CS 3330 introduction

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-like environment with a C compiler

will have department accounts, hopefully by end of week

SSH to portal.cs.virginia.edu – remote terminal

NX — remote desktop to a department Linux machine

instructions off course website (Collab)

also some other options

homework: C environment

officially supported: department machines (SSH [terminal] or NX [remote desktop])

some other options (for *most* assignments):

- Linux (native or VM)
  - 2150 VM image should work
- most assignments can Windows Subsystem for Linux natively
- most assignments can use OS X natively
  - notable exception: next week's lab+homework

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

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, ...

x86-64 assembly translation?

```
int x, y, z;
int get_sum() {
    return x + y + z;
}
```

equivalent assembly:

A.
```
// Intel syntax
get_sum:
    mov RAX, [x]
    add RAX, [RAX+y]
    add RAX, [RAX+z]
    ret
```

```
// AT&T syntax
get_sum:
    mov x, %rax
    add y(%rax), %rax
    add z(%rax), %rax
    ret
```

B.
```
// Intel syntax
get_sum:
    mov RAX, [x]
    add RAX, [y]
    add RAX, [z]
    ret
```

```
// AT&T syntax
get_sum:
    mov x, %rax
    add y, %rax
    add z, %rax
    ret
```

C. both A and B

D. neither A nor B

explanation

`mov RAX, [x]` / `mov x, %rax`

RAX ← memory[address of x]

`add RAX, [RAX+y]` / `add y(%rax), %rax`

RAX ← RAX + memory[RAX + address of y]
(if y is an array of long, similar effect to RAX ← y[RAX/sizeof(long)])

`add RAX, [y]` / `add y, %rax`

RAX ← RAX + memory[address of y]

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

Y86-64??

Y86: our textbook's X86-64 subset

- hope: leverage 2150 assembly knowledge

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

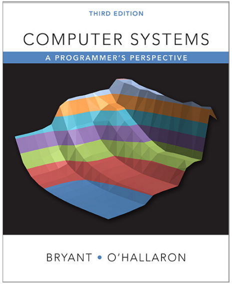
`x += y`	"Higher-level" language: C
`add %rbx, %rax`	Assembly: X86-64
$60\ 03_{\text{SIXTEEN}}$	Machine code: Y86
	Hardware Design Language: HCLRS
	Gates / Transistors / Wires / Registers

textbook

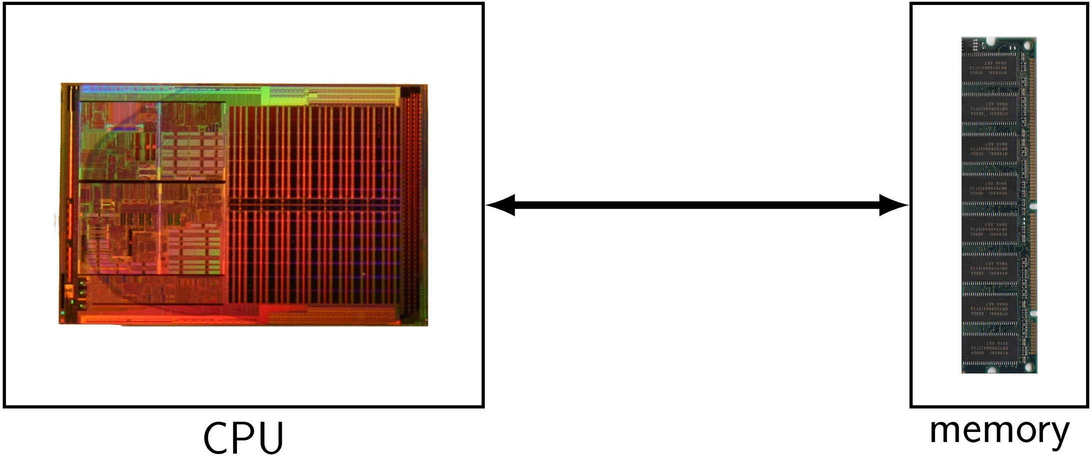
Computer Systems: A Programmer's Perspective

HCL assignments follow pretty closely

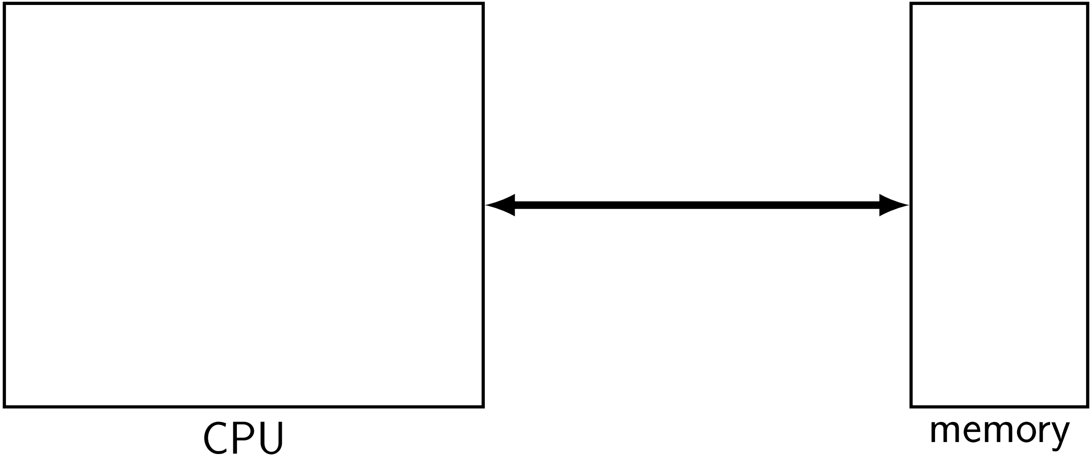
(useful, but less important for other topics)



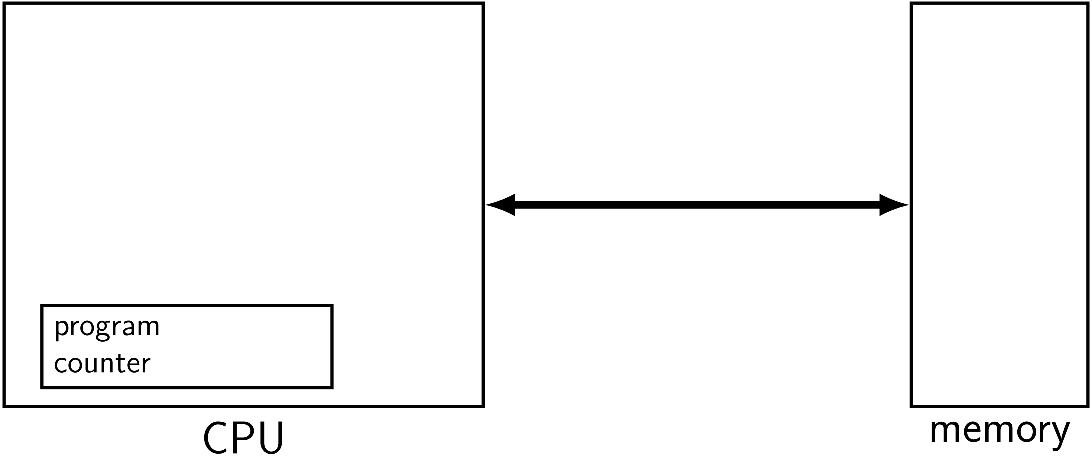
processors and memory (physically)



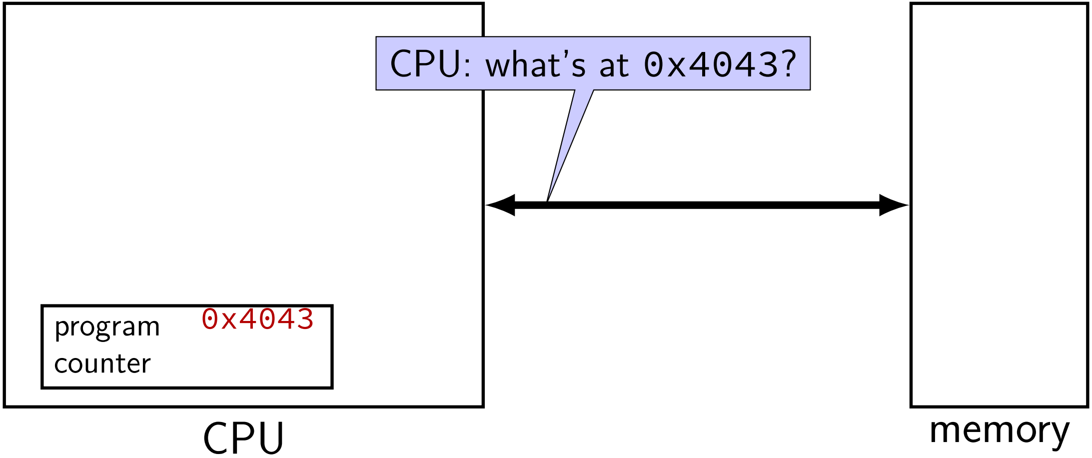
processors and memory (connection)



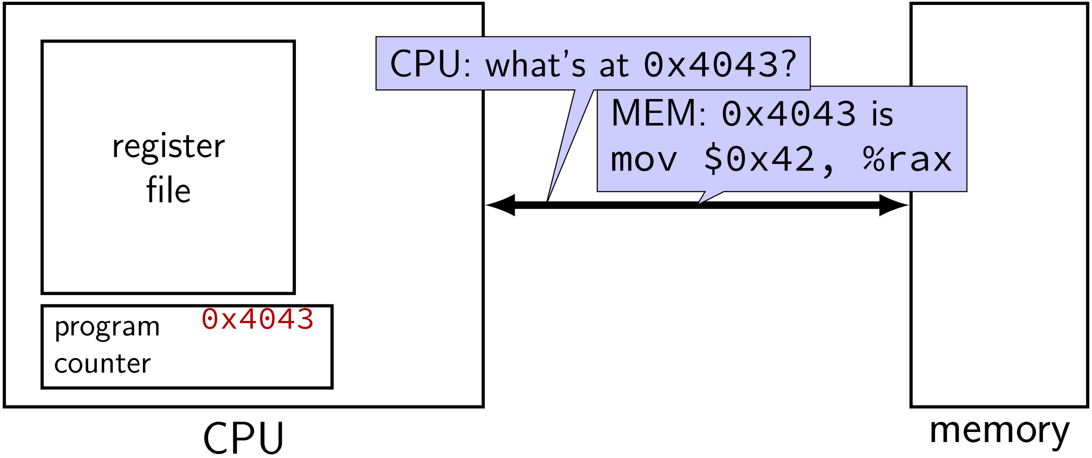
processors and memory (connection)



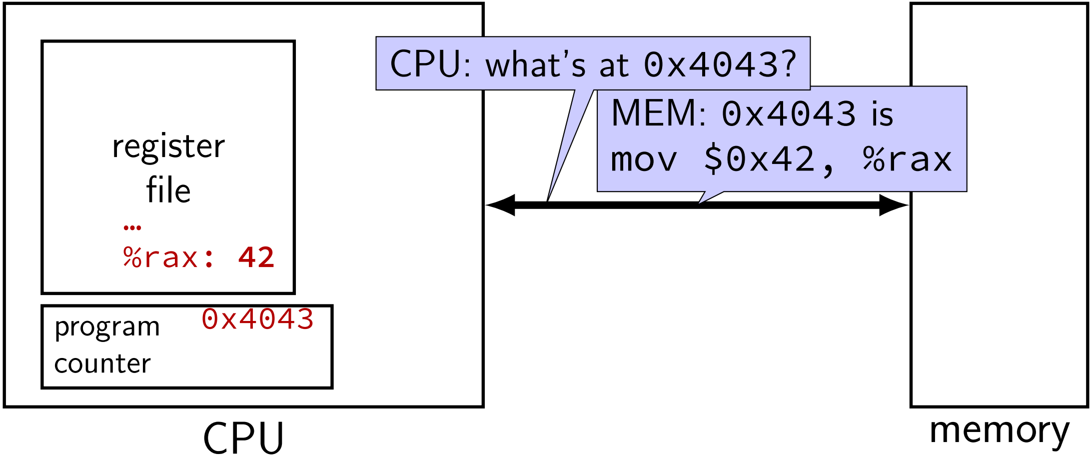
processors and memory (connection)



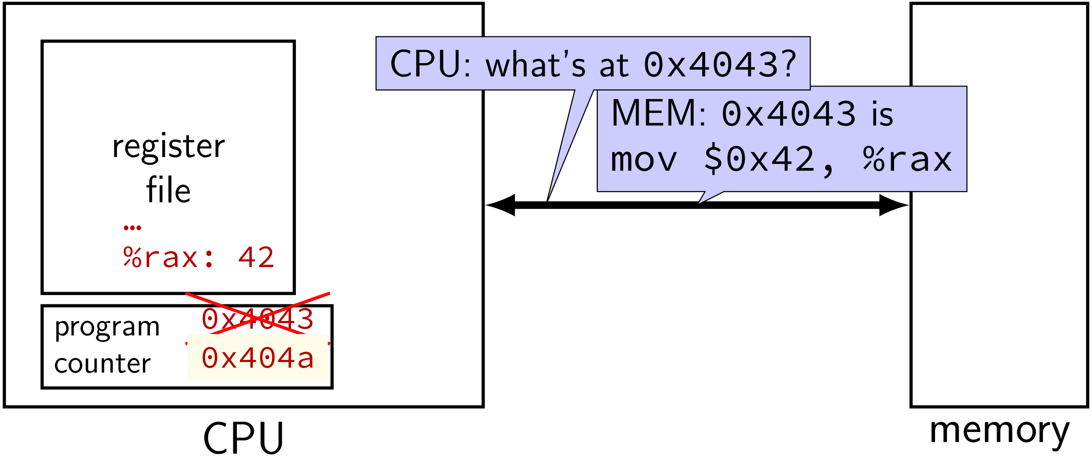
processors and memory (connection)



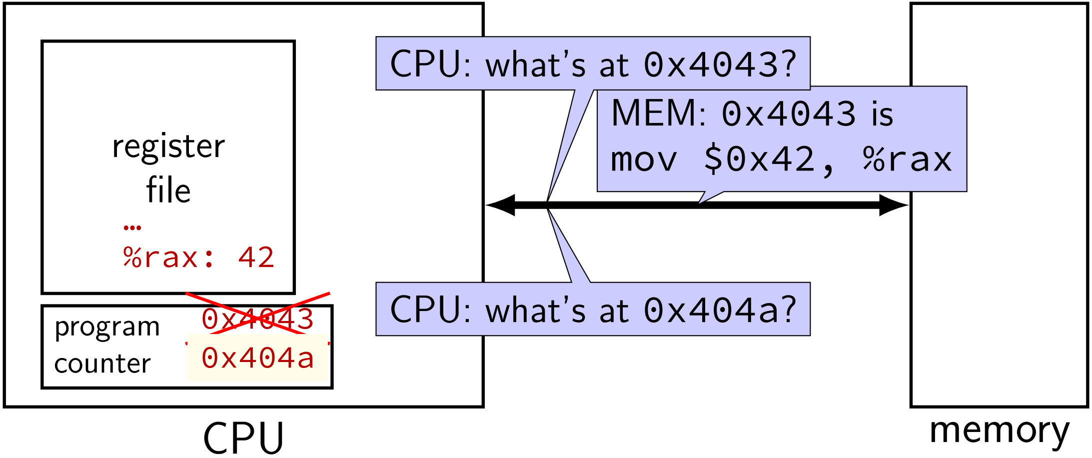
processors and memory (connection)



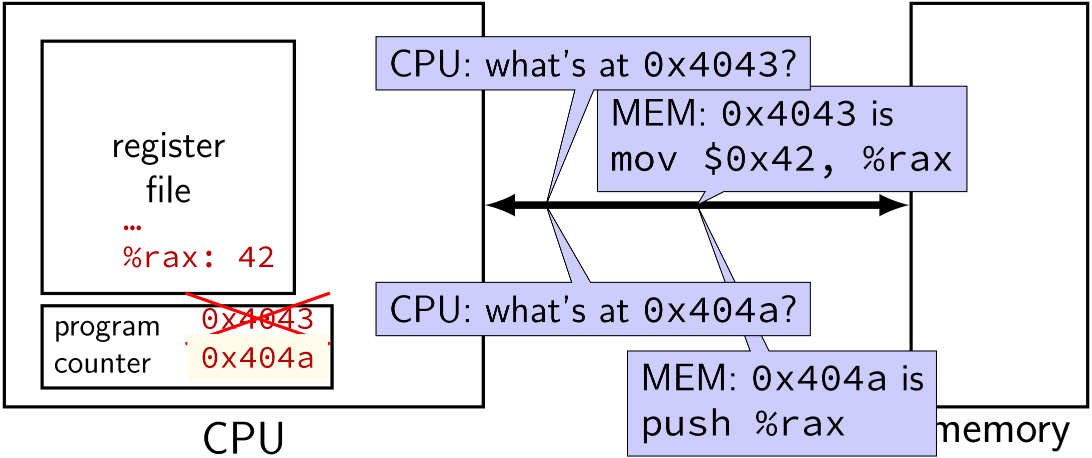
processors and memory (connection)



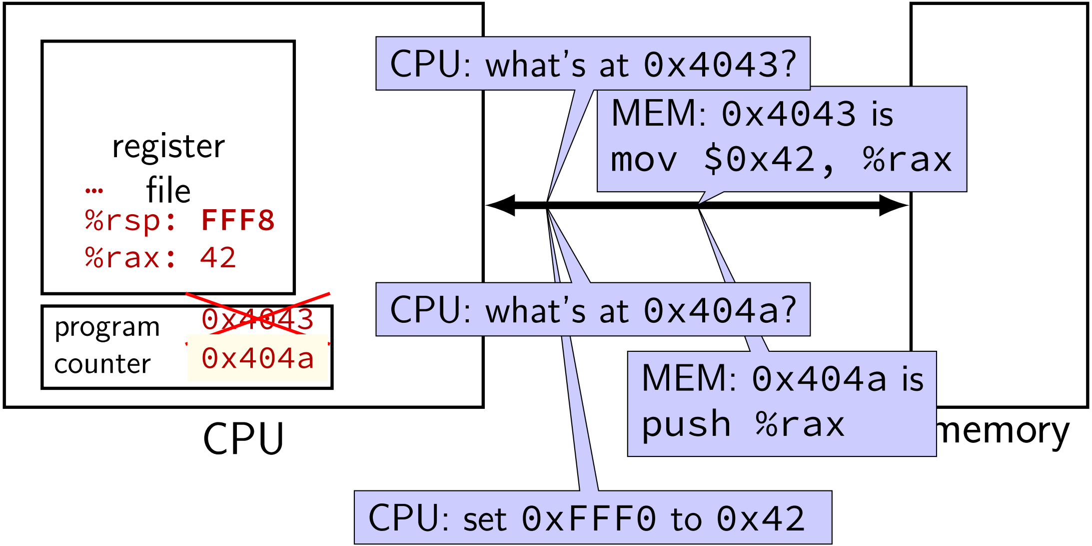
processors and memory (connection)



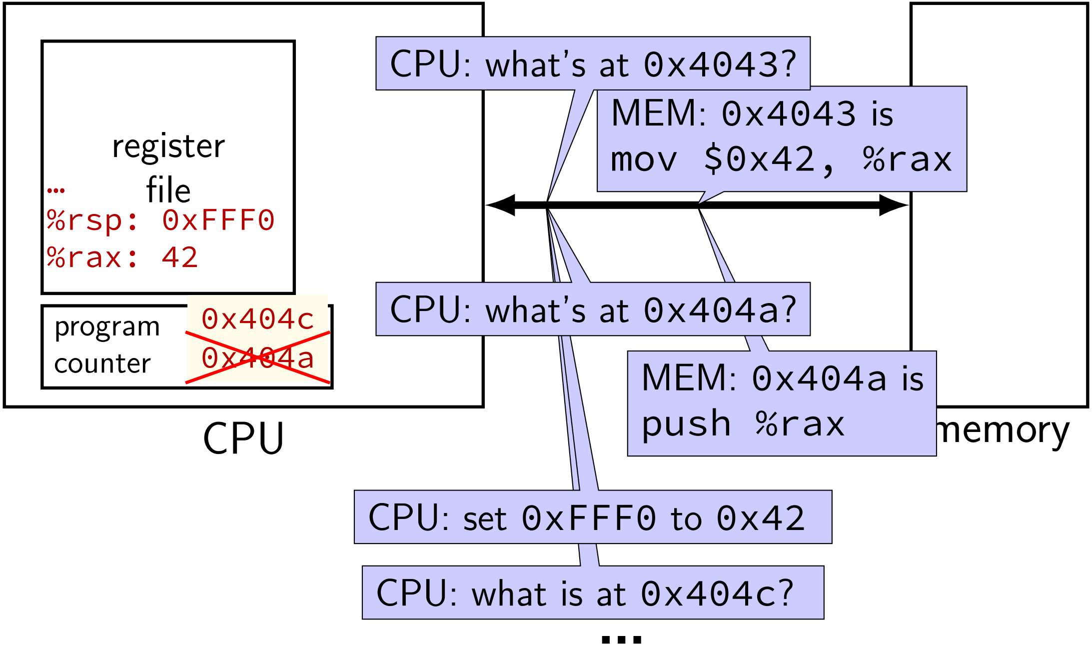
processors and memory (connection)



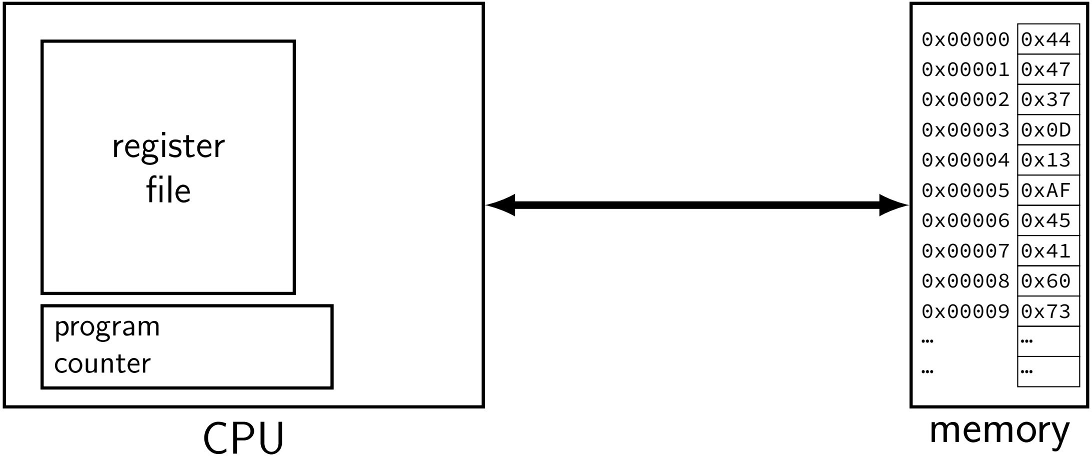
processors and memory (connection)



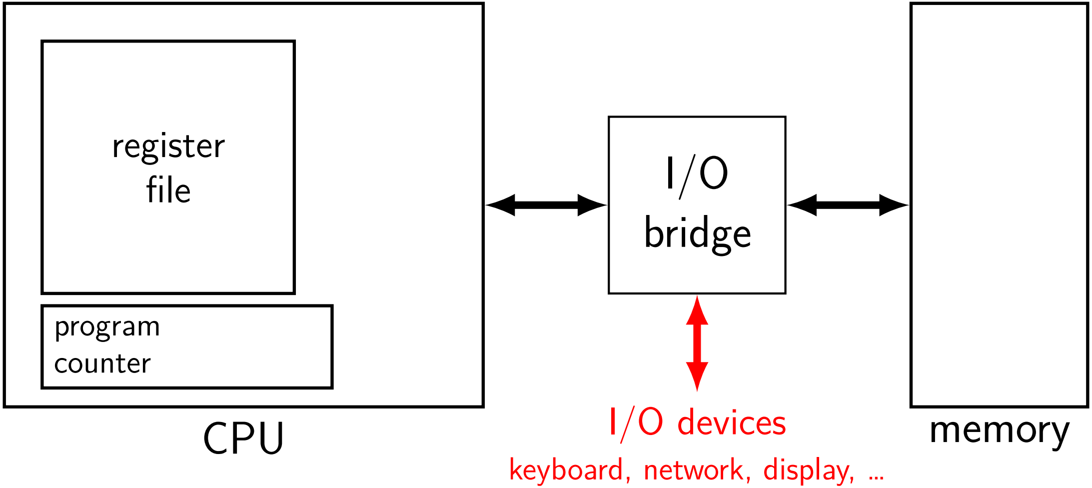
processors and memory (connection)



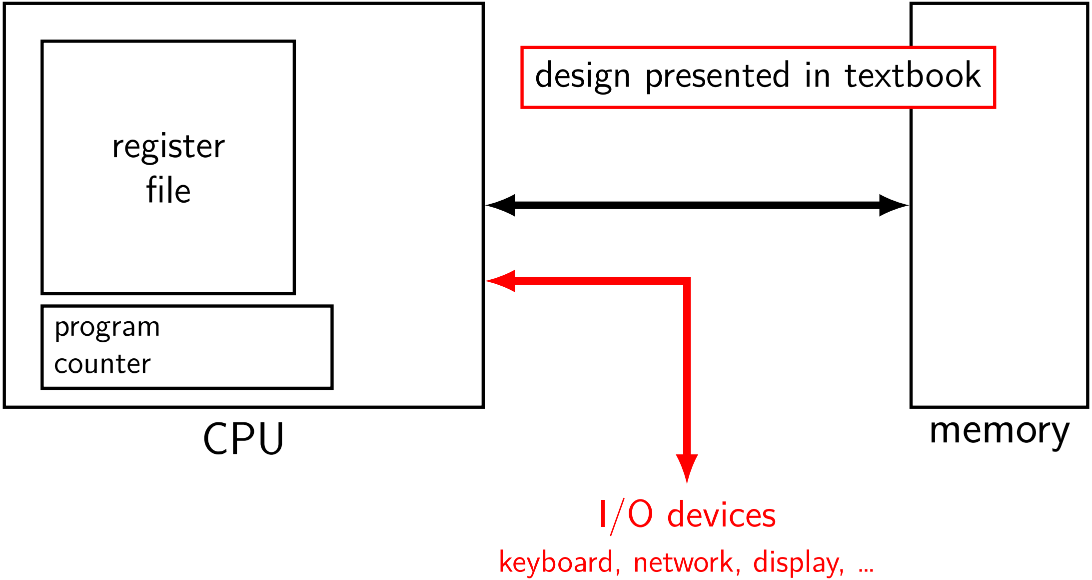
processors and memory (memory really?)



processors and memory and I/O



processors and memory and I/O [alternate]



exercise

suppose a processor is executing the following instruction

`movq 0x123400, %rax` (AT&T syntax)

`MOV RAX, [0x123400]` (Intel syntax)

which moves the value at memory location `0x123400` to %rax

in the processor + memory bus model, how many times is a message sent from the processor to the memory?

exercise

suppose a processor is executing the following instruction

`movq 0x123400, %rax` (AT&T syntax)

`MOV RAX, [0x123400]` (Intel syntax)

which moves the value at memory location `0x123400` to %rax

in the processor + memory bus model, how many times is a message sent from the processor to the memory?

answer: 2

CPU → MEM: What's at (instruction address)?

MEM → CPU: It's (the machine code for the mov)?

CPU → MEM: What's at 0x123400?

MEM → CPU: It's (the value)

exercise

suppose a processor is executing the following instruction

`movq 0x123400, %rax` (AT&T syntax)

`MOV RAX, [0x123400]` (Intel syntax)

which moves the value at memory location `0x123400` to %rax

in the processor + memory bus model, how many times is a message sent from the processor to the memory?

answer: 2

CPU → MEM: What's at (instruction address)?

MEM → CPU: It's (the machine code for the mov)?

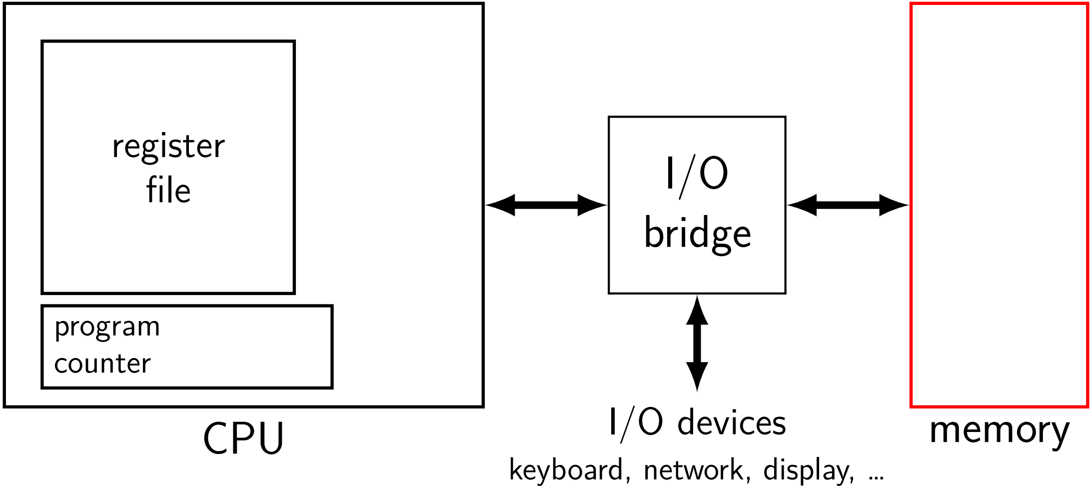
CPU → MEM: What's at 0x123400?

MEM → CPU: It's (the value)

(next instruction)

CPU → MEM: What's at (next instruction address)?

processors and memory



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

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

co-instructor

Sergiu Mosanu

computer engineering PhD student

we will be splitting lectures

lectures and labs attendance

we won't check lecture/lab attendance

lectures will be recorded (assuming not tech. difficulties)

remote submission of labs is possible

not attending lectures?

if you rely on the lecture recordings, I recommend...

a regular schedule of watching them

pausing+trying to answer in-lecture questions

writing down questions you have

...and asking them in Piazza and/or office hours and/or lab

coursework

labs — grading: full credit if threshold amount completed

- none this/next week
- intended: can reliably get 100% within lab time proper
- threshold often somewhat less than full lab
- collaboration permitted
- due by 11:59pm lab day

homework assignments — introduced by lab (mostly)

- due at 4:59pm lab day
- complete individually

weekly quizzes

final exam

coursework

labs — grading: full credit if threshold amount completed

- none this/next week
- intended: can reliably get 100% within lab time proper
- threshold often somewhat less than full lab
- collaboration permitted
- due by 11:59pm lab day

homework assignments — introduced by lab (mostly)

- due at 4:59pm lab day
- complete individually

weekly quizzes

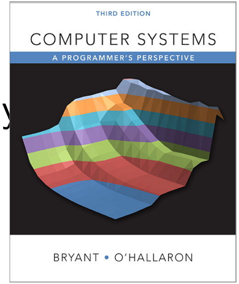
final exam

textbook

Computer Systems: A Programmer's Perspective

recommended — HCL assignments follow pretty closely

(useful, but less important for other topics)



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 next week)

released Thursday night, due Tuesday before lecture

from lecture that week

first quiz after next week

two lowest quiz grades dropped

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 me if illness, etc.

getting help tools

non-real-time help: Piazza (discussion forum)

labs: in person, specified location

office hours: specified on website, calendar

- some in-person, some remote
- online queue for TA help (may not be used for in-person OH)

office hour format

current plan: some in-person and some remote

which is when be noted on schedule

- never in-person+remote at same time

remote times mostly late times or lower-demand days

on the office hour queue

for remote and *some* in-person office hours

sorted by last time helped

- but hope to have enough help that it doesn't matter much

first approx 3 slots may be first-come first-served

we may reset those first three slots between office hours

goal 1: being on the queue overnight won't help you

goal 2: try to spread out the TA help

your TODO list

department account and/or C environment working

- should have department account if you were registered yesterday

before lab next week

grading

Quizzes: 30%

Homeworks: 40%

Labs: 15%

Final Exam: 15%

upcoming lab/HW

bomblab/hw:

using debugger/disassembler,
figure out "correct" input for a program

may want to review x86-64 assembly from CS 2150
(or see textbook chapter/writeup linked off assignment)

quiz demo

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
0x00000000	0xA0

```
int *x = (int*)0x42000;
printf("%d\n", *x);
```

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
0x00000000	0xA0

```
int *x = (int*)0x42000;
printf("%d\n", *x);
```

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
0x00000000	0xA0

```
int *x = (int*)0x42000;
printf("%d\n", *x);
```

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
0x00000000	0xA0

```
int *x = (int*)0x42000;
printf("%d\n", *x);
```

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
0x00000000	0xA0

```
int *x = (int*)0x42000;
printf("%d\n", *x);
```

0x03020100 = 50462976

little endian
(least significant byte has lowest address)

0x00010203 = 66051

big endian
(most significant byte has lowest address)

exercise

buffer

```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i];
    buffer[i+4] = ptr_value2[i];
}
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

What is `value1` after this runs on a little-endian system?

A. `0x0F654321` **B.** `0x123456F0` **C.** `0x3456789A`
D. `0x345678F0` **E.** `0x9A123456` **F.** `0x9A785634`
G. `0xF0123456` **H.** `0xF2345678` **I.** something else

exercise

buffer

```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i];
    buffer[i+4] = ptr_value2[i];
}
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

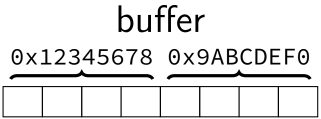
What is `value1` after this runs on a little-endian system?

A. 0x0F654321 **B.** 0x123456F0 **C.** 0x3456789A

D. 0x345678F0 **E.** 0x9A123456 **F.** 0x9A785634

G. 0xF0123456 **H.** 0xF2345678 **I.** something else

exercise




```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i];
    buffer[i+4] = ptr_value2[i];
}
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

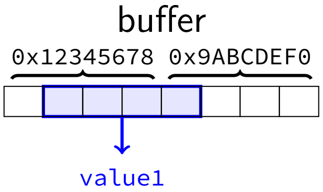
What is `value1` after this runs on a little-endian system?

A. 0x0F654321 **B.** 0x123456F0 **C.** 0x3456789A

D. 0x345678F0 **E.** 0x9A123456 **F.** 0x9A785634

G. 0xF0123456 **H.** 0xF2345678 **I.** something else

exercise




```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i];
    buffer[i+4] = ptr_value2[i];
}
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

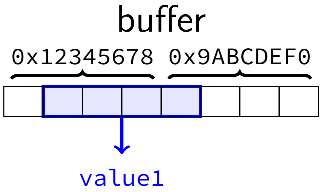
What is `value1` after this runs on a little-endian system?

A. 0x0F654321 **B.** 0x123456F0 **C.** 0x3456789A

D. 0x345678F0 **E.** 0x9A123456 **F.** 0x9A785634

G. 0xF0123456 **H.** 0xF2345678 **I.** something else

exercise




```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i];
    buffer[i+4] = ptr_value2[i];
}
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

What is `value1` after this runs on a little-endian system?

A. `0x0F654321` **B.** `0x123456F0` **C.** `0x3456789A`
D. `0x345678F0` **E.** `0x9A123456` **F.** `0x9A785634`
G. `0xF0123456` **H.** `0xF2345678` **I.** something else

exercise visualization

value1 (bytes in hex)

0	1	2	3
78	56	34	12

0x12345678

value2 (bytes in hex)

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
?	?	?	?	?	?	?	?

```
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i]; buffer[i+4] = ptr_value2[i];
}
```

value1

0	1	2	3
78	56	34	12

0x12345678

value2

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
78	56	34	12	F0	DE	BC	9A

```
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

value1

0	1	2	3
56	34	12	F0

0xF0123456

value2

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
78	56	34	12	F0	DE	BC	9A

exercise visualization

value1 (bytes in hex)

0	1	2	3
78	56	34	12

0x12345678

value2 (bytes in hex)

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
?	?	?	?	?	?	?	?

```
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i]; buffer[i+4] = ptr_value2[i];
}
```

value1

0	1	2	3
78	56	34	12

0x12345678

value2

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
78	56	34	12	F0	DE	BC	9A

```
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

value1

0	1	2	3
56	34	12	F0

0xF0123456

value2

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
78	56	34	12	F0	DE	BC	9A

exercise visualization

value1 (bytes in hex)

0	1	2	3
78	56	34	12

0x12345678

value2 (bytes in hex)

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
?	?	?	?	?	?	?	?

```
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i]; buffer[i+4] = ptr_value2[i];
}
```

value1

0	1	2	3
78	56	34	12

0x12345678

value2

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
78	56	34	12	F0	DE	BC	9A

```
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

value1

0	1	2	3
56	34	12	F0

0xF0123456

value2

0	1	2	3
F0	DE	BC	9A

0x9ABCDEF0

buffer

0	1	2	3	4	5	6	7
78	56	34	12	F0	DE	BC	9A

exercise visualization

value1 (bytes in hex)

78	56	34	12
0	1	2	3

0x12345678

value2 (bytes in hex)

F0	DE	BC	9A
0	1	2	3

0x9ABCDEF0

buffer

?	?	?	?	?	?	?	?
0	1	2	3	4	5	6	7

```
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i]; buffer[i+4] = ptr_value2[i];
}
```

value1

78	56	34	12
0	1	2	3

0x12345678

value2

F0	DE	BC	9A
0	1	2	3

0x9ABCDEF0

buffer

78	56	34	12	F0	DE	BC	9A
0	1	2	3	4	5	6	7

```
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

value1

56	34	12	F0
0	1	2	3

0xF0123456

value2

F0	DE	BC	9A
0	1	2	3

0x9ABCDEF0

buffer

78	56	34	12	F0	DE	BC	9A
0	1	2	3	4	5	6	7

backup slides