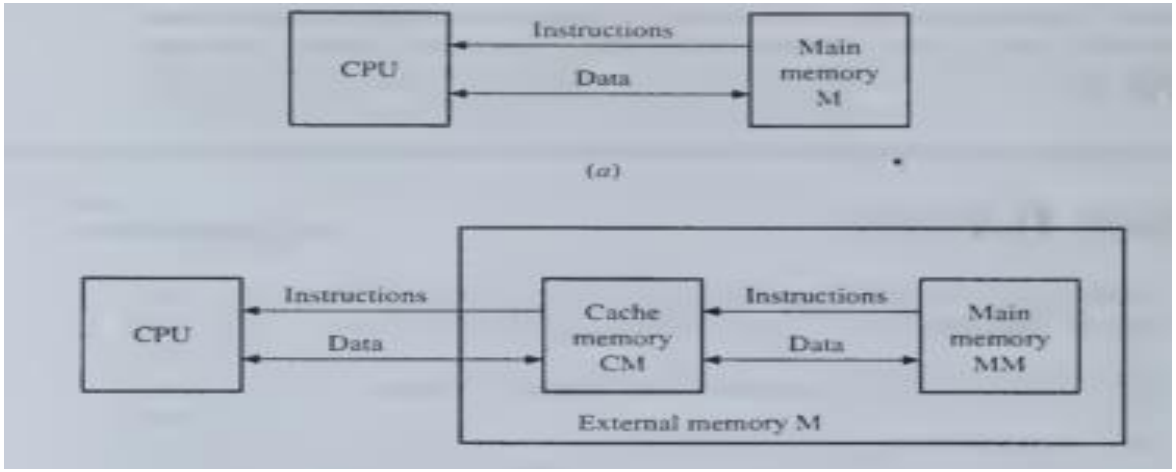
Processor Basics

CPU ORGANIZATION

- Program execution is carried out as follows:
  - The CPU transfers instructions and, when necessary, their input data (operands) from main memory to registers in the CPU.
  - The CPU executes the instructions in their stored sequence except when the execution sequence is explicitly altered by a branch instruction.
  - When necessary, the CPU transfers output data (results) from the CPU registers to main memory.

- The CPU communicates directly with the main memory M, which is typically a high-capacity multichip random-access memory (RAM).
- Many computers have a cache memory CM positioned between the CPU and main memory. The cache CM is smaller and faster than main memory and may reside, wholly or in part, on the same chip as the CPU.



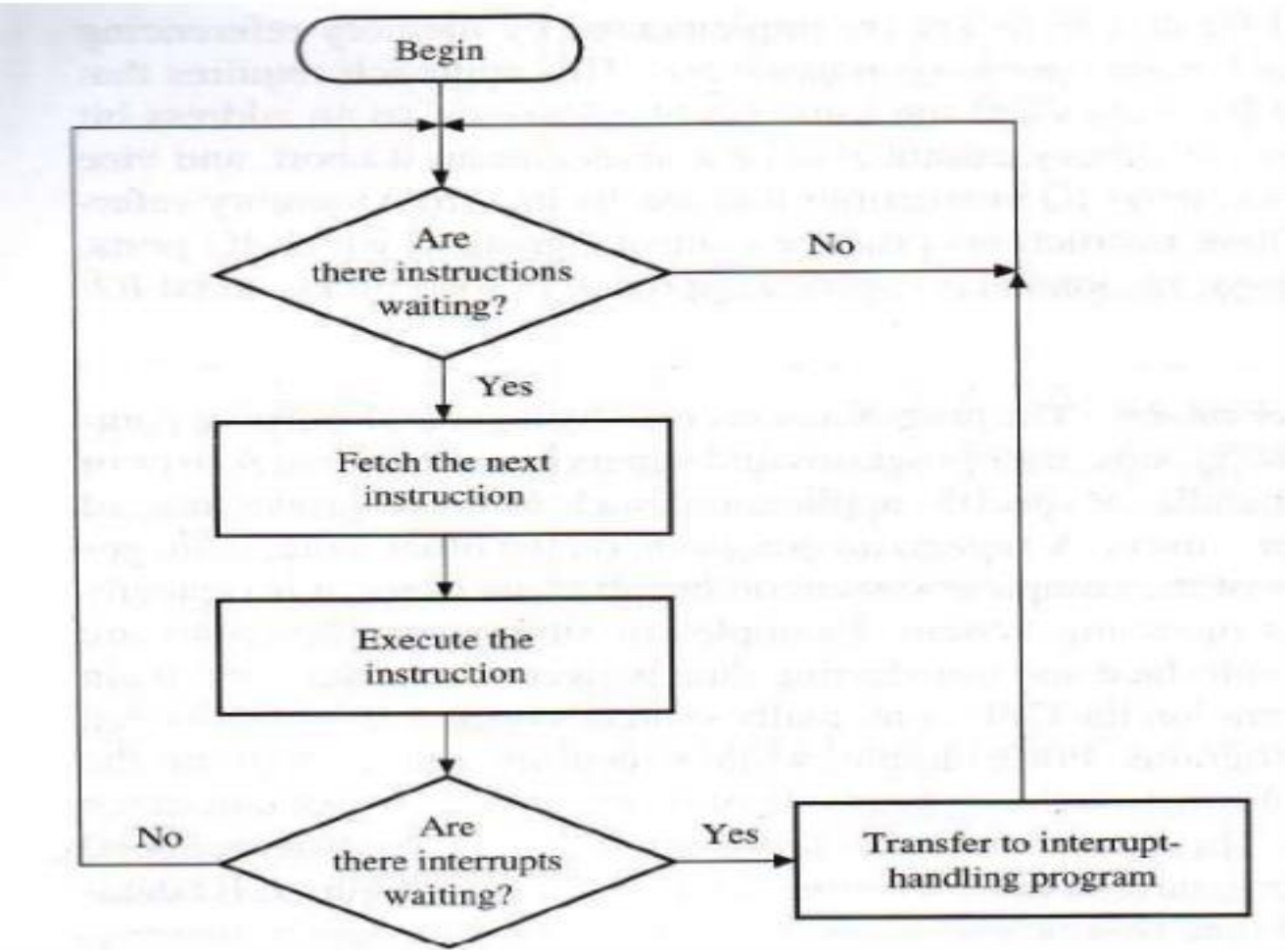
- The CPU communicates with IO devices in the same way as it communicates with external memory.
- The IO devices are associated with addressable registers called IO Ports to which CPU can store a word or load a word

- Two ways to access the IO ports
  - Memory mapped IO
  - IO Mapped IO

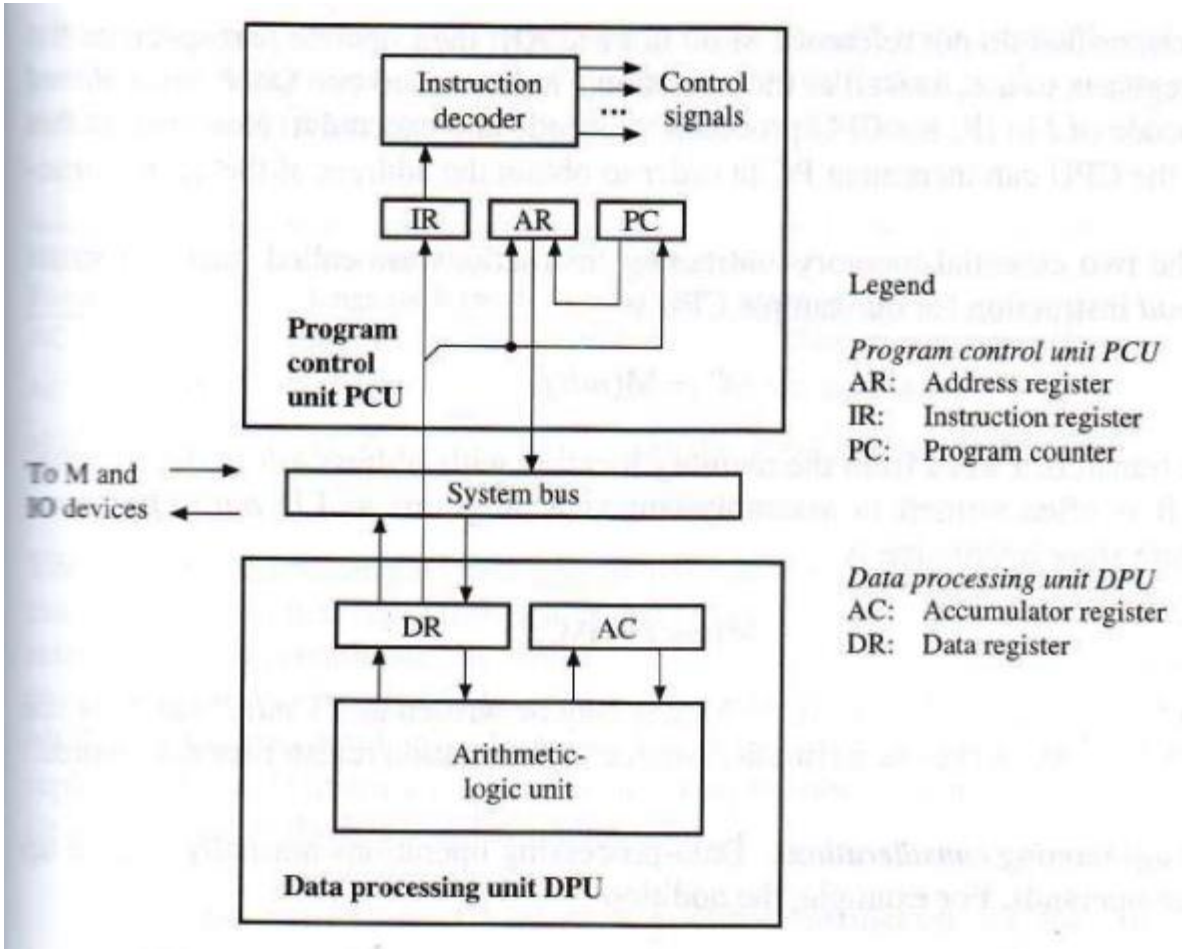
- Programs are classified into two groups
  - User Programs or Application Programs
  - Supervisor Programs (OS) : manages routine aspects of the computer on behalf of user.

A CPU can be designed such that the requests it receives from IO devices is serviced directly. Such a request is known as interrupt.

CPU operation



Accumulator Based CPU



Accumulator Based CPU

- We consider an n-bit ALU and two data registers AC and DR.
- The instructions are of the form X1:=f(X1,X2), where X1 and X2 denotes CPU registers or memory.
- Each instruction I=op.adr

  where op=opcode and adr denotes address of memory.

  IR.AR=M(PC)

  IR:=op, AR:=adr

The instruction Z:=X+Y denotes three distinct operands.

The accumulator based processor is a single address instruction

HDL format	Assembly-language format	Narrative format (comment)
AC := M(*X*);	LD *X*	Load *X* from M into accumulator AC.
DR := AC;	MOV DR, AC	Move contents of AC to DR.
AC := M(*Y*);	LD *Y*	Load *Y* into accumulator AC.
AC := AC + DR;	ADD	Add DR to AC.
M(Z) := AC;	ST Z	Store contents of AC in M.

This program fragment uses only load and store instructions to access memory, a feature called load/store architecture

Instruction set of Accumulator Based CPU

Type	Instruction	HDL format	Assembly-language format	Narrative format (comment)
Data transfer	Load	AC := M(*X*)	LD X	Load *X* from M into AC.
	Store	M(*X*) := AC	ST X	Store contents of AC in M as *X*.
	Move register	DR := AC	MOV DR, AC	Copy contents of AC to DR.
	Move register	AC := DR	MOV AC, DR	Copy contents of DR to AC.
Data processing	Add	AC := AC + DR	ADD	Add DR to AC.
	Subtract	AC := AC – DR	SUB	Subtract DR from AC.
	And	AC := AC ***and*** DR	AND	And bitwise DR to AC.
	Not	AC := ***not*** AC	NOT	Complement contents of AC
Program control	Branch	PC := M(*adr*)	BRA adr	Jump to instruction with address *adr*.
	Branch zero	**if** AC = 0 **then** PC := M(*adr*)	BZ adr	Jump to instruction *adr* if AC = 0.

- A CPU can be designed to implement memory referencing instructions of the form

  AC:=f(AC,M(adr))

  Now Z:=X+Y can be implemented by

```
AC:=M(X)          ;LD X
AC:=AC+M(Y)       ;ADD Y
M(Z):=AC          ;ST Z
```

Data Representation

- Every information can be either
  - Instruction
  - Data

Data can be either

Numerical

Fixed Point

Floating Point

Non Numerical Data

ASCII

EBCDIC

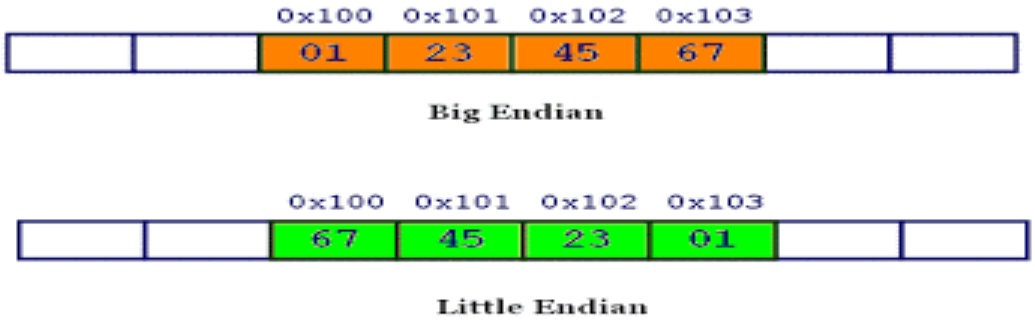
The Numerical data can be represented in the binary or decimal format

The data can be in Bits, Bytes, Word, Double Word.

Suppose that a sequence *w0,w1,…wm* are m-4 byte number words is stored and each word *wi* is represented by $B_{i,3}$, $B_{i,2}$, $B_{i,1,}$ $B_{i,0}$

W0, W1,… Wm= $B_{0,3}$, $B_{0,2}$, $B_{0,1,}$ $B_{0,0}$, $B_{1,3}$, $B_{1,2}$, $B_{1,1,}$ $B_{1,0}$ … $B_{m,3}$, $B_{m,2}$, $B_{m,1,}B_{m,0}$

Memory representation of integer ox01234567 inside Big and little endian machines



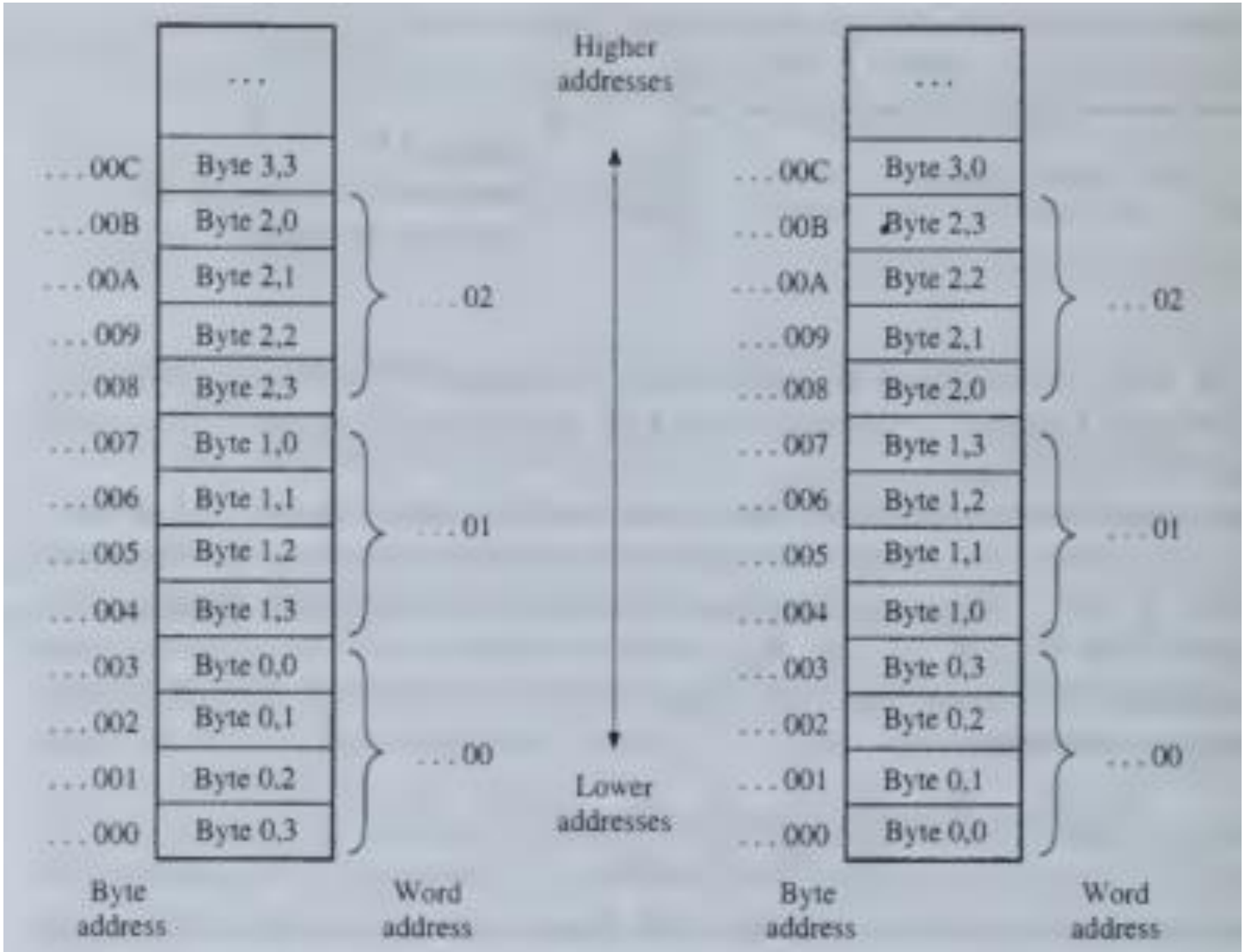


Figure 3.18
Basic byte storage methods: (*a*) big-endian and (*b*) little-endian.

Fixed point numbers

- In selecting a number representation following factors are taken into account:
  - The number type to be represented: Real or integer number
  - The range of values likely to be encountered
  - The precision of the numbers, which indicates the accuracy
  - The cost of the hardware required to store and process the number.

Sign Integer Representation

- For fixed point numbers the representation are

  1.Sign magnitude representation

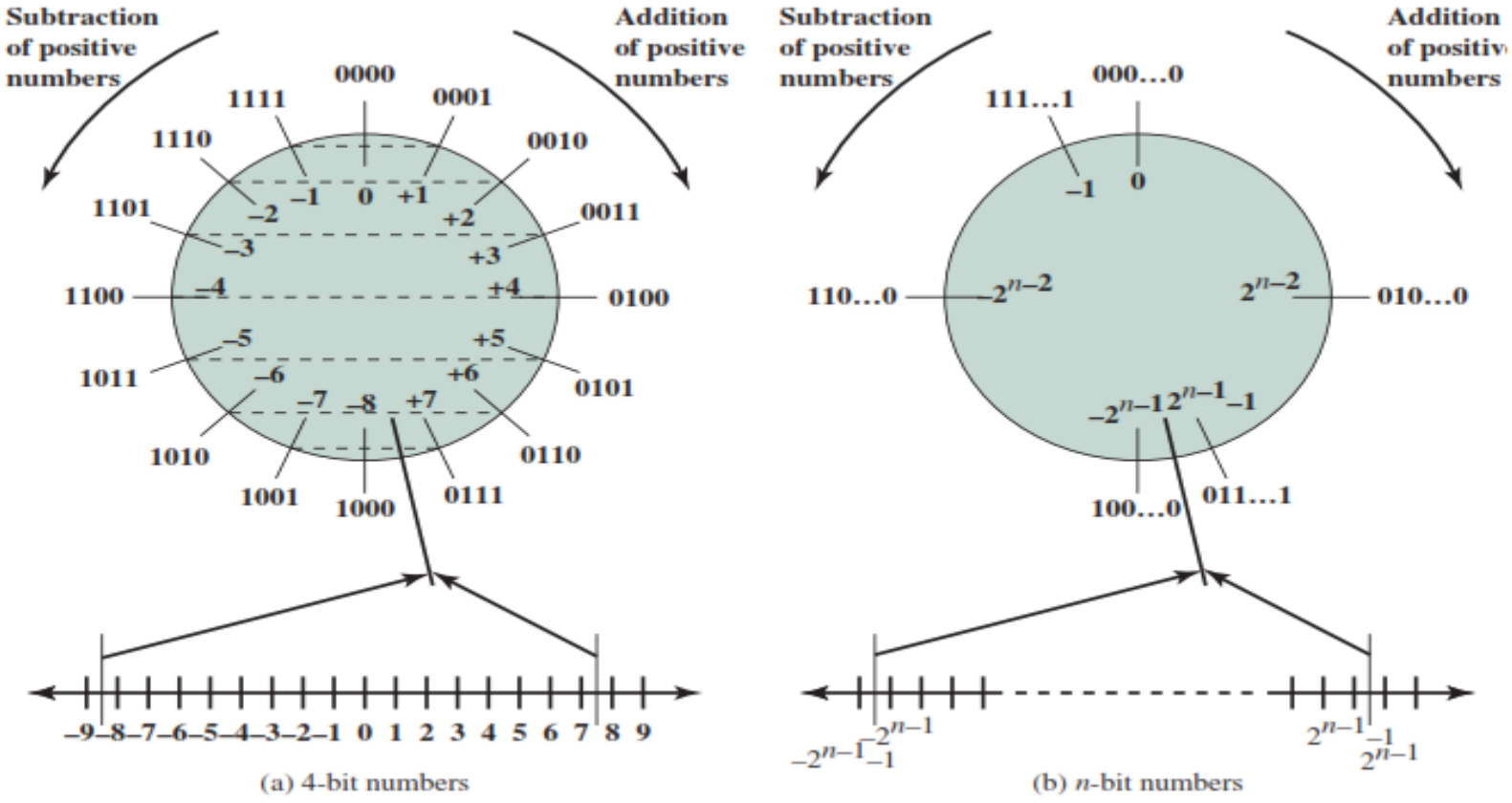
$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

  2. 2's complement

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

  3. 1's complement

Geometric Depiction of Twos Complement Integers



Overflow

- If the result of an arithmetic operation is to too large (positive or negative) to fit into the resultant bit-group, then arithmetic o**verflow** occurs.
- If 2 Two's Complement numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign.
- Overflow never occurs when adding operands with different signs.

Example

01010000 = 80

\+ 01010000 = 80

10100000 = −96

- Consider adding the numbers -7 and -6 represented in 2's complement using 4 bits. What is the result of the computation?

7 → -7: 0 1 1 1 → 1 0 0 0 → 1 0 0 1

6 → -6: 0 1 1 0 → 1 0 0 1 → 1 0 1 0

---

0 0 1 1

Result is positive (3)! Overflow.

Inputs			Outputs	
x_{n-1}	y_{n-1}	c_{n-2}	z_{n-1}	v
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0

$$v = \bar{x}_{n-1}\bar{y}_{n-1}c_{n-2} + x_{n-1}y_{n-1}\bar{c}_{n-2}$$

Sign extension rule

- Representing a number using more bits
- Preserve the numeric value
  - For 2's complement replicate the sign bit to the left
  - unsigned values: extend with 0s
- Example for 2's complement: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Decimal Number Code

Advantage of Decimal Codes are
I. Ease of Conversion between the internal computer representation.
2. Easily interpreted by Humans.

Disadvantage of decimal codes are
I. They use more bits to represent a number than the binary codes
2. The circuitry required to perform arithmetic using decimal operands is more complex than that needed for binary arithmetics

Example of decimal number codes are
BCD, ASCII, EXCESS 3 , Two out of five code.

Hexadecimal Numbers: Numbers are ranging from 0,1,2,...F

Floating Point Numbers

Floating-point numbers - provide a dynamic range of representable real numbers without having to scale the operands
Given a fixed number of digits, the floating-point representation covers a wider range of values compared to a fixed-point representation.

Representation - similar to scientific notation

Two parts - significand (or mantissa) M and exponent (or characteristic) E

The floating-point number F represented by the pair (M,E) has the value - the value $F=MB^e$ (- base of exponent)

Base - common to all numbers in a given system - implied
- not included in the representation of a floating point number

What can be the minimum and maximum number for a 3 bit exponent and 3 bit mantissa?

This can be represented as shown in the figure

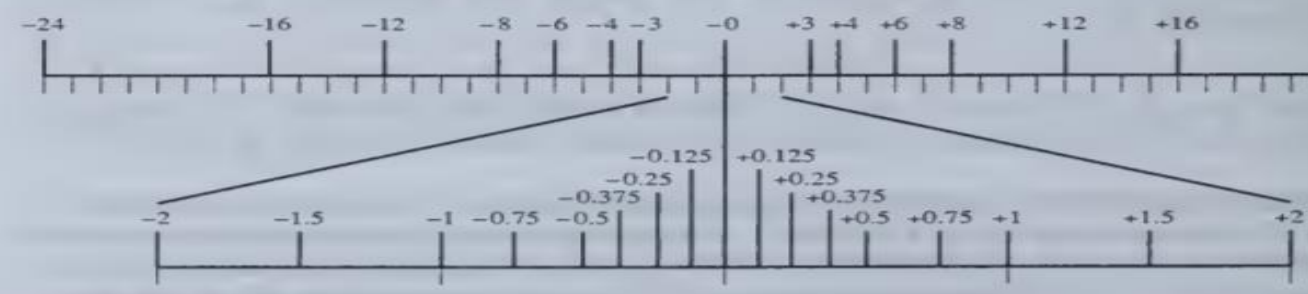


Figure 3.24
The real numbers representable by a hypothetical 6-bit, floating-point format.

Floating point Numbers

Examples of floating-point numers in base 2 …

1.00101×2^{23} , 0.0100101×2^{25} ,

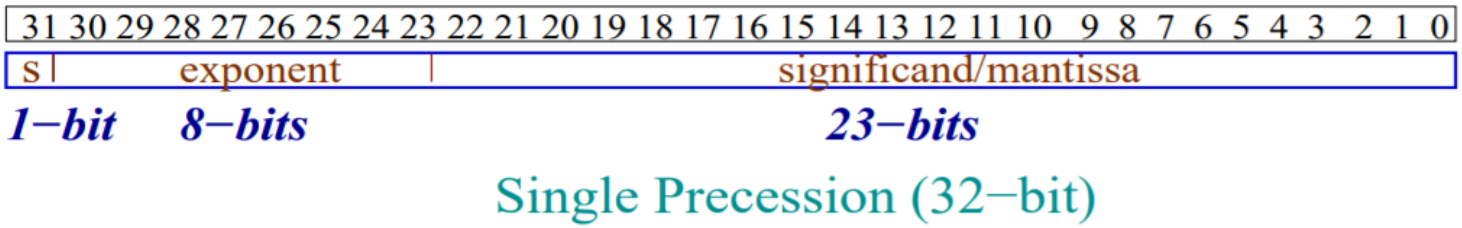
-1.101101×2^{-3} , -1101.101×2^{-6}

- Floating-point numbers should be normalized
- Exactly one non-zero digit should appear before the point
- In a decimal number, this digit can be from 1 to 9
- In a binary number, this digit should be 1
- Normalized FP Numbers: 5.341×10^{3} and -1.101101×2^{-3}
- NOT Normalized: 0.05341×10^{5} and -1101.101×2^{-6} Floating-Point Numbers decimal point binary point

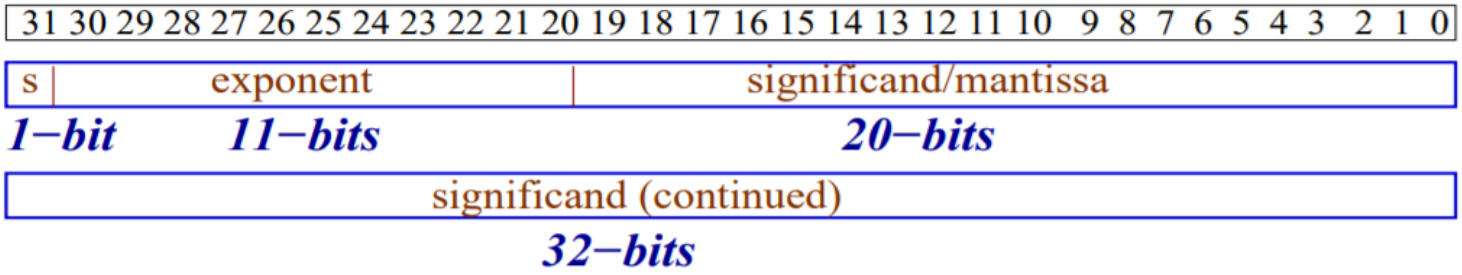
When necessary, numbers are shifted left by increasing the exponent until leading zeros are eliminated. This process is called **Normalization**.

IEEE floating point representation

- The IEEE (Institute of Electrical and Electronic Engineers) is an international organization that has designed specific binary formats for storing floating point numbers.
- The IEEE defines two different formats with different precisions: single and double precision. Single precision is used by float variables in C and double precision is used by double variables.
- Intel's math coprocessor also uses a third, higher precision called extended precision. In fact, all data in the coprocessor itself is in this precision. When it is stored in memory from the coprocessor it is converted to either single or double precision automatically.



Single Precession (32−bit)



Double Precession (64−bit)

Types of Data

Data represented in this format are classified in five groups.

- Normalized numbers,
- Zeros,
- Subnormal(denormal) numbers,
- Infinity
- not-a-number (nan).

Special meanings for IEEE floats.

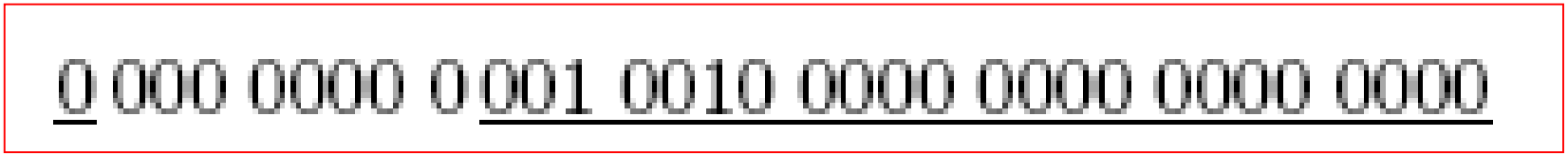
$e = 0$	and $f = 0$	denotes the number zero (which can not malized) Note that there is a +0 and -0
$e = 0$	and $f \neq 0$	denotes a *denormalized number*. These cussed in the next section.
$e = \mathrm{FF}$	and $f = 0$	denotes infinity (∞). There are both and negative infinities.
$e = \mathrm{FF}$	and $f \neq 0$	denotes an undefined result, known (Not a Number).

An undefined result is produced by an invalid operation such as trying to find the square root of a negative number, adding two infinities, etc.

- **Normalized single precision numbers can range in magnitude from 1.0×2^{-126} ($\approx 1.1755 \times 10^{-35}$) to $1.11111\ldots \times 2^{127}$ ($\approx 3.4028 \times 10^{35}$).**

Denormalized numbers

- Denormalized numbers can be used to represent numbers with magnitudes too small to normalize (i.e. below 1.0×2^{-126}).
- E.g., $1.001_2 \times 2^{-129}$ ($\approx 1.6530 \times 10^{-39}$). in the unnormalized form: $0.01001_2 \times 2^{-127}$.
- To store this number, the biased exponent is set to 0 and the fraction is the complete significand of the number written as a product with 2^{-127}



Why biased exponent?

- **For faster comparisons (for sorting, etc.), allow integer comparisons of floating point numbers:**

- **Unbiased exponent:**

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

- **Biased exponent:**

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

How would 23.85 be stored?

10111.1101100110.

- First, it is positive so the sign bit is 0.
- Next, the true exponent is 4, so the biased exponent is $7F+4 = 83_{16}$.
- Finally, the fraction is 01111101100110011001100 (remember the leading one is hidden).

- -23 $\underline{0}\,100\ 0001\ 1\underline{011\ 1110\ 1100\ 1100\ 1100\ 1100}_2 = 41BECCCC_{16}$

CC CD. Do not take the two's complement!

INSTRUCTION SET

Instruction Format : The purpose of an instruction is to specify both an operation to be carried out and the set of operands.

In the assembly code most of the instructions form are

OP x1,x2,x3…

The operation is specified by opcode OP and
Xi represents the address field either register or memory.

INSTRUCTION FORMATS

•The next consideration for architecture design concerns how the CPU will store data.

- We have three choices:
    1. A stack architecture
    2. An accumulator architecture
    3. A general purpose register architecture.
- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.

Stack Architecture

- In a stack architecture, instructions and operands are implicitly taken from the stack.
  - A stack cannot be accessed randomly.
- In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.
  - One operand is in memory, creating lots of bus traffic.
- In a general purpose register (GPR) architecture, registers can be used instead of memory.
  - Faster than accumulator architecture.
  - Efficient implementation for compilers.
  - Results in longer instructions.

- Most systems today are GPR systems.
- There are three types of instructions:
  - Memory-memory where two or three operands may be in memory.
  - Register-memory where at least one operand must be in a register.
  - Load-store where no operands may be in memory.
- The number of operands and the number of available registers has a direct affect on instruction length.

- Stack machines use one- and zero-operand instructions.
- **`LOAD`** and **`STORE`** instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- **`PUSH`** and **`POP`** operations involve only the stack's top element.
- Binary instructions (e.g., **`ADD`**, **`MULT`**) use the top two items on the stack.

- Stack architectures require us to think about arithmetic expressions a little differently.
- We are accustomed to writing expressions using *infix* notation, such as: Z = X + Y.
- Stack arithmetic requires that we use *postfix* notation: Z = XY+.
  - This is also called *reverse Polish notation*.

- The principal advantage of postfix notation is that parentheses are not used.

- For example, the infix expression,
  Z = (X * Y) + (W * U), becomes:
  Z = X Y * W U * + in postfix notation.
  In a stack ISA, the postfix expression,
  Z**= X Y * W U * +**
  might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
PUSH Z
```

• In a one-address ISA, the infix expression,

Z = X * Y + W * U

looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

• In a two-address ISA, (e.g.,Intel, Motorola), the infix expression,

Z = X * Y + W * U

might look like this:

```
LOAD R1,X
MULT R1,Y
LOAD R2,W
MULT R2,U
ADD  R1,R2
STORE Z,R1
```

- With a three-address ISA, (e.g.,mainframes), the infix expression,

Z = X * Y + W * U

might look like this:

```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

ADDRESSING MODES

- Addressing modes are concerned with how the CPU accesses the operands used by its instructions.

- Specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

- Computers use addressing mode techniques for the following purposes:-

    1.) To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data and various other purposes.

    2.) To reduce the number of bits in the addressing field of the instructions. Other computers use a single binary for operation & Address mode. The mode field is used to locate the operand. Address field may designate a memory address or a processor register. There are 2 modes that need no address field at all (Implied & immediate modes).

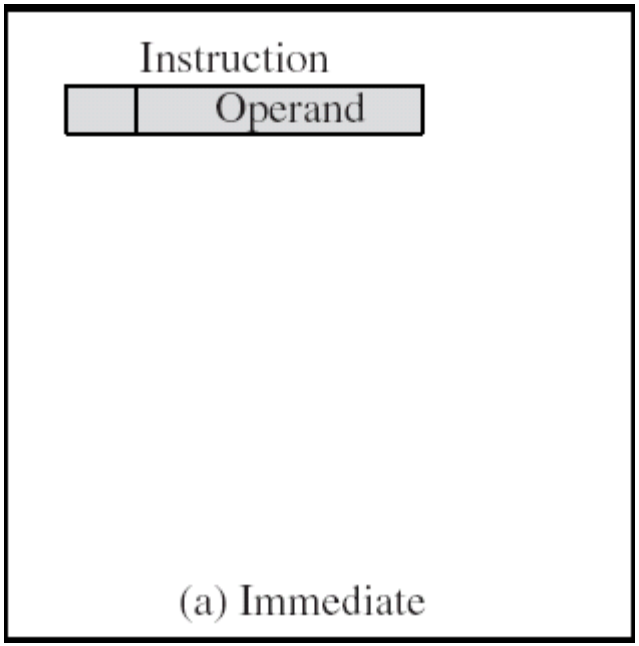
Addressing Modes

Different Addressing Modes are

1. Direct
2. Indirect
3. Register
4. Immediate
5. Register Indirect
6. Displacement (Indexed)
7. Stacks

Immediate Addressing

- Operand is part of instruction
- Operand = address field
- e.g. ADI A, 5h
- MVI B, 15
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range



Direct Addressing

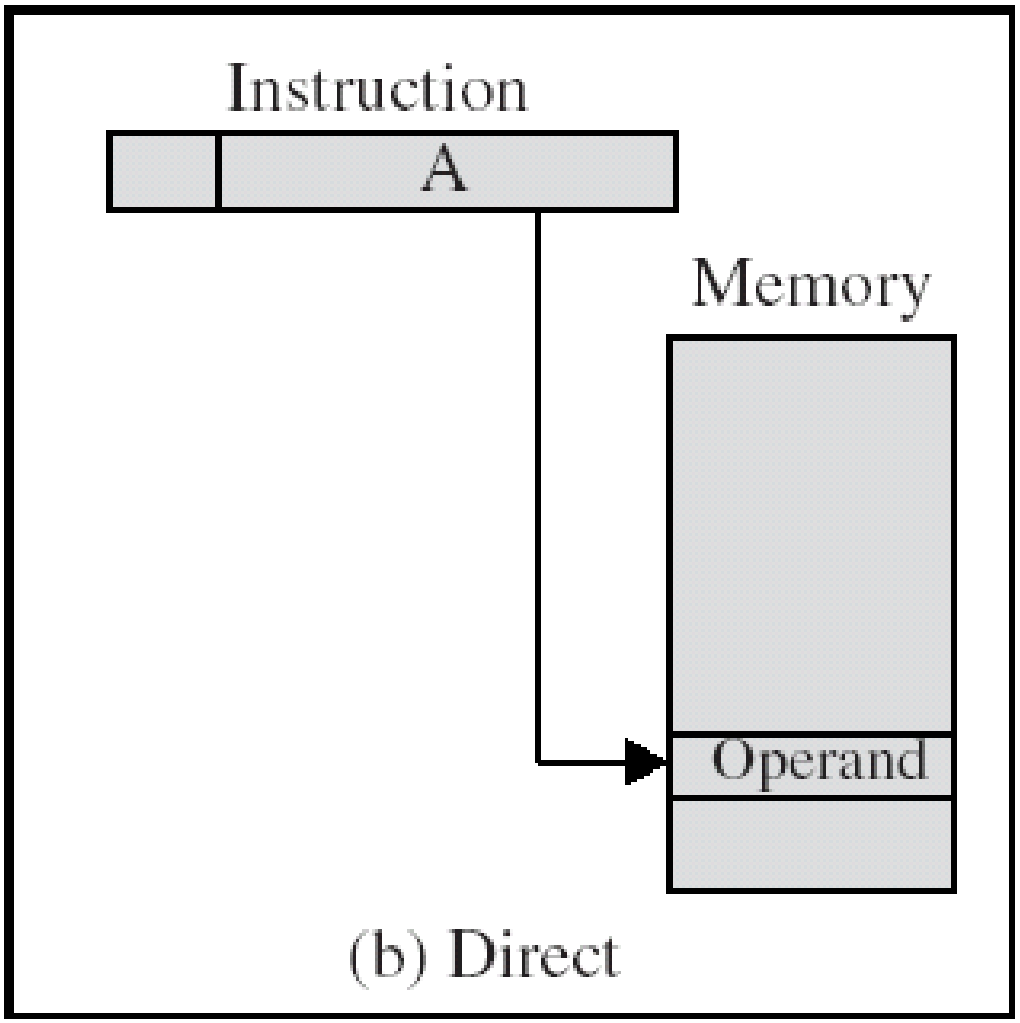
- Address field contains address of operand
- Effective address EA = address field (A)

ADD AX, value

Value DB 05h

  - Add contents of cell value to accumulator AX
  - Look in memory at address value for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space

Direct Addressing Diagram



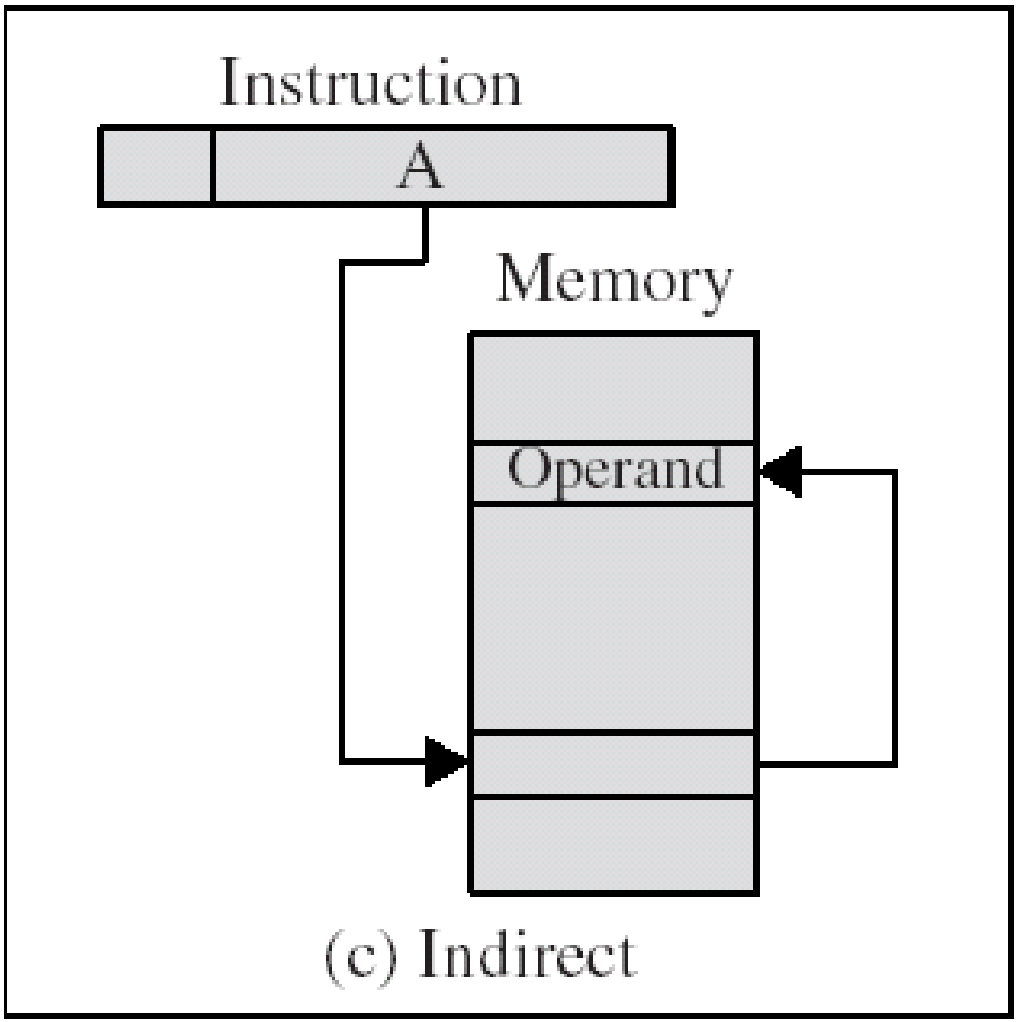
Indirect Addressing (1/2)

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- EA =(A)
  - Look in A, find address (A) and look there for operand
- e.g. ADD AX, (A)
  - Add contents of cell pointed to by contents of A to accumulator

Indirect Addressing (2/2)

- Large address space
- 2^n where n = word length
- May be nested, multilevel, cascaded
  - e.g. EA = (((A)))
- Multiple memory accesses to find operand
- Hence slower

Indirect Addressing Diagram



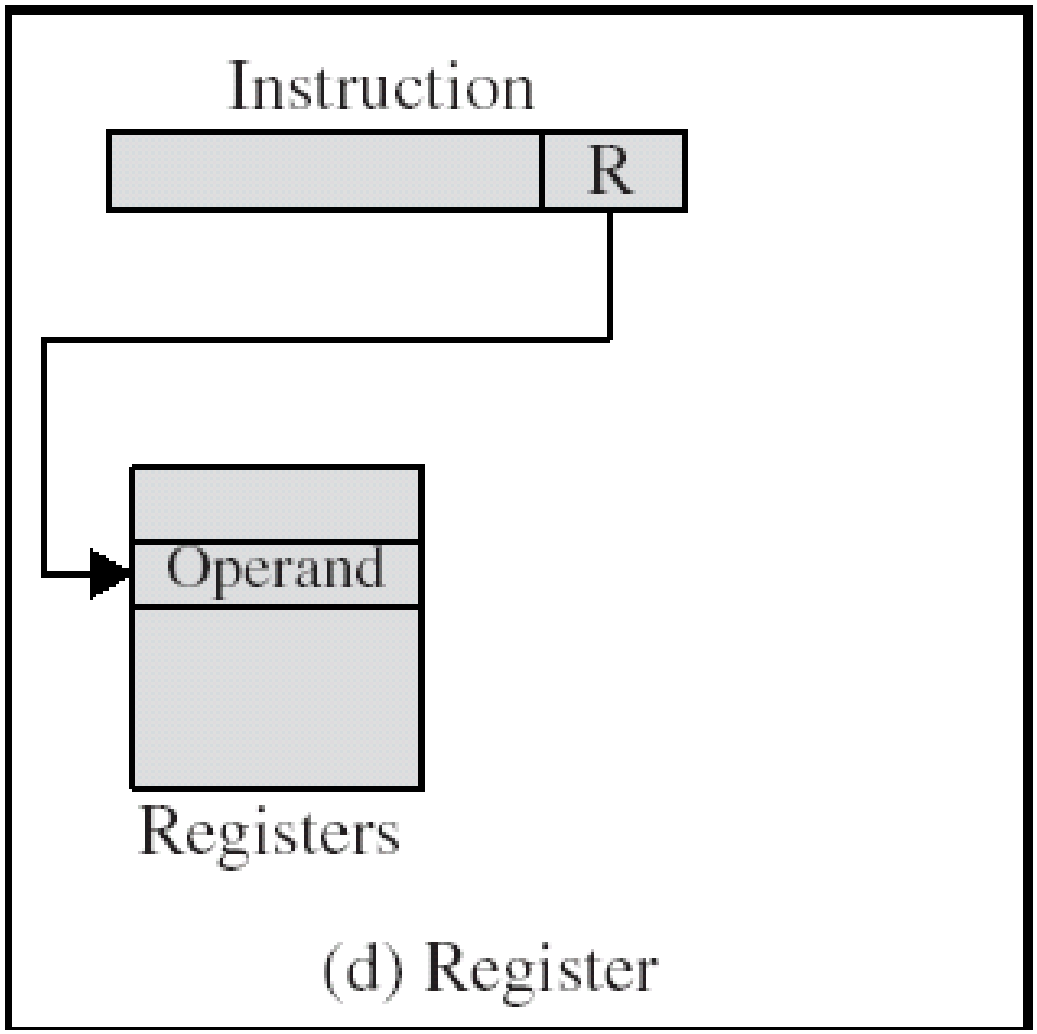
Register Addressing (1/2)

- Operand is held in register named in address filed
- EA = R
- Limited number of registers
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch
  - MOV AX, BX
  - ADD AX, BX

Register Addressing (2/2)

- No memory access
- Very fast execution
- Very limited address space
- Multiple registers helps performance
  - Requires good assembly programming or compiler writing
  - For Ex. In C programming
    - register int a;

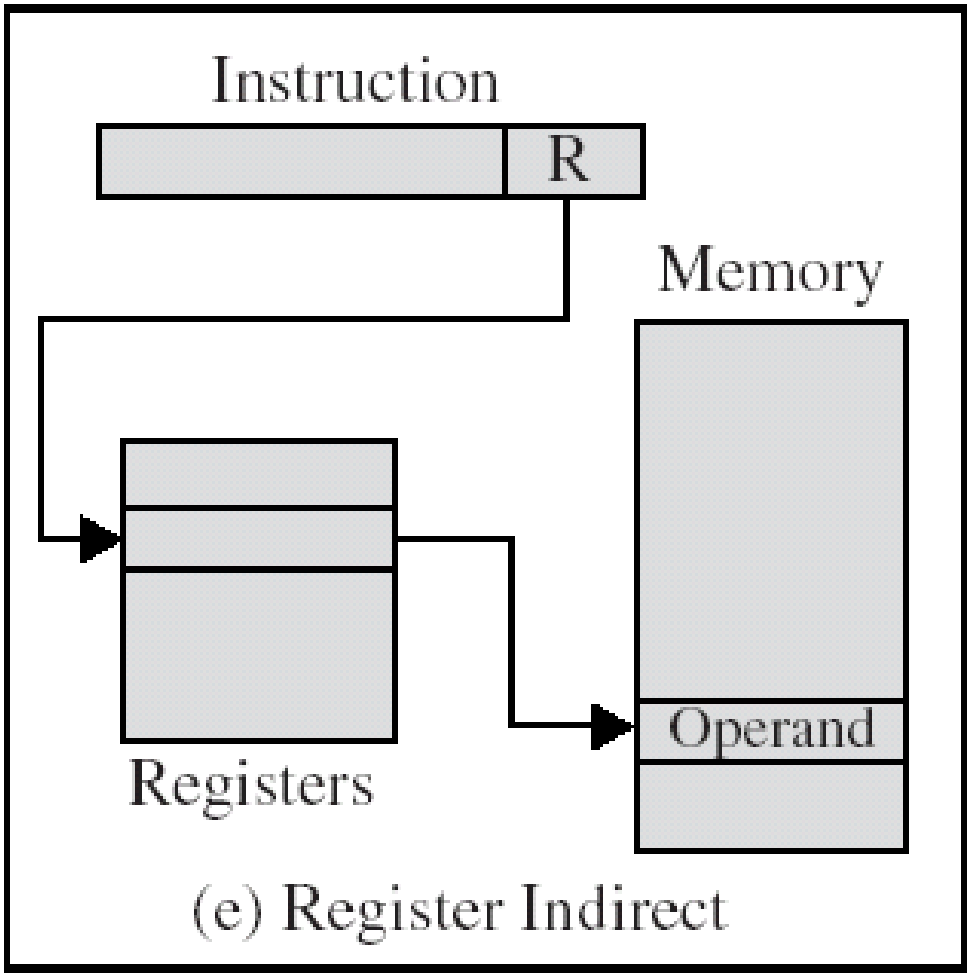
Register Addressing Diagram



Register Indirect Addressing

- C.f. indirect addressing
- EA = (R)
- Operand is in memory cell pointed to by contents of register R
- Large address space (2^n)
- One fewer memory access than indirect addressing

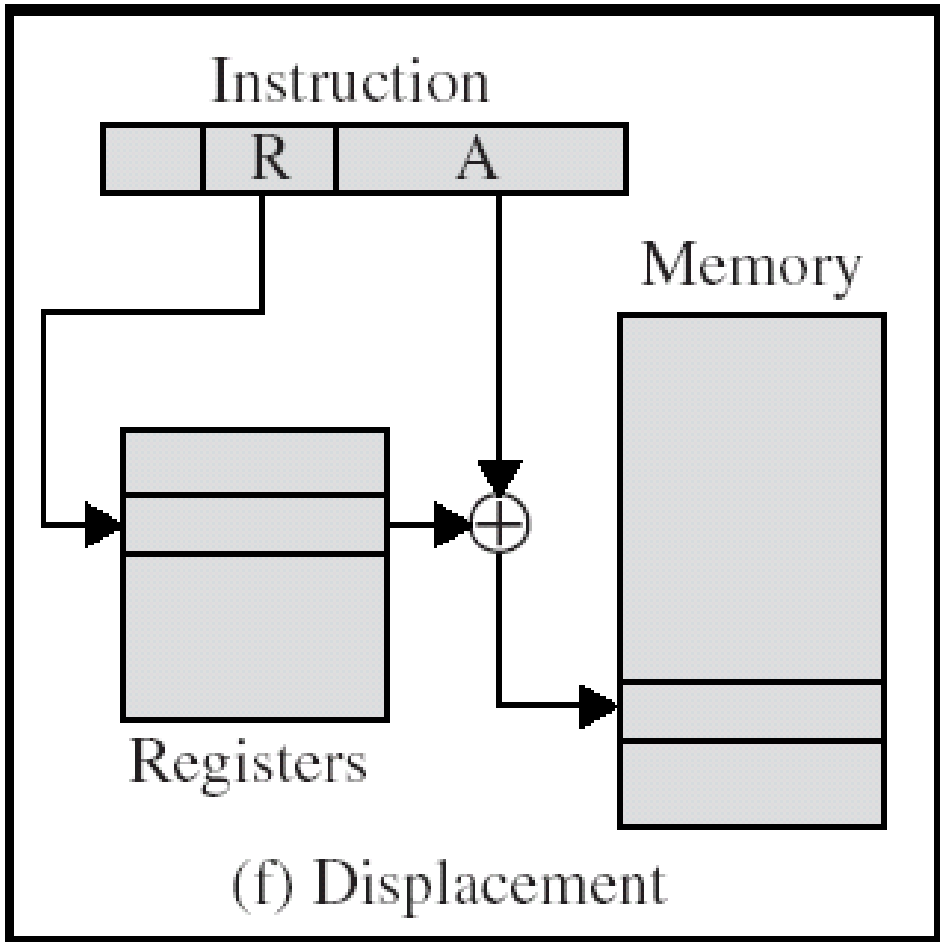
Register Indirect Addressing Diagram



Displacement Addressing

- EA = A + (R)
- Effective address=start address + displacement
- Effective address=Offset + (Segment Register)
- Use direct and register indirect
- Address field hold two values
  - A = base value
  - R = register that holds displacement or vice versa

Displacement Addressing Diagram



Relative Addressing (PC-Relative)

- A version of displacement addressing
- R = Program counter, PC
- EA = A + (PC)
- i.e. get operand from A cells from current location pointed to by PC

  locality of reference & cache usage

Base-Register Addressing

- A holds displacement
  - EA = (CS) + A
- R holds pointer to base address
- R may be explicit or implicit
- e.g. segment registers in 80x86

Indexed Addressing

- A = base
- R = displacement
  - EA = A + (R)
- Good for accessing arrays
  - EA = A + (R)
  - R++

Central Processing Unit

Addressing modes

Numerical Example

PC=200	R1=400
XR=100	AC

Addressing mode	eff. Add	Content of AC
Direct Address	500	800
Immediate operand	201	500
Indirect Address	800	300
Relative Address	702(PC=PC+2)	325
Indexes Address	600(XR+500)	900
Register	---	400
Register Indirect	400	700
Auto-increment	400	700
Auto-decrement	399	450

Tabular list

Address	Memory
200	Load to AC | Mode
201	Address=500
202	Next Instruction
399	450
400	700
500	800
600	900
702	325
800	300

Encoding an Instruction Set

- The instructions are encoded into a binary representation for execution by the processor.
- This representation affects not only the size of the compiled program but also the implementation of the processor, which must decode this representation to quickly find the operation and its operands.
- The operation is typically specified in one field, called the *opcode.*

Encoding an Instruction Set

The architect must balance several competing forces when encoding the instruction set:

1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation.

Encoding of instructions

To encode an instruction a unique numeric opcode value for each instruction is assigned.
With an n-bit number, there are 2^n different possible opcodes
If you have 128 truly unique instructions, there's little you can do other than to decode each instruction individually. However, in most architectures the instructions are not completely independent of one another.
For Ex: MOV A,B and MOV C,D
we could encode instructions like MOV with a sub-opcode and encode the operands using other strings of bits within the opcode.

Types of Opcode

CPUs with unnecessarily long instructions consume extra memory for their programs.

The overall performance of the CPU will be affected.

Decision depends on range of addressing modes supported

- Fixed Length – Simple, easily decoded
  – Larger code size
- Variable Length – More complex, harder to decode
  – More compact, efficient use of memory
    - Fewer memory references
  – Complex pipeline: instructions vary greatly in both size and amount of work to be performed

Expanded Opcode

•For a given instruction format length, there is a trade-off between the number of bits used for the op-code and the number used for operands (e.g., addresses)

- More op-code bits means more instructions
- More address bits means more addressable locations

•By using specific prefixes of the op-code, you can get varying number of variable length op-codes

•In any instruction set, not all instructions require the same number of operands.

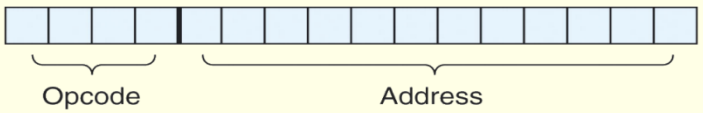
- Operations that require no operands, such as HALT, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use **expanding opcodes**.

Expanding Opcodes

- **Example 1:** Consider a machine with 16-bit instructions and 16 registers.
  - The instruction format can have several structures:
    - Opcode + Memory address:

      → If we have 4KB byte addressable memory we need 12 bits to specify an address location

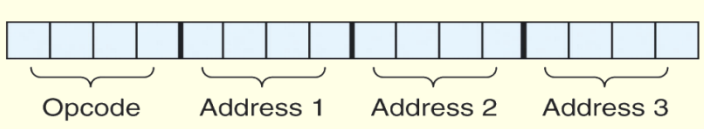
      → The remaining 4 bits are used for the opcode: 16 instruction are hence available



    - Opcode + Registers Addresses

      → we need 4 bits to select one of the 16 available registers

      → Suppose we have 4 bits opcode, we could encode 16 different instructions with three operands each (3 x 4 bits = 12 bits).

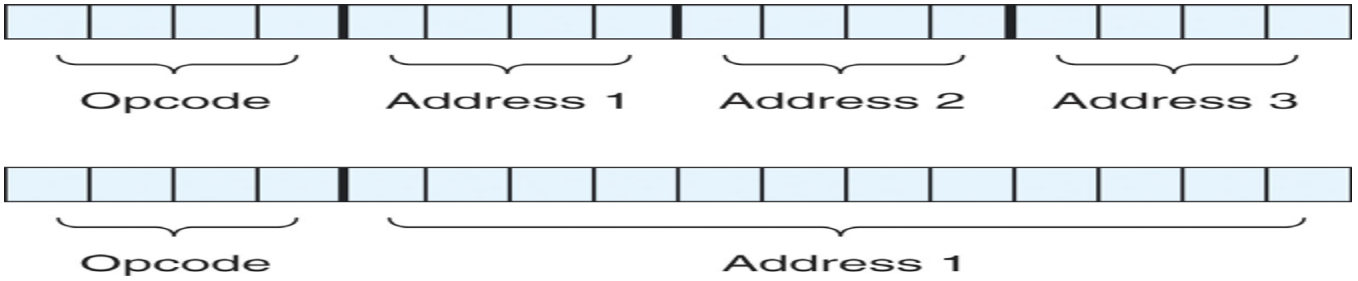


Expanding Opcodes

- **Example 2:** Consider a machine with 16-bit instructions and 16 registers. And we wish to encode the following instructions:
  - 15 instructions with 3 addresses
  - 14 instructions with 2 addresses
  - 31 instructions with 1 address
  - 16 instructions with 0 addresses

  Can we encode this instruction set in 16 bits?
  - **Answer:** Yes if we use **expanding opcodes**

Example(Contd)




```
0000 R1   R2   R3
...                    } 15 3-address codes
1110 R1   R2   R3

1111 0000 R1   R2
...                    } 14 2-address codes
1111 1101 R1   R2

1111 1110 0000 R1
...                    } 31 1-address codes
1111 1111 1110 R1

1111 1111 1111 0000
...                    } 16 0-address codes
1111 1111 1111 1111
```

Expanding Opcodes

- How do we know if the instruction set we want is possible when using expanding opcodes?
  - We must determine if we have enough bits to create the desired number of bits patterns

Expanding Opcodes

- **Going back to Example 2:**
  - The first 15 instructions account for:
    $15x2^4x2^4x2^4 = 15 \times 2^{12} = 61440$ bit patterns
  - The next 14 instructions account for:
    $14 \times 2^4 \times 2^4 = 15 \times 2^8 = 3584$ bit patterns
  - The next 31 instructions account for:
    $31 \times 2^4 = 496$ bit patterns
  - The last 16 instructions account for 16 bit patterns
  - In total we need 61440 + 3584 + 496 + 16 = 65536 different bit patterns
  - Having a total of 16 bits we can create $2^{16} = 65536$ bit patterns
  - **We have an exact match with no wasted patterns.**
  - **So our instruction set is possible.**

Expanding Opcodes

- **Example 3:** Is it possible to design an expanding opcode to allow the following to be encoded with a 12-bit instruction? Assume a register operand requires 3 bits.
  - **4 instructions with 3 registers**
  - **255 instructions with 1 register**
  - **16 instructions with 0 register**

Expanding Opcodes

- **Solution:**
  - The first 4 instructions account for:
    - $4x2^3x2^3x2^3 = 4 \times 2^9 = 2048$ bit patterns
  - The next 255 instructions account for:
    - $255 \times 2^3 = 2040$ bit patterns
  - The last 16 instructions account for 16 bit patterns
  - In total we need 2048 + 2040 + 16 = 4104 bit patterns
  - With 12 bit instruction we can only have $2^{12} = 4096$ bit patterns
  - **Required bit patterns (4104) is more than what we have (4096), so this instruction set is not possible with only 12 bits.**

- Requirements to be satisfied by an instruction set are
  - It should be complete (i.e any function can be constructed using a reasonable amount of memory space)
  - It should be efficient (i.e frequently required functions can be performed rapidly using few instructions)
  - It should be regular (i.e the instruction set should contain expected opcodes and addressing modes)
  - It should be compatible with existing machines

- Instructions are divided into the following groups
  - Data transfer instructions
  - Arithmetic instructions
  - Logical instructions
  - Program Control Instructions
  - Input-Output instructions

Error Detection and Control

Parity Checking:

- It is also known as a parity check.
- It has a cost-effective mechanism for error detection.
- In this technique, the redundant bit is known as a parity bit. It is appended for every data unit. The total number of 1s in the unit should become even, which is known as a parity bit.

Hamming code

- Hamming code is a linear code that is useful for error detection up to two bit errors. It is capable of correcting single-bit errors.
- The ease of use of hamming codes makes it suitable for use in computer memory and single-error correction.
- *N* data bits or *K* check bits, we must have $2^K\text{-}1 \geq \mathrm{N} + K$

Hamming Code(Contd)

- *k* parity bits are added to an *n*-bit data word, forming a new word of *n* + *k* bits.
- The bit positions are numbered in sequence from 1 to *n* + *k*.
- Those positions numbered with powers of two are reserved for the parity bits.
- The remaining bits are the data bits.
- The code can be used with words of any length.

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1

Hamming Code(Contd)

C1 = D1⊕D2⊕ D4⊕D5⊕ D7
C2 = D1⊕ D3⊕D4⊕ D6⊕D7
C4 = D2⊕D3⊕D4 ⊕ D8
C8 = D5⊕D6⊕D7⊕D8
Let the Data bits stored are : 0 0 1 1 1 0 0 1
Data is fetched as : 0 0 1 1 0 0 0 1

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1
Word stored as	0	0	1	1	0	1	0	0	1	1	1	1
Word fetched as	0	0	1	1	0	1	1	0	1	1	1	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Check bit					0				0		0	1

Figure 5.10 Check Bit Calculation

Hamming Code(Contd)

Checksum for the stored word
C1 = 1⊕0⊕1⊕1⊕0 = 1
C2 = 1⊕0⊕1⊕1⊕0 = 1
C4 = 0⊕0⊕1⊕0 = 1
C8 = 1⊕1⊕0⊕0 = 0

Checksum for the fetched word
C1 = 1⊕0⊕1⊕1⊕0 = 1
C2 = 1⊕1⊕1⊕1⊕0 = 0
C4 = 0⊕1⊕1⊕0 = 0
C8 = 1⊕1⊕0⊕0 = 0

	C8	C4	C2	C1
	0	1	1	1
⊕	0	0	0	1
	0	1	1	0

This indicates that there is error at 6th position.

Hamming Code(Contd)

The basic Hamming code can detect and correct an error in only a single bit. Multiple-bit errors are detected, but they may be corrected erroneously, as if they were single-bit errors.

By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors.

If we include this additional parity bit in the 12-bit coded word becomes

$001110010100P_{13}$,

P_{13} is evaluated from the exclusive-OR of the other 12 bits. This produces the 13-bit word

The following four cases can occur:

If $C = 0$ and $P = 0$ No error occurred.

If $C \neq 0$ and $P = 1$ A single error occurred that can be corrected.

If $C \neq 0$ and $P = 0$ A double error occurred that is detected but cannot be corrected.

If $C = 0$ and $P = 1$ An error occurred in the $P13$ bit.

RISC and CISC Processor

- RISC-Reduced Instruction Set Processor
- Relatively few instruction types and addressing modes
- Fast,single cycle instruction execution
- Fixed and easily decoded instruction format
- Hardwired Control unit
- Memory access limited to Load and Store instructions.
- Use of compilers to optimize object code performance.

CISC

- Complex Instruction Set computers
- Complex Instructions and addressing modes
- Instruction decoder is complex as variable length instruction is used.
- Multicycle instruction execution.
- Microprogrammed control unit is designed
- Many instructions can perform the operation with the memory contents.
- The compiler requires little effort to translate high-level programs or statement languages into assembly or machine language.

MACROS and SUBROUTINES

For simplifying program group of instructions is treated as single entity.
Macro- Places a portion of assembly language code between appropriate directives

Syntax: name MACRO op1,op2…

…

ENDM

Ex: Ldai MACRO *adr*

LHLD *ADR*

MOV A,M

ENDM

Subroutine

A subroutine or procedure is a sequence of instructions that can be invoked by name, much like a single instruction.

```
CALL SUB1
Next:   …
        …
SUB1    …
        …
        RETURN
```

THANK YOU