The Microprocessor Unit

CTEC1332
Software Engineering Practices
2023 Fall

Niagara College Canada
APPLIED DREAMS.

Computer (Review from CTEC1184)

- A **computer** is a programmable machine designed to sequentially and automatically carry out a sequence of arithmetic or logical operations.
- The particular sequence of operations can be changed readily, allowing the computer to solve more than one kind of problem.

Source: https://en.wikipedia.org/wiki/Computer

Microprocessor

- A **microprocessor** incorporates the functions of a computer's **central processing unit** (CPU) on a single **integrated circuit** (IC).

- It is a multi-purpose, **programmable** device that accepts digital data as **input**, processes it according to **instructions** stored in its memory, and provides results as **output**.

Source: https://en.wikipedia.org/wiki/Transistor_count

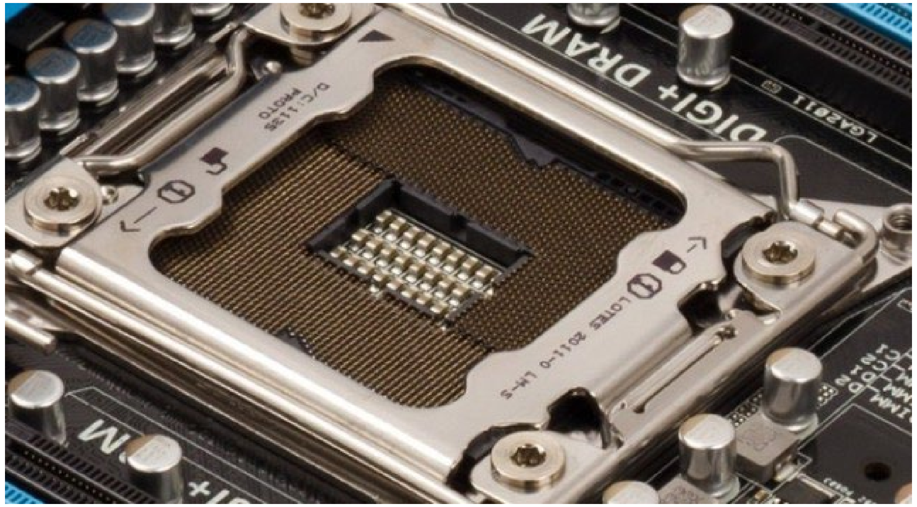
MPU

- The **microprocessor unit** (MPU) is classified as a complex, VLSI (very large scale integration) device.

- **VLSI** refers to the number of component parts that can be packed into the chip.

- In 2020, a microprocessor is comprised of several **billion** such components…

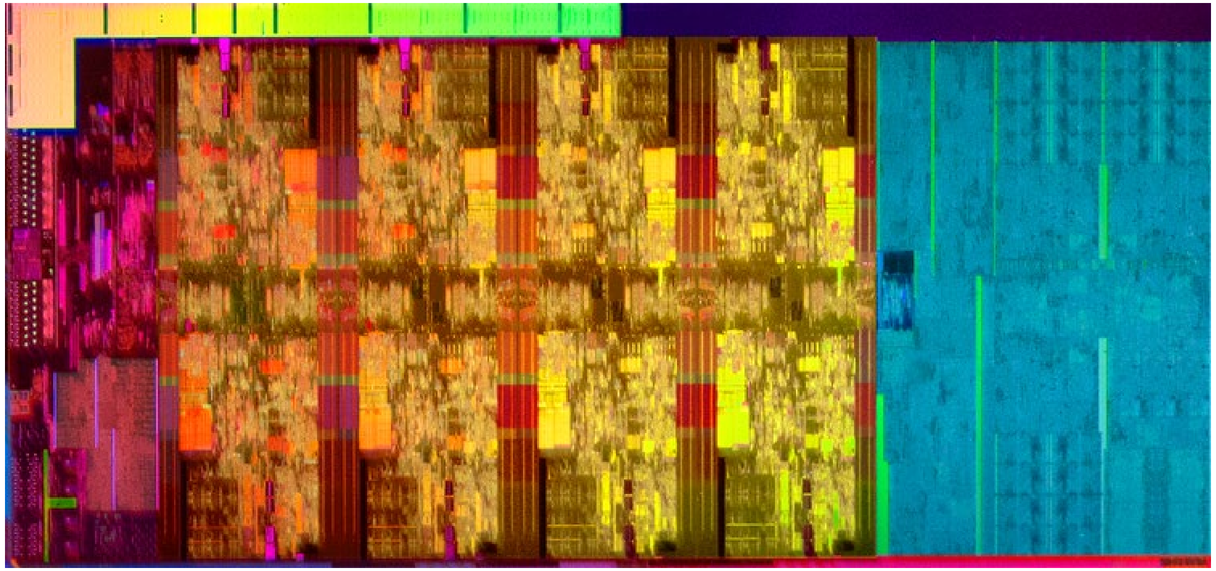
MPU



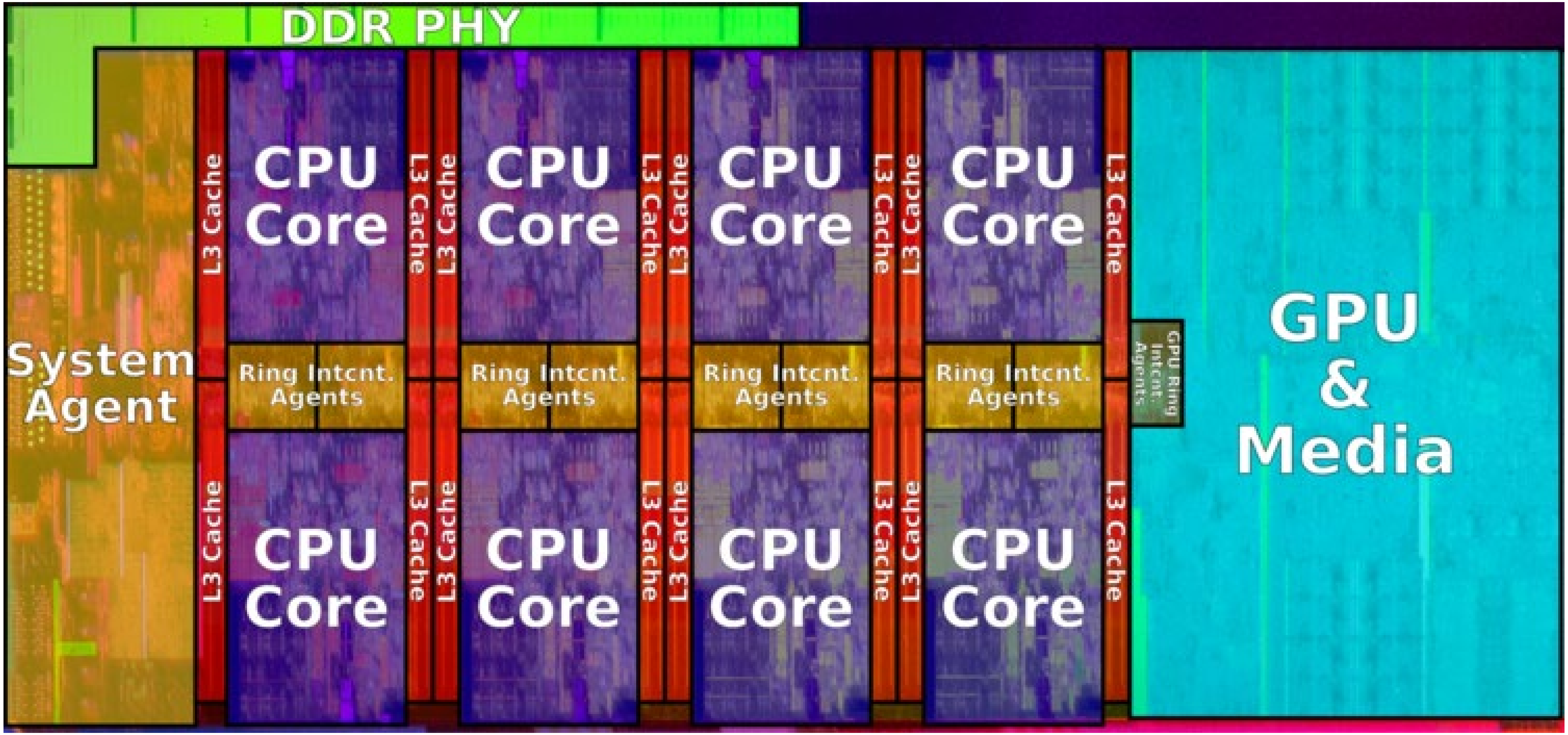
- The microprocessor circuit is attached to the **motherboard** by a multipin connector **socket**.

- Often, just the name of the MPU is used to describe the status of a whole computer system – for example, 2.4 GHz i9.

MPU – Chip Die (colourized)



MPU Chip Layout



MPU Functions

- The MPU's main functions are often summed up as being the “brains” of the computer but the functions are, more specifically,
  - addressing and data transfer,
  - decision making,
  - timing and control,
  - arithmetic and logic operations,

MPU Functions

- The MPU's main functions are often summed up as being the brains of the computer but the functions are, more specifically,
  - fetching data and instructions from memory,
  - decoding instructions,
  - reporting on computer status,
  - responding to control signals (reset and interrupts) from I/O devices.

Evolution of the Microprocessor Chip

A brief chronology of microprocessor chip development is

Year	Make & Model	Bits	Transistor Count
1971	Intel 4004	4	2250
1978	Intel 8086	16	29000
1979	Intel 8088	16/8	29000
1982	Intel 80286	16	134000
1985	Intel 80386	32	275000
1989	Intel 80486	32	1180235
1992	DEC Alpha 21064	64	1680000

Evolution of the Microprocessor Chip

A brief chronology of microprocessor chip development is

Year	Make & Model	Bits	Transistor Count
2003	AMD K8	64	105,900,000
2004	Intel Pentium 4	32	112,000,000
2006	Intel Core 2 Duo	64	291,000,000
2007	AMD K10	64	463,000,000
2018	Qualcomm Snapdragon 8cx	64	8,500,000,000
2019	Apple A13	64	8,500,000,000
2019	AMD Ryzen 9 3900X	64	9,890,000,000

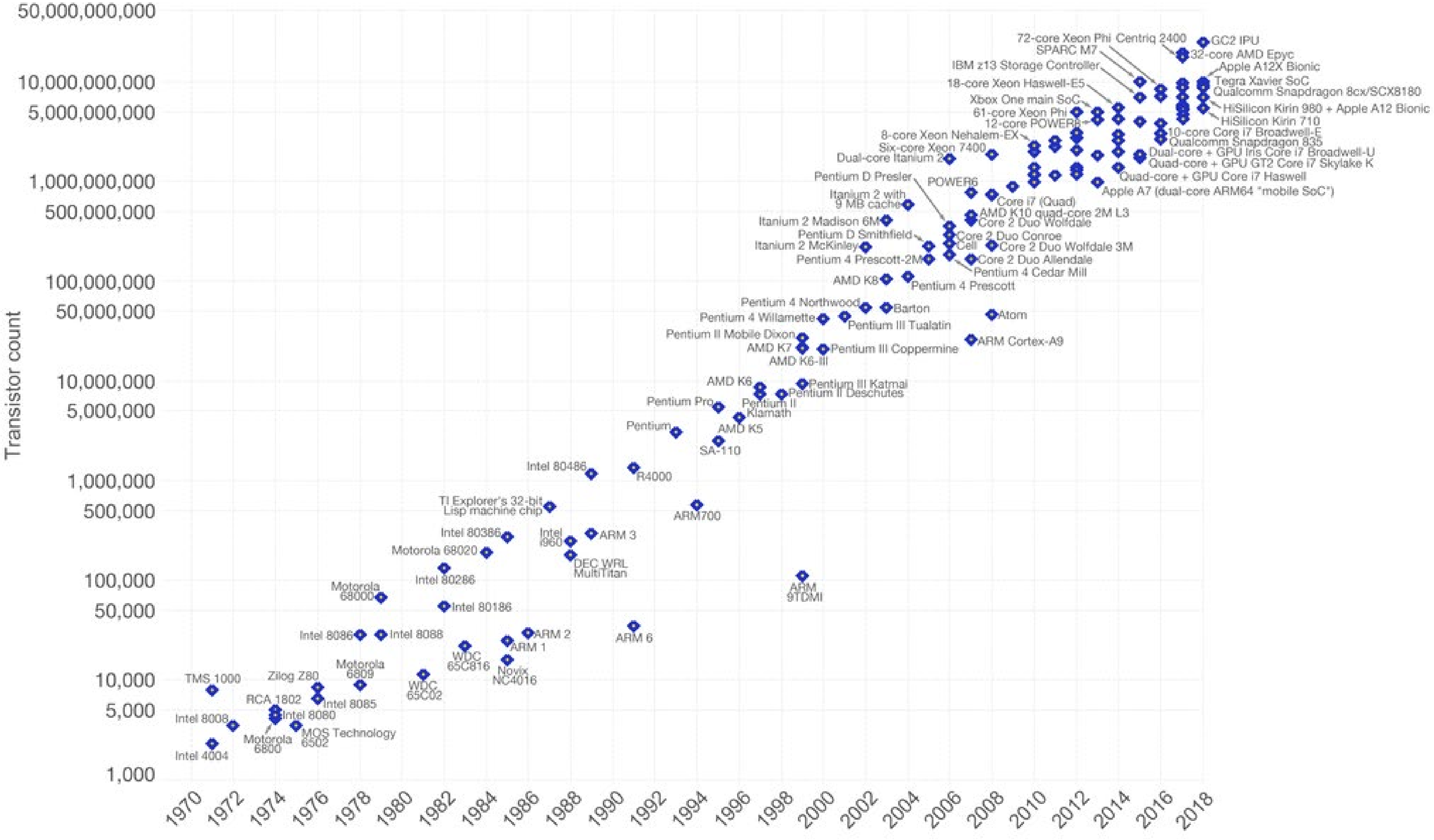
Evolution of the Microprocessor Chip: Present Day (2023)

- Intel: 13th Generation: Core i3, i5, i7, i9 (13000 series)
- AMD: Zen 3+, 4: Ryzen 5, 7, 9 (7000 series)
- ARM: Cortex A520/A740/X4
  - Qualcomm Snapdragon 7, 8, 8+; G1/G2/G3X; Microsoft SQ2
  - Apple A16 Bionic, M1 Ultra, M2/M2 Pro/M2 Max/M2 Ultra
  - Exynos (Samsung) 1300 series

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

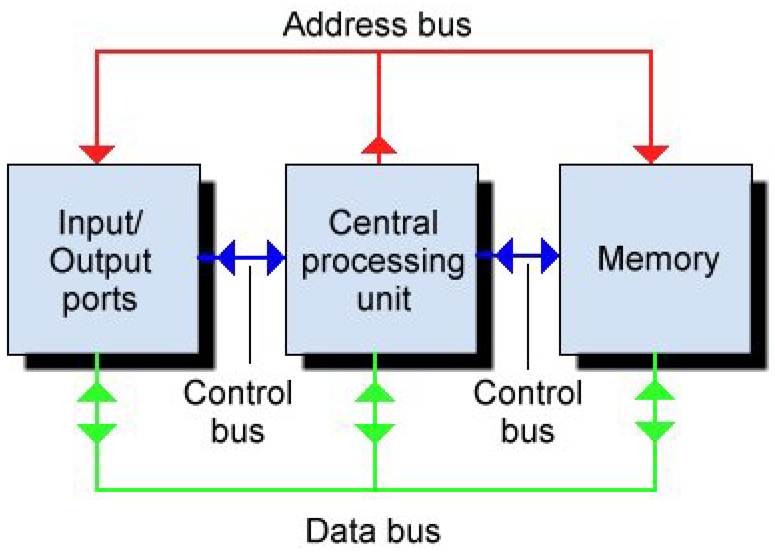
Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

MPU Architecture – Bus Structure

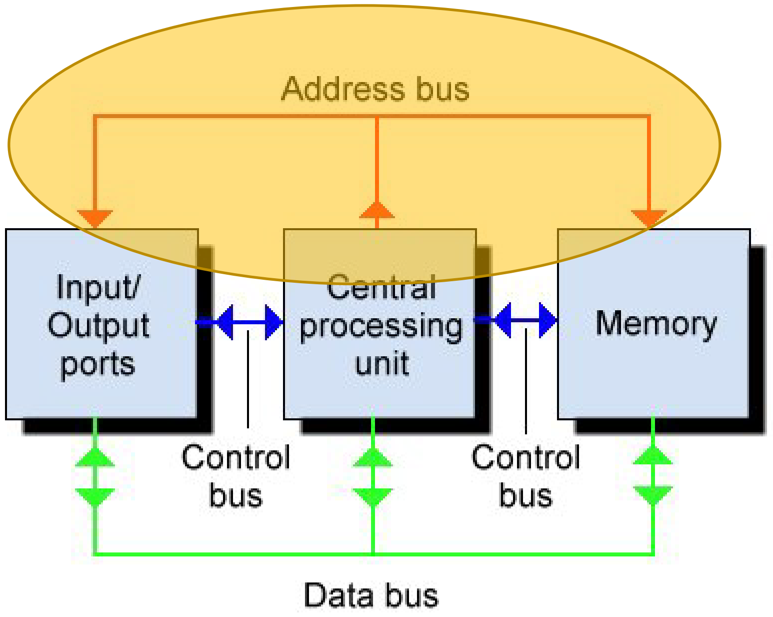


The MPU contains three buses – address, data and control.

A **bus** is a communication system that transfers data between components inside a computer.

https://en.wikipedia.org/wiki/Bus_(computing)

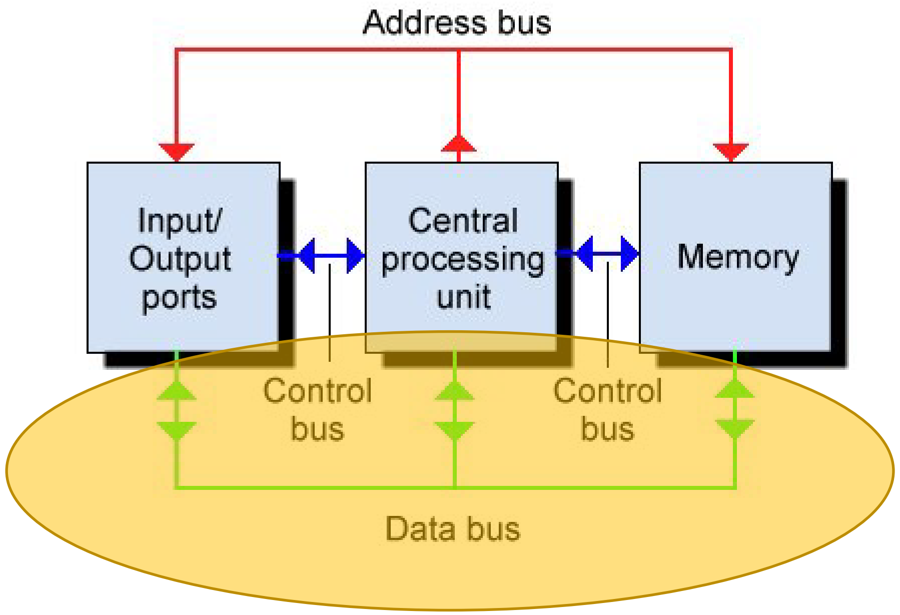
MPU Architecture – Bus Structure



The MPU contains three buses -- address, data and control:

1. **Address** bus. The MPU selects addresses of memory locations and I/O devices and places them on the address bus (one at a time).

MPU Architecture – Bus Structure

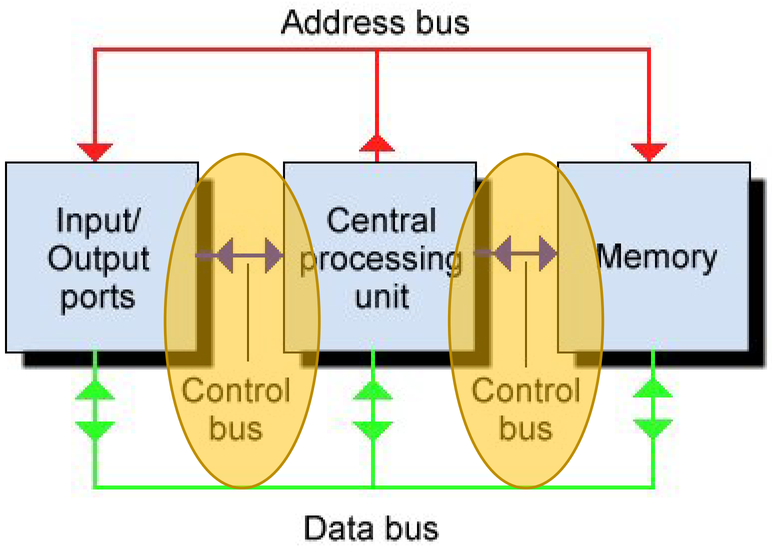


The MPU contains three buses -- address, data and control:

2. **Data bus**. This bidirectional bus is used to transfer information along the microprocessor, memory, and I/O ports.

   Many different devices are connected to this bus, but only one at a time is activated by either a chip select or chip enable signal.

MPU Architecture – Bus Structure



The MPU contains three buses -- address, data and control:

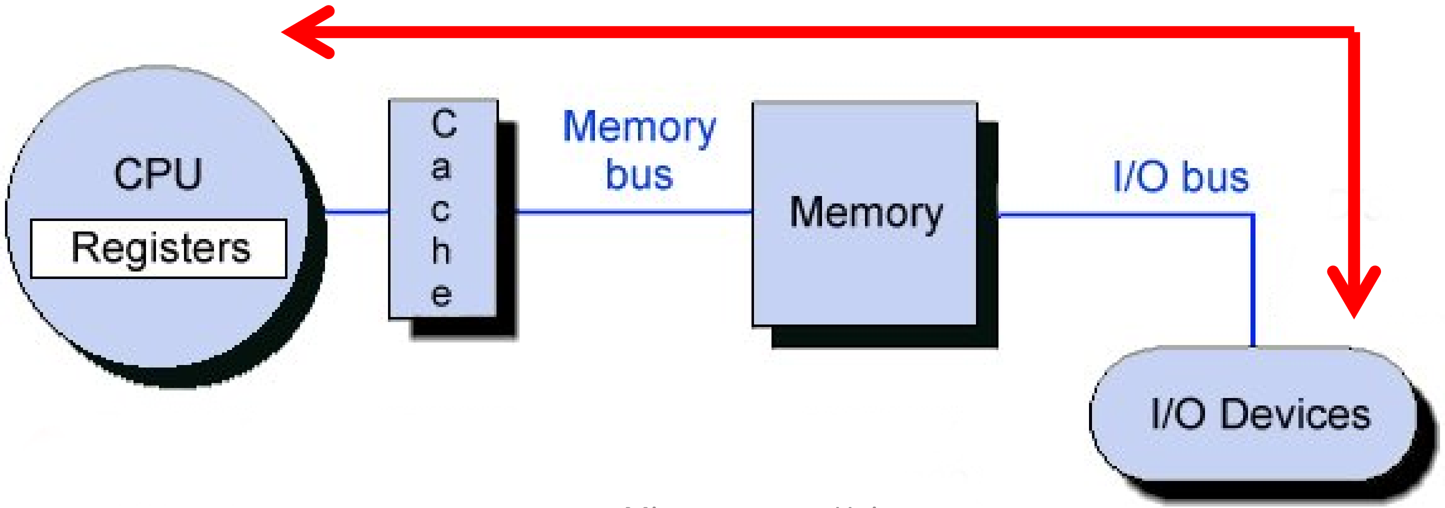
3. **Control bus.** Timing and memory control signals, such as address signaling, memory read/write, and other computer-controlled information, are carried by the control bus.

MPU Architecture – Memory Hierarchy

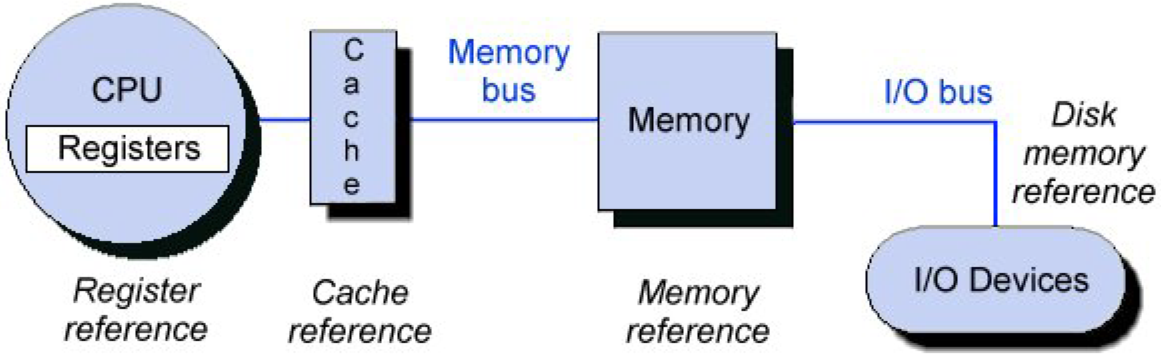
Storage Device (with programs and data) →

Memory → Processing by CPU →

Memory → Storage Device



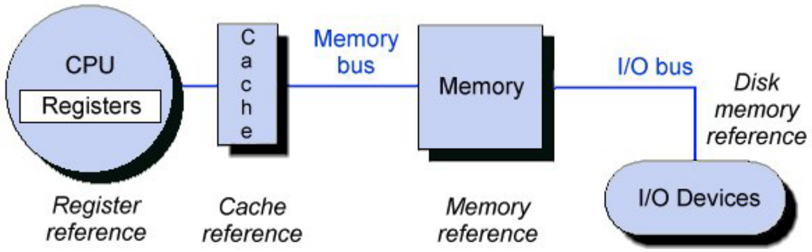
MPU Architecture – Memory Hierarchy



A *cache* is a small, fast memory located close to the CPU that holds the most recently accessed code or data. When the CPU does not find a data item it needs in the cache, a *cache miss* occurs, and the data is retrieved from main memory and put into the cache.

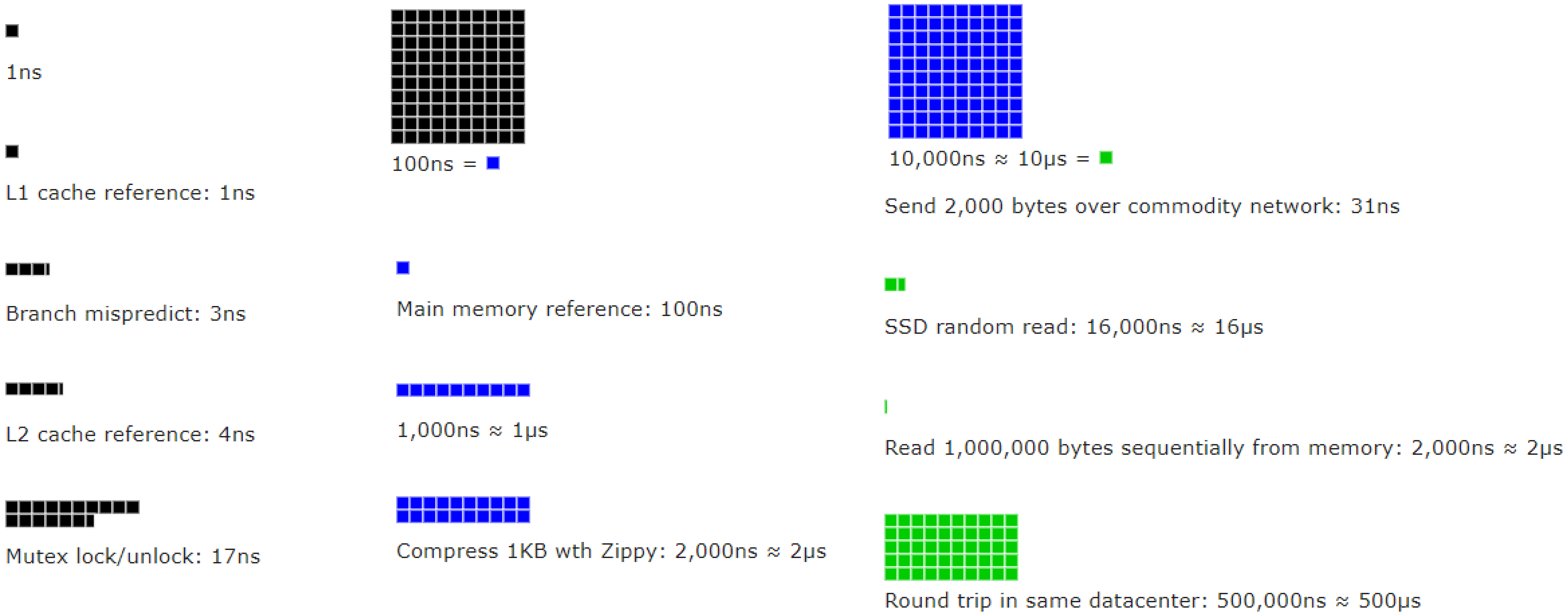
The typical levels in the hierarchy slow down and get larger as we move away from the CPU.

MPU Architecture – Memory Hierarchy



Level	**1**	**2**	**3**	**4**
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 kB	< 1 GB	4 GB - 64 GB	500 GB – 2 TB
Access time (ns)	1	< 10	> 10	> 40,000
Bandwidth (GB/s)	-	-	10-25	6
Managed by	Compiler	Hardware	Operating system	Operating system/user

Latency Numbers Every Programmer Should Know



1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1μs

Compress 1KB wth Zippy: 2,000ns ≈ 2μs

10,000ns ≈ 10μs = 🟩

Send 2,000 bytes over commodity network: 31ns

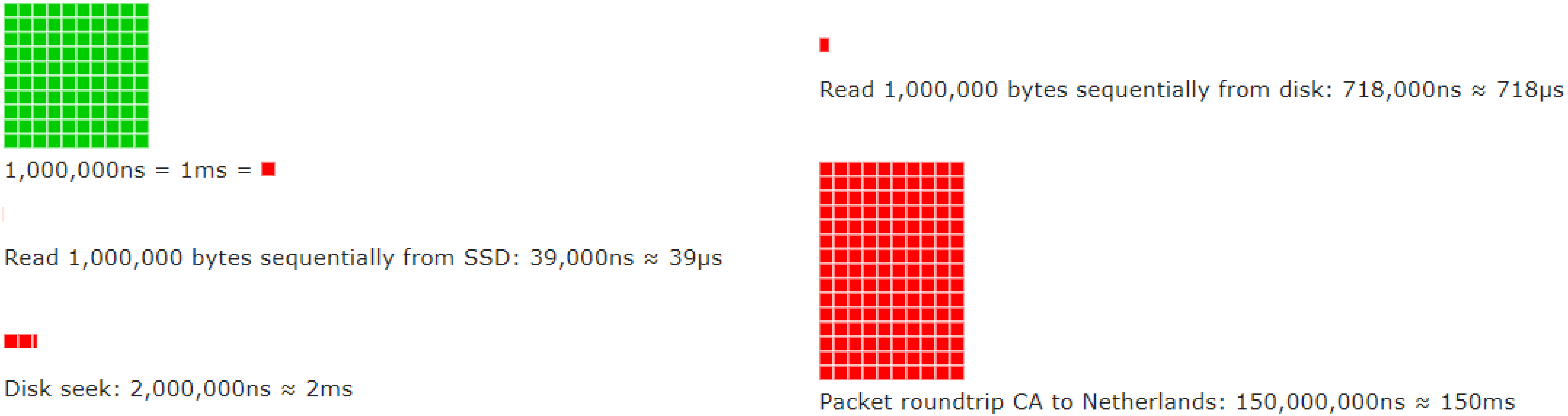
SSD random read: 16,000ns ≈ 16μs

Read 1,000,000 bytes sequentially from memory: 2,000ns ≈ 2μs

Round trip in same datacenter: 500,000ns ≈ 500μs

Source: https://colin-scott.github.io/personal_website/research/interactive_latency.html

Latency Numbers Every Programmer Should Know



1,000,000ns = 1ms = 🟥

Read 1,000,000 bytes sequentially from SSD: 39,000ns ≈ 39µs

Disk seek: 2,000,000ns ≈ 2ms

Read 1,000,000 bytes sequentially from disk: 718,000ns ≈ 718µs

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

Source: https://colin-scott.github.io/personal_website/research/interactive_latency.html

Latency Numbers Every Programmer Should Know

If L1 access is a second, then:

L1 cache reference : 0:00:01
Branch mispredict : 0:00:10
L2 cache reference : 0:00:14
Mutex lock/unlock : 0:00:50

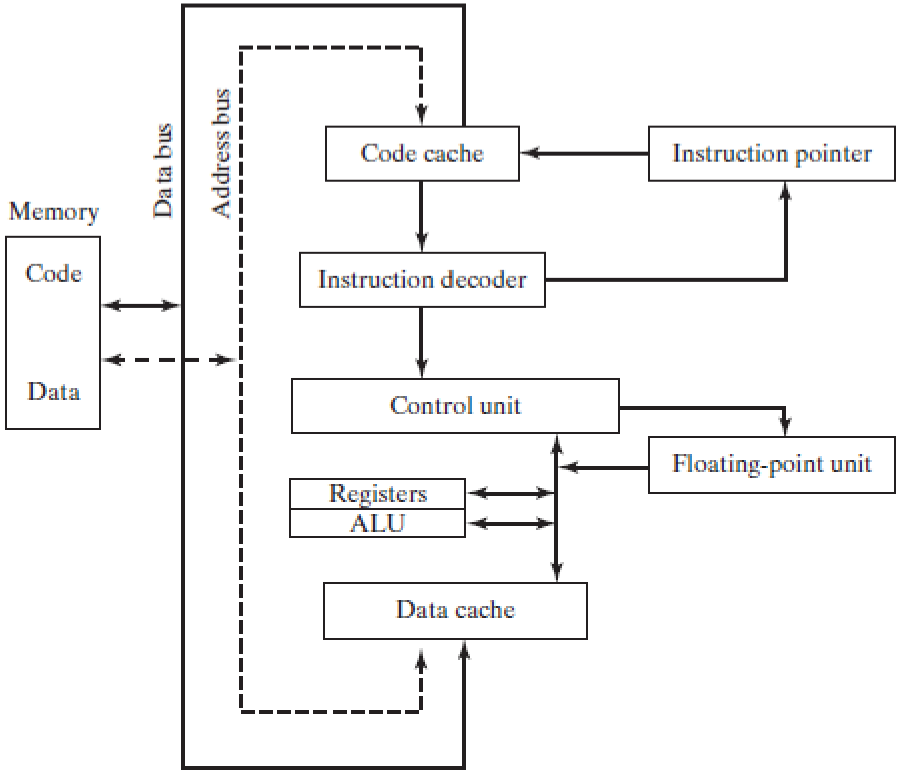
Main memory reference : 0:03:20
Compress 1K bytes with Zippy : 1:40:00
Send 1K bytes over 1 Gbps network : 5:33:20

Read 4K randomly from SSD : 3 days, 11:20:00
Read 1 MB sequentially from memory : 5 days, 18:53:20
Round trip within same datacenter : 11 days, 13:46:40
Read 1 MB sequentially from SSD : 23 days, 3:33:20

Disk seek : 231 days, 11:33:20
Read 1 MB sequentially from disk : 462 days, 23:06:40
Send packet CA->Netherlands->CA : 3472 days, 5:20:00

Source: https://gist.github.com/jboner/2841832

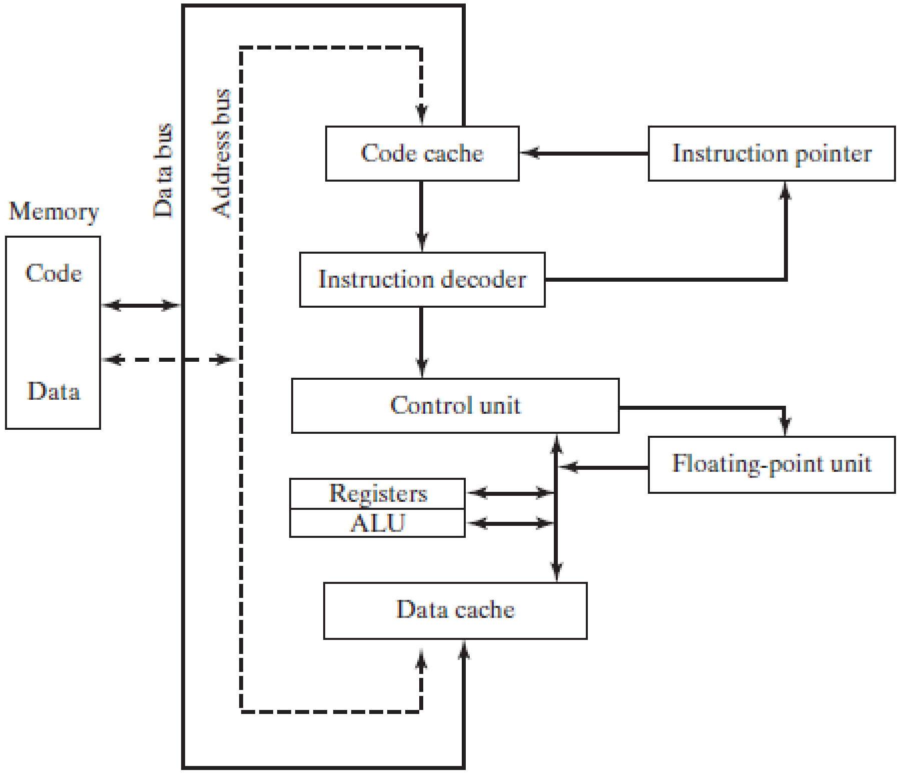
MPU Architecture – Instruction Execution



- If program instructions are to be read from memory, an address is placed on the address bus.
- Next, the memory controller places the requested code on the data bus.
- This allows the code to be available in the inside the code cache.

Source: https://www.allaboutcircuits.com/technical-articles/an-introduction-to-x86-processor-architecture/
Retrieved on July 28, 2020

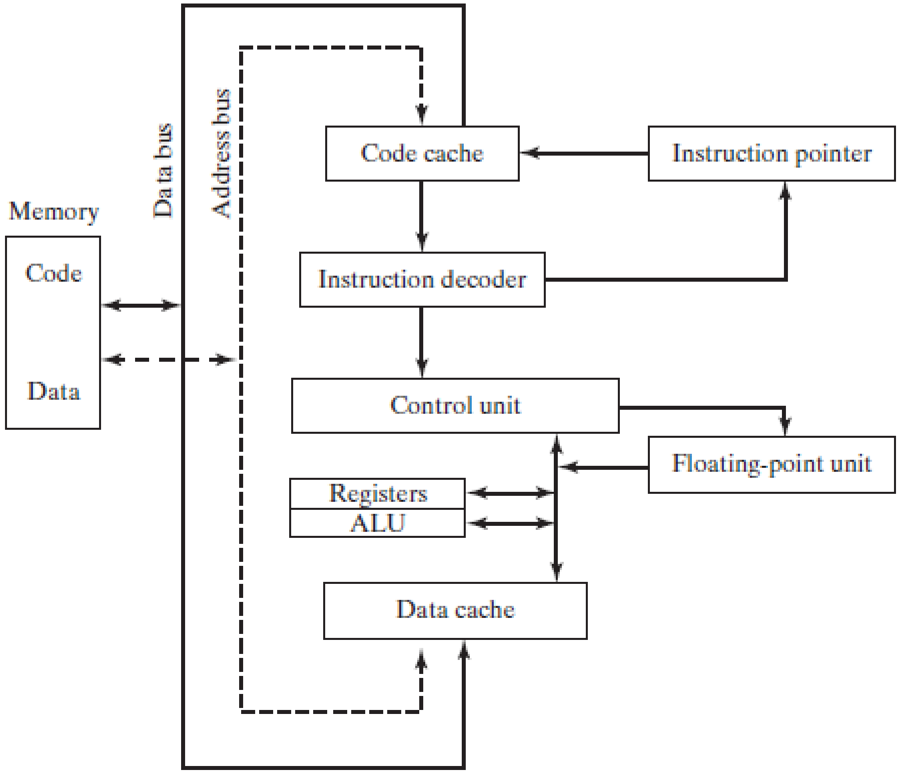
MPU Architecture – Instruction Execution



- The **instruction pointer**’s value (IP, also known as the “program counter” [PC]) determines which program instruction will execute next.
- The instruction is analyzed by the **instruction decoder**, causing the appropriate digital signals to be sent to the control unit.

Source: https://www.allaboutcircuits.com/technical-articles/an-introduction-to-x86-processor-architecture/
Retrieved on July 28, 2020

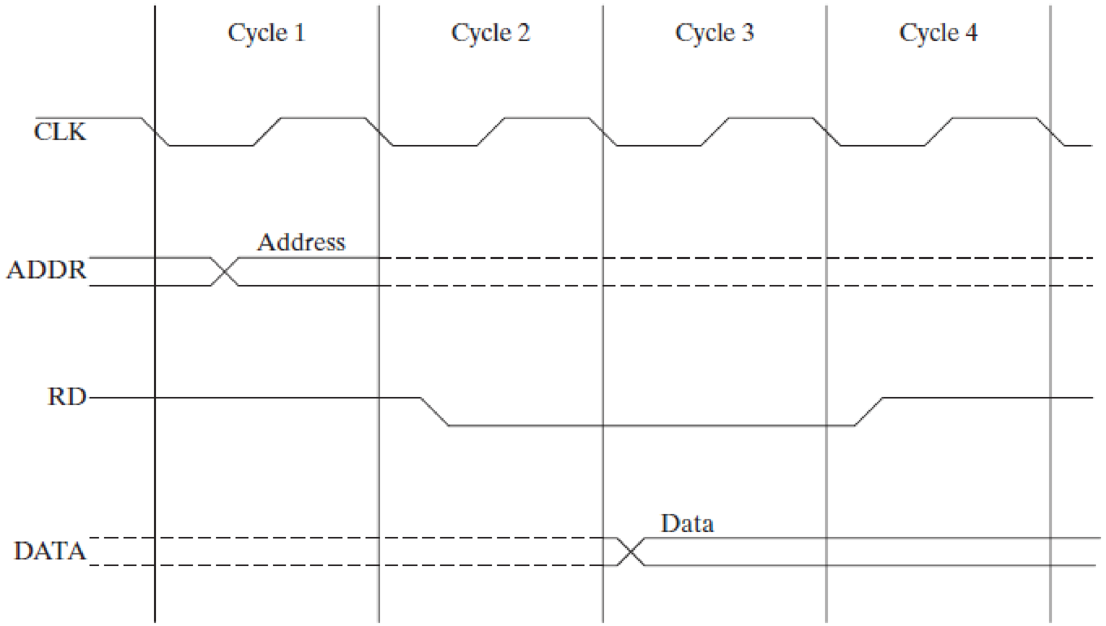
MPU Architecture – Instruction Execution



- The **control unit** (also known as the “controller sequencer”) coordinates the **ALU** (“arithmetic-logic unit”) and **floating-point unit**.
- The **control bus** is not shown, but it carries various signals the use the **system clock** to coordinate the transfer of data between the different CPU components.

Source: https://www.allaboutcircuits.com/technical-articles/an-introduction-to-x86-processor-architecture/
Retrieved on July 28, 2020

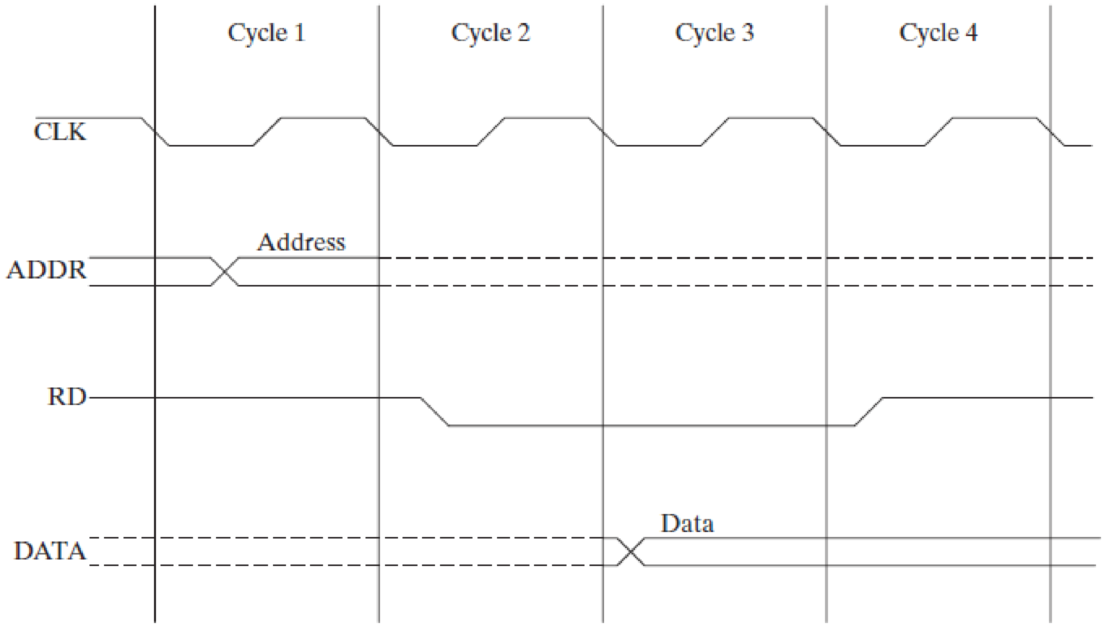
MPU Architecture – Reading from Memory



- Reading instructions (code) or data from memory uses the processor clock (**CLK**).
- The clock triggers on a falling edge (change from high to low [1 to 0]).

Source: https://www.allaboutcircuits.com/technical-articles/an-introduction-to-x86-processor-architecture/
Retrieved on July 28, 2020

MPU Architecture – Reading from Memory



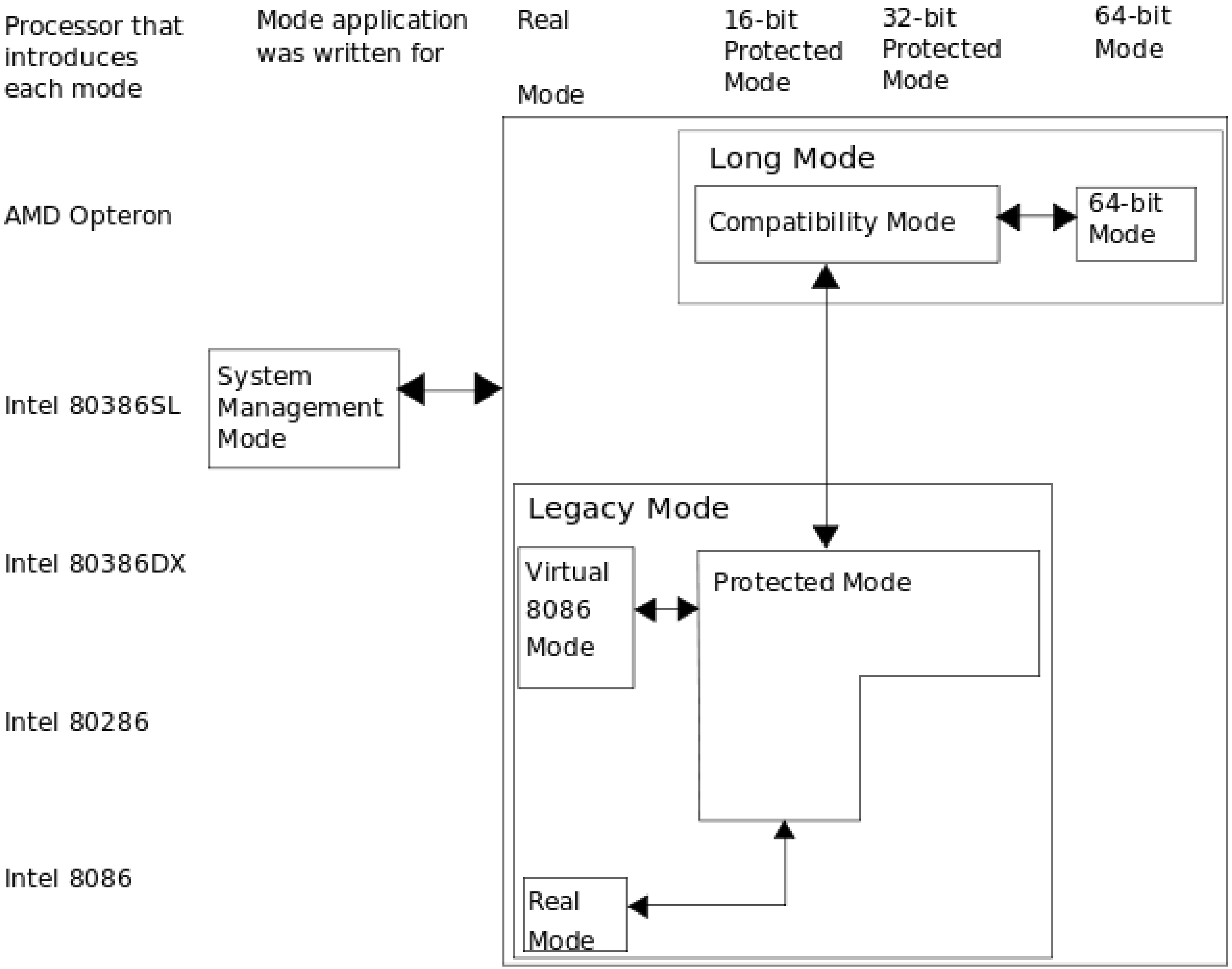
1. The address bits are placed on the address bus (**ADDR**).
2. The read line (**RD**) is set low.
3. The CPU waits one cycle to allow the memory controller to place the data on the data bus (**DATA**).
4. The read line goes high, which signals the CPU to read the data.

Source: https://www.allaboutcircuits.com/technical-articles/an-introduction-to-x86-processor-architecture/
Retrieved on July 28, 2020

Intel/AMD Processor Registers

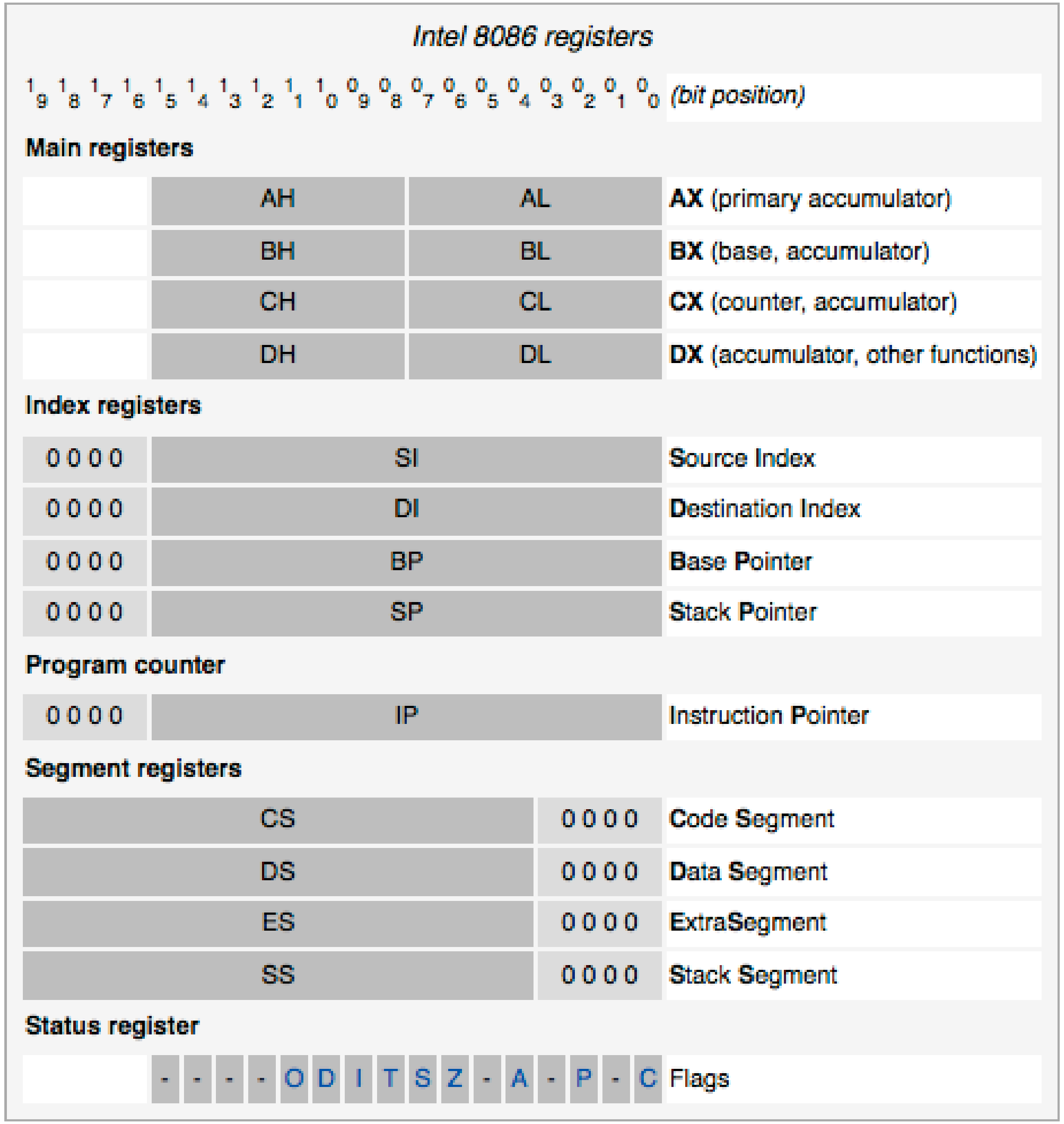
- The first processor in the **x86** family was the **8086**
- In modern processors, this mode of operation is known as “**Real mode**” (or “Real-address mode”)
- Registers in 8086
  - 4 segment registers (16-bit)
    - CS, DS, SS, ES
  - 8 general-purpose registers (16-bit)
    - AX, BX, CX, DX, SP, BP, SI, DI

CPU Operating Modes: State Diagram



https://en.wikipedia.org/wiki/X86-64

x86 Registers – 8-bit & 16-bit

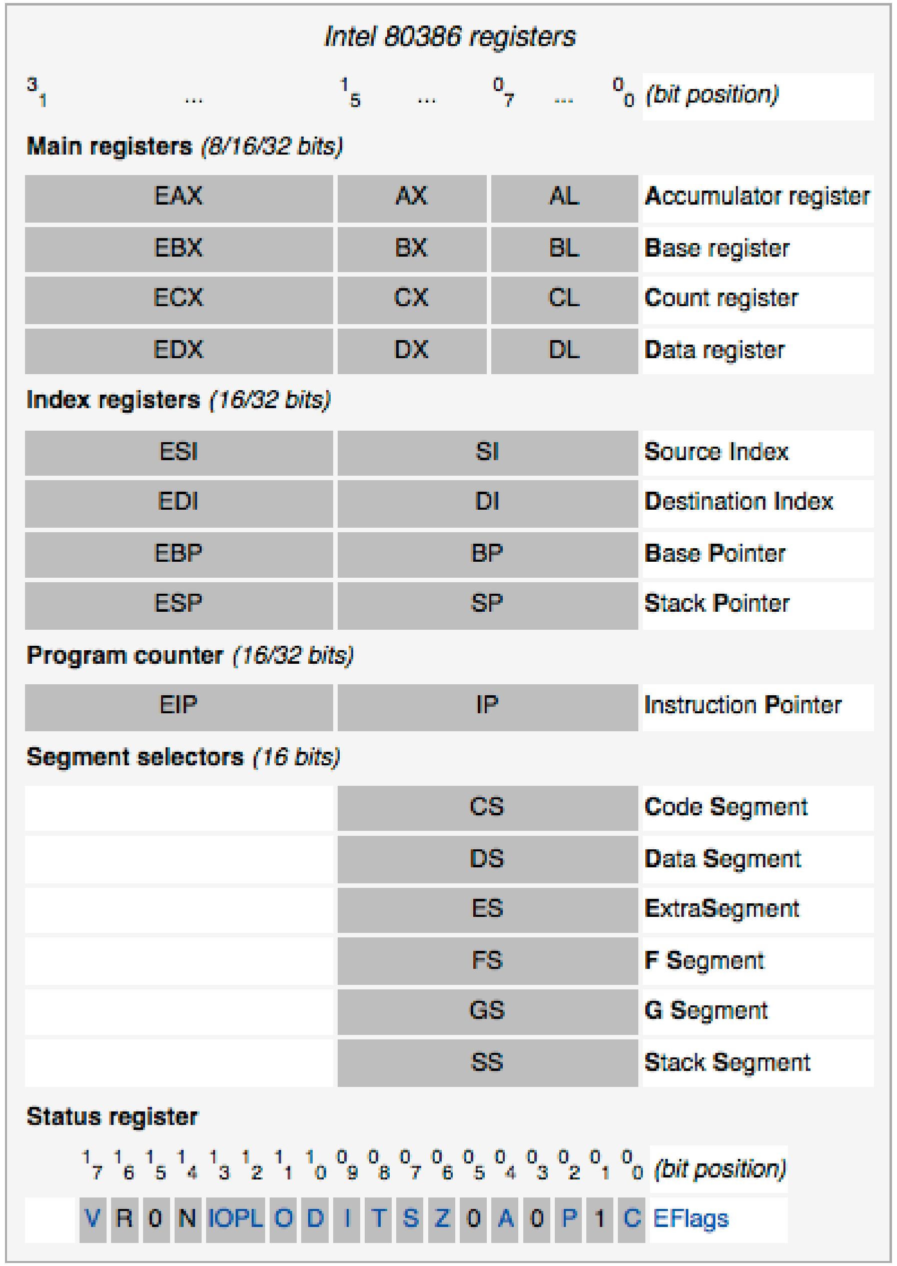


Intel 8086 registers

19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	*(bit position)*
Main registers																				
				AH								AL								**AX** (primary accumulator)
				BH								BL								**BX** (base, accumulator)
				CH								CL								**CX** (counter, accumulator)
				DH								DL								**DX** (accumulator, other functions)
Index registers																				
0 0 0 0				SI																Source Index
0 0 0 0				DI																Destination Index
0 0 0 0				BP																Base Pointer
0 0 0 0				SP																Stack Pointer
Program counter																				
0 0 0 0				IP																Instruction Pointer
Segment registers																				
CS																0 0 0 0				Code Segment
DS																0 0 0 0				Data Segment
ES																0 0 0 0				ExtraSegment
SS																0 0 0 0				Stack Segment
Status register																				
				-	-	-	-	O	D	I	T	S	Z	-	A	-	P	-	C	Flags

https://en.wikipedia.org/wiki/Intel_8086#Registers_and_instructions

x86 Registers – 32-bit



Intel 80386 registers

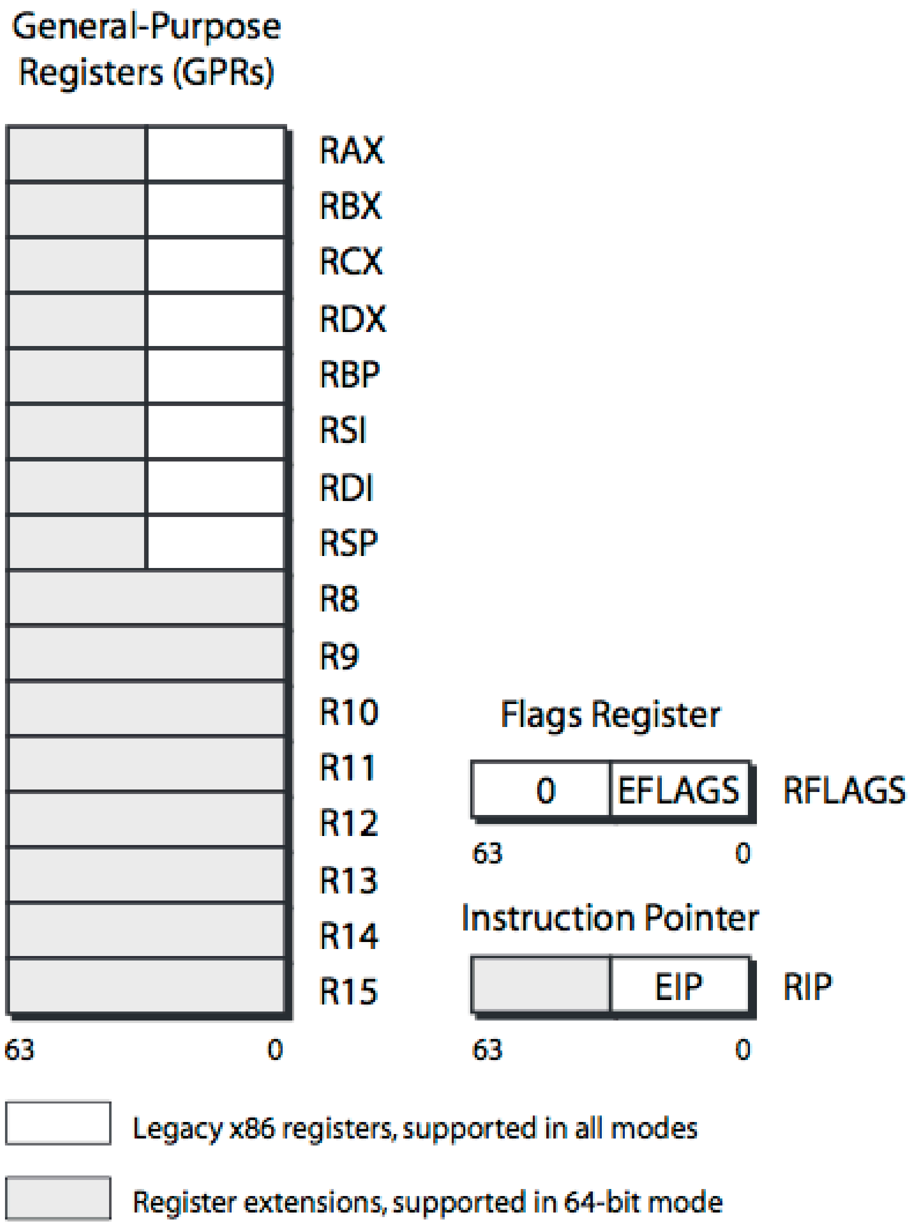
31 ...	15 ...	07 ... 00	*(bit position)*
Main registers *(8/16/32 bits)*			
EAX	AX	AL	**A**ccumulator register
EBX	BX	BL	**B**ase register
ECX	CX	CL	**C**ount register
EDX	DX	DL	**D**ata register
Index registers *(16/32 bits)*			
ESI	SI		**S**ource **I**ndex
EDI	DI		**D**estination **I**ndex
EBP	BP		**B**ase **P**ointer
ESP	SP		**S**tack **P**ointer
Program counter *(16/32 bits)*			
EIP	IP		**I**nstruction **P**ointer
Segment selectors *(16 bits)*			
	CS		**C**ode **S**egment
	DS		**D**ata **S**egment
	ES		**E**xtra**S**egment
	FS		**F** **S**egment
	GS		**G** **S**egment
	SS		**S**tack **S**egment

Status register

17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	*(bit position)*
V	R	0	N	IOPL		O	D	I	T	S	Z	0	A	0	P	1	C	EFlags

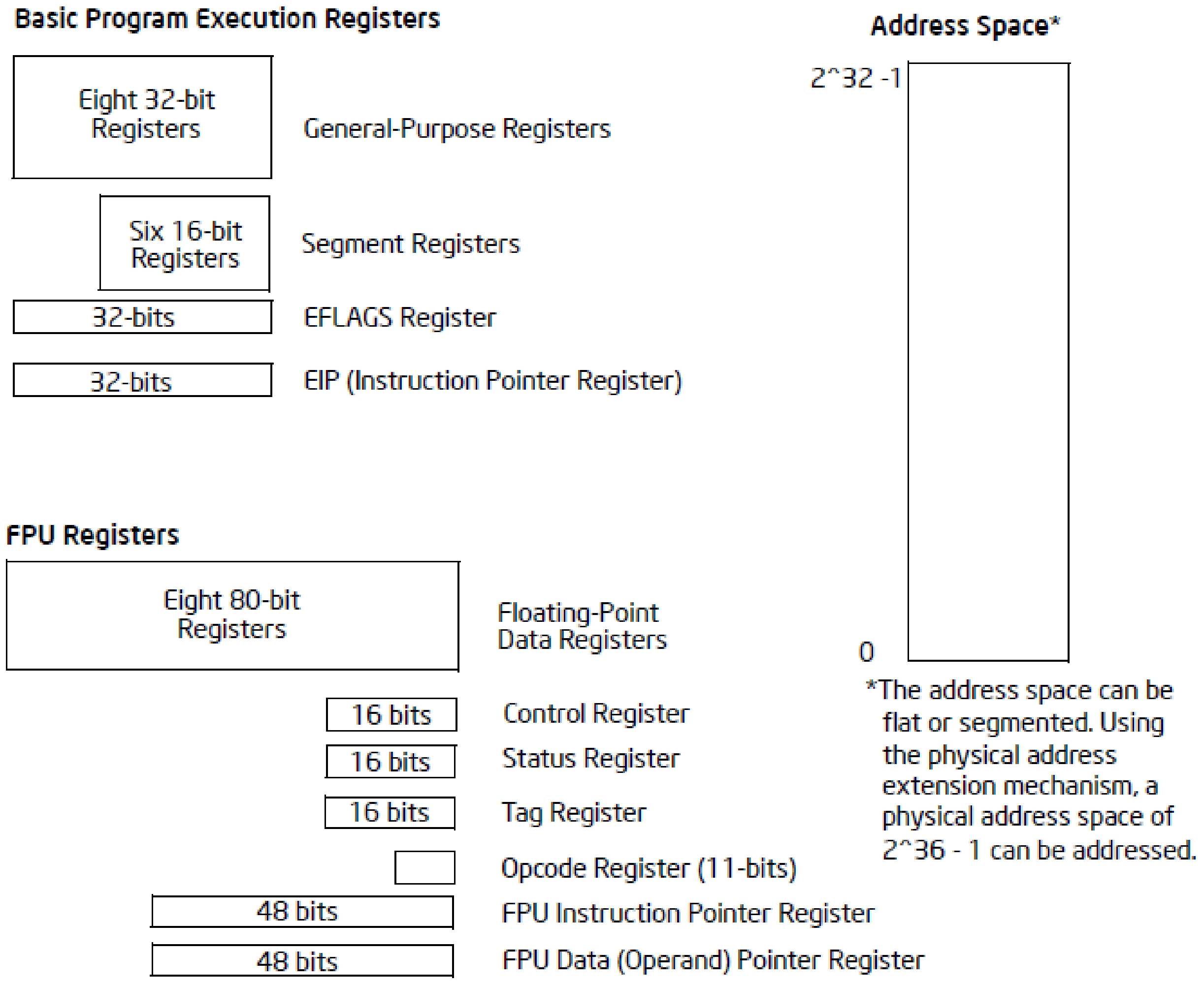
https://en.wikipedia.org/wiki/Intel_80386

X86-64/ Intel 64/ AMD64 Registers – 64-bit


General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
Flags Register
0
EFLAGS
RFLAGS
63
0
Instruction Pointer
EIP
RIP
63
0
Legacy x86 registers, supported in all modes
Register extensions, supported in 64-bit mode

http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/24592_APM_v11.pdf

Basic Execution Environment: Non-64-bit Modes



Basic Program Execution Registers

Eight 32-bit Registers — General-Purpose Registers

Six 16-bit Registers — Segment Registers

32-bits — EFLAGS Register

32-bits — EIP (Instruction Pointer Register)

FPU Registers

Eight 80-bit Registers — Floating-Point Data Registers

16 bits — Control Register

16 bits — Status Register

16 bits — Tag Register

Opcode Register (11-bits)

48 bits — FPU Instruction Pointer Register

48 bits — FPU Data (Operand) Pointer Register

Address Space*

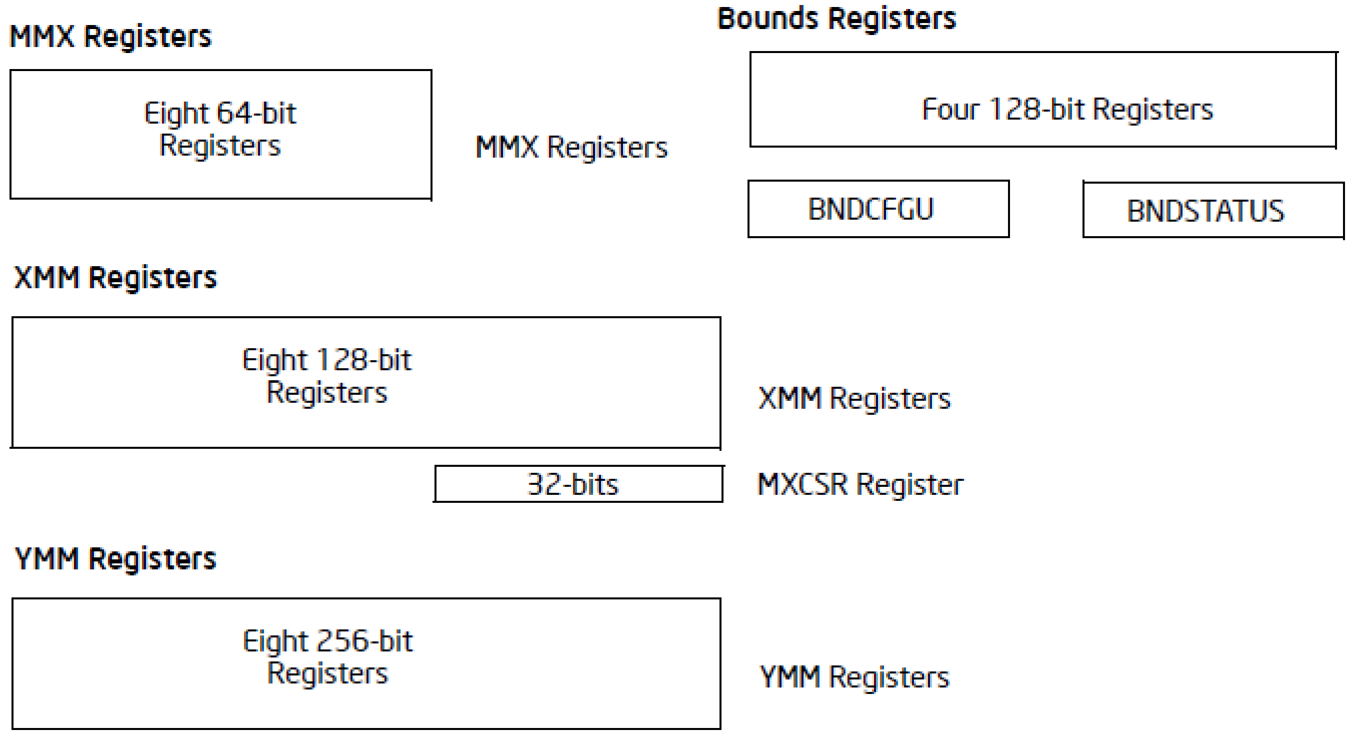
2^32 -1

0

*The address space can be flat or segmented. Using the physical address extension mechanism, a physical address space of 2^36 - 1 can be addressed.

Source: Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 1. Intel Corp., May 2020.

Basic Execution Environment: Non-64-bit Modes



MMX Registers

Eight 64-bit Registers

MMX Registers

Bounds Registers

Four 128-bit Registers

BNDCFGU

BNDSTATUS

XMM Registers

Eight 128-bit Registers

XMM Registers

32-bits

MXCSR Register

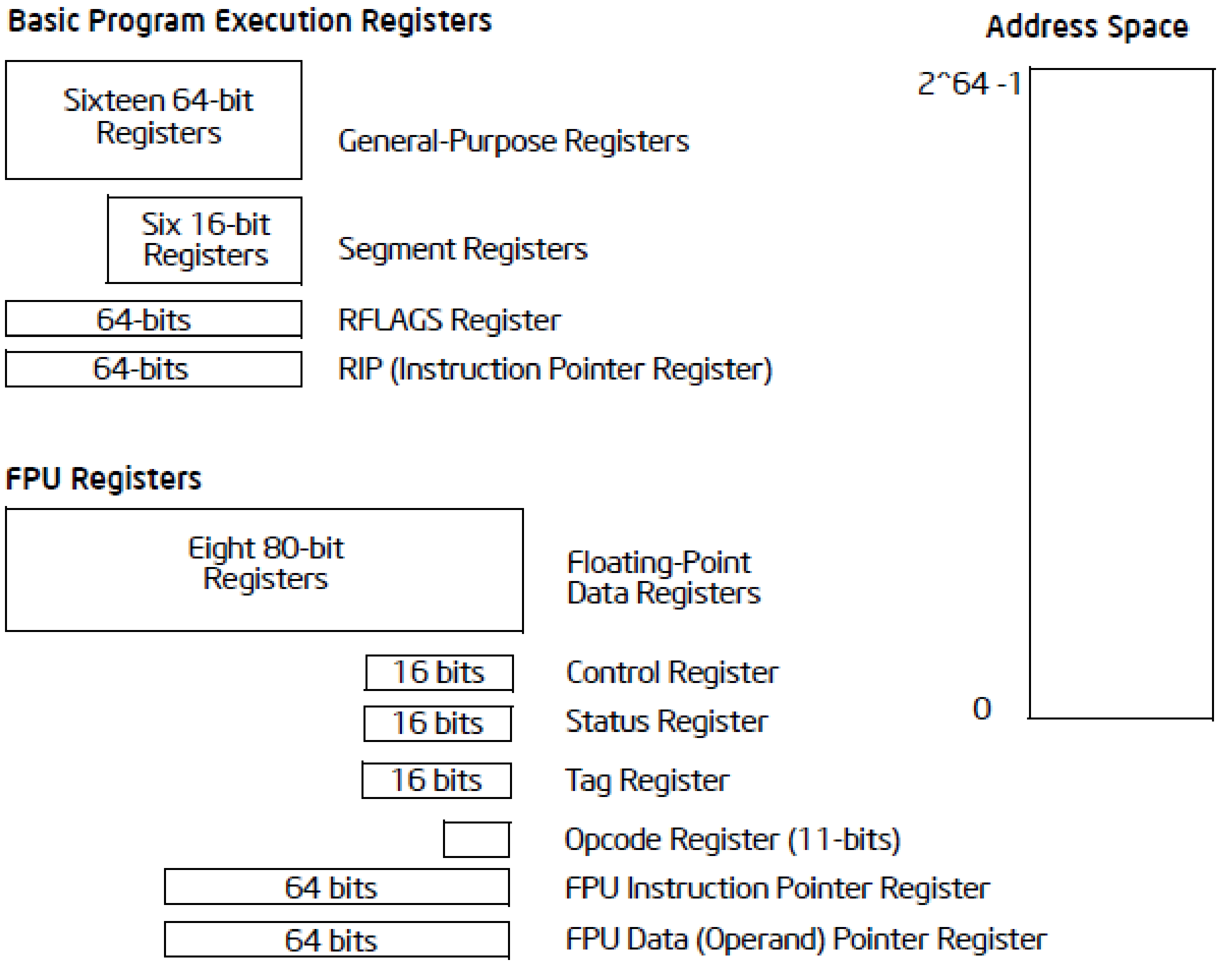
YMM Registers

Eight 256-bit Registers

YMM Registers

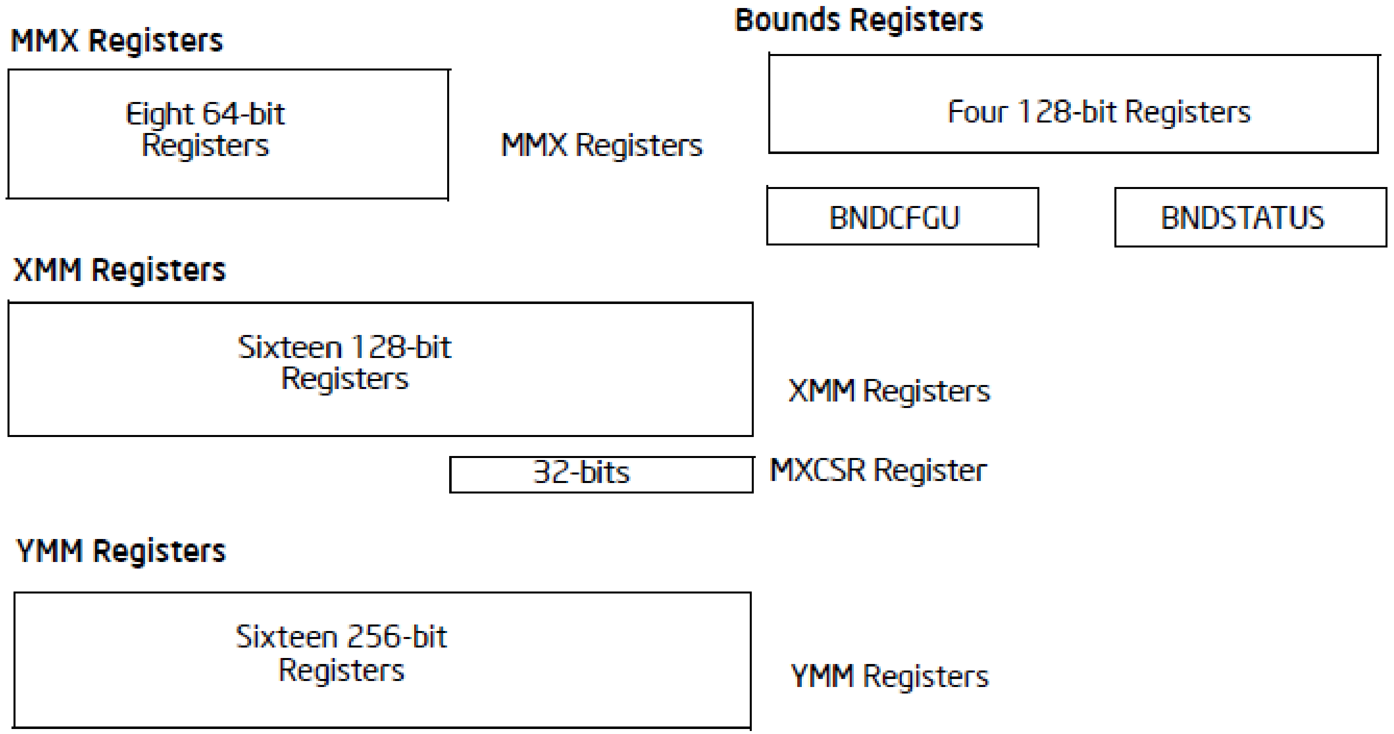
Source: Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 1. Intel Corp., May 2020.

Basic Execution Environment: 64-bit Mode



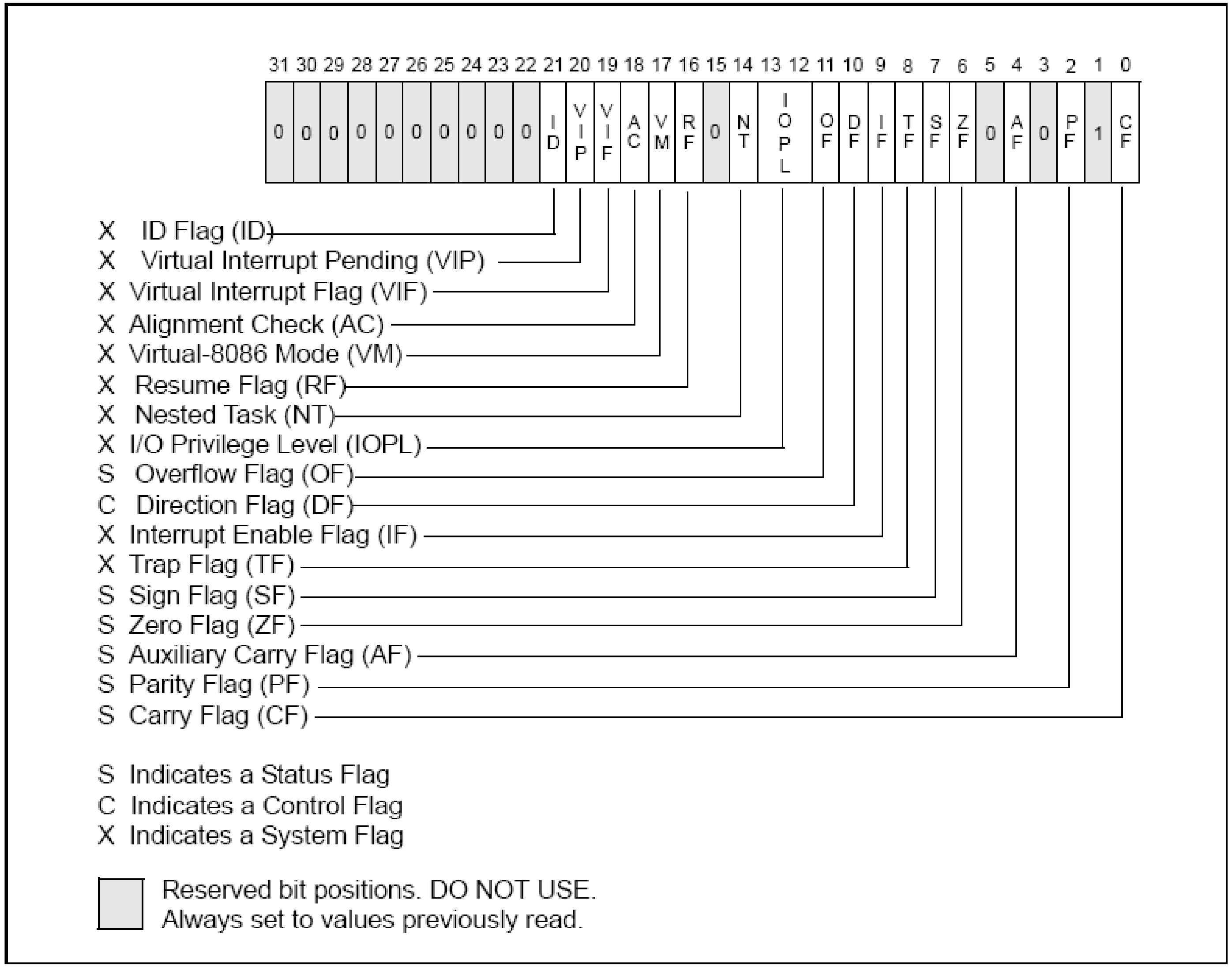
Source: Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 1. Intel Corp., May 2020.

Basic Execution Environment: 64-bit Mode



Source: Intel 64 and IA-32 Architectures Software Developer’s Manual, Vol. 1. Intel Corp., May 2020.

EFLAGS* in x86



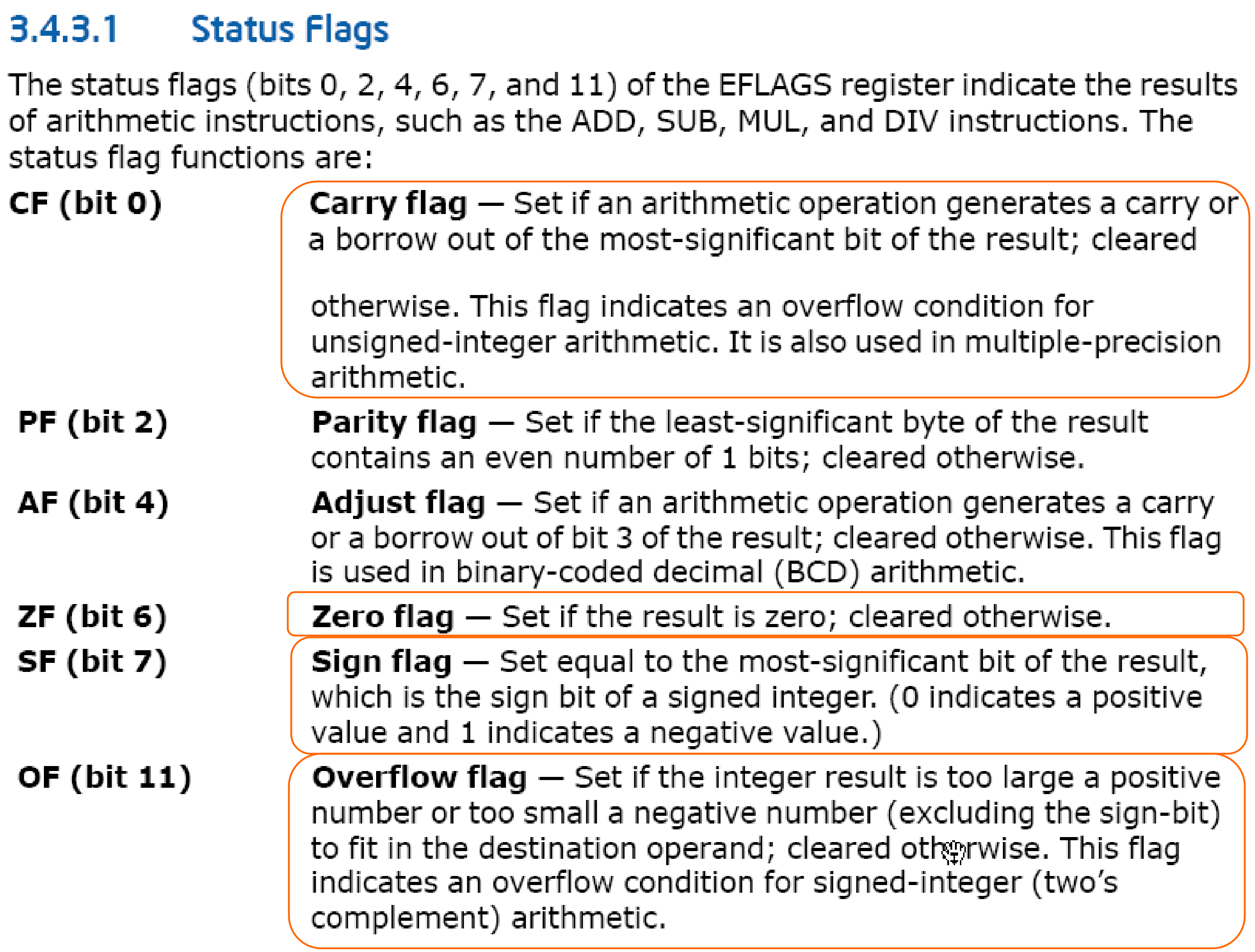
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
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.

* RFLAGS is the 64-bit equivalent:
Bits 22-63 are reserved

EFLAGS in x86



3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

CF (bit 0) **Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

PF (bit 2) **Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

AF (bit 4) **Adjust flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

ZF (bit 6) **Zero flag** — Set if the result is zero; cleared otherwise.

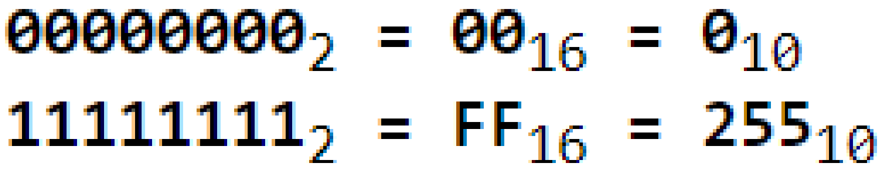
SF (bit 7) **Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

OF (bit 11) **Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two’s complement) arithmetic.

Machine Code and Assembly Language

- The native tongue of the computer is **binary**.

- In a computer, all words can be interpreted as 8-bit words –as **bytes**.



$$00000000_2 = 00_{16} = 0_{10}$$
$$11111111_2 = FF_{16} = 255_{10}$$

- If you were to communicate with the computer directly you would have to use 8-bit binary words as the basic means of telling the machine what to do.

Machine Code and Assembly Language

- As humans, it would be difficult for us to communicate where all the alphanumeric characters are just 0 and 1.
- This difficult and lowest-level language is called **machine language**.
- It is a little easier to use to express a byte (8-bit word) in hexadecimal; 67 in hex is more easily said and thought of than 01100111 in true binary.
- Because this is one step closer to human language, speaking in hex would be our next higher-level language.

Machine Code and Assembly Language

- When we “speak” to the computer, we are really speaking to the microprocessor.

- Its “brain” then takes over and issues timely commands to the rest of the machine necessary to accomplish a given task.

- The act of “speaking” is **programming**, and results in machine code.

- The “issuing of timely commands” is **program execution**, where the MPU interprets the code.

Machine Code and Assembly Language

- The designers of a microprocessor provide us with a third higher-level language called **assembly language**.

- It allows use to communicate with the microprocessor using **mnemonics**, short abbreviations such as **ADD**, **SUB**, **MUL** and **DIV**, that are more understandable to humans.

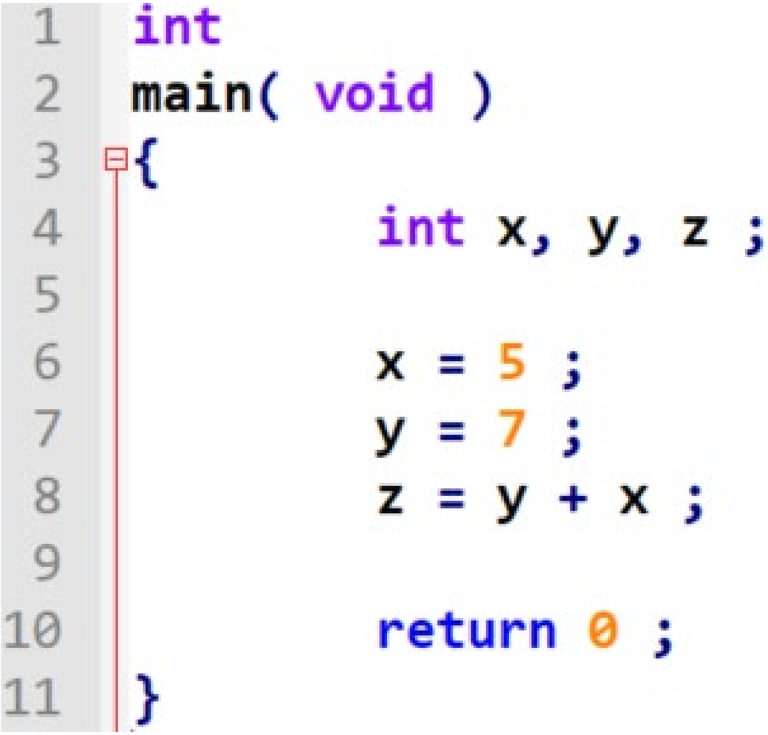
- Each microprocessor (or processor family) has a complete set of these, commonly referred to as the assembly language **instruction set**.

Machine Code and Assembly Language

- There is also assembly-to-machine-language translator for the microprocessor, a program commonly referred to as the **assembler**.
- Suppose we have a program will add two numbers (5 and 7) and store the sum in memory.
- I can write this program in machine code, assembly language, or in a high-level language like C.

Machine Code and Assembly Language

- Suppose we have a program will add two numbers (5 and 7) and store the sum in memory. Here is a **C** program:

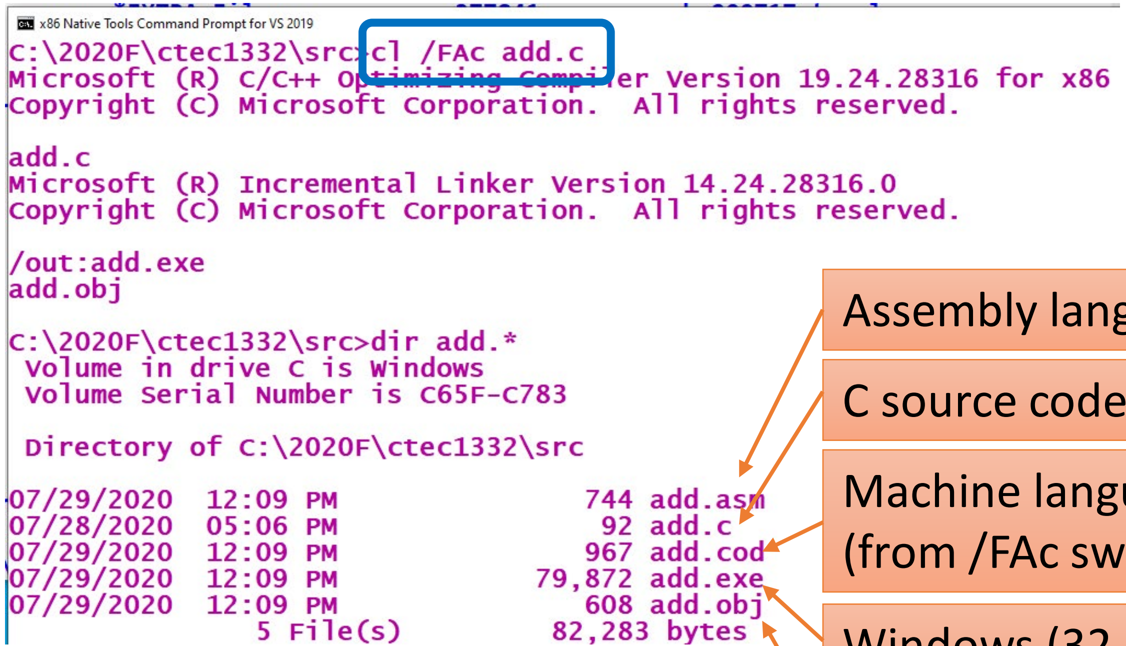


```
int
main( void )
{
        int x, y, z ;

        x = 5 ;
        y = 7 ;
        z = y + x ;

        return 0 ;
}
```

Machine Code and Assembly Language

- I can run the C compiler to translate this into both assembly language and machine code.

cl runs the Microsoft Compiler and Linker – It normally creates **.obj** and **.exe** files; **/FAc** is the compiler switch to output assembly language and machine code; **/FA** outputs assembly language only



```
C:\2020F\ctec1332\src>cl /FAc add.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.24.28316 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

add.c
Microsoft (R) Incremental Linker Version 14.24.28316.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:add.exe
add.obj

C:\2020F\ctec1332\src>dir add.*
 Volume in drive C is Windows
 Volume Serial Number is C65F-C783

 Directory of C:\2020F\ctec1332\src

07/29/2020  12:09 PM               744 add.asm
07/28/2020  05:06 PM                92 add.c
07/29/2020  12:09 PM               967 add.cod
07/29/2020  12:09 PM            79,872 add.exe
07/29/2020  12:09 PM               608 add.obj
               5 File(s)         82,283 bytes
```

Assembly language listing (from /FA switch)

C source code (saved by Notepad++)

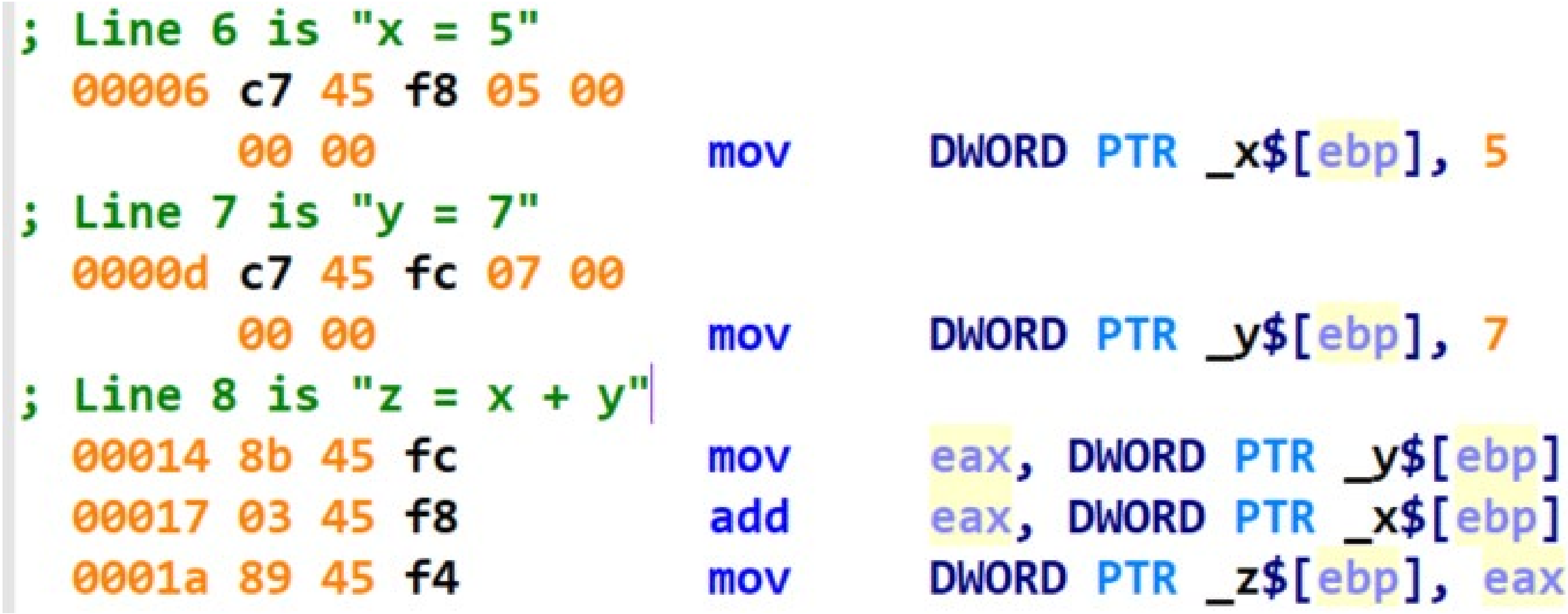
Machine language + assembly language listing (from /FAc switch)

Windows (32-bit) executable ("application")

Object file ("relocatable" machine code)

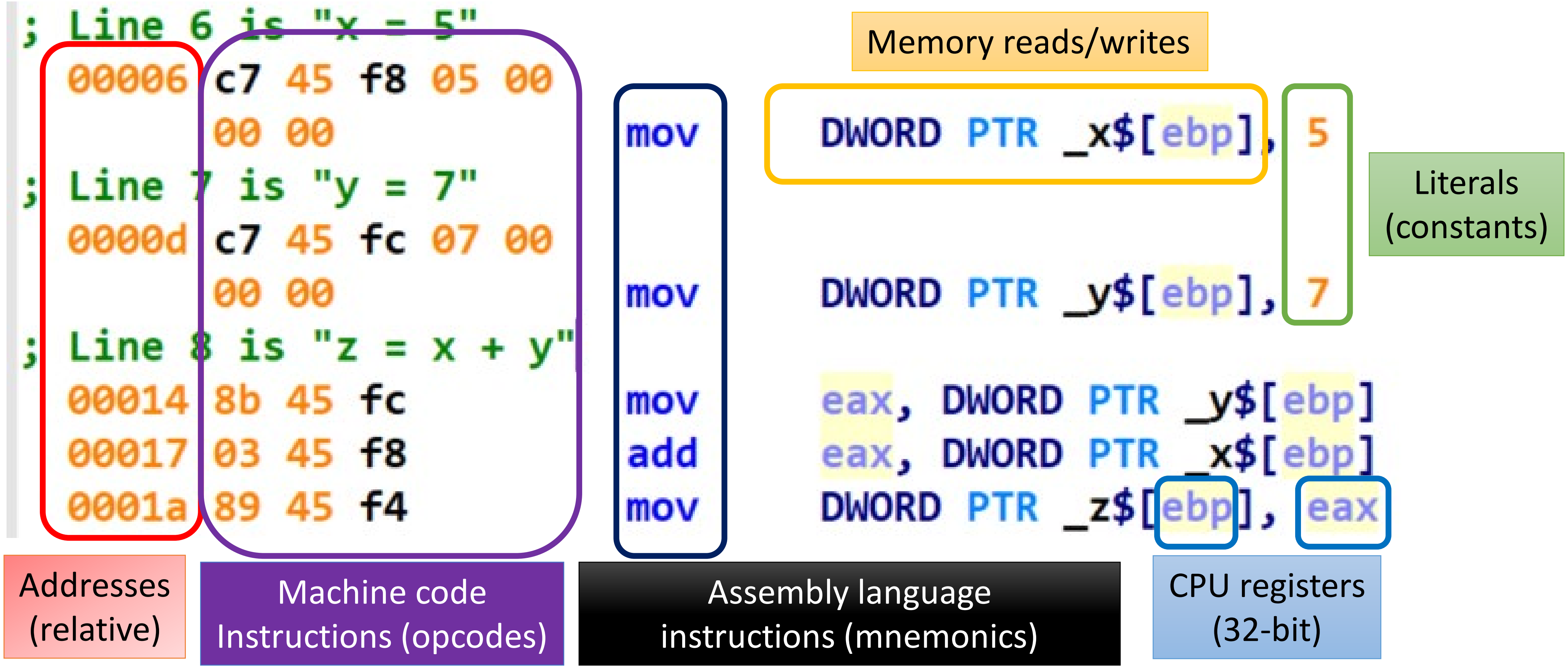
Machine Code and Assembly Language

- Here is (part of) the translation output that shows the correspondence between all three languages:



```
; Line 6 is "x = 5"
  00006 c7 45 f8 05 00
        00 00             mov      DWORD PTR _x$[ebp], 5
; Line 7 is "y = 7"
  0000d c7 45 fc 07 00
        00 00             mov      DWORD PTR _y$[ebp], 7
; Line 8 is "z = x + y"
  00014 8b 45 fc          mov      eax, DWORD PTR _y$[ebp]
  00017 03 45 f8          add      eax, DWORD PTR _x$[ebp]
  0001a 89 45 f4          mov      DWORD PTR _z$[ebp], eax
```

Machine Code and Assembly Language (32-bit)



```
; Line 6 is "x = 5"
  00006 c7 45 f8 05 00
        00 00             mov    DWORD PTR _x$[ebp], 5
; Line 7 is "y = 7"
  0000d c7 45 fc 07 00
        00 00             mov    DWORD PTR _y$[ebp], 7
; Line 8 is "z = x + y"
  00014 8b 45 fc          mov    eax, DWORD PTR _y$[ebp]
  00017 03 45 f8          add    eax, DWORD PTR _x$[ebp]
  0001a 89 45 f4          mov    DWORD PTR _z$[ebp], eax
```

Memory reads/writes

Literals (constants)

Addresses (relative)

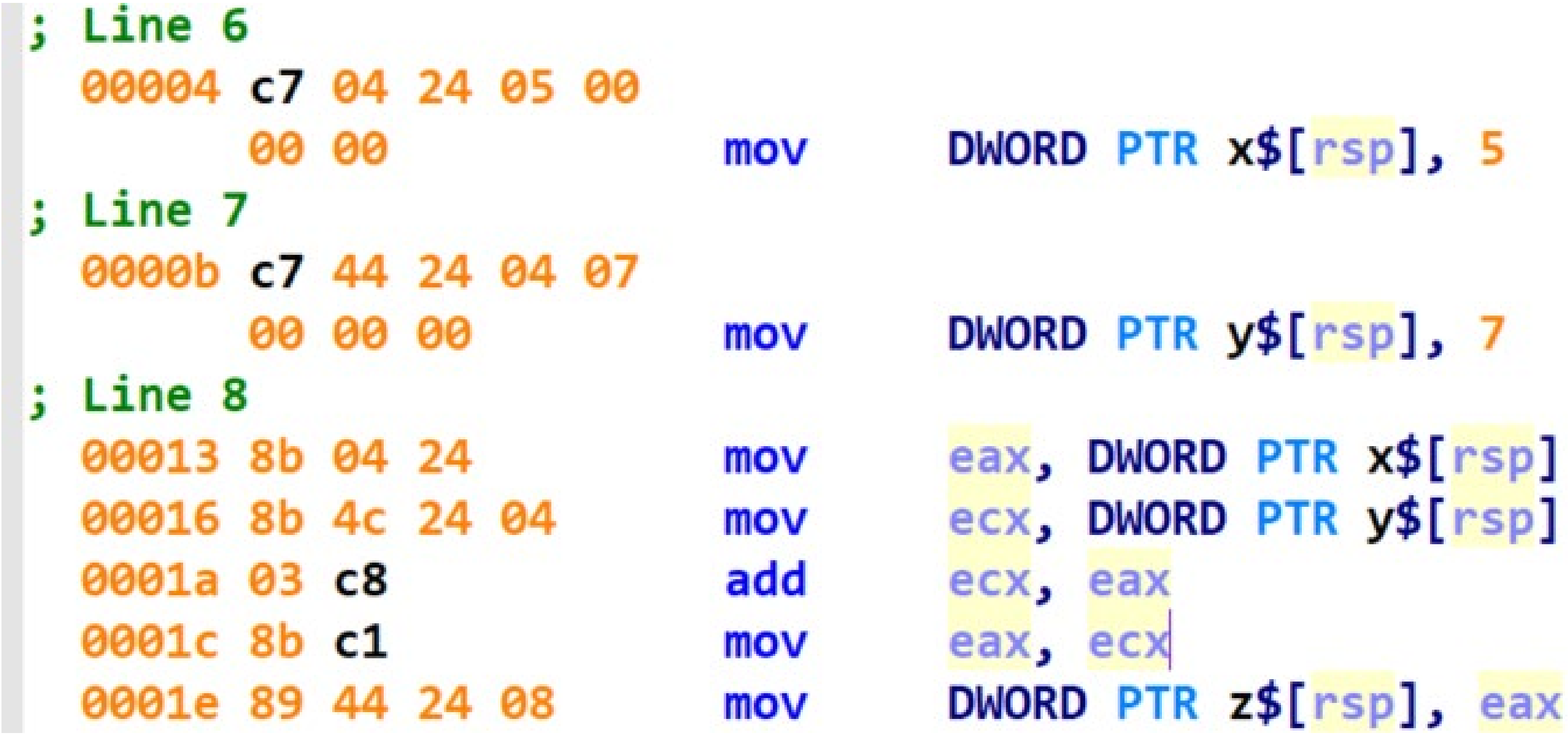
Machine code Instructions (opcodes)

Assembly language instructions (mnemonics)

CPU registers (32-bit)

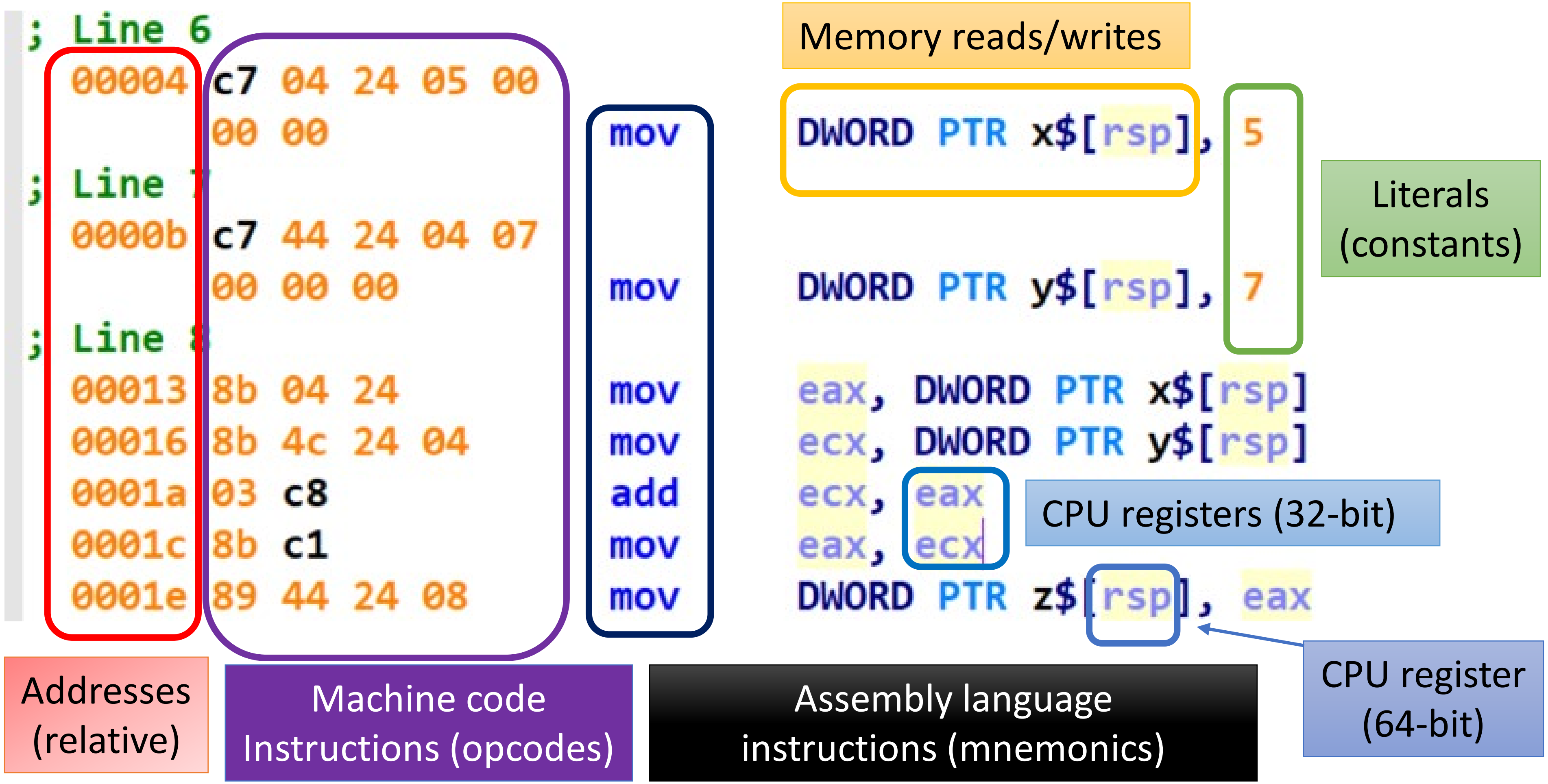
Machine Code and Assembly Language

- If I compile in **64-bit** mode, from a “x64 Native Tools Command Prompt” the code is slightly different:



```
; Line 6
  00004 c7 04 24 05 00
        00 00                mov     DWORD PTR x$[rsp], 5
; Line 7
  0000b c7 44 24 04 07
        00 00 00             mov     DWORD PTR y$[rsp], 7
; Line 8
  00013 8b 04 24             mov     eax, DWORD PTR x$[rsp]
  00016 8b 4c 24 04          mov     ecx, DWORD PTR y$[rsp]
  0001a 03 c8                add     ecx, eax
  0001c 8b c1                mov     eax, ecx
  0001e 89 44 24 08          mov     DWORD PTR z$[rsp], eax
```

Machine Code and Assembly Language (64-bit)



```
; Line 6
00004 c7 04 24 05 00
      00 00           mov    DWORD PTR x$[rsp], 5
; Line 7
0000b c7 44 24 04 07
      00 00 00        mov    DWORD PTR y$[rsp], 7
; Line 8
00013 8b 04 24        mov    eax, DWORD PTR x$[rsp]
00016 8b 4c 24 04     mov    ecx, DWORD PTR y$[rsp]
0001a 03 c8           add    ecx, eax
0001c 8b c1           mov    eax, ecx
0001e 89 44 24 08     mov    DWORD PTR z$[rsp], eax
```

Memory reads/writes

Literals (constants)

CPU registers (32-bit)

CPU register (64-bit)

Addresses (relative)

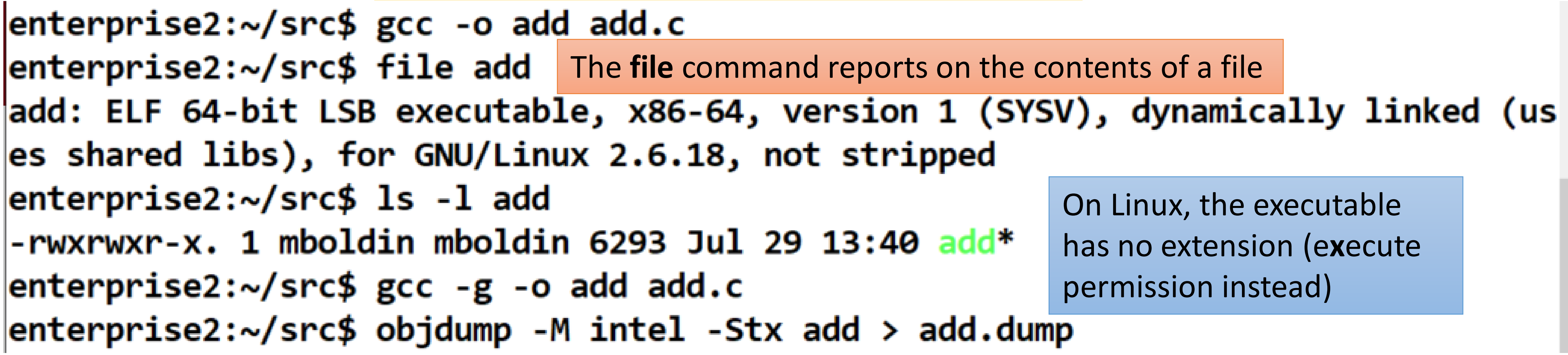
Machine code Instructions (opcodes)

Assembly language instructions (mnemonics)

Machine Code and Assembly Language

- If I compile on Linux (64-bit), the code is slightly different than the Windows version.

Run the GNU C Compiler on add.c; the output (executable) file is **add**.



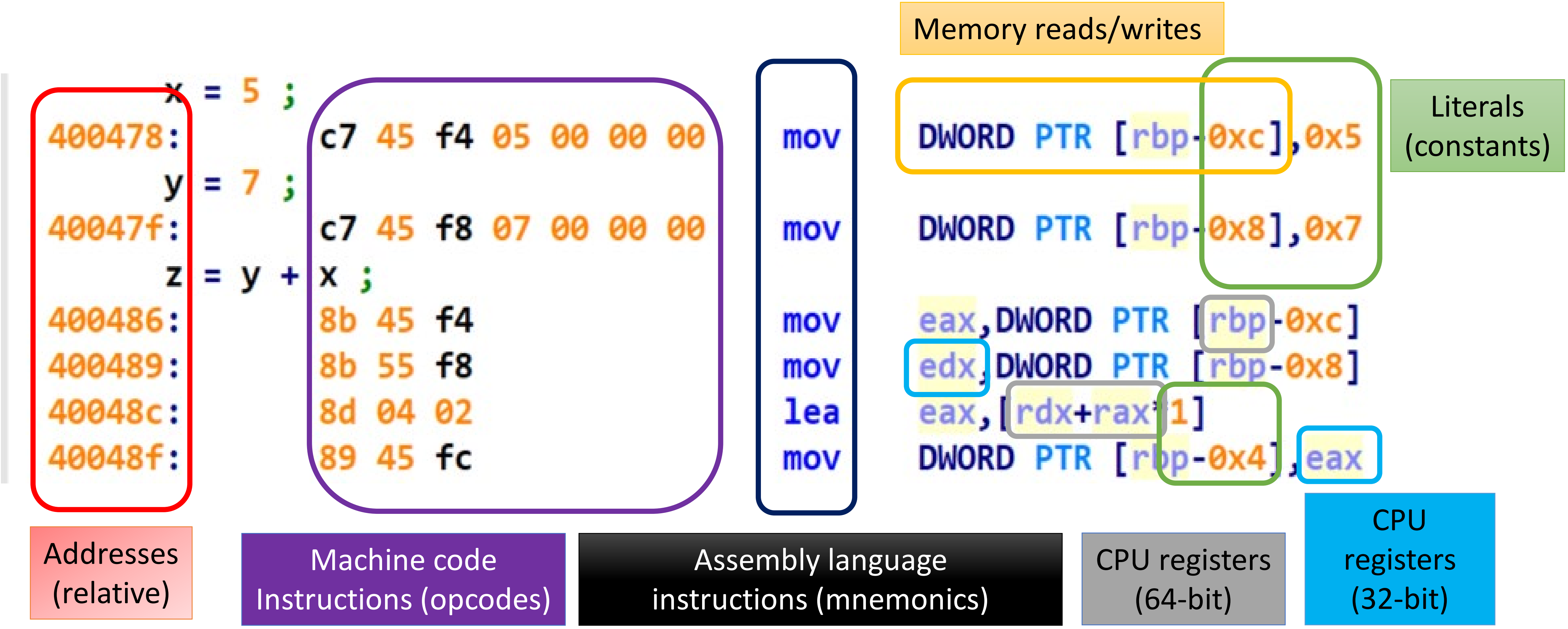
```
enterprise2:~/src$ gcc -o add add.c
enterprise2:~/src$ file add
add: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (us
es shared libs), for GNU/Linux 2.6.18, not stripped
enterprise2:~/src$ ls -l add
-rwxrwxr-x. 1 mboldin mboldin 6293 Jul 29 13:40 add*
enterprise2:~/src$ gcc -g -o add add.c
enterprise2:~/src$ objdump -M intel -Stx add > add.dump
```

The **file** command reports on the contents of a file

On Linux, the executable has no extension (e**x**ecute permission instead)

Recompile, with debug information (**-g** switch), then run **objdump** and capture the output into a text file.

Machine Code and Assembly Language (64-bit Linux)



```
        x = 5 ;
400478:      c7 45 f4 05 00 00 00    mov    DWORD PTR [rbp-0xc],0x5
        y = 7 ;
40047f:      c7 45 f8 07 00 00 00    mov    DWORD PTR [rbp-0x8],0x7
        z = y + x ;
400486:      8b 45 f4                mov    eax,DWORD PTR [rbp-0xc]
400489:      8b 55 f8                mov    edx,DWORD PTR [rbp-0x8]
40048c:      8d 04 02                lea    eax,[rdx+rax*1]
40048f:      89 45 fc                mov    DWORD PTR [rbp-0x4],eax
```

Memory reads/writes

Literals (constants)

Addresses (relative)

Machine code Instructions (opcodes)

Assembly language instructions (mnemonics)

CPU registers (64-bit)

CPU registers (32-bit)

THE END