SECR2033 Computer Organization and Architecture

Module 4
Instruction Set Architecture (ISA)

Objectives:

- To provide a more detailed look at machine instruction sets.
- To look at different instruction types and operand types, and how instructions access data in memory.
- To understanding how instruction sets are designed and how their function can help to understand the more intricate details of the architecture of the machine itself.

Module 4
Instruction Set Architecture (ISA)

Module 4
Instruction Set Architecture (ISA)

- ❑ Elements of a Machine Instruction
- ❑ Instruction Representation
- ❑ Instruction Types
- ❑ Number of Addresses
- ❑ Instruction Set Design

Let’s recall

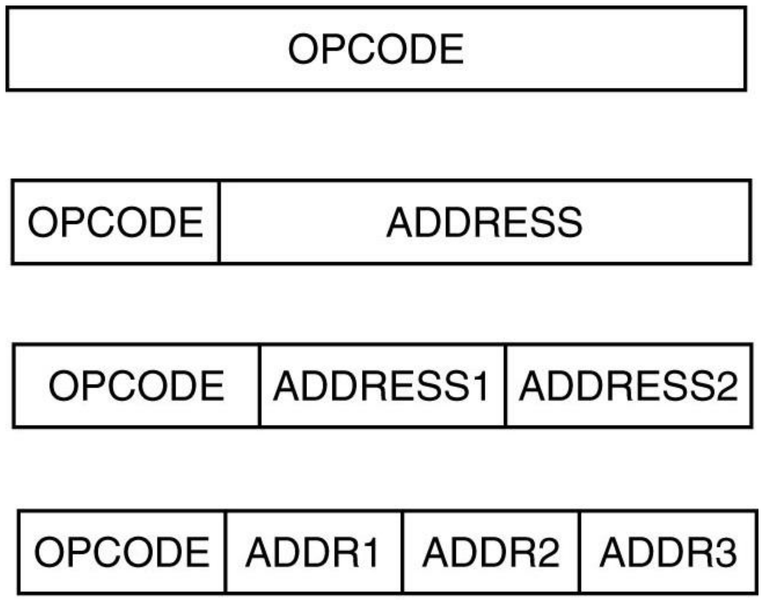
- Four common instruction formats:

(a) Zero-address instruction.

(b) One-address instruction

(c) Two-address instruction.

(d) Three-address instruction.



Example: Program to execute $Y = \frac{A - B}{C + (D \times E)}$

4 instructions

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

6 instructions

Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

8 instructions

Instruction		Comment
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

10 instructions

PUSH	C
PUSH	D
PUSH	E
MUL	
ADD	
PUSH	B
PUSH	A
SUB	
DIV	
POP	Y

(d) No-address instruction

How Many Addresses?

- Number of addresses per instruction is a basic design decision.

- More addresses:
  - More complex (powerful?) instructions.
  - Fewer instructions per program.
  - More registers:

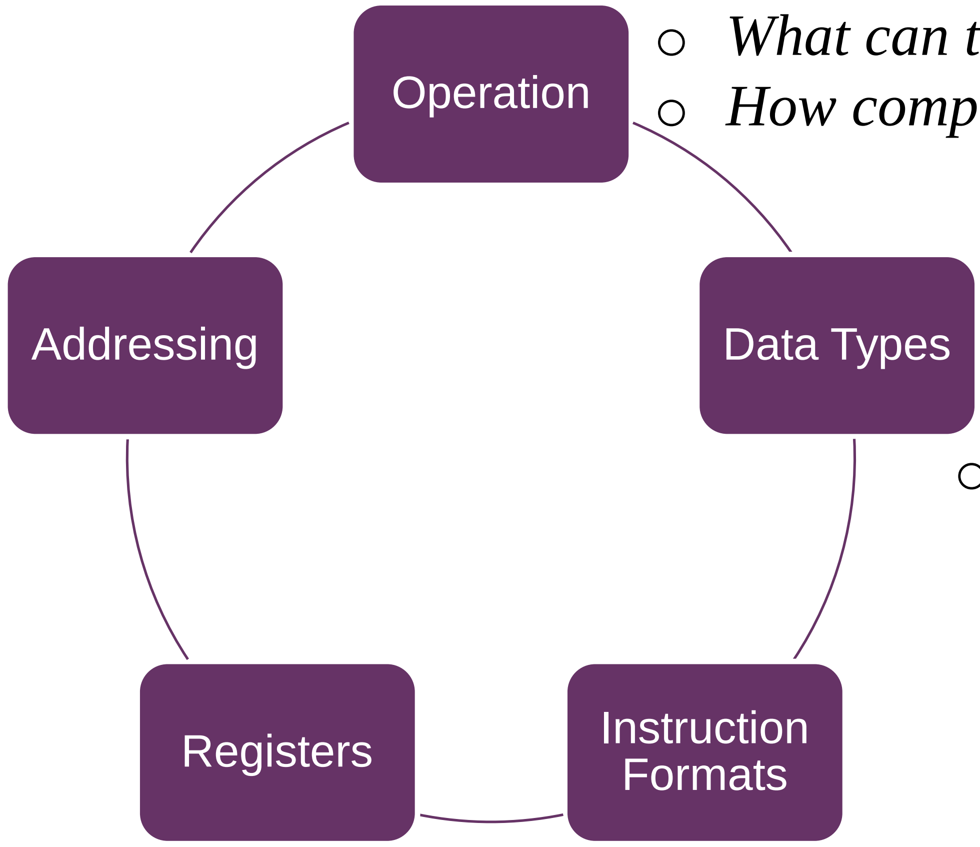
  → Inter-register operations are quicker.

- Fewer addresses:
  - Less complex (powerful?) instructions.
  - More instructions per program.
  - Faster fetch/execution of instructions.

Instruction Set Design

- The design of an instruction set is very complex because it affects so many aspects of the computer system.
- The instruction set defines many of the functions performed by the processor.
- The instruction set is the programmer's means of <u>controlling</u> the processor. Thus, programmer <u>requirements</u> must be considered in designing the instruction set.

- The most important of these fundamental design issues include the following:



- *How many operations?*
- *What can they do?*
- *How complex are they?*

- *The various types of data.*

- *Length of opcode field.*
- *Number of addresses.*

- *Number of CPU registers available.*
- *Which operations can be performed on which registers?*

- *The mode(s) by which the address of an operand is specified.*

Module 4
Instruction Set Architecture (ISA)

- ❑ Overview
- ❑ Numbers
- ❑ Characters
- ❑ Logical Data

Overview

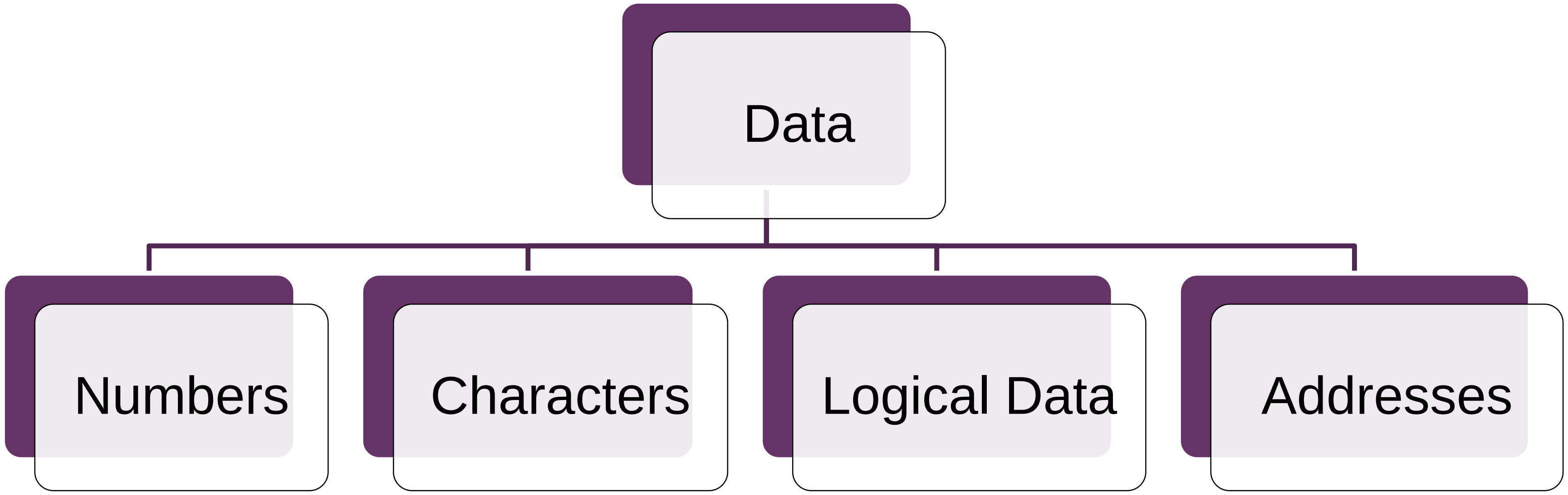
- Assembly language built from two pieces:

```
MOV   R1, R2
```

Opcode:
What to do with the data
(ALU operation)

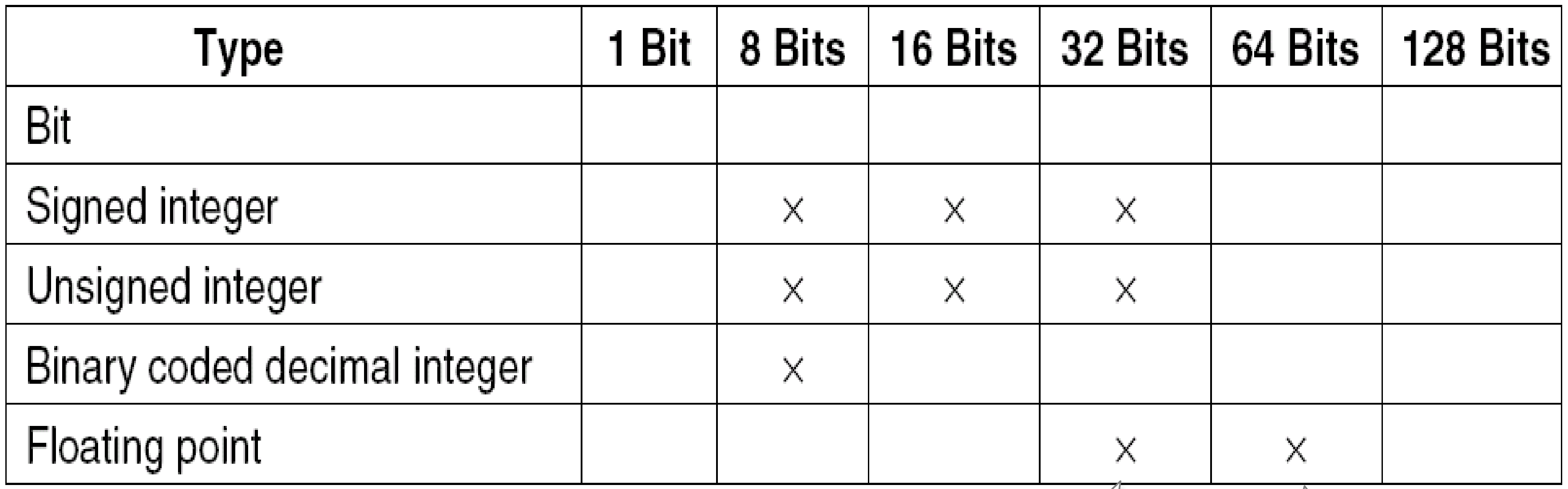
Operands
Where to get data (R2)
and put the results (R1)

- Machine instructions operate on data.
- The most important general categories of data:



(Section 4.4)

Table: Type of operand for Pentium 4



Type	1 Bit	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Bit						
Signed integer		×	×	×		
Unsigned integer		×	×	×		
Binary coded decimal integer		×				
Floating point				×	×	

Single Precision

Double Precision

Numbers

- All machine languages include numeric data types.
- An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited: (2 reasons)

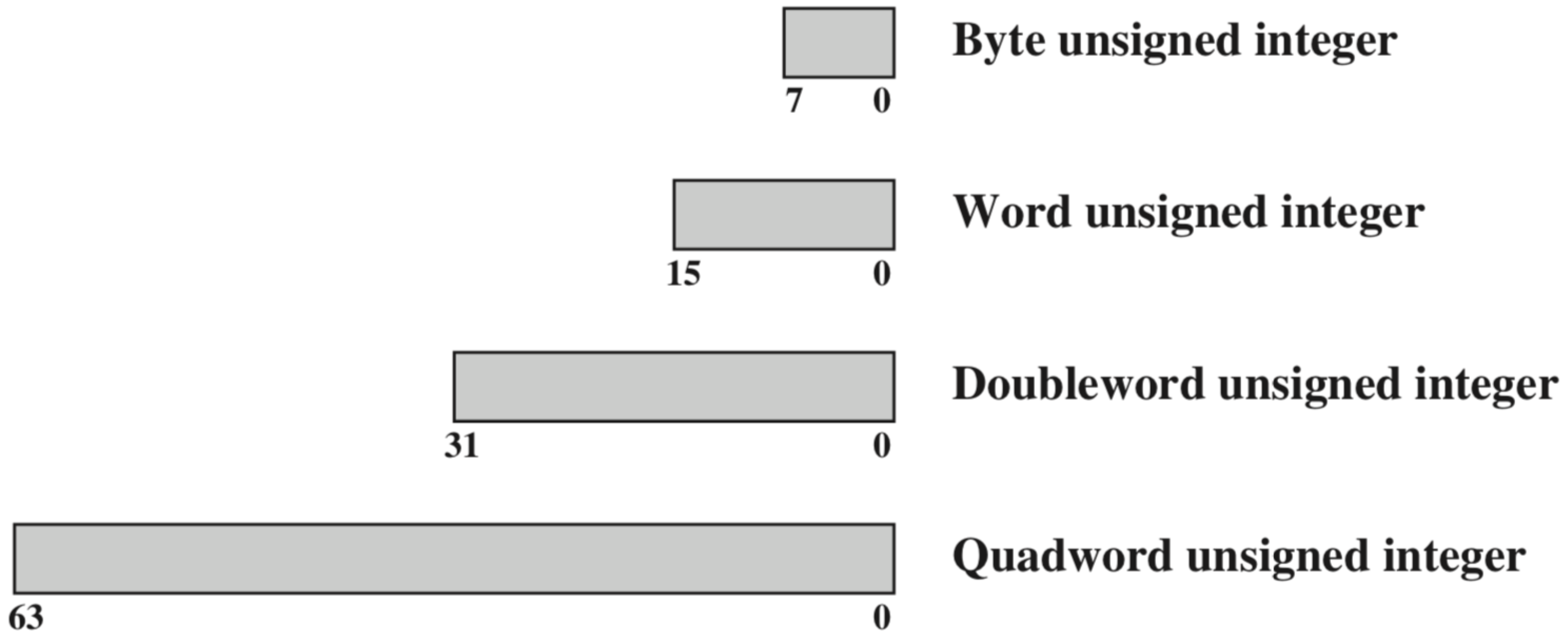
  - There is a limit to the magnitude of numbers representable on a machine.
  - A limit to the precision of floating-point numbers.

- 3 types of numerical data:
  - Binary integer or binary fixed point
  - Binary floating point
  - Decimal

Example 3a: x86 numerical data types.

(Unsigned integers)

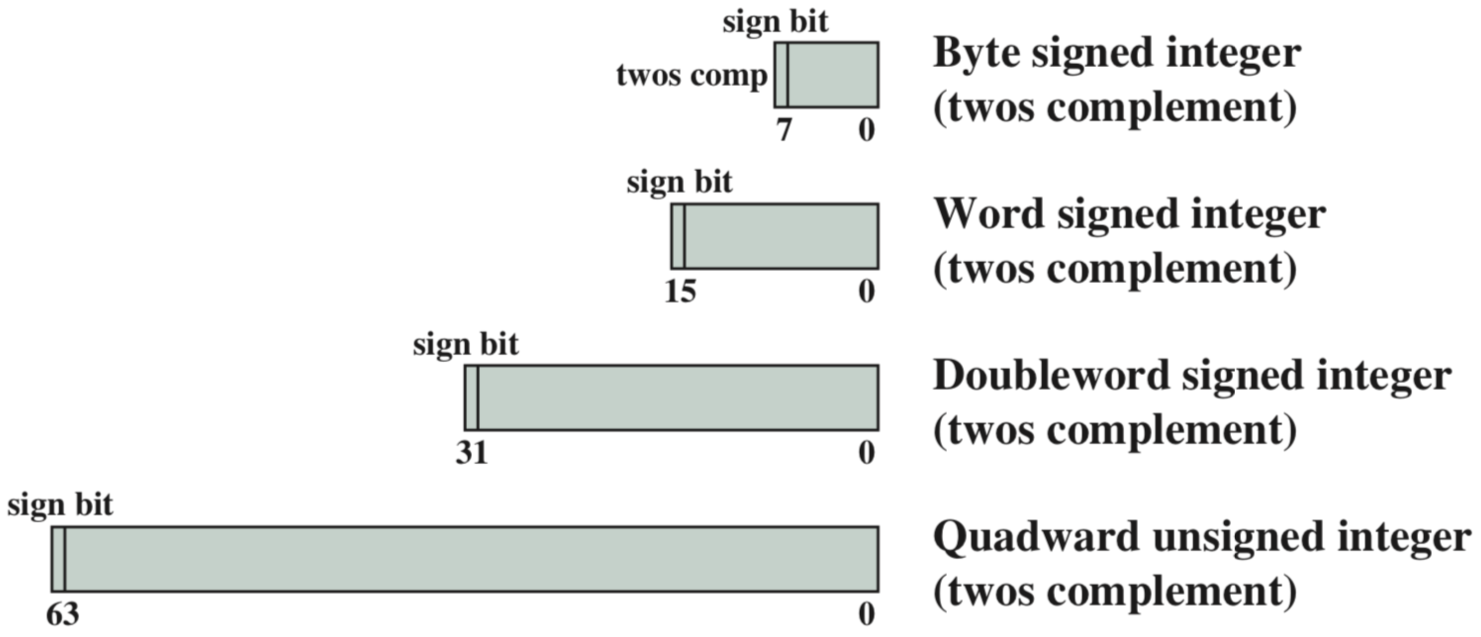
- The unsigned integers may be 16, 32, or 64 bits long.



Example 3b: x86 numerical data types.

(Signed integers)

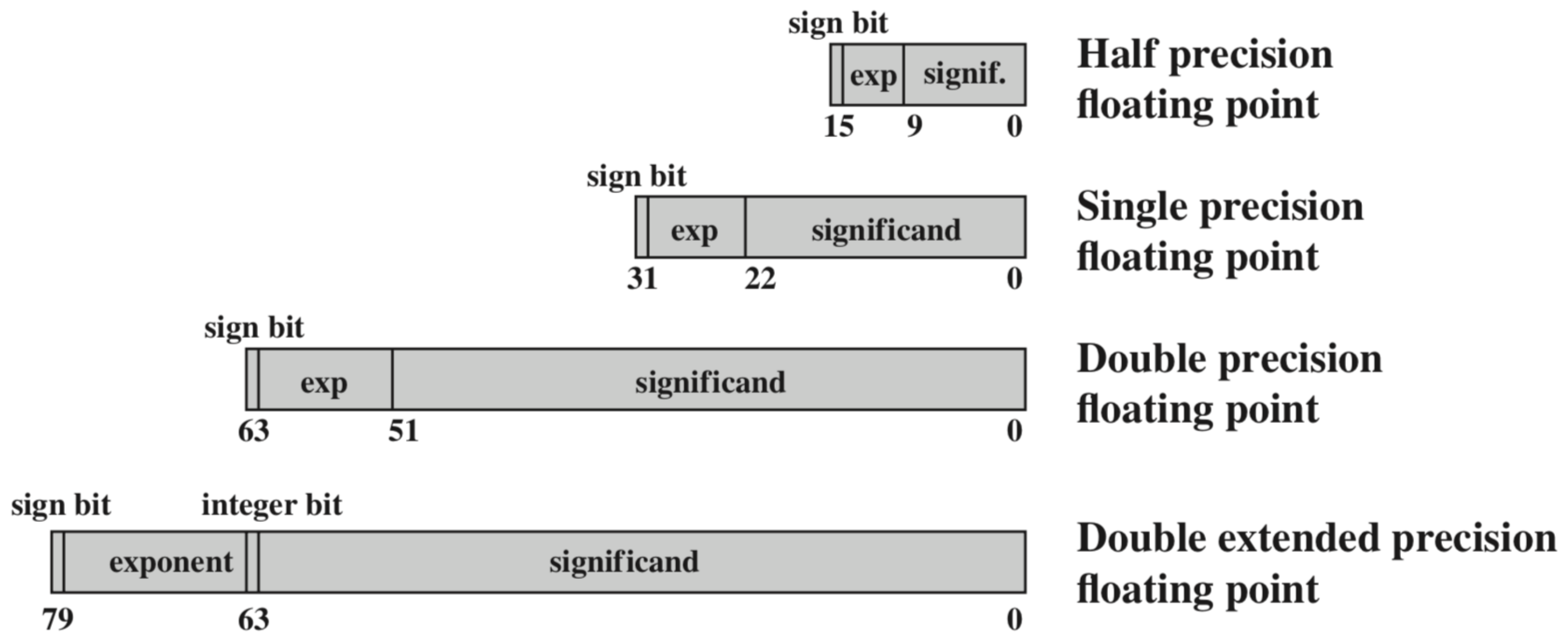
- The signed integers are in two's complement representation and may be 16, 32, or 64 bits long.



Example 3c: x86 numerical data types.

(Signed integers)

- The floating-point type actually refers to a set of types that are used by the floating-point unit and operated on by floating-point instructions → IEEE 754 standard.


sign bit
exp
signif.
15
9
0
Half precision floating point
sign bit
exp
significand
31
22
0
Single precision floating point
sign bit
exp
significand
63
51
0
Double precision floating point
sign bit
integer bit
exponent
significand
79
63
0
Double extended precision floating point

Characters

- A common form of data is text or character strings.
- Today, the most commonly used character → *United States as the American Standard Code for Information Interchange* (ASCII).
- Another code used to encode characters is the *Extended Binary Coded Decimal Interchange Code* (EBCDIC); used on IBM mainframes.

Logical Data

- Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data → *n*-bit unit.
- When data are viewed this way, they are considered to be logical data.
- Two advantages to the bit-oriented view:

  - ❑ Memory used efficiently to store an array of Boolean or binary data items for either values 1 (*true*) and 0 (*false*).
  - ❑ Easy to manipulate the bits of a data item.

Module 4

Instruction Set Architecture (ISA)

- ❑ Overview
- ❑ Immediate Addressing
- ❑ Direct Addressing
- ❑ Indirect Addressing
- ❑ Register Addressing
- ❑ Register Indirect Addressing
- ❑ Displacement Addressing

Overview

- Two issues arise in specifying the operands addresses and operations of instructions:

How the address of an operand is specified?

How the bits of an instruction are organized?

- A variety of addressing techniques has been employed to be able to reference a large range of locations in main memory or, for some systems, virtual memory.

Let’s recall what we have learned so far about ISA

Instruction Representation

- Each instruction is represented by a sequence of bits that divided into fields, corresponding to the constituent elements of the instruction.

- **Example** of simple instruction format:

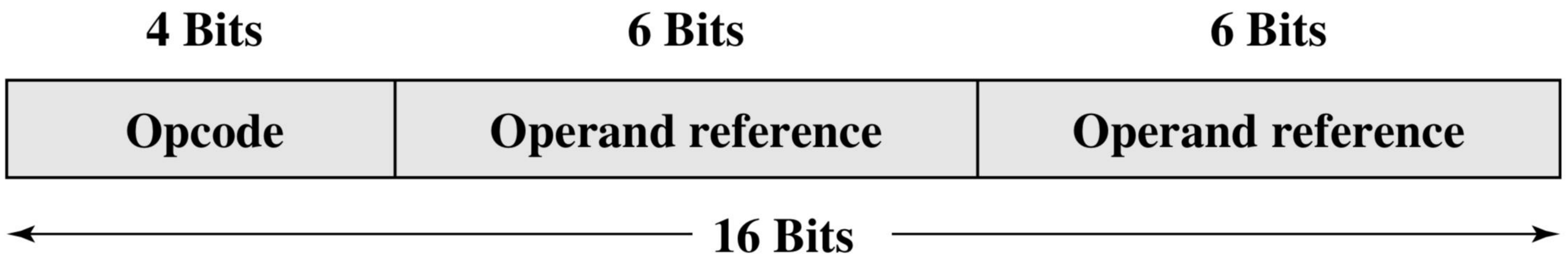
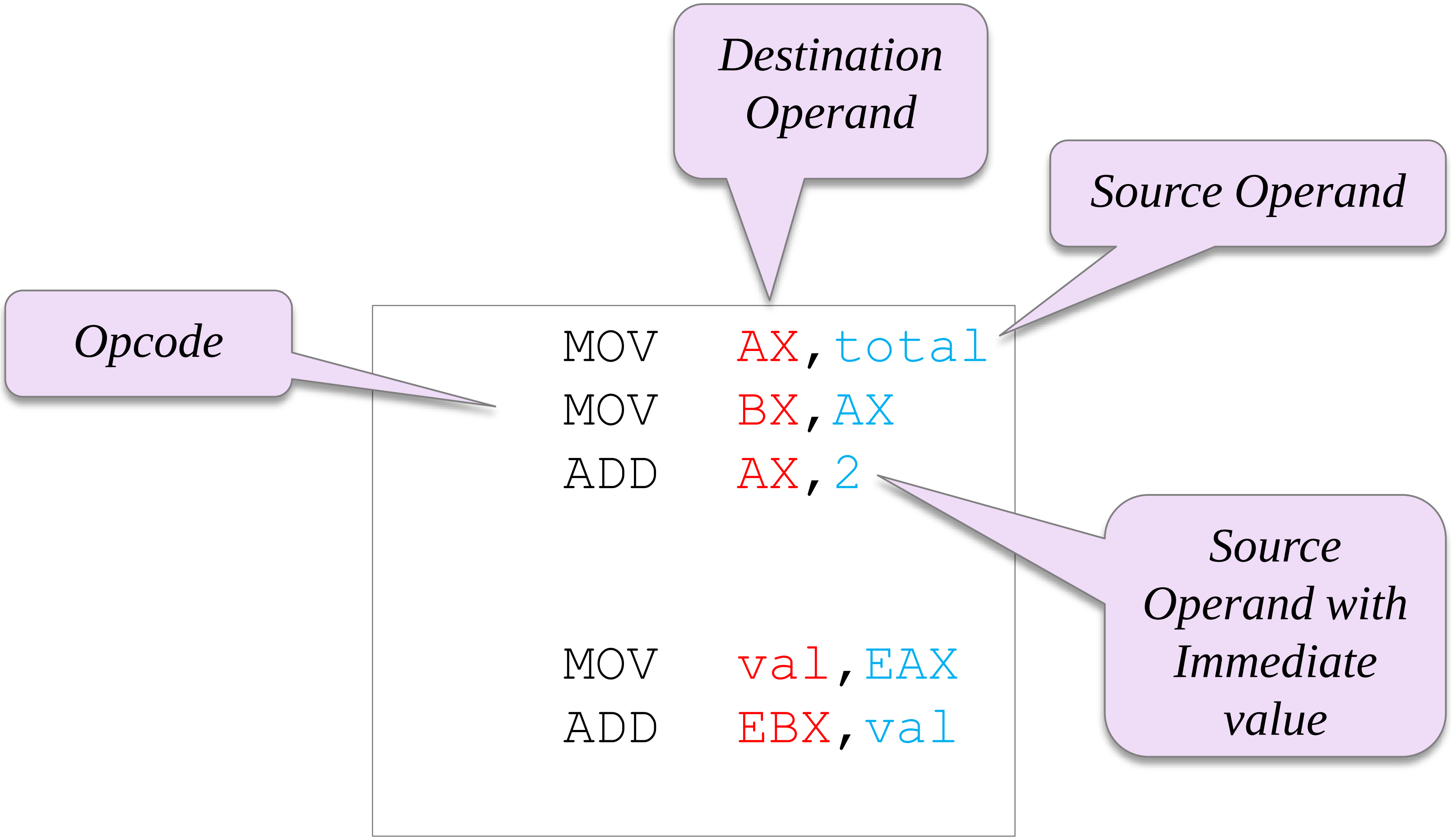


Table: The elements of a machine instruction.

Elements	Description
Operation code *(Opcode)*	○ Specifies the operation to be performed a binary code (e.g., `MOV`, `ADD`, `SUB`).
Source operand reference	○ The operation may involve one or more source operands, that is, operands that are inputs for the operation.
Result operand reference *(Destination).*	○ The operation may produce a result.
Next instruction reference	○ This tells the processor where to fetch the next instruction after the execution of this instruction is completed.

Invisible

Example 1: Portion of an assembly language.



```
MOV    AX,total
MOV    BX,AX
ADD    AX,2


MOV    val,EAX
ADD    EBX,val
```

- *Source* and result operands (*Destination*) can be in one of four areas:

Section 4.4: Addressing Mode

- Main or virtual memory : Memory address for both must be supplied.
- Processor (CPU) registers : One or more registers that can be referenced by instructions.
- Immediate : The value of the operand is contained in the field in the instruction executed.
- I/O device – Instruction specifies the I/O module and device for the operation

Example 2:

Current Ins. : 0000
Next Ins. : 0001

```
0000  MOV   AX,TOTAL
0001  MOV   BX,AX
0002  ADD   AX,2
0003  TARGET
0004  CALL  READINT
0005  MOV   VAL,EAX
0006  ADD   EBX,VAL
0007  JMP   TARGET
```

Operand: Memory

Operand: Register

Operand: Immediate value

Operand: From I/O

Next instruction is where TARGET is located = 0003

Current Ins. : 0007
Next Ins. : 0003

Operand contain immediate value

Notations:

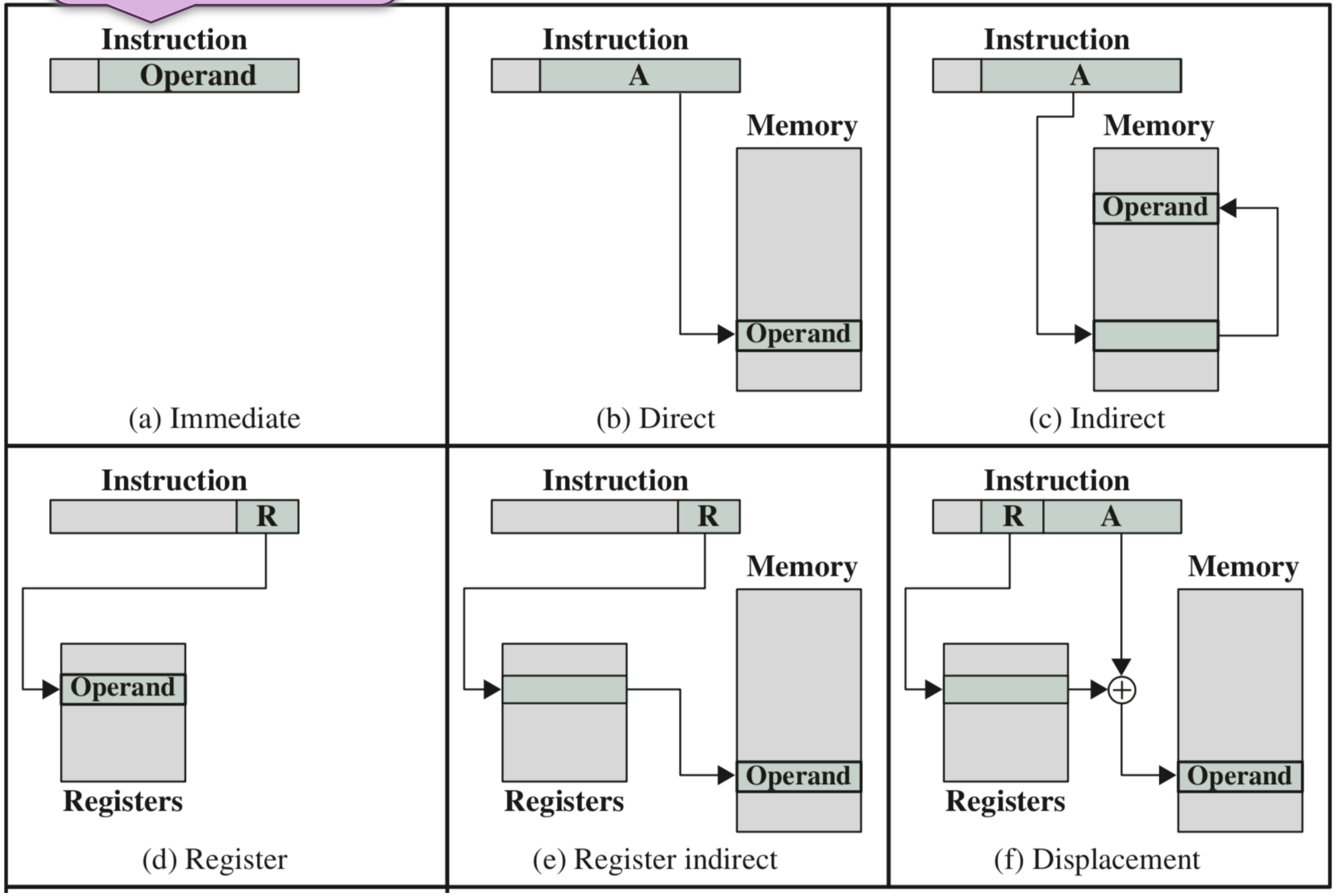


Figure: Addressing modes.

A = contents of an address field in instruction (memory address)

R = contents of an address field in instruction that refers to a register.

EA = actual (effective) address of the location containing the referenced operand.

(X) = contents of memory location X or register X.

A little bit about registers (further details in next module)

- Processor operations mostly involve processing data. This data can be stored in memory and accessed from thereon. However, reading data from and storing data into memory slows down the processor, as it involves complicated processes of sending the data request across the control bus and into the memory storage unit and getting the data through the same channel.

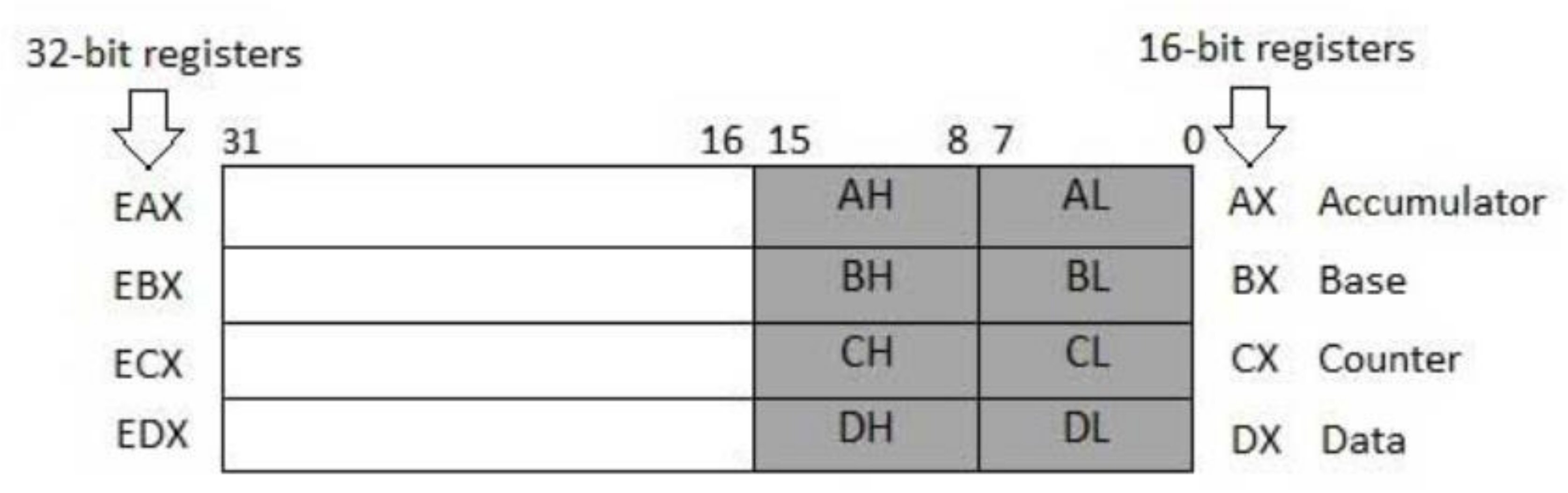
- To speed up the processor operations, the processor includes some internal memory storage locations, called **registers**

Registers

- General registers
  - Data registers; EAX, EBX, ECX, EDX
  - Pointer registers; EIP, ESP, EBP
  - Index registers; ESI, EDI
- Control registers (flag); OF, DF, IF, TF, SF, ZF, AF, PF, CF
- Segment registers (specific areas defined in a program for containing data, code and stack)
  - Code segment
  - Data segment
  - Stack segment

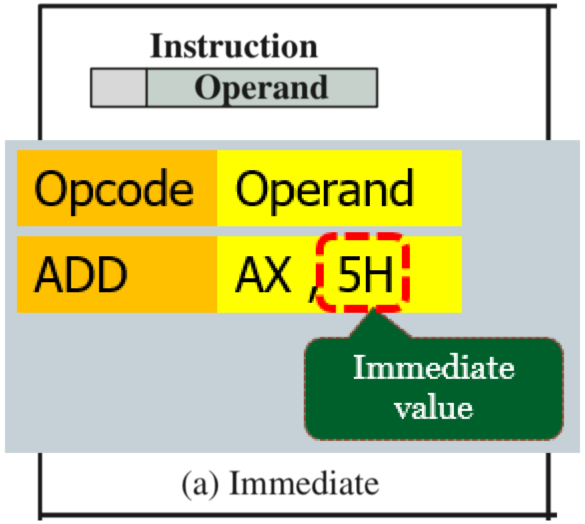
Let's recall

- Data registers; EAX, EBX, ECX, EDX



4

(a) Immediate Addressing



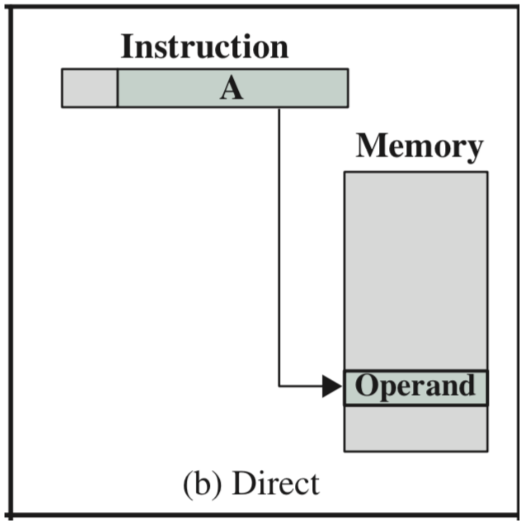
$$\text{Operand} = A$$

- The simplest form of addressing, in which the operand value is present in the instruction.
- Can defines and uses constant or set initial values of variables.

Saving one memory or cache → No memory reference other than the instruction fetch is required, (faster).

The size of the number is restricted to the size of the address field.

(b) Direct Addressing



$$EA = A$$

e.g. add EAX, A

- Look in memory at address A for operand
- Add contents of cell A to register EAX

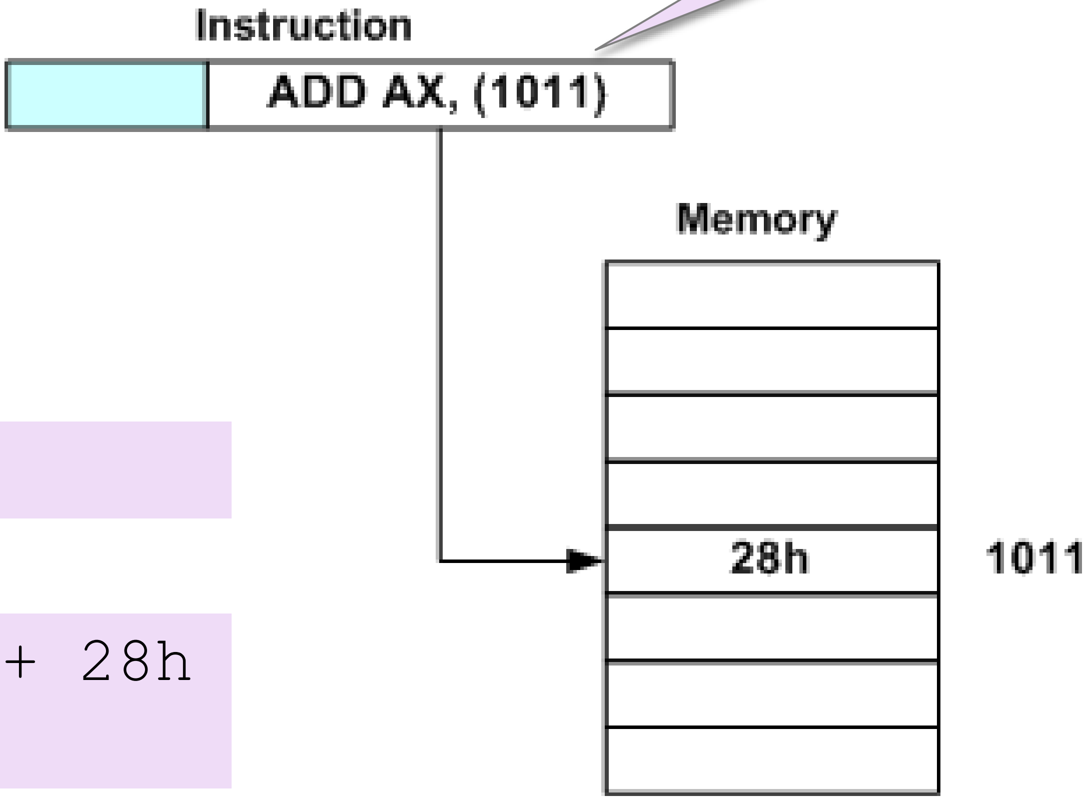
- The address field contains the *Effective Address* (EA) of the operand.
- EA=A=address field of (A)
- The address field refers to a main memory address

Requires only one memory reference and no special calculation.

It provides only a limited address space.

Example 4: *Direct addressing*.

Address of operand = 1011



```
AX = 10h
```

```
AX = 10h + 28h
   = 38h
```

Activity: *Visual Studio.*

- Retype the next following coding in Example 5 - 12
- Then proceed with these 2 tasks:
  - (A) Execute ; [Start Without Debugging]
  - (B) Debug ; F10

Example 5: *Direct addressing.*

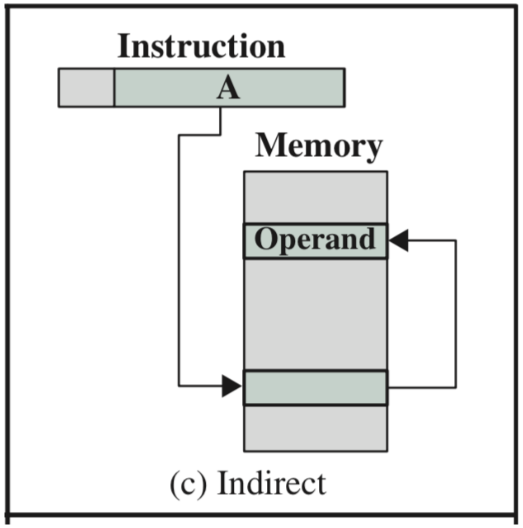
```
.data

val1   byte  10h
array1 word  2210h, 11h
array2 dword 123h, 234h

.code

main PROC
      mov  al,val1
      mov  bx,array1
      mov  ecx,array2
      call dumpregs
      exit
main ENDP
```

```
al  =       10h
bx  =     2210h
ecx = 00000123h
```

(*DumpRegs*)

```
EAX=7611ED10  EBX=7FFD2210  ECX=00000123  EDX=00401022
ESI=00000000  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=0040102D  EFL=00000246  CF=0  SF=0  ZF=1  OF=0
```

(c) Indirect Addressing



$$EA = (A)$$

Look in A, find address (A), and look there for operand.

Memory cell pointed to by address field contains the address of the operand @ (pointer to operand)

- Solution for the limitation of the address range in *direct addressing* → to have the address field address of a word in memory, full-length address of the operand.

No particular advantage.

Three or more memory references could be required to fetch an operand; slower.

the register in brackets must be incremented by a value that matches the array type → 1 for byte, 2 - word, 4 - dword.

Example 6: *Indirect addressing.*

```
.data

array1 byte  10h
array2 word  123h,234h
array3 dword 123456h

.code

main PROC
      mov  bl,[array1]
      mov  cx,[array2]
      mov  edx,[array3]
      mov  ax,[array2+2]
      call dumpregs
      exit
main ENDP
```

Move the content of the memory where the first byte of array1 is kept into BL

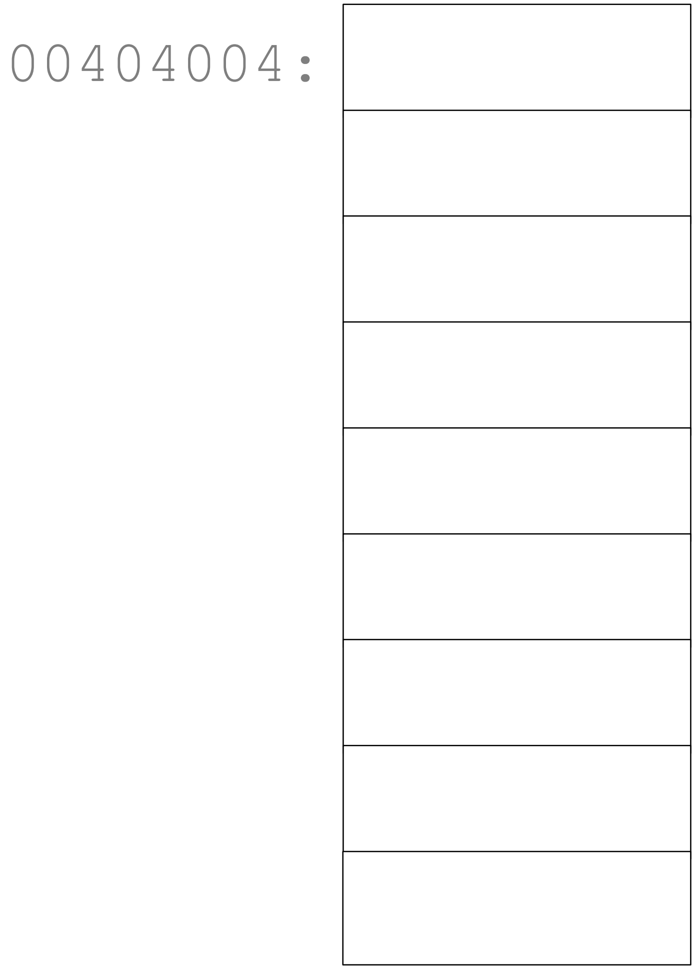
```
bl  =       10h
cx  =     0123h
edx = 00123456h
ax  =     0234h
```

(DumpRegs)

```
EAX=00000234  EBX=00000010  ECX=00000123  EDX=00123456
ESI=00404004  EDI=00404008  EBP=0012FF94  ESP=0012FF8C
EIP=0040104C  EFL=00000246  CF=0  SF=0  ZF=1  OF=0
```

Exercise 4.1:

```
.data

array1 byte   10h
array2 word   123h,234h
array3 dword 123456h
```

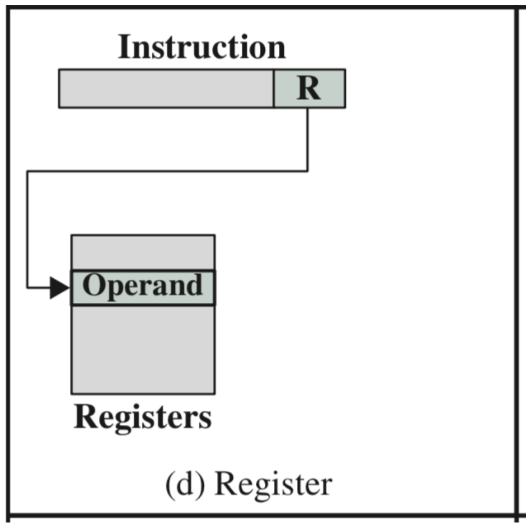


Given three arrays with multiple initializer in assembly language program. Complete the following diagram by writing:

(a) the offset for all memory. Assume the first offset is 00404004.

(b) the value of the data stored using *Little Endian Order*.

(d) Register Addressing



(d) Register

$$EA = R$$

- Similar to *direct addressing*, <u>except</u> the address field refers to a register rather than a main memory address.
- The programmer need to decide which values should remain in registers and which should be stored in main memory.

No time consuming memory references needed → faster.

It has limited number of registers → provides a limited address space.

Example 7: *Register addressing.*

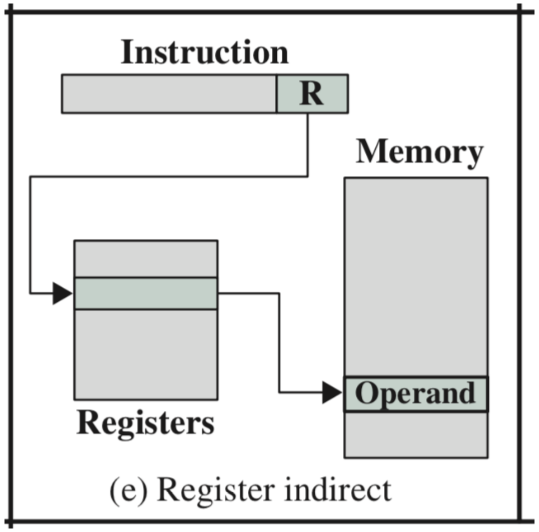
```
eax = 00000000h
ebx = 00002000h
ecx = 00003000h

eax = 00002000h
eax = 00005000h
```

```
.data

.code

main PROC
        mov   eax,0
        mov   ebx,2000h
        mov   ecx,3000h

        mov   eax,ebx
        add   eax,ecx

        call dumpregs
        exit
main ENDP
```

(*DumpRegs*)

```
EAX=00005000  EBX=00002000  ECX=00003000  EDX=00401005
ESI=00404004  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=00401028  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

(e) Register Indirect Addressing



$$\mathrm{EA} = (\mathrm{R})$$

- Similar to *indirect addressing*.
- The <u>advantages</u> and <u>limitations</u> of *register indirect addressing* are basically the same as for *indirect addressing*.

Register indirect addressing uses one less memory reference than *indirect addressing*.

Example 8:

Register indirect addressing.

```
.data
array1 byte  10h,11h,12h,13h
array2 word  123h,234h

.code

main PROC
       mov   esi,OFFSET array1
       mov   edi,OFFSET array2
       mov   bl,[esi]
       mov   cx,[edi]
       mov   edx,(esi)
       mov   al,[esi+1]
       call  dumpregs
       exit
main ENDP
```

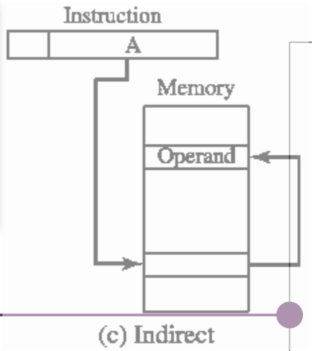
Address of the data that stored in register.

esi	=	00404004h
edi	=	00404008h
bl	=	10h
cx	=	0123h
edx	=	00404004h
al	=	11h

(*DumpRegs*)

```
EAX=00000011   EBX=00000010   ECX=00000123   EDX=00404004
ESI=00404004   EDI=00404008   EBP=0012FF94   ESP=0012FF8C
EIP=0040103D   EFL=00000246   CF=0   SF=0   ZF=1   OF=0
```

Read esi, go to memory address, get the value, store in BL



Example 8:

Register indirect addressing.

```
.data
array1 byte  10h,11h,12h,13h
array2 word  123h,234h

.code

main PROC
     mov  esi,OFFSET array1
     mov  edi,OFFSET array2
     mov  bl,[esi]
     mov  cx,[edi]
     mov  edx,(esi)
     mov  al,[esi+1]
     call dumpregs
     exit
main ENDP
```

Address that holds the first byte of array1 is stored into esi

Move contents of cell pointed by esi to register al

```
esi = 00404004h
edi = 00404008h
bl  =       10h
cx  =     0123h
edx = 00404004h
al  =       11h
```

(*DumpRegs*)

```
EAX=00000011  EBX=00000010  ECX=00000123  EDX=00404004
ESI=00404004  EDI=00404008  EBP=0012FF94  ESP=0012FF8C
EIP=0040103D  EFL=00000246  CF=0  SF=0  ZF=1  OF=0
```

Example 8:

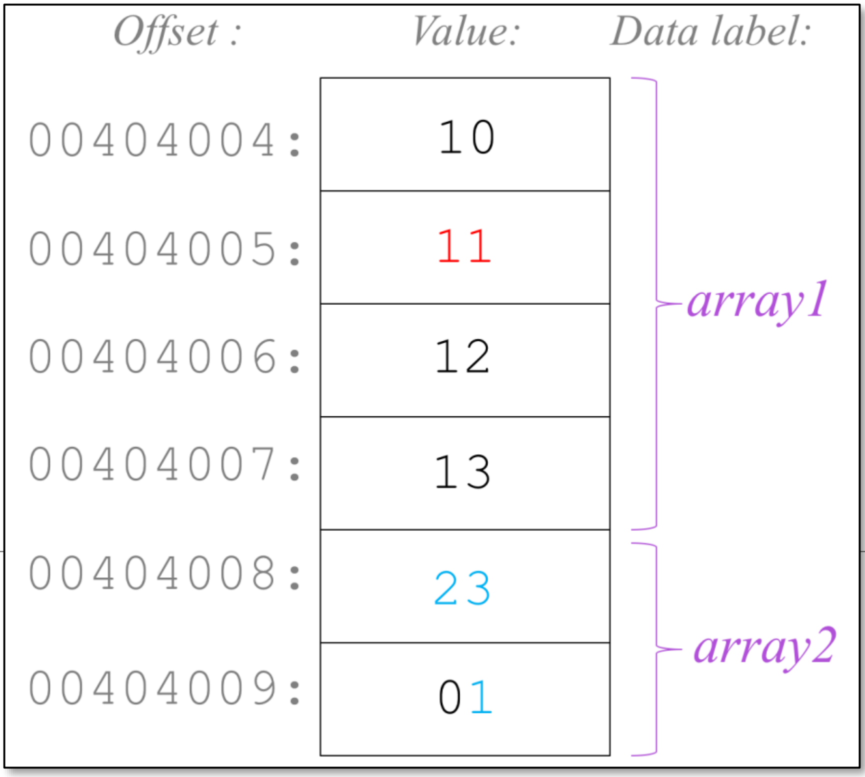
Register indirect addressing.

```
array1 byte  10h,11h,12h,13h
array2 word  123h,234h
```

```
esi = 00404004h
edi = 00404008h
```

```
mov  al,[esi+1]
```

```
al  =      11h
```



Example 9:

Register indirect addressing.

```
.data

.code

main PROC
      mov   ebx,404000h
      mov   dl,[ebx]
      inc   ebx
      mov   cl,[ebx]

      call  dumpregs
      exit
main ENDP
```

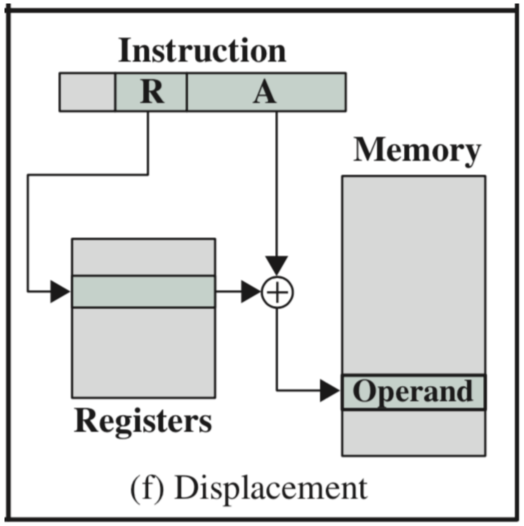
```
ebx  =  00404000h
dl   =        24h
ebx  =  00404001h
cl   =        55h
```

Offset :	*Value:*
00404000:	24
00404001:	55
00404002:	88

(*DumpRegs*)

```
EAX=00005000  EBX=00404001  ECX=00000055  EDX=00000024
ESI=00404000  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=0040103D  EFL=00000202  CF=0  SF=0  ZF=0  OF=0
```

(f) Displacement Addressing



$$EA = A + (R)$$

3 common displacement addressing techniques:

- Relative addressing
- Base-Register addressing
- Indexed addressing

- A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing.
- Address field hold two values:
  - A = base value
  - R = register that holds displacement

  (or vice versa)

(f.1) Relative Addressing

- Also called PC-relative addressing, the implicitly referenced register is the *Program Counter* (PC).

$$R = PC$$

$$EA = A + (PC)$$

- The next instruction address (shown in PC) is added to the address field to produce the EA.

$$\begin{aligned} Destination &= Target - Source \\ &= L1 - PC \\ &= 28h - 39h \\ &= -11h = EF \end{aligned}$$

Example 10:

Displacement Addressing.

(1) Relative addressing

–(ve) because its jumping backwards

```
00000014  B9 00000000      mov ecx,0
00000019  BA 00000000      mov edx,0
0000001E  B8 00000000      mov eax,0
00000023  B9 00000004      mov eax,4

00000028                L1:
00000028  66| 8B 98        mov bx,array2[eax]
          00000008 R
0000002F  83 C0 02         add eax,2
00000032  E8 00000000 E    call dumpregs
00000037  E2 EF            LOOP L1

                           exit
00000039  6A 00       *    push   +000000000h
```

PC = 39

Need to go to L1 which is in 00000028

PC = 39

(f.2) Base-Register Addressing

$$\boxed{EA = A + R}$$

- A holds displacement.
- R holds pointer to base address.
- R may be explicit or implicit.

Example 11:

Displacement Addressing.

(2) Base-register addressing

```
.data
count  dword 10h
array1 byte 10h,11h,12h,13h
array2 word 123h,234h,345h,456h
array3 dword 123456h,

.code
main PROC
     mov esi,offset array1
     mov ebx,10h
     mov ax,word ptr[ebx+esi]
     add ebx,esi
     mov cx,word ptr[ebx]

     call dumpregs
     exit
main ENDP
```

Base-index addressing

Base addressing

Because AX and CX is a word size, then 2 bytes will be put in.

Displacement Addressing.

(b) Base-register addressing

```
.data
count   dword 10h
array1  byte  10h,11h,12h,13h
array2  word  123h,234h,345h,456h
array3  dword 123456h, 23456789h

.code
main PROC
        mov esi,offset array1
        mov ebx,10h
        mov ax,word ptr[ebx+esi]
        add ebx,esi
        mov cx,word ptr[ebx]

        call dumpregs
        exit
main ENDP
```

Load a word size value into ax

```
esi = 00404004h
ebx = 00000010h
 ax =     6789h
ebx = 00404014h
 cx =     6789h
```

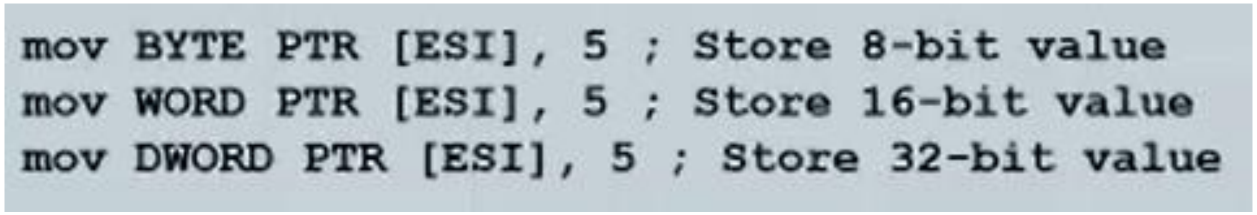
Offset :	*Value:* array3
00404014:	89
00404015:	67
00404016:	45
00404017:	23

(DumpRegs)

```
EAX=753E6789  EBX=00404014  ECX=00006789  EDX=00401005
ESI=00404004  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=00401028  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

Directives BYTE PTR, WORD PTR, DWORD PTR

- To get around this instance, we must use a pointer directive, such as;



```
mov BYTE PTR [ESI], 5 ; Store 8-bit value
mov WORD PTR [ESI], 5 ; Store 16-bit value
mov DWORD PTR [ESI], 5 ; Store 32-bit value
```

- In general, PTR operator forces expression to be treated as a pointer of specified type

- E.g

```
.data
num DWORD 0
.code
mov ax, WORD PTR (num)      ;load a word-size value from a DWORD
```

(f.3) Indexed Addressing

$$EA = A + R$$

- A = base
- R = displacement
- Good for accessing arrays

Exercise 4.1:

```
.data
array1 byte 10h,11h,12h,13h
array2 word 123h,234h,345h,456h
array3 dword 123456h,23456789h
```

Given three arrays with multiple initializer in assembly language program. Complete the following diagram by writing:

(a) the offset for all memory. Assume the first offset is 00404004.

(b) the value of the data stored using *Little Endian Order*.

Offset :	*Value:*
`00404004:`	

Offset :	*Value:*

Example 12:

Displacement Addressing.

(3) Indexed addressing

```
.data
array1 byte 10h,11h,12h,13h
array2 word 123h,234h,345h,456h
array3 dword 123456h, 23456789h

.code
main PROC
   mov eax,0
   mov ecx,4

L1:
   mov bx, array2[eax]
   add eax,2
   call dumpregs
   LOOP L1
   exit
main ENDP
```

Base

Displacement

```
mov bx,[array2+eax]
Add eax,2
```

CX is known as the count register, as the ECX, CX registers store the loop count in iterative operations

E

Displacement Addressing.

(3) Indexed addressing

```
eax  =  00000000h
ecx  =  00000004h

 bx  =       0123h
eax  =  00000002h

ecx  =  00000003h
```

```
dec  ecx
```

```
.data
array1 byte 10h,11h,12h,13h
array2 word 123h,234h,345h,456h
array3 dword 123456h, 23456789h

.code
main PROC
   mov eax,0
   mov ecx,4

L1:

   mov bx, array2[eax]
   add eax,2

   call dumpregs
   LOOP L1
   exit
main ENDP
```

```
mov bx,[array2+eax]

[00404008+00000000]
→ array2[00404008]
```

Example 12:

Displacement Addressing.

(3) Indexed addressing

2nd looping

```
L1:
    mov bx, array2[eax]
    add eax,2

    call dumpregs
    LOOP L1
    exit
main ENDP
```

```
 bx  =     0234h
eax  = 00000004h

ecx  = 00000002h
```

```
dec ecx
```

```
mov bx,[array2+eax]

[00404008+00000002]
→ array2[0040400A]
```

Example 12:

Displacement Addressing.

(3) Indexed addressing

3rd looping

```
L1:
    mov bx, array2[eax]
    add eax,2

    call dumpregs
    LOOP L1
    exit
main ENDP
```

```
 bx =      0345h
eax = 00000006h

ecx = 00000001h
```

```
dec ecx
```

```
mov bx,[array2+eax]

[00404008+00000004]
→ array2[0040400C]
```

Example 12:

Displacement Addressing.

(3) Indexed addressing

4th looping

Exit looping

```
L1:
    mov bx, array2[eax]
    add eax,2

    call dumpregs
    LOOP L1
    exit
main ENDP
```

```
mov bx,[array2+eax]
```

```
[00404008+00000006]
→ array2[0040400E]
```

```
 bx =      0456h
eax = 00000008h

ecx = 00000000h
```

```
dec ecx
```

The `DumpRegs` for the program:

```
EAX=00000002  EBX=00000123  ECX=00000004  EDX=00000000
ESI=00404004  EDI=00404008  EBP=0012FF94  ESP=0012FF8C
EIP=00401047  EFL=00000202  CF=0  SF=0  ZF=0  OF=0
```

```
EAX=00000004  EBX=00000234  ECX=00000003  EDX=00000000
ESI=00404004  EDI=00404008  EBP=0012FF94  ESP=0012FF8C
EIP=00401047  EFL=00000202  CF=0  SF=0  ZF=0  OF=0
```

```
EAX=00000006  EBX=00000345  ECX=00000002  EDX=00000000
ESI=00404004  EDI=00404008  EBP=0012FF94  ESP=0012FF8C
EIP=00401047  EFL=00000202  CF=0  SF=0  ZF=0  OF=0
```

```
EAX=00000008  EBX=00000456  ECX=00000001  EDX=00000000
ESI=00404004  EDI=00404008  EBP=0012FF94  ESP=0012FF8C
EIP=00401047  EFL=00000202  CF=0  SF=0  ZF=0  OF=0
```

Table: Basic addressing modes.

Mode	Algorithm	Principal Advantage 👍	Principal Disadvantage 👎
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Module 4
Instruction Set Architecture (ISA)

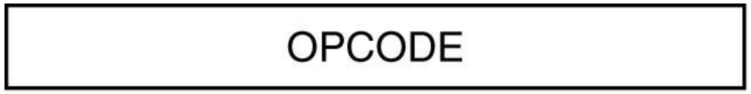
- Overview
- Instruction Length
- Allocation Bits
- Variable-Length Instructions

Overview

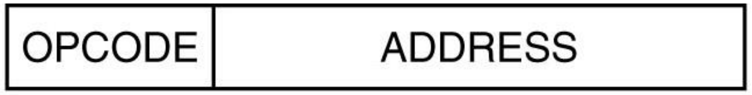
- An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields.
- An instruction format must include an opcode and, implicitly or explicitly, zero or more operands.
- The format must, implicitly or explicitly, indicate the addressing mode for each operand.
- For most instruction sets, more than one instruction format is used.

- Four common instruction formats:

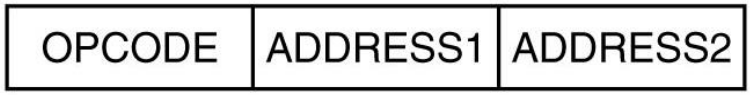
(a) Zero-address instruction.



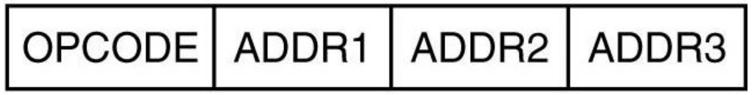
(b) One-address instruction



(c) Two-address instruction.



(d) Three-address instruction.



Instruction Length

- This decision affects, and is affected by:
  - memory size,
  - memory organization,
  - bus structure,
  - processor complexity, and
  - processor speed.
- Trade off between powerful instruction repertoire and saving space.
- Other issues: Instruction length equal or multiple to memory transfer length (bus system)?

Allocation of Bits

- We've looked at some of the factors that go into deciding the length of the instruction format.

- An equally difficult issue is how to allocate the bits in that format, and the trade-offs here are complex.

- The following interrelated factors go into determining the use of the addressing bits:
  - Number of addressing modes – if indicated implicitly, certain opcodes might always call for indexing; if explicit – one or more mode bits will be needed.
  - Number of operands – typical instruction has 2 operands – uses mode indicator for operand addresses.
  - Register versus memory – single user register (accumulator), one operand address is implicit and consume no instruction bits; for multiple registers – a few bits are needed to specify the registers.

- Number of register sets – have one set of general purpose registers with 32 or more registers in the set – for example for sets of <u>8 registers only 3 bits </u>are needed to identify the registers, <u>opcode will implicitly determine </u>which register set is being referenced.
- Address range – the range of addresses that can be referenced related to the number of bits.
- Address granularity – an address can reference a word or byte a the designer’s choice – byte addressing is convenient for character manipulation.

Variable-Length Instructions

- Designer may choose instead to provide a variety of instruction formats of different lengths.
- This tactic makes it easy to <u>provide a large repertoire</u> of opcodes, with different opcode lengths.
- Addressing can be more <u>flexible</u>, with various combinations of register and memory references plus addressing modes.
- With variable-length instructions, these many variations can be provided efficiently and compactly

4.6 Summary

- This chapter discussed on *what* an instruction set does by examining the types of operands and operations that may be specified by machine instructions.
- Described the various types of addressing modes common in instruction sets.
- Present an overview of essential characteristics of machine instructions.
- Summarize the issues and trade-offs involved in designing an instruction format.

Review Questions

4.1 What are the typical elements of a machine instruction?

4.2 What types of locations can hold source and destination operands?

4.3 List and briefly explain five important instruction set design issues.

4.4 What types of operands are typical in machine instruction sets?

Review Questions

4.5 Briefly define:

a) immediate addressing.
b) direct addressing.
c) indirect addressing.
d) register addressing.
e) register indirect addressing.
f) displacement addressing.
g) relative addressing.

4.6 What are the advantages of using a variable-length instruction format?