



## Chapter 4


# THE CENTRAL PROCESSING UNIT



## Language levels

- High-Level Languages
- Characteristics
  - Portable : to varying degrees
  - Complex:one statement can do much work
  - Expressive
  - Human readable


```
count = 0;
while (n>1)
{  count++;
   if (n&1)
       n = n*3+1;
   else
       n = n/2;
}
```



## Machine languages

- Characteristics
  - Not portable : specific to hardware
  - Simple : each instruction does a simple task
  - Not expressive : each instruction performs little work
  - Not human readable : requires lots of effort, requires tool support

0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
9222 9120 1121 A120 1121 A121 7211 0000
0000 0001 0002 0003 0004 0005 0006 0007
0008 0009 000A 000B 000C 000D 000E 000F
0000 0000 0000 FE10 FACE CAFE ACED CEDE
1234 5678 9ABC DEF0 0000 0000 F00D 0000
0000 0000 EEEE 1111 EEEE 1111 0000 0000
B1B2 F1F5 0000 0000 0000 0000 0000 0000



## Assembly languages

- Characteristics
  - Not portable : each assembly language instruction map to one machine language instruction
  - Simple : each instruction does a simple task
  - Not expressive
  - Human readable

```


movl    $0, %r10d
loop:   cmpl    $1, %r11d
        jle     endloop

        addl    $1, %r10d
        movl    %r11d, %eax
        andl    $1, %eax
        je      else

        movl    %r11d, %eax
        addl    %eax, %r11d
        addl    %eax, %r11d
        addl    $1, %r11d


        jmp     endif
else:   sarl    $1, %r11d
endif:  jmp     loop
endloop:

```



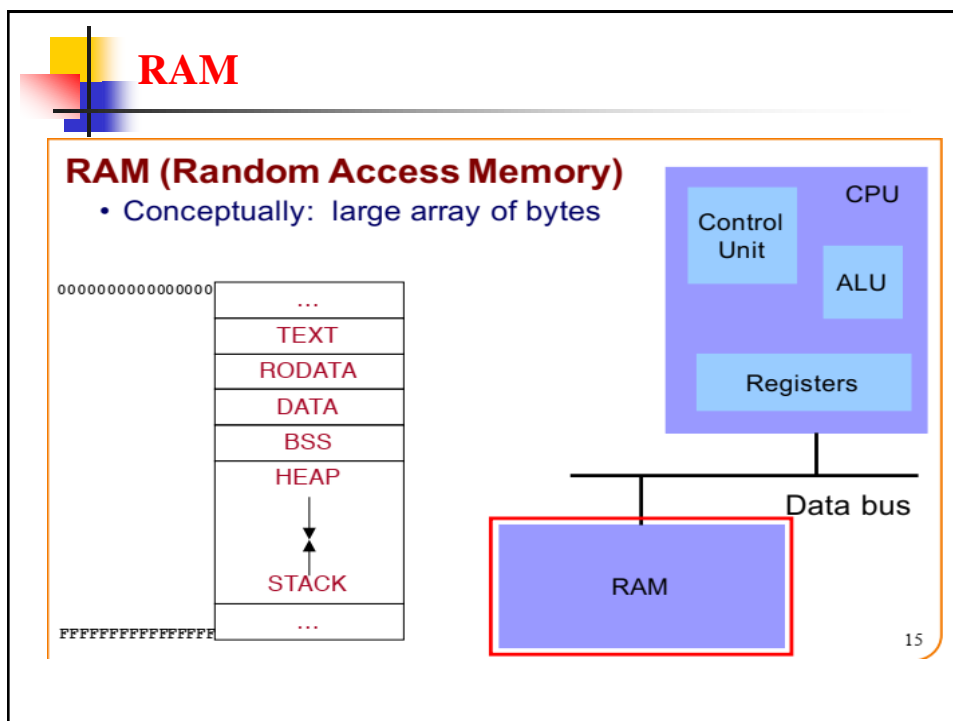
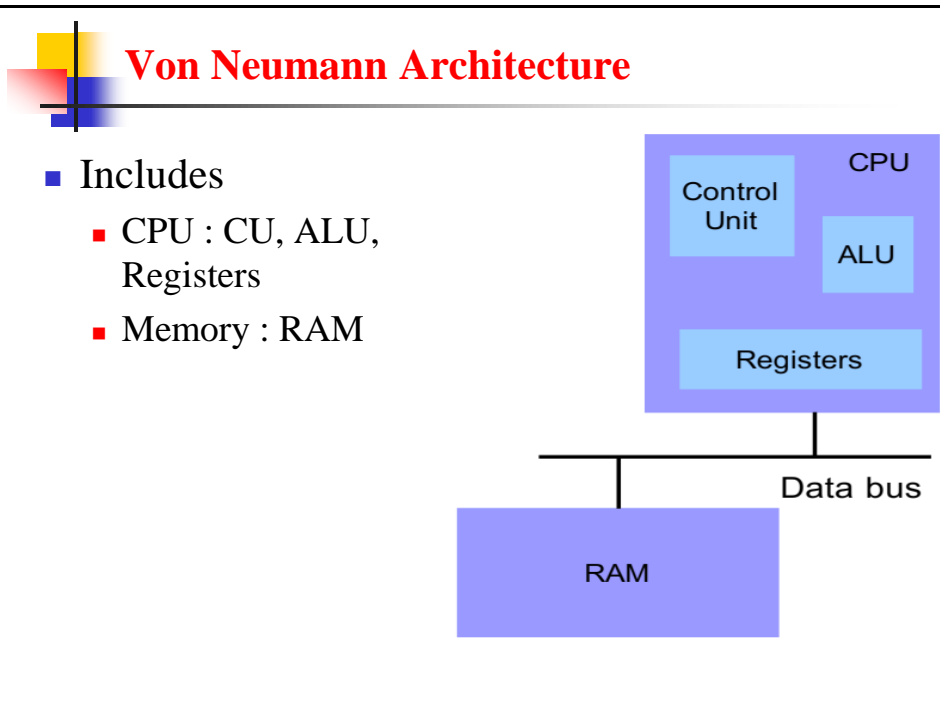
## Why learn assembly language ?

- Why learn assembly language ?
- Knowing assembly language helps you :
  - Write fast code : in assembly language, in a high – level language
  - Understand what's happening under the hood
    - Someone needs to develop future computer systems
    - Maybe that will be you



## Why Learn x86-64 Assembly Lang?

- Pros
  - X86-64 is popular
  - CourseLab computers are x86-64 computers
    - Program natively on CourseLab instead of using an emulator
- Cons
  - X86-64 assembly language is big
  - Each instruction is simple, but...
  - There are many instructions
  - Instructions differ widely





## Intel Microprocessors

- Intel introduced the 8086 microprocessor in 1979
- 8086, 8087, 8088, and 80186 processors
  - 16-bit processors with 16-bit registers
  - 16-bit data bus and 20-bit address bus
    - Physical address space =  $2^{20}$  bytes = 1 MB
  - 8087 **Floating-Point co-processor**
  - Uses **segmentation** and **real-address mode** to address memory
    - Each segment can address  $2^{16}$  bytes = 64 KB
  - 8088 is a less expensive version of 8086
    - Uses an 8-bit data bus
  - 80186 is a faster version of 8086



## Intel 80286 and 80386 Processors

- 80286 was introduced in 1982
  - 24-bit address bus  $\Rightarrow 2^{24}$  bytes = 16 MB address space
  - Introduced **protected mode**
    - Segmentation in protected mode is different from the real mode
- 80386 was introduced in 1985
  - First **32-bit processor** with 32-bit general-purpose registers
  - First processor to define the IA-32 architecture
  - 32-bit data bus and 32-bit address bus
  - $2^{32}$  bytes  $\Rightarrow$  4 GB address space
  - Introduced **paging**, **virtual memory**, and the **flat memory model**  $\rightarrow$  Segmentation can be turned off



## Intel 80486 and Pentium Processors

- 80486 was introduced 1989
  - Improved version of Intel 80386
  - On-chip **Floating-Point unit** (DX versions)
  - On-chip unified **Instruction/Data Cache** (8 KB)
  - Uses **Pipelining**: can execute up to 1 instruction per clock cycle
- Pentium (80586) was introduced in 1993
  - Wider 64-bit data bus, but address bus is still 32 bits
  - Two execution pipelines: U-pipe and V-pipe
    - **Superscalar** performance: can execute 2 instructions per clock/c
  - Separate 8 KB instruction and 8 KB data caches
  - **MMX instructions** (later models) for multimedia applications



## Intel P6 Processor Family

- P6 Processor Family: Pentium Pro, Pentium II and III
- Pentium Pro was introduced in 1995
  - **Three-way superscalar**: can execute 3 instructions per clock cycle
  - 36-bit address bus  $\Rightarrow$  up to 64 GB of physical address space
  - Introduced dynamic execution
    - **Out-of-order** and **speculative** execution
  - Integrates a 256 KB second level **L2 cache** on-chip
- Pentium II was introduced in 1997
  - Added **MMX instructions** (already introduced on Pentium MMX)
- Pentium III was introduced in 1999
  - Added **SSE instructions** and eight new 128-bit XMM registers




## Pentium 4 and Xeon Family

- Pentium 4 is a seventh-generation x86 architecture
  - Introduced in 2000
  - New micro-architecture design called Intel **Netburst**
  - Very deep instruction pipeline, scaling to very high frequencies
  - Introduced the **SSE2 instruction set** (extension to SSE)
    - Tuned for multimedia and operating on the 128-bit XMM registers
- In 2002, Intel introduced Hyper-Threading technology
  - Allowed 2 programs to run simultaneously, sharing resources
- Xeon is Intel's name for its server-class microprocessors
  - Xeon chips generally have more cache
  - Support larger multiprocessor configurations




## Pentium-M and EM64T

- Pentium M (**Mobile**) was introduced in 2003
  - Designed for **low-power** laptop computers
  - Modified version of Pentium III, optimized for power efficiency
  - Large second-level cache (2 MB on later models)
  - Runs at lower clock than Pentium 4, but with better performance
- Extended Memory 64-bit Technology (EM64T)
  - Introduced in 2004
  - 64-bit superset of the IA-32 processor architecture
  - 64-bit general-purpose registers and integer support
  - Number of general-purpose registers increased from 8 to 16
  - 64-bit pointers and flat virtual address space
  - Large physical address space: up to  $2^{40} = 1$  Terabytes



## 64-bit Processors


- Intel64
  - 64-bit linear address space
  - Intel: Pentium Extreme, Xeon, Celeron D, Pentium D, Core 2, and Core i7
- IA-32e Mode
  - Compatibility mode for legacy 16- and 32-bit applications
  - 64-bit Mode uses 64-bit addresses and operands



## CISC and RISC

- CISC – Complex Instruction Set Computer
  - Large and complex instruction set
  - Variable width instructions
  - Requires microcode interpreter
    - Each instruction is decoded into a sequence of micro-operations
  - Example: Intel x86 family
- RISC – Reduced Instruction Set Computer
  - Small and simple instruction set
  - All instructions have the same width
  - Simpler instruction formats and addressing modes
  - Decoded and executed directly by hardware
  - Examples: ARM, MIPS, PowerPC, SPARC, etc.





## Basic Program Execution Registers


- Registers are high speed memory inside the CPU
  - Eight 32-bit general-purpose registers
  - Six 16-bit segment registers
  - Processor Status Flags (EFLAGS) and Instruction Pointer (EIP)

32-bit General-Purpose Registers

EAX	EBP
EBX	ESP
ECX	ESI
EDX	EDI

16-bit Segment Registers

EFLAGS	CS	ES
EIP	SS	FS
	DS	GS



## General-Purpose Registers

- Used primarily for arithmetic and data movement
  - `mov eax, 10`      move constant 10 into register eax
- Specialized uses of Registers
  - EAX – **Accumulator** register
    - Automatically used by multiplication and division instructions
  - ECX – **Counter** register
    - Automatically used by LOOP instructions
  - ESP – **Stack Pointer** register
    - Used by PUSH and POP instructions, points to top of stack
  - ESI and EDI – **Source Index** and **Destination Index** register
    - Used by string instructions
  - EBP – **Base Pointer** register
    - Used to reference parameters and local variables on the stack

## Accessing Parts of Registers

- EAX, EBX, ECX, and EDX are 32-bit **Extended** registers
  - Programmers can access their 16-bit and 8-bit parts
  - Lower 16-bit of EAX is named
  - AX is further divided into
    - AL = lower 8 bits
    - AH = upper 8 bits
- ESI, EDI, EBP, ESP have only 16-bit names for lower half

Diagram illustrating the structure of the EAX register:  
 - 32-bit EAX register  
 - 16-bit AX register (lower half of EAX)  
 - 8-bit AH register (upper half of AX)  
 - 8-bit AL register (lower half of AX)  
 - Total: 8 bits + 8 bits = 16 bits for AX, 32 bits for EAX

32-bit	16-bit	8-bit (high)	8-bit (low)	32-bit	16-bit
EAX	AX	AH	AL	ESI	SI
EBX	BX	BH	BL	EDI	DI
ECX	CX	CH	CL	EBP	BP
EDX	DX	DH	DL	ESP	SP

## Accessing Parts of Registers

- Difference name of registers

**General purpose IA32 naming**

R0 EAX

R1 ECX

R2 EDX

R3 EBX

**General purpose IA32 naming**

R4 ESP

R5 EBP

R6 ESI

R7 EDI

**Old X86 naming**

AX

CX

DX

EX

**Old X86 naming**

SP

BP

SI

DI

**Data Registers**

**Pointer Registers**

**Index Registers**

## Special-Purpose & Segment Registers

### ■ EIP = Extended Instruction Pointer

- Contains address of next instruction to be executed

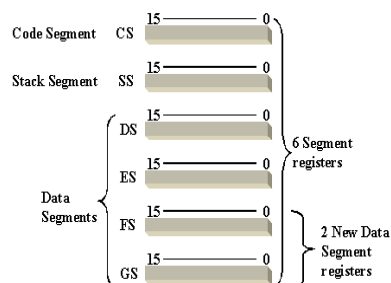
### ■ EFLAGS = Extended Flags Register

- Contains status and control flags
- Each flag is a single binary bit



### ■ Six 16-bit Segment Registers

- Support segmented memory
- Six segments accessible at a time
- Segments contain distinct contents
  - Code
  - Data
  - Stack



## EFLAGS Register

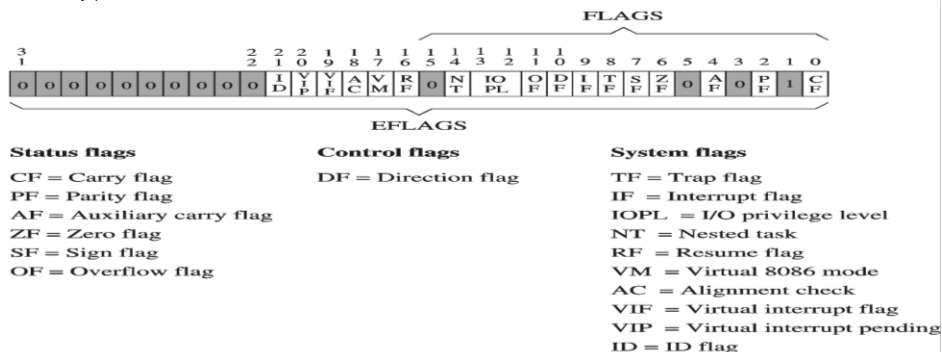
### ■ Status Flags


- Status of arithmetic and logical operations

### ■ Control and System flags

- Control the CPU operation


### ■ Programs can set and clear individual bits in the EFLAGS register





## Status Flags

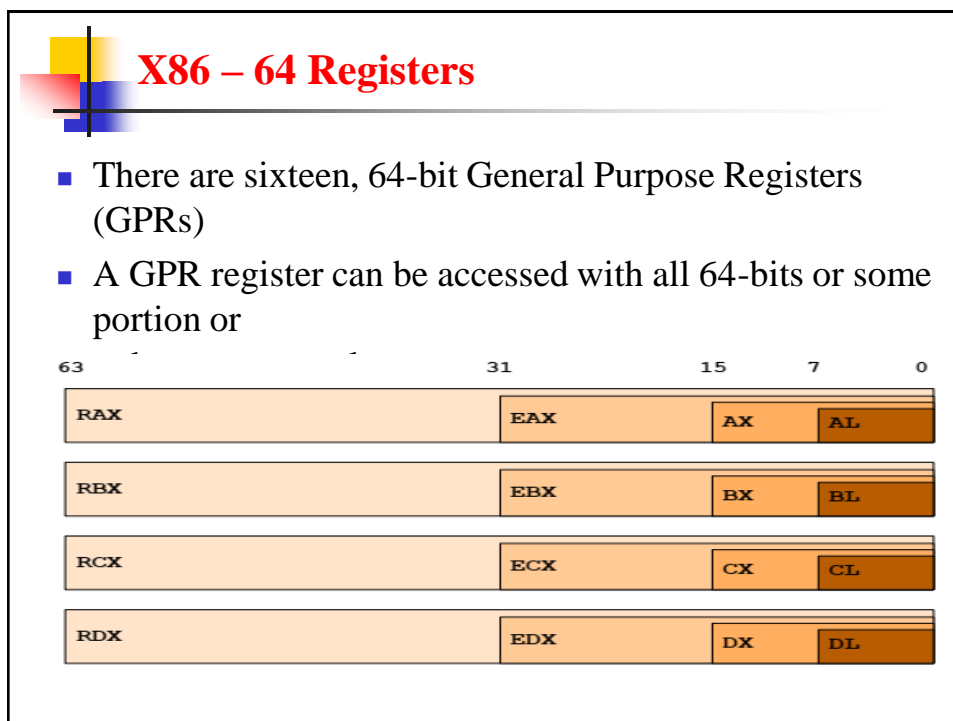
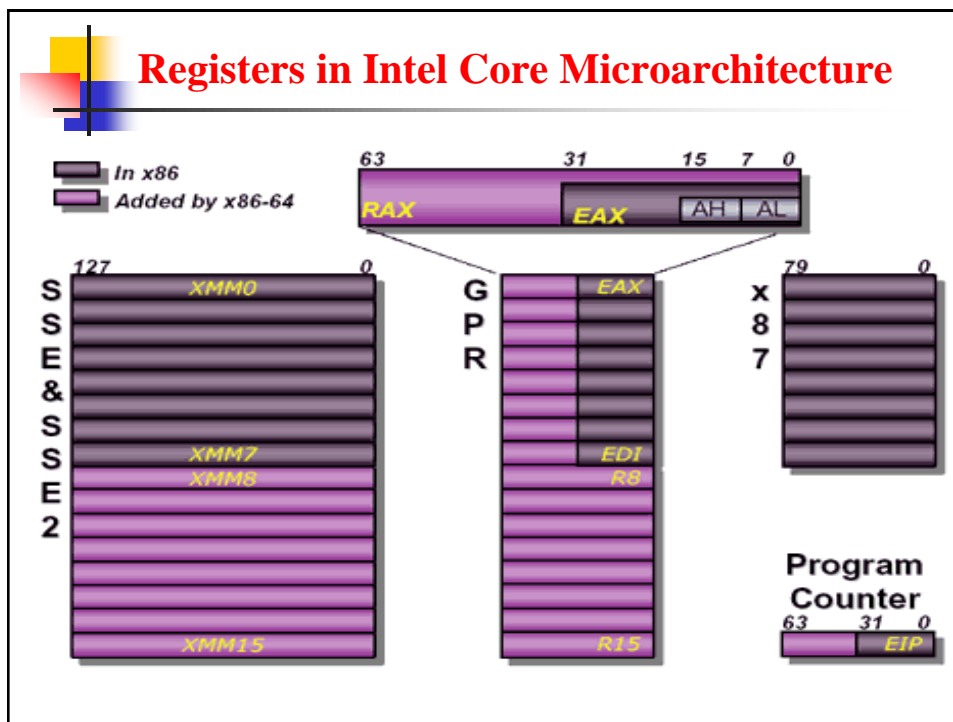
- Carry Flag
  - Set when **unsigned** arithmetic result is out of range
- Overflow Flag
  - Set when **signed** arithmetic result is out of range
- Sign Flag
  - Copy of **sign bit**, set when result is **negative**
- Zero Flag
  - Set when result is **zero**
- Auxiliary Carry Flag
  - Set when there is a **carry from bit 3 to bit 4**
- Parity Flag
  - Set when parity is **even**
  - Least-significant **byte** in result contains **even number of 1s**



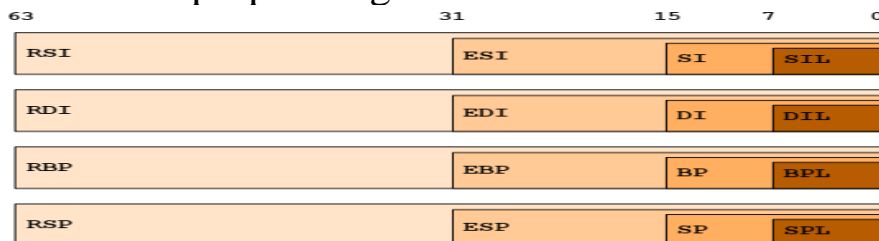
## Floating-Point, MMX, XMM Registers

- Floating-point unit performs high speed FP operations
- Eight 80-bit floating-point data registers
  - ST(0), ST(1), . . . , ST(7)
  - Arranged as a stack
  - Used for floating-point arithmetic
- Eight 64-bit MMX registers
  - Used with MMX instructions
- Eight 128-bit XMM registers
  - Used with SSE instructions

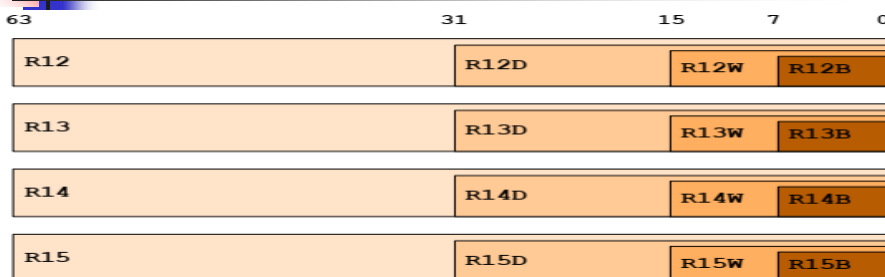
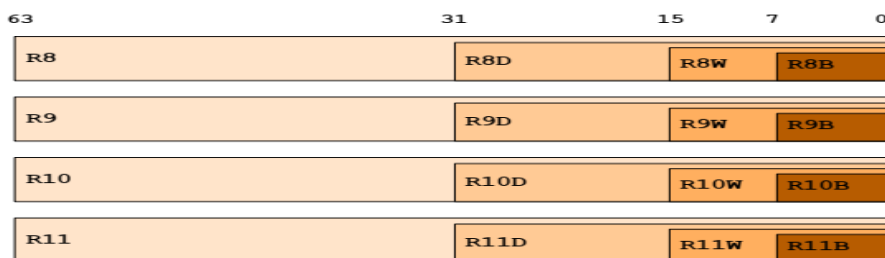
ST(0)
ST(1)
ST(2)
ST(3)
ST(4)
ST(5)
ST(6)
ST(7)



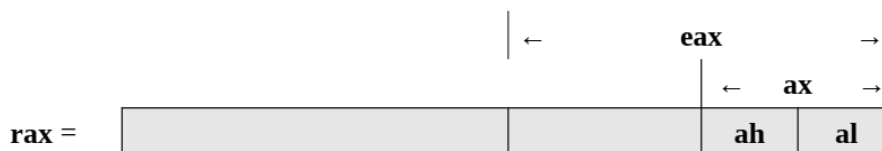
## ■ General purpose registers




RSP is unique; see upcoming slide




- When using data element sizes less than 64-bits (i.e., 32-bit, 16-bit, or 8-bit), the lower portion of the register can be accessed by using a different register name





64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
<b>rax</b>	<b>eax</b>	<b>ax</b>	<b>al</b>
<b>rbx</b>	<b>ebx</b>	<b>bx</b>	<b>bl</b>
<b>rcx</b>	<b>ecx</b>	<b>cx</b>	<b>cl</b>
<b>rdx</b>	<b>edx</b>	<b>dx</b>	<b>dl</b>
<b>rsi</b>	<b>esi</b>	<b>si</b>	<b>sil</b>
<b>rdi</b>	<b>edi</b>	<b>di</b>	<b>dil</b>
<b>rbp</b>	<b>ebp</b>	<b>bp</b>	<b>bpl</b>
<b>rsp</b>	<b>esp</b>	<b>sp</b>	<b>spl</b>
<b>r8</b>	<b>r8d</b>	<b>r8w</b>	<b>r8b</b>
<b>r9</b>	<b>r9d</b>	<b>r9w</b>	<b>r9b</b>
<b>r10</b>	<b>r10d</b>	<b>r10w</b>	<b>r10b</b>
<b>r11</b>	<b>r11d</b>	<b>r11w</b>	<b>r11b</b>
<b>r12</b>	<b>r12d</b>	<b>r12w</b>	<b>r12b</b>
<b>r13</b>	<b>r13d</b>	<b>r13w</b>	<b>r13b</b>
<b>r14</b>	<b>r14d</b>	<b>r14w</b>	<b>r14b</b>
<b>r15</b>	<b>r15d</b>	<b>r15w</b>	<b>r15b</b>



## RSP Register

- RSP (Stack Pointer) register
  - Contains address of top (low address) of current function's stack frame
- Allows use of the STACK section of memory
- (See Assembly Language: Function Calls lecture)

RSP


→

low memory


high memory

STACK frame

15



- RBP (Base Pointer Register)
  - is used as a base pointer during function calls
  - The **rbp** register should not be used for data or other uses.
- 



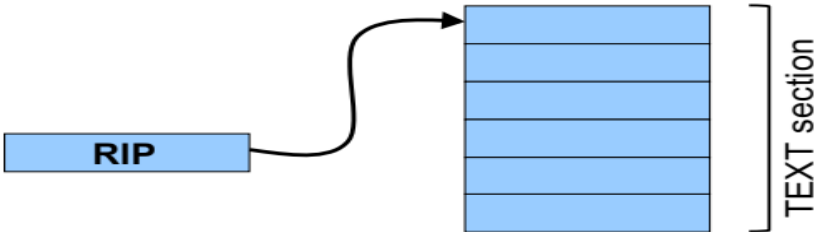
- Flag Register (rFlags)
 

Name	Symbol	Bit	Use
Carry	CF	0	Used to indicate if the previous operation resulted in a carry.
Parity	PF	2	Used to indicate if the last byte has an even number of 1's (i.e., even parity).
Zero	ZF	6	Used to indicate if the previous operation resulted in a zero result.
Sign	SF	7	Used to indicate if the result of the previous operation resulted in a 1 in the most significant bit (indicating negative in the context of signed data).
Direction	DF	10	Used to specify the direction (increment or decrement) for some string operations.
Overflow	OF	11	Used to indicate if the previous operation resulted in an overflow.



## RIP Register


- Special-purpose register...
- RIP (Instruction Pointer) register
  - Stores the location of the next instruction
    - Address (in TEXT section) of machine-language instructions to be executed next
  - Value changed:
    - Automatically to implement sequential control flow
    - By jump instructions to implement selection, repetition



The diagram illustrates the function of the RIP register. On the left, a blue rectangular box is labeled 'RIP'. A curved arrow originates from the right side of this box and points to the top of a vertical stack of six blue rectangular boxes. To the right of this stack is a vertical bracket with the label 'TEXT section' written vertically next to it, indicating that the stack represents memory instructions.


## Memory Segmentation

- Memory segmentation is necessary since the 20-bits memory addresses cannot fit in the 16-bits CPU registers
- Since x86 registers are 16-bits wide, a memory segment is made of  $2^{16}$  consecutive words (i.e. 64K words)
- Each segment has a number identifier that is also a 16-bit number (i.e. we have segments numbered from 0 to 64K)
- A memory location within a memory segment is referenced by specifying its offset from the start of the segment. Hence the first word in a segment has an offset of 0 while the last one has an offset of FFFFh
- To reference a memory location its logical address has to be specified. The logical address is written as:
  - Segment number:offset
- For example, A43F:3487h means offset 3487h within segment A43Fh.



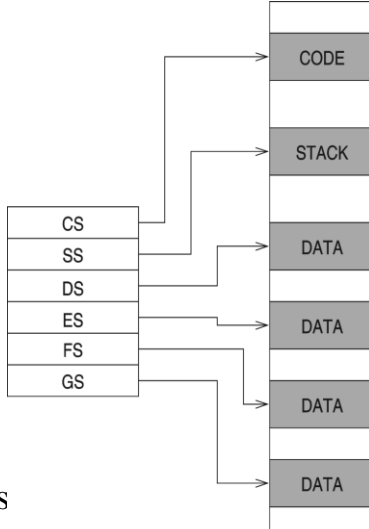
## Program Segments

- Machine language programs usually have 3 different parts stored in different memory segments:
  - **Instructions:** This is the code part and is stored in the code segment
  - **Data:** This is the data part which is manipulated by the code and is stored in the data segment
  - **Stack:** The stack is a special memory buffer organized as Last-In-First-Out (LIFO) structure used by the CPU to implement procedure calls and as a temporary holding area for addresses and data. This data structure is stored in the stack segment
- The segment numbers for the code segment, the data segment, and the stack segment are stored in the segment registers **CS**, **DS**, and **SS**, respectively.
- Program segments do not need to occupy the whole 64K locations in a segment



## Real Address Mode

- A program can access up to six segments at any time
  - Code segment
  - Stack segment
  - Data segment
  - Extra segments (up to 3)
- Each segment is 64 KB
- Logical address
  - Segment = 16 bits
  - Offset = 16 bits
- Linear (physical) address = 20 bits





## Logical to Linear Address Translation

$$\text{Linear address} = \text{Segment} \times 10 \text{ (hex)} + \text{Offset}$$

Example:

segment = A1F0 (hex)

offset = 04C0 (hex)

logical address = A1F0:04C0 (hex)

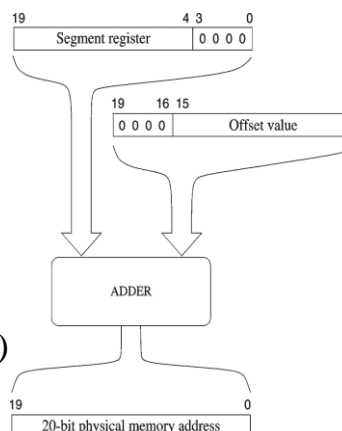
what is the linear address?

Solution:

**A1F00** (add 0 to segment in hex)

**+ 04C0** (offset in hex)

**A23C0** (20-bit linear address in hex)



## Segment Overlap

- There is a lot of overlapping between segments in the main memory.
- A new segment starts every 10h locations (i.e. every 16 locations).
- Starting address of a segment always has a 0h LSD.
- Due to segments overlapping logical addresses are not unique .

End of Segment 1	1000F
	1000E
	10000
End of Segment 0	0FFFF
	0FFFE
	00021
Start of Segment 2	00020
	0001F
	00011
Start of Segment 1	00010
	0000F
	00003
	00002
	00001
Start of Segment 0	00000

## Flat Memory Model

- Modern operating systems turn segmentation off
- Each program uses **one 32-bit linear address space**
  - Up to  $2^{32} = 4$  GB of memory can be addressed
  - Segment registers are defined by the operating system
  - All segments are mapped to the **same linear address space**
- In assembly language, we use **.MODEL flat** directive
  - To indicate the Flat memory model
- A **linear address** is also called a **virtual address**
  - Operating system maps **virtual address** onto **physical addresses**
  - Using a technique called **paging**

## Programmer View of Flat Memory

- Same base address for all segments
  - All segments are mapped to the **same linear address space**
- EIP Register
  - Points at next instruction
- ESI and EDI Registers
  - Contain data addresses
  - Used also to index arrays
- ESP and EBP Registers
  - ESP points at top of stack
  - EBP is used to address parameters and variables on the stack

Linear address space of a program (up to 4 GB)

32-bit address

ESI → DATA

EDI → DATA

32-bit address

EIP → CODE

32-bit address

EBP → STACK

ESP → STACK

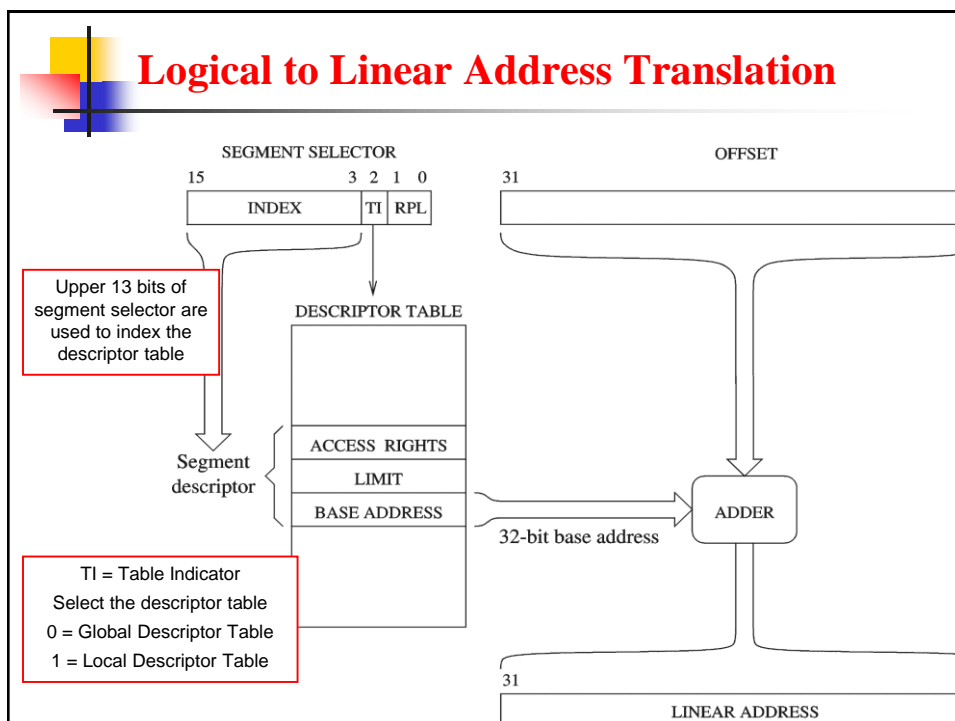
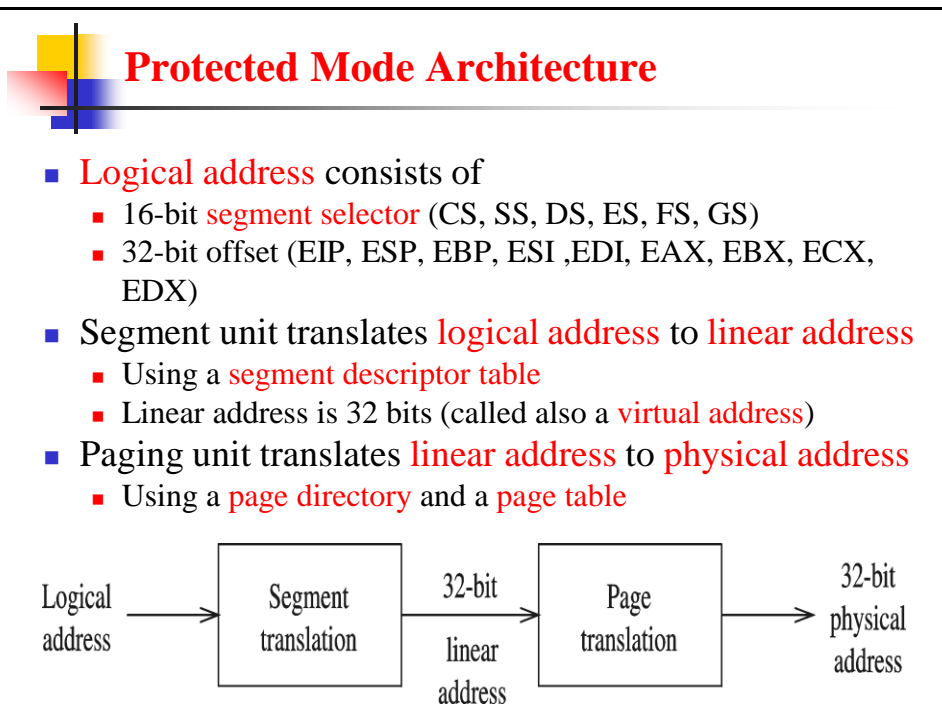
CS → Unused


DS → Unused

SS → Unused

ES → Unused


base address = 0 for all segments






## Segment Descriptor Tables

- **Global descriptor table (GDT)**
  - Only one GDT table is provided by the operating system
  - GDT table contains segment descriptors for all programs
  - Also used by the operating system itself
  - Table is initialized during boot up
  - GDT table address is stored in the **GDTR register**
  - Modern operating systems (Windows-XP) use one GDT table
- **Local descriptor table (LDT)**
  - Another choice is to have a unique LDT table for each program
  - LDT table contains segment descriptors for only one program
  - LDT table address is stored in the **LDTR register**



## Segment Descriptor Details


- **Base Address**
  - 32-bit number that defines the starting location of the segment
  - $32\text{-bit Base Address} + 32\text{-bit Offset} = 32\text{-bit Linear Address}$
- **Segment Limit**
  - 20-bit number that specifies the size of the segment
  - The size is specified either in bytes or multiple of 4 KB pages
  - Using 4 KB pages, segment size can range from 4 KB to 4 GB
- **Access Rights**
  - Whether the segment contains code or data
  - Whether the data can be read-only or read & written
  - Privilege level of the segment to protect its access



## Segment Visible and Invisible Parts

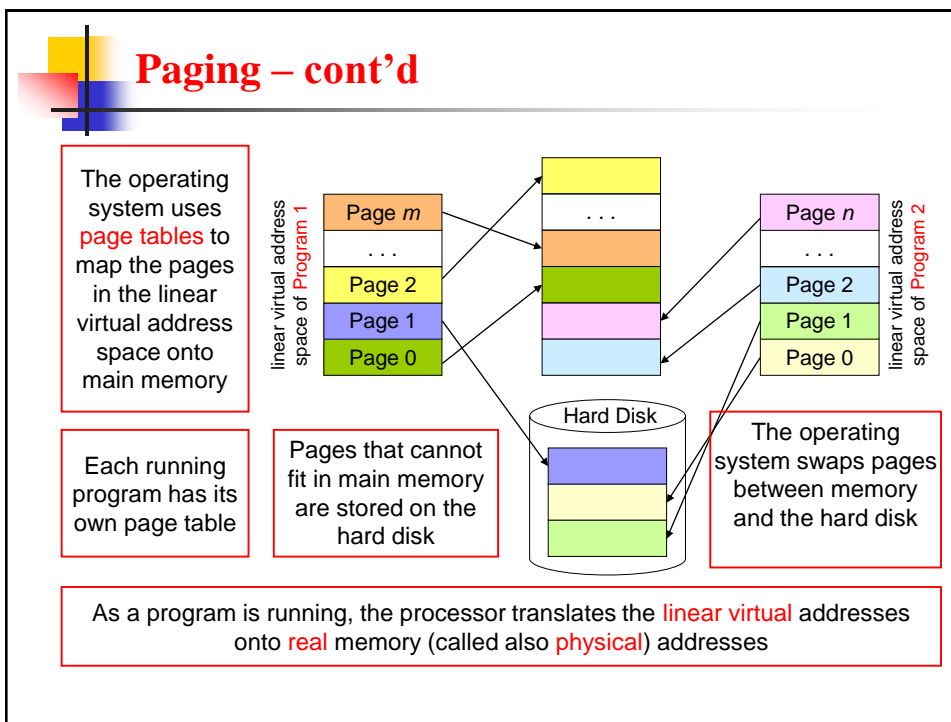
- Visible part = 16-bit Segment Register
  - CS, SS, DS, ES, FS, and GS are visible to the programmer
- Invisible Part = Segment Descriptor (64 bits)
  - Automatically loaded from the descriptor table

Visible part	Invisible part	
Segment selector	Segment base address, size, access rights, etc.	CS
Segment selector	Segment base address, size, access rights, etc.	SS
Segment selector	Segment base address, size, access rights, etc.	DS
Segment selector	Segment base address, size, access rights, etc.	ES
Segment selector	Segment base address, size, access rights, etc.	FS
Segment selector	Segment base address, size, access rights, etc.	GS



## Paging

- Paging divides the linear address space into ...
  - Fixed-sized blocks called pages, Intel IA-32 uses 4 KB pages
- Operating system allocates main memory for pages
  - Pages can be spread all over main memory
  - Pages in main memory can belong to different programs
  - If main memory is full then pages are stored on the hard disk
- OS has a Virtual Memory Manager (VMM)
  - Uses page tables to map the pages of each running program
  - Manages the loading and unloading of pages
- As a program is running, CPU does address translation
- Page fault: issued by CPU when page is not in memory




## x86 Assembly Language Syntax

- An assembly statement
  - 3 essentials: opcode, operands (dest, src)
  - E.g.,: add a, b, c => c = a+b
- Intel Syntax
  - opcode dest src
  - no suffix for opcode
  - immmed. val. is number
  - Plain register name
  - [ ] for memory operand
  - Example :
 

```
MOV EAX, 5
MOV EAX, [EBX+4]
```
- AT&T Syntax
  - opcode src dest
  - has suffix for opcode
  - b/w/l/q, 8|16|32|64-bits
  - immediate value has \$
  - Register name has %
  - ( ) for memory operand
  - Example :
 

```
MOVL $5, %EAX
MOV 4(%EBX), %EAX
```






## x86 Operand Addressing Modes

- Addressing mode is about where the operands are.
- Immediate Operands – operand values are part of the instruction
  - `movl $5, %eax`
- Register Operands – operand values are in registers
  - `add %ebx, %eax`
- Memory Operands – operand values are in memory
  - Addressed with 

SEGMENT	OFFSET
---------	--------
- Specifying the segment
  - Explicitly specify segments: `movl %eax, %es:(%ebx)`
  - Implicitly specify segments



- Implicitly specify segment

Reference Type	Register Used	Segment Type	Default Use
Instructions	CS	Code Segment	Instruction fetch
Stack	SS	Stack Segment	Anything stack (push etc.)
Local Data	DS	Data Segment	All data ref (non stack/str)
Strings	ES	Extra Segment	Dest of string instructions

## ■ Specifying the offset

- $\text{Offset} = \text{Base} + \text{Displacement} + (\text{Index} * \text{scale})$

Base		Disp		Index		Scale
EAX				EAX		1
EBX		8-BIT		EBX		
ECX	+		+	ECX		2
EDX		16-BIT		EDX		
ESP				EBP		4
EBP				ESI		
ESI		32-BIT		EDI		8
EDI						

- Displacement is a number
- Special rules about ESP and EBP
- Any component can be NULL

## ■ Example memory operands

### ■ Writing the memory operand

- Intel syntax: `segreg:[base+index*scale+disp]`
- AT&T syntax: `%segreg:disp(base, index, scale)`

### ■ Example:

- `int a[2][10];` and move `a[1][2]` into EAX

### ■ Intel syntax

```
mov ebx, a;
mov ecx, 2;
mov eax, ds:[ebx + ecx * 4h + 40]
```

### ■ AT&T syntax

```
movl &a, %ebx;
movl $2, %ecx;
movl %ds:40(%ebx, %ecx, 0x4),%eax
```

## From Assembly Language to Instructions

- Assembly language is for human
- Instructions are for machine
- One assembly language statement translates into one instruction
- Why this translation is important
  - x86 is CISC, i.e., instructions have variable lengths
  - To properly deliver/inspect malicious code at the correct memory location, attackers/defenders should be aware of the length of the instructions

## Summary of addressing modes

