Y86-64

last time

bitwise puzzle strategies

- divide-and-conquer
- using bitwise operations to compute multiple things at a time

instruction set architecture versus microarchitecture

- instruction set: interface between hardware and software
- what machine code means...

started: idea of RISC (reduced instruction set architecture) v CISC

- "classic" CISC: designed for assembly programmers, sometimes with instructions requiring many steps in hardware

Why RISC?

complex instructions were usually not faster

- (even though programs with simple instructions were bigger)

complex instructions were harder to implement

compilers were replacing hand-written assembly

- correct assumption: almost no one will write assembly anymore
- incorrect assumption: okay to recompile frequently

ISAs: who does the work?

CISC-like (harder to make hardware, easier to use assembly)

choose instructions with particular assembly language in mind?
hardware designer provides operations assembly-writers wants

let the hardware worry about optimizing it?

RISC-like (easier to make hardware, harder to use assembly)

choose instructions with particular HW implementation in mind?
hardware designer exposes things it can do efficiently to assembly-writers

building blocks for compiler to make efficient programs?

note: *general* differences — no firm RISC v. CISC line

ISAs: who does the work?

CISC-like (harder to make hardware, easier to use assembly)

choose instructions with particular assembly language in mind?
hardware designer provides operations assembly-writers wants

let the hardware worry about optimizing it?

RISC-like (easier to make hardware, harder to use assembly)

choose instructions with particular HW implementation in mind?
hardware designer exposes things it can do efficiently to assembly-writers

building blocks for compiler to make efficient programs?

note: *general* differences — no firm RISC v. CISC line

ISAs: who does the work?

CISC-like (harder to make hardware, easier to use assembly)

choose instructions with particular assembly language in mind?
hardware designer provides operations assembly-writers wants

let the hardware worry about optimizing it?

RISC-like (easier to make hardware, harder to use assembly)

choose instructions with particular HW implementation in mind?
hardware designer exposes things it can do efficiently to assembly-writers

building blocks for compiler to make efficient programs?

note: *general* differences — no firm RISC v. CISC line

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

is CISC the winner?

well, can't get rid of x86 features

- backwards compatibility matters

more application-specific instructions

but...compilers tend to use more RISC-like subset of instructions

modern x86: often convert to RISC-like "microinstructions"

- sounds really expensive, but ...
- lots of instruction preprocessing used in 'fast' CPU designs (even for RISC ISAs)

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

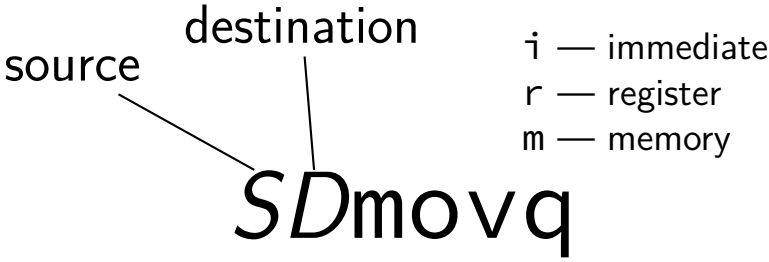
omits most of the 1000+ instructions

leaves

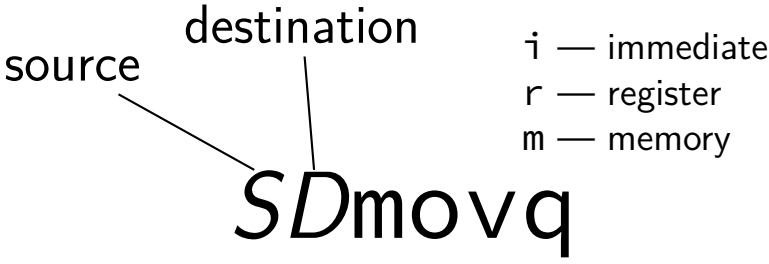
`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64: movq



Y86-64: movq

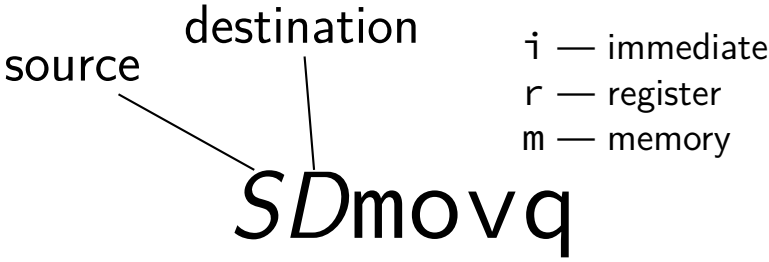


irmovq ~~immovq~~ ~~iimovq~~

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 $\leftarrow$ 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`

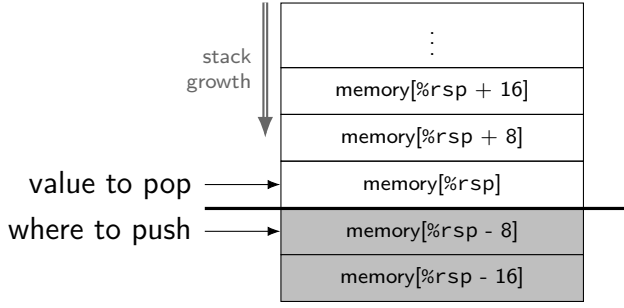
`%rsp` $\leftarrow$ `%rsp` $- 8$

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

`popq %rbx`

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

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



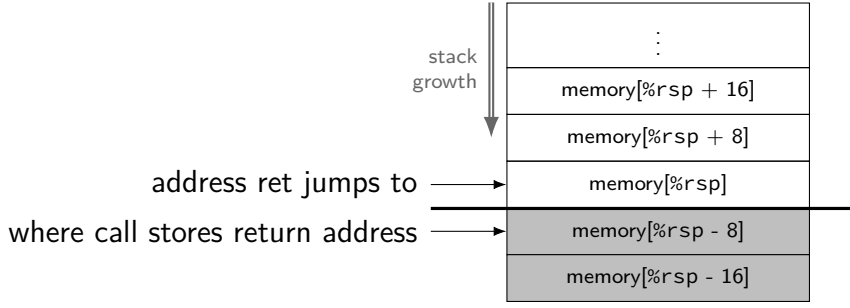
call/ret

`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~~

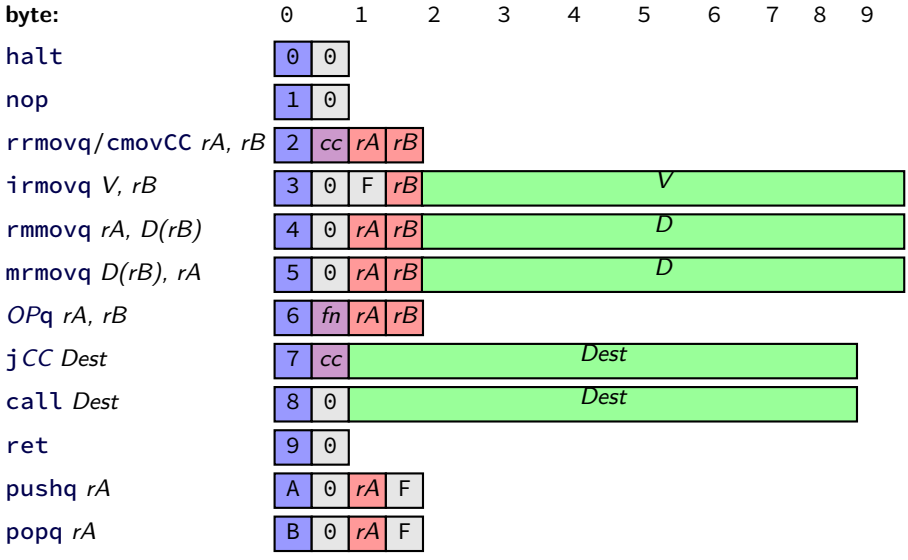
more registers

no “loops” within single instructions

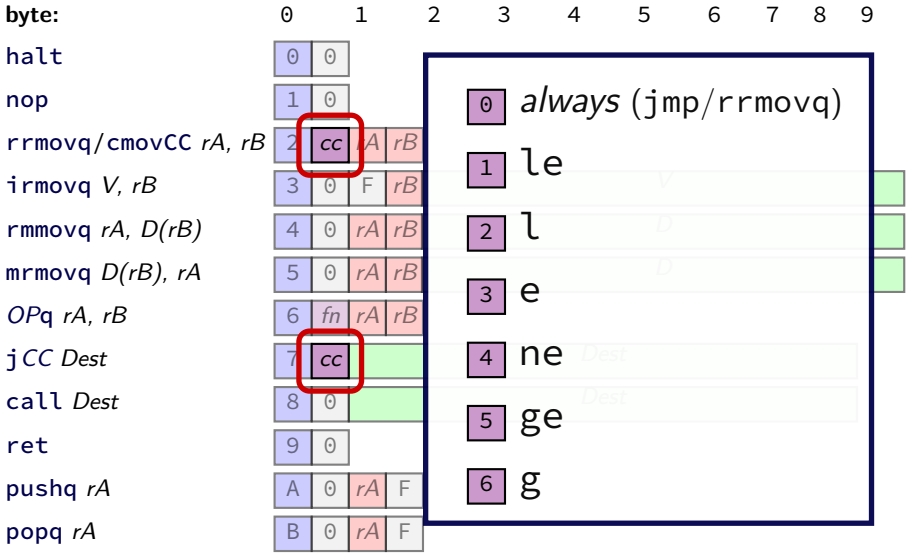
no instructions with two memory operands

few addressing modes

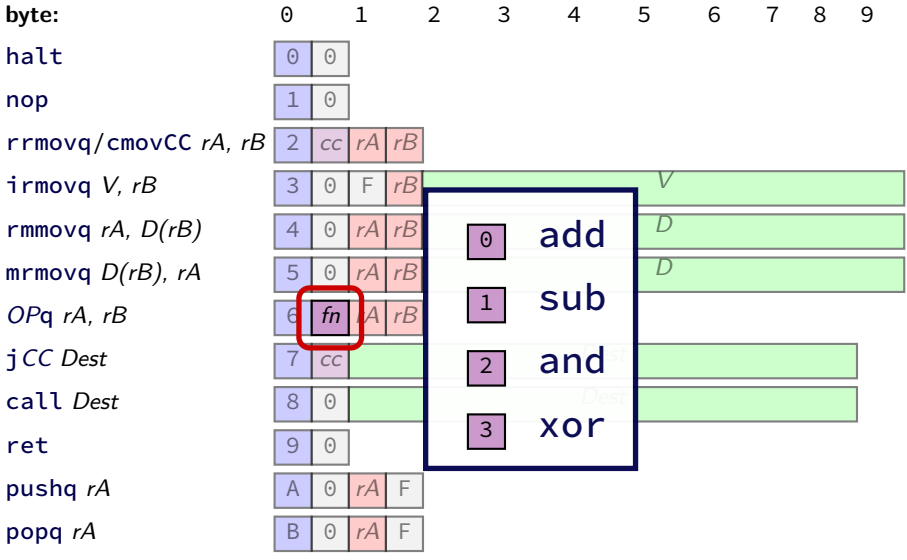
Y86-64 instruction formats



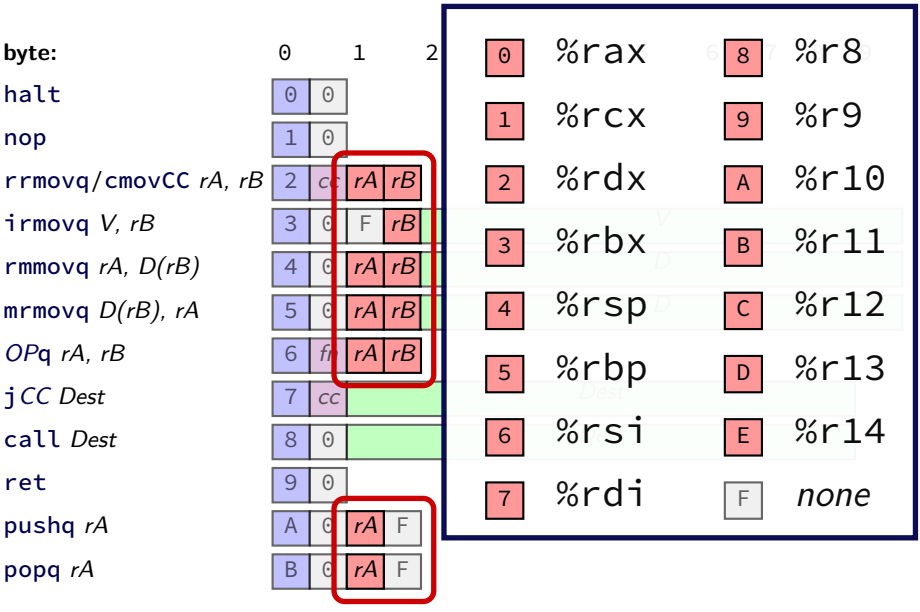
Secondary opcodes: `cmov`*cc*`/j`*cc*



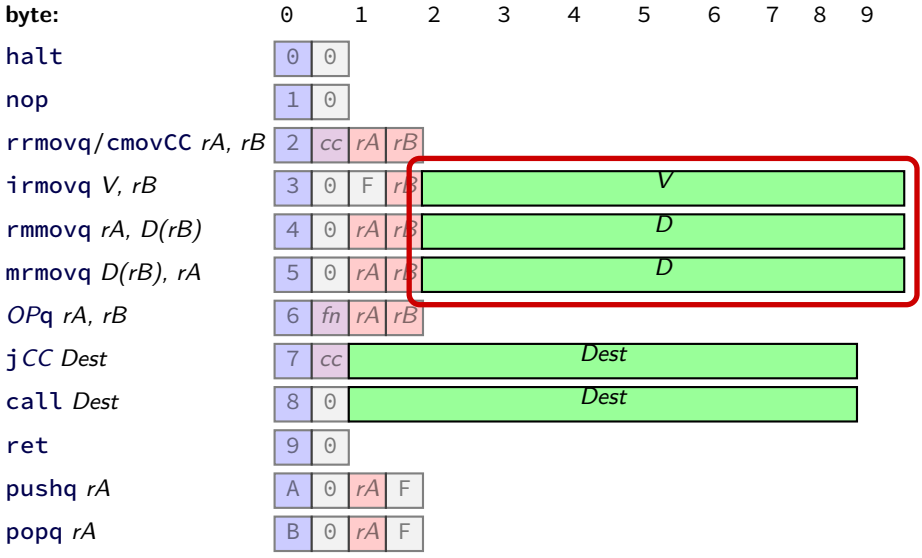
Secondary opcodes: *OP*q



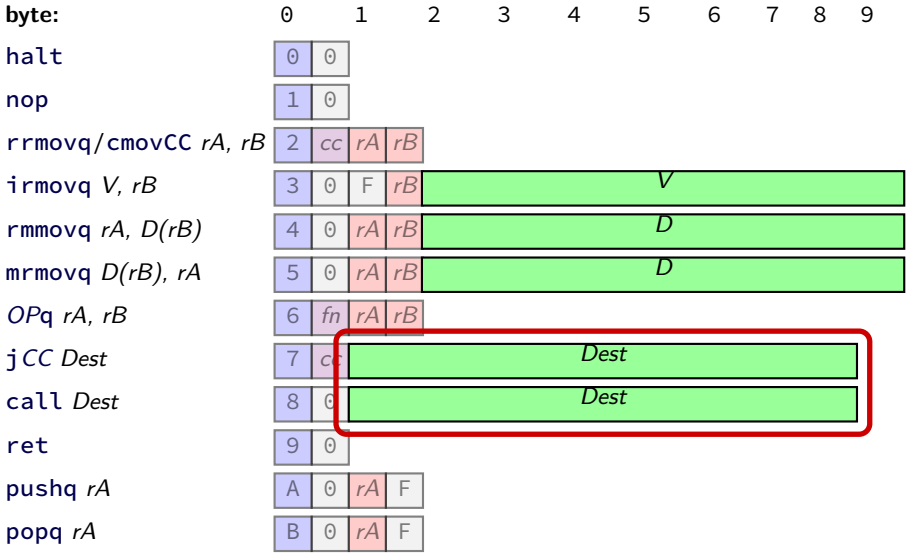
Registers: *rA, rB*



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

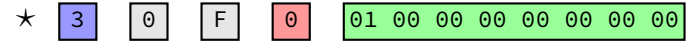
```
rrmovq %rdi, %rax
irmovq $1, %rdi
addq %rdi, %rax
ret
```

Y86-64 encoding (2)

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

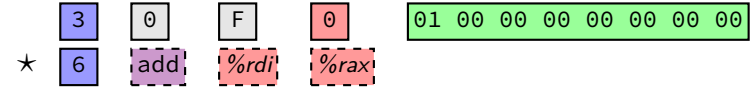
Y86-64 encoding (2)

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



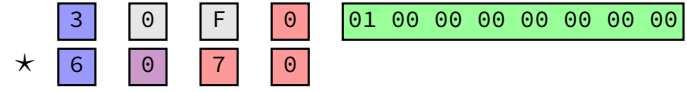
Y86-64 encoding (2)

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



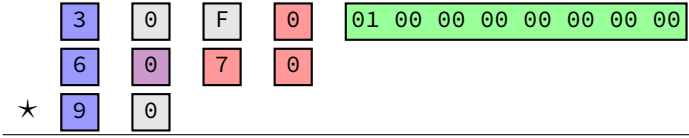
Y86-64 encoding (2)

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



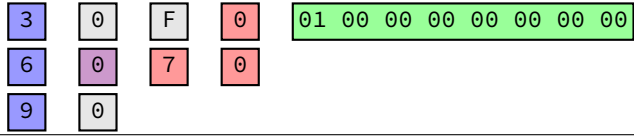
Y86-64 encoding (2)

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



Y86-64 encoding (2)

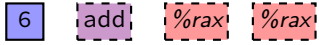
```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```




```
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)

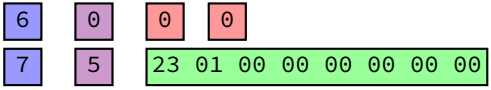
```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```

Y86-64 encoding (3)

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

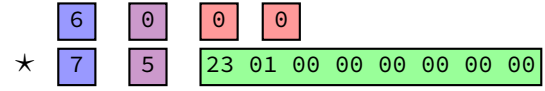
Y86-64 encoding (3)

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



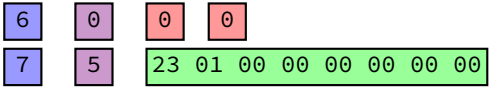
Y86-64 encoding (3)

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



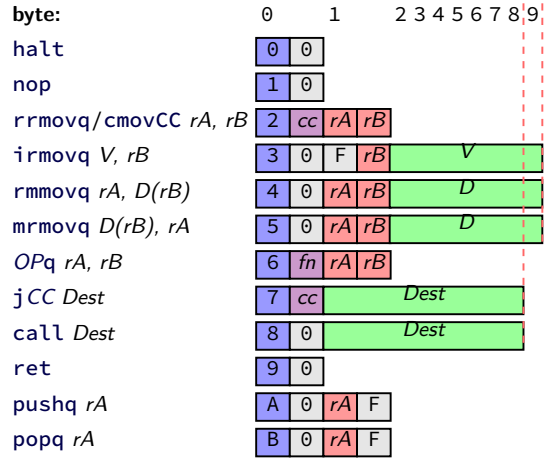
Y86-64 encoding (3)

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



Y86-64 decoding

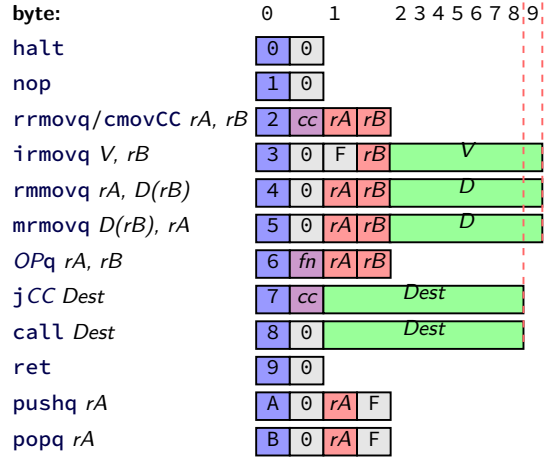
```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```



Y86-64 decoding

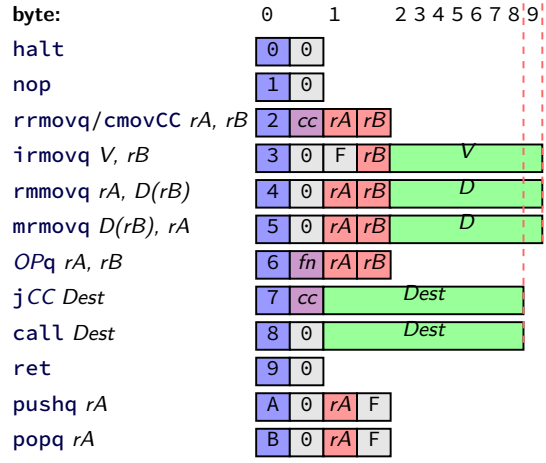
```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

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
```

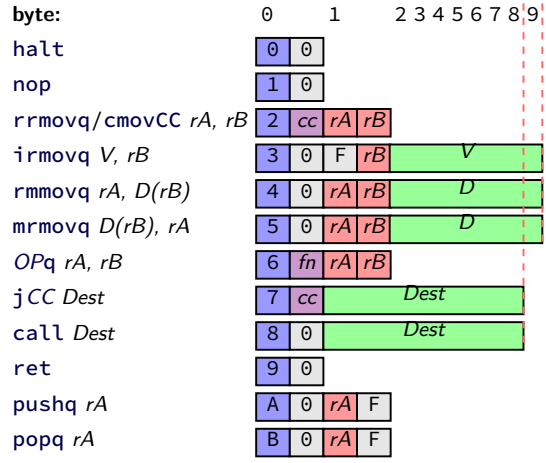


Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
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`

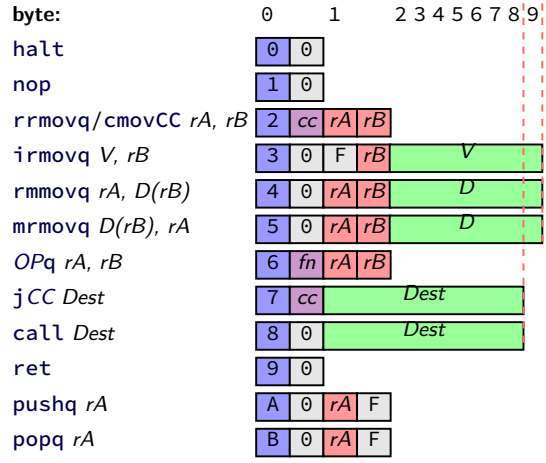


Y86-64 decoding

```
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

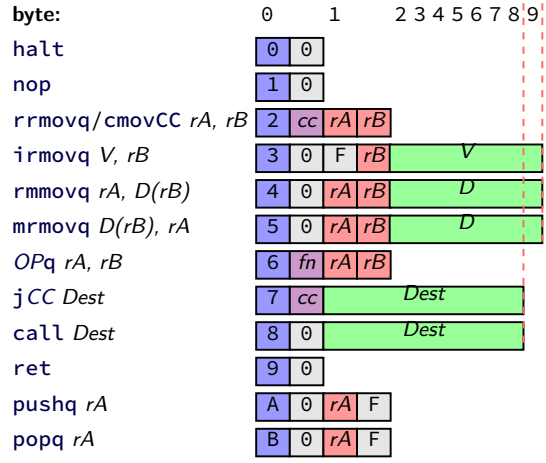


Y86-64 decoding

```
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
```

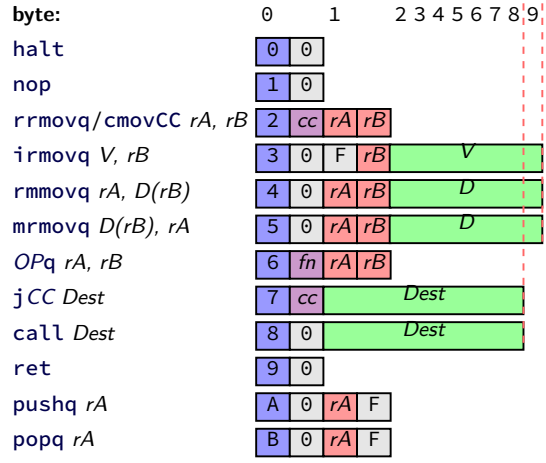
- 2 as cc: `l` (less than)
- hex 84 00… as little endian *Dest: 0x84*



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
```



Back up slides

ISAs: who does the work?

CISC-like

less work for assembly-writers

more work for hardware

choose assembly, design instructions?

harder to build/test CPU

design new instrs for target apps?

RISC-like

more work for assembly-writers

less work for hardware

design for particular kind of HW?

easier to build/test CPU

spend more time optimizing HW?