COMPUTER ARCHITECTURE

Instruction Level Parallelism Exploitation

Contents

- Introduction to pipelining.
- Hazards.
  - Structural hazards.
  - Data hazards
  - Control hazards.
    - Compile-time alternatives.
    - Run-time alternatives
- Multi-cycle operations.

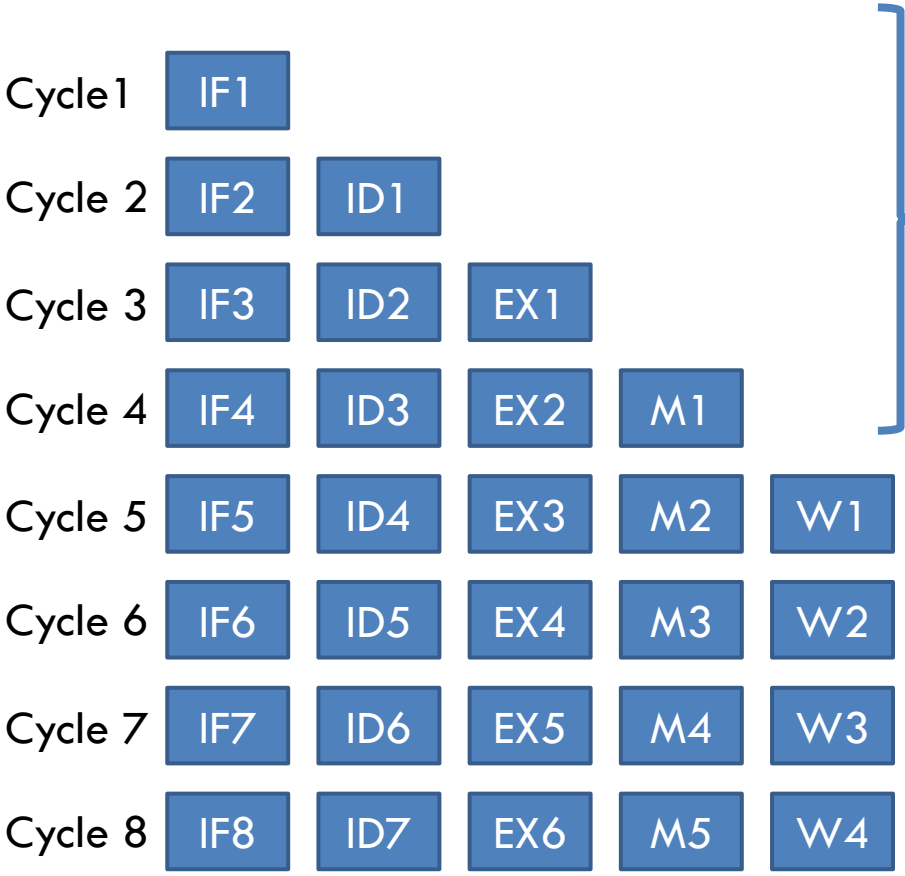
3

Introduction to pipelining

Pipeline

- Implementation technique: Multiple instructions overlap their execution over time.
  - A costly operation is divided into several simpler sub-operations.
  - Sub-operation execution in stages.
- **Effects:**
  - Throughput is increased.
  - Latency does not decrease.

Pipeline



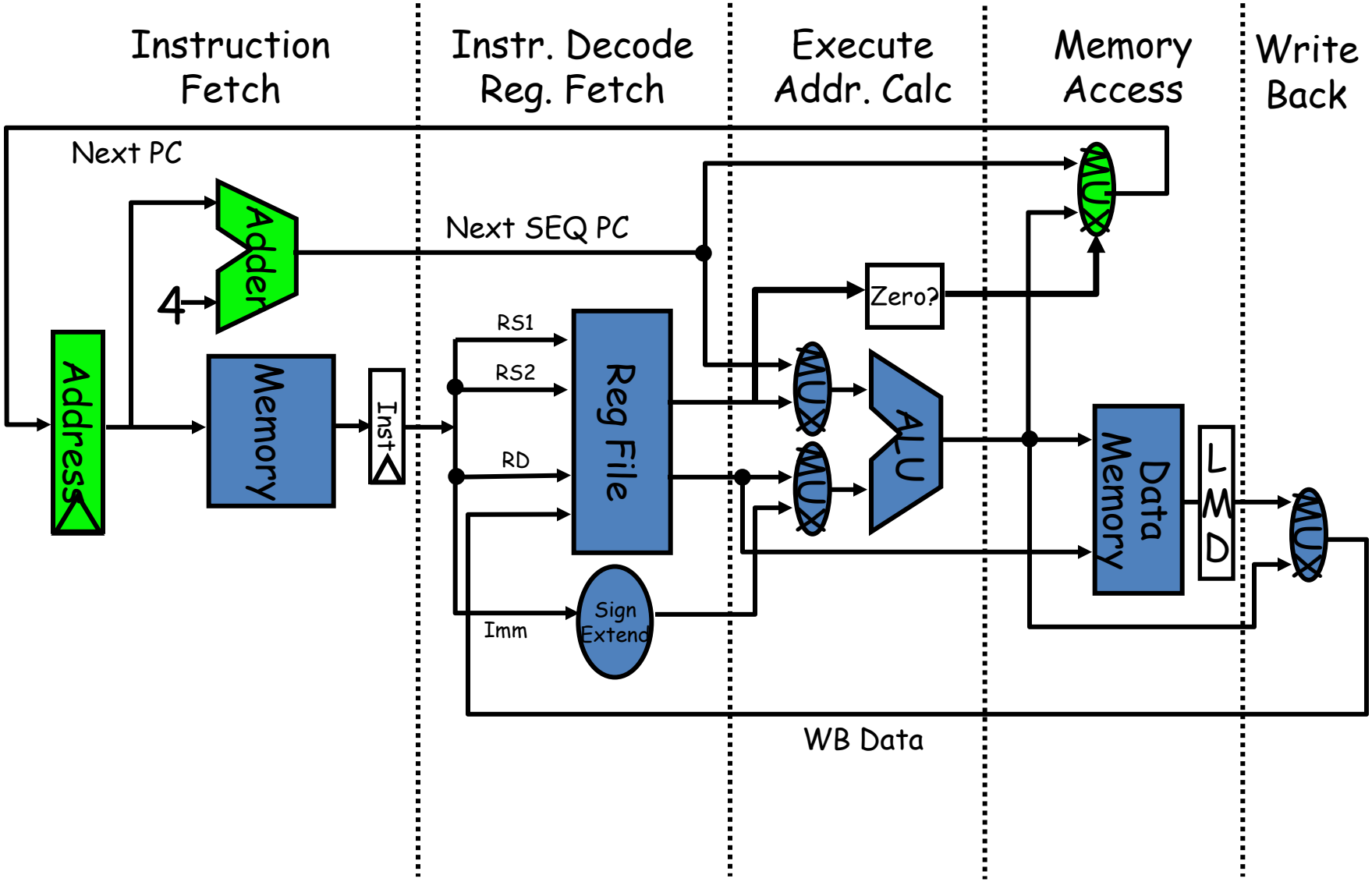
Filling the *pipeline*

Latency→ 5 cycles

Throughput (ideal) → 1 instruction per cycle

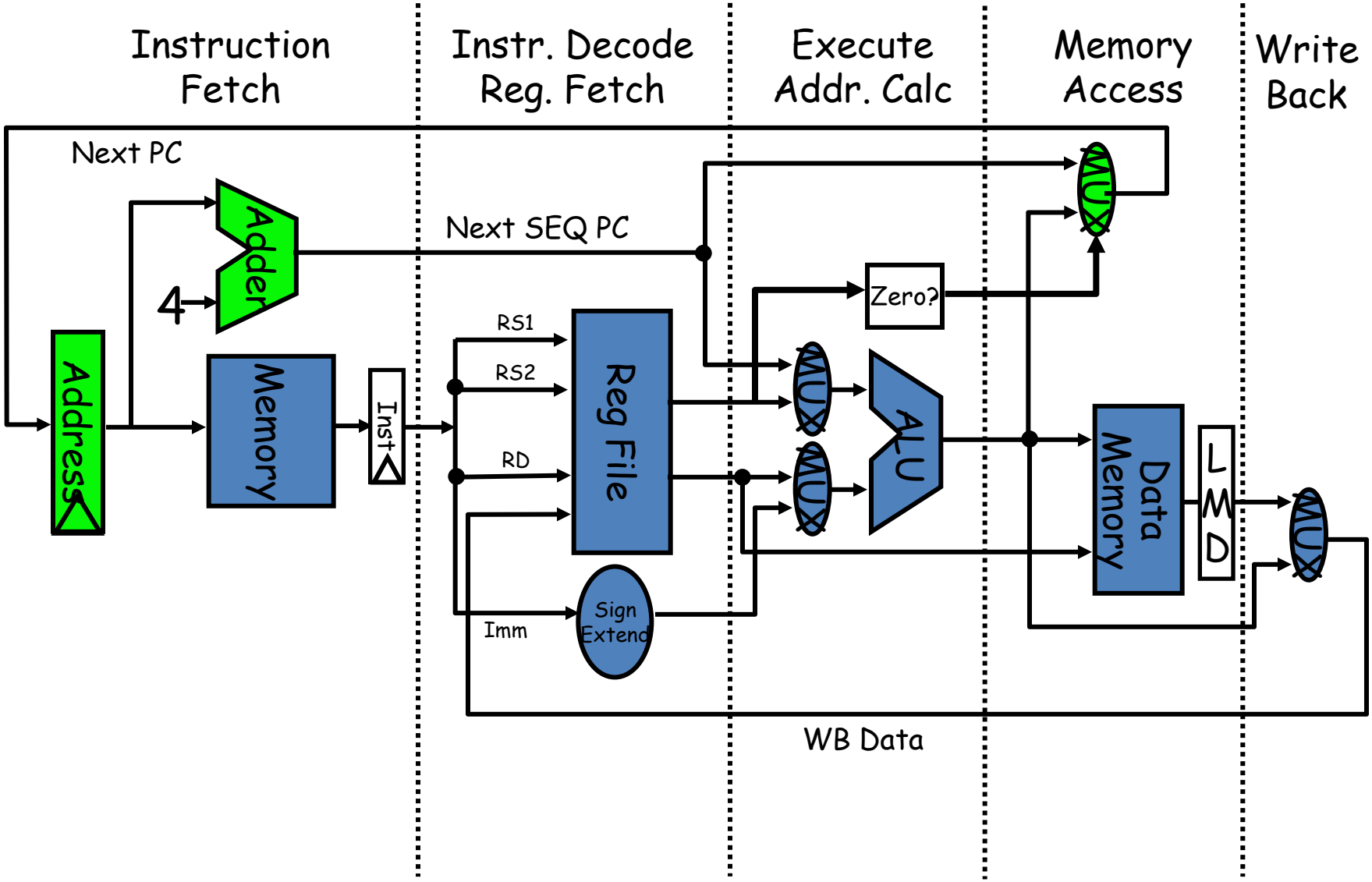
- **Instruction Fetch** (*IF*)
  - Send PC value to memory.
  - Fetch current instruction.
  - Update PC (e.g. add 4).

Pipeline stages



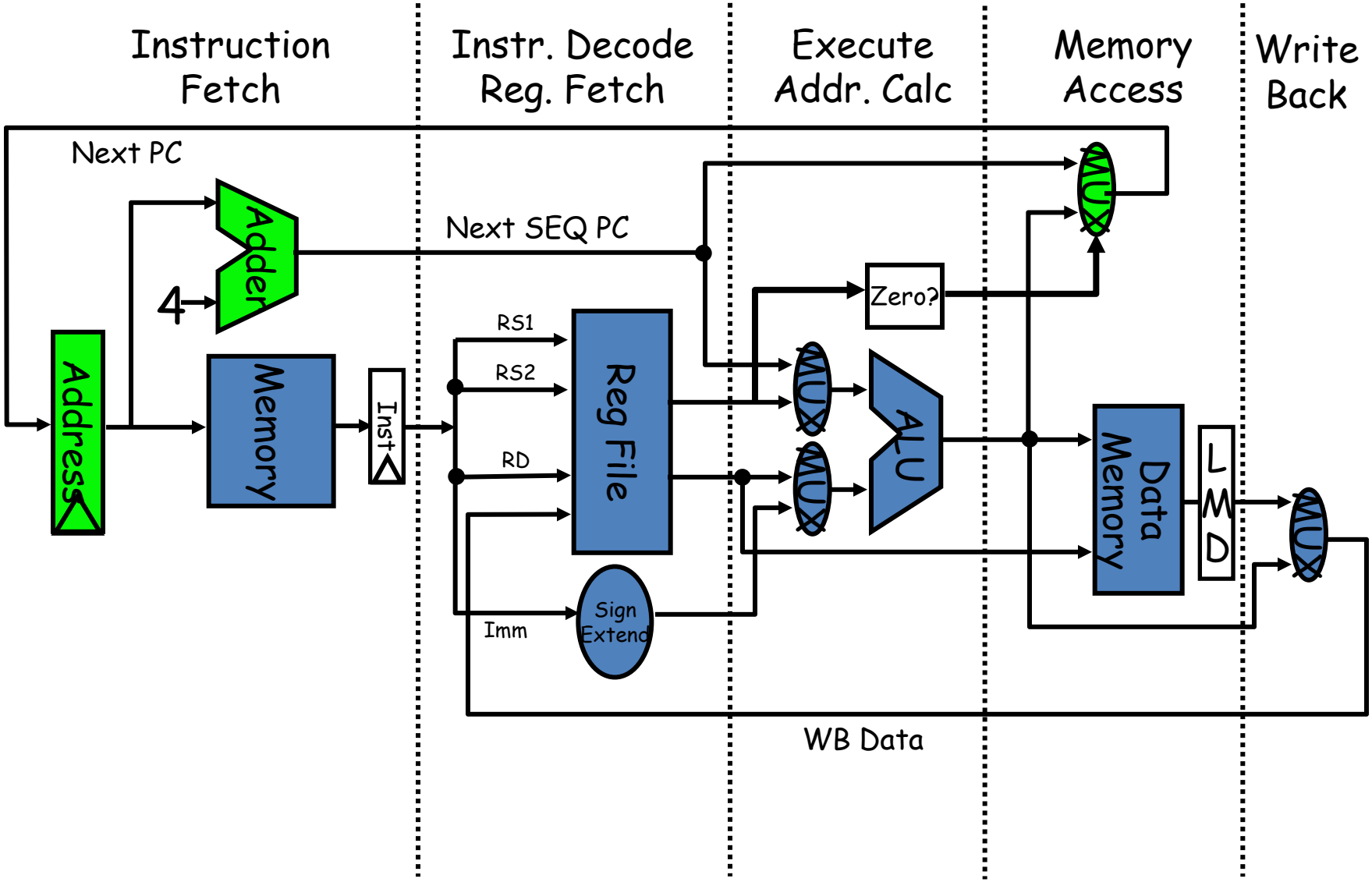
- **Instruction Decode** (*ID*)
  - Decode current instruction.
  - Read referenced source registers values.
  - Perform equality test on register values.
  - Sign-extend offset field of instruction.
  - Compute possible branch target (offset + incremented PC).

Pipeline stages



- **Execution / Effective Address**(*EX*)
  - ALU operates on operands:
    - Memory reference: ALU adds base register and offset to form effective address.
    - Register-Register: ALU performs operation o values from register file.
    - Register-Immediate: ALU performs operation on value from register file and sign extended immediate.

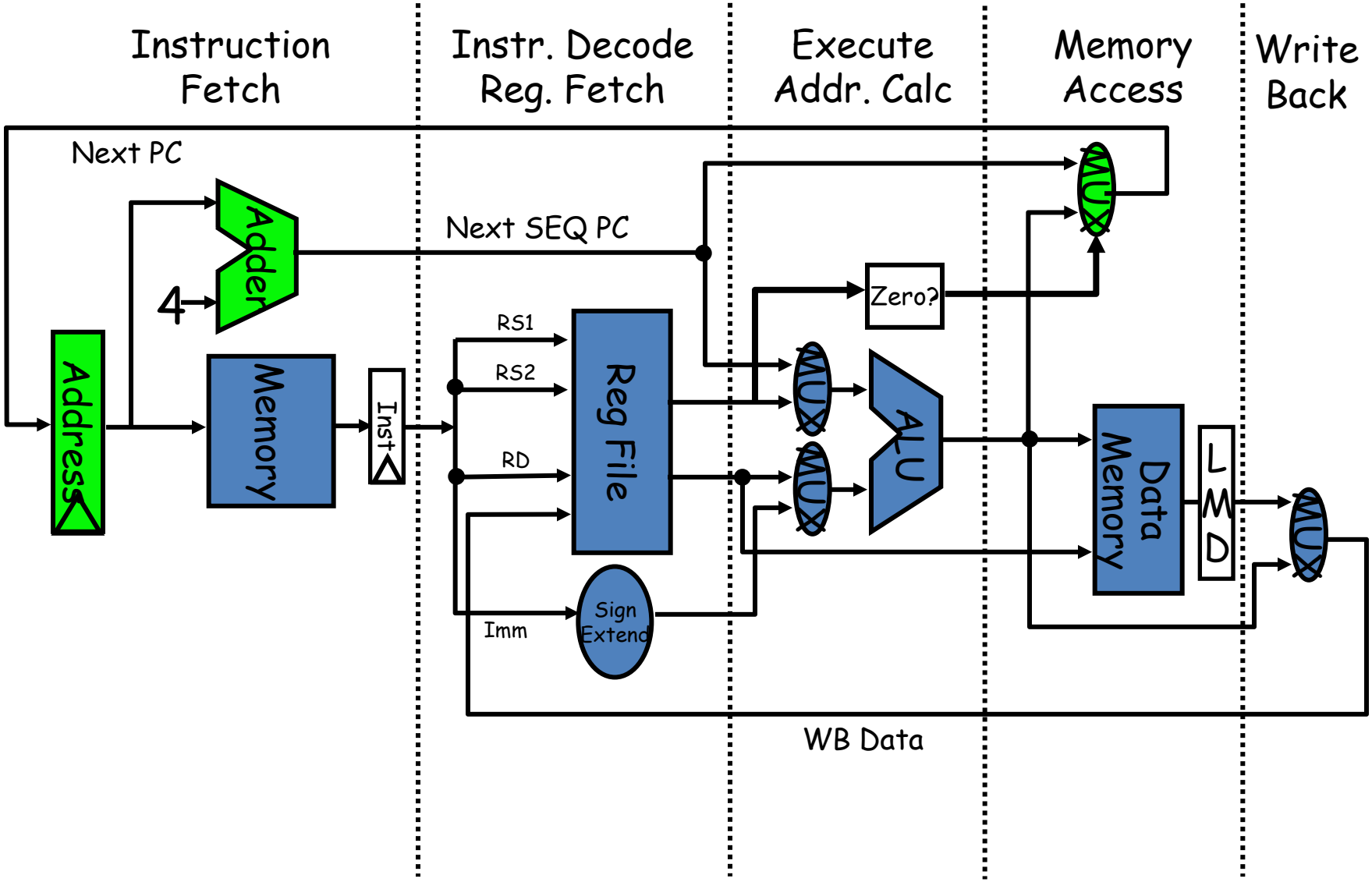
Pipeline stages



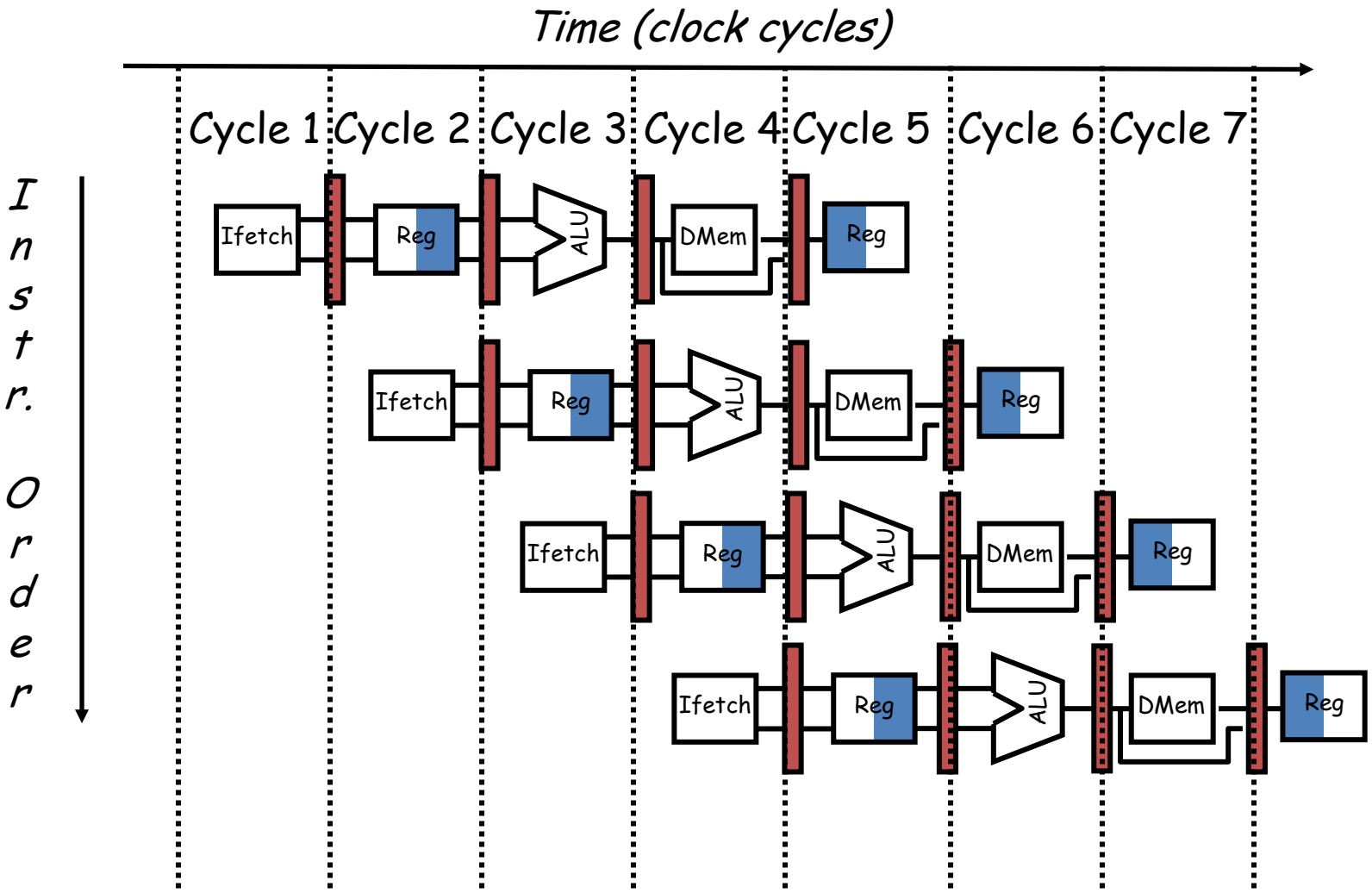
- **Memory Access** (*MEM*)
  - Load instructions:
    - Read using effective address computed in EX.
  - Store instructions:
    - Write data from second register read from register file in ID into effective address computed in EX.

- **Write-back** (WB)
  - Write result (from memory or ALU) into register file.

Pipeline stages



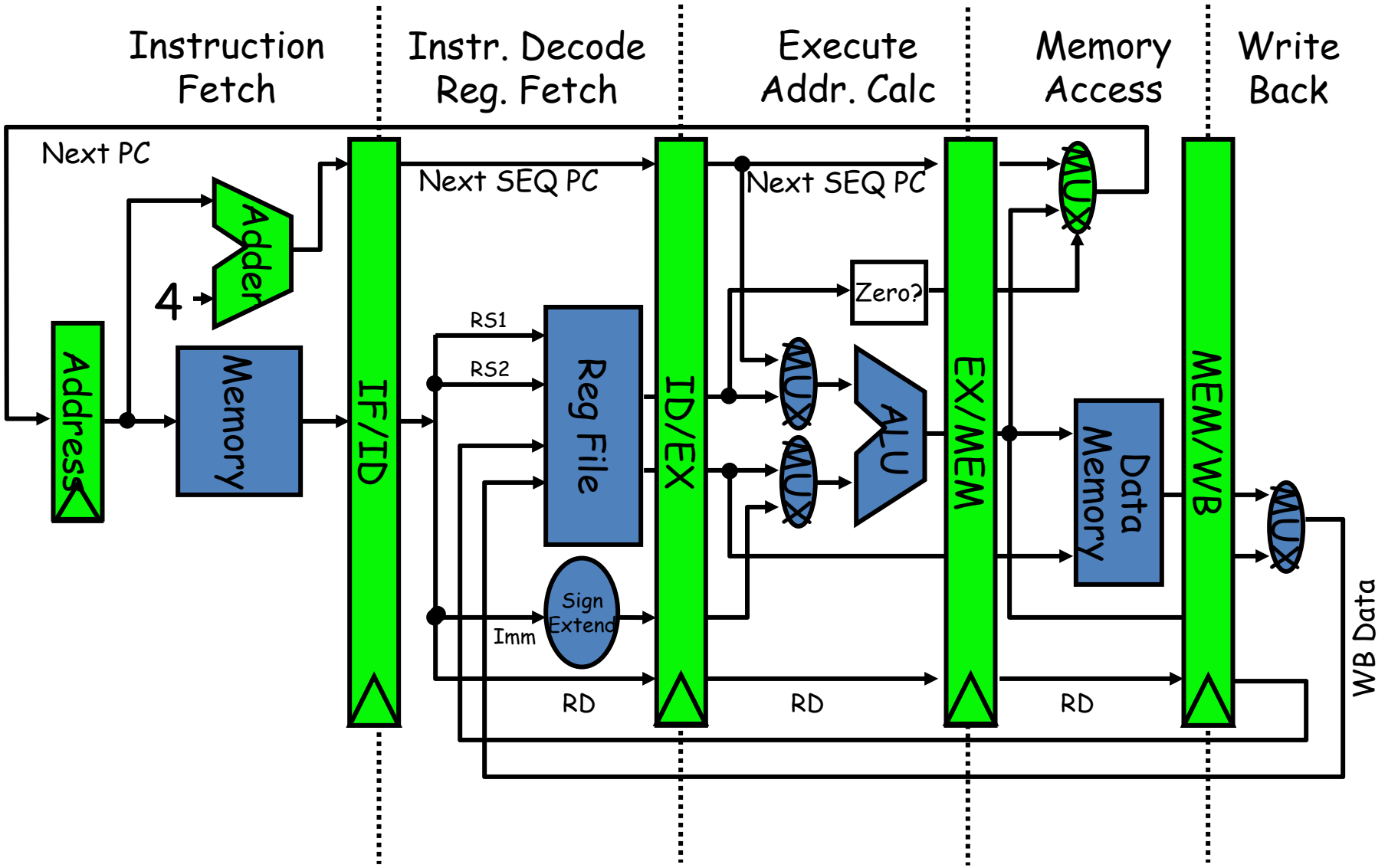
Pipeline over time



Pipeline effects

- A n-depth pipeline, has n times the needed bandwidth compared to the non-pipelined version when clock rate is the same.
  - Solution: Caching, caching, caching, …
- Separation into data and instruction caches suppresses some memory conflicts.
- Instructions in the pipeline should not try to use the same resource at the same time.
  - Solution: Introduce registers at every stage boundary.

Stages communication



Example

- Non-pipelined processor:
  - Clock cycle: 1ns
  - ALU operations (40%) and branching (20%): 4 cycles.
  - Memory operations (40%): 5 cycles.
  - Pipeline overhead: 0.2 ns
- ¿Which is the pipelined version speedup?

$$t_{orig} = \text{clock cycle} \times \text{CPI} = 1ns \times (0.6 \times 4 + 0.4 \times 5) = 4.4ns$$

$$t_{pipeline} = 1ns + 0.2ns$$

$$S = \frac{4.4ns}{1.2ns} = 3.7$$

18

Pipeline hazards

Structural hazards

Data hazards

Control hazards

Hazards

- **Hazard**: Situation preventing next instruction to start at the expected clock cycle.
  - Hazards reduce pipelined architectures performance.

- Hazards:
  - Structural hazard.
  - Data hazard.
  - Control hazard.

- Simple approach for hazards:
  - Stall the instruction flow.
  - Already started instructions will continue.

Structural hazard

- When the processor cannot support all possible instruction sequences.
  - Two pipeline stages need to use the same resource at the same cycle.
- Reasons:
  - Functional units which are not fully pipelined.
  - Functional units which are not duplicated.
- These hazards can be avoided at the cost of a more expensive hardware.

Impact of stalls

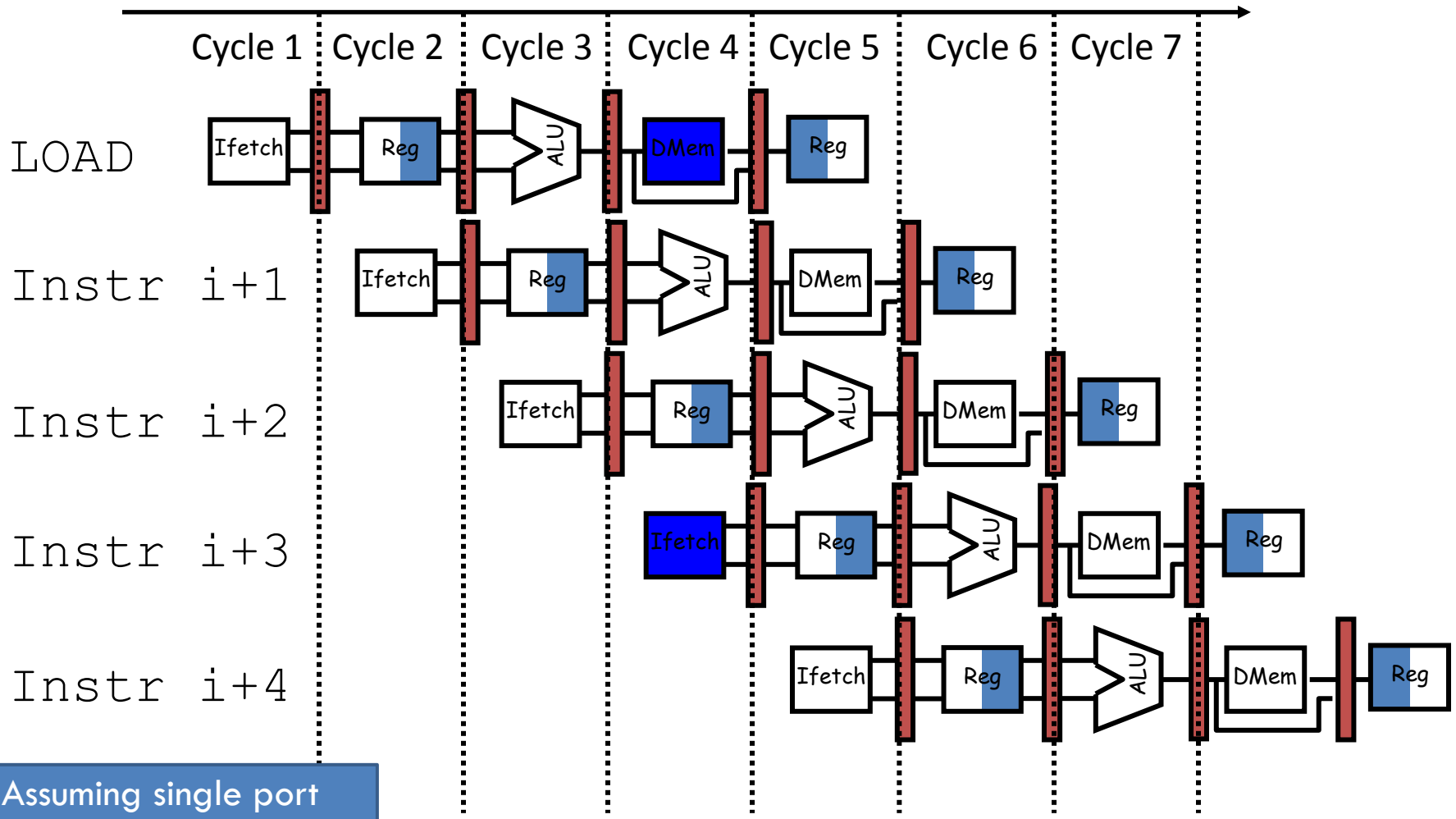
- Pipeline speedup.

$$S = \frac{\text{average instr time non pipelined}}{\text{average instr time pipelined}} = \frac{CPI_{\text{nonpipelined}} \times \text{cycle}_{\text{nonpipelined}}}{CPI_{\text{pipelined}} \times \text{cycle}_{\text{pipelined}}}$$

  - Pipeline ideal CPI is 1.
    - Need to add stall cycles per instruction.
  - Non pipelined processor:
    - CPI=1, but clock cycle much higher.
    - Clock cycle is N times pipelined cycle.
      - N is pipeline depth

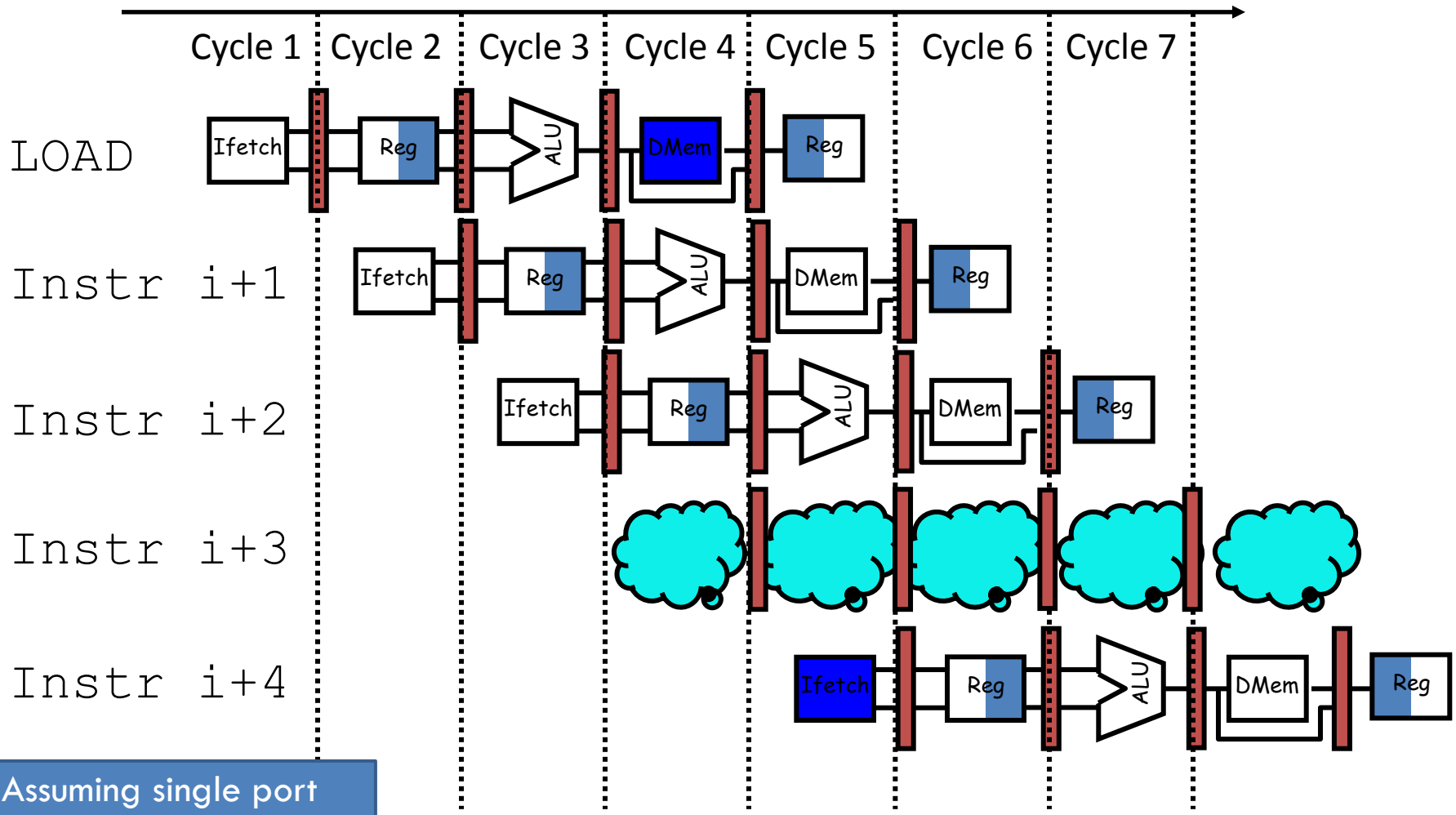
$$S = \frac{\text{pipeline depth}}{1 + \text{stalls per instruction}}$$

Structural hazard: Example



Assuming single port memory

Structural hazard: Example



Assuming single port memory

Example

- Two alternative designs:
  - A: No structural hazard. Clock cycle →1ns.
  - B: With structural hazards. Clock cycle →0.9 ns.
  - Data access instructions with hazards: 30%.

- ¿Which is the fastest alterantive?

$$t_{inst}(A) = CPI \times cycle = 1 \times 1ns = 1ns$$
$$t_{inst}(A) = CPI \times cycle = (0.6 \times 1 + 0.4 \times (1+1)) \times 0.9 = 1.4 \times 0.9 = 1.26ns$$

25

Pipeline hazards

Structural hazards

Data hazards

Control hazards

Riesgos de datos

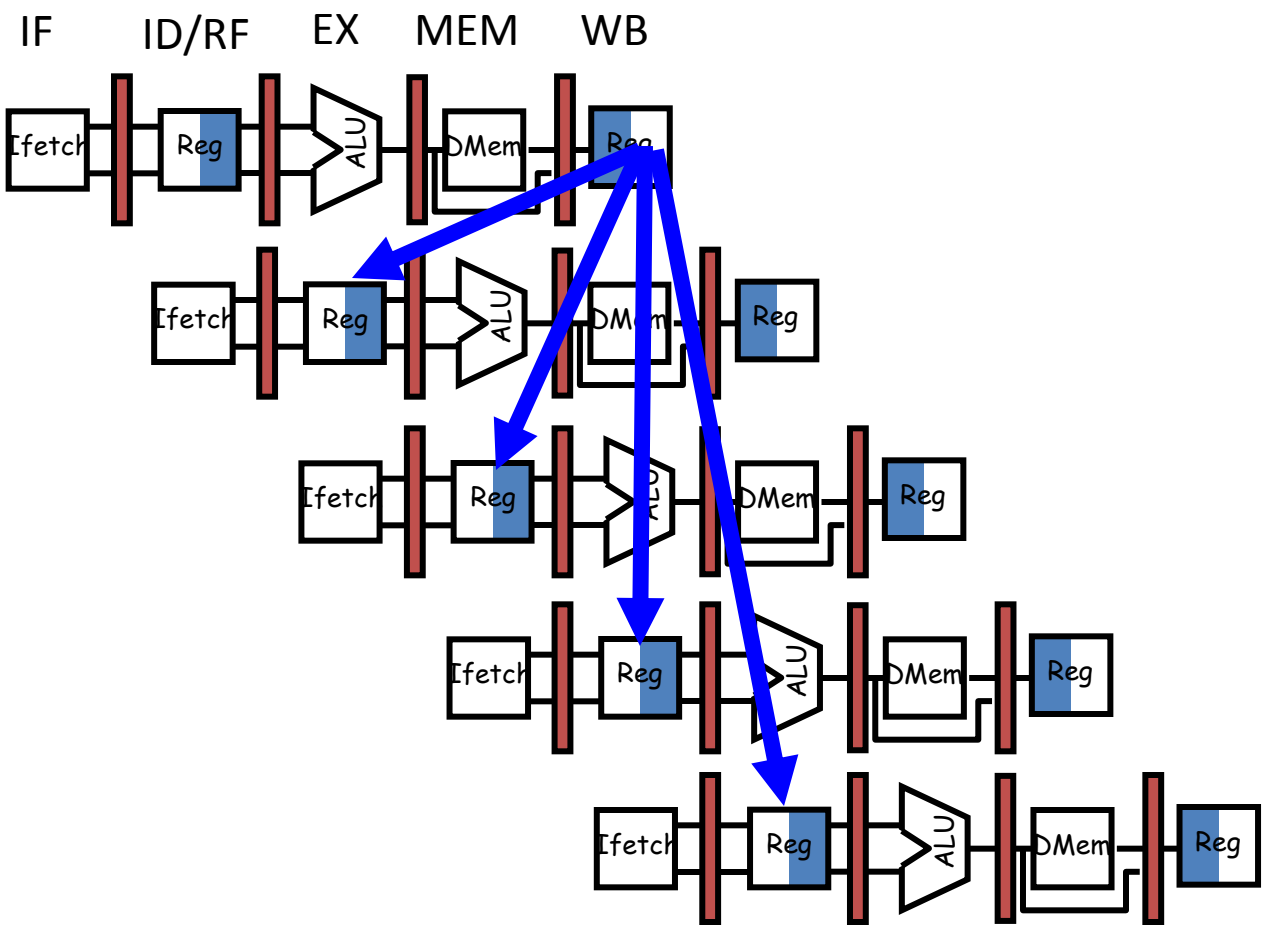
- A **data hazard** happens when the pipeline modifies the read/write access order to operands.

```
I1: DADD R1, R2, R3
I2: DSUB R4, R1, R5
I3: AND R6, R1, R7
I4: OR R8, R1, R9
I5: XOR R10, R1, R11
```

- I2 reads R1 before I1 modifies it.
- I3 reads R1 before I1 modifies it.
- I4 gets the right value
  - Register file read in second half of cycle.
- I5 gets right value.

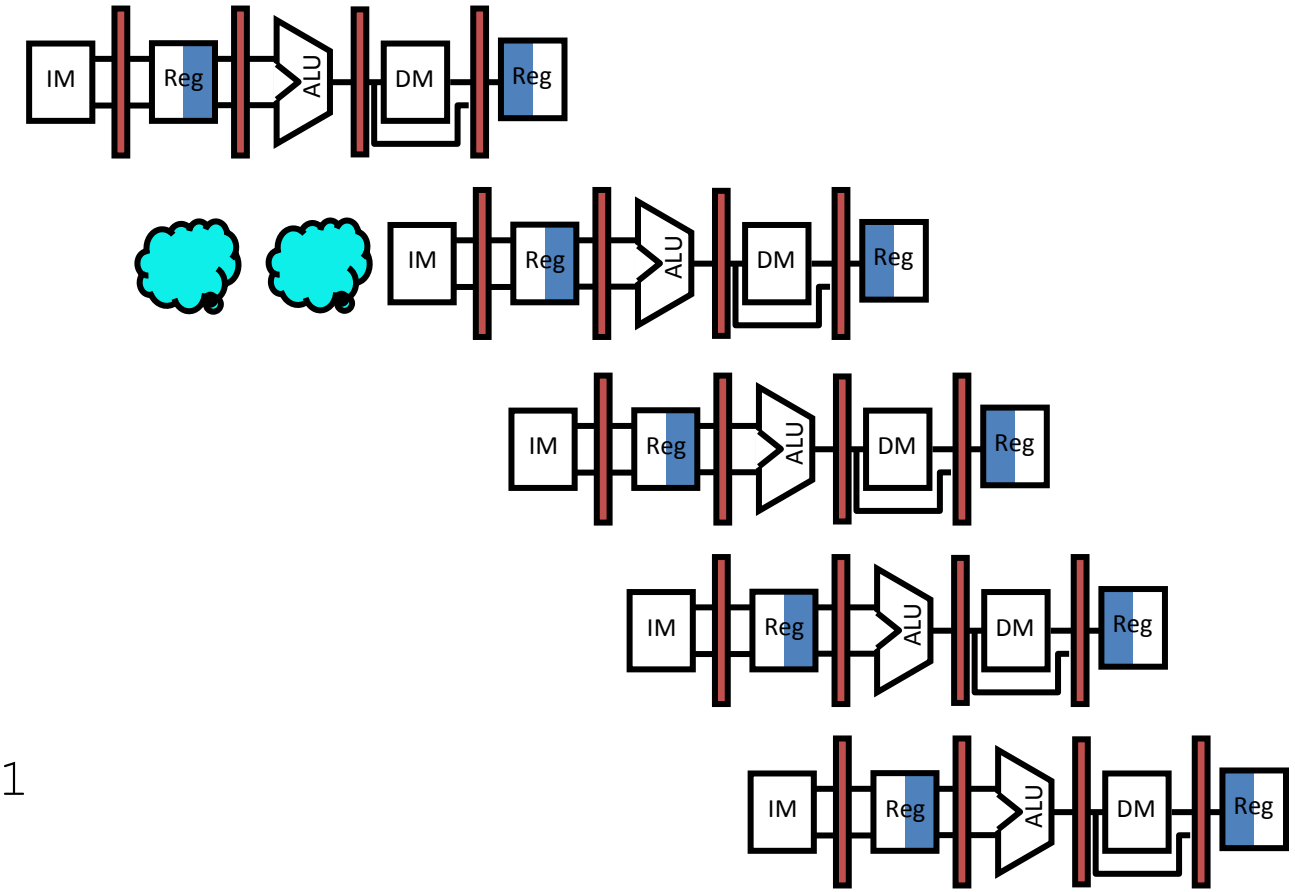
Data hazard


```
DADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8,R1,R9
XOR R10, R1, R11
```



Stalls in data hazards


```
DADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8,R1,R9
XOR R10, R1, R11
```



Data hazard: RAW

- Read After Write.
  - Instruction **i+1** tries to read a datum before instruction **i** writes it.

```
i:      add r1, r2, r3
i+1:    sub r4, r1, r3
```

- If there is a data dependency:
  - Instructions can neither be executed in parallel nor overlap.
- Solutions:
  - Hardware detection.
  - Compiler control.

Data hazard: WAR

- Write After Read:
  - Instruction **i+1** modifies operand before instruction **i** reads it.

```
i:      sub r4, r1, r3
i+1:    add r1, r2, r3
i+2:    mul r6, r1, r7
```

- Also known as anti-dependency (compiler domain).
  - Name reuse.
- Cannot happen in MIPS with 5-stages pipeline.
  - All instructions with 5 stages.
  - Reads are always in stage 2.
  - Writes are always in stage 5.

Data hazards: WAW

- Write After Write:
  - Instruction **i+1** modifies operand before instruction **i** modifies it.

```
i:      sub r1, r4, r3
i+1:    add r1, r2, r3
i+2:    mul r6, r1, r7
```

- Also known as output dependency (compiler domain).
  - Name reuse.
- Cannot happen in MIPS with 5-stages pipeline.
  - All instructions with 5 stages.
  - Writes are always in stage 5.

Solutions to data hazards

- RAW dependencies:
  - Forwarding.
- WAR y WAW dependencies:
  - Register renaming:
    - Statically by compiler.
    - Dynamically by hardware.

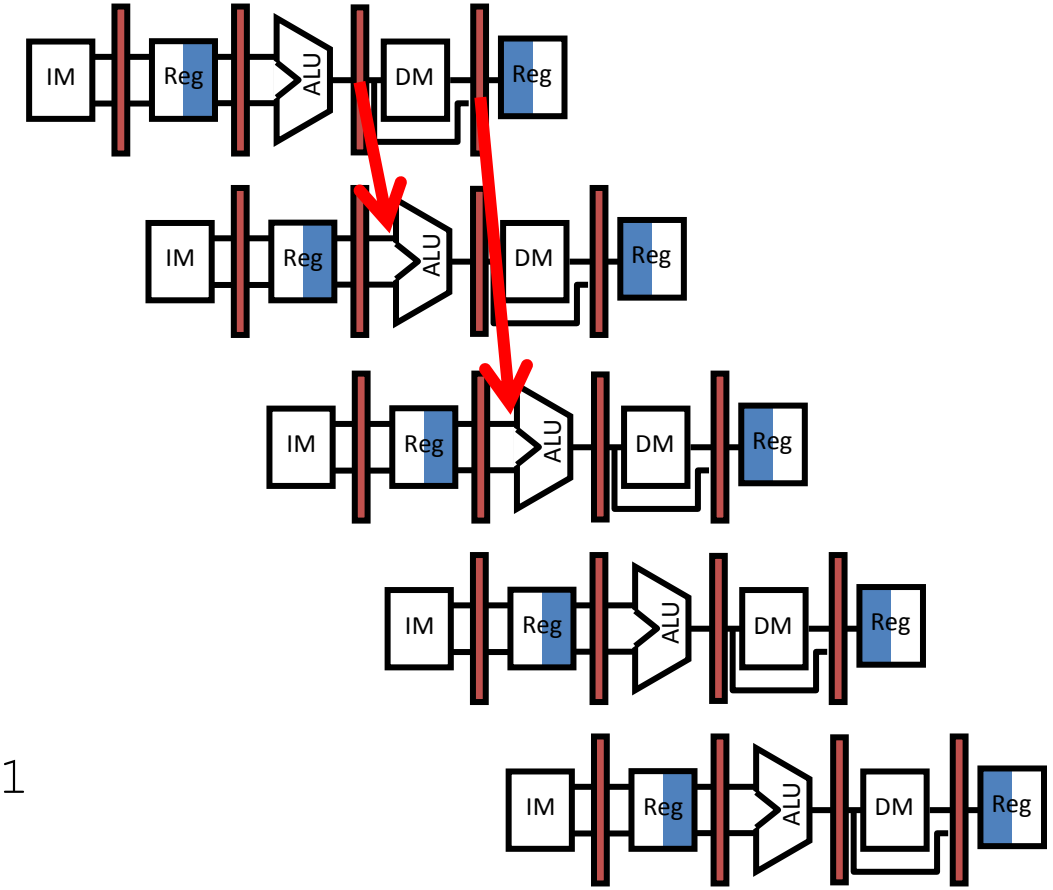
Forwarding

- Technique to avoid some data stalls.
- Basic idea:
  - No need to wait until result is written into register file.
  - Result is already in pipeline registers.
  - Use this value instead of the one from the register file.

- Implementation:
  - Results from EX and MEM stages written into ALU input registers.
  - Forwarding logic selects between real input and forwarding register.

Forwarding

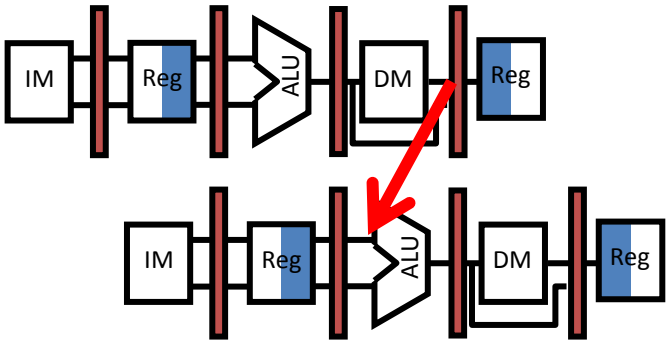

```
DADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8,R1,R9
XOR R10, R1, R11
```



Forwarding limitations

- Not every hazard can be avoided with forwarding.
  - You cannot travel backwards in time!

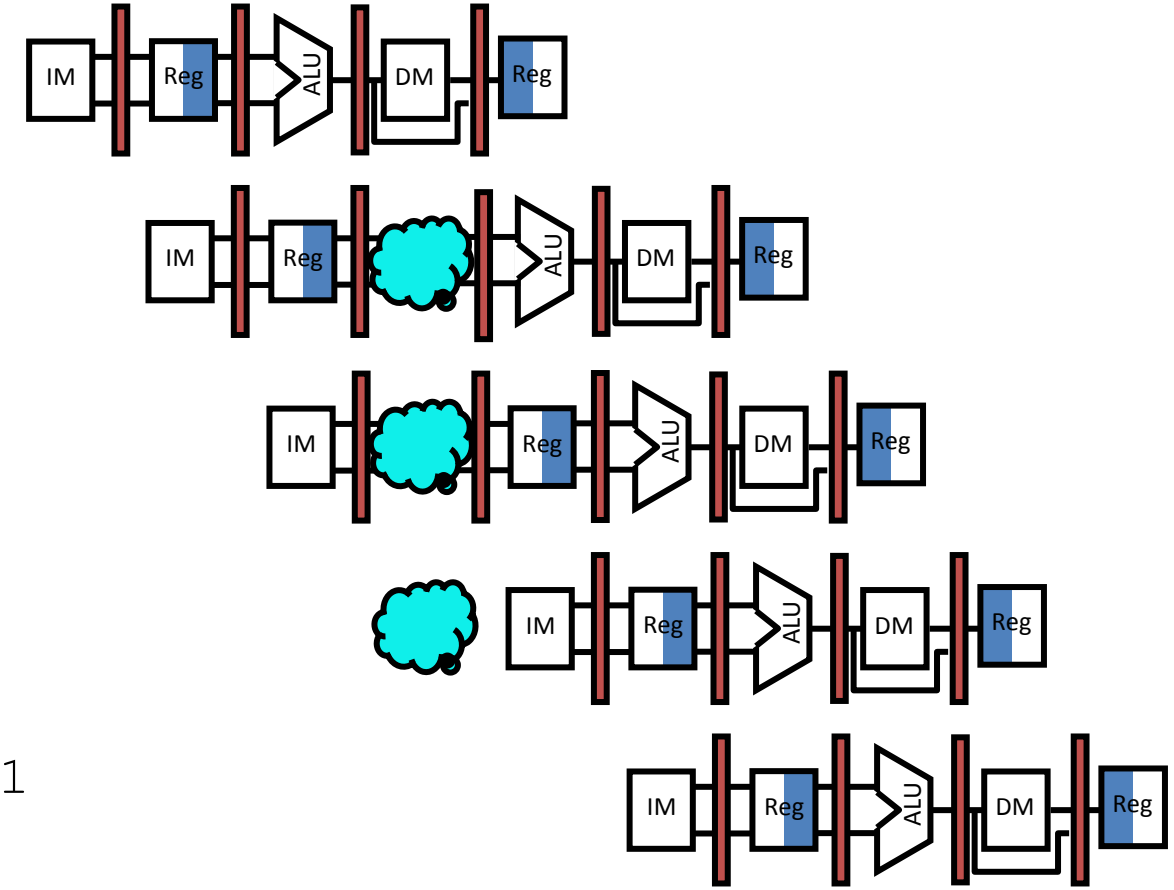
```
I1: LD R1, (0)R2
I2: DSUB R4, R1, R5
I3: AND R6, R1, R7
I4: OR R8, R1, R9
I5: XOR R10, R1, R11
```



In this case a stall is needed

Memory access stalls

```
LD R1, 0(R2)
DSUB R4, R1, R5
AND R6, R1, R7
OR R8,R1,R9
XOR R10, R1, R11
```



Pipeline hazards

Structural hazards

Data hazards

Control hazards

- **Compile time alternatives**
- Run-time alternatives

Control hazard

- A control hazard is associated to a PC modification instruction.
  - Next instruction is not known until current one completes.

- Terminology:
  - Taken branch: PC is updated.
  - Not taken branch: PC is not updated.

- Problem:
  - Pipeline assumes branch will not be taken.
  - What if, after ID, we find out branch needs to be taken?

Alternatives in control hazards

- **Compile-time**: Fixed for all program execution.
  - Software may try to minimize impact if it knows hardware behavior.
  - Compiler can do this job.

- **Run-time**: Variable behavior during program execution.
  - Tries to predict what software will do.

Control hazards: Static solutions

- Alternatives:
  - Pipeline freezing.
  - Fixed prediction.
    - Always not taken.
    - Always taken.
  - Delayed branching.

- In many cases the compiler needs to know what will be done to reduce negative impacts.

Pipeline Freezing

- **Idea**: If current instruction is a branch stop or flush subsequent instructions from the pipeline until target is known.
  - Run-time penalty is known.
  - Software (compiler) cannot do anything.

- Branch target is known in ID stage
  - Repeat next instruction FETCH.

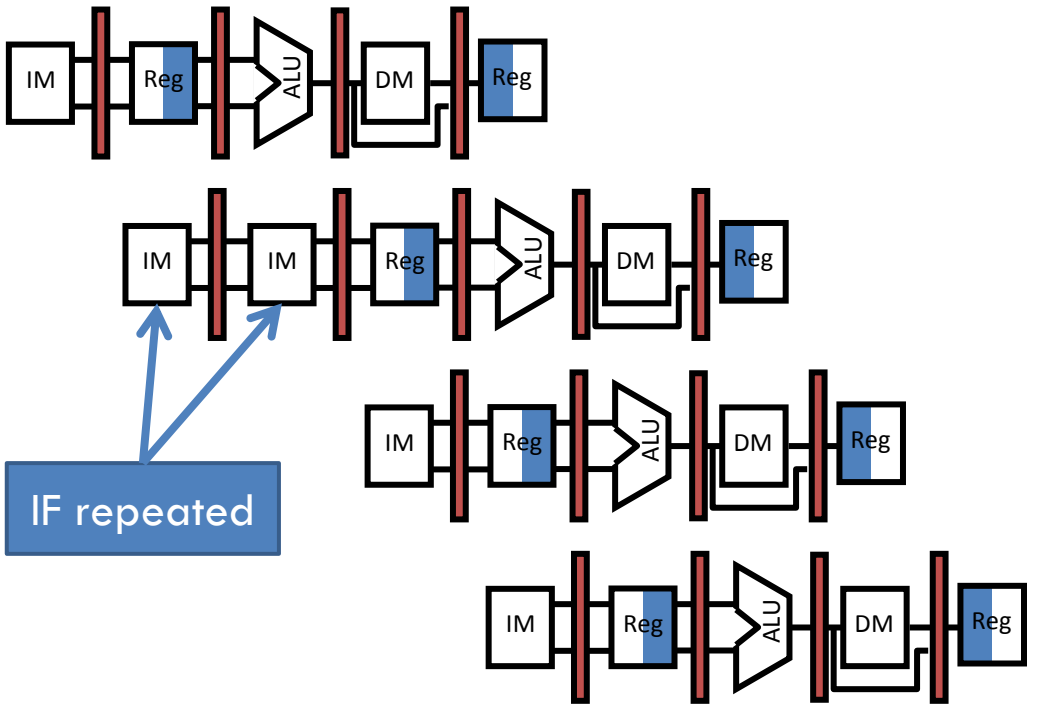
FETCH repetition

Branch Instr.

Instr. i+1

Instr. i+2

Instr. i+3



IF repetition equivalent to a stall

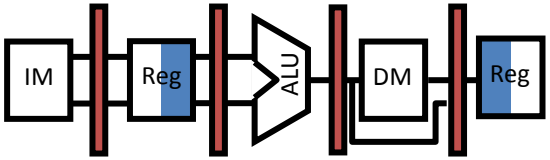
A stall per branch may lead to a performance loss from 10% to 30%

Fixed prediction: not-taken

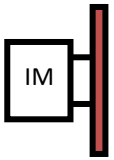
- **Idea**: Assume branch will be **not-taken**.
  - Avoids updating processor state until branch not taken is confirmed.
  - If branch is taken, subsequent instructions are retired from pipeline and next instruction is fetched from branch target.
    - Transforms instructions in NOPs.
- Compiler task:
  - Organize code setting most frequent option as not-taken and inverting condition if needed.

Fixed prediction: not-taken

Branch Instr.

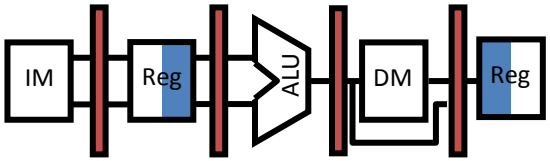


Instr. i+1

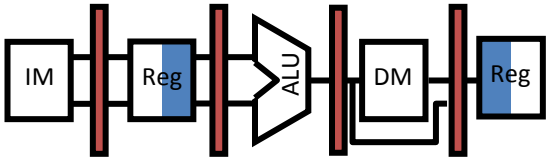


Inactive

Branch target



Instr. i+1

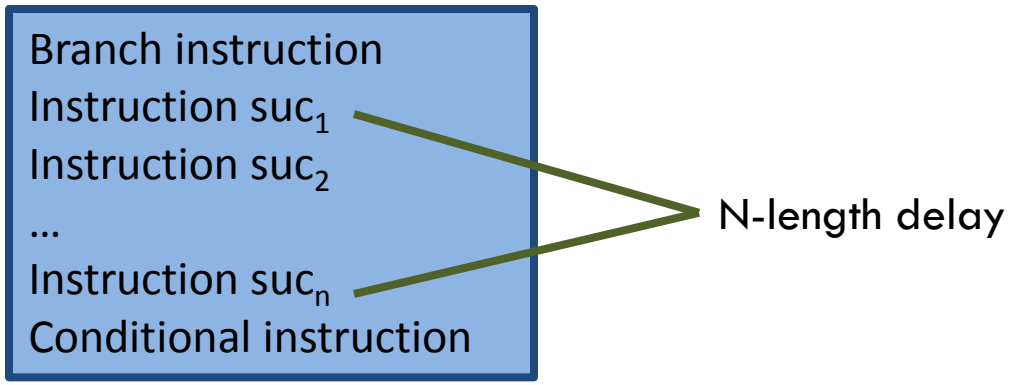


Fixed prediction: taken

- **Idea**: Assume branch will be **taken**.
  - As soon as branch is decoded and target is computed, target instructions start to be fetched.
  - In a 5-stages pipeline does not provide improvements.
    - Target address not known until branch decision is made.
    - Useful in processors with complex and slow conditions.
- Compiler task:
  - Organize code setting most frequent option as taken and inverting condition if needed.

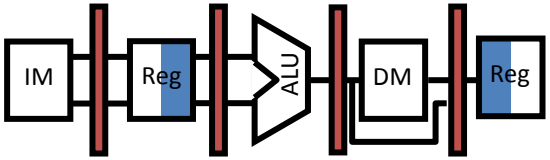
Delayed branching

- **Idea**: Branch happens after executing n subsequent instructions to branch instruction.
  - In 5-stages pipeline: 1 delay slot.

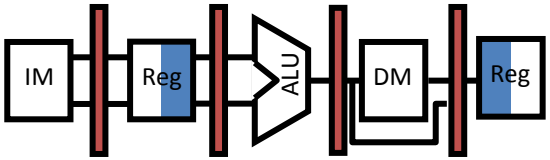


Delayed branching

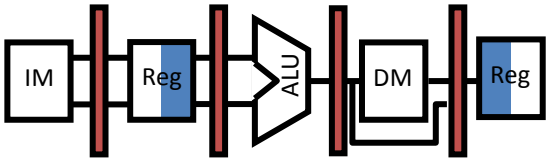
Branch Instruction



Delayed instruction



Next or target instruction



Delayed branching: Compiler


```
        DADD R1, R2, R3
        BEQZ R2, LABEL
        Delay slot
        XOR R5, R6, R7
LABEL:  AND R8, R9, R10
```

```
        ~~DADD R1, R2, R3~~
        BEQZ R2, LABEL
        DADD R1, R2, R3
        XOR R5, R6, R7
LABEL:  AND R8, R9, R10
```

Preferred

```
LABEL:  DADD R1, R2, R3
        XOR R5, R6, R7
        BEQZ R5, LABEL
        Delay slot
        AND R8, R9, R10
```

```
~~LABEL:~~ DADD R1, R2, R3
LABEL:  XOR R5, R6, R7
        BEQZ R5, LABEL
        DADD R1, R2, R3
        AND R8, R9, R10
```

XOR cannot move to delay slot due to data dependency

```
        DADD R1, R2, R3
        BEQZ R1, LABEL
        Delay slot
        XOR R5, R6, R7
LABEL:  AND R8, R9, R10
```

```
        DADD R1, R2, R3
        BEQZ R1, LABEL
        XOR R5, R6, R7
        ~~XOR R5, R6, R7~~
LABEL:  AND R8, R9, R10
```

Only if R5 is not used after LABEL

Delayed branching

- Compiler effectiveness for the 1-slot case:
  - Fills around 60% slots.
  - Around 80% instructions executed in slots are useful for computation.
  - Around 50% slots filled usefully.

- With deeper pipelines and multiple issue more slots are needed.
  - Need to move to more popular dynamic approaches.

Pipeline performance with branch fixed prediction

- Branch stalls number depends on:
  - Branch frequency.
  - Branch penalty.

$$\text{stall cycles from branches} = \text{branch frequency} \times \text{branch penalty}$$

$$S = \frac{\text{pipeline depth}}{1 + \text{branch frequency} \times \text{branch penalty}}$$

Example

- MIPS R4000 (deeper pipeline).
  - 3 stages before knowing branch target.
  - 1 additional stage to evaluate condition.
  - Assuming no data stalls in comparisons.
  - Branch frequency:
    - Unconditional branch: 4%.
    - Conditional branch, not-taken: 6%
    - Conditional branch, taken: 10%

Branch scheme	Penalty Unconditional	Penalty Not-taken	Penalty Taken
Flush pipeline	2	3	3
Predict taken	2	3	2
Predict not-taken	2	0	3

Solution

Branch scheme	Unconditional Branch	Branch not-taken	Branch taken	Total
Frequency	4%	6%	10%	20%
Flush pipeline	0.04 x 2 = 0.08	0.06 x 3=0.18	0.10 x 3=0.30	0.56
Predict taken	0.04 x 2 = 0.08	0.06 x 3 = 0.18	0.10 x 2 = 0.20	0.46
Predict not-taken	0.04 x 2 = 0.08	0.06 x 0 = 0.00	0.10 x 3 = 0.30	0.38

Contribution over ideal CPI

***Speedup* of predicting taken over flushing pipeline.**

$$S = \frac{1+0.56}{1+0.46} = 1.068$$

***Speedup* of predicting not-taken over flushing pipeline**

$$S = \frac{1+0.56}{1+0.38} = 1.130$$

Pipeline hazards

Structural hazards

Data hazards

Control hazards

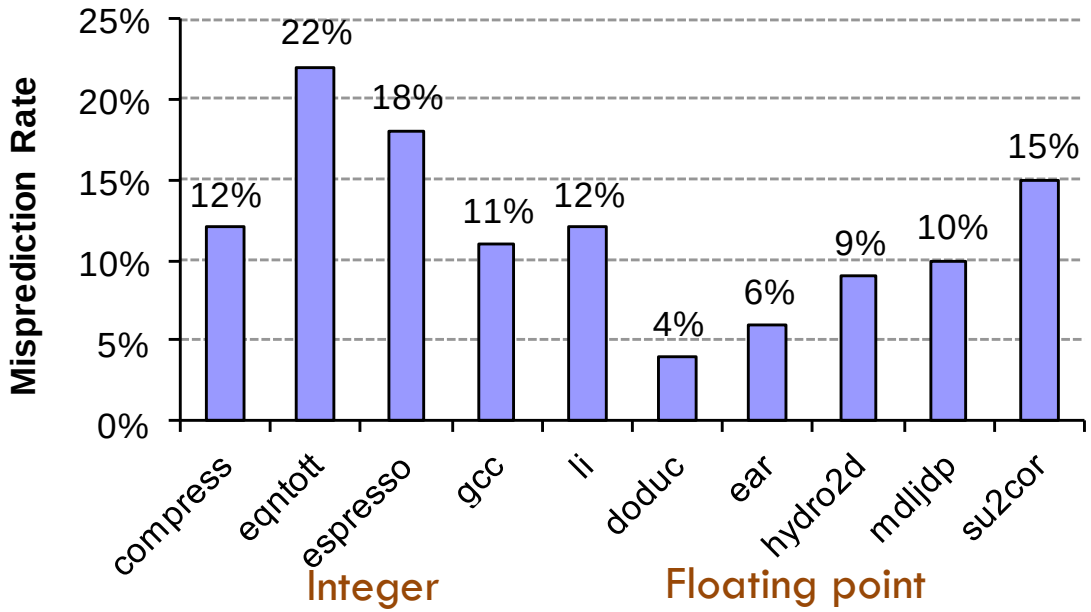
- Compile-time alternatives
- **Run-time alternatives**

Branching and run-time

- Each branch is strongly biased:
  - Either it is taken most of the time,
  - Or it is not taken most of the time.

- **Prediction based on execution profile:**
  - Run once to collect statistics.
  - Collected information used to modify code and take advantage of information.

Prediction with execution profile

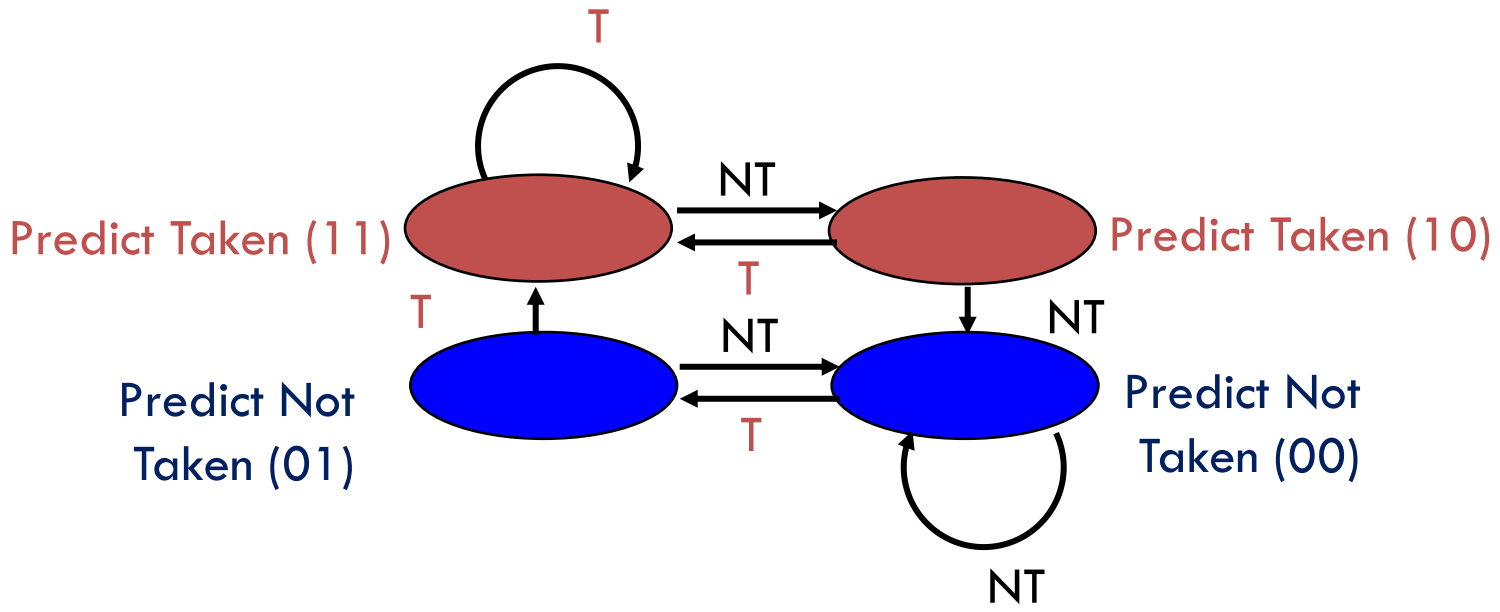


- SPEC92: Branch frequency → 3% a 24%
- **Floating point**.
  - Misprediction rate. Average: 9%. Standard deviation: 4%
- **Integer**.
  - Misprediction rate. Average: 15%. Standard deviation: 5%

Dynamic prediction: BHT

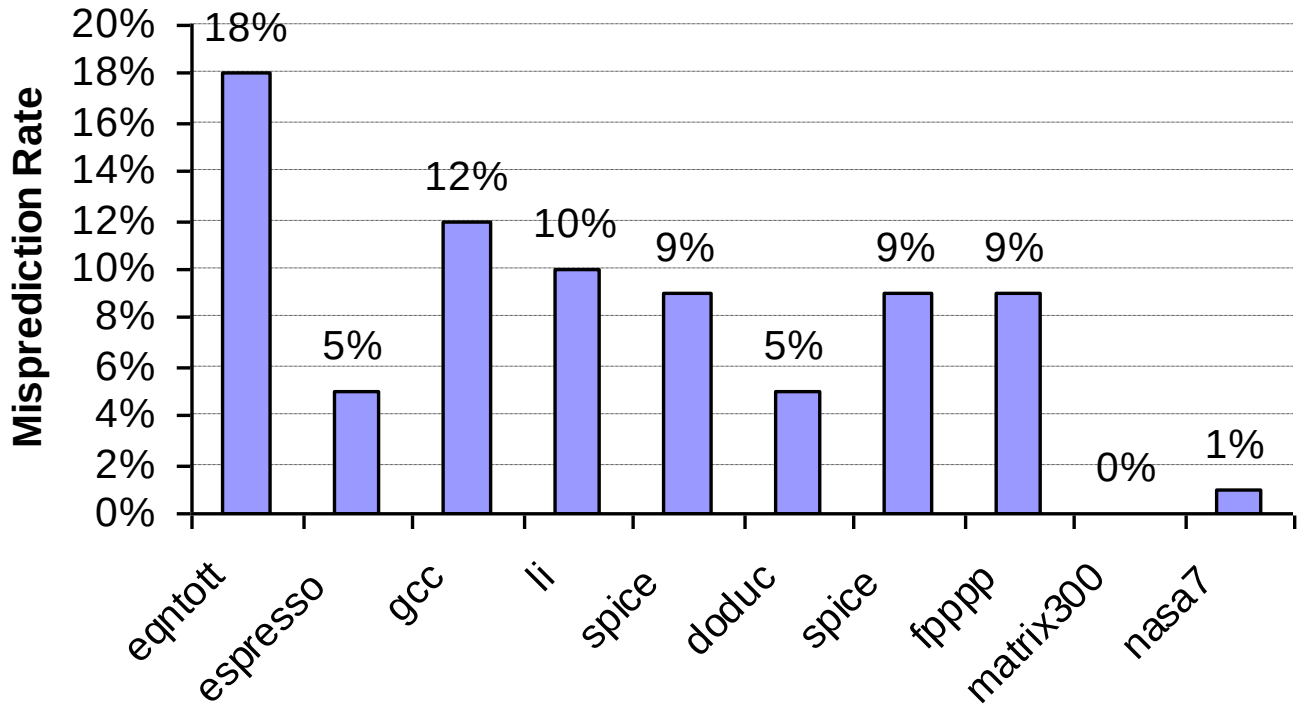
- **B**ranch **H**istory **T**able:
  - **Index**: Lower portion of branch instruction address (PC).
  - **Value**: 1 bit (branch taken or not last time).

- **Improvement**: Use more bits to increase precision.



BHT: Precision

- Mispredictions:
  - Wrongly predict branch outcome.
  - History of different branch in table entry.
- BHT results of 2 bits en 4K entries:



Branch dynamic prediction

- Why does branch prediction work?
  - Algorithms exhibits regularities.
  - Data structures exhibit regularities.

- Is dynamic prediction better than static prediction?
  - It looks like.
  - There is a small amount of important branches in programs with dynamic behavior.

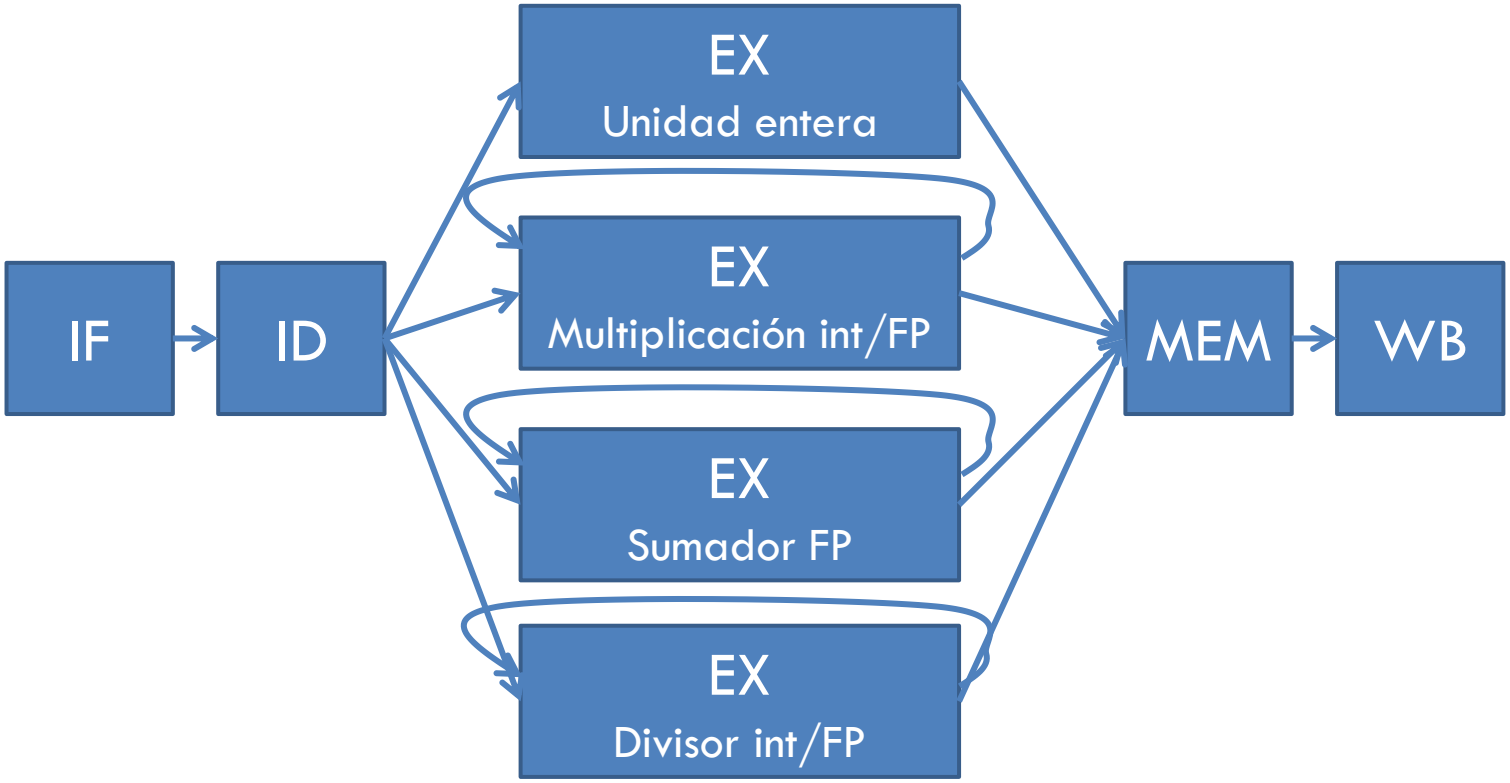
Multi-cycle operations

Floating point operations

- Floating point operations in a single cycle?
  - A extremely long clock cycle.
    - Impact on global performance.
  - FPU control logic very complex.
    - Enormous amount of logic in FP units.

- **Alternative**: Pipelining floating point unit.
  - Execution stage is repeated several times.
  - Multiple functional units in EX.
    - Example: Integer unit, FP and integer multiplier, FP adder, FP and integer divider.

Pipelining and floating point



- EX stage may have a duration more than 1 clock cycle.

Latencies and initiation interval

- **Latency**: Number of cycles between an instruction producing a result and an instruction using that result.

- **Initiation interval**: Number of cycles between two instructions using the same functional units.

Operation	Latency	Initiation Interval
ALU entera	0	1
Loads	1	1
FP addition	3	1
FP multiplication	6	1
FP division	24	25

Summary

- Pipelined architectures require higher memory bandwidth.
- Pipeline hazards lead to stalls.
  - Performance degradation.
- Stalls due to data hazards may be mitigated with compiler techniques.
- Stalls due to control hazards may be reduced with:
  - Compile-time alternatives.
  - Run-time alternatives.
- Multi-cycle operations allow for shorter clock cycles.

References

- **Computer Architecture. A Quantitative Approach.**
  Fifth Edition.
  Hennessy y Patterson.
  Sections C.1, C2 y C5.

- Recommended exercises:
  - C.1, C.2, C.3, C.4, C.5