arm

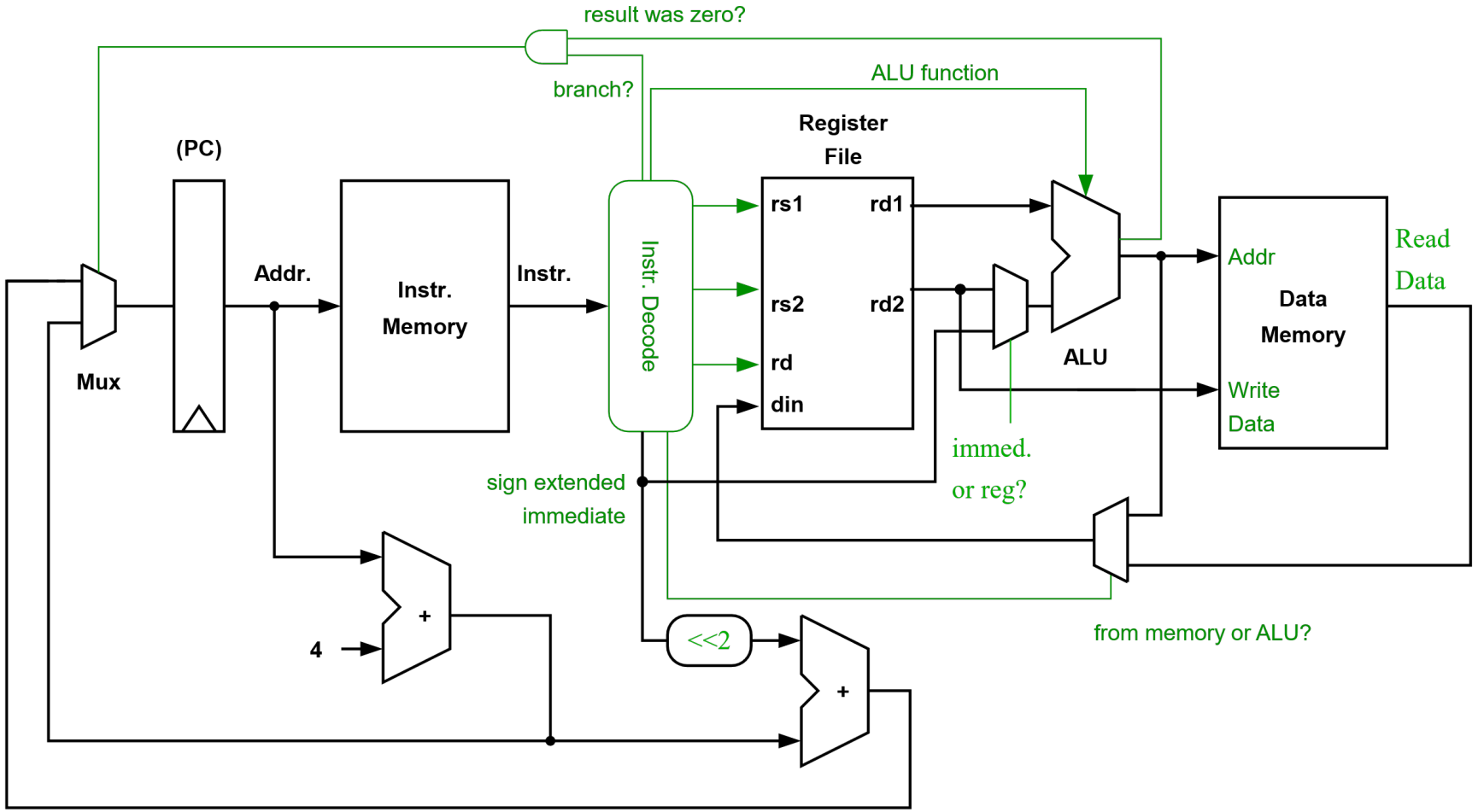
Pipelining

Module 3

Module Syllabus

- Pipelining and implementation
- Pipeline hazards and data dependency
- Pipeline and performance

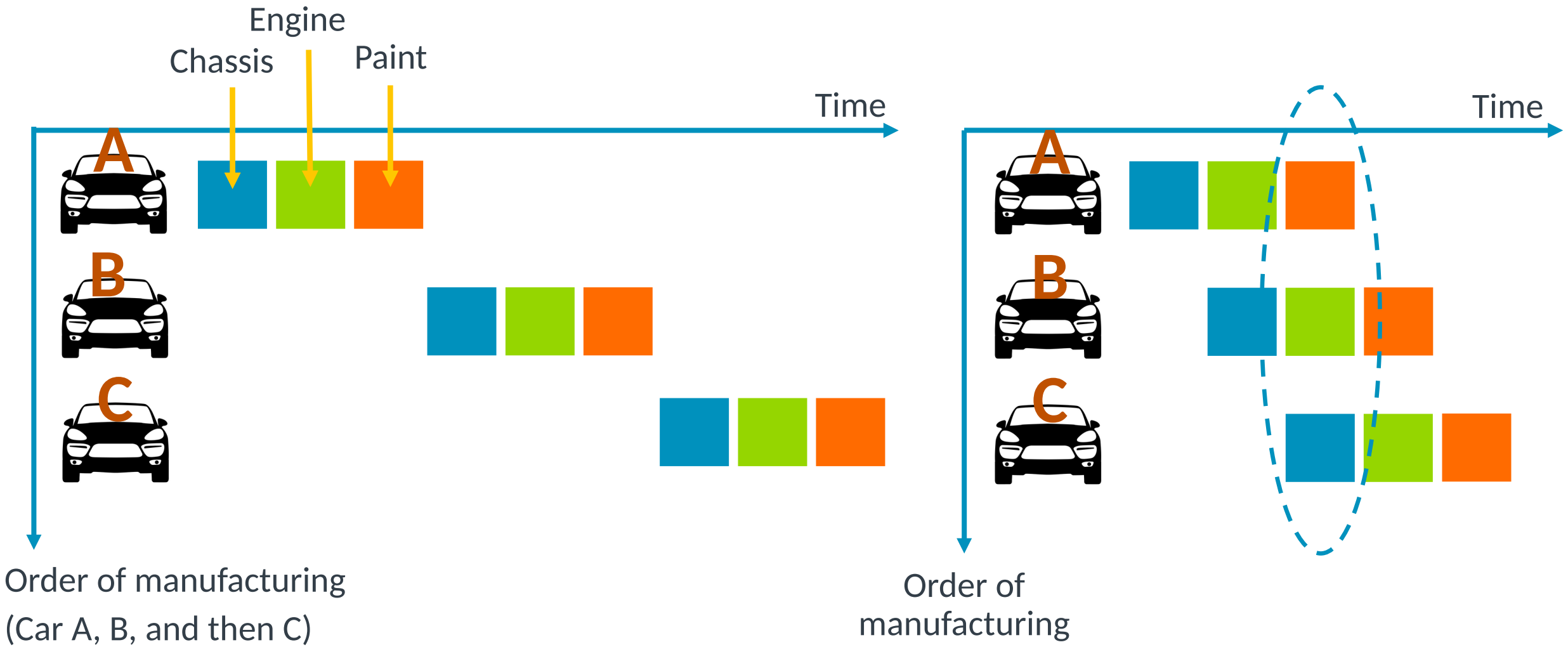
Our Simple Processor



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?



arm

What Is Pipelining?

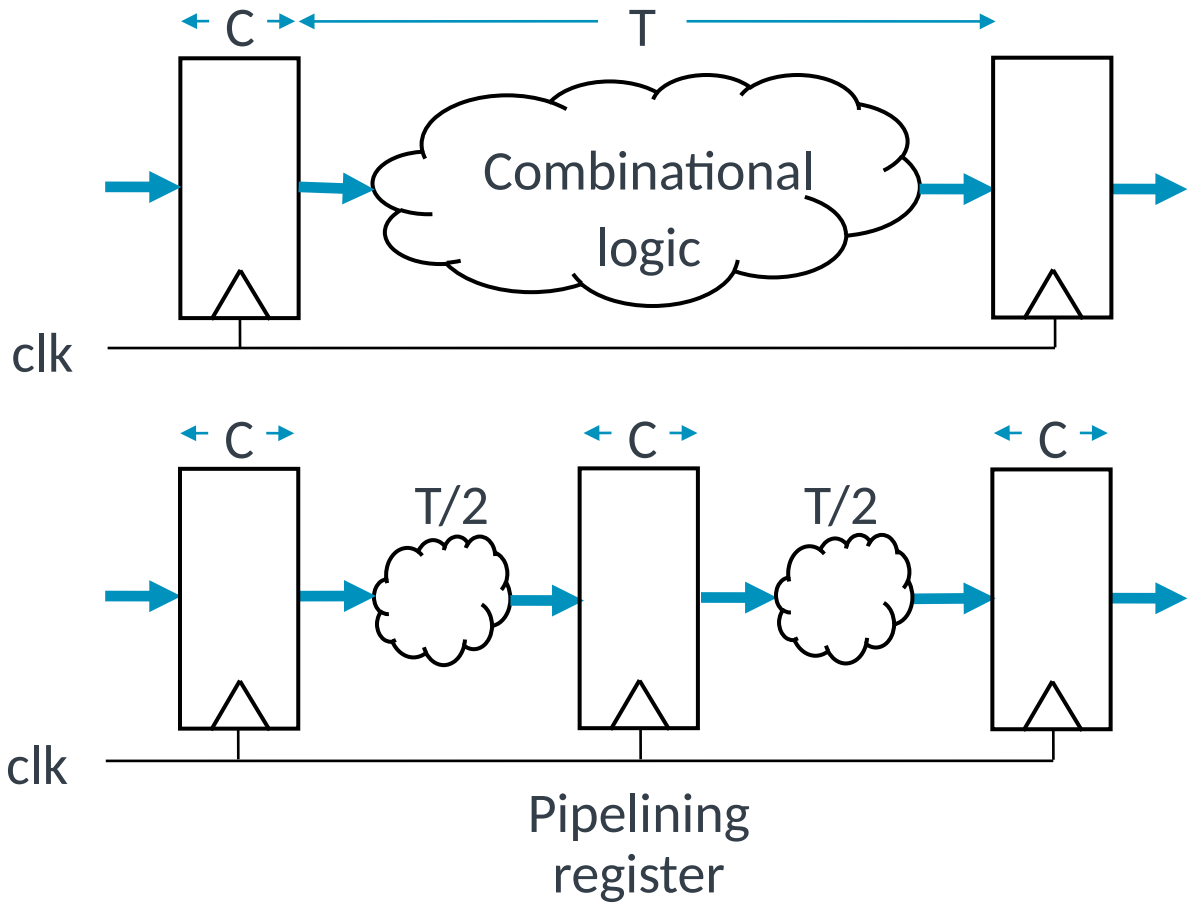
- We arrange for the different phases of execution to be overlapped. We aim to exploit “temporal” parallelism.
- How are latency and throughput affected?

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

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?

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.

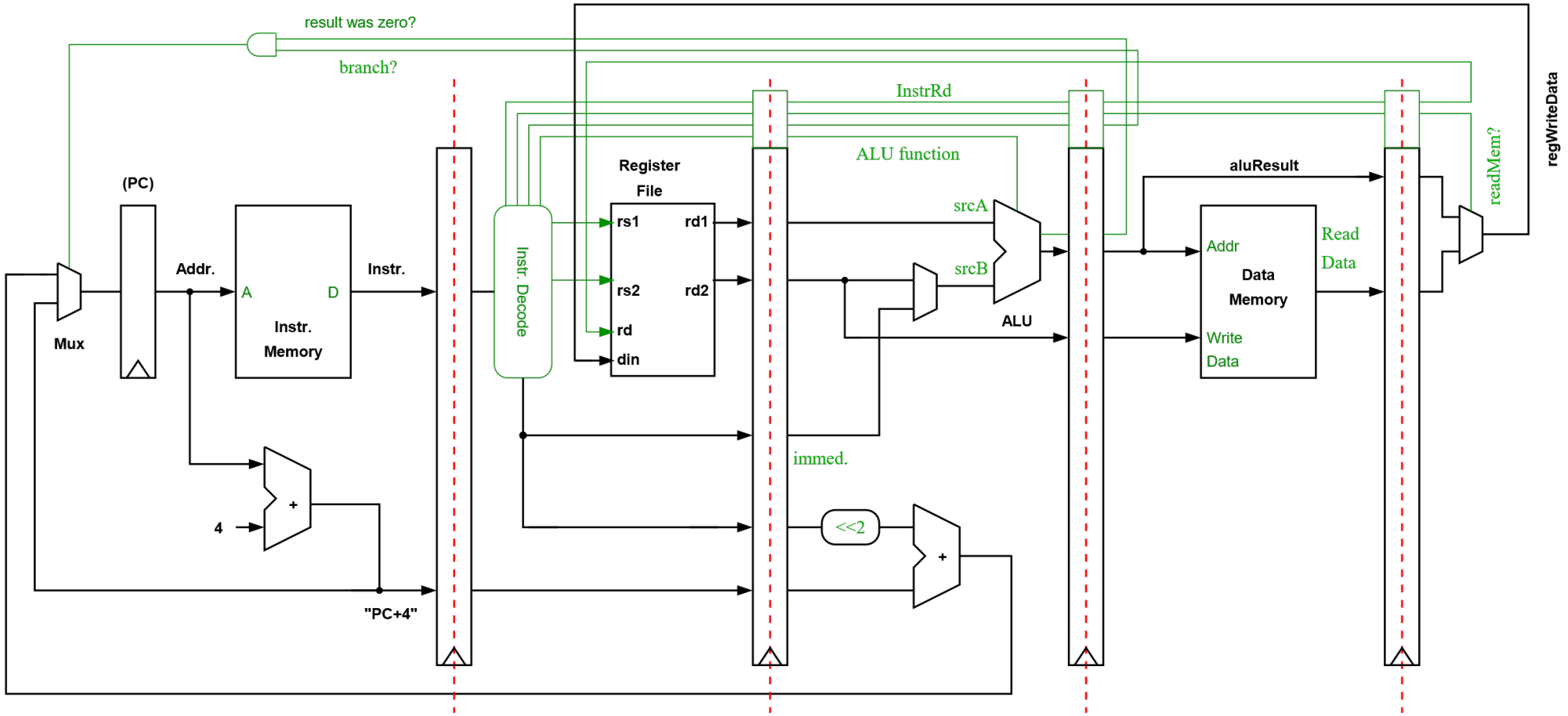
Hence, our **throughput** will double, too.

arm

Pipelining Our Processor

- We can insert **pipelining registers** into our processor to divide the logic into different pipeline stages.
- If added carefully, the worst-case delay between any two registers will be reduced.
- 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.

Pipelining Our Processor

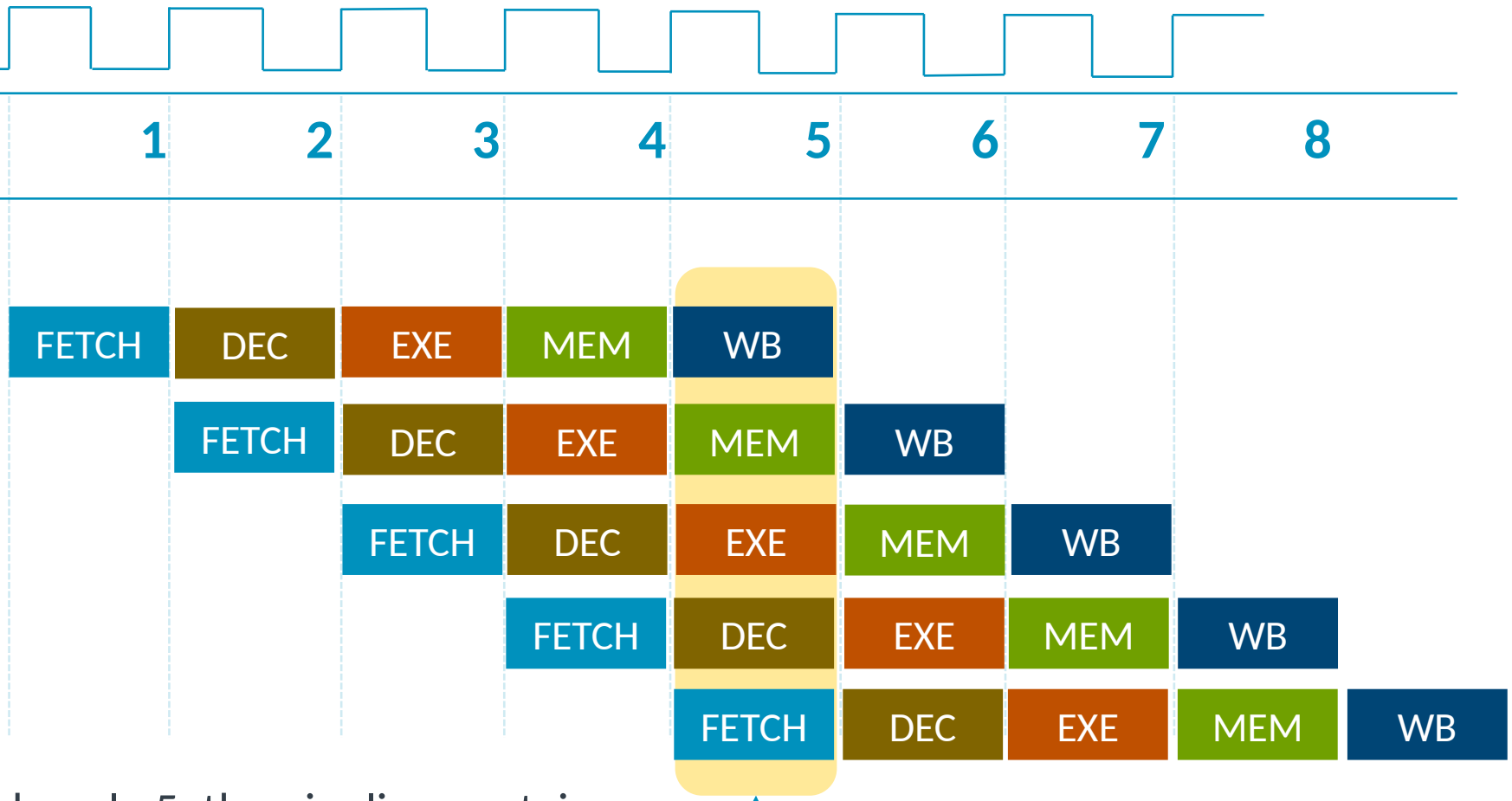


Pipelining Our Processor

- We've simply taken our original datapath and added pipelining registers.
  (Note: its behavior is now different to our unpipelined processor – we'll revisit this)
- This has created a 5-stage pipeline:
  - **FETCH** – access our instruction memory.
  - **DECODE** – decode our instruction and read the source registers.
  - **EXECUTE** – perform an ALU operation, calculate a memory address, or compute a branch target address.
  - **MEMORY** – access our data memory.
  - **WRITEBACK** – write to the register file.

The Pipeline in Action

- Clock
- Cycle
- Instruction
- 1. LDR X1, [X2]
- 2. ADD X2, X2, X3
- 3. STR X4, [X5], #4
- 4. SUB X0, X0, #1
- 5. LDW X5, [X3]



On clock cycle 5, the pipeline contains all the instructions listed in different pipeline stages

arm

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

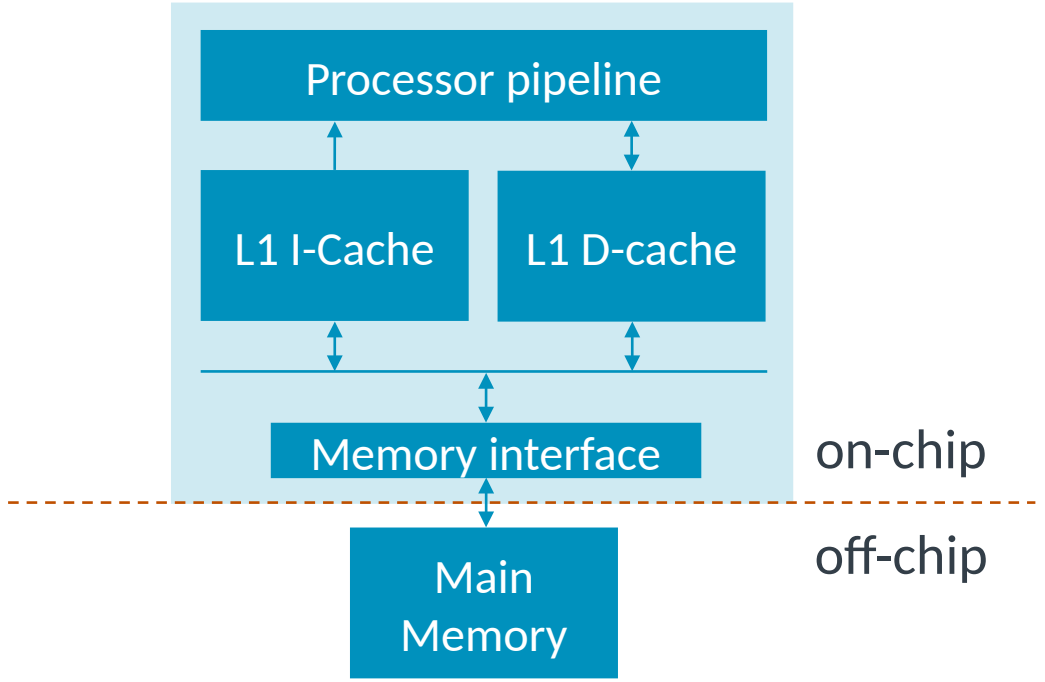
Note: to make the best use of pipelining, we should avoid stalling as much as possible. (without increasing our clock period!) Remember:

Time = instructions executed×Clocks Per Instruction (CPI)×clock period

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.**



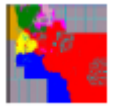
A single Arm Cortex A35 with 8K L1 instruction and data caches, no L2

For a 28 nm process:

Area: < 0.4 mm^2

Clock: 1 GHz

Power: ~90 mW



arm

Pipeline Hazards

Pipeline Hazards

- Pipelining allows new instructions to start while others are still in the pipeline, i.e., the execution of instructions is overlapped.
- There may be cases where an instruction must wait and not move forward in the pipeline to ensure correctness. These cases are known as pipeline hazards:
  - **Structural hazard** – arise from resource conflicts
  - **Data hazard** – arise from the need to ensure we always respect inter-instruction data dependencies
  - **Control hazard** – are caused by instructions that change the PC, i.e., branches and jumps (details in next module)

Structural Hazard

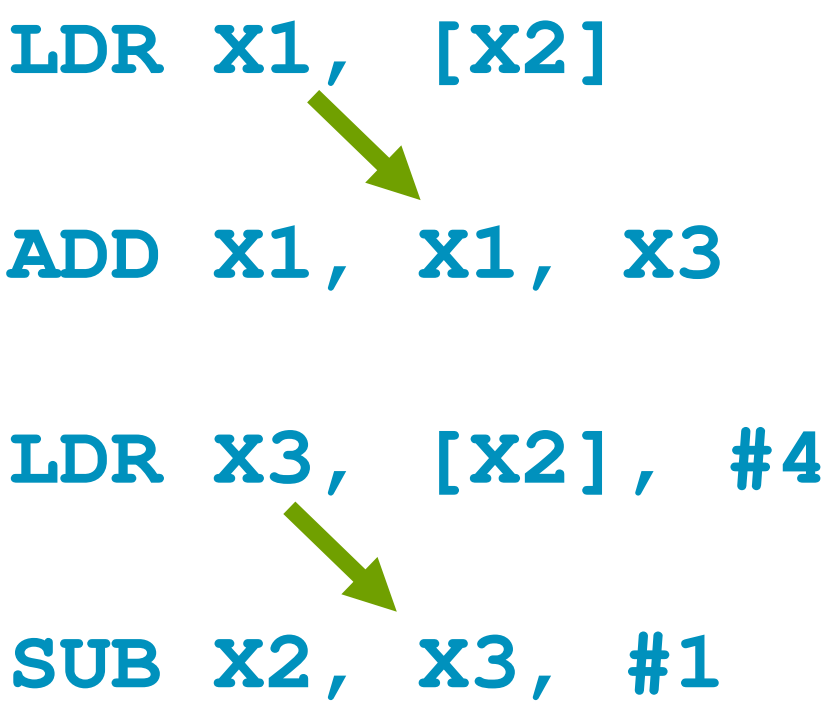
- Instructions may need to stall in order to wait for access to a shared resource, e.g.:
  - A functional unit that is not pipelined
  - A register file read port or register file write port
  - A memory
- Why permit any structural hazards?
  - Designing for the worst-case may reduce the average (common) case performance, i.e., the added complexity may reduce our CPI but increase our clock period.
  - Adding support for the worst-case may be too costly (e.g., in terms of power and area). We may have strict budgets or may want to use these limited resources elsewhere.

Data Dependencies – True Data Dependencies

True data dependence (green arrow). Also known as a **Read-After-Write (RAW)** dependence.

If we imagine two instructions, *i* followed by *j*

If *j* uses the result of *i*, we say that *j* is **data dependent** on *i*. This is an example of a true data dependence (a read after a write).

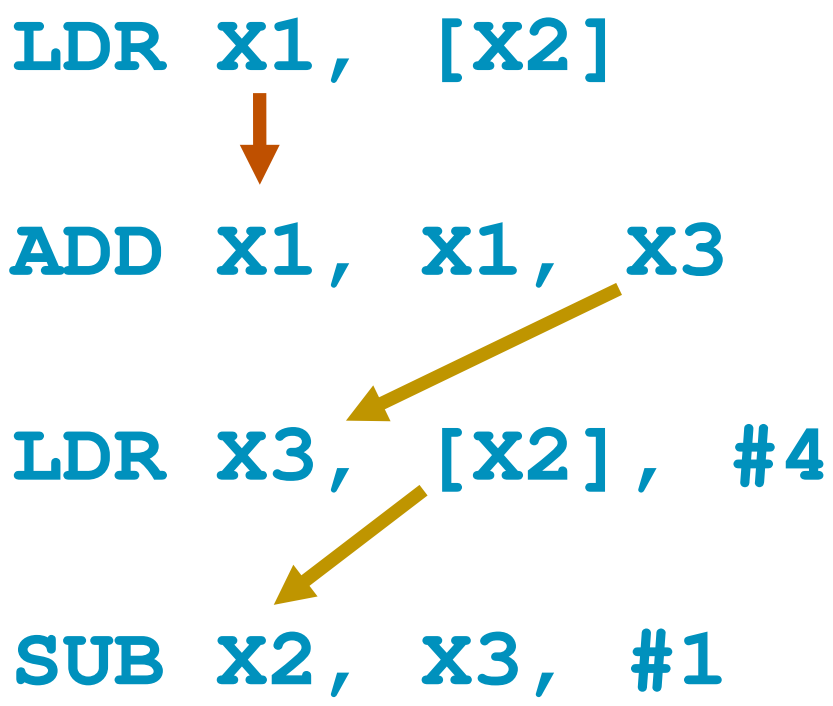


arm

Data Dependencies – Name Dependencies

Name dependencies may also exist when two instructions refer to the same register. Unlike true data dependencies, no data are communicated:

- Output dependencies (red arrow)
- Anti-dependence (gold arrow)



arm

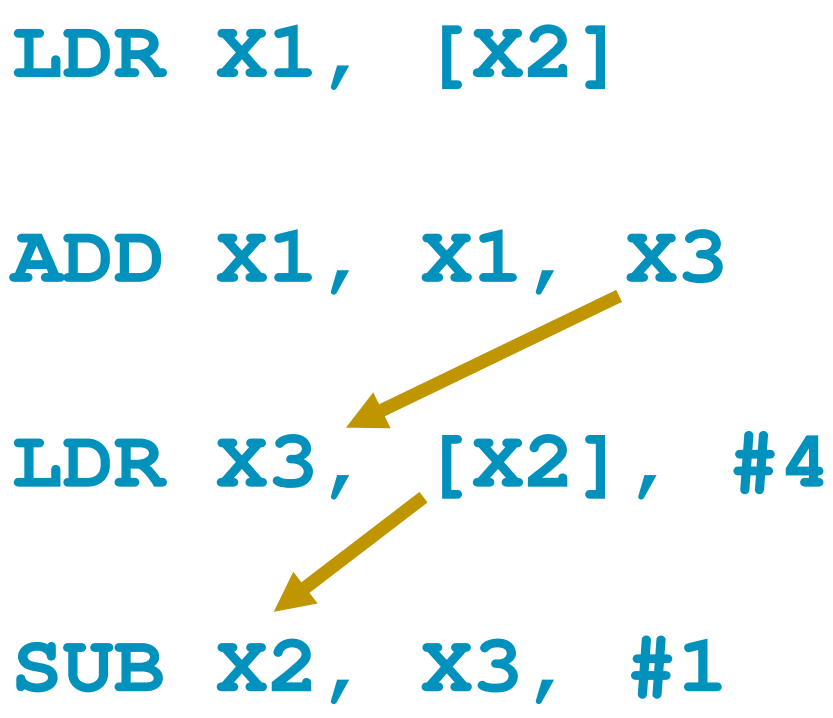
Data Dependencies – Name Dependencies

- **Output dependence** (red arrow). Also known as a **Write-After-Write (WAW)** dependence
- We need to ensure we don't reorder writes to the same register. This would mean subsequent instructions could receive the wrong data value.

arm

Data Dependencies – Name Dependencies

- **Anti-dependence** (gold arrows). Also known as a **Write-After-Read (WAR)** dependence
- Again, we need to be careful not to overwrite a register whose current value is still required by an earlier instruction.
- *E.g.*, we can't schedule the STR instruction before the ADD instruction.



arm

Data Hazards

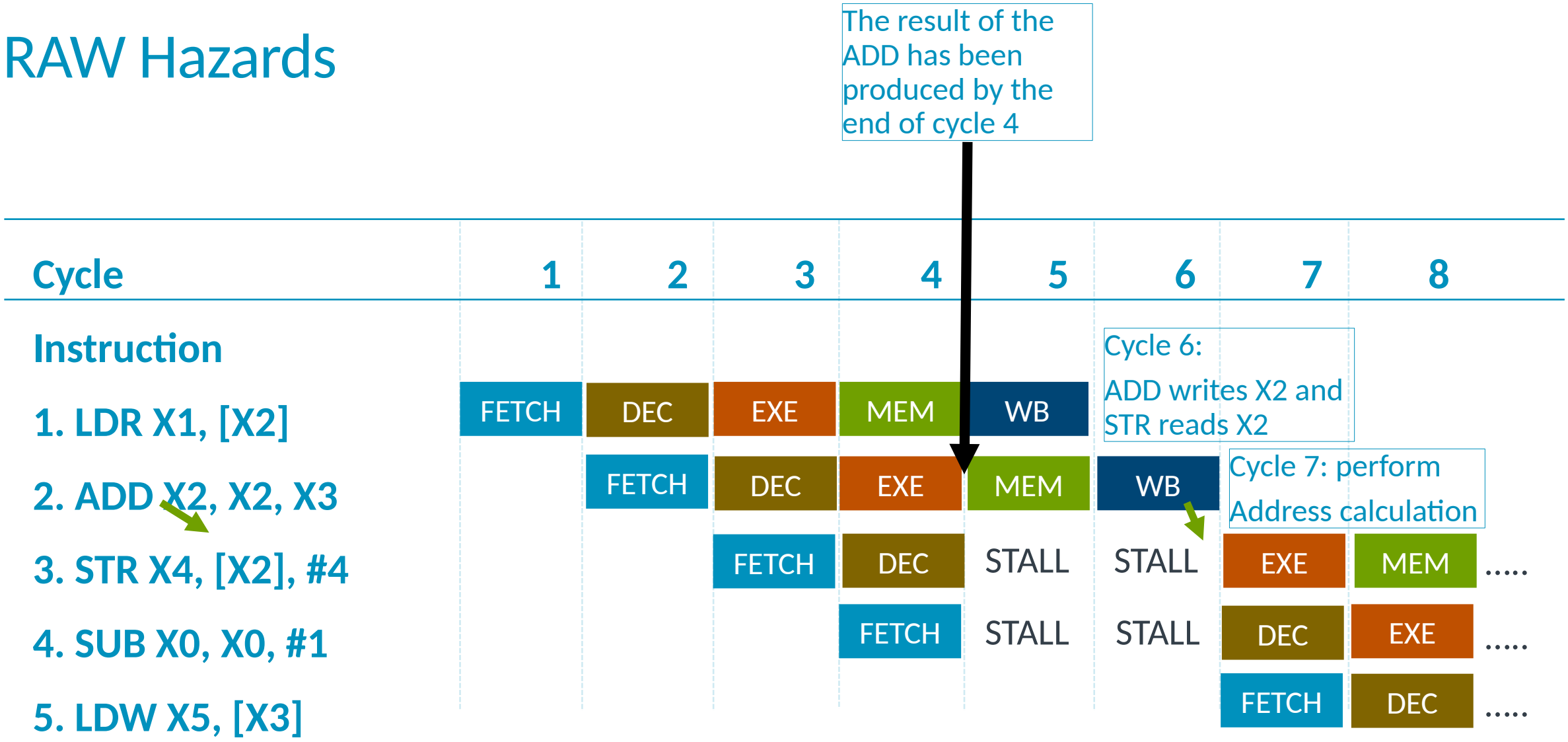
A data hazard is created whenever the parallel execution of instructions makes it possible for a dependency to be violated.

Read After Write (RAW) hazards – produced by true data dependencies, i.e., *j* attempts to read a source register before *i* writes to it.

Write After Write (WAW) hazards – produced by output dependencies, i.e., *j* tries to write to a destination register before it is written by *i*.

Write After Read (WAR) hazards – produced by anti-dependencies, i.e., *j* tries to write to a destination before it is read by *i*.

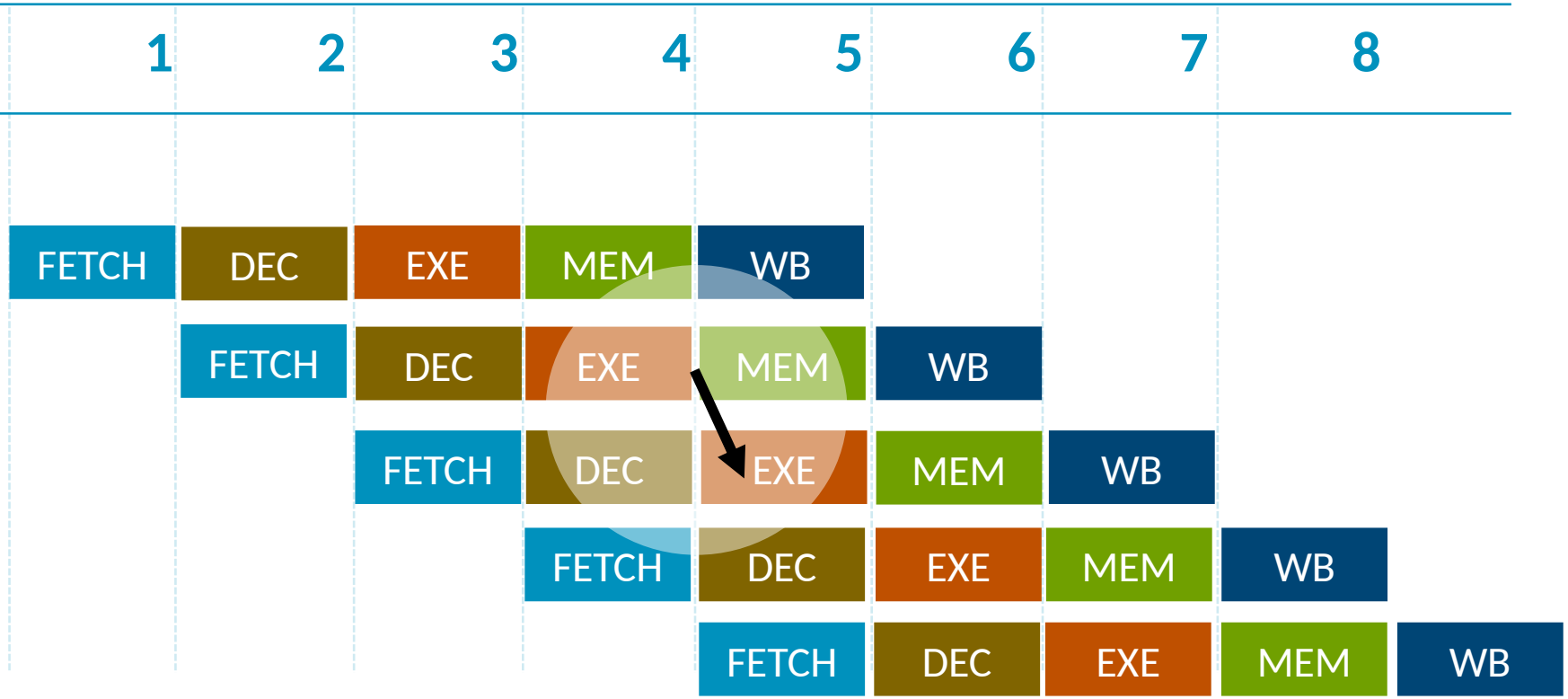
RAW Hazards



arm

Data Forwarding

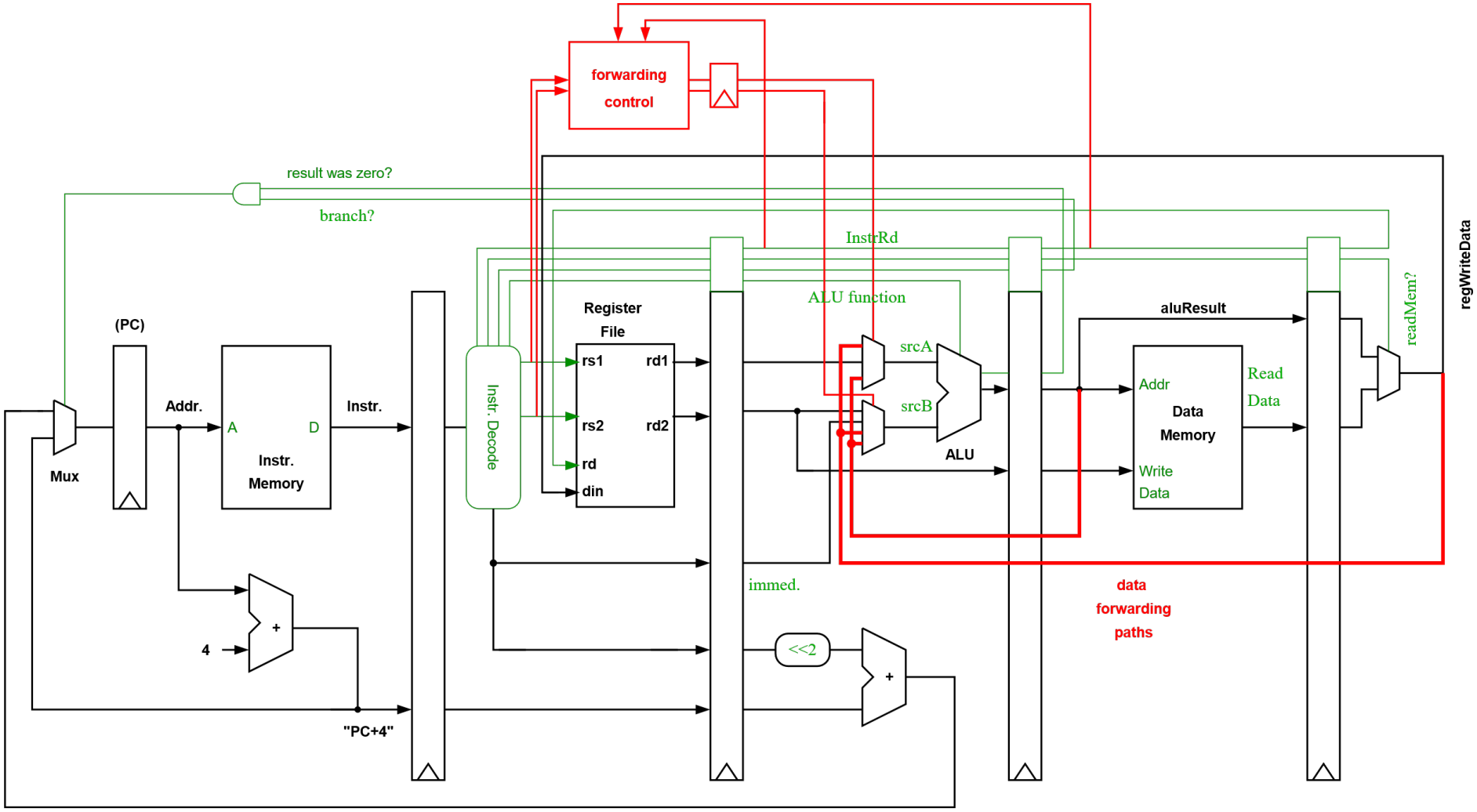
- Cycle
- Instruction
- 1. LDR X1, [X2]
- 2. ADD X2, X2, X3
- 3. STR X4, [X2], #4
- 4. SUB X0, X0, #1
- 5. LDW X5, [X3]



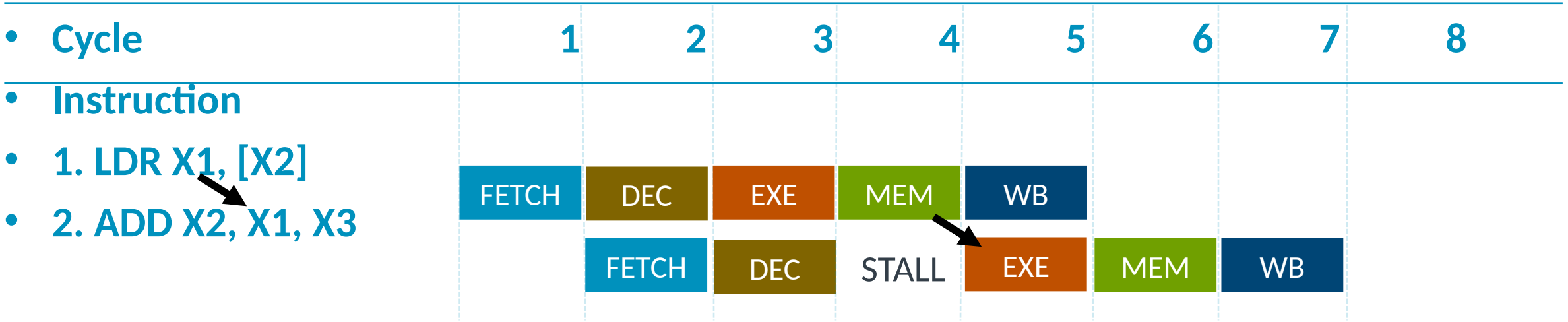
To avoid stalling due to RAW hazards, we must "forward" the result from the execute stage's pipeline register to the input of the ALU (rather than communicating via the register file).

arm

Data Forwarding



Load-use delay

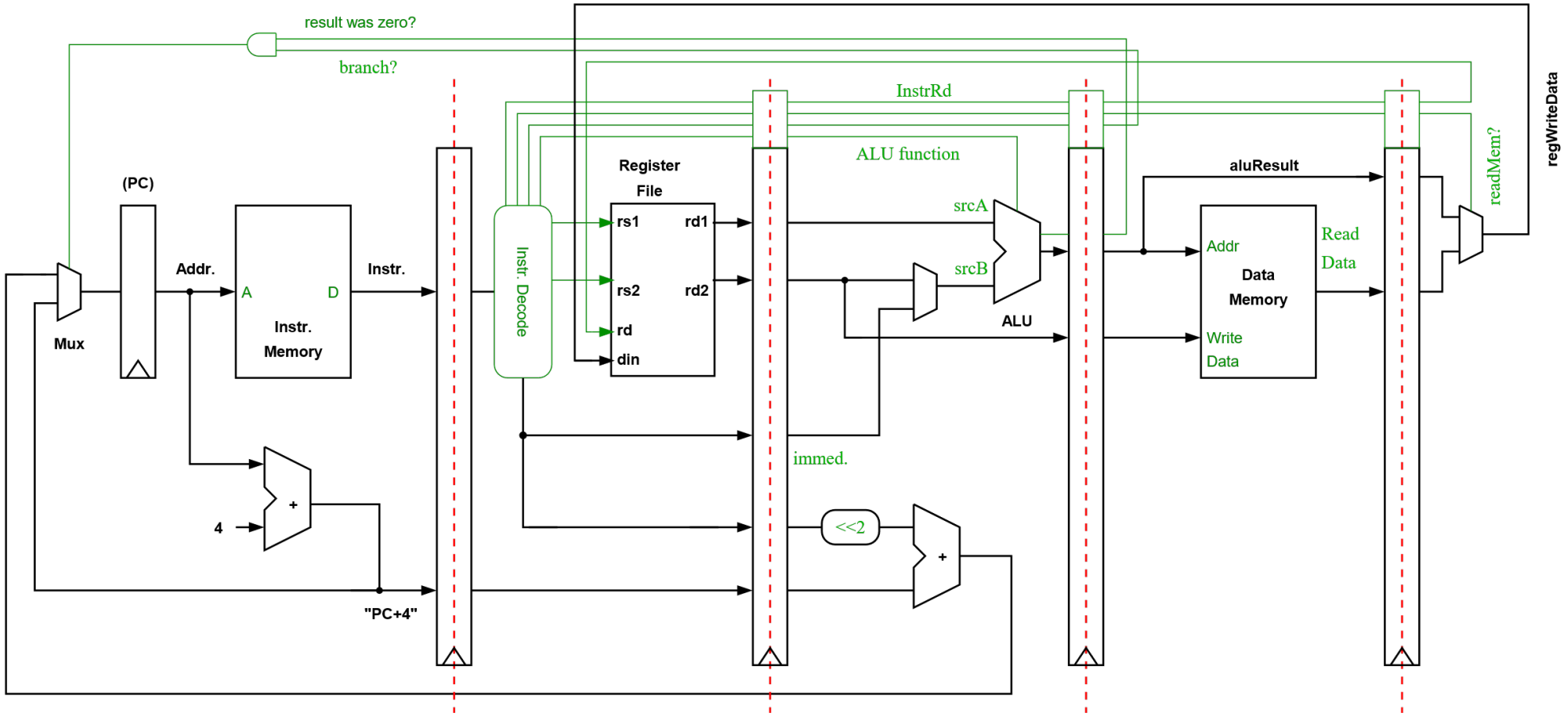


The LDR instruction loads data from memory in the “MEM” pipeline stage.

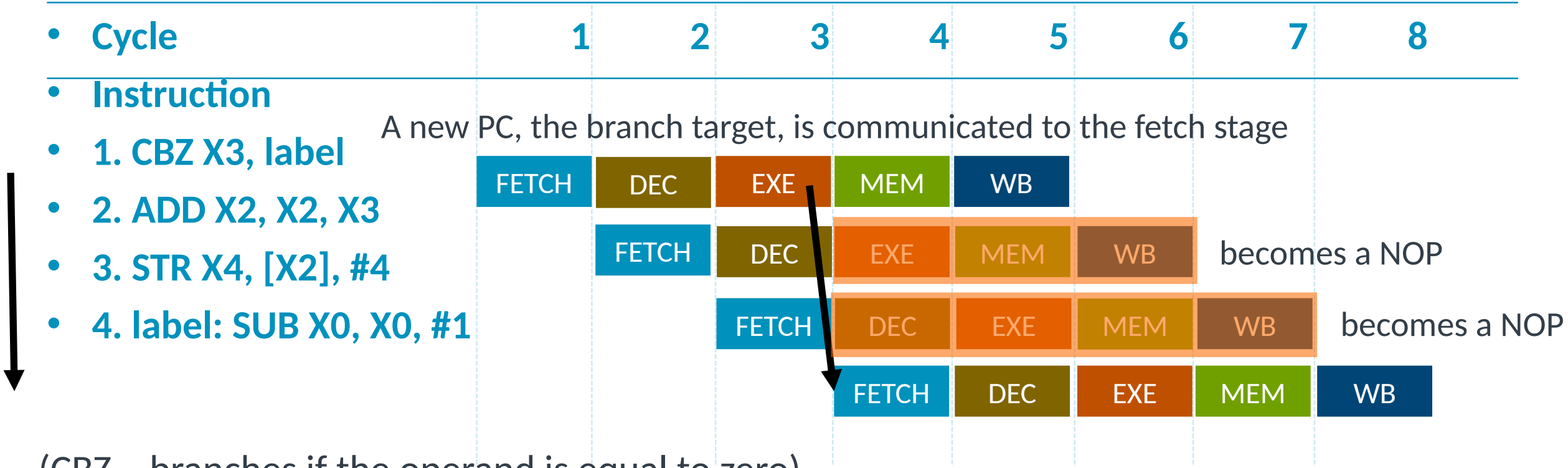
Even with data forwarding it is not possible to execute the LDR and ADD instructions at the same time. The ADD must be stalled.

arm

Control Hazards



Control Hazard



(CBZ – branches if the operand is equal to zero)

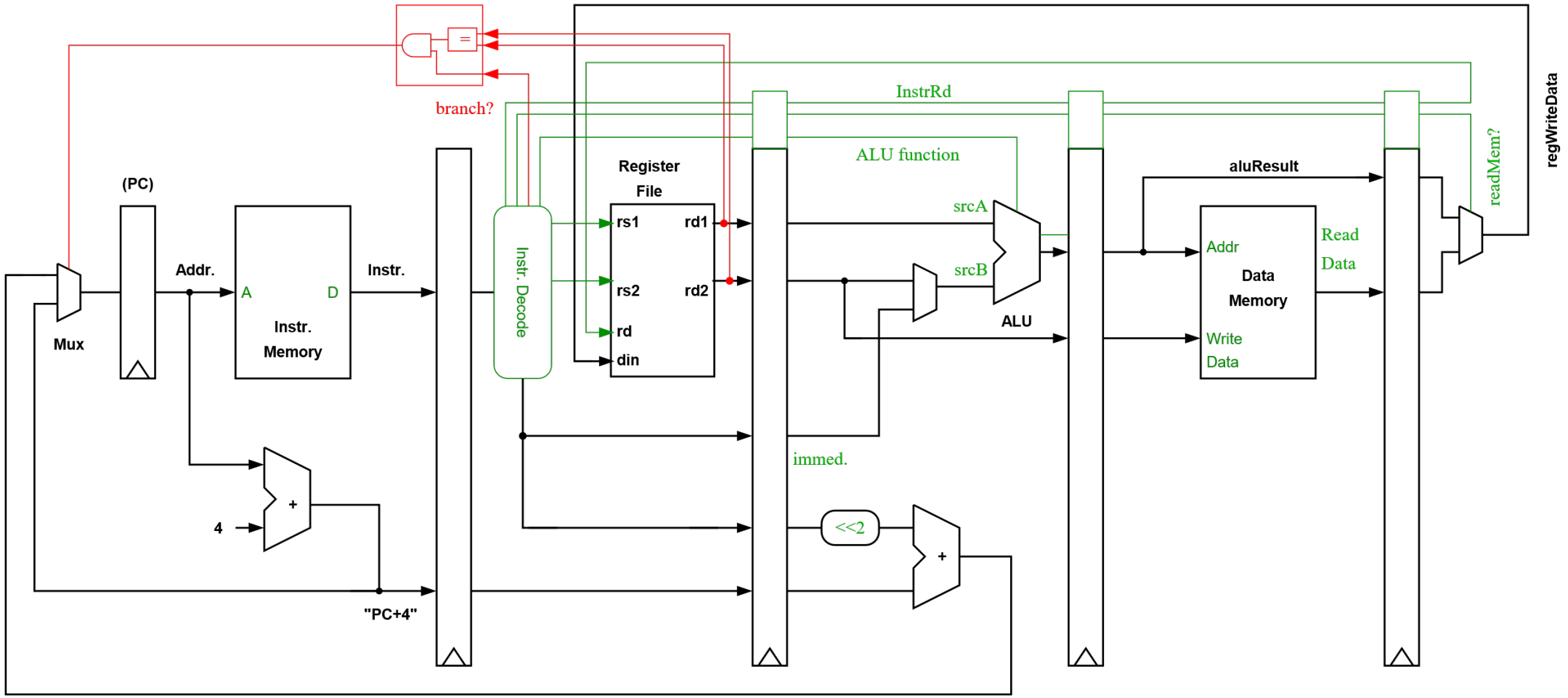
If the branch is evaluated in the execute stage, and it is taken, we must convert the two instructions that follow it into NOPs (we waste two cycles).

arm

Control Hazards – Evaluate Branch in Decode Stage

- To reduce the cost of branches, we could evaluate the branch in the decode stage.
- Now, a taken branch only involves a single “dead” cycle.
- Potential data hazards
  - If the instruction immediately before the branch writes to the register than the branch tests, we must stall for one cycle (i.e., until this instruction generates its result).
  - We will also need forwarding paths from the EXE and MEM pipeline stages to the decode stage.

Control Hazards – Evaluate Branch in Decode Stage



arm

Pipeline and Performance

Pipeline CPI

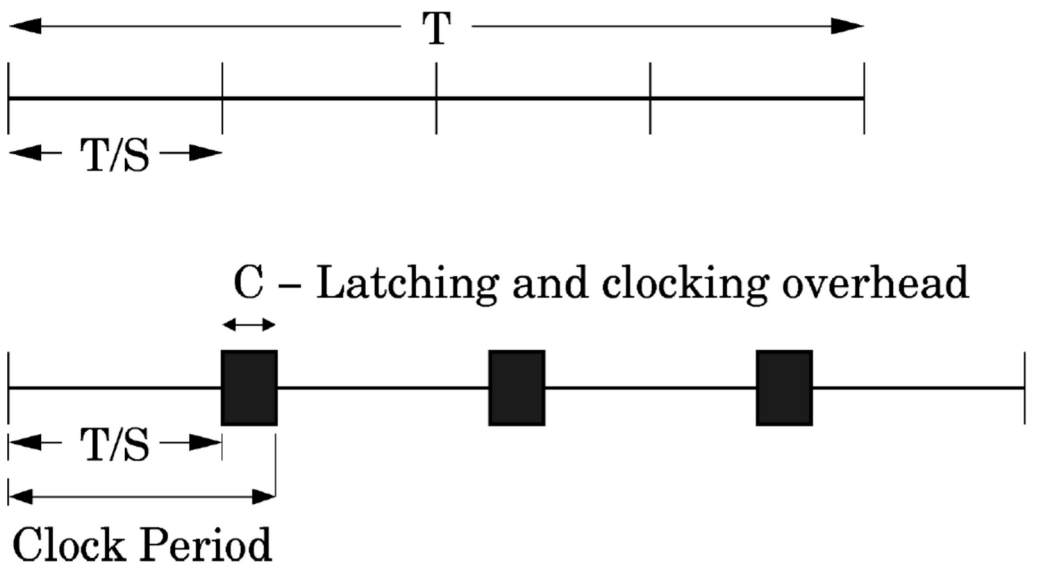
Pipeline CPI = ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

Stalls can be reduced by using a combination of compiler (e.g., scheduling) and hardware techniques.

Hardware techniques typically increase the area and complexity of our processor. Consequently, power consumption typically grows quickly (not just linearly) as we try to boost the performance of our processor.

An Analytical Model of Performance

- Let's construct a simple analytical model of pipeline performance.
- We start with a critical path of delay T.
- Divide it into S stages of delay T/S.
- We then add a clocking overhead C to our clock period, to give T/S+C.



arm

An Analytical Model of Performance

We can now create a simple analytical model of pipeline performance.

Pipeline CPI = ideal pipeline CPI + pipeline stalls per instruction

Freq = 1 / (clock period) = 1 / (T/S + C)

Throughput = Freq / CPI

Let's assume stalls occur at frequency b, and their cost is proportional to pipeline depth, say (S-1), now:

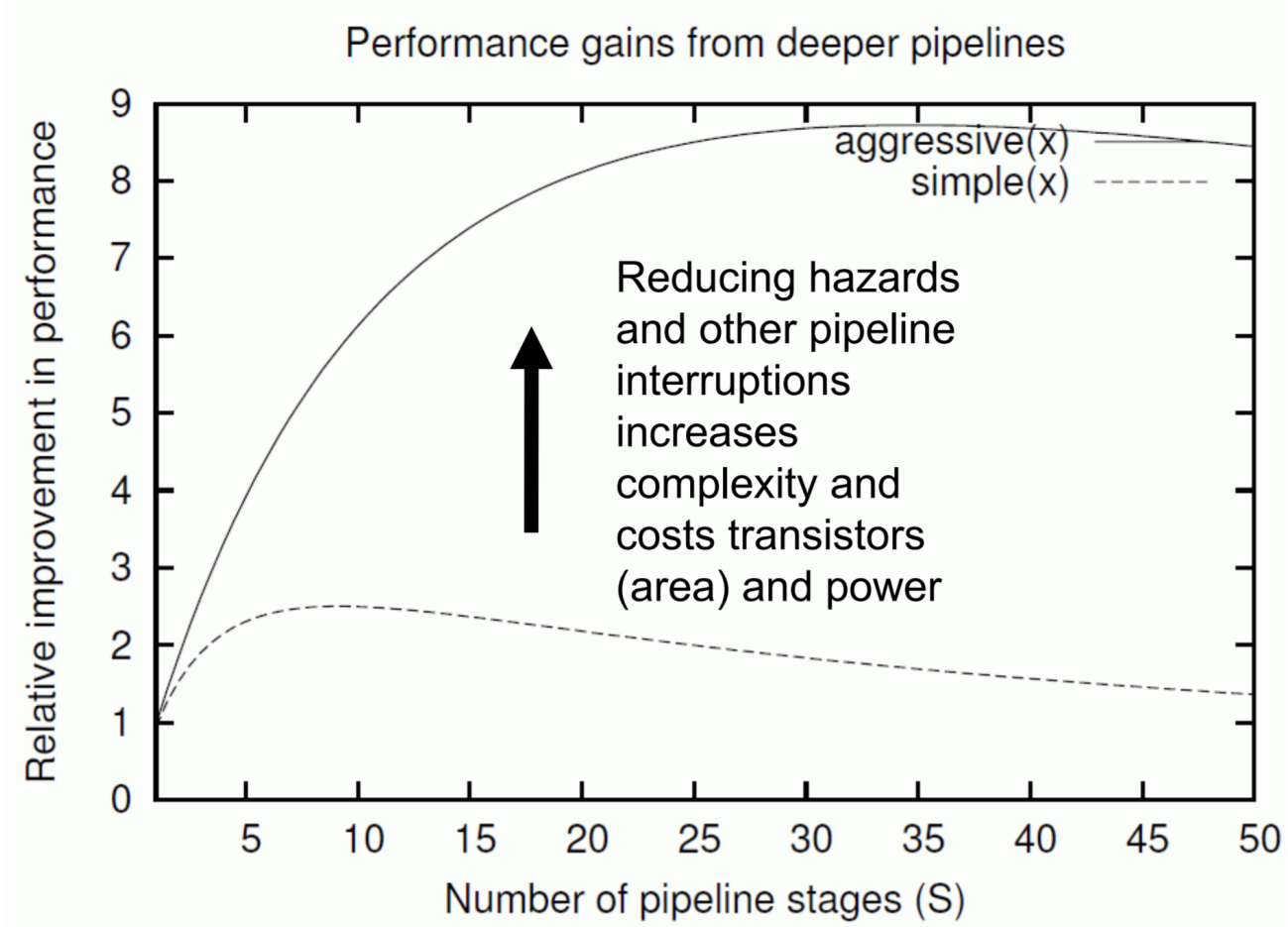
Throughput = 1 / (1+(S-1)b) x 1 / (T/S+C)

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

arm

Typical Pipeline Lengths

- 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?

Summary

Pipelining is often an effective and efficient way to improve performance. Of course, we must be careful that gains from a faster clock are not lost due to the cost of stalling the pipeline.

We must take care to:

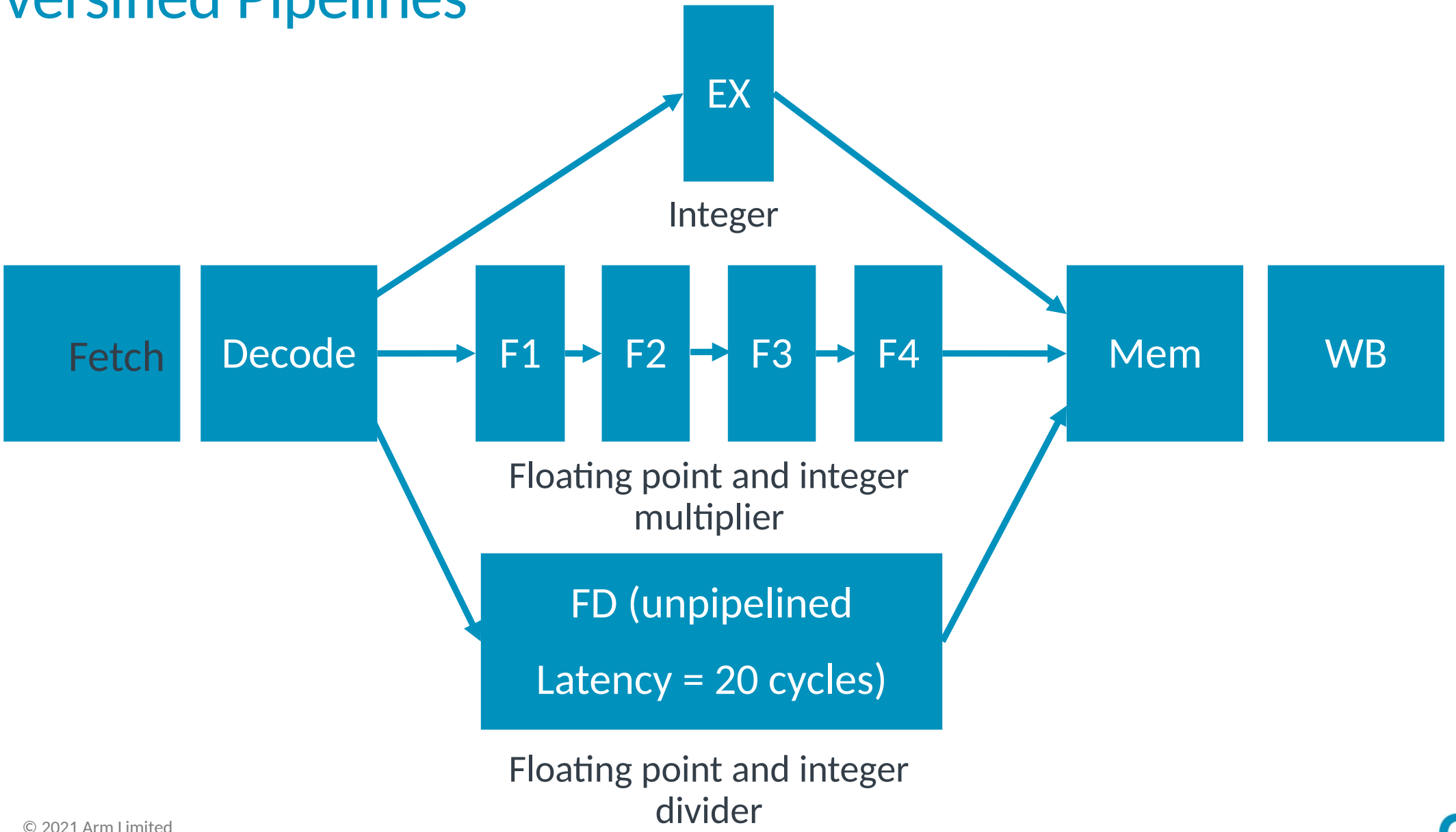
1. Supply instructions without stalling:
   - Ensure branches are handled efficiently in our pipeline.
   - Try to minimize our instruction cache miss rate.
2. Supply data – minimizing our data cache miss rate.
3. Minimize pipeline stalls due to structural hazards and data hazards.

Backup Slides:
Diversified Pipelines &
Arm10 Pipeline Case Study

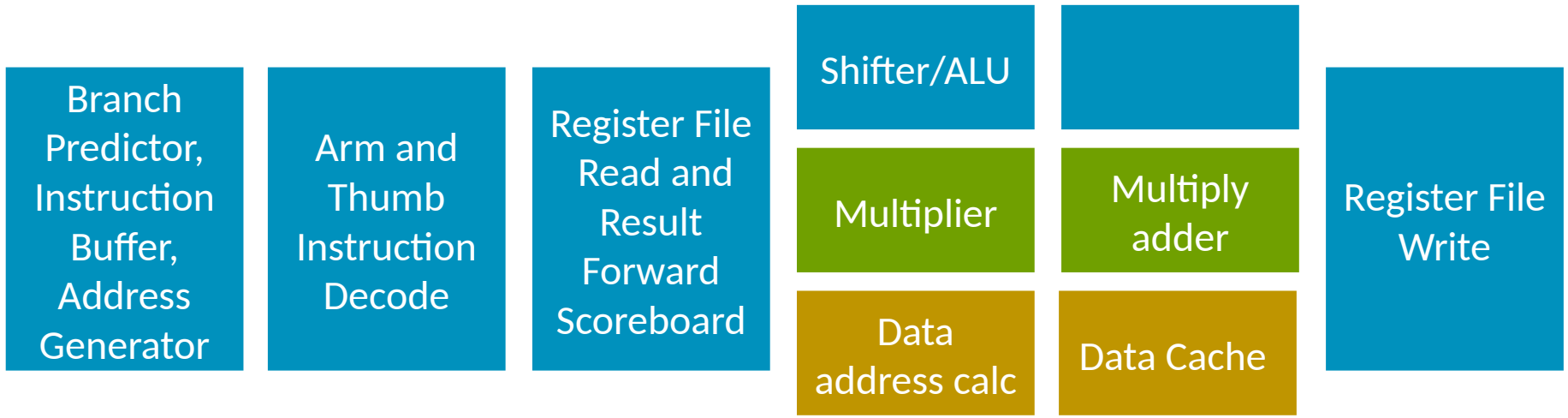
Diversified Pipelines

- It is impractical to require that all instructions execute in a single cycle.
- We also want to avoid sending all instructions down a single long pipeline.
- We can instead introduce multiple (or “diversified”) execution pipelines.
- Could this introduce new hazards?

Diversified Pipelines



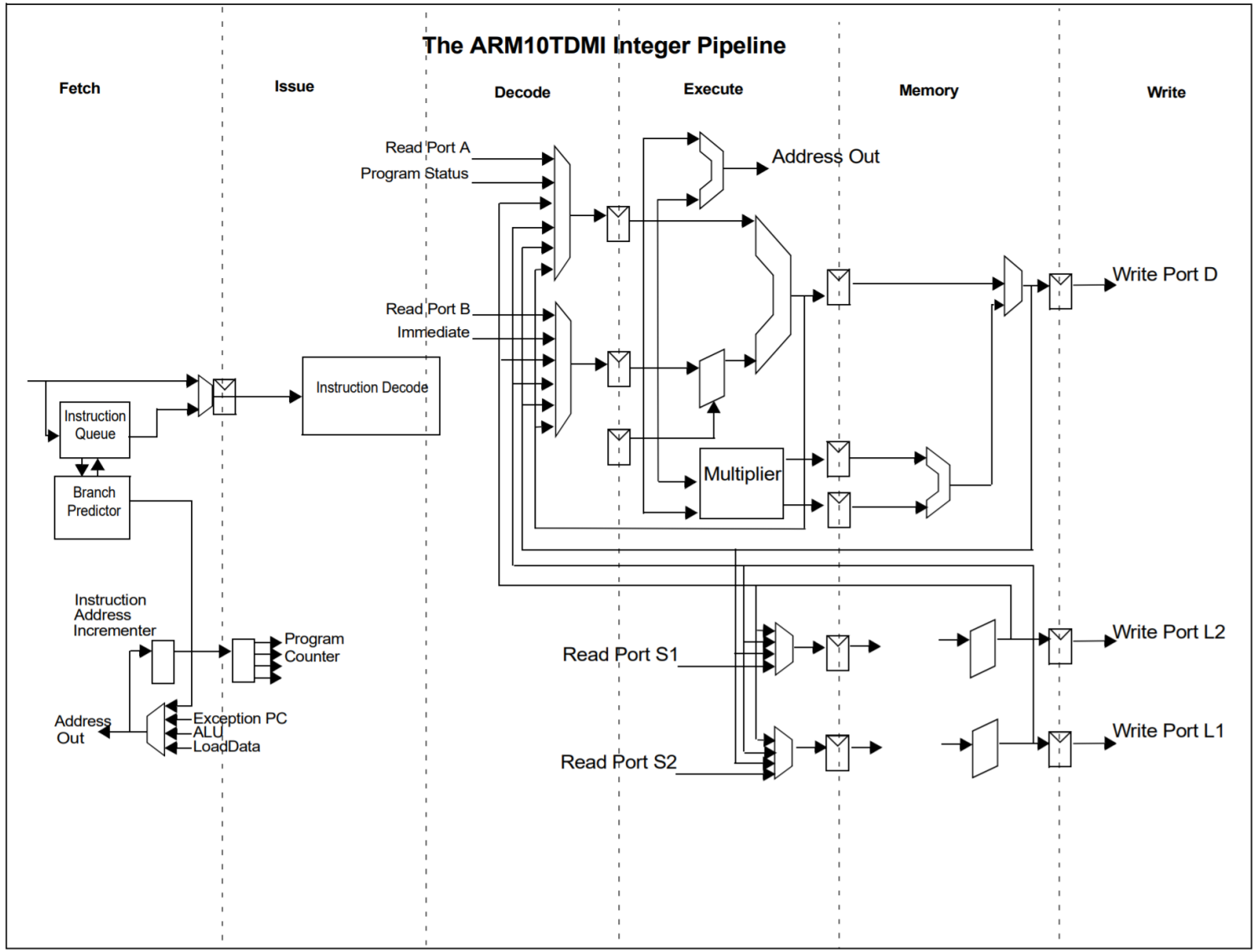
Example: Arm10 Pipeline (1999)



Buffers up to 3 instructions

We can have numerous instructions executing in parallel in these stages

Arm10 Pipeline



Arm10 Pipeline Hazards

- ALU instructions need not necessarily wait for load/store operations in the load/store pipeline to complete, i.e., they may bypass or overtake them if they are independent.
- Data Hazards
  - We cannot allow all ALU instructions to bypass memory instructions. We must respect dependencies.
  - RAW (i.e., the ALU operation requires the result from a load) or a WAW hazard
- Structural Hazards
  - Load/store or multiply pipeline is occupied.
- Check for hazards late.
  - The use of interlocks is minimized by checking for hazards late, not simply in the decode stage.