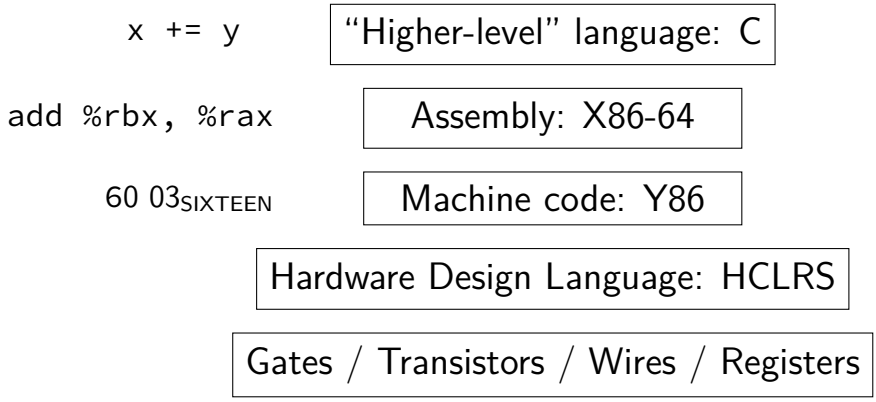
introduction / layers of abstraction

changelog

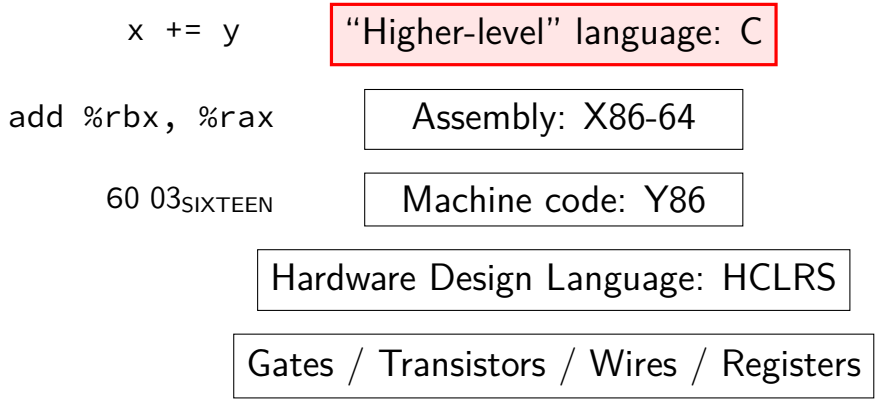
Changes since first lecture:

24 August 2021: processor + memory bus exercise: fix extra % and extra operand in Intel syntax version of instruction

layers of abstraction



layers of abstraction



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

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

some other options:

- 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

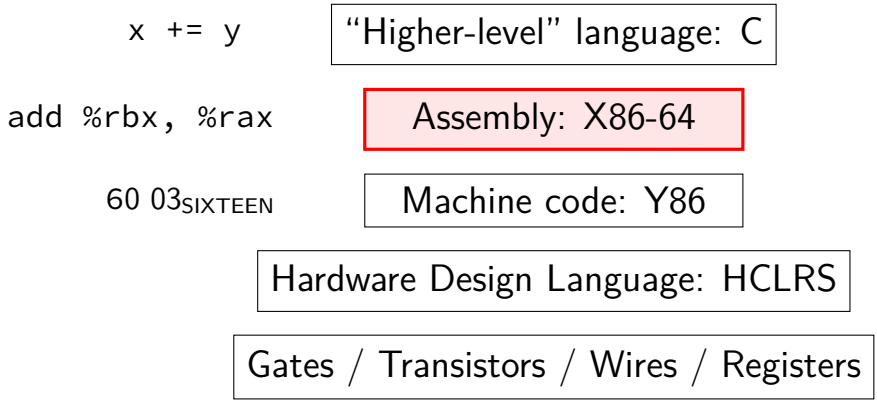
trouble? we'll say to use department machines

most assignments: C and Unix-like environment

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

previously written in D + needed D compiler

layers of abstraction

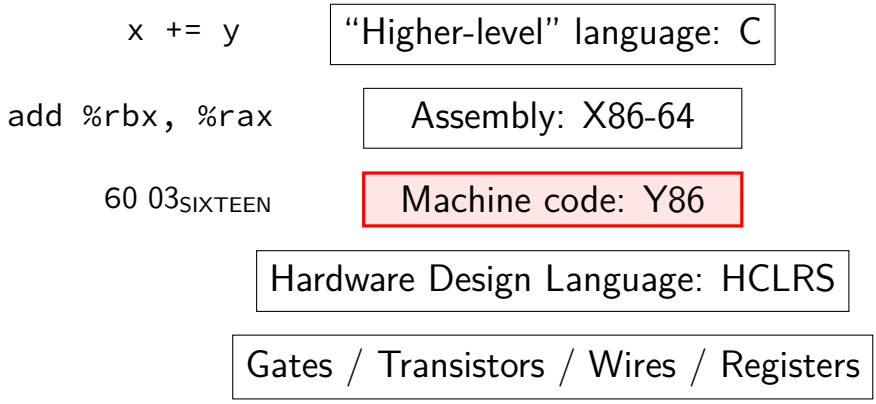


X86-64 assembly

in theory, you know this (CS 2150)

in reality, ...

layers of abstraction



Y86-64??

Y86: our textbook's X86-64 subset

- hope: leverage 2150 assembly knowledge

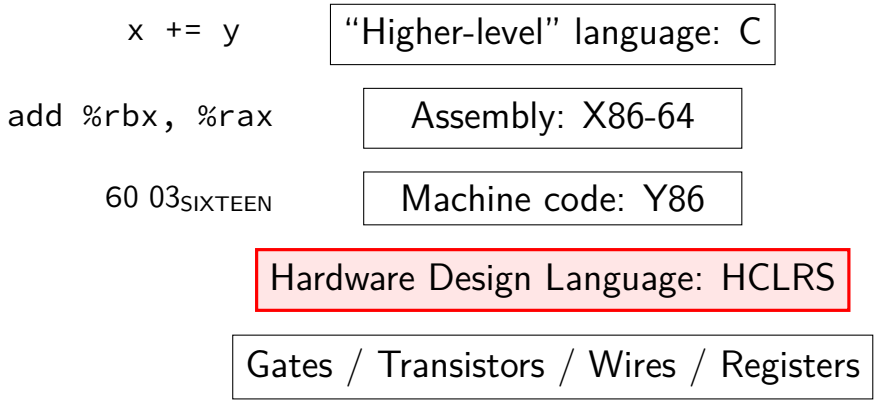
much simpler than real X86-64 encoding

- (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

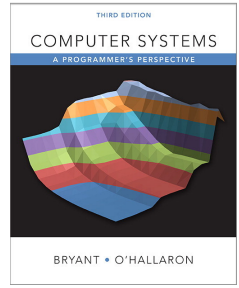


textbook

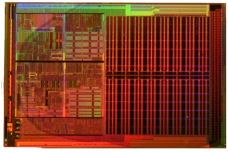
Computer Systems: A Programmer's Perspective

HCL assignments follow pretty closely

(useful, but less important for other topics)



processors and memory

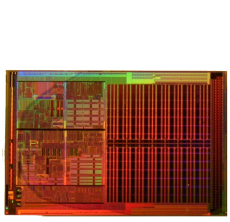


processor

memory

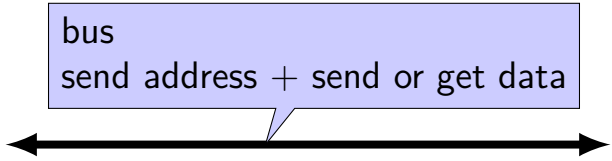
Images:
Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons
SDRAM by Arnaud 25, via Wikimedia Commons

processors and memory



processor

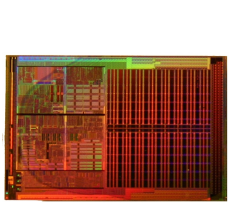
bus
send address + send or get data



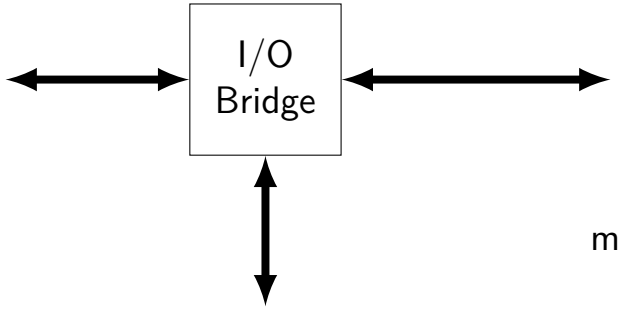
memory

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



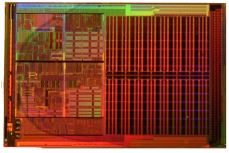
memory

to I/O devices

keyboard, mouse, wifi, ...

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

processors and memory



processor

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

Bridge

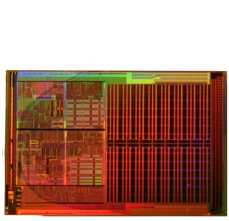
memory

to I/O devices

keyboard, mouse, wifi, ...

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

processors and memory



processor

CPU: send PC: `0x04000`

MEM: send machine code: `pushq %rbp`

Bridge

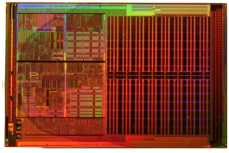
memory

to I/O devices

keyboard, mouse, wifi, ...

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

processors and memory



processor

CPU: send PC: `0x04000`

MEM: send machine code: `pushq %rbp`

CPU: send `0x7fff830` (stack), `0x1234` (rbp)

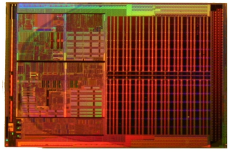
memory

to I/O devices

keyboard, mouse, wifi, ...

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

processors and memory



processor

CPU: send PC: `0x04000`

MEM: send machine code: `pushq %rbp`

CPU: send `0x7fff830` (stack), `0x1234` (rbp)

MEM: stored it

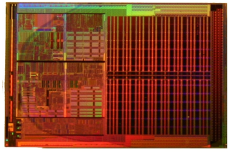
memory

to I/O devices

keyboard, mouse, wifi, ...

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

processors and memory



processor

CPU: send PC: `0x04000`

MEM: send machine code: `pushq %rbp`

CPU: send `0x7fff830` (stack), `0x1234` (rbp)

MEM: stored it

memory

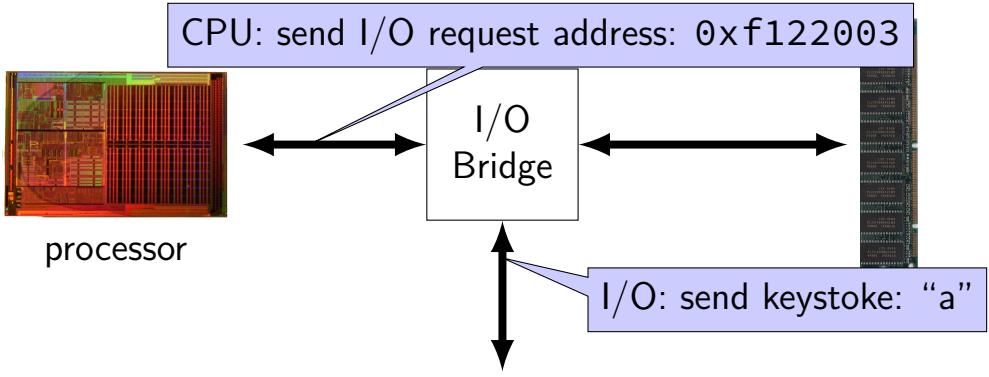
CPU: next PC: `0x04001`

to I/O devices

keyboard, mouse, wifi, ...

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

processors and memory



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

exercise

suppose a processor is executing the following instruction
`movq 0x123400, %rax` (AT&T syntax)
`MOV RAX, [0x123400]` (Intel syntax)

which moves the value of 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 of 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 (retrieve instruction + ask for address 0x123400)

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

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

- threshold often somewhat less than full lab
- collaboration permitted

homework assignments — introduced by lab (mostly)

- due at 9:30am lab day
- complete individually

weekly quizzes

final exam

coursework

labs — grading: full credit if threshold amount completed

- threshold often somewhat less than full lab
- collaboration permitted

homework assignments — introduced by lab (mostly)

- due at 9:30am lab day
- complete individually

weekly quizzes

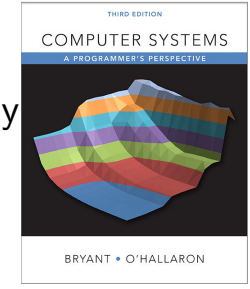
final exam

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 Thursday)

released Thursday night, due Tuesday before *first* lecture

from lecture that 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 (might not be used in smaller rooms)

office hour format

current plan: mix of remote (Discord) + in-person

generally: believe in-person usually more efficient

- but some other practicalities

which it is will be noted on schedule

- never in-person+remote at smae time

common office hour queue

on the office hour queue

except for first three slots, queue is sorted by last time helped

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

- department accounts should happen by this weekend

before lab next week

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)

grading

Quizzes: 30%

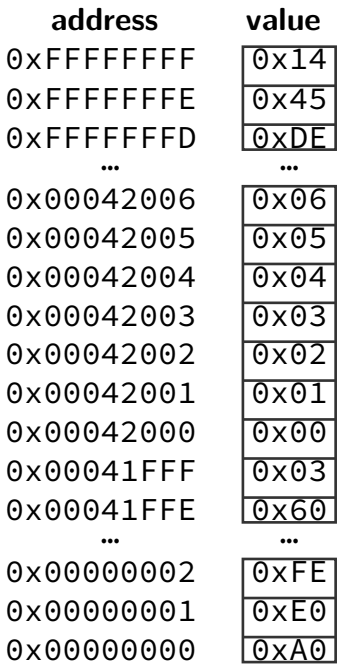
Homeworks: 40%

Labs: 15%

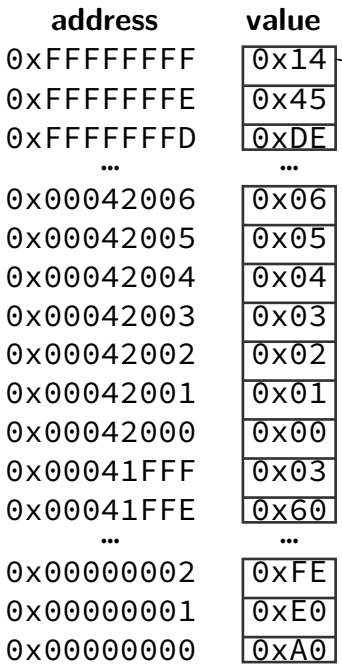
Final Exam: 15%

quiz demo

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



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

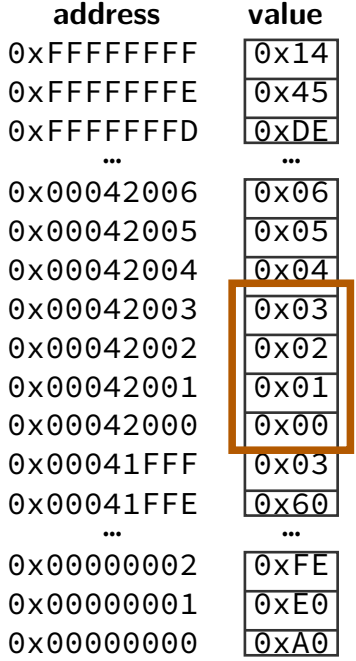
address	value
0x00000000	0xA0
0x00000001	0xE0
0x00000002	0xFE
…	…
0x00041FFE	0x60
0x00041FFF	0x03
0x00042000	0x00
0x00042001	0x01
0x00042002	0x02
0x00042003	0x03
0x00042004	0x04
0x00042005	0x05
0x00042006	0x06
…	…
0xFFFFFFFD	0xDE
0xFFFFFFFE	0x45
0xFFFFFFFF	0x14

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
0x00000000	0xA0

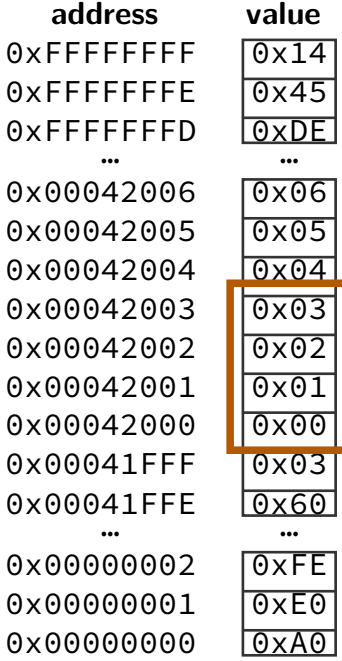
```
int *x = (int*)0x42000;
printf("%d\n", *x);
```

endianness




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

endianness

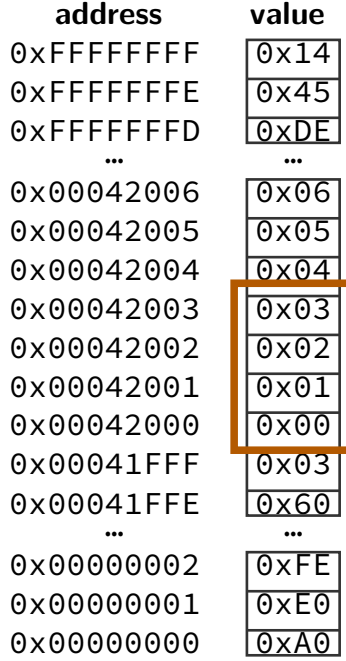



```
int *x = (int*)0x42000;
printf("%d\n", *x);
```

0x03020100 = 50462976

0x00010203 = 66051

endianness




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

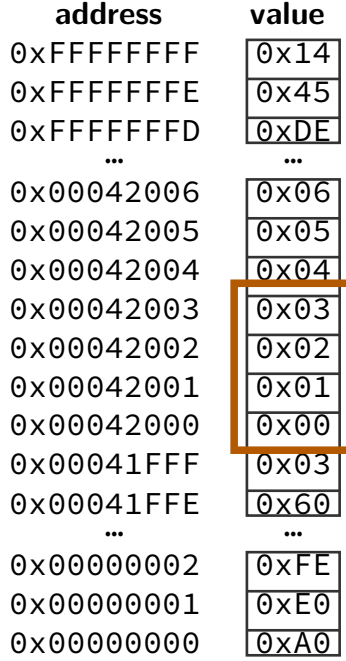
0x03020100 = 50462976

little endian
(least significant byte has lowest address)

0x00010203 = 66051

big endian
(most significant byte has lowest address)

endianness




```
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

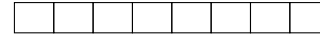
```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy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

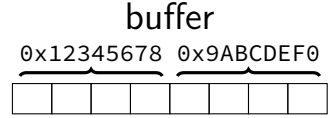


```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy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

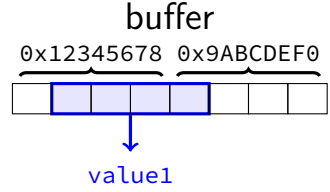



```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy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

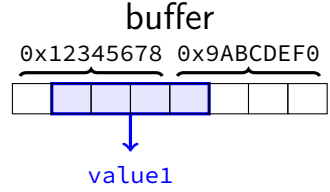



```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy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



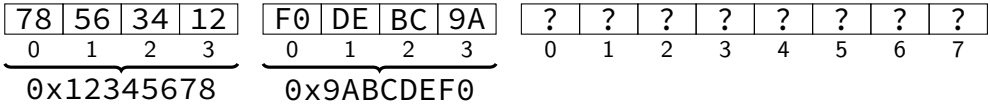

```
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy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

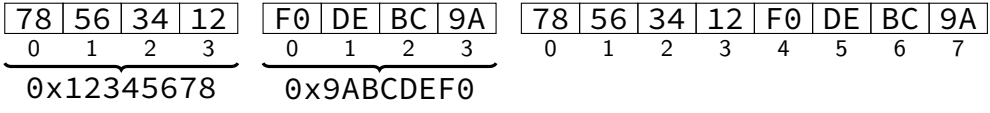
exercise visualization

value1 (bytes in hex) value2 (bytes in hex) buffer



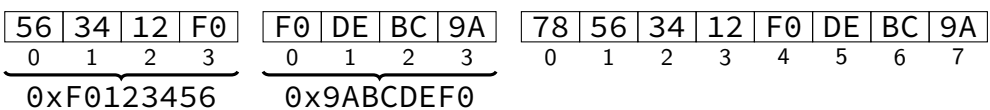

```
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i]; buffer[i+4] = ptr_value2[i];
}
```

value1 value2 buffer



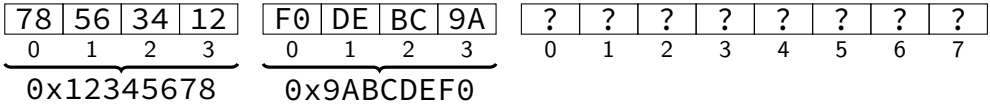

```
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

value1 value2 buffer



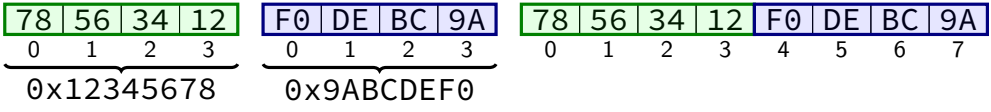
exercise visualization

value1 (bytes in hex) value2 (bytes in hex) buffer



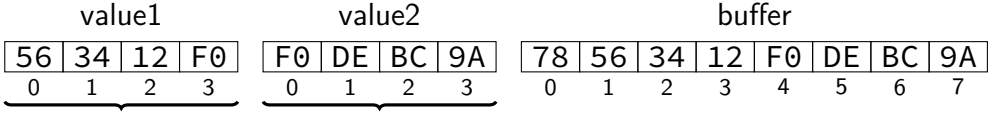

```
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i]; buffer[i+4] = ptr_value2[i];
}
```

value1 value2 buffer



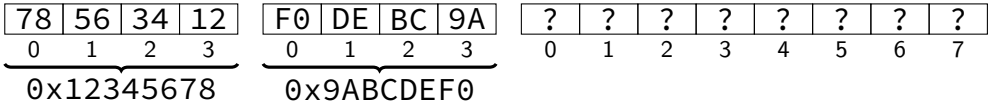

```
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

value1 value2 buffer



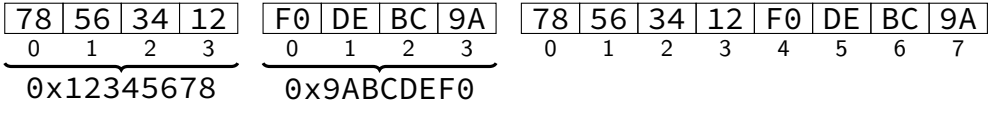
exercise visualization

value1 (bytes in hex) value2 (bytes in hex) buffer



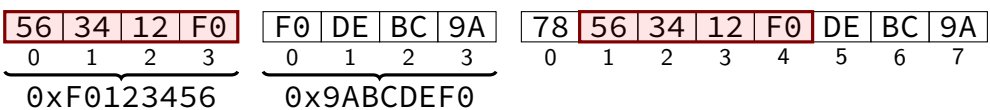

```
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i]; buffer[i+4] = ptr_value2[i];
}
```

value1 value2 buffer



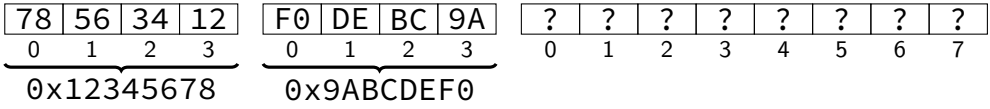

```
for (int i = 0; i < 4; ++i) { /* copy buffer[1..5] into value1 */
    ptr_value1[i] = buffer[i+1];
}
```

value1 value2 buffer



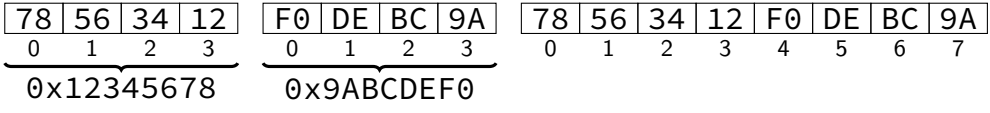
exercise visualization

value1 (bytes in hex) value2 (bytes in hex) buffer



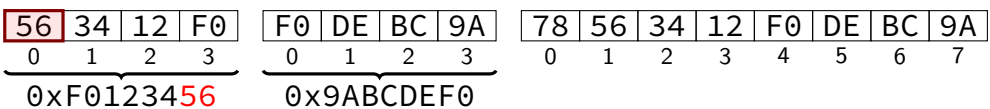

```
for (int i = 0; i < 4; ++i) { /* copy value1/2 into buffer */
    buffer[i] = ptr_value1[i]; buffer[i+4] = ptr_value2[i];
}
```

value1 value2 buffer




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

value1 value2 buffer



AT&T versus Intel syntax by example

```
movq $42, (%rbx)
                mov QWORD PTR [rbx], 42
subq %rax, %r8
                sub r8, rax
movq $42, 100(%rbx,%rcx,4)
                mov QWORD PTR [rbx+rcx*4+100], 42
jmp *%rax
                 jmp rax
jmp *1000(%rax,%rbx,8)
                jmp QWORD PTR [RAX+RBX*8+1000]
```

AT&T versus Intel syntax (1)

AT&T syntax:

```
movq $42, (%rbx)
```

Intel syntax:

```
mov QWORD PTR [rbx], 42
```

effect (pseudo-C):

```
memory[rbx] <- 42
```

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T syntax: addressing

`100(%rbx)`: `memory[rbx + 100]`

`100(%rbx,8)`: `memory[rbx * 8 + 100]`

`100(,%rbx,8)`: `memory[rbx * 8 + 100]`

`100(%rcx,%rbx,8)`:
`memory[rcx + rbx * 8 + 100]`

`100`:
`memory[100]`

`100(%rbx,%rcx)`:
`memory[rbx+rcx+100]`

AT&T versus Intel syntax (3)

`r8` ← `r8 - rax`

AT&T syntax: **`subq`** `%rax, %r8`
Intel syntax: **`sub`** `r8, rax`

same for **`cmp`**

- after **`cmpq`** `%rax, %r8`,
  `jg` jumps if `%r8` is greater

AT&T syntax: addresses

```
addq 0x1000, %rax
// Intel syntax: add rax, QWORD PTR [0x1000]
// rax ← rax + memory[0x1000]
addq $0x1000, %rax
// Intel syntax: add rax, 0x1000
// rax ← rax + 0x1000
```

no $ — probably memory address

AT&T syntax in one slide

destination last

`()` means value in memory

`disp(base, index, scale)` same as
`memory[disp + base + index * scale]`

- omit disp (defaults to `0`)
- and/or omit base (defaults to `0`)
- and/or scale (defualts to `1`)

`$` means constant

plain number/label means value in memory

extra detail: computed jumps

```
jmpq *%rax
// Intel syntax: jmp RAX
    // goto RAX
jmpq *1000(%rax,%rbx,8)
// Intel syntax: jmp QWORD PTR[RAX+RBX*8+1000]
    // read address from memory at RAX + RBX * 8 + 1
    // go to that address
```

AT&T versus Intel syntax by example

```
movq $42, (%rbx)
                mov QWORD PTR [rbx], 42
subq %rax, %r8
                sub r8, rax
movq $42, 100(%rbx,%rcx,4)
                mov QWORD PTR [rbx+rcx*4+100], 42
jmp *%rax
                 jmp rax
jmp *1000(%rax,%rbx,8)
                jmp QWORD PTR [RAX+RBX*8+1000]
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

swap (Intel syntax)

```
swap:
  mov RAX, QWORD PTR [RDI]
  mov RDX, QWORD PTR [RSI]
  mov QWORD PTR [RDI], RDX
  mov QWORD PTR [RSI], RAX
  ret
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

as pseudocode

```
swap:
    RAX ← memory[RDI (arg 1)]
    RDX ← memory[RSI (arg 2)]
    memory[RDI (arg 1)] ← RDX
    memory[RSI (arg 2)] ← RAX
    return
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

registers

%rax	???
%rdx	???
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFFF8
…	…

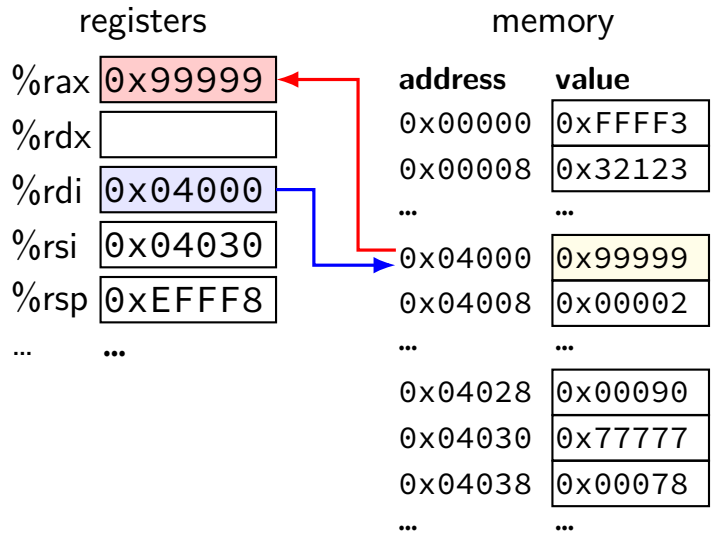
memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
…	…
0x04000	0x99999
0x04008	0x00002
…	…
0x04028	0x00090
0x04030	0x77777
0x04038	0x00078
…	…

swap

swap (AT&T syntax)

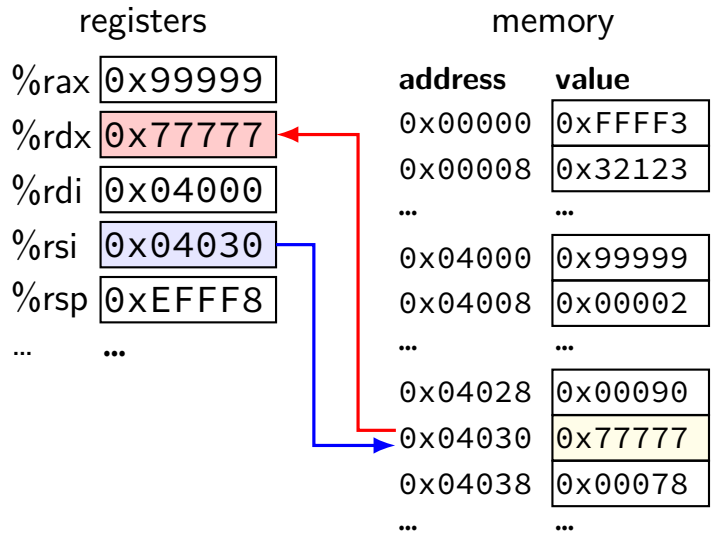
```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```



swap

swap (AT&T syntax)

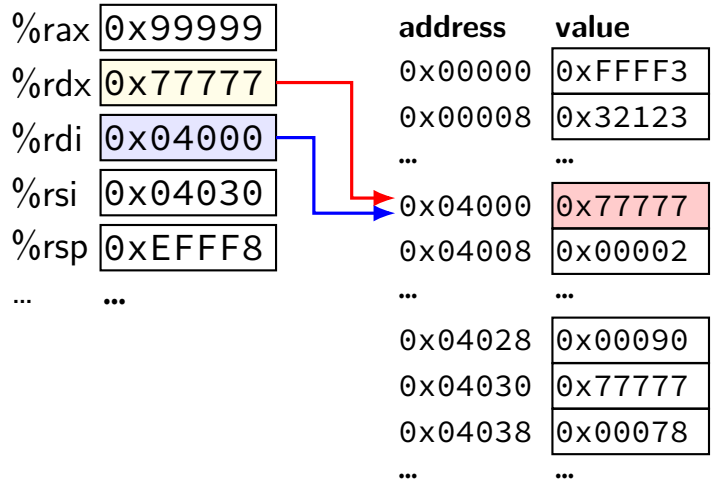
```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```



swap

swap (AT&T syntax)

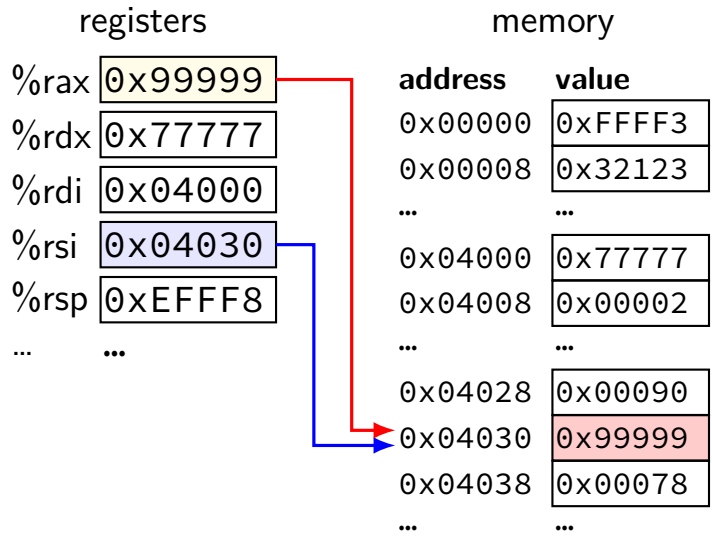
```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```



swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```



swap

swap (AT&T syntax)

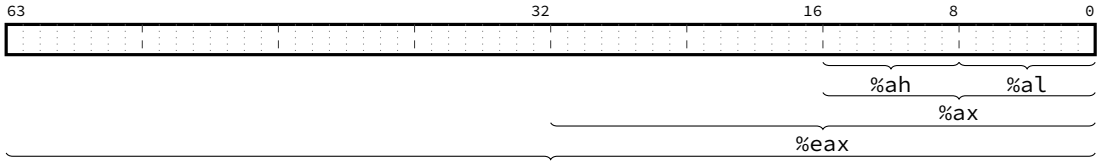
```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

registers

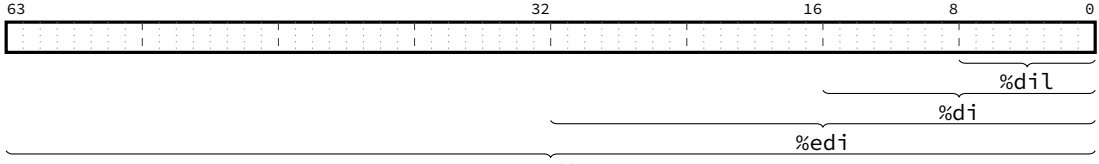
%rax	0x99999
%rdx	0x77777
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFFF8
…	…

memory

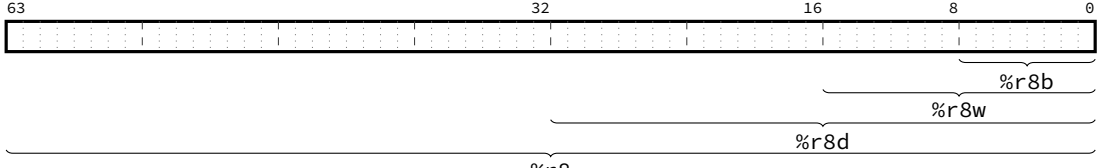
address	value
0x00000	0xFFFF3
0x00008	0x32123
…	…
0x04000	0x77777
0x04008	0x00002
…	…
0x04028	0x00090
0x04030	0x99999
0x04038	0x00078
…	…



+ similar for %rcx, %rbx, %rdx



+ similar for %rbp, %rsi, %rsp



+ similar for %r9 through %r15

recall: x86-64 general purpose registers

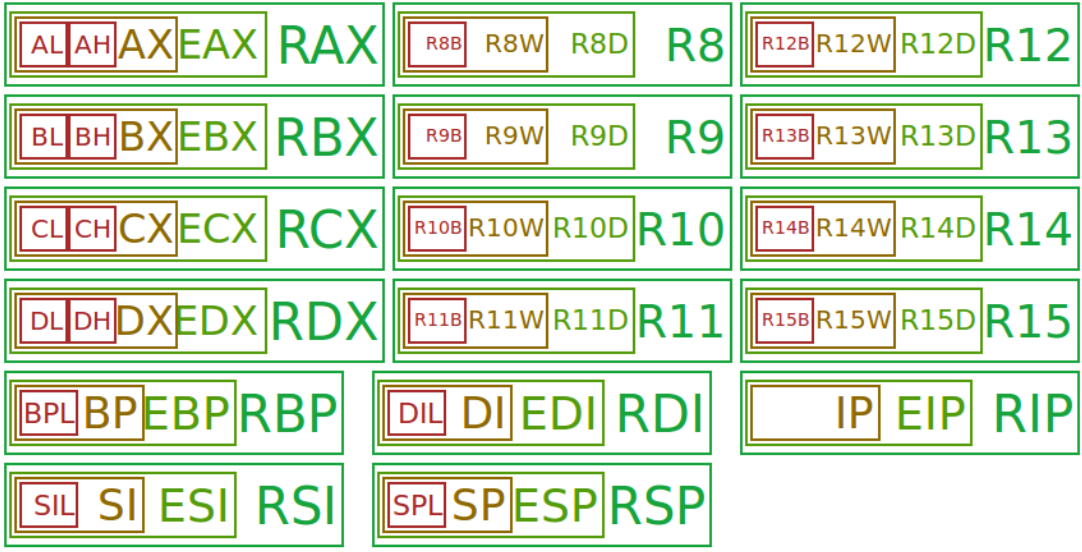


Figure: Immae via Wikipedia

overlapping registers (1)

setting 32-bit registers — clears corresponding 64-bit register

```
movq $0xFFFFFFFFFFFFFFFF, %rax
movl $0x1, %eax
```

%rax is `0x1` (not `0xFFFFFFFF00000001`)

overlapping registers (2)

setting 8/16-bit registers: don't clear 64-bit register

```
movq $0xFFFFFFFFFFFFFFFF, %rax
movb $0x1, %al
```

%rax is `0xFFFFFFFFFFFFFF01`

labels (1)

labels represent addresses

labels (2)

```
    addq string, %rax
    // intel syntax: add rax, QWORD PTR [label]
    // rax ← rax + memory[address of "a string"]
    addq $string, %rax
    // intel syntax: add rax, OFFSET label
    // rax ← rax + address of "a string"
string:  .ascii "a string"
```

`addq label`: read value at the address

`addq $label`: use address as an integer constant

exericse

```
hello:
    .string "Hello, World!" ; nul-terminated string
example:
    movb hello+1, %bl
    subb $1, %bl
    movb %bl, hello
    movq $hello, %rdi
    ; int puts(const char *s [%rdi])
    callq puts
    ret
```

What is the the argument to puts, %rdi?

A. a pointer to 'Hello, World!' B. a pointer to 'dello, World!'
C. a pointer to 'Hdllo, World!' D. a pointer to 'fello, World!'
E. a pointer to 'Jello, World!' F. a pointer to a different string
G. an integer constructed from the ASCII for 'Hello, W' (puts probably crashes)
H. an integer constructed from the ASCII for 'Jello, W' (puts probably crashes)
I. an integer constructed from the ASCII for a different string (puts probably crashes)

on LEA

LEA = **L**oad **E**ffective **A**ddress

effective address = computed address for memory access

syntax looks like a **mov** from memory, but...

skips the memory access — just uses the address

(sort of like & operator in C?)

`leaq 4(%rax), %rax` ≈ `addq $4, %rax`

on LEA

LEA = **L**oad **E**ffective **A**ddress

effective address = computed address for memory access

syntax looks like a **mov** from memory, but...

skips the memory access — just uses the address

(sort of like & operator in C?)

`leaq 4(%rax), %rax` ≈ `addq $4, %rax`

"address of memory[rax + 4]" = rax + 4

LEA tricks

```
leaq (%rax,%rax,4), %rax
```

rax $\leftarrow$ rax $\times$ 5

rax $\leftarrow$ `address-of(memory[rax + rax * 4])`

```
leaq (%rbx,%rcx), %rdx
```

rdx $\leftarrow$ rbx $+$ rcx

rdx $\leftarrow$ `address-of(memory[rbx + rcx])`

exercise: what is this function?

```
mystery:
    leal 0(,%rdi,8), %eax
    subl %edi, %eax
    ret
```

```
int mystery(int arg) { return ...; }
```

A. `arg * 9`
B. `-arg * 9`
C. `arg * 8`
D. `-arg * 7`
E. none of these
F. it has a different prototype

exercise: what is this function?

```
mystery:
    leal 0(,%rdi,8), %eax
    subl %edi, %eax
    ret
```

```
int mystery(int arg) { return ...; }
```

A. `arg * 9`
B. `-arg * 9`
C. `arg * 8`
D. `-arg * 7`
E. none of these
F. it has a different prototype