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

<table>
<thead>
<tr>
<th>instruction set</th>
<th>instr. length</th>
<th># normal registers</th>
<th>approx. # instrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-64</td>
<td>1–15 byte</td>
<td>16</td>
<td>1500</td>
</tr>
<tr>
<td>Y86-64</td>
<td>1–10 byte</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>ARMv7</td>
<td>4 byte*</td>
<td>16</td>
<td>400</td>
</tr>
<tr>
<td>POWER8</td>
<td>4 byte</td>
<td>32</td>
<td>1400</td>
</tr>
<tr>
<td>MIPS32</td>
<td>4 byte</td>
<td>31</td>
<td>200</td>
</tr>
<tr>
<td>Itanium</td>
<td>41 bits*</td>
<td>128</td>
<td>300</td>
</tr>
<tr>
<td>Z80</td>
<td>1–4 byte</td>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>VAX</td>
<td>1–14 byte</td>
<td>8</td>
<td>150</td>
</tr>
<tr>
<td>z/Architecture</td>
<td>2–6 byte</td>
<td>16</td>
<td>1000</td>
</tr>
<tr>
<td>RISC V</td>
<td>4 byte*</td>
<td>31</td>
<td>500*</td>
</tr>
</tbody>
</table>
other choices: condition codes?

instead of:

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

could do:

```assembly
/* _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
separate 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

source → destination

i — immediate
r — register
m — memory

SDmovq
Y86-64: movq

source

destination

i — immediate
r — register
m — memory

irmovq
rrmovq
mrmovq

immovq

immovq

immovq

immovq

immovq

immovq

immovq

immovq

immovq

immovq

immovq

immovq

immovq
Y86-64: movq

source  \( \rightarrow \) destination

i — immediate
r — register
m — memory

\( \text{SDmovq} \)

irmovq  \( \xrightarrow{\text{immovq}} \)
rrmovq  \( \xrightarrow{\text{rmmovq}} \)
mrmovq  \( \xrightarrow{\text{mmmovq}} \)
Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves
\begin{itemize}
\item addq
\item jmp
\item pushq
\item subq
\item \texttt{jCC}
\item popq
\item \texttt{andq}
\item \texttt{cmovCC}
\item movq (renamed)
\item xorq
\item call
\item hlt (renamed)
\item nop
\item ret
\end{itemize}

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:

```assembly
jle skip_move
rrmovq %rax, %rbx
```

`skip_move:`
// ...

can do:

```assembly
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: rmovq %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)

\[ r_{12} \leftarrow \text{memory}[10 + r_{11}] + r_{12} \]

Invalid: \texttt{addq 10(%r11), %r12}

Instead: \texttt{mrmovq 10(%r11), %r11} /* overwrites %r11 */ \texttt{addq %r11, %r12}
Y86-64: accessing memory (1)

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

Invalid: \( \text{addq } 10(\%r11), \%r12 \)

Instead:

\[ \text{mrmovq } 10(\%r11), \%r11 \]
/* overwrites \%r11 */

\[ \text{addq } \%r11, \%r12 \]
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`
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)

\[ r_{12} \leftarrow r_{12} + 1 \]

Invalid: \( \text{addq} \ $1, \ %r_{12} \)
Y86-64 constants (2)

\[ r12 \leftarrow r12 + 1 \]

Invalid: \[ \text{addq $1, \%r12} \]

Instead, need an extra register:

\[ \text{irmovq $1, \%r11} \]
\[ \text{addq \%r11, \%r12} \]
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__

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Condition Code Bit Test</th>
<th>Value Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>le</td>
<td>SF = 1 or ZF = 1</td>
<td>value ≤ 0</td>
</tr>
<tr>
<td>l</td>
<td>SF = 1</td>
<td>value &lt; 0</td>
</tr>
<tr>
<td>e</td>
<td>ZF = 1</td>
<td>value = 0</td>
</tr>
<tr>
<td>ne</td>
<td>ZF = 0</td>
<td>value ≠ 0</td>
</tr>
<tr>
<td>ge</td>
<td>SF = 0</td>
<td>value ≥ 0</td>
</tr>
<tr>
<td>g</td>
<td>SF = 0 and ZF = 0</td>
<td>value &gt; 0</td>
</tr>
</tbody>
</table>

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

```assembly
cmpq %r11, %r12
ejle somewhere
```

maybe:

```assembly
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

address ret jumps to where call stores return address

```
stack growth
...
memory[%rsp + 16]
memory[%rsp + 8]
memory[%rsp]
memory[%rsp - 8]
memory[%rsp - 16]
```
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

separate 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

<table>
<thead>
<tr>
<th>instruction</th>
<th>byte format</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>rrmovq/cmovCC rA, rB</td>
<td>2 cc rA rB</td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3 0 F rB</td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4 0 rA rB</td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5 0 rA rB</td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn rA rB</td>
<td></td>
</tr>
<tr>
<td>jCC Dest</td>
<td>7 cc Dest</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0 rA F</td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0 rA F</td>
<td></td>
</tr>
</tbody>
</table>
Secondary opcodes: cmovcc/jcc

<table>
<thead>
<tr>
<th>byte:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovq/cmovCC rA, rB</td>
<td>cc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jCC Dest</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **always (jmp/rrmovq)**
- 0: always (jmp/rrmovq)
- 1: le
- 2: l
- 3: e
- 4: ne
- 5: ge
- 6: g
Secondary opcodes: \( OPq \)

<table>
<thead>
<tr>
<th>byte:</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( 4 )</th>
<th>( 5 )</th>
<th>( 6 )</th>
<th>( 7 )</th>
<th>( 8 )</th>
<th>( 9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{halt} )</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{nop} )</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{rrmovq/cmovCC } rA, rB )</td>
<td>2</td>
<td>( \text{cc} )</td>
<td>( rA )</td>
<td>( rB )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{irmovq } V, rB )</td>
<td>3</td>
<td>0</td>
<td>( \text{F} )</td>
<td>( rB )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{rmmovq } rA, D(rB) )</td>
<td>4</td>
<td>0</td>
<td>( rA )</td>
<td>( rB )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{mrmovq } D(rB), rA )</td>
<td>5</td>
<td>0</td>
<td>( rA )</td>
<td>( rB )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{OPq } rA, rB )</td>
<td>6</td>
<td>( \text{fn} )</td>
<td>( rA )</td>
<td>( rB )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{jCC } \text{Dest} )</td>
<td>7</td>
<td>( \text{cc} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{call } \text{Dest} )</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{ret} )</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{pushq } rA )</td>
<td>A</td>
<td>0</td>
<td>( rA )</td>
<td>( \text{F} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{popq } rA )</td>
<td>B</td>
<td>0</td>
<td>( rA )</td>
<td>( \text{F} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( \text{add} \)
- \( \text{sub} \)
- \( \text{and} \)
- \( \text{xor} \)
## Registers: \( rA, rB \)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Byte Value</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>rrmovq/cmovCC ( rA, rB )</td>
<td>2 cc ( rA, rB )</td>
<td></td>
</tr>
<tr>
<td>irmovq ( V, rB )</td>
<td>3 6 F ( rB )</td>
<td></td>
</tr>
<tr>
<td>rmmovq ( rA, D(rB) )</td>
<td>4 6 ( rA, rB )</td>
<td></td>
</tr>
<tr>
<td>mrmmovq ( D(rB), rA )</td>
<td>5 6 ( rA, rB )</td>
<td></td>
</tr>
<tr>
<td>OPq ( rA, rB )</td>
<td>6 fn ( rA, rB )</td>
<td></td>
</tr>
<tr>
<td>jCC Dest</td>
<td>7 cc</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td></td>
</tr>
<tr>
<td>pushq ( rA )</td>
<td>A 6 ( rA) F</td>
<td></td>
</tr>
<tr>
<td>popq ( rA )</td>
<td>B 6 ( rA) F</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Reference</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V )</td>
<td>( D(rB) )</td>
</tr>
<tr>
<td>( D(rB) )</td>
<td>( D(rB) )</td>
</tr>
<tr>
<td>( rA )</td>
<td>( rA )</td>
</tr>
<tr>
<td>( rB )</td>
<td>( rB )</td>
</tr>
<tr>
<td>( %rax ) ( 0 )</td>
<td>( %r8 ) 8</td>
</tr>
<tr>
<td>( %rcx ) ( 1 )</td>
<td>( %r9 ) 9</td>
</tr>
<tr>
<td>( %rdx ) ( 2 )</td>
<td>A ( %r10 )</td>
</tr>
<tr>
<td>( %rbx ) ( 3 )</td>
<td>B ( %r11 )</td>
</tr>
<tr>
<td>( %rsp ) ( 4 )</td>
<td>C ( %r12 )</td>
</tr>
<tr>
<td>( %rbp ) ( 5 )</td>
<td>D ( %r13 )</td>
</tr>
<tr>
<td>( %rsi ) ( 6 )</td>
<td>E ( %r14 )</td>
</tr>
<tr>
<td>( %rdi ) ( 7 )</td>
<td>F ( none )</td>
</tr>
</tbody>
</table>

\( \%rax, \%rcx, \%rdx, \%rbx, \%rsp, \%rbp, \%rsi, \%rdi \)
### Immediates: $V, D, Dest$

<table>
<thead>
<tr>
<th>byte:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>halt</strong></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>nop</strong></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rrmovq/cmovCC $rA, rB</strong></td>
<td>2</td>
<td>cc</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>irmovq $V, rB</strong></td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rmmovq $rA, D(rB)$</strong></td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>mrmovq $D(rB), rA</strong></td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OPq $rA, rB</strong></td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>jCC Dest</strong></td>
<td>7</td>
<td>cc</td>
<td></td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>call Dest</strong></td>
<td>8</td>
<td>0</td>
<td></td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pushq $rA</strong></td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>popq $rA</strong></td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Immediates: \( V, D, \text{Dest} \)

<table>
<thead>
<tr>
<th>byte:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovq/cmovCC ( rA, rB )</td>
<td>2</td>
<td>cc</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq ( V, rB )</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq ( rA, D(rB) )</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmmovq ( D(rB), rA )</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq ( rA, rB )</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j( \text{Dest} )</td>
<td>7</td>
<td>cc</td>
<td></td>
<td></td>
<td></td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call ( \text{Dest} )</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq ( rA )</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq ( rA )</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)

```c
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
```

<table>
<thead>
<tr>
<th>3</th>
<th>0</th>
<th>F</th>
<th>0</th>
<th>01 00 00 00 00 00 00 00 00 00</th>
</tr>
</thead>
</table>

⋆
addOne:

irmovq $1, %rax
addq %rdi, %rax
ret
Y86-64 encoding (2)

addOne:

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

01 00 00 00 00 00 00 00 00 00
Y86-64 encoding (2)

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

```assembly
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
```
addOne:
  irmovq $1, %rax
  addq %rdi, %rax
  ret
doubleTillNegative:
/* suppose at address 0x123 */
addq   %rax, %rax
jge doubleTillNegative
doubleTillNegative:
/* suppose at address 0x123 */
addq %rax, %rax
jge doubleTillNegative
doubleTillNegative:
/* suppose at address 0x123 */
addq %rax, %rax
jge doubleTillNegative
doubleTillNegative:
/* suppose at address 0x123 */
addq     %rax, %rax
jge doubleTillNegative

6 0 0 0
7 5 23 01 00 00 00 00 00 00 00 00
Y86-64 encoding (3)

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

<table>
<thead>
<tr>
<th>6</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>23 01 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Y86-64 encoding (3)

doubleTillNegative:
/* suppose at address 0x123 */
addq %rax, %rax
jge doubleTillNegative
Y86-64 decoding

byte: 0 1 2 3 4 5 6 7 8 9

halt
nop
rrmovq/cmovCC rA, rB
irmovq V, rB
rmmovq rA, D(rB)
mrmovq D(rB), rA
OPq rA, rB
jcc Dest
call Dest
ret
pushq rA
popq rA
## 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
```

<table>
<thead>
<tr>
<th>byte:</th>
<th>0 1 2 3 4 5 6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0 0</td>
</tr>
<tr>
<td>nop</td>
<td>1 0</td>
</tr>
<tr>
<td>rrmovq/cmovCC rA, rB</td>
<td>2 cc rA rB</td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3 0 F rB V</td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4 0 rA rB D</td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5 0 rA rB D</td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn rA rB</td>
</tr>
<tr>
<td>jCC Dest</td>
<td>7 cc Dest</td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0 rA F</td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0 rA F</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrmovq/cmovCC rA, rB</td>
<td>2 cc rA rB</td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3 0 F rB V</td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4 0 rA rB D</td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5 0 rA rB D</td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn rA rB</td>
</tr>
<tr>
<td>jCC Dest</td>
<td>7 cc Dest</td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0 rA F</td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0 rA F</td>
</tr>
</tbody>
</table>
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 00

rrmovq %rcx, %rax
▶ 0 as cc: always
▶ 1 as reg: %rcx
▶ 0 as reg: %rax

byte:
<table>
<thead>
<tr>
<th>byte</th>
<th>cc</th>
<th>rA</th>
<th>rB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>cc</td>
<td>rA</td>
<td>rB</td>
</tr>
<tr>
<td>3</td>
<td>0F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
</tr>
<tr>
<td>7</td>
<td>cc</td>
<td></td>
<td>Dest</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td>Dest</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

rrmovq / cmovCC rA, rB
irmovq V, rB
rrmovq rA, D(rB)
mrmovq D(rB), rA
OPq rA, rB
jCC Dest
call Dest
ret
pushq rA
popq rA
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
```

```assembly
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
- nop
- rrmovq/cmovCC rA, rB
- irmovq V, rB
- rrmovq rA, D(rB)
- mrmovq D(rB), rA
- OPq rA, rB
- jCC Dest
- call Dest
- ret
- pushq rA
- popq rA

```
``
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
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

0 0
1 0
2 cc rA rB
3 0 F rB
4 0 rA rB
5 0 rA rB
6 fn rA rB
7 cc Dest
8 0 Dest
9 0
A 0 rA F
B 0 rA F
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 00

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

<table>
<thead>
<tr>
<th>byte:</th>
<th>0 1 2 3 4 5 6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0 0</td>
</tr>
<tr>
<td>nop</td>
<td>1 0</td>
</tr>
<tr>
<td>rrmovq/cmovCC rA, rB</td>
<td>2 cc rA rB</td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3 0 F rB V</td>
</tr>
<tr>
<td>rrmovq rA, D(rB)</td>
<td>4 0 rA rB D</td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5 0 rA rB D</td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn rA rB</td>
</tr>
<tr>
<td>jcc Dest</td>
<td>7 cc Dest</td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0 rA F</td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0 rA F</td>
</tr>
</tbody>
</table>
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
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