



Computer Architecture and Assembly Language

COS 217

Goals of Today's Lecture



- **Computer architecture**
 - Central processing unit (CPU)
 - Fetch-decode-execute cycle
 - Memory hierarchy, and other optimization
- **Assembly language**
 - Machine vs. assembly vs. high-level languages
 - Motivation for learning assembly language
 - Intel Architecture (IA32) assembly language



Levels of Languages

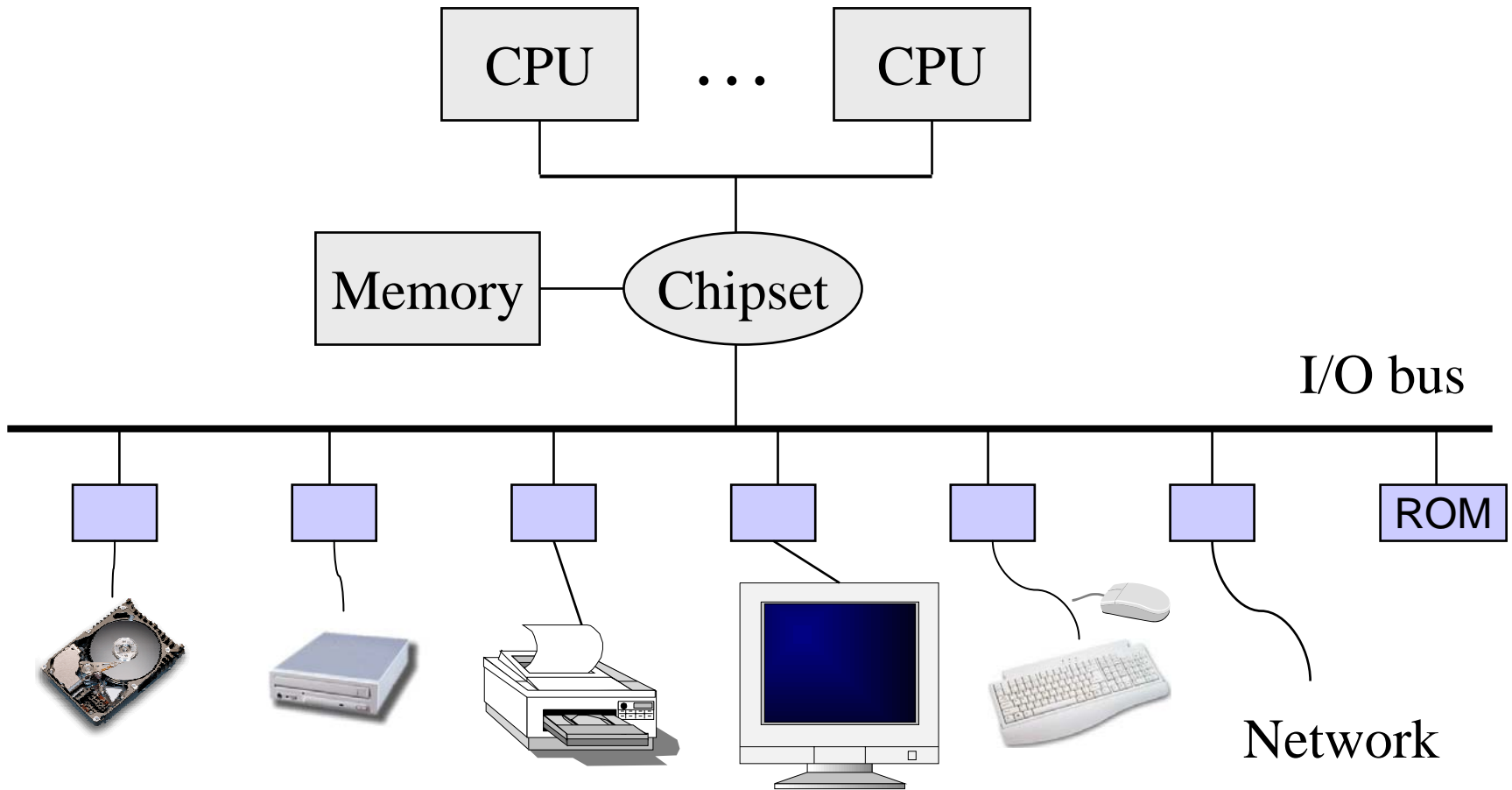
- **Machine language**
 - What the computer sees and deals with
 - Every command is a sequence of one or more numbers
- **Assembly language**
 - Command numbers replaced by letter sequences that are easier to read
 - Still have to work with the specifics of the machine itself
- **High-level language**
 - Make programming easier by describing operations in a natural language
 - A single command replaces a group of low-level assembly language commands

Why Learn Assembly Language?



- Understand how things work underneath
 - Learn the basic organization of the underlying machine
 - Learn how the computer actually runs a program
 - Design better computers in the future
- Write faster code (even in high-level language)
 - By understanding which high-level constructs are better
 - ... in terms of how efficient they are at the machine level
- Some software is still written in assembly language
 - Code that really needs to run quickly
 - Code for embedded systems, network processors, etc.

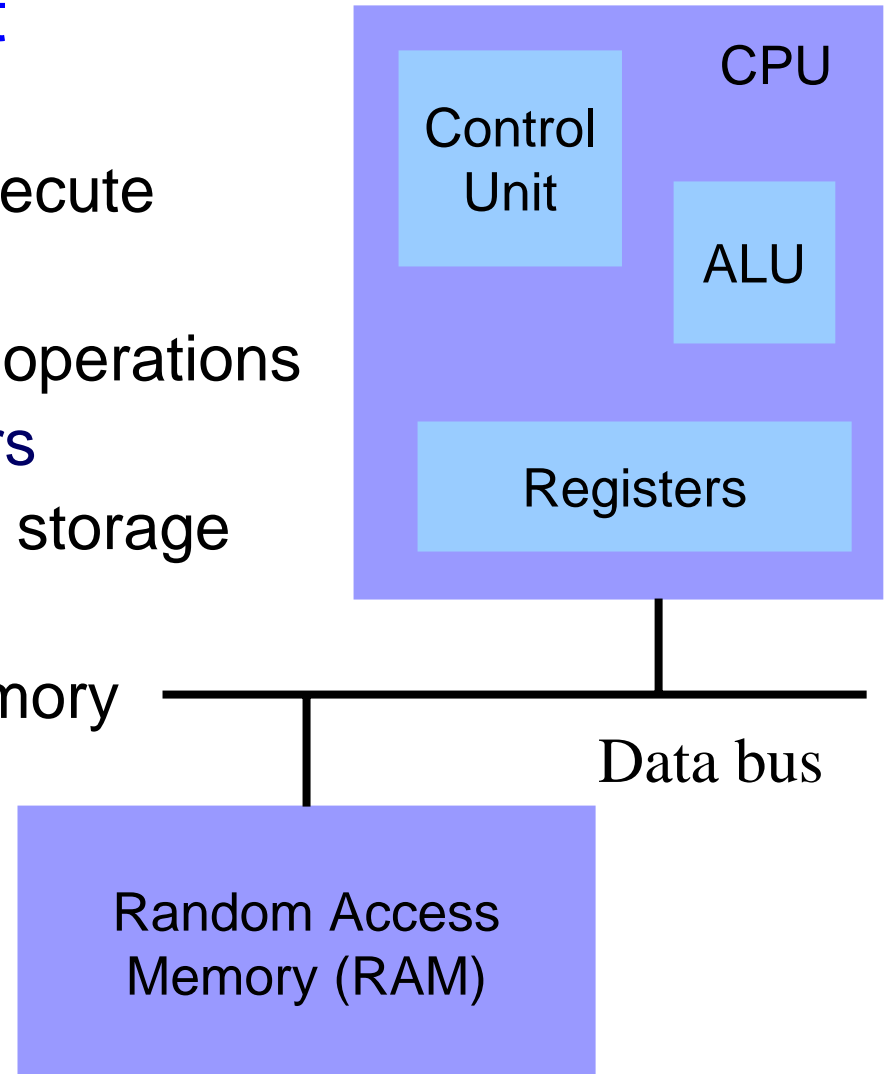
A Typical Computer



Von Neumann Architecture



- **Central Processing Unit**
 - Control unit
 - Fetch, decode, and execute
 - Arithmetic and logic unit
 - Execution of low-level operations
 - General-purpose registers
 - High-speed temporary storage
 - Data bus
 - Provide access to memory
- **Memory**
 - Store instructions
 - Store data



Control Unit



- **Instruction pointer**
 - Stores the location of the next instruction
 - Address to use when reading from memory
 - Changing the instruction pointer
 - Increment by one to go to the next instruction
 - Or, load a new value to “jump” to a new location
- **Instruction decoder**
 - Determines what operations need to take place
 - Translate the machine-language instruction
 - Control the registers, arithmetic logic unit, and memory
 - E.g., control which registers are fed to the ALU
 - E.g., enable the ALU to do multiplication
 - E.g., read from a particular address in memory



Example: Kinds of Instructions

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

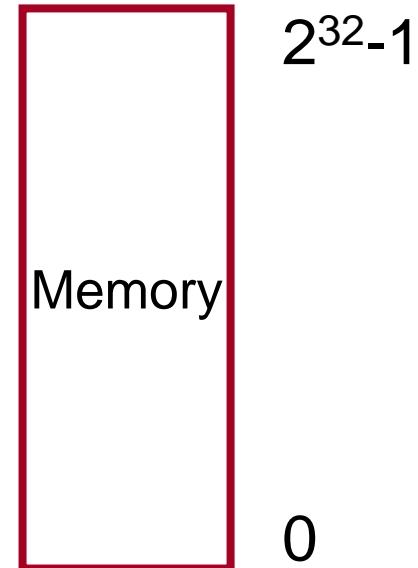
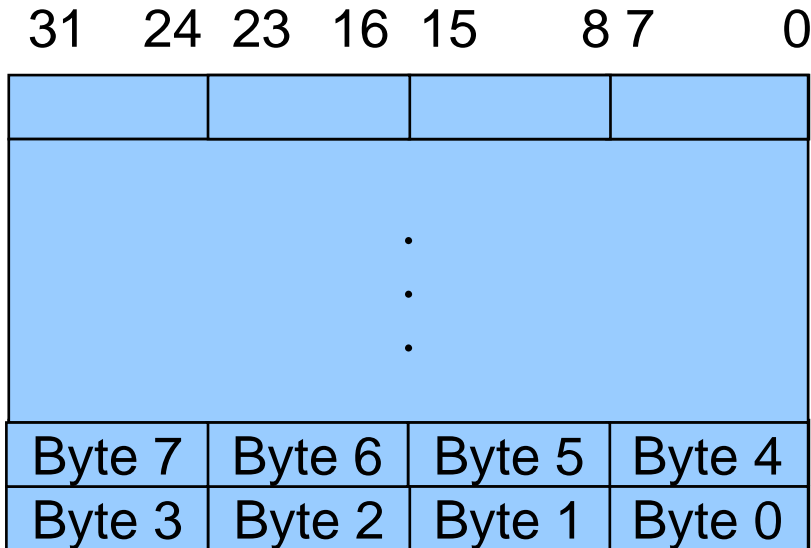
- **Storing values in registers**
 - count = 0
 - n
- **Arithmetic and logic operations**
 - Increment: count++
 - Multiply: n * 3
 - Divide: n/2
 - Logical AND: n & 1
- **Checking results of comparisons**
 - while (n > 1)
 - if (n & 1)
- **Jumping**
 - To the end of the while loop (if “n > 1”)
 - Back to the beginning of the loop
 - To the else clause (if “n & 1” is 0)

Size of Variables



- Data types in high-level languages vary in size
 - Character: 1 byte
 - Short, int, and long: varies, depending on the computer
 - Pointers: typically 4 bytes
 - Struct: arbitrary size, depending on the elements
- Implications
 - Need to be able to store and manipulate in multiple sizes
 - Byte (1 byte), word (2 bytes), and extended (4 bytes)
 - Separate assembly-language instructions
 - e.g., addb, addw, addl
 - Separate ways to access (parts of) a 4-byte register

Four-Byte Memory Words



Byte order is little endian

IA32 General Purpose Registers



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

General-purpose registers

Registers for Executing the Code



- Execution control flow
 - Instruction pointer (EIP)
 - Address in memory of the current instruction
 - Flags (EFLAGS)
 - Stores the status of operations, such as comparisons
 - E.g., last result was positive/negative, was zero, etc.
- Function calls (more on these later!)
 - Stack register (ESP)
 - Address of the top of the stack
 - Base pointer (EBP)
 - Address of a particular element on the stack
 - Access function parameters and local variables

Other Registers that you don't much care about



- Segment registers
 - CS, SS, DS, ES, FS, GS
- Floating Point Unit (FPU) (x87)
 - Eight 80-bit registers (ST0, ..., ST7)
 - 16-bit control, status, tag registers
 - 11-bit opcode register
 - 48-bit FPU instruction pointer, data pointer registers
- MMX
 - Eight 64-bit registers
- SSE and SSE2
 - Eight 128-bit registers
 - 32-bit MXCRS register
- System
 - I/O ports
 - Control registers (CR0, ..., CR4)
 - Memory management registers (GDTR, IDTR, LDTR)
 - Debug registers (DR0, ..., DR7)
 - Machine specific registers
 - Machine check registers
 - Performance monitor registers

Reading IA32 Assembly Language



- **Assembler directives: starting with a period (“.”)**
 - E.g., “.section .text” to start the text section of memory
 - E.g., “.loop” for the address of an instruction
- **Referring to a register: percent size (“%”)**
 - E.g., “%ecx” or “%eip”
- **Referring to a constant: dollar sign (“\$”)**
 - E.g., “\$1” for the number 1
- **Storing result: typically in the second argument**
 - E.g. “addl \$1, %ecx” increments register ECX
 - E.g., “movl %edx, %eax” moves EDX to EAX
- **Comment: pound sign (“#”)**
 - E.g., “# Purpose: Convert lower to upper case”

Detailed Example

n %edx
count %ecx



```
count=0;
```

```
while (n>1) {
```

```
    count++;
```

```
    if (n&1)
```

```
        n = n*3+1;
```

```
    else
```

```
        n = n/2;
```

```
}
```

```
    movl    $0, %ecx
```

```
.loop:
```

```
    cmpl    $1, %edx
```

```
    jle     .endloop
```

```
    addl    $1, %ecx
```

```
    movl    %edx, %eax
```

```
    andl    $1, %eax
```

```
    je      .else
```

```
    movl    %edx, %eax
```

```
    addl    %eax, %edx
```

```
    addl    %eax, %edx
```

```
    addl    $1, %edx
```

```
    jmp     .endif
```

```
.else:
```

```
    sarl    $1, %edx
```

```
.endif:
```

```
    jmp     .loop
```

```
.endloop:
```

Machine-Language Instructions



Instructions have the form

op source, dest “dest \leftarrow dest \oplus source”

operation (move, add, subtract, etc.)

first operand (and destination)

second operand

Machine Language



- Machine language encodes instructions as a sequence of integers easily decodable (fast!) by the machine
- Instruction format:



Opcode specifies “what operation to perform” (add, subtract, load, jump, etc.)

Operand specifies what data on which to perform the operation (register A, memory at address B, etc.)



Instruction

- **Opcode**
 - What to do
- **Source operands**
 - Immediate (in the instruction itself)
 - Register
 - Memory location
 - I/O port
- **Destination operand**
 - Register
 - Memory location
 - I/O port
- **Assembly syntax**

Opcode source1, [source2,] destination

How Many Instructions to Have?



- Need a certain minimum set of functionality
 - Want to be able to represent any computation that can be expressed in a higher-level language
- Benefits of having many instructions
 - Direct implementation of many key operations
 - Represent a line of C in one (or just a few) lines of assembly
- Disadvantages of having many instructions
 - Larger opcode size
 - More complex logic to implement complex instructions
 - Hard to write compilers to exploit all the available instructions
 - Hard to optimize the implementation of the CPU

CISC vs. RISC



Complex Instruction Set Computer

(old fashioned, 1970s style)

Examples:

Vax (1978-90)

Motorola 68000 (1979-90)

8086/80x86/Pentium (1974-2025)

Instructions of various lengths,
designed to economize on
memory (size of instructions)

Reduced Instruction Set Computer

("modern", 1980s style)

Examples:

MIPS (1985-?)

Sparc (1986-2006)

IBM PowerPC (1990-?)

ARM

Instructions all the same size and
all the same format, designed to
economize on decoding
complexity (and time, and power
drain)



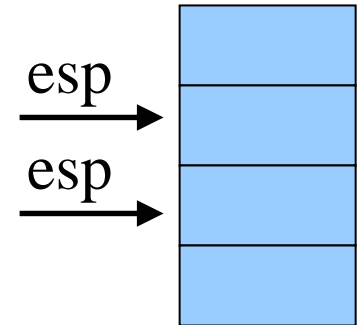
Data Transfer Instructions

- **mov{b,w,l} source, dest**

- General move instruction

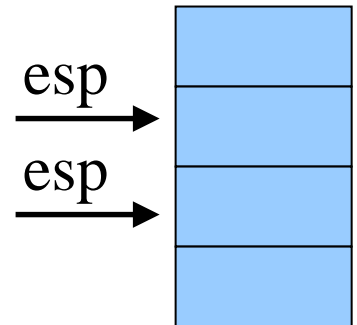
- **push{w,l} source**

```
pushl %ebx    # equivalent instructions
              subl $4, %esp
              movl %ebx, (%esp)
```



- **pop{w,l} dest**

```
popl %ebx    # equivalent instructions
              movl (%esp), %ebx
              addl $4, %esp
```



- **Many more in Intel manual (volume 2)**

- Type conversion, conditional move, exchange, compare and exchange, I/O port, string move, etc.



Data Access Methods

- **Immediate addressing:** data stored in the instruction itself
 - `movl $10, %ecx`
- **Register addressing:** data stored in a register
 - `movl %eax, %ecx`
- **Direct addressing:** address stored in instruction
 - `movl 2000, %ecx`
- **Indirect addressing:** address stored in a register
 - `movl (%eax), %ebx`
- **Base pointer addressing:** includes an offset as well
 - `movl 4(%eax), %ebx`
- **Indexed addressing:** instruction contains base address, and specifies an index register and a multiplier (1, 2, or 4)
 - `movl 2000(,%ecx,1), %ebx`



Effective Address

$$\text{Offset} = \left(\begin{array}{c} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{array} \right) + \left(\begin{array}{c} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{array} \right) * \left(\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \right) + \left(\begin{array}{c} \text{None} \\ 8\text{-bit} \\ 16\text{-bit} \\ 32\text{-bit} \end{array} \right)$$

Base Index scale displacement

- Displacement `movl foo, %eax`
- Base `movl (%eax), %ebx`
- Base + displacement `movl foo(%eax), %ebx`
`movl 1(%eax), %ebx`
- (Index * scale) + displacement `movl (,%eax,4), %ebx`
- Base + (index * scale) + displacement `movl foo(,%eax,4), %ebx`



Bitwise Logic Instructions

- Simple instructions

and{b,w,l} source, dest

or{b,w,l} source, dest

xor{b,w,l} source, dest

not{b,w,l} dest

sal{b,w,l} source, dest (arithmetic)

sar{b,w,l} source, dest (arithmetic)

dest = source & dest

dest = source | dest

dest = source ^ dest

dest = ^dest

dest = dest << source

dest = dest >> source

- Many more in Intel Manual (volume 2)

- Logic shift
- Rotation shift
- Bit scan
- Bit test
- Byte set on conditions



Arithmetic Instructions

- Simple instructions

- `add{b,w,l} source, dest` $dest = source + dest$
- `sub{b,w,l} source, dest` $dest = dest - source$
- `inc(b,w,l) dest` $dest = dest + 1$
- `dec{b,w,l} dest` $dest = dest - 1$
- `neg(b,w,l) dest` $dest = \sim dest$
- `cmp{b,w,l} source1, source2` $source2 - source1$

- Multiply

- `mul (unsigned) or imul (signed)`
`mul %ebx # edx, eax = eax * ebx`

- Divide

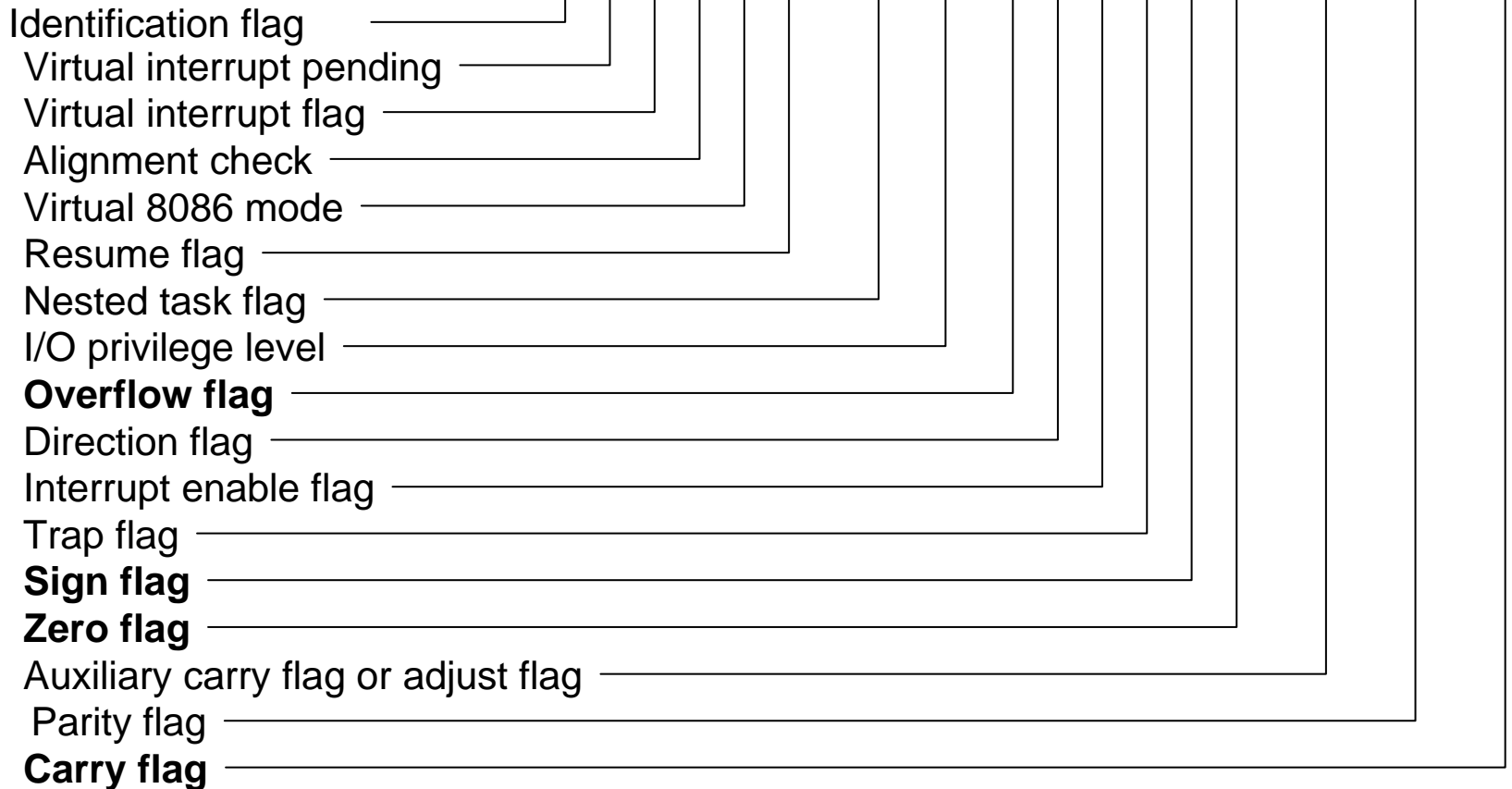
- `div (unsigned) or idiv (signed)`
`idiv %ebx # edx = edx, eax / ebx`

- Many more in Intel manual (volume 2)

- `adc, sbb, decimal arithmetic instructions`

EFLAG Register &

Condition Codes





Branch Instructions

- Conditional jump

- $j\{l,g,e,ne,\dots\}$ target if (condition) {eip = target}

Comparison	Signed	Unsigned	
=	e	e	<i>“equal”</i>
≠	ne	ne	<i>“not equal”</i>
>	g	a	<i>“greater,above”</i>
≥	ge	ae	<i>“...-or-equal”</i>
<	l	b	<i>“less,below”</i>
≤	le	be	<i>“...-or-equal”</i>
overflow/carry	o	c	
no ovf/carry	no	nc	

- Unconditional jump

- jmp target
 - jmp *register

Making the Computer Faster



- **Memory hierarchy**
 - Ranging from small, fast storage to large, slow storage
 - E.g., registers, caches, main memory, disk, CDROM, ...
- **Sophisticated logic units**
 - Have dedicated logic units for specialized functions
 - E.g., right/left shifting, floating-point operations, graphics, network,...
- **Pipelining**
 - Overlap the fetch-decode-execute process
 - E.g., execute instruction i , while decoding $i-1$, and fetching $i-2$
- **Branch prediction**
 - Guess which way a branch will go to avoid stalling the pipeline
 - E.g., assume the “for loop” condition will be true, and keep going
- **And so on... see the Computer Architecture class!**

Memory Hierarchy



Capacity

Access time

10^2 bytes

Register: 1x

10^4 bytes

L1 cache: 2-4x

10^5 bytes

L2 cache: ~10x

10^6 bytes

L3 cache: ~50x

10^9 bytes

DRAM: ~200-500x

10^{11} bytes

Disks: ~30M x

10^{12} bytes

CD-ROM Jukebox: >1000M x

Conclusion



- **Computer architecture**
 - Central Processing Unit (CPU) and Random Access Memory (RAM)
 - Fetch-decode-execute cycle
 - Instruction set
- **Assembly language**
 - Machine language represented with handy mnemonics
 - Example of the IA-32 assembly language
- **Next time**
 - Portions of memory: data, bss, text, stack, etc.
 - Function calls, and manipulating contents of the stack