

# Systems Architecture

IN1006

## Pipelining and Parallelism

—Dr H. Asad





# Where are we?

- Components of computers
- Data representation
- Logic Gates – computer circuits
- Simple computer, assembly programming – MARIE
- Memory hierarchy
- **Pipelining and parallelism**
- System Software



# Question

- Consider the following statements, do you think a CPU always need to run these instructions sequentially (i.e. one after the other)? Is there a way to improve the performance of a CPU?
- $X = a + b;$
- $Y = c + d;$
- $Z = X + Y$



# Outline

- Pipelining
  - MIPS instruction set
  - Pipelining and Hazards
  - Higher level Parallelism
- 
- Note: material on the MIPS architecture are from the book: “*Computer Organization & Design*” by Patterson and Hennessy
  - All essential material is covered in the main textbook of the module, i.e., “*Computer Organization and Architecture*”

# Laundry processing in the real world!

- Assume that laundry is like a processor. Washing clothes is like an instruction. It requires two steps: washing and drying up.
- Assume that washing takes 1h and drying takes 1h.
- Thus, to wash your clothes you need 2h, if you use a single washing and drying machine.
- In 8h there can be  $8/2=4$  loads of washing clothes. So, every 2h we have a new load.

Hour	1	2	3	4	5	6	7	8
JOBS	W	D						
			W	D				
					W	D		
							W	D

- What is the problem here?
- Computer CPUs may suffer from the same problem

# Laundry processing with Pipelining

- Since washing is separate from drying you can use a different machine for each task
- **So, these tasks can be parallelized for different loads!**
- So, we have a new load out every 1h instead of 2h!!!

Hour	1	2	3	4	5	6	7	8
JOBS	W	D						
		W	D					
			W	D				
				W	D			
					W	D		
						W	D	
							W	D



**PIPELINE YOUR LAUNDRY**

# Two Performance Measurements

- 1) **Execution time:** Time it takes to do a single task from start to finish (also known as **latency**)

Hour	1	2	3	4	5	6	7	8
JOBS	W	D						
			W	D				
					W	D		
							W	D

L: ?

T(8h): 4

- 2) **Throughput:** The total amount of work we can get done in any given time

Hour	1	2	3	4	5	6	7	8
JOBS	W	D						
		W	D					
			W	D				
				W	D			
					W	D		
						W	D	
							W	D

L: ?

T(8h): 7



# Pipelining at the Conceptual Level

- Each task can be broken down into a series of smaller tasks
- When these sub-tasks **do not compete for resources**, they can happen at the same time
- Pipelining is a way of executing the smaller tasks in **parallel**
- Simultaneous/**parallel tasks use different resources**
- **Performance:**
  - Pipelining doesn't reduce the latency of a single task BUT
  - It can help increase throughput
- Theoretical speedup (against the non-pipeline) depends on number of **pipe stages**
- **Pipelines stall** (reduces speedup) if there are **dependencies**



# Systems Architecture

IN1006

## Pipelining and Parallelism in Computers

—Dr H. Asad



# How does this apply to computers?

- Execution of an instruction can be broken down into a number of subtasks performed in a fixed sequence
- **These smaller subtasks can often be executed in parallel**
- Such parallel execution is called
  - **instruction pipelining** also known as **instruction level parallelism (ILP)**
- Pipelining keeps the processor busy by scheduling subtasks for different instructions to execute concurrently
- **BUT can we do this?**
- Yes, processors, for example, implement instructions using **F-D-E cycle**

# Let's Remember the FDE Cycle

- The **fetch-decode-execute (FDE)** cycle is the series of steps that a computer carries out when it runs a program.
- It first fetches an instruction from memory, and place it into the IR.
- Once in the IR, it is decoded to determine what needs to be done next. If a memory value (operand) is involved in the operation, it is retrieved and placed into the MBR.
- With everything in place, the instruction is executed.
- **Example:** FDE for the instruction ADD 105:

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR $\leftarrow$ PC	101	1104	101	0023	0023
	IR $\leftarrow$ M[MAR]	101	3105	101	0023	0023
	PC $\leftarrow$ PC + 1	102	3105	101	0023	0023
Decode	MAR $\leftarrow$ IR[11-0]	102	3105	105	0023	0023
	(Decode IR[15-12])	102	3105	105	0023	0023
Get operand	MBR $\leftarrow$ M[MAR]	102	3105	105	FFE9	0023
Execute	AC $\leftarrow$ AC + MBR	102	3105	105	FFE9	000C

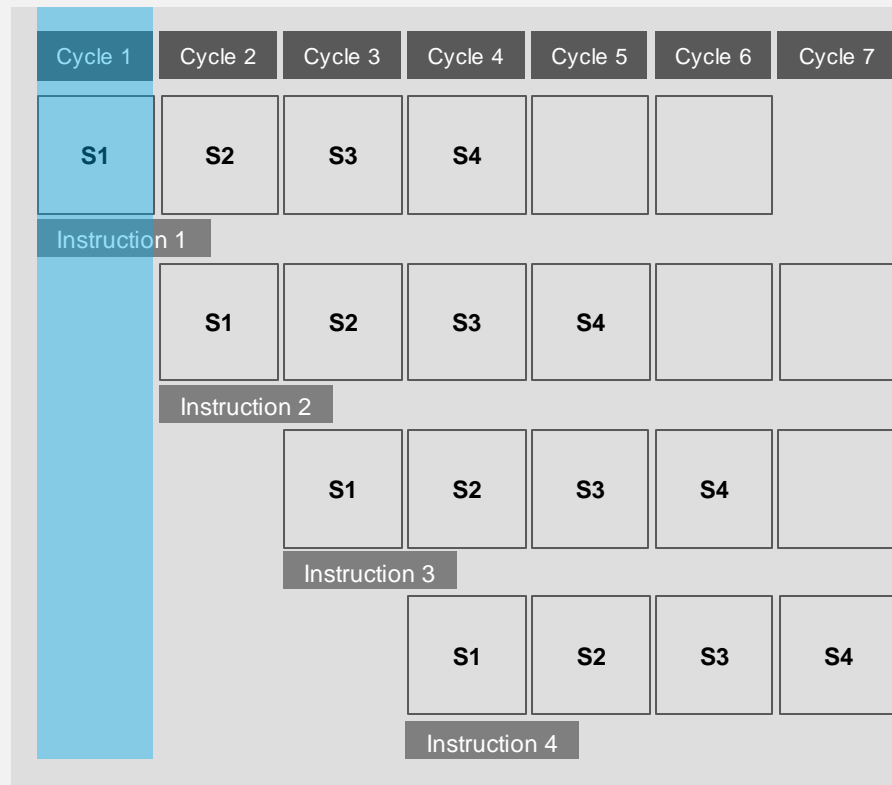
# Pipelining - An example

- Suppose a fetch-decode-execute cycle were broken into the following smaller steps.
- In particular, suppose we have a **Four-stage** pipeline.
  - 1) **S1** fetches the instruction,
  - 2) **S2** decodes it and determines the address of the operands,
  - 3) **S3** fetches operand,
  - 4) **S4** executes the instruction, and stores the result.

# Pipelining: An example

For every clock cycle, one small step is carried out, and **the stages are overlapped**.

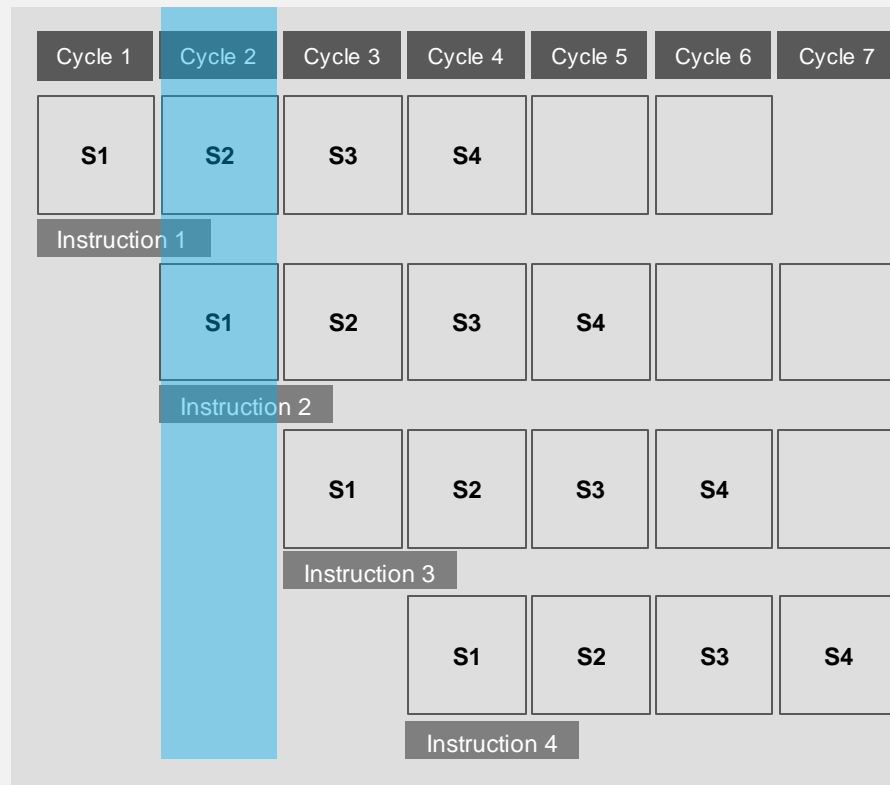
- **S1** fetches the instruction
- **S2** decodes it and determines the address of the operands
- **S3** fetches operand
- **S4** executes the instruction and stores the result.



# Pipelining: An example

For every clock cycle, one small step is carried out, and **the stages are overlapped**.

- **S1** fetches the instruction
- **S2** decodes it and determines the address of the operands
- **S3** fetches operand
- **S4** executes the instruction and stores the result.



# Pipelining: An example

For every clock cycle, one small step is carried out, and **the stages are overlapped**.

- **S1** fetches the instruction
- **S2** decodes it and determines the address of the operands
- **S3** fetches operand
- **S4** executes the instruction and stores the result.



# Pipelining: An example

For every clock cycle, one small step is carried out, and **the stages are overlapped**.

- **S1** fetches the instruction
- **S2** decodes it and determines the address of the operands
- **S3** fetches operand
- **S4** executes the instruction and stores the result.





# Pipelining: An example

For every clock cycle, one small step is carried out, and **the stages are overlapped**.

- **S1** fetches the instruction
- **S2** decodes it and determines the address of the operands
- **S3** fetches operand
- **S4** executes the instruction and stores the result.

Throughput?

Throughput improvement  
over sequential execution?



# Systems Architecture

IN1006

## Pipelining and Parallelism- Hazards

—Dr H. Asad



# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- S1: fetch instruction
- S2: decode/calculate address
- S3: fetch op

Instruction	Time period →												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(Branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
7							S1	S2	S3	S4			
8								S1	S2	S3	S4		
9									S1	S2	S3	S4	
10										S1	S2	S3	S4

# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- **S1:** fetch instruction
- **S2:** decode/calculate address
- **S3:** fetch operand
- **S4:** execute and store result

Assumptions:

- Data and instructions can be fetched in parallel
- **Instruction 3 is a conditional branch** (i.e. 'if then else' or **MARIE skipcond instruction**):  
If part → execute Instruction 4  
Else part → execute Instruction 8

	Time period →												
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(Branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
7							S1	S2	S3	S4			
8								S1	S2	S3	S4		
9									S1	S2	S3	S4	
10										S1	S2	S3	S4

# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- **S1:** fetch instruction
- **S2:** decode/calculate address
- **S3:** fetch operand
- **S4:** execute and store result

Assumptions:

- Data and instructions can be fetched in parallel
- **Instruction 3 is a conditional branch** (i.e. 'if then else' or **MARIE skipcond instruction**):  
If part → execute Instruction 4  
Else part → execute Instruction 8

	Time period →												
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(Branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
7							S1	S2	S3	S4			
8								S1	S2	S3	S4		
9									S1	S2	S3	S4	
10										S1	S2	S3	S4

Yes: correctly executes

# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- **S1:** fetch instruction
- **S2:** decode/calculate address
- **S3:** fetch operand
- **S4:** execute and store result

Assumptions:

- Data and instructions can be fetched in parallel
- **Instruction 3 is a conditional branch** (i.e. 'if then else' or **MARIE skipcond instruction**):  
If part → execute Instruction 4  
Else part → execute Instruction 8

	Time period →												
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(Branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
7							S1	S2	S3	S4			
8								S1	S2	S3	S4		
9									S1	S2	S3	S4	
10										S1	S2	S3	S4

Yes: correctly executes

Yes: correctly executes

# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- **S1:** fetch instruction
- **S2:** decode/calculate address
- **S3:** fetch operand
- **S4:** execute and store result

Assumptions:

- Data and instructions can be fetched in parallel
- **Instruction 3 is a conditional branch** (i.e. 'if then else' or **MARIE skipcond instruction**):  
If part → execute Instruction 4  
Else part → execute Instruction 8

	Time period →													
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	
1	S1	S2	S3	S4										Yes: correctly executes
2		S1	S2	S3	S4								Yes: correctly executes	
(Branch) 3			S1	S2	S3	S4								
4				S1	S2	S3								Yes: correctly executes
5					S1	S2								
6						S1								
7							S1	S2	S3	S4				
8								S1	S2	S3	S4			
9									S1	S2	S3	S4		
10										S1	S2	S3	S4	

# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- **S1:** fetch instruction
- **S2:** decode/calculate address
- **S3:** fetch operand
- **S4:** execute and store result

Assumptions:

- Data and instructions can be fetched in parallel
- **Instruction 3 is a conditional branch** (i.e. 'if then else' or **MARIE skipcond instruction**):  
If part → execute Instruction 4  
Else part → execute Instruction 8

Instruction	Time period →												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(Branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
7							S1	S2	S3	S4			
8								S1	S2	S3	S4		
9									S1	S2	S3	S4	
10										S1	S2	S3	S4

Yes: correctly executes

Yes: correctly executes

Yes: correctly executes

Incorrect instruction, no execution



# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- **S1:** fetch instruction
- **S2:** decode/calculate address
- **S3:** fetch operand
- **S4:** execute and store result

Assumptions:

- Data and instructions can be fetched in parallel
- **Instruction 3 is a conditional branch** (i.e. 'if then else' or **MARIE skipcond instruction**):  
If part → execute Instruction 4  
Else part → execute Instruction 8

Instruction	Time period →												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(Branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
7							S1	S2	S3	S4			
8								S1	S2	S3	S4		
9									S1	S2	S3	S4	
10										S1	S2	S3	S4

Yes: correctly executes

Yes: correctly executes

Yes: correctly executes

Incorrect instruction, no execution

Incorrect instruction, no execution

# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- **S1:** fetch instruction
- **S2:** decode/calculate address
- **S3:** fetch operand
- **S4:** execute and store result

Assumptions:

- Data and instructions can be fetched in parallel
- **Instruction 3 is a conditional branch** (i.e. 'if then else' or **MARIE skipcond instruction**):

If part → execute Instruction 4

Else part → execute Instruction 8

Instruction	Time period →												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(Branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
7							S1	S2	S3	S4			
8								S1	S2	S3	S4		
9									S1	S2	S3	S4	
10										S1	S2	S3	S4

Yes: correctly executes

Yes: correctly executes

Yes: correctly executes

Incorrect instruction, no execution

Incorrect instruction, no execution

Incorrect instruction, no execution

# Instruction Pipeline with a Conditional Branch

Suppose that we have a 4-stage pipeline:

- **S1:** fetch instruction
- **S2:** decode/calculate address
- **S3:** fetch operand
- **S4:** execute and store result

Assumptions:

- Data and instructions can be fetched in parallel
- **Instruction 3 is a conditional branch** (i.e. 'if then else' or **MARIE skipcond instruction**):  
If part → execute Instruction 4  
Else part → execute Instruction 8

	Time period →													
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	
1	S1	S2	S3	S4										Yes: correctly executes
2		S1	S2	S3	S4									Yes: correctly executes
(Branch) 3			S1	S2	S3	S4								Yes: correctly executes
4				S1	S2	S3								Incorrect instruction, no execution
5					S1	S2								Incorrect instruction, no execution
6						S1								Incorrect instruction, no execution
7							S1	S2	S3	S4				Yes: correctly executes
8								S1	S2	S3	S4			
9									S1	S2	S3	S4		
10										S1	S2	S3	S4	

# Pipeline Hazards

- We assume that the pipeline can be kept filled at all times BUT
- This is not always the case as **pipeline hazards** may arise causing **pipeline conflicts** and **stalls**.
- Hazards are situations that prevent starting the next instruction in the next cycle
- Three Types of pipeline hazards:
  - 1) **Control Hazards:** Loading instructions into pipeline before the result of a decision is known, e.g. loading instructions after a branch
  - 2) **Data Hazards:** Instruction depends upon the results of a previous instruction still in the pipeline, e.g. compound math expressions
  - 3) **Structural (or Resource) Hazards:** Hardware cannot support some combinations of instructions in the same clock cycle, e.g.
    - two simultaneous memory accesses
    - one instruction is reading from memory and another is writing at the same stage

# Microprocessor without Interlocked Pipelined Stages (MIPS)<sup>1</sup>

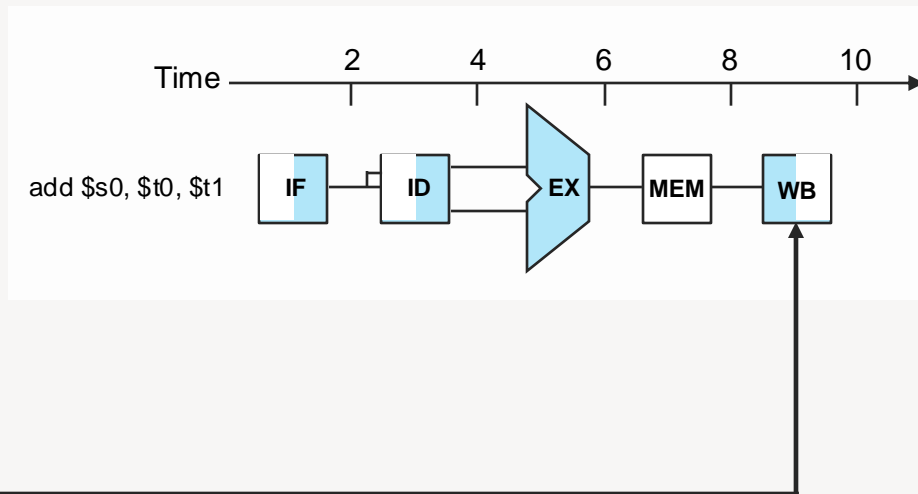
- a reduced instruction set computer (RISC) instruction set architecture (ISA)
- originally, MIPS was designed for general-purpose computing
- As of April 2017, MIPS processors are used in embedded systems such as residential gateways and routers

<sup>1</sup>[https://en.wikipedia.org/wiki/MIPS\\_architecture](https://en.wikipedia.org/wiki/MIPS_architecture)

# Notation for MIPS Pipeline

## Five stages, one step per stage:


- **IF:** Instruction fetch from memory
- **ID:** Instruction decode and register read
- **EX:** Execute operation or calculate address
- **MEM:** Access memory operand
- **WB:** Write result back to register



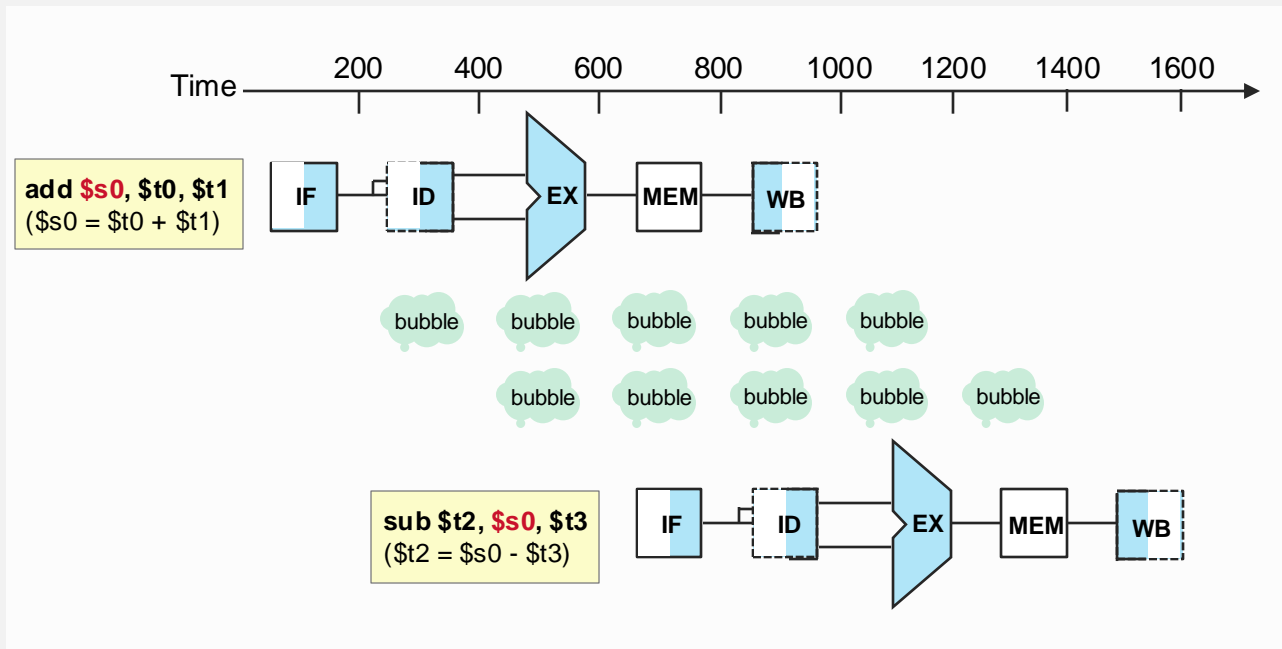
## Example instruction:

- **add** `$s0, $t0, $t1` →  $\$s0 = \$t0 + \$t1$   
where  $\$x$  denotes the contents of register named  $x$
- **add** does not write until **5<sup>th</sup> stage** of the pipeline

# The MIPS Instruction Set

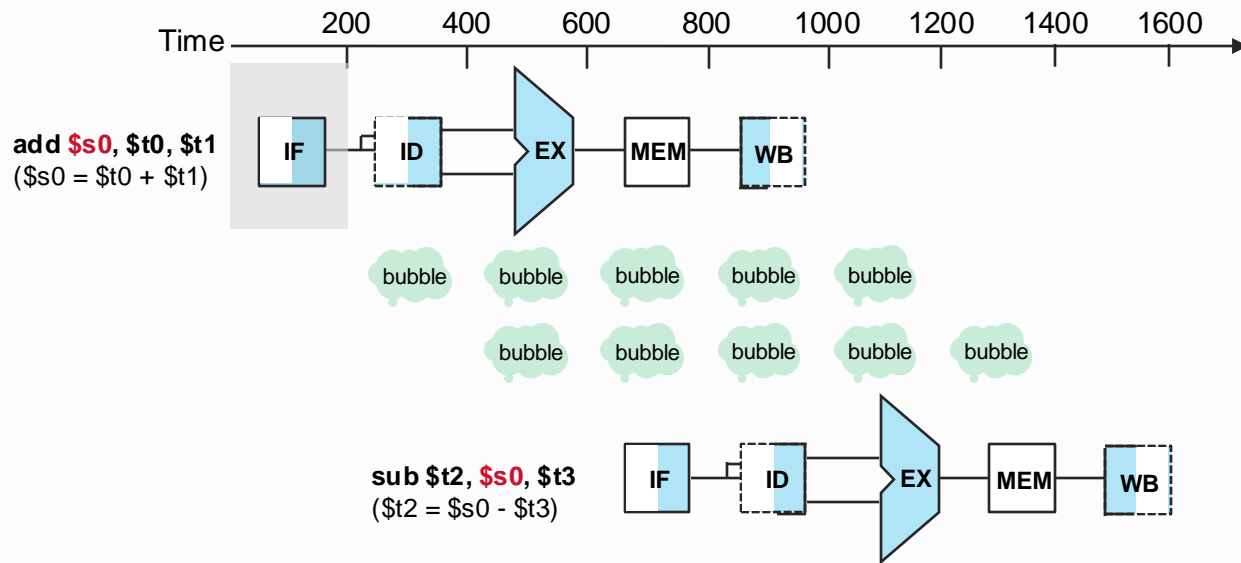
- Stanford MIPS ([http://en.wikipedia.org/wiki/MIPS\\_instruction\\_set](http://en.wikipedia.org/wiki/MIPS_instruction_set))
- Add and subtract, three operands
  - Two sources and one destination
  - MIPS has a  $32 \times 32$ -bit registers
  - **\$x notation denotes the contents of register named x**
- All arithmetic operations have the following form
  - **add \$s0, \$t0, \$t1**  $\rightarrow$   $\$s0 = \$t0 + \$t1$
  - **sub \$t3, \$t0, \$t1**  $\rightarrow$   $\$t3 = \$t0 - \$t1$
  - **lw \$t0, 32(\$s3)**  $\rightarrow$  load word from memory address 32(\$s3) into register \$t0
  - **sw \$t0, 48(\$s3)**  $\rightarrow$  store word from register t0 into memory address 48(\$s3)
- Memory address: example, 32(\$s3)

# Data Hazard - an example



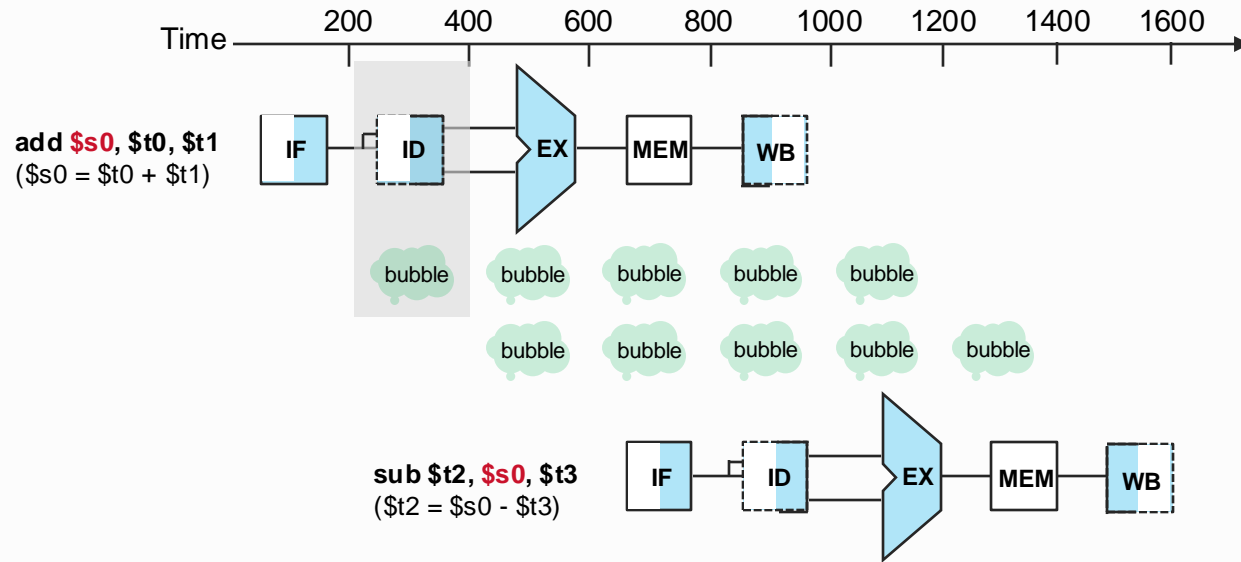


## Data Hazard - an example



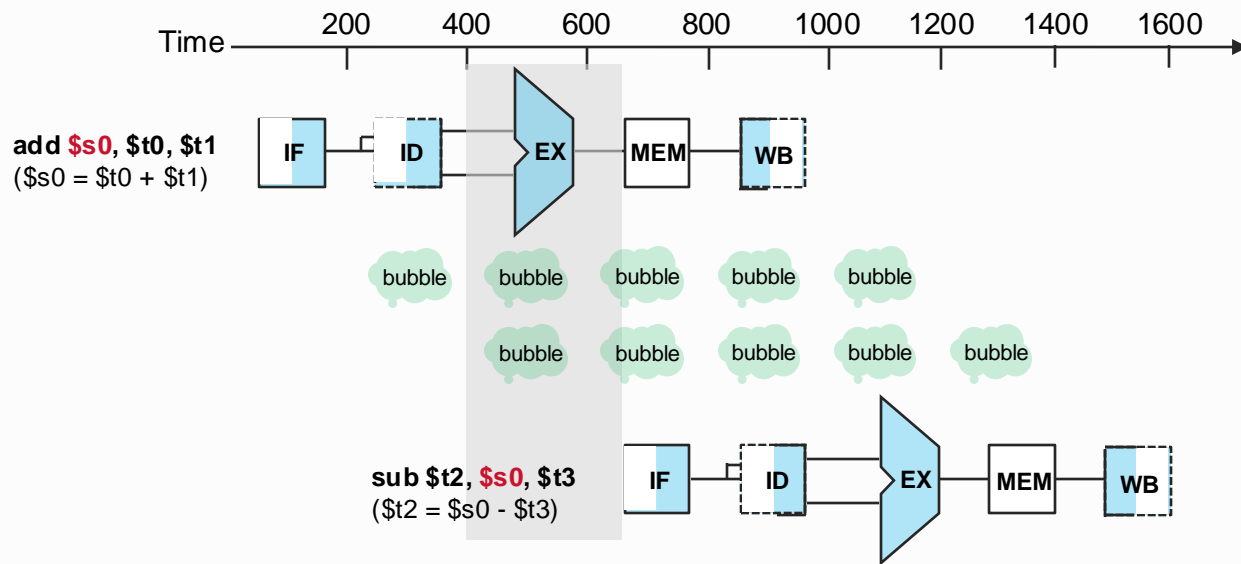
**1<sup>st</sup> cycle:**  
Instruction  
is Fetched

# Data Hazard - an example



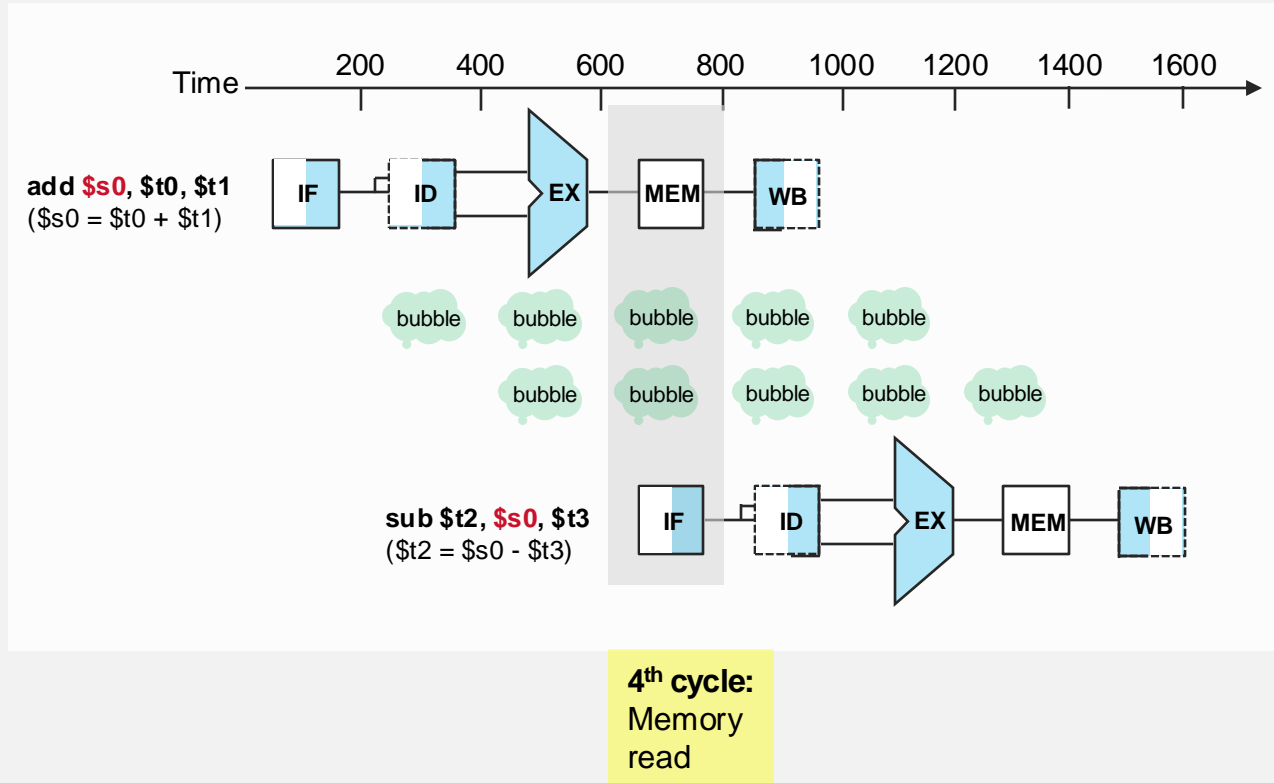
**2<sup>nd</sup> cycle:**  
Information Decoded

## Data Hazard - an example

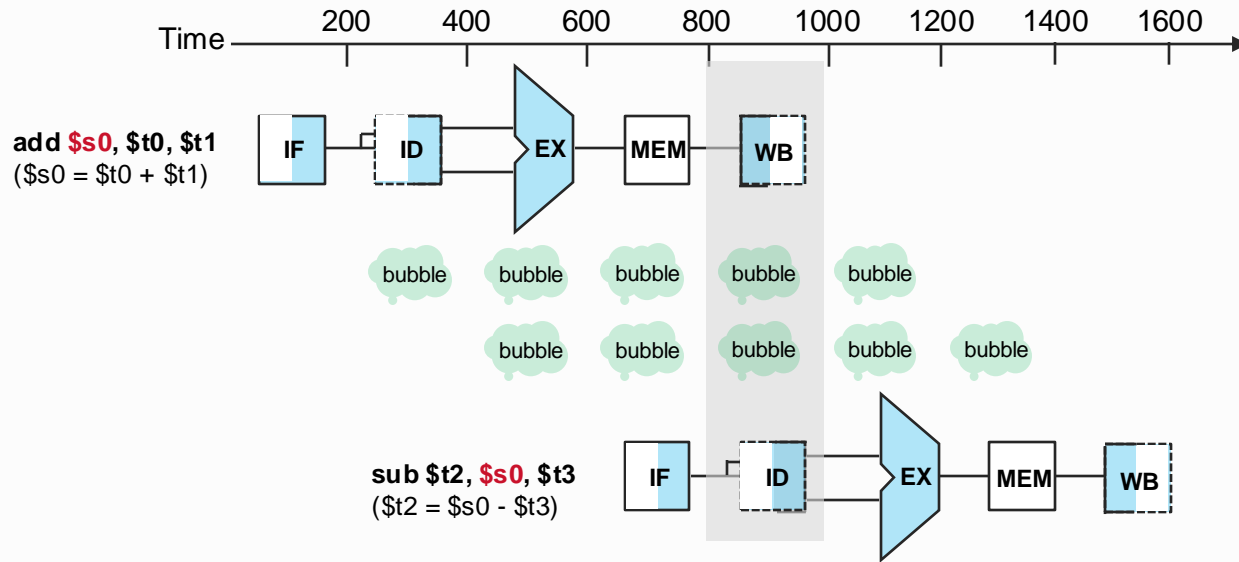


### 3<sup>rd</sup> cycle: Execution

# Data Hazard - an example

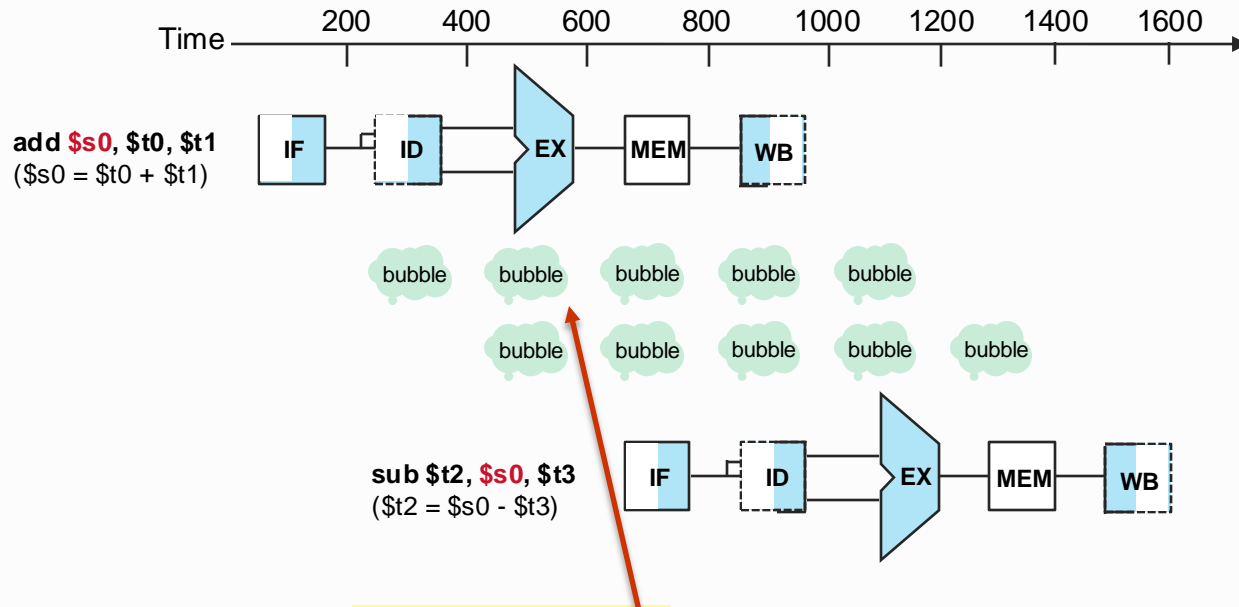


# Data Hazard - an example



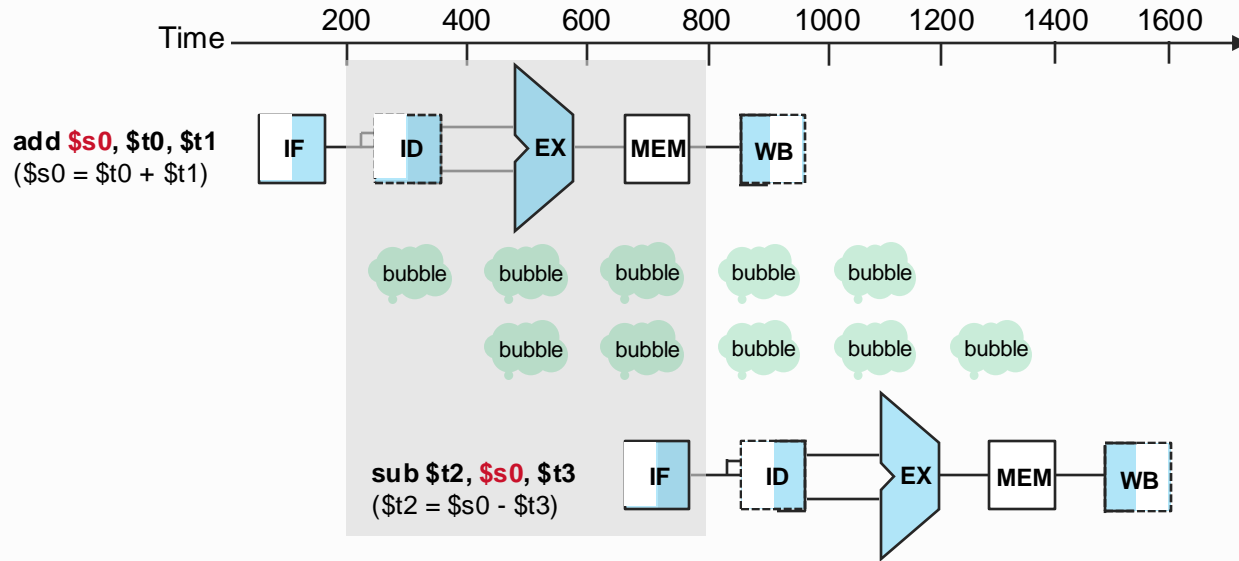
**5<sup>th</sup> cycle:**  
**Memory Write Back**  
**add** does not write until  
this stage of the pipeline

# Data Hazard - an example



**sub** needs to read  
data in pipeline

# Data Hazard - an example



Therefore, the **stall** would have to last three stages if it is to **align/sync write with read (EX after WB)** → insert stalls/bubbles or no-op operations.

# Systems Architecture

IN1006

## Pipelining and Parallelism- Hazard Solutions

—Dr H. Asad

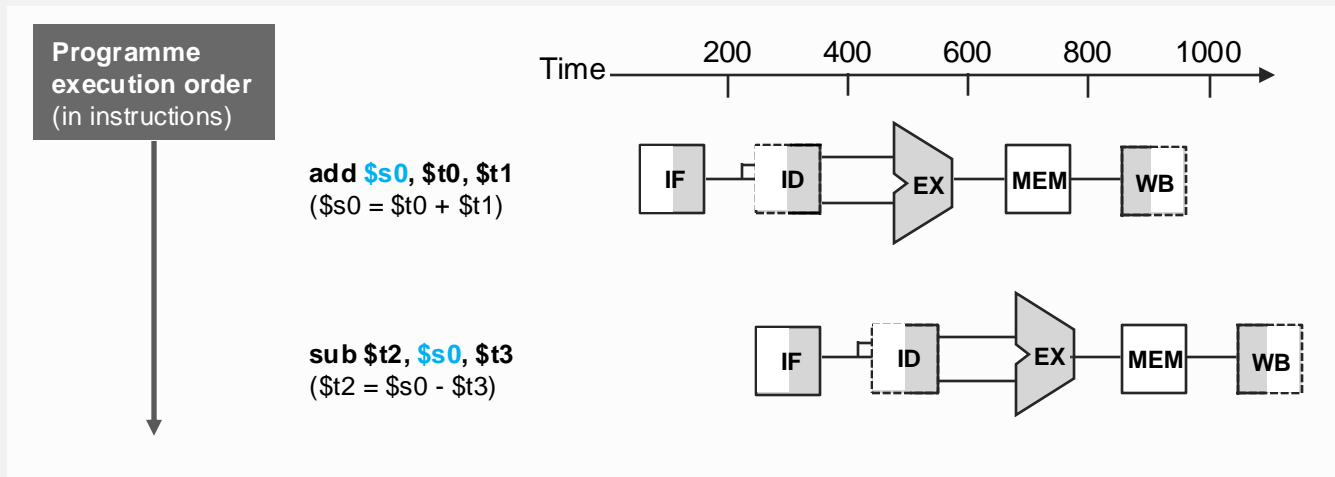




# Resolving Data Hazards – Solution 1 (forwarding)

## Forwarding:

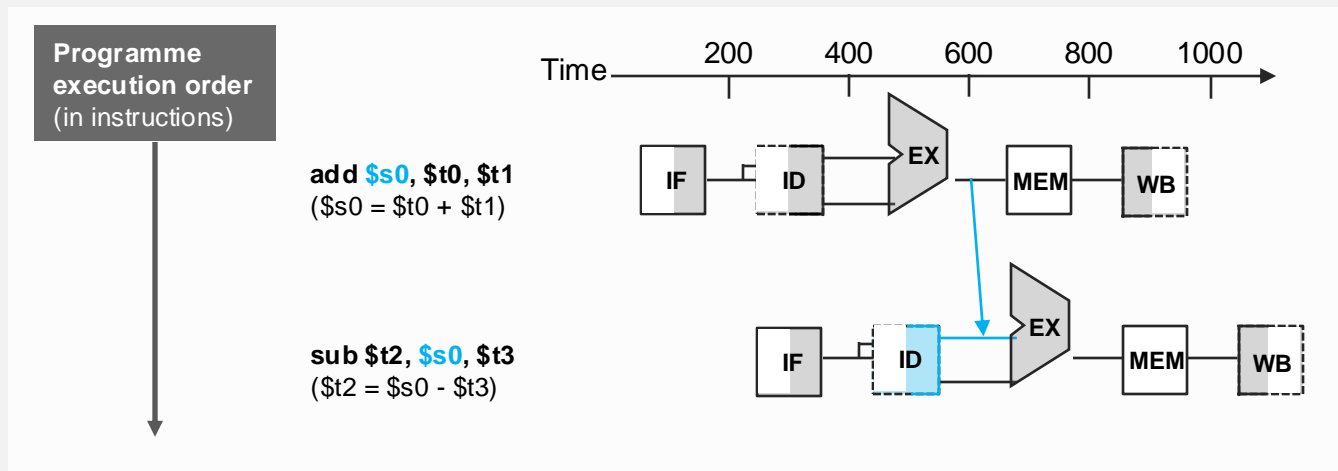
- Use the results before they are written to the registers (**needs extra logic**).
- Output of ALU for add is forwarded to input of ALU for sub, overriding the value retrieved for \$s0.
- Don't wait until (WB) when the result is written to \$s0
- **Forwarding is a hardware solution**



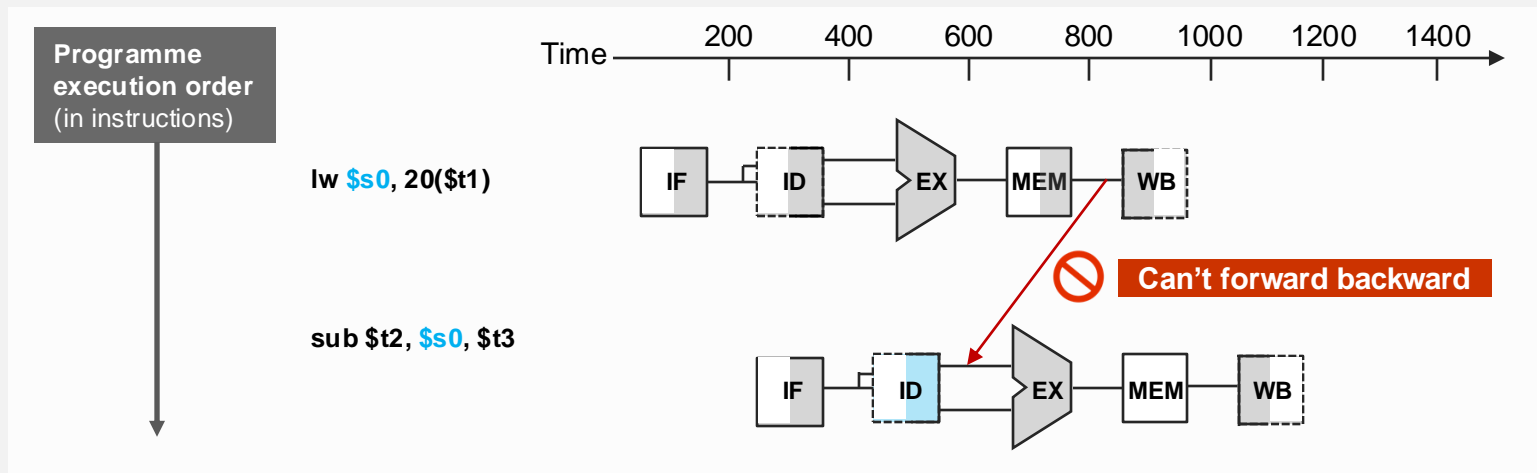
# Resolving Data Hazards – Solution 1 (forwarding)

## Forwarding:

- Use the results before they are written to the registers (**needs extra logic**).
- Output of ALU for add is forwarded to input of ALU for sub, overriding the value retrieved for \$s0.
- Don't wait until (WB) when the result is written to \$s0
- **Forwarding is a hardware solution**

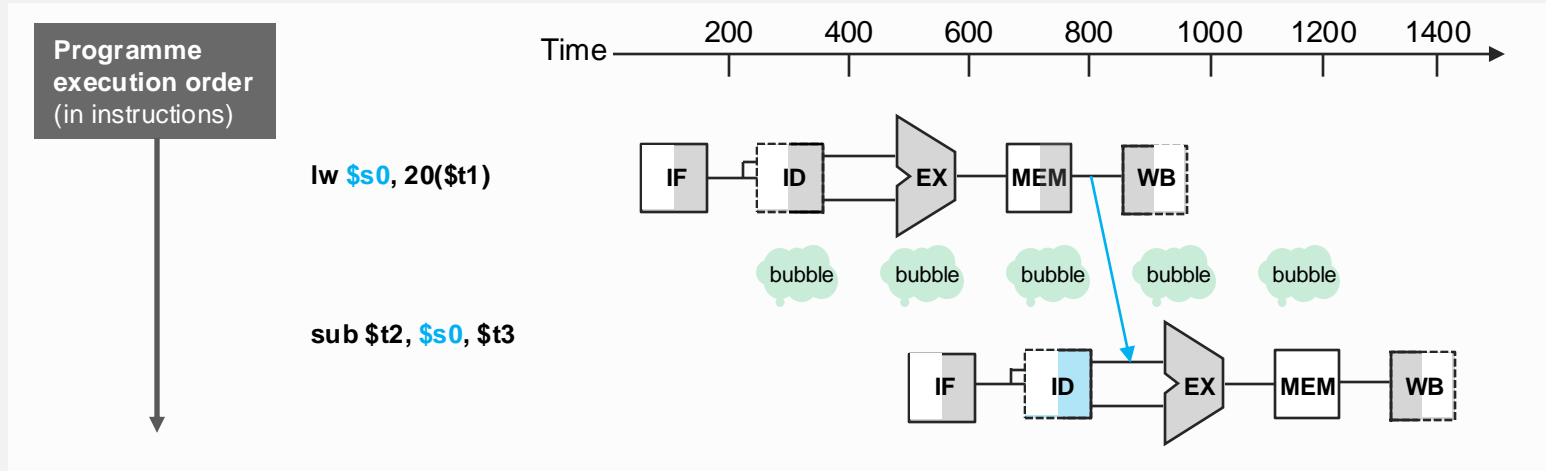


# Resolving Data Hazards – Solution 2 (load-use)



**Load-Use:** if value not computed when needed, **can't forward backward in time**

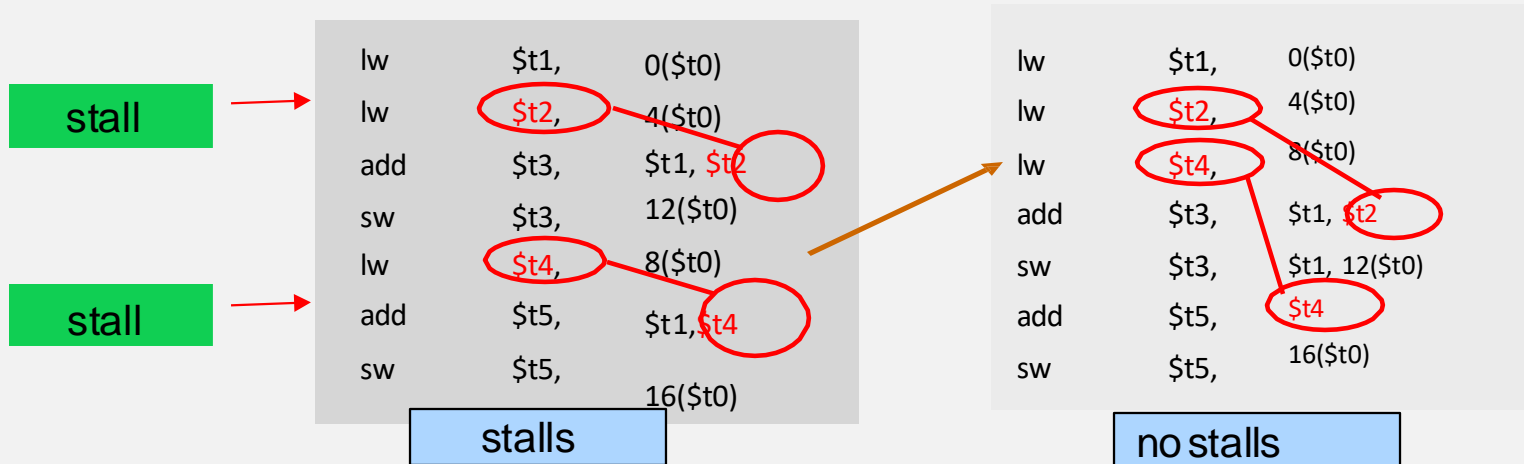
# Resolving Data Hazards – Solution 2 (load-use)



- We use both bubbles and forwarding
- Instructions which cannot be executed repeat their pipeline stage until they can advance (**interlock**)

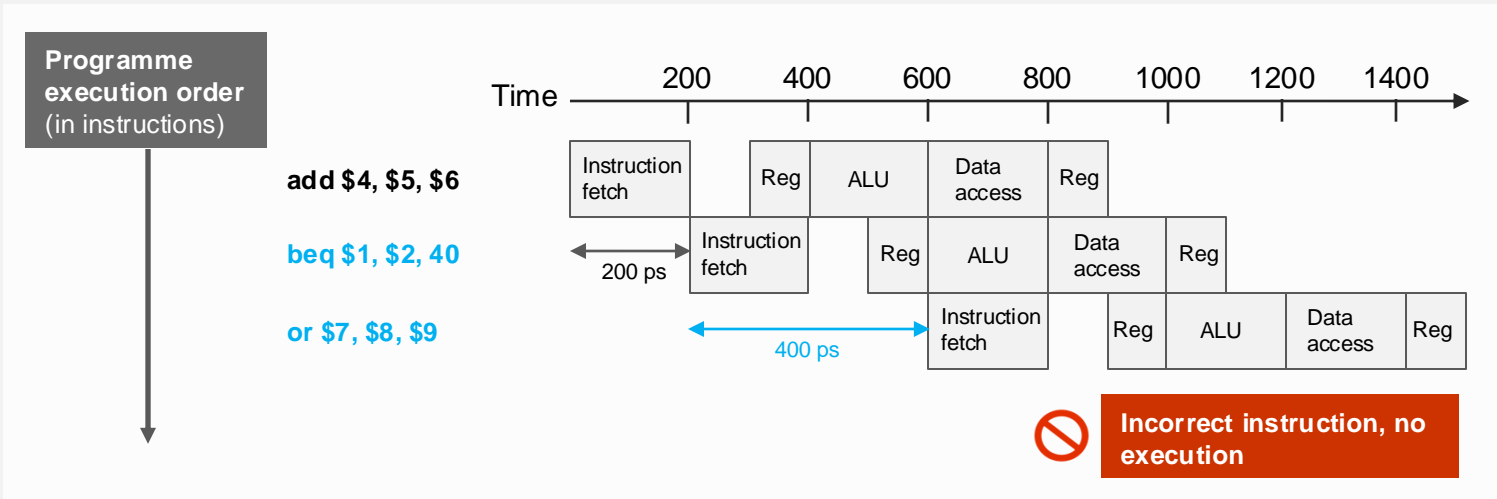
# Resolving Data Hazards – Solution 3 (Compiler optimisation)

- **Compiler Optimisation:** Re-arrange instructions to avoid data hazards where possible (software solution)
- Assume the C code:
  - $A = B + E$ ;
  - $C = B + F$ ;
  - B is stored in memory in  $0(\$t0)$ , E in  $4(\$t0)$ , F in  $8(\$t0)$



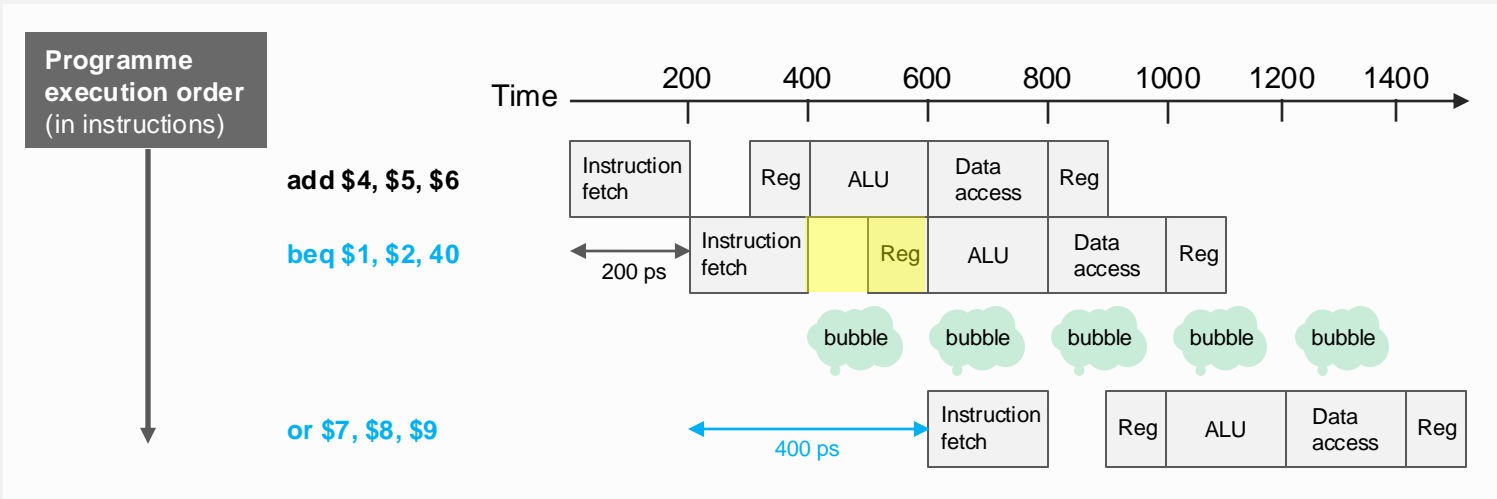
# Resolving Control Hazards: Solution 1

- **Stall on Branch:** Wait until branch outcome determined before fetching next instruction.
- Branch determines flow of control. Fetching next instruction depends on branch outcome. Pipeline can't always fetch correct instruction. Still working on ID stage of branch.
- In MIPS pipeline: Need to **compare registers** and compute target early in the pipeline. **Add hardware to do it in ID stage.** Example instruction branch if equal: **beq rs, rt, L1.** if (rs == rt) branch to instruction labelled L1.



# Resolving Control Hazards: Solution 1

- **Stall on Branch:** Wait until branch outcome determined before fetching next instruction.
- Branch determines flow of control. Fetching next instruction depends on branch outcome. Pipeline can't always fetch correct instruction. Still working on ID stage of branch.
- In MIPS pipeline: Need to **compare registers** and compute target early in the pipeline. **Add hardware to do it in ID stage.** Example instruction branch if equal: **beq rs, rt, L1.** if (rs == rt) branch to instruction labelled L1.





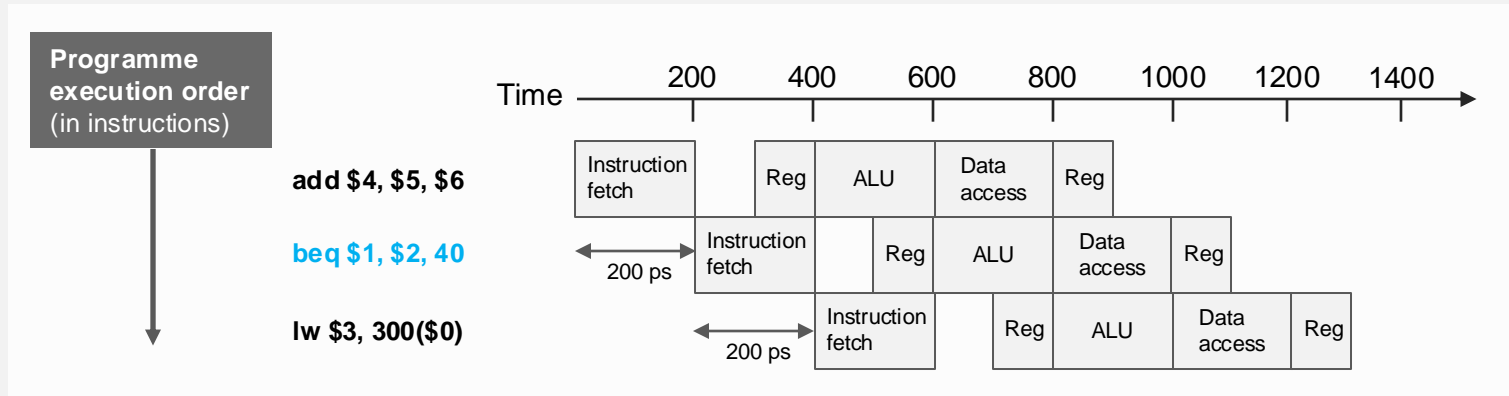
## Resolving Control Hazards: Solution 2

- **Branch Prediction:** Assume an outcome for decisions (e.g. branch always fails) and load pipeline accordingly (**speculative execution**).
- It uses logic to predict the outcome of the condition.
  - If the prediction is correct then the pipeline remains full at all times and the decision costs nothing
  - If the prediction is incorrect then the pipeline will stall (as before) and new instructions have to be loaded.



# Resolving Control Hazards: Solution 2

Prediction correct



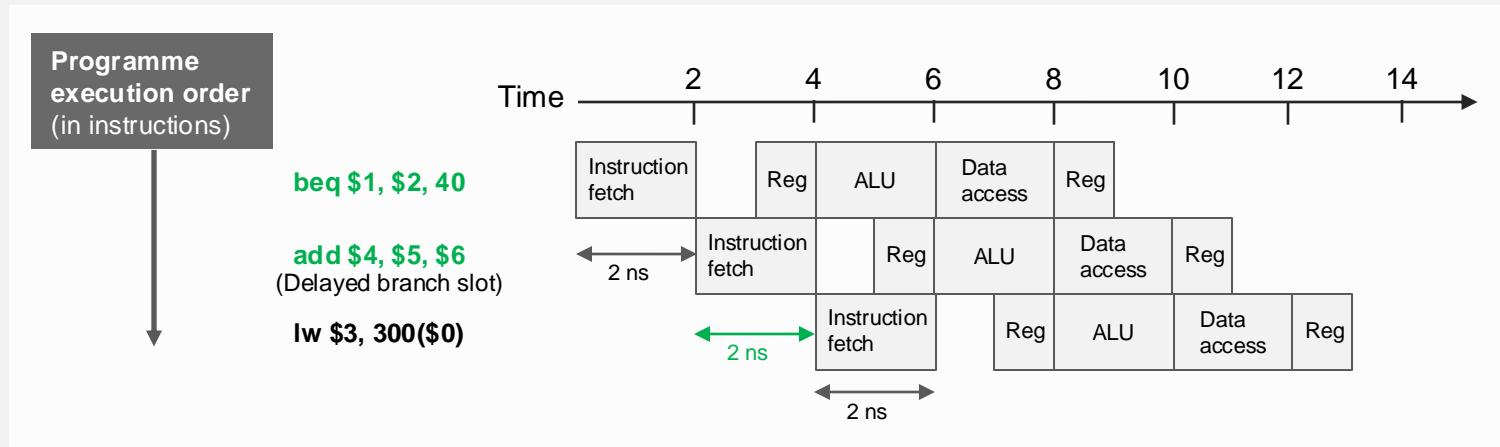
# Resolving Control Hazards – Combined Solution

## Delayed Decision:

Rearrange the instructions so that **an instruction that does not affect the decision is executed after the decision** filling the bubble that would occur if the pipeline were to stall.

add	\$4,	\$5,	\$6
beq	\$1,	\$2,	40
lw	\$3,	300(\$0)	

beq	\$1,	\$2,	40
add	\$4,	\$5,	\$6
lw	\$3,	300(\$0)	



# Systems Architecture

IN1006

## Pipelining and Parallelism- Current Trends

—Dr H. Asad



# Current Pipelining Trends

- Super (aka deep) pipelining → **increase number of stages in execution of statements**
  - Maximum speedup is related to number of pipeline stages, some recent processors have **eight or more stages**
- There is a **trade-off** as **deep** (many stages) **pipelines** are **more vulnerable to hazards**
  - More complex measures needed to deal with this...
  - Requires more logic to implement...
- Deep pipelines usually require additional logic

# Instruction Level Parallelism (ILP)

- **Instruction-level parallelism (ILP)** is a measure

of how many of the operations in a computer program can be performed simultaneously

- ILP so far: decompose an instruction into smaller stages and allow overlaps these stages

- Alternative ILP: **allow individual instructions to overlap**

- For example:  
Operation 1:  $e = a + b$

Operation 2:  $f = c + d$

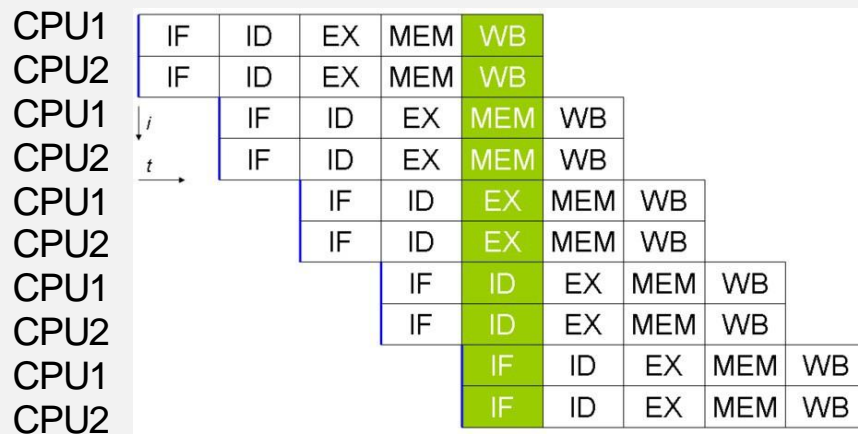
Operation 3:  $g = e * f$

- Operation 3 depends on the results of 1 and 2
- **Operations 1 and 2 do not depend on other operations, so they can be executed in parallel**

# Superscalar Architectures

## ■ Superscalar Pipelining

- Replicate pipeline hardware so the processor can execute multiple instructions in different pipelines simultaneously i.e. multiple execution units
- This requires more than one instruction to be fetched and executed per cycle
- More strain on cache
  - But, can exploit ILP further
  - Examples: IBM's PowerPC, Sun's UltraSparc





# Superscalar pipelining: Issues

- Need to keep execution units full
  - ILP is key to this.
- If instructions depend on each other, this can affect ILP
- Requires hardware:
  - more execution units (globally faster)
  - more transistors means more power and more delay (locally slower)
- It's always a trade-off
- Methods such as **out-of-order execution** help

# Out-of-Order Execution

- Seen so far: **In-order-execution**
- Fetch instruction.
- If input operands are available (in processor registers, for instance), the instruction is dispatched to the appropriate functional unit.
- The instruction is executed by the appropriate functional unit.
- The functional unit writes the results back to the register file.



# Out-of-Order Execution

- Seen so far: **In-order-execution**
- Fetch instruction.
- If input operands are available (in processor registers, for instance), the instruction is dispatched to the appropriate functional unit.
- The instruction is executed by the appropriate functional unit.
- The functional unit writes the results back to the register file.

**Stalls**

# Out-of-Order Execution

- Seen so far: In-order-execution

- Fetch instruction.
- If input operands are available (in processor registers, for instance), the instruction is dispatched to the appropriate functional unit.
- The instruction is executed by the appropriate functional unit.
- The functional unit writes the results back to the register file.

**Stalls**

## Out-of-order-execution

- Fetch instruction.
- Dispatch instruction to an **instruction queue**.
- The instruction waits in the queue until all its input operands are available. The instruction is then allowed to leave the queue before earlier instructions.
- The instruction is issued to the appropriate functional unit and executed by that unit.
- The results are queued.
- Only after all earlier instructions have their results written back to the register file, then this result is written back to the register file. This is called the **graduation** or **retire stage**.

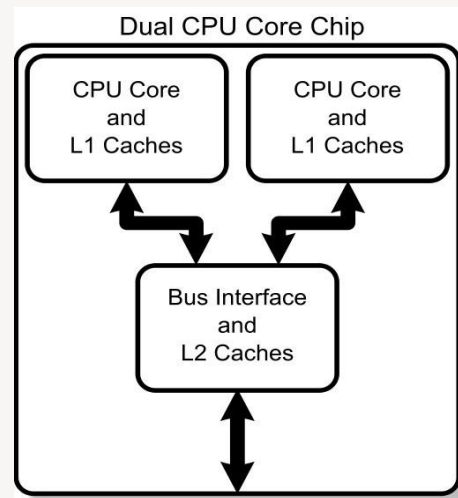


# Thread Level Parallelism (TLP)

- A **thread** in computer science is short for a **thread of execution**
- Threads allow a program to split itself into two or more simultaneously running tasks
- Like ILP, the more threads available the more parallel computing can be achieved

# Symmetric Multiprocessing (SMP)

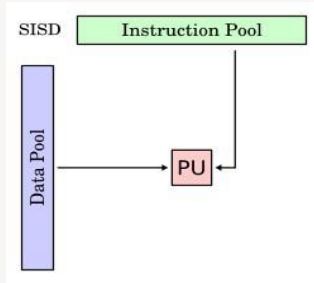
- SMP is a **multiprocessor computer architecture**:
  - It is an architecture with **two or more identical processors**, connected to a **single shared main memory**
- With SMP, TLP can be exploited by assigning threads across processors
- BUT issues may arise:
  - A shared bus is used to connect the processors to memory → limits on communication and bandwidth
  - Code needs to be thread-aware; not easy and may affect individual processor performance
  - Operating systems need to recognise SMP
  - Keeping memory and caches coherent is complex



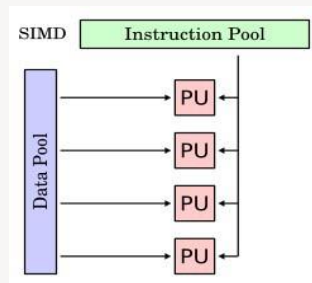
# Flynn's Taxonomy(categorization of forms of parallel computer architectures)

- **Single Instruction, Single Data stream (SISD)**
  - A sequential computer which exploits no parallelism in either the instruction or data streams
  - Examples: the traditional uniprocessor machines like a PC or traditional mainframes
- **Multiple Instruction, Single Data stream (MISD)**
  - Unusual due to the fact that multiple instruction streams generally require multiple data streams to be effective
  - e.g. critical control systems (fault tolerance)
- **Single Instruction, Multiple Data streams (SIMD)**
  - A computer which runs a single instruction on multiple data streams to perform operations which may be naturally parallelised, e.g. an array processor or GPU
- **Multiple Instruction, Multiple Data streams (MIMD)**
  - Multiple autonomous processors simultaneously executing different instructions on different data
  - Distributed systems are generally recognised to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space

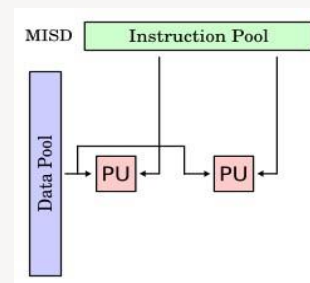
# Flynn's Taxonomy – summary



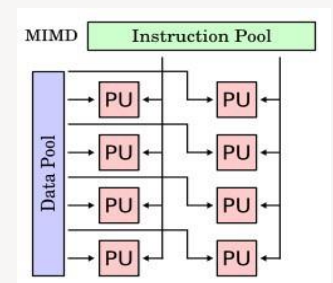
Single Instruction,  
Single Data stream



Single  
Instruction,  
Multiple Data  
streams



Multiple  
Instruction,  
Single Data  
stream



Multiple  
Instruction,  
Multiple Data  
streams

# SIMD: Vector processors

- Uses a bank of registers to represent vectors
- One instruction is applied to a set of registers
- Advantages:
  - Savings in fetching and decoding the instruction itself, which only has to be done once
  - The code itself is also smaller, which can lead to more efficient memory use
  - Parallelism for superscalar implementation

## Normal (Scalar) Processor

Do 10 times:

```
fetch  this  number
fetch  that  number
add them
put the result in memory
```

## Vector Processor:

```
fetch first group 10 numbers
fetch second group of 10 numbers
add them
put the results in memory
```



# Some examples

- SISD:

add X

sub Y

- SIMD:

add a,b,c

add x,y,z

- MISD:

add a,b,c

sub x,b,c





# Summary

- What is Pipelining?
- Pipeline Hazards
- Resolving Pipeline Hazards
- Current Pipeline Trends
- Out-of-Order Execution
- Superscalar Architectures
- TLP/SMP
- Multi-core processors
- Flynn's Taxonomy
- SIMD: Vector Processors
- Examples

## **School of Science & Technology**

City, University of London

Northampton Square

London

EC1V 0HB

United Kingdom

T: +44 (0)20 7040 5060

E: [SST-ug@city.ac.uk](mailto:SST-ug@city.ac.uk)

[www.city.ac.uk/department](http://www.city.ac.uk/department)

