جامعة روتشستر الأمريكية للتكنولوجيا في دبي

A Global American University in Dubai

CSEC 202 Reverse Engineering Fundamentals

Review of CPU Design & Architecture

Eng. Emad Abu Khousa

Sections: 600 | 601 | 602

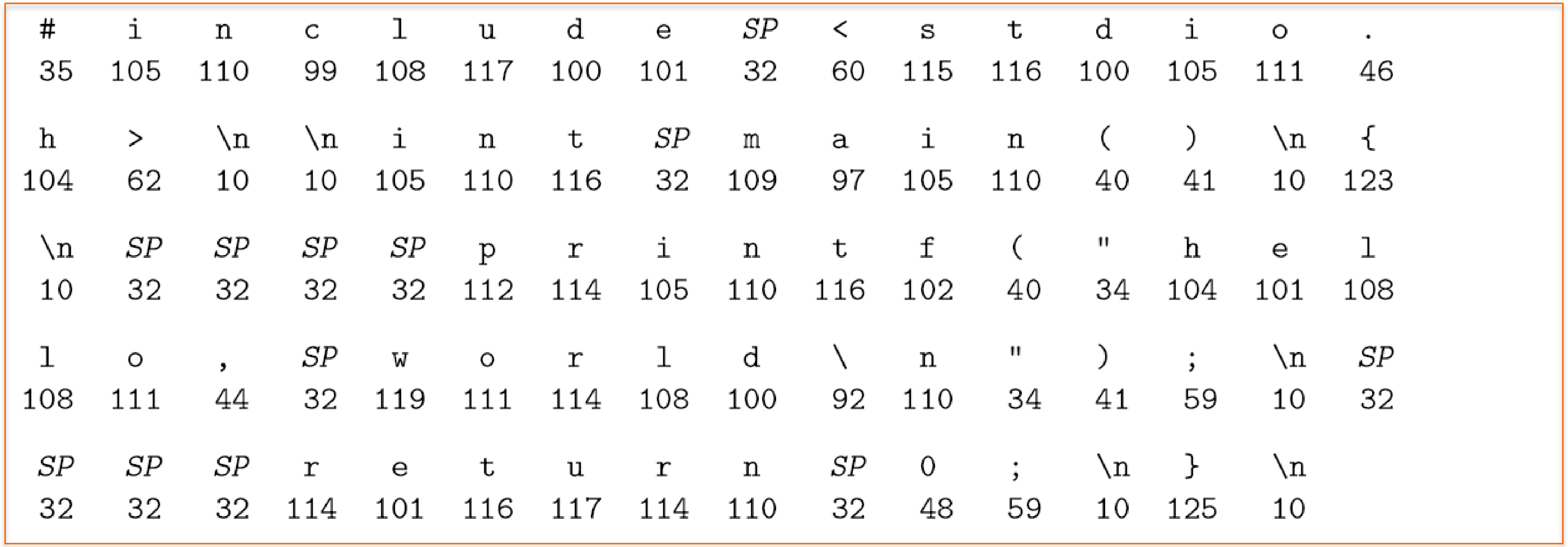
January 22, 2024

The Lifetime of Hello World

The Lifetime of hello.c

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

The Lifetime of hello.c – ASCII

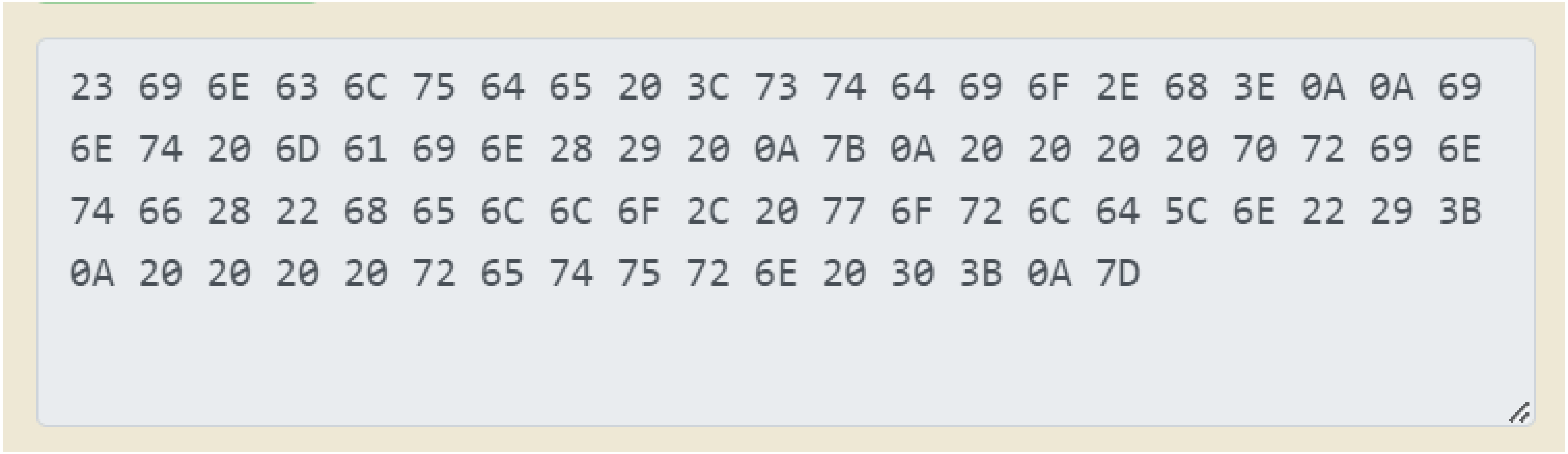


#	i	n	c	l	u	d	e	*SP*	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	*SP*	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	*SP*	*SP*	*SP*	*SP*	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	*SP*	w	o	r	l	d	\	n	"	)	;	\n	*SP*
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	*SP*	*SP*	r	e	t	u	r	n	*SP*	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

The ASCII text representation of `hello.c`.

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort.

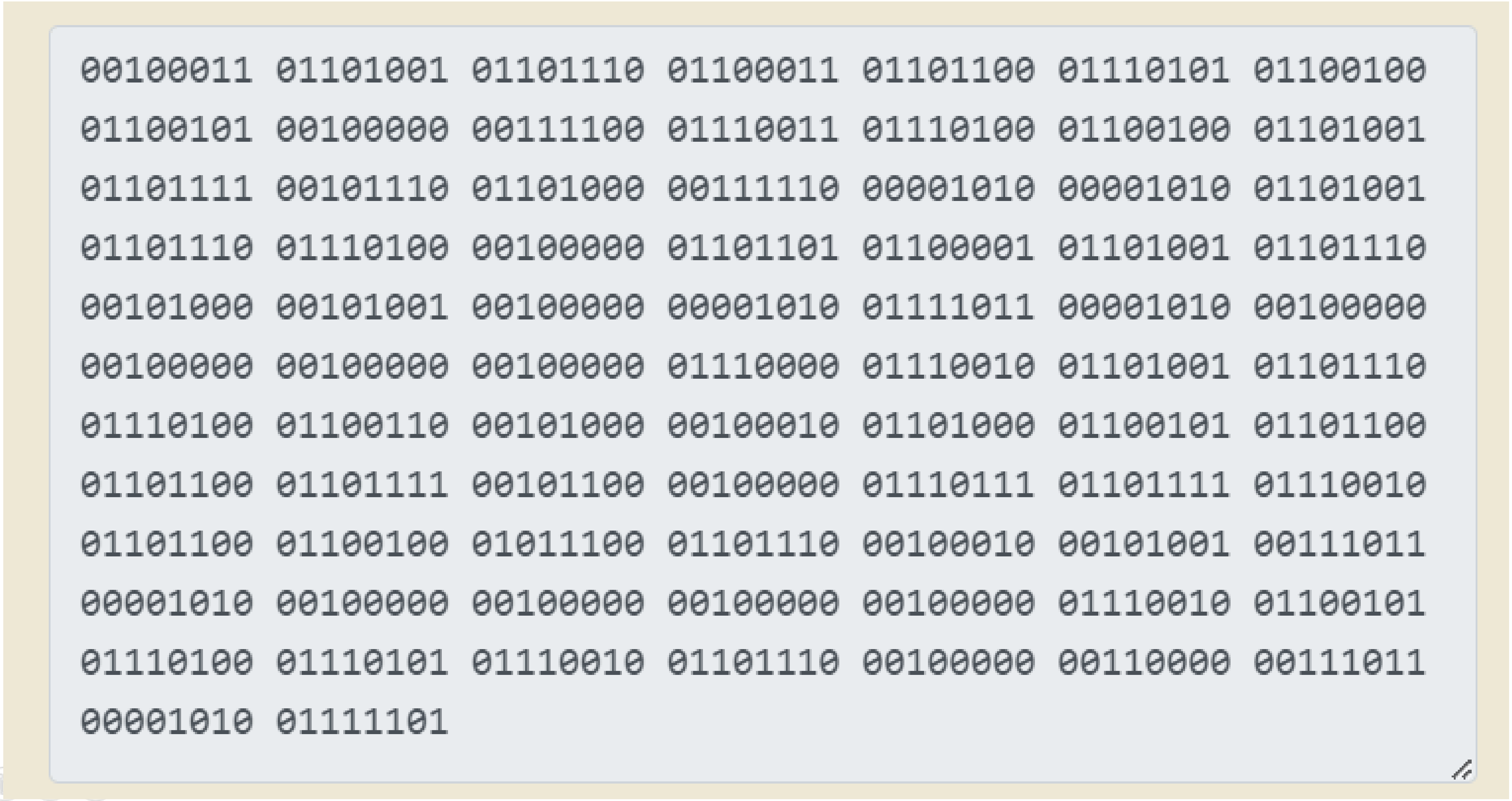
The Lifetime of hello.c – ASCII - Hex



```
23 69 6E 63 6C 75 64 65 20 3C 73 74 64 69 6F 2E 68 3E 0A 0A 69
6E 74 20 6D 61 69 6E 28 29 20 0A 7B 0A 20 20 20 20 70 72 69 6E
74 66 28 22 68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 5C 6E 22 29 3B
0A 20 20 20 20 72 65 74 75 72 6E 20 30 3B 0A 7D
```

https://www.rapidtables.com/convert/number/ascii-to-binary.html

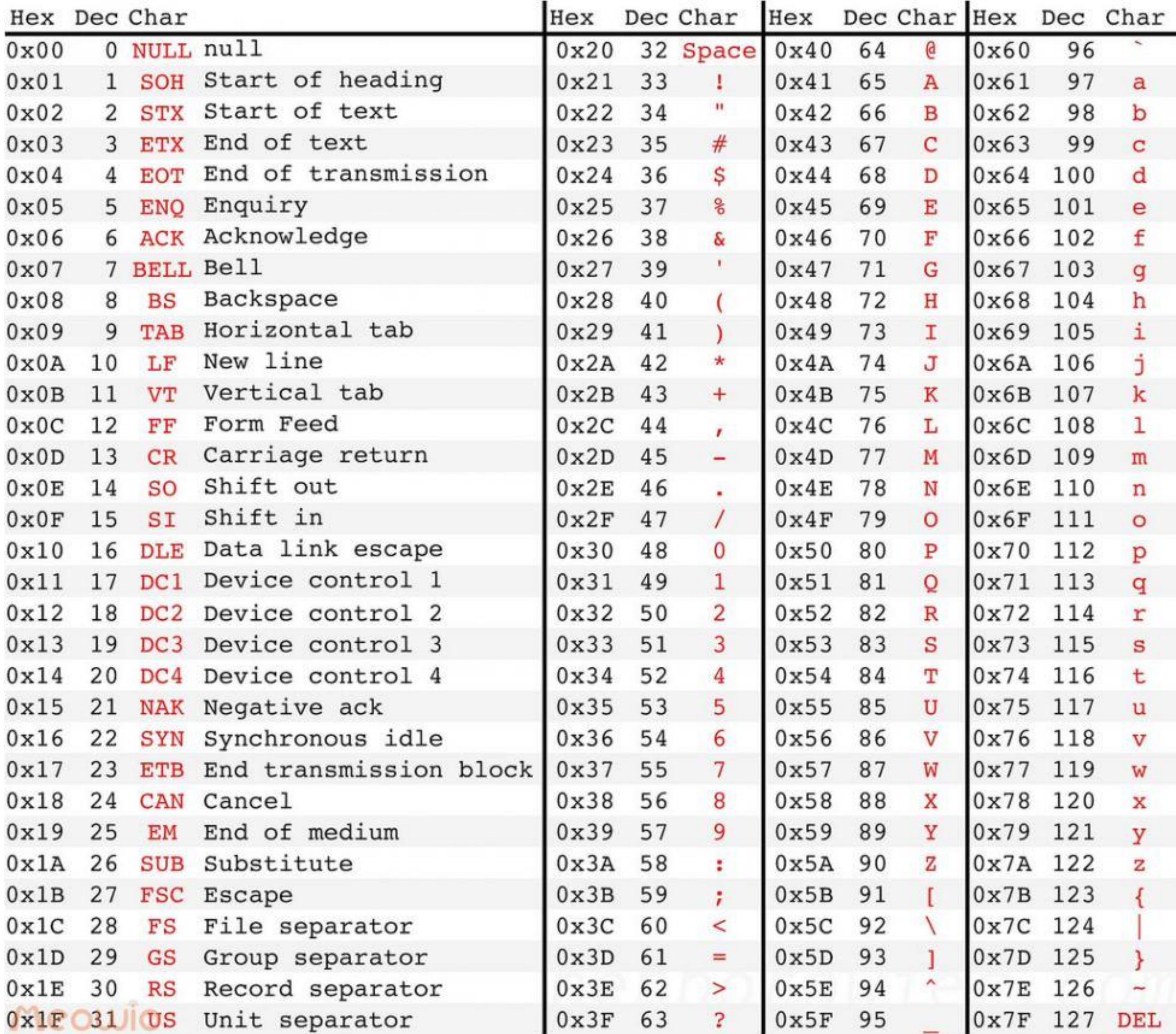
The Lifetime of hello.c – ASCII - Binary



```
00100011 01101001 01101110 01100011 01101100 01110101 01100100
01100101 00100000 00111100 01110011 01110100 01100100 01101001
01101111 00101110 01101000 00111110 00001010 00001010 01101001
01101110 01110100 00100000 01101101 01100001 01101001 01101110
00101000 00101001 00100000 00001010 01111011 00001010 00100000
00100000 00100000 00100000 01110000 01110010 01101001 01101110
01110100 01100110 00101000 00100010 01101000 01100101 01101100
01101100 01101111 00101100 00100000 01110111 01101111 01110010
01101100 01100100 01011100 01101110 00100010 00101001 00111011
00001010 00100000 00100000 00100000 00100000 01110010 01100101
01110100 01110101 01110010 01101110 00100000 00110000 00111011
00001010 01111101
```

RIT Rochester Institute of Technology of Dubai

The ASCII table is a character-encoding standard for electronic communication. ASCII stands for American Standard Code for Information Interchange. It assigns a unique decimal number to different characters, including letters, digits, punctuation marks, and control characters. For instance, in ASCII, the uppercase letter "A" is represented by the number 65, while the lowercase "a" is 97. Each entry in the ASCII table corresponds to a **byte** in the computer's memory, with the standard ASCII using 7 bits of the byte, giving it the ability to represent **128 unique characters**. ASCII is fundamental in computers and is used to translate human-readable text into a format that machines can process.



Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

Hands On – IDA – Static Analysis 101:

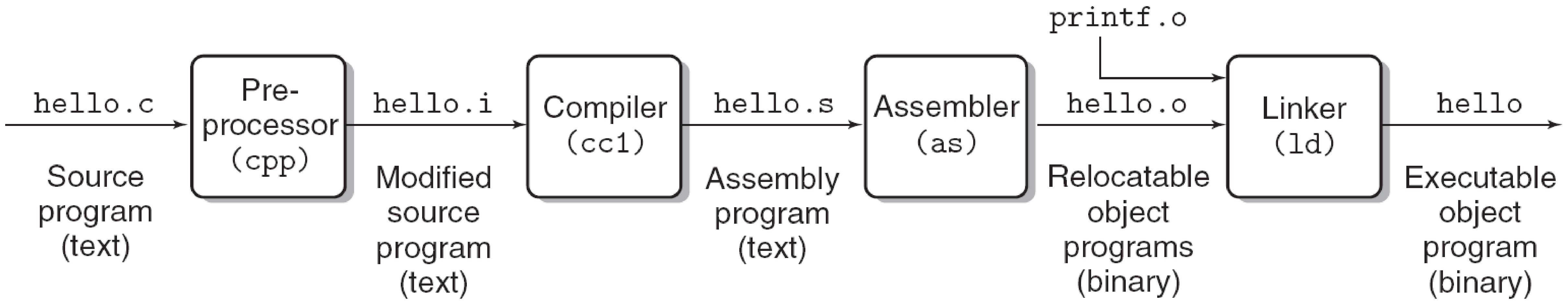
- Compile the **hello.c** file by typing the following command in the terminal:

```
gcc hello.c -o hello
```

  - This command creates an executable file named hello.
- Open IDA and load the newly created binary hello.
- In IDA, search for the string "**hello, world**" to find where it appears in the code.
- Note down the **memory address** where the string "hello, world" is located.
- Observe and write down how the string is represented in memory by switching to the Hex View in IDA.
- Determine whether the compiled binary is 64-bit or 32-bit. This can be identified by examining the file's properties or analyzing the architecture-specific details within IDA.

The Compilation System: gcc hello.c -o hello

-



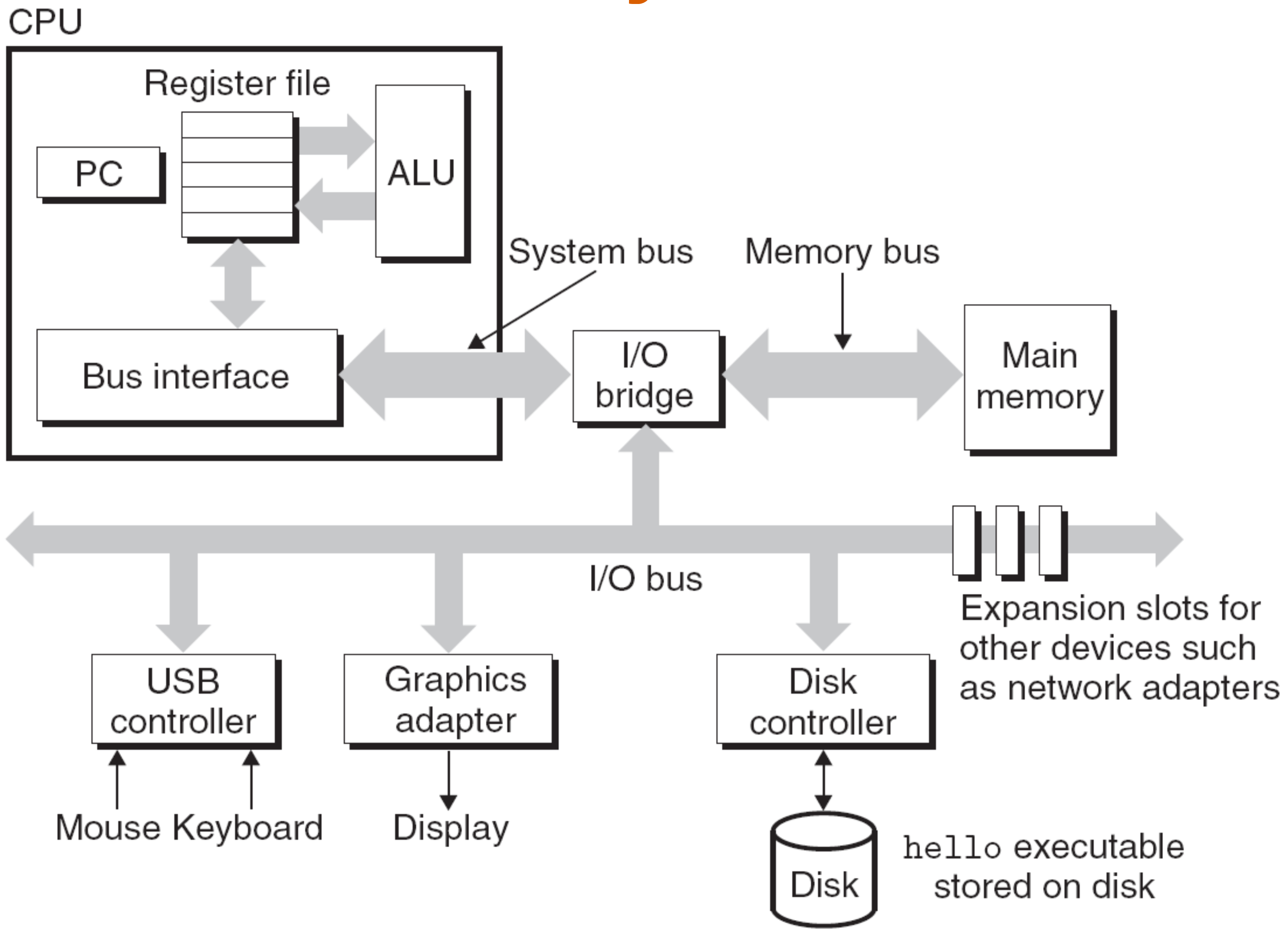
The compilation system.

The Compilation System: gcc hello.c -o hello

- The four phases of compiling a C program involve different programs:
- **Preprocessor (cpp):** Prepares the source code by processing directives like #include.
- **Compiler (gcc / ccl):** Transforms the preprocessed code into assembly language.
- **Assembler (as):** Converts assembly language into machine code (object files).
- **Linker (ld):** Combines object files and libraries into a final executable.

Hardware Organization of a System

Hardware organization of a typical system. CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

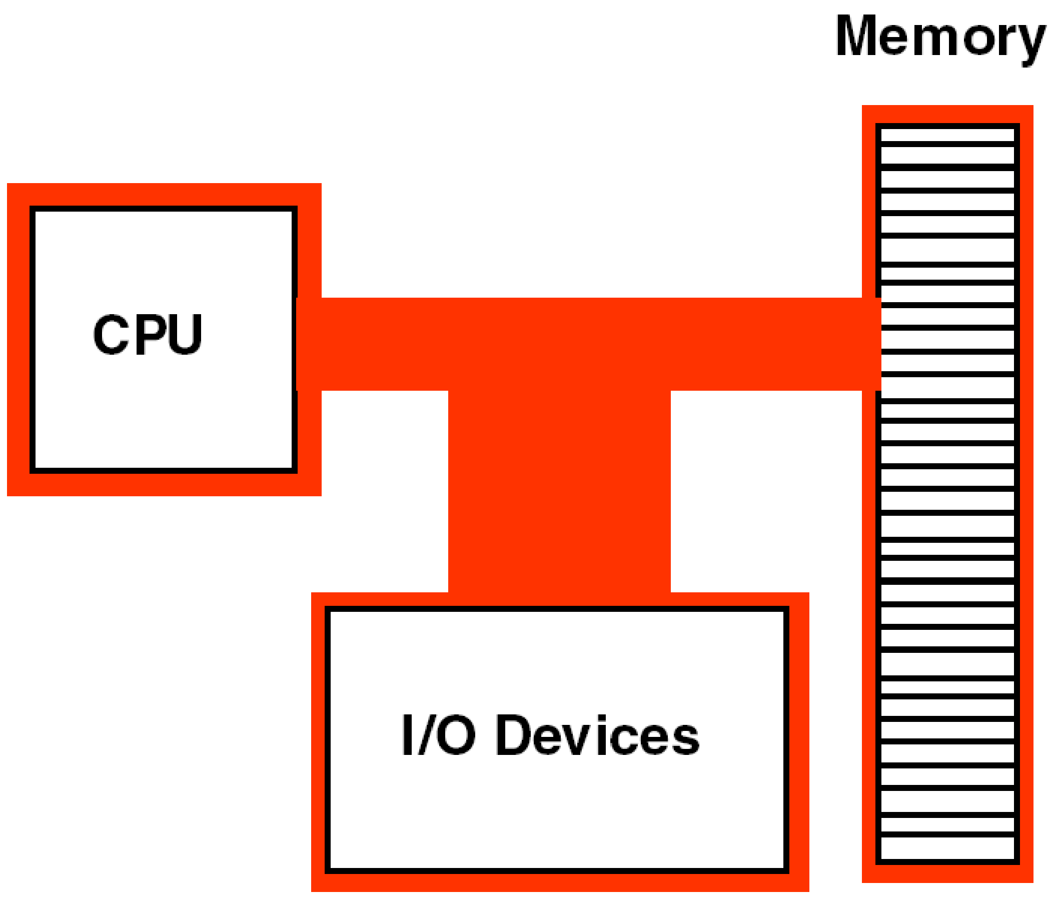


The Von Neumann Architecture

The Von Neumann Architecture

Von Neumann Architecture refers to a design model for computers where the processing unit, memory, and input-output devices are interconnected through a single, central system bus.

This architecture was first proposed by John von Neumann, a Hungarian-American mathematician and physicist, in the mid-20th century.



Check your understanding:

What are the four main components of Von Neumann Architecture?

a) Central Processing Unit (CPU), Registers, Cache, And Input-Output (I/O) Devices
b) Central Processing Unit (CPU), Memory (RAM And Secondary Memory), Input-Output (I/O) Devices, And System Bus
c) Arithmetic Logic Unit (ALU), Control Unit (CU), Memory, And System Bus
d) Central Processing Unit (CPU), Primary Memory, External Storage Devices, And Input-Output (I/O) Devices

Hardware Organization of a System:

There are four main components within the Von Neumann Architecture. These components work together to enable processing, storage, and communication within the computer system. They are:

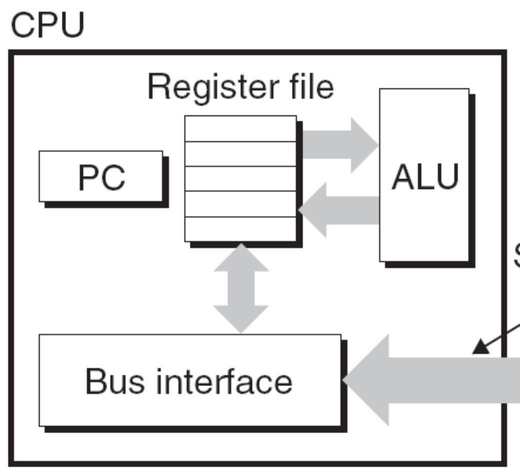
- **Central Processing Unit (CPU):** The part of a computer that carries out instructions and performs arithmetic, logical, and control operations.
- **Memory:** A place where the computer stores and retrieves data and instructions. Memory is divided into two types: primary memory, such as Random Access Memory (RAM), and secondary memory, like hard disk drives and solid-state drives.
- **Input-Output (I/O) devices:** Components responsible for interfacing the computer with the external world. Examples of I/O devices include keyboards, mice, printers, and monitors.
- **System Bus:** A communication pathway that connects the CPU, memory, and I/O devices, enabling data and control signals to flow between these components.

The smooth interaction of these four components contributes towards the efficient functioning of a computer system built on the principles of Von Neumann Architecture.

CPU- Central Processing Unit (Processor):

The components of (CPU) include:

- **Arithmetic Logic Unit (ALU):** Conducts mathematical and logical operations.
- **Control Unit (CU):** Interprets instructions from memory and manages execution.
- **Registers:** Fast storage areas for temporary data, including the Program Counter (PC), which holds the address of the next instruction.
- **Cache:** Stores commonly accessed information to expedite access.
- **Bus Interface Unit:** Facilitates data communication between the CPU and other components.



CPU- Central Processing Unit (Processor):

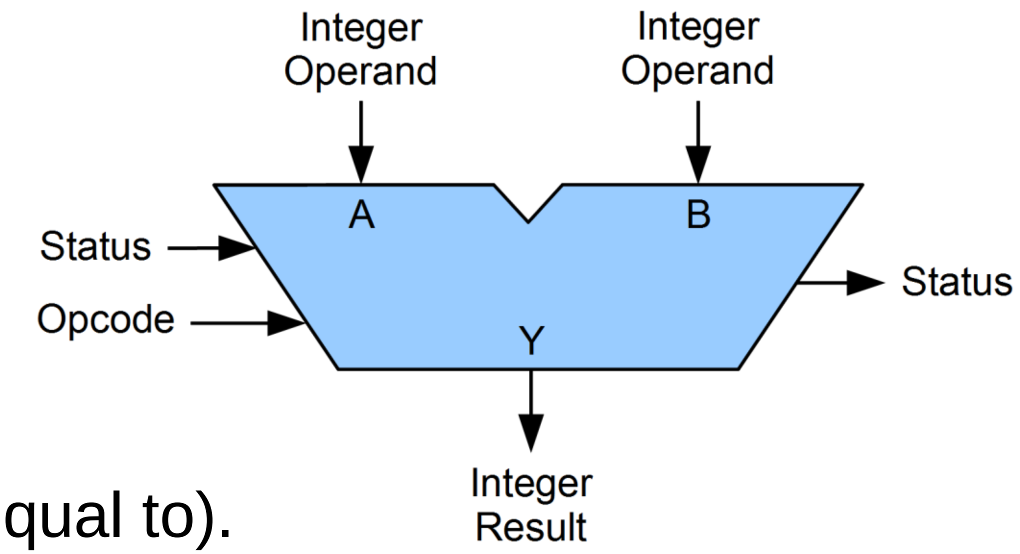
The components of (CPU) include:

- **Clock:** Synchronizes the operation of CPU components with a constant timing signal.
- Instruction Decoder: Converts instructions into signals for the CPU.
- **Memory Management Unit (MMU):** Oversees memory hierarchy and access.
- **Floating-Point Unit (FPU):** Specializes in processing floating-point operations.
- **Program Counter (PC)/Instruction Pointer (IP):** Contains the address of the current instruction being executed or to be executed next.

CPU- Central Processing Unit (Processor):

ALU (Arithmetic Logic Unit):

- A crucial component within the CPU.
- Performs computing functions involving integers.
- **Arithmetic Operations**:
  - Addition, subtraction, multiplication, division.
- **Logic Operations:**
  - Comparisons (e.g., less than, greater than, equal to).
- Numeric Processing:
  - All data reduced to numeric form.
- Constant Operation Handling:
  - Always active, handling operations rapidly.
- Result Storage:
  - Stores results in registers, memory, or outputs.



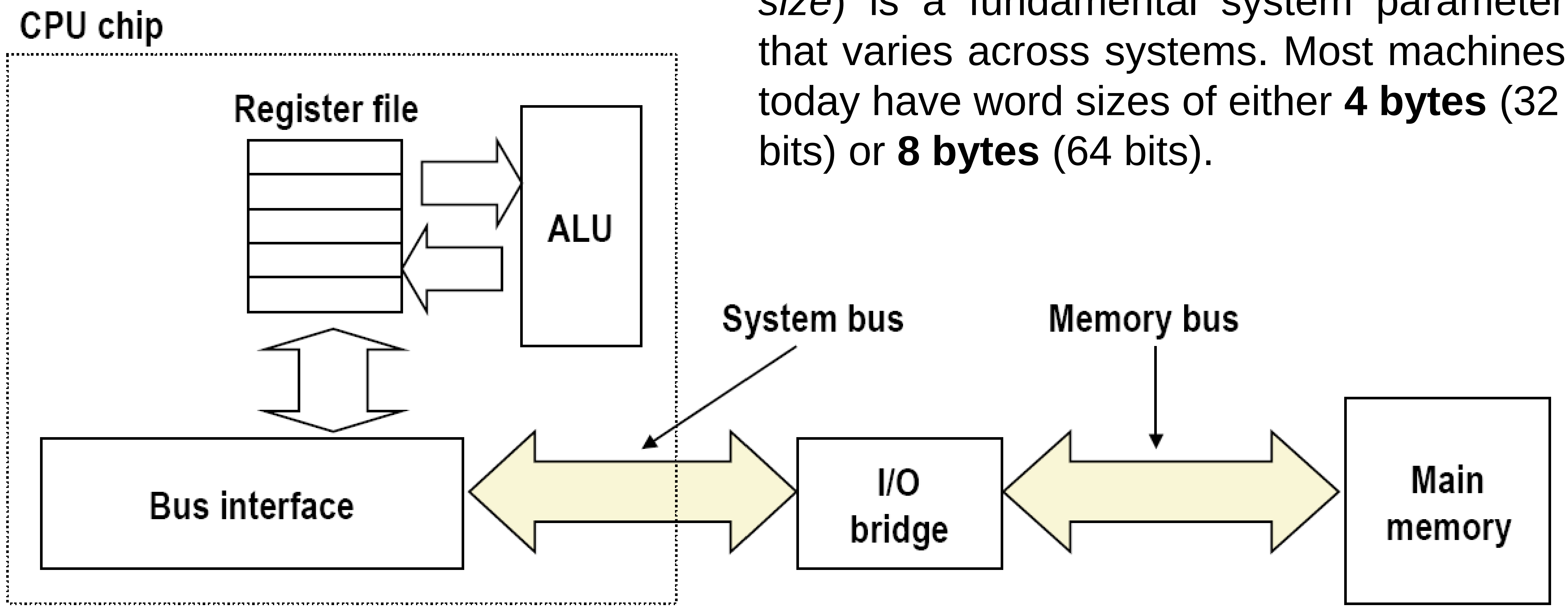
CPU- Central Processing Unit (Processor):

Basic CPU Operations:

- **Load:** Transfer data from memory to a register within the CPU.
- **Store:** Transfer data from a CPU register back into memory.
- **Operate:** Perform arithmetic or logical operations using the ALU and store results in a register.
- **Jump:** Change the flow of execution by updating the PC with a new address.

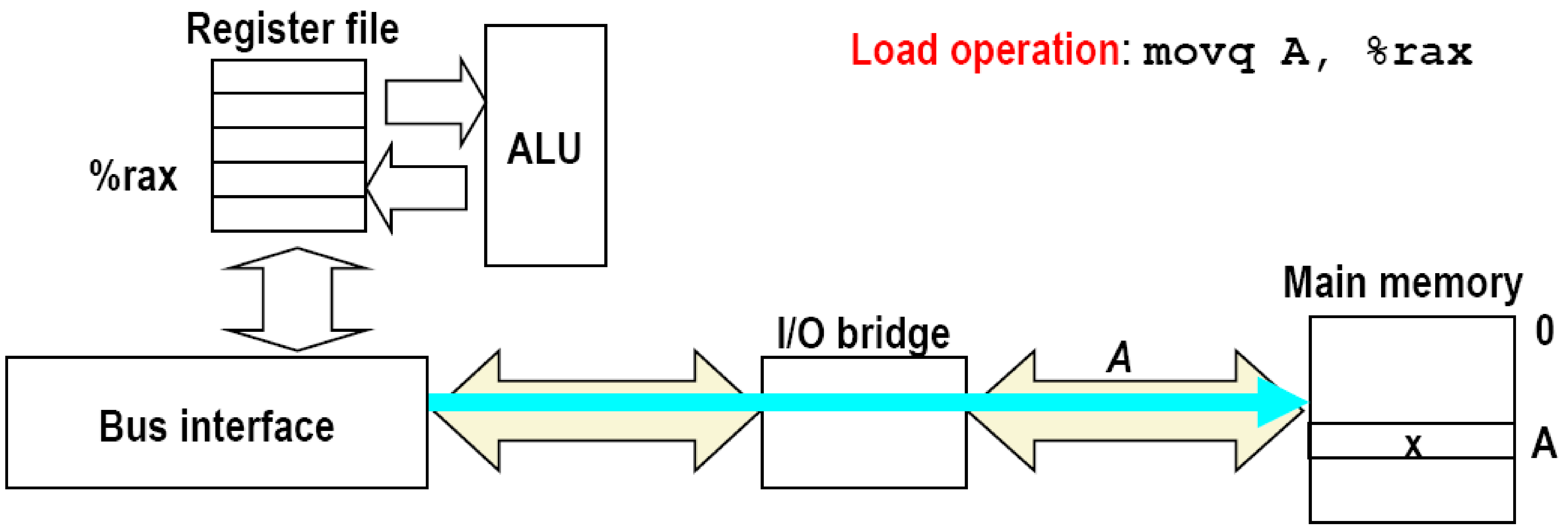
CPU-BUS-Memory:

The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. Most machines today have word sizes of either **4 bytes** (32 bits) or **8 bytes** (64 bits).



Memory Read Transition (1/3)

CPU places address A on the memory bus



Hands on:

Compose an Assembly command for the x86 32-bit architecture that writes the value located at memory address A into the accumulator

Hands on:

MOV EAX, DWORD PTR [A]

- **MOV** is the mnemonic for "move."
- **EAX** is the 32-bit accumulator register.
- **DWORD PTR** specifies that you are working with a double-word (32 bits) of data.
- **[A]** represents the memory address labeled as "A" from which you want to move

Hands on:

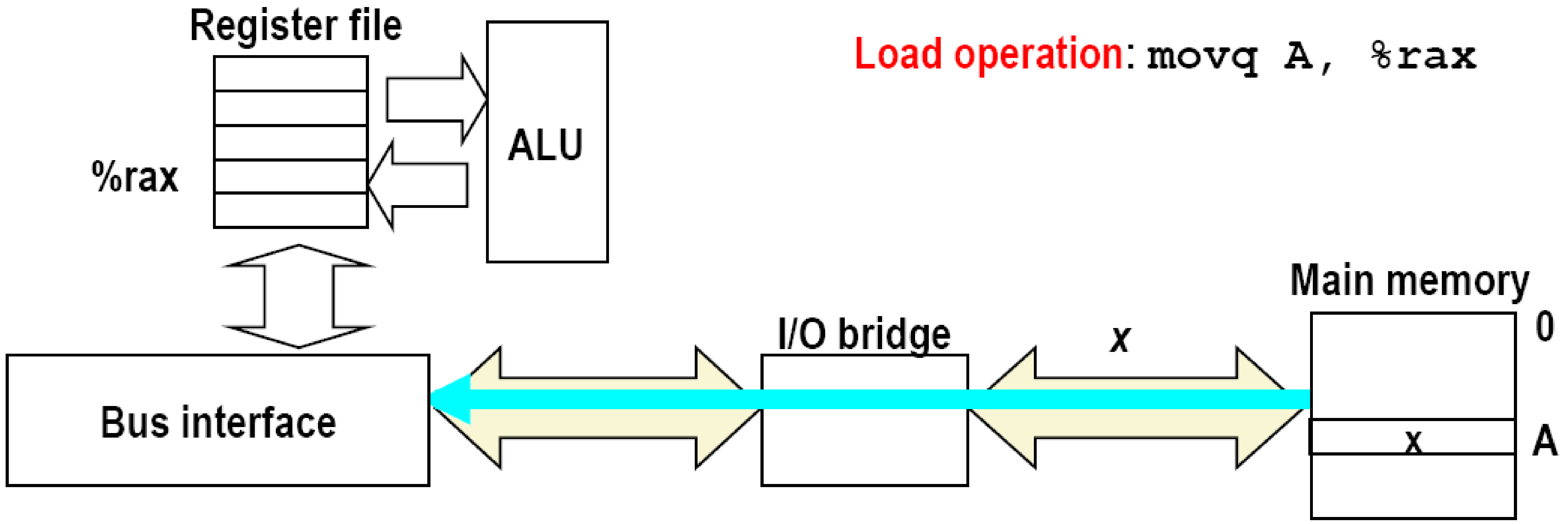
Is MOV EAX, [A] correct?

Yes. In assembly language, the instruction MOV EAX, DWORD PTR [A] explicitly tells the assembler that EAX should be loaded with a 32-bit value from the memory address labeled A. The **DWORD PTR** part is a type specifier that clarifies the size of the operand being moved.

On the other hand, **MOV EAX, [A]** also moves data from memory into EAX, but here the size of the data (in this case, 32 bits because EAX is a 32-bit register) is implied by the destination operand EAX. The assembler assumes that you are moving a value size that matches the destination register size.

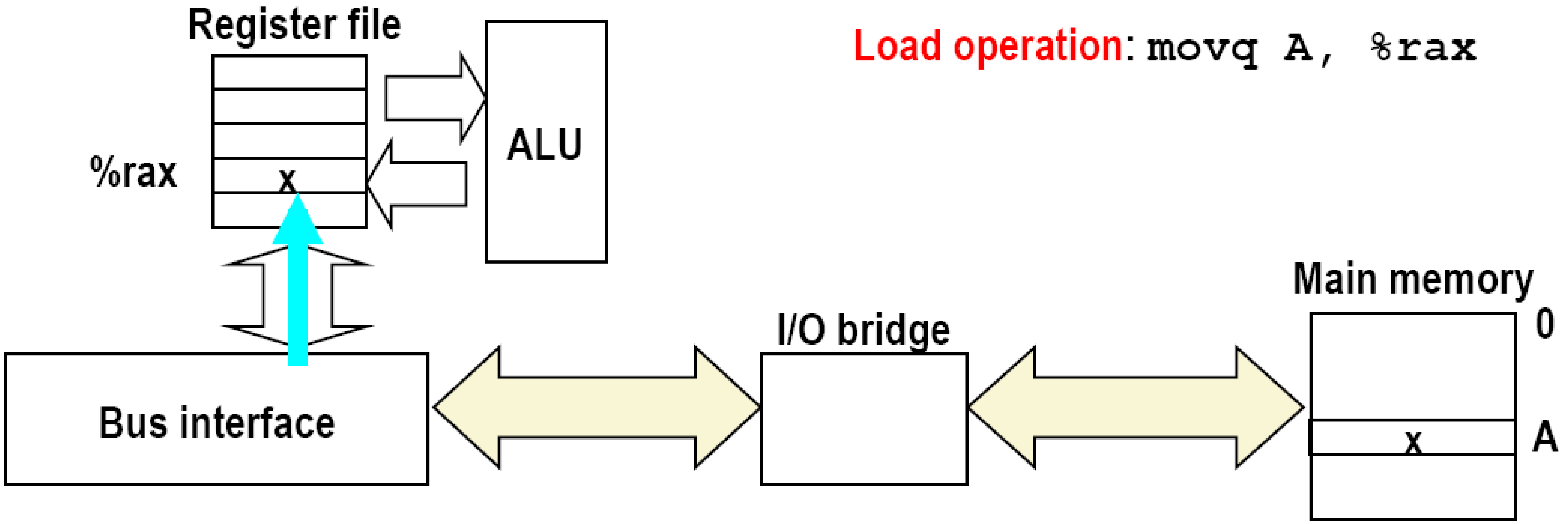
Memory Read Transition (2/3)

Main memory reads A from the memory bus, retrieves word x, and places it on the bus



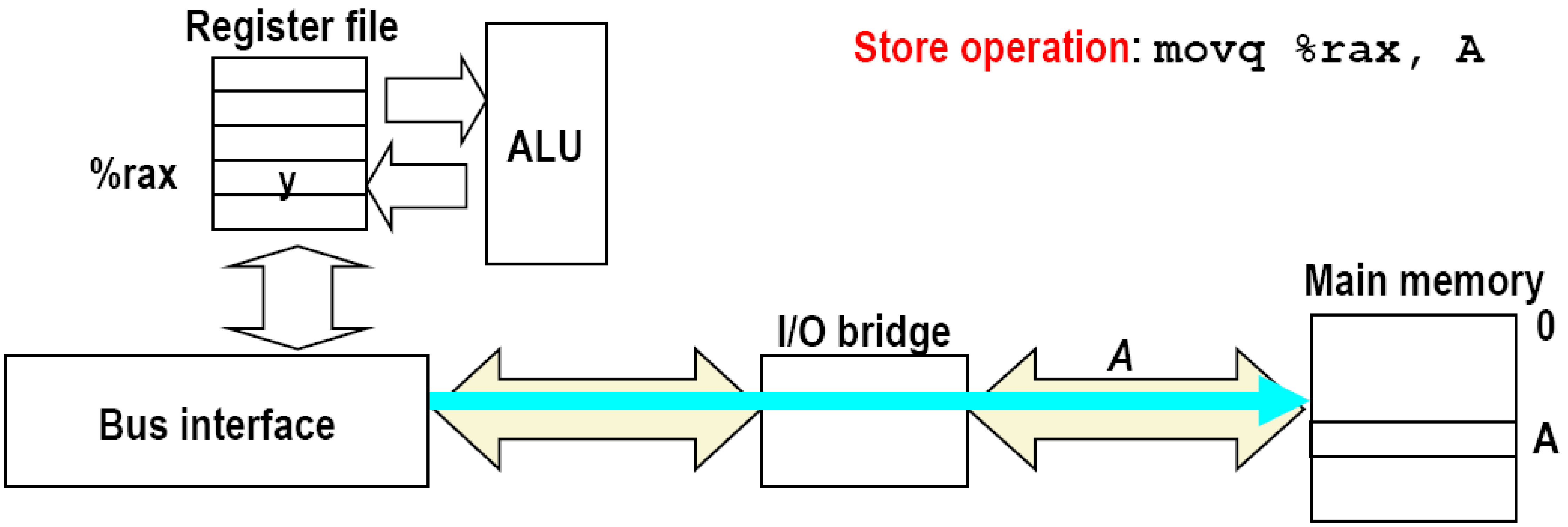
Memory Read Transition (3/3)

CPU reads word x from the bus and copies it into register %rax



Memory Write Transition (1/3)

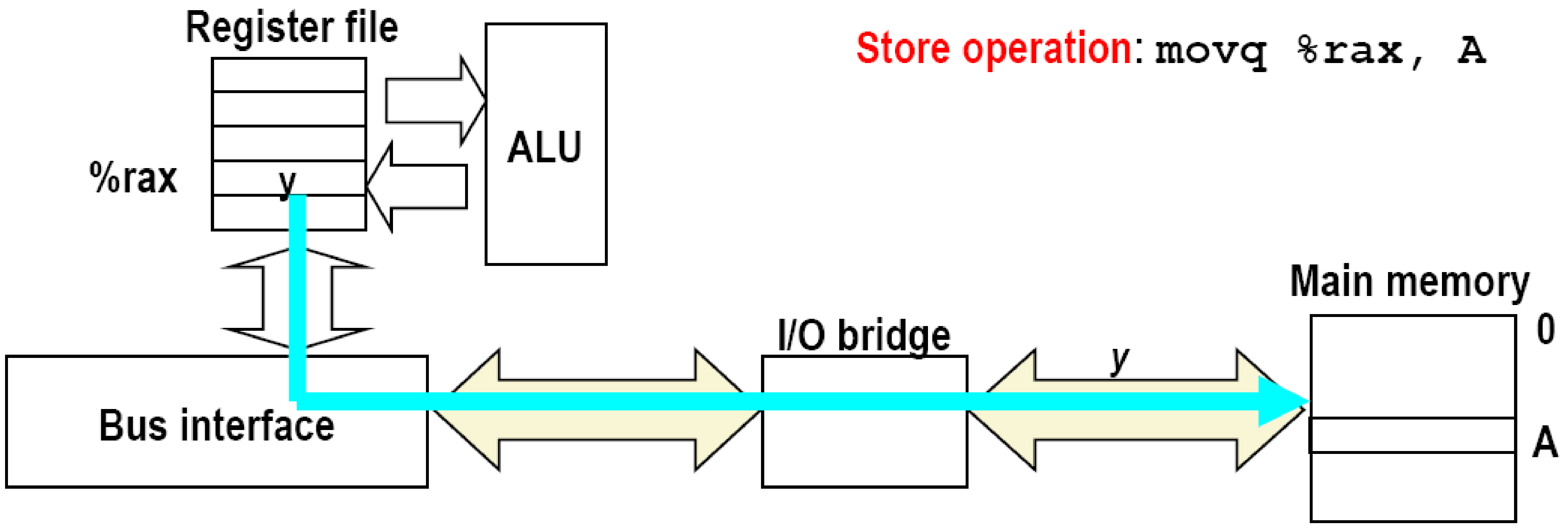
CPU puts memory address A on the bus, and main memory reads it, awaiting the corresponding data word



Memory Write Transition (2/3)

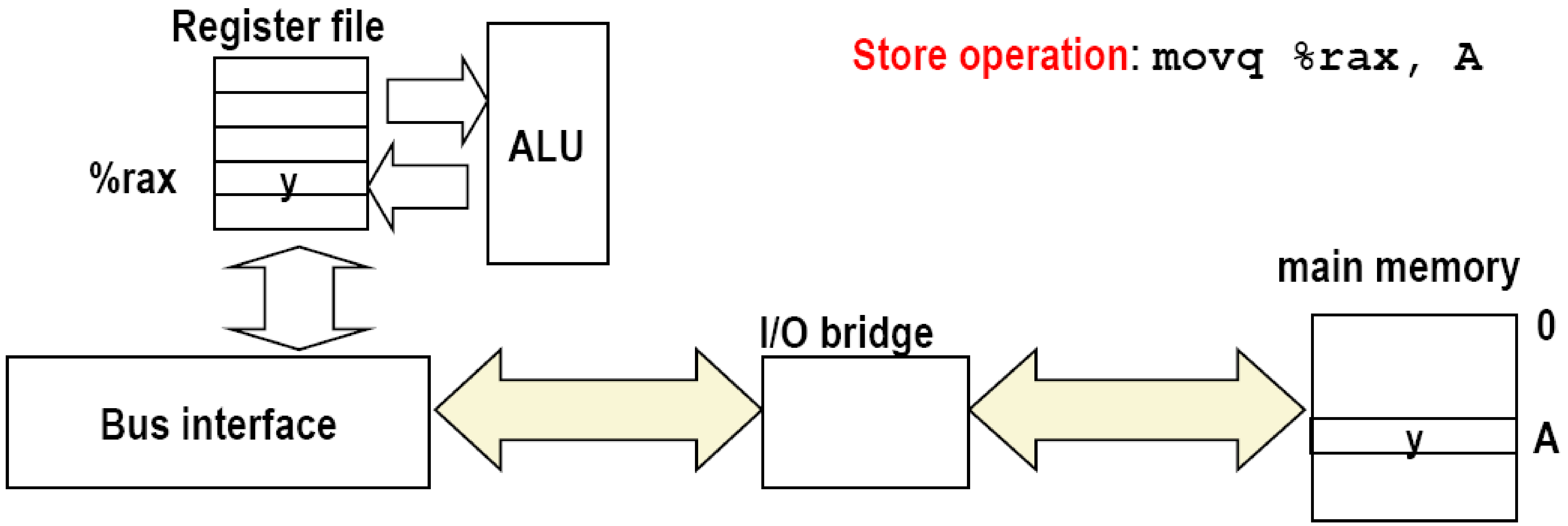
CPU puts data word 'y' on the bus

Store operation: `movq %rax, A`



Memory Write Transition (3/3)

Main memory reads data word 'y' from the bus and stores it at address A



Store operation: `movq %rax, A`

Hands on:

Perform data manipulation tasks in x86 assembly language:

Load Data from Memory:

- Retrieve the data stored at memory address “A”
- Place this data into the 'eax' register.

1. **Perform Arithmetic Operation:**
   - Add the value **10** to the data stored in the “eax” register.
   - Store the result back in the “eax” register.
2. **Write the Result Back to Memory:**
   - Send the result (the updated value in 'eax') back to the same memory address “A”
3. **Determine ALU Operation: Explain the role of the ALU in these operations.**

Hands on:

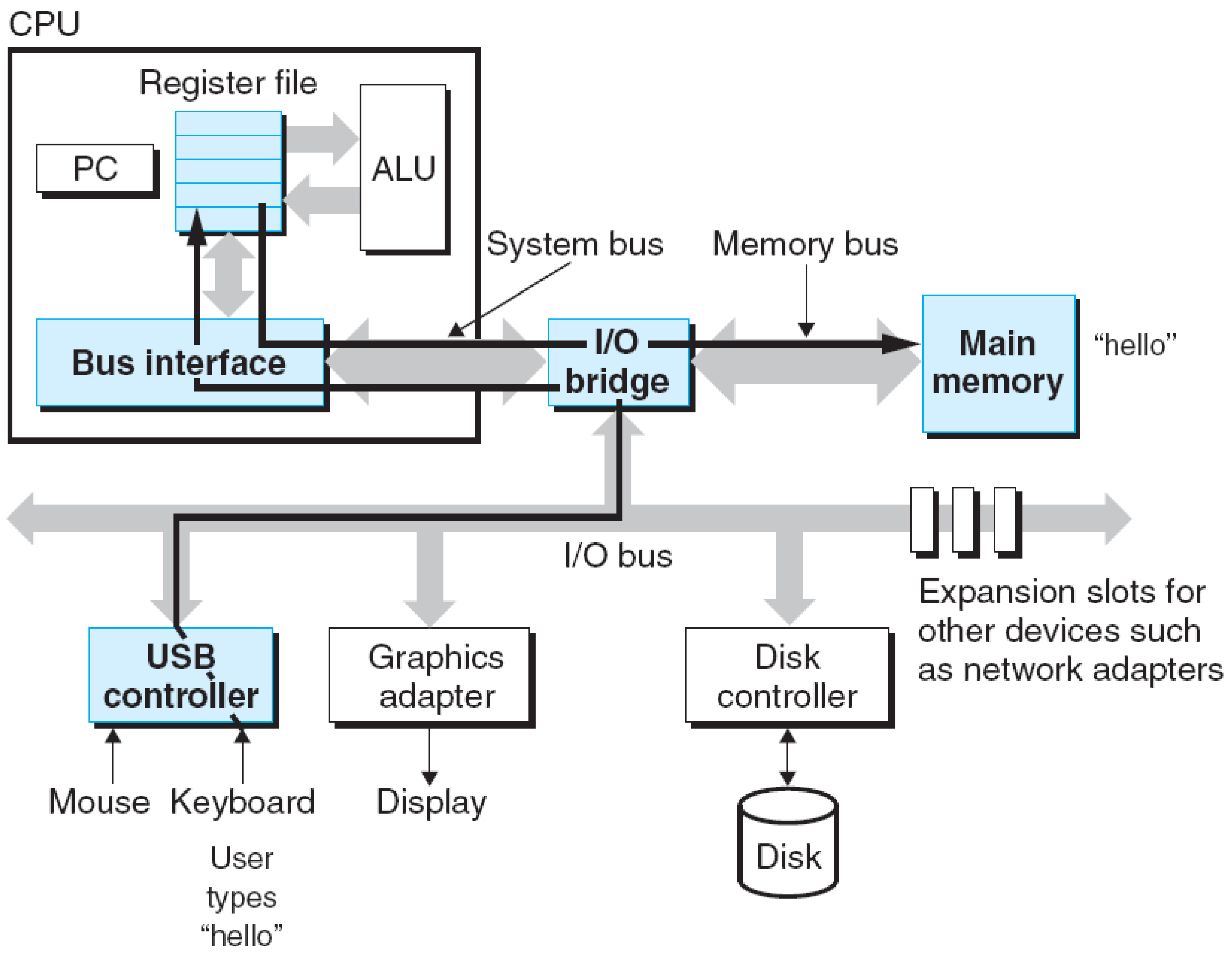
```
section .data
    A dd 42

section .text
global main

main:
    mov eax, dword [A]
    add eax, 10
    mov dword [A], eax

section .bss
```

The **PTR** keyword is optional and often omitted when the **operand size is clear** from the context. Including PTR is more about making the intention explicit for clarity, which can be especially useful in more complex assembly language programs where the operand size might not be immediately apparent.

Running the hello Program

Figure 1.5
Reading the `hello` command from the keyboard.

Typing the Command (Figure 1.5): When you type **./hello** at the keyboard, the shell program reads each keystroke and stores the command in memory. The keystrokes are processed by the CPU after being received by the USB controller for the keyboard.



Running the hello Program

Loading the Program (Figure 1.6): The hello program is loaded from the disk into the main memory using Direct Memory Access (DMA). This process bypasses the CPU, allowing the data to move directly from the disk to the main memory.

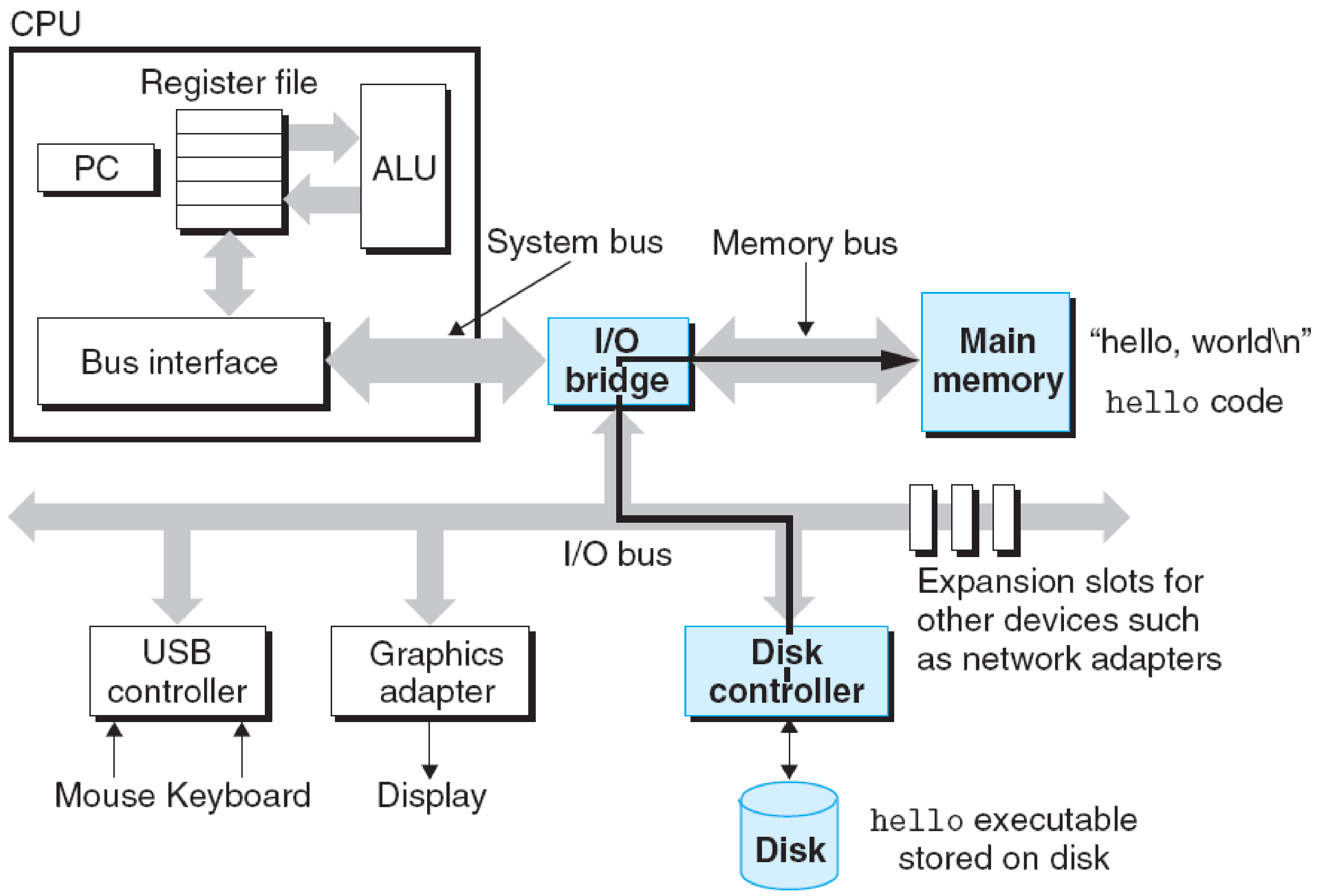


Figure 1.6 Loading the executable from disk into main memory.

Running the hello Program

Executing the Program (Figure 1.7): With the program loaded into main memory, the CPU begins executing its instructions. It copies the "hello, world\n" string from memory into the CPU's registers and then to the display through the graphics adapter, resulting in the text appearing on your screen.

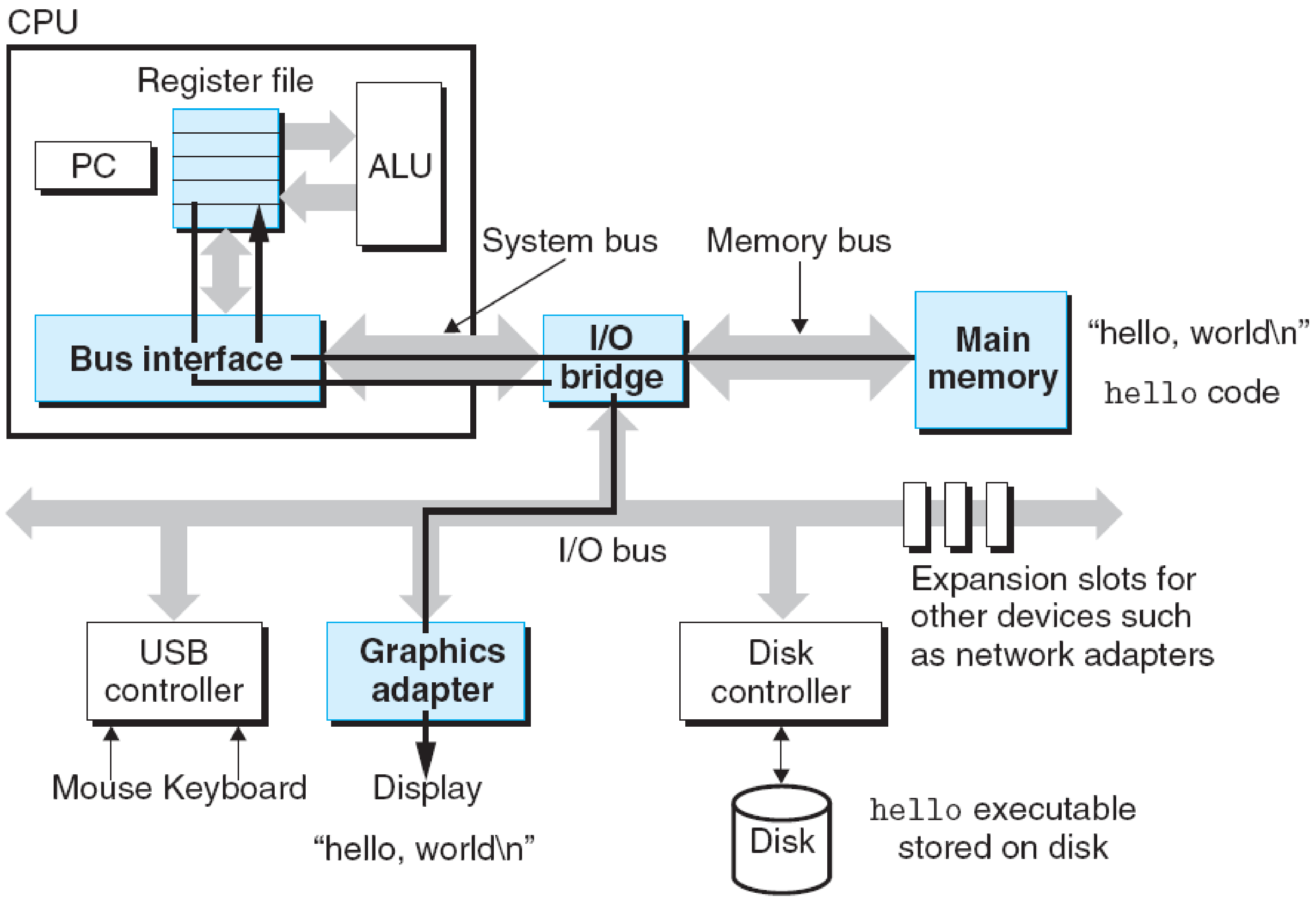
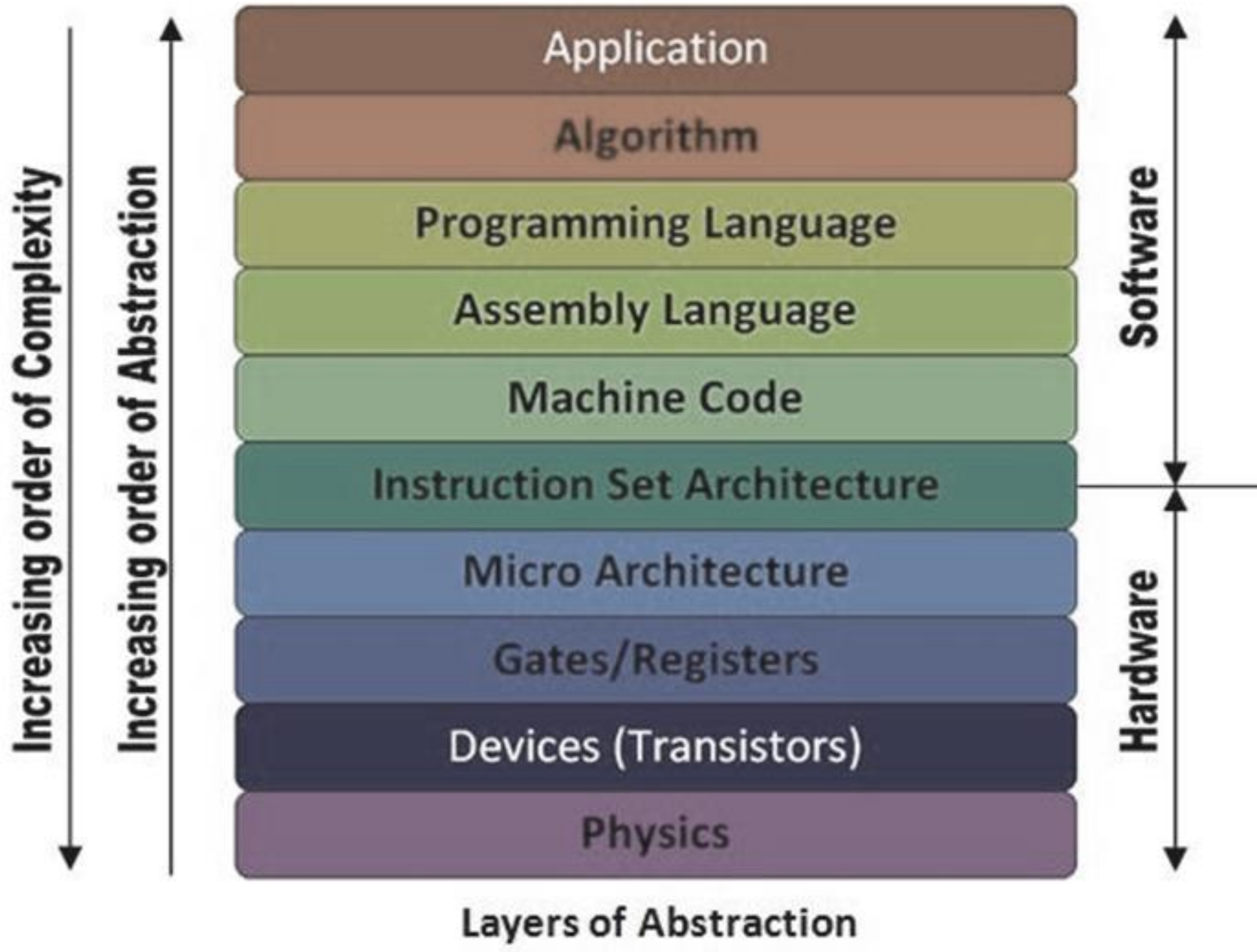


Figure 1.7 Writing the output string from memory to the display.

CPU Design & Architecture

Levels of Abstraction in Computer Systems

The software layers include applications, algorithms, programming languages, assembly language, and machine code. Below these are the hardware layers: the Instruction Set Architecture (ISA), which is implemented by the microarchitecture, followed by the gates and registers made from transistors, and ultimately grounded in physics. The division between software and hardware is also highlighted, with ISA serving as a bridge between the two.



The Instruction Set Architecture (ISA)

Have you ever wondered how your software programs tell your computer what to do?

- The ISA acts as an **interface** between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.
  - **Instructions:** The actual commands that tell the CPU what operations to perform, such as arithmetic operations, moving data, or controlling other hardware functions.
  - **Registers:** Small, fast storage locations in the CPU that instructions use to operate quickly on data.
  - **Data Types:** Defines the types of data the CPU can process, like integers, floating-point numbers, and others.
  - **Addressing Modes:** Methods for specifying operands for the CPU instructions.
  - **I/O Model:** Defines how the CPU interacts with the system's input/output mechanisms.

The Instruction Set Architecture (ISA)

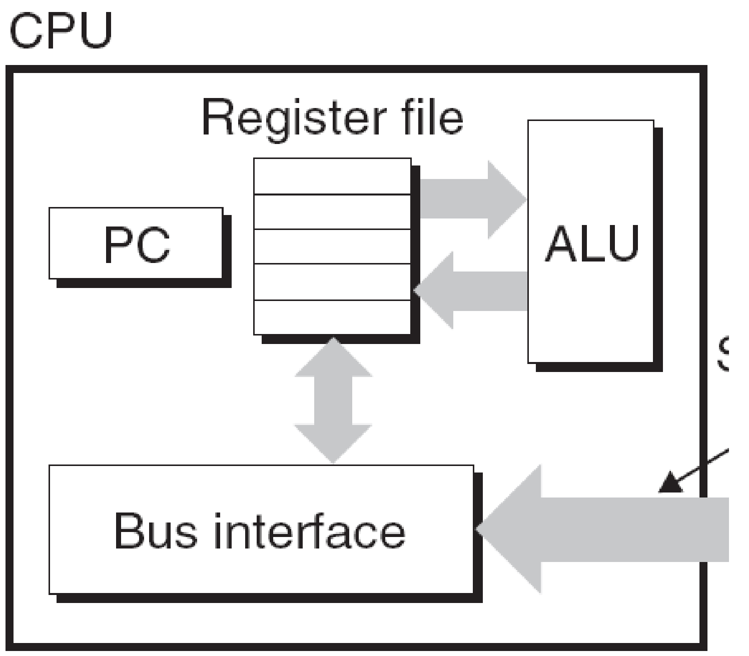
Have you ever wondered how your software programs tell your computer what to do?

- The ISA acts as an **interface** between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.
  - **Instructions:** The actual commands that tell the CPU what operations to perform, such as arithmetic operations, moving data, or controlling other hardware functions.
  - **Registers:** Small, fast storage locations in the CPU that instructions use to operate quickly on data.
  - **Data Types:** Defines the types of data the CPU can process, like integers, floating-point numbers, and others.
  - **Addressing Modes:** Methods for specifying operands for the CPU instructions.
  - **I/O Model:** Defines how the CPU interacts with the system's input/output mechanisms.

The Instruction Set Architecture (ISA)

Microarchitecture:
Microarchitecture, sometimes referred to as computer organization, is essentially how a particular ISA is translated into an actual silicon design, or the physical circuitry, on a processor. It's the layer that gives the ISA a physical form and function.

This includes the design of various subsystems such as the arithmetic logic unit (ALU), control unit (CU), memory hierarchies (registers, L1/L2/L3 caches), and more.



The Instruction Set Architecture (ISA)

Have you ever wondered how your software programs tell your computer what to do?

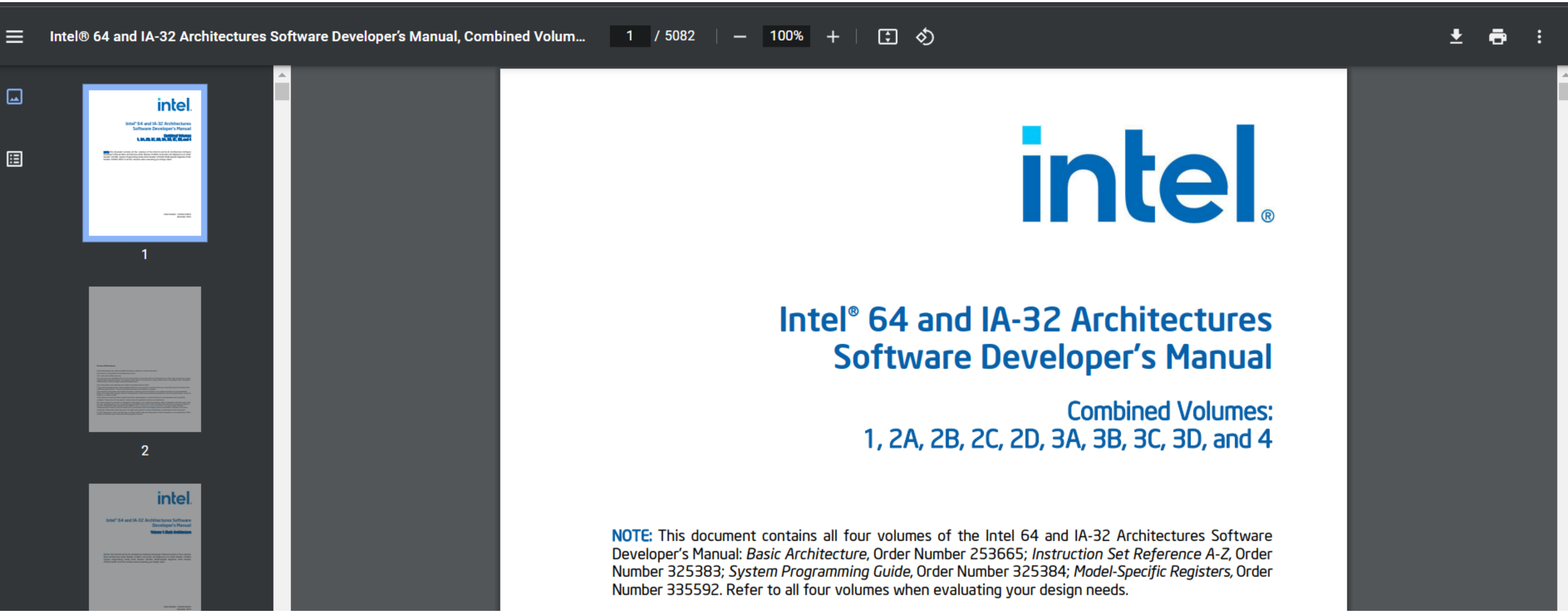
Example ISAs:

Intel x86: One of the most common ISAs for desktops and laptops, known for a complex set of variable-length instructions (**CISC**).

ARM: Used in most smartphones and tablets, known for its simple, fixed-length instructions designed for efficiency (**RISC**).

MIPS: Another RISC ISA often used in academic settings for teaching due to its simplicity and regularity.

- .



Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volum... 1 / 5082 100%

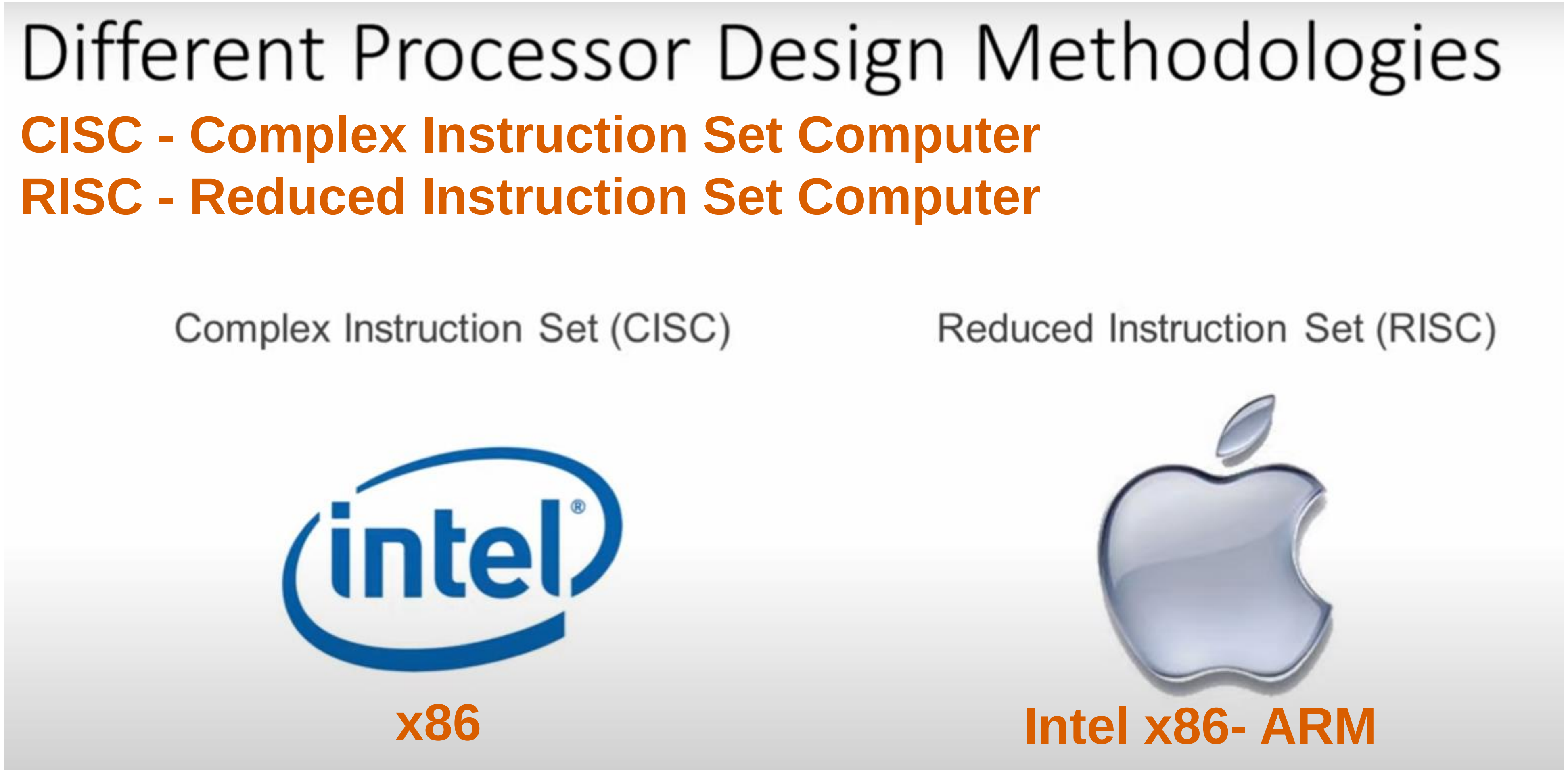
Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

CISC and RISC



Different Processor Design Methodologies

CISC - Complex Instruction Set Computer
RISC - Reduced Instruction Set Computer

Complex Instruction Set (CISC)	Reduced Instruction Set (RISC)
x86	Intel x86- ARM

CISC and RISC

RSIC: ARM Assembly Instructions	CISC: x86 Assembly Instruction
LDR R3, [R3] ADDS R2, R3, #1 STR R2, [R3]	INC dword ptr [eax]

- In ARM, it requires three separate instructions to load the value into a register (LDR), increment the register (ADD), and store the value back into memory (STR).

- In contrast, x86 can increment a value in memory directly with a single instruction like INC or ADD, thanks to its capability to perform certain operations directly on memory without the need for a register intermediary.

CISC and RISC

Complexity: CISC has complex instructions capable of performing multiple operations in one instruction, while RISC has simpler, more uniform instructions.

Execution Speed: RISC typically executes instructions faster due to their simplicity, whereas CISC may take more cycles per instruction due to complexity.

Instruction Length: CISC instructions vary in length, whereas RISC instructions are of a fixed size.

Number of Instructions: A task might require fewer instructions in CISC due to their complexity, but more in RISC.

Hardware Design: RISC design is generally simpler, leading to potentially more efficient use of the silicon area.

Usage: CISC is common in personal computers (e.g., Intel's x86 architecture), while RISC is widely used in mobile devices (e.g., ARM architecture).

RIT

Registers

Registers

Registers in a CPU are a key element of the Instruction Set Architecture (ISA) because they are **the fastest type of memory** where the CPU stores and retrieves the data that is immediately needed for execution.

The ISA defines:

- The number and size of registers available.
- The specific functions of each register (e.g., data, address, general-purpose, or special-purpose registers).
- How instructions reference these registers for performing operations.

BASIC PROGRAM EXECUTION REGISTERS

IA-32 architecture provides 16 basic program execution registers for use in general system and application programing. These registers can be grouped as follows:

1. **General-purpose registers :**These eight registers are available for storing operands and pointers.
2. **Segment registers:** These registers hold up to six segment selectors.
3. **EFLAGS (program status and control) register** :The EFLAGS register report on the status of the program being executed and allows limited (application-program level) control of the processor.
4. **EIP (instruction pointer) register:** The EIP register contains a 32-bit pointer to the next instruction to be executed.

Rochester Institute of Technology of Dubai

BASIC PROGRAM EXECUTION REGISTERS

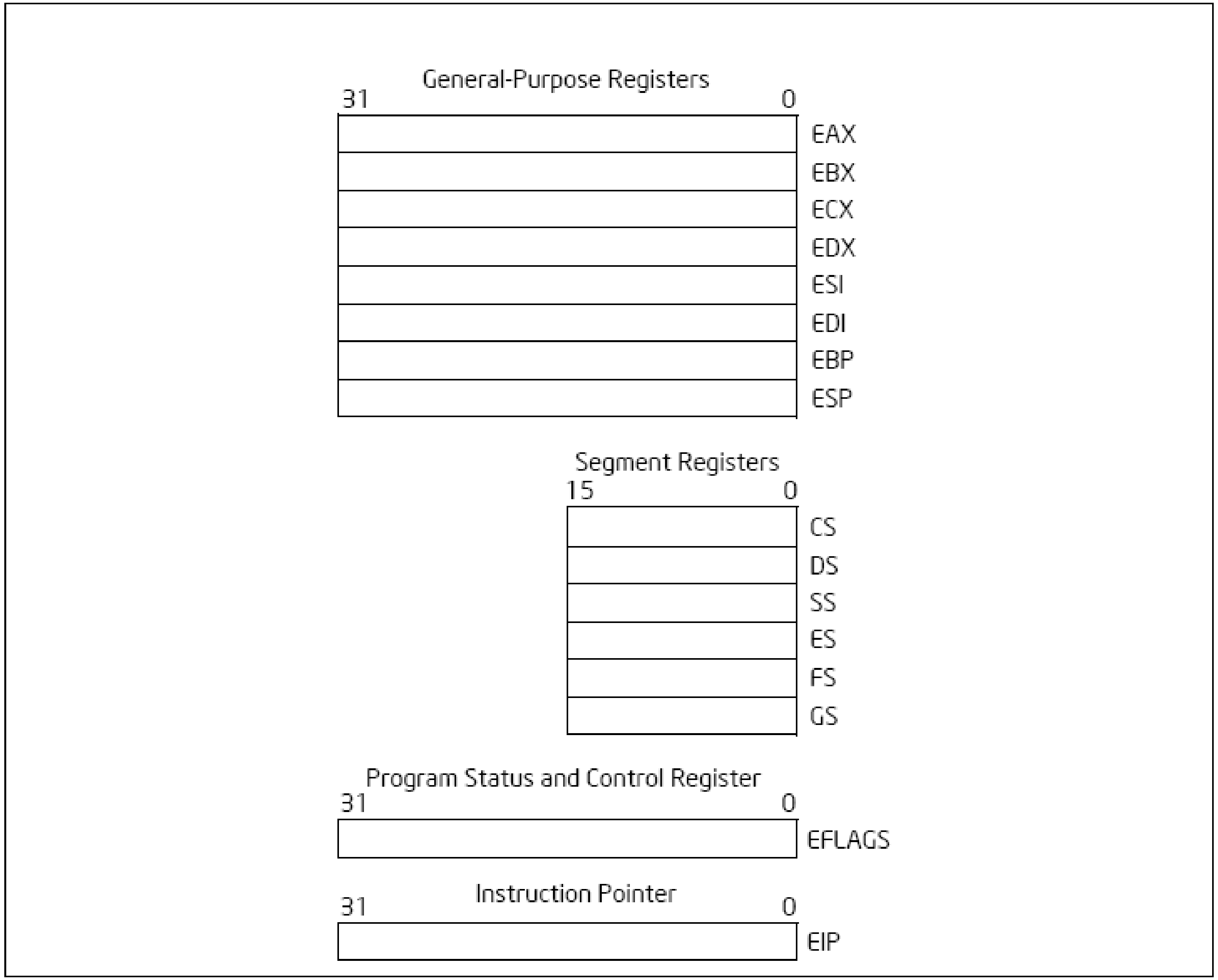
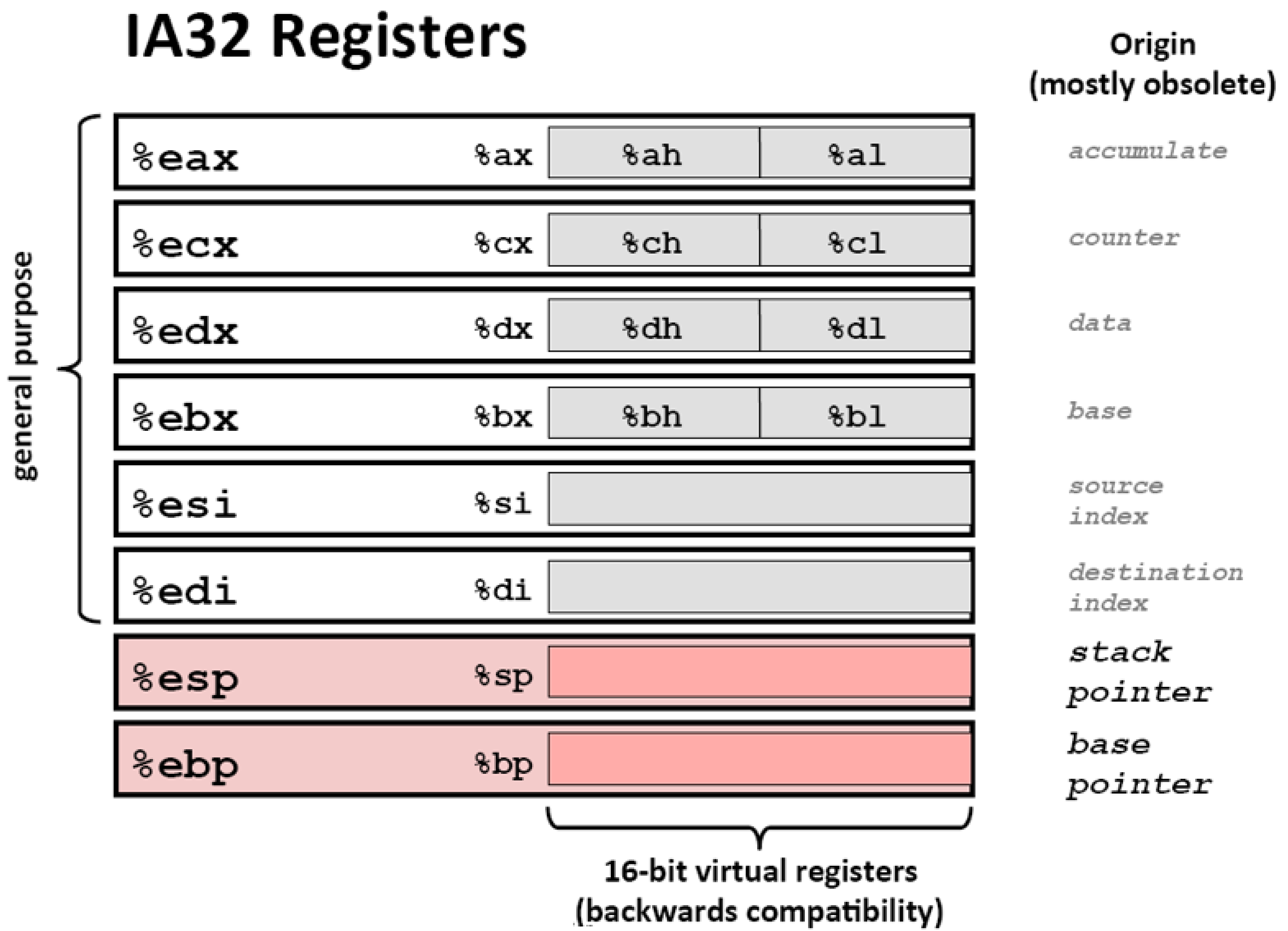


Figure 3-4. General System and Application Programming Registers

General-Purpose Registers

Overview of 32-bit general-purpose registers in x86 architecture: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.
Primary uses: storing operands for operations, address calculations, and memory pointers.



General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic **operations.**
- Operands for **address** calculations.
- Memory **pointers**.

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the **ESP** register.

The ESP register holds the stack pointer and as a general rule should not be used for another purpose.

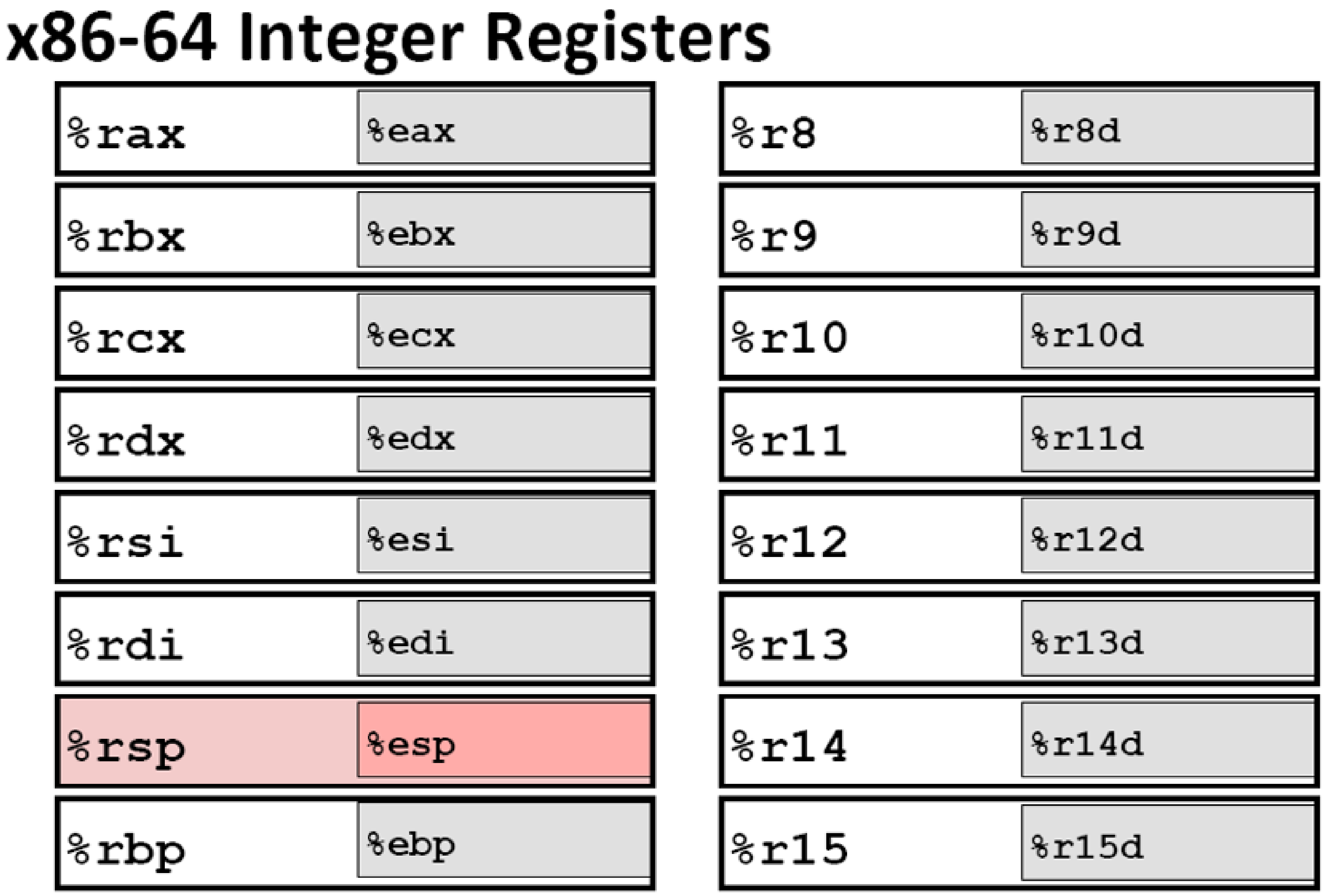
General-Purpose Registers

In the x86 architecture, each general-purpose register has typical uses:

1. **EAX:** Accumulator for operands and results; often used in arithmetic operations.
2. **EBX:** Base register; often used in indexed addressing.
3. **ECX:** Counter for string and loop operations.
4. **EDX:** Data register; extends the accumulator for certain operations.
5. **ESI:** Source index for string operations.
6. **EDI:** Destination index for string operations.
7. **EBP**: Base pointer for stack frames.
8. **ESP:** Stack pointer; points to the top of the stack.

General-Purpose Registers

The registers have been extended in the x86-64 architecture, indicated by an 'R' prefix instead of 'E'. For example, the 32-bit 'EAX' register becomes 'RAX' in 64-bit, expanding from 32 bits to 64 bits. This allows for more efficient processing and handling of larger data types and values, enabling more complex computations and addressing a larger memory space.



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

Segment Register

The segment registers (CS, DS, SS, ES, FS, and GS) hold **16-bit segment** selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

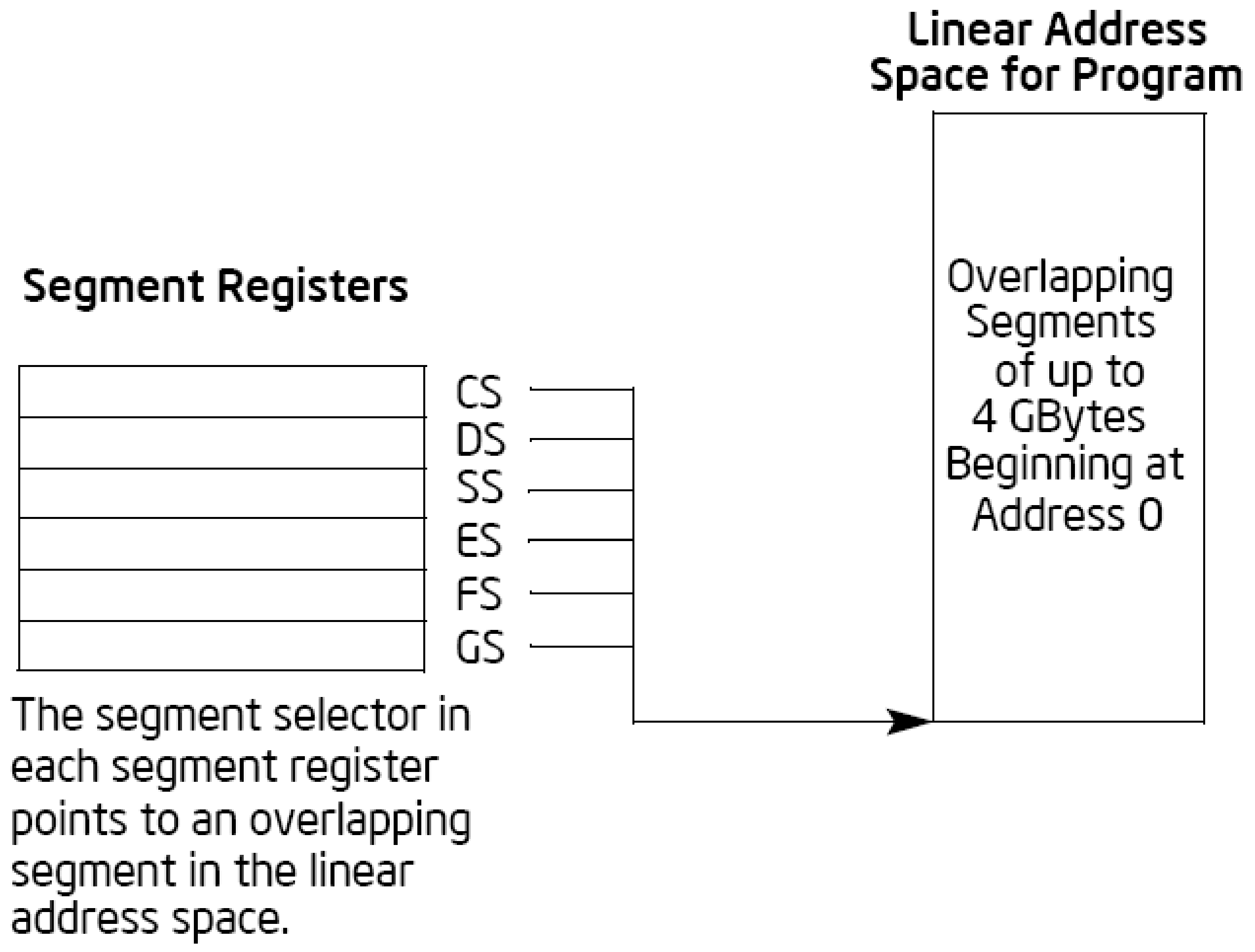

Linear Address Space for Program
Segment Registers
CS
DS
SS
ES
FS
GS
Overlapping Segments of up to 4 GBytes Beginning at Address 0
The segment selector in each segment register points to an overlapping segment in the linear address space.

Figure 3-6. Use of Segment Registers for Flat Memory Model

Segment Registers

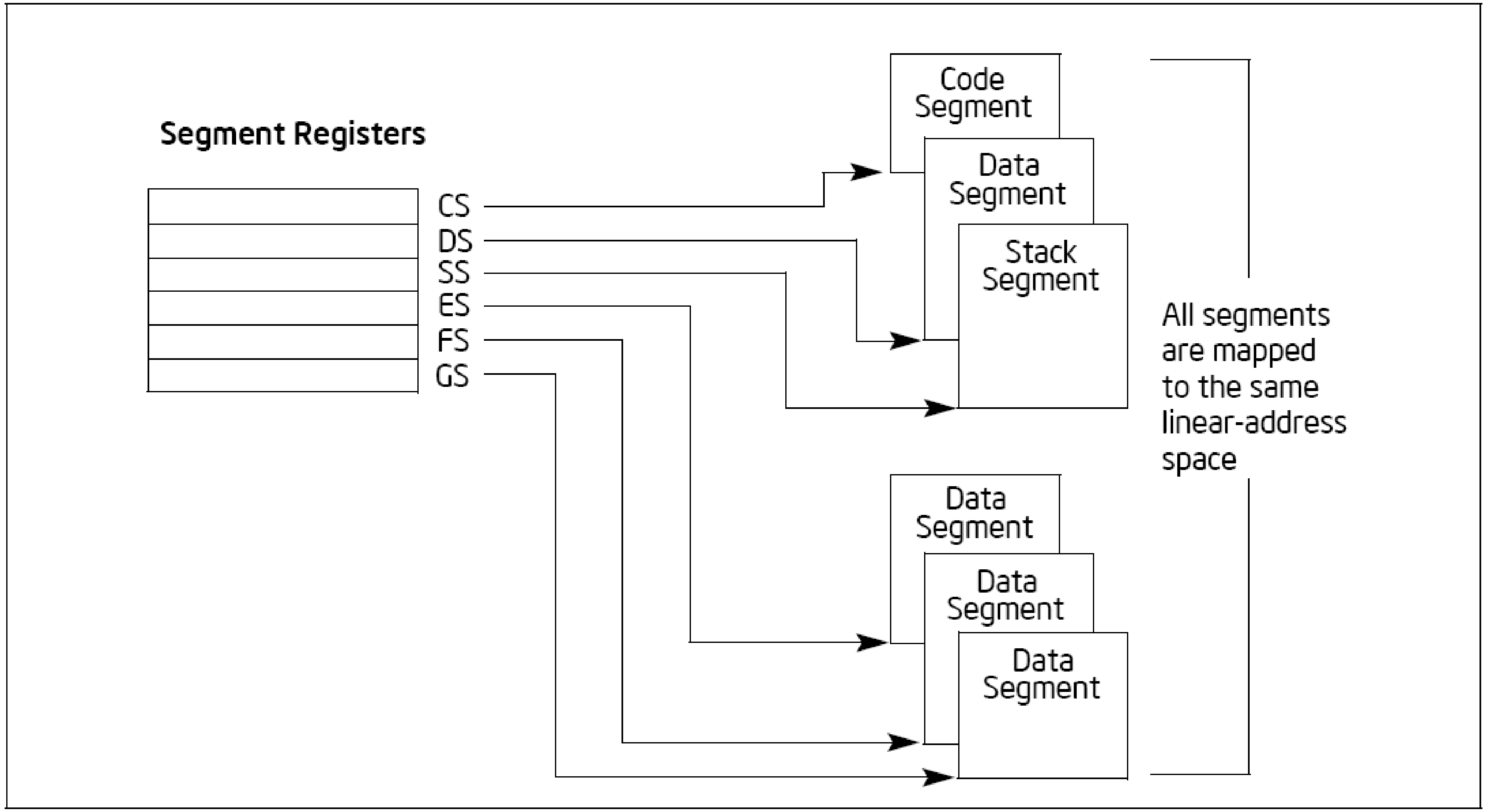
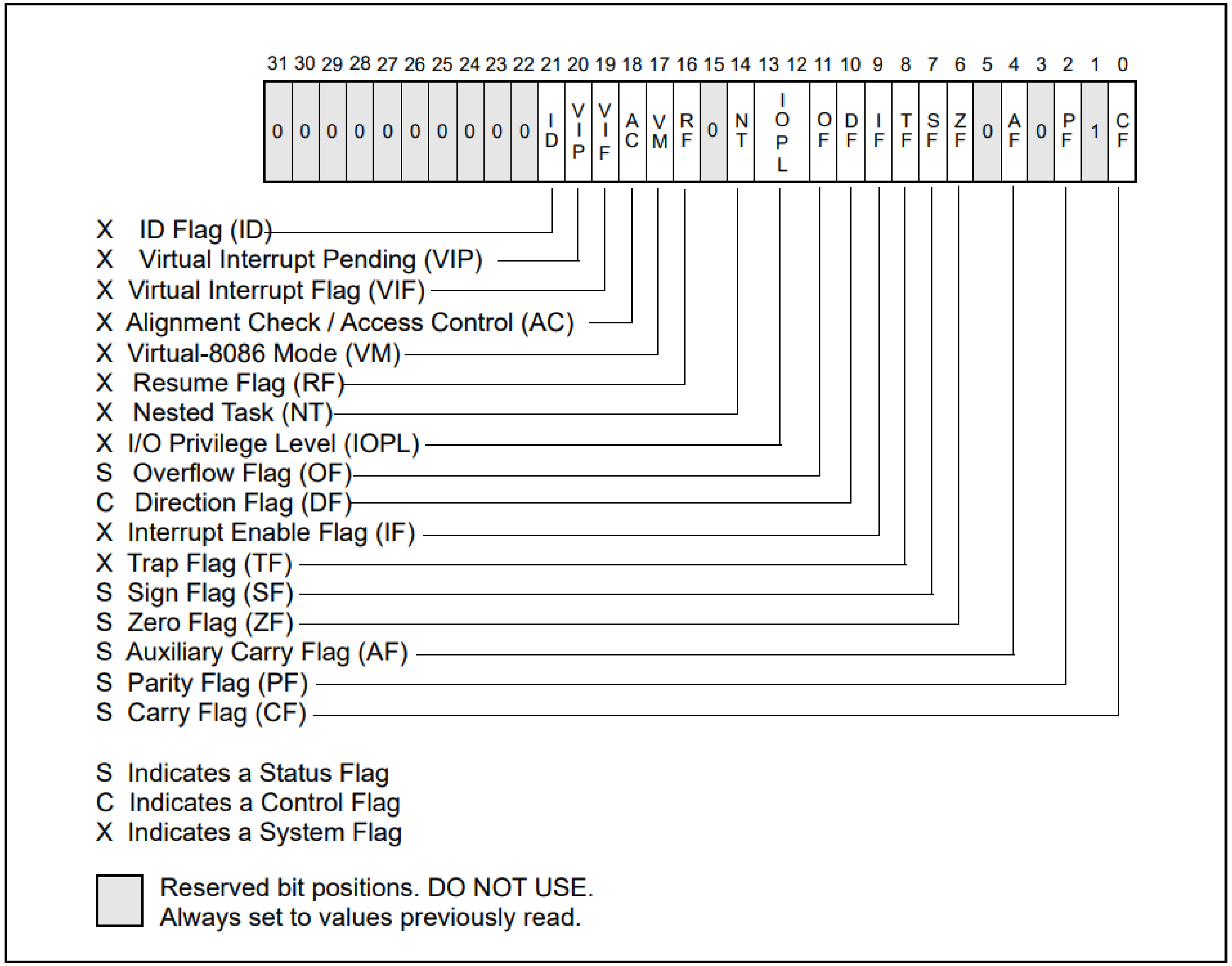


Figure 3-7. Use of Segment Registers in Segmented Memory Model

EFLAGS Register

The EFLAGS register in x86 architecture is a special register that contains a collection of flags used by the processor to indicate the status of operations and to control certain processor functions.



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE. Always set to values previously read.

Figure 3-8. EFLAGS Register

EFLAGS Register

The EFLAGS register contains status flags that reflect the outcome of arithmetic operations. These flags include:

- **CF (Carry Flag):** Indicates if an arithmetic operation has resulted in a carry out of the most significant bit, relevant for unsigned integer arithmetic.
- **PF (Parity Flag):** Shows if the low-order byte of the result has an even number of set bits.
- **AF (Auxiliary Carry Flag):** Set if there's a carry out of the least significant nibble, used for BCD arithmetic.
- **ZF (Zero Flag):** Set if the result of an arithmetic operation is zero.
- **SF (Sign Flag):** Reflects the sign of the result of an operation; set if negative.
- **OF (Overflow Flag):** Indicates an overflow in signed integer arithmetic.

EFLAGS Register

1. CF (Carry Flag):
ADD AL, 0xFF ; If AL contains 0x01 or greater, this will set the CF because the result wraps around.

2. ZF (Zero Flag):
XOR EAX, EAX ; This will set ZF as the result is zero.

3. SF (Sign Flag):
SUB EAX, EAX ; Followed by DEC EAX. EAX will be 0xFFFFFFFF, setting the SF.

4. OF (Overflow Flag):
ADD EAX, 0x7FFFFFFF ; If EAX also contains 0x7FFFFFFF, adding them will set the OF.

Check Your Understanding:

The Zero Flag (ZF) is set when the result of an operation is zero. Which instruction always sets the ZF?

a) XOR EAX, EAX

b) ADD EAX, 1

c) SUB EAX, EAX

d) CMP EAX, EAX

Check Your Understanding:

If the Sign Flag (SF) is set after an instruction, what can be inferred about the result?

a) It is zero.
b) It is a positive number.
c) It is a negative number.
d) An unsigned overflow occurred

Check Your Understanding:

Which instruction would you use to compare two values and set flags accordingly without changing the values?

a) ADD
b) SUB
c) XOR
d) CMP

Check Your Understanding:

What is the result of the following instruction in terms of the Zero Flag (ZF)?

SUB EAX, EAX

a) ZF is cleared.
b) ZF is set.
c) ZF is unchanged.
d) ZF state is unpredictable.

Check Your Understanding:

```
MOV EAX, 5
CMP EAX, 5
JE Label
```

After executing the above instructions, what will be the state of the Zero Flag (ZF) when the comparison happens?

a) The ZF will be set (ZF = 1) because EAX is not equal to 5.
b) The ZF will be cleared (ZF = 0) because EAX is equal to 5.
c) The ZF will be set (ZF = 1) because EAX is equal to 5.
d) The ZF state is unpredictable.

RIT

Memory

RIT

...0x2BCON10U

Course Overview

- **Title: "CSEC 202 - Reverse Engineering Fundamentals"**

Instructor	Office	Phone	Email	Semester-Year
Emad Abu Khousa	B217		eakcad@rit.edu	Spring-2024
Office Hours:	M: 12:00-01:00 TR: 11:00-12:00			

•	**600:**	**TR**	**12:05-01:20,**	**Room D-114**
•	**601:**	**MW**	**01:05-02:25,**	**Room C-109**
•	**602:**	**TR**	**01:30-02:50,**	**Room D-207**

RIT

Thank You and Q&A