CS3330 Overview

Changelog

Corrections made in this version not in first posting:

22 Feb 2017: slide 37 (endianness): "has first" to "byte first"; make it clearer which example is which

Layers of Abstraction

`x += y` — "Higher-level" language: C

`add %rbx, %rax` — Assembly: X86-64

60 03_{SIXTEEN} — Machine code: Y86

???

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

???

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 a C compiler

options:

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

assignment compatibility

supported platform: lab machines

many use laptops

- OS X natively, or Linux VM on anything

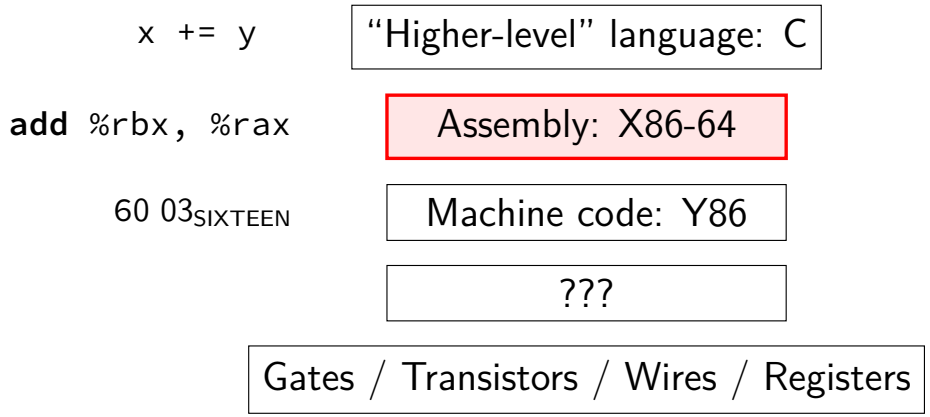
trouble? we'll say to use lab machines

most assignments: C and Unix-like environment

performance assignments: VMs not a good idea

- need precise timing

Layers of Abstraction



X86-64 assembly

in theory, you know this (CS 2150)

in reality, ...

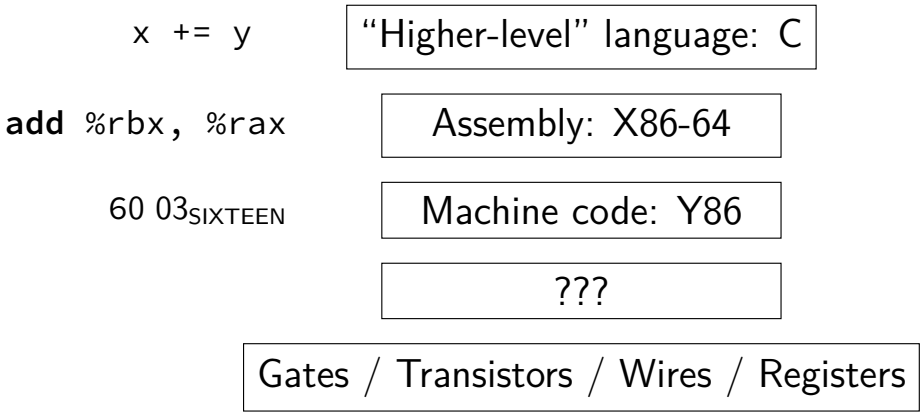
32 versus 64-bit note

some of you may have learned 32-bit in 2150

- (the course has changed)

differences mostly: more, bigger registers

Layers of Abstraction



Y86-64??

Y86: our textbook's X86-64 subset

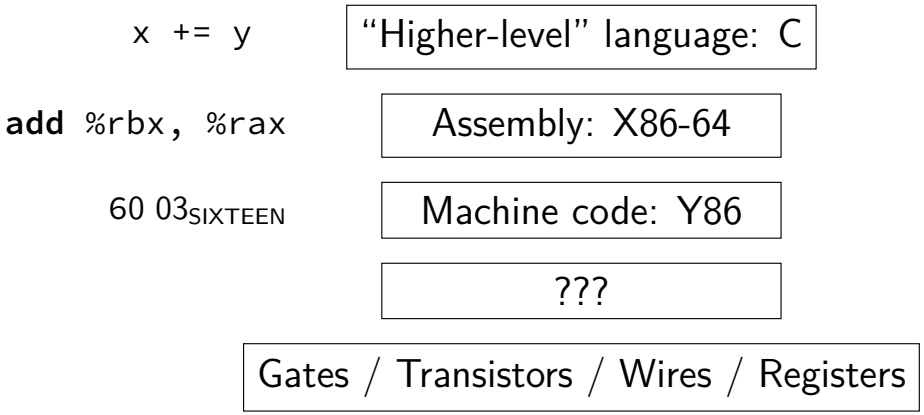
much simpler than real X86-64 encoding

(which we will not cover)

not as simple as 2150's ICBM

- variable-length encoding
- mostly full register set
- full conditional jumps
- stack-manipulation instructions

Layers of Abstraction



Hardware

most of the semester

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: two issues

parallelism

fast hardware is parallel
needs multiple things to do

caching

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

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

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

4M

256T

24

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

24

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

24

Layers of Abstraction

`x += y` — "Higher-level" language: C

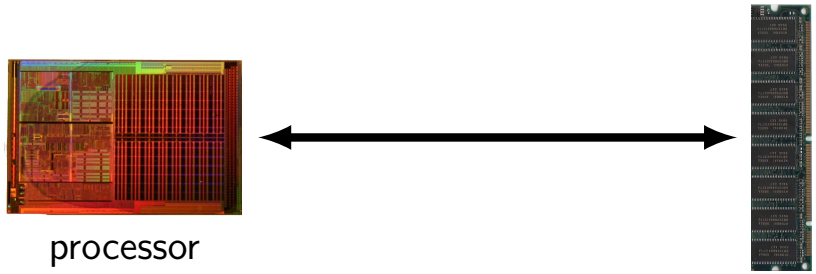
`add %rbx, %rax` — Assembly: X86-64

60 03_{SIXTEEN} — Machine code: Y86

???

Gates / Transistors / Wires / Registers

processors and memory

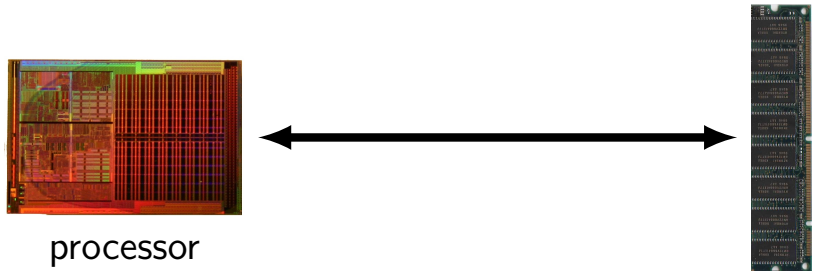


processor

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



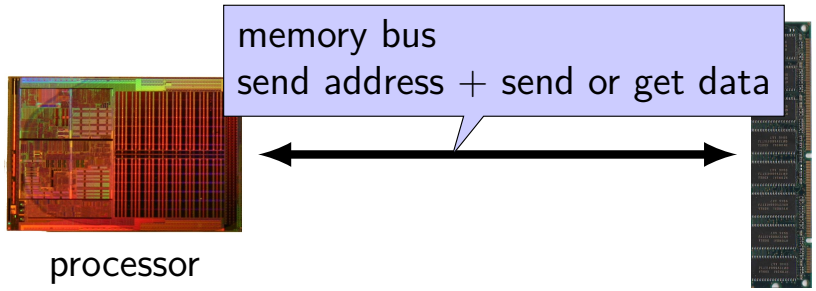
processor

memory

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

26

processors and memory



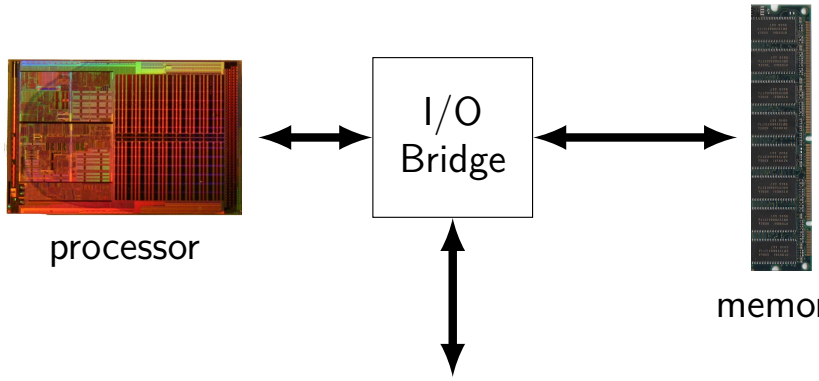
processor

memory

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

26

processors and memory



processor

memory

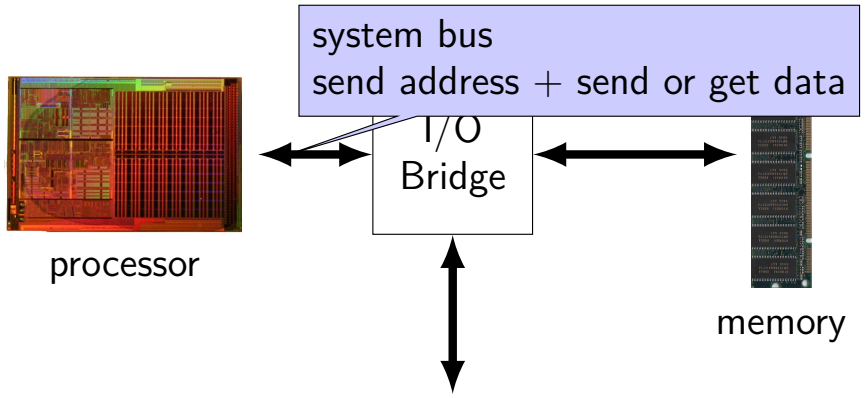
to I/O devices

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

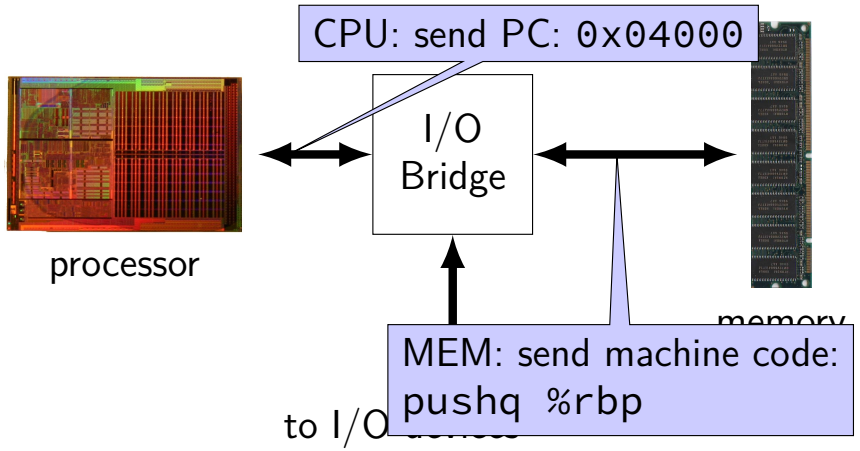
26

processors and memory



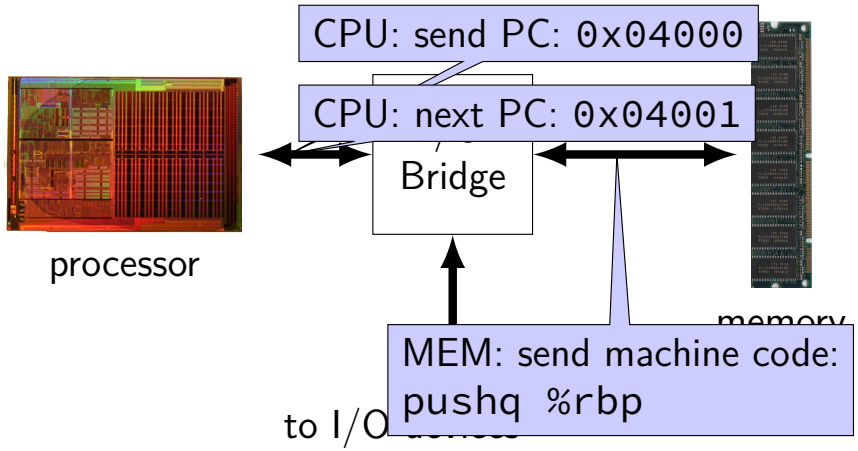
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



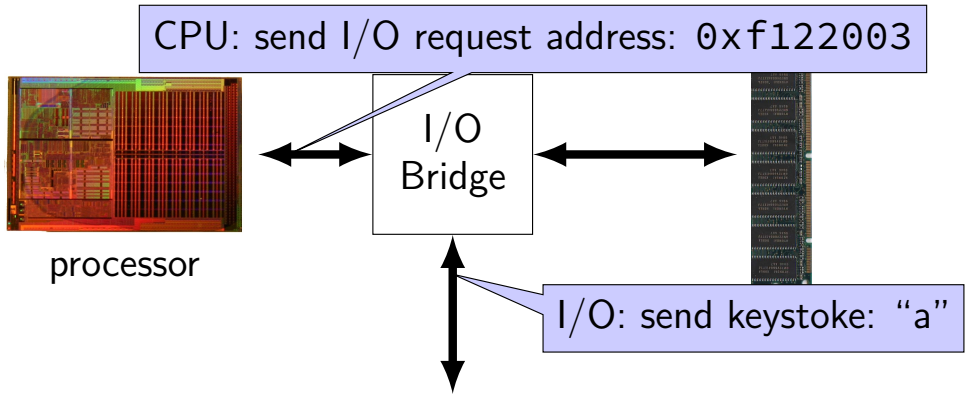
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

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

Layers of Abstraction

`x += y` — "Higher-level" language: C

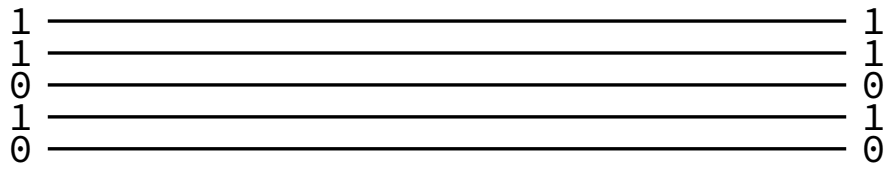
`add %rbx, %rax` — Assembly: X86-64

60 03_{SIXTEEN} — Machine code: Y86

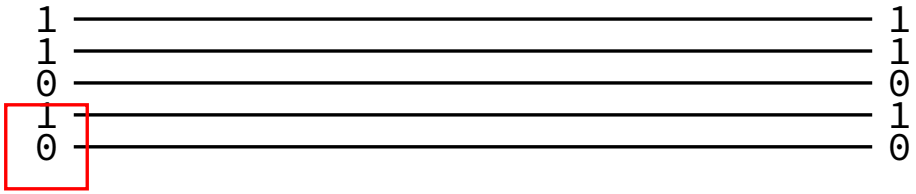
???

Gates / Transistors / Wires / Registers

Circuits: Wires

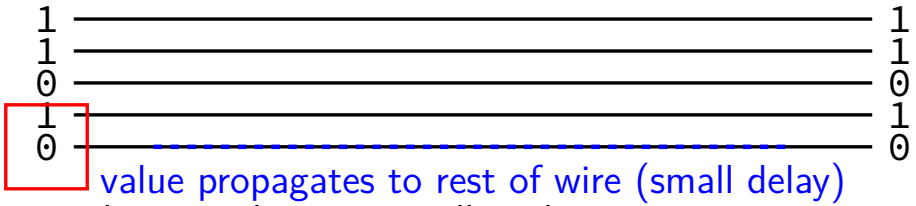


Circuits: Wires



binary value — actually voltage

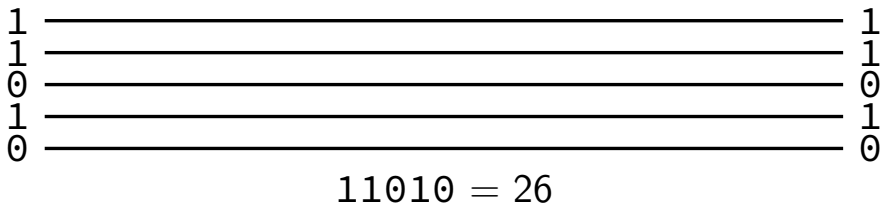
Circuits: Wires



value propagates to rest of wire (small delay)

binary value — actually voltage

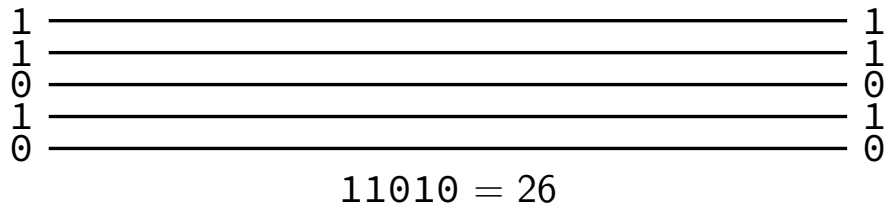
Circuits: Wire Bundles



11010 = 26

Circuits: Wire Bundles

same as

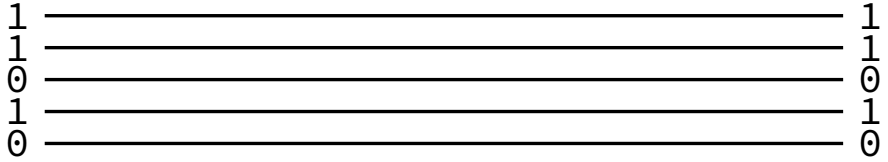


11010 = 26

Circuits: Wire Bundles

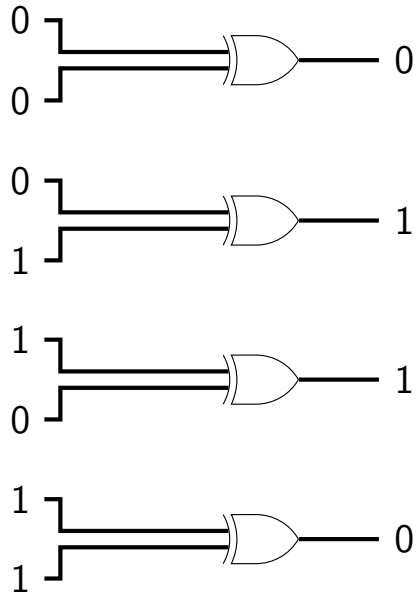
same as

same as



11010 = 26

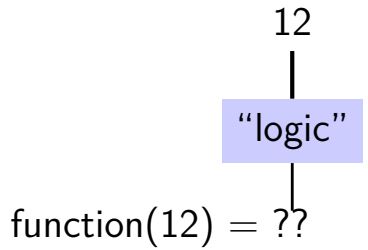
Circuits: Gates



Circuits: Logic

want to do calculations?

generalize gates:

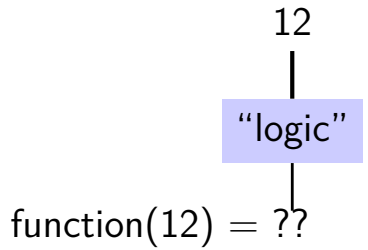


Circuits: Logic

want to do calculations?

generalize gates:

output wires contain result of function on input
changes as input changes (with delay)



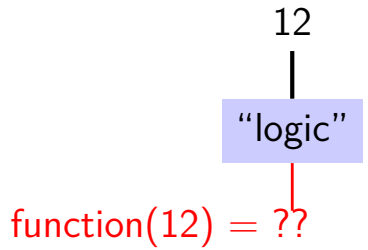
Circuits: Logic

want to do calculations?

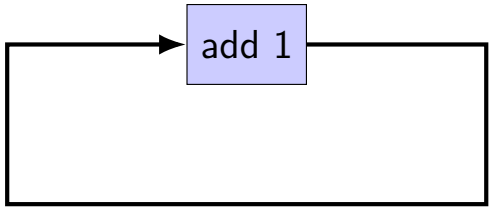
generalize gates:

output wires contain result of function on input
changes as input changes (with delay)

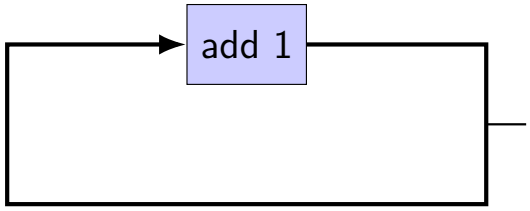
need not be same width as output



example: (broken) counter circuit

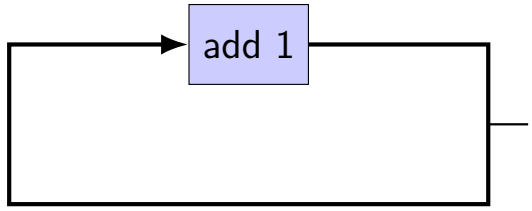


example: (broken) counter circuit



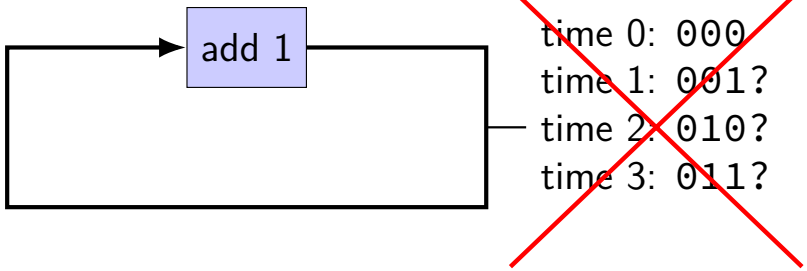
time 0: 000

example: (broken) counter circuit



time 0: 000
time 1: 001?
time 2: 010?
time 3: 011?

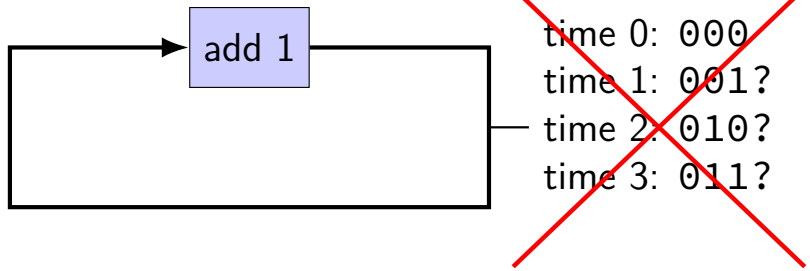
example: (broken) counter circuit



time 0: 000
time 1: 001?
time 2: 010?
time 3: 011?

circuit is not stable

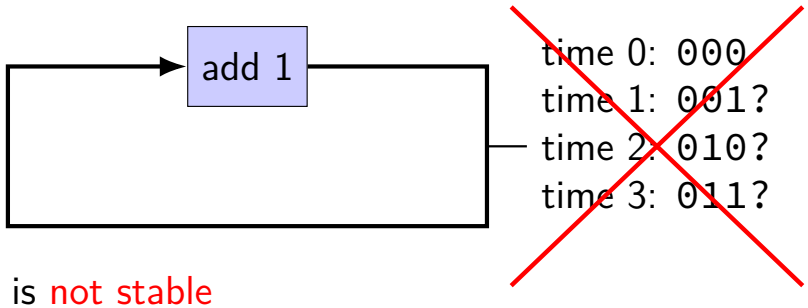
example: (broken) counter circuit



time 0: 000
time 1: 001?
time 2: 010?
time 3: 011?

circuit is not stable
transient values during changes
can't transition from 001 to 010 without 011 or 000

example: (broken) counter circuit



circuit is not stable

transient values during changes

can't transition from 001 to 010 without 011 or 000

halfway voltages — hard to predict behavior

circuits: state

logic performs calculations all the time

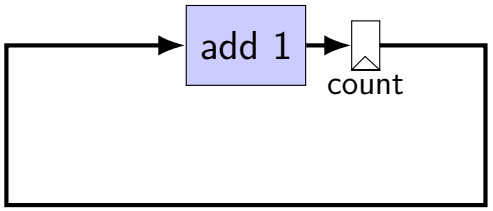
never stores values!

need extra elements to store values

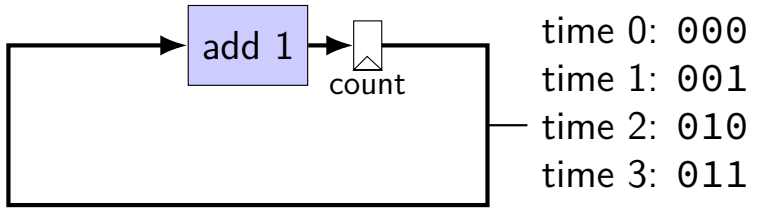
registers, memory

more on these later in the course

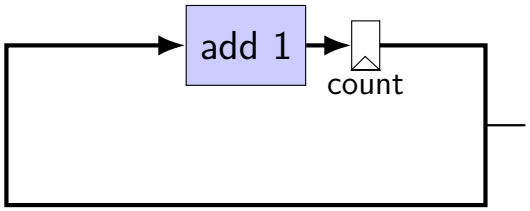
example: counter circuit (corrected)



example: counter circuit (corrected)



example: counter circuit (corrected)



time 0: 000
time 1: 001
time 2: 010
time 3: 011

add register to store current count
updates based on "clock signal" (not shown)
avoids intermediate updates
much more on this later in the semester

Layers of Abstraction

`x += y` — "Higher-level" language: C

`add %rbx, %rax` — Assembly: X86-64

60 03_{SIXTEEN} — Machine code: Y86

???

Gates / Transistors / Wires / Registers

memory

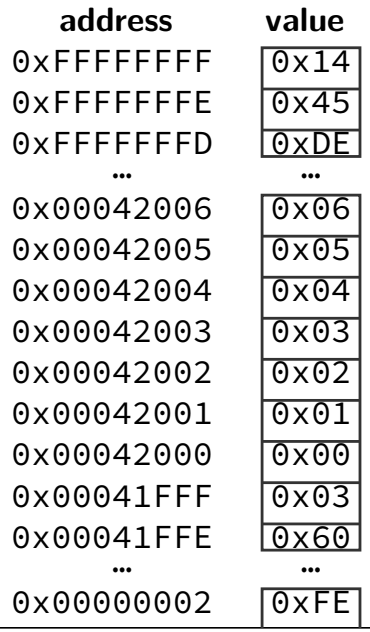
address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	0xDE
...	...
0x00042006	0x06
0x00042005	0x05
0x00042004	0x04
0x00042003	0x03
0x00042002	0x02
0x00042001	0x01
0x00042000	0x00
0x00041FFF	0x03
0x00041FFE	0x60
...	...
0x00000002	0xFE

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

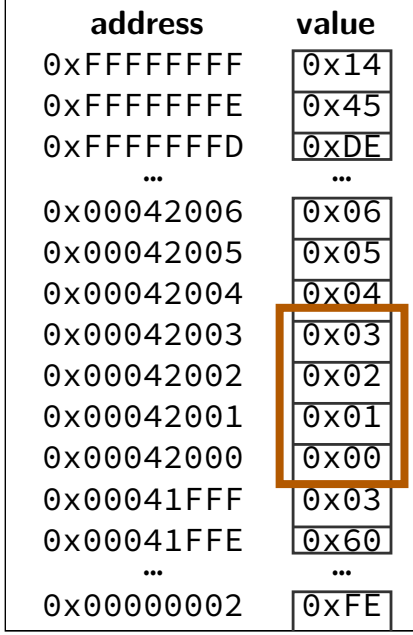
array of bytes (byte = 8 bits)

endianness



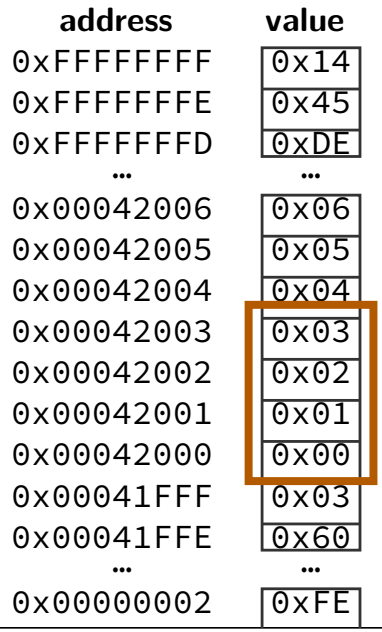

```
int *x = (int*)0x42000;
cout << *x << endl;
```


endianness




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


endianness

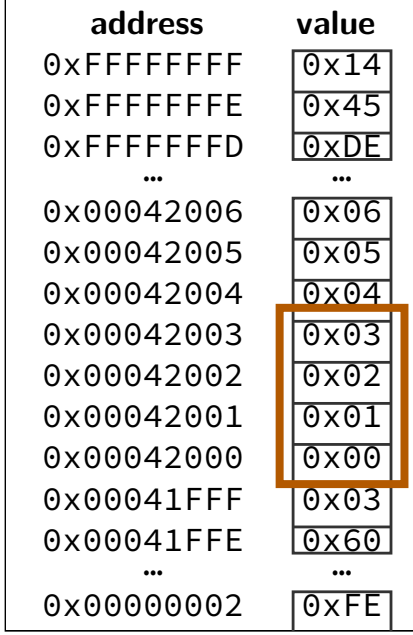



```
int *x = (int*)0x42000;
cout << *x << endl;
```

0x03020100 = 50462976

0x00010203 = 66051

endianness




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

0x03020100 = 50462976

little endian
(least significant byte first)

0x00010203 = 66051

big endian
(most significant byte first)

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

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

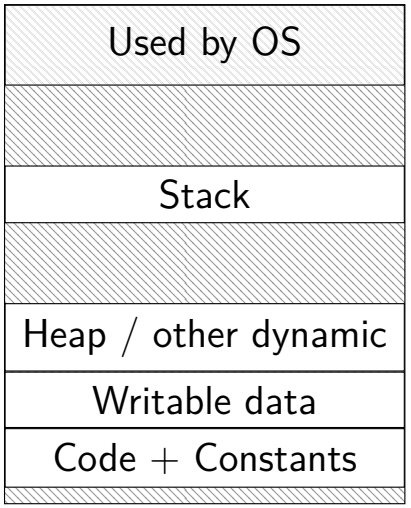
0x03020100 = 50462976

little endian
(least significant byte first)

0x00010203 = 66051

big endian
(most significant byte first)

program memory (x86-64 Linux)



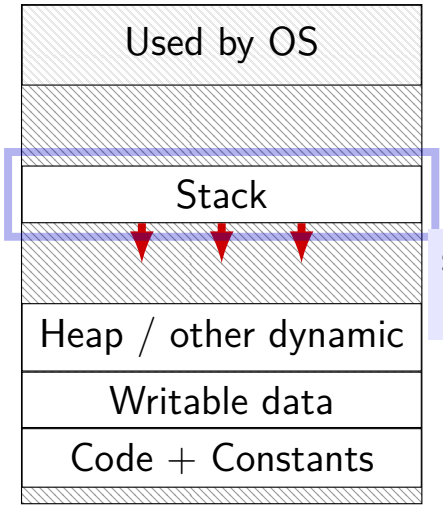
0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

0x0000 0000 0040 0000

program memory (x86-64 Linux)



0xFFFF FFFF FFFF FFFF

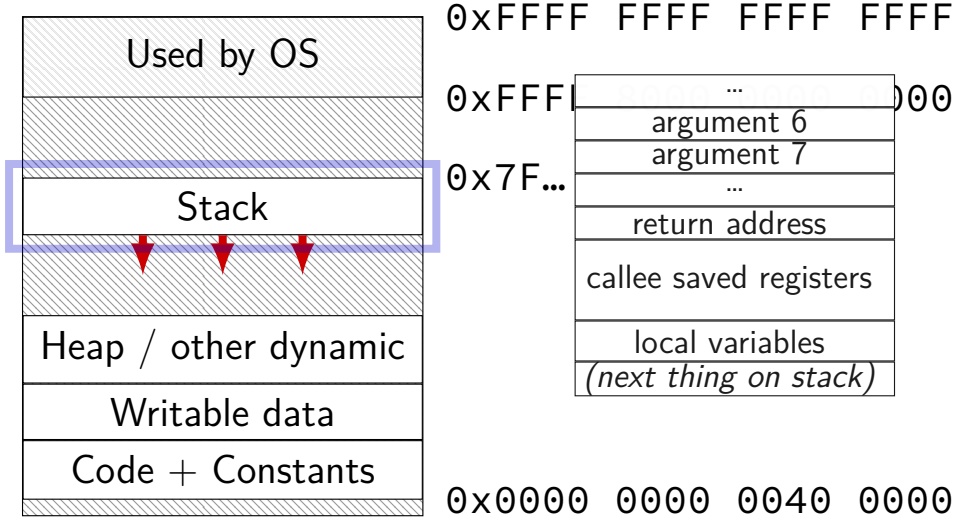
0xFFFF 8000 0000 0000

0x7F…

stack grows down
"top" has smallest address

0x0000 0000 0040 0000

program memory (x86-64 Linux)

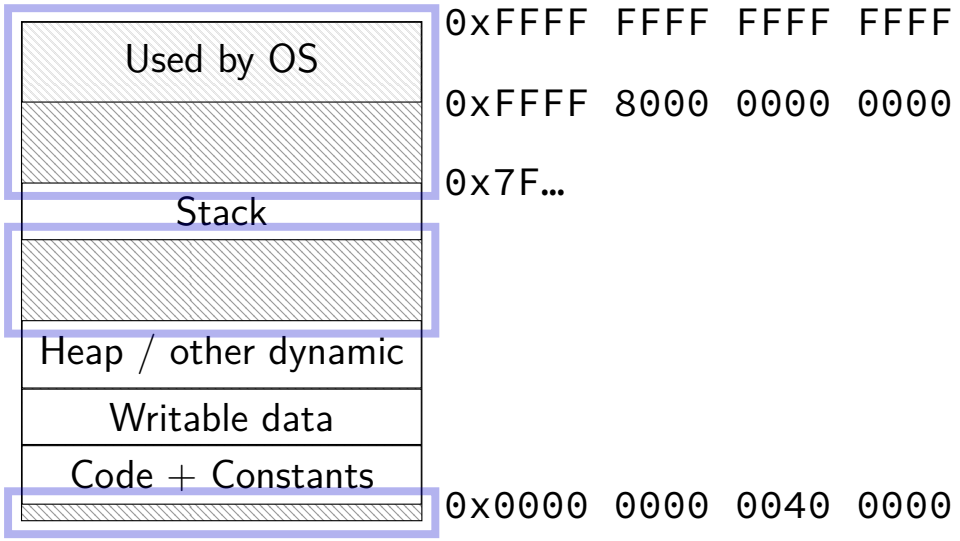


0xFFFF FFFF FFFF FFFF

0x7F…

0x0000 0000 0040 0000

program memory (x86-64 Linux)



parallelism

hardware is parallel by default

much of this class:

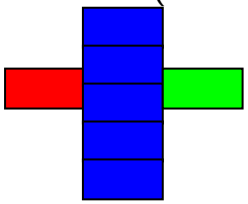
getting the most out of parallelism

parallelism and bottlenecks

Serial:

7 time units

Parallel (blue 5x faster):



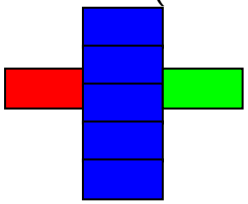
3 time units

parallelism and bottlenecks

Serial:

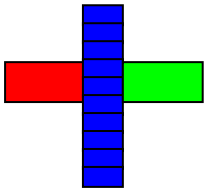
7 time units

Parallel (blue 5x faster):



3 time units

Parallel (blue 10x faster):



2 time units

Amdahl's Law

formula in textbook

benefits of speedup limited by non-sped-up parts

parallelism:

anything not parallelized will be significant

or in math:

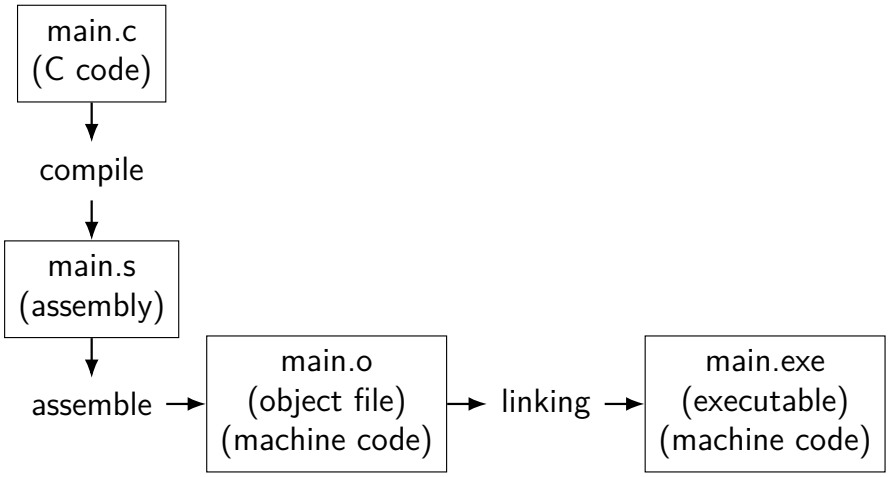
$$\text{time} = \text{serial part} + \text{parallel part} \div \text{parallelism}$$

Not just parallelism

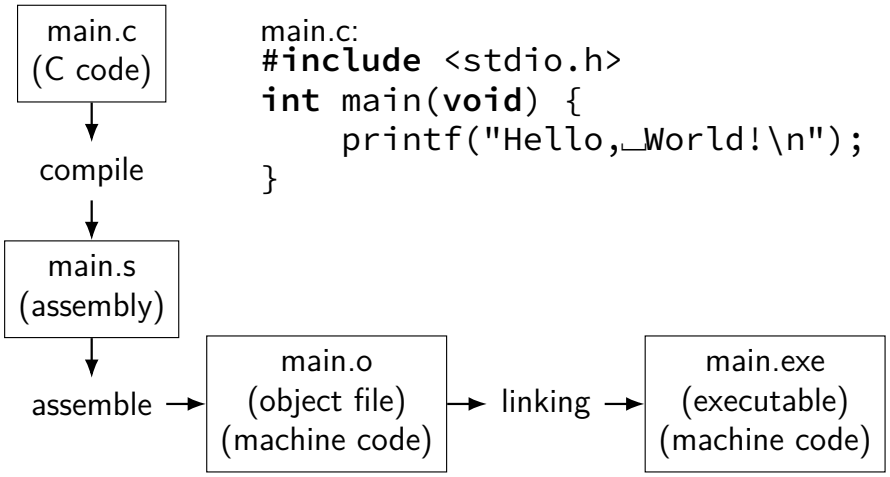
$$\text{time} = \text{serial part} + \text{parallel part} \div \text{parallelism}$$

$$\text{time} = \text{unoptimized part} + \text{optimized part} \div \text{speedup}$$

Preview: Compilation pipeline



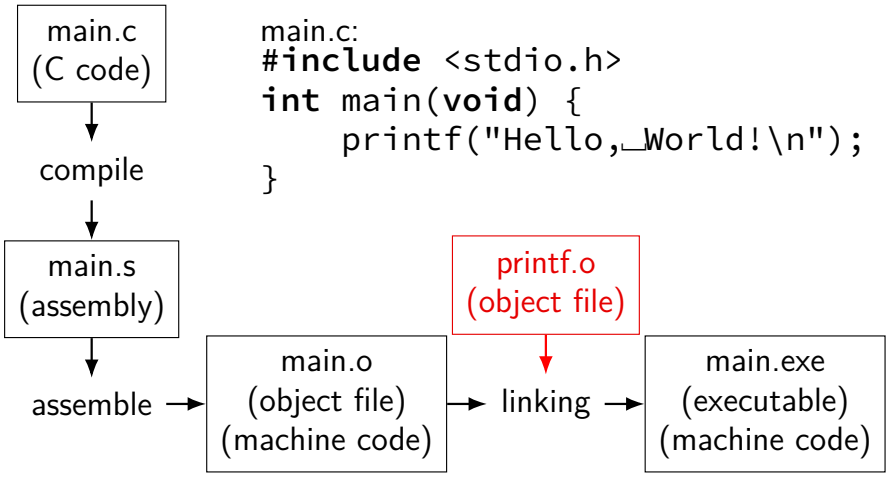
Preview: Compilation pipeline



main.c:

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

Preview: Compilation pipeline



main.c:

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


approximate outline

Weeks 1–2: C, assembly, bit fiddling

Weeks 3–5: Y86 instructions, basic CPU design

Exam 1

Weeks 7–9: pipelined CPUs

Weeks 10: caching

Exam 2

Weeks 11–12: performance programming

Weeks 13–15: exceptions and virtual memory

Final Exam

Coursework

quizzes — pre/post lecture

- you will need to read

labs — mostly graded on effort (did you make reasonable progress?)

homework assignments — introduced by lab (mostly)

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

exams — multiple choice/short answer — 2 + final

Quizzes?

linked off course website (demo)

pre-quiz, on reading – released by Saturday evening, due Tuesday, 10:30 AM

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

lowest 10% will be dropped

first quiz — tonight

- short — mainly to get you used to it

Attendance?

Lecture: 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% up to 48 hours late
- -20% up to one week (168 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 for this lecture
- pre-quiz for next 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%