Assembly Recap

Paolo Falcarin

Ca' Foscari University of Venice

Department of Enviromental Sciences, Informatics and Statistics

paolo.falcarin@unive.it

CM0626 – Software Security

CM0631-2 – Software Security

18 February 2025

Assembly Programming

- Topics
  - Assembly Programmer's Execution Model
  - Accessing Information
    - Registers
    - Memory
  - Arithmetic operations

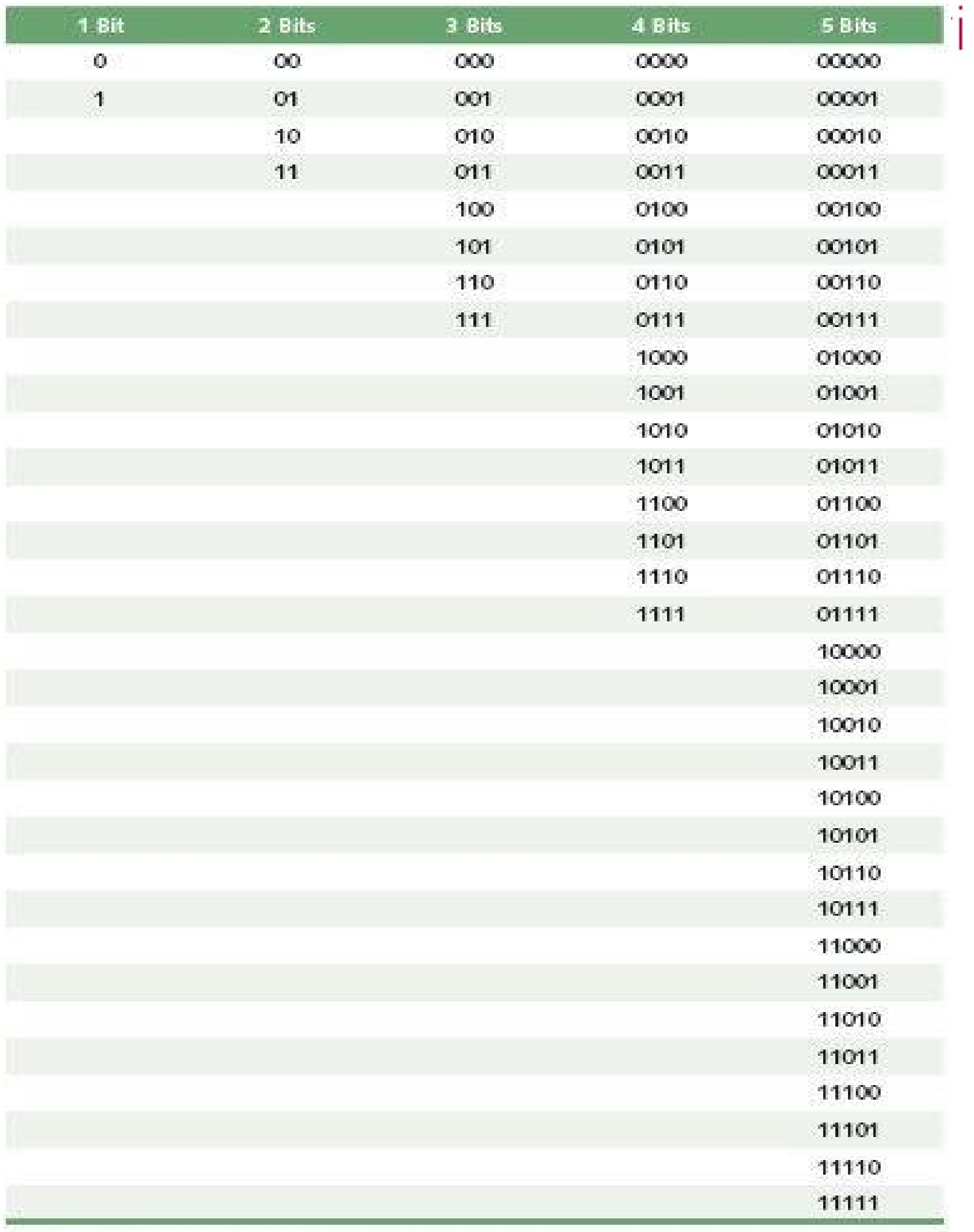
Binary representation

- *How many things can* n bits *represent?*

- With 1 bits: 2^1 things

- By doubling the bits, that is 2 bits: 2^2 things

In general, with n bits we can thus represent: 2^n things



1 Bit	2 Bits	3 Bits	4 Bits	5 Bits
0	00	000	0000	00000
1	01	001	0001	00001
	10	010	0010	00010
	11	011	0011	00011
		100	0100	00100
		101	0101	00101
		110	0110	00110
		111	0111	00111
			1000	01000
			1001	01001
			1010	01010
			1011	01011
			1100	01100
			1101	01101
			1110	01110
			1111	01111
				10000
				10001
				10010
				10011
				10100
				10101
				10110
				10111
				11000
				11001
				11010
				11011
				11100
				11101
				11110
				11111

FIGURE 3.4 Bit combinations

Representing text

Fortunately, the number of characters to represent is finite (whew!), so list them all and assign each a binary string

Character set

A list of characters and the codes used to represent each one.

Computer manufacturers agreed to standardize

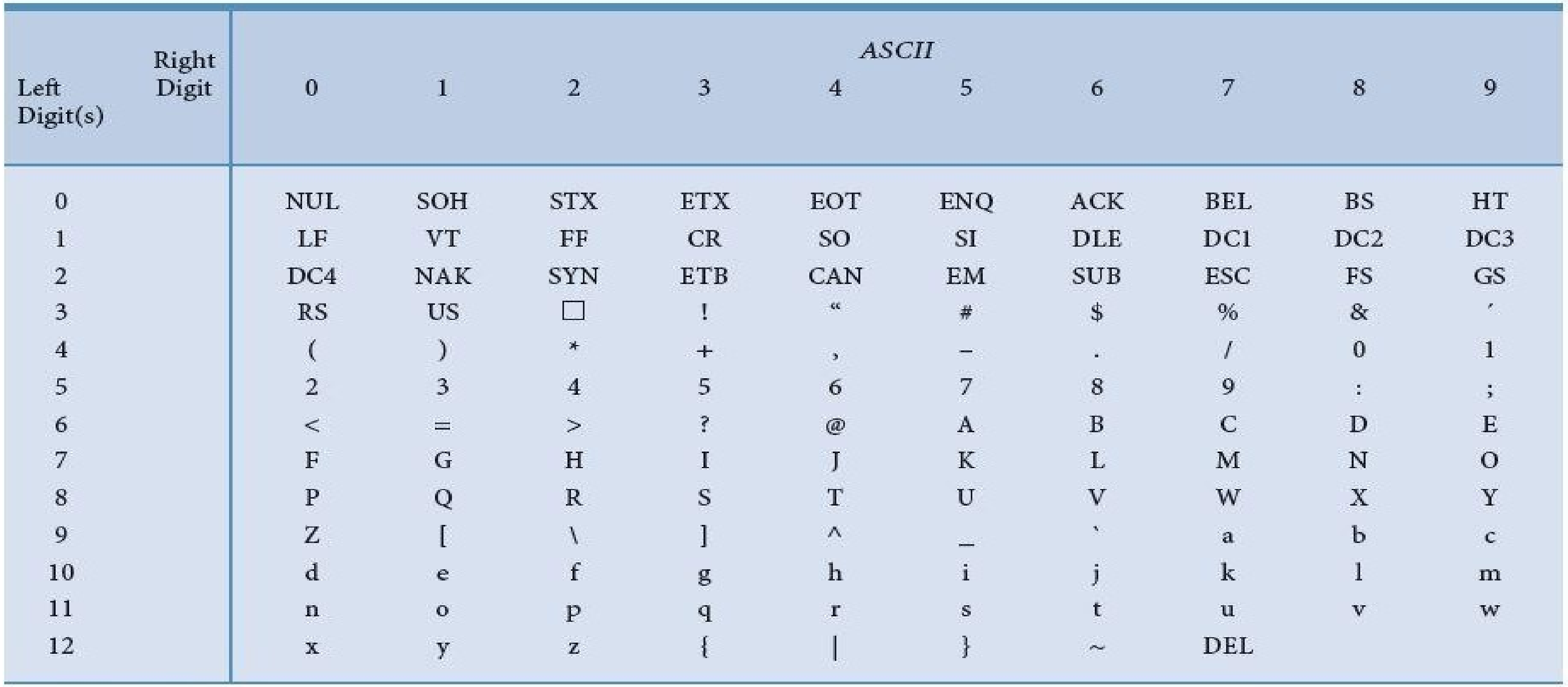
Example: ASCII character set

American Standard Code for Information Interchange

7 bits version allows 128 unique characters

See **UTF-8** as more modern standard character encoding

ASCII Character Set Mapping

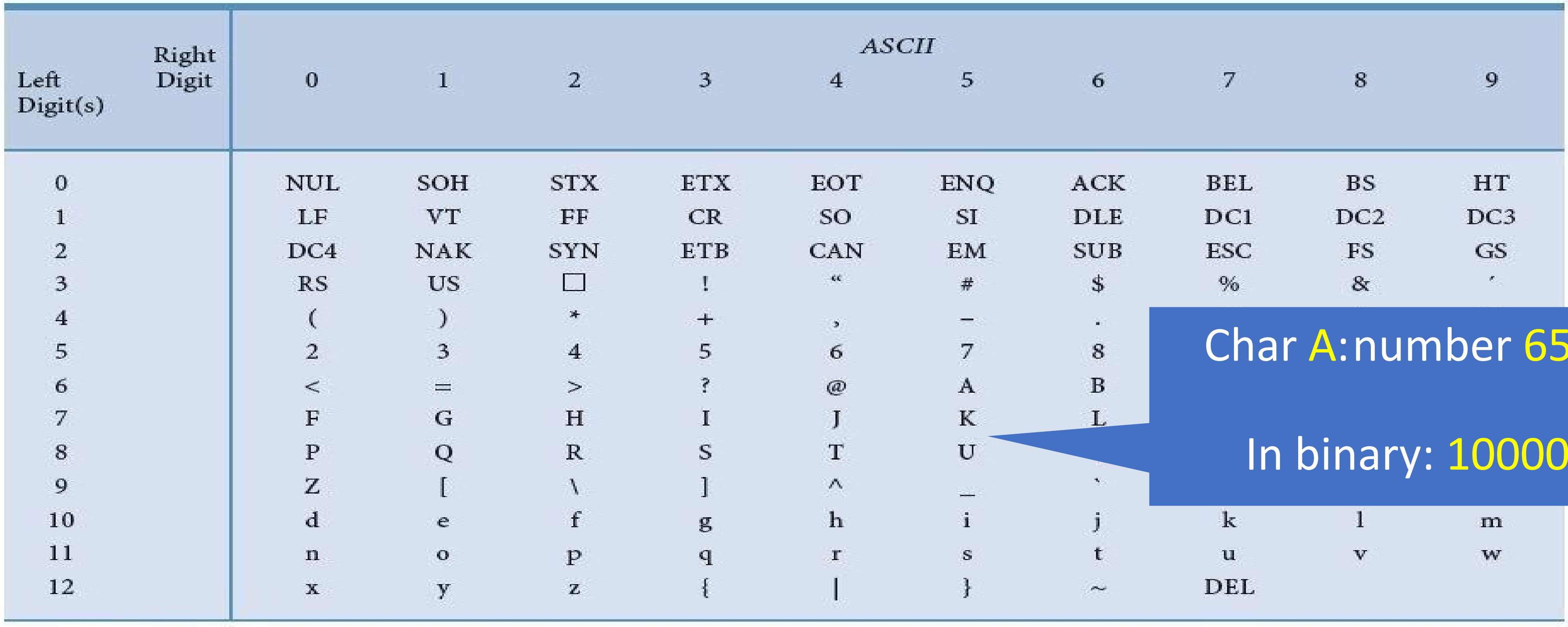


Right Digit	ASCII									
Left Digit(s)	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

FIGURE 3.5 The ASCII character set

ASCII Character Set Mapping

Ca' Foscari University of Venice



Left Digit(s) \ Right Digit	0	1	2	3	4	5	6	7	8	9
					ASCII					
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	"	#	$	%	&	'
4	(	)	*	+	,	−	.			
5	2	3	4	5	6	7	8			
6	<	=	>	?	@	A	B			
7	F	G	H	I	J	K	L			
8	P	Q	R	S	T	U				
9	Z	[	\	]	^	_	`			
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

FIGURE 3.5 The ASCII character set

Char A: number 65 = 2^6

In binary: 1000001

Unicode Character Set

- Unicode Standard is an information technology standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.
- The standard, which is maintained by the Unicode Consortium, defines 144,697 characters covering 159 modern and historic scripts, as well as symbols, emoji, and non-visual control and formatting codes.

Positional notation with a generic base B

- **Value of a number** represented in a **generic base B** with **symbols used for digits representing quantities: 0,1, …., B-1**

B is the base of the number

$$d_{n-1} * B^{n-1} + d_{n-1} * B^{n-2} + \dots + d_1 * B^1 + d_0 * B^0$$

n is the number of digits in the number

d_i is the digit in the ith position in the number

642 (**$d_2d_1d_0$** **expressed in base B=10**) is equal to:

$$6 * 10^2 + 4 * 10^1 + 2 * 10^0 = 600 + 40 + 2$$

Binary integer numbers

- Decimal numbers have base 10 and need 10 digit symbols:

  0,1,2,3,4,5,6,7,8,9

- Exadecimal numbers have base 16 and need 16 digit symbols:

  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Binary numbers have base 2 and only need 2 digit symbols:

0,1

Converting from binary to decimal

Ca' Foscari University of Venice

What is the decimal equivalent N_{10} *of the binary number* N_2 *=1101101?*

$$
\begin{array}{rlll}
 & 1 \times 2^6 = 1 \times 64 & = 64 & (2^6 = \text{64-lets}) \\
+ & 1 \times 2^5 = 1 \times 32 & = 32 & (2^5 = \text{32-lets}) \\
+ & 0 \times 2^4 = 0 \times 16 & = 0 & (2^4 = \text{16-lets}) \\
+ & 1 \times 2^3 = 1 \times 8 & = 8 & (2^3 = \text{8-lets}) \\
+ & 1 \times 2^2 = 1 \times 4 & = 4 & (2^2 = \text{quadruplet}) \\
+ & 0 \times 2^1 = 0 \times 2 & = 0 & (2^1 = \text{pairs}) \\
+ & 1 \times 2^0 = 1 \times 1 & = 1 & (2^0 = \text{units}) \\
\hline
 & \textbf{Sum} & \textbf{= 109} & \text{expressed in base 10}
\end{array}
$$

Apply the formula: $d_{n-1} * B^{n-1} + d_{n-1} * B^{n-2} + \ldots + d_1 * B^1 + d_0 * B^0$

Converting from binary to hex

Ca' Foscari University of Venice

- *What is the hexadecimal equivalent* N_{16} *of the binary number* N_2 = *010 0110 1101?*
- Split the binary string in groups of 4 digits, starting from the right
- Each 4 digit group can represent numbers from 0 to 15 (0 to F in hex)
- N_2 = *010 0110 1101* = *2 6 13* = *26D in base 16*
- Hex numbers are usually represented with the 0x prefix: 0x26D

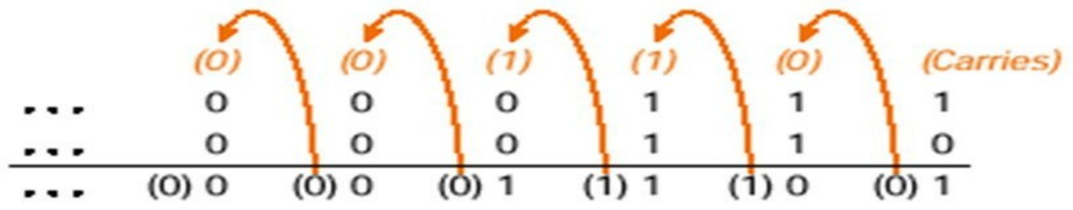
Formula with B=16: $d_{n-1} * B^{n-1} + d_{n-1} * B^{n-2} + \ldots + d_1 * B^1 + d_0 * B^0$

Sum = $2x16^2$ + 6X16 + 13 = 512 + 96 + 13 = 621 expressed in base 10

Binary Arithmetic

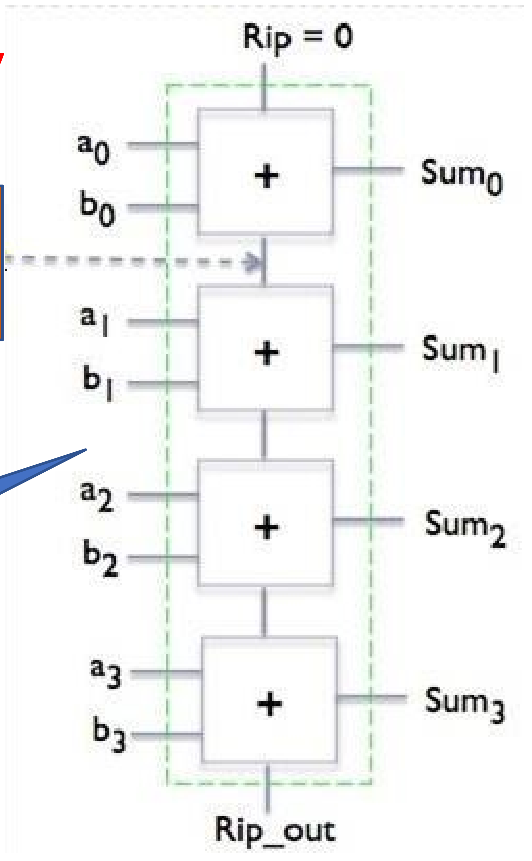
- Remember that there are only 2 digit symbols in binary, 0 and 1. Therefore 1 + 1 = 0 with a *carry*
- *This is analogous to decimal arithmetic:* 5 + 5 = 0 with a *carry*

Example of 7 (111) + 6 (110) = 13 (1101):



Carry propagation

Arithmetic Logic Unit (ALU) made of Single-bit Adders



Real numbers to binary

- Since any real number can be written in binary as *$\pm 1.????...?? \times 2^N$* (binary scientific notation), computers use a *discrete representation* inspired to it

- **Single precision** (32 bits) **Floating-point representation**:
  - 1 bit for the sign (negative vs. positive)
  - 8 bits for the exponent *N* of 2^N
  - 23 bits for the mantissa *????..??*

- **Double precision** uses 64 bits, and it is more accurate (but expensive)

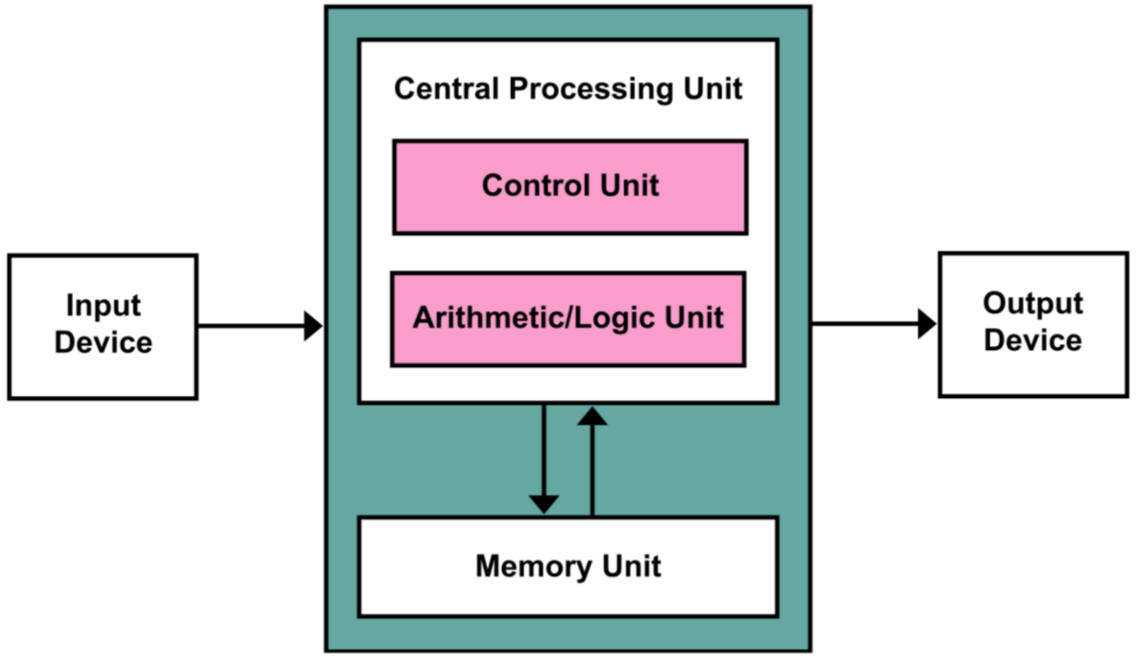
- In this way, we increase the real numbers we can represent by using a finite number of bits

Different Binary Units

UNIT	VALUE	STORAGE
Bit	1 or 0	
Byte (B)	8 Bits	Character
Kilobyte (KB)	1024 Bytes = 2^{10} Bytes	Half page of text
Megabyte (MB)	1024 Kilobytes = 2^{20} Bytes	About 2 mins MP3 file
Gigabyte (GB)	1024 Megabytes = 2^{30} Bytes	About one hour Movie
Terabyte (TB)	1024 Gigabytes = 2^{40} Bytes	128 DVD Movies
Petabyte (PB)	1024 Terabyte = 2^{50} Bytes	7 billion Facebook photos
Exabyte (EB)	1024 Petabyte = 2^{60} Bytes	50,000 years of DVD
Zettabyte (ZB)	1024 Exabyte = 2^{70} Bytes	Global internet traffic per year

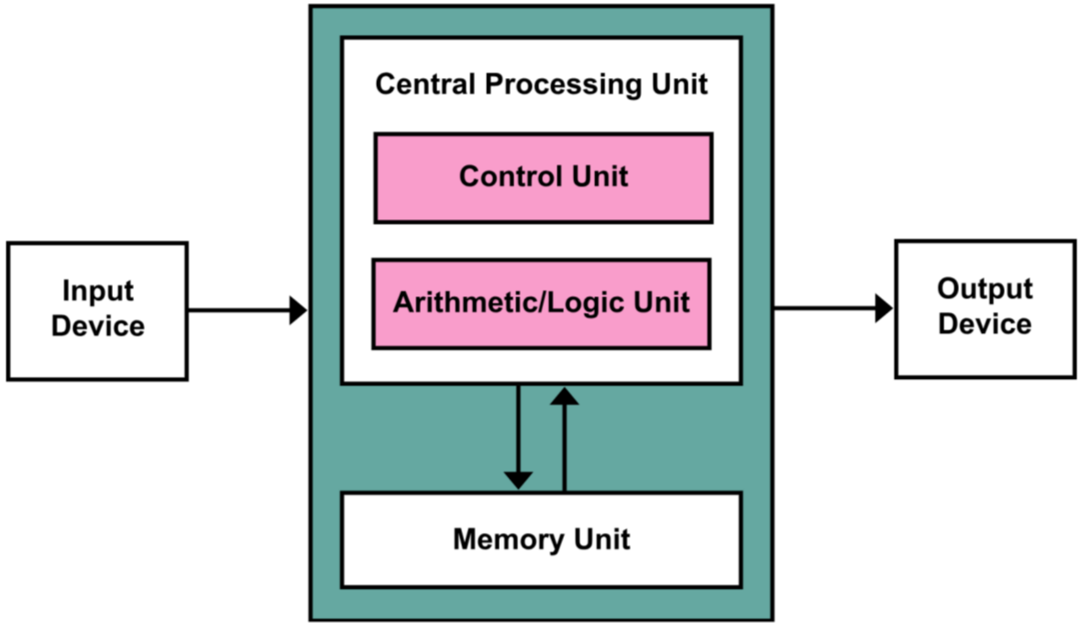
Von Neumann architecture of a computer

- Architecture composed of
  - CPU: Central processing unit
    - ALU to perform arithmetic/logic operations
  - Memory to store
    - Programs (stored-program)
    - Data accessed by program
  - Input/Output (I/O) devices
    - including disks and SSD

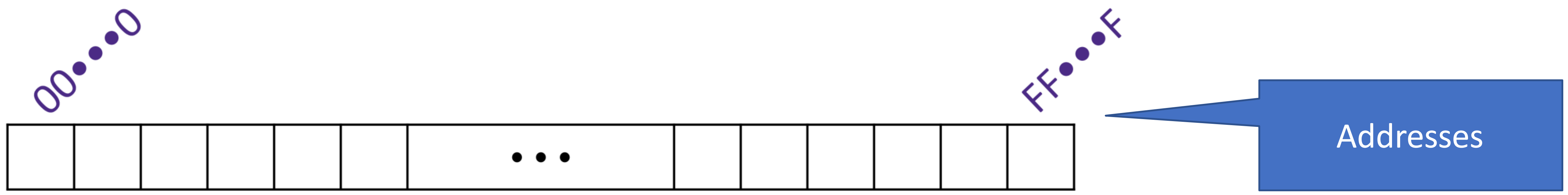


Von Neumann architecture of a computer

- Flexibility of Von Neumann architecture
  - Memory contains just numbers.
  - The meaning of those numbers depends on the context
    - Data used by the program
    - Program instructions itself



Memory organization



- Conceptually, memory is a single, large vector of bytes (8 bits), each with a unique address
- The value of each byte in memory can be read and written
- Programs refer to bytes in memory by their addresses
- But not all values fit in a single byte… for example the maximum unsigned integer we can store in a byte is:
  - 255_{10} (11111111_2)

Machine languages - between HW and SW layers

- CPU are designed to recognize **machine instructions** stored in memory as sequences of bits
- The (short) list of instructions with their binary encoding is called **machine languages**
- Example of machine language
  - ADD A,B,C where A,B, and C are operands, and the semantics is: $A \leftarrow B+C$
  - Encoding on 4 Bytes, where A, B and C are fast memory registers in the CPU

OPCODE	**Target operand**	**Source oper. 1**	**Source oper. 2**
Encoding ADD operation	Encoding reg. A	Encoding reg. B	Encoding reg. C
17 bits	**5 bits**	**5 bits**	**5 bits**

Program execution

- The control flow is the order in which instructions are fetched/executed and it corresponds to the sequential order in which instruction are stored in memory

- The control flow of the CPU is guided by an *address in memory*
    - The Program Counter (**PC**) maintains for a running program the address of the next machine instruction to execute
    - The program counter is also called Registry Instruction Pointer (RIP)

Program execution

- The CPU continuously **repeats** the following 3 steps:
    1. **Fetch** from memory the next instructions pointed by the PC
    2. **Decode** the instruction
    3. **Execute** the instruction, and *update* the PC to the next instruction to fetch

- The sequential **control flow can be modified** by the program, executing special instruction of JUMP/BRANCH.

Intel x86 (IA32/64) Processors

- Totally Dominate Computer Market

- Evolutionary Design
  - Starting in 1978 with 8086 (really 1971 with 4004)
  - Added more features as time went on
  - Still support old features, although obsolete
- Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But Intel has done just that!
    - Well…in terms of speed; less so for low power

X86 Evolution: Milestones

Name	Date	Transistors	Frequency
• 4004	1971	2.3K	108 KHz

- 4-bit processor. First 1-chip microprocessor
- Didn't even have interrupts!

Name	Date	Transistors	Frequency
• 8008	1972	3.3K	200-800 KHz

- Like 4004, but with 8-bit ALU

Name	Date	Transistors	Frequency
• 8080	1974	6K	2 MHz

- Compatible at source level with 8008
- Processor in first "kit" computers
- Pricing caused it to beat similar processors with better programming models
  - Motorola 6800
  - MOS Technologies (MOSTEK) 6502

X86 Evolution: Milestones

Name	Date	Transistors	Frequency
• 8086	1978	29K	5-10 MHz

- 16-bit processor. Basis for IBM PC & DOS
- Limited to 1MB address space. DOS only gives you 640K

Name	Date	Transistors	Frequency
• 80286	1982	134K	4-12 MHz

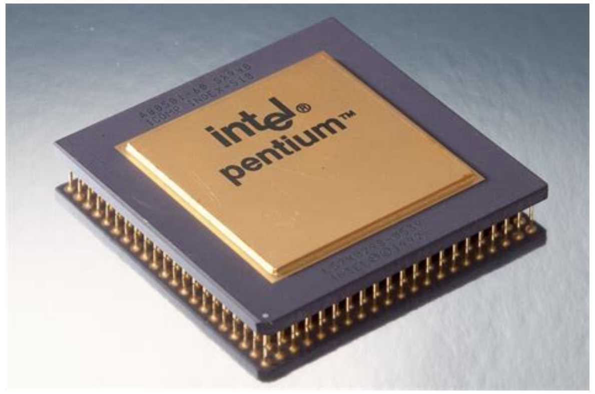
- Added elaborate, but not very useful, addressing scheme
- Basis for IBM PC-AT and Windows

Name	Date	Transistors	Frequency
• 386	1985	275K	16-33 MHz

- Extended to 32 bits. Added “flat addressing”
- Capable of running Unix
- By default, Linux/gcc compiling for 32-bit x86 machines use no instructions introduced in later models

X86 Evolution: Milestones

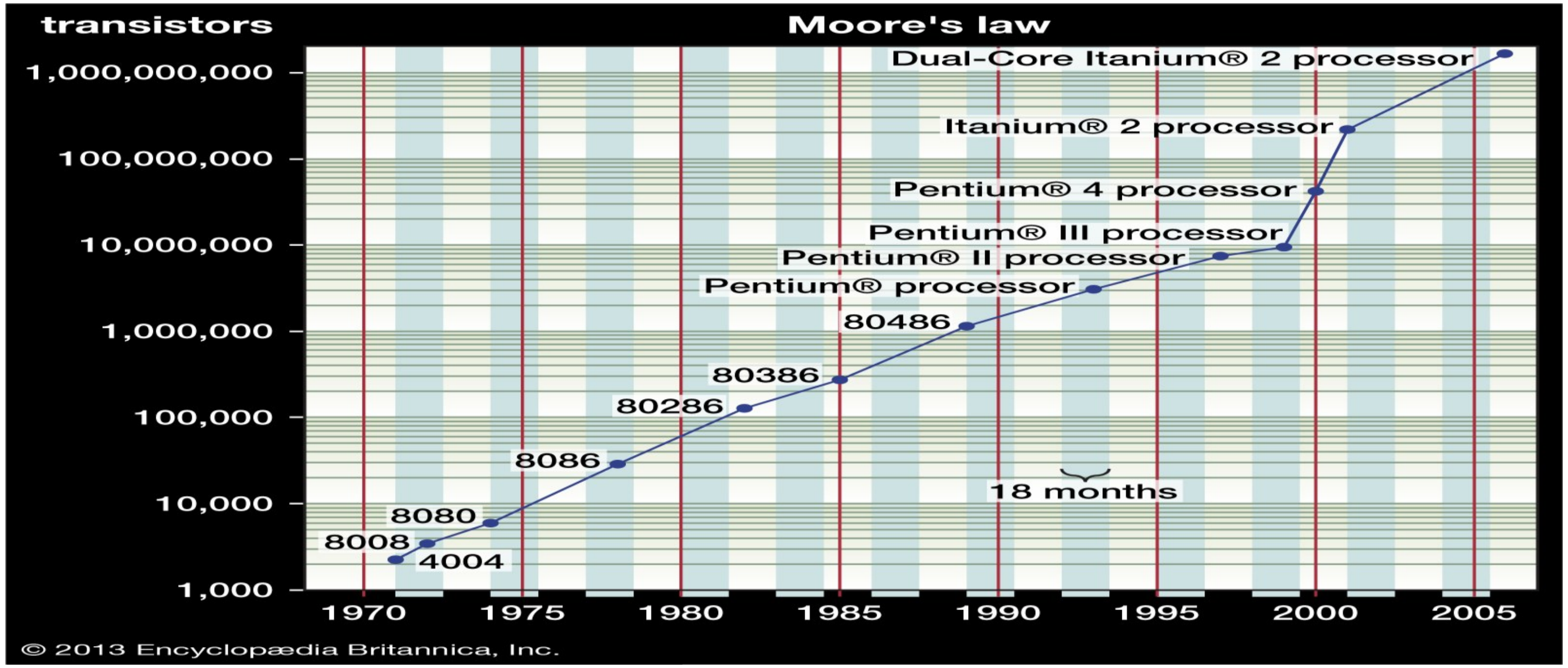
Name	Date	Transistors	Frequency
• 486	1989	1.9M	16-150 MHz
• Pentium P5	1993	3.1M	60-66 MHz
• Pentium 4E	2004	125M	2.8-3.8 GHz
• First 64-bit Intel x86 processor			
• Core 2	2006	291M	1.0-3.5 GHz
• First multi-core Intel processor			
• Core i7	2008	731M	1.7-3.9 GHz
• Ivy Bridge	2012	0.6-4.3B	3.2-4.0 GHz
• Core i9	2017	Billions	2.6 GHz
• 18 cores, 24.75 MB L3 cache.			



Moore’s Law

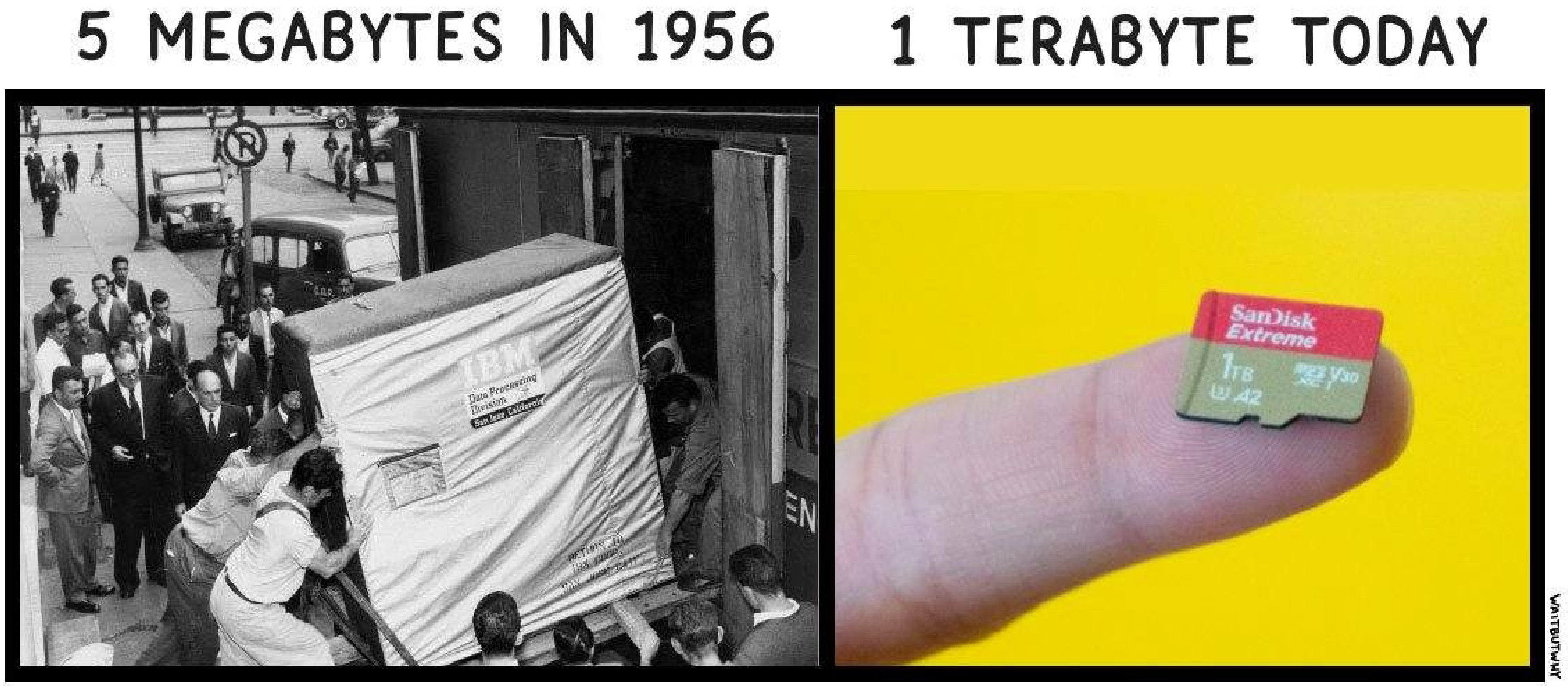
Ca’ Foscari University of Venice

Gordon E. Moore observed that the number of transistors on a computer chip was doubling about every 18–24 months. As shown in the logarithmic graph of the number of transistors on Intel's processors at the time of their introduction, his “law” was being obeyed.



Hardware in last decades

- Hardware in the last decades got bigger capacity and smaller size



X86 Evolution: Clones

- Advanced Micro Devices (AMD)
  - Historically
    - AMD has followed just behind Intel
    - A little bit slower, a lot cheaper
  - Late 1990s
    - Recruited top circuit designers from Digital Equipment Corp.
    - Exploited fact that Intel distracted by Itanium
    - Became close competitors to Intel
  - Developed own extension to 64 bits (called x86_64)
  - Intel adopted in early 2000's after Itanium
    - Has recovered lead in semiconductor technology
    - AMD has fallen behind again

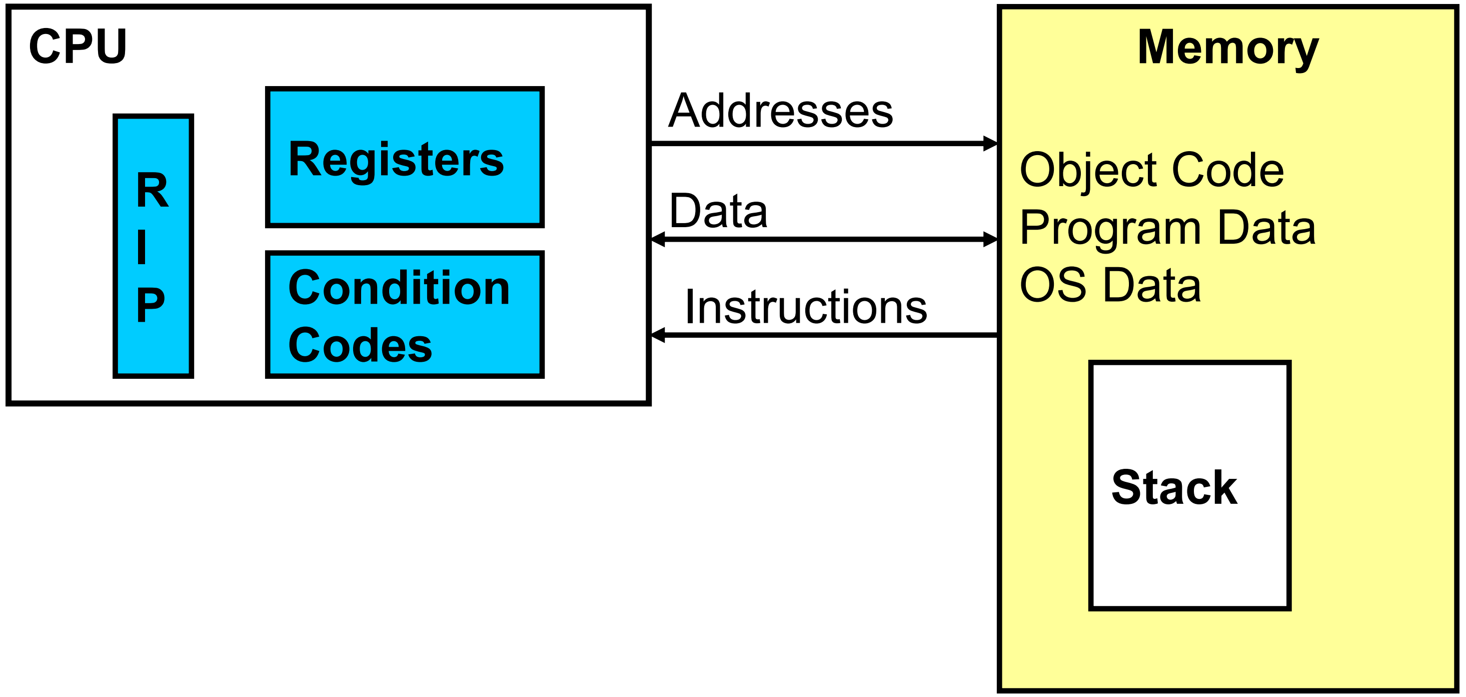
Definitions

- Instruction Set Architecture (ISA): the parts of a processor design that one needs to understand or write assembly code (instruction set specs, registers)
  - CISC (complex instruction set computer) architecture has many specialized instructions (with different lengths) some rarely used in programs:
    - Intel: x86, IA32, Itanium, x86-64
  - RISC (Reduced Instruction Set Computer) simplifies the processor by efficiently implementing only the instructions (with the same length) frequently used :
    - ARM: used in almost all mobile phones
- Microarchitecture: Implementation of the architecture.
  - Examples: cache memory sizes and core frequency.
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code

Assembly Programmer's View

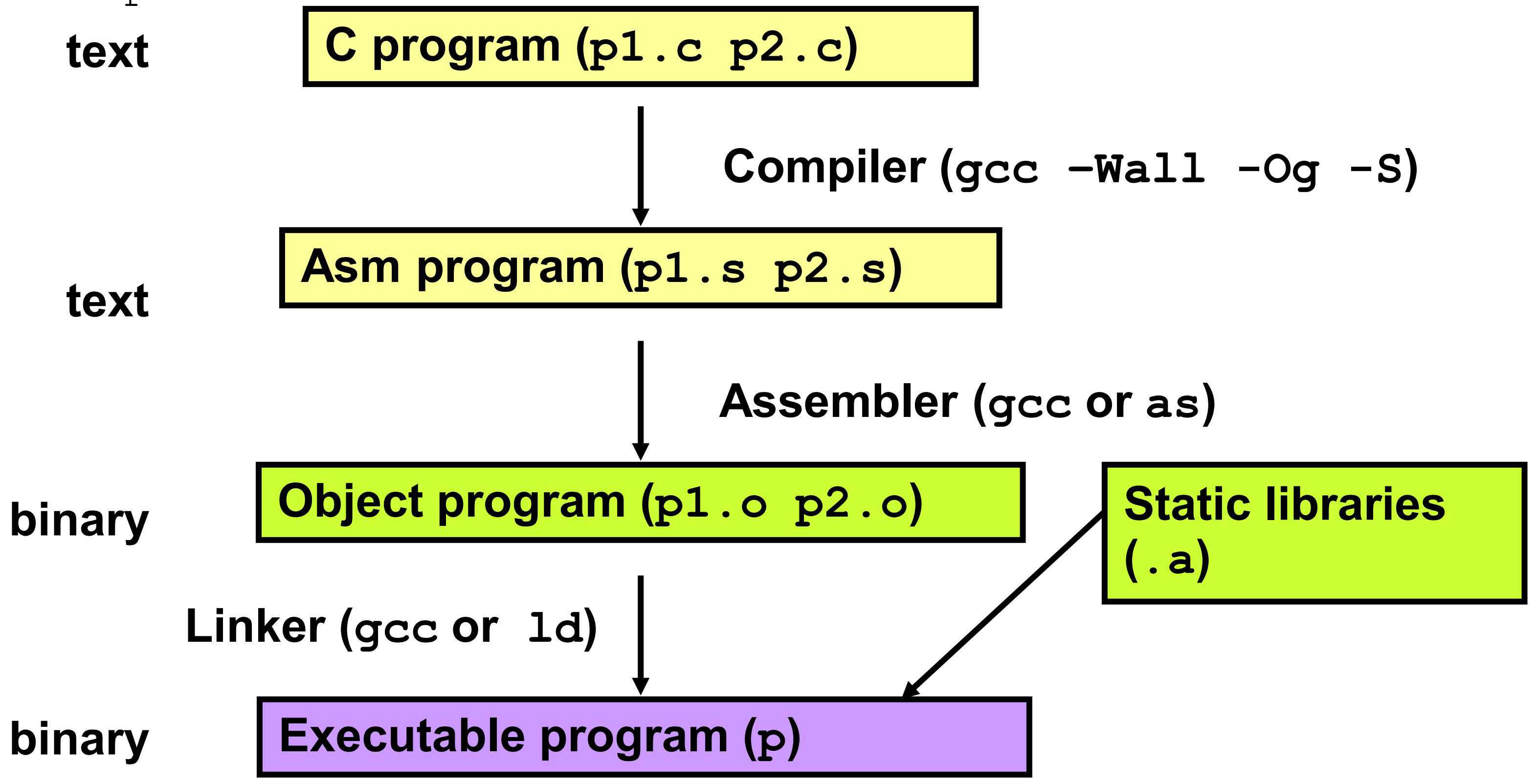
- Programmer-Visible State
  - RIP (Program Counter)
    - Address of next instruction
  - Register File
    - Heavily used program data
  - Condition Codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- Memory
  - Byte-addressable array
  - Code, user data, (most) OS data
  - Includes stack used to support procedures



Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Wall -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) (new to recent versions of gcc)
  - Put resulting binary in file `p`



Compiling Into Assembly

Ca' Foscari University of Venice

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtain with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`   (Note: removed a bunch of irrelevant *pseudo-ops* intended for the assembler)

Warning: May get very different results on other machines due to different versions of gcc and different compiler settings.

Assembly Characteristics

- Minimal Integer data types… of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (un-typed pointers)
- Floating-point data of 4, 8, 10 bytes
  - No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
  - Code is also just byte sequences encoding instructions

Assembly Characteristics

- Primitive operations
  - Perform arithmetic function on register or memory data
  - Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches

Object Code

Ca' Foscari University of Venice

Sample code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address `0x0400595`

- Assembler
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for malloc, printf
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution

Machine Instruction Example

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:   48 89 03
```

- C Code
  - Store value t where designated by dest
- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - t: Register %rax
    - dest: Register %rbx
    - *dest: Memory M[%rbx]
- Object Code
  - 3-byte instruction
  - Stored at address 0x40059e

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

- Disassembler
  - objdump –d sum
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either a.out (complete executable) or .o file

Alternate Disassembly

Ca' Foscari University of Venice

Object

```
0x0400595:
   0x53
   0x48
   0x89
   0xd3
   0xe8
   0xf2
   0xff
   0xff
   0xff
   0x48
   0x89
   0x03
   0x5b
   0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

What Can be Disassembled?

Ca' Foscari University of Venice

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

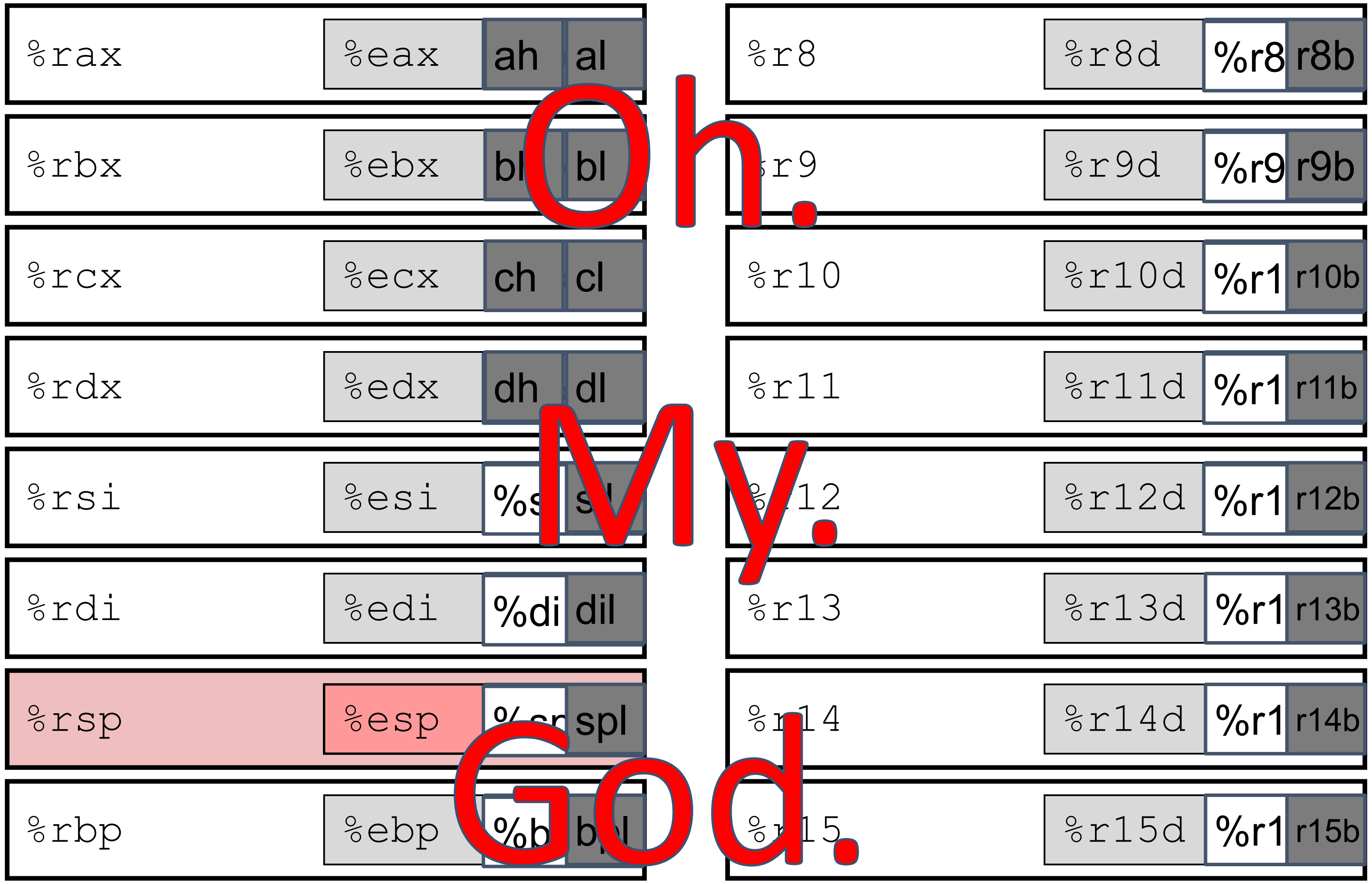
```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55             push   %ebp
30001001:  8b ec          mov    %esp,%ebp
30001003:  6a ff          push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91
```

Reverse engineering forbidden by Microsoft End User License Agreement

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

x86-64 Integer Registers



Oh.
My.
God.

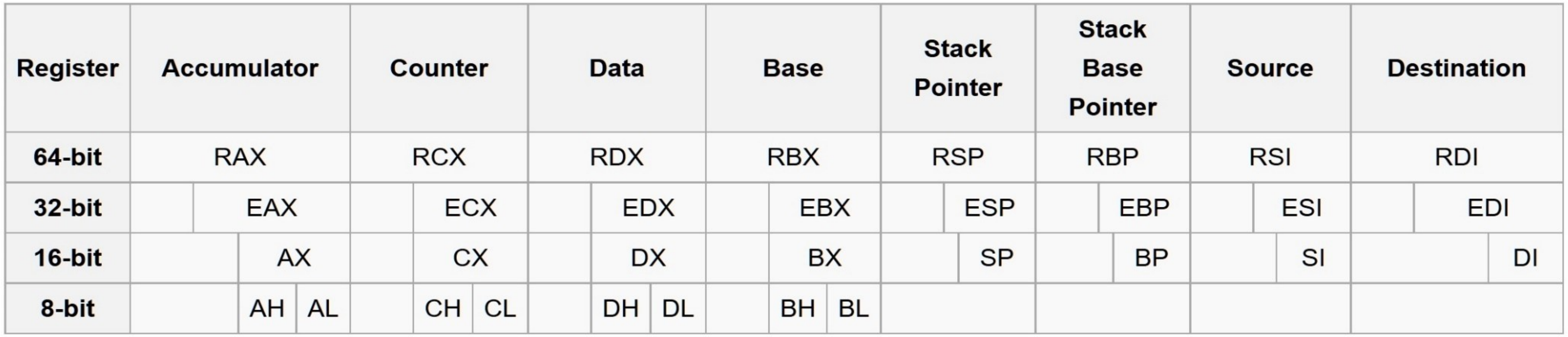
The 8 General-Purpose Registers (GPR)

- Accumulator register (AX). Used in arithmetic operations.
- Counter register (CX). Used in shift/rotate instructions and loops.
- Data register (DX). Used in arithmetic operations and I/O operations.
- Base register (BX). Used as a pointer to data (located in segment register DS, when in segmented mode).
- Stack Pointer register (SP). Pointer to the top of the stack.
- Stack Base Pointer register (BP). Used to point to the base of the stack.
- Source Index register (SI). Used as a pointer to a source in stream operations.
- Destination Index register (DI). Used as a pointer to a destination in stream ops.

Register	Accumulator	Counter	Data	Base	Stack Pointer	Stack Base Pointer	Source	Destination
64-bit	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
32-bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
16-bit	AX	CX	DX	BX	SP	BP	SI	DI
8-bit	AH AL	CH CL	DH DL	BH BL				

The 8 General-Purpose Registers (GPR)

- All registers can be accessed in 16-bit and 32-bit modes.
- In 16-bit mode, the register is identified by its two-letter abbreviation from the list below.
- In 32-bit mode, this two-letter abbreviation is prefixed with an 'E' (*extended*). E.g., 'EAX' is the accumulator register as a 32-bit value.
- Similarly, in the 64-bit version, the 'E' is replaced with an 'R', so the 64-bit version of 'EAX' is called 'RAX'.
- It is also possible to address the first four registers (AX, CX, DX and BX) in their size of 16-bit as two 8-bit halves.
  - The least significant byte (LSB), or low half, is identified by replacing the 'X' with an 'L'.
  - The most significant byte (MSB), or high half, uses an 'H' instead.
  - For example, CL is the LSB of the counter register, whereas CH is its MSB.
- Total five ways to access the accumulator, counter, data and base registers: 64-bit, 32-bit, 16-bit, 8-bit LSB, and 8-bit MSB.
- The other four are accessed in only three ways: 64-bit, 32-bit and 16-bit.



Register	Accumulator	Counter	Data	Base	Stack Pointer	Stack Base Pointer	Source	Destination
64-bit	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
32-bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
16-bit	AX	CX	DX	BX	SP	BP	SI	DI
8-bit	AH AL	CH CL	DH DL	BH BL				

The 6 Segment Registers

- Stack Segment (SS). Pointer to the stack.
- Code Segment (CS). Pointer to the code.
- Data Segment (DS). Pointer to the data.
- Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').
- F Segment (FS). Pointer to more extra data ('F' comes after 'E').
- G Segment (GS). Pointer to still more extra data ('G' comes after 'F').

Most applications on most modern operating systems (like FreeBSD, Linux or Microsoft Windows) use a memory model that points nearly all segment registers to the same place (and uses paging instead), effectively disabling their use.

FS or GS is an exception to this rule, being used to point at thread-specific data.

X86 is Little-Endian

- The x86 is a little-endian architecture, meaning that the least significant byte of a multibyte data value is stored first, at a lower memory address than each subsequent byte of that data item.
- Big-endian data is stored in reverse order, with the most significant byte of a data value being stored at a lower memory address than each subsequent byte.
- Processors may be classified as big-endian or little-endian, or in some cases, both.

EFLAGS Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

The different use of these flags are:
0. **CF** : Carry Flag. Set if the last arithmetic operation carried (addition) or borrowed (subtraction) a bit beyond the size of the register. This is then checked when the operation is followed with an add-with-carry or subtract-with-borrow to deal with values too large for just one register to contain.
2. **PF** : Parity Flag. Set if the number of set bits in the least significant byte is a multiple of 2.
4. AF : Adjust Flag. Carry of Binary Code Decimal (BCD) numbers arithmetic operations.
6. **ZF** : Zero Flag. Set if the result of an operation is Zero (0).
7. **SF** : Sign Flag. Set if the result of an operation is negative.
8. **TF** : Trap Flag. Set if step by step debugging.
9. IF : Interruption Flag. Set if interrupts are enabled.
10. DF : Direction Flag. Stream direction. If set, string operations will decrement their pointer rather than incrementing it, reading memory backwards.

EFLAGS Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

11. **OF** : Overflow Flag. Set if signed arithmetic operations result in a value too large for the register to contain.
12-13. IOPL : I/O Privilege Level field (2 bits). I/O Privilege Level of the current process.
14. NT : Nested Task flag. Controls chaining of interrupts. Set if the current process is linked to the next process.
16. RF : Resume Flag. Response to debug exceptions.
17. VM : Virtual-8086 Mode. Set if in 8086 compatibility mode.
18. AC : Alignment Check. Set if alignment checking of memory references is done.
19. VIF : Virtual Interrupt Flag. Virtual image of IF.
20. VIP : Virtual Interrupt Pending flag. Set if an interrupt is pending.
21. ID : Identification Flag. Support for CPUID instruction if can be set.

Jumps and Loops

- The basic instructions for branching are jumps and loops
- Loops are instructions to repeat a block of code a certain number of times
- Jumps are instructions that branch to a distant labeled instruction when a flag condition is met
  - Status flags are modified by arithmetic instructions, so these are normally used just before a jump
  - Example: the JNZ (jump if not zero) instruction jumps to the destination-label if ZF = 0. Usage: JNZ destination-label

Jcond Instruction

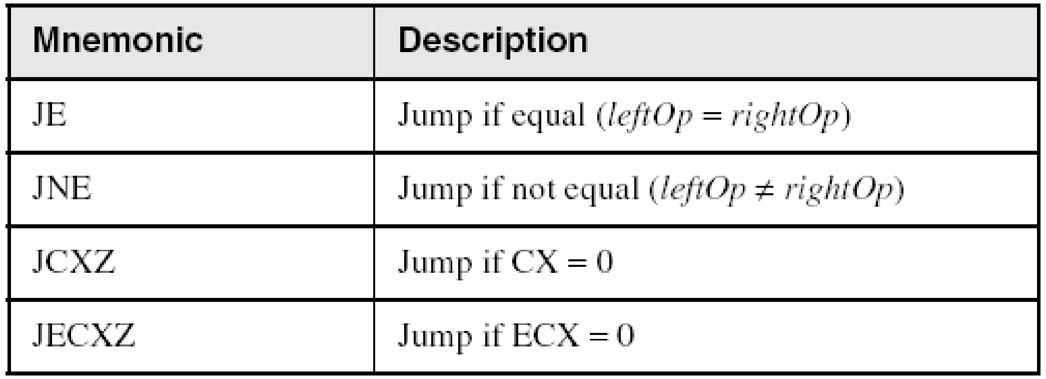
- A conditional jump instruction branches to a label when specific register or flag conditions are met
- Specific jumps:
  - JB, JC - jump to a label if the Carry flag is set
  - JE, JZ - jump to a label if the Zero flag is set
  - JS - jump to a label if the Sign flag is set
  - JNE, JNZ - jump to a label if the Zero flag is clear
  - JECXZ - jump to a label if ECX = 0

Jumps Based on Specific Flags

Ca' Foscari University of Venice

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Jumps Based on Equality



Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

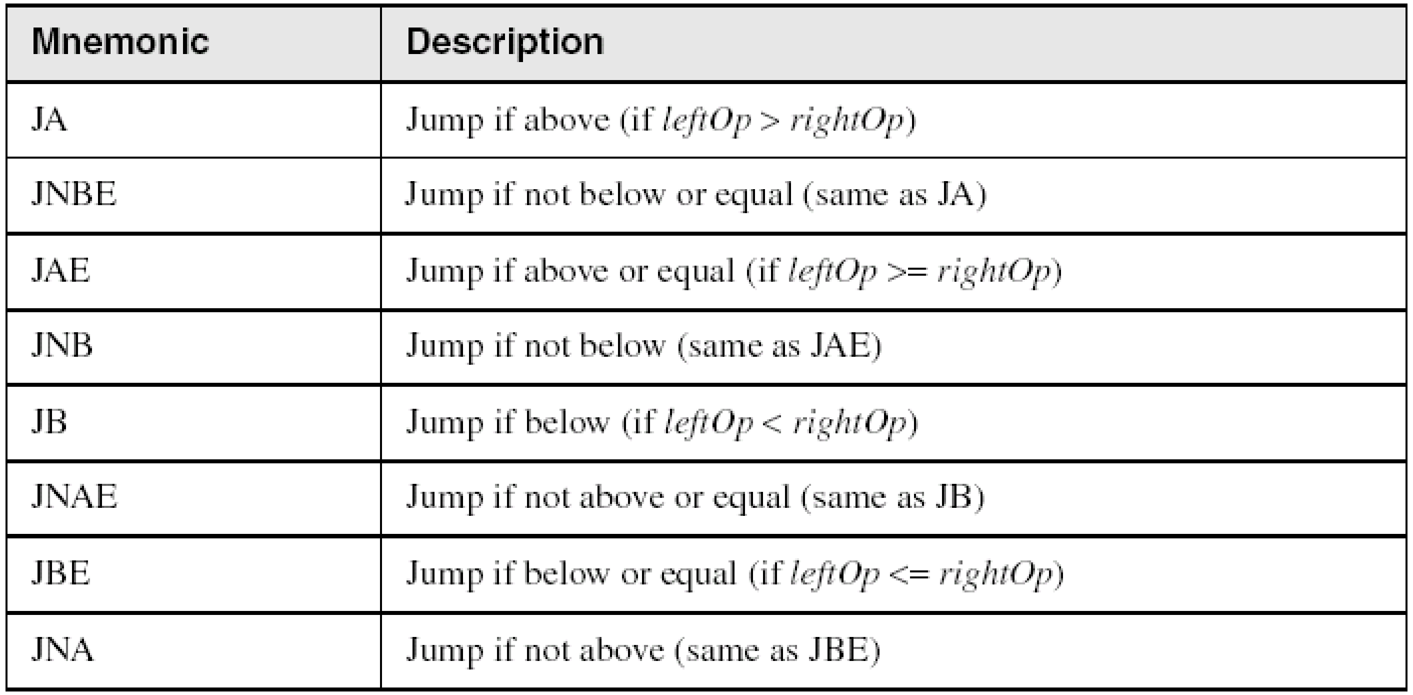
Note:

- Sometimes, the same jump instruction has 2 different mnemonics
  - Example: JZ $\leftrightarrow$ JE
- The value of the flag is the condition for jumping
  - Example: Jump to dest-label when the zero flag is set
- Jumping based on register condition: JCXZ and JECXZ

Jumping on an Inequality Condition

- Often, we need to branch when some value is larger (or smaller) than an other. Ex:
    - CMP eax, ebx
    - ;now jump somewhere when eax > ebx
- However, integer order (larger or smaller) depends on the chosen interpretation
- Ex: if AL contains 05h and BL contains A0h. Then:
    - AL > BL for a signed interpretation: for AL = +5, BL = -96
    - AL < BL for an unsigned interpretation: for AL = 5, BL = 160
- Hence, we have these two categories of jumps:
    - Unsigned comparison jumps
    - Signed comparison jumps

Jumps Based on Unsigned Comparisons

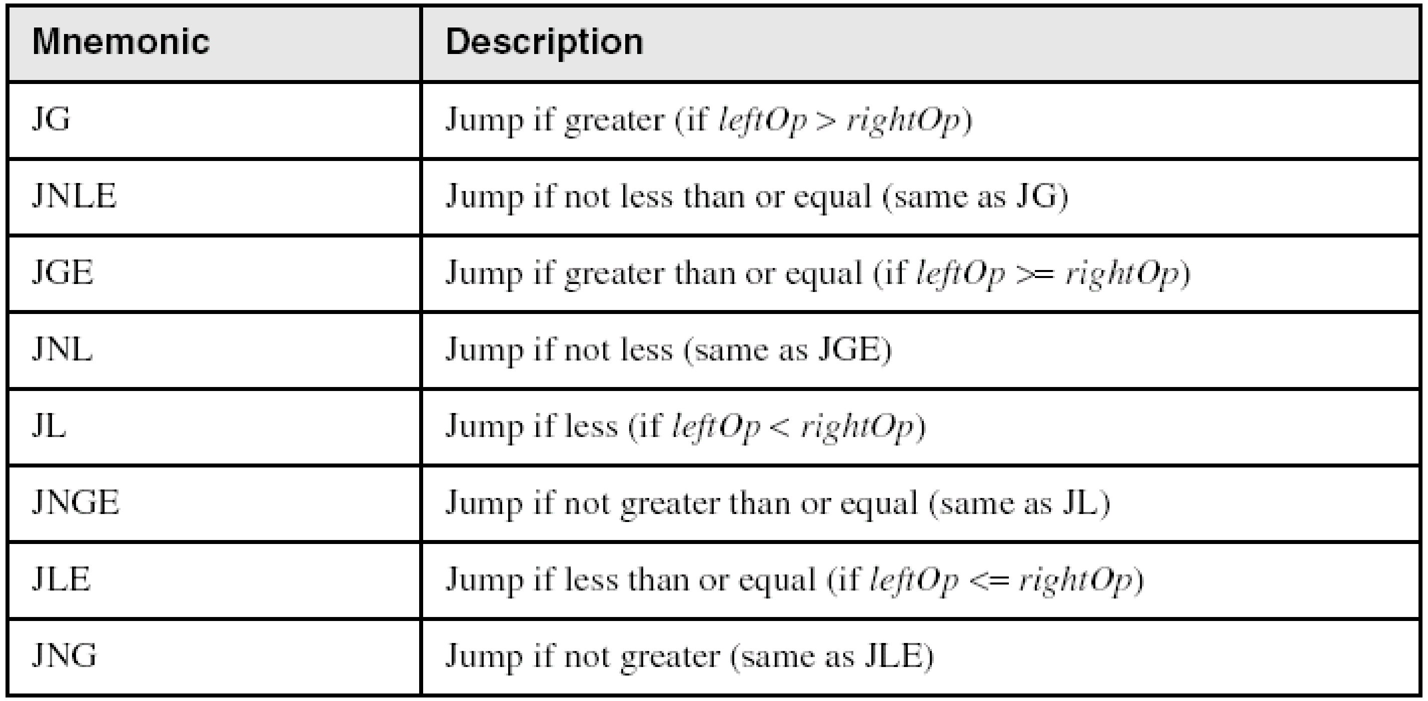


Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)
JNA	Jump if not above (same as JBE)

- Each of these instructions have 2 different mnemonics
- We normally used them just after a ***CMP op1,op2*** instruction and the jumping condition is given by an unsigned interpretation of the comparison

Jumps Based on Signed Comparisons

Ca' Foscari University of Venice



Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* >= *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* <= *rightOp*)
JNG	Jump if not greater (same as JLE)

- Each of these instructions have 2 different mnemonics
- We normally used them just after a ***CMP op1,op2*** instruction and the jumping condition is given by a signed interpretation of the comparison

Using Comparison Jumps

Ca' Foscari University of Venice

- CMP is normally used before a comparison jump
- Ex: to branch to exit when AX > BX under a signed interpretation (ex: AX=1, BX=FFFFh):
  - cmp ax,bx;   ax = +1, bx = -1
  - jg exit
- But to branch to exit when AX > BX under an unsigned interpretation:
  - cmp ax , bx;   ax = 1, bx = 65535
  - ja exit
  - Note that the jump is not performed when
    - AX=1 and BX =FFFFh

Unconditional JMP Instruction

- Sometimes we need to jump without a condition.
- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: JMP *target*
- Logic: EIP $\leftarrow$ *target*
- Example:

```
target:
        .
        .
        jmp target
```

Unconditional Jump

```
.data
        msg BYTE "hello",0
.code

        main   PROC
                mov edx,12345678h
                jmp over
                mov edx,OFFSET msg
                over:
                         CALL WriteString
                exit
        main   ENDP
end main
```

Instruction "mov edx,OFFSET msg" has not been executed

High-Level Flow Control Structures

- High-level languages uses high-level structures such as if-then-else, switch, while or repeat statements to control the flow of execution
  - algorithms are normally expressed with these high-level structures
- Processors only provide conditional and unconditional jump and loop instructions
  - we need to decompose the high-level control flow structures into low-level ones
- This can be done by using jump instructions

If-Then-Else

Ca' Foscari University of Venice

- **HLL Observation**

If *Condition*

{ Code-Block-1 }

else

{ Code-Block-2 }

The program *branches* to Code-Block-2 if *Condition* is *False*

- **Assembler Observation**

***Jcc* Code-Block-2**

The program *branches* to Code-Block-2 if *cc* e.g.,J*E* or J*NZ*, is *True*

- **Therefore HLL *not Condition* ⇔ Assembler *Jcc***

If-Then-Else

- Example:

```
if (op1 < op2) then
        statement 1
else
        statement 2
end_if
```

- Analysis:
  - there is a conditional jump (JXXX) to else when
    - op1 >= op2
  - there is an unconditional jump (JMP) from end of “statement 1” to end_if

- ASM solution for signed comparison:

```
cmp op1,op2
jge else_
 ;put statement 1 here
jmp end_if
else_:
 ;put statement 2 here
end_if:
```

- Note: “else” is a ASM reserved word. We use “else_” instead

While

- Example :

```
while (op1 < op2)
      statement
end do
```

- Analysis:
  - JXXX to end_do when
    - op1 >= op2
  - JMP from end_do to while

- ASM solution for an unsigned comparison:

```
do_while:
      cmp op1,op2
      jae end_do
       ;put statement here
      jmp do_while
end_do:
```

Case or Switch

- Example:

```
case input of
    'A' :DestA
    'B' :DestB
    'C' :DestC
end case
```

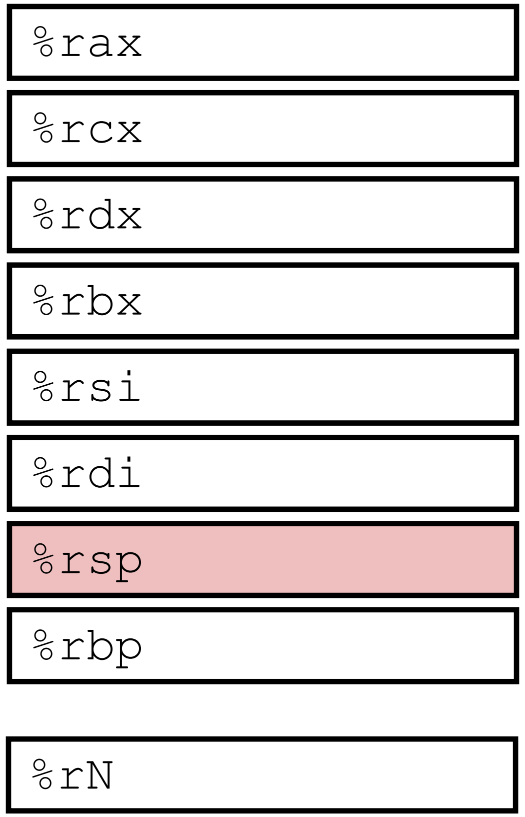
- Analysis: CMP and JXXX for each case

- ASM solution :

```
        cmp input,'A'
        jne L1
        JMP DestA
L1:
        cmp input,'B'
        jne L2
        JMP DestB
L2:
        cmp input,'C'
        jne L3
        JMP DestC
L3:
```

Moving Data

- Moving Data: **`movq`** ***Source, Dest***
- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400, $-533`**
    - Like C constant, but prefixed with **`'$'`**
    - Encoded with 1, 2, 4, or 8 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax, %r13`**
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other "address modes"



movq Operand Combinations

	Source	Dest	Src, Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	**`temp = 0x4;`**
		Mem	`movq $-147,(%rax)`	**`*p = -147;`**
	Reg	*Reg*	`movq %rax,%rdx`	**`temp2 = temp1;`**
		Mem	`movq %rax,(%rdx)`	**`*p = temp;`**
	Mem	*Reg*	`movq (%rax),%rdx`	**`temp = *p;`**

- *Cannot do memory-memory transfer with a single instruction*

Simple Addressing Modes

- Direct A Mem[A]
  - Memory address A is directly specified
  - Mostly used for static and global variables

```
movl 0x804acb8,%eax
```

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Pointer dereferencing in C

```
movq (%rcx),%rax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

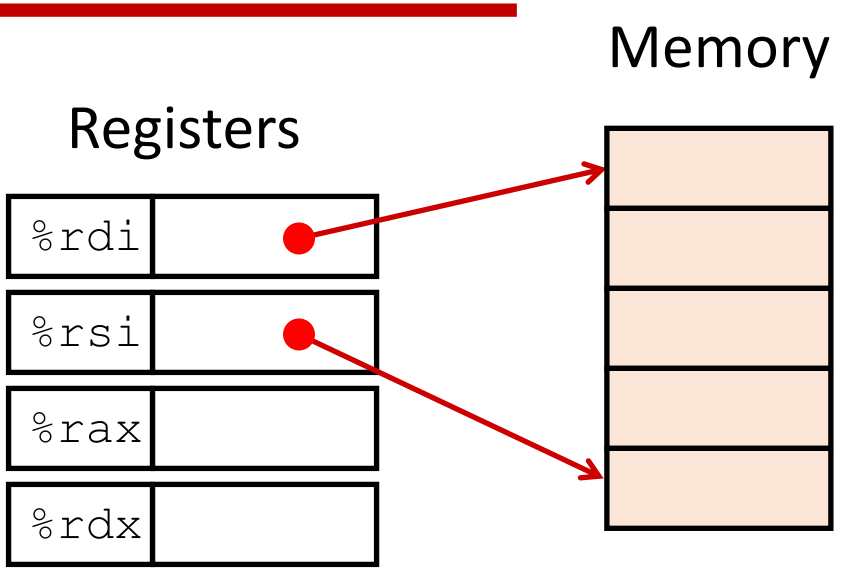
Example of Simple Addressing Modes

```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq     (%rdi), %rax
   movq     (%rsi), %rdx
   movq     %rdx, (%rdi)
   movq     %rax, (%rsi)
   ret
```

Understanding Swap()

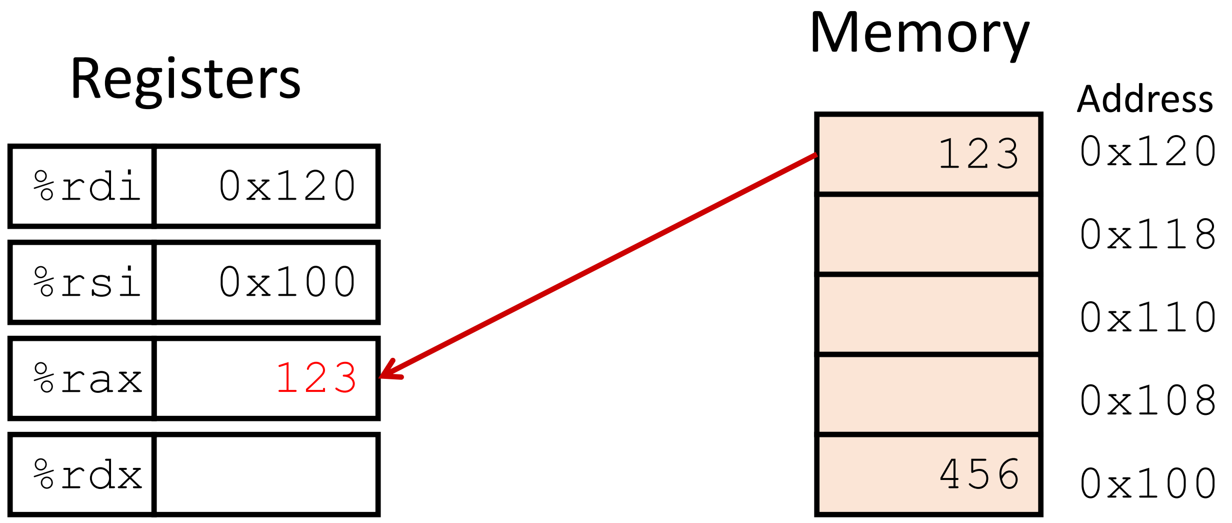
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

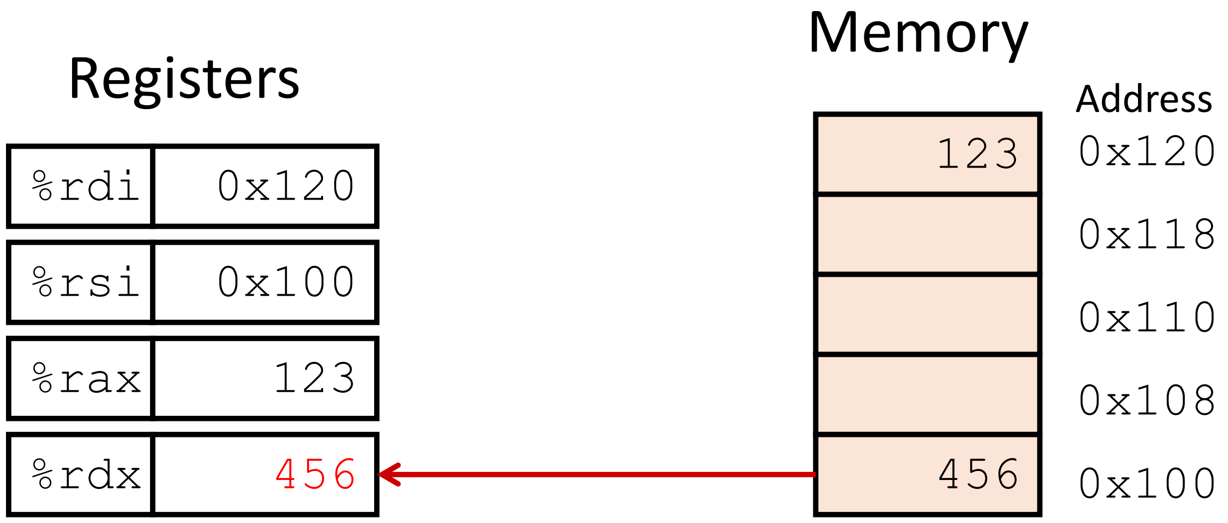
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



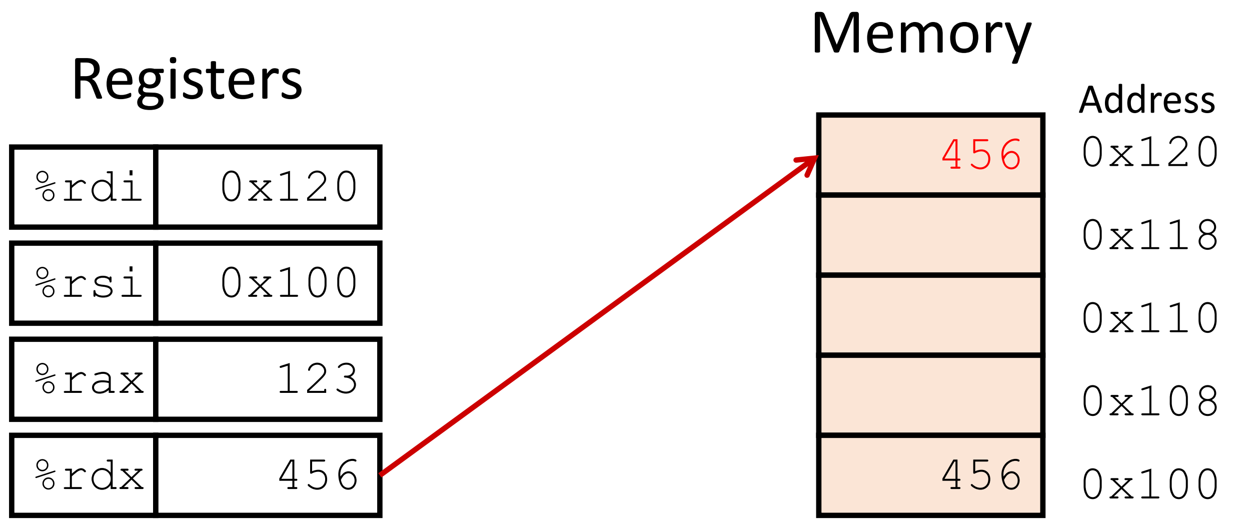
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



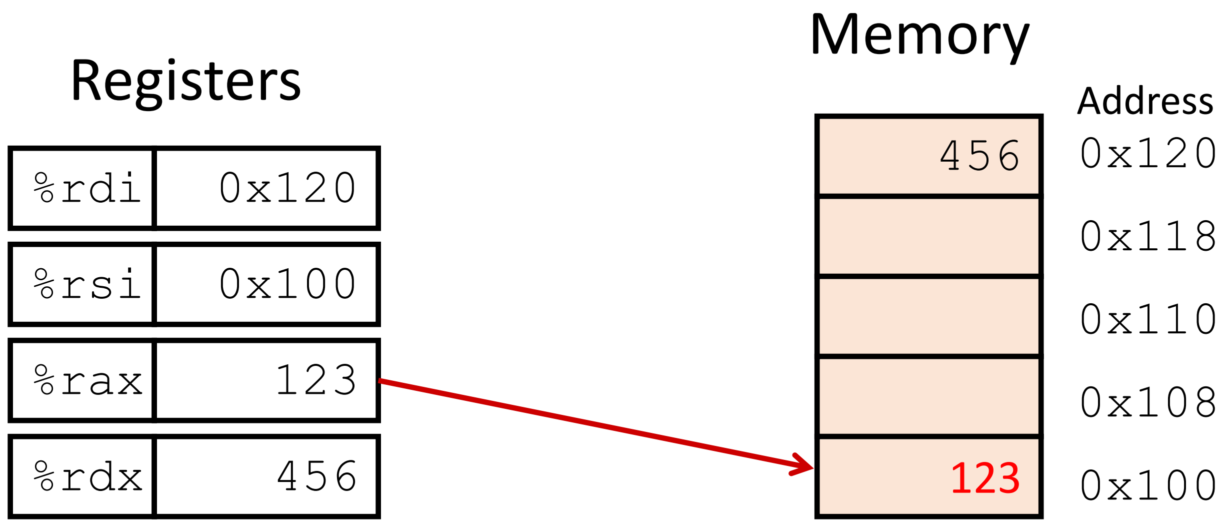
```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Complete Addressing Modes

Most General Form

D(Rb, Ri, S) **Mem[Reg[Rb] + S*Reg[Ri] + D]**

- D: Constant "displacement" 1, 2, or 4 bytes (but not 8)
  - Can be small (offset) or large (address)

Rb: Base register: Any of 16 integer registers	🡐 Array address	Reg[Rb] +D
Ri: Index register: Any, except for %rsp	🡐 Array index	Ri
S: Scale: 1, 2, 4, or 8	🡐 Size of Array element	S

Special Cases

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]
D	Mem[D]
(,Ri,S)	Mem[S*Reg[Ri]]

Address Computation Examples

%rdx	**0xf000**
%rcx	**0x100**

Expression	Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Address Computation Instruction

- leaq Src, Dest
  - Src is address mode expression
  - Set Dest to address denoted by expression
- Uses
  - Computing address without doing memory reference
    - E.g., translation of p = &x[i];
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8.
- Used heavily by compiler

leaq vs. movq

Assume dest is %rax:

- %rdi = 0xF000
- %rsi = 0x8
- Memory at 0xF000 = 0x12345
- Memory at 0xF008 = 0x6789A
- Memory at 0xF010 = 0xBCDEF

Src	leaq	movq
(%rdi)	0xF000	0x12345
8(%rdi)	0xF008	0x6789A
(%rdi,%rsi)	0xF008	0x6789A
(%rdi,%rsi,2)	0xF010	0xBCDEF
%rdi	Illegal!	0xF000

Some Arithmetic Operations

Ca' Foscari University of Venice

- Two-Operand Instructions:

Format		Computation	
addq	Src,Dest	Dest = Dest + Src	
subq	Src,Dest	Dest = Dest – Src	
imulq	Src,Dest	Dest = Dest * Src	
salq	Src,Dest	Dest = Dest << Src	Also called shlq
sarq	Src,Dest	Dest = Dest >> Src	Arithmetic
shrq	Src,Dest	Dest = Dest >> Src	Logical
xorq	Src,Dest	Dest = Dest ^ Src	
andq	Src,Dest	Dest = Dest & Src	
orq	Src,Dest	Dest = Dest | Src	

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)
- Note: immediate source limited to 4 bytes (sigh)

Some Arithmetic Operations

Ca' Foscari University of Venice

- One-Operand Instructions

`incq`	*Dest = Dest + 1*
`decq`	*Dest = Dest – 1*
`negq`	*Dest = – Dest*
`notq`	*Dest = ~Dest*