Modes, Registers and Addressing and Arithmetic Instructions

CS 217

Revisit IA32 General Registers

- 8 32-bit general-purpose registers (e.g. EAX)
- Each lower-half can be addressed as a 16-bit register (e.g. AX)
- Each 16-bit register can be addressed as two 8-bit registers (e.g AH and HL)

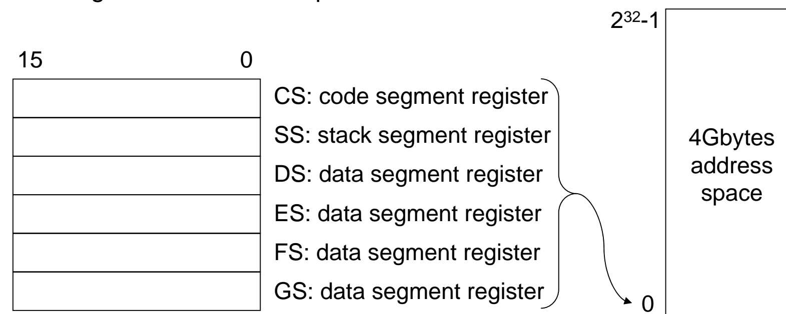
31 16	15 8	7 0		
	AH	AL	AX	EAX: Accumulator for operands, results
	BH	BL	BX	EBX: Pointer to data in the DS segment.
	CH	CL	CX	ECX: Counter for string, loop operations.
	DH	DL	DX	EDX: I/O pointer.
	SI			ESI: Pointer to DS data, string source
	DI			EDI: Pointer to ES data, string destination
	BP			EBP: Pointer to data on the stack
	SP			ESP: Stack pointer (in the SS segment)

EIP Register

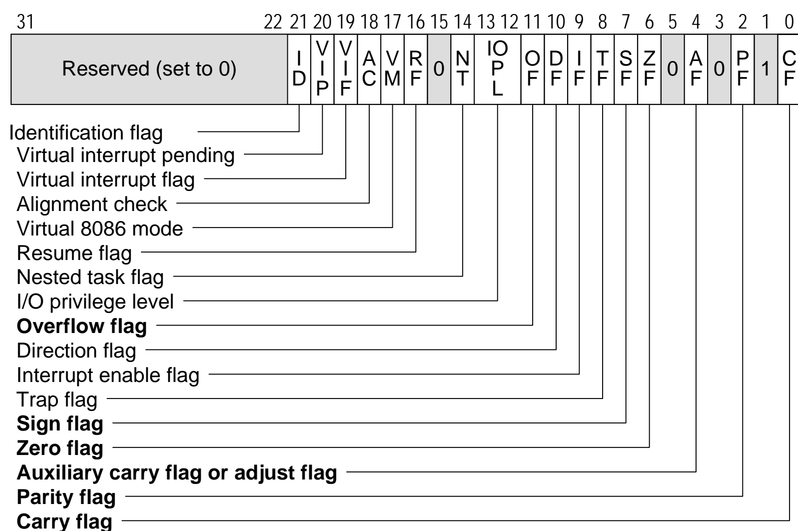
- Instruction Pointer or "Program Counter"
- Software change it by using
  - Unconditional jump
  - Conditional jump
  - Procedure call
  - Return

Segment Registers

- IA32 memory is divided into segments, pointed by segment registers
- Modern operating system and applications use the (unsegmented) memory mode: all the segment registers are loaded with the same segment selector so that all memory references a program makes are to a single linear-address space.



EFLAG Register



Other Registers

- 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

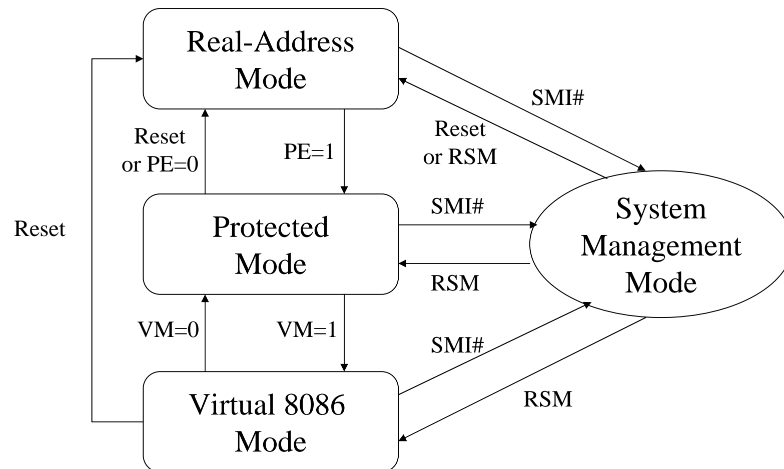
Three Addressing Models

- **Flat model**
  - **The modern way of memory addressing**
  - **All segment registers are loaded with 0**
- Segmented model
  - Segment registers are loaded differently
  - The goal is to increase protection
- Real-addressing model
  - Backward compatible with 8086
  - Each segment is 64Kbytes
  - Segments are laid out in 20-bit address space

Four Operating Modes

- **Real-address mode**
  - Let the processor address 1Mbytes of "real" memory (20-bit address).
  - Also called "unprotected" mode since operating system (such as DOS) code runs in the same mode as the user applications.
  - How: Power-up or a reset
  - Why: Backward compatible with early Intel processors such as 8086
  - Switch to protected mode: a single instruction
- **Protected mode**
  - Let the processor address 4GBytes of virtual memory (32-bit address) and will extend to 64-bit this year
  - Preferred mode for a modern operating system
  - Use virtual memory and provide protections.
- **System management mode**
  - For fast state snapshot and resumption (power management)
- **Virtual-8086 mode**
  - Allow the processor to execute 8086 code software in the protected, multi-tasking environment

IA32 Operating Mode Transition



PE is a flag in control register CR0

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

Types of Instructions

- Data transfer: move from source to destinatio
- Arithmetic: arithmetic on integer
- Floating point: x87 FPU move, arithmetic
- Logic: bitwise logic operations
- Control transfer: conditional and unconditional jumps, procedure calls
- String: move, compare, input and output
- Flag control: Control fields in EFLAGS
- Segment register: Load far pointers for segment registers
- SIMD
  - MMX: integer SIMD instructions
  - SSE: 32-bit and 64-bit floating point SIMD instructions
  - SSE2: 128-bit integer and float point SIMD instructions
- System
  - Load special registers and set control registers (including halt)

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.

Immediate Operands

- All arithmetic instructions allow immediate as source operands
- Example in gcc assembly

```
movl $10, %eax        # move 10 to EAX registe
```


Register Operands (subset)

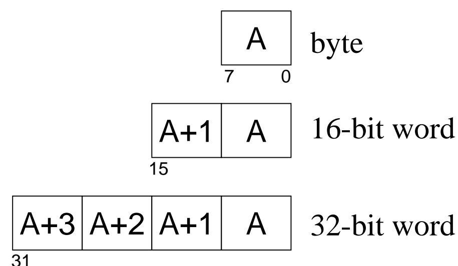
- General-purpose:
  - eax, ebx, ecx, edx, esi, edi, esp, ebp
  - ax, bx, cx, dx, si, di, sp, bp
  - ah, bh, ch, dh, al, bl, cl, dl
- Segment registers
  - cs, ds, ss, es, fs, gs
- Assembly syntax

```
addl $12, %ebx
```


Memory Operands

- Addressing memory
  - 8-bit is the smallest unit
  - 32-bit addresses (protected mode, will be extended to 64-bit later)
  - IA32 is little endian
- Examples

```
movb  $0x4a, %al
movw  $5, %ax
movl  $7, %eax
```



Effective Address

Offset = Base {eax, ebx, ecx, edx, esp, ebp, esi, edi} + Index {eax, ebx, ecx, edx, esp, ebp, esi, edi} * scale {1, 2, 3, 4} + displacement {None, 8-bit, 16-bit, 32-bit}

- 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	dest = source & dest
or{b,w,l} source, dest	dest = source | dest
xor{b,w,l} source, dest	dest = source ^ dest
not{b,w,l} dest	dest = ^dest
sal{b,w,l} source, dest (arithmetic)	dest = dest << source
sar{b,w,l} source, dest (arithmetic)	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 = ^dest
  - cmp{b,w,l} source1, source2 — source2 – source1
- Multiply
  - mul (unsigned) or imul (signed)

```
mull %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

Number Systems

- General form of a number in ***base*** ***b*** is

$$x = x_n b^n + x_{n-1} b^{n-1} + \ldots + x_1 b^1 + x_0 b^0 + x_{-1} b^{-1} + \ldots + x_{-m} b^{-m}$$

where x_i are the ***positional coefficients***

- Modern computers use binary arithmetic, i.e., base 2

$$\begin{aligned} 140_{10} &= 1 \times 10^2 + 4 \times 10^1 + 0 \times 10^0 \\ &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= 10001100_2 \\ &= 2 \times 8^2 + 1 \times 8^1 + 4 \times 8^0 = 214_8 \\ &= 8 \times 16^1 + \mathrm{C} \times 16^0 = 8\mathrm{C}_{16} \end{aligned}$$

Conversion

- To convert from decimal to binary, divide by 2 repeatedly, read remainders up.

```
2|140
 2|70    0  ^
 2|35    0  |
 2|17    1  |
  2|8    1  |
  2|4    0  |
  2|2    0  |
  2|1    0  |
    0    1  |

8|140
 8|17    4  ^
  8|2    1  |
    0    2  |
```

- Easier to convert to octal, then to binary

```
          8   C       hex
140 =  10001100       binary
        2  1  4       octal
```

Addition

- Addition in base ***b***

$$
\begin{aligned}
& x_n b^n + x_{n-1} b^{n-1} + x_{n-2} b^{n-2} + \ldots + x_1 b^1 + x_0 b^0 \\
& + y_n b^n + y_{n-1} b^{n-1} + y_{n-2} b^{n-2} + \ldots + y_1 b^1 + y_0 b^0 \\
\hline
z_{n+1} b^{n+1} + & z_n b^n + z_{n-1} b^{n-1} + z_{n-2} b^{n-2} + \ldots + z_1 b^1 + z_0 b^0
\end{aligned}
$$

where $S_i = x_i + y_i + C$, $C = S_{i-1}/b$, and $z_i = S_i \bmod b$ where $S_{-1} = 0$

- Addition in base 2:

```
  00101101
+ 10011001
  --------
  11000110
```

- the sum might have ***one*** more digit than the largest operand

Multiplication

- Multiplication in base 2: `00101101 * 10111001`

```
1 00101101
0  00000000
1   00101101
1    00101101
1     00101101
0      00000000
0       00000000
1        00101101
-------------------
  010000010000101
```

- The product has about as many digits as the two operands combined, i.e.

$$\log(a \times b) = \log(a) + \log(b)$$

Machine Arithmetic

- Computers usually have a fixed number of binary digits (“bits”), e.g., 32 bits
- For example, using 6 bits, numbered 0 to 5 from the right

  largest number $111111_2 = 63_{10} = 2^6 - 1$

  smallest number $000000_2 = 0$

- What is 50 + 20?

```
  110010
+ 010100
  ------
 1000110
```

- The highest bit doesn’t fit, so we get $000110_2 = 6_{10}$
- Spilling over the lefthand side is ***overflow***

Signed Magnitude

- ***Sign-magnitude*** notation:

  bit $n-1$ is the sign; 0 for +, 1 for -

  bits $n-2$ through 0 hold an unsigned number

  largest number $011111_2 = 31_{10} = 2^{6-1} - 1$

  smallest number $111111_2 = -31_{10} = -(2^{6-1} - 1)$

- Addition and subtraction are complicated when signs differ
- Sign-magnitude is rarely used

One's Complement

- ***One's-complement*** notation: *-k = (2ⁿ-1) - k = 11111...(n bits) - k*

  bit $n-1$ is the sign; bits $n-2$ through 0 hold an unsigned number

  bits $n-2$ through 0 hold ***complement*** of negative numbers

  largest number $011111_2 = 31_{10} = 2^{6-1}-1$

  smallest number $100000_2 = -31_{10} = -(2^{6-1}-1)$

  $$-k_{1C} = \hat{}\, k$$

- Addition and subtraction are easy, but there are ***2*** representations for 0

$$a - b = a + (r^n - 1 - b) + 1$$

$$a - b = a + b_{1C} + 1$$

Two's Complement

- ***Two's-complement*** notation: $-k = 2^n - k = \mathbf{(2^n-1) - k} + 1$

  bit $n-1$ is the sign; bits $n-2$ through 0 hold an unsigned number

  bits $n-2$ through 0 hold the ***complement*** of a negative number ***plus 1***

  largest number $011111_2 = 31_{10} = 2^{6-1}-1$

  smallest number $100000_2 = -32_{10} = -2^{6-1}$; note ***asymmetry***

  $$-k_{2C} = \hat{}\, k + 1$$

- To negate a 2's compl. number: first complement all the bits, then add 1

	start with	complement	increment	
+6	000110	111001	111010	-6
-6	111010	000101	000110	+6
+0	000000	111111	000000	-0
+1	000001	111110	111111	-1
+31	011111	100000	100001	-31
-31	100001	011110	011111	+31
-32	100000	011111	100000	-32

Two's Complement (cont)

- Adding 2's-complement numbers: ignore signs, add unsigned bit strings

```
  +20     010100          -20     101100
+ - 7   + 111001        + + 7   + 000111
  ———     ——————          ———     ——————
  +13     001101          -13     110011

  +20     010100          -20     101100
+ + 7   + 000111        + - 7   + 111001
  ———     ——————          ———     ——————
  +27     011011          -27     100101
```

$$a - b = a + (r^n - 1 - b) + 1$$

$$a - b = a + b_{2C}$$

- Signed overflow occurs if

  the carry ***into*** the sign bit differs from the carry ***out*** of the sign bit

```
  +20     010100          -20     101100
+ +17   + 010001        + -17   + 101111
  ———     ——————          ———     ——————
  -27     100101          +27     011011
```

- Same hardware for ***both*** unsigned and signed, but flags ***two*** conditions

  overflow signed overflow

  carry unsigned overflow

Sign Extension

- To convert from a small signed integer to a larger one, copy the sign bit

```
               +5            -5
4 bits       0101          1011
8 bits   00000101      11111011
```

- To convert a large signed integer to a smaller one: check trunced bits

```
               +5            -5
8 bits   00000101      11111011
4 bits       0101          1011      OK!

              +20           -20
8 bits   00010100      11101100
4 bits       0100          1100      Bad!
```

- Hardware does extension, but ***may not*** check for truncation; nor does C

```
short small = -50; long big = small;
printf("%d %d\n", small, big);          -50 -50

long big = 40000; short small = big;
printf("%d %d\n", small, big);          -25536 40000

char c = 255;
printf("%d\n", c);                      -1
```

Summary

- IA32 is a complex machine
  - Three memory models: flat, segmented, real-address
  - Four operating modes: real, protected, system mgmt, virtual 8086
  - Many kinds of instructions
- Things to remember
  - Five types of memory operands (immediate, base, base+displacement, index*scale + displacement, base+index*scale+displacement
  - Two's complement