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

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

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

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

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

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

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

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

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_{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_{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

options:

- lab accounts + SSH
- Linux (native or VM)
  - 2150 VM image should work
- online IDE (e.g. Cloud9, Koding)

assignment compatibility

supported platform: lab machines

many use laptops

trouble? we'll say to use lab 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_{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

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

layers of abstraction

`x += y` — "Higher-level" language: C

`add %rbx, %rax!` — Assembly: X86-64

60 03SIXTEEN — 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_{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_{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

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$_{\text{SIXTEEN}}$ — 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_{SIXTEEN}	Machine code: Y86
	Hardware Design Language: HCLRS
	Gates / Transistors / Wires / Registers

layers of abstraction

Example	Layer
`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

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

Example	Layer
`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

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_{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
- mostly full register set
- full conditional jumps
- stack-manipulation instructions

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

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

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

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

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

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	
$\mathbf{2^{10}}$	**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	
	...	
$\mathbf{2^{20}}$	1 048 576	**M** (or Mi)
	...	
$\mathbf{2^{30}}$	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

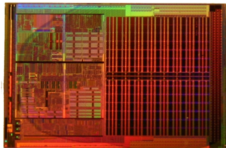
16G $= 16 \cdot 2^{30} = 2^{30+4} = 2^{34}$

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

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

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

processors and memory

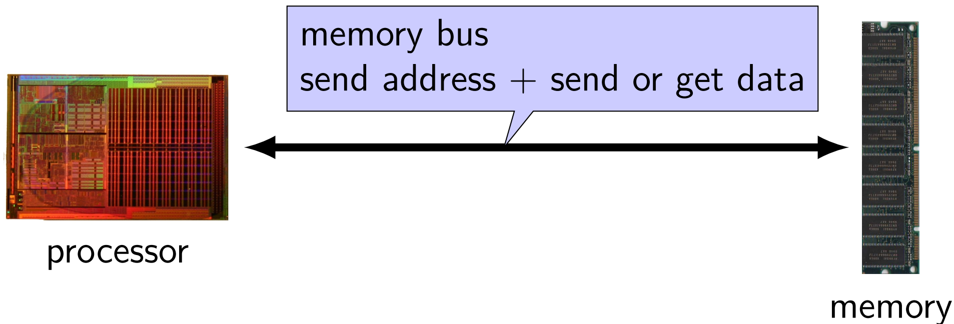


processor

memory

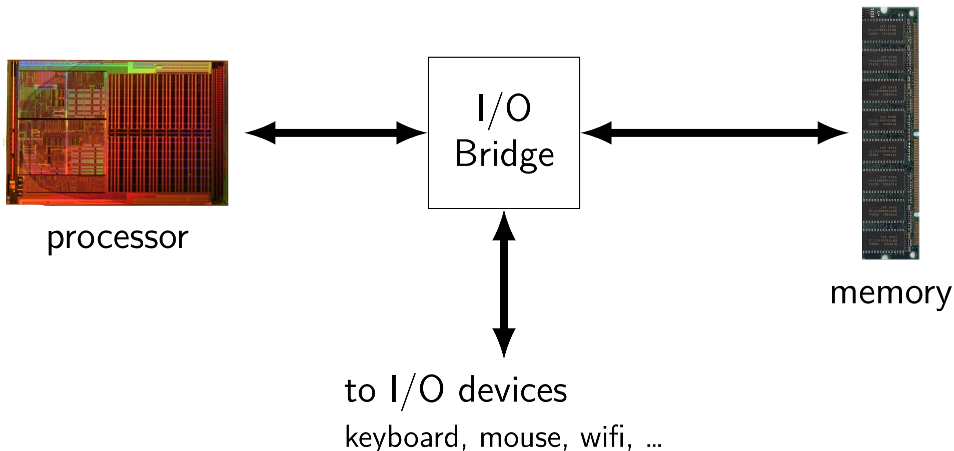
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



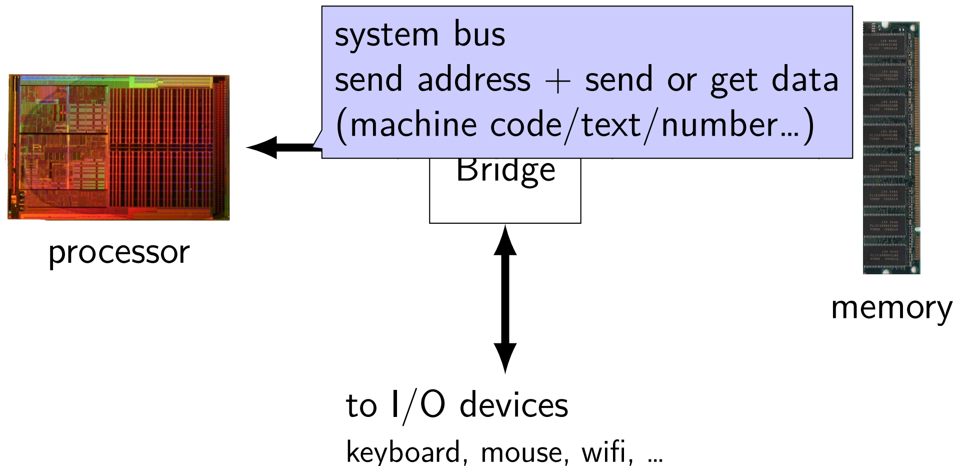
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



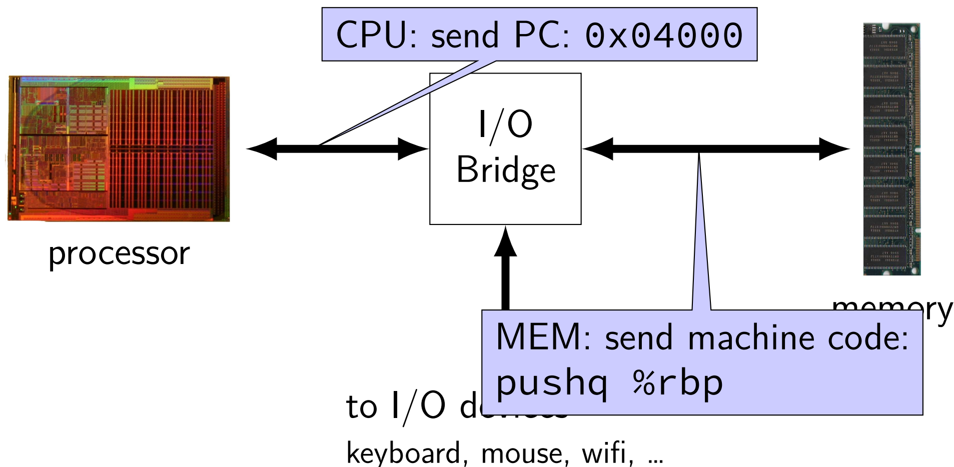
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



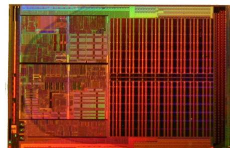
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



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

Bridge

memory

to I/O devices

keyboard, mouse, wifi, …

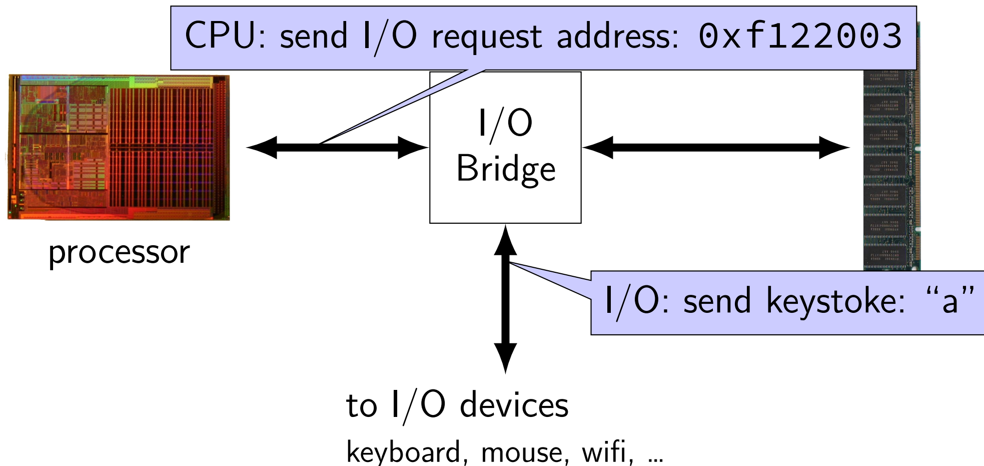
CPU: send PC: `0x04000`

CPU: next PC: `0x04001`

MEM: send machine code: `pushq %rbp`

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



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;
```

0x03020100 = 50462976

little endian
(least significant byte has lowest address)

0x00010203 = 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;
```

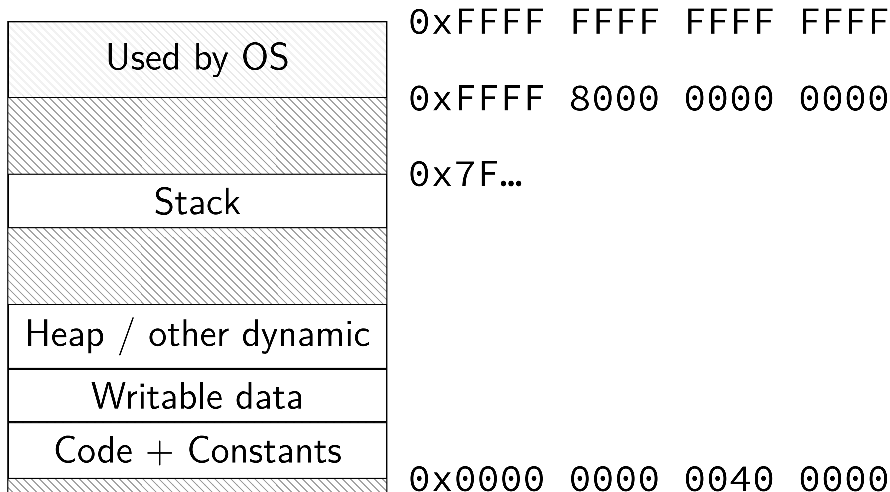
0x03020100 = 50462976

little endian
(least significant byte has lowest address)

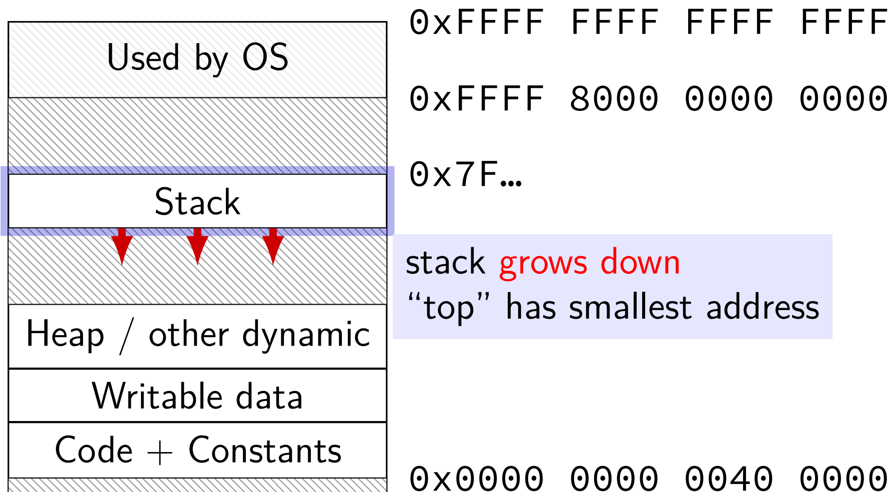
0x00010203 = 66051

big endian
(most significant byte has lowest address)

program memory (x86-64 Linux)

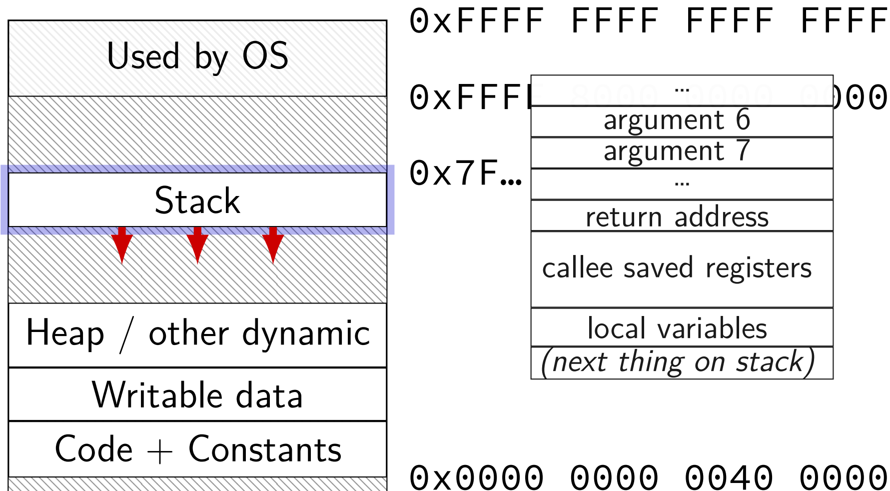


program memory (x86-64 Linux)

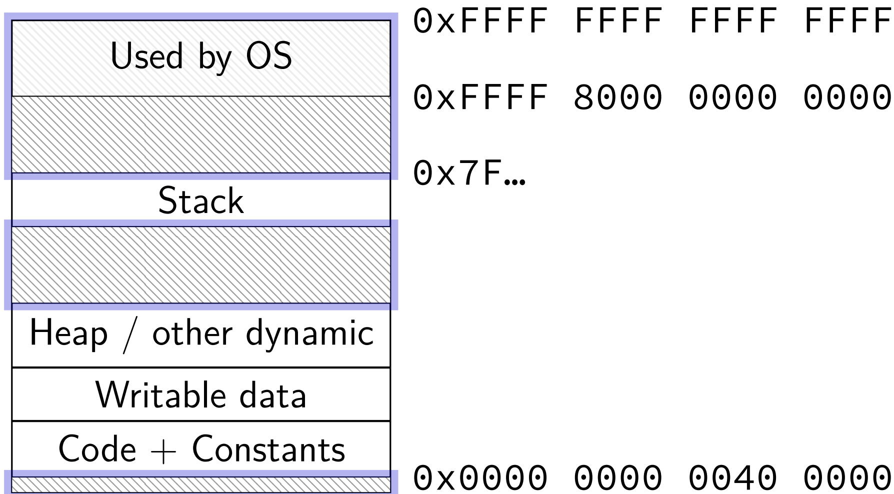


stack grows down
"top" has smallest address

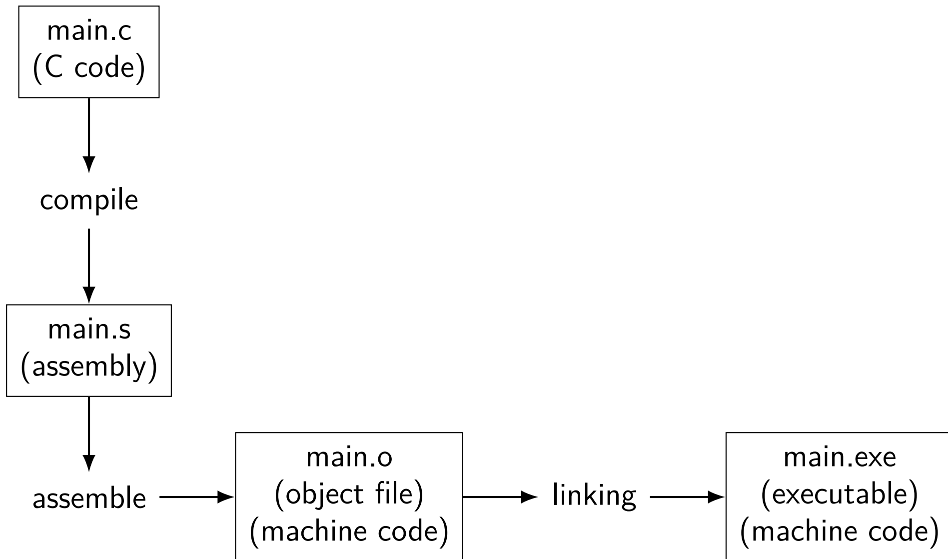
program memory (x86-64 Linux)



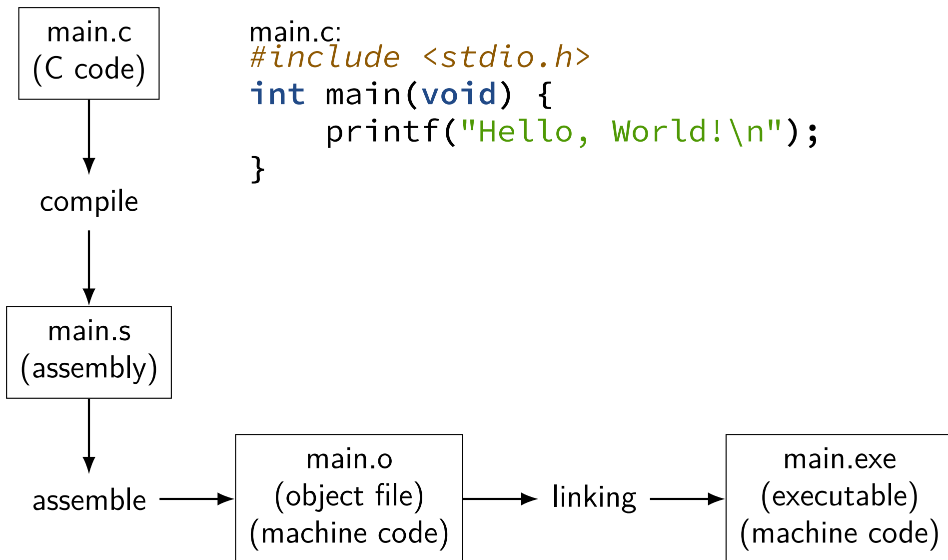
program memory (x86-64 Linux)



compilation pipeline



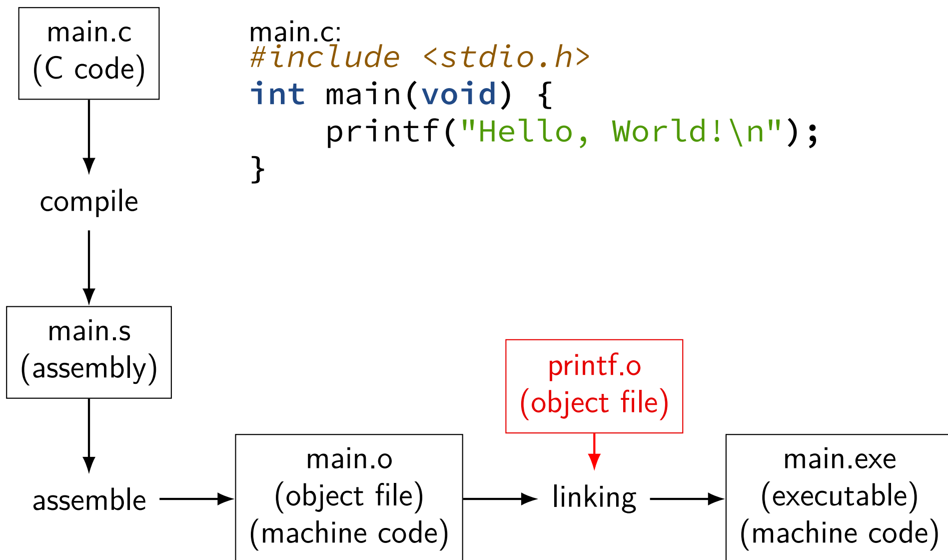
compilation pipeline



main.c:

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

compilation pipeline



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

```
#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.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

```
#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

```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

```
#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

```
#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

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

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
```

lecturers

Samira and I will be splitting lectures

same(ish) lecutre in each section

coursework

quizzes — pre/post week of lecture

  you will need to read

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 — multiple choice/short answer — 2 + final

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, 12:15 PM

post-quiz, on lecture topics — released Thursday evening, due following Saturday, 11:59PM

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

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

first quiz — Thursday

- short — mainly to get you used to it

attendance?

lecture: strongly recommended but not required.

lectures are recorded to help you review

lab: electronic, remote-possible submission, usually. one exception.

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

Quizzes!

- post-quiz after Thursday lecture
- pre-quiz before Tuesday lecture

lab account and/or C environment working

- lab accounts should happen by this weekend

before lab next week

grading

Quizzes: 10% (10% dropped)

Midterms (2): 30%

Final Exam (cumulative): 20%

Homework + Labs: 40%