assembly / ISAs

Changelog

Changes made in this version not seen in first lecture:

- 5 September 2017: slide 3: lea destination should have been rax, not rsp
- 5 September 2017: slide 10: signed result w/o truncation is $-2^{64}+1$, not $-2^{64}-1$
- 5 September 2017: slide 12: changed version B and C to use jge
- 5 September 2017: slide 14: fix bugs in third version: compare to 10 (not 9), end with %rbx set
- 8 September 2017: slide 11: use %rax, not %rbx

last time

AT&T syntax

```
movq (%rax,%rcx,4), %rbx =
mov RBX, QWORD PTR [RAX + RCX * 4]
```

C to assembly

- strategy — write with gotos first

condition codes

- set by arithmetic instructions + cmp or test
- used by conditional jump
- names based on subtraction (cmp)
- result = 0: equal; result positive: greater; result negative: less than
- ...more detail today

the quiz: ASM

```
movq %rsp, %rax
    # %rax ← %rsp = X
pushq %rax
    # %rsp ← %rsp - 8 = X - 8
    # memory[%rsp] = %rax
subq %rsp, %rax
    # %rax ← %rax - %rsp = X - (X - 8) = 8
leaq (%rax, %rax, 2), %rax
    # %rax ← %rax + %rax * 2 = 8 + 8 * 2 = 24
```

upcoming labs

this week: pointer-heavy code in C

use tool to find malloc/free-related mistakes

fix broken circular doubly-linked list implementation

next week: in-lab quiz

implement library functions strlen/strsep

no notes

last time

AT&T syntax

- `movq (%rax,%rcx,4), %rbx` = `mov RBX, QWORD PTR [RAX + RCX * 4]`

C to assembly

- strategy — write with gotos first

condition codes

- set by arithmetic instructions + cmp or test
- used by conditional jump
- names based on subtraction (cmp)
- result = 0: equal; result positive: greater; result negative: less than
- ...more detail today

condition codes and jumps

jg, **jle**, etc

named based on interpreting result of subtraction

zero: equal

negative: less than

positive: greater than

condition codes: closer look

x86 condition codes:

- `ZF` ("zero flag") — was result zero? (sub/cmp: equal)
- `SF` ("sign flag") — was result negative? (sub/cmp: less)
- `CF` ("carry flag") — did computation overflow (as unsigned)?
- `OF` ("overflow flag") — did computation overflow (as signed)?
- (and one more we won't talk about)

GDB: part of "eflags" register

set by cmp, test, arithmetic

closer look: condition codes (1)

```
movq $−10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

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

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

closer look: condition codes (1)

```
movq $−10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

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

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

closer look: condition codes (1)

```
movq $−10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

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

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

closer look: condition codes (1)

```
movq $−10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

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

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

closer look: condition codes (1)

```
movq $−10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

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

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

exercise: condition codes (2)

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

$\texttt{ZF} =$?
$\texttt{SF} =$?
$\texttt{OF} =$?
$\texttt{CF} =$?

closer look: condition codes (2)

```
// 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

closer look: condition codes (2)

```
// 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

closer look: condition codes (2)

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

as signed: $-2^{63} - (2^{63} - 1) =$ ~~$-2^{64}+1$~~ 1 (overflow)

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

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

closer look: condition codes (2)

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

as signed: $-2^{63} - (2^{63} - 1) =$ ~~$-2^{64}+1$~~ 1 (overflow)

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

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

closer look: condition codes (2)

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

as signed: $-2^{63} - (2^{63} - 1) =$ ~~$-2^{64}+1$~~ 1 (overflow)

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

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

closer look: condition codes (3)

```
  movq  $−1, %rax
  addq  $−2, %rax
// result = -3
```

as signed: $-1 + (-2) = -3$

as unsigned: $(2^{64} - 1) + (2^{64} - 2) =$ ~~$2^{65} - 3$~~ $2^{64} - 3$ (overflow)

ZF $= 0$ (false)	not zero	result not zero

closer look: condition codes (3)

```
  movq  $-1, %rax
  addq  $-2, %rax
// result = -3
```

as signed: $-1 + (-2) = -3$

as unsigned: $(2^{64} - 1) + (2^{64} - 2) = \cancel{2^{65} - 3}\ 2^{64} - 3$ (overflow)

ZF $= 0$ (false)	not zero	result not zero
SF $= 1$ (true)	negative	result is negative
OF $= 0$ (false)	no overflow as signed	correct for signed
CF $= 1$ (true)	overflow as unsigned	incorrect for unsigned

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 to assembly (1)

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


while to assembly (1)

```
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 (3)

```
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(,%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
    ...
```


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
```


preview: our Y86 condition codes

ZF (zero flag), SF (sign flag)

just won't handle overflow/underflow

microarchitecture v. instruction set

microarchitecture — design of the hardware
- "generations" of Intel's x86 chips
- different microarchitectures for very low-power versus laptop/desktop
- changes in performance/efficiency

instruction set — interface visible by software
- what matters for software compatibility
- many ways to implement (but some might be easier)

selected instruction set design concerns

ease of creating efficient assembly/machine code

ease of designing efficient/cheap/low power/…hardware

flexibility for the future

ISAs being manufactured today

x86 — dominant in desktops, servers

ARM — dominant in mobile devices

POWER — Wii U, IBM supercomputers and some servers

MIPS — common in consumer wifi access points

SPARC — some Oracle servers, Fujitsu supercomputers

z/Architecture — IBM mainframes

Z80 — TI calculators

SHARC — some digital signal processors

Itanium — some HP servers (being retired)

RISC V — some embedded

…

ISA variation

instruction set	instr. length	# normal registers	*approx.* # instrs.
x86-64	1–15 byte	16	1500
Y86-64	1–10 byte	15	18
ARMv7	4 byte*	16	400
POWER8	4 byte	32	1400
MIPS32	4 byte	31	200
Itanium	41 bits*	128	300
Z80	1–4 byte	7	40
VAX	1–14 byte	8	150
z/Architecture	2–6 byte	16	1000
RISC V	4 byte*	31	500*

other choices: condition codes?

instead of:

```
cmpq %r11, %r12
je somewhere
```

could do:

```
/* _B_ranch if _EQ_ual */
beq  %r11, %r12, somewhere
```


other choices: addressing modes

ways of specifying operands. examples:

x86-64: `10(%r11,%r12,4)`

ARM: `%r11 << 3` (shift register value by constant)

VAX: `((%r11))` (register value is pointer to pointer)

other choices: number of operands

`add src1, src2, dest`
ARM, POWER, MIPS, SPARC, …

`add src2, src1=dest`
x86, AVR, Z80, …

VAX: both

other choices: instruction complexity

instructions that write multiple values?
x86-64: **push**, **pop**, **movsb**, …

more?

CISC and RISC

RISC — Reduced Instruction Set Computer

reduced from what?

CISC and RISC

RISC — Reduced Instruction Set Computer

reduced from what?

CISC — Complex Instruction Set Computer

some VAX instructions

`MATCHC` *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

`POLY` *x*, *coefficientsLen*, *coefficientsPtr*
Evaluate the polynomial whose coefficients are pointed to by *coefficientPtr* at the value *x*.

`EDITPC` *sourceLen*, *sourcePtr*, *patternLen*, *patternPtr*
Edit the string pointed to by *sourcePtr* using the pattern string specified by *patternPtr*.

microcode

`MATCHC` *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

loop in hardware???

typically: lookup sequence of microinstructions (“microcode”)

secret simpler instruction set

Why RISC?

complex instructions were usually not faster

complex instructions were harder to implement

compilers, not hand-written assembly

typical RISC ISA properties

fewer, simpler instructions

seperate instructions to access memory

fixed-length instructions

more registers

no "loops" within single instructions

no instructions with two memory operands

few addressing modes

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64: movq

source destination

*SD*movq

i — immediate
r — register
m — memory

Y86-64: movq

source destination

*SD*movq

i — immediate
r — register
m — memory

irmovq	~~immovq~~	~~iimovq~~
rrmovq	rmmovq	~~rimovq~~
mrmovq	~~mmmovq~~	~~mimovq~~

Y86-64: movq

source destination

*SD*movq

i — immediate
r — register
m — memory

irmovq	~~immovq~~
rrmovq	rmmovq
mrmovq	~~mmmovq~~

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

addq	jmp	pushq
subq	j*CC*	popq
andq	cmov*CC*	movq (renamed)
xorq	call	hlt (renamed)
nop	ret	

much, much simpler encoding

cmovCC

conditional move

exist on x86-64 (but you probably didn't see them)

Y86-64: register-to-register only

instead of:

```
    jle skip_move
    rrmovq %rax, %rbx
skip_move:
    // ...
```

can do:

```
    cmovg %rax, %rbx
```


Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`jCC`	`popq`
`andq`	`cmovCC`	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

halt

(x86-64 instruction called `hlt`)

Y86-64 instruction `halt`

stops the processor

- otherwise — something's in memory "after" program!

real processors: reserved for OS

Y86-64: specifying addresses

Valid: `rmmovq %r11, 10(%r12)`

Y86-64: specifying addresses

Valid: `rmmovq %r11, 10(%r12)`

Invalid: ~~`rmmovq %r11, 10(%r12,%r13)`~~

Invalid: ~~`rmmovq %r11, 10(,%r12,4)`~~

Invalid: ~~`rmmovq %r11, 10(%r12,%r13,4)`~~

Y86-64: accessing memory (1)

r12 ← memory[10 + r11] + r12

Invalid: ~~`addq 10(%r11), %r12`~~

Y86-64: accessing memory (1)

r12 ← memory[10 + r11] + r12

Invalid: ~~`addq 10(%r11), %r12`~~

Instead:

```
mrmovq 10(%r11), %r11
/* overwrites %r11 */

addq %r11, %r12
```


Y86-64: accessing memory (2)

r12 ← memory[10 + 8 * r11] + r12

Invalid:~~`addq 10(,%r11,8), %r12`~~

Y86-64: accessing memory (2)

r12 ← memory[10 + 8 * r11] + r12

Invalid: ~~`addq 10(,%r11,8), %r12`~~

Instead:

```
/* replace %r11 with 8*%r11 */
addq %r11, %r11
addq %r11, %r11
addq %r11, %r11

mrmovq 10(%r11), %r11
addq %r11, %r12
```


Y86-64 constants (1)

```
irmovq $100, %r11
```

only instruction with non-address constant operand

Y86-64 constants (2)

r12 ← r12 + 1

Invalid: ~~`addq $1, %r12`~~

Y86-64 constants (2)

r12 ← r12 + 1

Invalid: ~~`addq $1, %r12`~~

Instead, need an extra register:

```
irmovq $1, %r11
addq %r11, %r12
```

Y86-64: operand uniqueness

only one kind of value for each operand

instruction name tells you the kind

(why **movq** was 'split' into four names)

Y86-64: condition codes

ZF — value was zero?

SF — sign bit was set? i.e. value was negative?

this course: no OF, CF (to simplify assignments)

set by **addq**, **subq**, **andq**, **xorq**

not set by anything else

Y86-64: using condition codes

subq SECOND, FIRST (value = FIRST - SECOND)

j__ or cmov__	condition code bit test	value test
le	$\mathrm{SF} = 1$ or $\mathrm{ZF} = 1$	value ≤ 0
l	$\mathrm{SF} = 1$	value < 0
e	$\mathrm{ZF} = 1$	value $= 0$
ne	$\mathrm{ZF} = 0$	value $\neq 0$
ge	$\mathrm{SF} = 0$	value ≥ 0
g	$\mathrm{SF} = 0$ and $\mathrm{ZF} = 0$	value > 0

missing OF (overflow flag); CF (carry flag)

Y86-64: conditionals (1)

~~cmp~~, ~~test~~

Y86-64: conditionals (1)

~~cmp~~, ~~test~~

instead: use side effect of normal arithmetic

Y86-64: conditionals (1)

~~cmp~~, ~~test~~

instead: use side effect of normal arithmetic

instead of

```
cmpq %r11, %r12
jle somewhere
```

maybe:

```
subq %r11, %r12
jle
```

(but changes %r12)

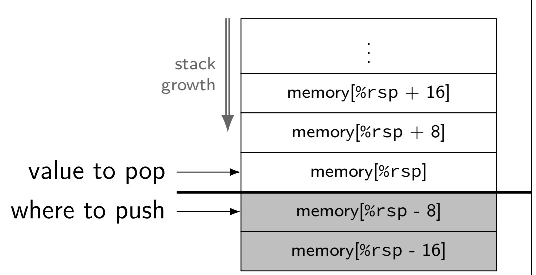
push/pop

pushq %rbx

%rsp ← %rsp − 8
memory[%rsp] ← %rbx

popq %rbx

%rbx ← memory[%rsp]
%rsp ← %rsp + 8



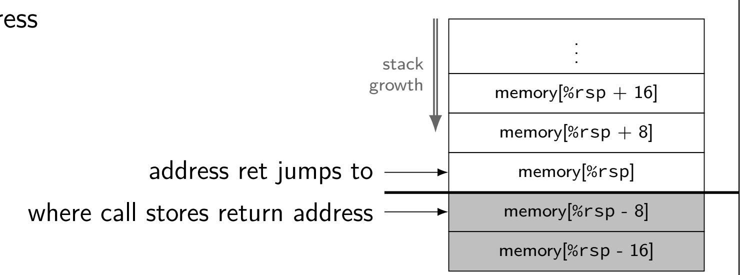
call/ret

call LABEL

push PC (next instruction address) on stack
jmp to LABEL address

ret

pop address from stack
jmp to that address



Y86-64 state

%r*XX* — 15 registers

- ~~%r15~~ missing
- smaller parts of registers missing

ZF (zero), SF (sign), ~~OF (overflow)~~

- book has OF, we'll not use it
- ~~CF~~ (carry) missing

Stat — processor status — halted?

PC — program counter (AKA instruction pointer)

main memory

typical RISC ISA properties

fewer, simpler instructions

seperate instructions to access memory

~~fixed-length instructions~~

more registers

no "loops" within single instructions

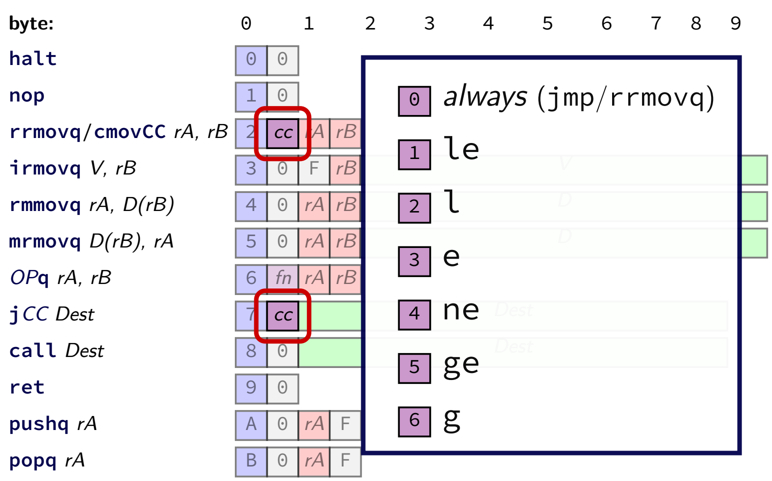
no instructions with two memory operands

few addressing modes

Y86-64 instruction formats

byte:	0		1		2 3 4 5 6 7 8 9
`halt`	0	0			
`nop`	1	0			
`rrmovq/cmovCC` *rA, rB*	2	*cc*	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*	
`j`*CC Dest*	7	*cc*	*Dest*		
`call` *Dest*	8	0	*Dest*		
`ret`	9	0			
`pushq` *rA*	A	0	*rA*	F	
`popq` *rA*	B	0	*rA*	F	

Secondary opcodes: `cmov`*cc*/`j`*cc*



Secondary opcodes: *OP*q

byte:	0	1	2 3 4 5 6 7 8 9
halt	0 0		
nop	1 0		
rrmovq/cmovCC *rA*, *rB*	2 *cc*	*rA* *rB*	
irmovq *V*, *rB*	3 0	F *rB*	*V*
rmmovq *rA*, *D*(*rB*)	4 0	*rA* *rB*	*D*
mrmovq *D*(*rB*), *rA*	5 0	*rA* *rB*	*D*
*OP*q *rA*, *rB*	6 *fn*	*rA* *rB*	
j*CC* *Dest*	7 *cc*	*Dest*	
call *Dest*	8 0	*Dest*	
ret	9 0		
pushq *rA*	A 0	*rA* F	
popq *rA*	B 0	*rA* F	

0	add
1	sub
2	and
3	xor

Registers: *rA*, *rB*

byte:	0	1	2 3 4 5 6 7 8 9
halt	0 0		
nop	1 0		
rrmovq/cmovCC *rA*, *rB*	2 *cc*	*rA* *rB*	
irmovq *V*, *rB*	3 0	F *rB*	*V*
rmmovq *rA*, *D*(*rB*)	4 0	*rA* *rB*	*D*
mrmovq *D*(*rB*), *rA*	5 0	*rA* *rB*	*D*
*OP*q *rA*, *rB*	6 *fn*	*rA* *rB*	
j*CC* *Dest*	7 *cc*	*Dest*	
call *Dest*	8 0	*Dest*	
ret	9 0		
pushq *rA*	A 0	*rA* F	
popq *rA*	B 0	*rA* F	

0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	*none*

Immediates: *V*, *D*, *Dest*

byte:	0	1	2 3 4 5 6 7 8 9
halt	0 0		
nop	1 0		
rrmovq/cmovCC *rA*, *rB*	2 *cc*	*rA* *rB*	
irmovq *V*, *rB*	3 0	F *rB*	*V*
rmmovq *rA*, *D*(*rB*)	4 0	*rA* *rB*	*D*
mrmovq *D*(*rB*), *rA*	5 0	*rA* *rB*	*D*
*OP*q *rA*, *rB*	6 *fn*	*rA* *rB*	
j*CC* *Dest*	7 *cc*	*Dest*	
call *Dest*	8 0	*Dest*	
ret	9 0		
pushq *rA*	A 0	*rA* F	
popq *rA*	B 0	*rA* F	

Immediates: *V*, *D*, *Dest*

byte:	0	1	2 3 4 5 6 7 8 9
halt	0 0		
nop	1 0		
rrmovq/cmovCC *rA*, *rB*	2 *cc*	*rA* *rB*	
irmovq *V*, *rB*	3 0	F *rB*	*V*
rmmovq *rA*, *D*(*rB*)	4 0	*rA* *rB*	*D*
mrmovq *D*(*rB*), *rA*	5 0	*rA* *rB*	*D*
*OP*q *rA*, *rB*	6 *fn*	*rA* *rB*	
j*CC* *Dest*	7 *cc*	*Dest*	
call *Dest*	8 0	*Dest*	
ret	9 0		
pushq *rA*	A 0	*rA* F	
popq *rA*	B 0	*rA* F	

Y86-64 encoding (1)

```
long addOne(long x) {
    return x + 1;
}
```

x86-64:

```
movq %rdi, %rax
addq $1, %rax
ret
```

Y86-64:

Y86-64 encoding (1)

```
long addOne(long x) {
    return x + 1;
}
```

x86-64:

```
movq %rdi, %rax
addq $1, %rax
ret
```


Y86-64:

```
irmovq  $1,   %rax
addq    %rdi, %rax
ret
```


Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```

★ 3 0 F %rax 01 00 00 00 00 00 00 00

Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```


★ 3 0 F 0 01 00 00 00 00 00 00 00

Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```

3 0 F 0 01 00 00 00 00 00 00 00
★ 6 add %rdi %rax

Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```

3 0 F 0 01 00 00 00 00 00 00 00
★ 6 0 7 0

Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```

3 0 F 0 01 00 00 00 00 00 00 00
6 0 7 0
★ 9 0

Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```

3 0 F 0 01 00 00 00 00 00 00 00
6 0 7 0
9 0

30 F0 01 00 00 00 00 00 00 00 60 70 90

Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```

---

6 | add | %rax | %rax

Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```

⋆ 6 | add | %rax | %rax

Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```

⋆ 6 | 0 | 0 | 0

Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```

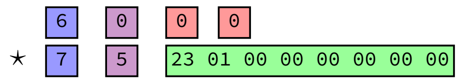
---

6 | 0 | 0 | 0

7 | 5 | 23 01 00 00 00 00 00 00

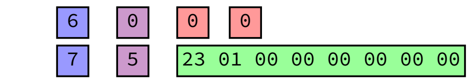
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



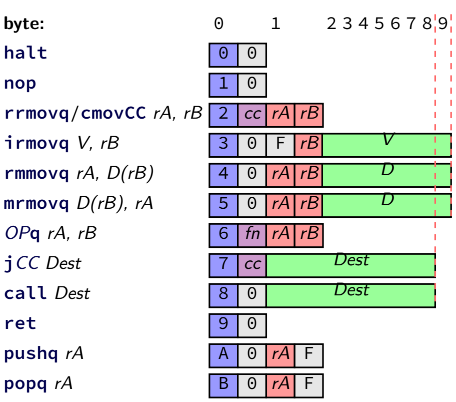
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



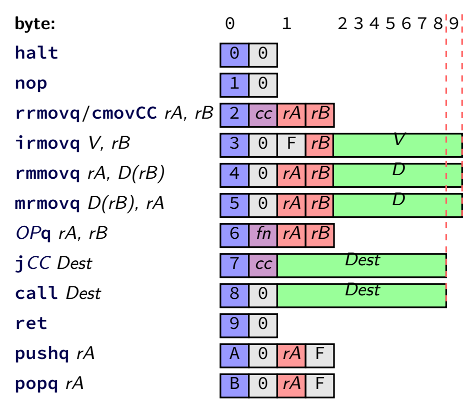
Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```



Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```



Y86-64 decoding

20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00

`rrmovq %rcx, %rax`

- 0 as cc: always
- 1 as reg: `%rcx`
- 0 as reg: `%rax`

byte:	0		1		2 3 4 5 6 7 8 9
`halt`	0	0			
`nop`	1	0			
`rrmovq/cmovCC` *rA, rB*	2	*cc*	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*	
`j`*CC Dest*	7	*cc*	*Dest*		
`call` *Dest*	8	0	*Dest*		
`ret`	9	0			
`pushq` *rA*	A	0	*rA*	F	
`popq` *rA*	B	0	*rA*	F	

Y86-64 decoding

20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00

```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
```

- 0 as fn: add
- 1 as fn: sub

byte:	0		1		2 3 4 5 6 7 8 9
`halt`	0	0			
`nop`	1	0			
`rrmovq/cmovCC` *rA, rB*	2	*cc*	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*	
`j`*CC Dest*	7	*cc*	*Dest*		
`call` *Dest*	8	0	*Dest*		
`ret`	9	0			
`pushq` *rA*	A	0	*rA*	F	
`popq` *rA*	B	0	*rA*	F	

Y86-64 decoding

20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00

```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
jl     0x84
```

- 2 as cc: `l` (less than)
- hex 84 00… as little endian *Dest*: *0x84*

byte:	0		1		2 3 4 5 6 7 8 9
`halt`	0	0			
`nop`	1	0			
`rrmovq/cmovCC` *rA, rB*	2	*cc*	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*	
`j`*CC Dest*	7	*cc*	*Dest*		
`call` *Dest*	8	0	*Dest*		
`ret`	9	0			
`pushq` *rA*	A	0	*rA*	F	
`popq` *rA*	B	0	*rA*	F	

Y86-64 decoding

20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00

```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
jl     0x84
rrmovq %rax, %rcx
jmp    0x68
```


byte:	0		1		2 3 4 5 6 7 8 9
`halt`	0	0			
`nop`	1	0			
`rrmovq/cmovCC` *rA, rB*	2	*cc*	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*	
`j`*CC Dest*	7	*cc*	*Dest*		
`call` *Dest*	8	0	*Dest*		
`ret`	9	0			
`pushq` *rA*	A	0	*rA*	F	
`popq` *rA*	B	0	*rA*	F	

Y86-64: convenience for hardware

4 bits to decode instruction size/layout

(mostly) uniform placement of operands

jumping to zeroes (uninitialized?) by accident halts

no attempt to fit (parts of) multiple instructions in a byte

byte:	0		1		2 3 4 5 6 7 8 9
`halt`	0	0			
`nop`	1	0			
`rrmovq/cmovCC` *rA, rB*	2	*cc*	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*	
`j`*CC Dest*	7	*cc*	*Dest*		
`call` *Dest*	8	0	*Dest*		
`ret`	9	0			
`pushq` *rA*	A	0	*rA*	F	
`popq` *rA*	B	0	*rA*	F	

Y86-64

Y86-64: simplified, more RISC-y version of X86-64

minimal set of arithmetic

only movs touch memory

only jumps, calls, and movs take immediates

simple variable-length encoding

next time: implementing with circuits

backup slides

closer look: condition codes (3)

```
  movq  $-1, %rax
  testq %rax, %rax
// result = -1
```

ZF = 0 (false)	not zero	rax not zero
SF = 1 (true)	negative	rax is negative
OF = 0 (false)	no overflow as signed	operation can't overflow
CF = 0 (false)	no oveflow as unsigned	operation can't overflow

exercise: condition codes and jXX

$\texttt{ZF} = 0$ (false)
$\texttt{SF} = 1$ (true)
$\texttt{OF} = 0$ (false)
$\texttt{CF} = 1$ (true)

jle (signed less than equal) should do what?

ja (unsigned greater than) should do what?

logical operators

return 1 for true or 0 for false

```
( 1 && 1 ) == 1          ( 1 || 1 ) == 1
( 2 && 4 ) == 1          ( 2 || 4 ) == 1
( 1 && 0 ) == 0          ( 1 || 0 ) == 1
( 0 && 0 ) == 0          ( 0 || 0 ) == 0
(-1 && -2) == 1          (-1 || -2) == 1
("" && "") == 1          ("" || "") == 1
! 1       == 0
! 4       == 0
!-1       == 0
! 0       == 1
```


recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**false**	**true**
false	false	false
true	false	true

recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**false**	**true**
false	false	false
true	false	true

recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	false	true
false	false	false
true	false	true

recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	false	true
false	false	false
true	false	true

recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	false	true
false	false	false
true	false	true

&& to assembly

```
return foo() && bar();
```

&& to assembly

```
return foo() && bar();
```

```
    result = foo();
    if (result == 0) goto skip_bar;
    result = bar();
skip_bar:
    result = (result != 0);
```


&& to assembly

```
return foo() && bar();
```

```
  call   foo
  testl  %eax, %eax // result is %eax (return val)
  je     skip_bar   // if result == 0 (equal for cmp)...
  call   bar
  testl  %eax, %eax // result is %eax (return val)
skip_bar:
  setne  %al        // set %al (low 8 bits of %eax)
                    // to 1 if result != 0, to 0 otherwise
  movzbl %al, %eax  // add zeroes to rest of %eax
```


recall: short-circuit (||)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() || one());
    printf(">␣%d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	false	true
false	false	true
true	true	true

recall: short-circuit (||)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() || one());
    printf(">␣%d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	false	true
false	false	true
true	true	true

recall: short-circuit (||)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() || one());
    printf(">␣%d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	false	true
false	false	true
true	true	true

recall: short-circuit (||)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() || one());
    printf(">␣%d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	false	true
false	false	true
true	true	true

recall: short-circuit (||)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() || one());
    printf(">␣%d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	false	true
false	false	true
true	true	true

exercise

```
  movq $3, %rax
  movq $2, %rbx
start_loop:
  addq %rbx, %rbx
  cmpq $3, %rbx
  subq $1, %rax
  jg start_loop
```

What is the value of %rbx after this runs?

A. 2 D. 16
B. 4 E. 32
C. 8 F. something else

Y86-64: simple condition codes (1)

If `%r9` is -1 and `%r10` is 1:

```
subq %r10, %r9
```

r9 becomes $-1 - (1) = -2$.

SF $= 1$ (negative)
ZF $= 0$ (not zero)

```
andq %r10, %r10
```

r10 becomes 1

SF $= 0$ (non-negative)
ZF $= 0$ (not zero)

the quiz: C (1)

```
int array[3]; int *ptr;
```

`sizeof(ptr) == sizeof(int *)` (pointer size; lab: 8)

`sizeof(array) = 3 * sizeof(int)` (lab: $3 \times 4 = 12$)

the quiz: C (2)

```
typedef struct foo { ... } bar;
    struct foo a_variable_of_this_type;
```

common for, e.g., linked list

```
    bar a_variable_of_this_type;
```

wrong type: ~~`struct bar a_variable`~~
wrong type: ~~`foo a_variable`~~

```
typedef FIRST SECOND;
```

'`FIRST a_variable;`' same as '`SECOND a_variable;`'

exericse: || to assembly

```
if (foo() || bar()) quux();
---------------------------------
  call    foo
  cmpl    $0, %eax
  ____    skip_bar   // (1)
  call    bar
  cmpl    $0, %eax
skip_bar:
  ____    skip_quux  // (2)
  call    quux
skip_quux:
```

What belongs in the blanks?

A. **jg/jle**
B. **jne/jne**
C. **jne/je**
D. **je/jne**
E. something else
F. there's no instructions that make this work