assembly 2

changelog

Changes since first lecture:

31 August 2021: be more explicit about what call pushes

31 August 2021: condition codes example (1b): correct last example (swap -10/0)

1 September 2021: rephrase assembly to C ASM exericse; add soln slide

last time

AT&T syntax

- destination last
- $: constant; %: register
- disp(base,index,scale) + same with parts omitted

labels replaced by address (number) of thing labelled

lea — compute address and place it in destination

- "effective address" — actual data address used by an instruction

`%rip` — program counter

- typically used in mov/lea address computation
- label(%rip) = label (very special case)

Linux x86-64 calling convention

registers for first 6 arguments:

- `%rdi` (or `%edi` or `%di`, etc.), then
- `%rsi` (or `%esi` or `%si`, etc.), then
- `%rdx` (or `%edx` or `%dx`, etc.), then
- `%rcx` (or `%ecx` or `%cx`, etc.), then
- `%r8` (or `%r8d` or `%r8w`, etc.), then
- `%r9` (or `%r9d` or `%r9w`, etc.)

rest on stack (pushed before return address)

return value in `%rax`

don't memorize: Figure 3.28 in book

x86-64 calling convention example

```c
int foo(int x, int y, int z) { return 42; }
...
    foo(1, 2, 3);
...
```

```
...
    // foo(1, 2, 3)
    movl $1, %edi
    movl $2, %esi
    movl $3, %edx
    call foo  // call pushes address of next instruction
              // then jumps to foo
...
foo:
    movl $42, %eax
    ret
```

call/ret

call:

- push address just past the end of the call instruction on the stack
  - "return address" – where we expect to go when function finishes

ret:

- pop address from stack; jump

callee-saved registers

functions must preserve these

`%rsp` (stack pointer), `%rbx`, `%rbp` (frame pointer, maybe)

`%r12`–`%r15`

caller/callee-saved

```
foo:
    pushq %r12 // r12 is callee-saved
    ... use r12 ...
    popq %r12
    ret

...
other_function:
    ...
    pushq %r11 // r11 is caller-saved
    callq foo
    popq %r11
```

selected things we won't cover (today)

floating point; vector operations (multiple values at once)

- special registers: %xmm0 through %xmm15

segmentation (special registers: %ds, %fs, %gs, ...)

lots and lots of instructions

conditionals in x86 assembly

```
    if (rax != 0)
        foo();
```

```
    cmpq $0, %rax
    // ***
    je skip_call_foo
    call foo
skip_call_foo:
```

---

how does `je` know the result of the comparison?

what happens if we add extra instructions at the ***?

condition codes

x86 has condition codes

special registers set by (almost) all arithmetic instructions

- addq, subq, imulq, etc.

store info about last arithmetic result

- was it zero? was it negative? etc.

condition codes and jumps

`jg`, `jle`, etc. read condition codes

named based on interpreting result of subtraction

alternate view: comparing result to 0

0: equal; negative: less than; positive: greater than

condition codes: closer look

`ZF` ("zero flag") — was result zero? (sub/cmp: equal)

- e.g. JE (jump if equal) checks for ZF $= 1$

`SF` ("sign flag") — was result negative? (sub/cmp: less)

- e.g. JL (jump if less than) checks for SF $= 1$ (plus extra case for overflow)
- e.g. JLE checks for SF $= 1$ or ZF $= 1$ (plus overflow)

(and some more, e.g. to handle overflow)

condition codes: closer look

`ZF` (“zero flag”) — was result zero? (sub/cmp: equal)

- e.g. JE (jump if equal) checks for ZF $= 1$

`SF` (“sign flag”) — was result negative? (sub/cmp: less)

- e.g. JL (jump if less than) checks for SF $= 1$ (plus extra case for overflow)
- e.g. JLE checks for SF $= 1$ or ZF $= 1$ (plus overflow)

`OF` (“overflow flag”) — did computation overflow (as signed)?

- we won't test on this/use it in later assignments
- signed conditional jumps: JL, JLE, JG, JGE, …

`CF` (“carry flag”) — did computation overflow (as unsigned)?

- we won't test on this/use it in later assignments
- unsigned conditional jumps: JB, JBE, JA, JAE, …

(and one more)

condition codes: closer look

`ZF` (“zero flag”) — was result zero? (sub/cmp: equal)

- e.g. JE (jump if equal) checks for ZF $= 1$

`SF` (“sign flag”) — was result negative? (sub/cmp: less)

- e.g. JL (jump if less than) checks for SF $= 1$ (plus extra case for overflow)
- e.g. JLE checks for SF $= 1$ or ZF $= 1$ (plus overflow)

`OF` (“overflow flag”) — did computation overflow (as signed)?

- we won’t test on this/use it in later assignments
- signed conditional jumps: JL, JLE, JG, JGE, ...

`CF` (“carry flag”) — did computation overflow (as unsigned)?

- we won’t test on this/use it in later assignments
- unsigned conditional jumps: JB, JBE, JA, JAE, ...

(and one more)

condition codes (and other flags) in GDB

```
(gdb) info registers
rax            0x0                 0
rbx            0x555555555150      93824992235856
rcx            0x555555555150      93824992235856
...
rip            0x55555555513a      0x55555555513a <m
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
...
```

ZF = 1 (listed); SF, OF, CF clear

some other flags that you can lookup (PF, IF) also shown

condition codes example (1)

```
movq $−10, %rax
movq $20, %rbx
subq %rax, %rbx // %rbx - %rax = 30
  // result > 0: %rbx was > %rax
jle foo // not taken; 30 > 0
```

condition codes example (1)

```
movq $−10, %rax
movq $20, %rbx
subq %rax, %rbx // %rbx - %rax = 30
  // result > 0: %rbx was > %rax
jle foo // not taken; 30 > 0
```

30: SF = 0 (not negative), ZF = 0 (not zero)

condition codes example (1b)

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx // %rbx - %rax = 30
jle foo // not taken; 30 > 0
```

30: SF = 0 (not negative), ZF = 0 (not zero)

```
movq $20, %rax
movq $20, %rbx
subq %rax, %rbx // %rbx - %rax = 0
jle foo // taken; 0 <= 0
```

0: SF = 0 (not negative), ZF = 1 (zero)

```
movq $0, %rax
movq $-10, %rbx
subq %rax, %rbx // %rbx - %rax = -10
jle foo // taken; -10 <= 0
```

-10: SF = 1 (negative), ZF = 0 (not zero)

condition codes and cmpq

"last arithmetic result"???

then what is cmp, etc.?

`cmp` does subtraction (but doesn't store result)

similar `test` does bitwise-and

`testq %rax, %rax` — result is %rax

what sets condition codes

most instructions that compute something set condition codes

some instructions only set condition codes:

- `cmp` $\sim$ `sub`
- `test` $\sim$ `and` (bitwise and — later)
- `testq %rax, %rax` — result is %rax

some instructions don't change condition codes:

- `lea`, `mov`
- control flow: `jmp`, `call`, `ret`, `jle`, etc.

how do you know? — check processor's manual

condition codes example (2)

```
movq $–10, %rax  // rax <- (-10)
movq $20, %rbx   // rbx <- 20
cmpq %rax, %rbx  // set cond codes w/ rbx - rax
jle foo // not taken; %rbx - %rax > 0
```

condition codes example (2)

```
movq $−10, %rax  // rax <- (-10)
movq $20, %rbx   // rbx <- 20
cmpq %rax, %rbx  // set cond codes w/ rbx - rax
jle foo // not taken; %rbx - %rax > 0
```

%rbx - %rax = 30: SF = 0 (not negative), ZF = 0 (not zero)

omitting the cmp

```
    movq $99, %r12              // x (r12) ← 99
start_loop:
    call foo                    // foo()
    subq $1, %r12               // x (r12) ← x - 1
    cmpq $0, %r12
    // compute x (r12) - 0 + set cond. codes
    jge  start_loop             // r12 >= 0?
                                // or result >= 0?
```

```
    movq $99, %r12              // x (r12) ← 99
start_loop:
    call foo                    // foo()
    subq $1, %r12               // x (r12) ← x - 1
    jge  start_loop             // new r12 >= 0?
```

condition code exercise

```
movq %rcx, %rdx
subq $1, %rdx
addq %rdx, %rcx
```

Assuming no overflow, possible values of SF, ZF?

A. SF $= 0$, ZF $= 0$
B. SF $= 1$, ZF $= 0$
C. SF $= 0$, ZF $= 1$
D. SF $= 1$, ZF $= 1$

condition code exercise

```
movq %rcx, %rdx
subq $1, %rdx
addq %rdx, %rcx
```

~~Assuming no overflow,~~ possible values of SF, ZF?

A. SF = 0, ZF = 0
B. SF = 1, ZF = 0
C. SF = 0, ZF = 1
D. SF = 1, ZF = 1

if-to-assembly (1)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

if-to-assembly (1)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

```
            if (b < 42) goto after_then;
            a += 10;
            goto after_else;
after_then: a *= b;
after_else:
```

if-to-assembly (2)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

```
// a is in %rax, b is in %rbx
    cmpq $42, %rbx    // computes rbx - 42 to 0
                      // i.e compare rbx to 42
    jl after_then     // jump if rbx - 42 < 0
                      // AKA rbx < 42
    addq $10, %rax    // a += 10
    jmp after_else
after_then:
    imulq %rbx, %rax // rax = rax * rbx
after_else:
```

exercise

```
    subq %rax, %rbx
    addq %rbx, %rcx
    je after
    addq %rax, %rcx
after:
```

Same as which of these C snippets? (rax = var. assigned to register %rax, etc.)

A

```
rbx -= rax;
rcx += rbx;
if (rcx == 0) {
    rcx += rax;
}
```

B

```
rbx -= rax;
rcx += rbx;
if (rbx == rcx) {
    rcx += rax;
}
```

C

```
rbx -= rax;
rcx += rbx;
if (rbx + rcx == 0) {
    rcx += rax;
}
```

D

```
rcx += (rbx - rax);
if (rcx == (rbx - rax)) {
    rcx += rax;
}
```

exercise

```
    subq %rax, %rbx
    addq %rbx, %rcx
    je after
    addq %rax, %rcx
after:
```

would be correct with flipped compare

Same as which of these C snippets? (rax = var. assigned to register %rax, etc.)

A

```
rbx -= rax;
rcx += rbx;
if (rcx != 0) {
    rcx += rax;
}
```

B

```
rbx -= rax;
rcx += rbx;
if (rbx == rcx) {
    rcx += rax;
}
```

C

```
rbx -= rax;
rcx += rbx;
if (rbx + rcx == 0) {
    rcx += rax;
}
```

D

```
rcx += (rbx - rax);
if (rcx == (rbx - rax)) {
    rcx += rax;
}
```

while-to-assembly (1)

```
while (x >= 0) {
    foo()
    x--;
}
```

while-to-assembly (1)

```
while (x >= 0) {
    foo()
    x--;
}
```

```
start_loop:
    if (x < 0) goto end_loop;
    foo()
    x--;
    goto start_loop:
end_loop:
```

while-to-assembly (2)

```
start_loop:
    if (x < 0) goto end_loop;
    foo()
    x--;
    goto start_loop:
end_loop:
```

```
start_loop:
    cmpq $0, %r12
    jl end_loop // jump if r12 - 0 < 0
    call foo
    subq $1, %r12
    jmp start_loop
```

while exercise

```
while (b < 10) { foo(); b += 1; }
```

Assume b is in callee-saved register %rbx. Which are correct assembly translations?

```
// version A
start_loop:
  call foo
  addq $1, %rbx
  cmpq $10, %rbx
  jl start_loop
```

```
// version B
start_loop:
  cmpq $10, %rbx
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
```

```
// version C
start_loop:
  movq $10, %rax
  subq %rbx, %rax
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
```

while exercise: translating?

```
while (b < 10) {
    foo();
    b += 1;
}
```

while exercise: translating?

```
while (b < 10) {
    foo();
    b += 1;
}
```

```
start_loop: if (b < 10) goto end_loop;
            foo();
            b += 1;
            goto start_loop;
end_loop:
```

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

```
start_loop:
  cmpq $10, %rbx
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
    ...
    ...
    ...
    ...
```

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

```
start_loop:
  cmpq $10, %rbx
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
    ...
    ...
    ...
    ...
```

```
  cmpq $10, %rbx
  jge end_loop
start_loop:
  call foo
  addq $1, %rbx
  cmpq $10, %rbx
  jne start_loop
end_loop:
    ...
    ...
    ...
```

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

```
start_loop:
  cmpq $10, %rbx
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
    ...
    ...
    ...
    ...
```

```
  cmpq $10, %rbx
  jge end_loop
start_loop:
  call foo
  addq $1, %rbx
  cmpq $10, %rbx
  jne start_loop
end_loop:
    ...
    ...
    ...
```

```
  cmpq $10, %rbx
  jge end_loop
  movq $10, %rax
  subq %rbx, %rax
  movq %rax, %rbx
start_loop:
  call foo
  decq %rbx
  jne start_loop
  movq $10, %rbx
end_loop:
```

compiling switches (1)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    default: ...
}
```

```
    // same as if statement?
    cmpq $1, %rax
    je code_for_1
    cmpq $2, %rax
    je code_for_2
    cmpq $3, %rax
    je code_for_3
    ...
    jmp code_for_default
```

compiling switches (2)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 100: ...; break;
    default: ...
}
```

```
    // binary search
    cmpq $50, %rax
    jl code_for_less_than_50
    cmpq $75, %rax
    jl code_for_50_to_75
    ...
code_for_less_than_50:
    cmpq $25, %rax
    jl less_than_25_cases
    ...
```

compiling switches (3a)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 100: ...; break;
    default: ...
}
```

```
// jump table
cmpq $100, %rax
jg code_for_default
cmpq $1, %rax
jl code_for_default
jmp *table - 8(,%rax,8)
```

```
table:
  // not instructions
  // .quad = 64-bit (4 x 16) constant
  .quad code_for_1
  .quad code_for_2
  .quad code_for_3
  .quad code_for_4
    ...
```

compiling switches (3b)

```
jmp *table-8(,%rax,8)
```

suppose RAX $= 2$,
`table` located at 0x12500

compiling switches (3b)

```
jmp *table-8(,%rax,8)
```

suppose RAX = 2,
table located at 0x12500

	address	value
	…	…
	0x124F8	…
table	0x12500	0x13008
table + 0x08	0x12508	0x130A0
table + 0x10	0x12510	0x130C8
table + 0x18	0x12518	0x13110
	…	…
	…	…
code_for_1	0x13008	…
	…	…
	…	…
code_for_2	0x130A0	…
	…	…

table — list of code addresses

compiling switches (3b)

```
jmp *table-8(,%rax,8)
```

suppose RAX $= 2$,
`table` located at 0x12500

	address	value
	…	…
	0x124F8	…
table	0x12500	0x13008
table + 0x08	0x12508	0x130A0
table + 0x10	0x12510	0x130C8
table + 0x18	0x12518	0x13110
	…	…
	…	…
code_for_1	0x13008	…
	…	…
	…	…
code_for_2	0x130A0	…
	…	…

$(\text{table} - 8) + \text{rax} \times 8 =$
$\texttt{0x124F8} + \texttt{0x10} = \texttt{0x12508}$

compiling switches (3b)

```
jmp *table-8(,%rax,8)
```

suppose RAX $= 2$,
`table` located at 0x12500

	address	value
	...	...
	0x124F8	...
table	0x12500	0x13008
table + 0x08	0x12508	0x130A0
table + 0x10	0x12510	0x130C8
table + 0x18	0x12518	0x13110
	...	...
	...	...
code_for_1	0x13008	...
	...	...
	...	...
code_for_2	0x130A0	...
	...	...

pointer to machine code

computed jumps

```
    cmpq $100, %rax
    jg code_for_default
    cmpq $1, %rax
    jl code_for_default
    // jump to memory[table + rax * 8]
      // table of pointers to instructions
    jmp *table(,%rax,8)
    // intel: jmp QWORD PTR[rax*8 + table]
    ...
table:
    .quad code_for_1
    .quad code_for_2
    .quad code_for_3
    ...
```

backup slides

exercise

(ignoring overflow) `jge` is taken (jumps to target) when

A. ZF $= 1$
B. SF $= 1$
C. SF $= 1$ and ZF $= 0$
D. SF $= 1$ or ZF $= 1$
E. SF $= 0$
F. something else

condition codes example plus overflow (1)

```
movq $−10, %rax
  // same as: mov $0xFFFFFFFFFFFFFFF6, %rax
movq $20, %rbx
cmpq %rax, %rbx
  // %rbx - %rax
jle foo // not taken; signed: 30 > 0
jbe foo // taken; unsigned: very negative <= 0
```

as signed: 30: SF = 0 (not negative), ZF = 0 (not zero)

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30}$ 30 (overflow!)

`OF` $= 0$ (false) no overflow as signed

`CF` $= 1$ (true) overflow as unsigned

`jbe` (jump below/equal) uses CF to give correct result w/ overflow

condition codes example plus overflow (2)

```
movq $5000000000000000000, %rax
movq $6000000000000000000, %rbx
addq %rax, %rbx
  // %rbx + %rax = (incorrect) -744674407370955161
  // %rbx + %rax = 11000000000000000000 (unsigned,
jle foo // not taken; true signed result > 0
jbe foo // not taken; true unsigned result > 0
```

SF = 1 (negative as signed), ZF = 0 (not zero)

OF = 1 (true) overflow as signed

CF = 0 (false) overflow as unsigned

jle uses OF to realize true result is positive, even though SF is set

do-while-to-assembly (1)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

do-while-to-assembly (1)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    int x = 99;
start_loop:
    foo()
    x--;
    if (x >= 0) goto start_loop;
```

do-while-to-assembly (2)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    cmpq $0, %r12
    // computes r12 - 0 = r12
    jge start_loop // jump if r12 - 0 >= 0
```

condition codes examples (4)

```
movq $20, %rbx
addq $−20, %rbx // result is 0
movq $1, %rax   // irrelevant to cond. codes
je   foo        // taken, result is 0
```

condition codes example: no cmp (3)

```
movq $—10, %rax  // rax ← (-10)
movq $20, %rbx   // rbx ← 20
subq %rax, %rbx  // rbx ← rbx - rax = 30
jle  foo // not taken, %rbx - %rax > 0


movq $20, %rbx   // rbx ← 20
addq $—20, %rbx  // rbx ← rbx + (-20) = 0
je   foo         // taken, result is 0
                 // x - y = 0 -> x = y
```

condition codes: exercise with overflow (1)

```
  // 2^63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2^63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

$\texttt{ZF} = ?$

$\texttt{SF} = ?$

$\texttt{OF} = ?$

$\texttt{CF} = ?$

condition codes: exercise with overflow (1)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \cancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\texttt{ZF} = 0$ (false) not zero rax and rbx not equal

condition codes: exercise with overflow (1)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \cancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\mathtt{ZF} = 0$ (false) not zero rax and rbx not equal

condition codes: exercise with overflow (1)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \cancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\mathrm{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathrm{SF} = 0$ (false)	not negative	rax <= rbx (if correct)

condition codes: exercise with overflow (1)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \cancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\mathtt{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathtt{SF} = 0$ (false)	not negative	rax <= rbx (if correct)
$\mathtt{OF} = 1$ (true)	overflow as signed	incorrect for signed

condition codes: exercise with overflow (1)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \cancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\texttt{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\texttt{SF} = 0$ (false)	not negative	rax <= rbx (if correct)
$\texttt{OF} = 1$ (true)	overflow as signed	incorrect for signed
$\texttt{CF} = 0$ (false)	no overflow as unsigned	correct for unsigned

recall: x86-64 general purpose registers

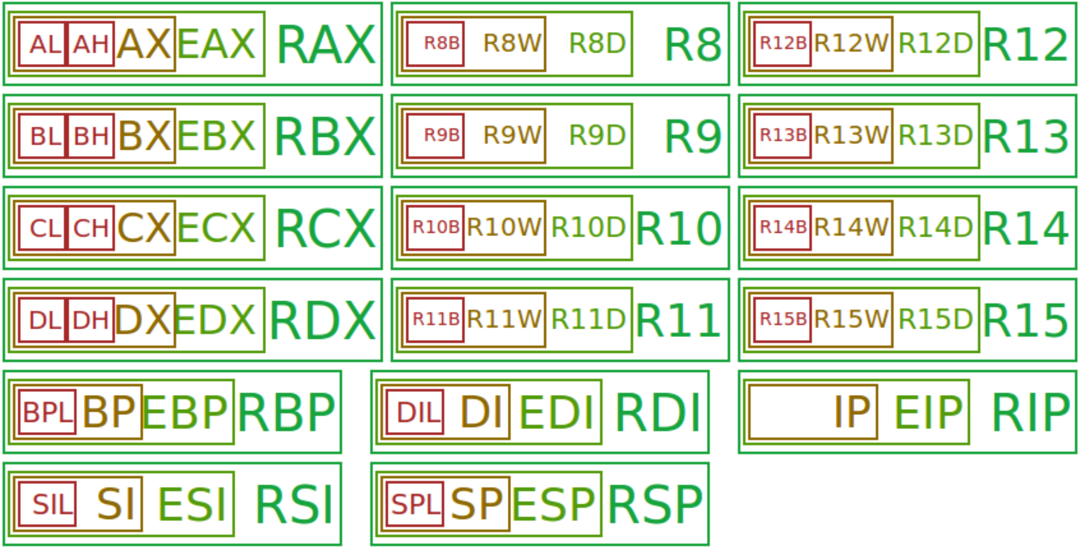

Figure: Immae via Wikipedia

authoriative source (1)

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D

authoriative source (2)

System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.99.7

Edited by

Michael Matz[1], Jan Hubička[2], Andreas Jaeger[3], Mark Mitche

November 17, 2014

question

```
pushq $0x1
pushq $0x2
addq $0x3, 8(%rsp)
popq %rax
popq %rbx
```

What is value of `%rax` and `%rbx` after this?

a. `%rax` = `0x2`, `%rbx` = `0x4`
b. `%rax` = `0x5`, `%rbx` = `0x1`
c. `%rax` = `0x2`, `%rbx` = `0x1`
d. the snippet has invalid syntax or will crash
e. more information is needed
f. something else?