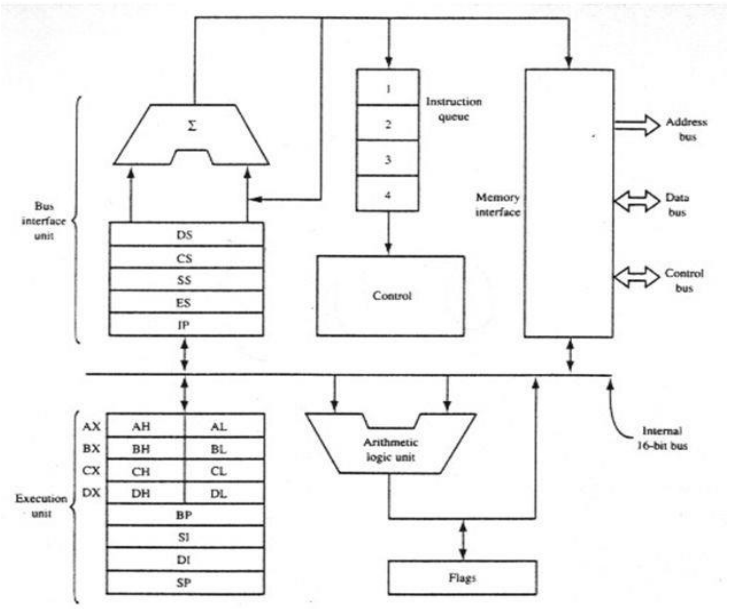
UNIT – III

Introduction to x86 architecture.
Instruction set architecture of a CPU: Registers, instruction execution cycle, RTL Interpretation of instructions, addressing modes, instruction set.
CPU Control unit design: Hardwired and micro-programmed design approaches

X86 Architecture



REGISTERS The processor provides 16 registers for use in general system and application programming. These registers can be grouped as follows:
• **General-purpose data registers**. These eight registers are available for storing operands and pointers.
• **Segment registers**. These registers hold up to six segment selectors.
• **Status and control registers**. These registers report and allow modification of the state of the processor and of the program being executed.

General-Purpose Data Registers
The 32-bit general-purpose data registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:
• Operands for logical and arithmetic operations
• Operands for address calculations
• Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for any other purpose.

Segment Registers
The 6 Segment Registers are:

- Stack Segment (SS). Pointer to the stack.
- Code Segment (CS). Pointer to the code.
- Data Segment (DS). Pointer to the data.

- Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').
- F Segment (FS). Pointer to more extra data ('F' comes after 'E').
- G Segment (GS). Pointer to still more extra data ('G' comes after 'F').

Most applications on most modern operating systems (FreeBSD, Linux or Microsoft Windows) use a memory model that points nearly all segment registers to the same place and uses paging instead, effectively disabling their use. Typically the use of FS or GS is an exception to this rule, instead being used to point at thread-specific data.

x86 Processor Registers and Fetch-Execute Cycle

There are 8 registers that can be specified in assembly-language instructions: eax, ebx, ecx, edx, esi, edi, ebp, and esp. Register esp points to the "top" word currently in use on the stack (which grows down).

Register ebp is typically used as a pointer to a location in the stack frame of the currently executing function.

Register ecx can be used in binary arithmetic operations to hold the second operand.

There are two registers that are used implicitly in x86 programs and cannot be referenced by name in an assembly language program.

These are eip, the "instruction pointer" or "program counter"; and eflags, which contains bits indicating the result of arithmetic and compare instructions.

The basic operation of the processor is to repeatedly fetch and execute instructions.

```
while (running) {
    fetch instruction beginning at address in eip;
    eip <- eip + length of instruction;
    execute fetched instruction;
}
```

Execution continues sequentially unless execution of an instruction causes a jump, which is done by storing the target address in eip (this is how conditional and unconditional jumps, and function call and return are implemented).

Addressing modes

The addressing mode indicates how the operand is presented.

Register Addressing

Operand address R is in the address field.

```
mov ax, bx ; moves contents of register bx into ax
```

Immediate

Aactual value is in the field.

```
mov ax, 1  ; moves value of 1 into register ax
```

Or:

```
mov ax, 010Ch ; moves value of 0x010C into register ax
```

Direct memory addressing

Operand address is in the address field.

```
.data
my_var dw 0abcdh ; my_var = 0xabcd
.code
mov ax, [my_var] ; copy my_var content in ax (ax=0xabcd)
```

Direct offset addressing

Uses arithmetics to modify address.

```
byte_tbl db 12,15,16,22,.....; Table of bytes
mov al,[byte_tbl+2]
mov al,byte_tbl[2] ; same as the former
```

Register Indirect

Field points to a register that contains the operand address.

```
mov ax,[di]
```

The registers used for indirect addressing are BX, BP, SI, DI

Base-index

```
mov ax,[bx + di]
```

For example, if we are talking about an array, BX contains the address of the beginning of the array, and DI contains the index into the array.

Base-index with displacement

```
mov ax,[bx + di + 10]
```

CPU Operation Modes

Real Mode

Real Mode is a holdover from the original Intel 8086. The Intel 8086 accessed memory using 20-bit addresses. But, as the processor itself was 16-bit, Intel invented an addressing scheme that provided a way of mapping a 20-bit addressing space into 16-bit words. Today's x86 processors start in the so-called Real Mode, which is an operating mode that mimics the behavior of the 8086, with some very tiny differences, for backwards compatibility.

Protected Mode

Flat Memory Model

If programming in a modern operating system (such as Linux, Windows), you are basically

programming in flat 32-bit mode. Any register can be used in addressing, and it is generally more efficient to use a full 32-bit register instead of a 16-bit register part. Additionally, segment registers are generally unused in flat mode, and it is generally a bad idea to touch them.

Multi-Segmented Memory Model

Using a 32-bit register to address memory, the program can access (almost) all of the memory in a modern computer. For earlier processors (with only 16-bit registers) the segmented memory model was used. The 'CS', 'DS', and 'ES' registers are used to point to the different chunks of memory. For a small program (small model) the CS=DS=ES. For larger memory models, these 'segments' can point to different locations.

Register Transfer Language And Micro Operations:

Register Transfer language:

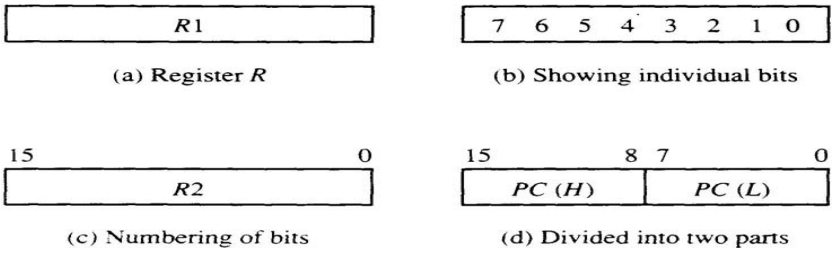
- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic
- The modules are interconnected with common data and control paths to form a digital computer system
- The operations executed on data stored in registers are called microoperations
- A microoperation is an elementary operation performed on the information stored in one or more registers
- Examples are shift, count, clear, and load
- Some of the digital components from before are registers that implement microoperations
- The internal hardware organization of a digital computer is best by specifying
    - The set of registers it contains and their functions
    - The sequence of microoperations performed on the binary information stored
    - The control that initiates the sequence of microoperations

Use symbols, rather than words, to specify the sequence of microoperations
The symbolic notation used is called a register transfer language
A programming language is a procedure for writing symbols to specify a given computational process
Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations

Register Transfer

Designate computer registers by capital letters to denote its function.
The register that holds an address for the memory unit is called MAR.
The program counter register is called PC.
IR is the instruction register and R1 is a processor register
The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1
Refer to Figure 4.1 for the different representations of a register

Figure 4-1 Block diagram of register.



- Designate information transfer from one register to another by R2 □ R1
- This statement implies that the hardware is available
    - The outputs of the source must have a path to the inputs of the destination
    - The destination register has a parallel load capability
- If the transfer is to occur only under a predetermined control condition, designate it by

If (P = 1) then (R2 □ R1) or, P: R2 □ R1,where P is a control function that can be either 0 or 1

- Every statement written in register transfer notation implies the presence of the required hardware construction

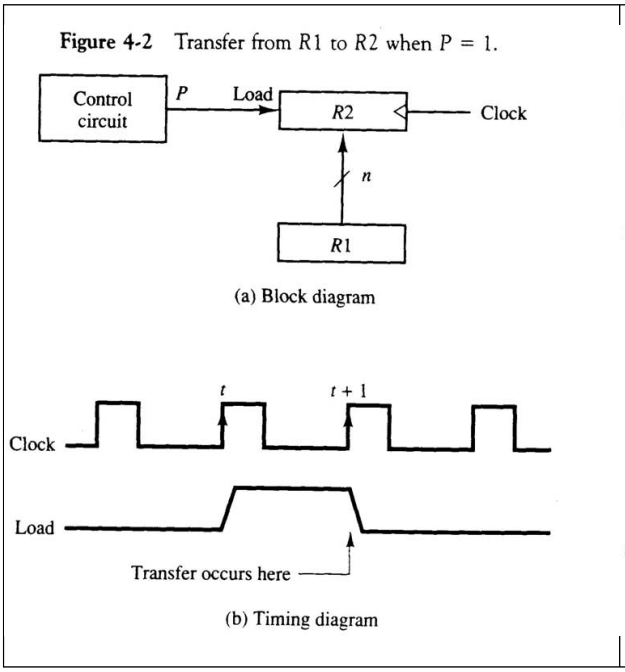


Figure 4-2 Transfer from R1 to R2 when P = 1.

TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	MAR, R2
Parentheses ( )	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Comma ,	Separates two microoperations	R2 ← R1, R1 ← R2

Arithmetic Micro-operations

There are four categories of the most common micro operations:

Register transfer: transfer binary information from one register to another

Arithmetic: perform arithmetic operations on numeric data stored in registers

Logic: perform bit manipulation operations on non-numeric data stored in registers

Shift: perform shift operations on data stored in registers

The basic arithmetic micro operations are addition, subtraction, increment, decrement, and shift

Example of addition: R3 □ R1+R2

Subtraction is most often implemented through complementation and addition

Example of subtraction: R3 □ R1 +~~R2~~ + 1 (strikethrough denotes bar on top – 1"s complement of R2)

Adding 1 to the 1"s complement produces the 2"s complement

Adding the contents of R1 to the 2"s complement of R2 is equivalent to subtracting

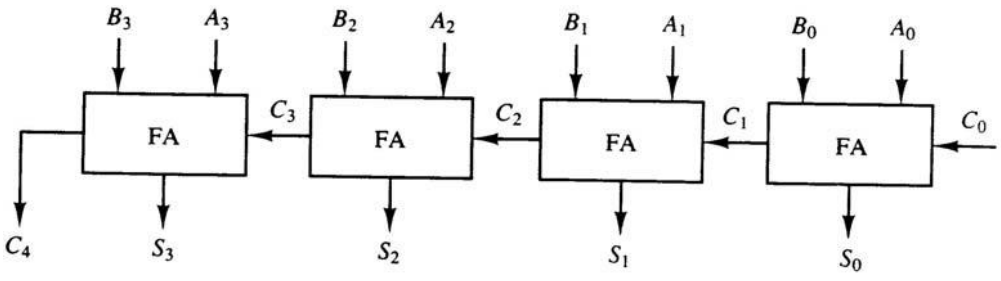


Figure 4-6 4-bit binary adder.

Multiply and divide are not included as micro operations
A micro operation is one that can be executed by one clock pulse
Multiply (divide) is implemented by a sequence of add and shift micro operations (subtract and shift)
To implement the add micro operation with hardware, we need the registers that hold the data and the digital component that performs the addition
A full-adder adds two bits and a previous carry

A binary adder is a digital circuit that generates the arithmetic sum of two binary numbers of any length
A binary added is constructed with full-adder circuits connected in cascade
An n-bit binary adder requires n full-adders

The subtraction A-B can be carried out by the following steps
Take the 1"s complement of B (invert each bit)

Get the 2"s complement by adding 1
Add the result to A
The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder
The increment micro operation adds one to a number in a register
This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one
If the increment is to be performed independent of a particular register, then use half-adders connected in cascade

An n-bit binary incrementer requires n half-adders

Each of the arithmetic micro operations can be implemented in one composite arithmetic circuit
The basic component is the parallel adder
Multiplexers are used to choose between the different operations
The output of the binary adder is calculated from the following sum: $D = A + Y + C_{in}$

Logic Microoperations

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately
- Example: the XOR of R1 and R2 is symbolized by
  P: R1 □ R1⊕ R2
- Example: R1 = 1010 and R2 = 1100

  1010 Content of R1
  1100 Content of R2

  0110 Content of R1 after P = 1

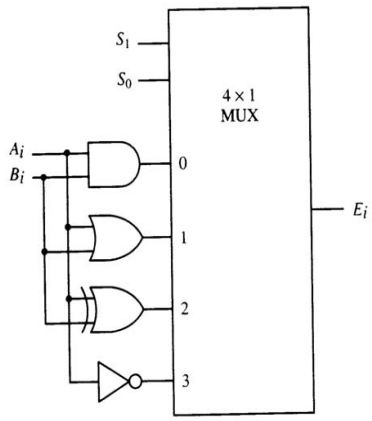
- Symbols used for logical microoperations:
  - OR: □
  - AND: □
  - XOR: ⊕
- The + sign has two different meanings: logical OR and summation
- When + is in a microoperation, then summation

- When + is in a control function, then OR
- Example:

P + Q: R1 □ R2 + R3, R4 □ R5 □ R6

- There are 16 different logic operations that can be performed with two binary variables
- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers
- All 16 microoperations can be derived from using four logic gates

Figure 4-10 One stage of logic circuit.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

- Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register
- The selective-set operation sets to 1 the bits in A where there are corresponding 1‟s in B

1010 A before
1100 B
(logic operand) 1110 A after A □ A □ B

- The selective-complement operation complements bits in A where there are corresponding 1‟s in B

1010 A before
1100 B
(logic operand) 0110 A after
A □ A⊕B

- The selective-clear operation clears to 0 the bits in A only where there are corresponding 1‟s in B

1010 A before
1100 B (logic operand) 0010 A after A □ A □ B

- The mask operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0‟s in B

1010 A before
1100 B
(logic operand) 1000 A
after A □ A □ B

- The insert operation inserts a new value into a group of bits
- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

0110 1010 A before
0000 1111 B (mask)
0000 1010 A after masking

0000 1010 A before

1001 0000 B (insert)
1001 1010 A after insertion

- The clear operation compares the bits in A and B and produces an all 0"s result if the two number are equal

1010 A
1010 B
0000 A □ A⊕B

Shift Microoperations

Shift microoperations are used for serial transfer of data
They are also used in conjunction with arithmetic, logic, and other data- processing operations
There are three types of shifts: logical, circular, and arithmetic
A logical shift is one that transfers 0 through the serial input
The symbols shl and shr are for logical shift-left and shift-right by one position R1 □ shlR
The circular shift (aka rotate) circulates the bits of the register around the two ends without loss of information
The symbols cil and cir are for circular shift left and right

The arithmetic shift shifts a signed binary number to the left or right.
To the left is multiplying by 2, to the right is dividing by 2.
Arithmetic shifts must leave the sign bit unchanged.
A sign reversal occurs if the bit in R_{n-1} changes in value after the shift.
This happens if the multiplication causes an overflow.
An overflow flip-flop V_s can be usedto detect

theoverflow $V_s = R_{n-1} \oplus R_{n-2}$

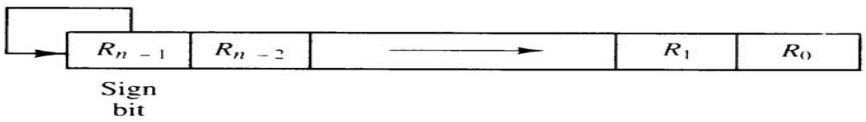


Figure 4-11 Arithmetic shift right.

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration: one to load the value and another to shift
- In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit
- The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register
- This can be constructed with multiplexers

Arithmetic Logic Unit

- The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers
- To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU
- The ALU performs an operation and the result is then transferred to a destination register
- The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period

Micro Programmed Control

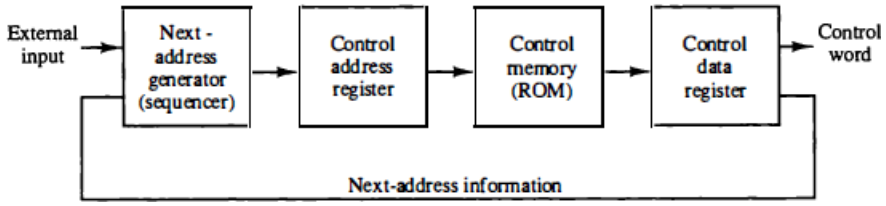
A control unit whose binary control variables are stored in memory is called a microprogrammed control unit. Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes a microprogram .Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk.

Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading.

A memory that is part of a control unit is referred to as a control memory.

Figure 7-1 Microprogrammed control organization.



The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory.

The control data register holds the present microinstruction while the next address is computed and read from memory.

The data register is sometimes called a pipeline register.

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established, there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory. It should be mentioned that most computers based on the reduced instruction set computer (RISC).

Address Sequencing

Microinstructions are stored in control memory in groups, with each group specifying a routine.

The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process.

A mapping procedure is a rule that transforms the instruction code into a control memory address

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return

Conditional Branching

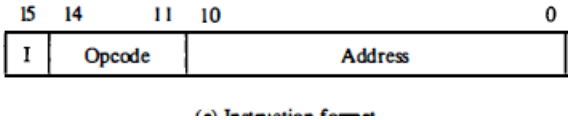
Special Bits : The branch logic provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions

Branch Logic : The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented. This can be implemented with a multiplexer.

Mapping of Instruction: A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction.

Microinstruction Format

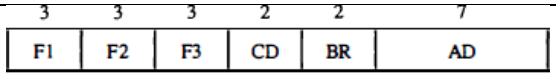
The microinstruction format for the control memory is shown in Fig. The 20 bits of the microinstruction are divided into four functional parts. The three fields F1, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 27 words.



(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from Fl.

$$DR \leftarrow M[AR] \quad \text{with F2 = 100}$$

$$\text{and} \quad PC \leftarrow PC + 1 \quad \text{with F3 = 101}$$