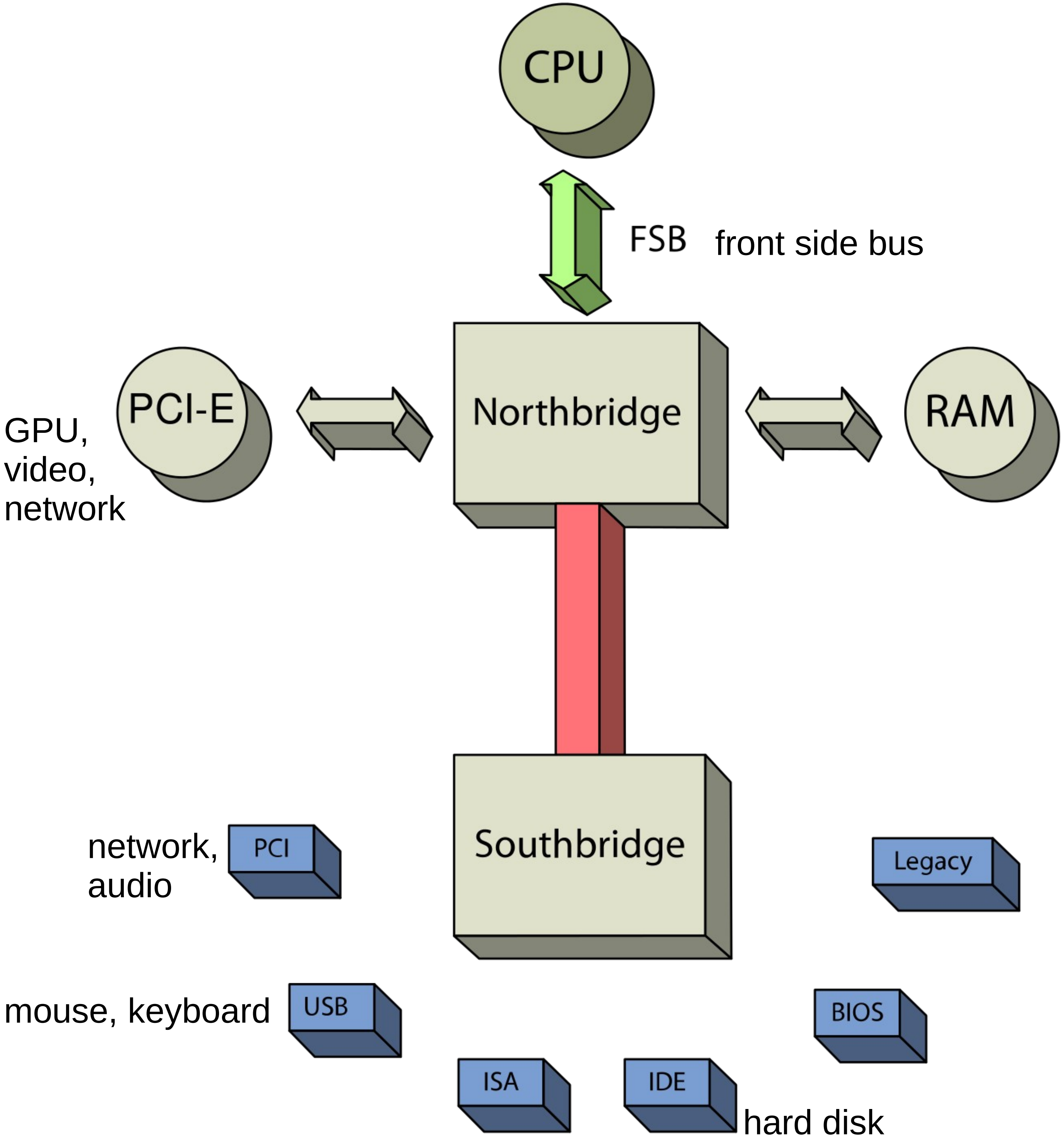
Intel 32-bit architecture

Also known as:

- IA-32
- x86
- x86-32



front side bus

GPU,
video,
network

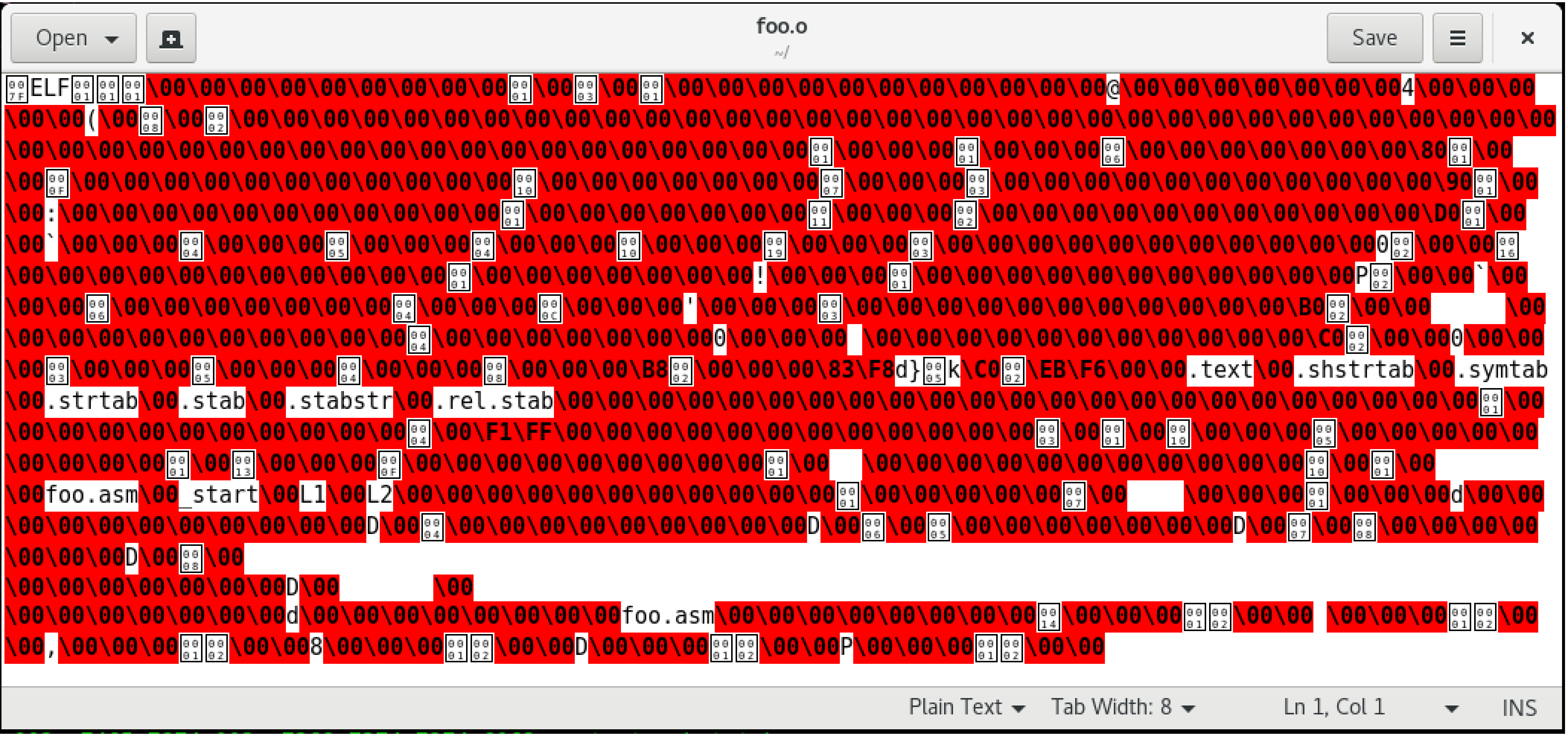
network,
audio

mouse, keyboard

hard disk

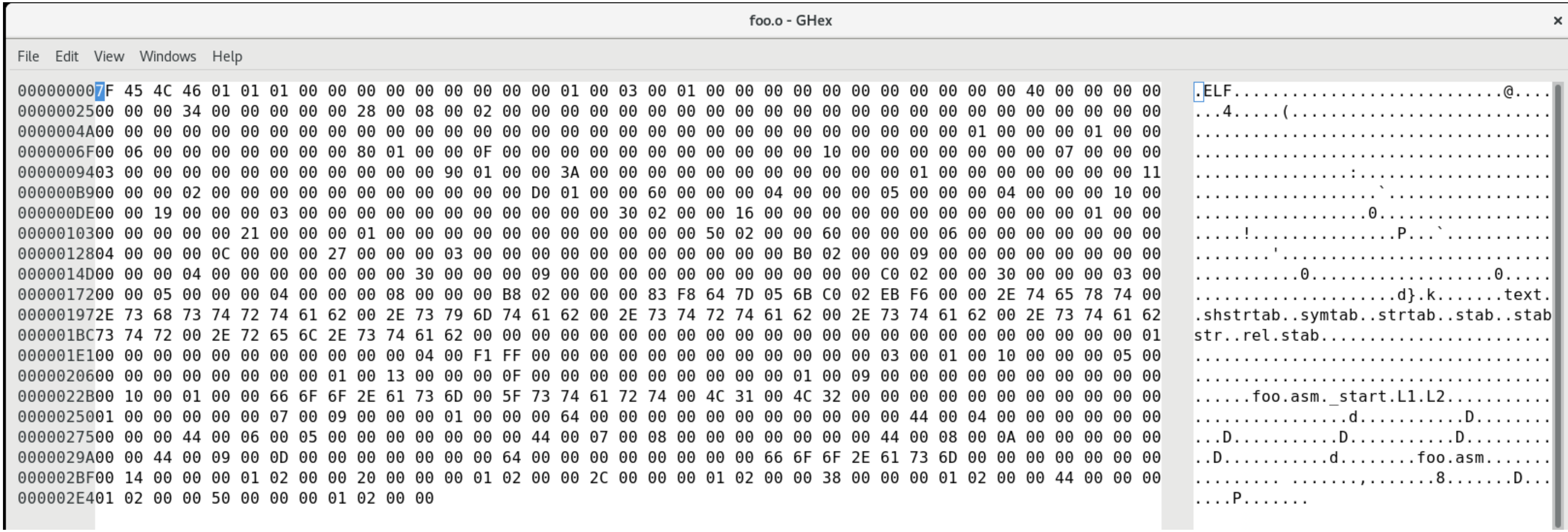
```
; A simple x86 program
mov eax, 2
L1:
cmp eax, 100
jge L2
imul eax, 2
jmp L1
L2:
```

- nasm -f elf -gstabs foo.asm
- ld -o foo -m elf_i386 foo.o

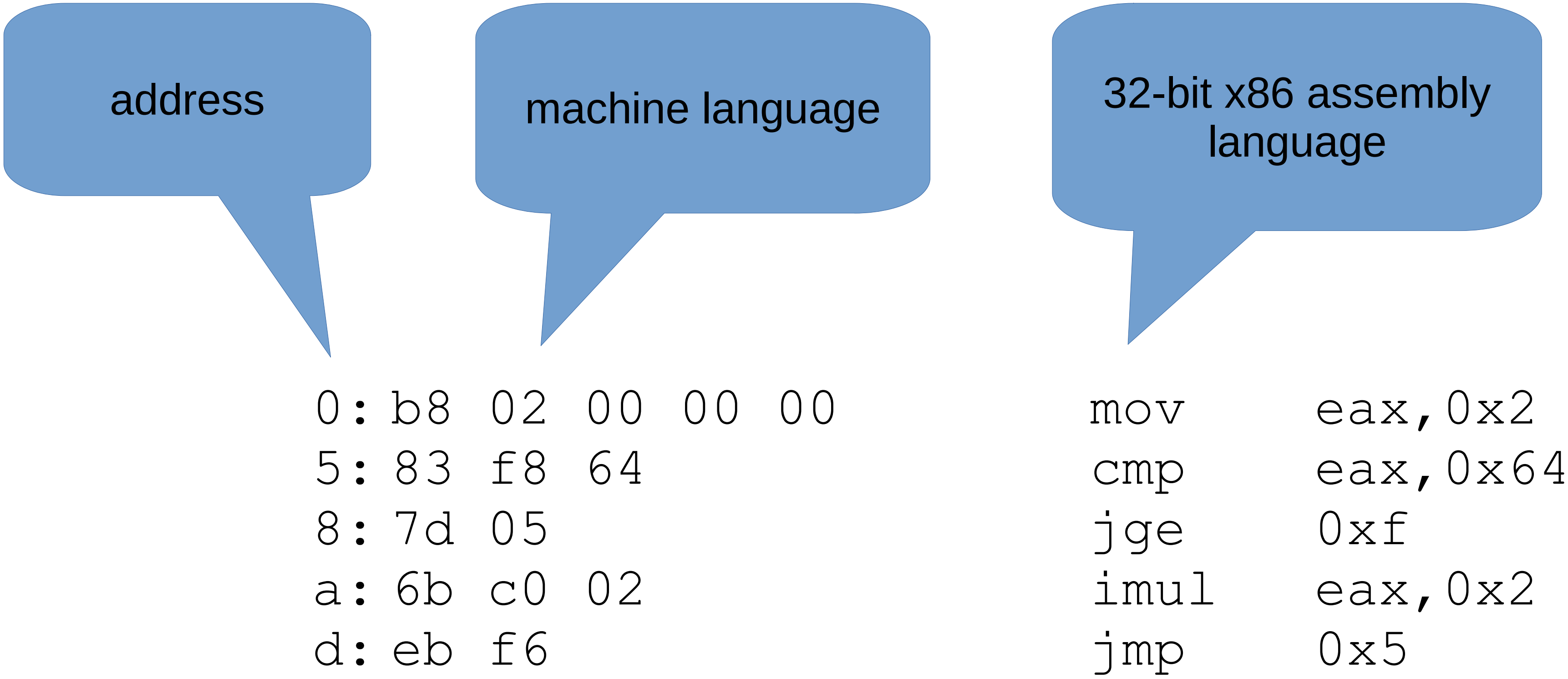


```
7FELF010101\00\00\00\00\00\00\00\00\0001\0003\0001\00\00\00\00\00\00\00\00\00\00\00@\00\00\00\00\00\00\004\00\00\00
\00\00(\0008\0002\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00
\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\0001\00\00\0001\00\00\0006\00\00\00\00\00\00\00\80 01\00
\000F\00\00\00\00\00\00\00\00\00\00\0010\00\00\00\00\00\00\0007\00\00\0003\00\00\00\00\00\00\00\00\00\00\00\9001\00
\00:\00\00\00\00\00\00\00\00\00\00\0001\00\00\00\00\00\00\0011\00\00\0002\00\00\00\00\00\00\00\00\00\00\00\D001\00
\00`\00\00\0004\00\00\0005\00\00\0004\00\00\0010\00\00\0019\00\00\0003\00\00\00\00\00\00\00\00\00\00\000 02\00\00\0016
\00\00\00\00\00\00\00\00\00\00\0001\00\00\00\00\00\00\00!\00\00\0001\00\00\00\00\00\00\00\00\00\00\00P02\00\00`\00
\00\0006\00\00\00\00\00\00\0004\00\00\000C\00\00\00'\00\00\0003\00\00\00\00\00\00\00\00\00\00\00\B002\00\00    \00
\00\00\00\00\00\00\00\00\00\0004\00\00\00\00\00\00\000\00\00\00 \00\00\00\00\00\00\00\00\00\00\00\C002\00\000\00\00
\0003\00\00\0005\00\00\0004\00\00\0008\00\00\00\B802\00\00\00\83\F8d}05k\C002\EB\F6\00\00.text\00.shstrtab\00.symtab
\00.strtab\00.stab\00.stabstr\00.rel.stab\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\0001\00
\00\00\00\00\00\00\00\00\00\0004\00\F1\FF\00\00\00\00\00\00\00\00\00\00\00\0003\0001\0010\00\00\0005\00\00\00\00\00\00
\00\00\00\0001\0013\00\00\000F\00\00\00\00\00\00\00\00\0001\00 \00\00\00\00\00\00\00\00\00\00\0010\0001\00
\00foo.asm\00_start\00L1\00L2\00\00\00\00\00\00\00\00\00\00\0001\00\00\00\00\0007\00   \00\00\0001\00\00\00d\00\00
\00\00\00\00\00\00\00\00\00D\0004\00\00\00\00\00\00\00\00\00D\0006\0005\00\00\00\00\00\00\00D\0007\0008\00\00\00\00
\00\00\00D\0008\00
\00\00\00\00\00\00\00D\00       \00
\00\00\00\00\00\00\00d\00\00\00\00\00\00\00\00foo.asm\00\00\00\00\00\00\00\0014\00\00\000102\00\00 \00\00\000102\00
\00,\00\00\000102\00\008\00\00\000102\00\00D\00\00\000102\00\00P\00\00\000102\00\00
```

```
01111111 01000101 01001100 01000110 00000001 00000001 00000001 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000001 00000000 00000011 00000000 00000001 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00110100 00000000 00000000 00000000 00000000 00000000 00101000 00000000
00001000 00000000 00000010 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000 00000001 00000000 00000000 00000000
00000110 00000000 00000000 00000000 00000000 00000000 00000000 00000000
10000000 00000001 00000000 00000000 00001111 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00010000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000111 00000000 00000000 00000000 00000011 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
10010000 00000001 00000000 00000000 00111010 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00010001 00000000 00000000 00000000 00000010 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
11010000 00000001 00000000 00000000 01100000 00000000 00000000 00000000
00000100 00000000 00000000 00000000 00000101 00000000 00000000 00000000
00000100 00000000 00000000 00000000 00010000 00000000 00000000 00000000
00011001 00000000 00000000 00000000 00000011 00000000 00000000 00000000
```



```
000000007F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 01 00 03 00 01 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00 00   .ELF............................@....
0000002500 00 00 34 00 00 00 00 00 28 00 08 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...4.....(...........................
0000004A00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 01 00 00   .....................................
0000006F00 06 00 00 00 00 00 00 00 80 01 00 00 0F 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00 07 00 00 00   .....................................
0000009403 00 00 00 00 00 00 00 00 00 00 00 90 01 00 00 3A 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 11   ................:....................
000000B900 00 00 02 00 00 00 00 00 00 00 00 00 00 D0 01 00 00 00 60 00 00 00 04 00 00 05 00 00 04 00 00 10 00 00 00 00   ...................`.................
000000DE00 00 19 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00 30 02 00 00 16 00 00 00 00 00 00 00 00 00 00 00 01 00 00   ..................0..................
0000010300 00 00 00 00 21 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 50 02 00 00 60 00 00 06 00 00 00 00 00 00 00 00   .....!...............P...`...........
0000012804 00 00 00 0C 00 00 00 27 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00 B0 02 00 09 00 00 00 00 00 00 00 00 00   ........'............................
0000014D00 00 00 04 00 00 00 00 00 00 00 30 00 00 00 09 00 00 00 00 00 00 00 00 00 00 00 C0 02 00 00 30 00 00 00 03 00   ...........0...................0.....
0000017200 00 05 00 00 00 04 00 00 00 08 00 00 00 B8 02 00 00 00 83 F8 64 7D 05 6B C0 02 EB F6 00 00 2E 74 65 78 74 00   .....................d}.k.......text.
000001972E 73 68 73 74 72 74 61 62 00 2E 73 79 6D 74 61 62 00 2E 73 74 72 74 61 62 00 2E 73 74 61 62 00 2E 73 74 61 62   .shstrtab..symtab..strtab..stab..stab
000001BC73 74 72 00 2E 72 65 6C 2E 73 74 61 62 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01   str..rel.stab........................
000001E100 00 00 00 00 00 00 00 00 00 00 04 00 F1 FF 00 00 00 00 00 00 00 00 00 00 00 03 00 01 00 10 00 00 00 05 00 00   .....................................
0000020600 00 00 00 00 00 00 00 01 00 13 00 00 00 0F 00 00 00 00 00 00 00 00 01 00 09 00 00 00 00 00 00 00 00 00 00 00   .....................................
0000022B00 10 00 01 00 00 66 6F 6F 2E 61 73 6D 00 5F 73 74 61 72 74 00 4C 31 00 4C 32 00 00 00 00 00 00 00 00 00 00 00   ......foo.asm._start.L1.L2...........
0000025001 00 00 00 00 00 07 00 09 00 00 00 01 00 00 00 64 00 00 00 00 00 00 00 44 00 04 00 00 00 00 00 00 00 00 00 00   ................d.......D............
0000027500 00 00 44 00 06 00 05 00 00 00 00 00 00 44 00 07 00 08 00 00 00 00 00 44 00 08 00 0A 00 00 00 00 00 00 00 00   ...D..........D.........D............
0000029A00 00 44 00 09 00 0D 00 00 00 00 00 00 00 64 00 00 00 00 00 00 00 00 66 6F 6F 2E 61 73 6D 00 00 00 00 00 00 00   ..D...........d........foo.asm.......
000002BF00 14 00 00 00 01 02 00 00 20 00 00 00 01 02 00 00 2C 00 00 00 01 02 00 00 38 00 00 00 01 02 00 00 44 00 00 00   ......... .......,.......8.......D...
000002E401 02 00 00 50 00 00 00 01 02 00 00                                                                                  ....P.......
```



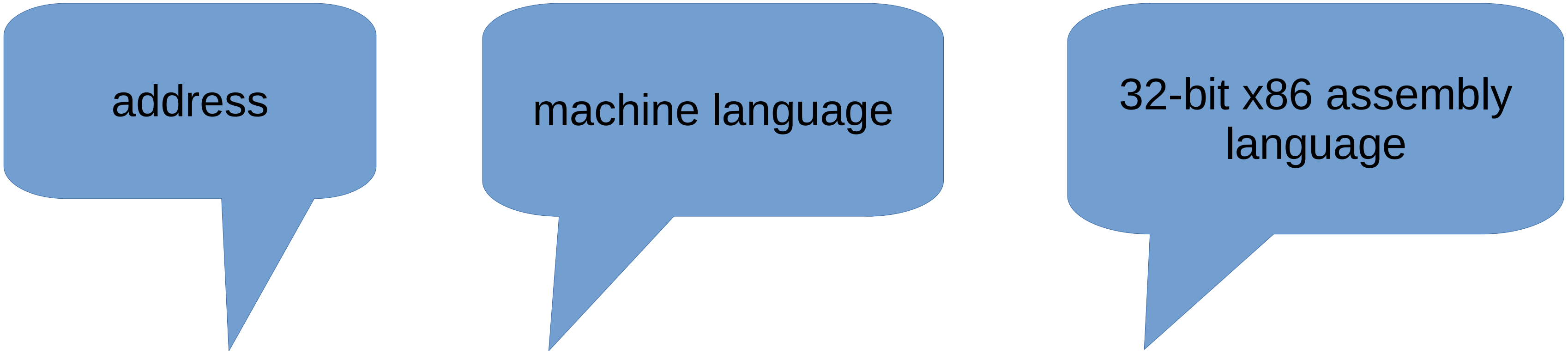
address

machine language

32-bit x86 assembly language

```
0: b8 02 00 00 00        mov     eax,0x2
5: 83 f8 64              cmp     eax,0x64
8: 7d 05                 jge     0xf
a: 6b c0 02              imul    eax,0x2
d: eb f6                 jmp     0x5
```

Object file disassembly. No symbols, addresses start from zero.



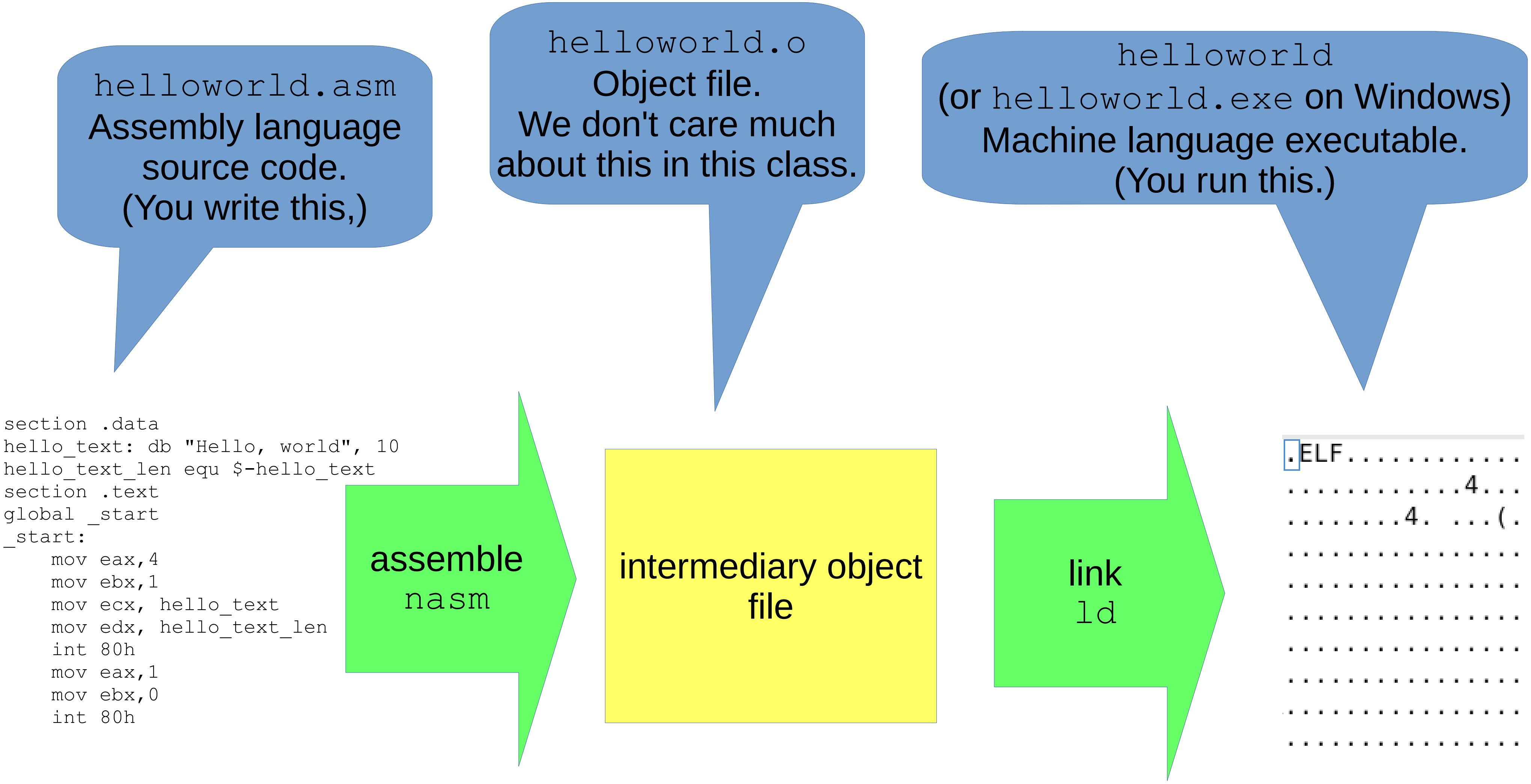
address

machine language

32-bit x86 assembly language

```
48060:   b8 02 00 00 00     mov    eax, 2
48065 <L1>:
48065:   83 f8 64           cmp    eax, 100
48068:   7d 05              jge    4806f <L2>
4806a:   6b c0 02           imul   eax, 2
4806d:   eb f6              jmp    48065 <L1>
4806f <L2>:
```

Executable file disassembly. Symbols are maintained, addresses are absolute.



`helloworld.asm`
Assembly language source code.
(You write this,)

`helloworld.o`
Object file.
We don't care much about this in this class.

`helloworld`
(or `helloworld.exe` on Windows)
Machine language executable.
(You run this.)

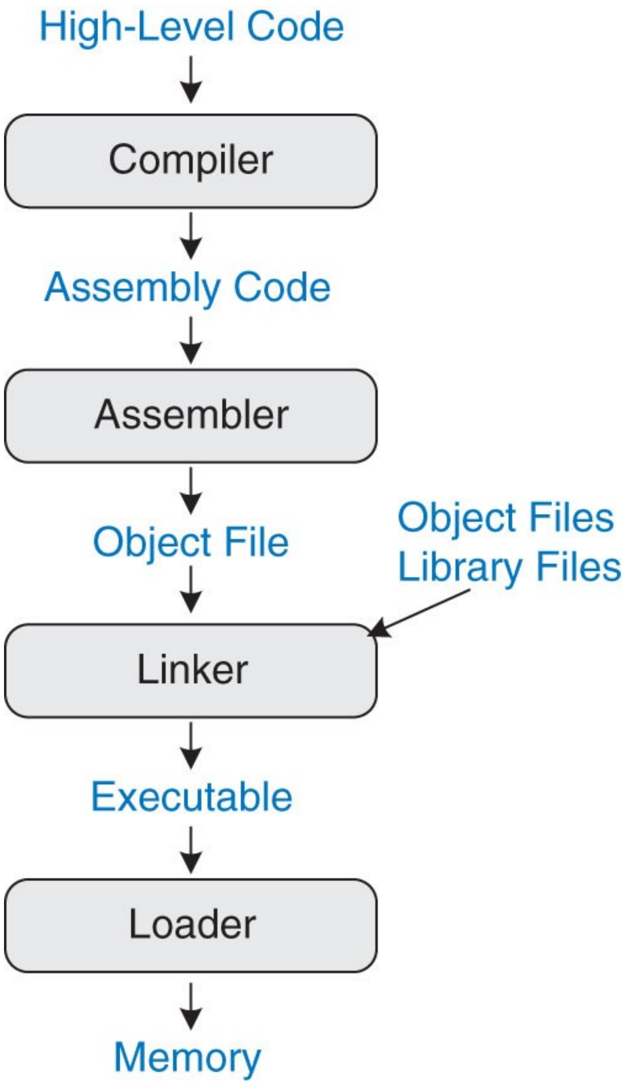
```
section .data
hello_text: db "Hello, world", 10
hello_text_len equ $-hello_text
section .text
global _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx, hello_text
    mov edx, hello_text_len
    int 80h
    mov eax,1
    mov ebx,0
    int 80h
```

assemble
`nasm`

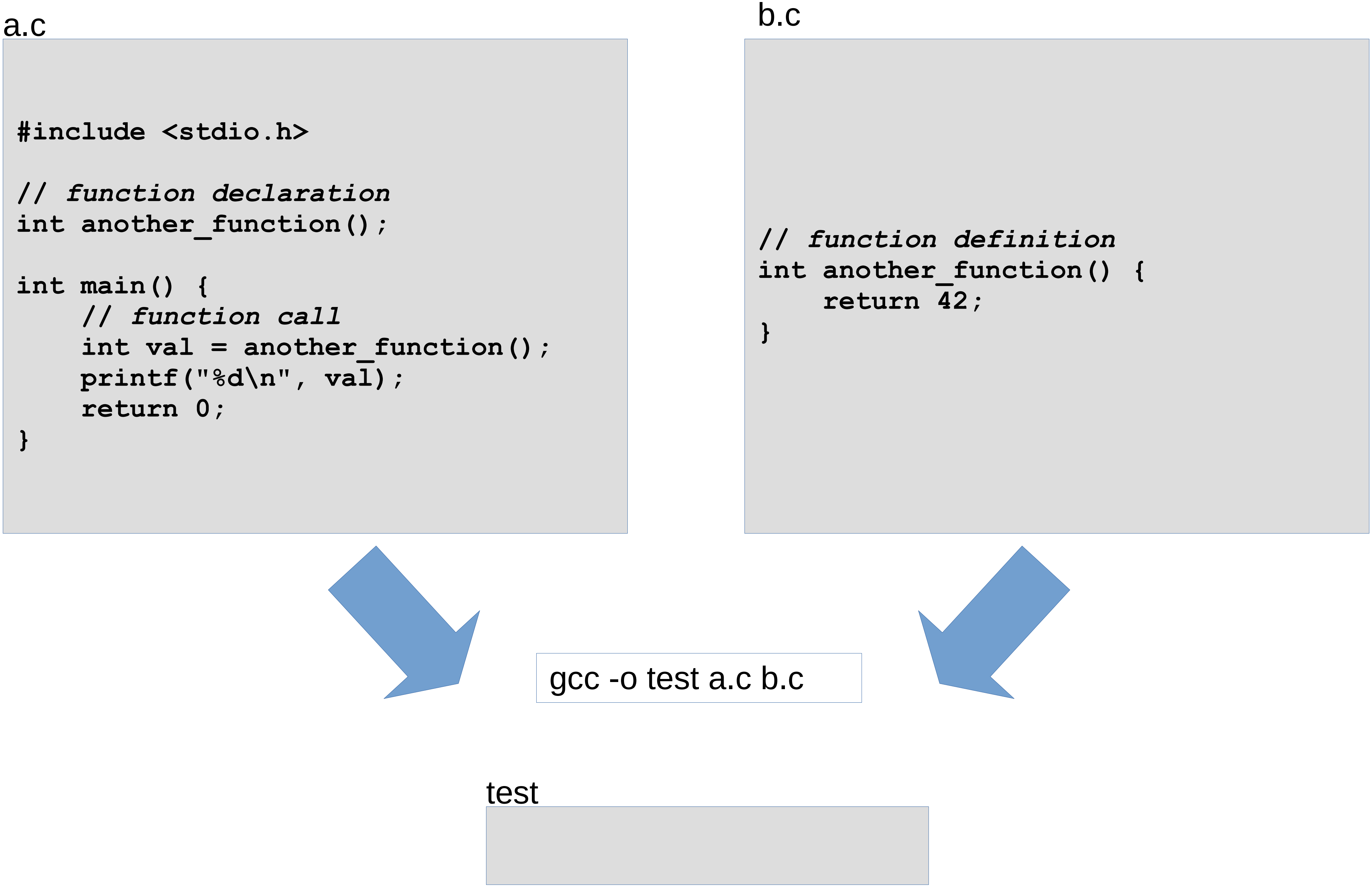
intermediary object file

link
`ld`

```
.ELF............
............4...
........4.   ...(.
................
................
................
................
................
................
................
```



Steps for translating and starting a program



a.c

```
#include <stdio.h>

// function declaration
int another_function();

int main() {
    // function call
    int val = another_function();
    printf("%d\n", val);
    return 0;
}
```

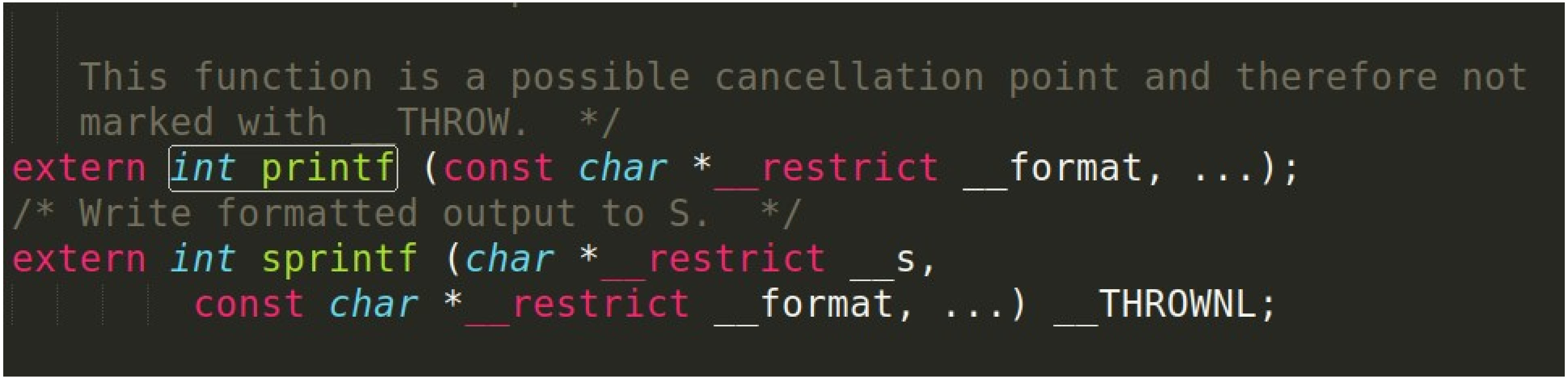
b.c

```
// function definition
int another_function() {
    return 42;
}
```

```
gcc -o test a.c b.c
```

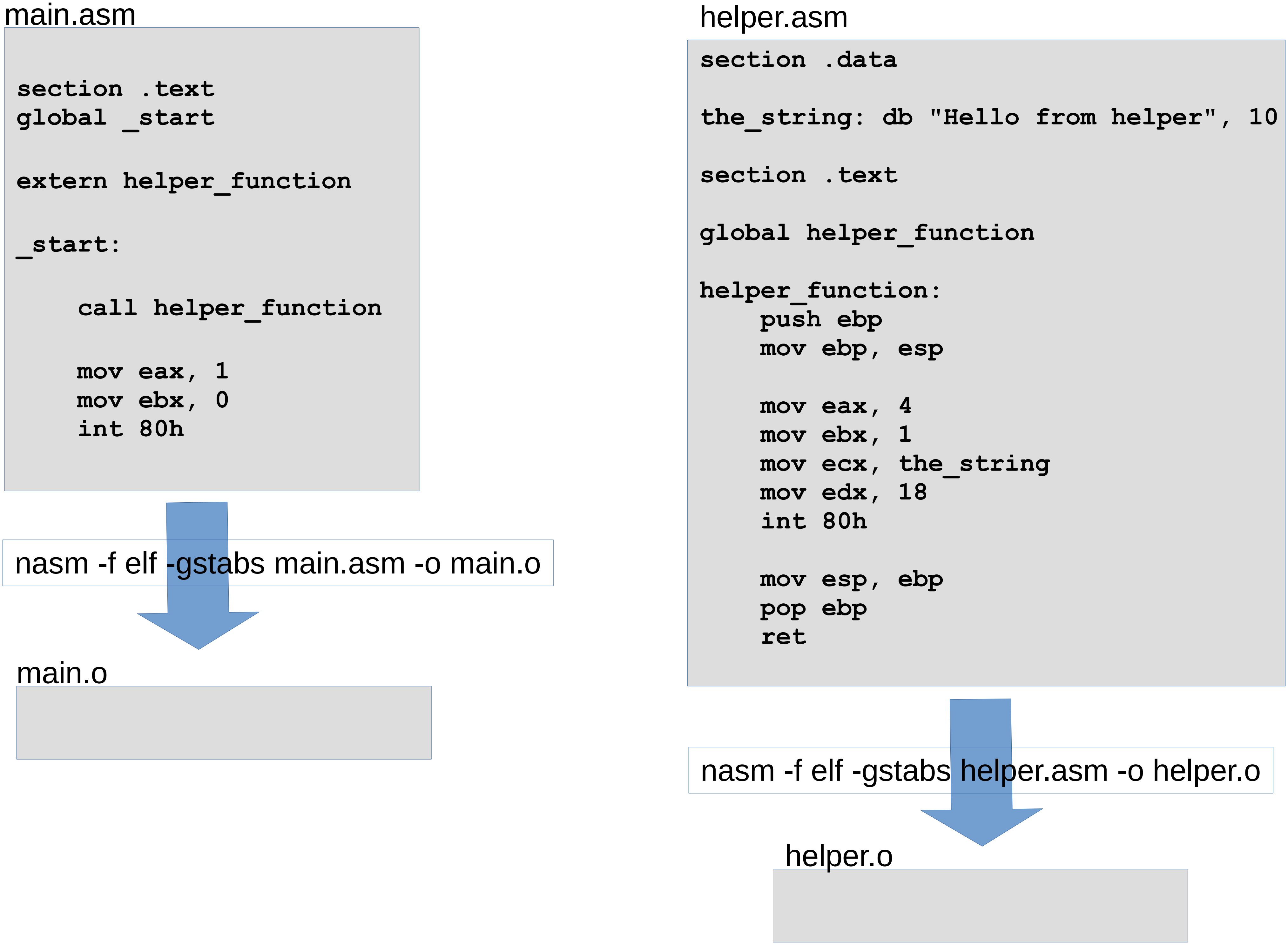
test

stdio.h



```
   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
extern int printf (const char *__restrict __format, ...);
/* Write formatted output to S.  */
extern int sprintf (char *__restrict __s,
		    const char *__restrict __format, ...) __THROWNL;
```

Header files usually contain declarations, not definitions.



main.asm

```
section .text
global _start

extern helper_function

_start:

    call helper_function

    mov eax, 1
    mov ebx, 0
    int 80h
```

```
nasm -f elf -gstabs main.asm -o main.o
```

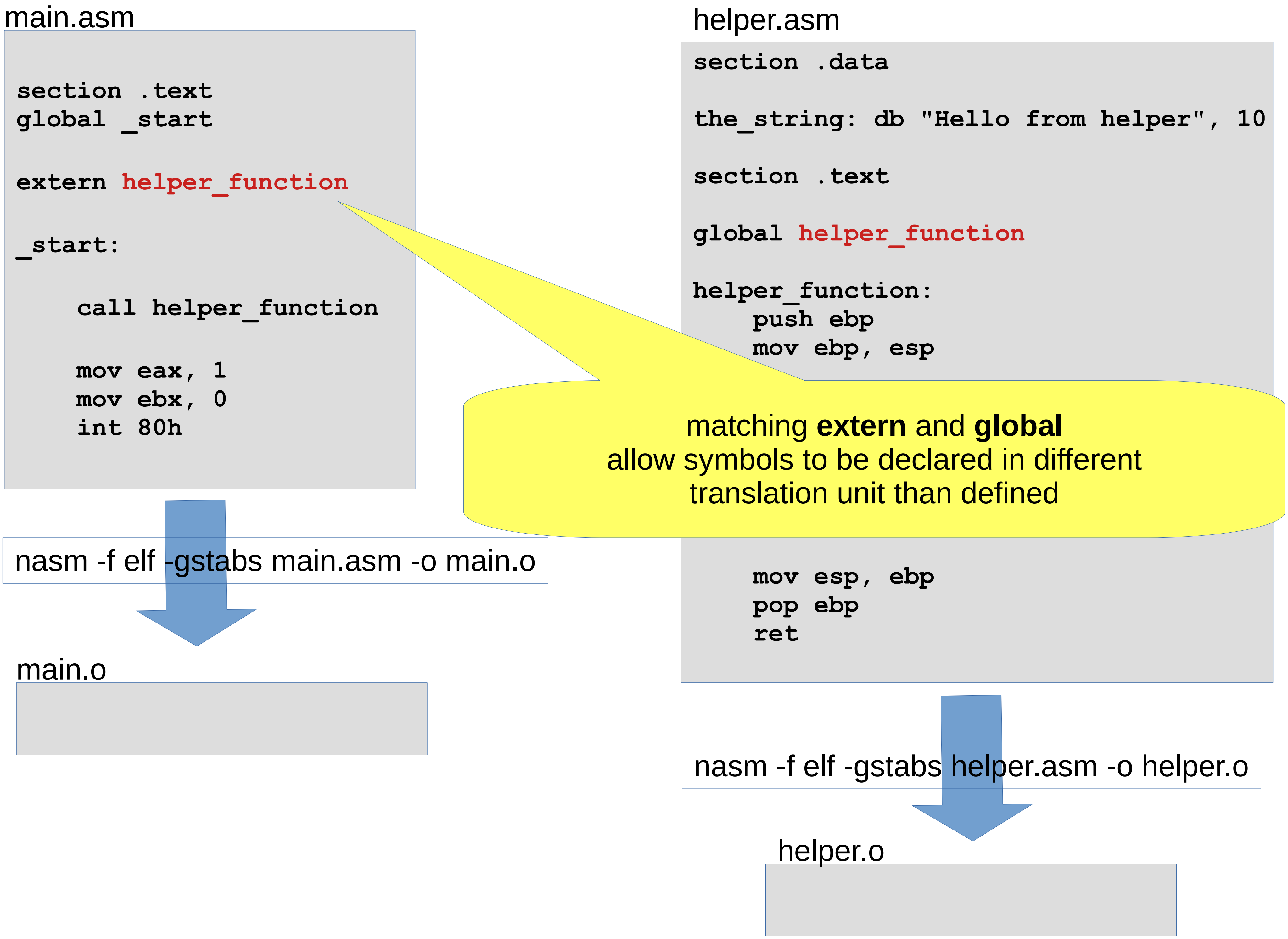
main.o

helper.asm

```
section .data

the_string: db "Hello from helper", 10

section .text

global helper_function

helper_function:
    push ebp
    mov ebp, esp

    mov eax, 4
    mov ebx, 1
    mov ecx, the_string
    mov edx, 18
    int 80h

    mov esp, ebp
    pop ebp
    ret
```

```
nasm -f elf -gstabs helper.asm -o helper.o
```

helper.o



main.asm

```
section .text
global _start

extern helper_function

_start:

    call helper_function

    mov eax, 1
    mov ebx, 0
    int 80h
```

```
nasm -f elf -gstabs main.asm -o main.o
```

main.o

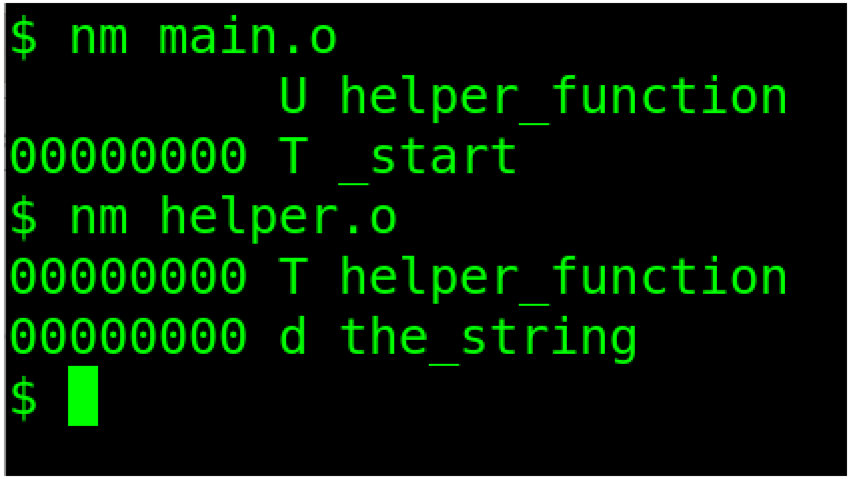
helper.asm

```
section .data

the_string: db "Hello from helper", 10

section .text

global helper_function

helper_function:
    push ebp
    mov ebp, esp

    mov esp, ebp
    pop ebp
    ret
```

matching **extern** and **global**
allow symbols to be declared in different
translation unit than defined

```
nasm -f elf -gstabs helper.asm -o helper.o
```

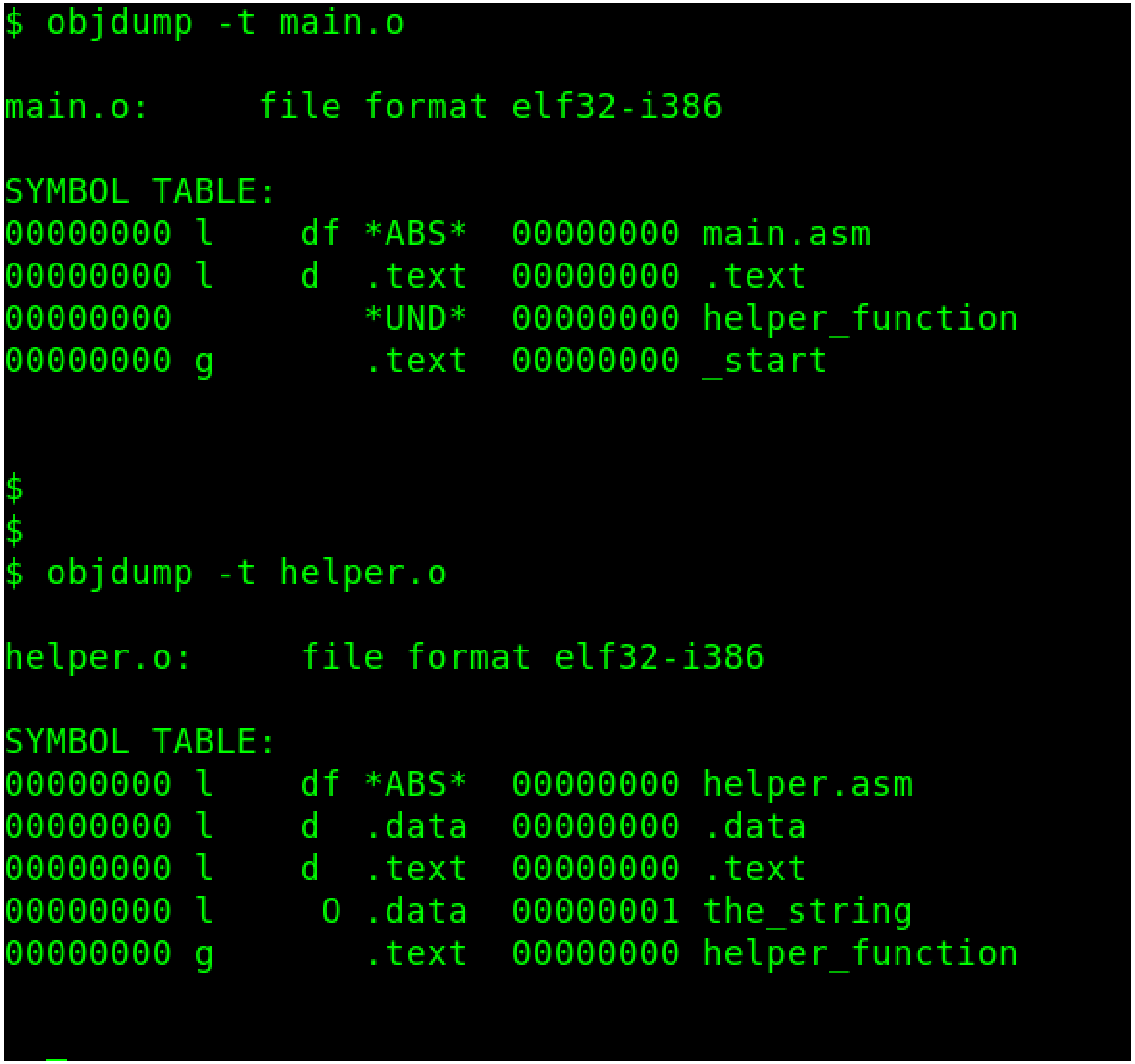
helper.o



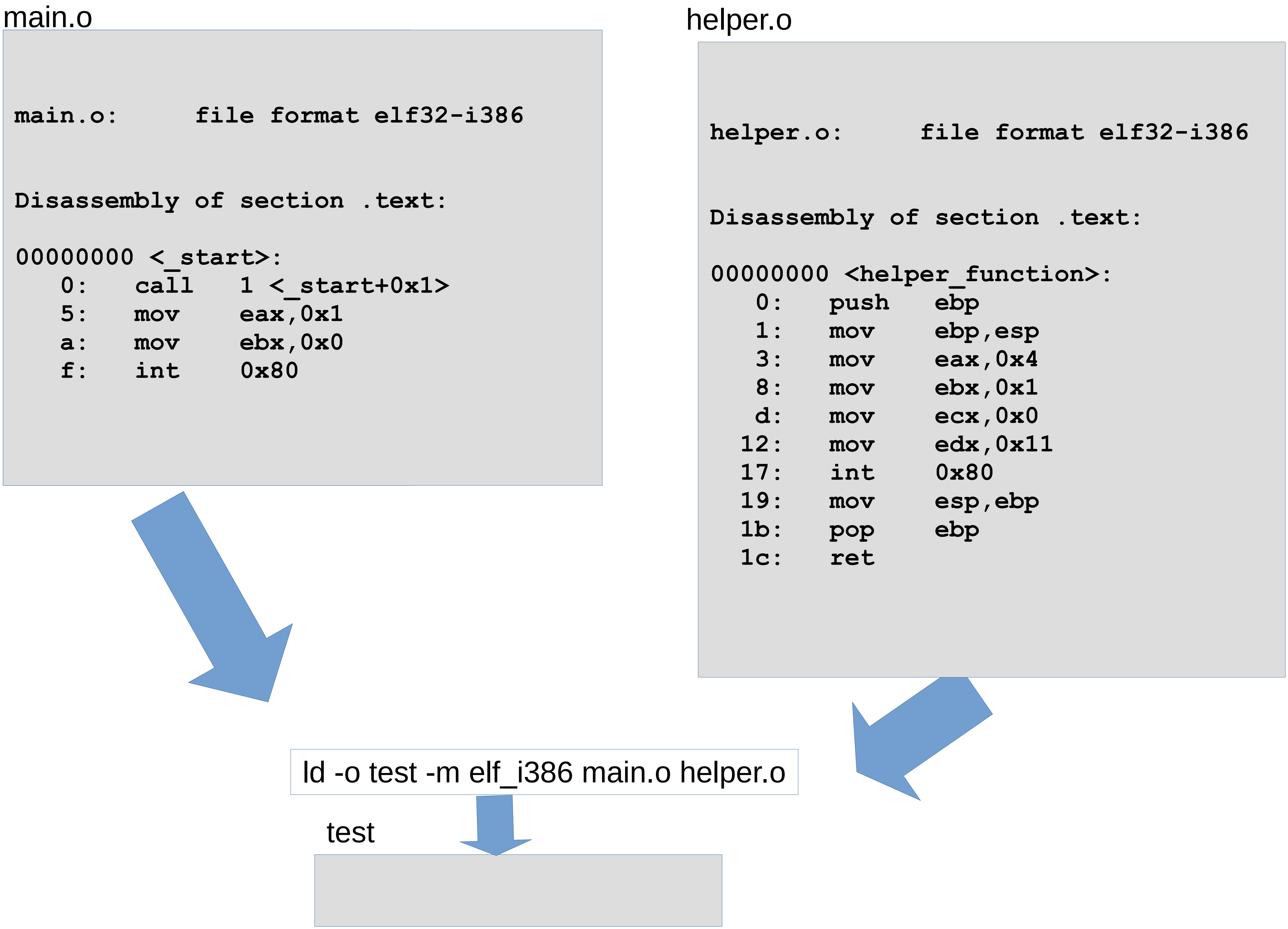
```
$ nm main.o
         U helper_function
00000000 T _start
$ nm helper.o
00000000 T helper_function
00000000 d the_string
$ 
```

nm tool will examine contents of object files.

T -- identifies an exported code symbol defined in this module
U -- identifies an symbol undefined in this module
d -- identifies a non-exported data symbol defined in this module



```
$ objdump -t main.o

main.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 main.asm
00000000 l    d  .text  00000000 .text
00000000         *UND*  00000000 helper_function
00000000 g       .text  00000000 _start



$
$
$ objdump -t helper.o

helper.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 helper.asm
00000000 l    d  .data  00000000 .data
00000000 l    d  .text  00000000 .text
00000000 l     O .data  00000001 the_string
00000000 g       .text  00000000 helper_function
```



main.o

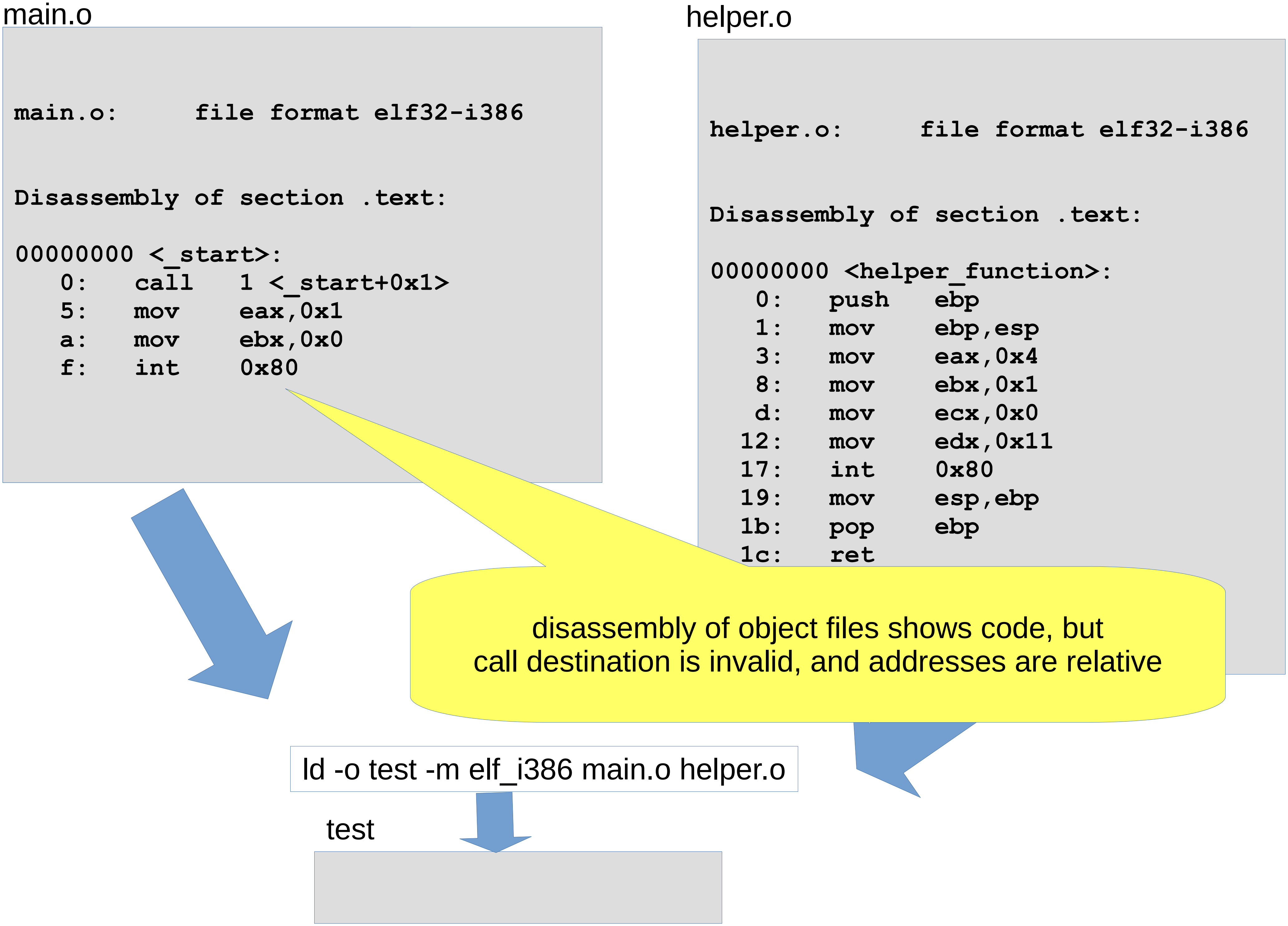
```
main.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   call   1 <_start+0x1>
   5:   mov    eax,0x1
   a:   mov    ebx,0x0
   f:   int    0x80
```

helper.o

```
helper.o:     file format elf32-i386


Disassembly of section .text:

00000000 <helper_function>:
   0:   push   ebp
   1:   mov    ebp,esp
   3:   mov    eax,0x4
   8:   mov    ebx,0x1
   d:   mov    ecx,0x0
  12:   mov    edx,0x11
  17:   int    0x80
  19:   mov    esp,ebp
  1b:   pop    ebp
  1c:   ret
```

```
ld -o test -m elf_i386 main.o helper.o
```

test



main.o

```


main.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   call   1 <_start+0x1>
   5:   mov    eax,0x1
   a:   mov    ebx,0x0
   f:   int    0x80
```

helper.o

```


helper.o:     file format elf32-i386


Disassembly of section .text:

00000000 <helper_function>:
   0:   push   ebp
   1:   mov    ebp,esp
   3:   mov    eax,0x4
   8:   mov    ebx,0x1
   d:   mov    ecx,0x0
  12:   mov    edx,0x11
  17:   int    0x80
  19:   mov    esp,ebp
  1b:   pop    ebp
  1c:   ret
```

disassembly of object files shows code, but
call destination is invalid, and addresses are relative

```
ld -o test -m elf_i386 main.o helper.o
```

test

test

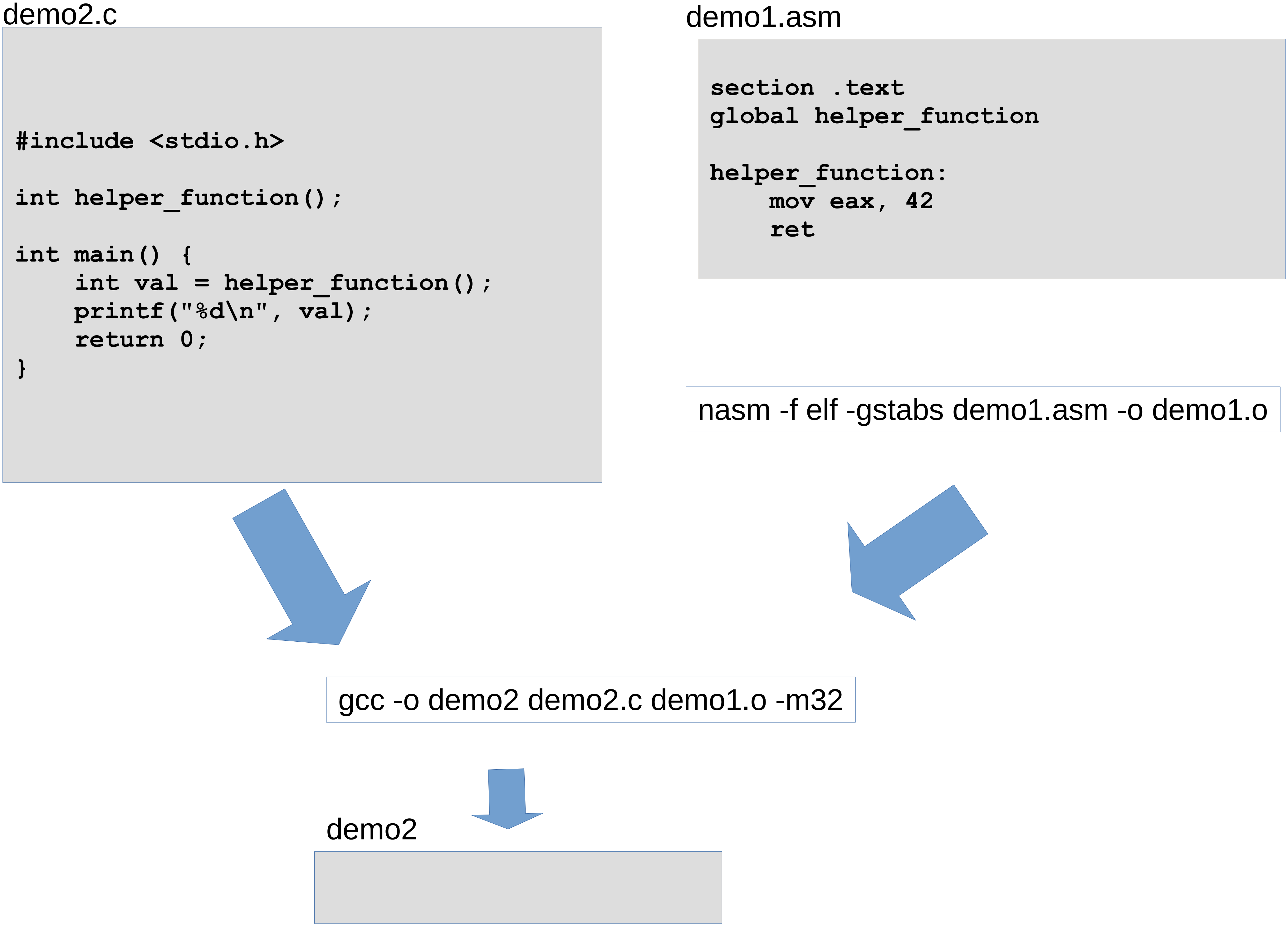
```
test:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:   e8 1b 00 00 00          call   8049020 <helper_function>
 8049005:   b8 01 00 00 00          mov    eax,0x1
 804900a:   bb 00 00 00 00          mov    ebx,0x0
 804900f:   cd 80                   int    0x80

08049020 <helper_function>:
 8049020:   55                      push   ebp
 8049021:   89 e5                   mov    ebp,esp
 8049023:   b8 04 00 00 00          mov    eax,0x4
 8049028:   bb 01 00 00 00          mov    ebx,0x1
 804902d:   b9 00 a0 04 08          mov    ecx,0x804a000
 8049032:   ba 12 00 00 00          mov    edx,0x12
 8049037:   cd 80                   int    0x80
 8049039:   89 ec                   mov    esp,ebp
 804903b:   5d                      pop    ebp
 804903c:   c3                      ret
```

test

```
test:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:   e8 1b 00 00 00          call   8049020 <helper_function>
 8049005:   b8 01 00 00 00          mov    eax,0x1
 804900a:   bb 00 00 00 00          mov    ebx,0x0
 804900f:   cd 80                   int    0x80

08049020 <helper_function>:
 8049020:   55                      push   ebp
 8049021:   89 e5                   mov    ebp,esp
 8049023:   b8 04 00 00 00          mov    eax,0x4
 8049028:   bb 01 00 00 00          mov    ebx,0x1
 804902d:   b9 00 a0 04 08          mov    ecx,0x804a000
 8049032:   ba 12 00 00 00                 edx,0x12
 8049037:   cd 80                   in     0x80
 8049039:   89 ec                   mov
 804903b:   5d                      pop    e
 804903c:   c3
```

Linked executable code.

Addresses are absolute.
Symbol references are resolved.



demo2.c

```
#include <stdio.h>

int helper_function();

int main() {
    int val = helper_function();
    printf("%d\n", val);
    return 0;
}
```

demo1.asm

```
section .text
global helper_function

helper_function:
    mov eax, 42
    ret
```

```
nasm -f elf -gstabs demo1.asm -o demo1.o
```

```
gcc -o demo2 demo2.c demo1.o -m32
```

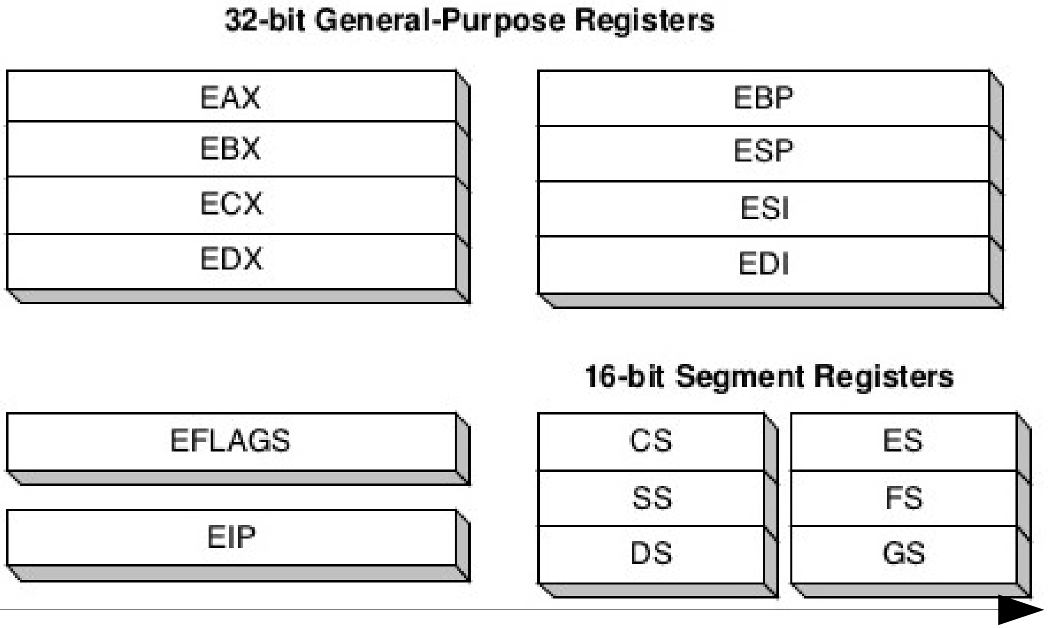
demo2

E15 vs E20 vs IA-32

	E15	E20	IA-32
Memory	Sixteen 12-bit cells of instruction ROM, no data memory	8192 16-bit cells of mixed instruction/data RAM	Up to 4GB of mixed instruction/data RAM
Registers	Four 4-bit registers	Seven general-purpose 16-bit registers	Eight 32-bit general-purpose, eight 80-bit floating point, six 32-bit segment registers, plus various vector, debug, and system registers
Instructions	11 unique instruction mnemonics	13 instructions, 3 psuedo-instructions	About 981 unique instruction mnemonics, considerably more if variations of operand type are taken into account
Real-world use	none	none	Probably in your computer right now

Our basic 32-bit registers are EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI.

We will ignore the segment registers CS, SS, DS, ES, FS, GS.



32-bit General-Purpose Registers

EAX
EBX
ECX
EDX

EBP
ESP
ESI
EDI

16-bit Segment Registers

EFLAGS

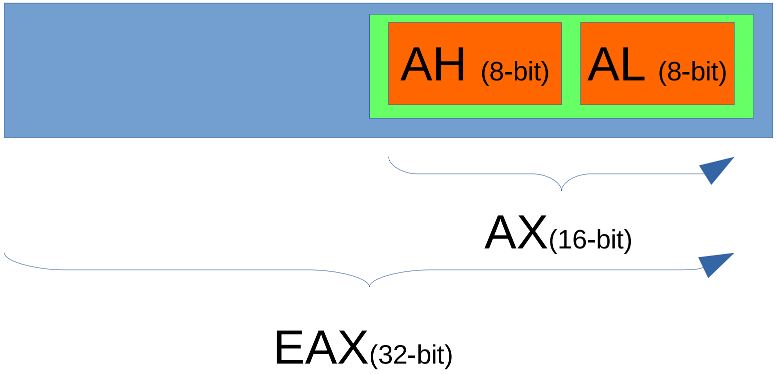
EIP

CS
SS
DS

ES
FS
GS

Portions of EAX, EBX, ECX, and EDX can be named and accessed separately.

We have 16-bit registers AX, BX, CX, DX; and 8-bit registers AH, AL, BH, BL, CH, CL, DH, DL.



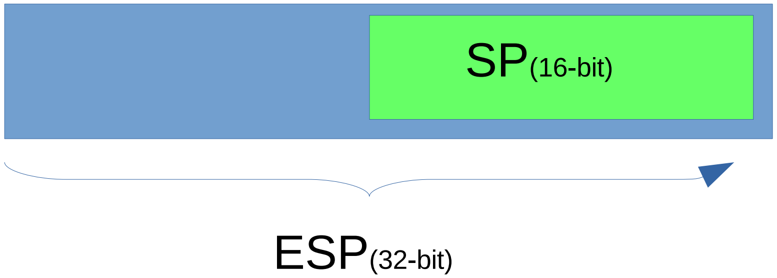
AH (8-bit) AL (8-bit)

AX(16-bit)

EAX(32-bit)

Portions of EBP, ESP, ESI, and EDI can be named and accessed separately.

We have 16-bit registers BP, SP, SI, and DI.

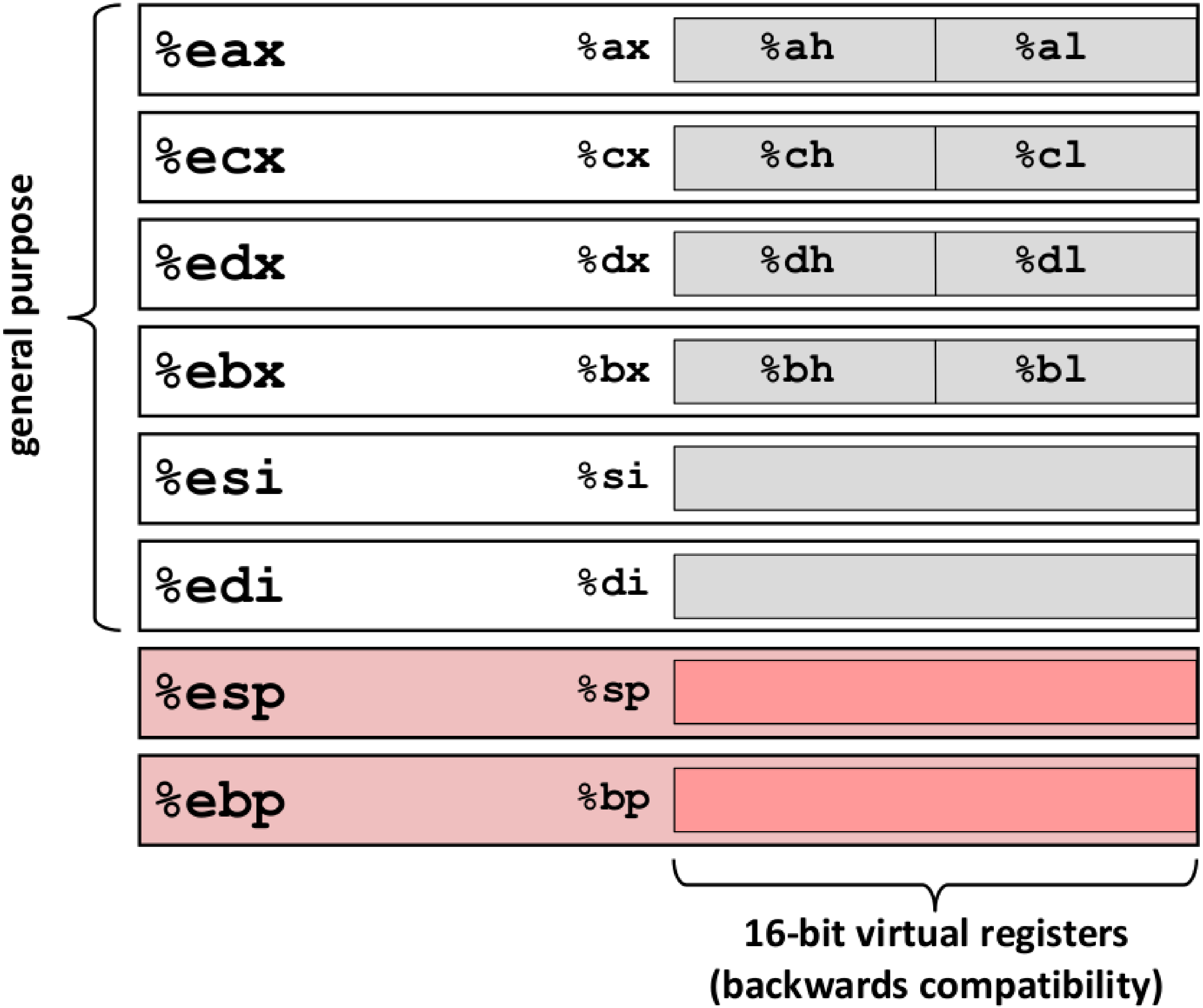


SP(16-bit)

ESP(32-bit)

EIP cannot be accessed directly, but it can be changed with the JMP instruction. EFLAGS cannot be accessed directly, but it contains important bit-size flags like the zero flag. The segment registers are set up by the OS and you generally don't want to mess with them.

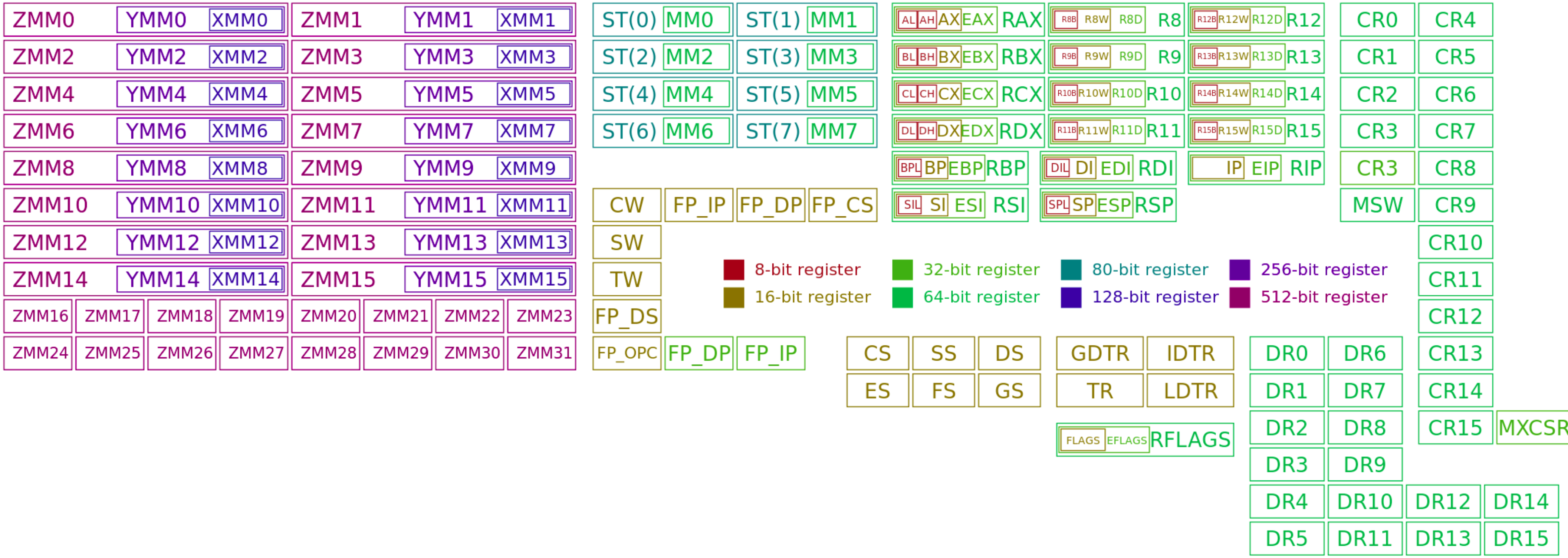
Some History: IA32 Registers



					Origin (mostly obsolete)
general purpose	%eax	%ax	%ah	%al	*accumulate*
	%ecx	%cx	%ch	%cl	*counter*
	%edx	%dx	%dh	%dl	*data*
	%ebx	%bx	%bh	%bl	*base*
	%esi	%si			*source index*
	%edi	%di			*destination index*
	%esp	%sp			***stack pointer***
	%ebp	%bp			***base pointer***

16-bit virtual registers
(backwards compatibility)

x86-64 register file



ZMM0 YMM0 XMM0 ZMM1 YMM1 XMM1 ST(0) MM0 ST(1) MM1 AL AH AX EAX RAX R8B R8W R8D R8 R12B R12W R12D R12 CR0 CR4

ZMM2 YMM2 XMM2 ZMM3 YMM3 XMM3 ST(2) MM2 ST(3) MM3 BL BH BX EBX RBX R9B R9W R9D R9 R13B R13W R13D R13 CR1 CR5

ZMM4 YMM4 XMM4 ZMM5 YMM5 XMM5 ST(4) MM4 ST(5) MM5 CL CH CX ECX RCX R10B R10W R10D R10 R14B R14W R14D R14 CR2 CR6

ZMM6 YMM6 XMM6 ZMM7 YMM7 XMM7 ST(6) MM6 ST(7) MM7 DL DH DX EDX RDX R11B R11W R11D R11 R15B R15W R15D R15 CR3 CR7

ZMM8 YMM8 XMM8 ZMM9 YMM9 XMM9 BPL BP EBP RBP DIL DI EDI RDI IP EIP RIP CR3 CR8

ZMM10 YMM10 XMM10 ZMM11 YMM11 XMM11 CW FP_IP FP_DP FP_CS SIL SI ESI RSI SPL SP ESP RSP MSW CR9

ZMM12 YMM12 XMM12 ZMM13 YMM13 XMM13 SW CR10

ZMM14 YMM14 XMM14 ZMM15 YMM15 XMM15 TW CR11

ZMM16 ZMM17 ZMM18 ZMM19 ZMM20 ZMM21 ZMM22 ZMM23 FP_DS CR12

ZMM24 ZMM25 ZMM26 ZMM27 ZMM28 ZMM29 ZMM30 ZMM31 FP_OPC FP_DP FP_IP CS SS DS GDTR IDTR DR0 DR6 CR13

ES FS GS TR LDTR DR1 DR7 CR14

FLAGS EFLAGS RFLAGS DR2 DR8 CR15 MXCSR

DR3 DR9

DR4 DR10 DR12 DR14

DR5 DR11 DR13 DR15

🟥 8-bit register 🟩 32-bit register 🟦 80-bit register 🟪 256-bit register

🟫 16-bit register 🟩 64-bit register 🟦 128-bit register 🟪 512-bit register

https://upload.wikimedia.org/wikipedia/commons/1/15/Table_of_x86_Registers_svg.svg

Syntax

- Numeric literals
  - must begin with a digit 0-9
  - decimal: `53`
  - hex: `0x53` or `53h`
  - binary: `010101b`
  - `ffh` is not a numeric literal! you must type `0ffh` or `0xff`
  - Note the difference between `mov bh,ah` and `mov bh,0ah`!
    - `mov bh,ah`: move the value of register ah into register bh
    - `mov bh,0ah`: move the value numeric literal 0ah (=10 decimal) into bh
- Comments
  - comments begin with a semicolon and continue to the end of the line

```
mov eax, 53    ; this is a comment
```

NASM numeric literals

```
mov     ax,200          ; decimal
mov     ax,0200         ; still decimal
mov     ax,0200d        ; explicitly decimal
mov     ax,0d200        ; also decimal
mov     ax,0c8h         ; hex
mov     ax,$0c8         ; hex again: the 0 is required
mov     ax,0xc8         ; hex yet again
mov     ax,0hc8         ; still hex
mov     ax,310q         ; octal
mov     ax,310o         ; octal again
mov     ax,0o310        ; octal yet again
mov     ax,0q310        ; octal yet again
mov     ax,11001000b    ; binary
mov     ax,1100_1000b   ; same binary constant
mov     ax,1100_1000y   ; same binary constant once more
mov     ax,0b1100_1000  ; same binary constant yet again
mov     ax,0y1100_1000  ; same binary constant yet again
```

Syntax

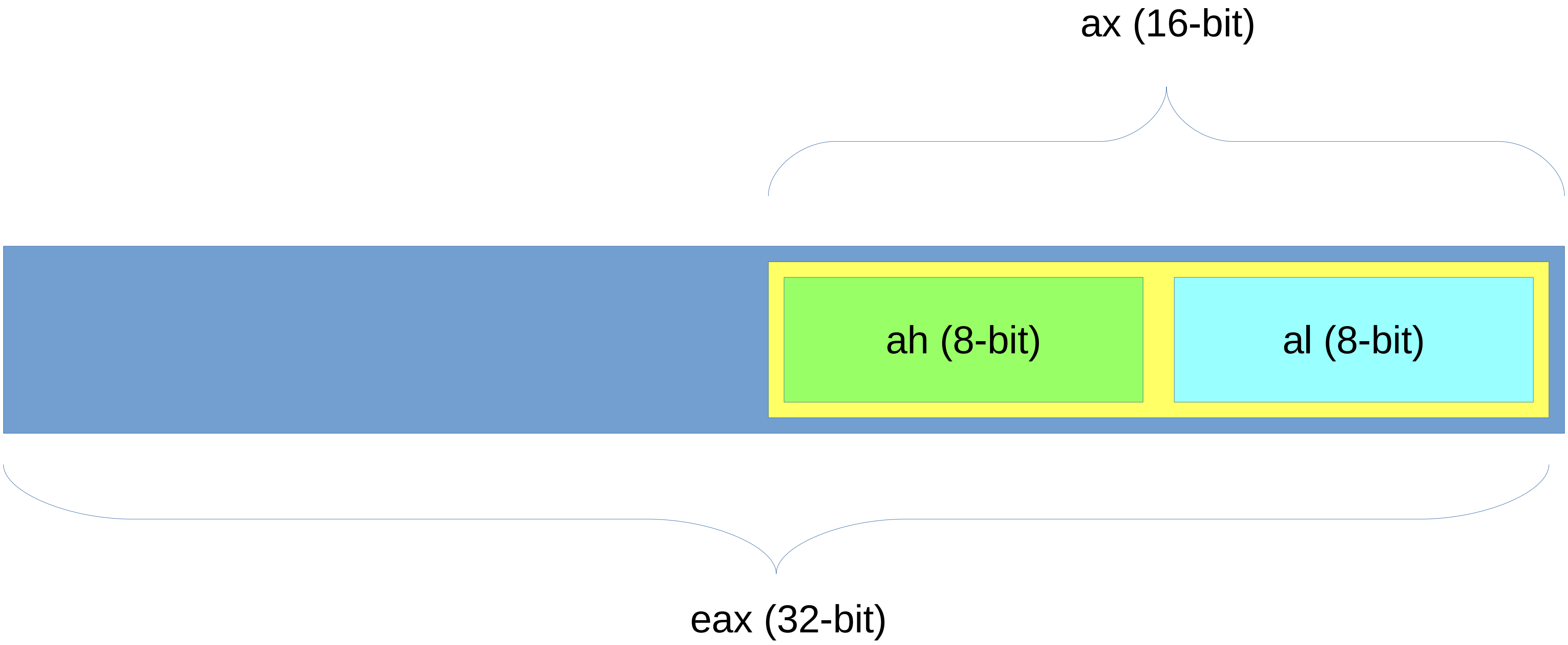
- Instructions
  - `mov eax, 53`
    - comma between operands
    - no comma after opcode
    - the destination register is usually on the *left*
    - here, we are `mov`ing immediate value 53 into register eax
  - `mov eax, ebx`
    - same opcode can apply to register-register operations
    - here, we are `mov`ing register ebx into register eax
  - `mov eax, [53]`
    - memory accesses are denoted with square brackets `[ ]`
    - here, we are moving the content of the dword at address 53 into register eax
  - `mov [53], eax`
    - here, we are moving the content of register eax into the dword at address 53
  - `mov eax, [ebx]`
    - here, we are moving the dword pointed to by register ebx into register eax
  - `mov [eax], ebx`
    - here, we are moving the value of register ebx into the dword pointed to by register eax

Syntax

- Instructions
  - `mov 53, eax`
    - invalid. Can't move into an immediate
  - `mov [eax], [ebx]`
    - invalid. Maximum of one memory access per instruction
    - We have to rewrite this in two steps:

```
mov ecx, [ebx]
mov [eax], ecx
```

Nested registers



Note that modifying a register will also modify all registers that it overlaps with.
For example, changing ah will also change ax and eax.

Register examples

```
mov eax, 0xf0f0f0f0 ; set eax (32-bit) to 0xf0f0f0f0
mov ax, 0xabab      ; set ax (16-bit) to 0xabab
                    ; eax is now 0xf0f0abab
add eax, 1          ; eax is now 0xf0f0abac
```

```
mov eax, 0xabababab  ; set eax (32-bit) to 0xabababab
mov al, 0xff         ; set al (8-bit) to 0xff
                     ; eax is now 0xabababff
add eax, 1           ; eax is now 0xababac00
```

```
mov eax, 0xabababab  ; set eax (32-bit) to 0xabababab
mov al, 0xff         ; set al (8-bit) to 0xff
                     ; eax is now 0xabababff
add al, 1            ; eax is now 0xababab00
```

Syntax

- Labels
  - identify an address by name, rather than by number
  - rules:
    - alphanumeric characters, plus underscore
    - must begin with a letter
    - may also begin with a period, for local labels
- Code labels
  - to define a label, just give its name, followed by a colon
  - to use it, just refer to it by name

```
  mov eax, 0
loop1:
  add eax, 1
  cmp eax, 50
  jne loop1
```

- Data labels
  - labels may identify an address that can be used as a pointer or immediate

```
section .data
age: dd 0         ; allocate a dword, value 0, its address is age
section .text
mov eax, [age]    ; read the value at that address
mov ebx, age      ; store the address in the register
```

Sizes

	size	register examples	directive
byte	8 bits	ah, al, bh, bl, ...	db
word	16 bits (two bytes)	ax, bx, cx, dx, ...	dw
dword	32 bits (four bytes)	eax, ebx, ecx, edx, ...	dd
qword	64 bits (eight bytes)	rax, rbx, rcx, rdx, ...	dq

Syntax

- Sections
  - There are three sections we care about
    - text: you put your code here
    - data: your put your *initialized* global data here
      - use db, dw, dd, dq to store data
    - bss: you put your *uninitialized* global data here
      - use resb, resw, resd, resq to allocate space

```
section .data
my_favorite_string: db "Here is an initialized value."
some_number: dd 42
    ; db means "direct bytes", i.e. directly input these bytes into memory
    ; the label's value is the address of the beginning of this memory region

section .bss
uninitialized_32byte_buffer: resb 32
    ; resb means "reserve bytes" i.e. reserve 32 bytes of memory,
    ; without initial value
    ; the label's value is the address of the beginning of this memory region

section .text
global _start
_start:                 ; the special _start label identifies the
                        ; first instruction to run
; your code goes here
mov [uninitialized_32byte_buffer], ah
```

Syntax

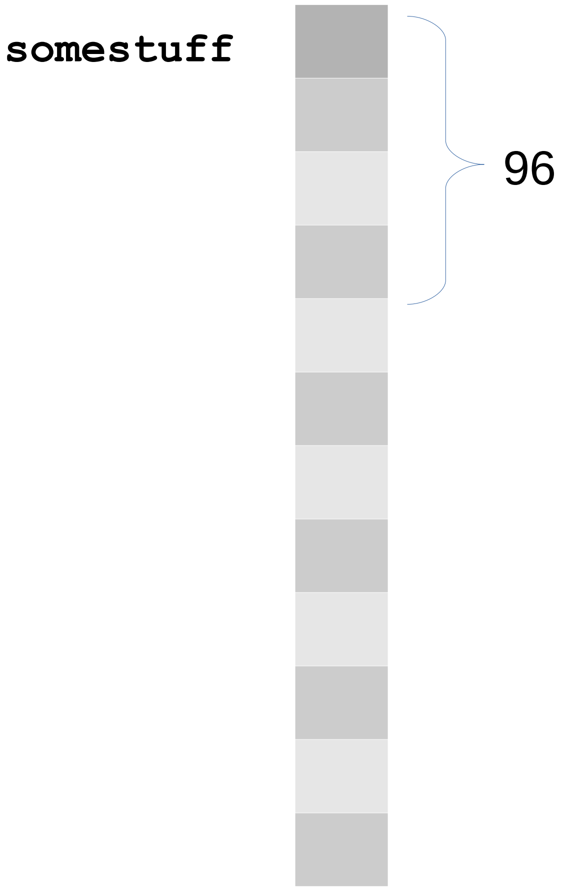
- Typecasts
  - Usually, the size of an operand is implicit:
    - `mov eax, 53h`
      - I am storing a dword in eax, because eax is a 32-bit register
    - `mov [my_data], ax`
      - I am writing a word into memory, because ax is a 16-bit register
  - Sometimes, the size of the operand cannot be inferred automatically
    - `mov [my_data], 53h`
    - How much data am I moving? Am I moving a byte? word? dword? qword?
    - The assembly doesn't and will refuse this instruction: "operation size not specified"
  - Specify the size explicitly
    - `mov dword [my_data], 53h  ; store four bytes in memory: 00000053`
    - `mov word [my_data], 53h  ; store two bytes in memory: 00053`
    - `mov byte [my_data], 53h   ; store one byte in memory: 53`

Instructions vs Directives

- Instructions (e.g. **mov**, **jmp**, **add**, etc) are translated by the assembler into machine language, which can be run by the processor. Every instruction has a corresponding binary machine language representation, just like in E15 and E20.
- Directives are commands to the assembler that don't necessarily correspond to bytes in machine language. For example: **include** (include an external file), **dd** (insert raw dwords into the output stream), **section** (tells the assembler which part of the output stream to write to), etc

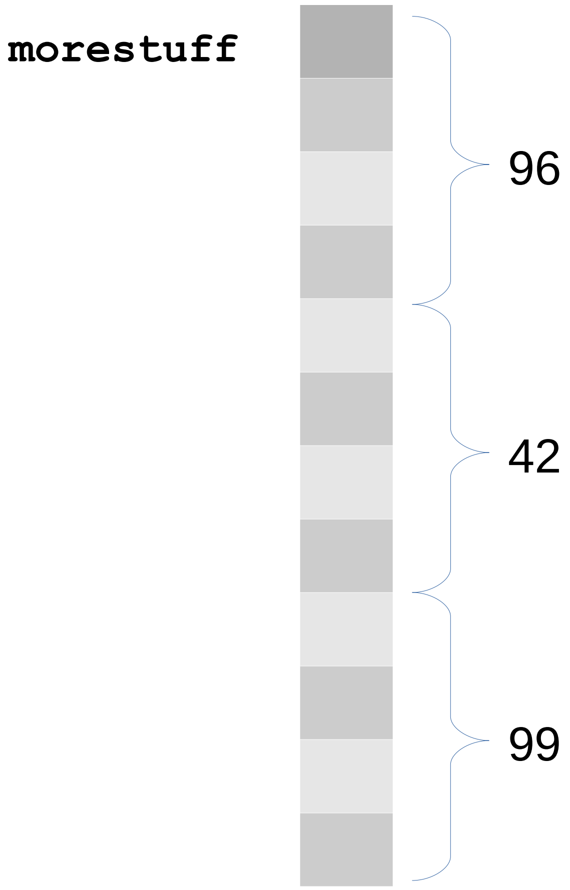
Directives example

```
somestuff: dd 96
    ; at address somestuff
    ; allocate one dword (32-bit),
    ; initialized to value 96
```



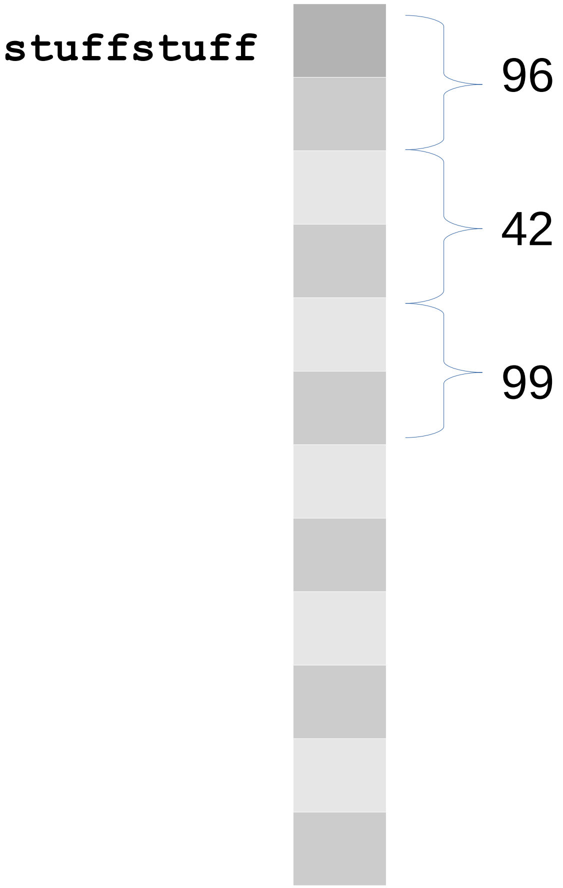
Directives example

```
morestuff: dd 96, 42, 99
    ; at address morestuff, allocate dword 96
    ; at address morestuff+4, allocate dword 42
    ; at address morestuff+8, allocate dword 99
```



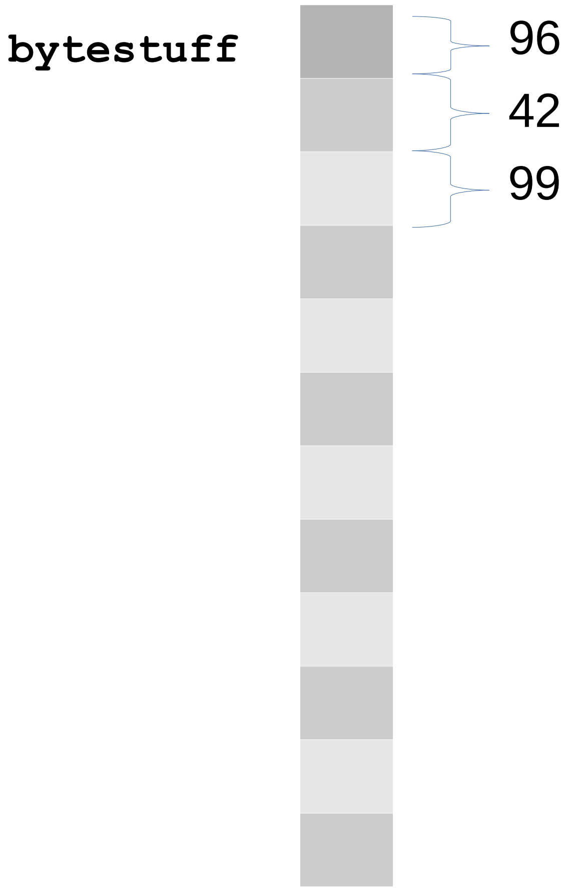
Directives example

```
stuffstuff: dw 96, 42, 99
    ; at address stuffstuff, allocate word (16-bit) 96
    ; at address stuffstuff+2, allocate word 42
    ; at address stuffstuff+4, allocate word 99
```



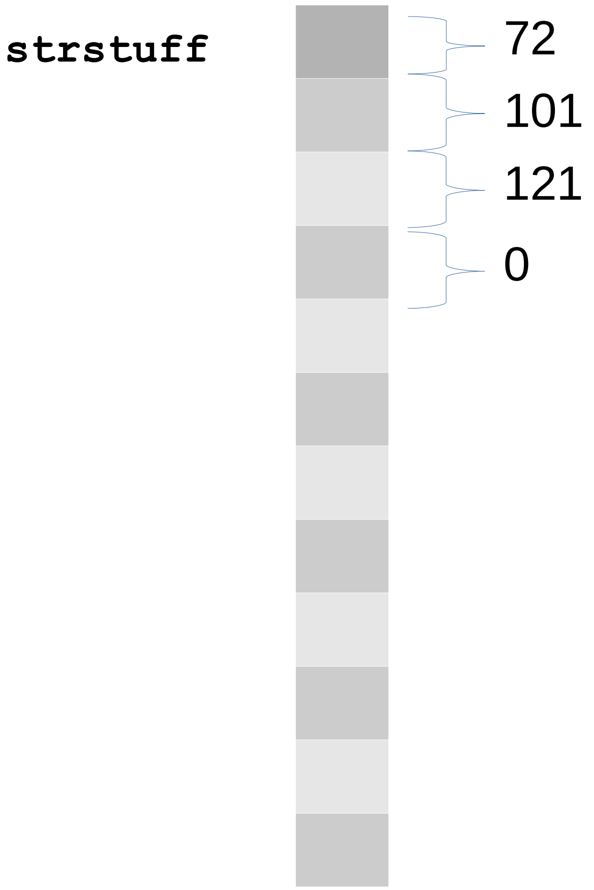
Directives example

```
bytestuff: db 96, 42, 99
    ; at address bytestuff, allocate byte (8-bit) 96
    ; at address bytestuff+1, allocate byte 42
    ; at address bytestuff+2, allocate byte 99
```



Directives example

```
strstuff: db "Hey",0
    ; at address strstuff, allocate byte (8-bit) "H" (72)
    ; at address strstuff+1, allocate byte "e" (101)
    ; at address strstuff+2, allocate byte "y" (121)
    ; at address strstuff+3, allocate byte 0
```



Directives

section	Subsequent assembly is applied to one of the executable's sections (usually text, data, or bss)
db, dw, dd, dq	Insert initialized literal data (of bytes, word, dword, qword)
resb, resw, resd, resq	Reserve space for uninitialized data
global	Mark a symbol as global, i.e. visible outside of this linker unit
equ	Define compile-time symbol to a particular value
;	comment

Directives, unlike opcodes, are not translated directly into machine language; instead, they provide commands to the assembler.

Unlike in E20, x86 assembly supports different kinds of operands on the same opcode. So we can (often) use immediate values, registers, and memory addresses. The `[]` notation always signifies a memory access.

```
mov eax, 5
mov eax, some_label
```

Store the immediate value 5 into the 32-bit register eax.
The destination register is on the *left*, in most cases.

```
mov eax, ebx
```

Store the value of register ebx into register eax.

```
mov eax, [5]
mov eax, [some_label]
```

Get the 32-bit value starting at memory cell address 5, and store it into register eax.

```
mov eax, [ebx]
```

Get the 32-bit value starting at the memory cell address stored in register ebx, and store it into eax. In other words, *dereference the pointer* in ebx.

```
mov eax, [ebx]
```

Let's say ebx stores the value 0x8040. We interpret that as a memory address.

eax is a 32-bit register, so we need to read 32 bits from memory.

Each memory cell is one byte, so we need to read 4 cells.

Starting at the address in ebx, that means addresses 8040, 8041, 8042, and 8043. Each address has the byte value shown in the table.

Because x86 is a *little-endian* architecture, we combine the values in those cells in least-significant-first order, i.e. 0xd84ba045.

And *that* is the value that we store in eax.

addr	val
803f	23
8040	45
8041	a0
8042	4b
8043	d8
8044	3b

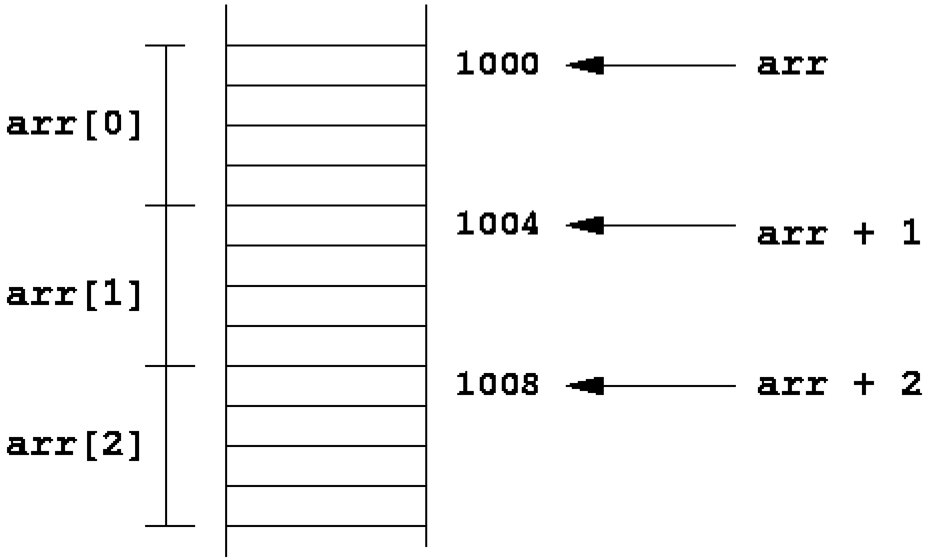
Memory access sizes

```
mov eax, [ebx] ; read a DWORD
mov ax,  [ebx] ; read a WORD
mov ah,  [ebx] ; read a BYTE
```

```
mov [ebx], eax ; write a DWORD
mov [ebx], ax ; write a WORD
mov [ebx], ah ; write a BYTE
```

Note that the size of the pointer must always be 32-bit, regardless of the size of the value written to memory.

Accessing arrays



The array begins at address `arr` (in this case, 1000). To calculate the address of the *n*th element of the array, assuming that all elements have size *s* (in this case 4), we use the formula:

$\text{addr}_n = \texttt{arr} + n * s$

We can use the memory scaling operands to do this all in one step:

```
mov eax, [ebx * 4 + arr]
```

where `arr` stores the base address of the array, a constant; `ebx` stores the element number; and the value at that location is put into `eax`.

Multibyte values

- **`mov [foobar], eax`**
  - Moves the 32-bit value from register eax to the memory location identified by label `foobar`
  - Equivalent to E20:
    - **`sw $eax, foobar($0)`**
  - However, on Intel, each memory cell is 8-bits. How can we write a 32-bit value to memory?
    - Answer: we write to four consecutive memory cells, starting at `foobar`
- Any multibyte value can be expressed as a sequence of bytes
  - 7432 is too big for one byte
  - In binary, it's 00011101 00001000
  - In hex, it's 0x1d08 = 0x1d; 0x08

Multibyte values

- Let's say eax = 2882400018 = 0xabcdef12
  - mov [foobar], eax

Address	foobar+0	foobar+1	foobar+2	foobar+3
Value	0x12	0xef	0xcd	0xab

- Let's say eax = 500 = 0x1f4
  - mov [foobaz], eax

Address	foobaz+0	foobaz+1	foobaz+2	foobaz+3
Value	0xf4	0x01	0x00	0x00

  - mov ebx, [foobaz]
    - Will read 4 bytes in least-significant-to-most-significant order
    - Thereafter, ebx == 0x1f4 == eax

Multibyte values

- Let's say ax = 43981 = 0xabcd
  - mov [foobar], ax

Address	foobar+0	foobar+1
Value	0xcd	0xab

- Let's say ax = 500 = 0x1f4
  - mov [foobaz], ax

Address	foobaz+0	foobaz+1
Value	0xf4	0x01

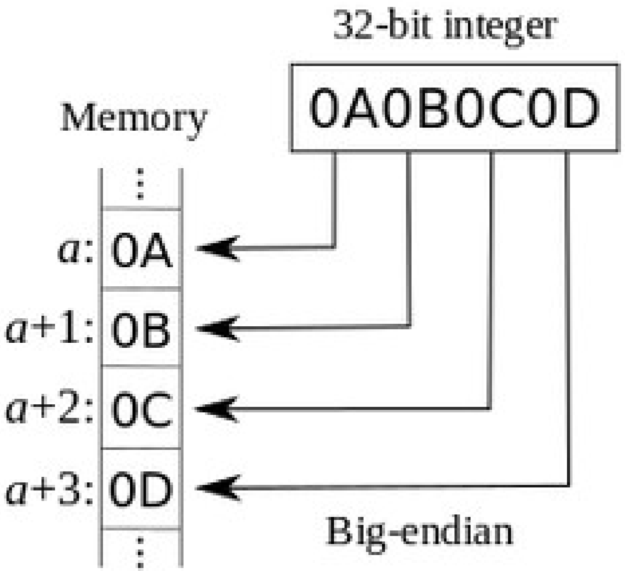
Multibyte values

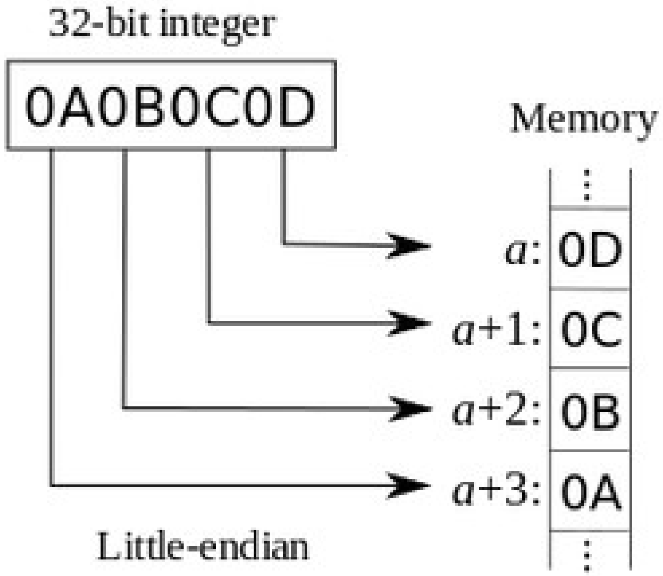
- Let's say ah = 4 = 0x4
  - mov [foobar], ah

Address	foobar+0
Value	0x04

- Let's say al = 255 = 0xff
  - mov [foobaz], al

Address	foobaz+0
Value	0xff


32-bit integer
0A0B0C0D
Memory
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian


32-bit integer
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian

Endianism

- Consider:

```
mov eax, 0x12345678
mov [foobar], eax
mov bx, [foobar]
mov ch, [foobar]
```

- What is the final value of bx and ch?

Endianism

- Consider:

```
mov eax, 0x12345678
mov [foobar], eax
mov bx, [foobar]
mov ch, [foobar]
```

Address	**Value**
foobar+0	78
foobar+1	56
foobar+2	34
foobar+3	12

- What is the final value of bx and ch?
  - bx = 0x5678
  - ch = 0x78

Endianism

- Consider:

```
mov eax, 0xabcdef

mov [foobar], eax
mov [foobar+2], eax
mov [foobar+4], ah
```

Address	**Value**
foobar+0	
foobar+1	
foobar+2	
foobar+3	
foobar+4	
foobar+5	

- What is the final value the array starting at foobar?

Endianism

- Consider:

```
mov eax, 0xabcdef

mov [foobar], eax
mov [foobar+2], eax
mov [foobar+4], ah
```

Address	Value
foobar+0	ef
foobar+1	cd
foobar+2	ef
foobar+3	cd
foobar+4	cd
foobar+5	00

- What is the final value the array starting at foobar?
  - ef cd ef cd cd 00

Strings in x86

Different programming languages use different internal representations of strings. All of these forms must be expressible in assembly language. Therefore, there is no single way to express strings in assembly language.
All strings are finite sequences of characters, so we need a way to express the characters as well as the length.

```
section .data
mystring:
    db "Hello", 0
```

Char	"H"	"e"	"l"	"l"	"o"	NUL
Numeric	72	101	108	108	111	0

C-style strings are used when you define a string with **`char*`** in C or C++. Each character occupies one byte. Their length is indicated by a zero-valued sentinel byte at the end of the string.
Pros:

- Easy to write and use.
- Allowed unlimited length.

Cons:

- Calculating the length of the string requires walking the whole strings, in O(n) time.
- The body of the string cannot contain a zero byte, making it unsuitable for some kinds of data.

Strings in x86

Different programming languages use different internal representations of strings. All of these forms must be expressible in assembly language. Therefore, there is no single way to express strings in assembly language.
All strings are finite sequences of characters, so we need a way to express the characters as well as the length.

```
section .data
mystring:
   db 5, "Hello"
```

Char	ENQ	"H"	"e"	"l"	"l"	"o"
Numeric	5	72	101	108	108	111

Pascal-style strings store their length separately, typically before the character values.
Pros:

- Finding the length is done in O(1) time.
- Body of string can contain any character.

Cons:

- Maximum string size is limited. In this case, the length is one byte, so a string cannot be longer than 255 characters.

Strings in x86

Different programming languages use different internal representations of strings. All of these forms must be expressible in assembly language. Therefore, there is no single way to express strings in assembly language.
All strings are finite sequences of characters, so we need a way to express the characters as well as the length.

```
section .data
mystring:
    db "H", 0, "i", 0, 61,
       216, 68, 222, 0, 0
```

Char	"H"	NUL	"i"	NUL					NUL	NUL
Numeric	72	0	105	0	61	216	68	222	0	0

Multibyte string encodings may use more than one byte for each character. Such encodings include UTF-16, UTF-32, UCS-2, UCS-4. In the example above, we show a UTF-16-encoded string with C-style sentinel. Here, each character occupies at least two bytes, and some, rarer characters ocupy more. Note that character encoding is orthogonal to the question of length marking.

Pros:

- Can represent more than just ASCII characters, such as foreign alphabets, emojis, etc.

Cons:

- Length calculation, indexing, and other operations are necessarily O(n) and harder to implement.

Operand types

Immediate operand	mov eax, 5
Register operand	mov eax, ebx
Memory operand (label)	some_variable: dd 5 mov eax, [some_variable] mov [some_variable], eax
Memory operand (register)	mov eax, [ebx] mov [ebx], eax
Memory operand (offset)	some_string: resb 20 mov ah, [some_string+ebx]
Memory operand (scaling)	some_array: resd 20 mov eax, [some_array+ebx*4]
Double memory operand (ILLEGAL)	mov [some_variable], [another_variable]
Memory and imm operands, with typecast	mov byte [ecx], 3 mov byte ptr [ecx], 3

C	IA-32
a = a+b;	add eax, ebx
a++;	inc eax
a += 2;	add eax, 2
a = (a-b)*(c-d);	sub eax, ebx sub ecx, edx mul eax, ecx
a = 53;	mov eax, 53

if (a==53)
   a -= 1;
else
   a += 1;
	cmp eax, 53
	jne L1
	dec eax
	jmp L2
	L1:
	inc eax
	L2:

if (a==53)
   a -= 1;
a += 1;
	cmp eax, 53
	jne L1
	dec eax
	L1:
	inc eax

C	IA-32

while (a<100)
  a*=2;
	L1:
	cmp eax, 100
	jge L2
	imul eax, 2
	jmp L1
	L2:

int arr[34];
int a = 0;
for (int c=0; c<34; c++)
   a += arr[c];
	xor eax, eax
	xor ecx, ecx
	L1:
	cmp ecx, 34
	jge L2
	add eax, [arr + ecx * 4]
	inc ecx
	jmp L1
	L2:

int arr[34];
int a = 0;
int *c = &arr[0];
while (c < arr+34)
   a+=*(c++);
	mov ecx, arr
	xor eax, eax
	L1:
	cmp ecx, arr + 34 * 4
	jge L2
	add eax, [ecx]
	add ecx, 4
	jmp L1
	L2:

C	IA-32 and Linux syscall
printf("Hello\n");	section .data mystr: db "Hello",10 section .text mov eax, 4 mov ebx, 1 mov ecx, mystr mov edx,6 int 80h

char my_str[32];
fgets(my_str, 32, stdin);
	section .bss
	my_str: resb 32
	section .text
	mov eax, 3
	mov ebx, 0
	mov ecx, my_str
	mov edx, 32
	int 80h

exit(0);	mov eax, 1 mov ebx, 0 int 80h

A digression on interrupts

- Interrupts are signals to the OS.
- The "interrupt vector table" is a table of pointers to OS code for handling various conditions.
- Many interrupts are generated by hardware or CPU events, but some are generated by software by the INT opcode.

```
IVT Offset | INT #     | Description
-----------+-----------+-----------------------------------
0x0000     | 0x00      | Divide by 0
0x0004     | 0x01      | Reserved
0x0008     | 0x02      | NMI Interrupt
0x000C     | 0x03      | Breakpoint (INT3)
0x0010     | 0x04      | Overflow (INTO)
0x0014     | 0x05      | Bounds range exceeded (BOUND)
0x0018     | 0x06      | Invalid opcode (UD2)
0x001C     | 0x07      | Device not available (WAIT/FWAIT)
0x0020     | 0x08      | Double fault
0x0024     | 0x09      | Coprocessor segment overrun
0x0028     | 0x0A      | Invalid TSS
0x002C     | 0x0B      | Segment not present
0x0030     | 0x0C      | Stack-segment fault
0x0034     | 0x0D      | General protection fault
0x0038     | 0x0E      | Page fault
0x003C     | 0x0F      | Reserved
0x0040     | 0x10      | x87 FPU error
0x0044     | 0x11      | Alignment check
0x0048     | 0x12      | Machine check
0x004C     | 0x13      | SIMD Floating-Point Exception
0x00xx     | 0x14-0x1F | Reserved
0x0xxx     | 0x20-0xFF | User definable
```

System calls

- Interface to operating system from application
- Vary between operating systems
- Provides services such as:
  - File access (open, read, write, close, seek, etc)
  - Network access (socket, accept, listen, bind, etc)
  - Directory access (mkdir, rmdir, rename, etc)
  - Process management (exec, kill, getpid, etc)
  - Memory allocation (brk, mmap, etc)
  - Hardware access (ioctl)
  - Privileged operations for OS tools (chroot, setuid, reboot, etc)
  - And many more

I/O

Consider:

```
cout << "Hello world";
```

What is cout? What operation is being performed here?

I/O

Consider:

```
cout << "Hello world";
```

What is cout? What operation is being performed here?

Answer:

cout is a C++ *stream*.

We are inserting (sending) the given data to the stream, causing it to appear on the screen. We use the overloaded **<<** operator.

cout is a stream corresponding to *standard output*, the usual destination for output from a program. In order to implement this behavior, the insertion operator must invoke a system call. In assembly language, we will invoke the system call directly.

• I/O = Input/Output

- Sending data to, or receiving data from, some external component. Examples:
  - reading/writing a file
  - sending/receiving network data
  - putting data on the screen
  - reading user input from keyboard
- You're already familiar with these tasks in C++ via *streams*:
  - putting data on the screen, reading input from keyboard:
    - **`cout << "Please type a number";`**
    - **`int num; cin >> num;`**
  - writing a file:
    - **`fostream f("myfile.txt");`**
      `f << "Some data";`
  - Here, **`cout`**, **`cin`**, and **`f`** are *streams* that correspond to data sources and/or sinks.
- When doing I/O through system calls, we must similarly specify the source or sink.
  - For the screen and keyboard, these will be **`stdout`** and **`stdin`**, respectively.

Linux system calls

- On IA-32, invoked via software interrupt
  - int 80h
- On IA-64, invoked via dedicated opcode
  - syscall
- In either case, signals OS to interrupt current application
  - OS halts application, examines parameters, performs service, and resumes application
- Application must set up parameters to call in registers

Linux system calls: a partial list

https://web.archive.org/web/20200727064105/http://shell-storm.org/shellcode/files/syscalls.html

eax	Name	Source	ebx	ecx	edx
1	sys_exit	kernel/exit.c	int	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t
5	sys_open	fs/open.c	const char *	int	int
6	sys_close	fs/open.c	unsigned int	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int
8	sys_creat	fs/open.c	const char *	int	-
9	sys_link	fs/namei.c	const char *	const char *	-
10	sys_unlink	fs/namei.c	const char *	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-
12	sys_chdir	fs/open.c	const char *	-	-
13	sys_time	kernel/time.c	int *	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t
15	sys_chmod	fs/open.c	const char *	mode_t	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t
18	sys_stat	fs/stat.c	char *	struct	-

Linux system calls: an example

eax	Name	Source	ebx	ecx	edx
1	sys_exit	kernel/exit.c	int	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t

- sys_write is used to write bytes to a file handle
  - file handles can refer to actual files, network sockets, or terminals
- The table tells us how to call it:
  - eax always selects which system call
  - Other parameters are in ebx, ecx, edx, as necessary
  - The table uses C syntax to express the parameters
  - Once you set up the parameters, invoke the syscall via int 80h

Linux system calls: an example

eax	Name	Source	ebx	ecx	edx
1	sys_exit	kernel/exit.c	int	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t

The number of the system call goes in eax

The file handle goes in ebx

The address of the data goes in ecx

The number of bytes to write goes in edx

You get a file handle:

- by opening a file
- by creating a network socket
- in addition:
  - 0 = stdin
  - 1 = stdout
  - 2 = stderr

Linux system calls: an example

eax	Name	Source	ebx	ecx	edx
1	sys_exit	kernel/exit.c	int	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t

- sys_exit is used to terminate the application
  - this is better than infinite looping!
  - it allows the OS to reclaim the application resources in an orderly manner

Linux system calls: an example

eax	Name	Source	ebx	ecx	edx
1	sys_exit	kernel/exit.c	int	-	-
2	sys_fork	arch/i386/kernel/ process.c	struct pt_regs	-	-
	sys_read	fs/read_write.c	unsigned in	har *	size_t
4	sys_write	fs/read_write.c	unsigned i		

- s_exit is used to termi

better than infinite

s the OS to reclai s

rderly manner

The number of the system call goes in eax

The "exit status" goes in ebx. The exit status can be used to communicate success or failure to another application. Traditionally, zero means success. Maximum value is 255.

This is equivalent to the return value of **`main`** in C/C++.

```
int main() {
    ....
    return 0; // status ok
}
```

Linux system calls: an example

eax	Name	Source	ebx	ecx	edx
1	sys_exit	kernel/exit.c	int	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t

- sys_read reads data from a file descriptor

Linux system calls: an example

eax	Name	Source	ebx	ecx	edx
1	sys_exit	kernel/exit.c	int	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t

- sys_read reads data from a file descriptor

The number of the system call goes in eax

The file descriptor to read from. Use 0 for stdin (usually keyboard).

Pointer to buffer where to store the data.

Size of buffer, i.e. maximum number of bytes to read.

Linux system calls: putting it together

```
section .data

hello_text:
    db "Hello, world", 10      ; string includes newline char

section .text
global _start

_start:

; Print the given string
    mov eax, 4                ; select write syscall
    mov ebx, 1                ; file handle: stdout
    mov ecx, hello_text       ; address of string
    mov edx, 13               ; length of string
    int 80h                   ; do it

; We're done, tell the OS to kill us
    mov eax, 1                ; select exit syscall
    mov ebx, 0                ; exit status
    int 80h                   ; do it

; We never get here
```

The most useful IA-32 instructions: arithmetic

Instruction	Example	Meaning
add	add EAX, EBX	EAX = EAX + EBX
subtract	sub EAX, EBX	EAX = EAX - EBX
add immediate	add EAX, 200	EAX = EAX + 200
signed multiply	imul EBX	EDX:EAX = EAX * EBX
	imul ECX, EBX	ECX = ECX * EBX
	imul ECX, 200	ECX = ECX * 200
unsigned multiply	mul ECX	EDX:EAX= EAX * ECX

signed divide	idiv ECX	EAX = EDX:EAX / ECX;
		EDX = EDX:EAX % ECX

unsigned divide	div ECX	EAX = EDX:EAX / ECX;
		EDX = EDX:EAX % ECX

The most useful IA-32 instructions: logic

Instruction	Example	Meaning
bitwise and	and EAX, EBX	EAX = EAX & EBX
bitwise or	or EAX, EBX	EAX = EAX | EBX
shift left logical	shl EAX, CL	EAX = EAX << CL
shift right logical (unsigned)	shr EAX, CL	EAX = EAX >>> CL
bitwise xor	xor EAX, EBX	EAX = EAX ^ EBX
arithmetic (signed) right shift	sar EAX, EBX	EAX = EAX >> EBX

The most useful IA-32 instructions: data transfer

Instruction	Example	Meaning
move	mov EAX, EBX	EAX = EBX
	mov EAX, 200	EAX = 200
	mov EAX, [EBX]	EAX = memory[EBX]
	mov EAX, [label + ESI * 4]	EAX = memory[label + ESI * 4]
	mov EAX, [EBX+ESI*4+2]	EAX = memory[EBX+ESI*4+2]
	mov [EBX], EAX	memory[EBX] = EAX

push	push EAX	ESP = ESP - 4;
		memory[ESP] = EAX

pop	pop EAX	EAX = memory[ESP];
		ESP = ESP + 4

The most useful IA-32 instructions: conditionals

Instruction	Example	Meaning
Compare	cmp eax, ebx	Set control flags: ZF, CF, OF, SF
Unconditional jump	jmp label	EIP = label
Conditional jump	je label jz label	jump if ZF
	jne label jnz label	jump if !ZF
	jecxz label jcxz label	jump if ECX == 0 jump if CX == 0
	jc label	jump if CF
	jnc label	jump if !CF
Signed comparison	jl label jg label jle label jge label	jump if SF != OF jump if !ZF && SF == OF jump if ZF || SF != OF jump SF == OF
Unsigned comparison	jb label ja label jbe label jae label	jump if CF jump if !CF && !ZF jump if CF || ZF jump if !CF

OF = overflow flag; set when ALU operation on numbers of the same sign give a different sign in the result
CF = carry flag; set when ALU operation on numbers results in a carry out, or borrow on subtract
ZF = zero flag; set when ALU operation on numbers results in zero
SF = sign flag; set when ALU operation on numbers results in negative number (i.e. high bit is set)

The most useful IA-32 instructions: other stuff

Instruction	Example	Meaning
function call	call label	push EIP and jump to label
function return	ret	pop address and jump to it
function return and stack clean-up	ret 8	pop address and jump to it, then add 8 to esp
increment	inc EAX	EAX = EAX + 1
decrement	dec EAX	EAX = EAX - 1
no operation	nop	do nothing
exchange	xchg EAX, EBX	swap the values in the two registers
system call	int 80h	invoke a system-defined interrupt; useful mainly for system calls

mov — Move (Opcodes: 88, 89, 8A, 8B, 8C, 8E, ...)

The mov instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

Syntax

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

Examples

mov eax, ebx — copy the value in ebx into eax

mov byte ptr [var], 5 — store the value 5 into the byte at location var

push — Push stack (Opcodes: FF, 89, 8A, 8B, 8C, 8E, ...)

The push instruction places its operand onto the top of the hardware supported stack in memory. Specifically, push first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses.

Syntax

push <reg32>
push <mem>
push <con32>

Examples

push eax — push eax on the stack
push [var] — push the 4 bytes at address *var* onto the stack

pop — Pop stack

The pop instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location). It first moves the 4 bytes located at memory location [SP] into the specified register or memory location, and then increments SP by 4.

Syntax

pop <reg32>
pop <mem>

Examples

pop edi — pop the top element of the stack into EDI.
pop [ebx] — pop the top element of the stack into memory at the four bytes starting at location EBX.

lea — Load effective address

The lea instruction places the *address* specified by its second operand into the register specified by its first operand. Note, the *contents* of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region.

Syntax

lea <reg32>,<mem>

Examples

lea edi, [ebx+4*esi] — the quantity EBX+4*ESI is placed in EDI.
lea eax, [var] — the value in *var* is placed in EAX.
lea eax, [val] — the value *val* is placed in EAX.

add — Integer Addition

The add instruction adds together its two operands, storing the result in its first operand. Note, whereas both operands may be registers, at most one operand may be a memory location.

Syntax

add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>

Examples

add eax, 10 — EAX ← EAX + 10
add BYTE PTR [var], 10 — add 10 to the single byte stored at memory address var

sub — Integer Subtraction

The sub instruction stores in the value of its first operand the result of subtracting the value of its second operand from the value of its first operand. As with add

Syntax

sub <reg>,<reg>

sub <reg>,<mem>

sub <mem>,<reg>

sub <reg>,<con>

sub <mem>,<con>

Examples

sub al, ah — AL ← AL - AH

sub eax, 216 — subtract 216 from the value stored in EAX

inc, dec — Increment, Decrement

The inc instruction increments the contents of its operand by one. The dec instruction decrements the contents of its operand by one.

Syntax

inc <reg>

inc <mem>

dec <reg>

dec <mem>

Examples

dec eax — subtract one from the contents of EAX.

inc DWORD PTR [var] — add one to the 32-bit integer stored at location *var*

imul — Integer Multiplication

The imul instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above).

The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e. first) operand must be a register.

The three operand form multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.

Syntax

```
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

Examples

imul eax, [var] — multiply the contents of EAX by the 32-bit contents of the memory location *var*. Store the result in EAX.

imul esi, edi, 25 — ESI → EDI * 25

idiv — Integer Division

The idiv instruction divides the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant four bytes and EAX as the least significant four bytes) by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX.

Syntax

idiv <reg32>
idiv <mem>

Examples

idiv ebx — divide the contents of EDX:EAX by the contents of EBX. Place the quotient in EAX and the remainder in EDX.

idiv DWORD PTR [var] — divide the contents of EDX:EAX by the 32-bit value stored at memory location *var*. Place the quotient in EAX and the remainder in EDX.

and, or, xor — Bitwise logical and, or and exclusive or

These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.

Syntax

```
and <reg>,<reg>
and <reg>,<mem>
and <mem>,<reg>
and <reg>,<con>
and <mem>,<con>
or <reg>,<reg>
or <reg>,<mem>
or <mem>,<reg>
or <reg>,<con>
or <mem>,<con>
xor <reg>,<reg>
xor <reg>,<mem>
xor <mem>,<reg>
xor <reg>,<con>
xor <mem>,<con>
```

Examples

and eax, 0fH — clear all but the last 4 bits of EAX.

xor edx, edx — set the contents of EDX to zero.

not — Bitwise Logical Not

Logically negates the operand contents (that is, flips all bit values in the operand).

Syntax

not <reg>

not <mem>

Example

not BYTE PTR [var] — negate all bits in the byte at the memory location *var*.

neg — Negate

Performs the two's complement negation of the operand contents.

Syntax

neg <reg>

neg <mem>

Example

neg eax — EAX → - EAX

shl, shr — Shift Left, Shift Right

These instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros. The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL. In either case, shifts counts of greater then 31 are performed modulo 32.

Syntax

shl <reg>,<con8>

shl <mem>,<con8>

shl <reg>,<cl>

shl <mem>,<cl>

shr <reg>,<con8>

shr <mem>,<con8>

shr <reg>,<cl>

shr <mem>,<cl>

Examples

shl eax, 1 — Multiply the value of EAX by 2 (if the most significant bit is 0)

shr ebx, cl — Store in EBX the floor of result of dividing the value of EBX by 2^n where*n* is the value in CL.

jmp — Jump

Transfers program control flow to the instruction at the memory location indicated by the operand.

Syntax

jmp <label>

Example

jmp begin — Jump to the instruction labeled begin.

jcondition — Conditional Jump

These instructions are conditional jumps that are based on the status of a set of condition codes that are stored in a special register called the *machine status word*. The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed. For example, the jz instruction performs a jump to the specified operand label if the result of the last arithmetic operation was zero. Otherwise, control proceeds to the next instruction in sequence.

A number of the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, cmp (see below). For example, conditional branches such as jle and jne are based on first performing a cmp operation on the desired operands.

Syntax

```
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)
```

Example

```
cmp eax, ebx
jle done
```

If the contents of EAX are less than or equal to the contents of EBX, jump to the label *done*. Otherwise, continue to the next instruction.

cmp — Compare

Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand.

Syntax

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

Example

```
cmp DWORD PTR [var], 10
jeq loop
```

If the 4 bytes stored at location *var* are equal to the 4-byte integer constant 10, jump to the location labeled *loop*.

call, **ret** — Subroutine call and return

These instructions implement a subroutine call and return. The call instruction first pushes the current code location onto the hardware supported stack in memory (see the push instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.

The ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack (see the pop instruction for details). It then performs an unconditional jump to the retrieved code location.

Syntax

```
call <label>
ret
```

A digression on signed vs unsigned conditional jumps

When comparing *signed* integers, you should use jl, jg, jle, and jge.
When comparing *unsigned* integers, you should use jb, ja, jbe, and jae.
Why?
And why don't we have separate conditionals for (in)equality?

Consider these two 16-bit numbers:

ax= 1000000110010001
bx= 0000000000110000

Which is bigger?

A digression on signed vs unsigned conditional jumps

When comparing *signed* integers, you should use jl, jg, jle, and jge.
When comparing *unsigned* integers, you should use jb, ja, jbe, and jae.
Why?
And why don't we have separate conditionals for (in)equality?

Consider these two 16-bit numbers:

		Unsigned	2's complement
ax=	**1000000110010001**	**33169**	**-32367**
bx=	**0000000000110000**	**48**	**48**

Which is bigger?

A digression on signed vs unsigned conditional jumps

When comparing *signed* integers, you should use jl, jg, jle, and jge.
When comparing *unsigned* integers, you should use jb, ja, jbe, and jae.
Why?
And why don't we have separate conditionals for (in)equality?

Consider these two 16-bit numbers:

		Unsigned	2's complement
ax=	**1000000110010001**	**33169**	**-32367**
bx=	**0000000000110000**	**48**	**48**

Which is bigger?

```
cmp ax, bx
jg foobar         ; -32367 > 48 -- WILL NOT branch

cmp ax, bx
ja foobar         ; 33169 > 48 -- WILL branch
```

A digression on signed vs unsigned conditional jumps

When comparing *signed* integers, you should use jl, jg, jle, and jge.
When comparing *unsigned* integers, you should use jb, ja, jbe, and jae.
Why?
And why don't we have separate conditionals for (in)equality?

Comparison	Signed	Unsigned
<	**jl** jump if less	**jb** jump if below
>	**jg** jump if greater	**ja** jump if above
<=	**jle** jump if less or equal	**jbe** jump if below or equal
>=	**jge** jump if greater or equal	**jae** jump if above or equal
==	**je/jz** **jump if equal/jump if zero**	
!=	**jne/jnz** jump if not equal/jump if not zero	

Unsigned comparison implementation

Unsigned comparison	jb label ja label jbe label jae label	jump if CF jump if !CF && !ZF jump if CF || ZF jump if !CF

As on E15, cmp is implemented in terms of subtraction.

Zero flag (ZF) is set when the difference between two operands is zero.

Carry flag (CF) is set when an addition results in a carry, or a subtraction results in a borrow

```
  1011
- 0111
------
  0100
```

```
  0011
- 1011
------
  1100
```

Unsigned comparison implementation

Unsigned comparison	jb label	jump if CF
	ja label	jump if !CF && !ZF
	jbe label	jump if CF || ZF
	jae label	jump if !CF

As on E15, cmp is implemented in terms of subtraction.

Zero flag (ZF) is set when the difference between two operands is zero.

Carry flag (CF) is set when an addition results in a carry, or a subtraction results in a borrow

```
  1011
- 0111
------
  0100
```

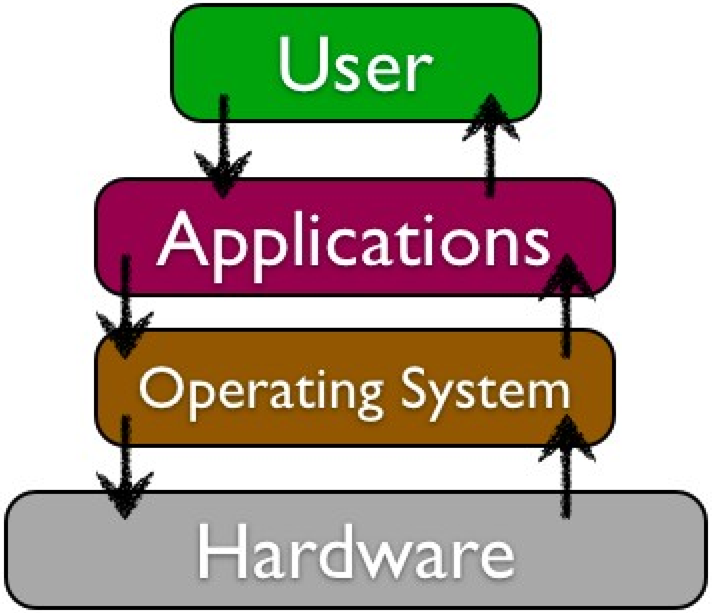
11 < 7?

no borrow required, CF not set

```
  0011
- 1011
------
  1100
```

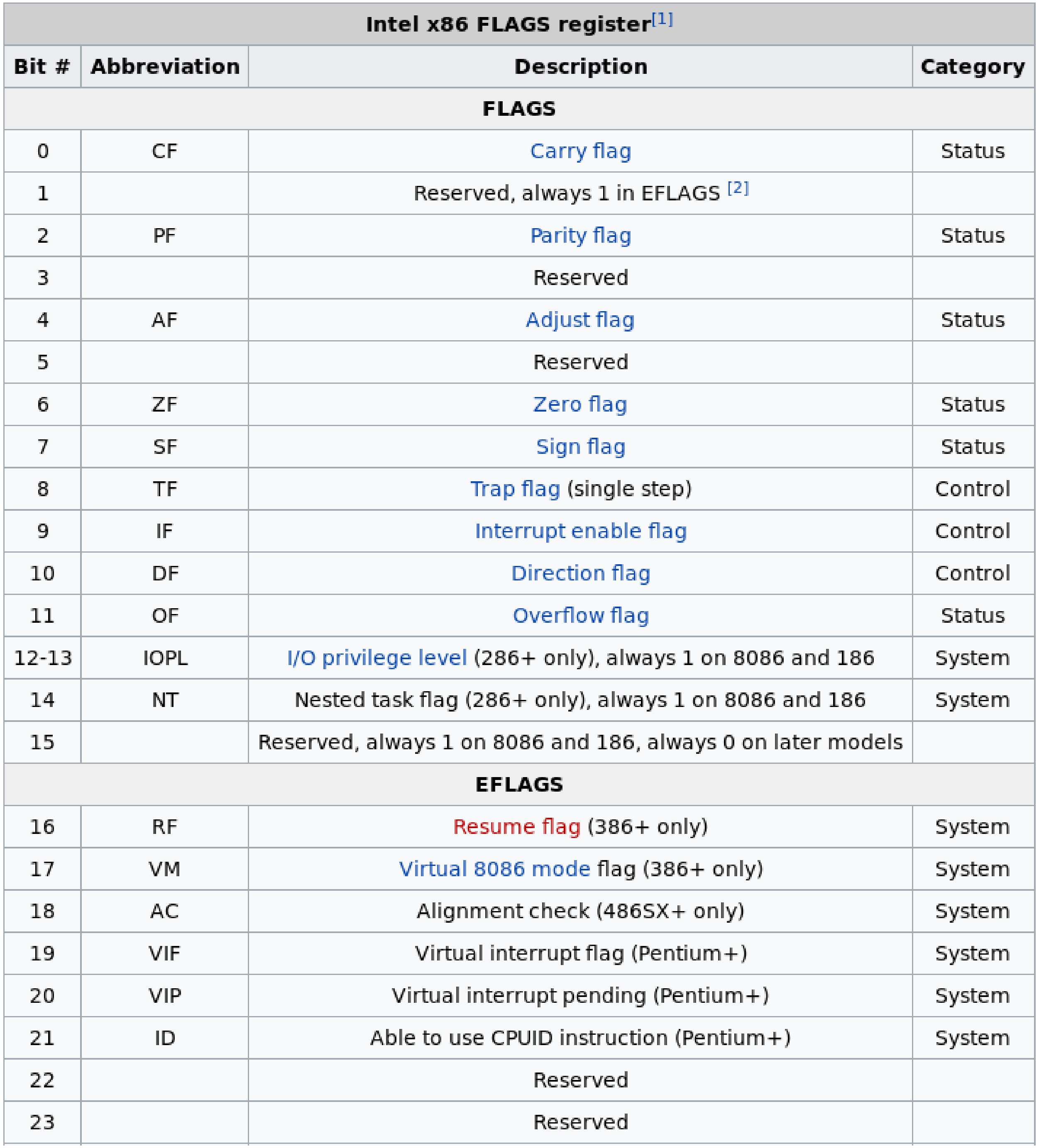
7 < 11?

borrow required, CF set



We will focus on application software. Such programs are typically invoked by the operating system (for example, when the user clicks on the appropriate file). Application software communicates with the operating system by way of *system calls*, by which means it can request services, such as:

Accessing a file on disk
Accessing the network
Interfacing with the user (e.g. screen, keyboard, mouse, etc)
Interfacing with other applications (shared memory, pipes, etc)



Intel x86 FLAGS register[1]

Bit #	Abbreviation	Description	Category
FLAGS			
0	CF	Carry flag	Status
1		Reserved, always 1 in EFLAGS [2]	
2	PF	Parity flag	Status
3		Reserved	
4	AF	Adjust flag	Status
5		Reserved	
6	ZF	Zero flag	Status
7	SF	Sign flag	Status
8	TF	Trap flag (single step)	Control
9	IF	Interrupt enable flag	Control
10	DF	Direction flag	Control
11	OF	Overflow flag	Status
12-13	IOPL	I/O privilege level (286+ only), always 1 on 8086 and 186	System
14	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System
15		Reserved, always 1 on 8086 and 186, always 0 on later models	
EFLAGS			
16	RF	Resume flag (386+ only)	System
17	VM	Virtual 8086 mode flag (386+ only)	System
18	AC	Alignment check (486SX+ only)	System
19	VIF	Virtual interrupt flag (Pentium+)	System
20	VIP	Virtual interrupt pending (Pentium+)	System
21	ID	Able to use CPUID instruction (Pentium+)	System
22		Reserved	
23		Reserved	

etc, etc... the rest are reserved.

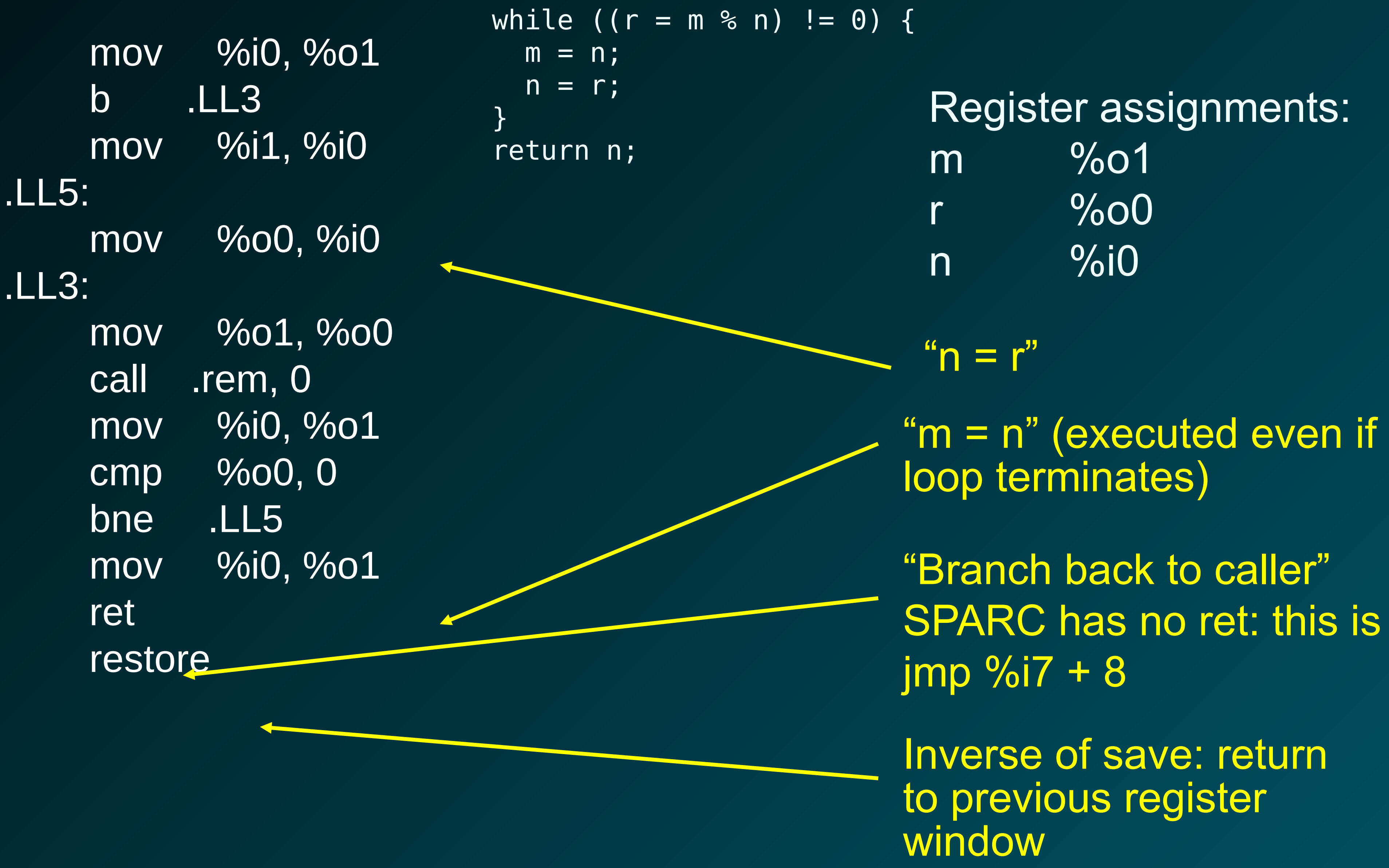
You can't access EFLAGS directly, but you can put it on the stack and pop it from the stack with the PUSHF/PUSHFD and POPF/POPFD instructions.

In addition, some flag bits can be manipulated individually, e.g. CLD/STD will clear or set the direction flag; CLI/STI will clear or set the interrupt flag.

The conditional jump (Jcc) instructions use several flag bits as input, e.g. JC will jump when the carry flag is set.

Most arithmetic and compare instructions will set some flags.

Euclid’s Algorithm on the SPARC



```
        mov     %i0, %o1
        b       .LL3
        mov     %i1, %i0
.LL5:
        mov     %o0, %i0
.LL3:
        mov     %o1, %o0
        call    .rem, 0
        mov     %i0, %o1
        cmp     %o0, 0
        bne     .LL5
        mov     %i0, %o1
        ret
        restore
```

```
while ((r = m % n) != 0) {
  m = n;
  n = r;
}
return n;
```

Register assignments:

m	%o1
r	%o0
n	%i0

“n = r”

“m = n” (executed even if loop terminates)

“Branch back to caller”
SPARC has no ret: this is
jmp %i7 + 8

Inverse of save: return to previous register window