SECR2033
Computer Organization and Architecture

Lecture slides prepared by "Computer Organization and Architecture", 9/e, by William Stallings, 2013.

1

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)

4.1 Introduction

- Overview
- Hierarchy of Computer Languages
- General Concepts: x86 Processor Architecture
- Design Decisions for Instruction Sets
- ISA Level

4

Overview

ALU (Arithmetic Logic Unit)

- One boundary where the computer designer and the computer programmer can view the same machine is the machine instruction set.
- Implementing the _____ is a task that in large part involves implementing the machine instruction set.
- The user who chooses to program in machine language (actually, in assembly language) becomes aware of the register and memory structure, the types of data directly supported by the machine, and the functioning of the ALU.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.248. 5

4

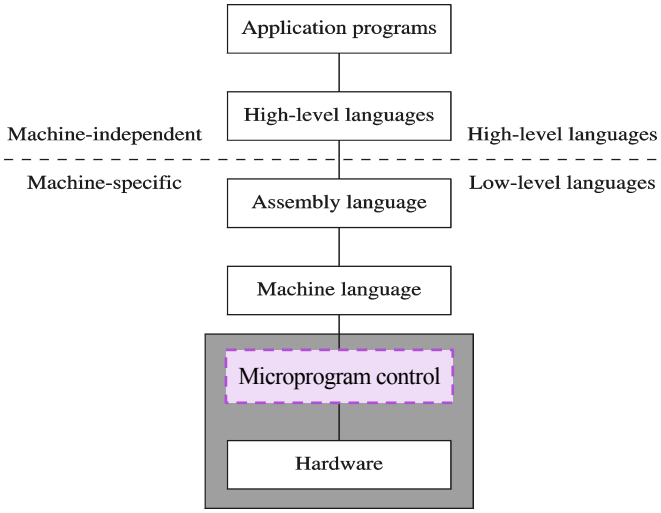
Some important questions to ask:

- *What is assembly language?*
- *Why learn assembly language?*
- *What is machine language?*
- *How is assembly related to machine language?*
- *What is an assembler?*
- *How is assembly related to high-level language?*
- *Is assembly language portable?*

6

4

Hierarchy of Computer Languages



7

4

Assembly and Machine Languages

Machine language:

- Native to a processor: executed directly by hardware.
- Instructions consist of binary code: 1s and 0s

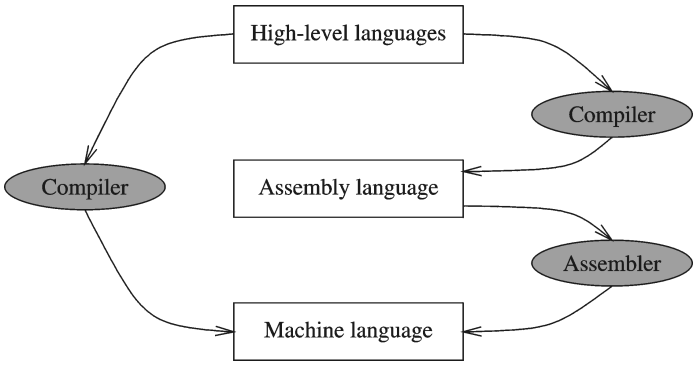
Assembly language:

- A programming language that uses *symbolic names* to represent operations, registers and memory locations.
- Slightly higher-level language.
- Readability of instructions is better than machine language.
- One-to-one correspondence with machine language instructions.

8

4

Compiler and Assembler



translate high-level programs to machine code:
→ either directly, or Indirectly via an *assembler*.

translate assembly to machine code.

9

4

High-Level Languages: Advantages

Program development is faster: fewer instructions to code.

Program maintenance is easier: same reasons as above.

However, *Assembly language* programs are *not portable*.

Programs are portable:

- ❑ Contain few machine-dependent details → Can be used with little or no modifications on different machines.
- ❑ Compiler translates to the target machine language.

10

Why Assembly Languages?

4

- Accessibility to system hardware:
  - ❑ Assembly language is useful for implementing system software.
  - ❑ Also useful for small embedded system applications.
- Space and Time efficiency:
  - ❑ Understanding sources of program inefficiency.
  - ❑ Tuning program performance.
  - ❑ Writing compact code.

11

HLL (High Level Language)

4

- Writing assembly programs gives the computer designer the needed deep understanding of the instruction set and how to design one.
- To be able to write compilers for HLLs, we need to be expert with the machine language. Assembly programming provides this experience.

12

4

General Concepts: X86 Processor Architecture

- This section describes the architecture of the x86 processor family and its host computer system from a programmer's point of view.
- _____ is a great tool for learning how a computer works, and it requires you to have a working knowledge of computer hardware.
- The concepts in this section will help to understand the assembly language code that be written.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.29. 13

4

Basic Microcomputer Design

(*Section* 5.2)

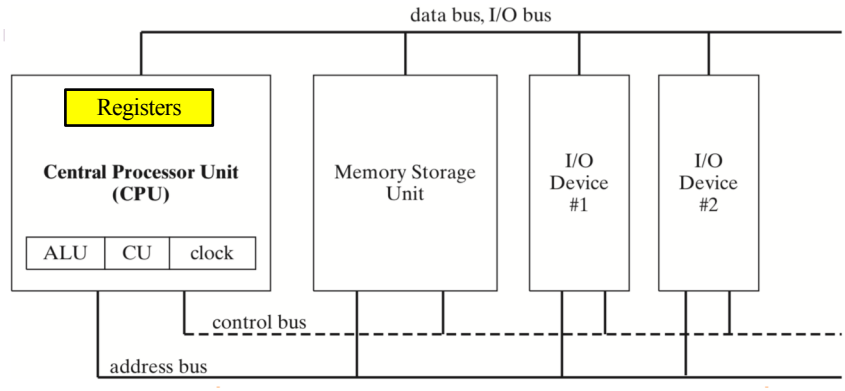


Figure: Block diagram of a microcomputer.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.30. 14

CPU (Central Processing Unit)

4

- The CPU does the *calculations* and *logic operations* → contains a limited number of storage locations as follow:

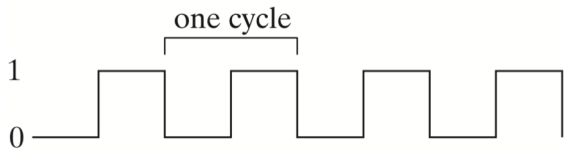
Elements	Description
Clock	○ synchronizes the internal CPU operations.
Control Unit (*CU*)	○ coordinates sequence of execution steps.
_____	○ performs arithmetic and bitwise processing.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.30. 15

4

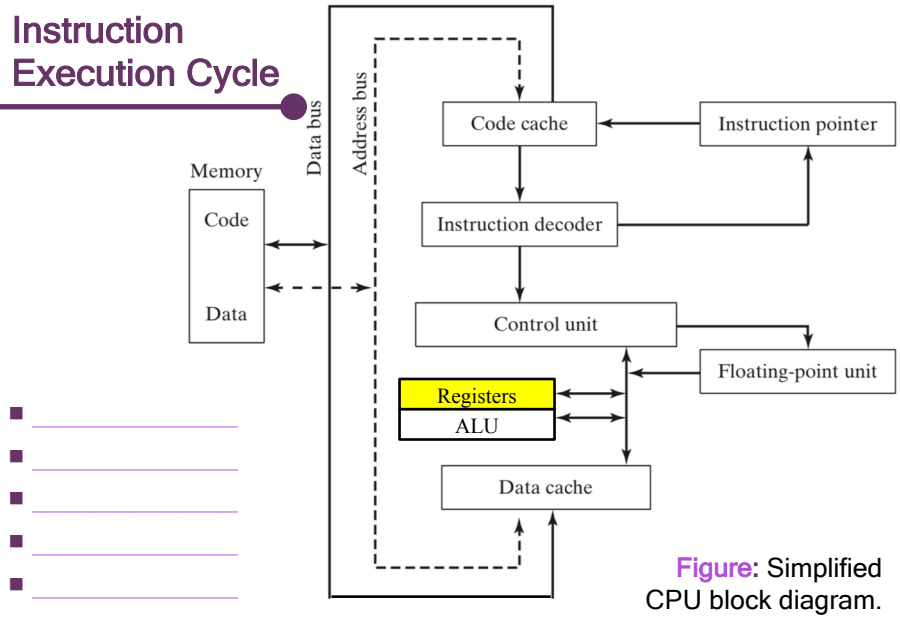
Clock

- synchronizes all CPU and BUS operations.
- machine (clock) cycle measures time of a single operation.
- clock is used to trigger events.



Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.31. 16

Instruction Execution Cycle



- _____
- _____
- _____
- _____
- _____

Figure: Simplified CPU block diagram.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.31.

4

18

4

Reading From Memory

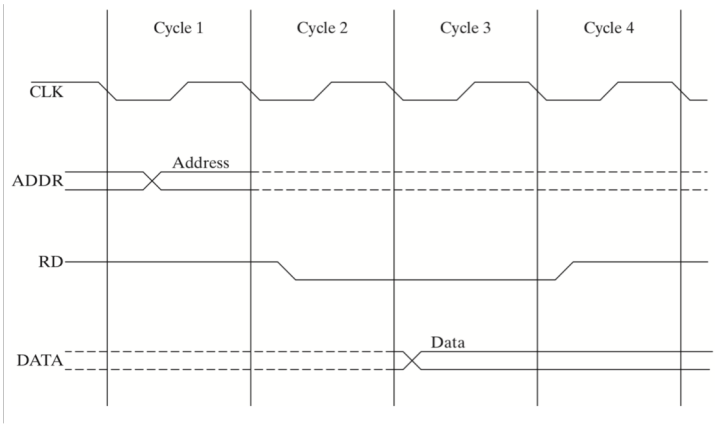


Figure: Memory read cycle.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.33. 19

4

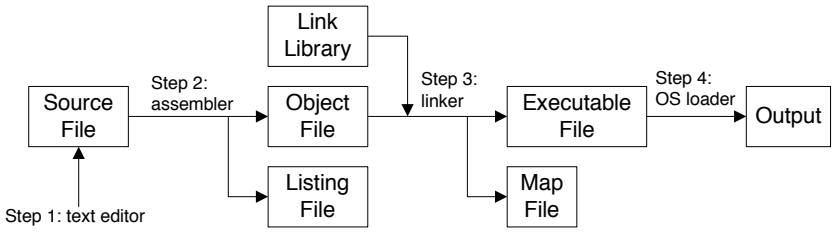
- Multiple machine cycles are required when reading from memory, because it responds much more slower than the CPU.
- The steps are:
  - **Cycle 1**: address placed on address bus (ADDR).
  - **Cycle 2**: *Read Line* (RD) set low (0).
  - **Cycle 3**: CPU waits one cycle for memory to respond.
  - **Cycle 4**: *Read Line* (RD) goes to high (1), indicating that the data is on the data bus (DATA).

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.33. 20

4

How Program Run?

Assembling, Linking, and Running Programs



- The diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.

21

4

Listing File

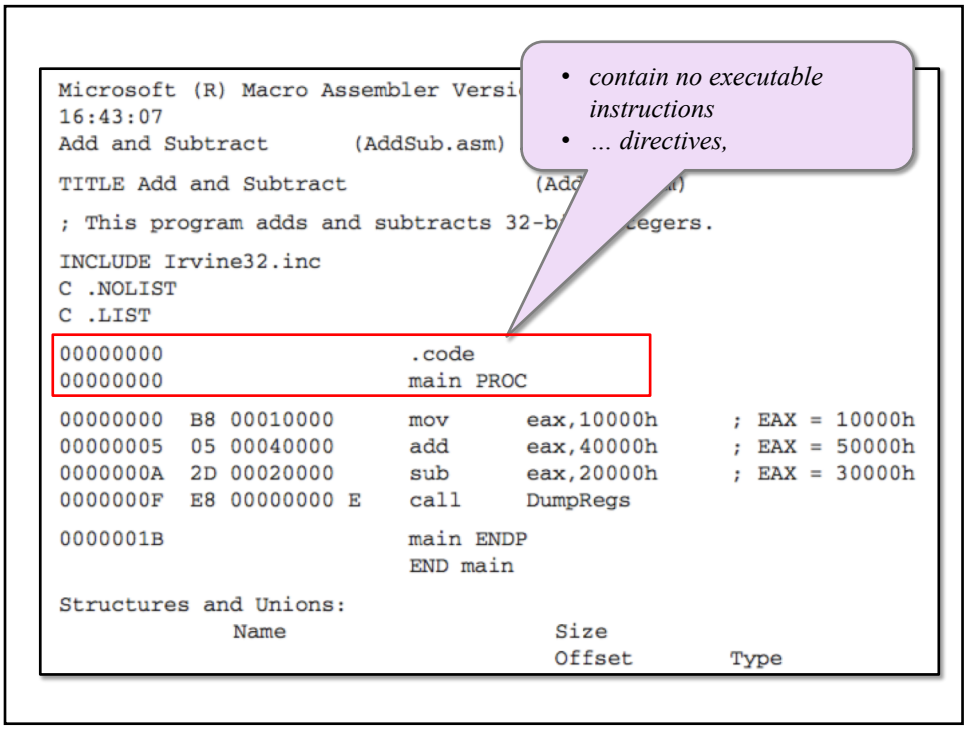
- Use it to see how your program is compiled.
- Contains:
  - *source code*
  - *addresses*
  - *object code (machine language)*
  - *segment names*
  - *symbols (variables, procedures, and constants)*
- **Example**: `addSub.lst`

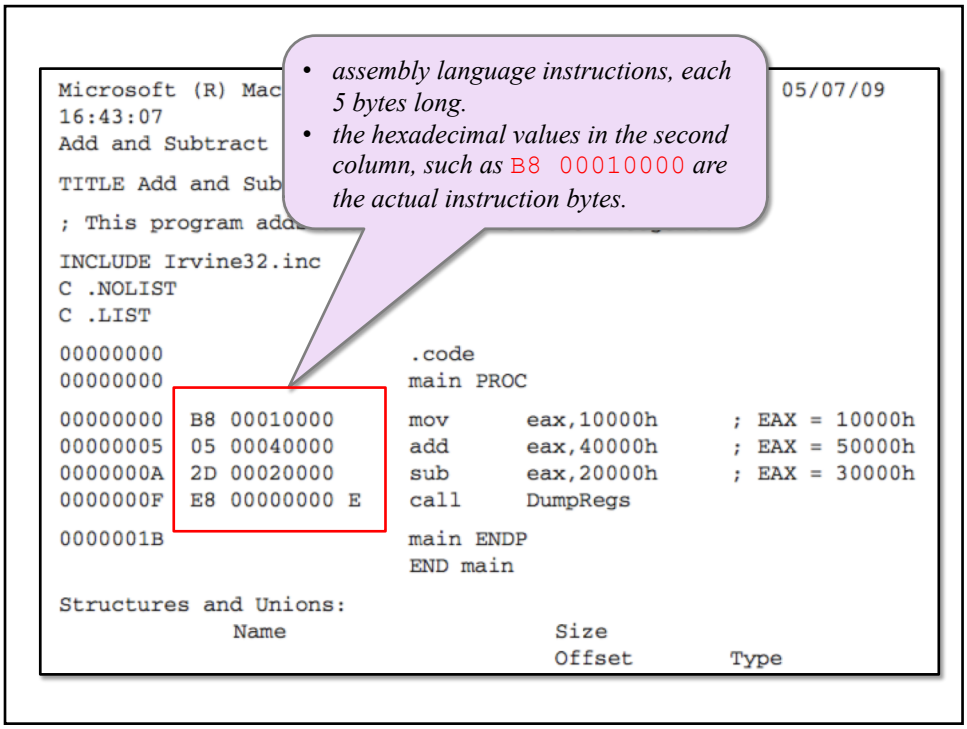
22

```
Microsoft (R) Macro Assembler Version 9.00.30729.01    05/07/09
16:43:07
Add and Subtract        (AddSub.asm)         Page 1 - 1

TITLE Add and Subtract                (AddSub.asm)

; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc
C .NOLIST
C .LIST

00000000                    .code
00000000                    main PROC

00000000  B8 00010000       mov     eax,10000h      ; EAX = 10000h
00000005  05 00040000       add     eax,40000h      ; EAX = 50000h
0000000A  2D 00020000       sub     eax,20000h      ; EAX = 30000h
0000000F  E8 00000000 E     call    DumpRegs

0000001B                    main ENDP
                            END main

Structures and Unions:
              Name                          Size
                                            Offset      Type
```

```
Microsoft (R) Macro Assembler Version 9.00.30729.01    05/07/09
16:43:07
Add and Subtract        (AddS

TITLE Add and Subtract

; This program adds and su

INCLUDE Irvine32.inc
C .NOLIST
C .LIST

00000000                    .code
00000000                    main PROC

00000000  B8 00010000       mov     eax,10000h      ; EAX = 10000h
00000005  05 00040000       add     eax,40000h      ; EAX = 50000h
0000000A  2D 00020000       sub     eax,20000h      ; EAX = 30000h
0000000F  E8 00000000 E     call    DumpRegs

0000001B                    main ENDP
                            END main

Structures and Unions:
              Name                          Size
                                            Offset      Type
```

- *32-bit addresses*
- *indicate the relative byte distance of each statement from the beginning of the program's code area*


Microsoft (R) Macro Assembler Versi
16:43:07
Add and Subtract (AddSub.asm)
TITLE Add and Subtract (Add
; This program adds and subtracts 32-b egers.
INCLUDE Irvine32.inc
C .NOLIST
C .LIST
00000000 .code
00000000 main PROC
00000000 B8 00010000 mov eax,10000h ; EAX = 10000h
00000005 05 00040000 add eax,40000h ; EAX = 50000h
0000000A 2D 00020000 sub eax,20000h ; EAX = 30000h
0000000F E8 00000000 E call DumpRegs
0000001B main ENDP
END main
Structures and Unions:
Name Size
Offset Type
• contain no executable instructions
• ... directives,


Microsoft (R) Mac 05/07/09
16:43:07
Add and Subtract
TITLE Add and Sub
; This program add
INCLUDE Irvine32.inc
C .NOLIST
C .LIST
00000000 .code
00000000 main PROC
00000000 B8 00010000 mov eax,10000h ; EAX = 10000h
00000005 05 00040000 add eax,40000h ; EAX = 50000h
0000000A 2D 00020000 sub eax,20000h ; EAX = 30000h
0000000F E8 00000000 E call DumpRegs
0000001B main ENDP
END main
Structures and Unions:
Name Size
Offset Type
• assembly language instructions, each 5 bytes long.
• the hexadecimal values in the second column, such as B8 00010000 are the actual instruction bytes.

4

Map File

- Information about each program segment:
  - *starting address*
  - *ending address*
  - *size*
  - *segment type*

- **Example**: `addSub.map`

AddSubMap.txt

```
Start  Stop   Length Name                   Class
00000H 006E2H 006E3H _TEXT                  CODE
006E4H 008FDH 0021AH _DATA                  DATA
00900H 028FFH 02000H STACK                  STACK
02900H 02AFFH 00200H _BSS                   BSS

Origin   Group
006E:0   DGROUP

Program entry point at 0000:0000
```

27

4

Design Decisions for Instruction Sets

- When a computer architecture is in the design phase, the _____ must be determined before many other decisions can be made.
- Selecting this format is often quite difficult because the instruction set must match the architecture.
- If the architecture is well designed, it could last for decades.

Linda Null and Julia Lobur (2003). *The Essentials of Computer Organization and Architecture*. United States: Jones and Bartlett Publishers. p.200.

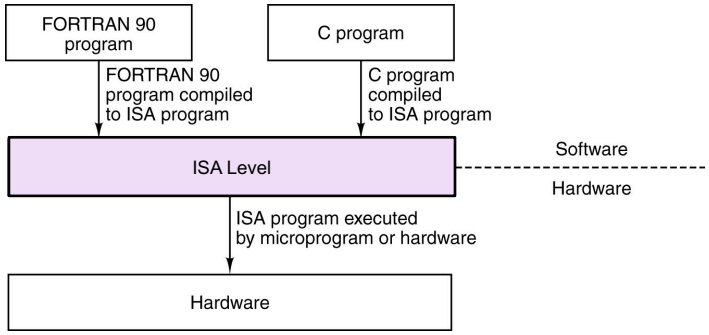
28

4

ISA Level

ISA (Instruction Set Architecture)

- ISA Level defines the _____ between the compilers (high level language) and the hardware. It is the language that both them understand.



29

4

- _____ (ISAs) are measured by several different factors, including:

(1) the amount of space a program requires;
(2) the complexity of the instruction set;
(3) the length of the instructions; and
(4) the total number of instructions.

Linda Null and Julia Lobur (2003). *The Essentials of Computer Organization and Architecture*. United States: Jones and Bartlett Publishers. p.200.

30

Module 4
Instruction Set Architecture (ISA)

Elements of a Machine Instruction

4

- The operation of the processor is determined by the instructions it executes, referred to as machine instructions or computer instructions.
- The collection of different instructions that the processor can execute is referred to as the processor's instruction set.
- Each instruction must contain the information required by the processor for execution.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.428. 32

4

Table: The elements of a machine instruction.

Elements	Description
Operation code (_____)	○ 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*

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.428. 33

4

Example 1: Portion of an assembly language.

Destination Operand

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

MOV   val,EAX
ADD   EBX,val
```

Source Operand with Immediate value

34

4

- *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

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.429. 35

4

Example 2:

Operand: Memory

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: Register

Operand: Immediate value

Operand: From I/O

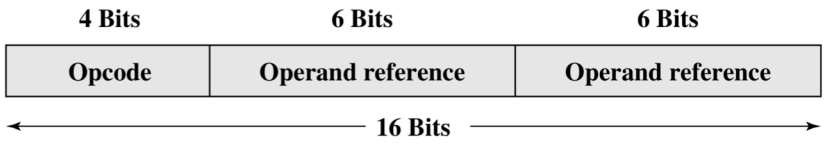
Current Ins. : 0007
Next Ins. : 0003

Next instruction is where TARGET is located = 0003

36

4 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:



William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.429-430. 37

4

It is difficult for the programmer to deal with binary representations of machine instructions.

- Thus, it has become common practice to use a symbolic representation of machine instructions.
- _____ are represented by abbreviations, called *mnemonics*, that indicate the operation.
- Common **examples**:

ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.430. 38

4

- **Example**:

Instruction type	Opcode	Symbolic representation	Description
Data transfer	00001010	LOAD MQ,M(X)	Transfer contents of memory location X to register MQ

What the processor (CPU) see.

What the programmer see.

- During instruction execution:
  - an instruction is read into an Instruction Register (IR) in the processor.
  - The processor must be able to extract the data from the various instruction fields to perform the required operation.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.430. 39

4

Instruction Types

- Consider a high-level language instruction that could be expressed in a language such as C.
- **Example**:

```
Total = Total + stuff
```

A single C instruction may require 3 machine instructions; This is typical of the relationship between a high-level language and a machine language.

- In assembly language:
  1) Load a register with the contents of memory (for `Total`).
  2) Add the contents of memory (for `stuff`) to the register.
  3) Store the content of the register to memory location (for `Total`).

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.431. 40

4

- Translating the language:

English: `Total` is assigned the sum of `Total` and `stuff`.

High-Level Language: `Total = Total + stuff`

A statement in a high-level language is translated typically into several machine-level instructions

```
mov  eax,Total
add  eax,stuff
mov  Total,eax
```

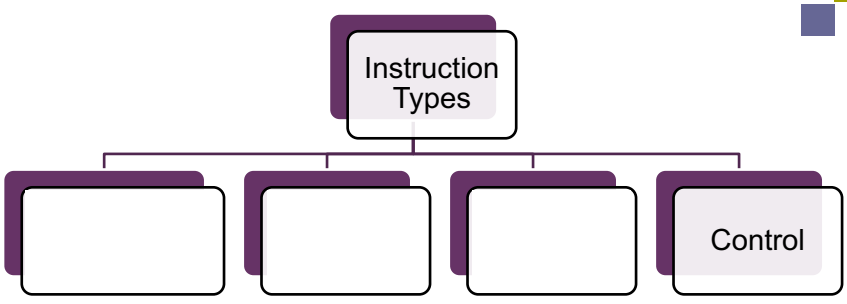
Intel Assembly Language:

```
A1 00404000
83 00404004
A3 00404008
```

Intel Machine Code:

41

4



- *Arithmetic and logic instructions.*
- *Movement of data in/out register and/or memory.*
- *I/O instructions.*
- *Test and branch instructions.*

Figure: Categories of instruction types.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.431. 42

4

Table: Examples of *data processing*.

Instruction	Description
Arithmetic	
ADD	Add operands.
SUB	Subtract operands.
MUL	Unsigned integer multiplication, with byte, word, or double word operands, and word, doubleword, or quadword result.
IDIV	Signed divide.
Logical	
AND	AND operands.
BTS	Bit test and set. Operates on a bit field operand. The instruction copies the current value of a bit to flag CF and sets the original bit to 1.
BSF	Bit scan forward. Scans a word or doubleword for a 1-bit and stores the number of the first 1-bit into a register.
SHL/SHR	Shift logical left or right.
SAL/SAR	Shift arithmetic left or right.

4

Table: Examples of *data storage* and *data movement*.

Instruction	Description
Data Movement	
MOV	Move operand, between registers or between register and memory.
PUSH	Push operand onto stack.
PUSHA	Push all registers on stack.
MOVSX	Move byte, word, dword, sign extended. Moves a byte to a word or a word to a doubleword with twos-complement sign extension.
LEA	Load effective address. Loads the offset of the source operand, rather than its value to the destination operand.
XLAT	Table lookup translation. Replaces a byte in AL with a byte from a user-coded translation table. When XLAT is executed, AL should have an unsigned index to the table. XLAT changes the contents of AL from the table index to the table entry.
IN, OUT	Input, output operand from I/O space.

4

Table: Examples of *control* (test and branch).

Instruction	Description
Control Transfer	
JMP	Unconditional jump.
CALL	Transfer control to another location. Before transfer, the address of the instruction following the CALL is placed on the stack.
JE/JZ	Jump if equal/zero.
LOOPE/LOOPZ	Loops if equal/zero. This is a conditional jump using a value stored in register ECX. The instruction first decrements ECX before testing ECX for the branch condition.
INT/INTO	Interrupt/Interrupt if overflow. Transfer control to an interrupt service routine.

4

46

4 Number of Addresses

- One of the traditional ways in describing processor architecture is using the _____ contained in each instruction .

What is the maximum number of addresses one might need in an instruction?

- In most architectures, most instructions have one, two, or three operand addresses.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.432. 47

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

Instruction		Comment
SUB	Y, A, B	Y ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction		Comment
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC − B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

(c) One-address instructions

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

(d) No-address instruction

4 instructions

6 instructions

10 instructions

8 instructions

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.417. 48

(a) Three-address instructions

$$Y = \frac{A - B}{C + (D \times E)}$$

Instruction		Comment
SUB	Y, A, B	Y ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

instructions

- 3 addresses: *Operand 1*, *Operand 2*, *Result* (Destination).
- May be a forth address - next instruction (usually implicit, obtained from PC (*Program Counter*)).
- Example below: *T* = temporary location used to store intermediate results.
- Not common in use.
- Needs very long words to hold everything.

49

(b) Two-address instructions

$$Y = \frac{A - B}{C + (D \times E)}$$

Instruction		Comment
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

instructions

- 2 addresses: *Operand*, *Result* (Destination).
- Reduces length of instruction and space requirements.
- Requires some extra works:
  - Temporary storage to hold some results.
  - Done to avoid altering the operand value.

50

(c) One-address instructions

$$Y = \frac{A - B}{C + (D \times E)}$$

Instruction	Comment
LOAD D	AC ← D
MPY E	AC ← AC × E
ADD C	AC ← AC + C
STOR Y	Y ← AC
LOAD A	AC ← A
SUB B	AC ← AC − B
DIV Y	AC ← AC ÷ Y
STOR Y	Y ← AC

instructions

- 1 address
- Implicit second address.
- Usually use a *CPU register* (accumulator)
  - Supplies 1 operand and store result.
  - One memory address used for other operand.
- Common on early machines.

51

(d) No-address instructions

$$Y = \frac{A - B}{C + (D \times E)}$$

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

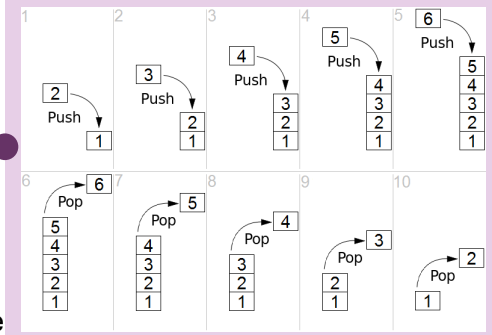
instructions

- 0 (zero) address
- All addresses implicit.
- Usually use a *stack* (a push down stack in CPU).
- There are two *Opcodes* with one *operand*: PUSH op, POP op.

52

(d) No-address instructions

Stack Machine



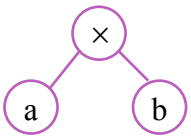
- A _____ is an abstract data type and data structure based on the principle of *Last In First Out* (LIFO).
- *Stack machine*: Java Virtual Machine.
- *Call stack* of a program, also known as a function stack, execution stack, control stack, or simply the stack.
- Application: *Reverse Polish Notation (RPN)*, *Depth-First-Search (DFS)*

RPN will be discussed in next example

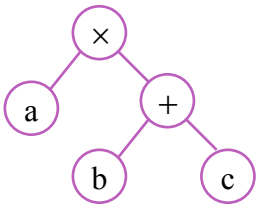
https://en.wikipedia.org/wiki/Stack_(abstract_data_type) 53

Expression tree

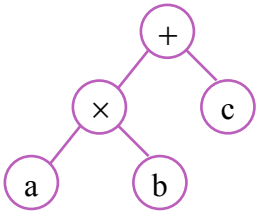
$a \times b$



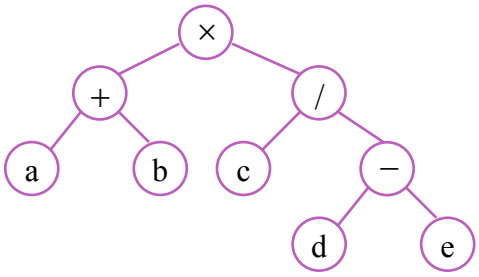
$a \times (b + c)$



$a \times b + c$

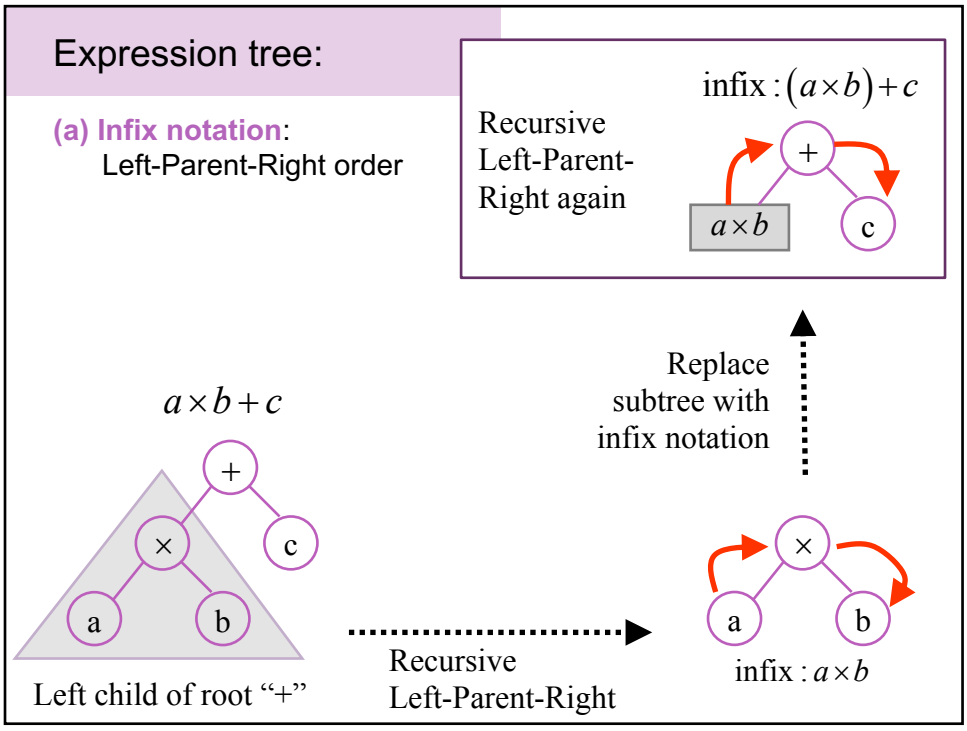


$(a + b) \times (c / (d - e))$



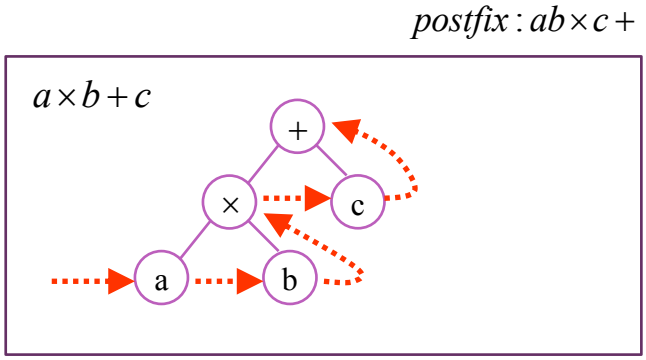
Expression tree:

(a) Infix notation:
Left-Parent-Right order



Expression tree:

(b) Postfix notation:
Left-Right-Parent order



4

Reverse Polish Notation (RPN)

- Precedence of multiplication is higher than addition, we need parenthesis to guarantee execution order.
- However in the early 1950s, the Polish logician Jan Lukasiewicz observed that parentheses are not necessary in postfix notation, called *RPN* (*Reverse Polish Notation*).
- The Reverse Polish scheme was proposed by F.L. Bauer and E.W. Dijkstra in the early 1960s to reduce computer memory access and utilize the *stack* to evaluate expressions .

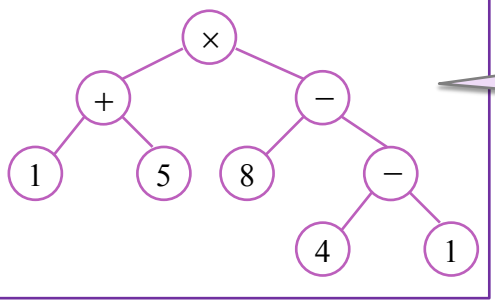
57

4

Example: Reverse Polish Notation (RPN) → *Postfix order*

Infix : $(1+5)\times(8-(4-1))$

Expression tree



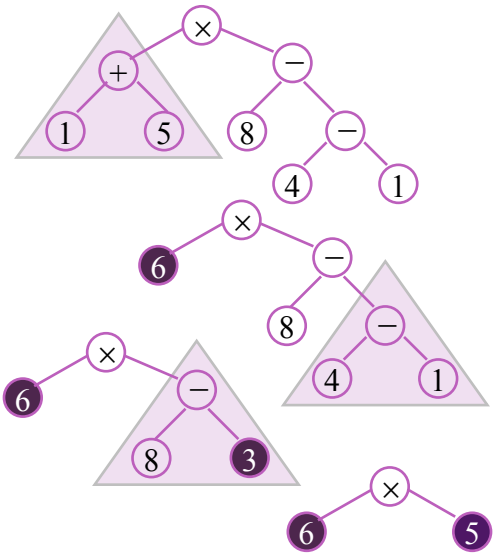
Postfix:

(parenthesis free)

58

Example: Reverse Polish Notation (RPN) → *Postfix order* [1]

$Infix : (1+5) \times (8-(4-1))$



$Postfix : \; 15+841--\times$

$1+5=6$

$6841--\times$

$4-1=3$

$683-\times$

$8-3=5$

59

Example: Evaluate RPN expression [1]

4

Expression

30

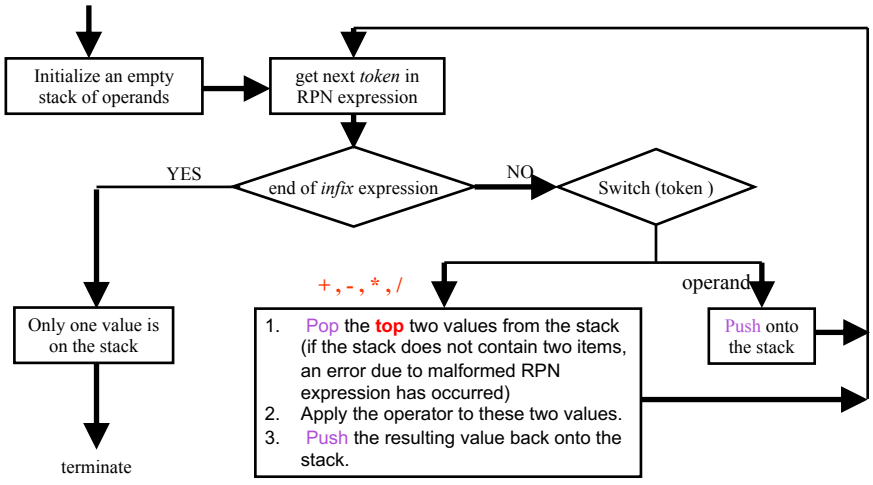
$Infix \;\; : \; (1+5) \times (8-(4-1))$

$Postfix : \; 15+841--\times$

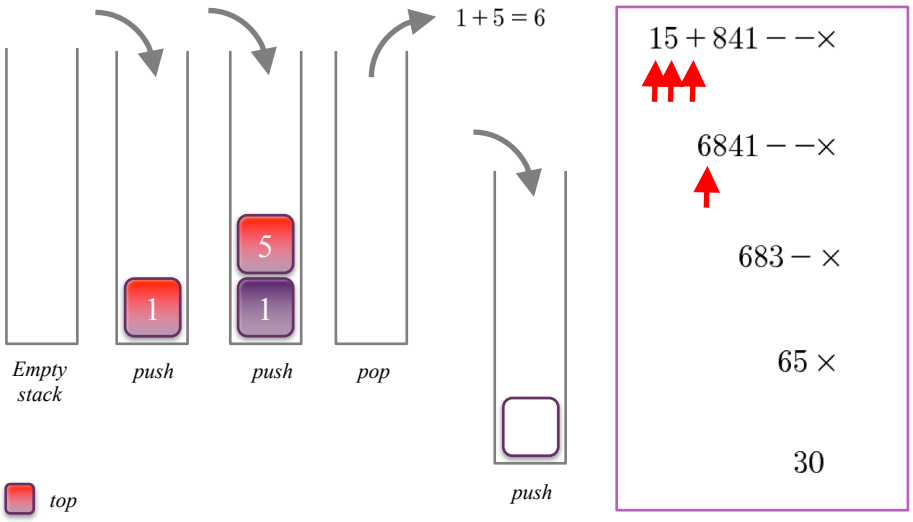
- **Scanned** from left to right until an operator is found, then the last two operands must be retrieved and combined.
- Order of operands satisfy LIFO, so we can use _____ to store operands and then evaluate RPN expression.

60

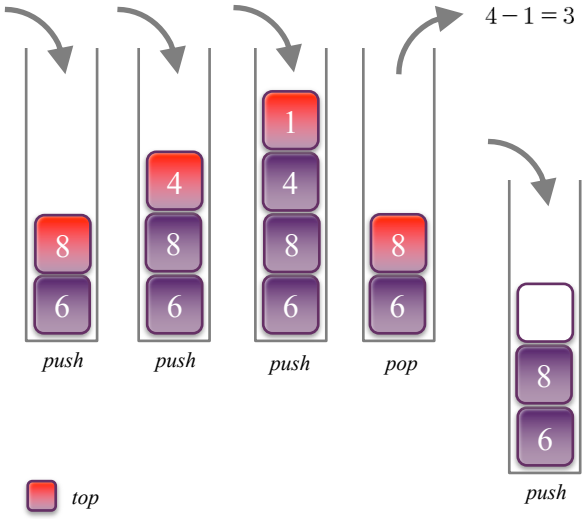
Example: Evaluate RPN expression → *Flow Chart*



Example: Evaluate RPN expression [2]



Example: Evaluate RPN expression [2]



$15 + 841 - - \times$

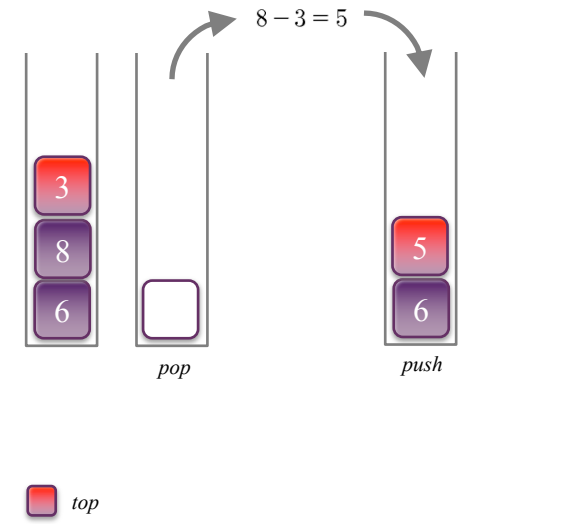
$6841 - - \times$

$683 - \times$

$65 \times$

30

Example: Evaluate RPN expression [2]



$15 + 841 - - \times$

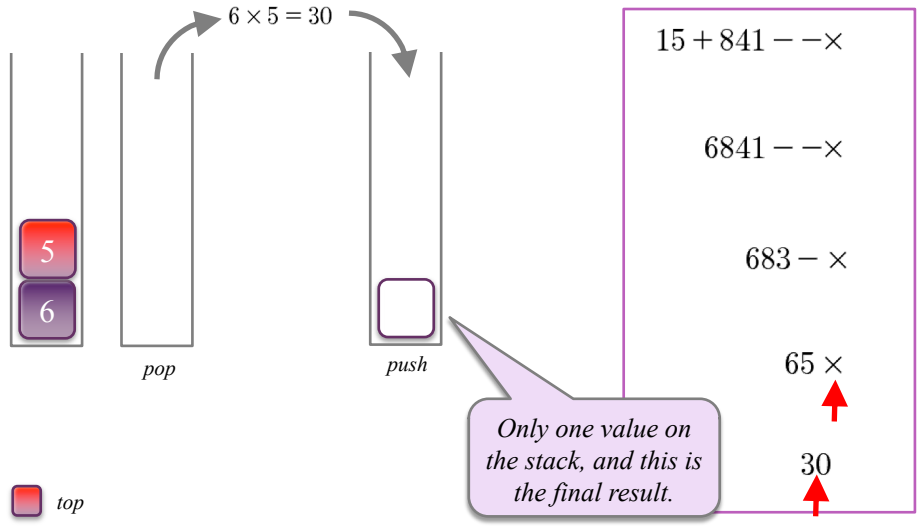
$6841 - - \times$

$683 - \times$

$65 \times$

30

Example: Evaluate RPN expression [2]



How Many Addresses?

4

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

More addresses:	Fewer addresses:
More complex (powerful?) instructions.	Less complex (powerful?) instructions.
_____ instructions per program.	_____ instructions per program.
More registers: → Inter-register operations are quicker.	Faster fetch/execution of instructions.

66

4

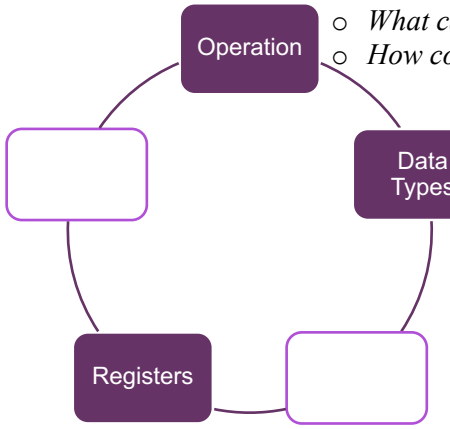
Instruction Set Design

- 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 _____.
- The instruction set is the programmer's means of controlling the processor. Thus, programmer requirements must be considered in designing the instruction set.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.434. 67

4

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



- *How many operations?*
- *What can they do?*
- *How complex are they?*
- *The mode(s) by which the address of an operand is specified.*
- *The various types of data.*
- *Number of CPU registers available.*
- *Which operations can be performed on which registers?*
- *Length of opcode field.*
- *Number of addresses.*

68

Module 4
Instruction Set Architecture (ISA)

- Overview
- Numbers
- Characters
- Logical Data

4

Overview

- Assembly language built from two pieces:

```
MOV   R1, R2
```

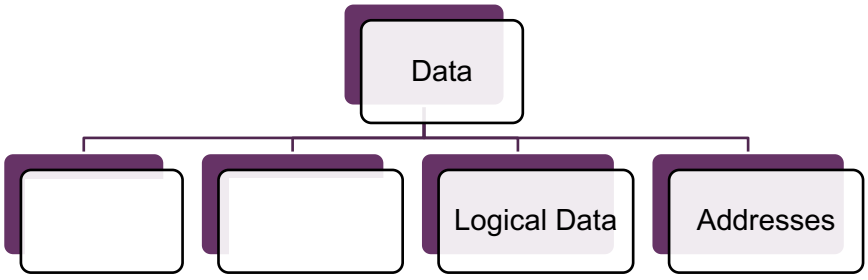
_____: *What to do with the data (ALU operation)*

_____: *Where to get data (R2) and put the results (R1)*

70

4

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



(*Section* 4.4)

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.435. 71

4

Table: Type of operand for Pentium 4

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

Single Precision

72

4

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

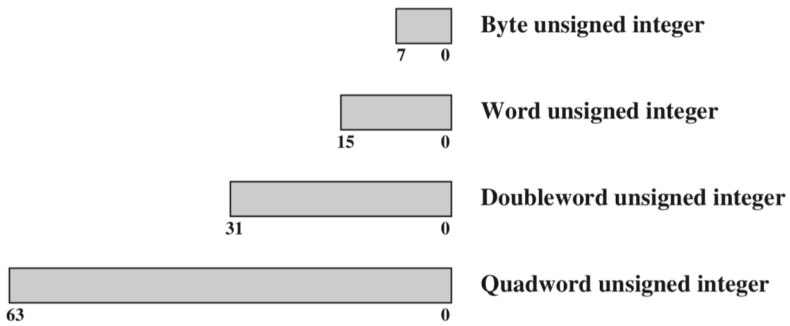
William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.435. 73

4

Example 3a: x86 numerical data types.

(*Unsigned integers*)

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



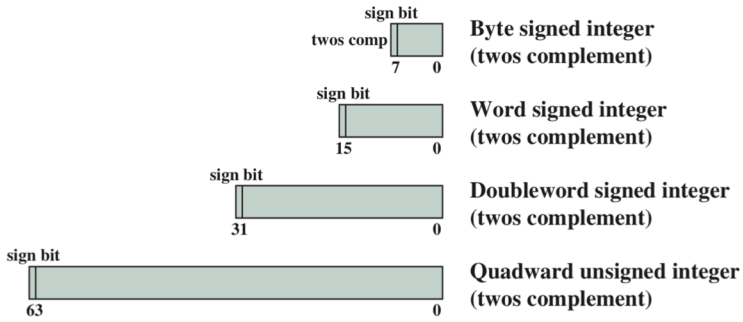
William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.424. 74

4

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.

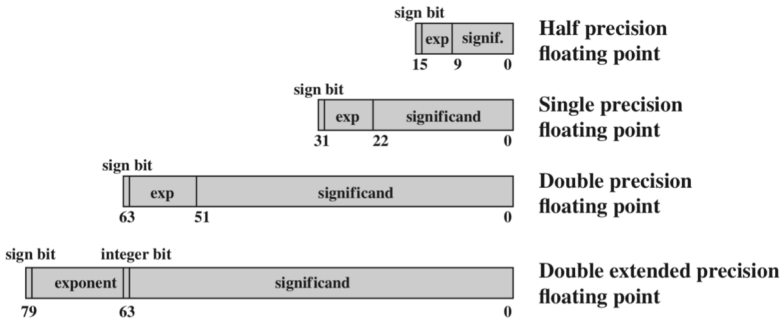


4

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.



4

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.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.436. 77

4

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.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.436. 78

Module 4
Instruction Set Architecture (ISA)

4.1 Introduction
4.2 Machine Instruction Characteristics
4.3 Types of Operands
4.4 Addressing Modes
4.5 Instruction Formats
4.6 Summary

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

4 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.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.474. 80

Notations:

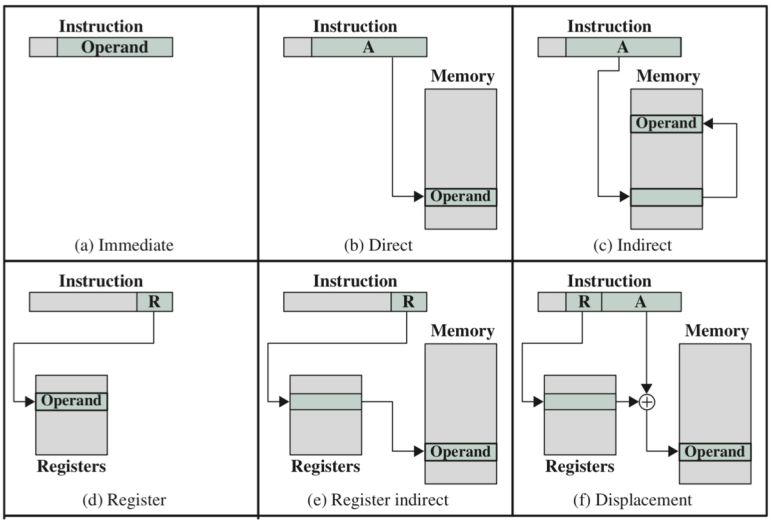


Figure: Addressing modes.

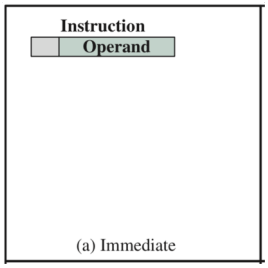
A = contents of an address field in instruction.

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.

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.458. 81



Operand = A

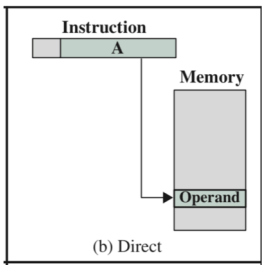
4 (a) Immediate Addressing

- 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.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.476. 82



EA = A

4

(b) Direct Addressing

- The address field contains the *Effective Address* (EA) of the operand.

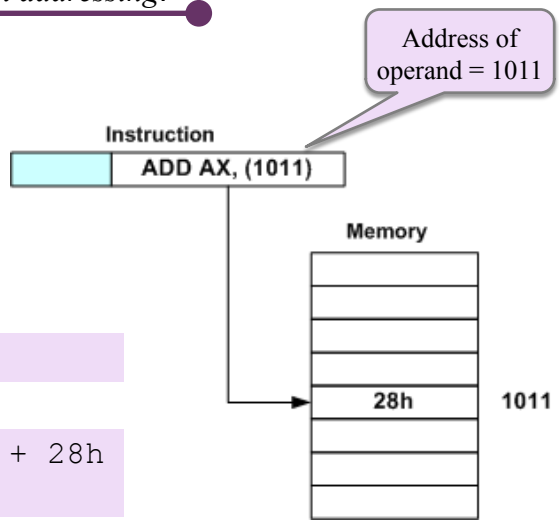
Requires only one memory reference and no special calculation.

It provides only a limited address space.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.476. 83

4

Example 4: *Direct addressing.*




```
AX = 10h
```

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

84

4

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*.

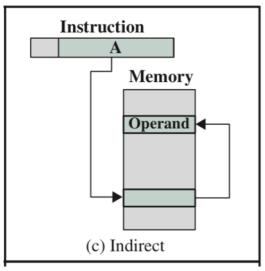
```
.data

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  =
bx  =
ecx =
```

(*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
```

86



$EA = (A)$

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

4 (c) Indirect Addressing

- 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.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.477. 87

Example 6: *Indirect addressing*.

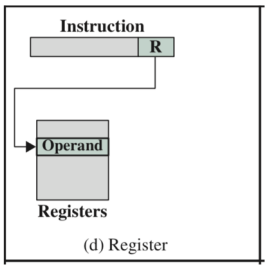
```
.data

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
```

```
bl  =
cx  =
edx =
ax  =
```

(*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
```

88



EA = R

4 (d) Register Addressing

- Similar to *direct addressing*, except 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.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.476. 89

Example 7: *Register addressing.*

```
.data

.code

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

      mov  eax,ebx
      add  eax,ecx

      call dumpregs
      exit
main ENDP
```

eax =

ebx =

ecx =

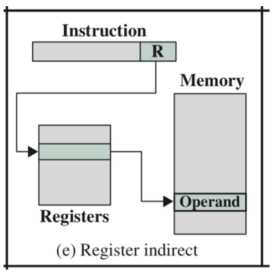
eax =

eax =

(*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
```

90



EA = (R)

4 (e) Register Indirect Addressing

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

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

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.478. 91

Example 8:

Register indirect addressing.

```
esi =
edi =
bl  =
cx  =
edx =
al  =
```

```
.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.

(*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
```

92

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
```

Offset :	Value:	Data label:
00404004:		array1
00404005:		
00404006:		
00404007:		
00404008:		array2
00404009:		

93

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
```

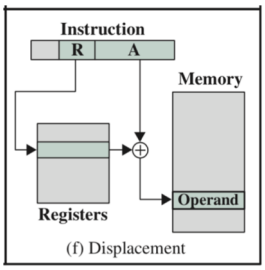
```
ebx =
dl  =
ebx =
cl  =
```

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
```

94



$$EA = A + (R)$$

(f) Displacement 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)

3 common displacement addressing techniques:

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.478. 95

96

(f.1) Relative Addressing

4

- 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.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.479. 97

Example 10:

Displacement Addressing.

(1) *Relative addressing*

$$Destination = Target - Source$$
$$= L1 - PC$$
$$=$$
$$=$$

```
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
```

–(ve) because its jumping backwards

PC = 39

Need to go to L1 which is in 00000028

98

(f.2) Base-Register Addressing

4

$$EA = A + R$$

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

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.479. 99

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

100

Example 11:

Displacement Addressing.
(b) Base-register addressing

```
esi =
ebx =
 ax =
ebx =
 cx =
```

```
.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
```

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
```

101

(f.3) Indexed Addressing

4

$$EA = A + R$$

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

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.479. 102

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

103

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
```

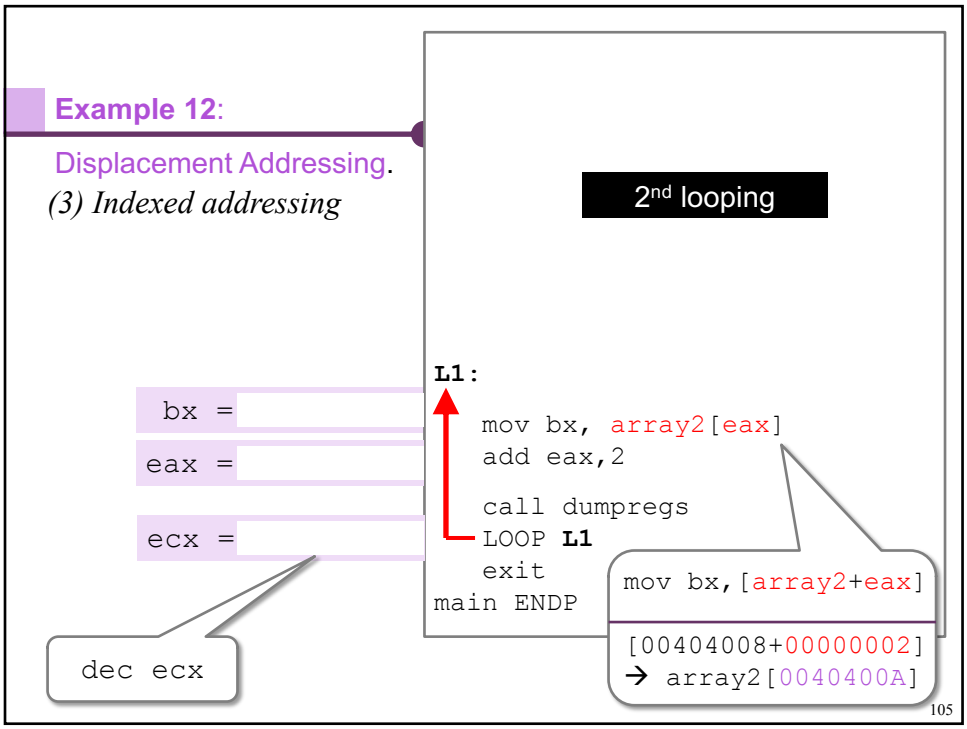
```
eax =
ecx =
 bx =
eax =
ecx =
```

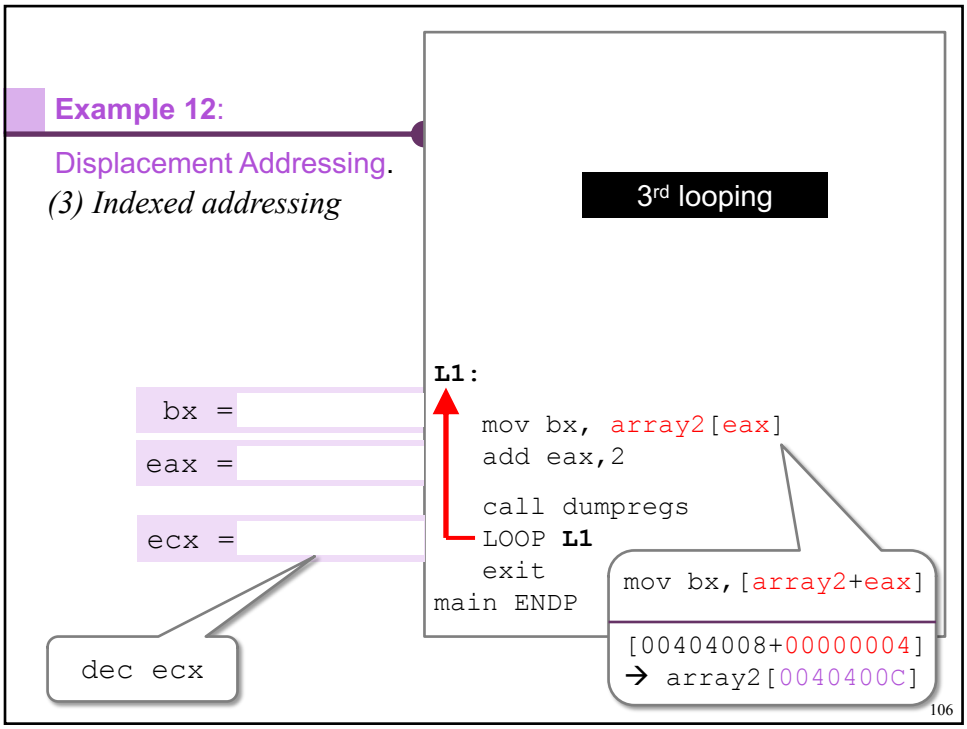
dec ecx

mov bx,[array2+eax]

[00404008+00000000]
→ array2[00404008]

104


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 =
eax =
ecx =
dec ecx
mov bx,[array2+eax]
[00404008+00000002]
→ array2[0040400A]
105


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 =
eax =
ecx =
dec ecx
mov bx,[array2+eax]
[00404008+00000004]
→ array2[0040400C]
106

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
```

bx =

eax =

ecx =

dec ecx

mov bx,[array2+eax]

[00404008+00000006]
→ array2[0040400E]

107

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
```

108

4

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

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.459. 109

4

110

Module 4
Instruction Set Architecture (ISA)

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

4 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.

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.469. 112

4

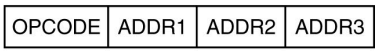
- Four common instruction formats:

(a) Zero-address instruction.

(b) One-address instruction

(c) Two-address instruction.

(d) Three-address instruction.



113

4

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)?

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.469. 114

4

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.

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.470. 115

4

- The following interrelated factors go into determining the use of the addressing bits:
  - Number of _____ – if indicated implicitly, certain opcodes might always call for indexing; if explicit – one or more mode bits will be needed.
  - Number of _____ – 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 need to specify the registers.

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.470. 116

4

- Number of _____ – have one set of general purpose registers with 32 or more registers in the set – for example sets of 8 registers only 3 bits are needed to identify the registers, opcode will implicitly determine 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.

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.471. 117

4

Variable-Length Instructions

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

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.473. 118

4.6 Summary

4

- 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.

William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.413, 457. 119