ملاحظة: المكثف بهدف لتلخيص اهم النقاط لتصميم و تنظيم الحاسوب, بس ما بظمنلك تجيب العلامة الكاملة, رح تستفيد من هاظ المكثف لو درست المادة قبل تدرس هان, و انا مش مسؤول عن اي حد ما جاب العلامة الي بدو اياها

Chapters 1+2

Mostly revision, to summarize

Computer architecture: the computer's user-facing structure and behavior, detailing functional modules like instruction sets, information formats, and addressing modes.

Computer organization: how hardware components are interconnected and how they function together.

Computer design involves the development of hardware for the computer system following the formulation of specifications. It encompasses the implementation of the system's components by the designer.

Design Levels

- Architecture (highest)
- Logic
- Electronic (lowest)

Truth Tables

AND

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

NOT

A	Out
0	1
1	0

NAND

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

NOR

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

BUF (Buffer)

A	Out
0	0
1	1

XOR

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

A	B	Out
0	0	1
0	1	0
1	0	0

A	B	Out
1	1	1

SOP VS POS

SOP (Sum of Product) (Minterms) example =A’B’C+AB’C’+AB’C+ABC’+ABC

POS (Product of Sum) (Maxterms) example = (A+B+C)(A+B’+C)(A+B’+C’)

Boolean algebra rules

- X + 0 = X
- X · 0 = 0
- X + 1 = 1
- X · 1 = X
- X + X = X
- X · X = X
- X + X' = 1
- X · X' = 0
- X + Y = Y + X
- X · Y = Y · X
- X + (Y + Z) = (X + Y) + Z
- X · (Y + Z) = X · Y + X · Z
- X + Y · Z = (X + Y) · (X + Z)
- (X + Y)' = X' · Y'
- (X · Y)' = X' + Y'
- (X')' = X

Combinational vs Sequential

Combinational Logic Circuits: Output depends only on the input at the time, with no memory.

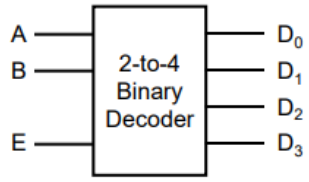
Sequential Logic Circuits: Output relies on both the current input and the circuit's current state, incorporating memory.

tldr : combinational has no memory

Decoder

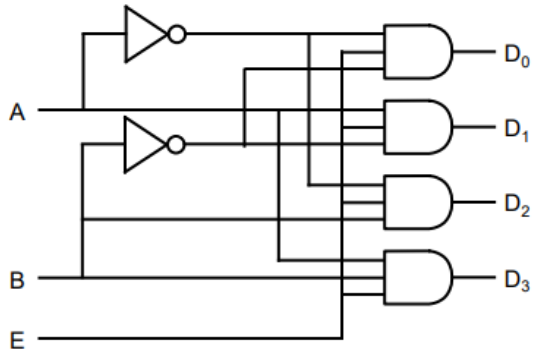
A binary decoder with n inputs and m=2^n outputs generates all minterms, where only one output is active at a time when enabled, with the option to use an inverter for active low enable or NAND gates for active low output.

example



Decoder Truth Table

Inputs			Outputs			
E	B	A	D_3	D_2	D_1	D_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

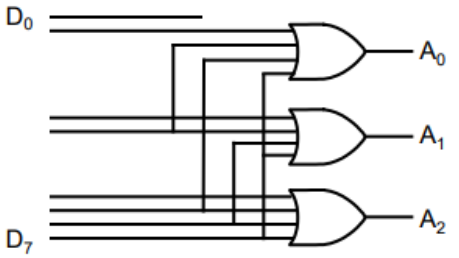


Encoder

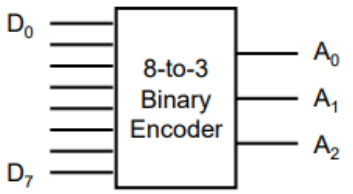
The encoder function takes a single active input number and generates its binary value output, such that if the input number 3 is active, the output is 011.

example

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1



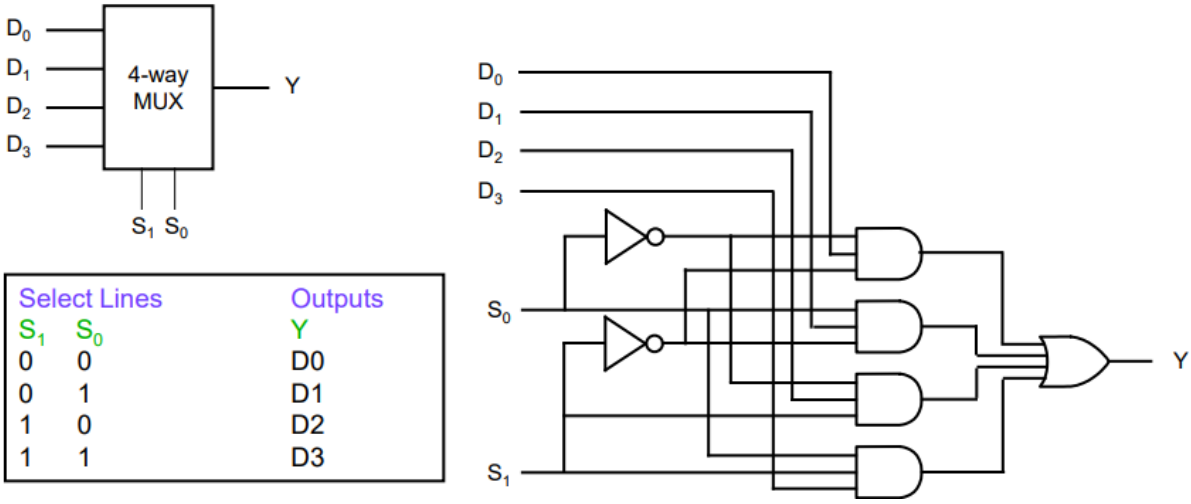
$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$



- Binary encoder requires detection of no active input and resolution of conflicts in case of multiple active inputs.
- Priority is given to higher-numbered inputs when resolving conflicts.
- Preprocessing ensures that only the highest active input is set while others are reset.
- An "Idle" state is activated when no input is active.
- Specific logic is defined for each output bit based on the active inputs and their complements to handle the "Idle" state.

Multiplexers

A multiplexer (MUX) selects one output from multiple inputs based on select lines, typically built as a many-to-one circuit using decoders and AND gates, with an enable option to deactivate all outputs.



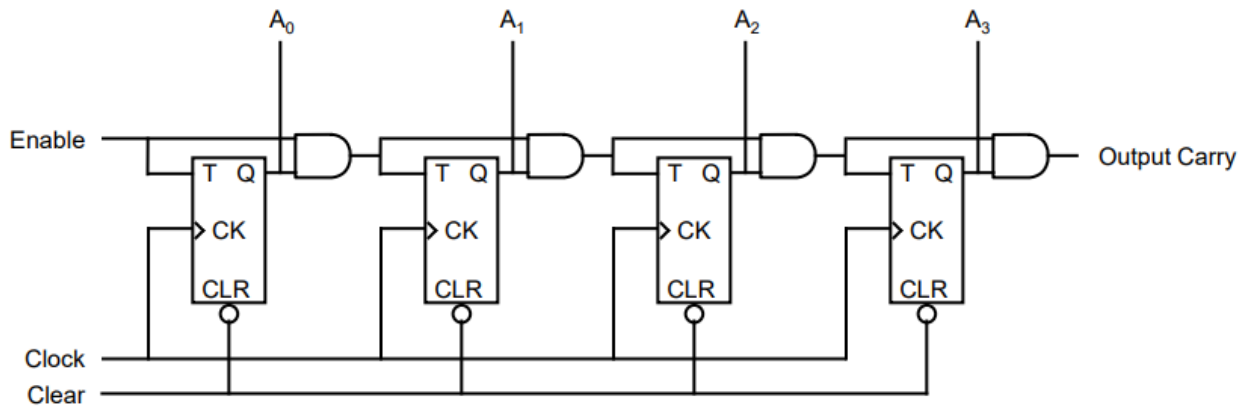
Decoders and multiplexers can be used in implementing because they implement all the minterms.

With suitable MUX size, a function can be implemented directly, while with less complex sizes, one may need logic gate .

Decoders have to be of proper size, and an OR gate is used.

Binary Counter

a circuit where all flip-flops reset together with a master asynchronous clear signal. Changes only happen on the rising edge of the clock signal (0 to 1). Each flip-flop divides the input clock frequency by 2, leading to an overall division of the clock frequency by powers of 2 as it passes through successive flip-flops.



ROM & RAM

Memory consists of cells storing binary bits. Addresses decode rows for read/write operations. Memory sizes are 2^m locations with n bits (e.g., 4, 8, 16). Memories are either read-only (ROM) or read-write (RWM). Silicon memories are randomly accessed (RAM), while magnetic memories are sequential.

Chapters 3

Normalization: writing a number with only 1 non-zero digit before the decimal (scientific notation)

i.e

0.001 turn to $1 * 10^{-3}$

Popular Radices:

1. Binary: Base 2
2. Octal: Base 8
3. Decimal: Base 10
4. Hexadecimal: Base 16

Binary Coded Decimal (BCD):

- Groups of 4 bits independently mapped into decimal digits
- Six invalid combinations exist

ASCII:

- 26 Capital letters (A, B, ..., Z)
- 26 Small letters (a, b, ..., z)
- 10 numbers (0, 1, ..., 9)
- 32 controls (XON, XOFF, DEL, ESCAPE, ...)
- Remaining are special symbols (, ~, `, ^, ...)

Radix Systems:

- Characterized by their base, such as radix 10 (decimal) and radix 2 (binary).
- Each radix system has corresponding complements: 9's and 10's complements for radix 10, 0's and 1's complements for radix 2.
- The size of a number, indicated by the number of digits, holds significance even if the value represented is small.

Fixed-Point Representation:

- **Unsigned Numbers:** Represent only non-negative integers, ranging from 0 to (2^N - 1), where N is the number of bits.
- **Signed Numbers:** Allow representation of both positive and negative integers.
  - *Signed Magnitude:* A sign bit indicates the sign (0 for positive, 1 for negative), and the remaining bits represent the magnitude.
  - *2's Complement Signed Numbers:* Negative numbers are represented in 2's complement form.
- **Overflow and Addition:** Signed numbers never overflow if the numbers to be added have different signs.

Floating-Point Representation:

- The general form is (-1)^S *1.M* 2^E.
  - S: sign bit
  - M: mantissa
  - E: exponent
- The exponent covers both positive and negative ranges.

Single Precision (32-bit) Numbers:

- 1 bit for sign
- 8 bits for biased exponent
- 24 bits for mantissa

Double Precision (64-bit) Numbers:

- Similar to single precision but with increased precision.
- 1 bit for sign
- 11 bits for biased exponent
- 54 bits for mantissa

Advantages of Double Precision over Single Precision:

- Wider range
- Increased accuracy with 54 bits for the mantissa

Given: `0 1001000 | 1100`

Steps:

1. Mantissa Calculation:
    - Binary: `1001000`
    - Decimal: 64+8=72
2. Exponent Calculation:
    - Binary: `1100`
    - Fractional Part: 1/2+1/4=3/4 = .75
3. Combine Mantissa and Exponent:

    =72.75

Result:

- 72.75

Given: `110 | 1`

Steps:

1. Mantissa Calculation:
    - Binary: 110
    - Decimal: 4+2=6
2. Exponent Calculation:
    - Binary: `1000000`
    - Decimal: 1/2
    - Fractional Part: .5
3. Combine Mantissa and Exponent:

    6.5

Result:

- 6.5

Floating-Point Number Conversion Process

1. **Convert Number to Binary**: Convert the given decimal number into its binary representation.
2. **Normalize the Binary Representation**: Identify the leftmost non-zero bit (1 bit) in the binary representation obtained in step 1. Adjust the decimal point to create a normalized binary number in scientific notation.

3. **Calculate the Exponent**: Compute the exponent required to represent the normalized binary number in scientific notation. For IEEE 754 single-precision format, add a bias of 127 (2^8 -1 (to compensate for the s)) to the actual exponent to obtain the biased exponent.
4. **Convert the Exponent to Binary**: Represent the biased exponent obtained in step 3 in binary form. Ensure it is represented using 8 bits.
5. **Extract the Mantissa**: The mantissa represents the fractional part of the normalized binary number obtained in step 2. It consists of all the bits after the binary point.
6. **Compose the Floating-Point Representation**: Assemble the floating-point representation by arranging the bits in the following order:
    - Sign bit (0 for positive numbers, 1 for negative numbers)
    - Exponent bits (obtained in step 4)
    - Mantissa bits (obtained in step 5)

Final representation: "Sign bit, Exponent bits, Mantissa bits".

example= convert $(23.75)_{10}$ to a floating point

23 = 10111

.75 = .11

$(23.75)_{10} = (10111.11)_2 = 1.011111 * 2^4$

E = 127+4= $(131)_{10}$ = 10000011

M = 011111

S = 0

$(23.75)_{10}$ = 0 10000011 011111 (plus enough zeros to make it 23)

example= convert $(12)_{10}$ to a floating point

12 = 1100 = 1,100 2^3

E = 127+3 = 10000010

M = 100

S = 0

$(12)_{10}$ =0 10000010 100 00000000000000000000

example= convert $(-6.5)_{10}$ to a floating point

6 = 110

.5 = 1

110,1 = 1,101 * 2^{2}

E = 127+2 = 129 = 10000001

M = 101

S = 1

(-6.5) = 1 10000001 101 00000000000000000000

Floating-Point Number to decimal Conversion Process

Sign (S):

- Check the first bit of the binary representation:
  - If it is 1, the decimal is negative.
  - If it is 0, the decimal is positive.

Exponent (E):

- Convert the binary representation to decimal format.
- Subtract 127 from the obtained decimal number to determine the exponent.

Mantissa (M):

- No adjustments are required; retain the mantissa as it is.

Normalization Inversion:

- Record the exponent and mantissa obtained.
- Reverse the normalization process by undoing the normalization steps.

Conversion:

- Convert the inverted normalized values back to decimal representation for the final result.

example: convert the following to decimal

1. 1 10000011 1000101

- **Sign (S):** 1 (Negative)

- **Exponent (E):** 10000011 = 131
  131−127=4
- **Significand (M):** 1000101

Calculation: $1,1000101 \times 2^4 = 11000,101$

Result:
=−24.625

2. 1 10000001 101010

- **Sign (S):** 1 (Negative)
- **Exponent (E):** 10000001 =129
  129−127=2
- **Significand (M):** 101010

Calculation: $1,101010 \times 2^2 = 110,1010$

Result:
=−6,625

Other Binary Codes

Gray Code

- Each code differs by only one bit from its adjacent code.
- Physical adjacency implies logical adjacency.

Other Numeric Codes

- **BCD 8421**: Binary-Coded Decimal with weights 8-4-2-1.
- **2421 Weight**: Similar to BCD, but with different weight assignments.
- **Excess-3**: A binary-coded decimal system where each digit is represented by adding 3 to the corresponding binary value.
- **2-out-of-5 Codes**: A numeric encoding scheme using combinations of two out of five possible elements.

Alphanumeric Codes

Other Alphanumeric Codes

- **EBCDIC**: Extended Binary Coded Decimal Interchange Code, historically used by early IBM machines.

- **Unicode**: A character encoding standard capable of representing symbols from all languages, available in 16-bit and 32-bit formats.

Error Detection Codes

- **Parity Bit**: Adding a parity bit to each string doubles the space required.
- Only half of the space is valid, making it possible to detect an intentional change.

Parity: Generation & Checking

- Data transmitted over a link or stored is prone to errors.
- Extra bits act as guards, providing invalid combinations allowing error detection.
- The simplest error detection method is parity. A single bit (0 or 1) is added to the data to ensure the total number of 1s in the string is odd or even.
- Upon receiving the data or retrieving it from storage, a similar circuit checks whether the guard bit is consistent.

Chapter 4

Main Components of a PC

1. CPU
2. Main memory
3. Input/output devices

Two Types of Logic Circuits in Digital Computers

1. Combinational Logic Circuits
2. Sequential Logic Circuits

Register Transfer Languages

1. Register Transfer
2. Bus Transfer
    - Internal Bus (inside CPU)
    - External Bus (from CPU to other devices)
3. Memory Transfer

Types of Microoperations:

1. Arithmetic microoperations

2. Logic microoperations
3. Shift microoperations

Registers in a PC:
Registers are fundamental components in a PC used across all units. They primarily store data and perform logical and arithmetic operations. Registers can be divided into two types:

1. Memory Register: Stores binary data, akin to memory.
2. Operational Register: Executes logical and arithmetic operations, similar to Arithmetic Logic Units.

Memory Types:

1. Main Memory: Faster and at a higher level than normal registers, used to store programs and data accessible to the processor. Organized with addresses to locate specific data.
2. ROM: Stores instructions.

All memories serve specific purposes and are made up of cells, with each cell containing registers that store a single bit (0 or 1). Memories typically have an n-bit (input) and an m-bit (output), with n addresses, and two main inputs: the read gate (to load data from memory to the outside environment) and the write gate (to store data to memory).

Size of Memory:
The size of memory is determined by the number of registers it contains (r = memory). All memories have an address, and the address line determines where read and write operations occur (k = number of memory lines) (r = 2^k). Memories also have a specific number of inputs and outputs.

Microoperations:

Microoperations: elementary operations performed during one clock pulse on the information stored in one or more registers.

Processor Datapath: the arena where data undergoes processing, it encompasses registers, buses, and the arithmetic and logic unit (ALU).

General: Ri ← F (Rj, Rk) , where F is some function like shift, add, ...

Register Transfer Language (RTL):

RTL : symbolic notation used to describe the microoperation transfers among registers and memory locations

The internal organization of a computer includes registers with specific functions, a set of microoperations determined by its design, and control signals governing their execution.

RTL Elements:

1. Set of registers and their functions
2. Set of microoperations and possible microoperations provided by the organization of the PC
3. Control signals that initiate the sequence of microoperations

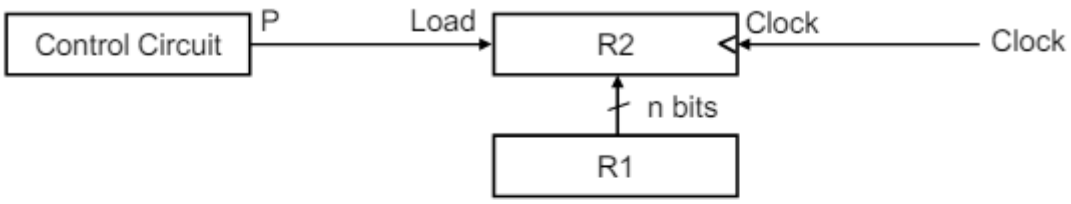
Example: Operation ADD R1, R2

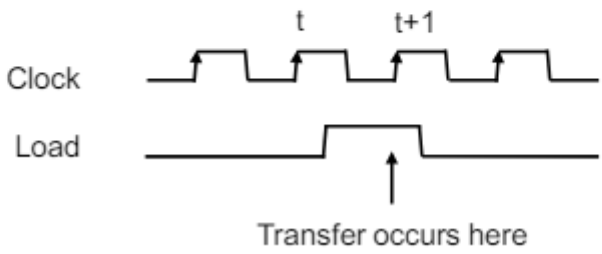
- Registers R1 and R2 act as data registers, holding numbers
- ADD instruction adding two registers
- Control signals:
  - Copies R1 to BUS A and R2 to BUS B of the ALU
  - Commands the ALU to perform addition
  - Moves the output of ALU back to R1

Register Block Diagram

Designation of a Register:

1. A register
2. A portion of a register
3. A bit of a register





Register Transfer

Representation of a transfer
R2 ← R1

A simultaneous copy of all bits of the source to the destination register, during one clock pulse

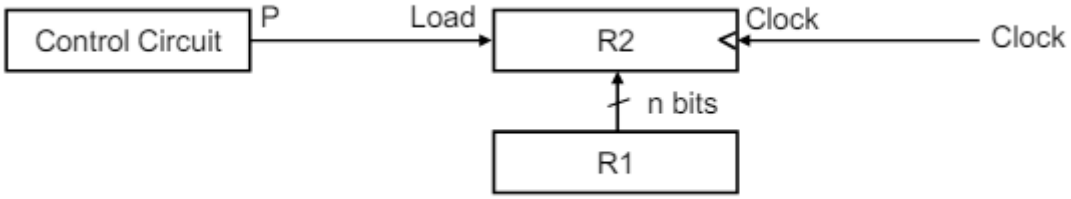
Representation of a controlled (Conditional) transfer

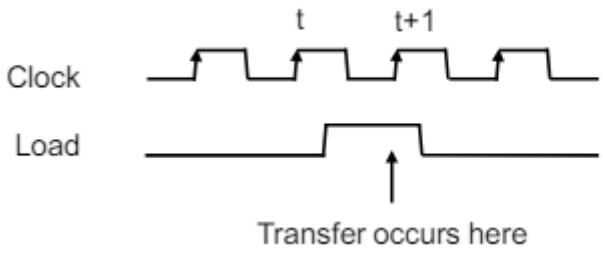
P: R2 ← R1

A binary condition (P=1) which determines when the transfer is to occur If (P=1) then (R2 ← R1)

Implementation of the statement P: R2 ← R1

1. Block Diagram
2. Timing Diagram





Basic Symbols for RTL

Capital Alpha Numerals

- **Description:** Denotes registers like MAR and R2

Parenthesis ( )

- **Description:** Denotes part of registers like R1(3:0)

Arrow ←

- **Description:** Denotes transfer of information

Colon :

- **Description:** Terminates control function like P:

Comma ,

- **Description:** Separates parallel microoperations like A ← B, C ← A

Bus Based Transfer

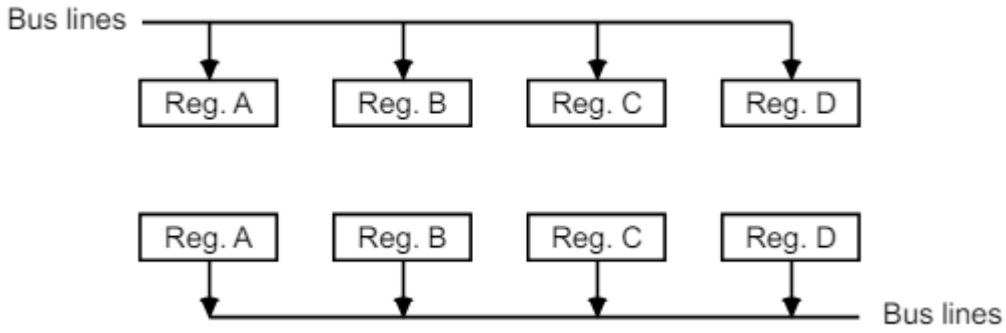
Bus: a shared media path (a group of wires) over which information is transferred, from any of several sources to any of several destinations

Bus is a shared media, used instead of the costly but high performance 1-to-1 connection of registers and memory

This is cheaper and less space consuming but less in performance, as only two parties can be involved at one time

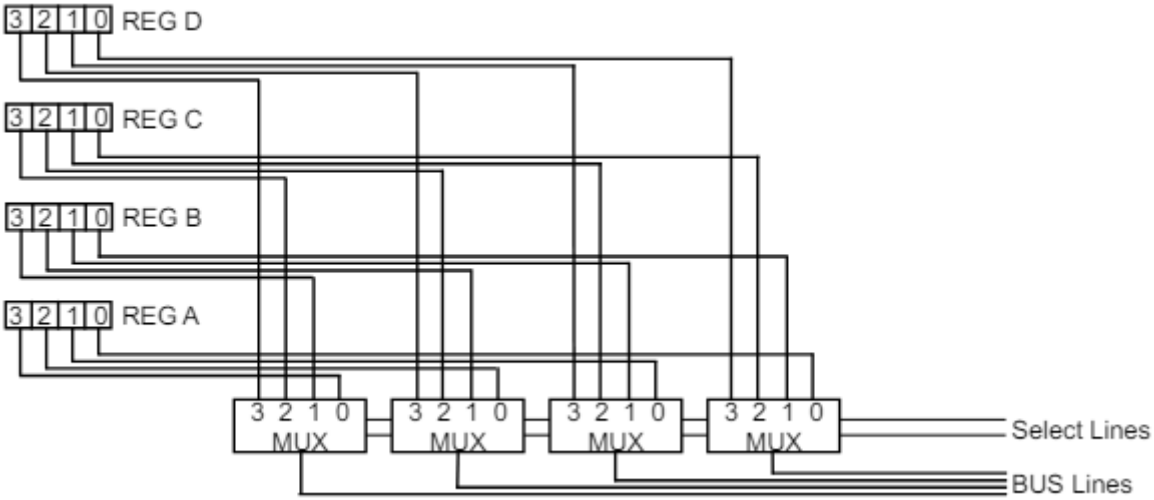
Method	Price	Performance
Bus	Cheaper	Less Performance (two parties can be involved at one time)
Register memory connection	More Expensive	Higher Performance

bus to register and register to bus example:



Bus Register Transfer

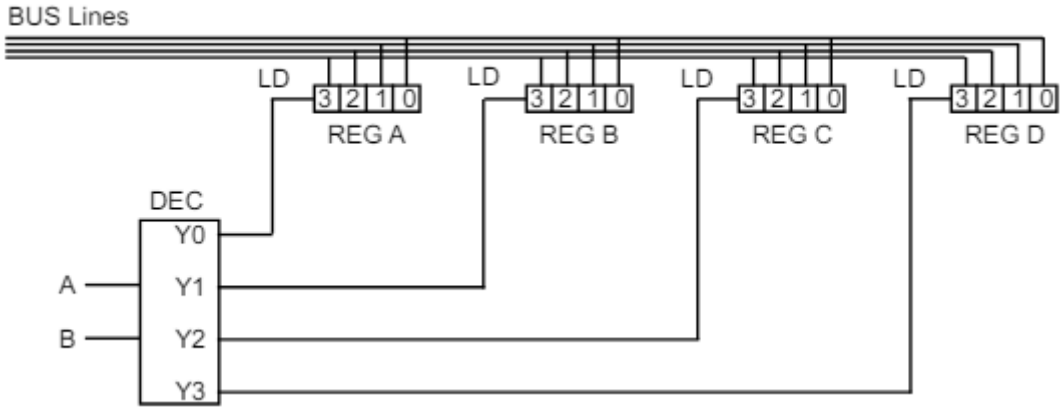
All registers monitor the BUS. Select lines indicate which register can access data. LD (load enabling signal) lets one register retrieve data, clocking in on the rising edge of the clock signal.



Register to Bus Transfer

Mux

- Register to bus has the form BUS
- Ri based on Select Lines
- Select Lines 00 causes all bits of REG A to be switched to BUS ← Using MUX is just one way to do this



Tri State

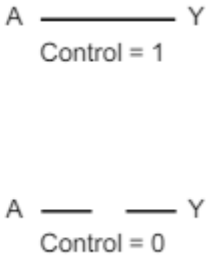
Tri-state buffers can be used in register to bus interface

- If C=1 Then Y is connected to A
- If C=0 Then Y is disconnected from A

Actually connected and disconnected correspond to very low (10’s Ω) and very high impedance (10’s M Ω)

Tri-State Buffer

C	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1



Memory Transfer

Memory operations hinge on the Address Register (AR) for specifying memory read or write locations. Control inputs like Read (RD) and Write (WE) dictate data flow.

The Data Register (DR) stores read or to-be-written memory data.

In systems with multiple memory modules, the Chip Select (CS) control input selects a specific module.

Memory read microoperation
DR ← M[AR]

Memory write microoperation
M[AR] ←DR

Register Transfer Microoperations examples

1. Copy content of reg Rj into reg Ri
   Ri ← Rj
2. Copy the address portion of reg. DR into reg. AR
   AR ← DR
3. Copy a binary constant (C) into reg. Ri
   Ri ← C
4. Copy content of Ri into bus A and, at the same time copy content of bus A into Rj (equivalent to Rj ← Ri)
   ABUS ←Ri, Rj ←ABUS
5. Memory Read
   DR ←M[AR]
6. Memory Write
   M[AR] ←DR

Microoperations types

1. Arithmetic microoperations:
   - Add
   - Subtract

- Increment
- decrement
- Negate (2’s complement)

2. Logic microoperations:
    - AND
    - OR
    - NOT (Complement)
3. Shift microoperations:
    - Shift: Left & Right, Logical and Arithmetic
    - Rotate or Circulate
4. Register transfer microoperations:
    - Pass content through

Arithmetic Microoperations examples

1. Contents of Ri plus Rj copied to Rk
   Rk ←Ri + R
2. Contents of Ri minus Rj copied to Rk
   Rk ← Ri - Rj or Rk ← Ri + Rj’+ 1
3. Complement the contents of Ri
   Ri ← Ri’
4. 2's complement the contents of Ri (negate)
   Ri ←Ri’+ 1
5. Increment the content of Ri by 1
   Ri ←Ri + 1
6. Increment the content of Ri by 1
   Ri ←Ri - 1

4-bit Adder Using set of Full Adders (FA)

also known as ripple carry adder

Time to deliver the sum is proportional to the size

If we invert the B's, yielding the 1's complement of the number, we obtain the difference reduced by 1.

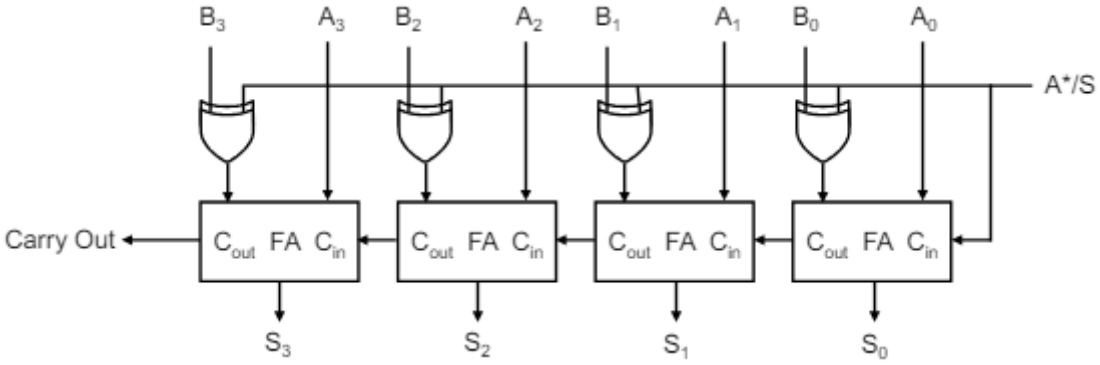
If we subsequently add 1, we achieve the difference itself.

Incrementer Using set of Half Adders (HA)

When adding 1 to a 4-bit number, we only need the first bit's carry. Thus, a Half Adder (HA) is enough. Processors use logic that adds or subtracts 1 to increment or decrement numbers, not carrying through each stage.

4-bit Adder/Subtractor Using set of Full Adders

There are two cases 1. If M=0 then C0=0 and the XOR passes the B's as are, i.e. ADD 2. If M=1 then C0=1 and the XOR complement the B's, i.e. SUBTRACT



Logic Microoperations

Binary operations on strings of bits in registers, Useful for bit manipulations on binary data

1. AND (∧): Masks out specific groups
2. OR (|): Merges binary or character data
3. NOT (¬) Inverts data or mask

Useful for making logical decisions based on the bit value

1. Selective-set (A ∨ B): Sets specific bits to 1 based on conditions.
2. Selective-clear (A ∧ B): Clears specific bits to 0 based on conditions.
3. Selective-complement (A ⊕ B): Toggles specific bits based on conditions.

examples:

Examples: A 1 0 1 1 0 0 1 0
B 0 1 0 1 0 0 1 1
A' 0 1 0 1 1 1 0 1
A ∨ B 1 1 1 1 0 0 1 1
A ∧ B 0 0 0 1 0 0 1 0
A ⊕ B 1 1 1 0 0 0 0 1

X 0 0 1 1 Y 0 1 0 1	Boolean Function	Microoperation	Name
0 0 0 0	F00=0	F ← 0	Clear
0 0 0 1	F01=XY	F ← A∧B	AND
0 0 1 0	F02=XY'	F ← A∧B'	
0 0 1 1	F03=X	F ← A	Transfer A
0 1 0 0	F04=X'Y	F ← A'∧B	
0 1 0 1	F05=Y	F ← B	Transfer B
0 1 1 0	F06=X⊕Y	F ← A⊕B XOR	
0 1 1 1	F07=X+Y	F ← A∨B	OR
1 0 0 0	F08=(X+Y)'	F ← (A∨Y)'	NOR
1 0 0 1	F09=(X⊕Y)'	F ← (A⊕B)'	XNOR
1 0 1 0	F10=Y'	F ← B'	Complement B
1 0 1 1	F11=X+Y'	F ← A∨B'	
1 1 0 0	F12=X'	F ← A'	Complement A
1 1 0 1	F13=X'+Y	F ← A∨B'	
1 1 1 0	F14=(XY)'	F ← (A∧B)'	NAND
1 1 1 1	F15=1	F ← 1	Set to all 1's

Shift Microoperations

Logical Shift:

- Zero is shifted in at one end, and a bit is dropped from the other end.

Arithmetic Shift:

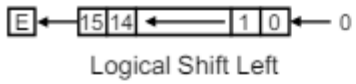
- Similar to logical shift, but the sign bit is copied instead of inserting zero (Right), and changing the sign bit sets the Overflow flag (Left).

Circular Shift: like regular shift, but instead of dropping the bit at one end, it's inserted into the other end.

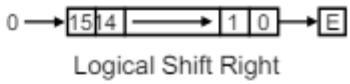
Left Shift is equivalent to multiplying by 2
Right Shift equivalent to dividing by 2

Examples:

Symbol	Description
R ← shl R	Shift-left register R
R ← shr R	Shift-right register R
R ← cil R	Circular shift-left register R
R ← cir R	Circular right-shift register R
R ← ashl R	Arithmetic shift-left register R
R ← ashr R	Arithmetic shift-right register R



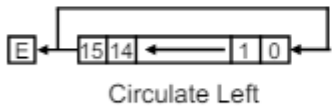
Logical Shift Left



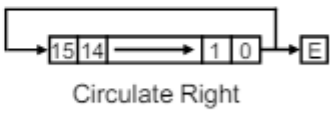
Logical Shift Right

Arithmetic Shift Left
Sign Change Overflow Flag

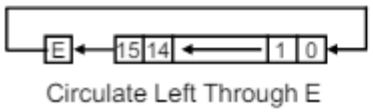
Arithmetic Shift Right
Preserve Sign, BIT 15 fixed



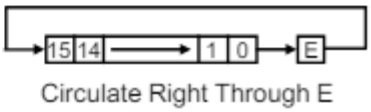
Circulate Left



Circulate Right



Circulate Left Through E



Circulate Right Through E

Shift Right inserts 0 in the leftmost bit through SRin=0 and drops the content of the leftmost bit.

Shift Left inserts 0 in the rightmost bit through SLin=0 and drops the content of the rightmost bit.

Typically, dropped bits get copied to a flag for testing.

Circulate: Instead of inserting at one end and dropping from the other, the end bits are connected to close the circle. Circulation typically copies the dropped bits onto some flag for testing.

Arithmetic Logic Shift Unit (ALSU)

The 1-bit ALSU manages arithmetic, logic, and shift operations on bit I of strings A and B. For shifting the ith bit of string A, adjacent bits are involved.

S1 and S0 determine operations within AU and LU.

S3 and S2 choose the output of AU, LU, or one of the other two inputs that map bits of input string A, shifting left or right.

Chapter 5

Instruction:
an arithmetic or logical operation, consists of operation of code and addressing operant

Basic unit of computing, usually divided into operation code, operand, address, addressing mode, etc.

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

Instruction Code:
Group of bits that tell the computer to perform a specific operation (a sequence of micro-operations)

Addressing Modes (operation field): found in the instruction format, selects the type of operation that must be done on the operands in the register

Basic Addressing Modes:

1. Immediate
2. Register
3. Memory Direct
4. Memory Indirect

data stored in the register or memory is decided based on the addressing mode type

addressing modes sets Regulations on how to use the address field before the fetching the operands process

we have different addressing modes for

1. programming variety for the user (to let him have a chance to use pointers, counter for loop control, and indexing of data, and changing it's place on the memory)
2. reducing the addressing field's number of bits in the instruction (with different modes, gives the programmer (especially the expert in assembly) the flexibility to write a code suitable from the number of instruction and run time)

we decide the addressing mode based on operation of code

instruction format generally is made from code and data formats, made from 3 fields

1. operation of code (operation field): selects the operation to be done
2. mode field: selects the way we get the required operant or the address
3. address field: selects the address of the memory location or the register

A processor with:

1. Register to be used as default operand, Accumulator
2. Memory to hold code and data

Instruction code with two parts

1. An operation code that tells the processor what to do
2. An address part that tells the processor where the data is located, to operate on it along with the register content

The address has 12 bits, while the opcode has 4

if have 4 bits opcode means we have 16 probability

2^4 operation + 2^{12} word = 4096 word

addressing modes express where the operands is and how to handle that part, and has different types

immediate addressing mode: operand is a part of the instruction (the instruction contains the operand field instead of the address field)

direct addressing mode: used to select the factors address in the memory

indirect addressing mode: the value inside the instruction is an address to the factor's address

Effective Address (EA) is the address used to access:

- An operand for a computation-type instruction, or
- A target address for a branch-type instruction

Example

- Direct: Address 457 points to the operand (126). EA=457 (last digit is 0)
- Indirect: Address 457 points to an address to the operand, it points to 1350, which is the address to the operand (438). EA = M[457] (last digit is 1)

to perform some instructions, we need 2 operands (source and destination)

source: location which the cpu reads the data from
Destination: the place where the cpu stores data

there are many types of operands:

- in the instruction itself
- as a type of inner register
- in the memory or an I/O Gate

the main job of a processor is to perform instructions from the main memory

the instruction type must be capable of performing 4 types of operations

1. data transfer (movement between memory and data processor (read and write))
2. arithmetic and logic operations (addition and subtraction compare etc.)
3. program sequencing and flow control (branch instructions)
4. input output transfer (to transfer data to and from the real world)

in the immediate addressing mode how does the computer differentiate between the different modes:
the last part of the operation

BC Components

1. Memory
2. Input/Output
3. Processor
    1. Datapath (include)
        1. ALU
        2. Registers (types)
            1. General purpose like AC and TR
            2. Special purpose like the others
        3. BUS
4. Control (includes)
    1. Clock signal to synchronize the data transfers of all the registers
    2. Bus control signals S2 S1 S0
    3. INC, CLR and LD of all registers
    4. Memory RD and WE signals

BC Instruction Formats

1. Non-memory-Reference Instructions; Opcode = 111 - I = 0 means Register-Reference Instruction - I = 1 means Input-Output Instruction

2. Memory-Reference Instructions: Opcode != 111
    - I = 0 means direct addressing
    - I = 1 means indirect addressing

Opcodes 000 ~ 110 stand for And, Add, Load etc

Instruction Set Completeness

Instruction set must provide for constructing programs to evaluate any computable function

Instruction Types:

1. Functional Instructions; Arithmetic, Logic, and Shift instructions (like)
    - ADD, CMA, INC, CIR, CIL, AND & CLA
2. Transfer Instructions; Register-memory data transfers (like)
    - LDA & STA
3. Control Instructions; Program sequencing and control (like)
    - BUN, BSA & ISZ
4. Input/Output Instructions; Input and output (like)
    - INP and OUT

Missing Instructions

CMA is register-ref

AND is memory-ref (because it requires a second operand, which is always in memory)

Missing instructions:

1. OR: DeMorgan's NOT & AND
2. SUB: CMP & ADD with E=1
3. MUL: ADD repeatedly
4. DIV: SUB repeatedly
5. DEC: CMA, INC & CMA
6. SHR: CIR with E=0
7. SHL: CIL with E=0
8. STE: CLE & CME

Example: DEC operation on a memory location content, x = (105)10

LDA: AC = $(01101001)_2$
CMA: AC = $(10010110)_2$
INC: AC = $(10010111)_2$
CMA: AC = $(01101000)_2$, AC = $(104)_{10}$

Instruction Cycle

Each instruction in the basic computer has 3 or 4 phases:

1. Instruction Fetch Phase; consumes 2 clock cycles (T_0 and T_1)

2. Instruction Decode Phase; consumes 1 clock cycle (T_2)
3. Data Fetch Phase; consumes either 0 or 1 cycle
    - REG & I/O Instructions has no Data Fetch, so 0 cycles
    - MEM Instructions has Data Fetch, and consumes 1 Cycle (T3)
4. Instruction Execute Phase; consumes 1, 2 or 3 cycles
    - REG & I/O start execution in T3 and completes at the end of T3
    - MEM instructions fetches data in T3 , and starts execution in T4 ; then
        - Some finish in T4 , like STA
        - Some finish in T5 , like AND
        - Some finish in T6 , like ISZ

Instruction Fetch & Decode

1. Instruction Fetch Phase T0 : AR <-- PC (S2S1S0=010, AR_LD=1) T1 : IR <-- M [AR], PC <-- PC + 1 (S2S1S0=111, IR_LD=1, PC_INC=1)

2. Instruction Decode Phase
    1. T2 : D0 , . . . , D7 <-- Decode IR(14-12), AR <-- IR(11-0), I <-- IR(15)
3. [Operand Fetch &] Execute Phase
    - T_3 , T_4 , T_5 and T_6
4. Two types of instructions
    1. REG & I/O: No data fetch, execute in T3 and finish
    2. MEM execute: Data fetch in T_3 , execute in T4 , T_4 & T_5 , or T_4 , T_5 and T_6

Simply
- T_0 causes MUX S's to select PC and AR to LD
- T_1 causes MUX S's to select MEM, IR to LD and PC to INC

So:
S_1 is set if T_0 or T_1
S_0 is set if only T_1

Operand Fetch & Execute Paths

Based on the instruction type, one of four paths will be selected in T_3

1. Register instructions
    - Identified when D_7^I' (' means not)^T_3=1

- Start execution at T_3

2. I/O instructions
   - Identified when D_7^I^T_3=1
   - Start execution at T_3
3. I/O instructions
   - Identified when D'$_7$^I'^T_3=1
   - Do NOTHING in T_3 start execution in T_4
4. Memory indirect
   - Identified when D'$_7$^I^T_3=1
   - AR <-- M[AR] in T3 and start execution in T_4

- register and input-output instructions execute in T_3
- memory instructions fetch data in T_3 and start execution in T_4

Register Reference Instructions

Identified when D7=1 & I=0, operation specified in IR(11-0)
Execution starts with timing signal T3
Let r =D_7^I'^T_3 and Bi = IR(i) , i = 11, 10, 9, …, 0

Memory Reference Instructions

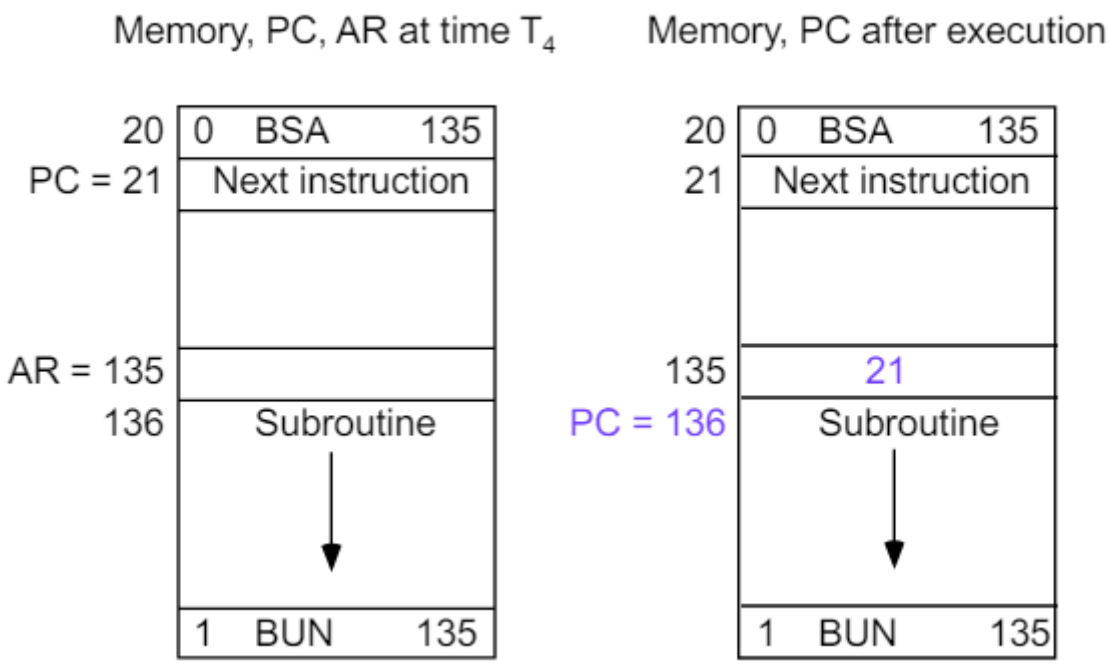
- Effective Address in AR if T_2^I' or T_3^I
- So, by end of T_3 AR points to the operand in memory
- Memory assumed fast enough to complete reading or writing in 1 cycle
- Memory reference instructions start with T_4

Instruction	Opcode Condition	Operation
AND	D_0	AC ← AC ∧ M[AR]
ADD	D_1	AC ← AC + M[AR], E ← C_{out}
LDA	D_2	AC ← M[AR]
STA	D_3	M[AR] ← AC
BUN	D_4	PC ← AR
BSA	D_5	M[AR] ← PC, PC ← AR + 1
ISZ	D_6	M[AR] ← M[AR] + 1, M[AR]+1=0: PC ← PC+1

AND	$D_0 \wedge T_4$:	DR ← M[AR]
	$D_0 \wedge T_5$:	AC ← AC ∧ DR, SC ← 0
ADD	$D_1 \wedge T_4$:	DR ← M[AR]
	$D_1 \wedge T_5$:	AC ← AC + DR, E ← Cout, SC ← 0
LDA	$D_2 \wedge T_4$:	DR ← M[AR]
	$D_2 \wedge T_5$:	AC ← DR, SC ← 0
STA	$D_3 \wedge T_4$:	M[AR] ← AC, SC ← 0
BUN	$D_4 \wedge T_4$:	PC ← AR, SC ← 0
BSA	$D_5 \wedge T_4$:	M[AR] ← PC, AR ← AR + 1
	$D_5 \wedge T_5$:	PC ← AR, SC ← 0
ISZ	$D_6 \wedge T_4$:	DR ← M[AR]
	$D_6 \wedge T_5$:	DR ← DR + 1
	$D_6 \wedge T_6$:	M[AR] ← DR, DR=0: PC ← PC + 1, SC ← 0

Branch After Saving Address (BSA)

- Call subroutine at address 135 while executing at address 20
- Return address, which is 21 is saved to location 135
- Transfer execution to the next location, i.e. 136
- At the end, an indirect unconditional branch is to the beginning, which is 135, i.e. to where it finds the return address, which is 21



Input/Output and Interrupt

Line Styles
1. Thick are Parallel
2. Thin are Serial
3. Dashed are control

Line Color
- Blue from BUS
- Red to ALU

FGI and FGO modified by both the control unit and the external devices

---

Input-Output Configuration

- The keyboard sends and receives serial information
- The serial information from the keyboard is shifted into INPR
- The serial information for the printer is stored in the OUTR
- INPR and OUTR communicate with the computer in parallel and with the outside serially
- The flags are needed to synchronize the timing difference between I/O device and the computer
- INPR Input register - 8 bits
- OUTR Output register - 8 bits
- FGI Input flag - 1 bit
- FGO Output flag - 1 bit
- IEN Interrupt enable - 1 bit

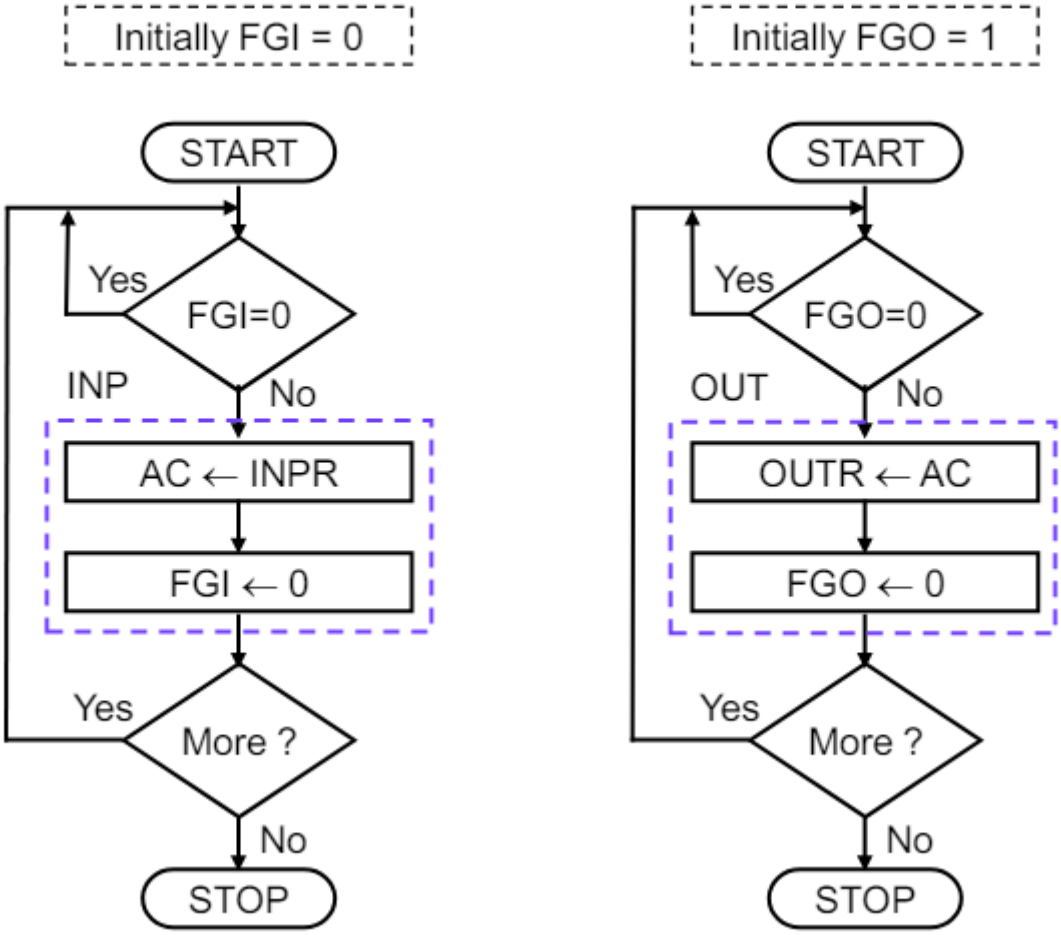
I/O Reference Instructions

- I/O Instructions identified when D_7=1 and I=1
- I/O Instruction is specified in B_{11} ~ B_6 of IR
- Execution starts with timing signal T_3
- I/O instructions include flags control and interrupt control
- p = D_7^I^$_3$ and Bi = IR(i) , i = 11, 10, 9, …, 6

Instruction	Condition	Microoperations
INP	$p \wedge B_{11}$:	AC(0-7) ← INPR, FGI ← 0
OUT	$p \wedge B_{10}$:	OUTR ← AC(0-7), FGO ← 0
SKI	$p \wedge B_9$:	FGI=1: PC ← PC+1
SKO	$p \wedge B_8$:	FGO=1: PC ← PC+1
ION	$p \wedge B_7$:	IEN ← 1
IOF	$p \wedge B_6$:	IEN ← 0

CPU and I/O Device Interactions

- Can get input only if FGI=1 and can send output only if FGO=1
- This explains the initial condition of FGI and FGO



Program-Controlled Input/Output

- CPU continuously involved - CPU slowed down to I/O speed, wasting much of its time - Simple and little hardware

```
Input

WAIT:   SKI
        BUN     WAIT
        INP
        STA     BUF
```

```
Output

        LDA     BUF
WAIT:   SKO
        BUN     WAIT
        OUT
```

Interrupt-Initiated Input/Output

- Open communication only when some data has to be passed, through interrupt.
- The I/O interface, instead of the CPU, monitors the I/O device
- When the interface finds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU
- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns
- IEN is an interrupt-enable flip-flop that can be set and cleared by instructions

---

Interrupt Cycle & Instruction Cycle

The computer has two cycles:

1. Instruction cycle; the normal instruction execution behavior
2. Interrupt cycle, deviation from the normal coarse of action

- The interrupt cycle is a hardware implementation of a branch and save return address operation

If interrupt is enabled (IEN=1) and at least one of the flags is set (FGO=1 or FGI=1), then:
1. Current PC is saved to memory location 0

2. Control is transferred to location 1 (where a branch instruction to the interrupt service routine is stored)
3. At the end of the interrupt service routine, an Indirect unconditional branch to location 0 is executed, to take us back to the takeoff point

- Register Transfer Statements for Interrupt Cycle:
  - IEN ^ (FGI+FGO)(T'_0^T'_1^T'_2): R ^ 1
- Necessary to update R only after decoding, to avoid executing microoperations from both sides (instruction and interrupt cycle)
- The fetch and decode phases of the instruction cycle must be modified
- The T_0 , T_1 and T_2 conditions must be ANDed with R, to become $R'T_0$, $R'T_1$ and $R'T_2$

The interrupt cycle :
- R^T_0: AR<-- 0, TR <-- PC
- R^T_1: M[AR]<-- 0, PC <-- 0
- R^T_2: PC <-- PC+1, IEN<-- 0, R<-- 0,SC<-- 0

- We can only INC, LD or CLR the PC, we CLR then INC to perform PC <-- 1

Design of Basic Computer (BC)

- Hardware Components of BC
  1. A memory unit: 4096 x 16
  2. Registers: AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC
  3. Flip-Flops (Status): I, S, E, R, IEN, FGI, and FGO
  4. Decoders a 3x8 Opcode decoder and a 3x8 timing decoder
  5. Common bus: 16 bits
  6. Adder/Logic Unit
- Control Logic Gates
  1. Registers Controls
  2. Read and Write Controls of Memory
  3. Set, Clear, or Complement Controls of the flip-flops
  4. S2 , S1 , S0 Controls to select a register for the bus
  5. Adder/Logic Unit Controls

Register Control

To find the controls of AR, scan all of the register transfer statements that change the content of AR.

- Below is a list of them
  1. R'^T_0 : AR <-- PC / LD AR
  2. R'^T_2 : AR <-- IR(11-0) / LD AR
  3. D' $_7$^ I ^$_3$: AR <-- M[AR] / LD AR
  4. R^T_0 : AR <-- 0 / CLR AR
  5. D_5^T_4 : AR <-- AR + 1 / INC AR
- In terms of each input, they are:
  1. AR_LD = R'^T_0 + R'^T_2 + D' $_7$^I^T_3
  2. AR_CLR = R^T_0
  3. AR_INC = D_5^ T_4
- Controls of the rest (registers, memory, status and other flip flops) follow the same rule
- To see how easy, the LD of TR is simply the condition R ^ T_0

Flag Control and Flag Control Logic

To find the controls of Interrupt Flip-Flop, IEN, scan all of the register transfer statements that change the content of IEN. In terms of p = D_7^I^T_3 , they are

p^B_7 : IEN <-- 1 (from I/O Instruction)

p^B_6 : IEN <-- 0 (from I/O Instruction)

R^T_2 : IEN <-- 0 (from Interruption)

- J = p^B_7
- K = p^B_6 + R^T_2

Bus Controls

Scan the statements that cause the bus to be loaded by registers or memory

- AR as an example has the following effect on the bus
  - D_4^T_4 : PC <-- AR
  - D_5^T_5 : PC <-- AR

- $x1 = D_4 \wedge T_4 + D_5 \wedge T_5$

x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	none
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	MEM

Condition	Microoperation
$D_0 \wedge T_5$:	AC ← AC ∧ DR
$D_1 \wedge T_5$:	AC ← AC + DR
$D_2 \wedge T_5$:	AC ← DR
$p \wedge B_{11}$:	AC(0-7) ← INPR
$r \wedge B_9$:	AC ← AC'
$r \wedge B_7$:	AC ← shr AC, AC(15) ← E
$r \wedge B_6$:	AC ← shl AC, AC(0) ← E
$r \wedge B_{11}$:	AC ← 0
$r \wedge B_5$:	AC ← AC + 1

شرح حبوش

الدكتور شرح محاضرة كملخص لشباتر 5, برضو حكا ركزو عالشابتر

Types of Languages

1. Machine Language
2. Assembly Language
3. High Level Language
4. Application Language

Addressing Modes

A side of group of instructions, instructions in most CPUs, and decides which instruction type to use and has been used in the building of certain instruction modes, and how to define instructions for the machine language, and these modes focus on the use of programmers who use assembly language.

addressing modes are an aspect of the instruction set architecture in most CPU designs, The various addressing modes that are defined in a given instruction set architecture, define how the machine language instructions in that architecture identifies the operants of each instruction.

direct vs immediate

6 bits to 8 registers -> we use 24 bit registers

addressing mode: is used to find out where are and how to handle the operants

---

addressing mode types

1. immediate

Found in the 2 instruction of the operant

2. Direct

Clarifies the 2nd part of the instruction and determines the operant's memory address (we do the operation on the address and not the data found in the operant)

3. Indirect

if we have indirect addressing, it means that the value found is an address for the operand's address, and so it is the 2nd part of the instruction and determines or contains the address of the operand's address

16/15	**15/11**	**11/0**
0/1	Opcode	Address
0 for direct/ 1 for indirect		

direct addressing mode: goes to the address no problem

Indirect addressing mode: takes the value, and goes to the address, which has an address, which has an operand (like a pointer to the data)

these modes are important in the creation of any delicate machine

from this we have addressing modes that clarifies that the 2nd part from the instruction and how to handle and if the operand is direct or the address of the operand or the address of the cells the operand address requires like indirect

---

Question: the instruction format is made using 16 bits (0-11 for address/ 11-15 for opcode/ 15-16 for Addressing), how can the device figure out the addressing mode.

answer: the 1 bit decides the address

the direct addressing mode gives the operand while the indirect gives the address which is made from 11 bits that contains the program counter (registration address) that contains the instruction address that will be performed and so it contains a 12 bit address, the instructions are contained in the memory

in this case the register size is the same size as the memory (based on the number of busses)

the address register is linked to the memory and it's made from 12 bits and contains the memory address (contains the address that will be sent to the memory and communicates directly with the memory)

Data Register: made from 16 and it's purpose is to contain the operand that comes from the memory and we do operations on.

accumulator: made from 16 bits, and any operation that happens in the alu is stored in the ac

Instruction Register: 16 bits and contains the instructions currently done

Temporary Register: 16 bits, is needed for some operations that need a temp register

I/O Register: made from 8 bits that contain input output bits

Common bus: device linked by a common carrier between devices so Information and data is transferred

---

Computer Instruction Format

1. Memory Reference Instruction
    - instructions that deal with addressing memory directly or indirectly, starts with 0111 (why)
    - we use 7 bits because the 111 is occupied by something else (I/O situations)
2. Register Reference Instruction
    - from 000 to 110
    - is a bunch of instructions to deal with registers and is made from a bunch of formats in the following shape

15-14	14-11	11-0
0	111	Register Operand

3. Input/Output Instructions
    - is occupied by the memory format

how can we Coordinate between instruction formats:

using the designer

Execution Unit: circuit that controls the computer components, has inputs and outputs

- we can do some operations to the control unit to execute using 2 ways
    1. hard wired implemntation
    2. micro programmed operation

Instruction Cycle

Fetching: Getting instructions from memory

Decode: The Process of reading the effective address from the instruction if it has an indirect address

Execute: Implementation of the instruction

The Program

A group of instructions found in the memory

How is a program executed

1. the instruction is transferred to the instruction register
2. the instruction is then analyzed
3. we then go to the processing cycle and move on to the next instruction

we get the instruction from the memory, and the program counter is what gets it using it's address

---

Chapter 6

ملاحظة: بس الي شرحو الدكتور من اخر فيديو

Machine Language and Assembly Language

Program: a set of instructions that the data processing task requires to be done

Hierarchy of Programming Languages

1. Machine Language

binary code, we also have octal and decimal and hexadecimal

2. Assembly Language

considered a middle ground between machine lang and high level languages

we can change high level language into assembly using either a assembler or a compiler

the program wrote in this language needs less space, and less execution time

assembly program are specially used in locations that need a quick response time/ like controls in operating systems

knowledge in assembly language gives you a deeper definition in computer architecture and how a computer works, which can't be given by a high level language

we can use special programs like interrupt from the Input or Output systems on the processor units

so when writing in this language we need to have a good knowledge in instruction sets and rules and hardware that the program will be performed on, otherwise nothing will work

the numbering system used in assembly is a lot

how can we use the binary system with assembly

binary is the system used in computers generally, and the programmer must represent the numbers in 8 bits/16/32 ...etc., and we put the numbers as signed or unsigned

example: al,5

or al = 00000110

or move ax,5

these are generally instructions that is used to move and put values in a specific place, when we say move al (a register from 8 bits) and ax (register from 16 bit)

assembly language syntax (line of code)

all commercial computers in stores, designers occupy the assembly language of their choice and spend it on a catalogue that explain what these instructions mean

we must know that ways of writing are split into lines in assembly language, and are written in 3 columns (fields)

1. label field

May be empty or may specify a symbolic address consists of up to 3 characters

Terminated by a comma

might have a title or a Symbol or 3 chars

2. Instruction Field

Specifies a machine or a pseudo instruction

- May specify one of the following:

1. Memory reference instruction, with two or three symbols separated by spaces (example)
    - ADD OPR (direct address MRI)
    - ADD PTR I (indirect address MRI)
2. Register or Input-Output reference or input-output instruction, having no address part
3. Pseudo instruction with/without an operand (for the assembler), like labels

3. Comment Fields

Optional comment for program readability

Examples

ORG N /Hex number N is memory location for the next instruction or data

END / Denotes the end of symbolic program

DEC N & HEX N / Signed decimal number N or hexadecimal number N to be converted to the binary

---

Assemblers

programs that produce code in one language (object code) from a code written in a another language (source code)

i.e: high-level language programs to machine-level language

Assembly language is a symbolic language with directives

Assemblers: Special case of compilers, where the source code is a program written in assembly language

Assemblers have 2 passes:

1st Pass: generates a table that correlates all user defined (address) symbols with their binary equivalent values

2nd Pass: performs binary translation of instructions and data, using fixed tables and tables built in the 1st pass (Pseudo-Instruction Table, MRI Table, Non-MRI Table and Address Symbol Table

---

Some Symbols in assembly language instructions

registers or general registers are very important in doing mathematical operations, and registers in the cpu for example: display in calculator

we can use a register here for general use like ax to add 3+2
it would like this

mov ax,3

add ax,2

general purpose register

does not have special architecture

we have low (0-7) register and high (8-12) registers and then 14-15-16, we can use any of them for sign

segment register

a bunch of segment register (stack segment, code segment, data segment, extra segment, pointer segment)

index register

flax registers

Programming Arithmetic & Logic Operations

Implementation of Arithmetic and Logic Operations
Hardware: Operation implemented as one machine instruction
Software: Operation implemented as a program using many instructions

Hardware implementation is better (reasons):

1. Runs faster
2. Consume less space in memory

Subroutines

Set of instructions that can be used in a program many times

Need linking; a way to call and return to the calling point

Subroutine is performed many times until we get to something called branching

Data is passed from the calling program to the subroutine and back from the subroutine to the calling program in two ways

1. By value, through registers
2. By reference, through a pointer to memory locations

Character Manipulation

This code reads two characters from the input port, one at a time Also, packs the two characters in the 16-bit accumulator, with the 1^{st} character in the higher order byte, and the 2^{nd} in the lower order byte

---

Interrupt Service

provide efficient way of handling slow devices

Tasks of Interrupt Service Routine

- Save the State of CPU: Contents of processor registers and flags (PC and Flags are must)
- Identify the source of Interrupt: Check which flag is set
- Serve the device whose flag is set: Execute Input or Output Subroutine
- Restore the State of the CPU: Restore contents of processor registers and flags and Turn the interrupt facility on
- Return to the running program: Load PC of the interrupted program

registers that take 16 bits

1. ax
2. bx
3. cx
4. dx
