

Topic 3

Fundamental Processors

Xinfei Guo
xinfei.guo@sjtu.edu.cn

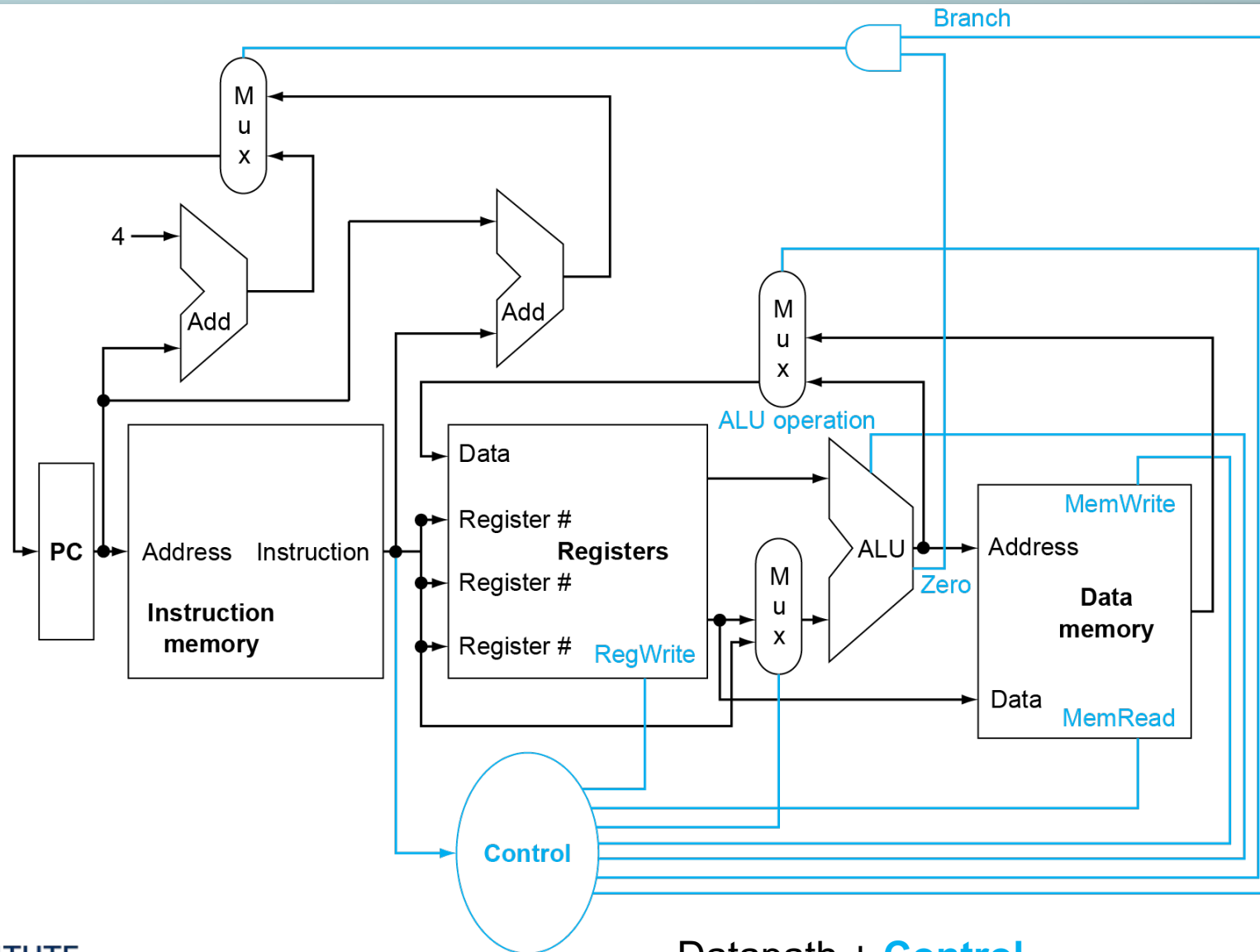
May 20th, 2024



T3 learning goals

- Fundamental Processors (In-order Pipelines)
 - Pipelining
 - Hazards

Recap: a simple processor



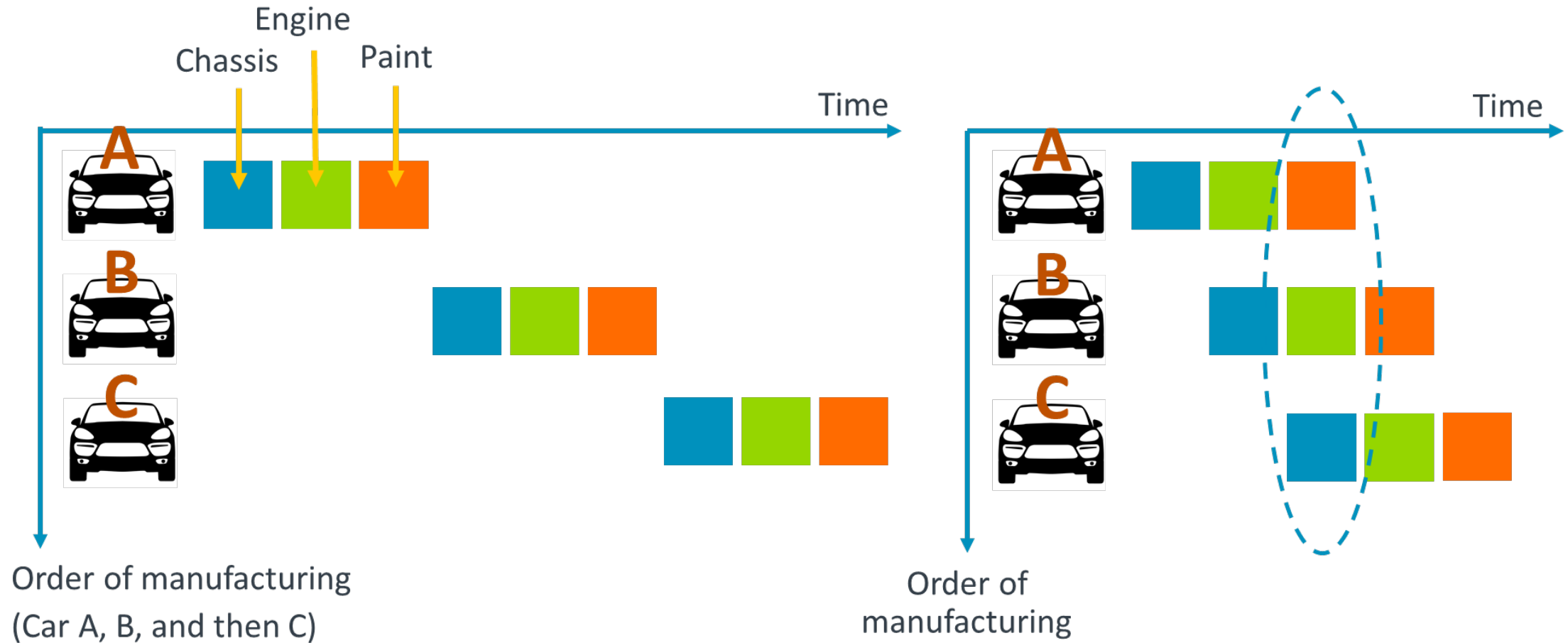
A single cycle
RISC-V
Processor

Datapath + Control

A Simple Processor

- Our processor executes each instruction in one clock cycle, i.e., it has a Cycles Per Instruction (**CPI**) of 1.
- The minimum clock period will be the worst-case path through all of the logic and memories shown (**plus some margin for variations in Process, Voltage, and Temperature, also known as “PVT”**).
- How might we improve our clock frequency without significantly increasing CPI?

What Is Pipelining?

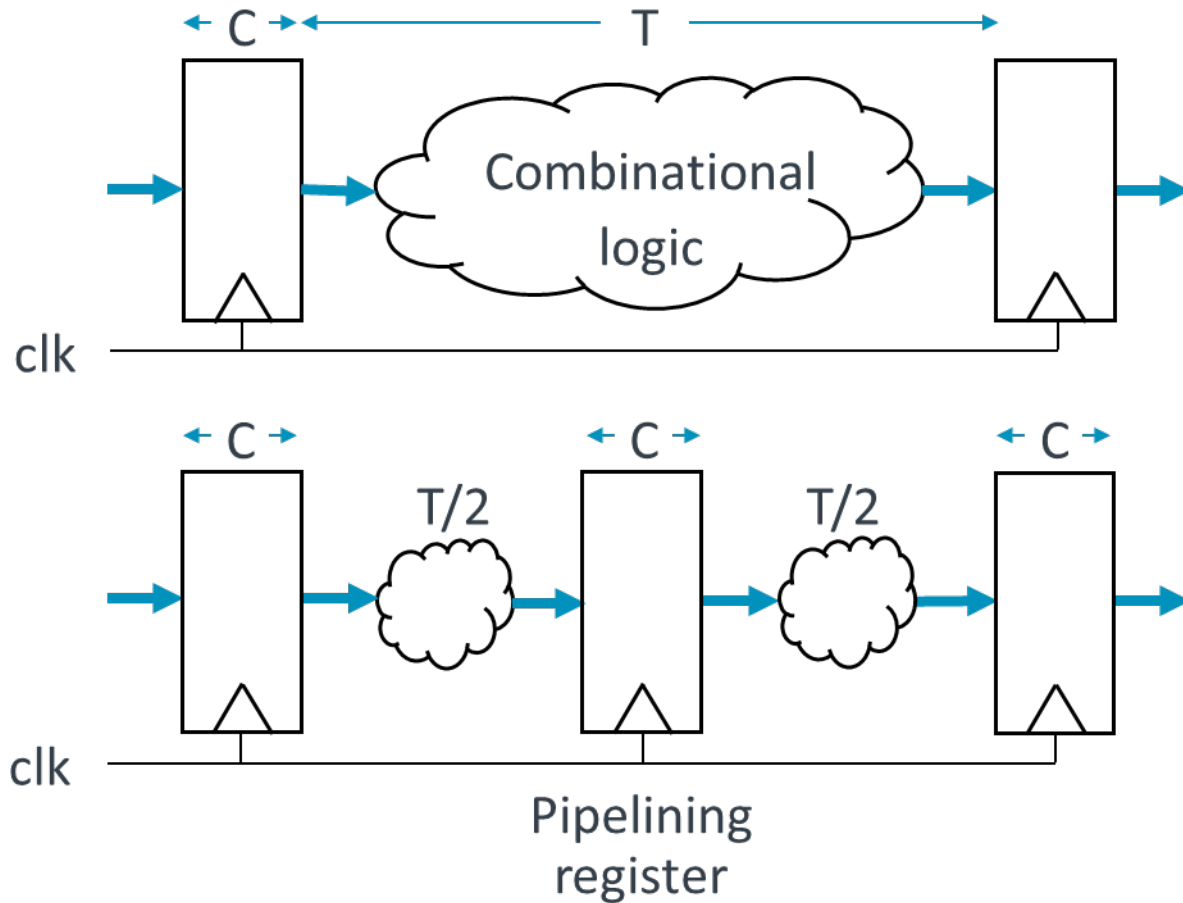


We arrange for the different phases of execution to be overlapped. We aim to exploit “temporal” parallelism.

Pipelining in computer architecture

- Pipelining became an universal technique in 1985
 - Overlaps execution of instructions
 - Exploits “Instruction Level Parallelism”
- Pipelining **doesn't** help latency of single task, it helps **throughput** of entire workload
- Multiple tasks operating simultaneously using different resources
- **“Potential” speedup = Number pipe stages**
- Time to “fill” pipeline and time to “drain” it reduces speedup

Pipelining



Clock period = $T + C$

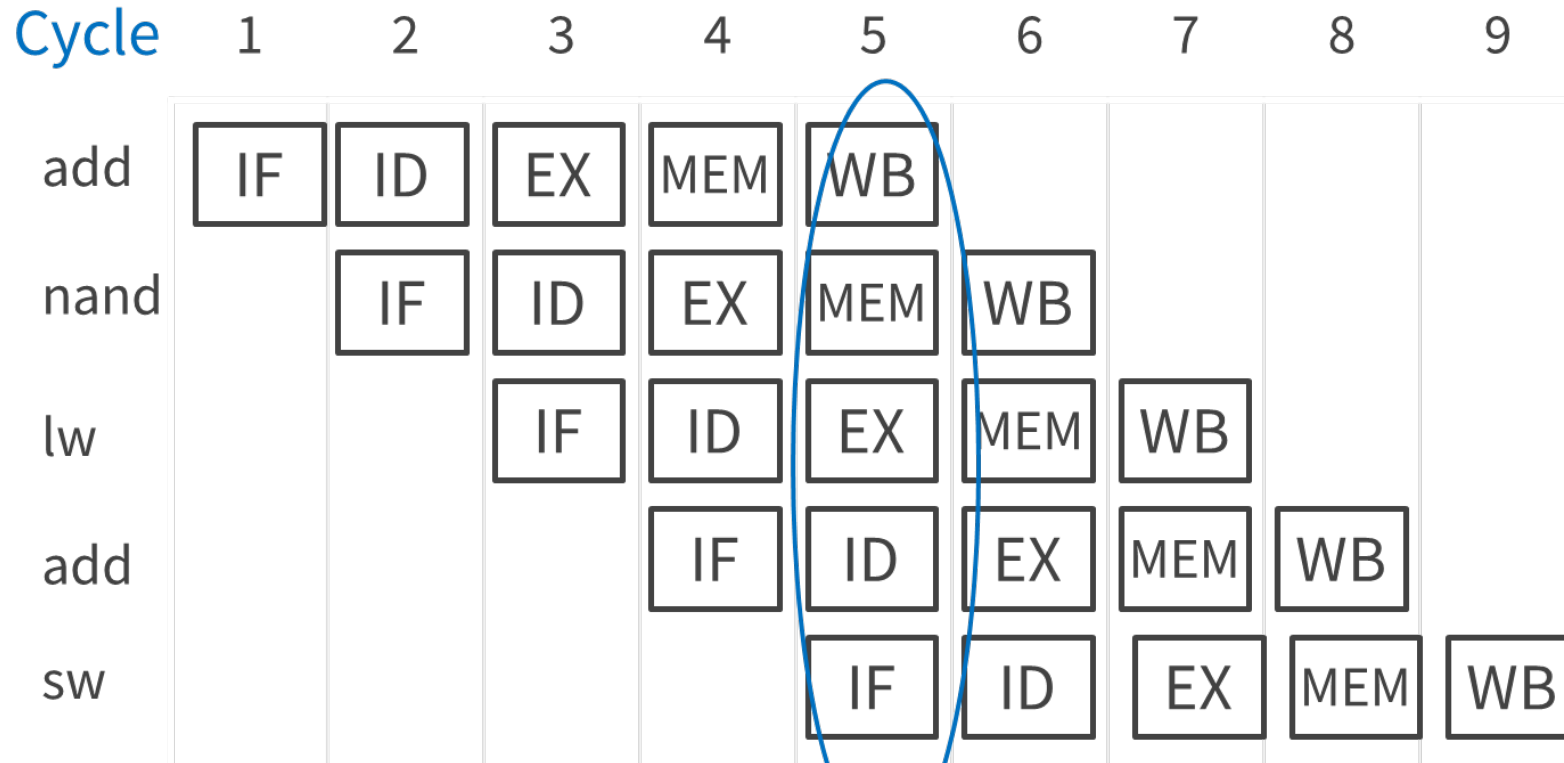
Now, if we create two pipeline stages:

Clock period = $T/2 + C$

If C is small, our clock frequency has almost doubled.

Hence, our **throughput** will double, too.

Pipelined Processor



Latency: 5 cycles

Throughput: 1 insn/cycle

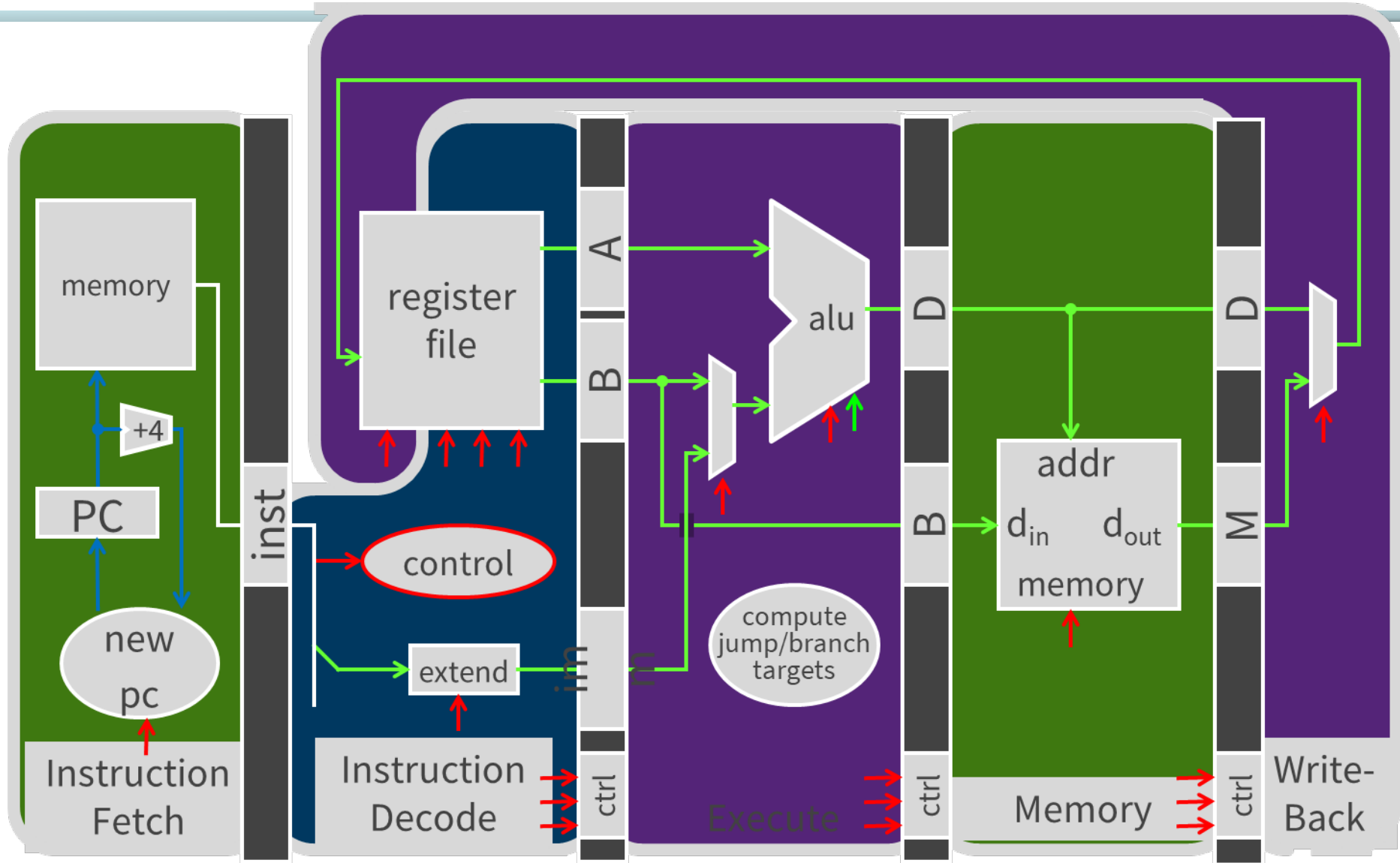
Concurrency: 5

CPI = 1 Why?

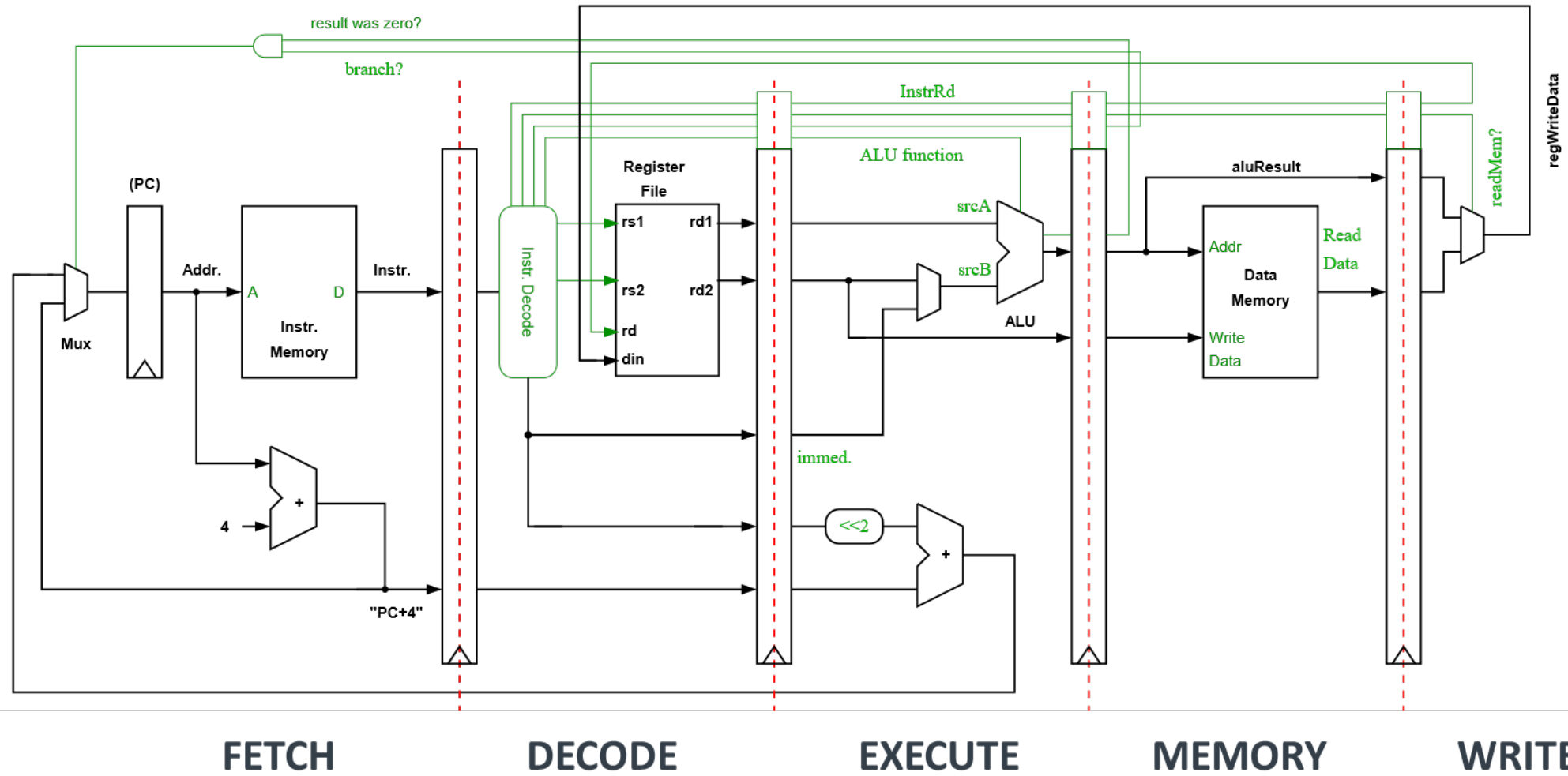
ISA vs. Pipeline

- RISC-V is perfect for pipelining, why?
 - Instructions same length
 - 32 bits, easy to fetch and then decode
 - 4 types of instruction formats
 - Easy to route bits between stages
 - Can read a register source before even knowing what the instruction is
 - Memory access through lw and sw only
 - Access memory after ALU

Pipelined Processor (VE370)



Pipelined Processor (ARM)



FETCH

DECODE

EXECUTE

MEMORY

WRITEBACK

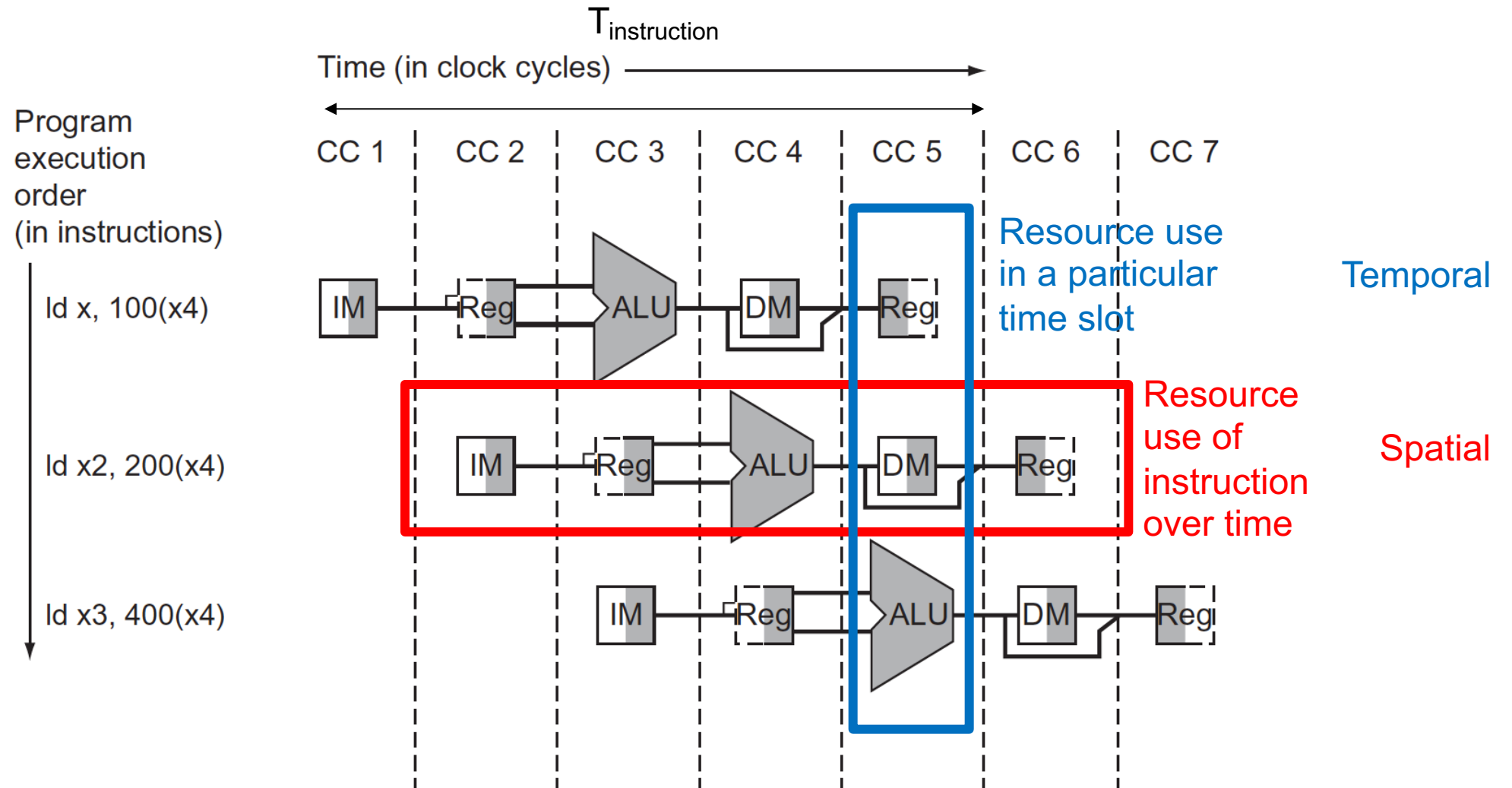
The same principle no matter what ISA!

source: ARM

Pipeline Stages (5-stage)

Stage	Perform Functionality	Latch values of interest
Fetch (IF)	Use PC to index Program Memory, increment PC	Instruction bits (to be decoded) PC + 4 (to compute branch targets)
Decode (ID)	Decode instruction, generate control signals, read register file	Control information, Rd index, immediates, offsets, register values (Ra, Rb), PC+4 (to compute branch targets)
Execute (EXE)	Perform ALU operation Compute targets (PC+4+offset, etc.) in case this is a branch, decide if branch taken	Control information, Rd index, etc. Result of ALU operation, value in case this is a store instruction
Memory (MEM)	Perform load/store if needed, address is ALU result	Control information, Rd index, etc. Result of load, pass result from execute
Writeback (WB)	Select value, write to register file	

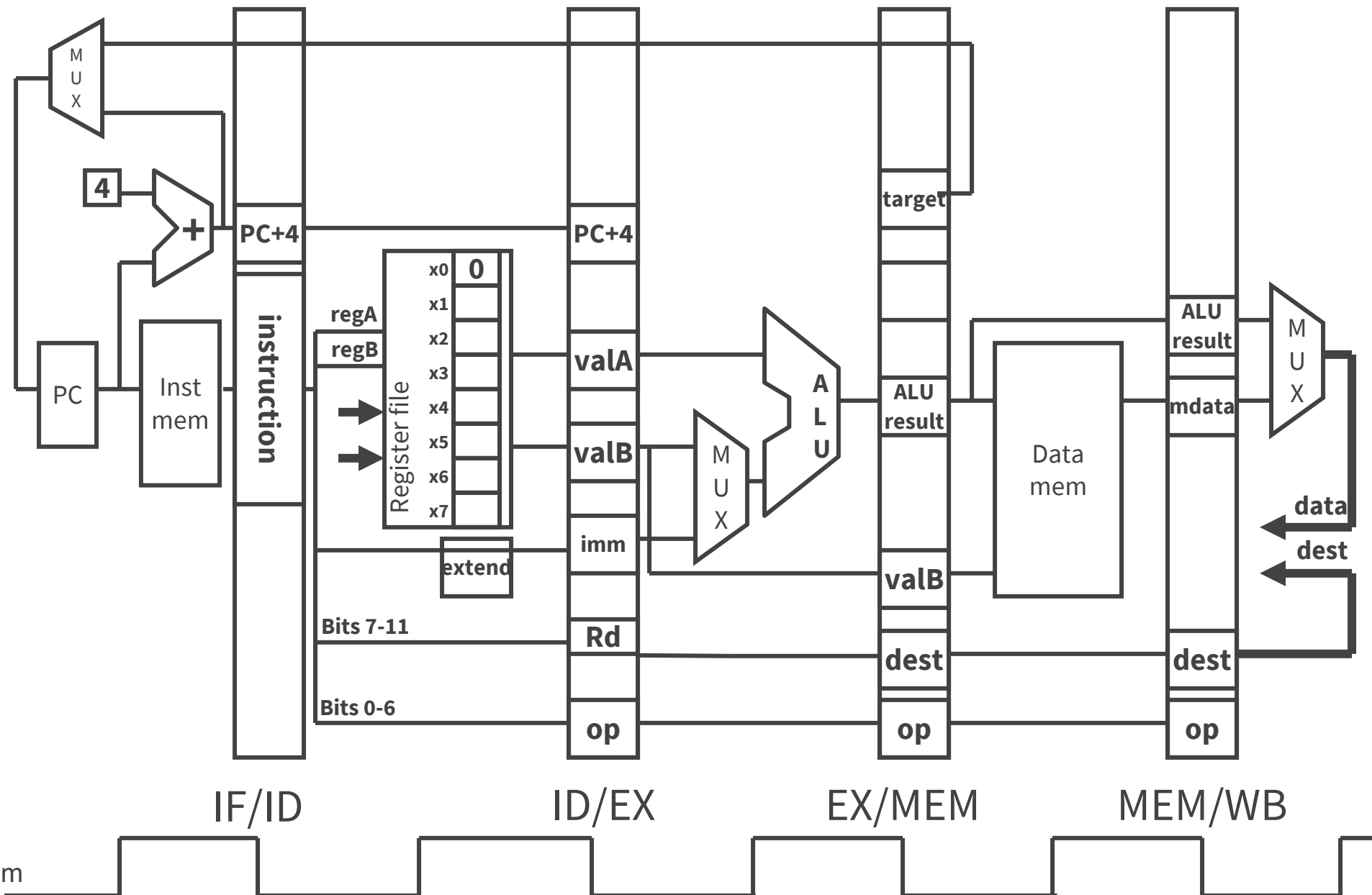
Resource Usage



Example: Sample Code

```
add    x3 ← x1, x2
nand   x6 ← x4, x5
lw     x4 ← x2, 20
add    x5 ← x2, x5
sw     x7 → x3, 12
```

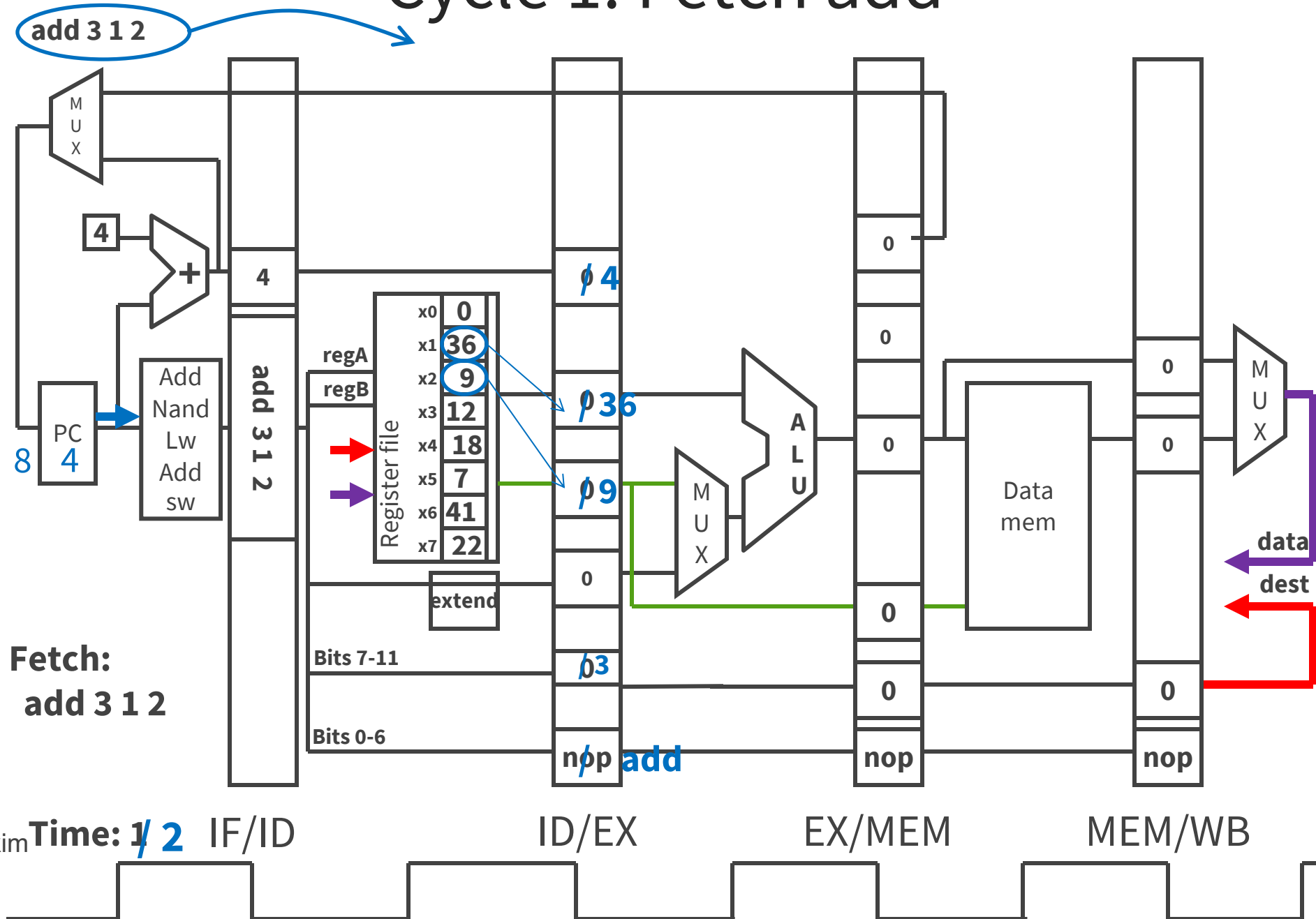
Assume 8-register machine



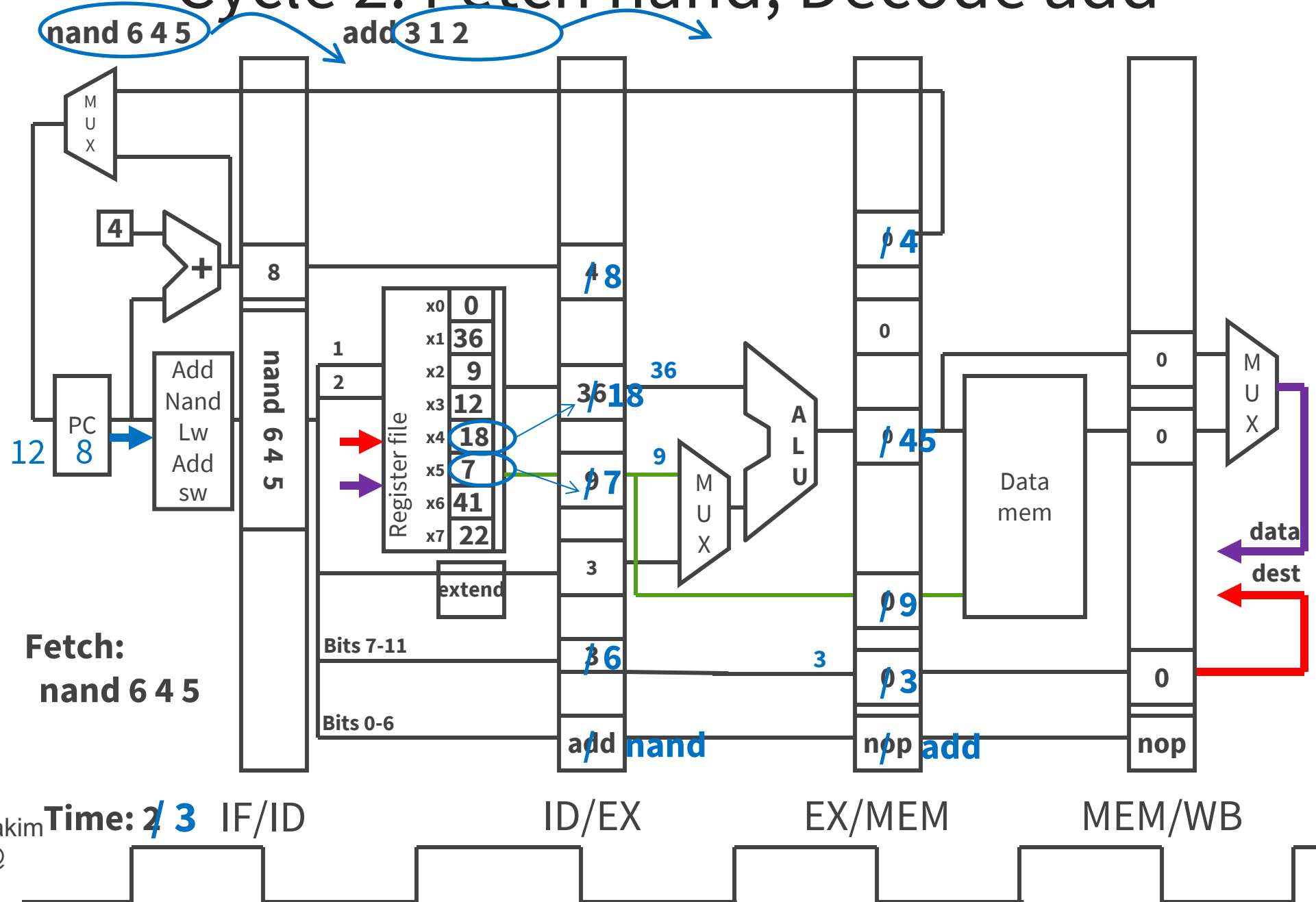
**At time 1,
Fetch
add x3 x1 x2**



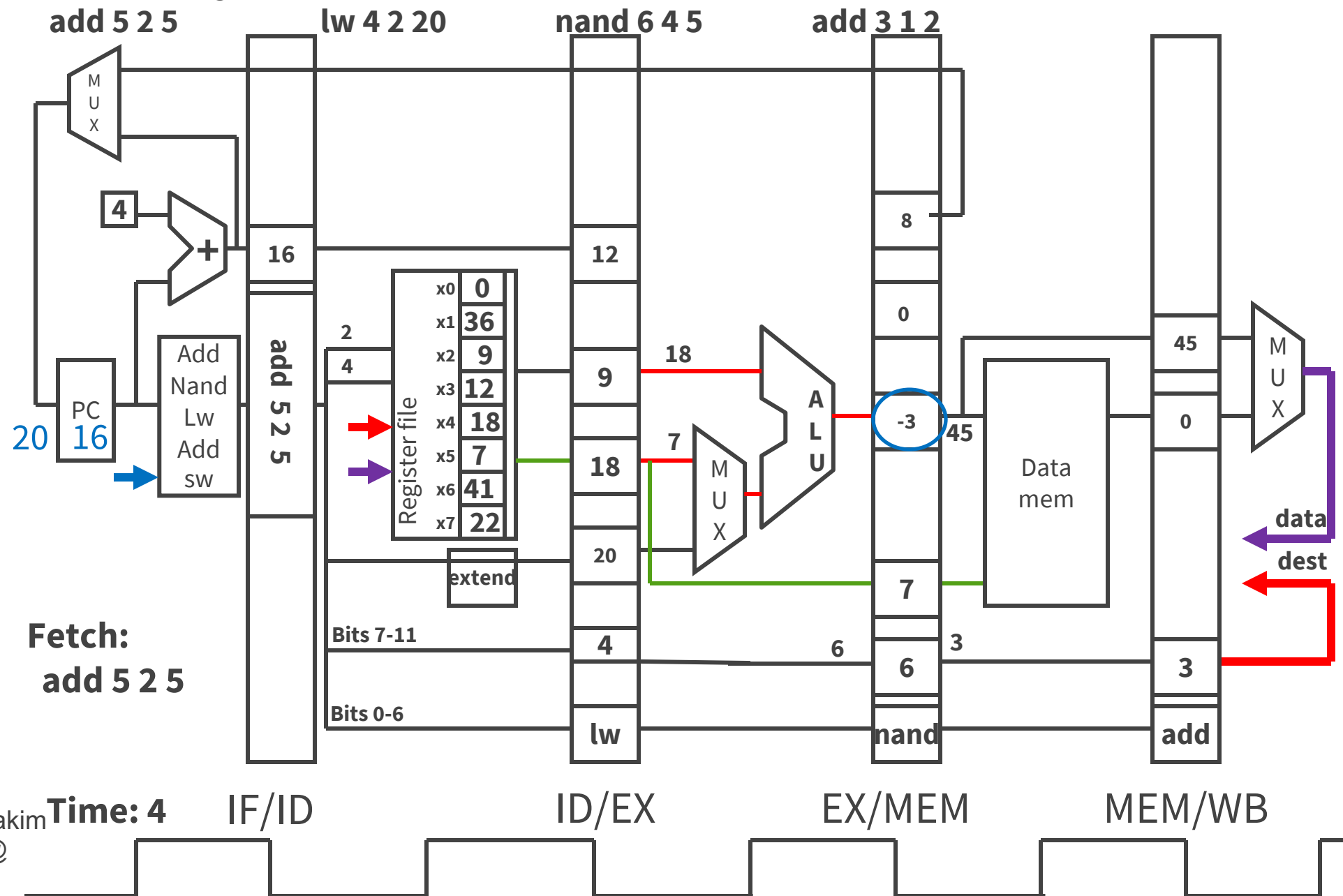
Cycle 1: Fetch add



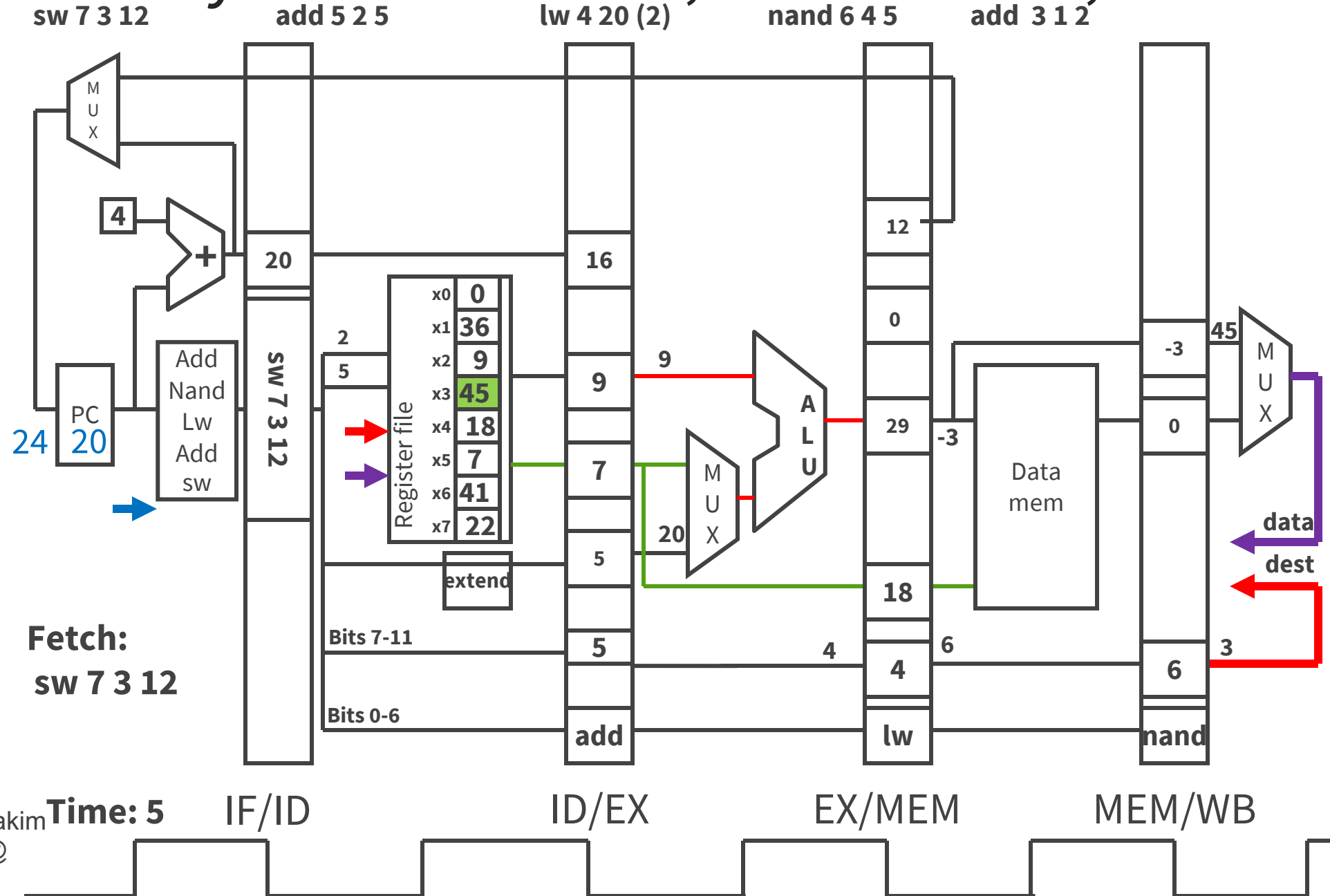
Cycle 2: Fetch nand, Decode add



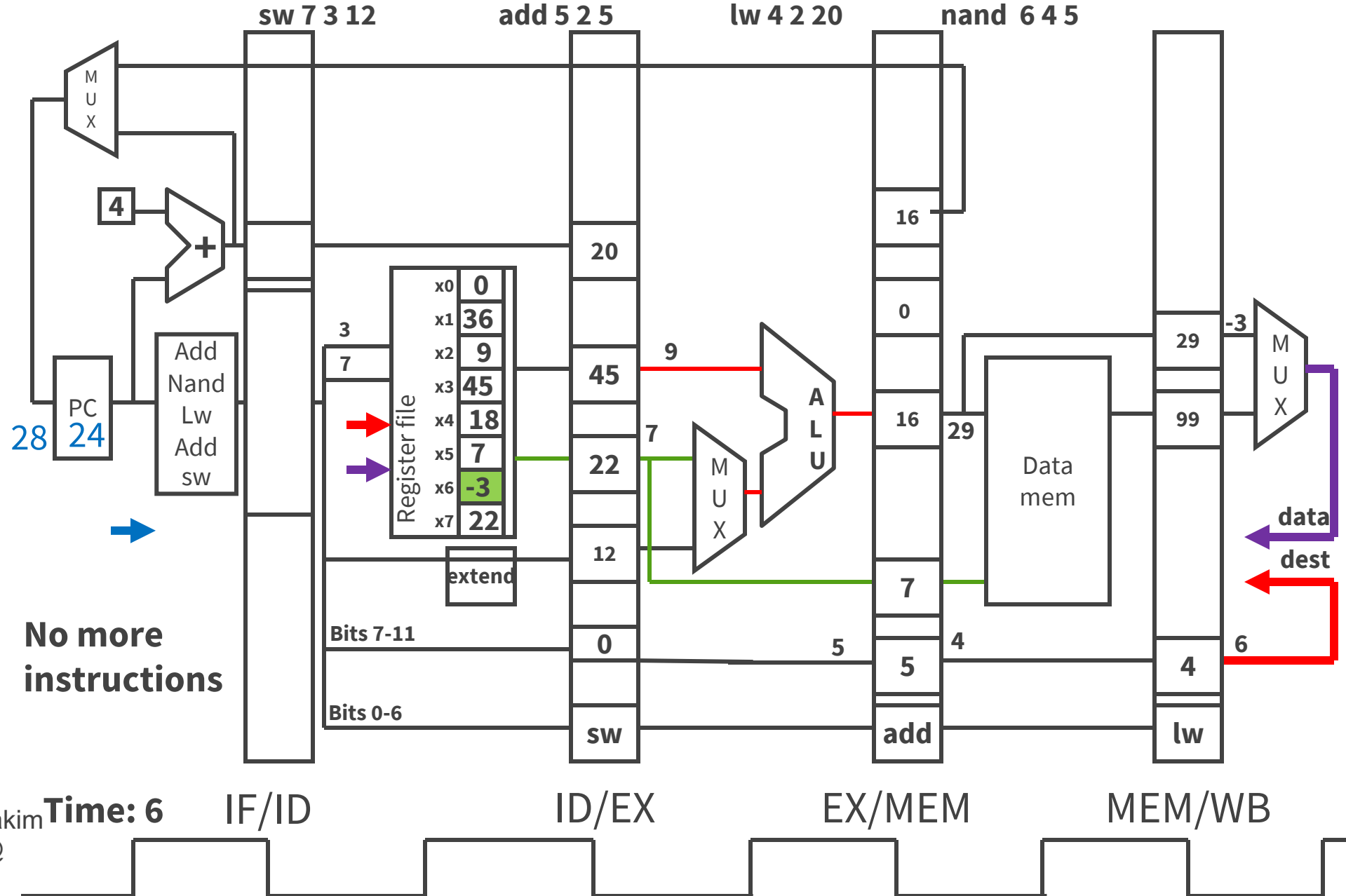
Cycle 4: Fetch add, Decode lw, ...



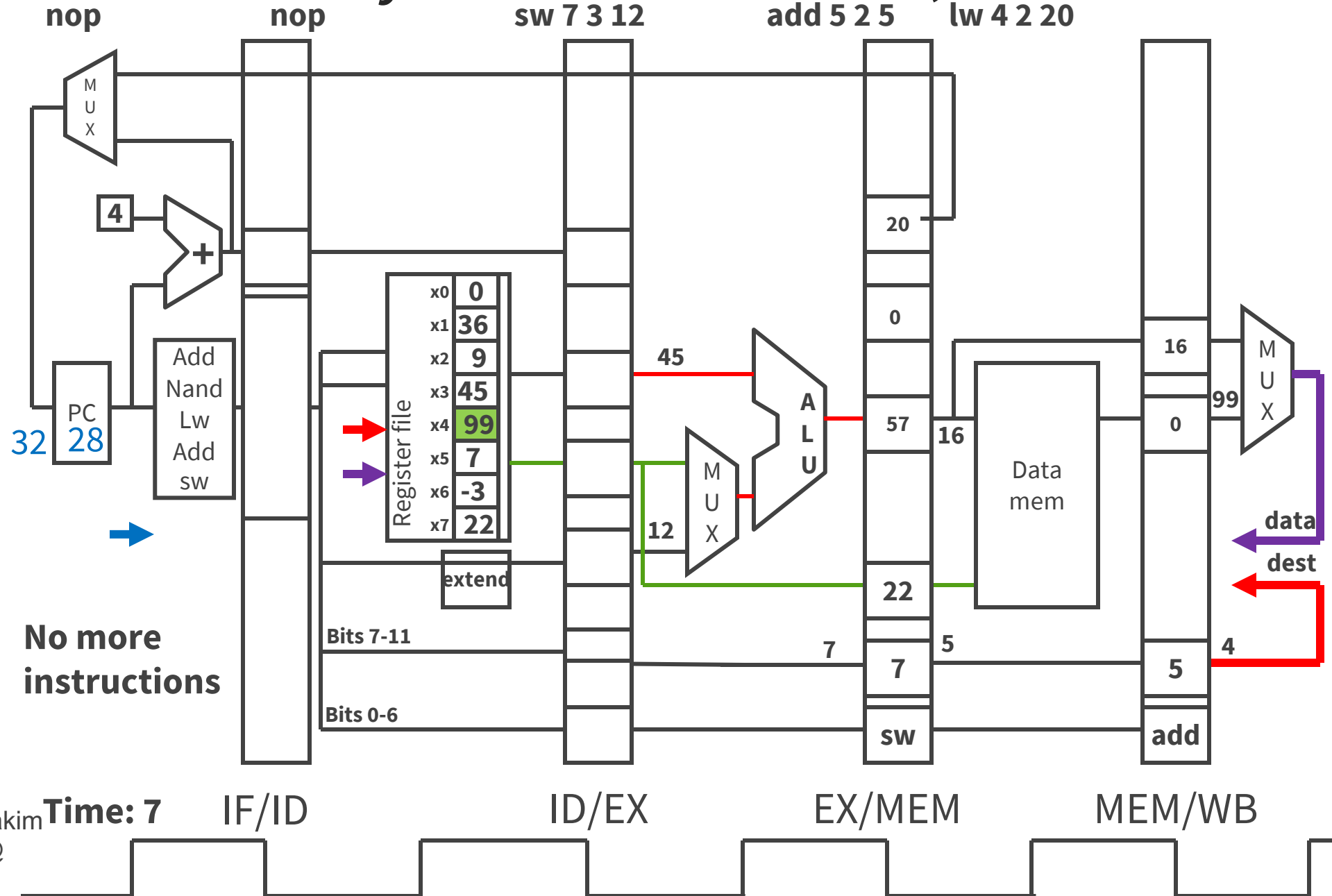
Cycle 5: Fetch sw, Decode add, ...



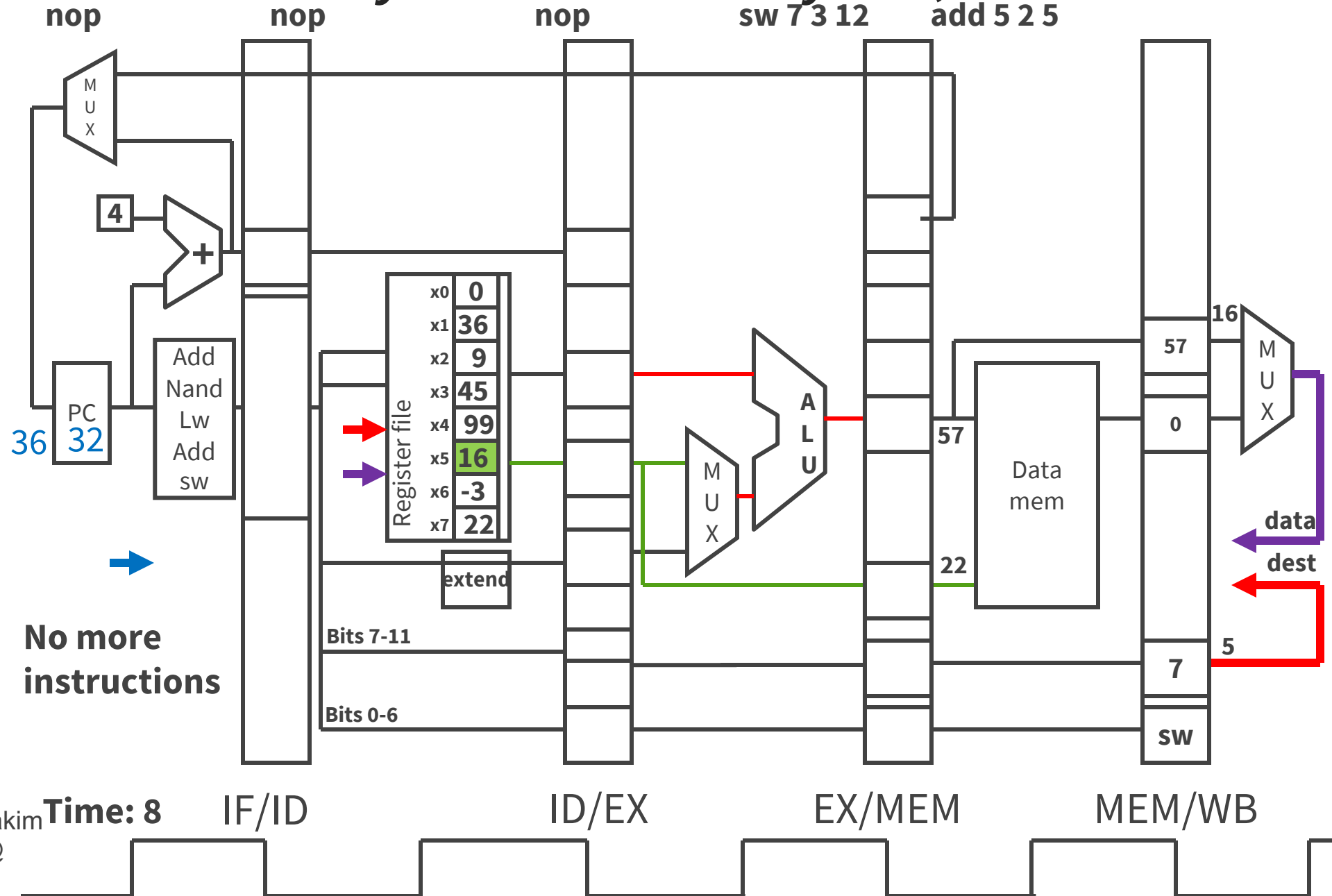
Cycle 6: Decode sw, ...



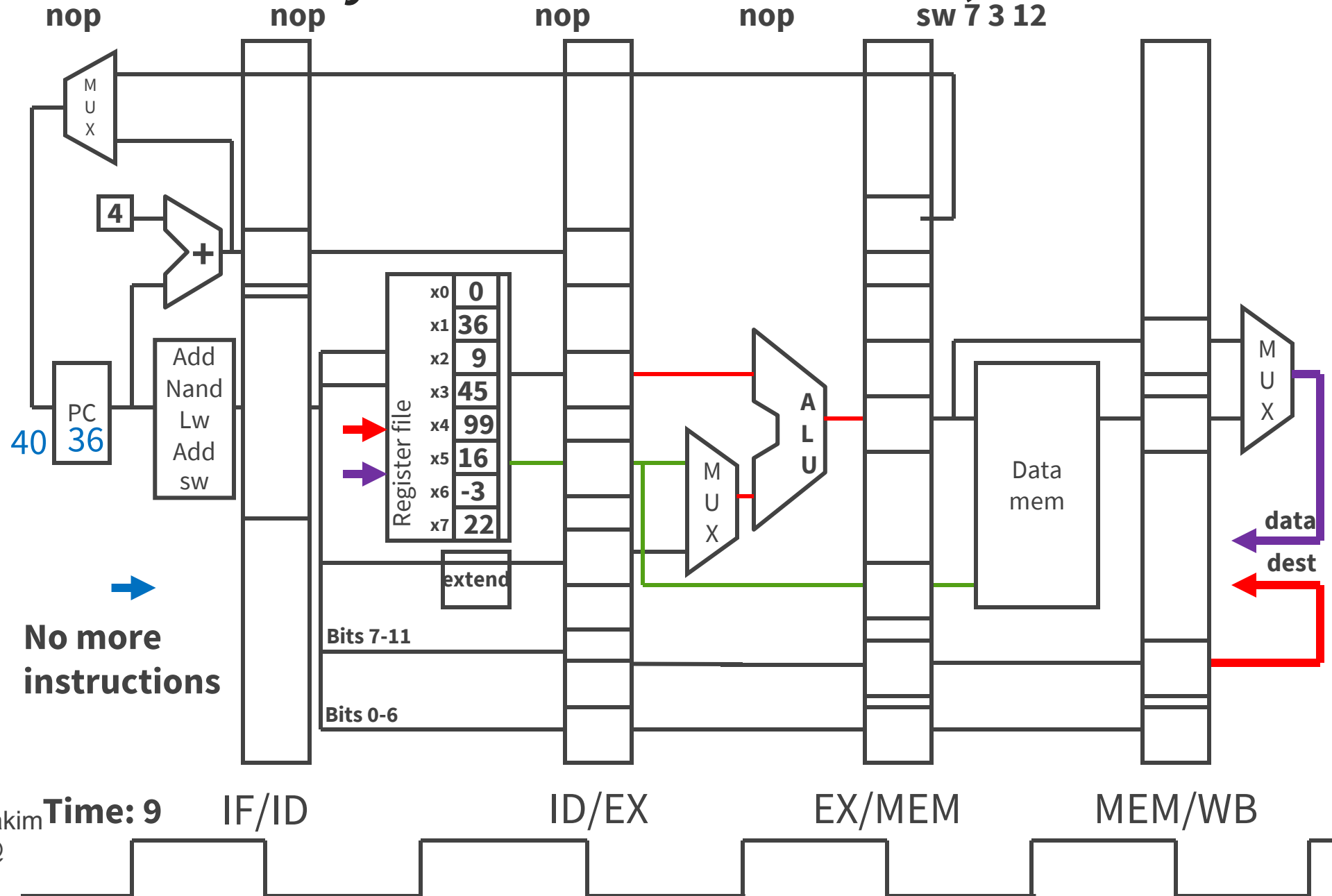
Cycle 7: Execute sw, ...



Cycle 8: Memory sw, ...



Cycle 9: Writeback sw, ...



A quick check

Pipelining is great because:

- A. You can fetch and decode the same instruction at the same time.
- B. You can fetch two instructions at the same time.
- C. You can fetch one instruction while decoding another.
- D. Instructions only need to visit the pipeline stages that they require.
- E. C and D

Principles of Pipelined Implementation

- Break datapath into **multiple** cycles (e.g. 5 in VE370)
 - Parallel execution increases throughput
 - **Balanced pipeline** very important
 - Slowest stage determines clock rate
 - Imbalance kills performance
- Add **pipeline registers (flip-flops)** for isolation
 - Each stage begins by reading values *from* latch
 - Each stage ends by writing values *to* latch
- We will also need to be careful to pipeline our control signals so that decoded control information accompanies each instruction as they progress down the pipeline.
- Resolve hazards

An ideal pipeline

- In the ideal case, when our pipeline never stalls, our CPI will equal 1 (and IPC = 1).
- If we need to stall the pipeline, our CPI will increase, e.g.:
 - If we must stall for 1 cycle for 20% of instructions, and 3 cycles for 5% of instructions, our new CPI would be:

$$\text{Pipeline CPI} = \text{ideal pipeline CPI} + \text{pipeline stalls per instruction}$$
$$= 1 + 1 \times 0.20 + 3 \times 0.05 = 1.35$$

- *Note: to make the best use of pipelining, we should avoid stalling as much as possible. (without increasing our clock period!) Remember:*

$$\text{Time} = \text{instructions executed} \times \text{Clocks Per Instruction (CPI)} \times \text{clock period}$$

Pipelining and Maintaining Correctness

- We can break the execution of instructions into stages and overlap the execution of different instructions.
- We need to ensure that the results produced by our new pipelined processor are no different to the unpipelined one.
- What sort of situations could cause problems once we pipeline our processor? Dependency! Stall!

Stalling to Access Memory

The latency of accessing off-chip memory (DRAM) is typically 10-100 times higher than our pipelined processor's clock period. In order to avoid stalling, we must use on-chip **caches**. (Recall locality)

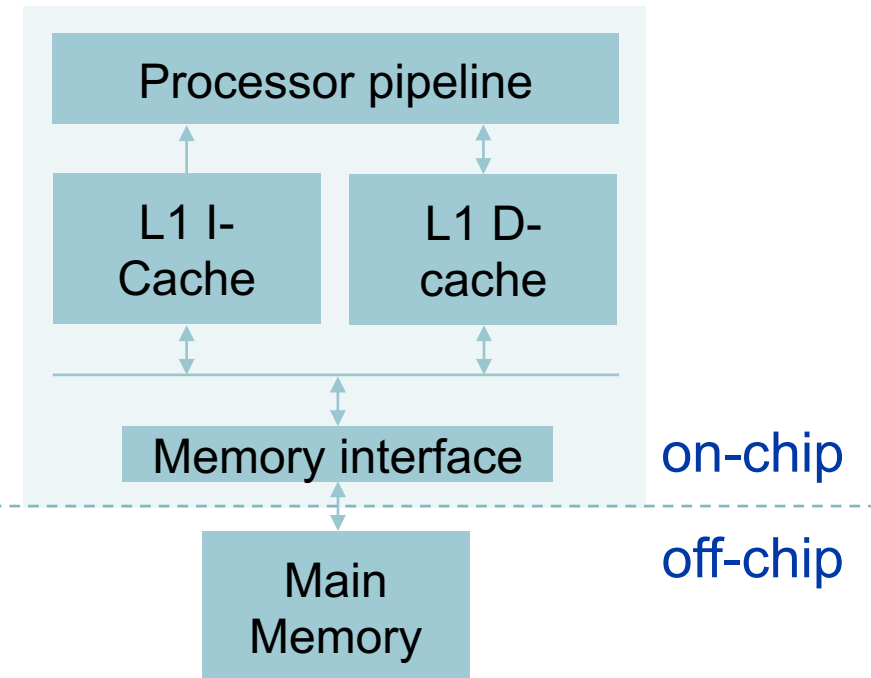
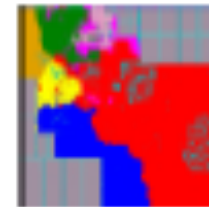
A single Arm Cortex A35 with 8K L1 instruction and data caches, no L2

For a 28 nm process:

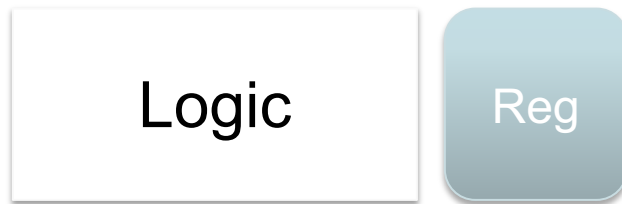
Area: $< 0.4 \text{ mm}^2$

Clock: 1 GHz

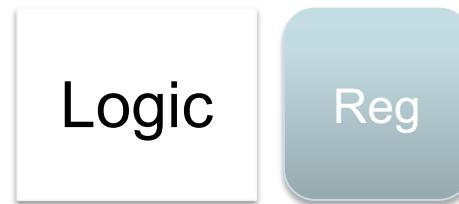
Power: $\sim 90 \text{ mW}$



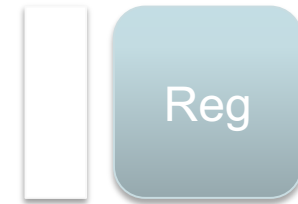
Q: The more pipeline stages, the faster?



Few stages

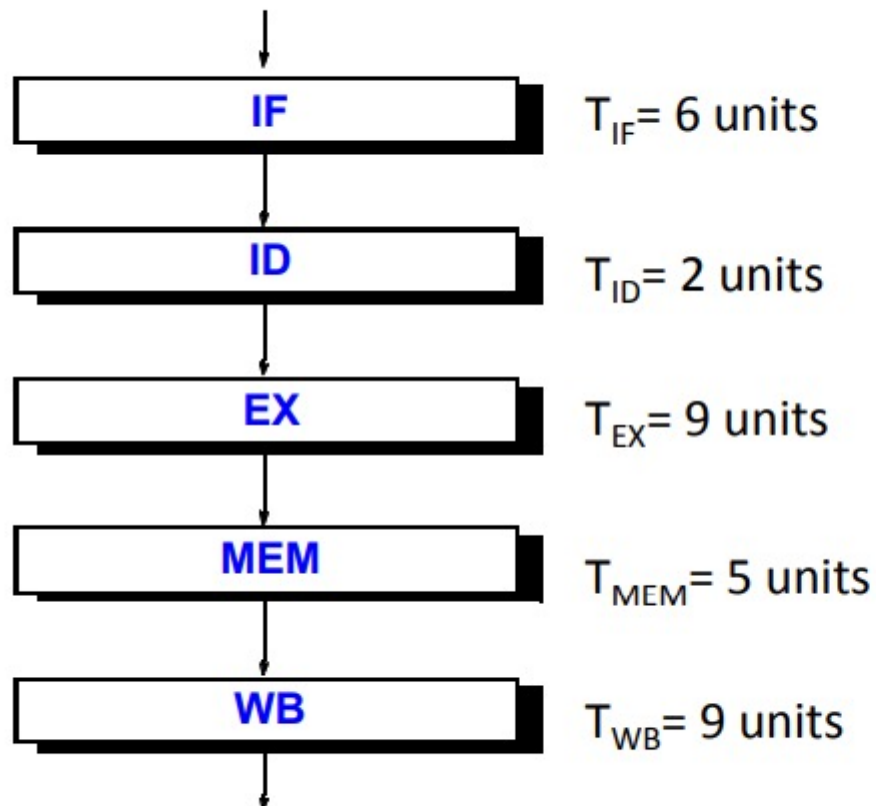


More stages



Even more stages

Balancing pipeline?



Without pipelining

$$T_{\text{cyc}} \approx T_{\text{IF}} + T_{\text{ID}} + T_{\text{EX}} + T_{\text{MEM}} + T_{\text{WB}} \\ = 31$$

Pipelined

$$T_{\text{cyc}} \approx \max\{T_{\text{IF}}, T_{\text{ID}}, T_{\text{EX}}, T_{\text{MEM}}, T_{\text{WB}}\} \\ = 9$$

$$\text{Speedup} = 31 / 9$$

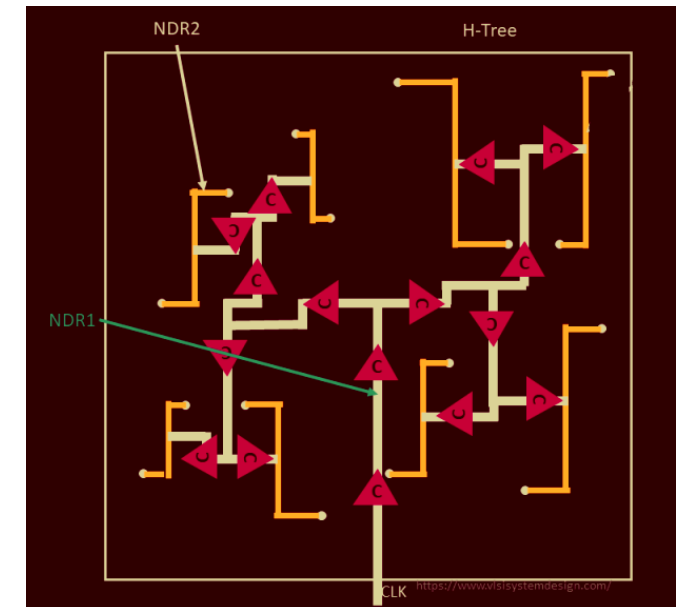
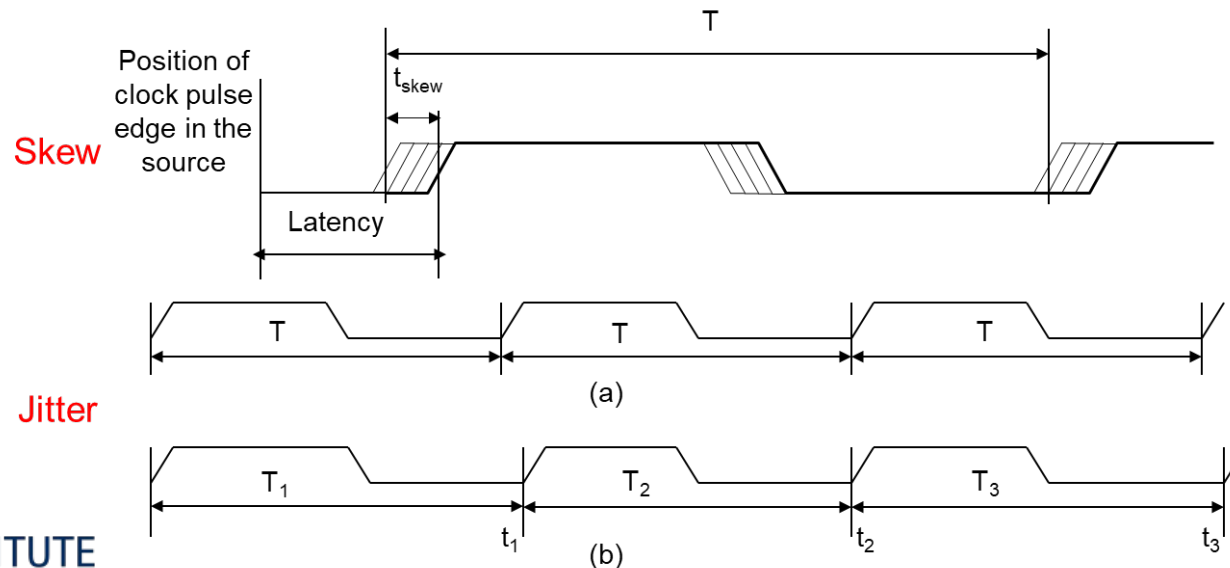
*Can we do better in terms of
either performance or
efficiency?*

Balancing pipeline?

- Two Methods for Stage Quantization:
 - Merging of multiple stages
 - Further subdividing a stage
- Recent Trends:
 - Deeper pipelines (more and more stages)
 - Pipeline depth growing more slowly since Pentium 4. Why?
 - Multiple pipelines (sub pipelines)
 - Pipelined memory/cache accesses (tricky)

Q: The more pipeline stages, the faster?

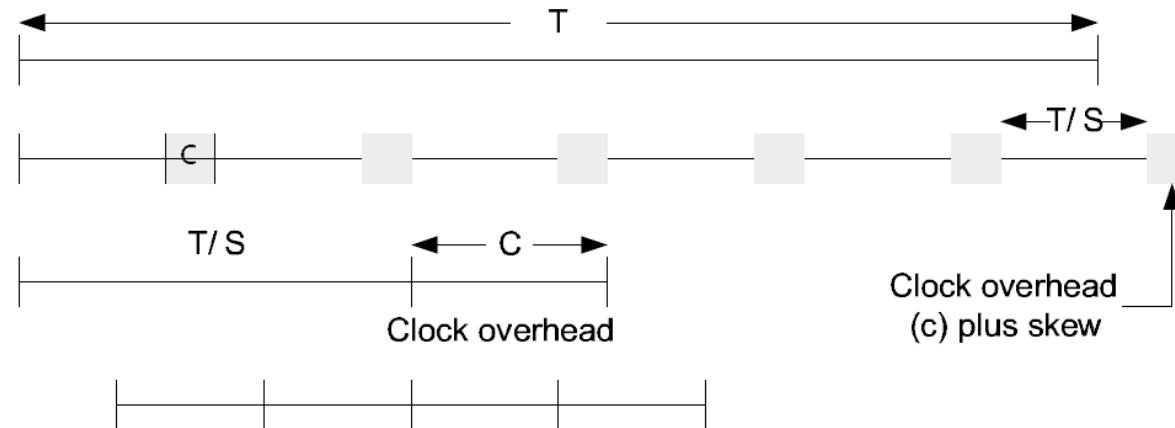
- So far we have assumed pipeline registers are “free”
- The fact
 - Clocks have skews and jitters (can be added to each register)
 - Clock skew = systematic clock edge variation between sites
 - Mainly caused by delay variations introduced by manufacturing variations
 - Random variation
 - Clock jitter = variation in clock edge timing between clock cycles
 - Mainly caused by noise



An Analytical Model of Performance

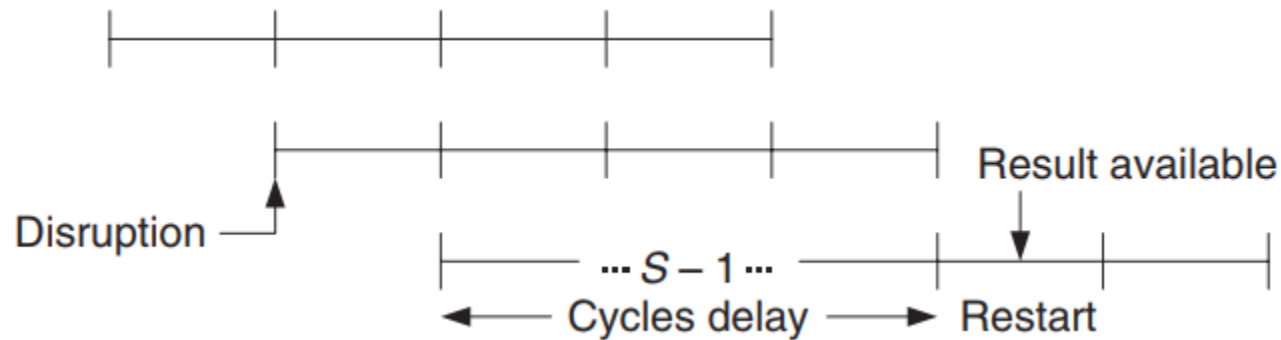
- let the total instruction execution without pipelining and associated clock overhead be T
- in a pipelined processor, let S be the number of segments
 $S - 1$ is number of cycles lost due to a pipeline break
- C = clock overhead incl. fixed skew

each stage carries the
clock overhead
(partitioning overhead)

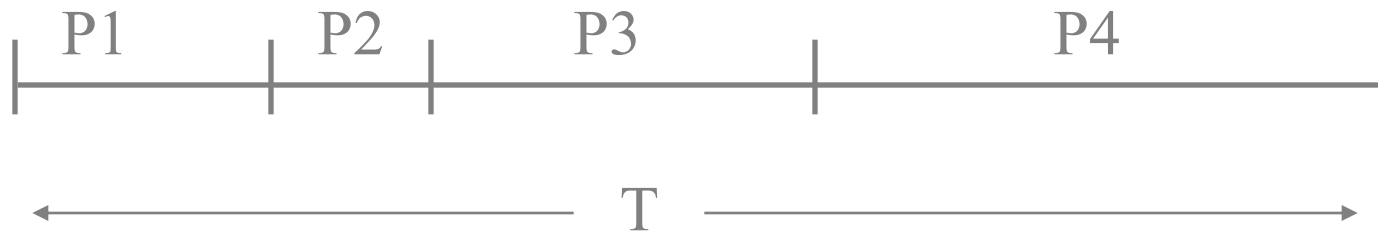


Probability of break (b)

- In an idealized pipelined processor, **if there are no code delays**, it processes instructions at the rate of one per cycle
- But delays can occur (incorrectly guessed or unexpected branches)
- Suppose these interruptions occur with frequency **b** and have the effect of invalidating the $S - 1$ instructions prepared to enter or already in, the pipeline (the worst case)



Optimum pipelining



$P_{\max i}$ = delay of the i th functional unit

suppose $T = \sum_i P_{\max i}$ without clock overhead

S = number of pipeline segments

C = clock overhead

$T/S \geq \max (P_{\max i})$ [quantization]

Cycle
time

Avg. Time
/ segment

Clock
overhead

$$\Delta t = T/S + C$$

$$\text{performance} = 1 / (1 + (S - 1)b) \quad [\text{IPC}]$$

$$\text{throughput} = G = \text{performance} / \Delta t \quad [\text{IPS}]$$

$$G = \left(\frac{1}{1 + (S - 1)b} \right) \times \left(\frac{1}{(T/S) + C} \right)$$

Find S for optimum performance by solving for S:

$$\frac{dG}{dS} = 0$$

$$\text{we get } S_{\text{opt}} = \sqrt{\frac{(1 - b)T}{bC}}$$

Find S_{opt}

- estimate b
 - use instruction traces
- find T and C from design details
 - feasibility studies

$$S_{opt} = \sqrt{\frac{(1 - b) T}{bC}}$$

$$T_{instr} = T + S \times (\text{clocking overhead}) = T + SC, \text{ or } S(T_{seg} + C) = S\Delta t.$$

- example: (Exercise: how to get these numbers?)

b	k	T (ns)	C (ns)	S _{opt}	G (MIPS)	f (MHZ)	CPI	Clock Overhead %
0.1	0.05	15	0.5	16.8	270	697	2.58	34.8%
0.1	0.05	15	1	11.9	206	431	2.09	43.1%
0.2	0.05	15	0.5	11.2	173	525	3.04	26.3%
0.2	0.05	15	1	7.9	140	335	2.39	33.5%

5% of additional cycle time to address quantization errors or other overhead

Quantization + other considerations

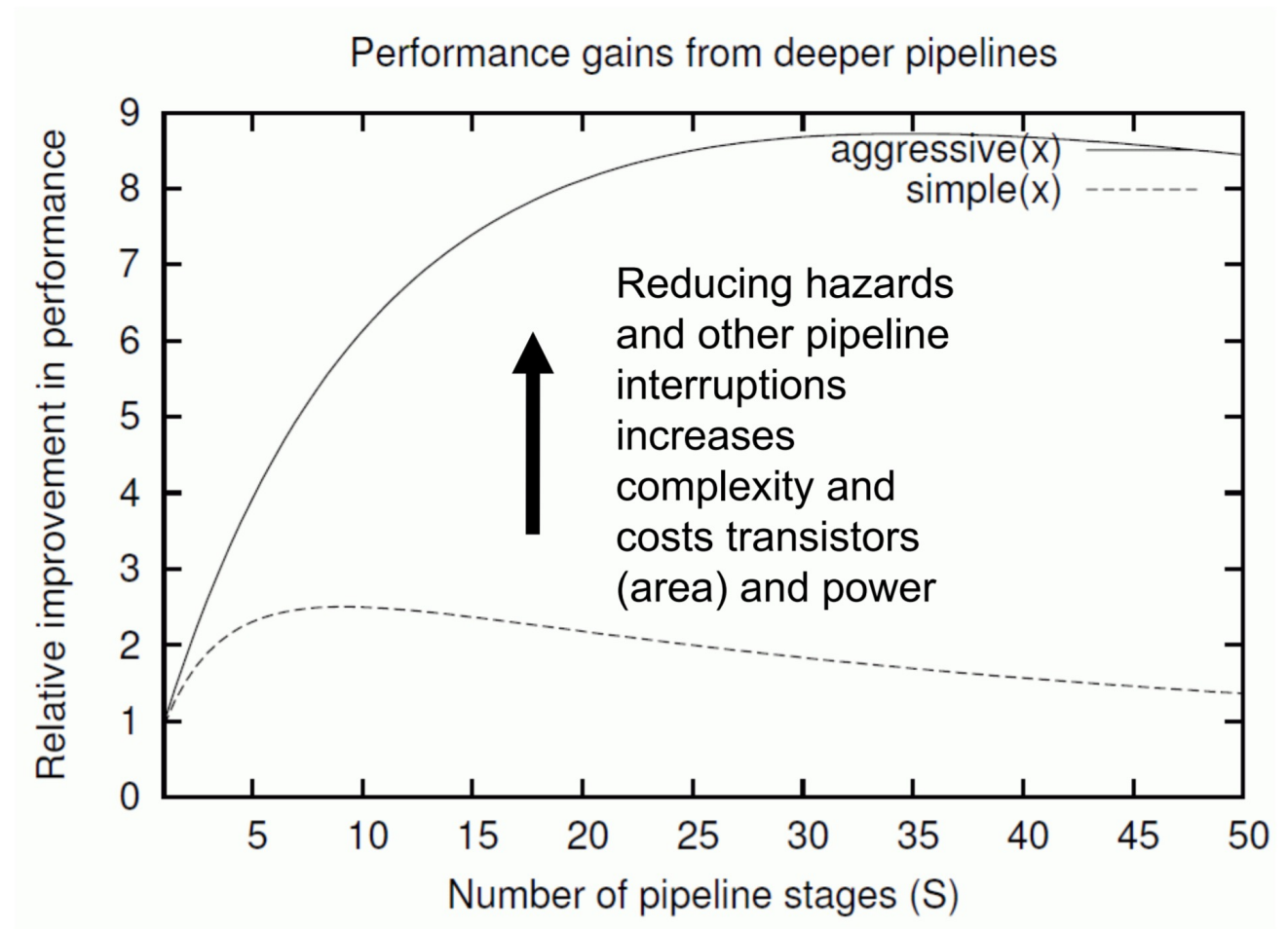
- Quantization effects
 - T cannot be arbitrarily divided into segments
 - segments defined by functional unit delays
 - some segments cannot be divided; others can be divided only at particular boundaries
- Some functional operations are atomic (can't be interrupted)
 - cycle: usually not cross function unit boundary
- S_{opt}
 - ignores cost/area of extra pipeline stages
 - ignores quantization loss
 - largest S to be used

Microprocessor design practice

- tradeoff around design target
- optimal in-order integer RISC: 5-10 stages
 - performance: relatively flat across this range
 - deeper for out-of-order or complex ISA (e.g. Intel Architectures)
- use longer pipeline (higher frequency) if
 - FP/multimedia vector performance important
 - clock overhead low
- else use shorter pipeline
 - especially if area/power/effort are critical

Optimal Pipeline Depth

- $T = 5 \text{ ns}$, penalty of interruption is $(S-1)$
- **Simple pipeline design**
 - $C = 300 \text{ ps}$
 - Pipeline interruption every 6 instructions
- **Aggressive pipeline design**
 - $C = 100 \text{ ps}$
 - Pipeline interruption every 25 instructions



Optimal Pipeline Depth

- Area optimized cores may have 2-3 stages.
- Simple, efficient scalar pipelines are normally implemented with 5-7 stages.
- Higher performance cores, which fetch and issue multiple instructions in a single cycle, may have 8-16 stages.
- Pipeline lengths for general-purpose processors peaked at 31 stages with Intel's Pentium 4 in 2004. Why have they reduced since 2004 instead of increasing? (hint: hazard)

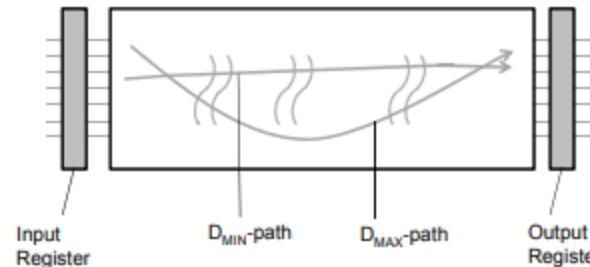
Performance

- High clock rates with small pipeline segments may (or may not) produce better performance.
- Two basic factors enabling clock rate advances:
 - Increased control over clock overhead.
 - Increased number of segments in the pipelines.
- **Low clock overhead (small C) may cause higher pipeline segmentation, but performance does not correspondingly improve unless we also decrease pipeline disruption, b**

$$S_{\text{opt}} = \sqrt{\frac{(1 - b) T}{bC}}$$

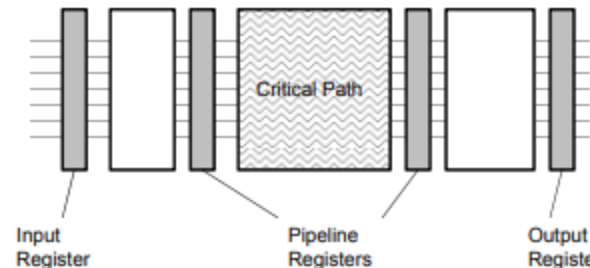
Other pipelining techniques?

- Asynchronous or self-timed clocking
 - avoids clock distribution problems, but has its own overhead
- Multi-phase domino clocking
 - skew tolerant and low clock overhead; lots of power required and extra area
- Wave pipelining
 - ultimate limit on Δt
$$\Delta t \geq P_{\max} - P_{\min} + C$$



(a)

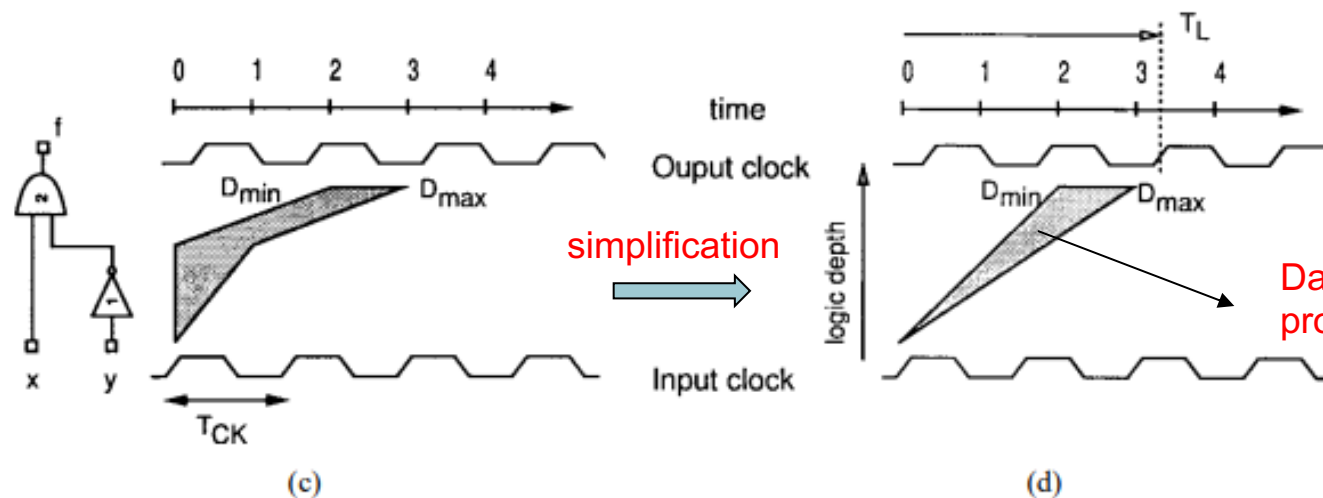
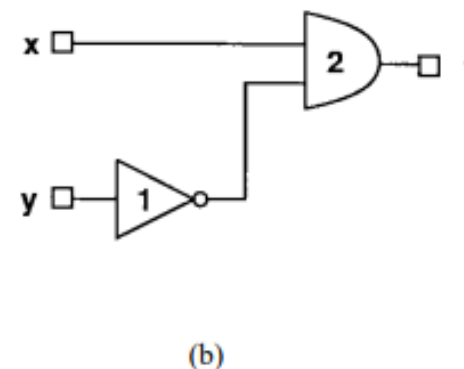
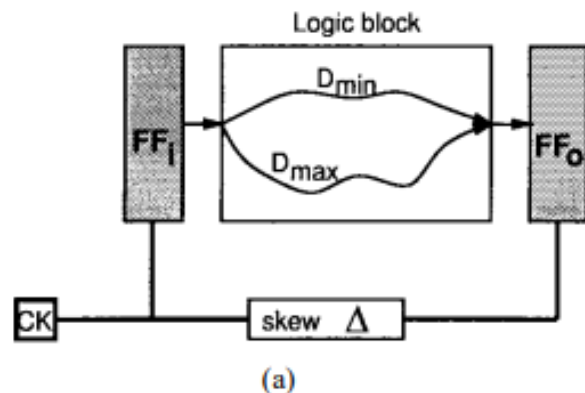
Wave-Pipeline



Conventional Pipeline

The original paper is available on Canvas.

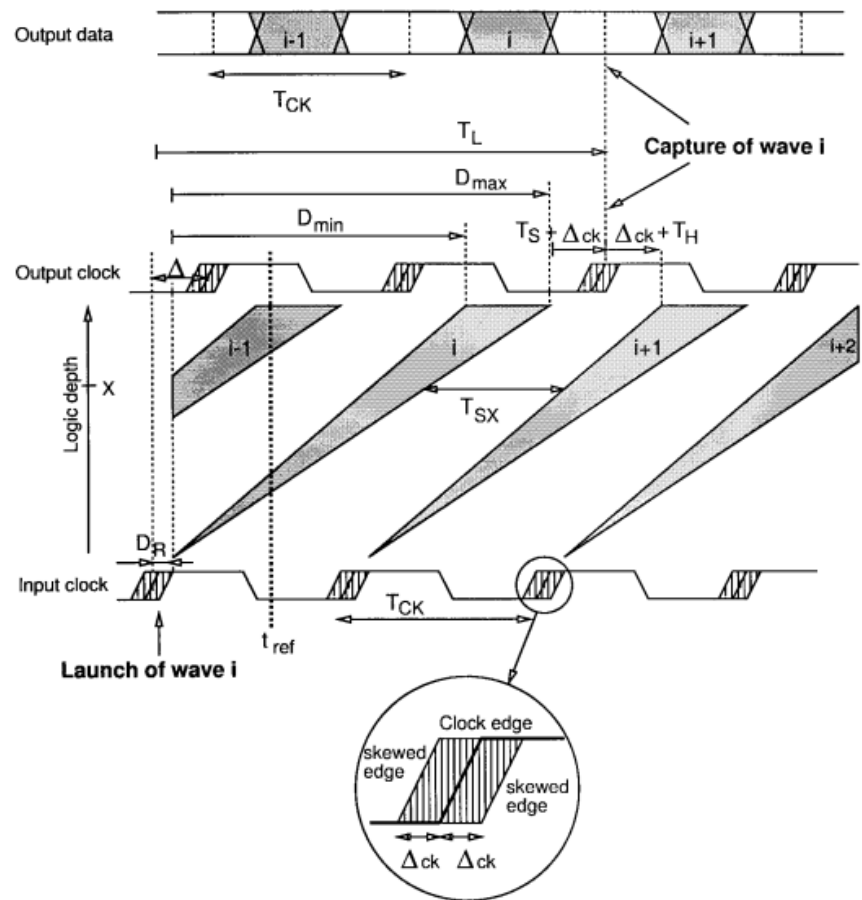
Wave pipelining



Taken from VE481, 2021 FA

Wave pipelining

- output data is clocked after the latest data has arrived at the outputs and before the earliest data from the next clock cycle arrives at the outputs.



$$T_L > D_R + D_{MAX} + T_S + \Delta_{CK}$$

$$T_L < T_{CK} + D_R + D_{MIN} - (\Delta_{CK} + T_H).$$


$$T_{CK} > (D_{MAX} - D_{MIN}) + T_S + T_H + 2\Delta_{CK}$$

clock overhead

Pipeline Summary

- Pipelining is often an effective and efficient way to improve performance. We must be careful that gains from a faster clock are not lost due to the cost of stalling the pipeline.
- Pipelining registers are not free!
- We must take care of Hazards!

What is hazard?

- Dependence: relationship between two instructions – Not a bad thing
- Hazards 
 - A situation that prevents starting the next instruction in the next clock cycle
 - Correctness problems associated w/ processor design
 - Hazards are a bad thing: stalls reduce performance (recall CPI) or complicate the processor design

Hazards

- Structural hazards
 - Same resource needed for different purposes at the same time (Possible: ALU, Register File, Memory)
 - Why do we have IM and DM separately?
- Data hazards (Both memory and register)
 - Instruction output needed before it's available
 - Need to wait for previous instruction to complete its data read/write
- Control hazards (Procedural Dependence)
 - Flow of execution depends on previous instruction
 - One instruction affects whether another executes at all

A quick quiz

Which of the following statements is true?

- A. Whether there is a data **dependence** between two instructions depends on the machine the program is running on.
- B. Whether there is a data **hazard** between two instructions depends on the machine the program is running on.
- C. Both A & B
- D. Neither A nor B

Types of Data hazards

- Consider executing a sequence of register-register instructions of type:

Data dependence

$x3 \leftarrow x1 \text{ op } x2$ **Read-after-Write (RAW)**
...
 $x5 \leftarrow x3 \text{ op } x4$

Anti-dependence

$x3 \leftarrow x1 \text{ op } x2$ **Write-after-Read (WAR)**
...
 $x1 \leftarrow x4 \text{ op } x5$

Output-dependence

$x3 \leftarrow x1 \text{ op } x2$ **Write-after-Write (WAW)**
...
 $x3 \leftarrow x6 \text{ op } x7$

Note that the RAR (read after read) case is not a hazard.

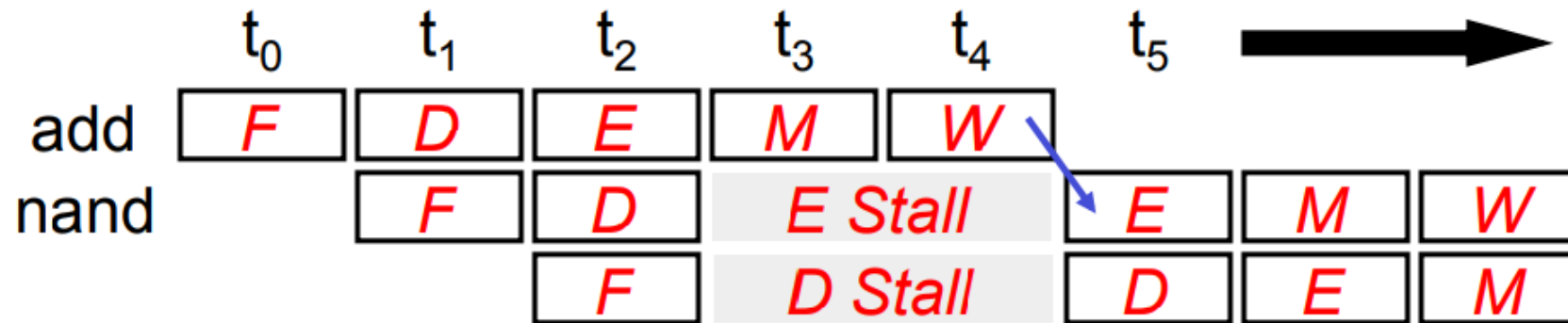
The Cost of Deeper Pipelines

- Instruction pipelines are not ideal

- Assume

add 3 1 2
nand 5 3 4

RAW!!



Terminology

- Hazard Resolution

- Static Method: Performed at compiled Time in software
- Dynamic Method: Performed at run Time using hardware

- **Interlock**

- Hardware mechanisms for dynamic hazard resolution
- Wait for hazard to clear by holding dependent instructions in issue stage

Handling Data Hazards

- Avoidance (static)
 - Make sure that there are no hazards in the code
- Interlock/stall (dynamic)
 - Stall until earlier instructions finish
- forward/bypass (dynamic)
 - Resolve hazard earlier by bypassing value as soon as available
- Second and third require detection

Avoidance (static)

- Programmer/compiler must know implementation details
 - Insert nops/bubbles between dependent instructions

```
add    1  2  3    ← write R3 in cycle 5
nop
nop
nand   3  4  5    ← read R3 in cycle 6
```


Write after write hazard

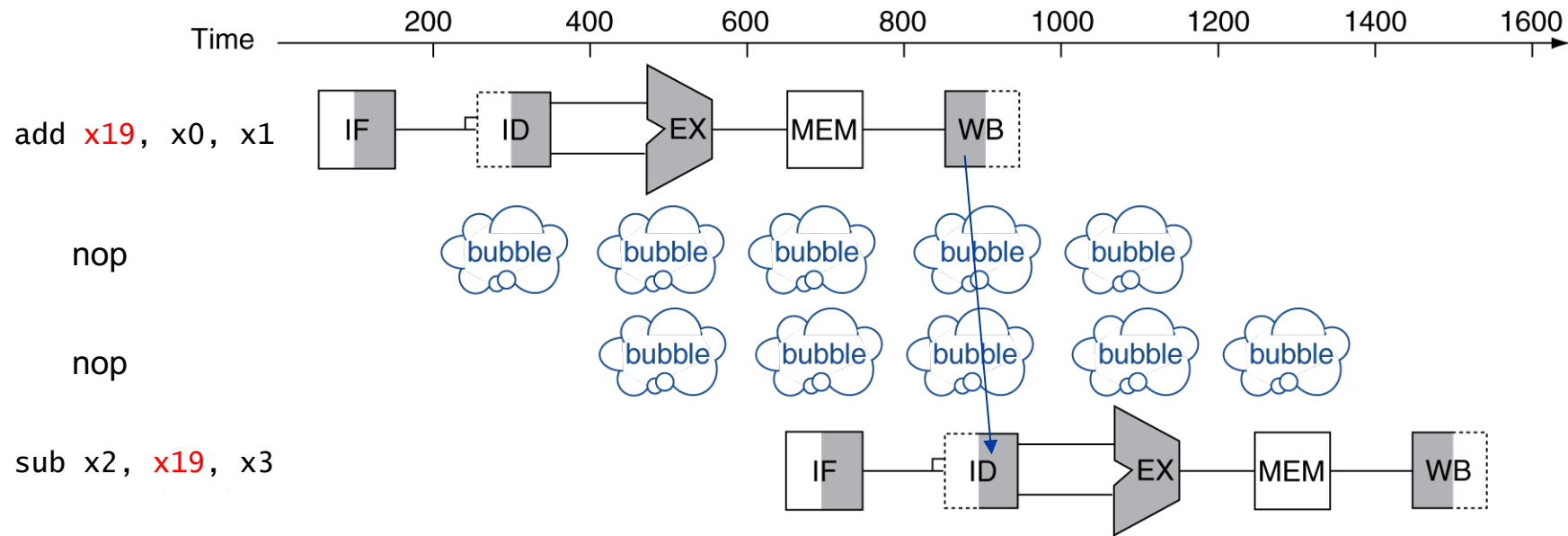
- Can this happen? If so, how?
 - Can't happen in vanilla pipeline (register writes are in order)
 - Can happen: multi-cycle operations (e.g., FP ops, cache misses)
 - `div r1, r3, r4`
 - ...
 - `add r1, r10, r5`

Any problems?

- Code size
 - Higher instruction cache footprint
 - Longer binary load times
 - Worse in machines that execute multiple instructions / cycle
 - Intel Itanium – 25-40% of instructions are nops
- Slower execution
 - CPI=1, but many instructions are nops

Interlock/Stall (dynamic)

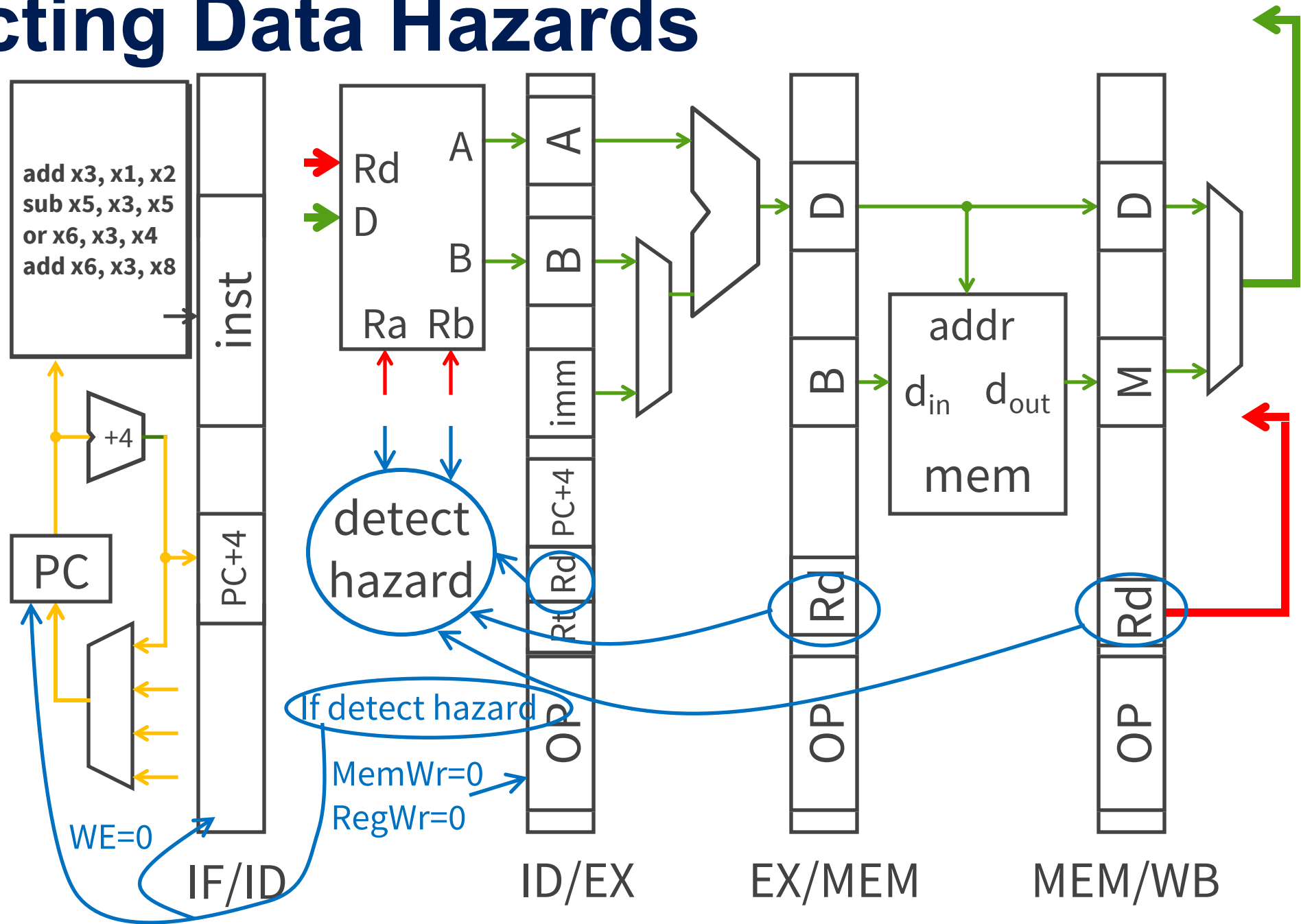
- Pause current and subsequent instructions till safe
 - By inserting **bubbles** or **stalls**
nop # no operation
 # machine code: 0x00000000



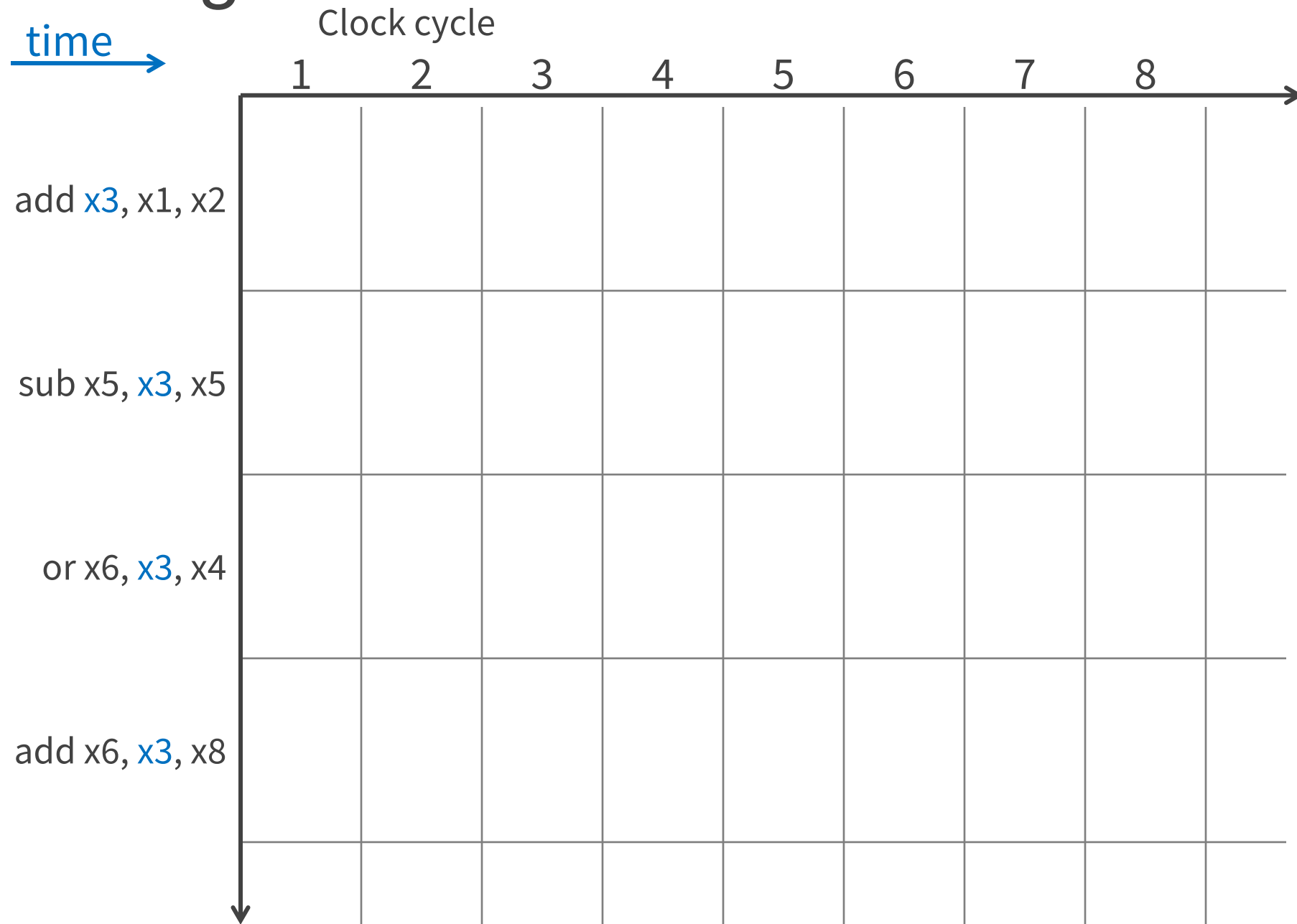
Interlock/Stall

- How to stall an instruction in ID stage
 - prevent IF/ID pipeline register update
 - stalls the ID stage instruction
 - convert ID stage instr into **nop** for later stages
 - innocuous “bubble” passes through pipeline
 - prevent PC update
 - stalls the next (IF stage) instruction
- How to detect?
 - Compare regA & regB with DestReg of preceding insn
 - It is a 3 bit comparator

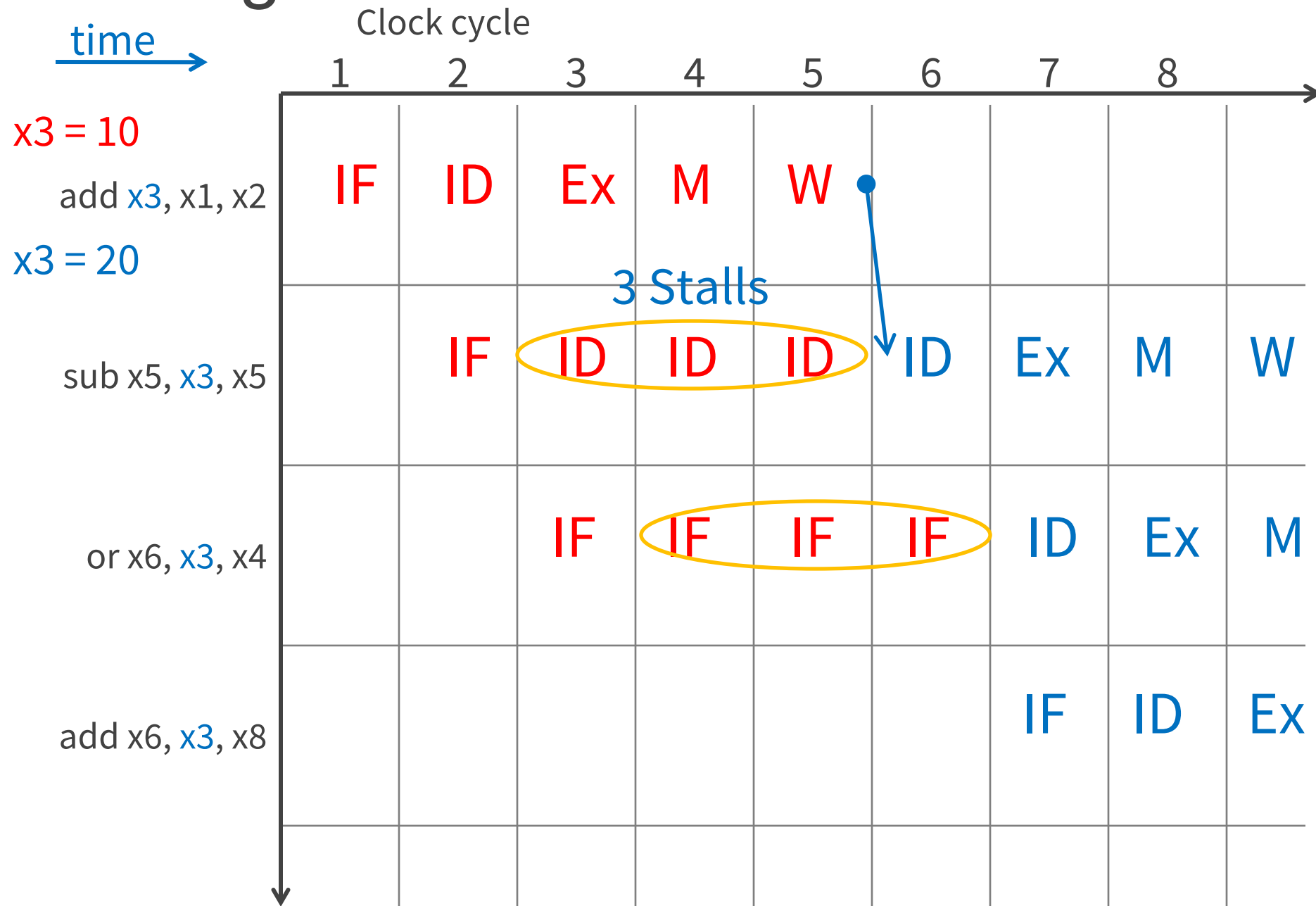
Detecting Data Hazards



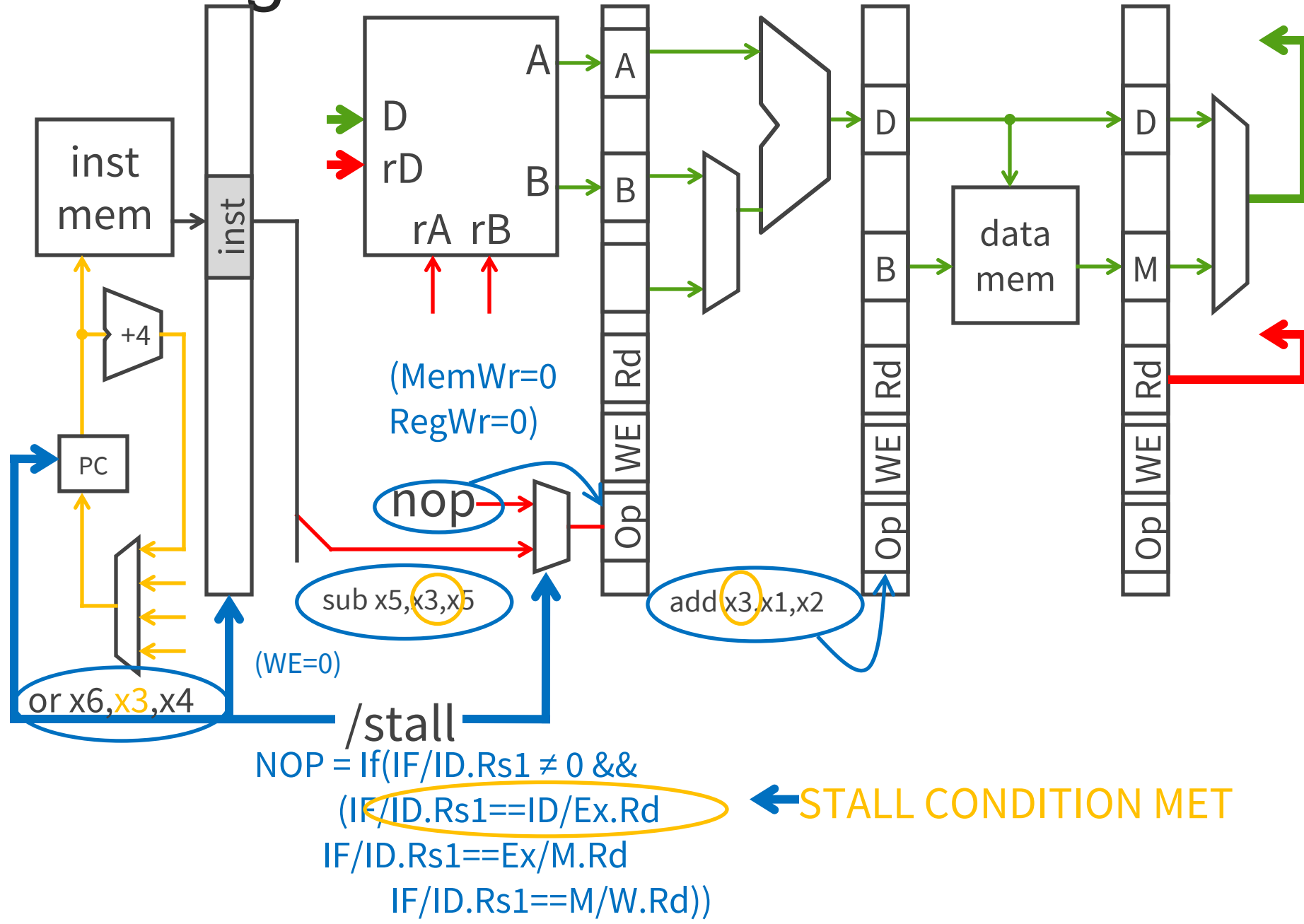
Stalling



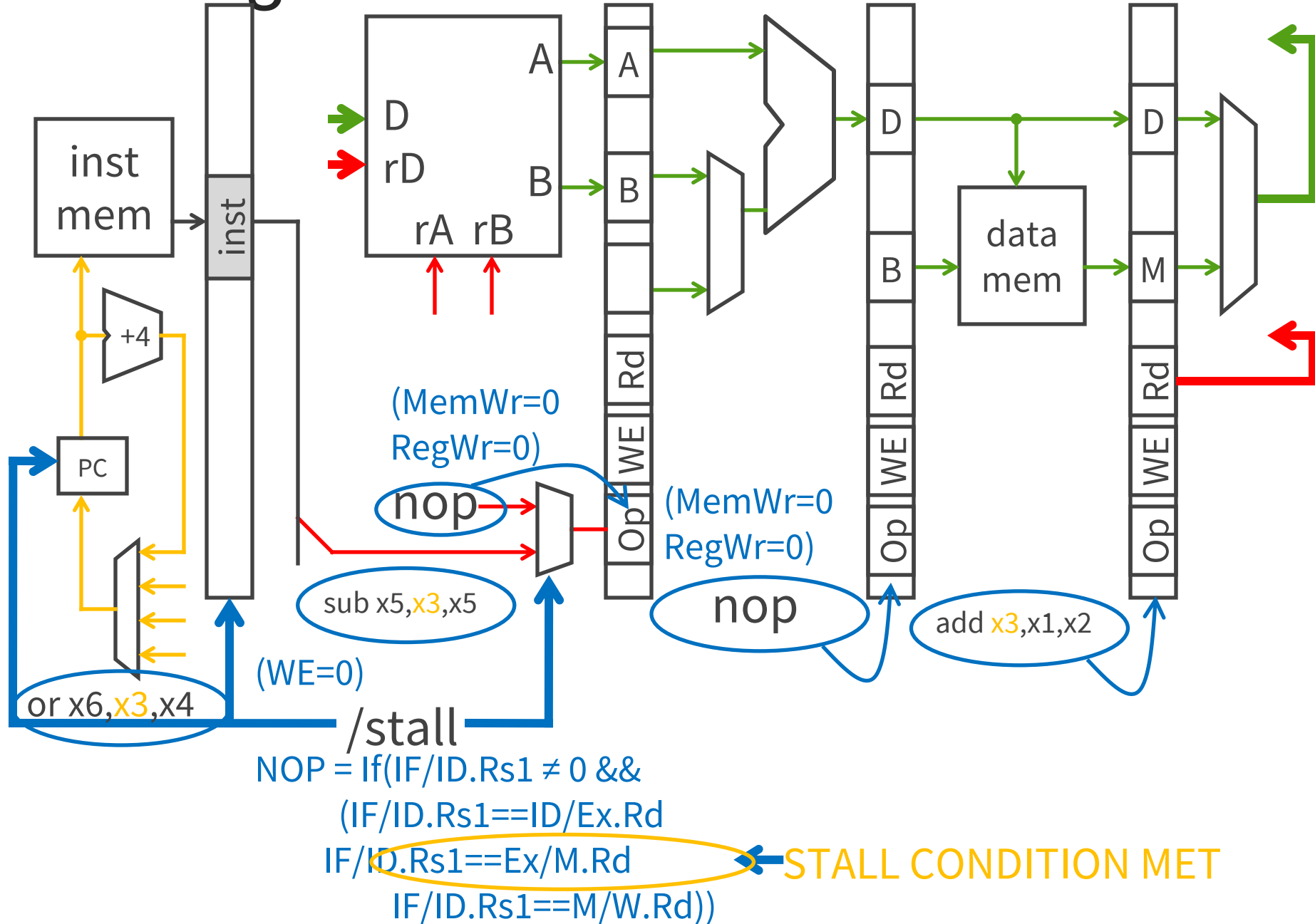
Stalling



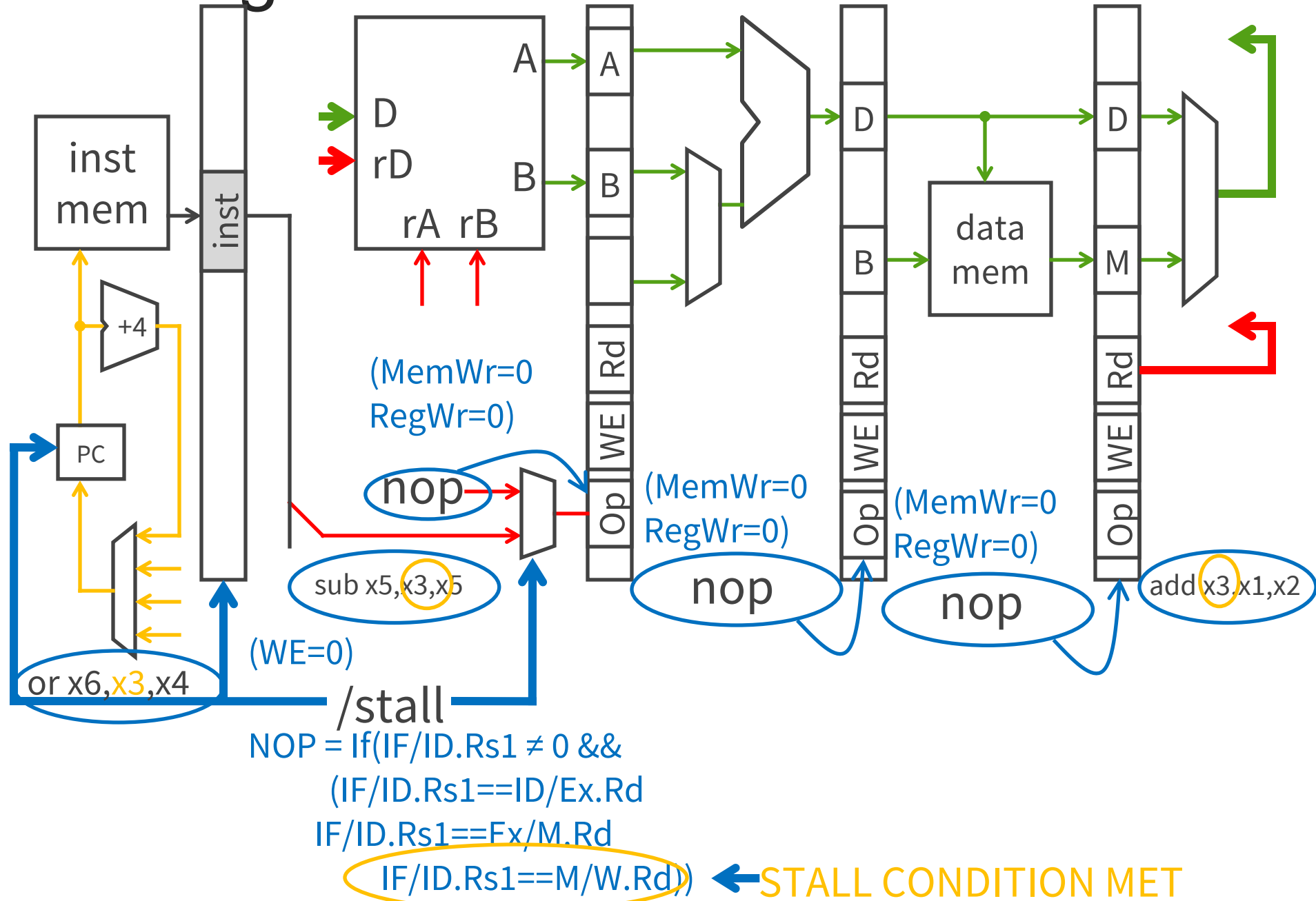
Stalling



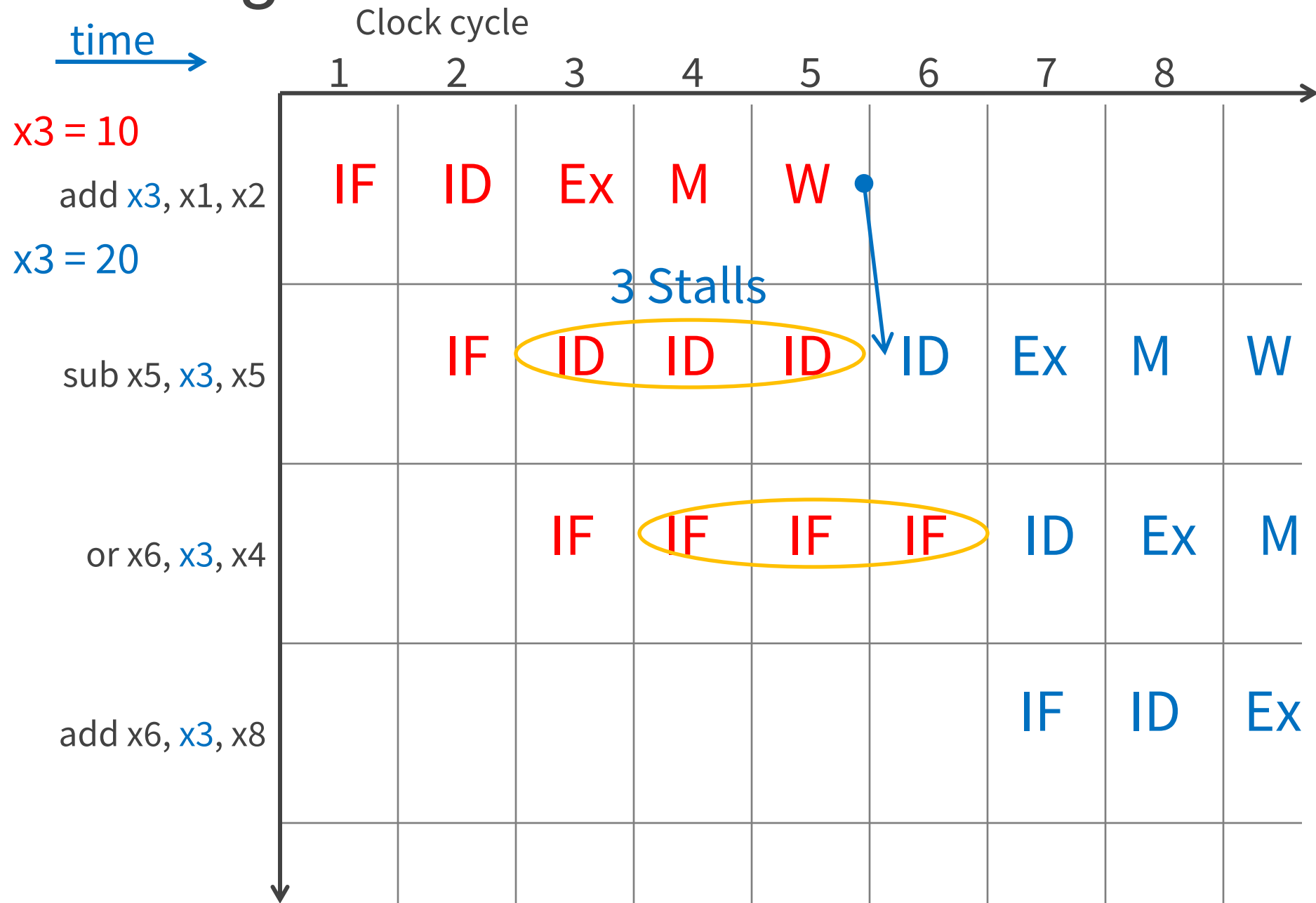
Stalling



Stalling



Stalling

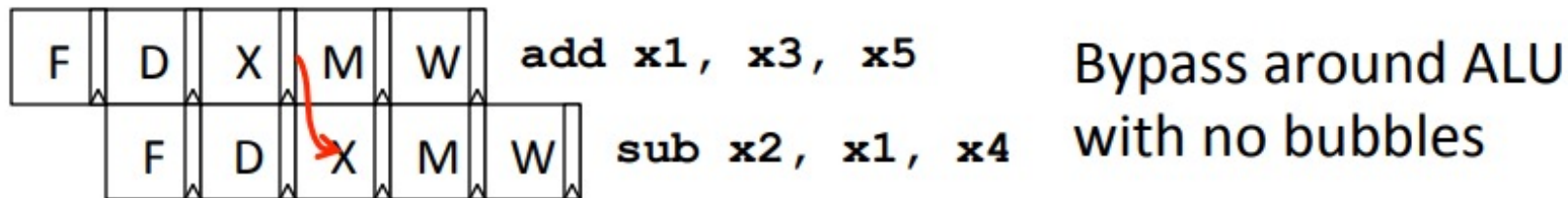
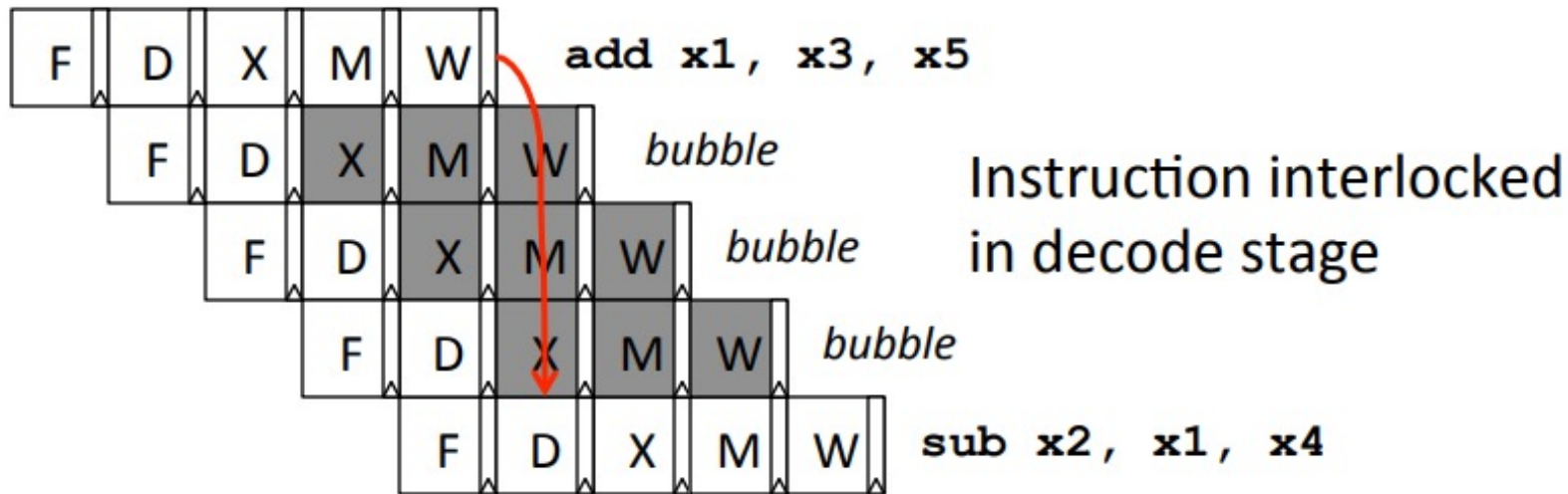


Stalls and Performance

- CPI increases on every hazard
- Stalls reduce performance
 - But stalls may be required to get correct results
- Are there better solutions?
 - Forwarding/Bypassing

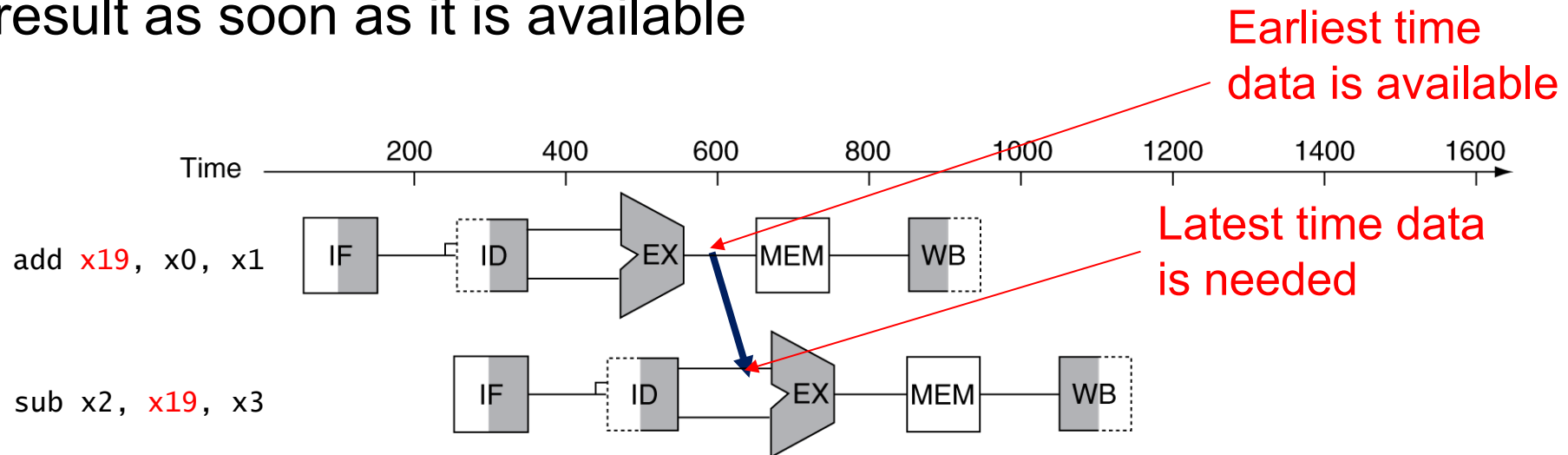
Interlock/Stall Versus Bypassing

add x1, x3, x5
sub x2, x1, x4



Detect & Forward

- Detection
 - Same as detect and stall, but...
 - each possible hazard requires different forwarding paths
- Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register).
- Use result as soon as it is available

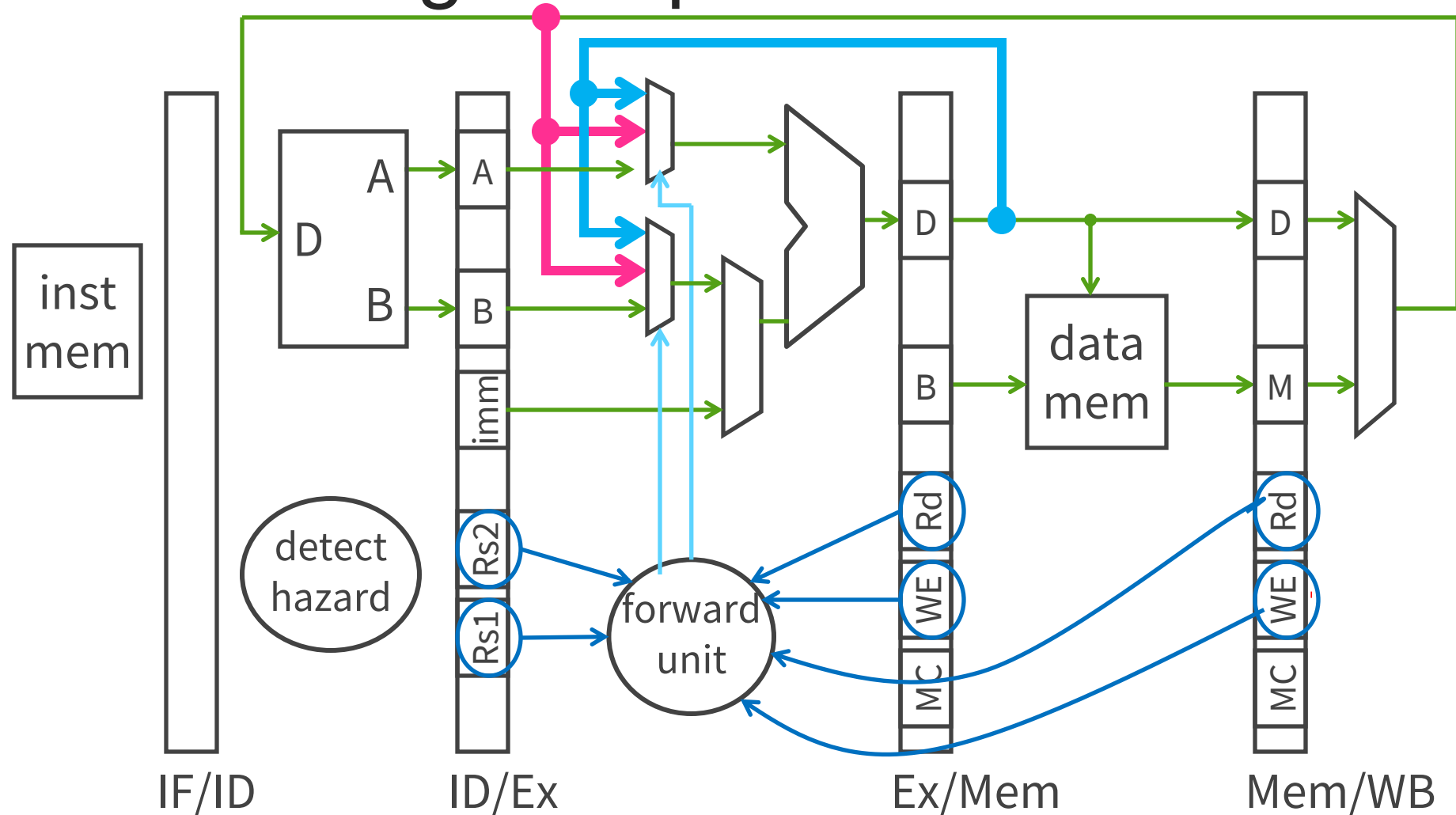


Can't connect input and output of a component directly

Detect & Forward

- Add mux in front of ALU to select source
- “bypassing logic” often a critical path in wide-issue machines
 - # paths grows quadratically with machine width

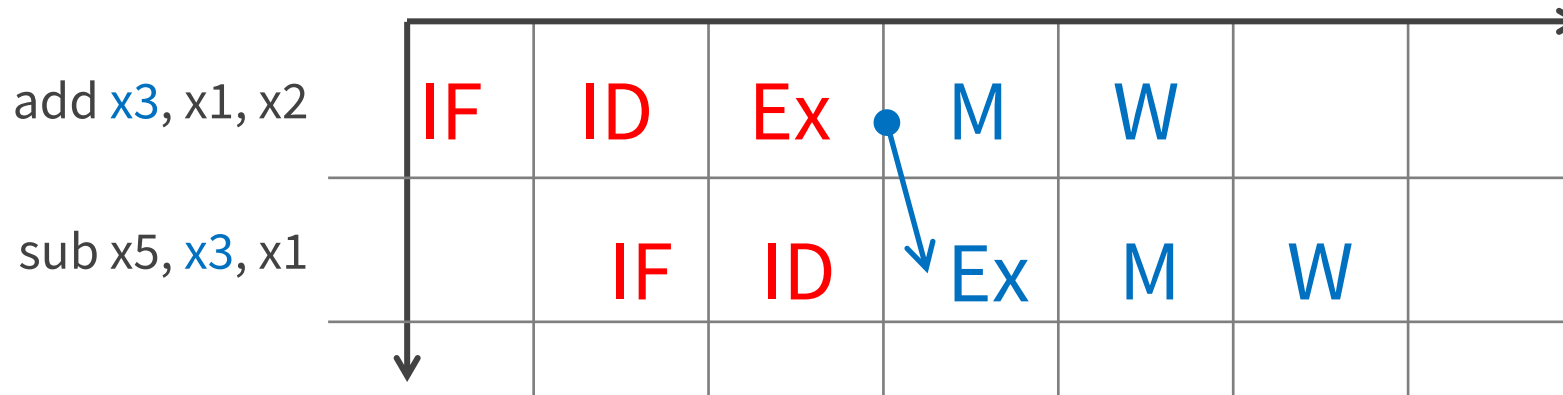
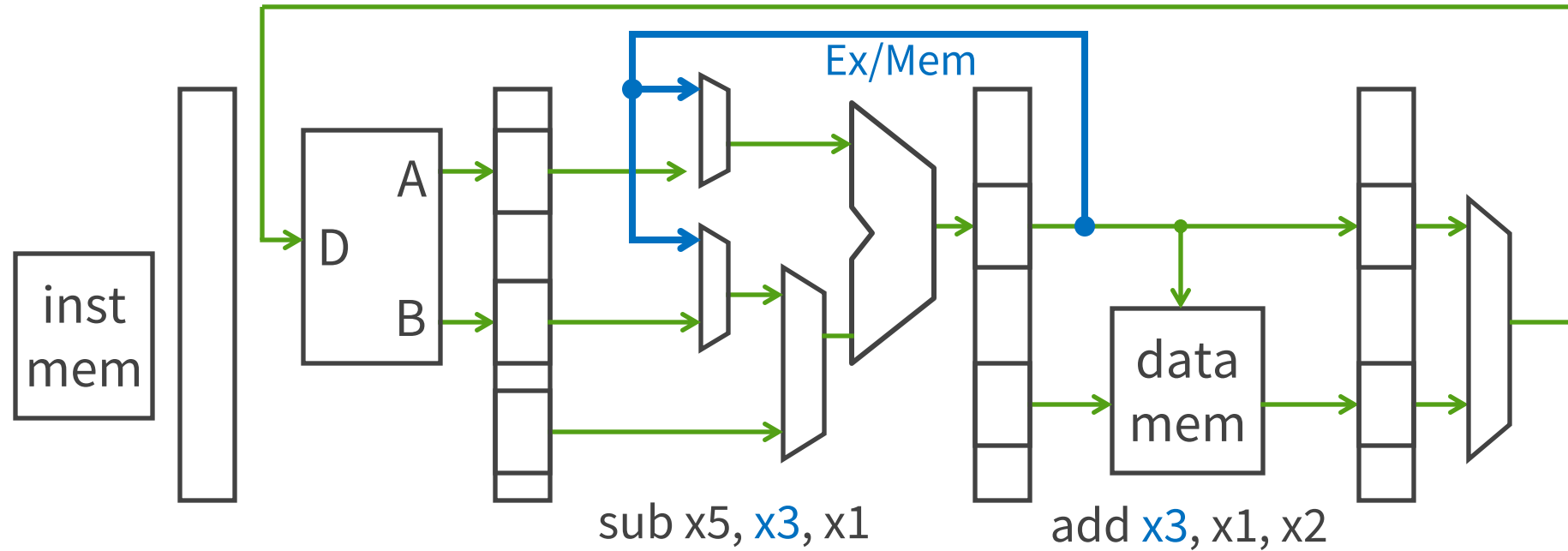
Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M → Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

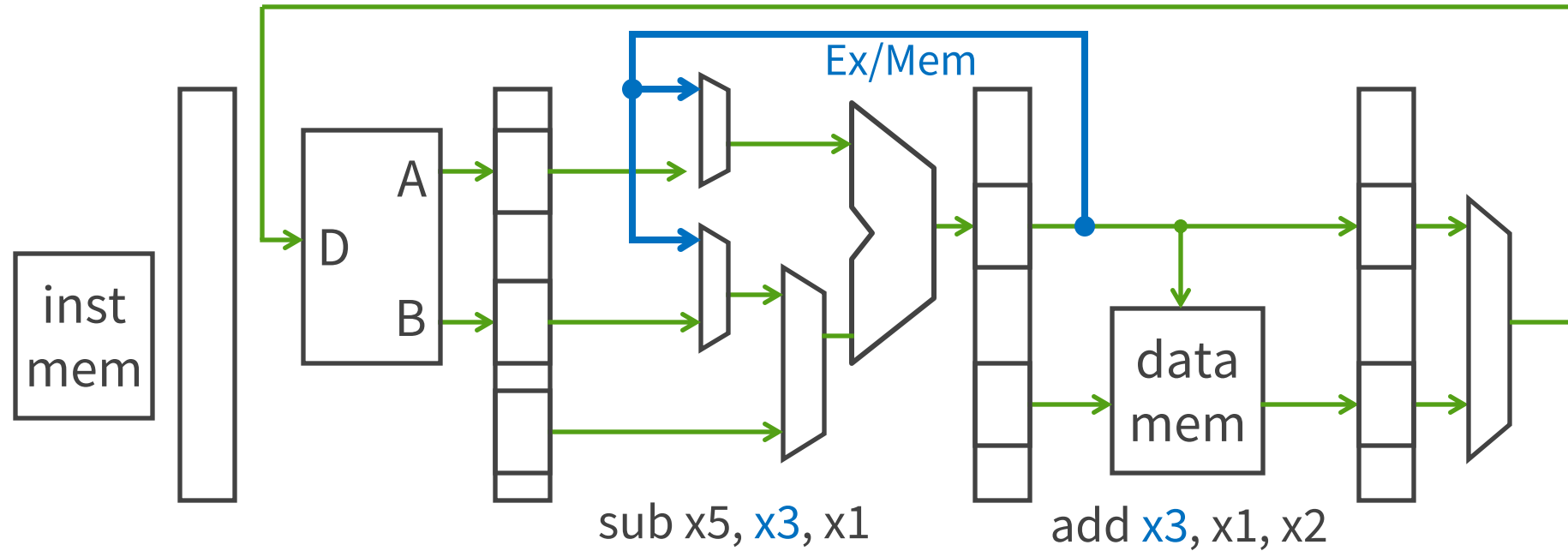
Forwarding Datapath 1: Ex/MEM → EX



Problem: EX needs ALU result that is in MEM stage

Solution: add a bypass from EX/MEM.D to start of EX

Forwarding Datapath 1: Ex/MEM → EX



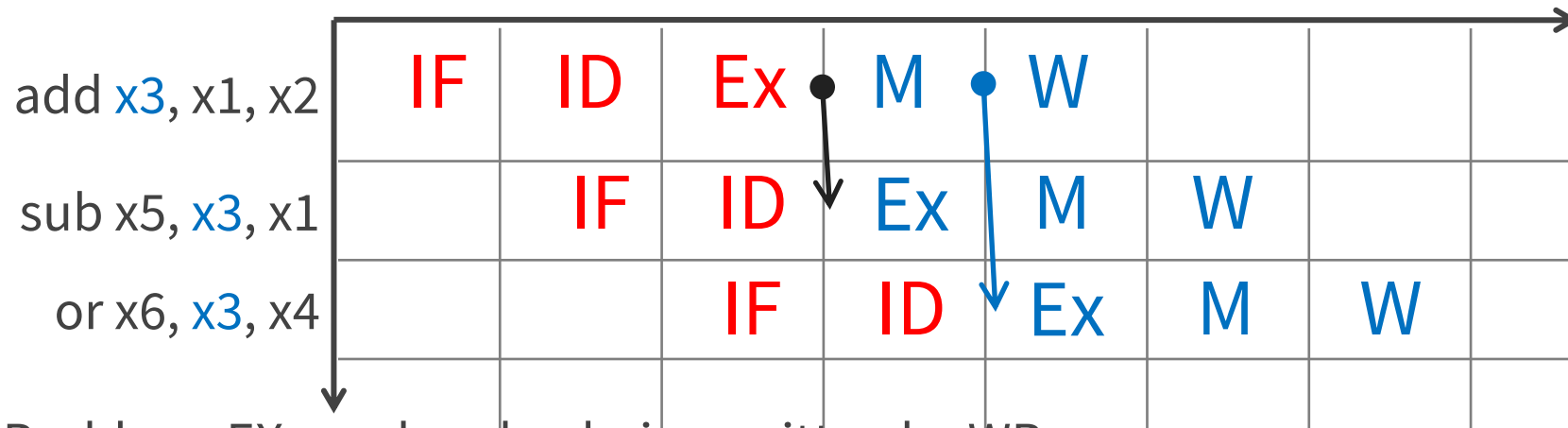
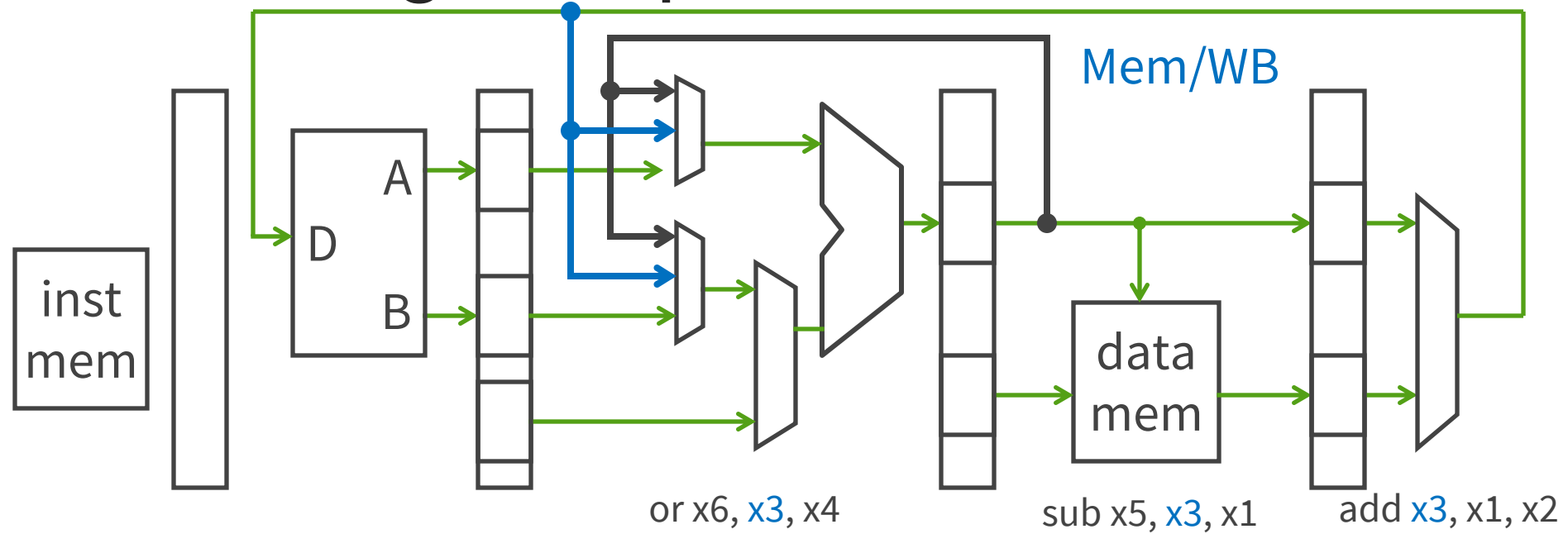
Detection Logic in Ex Stage:

forward = (Ex/M.WE && EX/M.Rd != 0 &&
ID/Ex.Rs1 == Ex/M.Rd)
|| (same for Rs2)

[illegible]

Copyright: Prof. Hakim Weatherspoon @ Cornell, CS 3410

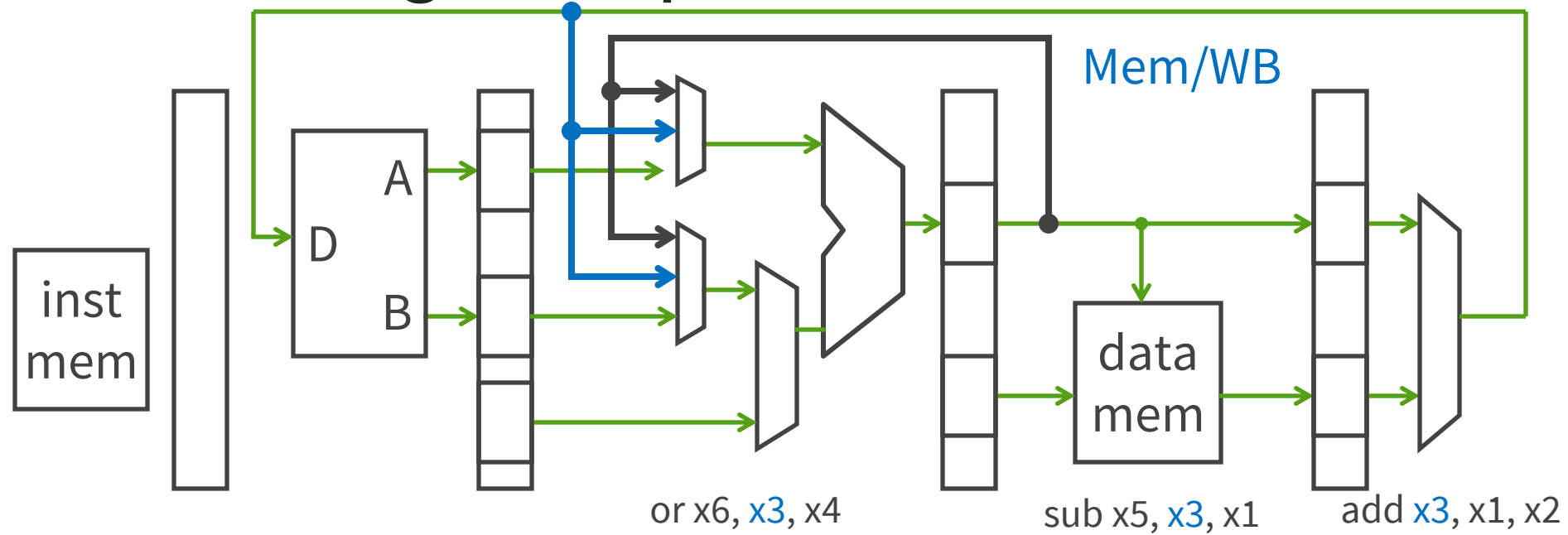
Forwarding Datapath 2: Mem/WB → EX



Problem: EX needs value being written by WB

Solution: Add bypass from WB final value to start of EX

Forwarding Datapath 2: Mem/WB→ EX



Detection Logic:

$\text{forward} = (\text{M/WB.WE} \ \&\& \ \text{M/WB.Rd} \neq 0 \ \&\&$

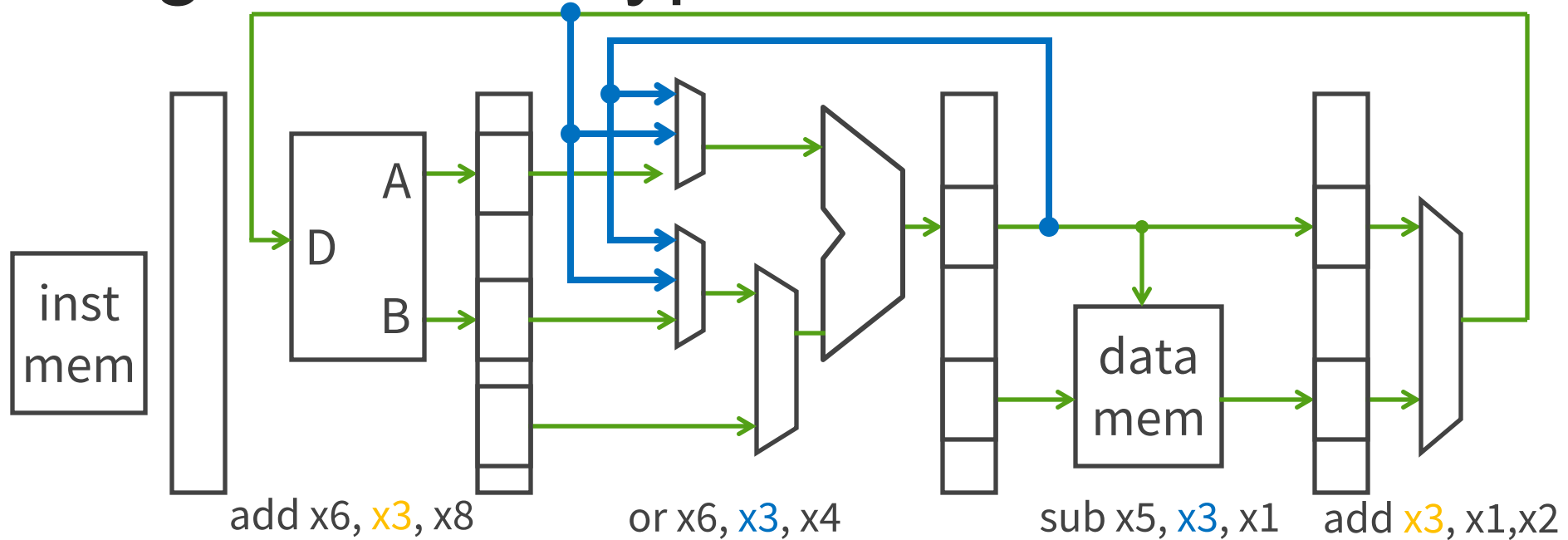
$\text{ID/Ex.Rs1} == \text{M/WB.Rd} \ \&\&$

$\text{not} (\text{Ex/M.WE} \ \&\& \ \text{Ex/M.Rd} \neq 0 \ \&\&$

$\text{ID/Ex.Rs1} == \text{Ex/M.Rd})$

\parallel (same for Rs2)

Register File Bypass

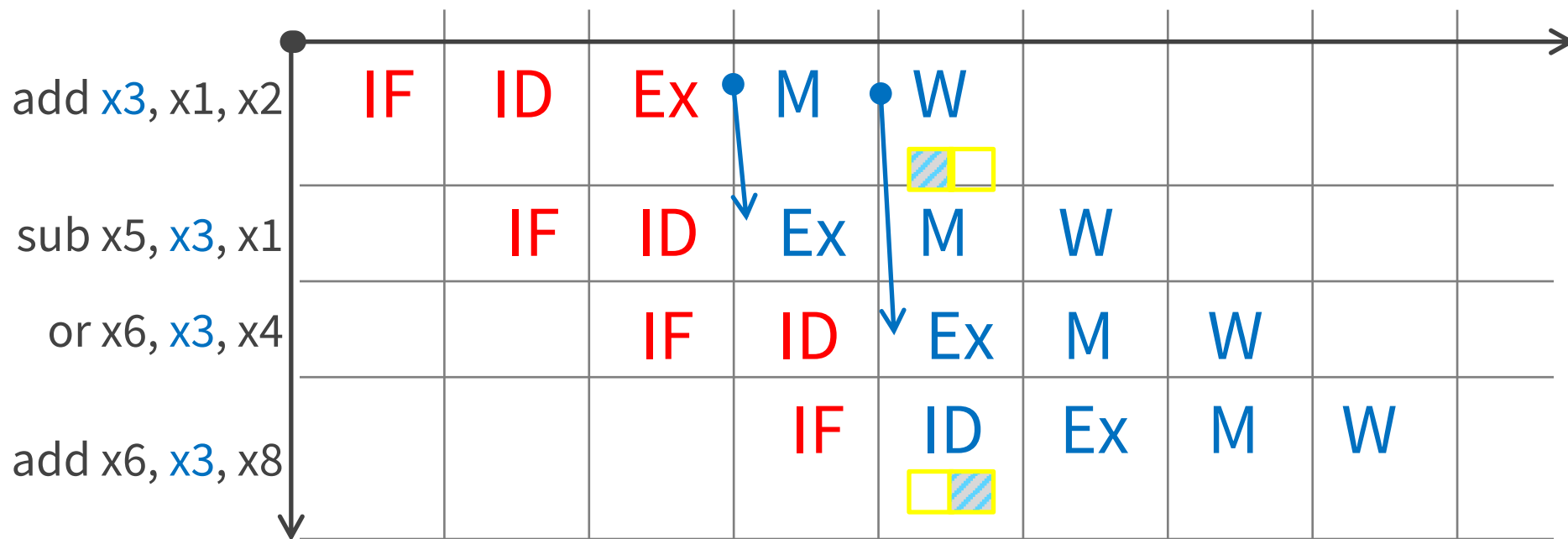
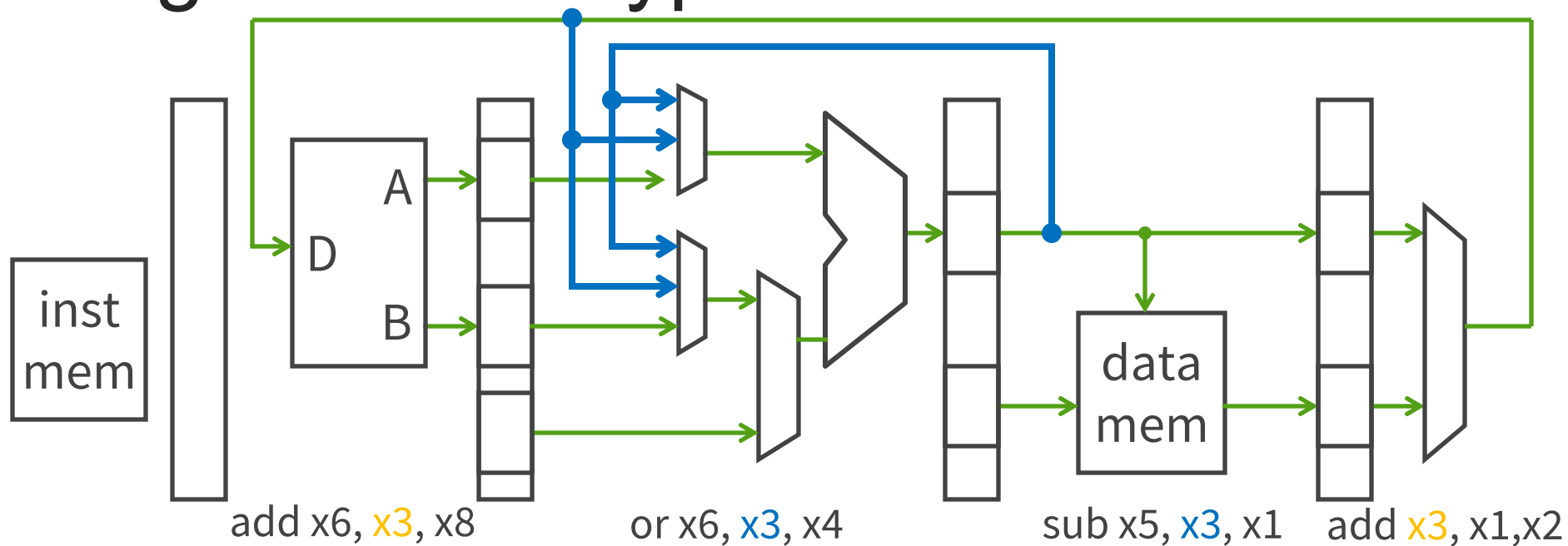


Problem: Reading a value that is currently being written

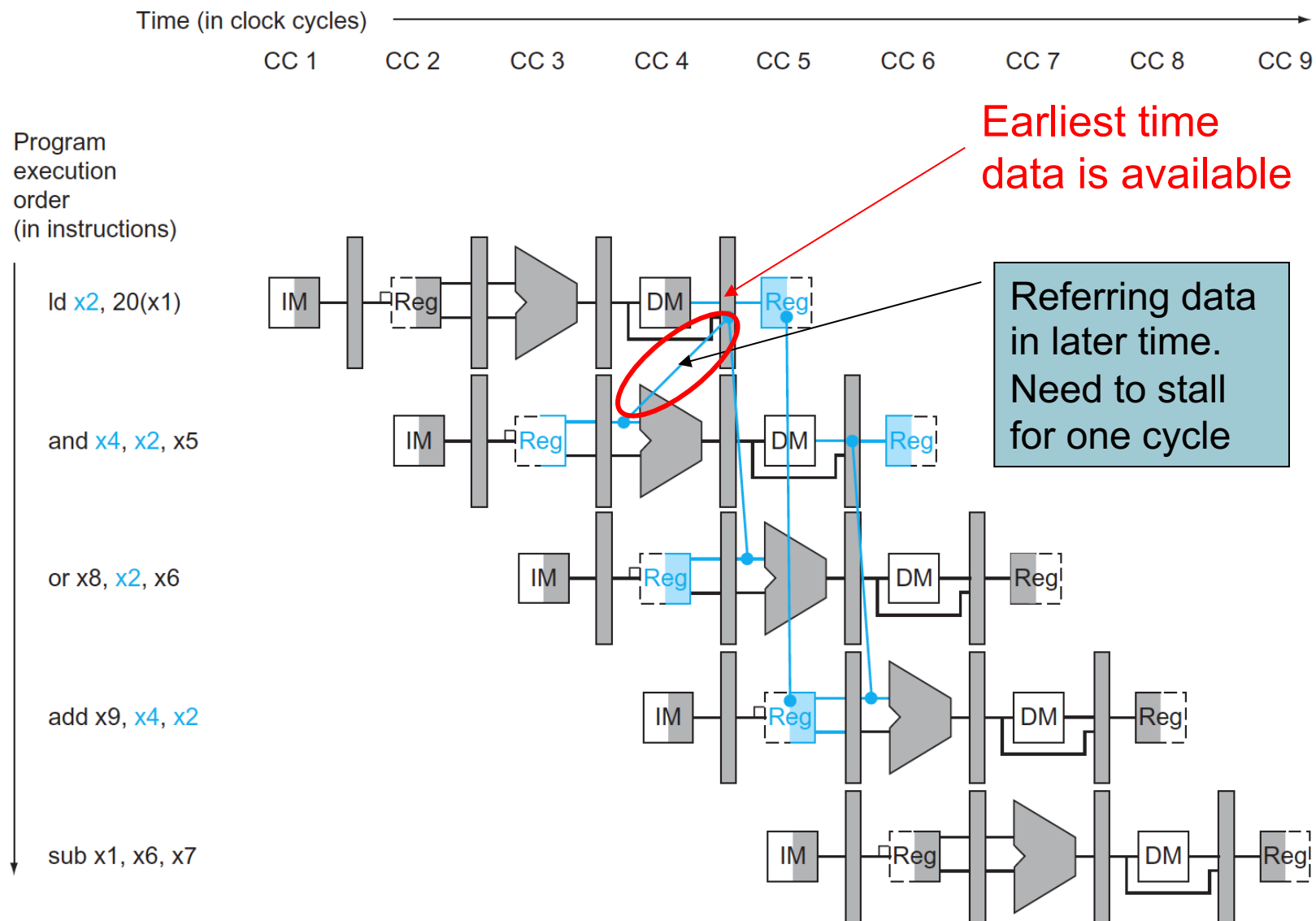
Solution: **just negate register file clock**

- writes happen at end of first half of each clock cycle
- reads happen during second half of each clock cycle

Register File Bypass



Forwarding May Fail



Load-Use Hazard Detection

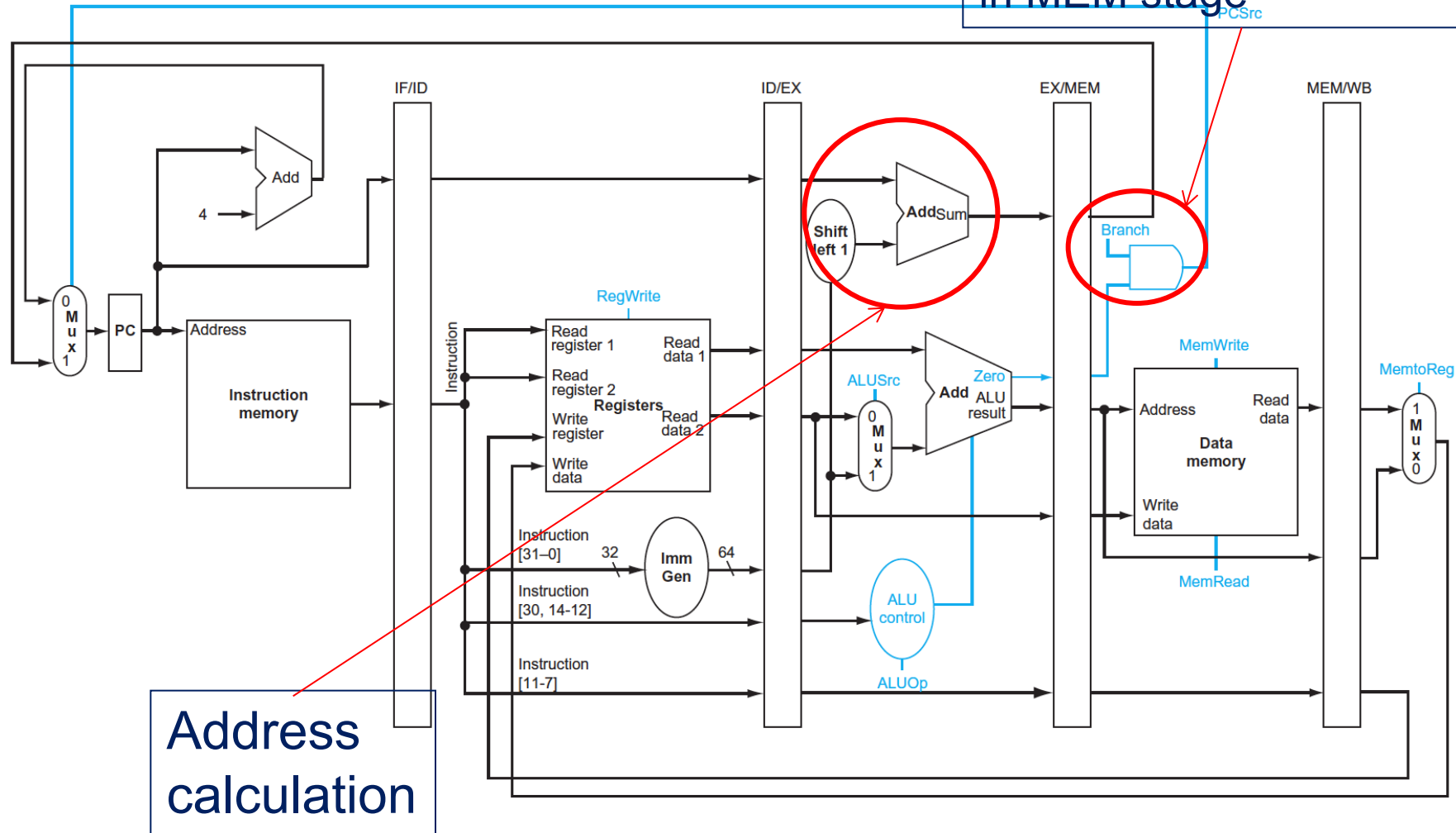
- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
 - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
 - ID/EX.MemRead and
((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
(ID/EX.RegisterRd = IF/ID.RegisterRs2))
- Hazard Detection is in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

Data Hazard Summary

- Stall
 - Pause current and all subsequent instructions
- Forward/Bypass
 - Try to steal correct value from elsewhere in pipeline
 - Otherwise, fall back to stalling or require a delay slot
- Detection is important!

Revisit Pipelining

Determination for branch
in MEM stage



Address
calculation

Control Hazards

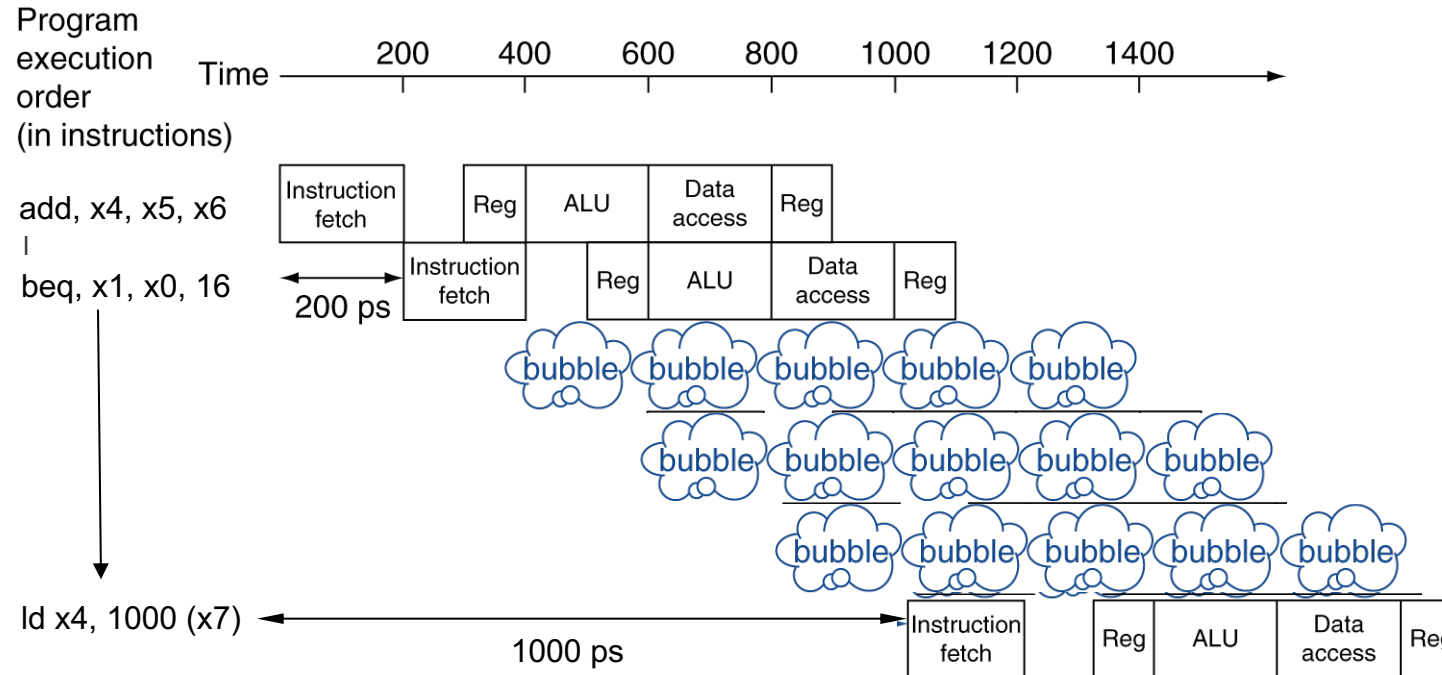
- RISC-V Branch Instruction
 - `beq x10, x11, 2000` // if `x10 == x11`, go to location with respect to `2000ten = 0111 1101 0000`
- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction (Branch Hazard) because
 - When Branch instruction is still in the **ID** stage, target instruction is needed in the **IF** stage

Handling Control Hazards

- Stall on branch
- Always assume branch not taken or taken
- Branch prediction
- Delayed Branch

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Assume Branch Not Taken

- If we are right, lucky us!
- If we are wrong, penalty will be to flush some (up to 3) instructions
- Penalty may be reduced
 - By making earlier decision on branch or not
 - Need to compare registers and compute target early in the pipeline

A quick quiz

- Under the condition “Assume Branch Not Taken”, if we evaluate the branch in the execute stage, we would lose two cycles every time we encountered a taken branch. If we assume 20% of instructions are branches, and 60% are taken, what is the CPI?
 - A. 1
 - B. 1.24
 - C. 1.4
 - D. 1.12
 - E. None of the above, can't decide.

Branch Prediction

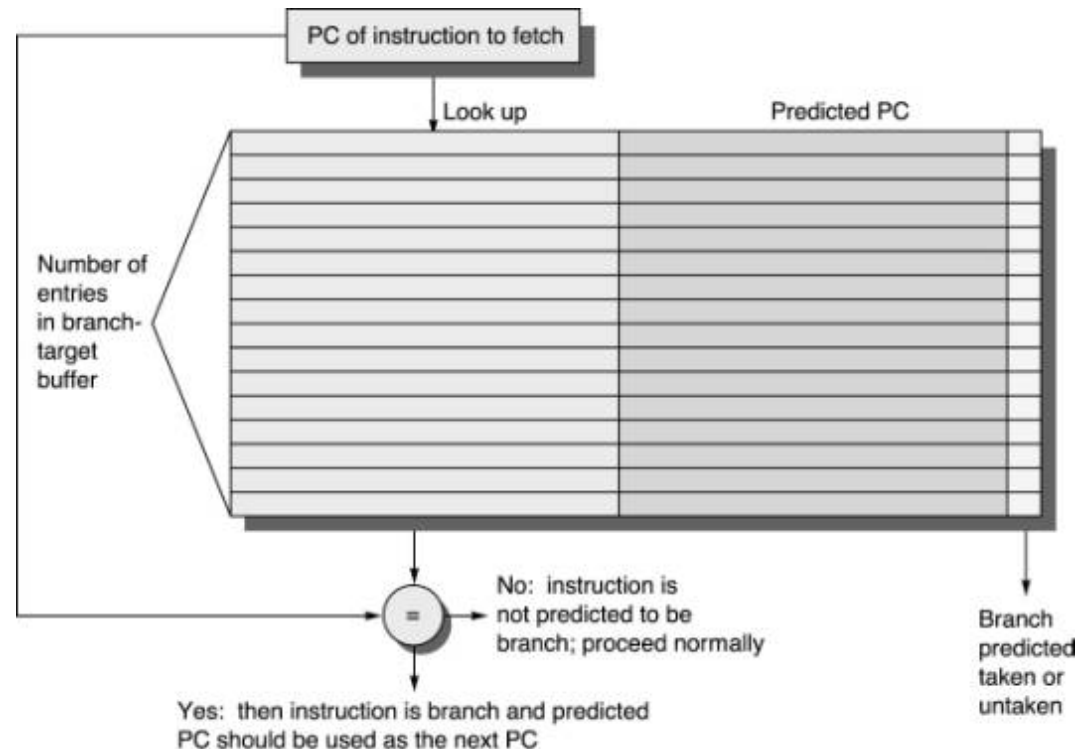
- Static prediction
 - Based on typical branch behavior
 - All decisions are made at compile time
 - Example: loop and if-statement branches
 - Could predict backward branches taken
 - Could predict forward branches not taken
- Dynamic prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume future behavior will continue the trend
 - If wrong, take penalty, and update history

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
 - Branch prediction buffer (aka branch history table)
 - Indexed by recent branch instruction addresses
 - Stores outcome (taken/not taken)
 - To execute a branch
 - Check table, expect the same outcome
 - Start fetching from fall-through or target
 - If wrong, flush pipeline and flip prediction

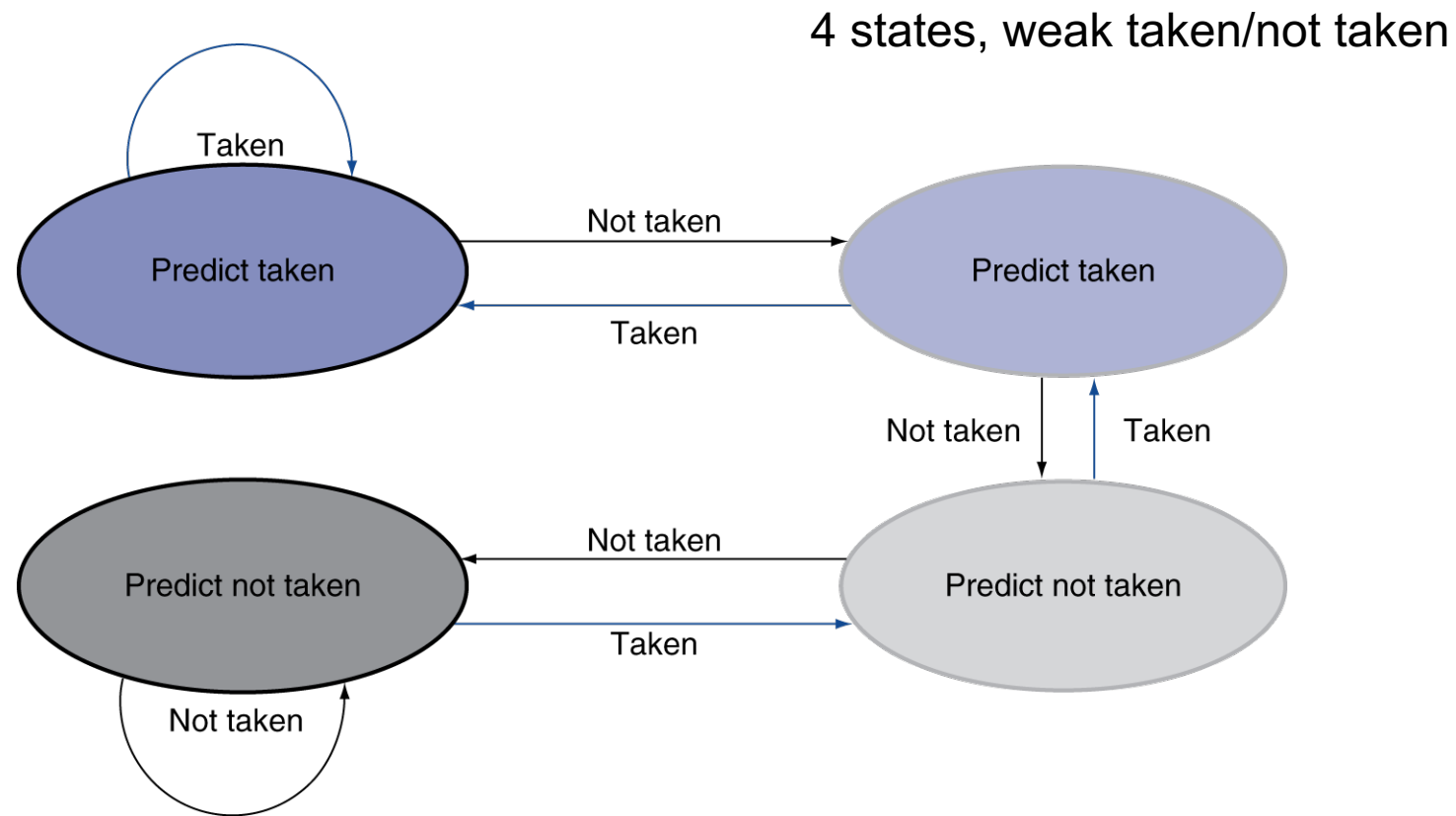
Dynamic Branch Prediction Components

- Branch prediction/target buffer or branch history table (BTB)
 - A small memory indexed by the lower portion of the address of the branch instruction
 - The memory contains a bit that says whether the branch was recently taken or not. (If the prediction is incorrect, the prediction bit is inverted and stored back.)



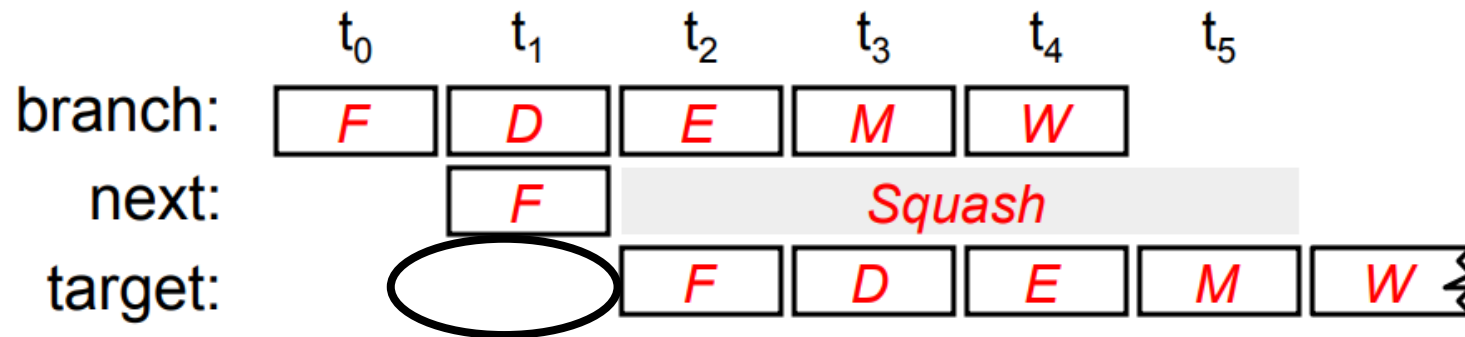
2-Bit Predictor

- Only change prediction on two successive mispredictions



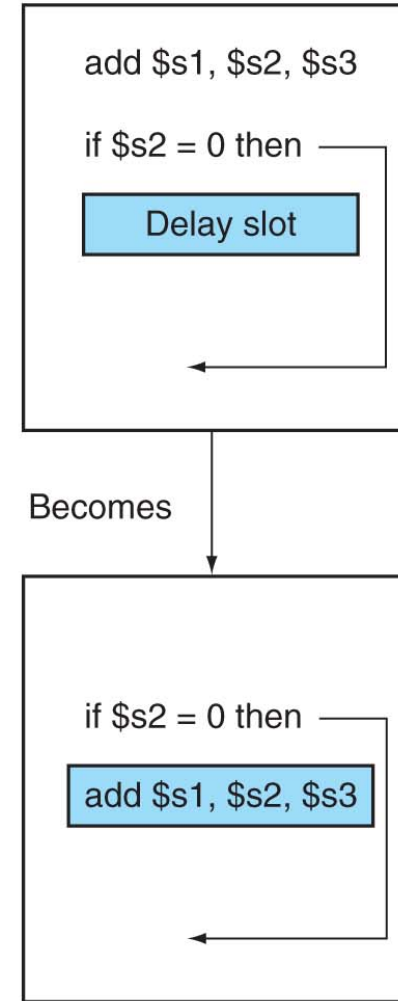
Delayed Branch

- Always delay the branch
 - With one instruction, NOP or usually a real instruction, for 5 stage pipeline
 - Maybe more delays for deeper pipeline
- Requires carefully designed compiler



Example for Delayed Branch

- Will remove the 1 clock cycle penalty
- Will work only if instructions can be found to fill the delay slot



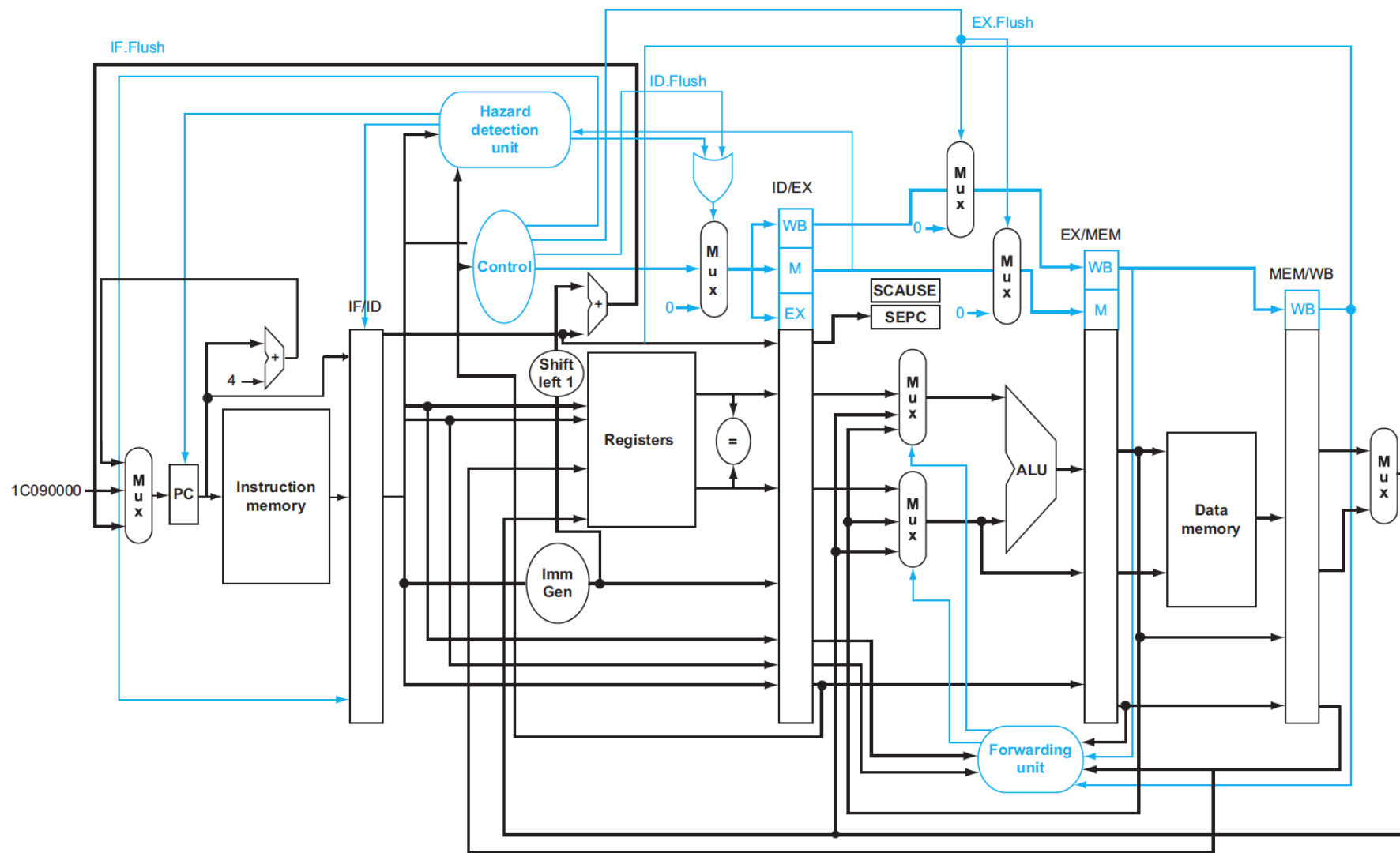
Delayed Branch

- Always execute the instruction immediately following branch
 - Called Branch delay slot
 - Further reduce branch penalty
- One branch delay slot becomes insufficient
 - When pipeline gets deeper or multiple instructions per clock cycle
 - Dynamic branch prediction is better choice

Pipeline Hazard Checklist (So far)

- Data dependencies (memory/register)
 - True Data Dependence (RAW)
 - Anti Dependence (WAR)
 - Output Dependence (WAW)
- Control Dependences

Where we are?



Five-stage in-order pipelined processor (End of VE370)...

Where are we Heading?

- T4: Advanced Processors

Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Prof. Ron Dreslinski @ UMich, EECS 470
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo@JI, VE370 2021 SU
- Xinfei Guo@JI, VE470 2022 SU

Action Items

- Check the lab schedules and get ready
- Reading Materials
 - Review pipeline & hazards topics from VE370
 - P&H, Computer Organization and Design RISC-V Edition, Ch. 4.5 – 4.6, 5.7
 - P&H, Computer Architecture, Ch. 3.1, 3.2, Appendix C.2
 - Wave pipeline paper