for Input from Keyboard Mov AH, 01H

for output on Screen Mov AH, 02H

Input

example Mov AH, 01H

INT 21H

→ Now Character is stored in AL

So if I want to store it in an Array

```
.data
Arr dB 5 dup(?)
.Code
Mov Cx, 5
Mov Si, 0
Label: Mov AH, 01H
INT 21H
Mov arr[si], AL
Inc Si
Loop Label
```

Output

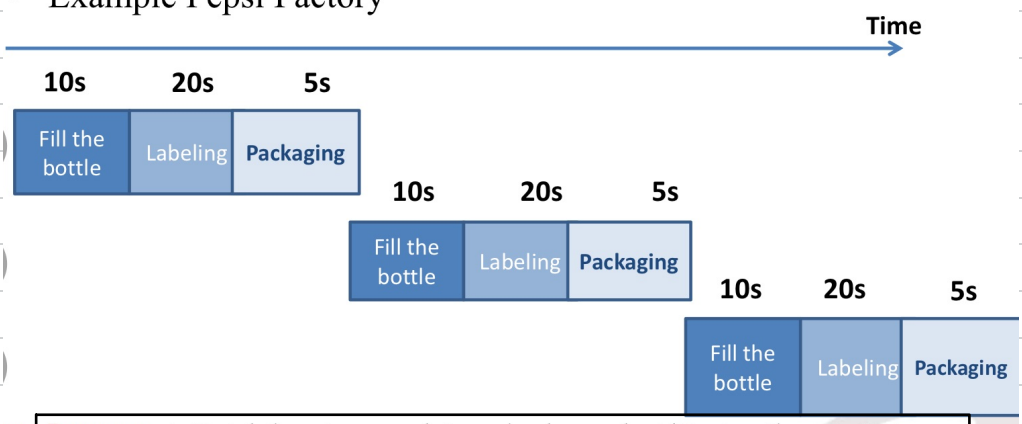
```
MOVDL, ARR[si]
Mov AH, 02h
INT 21H
```

→ Now it will output character stored in DL

```
.data
Arr dB 'Hello'
.Code
Mov Cx, 5
print_loop: Mov DL, arr[si]
Mov AH, 02h
INT 21H
INC SI
Loop print_loop
```

Sequential Designs

- Example Pepsi Factory



- Latency => Total time to complete a single product/ Instruction => Latency = 20+10+5 = 35s
- Throughput => Task completed per second = 1/35s

→ Definitions for sequential

Pipelining In Intel 8086

Pipelining

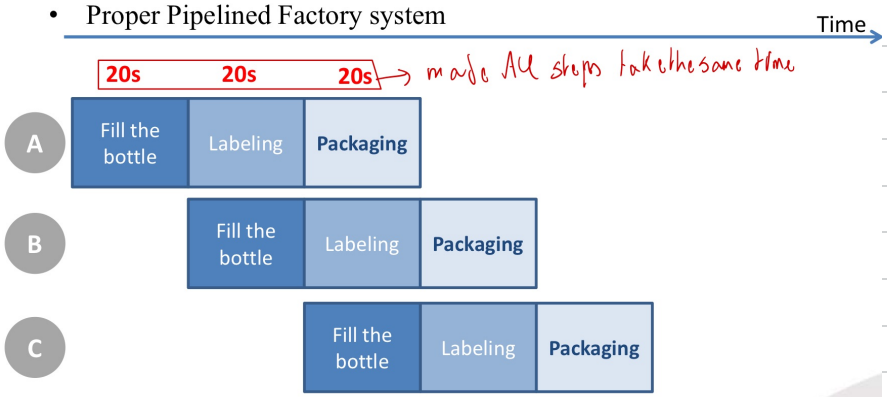
- **Critical Path:** Longest path or stage in the circuit between registers (Min. allowed clock period)
- **Latency**: Time delay from a certain input to receive its corresponding output (s / Instruction) As if 20 + 20 + 20

  Latency = No. of pipelined stages * Longest stage time
- **Throughput:** Maximum allowed clock frequency → Number of instructions completed per second (Instruction / s)

$$\text{Throughput} = \frac{1}{\text{Longest stage time (critical path)}}$$

→ Definitions for pipelining

- Proper Pipelined Factory system



- Latency = 20*3 = 60s
- Critical Path = 20s
- Throughput = 1/20s

- Processor Performance Equation:
  - *Clock cycle time* → Hardware technology and organization
  - *CPI (Clocks per Instruction)* → Organization and computer type
  - *Instruction count* → Instruction set architecture (total)

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

IC x CPI x Clock Cycle Time = CPU Time

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

n → represents number of distinct Instructions

IC_i → Instruction Count of ith instruction type

CPI_i → represent Number of clock cycles the i^{th} instruction takes to execute

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

Instruction Count (IC): This refers to the total number of instructions that a program executes. It is determined by the Instruction Set Architecture (ISA) of the program.

CPI (Clocks per Instruction): This is the average number of clock cycles each instruction takes to execute. It depends on the CPU's architecture and how it processes each instruction (organization and computer type). For example, simple instructions may take fewer cycles, while complex ones may take more.

Clock Cycle Time: This is the time taken for a single cycle of the CPU clock. It is influenced by the hardware technology and organization of the CPU. A faster clock (shorter cycle time) means the CPU can execute more instructions per second.

❖ Example

An application running on a 1GHz pipelined processor has 55% load-store, 30% arithmetic, and 15% jump instructions. The individual CPIs of these instructions are 5, 4 and 4, respectively.

- Determine the overall CPI of this program execution on the given processor.

Instruction	Occurrence(OLD)	CPI(OLD)
Load-Store	55%	5
Arithmetic	30%	4
Branch	15%	4

percentage

Overall CPI = $\sum_i CPI(i)\ x\ \frac{instruction\ occurance\ (i)}{Total\ number\ of\ instructions}$

Overall CPI = 0.55*5 + 0.30*4 + 0.15*4

Overall CPI = 4.55 cycles/instruction

or 0.55 + (1-0.25)

Instruction	Occurrence(OLD)	CPI(OLD)	Occurrence(NEW)	CPI(NEW)
Load-Store	55%	5	?	5
Arithmetic	30%	4	?	4
Branch	15%	4	?	6

Reduction in load-store = .25 * 55 = 13.75%
Reduction in Arithmetic = 0.05 * 30 = 1.5%
New overall instruction occurrence = (55% - 13.75%) + (30% - 1.5%) + 15%
= 84.75%

Overall CPI = $\sum_i CPI(i)\ x\ \frac{instruction\ occurance\ (i)}{Total\ number\ of\ instructions}$

- New load-store occurrence = $\frac{55-13.75}{84.75} = 48.67\%$
- New Arithmetic occurrence = $\frac{30-1.5}{84.75} = 33.62\%$
- New Branch occurrence = $\frac{15}{84.75} = 17.69\%$

Overall CPI = 0.4867*5 + 0.3362*4 + 0.1769*6

4.85 cycles/instruction

RISC

- RISC is the Reduced Instruction Set Computer.
- All operations on data apply to data in registers and typically change the entire register (**32 bits per register**).
- The **instruction** formats are few in number, with all instructions typically being one size (**32 bits**) which is preferred for pipelining.
- The **only operations that affect memory** are **load** and **store** operations that move data from memory to a register or to memory from a register, respectively.

Load R1, A → load operand in memory Address A in R1
Load R2, B → load operand in memory Address A in R2
Add r3, r2, r1 → r3 = r1 + r2
Store R3, A → store the content in register R3 into the A memory location.
↳ Stores in A (affects the memory)

Pipelining is not quite that easy!

Hazards that arise in the pipeline prevent the next instruction from executing during its designated clock cycle. There are three types of hazards:

- **Structural hazards:** Hardware cannot support certain combinations of instructions (two instructions in the pipeline require the same resource e.g. memory).
- **Data hazards:** Instruction depends on result of prior instruction still in the pipeline
- **Control hazards:** arise from the pipelining of branches and other instructions that change the PC

MIPS INSTRUCTION FORMAT

1. **R-Type:**

Instructions are used when all the data values used by the instruction are located in registers.

Add $s1,$s2,$s3

2. **I-Type:**

Instructions are used when the instruction must operate on an immediate value and a register value. Immediate values may be a maximum of 16 bits long

Addi $s1,$s2,100 → immediate

3. **J-Type:**

Instructions are used when a jump needs to be performed. The J instruction has the most space for an immediate value, because addresses are large numbers

MIPS INSTRUCTION FORMAT

R-Type

Register and Register

6	5	5	5	5	6
Opcode	rs	rt	rd	shift	Function

Register - Register ALU Operations (**rd←rs funct rt**)

Function encodes the data path operation: Add, Sub,.....
Read/write special registers and moves

I-Type

Immediate and Register

6	5	5	16
Opcode	rs	rt	Immediate

Source ↘ Target

Encodes: Loads and stores of bytes, half words, words, double words.
All immediate (**rt←rs op immediate**)

Conditional branch instructions (**rs is register, rd unused**)
Jump register, jump and link register (**rd=0, rs=destination,immediate=0**)

J-Type

Jump

6	26
Opcode	Address

Jump and Jump and link
Trap and return from exception

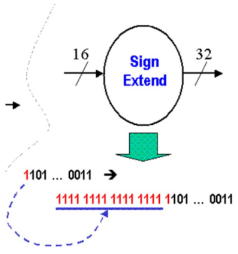
SIGN EXTENDER

To make the immediate Data the same size As the register

+100 → 0000 ....

-100 → 1111 ....
↳ make all left 1s

- The sign extender adds 16 bits to a 16-bit word repeating the most significant bit, resulting in a 32-bit word.
- The addition of 16 bits with same value as the most significant bit, results in a sign extension in twos complement representation.



LOAD/STORE INSTRUCTIONS

lw $r1, offset ($r2),

- **offset** denotes a memory address offset applied to the base address in register $r2.
- The **lw** instruction reads from memory and writes into register $r1.
- The **sw** instruction reads from register $r1 and writes into memory.
- In order to compute the memory address, the MIPS ISA specification says that we have to sign-extend the 16-bit offset to a 32-bit signed value. This is done using the sign extender.

Instruction

The datapath for the load/store instructions are carried out according to the following steps:

1.Register Access where the input is recorded from the register file, to implement the instruction, data, or address **fetch** step of the fetch-decode-execute cycle.

2.Memory Address Calculation the base address and the offset are combined together to produce the actual memory address. The sign extender and ALU are used in this step.

3.If the **instruction is a load**, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

4.Write the result into the register file, if it comes from the memory system (for a load instruction).

→ Summary for the steps

Lw r1, 200 (r2) r2 = 53

* lw → loadword This moves a value from a certain address from the memory and stores it in r1

The Address is gotten by offset + value stored in r2

ex: Address = 200 + 53 = 253

r1 = 77

77	253

* Sw → Store word Sw r1, 200 (r2)

Store the value inside r1 int Memory Address at (r2 + offset)

Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle

- **Structural hazards**: Hardware cannot support this combination of instructions (single person to fold and put clothes away) i.e., Two different instructions use same hardware in same cycle
- **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
- **Control hazards**: - Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
- -Pipelining of branches & other instructions that change the PC

*Structural ⟶ 2 Instructions Accessing Memory at the same time

Solutions: Add more functional units / memory ports

*Data ⟶ Data Dependancy

ADD R1, R2, R3
SUB R4, R1, R5

*Control ⟶ Jumping ex

BEQ R1, R2, Label
ADD R3, R4, R5

After BEQ is waiting whether to jump or not The CPU fetches ADD regardless [May discard it if The jump is done]

Hazard Type	Description	Example	Solution
Structural Hazard	Occurs when there are insufficient resources to execute instructions concurrently.	Memory access conflicts (e.g., reading and writing to memory at the same time).	Add more resources (e.g., more functional units, memory ports).
Data Hazard	Occurs when an instruction depends on the result of a previous instruction.	`ADD` followed by `SUB` using the same register.	Forwarding, pipeline stalls, or instruction reordering.
Control Hazard	Occurs when the CPU has to decide which instruction to fetch due to a branch.	Branch instructions (`BEQ` , `BNE` , `JMP`).	Branch prediction, pipeline stalls, delayed branching.