Y86-64

last time

wasted work in bitwise operators
- distributive property
- taking advantage of bitwise doing multiple OR/etc. at once
- "fan in" tree pattern

instruction set versus microarchitecture

big ISA descisions
- # of instructions? # of registers? variable v fixed encoding?
- 2 or 3 operand? condition codes?

RISC v CISC philosophies
- CISC: choose what's convenient for assembly writers
- RISC: expose hardware capabilities for compiler

typical RISC properties
- fewer, simpler instructions
- separate memory load/store instructions

quiz Q1

```
(((x & 0xFF0F) >> 2) & 0xFF0)
(((abcd efgh ijkl mnop & 0xFF0F) >> 2) & 0xFF0)
((abcd efgh 0000 mnop >> 2) & 0xFF0)
(00ab cdef gh00 00mn & 0xFF0)
0000 cdef gh00 0000
```

```
(((x & 0xFF0) >> 2) & 0xFF0F)
(((abcd efgh ijkl mnop & 0xFF0) >> 2) & 0xFF0F)
((0000 efgh ijkl 0000 >> 2) & 0xFF0F)
(0000 00ef ghij kl00 & 0xFF0F)
0000 00ef 0000 kl00
```

```
0000 cdef gh00 0000
0000 00ef 0000 kl00
```

---

abcd efgh ijkl mnop $\rightarrow$ 0011 0011 0011 0000

quiz Q1 [alt]

```
(((x & 0xFF0F) >> 2) & 0xFF0)
(((???? abcd efgh ijkl mnop & 0xFF0F) >> 2) & 0xFF0
((0000 abcd efgh 0000 mnop >> 2) & 0xFF0)
(0000 00ab cdef gh00 00mn & 0xFF0)
0000 0000 cdef gh00 0000
```

```
(((x & 0xFF0) >> 2) & 0xFF0F)
(((???? abcd efgh ijkl mnop & 0xFF0) >> 2) & 0xFF0
((0000 0000 efgh ijkl 0000 >> 2) & 0xFF0F)
(0000 0000 00ef ghij kl00 & 0xFF0F)
0000 0000 00ef 0000 kl00
```

```
0000 0000 cdef gh00 0000
0000 0000 00ef 0000 kl00
```

---

abcd efgh ijkl mnop $\rightarrow$ 0011 0011 0011 0000

quiz Q2+3

```
unsigned long swapAndFlip(unsigned long x) {
    unsigned long low = x & 0xFF; // least significant 8 bits of x
    unsigned long high = x >> 56; // most significant 8 bits of x
    // low with least/most significant bit of byte flipped
    // XOR with mask with most+least sig bits set to flip them
    unsigned long low_flip = low ^ 0x81; // 1000 0001
    // relocate low to most significant byte
    unsigned long low_flip_as_high = low_flip << 56;
    // everything but least/most significant byte
    // mask has every bit set but those in LSByte/MSByte
    unsigned long other_bits_of_x = x & 0x00FFFFFFFFFFFF00;
    return (
        low_flip_as_high |
        high |
        other_bits_of_x
    );
}
```

quiz Q4

instruction that finds the length of a string given a starting memory location

- arbitrary number of memory accesses (typically memory accesses are separate instructions on RISC)
- requires something like loop

✓ instruction that finds the index of the first zero byte in an 8-byte value stored in a register

- funny arithmetic on a single register; can be implemented with no loop (8 checks)

quiz Q5

adding a special "increment register by 1" instruction which needs fewer bytes of machine code than an equivalent "add constant to register" instruction

- means we'll have more instruction since this must be in addition to normal adds
- not likely to be more efficient to run in hardware than normal add
- program size usually not major concern of RISC

✓ providing a "add constant to register" instruction whose machine code is always the same length, no matter how large the constant is

- simplifies hardware by keeping instruction size constant

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

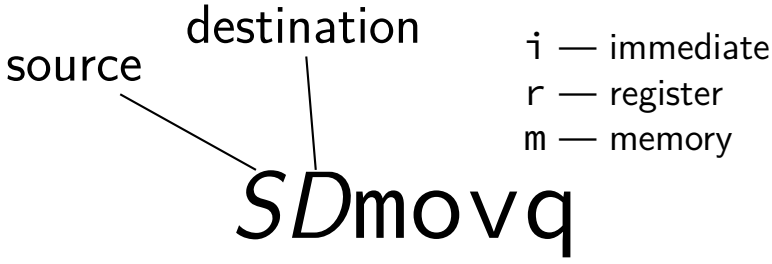
omits most of the 1000+ instructions

leaves

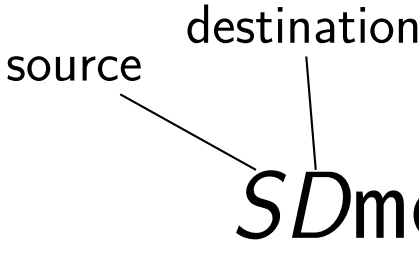
`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64: movq



Y86-64: movq



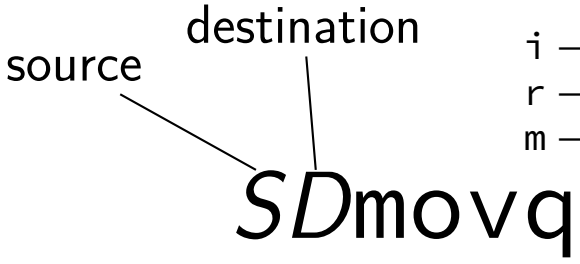
i — immediate
r — register
m — memory

irmovq ~~immovq~~ ~~iimovq~~

rrmovq rmmovq ~~rimovq~~

mrmovq ~~mmmovq~~ ~~mimovq~~

Y86-64: movq



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
```

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: exercise

r12 ← memory[10 + r11] + r12

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

How to simulate *assuming overwriting %r11 is okay*?

A.
```
rmmovq %r11, 10(%r11)
addq %r11, %r12
```

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

C.
```
mrmovq 10(%r11), %r11
addq %r11, %r12
rmmovq %r12, 10(%r11)
```

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

Y86-64: accessing memory (1)

r12 $\leftarrow$ memory[10 + r11] + r12

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

Y86-64: accessing memory (1)

r12 $\leftarrow$ memory[10 + r11] + r12

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

Instead:

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

addq %r11, %r12
```

Y86-64 constants (1)

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

only instruction with non-address constant operand

Y86-64 constants (2)

r12 $\leftarrow$ r12 + 1

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

Y86-64 constants (2)

r12 $\leftarrow$ 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)

push/pop

`pushq %rbx`

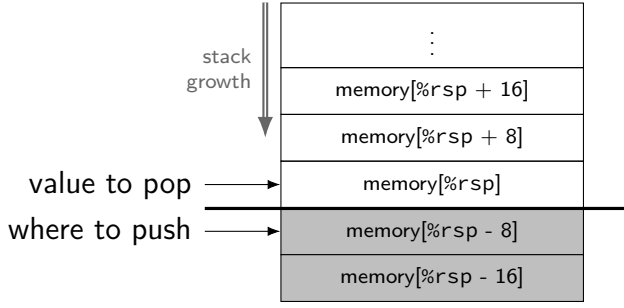
`%rsp` $\leftarrow$ `%rsp` $- 8$

memory[`%rsp`] $\leftarrow$ `%rbx`

`popq %rbx`

`%rbx` $\leftarrow$ memory[`%rsp`]

`%rsp` $\leftarrow$ `%rsp` $+ 8$



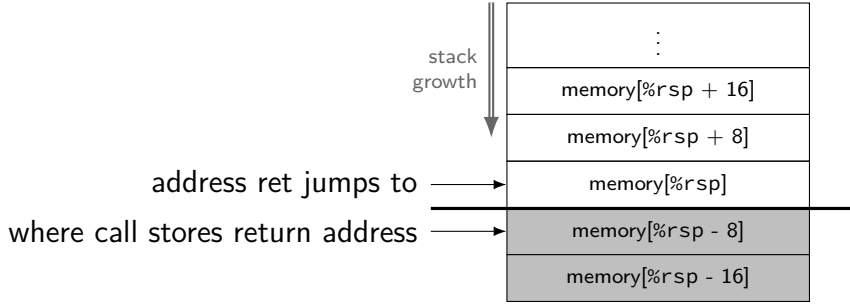
call/ret

`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
- no cmp, use sub, etc. instead
- ~~CF~~ (carry) missing

Stat — processor status — halted?

PC — program counter (AKA instruction pointer)

main memory

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
- no `cmp`, use `sub`, etc. instead
- ~~CF~~ (carry) missing

`Stat` — processor status — halted?

PC — program counter (AKA instruction pointer)

main memory

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
- no cmp, use sub, etc. instead
- ~~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~~

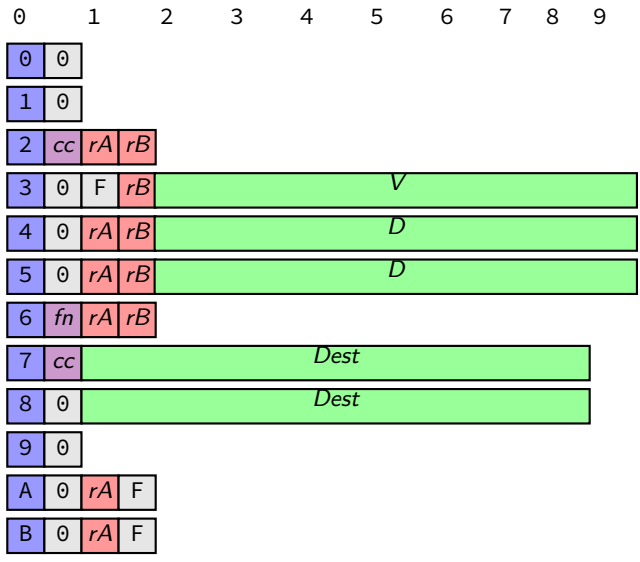
more registers

no “loops” within single instructions

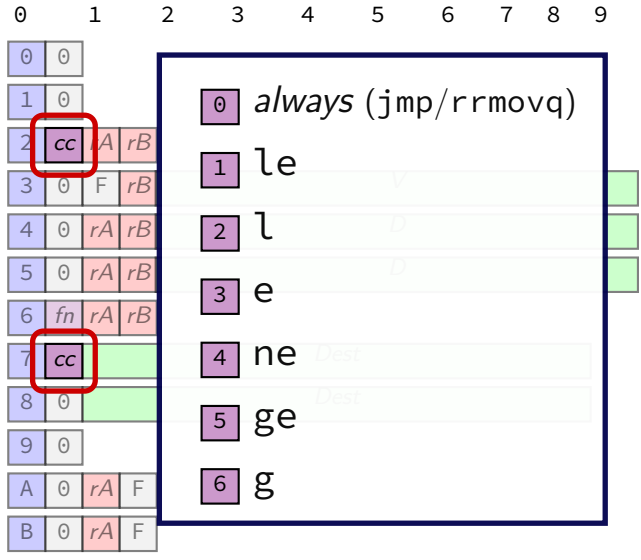
no instructions with two memory operands

few addressing modes

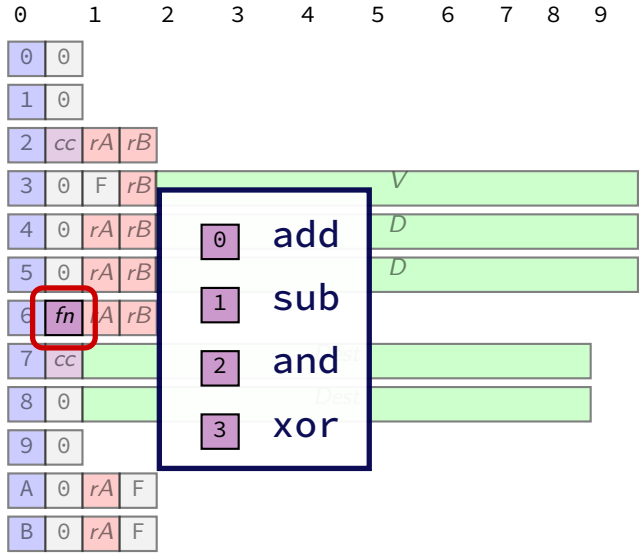
Y86-64 instruction formats



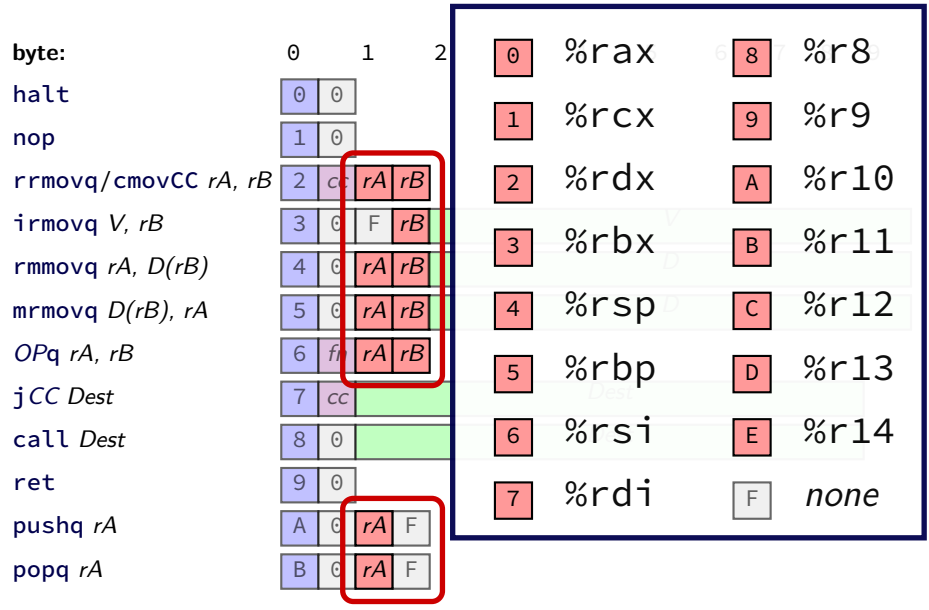
Secondary opcodes: `cmovcc`/`jcc`



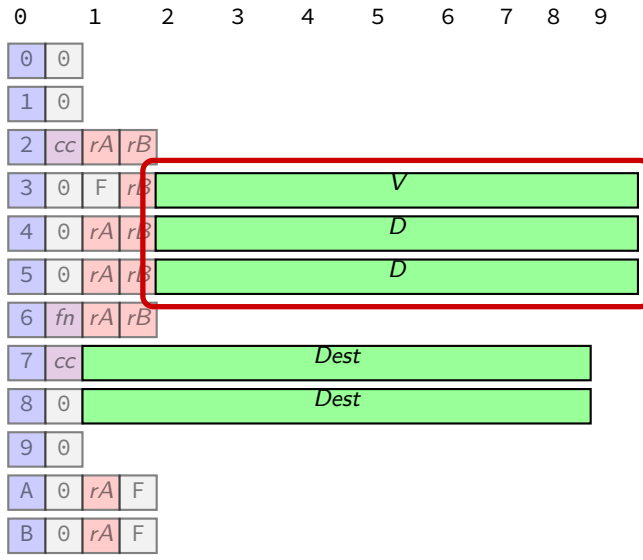
Secondary opcodes: *OP*q



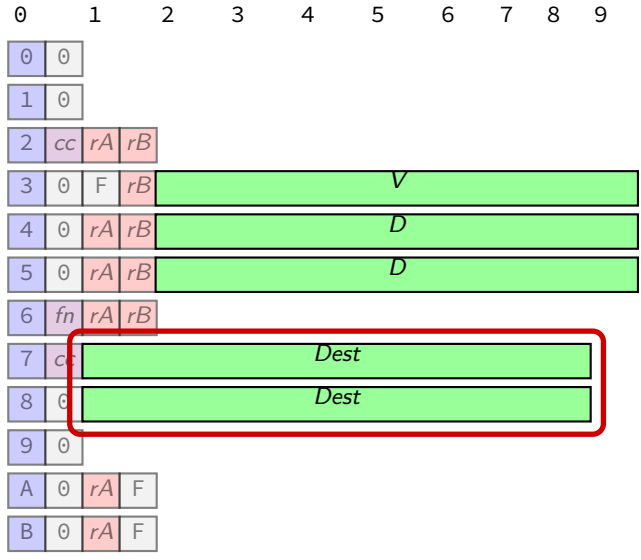
Registers: *rA, rB*



Immediates: *V, D, Dest*



Immediates: *V, D, Dest*



using YAS

download HCLRS (we'll use it later)

extract the archive

run `make`

example.ys

example.ys:

```
    irmovq $100, %rax
    irmovq $1, %rcx
    irmovq $10, %rdx
loop:
    subq %rdx, %rax
    subq %rcx, %rdx
    jg loop
    halt
```

example.yo

run `tools/yas example.ys`

example.yo:

```
0x000: 30f06400000000000000 |     irmovq $100, %rax
0x00a: 30f10100000000000000 |     irmovq $1, %rcx
0x014: 30f20a00000000000000 |     irmovq $10, %rdx
0x01e:                      | loop:
0x01e: 6120                 |     subq %rdx, %rax
0x020: 6112                 |     subq %rcx, %rdx
0x022: 761e00000000000000   |     jg loop
0x02b: 00                   |     halt
```

Y86-64 encoding (1)

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

x86-64:

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

Y86-64 translation?

A

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

B

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

C

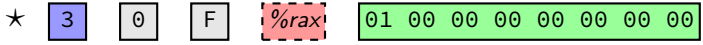
```
irmovq $1, %rax
addq %rdi, %rax
ret
```

D

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

Y86-64 encoding (2)

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



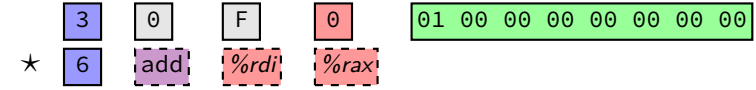
Y86-64 encoding (2)

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

★ `3` `0` `F` `0` `01 00 00 00 00 00 00 00`

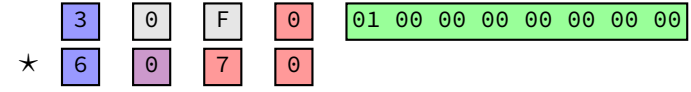
Y86-64 encoding (2)

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



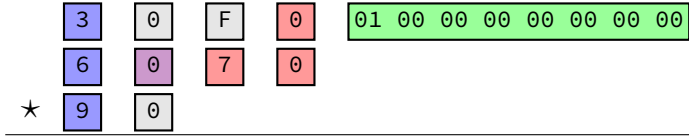
Y86-64 encoding (2)

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



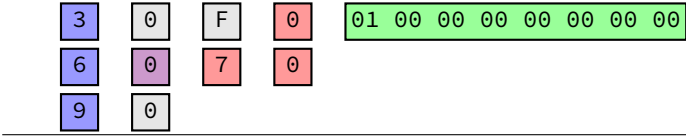
Y86-64 encoding (2)

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



Y86-64 encoding (2)

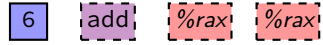
```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```




```
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
```



Y86-64 encoding (3)

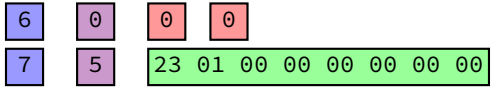
```
doubleTillNegative:
/* suppose at address 0x123 */
  addq     %rax, %rax
  jge doubleTillNegative
```

Y86-64 encoding (3)

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

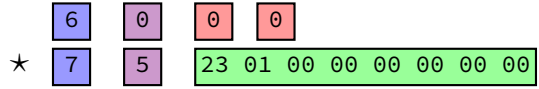
Y86-64 encoding (3)

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



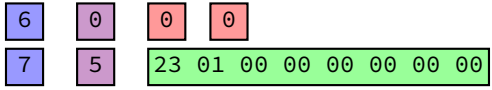
Y86-64 encoding (3)

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



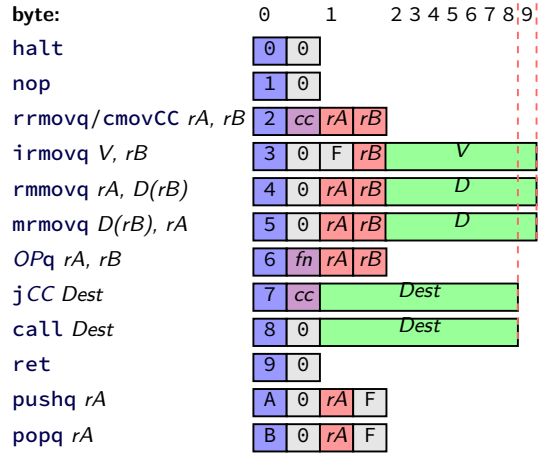
Y86-64 encoding (3)

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



Y86-64 decoding

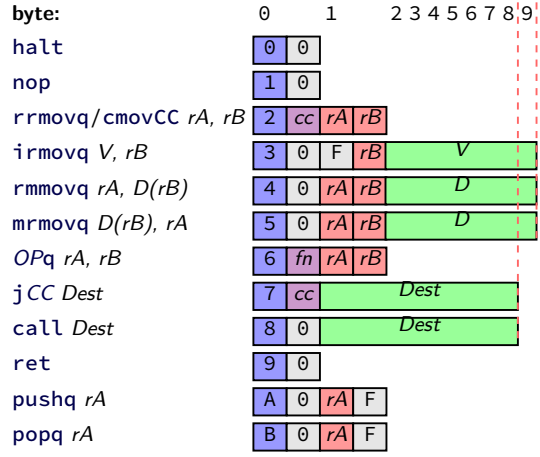
```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00 00
```



Y86-64 decoding

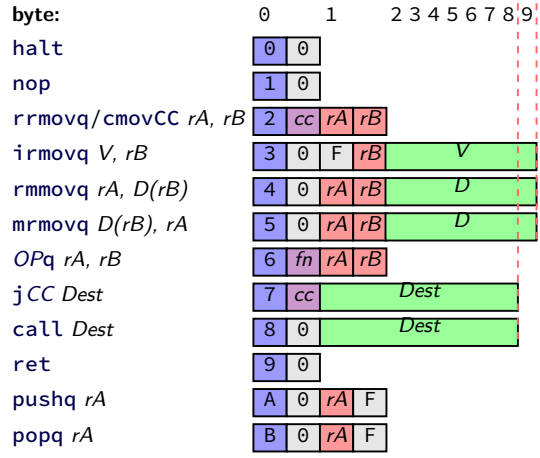
```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

exercise: types of first three instructions?



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
```

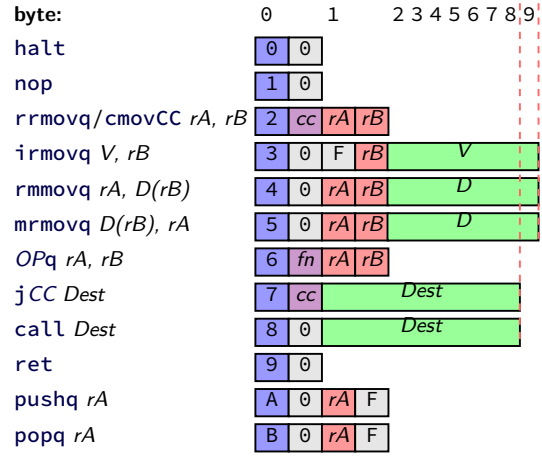


Y86-64 decoding

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

`rrmovq %rcx, %rax`

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

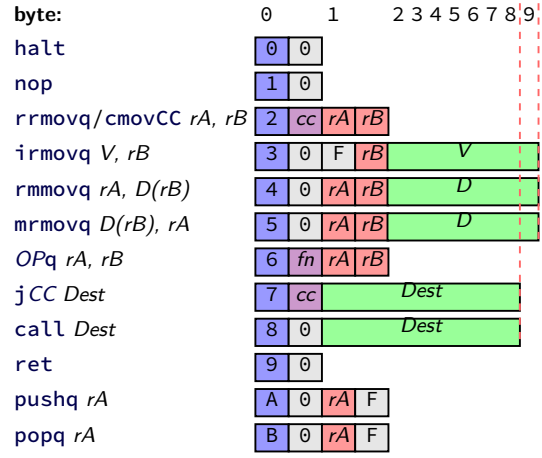


Y86-64 decoding

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

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

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

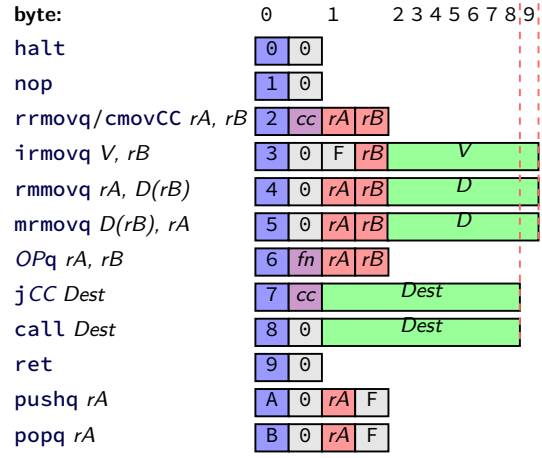


Y86-64 decoding

```
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
```

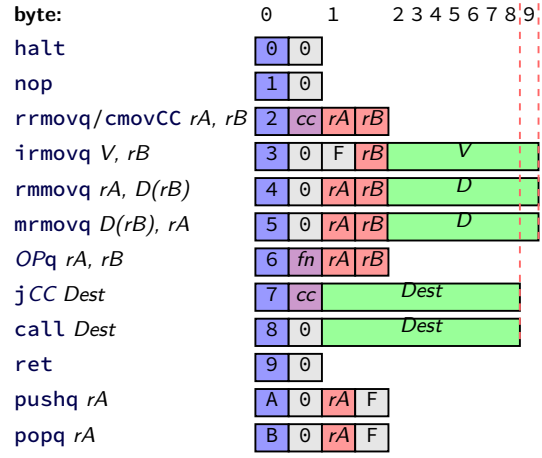
- 2 as cc: `l` (less than)
- hex 84 00… as little endian *Dest: 0x84*



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
```

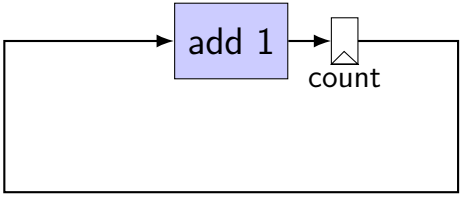
```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
jl     0x84
rrmovq %rcx, %rdx
rrmovq %rax, %rcx
jmp    0x68
```



describing hardware

how do we describe hardware?

pictures?



circuits with pictures?

yes, something you can do

such commercial tools exist, but...

not commonly used for processors

hardware description language

programming language for hardware

(typically) text-based representation of circuit

often abstracts away details like:

- how to build arithmetic operations from gates
- how to build registers from transistors
- how to build memories from transistors
- how to build MUXes from gates
- ...

those details also not a topic in this course

our tool: HCLRS

built for this course

assumes you're making a processor

somewhat different from textbook's HCL

backup slides