Pipelined Datapath and Control

Chapter 6

DATAPATH AND CONTROL

We started with the **single-cycle implementation**, in which a single instruction is executed over a single cycle. In this scheme, a cycle's clock period must be defined to be as long as necessary to execute the longest instruction. But this results in a lot of waste – both in terms of time and space since we need multiple of the same kinds of datapath elements to execute a single instruction.

Then, we looked at **multi-cycle implementation**. In this scheme, instructions are broken up over general steps and each step is performed over a single clock cycle. As a result, we have a smaller clock cycle and we are able to reuse datapath elements in different cycles. However, we are still limited to executing one instruction at a time.

PIPELINING

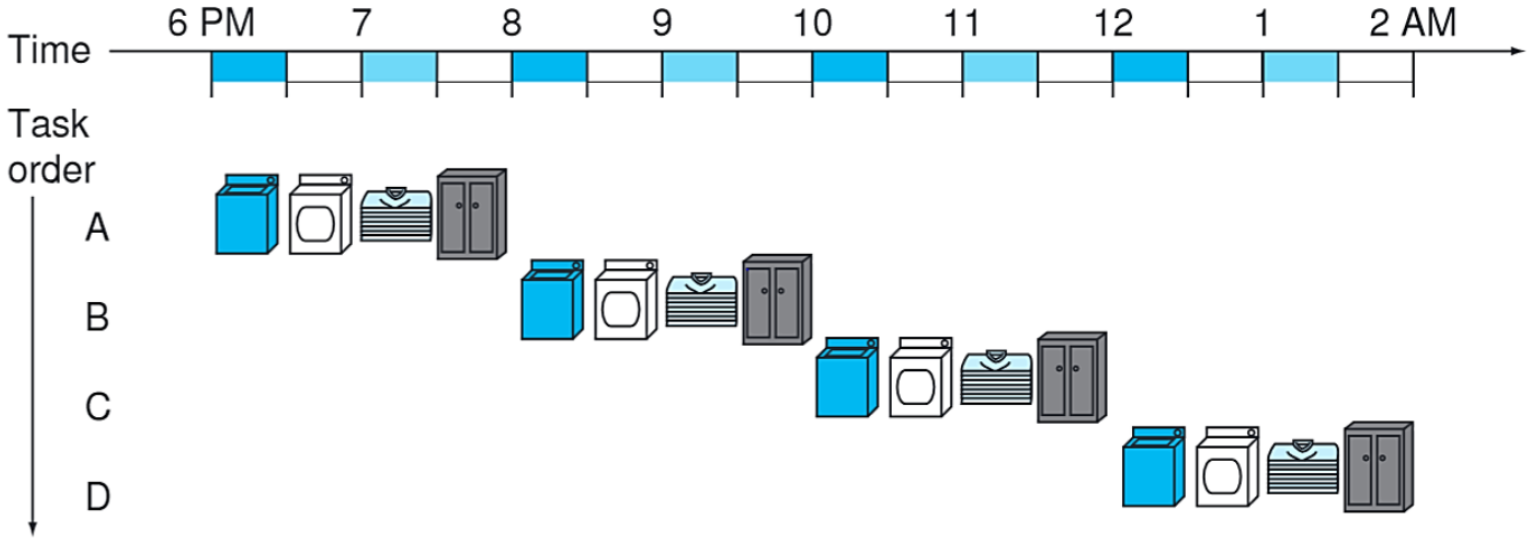
Now we're going to build upon what we know and look at pipelining.

Pipelining involves not only executing an instruction over multiple cycles, but also executing multiple instructions per cycle. In other words, we're going to overlap instructions.

There is a classic, intuitive analogy that will help us understand pipelining. Let's do some laundry!

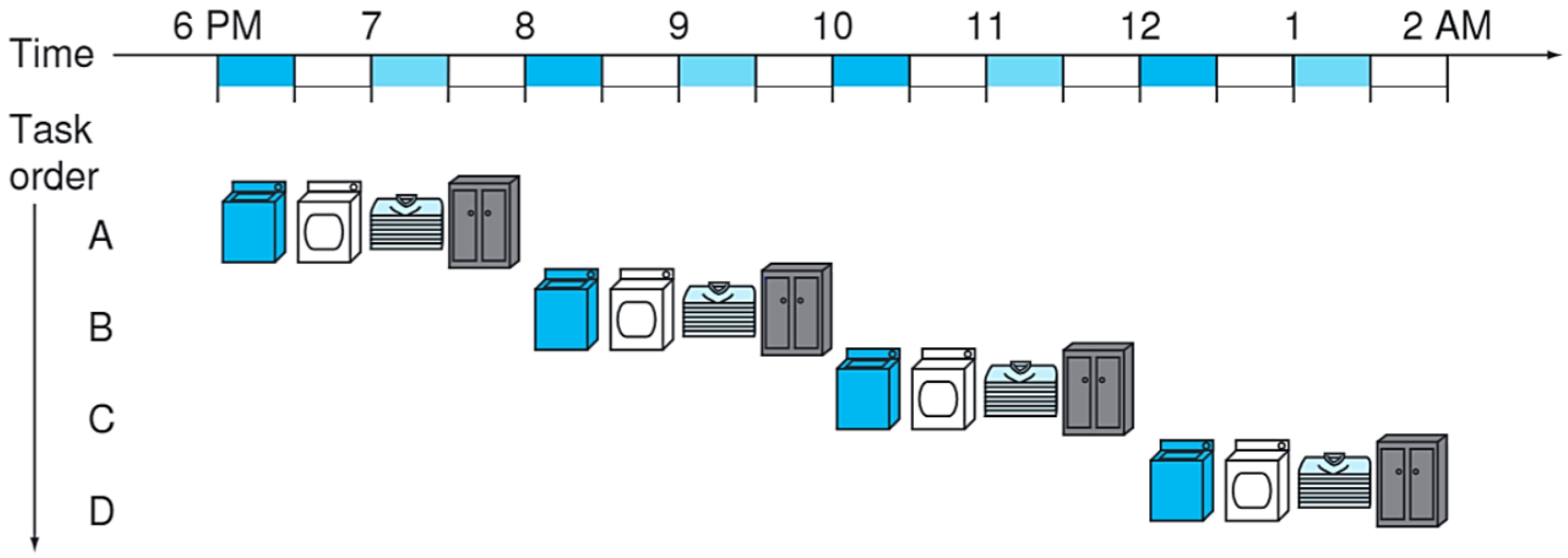
LAUNDRY ANALOGY

Let's say we have a couple of loads of laundry to do. Each load of laundry involves the following steps: washing, drying, folding, and putting away.



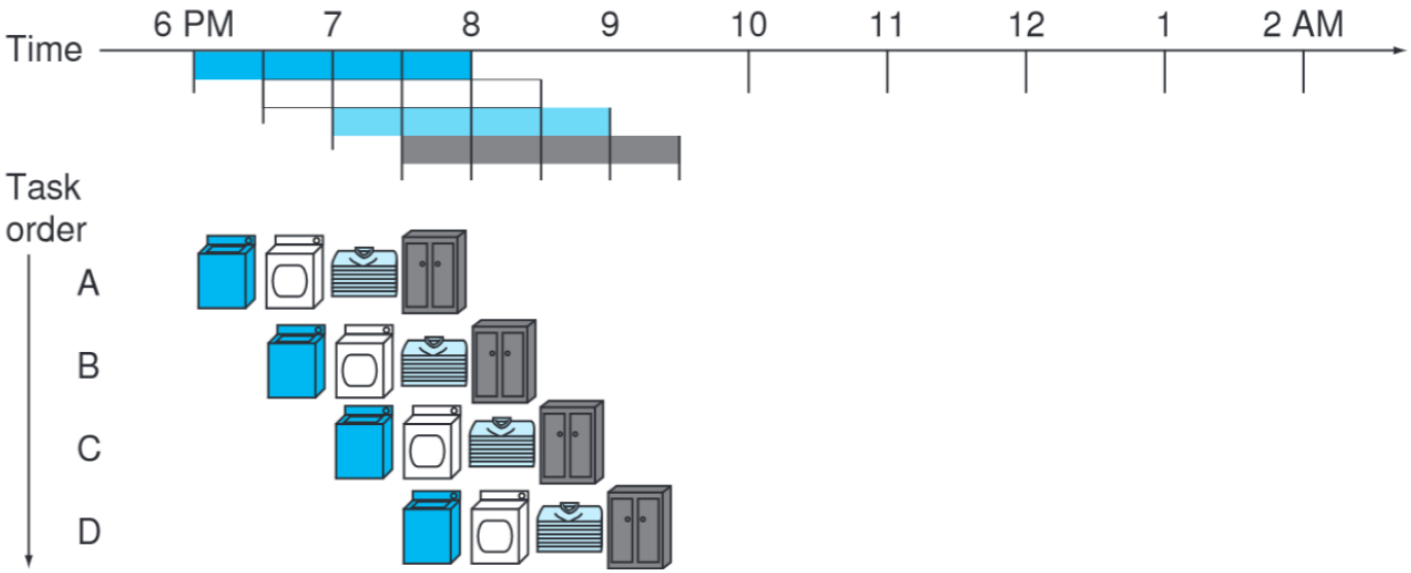
LAUNDRY ANALOGY

We can perform one step every thirty minutes and start the next load after the previous load has finished. This is similar to multi-cycle implementation.



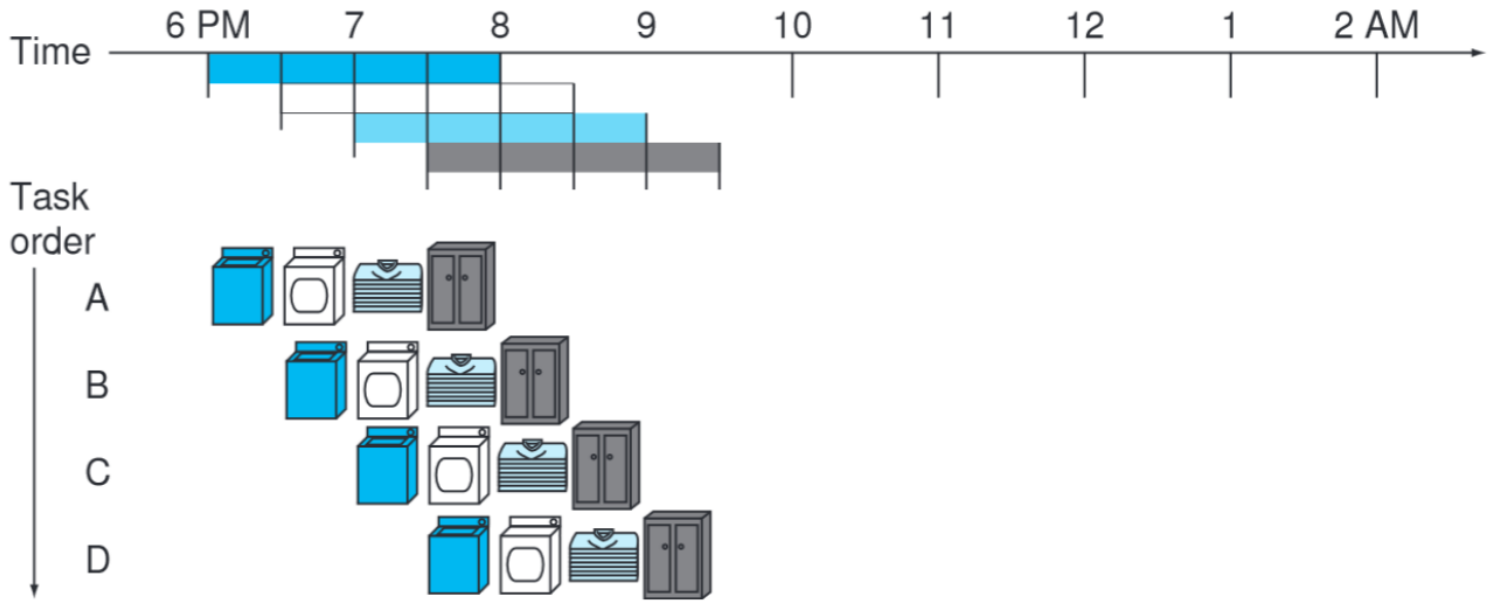
LAUNDRY ANALOGY

There's no reason why we shouldn't start the next load right after the first load is out of the washer. The washer is available now, after all. This is analogous to pipelining.



LAUNDRY ANALOGY

Notice that now we are using parallelism to finish four loads in only 3.5 hours, as opposed to the multi-cycle method which has us doing laundry until 2 AM.



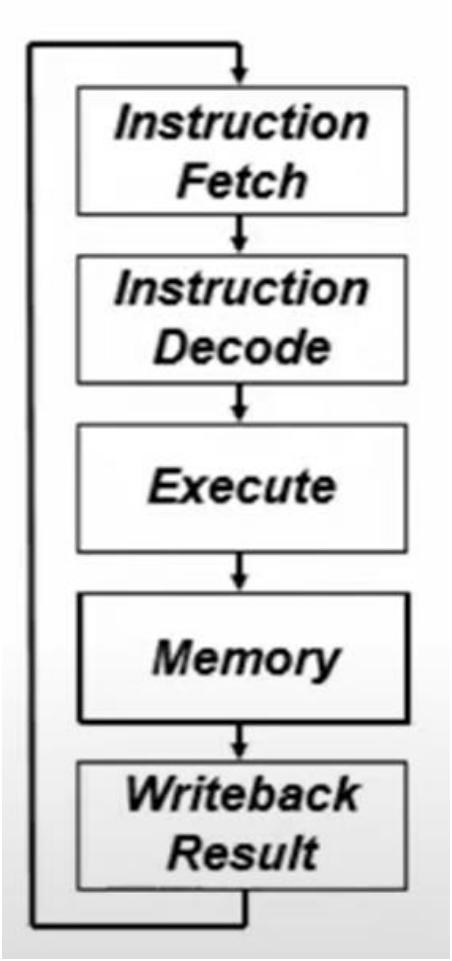
PIPELINING

- Pipelining essentially involves creating an assembly line for instruction execution.
- Each step in the pipeline is called a ***stage***.
- Multiple instructions can be processed in parallel as long as they are at different stages.
- Pipelining is really just like multi-cycle implementation, except we start the next instruction as soon as we can. Pipelining therefore **increases the throughput**, but **not the instruction latency**, when compared to multi-cycle.
- The speedup is ideally the same as the number of stages in the pipeline, as long as the number of instructions is much larger than the number of stages.

PIPELINING STAGES

We already know roughly what the stages are:

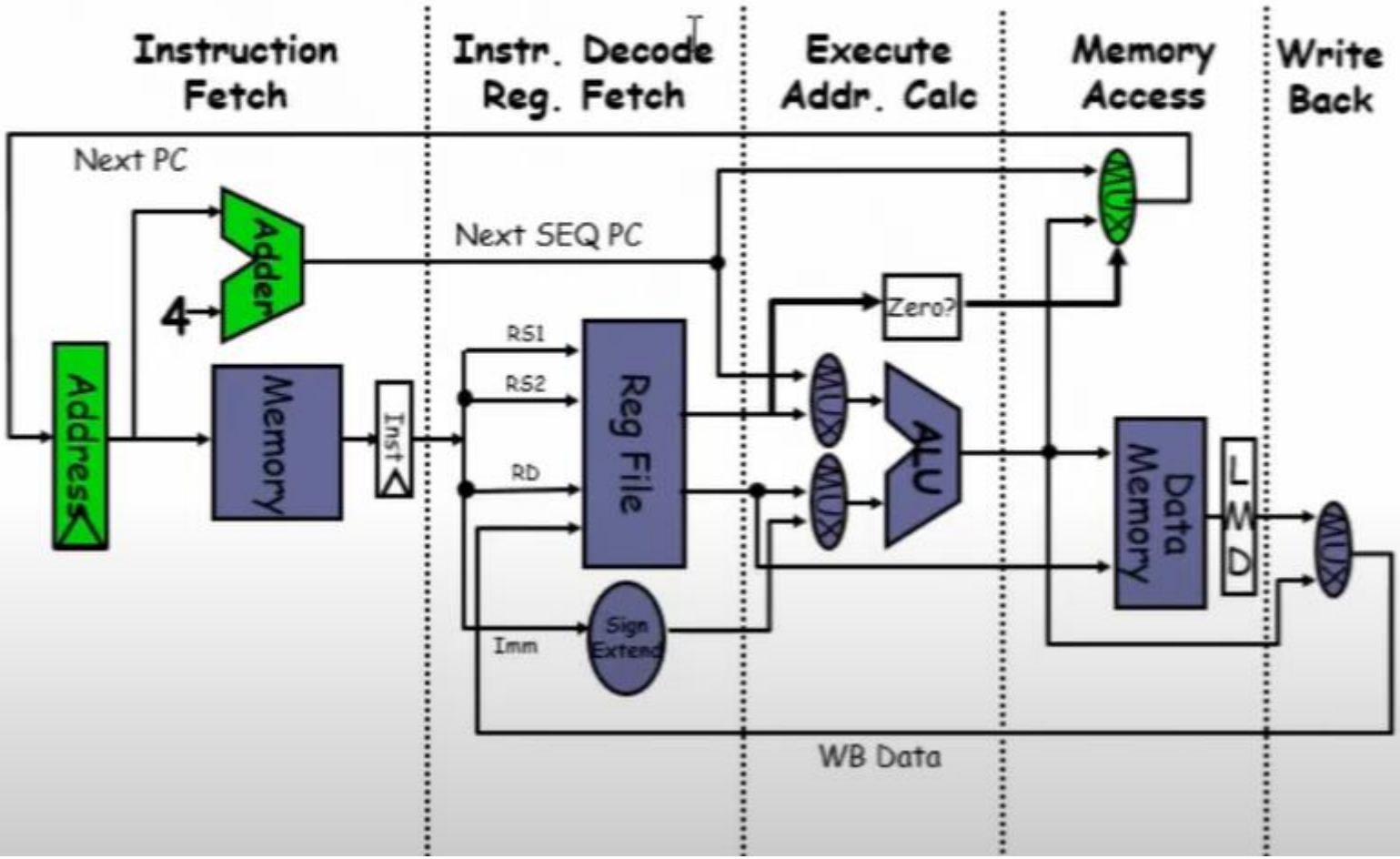
- **IF** – Instruction Fetch.
- **ID** – Instruction Decode.
- **EX** – Execution or Address Calculation.
- **Mem** – Data Memory Access.
- **WB** – Write Back.



PIPELINING STAGES

- IF stage: fetches the instruction from the instruction cache and increments the PC.
- ID stage: decodes the instruction, reads source registers from register file, sign-extends the immediate value, calculates the branch target address and checks if the branch should be taken.
- EX stage: calculates addresses for accessing memory, performs arithmetic/logical operations on either two register values or a register and an immediate.
- MEM stage: load a value from or store a value into the data cache.
- WB stage: update the register file with the result of an operation or a load.

5 Steps of MIPS Datapath



1st and 2nd Instruction cycles

- **Instruction fetch (IF)**

  IR ← Mem[PC];

  NPC ← PC + 4

- **Instruction decode & register fetch (ID)**

  A ← Regs[$IR_{11..15}$];

  B ← Regs[$IR_{16..20}$];

  Imm ← ((IR_{15})^{16} # # $IR_{0..15}$)

3rd Instruction cycle

- Execution & effective address (EX)
  - Memory reference
    - ALUOutput ← A + Imm
  - Register - Register ALU instruction
    - ALUOutput ← A *func* B
  - Register - Immediate ALU instruction
    - ALUOutput ← A *op* Imm
  - Branch
    - ALUOutput ← NPC + Imm; Cond ← (A *op* B)

4th Instruction cycle

- Memory access & branch completion (MEM)
  - Memory reference
    - PC ← NPC
    - LMD ← Mem[ALUOutput] (load)
    - Mem[ALUOutput] ← B (store)
  - Branch
    - if (cond) PC ← ALUOutput; else PC ← NPC

5th Instruction cycle

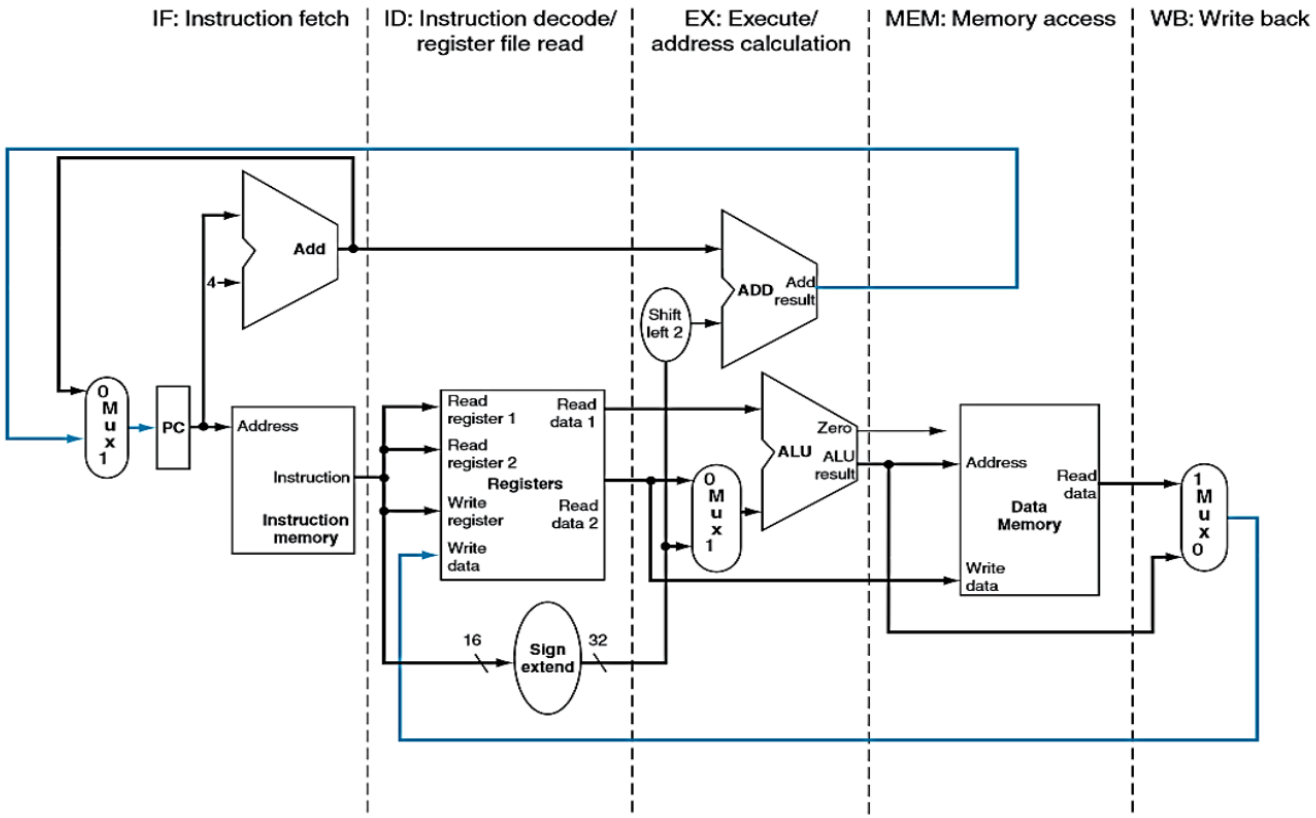
- Write-back (WB)
  - Register - register ALU instruction
    - $Regs[IR_{21..25}] \leftarrow ALUOutput$
  - Register - immediate ALU instruction
    - $Regs[IR_{16..20}] \leftarrow ALUOutput$
  - Load instruction
    - $Regs[IR_{16..20}] \leftarrow LMD$

PIPELINING STAGES

We start by taking the single-cycle datapath and dividing it into 5 stages.

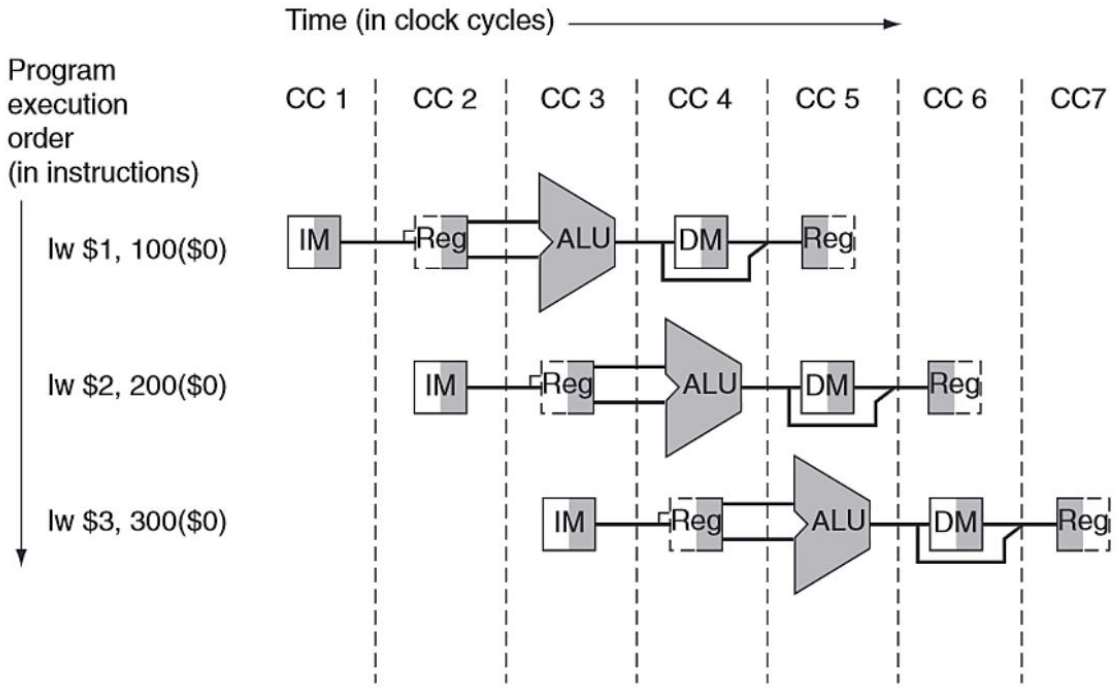
A 5-stage pipeline allows 5 instructions to be executing at once, as long as they are in different stages.

All of the data moves from left-to-right with two exceptions: writing to the register file and writing to the PC.



PIPELINING STAGES

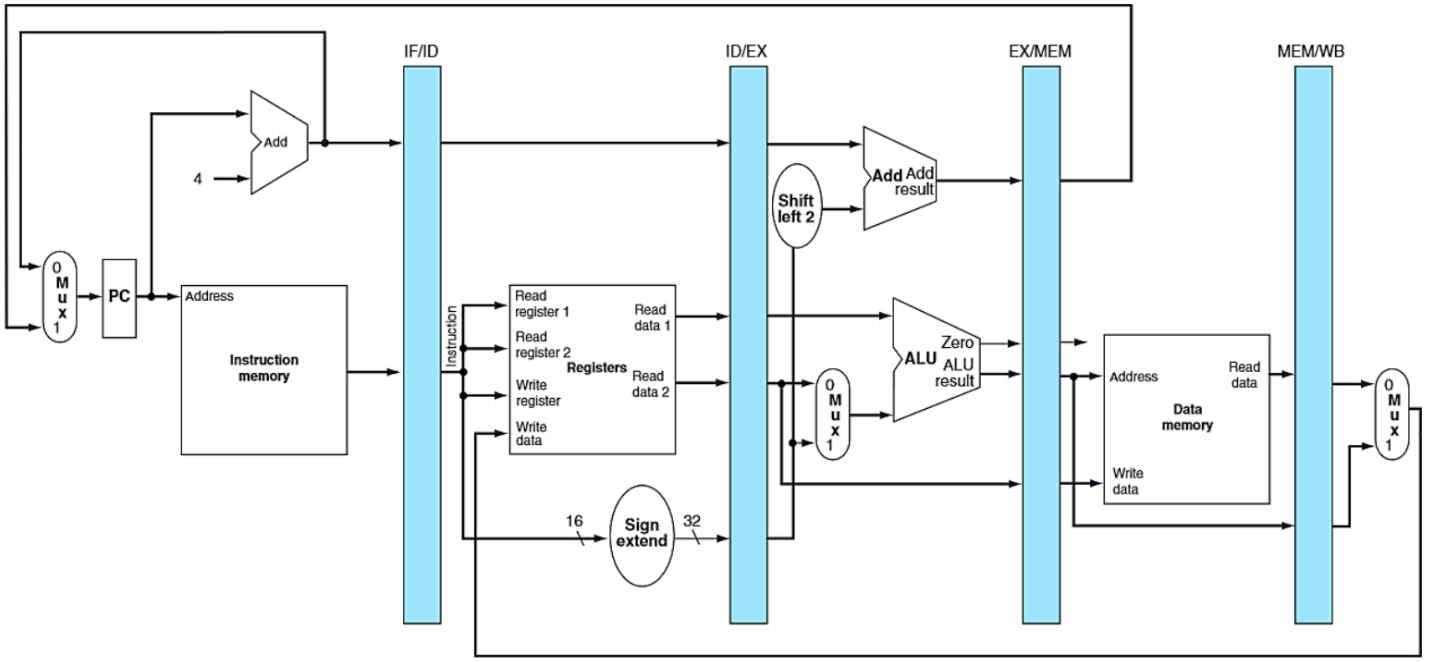
Note that in every cycle, each element is only used by at most one instruction.



PIPELINING STAGES

Even though the datapath is similar to single-cycle, we need to note that we are still executing across multiple cycles.

Therefore, we add **pipeline registers to store data across cycles.**



PIPELINING SPEEDUP

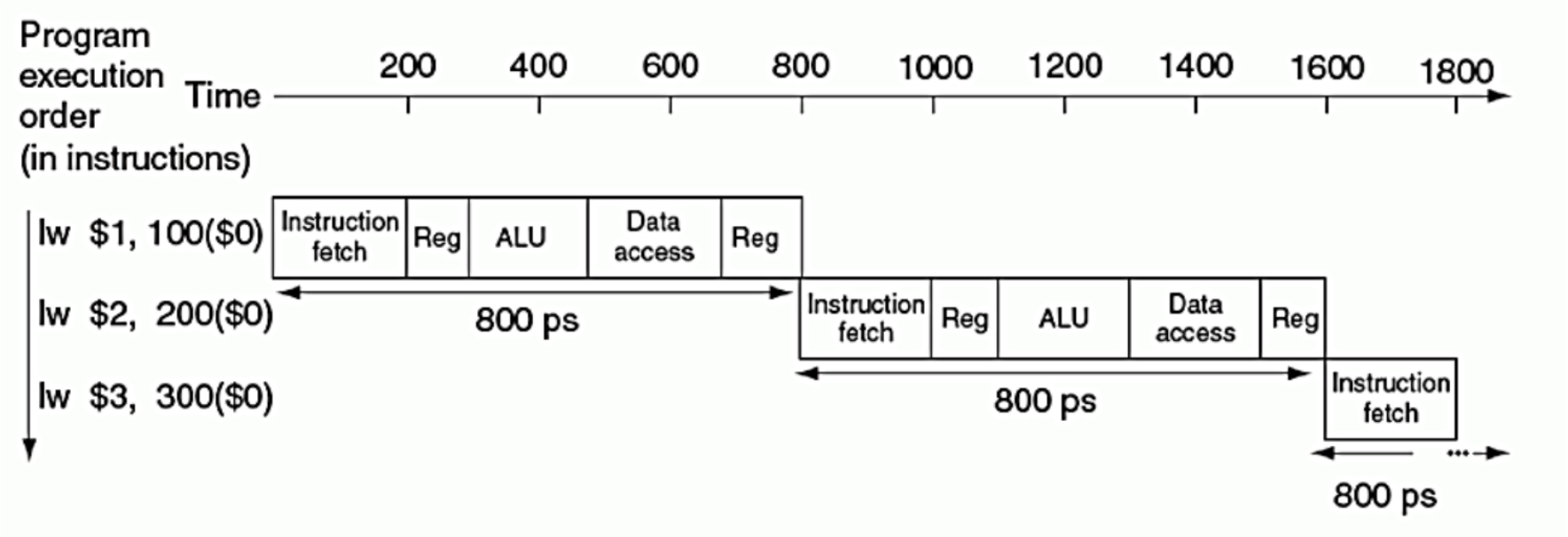
Let's look at an example. Say we want to perform three load word instructions in a row. The operation times for the major functional units are 200 ps for memory access, 200 ps for ALU operations, 100 ps for register file read/writes.

Given

Instruction	IF	ID	EX	MEM	WB	Total
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps

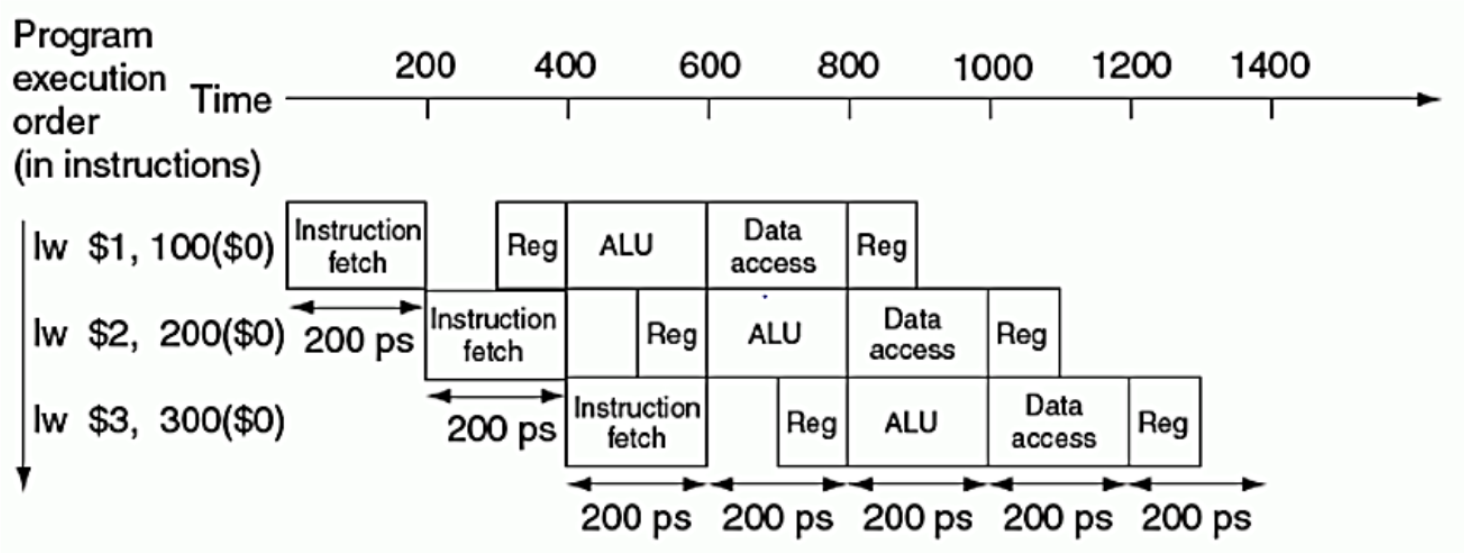
PIPELINING SPEEDUP

In the single-cycle implementation, lw takes 1 cycle totaling 800 ps. We cannot start the next instruction until the last cycle ends so the time between the first and fourth instruction is 2400 ps.



PIPELINING SPEEDUP

In the pipelining implementation, lw takes 5 cycles totaling 1000 ps. This is because **every cycle needs to be as long as the longest cycle**, which is 200 ps. We start the next instruction as soon as possible. The time between the first and fourth instruction is 600 ps.



PIPELINING SPEEDUP

It is important to take note of the fact that the pipelined implementation has a **greater** latency per instruction.

However, this is ok because the advantage we gain with pipelining is increased throughput, which is more important because real programs execute billions of instructions.

PIPELINING SPEEDUP

As stated before, the ideal speedup is equivalent to the number of stages in the pipeline. We can express this in a concise formula. Let TBI be Time Between Instructions.

$$TBI_{pipelined} = \frac{TBI_{non-pipelined}}{Number\ of\ Stages}$$

There are several reasons why we may not obtain ideal speedup:

- **Stages are not perfectly balanced** (leading to an increase in latency).
- **Storing and retrieving information between stages has some overhead.**
- **Hazards.**

Speedup = Time Without pipelining / Time with pipelining

PIPELINING SPEEDUP

As you may have already noticed, our `lw` example does *not* exhibit 5-fold speedup even though there are 5 stages. We have an overall completion time of 2400 ps for single-cycle and an overall completion time of 1400 ps for pipelining. This is merely a 1.7 times speedup.

Imagine instead that we are executing 1,000,000 lw instructions. For single-cycle, this means 800,000,000 ps since each instruction requires 800 ps. But for pipelining, this only means 200,000,800 ps since each additional instruction only adds 200 ps.

$$\frac{8{,}000{,}000\ ps}{2{,}000{,}800\ ps} \cong \frac{8}{2} = 4$$

When we increase the number of instructions, we get roughly 4 times speedup.

MIPS AND PIPELINING

MIPS was designed with pipelining in mind.

- All MIPS instructions are **the same length** – 32 bits.
  - This makes the IF phase universal and simple to implement.
- There are only **three instruction formats,** and source register fields are always in the same place.
  - This means we can read the register file in the ID phase before we even know what the instruction is.
- The only memory operations occur in **load and store** instructions.
  - We can dedicate the ALU to computing addresses in these stages.
- Memory accesses must be **aligned.**
  - No need to worry about multiple data memory accesses per instruction.

PIPELINE HAZARDS

- **Dependencies**: relationships between instructions that prevent one instruction from being moved past another.
- **Hazards**: situation where next instruction cannot execute in the following cycle.
  - Three types: **structural, data,** and **control.**
- **Stalls**: technique of stalling an instruction until a pipeline hazard no longer exists.

PIPELINE HAZARDS

A **structural hazard** occurs when a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the particular combination of instructions that are set to execute in the given clock cycle. Try to access Same resource at a time

In the laundry analogy, a structural hazard might occur if we used a combo washer/dryer instead of separate washer and dryer machines.

Imagine the following instructions are executed over 8 clock cycles. Notice how in cycle 4, we have a MEM and IF phase executing. If there is only one single memory unit, we will have a structural hazard.

cycle	1	2	3	4	5	6	7	8
Inst 1	IF	ID	EX	MEM	WB			
Inst 2		IF	ID	EX	MEM	WB		
Inst 3			IF	ID	EX	MEM	WB	
Inst 4				IF	ID	EX	MEM	WB

PIPELINE HAZARDS

A **data hazard** occurs when a planned instruction cannot execute in the proper clock cycle because the data that is needed is not yet available.

Consider the following instructions:

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

cycle	1	2	3	4	5	6
add	IF	ID	EX	MEM	WB	
sub		IF	ID	EX	MEM	WB

PIPELINE HAZARDS

The worst case solution involves stalling the sub instruction for 3 cycles. While this resolves our dependency issue, it's not ideal. As a note, it is part of the compiler's job to identify dependencies like this and reorder instructions if possible. But we cannot rely on that solution either.

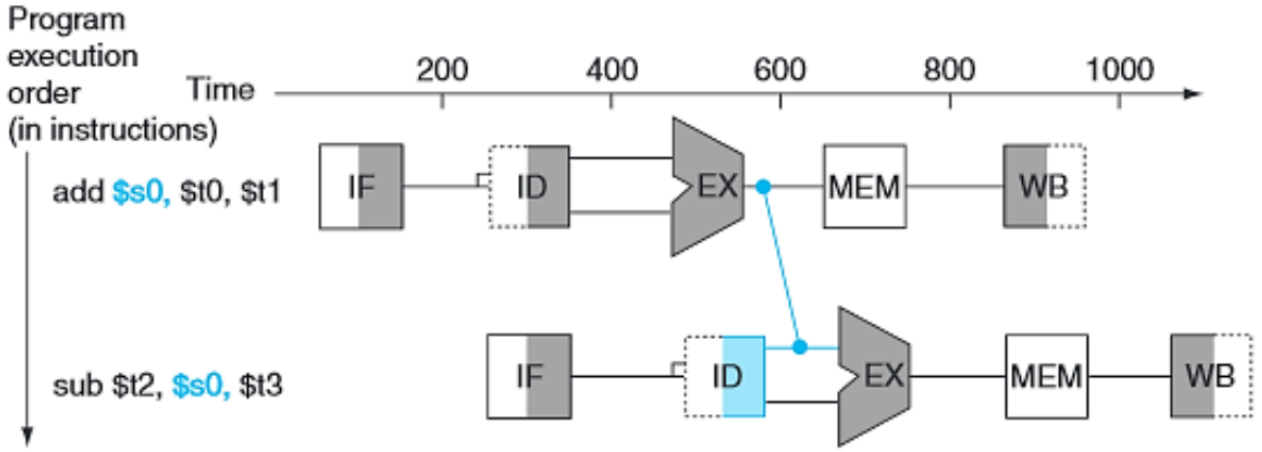
```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

cycle	1	2	3	4	5	6	7	8	9
add	IF	ID	EX	MEM	WB				
sub					IF	ID	EX	MEM	WB

PIPELINE HAZARDS

Another solution for the problem is known as **forwarding** (or **bypassing**). This method involves retrieving the data from internal buffers rather than waiting for the data to be updated in the register file or data memory.

Because the result of the add operation is available at the end of the EX stage, we can grab its value for use in the subsequent instruction.

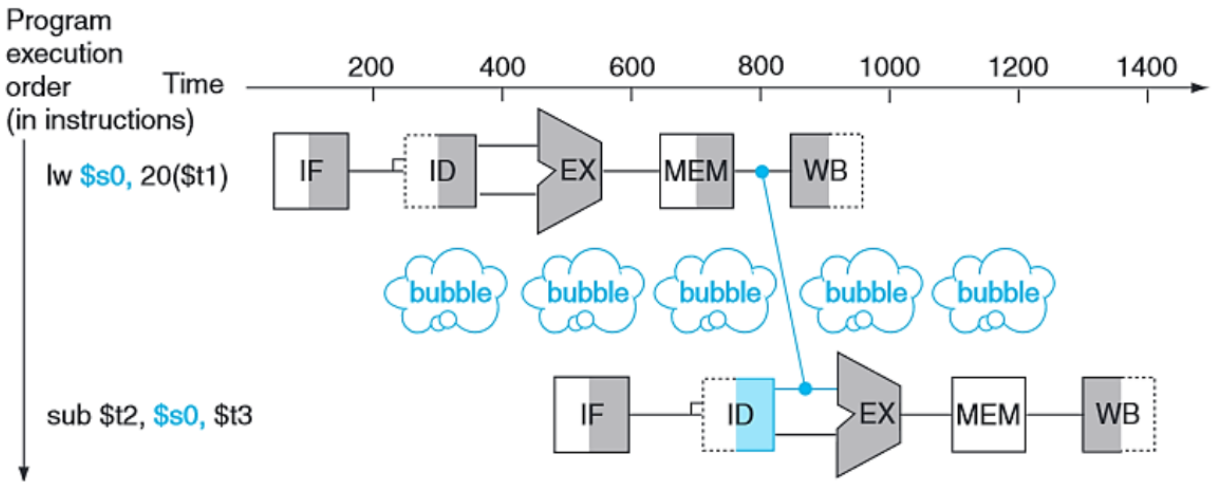


PIPELINE HAZARDS

We cannot always prevent all stalls with forwarding. Consider the following instructions.

```
lw    $s0, 20($t1)
sub   $t2, $s0, $t3
```

Even if we use forwarding, the new contents of $s0 are only available after load word's MEM stage. So we'll have to stall the sub instruction one cycle.



PIPELINE HAZARDS

Consider the following C code:

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

Here is the equivalent MIPS code assuming all variables are in memory and are addressable as offsets from $t0:

```
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)
```

Find the hazards and reorder the instructions to avoid any stalls.

PIPELINE HAZARDS

Find the hazards and reorder the instructions to avoid any stalls.

```
lw   $t1, 0($t0)  # $t1 written in stage 5 (cycle 5).
lw   $t2, 4($t0)  # $t2 written in stage 5 (cycle 6).
add  $t3, $t1,$t2 # $t1, $t2 read in stage 2 (cycle 4).
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

PIPELINE HAZARDS

Find the hazards and reorder the instructions to avoid any stalls.

```
lw   $t1, 0($t0)  # $t1 written in stage 5 (cycle 5).
lw   $t2, 4($t0)  # $t2 written in stage 5 (cycle 6).
add  $t3, $t1,$t2 # $t1, $t2 read in stage 2 (cycle 4).
                  # $t3 written in stage 5 (cycle 7).
sw   $t3, 12($t0) # $t3 read in stage 2 (cycle 5).
lw   $t4, 8($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

PIPELINE HAZARDS

Find the hazards and reorder the instructions to avoid any stalls.

```
lw   $t1, 0($t0)  # $t1 written in stage 5 (cycle 5).
lw   $t2, 4($t0)  # $t2 written in stage 5 (cycle 6).
add  $t3, $t1,$t2 # $t1, $t2 read in stage 2 (cycle 4).
                  # $t3 written in stage 5 (cycle 7).
sw   $t3, 12($t0) # $t3 read in stage 2 (cycle 5).
lw   $t4, 8($t0)  # $t4 written in stage 5 (cycle 9).
add  $t5, $t1,$t4 # $t4 read in stage 2 (cycle 7).
sw   $t5, 16($t0)
```

PIPELINE HAZARDS

Find the hazards and reorder the instructions to avoid any stalls.

```
lw   $t1, 0($t0)  # $t1 written in stage 5 (cycle 5).
lw   $t2, 4($t0)  # $t2 written in stage 5 (cycle 6).
add  $t3, $t1,$t2 # $t1, $t2 read in stage 2 (cycle 4).
                  # $t3 written in stage 5 (cycle 7).
sw   $t3, 12($t0) # $t3 read in stage 2 (cycle 5).
lw   $t4, 8($t0)  # $t4 written in stage 5 (cycle 9).
add  $t5, $t1,$t4 # $t4 read in stage 2 (cycle 7).
                  # $t5 written in stage 5 (cycle 10).
sw   $t5, 16($t0) # $t5 read in stage 2 (cycle 8).
```

PIPELINE HAZARDS

If we are using a pipelined processor with forwarding, we have the following stages executing in each cycle:

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
lw $t4,8($t0)						IF	ID	EX	MEM	WB			
add $t5,$t1,$t4								IF	ID	EX	MEM	WB	
sw $t5,16($t0)									IF	ID	EX	MEM	WB

PIPELINE HAZARDS

We can move the third load word instruction up since it has no dependencies from previous instructions.

```
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)
```

PIPELINE HAZARDS

The reordering allows us to execute the program in two fewer cycles than before.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
lw $t4,8($t0)			IF	ID	EX	MEM	WB						
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
add $t5,$t1,$t4						IF	ID	EX	MEM	WB			
sw $t5,16($t0)							IF	ID	EX	MEM	WB		

PIPELINE HAZARDS

The third type of hazard, a **control hazard** (or **branch hazard**), occurs when the flow of instruction addresses is not known at the time that the next instruction must be loaded. Let's say we have the following instructions.

```
beq   $t0, $t1, L1
sub   $t2, $s0, $t3
```

We have a problem: we do not know what the next instruction should be until the end of the third cycle. But we're automatically fetching the next instruction in the second cycle. We will run into a similar problem with jumps as well.

cycle	**1**	**2**	**3**	**4**	**5**	**6**
beq	IF	ID	EX	MEM	WB	
sub		IF	ID	EX	MEM	WB

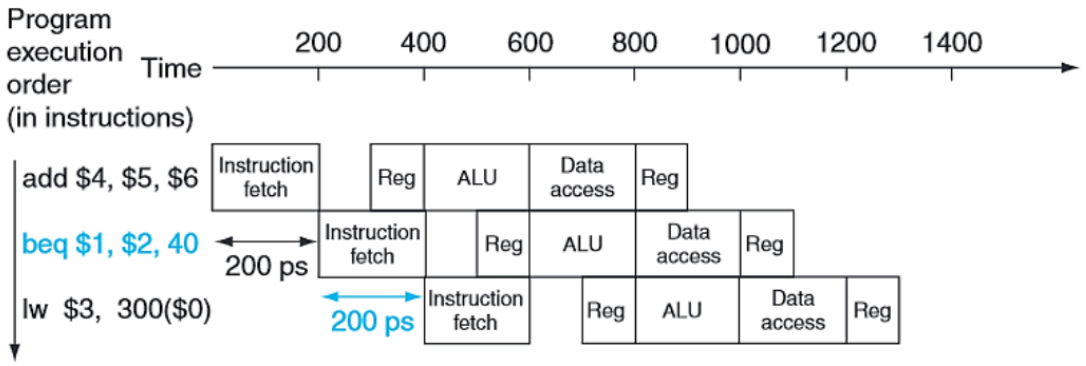
PIPELINE HAZARDS

- Problem: The processor does not know soon enough
  - whether or not a conditional branch should be taken.
  - the target address of a transfer-of-control instruction.
- Solutions:
  - **Stall** until the necessary information becomes available.
  - **Predict** the outcome and act accordingly.

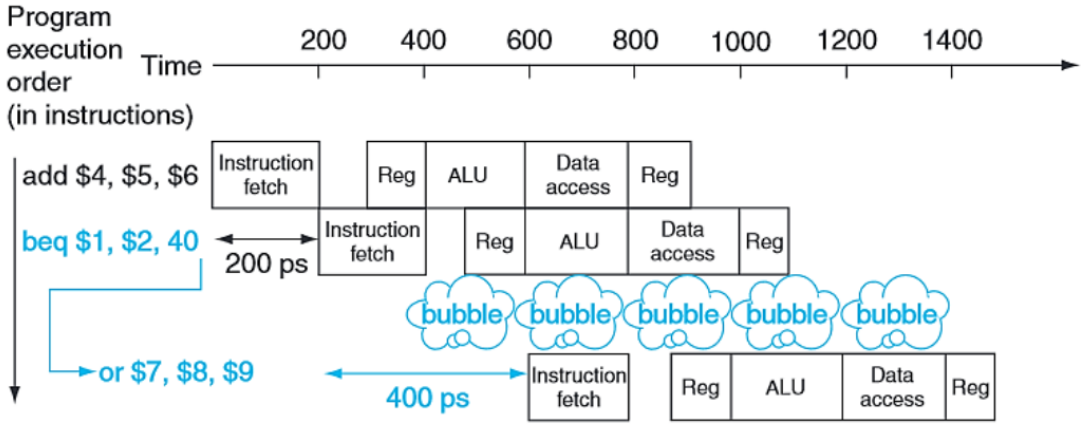
If we stall until the branch target is known, we will always incur a penalty for stalling. However, if we predict that the branch is not taken and act accordingly, we will only incur a penalty when the branch actually is taken.

PIPELINE HAZARDS

When predicting that a branch is not taken, we proceed as normal. If the branch is not taken, there is no issue.



If the branch is taken, however, we incur a stalling penalty. This penalty can vary but even in a highly optimized pipeline we will have to essentially "stall" for a cycle.



EXERCISES

For the code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

```
lw $t0,0($t0)
add $t1,$t0,$t0
```

For the code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

```
add $t1,$t0,$t0
addi $t2,$t0,5
addi $t4,$t1,5
```

EXERCISES

For the code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

```
addi  $t1,$t0,1
addi  $t2,$t0,2
addi  $t3,$t0,2
addi  $t3,$t0,4
addi  $t5,$t0,5
```