CS 3330 introduction

layers of abstraction

`x += y`

“Higher-level” language: C

`add %rbx, %rax`

Assembly: X86-64

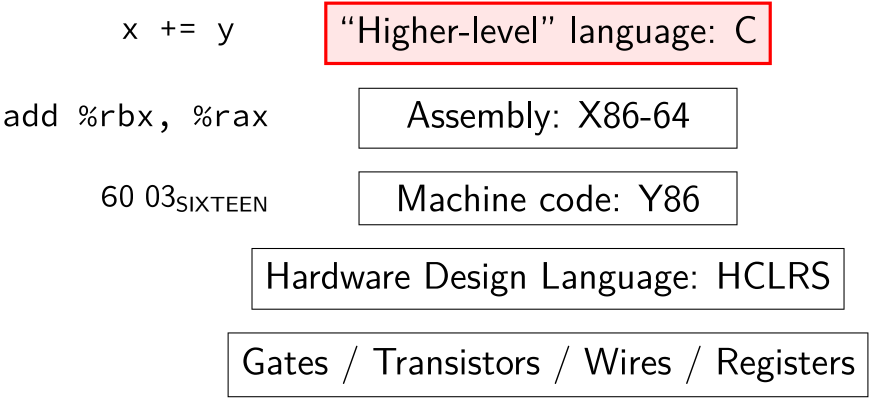
$60\ 03_{\text{SIXTEEN}}$

Machine code: Y86

Hardware Design Language: HCLRS

Gates / Transistors / Wires / Registers

layers of abstraction



`x += y`

“Higher-level” language: C

`add %rbx, %rax`

Assembly: X86-64

60 03$_{\text{SIXTEEN}}$

Machine code: Y86

Hardware Design Language: HCLRS

Gates / Transistors / Wires / Registers

why C?

almost a subset of C++

- notably removes classes, new/delete, iostreams
- other changes, too, so C code often not valid C++ code

direct correspondence to assembly

why C?

almost a subset of C++

- notably removes classes, new/delete, iostreams
- other changes, too, so C code often not valid C++ code

direct correspondence to assembly

Should help you understand machine!
Manual translation to assembly

why C?

almost a subset of C++

- notably removes classes, new/delete, iostreams
- other changes, too, so C code often not valid C++ code

direct correspondence to assembly

But "clever" (optimizing) compiler might be confusingly indirect instead

homework: C environment

get Unix environment with a C compiler

will have department accounts, hopefully by end of week

- portal.cs.virginia.edu or NX
- instructions off course website (Collab)

some other options:

- Linux (native or VM)
    - 2150 VM image should work
- some assignments can use OS X natively
- some assignments can Windows Subsystem for Linux natively

assignment compatibility

supported platform: department machines

many use laptops

trouble? we'll say to use department machines

most assignments: C and Unix-like environment

also: tool written in Rust — but we'll provide binaries

previously written in D + needed D compiler

layers of abstraction

`x += y` “Higher-level” language: C

`add %rbx, %rax` Assembly: X86-64

$60\ 03_{\text{SIXTEEN}}$ Machine code: Y86

Hardware Design Language: HCLRS

Gates / Transistors / Wires / Registers

X86-64 assembly

in theory, you know this (CS 2150)

in reality, ...

layers of abstraction

`x += y` “Higher-level” language: C

`add %rbx, %rax` Assembly: X86-64

60 03_{SIXTEEN} Machine code: Y86

Hardware Design Language: HCLRS

Gates / Transistors / Wires / Registers

Y86-64??

Y86: our textbook's X86-64 subset

much simpler than real X86-64 encoding

- (which we will not cover)

not as simple as 2150's IBCM

- variable-length encoding
- more than one register
- full conditional jumps
- stack-manipulation instructions

layers of abstraction

`x += y`	“Higher-level” language: C
`add %rbx, %rax`	Assembly: X86-64
$60\ 03_{\text{SIXTEEN}}$	Machine code: Y86
	Hardware Design Language: HCLRS
	Gates / Transistors / Wires / Registers

goals/other topics

understand how hardware works for...

program performance

what compilers are/do

weird program behaviors (segfaults, etc.)

goals/other topics

understand how hardware works for...

program performance

what compilers are/do

weird program behaviors (segfaults, etc.)

program performance

naive model:

- one instruction = one time unit

number of instructions matters, but …

program performance: issues

parallelism

fast hardware is parallel
needs multiple things to do

caching

accessing things recently accessed is faster
need reuse of data/code

(more in other classes: algorithmic efficiency)

goals/other topics

understand how hardware works for...

program performance

what compilers are/do

weird program behaviors (segfaults, etc.)

what compilers are/do

understanding weird compiler/linker rrors

if you want to make compilers

debugging applications

goals/other topics

understand how hardware works for...

program performance

what compilers are/do

weird program behaviors (segfaults, etc.)

weird program behaviors

what is a segmentation fault really?

how does the operating system interact with programs?

if you want to handle them — writing OSs

interlude: powers of two

	…	
2^{0}	1	
2^{1}	2	
2^{2}	4	
2^{3}	8	
2^{4}	16	
2^{5}	32	
2^{6}	64	
2^{7}	128	
2^{8}	256	
2^{9}	512	
2^{10}	**1 024**	**K** (or Ki)
	…	
2^{11}	2 048	
2^{12}	4 096	
2^{13}	8 192	
2^{14}	16 384	
2^{15}	32 768	
2^{16}	65 536	
	…	
2^{20}	1 048 576	**M** (or Mi)
	…	
2^{30}	1 073 741 824	**G** (or Gi)
2^{31}	2 147 483 648	
2^{32}	4 294 967 296	
	…	

powers of two: forward

2^{35}

2^{21}

2^{9}

2^{14}

powers of two: forward

$2^{35} = 2^5 \cdot 2^{30} = 32G$ (30 = G)

2^{21}

2^{9}

2^{14}

powers of two: forward

$2^{35} = 2^5 \cdot 2^{30} = 32G$ (30 = G)

2^{21}

2^{9}

2^{14}

powers of two: forward

$2^{35} = 2^5 \cdot 2^{30} = 32G$ (30 = G)

$2^{21} = 2^1 \cdot 2^{20} = 2M$ (20 = M)

2^9

2^{14}

powers of two: forward

$2^{35} = 2^5 \cdot 2^{30} = 32G$ (30 = G)

$2^{21} = 2^1 \cdot 2^{20} = 2M$ (20 = M)

$2^9 = 512$

2^{14}

powers of two: forward

$2^{35} = 2^5 \cdot 2^{30} = 32G$ (30 $=$ G)

$2^{21} = 2^1 \cdot 2^{20} = 2M$ (20 $=$ M)

$2^9 = 512$

$2^{14} = 2^4 \cdot 2^{10} = 16K$

powers of two: backward

16G

128K

4M

256T

powers of two: backward

16G $= 16 \cdot 2^{30} = 2^{30+4} = 2^{34}$

128K

4M

256T

powers of two: backward

16G $= 16 \cdot 2^{30} = 2^{30+4} = 2^{34}$

128K $= 128 \cdot 2^{10} = 2^{10+7} = 2^{17}$

4M

256T

powers of two: backward

16G $= 16 \cdot 2^{30} = 2^{30+4} = 2^{34}$

128K $= 128 \cdot 2^{10} = 2^{10+7} = 2^{17}$

4M $= 4 \cdot 2^{20} = 2^{20+2} = 2^{22}$

256T $= 256 \cdot 2^{40} = 2^{40+8} = 2^{48}$

lecturers

Venkat and I will be splitting the course

- most lectures in first half of semester: me
- most lectures in second half of semester: Venkat

same(ish) lecture in each section

coursework (1)

labs — grading: did you make reasonable progress?

- collaboration permitted

homework assignments — introduced by lab (mostly)

- due at noon on the next lab day (mostly)
- complete individually

exams

coursework (2)

quizzes

early (my) part of course

- weekly pre/post-quizzes, starting around add deadline
- pre-quizzes based on *readings*
- post-quizzes based on lectures

later part of course

- Venkat plans problem sets to replace quizzes
- sometimes (mostly?) tied with homework, sometimes seperate

on lecture/lab/HW synchronization

labs/HWs not quite synchronized with lectures

main problem: want to cover material **before you need it** in lab/HW

quizzes?

linked off course website (demo)

pre-quiz, on reading – released by Saturday evening, due Tuesdays before lecture

post-quiz, on lecture topics — released Thursday evening, due following Monday afternoon

each quiz 90 minute time limit (+ adjustments if SDAC says)

lowest 10% (approx. 2 quizzes) will be dropped

first quiz — next week's post-quiz

attendance?

lecture: strongly recommended.

I will record my lectures to help you review

lab: often electronic, remote-possible submission

late policy

exceptional circumstance? contact us.

otherwise, for homeworks only:

- -10% 0 to 48 hours late
- -15% 48 to 72 hours late
- -100% otherwise

late quizzes, labs: no

- we release answers
- talk to us if illness, etc.

TAs/Office Hours

office hours will be posted on calendar on the website

should be plenty

use them

your TODO list

department account and/or C environment working

department accounts should happen by this weekend

before lab next week

grading

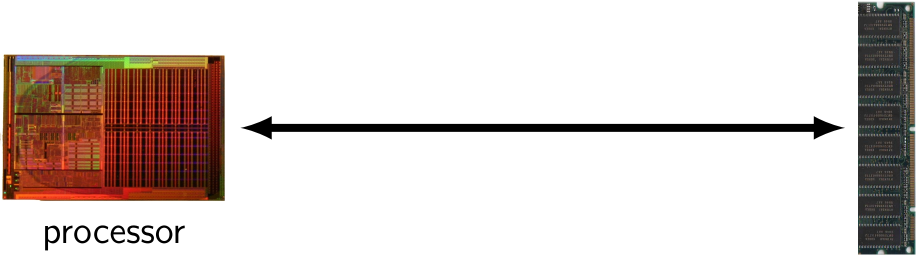
Quizzes: 5% [tentative] (10% dropped)

Midterms (2): 30%

Final Exam (cumulative): 20%

Homework + Labs + Problem Sets: 45%

processors and memory

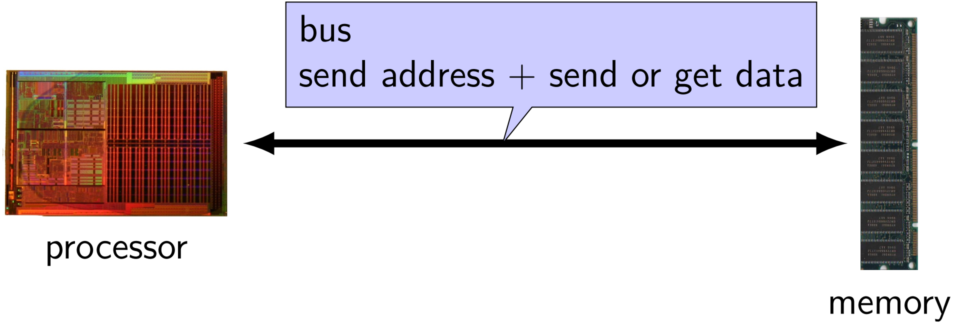


processor

memory

Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



bus

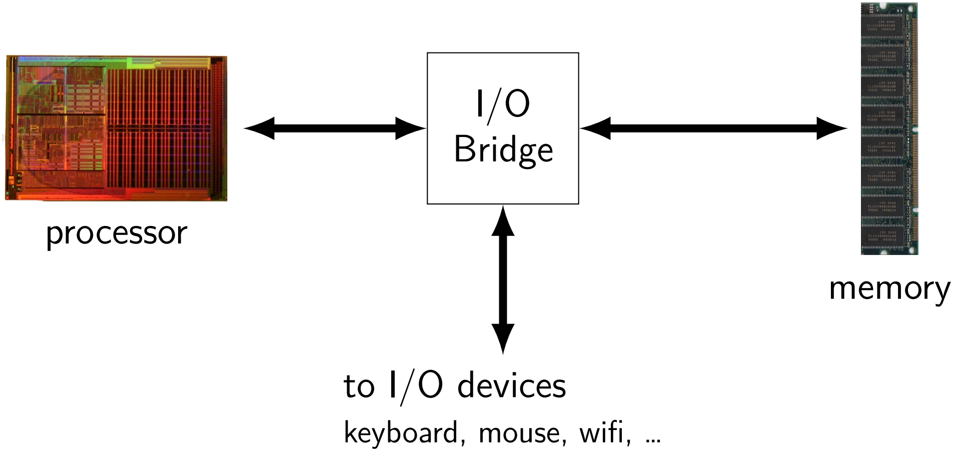
send address + send or get data

processor

memory

Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



processor

I/O Bridge

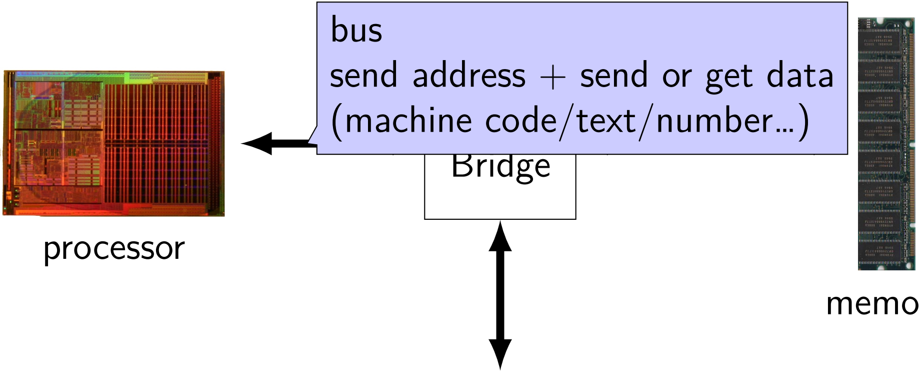
memory

to I/O devices

keyboard, mouse, wifi, …

Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



bus
send address + send or get data
(machine code/text/number...)

Bridge

processor

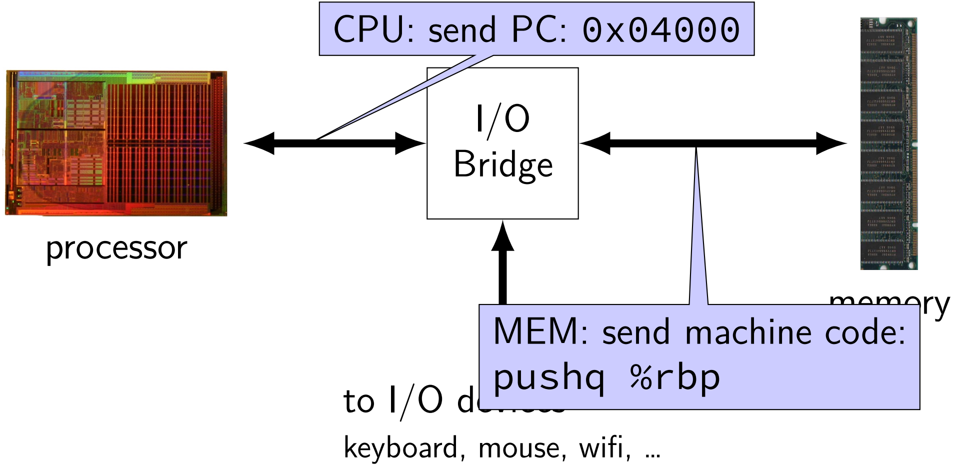
memory

to I/O devices

keyboard, mouse, wifi, ...

Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



CPU: send PC: `0x04000`

processor

I/O Bridge

memory

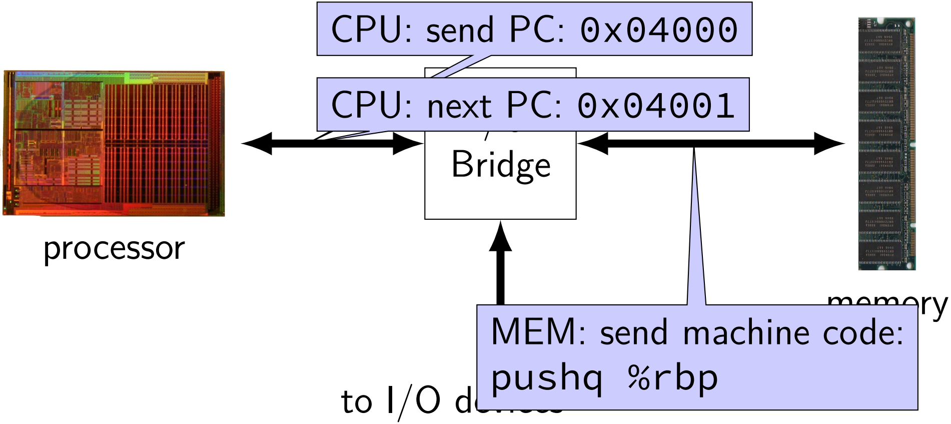
MEM: send machine code: `pushq %rbp`

to I/O devices

keyboard, mouse, wifi, …

Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



CPU: send PC: `0x04000`

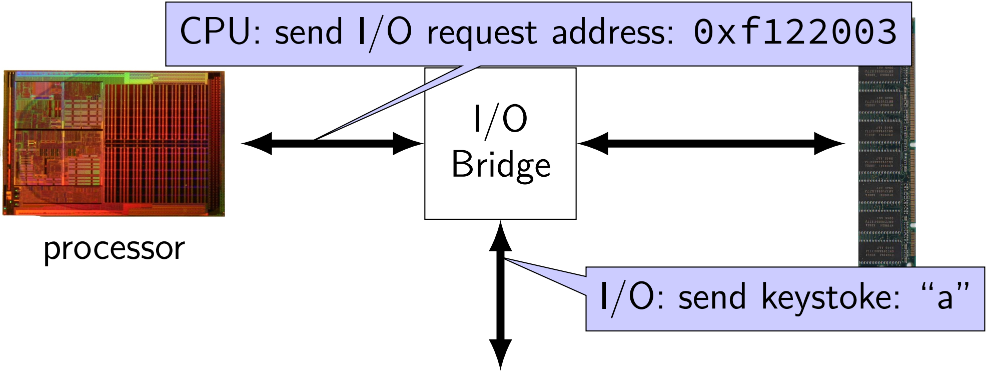
CPU: next PC: `0x04001`

MEM: send machine code:
`pushq %rbp`

keyboard, mouse, wifi, …

Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



CPU: send I/O request address: `0xf122003`

I/O Bridge

processor

I/O: send keystoke: “a”

to I/O devices

keyboard, mouse, wifi, …

Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

memory

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
…	…
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
…	…
0x00000002	0xFE
0x00000001	0xE0
0x00000000	0xA0

memory

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
…	…
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
…	…
0x00000002	0xFE
0x00000001	0xE0
0x00000000	0xA0

array of bytes (byte = 8 bits)
CPU interprets based on how accessed

memory

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
…	…
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
…	…
0x00000002	0xFE
0x00000001	0xE0
0x00000000	0xA0

address	value
0x00000000	0xA0
0x00000001	0xE0
0x00000002	0xFE
…	…
0x00041FFE	0x60
0x00041FFF	0x03
0x00042000	0x00
0x00042001	0x01
0x00042002	0x02
0x00042003	0x03
0x00042004	0x04
0x00042005	0x05
0x00042006	0x06
…	…
0xFFFFFFFD	0xDE
0xFFFFFFFE	0x45
0xFFFFFFFF	0x14

endianness

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
…	…
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
…	…
0x00000002	0xFE
0x00000001	0xE0

```
int *x = (int*)0x42000;
cout << *x << endl;
```

endianness

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
…	…
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
…	…
0x00000002	0xFE
0x00000001	0xE0

```
int *x = (int*)0x42000;
cout << *x << endl;
```

endianness

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
…	…
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
…	…
0x00000002	0xFE
0x00000001	0xE0

```
int *x = (int*)0x42000;
cout << *x << endl;
```

0x03020100 = 50462976

0x00010203 = 66051

endianness

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
…	…
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
…	…
0x00000002	0xFE
0x00000001	0xE0

```
int *x = (int*)0x42000;
cout << *x << endl;
```

$\texttt{0x03020100} = \boxed{50462976}$

little endian
(least significant byte has lowest address)

$\texttt{0x00010203} = \boxed{66051}$

big endian
(most significant byte has lowest address)

endianness

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
…	…
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
…	…
0x00000002	0xFE
0x00000001	0xE0

```
int *x = (int*)0x42000;
cout << *x << endl;
```

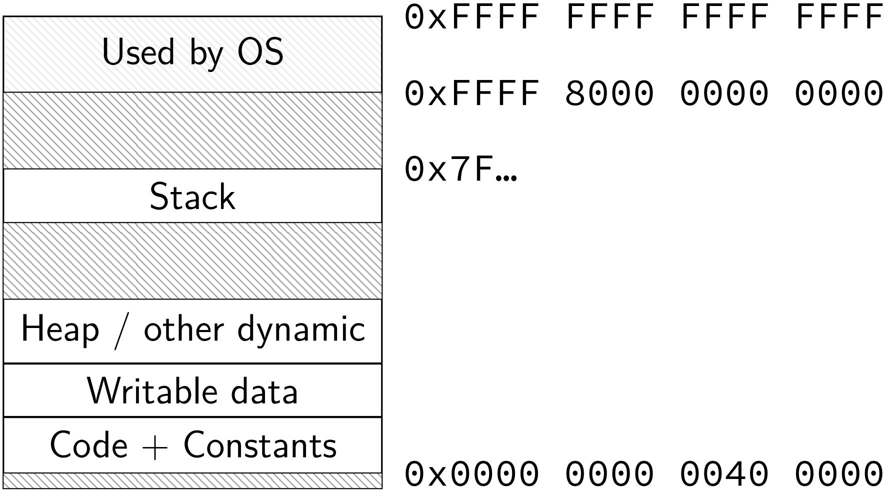
`0x03020100` $= \boxed{50462976}$

little endian
(least significant byte has lowest address)

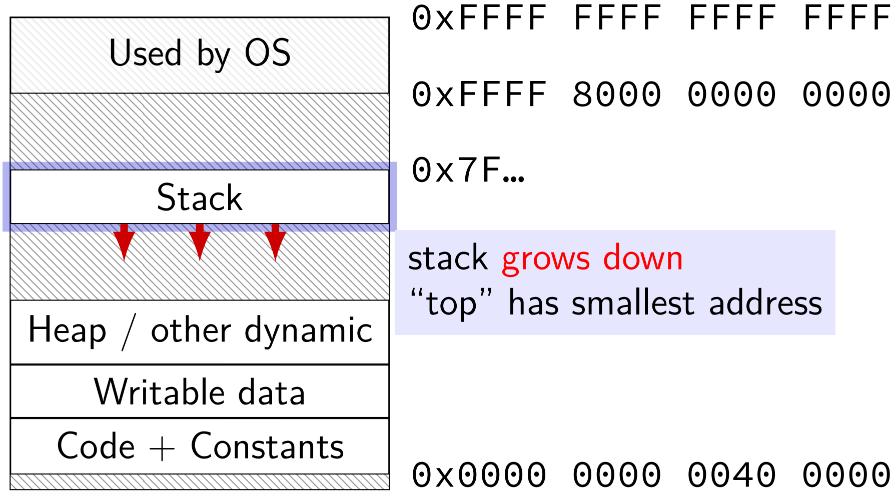
`0x00010203` $= \boxed{66051}$

big endian
(most significant byte has lowest address)

program memory (x86-64 Linux)

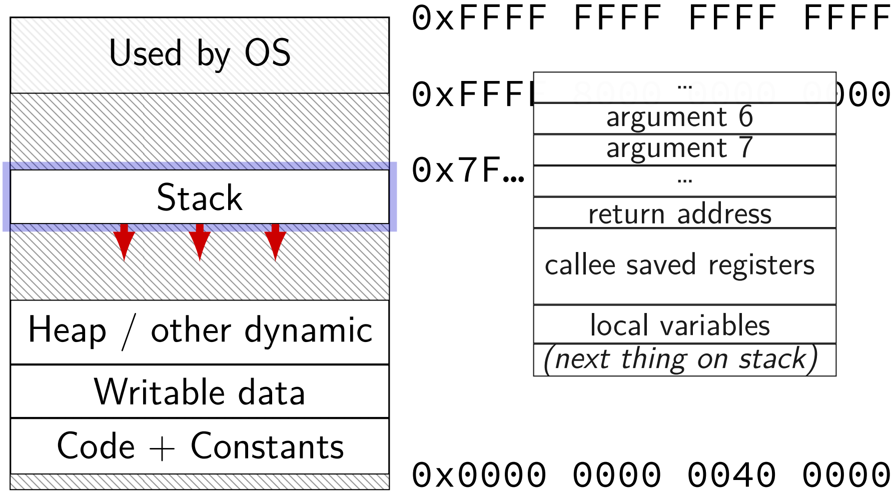


program memory (x86-64 Linux)



stack grows down
“top” has smallest address

program memory (x86-64 Linux)



Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

0x0000 0000 0040 0000

…

argument 6

argument 7

…

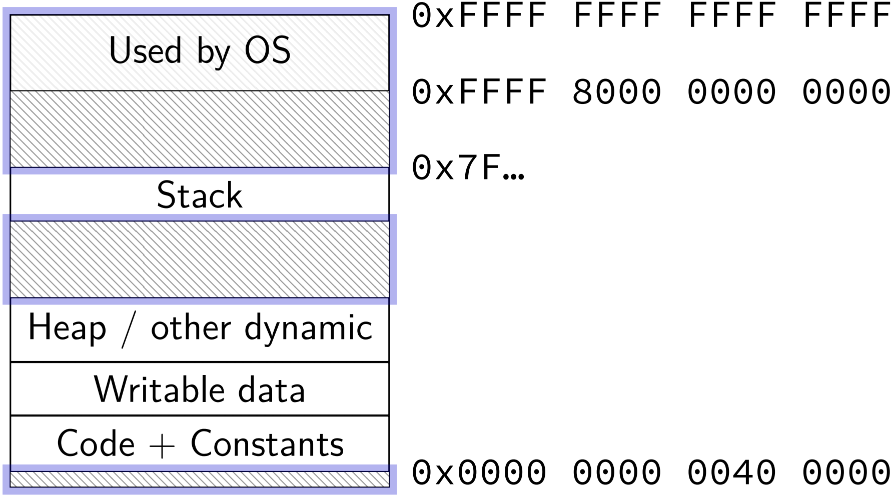
return address

callee saved registers

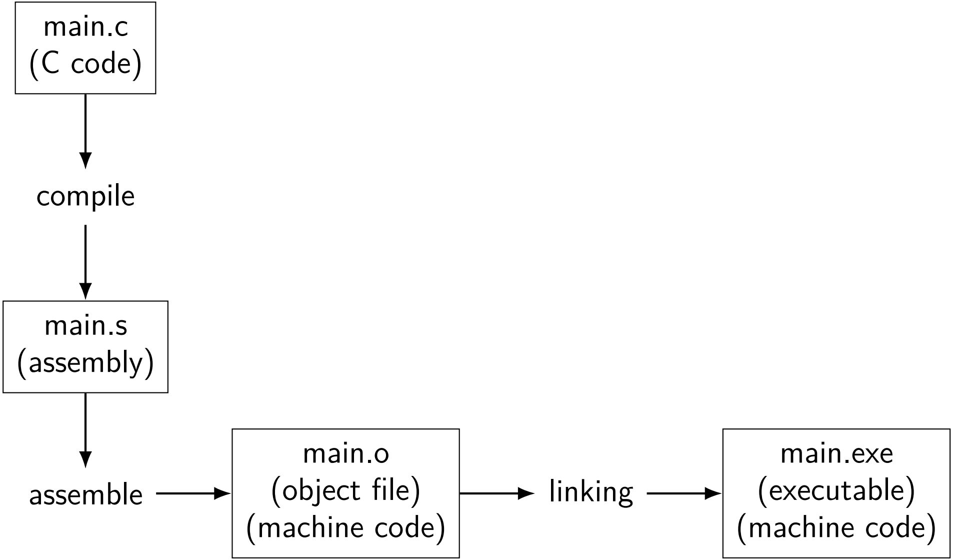
local variables

(next thing on stack)

program memory (x86-64 Linux)



compilation pipeline



main.c
(C code)

compile

main.s
(assembly)

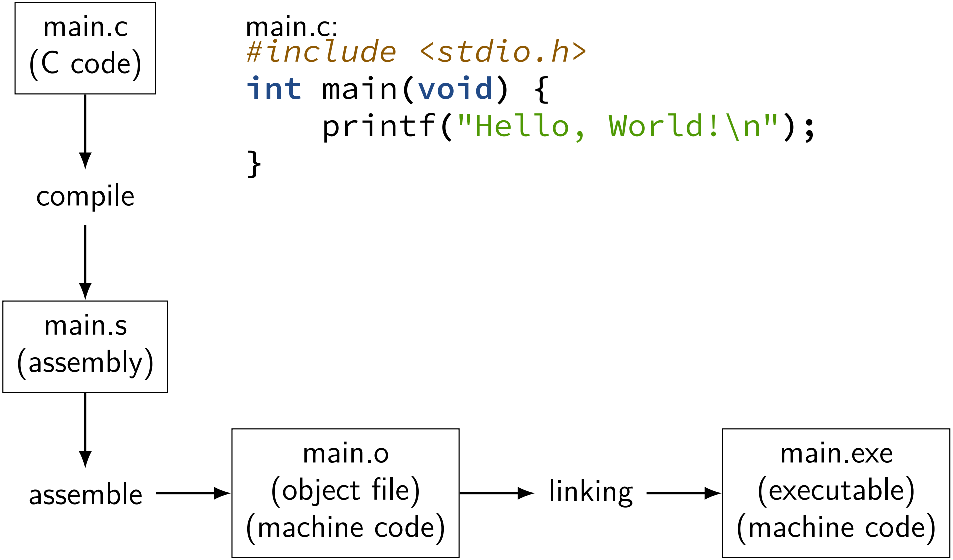
assemble

main.o
(object file)
(machine code)

linking

main.exe
(executable)
(machine code)

compilation pipeline



main.c (C code)

compile

main.s (assembly)

assemble

main.o (object file) (machine code)

linking

main.exe (executable) (machine code)

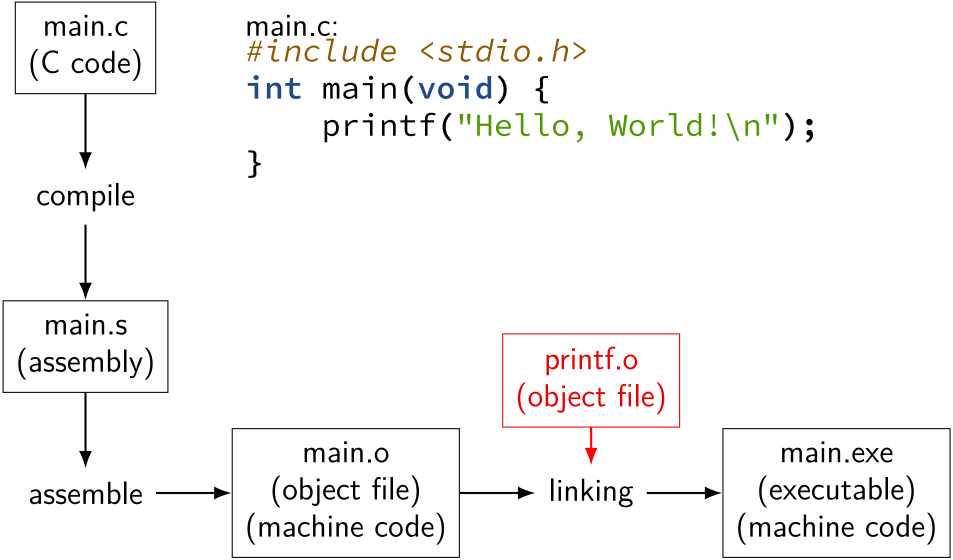
main.c:

```
#include <stdio.h>
int main(void) {
    printf("Hello, World!\n");
}
```

compilation pipeline

main.c:

```
#include <stdio.h>
int main(void) {
    printf("Hello, World!\n");
}
```



main.c
(C code)

compile

main.s
(assembly)

assemble

main.o
(object file)
(machine code)

printf.o
(object file)

linking

main.exe
(executable)
(machine code)

compilation commands

compile:	`gcc -S file.c`	⇒	`file.s` (assembly)
assemble:	`gcc -c file.s`	⇒	`file.o` (object file)
link:	`gcc -o file file.o`	⇒	`file` (executable)
c+a:	`gcc -c file.c`	⇒	`file.o`
c+a+l:	`gcc -o file file.c`	⇒	`file`

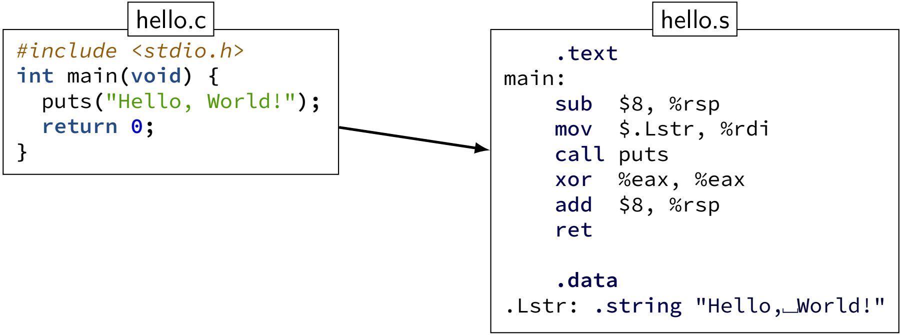
...

what’s in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

what's in those files?



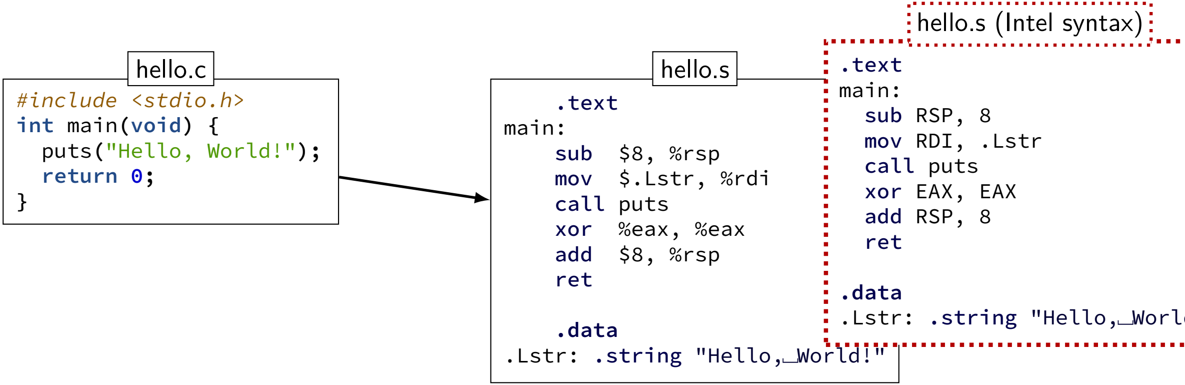
hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

what’s in those files?



hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

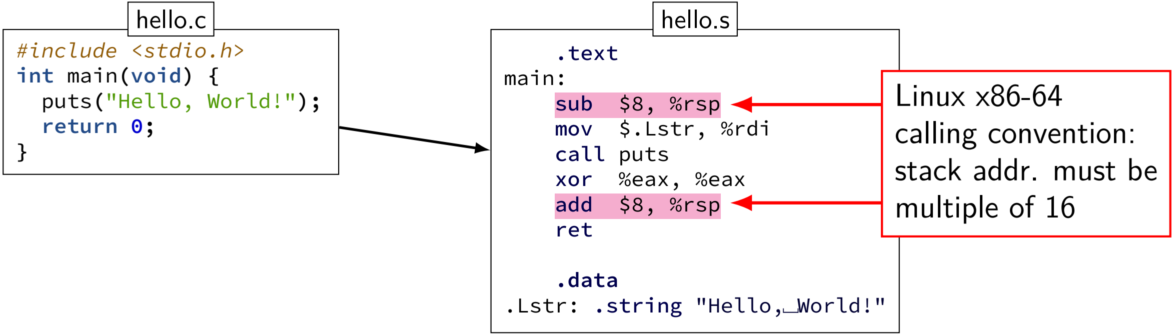
hello.s

```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.s (Intel syntax)

```
.text
main:
  sub RSP, 8
  mov RDI, .Lstr
  call puts
  xor EAX, EAX
  add RSP, 8
  ret

.data
.Lstr: .string "Hello,␣Worl
```

what’s in those files?



hello.c

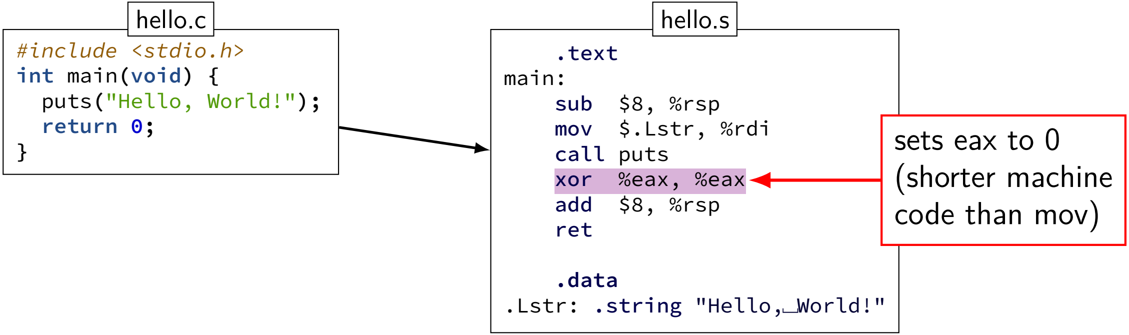
```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
main:
    sub   $8, %rsp
    mov   $.Lstr, %rdi
    call  puts
    xor   %eax, %eax
    add   $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

Linux x86-64 calling convention: stack addr. must be multiple of 16

what's in those files?



hello.c

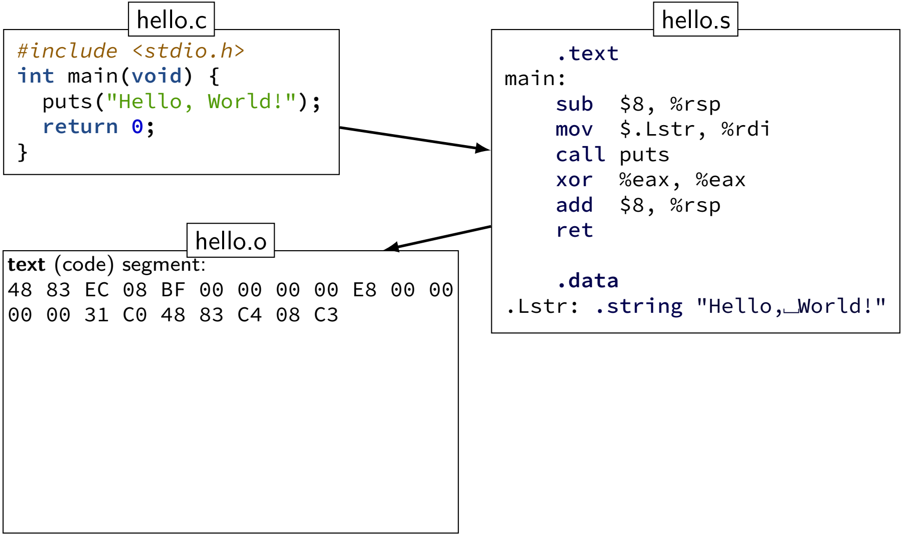
```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

sets eax to 0 (shorter machine code than mov)

what’s in those files?



hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

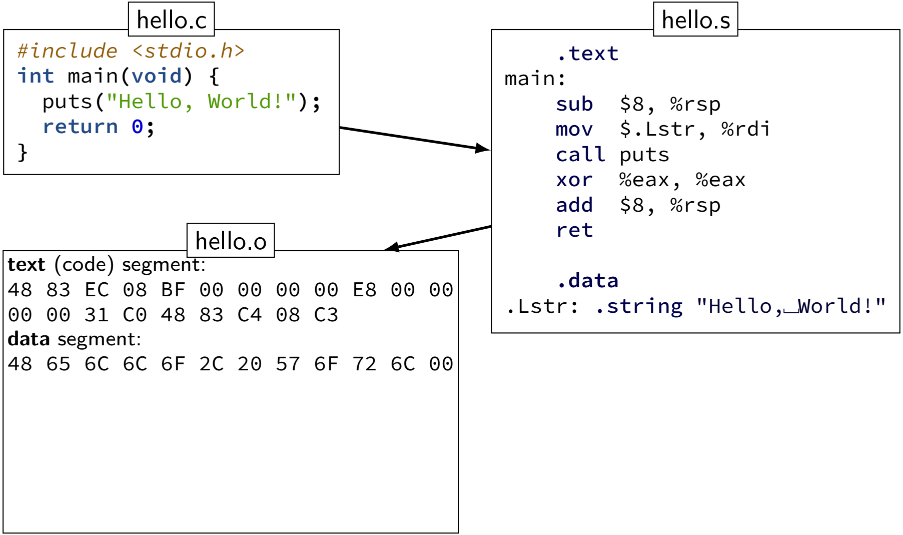
```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

what's in those files?



hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

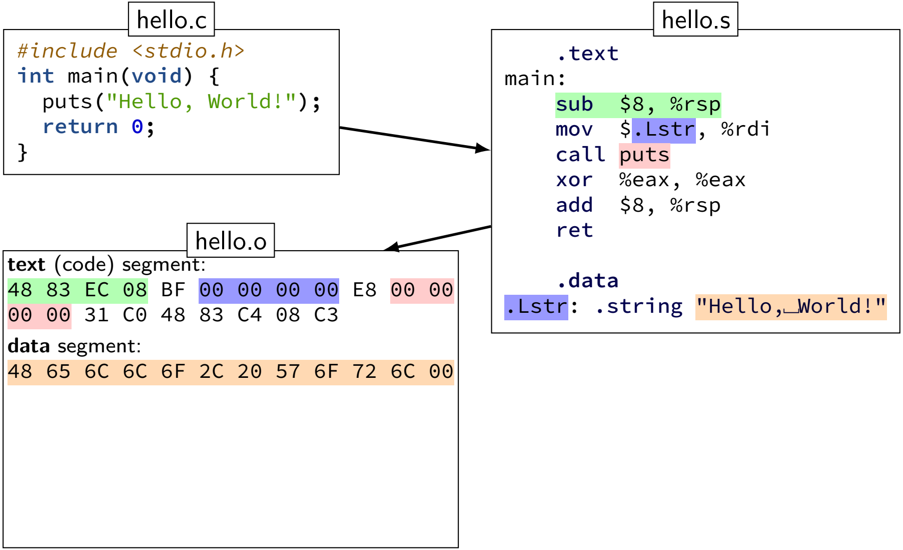
text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

what's in those files?



hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
     .text
main:
     sub  $8, %rsp
     mov  $.Lstr, %rdi
     call puts
     xor  %eax, %eax
     add  $8, %rsp
     ret

     .data
.Lstr: .string "Hello,␣World!"
```

hello.o

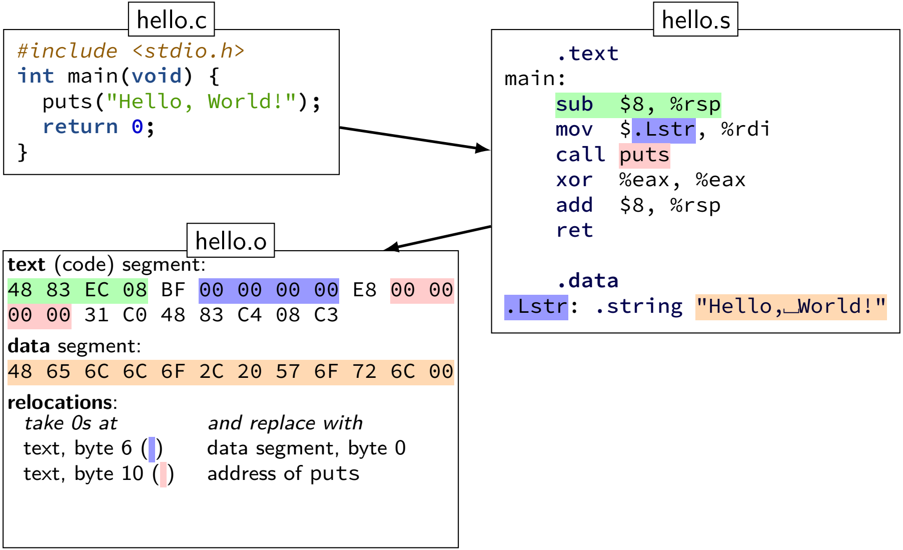
text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

what's in those files?



hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
main:
    sub   $8, %rsp
    mov   $.Lstr, %rdi
    call  puts
    xor   %eax, %eax
    add   $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

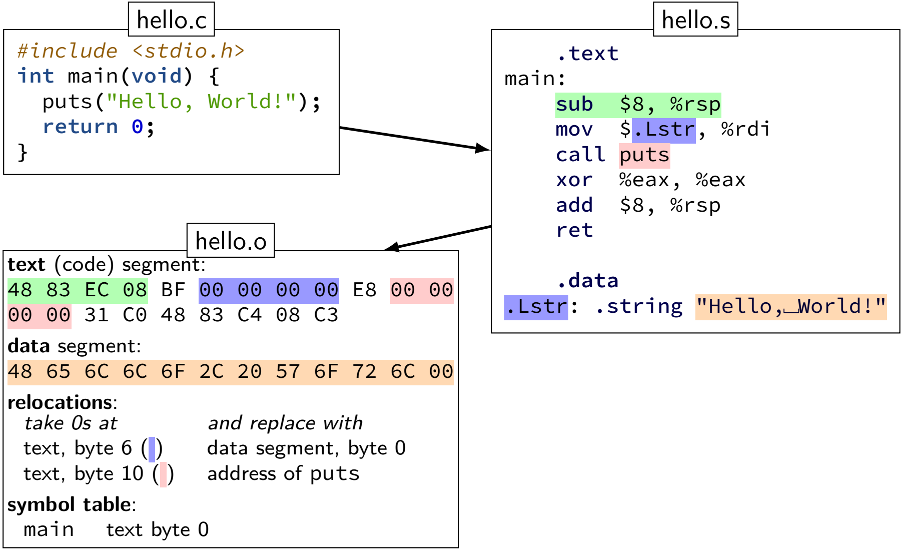
data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at	*and replace with*
text, byte 6 (🟦)	data segment, byte 0
text, byte 10 (🟥)	address of puts

what's in those files?



hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
main:
    sub   $8, %rsp
    mov   $.Lstr, %rdi
    call  puts
    xor   %eax, %eax
    add   $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

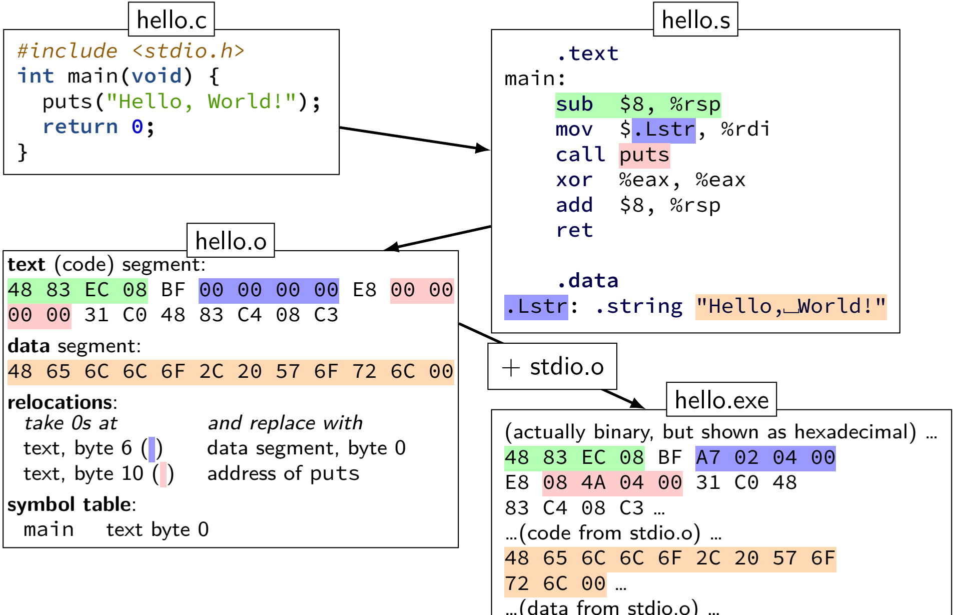
relocations:

take 0s at	*and replace with*
text, byte 6 (🟦)	data segment, byte 0
text, byte 10 (🟥)	address of puts

symbol table:

main	text byte 0

what’s in those files?



hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at	*and replace with*
text, byte 6 (🟦)	data segment, byte 0
text, byte 10 (🟥)	address of puts

symbol table:

`main` text byte 0

+ stdio.o

hello.exe

(actually binary, but shown as hexadecimal) …

```
48 83 EC 08 BF A7 02 04 00
E8 08 4A 04 00 31 C0 48
83 C4 08 C3 …
```

…(code from stdio.o) …

```
48 65 6C 6C 6F 2C 20 57 6F
72 6C 00 …
```

…(data from stdio.o) …

hello.s

```
        .section        .rodata.str1.1,"aMS",@progbi
.LC0:
        .string "Hello, World!"
        .text
        .globl  main
main:
        subq    $8, %rsp
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        addq    $8, %rsp
        ret
```

exercise (1)

main.c:

```c
#include <stdio.h>
void sayHello(void) {
    puts("Hello, World!");
}
int main(void) {
    sayHello();
}
```

Which files contain the **memory address** of `sayHello`?

A. main.s (assembly)

B. main.o (object)

C. main.exe (executable)

D. B and C

E. A, B and C

F. something else

exercise (2)

main.c:

```
#include <stdio.h>
void sayHello(void) {
    puts("Hello, World!");
}
int main(void) {
    sayHello();
}
```

Which files contain the **literal ASCII string** of `Hello, World!`?

A. main.s (assembly)
B. main.o (object)
C. main.exe (executable)
D. B and C
E. A, B and C
F. something else

relocation types

machine code doesn't always use addresses as is

"call function 4303 bytes later"

linker needs to compute "4303"

- extra field on relocation list

dynamic linking (very briefly)

dynamic linking — done when application is loaded

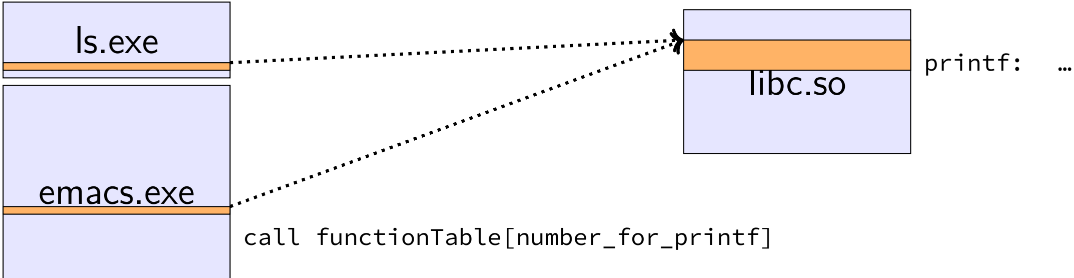
- idea: don't have N copies of `printf` on disk
- other type of linking: *static* (`gcc -static`)

load executable file + its libraries into memory when app starts

often extra indirection:

```
call functionTable[number_for_printf]
```

linker fills in `functionTable` instead of changing `calls`



```
call functionTable[number_for_printf]
```

ldd /bin/ls

```
$ ldd /bin/ls
    linux-vdso.so.1 =>  (0x00007ffcca9d8000)
    libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1
            (0x00007f851756f000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
            (0x00007f85171a5000)
    libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
            (0x00007f8516f35000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
            (0x00007f8516d31000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f8517791000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
            (0x00007f8516b14000)
```

backup slides

objdump -sx test.o (Linux) (1)

```
test.o:     file format elf64–x86–64
test.o
architecture: i386:x86–64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000000  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000040  2**0
                  ALLOC
  3 .rodata.str1.1 0000000e  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .text.startup 00000014  0000000000000000  0000000000000000  0000004e  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  5 .comment      0000002b  0000000000000000  0000000000000000  00000062  2**0
                  CONTENTS, READONLY
  6 .note.GNU–stack 00000000  0000000000000000  0000000000000000  0000008d  2**0
                  CONTENTS, READONLY
  7 .eh_frame     00000030  0000000000000000  0000000000000000  00000090  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

objdump -sx test.o (Linux) (2)

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata.str1.1 0000000000000000 .rodata.str1.1
0000000000000000 l    d  .text.startup  0000000000000000 .text.startup
0000000000000000 l    d  .note.GNU–stack        0000000000000000 .note.GNU–stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l       .rodata.str1.1 0000000000000000 .LC0
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text.startup  0000000000000014 main
0000000000000000         *UND*  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*  0000000000000000 puts
```

columns:

- memory address (not yet assigned, so 0)
- flags: `l`=local, `g`=global, `F`=function, ...
- section (.text, .data, .bss, ...)
- offset in section
- name of symbol

objdump -sx test.o (Linux) (3)

```
RELOCATION RECORDS FOR [.text.startup]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_PC32     .LC0—0x0000000000000004
000000000000000c R_X86_64_PLT32    puts—0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text.startup

Contents of section .rodata.str1.1:
 0000 48656c6c 6f2c2057 6f726c64 2100      Hello, World!.
Contents of section .text.startup:
 0000 488d3d00 00000048 83ec08e8 00000000  H.=....H........
 0010 31c05ac3                             1.Z.
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520372e  .GCC: (Ubuntu 7.
 0010 332e302d 32377562 756e7475 317e3138  3.0—27ubuntu1~18
 0020 2e303429 20372e33 2e3000             .04) 7.3.0.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 14000000 1c000000  ................
 0020 00000000 14000000 004b0e10 480e0800  .........K..H...
```