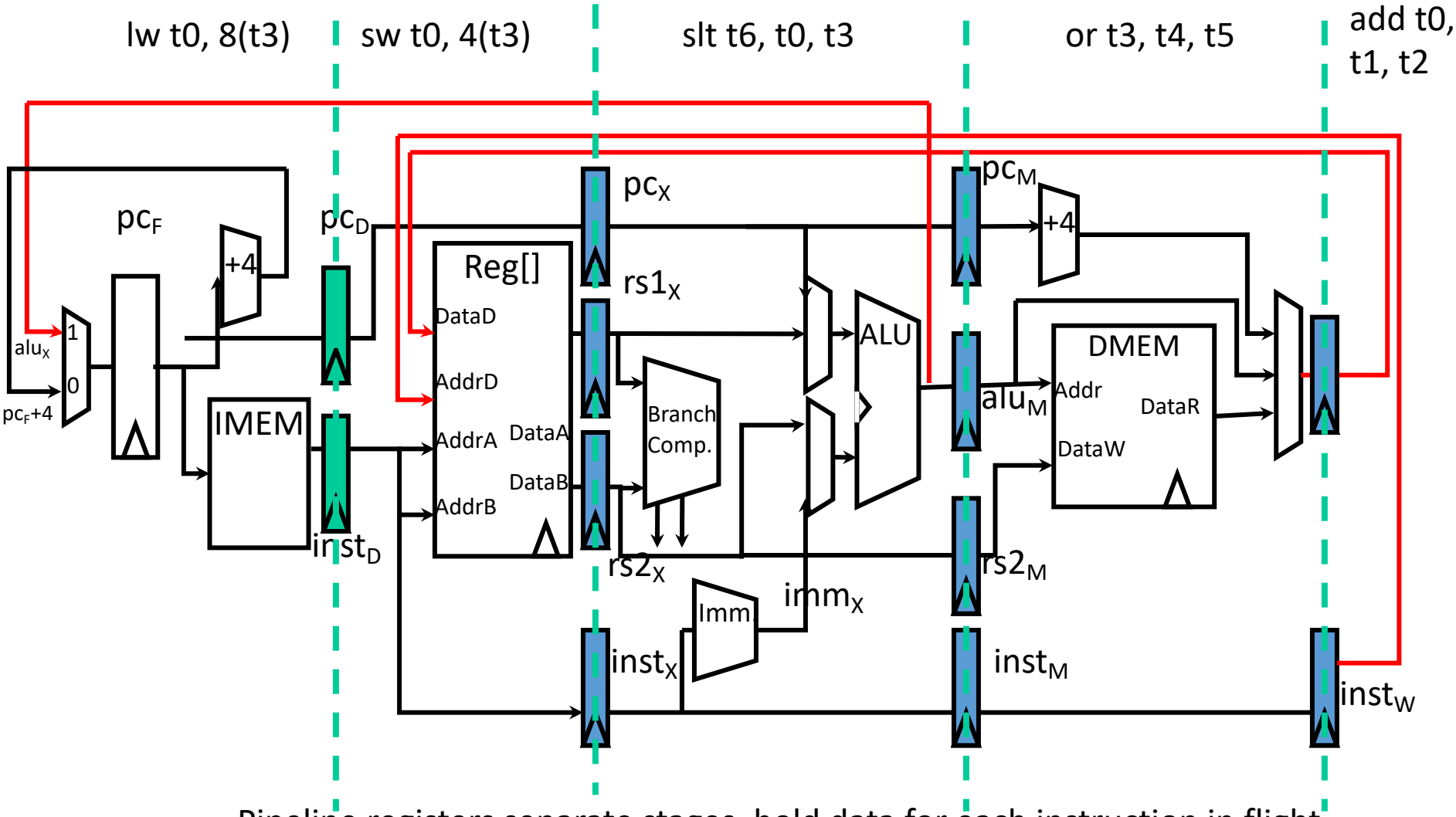
1) *Structural hazard*
   - A required resource is busy (e.g. needed in multiple stages)
2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data write
3) *Control hazard*
   - Flow of execution depends on previous instruction

Structural Hazard: Regfile!

- RegFile: Used in ID and WB!



Each stage operates on different instruction



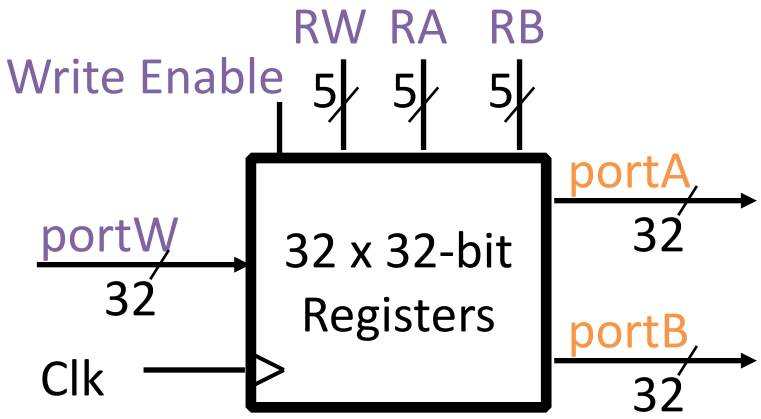
Pipeline registers separate stages, hold data for each instruction in flight

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	EX	MM	WB
0x0F	Instuction				IF	-	-	-	-	ID		

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; good for single-stage
  2) Double Pumping: Split RegFile access in two. Prepare to write during 1st half, write on *rising* edge, read during 2nd half of each clock cycle (*falling* edge)
     - Will save us a cycle later...
     - Possible because RegFile access is fast (takes less than half the time of ALU stage)
- **Conclusion:** It is OK to read and Write to registers during same clock cycle

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	Instruction		IF	ID	EX	MM	WB					
0x08	Instruction			IF	ID	EX	MM	WB				
0x0C	**addi a3, a0, 5**				**IF**	**ID**	**EX**	**MM**	**WB**			
0x0F	Instuction					IF	ID	EX	MM	WB		

Structural Hazard: Memory Access

instruction sequence

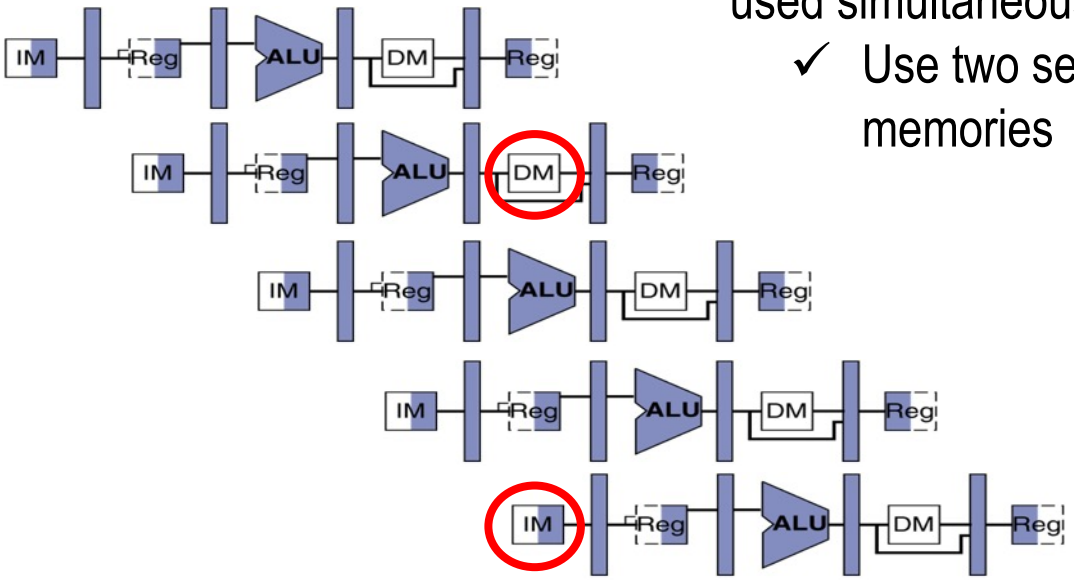
add t0, t1, t2

or t3, t4, t5

slt t6, t0, t3

sw t0, 4(t3)

lw t0, 8(t3)



- 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)

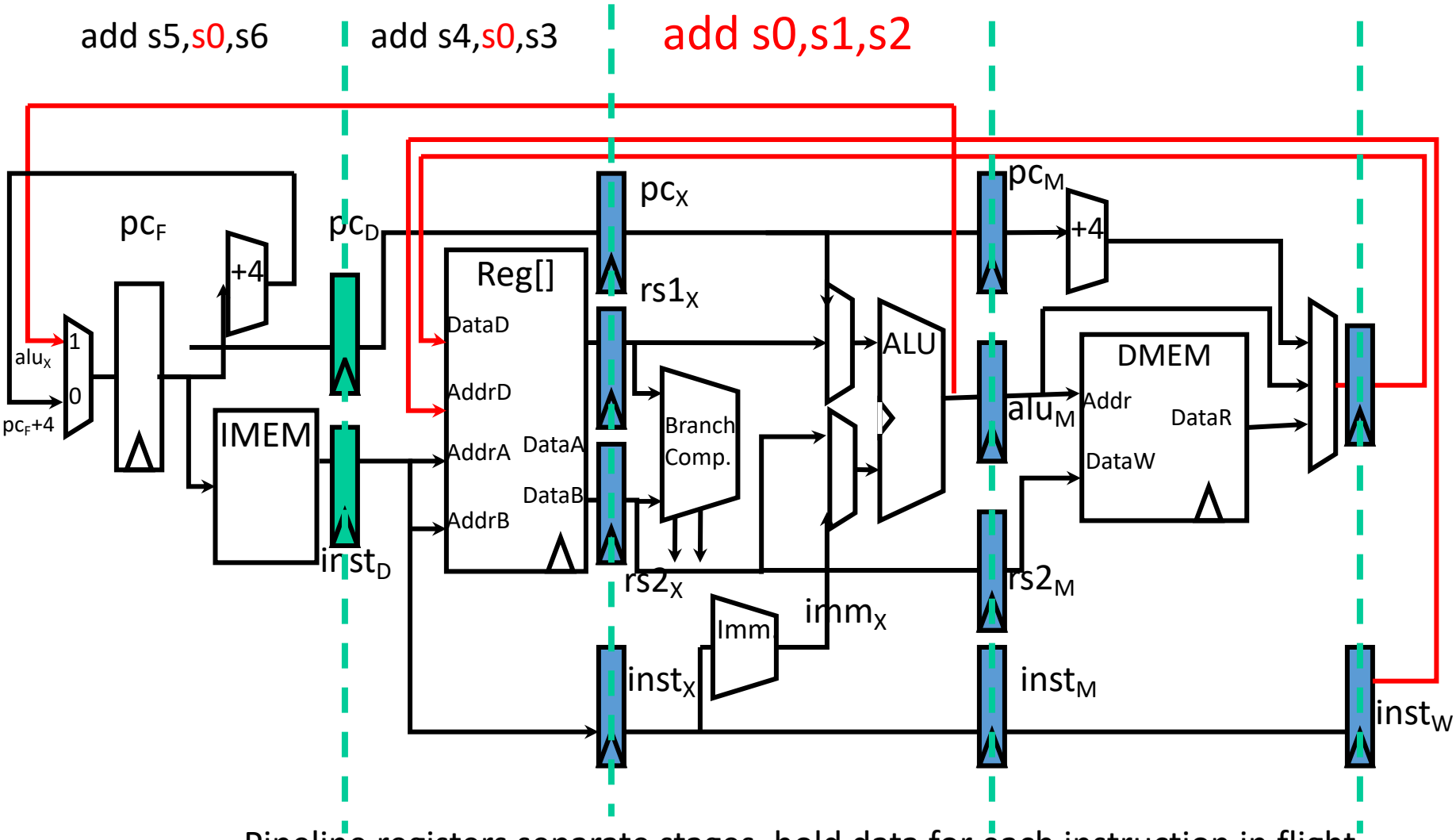
- Consider the following sequence of instructions:

```
add   s0, s1, s2
sub   s4, s0, s3
and   s5, s0, s6
or    s7, s0, s8
xor   s9, s0, s10
```

Stored during WB

Read during ID

Each stage operates on different instruction

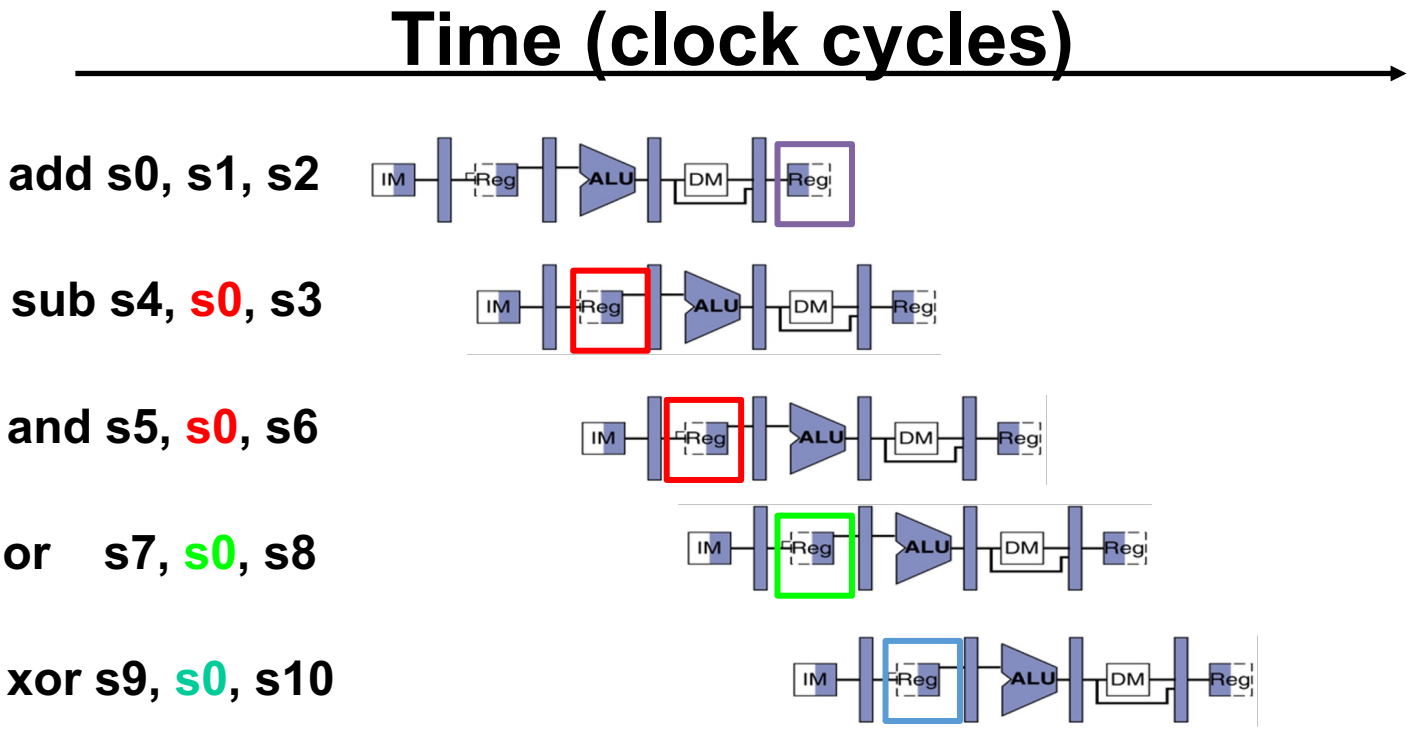


Pipeline registers separate stages, hold data for each instruction in flight

2. Data Hazards (2/2)

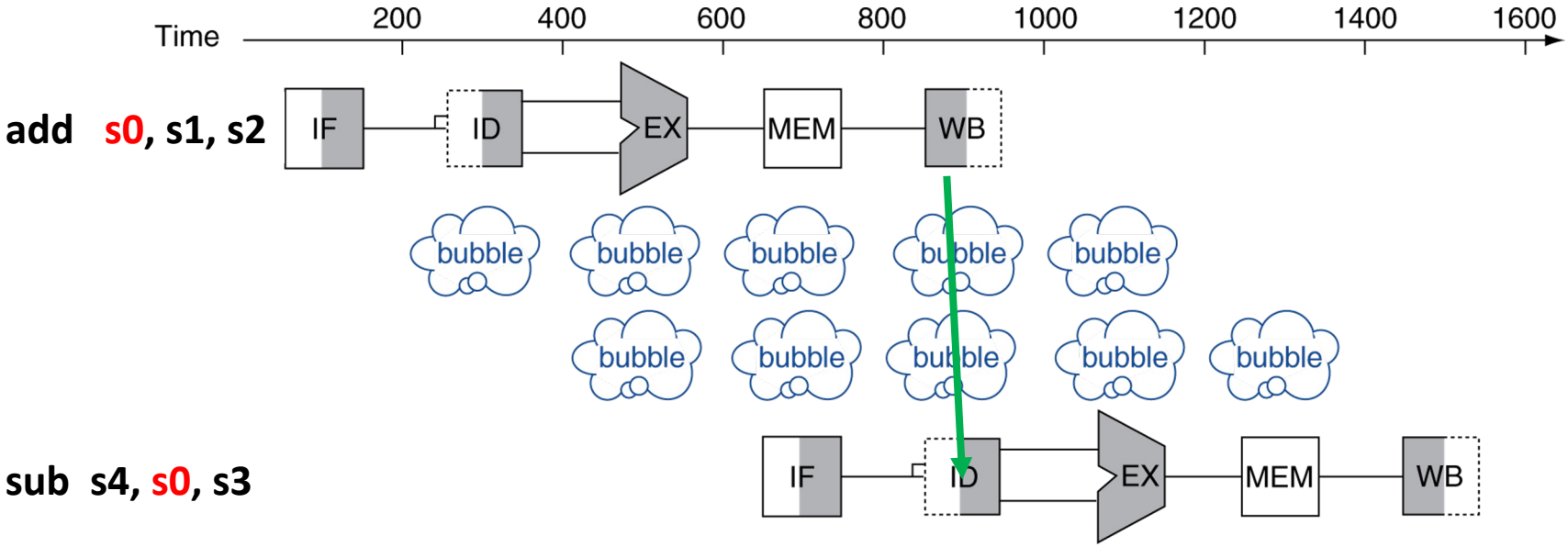
Identifying data hazards:

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

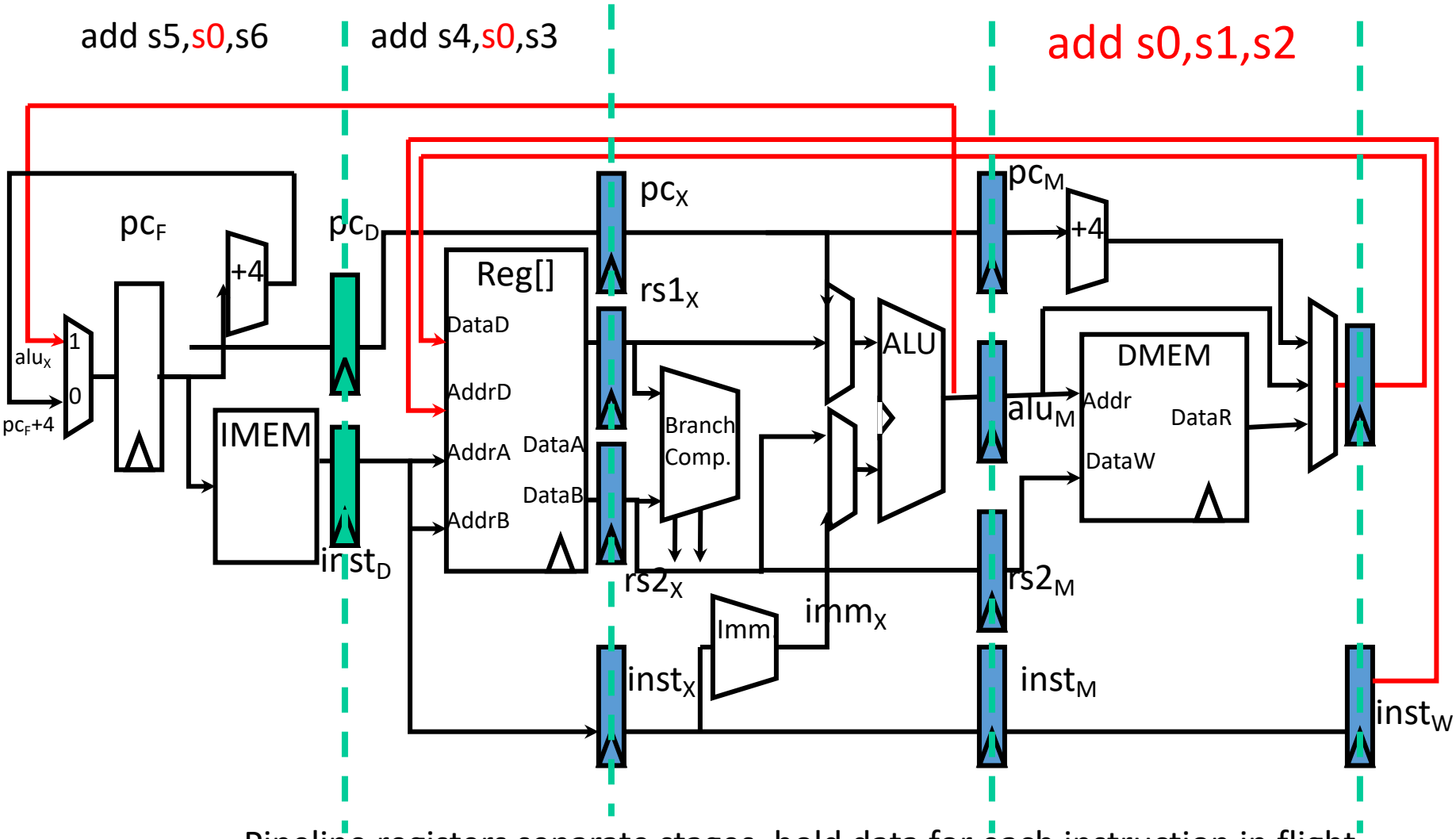


Solution 1: Stalling

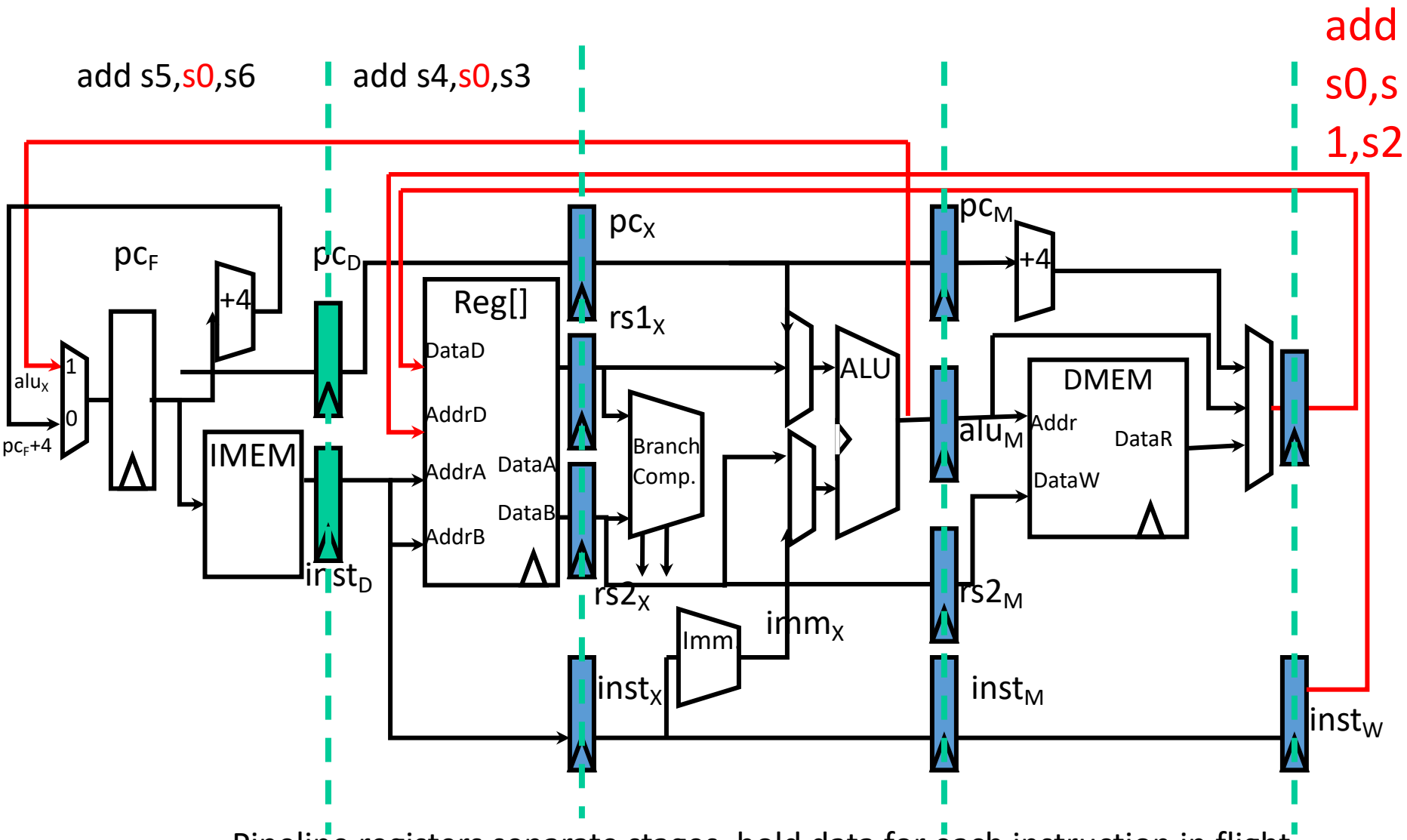
- Problem: Instruction depends on result from previous instruction
  - add s0, s1, s2
    sub s4, s0, s3



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



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

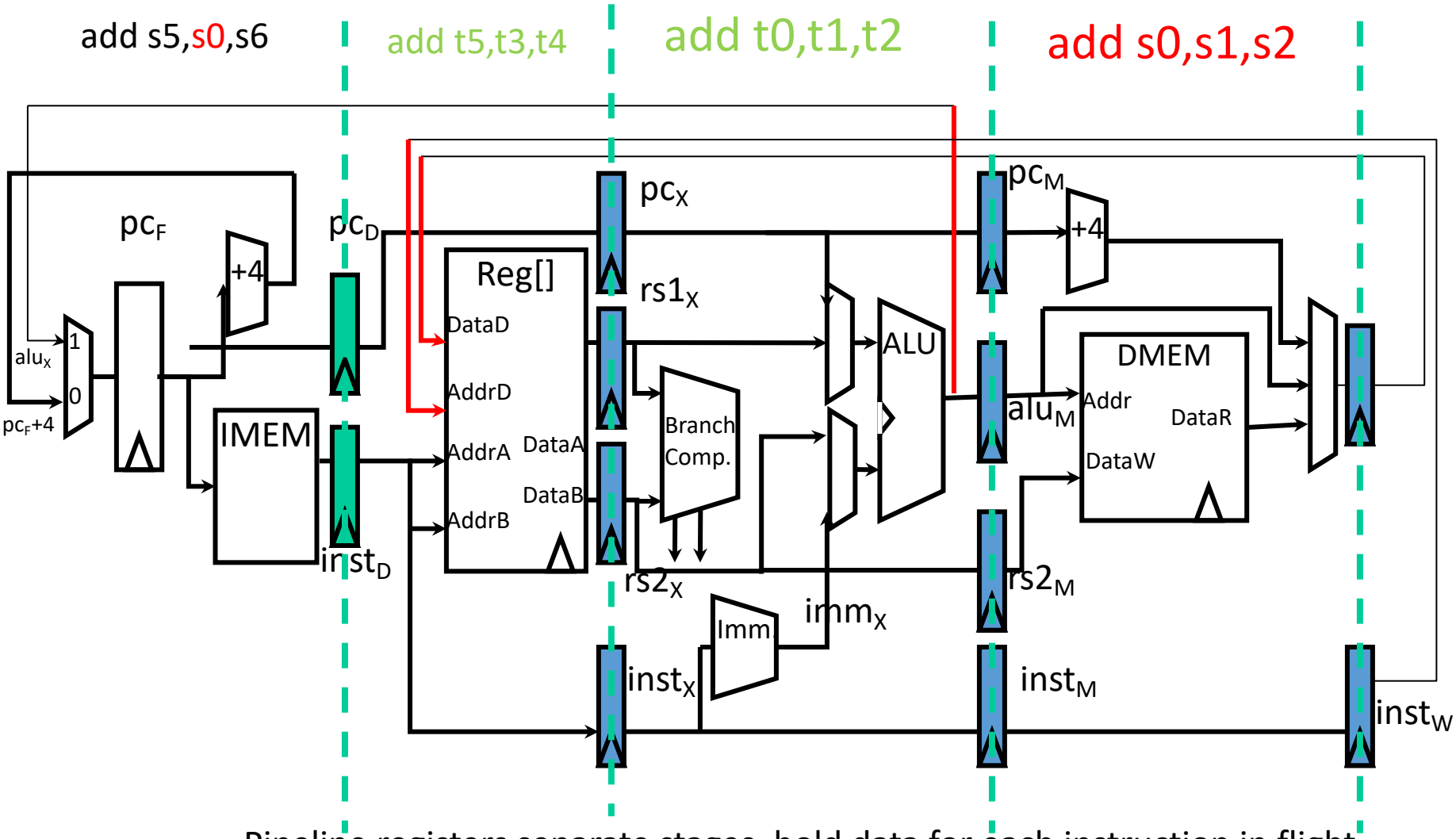


Pipeline registers separate stages, hold data for each instruction in flight

Data Hazard

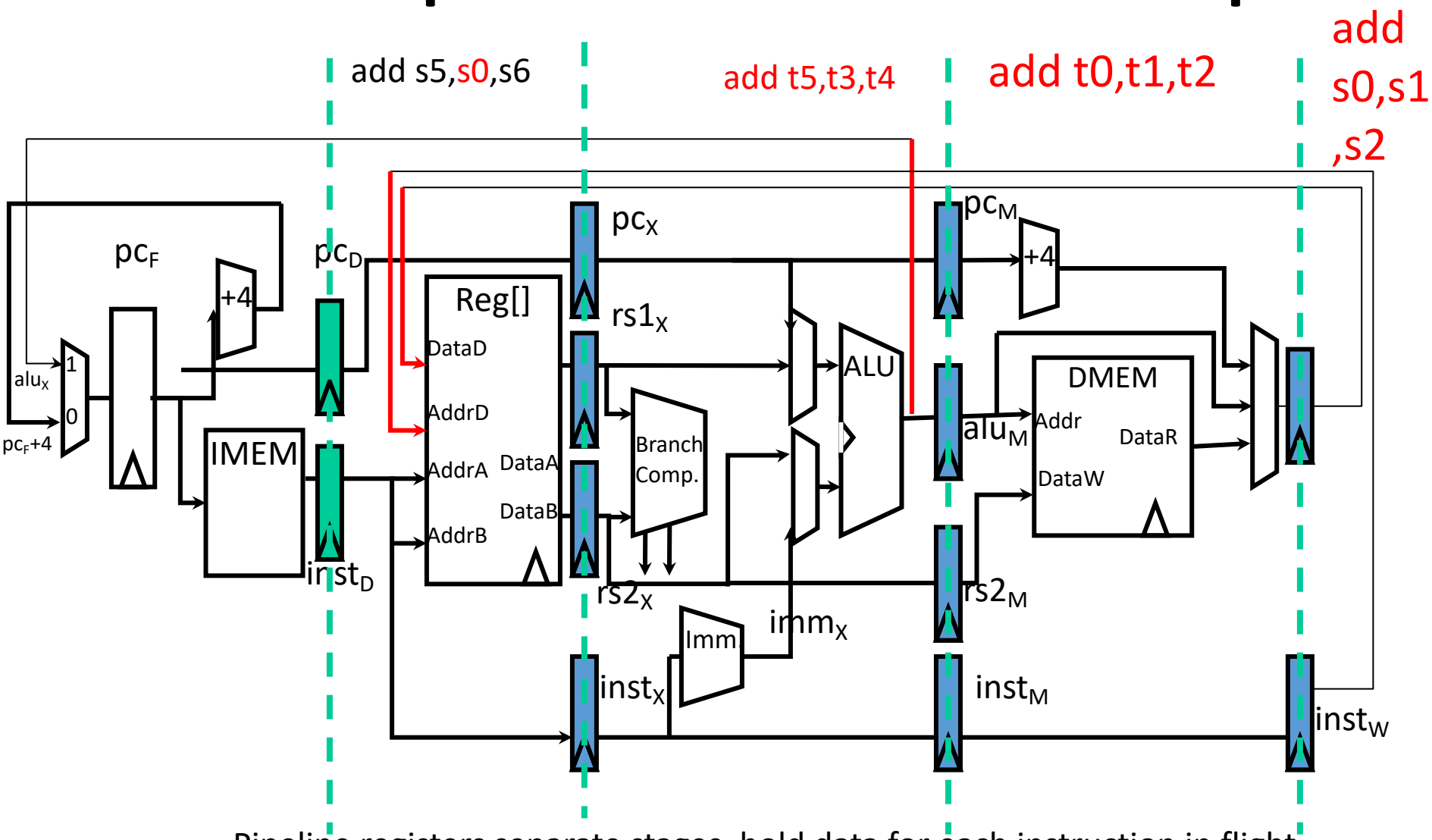
Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	add s0, s1, s2	IF	ID	EX	MM	WB						
0x04	sub s4, s0, s3		IF	ID	ID	ID	EX	MM	WB			
0x08	and s5, s0, s6			IF	IF	IF	ID	EX	MM	WB		
0x0C	or s7, s0, s8						IF	ID	EX	MM	WB	

Add independent instructions or nops



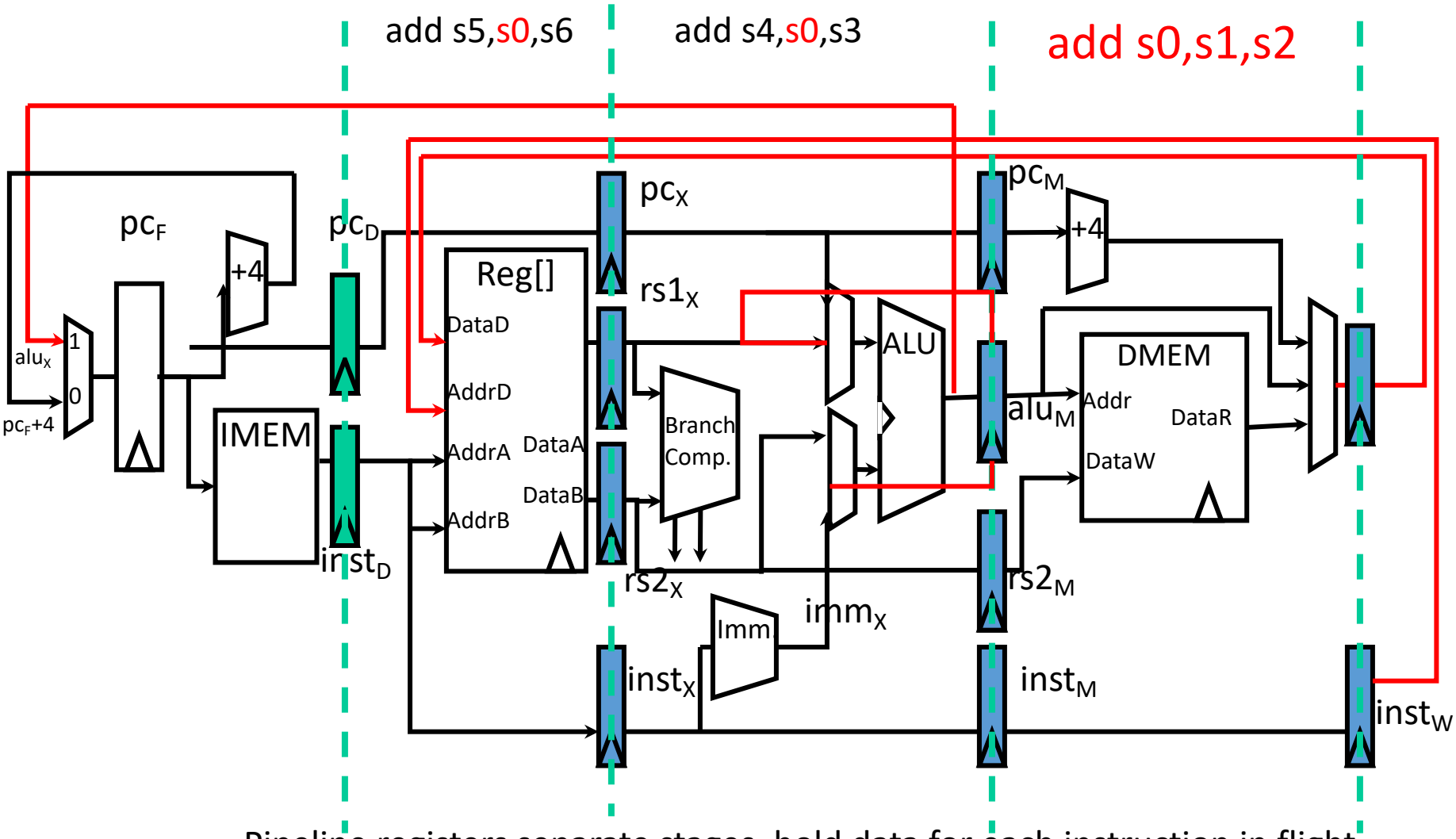
Pipeline registers separate stages, hold data for each instruction in flight

Add independent instructions or nops



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

Forwarding



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

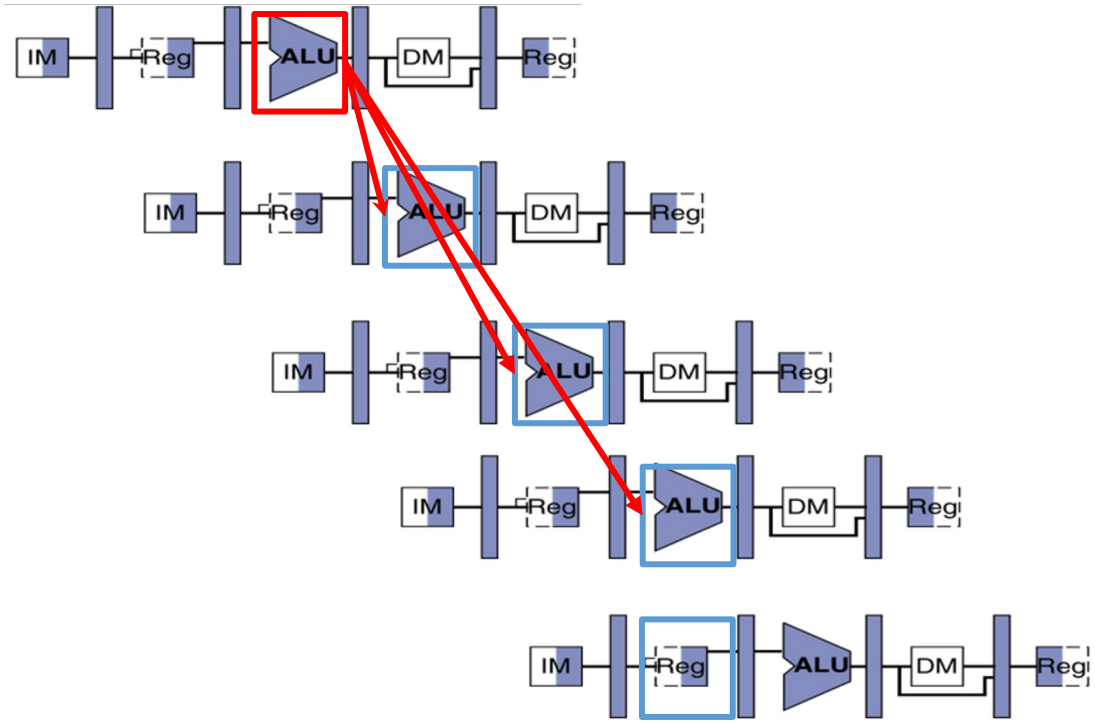
Data Hazard with Forwarding

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 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
```



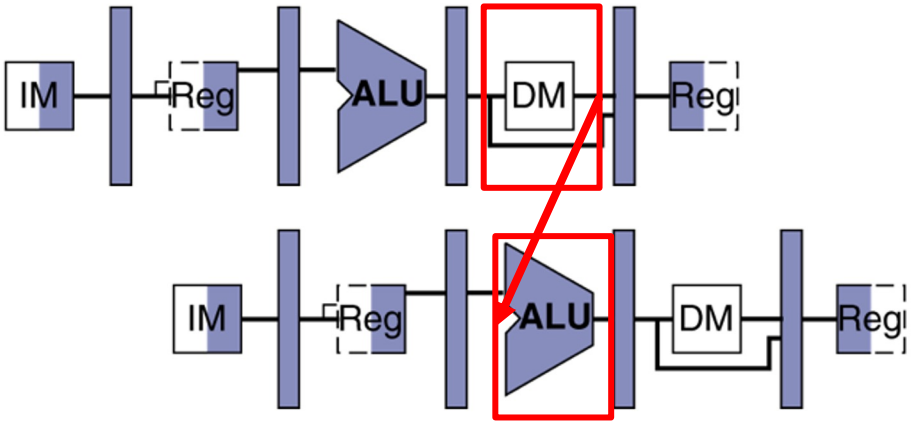
Forwarding: get operand from pipeline stage, rather than register file

Data Hazard: Loads (1/2)

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

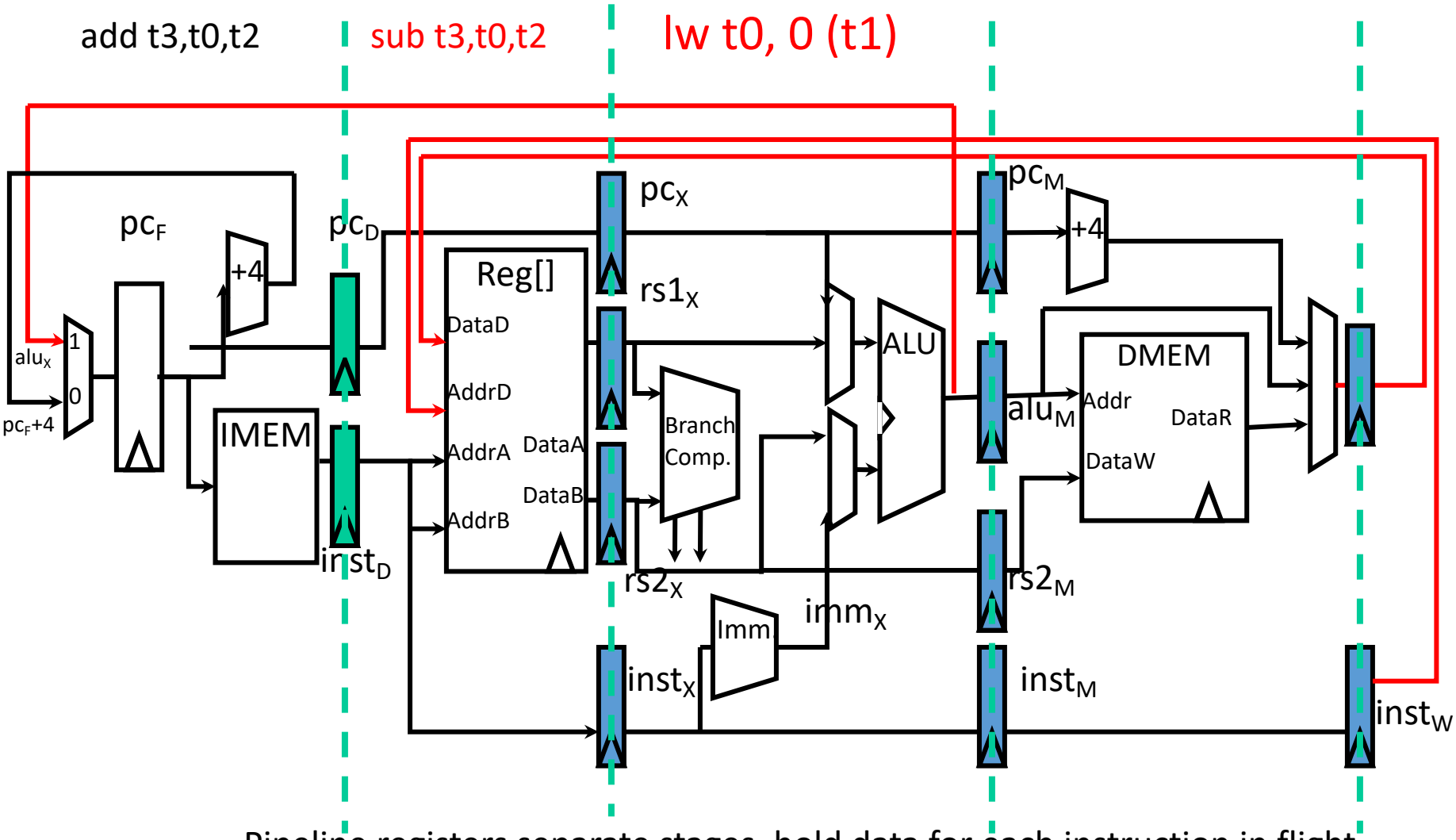
lw t0, 0(t1)

sub t3, t0, t2



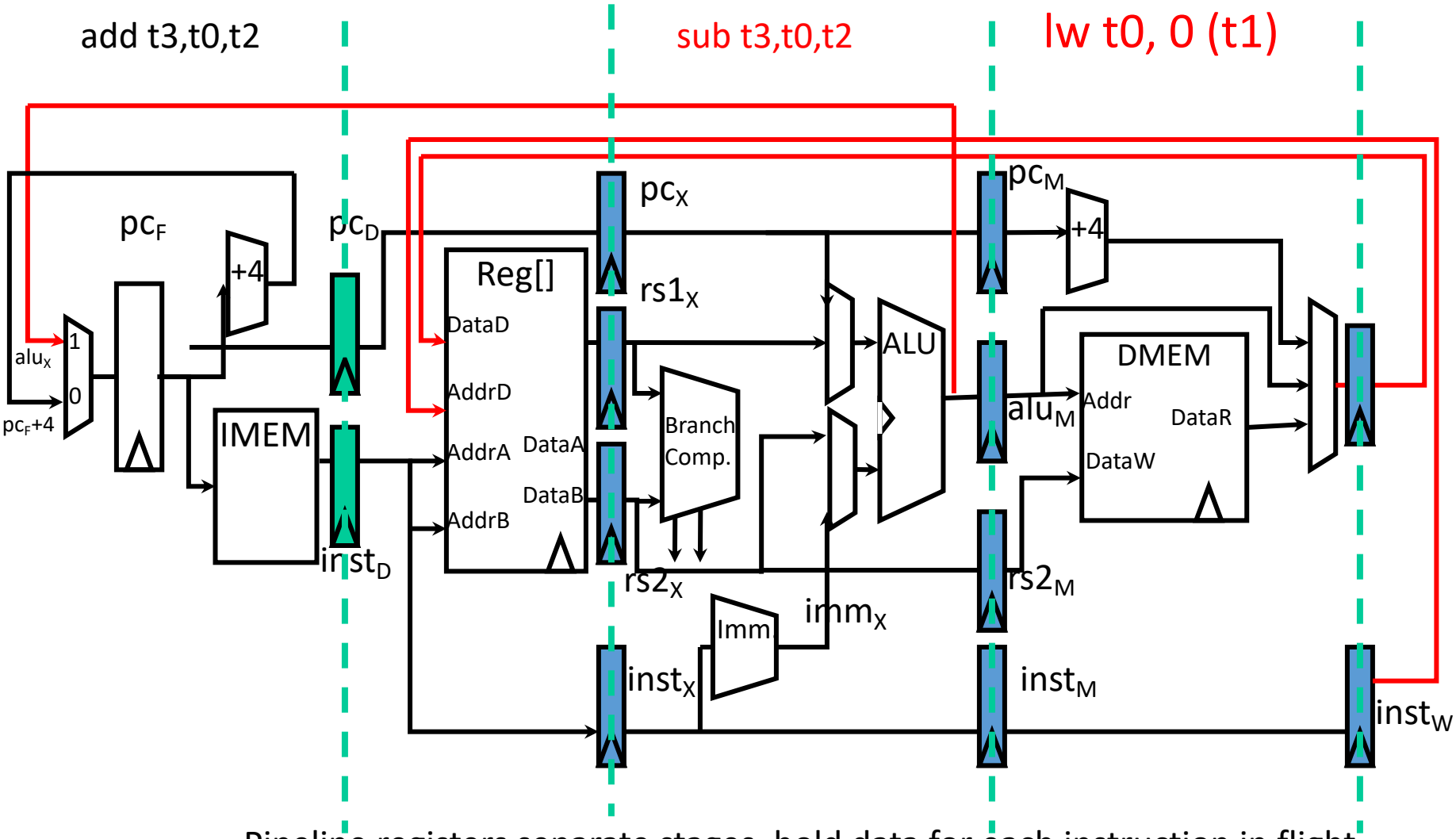
- Can't solve all cases with forwarding
  - Must *stall* instruction dependent on load (sub), then forward after the load is done (more hardware)

Each stage operates on different instruction



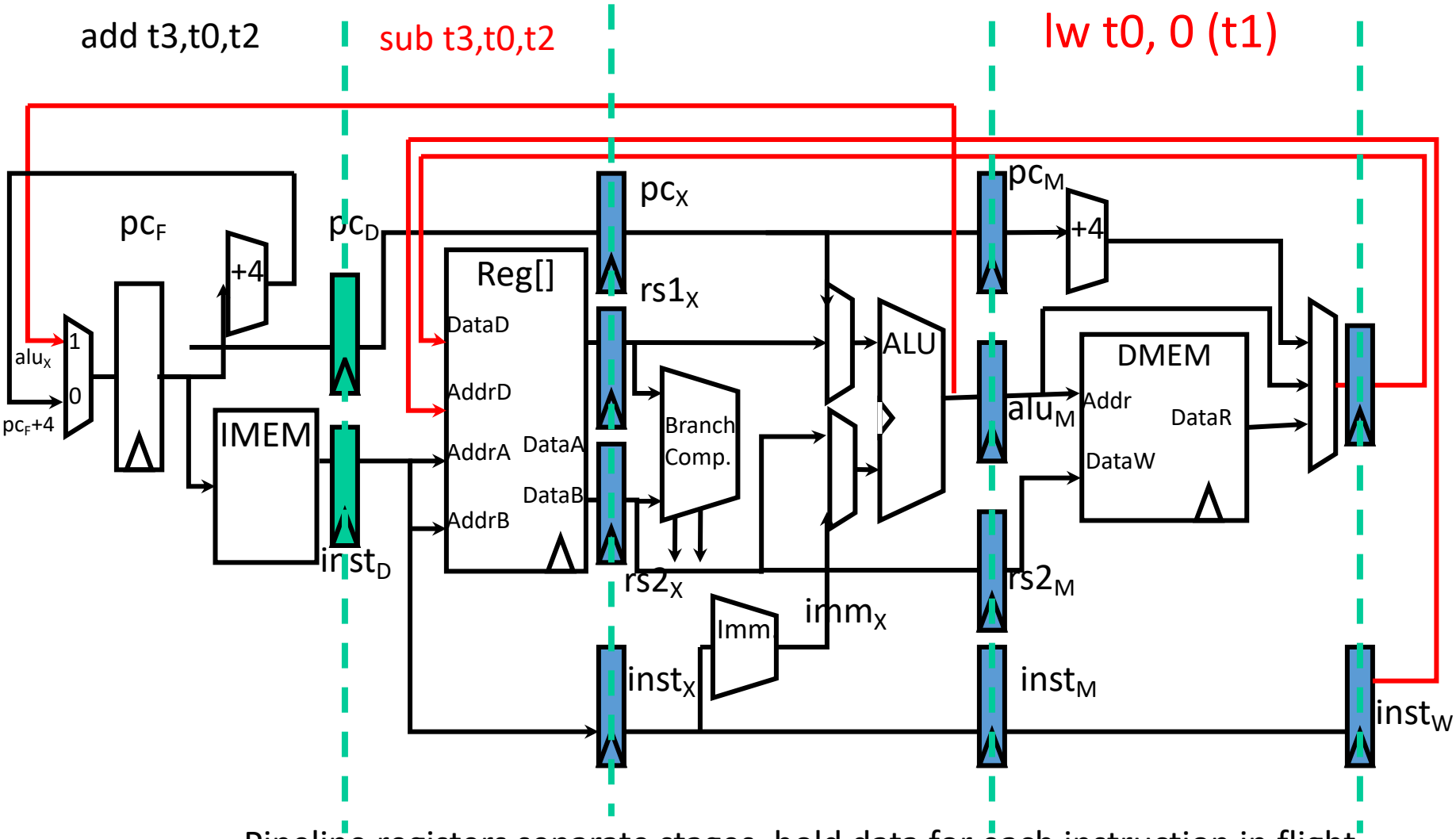
Pipeline registers separate stages, hold data for each instruction in flight

Incorrect Exe: LW not available until later



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

Correct Exe: Stall



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

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!

Load Data Hazards with Forwarding

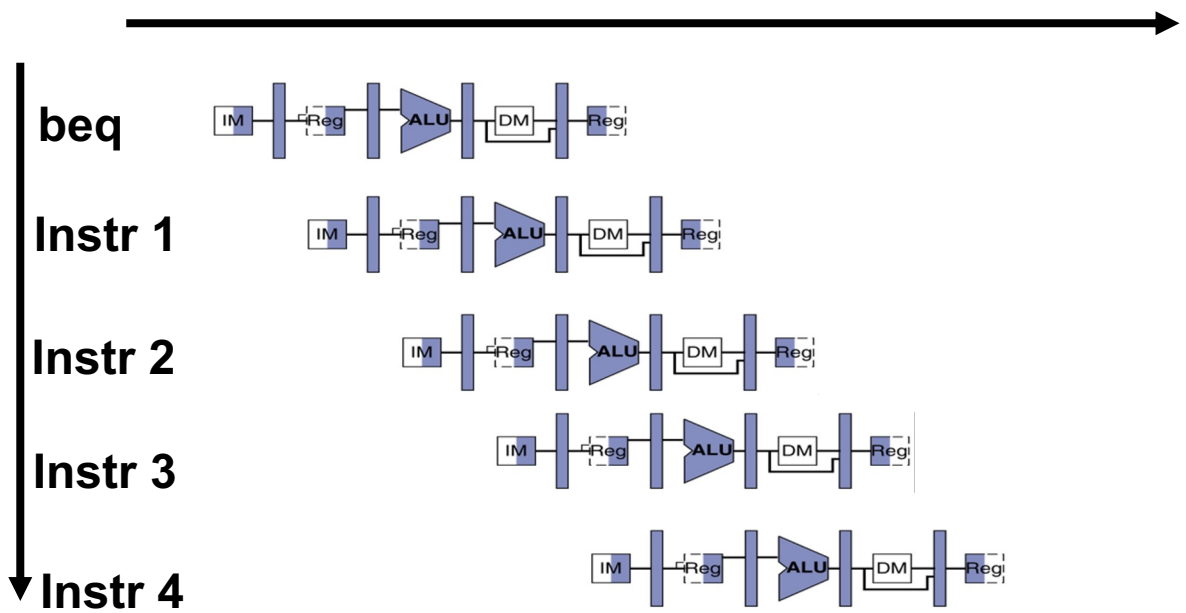
Addr	Inst | Cycle	0	1	2	3	4	5	6	7	8	9	10
0x00	lw s0, 0 (t1)	IF	ID	EX	MM	WB						
0x04	sub s4, s0, s3		IF	ID		EX	MM	WB				
0x08	and s5, s0, s6			IF	IF	ID	EX	MM	WB			
0x0C	or s7, s0, s8					IF	ID	EX	MM	WB		

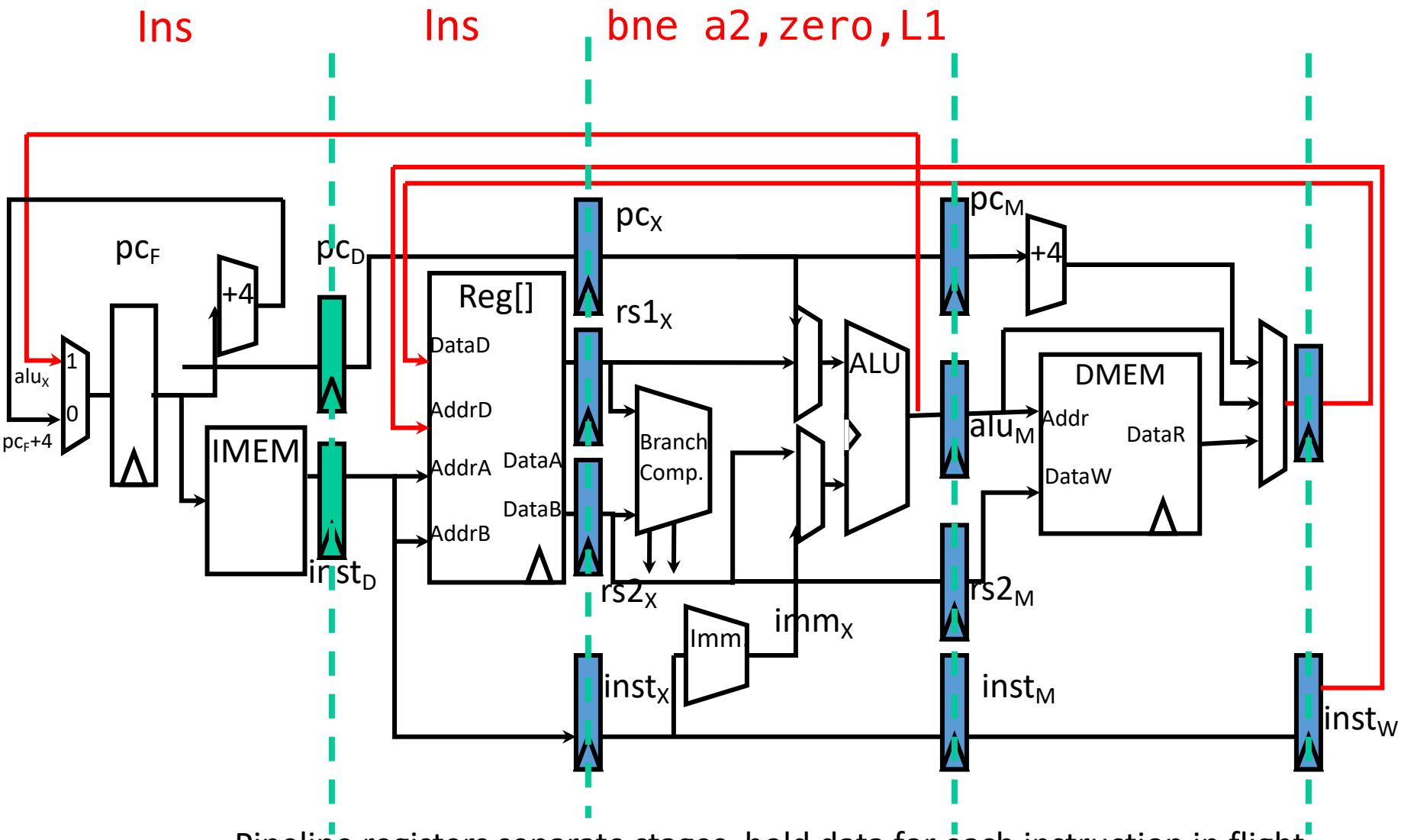
3. Control Hazards

- 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

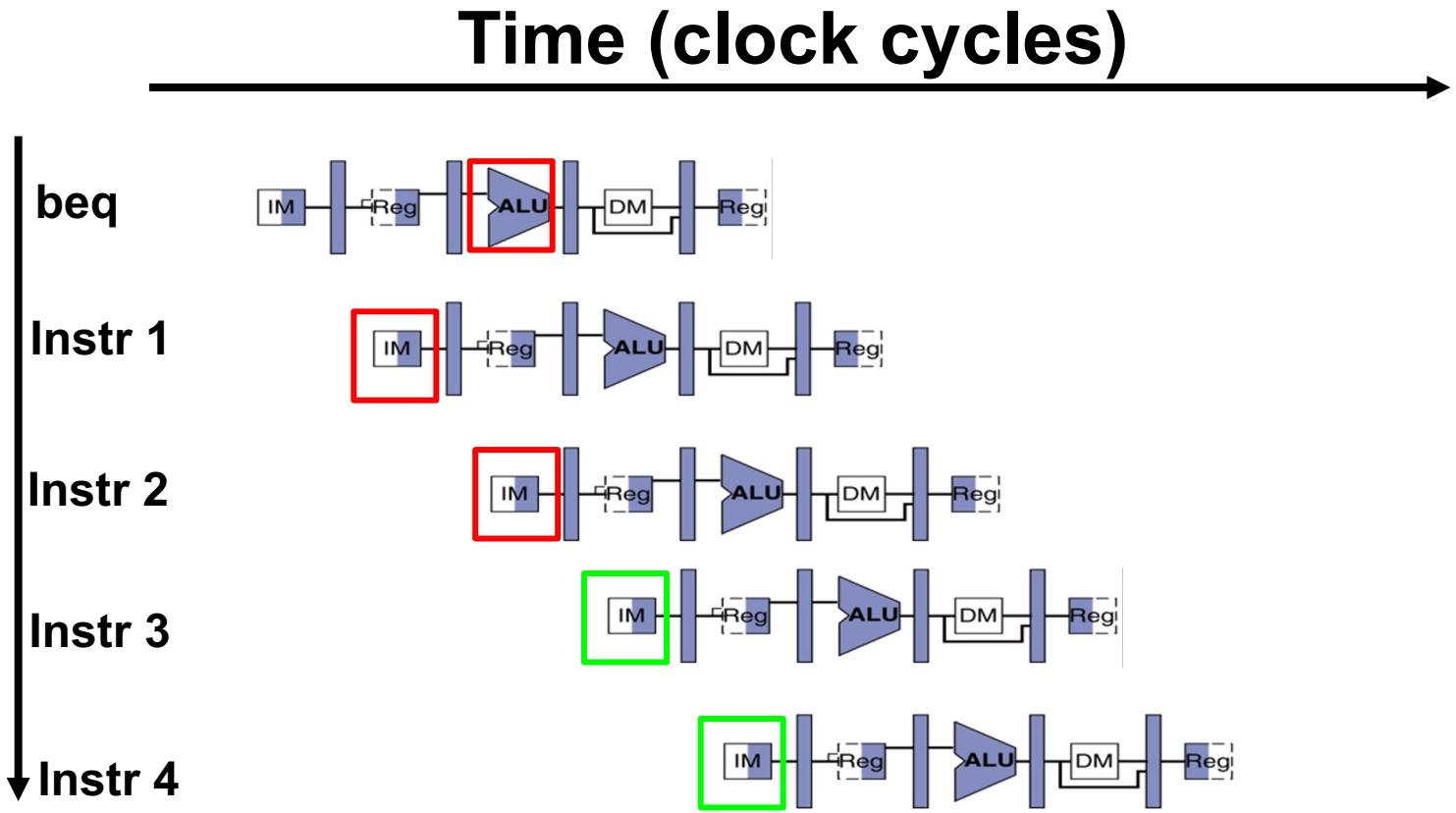




Pipeline registers separate stages, hold data for each instruction in flight

Branch Stall

- How many bubbles required for branch?



Taken branch (Stall 2 cycles)

Address	Ins-Cycle	0	1	2	3	4	5	6		
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									
0x0c	jal zero, 0x00000014									
0x10	addi a3, zero, 0					IF	ID	EX		
0x14	ecall						IF	ID		

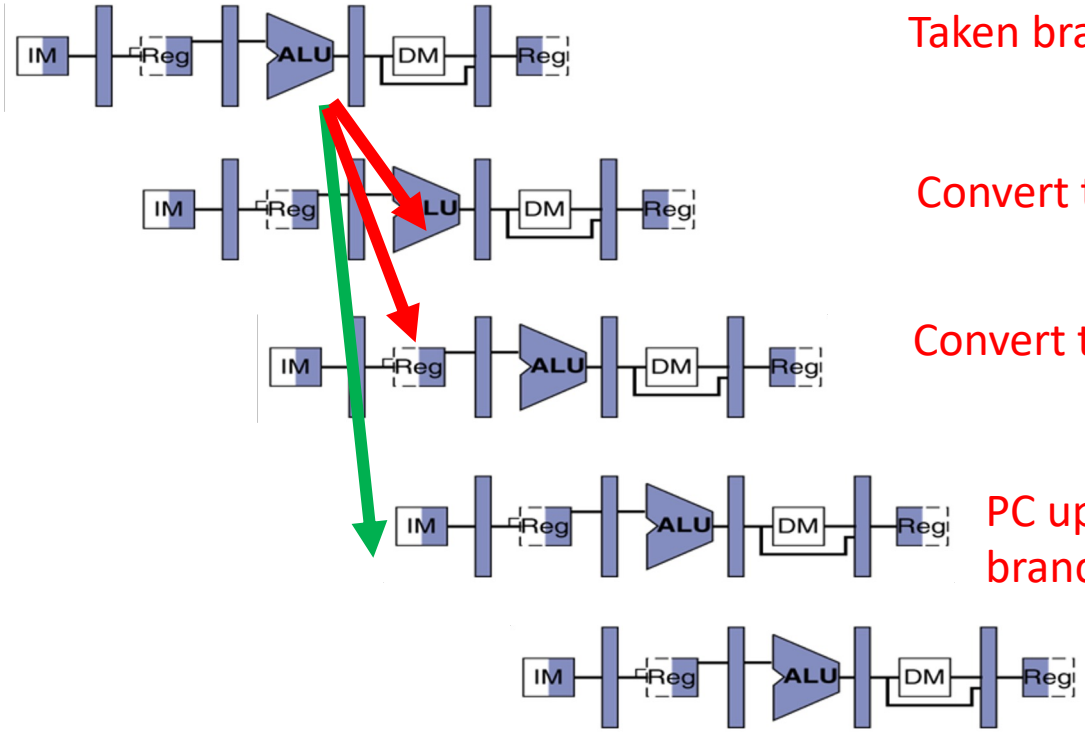
Kill Instructions after Branch if Taken

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

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!

Taken branch (Flush 2 cycles; same as stall)

Address	Ins-Cycle	0	1	2	3	4	5	6
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
0x14	ecall						IF	ID

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?

Not taken w/ Prediction (saved 2 cycles)

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
0x0c	jal zero, 0x00000014				IF	ID	EX

Taken branch (Flush 2 cycles)

Address	Ins-Cycle	0	1	2	3	4	5	6
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
0x14	ecall						IF	ID

Taken w/ Incorrect Prediction (same as stalling)

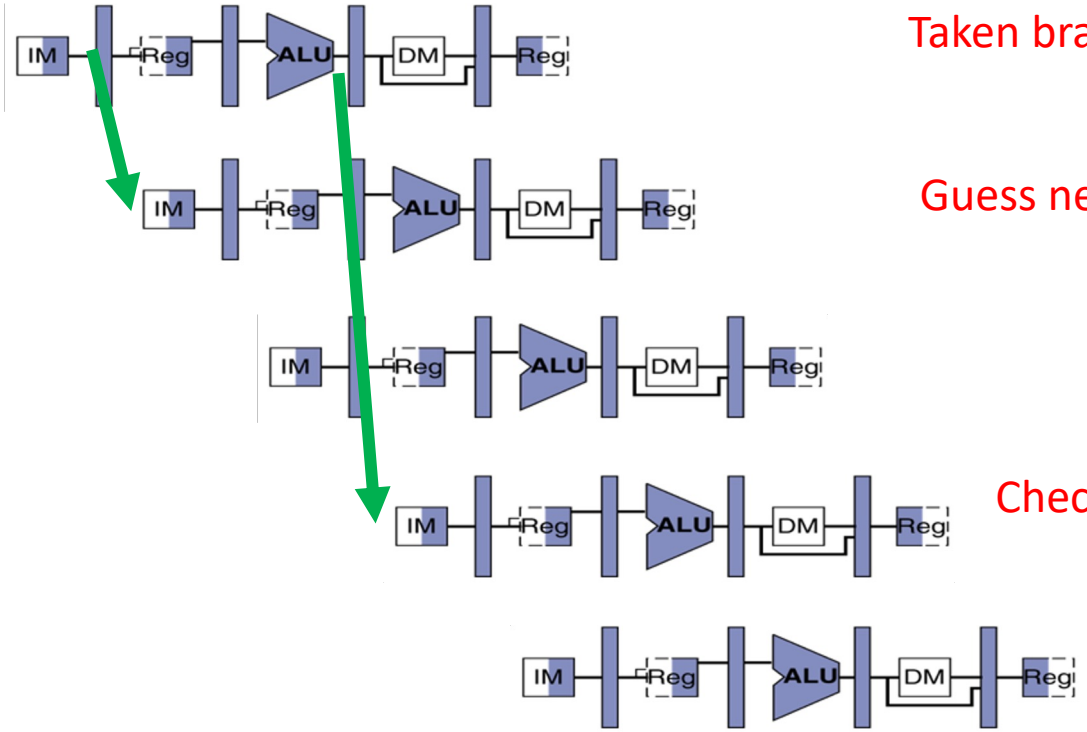
0x00	add a2, a1, a0	IF	ID	EX	MM	WB	
0x04	beq a2, zero, 0x00000010		IF	ID	EX	MM	WB
0x08	addi a3, zero, 1			IF	ID	EX	MM
0x0c	jal zero, 0x00000014				IF	ID	EX

Branch Prediction

beq t0, t1, label

label: …..

…..



Taken branch

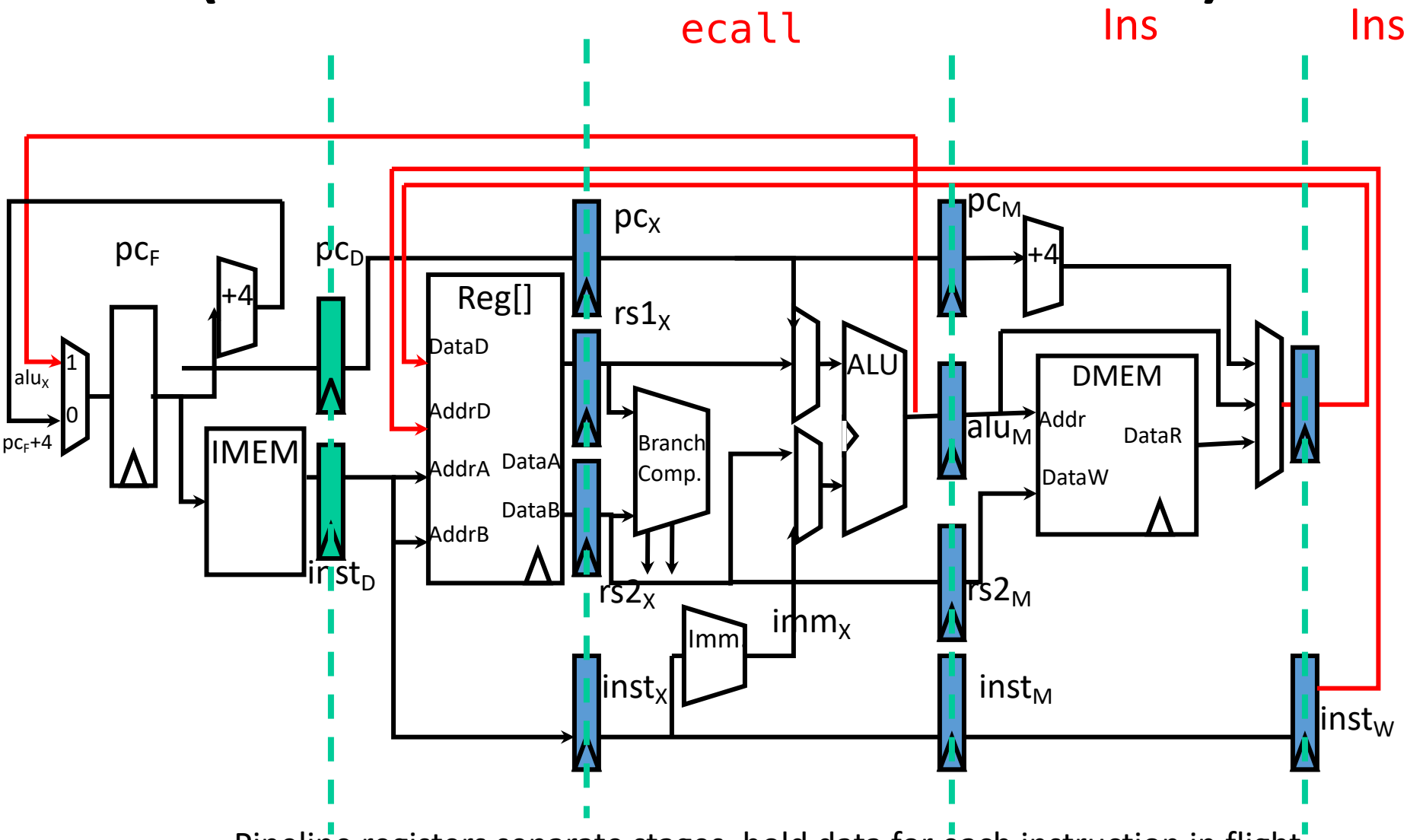
Guess next PC!

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

Ecall (Stall in EX for instructions ahead)



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

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