assembly syntax/review

Changelog

27 August 2020: fixup animation for last two movs on swap slide

27 August 2020: edited %bx to %bl, because otherwise we can't movb

last time

logistics + course preview

layers of abstraction

what a processor does (simple model)

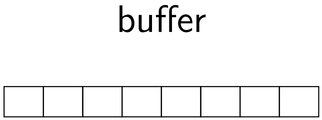
- fetch instruction from memory
- interpret machine code
- possibly get/set values in memory for instruction
- fetch next instruction from memory
- ...

endianness

- little endian = least significant byte in lowest address
- least significant byte — part with least influence on value
- think: "1's place"

quiz demo

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

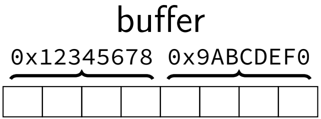
buffer

```
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

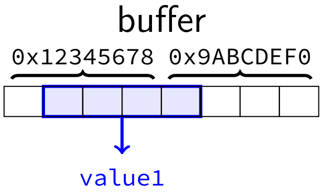



```c
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy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

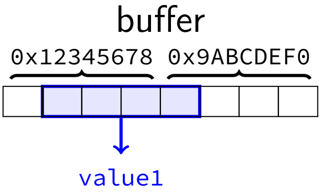



```c
unsigned char buffer[8] =
    { 0, 0, /* ..., */ 0 };
/* uint32_t = 32-bit unsigned int */
uint32_t value1 = 0x12345678;
uint32_t value2 = 0x9ABCDEF0;
unsigned char *ptr_value1 = (unsigned char *) &value1;
unsigned char *ptr_value2 = (unsigned char *) &value2;
for (int i = 0; i < 4; ++i) { /* copy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

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

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

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

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
…	…

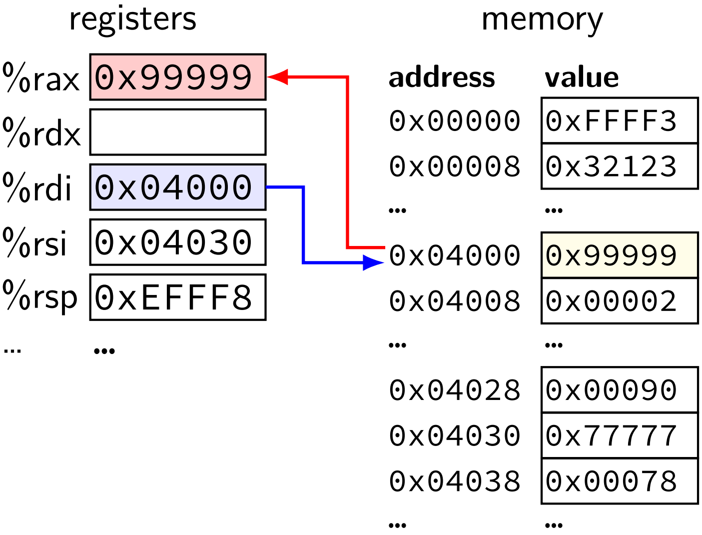
memory

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

swap

swap (AT&T syntax)

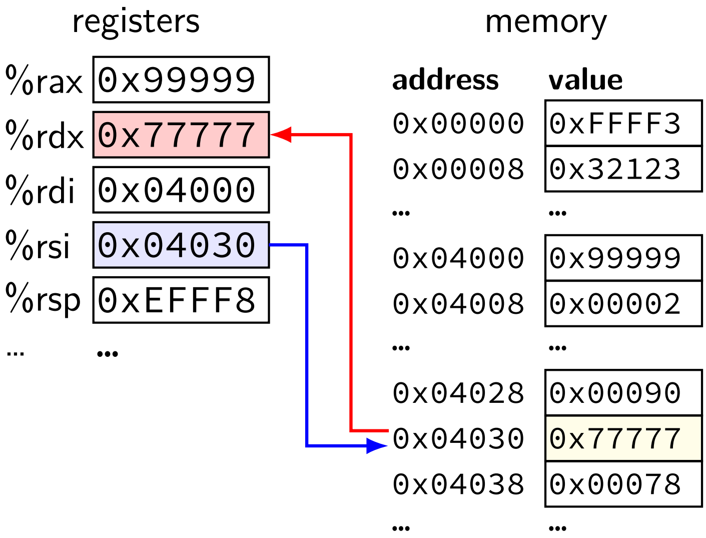
```
// swap(long *rdi,
//      long *rsi)
swap:
   movq (%rdi), %rax
   movq (%rsi), %rdx
   movq %rdx, (%rdi)
   movq %rax, (%rsi)
   ret
```



swap

swap (AT&T syntax)

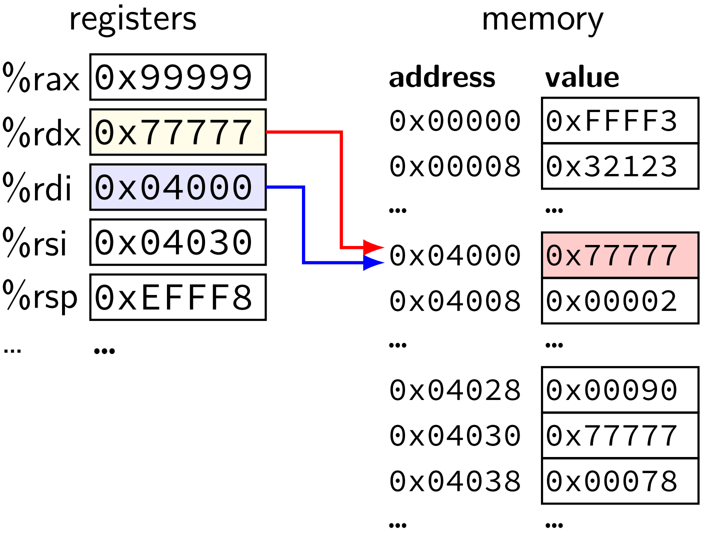
```
// swap(long *rdi,
//      long *rsi)
swap:
   movq (%rdi), %rax
   movq (%rsi), %rdx
   movq %rdx, (%rdi)
   movq %rax, (%rsi)
   ret
```



swap

swap (AT&T syntax)

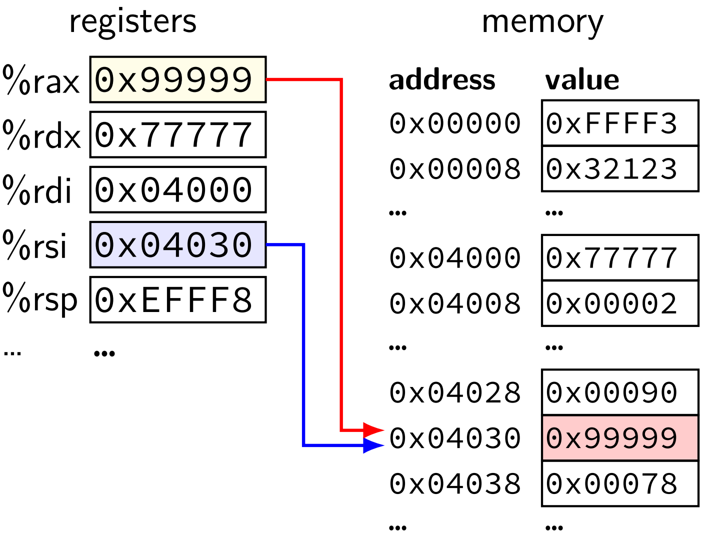
```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```



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
…	…

recall: x86-64 general purpose registers

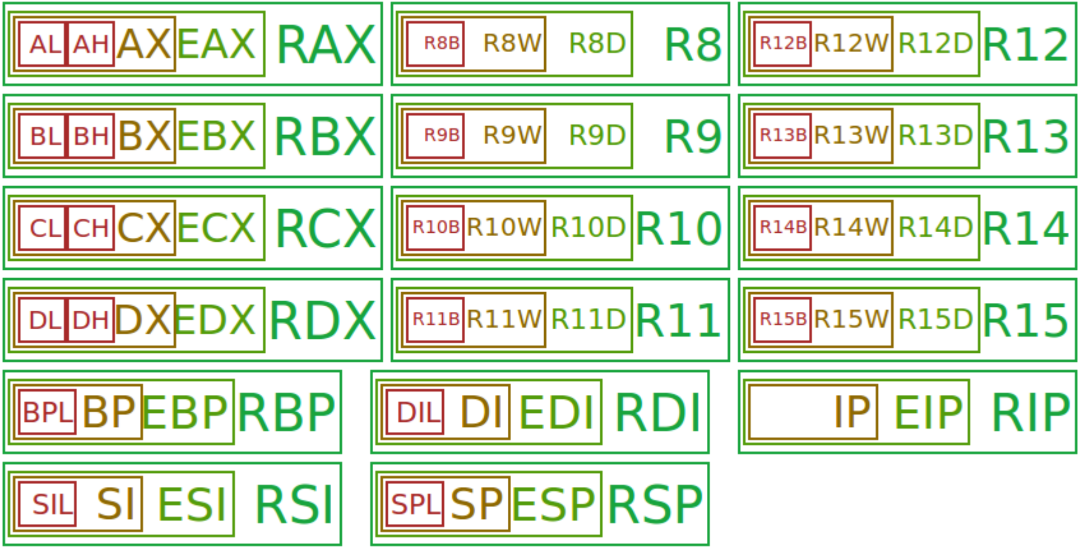


Figure: Immae via Wikipedia

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

backup slides

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
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)