Topic 3

Fundamental Processors I

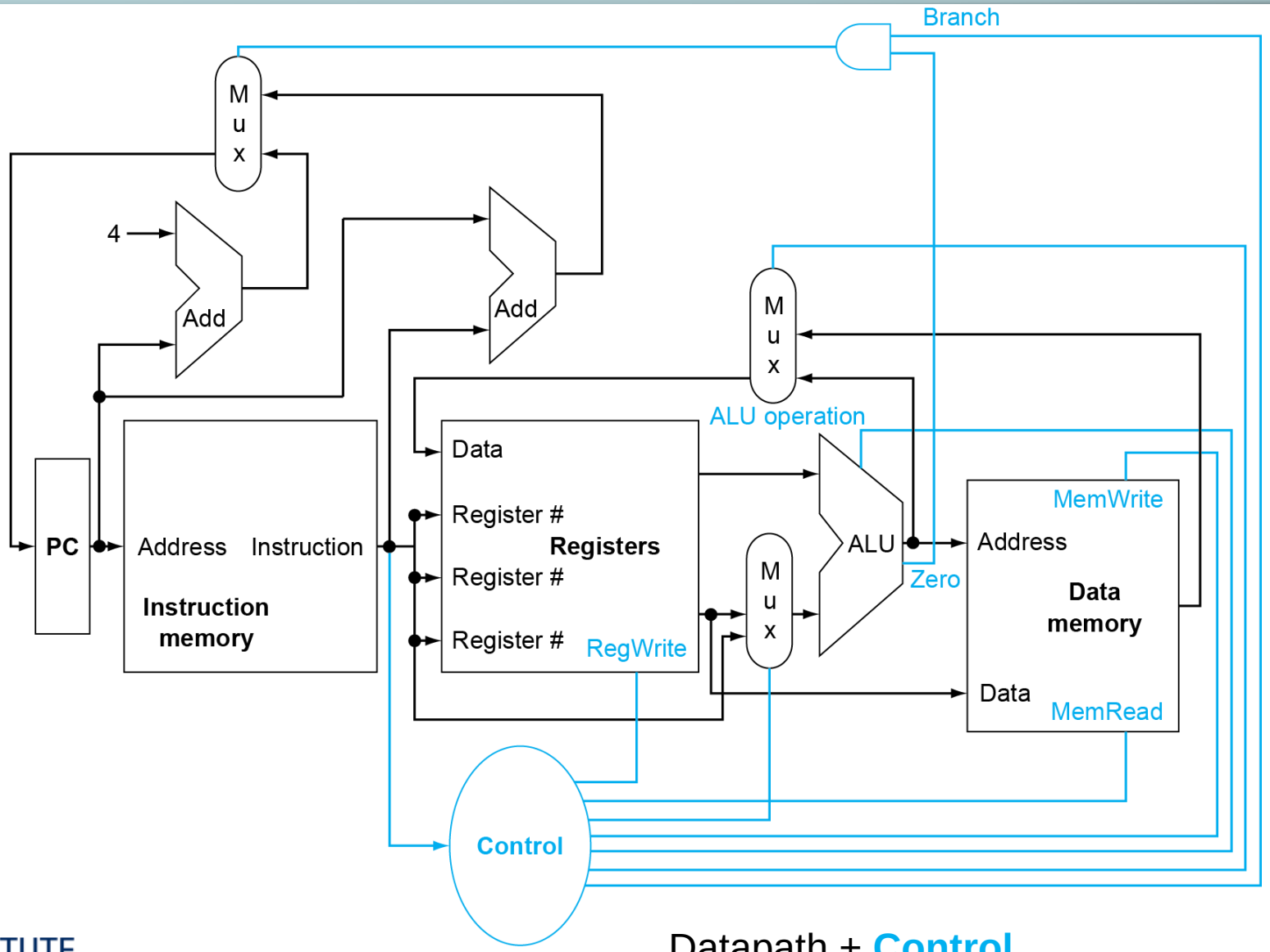
Xinfei Guo
xinfei.guo@sjtu.edu.cn

May 18th, 2022

T3 learning goals

- Fundamental Processors (review)
  - Section I: Pipelining
  - Section II: Hazards

Starting from a simple processor



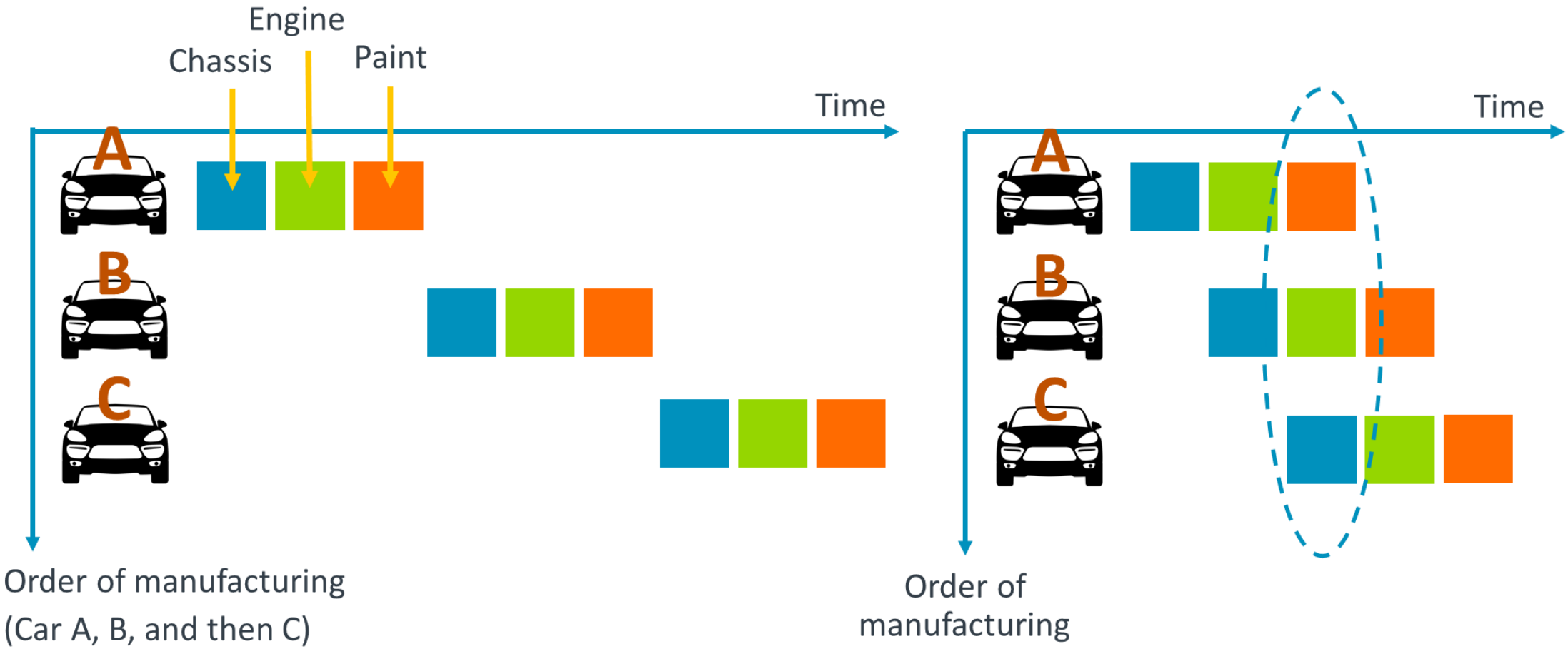
A single cycle RISC-V Processor

Datapath + **Control**

A Simple Processor

- Our processor executes each instruction in one clock cycle, i.e., it has a Clocks 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?



source: ARM

What Is Pipelining?

- We arrange for the different phases of execution to be overlapped. We aim to exploit “temporal” parallelism.
- How are latency and throughput affected?

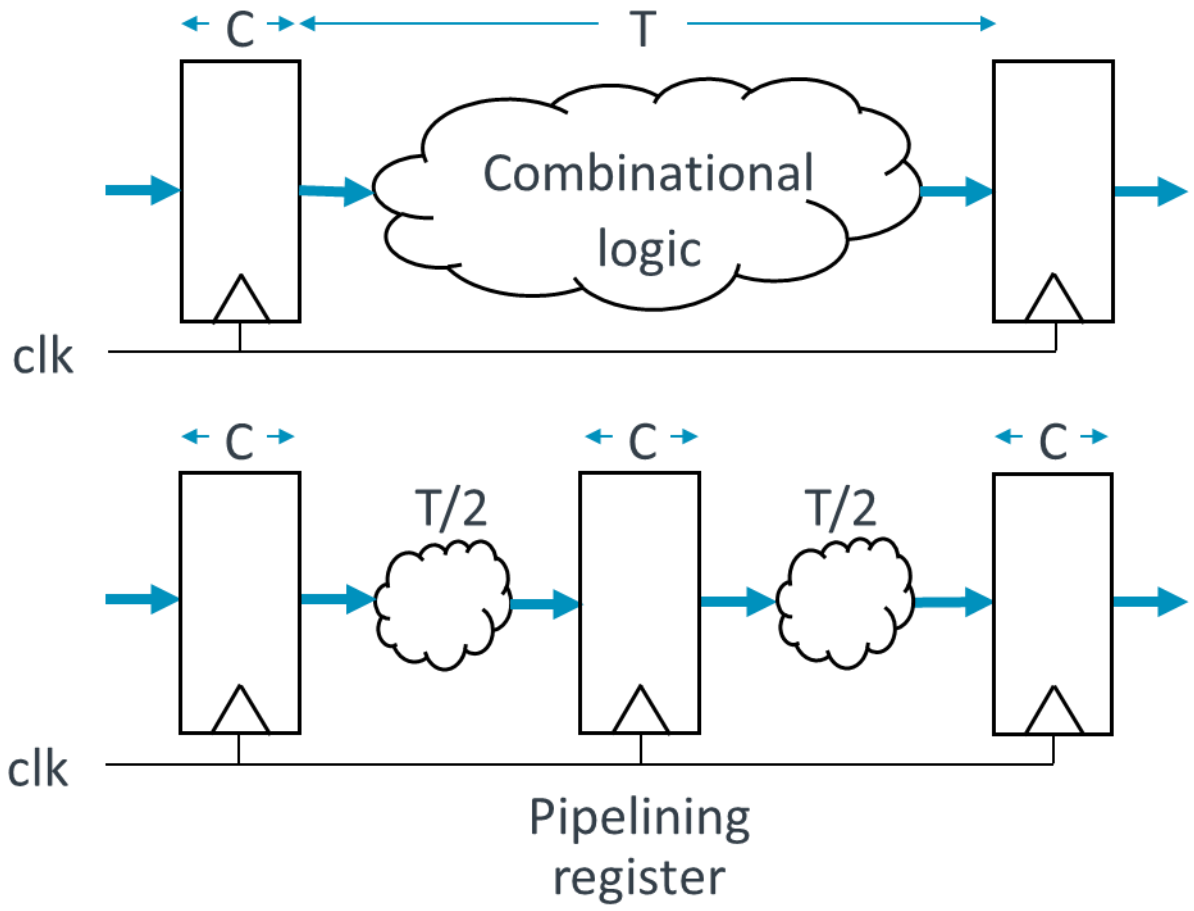
Volkswagen Beetle Assembly Line
(By Alden Jewell, license: CC BY 2.0)

source: ARM

Pipelining in computer architecture

- Pipelining become 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

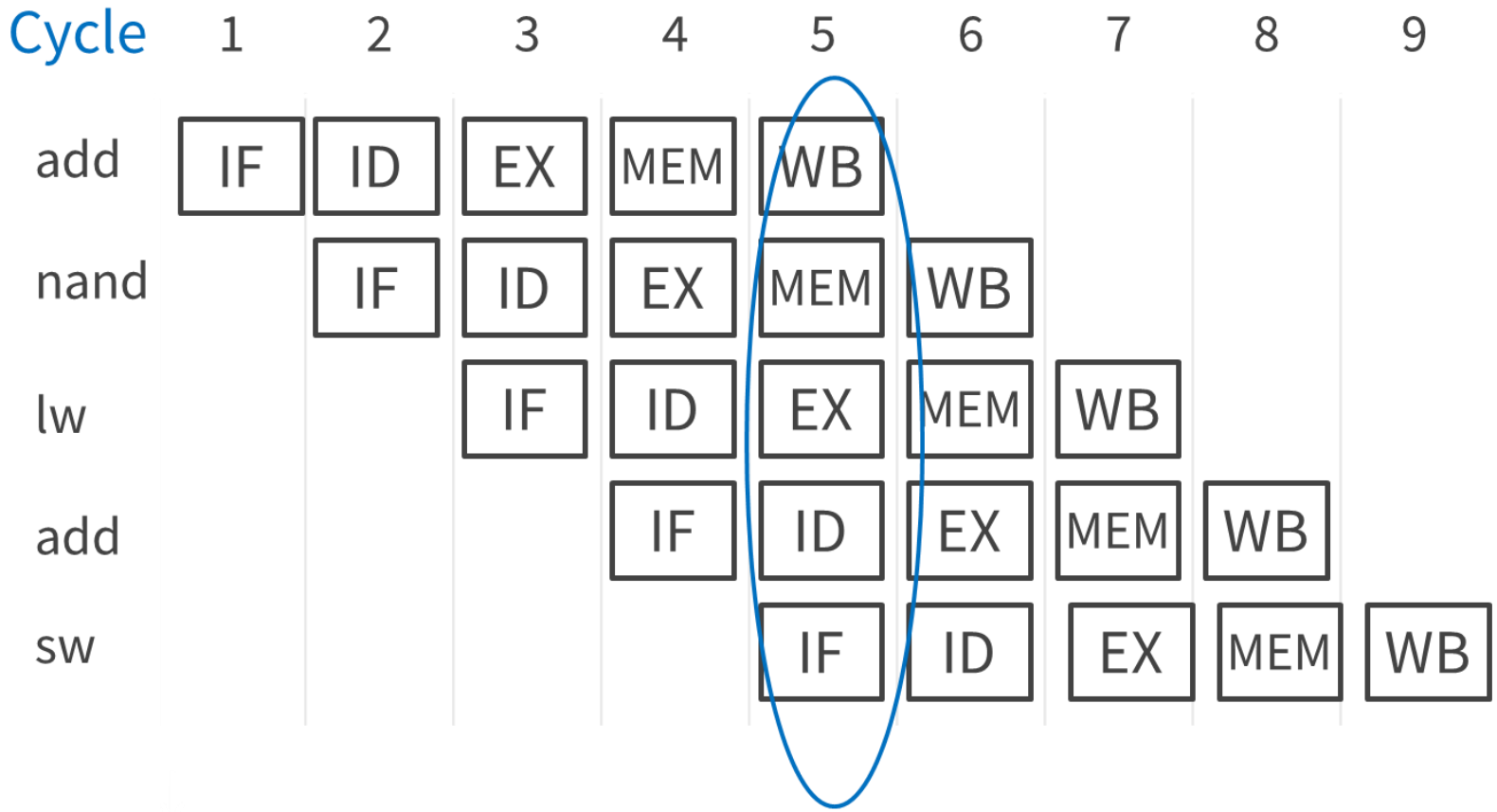
Now, if we create two pipeline stages:

Clock period = T/2 + C

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

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

Pipelined Processor

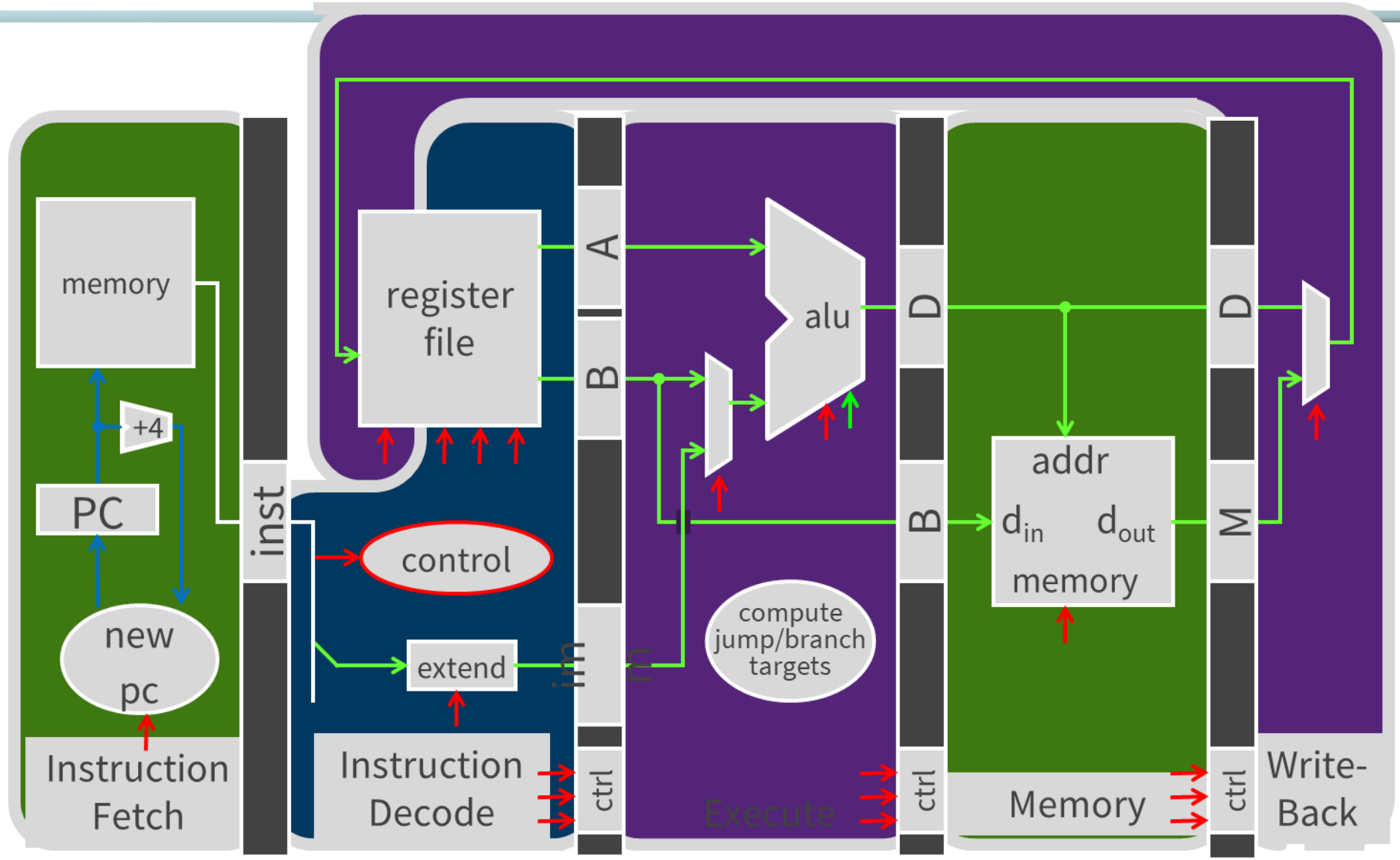


Latency: 5 cycles
Throughput: 1 insn/cycle
Concurrency: 5

CPI = 1 **Why?**

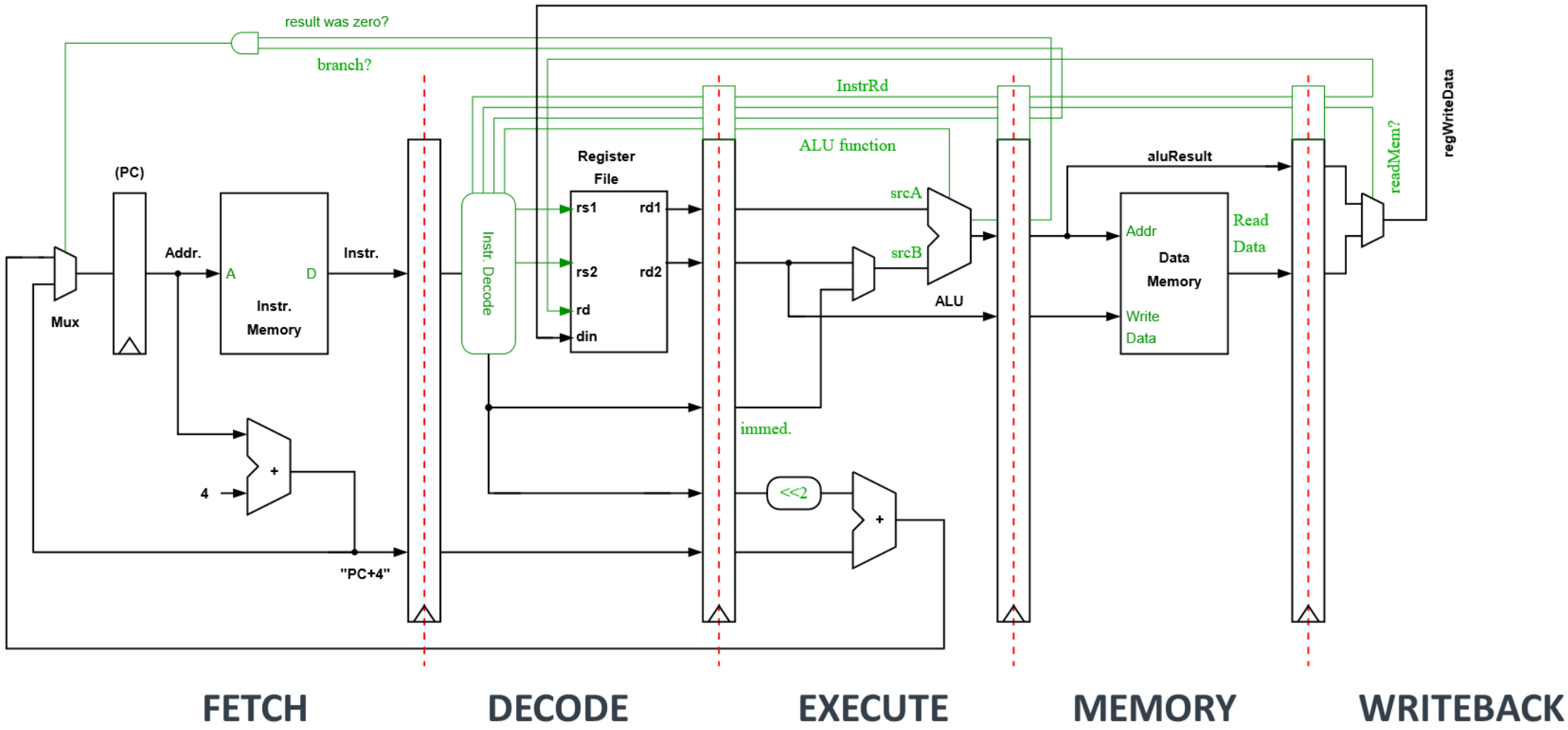
source: Prof. Hakim Weatherspoon @Cornell

Pipelined Processor (VE370)



source: Prof. Hakim Weatherspoon @Cornell

Pipelined Processor (ARM)



The same principle no matter what ISA!

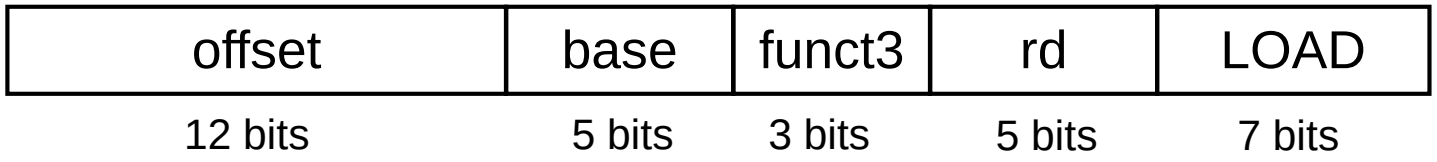
source: ARM

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	

Example (ld instruction -> I-format)

A RISC-V load instruction



- The value loaded from memory is stored in register rd
- This is very similar to the add-immediate operation but used to create address not to create final result

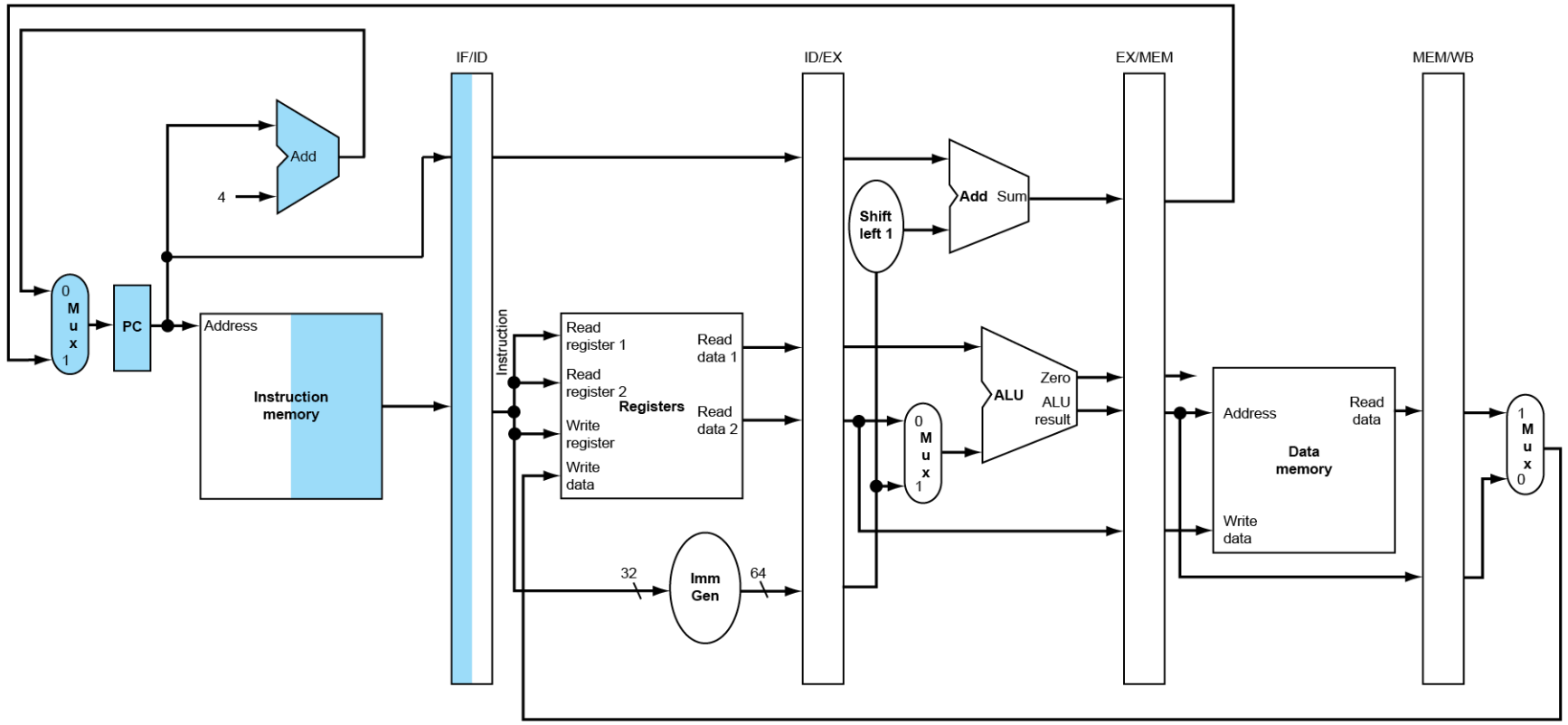
IF for Load, Store, …

I-type	immediate[11:0]	rs1	funct3	rd	opcode	Loads & immediate arithmetic

ld

Instruction fetch

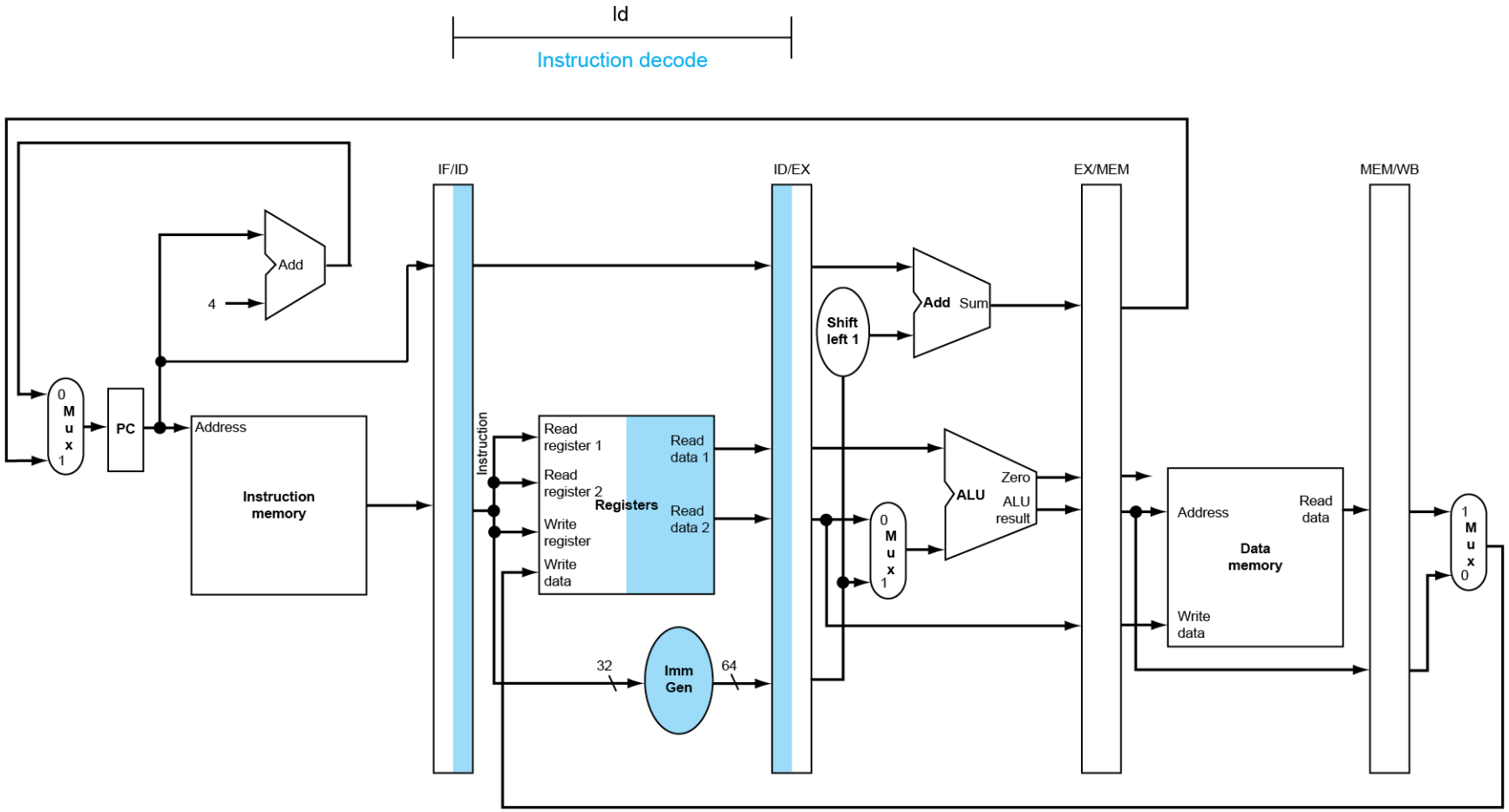
Load: Read memory and update register



- highlight the right half of registers or memory when they are being read and highlight the left half when they are being written

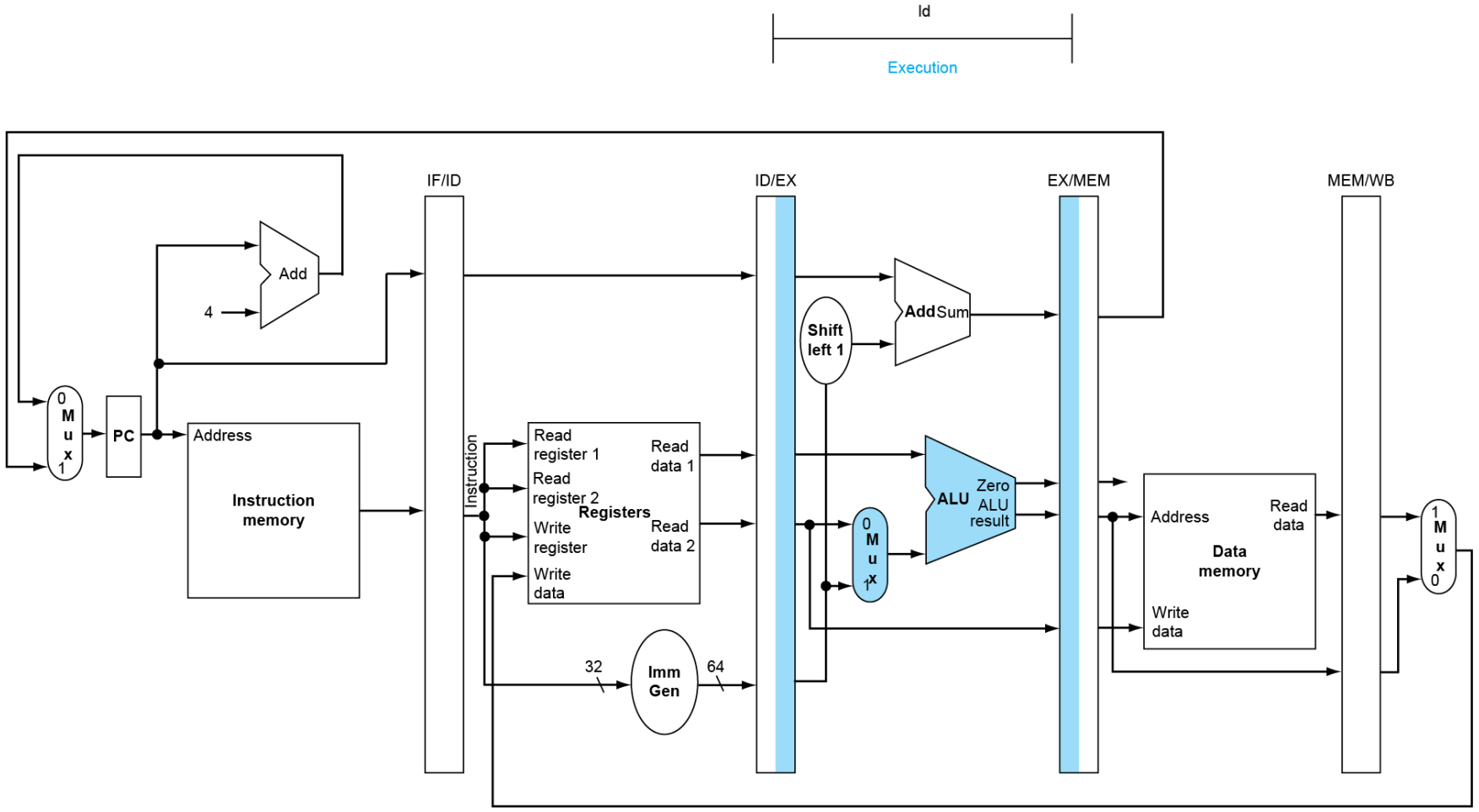
Taken from VE370, 2021 SU

ID for Load, Store, …



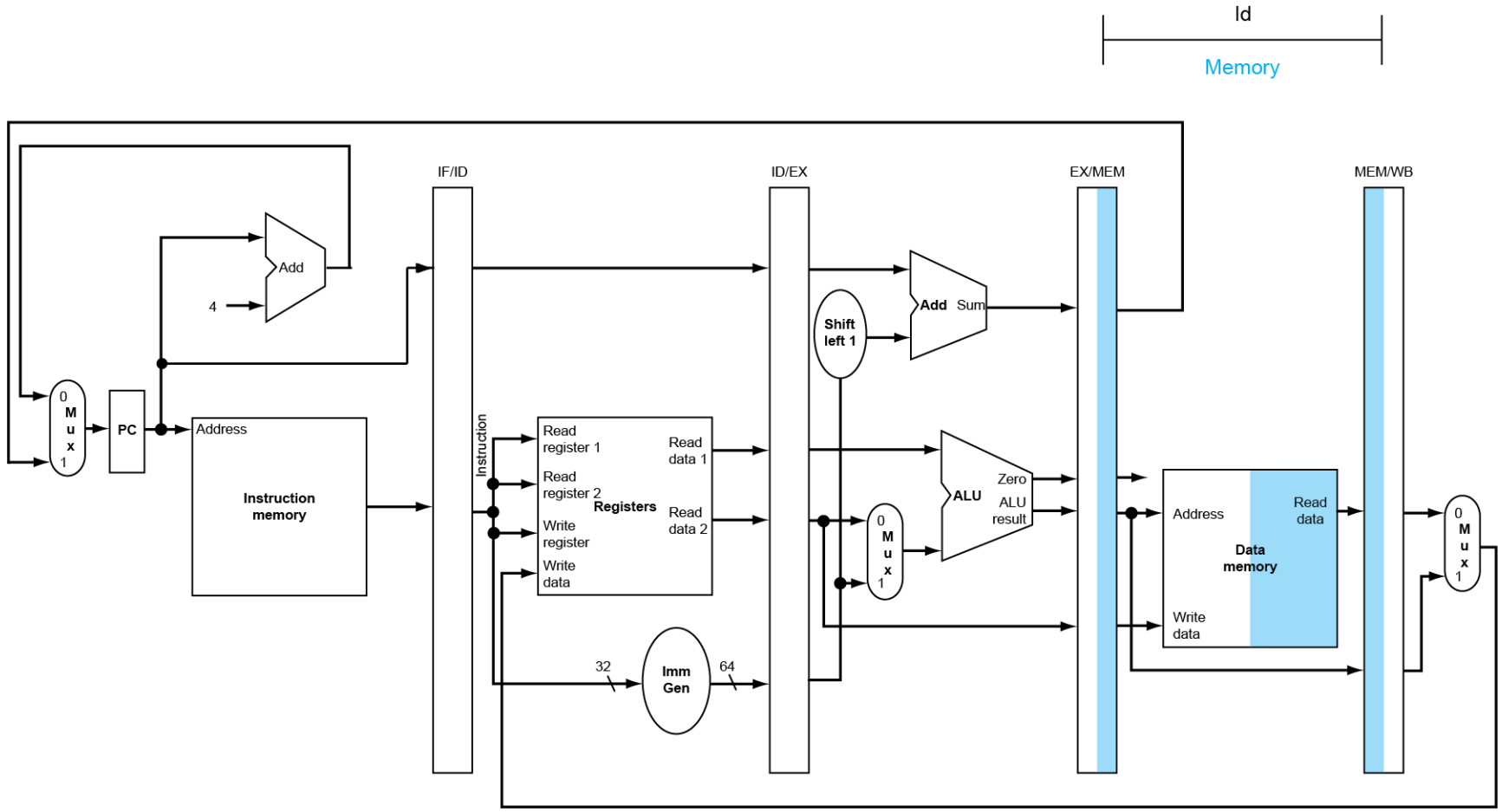
Taken from VE370, 2021 SU

EX for Load



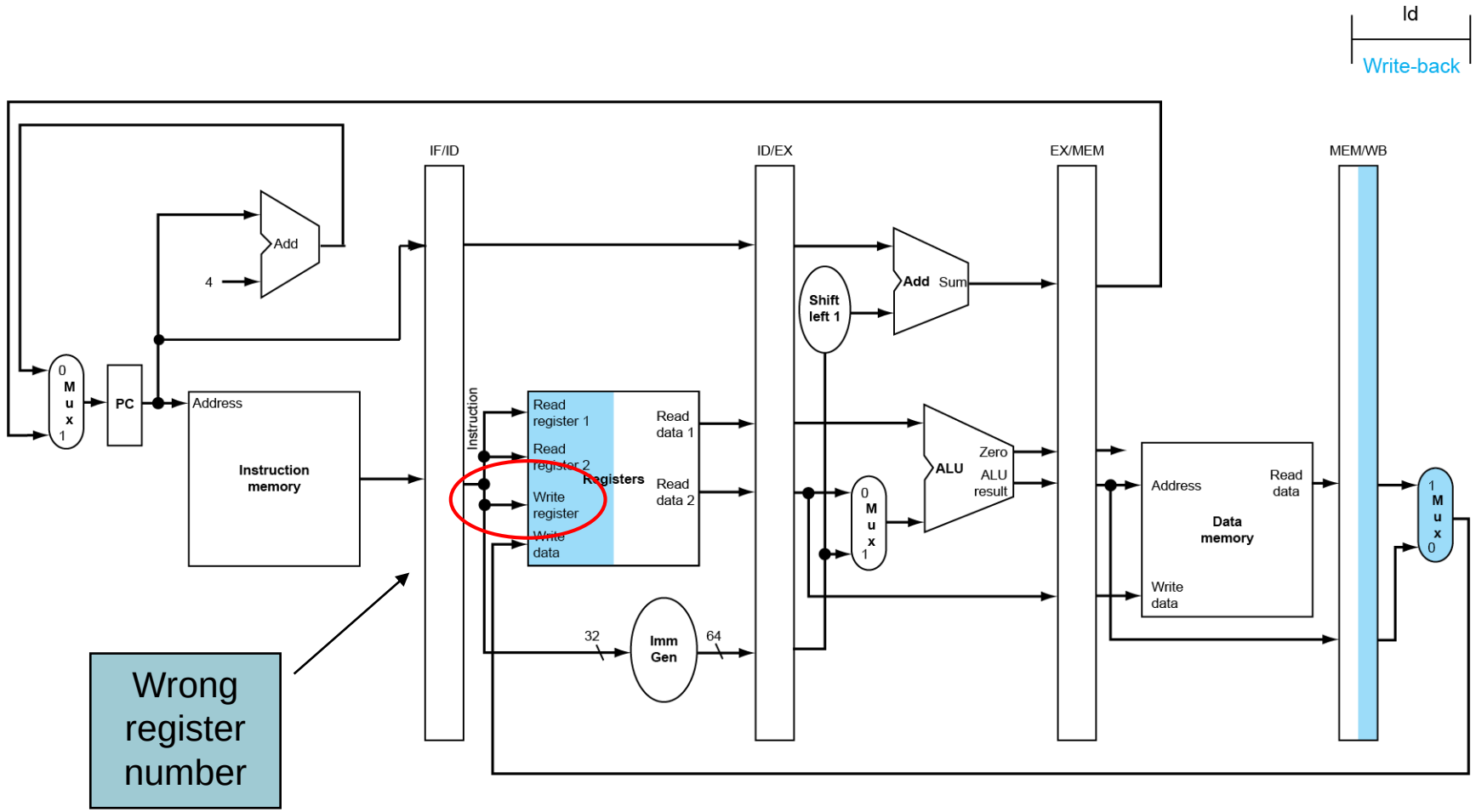
Taken from VE370, 2021 SU

MEM for Load



Taken from VE370, 2021 SU

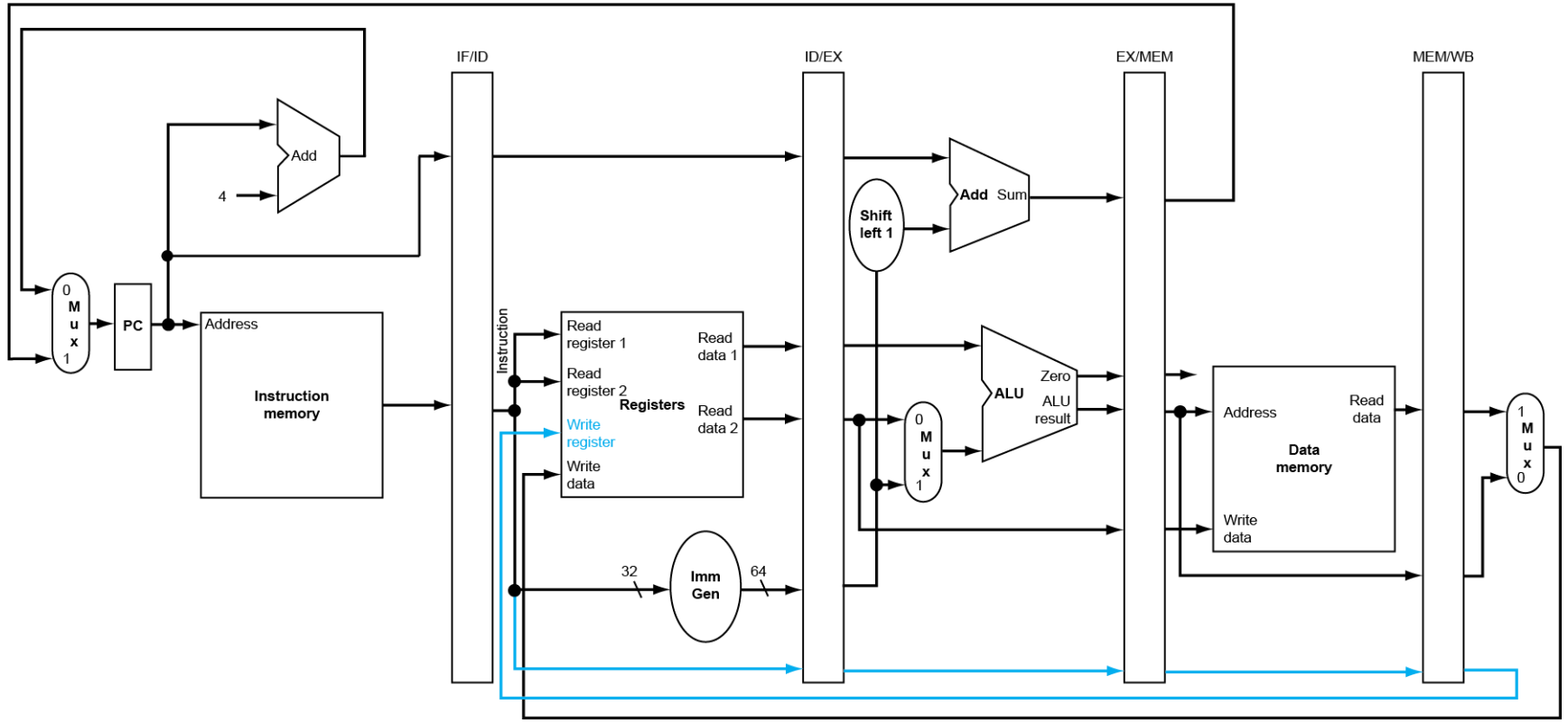
WB for Load



The instruction in the IF/ID pipeline register supplies the write register number, yet this instruction occurs considerably after the load instruction!

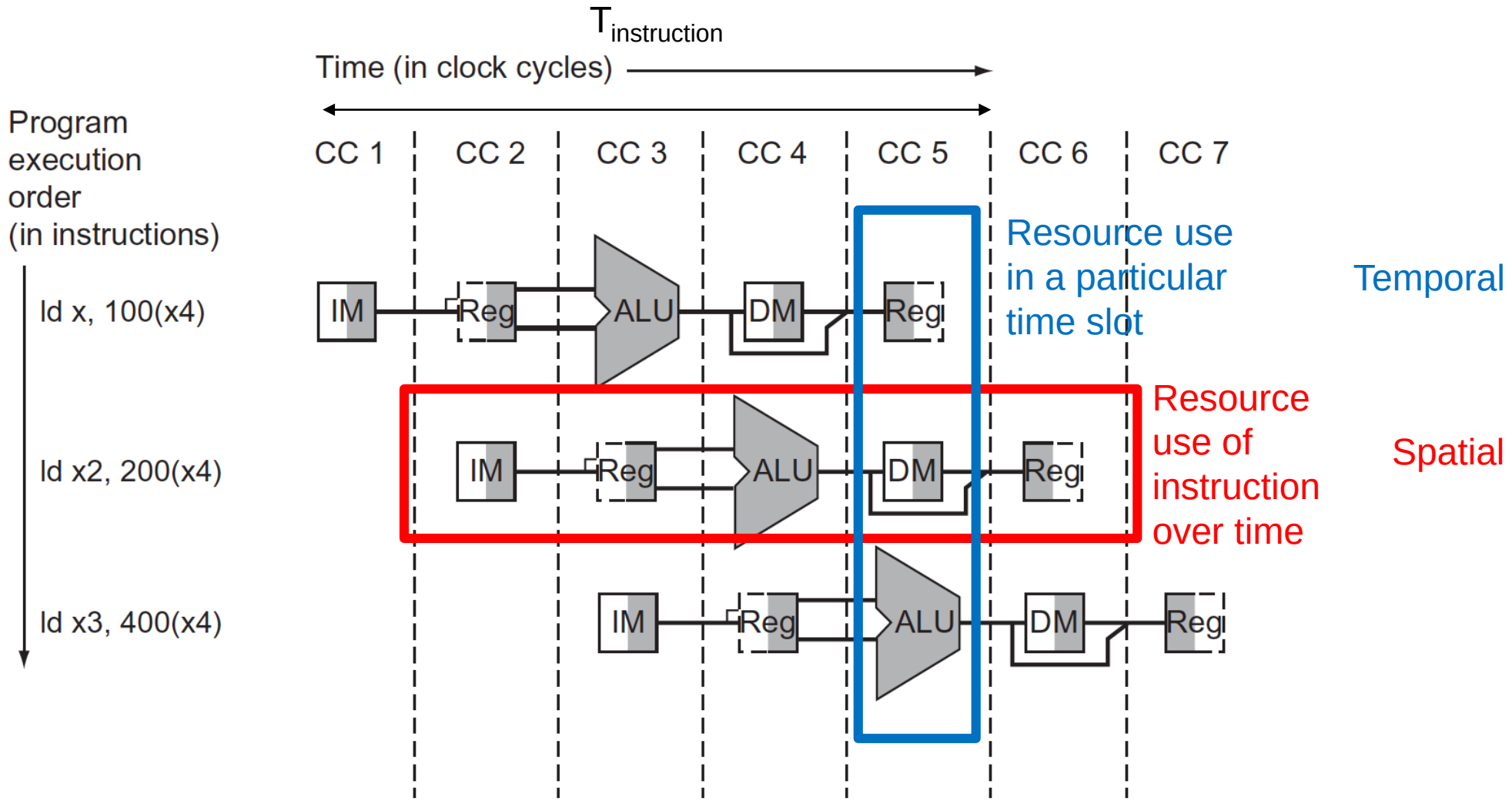
Corrected Datapath for Load

- Pass alive signals along through the pipeline
- Has to write/read register file at the same time
  - Writing reg in first half of clock
  - Reading reg in second half of clock



Taken from VE370, 2021 SU

Resource Usage



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:

Pipeline CPI = ideal pipeline CPI + pipeline stalls per instruction

= 1 + 1*0.20 + 3*0.05 = 1.35

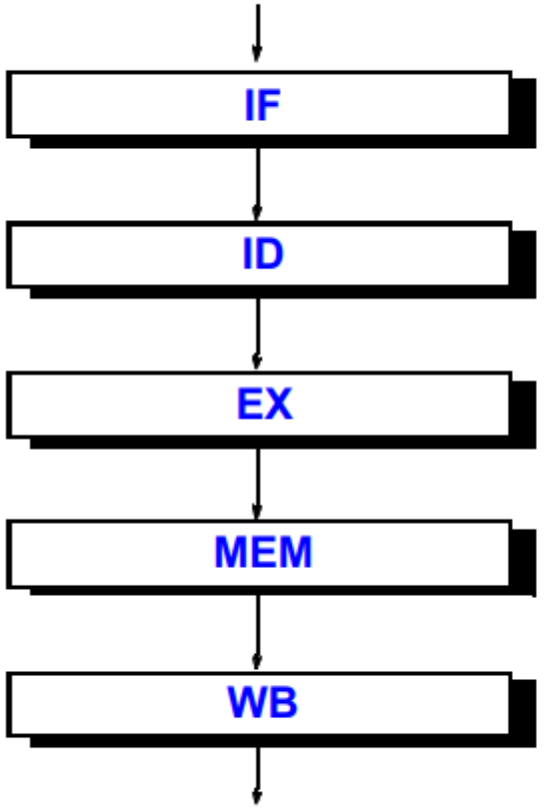
- *Note: to make the best use of pipelining, we should avoid stalling as much as possible. (without increasing our clock period!) Remember:*

Time = instructions executed×Clocks Per Instruction (CPI)×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!

Balancing pipeline?



T_{IF}= 6 units

T_{ID}= 2 units

T_{EX}= 9 units

T_{MEM}= 5 units

T_{WB}= 9 units

Without pipelining

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

= 31

Pipelined

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

= 9

Speedup= 31 / 9

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

© Wenisch 2016 -- Portions © Austin, Brehob, Falsafi, Hill, Hoe,Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykuma

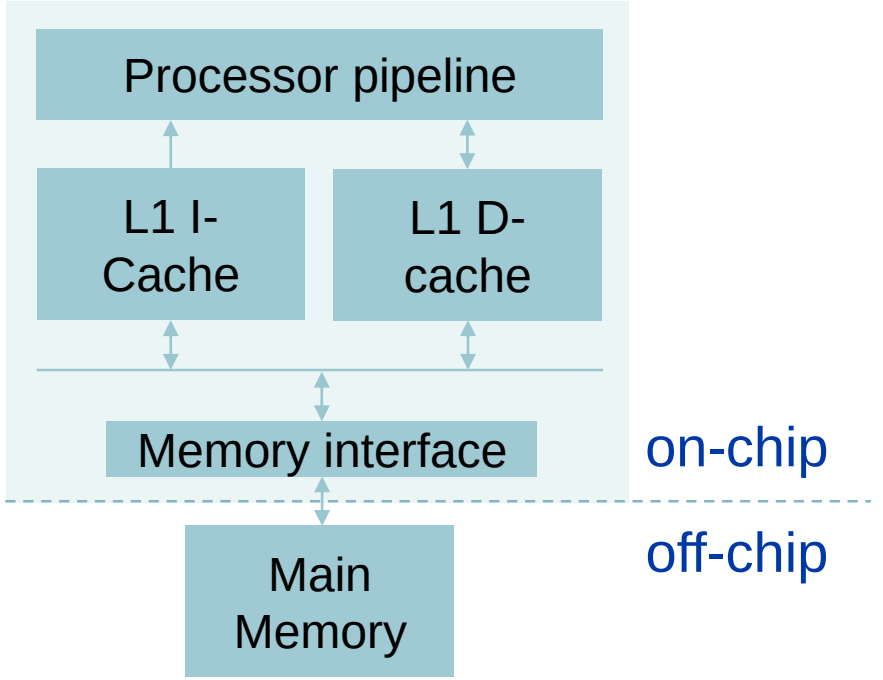
Balancing pipeline?

- 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)

© Wenisch 2016 -- Portions © Austin, Brehob, Falsafi, Hill, Hoe,Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykuma

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)



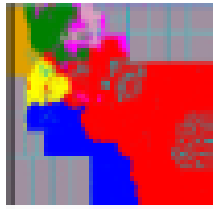
A single Arm Cortex A35 with 8K L1 instruction and data caches, no L2

For a 28 nm process:

Area: < 0.4 mm^2

Clock: 1 GHz

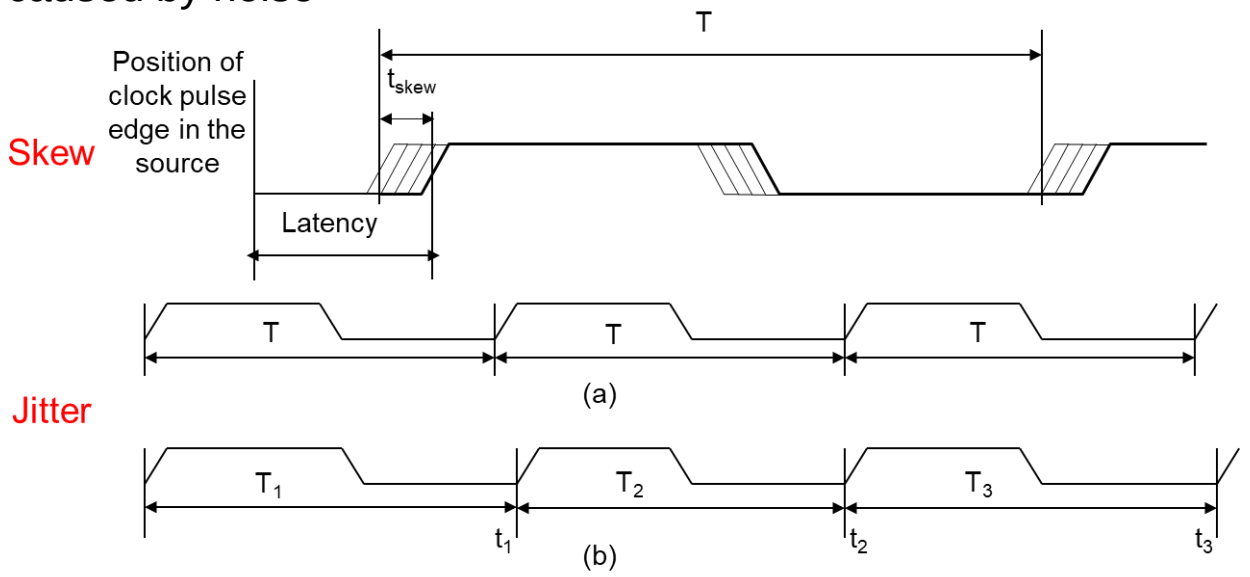
Power: ~90 mW



Figures: ARM

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

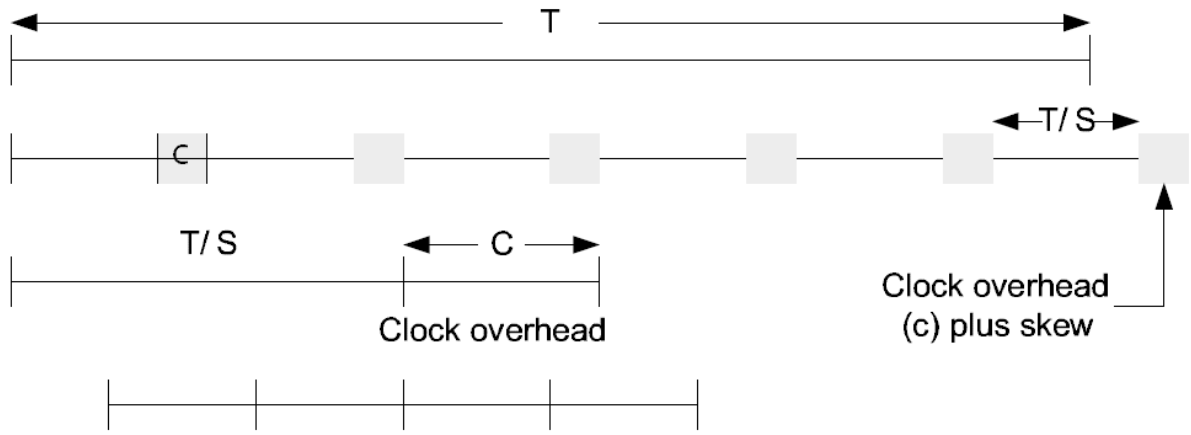


An Analytical Model of Performance

- let the total instruction execution without pipelining and associated clock overhead be T
- in a pipelined processor, let S be the number of segments

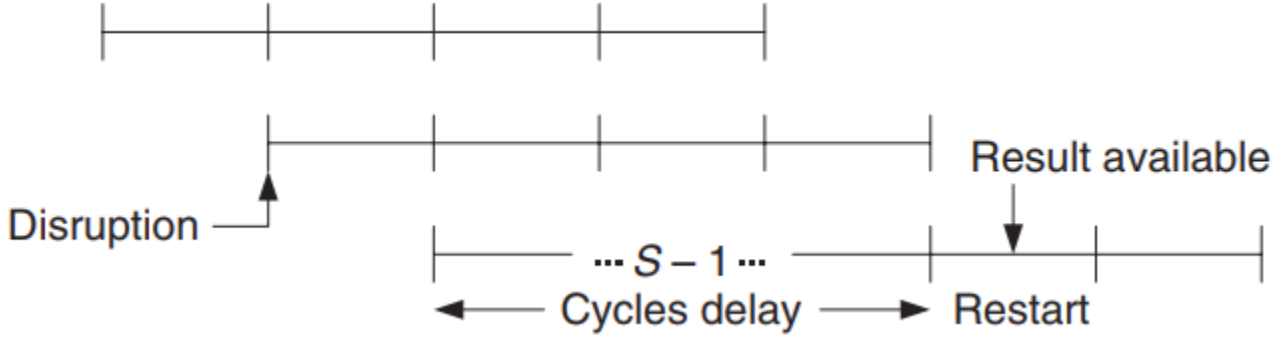
  S - 1 is number of cycles lost due to a pipeline break
- C = clock overhead incl. fixed skew

each stage carries the clock overhead (partitioning overhead)

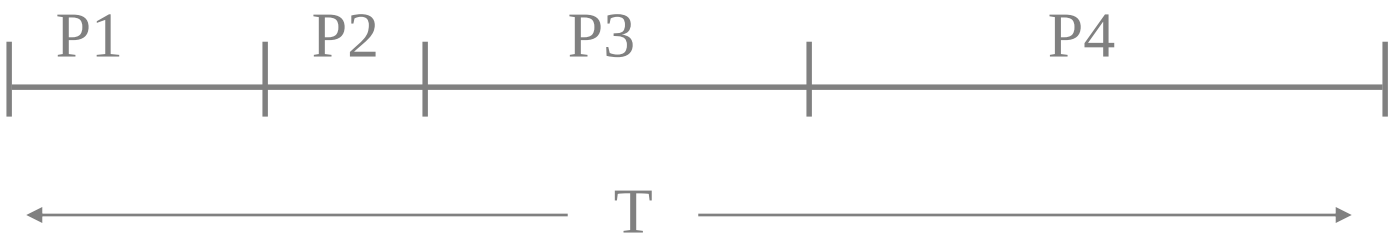


probability of break (b)

- 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 $T = \Sigma_i P_{max\ i}$ 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

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

performance = $1/(1+(S-1)b)$ [IPC]

throughput = G = performance / Δt [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:

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

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

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

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

b	k	T (ns)	C (ns)	Sopt	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

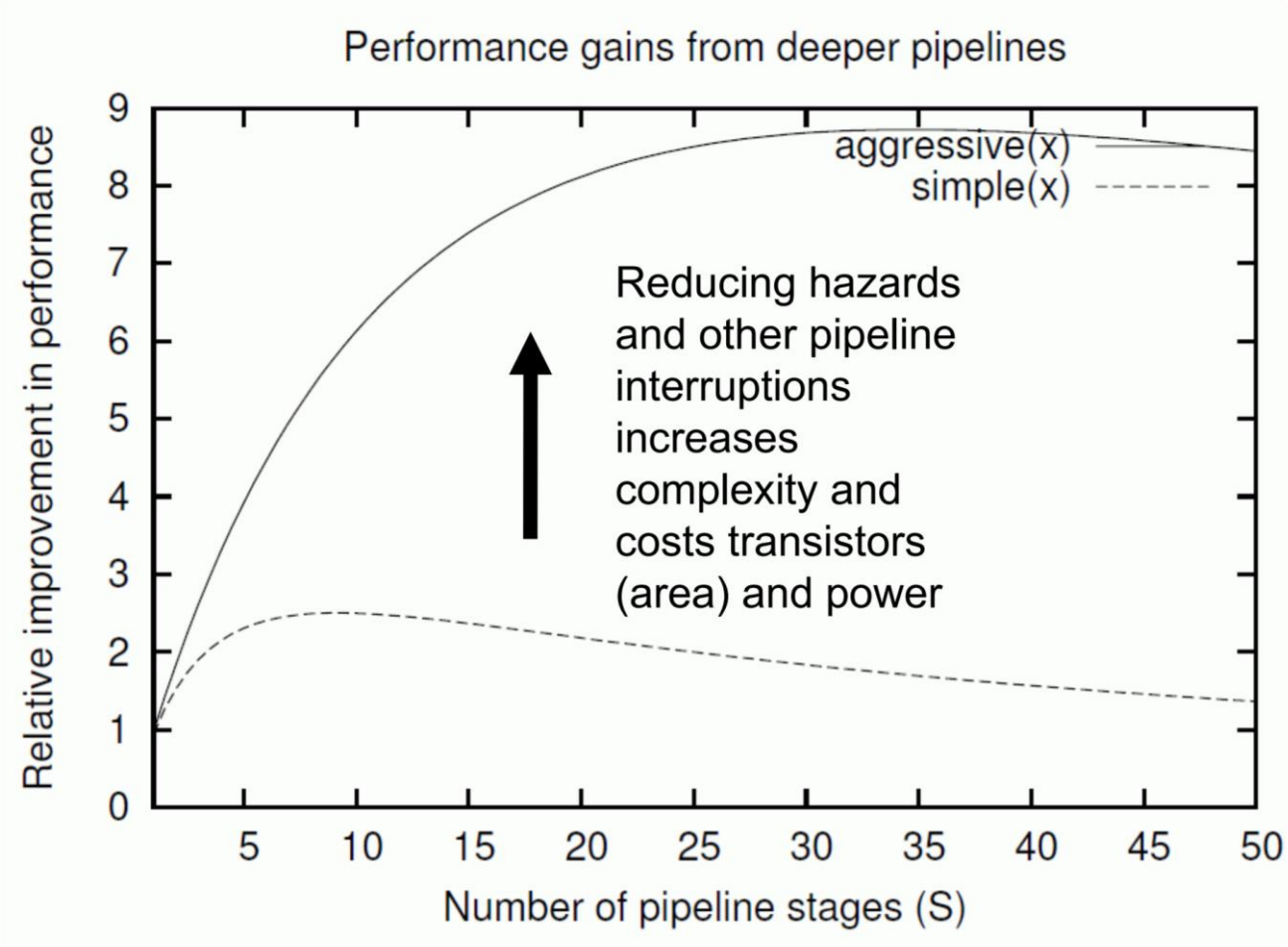
- quantization effects
  - T cannot be arbitrarily divided into segments
  - segments defined by functional unit delays
  - some segments cannot be divided; others can be divided only at particular boundaries
- some functional operations are atomic (can’t be interrupted)
  - cycle: usually not cross 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 ns, penalty of interruption is (S-1)
- **Simple pipeline design**
  - C= 300 ps
  - Pipeline interruption every 6 instructions
- **Aggressive pipeline design**
  - C = 100 ps
  - Pipeline interruption every 25 instructions



Source: Robert Mullins, University of Cambridge

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

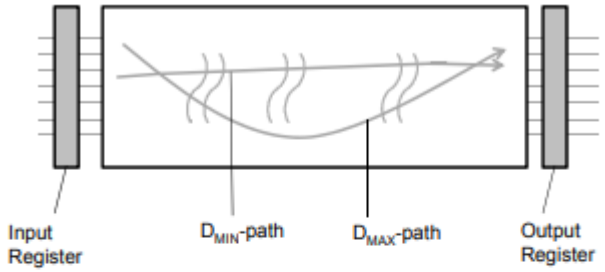
- High clock rates with small pipeline segments may (or may not) produce better performance.
- Two basic factors enabling clock rate advances:
  - Increased control over clock overhead.
  - Increased number of segments in the pipelines.
- **Low clock overhead (small *C*) *may cause higher pipeline segmentation,* 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?

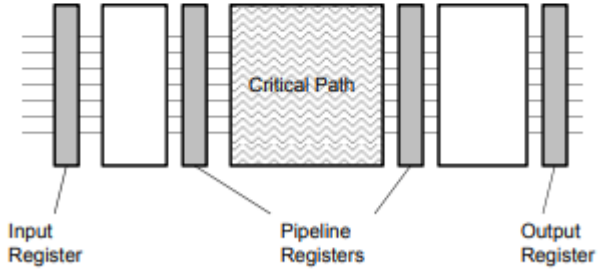
- asynchronous or self-timed clocking
  - avoids clock distribution problems, but has its own overhead
- multi-phase domino clocking
  - skew tolerant and low clock overhead; lots of power required and extra area
- wave pipelining
  - ultimate limit on Δt

$$\Delta t >= P_{max} - P_{min} + C$$



(a)

Wave-Pipeline



(b)

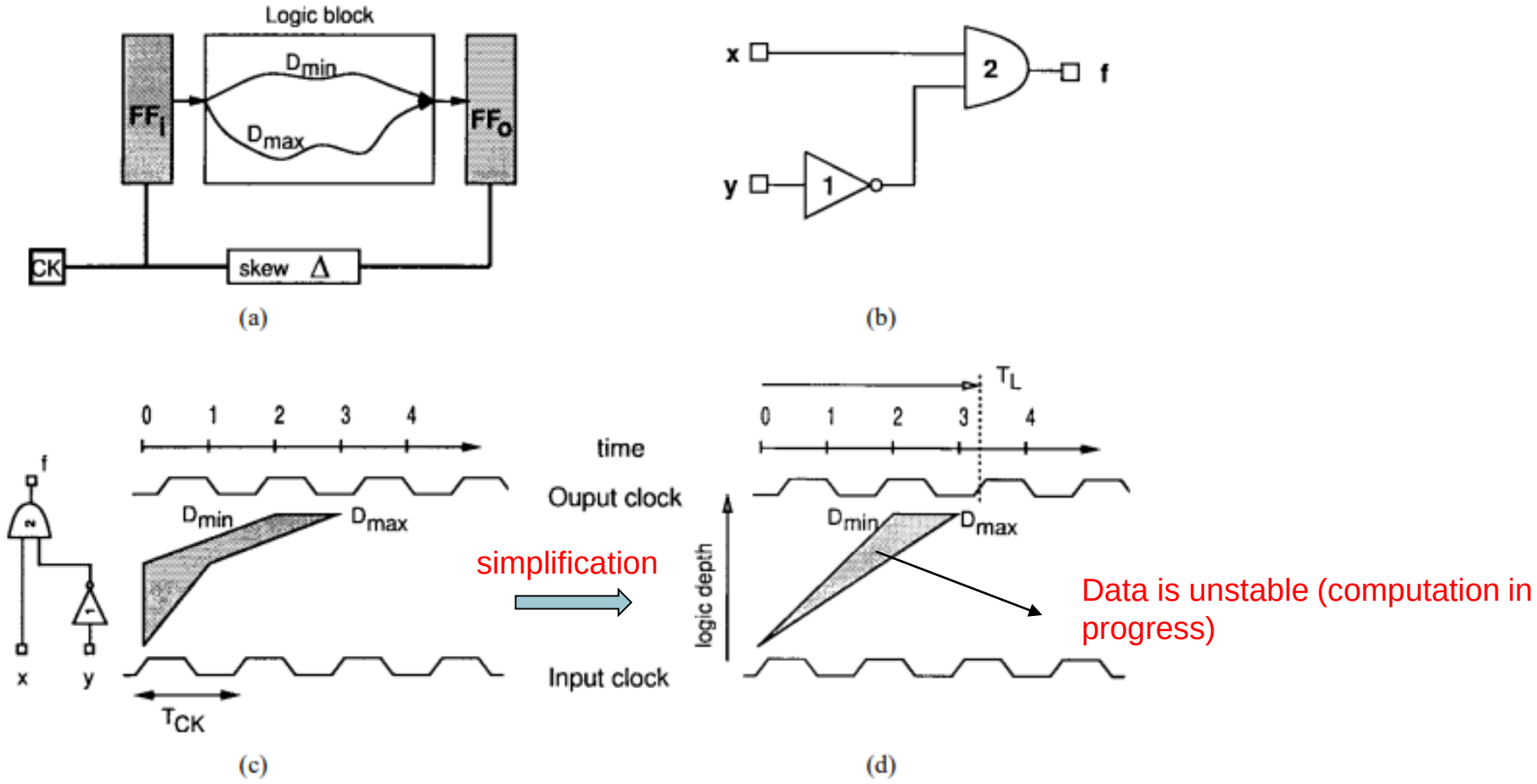
Conventional Pipeline

The original paper is available on Canvas.

https://www.cs.princeton.edu/courses/archive/fall2001/cs597a/wave.pdf

Taken from VE481, 2021 FA

Wave pipelining

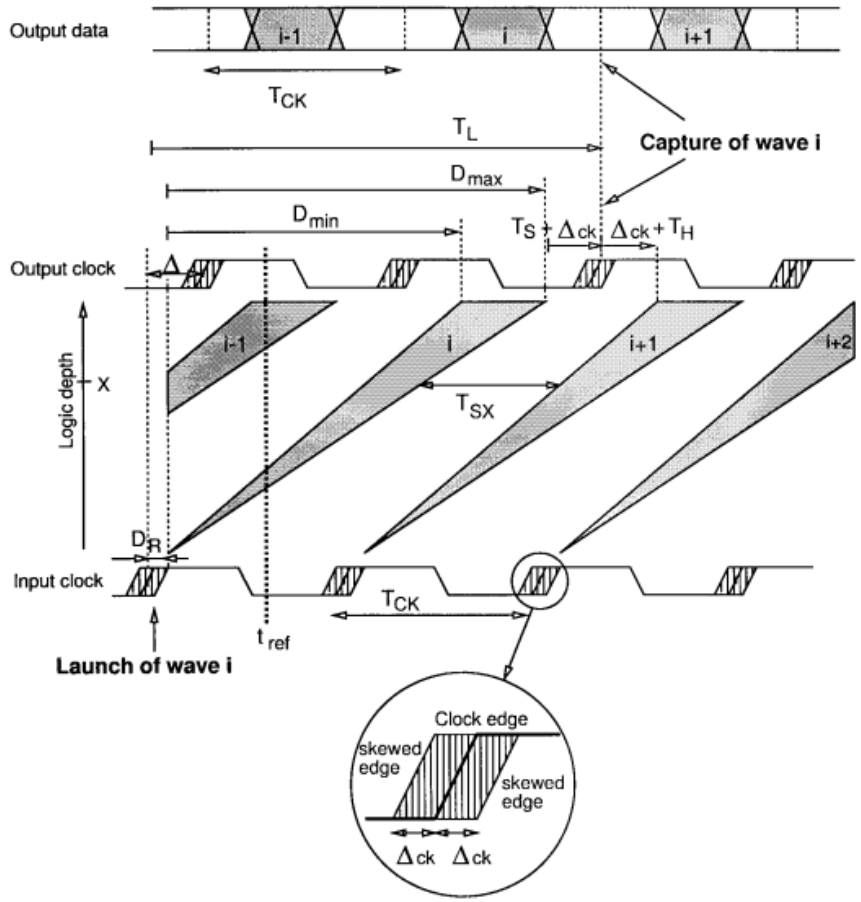


Taken from VE481, 2021 FA

https://www.cs.princeton.edu/courses/archive/fall2001/cs597a/wave.pdf

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.



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

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

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

clock overhead

https://www.cs.princeton.edu/courses/archive/fall2001/cs597a/wave.pdf

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!

Where are we Heading?

- T3: Fundamental Processors II
  - Hazards

Acknowledgement

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

- ARM Courseware
- Prof. Ron Dreslinski @ UMich, EECS 470
- Xinfei Guo@JI, VE370 2021 SU
- Xinfei Guo@JI, VE481 2021 FA
- Prof. Hakim Weatherspoon @ Cornell, CS 3410

Action Items

- Check the lab schedules and get ready
- HW#1 is upcoming
- Reading Materials
  - P&H, Computer Organization and Design RISC-V Edition, Ch. 4.5 – 4.6
  - Ch. 3.1
  - Wave pipeline paper