COSC 458-647
Application Software Security

Today

- Recall
  - Hexadeximal
  - Big v.s. Little Endian
- x86
  - Registers
  - Memory Addresses
  - Memory organization & addressing modes

Hexadecimal

Overview

- Hexadecimal (hex) ~ base 16 number system
- Use 0 through 9 and ...

A = 10

B = 11

C = 12

D = 13

E = 14

F = 15

Decimal Example

$$
\begin{aligned}
2657 &= 2000 + 600 + 50 + 7 \\
&= 2*1000 + 6*100 + 5*10 + 7*1 \\
&= 2*10^3 + 6*10^2 + 5*10^1 + 7*10^0
\end{aligned}
$$

Binary Example

$1011_2 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

$= 1*8 + 0*4 + 1*2 + 1*1$

$= 8 + 2 + 1 = 11_{10}$

Hexadecimal Example

$$A4F_{16} = 10*16^2 + 4*16^1 + 15*16^0$$
$$= 10*256 + 4*16 + 15*1$$
$$= 2560 + 64 + 15 = 2639_{10}$$

Hexadecimal → Decimal

$61_{16} = ?$

$F23_{16} = ?$

Now convert the above to binary...

So why do we use hex?

- Binary is annoying to read
- Hexadecimal is slightly easier
- Binary ←→ Hexadecimal is painless
- Example: $1110\ 1010\ 1001\ 0101_2$ = ?

Binary → Hexadecimal

1. Split the binary number up into 4-bit sections

2. Determine the hexadecimal value of each section

3. …you're done

Example: 1110100101110101010000101

Hexadecimal → Binary

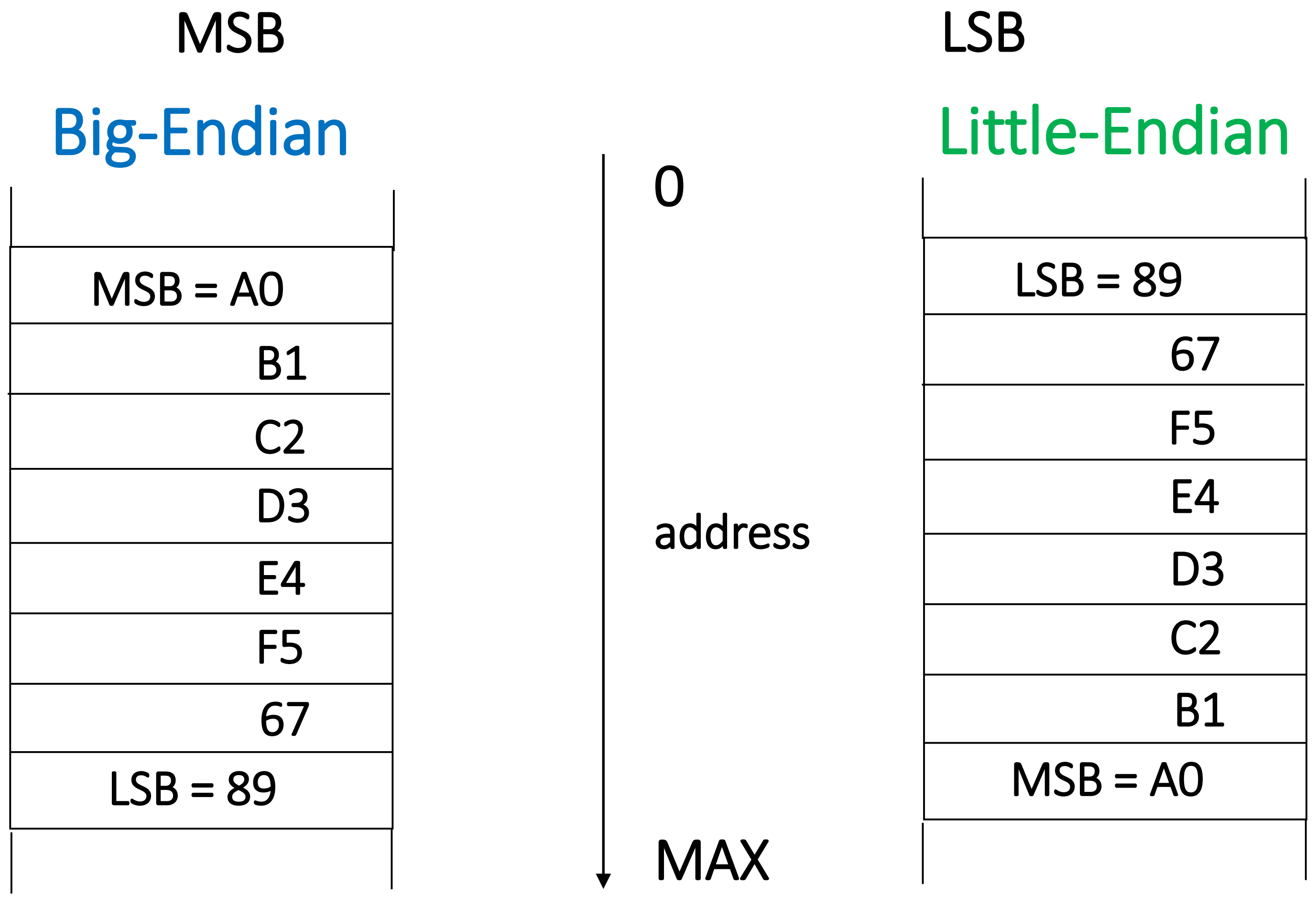
1. Determine the 4-bit binary value for each hexadecimal digit

2. … you’re done

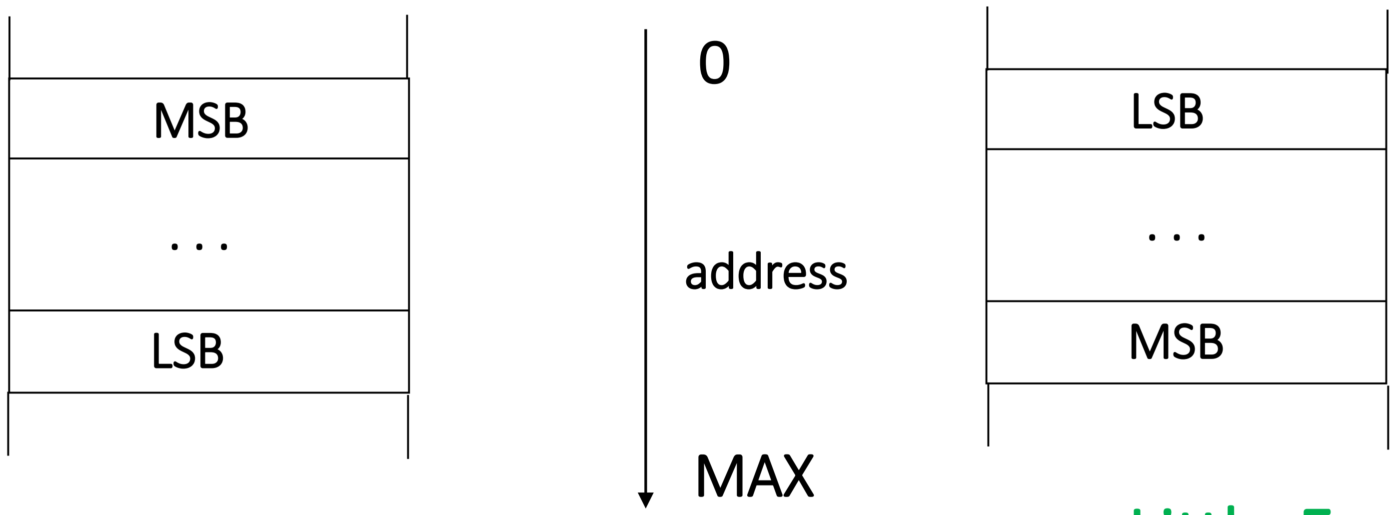
Little-Endian vs. Big-Endian Representation of Integers

Little-Endian vs. Big-Endian Representation

A0 B1 C2 D3 E4 F5 67 89_{16}



Little-Endian vs. Big-Endian Camps



Big-Endian

- Motorola 68xx, 680x0
- IBM
- Hewlett-Packard
- Sun SuperSPARC
- Internet TCP/IP

Bi-Endian

- Motorola Power PC
- Silicon Graphics MIPS

Little-Endian

- Intel
- AMD
- DEC VAX
- RS 232

Little-Endian vs. Big-Endian

Advantages and Disadvantages

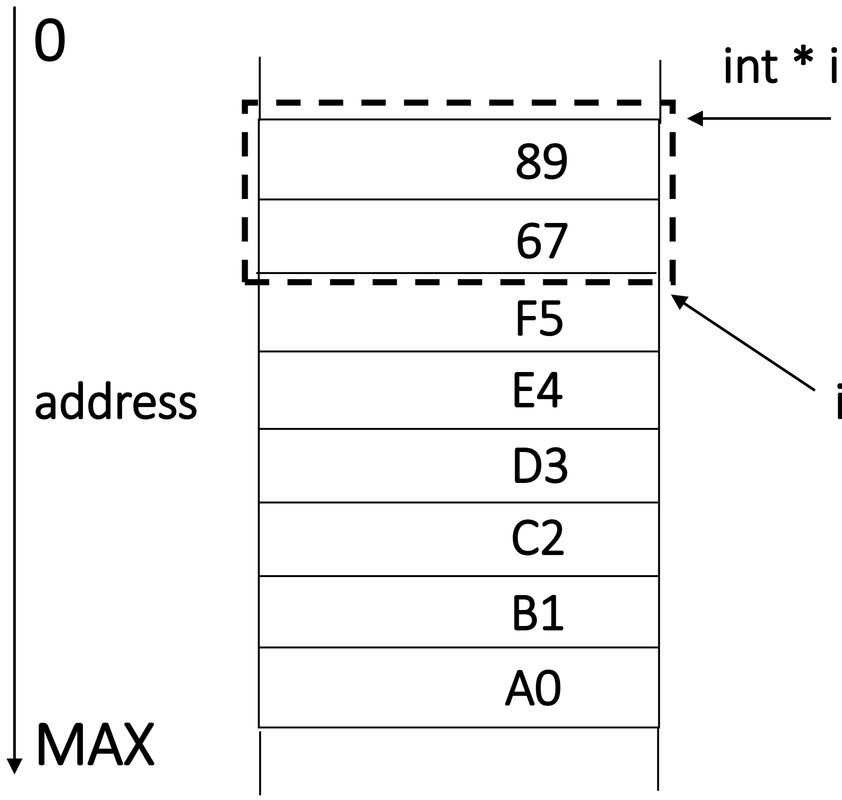
Big-Endian

- easier to determine a sign of the number
- easier to compare two numbers
- easier to divide two numbers
- easier to print

Little-Endian

- easier addition and multiplication of multiprecision numbers

Pointers (1)

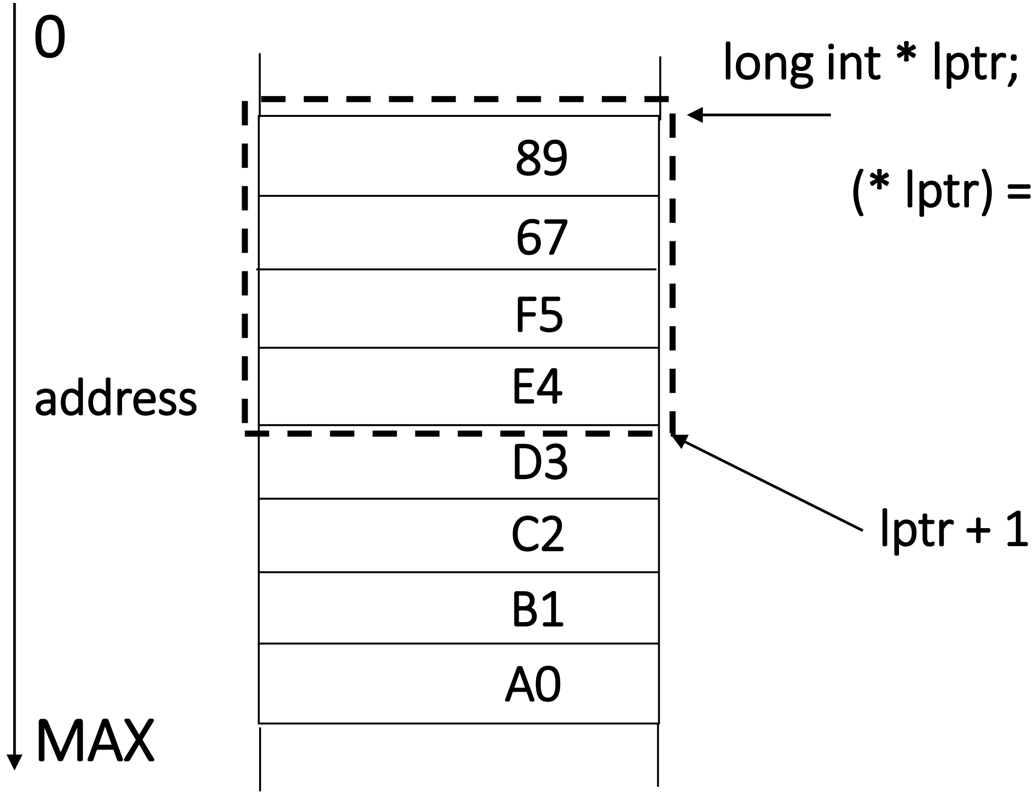


int * iptr;

iptr+1

Big-Endian	Little-Endian
(* iptr) = 8967;	(* iptr) = 6789;

Pointers (2)



Big-Endian	Little-Endian
(* lptr) = 8967F5E4;	(* lptr) = E4F56789;

Bonus question

- Write a C/C++ program to check whether your system is using Big or Little Endian representation

2’s Complement Arithmetic

(Adopted from projectLeadTheWay)

2’s Complement Arithmetic

This presentation will demonstrate

- That subtracting one number from another is the same as making one number negative and adding.
- How to create negative numbers in the binary number system.
- The 2’s Complement Process.
- How the 2’s complement process can be use to add (and subtract) binary numbers.

Negative Numbers?

- Digital electronics requires frequent addition and subtraction of numbers. You know how to design an adder, but what about a subtract-er?
- A subtract-er is not needed with the 2’s complement process. The 2’s complement process allows you to easily convert a positive number into its negative equivalent.
- Since subtracting one number from another is the same as making one number negative and adding, the need for a subtract-er circuit has been eliminated.

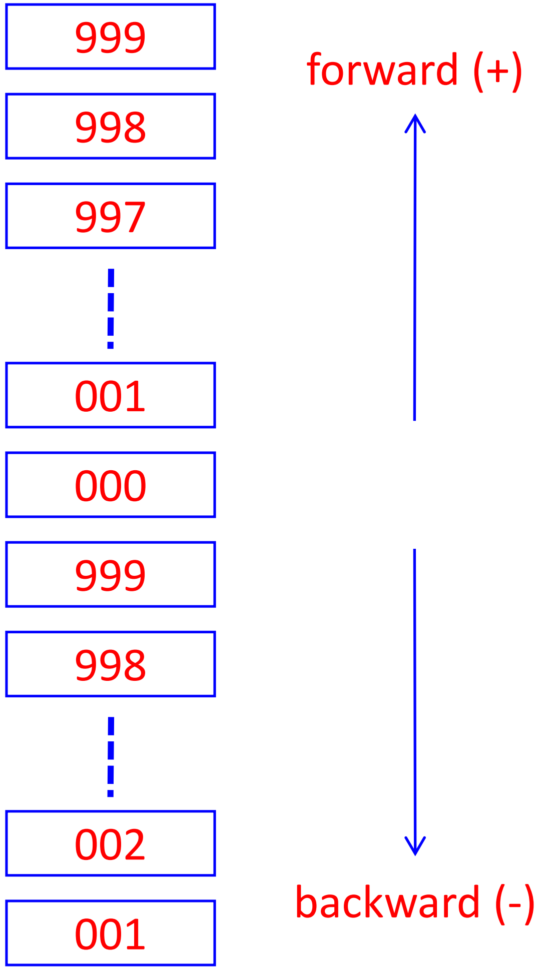
How To Create A Negative Number

- In digital electronics you cannot simply put a minus sign in front of a number to make it negative.
- You must represent a negative number in a *fixed-length* binary number system. All signed arithmetic must be performed in a *fixed-length* number system.
- A physical *fixed-length* device (usually memory) contains a fixed number of bits (usually 4-bits, 8-bits, 16-bits) to hold the number.

3-Digit Decimal Number System

A bicycle odometer with only three digits is an example of a fixed-length decimal number system.

The problem is that without a negative sign, you cannot tell a +998 from a -2 (also a 998). Did you ride forward for 998 miles or backward for 2 miles?



Negative Decimal

How do we represent negative numbers in this 3-digit decimal number system without using a sign?

→Cut the number system in half.

→Use 001 – 499 to indicate positive numbers.

→Use 500 – 999 to indicate negative numbers.

→Notice that 000 is not positive or negative.

+499	499
+498	498
+497	497
⋮	⋮
+001	001
000	000
-001	999
-002	998
⋮	⋮
-499	501
-500	500

pos(+)

neg(-)

"Odometer" Math Examples

$$
\begin{array}{r}
3 \\
+\ \ 2 \\
\hline
5
\end{array}
\qquad
\begin{array}{r}
003 \\
+\ \ 002 \\
\hline
005
\end{array}
\qquad\qquad
\begin{array}{r}
6 \\
+\ (-3) \\
\hline
3
\end{array}
\qquad
\begin{array}{r}
006 \\
+\ \ 997 \\
\hline
1]003
\end{array}
$$

Disregard Overflow

$$
\begin{array}{r}
(-5) \\
+\ \ 2 \\
\hline
(-3)
\end{array}
\qquad
\begin{array}{r}
995 \\
+\ \ 002 \\
\hline
997
\end{array}
\qquad\qquad
\begin{array}{r}
(-2) \\
+\ (-3) \\
\hline
(-5)
\end{array}
\qquad
\begin{array}{r}
998 \\
+\ \ 997 \\
\hline
1]995
\end{array}
$$

Disregard Overflow

It Works!

Complex Problems

- The previous examples demonstrate that this process works, but how do we *easily* convert a number into its negative equivalent?
- In the examples, converting the negative numbers into the 3-digit decimal number system was fairly easy. To convert the (-3), you simply counted backward from 1000 (i.e., 999, 998, 997).
- This process is not as easy for large numbers (e.g., -214 is 786). How did we determine this?
- To convert a large negative number, you can use the 10's Complement Process.

10’s Complement Process

The **10’s Complement** process uses base-10 (decimal) numbers. Later, when we’re working with base-2 (binary) numbers, you will see that the **2’s Complement** process works in the same way.

First, complement all of the digits in a number.

- A digit’s complement is the number you add to the digit to make it equal to the largest digit in the base (i.e., 9 for decimal). The complement of 0 is 9, 1 is 8, 2 is 7, etc.

Second, add 1.

- Without this step, our number system would have two zeroes (+0 & -0), which no number system has.

10's Complement Examples

Example #1

$$\underbrace{\begin{array}{r} -003 \\ \downarrow\downarrow\downarrow \\ 996 \end{array}}_{\text{Complement Digits}}$$

$$\underbrace{\begin{array}{r} 996 \\ +1 \\ \hline 997 \end{array}}_{\text{Add 1}}$$

Example #2

$$\underbrace{\begin{array}{r} -214 \\ \downarrow\downarrow\downarrow \\ 785 \end{array}}_{\text{Complement Digits}}$$

$$\underbrace{\begin{array}{r} 785 \\ +1 \\ \hline 786 \end{array}}_{\text{Add 1}}$$

8-Bit Binary Number System

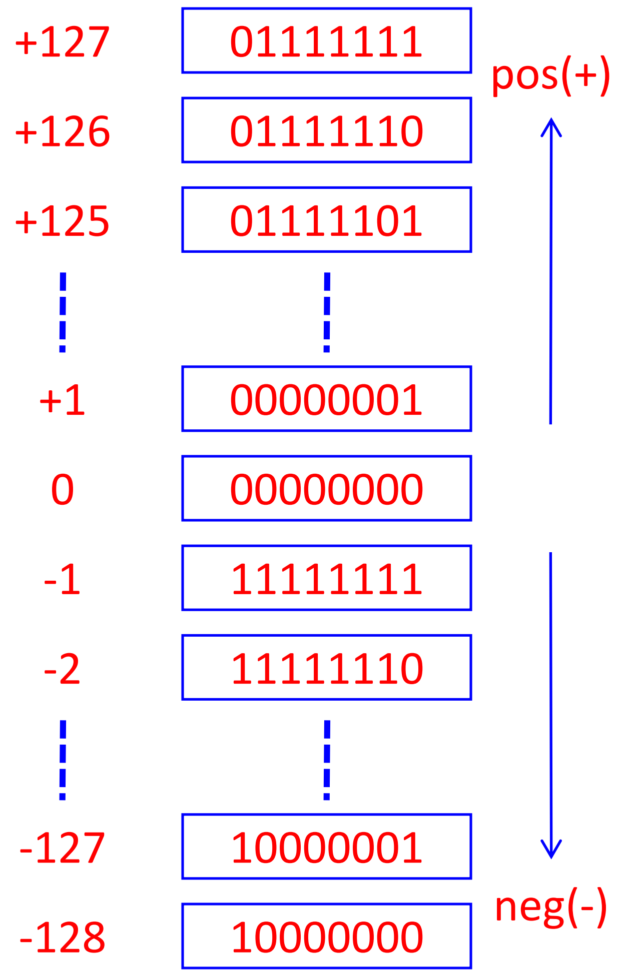
Apply what you have learned to the binary number systems. How do you represent negative numbers in this 8-bit binary system?

→Cut the number system in half.

→Use 00000001 – 01111111 to indicate positive numbers.

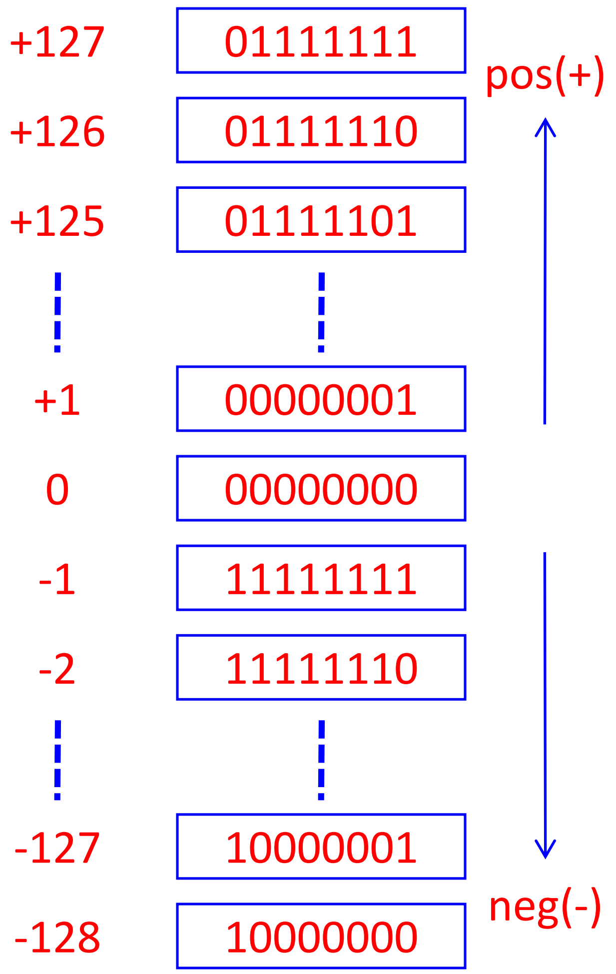
→Use 10000000 – 11111111 to indicate negative numbers.

→Notice that 00000000 is not positive or negative.



Sign Bit

- What did do you notice about the most significant bit of the binary numbers?
- The MSB is (0) for all positive numbers.
- The MSB is (1) for all negative numbers.
- The MSB is called the sign bit.
- In a signed number system, this allows you to instantly determine whether a number is positive or negative.



2'S Complement Process

The steps in the **2's Complement** process are similar to the 10's Complement process. However, you will now use the base two.

First, complement all of the digits in a number.

- A digit's complement is the number you add to the digit to make it equal to the largest digit in the base (i.e., 1 for binary). In binary language, the complement of 0 is 1, and the complement of 1 is 0.

Second, add 1.

- Without this step, our number system would have two zeroes (+0 & -0), which no number system has.

2’s Complement Examples

Example #1

```
 5 = 00000101
     ↓↓↓↓↓↓↓↓
     11111010
           +1
     --------
-5 = 11111011
```

$\left.\begin{array}{l} 5 = 00000101 \\ \downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow \\ 11111010 \end{array}\right\}$ Complement Digits

$\left.\begin{array}{l} 11111010 \\ +1 \\ -5 = 11111011 \end{array}\right\}$ Add 1

Example #2

```
-13 = 11110011
      ↓↓↓↓↓↓↓↓
      00001100
            +1
      --------
 13 = 00001101
```

$\left.\begin{array}{l} -13 = 11110011 \\ \downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow \\ 00001100 \end{array}\right\}$ Complement Digits

$\left.\begin{array}{l} 00001100 \\ +1 \\ 13 = 00001101 \end{array}\right\}$ Add 1

Using The 2's Compliment Process

Use the 2's complement process to add together the following numbers.

$$\begin{array}{r} \text{POS} \\ +\ \ \text{POS} \\ \hline \text{POS} \end{array} \Rightarrow \begin{array}{r} 9 \\ +\ \ 5 \\ \hline 14 \end{array} \qquad \begin{array}{r} \text{NEG} \\ +\ \ \text{POS} \\ \hline \text{NEG} \end{array} \Rightarrow \begin{array}{r} (-9) \\ +\ \ 5 \\ \hline -\ 4 \end{array}$$

$$\begin{array}{r} \text{POS} \\ +\ \ \text{NEG} \\ \hline \text{POS} \end{array} \Rightarrow \begin{array}{r} 9 \\ +\ (-5) \\ \hline 4 \end{array} \qquad \begin{array}{r} \text{NEG} \\ +\ \ \text{NEG} \\ \hline \text{NEG} \end{array} \Rightarrow \begin{array}{r} (-9) \\ +\ (-5) \\ \hline -\ 14 \end{array}$$

POS + POS → POS Answer

If no 2's complement is needed, use regular binary addition.

$$\begin{array}{r} 9 \\ +\quad 5 \\ \hline 14 \end{array} \quad \begin{array}{c} \longrightarrow \\ \longrightarrow \\ \\ \longleftarrow \end{array} \quad \begin{array}{r} 00001001 \\ +\ 00000101 \\ \hline 00001110 \end{array}$$

POS + NEG → POS Answer

Take the 2’s complement of the negative number and use regular binary addition.

$$\begin{array}{r} 9 \\ +\ (-5) \\ \hline 4 \end{array} \qquad \begin{array}{r} 00001001 \\ +\ 11111011 \\ \hline \cancel{1}]\,00000100 \end{array}$$

8th Bit = 0: Answer is Positive
Disregard 9th Bit

$$\left.\begin{array}{r} 00000101 \\ \downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow \\ 11111010 \\ +1 \\ \hline 11111011 \end{array}\right\}$$ 2’s Complement Process

POS + NEG → NEG Answer

Take the 2’s complement of the negative number and use regular binary addition.

```
  (-9)          11110111
+   5         + 00000101
------        ----------
   -4           11111100
```

$-\boxed{4}$

8th Bit = 1: Answer is Negative

To Check: Perform 2’s Complement On Answer

```
 11111100
 ↓↓↓↓↓↓↓↓
 00000011
       +1
---------
 00000100
```

2’s Complement Process

```
 00001001
 ↓↓↓↓↓↓↓↓
 11110110
       +1
---------
 11110111
```

NEG + NEG → NEG Answer

Take the 2's complement of both negative numbers and use regular binary addition.

$$\begin{array}{r} (-9) \\ +\ \ (-5) \\ \hline -\boxed{14} \end{array} \quad \begin{array}{c} \longrightarrow \\ \longrightarrow \\ \longleftarrow \end{array} \quad \begin{array}{r} 11110111 \\ +\ 11111011 \\ \hline \cancel{1}]\,11110010 \end{array}$$

2's Complement Numbers, See Conversion Process In Previous Slides

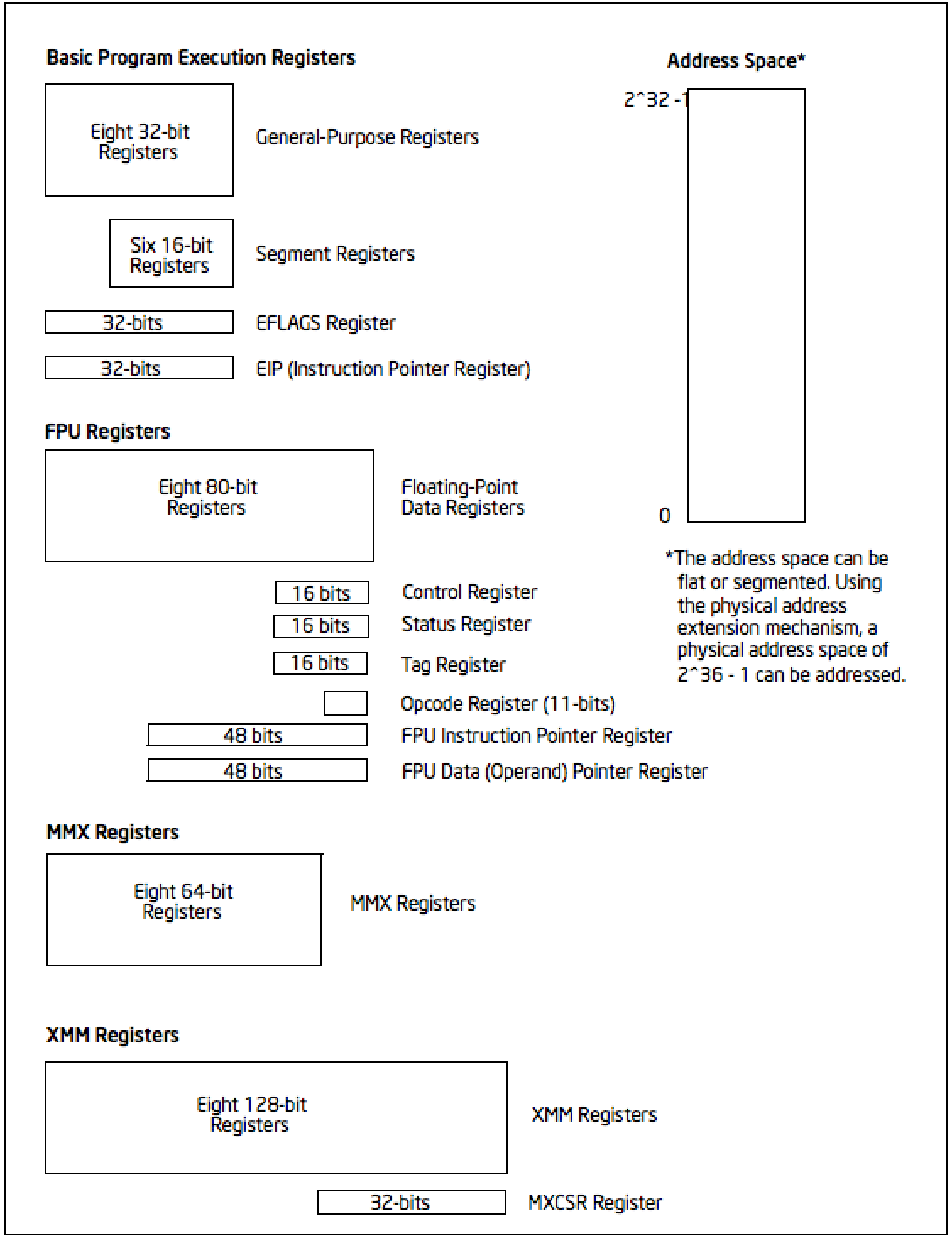
8^{th} Bit = 1: Answer is Negative
Disregard 9^{th} Bit

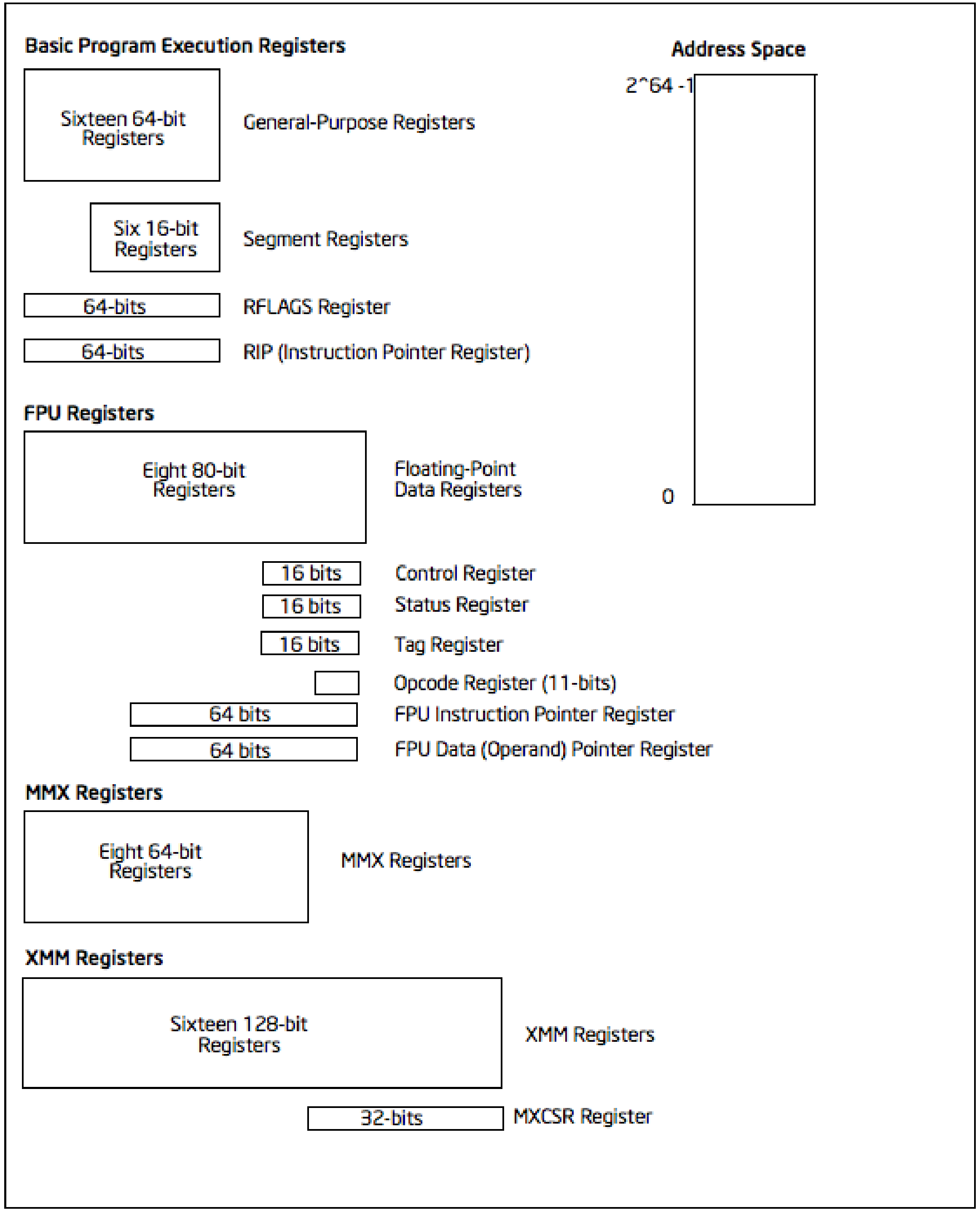
To Check: Perform 2's Complement On Answer

$$\begin{array}{r} 11110010 \\ \downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow \\ 00001101 \\ +1 \\ \hline 00001110 \end{array}$$

x86 basis

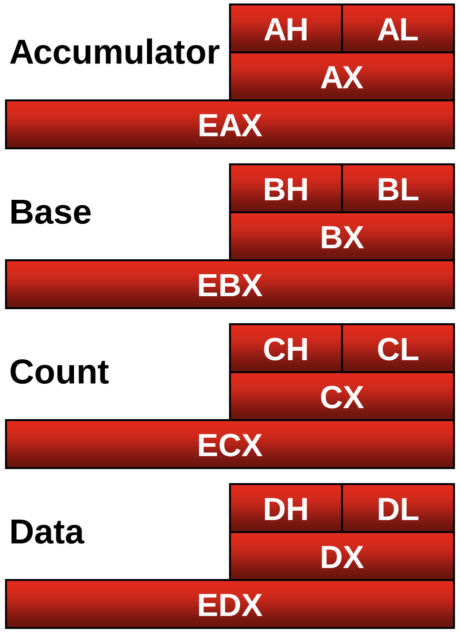
Basic Execution Environment



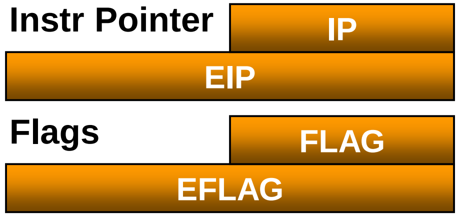


x86 Registers at a Glance

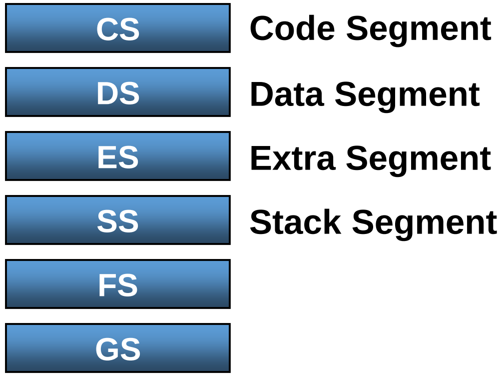
General Purpose



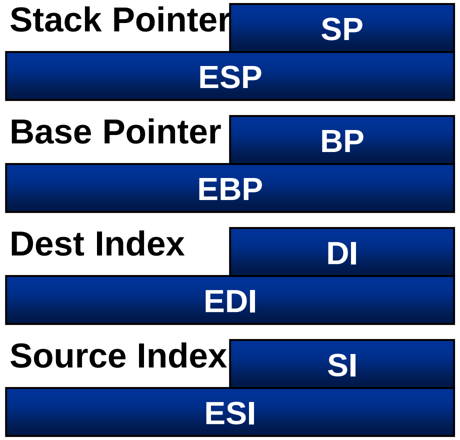
Special Registers



Segment Registers



Index Registers

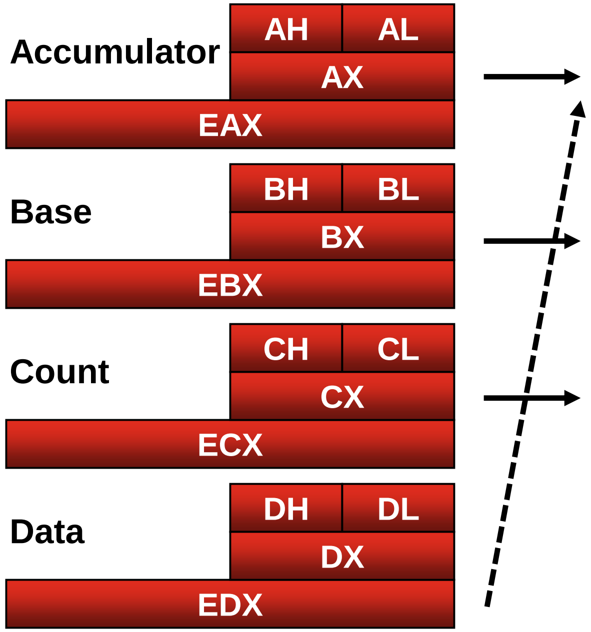


General-Purpose Registers

31–16	15–8	7–0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

x86 Registers at a Glance

General Purpose



Used for I/O port access, arithmetic, interrupt calls, etc...

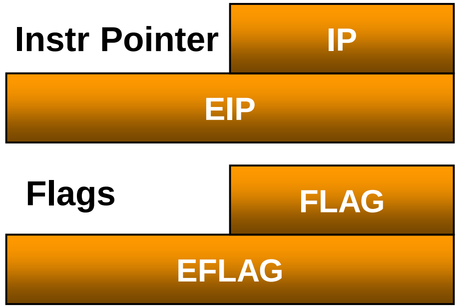
Used as a base pointer for memory access. Gets some interrupt return values

Used as a loop counter and for shifts. Gets some interrupt values

Address	Name	Description
EAX*	Accumulator Register	calculations for operations and results data
EBX	Base Register	pointer to data in the DS segment
ECX*	Count Register	counter for string and loop operations
EDX*	Data Register	input/output pointer

x86 Registers at a Glance

Special Registers



→ Holds the offset of the next instruction. **<u>It can only be read</u>**

→ The EFLAGS register hold the **<u>state</u>** of the processor. It is modified by many intructions and is used for comparing some parameters, conditional loops and conditionnal jumps.

x86 Registers at a Glance

Special Registers

Segment Registers

CS **Code Segment**

DS **Data Segment**

ES **Extra Segment**

SS **Stack Segment**

FS

GS

Holds the Code segment in which your program runs. Changing its value might make the computer hang.

Holds the Data segment that your program accesses. Changing its value might give erronous data.

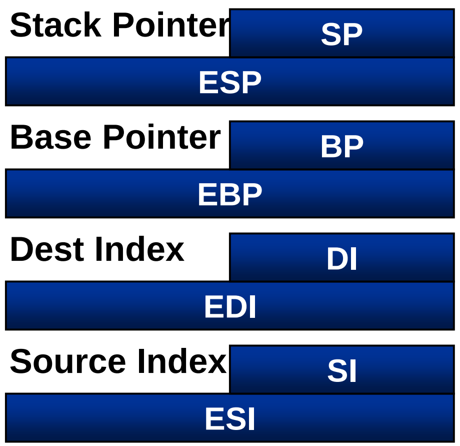
These are extra segment registers available for far pointer addressing like video memory and such.

Holds the Stack segment your program uses. Sometimes has the same value as DS. Changing its value can give unpredictable results, mostly data related.

Address	Name	Description
CS	Code Segment	where instructions being executed are stored
DS, ES, FS, GS	Data Segment	data segment
SS	Stack Segment	where the stack for the current program is stored

x86 Registers at a Glance

Index Registers



Holds the top address of the stack. Should not be used of not backed up

Holds the base address of the stack

Used for string, memory array copying and setting and for far pointer addressing with ES

Used for string and memory array copying

ESI	Source Index	source pointer for string operations
EDI	Destination Index	destination pointer for string operations
ESP	Stack Pointer	stack pointer, **should not be used**
EBP	Base Pointer	pointer to data on the stack

x86 Registers (cont'd)

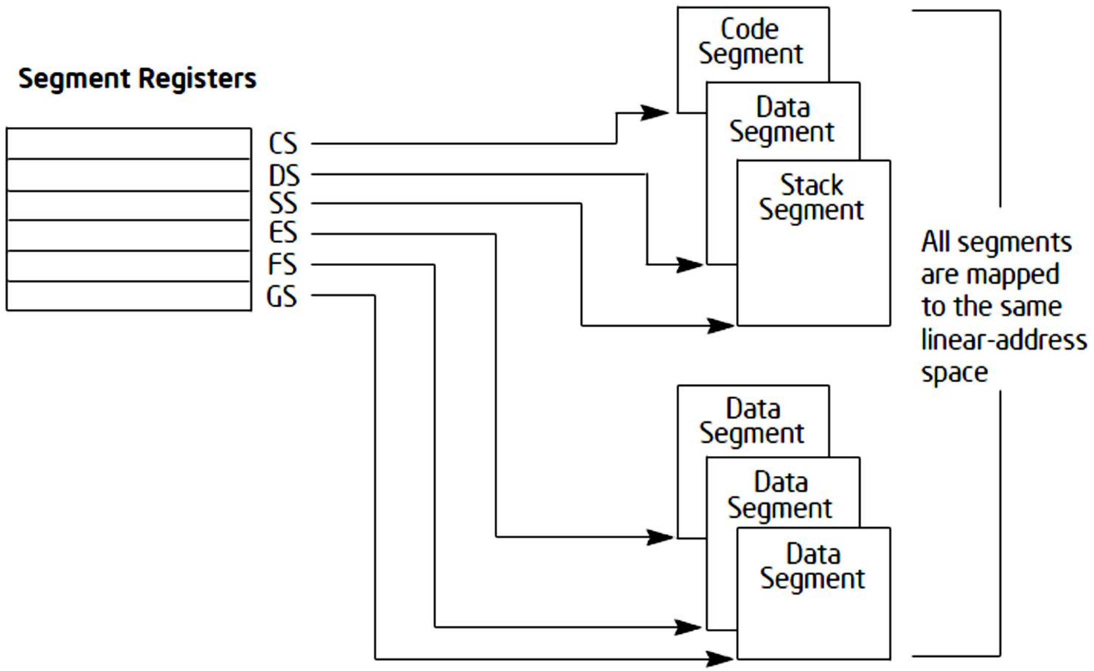
Address	Name	Description
EAX*	Accumulator Register	calculations for operations and results data
EBX	Base Register	pointer to data in the DS segment
ECX*	Count Register	counter for string and loop operations
EDX*	Data Register	input/output pointer
ESI	Source Index	source pointer for string operations
EDI	Destination Index	destination pointer for string operations
ESP	Stack Pointer	stack pointer, **should not be used**
EBP	Base Pointer	pointer to data on the stack

Address	Name	Description
CS	Code Segment	where instructions being executed are stored
DS, ES, FS, GS	Data Segment	data segment
SS	Stack Segment	where the stack for the current program is stored

Segment Registers

Six 16-bit Registers | Segment Registers

- 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.



- **CS : Instructions**
  All instruction fetches
- **SS : Stack**
  All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.
- **DS : Local Data**
  All data references, except when relative to stack or string destination.
- **ES : Destination Strings**
  Destination of string instructions, eg. MOVS.

The EFLAGS Register

- Contains the status and control flags that are set by the instructions executed. These indicate the result of the instruction execution.

Flag	**Intel Mnemonic**	**Notes**
Overflow	OF	
Direction	DF	Indicates the direction of string processing. 1 means highest address to lowest. 0 means lowest address to highest address
Interrupt Enable	IF	Set to 1 if interrupts are enabled. This is always set to 1 by a user mode debugger
Sign	SF	
Zero	ZF	
Auxiliary Carry	AF	Indicates a carry/borrow in BCD arithmetic
Parity	PF	
Carry	CF	

The EFLAGS Register

Bit	Label	Description
0	CF	Carry flag
2	PF	Parity flag
4	AF	Auxiliary carry flag
6	ZF	Zero flag
7	SF	Sign flag
8	TF	Trap flag
9	IF	Interrupt enable flag
10	DF	Direction flag
11	OF	Overflow flag
12-13	IOPL	I/O Privilege level
14	NT	Nested task flag
16	RF	Resume flag
17	VM	Virtual 8086 mode flag
18	AC	Alignment check flag (486+)
19	VIF	Virtual interrupt flag
20	VIP	Virtual interrupt pending flag
21	ID	ID flag

Those that are not listed are reserved by Intel.

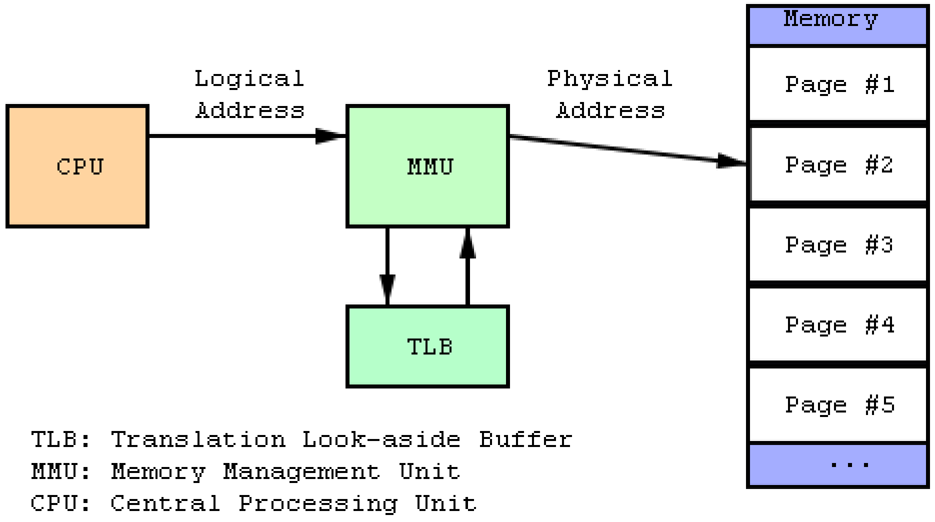
A note on GP registers

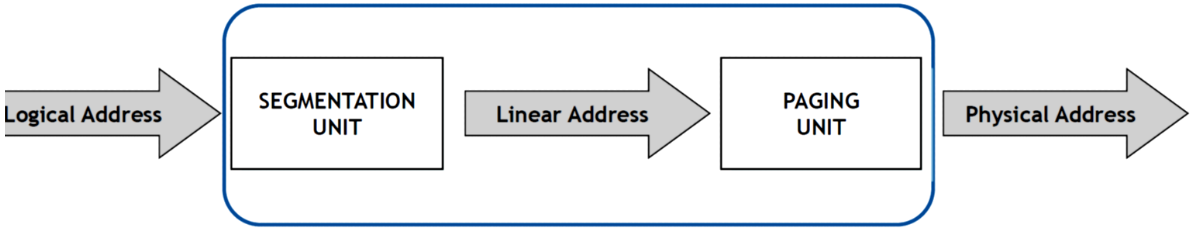
- GP registers can be used with a great deal of flexibility (in 80386 and newer processors)
- Each GP register is meant to be used for specific purposes…
- Memorizing the names of the registers will help understand how to use them
- Learning how to manage your registers will help you develop good programming practices
- You will find that you are generally short of registers

Memory Addresses

- Logical Address
  - Included in the machine language instruction
  - The address of an operand or of an instruction
  - Consists of segment (16bit) and offset (32bit)
    - offset - distance from the start of the segment to the actual address
- Linear Address (known as virtual address)
  - A single 32-bit unsigned integer
  - Range: 0x00000000 ~ 0xffffffff(4GB)
- Physical Address
  - Used to address memory cells included in memory chips
  - Represented as 32-bit unsigned integer

Memory Addresses





MMU(Memory Management Unit)

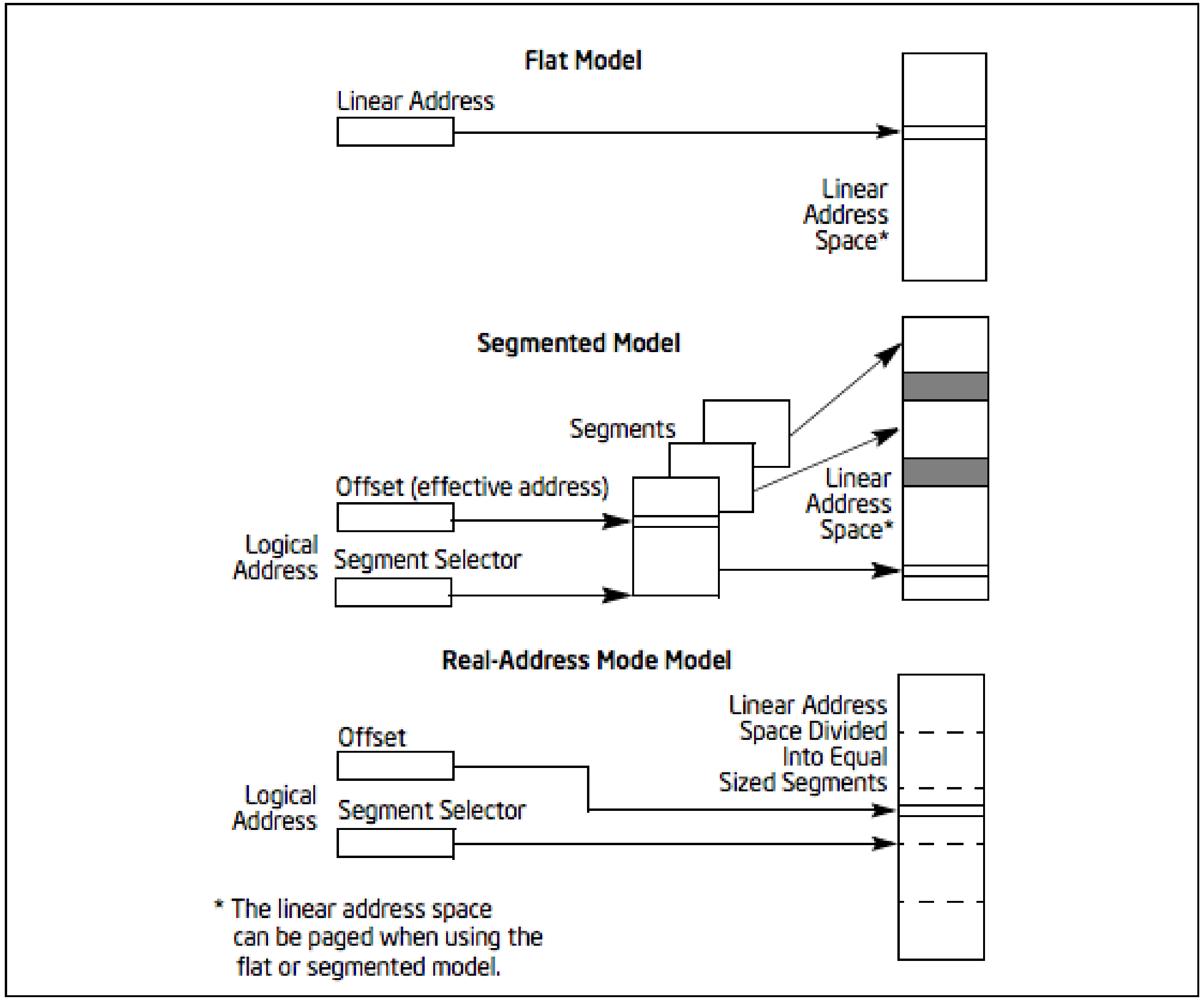
+ Most MMUs use an in-memory table of items called a "page table," containing one "page table entry" (PTE) per page, to map virtual page numbers to physical page numbers in main memory

+ An associative cache of PTEs is called a translation lookaside buffer (TLB) and is used to avoid the necessity of accessing the main memory every time a virtual address is mapped.

+ Modern MMUs typically divide the virtual address space(the range of addresses used by the processor) into pages

Each having a size which is a power of 2 (few KBs → much larger)

Memory models



- No segmentation
- Code, Data, stacks are all contained in this address space.
- 32 bit addressing

- Code, Data, stacks are typically contained in separate segments for better isolation.
- 32 bit addressing (32 bit offset, 16 bit seg. selector)

- Compatibility mode for 8086 processor.
- 20 bit addressing (16 bit offset, 16 seg. selector)

x86 memory addressing modes

- Width of the address bus determines the amount of addressable memory

- The amount of addressable memory is NOT the amount of physical memory available in your system

- Real mode addressing

- Protected mode addressing

Real mode memory addressing

- This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode).
- The processor is placed in real-address mode following power-up or a reset.
- A throwback to the age of 8086, 20-bit address bus, 16-bit data bus
- In real mode we can only address memory locations 0 through 0xFFFFF. Used only with 16-bit registers
- We will not be using real mode!

Real mode memory addressing (cont’d)

- Memory address format **Segment** : **Offset**

- Linear address obtained by:
  - Shifting segment left by 4 bits
  - Adding offset

- Example: 2222:3333 → Linear address: 25553
- Example: 2000:5553 → Linear address: 25553

- THIS WILL NOT APPLY TO US IN 32-bit PROTECTED MODE!

Protected mode memory addressing

- This mode is the native state of the processor
- Support virtual-8086 mode to execute “real-address mode” 8086 software in a protected, multi-tasking environment.
- Segmentation, 32bit addressing:
  - 32-bit address bus, 32-bit data bus, 32-bit registers
- Up to 4 Gigabytes of addressable memory
- 80386 and higher operate in either real or protected mode

Next class

- Memory segmentation
- Assembly language
    - Bring your laptop

Checking sizes of compiled program’s segments

check-endianness.c

```
#include <stdio.h>

int main () {
  unsigned int x = 0x76543210;
  char *c = (char*) &x;

  if (*c == 0x10)   {
    printf ("Underlying architecture is little endian. \n");
  } else {
     printf ("Underlying architecture is big endian. \n");
  }

  return 0;
}
```

Checking sizes of compiled program's segments

- `gcc check-endianness.c -o check-endianness`
- `size check-endianness`