COMP2611: Computer Organization

Introduction to the Pipelined Processor

- Two techniques for designing high-performance processors:
  - **Pipelining**
  - **Multiprocessing**

- Both techniques exploit **parallelism**:
  - **Pipelining**: parallelism **among multiple instructions**
  - **Multiprocessing**: parallelism **among multiple processors**

- In this course, we only focus on **pipelining**



Single Cycle Datapath: the problem

- Single Cycle Datapath: simple but inefficient. Longest path (consists of 5 execution stages) determines the duration of the clock cycle
  - Not every instruction needs all the 5 stages, yet every instruction takes a complete clock cycle => **unnecessary performance penalty!**

- Multi-cycle Datapath: more efficient, but still wastes resources
  - **Functional units may remain idle in some clock cycles** until the completion of the Instruction (e.g., ALU unused in stages 4 and 5, the register file is unused in stages 3, 4, 5)

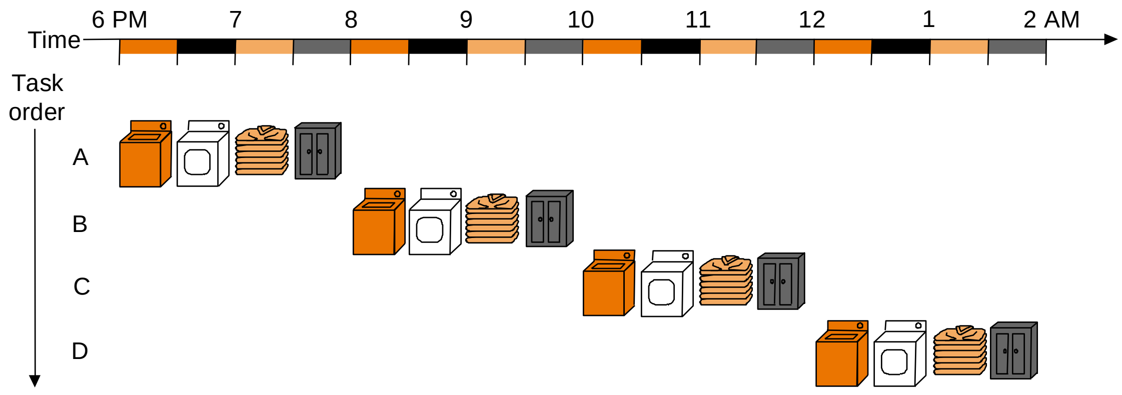
Pipelining can improve the performance further, in terms of throughput and efficiency by reusing functional units for other instructions

1- Introduction to pipelining

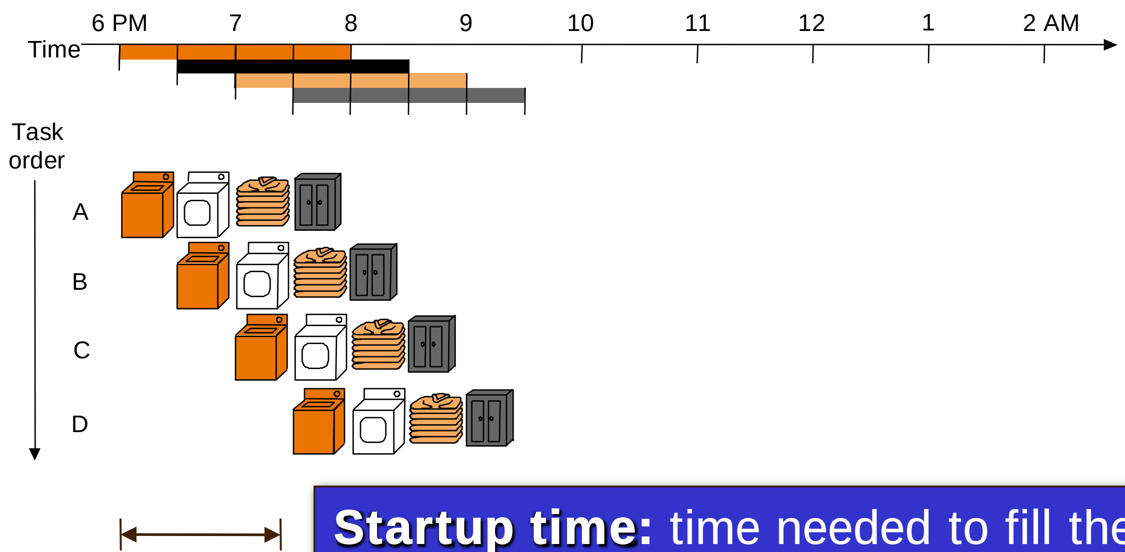
Analogy: Pipelining Laundry

- Let's say we have 4 batches of laundry work

In serial



Pipeline



Startup time: time needed to fill the pipeline

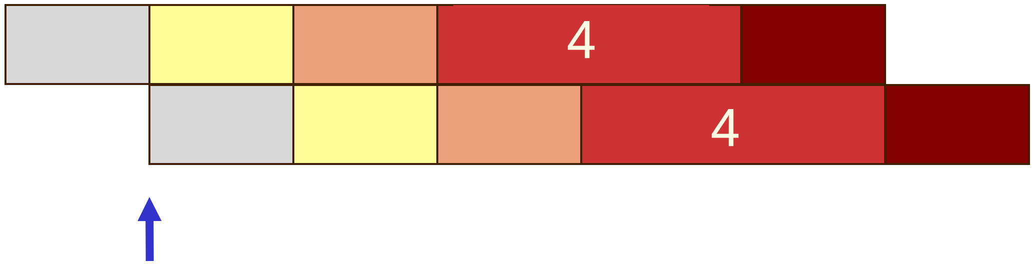
Pipelining Principles

Key characteristics:

- **Multiple tasks** are processed simultaneously
- Ideally, these tasks should be **independent** of each other
- Pipelining **does not help the latency** of a single task
- But, it **helps the throughput** of the entire workload
- Completion order in pipelined execution = that in sequential execution

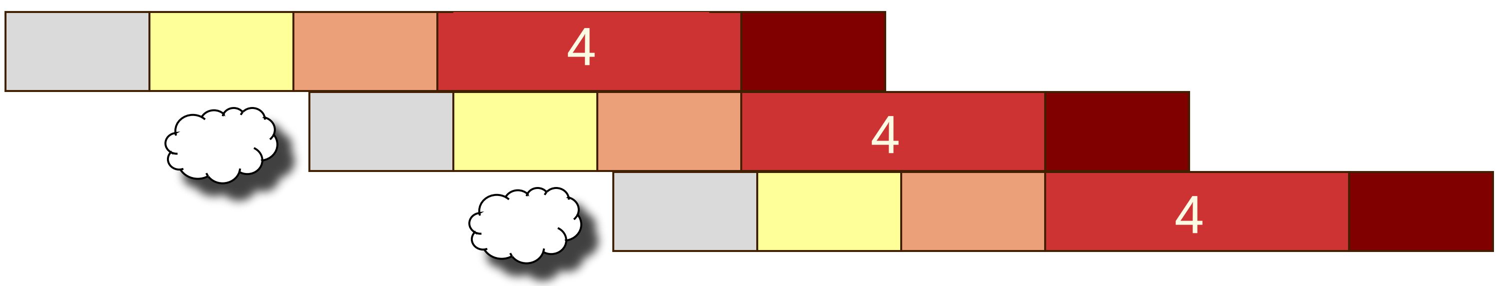
How much can a pipeline improve?

- **Potential speedup = number of pipeline stages**
- The pipeline rate is limited by the slowest pipeline stage
  - Unbalanced lengths of pipeline stages can reduce speedup. Why?



- Can I align the pipeline stages like the above?
- Answer: NO because the tasks executing in parallel are not independent (task 4 overlaps task 4)

- The condition to align is to make sure NO OVERLAP of any stages?



MIPS Pipeline: the performance benefit

- Pipeline performance (example):

  Assume we require:

  - 100 picoseconds for register read or write
  - 200 picoseconds for all other stages

Instruction	Instruction fetch	Register read	ALU op	Memory access	Register write	Total time
Load Word (lw)	200ps	100 ps	200ps	200ps	100 ps	800ps
Store Word (sw)	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
Branch (beq)	200ps	100 ps	200ps			500ps

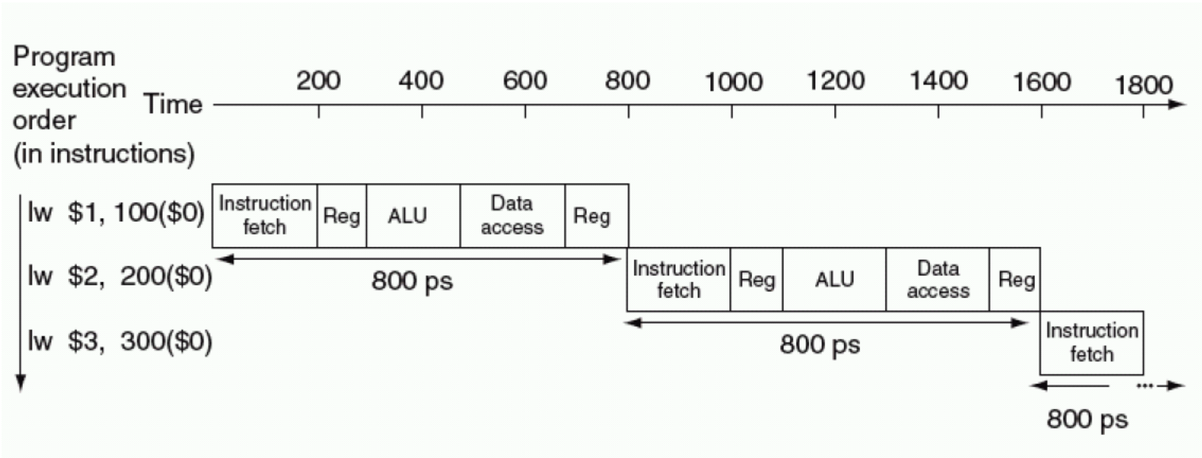
MIPS Pipeline: the performance benefit

- The instruction **delays** in the example:
  - 800 ps (single cycle datapath)
  - 1000 ps (pipelined datapath)
- The instruction **throughputs** in the example:
  - 1 instruction per 800 ps (single cycle datapath)
  - 1 instruction per 200 ps (long time average for pipelined datapath)
- Pipelining does not improve the latency of a single instruction, it improves the throughput of the system (i.e., the datapath)

- In general (ideally), if we have a N stage pipeline:
  - We need N-1 cycles to fill the pipeline,
  - Then one instruction will finish per cycle
  - So the throughput is: Clock Rate x IC/(IC + N - 1), with IC >> N

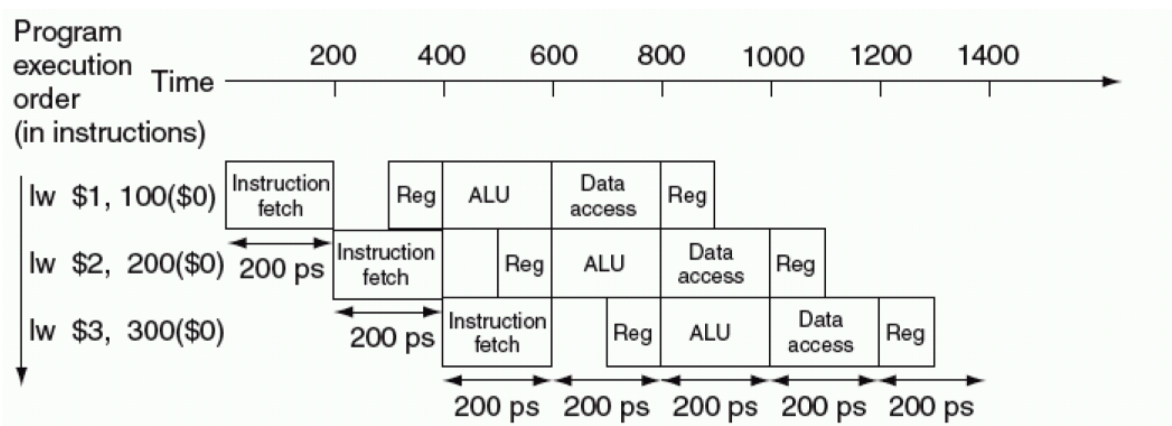
MIPS Pipeline: the performance benefit

- Single-cycle, non-pipelined execution:



Total execution time = 2400 ps

- Multicycle: > 2400 ps
- Pipelined execution



Total execution time =1400 ps

A program, 1000 instructions, all independent of each other
30% take 30ns, 40% take 40ns, 30% take 50ns

Single-cycle implementation, (cycle time = 50ns)

- CPI = 1.0,
- Time = # of instr. * CPI * cycle time = 1000 * 1.0 * 50ns = 50ms

5-stage multi-cycle implementation, (cycle time = 10ns)

- CPI = 0.3 * 3 + 0.4 * 4 + 0.3 * 5 = 4.0
- Time = 1000 * 4.0 * 10ns = 40ms

5-stage pipeline implementation, (cycle time = 10ns)

- CPI = (4 cycles /*startup*/ + 1000 cycles /*pipelined*/) / 1000
  = 1.004
- Time = 1000 * 1.004 * 10ns = 10.04ms

MIPS Pipeline

ISA design affects the complexity of pipeline implementation.

MIPS ISA is designed for pipelining

- **All instruction are of the same length** (32-bit)
  - Easy to fetch one instruction in first stage of the pipeline and decode it in the second
- **It has just a few similar instruction formats**
  - With the source register fields being located in the same place in all instructions, 2nd stage can read the register file while decoding the type of instruction just fetched
- **Memory operands only appear in loads and stores**
  - We can use the execute stage to calculate the memory address and then access memory in the following stage
- **Alignment of memory operands on word boundaries**
  - We need not worry about a single data transfer instruction requiring two memory accesses; the data can be transferred between processor and memory in a single pipeline stage

MIPS Pipeline

Execution of each instruction is broken into 5 stages: (in the order of execution)

- **IF** : Fetch the instruction from memory
- **ID** : Instruction decode & register read
- **EX** : Perform ALU operation
- **MEM** : Memory access (if necessary)
- **WB** : Write result back to register

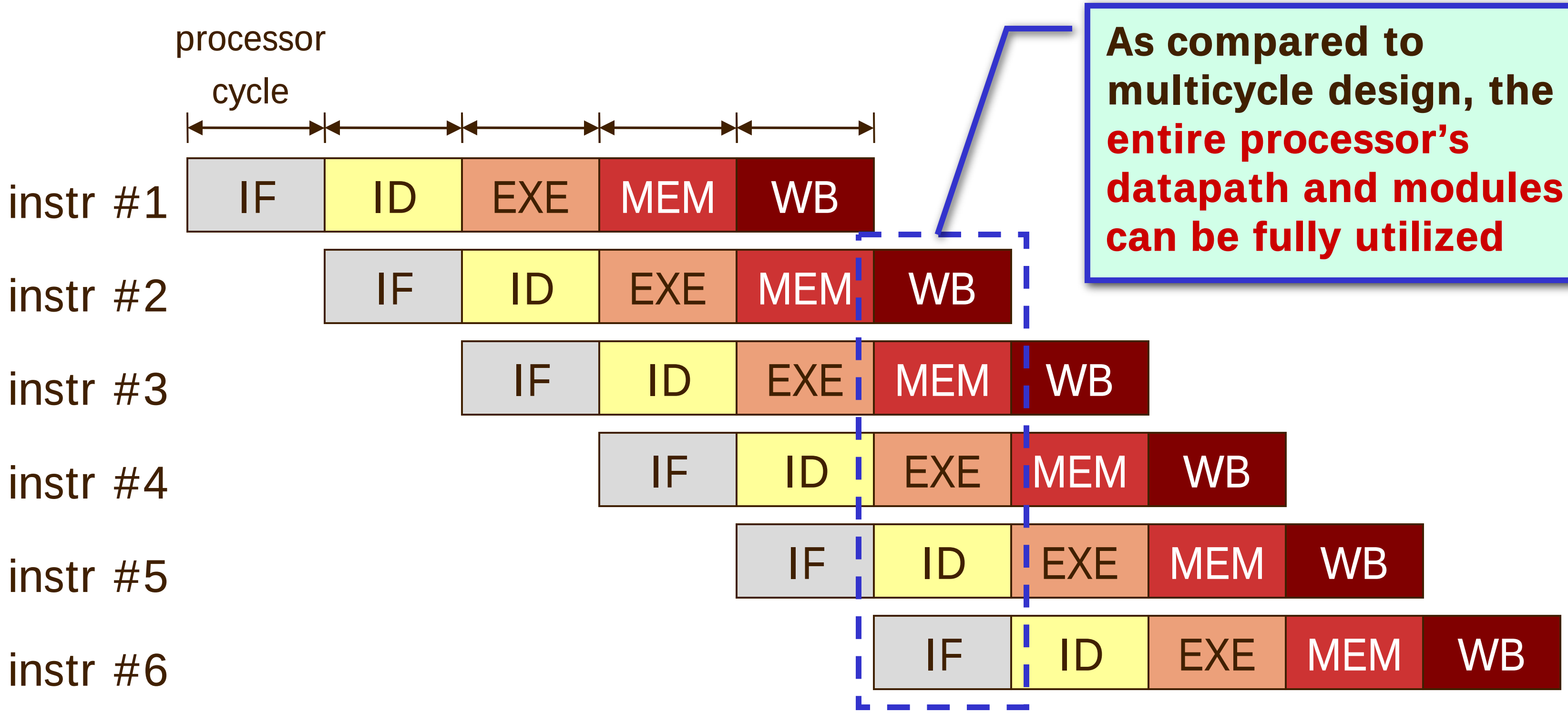
Each stage uses a different hardware unit and takes one clock cycle to complete.

- Instructions are allowed to share the different hardware units of the datapath as long as they are using the different hardware units (pipelining).

Real Stuff: Pipelining Instructions

In pipelined processor,

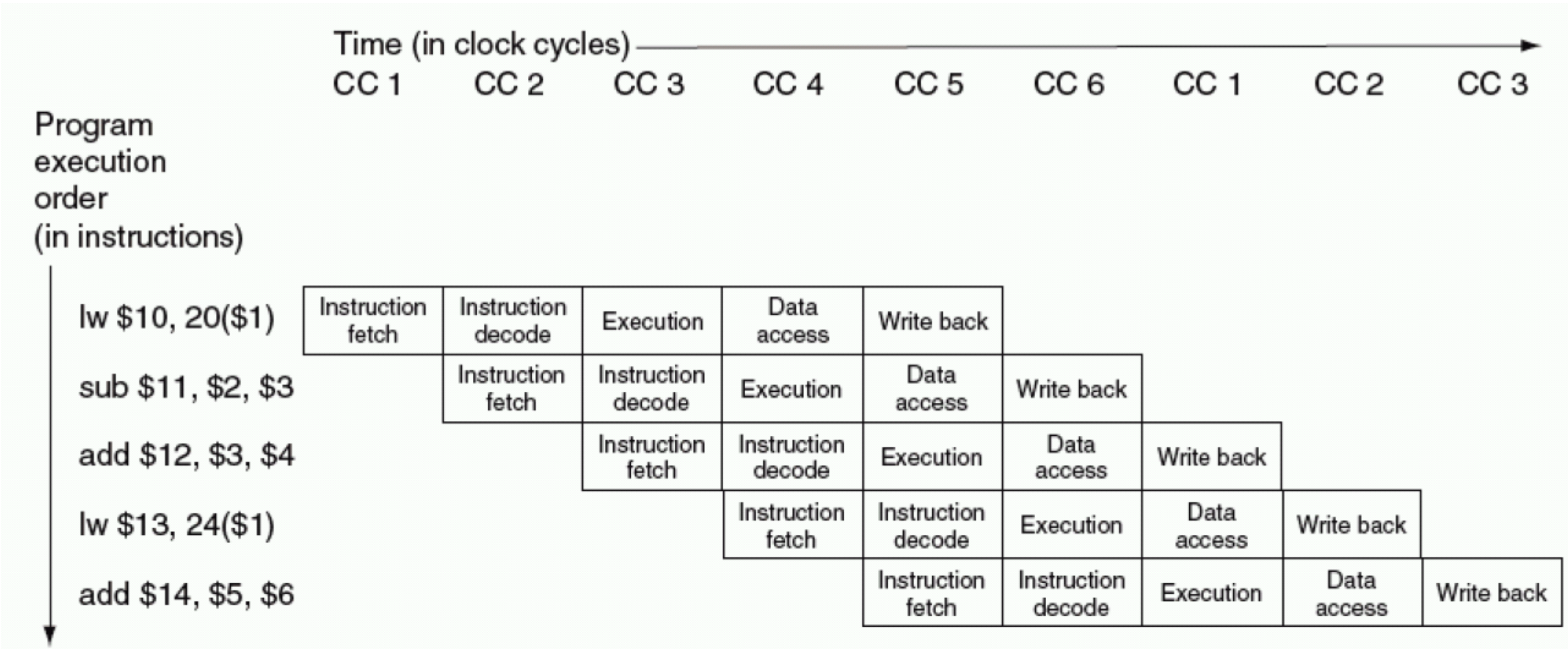
- ❑ Each instruction takes multiple steps (like **multicycle approach**)
- ❑ Each step is independent of each other and takes different datapath



As compared to multicycle design, the entire processor's datapath and modules can be fully utilized

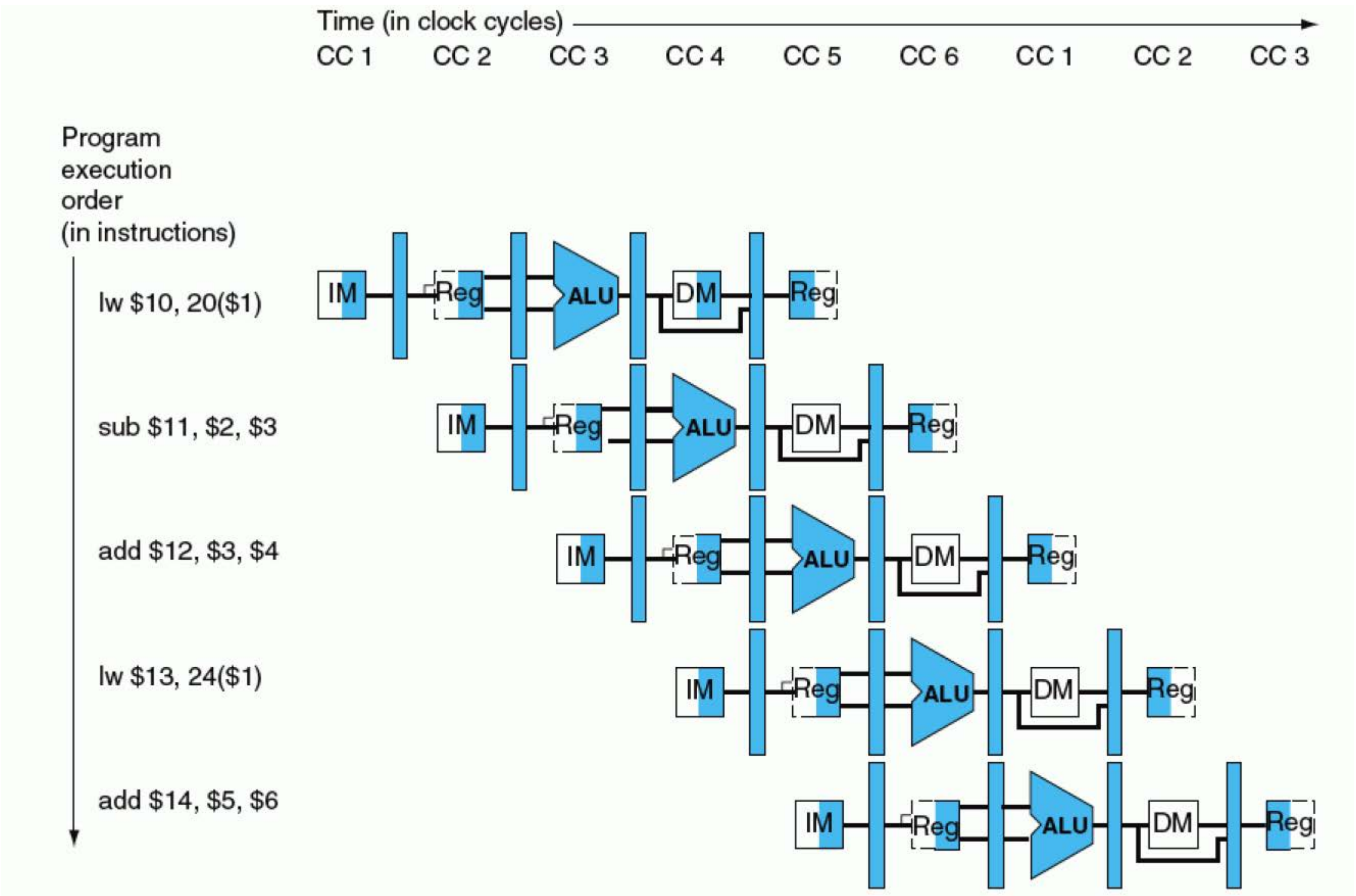
- ❑ At each cycle, one instruction is fetched and sent to the processor
- ❑ Ideally, after pipeline is fully filled, one instruction completes each cycle

- The following diagram shows the execution of a series of instructions.

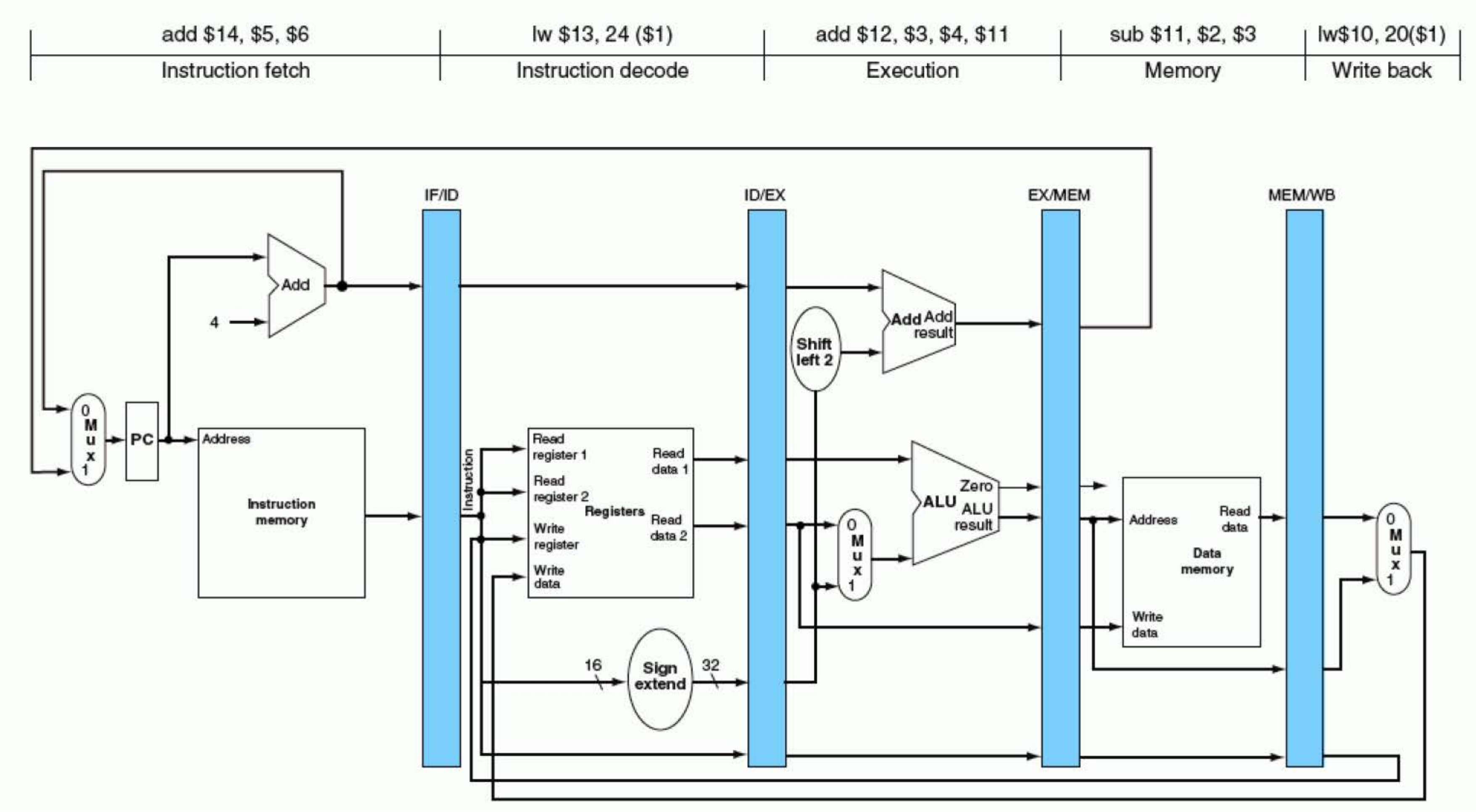


Multi-clock-cycle pipeline diagram: graphical view

- The multi-clock-cycle form showing the hardware utilizations.



Single-clock-cycle diagram in CC5



2- Dependencies and Hazards

Dependences in Programs

Sometimes these dependences cause the pipeline to not fully fill

- Execution stops to wait for **data** or **control** to be produced
  - i.e. next instruction cannot be executed in the next cycle

Data dependence

- Example:

```
lw   $1, 200($2)
add  $3, $4, $1
```

- **add** can't carry out **ID** until **$1** has the updated value from the 5th stage of **lw**

Control dependence

- Example:

```
bne  $1, $2, target
add  $3, $4, $5
```

- **add** can't carry out **IF** until **bne** has completed the comparison in the 3rd stage of **bne**

Pipeline Hazards

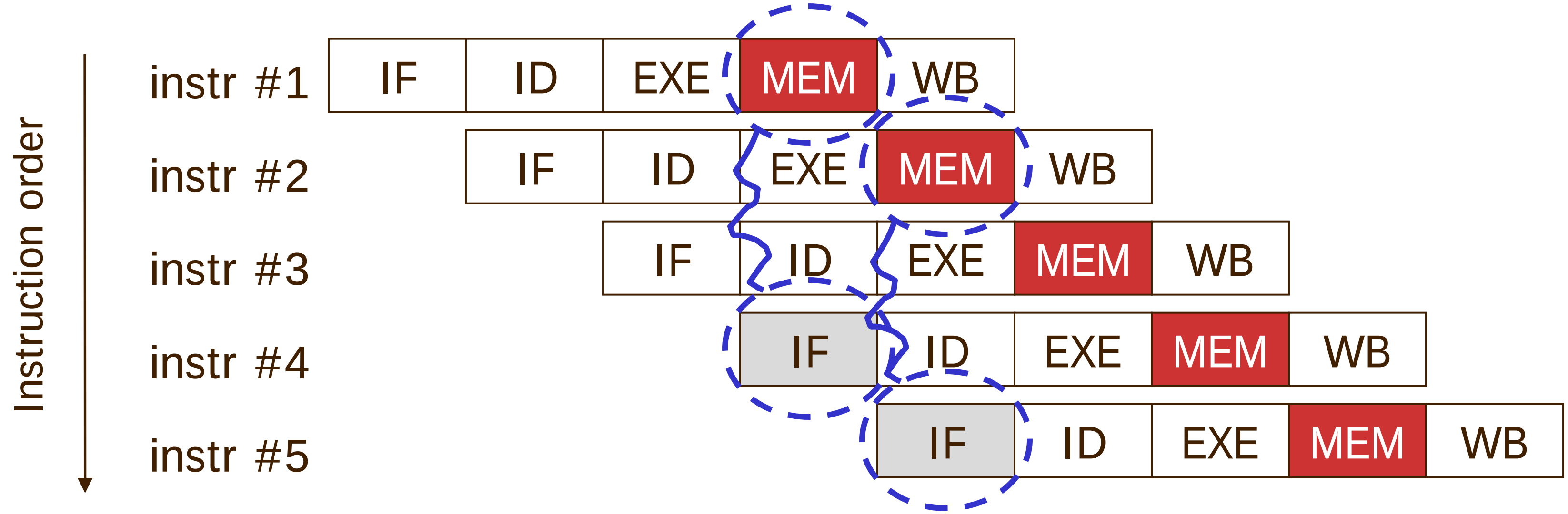
- **Hazards** are situations in pipelining when the next instruction cannot be executed in the following clock cycle (or pipeline stall)
- Hazards reduce the performance of pipelining

Three types of pipelined hazards

- **Structural hazards:** hardware cannot support the combination of instructions to execute in the same clock cycle
- **Control hazards:** which instruction to execute next depends on the results of a previous instruction still in the pipeline
- **Data hazards:** an instruction depends on the results of a previous instruction still in the pipeline

Structural Hazards Example: Single Memory

- ❑ If instructions #1 and #2 are load operations ➔ structural hazard
- ➢ Instruction fetch (#4, #5) and data load (#1, #2) need memory access

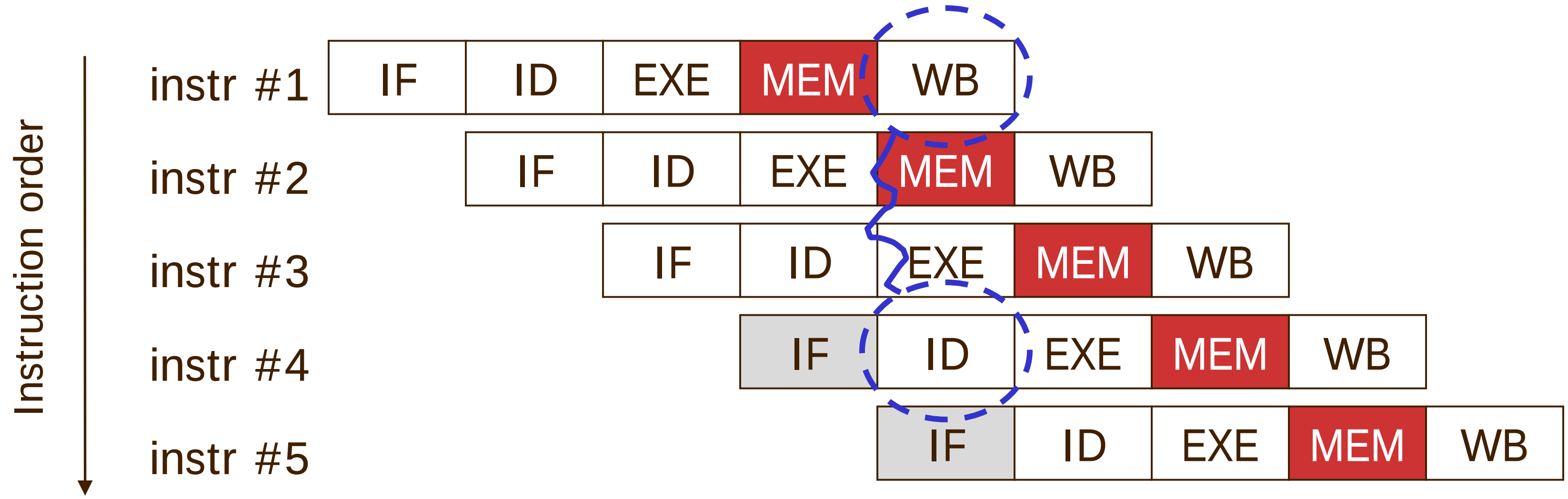


Read same memory twice in same clock cycle
(assumption: memory can only service one at a time)

- ❑ Solution?
- ➢ Add memory ports to allow parallel accesses to independent addresses
- ➢ Use caching with separate Level 1 Data cache and Instruction cache

Structural Hazards Example: Registers

- If instr. #1 is a store operation ➔ structural hazard with instr. #4
  - As instr. #1 wants to write while instr. #4 wants to read the register file



Can't read and write to registers simultaneously

- Solution?

Structural Hazards Example: Registers (Cont.)

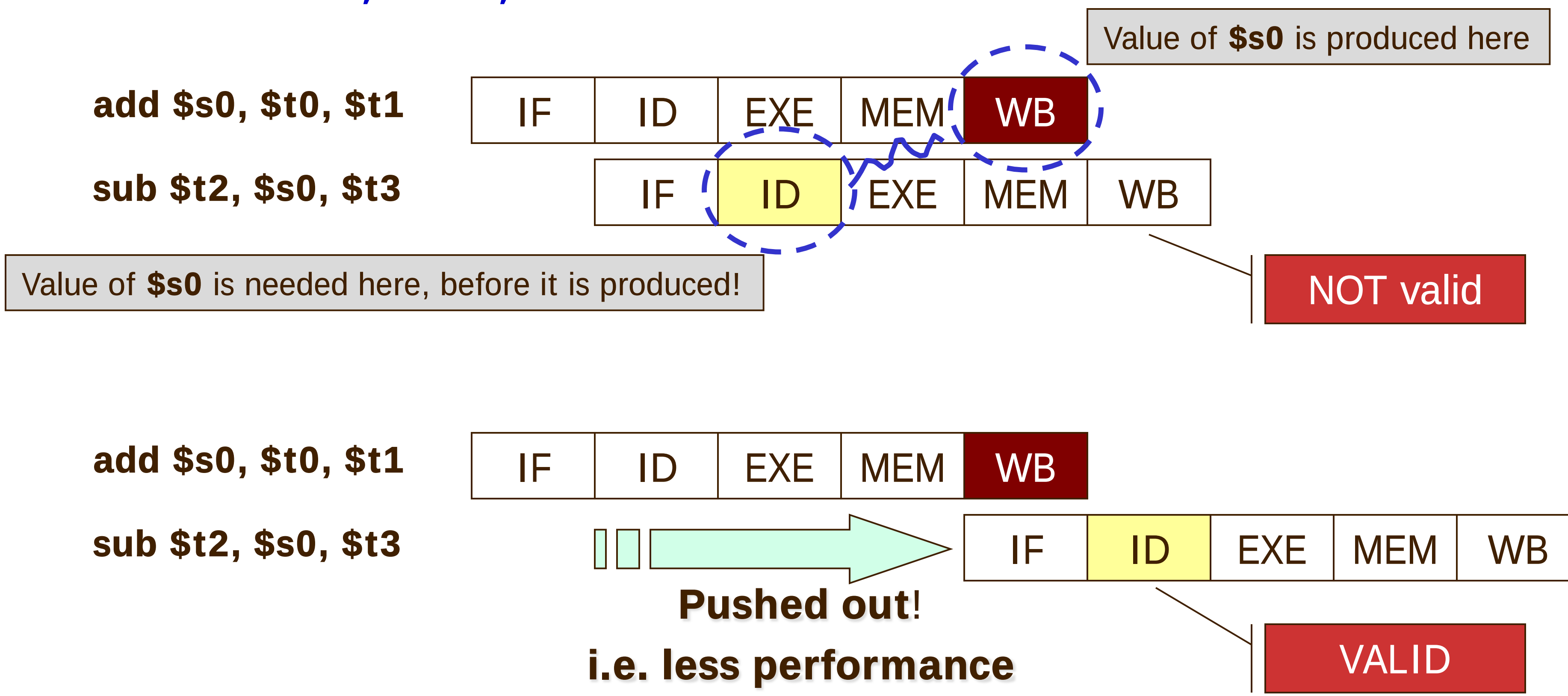
Fact: Register access is VERY fast;

- Takes less than half the time of ALU stage

- Solution:
  - always Write to registers during 1st half of each clock cycle
  - always Read from Registers during 2nd half of each clock cycle
  - Result: can perform write the Read during the same clock cycle

- **Data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline**
- e.g.

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```



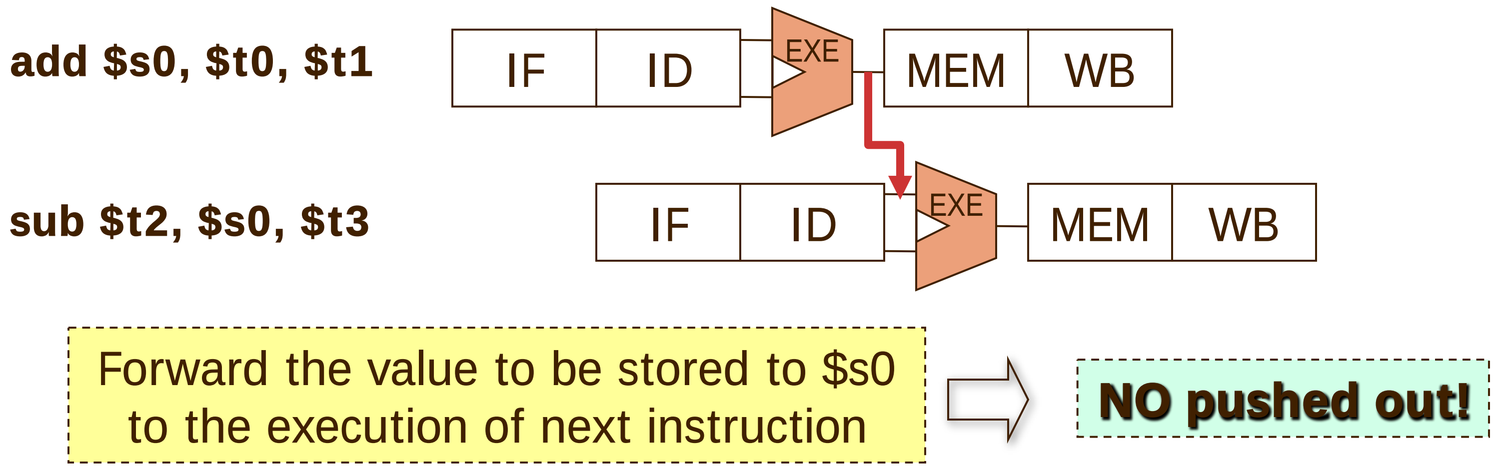
Data Hazards (2)

Observation:

- We don't need to wait for the instruction to complete before we try to resolve the data hazard

Solution:

- Add extra hardware to retrieve the missing item early from the internal resources, i.e. **forwarding** or **bypassing**

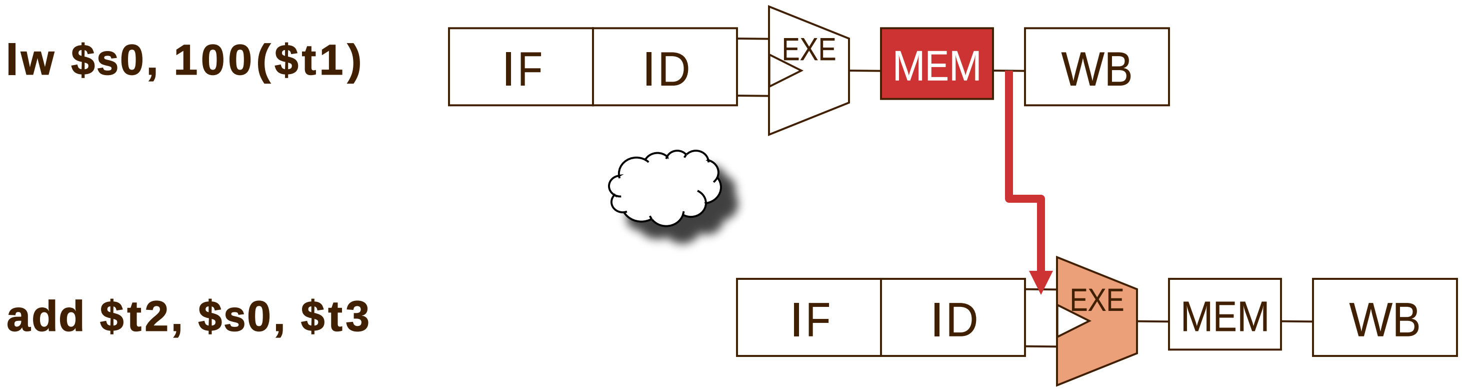


But, **pipeline stall** (nickname bubble) happens even with forwarding

- When a **R-format** instruction following a **load** tries to use the data
- e.g.

```
lw     $s0, 100($t0)
add    $t2, $s0, $t3
```

- Why still bubble?
  - Because the earliest time we can forward is from MEM, not EXE



We say **one** bubble is introduced, or processor has to stall for **one** cycle

Reordering Code to Avoid Pipeline Stalls

- Consider code segment in C below
- All variables are in memory and are addressable as offsets from **`$t0`**

```
A = B + E;
C = B + F;
```

Original version:

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

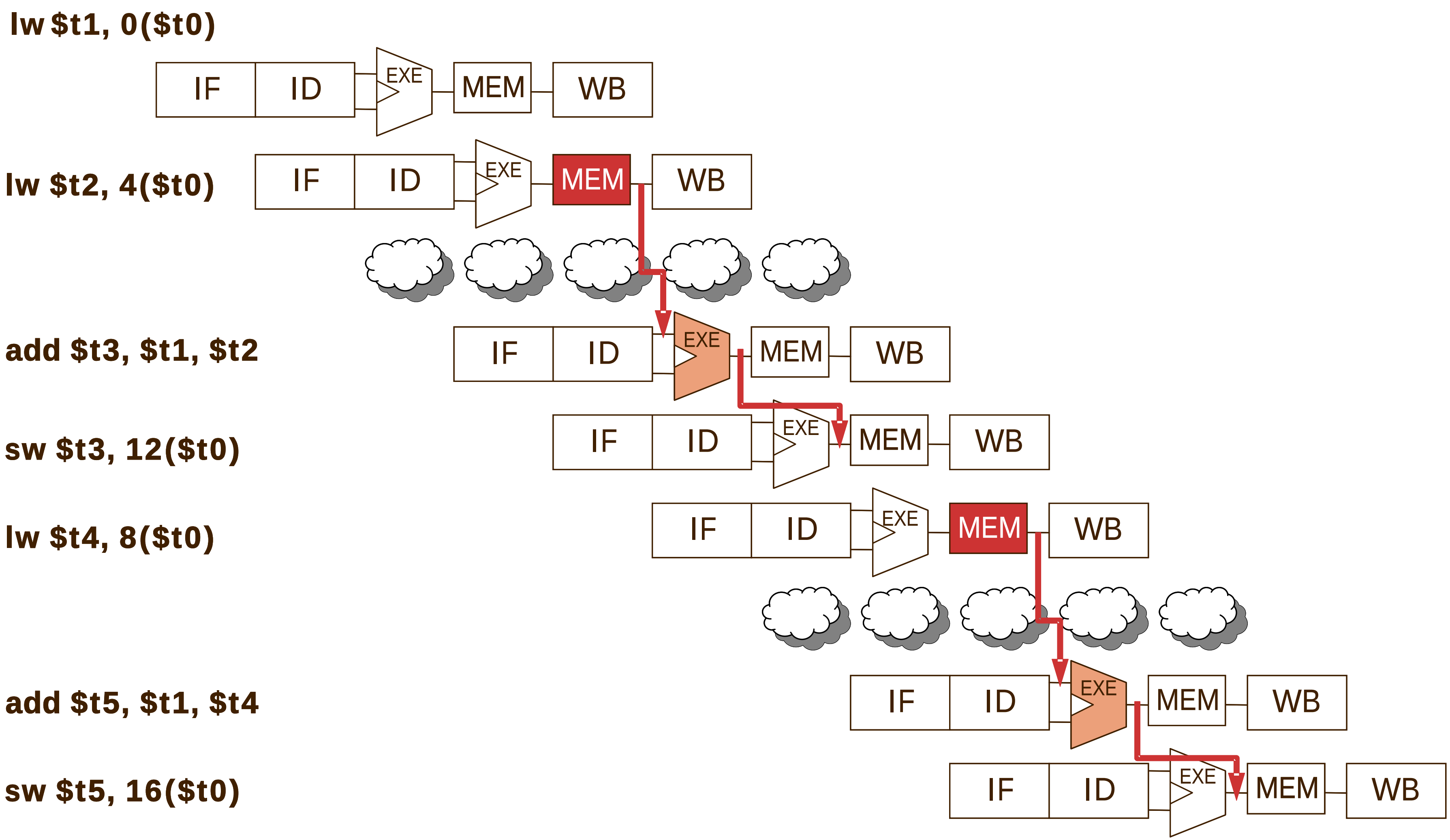
Both **`add`** instructions have a hazard on the immediately proceeding **`lw`** instruction

Reordered version:

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

Reordering the instructions by filling the bubbles with other useful but independent “work”

Sketching the Pipeline



Sketching the Pipeline After Code Reordering

lw $t1, 0($t0)

lw $t2, 4($t0)

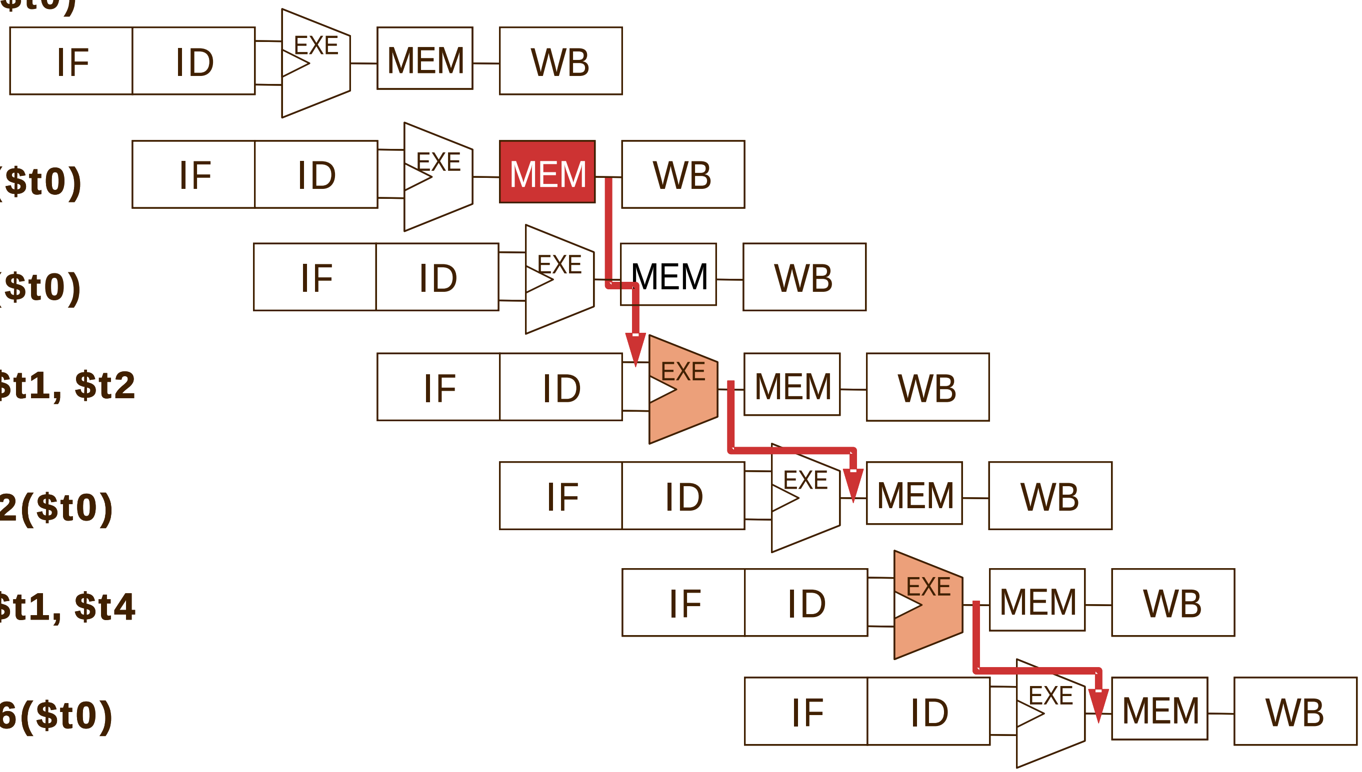
lw $t4, 8($t0)

add $t3, $t1, $t2

sw $t3, 12($t0)

add $t5, $t1, $t4

sw $t5, 16($t0)



- Execution time = cycles for the ideal case + # of bubbles
  = startup time + # of instructions + # of bubbles

Original version:

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

Reordered version:

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

- Execution time (original) = 4 + 7 + 2 = 13
- Execution time (reorder) = 4 + 7 = 11

- Assume **$s0** carry **A**, **$s1** carry **i**, **$s2** carry **n**

```
for (i=2; i<n; i++)
  A[i] = A[i-1] + C;    // C is a constant
```

Assembly version:

```
  addi  $t0, $s0, 0
LOOP:
  addi  $t0, $t0, 4
  lw    $t1, 0($t0)
  addi  $t1, $t1, C
  sw    $t1, 4($t0)
  addi  $s1, $s1, 1
  bne   $s1, $s2, LOOP
```

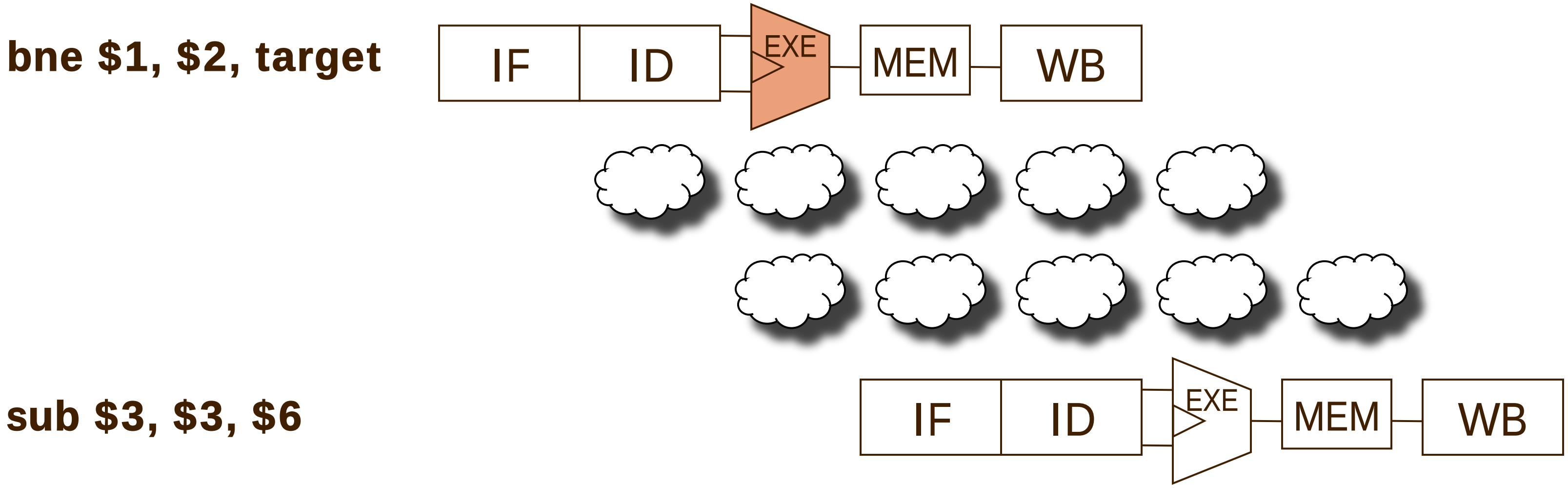
Assembly version:

```
  addi  $t0, $s0, 0
LOOP:
  addi  $t0, $t0, 4
  lw    $t1, 0($t0)
  addi  $s1, $s1, 1
  addi  $t1, $t1, C
  sw    $t1, 4($t0)
  bne   $s1, $s2, LOOP
```

Control Hazards

- **Control hazards arise from the need to make a decision based on the results of one instruction while others are executing**
- e.g.

```
          bne   $1, $2, target
          add   $3, $4, $5
target:   sub   $3, $3, $6
```

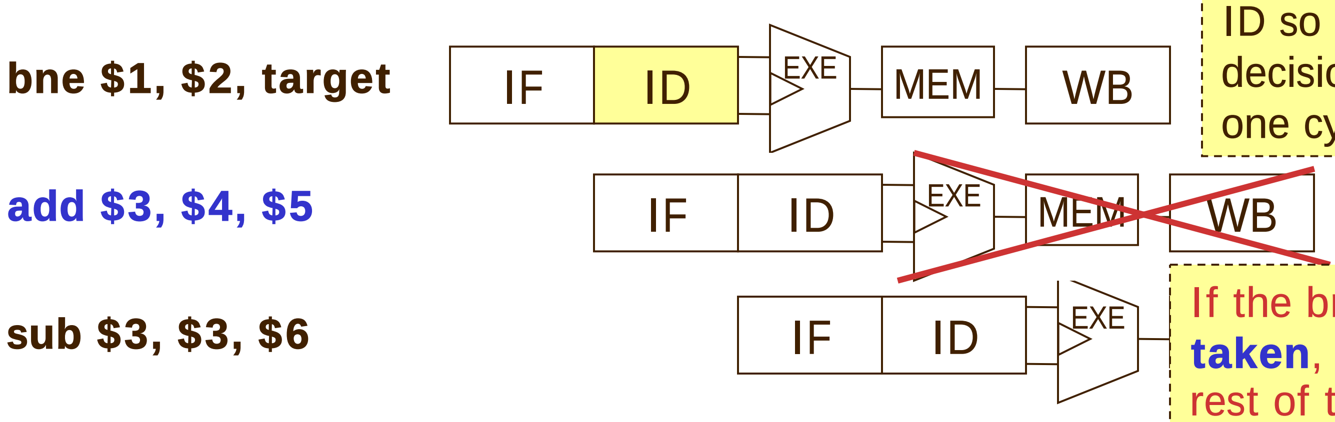


- Simple approach to reduce the stalling cycles from 2 to 1:
  - Dedicate a separate comparator in ID stage for branch comparison

```
        bne   $1, $2, target
        add   $3, $4, $5
target: sub   $3, $3, $6
```

- Rather than stall, fill the bubble with work from **add**
- Basically, predict branch not taken. Why it works?

This example has a dedicated comparator in ID so that branch decision can be made one cycle early



If the branch is to be **taken**, stop executing the rest of the stages for **add**

Reason:

- If the branch is really not taken, **add** already started, no pipeline stall!
- So, **reduce the frequency of stalling** due to branches

Summary

- Pipelining improves processor performance by overlapping instructions
  - Pipelining principles
  - The number of pipe stages decides the potential speedup
  - But, there is a limit in the number of pipe stage we can have

- Pipeline hazards
  - Structural hazard
  - Data hazard
  - Control hazard

- Techniques to reduce the negative impact of pipeline hazards
  - Forwarding
  - Reorder the code
  - Make branch decision early
  - Fill bubble with potential work