Chapter 1

A computer: a sophisticated electronic calculating machine

A computer does the following:
1. Accepts input information
2. Processes information according to a list of internally stored instructions
3. Produces the resulting output information

Functions performed by a computer

1. Accepting information to be processed as input.
2. Storing instructions to process the information.
3. Processing the information according to the instructions.
4. Providing the processing results as output.

Functional units of a computer

1. Input/Output (includes):

A. Input Units: Accepts information, either by
1. Human operators,
2. Electromechanical devices
3. Other computers

B. Output unit: sends results of processing to either a
1. monitor display
2. printer

2. Memory

Stores Information
examples:
A. Instructions

B. Data

3. Processor (includes)

B. Control unit: coordinates various actions (examples):
1. Input,
2. Output
3. Processing

Information in a computer

Instructions

Instructions specify commands to :

1. Transfer info within a computer (e.g., from memory to ALU)
2. Transfer info between the computer and I/O devices (e.g., from keyboard to computer, or computer to printer)
3. Perform arithmetic and logic operations (e.g., Add two numbers, Perform a logical AND).

Program: A sequence of instructions to perform a task, is stored in the memory

Question: what does a Processor do

1. fetches instructions that make up a program from the memory
2. performs the operations stated in those instructions.

Data

Data: operands upon which instructions operate.

Data could be either
1. Numbers
2. Encoded characters

Data could also mean any digital information.

Bits: Data that is encoded as a string of binary digits.

Input unit

Input Unit Tasks:
1. Interfaces with input devices
2. Accepts binary information from the input devices.
3. Presents binary information in a format expected by the computer.
4. Transfers information to the memory or processor.

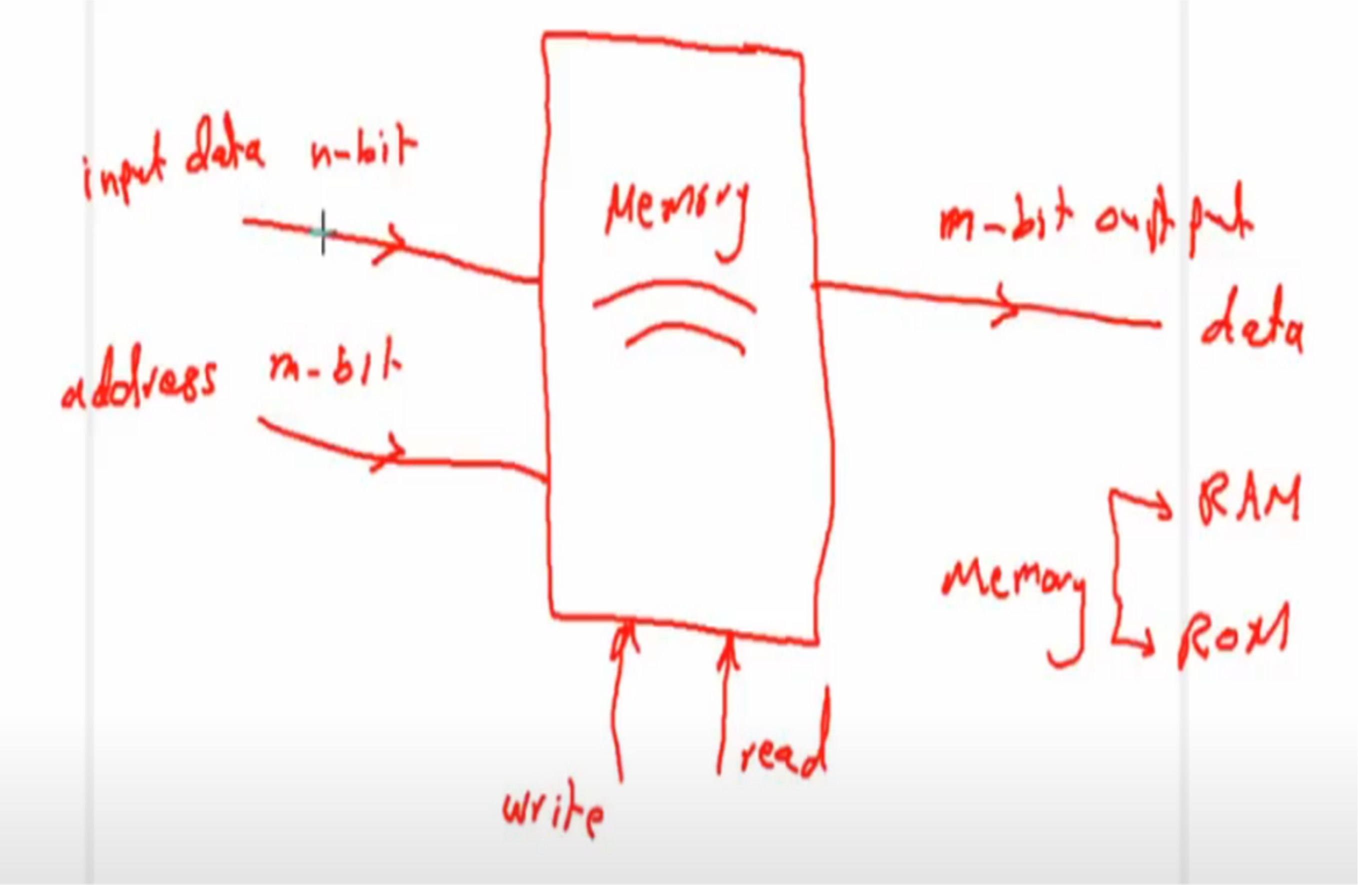
Memory unit

- Stores instructions and data
- To store data, memory unit thus stores bits (as data is a series of bits).
- Processor reads instructions and reads/writes data from/to the memory during the execution of a program.
- In theory, instructions and data could be fetched one bit at a time
- In practice, they are fetched as a group
- word: Group of bits stored or retrieved at a time
- word length: Number of bits in a word
- to read/write to and from memory, a processor should know where to look:
- Address: each word's location.

The CPU includes:
- Control unit

- Registers
- ALU

Memory Drawing :



Data Lines and Address Lines

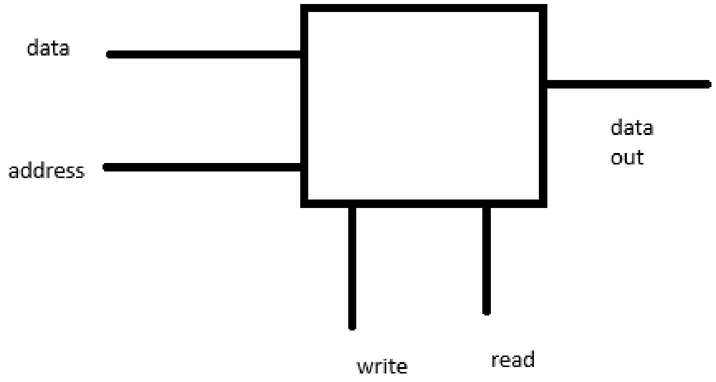
in an example n_1 x n_2

The Data In/Out (aka Data Bus or Data Line) is equal to n_2

Meanwhile the Address Line (or address bus) can be calculated using the following formula

Address Line = $\log_2$(n1)

we might need to draw the block diagram of a memory, which looks like this



The Memory has a location size that's equal to n_1, starting from 0 to n_1-1, each location storing $n_2$bits

in a location, we convert it to binary to find the location number
(we added 0s before the number equal to the size of the memory)

example: in location 15 the address number is 0000001111

list of 2^n

Number	Power
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256

Number	Power
2^9	512
2^{10}	1024

kilo = 2^{10}
mega = 2^{20}
giga = 2^{30}

Exercise:

we have the number 1024 x 16

1. Find the data in/out size
   answer: 16
2. find the address line
   answer: $\log_2(1024) = 10$

we have the memory size: 8k x 8

1. Find the data bus
   answer: 8
2. find the address line
   answer: $\log_2(8k = 8 * 1024 = 2^3 \times 2^{10}) = 13$

Memory unit (contd..)

Processor reads/writes based on the memory address:
Using the address, We can access any word location in a short time.
The RAM provides fixed access time independent of the location of the word.
Memory Access Time: Access time.

Primary storage is insufficient to store large amounts of data and programs, we can add more but it's expensive.

we can Store large amounts of data on secondary storage devices (examples):
1. Magnetic disks and tapes.
2. Optical disks (CD-ROMS).

Arithmetic and logic unit (ALU)

Operations executed in the ALU
1. Arithmetic operations (addition, subtraction.)
2. Logic operations (comparison of numbers)

Operands are stored in general purpose registers available in the ALU, where there Access times is faster than the cache.

The results are stored back in the memory or retained in the processor for immediate use

Output unit

•Computers represent information in a specific binary form. Output units:

- Interface with output devices.
- Accept processed results provided by the computer in specific binary form.
- Convert the information in binary form to a form understood by an output device.

examples:
1. Printer
2. Graphics display
3. Speakers

Control unit

Operation of a computer can be summarized as
1. Accepts information from the input units (Input unit).
2. Stores the information (Memory).
3. Processes the information (ALU).
4. Provides processed results through the output units (Output unit).

How are the functional units connected?

•For a computer to achieve its operation, the functional units need to communicate with each other.

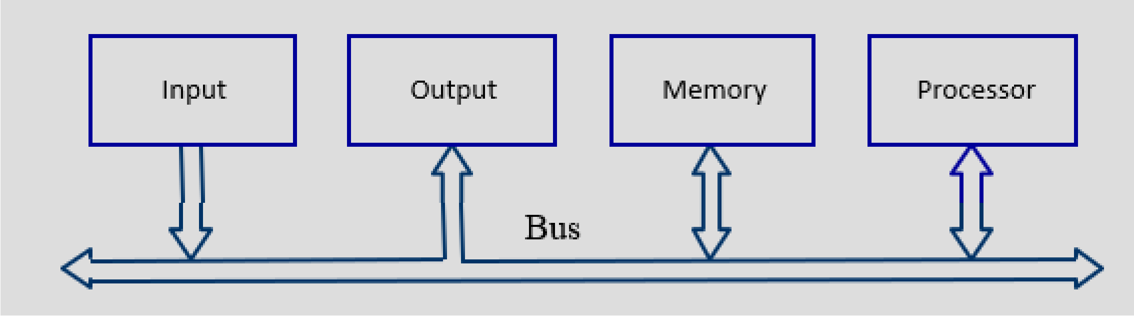
•In order to communicate, they need to be connected.

•Functional units may be connected by a group of parallel wires.

•The group of parallel wires is called a bus.

•Each wire in a bus can transfer one bit of information.

•The number of parallel wires in a bus is equal to the word length of a computer

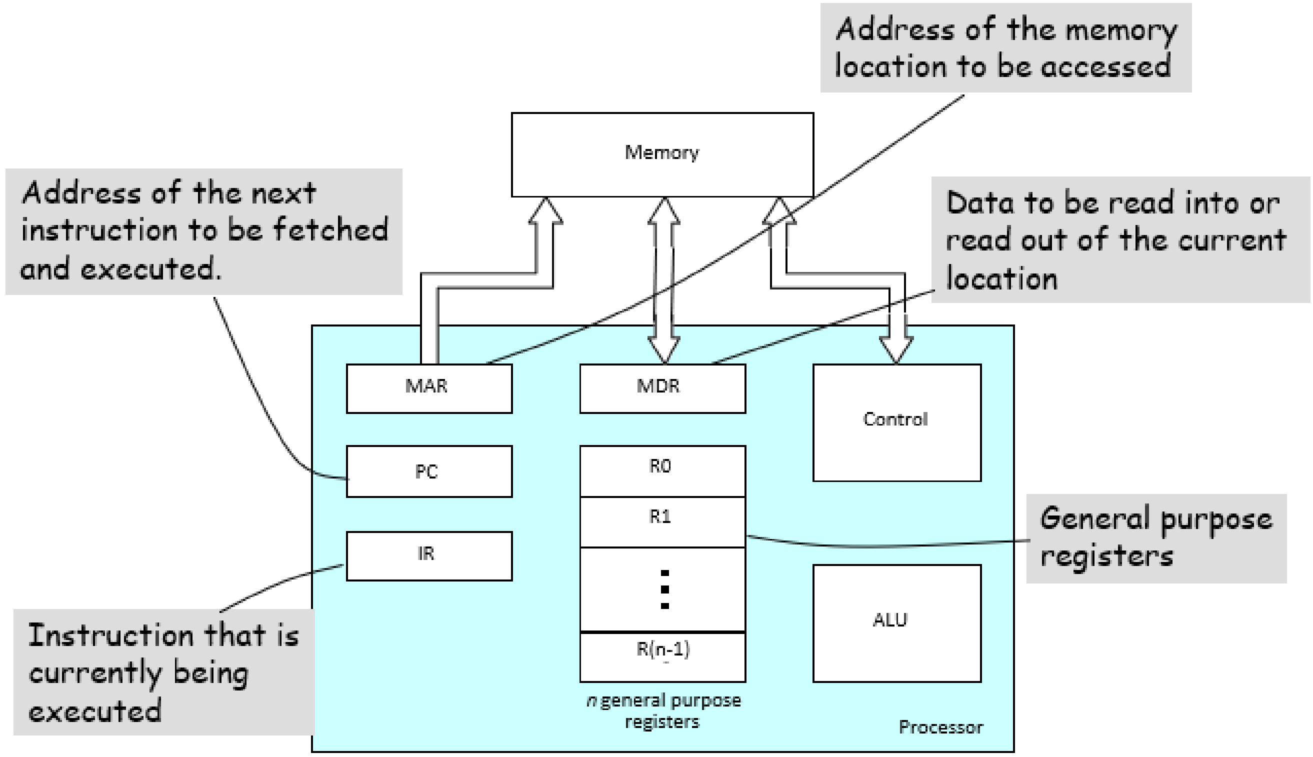


Chapter 2

Execution of an instruction:

steps involved in the execution of an instruction by a processor:
1. Fetch an instruction from the memory.

2. Fetch the operands.
3. Execute the instruction.
4. Store the results.



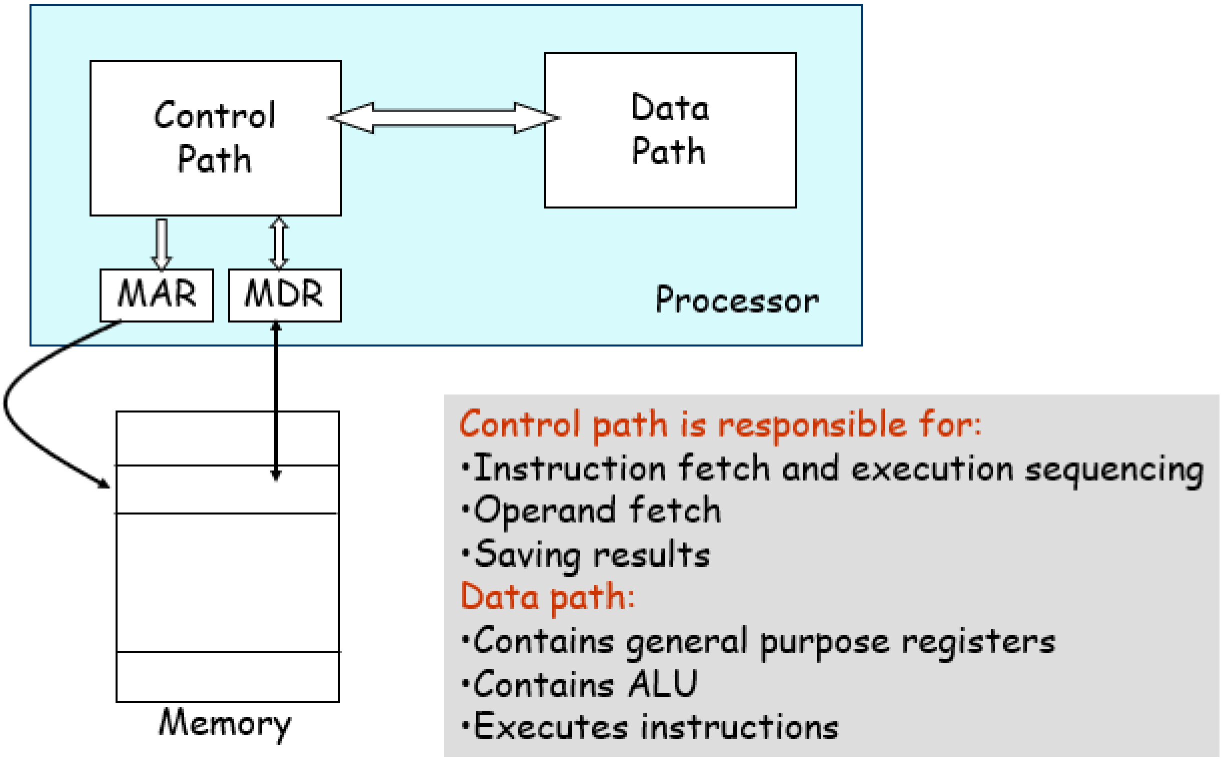
Address of the memory location to be accessed

Address of the next instruction to be fetched and executed.

Data to be read into or read out of the current location

General purpose registers

Instruction that is currently being executed



Control path is responsible for:
- Instruction fetch and execution sequencing
- Operand fetch
- Saving results

Data path:
- Contains general purpose registers
- Contains ALU
- Executes instructions

Basic processor architecture has several registers to assist in the execution of the instructions.

Registers in the control path

Instruction Register (IR): Instruction that is currently being executed.

Program Counter (PC): Address of the next instruction to be fetched and executed.

Memory Address Register (MAR): Address of the memory location to be accessed.

Memory Data Register (MDR): Data to be read into or read out of the current memory location

Fetch/Execute cycle

Instruction fetch: Fetching the instruction from the memory location, and placing it in the IR.

Instruction execute: Instruction in the IR is examined (decoded) to determine which operation is to be performed, then the operand is fetched and executed, the results is then stored in destination location.

The cycle repeats indefinitely.

Memory organization

Collection of 8 bits known as a “byte”, is more simple than a word.

Bytes are grouped into words, word length can be expressed as a number of bytes:

Word length = 16 bits = 2 bytes.

Size	Age
2 bytes	Older architectures
4 bytes	Current architectures
8+ bytes	Modern architectures

To obtain info from the memory, we need to specify the “address” of the memory location.

a memory has a sequence of bits, but assigning addresses to each bit is impractical and unnecessary, so we assign to bytes instead (Byte addressable memory).

if k bits are used to hold a memory location address:
memory size (in bytes)= 2^k

examples: k = 24
memory size = 2^{24} = 16,777,216

Memory is viewed as a sequence of bytes, starting from 0 to 2^k-1

example when k =2
we start from 0 to 3

Number of words = Memory size(bytes)/ Word length(bytes)

MAR register: contains the address of the memory location addressed

MDR: contains either the data to be written to that address or read from that address.

Memory operations

A. Memory read or load:
1. Place memory address to be read into MAR.
2. Issue a Memory_read command to the memory.
3. Data read from the memory is placed into MDR automatically (by control logic).

B. Memory write or store:
1. Place memory address to be written into MAR.
2. Place data to be written into MDR.
3. Issue Memory_write command to the memory.
4. Data in MDR is written to the memory automatically (by control logic).

Instruction types

Computer instructions must be capable of performing 4 types of operations.

1. Data transfer/movement between memory and processor registers
   E.g., memory read, memory write
   example: Move A, B (B = A). (move the value of A into B)
   Move A, R1 (R1 = A).
2. Arithmetic and logic operations:
   E.g., addition, subtraction, comparison between two numbers.
   example: Add A, B, C (C = A + B)

3. Program sequencing and flow of control:
   E.g., Branch instructions
   example: Jump Label (Jump to the subroutine which starts at Label).
4. Input/output transfers to transfer data to and from the real world.
   example: Input PORT, R5 (Read from i/o port “PORT” to register R5).

Specifying operands in instructions

Operands: entities operated by the instructions

Operands may have to be fetched from a memory location to execute an operation.

They may also be stored in the general purpose registers.
Intermediate value of some computation which will be required immediately for subsequent computations.
Registers also have addresses.

Specifying the operands on which the instruction is to operate involves specifying the addresses of the operands.
Address can be of a memory location or a register.

Source and destination operands

Operation may be specified as:
1. Operation source1
2. source2
3. destination

An operand is called a source operand if it's on the the right-hand side of an expression.

An operand is called a destination operand if it's on the the left-hand side of an expression.

example:
Add A, B, C (C = A+B)
A and B are source operands
C is a destination operand.

tldr: result is the destination while givens are source

Operands can be both a source and a destination (example):
Add A, B (B = A + B)
B is both a source and a destination

Instruction types

classified based on the number of operand addresses they include (3,2,1, or 0).

A. 3-address are almost always instructions that implement binary operations.
example: Add A, B, C (C = A + B)
need 3 * k bit to specify the operand addresses.
operand addresses are memory locations that are too big to fit in one word.

B. 2-address instructions(two types)

where one operand serves as a source and destination:
example: Add A, B (B = A + B)
2 * k bits to specify an instruction
This may also be too big to fit into a word.

where at least one operand is a processor register:
example: Add A, R1 (R1 = A + R1)

1-address instructions: require only one operand.
example: Clear A (A = 0)

0-address instructions: do not operate on operands
example: Halt (Halt the computer)

Addressing Modes.

Addressing Modes: Ways in which the operand's address is specified in an instruction

A) Register mode:
- Operand is the contents of a processor register.
- Address of the register is given in the instruction.
- E.g. Clear R1

B) Absolute mode:
- Operand is in a memory location.

- Address of the memory location is given explicitly in the instruction.
- E.g. Clear A
- Also called as “Direct mode” in some assembly languages

The above modes can be used to represent variables

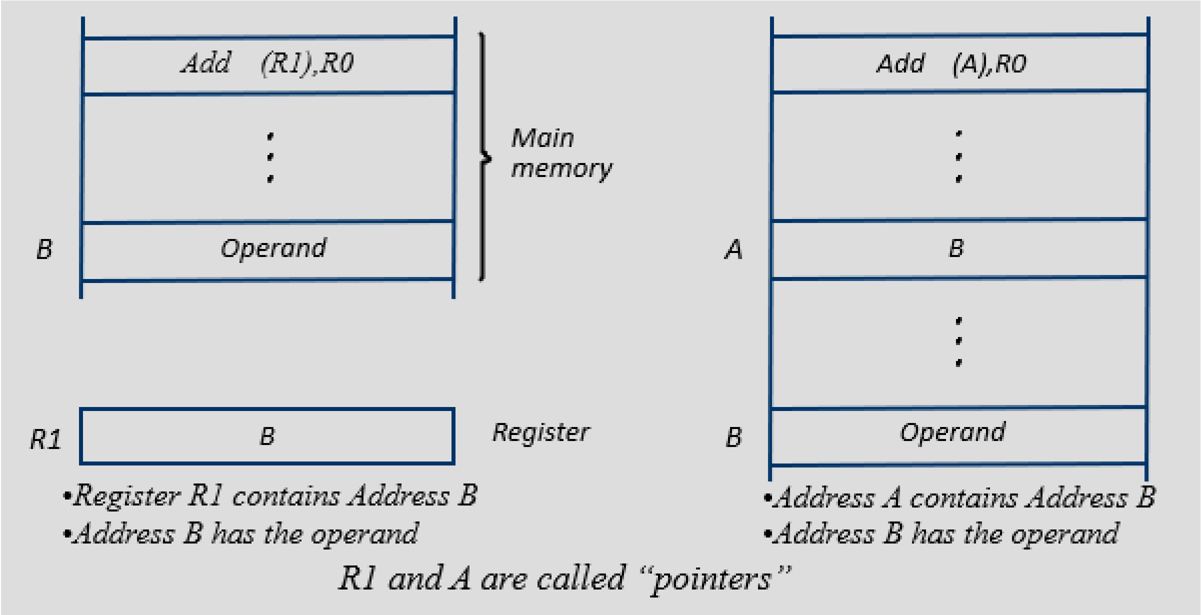
C) Immediate mode:
Operand is given explicitly in the instruction.
E.g. Move #200, R0
Can be used to represent constants.

All 3 modes contained either the address of the operand or the operand itself.

Some instructions provide info from which the memory address of the operand can be determined
they provide the “Effective Address” of the operand.
They do not provide the operand or the address of the operand explicitly.

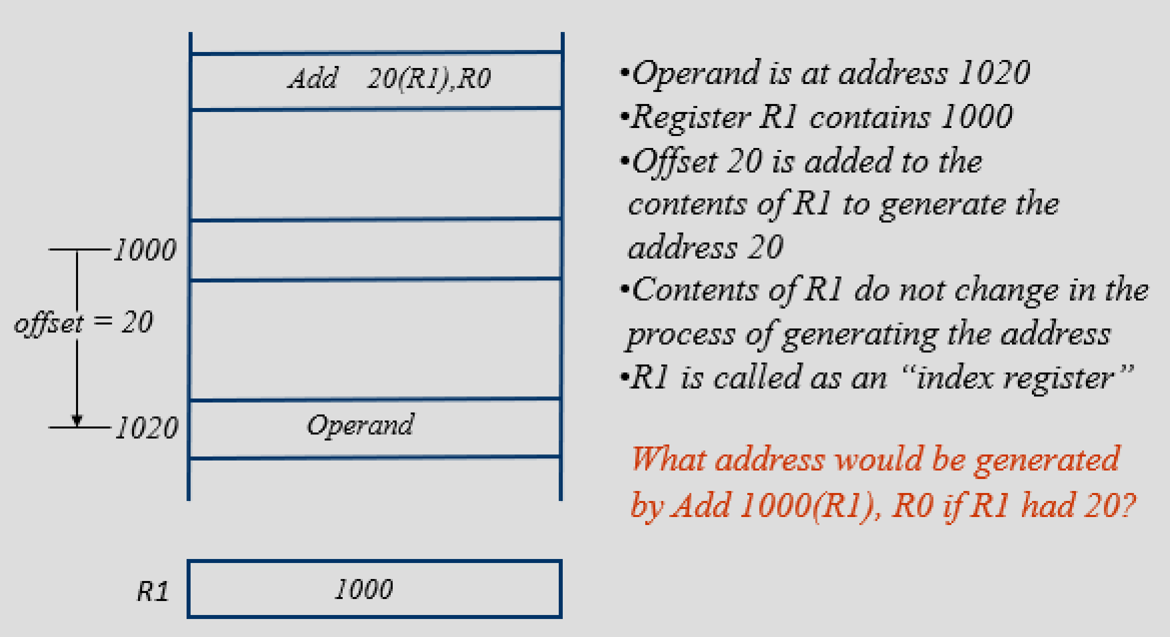
Different ways in which “Effective Address” of the operand can be generated.

A) Effective Address of the operand is the contents of a register or a memory location whose address appears in the instruction (Indirect Mode).


Add (R1),R0
Main memory
B
Operand
R1
B
Register
Add (A),R0
A
B
B
Operand
•Register R1 contains Address B
•Address B has the operand
•Address A contains Address B
•Address B has the operand
R1 and A are called “pointers”

The effective address is the value stored at a memory location or register specified in the instruction.

B) Effective Address of the operand is generated by adding a constant value to the contents of the register (Indexing Mode).



- Operand is at address 1020
- Register R1 contains 1000
- Offset 20 is added to the contents of R1 to generate the address 20
- Contents of R1 do not change in the process of generating the address
- R1 is called as an "index register"

What address would be generated by Add 1000(R1), R0 if R1 had 20?

The effective address is calculated by adding a constant offset to the contents of a specified register.

D) Relative mode
- Effective Address is generated by adding a constant value to the contents of the Program Counter.
- Variation of the Indexing Mode, where the index register is the PC instead of a general purpose register.
- When the instruction is being executed, the PC holds the address of the next instruction in the program.
- Useful for specifying target addresses in branch instructions.

Relative Mode: Addressed location that's "relative" to the PC

E) Autoincrement mode:
- Effective address of the operand is the contents of a register specified in the instruction.
- After accessing the operand, the contents of this register are automatically incremented to point to the next consecutive memory location.
- example: (R1)+

F) Autodecrement mode:
- Effective address of the operand is the contents of a register specified in the instruction.
- Before accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location.
- example: -(R1)

Both the above 2 modes are useful for implementing "Last-In-First-Out" data structures.

Implicitly, The increment and decrement amounts are 1, which allow us to access individual bytes in a byte addressable memory.

as info is stored and retrieved one word at a time, In most computers, increment and decrement amounts are equal to the word size in bytes.

example: If the word size is 4 bytes (32 bits):
Autoincrement increments the contents by 4.
Autodecrement decrements the contents by 4.

Chapter 3

Instruction execution and sequencing

to complete a meaningful task, some instructions need to be executed.

During the fetch/execution cycle of one instruction, The Program Counter (PC) is updated with the address of the next instruction:
PC contains the address of the memory location from which the next instruction is to be fetched.

The contents of the PC point to the next instruction, when the instruction completes its cycle.

A sequence of instructions can be executed to complete a task.

Simple processor model

Processor has a number of general purpose registers.

Word length = 32 bits (4 bytes).

Memory is byte addressable.

Each instruction is one word long.

Instructions allow one memory operand per instruction.
One register operand is allowed + one memory operand.

used for simple tasks:
Add two numbers stored in memory locations A and B.

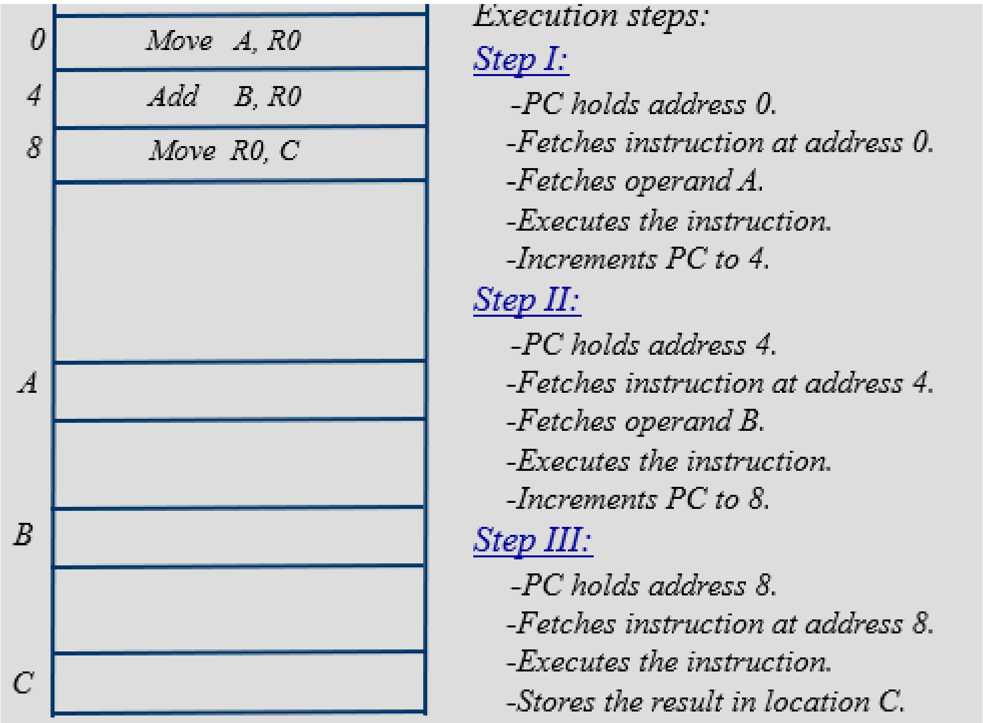
Store the result of the addition in memory location C.

examples:
(Add   B, R0): Add the contents of location B to register R0

(Move A, R0): Move the contents of location A to register R0

(Move R0, C): Move the contents of register R0 to location C

Straight line sequencing example:



Execution steps:

Step I:

-PC holds address 0.
-Fetches instruction at address 0.
-Fetches operand A.
-Executes the instruction.
-Increments PC to 4.

Step II:

-PC holds address 4.
-Fetches instruction at address 4.
-Fetches operand B.
-Executes the instruction.
-Increments PC to 8.

Step III:

-PC holds address 8.
-Fetches instruction at address 8.
-Executes the instruction.
-Stores the result in location C.

Consider the following task:

Add 10 numbers, number of numbers to be added (in this case 10) is stored in location N, numbers are located in the memory at NUM1, ... NUM10, Store the result in SUM.

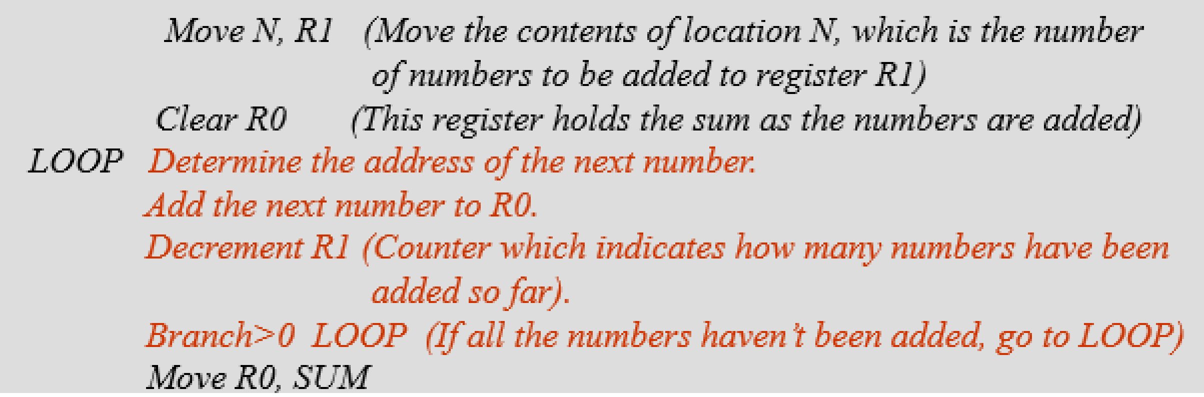
Steps:

1. Move NUM1, R0   (Move the contents of location NUM1 to register R0)
2. Add   NUM2, R0   (Add the contents of location NUM2 to register R0)
3. repeat step2 for the rest of the number (add NUMn, R0)
4. Move R0, SUM   (Move the contents of register R0 to location SUM)

Separate Add instruction to add each number in a list, leading to a long list of Add instructions.

Task can be accomplished in a compact way, by using the Branch instruction.

example:



```
            Move N, R1    (Move the contents of location N, which is the number
                           of numbers to be added to register R1)
            Clear R0      (This register holds the sum as the numbers are added)
LOOP   Determine the address of the next number.
       Add the next number to R0.
       Decrement R1 (Counter which indicates how many numbers have been
                           added so far).
       Branch>0  LOOP  (If all the numbers haven't been added, go to LOOP)
       Move R0, SUM
```

1. **Decrement R1:**
   - R1 initially holds the count NNN of numbers to be added.
   - Decrement R1 each time a new number is added.
   - Keeps track of numbers added so far.
2. **Branch > 0 LOOP:**
   - Checks if R1 is 0.
   - If R1 is 0, store the sum in R0 at memory location SUM (Move R0, SUM).
   - If not, get the next number and repeat (go to LOOP).
   - Note: (Branch > 0 LOOP) implicitly refers to R1.

Processor keeps track of the information about the previous operation's results.

condition code flags: Information recorded in individual bits (Common flags examples):
1. N (negative): set to 1 if result is negative, else cleared to 0
2. Z (zero): set to 1 if result is zero, else cleared to 0
3. V (overflow): set to 1 if arithmetic overflow occurs, else cleared
4. C (carry): set to 1 if a carry-out results, else cleared

condition code register (status register): Grouped together flags in a special purpose register

example

```
If the result of Decrement R1 is 0, then flag Z is set.

Branch> 0, tests the Z flag.

If Z is 1, then the sum is stored.

Else the next number is added.
```

Branch instructions alter the sequence of program execution

The PC holds the address of the next instruction to be executed, we can do so, by loading a new value into the PC.

The Processor fetches and executes instruction at this new address, instead of the instruction located at the location that follows the branch.

Branch target: a new address

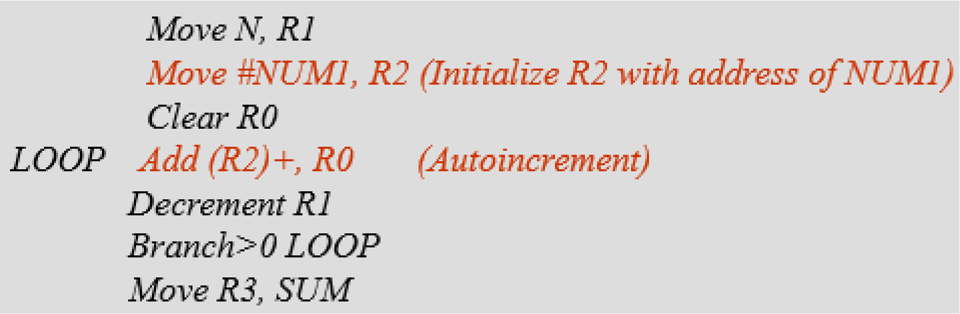
Conditional branch instructions: Instructions that cause a branch only if a specified condition is satisfied.

if there isn't one, the PC is incremented in a normal way, and the next sequential instruction is fetched and executed.

They use condition code flags to check if the conditions are satisfied.

we can do the same using autoincrement mode

example:



```
          Move N, R1
          Move #NUM1, R2 (Initialize R2 with address of NUM1)
          Clear R0
LOOP      Add (R2)+, R0        (Autoincrement)
          Decrement R1
          Branch>0 LOOP
          Move R3, SUM
```

- **Move N, R1**: This instruction loads the number of elements (N) to be summed into register R1, which acts as a loop counter.
- **Move `#NUM1`, R2**: Similar to the first example, this sets R2 to point to the start of the sequence (address of `NUM1`).
- **Clear R0**: Clears the accumulator register R0, setting it to zero, ready to accumulate the sum.
- **LOOP**: A label marking the start of the loop.
- **Add (R2)+, R0**: Adds the value at the address pointed to by R2 to R0 and increments R2 to point to the next memory location.
- **Decrement R1**: Decrements the loop counter R1. If R1 reaches zero, the loop should end.
- **Branch>0 LOOP**: If R1 is greater than zero (meaning the loop isn't finished), control jumps back to the LOOP label to repeat the process.

- **Move R3, SUM**: After the loop finishes, the final sum in R0 is moved to the memory location labeled `SUM`.

Stacks

Stack: A list of data elements, (words or bytes) with a restriction that elements can be added or removed at one end of the stack.

Top of the stack: End from which elements are added.

Bottom of the stack: Other end of the stack.

Other Names for stacks:

1. Pushdown stack.
2. Last in first out (LIFO) stack

Push: placing a new item on the stack.

Pop: Removing the top item from the stack.

Data stored in the computer's memory can be organized as a stack.

Successive elements occupy successive memory locations.

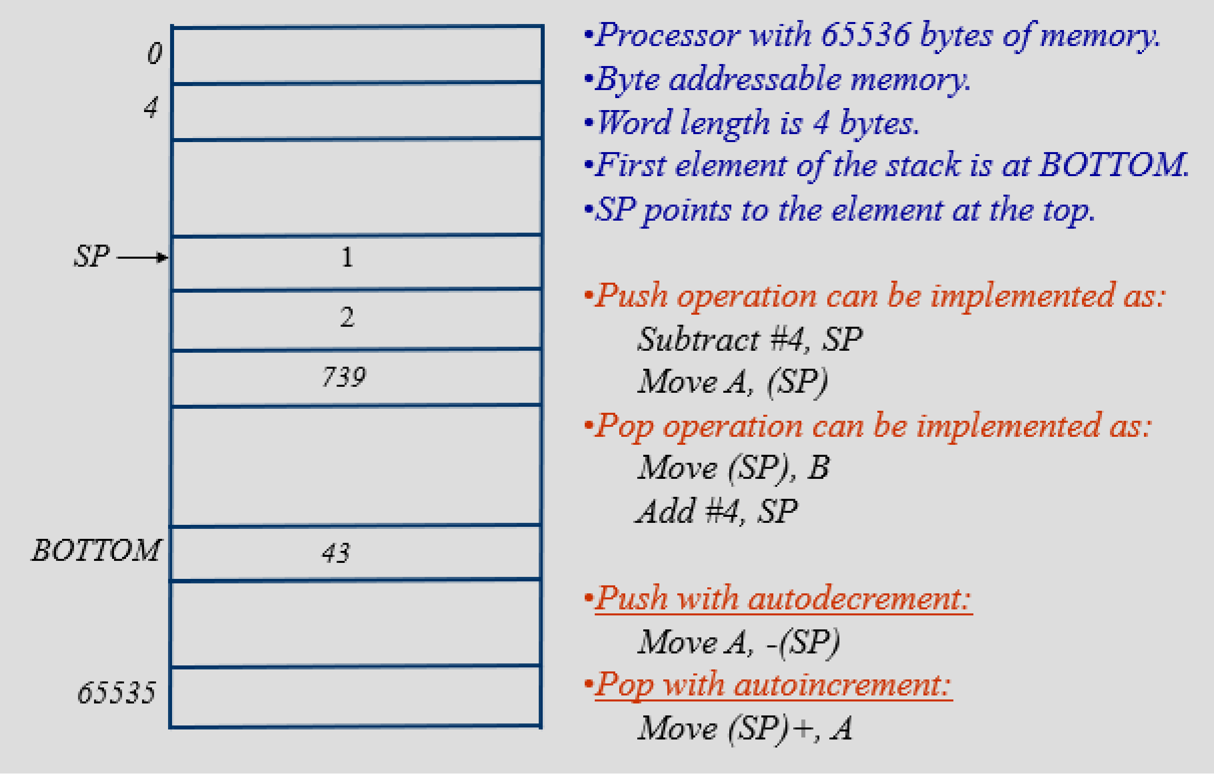
When new elements are pushed on to the stack they are placed in successively lower address locations.

Stack grows in direction of decreasing memory addresses.

Stack Pointer (SP): A processor register used to keep track of the address of the element that is at the top at any given time.

A general purpose register could serve as a stack pointer.

Example:



- *Processor with 65536 bytes of memory.*
- *Byte addressable memory.*
- *Word length is 4 bytes.*
- *First element of the stack is at BOTTOM.*
- *SP points to the element at the top.*

- *Push operation can be implemented as:*
  Subtract #4, SP
  Move A, (SP)
- *Pop operation can be implemented as:*
  Move (SP), B
  Add #4, SP

- *Push with autodecrement:*
  Move A, -(SP)
- *Pop with autoincrement:*
  Move (SP)+, A

Explanation

Memory Details

- The processor has a total of 65,536 bytes (or 64 kilobytes) of addressable memory.
- Each individual byte in memory has a unique address.
- The processor operates on data units of 4 bytes (or 32 bits) at a time.

Stack Characteristics

- The stack grows upwards in memory, with the first element placed at the lowest addressable location, referred to as "BOTTOM."
- The stack pointer (SP) is a register that holds the memory address of the current top element in the stack.

Stack Operations

- **Push operation:**
  - **Subtract #4, SP:** Decrements the stack pointer by 4 bytes to make space for the new element.
  - **Move A, (SP):** Stores the value in register A at the memory location pointed to by SP (the new top of the stack).
- **Pop operation:**

    - **Move (SP), B:** Copies the value at the memory location pointed to by SP (the current top of the stack) into register B.
    - **Add #4, SP:** Increments the stack pointer by 4 bytes, effectively removing the top element from the stack.
- **Push with autodecrement:**
    - **Move A, -(SP):** Decrements the stack pointer by 4 bytes and then stores the value in register A at the new memory location pointed to by SP.
- **Pop with autoincrement:**
    - **Move (SP)+, A:** Copies the value at the memory location pointed to by SP into register A and then increments the stack pointer by 4 bytes.

Additional Notes

- The addresses "1", "2", "739", and "43" represent example memory locations.
- The "BOTTOM" address (65535) indicates the base of the memory space.

Subroutines

Subroutines: subtasks that are repeated on different data values in a program

if a program requires the use of a subroutine, it branches to it.

Calling a subroutine: Branching to the subroutine

Call: Instruction that performs the branch operation.

Subroutine Return: The calling program continuing with executing the instruction immediately after the instruction that called the subroutine after completing the subroutine.

Return: Instruction that performs the above operation.

Subroutine may be called from many places in the program.

How does the subroutine know which address to return to?

- when an instruction is being executed, the PC holds the address of the next instruction to be executed, This is the address to which the subroutine must return, it must be saved by the Call instruction.

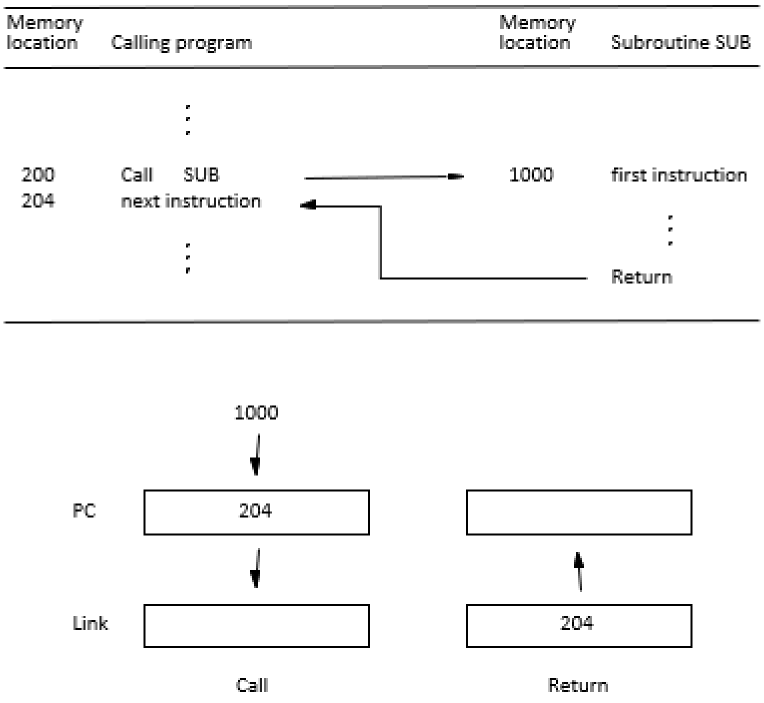
Subroutine linkage method: Way in which a processor makes it possible to call and return from a subroutine.

Link register: a register that stores the return address

Call instruction:
1. Stores the contents of the PC in a link register.
2. Branches to the subroutine

Return instruction:
1. Branch to the address contained in the link register.

Subroutine Example



•Calling program calls a subroutine, whose first instruction is at address 100.
•The Call instruction is at address 200.
•While the Call instruction is being executed, the PC points to the next instruction at address 204.
•Call instructions stores address 204 in the Link register, and loads 1000 into the PC.
•Return instruction loads back the address 204 from the link register into the PC.

1. Memory Layout:

- Two sections of memory: one for the calling program and one for the subroutine.
- The calling program contains instructions like "Call SUB" and "next instruction."
- The subroutine contains its own instructions starting at address 1000.

2. Call Instruction:

- The "Call SUB" instruction is located at memory address 200 in the calling program.
- When executed, the program counter (PC) points to the next instruction at address 204.

- The "Call" instruction stores the address of the next instruction (204) in a special register called the "Link register."
- It then loads the starting address of the subroutine (1000) into the PC.

3. Subroutine Execution:

- The PC now points to the first instruction of the subroutine at address 1000.
- The subroutine executes its instructions.

4. Return Instruction:

- The subroutine ends with a "Return" instruction.
- This instruction loads the address stored in the Link register (204) back into the PC.

5. Returning to Calling Program:

- The PC now points back to the instruction following the "Call SUB" instruction (address 204).
- The program resumes execution from that point.

Subroutines and stack

For nested subroutines
- After the last subroutine in the nested list completes execution, the return address is needed to execute the return instruction.
- This return address is the last one that was generated in the nested call sequence.
- Return addresses are generated and used in a “Last-In-First-Out” order.

- Push the return addresses onto a stack as they are generated by subroutine calls.
- Pop the return addresses from the stack as they are needed to execute return instructions.

Subroutines return and call

SUBROUTINE CALL:

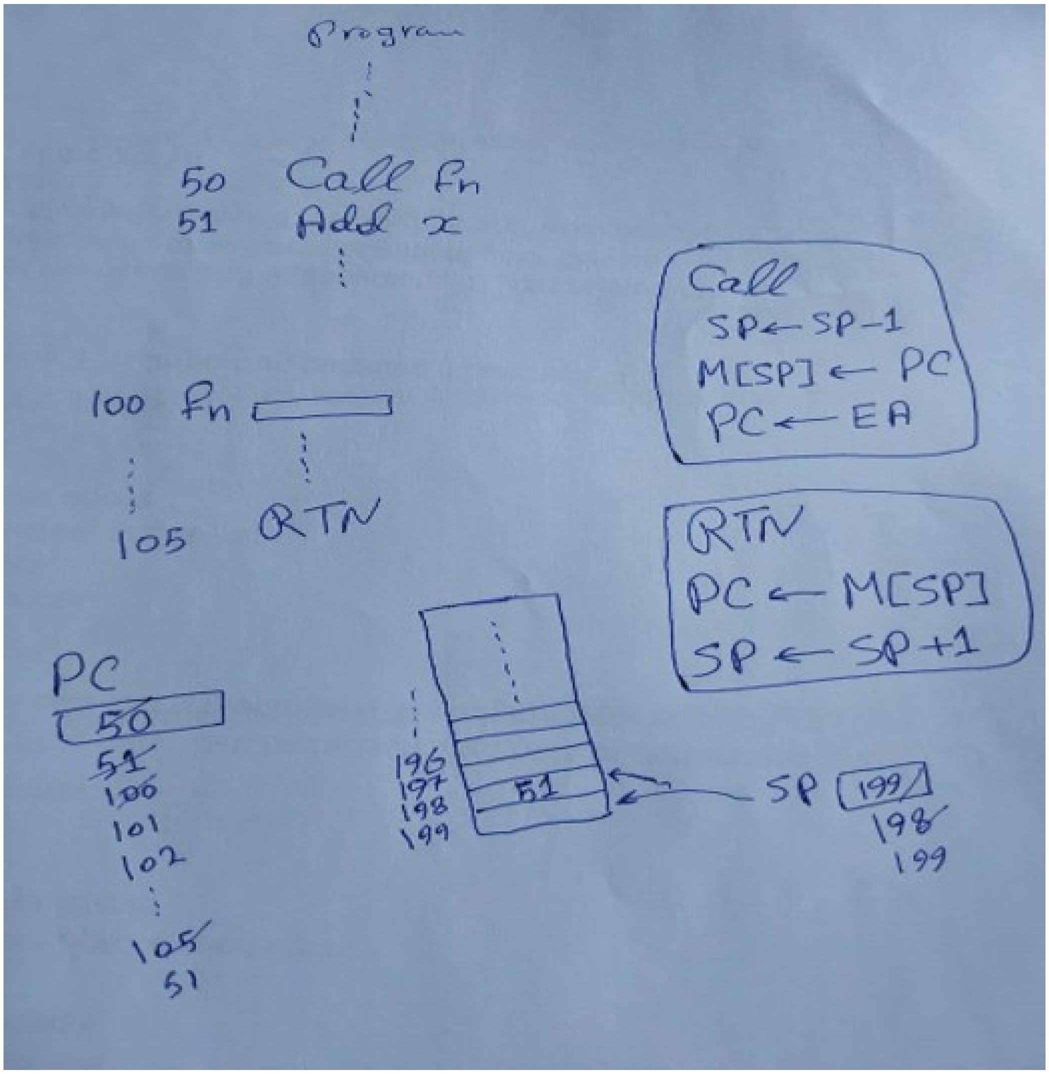
1. Call subroutine
2. Jump to subroutine
3. Branch to subroutine

4. Branch and save return address

Two Most Important Operations are Implied:

- Branch to the beginning of the Subroutine (Same as the Branch or Conditional Branch)
- Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine.
- Locations for storing Return Address:
    1. Fixed Location in the subroutine(Memory)
    2. Fixed Location in memory
    3. In a processor Register
    4. In a memory stack (most efficient way)

Example



Imagine we have program that does program stuff, when it reaches instruction n50 (call fn) and n51 (add x), the program will first go the fn on address 100 and then goes to address 105 (RTN), RTN will then return him to add x, then we have a program counter to store the next instruction to be executed (the address).

we have a stack with addresses 196,197,198,199, the stack pointer point to 199 (on the bottom), when we wanna push for example on 199, the sp then points to 198 and so forth.

the pc has the address of instruction 50 (call), when we wanna execute it, the pc goes to the next instruction's address (51).

when we execute the call instruction, first sp will store sp-1 (if for example sp points to 199, it will then point to 198) (SP <- SP-1)

then (M[SP]<-PC), the memory location pointed to by the stack pointer which is on address will then store the content of pc (which is 51)

then (PC<- EA) the pc will store the content of the Effective Address, which is the address of the subroutine (100).

when the call function is done, the RTN will start

PC <- M[SP] the memory of SP's content (basically the content of where SP point's to((198)) will be stored in the pc (program counter) (in this case 51 will be stored)

(SP <- SP+1) the SP will then point to 199 (pop)

the pc will then perform instruction 51 which is add x

Assembly language

Information is stored in a patterns of 0s and 1s.

Such patterns are awkward when preparing programs.

Symbolic names are used to represent patterns.

When we write programs for a specific computer, the normal words need to be replaced by acronyms called mnemonics.
E.g., MOV, ADD, INC

A complete set of symbolic names and rules for their use constitute a programming language, referred to as the Assembly language.

Programs written in assembly language need to be translated into a form understandable by the computer (examples):
1. Binary
2. machine language form.

Assembler: what Translates from assembly language to machine language.
Source Program: Original program in assembly language.
Object program: Assembled machine language program.

Each mnemonic represents the binary pattern, or OP code for the operation performed by the instruction.

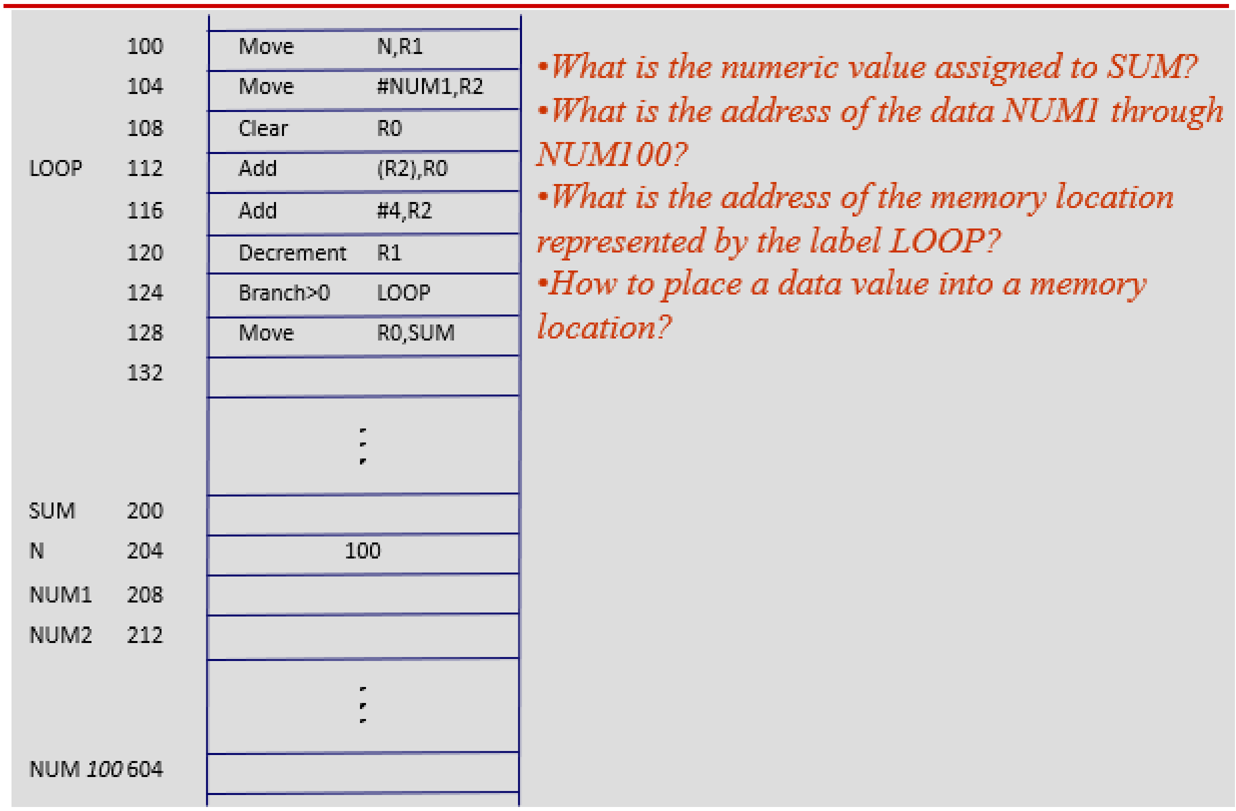
Assembly language must also have a way to indicate the addressing mode being used for operand addresses.

Sometimes the addressing mode is indicated in the OP code mnemonic.
E.g., ADDI may be a mnemonic to indicate an addition operation with an immediate operand.

Assembly language allows programmer to specify other information necessary to translate the source program into an object program (examples).
1. How to assign numerical values to the names.
2. Where to place instructions in the memory.
3. Where to place data operands in the memory.

Assembler directives or commands: statements which provide additional information to the assembler to translate source program into an object program.

Exercise:



	100	Move	N,R1
	104	Move	#NUM1,R2
	108	Clear	R0
LOOP	112	Add	(R2),R0
	116	Add	#4,R2
	120	Decrement	R1
	124	Branch>0	LOOP
	128	Move	R0,SUM
	132		
		⋮	
SUM	200		
N	204	100	
NUM1	208		
NUM2	212		
		⋮	
NUM *100*	604		

•What is the numeric value assigned to SUM?

•What is the address of the data NUM1 through NUM100?

•What is the address of the memory location represented by the label LOOP?

•How to place a data value into a memory location?

1. **What is the numeric value assigned to SUM?**

   The value assigned to `SUM` in the memory is `200`, as shown in the table. This is the memory address where the result of the operations will be stored.

2. **What is the address of the data NUM1 through NUM100?**

   - `NUM1` is located at address `208`.
   - `NUM2` is located at address `212`.
   - Based on the pattern, the address of `NUM100` would be calculated as:

   text

   Copy code

   ```
   Address of NUM100 = 208 + (100-1) × 4 = 208 + 396 = 604
   ```

   So, `NUM100` is located at address `604`.

3. **What is the address of the memory location represented by the label LOOP?**

   The label `LOOP` corresponds to the instruction at address `112`.

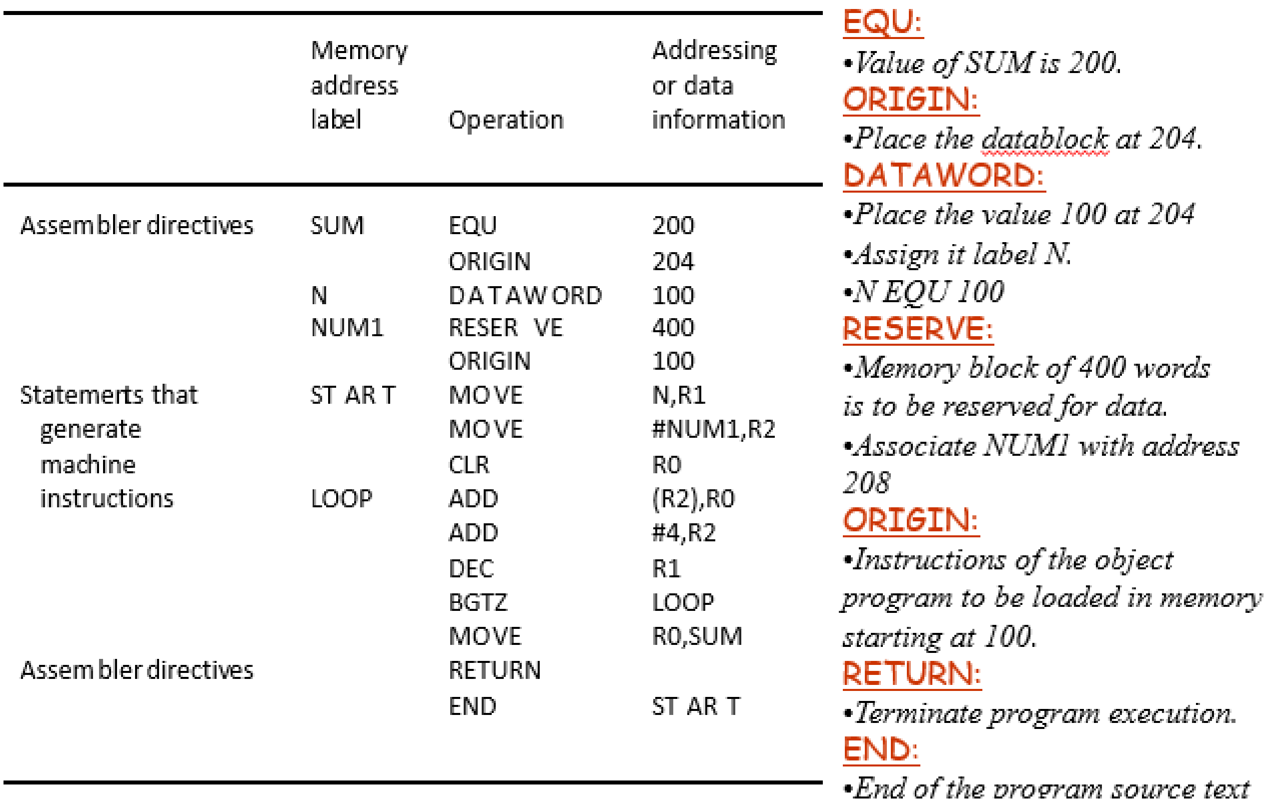
4. **How to place a data value into a memory location?**

   To place a data value into a memory location, you use the `Move` instruction. For example:

- `Move R0,SUM` will store the value in register `R0` into the memory location labeled as `SUM`.
- `Move #NUM1,R2` will move the immediate value `NUM1` into register `R2`.

Example: Memory Addressing and Assembler Directives

This example illustrates how various assembler directives and statements generate machine instructions, particularly focusing on memory addressing. The table explains the operation of each directive and instruction, showing the corresponding memory address labels, operations, and addressing or data information.



	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statemerts that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

EQU:
•*Value of SUM is 200.*
ORIGIN:
•*Place the datablock at 204.*
DATAWORD:
•*Place the value 100 at 204*
•*Assign it label N.*
•*N EQU 100*
RESERVE:
•*Memory block of 400 words is to be reserved for data.*
•*Associate NUM1 with address 208*
ORIGIN:
•*Instructions of the object program to be loaded in memory starting at 100.*
RETURN:
•*Terminate program execution.*
END:
•*End of the program source text*

Explanation:

- **EQU:** Defines the value of `SUM` as 200.
- **ORIGIN:** Specifies that the `datablock` starts at memory location 204.
- **DATAWORD:** Assigns the value 100 to the label `N`, located at address 204.
- **RESERVE:** Reserves a memory block of 400 words, starting at address 208 for `NUM1`.
- **LOOP:** The loop demonstrates a basic sequence of instructions involving memory operations and a branch instruction that loops until a condition is met.
- **RETURN:** Marks the end of the program execution.

- **END:** Signifies the end of the program source text.

Assembly language (continued)

Assembly language instructions have a generic form:

- Label Operation Operand(s) Comment

Four fields are separated by a delimiter, typically one or more blank characters.

Label is optionally associated with a memory address:

May indicate the address of an instruction to be executed and the address of a data item.

How does the assembler determine the values that represent names?

Value of a name may be specified by EQU directive.

•SUM EQU 100

A name may be defined in the Label field of another instruction, value represented by the name is determined by the location of that instruction in the object program.

•E.g., BGTZ LOOP, the value of LOOP is the address of the instruction ADD (R2) R0

Assembler scans through the source program, keeps track of all the names and corresponding numerical values in a "symbol table".

When a name appears second time, it is replaced with its value from the table.

What if a name appears before it is given a value, for example, branch to an address that hasn't been seen yet (forward branch)?

- Assembler can scan through the source code twice.
- First pass to build the symbol table.
- Second pass to substitute names with numerical values.
- Two pass assembler.

Encoding of machine instructions

Instructions specify the operation to be performed and the operands to be used.

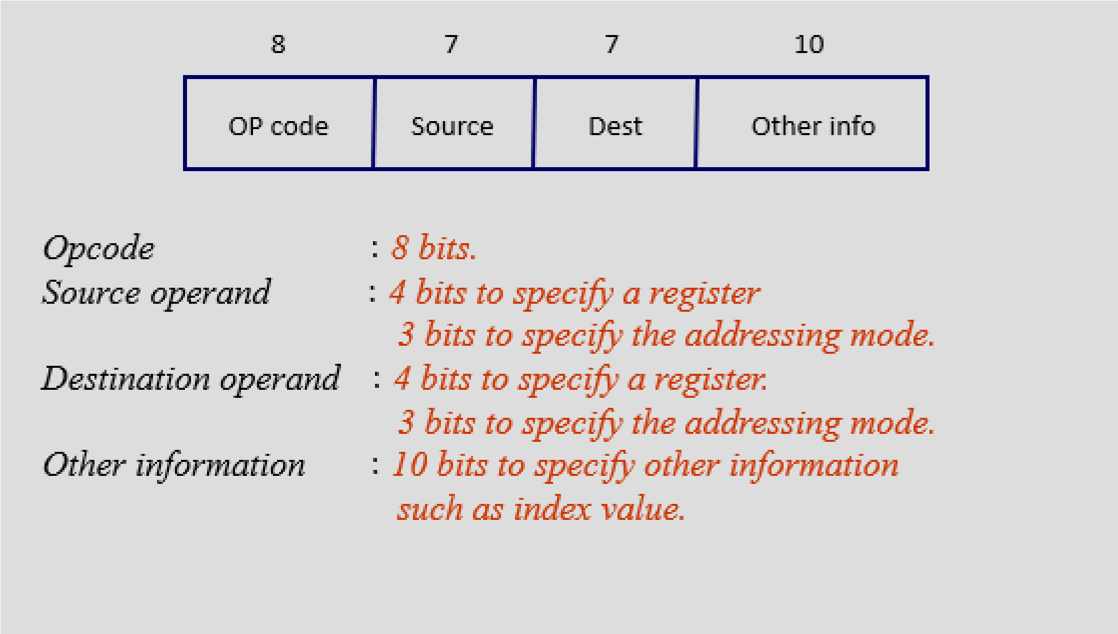
OP code: Which operation is to be performed and the addressing mode of the operands.

Consider a processor with:

1. Word length 32 bits.
2. 16 general purpose registers, requiring 4 bits to specify the register.
3. 8 bits are set aside to specify the OP code.
4. 256 instructions can be specified in 8 bits.

Example 1

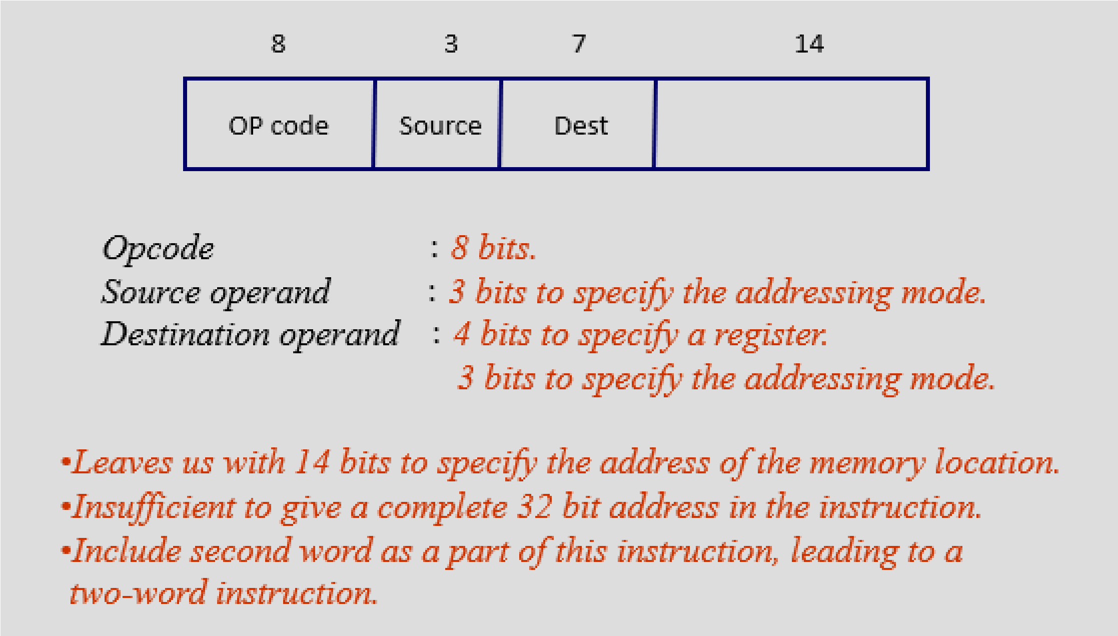
One-word instruction format.



8	7	7	10
OP code	Source	Dest	Other info

Opcode : *8 bits.*
Source operand : *4 bits to specify a register*
3 bits to specify the addressing mode.
Destination operand : *4 bits to specify a register.*
3 bits to specify the addressing mode.
Other information : *10 bits to specify other information*
such as index value.

Example 2

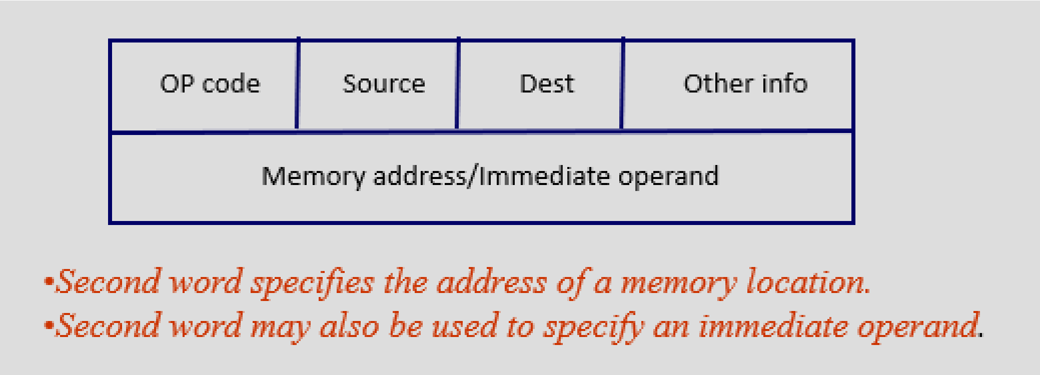


8	3	7	14
OP code	Source	Dest	

Opcode : *8 bits.*
Source operand : *3 bits to specify the addressing mode.*
Destination operand : *4 bits to specify a register.*
3 bits to specify the addressing mode.

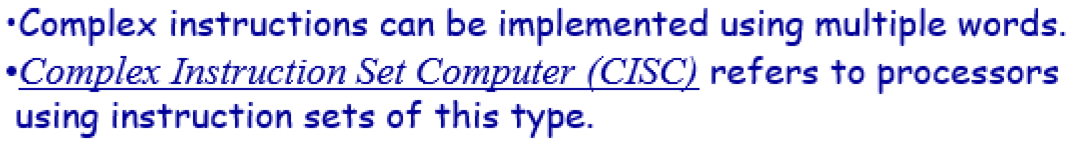
- *Leaves us with 14 bits to specify the address of the memory location.*
- *Insufficient to give a complete 32 bit address in the instruction.*
- *Include second word as a part of this instruction, leading to a two-word instruction.*

Source operand is 3 bits cause it's specified using direct addressing mode



OP code	Source	Dest	Other info
Memory address/Immediate operand			

- *Second word specifies the address of a memory location.*
- *Second word may also be used to specify an immediate operand.*



- Complex instructions can be implemented using multiple words.
- *Complex Instruction Set Computer (CISC)* refers to processors using instruction sets of this type.

One word for the first pic, 2nd word for the 2nd pic

Instruction Codes

Program: A set of instructions that specify the operations, operands, and the sequence by which processing has to occur

Instruction Code: A group of bits that tells the computer to perform an operation

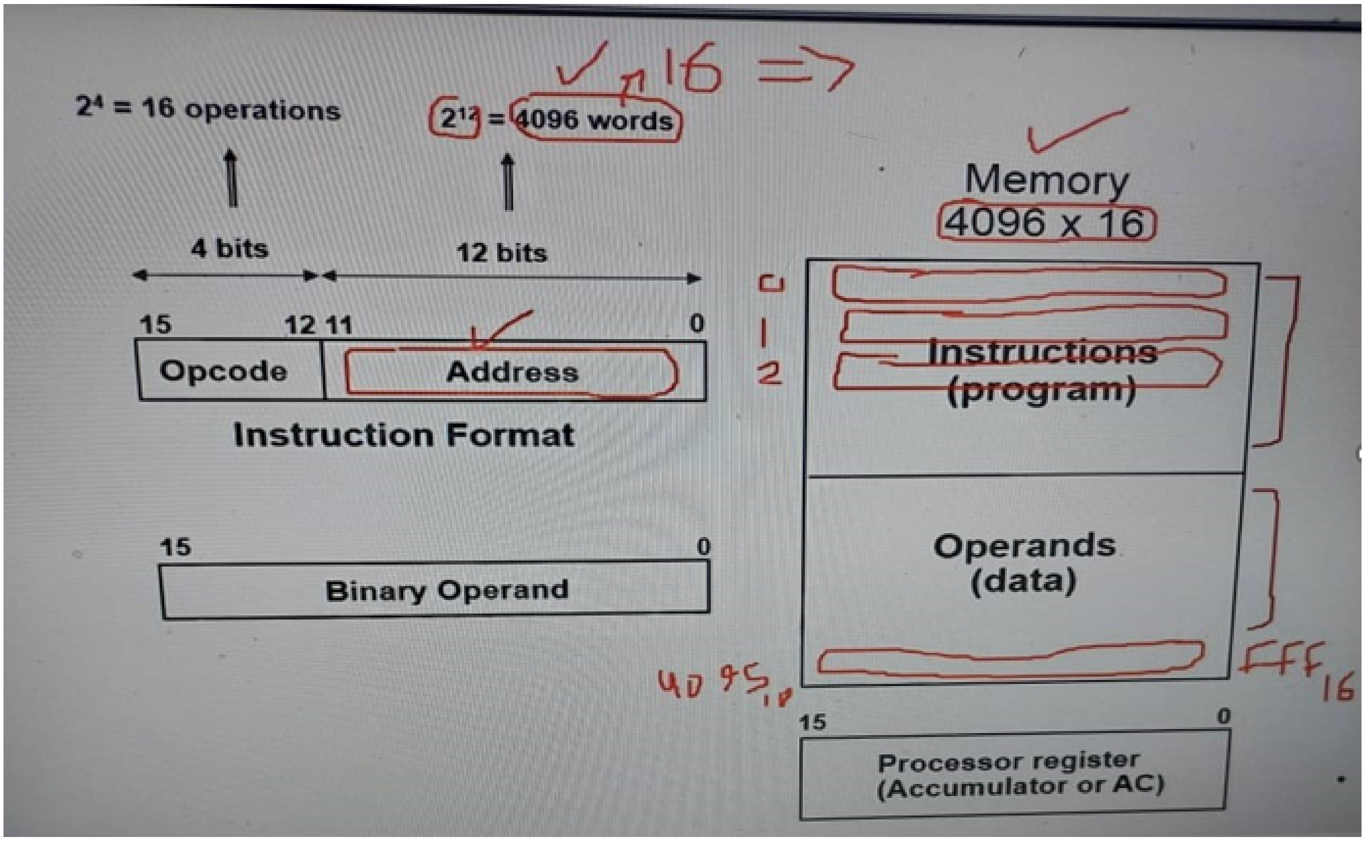
Instruction Code Format

Opcode	Operand (address)

Operation code: A group of bits that define the operation (add, subtract, shift)

- The operation (or macrooperation) specifies a set of microoperations

Operand (or address of operand): Data stored in registers or in the memory, on which the operation is to be performed.



we have a register with 16 bits (from 0-15), from 0-11 is for the address, and the rest for the opcode, subtract 1 bit from the opcode for other stuff, leaving it with 3 bits, where we can do 8 operations with it, we also have 4096 locations in the memory (as we have 12 bits for the address).

the memory is divided into 2 parts, one part for the program and another for the operand, and the memory is divide from word$_0$ to word$_{4095}$

we also have a processor register (or accumulator), it's one of the most important registers in the processor.

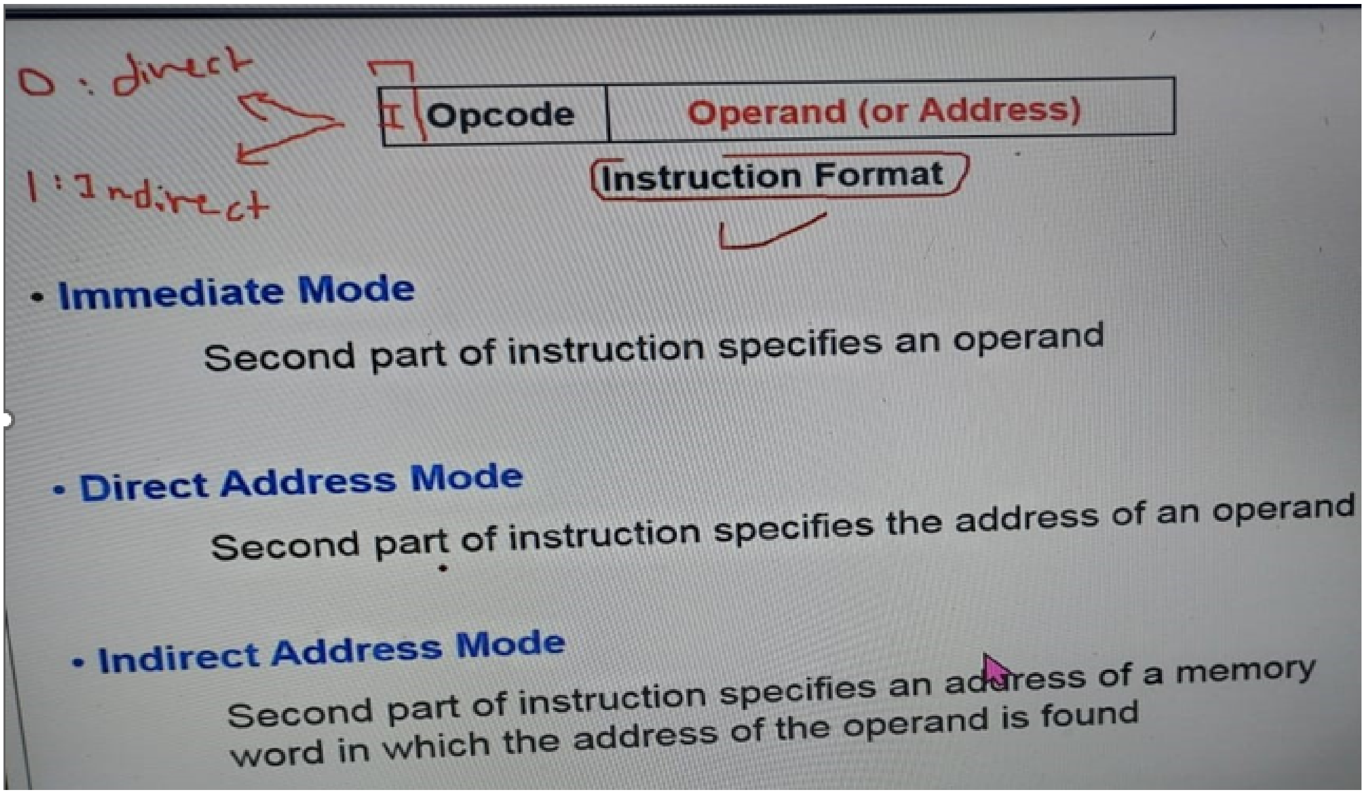
What is an accumulator in the context of computing and programming?

```
Accumulator: a register, or a memory location used to store the intermediate
results of arithmetic and logical operations. It plays a crucial role in
performing calculations and processing data in various applications.
```

Why is the accumulator important in computing?

```
The accumulator serves as a temporary storage location for arithmetic
operations,  (CPU) to perform complex calculations.
It simplifies the processing of multiple operations, enhancing the efficiency
and speed of the computing system.
```

Direct vs Indirect Addressing modes


0 : direct
1 : Indirect

I	Opcode	Operand (or Address)

Instruction Format

- **Immediate Mode**

  Second part of instruction specifies an operand

- **Direct Address Mode**

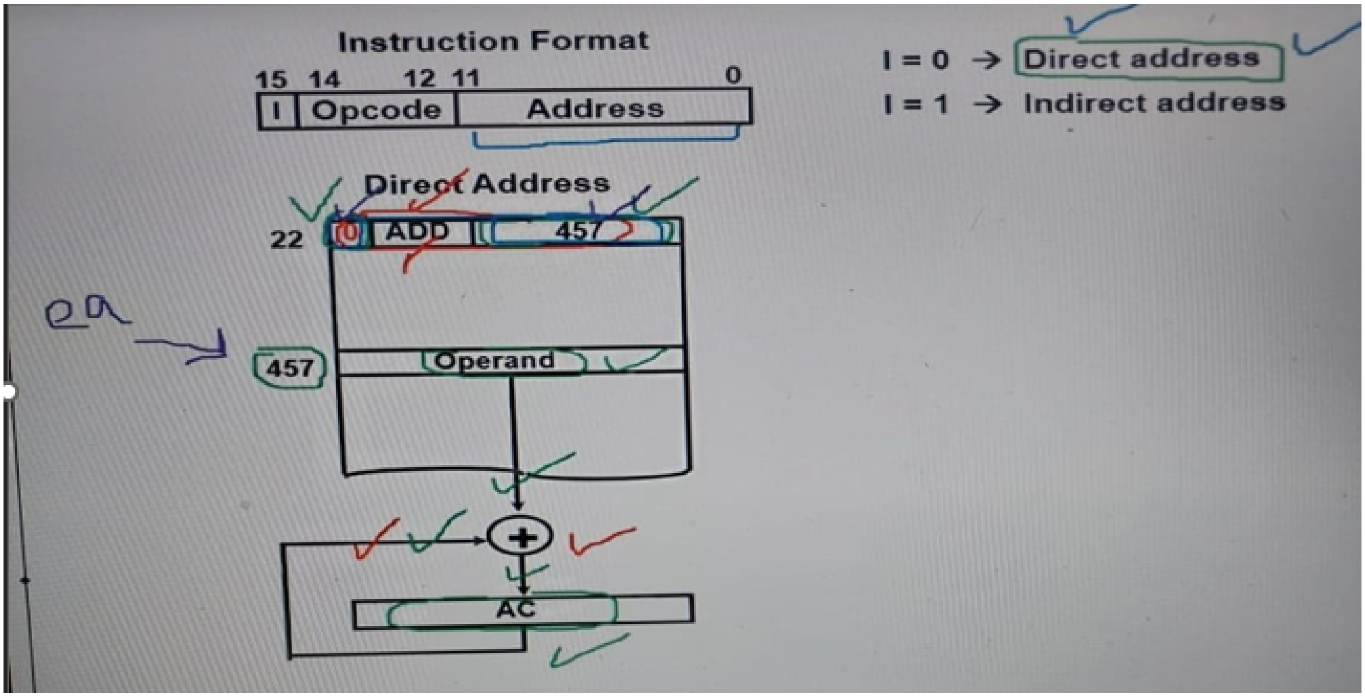
  Second part of instruction specifies the address of an operand

- **Indirect Address Mode**

  Second part of instruction specifies an address of a memory word in which the address of the operand is found

The last bit in the instruction represents a mode, either Direct (0) or Indirect (1) (It's called immediate mode)

Example 1



Address = 457

Opcode = ADD

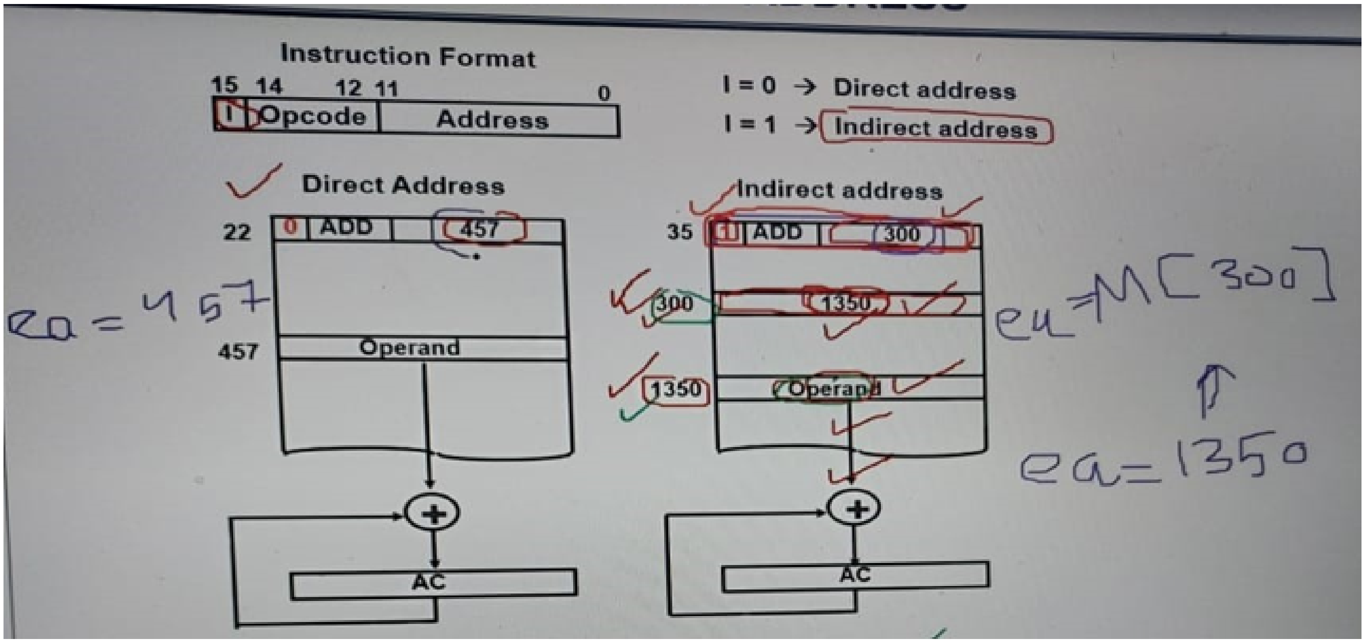
I = 0 (direct)

we go to 457 and find the instruction we need (in this case it's add), the operand then does the adding, and enters the ac which gives us the final result

final result is: AC <-- AC + M[457]

457 is the effective address (basically the address that has the operand)

Example 2



Direct Part is the same as the above

Indirect Part

Address = 300

Operand = add

IA = 1 (Indirect)

we first go to 300, where we find 1350, if it was direct it would contain an operand, when we go to 1350 we find the operand.

basically indirect mode we have the address of an address of an operand

the operand is add so we basically add the ac to the memory location of the operand and save it in the ac

AC <-- AC + M[M[300]] (the memory of 300's memory) which equals 1350

so the final answer is AC <-- AC+ M[1350] or AC <-- AC+ Operand

ea = 1350

Effective Address(EFA, EA): The address of the operand in a computation-type instruction or the target address in a branch-type instruction

Encoding of machine instructions (contd..)

Insist that all instructions must fit into a single 32 bit word:

- Instruction cannot specify a memory location or an immediate operand.
- ADD R1, R2 can be specified.
- ADD LOC, R2 cannot be specified.
- Use indirect addressing mode: ADD (R3), R2
- R3 serves as a pointer to memory location LOC.

How to load address of LOC into R3?

- Relative addressing mode.

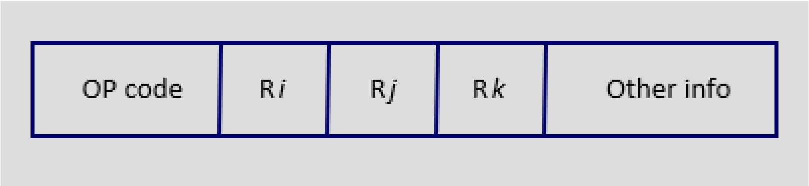
Reduced Instruction Set Computers (RISC): Restriction that an instruction must occupy only one word.

Manipulation of data must be performed on operands already in processor registers.

Restriction may require additional instructions for tasks.

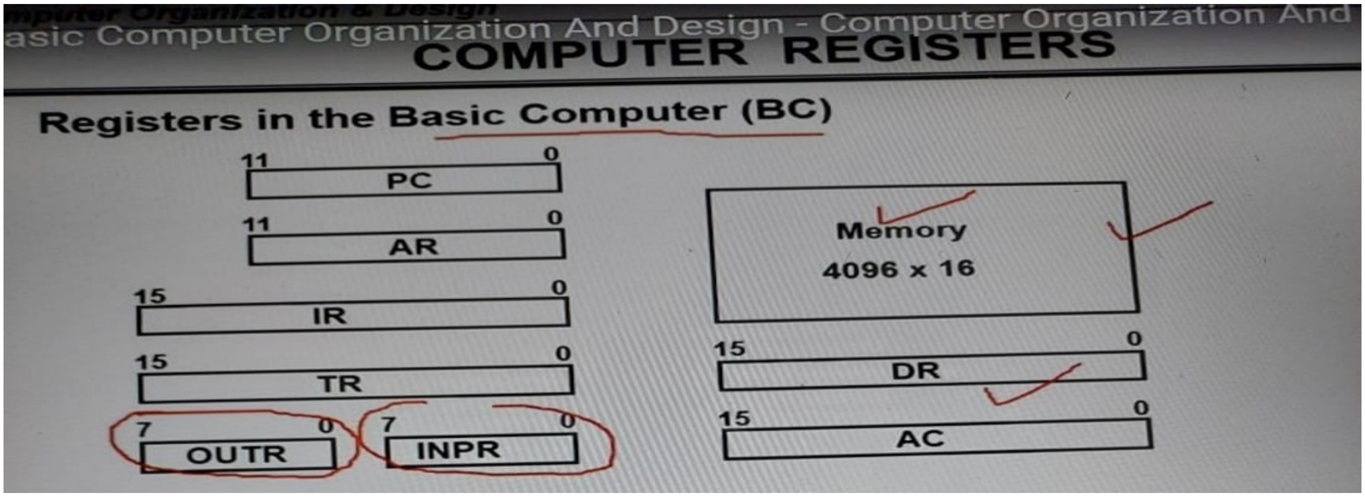
However, it is possible to specify three operand instructions into a single 32-bit word, where all three operands are in registers:

Three-operand instruction



Chapter 4

Computer in basic register


asic Computer Organization And Design - Computer Organization And
COMPUTER REGISTERS
Registers in the Basic Computer (BC)
11
PC
0
11
AR
0
15
IR
0
15
TR
0
7
OUTR
0
7
INPR
0
Memory
4096 x 16
15
DR
0
15
AC
0

List of BC Registers

Symbol	num of bit	Name	Function
DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of next instruction
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Ouput Register	Holds output character

Memory = 4096 (2^{12}) word, from word 0 to word 4095, each word contains 16 bits

some registers are 16 bits and some are 12 bits and some are 8 bits

16 bits are registers that deal with the memory (as the word length = 16 bits).

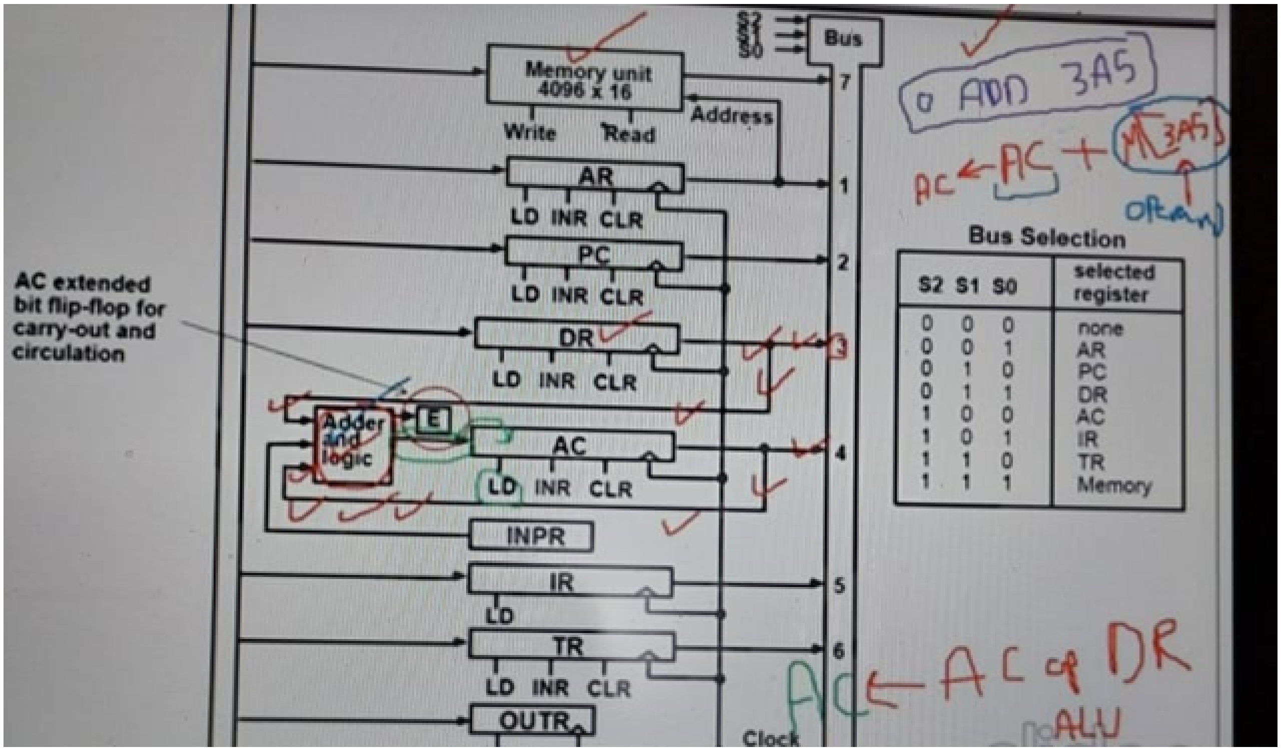
12 bits are registers that deal with the address (as the address length equals $\log_2(4096) = 12$)

Note: (the above bit size is handled depending on the memory)

8 bits registers are dealt with characters, that have 8 bits typically

---

Common Bus System



Bus Selection

S2	S1	S0	selected register
0	0	0	none
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

The above drawing depicts a memory unit with 4096 word, each word has 16 bits, below it in order is the Address Register, Program Counter, Data Register, Accumulator, Input Register, Instruction Register, Temporary Register and Finally the Output Register.

we also have an adder that does mathematical operations near the AC, the memory and register must be connected using a bus, that covers from and to all registers, with all registers and the memory unit sending data from it, which lets all registers communicate with each other

for example when the memory unit sends something in it (whether it is operand or address etc) (symbolized using the out arrow), the bus moves around, and when it reaches the other side (symbolized using the in arrow), the data goes inside the register

there are 7 registers that send data to the bus, but not all send the data at the same time, only one at a time

we pick a register based on which signals are active with if none of them are active no register sends data (depending on the instruction)

s_2	s_1	s_0	selected register
0	0	0	None
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC

s_2	s_1	s_0	**selected register**
1	0	1	IR
1	1	0	TR
1	1	1	Memory

the adder is linked with the AC and vice versa

we have a bit called E, it's size is 1 bit, and called an extended bit, it's an AC extended bit flip-flop for carry-out and circulation.

the instruction (0 add 3A5) means the following

0 means it's a direct addressing mode

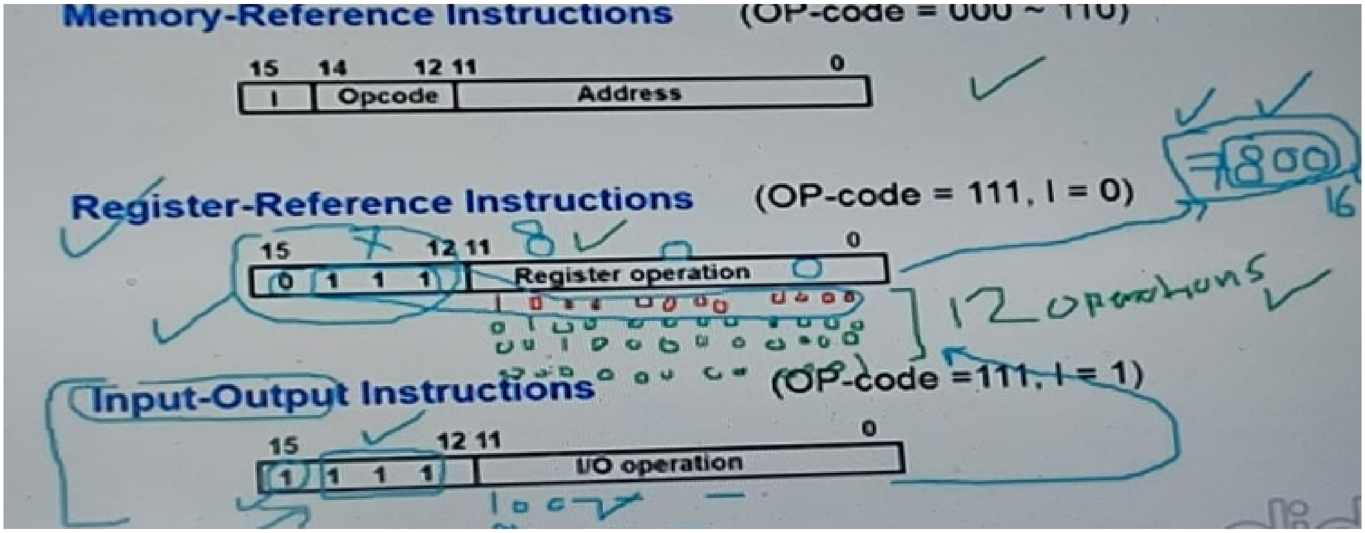
the Add.. means add the following

The 3A5 (hexadecimal) is the address which contains the operand

the computer goes to the address in the memory, the operand then goes to the data register, and then goes to the AC which contains a value, and when the loading operation starts the adding operation starts as well

Final Solution: AC <-- AC +M[3A5]

Basic Computer Instruction Code Formats



1. Memory-Reference Instruction:

0-11 is the address, which contains the address of the operand or the EA

12-14 contains the operations which happen on the operands

15 is the bit which implies the addressing mode

opcode is only from 0 to 6 (7 operations)

2. Register-Reference Instruction:

0-11 Register Operation

12-14 Opcode

15 is always 0

here we have 4096 operations, from which we only need 12, to keep the register format's fixed, (we can add more)

operation example

bit0	**bit1**	**bit2**	**bit3**	**bit4**	**bit5**	**bit6**	**bit7**	**bit8**	**bit9**	**bit10**	**bit11**
1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1

Op code = 111 I=0 = 7

3. Input/Output Instruction

Op code = 111 / Indirect Addressing Mode (I=1) = F

0-11 I/O operations

operations

bit0	bit1	bit2	bit3	bit4	bit5	bit6	bit7	bit8	bit9	bit10	bit11
1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1

Basic Operation Instruction

Memory-Reference Instructions

Num	Symbol	HEX Code IF I=0	HEX Code IF I=1	Description
1	AND	0xxx	8xxx	AND memory word to AC
2	ADD	1xxx	9xxx	Add memory word to AC
3	LDA	2xxx	Axxx	Load AC from memory
4	STA	3xxx	Bxxx	Store content of AC into memory
5	BUN	4xxx	Cxxx	Branch unconditionally
6	BSA	5xxx	Dxxx	Branch and save return address
7	ISZ	6xxx	Exxx	Increment and skip if zero

Register-Reference Instructions

	Symbol	HEX Code (I=0)	Description
1	CLA	7800	Clear AC
2	CLE	7400	Clear E
3	CMA	7200	Complement AC

	Symbol	HEX Code (I=0)	Description
4	CME	7100	Complement E
5	CIR	7080	Circulate right AC and E
6	CIL	7040	Circulate ;eft AC and E
7	INC	7020	Increment AC
8	SPA	7010	Skip next instr. if AC is positive
9	SNA	7008	Skip next instr. if AC is negative
10	SZA	7004	Skip next instr. if AC is zero
11	SZE	7002	Skip next instr. if E is zero
12	HLT	7001	Half Computer

Input/Output Instructions

Num	Symbol	HEX Code (I=1)	Description
1	INP	F800	Input character to AC
2	OUT	F400	Output character from AC
3	SKI	F200	Skip on input flag
4	SKO	F100	Skip on output flag
5	ION	F080	Interrupt on
6	IOF	F040	Interrupt off

How to solve a problem using them

1. Convert The number into binary (to figure out if the addressing mode)
2. look at the first number (either in hex or binary) and based on the above table figure out the instruction
3. write how the instruction will look like

Examples

153B

0001 0101 0011 1011

Direct Address

ADD

AC <- AC + M[53B]

24B4

0010 0100 1011 0100

LDA

Direct Address

AC <- M[4B4]

A3B4

1010 0011 1011 0100

Indirect Address

LDA

AC <- M[M[3B4]]

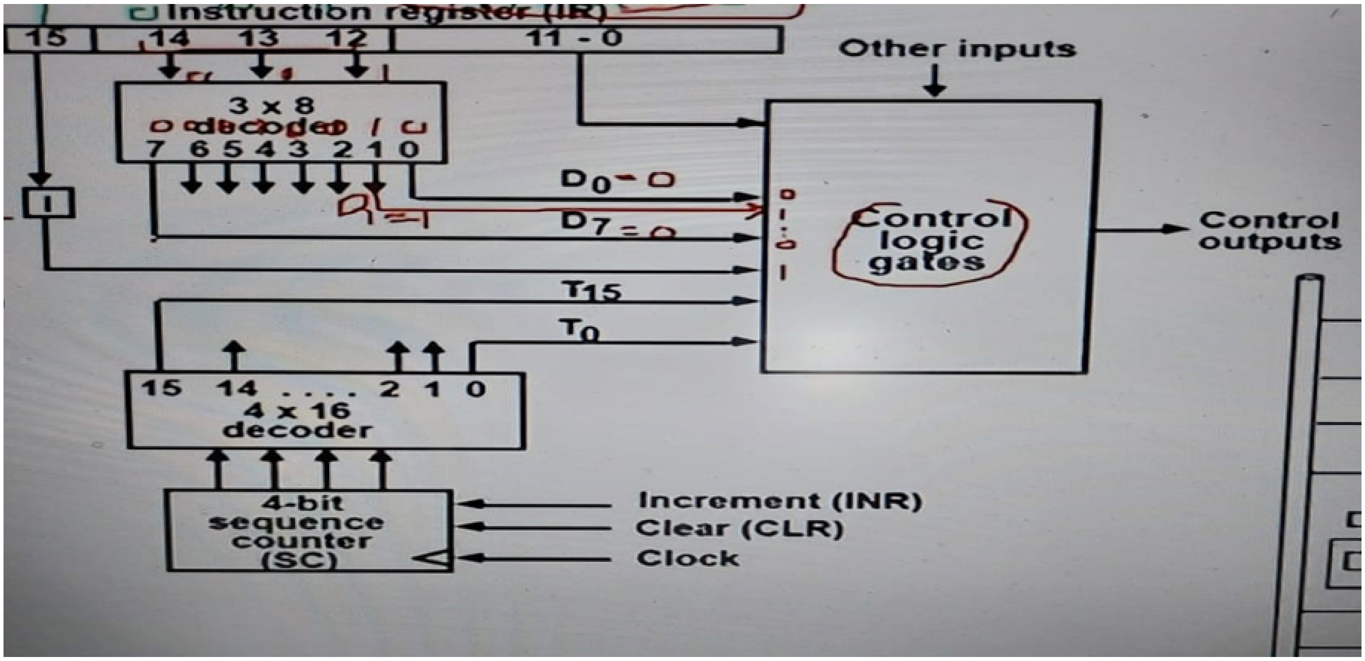
Instruction set Completeness

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

Instruction Types

1. Functional Instructions
    - Arithmetic, logic, and shift instructions
    - ex: ADD, CMA, INC, CIR, CIL, AND, CLA
2. Transfer Instructions
    - Data transfers between the main memory and the processor registers
    - ex: LDA, STA
3. Control Instructions
    - Program sequencing and control
    - ex: BUN, BSA, ISZ
4. Input/Output Instructions
    - Input and output
    - ex: INP, OUT

Timing and Control



when we wanted to do an operation, it was performed when we had a signal, this circuit is responsible for signals, they enter the clock, and generate signals

below we have a 4-bit sequence counter, since it's 4 bits it counts from 0 to 15

we also have a decoder, where 4 bits enter and 16 bit are outputted, but only one of them can be active (1) at a time.

in the sc we have an increment, when for example starts with 0, all bits will be zero, and the t0 will have a value of 1, when the clock and INR increments the counter by 1, the counter will be 1 and t1 will be 0, and then it will become 2, and so forth, until we reach 15 with t15 becoming 1.

when we add an extra 1 the counter will return to 0, because when adding 1 to 1111 to will overflow and we will have 0000 and 1 in hand

the control logic gates accepts the active timing at a time

when clear is on (or valid or true), it clears the counter (doesn't have to be 15, it could be at anytime)

the signals enter through the clock into the SC

if the clear doesn't reset the SC, then it will be reset when it reaches 15 the 2nd input the enters the control logic gate is the register, the first 12 bits are either Register operation or I/O operation, or an address (of operand of instruction), which are an input to the Logic Gate

the 2nd input from the register in the 3 bits 12-14, that enter the decoder, which gives us 8 outputs from the decoder (from d0 to d7)

the final bit has it's own input

Question: when does an SC get reset
Ans: when the counter is full (1111)

Question: in a register's decoder, when does the 7 become a 1 (in on)
Ans: the 7 becomes on when the IR is either a register operation or an I/O operation

Example:

Say we have $(93A5)_{16}$

the input for the control logic gate will be ?

93A5

1001 0011 1011 0101

1001

1 goes to bit 15

001 goes to bits 14,13,12 in that order

0011 1011 0101 fills up from 0-11 (an address) (indirect as bit 15=1)

D1 will be on (1) and the rest will be off (0)

bits from 0-11 will enter the gate as is, as address of address of operand

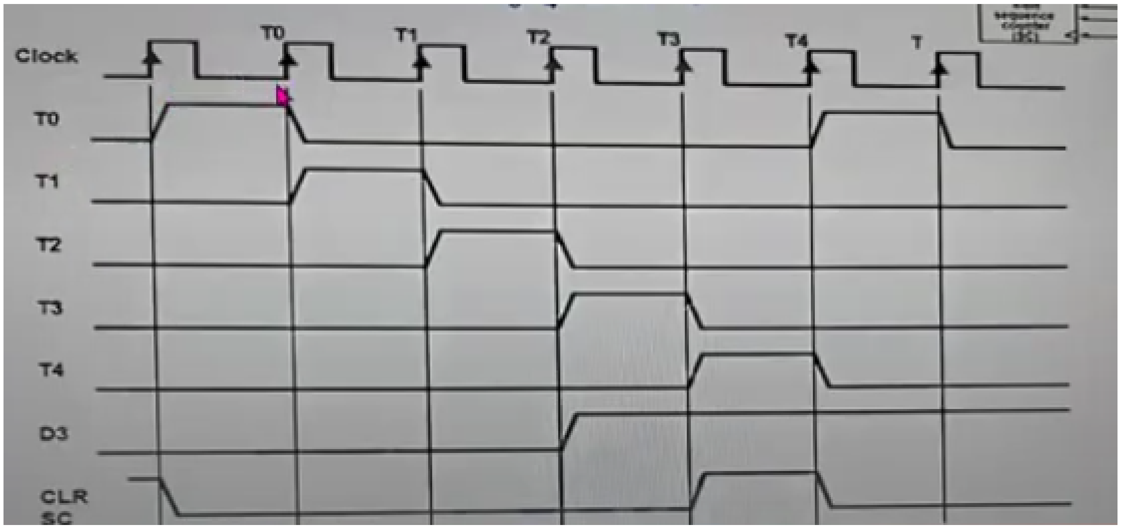
the signals that's coming from the generator looks like a wave, when it's down the value is 0, and when it's up the value is 1

Timing Signals

- generated by a 4 bit sequence counter and a 4 * 16 decoder
- The SC can be incremented or cleared

Example: $T_0T_1T_2T_3T_4T_0T_1$......

Assume: at time T_4 SC is cleared to 0 if decoder output D3 is active



The timing's on the picture move with the clock, if the wave rises then that means that it's signal is 1, and if the waves falls it means it's signal is 0

based on what we assumed if D3 rises the SC's value is reset to 0 when we reach T4
D_3T_4: SC<-0

Important NOTE

we stopped at slide 31, so I can't explain the rest of the chapter

---

Chapter 5

Instruction Cycle

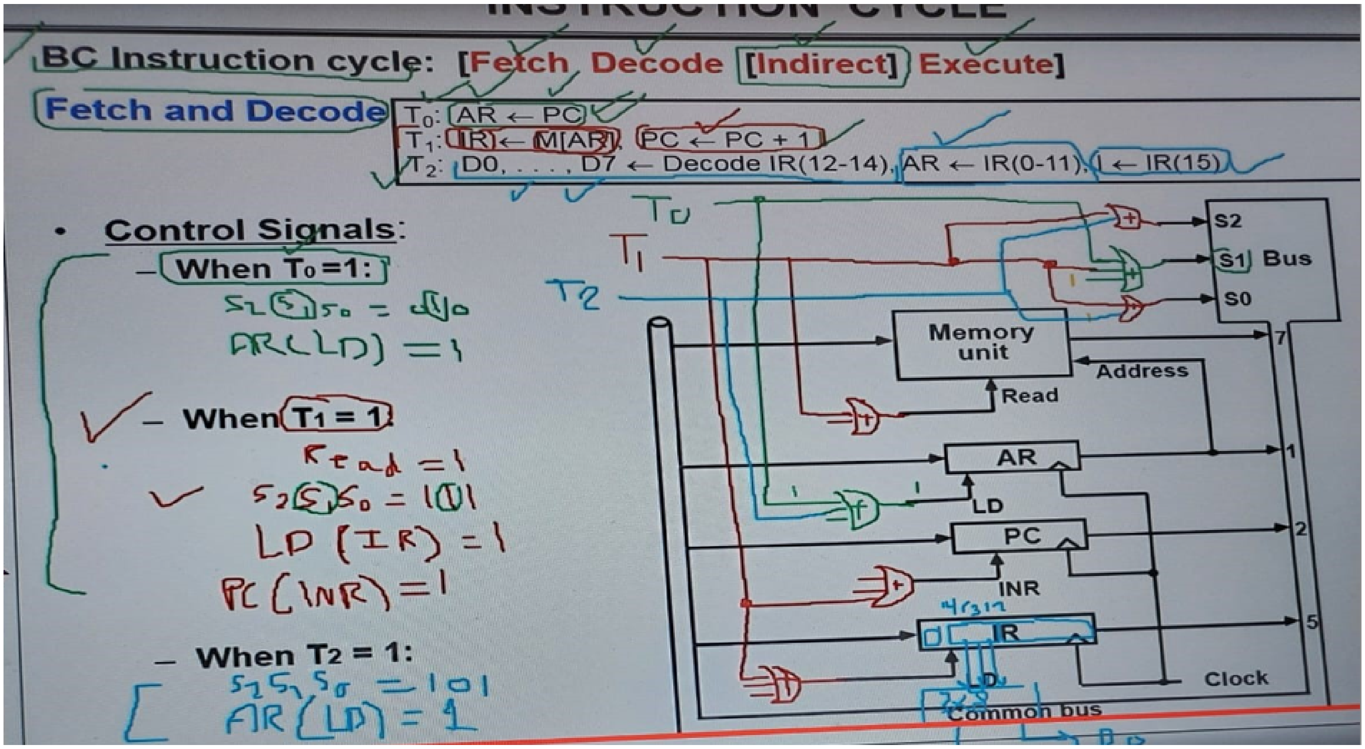
Phases of Instruction cycles:

1. Fetch Instruction from memory
   - (IR <- M[PC].
   - PC<-PC+1)
2. Decode the instruction
   - Read effective address from memory if the instruction has indirect address
3. Execute the instruction

- T_0 : AR <- PC,
- IR <- M[AR]

we can do 2 microoperations in the same time, when 1 for example sends data to the bus, and the other doesn't + there is nothing in common in between them

in the above example we can't do both in one T, as they share an AR in common, so the 2nd instruction needs to go to T_1



BC Instruction cycle: [Fetch, Decode [Indirect] Execute]

Fetch and Decode
T_0: AR ← PC
T_1: IR ← M[AR], PC ← PC + 1
T_2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)

- Control Signals:
  - When T_0=1:
    $S_2 S_1 S_0$ = 010
    AR(LD) = 1
  - When T_1 = 1:
    Read = 1
    $S_2 S_1 S_0$ = 101
    LD (IR) = 1
    PC (INR) = 1
  - When T_2 = 1:
    $S_2 S_1 S_0$ = 101
    AR (LD) = 1

When T_0 is 1

The address in the PC's will be transferred to the AR (how)
the PC has a connection to the bus, so does the AR, the AR is also connected to the memory, and The PC is connected to the address bus, but are not connected to each other, to send data from the PC to the AR, The pc first sends data to the bus line bit 2, The s from 0 to 2 on top's job is turn the specific channel on (by having the value 010), the address instruction will then move until it reaches the AR, for the AR to accept the address of the pc, we need to make the value of LD to equal 1

T_0 gets linked to S_1 using an OR gate, that has many inputs (not if we we're made sure that there was no other instruction, we can link without an OR gate)

T_0 gets linked to an OR gate that's linked to the LD

When T_1 is 1

The memory of the address register will be transferred to the Instruction Register

the instruction is in the memory unit and we know it's address

the program counter will be incremented by 1 to go to the next instruction's register

we start by making the read has the value of 1 (turned on) to read the AR's Address's content, which is then sent to the bus on bit number 7

the value of s 0-1-2 will be 1-1-1

To allow the function to process, the value on LD of the IR has to be set to one (LD[IR] = 1)

When T_2 is on

D_{0-7} are coming from bits 12-14

IR(0-11) sends to the AR

and The I gets it's bit from IR(15)

when the IR get's an instruction, bits (0-11) go to the AR, using the bus

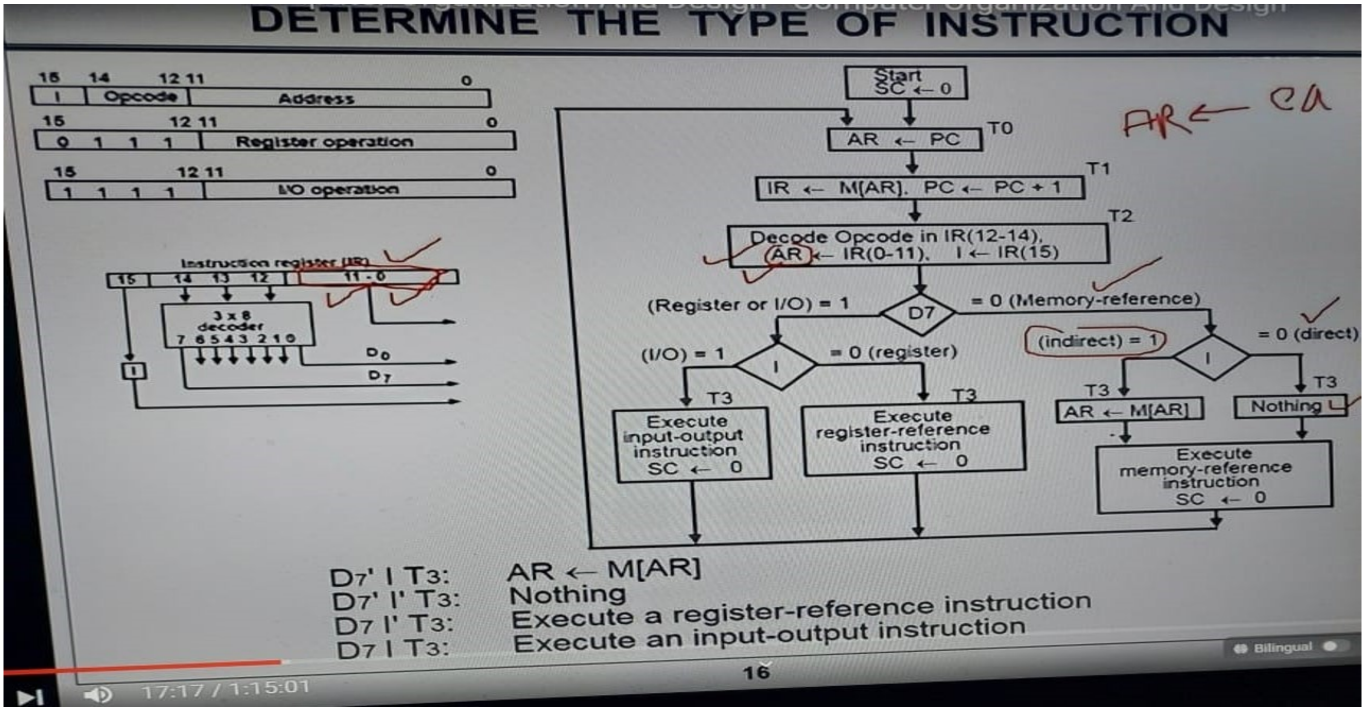
we need to open the bus bit number 5

we also need to make the LD bit equal 1, to allow the content reach the AR

the value of S 0-1-2 = 101

T_2 will be linked to S_0 and S_2 using an OR gate

Determine The type of Instruction



DETERMINE THE TYPE OF INSTRUCTION

D7' I T3: AR ← M[AR]
D7' I' T3: Nothing
D7 I' T3: Execute a register-reference instruction
D7 I T3: Execute an input-output instruction

we have a flow chart that explains how to determine the type of instruction

Start SC <-0, when equals 0 we move the content of the PC to the AR (t_0)

T_1 will then start, with The value of the PC getting incremented by 1 and storing it in the pc (telling the pc to go to the next instruction), then we send the Instruction from the memory and send it to the IR

T_2, we start by looking at D7 (the bits from 0-1 + D7)

if D7's value is 0 then that means the 12 bits are memory reference , we then check for bit 15's value, where if it is 0 then it is an effective address (the address of the operand) , and if it's 1 it's Address of Address (Direct or Indirect)

if D7 = 0, then we proceed to the operation Instruction after which we execute the Memory-Reference Instruction

D_7^ I^ T_3 = Nothing

if D7 = 1, then we go to an address that when we go to gives us the Instruction after which we execute the Memory-Reference Instruction

D_7^ I T_3 = AR <- M[AR]

if D7's Value is 1 then that means the 12 bits are Register or I/O, we then check for bit 15's value if I's value is 0 then it is a Register reference instruction

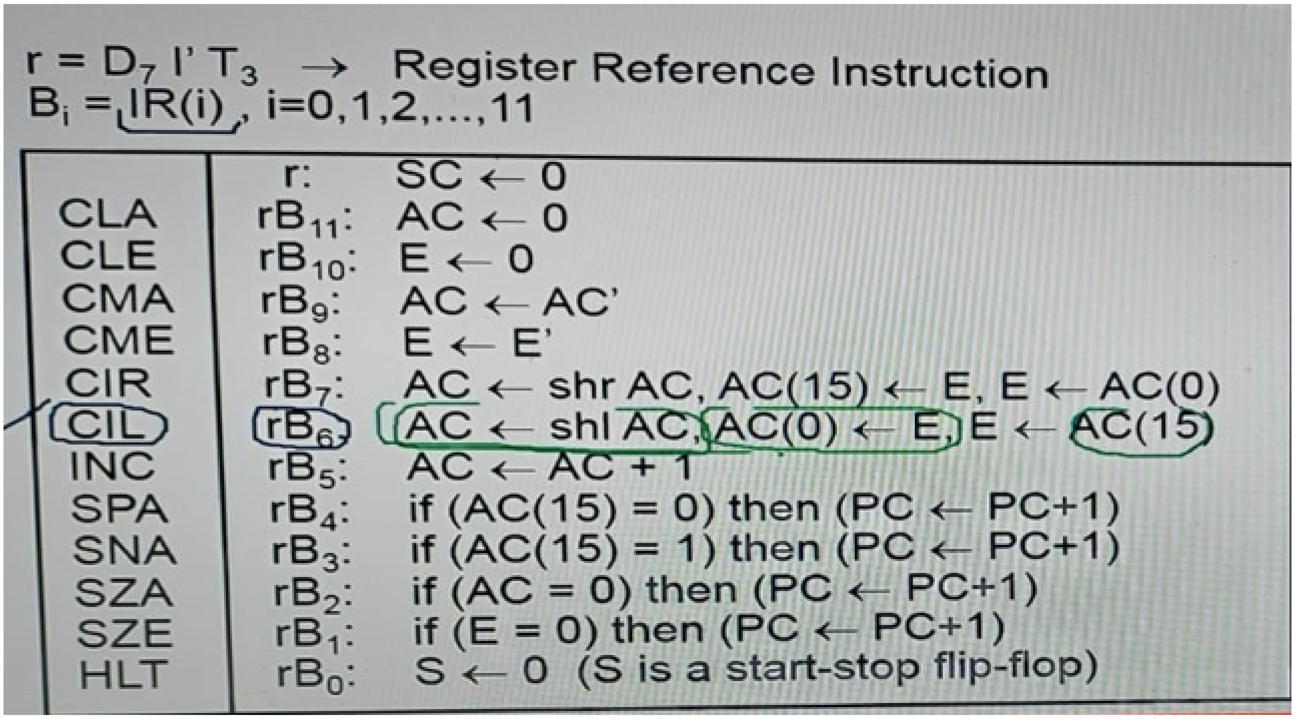
D_7 I^ T_3 = Execute Register reference instruction

and if it's 1 then it is Input/Output Instruction

D_7 I T_3 = Execute Input/Output Instruction

Register Reference Instruction

- Register Reference Instructions are identified when D7 = 1, I = 0
- Register operation is specified by bits B_0-B_{11} of IR
- Execution starts with timing signal T3



$r = D_7 I' T_3$ → Register Reference Instruction

$B_i = \underbrace{IR(i)}$, i=0,1,2,...,11

	r:	SC ← 0
CLA	rB_{11}:	AC ← 0
CLE	rB_{10}:	E ← 0
CMA	rB_9:	AC ← AC'
CME	rB_8:	E ← E'
CIR	rB_7:	AC ← shr AC, AC(15) ← E, E ← AC(0)
CIL	rB_6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB_5:	AC ← AC + 1
SPA	rB_4:	if (AC(15) = 0) then (PC ← PC+1)
SNA	rB_3:	if (AC(15) = 1) then (PC ← PC+1)
SZA	rB_2:	if (AC = 0) then (PC ← PC+1)
SZE	rB_1:	if (E = 0) then (PC ← PC+1)
HLT	rB_0:	S ← 0 (S is a start-stop flip-flop)

the instruction is done when both r and the appropriate B is 1

(skipping through CLA to INC as they are self explanatory)

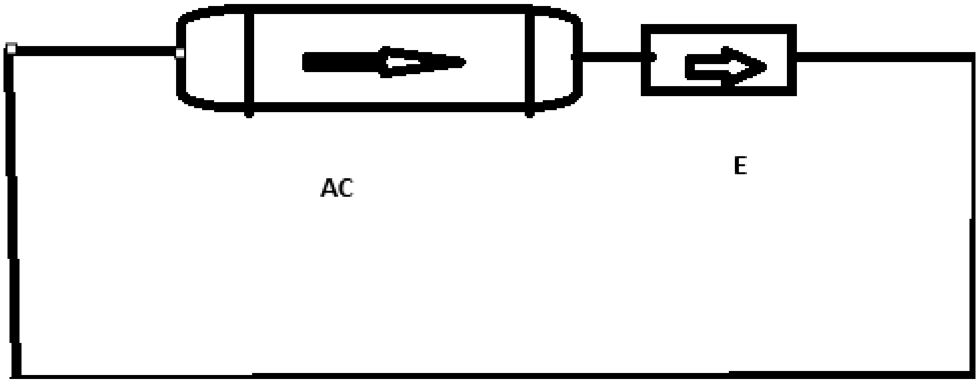
SPA: skip when AC is positive; if the AC bit num15 is positive (0), increment the PC (like skipping the next instruction)

question: explain how can the instruction SPA be executed, and what's the affect of this instruction
Ans: Imagine we have 3 instructions, 100 101 102, the current instruction being executed at location 100, now the PC in this case will be pointing to instruction at location 101, when the

SPA is performed (assuming conditions have been fulfilled), the PC will increment by 1 and point to 102, when it's finished being performed, the instruction at the pc (102) will then go to the IR, the affect is that the pc skipped 101

question: explain how can the instruction CIR be executed.
The bit to the right of the AC will be shifted and Placed in the AC, and the extended bit will then go to bit num 15, and bit num 0 will go to the extended bit

(note you also need to draw how the end product will look like, example)



SNA: when AC = 0, the PC will increment by 1 (as in the whole AC)

SZE: when the extended bit = 0, the PC will increment by 1

HLT: stops the whole program

(our professor stopped to here)