

Lec 1

Computer Science and Architecture.

Architecture :

Conceptual design & fundamental operational structure.

Architecture

{ CPU design

- Instructions
- Addressing modes
- Data Format

Organization :

Implementation of architecture.

Deals with physical devices and their interconnections.

The goal is improving the performance

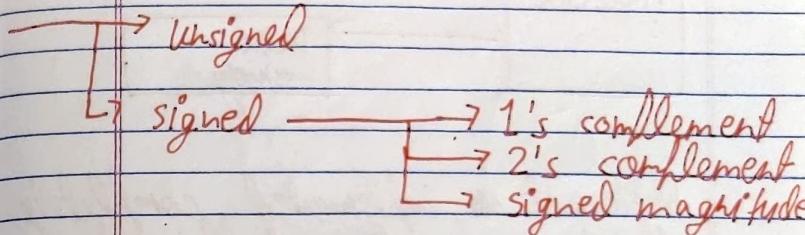
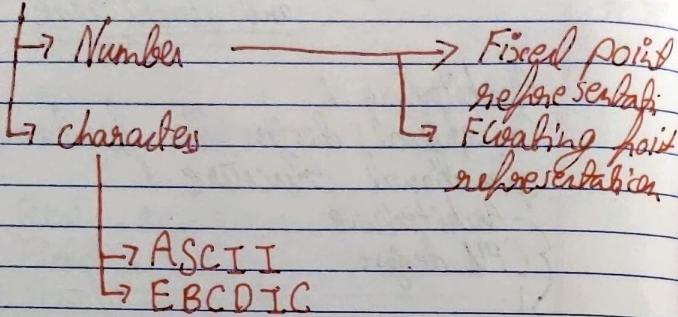
organization

- I/O organization
- Memory organization
- Performance improvement.

Date. _____
Page No. _____

Data Format \Rightarrow representing data in binary

Data Format

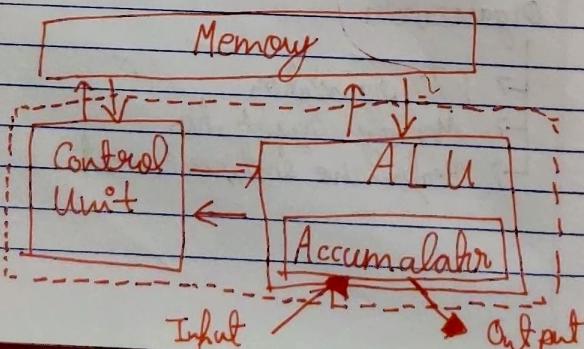


In this course we only cover floating point representation.

Von Neumann's architecture

Also called:
'Shared Program Architecture'

Basic structure is

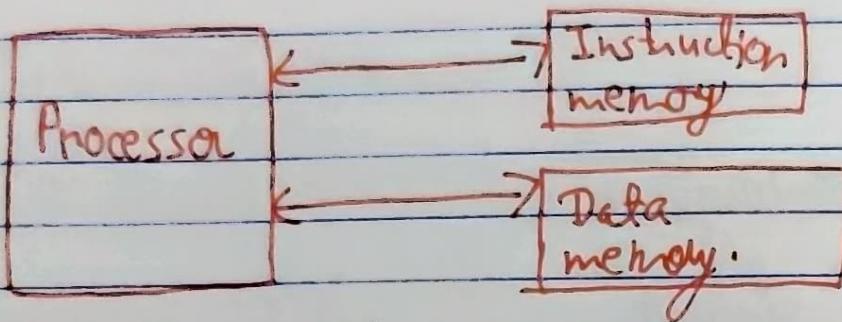


Bottleneck in Von Neumann's architecture:

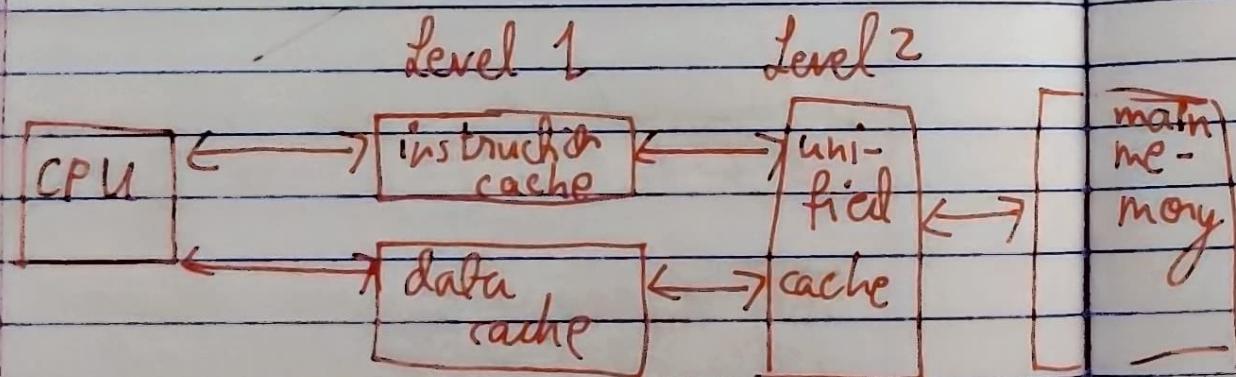
The instruction & data both cannot be fetched from memory simultaneously.

Harvard Architecture

A proposed solution to bottleneck, the memory is split into instruction memory & data memory.



While not ~~is~~ implemented completely,
it is used to implement
cache memory.



Dec 2

Date. _____

Page No. _____

Components of Computer

- CPU
 - ↳ Control Unit
 - ↳ ALU
- Memory
 - ↳ Primary / Main memory
 - ↳ Secondary / Auxiliary memory
- I/O Devices

Other Components

- System Buses
- CPU registers

System Bus \Rightarrow Connection line
b/w 2 devices within
Computer system.

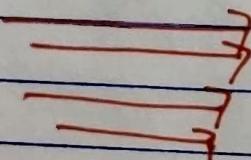
Types of system Buses

System Buses

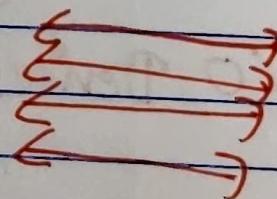
- └ Address Bus
- └ Data Bus
- └ Control Bus

- * Address Bus is unidirectional
- * Data Bus is bidirectional (Duplex)
- * Control Bus ~~is~~, every single line is unidirectional.

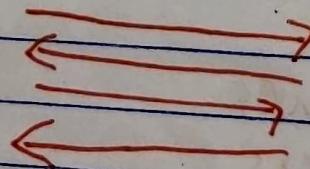
Address Bus :



Data Bus :



Control Bus :



Memory signals for CPU

- ① Wait signal
- ② Ready signal.

We need these since CPU is faster than memory & memory needs to tell CPU when it is busy.

Memory Cycle Time

Time from accepting a request till memory gets ready again to perform next operation.

CPU Registers

Small memories stored in CPU.

Types

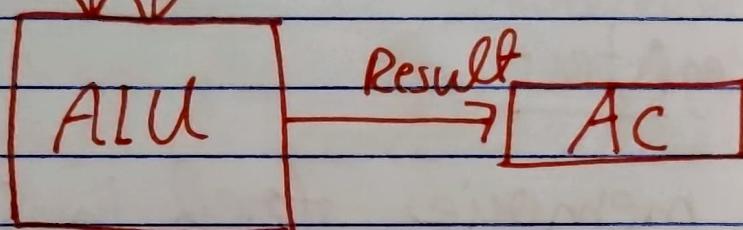
CPU Registers

- ↳ General Purpose Registers (GPR)
- ↳ Special Purpose Registers
 - ↳ Accumulator (AC)
 - ↳ Program Counter (PC)
 - ↳ Instruction Register (IR)
 - ↳ Stack Pointer (SP)
 - ↳ Flag Register / Program Status Word (PSW)
 - ↳ Address Register (AR)
 - ↳ Data Register (MDR) / MDR
- Memory

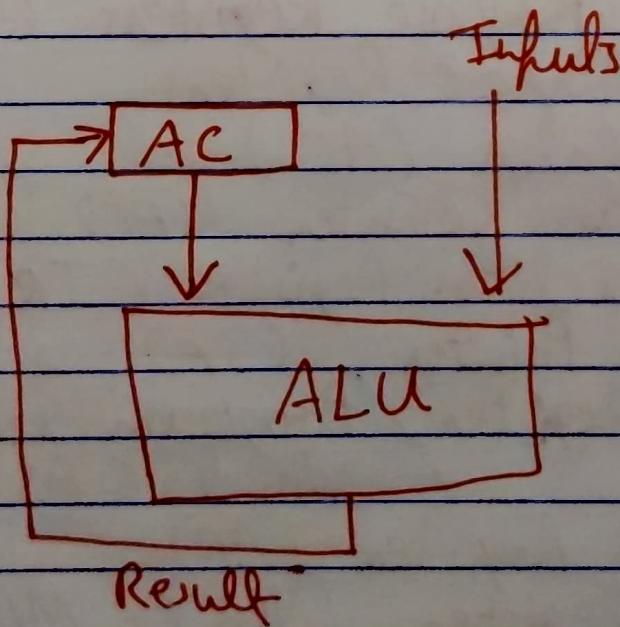
Accumulator :

Used to stored result of ALU
& sometimes fig store one
of the input for ALU also

Inputs:



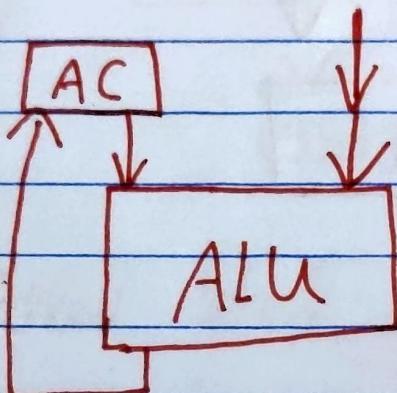
and sometimes,



called
Accumulator
based
architecture.

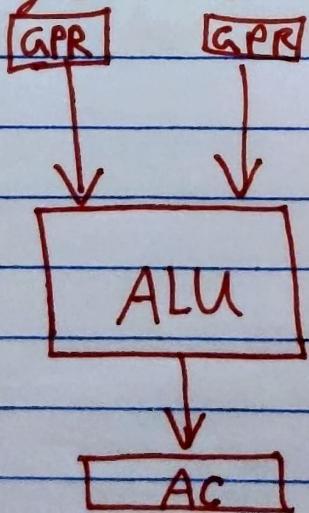
Types of architecture based on input to ALU

i) Accumulator based



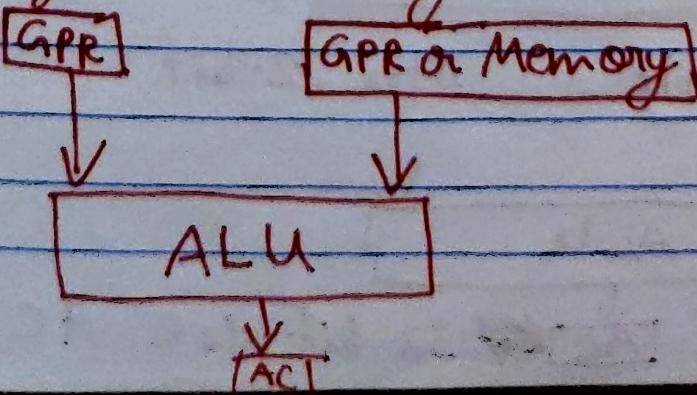
Eg,
 $AC' \leftarrow AC + R3$

ii) Register based

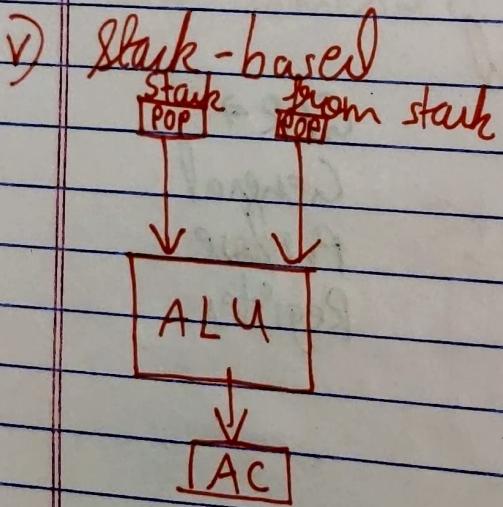
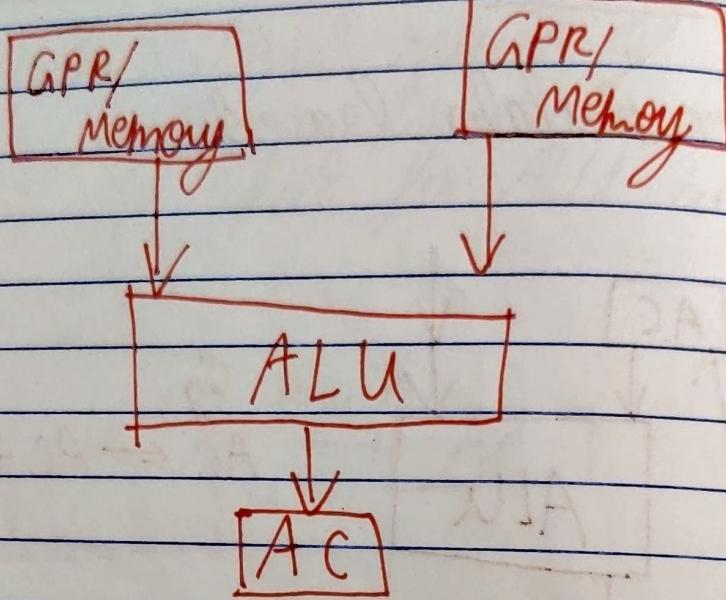


GPR \Rightarrow
General
Purpose
Register.

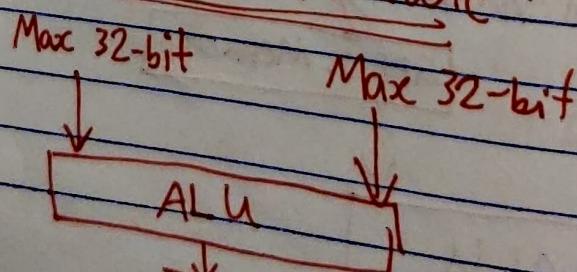
iii) Register-memory based



iv) Computer-system



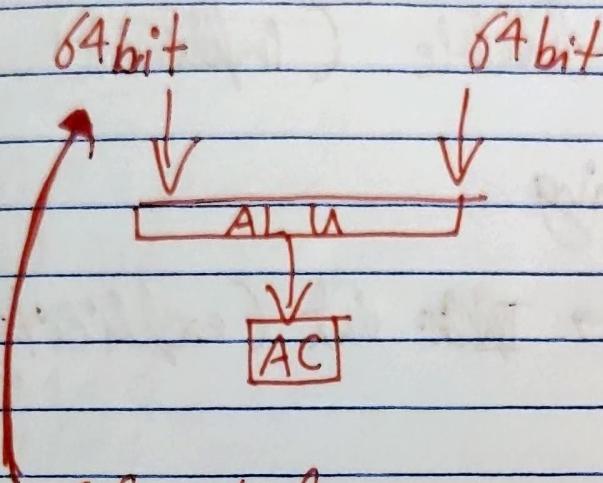
32-bit architecture



~~AC~~ 32 bit size Ac

Note: Accumulators were mainly used in single address computers. They are now considered ~~obsolete~~
Part No.
ALU now write directly to GPR.

Similarly in 64 bit.



Called Word size

Program Counter

To store address of next instruction

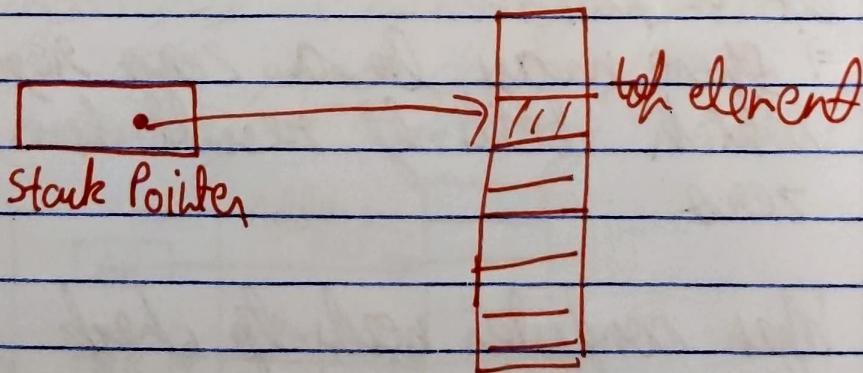
Lec 3

Instruction Register

The fetched instruction is stored in IR. It stores the current instruction which is being executed.

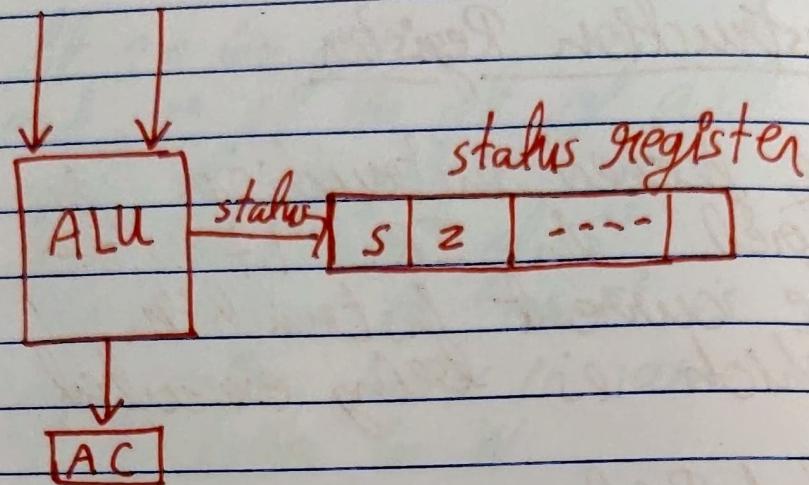
Stack Pointer

Points at the top of the stack allocated for running the program.



Flag Register / Status Register

All generates two outputs
Result of operation
Status.



It stores the status of ALU result.
For example,

⇒ There may be a sign register which tells if result is negative or positive

⇒ There may be a zero register which tells if result is zero.

When computer needs to check for conditional statements like $a > b$ or $a == b$, it can use status register flags to check.

Program Status Word (PSW)

In some implementations it only refers to the

Flag registers.

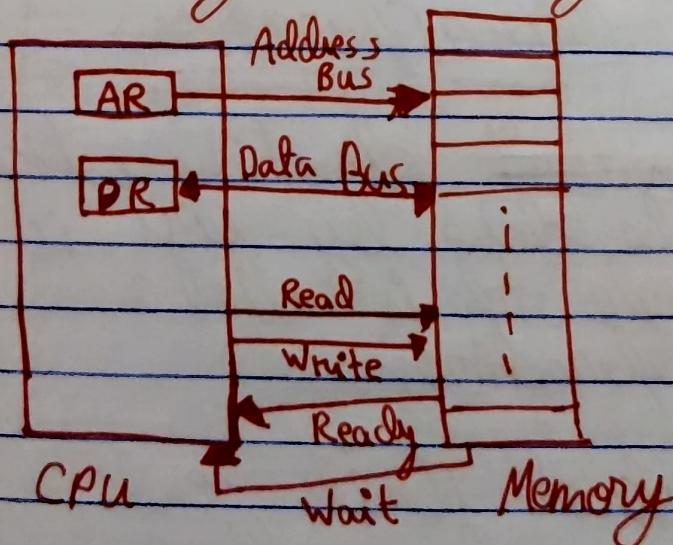
In others it refers to flag register + accumulator.

Address Register

Used to set send address to memory.

Data Register

Used to send data to memory (memory write) & to read data from memory (memory read).



Read/Write/Ready/Wait are components of control bus, connected to control unit.

Memory Read

- i) ~~To memory~~ CPU sends address
- ii) CPU sends read signal
- iii) Data is sent through data bus.

Memory Write

- i) CPU sends address
- ii) CPU send data through bus
- iii) CPU sends write signal
- iv) Memory writes the data

Micro Operation

The ~~smallest~~ operation which can be done by CPU is a single step is called a microoperation.

The Control Unit sends the command for these micro operation.

Basically, things like read or write cycles are complex. So they are broken into small steps. These small steps are micro operations.

Formally, A micro operation is an elementary operation performed with data stored in registers.

Microoperation

- ↳ Register transfer
- ↳ Arithmetic microoperations
- ↳ Logic microoperations
- ↳ Shift microoperations

Operations on CPU registers are micro operations

RTL (Register Transfer Language) is the symbolic notation of microoperations

In RTL,

1. Register Transfer

$$R_1 \leftarrow R_2$$

2. Comma, can be used to separate operations that may be executed simultaneously
 $R_1 \leftarrow R_2 + R_3, PC \leftarrow PC + 1$

3. Memory Transfer

Memory Read

$$R_1 \leftarrow M[\text{address}]$$

e.g., $R_1 \leftarrow M[2000]$

$$R_2 \leftarrow M[\text{AR}]$$

Memory Write

~~M~~ $M[\text{address}] \leftarrow R_1$

e.g., $M[2100] \leftarrow R_1$

~~AR~~ $\cancel{M[\text{AR}] \leftarrow R_1}$

$$M[\text{AR}] \leftarrow R_1$$

~~AR~~ Remember AR stores
the memory addresses

so $M[]$ symbolizes the
data in memory

Arithmetic microoperations

- Addition $R_1 \leftarrow R_2 + R_3$
- Subtraction $R_1 \leftarrow R_2 - R_3$
- Complement $R_1 \leftarrow \bar{R}_2$
- 2's Complement $R_1 \leftarrow \bar{R}_2 + 1$
- Addition with 2's Complement $R_1 \leftarrow R_2 + \bar{R}_3 + 1$
- Increment $R_1 \leftarrow R_1 + 1$
- Decrement $R_1 \leftarrow R_1 - 1$

Note : 2's complement gives the negation, so 2's complement of 5 is -5.

Logic type microoperation

- AND $R_1 \leftarrow R_2 \wedge R_3$
- OR $R_1 \leftarrow R_2 \vee R_3$
- X-OR $R_1 \leftarrow R_2 \oplus R_3$
- X-NOR $R_1 \leftarrow R_2 \odot R_3$

Shift Operations

- Logical Shift
- Circular Shift
(Rotation)
- Arithmetic Shift.

Logical Shift

Shift the bits left / right & add zero at vacant position.

Eg,

1001		
After	Left shift	0010
After	Right shift	0100

Circular Shift

The bit at extreme position is not discarded, bent added to other end.

Eg,

1001		
After	Left shift	0011
After	right shift	1100

Arithmetic Shift

Only applied on signed numbers.

After the shift sign of the number should not change.

1001
↑

~~Sign bit~~

Sign bit

1 \Rightarrow negative
0 \Rightarrow Positive

Ques Let's take 1001

Right shift \Rightarrow

1100
↑

1 bit was sign bit & hence was taken from original value.

Left shift \Rightarrow

Do the logical left shift

If the sign bit changes,

- go back to the original value else we get the logical left shifted value.

Eg,

1101

Left shift

1010



New value

1001

Left shift is 0010



Sign bit changes

so, ~~Left shift~~ arithmetic
left shift not allowed

An error will be generated
named Arithmetic Left Shift
Overflow.

Word addressable / Byte-addressable

* If memory addresses
can refer to a single word,
then memory is word addressable

* In word addressable, a single
memory cell has capacity
of a single word. (word
can be of any size, depending
on different systems) (usually its 32 bits)

In byte addressable memory,
a single memory cell only
stores a single byte.
(A byte is 8 bits).

Digital computer takes input in binary form and gives output as binary form.

Program statements are compiled to instructions. These instructions are in binary form.

Instruction \Rightarrow A group of bits which instruct computer to perform operations.

General instructions have two parts parts,

operation | operand info.

operation code/
opcode

Example, if a CPU has 8 bit instructions and 8 bits, first 3 bits are opcode,

There can be 2^3 distinct operations that are supported.

ISA (Instruction Set Architecture) is collection of all instructions / operators the CPU can support.

Size of ISA = no of distinct instructions
So in our example, 2^3 is max.

Note: All $2^3/8$ instructions may not be used in which size of ISA can be lowered.

Types of operation
(Based on Operand Information)

- 4- Address Instruction
- 3- Address Instruction
- 2 - " " "
- 1 - " " "
- 0 - " " "

4-address instruction

Maximum 4-addresses
can be specified

opcode	add 1	add 2	add 3	add 4
--------	-------	-------	-------	-------

operand

↓
next
instruction
address

4-Address instruction computers
don't have Program Counter
(PC).

Since relocation is hard in
4-address instructions, they
are rarely used.

3-address instruction

Maximum 3-address can
be specified.

opcode	addr 1	addr 2	addr 3
--------	--------	--------	--------

2-Address Instruction

Max 2-address can be specified within an instruction.

opcode	addr 1	addr 2
--------	--------	--------

Examples;

of 3 address $R_3 \leftarrow R_2 + R_1$
~~of 3 address~~

of 2 address $R_2 \leftarrow R_2 + R_1$

1-Address Instruction

Only a single address can be specified

opcode	address
--------	---------

Accumulator is used as second operand implicitly.

Hence, 1 address instruction use accumulator based architecture. (lec 2).

0-address Instruction

No address are mentioned, so ~~this~~ this instruction set is only used for stack based architecture.

The operands are taken from stack.

Note: Higher address instruction CPU can support lower address instruction,

Eg, A 3-address CPU can support 2-address, 1-address and 0-address as well.

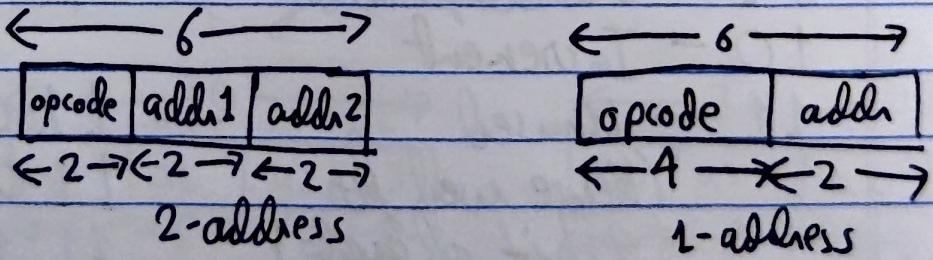
Lec - 6

Multiple Instruction Support

Consider a system which will support both 2-address and 1 address instructions

The system will have 6-bit instructions and 2-bit addresses.

How can we design a way to avoid conflict between 1-add & 2-add. instructions?



We start by looking at the instruction type with smaller op code. (i.e the 2-address one in our example).

Suppose we want 3 2-address instruc-

so one 2-bit opcode is unused.

This unused opcode is used to identify the operation. For instance, as in 3-address instruction.

So there has to be at least one unused opcode from maximum allowed.

Eg, suppose the 2-bit opcodes are,

00 - Add

01 - 2's complement

10 - Increment

11 - Unused

(Hence used for

4 bit opcodes)

- 11 00)

11 01)

11 10)

11 11)

Availability
4-bit
opcodes.

If there are more supported instructions, we will always start

with smallest opcode and use unused to identify higher bit opcodes.

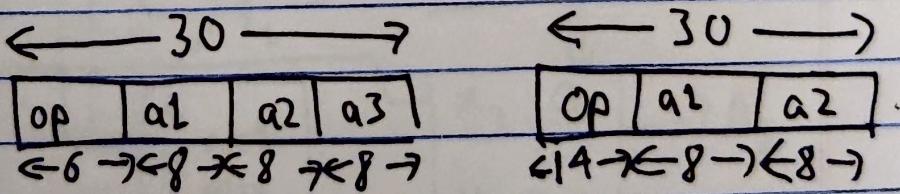
The amount of unused opcodes is adjusted we we want more higher bit opcodes.

(Q) A system has 3-address & 2-address instructions.

It has 30-bit instructions and 8-bit addresses.

There are ' x ' 3 address instructions how many 2-address instructions can be formulated?

(Sol)



As x 3-address opcodes are used out of total 2^8

$(2^8 - x)$ are unused

For each unused in 8-bit code, we have 8 bits remaining for 14 bit one.

So,

$$8(2^6 - x)$$

$8(64 - x)$ is maximum 2-address
Instruction possible.

Lec 7

Effective Address

- In a computation type instruction, the effective address is the address of operand in memory.
- In branch type instructions, the target (place to jump in program) address is effective address.

Instruction cycle

CPU performs 6 phases to execute a single instruction. These 6 phases form a complete instruction cycle.

i) Instruction fetch

The instruction is fetched using address in PC. The instruction is copied to IR.

PC is incremented by size of instruction.

ii) Instruction decode

The opcode is decoded for

the type of operation / instruction.

iii) Effective address calculation
Based on the type of instruction,
~~calculate the~~ decode / read
data & then move to the
decode the effective address.

iv) Operand fetch

Using the decoded effective
address, fetch data from memory
/ register.

v) Execution

Execute the operation using
fetched operands in ALU.

The result is stored in
Accumulator.

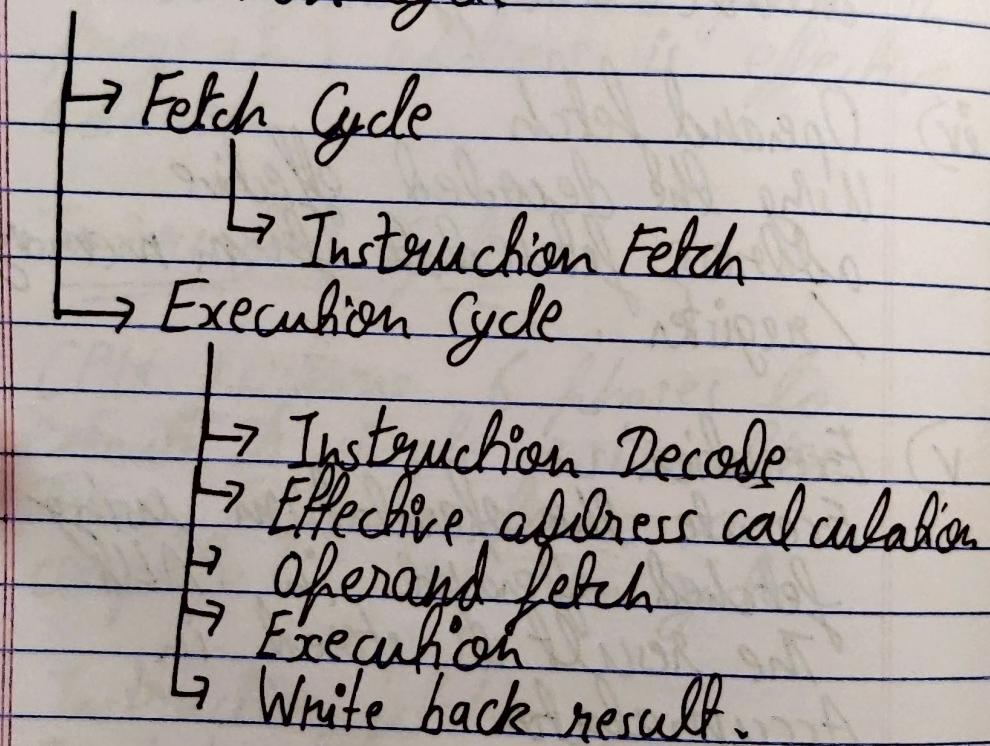
vi) Copy / write back result

Write result from accumulator
to the specified location
in memory / register.

~~More broadly,~~

More broadly, we have two different cycles, this type of categorization are named Fetch & Execution cycle.

* Sometimes a have called "Decode instead of Instruction Cycle"



* Every instruction does not require all 6 phases.

Addressing Mode

The addressing mode tell CPU how to obtain operand using address field value of instruction.

Implied mode

Opcode definition specifies operand also. So no mode is specified.

Immediate mode

The address field of instruction specifies operand value.

The data in address field is itself used as operand.

Direct mode / Absolute Mode

The address field of instruction specifies effective address, i.e. the operand is in memory at the given address.

Indirect mode

The address field of instruction specifies an address in memory which has the effective address.

The indirect mode is used to implement pointer.
When dereferencing we get data using indirect mode.

Lec 8

Addressing Modes

Register Mode / Register Direct Mode

The address field of instruction specifies a register which holds the operand.

Register Indirect Mode

The address field of instruction specifies a register which holds the effective address.

This mode is mainly used to shorten the instruction length, because register reference bytes are shorter than memory address size.

Autoincrement / Autodecrement Mode

- Used to access table of content (array) sequentially.
- Variant of register indirect mode.
- Content of register is automatically incremented or decremented.

Autoincrement ~~and decrement~~ is post increment ~~and decrement~~

so the operation is done ~~on~~
~~the~~ & then the data in register is changed to the next effective address in sequence.

On the other hand, Autodecrement is predecrement.

Index Register Mode

This mode is used to access a particular element of an array.

The address field on the instruction contains the base address.

The index value is stored in the index register (hence, name of the mode).

The base address and index value address are added to get effective address to operand.

$$A[i] = B + i \times w$$

Base
address

w is size of a single element
 $i \times w$ is index value

This mode hence mostly uses two instructions.

1. Index register $\leftarrow w * i$
2. Instruction with this addr. mode.

Implementation

Index Reg. is
special purpose

Index Reg. is
not special

opcode	mode	addr.
--------	------	-------

opcode	mode	Index Reg. Identifier
--------	------	-----------------------------

If program is relocated in memory,
the base address needs to be
changed if instruction

* As updating instructions is
time costly, the efficiency may
be lost.

Index, Base Register Mode

Rather than storing base address in instruction's address field, it is stored in a register called Base register.

$$\text{Effective addr.} = \frac{\text{Index Reg value}}{\text{Base Reg Value}}$$

If index reg & base reg are not special purpose, then instruction of type

Opcode	mode	Index Reg Reference	Base Reg Reference	addr. (optional)

Lec - 9

Index, Base Register + Displacement Mode

Suppose we have a struct in memory (or a record). We made an array of that struct.

Using Base + index we can get the ~~the~~ struct.

But if we want to access a member of struct, we will have to use displacement to that member in struct.

In the address field of instruction, the displacement is stored.
Base / Index value are stored in their registers.

Effective address = Base Reg + Index + Dis
Add. Reg place
Value Value mat

Scaled Mode

This mode is also used to access an element from array.

The address field has the base value and the index number is in register.

Effective address = Address field value
of instruction
+ index \times scaling
Reg Value factor

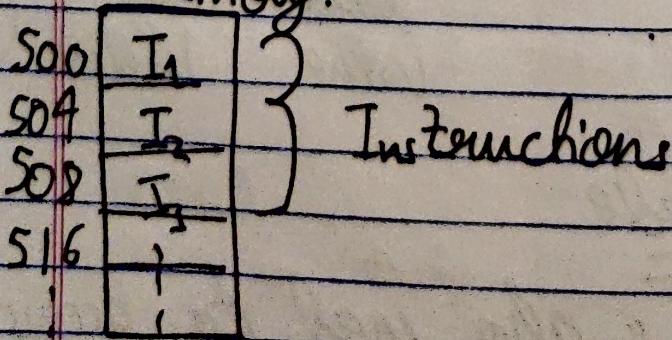
↓
taken

This mode is more efficient than
base register mode. implicitly

PC-Relative Mode

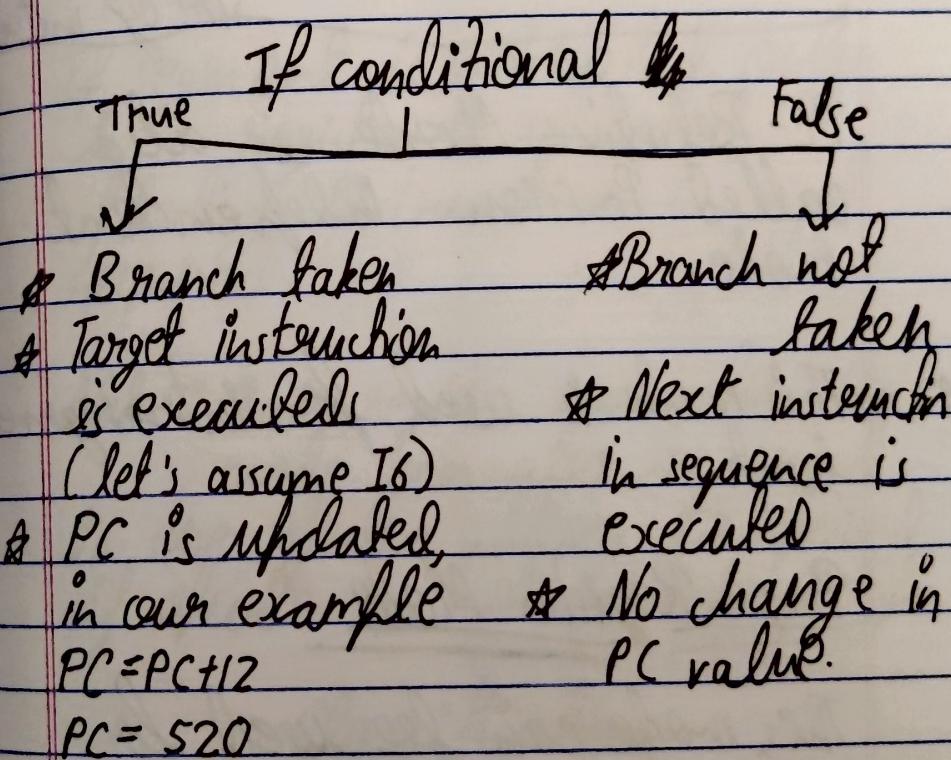
This mode is used for branch instruction.

Assume memory as
memory.



Assume CPU is executing instruction I₂, so PC = 508.

While decoding CPU decodes I₂ is Branch instruction,
(conditional)



opcode	Mode	address
--------	------	---------

In effective address PC-relative mode, the value is updated as,
 $PC = PC + \text{addr. field value of instruction}$

Hence, address field has relative location to skip (this value is sometimes called offset).

No. of instructions = address field value
skipped
size of instruction

PC-Relative Mode is also called Position Independent mode.

- * This mode is used for intrasegment jump.

Base Register Mode

This mode can be used for inter segment jumping.
(Inter segment means in another segment).

If branch taken, then PC is updated as

$$PC = \text{Base register} + \text{offset value}$$

Here, base register will store value to the start of the target segment.

Offset is in address field of instruction.

If segment is reloaded only base register is updated, updating instruction is not required.

Note : Base register can also be mode can also be used to store effective address to operand, if opcode is not branching type.

Classification of Modes

Computable

Non-computable

Direct \leftrightarrow Implied

Indirect \leftrightarrow Immediate
 \rightarrow Register Mask
 \rightarrow Register Indirect

Complementable

- Autoinc./Autodecrement
- Indexed Mode
- Index + Base Reg. Mode
- Index + Base Reg + Displacement
- Scaled
- PC-relative / Position independent
- Base Register

* PC-relative & Base register
can be used for branching
hence sometimes called
control modes.

* Remaining all used for data
access for operations.

Lec 10 is questions & on
addressing modes.

classmate _____

Date _____

Page _____

Lec 11

CPU Cycle

Time in which CPU performs one micro-operation

$$\text{Clock rate} = \frac{1}{\text{Clock Cycle}}$$

CPI (Cycles Per Instruction)

No. of cycles required to execute an instruction.

Execution Time

For 1 instruction the time taken is $CPI_{avg} * 1 \text{ cycle time}$

In execution of n instructions it is calculated as

$$= n * CPI_{avg} * (1 \text{ cycle time})$$

Suppose a program has m different types of instructions

Then average CPI is

$$\text{CPI}_{\text{avg}} = \frac{\sum_{i=1}^m n_i \times \text{CPI}_i}{\sum_{i=1}^m n_i}$$

n_i = no. of instruction of type i .

CPI_i = CPI for instruction in type i .

Million Instructions per second

(MIPS)

It is a way to measure CPU performance.

Suppose it takes t seconds to execute n instructions.

no. of instructions in 1 second = $\frac{n}{t}$

~~Time taken for instruction~~

Instructions per second = $\frac{n}{t}$.

Million Instructions per second = $\frac{n}{t \times 10^6}$.

MIPS = $\frac{n}{\text{Execution time} \times 10^6}$

Using formulae of Execution time

MIPS = $\frac{n}{\mu \times CPI_{avg} \times (\text{1 Cycle Time}) \times 10^6}$

MIPS = $\frac{1}{CPI_{avg} \times (\text{1 Cycle Time}) \times 10^6}$

MIPS varies based on number
and complexity of instructions.

So we use FLOPS (Floating Point
Operations per second) as a
standard metric.

Arithmetic and Logic Unit

