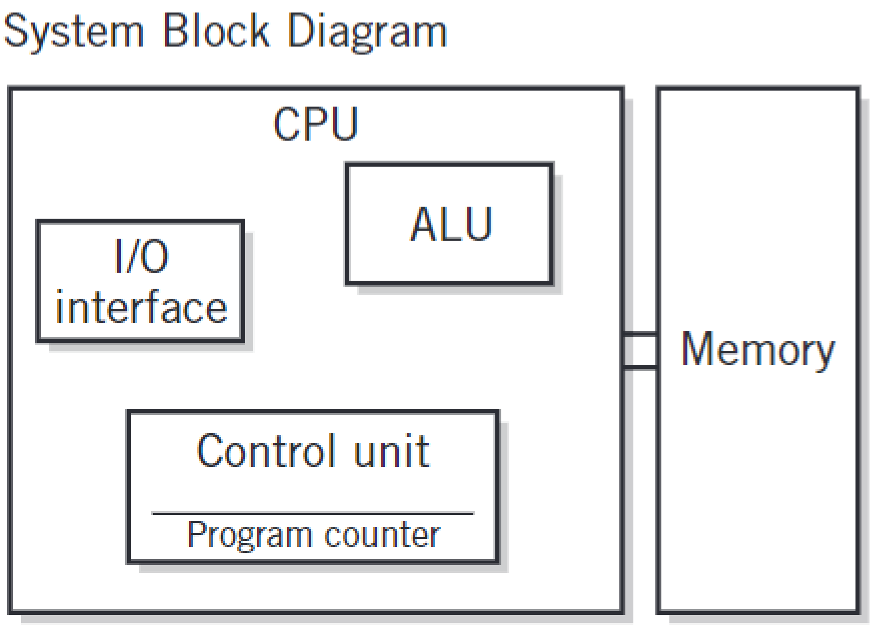
CONESTOGA
Connect Life and Learning

CSCN71010
Winter 2023

Computer Systems Architecture

Topic: the CPU

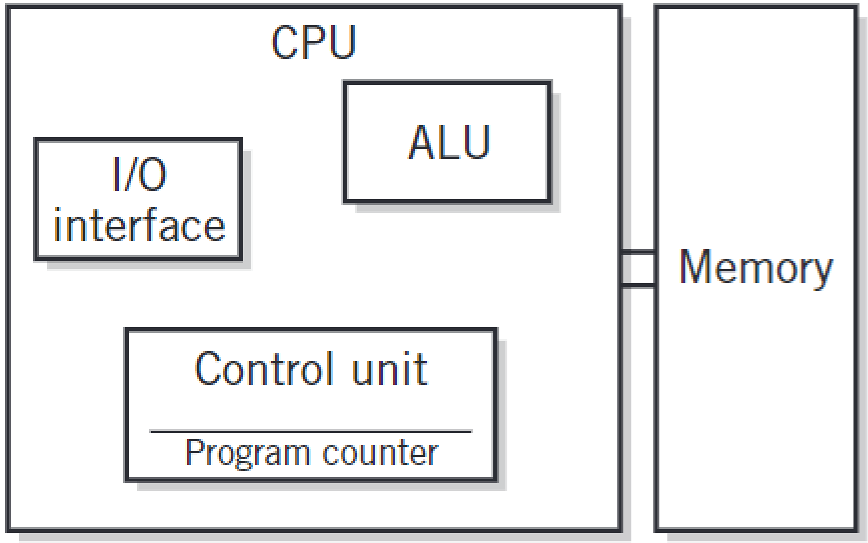
The CPU



- The Computer unit is made up conceptually of three major components:
  - the arithmetic/logic unit (ALU),
  - the control unit (CU),and
  - memory.
- The ALU and CU together are known as the central processing unit (CPU). An input/output(I/O) interface is also included in the diagram.

The CPU (cont’d)

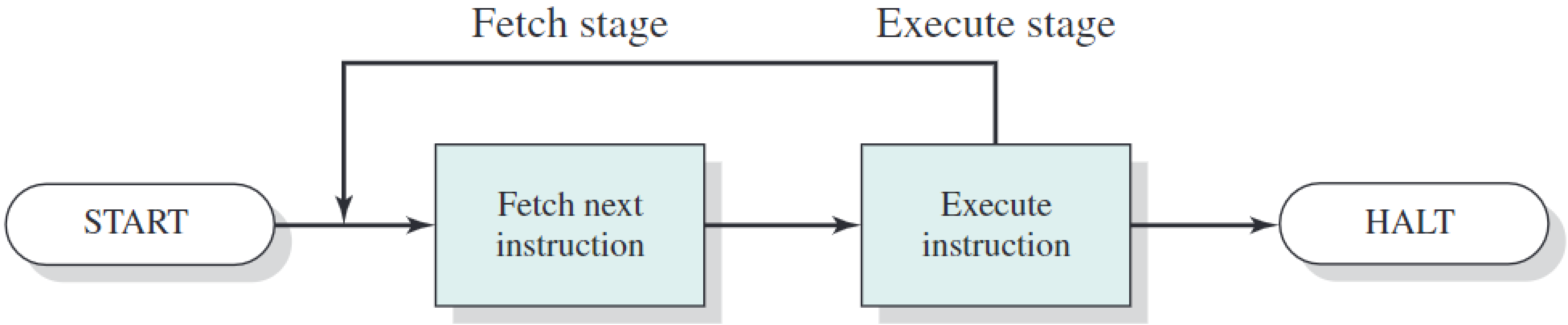
System Block Diagram



- The arithmetic/logic unit is the component of the CPU where data is held temporarily and where calculations take place.

- The control unit controls and interprets the execution of instructions. It does so by following a sequence of actions that correspond to the **fetch–execute** instruction cycle.
- The control unit determines the particular instruction to be executed by reading the contents of a program counter (PC), sometimes called an instruction pointer (IP).

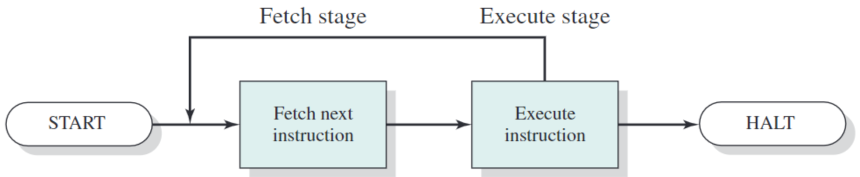
Fetch-Execute Cycle



- At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the next instruction to be fetched. Unless instructed otherwise, the processor always increments the PC after each instruction fetch so it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). For example, consider a simplified computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. The fetched instruction is loaded into the instruction register (IR). The instruction contains bits that specify the action the processor is to take.

Fetch-Execute Cycle (cont'd)

- The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:
  - **Processor-memory**: Data may be transferred from processor to memory, or from memory to processor.
  - **Processor-I/O**: Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
  - **Data processing**: The processor may perform some arithmetic or logic operation on data.
  - **Control**: An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor sets the program counter to 182. Thus, on the next fetch stage, the instruction will be fetched from location 182 rather than 150.

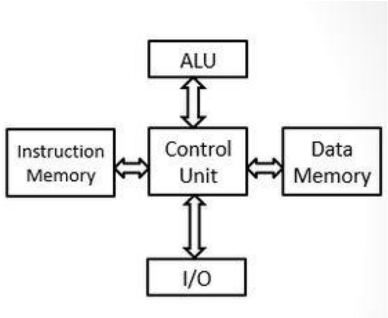


A few other details:

- Harvard v. Princeton (Von Neumann) architecture
- RISC v. CISC
- “Endian-ness”

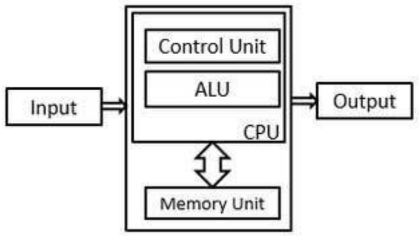
Architecture

- Harvard
  - Separate memory for instruction and data (also separate buses)
  - 1 clock cycle per instruction
  - Better performance characteristics
  - complex, pricey



Harvard Model

- Princeton (Von Neumann)
  - Memory is shared between instructions and data (combined bus)
  - 2 clock cycles per instruction
  - Lower performance characteristics
  - Simple, cheaper

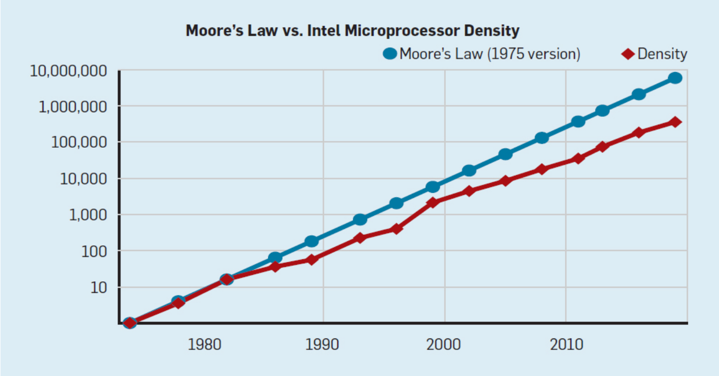


Von Neumann Model

https://www.polytechnichub.com/difference-harvard-architecture-von-neumann-architecture/

Architecture (cont'd)

- CISC
  - Complex Instruction Set Computing
    - Instructions can take several clock cycles
    - Variable length instructions
    - Large number of instructions
    - Compound addressing modes
    - Favours hardware (circuitry does the heavy lifting).



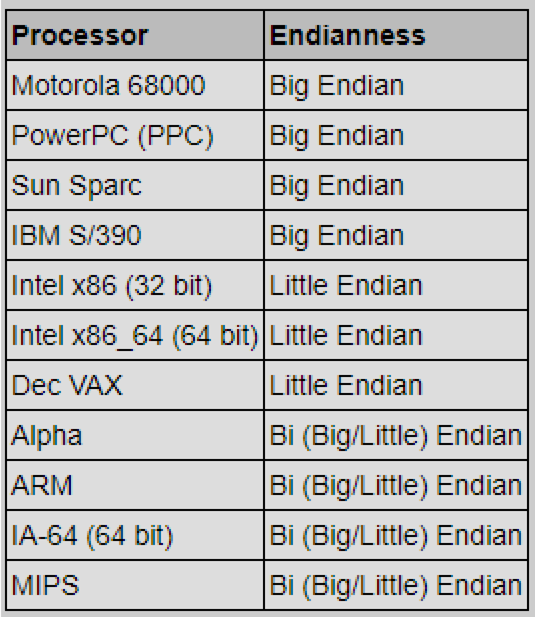
- RISC
  - Reduced Instruction Set Computing
    - Instruction will take 1 clock cycle
    - All instructions are same length
    - Smaller number of instructions
    - Limited addressing modes
    - Favours software (the compiler needs to build higher level functions from few, low-level instructions).

https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext

https://cacm.acm.org/magazines/2020/5/244325-will-risc-v-revolutionize-computing/fulltext

Architecture (cont'd)

- Endianness
  - Computer processors store data in either large (big) or small (little) endian format depending on the CPU processor architecture. The Operating System (OS) does not factor into the endianness of the system.
  - Big endian byte ordering is considered the standard.
  - Big endian byte ordering is in a form for human interpretation and is also the order most often presented by hex calculators.



Processor	Endianness
Motorola 68000	Big Endian
PowerPC (PPC)	Big Endian
Sun Sparc	Big Endian
IBM S/390	Big Endian
Intel x86 (32 bit)	Little Endian
Intel x86_64 (64 bit)	Little Endian
Dec VAX	Little Endian
Alpha	Bi (Big/Little) Endian
ARM	Bi (Big/Little) Endian
IA-64 (64 bit)	Bi (Big/Little) Endian
MIPS	Bi (Big/Little) Endian

- Big endian refers to the order where the most significant bytes come first.
  - This means that the bytes representing the largest values come first.

Decimal: 1025 ($2^{10} + 2^{1}$)

```
        Hex:  0x00000401
  BigEndian:  00000000 00000000 00000100 00000001
LittleEndian: 00000001 00000100 00000000 00000000
```

End of week 03

The CPU: Registers

- A register is a single, permanent storage location within the CPU used for a particular and defined purpose.
- A register is used to hold a binary value temporarily for storage, for manipulation, and/or for simple calculations.
- Each register is wired within the CPU to perform its specific role.
  - Each register serves a particular purpose. The register's size, the way it is wired, and even the operations that take place in the register reflect the specific function that the register performs in the computer:

    https://www.felixcloutier.com/x86/div
- Registers also differ from memory in that they are not addressed as a memory location would be, but instead are manipulated directly by the control unit during the execution of instructions.

The CPU: Registers (cont'd)

- Registers are used in many ways in a computer:
  - they may hold data being processed,
  - an instruction being executed,
  - a memory or I/O address to be accessed,
  - or even special binary codes used for some other purpose, such as codes that keep track of the status of the computer or the conditions of calculations that may be used for conditional branch instructions.
- Some registers serve many different purposes, while others are designed to perform a single, specialized task.

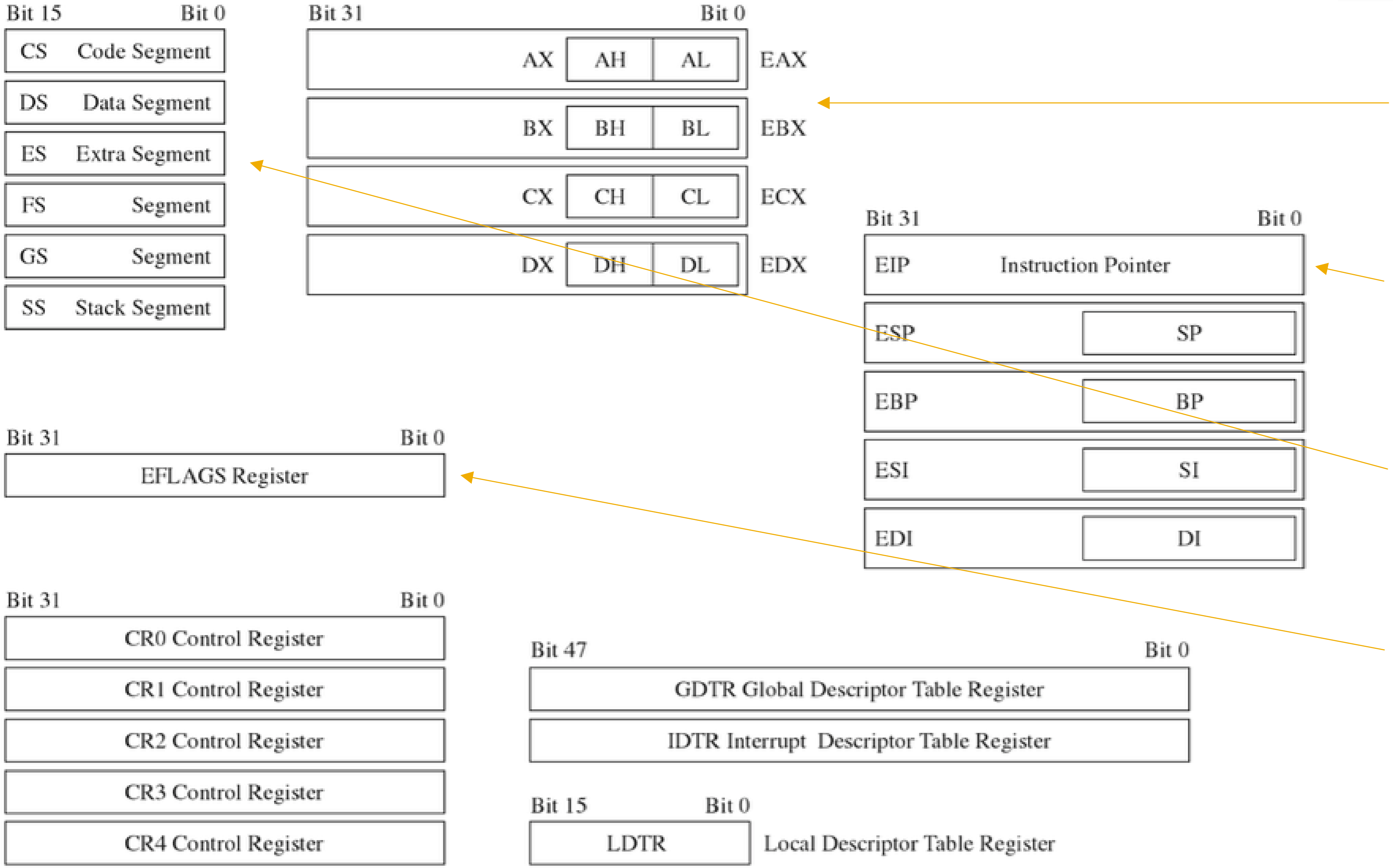
The CPU: Registers (cont'd)

- The **Accumulator**:
  - Modern CPUs provide several accumulators to act as "holding pens" for in process instructions and results; these are often known as general-purpose registers.
  - Some vendors also refer to general-purpose registers as user-visible or program-visible registers to indicate that they maybe accessed by the instructions in user programs.
- The control unit contains several important registers:
  - As already noted, the program counter register(PC or IP) holds the address of the current instruction being executed.
  - The instruction register (IR) holds the actual instruction being executed currently by the computer.
  - The memory address register (MAR) holds the address of a memory location. (to be discussed later)
  - The memory data register (MDR), sometimes known as the memory buffer register, will hold a data value that is being stored to or retrieved from the memory location currently addressed by the memory address register. (to be discussed later)
- The control unit will also contain several 1-bit registers, sometimes known as flags, that are used to allow the computer to keep track of special conditions such as arithmetic carry and overflow, power failure, and internal computer error. Usually, several flags are grouped into one or more status registers.

The CPU: Registers (cont'd)

- Most registers support four primary types of operations:
  - Registers can be loaded with values from other registers or from memory locations.
    - This operation destroys the previous value stored in the destination register, but the source register or memory location remains unchanged.
  - Data from another location can be added, subtracted, logically combined with the value previously stored in a register, leaving the sum, difference, changes in the register.
  - Data in a register can be modified:
    - shifted or rotated right or left by one or more bits.
    - Logical operations like AND, OR, XOR, etc
  - The value of data in a register can be tested for certain conditions, such as zero, positive, negative, or too large to fit in the register.

The CPU: Registers (cont'd)



Bit 15 Bit 0

CS Code Segment

DS Data Segment

ES Extra Segment

FS Segment

GS Segment

SS Stack Segment

Bit 31 Bit 0

AX AH AL EAX

BX BH BL EBX

CX CH CL ECX

DX DH DL EDX

Bit 31 Bit 0

EIP Instruction Pointer

ESP SP

EBP BP

ESI SI

EDI DI

Bit 31 Bit 0

EFLAGS Register

Bit 31 Bit 0

CR0 Control Register

CR1 Control Register

CR2 Control Register

CR3 Control Register

CR4 Control Register

Bit 47 Bit 0

GDTR Global Descriptor Table Register

IDTR Interrupt Descriptor Table Register

Bit 15 Bit 0

LDTR Local Descriptor Table Register

General registers

Used by the CPU during execution

Instruction pointer

Address of next instruction to execute

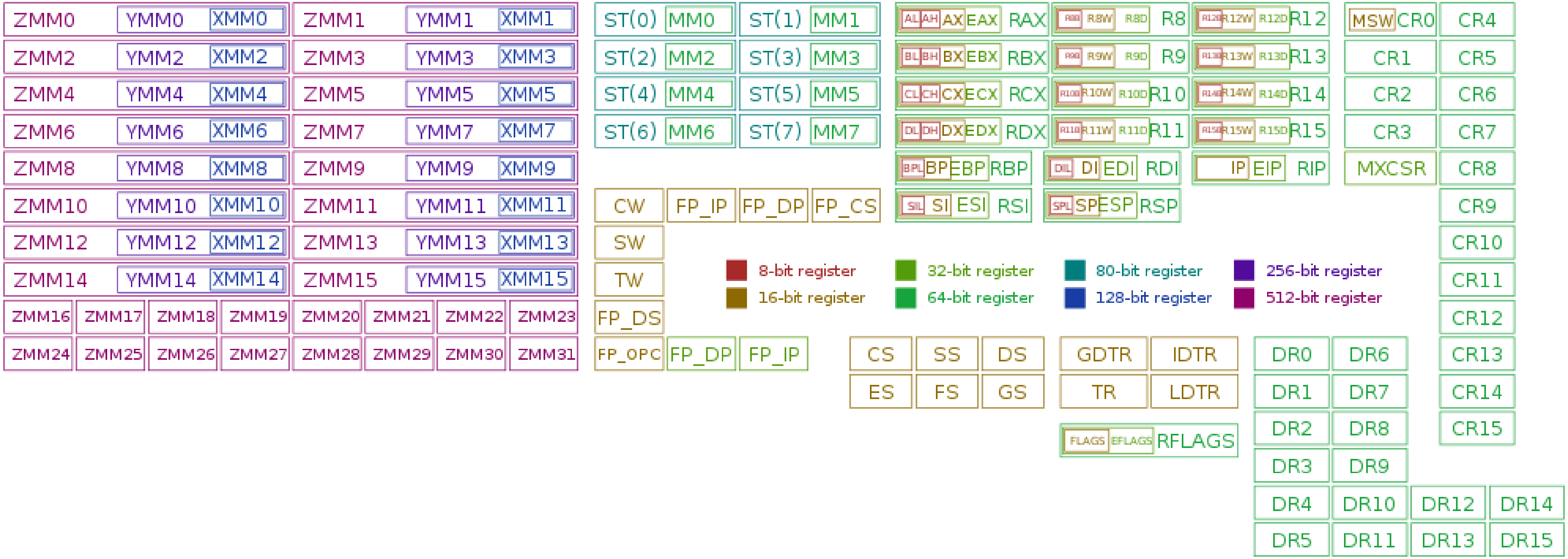
Segment registers

Used to track sections of memory

Status flags

Used to make decisions

X64…



ZMM0 YMM0 XMM0 | ZMM1 YMM1 XMM1 | ST(0) MM0 | ST(1) MM1 | AL AH AX EAX RAX | R8B R8W R8D R8 | R12B R12W R12D R12 | MSW CR0 | CR4

ZMM2 YMM2 XMM2 | ZMM3 YMM3 XMM3 | ST(2) MM2 | ST(3) MM3 | BL BH BX EBX RBX | R9B R9W R9D R9 | R13B R13W R13D R13 | CR1 | CR5

ZMM4 YMM4 XMM4 | ZMM5 YMM5 XMM5 | ST(4) MM4 | ST(5) MM5 | CL CH CX ECX RCX | R10B R10W R10D R10 | R14B R14W R14D R14 | CR2 | CR6

ZMM6 YMM6 XMM6 | ZMM7 YMM7 XMM7 | ST(6) MM6 | ST(7) MM7 | DL DH DX EDX RDX | R11B R11W R11D R11 | R15B R15W R15D R15 | CR3 | CR7

ZMM8 YMM8 XMM8 | ZMM9 YMM9 XMM9 | BPL BP EBP RBP | DIL DI EDI RDI | IP EIP RIP | MXCSR | CR8

ZMM10 YMM10 XMM10 | ZMM11 YMM11 XMM11 | CW | FP_IP | FP_DP | FP_CS | SIL SI ESI RSI | SPL SP ESP RSP | CR9

ZMM12 YMM12 XMM12 | ZMM13 YMM13 XMM13 | SW | CR10

ZMM14 YMM14 XMM14 | ZMM15 YMM15 XMM15 | TW | CR11

ZMM16 ZMM17 ZMM18 ZMM19 ZMM20 ZMM21 ZMM22 ZMM23 | FP_DS | CR12

ZMM24 ZMM25 ZMM26 ZMM27 ZMM28 ZMM29 ZMM30 ZMM31 | FP_OPC | FP_DP | FP_IP | CS SS DS | GDTR IDTR | DR0 DR6 | CR13

ES FS GS | TR LDTR | DR1 DR7 | CR14

FLAGS EFLAGS RFLAGS | DR2 DR8 | CR15

DR3 DR9

DR4 DR10 DR12 DR14

DR5 DR11 DR13 DR15

🟥 8-bit register	🟩 32-bit register	🟦 80-bit register	🟪 256-bit register
🟫 16-bit register	🟩 64-bit register	🟦 128-bit register	🟪 512-bit register

https://en.wikipedia.org/wiki/X86#x86_registers

The CPU: Instructions

- We have 2 categories of instructions:
  - privileged instructions
    - providing security, controlling memory access, and performing other functions.
    - the operating system will frequently be controlling many tasks and users, these instructions must not be available to the users' application programs.
  - application-level instructions
    - Also known as user-accessible instructions.
    - Programs that execute without privileges are said to execute in user space

The CPU: Instructions (cont'd)

- Data Movement Instructions
  - Move data from memory to general registers
  - Move data from general registers to memory,
  - Move data between different general registers
  - and, in some computers, move data directly between different memory locations without affecting any general register.
  - Variations on these instructions are frequently used to handle different data sizes. Thus, there may be:
    - a LOAD BYTE instruction,
    - a LOAD WORD(2 bytes),
    - a LOAD DOUBLE WORD(4 bytes), and
    - a LOAD QUAD WORD(8 bytes) within the same instruction set.

The CPU: Instructions (cont'd)

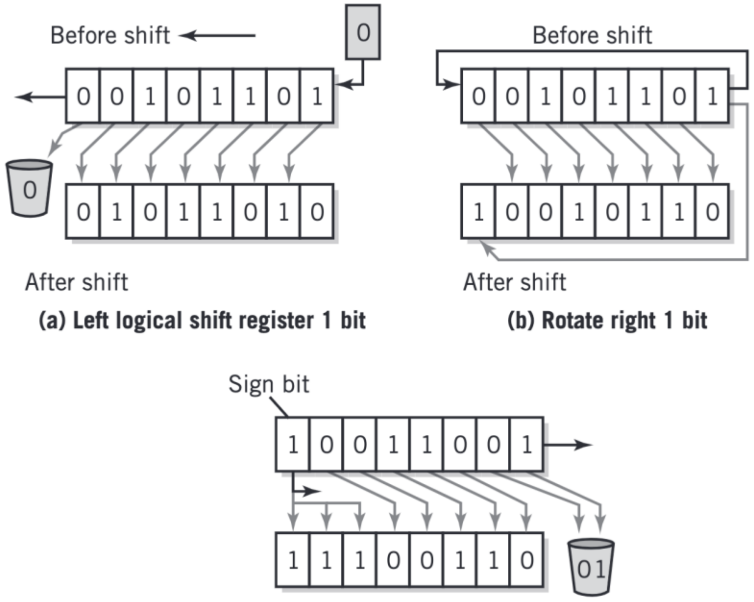
- Arithmetic Instructions
  - Every CPU instruction set includes integer addition and subtraction.
  - Provides instructions for integer multiplication and division.
  - Many instruction sets provide integer arithmetic for several different word sizes. As with the MOVE instructions, there may be several different integer arithmetic instruction formats providing various combinations of register and memory access in different addressing modes.
  - Most current CPUs also provide floating point arithmetic capabilities.
  - The instruction set generally provides standard arithmetic operations and instructions that convert data between various integer and floating-point formats. Some architectures also offer instructions for other, more specialized operations, such as square root, log, and trigonometry functions

The CPU: Instructions (cont'd)

- Boolean Logic Instructions
  - NOT instruction, which inverts the bits on a single operand,
  - AND,
  - (inclusive) OR, and
  - EXCLUSIVE-OR (XOR) instructions, which require a source and destination argument.
- Single Operand Manipulation Instructions
  - the instruction set will contain instructions for negating a value,
  - for incrementing a value,
  - for decrementing a value, and
  - for setting a register to zero.

The CPU: Instructions (cont'd)

- Bit Manipulation Instructions
  - provide instructions for setting and resetting individual bits in a data word.
- Shift and Rotate Instructions
  - Shift and rotate operations implement multiplication and division.
  - Logical shift instructions simply shift the data as you would expect, and zeros are shifted in to replace the bit spaces that have been vacated.


Typical Register Shifts and Rotates

(a) Left logical shift register 1 bit

(b) Rotate right 1 bit

(c) Right arithmetic shift 2 bits

The CPU: Instructions (cont'd)

- Program Control Instructions
  - control the flow of a program.
  - include jumps and branches:
    - both unconditional and conditional, and
    - CALL and RETURN instructions.
- Stack Instructions
  - stacks are also known as LIFO, for last-in, first-out, structures.
  - Computers do not generally provide special memory for stack use, although many machines provide special STACK instructions to simplify the bookkeeping task.

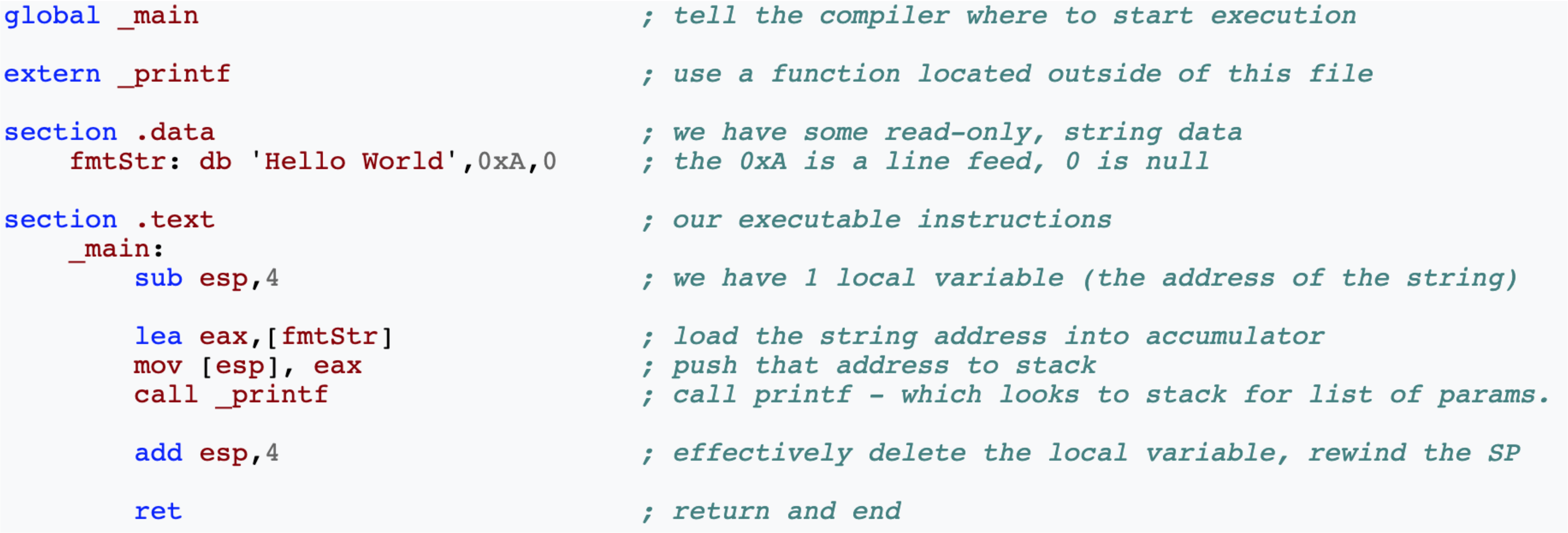
The CPU: Instructions (cont'd)

- Multiple Data Instructions
  - Multiple data instructions perform a single operation on multiple pieces of data simultaneously.
  - commonly known as SIMD instructions. (Single Instruction, Multiple Data).
  - The SIMD instructions provided on current Intel processors are typical.
  - One interesting characteristic of some multimedia arithmetic operations is that when the result of a calculation meets or exceeds a certain maximum value, the result is simply set to the maximum value, instead of overflowing. This characteristic is known as saturation.

<Assembly language programming>

Assembly Language Programming

- From last class:

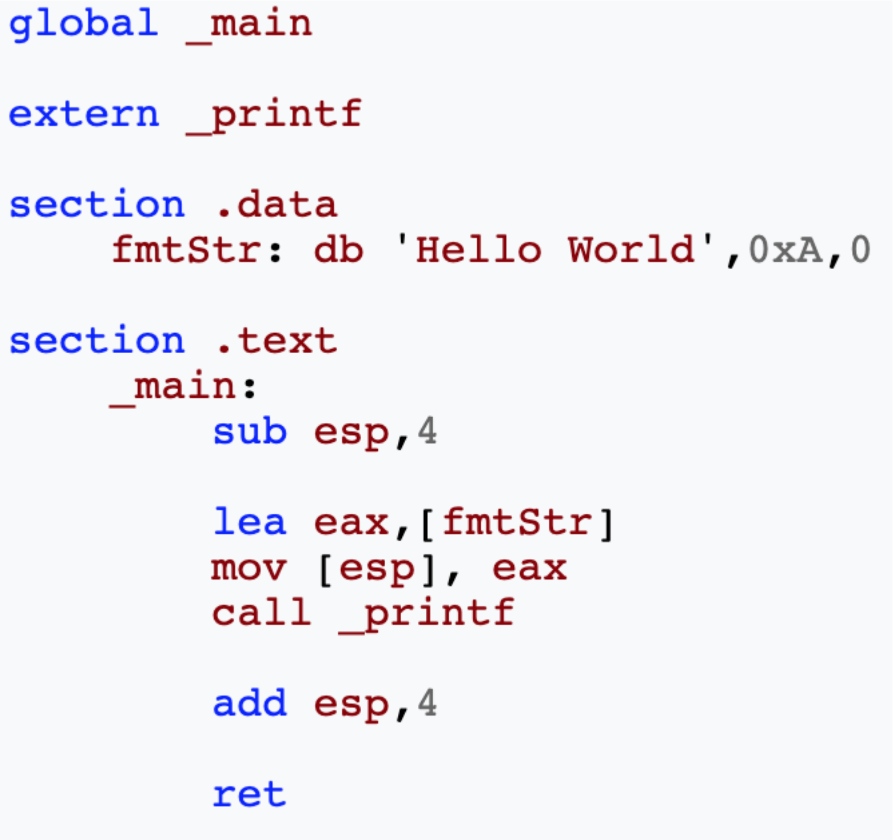


```
global _main                            ; tell the compiler where to start execution

extern _printf                          ; use a function located outside of this file

section .data                           ; we have some read-only, string data
    fmtStr: db 'Hello World',0xA,0      ; the 0xA is a line feed, 0 is null

section .text                           ; our executable instructions
    _main:
        sub esp,4                       ; we have 1 local variable (the address of the string)

        lea eax,[fmtStr]                ; load the string address into accumulator
        mov [esp], eax                  ; push that address to stack
        call _printf                    ; call printf - which looks to stack for list of params.

        add esp,4                       ; effectively delete the local variable, rewind the SP

        ret                             ; return and end
```

Assembly Language Programming (cont'd)

We are going to define the 'entry' point of this program to the label called _main. Once this program is loaded into memory (and all the addresses are sorted out), execution will commence at *_main:*

We tell the compiler that we are using a function called *_printf*, but it is not defined here, it is defined external to the program. The compiler will write an entry for the function into the compiled program. The linker will populate the actual address of the *_printf* function.



```
global _main

extern _printf

section .data
    fmtStr: db 'Hello World',0xA,0

section .text
    _main:
        sub esp,4

        lea eax,[fmtStr]
        mov [esp], eax
        call _printf

        add esp,4

        ret
```

Assembly Language Programming (cont'd)

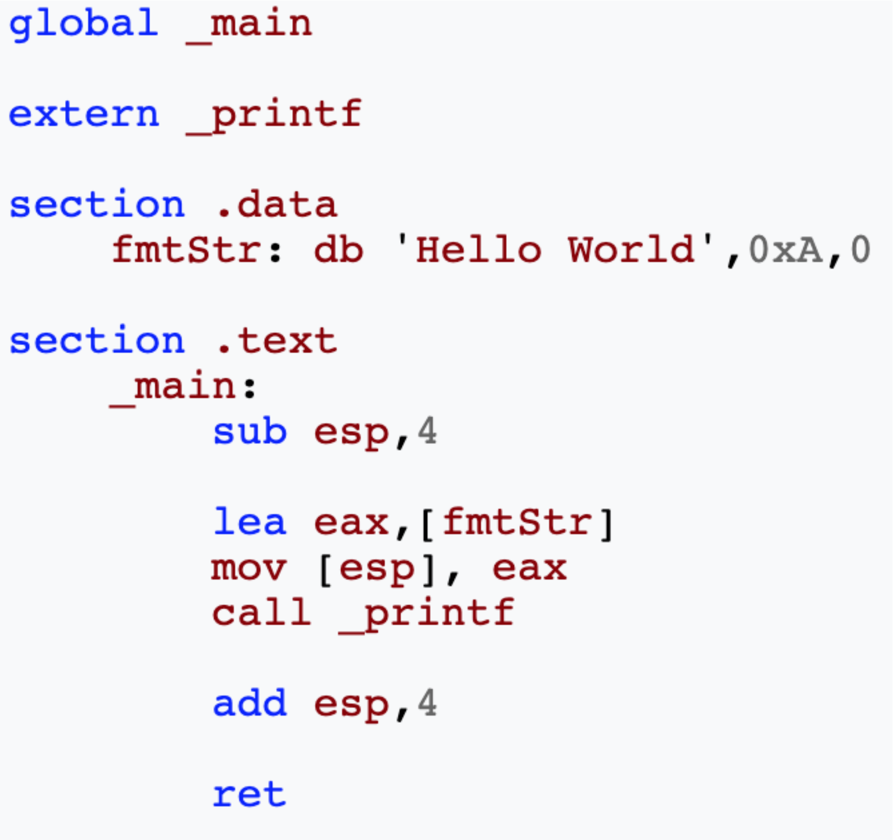
When a variable is included in **section .data**, memory is allocated for that variable when the source code is assembled and linked to an executable. Variable names are symbolic names, and references to memory locations and a variable can take one or more memory locations. The variable name refers to the start address of the variable in memory.

This variable is called a *string*, which is a contiguous list of characters. A string is a "list" or "array" of characters in memory.

A line feed or new line has code 10, and NULL has code 0.

section .data can also contain constants, which are values that cannot be changed in the program. They are declared in the following format:

<constant name>   equ   <value>



```
global _main

extern _printf

section .data
    fmtStr: db 'Hello World',0xA,0

section .text
    _main:
        sub esp,4

        lea eax,[fmtStr]
        mov [esp], eax
        call _printf

        add esp,4

        ret
```

Assembly Language Programming (cont'd)

section .text is where all the action is. This section contains the program code and starts with *_main:*

First make space for a single 32bit address on the stack

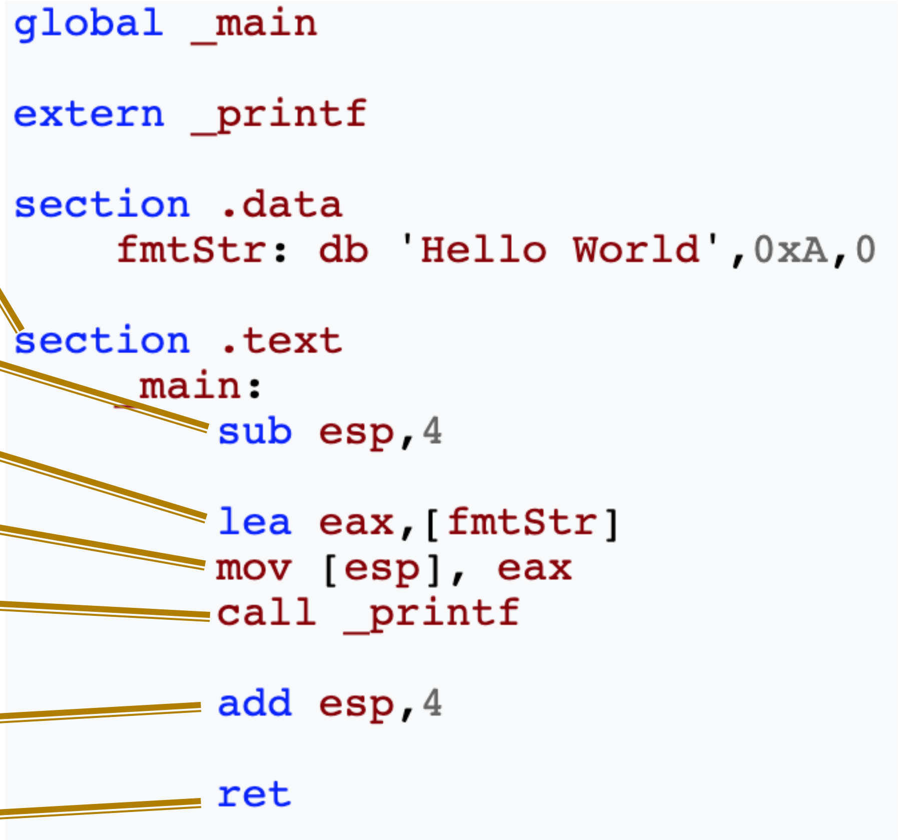
Load the address of the data string into the EAX register

Push that address onto the stack

Call *_printf*, which knows to look on the stack for the thing to display

Done, clear the address from the stack

Done.



```
global _main

extern _printf

section .data
    fmtStr: db 'Hello World',0xA,0

section .text
     _main:
        sub esp,4

        lea eax,[fmtStr]
        mov [esp], eax
        call _printf

        add esp,4

        ret
```