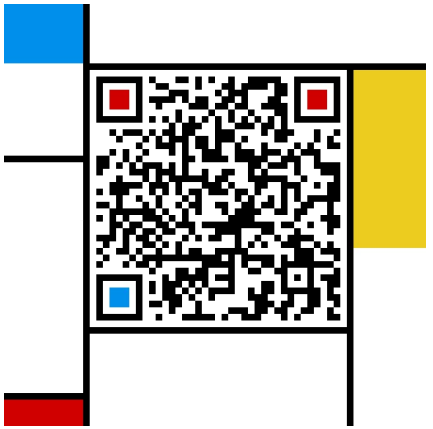
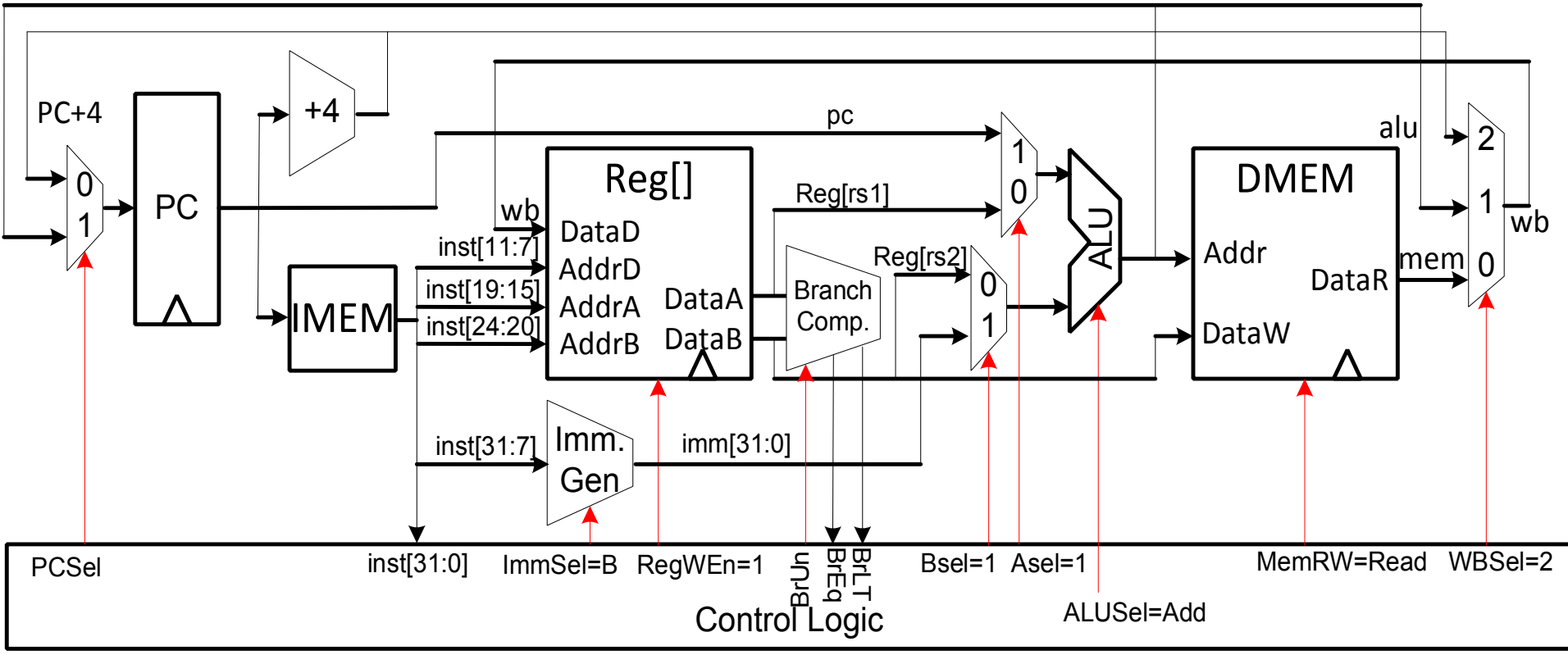
计算机组成与系统结构
Computer Organization & System Architecture

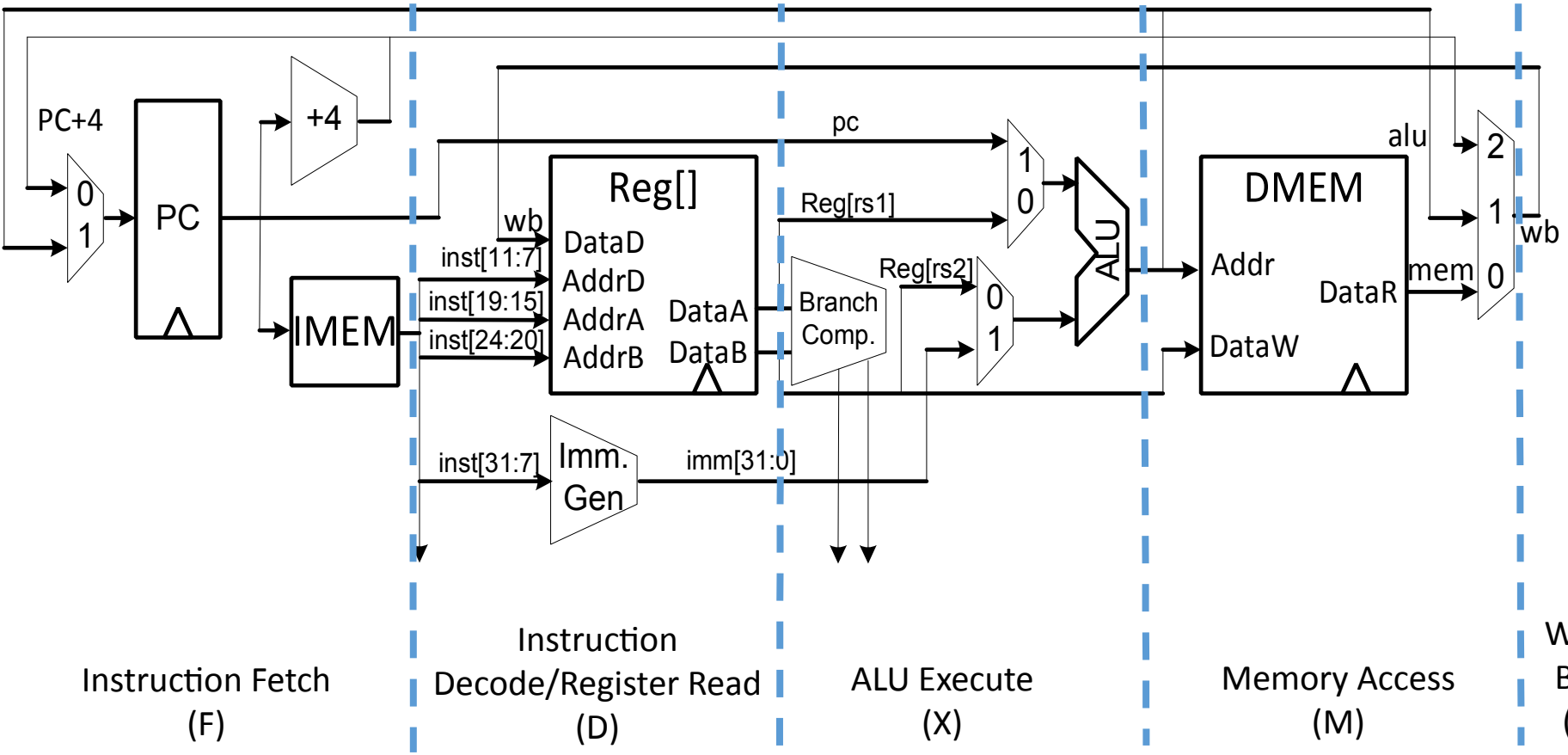
Huang Kejie (黄科杰) 百人计划研究员
Office: 玉泉校区老生仪楼 304
Email address: huangkejie@zju.edu.cn
HP: 17706443800



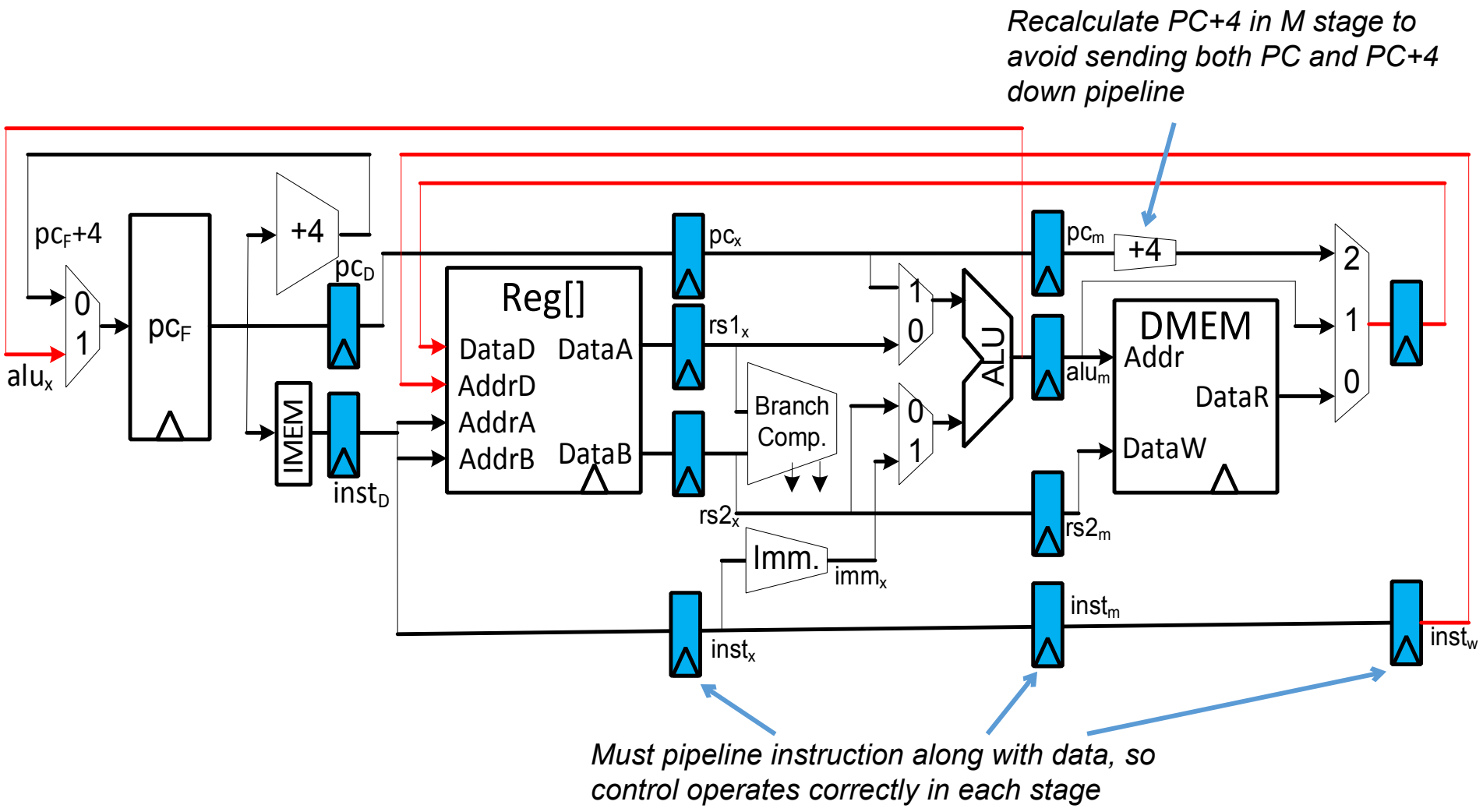
Single-Cycle RISC-V RV32I Datapath



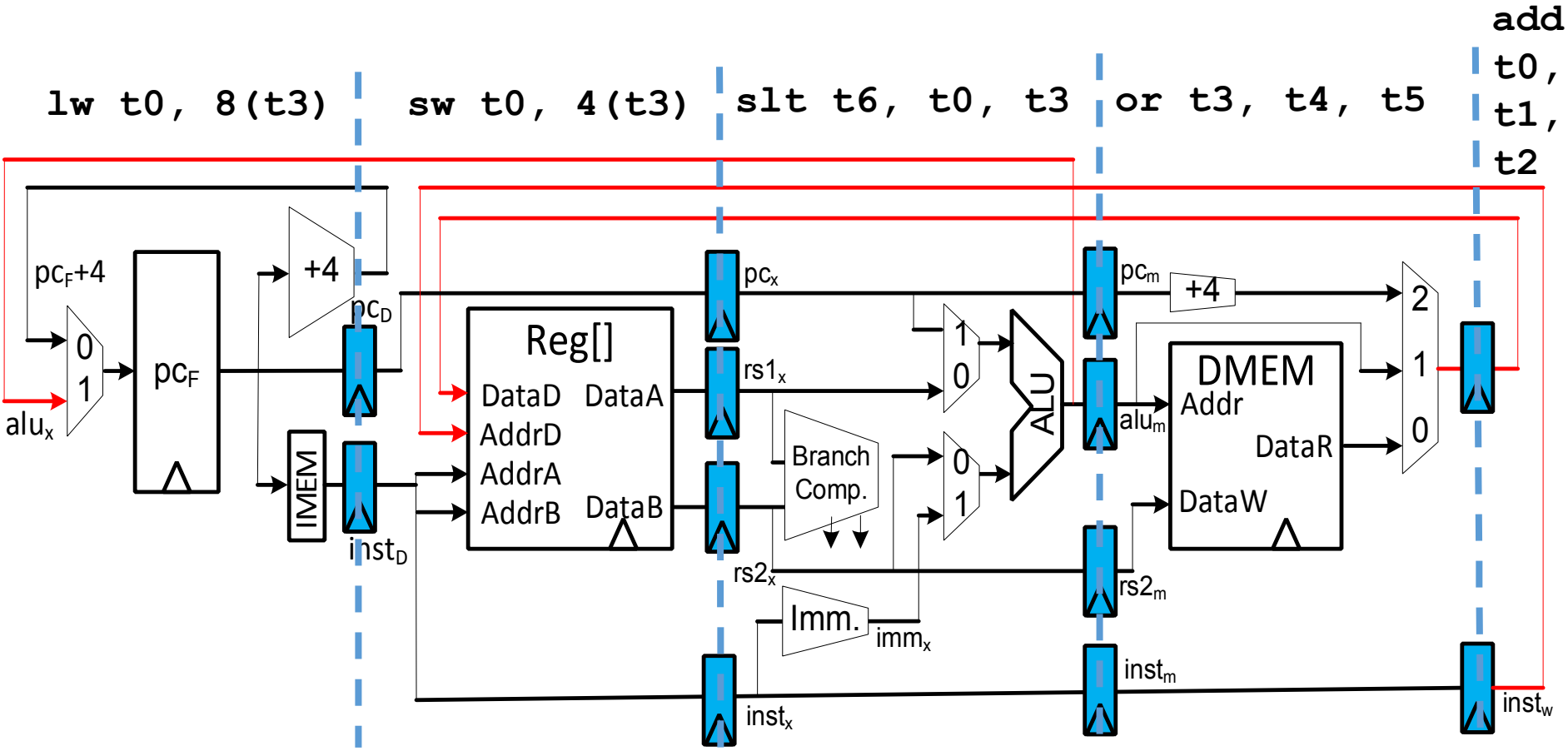
Pipelining RISC-V RV32I Datapath



Pipelined RISC-V RV32I Datapath



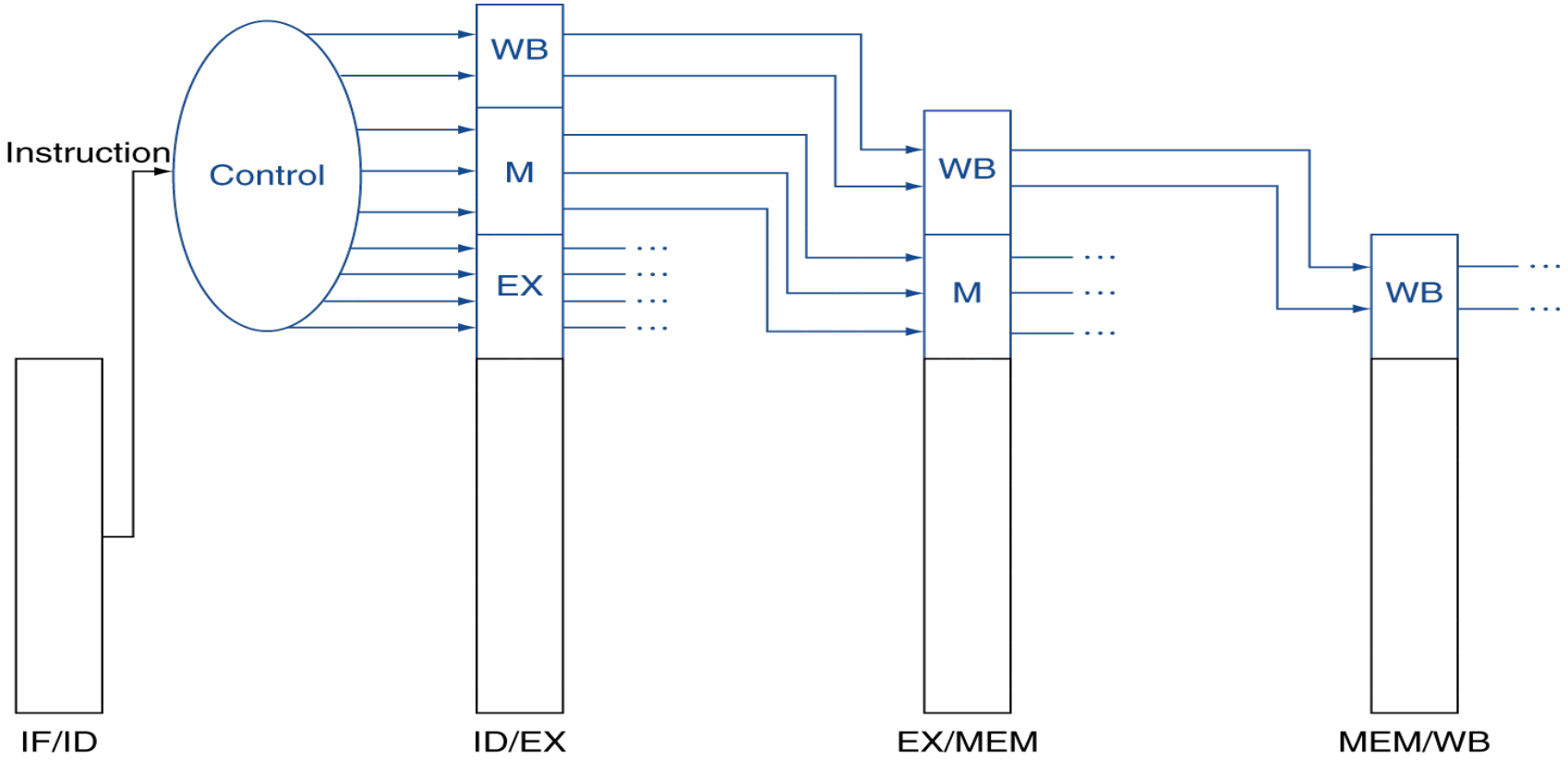
Each stage operates on different instruction



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

Pipelined Control

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



Hazards Ahead

Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*

- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → *data hazard*
  - Dependence may be for the next instruction's address
    → *control hazard (branches, exceptions)*

- Handling hazards generally introduces bubbles into pipeline and reduces ideal CPI > 1

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
    - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory

- Can always solve a structural hazard by adding more hardware

- Classic RISC 5-stage integer pipeline has no structural hazards by design
  - Many RISC implementations have structural hazards on multi-cycle units such as multipliers, dividers, floating-point units, etc., and can have on register writeback ports

Regfile Structural Hazards

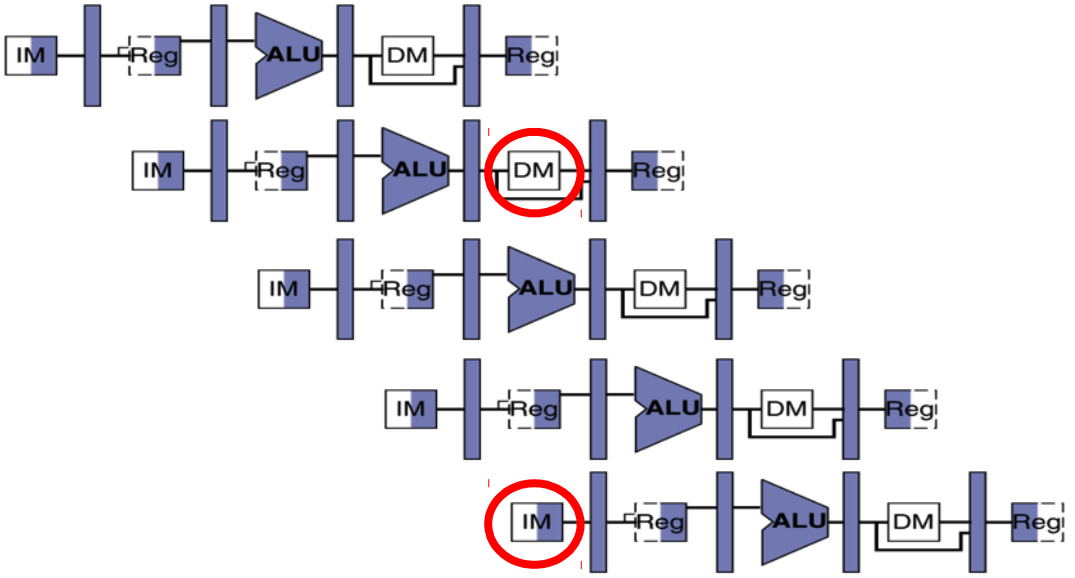
- Each instruction:
  - can read up to two operands in decode stage
  - can write one value in writeback stage
- Avoid structural hazard by having separate “ports”
  - two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously

Structural Hazard: Memory Access

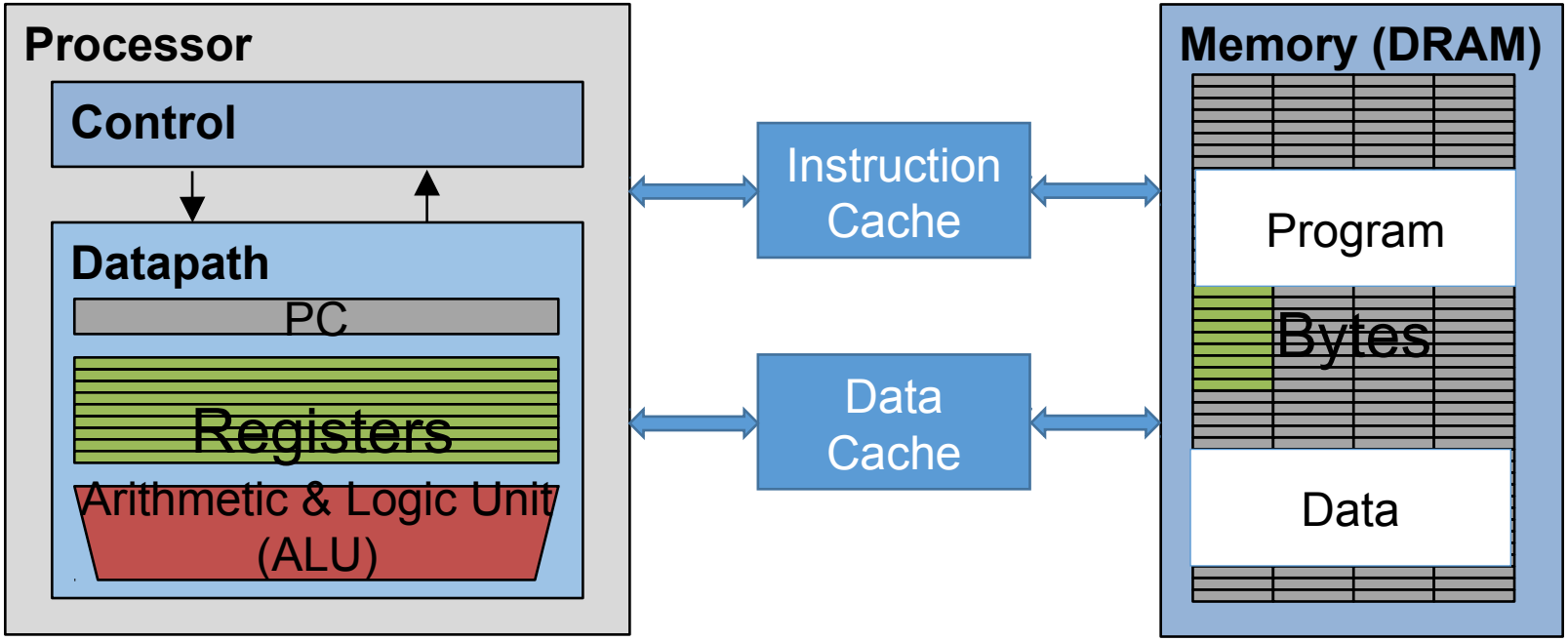
- Instruction and data memory used simultaneously
  - Use two separate memories

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 Caches



Caches: small and fast “buffer” memories

Structural Hazards – Summary

- Conflict for use of a resource

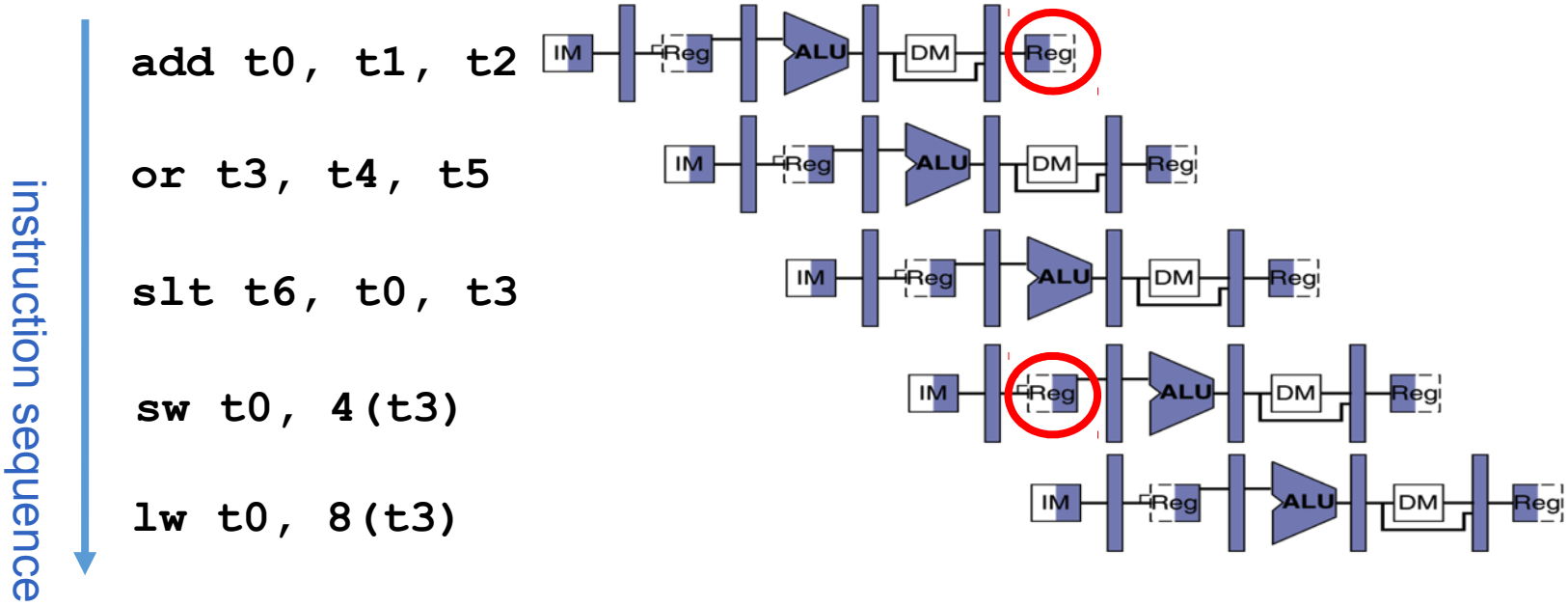
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Without separate memories, instruction fetch would have to *stall* for that cycle
    - All other operations in pipeline would have to wait

- 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

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?

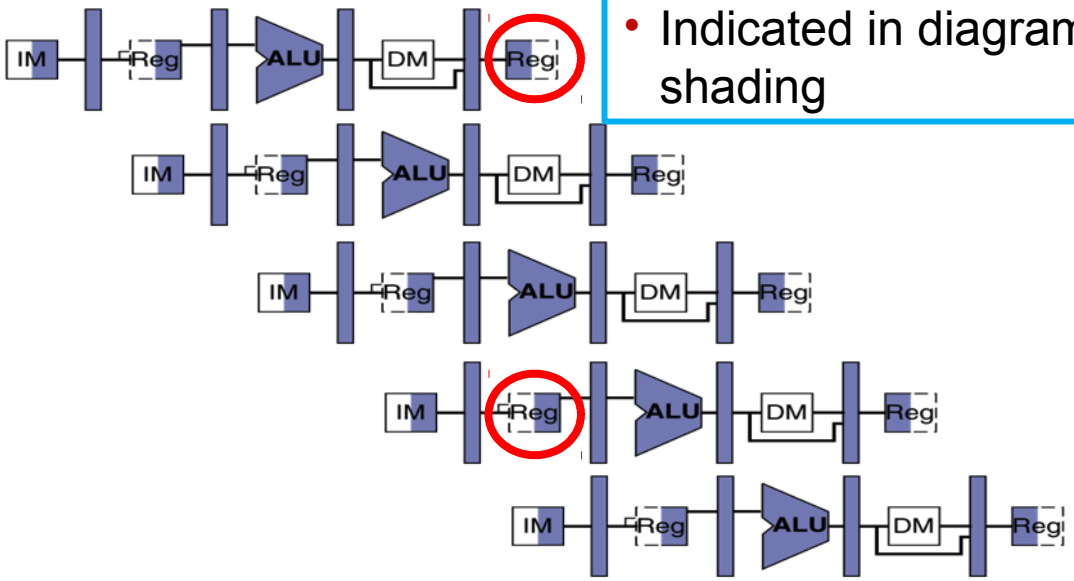


Register Access Policy

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

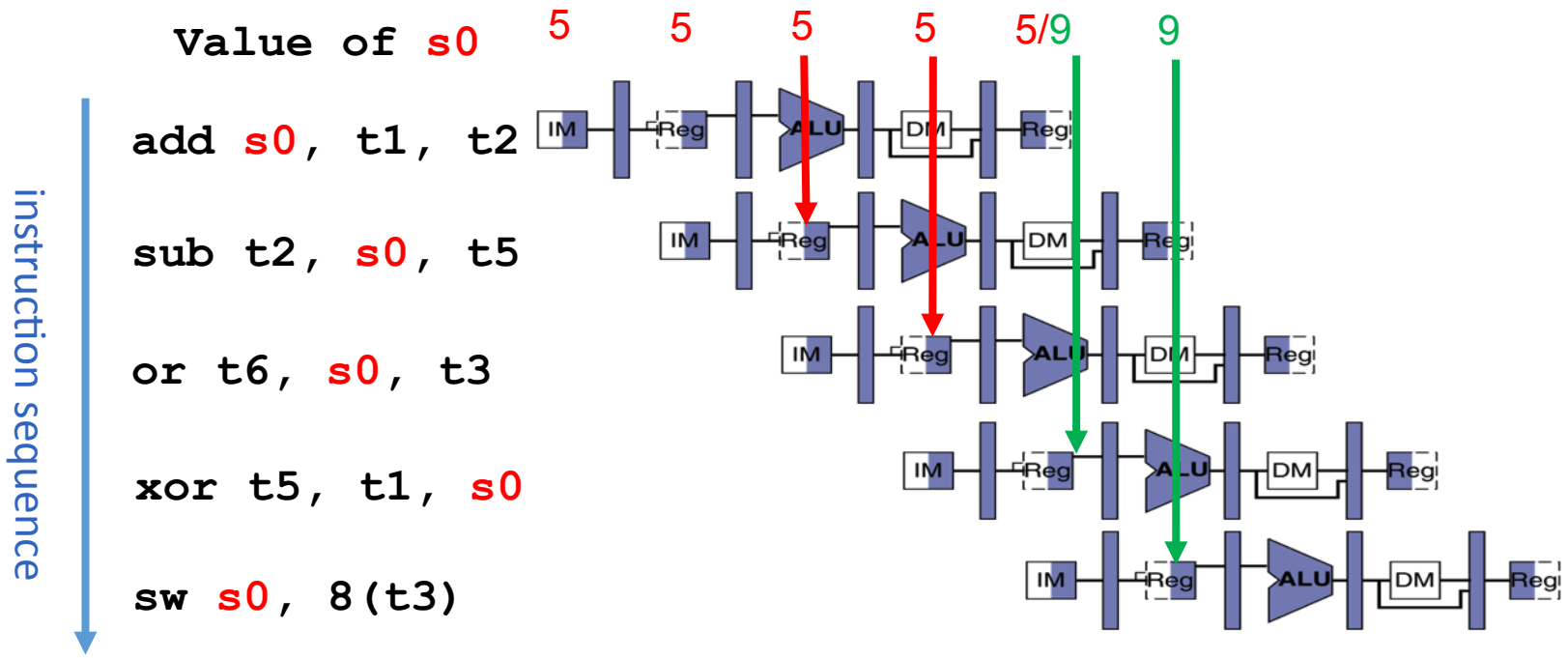
instruction sequence

```
add t0, t1, t2
or t3, t4, t5
slt t6, t0, t3
sw t0, 4(t3)
lw t0, 8(t3)
```



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

Data Hazard: ALU Result

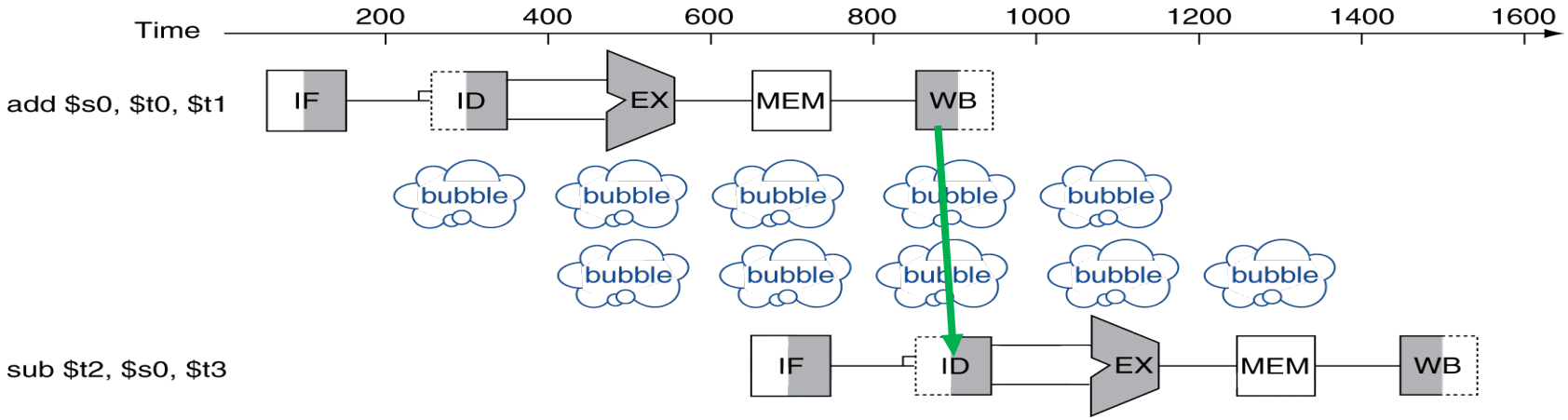


Without some fix, **`sub`** and **`or`** will calculate wrong result!

Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction

```
add  s0, t0, t1
sub  t2, s0, t3
```



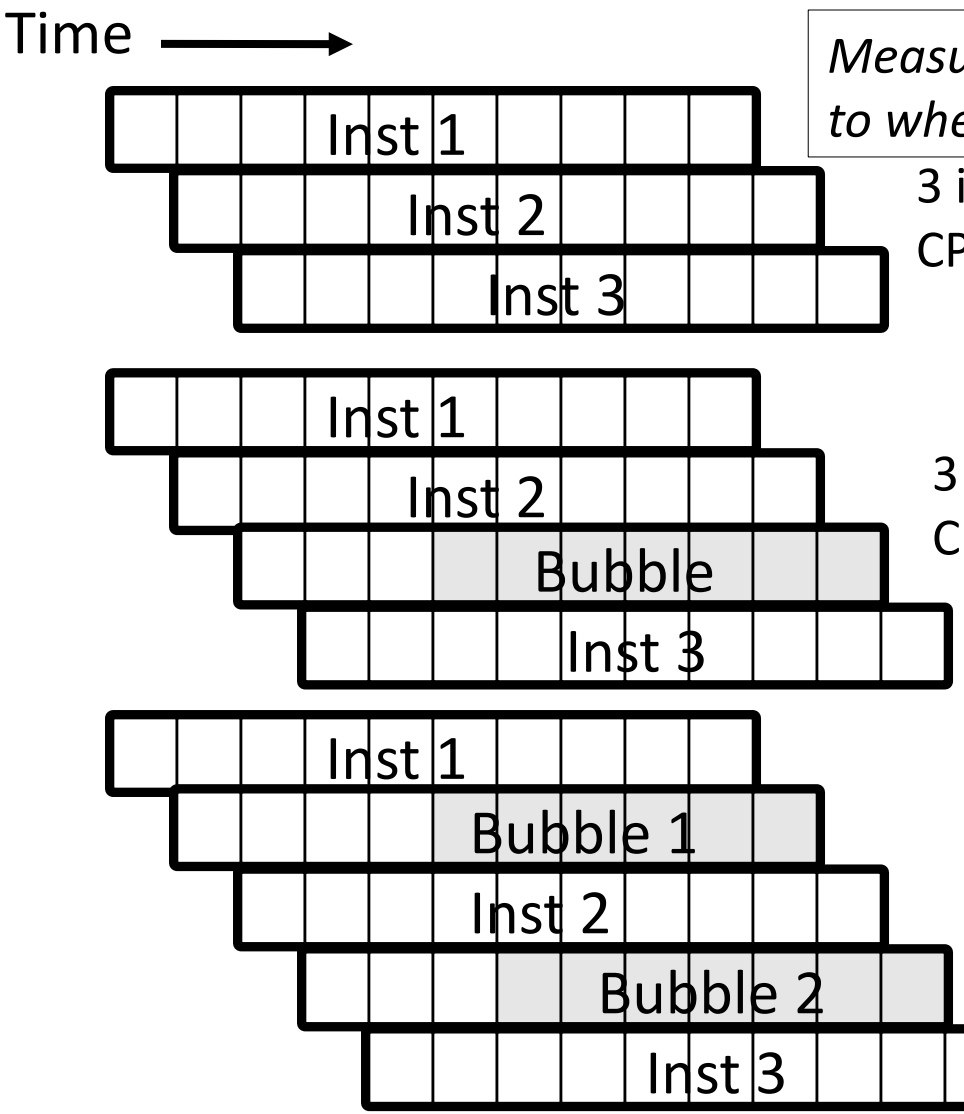
- Bubble:
  - effectively NOP: affected pipeline stages do “nothing”

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

Pipeline CPI Examples



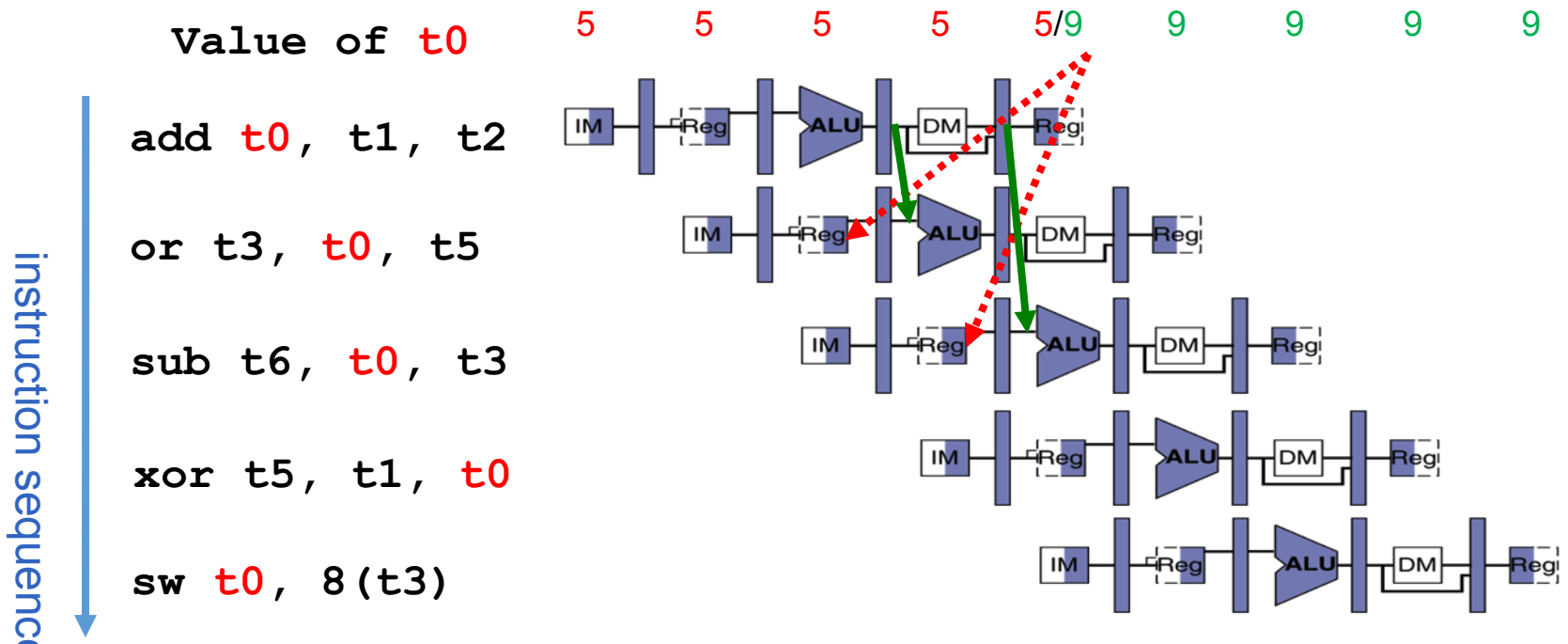
Measure from when first instruction finishes to when last instruction in sequence finishes.

3 instructions finish in 3 cycles
CPI = 3/3 =1

3 instructions finish in 4 cycles
CPI = 4/3 = 1.33

3 instructions finish in 5cycles
CPI = 5/3 = 1.67

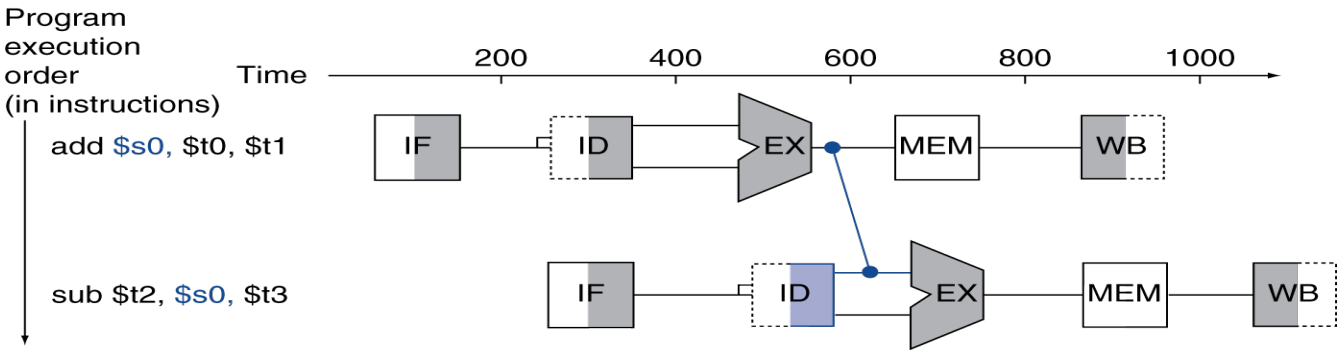
Solution 2: Forwarding



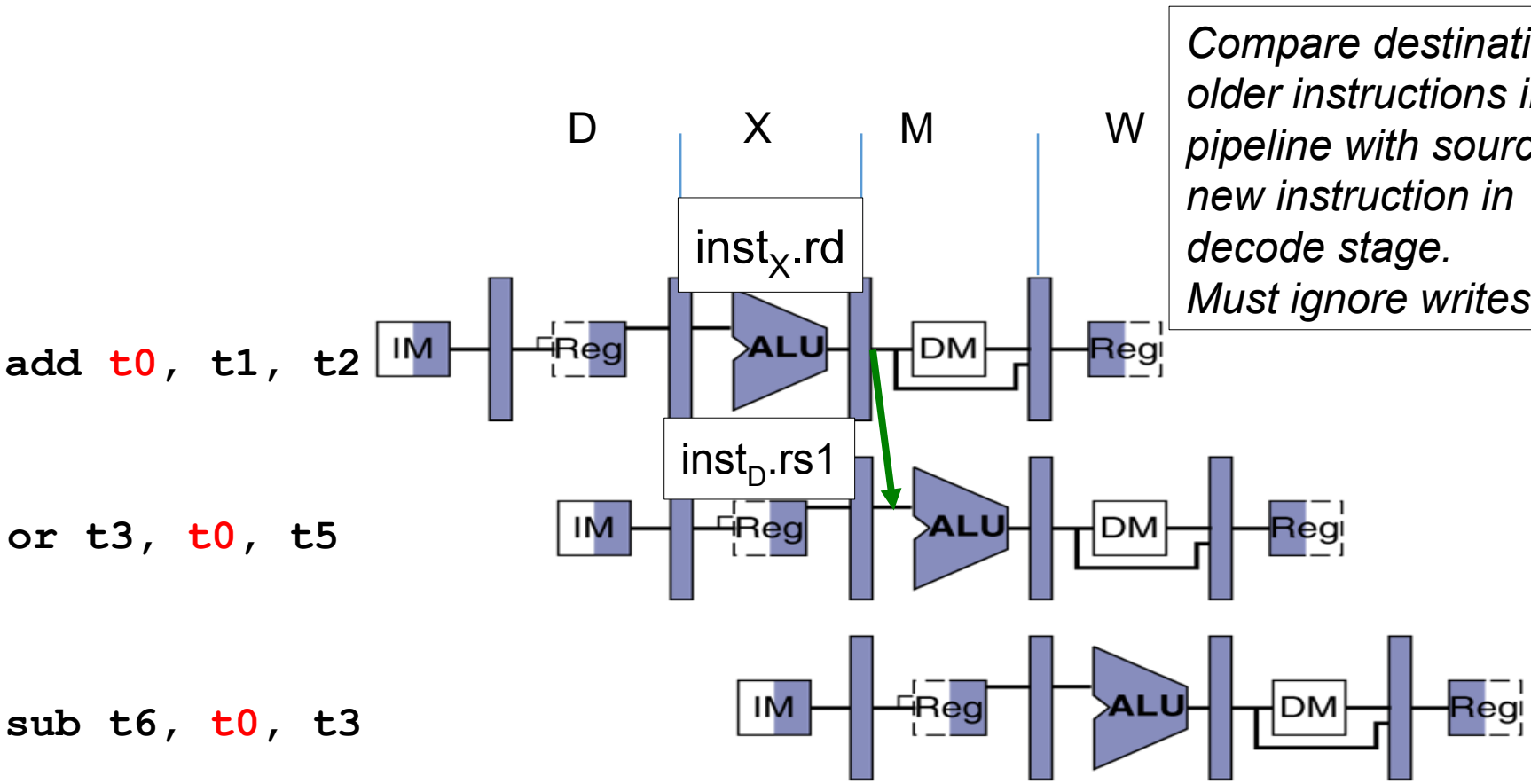
Forwarding: grab operand from pipeline stage, rather than register file

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

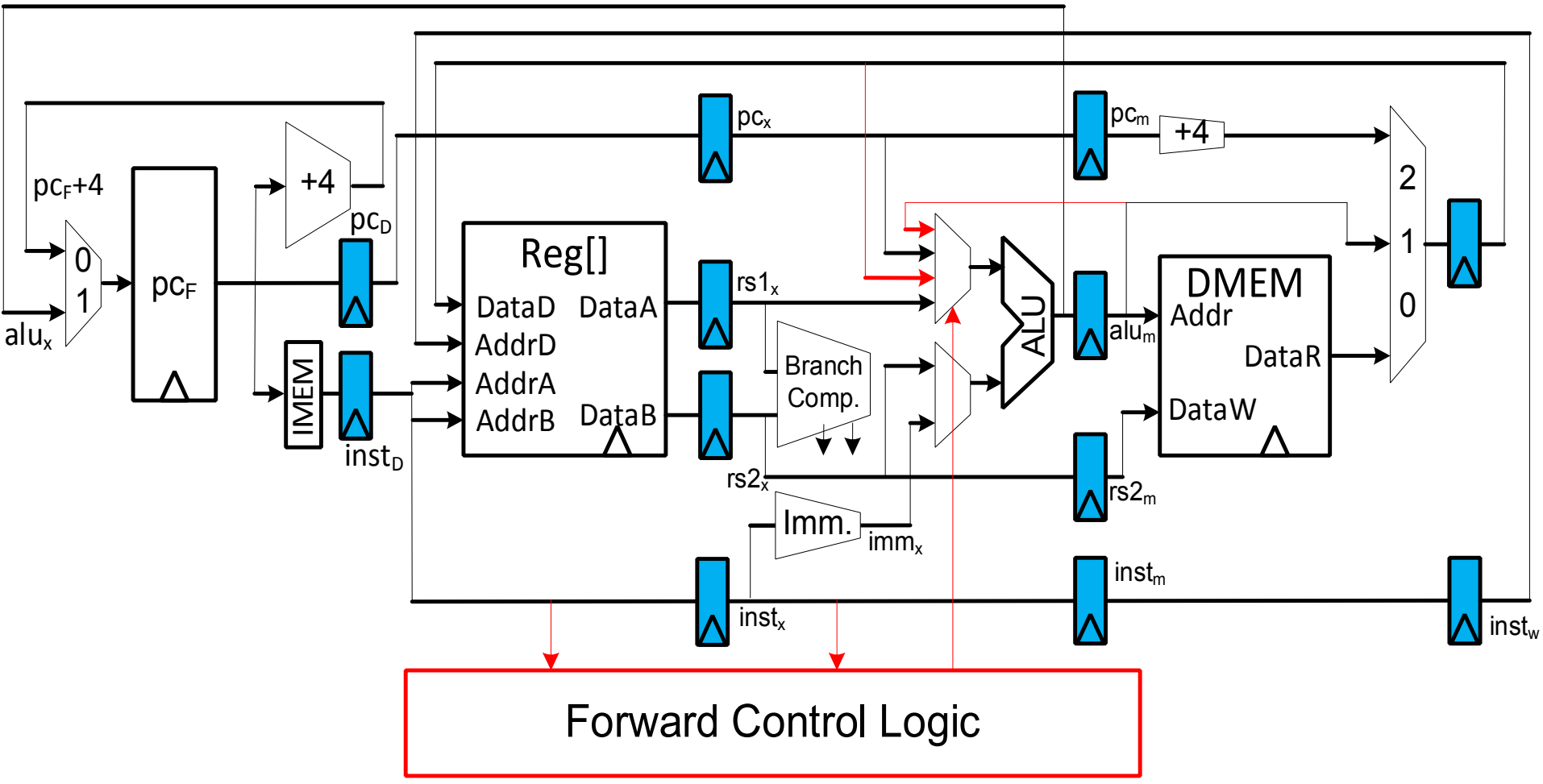


1) Detect Need for Forwarding (example)

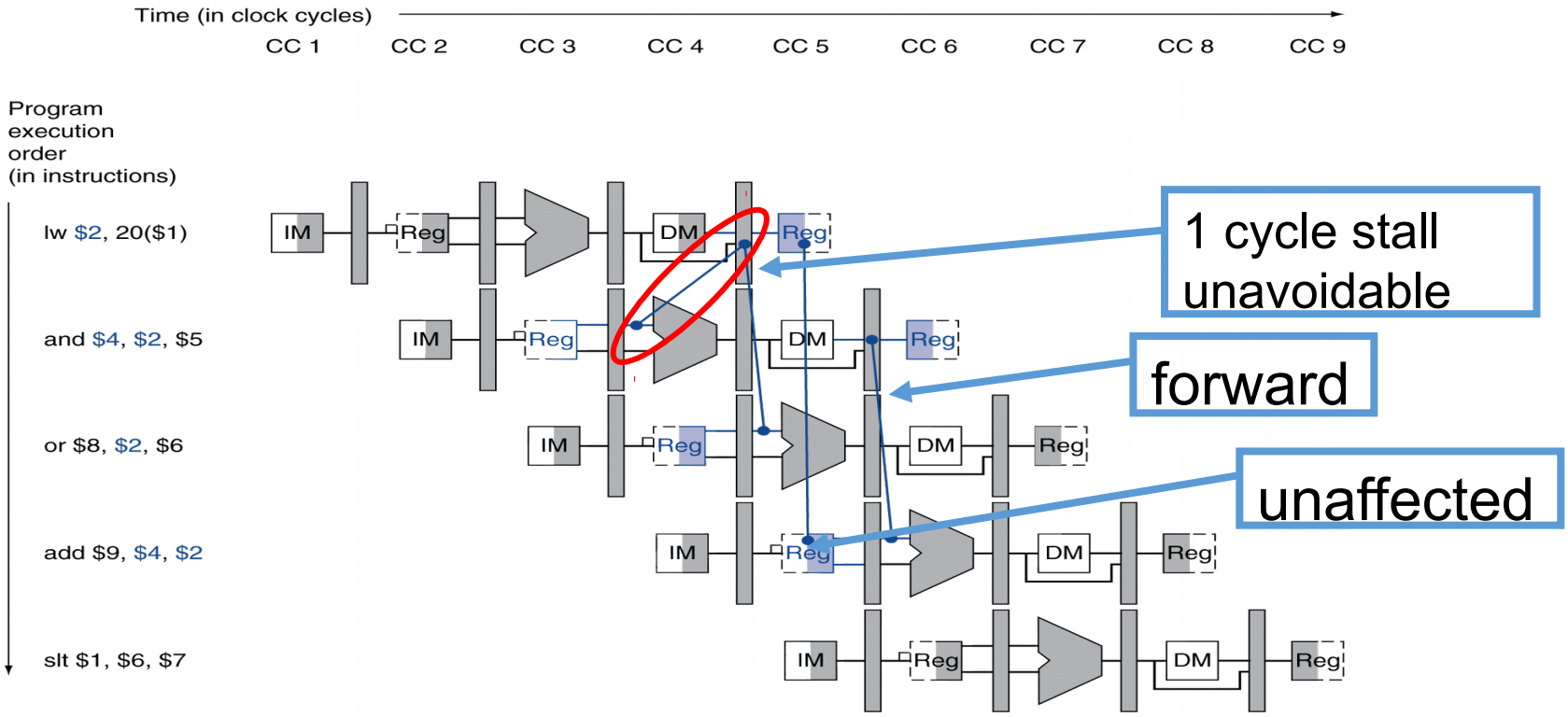


Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
Must ignore writes to x0!

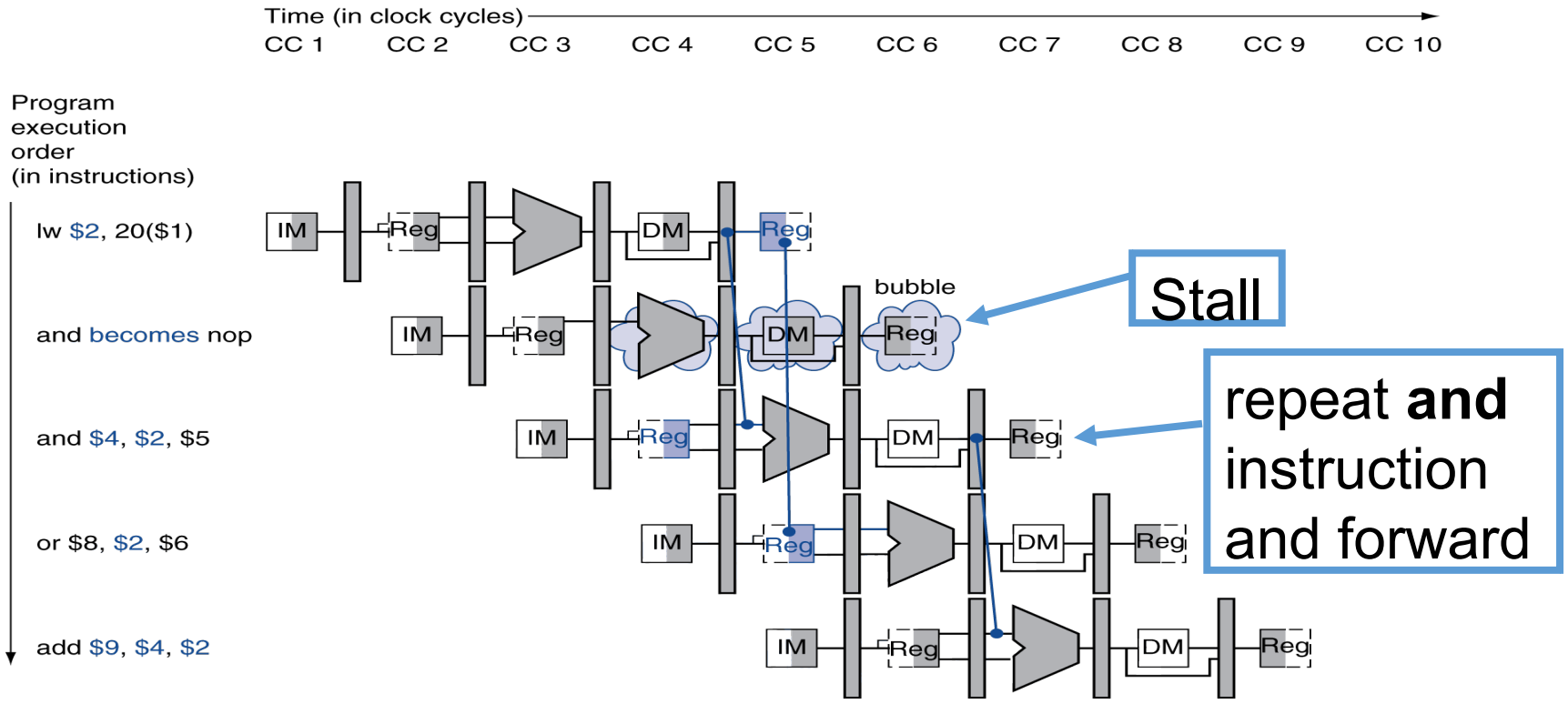
Forwarding Path



Load Data Hazard



Stall Pipeline



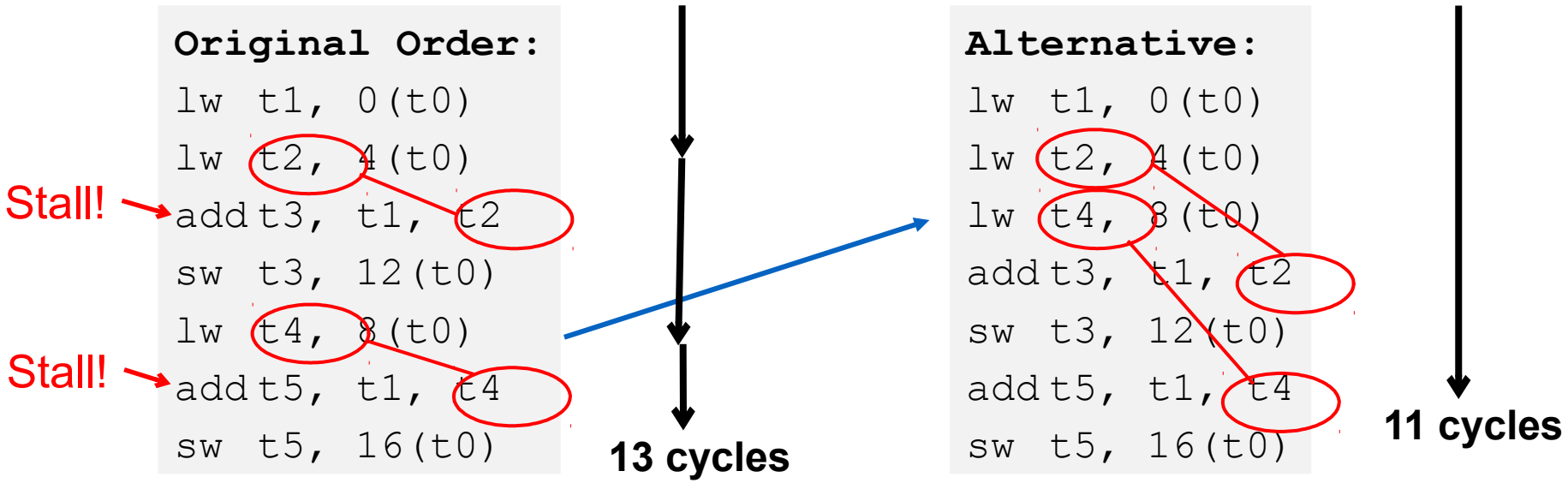
`lw` Data Hazard

- 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;`**

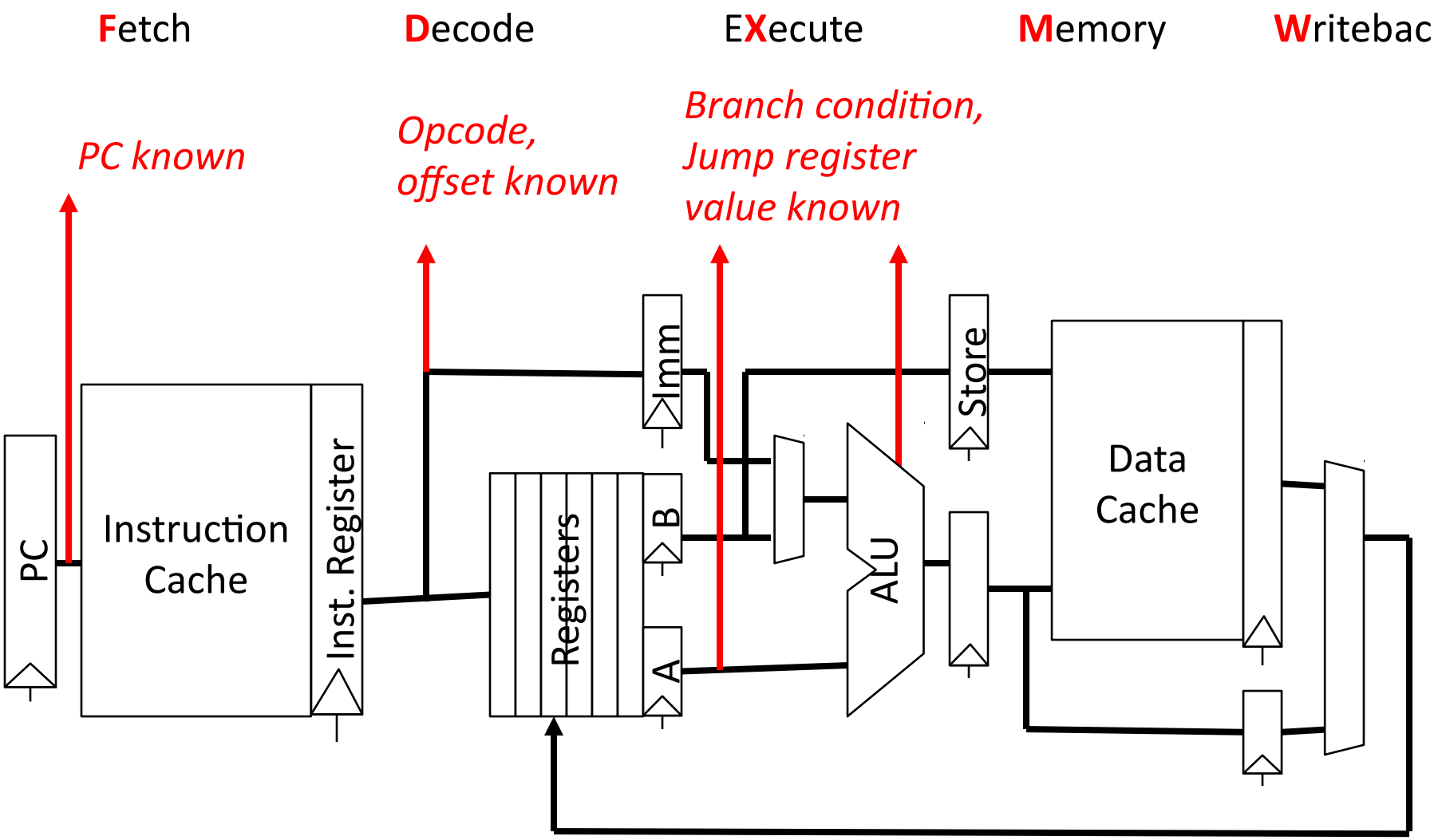


Control Hazards

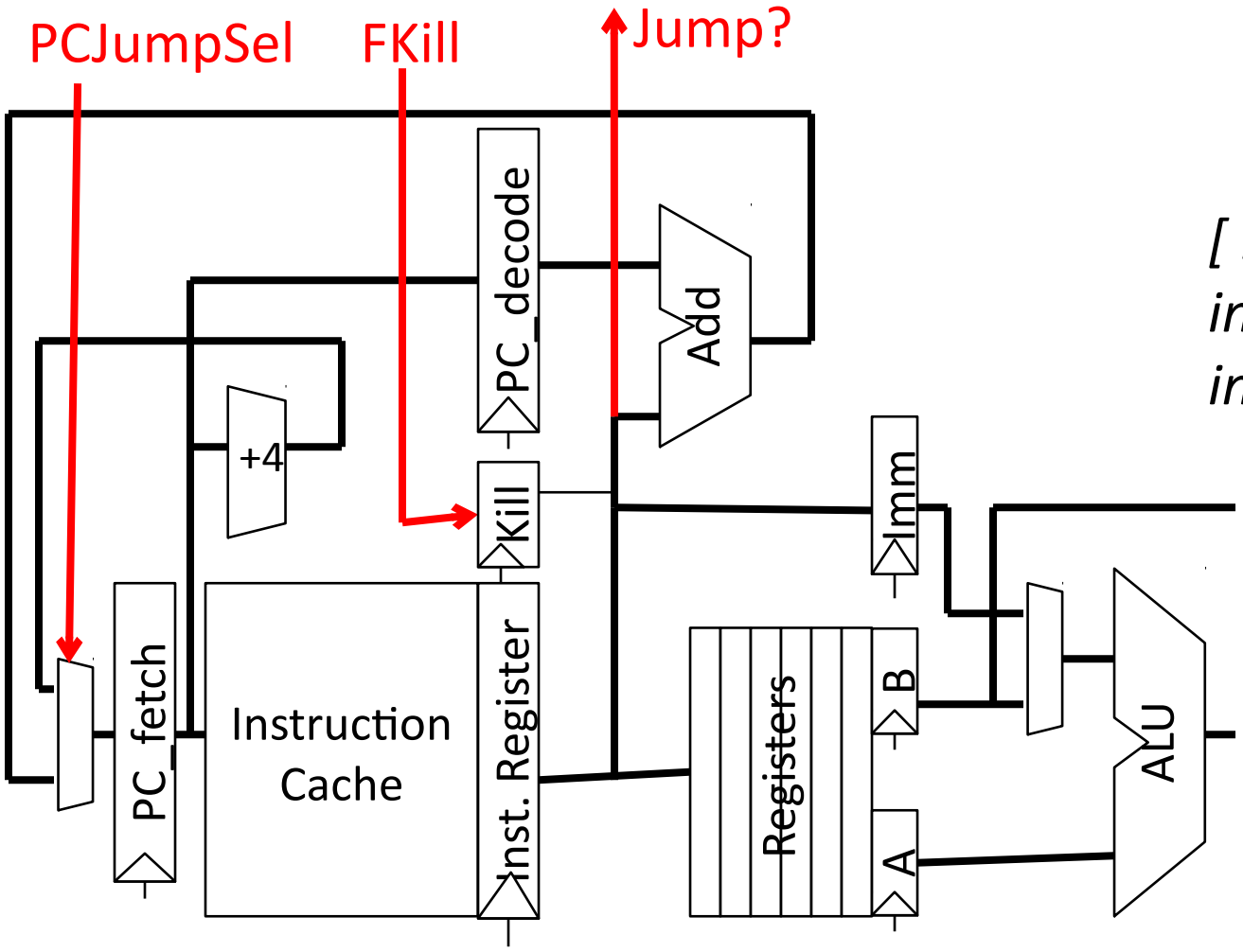
- What do we need to calculate next PC?

- For Unconditional Jumps
  - Opcode, PC, and offset
- For Jump Register
  - Opcode, Register value, and offset
- For Conditional Branches
  - Opcode, Register (for condition), PC and offset
- For all other instructions
  - Opcode and PC (and have to know it's not one of above)

Control flow information in pipeline

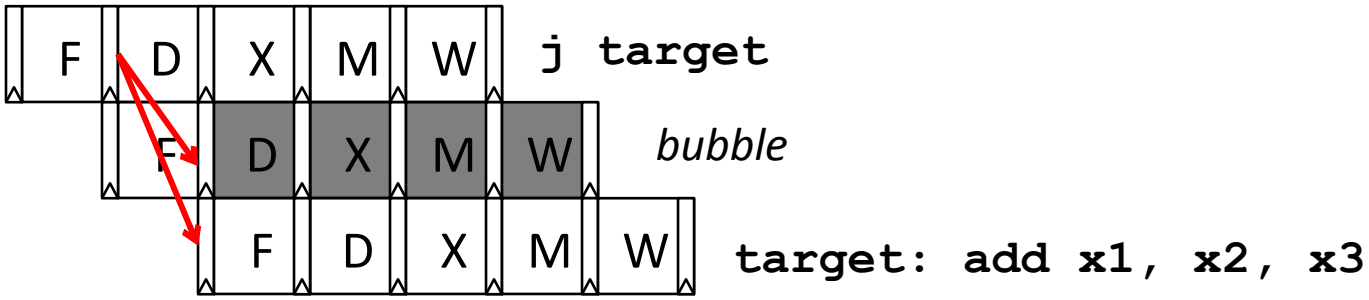


RISC-V Unconditional PC-Relative Jumps



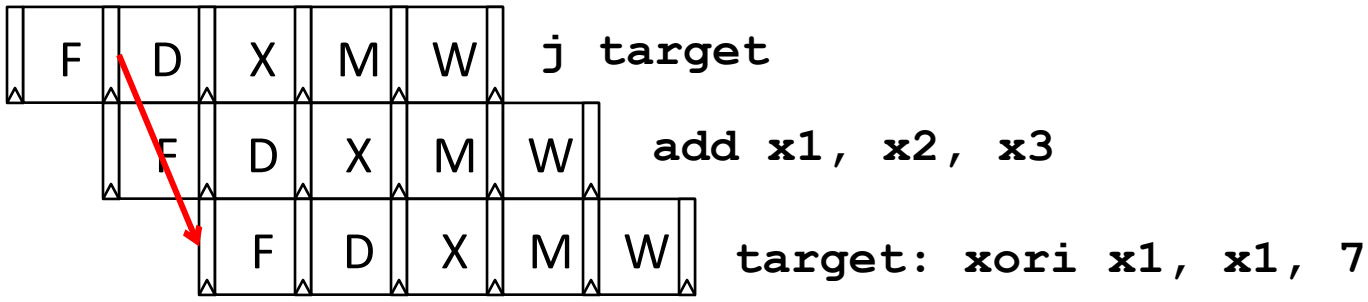
[Kill bit turns instruction into a bubble]

Pipelining for Unconditional PC-Relative Jumps



Branch Delay Slots

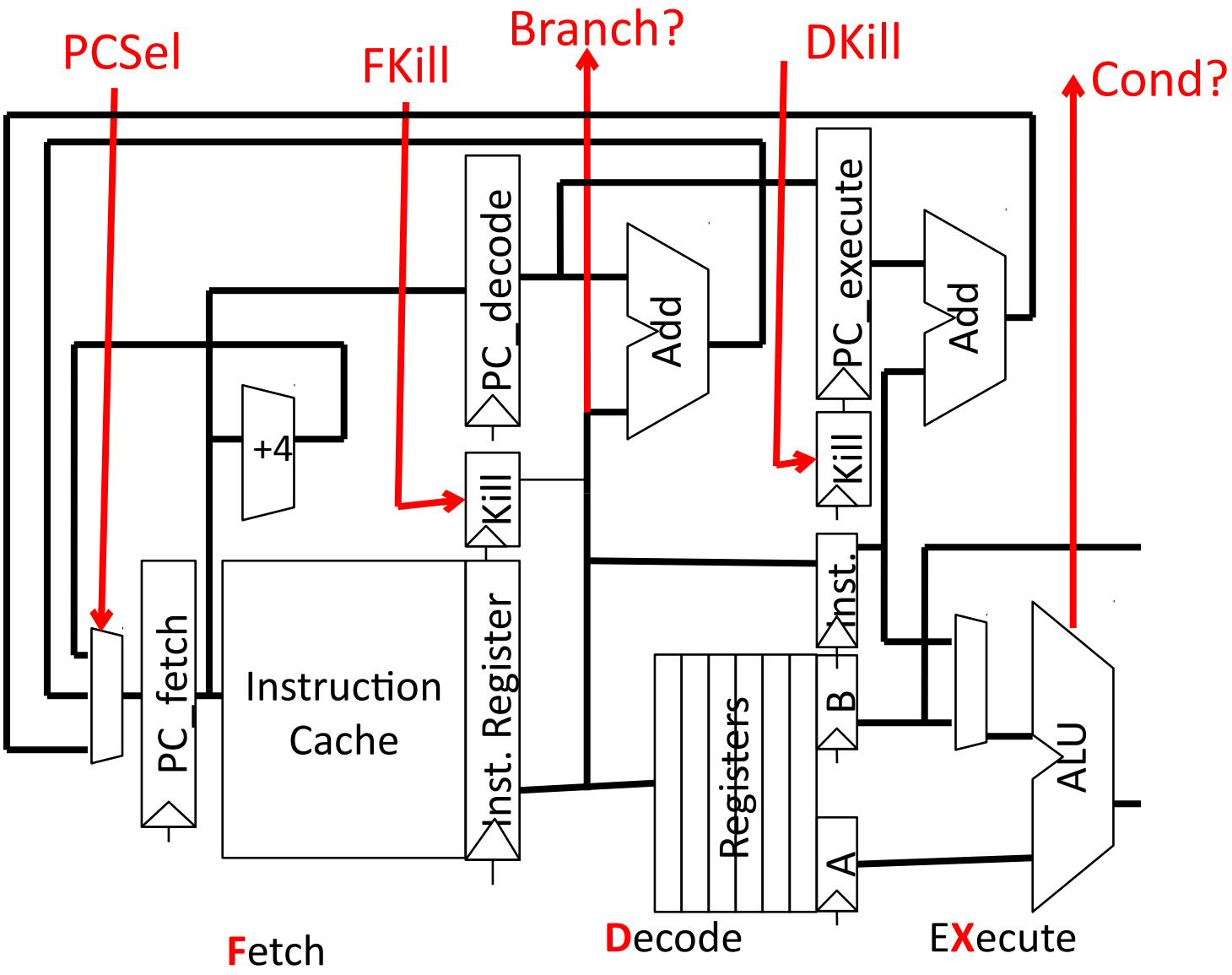
- Early RISCs adopted idea from pipelined microcode engines, and changed ISA semantics so instruction after branch/jump is always executed before control flow change occurs:
  - 0x100 j target
  - 0x104 add x1, x2, x3 // Executed before target
  - …
  - 0x205 target: xori x1, x1, 7
- Software has to fill delay slot with useful work, or fill with explicit NOP instruction



Post-1990 RISC ISAs don't have delay slots

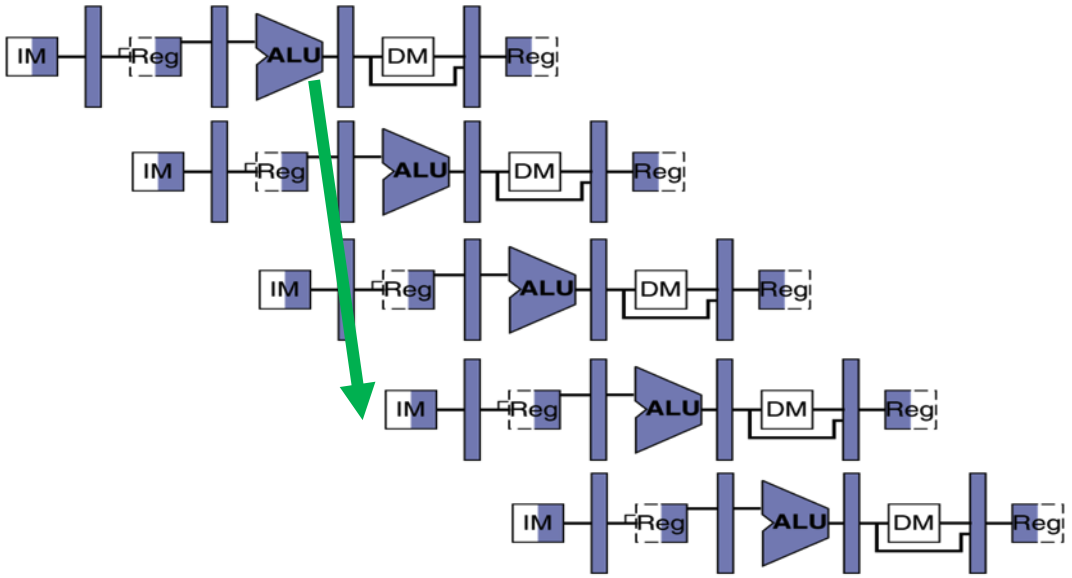
- Encodes microarchitectural detail into ISA
  - c.f. IBM 650 drum layout
- Performance issues
  - Increased I-cache misses from NOPs in unused delay slots
  - I-cache miss on delay slot causes machine to wait, even if delay slot is a NOP
- Complicates more advanced microarchitectures
  - Consider 30-stage pipeline with four-instruction-per-cycle issue
- Better branch prediction reduced need
  - Branch prediction in later lecture

RISC-V Conditional Branches



Pipelining for Conditional Branches

```
beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

xor t5, t1, s0

sw s0, 8(t3)
```



executed regardless of branch outcome!

executed regardless of branch outcome!!!

PC updated reflecting branch outcome

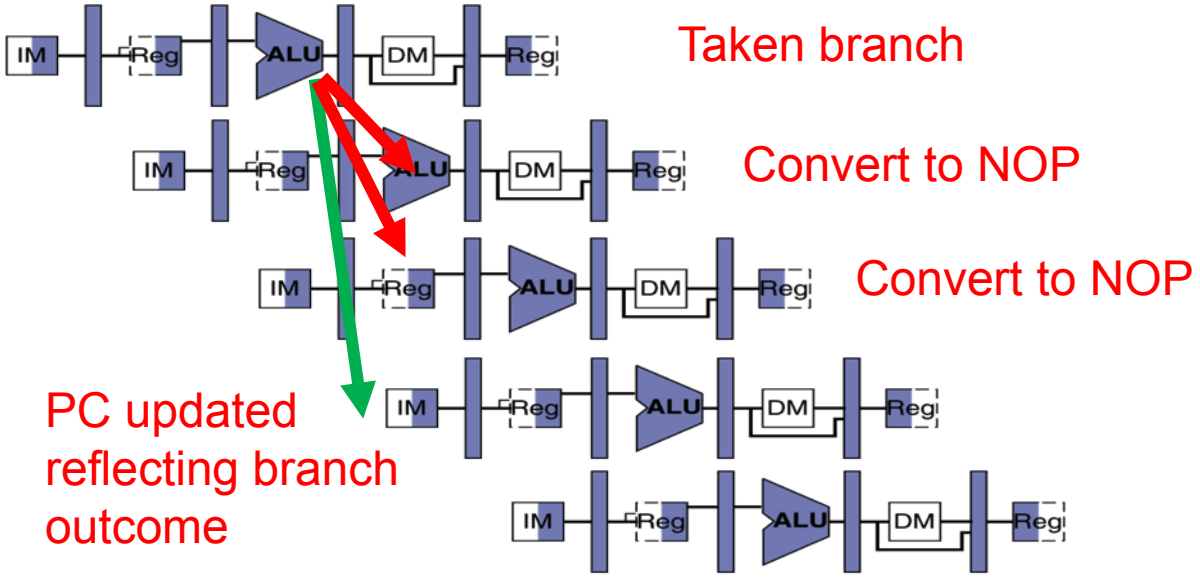
Observation

- 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

Kill Instructions after Branch if Taken

```
beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

label: xxxxxx
```

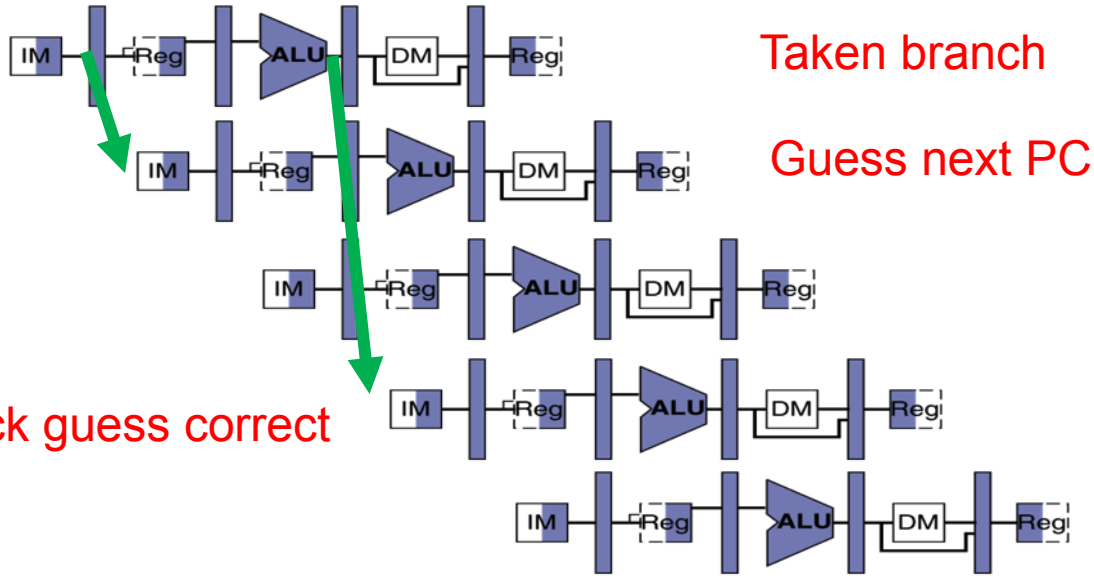


Reducing Branch Penalties

- 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

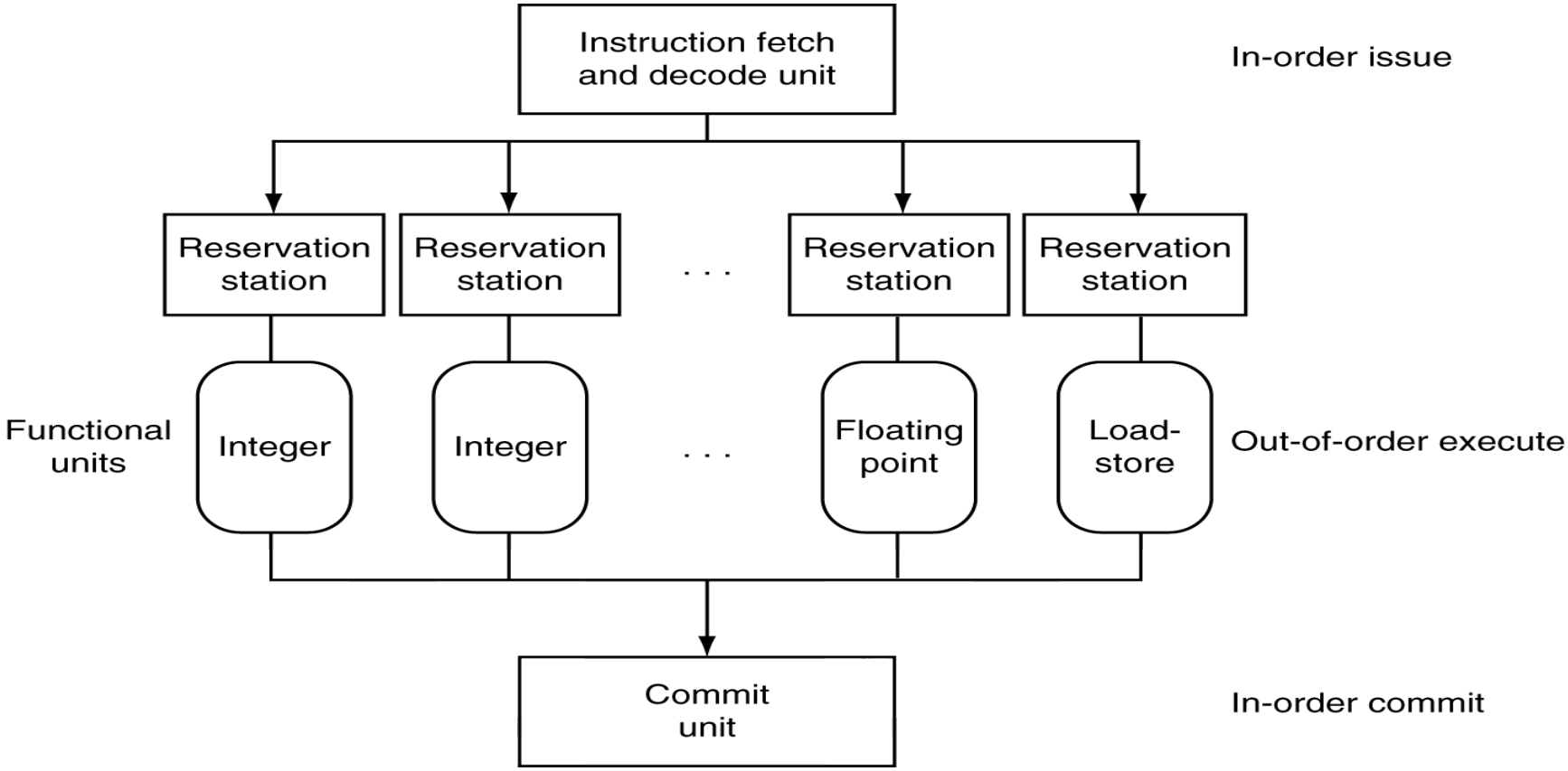
```
beq t0, t1, label

label: .....

.....
```



Increasing Processor Performance

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

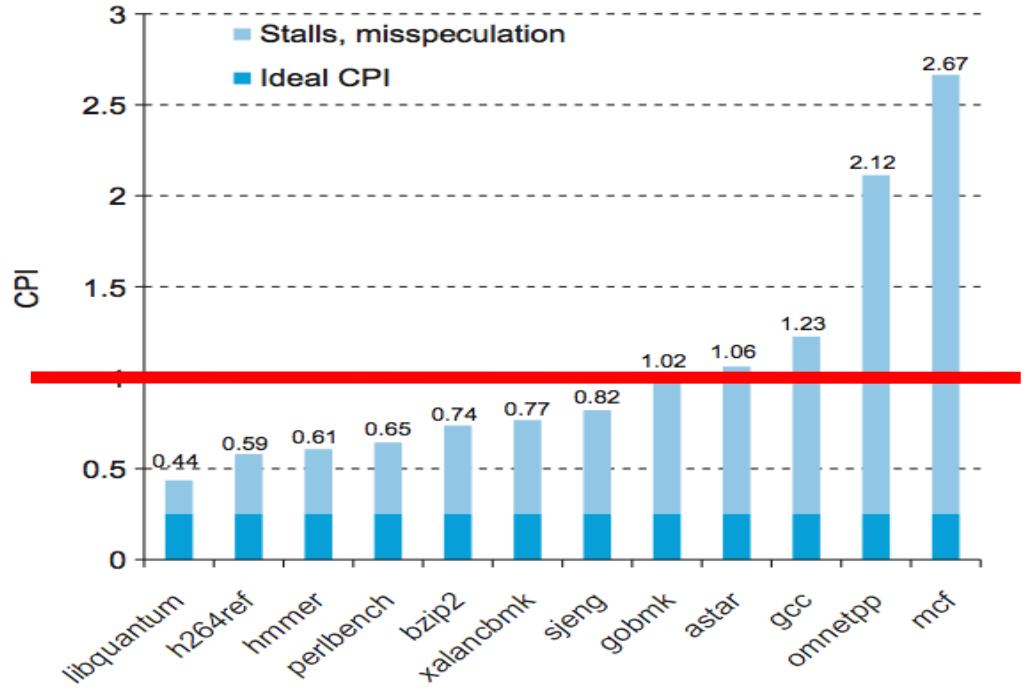
Superscalar Processor



P&H p. 340

Benchmark: CPI of Intel Core i7

CPI = 1



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

P&H p. 350

Why instruction may not be dispatched every cycle in classic 5-stage pipeline (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two-cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined load delay slots, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II (pipeline interlocks added in hardware)
    - MIPS:"Microprocessor without Interlocked Pipeline Stages"
- Jumps/Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.
NOPs reduce CPI, but increase instructions/program!

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