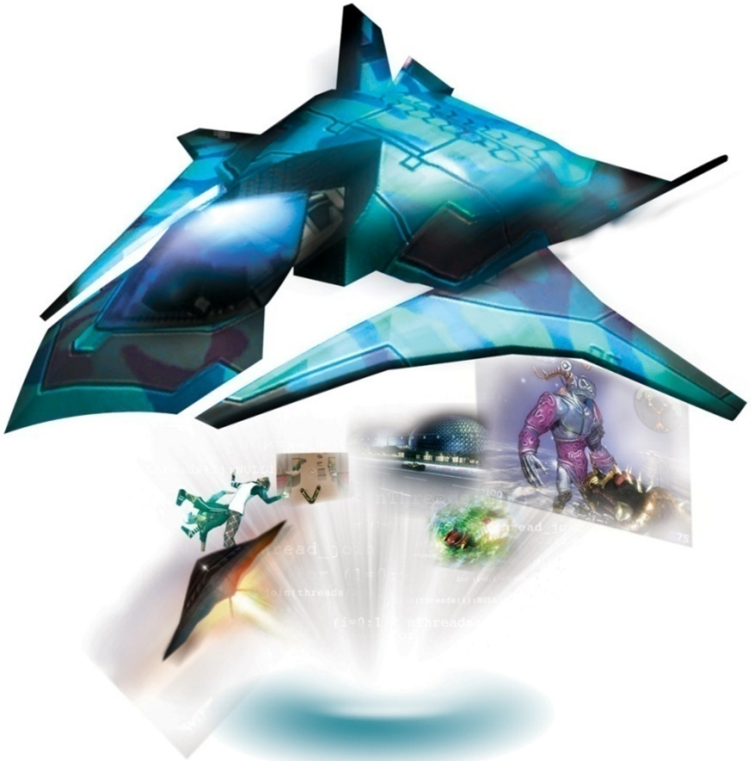
DM2112
DIGITAL ENTERTAINMENT SYSTEMS

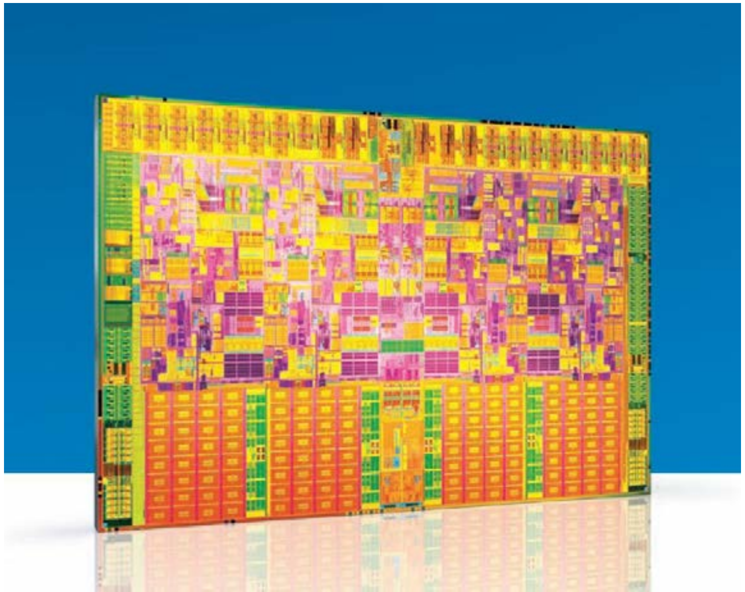


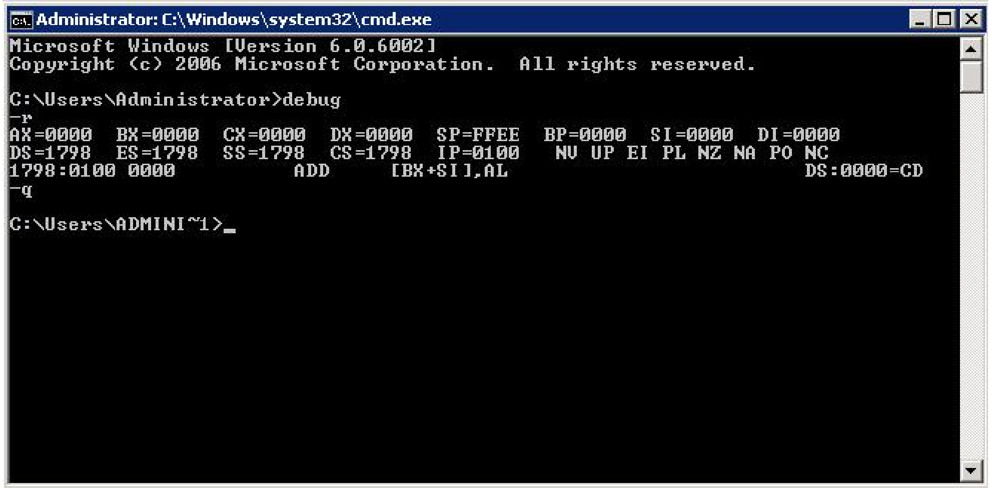
Week 12: Microcomputer Architecture

Today's Menu

- Basic Architecture & Registers
- Memory Addressing
- Programming the Microprocessor
- Introduction to Assembly Language
  - The MOV, PUSH, and POP instructions

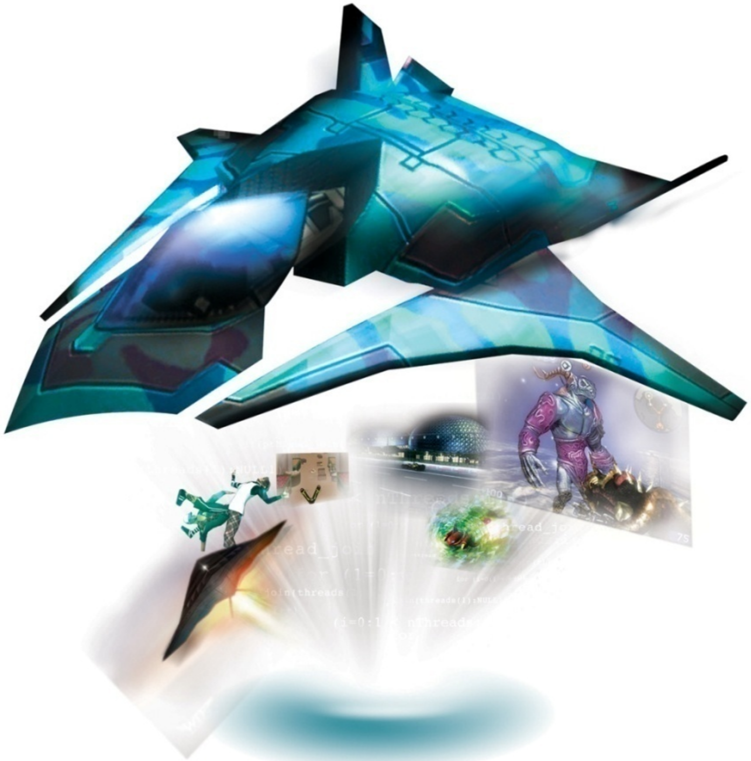
Introduction to Intel the Microprocessor Architecture



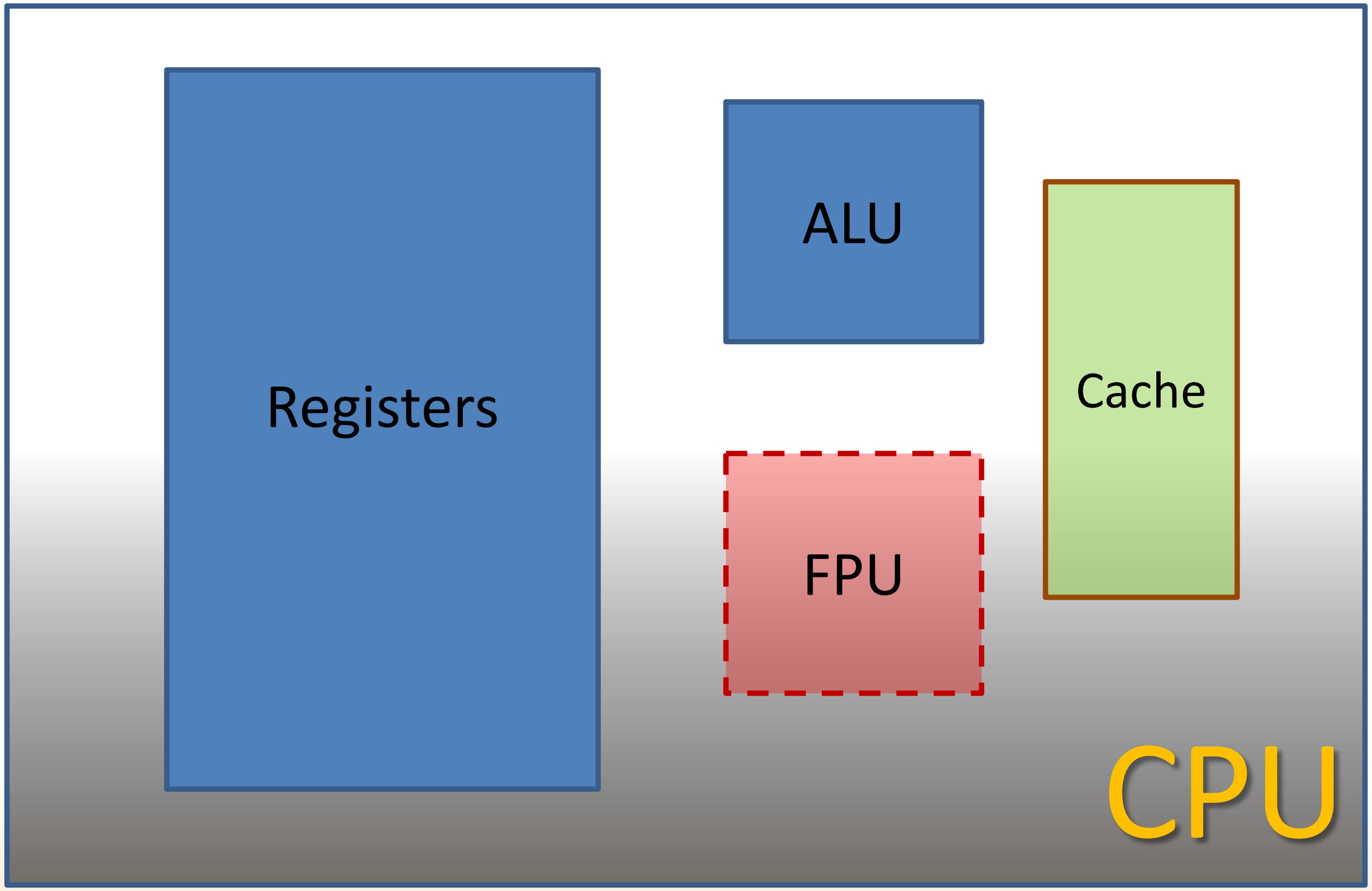


```
Microsoft Windows [Version 6.0.6002]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1798  ES=1798  SS=1798  CS=1798  IP=0100   NV UP EI PL NZ NA PO NC
1798:0100 0000          ADD     [BX+SI],AL                         DS:0000=CD
-q

C:\Users\ADMINI~1>_
```

BASIC ARCHITECTURE & REGISTERS



In a Sense, CPU contains…



The Basic Architecture of a 16-Bit CPU

Let's recap: What's a register?

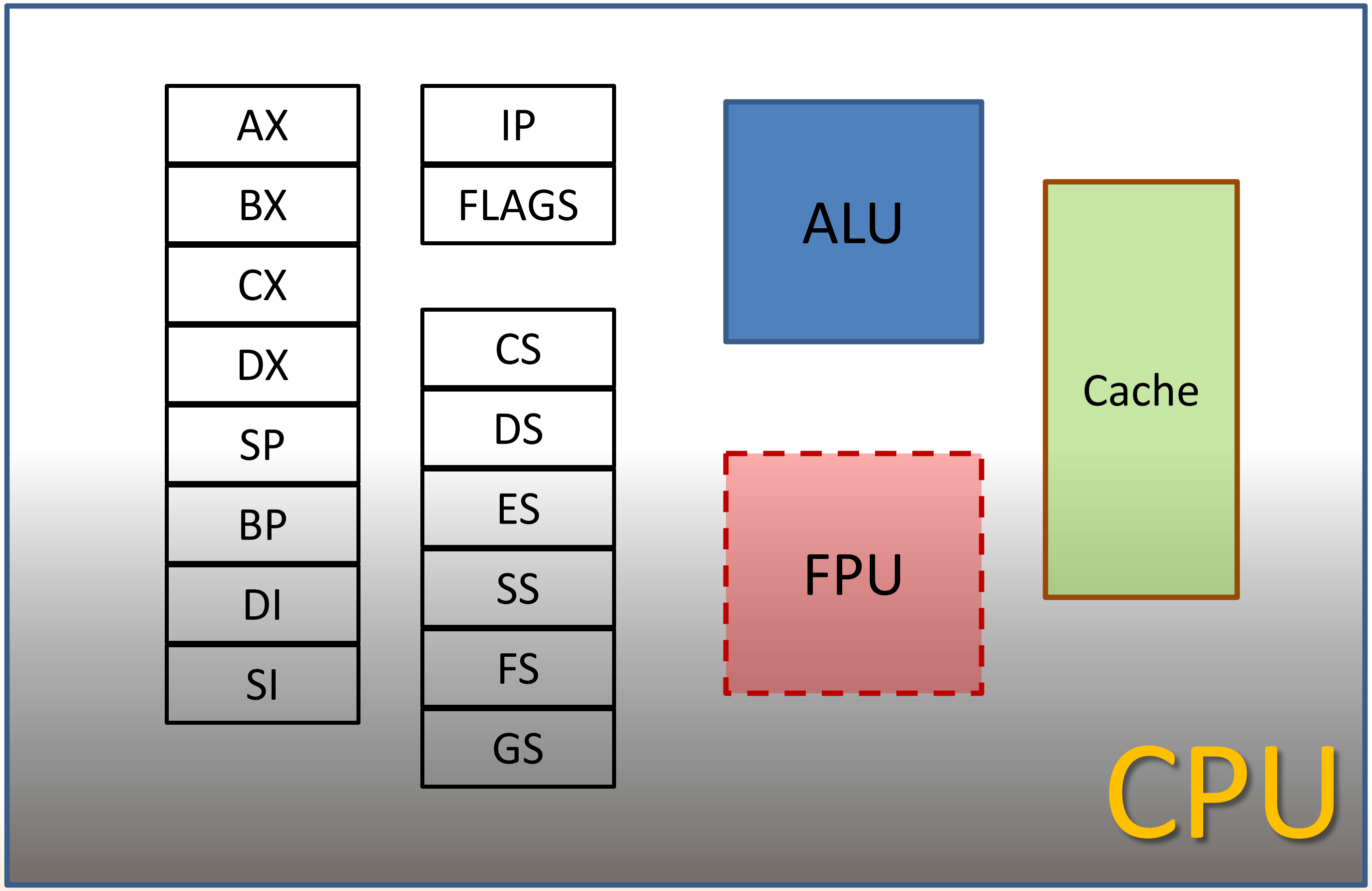
- Here's an overview of registers found in a 16-Bit Intel CPU
- Examples : Intel 8086, 80286 (generally known as x86 processors)

16 bits	
AX	**Accumulator**
BX	**Base index**
CX	**Count**
DX	**Data**
SP	**Stack pointer**
BP	**Base pointer**
DI	**Destination index**
SI	**Source index**

16 bits	
IP	**Instruction pointer**
FLAGS	**Flags**

CS	**Code segment**
DS	**Data segment**
ES	**Extra segment**
SS	**Stack segment**
FS	
GS	

In a Sense, CPU contains…



The Basic Architecture of a 16-Bit CPU

- The Intel microprocessor consists of registers that store data. We program the CPU by storing and moving data within registers.

- There are 4 general registers. One of the registers is the accumulator (called AX). This is its structure in a 16-bit CPU:

AH	AL

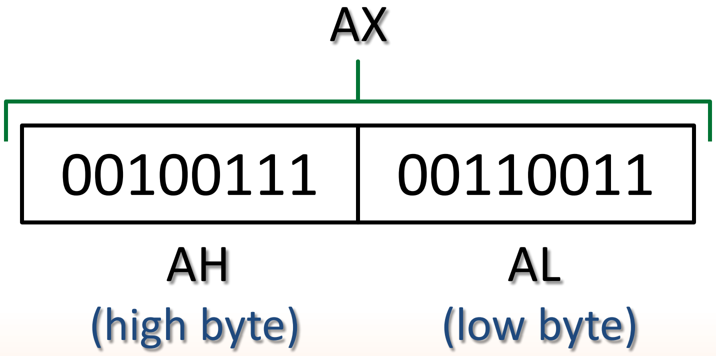
AX

AX stores the result from the ALU

The AX Register

- AX is split into **AH** and **AL**.
  - These are 8-bit registers that together store a 16 bit data.
  - AH stores the higher 8 bits (high order byte) while AL stores the lower 8 bits (low order byte).

- E.g. a 16-bit word such as 0010011100110011 is stored in AX as



AX

00100111	00110011

AH (high byte)

AL (low byte)

- AX is used for arithmetic instructions such as addition, multiplication, subtraction and division.

The BX, CX, and DX Registers

The 3 other general registers are called **BX** (base index), **CX** (count), and **DX** (data). They also have high order and low order byte registers (BH, BL, CH, CL, DH, DL).

BH	BL	**BX**
CH	CL	**CX**
DH	DL	**DX**

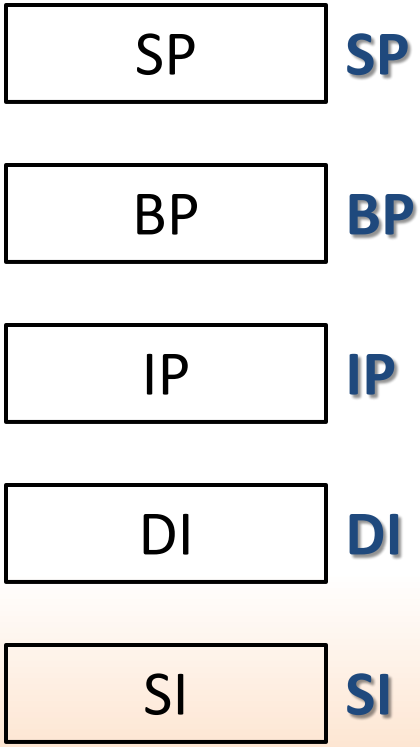
BX is used for memory addressing. It holds the offset address of a location in memory.

CX is a general register that stores the value of a counter. This counter is used for instructions such as shift and rotate.

DX is a general register that holds part of the result from arithmetic operations.

The SP, BP, IP, SI, and DI Registers

There are 3 pointer registers called **SP** (stack pointer), **BP** (base pointer) and **IP** (instruction pointer), and 2 index registers called **SI** (stack index) and **DI** (data index).



SP is used to address stack memory.

BP is used to point to a memory location.

IP is a special-purpose register used to find the next instruction to execute.

DI and **SI** are used to address string destination and source data for the string instructions respectively.

The FLAGS Register

The **FLAGS** register holds data that indicates the condition of the microprocessor and controls its operation.

- 16 bits total
- Each flag is 1 bit in size

- The rightmost 5 flags and the overflow flag change after many arithmetic and logic instructions execute. These 6 flags never change for any data transfer or program control operation.

- These 6 flags are called the **C** (carry), **P** (parity), **A** (auxiliary carry), **Z** (zero), **S** (sign), and **O** (overflow) flags.

The Contents of the FLAGS Register

Reserved bits

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
	NT	IOP 1	IOP 0	O	D	I	T	S	Z		A		P		C	**FLAGS**

C holds the carry after addition or the borrow after subtraction.

P is a 0 for odd parity and a 1 for even parity. Parity is the count of ones in a binary number expressed as even or odd. E.g. The binary number 0111 is odd parity because it contains 3 binary one bits (an odd number of one bits). The binary number 0101 is even parity.

A holds the half-carry after addition or the borrow after subtraction between bits positions 3 and 4 of the result.

Z shows that the result of an arithmetic or logic operation is zero. If Z=1, the result is zero. If Z=0, the result is not zero.

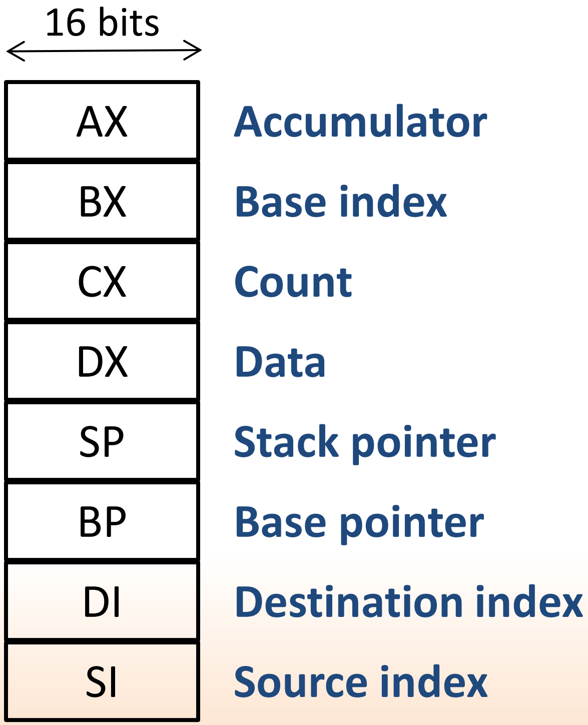
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
	NT	IOP 1	IOP 0	O	D	I	T	S	Z		A		P		C	**FLAGS**

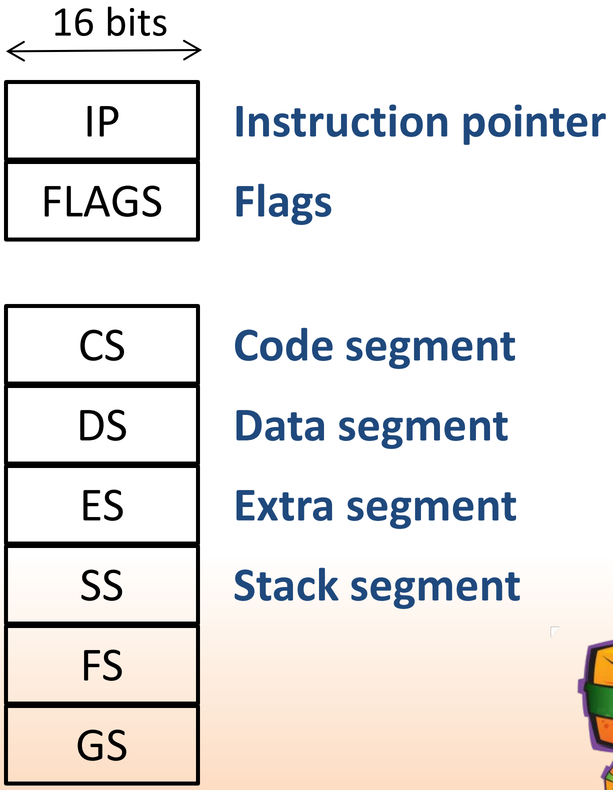
S holds the sign of the result after an arithmetic or logic instruction executes. If S=1, the sign bit is set or negative. If S=0, the sign bit is cleared or positive.

O indicates that the result from addition or subtraction has exceeded the capacity of the machine.

Recap (16-bit CPU)

CPU processing work mainly involves shifting bits of data in & out of registers to perform useful computation (i.e. add, subtract, mutiply, etc).

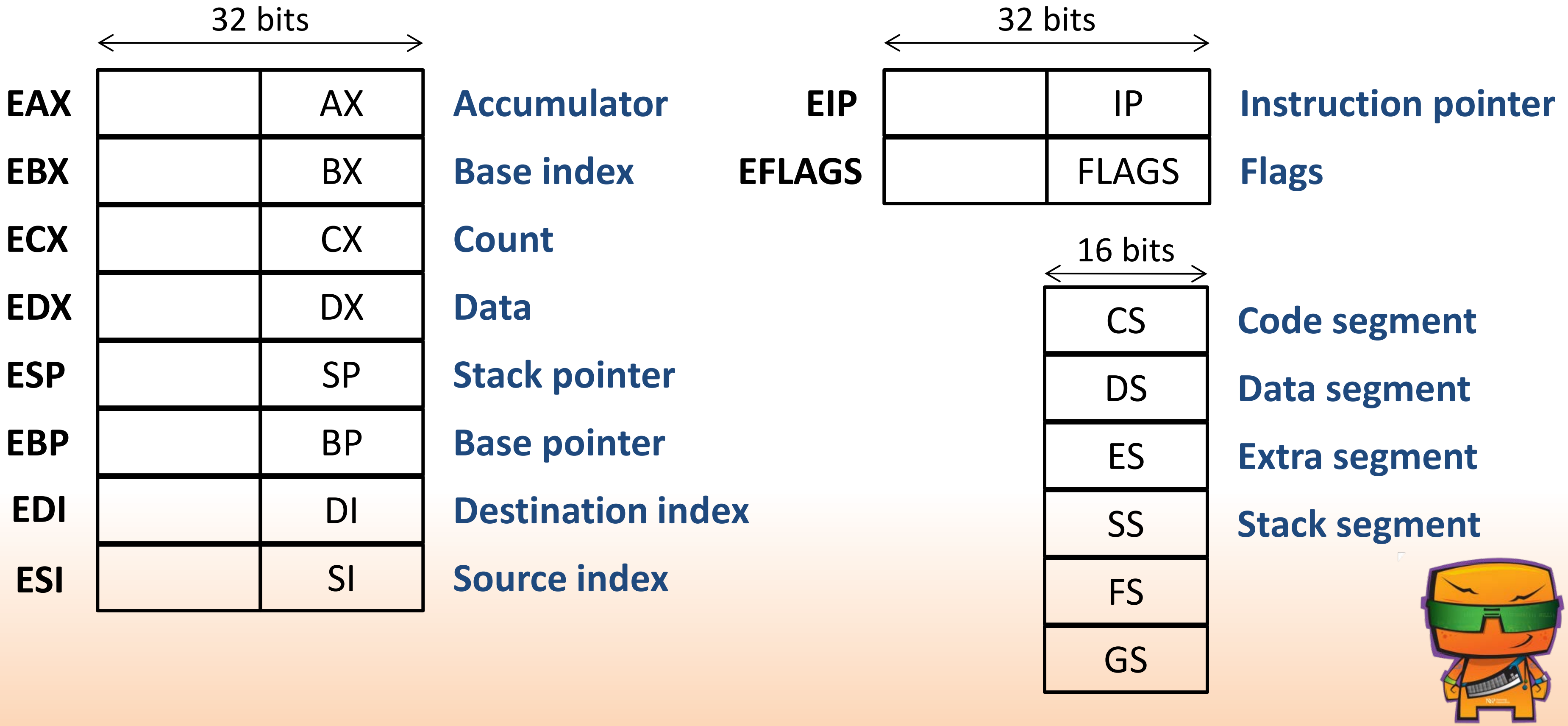




The Basic Architecture of a 32-Bit CPU

- In a 32-bit CPU all except the segment registers are expanded so that they can hold 32 bits of data. The names of these expanded registers have the letter ‘**E**’ in front of them. The 32-bit names are shown below on the left hand side.

- Intel 80386 and onwards are 32-bit processors. They’re also refered to as IA-32 processors.
  - ➢ Intel also has 64-bit processors. They are known as Intel 64 processors.
  - ➢ We will not learn about them in this module, except to say that they also use mostly 32-bit registers, and have some specialised 64-bit registers that have the ‘**R**’ prefix in their names.



32 bits

EAX		AX	**Accumulator**
EBX		BX	**Base index**
ECX		CX	**Count**
EDX		DX	**Data**
ESP		SP	**Stack pointer**
EBP		BP	**Base pointer**
EDI		DI	**Destination index**
ESI		SI	**Source index**

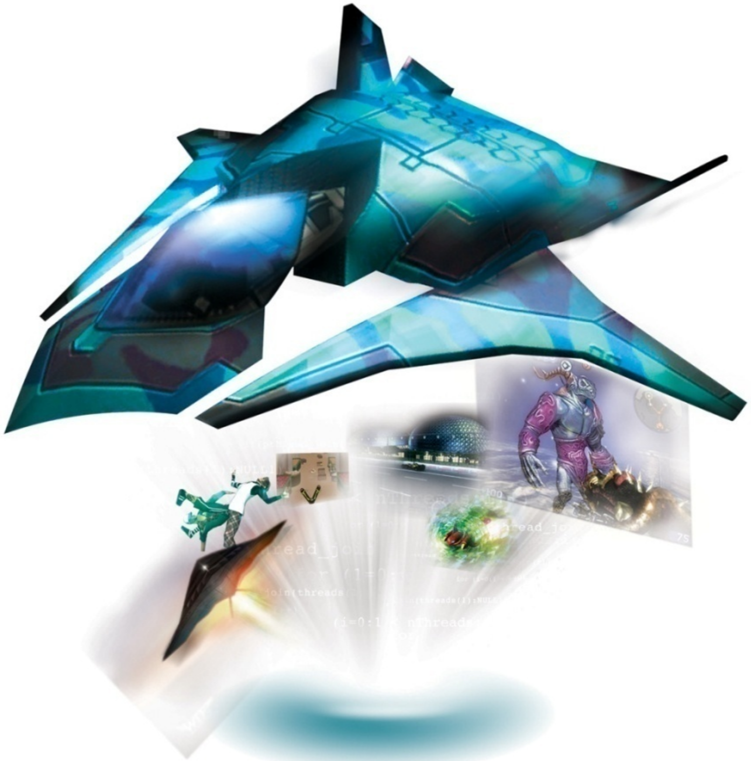
32 bits

EIP		IP	**Instruction pointer**
EFLAGS		FLAGS	**Flags**

16 bits

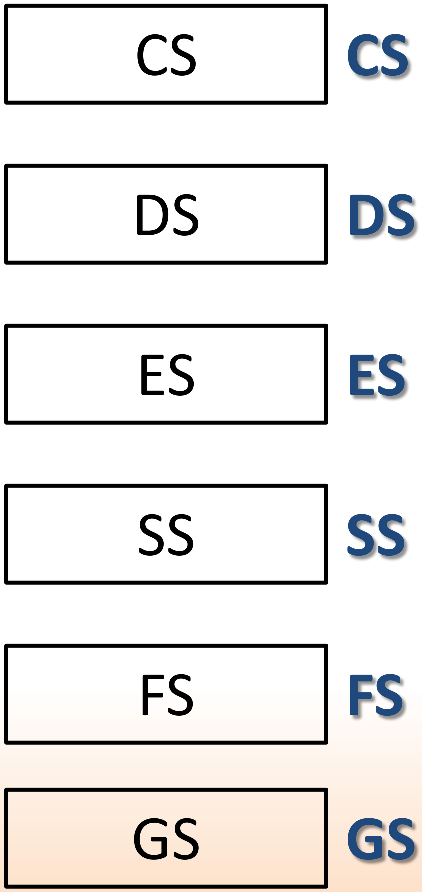
CS	**Code segment**
DS	**Data segment**
ES	**Extra segment**
SS	**Stack segment**
FS	
GS	

MEMORY ADDRESSING



The Segment Registers

The segment registers store the address of a section of memory (called a segment). Values in the segment registers are combined with the values in other registers to generate memory addresses.



CS (code segment) is a section of memory that holds the code used by the microprocessor.

DS (data segment) is a section of memory that contains the data used by a program.

ES (extra segment) is a section of memory that is used by some of the string instructions.

SS (stack segment) defines the area of memory used for the stack.

FS and **GS** are registers that can be used to define 2 extra segments of memory.

Memory Addressing Modes – Real Mode (1 MB RAM)

Intel microprocessors can operate in 2 modes: real or protected mode.

- Real mode allows access to only address first 1 MB of physical memory.
  - ➢ This first MB is called real memory or conventional memory.

- Microsoft Windows does not use Real mode
  - ➢ But all Intel microprocessors begin operation in Real mode by default when the processor is first started (i.e. booted) or reset.

- We take an in-depth look at real mode memory addressing because it gives a good example of how the segment registers are used.

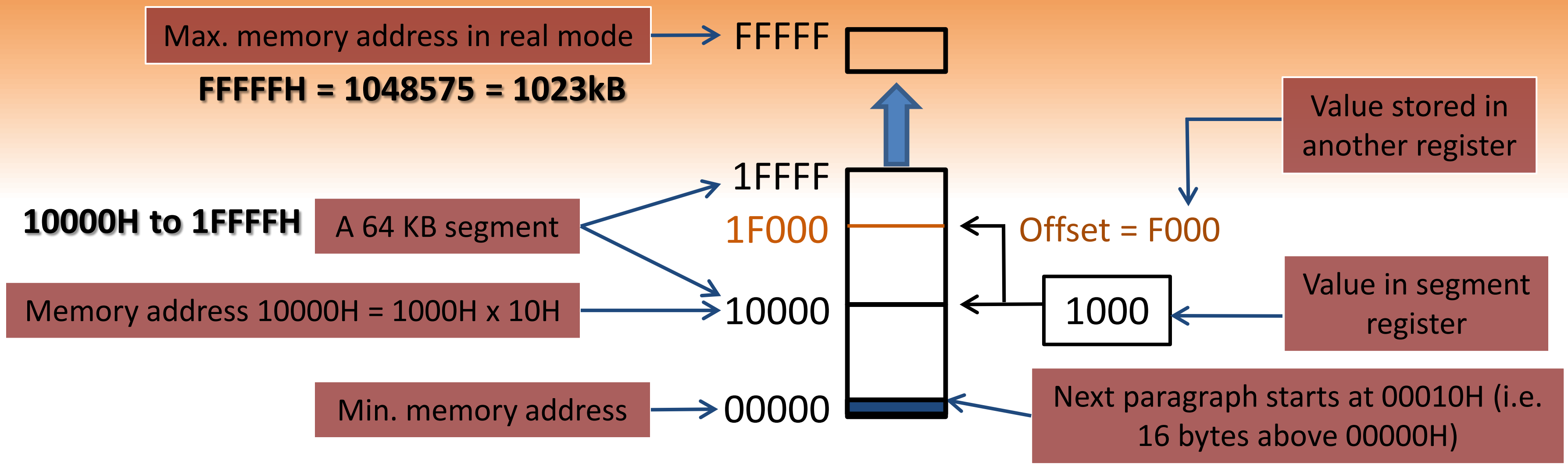
Real Mode – Segments & Offsets

In real mode, all memory addresses consist of a segment address and an offset address (also called a displacement).

Memory addresses consist of 2 parts. These parts are stored in a segment register and another register.

- In other words, the memory is divided into segments, and the offset address is used to access the memory location that contains the data.
- The value in the segment register contains the starting address of the segment, and the value of the segment address (i.e. the offset) is stored in another register.
- For convenience we will deal with numbers in hexadecimal form, although all values in a microprocessor are stored as binary numbers.
- Each segment in the real mode always has a length (i.e. a size) of 64k bytes (65535 bytes).

Memory in Real Mode



Max. memory address in real mode → FFFFF

FFFFFH = 1048575 = 1023kB

Value stored in another register

10000H to 1FFFFH A 64 KB segment → 1FFFF, 10000

1F000

Offset = F000

Memory address 10000H = 1000H x 10H → 10000

1000

Value in segment register

Min. memory address → 00000

Next paragraph starts at 00010H (i.e. 16 bytes above 00000H)

In memory diagrams it's convenient to use hexadecimal numbers to show memory addresses.

The maximum memory address in real mode is FFFFFH (the 'H' just means the address is written in hexadecimal. Similar to '0x' i.e. 0xFFFFF).

In this diagram the segment register has a value of 1000H, yet the starting address of this segment is 10000H. This is because in real mode each segment register has a value of 0H appended by the microprocessor (i.e. padded with 0H) on the rightmost end of the value. This turns the segment register into a 20-bit one (16+4 bits) and thus allows it to access a location within the first 1 MB of memory (a 16-bit register is not wide enough to access up to 1 MB of memory).

Since the segment addresses have a 0H appended, memory segments can only begin at every 16-byte boundary. This boundary is called a paragraph of memory.

Examples of Real Mode Segment Addresses

Segment Register	Starting Address		Ending Address
2000H	20000H	+ FFFFH →	2FFFFH
2001H	20010H		3000FH
2100H	21000H		30FFFH

The ending address of a segment is always FFFFH more than the starting address (because each segment is 64 KB in size).

The segment and offset address is sometimes written as a pair of segment-value:offset-value. E.g. 1000:2000 means the segment address is 10000H and the offset address is 2000H. So the address is 12000H.

Default Segment & Offset Registers

The microprocessor has a set of rules that define the combination of segment and offset registers. These rules apply in both real and protected modes.

The following table shows the default register combinations, and the specialized purpose of the generated addresses.

Segment	Offset	Special Purpose
CS(Code Segment)	IP (Instruction Pointer)	Instruction address
SS(Stack Segment)	SP (Stack Pointer) or BP (Base Pointer)	Stack address
DS(Data Segment)	BX (Base Index), DI (Dest. Index), SI (Source Index), an 8- or 16-bit number	Data address
ES (Extra Segment)	DI (for string instructions)	String destination address

The following table shows the default 32-bit register combinations.

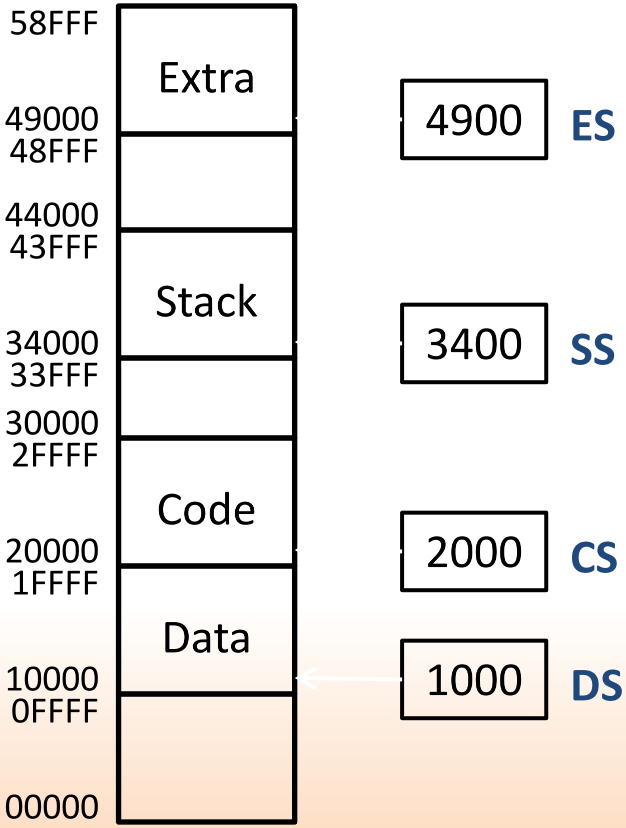
Segment	Offset	Special Purpose
CS	EIP	Instruction address
SS	ESP or EBP	Stack address
DS	EAX, EBX, ECX, EDX, EDI, ESI, an 8- or 16-bit number	Data address
ES	EDI (for string instructions)	String destination address

The code segment register is always used with the instruction pointer to address the next instruction in a program. The CS:IP (or CS:EIP) combination locates the next instruction to be executed by the microprocessor.

Another default combination is SS:SP (SS:ESP) or SS:BP (SS:EBP). This is used to access data in the stack. The stack is an area of memory for keeping temporary data.

Example of Memory Segments

This is an illustration of a memory system that shows values in some of the segment registers and how they are used to allocate the memory for 4 segments.



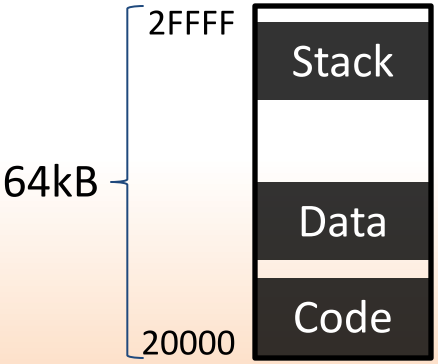
These are the addresses for the segments:

Segment	Start	End
Code	20000H	2FFFFH
Data	10000H	1FFFFH
Extra	49000H	58FFFH
Stack	34000H	43FFFH

Overlapping Memory Segments

- When a program is loaded into memory by DOS, it (i.e. DOS) will automatically calculate and assign the segment starting addresses (and thus load in the correct values into the segment registers).

- Segments need not occupy a full 64 KB of memory. They can touch or even overlap each other. Think of segments as windows that can be moved over any area of memory to access code or data.



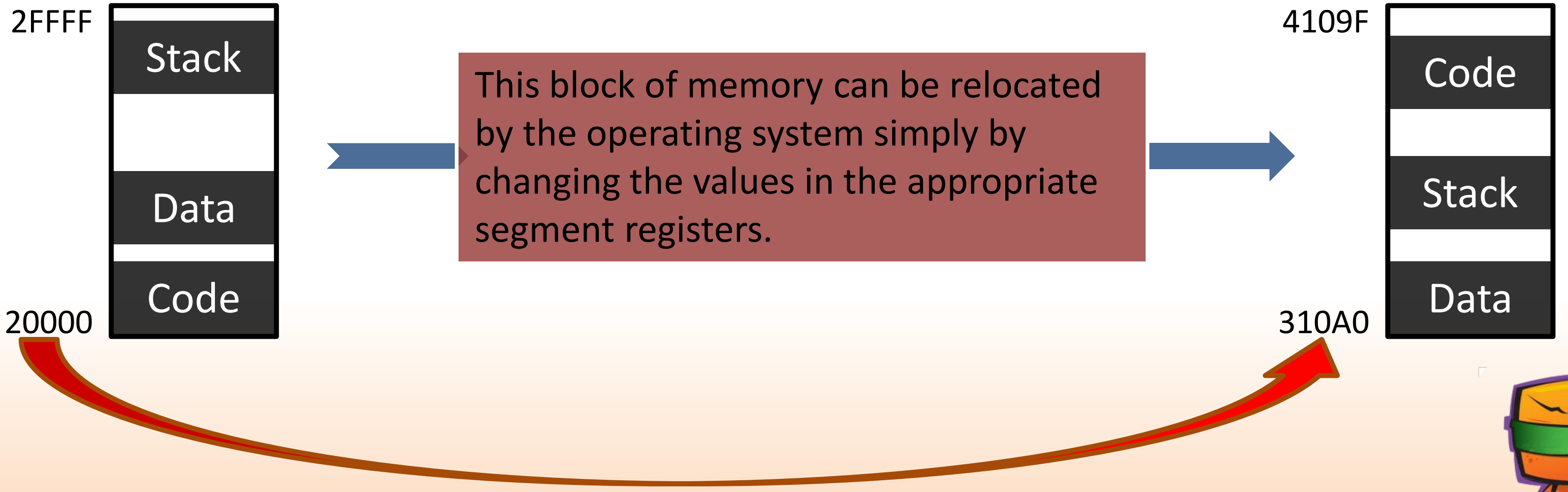
This is an example of segments overlapping. The data and stack segments overlap with the code segment, and the data segment overlaps with the stack segment.

Overlapping occurs because although each segment occupies 64 kB, not all of it needs to be actually used up by the segment.

This means segments do not need to start and end at the 64 kB boundaries. They can start and end at any 16 KB (i.e. paragraph) boundary.

Why Use Memory Segments?

Memory is divided into segments so that it's easy for the operating system to swap code and data in and out of memory when the need arises. All that is needed to relocate the code and data is for the operating system to change the values of the segment registers. The offset registers need not be changed.



This block of memory can be relocated by the operating system simply by changing the values in the appropriate segment registers.

Protected Mode Memory Addressing (> 1 MB RAM)

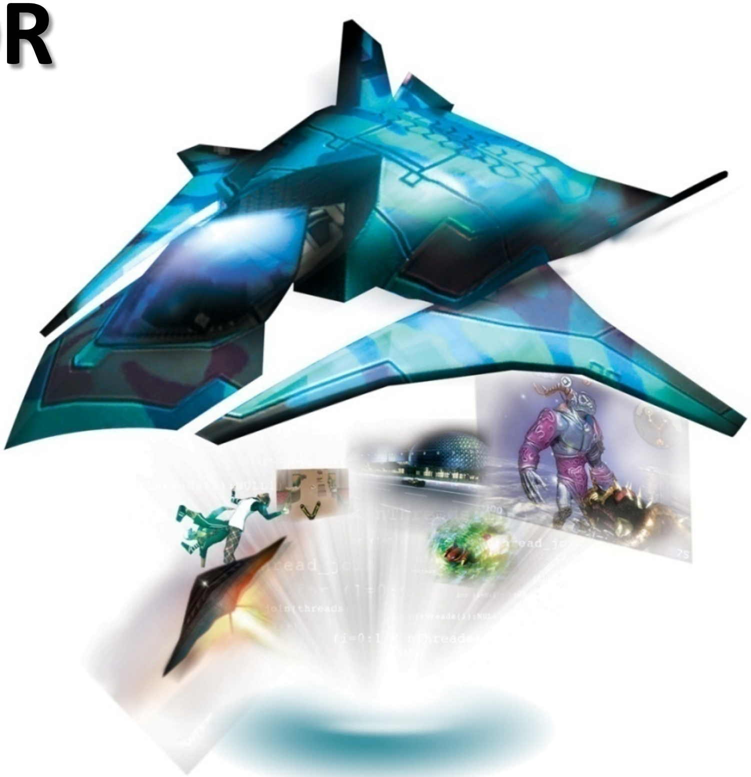
Protected mode memory addressing allows the processor (Intel 80286 and above) to access data and programs located above, as well as within, the first 1 MB of memory.

- Difference between protected mode and real mode
  - Segment address is no longer present.
  - Segment register now contains selector that selects a descriptor from a descriptor table.
  - This descriptor describes the memory segment’s location, length, and access rights.

- Offset register can now also hold a 32-bit number (instead of 16-bit)
  - Can address up to 4 GB.

- Programs written to function in the real mode will still function in protected mode because the segment and offset registers are still used to store memory addresses. The difference is in how the segment register is interpreted by the microprocessor.

PROGRAMMING THE MICROPROCESSOR

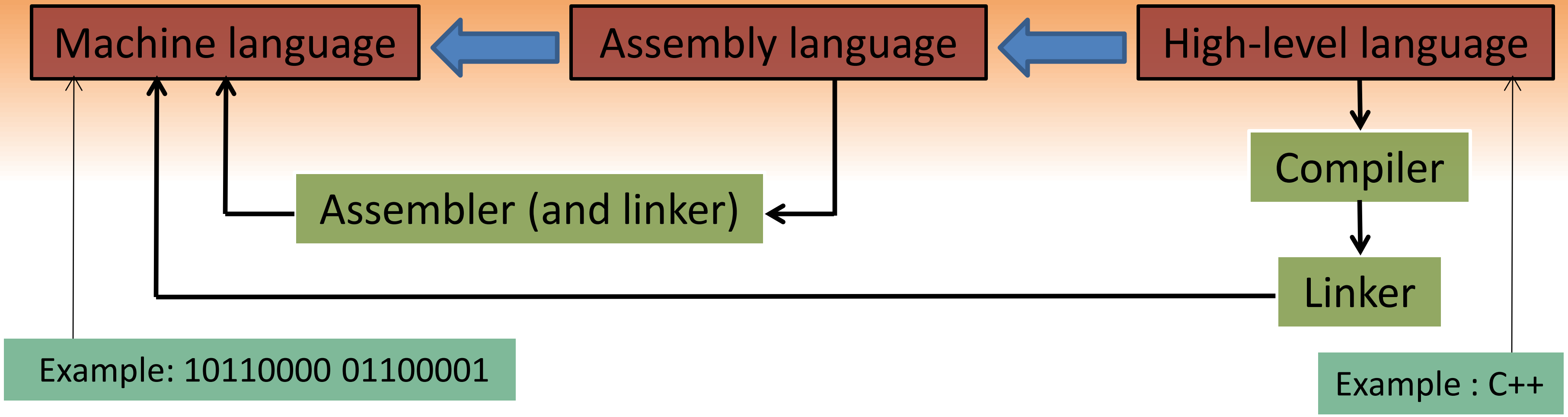


Programming the Microprocessor

- The most basic programming language of a microprocessor is machine language.

- Instructions are written as strings of bits.
  - Hard for humans to understand.
  - Different type of Processors may have different sets of binary codes.
  - Difficult to program in machine language.

- Assembly language was developed to make programming easier.
  - Instructions used versus string of bits.
  - In an easy to remember form called a mnemonic.



Machine language is the natural language of the microprocessor.

We can program the microprocessor using assembly language or a high-level language.

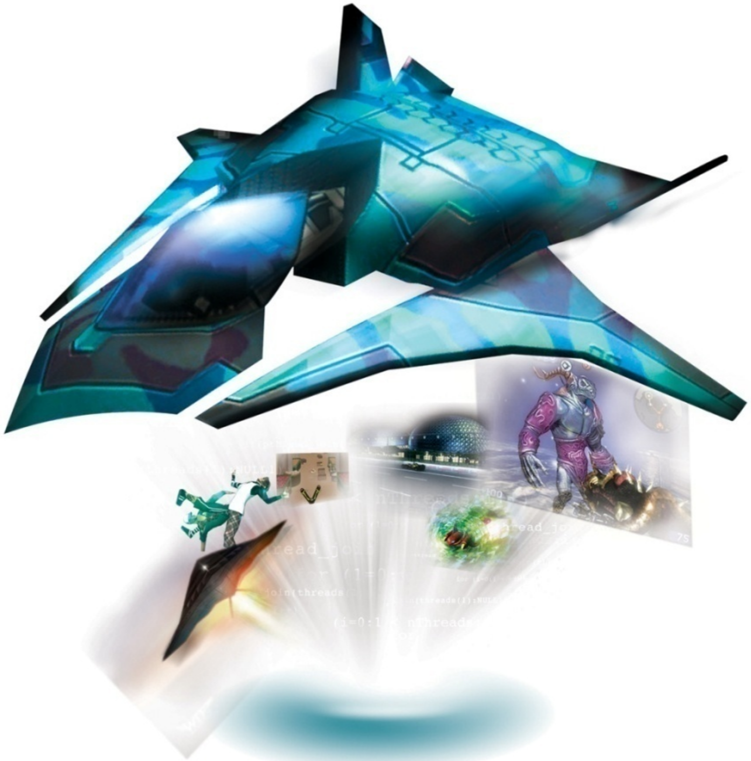
These languages need to be translated into machine language.

The assembler is a program that translates an assembly language program into machine language program that can be executed by the microprocessor.

The compiler is a program that translates a program written in high-level language (i.e. C++) into machine language.

Linker joins this machine language together with other required libraries, files and data into a machine language program that can be executed by the microprocessor.

INTRODUCTION TO ASSEMBLY LANAGUAGE



Introduction to Assembly Language

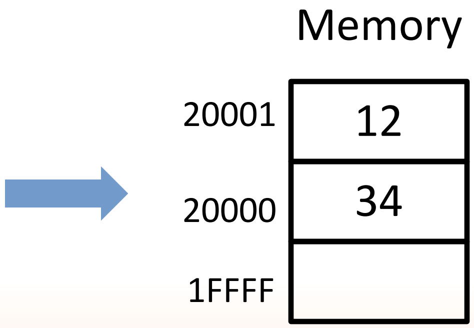
In assembly language, we program the microprocessor using individual instructions that move data into and out of the registers and memory locations.

- We study assembly language because it’s a good tool to help us understand how the microprocessor registers are used to program the microprocessor.

- One of the most common assembly language instructions is the MOV instruction. This is the data move instruction and it moves data to and from a location.

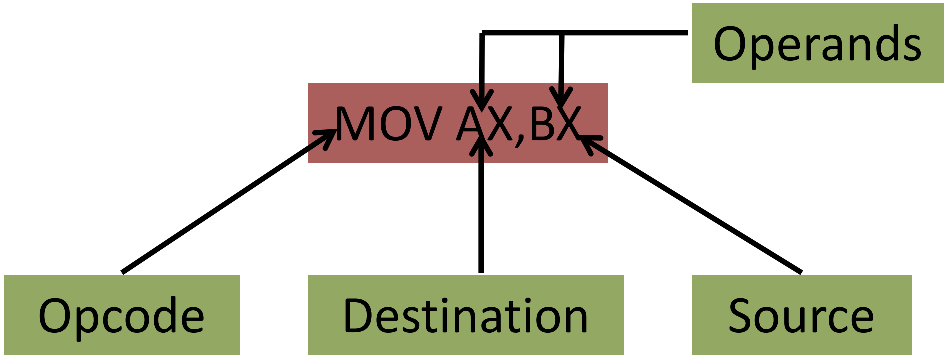
Whenever a word of data (i.e. 2 bytes) is placed into memory, the high order byte is always stored in the higher memory location.

The result of moving 1234H into memory address 20000H is shown on the right. **This is the case for 8-bit memory (i.e. each memory location can only hold 8 bits of data)**.



The MOV Instruction

The basic form of the MOV instruction specifies a source and destination. Here's an example:



Opcode is shortform for operation code. It tells the microprocessor which operation to perform. In this case MOV is the opcode. The "objects" needed in the instruction are called the operands.

The flow of data is always from right to left. So in this case the value in the BX register is moved (copied) into AX. The source does not change its value.

For all assembly language instructions the source never changes its value, except for the CMP and TEST instructions.

Data Addressing Modes

There are various ways that data can be accessed by the microprocessor. The following shows how the registers are used to access data in specific memory locations, and are called data addressing modes. We will use the MOV instruction to illustrate these modes.

- The DS register is used by default with any addressing mode that uses BX, DI, or SI to address memory.

Register:	MOV AX,BX
Immediate:	MOV CH,3AH
Direct:	MOV [1234H],AX
Register indirect:	MOV [BX],CL
Base-plus-index:	MOV [BX+SI],BP
Register relative:	MOV CL,[BX+4]
Base relative-plus-index:	MOV ARRAY[BX+SI],DX
Scaled index:	MOV [EBX+2 x ESI],AX

Immediate Addressing

Transfers the source, or a byte or word data into the destination register or memory location.

```
MOV  CH,3AH
```

Source is the data 3AH.

Destination is the register CH.

Direct Addressing

Moves a byte or word between a memory location and a register.

```
MOV  [1234H],AX
```

Source is register AX.

Destination is the memory location given by (DS x 10H + 1234H).

Register Indirect Addressing

Transfers a byte or word between a register and memory location addressed by an index or base register.

```
MOV  [BX],CL
```

Source is register CL.

Destination is the memory location given by (DS x 10H + BX).

Base-Plus-Index Addressing

Transfers a byte or word between a register and the memory location addressed by a base register plus an index register.

MOV [BX+SI],BP

Source is register BP.

Destination is the memory address given by (DS x 10H + BX + SI).

Register Relative Addressing

Moves a byte or word between a register and the memory location addressed by an index or base register plus a displacement.

MOV CL,[BX+4]

Source is the memory location given by (DS x 10H + BX + 4)

Destination is register CL.

Base Relative-Plus-Index Addressing

Transfers a byte or word between a register and the memory location addressed by a base and an index register plus displacement.

MOV ARRAY[BX+SI],DX

Source is register DX.

Destination is the memory location given by (DS x 10H + ARRAY + BX + SI).

Scaled-Index Addressing

The second register of a pair of registers is modified by the scale factor of 2x, 4x, or 8x to generate the memory address.

MOV [EBX+2 x ESI],AX

Source is register AX.

Destination is the memory location given by DS x 10H + EBX + 2 x ESI.

Remember that the calculations in the above examples of addressing modes assume the microprocessor is operating in **real mode**.

If the microprocessor is operating in **protected mode**, then the segment register is used to address a descriptor in a descriptor table that contains the base address of the segment.

This means that in protected mode, the value of (DS x 10H) isn’t used in the calculations for finding out the data segment’s starting address. Instead, the value of the base address that DS refers to in the descriptor table must be used.

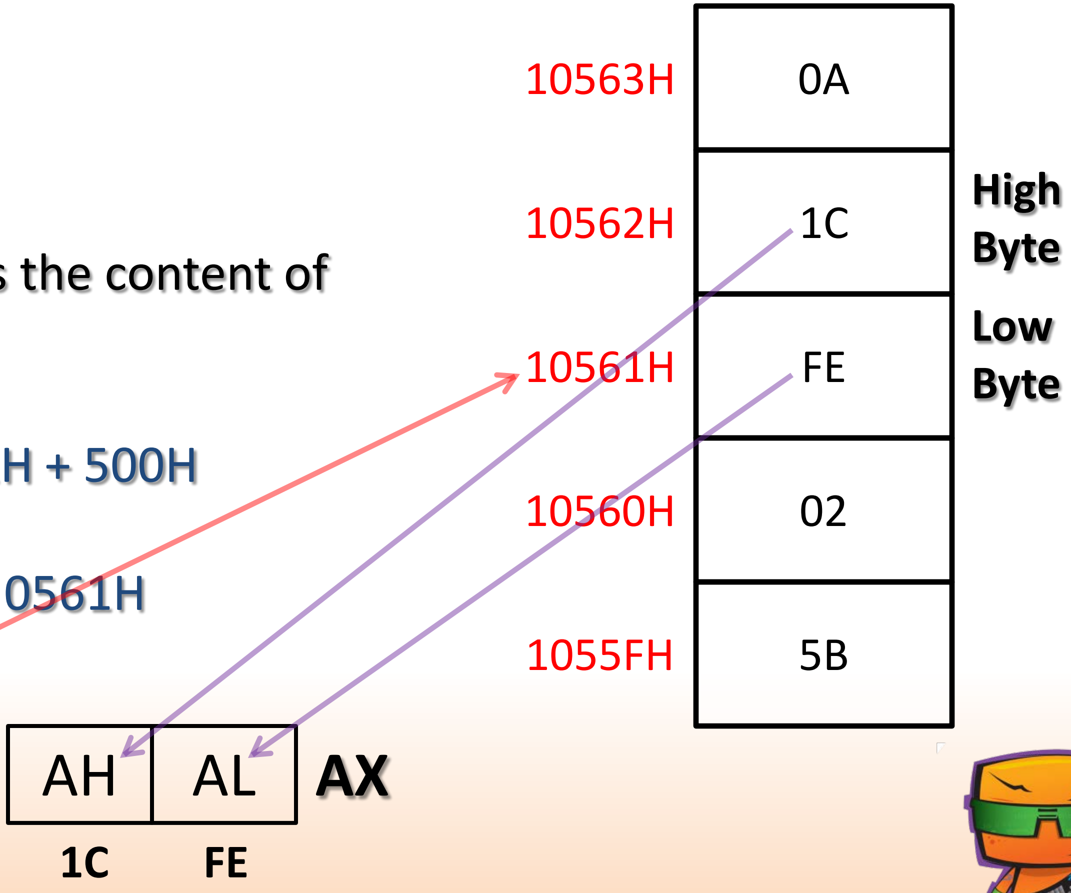
Example – Register Relative Addressing

Given the following register contents, and the memory layout shown opposite:
DS = 1000H
BX = 0050H
SI = 0011H

Given the instruction
MOV AX,[BX+SI+500H], what is the content of AX?

BX + SI + 500H = 0050H + 0011H + 500H
= 0561H
Memory Address = DS x 10H + 0561H
= 10561H

Answer: AX=1CFEH



10563H	0A
10562H	1C
10561H	FE
10560H	02
1055FH	5B

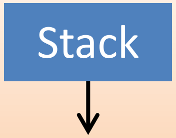
The Stack

- The stack is an area of memory that is used to store temporary data.
- It is a memory segment that follows the LIFO (Last-In-First-Out) algorithm (i.e. like a stack of paper – whatever you put in last must be taken out first).
- Data is added in to the stack using the PUSH instruction, and data is taken out using the POP instruction.
- The stack memory is maintained by the SP (stack pointer) and SS (stack segment) registers.

Remember that the stack segment’s starting address is (SS x 10H) only if the microprocessor is operating in real mode.

In protected mode, the value of (DS x 10H) isn’t used in the calculations for finding the stack segment’s starting address. Instead, the value of the base address that SS refers to in the descriptor table must be used.

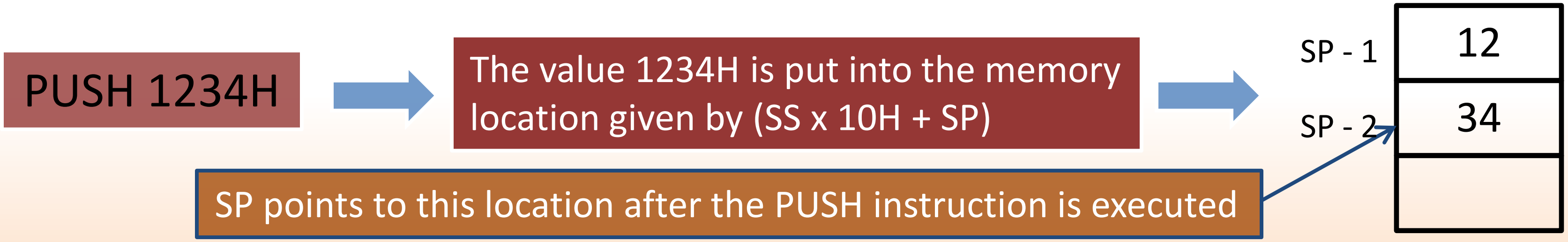
A stack grows downwards (i.e. each item added to the stack occupies a lower memory location).



Stack

The PUSH Instruction

- The PUSH instruction will place data onto the stack and cause the value in SP to decrement.
- Whenever a word of data (i.e. 16 bits) is pushed onto the stack, the higher 8 bits (high order byte) are placed in the memory location addressed by the value (SP – 1). The lower 8 bits (low order byte) are placed in the memory location addressed by the value (SP – 2).
- The value of SP is then decremented by 2, so that it points to the next available memory location on the stack.



PUSH 1234H

The value 1234H is put into the memory location given by (SS x 10H + SP)

SP - 1 | 12

SP - 2 | 34

SP points to this location after the PUSH instruction is executed

Note: We assume the stack memory is 8-bit.

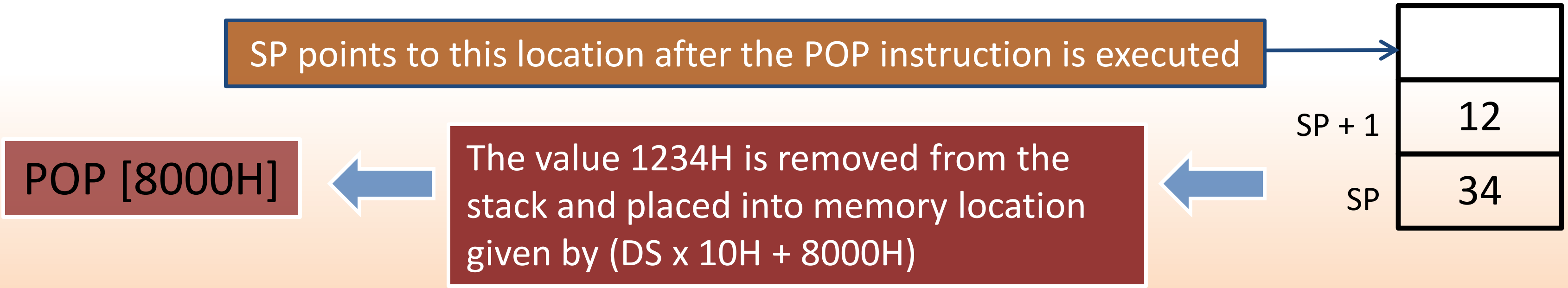
The POP Instruction

- The POP instruction will take data out of the stack and cause the value in SP to increment.

- Whenever a word of data (i.e. 16 bits) is popped from the stack, the lower 8 bits (low order byte) are removed from the memory location addressed by the value of SP. The higher 8 bits (high order byte) are removed from the memory location addressed by the value (SP + 1).

- The value of SP is then incremented by 2, so that it points to the next available data location on the stack.

E.g. If the bottom of the stack contains these values (we assume the stack memory is 8-bit):



SP points to this location after the POP instruction is executed

The value 1234H is removed from the stack and placed into memory location given by (DS x 10H + 8000H)

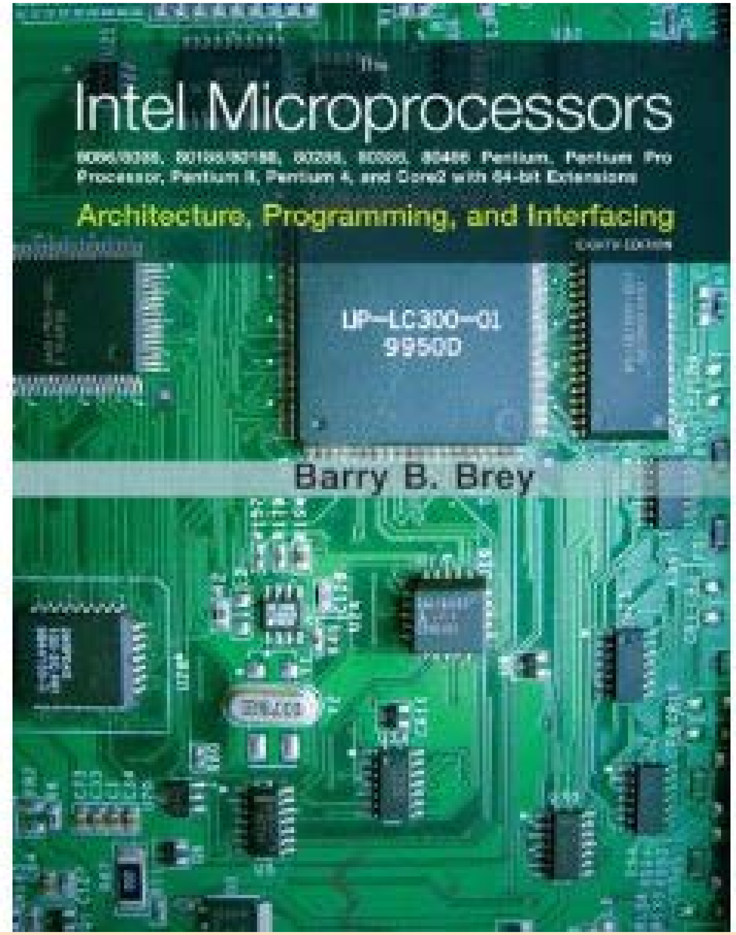
POP [8000H]

FYI: Some Other Assembly Language Instructions

- NOP : no operation mnemonic
  - Tells CPU not to change any states.
  - Commonly used for synchronisation/delay purposes.
- ADD : add operation
- SUB : subtraction operation
- MUL : multiply operation
- DIV : division operation

Acknowledgements

The information on these slides are based on the book The Intel Microprocessors, 8th Edition, by Barry B. Brey, Pearson Prentice Hall


The
Intel Microprocessors
8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, Pentium Pro Processor, Pentium II, Pentium 4, and Core2 with 64-bit Extensions
Architecture, Programming, and Interfacing
Eighth Edition
UP-LC300-01
9950D
Barry B. Brey