Computer Architecture and Organization

Chapter 8
Pipelining

What is Pipelining?

- Implementation technique whereby multiple instructions are overlapped in execution
- Throughput of an instruction pipeline is determined by how often an instruction exists the pipeline - The *time per instruction* does not change, only *throughput* increases
- The basic idea is to split the datapath into stages

What is Pipelining?

- Potential speedup = Number pipe stages

- Pipeline rate limited by slowest pipeline stage (***length*** of the machine cycle)

- Unbalanced lengths of pipe stages reduces speedup (involves overhead) - since the clock can run no faster than the time needed for the slowest pipeline stage

What is Pipelining?

- Time to “fill” pipeline and time to “drain” (**pipeline bubbles or** no-operations (NOPs)**, load delay)** it reduces speedup

- Dependencies (data, procedural, resource conflicts, anti-dependency) stall the pipeline - implies that there would be a chain of one or more hazards between the two instructions, thereby reducing or eliminating the overlap

True data dependency

- ADD r1, r2 (r1 := r1+r2;)
- MOVE r3,r1 (r3 := r1;)
- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished

Procedural Dependency

- Can not execute instructions after a branch in parallel with instructions before a branch
- Also, if instruction length is not fixed, instructions have to be decoded to find out how many fetches are needed
- This prevents simultaneous fetches

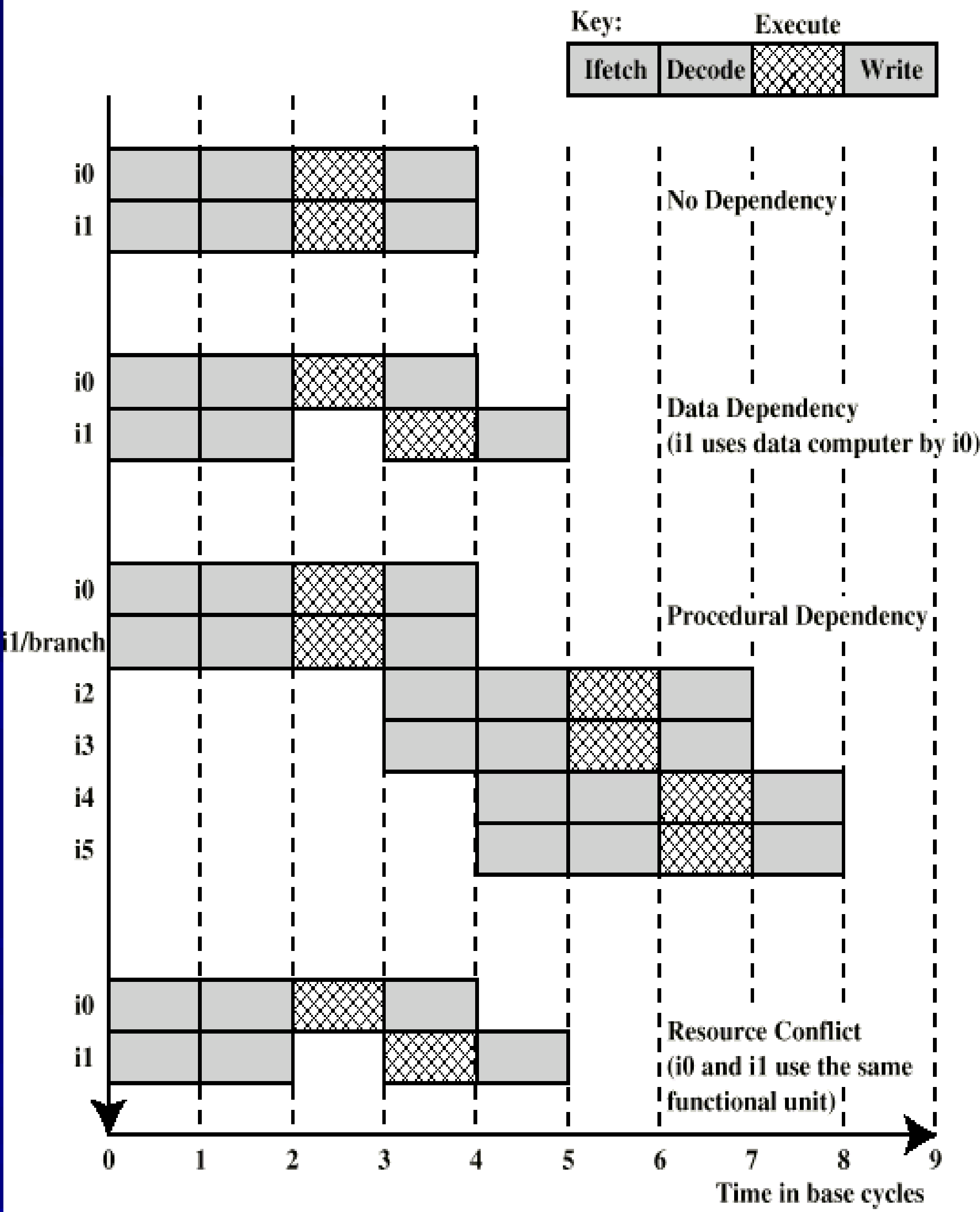
Resource Conflicts

- Two or more instructions requiring access to the same resource at the same time
  - e.g. two arithmetic instructions
- Can duplicate resources
  - e.g. have two arithmetic units

Anti-dependency

- Write-write dependency
  - R3:=R3 + R5; (I1)
  - R4:=R3 + 1;   (I2)
  - R3:=R5 + 1;   (I3)
  - R7:=R3 + R4; (I4)
  - I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3

Dependencies



Hardware Requirements

- Extra incrementer to update the PC more often (instead of the ALU)
- A separate MDR for loads (memory to CPU) and stores (CPU to memory)
- High memory bandwidth to accommodate more data to and from memory

Stage

- A *stage* of the pipeline is a portion of the pipeline execution where different phases in different instructions execute at the same time.
- Stages are connected one to the next to form a pipe - instructions enter at one end, progress through the stages, and exit at the other end.

Depth

- The *depth* of a pipeline is the number of instructions it can overlap

Designer’s Goal

To balance the length of each pipeline stage

For a perfectly balanced stage:

$$T_{ip} = \frac{T_{iup}}{N_s}$$

N_s = Number of pipe stages

T_{ip} = Time per instruction on the pipelined machine

T_{iup} = Time per instruction on unpipelined machine

Example 1

Given: Consider an unpipelined machine with 6 execution stages of lengths 55 ns, 55 ns, 60 ns, 60 ns, 55 ns, and 55 ns.

Determine: Instruction latency on this machine;

How much time does it take to execute 100 instructions?

Solution to Example 1

Instruction latency *= 55+55+60+60+55+55= 340 ns*

Time to execute 100 instructions *= 100*340 = 34,000 ns*

Example 2

Given: Suppose we introduce pipelining on this machine. Assume that when introducing pipelining, the clock skew adds 5ns of overhead to each execution stage

Determine: instruction latency on the pipelined machine

How much time does it take to execute 100 instructions?

Solution to Example 2

Remember that in the pipelined implementation, the length of the pipe stages must all be the same, i.e., the speed of the slowest stage plus overhead. With 10 ns overhead it comes to:

***The length of pipelined stage** = MAX(lengths of unpipelined stages) + overhead = 60 + 10 = 70 ns*

***Instruction latency** = 70 ns*

***Time to execute 100 instructions** = 70*6*1 + 70*1*99 = 420 + 6930 = 7350 ns*

What is the speedup obtained from pipelining?

Solution: not considering any **stalls** introduced by different types of hazards

Average instruction time not pipelined *= 340 ns*

Average instruction time pipelined *= 70 ns*

Speedup *= 340 / 70 = 4.857*

Instruction sequences (Sidetrack)

Arithmetic Instruction

1. Fetch the instruction from memory;
2. Decode the instruction (it is an arithmetic instruction, but the CPU has to find that out through a decode operation);
3. Fetch the operands from the register file;
4. Apply the operands to the ALU;
5. Write the result back to the register file.

Branch Instructions

1. Fetch the instruction from memory;
2. Decode the instruction (it is a branch instruction);
3. Fetch the components of the address from the instruction or register file;
4. Apply the components of the address to the ALU (address arithmetic);
5. Copy the resulting effective address into the PC, thus accomplishing the branch.

Load and Store Instructions Sequence

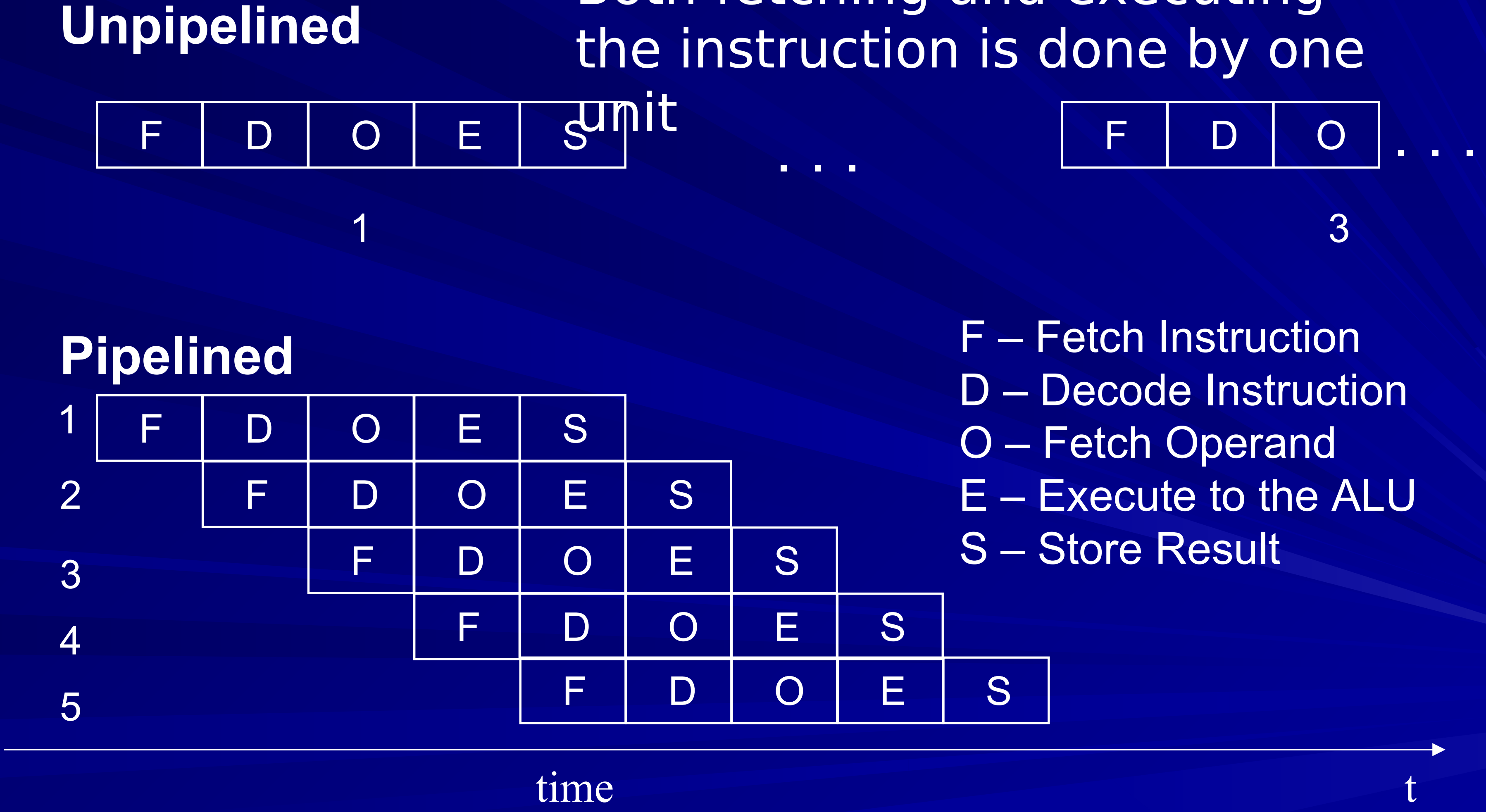
1. Fetch the instruction from memory;
2. Decode the instruction (it is a load or store instruction);
3. Fetch the components of the address from the instruction or register file;
4. Apply the components of the address to the ALU (address arithmetic);
5. Apply the resulting effective address to memory along with a read (load) or write (store) signal. If it is a write signal, the data item to be written must also be retrieved from the register file.

Frequency of occurrence of instruction types for a variety of languages. The percentages do not sum to 100 due to roundoff. (Adapted from [Knuth, 1991].)

Statement	**Average Percent of Time**
Assignment	47
If	23
Call	15
Loop	6
Goto	3
Other	7

Pipelined VS. Unpipelined

Both fetching and executing the instruction is done by one unit



F – Fetch Instruction
D – Decode Instruction
O – Fetch Operand
E – Execute to the ALU
S – Store Result

Hazards

- prevent the next instruction in the instruction stream from being executing during its designated clock cycle

- Three classes of hazards:
  - Structural hazards
  - Data hazards
  - Control hazards

Structural Hazard

- arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution

Common instances of structural hazards arise when :

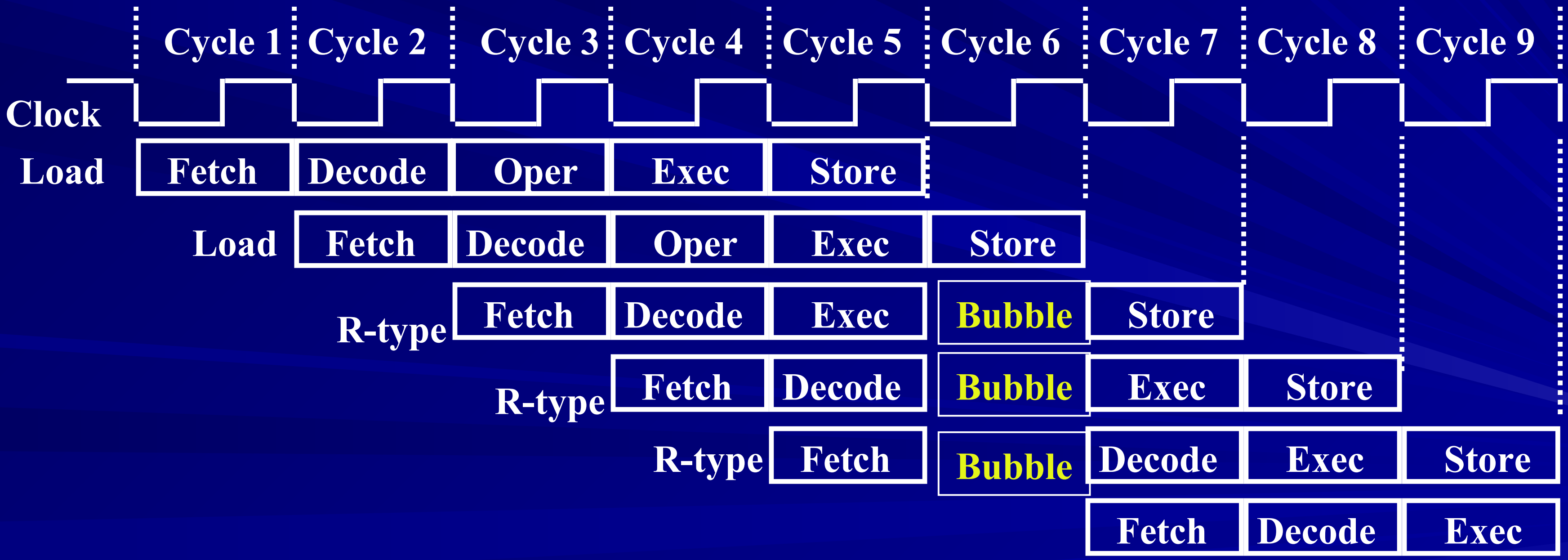
- Some functional unit is not fully pipelined - then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle
- Some resource has not been duplicated enough - to allow all combinations of instructions in the pipeline to execute

Solution of Structural Hazard

- stall the pipeline for one clock cycle when a data-memory access occurs (bubbles)
- Stall - prevent the succeeding instruction from doing its phase of the cycle
- allows the stalled instruction to proceed without conflict but defeats the purpose of overlapping the instructions for the stage

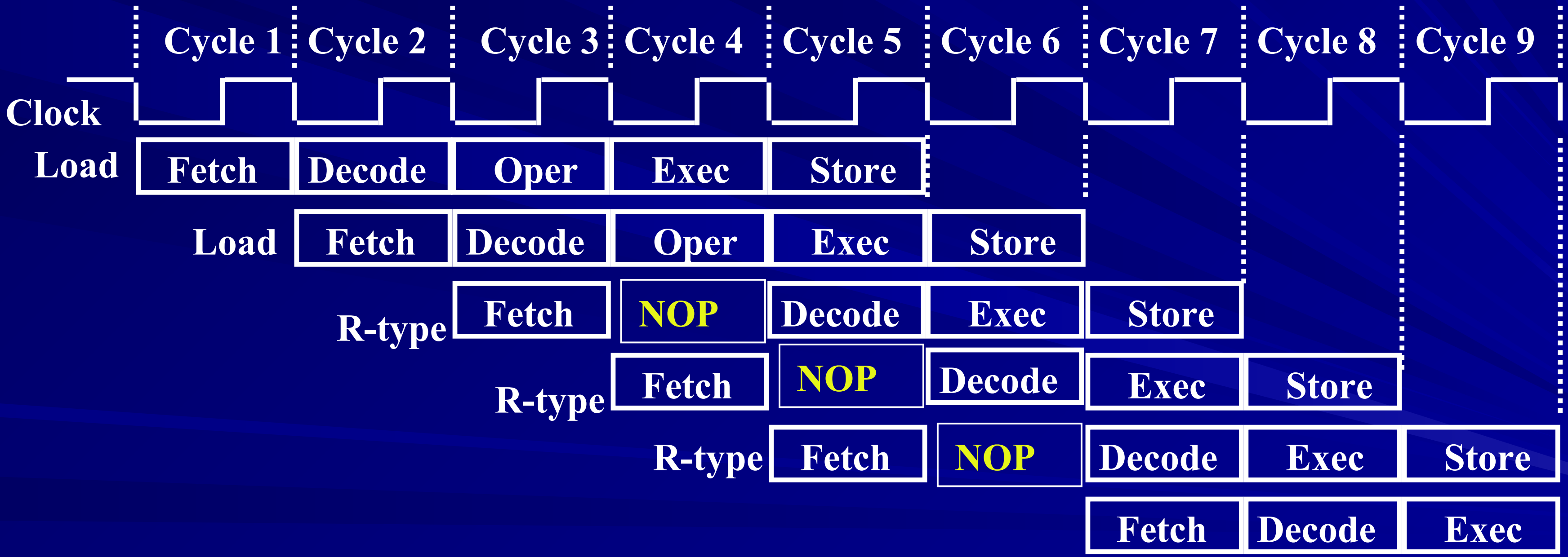
Pipeline Bubble

Insert a **bubble** into the pipeline to prevent two writes or stores at the same cycle



Delay Store Cycle

Add NOP (No Operation)



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
Load	Fetch	Decode	Oper	Exec	Store				
Load		Fetch	Decode	Oper	Exec	Store			
R-type			Fetch	NOP	Decode	Exec	Store		
R-type				Fetch	NOP	Decode	Exec	Store	
R-type					Fetch	NOP	Decode	Exec	Store
							Fetch	Decode	Exec

Data Hazard

- arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline

Data Hazard Classification

Consider two instructions ***i*** and ***j***, with ***i*** occurring before ***j***. The possible data hazards are :

- ***RAW (read after write)*** *- **j** tries to read a source before **i** writes it, so **j** incorrectly gets the old value*

- Solution: Forwarding ("Forward" result from one stage to another)
  - The ALU result from the Op Fetch /Exec register is always fed back to the ALU input latches
  - If the forwarding hardware detects that the previous ALU operation has written the register corresponding to the source for the current ALU operation, ***control logic*** selects the forwarded result as the ALU input rather than the value read from the register file

Data Hazard Classification

Consider two instructions ***i*** and ***j***, with ***i*** occurring before ***j***. The possible data hazards are :

- ***WAW (write after write)*** *-* ***j*** *tries to write an operand before it is written by* ***i.*** *The writes end up being performed in the wrong order, leaving the value written by* ***i*** *rather than the value written by* ***j*** *in the destination*
- Solution: Stall

Data Hazard Classification

Consider two instructions ***i*** and ***j***, with ***i*** occurring before ***j***. The possible data hazards are :

- ***WAR (write after read)*** *- **j** tries to write a destination before it is read by **i** , so **i** incorrectly gets the new value*
- Solution: Stall

Control Hazard

- arise from the pipelining of branches and other instructions that change the PC

- Solution: Stall the pipeline as soon as the branch is detected - Program Counter value changed to target address
  - Compute target address in advance

References

- David Patterson and John L. Hennessy. Computer Architecture: A Quantitative Approach. 2nd Edition. 1996.
- William Stallings. Computer Architecture and Organization. 2000
- Andrie Coronel and Ariel Maguyon. Computer Architecture slides. First Sem. SY 2006-2007, Ateneo de Manila University