THE UNIVERSITY OF ARIZONA
UA South

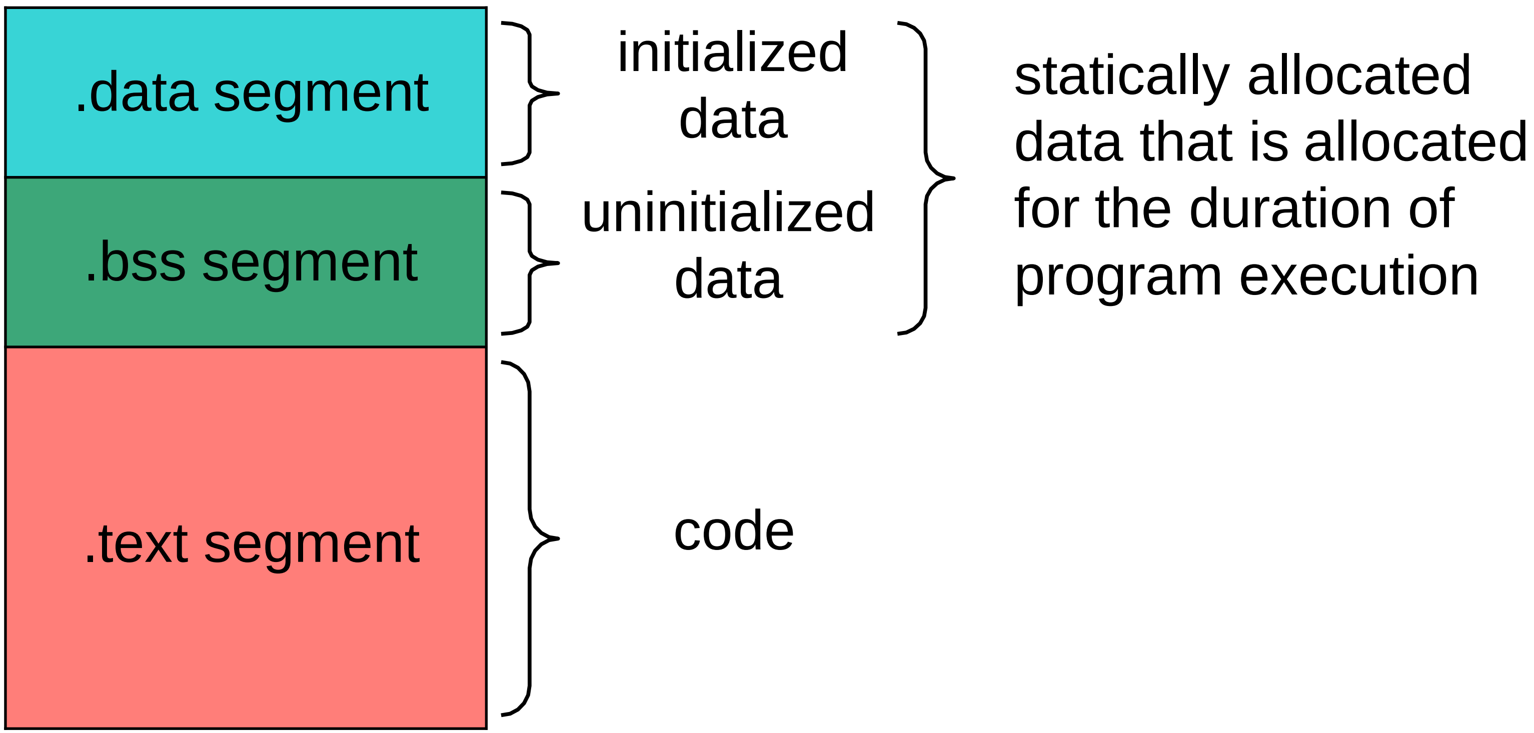
CYBV 471 Assembly Programming for Security Professionals
Week 5

X.86 Instruction and NASM Program Structure

Agenda

- **NASM Program Structure**
- **Define variables in .data section**
- **Big and Little Endian s**
- **Define variables in .bss section**
- **NASM program Skelton**
- **Binary Code and X.86 Instruction**
- **Basic Elements of Assembly Language Program**
  - Integer constants and expressions
  - Character and string constants
  - Reserved words and identifiers
  - Directives and instructions
  - Labels
  - Mnemonics and Operands
  - Comments

NASM Program Structure



Define variables in .data and .bss sections

- Both sections contains data directives that declare pre- allocated zones of memory

  section .data   (segment .data)
  ; Define variables with initialized values in the data section
  var_name (label) dx   intial_value   (d for define, x for data type)

  section .bss   (segment .bss)
  ; Define variables with uninitialized values in the data section
  var_name (label) resx memory size   (res for reserve, x for data type)

- The above “x” refers to the data size

Unit	Letter(x)	Size in bytes
byte	b	1
word	w	2
double word	d	4
quad word	q	8
ten bytes	t	10

Define variables in .data sections

section .data   (segment .data)

; Define variables with initialized values in the data section

var_name (label) dx   intial_value   (d for define, x for data type)

To declare a variable of initialized memory location using three elements:

- Label: the variable name used in the program to refer to that zone of memory
  - A pointer to the zone of memory, i.e., an address
- dx, where x is the appropriate letter for the size of the data being declared
- Initial value, with encoding information
  - default: decimal
  - b: binary
  - h: hexadecimal
  - o: octal
  - quoted: ASCII

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.

Decimal	Hexadecimal	Binary	Octal	Char
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^

Decimal	Hexadecimal	Binary	Octal	Char
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Define variables in .data section

var_name (label) dx intial_value (d for define, x for data type)

- Examples

L2 dw 1000

define a variable L2, 2-byte word, initialized to 1000 (decimal)

L3 db 110101b

define a variable L3, 1 byte, initialized to 110101 in binary

L4 db **0**A2h

define a variable L4, 1 byte, initialized to A2 in hex (note the '**0**')

L5 db 17o

define a variable L5, 1 byte, initialized to 17 in octal

L6 db "B"

define a variable L6, 1 byte, initialized to ASCII code for "B" (66d)

L7 dd 0FFFF1B78h

define a variable L7, 4-byte double word, initialized to 0FFFF1B78 in hex (note the '**0**')

Define Multiple Elements

- Examples

  L8 db 0, 1, 3, 4

  define a variable L8, has 4 bytes, initialized to ,0, 1, 3, 4 (decimal)

  L8 is a pointer to the first byte

  L9 times 100 db 0

  define a variable L9, has 100 bytes, initialized to 0 (decimal)

  L9 is a pointer to the first byte

  L10 db "Hello", 0

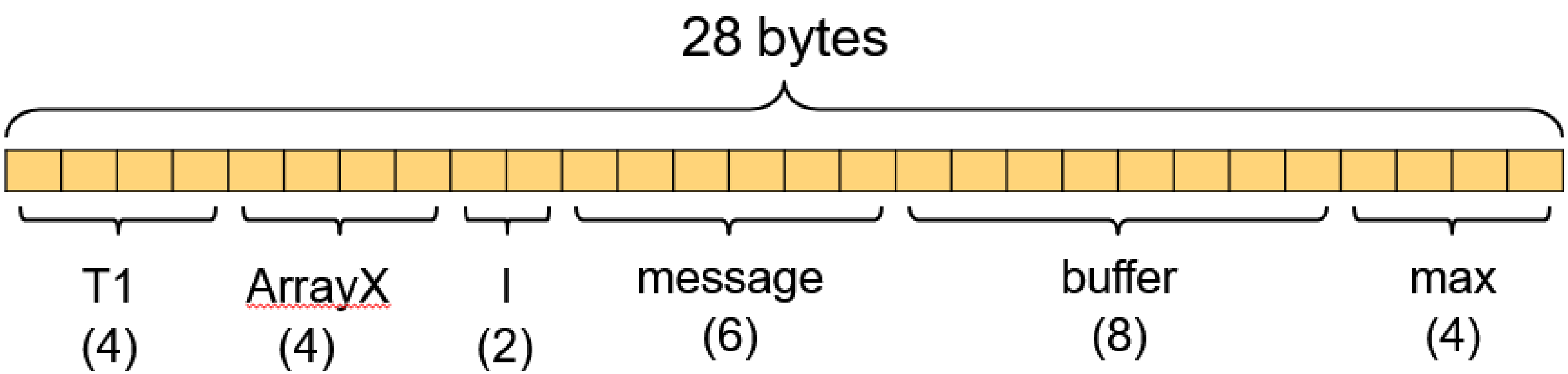
  db "H", "e", "l", "l", "o", 0

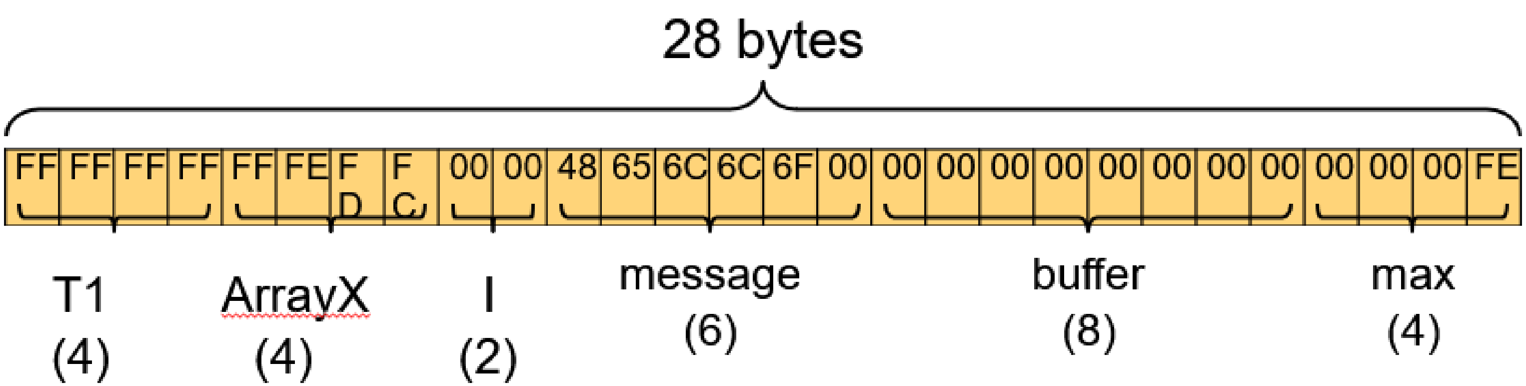
  define a variable L9, a null-terminated string of 6 bytes, initialized to "Hello\0"

  L10 is a pointer to the beginning of the string

- Examples

```
 T1          dd  -1                          ; 4 bytes in memory
 ArrayX   db 0FFh, 0FEh, 0FDh, 0FCh     ; 4 bytes in memory
 I           dw   0                          ; 2 bytes in memory
message   db “Hello”, 0                    ; 6 bytes in memory
buffer   times 8  db 0                     ; 8 bytes in memory
max         dd     254                       ; 4 bytes in memory
```





Big and Little Endian Representation

- **Big-Endian**
  - The MSB (Most Significant Byte) of a word is stored at the lowest memory address for that word
  - Subsequent bytes from MSB to LSB are stored in sequential addresses
- **Little-Endian**
  - The LSB of a word is stored at the lowest memory address for that word
  - Subsequent bytes from LSB to MSB are stored in sequential addresses

32-bit hexadecimal number

0x12345678

0x12 0x34 0x56 0x78

(MSB) *(LSB)*

Address	Big Endian	Little Endian
0	0x12 (MSB)	0x78 (LSB)
1	0x34	0x56
2	0x56	0x34
3	0x78	0x12

Little Endian Order

- All data types larger than a byte store their individual bytes in reverse order.
- The least significant byte occurs at the first (lowest) memory address.

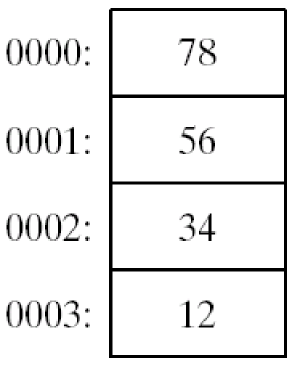
- Example:

  val1 DWORD 12345678h

  mov eax, val1 > eax = 0000

  mov eax, [val1] > eax = 78

  mov eax, [val1 + 1] > eax = 56



Big Endian Order

- All data types store its individual bytes in correct order in registers.
- All network data types larger than a byte store their individual bytes in correct order.
- The least significant byte occurs at the first (highest) memory address.
- Example:

12345678h will be stored in a register in correct order

val1 DWORD 12345678h

mov eax, val1 > eax = 0000

mov eax, [val1] > eax = 12

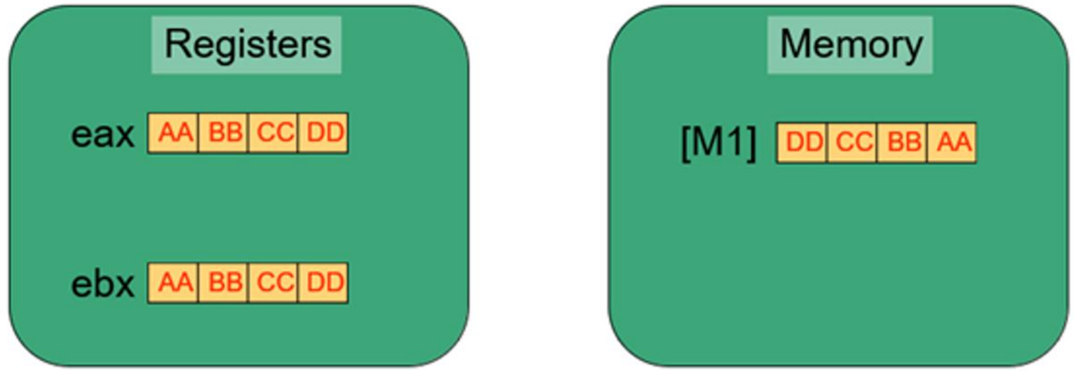
mov eax, [val1 + 1] > eax = 34

mov eax, DWORD [val1] > eax = 12345678

Offset	Value
0000:	12
0001:	34
0002:	56
0003:	78

Big and Little Endian Representation

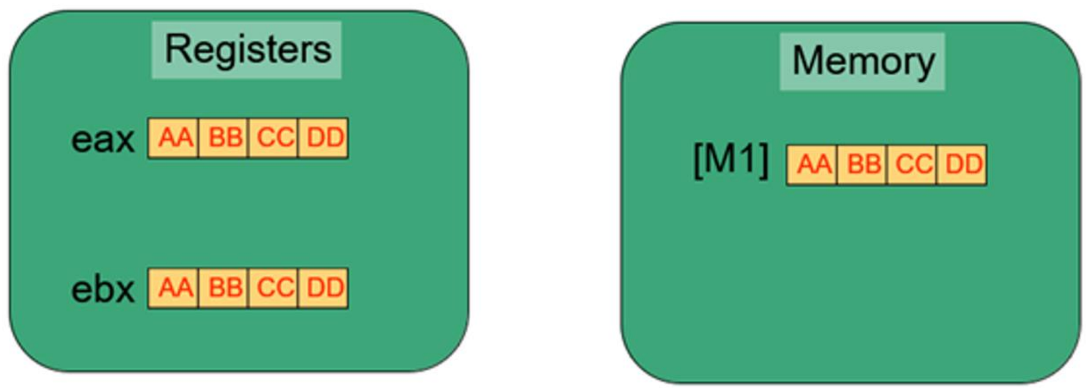
mov eax, 0AABBCCDDh
mov [M1], eax
mov ebx, [M1]



Big Endian

Little Endian

mov eax, 0AABBCCDDh
mov [M1], eax
mov ebx, [M1]



Big Endian

Big Endian

Big and Little Endian Representation

- Motorola and IBM processors use(d) Big Endian
- Intel/AMD uses Little Endian (used in this class)
- Date in registers follows Big Endian representation
- Data in memory follows Little Endian representation
- This only matters when writing **multi-byte** quantities to memory and reading them differently (e.g., byte per byte)

For example, the IP address 127.0.0.1 will be represented as 0x7F 00 00 01 in big-endian format (over the network) and 0x01 00 00 7F in little-endian format (locally in memory).

How defined variables look in Memory?

```
pixels        times 4      db     0FDh
x             dd      00010111001101100001010111010011b
blurb         db      "ad", "b", "h", 0
buffer        times  10    db   14o
min           dw      -19
```

x **dd** 00010111001101100001010111010011b

00010111	00110110	00010101	11010011
17h	36h	15h	D3

blurb **db** "ad", "b", "h", 0

a= 61, d = 64, b = 62, h = 68

min **dw** -19 (FFFD)

FD	FD	FD	FD	D3	15	36	17	61	64	62	68	00	0C	0C	0C	0C	0C	0C	0C	0C	0C	0C	ED	FF

pixels (4) x (4) blurb (5) buffer (10) min (2)

Declare variables in .bss sections

section .bss   (segment .bss)

; Define variables with uninitialized values in the data section

var_name (label) resx memory size   (res for reserve, x for data type)

- Examples

L1   resb 1

reserve uninitialized byte in the memory for the variable L1

L1 is a pointer to an address in the memory

L2   resw 100

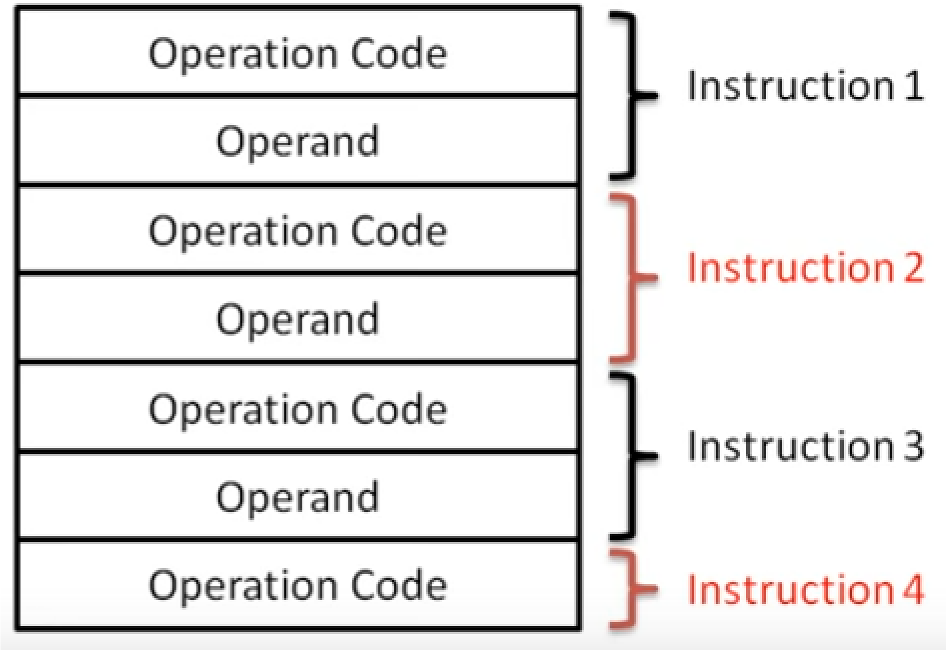
reserve 100 2-byte word in memory

L2 is a pointer to the first word address in the memory

Binary Code

- A machine code instruction is binary code that has the following forr
  - Operation code (Op Code) +   Operand
  - Operation code (Op code)

A machine code program consists of a sequence of Op Codes and Operands stored in the memory (RAM)



Op Code	Operand
00111110	00000001

For simplicity convert the binary code to Hexadecmial

0011 1110 0000 0001

3 E 0 1

Op Code	Operand
3E	01

RAM	
00111110	Address 1
00000001	Address 2

RAM

Binary Code

	RAM
Address 1	3E
Address 2	01

RAM

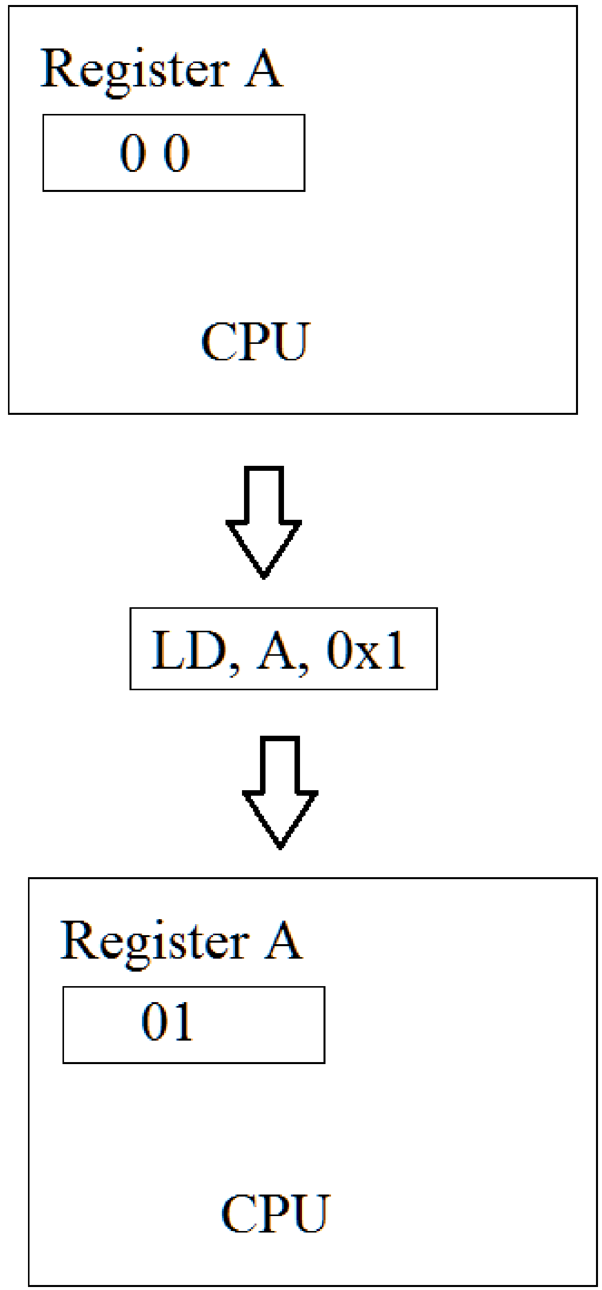
Hexdecimal Format

Disassembler

LD, A, 0x1

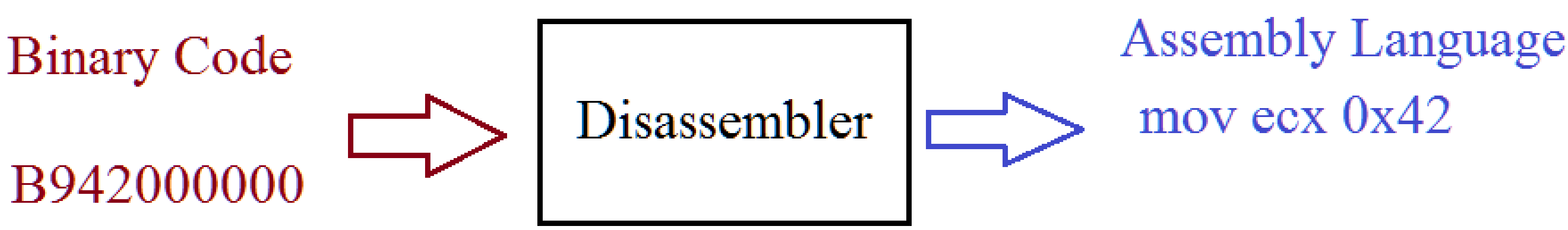
Operand

load (instruction), Register (destination), Source (value)


Register A
0 0
CPU
LD, A, 0x1
Register A
01
CPU

x86 instructions

- Instructions are the building blocks of assembly programs.
- **Mnemonic** **(**word identifies the instruction to execute) followed by **operands** (register and/or data)
- mov ecx 0x42
  - Move into Extended C Register the value 42 (hex)
- In binary code, the above instruction is B942000000



More information for .text section

Before and after running the instructions of your program there is a need for some "setup" and "cleanup"

We'll understand this later when covering the stack.

The text segment will always looks like the following

```
enter   0,0       ; setup the program
pusha
;
; Your program here
;
popa
mov     eax, 0    ; cleanup the program
leave
ret
```

NASM Program Skelton

A general NASM file could have the following format

```
; comment  (HelloWorld.asm)
include library files
section .data   (segment .data)
        ; Define variables with initialized values in the data section
       ; var_name  dx   value     (d for define, x for data type)

section .bss   (segment .bss)
    ; Define variables with uninitialized values in the data section
     var_name resx  memory size   (res for reserve, x for data type)

; Code goes in the text section
section .text       (segment .text)
        global _start
_start:
       enter 0,0
       pusha

        ; Code goes in the text section

        popa
        mov eax, 0
        leave
        ret
```

Rewrite First Assembly Program

```
segment .data
        msg:     db 'Hello world!',10
        msgLen:  equ $-msg

segment .text
        GLOBAL _start
_start:
        enter 0, 0
        pusha

        mov eax,4              ;  use 'write' system call = 4
        mov ebx,1              ; file descriptor 1 = STDOUT
        mov ecx, msg           ; string to write
        mov edx, msgLen        ; length of string to write
        int 80h                ; call the kernel

        ; Terminate program
        mov eax,1              ; 'exit' system call
        mov ebx,0              ; exit with error code 0
        int 80h                ; call the kernel

        popa
        mov eax, o
        leave
        ret
```

Basic Language Elements

- **Basic Elements of Assembly Language**
  - Integer constants and expressions
  - Character and string constants
  - Reserved words and identifiers
  - Directives and instructions
  - Labels
  - Mnemonics and Operands
  - Comments

Understand Integer Constants

- You can define an integer value as binary, decimal, binary, hexadecimal, or octal digits

  d – decimal: 1434d (base-10)

  b – binary: 1011b (base-2)

  h – hexadecimal:3ABh (base-16)

  o – octal:24o (base-8)

  q – octal:24q (base-8)

Integer Expressions

- An *integer expression* is a mathematical expression involving integer values and arithmetic operators.
- The result of an integer expression is an integer
- Operators and precedence levels:

Operator	Name	Precedence Level
( )	parentheses	1
+, -	unary plus, minus	2
*, /	multiply, divide	3
MOD	modulus	3
+, -	add, subtract	4

Expression	Value
16 / 5	3
-(3 + 4) * (6 - 1)	-35
-3 + 4 * 6 - 1	20
25 mod 3	1

Character and String Constants

- A *character* is a single character enclosed in single or double quotes.
- The assembler stores the value of each character in memory as the character's binary ASCII code.
- Examples:

'A' (will be stored in memory as in one byte as)

65d or 41h, or 1000001b, or 101o

"d" (will be stored in memory as in one byte as)

100d, 64h, 1100100b, or 144o

String Constants

- A *string* is a sequence of characters (including spaces) enclosed in single or double quotes:
- String is terminated by NULL (0)
- Example: ‘ABC’ (three characters)

  ‘A B C’ (five characters)

  '4096’ (four characters) (this is not 4096 value)

Embedded quotes are permitted when used in the manner shown by the following examples:

"This isn't a test"

'Say "Good night," Gracie’

- String literals are stored in memory as sequences of integer byte values
- For example, the string “ABCD” will be stored in memory in four bytes (41h, 42h, 43h, and 44h)

String Constants

- Some string operations requires the length of a string. In that case, you need to define the length of the string

```
msg db 'Hello, world!’      ; has 13 characters
 len equ $ - msg             ; get the length of the string
len equ 13                   ; define the length of the string
```

- Alternatively, you can store strings with a trailing sentinel character (0) to delimit a string instead of storing the string length explicitly.

```
msg db 'Hello, world!’, 0      // 0 terminate the string
```

In that case, you don’t need to determine the length of the string

Reserved Words

- Reserved words have special meaning in assembly language and can only be used in their context.
- You can’t use these reserved words to define variables
- There are different types of reserved words:
  - Instruction mnemonics, such as MOV (mov), ADD (add)
  - Register names such as EAX (eax)
  - Directives: Tell the assembler how to assemble programs (such as .DATA, .CODE)
  - Attributes, which provide size and usage information for variables and operands. (such as are BYTE and WORD)
  - Operators, used in constant expressions such as 3+ 5, 3*20
  - Predefined symbols: such as @data, which return constant integer value at a run time
  - A common list of reserved words can be found in Appendix A.

Instructions

- An instruction is an executable statement that becomes executable when a program is assembled.
- Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at run time.
- Executed at runtime by the CPU
- We use the Intel IA-32 instruction set
- An instruction contains:
  - Label (optional)
  - Mnemonic (required)
  - Operand (depends on the instruction)
  - Comment (optional)

Labels

- Act as place markers
- Data label (variable)
  - must be unique
  - example: count DWORD 100 (not followed by colon)
- Code label
  - target of jump and loop instructions
  - example: **target1:** (followed by colon)

```
            code instructions
            code
            JMP target1
            code
target 1:
            code
            -----
            code
```

Instructions

- Examples: MOV, ADD, SUB, MUL, INC, DEC
  - MOV: move (assign) one value to another
  - ADD: add two values
  - SUB: subtract one value from another
  - MUL: multiply two values
  - JMP: jump to a new location
  - INC: add 1
  - DEC: substract 1
  - CALL: call a procedure

Operands

- An operand is a value that is used for input or output for an instruction. Assembly language instructions can have between zero and three operand.
- Operand can be integer value, integer expression, a register, memory or input–output port.
- Examples of assembly language instructions having varying numbers of operands
  - No operands

```
stc                 ; set carry flag
```

  - One operand

```
inc eax             ;  increment eax register
inc var1            ;  increment var1 that stored in memory
```

  - Two operands

```
add ebx, ecx        ;  add content of ecx register to ebx register
sub var2, 25        ; substract 25 from var2 that is stored in memory
add eax,36 * 25     ; register, constant-expression
```

Comments

- Comments are good!
  - explain the program's purpose
  - when it was written, and by whom
  - revision information
  - tricky coding techniques
  - application-specific explanations
- Single-line comments or
- Multi-line comments

Comments

- Single-line comments
  - begin with semicolon (;)
  - Example: add eax, 5 ; add 5 to the content of register eax

- Multi-line comments
  - begin with COMMENT directive and a programmer-chosen character
  - end with the same programmer-chosen character
  - Example:

```
COMMENT !
This is a comment1.
This is a comment2.
!

COMMENT &
This is a comment1.
This is a comment2.
&
```

Defining Strings in Multiple Lines

To continue a single string across multiple lines, end each line with a comma:

```
menu BYTE "Checking Account",0dh,0ah,0dh,0ah,
          1. Create a new account",0dh,0ah,
          2. Open an existing account",0dh,0ah,
          3. Credit the account",0dh,0ah,
          4. Debit the account",0dh,0ah,
          5. Exit",0ah,0ah,
          Choice> ",0
```

Putting It All Together

You should know:

- **NASM Program Structure**
- **Define variables in .data section**
- **Big and Little Endian**
- **Define variables in .bss section**
- **Binary Code and X.86 Instruction**
- **Basic Elements of Assembly Language**
  - Integer constants and expressions
  - Character and string constants
  - Reserved words and identifiers
  - Directives and instructions
  - Labels
  - Mnemonics and Operands
  - Comments

Questions?

Coming Next Week

Assembly Language Instructions

Week 5 Assignments

- **Learning Materials**

  1- Week 4 Presentation

  2- Read pages 201-211 (in Ch.5): Duntermann, Jeff. Assembly Language Step by Step, Programming with Linux,

Assignment

1- Complete "Lab#5" by coming Sunday 11:59 PM.