CS3330 — overview

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

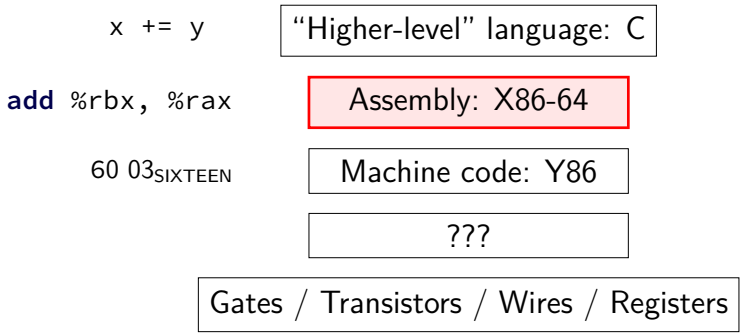
trouble? we'll say to use lab machines

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



X86-64 assembly

in theory, you know this (CS 2150)

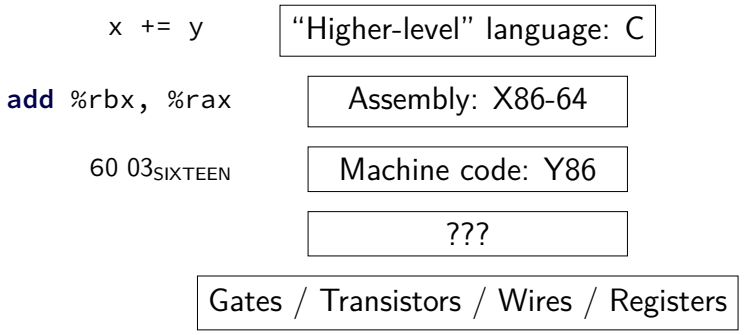
in reality, ...

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



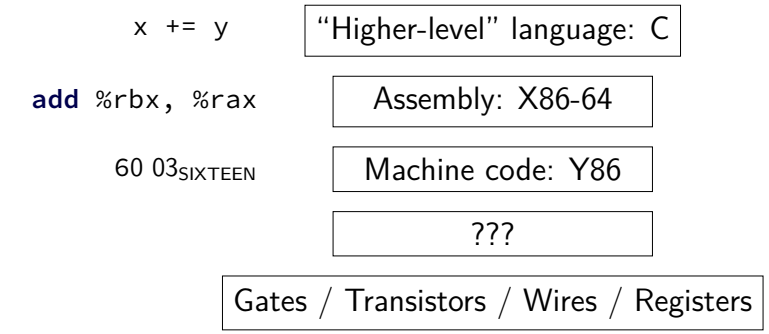
Y86-64??

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

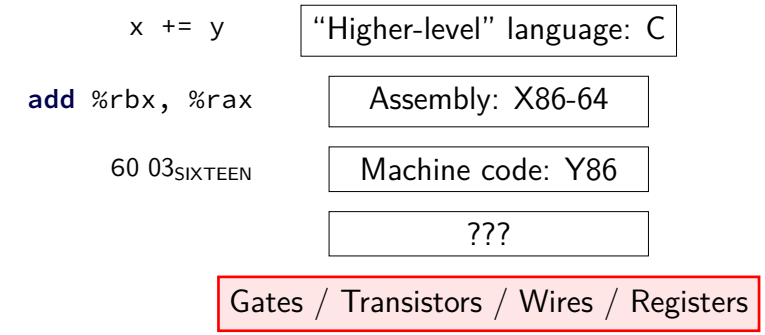
rest of today/tomorrow

brief preview of circuits, CPUs

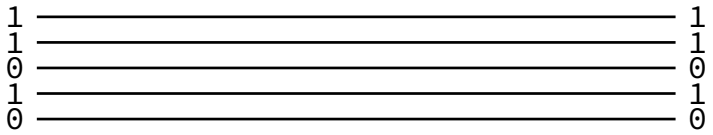
assembly and linking

selected things about C

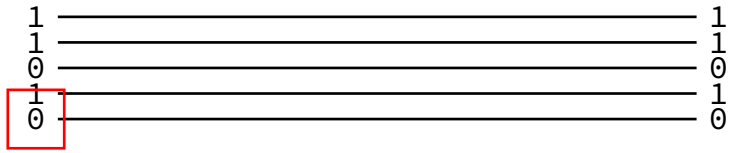
layers of abstraction



Circuits: Wires

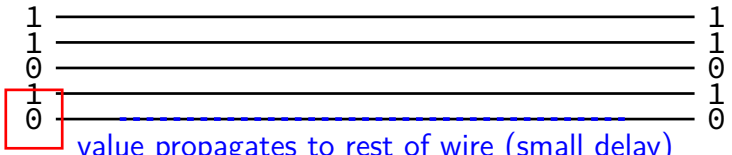


Circuits: Wires



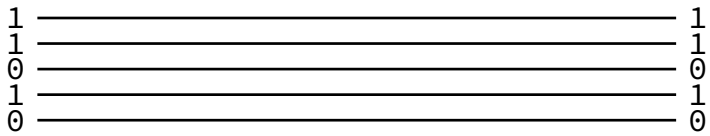
binary value — actually voltage

Circuits: Wires



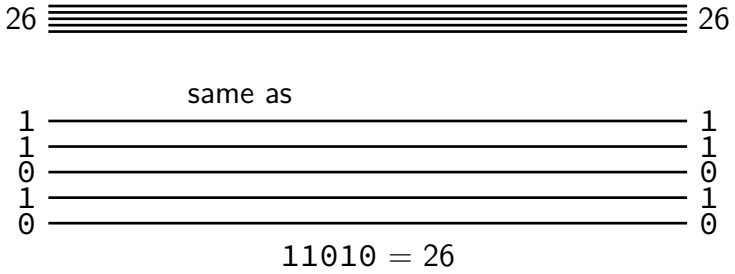
value propagates to rest of wire (small delay)
binary value — actually voltage

Circuits: Wire Bundles

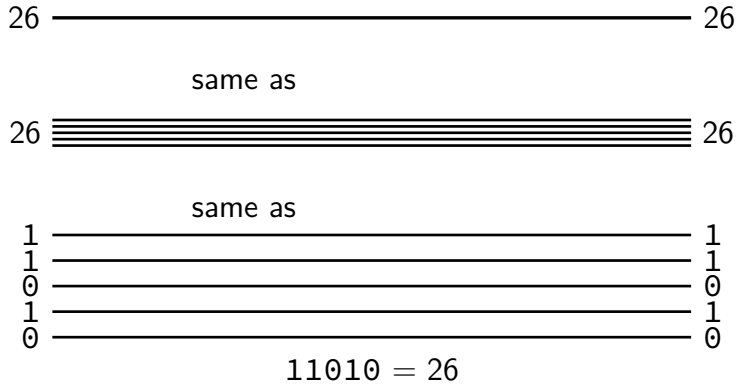


11010 = 26

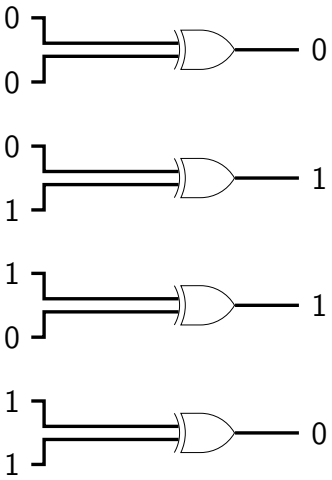
Circuits: Wire Bundles



Circuits: Wire Bundles



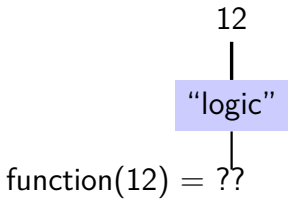
Circuits: Gates



Circuits: Logic

want to do calculations?

generalize gates:

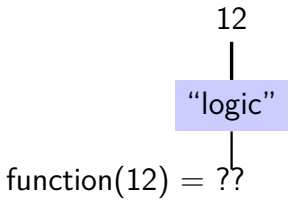


Circuits: Logic

want to do calculations?

generalize gates:

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



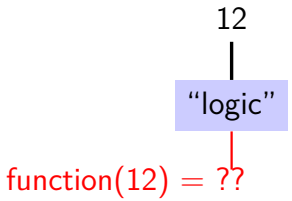
Circuits: Logic

want to do calculations?

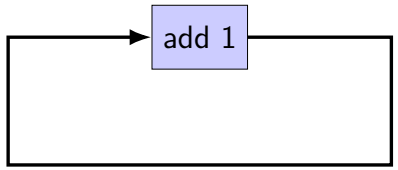
generalize gates:

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

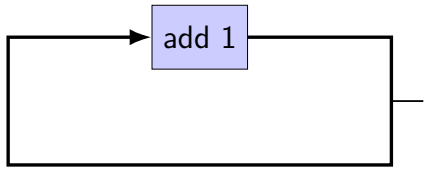
need not be same width as output



example: (broken) counter circuit

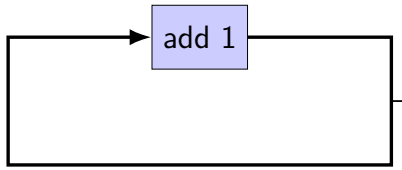


example: (broken) counter circuit



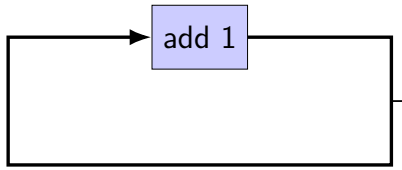
time 0: `000`

example: (broken) counter circuit



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

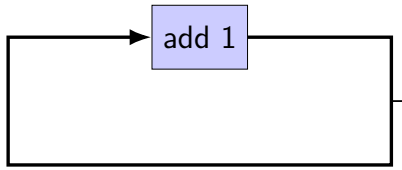
example: (broken) counter circuit



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

circuit is not stable

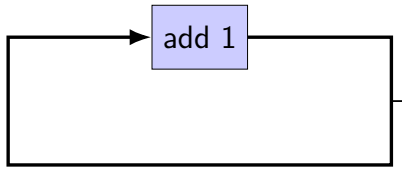
example: (broken) counter circuit



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



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
halfway voltages — hard to predict behavior

circuits: state

logic performs calculations all the time

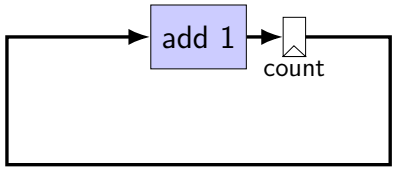
never stores values!

need extra elements to store values

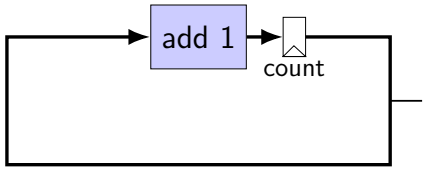
registers, memory

more on these later in the course

example: counter circuit (corrected)

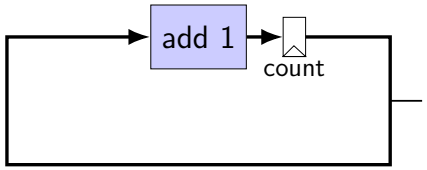


example: counter circuit (corrected)



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

example: counter circuit (corrected)

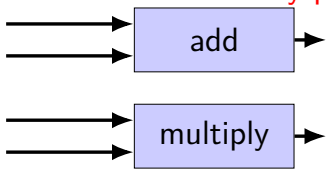


time 0: `000`
time 1: `001`
time 2: `010`
time 3: `011`

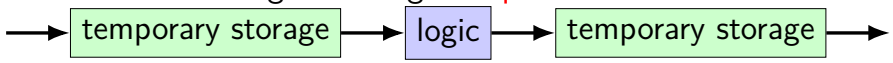
add register to store current count
updates based on "clock signal" (not shown)
avoids intermediate updates
much more on this later in the semester

parallel hardware

hardware is inherently parallel



most hardware design: making it sequential

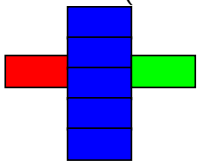


parallelism and bottlenecks

Serial:

7 time units

Parallel (blue 5x faster):



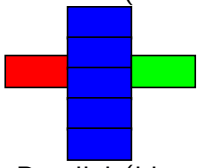
3 time units

parallelism and bottlenecks

Serial:

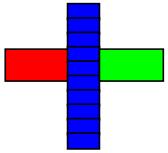
7 time units

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:

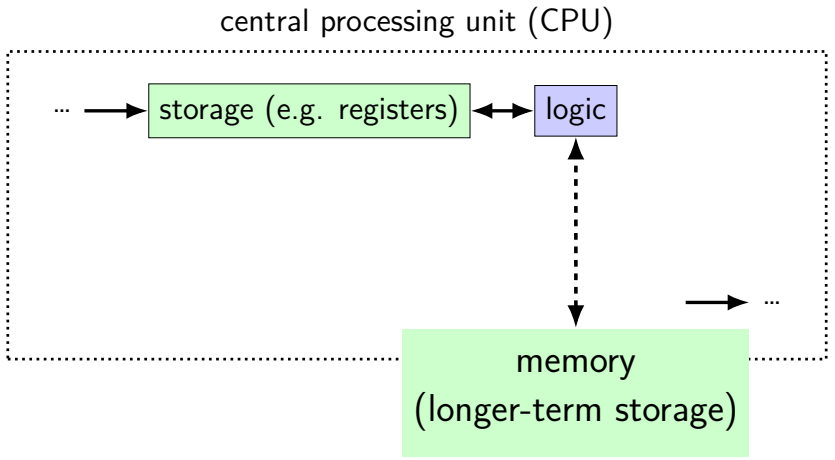
$$\text{time} = \text{serial part} + \text{parallel part} \div \text{parallelism}$$

not just parallelism

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

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

constructing a computer



layers of abstraction

`x += y` — "Higher-level" language: C

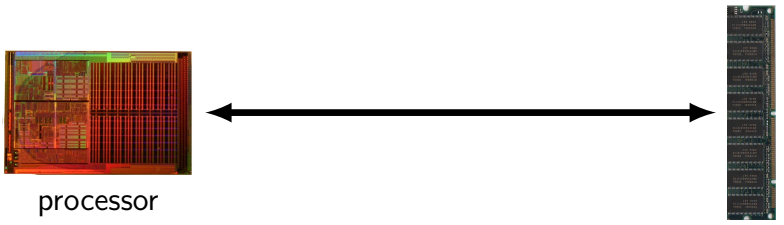
`add %rbx, %rax` — Assembly: X86-64

60 03_{SIXTEEN} — Machine code: Y86

???

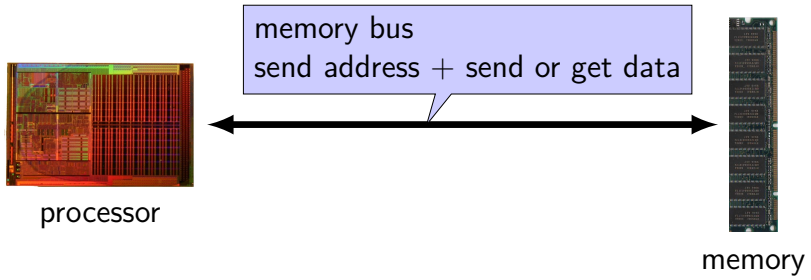
Gates / Transistors / Wires / Registers

processors and memory



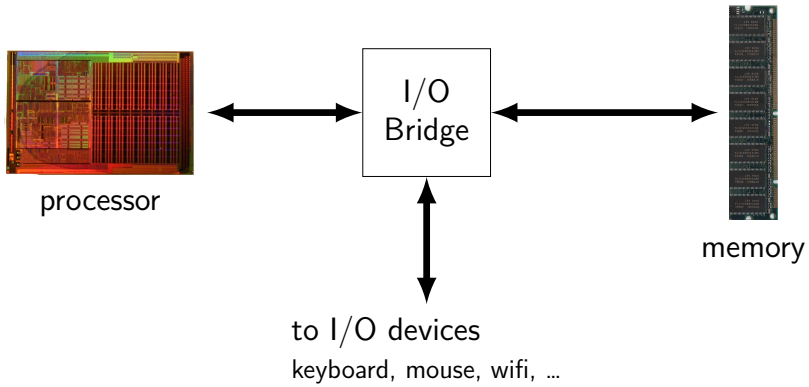
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



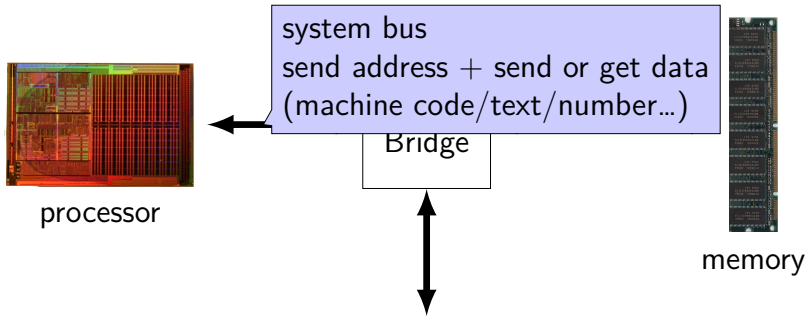
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



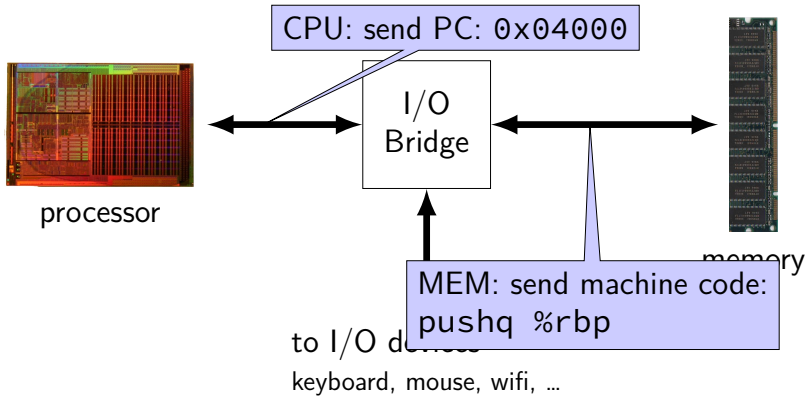
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



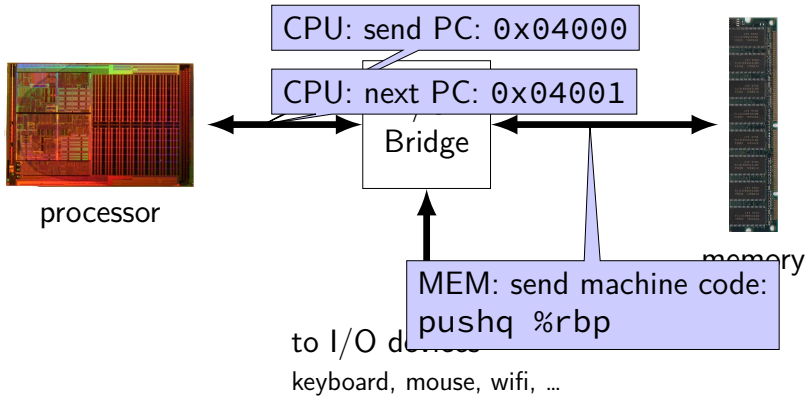
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



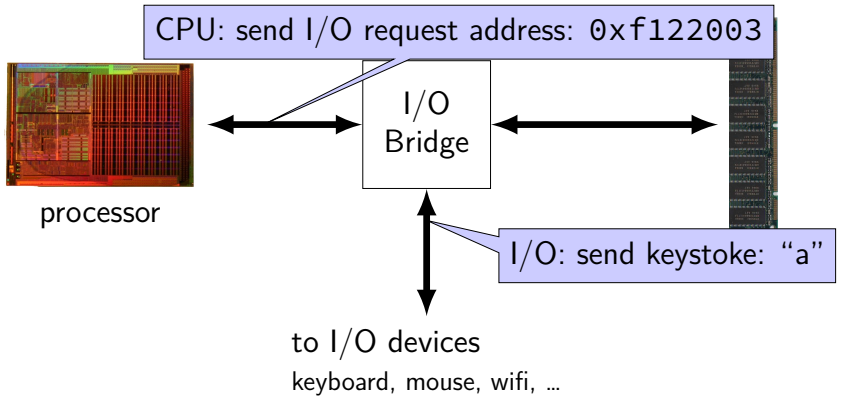
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



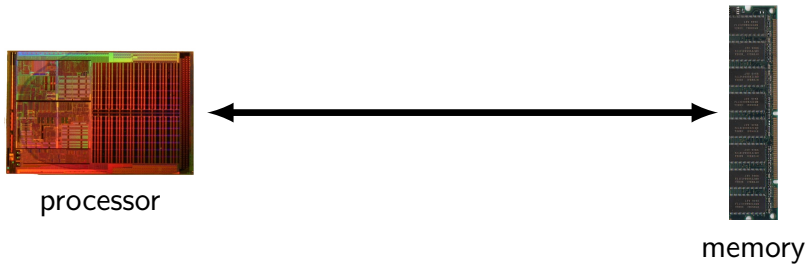
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

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

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)