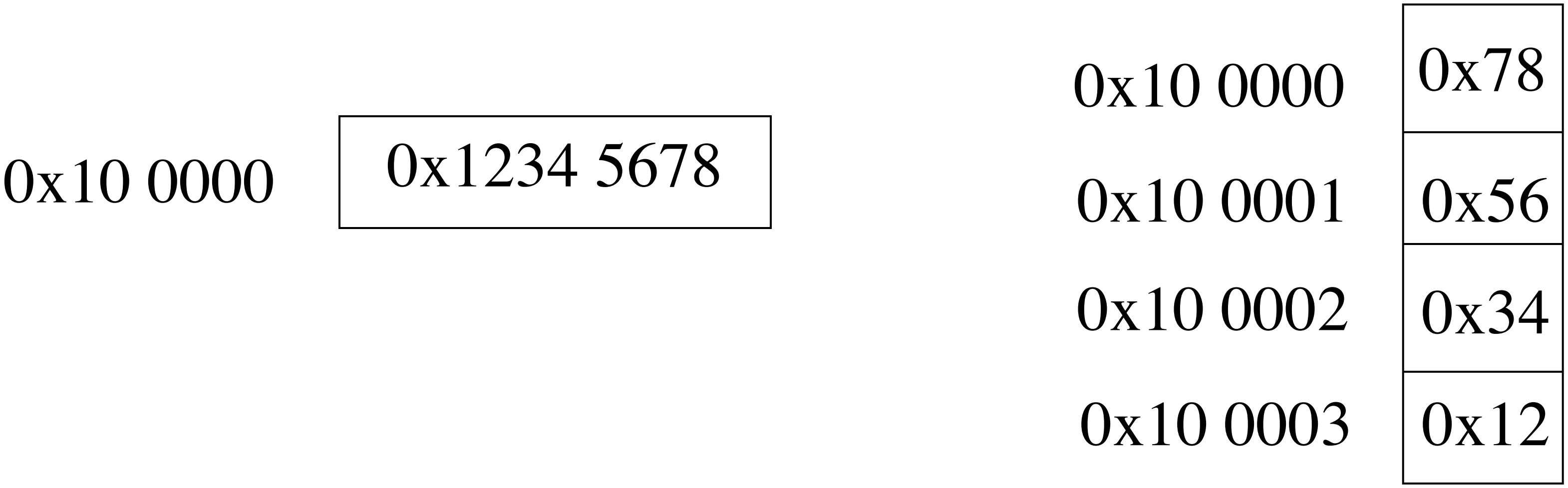
Chapter 11: x86 assembly language basics

PC (x86) architecture

- First 16-bit implementation: 8086 (1978)
- First 32-bit extension: 80386 (1985)
- Higher performance implementations:
  - 80486 (1989)
  - Pentium (1992)…, Pentium III, Pentium 4 (2001)
  - Core 2 Duo (2006), Core 2 Quad…
- *Not* a load/store architecture

View of memory

- 32-bit memory address space
- Little-endian:

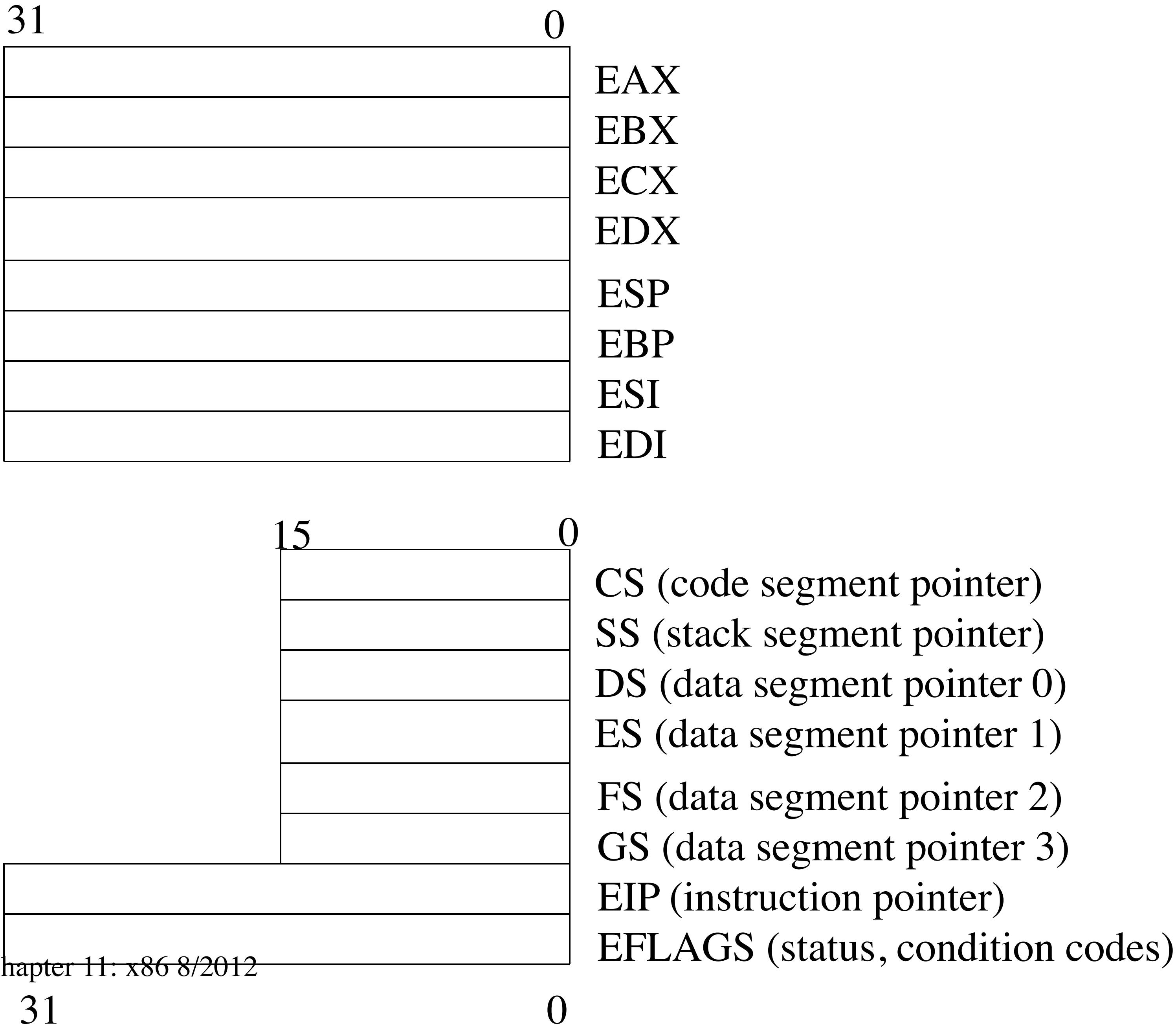


0x10 0000	0x1234 5678

0x10 0000	0x78
0x10 0001	0x56
0x10 0002	0x34
0x10 0003	0x12

View of registers

- 8 general purpose registers (32-bit wide)
  - EAX, EBX, ECX, EDX
  - ESP, EBP, ESI, EDI
- 6 segment registers (16-bit wide)
  - CS, SS, DS, ES, FS, GS
- Instruction pointer (EIP, same as PC)
- EFLAGS: condition codes (32-bit wide)



31 0

EAX

EBX

ECX

EDX

ESP

EBP

ESI

EDI

15 0

CS (code segment pointer)

SS (stack segment pointer)

DS (data segment pointer 0)

ES (data segment pointer 1)

FS (data segment pointer 2)

GS (data segment pointer 3)

EIP (instruction pointer)

EFLAGS (status, condition codes)

31 0

Assembly language syntax

- Registers marked with % (%esp, %eax etc)
- Constants marked with $ ($1, $0 etc)
- General format:
  - operation operand1 operand2 …
- *Rightmost* operand is destination operand

[gcc follows this syntax; other compilers/assemblers may have different conventions]

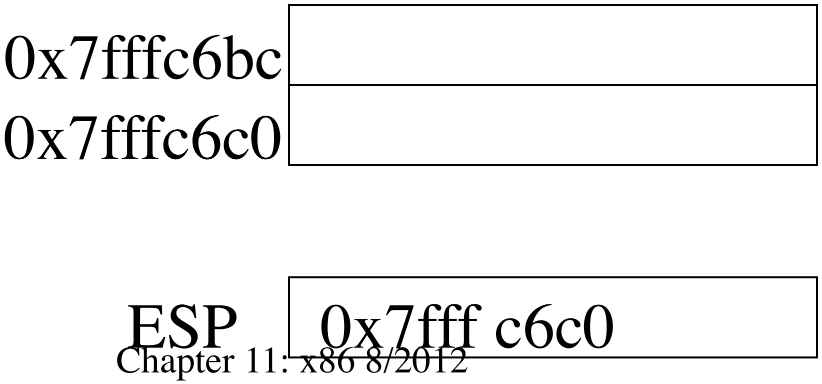
Data movement operations

- `movl operand1 operand2`
  - Move 32-bit integer from operand1 to operand2
  - Operand1 and operand2 can be
    - Constant (operand1 only): e.g. $1
    - Register: e.g. %eax
    - (register): e.g. (%esp)
    - Constant(register): e.g. 4(%esp)
  - But there are restrictions on what registers can be used to contain parts of addresses

Pushl/popl operation

- `pushl operand1`
  - Operand1 is pushed on the stack
  - Operand1 can have four options (same as movl)
- `popl operand1`: pop operand1 off the stack

Before: `pushl $3`

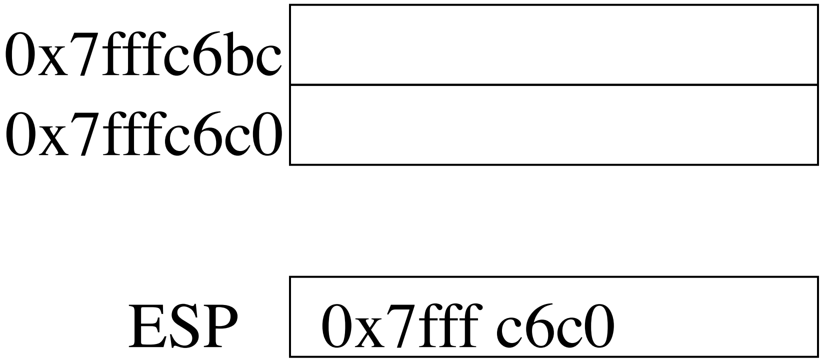


0x7fffc6bc

0x7fffc6c0

ESP 0x7fff c6c0

After: `pushl $3`



0x7fffc6bc

0x7fffc6c0

ESP 0x7fff c6c0

Arithmetic operations

- `addl operand1, operand2`
  - operand2 = operand2 + operand1
  - Operand1 can be one of four earlier options
  - Operand2 can be register or memory
- Other arithmetic operations:
  - subl
  - imull, idivl
  - incl, decl (operand1 only)

Compare instructions

- **EFLAGS register contains condition codes condition codes indicate status of previous computation**
  - CF: carry
  - OF: overflow
  - SF: sign
  - ZF: zero
- cmpl operand1, operand2
  - Compute (operand2 - operand1)
  - Condition codes changed based on (operand2 - operand1)

Compare examples

Execute: `cmpl %eax, %ebx`

- **Suppose %eax = 0x32, %ebx = 0x32**
  - **CF = OF = SF = 0, ZF = 1**
- **Suppose %eax = 0x31, %ebx = 0x32**
  - **CF = OF = SF = ZF = 0**
- **Suppose %eax = 0x32, %ebx = 0x31**
  - **CF = 1, OF = 0, SF = 1, ZF = 0**

[for signed operands, usually look at OF, SF, ZF]

Conditional branches

Conditional branches check conditional codes to decide whether to jump

Example: jump if greater than

- `jg label`
  - If (SF == 0 && ZF == 0) jump to label;
    else execute next instruction

Example of use:

```
cmpl %eax, %ebx      # if (%ebx > %eax)
jg target            #    goto target
```

More conditional branches

Some other conditional branches:

`jge`	`jump if >=`
`jl`	`jump if <`
`jle`	`jump if <=`
`jz`	`jump if == 0`
`jnz`	`jump if != 0`
`je`	`jump if ==`
`jne`	`jump if !=`
`jmp`	`jump always`

Call and return (near relative)

- `call label`
  - Return address (EIP+5) pushed on stack
  - EIP = address of label
- `ret`
  - EIP = contents popped off top of stack

Simple C program: pow.c

```
/* found on thecity.sfsu.edu in ~hsu/310/PROGS/pow.c */
#include <stdio.h>

int pow(int arg0, int arg1);

main()
{
  int res;

  res = pow(3,4);
  printf("%d\n",res);
}
```

Pow() code

```
int pow(int arg0, int arg1)
{
  int result,i;

  result = 1;
  for (i=1; i<=arg1;i++)
    result = result*arg0;
  return(result);
}
```

Compile to x86 assembly code

[copy pow.c to your home directory, of course]

Invoke gcc compiler on thecity.sfsu.edu:

```
gcc pow.c -S -O -fomit-frame-pointer
```

- **Generates pow.s (x86 assembly language file!)**

Excerpt from main

Pass arguments 3 and 4

Call pow

```
movl    $4, 4(%esp)
movl    $3, (%esp)
call    pow
```

Compiled pow()

Find label pow:

[Note: code may look different with different compilers…]

```
pow:
        pushl   %ebx
        movl    8(%esp), %ebx
        movl    12(%esp), %ecx
        movl    $1, %eax
        movl    $1, %edx
        cmpl    %ecx, %eax
        jg      .L9
```

pow() con’t…

```
.L7:
        imull   %ebx, %eax
        incl    %edx
        cmpl    %ecx, %edx
        jle     .L7
.L9:
        popl    %ebx
        ret
```

To see instruction format:

```
gcc pow.c -c -O -fomit-frame-pointer -g
objdump -d pow.o
```

```
00000021 <pow>:
  21:   53                       push   %ebx
  22:   8b 5c 24 08              mov    0x8(%esp),%ebx
  26:   8b 4c 24 0c              mov    0xc(%esp),%ecx
  2a:   b8 01 00 00 00           mov    $0x1,%eax
  2f:   ba 01 00 00 00           mov    $0x1,%edx
  34:   39 c8                    cmp    %ecx,%eax
  36:   7f 08                    jg     40 <pow+0x1f>
  38:   0f af c3                 imul   %ebx,%eax
  3b:   42                       inc    %edx
  3c:   39 ca                    cmp    %ecx,%edx
  3e:   7e f8                    jle    38 <pow+0x17>
  40:   5b                       pop    %ebx
  41:   c3                       ret
```