Unit-2

Part-2

# PROCESSOR ORGANIZATION

1

## Processor Organization

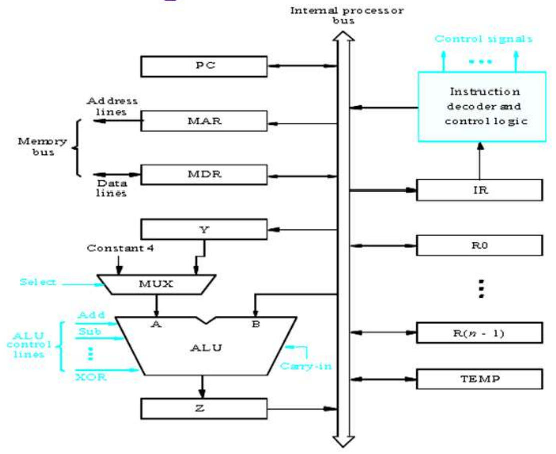
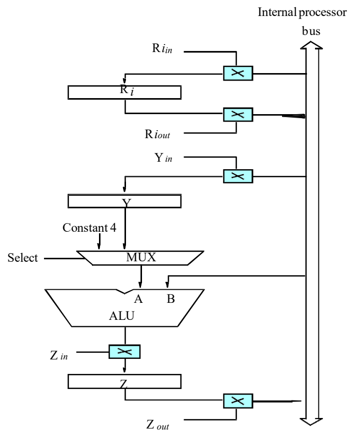


Figure 7.1. Single-bus organization of the datapath inside a processor.

2

## Register Transfers (MOV R2,R1)



Input and output gating for the registers

- The data transfer between registers and common bus is shown by a line with arrow heads.
- But in actual practice each register has input and output gating and these gates are controlled by corresponding control signal.
- $Ri_{in}$ and $Ri_{out}$ : control signals for input and output gating of register Ri.
- When $Ri_{in}$=1, the data available on the common data bus is loaded into register Ri.
- When $Ri_{out}$=1, the contents of Ri are placed on the common data bus.
- The signals $Ri_{in}$ and $Ri_{out}$ are commonly known as input enable and output enable signals of registers, respectively.

3

## Register Transfers

Consider the transfer of data from register R1 to R2. This can be done as follows:

1. Activate $R1_{out}$=1, this signal places the contents of register R1 on the common bus.
2. Activate $R2_{in}$=1, this loads data from the common data bus into the register R2.

4

## Micro Operations

- The primary function of a processor unit is to execute sequence of instructions stored in a memory.
- The sequence of operation involved in processing an instruction constitutes an instruction cycle, which can be divided into three major phases: fetch cycle, decode cycle and execute cycle.
- To perform fetch, decode and execute cycle the processor unit has to perform a set of operations called **Micro-operations** .

## Micro-operation includes:

- Perform an arithmetic or a logic operation on the data from CPU register and store the result in a CPU register.
- Transfer a word of data from one CPU register to another or to the ALU.
- Fetch the contents of a given memory location and load them into a CPU register.
- Store a word of data from a CPU register into a given memory location.

## 1. Performing an Arithmetic or Logic Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- The sequence of operations to add the contents of register R1 to those of R2 and store the result in R3.

Control Signals:

1. $R1_{out}$, $Y_{in}$
2. $R2_{out}$, Select Y, Add, $Z_{in}$
3. $Z_{out}$, $R3_{in}$

7

## 2. Fetching a Word from Memory

- The Processor loads required Address into MAR; at the same time it issue Read signal
- When the requested data is recieved from the memory it is stored into MDR.

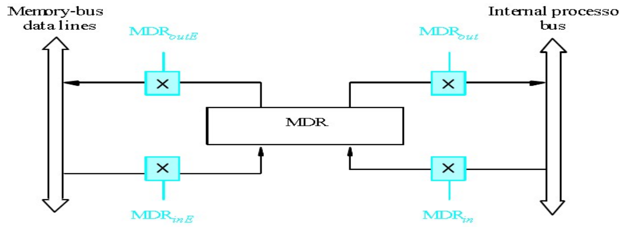
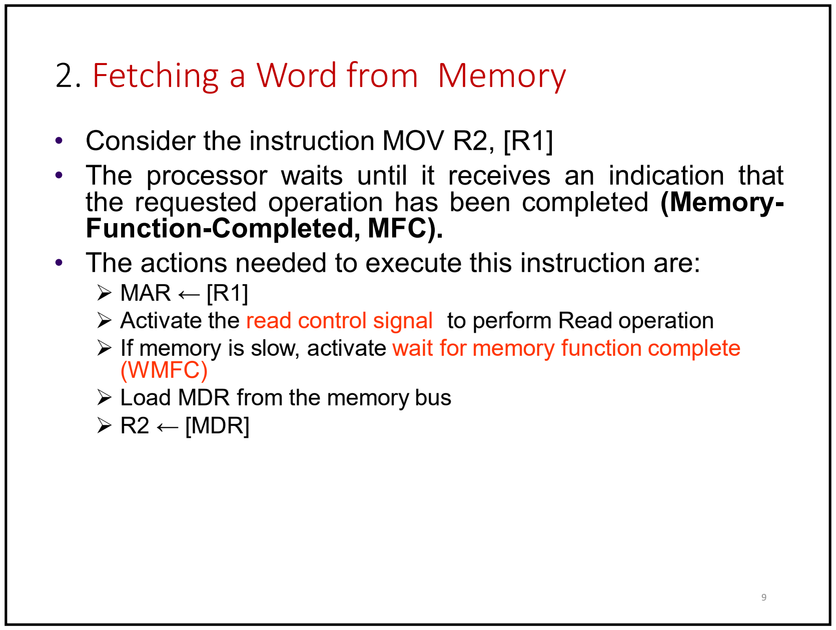


Figure 7.4. Connection and control signals for register MDR.

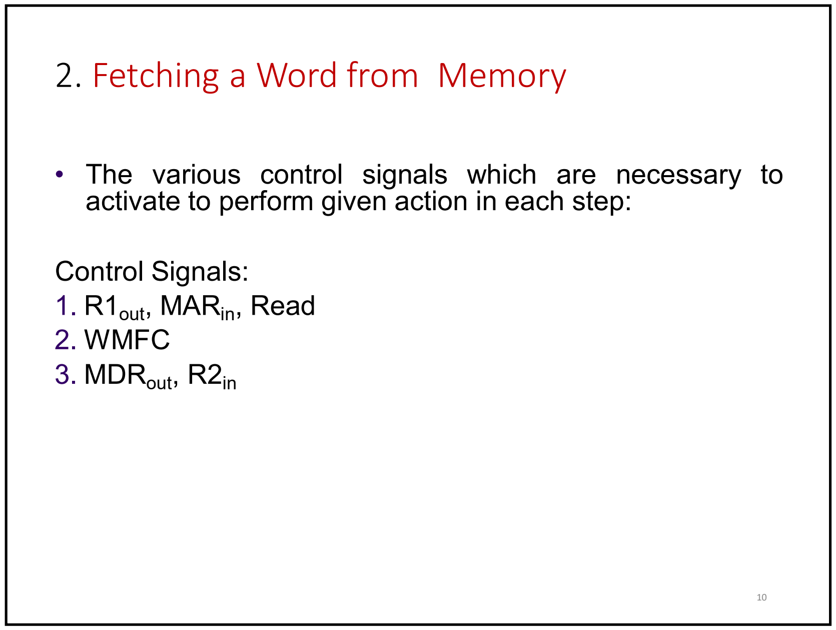
8



## 2. Fetching a Word from Memory

- Consider the instruction MOV R2, [R1]
- The processor waits until it receives an indication that the requested operation has been completed **(Memory-Function-Completed, MFC).**
- The actions needed to execute this instruction are:
  - ➢ MAR ← [R1]
  - ➢ Activate the read control signal to perform Read operation
  - ➢ If memory is slow, activate wait for memory function complete (WMFC)
  - ➢ Load MDR from the memory bus
  - ➢ R2 ← [MDR]

9



## 2. Fetching a Word from Memory

- The various control signals which are necessary to activate to perform given action in each step:

Control Signals:

1. $R1_{out}$, $MAR_{in}$, Read
2. WMFC
3. $MDR_{out}$, $R2_{in}$

10

## 3. Storing A Word In Memory:

- To write a word of data into a memory location, processor has to load the address of the desired memory location in the MAR, load the data to be written in memory in MDR and activate Write control signal.
- Consider the instruction MOVE [R2],R1.
- The actions needed to execute this instruction are:
  - MAR ← [R2]
  - MDR ← [R1]
  - Activate the write control signal to perform write operation
  - If memory is slow, activate wait for memory function complete (WMFC)

## 3. Storing A Word In Memory:

- The various control signals which are necessary to activate to perform given action in each step:

Control Signals:

1. $R2_{out}$, $MAR_{in}$
2. $R1_{out}$, $MDR_{in}$, Write
3. WMFC

## Execution of a Complete Instruction

- **ADD R1, [R2]**

  This instruction adds the contents of register R1 and the content of memory location specified by register R2 and store result in register R1

1. Fetch the instruction
2. Fetch the first operand (the contents of the memory location pointed to by R2)
3. Perform the addition
4. Load the result into R1

13

## Execution of a Complete Instruction

- Control Signals:

1. $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC

**ADD R1,[R2]**

**Micro Instructions:**

**Instruction Fetch(1-3):**

MAR ← PC
MDR ← M(MAR)
PC ← PC+1
IR ← MDR (opcode)

**Operand Fetch(4):**

MAR ← R2
MDR ← M(MAR)

**Execute Cycle(5-7):**

Y ← MDR
Z ← R1+Y
R1 ← Z

14

## Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.

15

## Execution of Branch Instructions

The control sequence for unconditional branch instruction is as follows:

1. $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC
3. $MDR_{out}$, $IR_{in}$

   The first three steps constitute the opcode fetch operation
4. Offset_field_of _$IR_{out}$, Select Y, Add, $Z_{in}$

   The contents of PC and the offset field of IR register are added and result is stored in register Z.
5. $Z_{out}$, $PC_{in}$, End

16

## Execution of Branch Instructions

- In case of conditional branch instruction the status of the condition code specified in the instruction is checked.
- If the status specified within the instruction matches with the current status of condition codes, the branch target address is loaded in the PC by adding the offset specified in the instruction to the contents of PC
- Otherwise processor fetches the next instruction in the sequence.

17

## Question

- Write control sequence with micro-program for

  **ADD R1,R2**

  including the instruction fetch phase? (Assume single bus architecture)
- Write control sequence for

  **SUB [R3]+,R1**

  Hint:

  R1 = R1 – (R3)

  R3 = R3 + 1

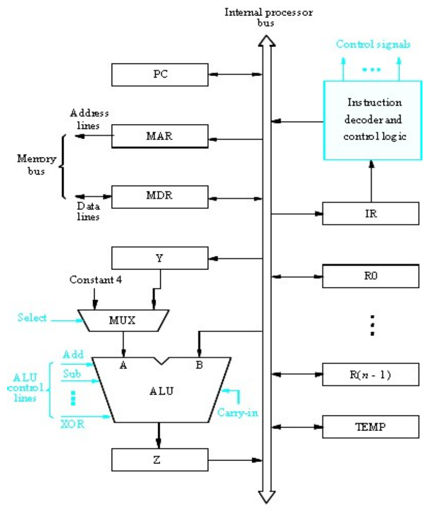
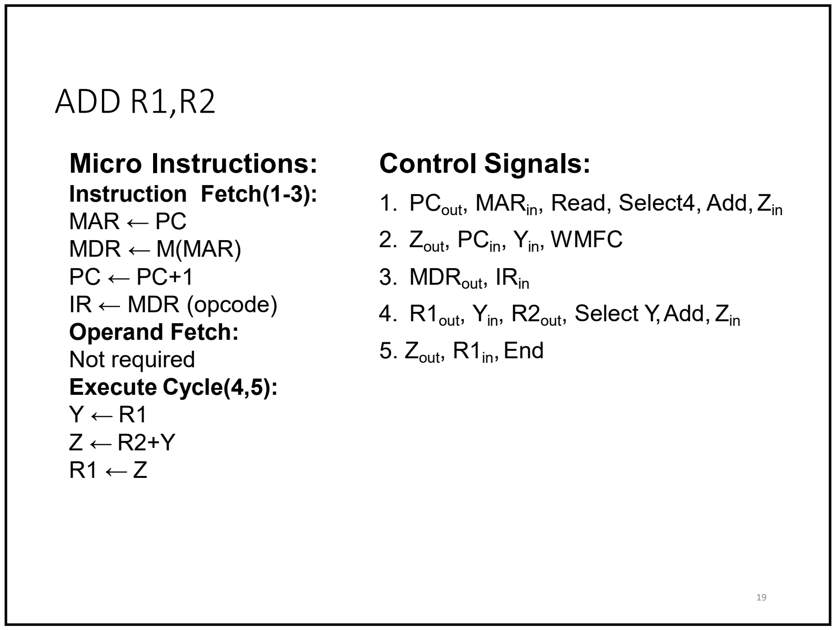


Figure 7.1. Single-bus organization of the datapath inside a processor.



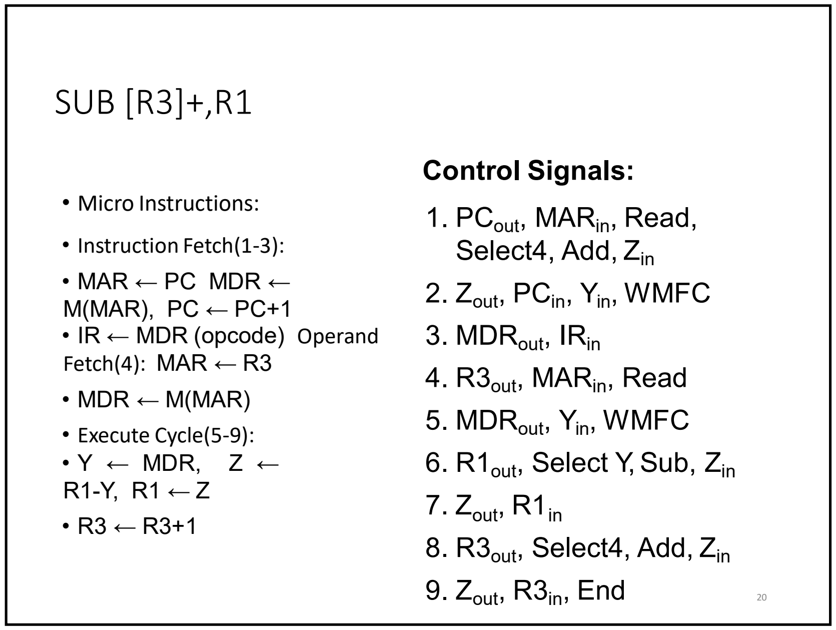
## ADD R1,R2

**Micro Instructions:**
**Instruction Fetch(1-3):**
MAR ← PC
MDR ← M(MAR)
PC ← PC+1
IR ← MDR (opcode)
**Operand Fetch:**
Not required
**Execute Cycle(4,5):**
Y ← R1
Z ← R2+Y
R1 ← Z

**Control Signals:**

1. $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC
3. $MDR_{out}$, $IR_{in}$
4. $R1_{out}$, $Y_{in}$, $R2_{out}$, Select Y, Add, $Z_{in}$
5. $Z_{out}$, $R1_{in}$, End

19



## SUB [R3]+,R1

- Micro Instructions:
- Instruction Fetch(1-3):
- MAR ← PC MDR ← M(MAR), PC ← PC+1
- IR ← MDR (opcode) Operand Fetch(4): MAR ← R3
- MDR ← M(MAR)
- Execute Cycle(5-9):
- Y ← MDR, Z ← R1-Y, R1 ← Z
- R3 ← R3+1

**Control Signals:**

1. $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC
3. $MDR_{out}$, $IR_{in}$
4. $R3_{out}$, $MAR_{in}$, Read
5. $MDR_{out}$, $Y_{in}$, WMFC
6. $R1_{out}$, Select Y, Sub, $Z_{in}$
7. $Z_{out}$, $R1_{in}$
8. $R3_{out}$, Select4, Add, $Z_{in}$
9. $Z_{out}$, $R3_{in}$, End

20

## Arithmetic Logic Unit (ALU)

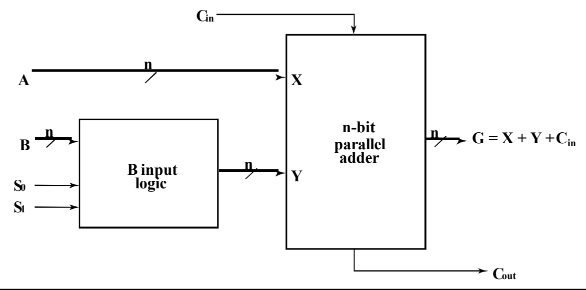
- In this and the next section, we deal with detailed design of typical ALUs and shifters
- Decompose the ALU into:
  - An arithmetic circuit
  - A logic circuit
  - A selector to pick between the two circuits
- Arithmetic circuit design
  - Decompose the arithmetic circuit into:
    - An n-bit parallel adder
    - A block of logic that selects four choices for the B input to the adder

21

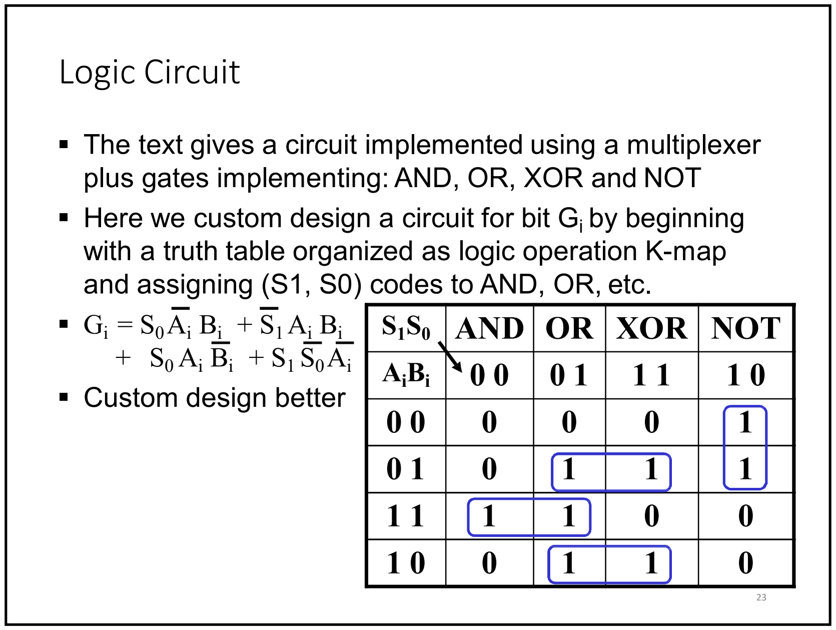
## Arithmetic Circuit Design

- **There are only four functions of B to select as Y in G = A + Y:**

| | $C_{in} = 0$ | Operation | $C_{in} = 1$ | Operation |
|---|---|---|---|---|
| • All 0's | G = A | Transfer of A | G = A + 1 | Increment |
| • B | G = A + B | Addition | G = A + B + 1 | |
| • $\overline{B}$ | G = A + $\overline{B}$ | Subtraction 1C | G = A + $\overline{B}$ + 1 | Subtraction 2C |
| • All 1's | G = A – 1 | Decrement | G = A | Transfer of A |



22

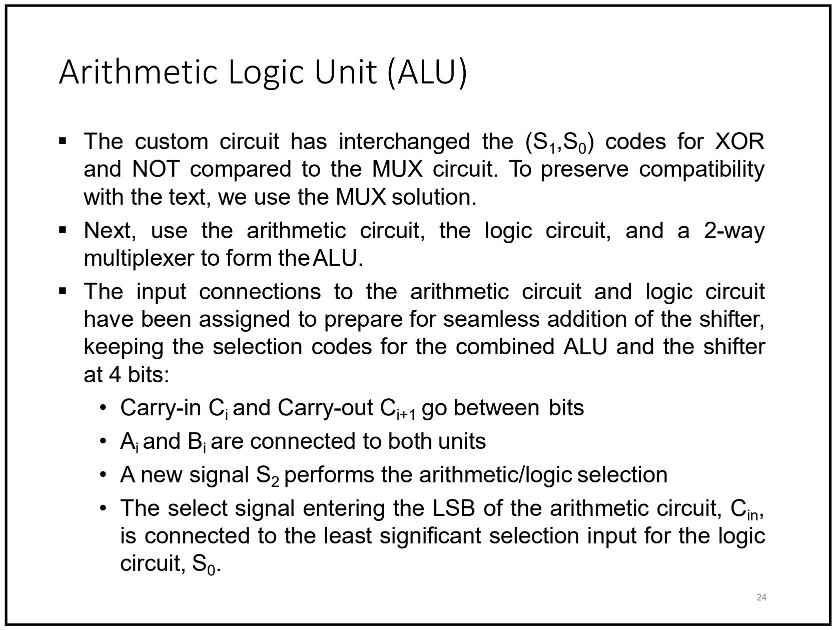


## Logic Circuit

- The text gives a circuit implemented using a multiplexer plus gates implementing: AND, OR, XOR and NOT
- Here we custom design a circuit for bit $G_i$ by beginning with a truth table organized as logic operation K-map and assigning (S1, S0) codes to AND, OR, etc.
- $G_i = S_0 \overline{A_i} B_i + \overline{S_1} A_i B_i + S_0 A_i \overline{B_i} + S_1 \overline{S_0} \overline{A_i}$
- Custom design better

| $S_1S_0$ \ $A_iB_i$ | AND<br>0 0 | OR<br>0 1 | XOR<br>1 1 | NOT<br>1 0 |
|---|---|---|---|---|
| 0 0 | 0 | 0 | 0 | 1 |
| 0 1 | 0 | 1 | 1 | 1 |
| 1 1 | 1 | 1 | 0 | 0 |
| 1 0 | 0 | 1 | 1 | 0 |

23

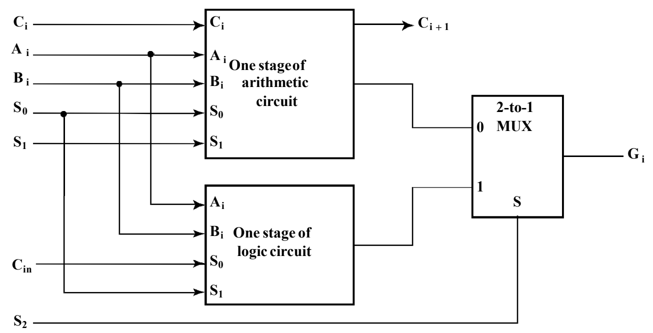


## Arithmetic Logic Unit (ALU)

- The custom circuit has interchanged the $(S_1,S_0)$ codes for XOR and NOT compared to the MUX circuit. To preserve compatibility with the text, we use the MUX solution.
- Next, use the arithmetic circuit, the logic circuit, and a 2-way multiplexer to form the ALU.
- The input connections to the arithmetic circuit and logic circuit have been assigned to prepare for seamless addition of the shifter, keeping the selection codes for the combined ALU and the shifter at 4 bits:
  - Carry-in $C_i$ and Carry-out $C_{i+1}$ go between bits
  - $A_i$ and $B_i$ are connected to both units
  - A new signal $S_2$ performs the arithmetic/logic selection
  - The select signal entering the LSB of the arithmetic circuit, $C_{in}$, is connected to the least significant selection input for the logic circuit, $S_0$.

24

## Arithmetic Logic Unit (ALU)

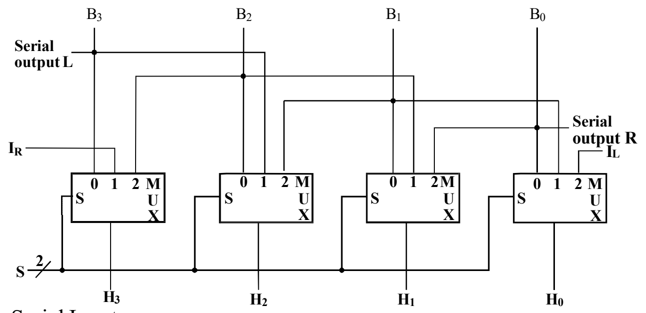


- The next most significant select signals, $S_0$ for the arithmetic circuit and $S_1$ for the logic circuit, are wired together, completing the two select signals for the logic circuit.
- The remaining $S_1$ completes the three select signals for the arithmetic circuit.

## Shifters

- Required for data processing, multiplication, division etc.
- Direction: Left, Right
- Number of positions with examples:
  - Single bit:
    - 1 position
    - 0 and 1 positions
  - Multiple bit:
    - 1 to n – 1 positions
    - 0 to n – 1 positions
- Filling of vacant positions
  - Many options depending on instruction set
  - Here, will provide input lines or zero fill

## 4-Bit Basic Left/Right Shifter



- Serial Inputs:
  - $I_R$ for right shift
  - $I_L$ for left shift
- Serial Outputs
  - R for right shift (Same as MSB input)
  - L for left shift (Same as LSB input)
- Shift Functions:

| | | |
|---|---|---|
| $(S_1, S_0) =$ | 00 | Pass B unchanged |
| | 01 | Right shift |
| | 10 | Left shift |
| | 11 | Unused |

27

## What is Pipelining?

- A technique used in advanced microprocessors where the microprocessor begins executing a second instruction before the first has been completed.
- A Pipeline is a series of stages, where some work is done at each stage. The work is not finished until it has passed through all stages.
- With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor until each instruction operation can performed.

28

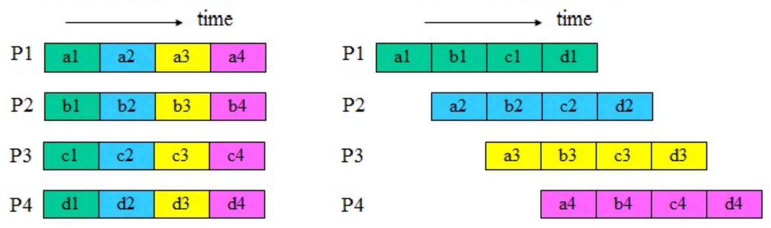
## How Pipeline Works?

- The pipeline is divided into segments and each segment can execute its operation concurrently with the other segments.
- Once a segment completes an operation, it passes the result to the next segment in the pipeline and fetches the next operations from the preceding segment.

29

## Basic Ideas

- Parallel processing
- Pipelined processing



Less inter-processor communication
Complicated processor hardware

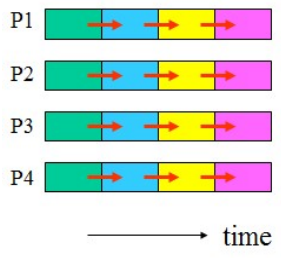
More inter-processor communication
Simpler processor hardware

| | |
|---|---|
| Colors: | different types of operations performed |
| a, b, c, d: | different data streams processed |

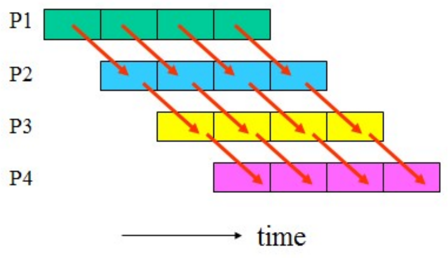
30

## Data Dependence

- Parallel processing requires NO data dependence between processors



- Pipelined processing will involve inter-processor communication

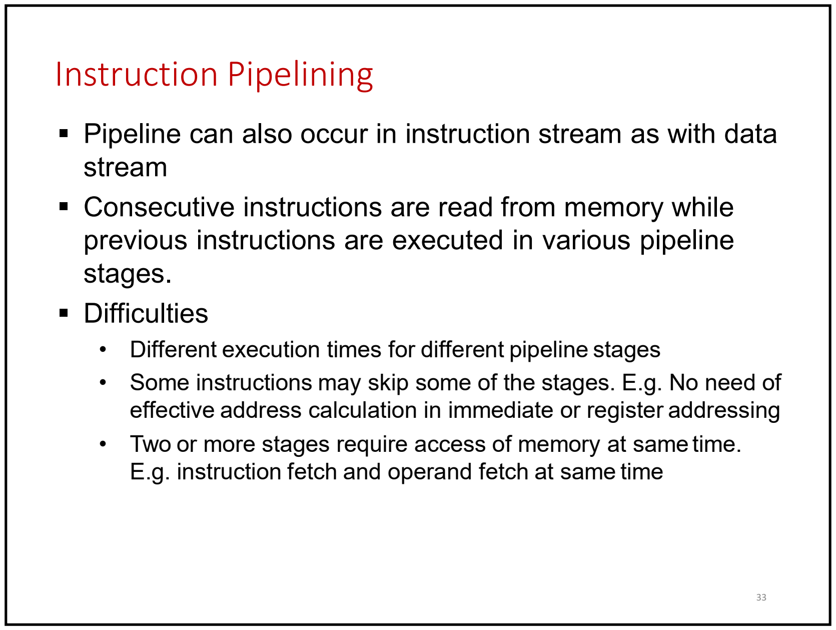


31

## Advantages/Disadvantages

- Advantages:
  - More efficient use of processor
  - Quicker time of execution of large number of instructions
- Disadvantages:
  - Pipelining involves adding hardware to the chip
  - Inability to continuously run the pipeline at full speed because of pipeline hazards which disrupt the smooth execution of the pipeline.

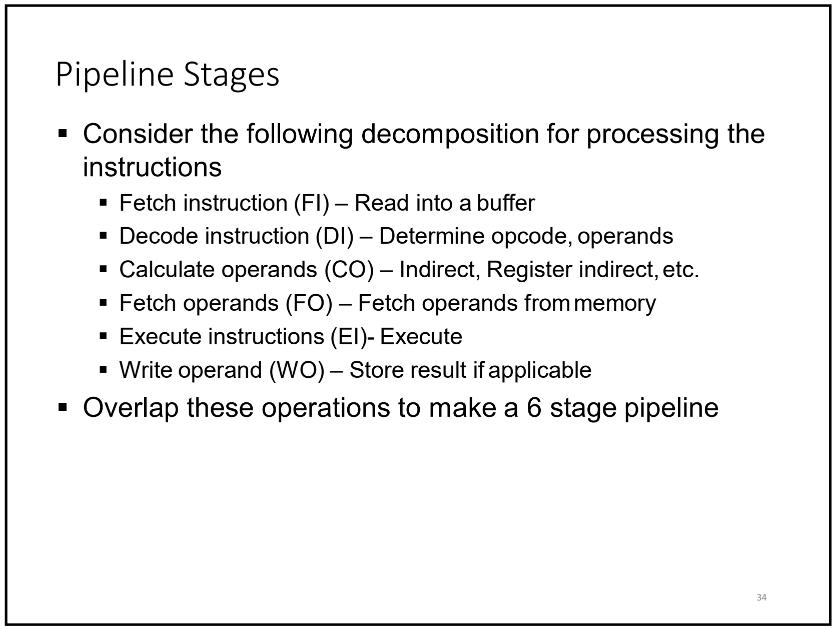
32



## Instruction Pipelining

- Pipeline can also occur in instruction stream as with data stream
- Consecutive instructions are read from memory while previous instructions are executed in various pipeline stages.
- Difficulties
  - Different execution times for different pipeline stages
  - Some instructions may skip some of the stages. E.g. No need of effective address calculation in immediate or register addressing
  - Two or more stages require access of memory at same time. E.g. instruction fetch and operand fetch at same time

33

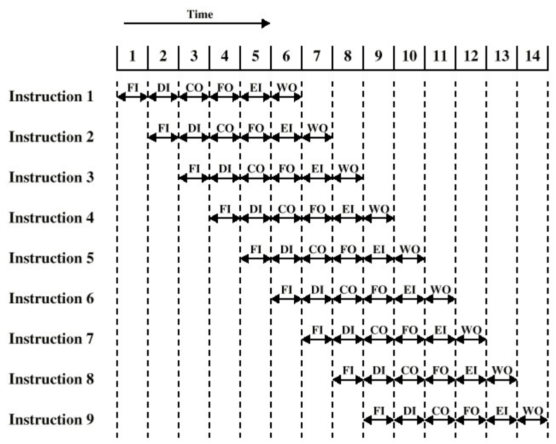


## Pipeline Stages

- Consider the following decomposition for processing the instructions
  - Fetch instruction (FI) – Read into a buffer
  - Decode instruction (DI) – Determine opcode, operands
  - Calculate operands (CO) – Indirect, Register indirect, etc.
  - Fetch operands (FO) – Fetch operands from memory
  - Execute instructions (EI)- Execute
  - Write operand (WO) – Store result if applicable
- Overlap these operations to make a 6 stage pipeline

34

## Timing of Instruction Pipeline - six stages



Reduction in instruction execution time from 54 time units to 14 time units

## Data Dependencies

- When two instructions access the same register.
- **RAW:** Read-After-Write
  - True dependency
- **WAR:** Write-After-Read
  - Anti-dependency
- **WAW:** Write-After-Write
  - False-dependency
- Key problem with regular *in-order* pipelines is RAW.

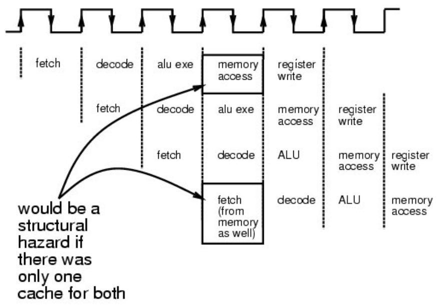
## Pipeline Hazards

- **Structural Hazard or Resource Conflict** – Two instructions need to access the same resource at same time.
- **Data Hazard or Data Dependency** – An instruction uses the result of the previous instruction before it is ready.
  - A hazard occurs exactly when an instruction tries to read a register in its ID stage that an earlier instruction intends to write in its WO stage.
- **Control Hazard or Branch Difficulty** – The location of an instruction depends on previous instruction
  - Conditional branches break the pipeline
    - Stuff we fetched in advance is useless if we take the branch
- Pipeline implementation need to detect and resolve hazards.

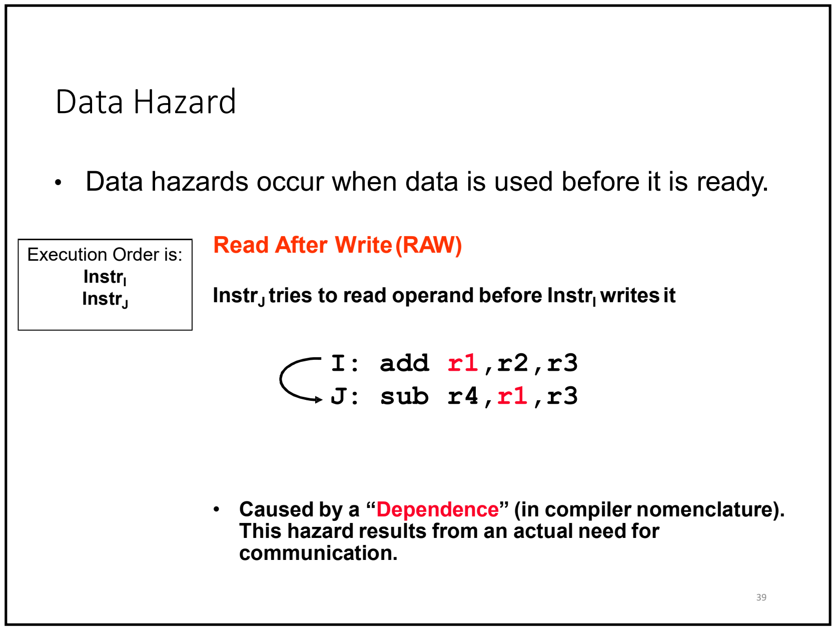
37

## Structural Hazard

- When IF stage requires memory access for instruction fetch, and MEM stage need memory access for operand fetch at the same time.
- Solutions:
  - Stalling (Waiting/Delaying)
    - Delayed by one clock cycle
  - Split cache
    - Separate cache for instructions(code cache) and operands(data cache)



38



## Data Hazard

- Data hazards occur when data is used before it is ready.

Execution Order is:
$\text{Instr}_I$
$\text{Instr}_J$

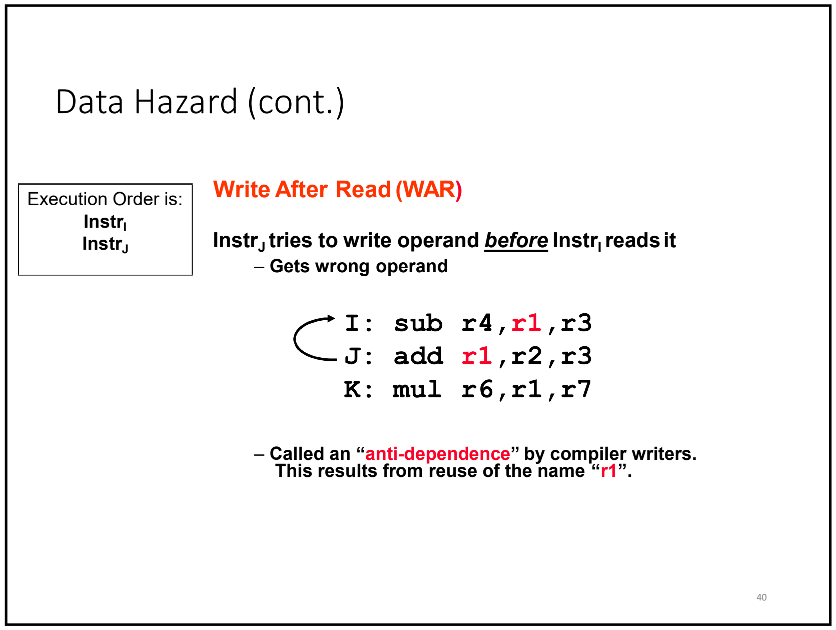
**Read After Write (RAW)**

**$\text{Instr}_J$ tries to read operand before $\text{Instr}_I$ writes it**

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- **Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.**

39



## Data Hazard (cont.)

Execution Order is:
$\text{Instr}_I$
$\text{Instr}_J$

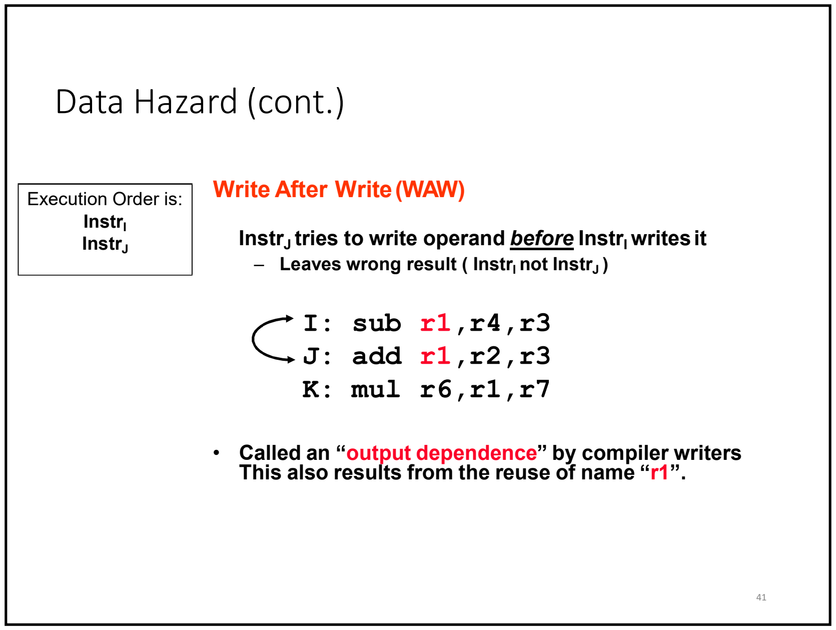
**Write After Read (WAR)**

**$\text{Instr}_J$ tries to write operand *before* $\text{Instr}_I$ reads it**
- **Gets wrong operand**

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".**

40



## Data Hazard (cont.)

Execution Order is:
**Instr$_I$**
**Instr$_J$**

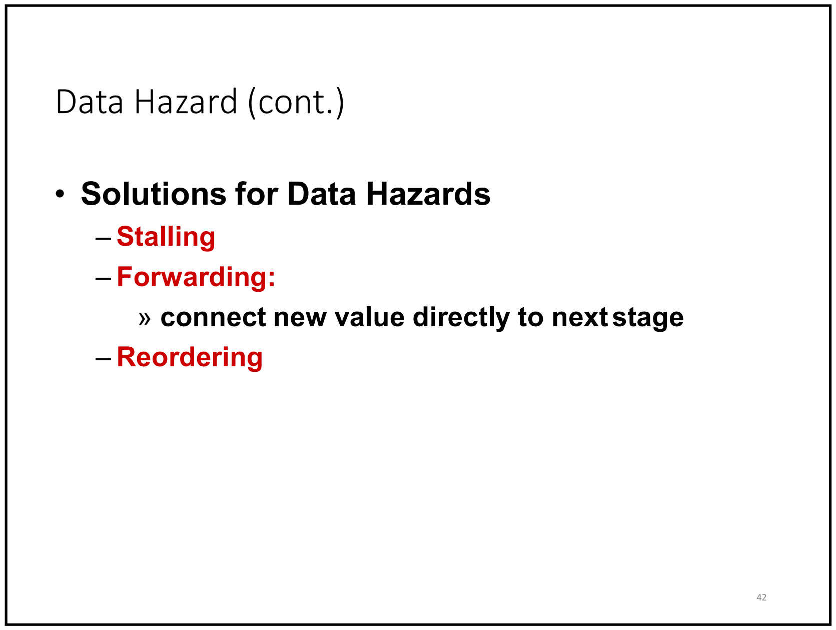
**Write After Write (WAW)**

**Instr$_J$ tries to write operand *before* Instr$_I$ writes it**
- **Leaves wrong result ( Instr$_I$ not Instr$_J$ )**

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an "output dependence" by compiler writers**
**This also results from the reuse of name "r1".**

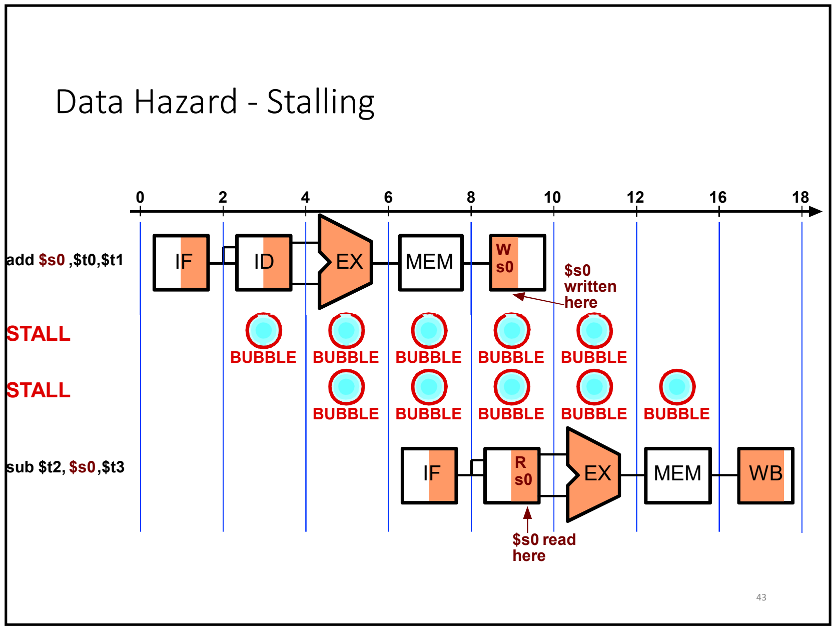
41



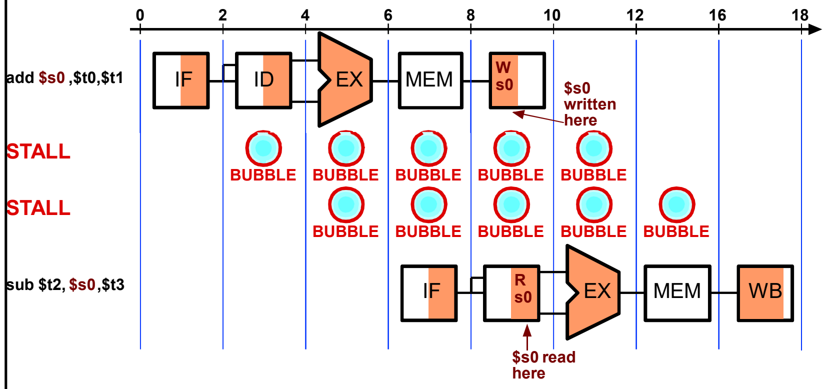
## Data Hazard (cont.)

- **Solutions for Data Hazards**
  - **Stalling**
  - **Forwarding:**
    - » **connect new value directly to next stage**
  - **Reordering**

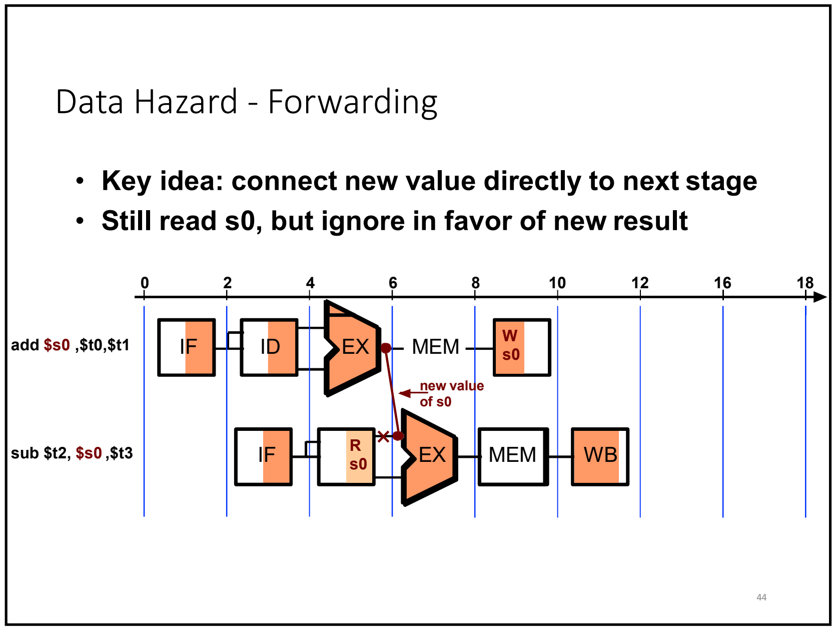
42



## Data Hazard - Stalling

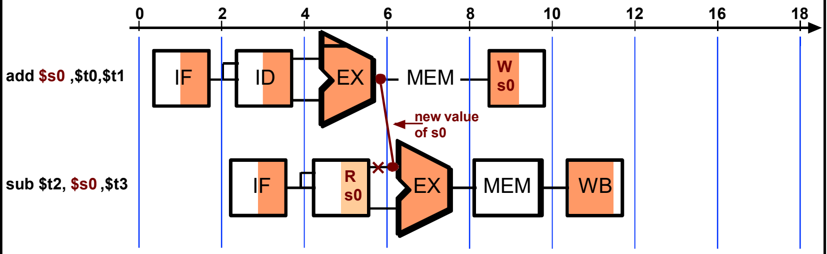


43

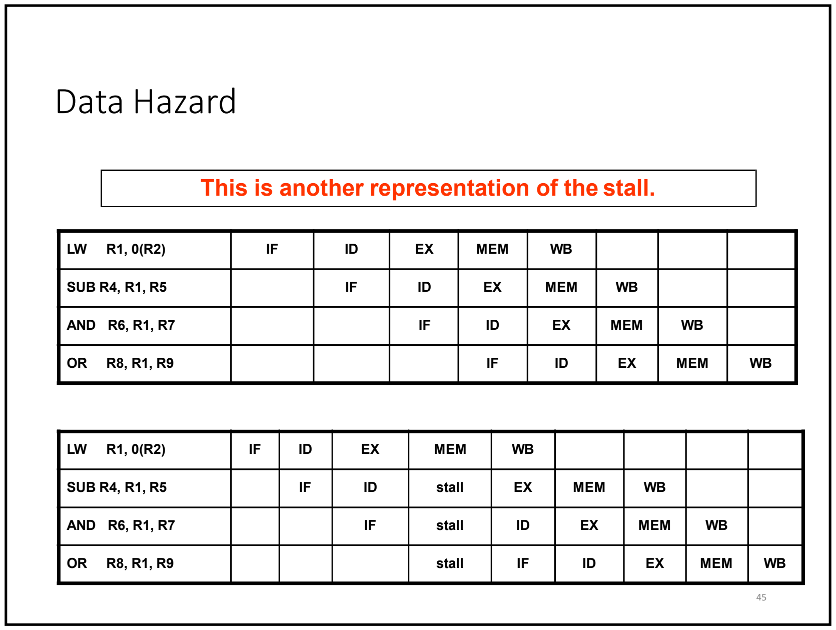


## Data Hazard - Forwarding

- Key idea: connect new value directly to next stage
- Still read s0, but ignore in favor of new result



44



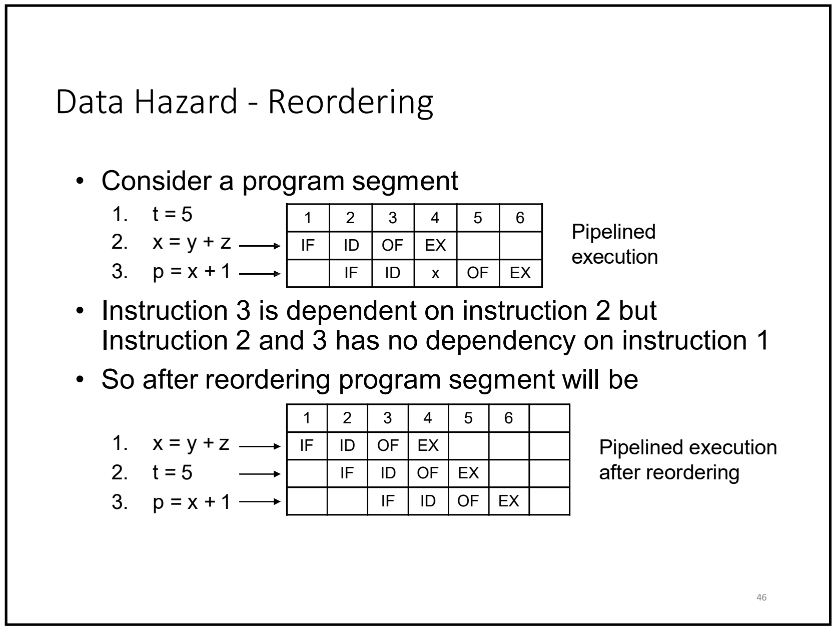
## Data Hazard

**This is another representation of the stall.**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **LW R1, 0(R2)** | **IF** | **ID** | **EX** | **MEM** | **WB** | | | |
| **SUB R4, R1, R5** | | **IF** | **ID** | **EX** | **MEM** | **WB** | | |
| **AND R6, R1, R7** | | | **IF** | **ID** | **EX** | **MEM** | **WB** | |
| **OR R8, R1, R9** | | | | **IF** | **ID** | **EX** | **MEM** | **WB** |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **LW R1, 0(R2)** | **IF** | **ID** | **EX** | **MEM** | **WB** | | | | |
| **SUB R4, R1, R5** | | **IF** | **ID** | **stall** | **EX** | **MEM** | **WB** | | |
| **AND R6, R1, R7** | | | **IF** | **stall** | **ID** | **EX** | **MEM** | **WB** | |
| **OR R8, R1, R9** | | | | **stall** | **IF** | **ID** | **EX** | **MEM** | **WB** |

45



## Data Hazard - Reordering

- Consider a program segment
  1. t = 5
  2. x = y + z
  3. p = x + 1

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| x = y + z | IF | ID | OF | EX | | |
| p = x + 1 | | IF | ID | x | OF | EX |

Pipelined execution

- Instruction 3 is dependent on instruction 2 but Instruction 2 and 3 has no dependency on instruction 1
- So after reordering program segment will be
  1. x = y + z
  2. t = 5
  3. p = x + 1

| | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|
| x = y + z | IF | ID | OF | EX | | | |
| t = 5 | | IF | ID | OF | EX | | |
| p = x + 1 | | | IF | ID | OF | EX | |

Pipelined execution after reordering

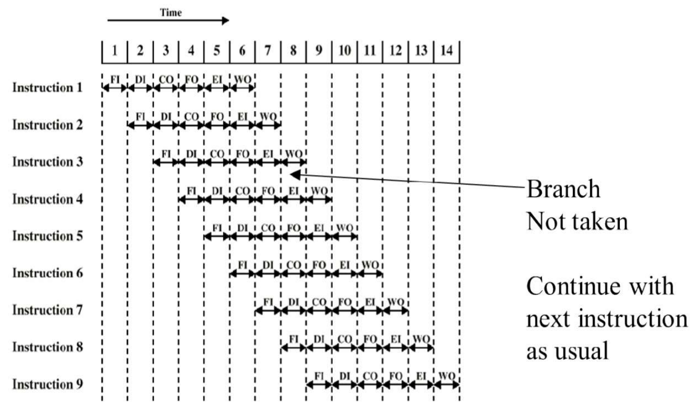
46

## Control Hazard

- Caused by branch instructions – unconditional and conditional branches.
  - Unconditional – branch always
  - Conditional – may or may not cause branching
- In pipelined processor following actions are critical:
  - Timely detection of a branch instruction
  - Early calculation of branch address
  - Early testing of branch condition for conditional branch instructions

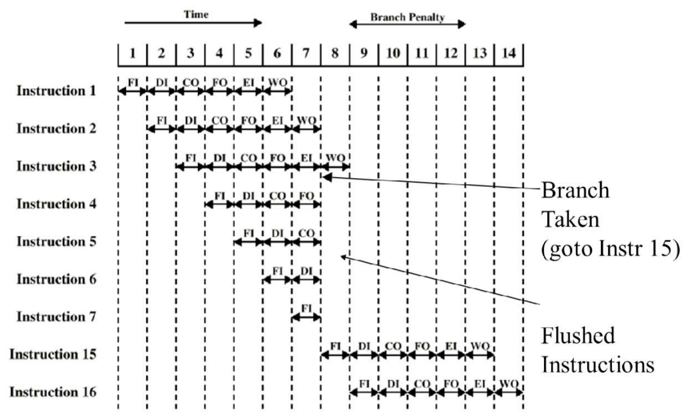
## Control Hazard (cont.)

- **Branch Not Taken**



## Control Hazard (cont.)

- **Branch in a Pipeline – Flushed Pipeline**

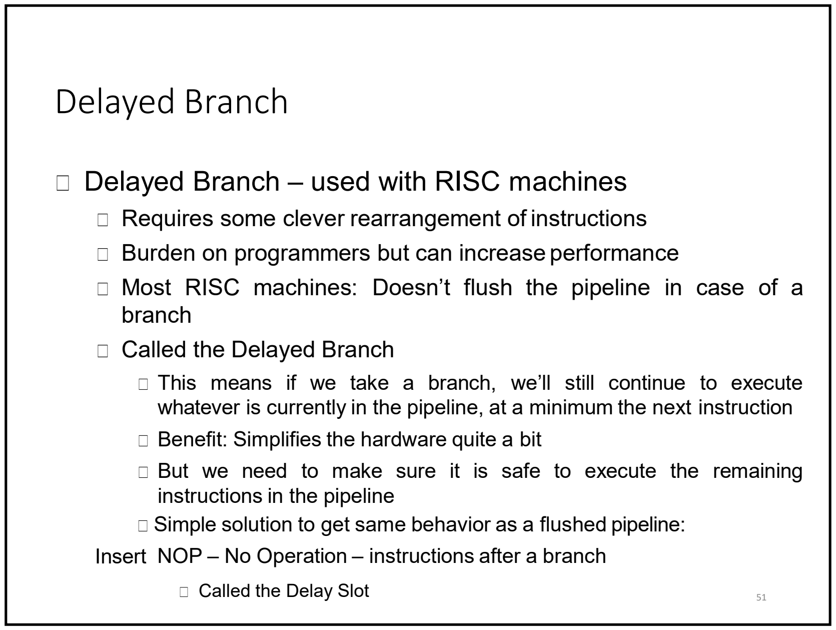


49

## Dealing with Branches

- Delayed branches
- Branch prediction
- Multiple Streams
- Prefetch Branch Target
- Loop buffer

50



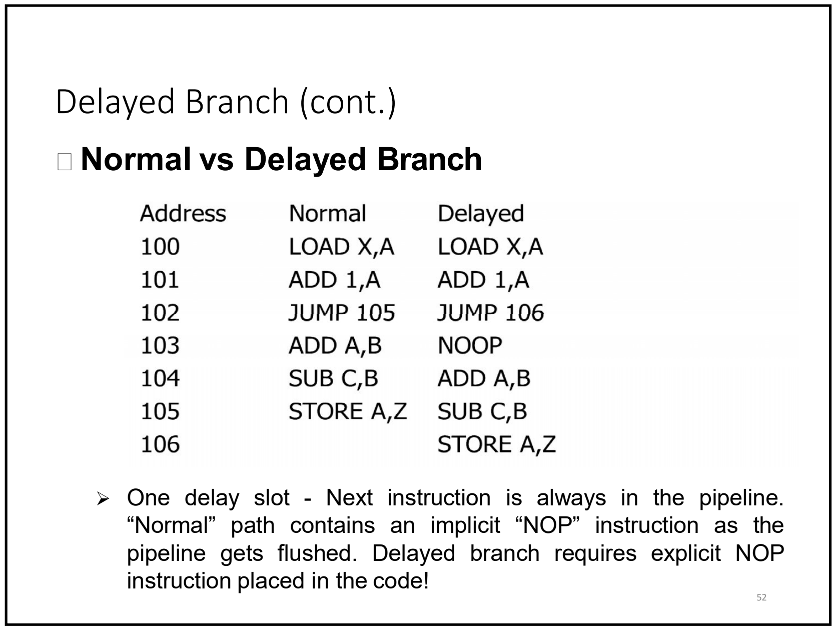
## Delayed Branch

- Delayed Branch – used with RISC machines
  - Requires some clever rearrangement of instructions
  - Burden on programmers but can increase performance
  - Most RISC machines: Doesn't flush the pipeline in case of a branch
  - Called the Delayed Branch
    - This means if we take a branch, we'll still continue to execute whatever is currently in the pipeline, at a minimum the next instruction
    - Benefit: Simplifies the hardware quite a bit
    - But we need to make sure it is safe to execute the remaining instructions in the pipeline
    - Simple solution to get same behavior as a flushed pipeline:

Insert NOP – No Operation – instructions after a branch

- Called the Delay Slot

51



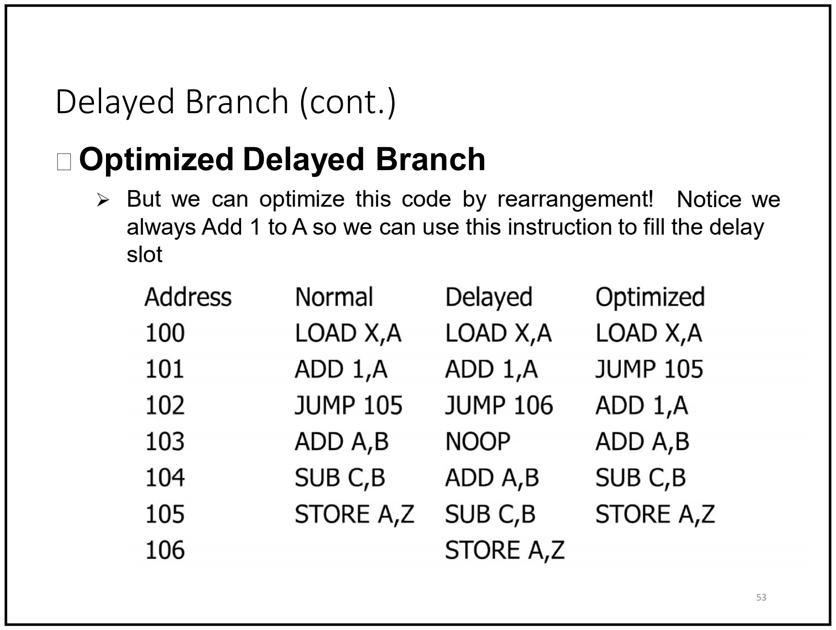
## Delayed Branch (cont.)

- **Normal vs Delayed Branch**

| Address | Normal | Delayed |
|---|---|---|
| 100 | LOAD X,A | LOAD X,A |
| 101 | ADD 1,A | ADD 1,A |
| 102 | JUMP 105 | JUMP *106* |
| 103 | ADD A,B | NOOP |
| 104 | SUB C,B | ADD A,B |
| 105 | STORE A,Z | SUB C,B |
| 106 | | STORE A,Z |

- One delay slot - Next instruction is always in the pipeline. "Normal" path contains an implicit "NOP" instruction as the pipeline gets flushed. Delayed branch requires explicit NOP instruction placed in the code!

52

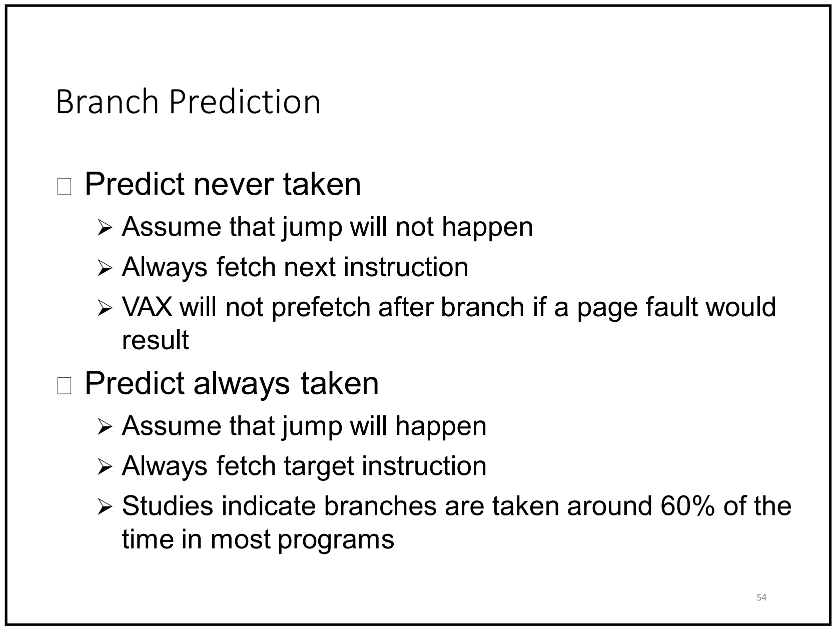


## Delayed Branch (cont.)

- **Optimized Delayed Branch**
  - But we can optimize this code by rearrangement! Notice we always Add 1 to A so we can use this instruction to fill the delay slot

| Address | Normal | Delayed | Optimized |
|---|---|---|---|
| 100 | LOAD X,A | LOAD X,A | LOAD X,A |
| 101 | ADD 1,A | ADD 1,A | JUMP 105 |
| 102 | JUMP 105 | JUMP 106 | ADD 1,A |
| 103 | ADD A,B | NOOP | ADD A,B |
| 104 | SUB C,B | ADD A,B | SUB C,B |
| 105 | STORE A,Z | SUB C,B | STORE A,Z |
| 106 | | STORE A,Z | |

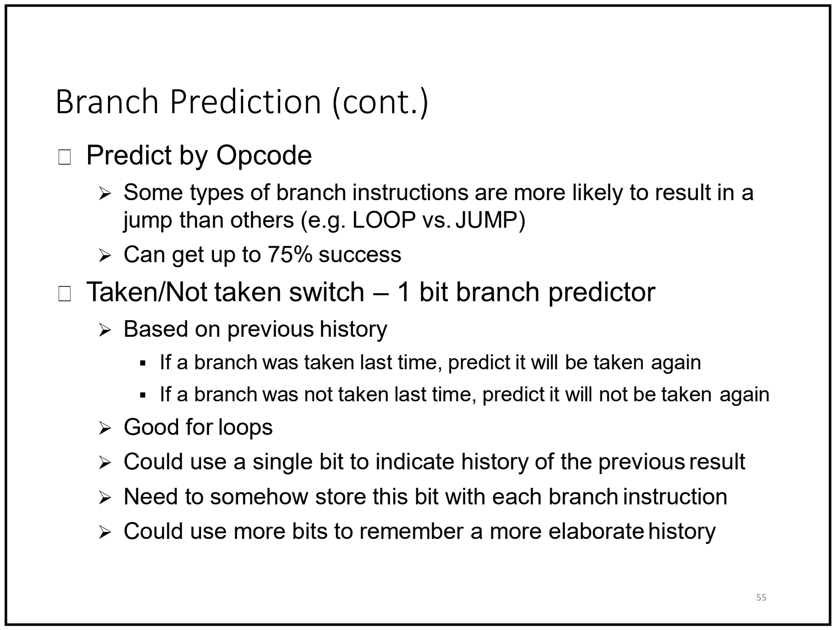
53



## Branch Prediction

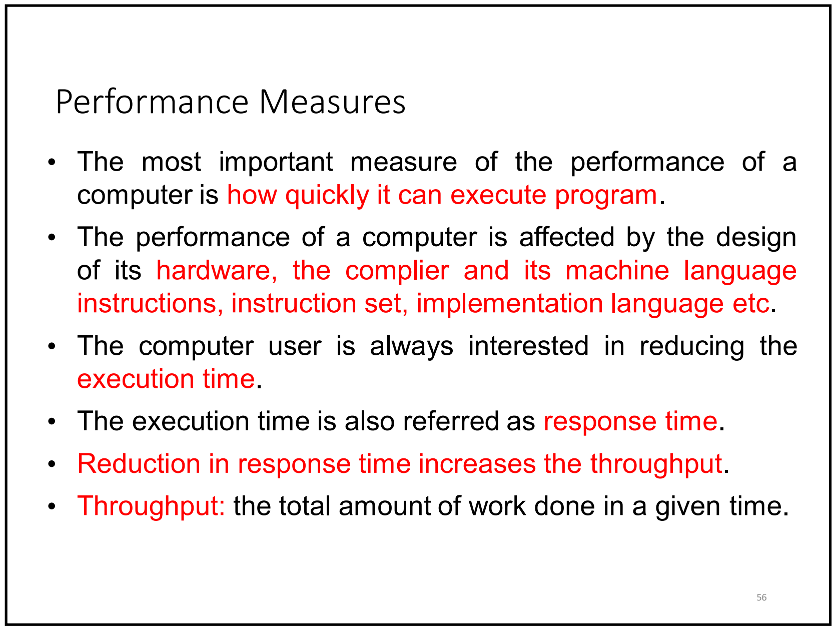
- Predict never taken
  - Assume that jump will not happen
  - Always fetch next instruction
  - VAX will not prefetch after branch if a page fault would result
- Predict always taken
  - Assume that jump will happen
  - Always fetch target instruction
  - Studies indicate branches are taken around 60% of the time in most programs

54



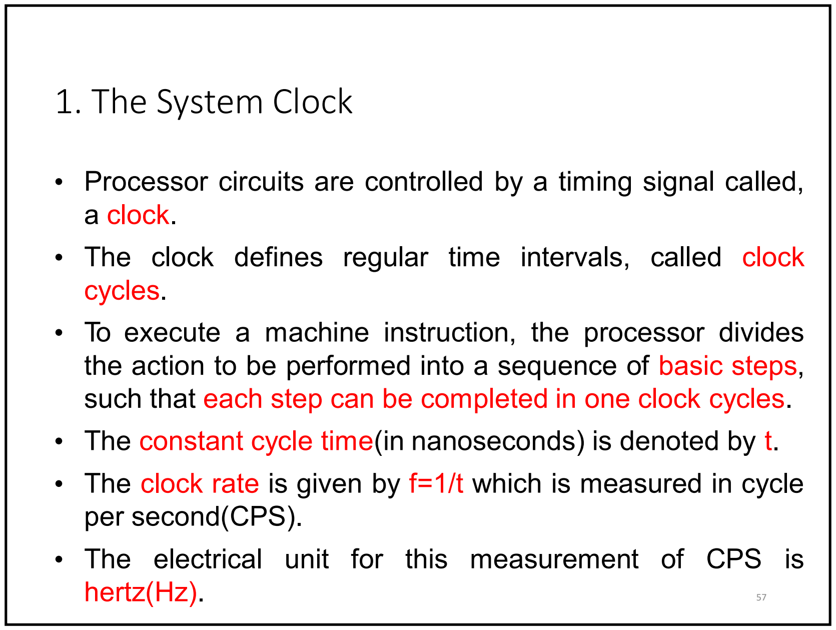
## Branch Prediction (cont.)

- Predict by Opcode
  - Some types of branch instructions are more likely to result in a jump than others (e.g. LOOP vs. JUMP)
  - Can get up to 75% success
- Taken/Not taken switch – 1 bit branch predictor
  - Based on previous history
    - If a branch was taken last time, predict it will be taken again
    - If a branch was not taken last time, predict it will not be taken again
  - Good for loops
  - Could use a single bit to indicate history of the previous result
  - Need to somehow store this bit with each branch instruction
  - Could use more bits to remember a more elaborate history



## Performance Measures

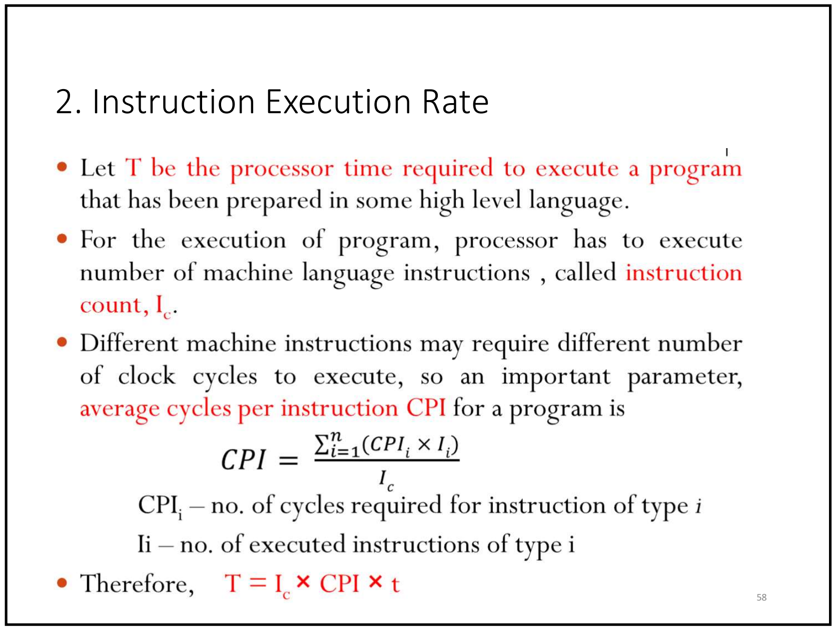
- The most important measure of the performance of a computer is how quickly it can execute program.
- The performance of a computer is affected by the design of its hardware, the complier and its machine language instructions, instruction set, implementation language etc.
- The computer user is always interested in reducing the execution time.
- The execution time is also referred as response time.
- Reduction in response time increases the throughput.
- Throughput: the total amount of work done in a given time.



## 1. The System Clock

- Processor circuits are controlled by a timing signal called, a clock.
- The clock defines regular time intervals, called clock cycles.
- To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps, such that each step can be completed in one clock cycles.
- The constant cycle time(in nanoseconds) is denoted by t.
- The clock rate is given by f=1/t which is measured in cycle per second(CPS).
- The electrical unit for this measurement of CPS is hertz(Hz).

57



## 2. Instruction Execution Rate

- Let T be the processor time required to execute a program that has been prepared in some high level language.
- For the execution of program, processor has to execute number of machine language instructions , called instruction count, $I_c$.
- Different machine instructions may require different number of clock cycles to execute, so an important parameter, average cycles per instruction CPI for a program is

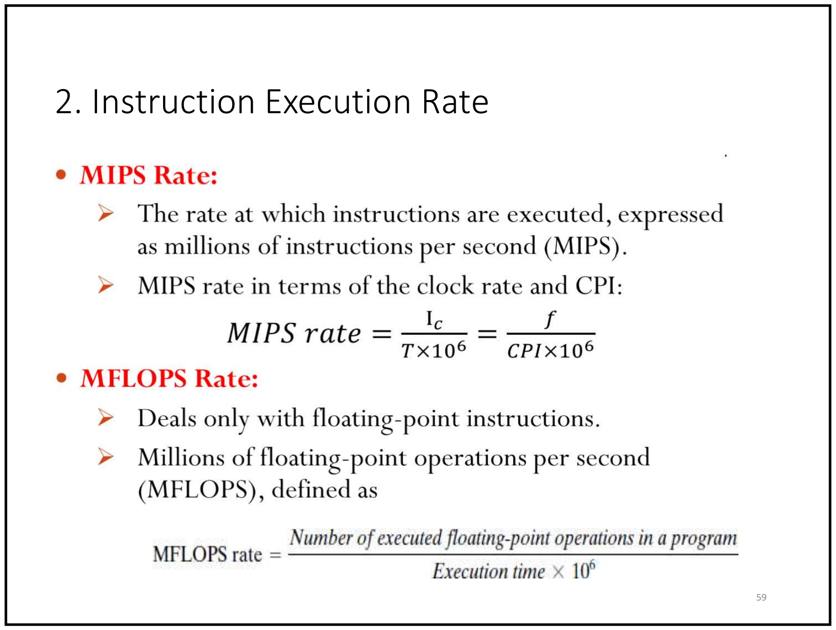
$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_c}$$

$CPI_i$ – no. of cycles required for instruction of type $i$

Ii – no. of executed instructions of type i

- Therefore, $T = I_c \times CPI \times t$

58



## 2. Instruction Execution Rate

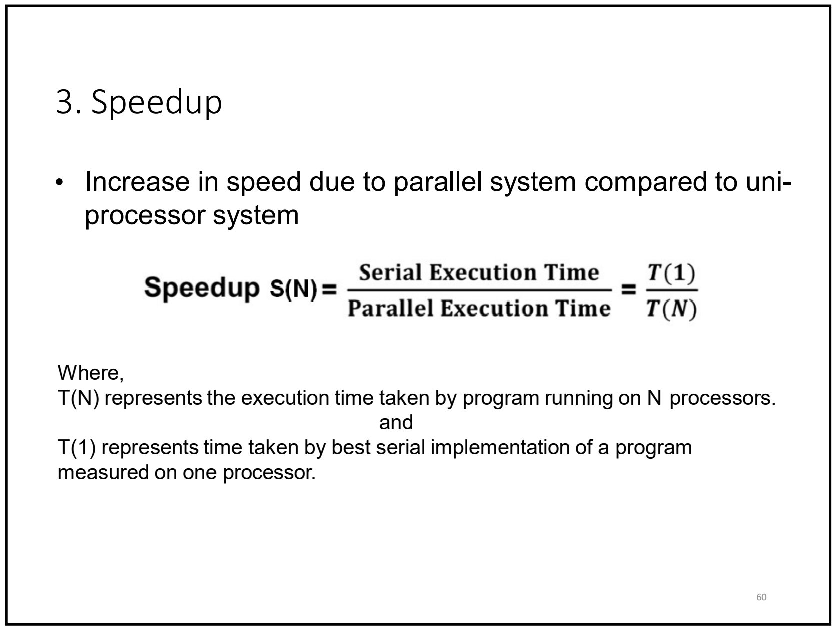
- **MIPS Rate:**
  - ➢ The rate at which instructions are executed, expressed as millions of instructions per second (MIPS).
  - ➢ MIPS rate in terms of the clock rate and CPI:

$$MIPS\ rate = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

- **MFLOPS Rate:**
  - ➢ Deals only with floating-point instructions.
  - ➢ Millions of floating-point operations per second (MFLOPS), defined as

$$\text{MFLOPS rate} = \frac{\textit{Number of executed floating-point operations in a program}}{\textit{Execution time} \times 10^6}$$

59



## 3. Speedup

- Increase in speed due to parallel system compared to uni-processor system

$$\textbf{Speedup } S(N) = \frac{\textbf{Serial Execution Time}}{\textbf{Parallel Execution Time}} = \frac{T(1)}{T(N)}$$

Where,
T(N) represents the execution time taken by program running on N processors.
and
T(1) represents time taken by best serial implementation of a program measured on one processor.

60

## 4. Efficiency

- The average contribution of the processors towards the global computation.
- It is defined as speedup divided by number of processors.

$$\textbf{Efficiency(N)} = \frac{\textbf{Speedup(N)}}{\textbf{N}}$$

Speedup(N) = speedup measured on N processors

## 5. Throughput ($\omega_p$)

- Number of programs a system can execute per unit time

$$\omega_p = \frac{f}{I_c \times CPI}$$

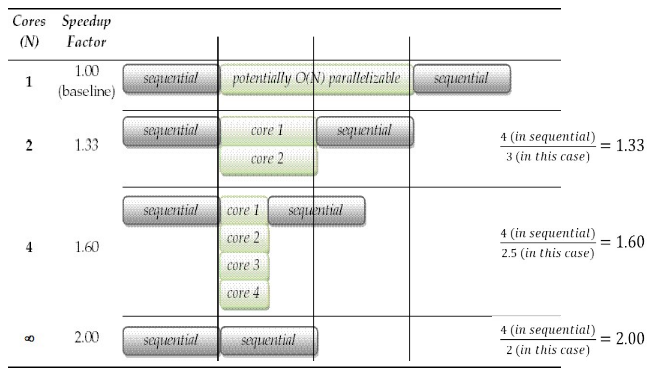
$$\omega_p = \frac{\textbf{Number of programs}}{\textbf{Time in seconds}}$$

## 6. Amdahl’s Law

- Gene Amdahl [AMDA67]
- Potential speed up of program using multiple processors
- Concluded that:
  - —Code needs to be parallelizable
  - —Speed up is bound, giving diminishing returns for more processors
- Task dependent
  - —Servers gain by maintaining multiple connections on multiple processors
  - —Databases can be split into parallel tasks

63

## 6. Amdahl’s Law


Cores (N)
Speedup Factor
1
1.00 (baseline)
sequential
potentially O(N) parallelizable
sequential
2
1.33
sequential
core 1
core 2
sequential
$\frac{4\ (in\ sequential)}{3\ (in\ this\ case)} = 1.33$
4
1.60
sequential
core 1
core 2
core 3
core 4
sequential
$\frac{4\ (in\ sequential)}{2.5\ (in\ this\ case)} = 1.60$
∞
2.00
sequential
sequential
$\frac{4\ (in\ sequential)}{2\ (in\ this\ case)} = 2.00$

64

## 6. Amdahl’s Law

- For program running on single processor
  - —Fraction f of code infinitely parallelizable with no scheduling overhead
  - —Fraction (1-f) of code inherently serial
  - —T is total execution time for program on single processor
  - —N is number of processors that fully exploit parallel portions of code

$$Speedup = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$$

- Conclusions
  - — *f* small, parallel processors has little effect
  - —*N* ->∞, speedup bound by 1/(1 – *f*)
    - – Diminishing returns for using more processors

65

## 6. Amdahl’s Law

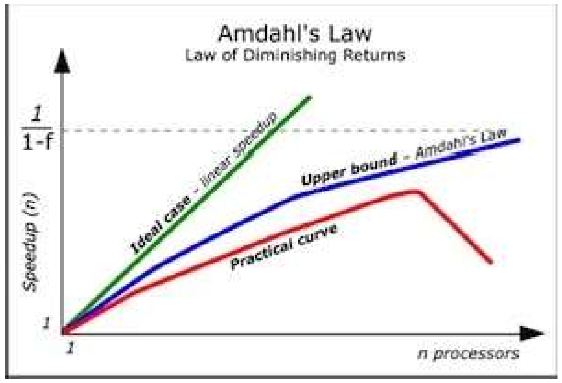
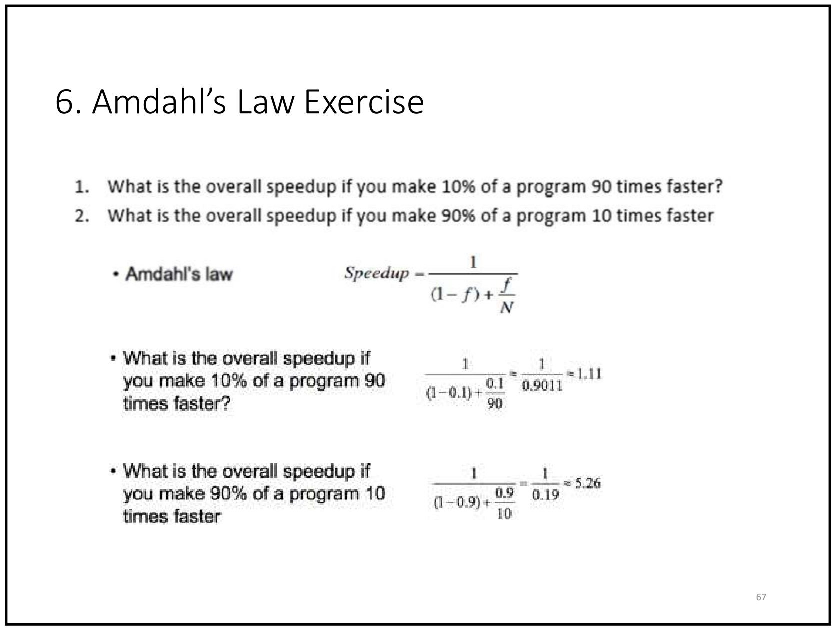


Fig. Speed-up vs number of processors for Amdahl’s law

66

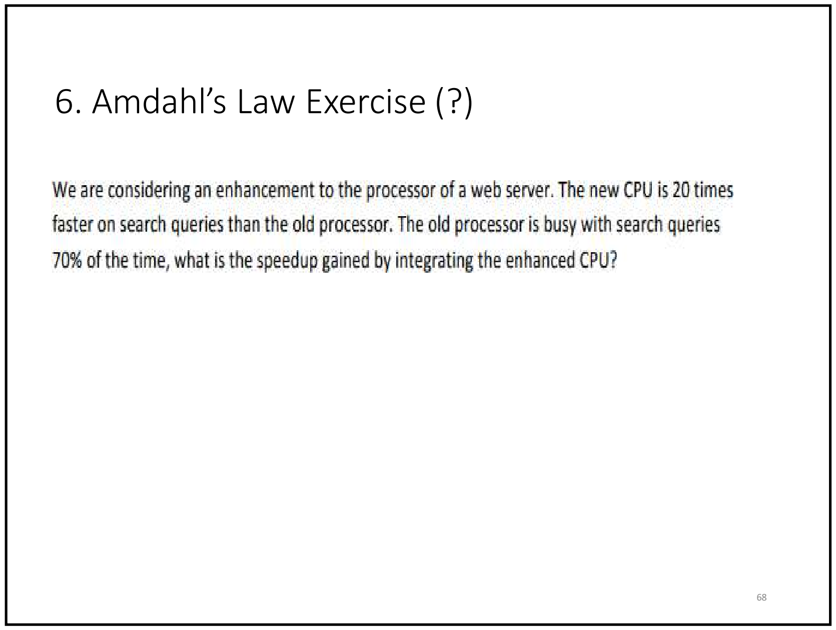


## 6. Amdahl’s Law Exercise

1. What is the overall speedup if you make 10% of a program 90 times faster?
2. What is the overall speedup if you make 90% of a program 10 times faster

- Amdahl's law $$Speedup = \frac{1}{(1-f)+\frac{f}{N}}$$
- What is the overall speedup if you make 10% of a program 90 times faster? $$\frac{1}{(1-0.1)+\frac{0.1}{90}} \approx \frac{1}{0.9011} \approx 1.11$$
- What is the overall speedup if you make 90% of a program 10 times faster $$\frac{1}{(1-0.9)+\frac{0.9}{10}} = \frac{1}{0.19} \approx 5.26$$

67



## 6. Amdahl’s Law Exercise (?)

We are considering an enhancement to the processor of a web server. The new CPU is 20 times faster on search queries than the old processor. The old processor is busy with search queries 70% of the time, what is the speedup gained by integrating the enhanced CPU?

68

Thank You

69