

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

RISC V — some embedded

...

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)

ISA variation

| instruction set | instr. length | # normal registers | <i>approx.</i> # 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

Why RISC?

complex instructions were usually not faster

complex instructions were harder to implement

compilers, not hand-written assembly

assumption: okay to require compiler modifications

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

ISAs: who does the work?

CISC-like (harder to implement, easier to use assembly)

- choose instructions with particular assembly language in mind?

- more options for hardware to optimize?

- ...but more resources spent on making hardware correct?

- easier to specialize for particular applications

- less work for compilers

RISC-like (easier to implement, harder to use assembly)

- choose instructions with particular HW implementation in mind?

- less options for hardware to optimize?

- simpler to build/test hardware

- ...so more resources spent on making hardware fast?

- more work for compilers

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

common x86 implementations convert to RISC-like
"microinstructions"

relatively cheap because lots of instruction preprocessing needed 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 | jCC | popq |
| andq | cmovCC | movq (renamed) |
| xorq | call | hlt (renamed) |
| nop | ret | |

much, much simpler encoding

Y86-64 instruction set

based on x86

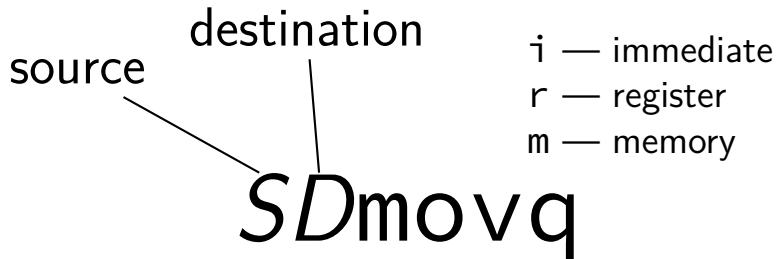
omits most of the 1000+ instructions

leaves

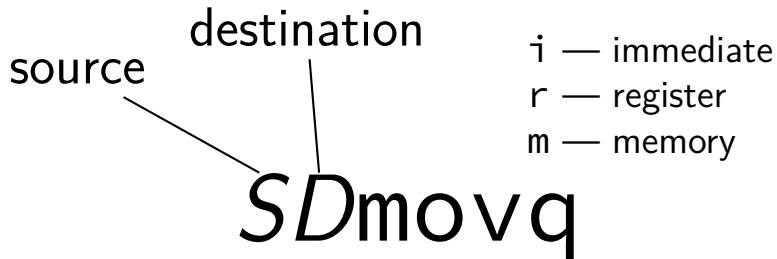
| | | |
|------|--------|----------------|
| addq | jmp | pushq |
| subq | jCC | popq |
| andq | cmovCC | movq (renamed) |
| xorq | call | hlt (renamed) |
| nop | ret | |

much, much simpler encoding

Y86-64: movq

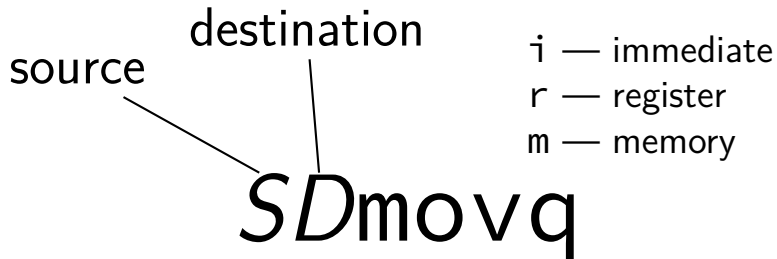


Y86-64: movq



| | | |
|--------|-------------------|-------------------|
| irmovq | immovq | imovq |
| rrmovq | rmmovq | rimovq |
| rrmovq | mmmovq | mimovq |

Y86-64: `movq`



| | |
|---------------------|--------------------------------|
| <code>irmovq</code> | <code>immovq</code> |
| <code>rrmovq</code> | <code>rmmovq</code> |
| <code>rrmovq</code> | <code>mmmovq</code> |

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

cmovCC

conditional move

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

x86-64: 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`)

x86-64 instruction `hlt`

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)

$r12 \leftarrow \text{memory}[10 + r11] + r12$

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

Y86-64: accessing memory (1)

$r12 \leftarrow \text{memory}[10 + r11] + r12$

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

Instead:

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

```
addq %r11, %r12
```

Y86-64: accessing memory (2)

$r12 \leftarrow \text{memory}[10 + 8 * r11] + r12$

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

Y86-64: accessing memory (2)

$r12 \leftarrow \text{memory}[10 + 8 * r11] + r12$

~~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)

$r12 \leftarrow r12 + 1$

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)

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 | SF = 1 or ZF = 1 | value \leq 0 |
| l | SF = 1 | value $<$ 0 |
| e | ZF = 1 | value = 0 |
| ne | ZF = 0 | value \neq 0 |
| ge | SF = 0 | value \geq 0 |
| g | SF = 0 and 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

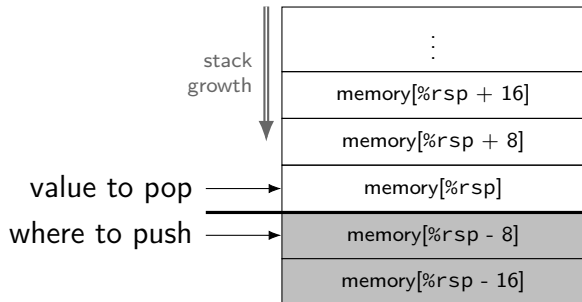
$\%rsp \leftarrow \%rsp - 8$

$\text{memory}[\%rsp] \leftarrow \%rbx$

popq %rbx

$\%rbx \leftarrow \text{memory}[\%rsp]$

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



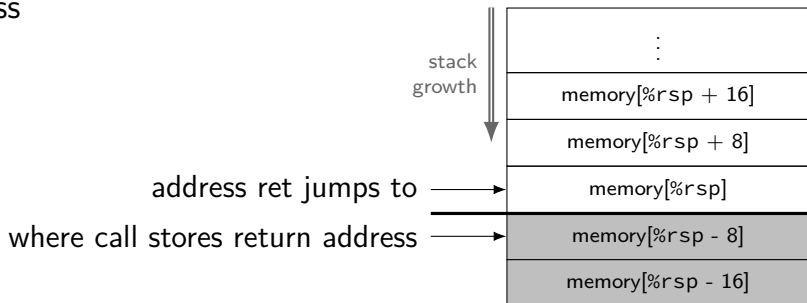
call/ret

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

`%rXX` — 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 <i>rA</i> , <i>rB</i> | 2 | cc | <i>rA</i> | <i>rB</i> | | | | | | |
| irmovq <i>V</i> , <i>rB</i> | 3 | 0 | F | <i>rB</i> | <i>V</i> | | | | | |
| rmmovq <i>rA</i> , <i>D(rB)</i> | 4 | 0 | <i>rA</i> | <i>rB</i> | <i>D</i> | | | | | |
| mrmmovq <i>D(rB)</i> , <i>rA</i> | 5 | 0 | <i>rA</i> | <i>rB</i> | <i>D</i> | | | | | |
| OPq <i>rA</i> , <i>rB</i> | 6 | fn | <i>rA</i> | <i>rB</i> | | | | | | |
| jCC <i>Dest</i> | 7 | cc | <i>Dest</i> | | | | | | | |
| call <i>Dest</i> | 8 | 0 | <i>Dest</i> | | | | | | | |
| ret | 9 | 0 | | | | | | | | |
| pushq <i>rA</i> | A | 0 | <i>rA</i> | F | | | | | | |
| popq <i>rA</i> | B | 0 | <i>rA</i> | F | | | | | | |

Secondary opcodes: cmovcc/jcc

| byte: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|--------------------------------|---|----|----|----|---|---|---|---|---|---|
| halt | 0 | 0 | | | | | | | | |
| nop | 1 | 0 | | | | | | | | |
| $\text{rrmovq/cmovCC } rA, rB$ | 2 | cc | rA | rB | | | | | | |
| $\text{irmovq } V, rB$ | 3 | 0 | F | rB | | | | | | |
| $\text{rmmovq } rA, D(rB)$ | 4 | 0 | rA | rB | | | | | | |
| $\text{mrmovq } D(rB), rA$ | 5 | 0 | rA | rB | | | | | | |
| $\text{OPq } rA, rB$ | 6 | fn | rA | rB | | | | | | |
| $\text{jCC } Dest$ | 7 | cc | | | | | | | | |
| $\text{call } Dest$ | 8 | 0 | | | | | | | | |
| ret | 9 | 0 | | | | | | | | |
| $\text{pushq } rA$ | A | 0 | rA | F | | | | | | |
| $\text{popq } rA$ | B | 0 | rA | F | | | | | | |

| | |
|---|---------------------|
| 0 | always (jmp/rrmovq) |
| 1 | le |
| 2 | l |
| 3 | e |
| 4 | ne |
| 5 | ge |
| 6 | g |

Secondary opcodes: *OPq*

| byte: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------------------------------------|---|----|-----------|-----------|---|-----|---|---|---|---|
| halt | 0 | 0 | | | | | | | | |
| nop | 1 | 0 | | | | | | | | |
| rrmovq/cmovCC <i>rA</i> , <i>rB</i> | 2 | cc | <i>rA</i> | <i>rB</i> | | | | | | |
| irmovq <i>V</i> , <i>rB</i> | 3 | 0 | F | <i>rB</i> | V | | | | | |
| rmmovq <i>rA</i> , <i>D(rB)</i> | 4 | 0 | <i>rA</i> | <i>rB</i> | 0 | add | D | | | |
| mrmmovq <i>D(rB)</i> , <i>rA</i> | 5 | 0 | <i>rA</i> | <i>rB</i> | 1 | sub | D | | | |
| <i>OPq</i> <i>rA</i> , <i>rB</i> | 6 | fn | <i>rA</i> | <i>rB</i> | 2 | and | | | | |
| <i>jCC Dest</i> | 7 | cc | Dest | | | | | | | |
| call <i>Dest</i> | 8 | 0 | Dest | | | | | | | |
| ret | 9 | 0 | | | | | | | | |
| pushq <i>rA</i> | A | 0 | <i>rA</i> | F | | | | | | |
| popq <i>rA</i> | B | 0 | <i>rA</i> | F | | | | | | |

Registers: rA , rB

| byte: | 0 | 1 | 2 |
|-------------------------|---|----|----------|
| halt | 0 | 0 | |
| nop | 1 | 0 | |
| $rrmovq/cmovCC\ rA, rB$ | 2 | cc | $rA\ rB$ |
| $irmovq\ V, rB$ | 3 | 0 | F rB |
| $rmmovq\ rA, D(rB)$ | 4 | 0 | $rA\ rB$ |
| $mrmmovq\ D(rB), rA$ | 5 | 0 | $rA\ rB$ |
| $OPq\ rA, rB$ | 6 | ff | $rA\ rB$ |
| $jCC\ Dest$ | 7 | cc | |
| $call\ Dest$ | 8 | 0 | |
| 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 | | | | | |
| mrmmovq $D(rB)$, rA | 5 | 0 | rA | rB | D | | | | | |
| OPq rA , rB | 6 | fn | rA | rB | | | | | | |
| jCC $Dest$ | 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 | | | | | |
| mrmmovq $D(rB)$, rA | 5 | 0 | rA | rB | D | | | | | |
| OPq rA , rB | 6 | fn | rA | rB | | | | | | |
| jCC $Dest$ | 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 |
|---|---|---|---|-------------------------|

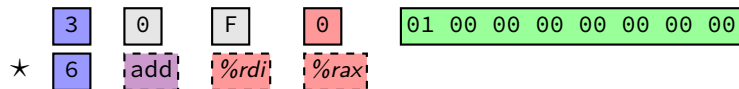
Y86-64 encoding (2)

addOne:

```
irmovq $1, %rax
```

```
addq %rdi, %rax
```

```
ret
```



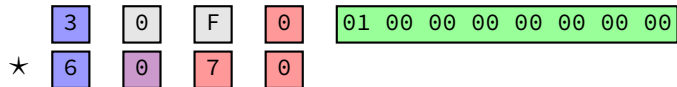
Y86-64 encoding (2)

addOne:

```
irmovq $1, %rax
```

```
addq %rdi, %rax
```

```
ret
```



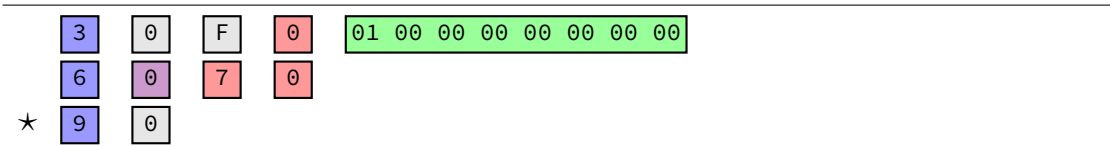
Y86-64 encoding (2)

addOne:

```
irmovq $1, %rax
```

```
addq %rdi, %rax
```

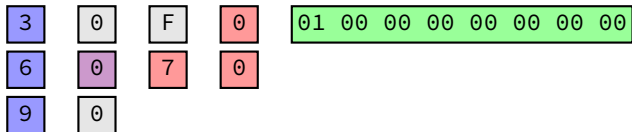
```
ret
```



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

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

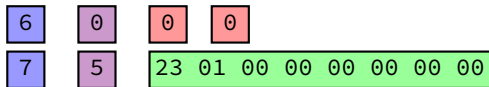
Y86-64 encoding (3)

doubleTillNegative:

/ suppose at address 0x123 */*

addq %rax, %rax

jge doubleTillNegative



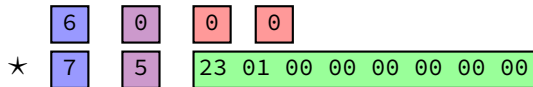
Y86-64 encoding (3)

doubleTillNegative:

/ suppose at address 0x123 */*

addq %rax, %rax

jge doubleTillNegative



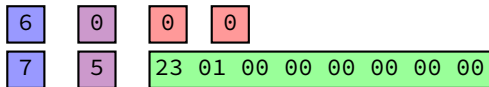
Y86-64 encoding (3)

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

| 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 | | | | | |
| OPq rA, rB | 6 | fn | rA | rB | | | | | | |
| jCC 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

| 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 | | | | | |
| OPq rA, rB | 6 | fn | rA | rB | | | | | | |
| jCC 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

- ▶ 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 | | | | | |
| OPq rA, rB | 6 | fn | rA | rB | | | | | | |
| jCC 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 | | | | | |
| OPq rA, rB | 6 | fn | rA | rB | | | | | | |
| jCC 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: 1 (less than)
- ▶ hex 8400... as little endian *Dest*:
0x84

| byte: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------------------|---|----|------|----|---|---|---|---|---|---|
| halt | 0 | 0 | | | | | | | | |
| nop | 1 | 0 | | | | | | | | |
| rrmovq/cmouvCC 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 | | | | | |
| OPq rA, rB | 6 | fn | rA | rB | | | | | | |
| jCC 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 | | | | | |
| OPq rA, rB | 6 | fn | rA | rB | | | | | | |
| jCC 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 | | | | | |
| OPq rA, rB | 6 | fn | rA | rB | | | | | | |
| jCC 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