Software Development Laboratory

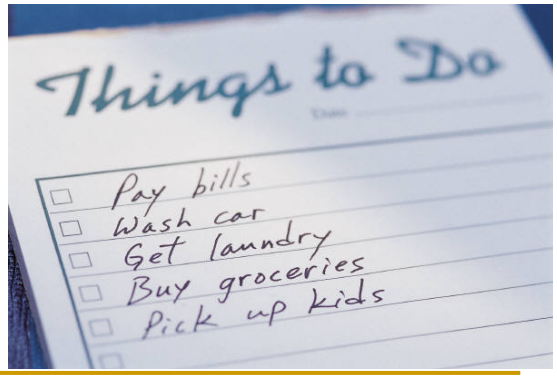
ソフトウェア開発演習
先導的ITスペシャリスト

x86 assembly lang.

Dept. of Computer Science
K. Gondow

Menu

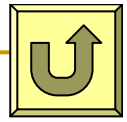
- Introduction to x86 assembly programming
- GNU assembler: as
- Inline assembly (for x86) in GCC
- x86 instructions (in AT&T style)



References

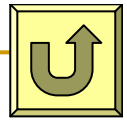
- Intel's manuals
  - Intel Architecture Software Developer’s Manual Volume, http://developer.intel.com/design/pentiumii/manuals/24319[012].htm
  - IA-32 インテルアーキテクチャ・ソフトウェア・デベロッパーズ・マニュア, http://www.intel.com/jp/developer/download/index.htm#ia32
- The Art of Assembly Language, ISBN-13: 978-1886411975,
  - http://webster.cs.ucr.edu/
- Computer Systems: A Programmer's Perspective (CS:APP) , ISBN-13: 978-0130340740, 2002
- x86 Assembly Language FAQ,
  - http://webster.cs.ucr.edu/Articles/X86FAQ/
- info as, info ld, info gcc, info binutils, info bfd

References (cont'd)

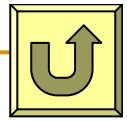


- MS-PE: Microsoft Portable Executable and Common Object File Format Specification
  - http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
- ELF: Executable and Linkable Format (ELF) Specification, version 1.2
  - http://www.x86.org/ftp/manuals/tools/elf.pdf
- SYSV-ABI-i386: System V Application Binary Interface, Intel386 Architecture Processor Suppliment, ed. 4, 1997,
  - http://www.caldera.com/developers/devspecs/abi386-4.pdf
- GCC-Inline-Assembly-HOWTO
  - http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
- OS FAQ Wiki
  - http://www.osdev.org/osfaq2/index.php/InlineAssembly

References (only in Japanese)



- プログラミングの力を生み出す本—インテルCPUのGNUユーザへ（改訂2版），ISBN-13: 978-4274132070
- はじめて読む8086—16ビット・コンピュータをやさしく語る，ISBN-13: 978-4871482455
- はじめて読む486—32ビットコンピュータをやさしく語る, ISBN-13: 978-4756102133
- x86アセンブラ入門—PC/ATなどで使われている80x86のアセンブラを習得, ISBN-13: 978-4789833424
- 高級言語プログラマのためのアセンブラ入門，ISBN-13: 978-4797332810



Why it is not that simple to learn assembly languages?

- Complex instruction set, especially of x86.
  - Non-orthogonal non-privilege instructions.
  - Complex privilege instructions (e.g., lidt, lgdt).
  - Many extended instructions (e.g., rdtsc, MMX, SSE).
- Notation of assembler directives (or pseudo opcodes).
- Notation of inline assembly.
- Application binary interface (ABI), including calling convention.
- Binary tools (e.g., GNU objdump)
- Hardware programming interface.
  - Just knowing in/out instructions is not enough.

Compile example (1)

- Let's see the assembly code emitted for fact.c

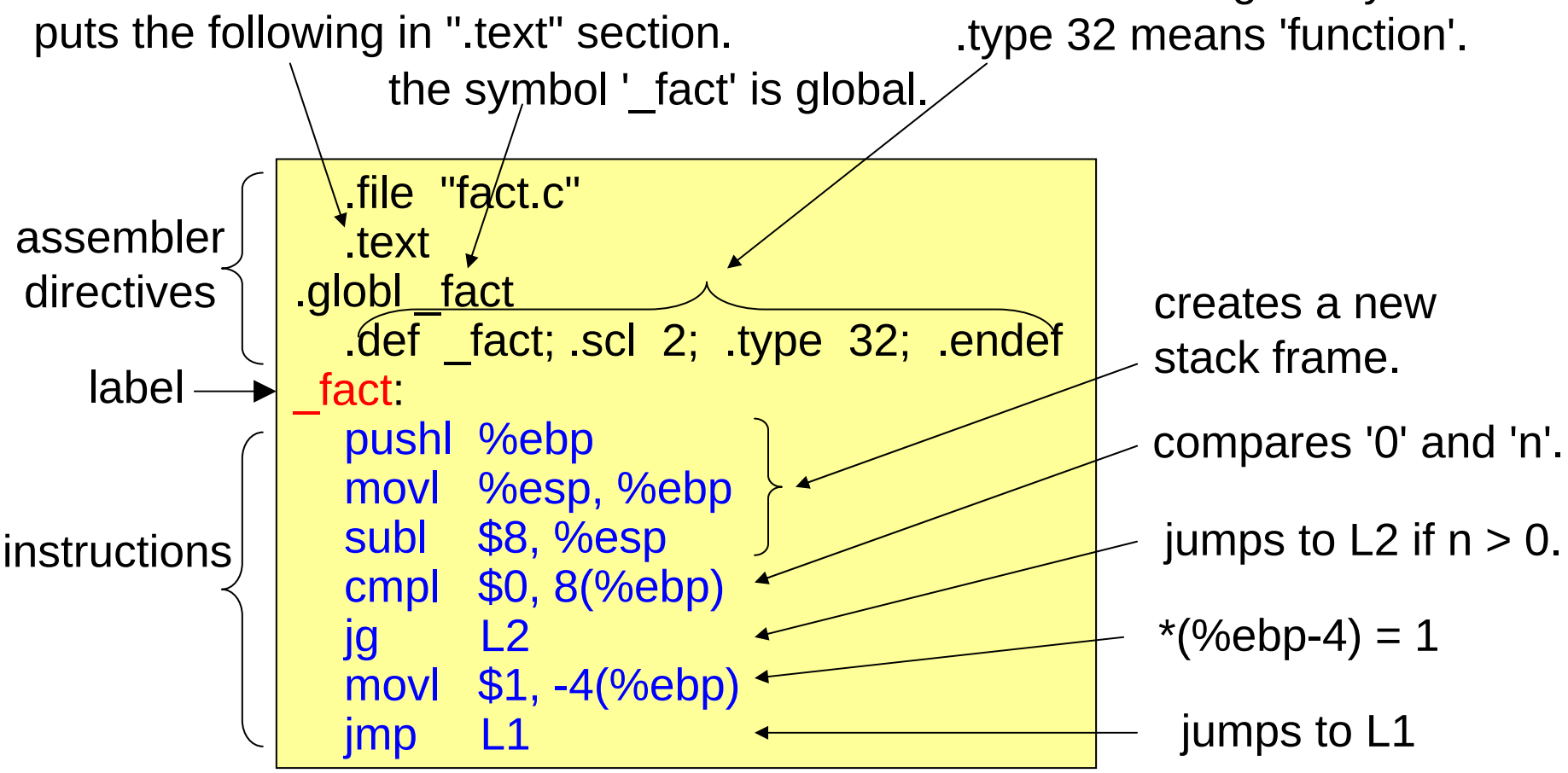
fact.c

```
int fact (int n) {
    if (n <= 0)
        return 1;
    else
        return n * fact (n - 1);
}
```

```
% gcc -v
gcc version 3.4.4 (cygming special)
%  uname -a
CYGWIN_NT-5.1  1.5.21(0.156/4/2)
i686 Cygwin
% gcc -S foo.c
% ls fact*
fact.c  fact.s
%
```

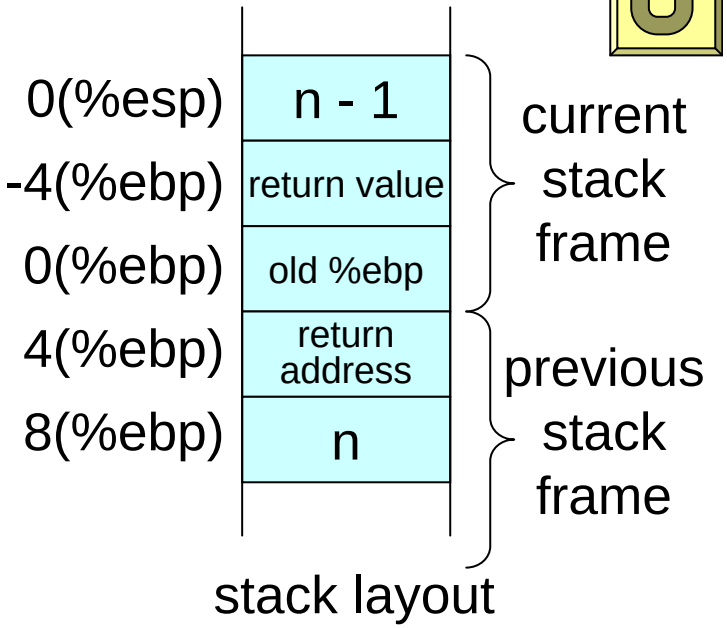
emitted assembly code

Compile example (2)



. (dot) means an assembler directive.
$ means an immediate value.
% means a register.
8(%ebp) means *(%ebp+8) in C.

Compile example (3)




```
L2:
    movl   8(%ebp), %eax
    decl   %eax
    movl   %eax, (%esp)
    call   _fact
    imull  8(%ebp), %eax
    movl   %eax, -4(%ebp)
L1:
    movl   -4(%ebp), %eax
    leave
    ret
```

%eax = n
%eax --
*%esp = %eax
call the function '_fact'

%eax = n * (return value);
*(%ebp - 4) = %eax

pop the stack frame

%eax = *(%ebp - 4)
%ebp = %esp; pop %ebp
return from the function '_fact'

Disassembler (objdump -d)

```
% gcc -c fact.c
% objdump -d fact.o
```

```
fact.o:     file format pe-i386
Disassembly of section .text:
00000000 <_fact>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 7d 08 00             cmpl   $0x0,0x8(%ebp)
   a:   7f 09                   jg     15 <_fact+0x15>
   c:   c7 45 fc 01 00 00 00    movl   $0x1,0xfffffffc(%ebp)
  13:   eb 13                   jmp    28 <_fact+0x28>
  15:   8b 45 08                mov    0x8(%ebp),%eax
  18:   48                      dec    %eax
  19:   89 04 24                mov    %eax,(%esp)
  1c:   e8 df ff ff ff          call   0 <_fact>
  21:   0f af 45 08             imul   0x8(%ebp),%eax
  25:   89 45 fc                mov    %eax,0xfffffffc(%ebp)
  28:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
  2b:   c9                      leave
  2c:   c3                      ret
  2d:   90                      nop
```

-4

AT&T style vs., Intel style

- Two different syntax for x86 assemblers.
  - GNU 'as' uses the AT&T-style syntax by default.

AT&T style

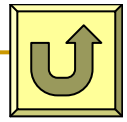
```
_fact:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
```

Intel style

```
_fact:
    push    ebp
    mov     ebp, esp
    sub     esp,   8
```

- Almost all x86 documents use Intel-style syntax.
- For details, see later and 8.12.2 of 'info as'.

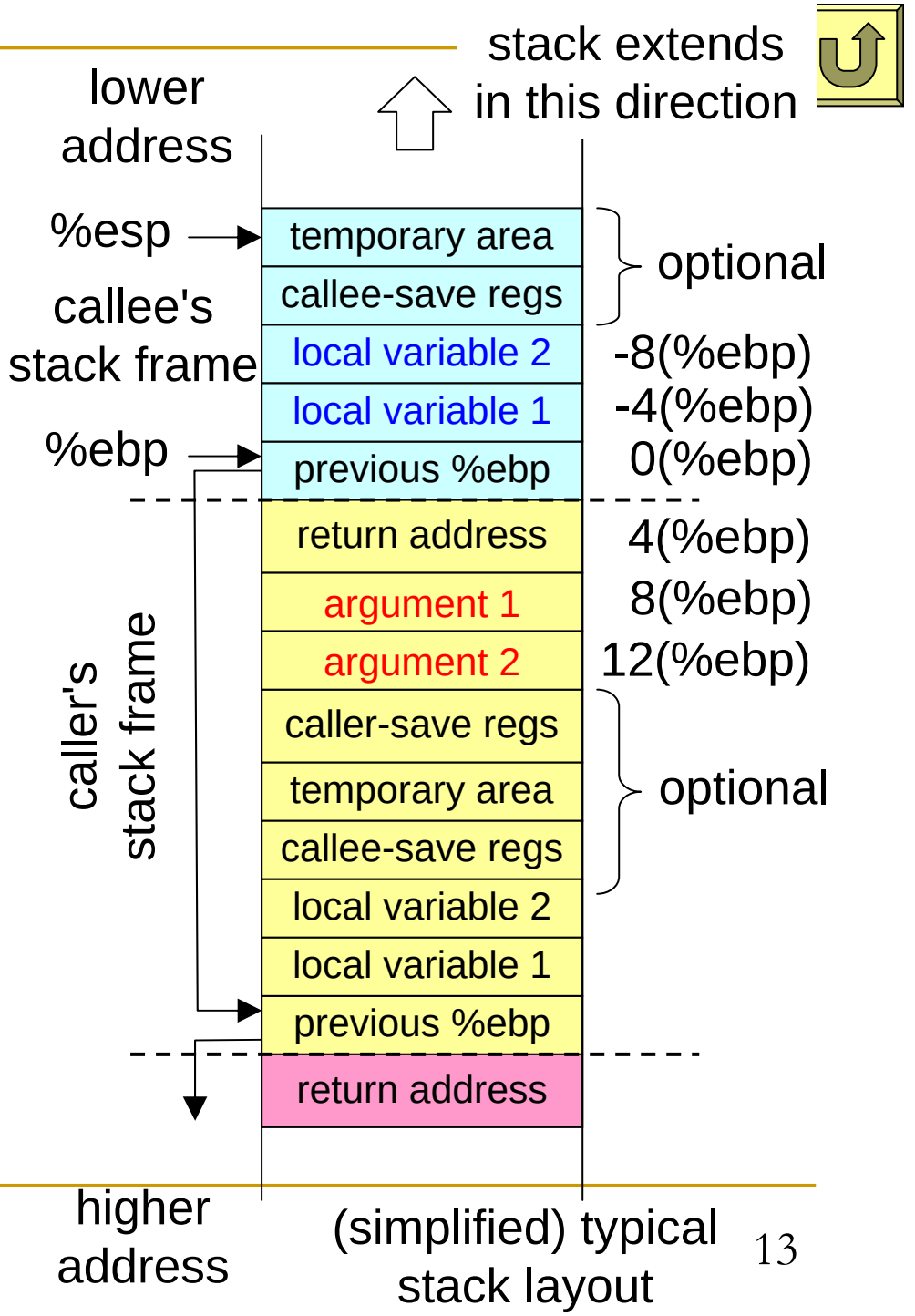
Calling convention



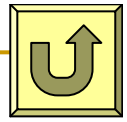
- Defines a standardized interface between caller and callee in machine-instruction level (i.e., in ABI):
  - Parameter passing.
  - Stack frame layout.
    - including whether the caller or callee unwinds the stack on return.
  - Register usage:
    - e.g., caller-/callee-save registers, parameters are pushed on the stack or placed in the registers
- Different platforms use different calling conventions.
  - Mismatch in calling convention causes a program to crash!
  - In GCC, you can change the calling convention by specifying the attributes like 'cdecl', 'stdcall', 'fastcall', etc.
  - SYSV-ABI-i386 includes a well-documented calling convention.

Stack layout

- Arguments are pushed on the stack in reverse order.
- An argument's size is increased if necessary, possibly with tail padding.
- Register usage:
  - %esp: stack pointer
  - %ebp: base pointer
  - %eax: return value
    - if its size <= 4 bytes
  - %ebx: GOT base register for PIC code.



caller-, callee-save registers



- Most registers are global, so the caller or callee needs to save/restore the register values.
  - caller-save registers: the caller is responsible for this.
  - callee-save registers: the callee is responsible for this.
- SYSV-ABI-i386 states:
  - caller-save registers: %eax, %ecx, %edx
  - callee-save registers: %ebp, %ebx, %edi, %esi, %esp

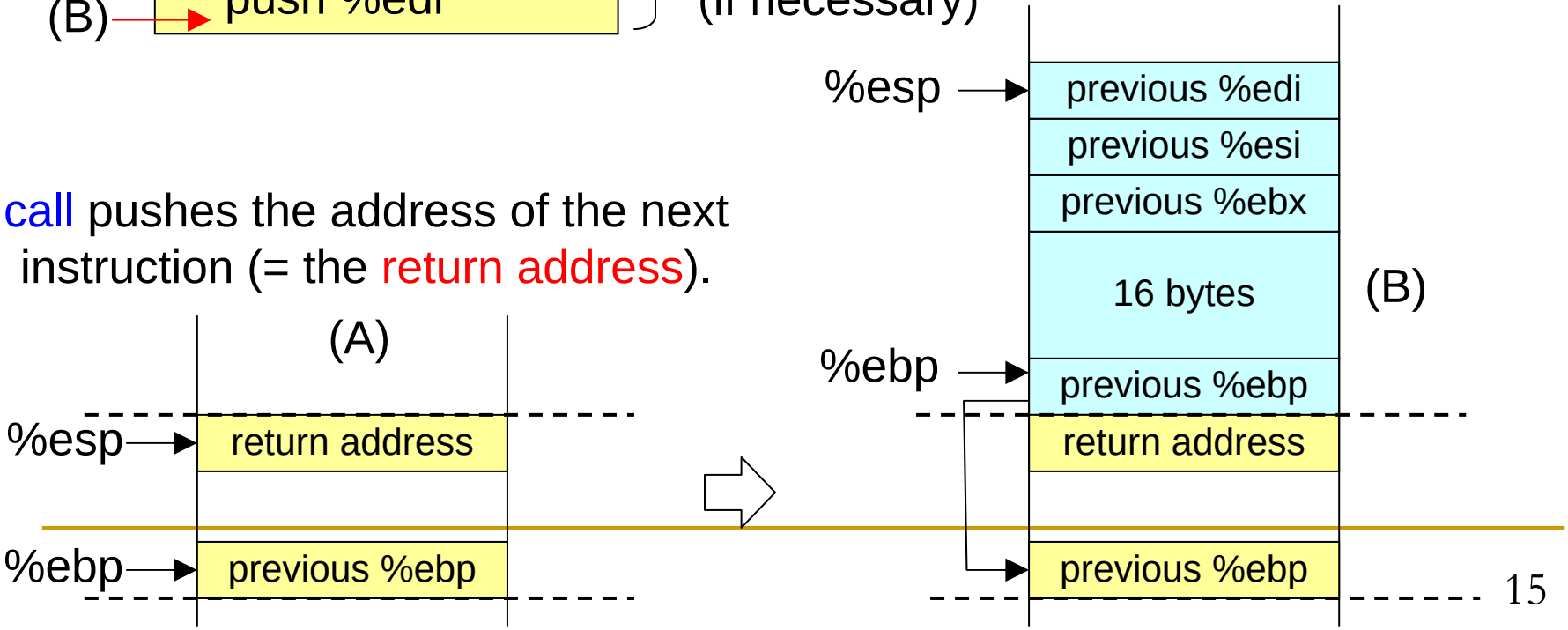
Typical function prologue

```
(A) prologue:
      push %ebp
      movl %esp, %ebp
      subl  $16,    %esp
      push %ebx
      push %esi
      push %edi
(B)
```

save frame pointer
set new frame pointer
allocate 16-byte stack frame probably for local variables

save callee-save registers (if necessary)

call pushes the address of the next instruction (= the return address).



Typical function epilogue

```
    movl  ???,  %eax
epilogue:
    popl  %edi
    pop   %esi
    pop   %ebx
    leave
    ret
```

(A) → epilogue:

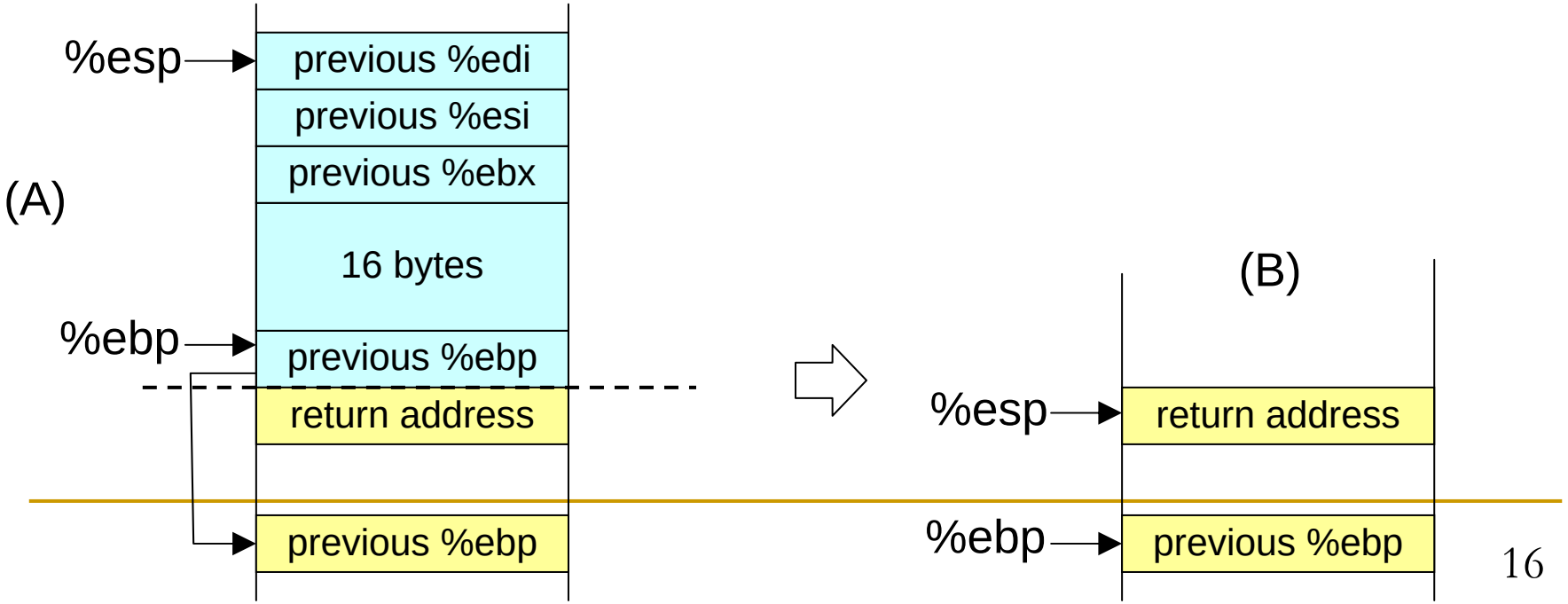
(B) → leave

set up the return value

restore callee-save registers (if necessary)

= movl %ebp, %esp; popl %ebp

return from the function



Calling assembly code from C

sub.s

```
    .text
.globl _sub
    .def  _sub; .scl  2;  .type  32;  .endef
_sub:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %eax
    movl   12(%ebp), %edx
    subl   %edx, %eax
    leave
    ret
```

main.c

```
#include <stdio.h>
int main (void) {
    printf ("%d\n", sub (23, 7));
}
```

```
% gcc main.c sub.s
% ./a.exe
16
%
```

Calling C from assembly code

main.s

```
    .text
.globl _main
    .def  _main; .scl  2;  .type  32;  .endef
_main:
    pushl  %ebp
    movl   %esp, %ebp
    pushl  $7
    pushl  $23
    call   _sub
    addl   $8, %esp
    leave
    ret
```

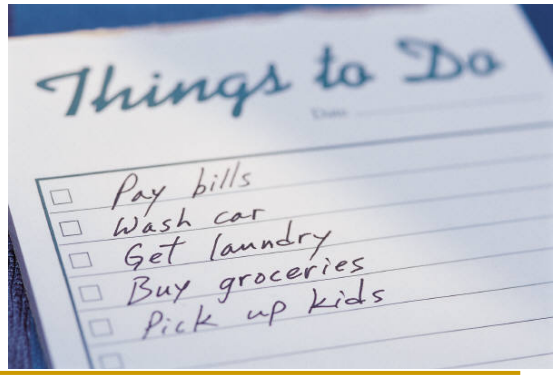
sub.c

```
int sub (int a, int b) {
    return a - b;
}
```

```
% gcc main.s sub.c
% ./a.exe
% echo $status
16
%
```

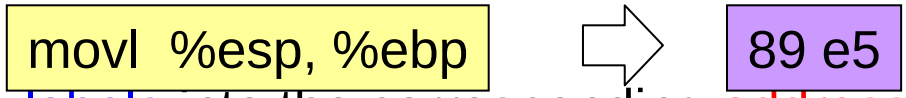
Menu

- Introduction to x86 assembly programming
- GNU assembler: as
- Inline assembly (for x86) in GCC
- x86 instructions (in AT&T style)

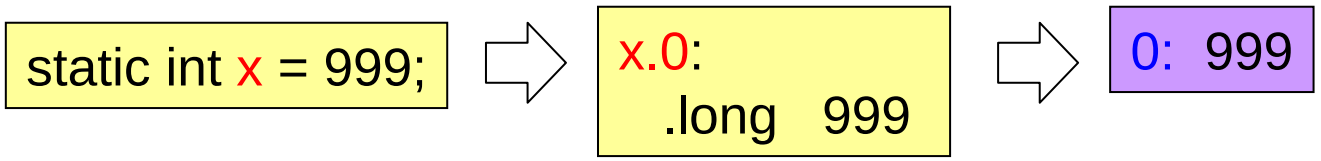


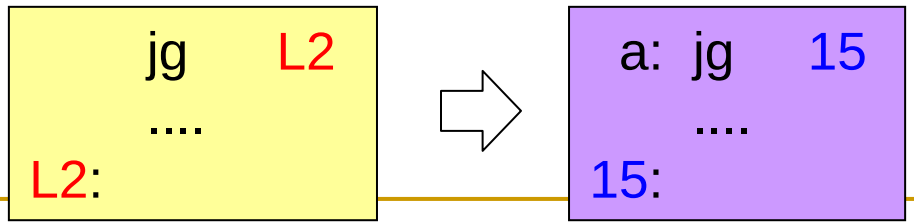
What the assembler 'as' does.(1)

- Translates mnemonic instructions into a binary representation of the corresponding machine instructions.
  - E.g.,



- Translates labels into the corresponding addresses.
  - E.g.,





What the assembler 'as' does.(2)

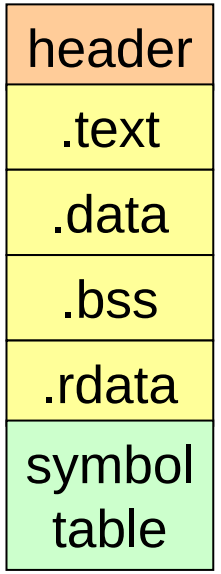
- Constructs the symbol table.

% nm foo.o

```
00000000 T _main
         U _printf
00000000 D _x
```

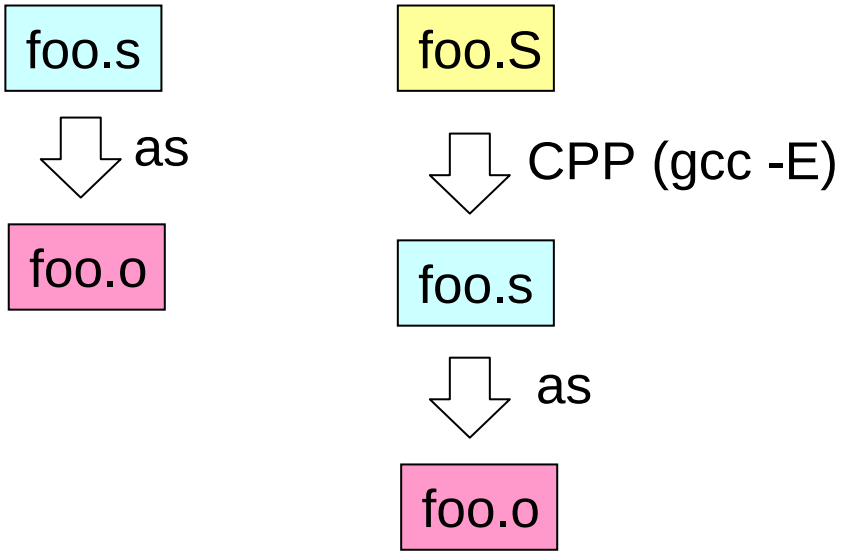
- Outputs all of the above to a specific binary format.
  - E.g., COFF format, ELF format.

```
foo.o:     file format pe-i386
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000050  00000000  00000000  000000b4  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000010  00000000  00000000  00000104  2**4
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000000  2**4
                  ALLOC
  3 .rdata        00000010  00000000  00000000  00000114  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, DATA ...
```

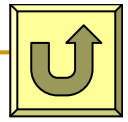


File suffixes: .s and .S

- .s (lowercase)
  - Assembled by 'as'.
- .S (uppercase)
  - First preprocessed by CPP, then assembled by 'as'.
  - CPP directives (e.g., #define), C-style comments, and macros can be used.



Comments in 'as'



- Delimited comments.
  - Cannot be nested.

```
/* this is a delimited comment. */
```

- Single-line comments:
  - Vary depending on the platforms.

```
# this is a single-line comment on x86.
```

```
!  this is a single-line comment on SPARC.
```

Statements

- A statement is terminated by newline or ';' (semicolon).
- Multiple statements can be placed on a single line if statements are delimited by ';' (semicolon).

```
pushl  %ebp
movl   %esp, %ebp
subl   $8, %esp
```

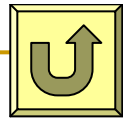
```
pushl  %ebp;
movl   %esp, %ebp;
subl   $8, %esp;
```

All are OK.

```
pushl  %ebp; movl   %esp, %ebp; subl   $8, %esp
```

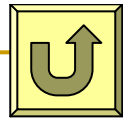
statement

Constants



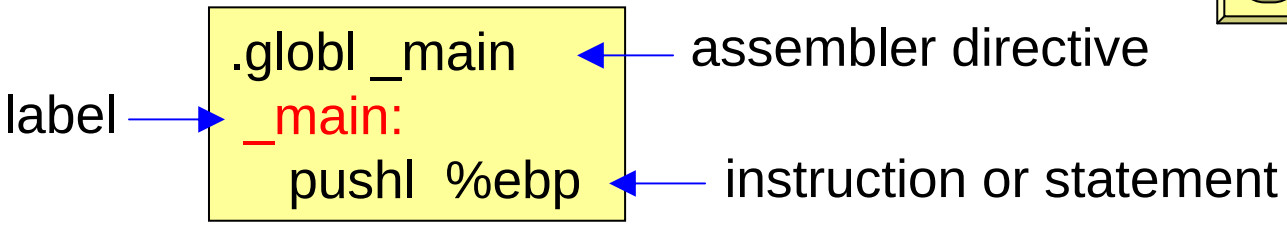
- Similar to C syntax.
- Numbers:
  - E.g., 74, 0112, 0x4a, 0b01001010 # All the same value
- Characters:
  - E.g., 'a, '\\, '\b, '\n, '\112, '\x4a
  - GNU 'as' also seems to accept C-style char like 'a'.
- Strings:
  - E.g., "Hello, world\n"

Symbols



- A symbol (or identifier)
  - [$_.a-zA-Z][$_.a-zA-Z0-9]*
    - A symbol begin with a letter or with one of '_$.', which can be followed by any string of digits, letters, '$', '_' and '.'.
  - Case-sensitive.
- _ (underscore)
  - In some platforms, '_foo' corresponds to 'foo' in C.

Labels (1)



- Label:
  - Can be placed at the beginning of a statement.
  - Is assigned the address that the assembler calculates.
  - Can occur as an operand.
  - Used to represent a variable name, function name, jump destination.
- Symbolic label:
  - Is a symbol followed by ':' (colon). (e.g., _main:)
  - Is global, so must be defined only once.
  - A symbol that begins with an 'L' is assumed to be local.
    - Not included in the object file's symbol table.
- Numeric label (explained later)

Labels (2): label as variable name

- Global variables are statically allocated in '.data' section.

foo.c

```
int foo = 999;
char bar = 15;
```

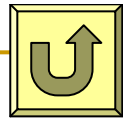
foo.s

```
.globl _foo
    .data
    .align 4
_foo:
    .long   999
.globl _bar
_bar:
    .byte   15
```

```
% nm foo.o
00000000 b .bss
00000000 d .data
00000000 t .text
00000004 D _bar
00000000 D _foo
```

Addresses are assigned to the variables.

Labels (3): label as function name



- Functions are statically allocated in '.text' section.

foo.c

```
int inc (int x)
{
    return x + 1;
}
```

foo.s

```
    .text
.globl _inc
    .def _inc; .scl 2; .type 32; .endef
_inc:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    incl    %eax
    popl    %ebp
    ret
```

```
% nm foo.o
00000000 b .bss
00000000 d .data
00000000 t .text
00000000 T _inc
```

Labels (4): label as jump destination

- Local labels are used for jump destinations.

```
int abs (int x)
{
    if (x < 0)
        x = -x;
    return x;
}
```

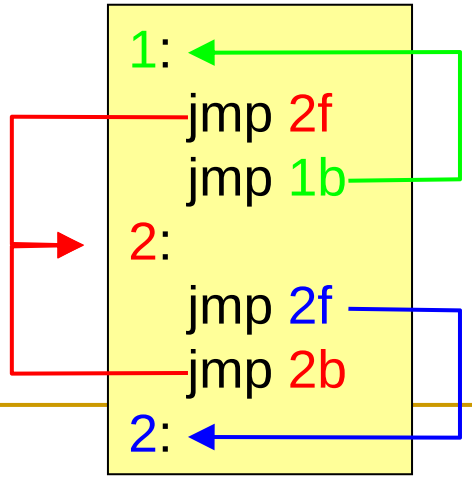
```
    .text
.globl _abs
    .def _abs; .scl 2; .type 32; .endef
_abs:
    pushl   %ebp
    movl    %esp, %ebp
    cmpl    $0, 8(%ebp)
    jns     L2
    negl    8(%ebp)
L2:
    movl    8(%ebp), %eax
    popl    %ebp
    ret
```

8(%ebp) - $0 >= 0?

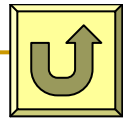
Jump short if not sign (SF=0)

Labels (5): Numeric label

- A single digit (0-9) followed by ':' (colon).
- Local, so can be redefined repeatedly.
  - So, useful when describing an inline assembly code.
  - Not included in the object file's symbol table.
- The suffixes 'b' (backward) or 'f' (forward) is added to the numeric labels, when used as a reference.



Labels (6): special dot (.)

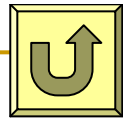


- '.' (dot) refers to the current address that 'as' is assembling into.

```
_foo:
    .long  .
```

_foo's value is the address of _foo.

AT&T-, vs., Intel-style (1)



- Can be selected by the GCC options:
  - -masm=intel, -masm=att (default)
- Can be selected by 'as' directives:
  - .intel_syntax, .att_syntax (default)
- Can be selected by the objdump options:
  - -M intel, -M att (default)
  - In AT&T mode, '-M suffix' adds the operand size suffixes (b, w, l, q).

AT&T-, vs., Intel-style (2)

sub.c

```
int sub (int a, int b) {
    return a - b;
}
```

```
% gcc -S -masm=intel sub.c
```

sub.s

```
  .intel_syntax
  .text
.globl _sub
  .def _sub; .scl 2; .type 32; .endef
_sub:
  push  ebp
  mov   ebp, esp
  mov   edx, DWORD PTR [ebp+12]
  mov   eax, DWORD PTR [ebp+8]
  sub   eax, edx
  pop   ebp
  ret
```

```
% objdump -d -M suffix sub.o
00000000 <_sub>:
0:   55              pushl  %ebp
1:   89 e5           movl   %esp,%ebp
3:   8b 55 0c        movl   0xc(%ebp),%edx
6:   8b 45 08        movl   0x8(%ebp),%eax
9:   29 d0           subl   %edx,%eax
b:   5d              popl   %ebp
c:   c3              retl
d:   90              nop
```

AT&T-, vs., Intel-style (3)

- Immediate operands, register operands, absolute jump/call operands

	AT&T	Intel
immediate operands	pushl $4	push 4
register operands	pushl %ebp	push ebp
absolute jump/call operands	jmp *0x100	jmp ds:0x100

- Operand order

	AT&T	Intel
operand order	addl $4, %eax	add eax, 4

AT&T-, vs., Intel-style (4)

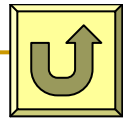
- Operand sizes

size	type	AT&T	Intel
1 byte	byte	movb $2, %al	mov al, byte ptr 2
2 byte	word	movw $2, %ax	mov ax, word ptr 2
4 byte	long	movl $2, %eax	mov eax, long ptr 2

- Immediate form far jump/call/return

	AT&T	Intel
jump	ljmp $section, $offset	jmp far section:offset
call	lcall $section, $offset	call far section:offset
return	lret $stack-adjust	ret far stack-adjust

AT&T-, vs., Intel-style (5)

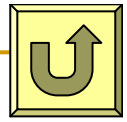


- Instruction names are the same except for some cases.

- Exception 1: The last letter in AT&T-style represents the operand size.
  - E.g., movl, movw, movb
- Exception 2: 'lcall/ljmp' in AT&T-style while 'call far/jmp far' in Intel-style.
- Exception 3: Conversion instructions:

AT&T	Intel	Description
cbtw	cbw	sign-extend byte in %al to word in %ax
cwtl	cwde	sign-extend word in %ax to long in %eax
cwtd	cwd	sign-extend word in %ax to long in %dx:%ax
cltd	cdq	sign-extend dword in %eax to quad in %edx:%eax

AT&T-, vs., Intel-style (6)



- Exception 4: Sign-extend, zero-extend mov instructions.
  - In AT&T-style, the suffixes that specify two sizes are embedded to 'movs..' and 'movez..', respectively.
  - In Intel-style, the mnemonics 'movsx' and 'movzx' are used.

AT&T suffix	Description
bl	byte -> long
bw	byte -> word
wl	word -> long
bq	byte -> quad
wq	word -> quad
lq	long -> quad

AT&T	Intel
movsbl %al, %edx	movsx edx, al

AT&T-, vs., Intel-style (7)

- Memory references

	memor reference form
AT&T-style	section: disp (base, index, scale)
Intel-style	section: [base + index * scale + disp]

- Effective address = section : base + index * scale + disp;

Only one comma is a syntax exception.

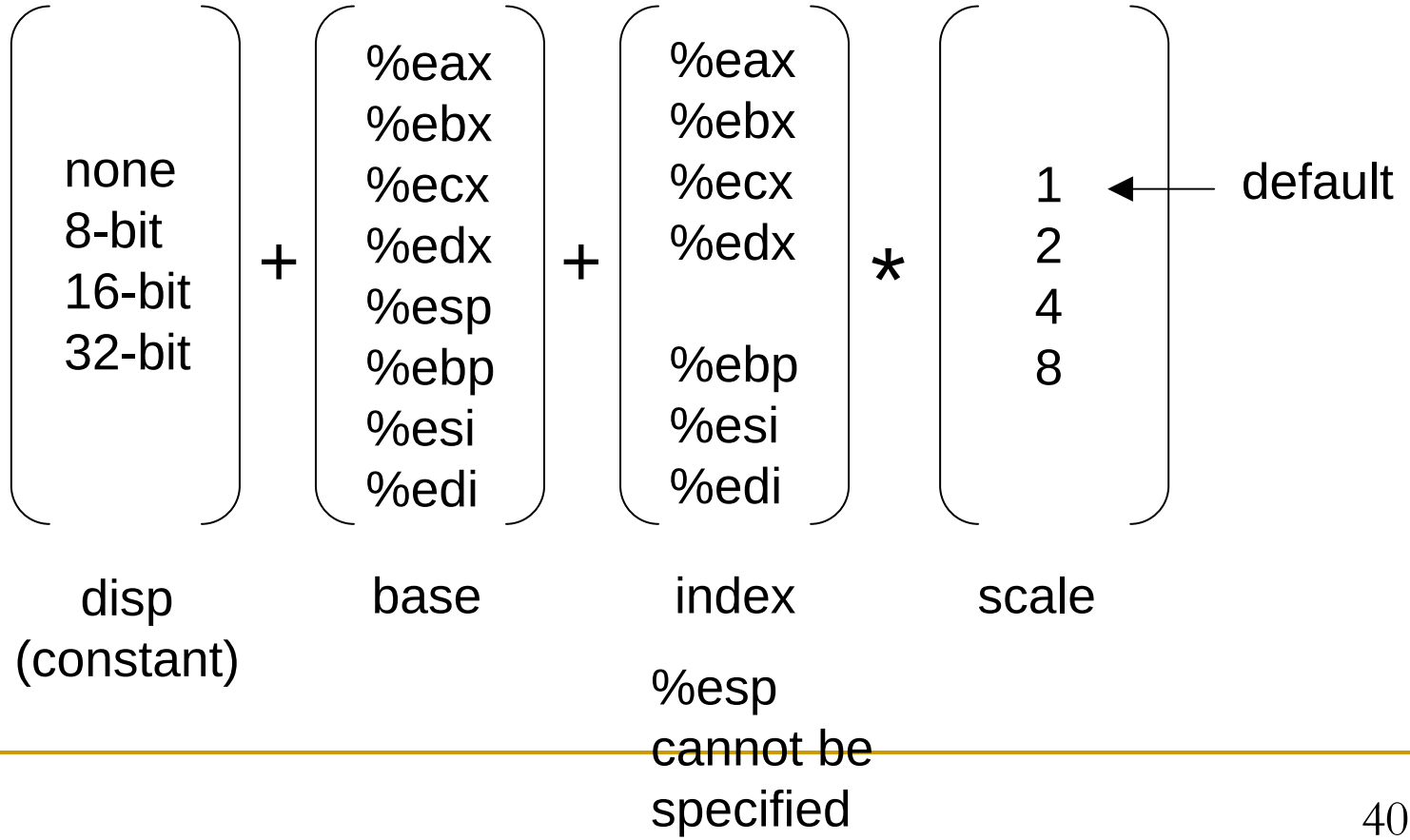
AT&T	Intel	description
-4(%ebp)	[ebp - 4]	disp and base
foo(,%eax,4)	[foo+eax*4]	disp, index and scale
foo(,1)	[foo]	disp and scale
%gs:foo	gs:foo	section and disp

assembly	C
$foo	foo
foo	*foo

is the same as 'foo'.

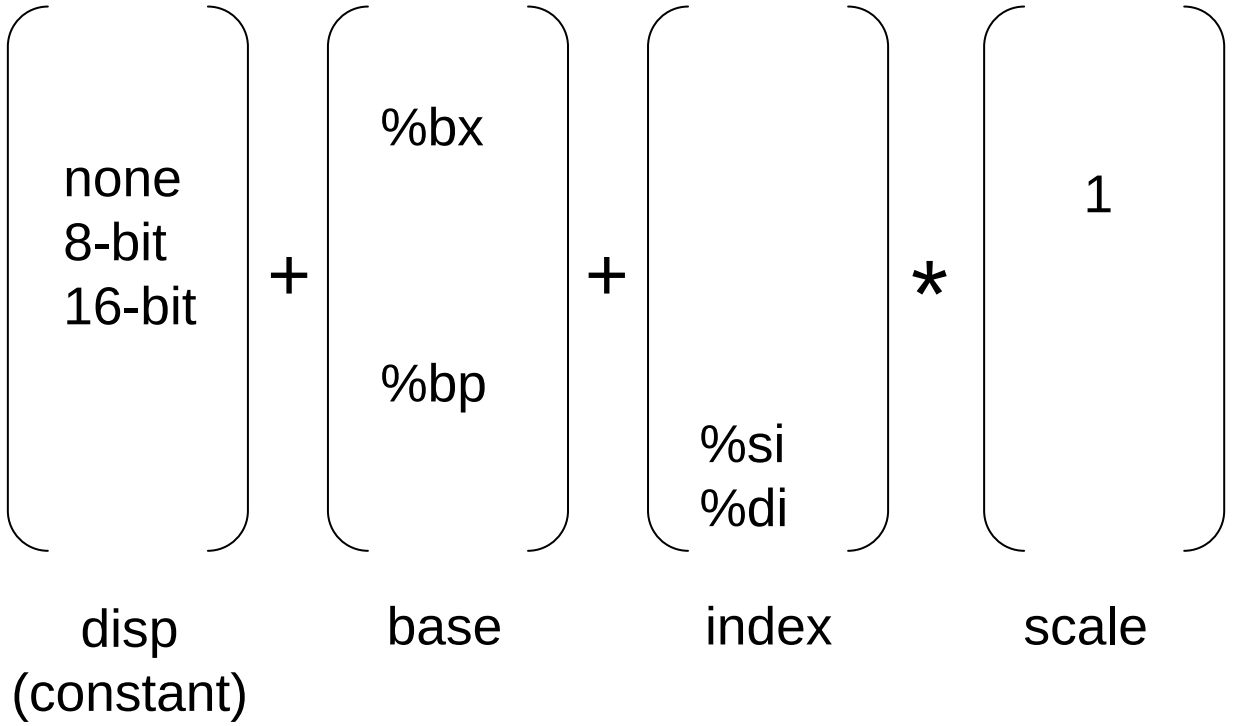
AT&T-, vs., Intel-style (8)

- Restrictions on memory references
  - In 32-bit protected mode.



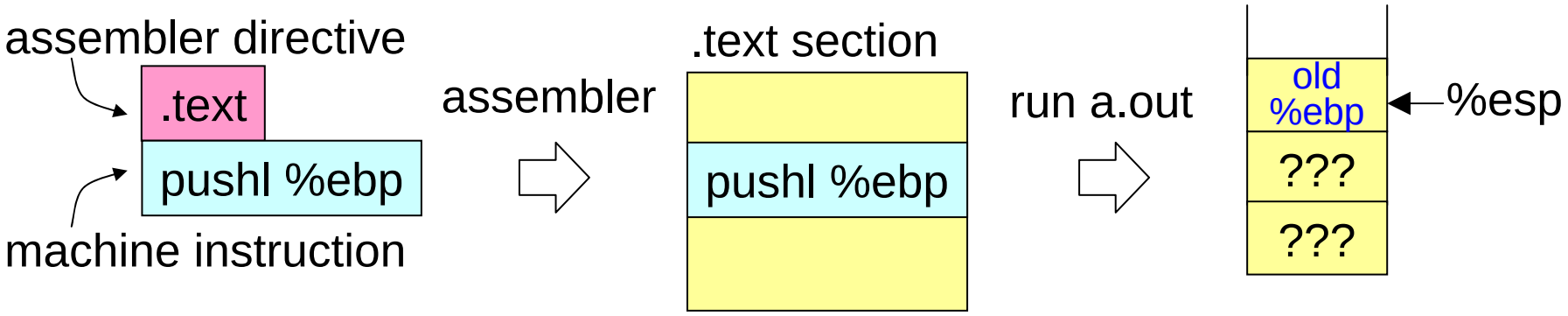
AT&T-, vs., Intel-style (9)

- Restrictions on memory references
  - In 16-bit protected mode or real mode.



Assembler directives (1)

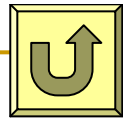
- Aka: pseudo opcodes, pseudo instructions 擬似命令
- All assembler directives begin with a period (.).
- Assembler directives are commands to the assembler.
  - Not related to (x86) machine instructions.
  - Assembler directives themselves don't generate any code.
  - Machine instructions are just data for the assembler.



Assembler directives are executed (by the assembler) at compile-time.

machine instructions are executed at run-time.

Assembler directives (2)



- Main categories:
  - Section information: .text, .data, .section
  - Data allocation: .ascii, .asciz, .byte, .word, .int (= .long), .fill, ...
  - Address adjustment: .align, .skip (= .space), .org
  - Link information: .common, .globl (= .global), .local
  - Symbol information: .size, .type
  - Debug information: .file, .ident
  - Macro processing: .set, .macro, .endm, .ifdef, .endif, .include
    - Recall that the files "*.S" will be preprocessed by CPP, so you don't need to use these macro directives.

Windows vs., Linux

```
int sub (int a, int b) {
    return a - b;
}
```

- Windows XP, Cygwin, GCC-3.3.4

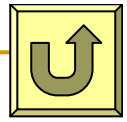
```
    .file   "sub.c"
    .text
.globl _sub
    .def _sub;  .scl 2; .type 32; .endef
_sub:
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %edx
    movl    8(%ebp), %eax
    subl    %edx, %eax
    popl    %ebp
    ret
```

- Linux-2.4.19, GCC-3.3.2

```
    .file "sub.c"
    .text
.globl  sub
    .type sub, @function
sub:
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %edx
    movl    8(%ebp), %eax
    subl    %edx, %eax
    popl    %ebp
    ret
    .size sub, .-sub
    .ident "GCC: (GNU) 3.3.2"
```

Recall dot (.) means the current address.

Sections: text, .data, ...



- A binary program is divided into sections.
- Most important sections:

section name	description
.text	binary code (i.e., instructions).
.rdata (.rodata)	read-only data (e.g., string literals).
.data	initialized global variables.
.bss	uninitialized global variables.

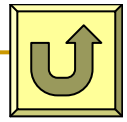
"foo"

int x=10;

int y;

- Other sections (in ELF): (not explained further)
  - Sections for shared libraries: .got, .plt, .interp, .dynamic, ...
  - Sections not loaded into memory: symbol table (.symtab), relocation info. (.rel.text, .rel.data), debug info. (.debug), line info. (.line), string table (.strtab), ...

'objdump' command

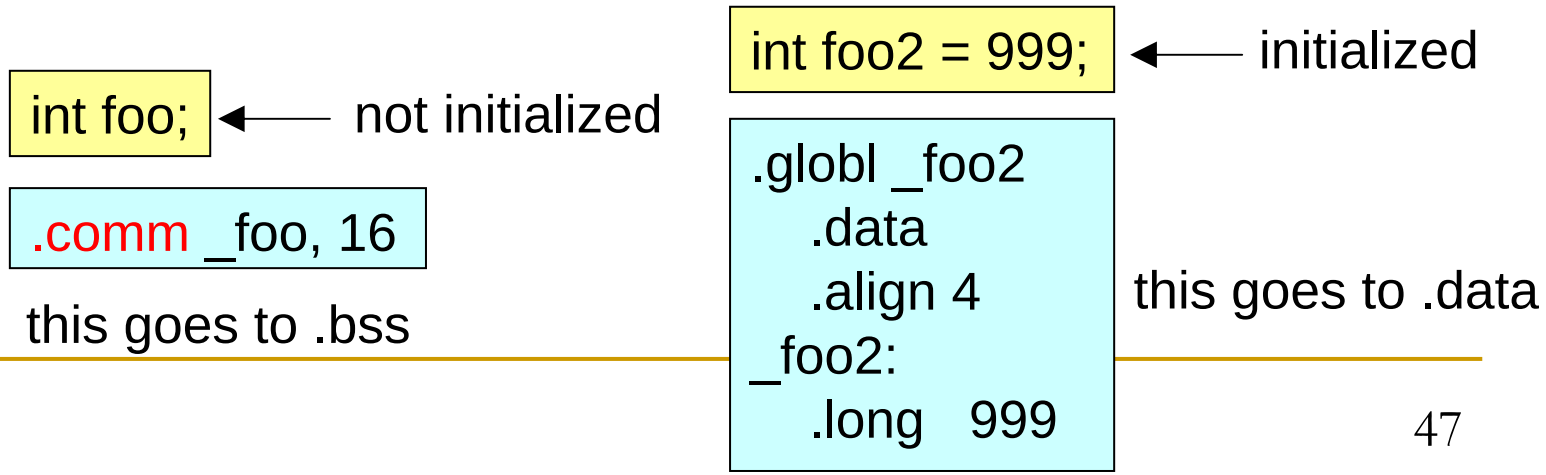


- 'objdump --section-headers' displays the list of sections.

```
%  objdump --section-headers foo.o
foo.o:     file format pe-i386
Sections:
Idx Name      Size          VMA         LMA         File off      Algn
  0 .text       00000020  00000000  00000000  000000b4  2**4
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data       00000010  00000000  00000000  000000d4  2**4
                CONTENTS, ALLOC, LOAD, DATA
  2 .bss        00000000  00000000  00000000  00000000  2**4
                ALLOC
  3 .rdata      00000010  00000000  00000000  000000e4  2**4
                CONTENTS, ALLOC, LOAD, READONLY, DATA
```

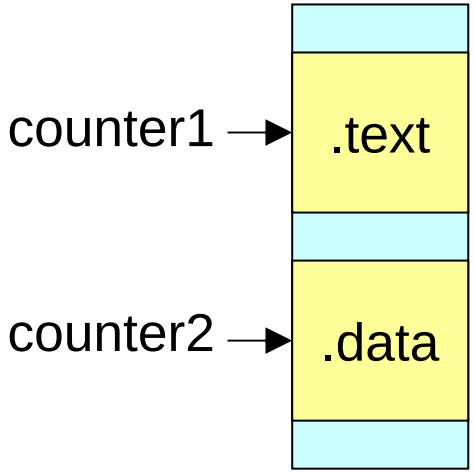
.bss section

- Abbrev. of "block started by symbol" for historical reason.
- .bss section does not exist in foo.o nor a.out.
  - .bss is created with zero values when a.out is loaded to memory.
  - Uninitialized variables (e.g., int a[1024]) in .bss section have only a name and a size, but no value (i.e., not allocated).
    - This contributes to reducing the binary size.
- .comm and .lcomm directives are used for uninitialized variables.



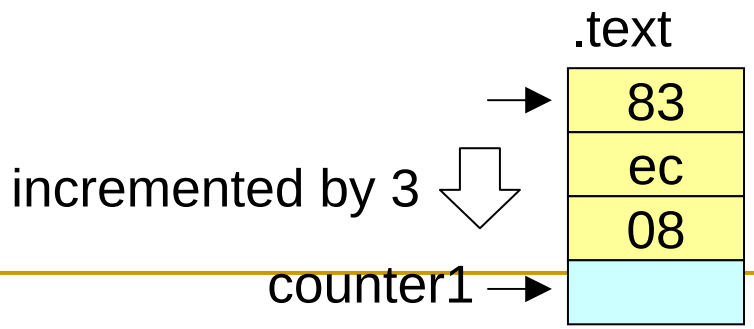
Location counters

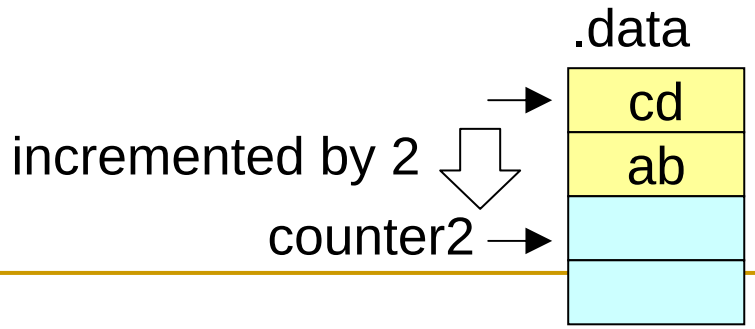
- Each section has an implicit location counter.
  - It begins at zero.
  - It is incremented by one for each byte assembled in the section.



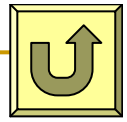

```
83 ec 08    sub $0x8,%esp
```

```
_foo:
  .word 0xABCD
```





Data allocation (1)



- .ascii "*string*" [, "*string*"]*
- .asciz "*string*" [, "*string*"]*
  - Assembles (i.e., emits) each string into consecutive addresses.
  - 'asciz' appends the null byte '\0' to the end of *string*, while 'ascii' does not.
- .byte *expression* [, *expression*]*
- .short *expression* [, *expression*]* (syn. .word)
- .long *expression* [, *expression*]* (syn. .int)
- .quad *expression* [, *expression*]*
  - Assembles each *expression* into the next 1, 2, 4, 8-byte integer, respectively.
- .fill *repeat* [, *size* [, *value*]]
  - Assembles *repeat* copies of *size* bytes, each of value *value*.
  - .fill has some bizarre behavior. Refer to 'info as' for the details.

Data allocation (2): example(1)

```
.data
_hoge1:
 .byte   0xAB
_hoge2:
 .short  0xCDEF
_hoge3:
 .int    0xDEADBEEF
_hoge4:
 .long   0xCAFEBABE
```

```
% nm foo.o
00000000 d _hoge1
00000001 d _hoge2
00000003 d _hoge3
00000007 d _hoge4
```

```
% objdump --section-headers foo.o
Idx Name  Size      VMA       LMA       File off  Algn
  0 .text 00000000  00000000  00000000  00000000  2**4
                    ALLOC, LOAD, READONLY, CODE
  1 .data 00000010  00000000  00000000  0000008c  2**4
                    CONTENTS, ALLOC, LOAD, DATA
```

```
% od -Ax -t x1 foo.o
   .... (omitted)
000080   00 00 00 00 00 00 00 00 80 00 00 c0 ab ef cd ef
000090   be ad de be ba fe ca 00 00 00 00 00 2e 66 69 6c
```

Recall that x86 is little endian.

Data allocation (3): example(2)

- The data allocated in ".text" section is interpreted as an instruction.
  - Similarly, the instruction in ".data" section is interpreted as data.
- Instructions and data are the same in the sense that both of them are assembled into binary data.
  - Interpretation makes the difference.

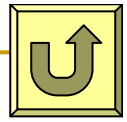
foo.s

```
.text
   nop
  .byte 0x90
.data
   nop
  .byte 0x90
```

```
% objdump -D foo.o
00000000 <.text>:
   0:   90      nop
   1:   90      nop

00000000 <.data>:
   0:   90      nop
   1:   90      nop
```

Address adjustment (1)



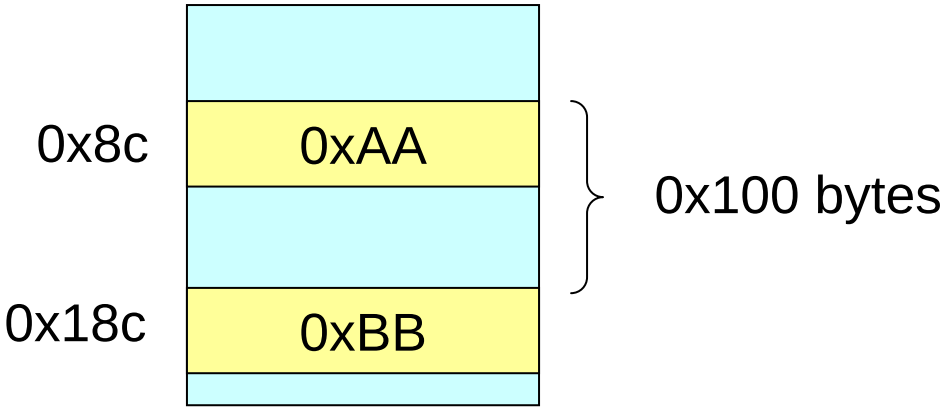
- .align *expression* [*, fill*]
  - Pads the location counter to a multiple of *expression*.
  - Default value of *fill* is zero (or nop).
- .skip *size* [, *fill*] (syn. .space)
  - Emits *size* bytes, each of value *fill*.
- .org *new-lc* [, *fill*] (syn. .origin)
  - Advances the location counter of the current section to *new-lc*.
  - you cannot use `.org' to move the location counter backwards.

Address adjustment (2)

foo.s

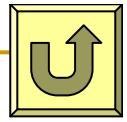
```
.data
  .byte 0xAA
  .org  0x100
  .byte 0xBB
```

foo.o




```
%  objdump -h foo.o
Idx Name Size      VMA       LMA       File off  Algn
  1 .data   00000110  00000000  00000000  0000008c  2**4
% od -Ax -t x1 foo.o
    ....
000080 00 00 00 00 00 00 00 00 80 00 00 c0 aa 00 00 00
000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
000180 00 00 00 00 00 00 00 00 00 00 00 00 bb 00 00 00
```

Link information (1)



- .comm *symbol*, *length* [, *align*]
- .lcomm *symbol*, *length* [, *align*]
  - .comm or .lcomm defines in the .bss section a global or local symbol *symbol*, respectively.
  - Used for uninitialized global or file-scope variables.
- .globl *symbol* (syn. .global)
- .local *symbol*
  - .globl makes the symbol *symbol* global, i.e., visible to the linker, while .local makes it file-scope, i.e., invisible to the linker.

Link information (2): example (1)

foo.c

```
          int foo;
static  int bar;
extern int afo;
```

- Variables on Windows

foo.s

```
.comm   _foo, 16    # 4
.lcomm  _bar, 16
```

- Variables on Linux

foo.s

```
.comm   foo,4,4
.local  bar
.comm   bar,4,4
```

Link information (3): example (2)

```
          int foo () {};
static  int bar () {};
extern int afo();
```

- Functions on Windows

```
.globl _foo
    .def _foo; .scl 2; .type 32; .endef
_foo:
    .... (omitted)

    .def _bar; .scl 3; .type 32; .endef
_bar:
    .... (omitted)
```

- Functions on Linux

```
.globl foo
    .type   foo, @function
foo:
    .... (omitted)
    .size   foo, .-foo

    .type   bar, @function
bar:
    .... (omitted)
   .size   bar, .-bar
```

Symbol information (1)

- .size *symbol*, *size*
  - Sets *size* for *symbol* in the symbol table.
- .type *symbol*, *type*
  - Sets *type* for *symbol* in the symbol table.

Symbol information (2)

PE/COFF storage class

IMAGE_SYM_CLASS_AUTOMATIC	1
IMAGE_SYM_CLASS_EXTERNAL	2
IMAGE_SYM_CLASS_STATIC	3
IMAGE_SYM_CLASS_REGISTER	4

PE/COFF type (MSB)

IMAGE_SYM_DTYPE_NULL	0
IMAGE_SYM_DTYPE_POINTER	1
IMAGE_SYM_DTYPE_FUNCTION	2
IMAGE_SYM_DTYPE_ARRAY	3

PE/COFF type (LSB)

IMAGE_SYM_TYPE_NULL	0
IMAGE_SYM_TYPE_VOID	1
IMAGE_SYM_TYPE_CHAR	2
IMAGE_SYM_TYPE_SHORT	3
IMAGE_SYM_TYPE_INT	4

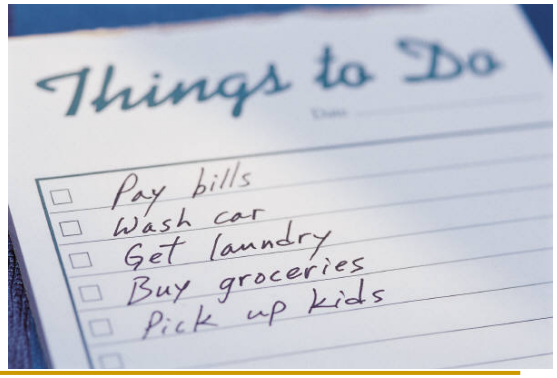
PE/COFF's type field contains 2 bytes.

- E.g., 32 (= 0x20) means a function.

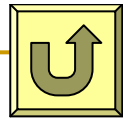
See MS-PE for the details.

Menu

- Introduction to x86 assembly programming
- GNU assembler: as
- Inline assembly (for x86) in GCC
- x86 instructions (in AT&T style)



Inline assembly code: overview



- Inline assembly code
  - Assembly code embedded in C.
  - Written by asm statements, a GCC-specific extension.
    - Where C expression operands can be specified as arguments.
- Why inline assembly code?
  - Optimization.
  - Access to processor-specific instructions (e.g., rdtsc).
  - Implementing system calls. (e.g., asm ("int 0x80");)
- Inline assembly code is nonportable!
  - Although C's raison d'etre is to write nonportable code,
    you should try to minimize the use of inline assembly code.

Inline assembly code: example (1)

foo.c

```
int foo (void) {
    asm ("nop");
}
```

inline assembly code

foo.s (some parts omitted)

```
_foo:
   pushl   %ebp
   movl    %esp, %ebp
/APP
   nop
/NO_APP
   popl    %ebp
   ret
```

embedded inline assembly code

Inline assembly code: example (2)

- Printing the value of the stack pointer (%esp).
  - We would like to store the value of '%esp' in a C variable 'addr'.

foo.c

```
#include <stdio.h>
int main (void) {
    void *addr;
    asm ("movl %%esp, %0":"=m"(addr));
    printf ("esp = %p\n", addr);
}
```

'%0' represents the first operand 'addr'. '%0' will be replaced by the operand corresponding to 'addr'.

```
asm ("movl %%esp, %0":"=m"(addr));
```

foo.s (some parts omitted)

```
/APP
    movl %esp, -4(%ebp)
/NO_APP
```

the local variable 'addr'

To produce one '%', '%%' must be written in the input.

operand constraint: '=' says the operand is an output, 'm' says memory is required.

Inline assembly code: example (3)

- Pentium family processors has a time-stamp counter.
- The time-stamp counter is 64-bit.
  - Set to 0 after the hardware reset.
  - Incremented every processor clock cycle.
- 'rdtsc' instruction loads the current count of the time-stamp counter into the %edx:%eax registers.

foo.c

```
#include <stdio.h>
int main (void) {
    unsigned long long c;
    asm ("rdtsc":"=A"(c));
    printf ("tsc = %llu\n", c);
}
```

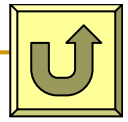
foo.s

```
/APP
   rdtsc
/NO_APP
   movl  %eax, -8(%ebp)
   movl  %edx, -4(%ebp)
```

variable 'c'

The operand constraint "A" says %edx:%eax registers are required.

Inline assembly code: syntax (1)



Syntax of asm statement

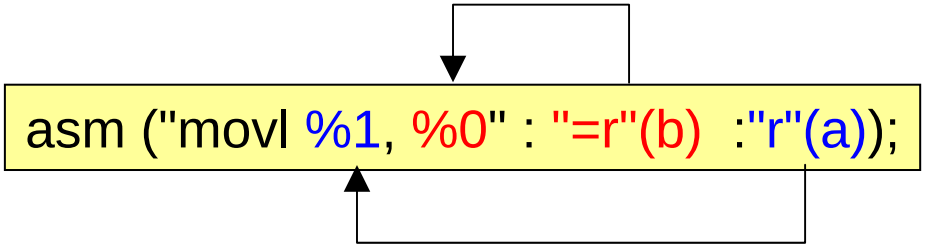
```
asm [volatile] ( assembler-template
                 [: output-operands
                 [: input-operands
                 [: clobbered-registers]]]
                 );
```

- 'volatile' is used to prevent GCC from deleting the asm statement as unused.
- '__asm__' and '__volatile__' can also be used.
  - To prevent the conflicts among identifiers.
- Always check the result of asm statements.
  - By visual observation of *.s files and the result of disassembling *.o files and a.out.

```
% gcc -S foo.c
% objdump -D foo.o
```

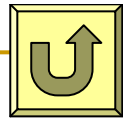
Inline assembly code: syntax (2)

- *assembler-template* is written as a string literal.
- *assembler-template* is similar to the printf format.
  - '%*n*' will be replaced by the corresponding operand.
    - %0, %1, %2, ... are associated with n-th operands, respectively, in *output-operands* and *input-operands*.



  - '%%' will be replaced by one '%'.
    - E.g., asm ("movl %%esp, %0": "=m"(addr));
    - This does not occur if there are no *input-operands* and no *output-operands*.
      - E.g., asm ("movl %esp, %eax");

Inline assembly code: syntax (3)



- *assembler-template* can contains multiple instructions.
  - By delimiting them with semicolons (;) or newlines (\n).

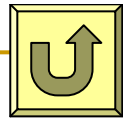
OK
```
asm ("nop; nop; nop");
```
delimited with semicolons.

OK
```
asm ("nop\n nop\n nop");
```
delimited with newlines.

OK
```
asm ("nop\n"
     "nop\n"
     "nop");
```
```
asm ("nop\n\t"
     "nop\n\t"
     "nop");
```
To write in multiple lines, just write multiple string literals, which will be concatenated into one string literal.

NG
```
asm ("nop
     nop
     nop");
```
In C, it is illegal to write raw newlines in string literals.

Inline assembly code: syntax (4)



- *output-operands* and *input-operands* consist of zero or more comma-separated operands.
- Each of them is of the form:

syntax of output-, input-operand.

"*operand-constraint*" (*C-expression*)

- In *output-operands*, *C-expression* must be a modifiable lvalue.
  - A modifiable lvalue is an expression that can appear in the left hand side of an assignment statement.
- *operand-constraint* tells GCC:
  - what kinds of registers an operand may be in,
  - whether the operand can be a memory reference,
  - etc.

Operand constraints (1)

selected from 'info gcc'

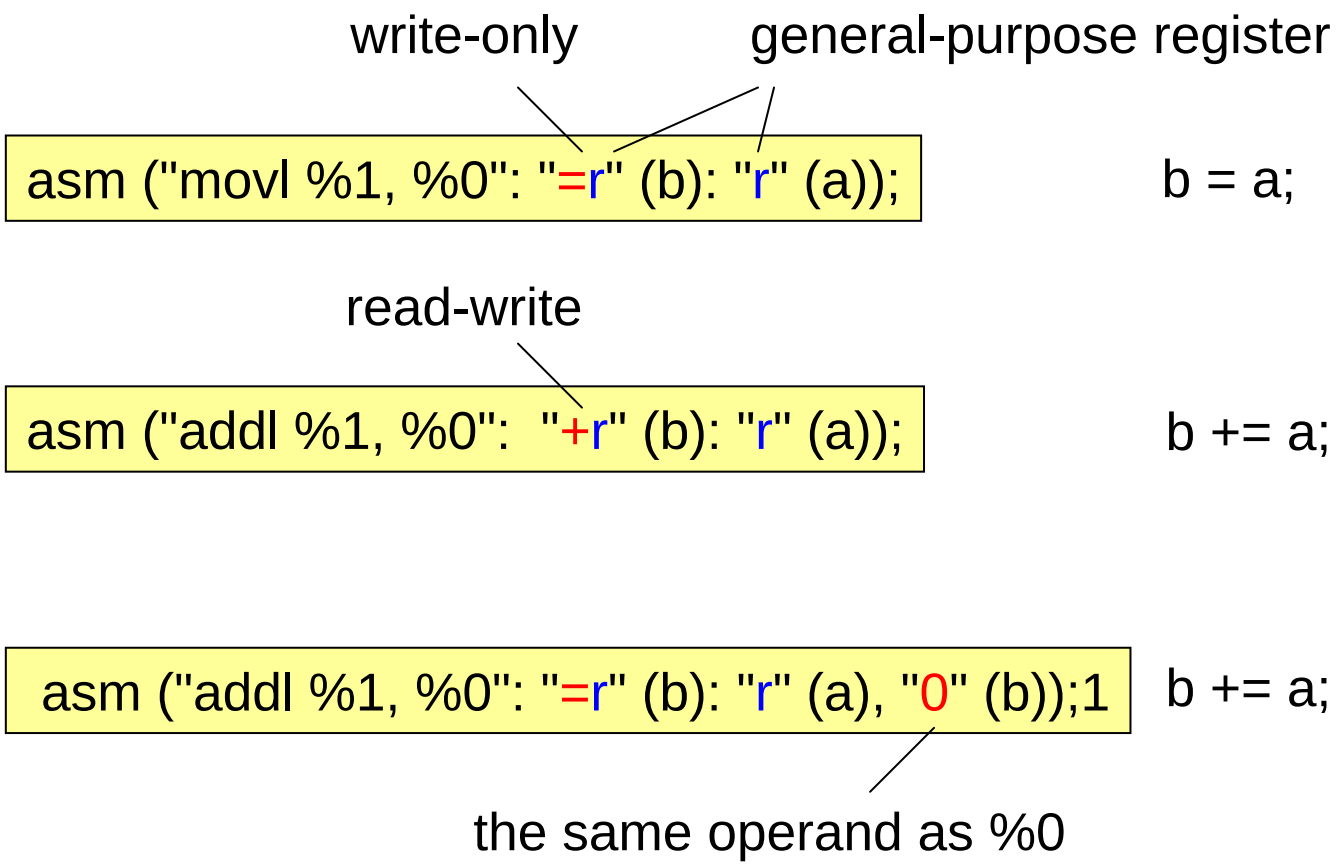
- Constraint modifier characters:
  - Put '=' or '+' in the first character of the constraint string.

=	this operand is write-only for this instruction.
+	this operand is both read and written by the instruction.

- Platform-independent constraint characters:

r	a general register is allowed.
m	a memory operand is allowed.
i	an immediate integer operand is allowed.
g	any general register, memory or immediate integer operand is allowed.
0	the same operand as '%0' is required. the same applies for 1, 2, 3, ..., 9.

Operand constraints (2): example



Operand constraints (3)

- x86-specific constraint characters:

selected from 'info gcc'

a	register %eax, %ax, or %al
b	register %ebx, %bx, or %bl
c	register %ecx, %cx, or %cl
d	register %edx, %dx, or %dl
q	register 'a', 'b', 'c', or 'd'
A	registers 'a' and 'd'
D	register %edi or %di
S	register %esi or %si
I	constant in range 0 to 31 (for 32-bit shifts)
J	constant in range 0 to 63 (for 64-bit shifts)
N	constant in range 0 to 255 (for in/out instructions)
K	constant in range -128 to 127 (for signed imm8 operands)
M	0, 1, 2, or 3 (scales for 'lea' instruction)

Operand constraints (4)

- *clobbered-registers*
  - Some instructions clobber some registers. We need to list those registers in *clobbered-registers*, as comma-separated strings.
    - To prevent unexpected overwrite of the register values.

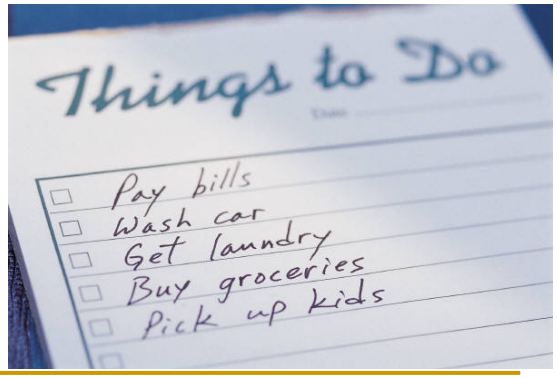
```
asm ("movl %0, %%edx"::"m"(a):"%edx");
```

  - We don't have to list the registers in *output-operands* or *input-operands*, since GCC checks them, but does not *assembler-template*.

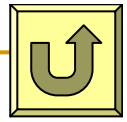
  - "cc" informs GCC that the condition code register (%eflags) is clobbered.
  - "memory" informs GCC that memory is modified in an unpredictable way, which causes GCC to not keep memory values cached in registers, and not optimize stores or loads to that memory.

Menu

- Introduction to x86 assembly programming
- GNU assembler: as
- Inline assembly (for x86) in GCC
- x86 instructions (in AT&T style)

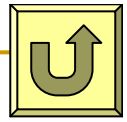


What we don't learn here



- Minor non-privilege instructions.
  - E.g., aaa (ASCII adjust after addition)
- Privilege instructions.
  - E.g., lidt (load interrupt description table)
- Floating point instructions
  - E.g., fadd
- Binary representations of machine instructions.

Overview of x86 (80386 and later)



- CISC (complex instruction set computer)
  - Multiple operations within a single instruction.
  - Many, non-orthogonal, variable-length instructions.
  - A few general-purpose registers.
  - Many addressing modes.
- 2-address code (only two operands)
  - E.g., addl %eax, %ebx # %ebx = %eax + %ebx"
  - cf., SPARC uses 3-address code.
    - E.g., add %l0, %l1, %l2 !%l2 = %l0 + %l1
- Variable-length instructions: 1 to 16 bytes.
  - cf., SPARC instructions have a fixed-length of 4bytes.

Operand notation (1)

abbrev. used here	notation used in Intel manuals	example	description
imm	*imm8, imm16, imm32*	$1234 $_foo	8-bit, 16-bit, 32-bit immediate value.
r	*r8, r16, r32*	%eax	8-bit, 16-bit, 32-bit general-purpose register.
r/m	*r/m8, r/m16, r/m32*	%eax 1234 (%eax) 4(%ebp)	8-bit, 16-bit, 32-bit general-purpose register, or memory
Sreg	*Sreg*	%ds	segment register

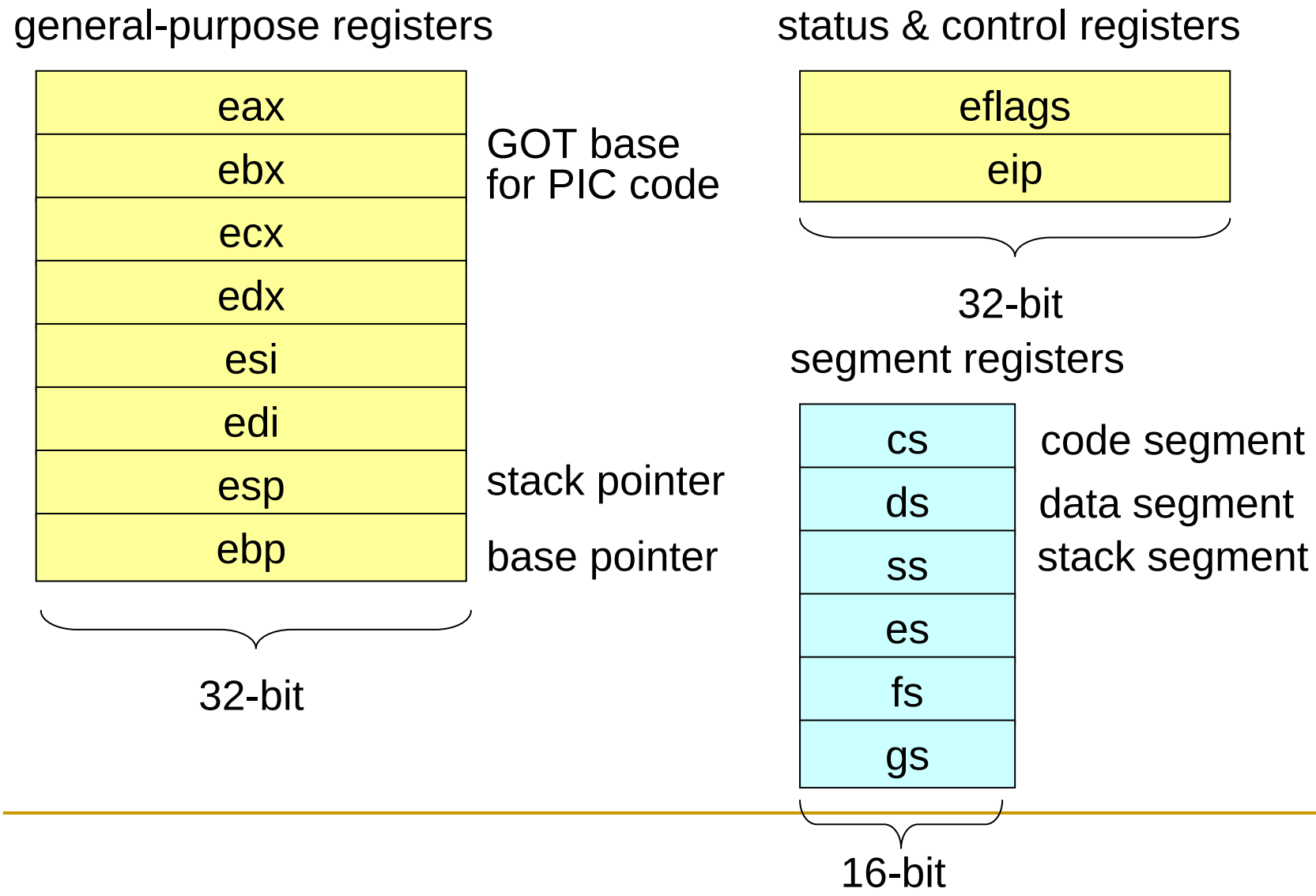
op1	the first operand (in AT&T style)
op2	the second operand (in AT&T style)

Operand notation (2)

- Notations for jump operands:

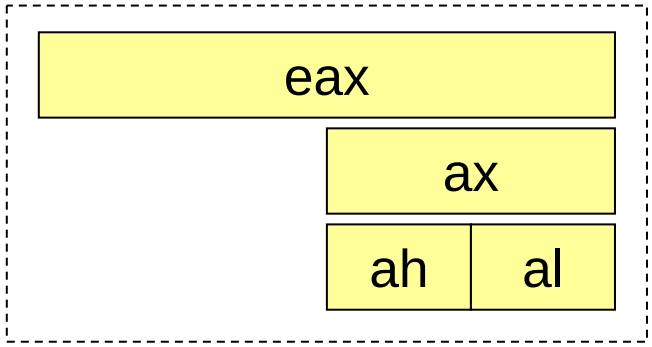
abbrev. used here	notation used in Intel manuals	description
rel8 *rel16/32*	*rel8* *rel16, rel32*	immediate relative address
m16:16/32	*m16:16* *m16:32*	far pointer in memory
ptr16:16/32	*ptr16:16* *ptr16:32*	immediate far pointer
m16/32&16/32	*m16&32* *m16&16* *m32&32*	data item pairs in memory

Application programming registers

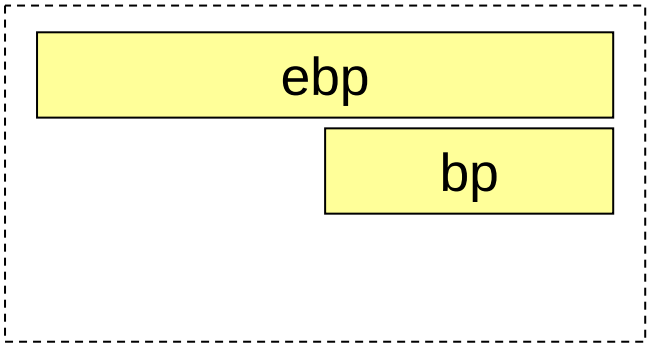


General-purpose registers

- A part of the registers can be referenced with other names:

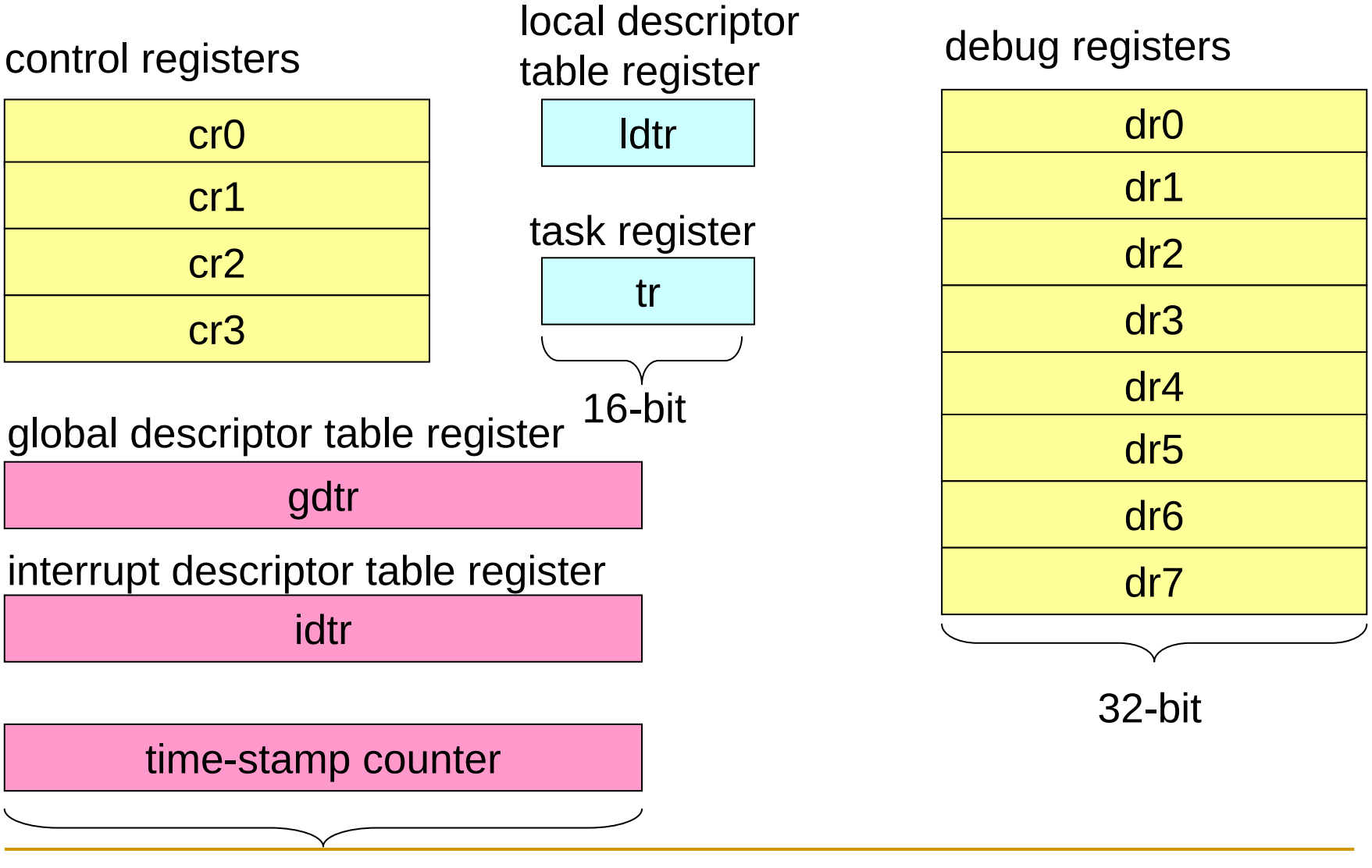


The same applies for ebx, ecx, edx.

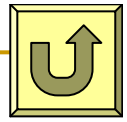


The same applies for esp, esi, edi, eip, eflags.

System registers (not exhaustive)



Other registers (not exhaustive)

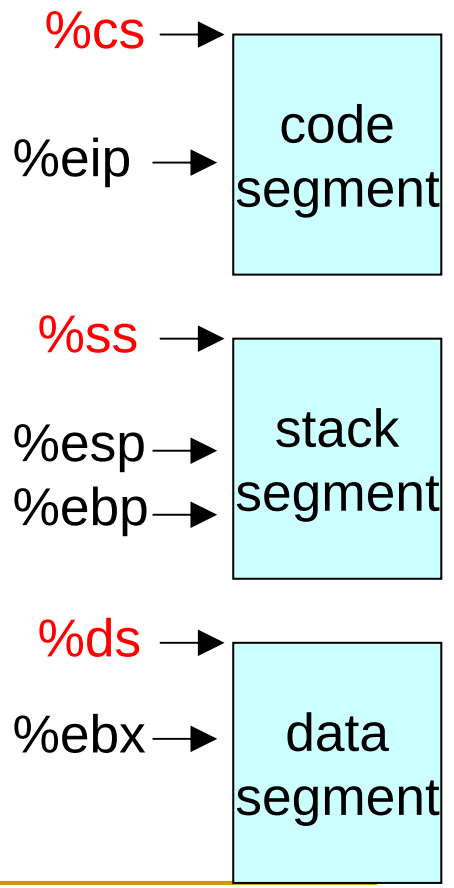


- 80-bit floating-point registers
  - st(0), ..., st(7)
- 64-bit MMX registers
  - mm0, ..., mm7
- 128-bit SSE registers
  - xmm0, ..., xmm7

Default segment selection rules (1)

- The processor automatically chooses a segment according to the following rules.

register used	segment used	default selection rule
%cs	code segment	All instruction fetches.
%ss	stack segment	All stack pushes and pops. Any memory reference using %ebp or %esp as a base register.
%ds	data segment	All data reference, except when relative to stack or string destination.
%es	data segment pointed to with %es	Destination of string instructions.

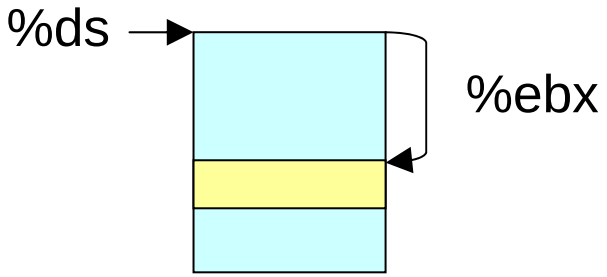


Default segment selection rules (2)

- E.g., %ds is chosen for '(%ebx)' in the right figure.

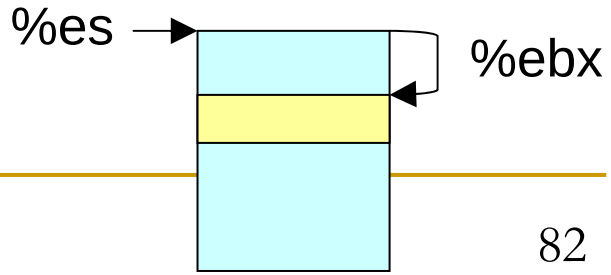
both are the same

```
movl        (%ebx), %eax
movl  %ds:(%ebx), %eax
```



- %ds default can be overridden by a segment override prefix like '%es:'.
- The following cannot be overridden.
  - Instruction fetches by %cs.
  - Destination strings in string instructions pointed to by %es.
  - Push and pop operations by %ss.

```
movl  %es:(%ebx), %eax
```



Operand addressing

- In general, an "addressing mode" refers to how to specify an operand.
  - Although an "addressing mode" originally refers to how to specify an address in memory location.

addressing mode	operand value	examples
immediate	constant (literal)	addl $0x100, %eax addl $_foo, %eax
register	content of register	addl %ebx, %eax

memory (direct addressing)	memory content specified by address constant	addl 0x100, %eax
		addl (0x100), %eax
		addl _foo, %eax

memory (indirect addressing)	memory content specified by registers	addl (%ebp), %eax
		addl -4(%ebp), %eax
		addl _foo(%ebp), %eax

Instruction prefixes (1)

- Divided into five groups:

segment override prefix	cs, ds, ss, es, fs, gs
operand size prefix	data16, data32
address size prefix	addr16, addr32
bus lock prefix	lock
string instruction prefix	rep, repe, repne, repnz, repz

- Used to modify instructions like the following examples.

```
40        inc  %eax
f0 40     lock inc  %eax
```

bus lock prefix

```
8b 10     mov      (%eax),  %edx
8b 10     movl  %ds:(%eax), %edx
64 8b 10  mov   %fs:(%eax),  %edx
```

section override prefix

Instruction prefixes (2)

- Examples of operand size prefixes

.code32 tells to 'as' that default operand/address size is 32-bit to be run in 32-bit mode.

code32.s

```
.code32
push        $0x1234
pushl       $0x1234
pushw       $0x1234
data16 push $0x1234
```

```
% gcc -c code32.s
% objdump -d -Msuffix code32.o
    68 34 12 00 00   pushl   $0x1234
    68 34 12 00 00   pushl   $0x1234
66 68 34 12          pushw   $0x1234
66 68 34 12          pushw   $0x1234
```

code16.s

```
.code16
push        $0x1234
pushl       $0x1234
pushw       $0x1234
data32 push $0x1234
```

```
% gcc -c code16.s
% objdump -d -Mi8086,suffix cod16.o
    68 34 12          pushw  $0x1234
66 68 34 12 00 00     pushl  $0x1234
    68 34 12          pushw  $0x1234
66 68 34 12 00 00     pushl  $0x1234
```

Instruction prefixes (3)

- Address size prefixes change the size of effective address.
- Examples of address size prefixes

code32.s

```
.code32
.text
push          4(%ebp)
pushl         4(%ebp)
pushw         4(%bp)
addr16 pushw 4(%bp)
```

```
% gcc -c code32.s
% objdump -d -Msuffix code32.o
ff 75 04                pushl  0x4(%ebp)
ff 75 04                pushl  0x4(%ebp)
67 66 ff 76 04  addr16 pushw   4(%bp)
67 66 ff 76 04  addr16 pushw   4(%bp)
```

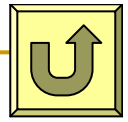
code16.s

```
.code16
.text
push          4(%ebp)
pushl         4(%ebp)
pushw         4(%bp)
addr32 pushl 4(%ebp)
```

```
% gcc -c code16.s
% objdump -d -Mi8086,suffix cod16.o
    67 ff 75 04  addr32 pushw 0x4(%ebp)
67 66 ff 75 04  addr32 pushl  0x4(%ebp)
       ff 76 04            pushw    4(%bp)
67 66 ff 75 04  addr32 pushl  0x4(%ebp)
```

nop: no operation

%eflags not affected.



- nop performs no operation, except for %eip.
- nop is a 1-byte instruction (0x90).

syntax	example	description
nop	nop	

mov: move (1)

%eflags not affected.

- mov instruction copies the first operand (in AT&T style) to the second operand.

syntax	example	description
mov *r, r/m*	movl %eax, %edx	%edx=%eax
	movl %eax, 4(%ebp)	*(%ebp+4)=%eax
mov *r/m, r*	movl 4(%ebp), %eax	%eax=*(%ebp+4)
mov *imm, r*	movl $999, %edx	%edx=999
mov *imm, r/m*	movl $999, 4(%ebp)	*(%ebp+4)=999

- mov instruction cannot directly copy from memory to memory.

NG `mov 4(%ebp), -4(%ebp)`

Note: some different machine instructions do the same thing.

```
c7 c1 22 33 44 55  movl  $0x55443322,%ecx
b9 22 33 44 55     movl  $0x55443322,%ecx
```

mov: move (2)

%eflags not affected.

- When one operand is a segment register, the other operand of mov instruction is limited to '*r/m16*'.

syntax	example	description
mov *r/m16, Sreg*	movw %ax, %es	%es=%ax
	movw 4(%ebp), %es	%es=*(%ebp+4)
mov *Sreg, r/m16*	movw %es, %ax	%ax=%es
	movw %es, 4(%ebp)	*(%ebp+4)=%es

- mov instruction from *Sreg* to *Sreg*, nor from *imm* to *Sreg* is not permitted.

NG mov *Sreg, Sreg*
NG mov *imm, Sreg*

NG movw $0x10, %es

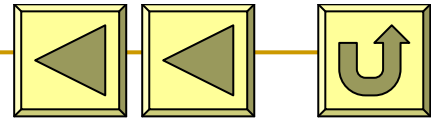
OK
movw $0x10, %ax
movw %ax, %es

xchg: exchange operands

%eflags not affected.

- xchg instruction exchanges the contents of two operands.
- In some cases, mov instruction is not atomic.
  - In contrast, 'lock xchg' is atomic, so xchg is useful for implementing synchronization mechanisms.

syntax	example	description
xchg *r, r/m*	movl %eax, %edx movl %eax, 4(%ebp)	
xchg *r/m, r*	movl 4(%ebp), %eax	



lea: load effective address

%eflags not affected.

- lea computes the effective address of the first operand, and stores the address in the second operand.
  - Note: lea never reads memory.

syntax	example	description
lea m, r16	leaw 4(%bx, %si), %bx	%bx=4+%bx+%si
lea m, r32	leal 0x100, %eax	%eax=0x100
	leal 4(%ebx,%esi, 4), %eax	%eax=4+%ebx+%esi*4

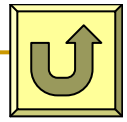
- Fast addition or multiplication with lea is common technique.

```
leal  4(%ebx,%esi, 4), %eax
```

both do the same thing, but lea computes it faster.

```
movl  $4,     %eax
addl  %ebx, %eax
sall  $2,     %esi
addl  %esi,  %eax
```

shift arithmetic left



push, pop: stack manipulation (1)

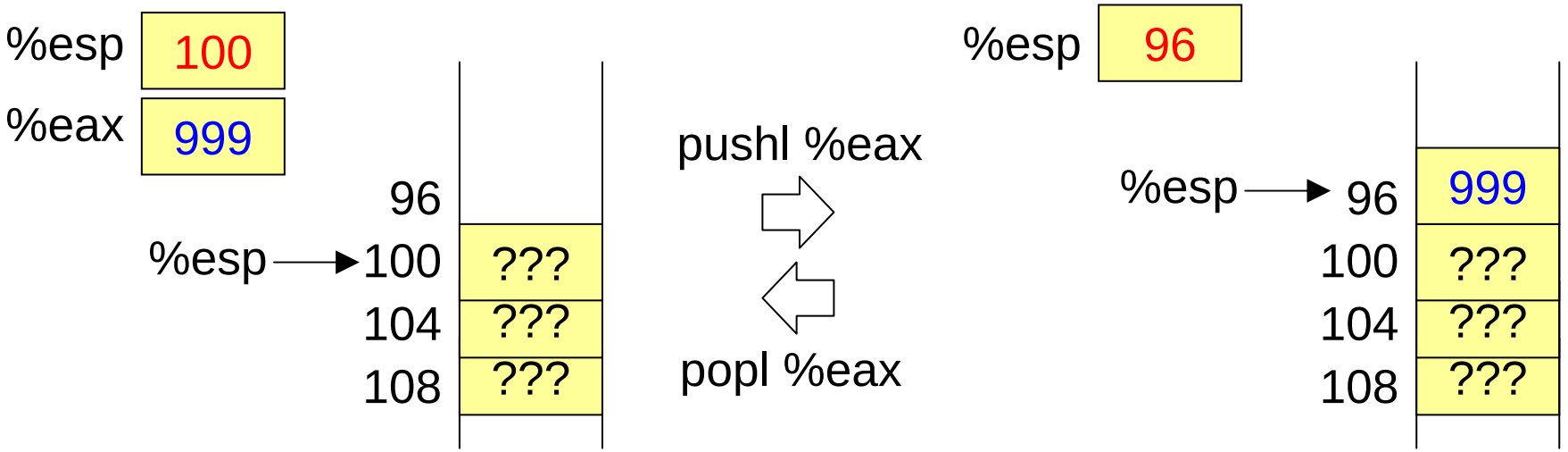
%eflags not affected.

- push decrements the stack pointer (%esp or %sp) and then stores the operand value on the stack top.
- pop loads the stack top value to the operand, and then increments the stack pointer.

syntax	example	description
push imm	pushl $999	%esp -=4; *(%esp)=999
push r/m16	pushw %ax	%sp -= 2; *(%sp) =%ax
push r/m32	pushl 4(%ebp)	%esp -= 4; *(%esp)=*(%ebp+4)
push Sreg	pushw %es	%sp -= 2; *(%sp) =%es

syntax	example	description
pop r/m16	popw %ax	%ax=*(%sp); %sp += 2;
pop r/m32	popl 4(%ebp)	*(%ebp+4)=*(%esp); %esp += 4;
pop Sreg	popw %es	%es=*(%sp); %sp += 2;

push, pop: stack manipulation (2)



- x86 stack grows down from higher address to lower address.

push, pop: stack manipulation (3)

%eflags not affected.

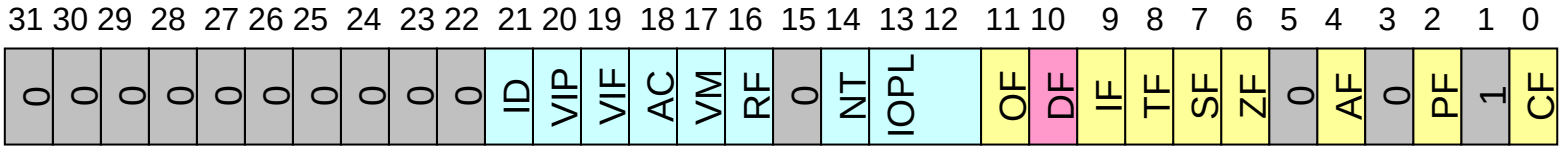
- pusha and popa saves/restores all general-purpose registers except %esp.

syntax	example	description
pusha	pusha	push %eax, %ecx, %edx, %ebx, %ebp, %esi, %edi
	pushal	(same as the above)
	pushaw	push %ax, %cx, %dx, %bx, %bp, %si, %di
popa	popa	pop %edi, %esi, %ebp, %ebx, %edx, %ecx, %eax
	popal	(same as the above)
	popaw	pop %di, %si, %bp, %bx, %dx, %cx, %ax

~~the order of registers does matter!~~
push/pop are performed in the order.

%eflags flag register (1)

- %eflags register



- Bits of %eflags are categorized:

  status flag

  control flag

  system flag

  reserved. (always set to values previously read.)

%eflags flag register (2)

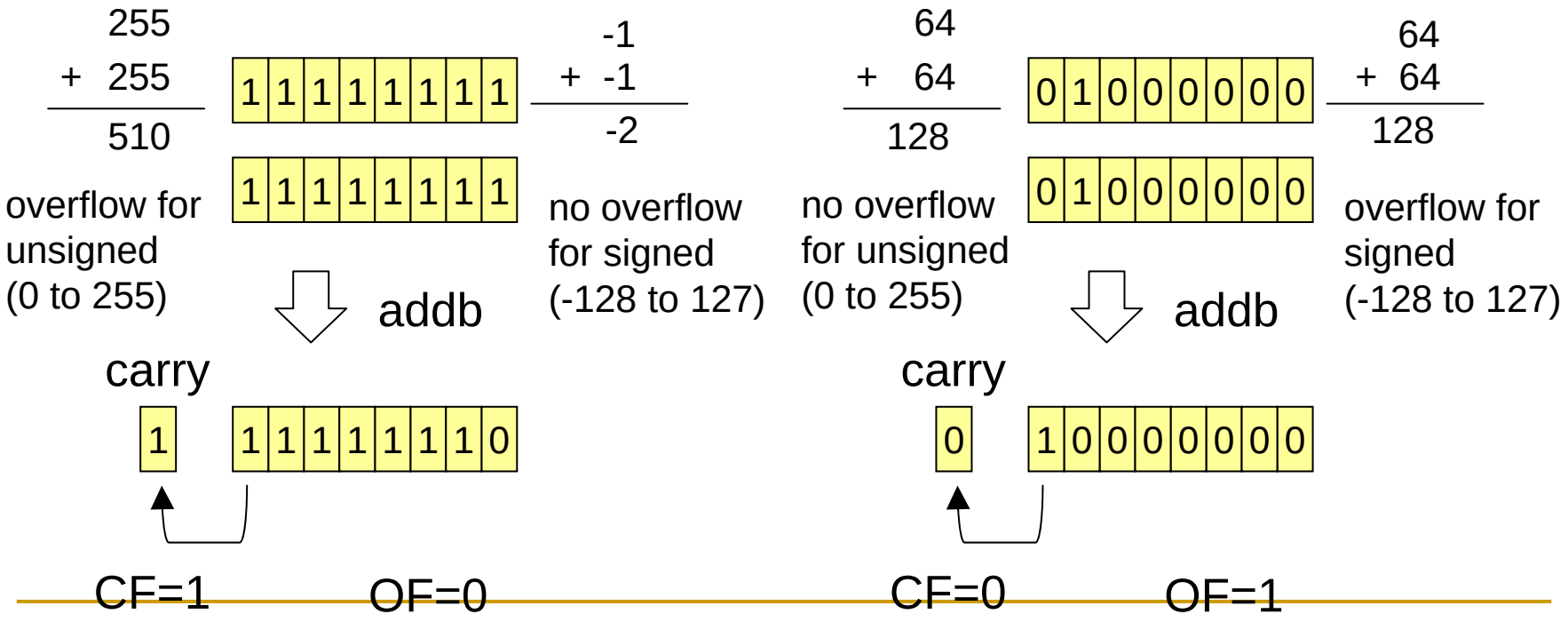
- Status flags

MSb=most significant bit 最上位ビット
BCD=binary coded decimal 二進化十進数

status flag	name	description
CF	carry flag	Set if an arithmetic operation generates a carry or a borrow from the MSB of the result; cleared otherwise.
PF	parity flag	Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.
AF	adjust flag	Set if a (BCD) arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise.
ZF	zero flag	Set if the result is zero; cleared otherwise.
SF	sign flag	Set equal to the MSb of the result, which is the sign bit of a signed integer.
OF	overflow flag	Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise.

%eflags flag register (3)

- difference between carry and overflow?
  - CF = 1 indicates an overflow for unsigned-integer arithmetic.
    - i.e., the result does not fit in the destination operand.
  - OF = 1 indicates an overflow for signed-integer arithmetic.



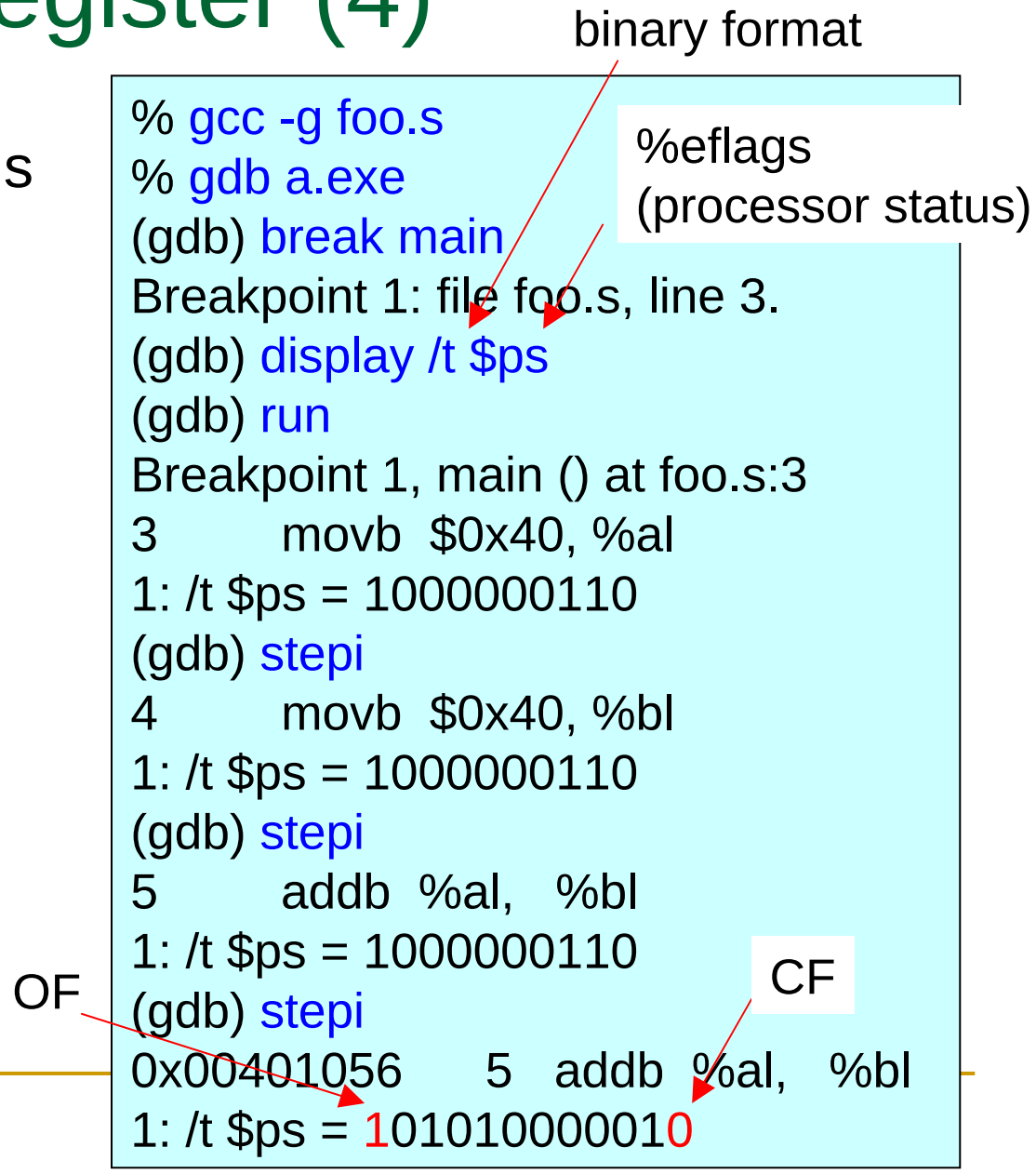
%eflags flag register (4)

- Let's observe %eflags using GNU gdb.

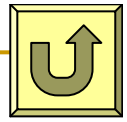
foo.s

```
.globl _main
_main:
  movb  $0x40, %al
  movb  $0x40, %bl
  addb  %al,   %bl
```

```
% gcc -g foo.s
% gdb a.exe
(gdb) break main
Breakpoint 1: file foo.s, line 3.
(gdb) display /t $ps
(gdb) run
Breakpoint 1, main () at foo.s:3
3         movb  $0x40, %al
1: /t $ps = 1000000110
(gdb) stepi
4         movb  $0x40, %bl
1: /t $ps = 1000000110
(gdb) stepi
5         addb  %al,   %bl
1: /t $ps = 1000000110
(gdb) stepi
0x00401056      5   addb  %al,   %bl
1: /t $ps = 101010000010
```



%eflags flag register (5)

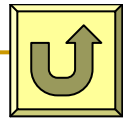


- Each flag in %eflags is set or cleared according to the result of the last instruction.
- The values of the flags are often used for conditional jump instructions.
  - Especially after cmp and test instructions.

```
cmpl   $0, 8(%ebp)
jg     L2
```

if 8(%ebp) > 0 then
jump to L2

%eflags flag register (6)



- %eflags can be examined or modified indirectly using the following instructions.
  - pushf/popf, pushfd/popfd save/restore the value of %eflags to/from stack.
  - lahf/sahf saves/restores the value of %eflags to/from %eax
  - Bit test instructions like bt, bts, btr, and btc.
- Other instructions related to %eflags.
  - jcc (conditional jump)
  - setcc (byte set on condition)
  - cmovcc (conditional move)
  - clc/stc/cmc (clear/set/complement carry flag)
  - cli/sti (clear/set interrupt flag)

Arithmetic logical instructions

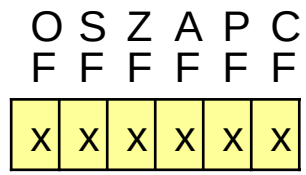
arithmetic	add, sub, mul,div, inc, dec, neg
logical	and, or, xor, not,
shift	sal, sar, shl, shr, rol, ror, rcl, rcr
compare	cmp, test
conversion	movs, movz, cbtw, cwtd

- Almost most of them modify %eflags, which will be shown using the following notation.

O F	S F	Z F	A F	P F	C F
	x	?	0	1	

(space)	not modified
x	modified
?	undefined
0	set
1	cleared

Arithmetic instructions (1)



syntax	example	description

add *imm*, *r/m*
add *r*, *r/m*
add *r/m*, *r*
	addl $123, %eax
	addl %eax, (%ebx)
	addl 4(%ebp), %eax
		%eax += 123;
		*(%ebx) += %eax;
		%eax += *(%ebp+4);

adc *op1*, *op2* (same as add)	adcl $123, %eax adcl %eax, (%ebx)	add with carry. op2 += op1 + CF;
sub *op1*, *op2* (same as add)	subl $123, %eax subl %eax, (%ebx)	%eax -= 123; *(%ebx) -= %eax;
sbb *op1*, *op2* (same as add)	sbbl $123, %eax sbbl %eax, (%ebx)	subtraction with borrow. op2 -= op1 + CF;

Arithmetic instructions (2)

- sub does not distinguish between signed or unsigned operands.
  - sub computes the result for both types.
  - OF, CF and SF flags can be used to know a borrow and the sign.

```
movb $0xFF,  %al
subb $0x01,  %al
```

0xFF	- 1
- 0x01	- 1
0xFE	- 2

OF=0
CF=0
SF=1

```
movb $0x80,  %al
subb $0x7F,  %al
```

0x80	- 128
- 0x7F	- 127
0x01	- 255

OF=1
CF=0
SF=0

```
movb $0x01,  %al
subb $0x02,  %al
```

0x01	1
- 0x02	- 2
0xFF	- 1

OF=0
CF=1
SF=1

Arithmetic instructions (3)

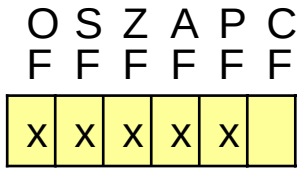
syntax	example	description
mul *r/m*	addl %ebx	unsigned multiply. %edx:%eax=%eax*%ebx;
imul *r/m* imul *r/m*, *r* imul *imm*, *r/m*, *r* imul *imm, r*	imull %ebx, %eax imull $4, %ebx, %eax imull $4, %eax	signed multiply. %eax *= %ebx; %eax = %ebx * 4; %eax *= 4;
div *r/m*	div %ebx	unsigned divide. %eax = (%edx:%eax) / %ebx; (%edx contains the remainder)
idiv *r/m*	idiv %ebx	signed divide. same as the above.

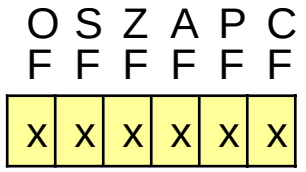
	O	S	Z	A	P	C
	F	F	F	F	F	F
mul imul	x	?	?	?	?	x

	O	S	Z	A	P	C
	F	F	F	F	F	F
div idiv	?	?	?	?	?	?

Arithmetic instructions (4)

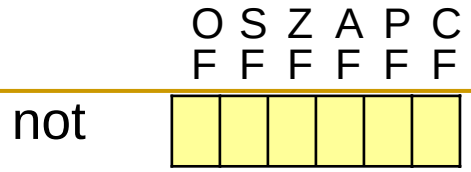
syntax	example	description
inc *r/m*	inc %eax	increment by 1. %eax++;
dec *r/m*	dec %eax	decrement by 1. %eax--;
neg *r/m*	neg %eax	2's complement negation. %eax = -%eax;

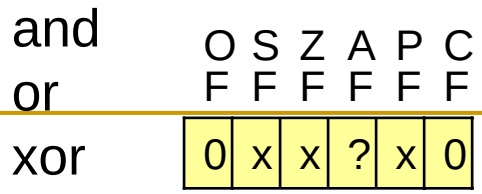




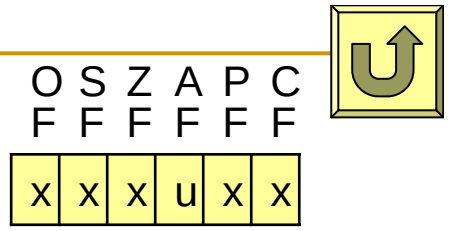
Logical instructions

syntax	example	description
not *r/m*	notl %eax	%eax = ~%eax;
and *imm*, *r/m* and *r*, *r/m* and *r/m*, *r*	andl $0xFFF, %eax andl %ebx, 4(%ebp) andl %eax, %ebx	%eax &= 0xFFF; *(%ebp+4) &= %eax; %ebx &= %eax;
or *op1*, *op2* (same as and)	or %eax, %ebx	%ebx |= %eax;
xor *op1*, *op2* (same as and)	xor %eax, %ebx	%ebx ^= %eax;

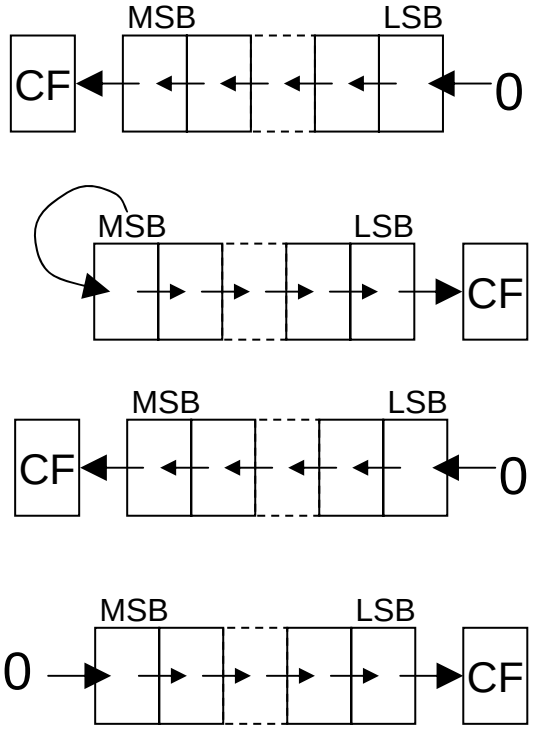




Shift instructions

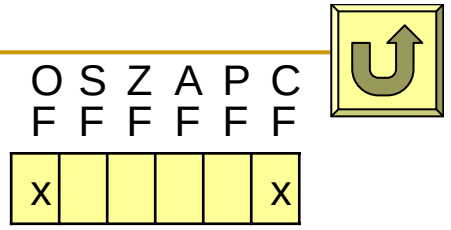


syntax	example	description
sal *r/m* sal *imm8*, *r/m* sal *%cx*, *r/m*	sall %eax sall $2, %eax sall %cl, %eax	shift arithmetic left.
sar *op1* [, *op2*] (same as sal)	sarl $2, %eax sarl %cl, %eax	shift arithmetic right.
shl *op1* [, *op2*] (same as sal)	shll $2, %eax shll %cl, %eax	shift (logical) left.
shr *op1* [, *op2*] (same as sal)	shrl $2, %eax shrl %cl, %eax	shift (logical) right.

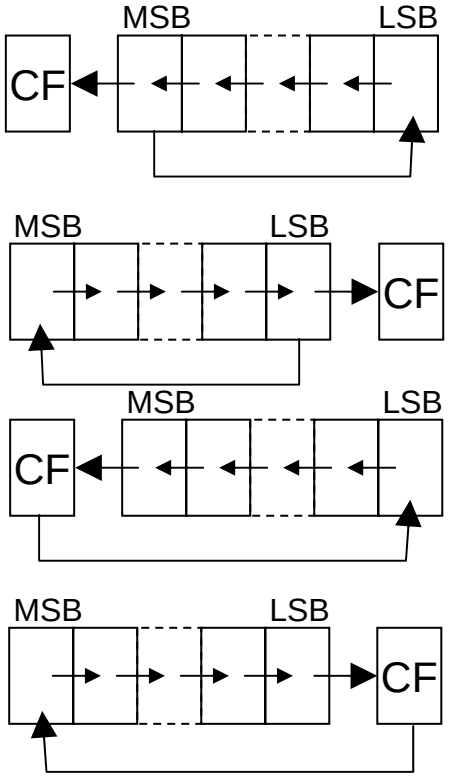


sal and shl do the same behavior.

Rotate instructions

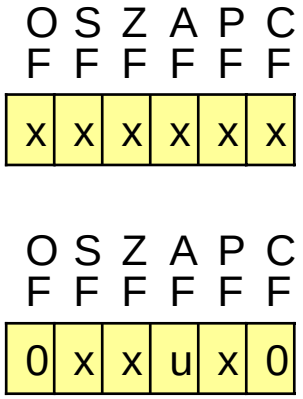


syntax	example	description
rol *r/m* rol *imm8*, *r/m* rol *%cx*, *r/m*	roll %eax roll $2, %eax roll %cl, %eax	rotate left.
ror *op1* [, *op2*] (same as sal)	rorl $2, %eax rorl %cl, %eax	rotate right.
rcl *op1* [, *op2*] (same as sal)	rcll $2, %eax rcll %cl, %eax	rotate through carry left.
rcr *op1* [, *op2*] (same as sal)	rcrl $2, %eax rcrl %cl, %eax	rotate through carry right.



Compare,test instructions

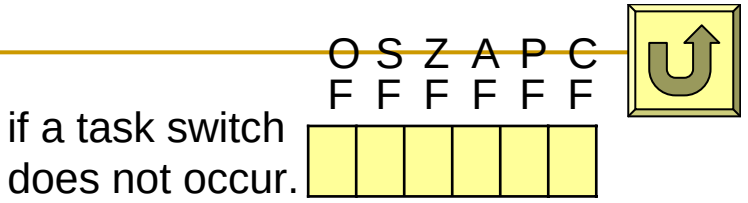
syntax	example	description
cmp *imm*, *r/m*	cmpl $123, %ebx	
cmp *r*, *r/m*	cmpl %ecx, %ebx	
cmp *r/m*, *r*	cmpl 4(%ebp), %ebx	
test *imm*, *r/m*	testl $123, %ebx	
test *r*, *r/m*	testl %ecx, %ebx	
test *r/m*, *r*	testl 4(%ebp), %ecx	



- cmp is the same as sub, but cmp does not modify the destination operand. cmp modifies only %eflags.
- test is the same as and, but test does not modify the destination operand. test modifies only %eflags.

cmp ≒ sub
test ≒ and

jmp: jump (1): syntax



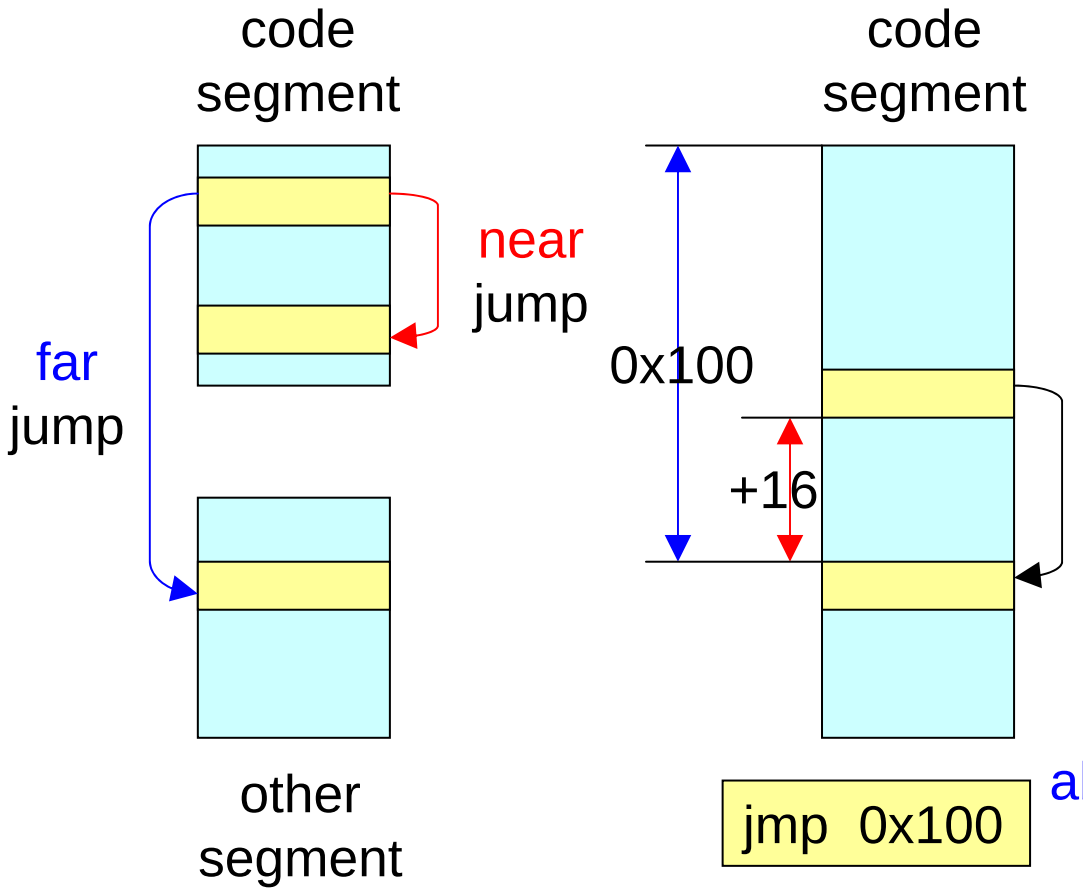
- jmp performs an unconditional jump.

Actually, GNU as always produces imm32 for these statements.

syntax	example	description
jmp *imm8*	jmp 0x10	short, relative, direct jump
	jmp _foo	(not prefixed with '$')
jmp *imm16(32)*	jmp 0x1000	near, relative, direct jump
	jmp _foo	(not prefixed with '$')
jmp *r/m*	jmp *%eax	near, absolute, indirect jump
jmp *imm16:16(32)*	ljmp $0x10, $0x100	far, absolute, direct jump
jmp *m16:16(32)*	ljmp *4(%ebp)	far, absolute, indirect jump

Far jumps in protected-mode (e.g., used for a task switch) is not explained here for simplicity.

jmp: jump (2): terminology



In x86, near direct jump implies relative jumps.

direct jump

```
jmp  0x1000
```

indirect jump

```
jmp  *(%ebx)
```

```
jmp  0x100
```

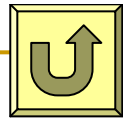
absolute jump

```
jmp  +16
```

relative jump

near jump with the range from -128 to 127 is called short jump.

jmp: jump (3): syntax anomalies



- dollar ($) and asterisk (*)
  - Relative direct jump operands are not prefixed by dollar ($), but far absolute long jump operands must be prefixed by dollar ($).
  - Absolute jump operands are prefixed by asterisk (*).

```
jmp  _foo
ljmp $0x10, $0x100
jmp  *%eax
```

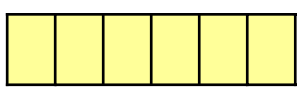
not prefixed by '$'.
prefixed by '$'.
prefixed by '*'.

- Mnemonic suffix (b, w, l)
  - Near relative direct jump instructions cannot be suffixed by 'b', 'w', 'l'.
    - 'info as' says "*jump instructions are always optimized to use the smallest possible displacements.*"

```
jmp    _foo
jmpb   _foo
jmpw   _foo
jmpl   _foo
```

OK
NG
NG
NG

jcc: conditional jump (1)



- jcc checks %eflags (CF, OF, PF, SF, ZF) or %ecx and, if the condition is met, jcc performs a jump.

syntax	example	description
jcc *rel*	jg _foo	jump if condition is met.

- jcc are often used with cmp.

```
cmpl   $0, 8(%ebp)
jg     L2
```

jump if 8(%ebp) > 0

- jcc does not suport far jumps.
  - To perform far jumps, combine jcc and jmp as follows.

NG

```
jz  $0x10, $0x100
```

```
jnz   1f
ljmp  $0x10, $0x100
1:
```

OK

jcc: conditional jump (2)

usage pattern

```
cmp op2, op1
jcc   rel
```

- conditional jumps for signed integers.

instruction	condition	flags	description
jg jnle	op2 > op1 !(op2 <= op1)	ZF==0 && SF==OF	greater not less nor equal
jge jnl	op2 >= op1 !(op2 < op1)	SF==OF	greater or equal not less
jle jng	op2 <= op1 !(op2 > op1)	ZF==1 || SF!=OF	less or equal not greater
jl jnge	op2 < op1 !(op2 >= op1)	SF!=OF	less not greater nor equal

- 'less' and 'greater' used for signed integers,
- while 'above' and 'below' used for unsigned integers.

jcc: conditional jump (3)

usage pattern

```
cmp op2, op1
jcc    rel
```

- conditional jumps for unsigned integers.

instruction	condition	flags	description
ja jnbe	op2 > op1 !(op2 <= op1)	CF==0 && ZF==0	above not below nor equal
jae jnb	op2 >= op1 !(op2 < op1)	CF==0	above or equal not below
jbe jna	op2 <= op1 !(op2 > op1)	CF==1 && ZF==1	below or equal not above
jb jnae	op2 < op1 !(op2 >= op1)	CF==1	below not above nor equal

- 'less' and 'greater' used for signed integers,
- while 'above' and 'below' used for unsigned integers.

jcc: conditional jump (4)

- conditional jumps for counters.

syntax	condition	description
jcxz rel8	%cx == 0	jump if %cx is zero
jecxz rel8	%ecx == 0	jump if %ecx is zero

- jcxz and jecxz only supports short jumps.
  - So, the jump range is -128 to 127.

NG

```
    jecxz  _foo
.skip 256
_foo:
```

```
%  gcc -c foo.s
foo.s:1: Error: value of 256 too large
                    for field of 1 bytes at 1
%
```

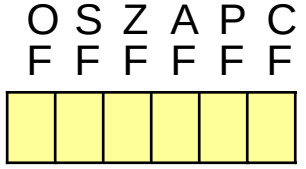
jcc: conditional jump (5)

- conditional jumps for flags.

instruction	flags	description
jc	CF==1	carry
jnc	CF==0	not carry
jo	OF==1	overflow
jno	OF==0	not overflow
js	SF==1	sign
jns	SF==0	not sign

instruction	flags	description
je	ZF==1	equal
jne	ZF==0	not equal
jz	ZF==1	zero
jnz	ZF==0	not zero
jpe	PF==1	parity even
jpo	PF==0	parity odd
jp	PF==1	parity
jnp	PF==0	not parity

loop (1)



- Loop if %ecx (or %cx) is not zero.
- Loop only supports short jumps.

syntax	example	description
loop *rel8*	loopl _foo	%(e)cx--; jump if %(e)cx!=0
loope *rel8* loopz *rel8*	loopel _foo loopzw _foo	%(e)cx--; jump if %(e)cx!=0 && ZF=1
loopne *rel8* loopnz *rel8*	loopnel _foo loopnzw _foo	%(e)cx--; jump if %(e)cx!=0 && ZF=0

loop (2)

- Example 1:

```
.globl _main
_main:
  movl  $10, %ecx
  movl  $0,  %eax
1:
  addl  %ecx, %eax
  loopnel 1b
```

jump if %ecx != 0

```
do {
   ....
} while (--%ecx != 0)
```

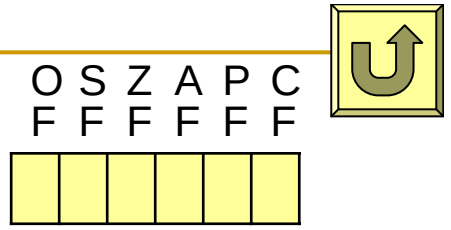
- Example 2:

```
.globl _main
_main:
  movl  $10, %ecx
  movl  $0,  %eax
1:
  addl  %ecx, %eax
  cmp   $19, %eax
  loopnel 1b
```

jump if %ecx != 0 && 19 != %eax

```
do {
   ....
} while (--%ecx != 0
          && 19 != %eax)
```

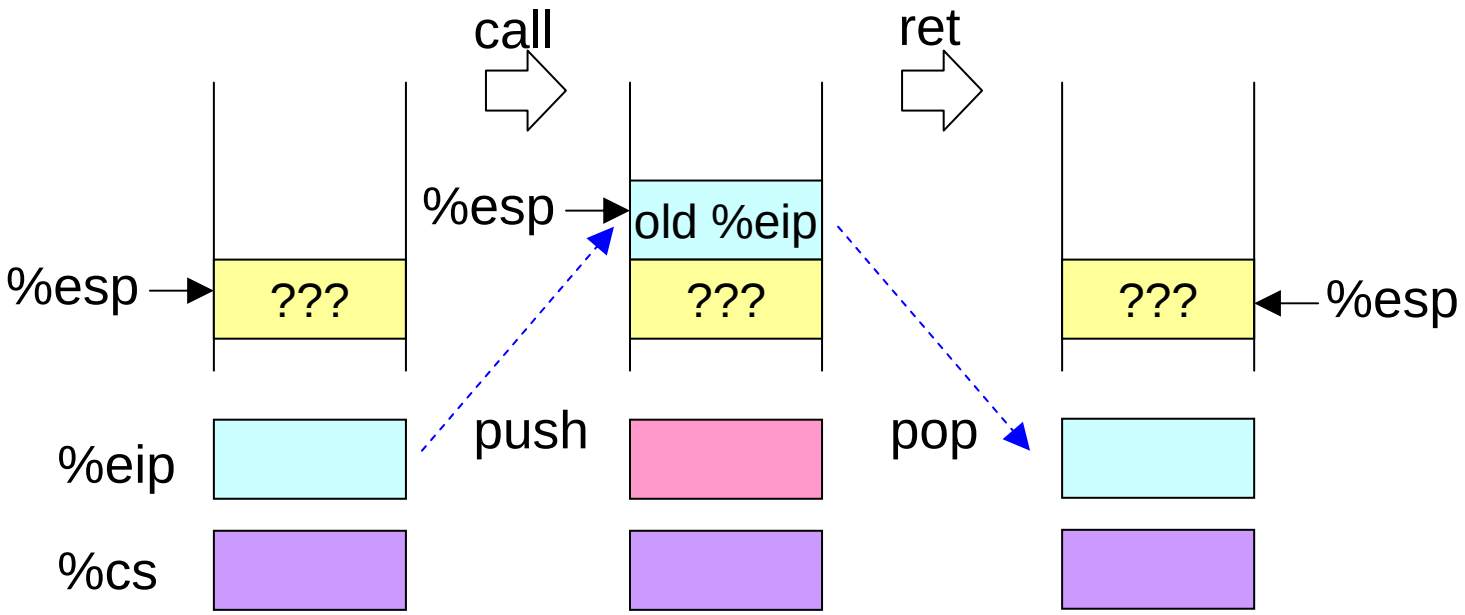
call, ret, enter, leave (1)



syntax	example	description
call *rel16/32*	call _foo	near, relative, direct call
call *r/m16/32*	call *%eax	near, absolute, indirect call
call *ptr16:16/32*	lcall $0x10, $0x100	far, absolute, direct call
call *m16:16/32*	lcall *4(%ebp)	far, absolute, indirect call
ret	ret	far/near return
ret *imm16*	ret $10	return and %esp += imm16

call, ret, enter, leave (2)

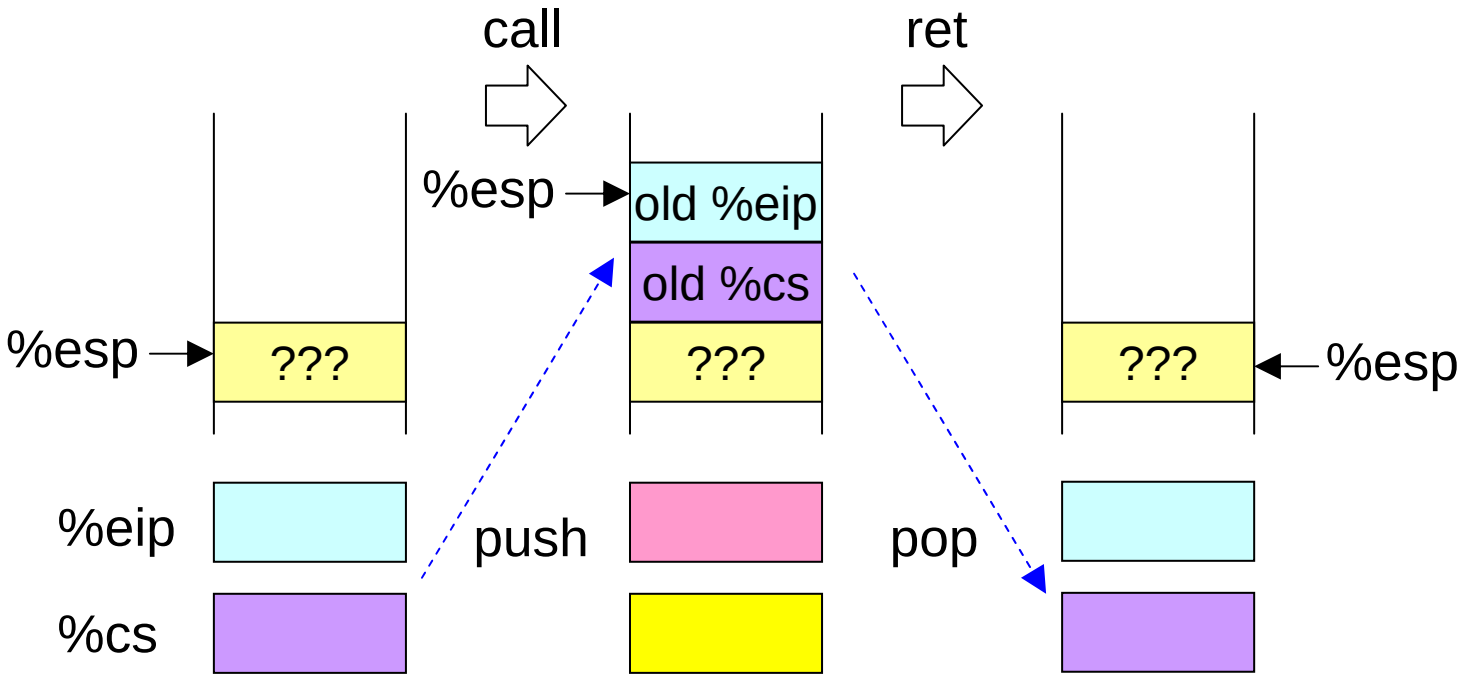
- stack usage on near call/ret.



  - The value of %eip pushed on the stack is also referred to as "return address".
  - The value of %cs is unchanged.

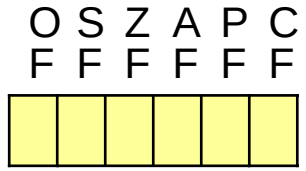
call, ret, enter, leave (3)

- stack usage on far call/ret



  - The value of %cs is also pushed on the stack, since the far jump call loads a new value to %cs.

call, ret, enter, leave (4)



- enter sets up a stack frame.
  - *imm8* is the nesting level. *imm16* is the size of the stack frame.
  - GCC does not use enter, probably because enter is much slower.
- leave releases the stack frame.

syntax	example	description
enter *imm8*, *imm16*	enter $0, $8	
leave	leavew leave	%sp=%bp; pop %bp %esp=%ebp; pop %ebp

enter $0, $8 =

```
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
```

software interruption (1)

1-byte long

syntax	example	description
int 3	int3 int $3	trap to debugger.
int *imm8*	int $4	software interrupt.
into	into	interrupt 4 if OF=1.
iret	iret	interrupt return.
bound *r16, m16&16* bound *r32, m32&32*	bound %dx, 4(%bp) bound %edx, 4(%ebp)	check array index against bounds.
ud2	ud2	raise invalid opcode exception.

int
bound
ud2

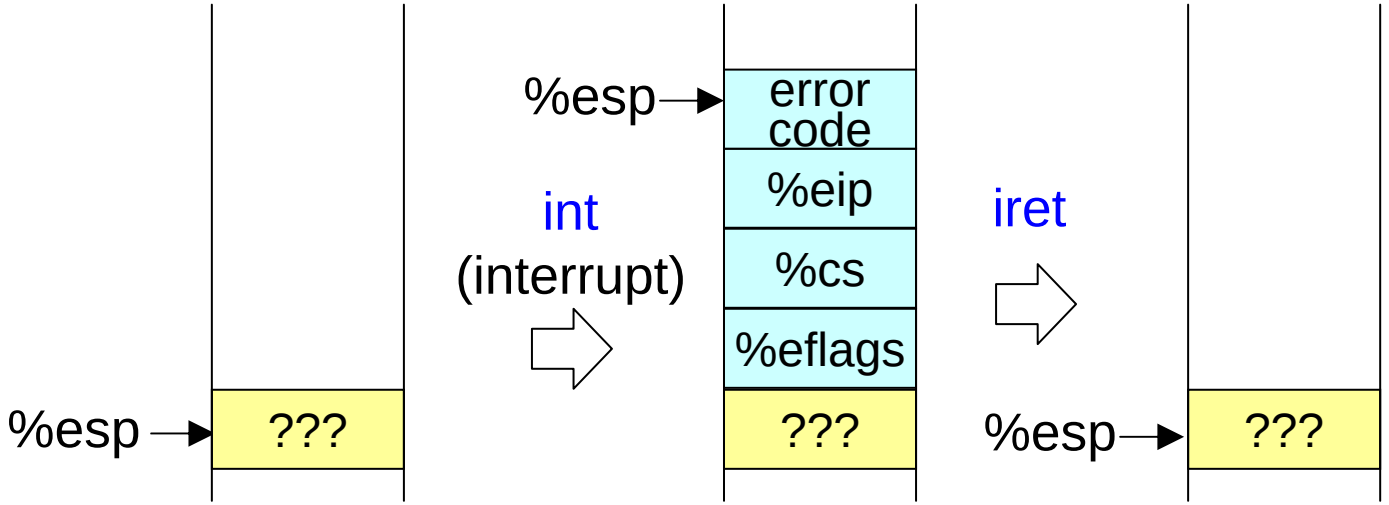
OF	SF	ZF	AF	PF	CF

iret

OF	SF	ZF	AF	PF	CF
x	x	x	x	x	x

software interruption (2)

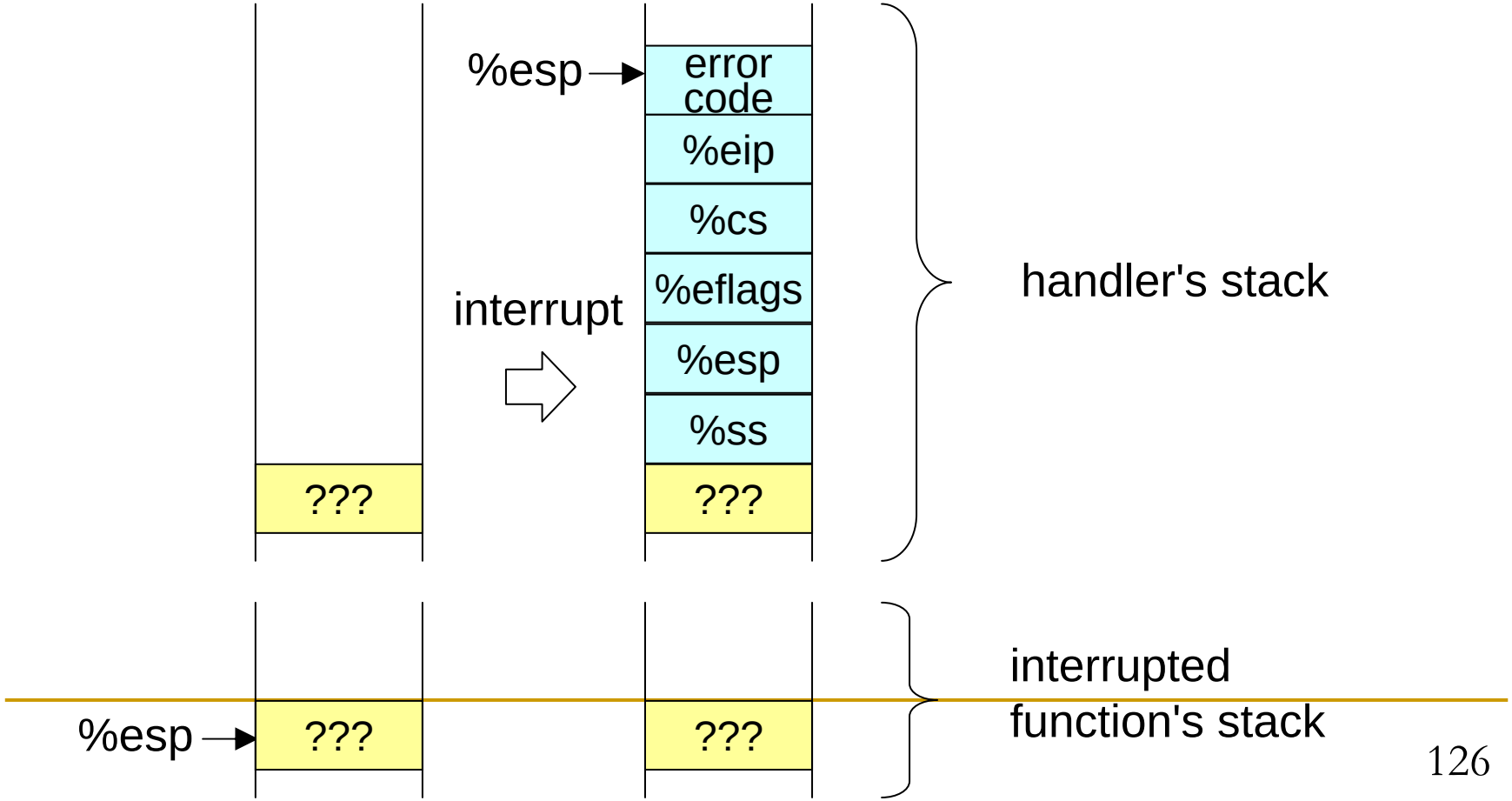
- stack usage on transfers to interrupt handlers.
  - with no privilege-level change (i.e., with no stack switch).



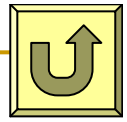
  - "error code" is optional, as some interrupts or faults do not push an error code.

software interruption (3)

- stack usage on transfers to interrupt handlers.
  - with privilege-level change (i.e., with stack switch).



software interruption (4)



foo.c

```
#include <stdio.h>
int main (void) {
    asm ("int3");
}
```

```
% gcc -g foo.c
% gdb ./a.exe
(gdb) run
Program received signal SIGTRAP,
Trace/breakpoint trap.
main () at foo.c:4
4       }
(gdb)
```

foo2.c

```
#include <stdio.h>
int main (void) {
    asm ("ud2");
}
```

```
% gcc -g foo2.c
% gdb ./a.exe
(gdb) run
Program received signal SIGILL,
     Illegal instruction.
main () at foo.c:3
3   asm ("ud2");
(gdb)
```

hlt (1)

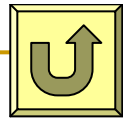
- hlt stops instruction execution.

syntax	example	description
hlt	hlt	stops instruction execution.

- An interrupt, NMI, or a reset will resume execution.
  - So, often combined with jmp.

```
1:
    hlt
    jmp 1b
```

hlt (2)

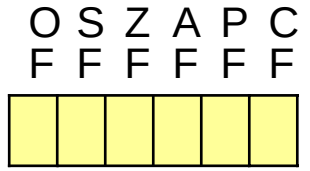


- hlt is a privileged instruction, so hlt cannot be executed in user space.

```
.global _main
_main:
    hlt
```

```
% gcc -g foo.s
% gdb ./a.exe
(gdb) run
Program received signal SIGILL,
     Illegal instruction.
main () at foo.s:3
3   hlt
(gdb)
```

in, out (1)

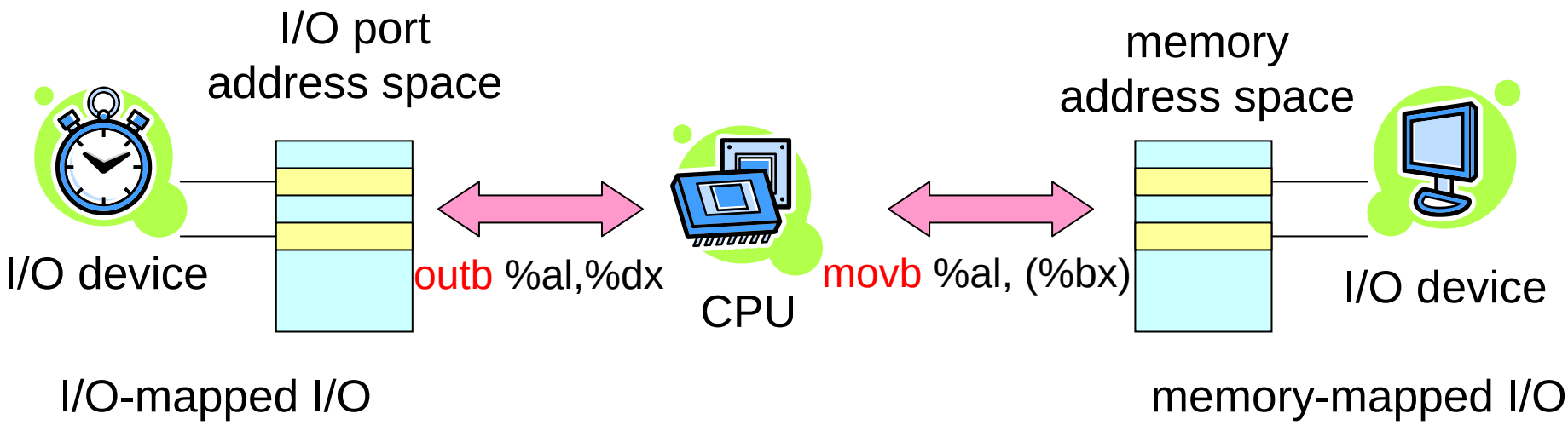


- in reads a data from the I/O port to %(e)ax or %al.
- out writes a data in %(e)ax or %al to the I/O port.
- The I/O port can be specified by *imm8* or %dx.

syntax	example	description
in *imm8*, %al	inb $10, %al	
in *imm8*, %(e)ax	inw $10, %ax	
in %dx, %al	inb %dx, %al	
in %dx, %(e)ax	inl %dx, %eax	
out %al, *imm8*	outb %al, $10	
out %(e)ax, *imm8*	outw %ax, $10	
out %al, %dx	outb %al, %dx	
out %(e)ax, %dx	outl %eax, %dx	

in, out (2)

- I/O-mapped I/O vs., memory-mapped I/O.
  - In I/O-mapped I/O, I/O devices can be accessed via a special address called I/O port, which is distinct from memory address.
  - In memory-mapped I/O, I/O devices can be accessed via a memory address.



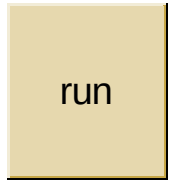
in, out (3)

Run

```
.code16
.text
  movb $0xED, %al
  movw $0x60, %dx
  outb %al, %dx      #  send command "set/reset LED"
  movw $0x64, %dx
wait:
  inb %dx, %al
  test $2, %al
  jnz wait           #  wait until IBF=0
  movw $0x60, %dx
  movb $2, %al
  outb %al, %dx      #   specify "Num Lock"
```

- The code fragment sets num-lock LED in real-mode.
- In most OSes, direct access to I/O port by user is not allowed.
- In Linux, the system calls like ioperm and iopl allow you to access I/O port.
  - Use them at your own risk! Wrong access to I/O port can cause your computer to crash.
- in/out instructions are usually used in embedded systems and device drivers.

Example of memory-mapped I/O




```
.code16
.text
movw $0xB800, %cx          # address of VRAM page 0
movw %cx,     %es
movw $340,    %bx          #  coord=(10, 3), 340=(80*(3-1)+10)*2
movb  $'A',   %es:(%bx)    #  write ASCII character 'A'
incw   %bx
movb  $0x0C,  %es:(%bx)    # write attribute byte (highlighted red)
```

- This code fragment writes 'A' at (10, 3) on the VGA display, by writing data to Video RAM (VRAM).
- VRAM is memory-mapped. So movw can be used to access VRAM.

Memory barrier (memory fence)

- Recent CPUs use out-of-order execution techniques.
- mfence, lfence, sfence cause a CPU to enforce an ordering constraint on memory operations.
  - Issued before and after the barrier instruction.

syntax	example	description
mfence	mfence	pentium4 and later
lfence	lfence	pentium4 and later
sfence	sfence	pentium3 and later

cf. sequence point in C.