Functioning and Performance of Processors

Computer Architecture

Riad Bourbia

Computer Sciences department

Guelma University

Presentation Outline

Processor Characteristics

- Instruction Execution Cycles
- Memory Access Models

Processor Performance

- Clock Frequency
- Cache Memory
- Pipeline

Characteristics of a Processor

❖ Instruction Set

Complex: CISC (Complex Instruction Set Computing)

Features: Numerous complex instructions, often requiring multiple clock cycles to execute.

Examples: Intel and AMD processors (x86 family).

Reduced: RISC (Reduced Instruction Set Computing)

Features: Fewer instructions, designed to execute in a few clock cycles.

Examples: Oracle Sparc and IBM PowerPC.

❖ Architecture Complexity

The number of transistors: The higher the transistor count, the more instructions the processor can execute per second.

❖ **Memory Elements** (Register Bank, Cache Memory)

Characteristics of a Processor

❖ Number of Bits Processed per Instruction (32 or 64 bits)

- Defines the size of the processor registers.
- In 64-bit architecture, integers and addresses increase from 32 bits (4 bytes) to 64 bits (8 bytes).

❖ Maximum Clock Speed

- The higher the clock speed, the more instructions the processor can execute per second.

❖ 1 CPU = Multiple Functional Units

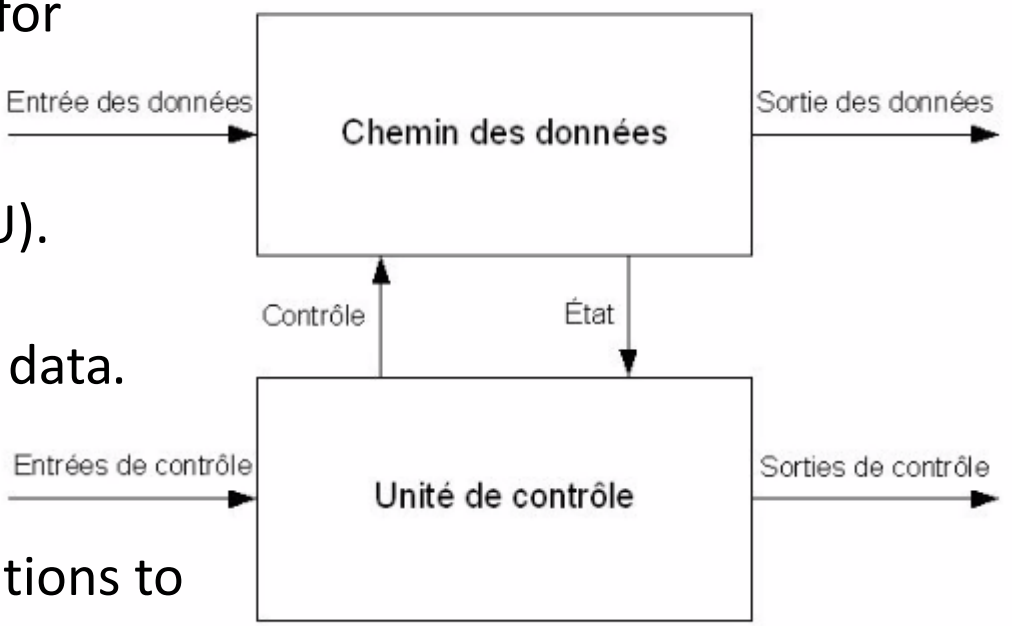
- **Bus Management Unit (I/O Units)**
- **Instruction Unit (Control Unit)**
- **Execution Unit**
  - Contains one or more **Arithmetic Logic Units (ALUs)** for integer and logical operations.
  - May also include one or more **Floating Point Units (FPUs)** for handling floating-point arithmetic operations

Characteristics of a Processor

A Processor is Composed of Two Parts:

1.The Datapath

1. Responsible for processing data.
2. Includes:
    1. **Registers:** Temporary storage for data.
    2. **Functional Units:** Such as an Arithmetic and Logic Unit (ALU).
    3. **Switching Mechanism:** For transferring and manipulating data.



2.The Control Unit

1. Responsible for sequencing operations to be performed by the datapath.
2. Operates based on:
    1. **External Inputs.**
    2. **Results of Previous Operations.**

Instruction Execution Cycles

1. Fetch Cycle

PC: 80000

Address	Instruction
80000	add $1, $2, $3
80004	...

1. Fetch the instruction to execute

- Place **PC** into **RA** (Address Register).
- Send a read command to the memory.
- Place the content of **RD** (Data Register) into **RI** (Instruction Register).

RI: add $1, $2, $3

2. Increment the Program Counter (PC)

- Either **PC** has an incrementer built in.
- Or the ALU is used.

PC: 80004

Instruction Execution Cycles

2. Instruction Decoding (Decoder)

- Identification of an addition between two registers with the result placed in a register.

3. Data Preparation (Sequencer)

- The contents of registers $2 and $3 are placed into the two input registers of the ALU.

4. Determining What to Do (Sequencer)

- Sending the signal for the addition operation to the ALU.

5. Performing the Operation (Sequencer)

- The ALU adds the two operands and places the result in its output register.
- The content of the ALU output register is transferred to register $1.

Datapath

A **datapath** is defined by:

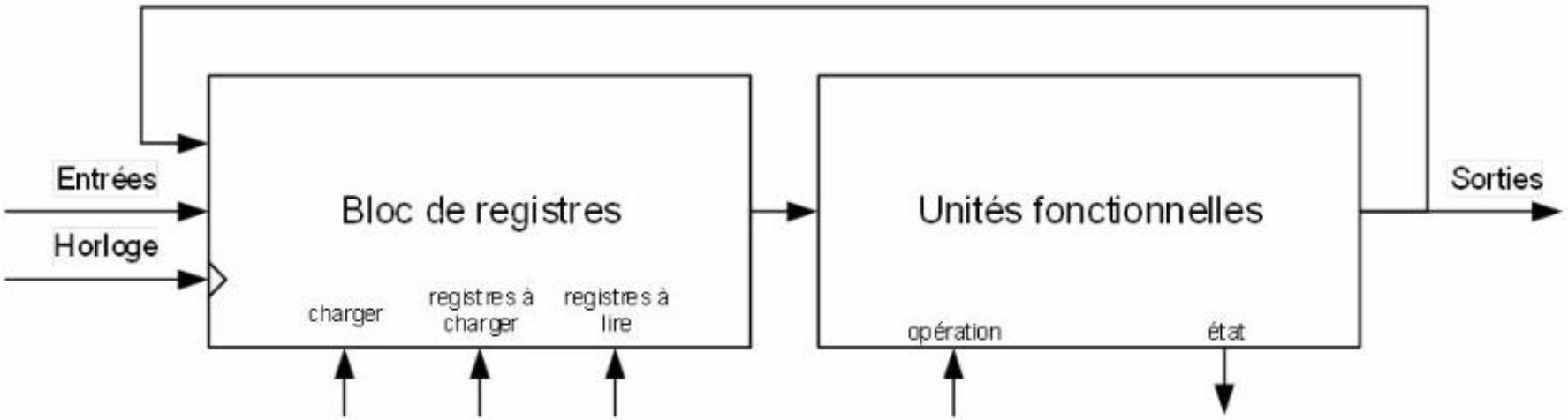
- The set of components required for the execution of an instruction: PC, ALU, register bank, memory, etc.
- Links between these components: data flow, read/write signals, multiplexing of shared units, etc.

Depending on the instructions, the required components and the existing links between them vary.

Datapath

A **datapath** has two main parts:

- A **register block** that stores the data to be processed and the results of previous operations; and,
- **Functional units** to perform operations on the data.



- **Register block** (Bloc de registres): Responsible for loading and reading registers.
- **Functional units** (Unités fonctionnelles): Responsible for operations and status management.

Memory Access Models

The processor executes operations with operands as parameters.

Several combinations are possible:

Example with an addition operation:

A=B+C Where A,B and C are values located in main memory.

Questions:

- Can they be accessed directly?
- Should they first be placed into general-purpose registers?
- Should they be placed in the ALU's accumulator?

General Memory Access Models:

Notation: (m,n) where:

m: Maximum number of operands per operation (e.g., addition).

n : Number of operands that can be accessed directly from main memory for a computation.

Memory Access Models

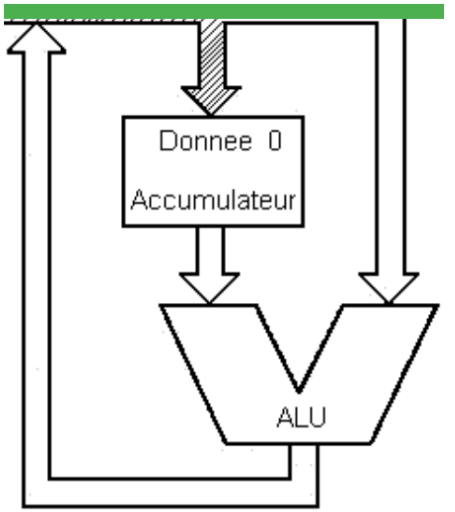
❖ Memory-Memory Model (3,3)

```
ADD  @A,  @B,  @C
```

3 operands, all directly accessible in memory.

❖ Memory-Accumulator Model (1,1)

```
LOAD  @B
ADD   @C
STORE @A
```



- The **LOAD** instruction places the content read into the accumulator register of the ALU.
- The accumulator holds the result after the calculation.
- **STORE** places it at the specified memory address.

Memory Access Models

❖ Memory-Register Model (2,1)

```
LOAD  R1,  @B      ;R1 = content at address B
ADD   R1,  @C      ; R1 = R1 + content at address C
STORE R1,  @A      ; Store the content of R1 at address A
```

Variant (2,0) possible with **ADD R1, R2**

❖ Stack Model (0,0)

Use of a stack to store operands and results.

```
PUSH @B
PUSH @C
ADD
POP  @A
```

Memory Access Models

Register-Register Model (3,0):

- No direct access to memory for a calculation operation.
- Everything passes through registers.

```
LOAD R1, @B
LOAD R2, @C
ADD R3, R1, R2
STORE R3, @A
```

- **Typical Architecture of RISC Processors**
- Currently the dominant architecture.
- Also called the **load-store model**.

Processor Performance

- Ongoing Efforts to Enhance CPU Performance

Evolution of Architectures : **Key points of this evolution:**

1. **Operating frequency**
2. **Cache memory**
3. **Parallelism and optimization of instruction sequences**
    - Pipeline
    - Superscalar and multi-core architectures

Each point influences another positively or negatively.

- Search for the best compromise.

1. Operating Frequency (Clock Speed)

Instruction Execution Time: CPU Cycle

- **Idea:** Reduce this cycle.
- **Increase the operating frequency.**

Advantages:

- More instructions executed in less time.

Disadvantages:

- **Technological and physical issues**:
  - Heat dissipation.
  - Requires processor cooling, which has its limits.

The clock period or cycle time (Tc) is essentially the time for an ALU operation. The reciprocal of the cycle time is the frequency. If the unit of time for the cycle is seconds, the frequency is in Hertz.

For example, a processor clocked at 500 MHz has a clock period:

$$P = \frac{1}{F}$$

$$T_c = \frac{1}{500 \times 10^6} = 2 \times 10^{-9} s = 2ns$$

Performance Indicator of the Central Processing Unit (CPU)

Two parameters can be used to measure a processor's performance:

1. **Response Time or Execution Time**:

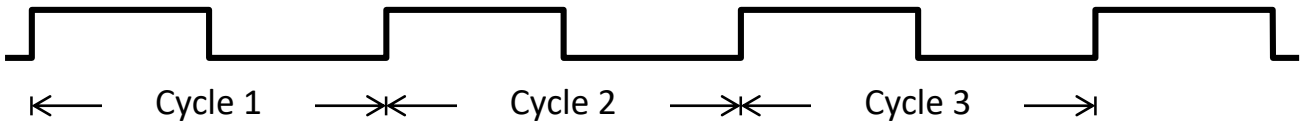
   - The elapsed time between the start and the end of a task's execution.
   - Response Time = CPU Time + Waiting Time (I/O, OS scheduling, etc.)

2. **Throughput**:

   - The total amount of work completed within a given period of time.

Clock Cycles

Clock cycle = Clock period = 1 / Clock rate



Clock rate = Clock frequency = Cycles per second

- 1 Hz = 1 cycle/sec 1 KHz = 10^3 cycles/sec
- 1 MHz = 10^6 cycles/sec 1 GHz = 10^9 cycles/sec
- 2 GHz clock has a cycle time = $1/(2\times10^9)$ = 0.5 nanosecond (ns)

We often use clock cycles to report CPU execution time

CPU Execution Time = CPU cycles × cycle time = $\frac{\text{CPU cycles}}{\text{Clock rate}}$

❖ To improve performance, we need to

- Reduce number of clock cycles required by a program, or
- Reduce clock cycle time (increase the clock rate)

Performance Equation

To execute a given program, it will require …

- Some number of machine instructions
- Some number of clock cycles
- Some number of seconds

We can relate CPU clock cycles to instruction count

$$\text{CPU cycles} = \text{Instruction Count} \times \text{CPI}$$

Performance Equation: (related to instruction count)

$$\text{Time} = \text{Instruction Count} \times \text{CPI} \times \text{cycle time}$$

Book's Definition of Performance

- For some program running on machine *X*

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

- *X* is *n* times faster than *Y*

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

This ratio indicates how much faster or slower machine X is compared to machine Y. A value greater than 1 implies that X is faster than Y, while a value less than 1 indicates that X is slower.

MIPS as a Performance Measure

- MIPS : Millions Instructions Per Second
- Sometimes used as performance metric
  - Faster machine $\Rightarrow$ larger MIPS
- MIPS specifies instruction execution rate

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} = \frac{\text{Clock Rate}}{\text{CPI} \times 10^6}$$

- We can also relate execution time to MIPS

$$\text{Execution Time} = \frac{\text{Inst Count}}{\text{MIPS} \times 10^6} = \frac{\text{Inst Count} \times \text{CPI}}{\text{Clock Rate}}$$

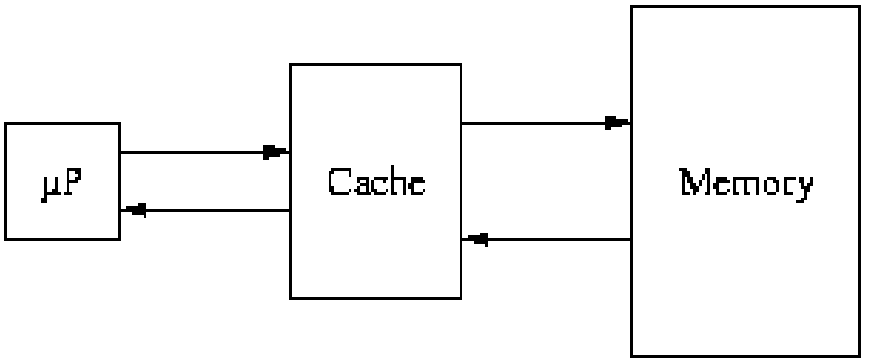
2. Cache Memory

Why? The processor requires a sustained flow of instruction and data reading.

- To avoid having to wait idly

Use of cache memory.

Cache memory holds a copy of the original data when retrieving or computing it is costly in terms of access time compared to accessing the cache. Once the data is stored in the cache, it is accessed directly from the cache rather than being retrieved or recalculated, thereby reducing the average access time.



2. Cache Memory

In microprocessors, several levels of cache are differentiated, often numbering three:

- **Level 1 cache (L1)**: The fastest and smallest cache (data cache may be separate from instruction cache).
- **Level 2 cache (L2)**: Slower but larger than L1.
- **Level 3 cache (L3)**: Even slower but significantly larger than L2.

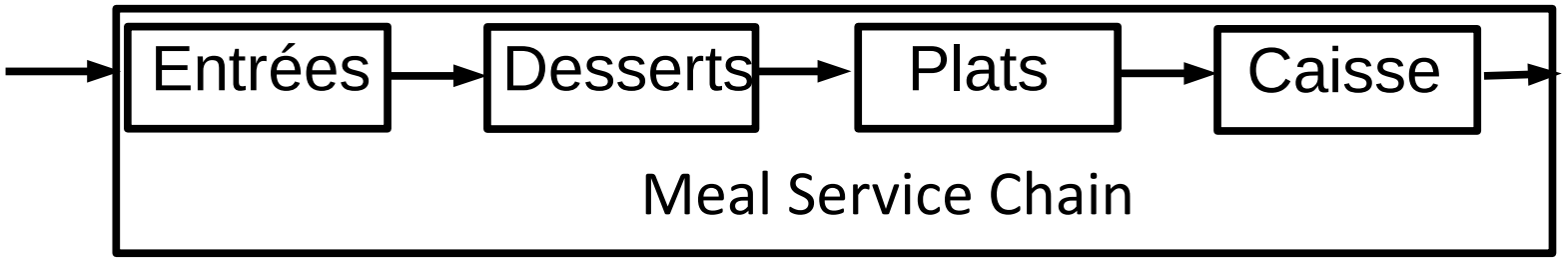
These caches may be located inside or outside the microprocessor.

3. Pipeline

Pipeline Principle Explained with an Example

University Restaurant : You pass, in order, by 4 stations:

- A display for appetizers
- A display for desserts
- A display for main dishes
- A cash register

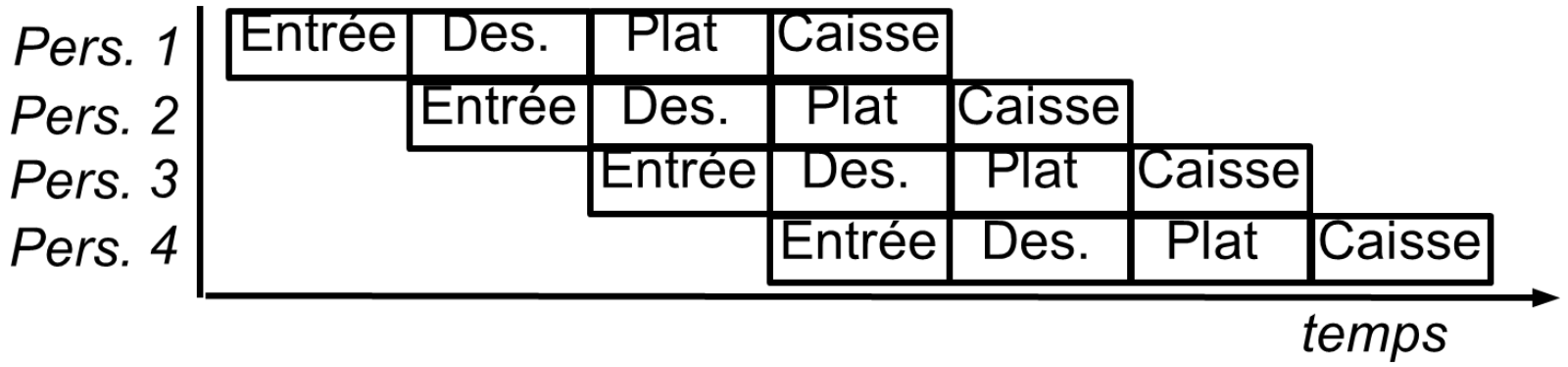


Pipeline

2 Modes of Use for Serving a Meal

- **One person at a time through the entire service chain**
  - When the person has passed through the entire chain and exited, another person enters to be served.

- **Multiple people at the same time, staggered**
  - One person at each display/station.
  - A person moves to the next station when it is free and they have finished with their current station.



Pipeline

❖ **Advantages of the Second Mode**

- Multiple people are served at the same time.
- Time-saving: more people pass through in the same amount of time.
- Better management of stations: always in use.

❖ **Disadvantages of the Second Mode**

- More difficult to "turn back" in the station chain.
- Requires additional synchronization and stations whose processing times are similar for better optimization.

Pipeline

In a processor, the use of a pipeline for executing an operation. An operation consists of several sub-operations:

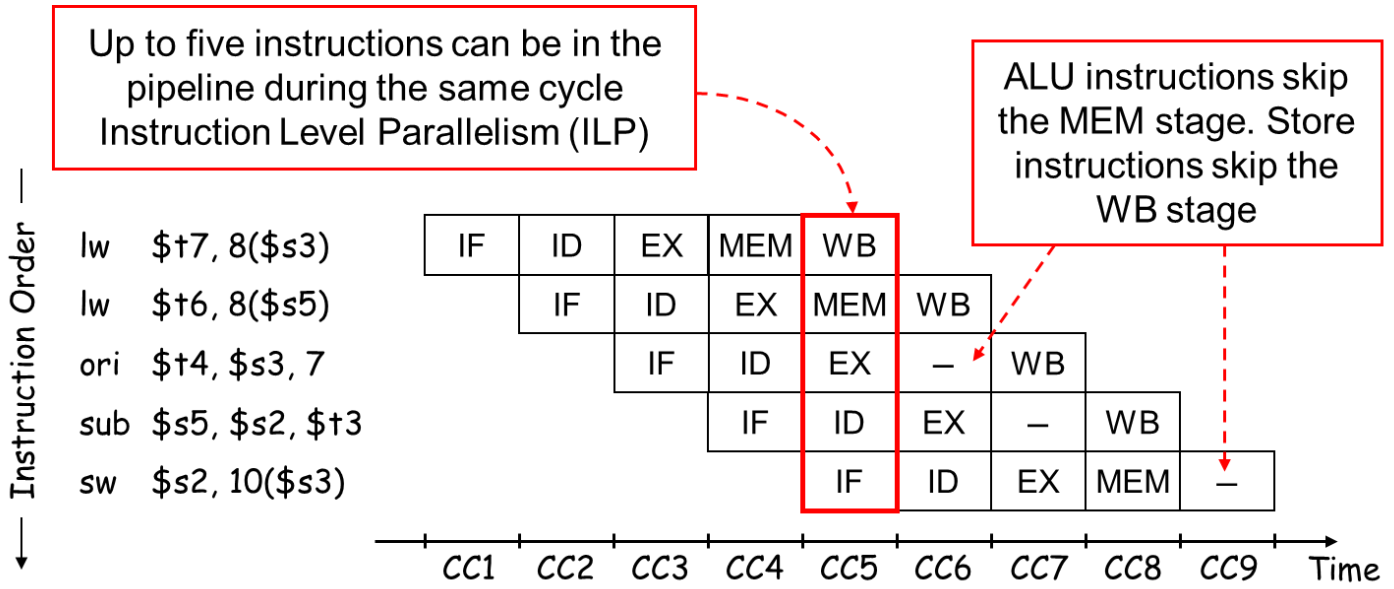
- **Pipeline** for executing these sub-operations.
- A sub-operation uses a sub-unit of the processor that is not used by other sub-operations (if possible...).

❖ **MIPS Processor ：** Five stages, one cycle per stage

1. IF: Instruction Fetch from instruction memory
2. ID: Instruction Decode, register read
3. EX: Execute operation, calculate load/store address or J/Br address
4. MEM: Memory access for load and store
5. WB: Write Back result to register

Pipeline

Staggered execution of multiple instructions at the same time



Significant gain using the pipeline

- **Without**: Sequential execution of 2 instructions in 10 cycles.
- **With**: Parallel execution of 5 instructions in 9 cycles.
- **Theoretical gain**, as there are many practical problems.
- **For optimization**: Transition time in each stage should be identical (or very close).

Pipeline - Depth

In the context of processors, the **pipeline depth** refers to the number of stages or steps in the pipeline. Each stage performs a specific part of the instruction execution process.

- **Currently in practice**: Around 15 stages
- **Examples of pipeline depths (number of stages)**
- **Intel Processors**:
  - i3, i5, i7: 14
  - Core 2 Duo and Mobile: 14 and 12
  - P4 Prescott: 31
  - P4 (before Prescott architecture): 20
  - Intel P3: 10
- **AMD Processors**:
  - K10: 16
  - Athlon 64: 12
  - AMD Athlon XP: 10
- **RISC-type Processors**:
  - Sun UltraSparc IV: 14
  - IBM Power PC 970: 16

Pipeline - Hazards

In the context of processor pipelines, **hazards** are situations that can prevent the next instruction in the pipeline from executing during its designated clock cycle. Hazards lead to delays, known as **pipeline stalls**. There are three main types of hazards:

1. **Data Hazards**:

Occur when instructions depend on the result of previous instructions that haven't been completed yet.

- **Example**: An instruction needs to use a value that hasn't been calculated yet.

2. **Control Hazards**:

Arise from changes in the control flow, such as jumps or branches.

- **Example**: A branch prediction turns out to be incorrect, and the pipeline must be flushed and restarted with the correct instructions.

3. **Structural Hazards**:

Happen when hardware resources are insufficient to execute all instructions in the pipeline simultaneously.

- **Example**: Multiple instructions need to access the same memory or register at the same time.

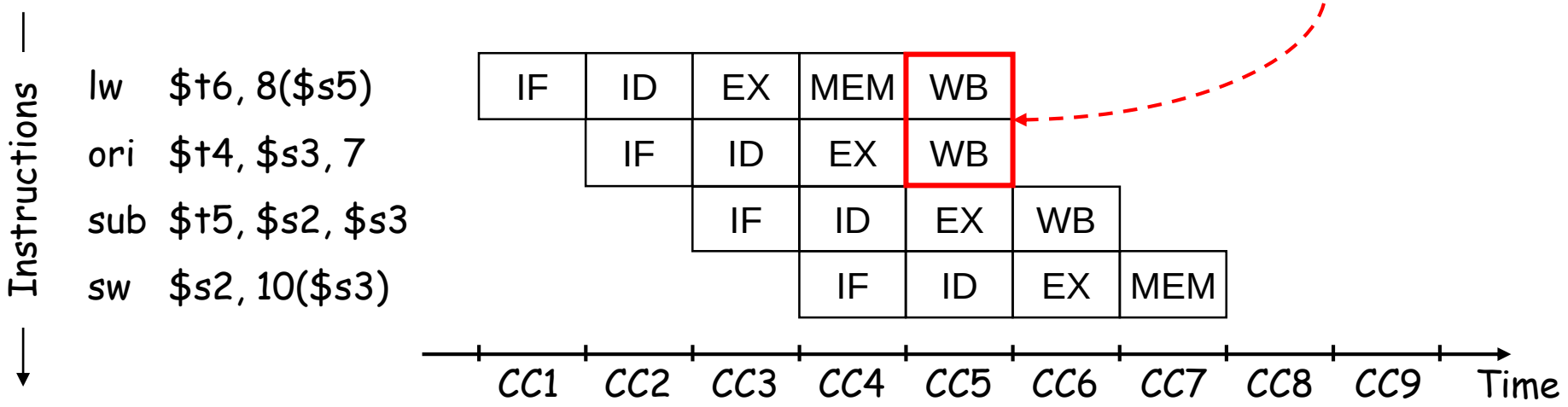
❖ **Hazards complicate pipeline control and limit performance**

Structural Hazards

- Problem
  - Attempt to use the same hardware resource by two different instructions during the same clock cycle
- Example
  - Writing back ALU result in stage 4
  - Conflict with writing load data in stage 5

Structural Hazard

Two instructions are attempting to write the register file during same cycle



Resolving Structural Hazards

- Serious Hazard:
  - Hazard cannot be ignored
- Solution 1: Delay Access to Resource
  - Must have mechanism to delay instruction access to resource
  - Delay all write backs to the register file to stage 5
    - ALU instructions bypass stage 4 (memory) without doing anything
- Solution 2: Add more hardware resources (more costly)
  - Add more hardware to eliminate the structural hazard
  - Redesign the register file to have two write ports
    - First write port can be used to write back ALU results in stage 4
    - Second write port can be used to write back load data in stage 5

Data Hazards

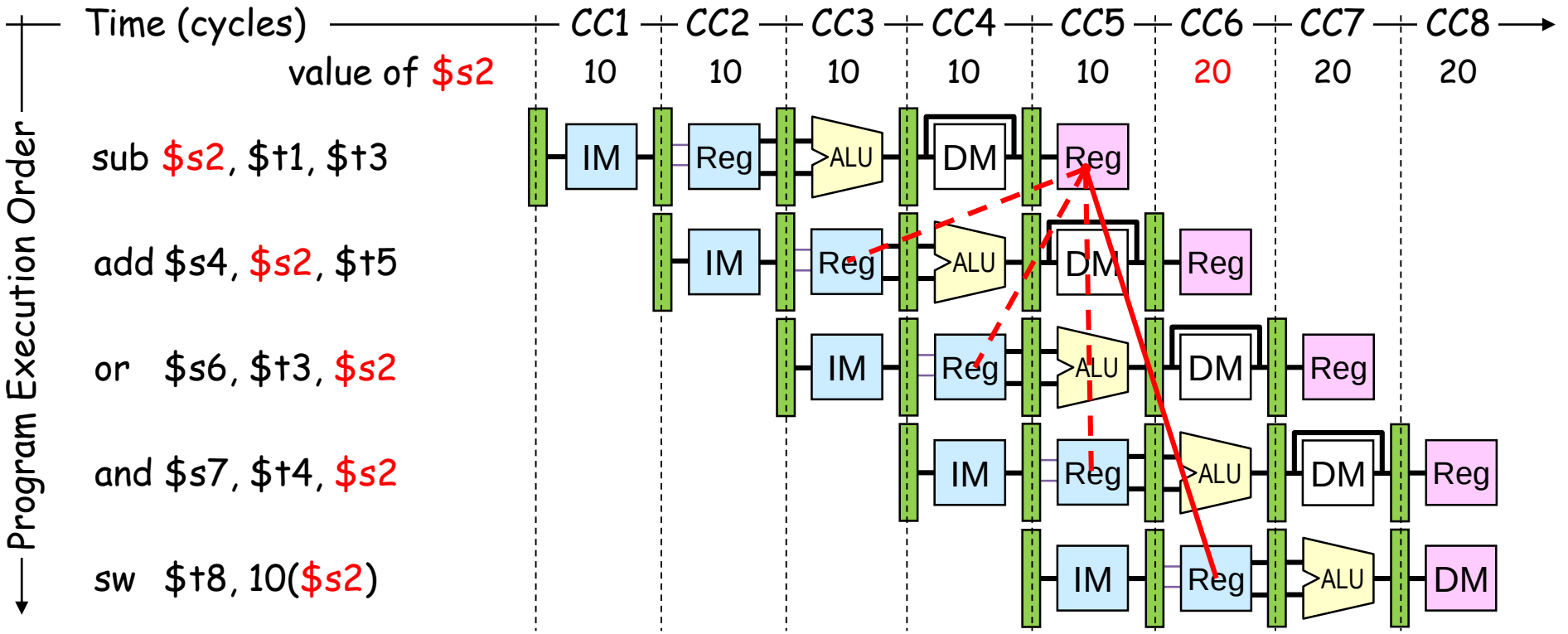
- Dependency between instructions causes a data hazard
- The dependent instructions are close to each other
  - Pipelined execution might change the order of operand access
- Read After Write – RAW Hazard
  - Given two instructions *I* and *J*, where *I* comes before *J*
  - Instruction *J* should read an operand after it is written by *I*
  - Called a data dependence in compiler terminology

```
I: add $s1, $s2, $s3        # $s1 is written

J: sub $s4, $s1, $s3        # $s1 is read
```

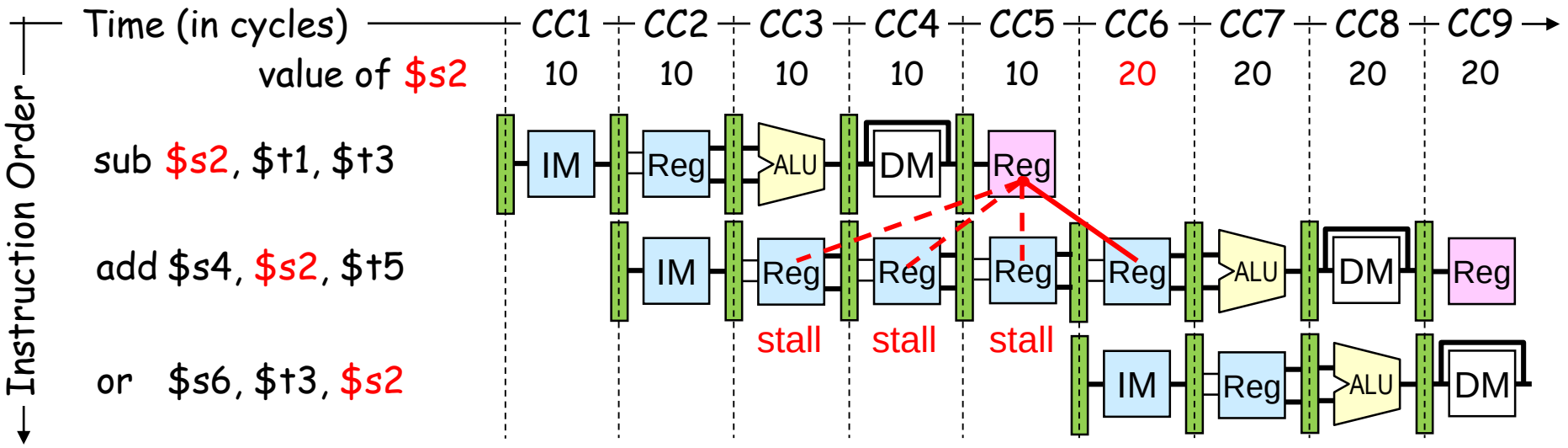
  - Hazard occurs when *J* reads the operand before *I* writes it

Example of a RAW Data Hazard



- Result of sub is needed by add, or, and, & sw instructions
- Instructions add & or will read old value of $s2 from reg file
- During CC5, $s2 is written at end of cycle, old value is read

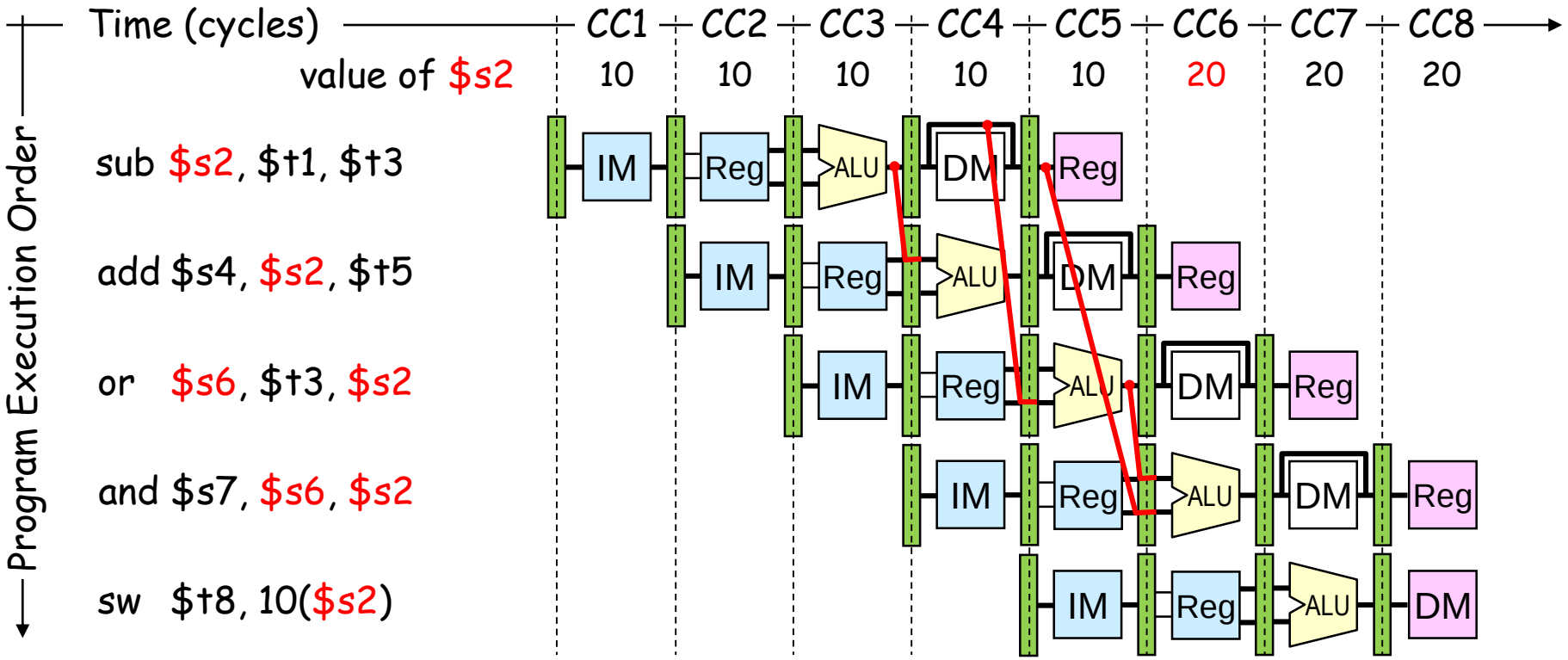
Solution 1: Stalling the Pipeline



- Three stall cycles during CC3 thru CC5 (wasting 3 cycles)
  - The 3 stall cycles delay the execution of **add** and the fetching of **or**
  - The 3 stall cycles insert 3 bubbles (No operations) into the ALU
- The **add** instruction remains in the second stage until CC6
- The **or** instruction is not fetched until CC6

Solution 2: Forwarding ALU Result

- The ALU result is forwarded (fed back) to the Register File Output
  - No bubbles are inserted into the pipeline and no cycles are wasted
- ALU result is forwarded from ALU, MEM, and WB stages



RAW Hazard Detection

- Current instruction is being decoded in the Decode stage
- Previous instruction is in the Execute stage
- Second previous instruction is in the Memory stage
- Third previous instruction is in the Write Back stage

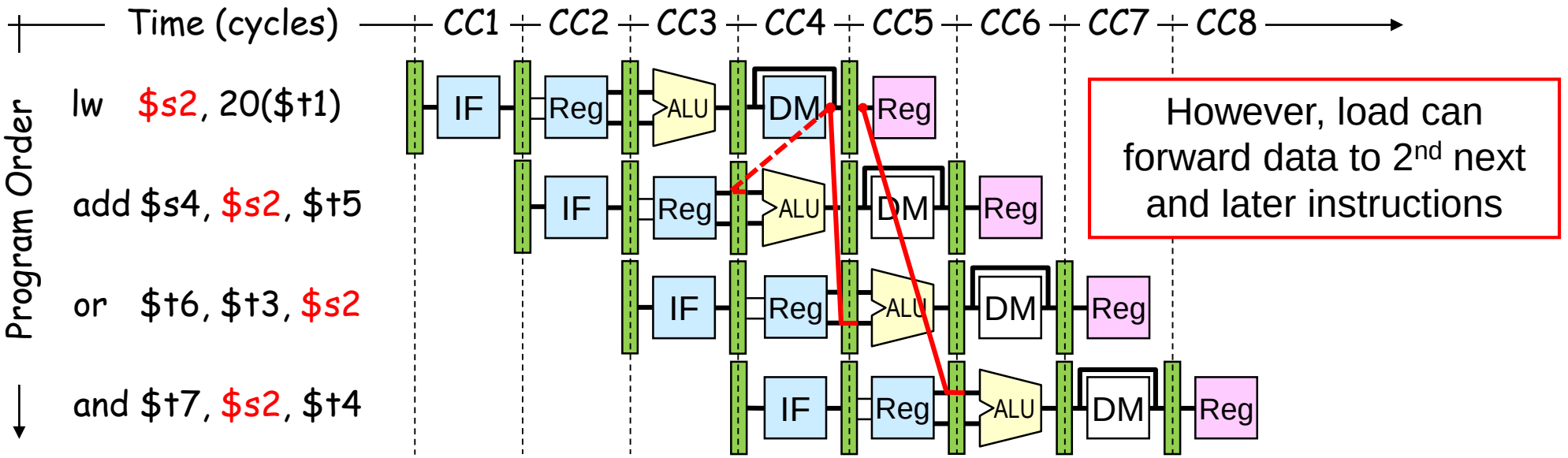
```
If      ((Rs != 0) and (Rs == Rd2) and (EX.RegWr))  ForwardA = 1
Else if ((Rs != 0) and (Rs == Rd3) and (MEM.RegWr)) ForwardA = 2
Else if ((Rs != 0) and (Rs == Rd4) and (WB.RegWr))  ForwardA = 3
Else    ForwardA = 0

If      ((Rt != 0) and (Rt == Rd2) and (EX.RegWr))  ForwardB = 1
Else if ((Rt != 0) and (Rt == Rd3) and (MEM.RegWr)) ForwardB = 2
Else if ((Rt != 0) and (Rt == Rd4) and (WB.RegWr))  ForwardB = 3
Else    ForwardB = 0
```

Load Delay

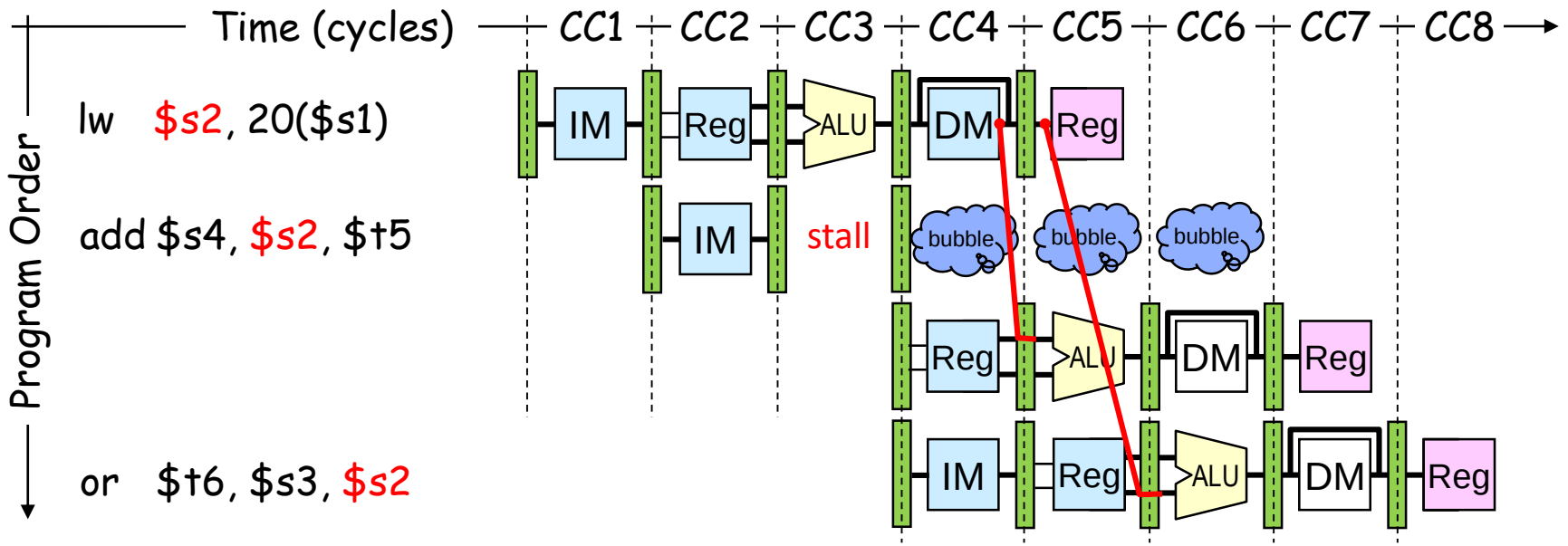
-

- Unfortunately, not all data hazards can be forwarded
  - Load has a delay that cannot be eliminated by forwarding
- In the example shown below …
  - The LW instruction does not read data until end of CC4
  - Cannot forward data to ADD at end of CC3 - NOT possible



Stall the Pipeline for one Cycle

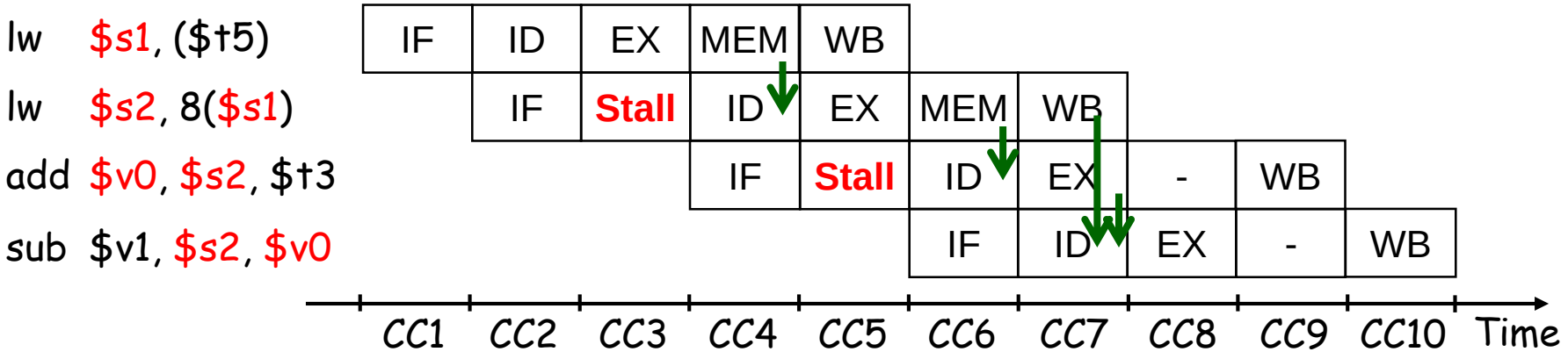
- ADD instruction depends on LW ➔ stall at CC3
  - Allow Load instruction in ALU stage to proceed
  - Freeze PC and Instruction registers (NO instruction is fetched)
  - Introduce a bubble into the ALU stage (bubble is a NO-OP)
- Load can forward data to next instruction after delaying it



Showing Stall Cycles

- Stall cycles can be shown on instruction-time diagram
- Hazard is detected in the Decode stage
- Stall indicates that instruction is delayed
- Instruction fetching is also delayed after a stall
- Example:

Data forwarding is shown using **green arrows**



Code Scheduling to Avoid Stalls

- Compilers reorder code in a way to avoid load stalls
- Consider the translation of the following statements:

```
A = B + C; D = E - F;  // A thru F are in Memory
```

- Slow code:

```
lw   $t0, 4($s0)     # &B = 4($s0)
lw   $t1, 8($s0)     # &C = 8($s0)
add  $t2, $t0, $t1   # stall cycle
sw   $t2, 0($s0)     # &A = 0($s0)
lw   $t3, 16($s0)    # &E = 16($s0)
lw   $t4, 20($s0)    # &F = 20($s0)
sub  $t5, $t3, $t4   # stall cycle
sw   $t5, 12($0)     # &D = 12($0)
```

❖ Fast code: No Stalls

```
lw   $t0, 4($s0)
lw   $t1, 8($s0)
lw   $t3, 16($s0)
lw   $t4, 20($s0)
add  $t2, $t0, $t1
sw   $t2, 0($s0)
sub  $t5, $t3, $t4
sw   $t5, 12($s0)
```

Name Dependence: Write After Read

- Instruction **J** should write its result after it is read by **I**
- Called anti-dependence by compiler writers

```
I: sub $t4, $t1, $t3    # $t1 is read
J: add $t1, $t2, $t3    # $t1 is written
```

- Results from reuse of the name **$t1**
- NOT a data hazard in the 5-stage pipeline because:
  - Reads are always in stage 2
  - Writes are always in stage 5, and
  - Instructions are processed in order
- Anti-dependence can be eliminated by **renaming**
  - Use a different destination register for **add** (eg, **$t5**)

Name Dependence: Write After Write

- Same destination register is written by two instructions
- Called output-dependence in compiler terminology

```
I: sub $t1, $t4, $t3          # $t1 is written
J: add $t1, $t2, $t3          # $t1 is written again
```

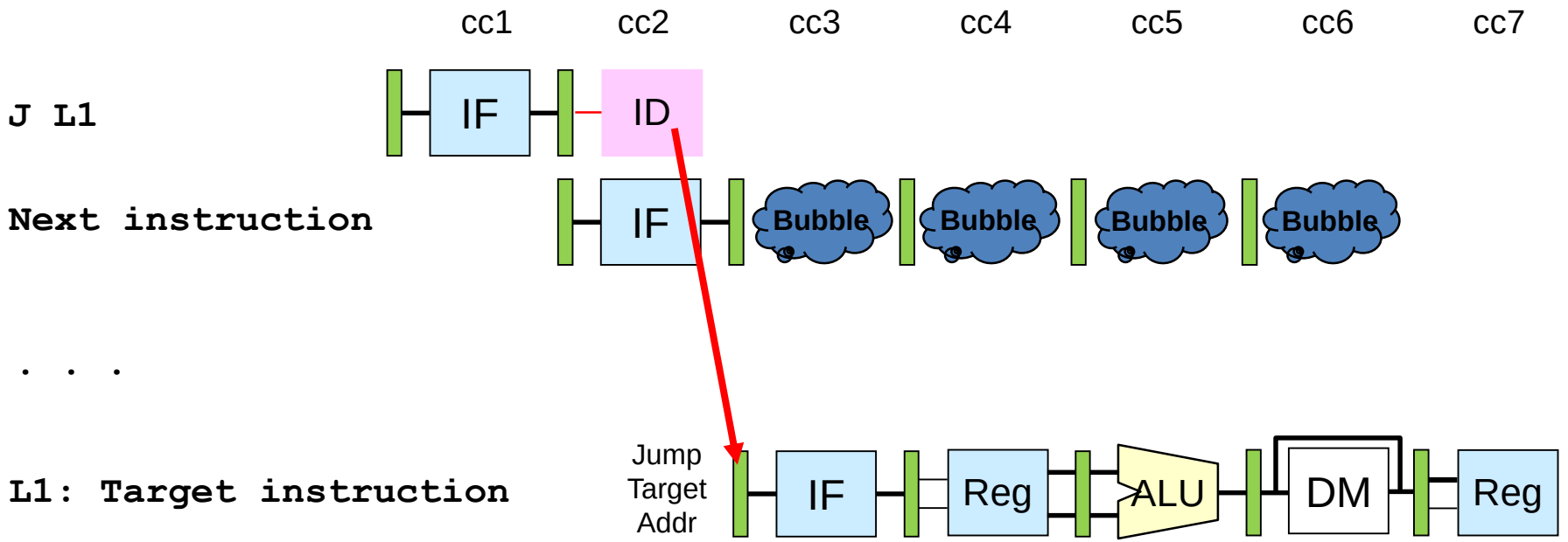
- Not a data hazard in the 5-stage pipeline because:
  - All writes are ordered and always take place in stage 5
- However, can be a hazard in more complex pipelines
  - If instructions are allowed to complete out of order, and
  - Instruction **J** completes and writes **$t1** before instruction **I**
- Output dependence can be eliminated by **renaming $t1**
- Read After Read is NOT a name dependence

Control Hazards

- Jump and Branch can cause great performance loss
- Jump instruction needs only the jump target address
- Branch instruction needs two things:
  - Branch Result — Taken or Not Taken
  - Branch Target Address
    - PC + 4 — If Branch is NOT taken
    - PC + 4 + 4 × immediate — If Branch is Taken
- Jump and Branch targets are computed in the ID stage
  - At which point a new instruction is already being fetched
  - Jump Instruction: 1-cycle delay
  - Branch: 2-cycle delay for branch result (taken or not taken)

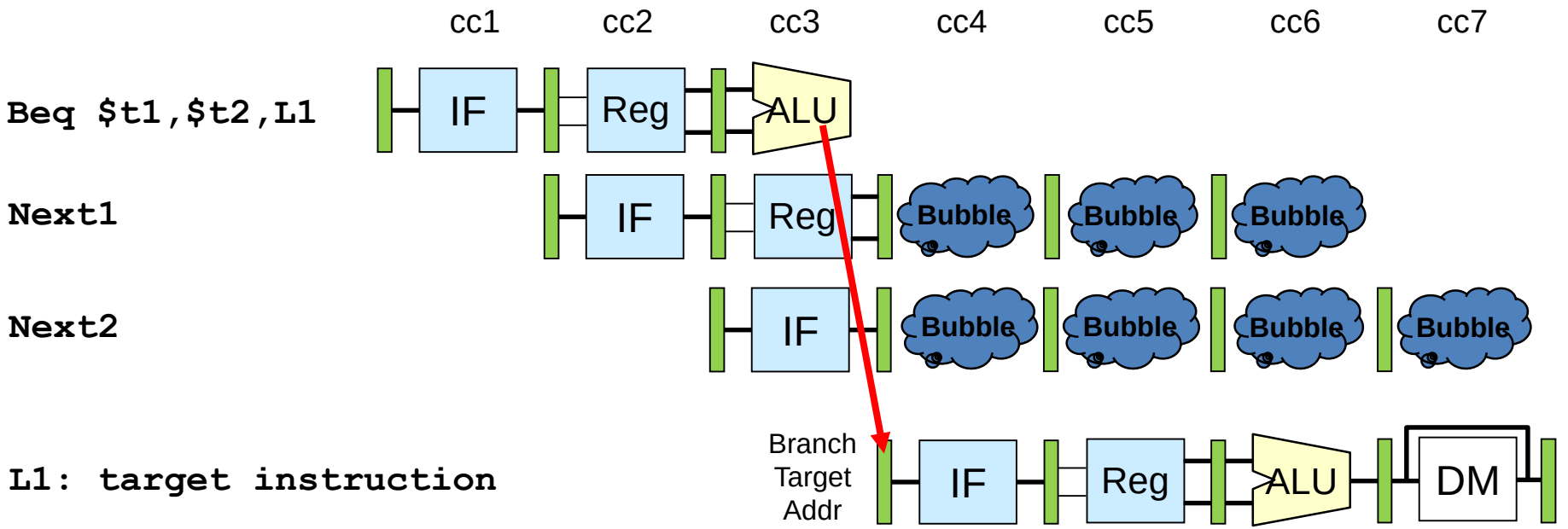
1-Cycle Jump Delay

- Control logic detects a Jump instruction in the 2nd Stage
- Next instruction is fetched anyway
- Convert Next instruction into bubble (Jump is always taken)



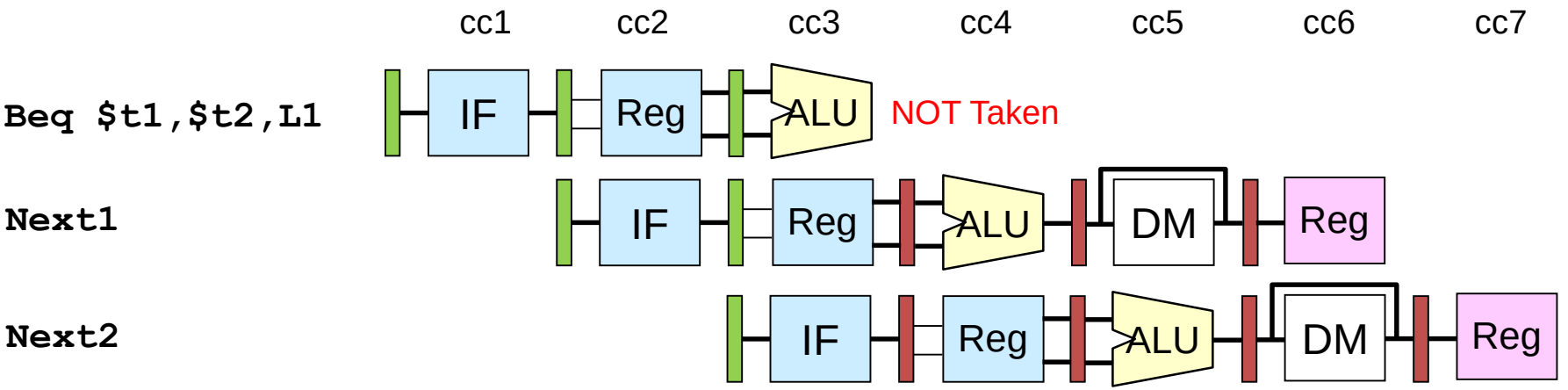
2-Cycle Branch Delay

- Control logic detects a Branch instruction in the 2nd Stage
- ALU computes the Branch outcome in the 3rd Stage
- Next1 and Next2 instructions will be fetched anyway
- Convert Next1 and Next2 into bubbles if branch is taken



Predict Branch NOT Taken

- Branches can be predicted to be NOT taken
- If branch outcome is NOT taken then
  - Next1 and Next2 instructions can be executed
  - Do not convert Next1 & Next2 into bubbles
  - No wasted cycles



Jump and Branch Impact on CPI

- Base CPI = 1 without counting jump and branch
- Unconditional Jump = 5%, Conditional branch = 20%
- 90% of conditional branches are taken
- Jump kills next instruction, Taken Branch kills next two
- What is the effect of jump and branch on the CPI?

Solution:

- Jump adds 1 wasted cycle for 5% of instructions = 1 × 0.05
- Branch adds 2 wasted cycles for 20% × 90% of instructions

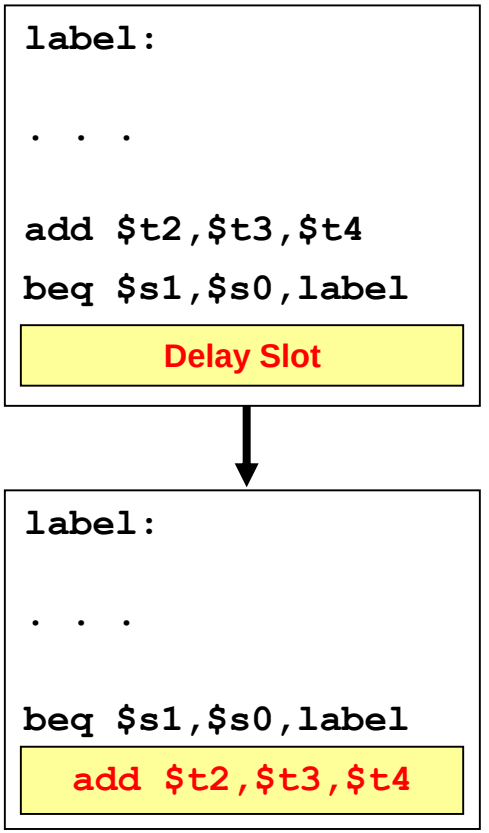
  = 2 × 0.2 × 0.9 = 0.36
- New CPI = 1 + 0.05 + 0.36 = 1.41 (due to wasted cycles)

Branch Hazard Alternatives

- Predict Branch Not Taken (previously discussed)
  - Successor instruction is already fetched
  - Do NOT kill instructions if the branch is NOT taken
  - Kill only instructions appearing after Jump or taken branch
- Delayed Branch
  - Define branch to take place AFTER the next instruction
  - Compiler/assembler fills the branch delay slot (for 1 delay cycle)
- Dynamic Branch Prediction
  - Loop branches are taken most of time
  - Must reduce the branch delay to 0, but how?
  - How to predict branch behavior at runtime?

Delayed Branch

- Define branch to take place after the next instruction
- MIPS defines one delay slot
  - Reduces branch penalty
- Compiler fills the branch delay slot
  - By selecting an independent instruction from before the branch
  - Must be okay to execute instruction in the delay slot whether branch is taken or not
- If no instruction is found
  - Compiler fills delay slot with a NO-OP



Drawback of Delayed Branching

- New meaning for branch instruction
  - Branching takes place after next instruction (Not immediately!)
- Impacts software and compiler
  - Compiler is responsible to fill the branch delay slot
- However, modern processors are deeply pipelined
  - Branch penalty is multiple cycles in deep pipelines
  - Multiple delay slots are difficult to fill with useful instructions
- MIPS used delayed branching in earlier pipelines
  - However, delayed branching lost popularity in recent processors
  - Dynamic branch prediction has replaced delayed branching

Zero-Delayed Branching

- How to achieve zero delay for a jump or a taken branch?
  - Jump or branch target address is computed in the ID stage
  - Next instruction has already been fetched in the IF stage

Solution

- Introduce a Branch Target Buffer (BTB) in the IF stage
  - Store the target address of recent branch and jump instructions
- Use the lower bits of the PC to index the BTB
  - Each BTB entry stores Branch/Jump address & Target Address
  - Check the PC to see if the instruction being fetched is a branch
  - Update the PC using the target address stored in the BTB

In Summary

- Three types of pipeline hazards
  - Structural hazards: conflicts using a resource during same cycle
  - Data hazards: caused by data dependencies between instructions
  - Control hazards: caused by branch and jump instructions
- Hazards limit the performance and complicate the design
  - Structural hazards: eliminated by careful design or more hardware
  - Data hazards are eliminated by forwarding
  - However, load delay cannot be eliminated and stalls the pipeline
  - Delayed branching reduces branch delay but needs compiler support
  - BTB with branch prediction can reduce branch delay to zero
  - Branch misprediction should kill the wrongly fetched instructions