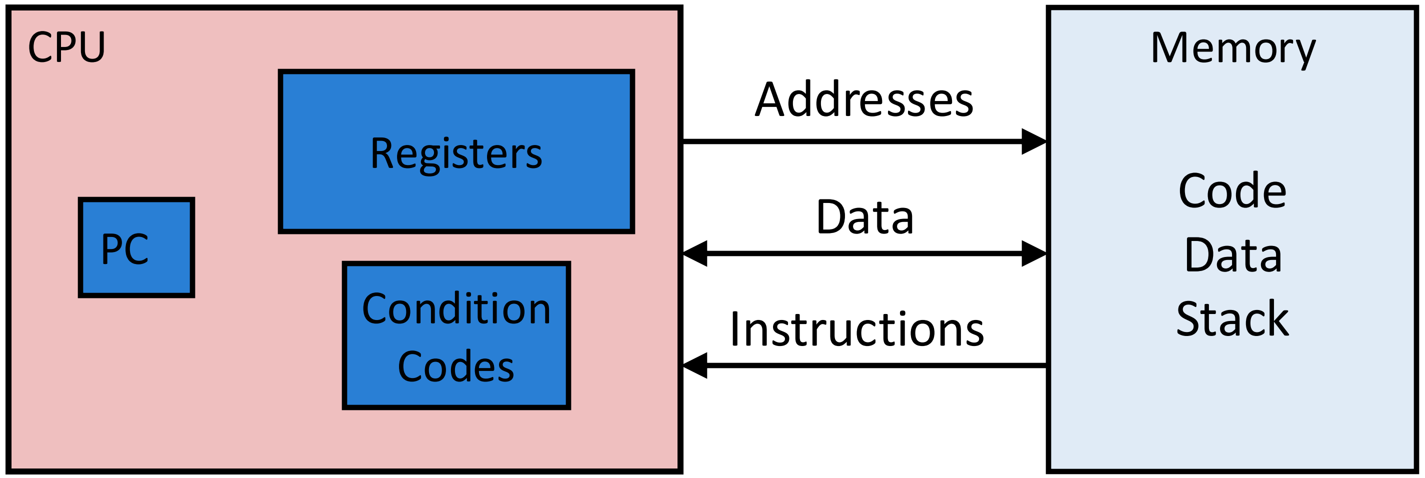
x86 (Dis)assembly

Georgios (George) Portokalidis

How the World Is (for CPUs)

- PC: Program counter
  - Address of next instruction
- Memory
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching



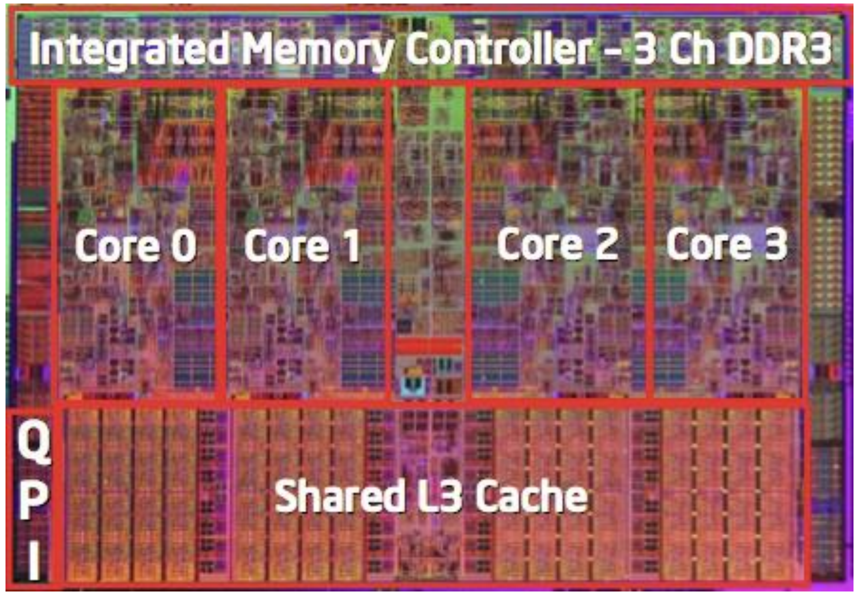
Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But Intel has done just that!
    - In terms of speed. Less so for low power.

Intel x86 Processors

- Machine Evolution

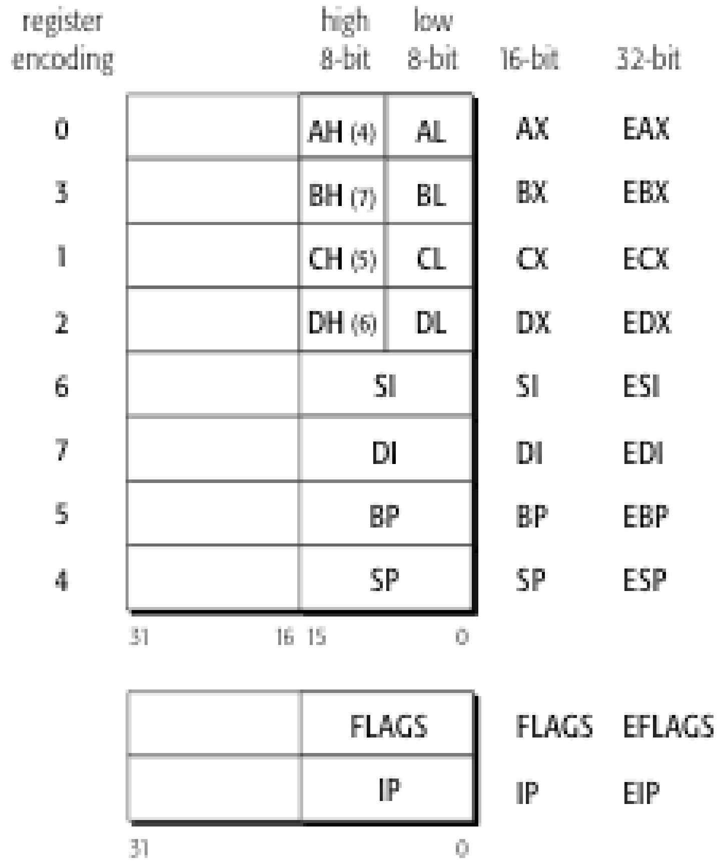
386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M



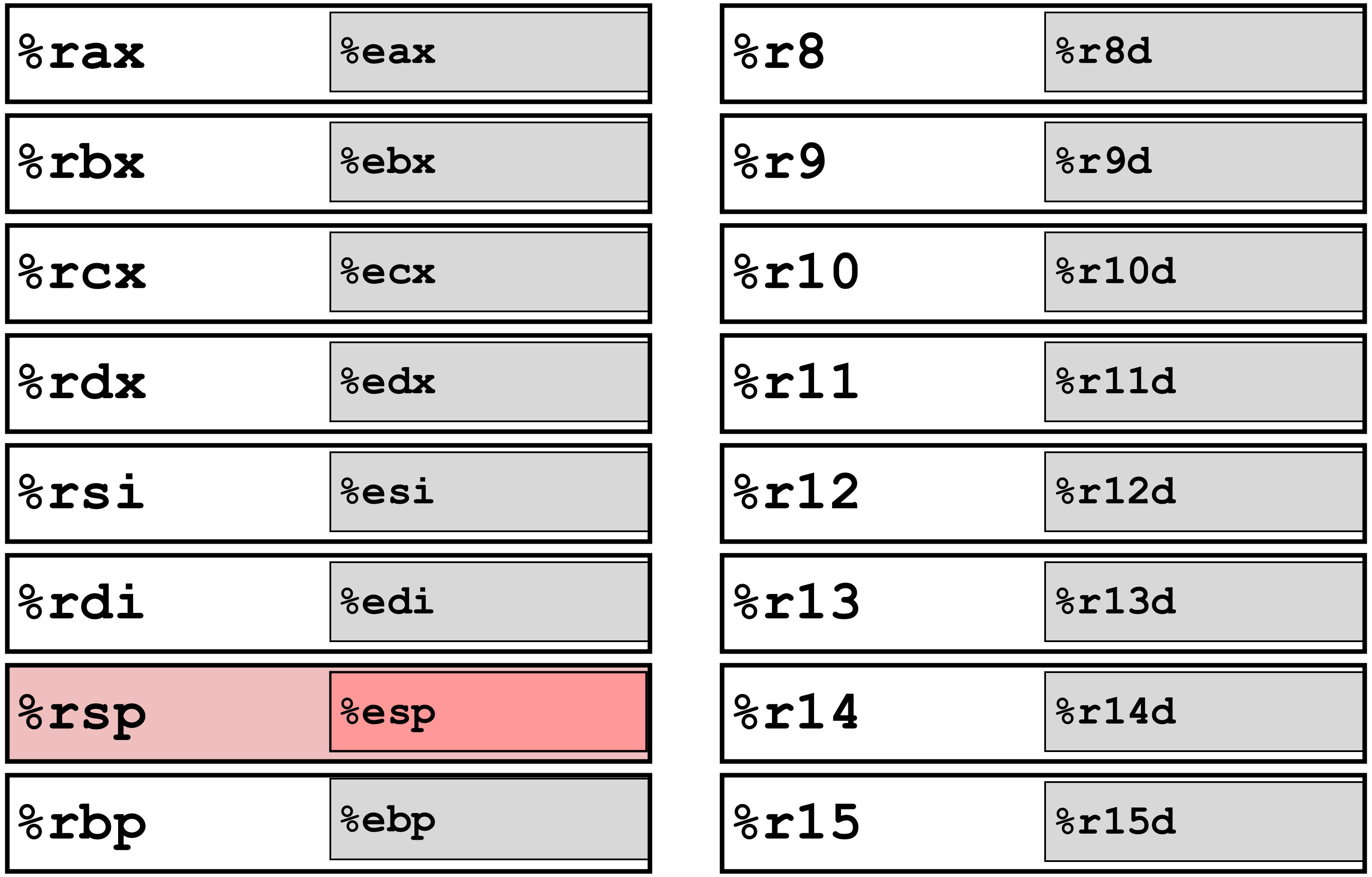
- Added Features
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores

x86 Integer Registers

- General purpose registers
  - On 32-bit architectures EAX, EBX, ECX, EDX, EDI, ESI, ESP, EBP
- The instruction pointer (IP)
  - Also referred to as program counter (PC)
  - EIP on 32-bit
- FLAGS register
  - Used for control flow operations, etc.
  - EFLAGS



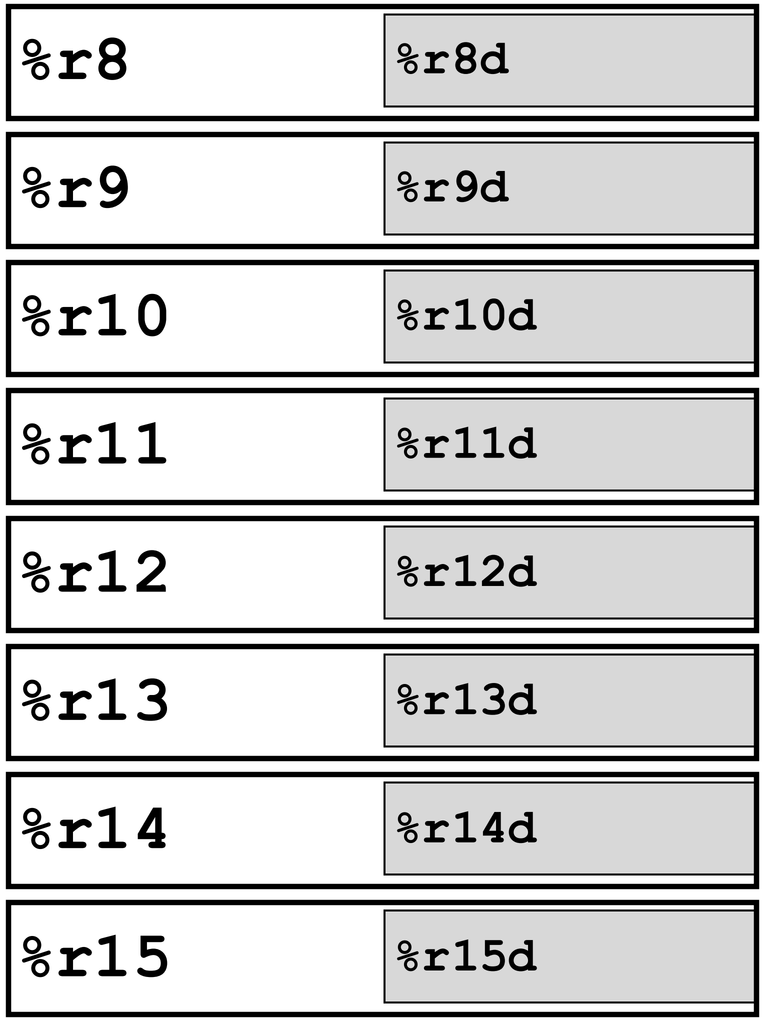
x86-64 Integer Registers



x86-64 Integer Registers

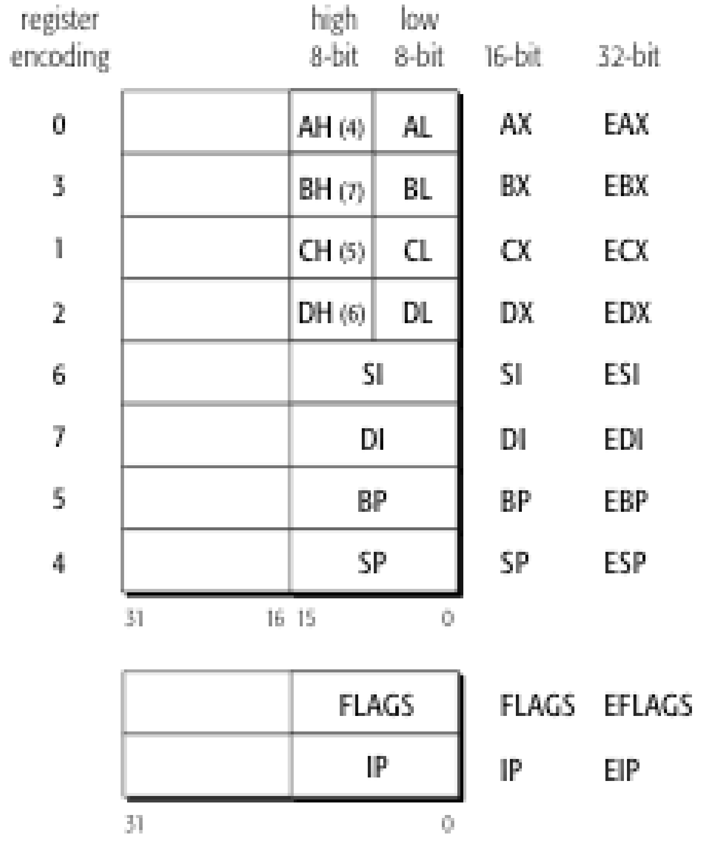
Can reference low-order bytes too

- d suffix for lower 32-bits (r8d)
- w suffix for lower 16-bits (r8w)
- b suffix for lower 8-bits (r8b)



Typical Register Uses

- EAX: accumulator
- EBX : Pointer to data
- ECX: Counter for string operations and loops
- EDX: I/O Operations
- EDI: Destination for string operations
- ESP: Stack pointer
- EBP: Frame pointer



Assembly Syntax

- Intel: OP dest, src
- AT&T: OP src, dest
- Unix systems prefer AT&T
  - We are going to use the same as the GNU assembler (gas syntax)

Assembly Instructions

- **pushq:** push quad word to stack
- **movq:** Move quad word
- **imull:** Signed multiply long
- **addl:** Add long

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    %edx, -28(%rbp)
movl    -20(%rbp), %eax
imull   -28(%rbp), %eax
movl    %eax, %edx
movl    -24(%rbp), %eax
addl    %edx, %eax
imull   -28(%rbp), %eax
```

Operand Sizes

Instructions include a suffix that indicates the size of the operand(s)

Register operands are prefixed with a **%**

pushq %rax

- Intel syntax does not include a suffix, size depends on the size of the operand
- If omitted size depends on arch

Register operands must match size
For example,
- quad → rax
- long → eax
- word → ax
- byte → ah or al

Memory Operands

- Parentheses indicate a memory operand
- Each memory address can be defined as: **Base+Index*Scale+Disp**
  - In AT&T syntax: disp(base, index, scale)
    - disp, index, and scale are optional

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    %edx, -28(%rbp)
movl    -20(%rbp), %eax
imull   -28(%rbp), %eax
movl    %eax, %edx
movl    -24(%rbp), %eax
addl    %edx, %eax
imull   -28(%rbp), %eax
```

Memory Addressing Modes

- Normal (B) Mem[Reg[R]]
  - Register R specifies memory base address
  - Pointer dereferencing in C

```
movq (%rcx),%rax
```

- Displacement D(B) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Memory Addressing Modes

- Most General Form

D(B,I,S) $\quad$ Mem[Reg[R_b]+S*Reg[R_i]+ D]

D:	Constant “displacement” 1, 2, or 4 bytes
Rb:	Base register
Ri:	Index register: Any, except for **`%rsp`**
S:	Scale: 1, 2, 4, or 8

```
movq 8(%rbp, %rax, 4),%rdx
```

Immediates

- Constants or immediates are defined using **$**
- In decimal, unless:
  - *0x* prefix is used → hexadecimal
  - *0* prefix is used → octal

```
addl    $1, %eax
```

Immediates can help you identify the syntax

Endianness

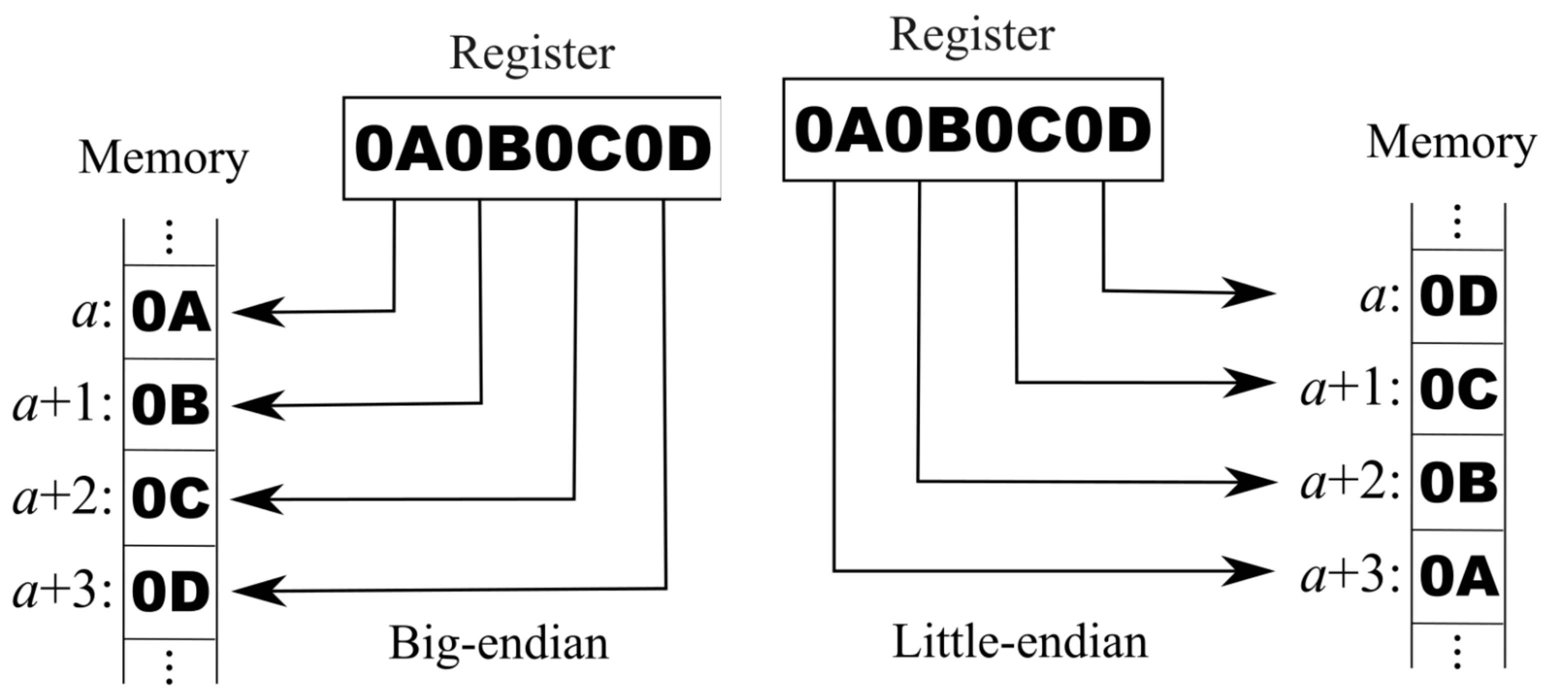
- Memory representation of multi-byte integers
- For example, the integer: 0x0A0B0C0Dh
- Big-endian ↔ highest order byte first

  0A 0B 0C 0D

- Little-endian ↔ lowest order byte first (X86)

  0D 0C 0B 0A

Endianness


Register
0A0B0C0D
Memory
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian
Register
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian

Load Effective Address

- **leaq Src, Dst**
  - Src is address mode expression
  - Set Dst to address denoted by expression
- Computing addresses without a memory reference
  - E.g., translation of p = &x[i];
- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8
- Example: **leaq (%rdi,%rdi,2), %rax**
- (Ab) used by the compiler to perform arithmetic operations without affecting EFLAGS

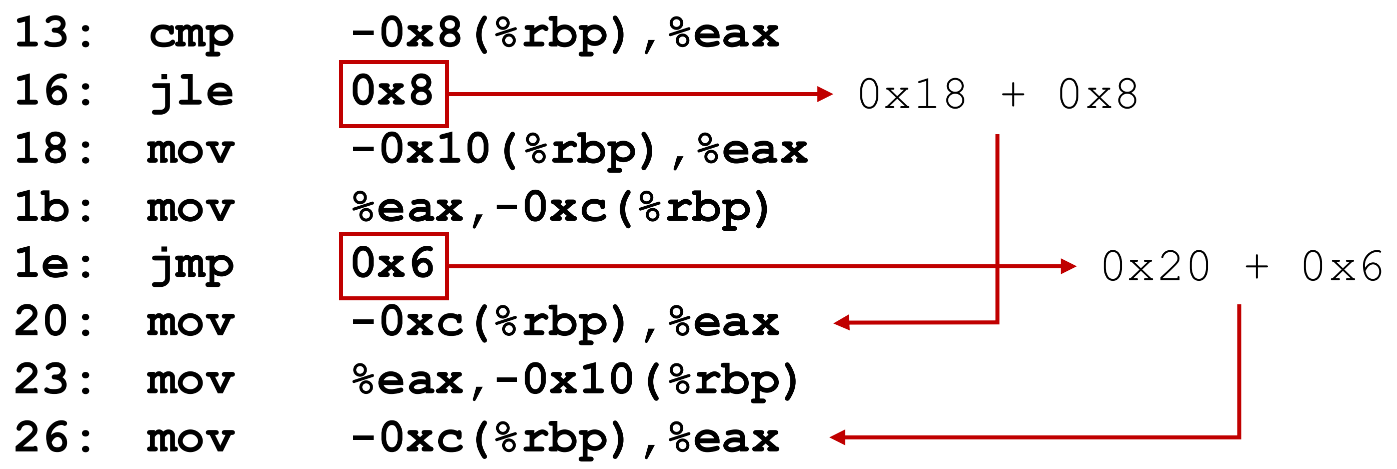
Control Flow Instructions (aka Branches)

- Jumps → Sets PC and redirects control flow
  - Relative (fixed) offset from next instruction (direct jump)
  - Absolute address in register or memory location (indirect jump, uses pointer)
- Conditional jumps → Sets PC and redirects control flow if condition code is true
  - Usually preceded by a comparison or test instruction
  - Always direct
  - Other architectures (e.g., ARM) allow other instructions to also execute conditionally
- Calls → Calls function at address
  - Relative (fixed) offset from next instruction (direct call)
  - Absolute address in register or memory location (indirect call, uses pointer)
  - Also pushes address of next instruction to stack (memory)
- Returns → Returns from current function
  - Pops address from stack
  - Sets PC to absolute address poped

$\Big\}$ We'll discuss these in detail later!

Jumps and Conditional Jumps

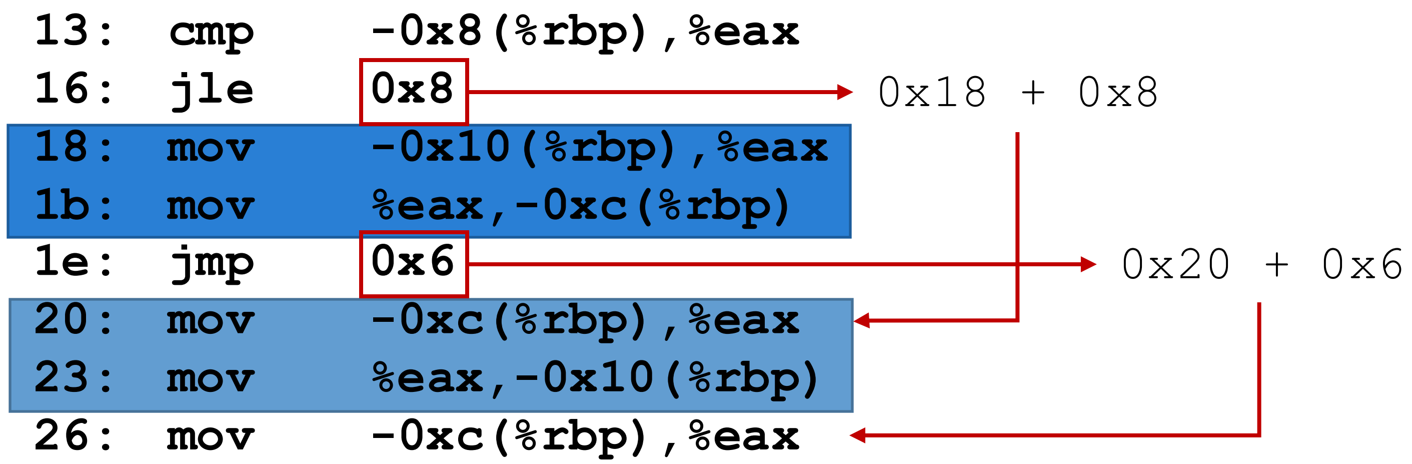
```
if (a > b) {
    c = d;
} else {
    d = c;
}
```



```
13:   cmp     -0x8(%rbp),%eax
16:   jle     0x8
18:   mov     -0x10(%rbp),%eax
1b:   mov     %eax,-0xc(%rbp)
1e:   jmp     0x6
20:   mov     -0xc(%rbp),%eax
23:   mov     %eax,-0x10(%rbp)
26:   mov     -0xc(%rbp),%eax
```

Jumps and Conditional Jumps

```
if (a > b) {
    c = d;
} else {
    d = c;
}
```



```
13:  cmp   -0x8(%rbp),%eax
16:  jle   0x8
18:  mov   -0x10(%rbp),%eax
1b:  mov   %eax,-0xc(%rbp)
1e:  jmp   0x6
20:  mov   -0xc(%rbp),%eax
23:  mov   %eax,-0x10(%rbp)
26:  mov   -0xc(%rbp),%eax
```

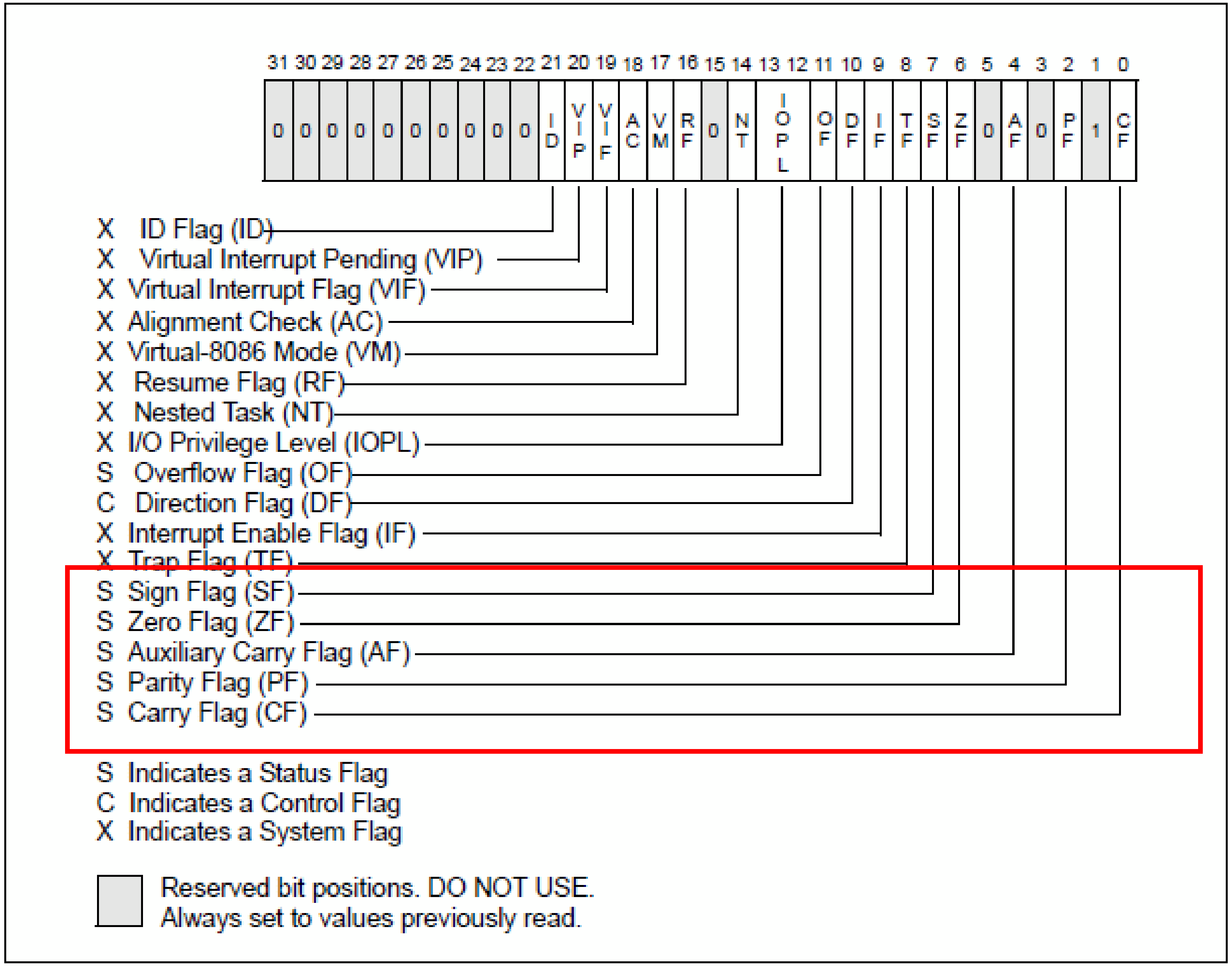
Jumps

- **jmp rel (`jmp 0x6`)**
  - **rel** can be 8, 16, or 32 bit immediate
  - Relative jump to next instruction: **target ← next instruction address + rel**
- **jmp *%reg (`jmp *%rax`)**
  - **reg** can be 16, 32 or 64-bit wide
  - Absolute jump to address in register
- **jmp *(%reg) (`jmp *(%rax`))**
  - **Memory** can be 16, 32 or 64-bit wide
  - Absolute jump to the address the register points to

Common Conditional Jumps

- **jcc rel (`jle 0x8`)**
  - **rel** can be 8, 16, or 32 bit immediate
  - If condition is met, relative jump to next instruction (**branch taken**): **target ← next instruction address + rel**
  - If condition is **not** met, execute next instruction (**branch fallthrough**)

jX	Description
`jmp`	**Unconditional**
`je`	**Equal / Zero**
`jne`	**Not Equal / Not Zero**
`jg`	**Greater (Signed)**
`jge`	**Greater or Equal (Signed)**
`jl`	**Less (Signed)**
`jle`	**Less or Equal (Signed)**


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

EFLAGS Register

Common Conditional Jumps

jX	Description	Condition
`jmp`	Unconditional	`1`
`je`	Equal / Zero	`ZF`
`jne`	Not Equal / Not Zero	`~ZF`
`jg`	Greater (Signed)	`~(SF^OF)&~ZF`
`jge`	Greater or Equal (Signed)	`~(SF^OF)`
`jl`	Less (Signed)	`(SF^OF)`
`jle`	Less or Equal (Signed)	`(SF^OF)|ZF`

Condition Codes Set by Compare

- Explicit Setting by Compare Instruction
  - **`cmpq`** *Src2*, *Src1*
  - **`cmpq b,a`** like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - **`testq`** *Src2*, *Src1*
    - **`testq b,a`** like computing **`a&b`** without setting destination
  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
  - **ZF set** when **`a&b == 0`**
  - **SF set** when **`a&b < 0`**

Absolute, Relative, and PC-Relative Addressing

- When using immediates (constants) jumps, conditional jumps, calls, etc. use relative addressing
  - Immediate is added to address of next address
- Memory operations always use absolute addressing!
- How can I load ADDR onto a register?

```
ADDR:  mov    -0x10(%ebp),%eax
```

Absolute, Relative, and PC-Relative Addressing

- When using immediates (constants) jumps, conditional jumps, calls, etc. use relative addressing
  - Immediate is added to address of next address
- Memory operations always use absolute addressing!
- How can I load ADDR onto a register?
  - x86 32-bit → using constant (code can't move)

```
        mov    ADDR, %edx
ADDR:   mov     -0x10(%ebp),%eax
```

Absolute, Relative, and PC-Relative Addressing

- When using immediates (constants) jumps, conditional jumps, calls, etc. use relative addressing
  - Immediate is added to address of next address

- Memory operations always use absolute addressing!

- How can I load ADDR onto a register?
  - x86 32-bit → using constant (code can't move)
  - x86-64 → RIP-relative addressing

What will be the value of ADDR?

```
        mov    ADDR, %edx
ADDR:   mov     -0x10(%ebp),%eax
```

```
        mov    ADDR(%rip), %edx
ADDR:   mov     -0x10(%ebp),%eax
```

Absolute, Relative, and PC-Relative Addressing

- When using immediates (constants) jumps, conditional jumps, calls, etc. use relative addressing
  - Immediate is added to address of next address

- Memory operations always use absolute addressing!

- How can I load ADDR onto a register?
  - x86 32-bit → using constant (code can't move)
  - x86-64 → RIP-relative addressing

What will be the value of ADDR?

```
        mov    ADDR, %edx
ADDR:   mov    -0x10(%ebp),%eax
```

```
        mov    $0(%rip), %edx
ADDR:   mov    -0x10(%ebp),%eax
```

Absolute, Relative, and PC-Relative Addressing

- When using immediates (constants) jumps, conditional jumps, calls, etc. use relative addressing
  - Immediate is added to address of next address
- Memory operations always use absolute addressing!
- How can I load ADDR onto a register?
  - x86 32-bit → using constant (code can't move) **OR using call/pop (movable code) aka GETPC**
  - x86-64 → RIP-relative addressing

```
        call    L1
L1:     pop     %edx
        add     $(ADDR-L1), %edx
ADDR:   mov     -0x10(%ebp),%eax
```

```
        mov     0(%rip), %edx
ADDR:   mov     -0x10(%ebp),%eax
```

Absolute, Relative, and PC-Relative Addressing

- When using immediates (constants) jumps, conditional jumps, calls, etc. use relative addressing
  - Immediate is added to address of next address
- Memory operations always use absolute addressing!
- How can I load ADDR onto a register?
  - x86 32-bit → using constant (code can't move) **OR using call/pop (movable code)**
  - x86-64 → RIP-relative addressing

```
        jmp    GETA
L1:     pop    %edx
        jmp    ADDR
GETA:   call   L1
ADDR:   mov    -0x10(%ebp),%eax
```

Alternative call/pop sequence

```
        mov    0(%rip), %edx
ADDR:   mov    -0x10(%ebp),%eax
```

Disassembling

- The process of recovering the assembly code of a binary
- Can be hard, specially for CISC variable-length instruction sets, like x86
- In this course we will gdb and objdump
  - gdb: while debugging use the disas command or the assembly layout
  - objdump –d: the utility attempts to disassemble a given binary