

CSIT 5740 Introduction to Software Security

Note set 3A

Dr. Alex LAM



DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

The set of note is adopted and converted from a software security course at the Purdue University by Prof. Antonio Bianchi

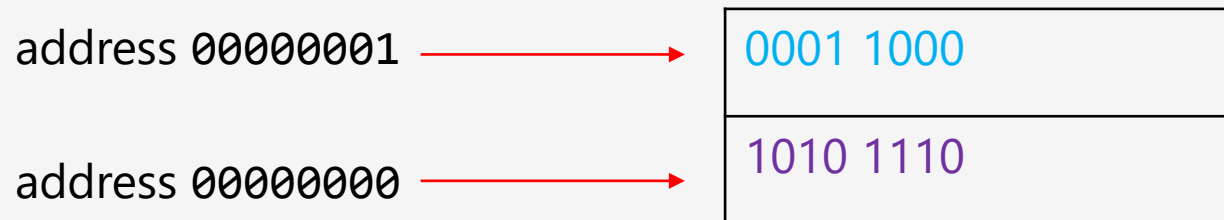
Assembly preliminaries

Data representation

- All data is represented by bits (full term: binary-digits)
 - Each bit carries a single binary value, either **0** or **1**
- We group **8 bits** of data together to form **a single byte**
- The following binary representation is 16-bit or 2-byte:
 - 0001 1000 1010 1110

Data representation

- “Byte” is the **unit of data storage** in most of the computers today.
- To enable access to computer memory efficiently. The memory of a computer is divided into “slots”, each slot corresponds to the storage space for a single “byte”.
- Each of the memory slots is assigned an integer number. And that number is called the memory address of the slot. If we have the “memory address”, we can just send it to the computer and the computer can retrieve the data from that memory slot (storing 1 byte of data).
- The previous binary representation 0001 1000 1010 1110 is 16-bit, it will take two memory slots to store, and it could be stored like the below (**little endian order**, more on this later):



Data representation

- Writing the binary representation is a bit clumsy, it can easily become too long
- A more compact way to write the same representation is to convert **every 4 bits** into a **single hexadecimal digit** using the following tables:

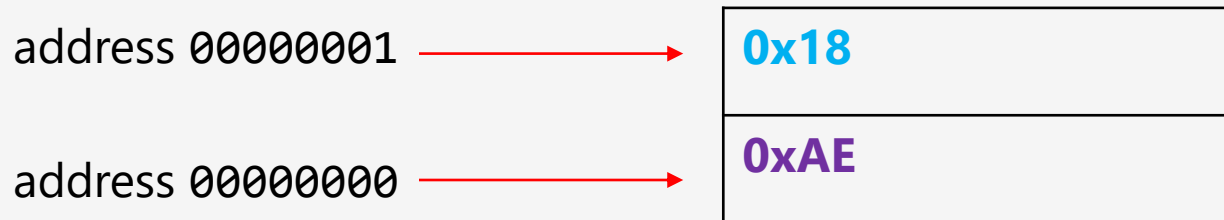
Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

- Using the above tables **0001 1000 1010 1110** could be represented as **18AE**
- To indicate it is the hexadecimal representation, we put “**0x**” to the left, writing it as **0x18AE** (to get back the original binary representation, just use the above tables reversely, how?)

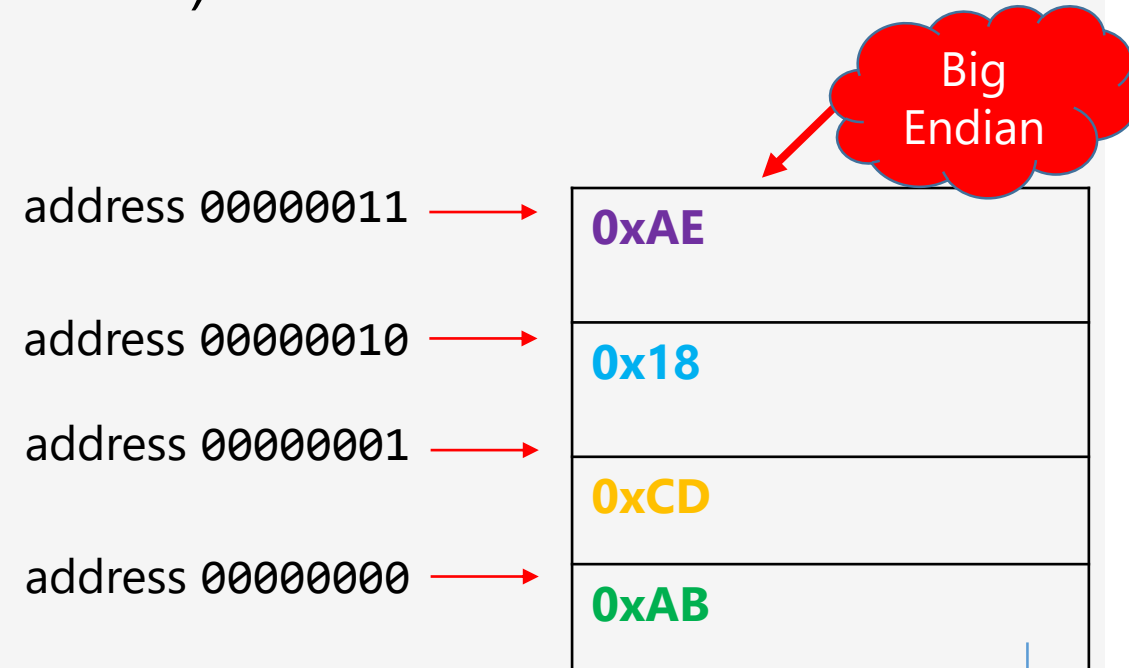
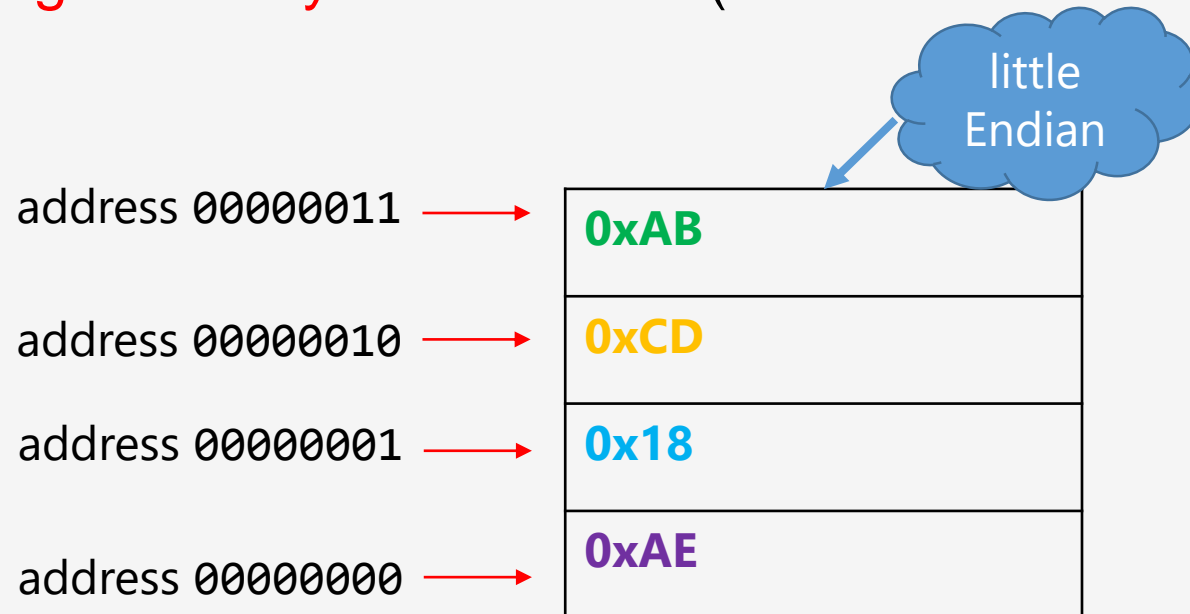
Data representation

- The representation **0x18AE** converts the original binary representation (**base 2**) into the hexadecimal representation (**base 16**)
- We typically write the data stored in the memory using the hexadecimal representations. Therefore instead of showing **0001 1000 1010 1110** in the memory, you will typically see the following in the memory slots (i.e. the software, like `gdb`, will display the following to you)



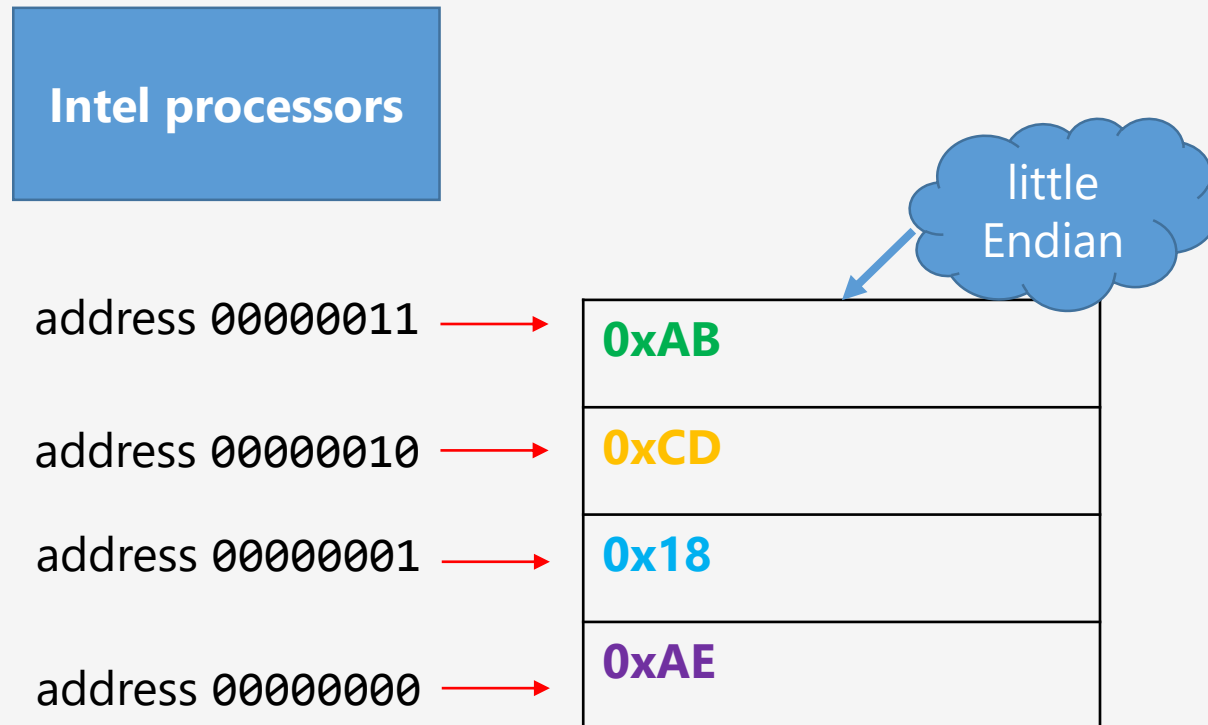
Data representation

- The data we would like to store is typically bigger than 1 byte, and will take multiple memory slots to store (each memory slot can store 1 byte only)
- The data/information represented by **0xABCD18AE** could be a number, 4 characters, an instruction, etc. It will be stored to the memory occupying 4 slots
- This multi-byte data could be stored in one of the following ways. Mind that the **rightmost byte** of the data (i.e. **AE** of **0xABCD18AE**) is called the “**end**”:



Data representation

- Intel processors use “**little endian**” approach to store multi-byte data
- When we read the memory for 4 bytes of data, remember to reconstruct it by assuming the little endian storage approach

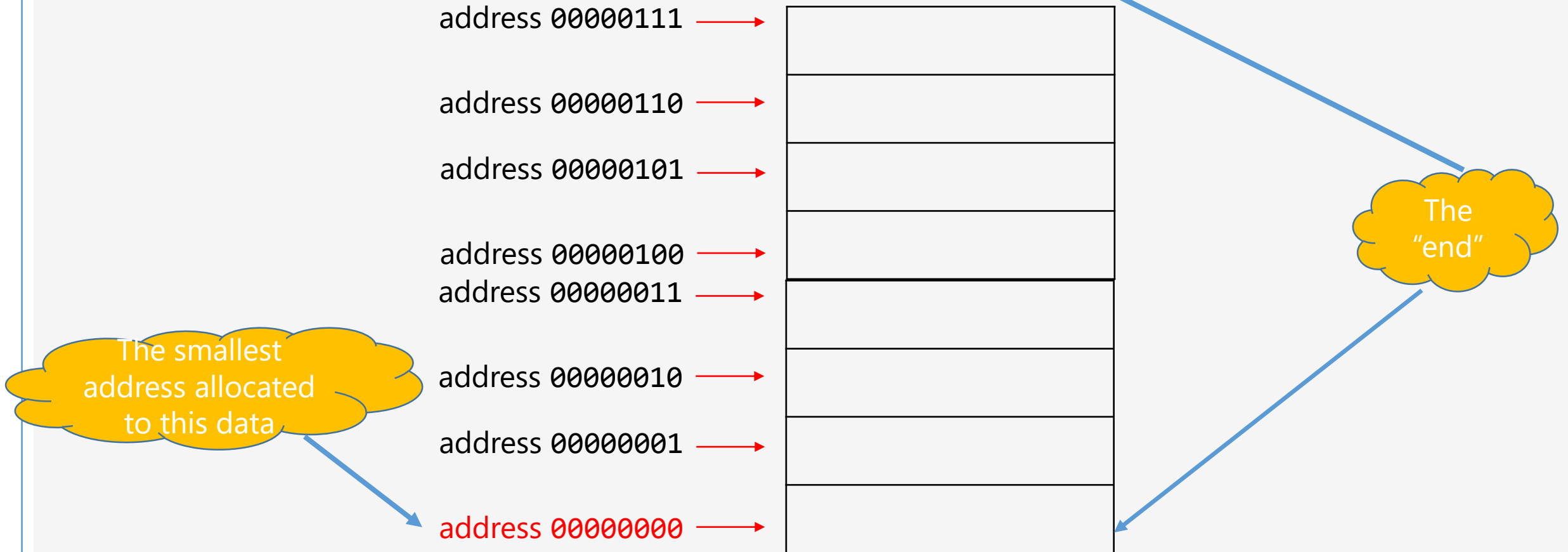


Data representation

- How is the 64 bit data `0xEAEBCEDABCD18AE` stored in the Intel processor memory?
 - First we need to identify the “end”. Here the “end” is `AE` , the “end” should be put in the memory slot with the lowest address.
 - After that it will be straight-forward to figure out where the remaining bytes are put

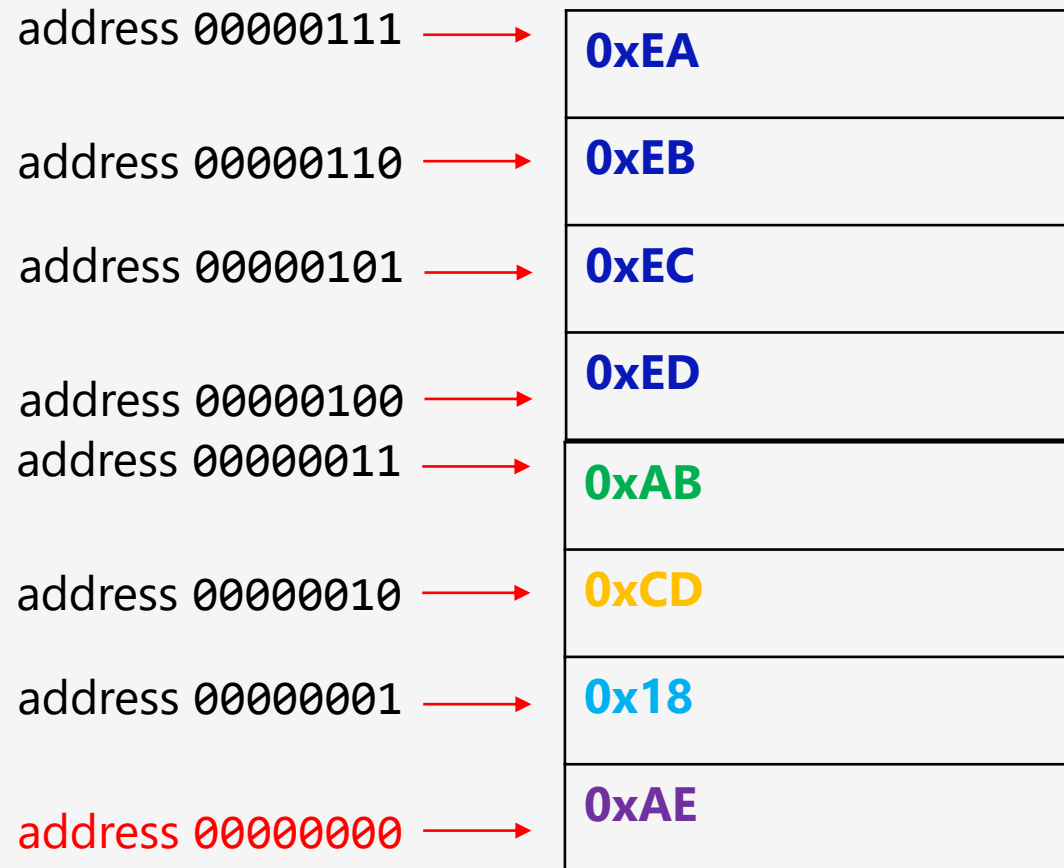
Data representation

- How is the 64 bit data `0xEAEBCEDABCD18AE` stored in the Intel processor memory (**little endian**)?



Data representation

- How is the 64 bit data **0xEAEBCEDABCD18AE** stored in the Intel processor memory (**little endian**)?



The smallest address allocated to this data

The "end"

Data representation

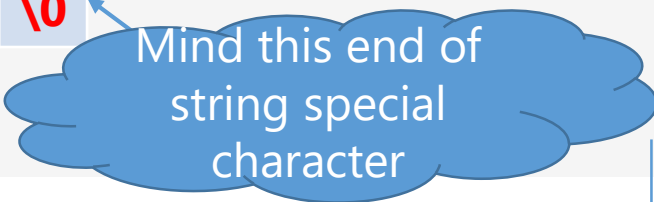
- Endianness is needed only when each piece of data to be stored is bigger than 1 byte



Data representation

- **Endianness is needed only when each piece of data to be stored is bigger than 1 byte.**
- How about the string “**Malware**” ?
- The string “**Malware**” is considered to be **8 pieces of data**
- Each piece of data is a character. The character is encoded in the ASCII scheme and the encoded character is exactly 1 byte.
- The computer will put the characters one after the other, from character number 0 at the lowest memory address, to character number 7 at the highest memory address. There is **no need to consider endianness** here (more on this in 3 slides).

Character index	0	1	2	3	4	5	6	7
	M	a	l	w	a	r	e	\0



Mind this end of string special character

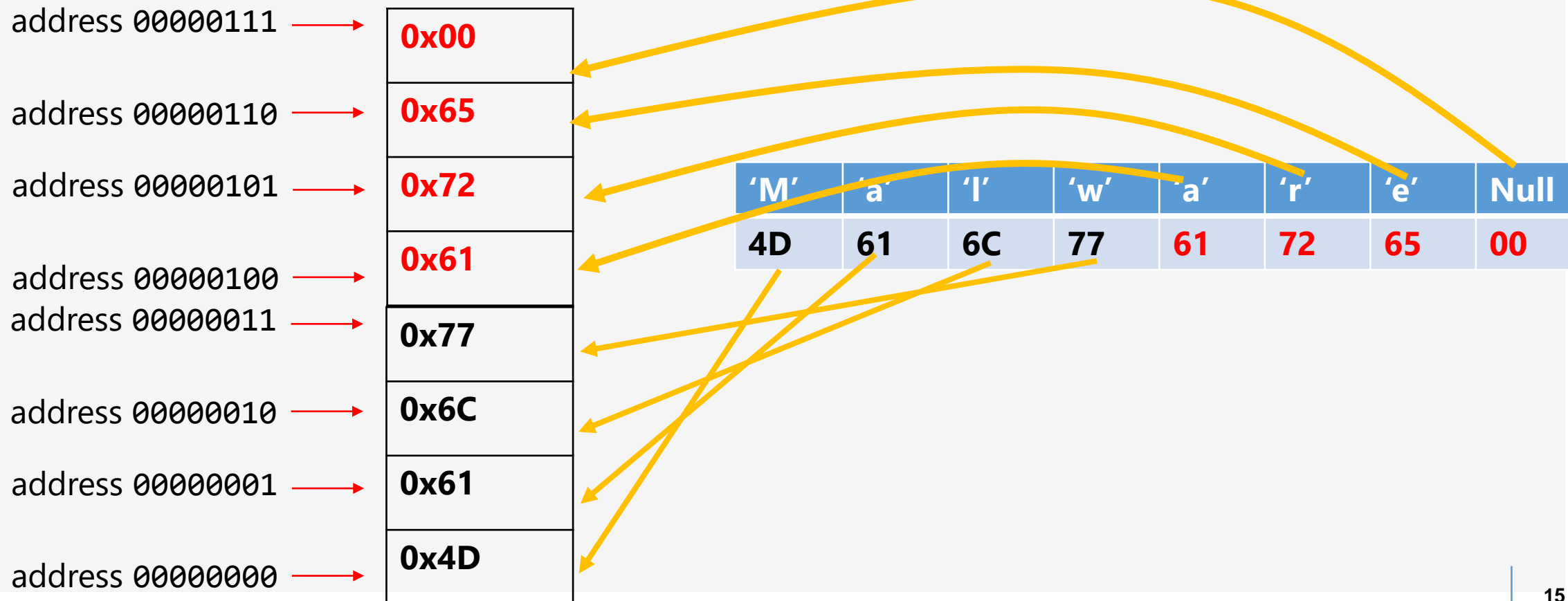
Data representation

- How is the string “Malware” stored in the memory?
- Then the characters ‘M’, ‘a’, ‘l’, ‘w’, ‘a’, ‘r’, ‘e’ are encoded into their corresponding ASCII codes using the ASCII table (2 slides later)

Character index	0	1	2	3	4	5	6	7
	4D	61	6C	77	61	72	65	00

Data representation

- The computer will put the characters one after the other, from character number 0 to character number 7 to the memory. There is **no need to consider endianness** here. Because each piece of data is exactly 1 byte.

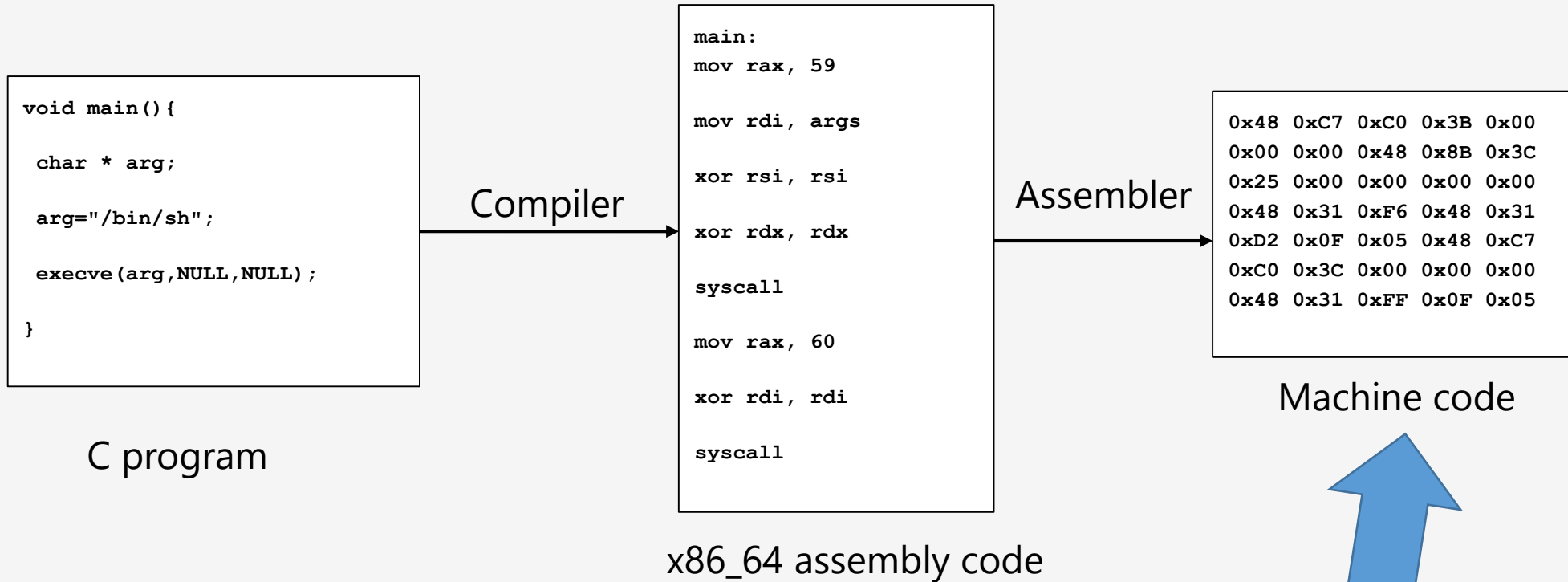


The ASCII table

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[59	3B	;	91	5B	[123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-~	63	3F	?	95	5F	_	127	7F	DEL

Running a program under x86

Running a program



This is what will be in the memory when the program runs

System software involved

- **Compiler:**
 - converts the high-level language code into the assembly code
- **Assembler:**
 - converts the assembly code into machine code (consists of 0's and 1's only)

x86 Assembly

The Intel x86 CPU Family

- 8088, 8086: 16 bit registers, real-mode only (with the x86-16 instruction set)
- 80286: 16-bit protected mode (with the x86-16 instruction set)
- 80386: 32-bit registers (with the i386 instruction set)
- 80486/Pentium/Pentium Pro/MMX/II/III/4/Xeon (with the i386/x86-64 instruction set)
- Starting from AMD Opteron in 2003: 64-bit registers and addresses (with the x86-64 or x64 instruction set)

Assembly Language

- We will be using the (x86-64) instruction set to write assembly programs
 - Unlike high-level programming language, **assembly is case insensitive** (i.e “mov eax, 5” and “mOv EAX,5” are equivalent)
- Two possible syntaxes to write assembly programs, with different ordering of the operands!
 - AT&T syntax (objdump, GNU Assembler)
 - mnemonic source, destination (`mov $5, %eax`) → left to right
 - Assign the `eax` register with the constant 5 (i.e. `eax=5`)
 - Note that AT&T syntax requires the `$` prefix for the constant and the `%` prefix for the register
 - **Intel syntax** (Microsoft Assembler, nasm, IDA Pro)
 - mnemonic destination, source (`mov eax, 5`) → right to left
 - Assign the `eax` register with 5 (i.e. `eax=5`)
 - Note that the Intel syntax, no prefixes for constant and register.
 - **We will mainly use Intel syntax in this course**

Assembly Language

- An assembly program consists of:
 - Directives: commands for the assembler
 - For example the “.data” directive tells the assembler to create all the variables under the “.data” section
 - Instructions: actual operations
- We will use nasm to “assemble” an assembly program converting it into the machine code. Machine code is the language the computer can understand.
- nasm should have been installed by default in the Kali Linux
 - Type “whereis nasm” at the prompt and you should see the path of nasm
 - To install, you just need to type “sudo apt install nasm”, make sure your machine is connected to the network when you do this

Assembly Language

- Disassembling tools (e.g., `objdump`, `gdb`, `Ghidra`) can show “almost perfect” assembly code from machine code, except in case of explicitly obfuscated code
- However, assemblers (e.g., `nasm`) can use (slightly) higher-level constructs
 - e.g., directives, named labels, (some) variable names
- Decompilers (e.g., `Ghidra`) can show “partially correctly” the original C code

X86 Registers (General Purpose)

There are 16 general purpose registers in the x86
a

Size (in Bits)			
64	32	16	8
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8~R15	R8D~R15D	R8W~R15W	R8L~R15L

There following are two important registers in the x86 architecture, they are consider “general purpose” because you can use them in any instruction, but they serve important functions and can not be used for storing arbitrary data

- **rbp**: base/frame pointer
- **rsp**: stack pointer
- And then there is the **rip special purpose register** (which is equivalent to programmer counter) that points to the instruction being executed

X86 Registers (General Purpose)

There are 16 general purpose registers in the x86
a

Size (in Bits)			
64	32	16	8
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8~R15	R8D~R15D	R8W~R15W	R8L~R15L

The registers below were originally envisioned for the following purposes, but now they **could be used for any data**

- **rax**: accumulator
- **rbx**: base index (for arrays)
- **rcx**: counter (for loops and strings)
- **rdx**: extend the precision of the accumulator
- **rdi**: destination index
- **rsi**: source index for string operations

X86 Registers (General Purpose)

<div>AL</div> <div>AH</div> <div>AX</div> <div>EAX</div> <div>RAX</div>	<div>R8B</div> <div>R8W</div> <div>R8D</div> <div>R8</div>	<div>R12B</div> <div>R12W</div> <div>R12D</div> <div>R12</div>
<div>BL</div> <div>BH</div> <div>BX</div> <div>EBX</div> <div>RBX</div>	<div>R9B</div> <div>R9W</div> <div>R9D</div> <div>R9</div>	<div>R13B</div> <div>R13W</div> <div>R13D</div> <div>R13</div>
<div>CL</div> <div>CH</div> <div>CX</div> <div>ECX</div> <div>RCX</div>	<div>R10B</div> <div>R10W</div> <div>R10D</div> <div>R10</div>	<div>R14B</div> <div>R14W</div> <div>R14D</div> <div>R14</div>
<div>DL</div> <div>DH</div> <div>DX</div> <div>EDX</div> <div>RDX</div>	<div>R11B</div> <div>R11W</div> <div>R11D</div> <div>R11</div>	<div>R15B</div> <div>R15W</div> <div>R15D</div> <div>R15</div>
<div>BPL</div> <div>BP</div> <div>EBP</div> <div>RBP</div>	<div>DIL</div> <div>DI</div> <div>EDI</div> <div>RDI</div>	<div>IP</div> <div>EIP</div> <div>RIP</div>
<div>SIL</div> <div>SI</div> <div>ESI</div> <div>RSI</div>	<div>SPL</div> <div>SP</div> <div>ESP</div> <div>RSP</div>	



8-bit register



32-bit register



16-bit register



64-bit register

CPU Registers (64-bit General Regs)

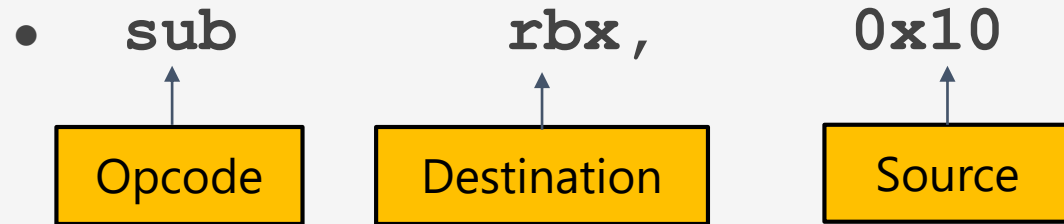
- The following extra registers are only in 64-bit mode
 - R8, R9, R10, R11, R12, R13, R14, R15
- The corresponding 32-bit registers are (lower 32 bits)
 - R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
- The corresponding 16-bit registers are (lower 16 bits)
 - R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W

CPU Special Registers (review)

- SP/ESP/RSP: Stack pointer
Decrement/increment by push/pop instructions
- BP/EBP/RBP: Stack base pointer (frame pointer)
Used to keep track of the stack pointer value when a function starts
- IP/EIP/RIP: Instruction pointer
Points to the next instruction to be executed
- SI/ESI/RSI: Typically used as source index for string operations
DI/EDI/RDI: Typically used as destination index for string operations

General x86 Instruction format

- Instructions are composed of an opcode and zero or more operands.

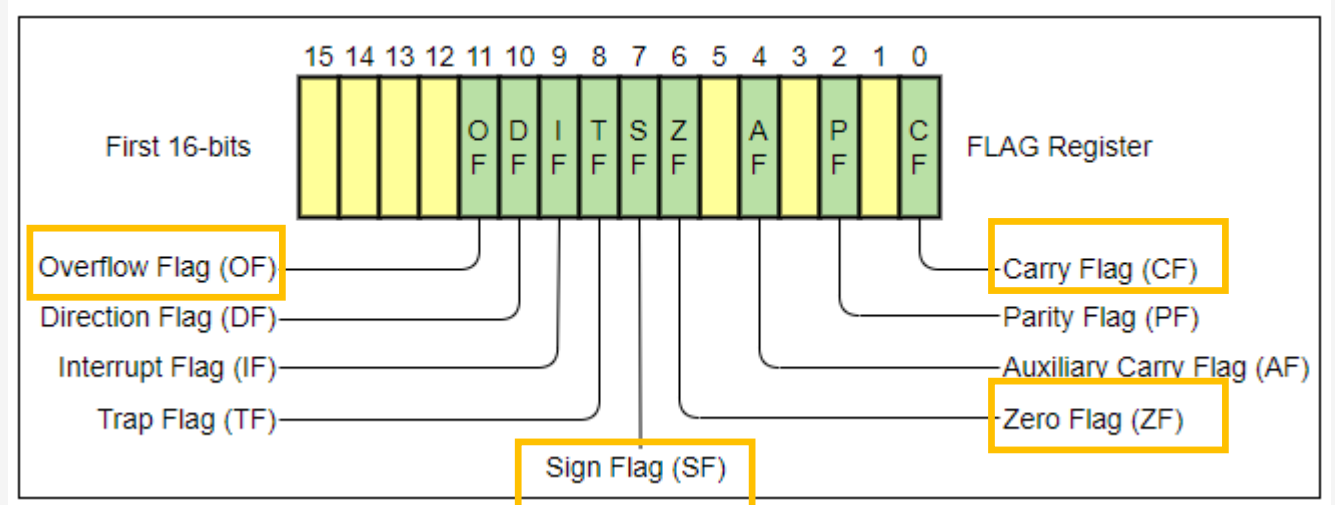


- Operation being done by the above: $\text{rbx} = \text{rbx} + 0\text{x}10$
- This is the Intel Syntax. AT&T syntax will be slightly different. We will stick with the Intel Syntax.
- This instruction uses a register and an immediate

CPU Special Registers: Flags

- Typically we only need to know the FLAGS register which is 16-bit in size as shown below:
- These flags are of interest

- CF: Carry flag
Whether arithmetic carry or borrow has been generated out of the most significant bit. If a carry or borrow has been generated out of the most significant bit, this flag will be 1. Otherwise it will be 0.
- ZF: Zero flag
Whether an arithmetic result is zero



The flags are automatically set by mov, cmp, and other instructions, used to determine jumps

CPU Special Registers: Flags

- Typically we only need to know the FLAGS register which is 16-bit in size as shown below:
- These flags are of interest

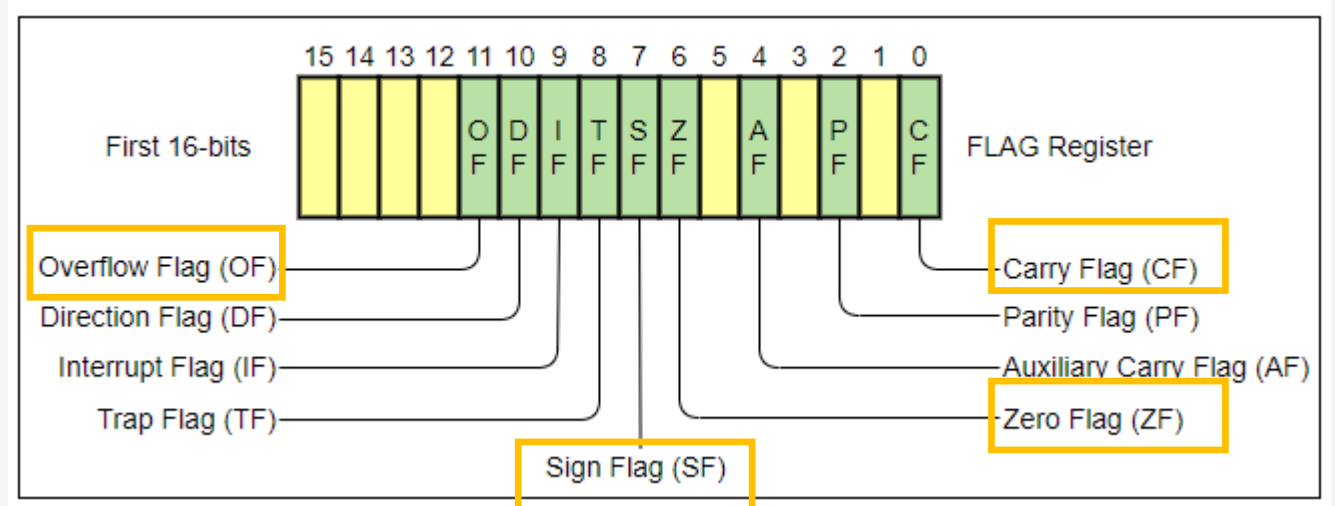
- SF: Sign flag

Whether the result of the last mathematical operation produced a value in which the most significant bit is 1 (indicating the result is negative).

- OF: Overflow flag

Detects only signed overflow:

- 1) positive + positive but result is negative
- 2) negative + negative but result is positive



The flags are automatically set by mov, cmp, and other instructions, used to determine jumps

CPU Special Registers: Flags

- A simple example to illustrate the flags:

```
mov dx, 0xff      ; mind that dx a 16-bit register
add dx, 1          ; dx=0x100 SF=0 ZF=0 CF=0
sub dx, 1          ; dx=0xff SF=0 ZF=0 CF=0
add dl, 1          ; mind that dl is a 8-bit register
                  ; dl=0x00 SF=0 ZF=1 CF=1

sub dl, 2          ; dl=0xfe SF=1 ZF=0 CF=0
mov dl, 3          ; dl=0x03 SF=0 ZF=0 CF=0
mov cl, 3          ; cl=0x03 SF=0 ZF=0 CF=0
mov bl, 2
cmp dl, cl         ; cmp performs "dl - cl" = 0x3-0x3=0
                  ; dl=0x03 cl=0x03 SF=0 ZF=1 CF=0
cmp bl, cl         ; cmp performs "bl - cl" = 0x2-0x3 = -1 = 0xFF
                  ; bl=0x02 cl=0x03 SF=1 ZF=0 CF=1
```

CPU Special Registers: Flags

- In general for the compare (`cmp`) instruction

`cmp dst, src:`

If `dst == src`, then the flags will be **ZF = 1, CF = 0**

If `dst < src`, then the flags will be **ZF = 0, CF = 1**

If `dst > src`, then the flags will be **ZF = 0, CF = 0**

Jump instruction can test ZF and CF to decide whether to jump to another place or to keep execute the instructions sequentially.

General x86 Instruction format

- Instructions are composed of an opcode and zero or more operands.
- `mov dword ptr [eax+ecx*4+0x20], edx`



- Operation being done by the above: `mem[eax+ecx*4+0x20] = edx`
 - `mem[x]` means the memory slot at address **x**
 - here it will copy 32bits/4bytes (dword) data from address x to edx
- This is the Intel Syntax. AT&T syntax will be slightly different. We will stick with the Intel Syntax.
- The `mov` instruction here is a memory reference instruction

The “mov” instruction

`mov eax, 5` → copies 5 into eax

`mov eax, ebx` → **copies** value of ebx into eax

`mov eax, dword ptr [ebp - 8]` → copies the contents of the memory pointed by `ebp - 8` into eax
the **ptr** keyword here indicates we are pointing to memory
dword indicates we are copying a dword (32-bit data size)

`mov eax, dword ptr [eax]` → copies the contents of the memory pointed by eax to eax

`mov dword ptr [edx + ecx * 2], eax` → moves the contents of eax into the memory
at address `edx + ecx * 2`

`mov ebx, 804a0e4h` → copies the value 804a0e4h into ebx

`mov eax, dword ptr [804a0e4h]` → copies a dword from address 804a0e4h into eax

1. No 2 memory accesses in the same instruction! (i.e. you can not read a data from memory and then copy it also to the memory)

2. Width: size of reference (byte, word, dword, qword → 8, 16, 32, 64 bits)

The “*lea*” instruction

- **lea** is known as the “load effective address” instruction
- It loads/copies an address calculated into a register
- For example “**lea rax, [rip+0x2ec6]**”
 - It is equivalent to the operation `rax = value_of(rip) + 0x2ec6`
 - `rip` is a 64-bit register here
 - Suppose `rip` is holding `0x0000787FFFFFFF0000`, then the instructions puts `0x0000787FFFFFFF0000 + 0x2ec6 = 0x0000787FFFFFF2EC6` into `rax`

The “push” instruction

- The **push** instruction “pushes/stores” a piece of data to the stack and decreases the stack pointer
- **push <register>** → decreases the stack pointer (esp/rsp) and saves the content of **<register>** in the newly pointed location
- For example “**push rax**”
 - Will decrease the stack pointer rsp by 8 to allocate 8 bytes of new space on the stack
 - Then it will put the 8-byte rax register value into the newly allocated 8-byte space on the stack
 - more on this with the help of a picture
 - “**push eax**” is the 32-bit version, it will decrease esp by 4 and put the 4-byte eax content to the stack

The “pop” instruction

- The **pop** instruction “pops/retrieves” a piece of data from the stack and increases the stack pointer
- **pop <register>** → retrieves the last piece of data (i.e. top) from the stack and stores it to **<register>**, then it increases the stack pointer (`esp/rsp`) to de-locate the data from the stack (i.e. the data will be out of the stack boundary)
- For example “**pop rax**”
 - Will copy 8-byte of data on the top of the stack to the 64-bit register
 - Then it will increase the stack pointer `rsp` by 8 to de-locate 8 bytes of space from the stack
 - more on this with the help of a picture
 - “**pop eax**” is the 32-bit version, it will copy the 4-byte data from the top of stack to `eax` and then increase `esp` by 4 to de-locate the data

The “call” instruction

- The **call** is a special instruction for making function calls
- **call funct** → stores the return address (address immediately after the `call` instruction itself) and jumps to `funct` to run it (i.e calls the ‘`funct`’ function)
- For example “**call factorial**”
 - ❑ stores the address of “instruction 4” to the Stack and then the program jumps to **factorial**
 - ❑ the instruction pointer (rip/eip) will be copied the address of instruction x by the `call` instruction, the computer executes the instruction by referring to rip.

main:

instruction 1

instruction 2

call factorial

instruction 4

: : :

factorial:

instruction x

instruction x + 1

ret

The “ret” instruction

- The **ret** is a special instruction for finishing a function call and returning back to the caller
- **ret** → pops the return address stored in stack and returns back to the caller
- For example “**ret**”
for the sample program at the right:
 - Will pop the **stored address** (of instruction 4) from the stack
 - and then the program will **jump** back to resume executing the main function at instruction 4 (i.e. function call returned)

main:

instruction 1
instruction 2
call factorial
instruction 4
: : :

factorial:

instruction x
instruction x + 1

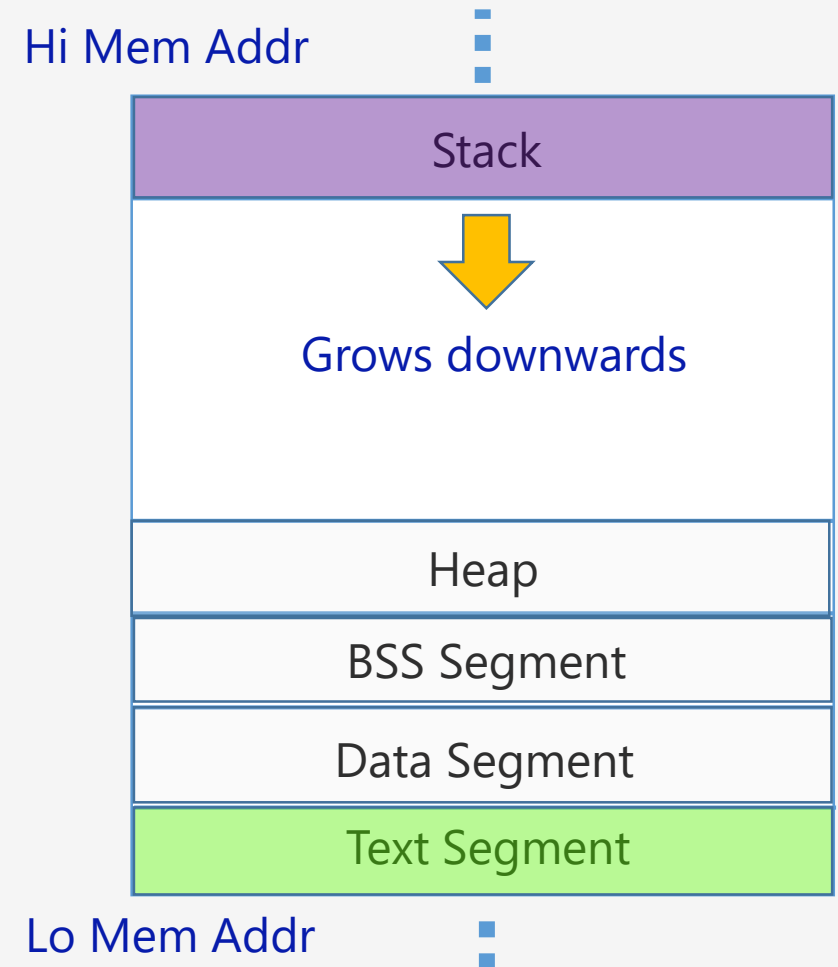
ret



x86 Memory Layout

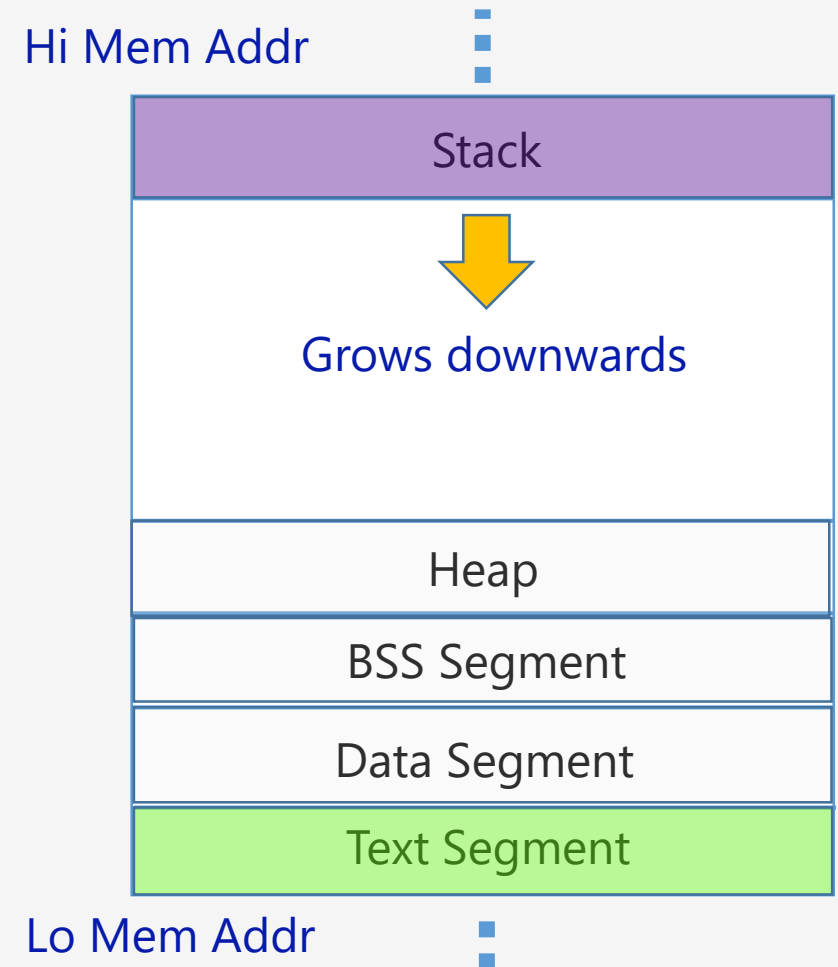
The Memory Model

- Important:
 - each storage unit in the memory is a byte (8-bit)
 - Each such a storage unit will have a distinct 32-bit or 64-bit number to indicate the address
 - Think of it as a street number



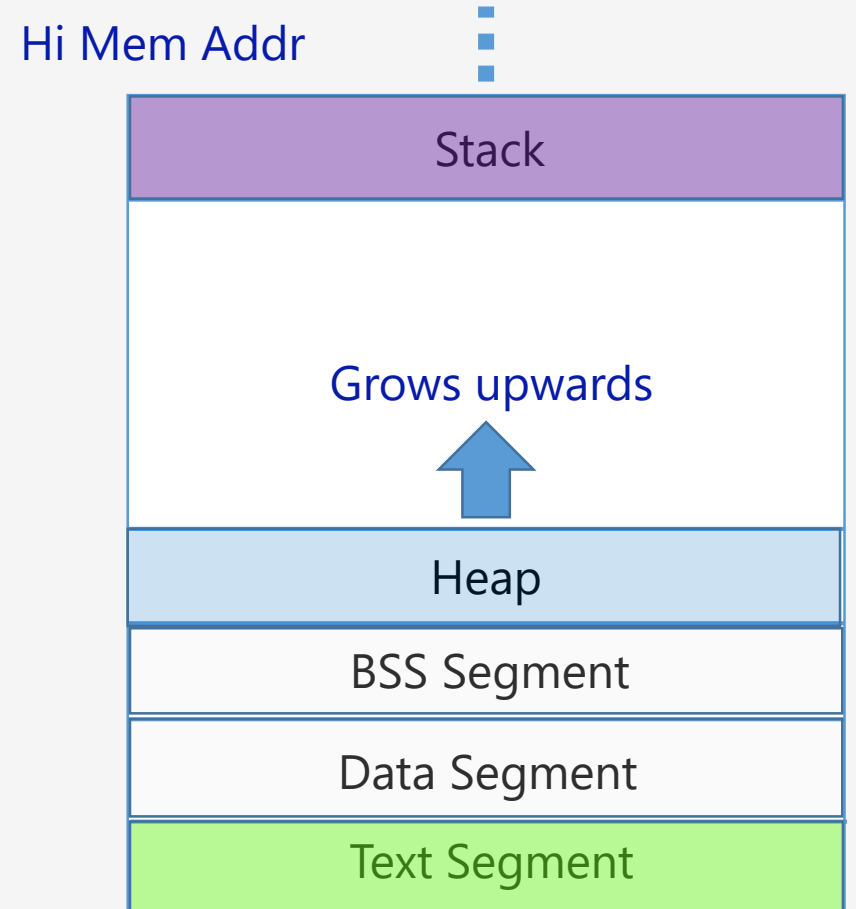
The Memory Model

- The Stack
 - Where function calls and function related data are stored
 - Local variables of function
 - Backup of base pointer **rbp**
 - Backup of return address after the function call **rip**



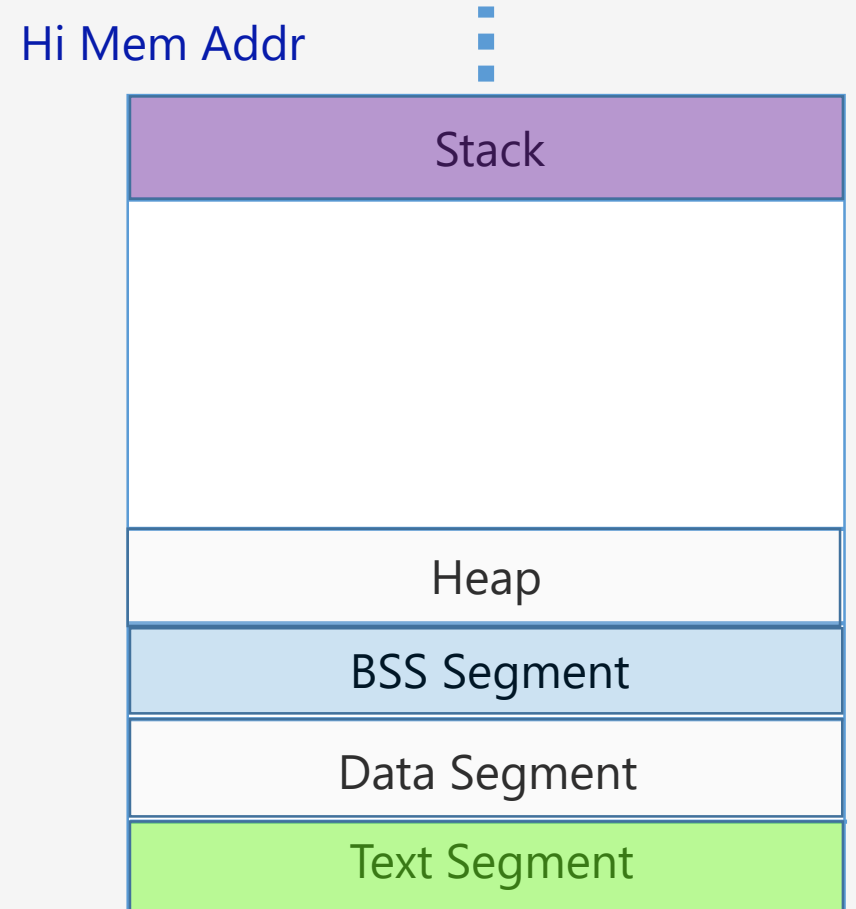
The Memory Model

- The Heap
 - Memory allocated at runtime
 - Objects created using “**new**”, memory slots created by **malloc()** or **calloc()**



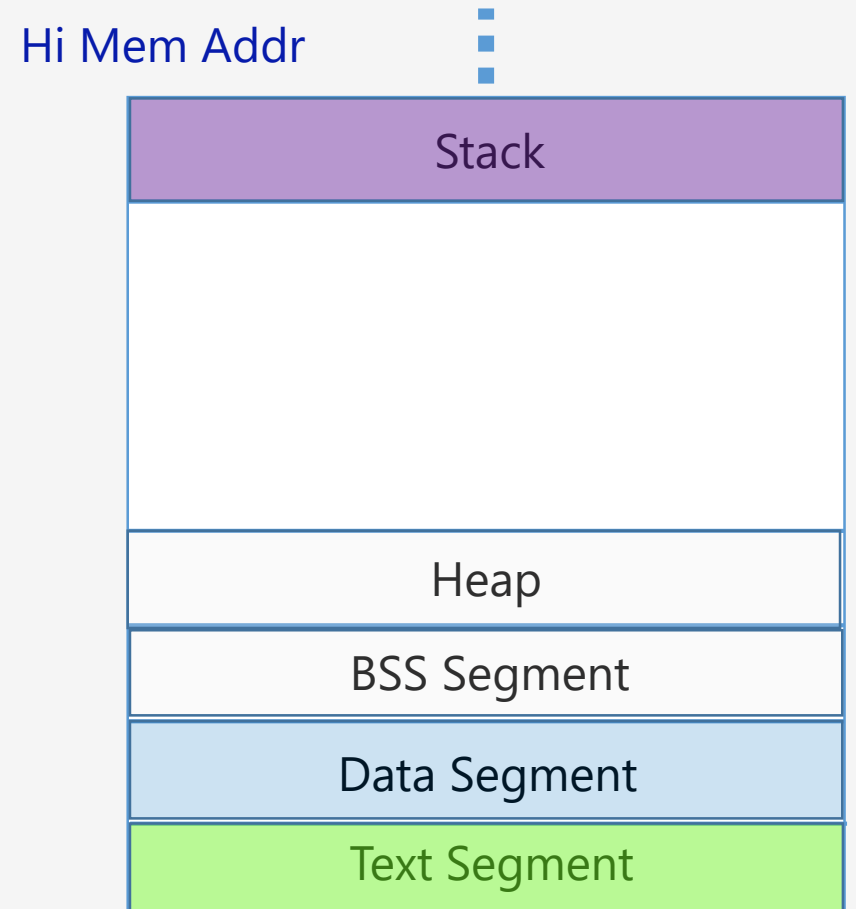
The Memory Model

- The BSS Segment
 - **uninitialized** global and static variables



The Memory Model

- The Data Segment
 - **initialized** global/static variables



The Memory Model

- The Text Segment
 - The place where the program (**instructions**) is located

