IA32 Instructions
and
Assembler Directives

CS 217

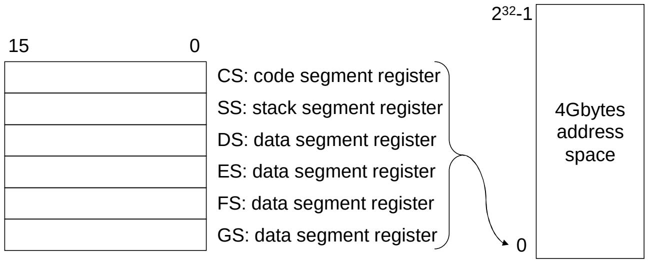
General-Purpose Registers

- Eight 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 AL)

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)

Segment Registers

- IA32 memory is divided into segments, pointed by segment registers
- Modern operating systems 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.

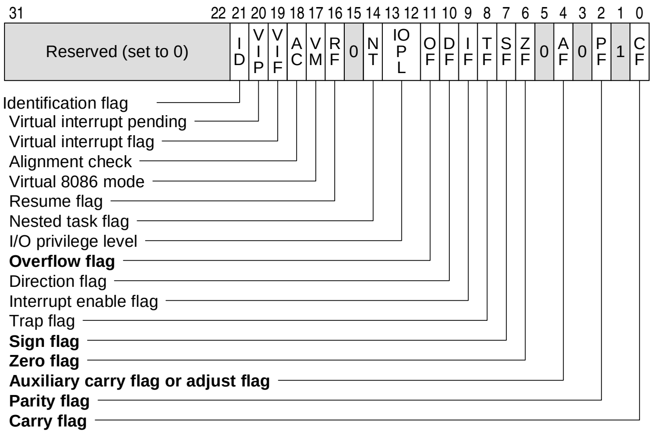


CS: code segment register
SS: stack segment register
DS: data segment register
ES: data segment register
FS: data segment register
GS: data segment register

EIP Register

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

EFLAG Register



The Six Flags

- **ZF (Zero Flag):** 1 if result is zero. 0 otherwise.
- **SF (Sign Flag):** Set equal to the most-significant bit of the result. Sign bit for a signed integer.
- **CF (Carry Flag):** Overflow condition for unsigned arithmetic.
- **OF (Overflow Flag):** Overflow condition for signed arithmetic.
- **PF (Parity Flag):** 1 if the least-significant byte of the result contains an even number of ones. 0 otherwise.
- **AF (Adjust Flag):** 1 if the arithmetic operation generated a carry/borrow out of bit 3 of the result. Used in Binary-Coded Decimal (BCD) arithmetic.

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

Instruction

- Opcode
  - What to do
- Source operands
  - Immediate: a constant embedded in the instruction itself
  - Register
  - Memory
  - I/O port
- Destination operand
  - Register
  - Memory
  - I/O port

GCC Assembly Examples

Syntax:

```
Opcode Source, Destination
```

```
movl  $5, %eax        # Move 32-bit "long-word"
  # Copy constant value 5 into register EAX
  # "Immediate" mode
```

```
movl  %eax, %ebx
  # Copy contents of register EAX into register EBX
```

Accessing Memory

Direct Addressing Mode

```
movl  500, %ecx
  # Copy long-word at memory address 500 into register ECX
  # "Little Endian:" byte 500 is least significant, byte 503 is most significant
```

Indirect Addressing Mode

```
movl  (%eax), %edx
  # Copy long-word at memory pointed to by register EAX into
  # register EDX
```

Address Computation

General form of memory addressing

DISPLACEMENT (%Base, %Index, SCALE)

Constant — Register — Register — {1, 2, 4 or 8}

Address = DISPLACEMENT + %Base + (%Index * SCALE)

- Not all four components are necessary

Examples

DISPLACEMENT (%Base, %Index, SCALE)

- DISPLACEMENT (Direct Addressing Mode)
  - `movl foo, %ebx`
- Base (Indirect Addressing Mode)
  - `movl (%eax), %ebx`
- DISPLACEMENT + Base (Base Pointer Addressing Mode)
  - `movl 20(%ebp), %ebx`
  - `movl foo(%eax), %ebx`
- DISPLACEMENT + (Index * SCALE) (Indexed Addressing Mode)
  - `movl foo(, %eax, 2), %ebx`
- DISPLACEMENT + Base + (Index * SCALE)
  - `movl foo(%eax, %ebx, 2), %ecx`

Types of Instructions

- Data transfer: copy data from source to destination
- Arithmetic: arithmetic on integers
- 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)

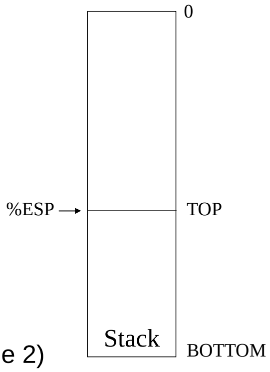
Data Transfer Instructions

- `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.

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)

```
mul %ebx          # edx:eax = eax * ebx
```

- Divide
  - div (unsigned) or idiv (signed)

```
idiv %ebx         # eax = edx:eax / ebx
                  # edx = edx:eax % ebx
```

- Many more in Intel manual (volume 2)
  - adc, sbb, decimal arithmetic instructions

Unsigned Integers: 4-bit Example

4-bit patterns	Unsigned Ints
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

```
  1001      9
 +1011    +11
 -----  -----
 10100     20
```

Overflow: when result does not fit in the given number of bits

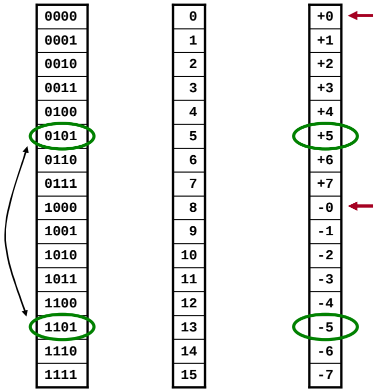
---

Signed Integers

Signed-Magnitude

- Designate the left-most bit as the “sign-bit”
- Interpret the rest of the bits as an unsigned number
- Example
  - 0101 is +5
  - 1101 is -5
- Addition and subtraction complicated when signs differ
- Two representations of 0: 0000 and 1000
- Rarely used in practice

Signed Integers



4-bit patterns	Unsigned Ints	Signed Magnitude
0000	0	+0
0001	1	+1
0010	2	+2
0011	3	+3
0100	4	+4
0101	5	+5
0110	6	+6
0111	7	+7
1000	8	-0
1001	9	-1
1010	10	-2
1011	11	-3
1100	12	-4
1101	13	-5
1110	14	-6
1111	15	-7

Signed Integers

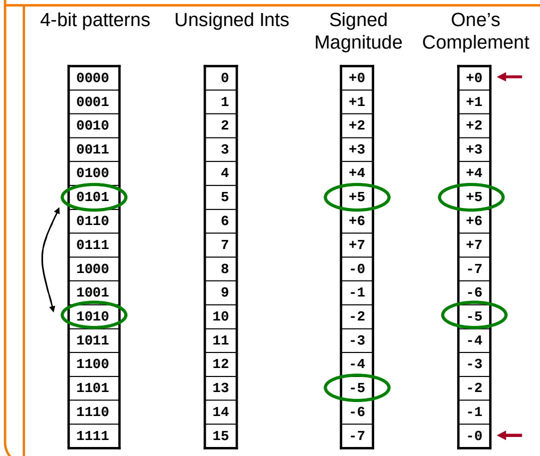
One’s Complement

- For positive integer k, -k has representation $(2^n - 1) - k$
- Example: 0101 is +5, 1010 is -5
- Negating k is same as flipping all the bits in k
- Left-most bit is the sign bit
- Addition and subtraction are easy
- For positive a and b,

$$a - b = a - b + 2^n - 1 + 1$$
$$= a + b_{1C} + 1$$

- Two representations of 0: 0000 and 1111

Signed Integers



4-bit patterns	Unsigned Ints	Signed Magnitude	One's Complement
0000	0	+0	+0
0001	1	+1	+1
0010	2	+2	+2
0011	3	+3	+3
0100	4	+4	+4
0101	5	+5	+5
0110	6	+6	+6
0111	7	+7	+7
1000	8	-0	-7
1001	9	-1	-6
1010	10	-2	-5
1011	11	-3	-4
1100	12	-4	-3
1101	13	-5	-2
1110	14	-6	-1
1111	15	-7	-0

Signed Integers

Two's Complement

- For positive integer k, -k has representation $2^n - k$
- Example: 0101 is +5, 1011 is -5
- Negating k is same as flipping all the bits in k and then adding 1
- Left-most bit is the sign bit
- Addition and subtraction are easy
- For positive a and b,

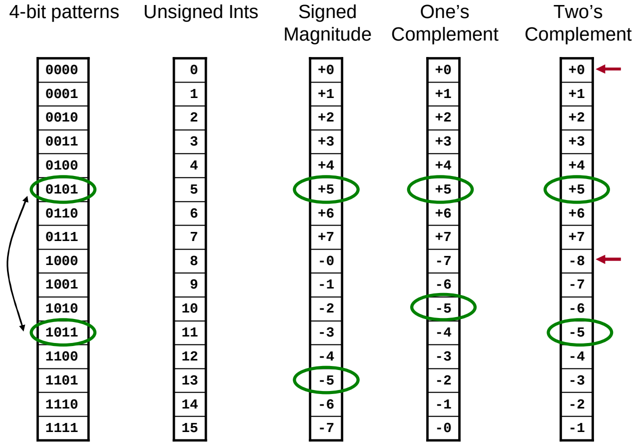
$$a - b = a - b + 2^n$$

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

- Same hardware used for adding signed and unsigned bit-strings

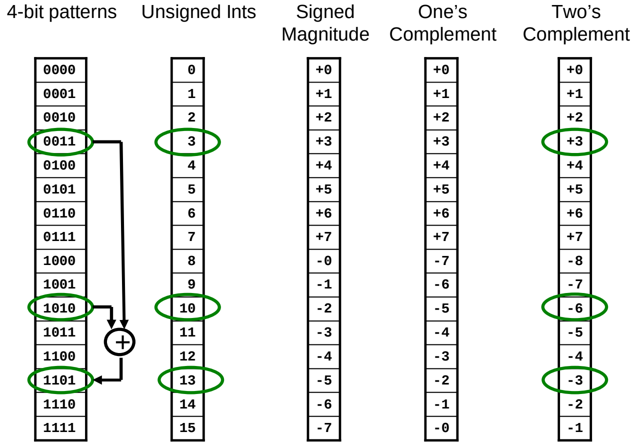
---

Signed Integers



4-bit patterns	Unsigned Ints	Signed Magnitude	One's Complement	Two's Complement
0000	0	+0	+0	+0
0001	1	+1	+1	+1
0010	2	+2	+2	+2
0011	3	+3	+3	+3
0100	4	+4	+4	+4
0101	5	+5	+5	+5
0110	6	+6	+6	+6
0111	7	+7	+7	+7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

Signed Integers



4-bit patterns	Unsigned Ints	Signed Magnitude	One's Complement	Two's Complement
0000	0	+0	+0	+0
0001	1	+1	+1	+1
0010	2	+2	+2	+2
0011	3	+3	+3	+3
0100	4	+4	+4	+4
0101	5	+5	+5	+5
0110	6	+6	+6	+6
0111	7	+7	+7	+7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

Branching Instructions

- Unconditional branch
  `jmp addr`
- Conditional branch
  - Perform arithmetic operation
  - Test flags in the EFLAGS register and jump

```
    cmpl %ebx, %eax   # eax - ebx
    je    L1
    ...               # ebx != eax
L1:
    ...               # ebx == eax
```

The Six Flags

- **ZF (Zero Flag):** 1 if result is zero. 0 otherwise.
- **SF (Sign Flag):** Set equal to the most-significant bit of the result. Sign bit for a signed integer.
- **CF (Carry Flag):** Overflow condition for unsigned arithmetic.
- **OF (Overflow Flag):** Overflow condition for signed arithmetic.
- **PF (Parity Flag):** 1 if the least-significant byte of the result contains an even number of ones. 0 otherwise.
- **AF (Adjust Flag):** 1 if the arithmetic operation generated a carry/borrow out of bit 3 of the result. Used in Binary-Coded Decimal (BCD) arithmetic.

Conditional Branch Instructions

- For both signed and unsigned integers

`je`	(ZF = 1)	Equal or zero
`jne`	(ZF = 0)	Not equal or not zero

- For signed integers

`jl`	(SF ^ OF = 1)	Less than
`jle`	((SF ^ OF) || ZF) = 1)	Less or equal
`jg`	((SF ^ OF) || ZF) = 0)	Greater than
`jge`	(SF ^ OF = 0)	Greater or equal

- For unsigned integers

`jb`	(CF = 1)	Below
`jbe`	(CF = 1 || ZF = 1)	Below or equal
`ja`	(CF = 0 && ZF = 0)	Above
`jae`	(CF = 0)	Above or equal

- For AF and PF conditions, FPU, MMX, SSE and SSE2
  - See the Intel manual (volume 1 and 2)

Branching Example: if-then-else

C program

```
if (a > b)


      c = a;




   else
      c = b;
```

Assembly program

```
      movl a, %eax
      cmpl b, %eax     # a-b
      jle  L1          # jump if a <= b

      movl a, %eax     # a > b branch
      movl %eax, c
      jmp     L2

  L1:                  # a <= b branch
      movl b, %eax
      movl %eax, c

  L2:                  # finish
```

Branching Example: for-loop

C program

```
for (i = 0; i < 100; i++) {
  ...
}
```

Assembly program

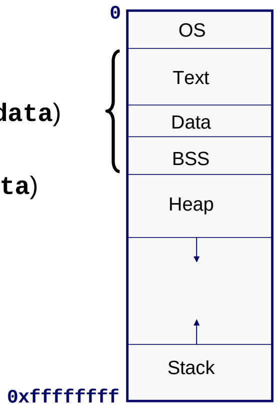
```
      movl $0, %edx          # i = 0
  loop_begin:
      cmpl $100, %edx        # i – 100
      jge loop_end           # end if i >= 100
      ...
      incl %edx              # i++
      jmp loop_begin
  loop_end:
```

Assembler Directives

- Identify code and data sections
- Allocate/initialize memory
- Make symbols externally visible or invisible

Identifying Sections

- Text (`.section .text`)
  - Contains code (instructions)
  - Contains the _start label
- Read-Only Data (`.section .rodata`)
  - Contains pre-initialized constants
- Read-Write Data (`.section .data`)
  - Contains pre-initialized variables
- BSS (`.section .bss`)
  - Contains zero-initialized variables



Initializing Data

```
      .section .data
p:    .byte 215          # one byte initialized to 215
q:    .word 150, 999     # two words initialized
r:    .long 1, 1000, 1000000  # three long-words
```

Can specify alignment constraints

```
      .align 4
s:    .long 555
```

Initializing ASCII Data

- Several ways for ASCII data

```
.byte  150,145,154,154,157,0  # a sequence of bytes

.ascii “hello”                # ascii without null char
.byte 0                       # add \0 to the end

.ascii “hello\0”

.asciz “hello”                # ASCII with \0

.string “hello”               # same as .asciz
```

Allocating Memory in BSS

- For global data
  `.comm symbol, nbytes [,desired-alignment]`
- For local data
  `.lcomm symbol, nbytes [,desired-alignment]`
- Example

```
.section .bss         # or just .bss
.equ   BUFSIZE 512    # define a constant
.lcomm BUF, BUFSIZE   # allocate 512 bytes
                      #     local memory for BUF
.comm x, 4, 4         # allocate 4 bytes for x
                      #     with 4-byte alignment
```

- BSS does not consume space in the object file

Making Symbols Externally Visible

- Default is local
- Specify globally visible
  `.globl symbol`
- Example: `int x = 1;`

```
   .section .data
   .globl x          # declare externally visible
   .align 4
x: .long  2
```

- Example: `foo(void){...}`

```
   .text
   .globl foo
foo:
   ...
   leave
   return
```

Summary

- Instructions manipulate registers and memory
  - Memory addressing modes
  - Unsigned and signed integers
- Branch instructions
  - The six flags in the EFLAGS register
  - Conditional branching
- Assembly language directives
  - Define sections
  - Allocate memory
  - Initialize values
  - Make labels externally visible