

ISAs and Y86-64

Changelog

Corrections made in this version not in first posting:
7 Feb 2017: slide 55: first byte of jge is 75, not 70 or 7F

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
- RISC V — some embedded

...

last time

- ISAs versus microarchitectures
- condition codes
- C loops to assembly loops
- lab Wednesday — in-lab quiz

strlen/strsep lab

two functions that operate on C strings
terminated by 'sentinel' \0

lab writeup has example implementations

you: code both under test conditions

4

anonymous feedback (1)

bunch of comments (some not quite as polite) like:

“According to the Schedule, only section 4.1 was required reading before class on Tuesday, however, at the top of the pre-quiz it stated to read sections 4.1–4.3. If you could be consistent, that would be nice. This way, we actually know what to study before taking the quiz. Thanks.”

pre-week — readings for **entire week**'s lectures

post-week — material from **entire week**'s lectures

5

anonymous feedback (2)

“why is Professor Khan lecturing instead of Reiss when he is in the lecture?”

should be less work for both of us — better quality?

need to stay in sync — labs/homeworks

yes, some will like one more than the other

yes, one has more experience teaching this course

6

anonymous feedback (3)

“bring back the .webm recordings of the lectures”

“Could we go over the quizzes in class the next day, so students who got it wrong can know how the problem is supposed to be solved? If that's not possible, could we receive solutions to the quizzes?”

7

anonymous feedback (4)

“I think it is unfair that when asked about the flags in lecture last week, Khan fumbled her answer and could not explain when each flag would be set yet that was still a quiz question. If Khan cannot explain to us in lecture which assembly command would set which flag...”

question on quiz was comparing cmpq and subq
cmpq is subtraction without storing result
shouldn't have needed to exactly how flags were set

8

anonymous feedback (5)

“I am looking over the lab writeup and I cannot understand what the strsep() method is supposed to be doing. I have looked all over the Internet and can't find anything that would be helpful. If we could spend maybe 5-10 minutes in class Tuesday just going over what the method is supposed to do then maybe I'll have a better idea of how to approach the problem.”

9

strsep (1)

```
char *strsep(char **ptrToString, char delimiter);
char buffer[] = "this_is_a_test";
char *ptr = buffer;
char *token;
while ((token = strsep(&ptr, '_')) {
    printf("[%s]", token);
}
/* output: [this][is][a][test] */
/* final value of buffer:
   "this\0is\0a\0test" */
```

10

strsep (2)

```
char *strsep(char **ptrToString, char delimiter);
char str[] = "this_is_a_test";
char *ptr = str;
char *token;
token = strsep(&ptr, '_');
/* token points to &buffer[0], string "this" */
/* ' ' after "this" replaced by '\0' */
/* ptr points to &buffer[5]:
   "is a test" */
```

11

anonymous feedback (6)

"I believe that the question "What does the output tell us about the input v?" has the incorrect answer. Consider the case in which v=6. Six is not a power of 2, yet it return s true (0010)."

```

! is logical; ~ is bitwise
&& is logical; & is bitwise
!4 == 0; ~4 == 0xFFF...B
f = v && !(v & (v - 1));
f = 6 && !(6 & 5);
f = 6 && !(4);
f = 6 && 0;
f = 0;
    
```

lists homework

```

short sentinel = -9999;
short *x = malloc(sizeof(short) * 4);
x[0] = 1; x[1] = 2; x[2] = 3;
x[3] = sentinel;
    
```

```

typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short) * 3);
x.ptr[0] = 1; ...
    
```

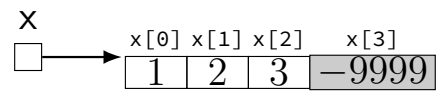
```

typedef struct node_t {
    short payload;
    list *next;
} node;
node *x = malloc(sizeof(node_t));
x->payload = 1;
x->next = ...
    
```

lists homework

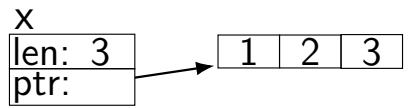
```

short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
    
```



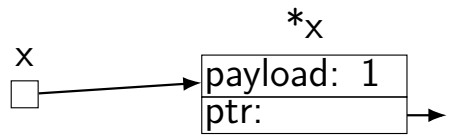
```

typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
    
```



```

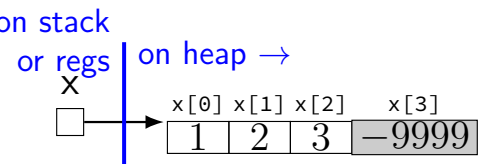
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
    
```



lists homework

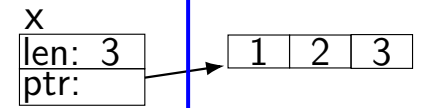
```

short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
    
```



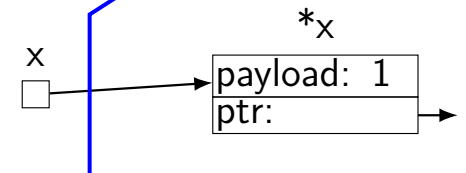
```

typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
    
```



```

typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
    
```



about our final

9 May, 7PM–10PM

location will be announced

form for conflicts will be announced

15

non-anonymous feedback

16

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*

17

Other choices: condition codes?

instead of:

```
testq %r11, %r12  
je somewhere
```

could do:

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

18

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)

19

Other choices: number of operands

add `dest, src1, src2`

ARM, POWER, MIPS, SPARC, ...

add `src2, src1=dest` (or swapped)

x86, AVR, Z80, ...

VAX: both

20

CISC and RISC

RISC — Reduced Instruction Set Computer

Reduced from what?

21

Other choices: instruction complexity

instructions that write multiple values?

x86-64: **push**, **pop**, **movsb**, ...

more?

22

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 *coefficientsPtr* at the value *x*.

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

23

microcode

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

loop in hardware???

typically: lookup sequence of **microinstructions**

secret simpler instruction set

24

Why RISC?

complex instructions were usually not faster

complex instructions were harder to implement

compilers, not hand-written assembly

25

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

26

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

27

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

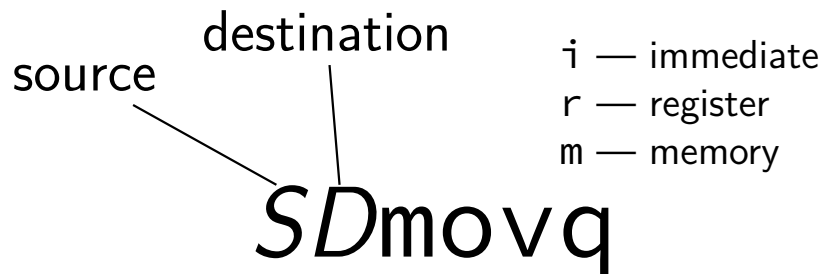
leaves

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

much, much simpler encoding

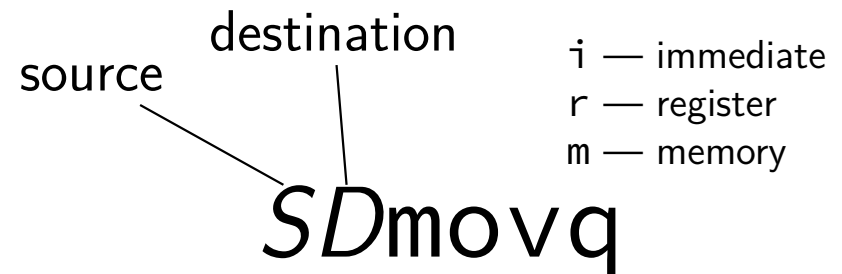
27

Y86-64: movq



28

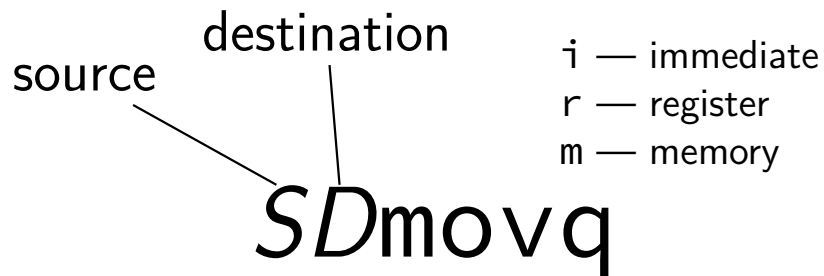
Y86-64: movq



```
irmovq  immovq  imovq
rrmovq  rmmovq    rimovq
mrmovq  mmmovq  mimovq
```

28

Y86-64: movq



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

28

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

29

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
```

30

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

31

halt

(x86-64 instruction called `hlt`)

Y86-64 instruction `halt`

stops the machine

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

real processors: reserved for OS

32

Y86-64: specifying addresses

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

33

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)`~~

33

Y86-64: accessing memory (1)

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

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

34

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
```

34

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
```

35

Y86-64 constants (1)

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

only instruction with non-address constant operand

35

36

Y86-64 constants (2)

$r12 \leftarrow r12 + 1$

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

37

Y86-64 constants (2)

$r12 \leftarrow r12 + 1$

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

Instead, need an extra register:

irmovq \$1, %r11

addq %r11, %r12

37

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)

38

Y86-64: Condition codes

ZF — value was zero?

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

set by all arithmetic/logic

addq, subq, andq, xorq

not set by anything else

39

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)

40

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)

41

Y86-64: Conditionals (1)

~~cmp, test~~

42

Y86-64: Conditionals (1)

~~cmp, test~~

instead: use side effect of normal arithmetic

42

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)

42

Y86-64: Conditionals (2)

~~cmp, test~~

instead: use side effect of normal arithmetic

instead of

```
test %r11, %r11
je somewhere
```

use:

```
andq %r11, %r11
je somewhere
```

(doesn't change %r11)

43

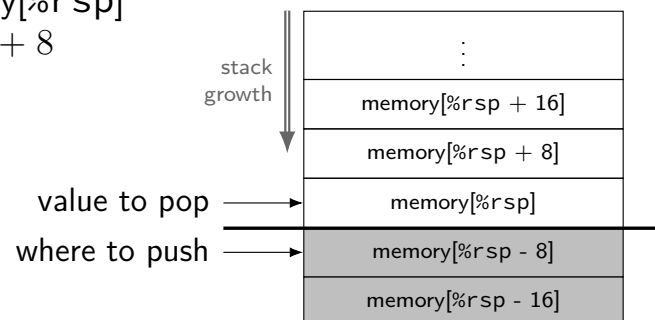
push/pop

pushq %rbx

```
%rsp ← %rsp - 8
memory[%rsp] ← %rbx
```

popq %rbx

```
%rbx ← memory[%rsp]
%rsp ← %rsp + 8
```



44

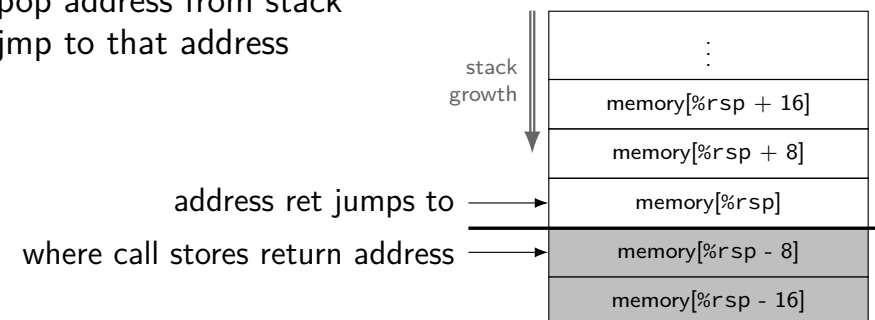
call/ret

call LABEL

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

ret

```
pop address from stack
jmp to that address
```



45

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

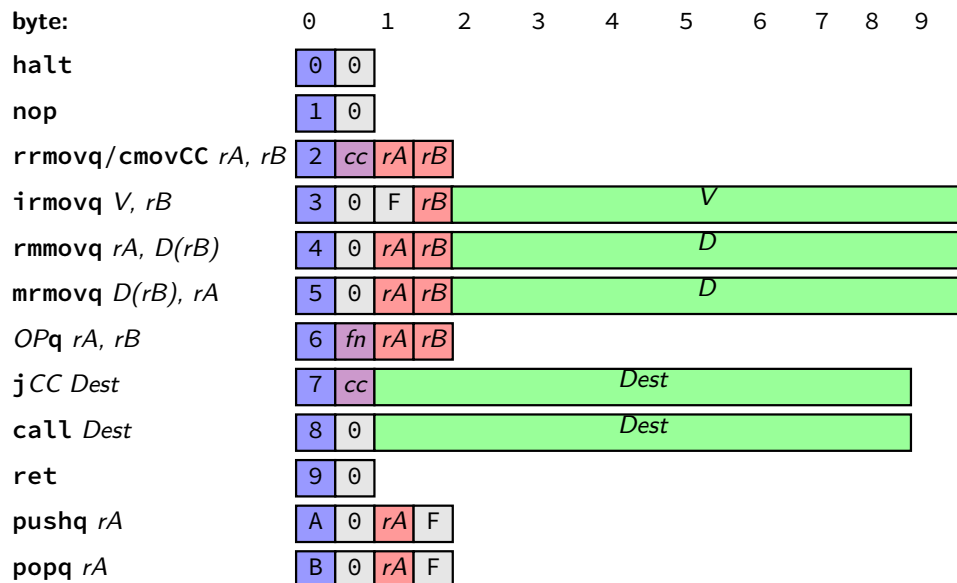
more registers

no "loops" within single instructions

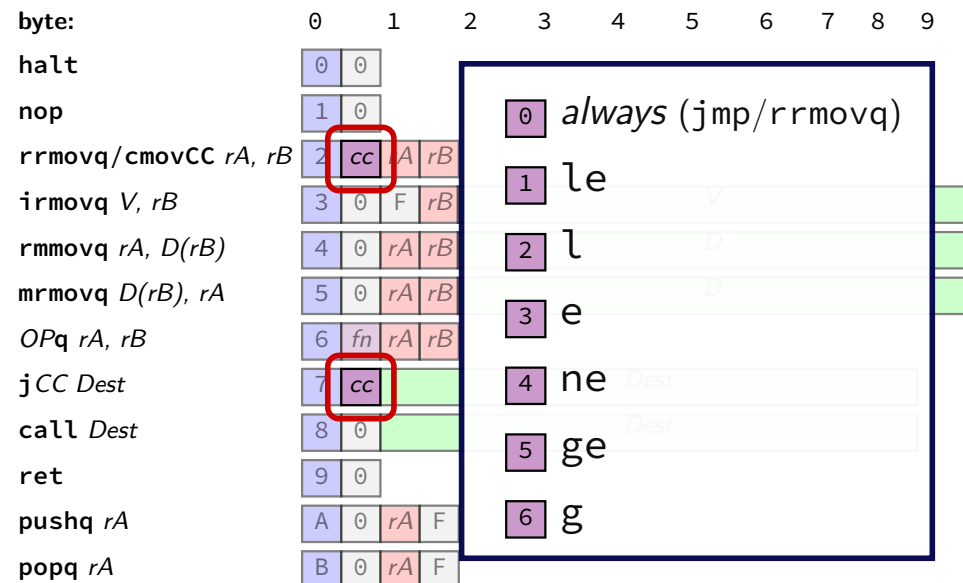
no instructions with two memory operands

few addressing modes

Y86-64 instruction formats



Secondary opcodes: cmovcc/jcc



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>	<i>V</i>					
rmmovq <i>rA</i> , <i>D(rB)</i>	4	0	<i>rA</i>	<i>rB</i>	<i>D</i>					
rrmovq <i>D(rB)</i> , <i>rA</i>	5	0	<i>rA</i>	<i>rB</i>	<i>D</i>					
<i>OPq</i> <i>rA</i> , <i>rB</i>	6	fn	<i>rA</i>	<i>rB</i>						
<i>jCC 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						

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 <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>					
rrmovq <i>D(rB)</i> , <i>rA</i>	5	0	<i>rA</i>	<i>rB</i>	<i>D</i>					
<i>OPq</i> <i>rA</i> , <i>rB</i>	6	fn	<i>rA</i>	<i>rB</i>						
<i>jCC 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						

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 <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>					
rrmovq <i>D(rB)</i> , <i>rA</i>	5	0	<i>rA</i>	<i>rB</i>	<i>D</i>					
<i>OPq</i> <i>rA</i> , <i>rB</i>	6	fn	<i>rA</i>	<i>rB</i>						
<i>jCC 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						

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

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>					
rrmovq <i>D(rB)</i> , <i>rA</i>	5	0	<i>rA</i>	<i>rB</i>	<i>D</i>					
<i>OPq</i> <i>rA</i> , <i>rB</i>	6	fn	<i>rA</i>	<i>rB</i>						
<i>jCC 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						

Y86-64 encoding (1)

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

x86-64:

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

Y86-64:

53

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
```

53

Y86-64 encoding (2)

addOne:

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

*

3	0	F	%rax	01 00 00 00 00 00 00 00
---	---	---	------	-------------------------

54

Y86-64 encoding (2)

addOne:

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

*

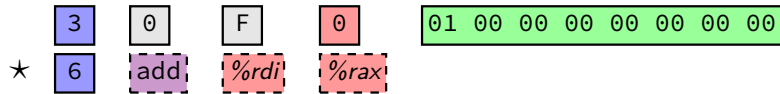
3	0	F	0	01 00 00 00 00 00 00 00
---	---	---	---	-------------------------

54

Y86-64 encoding (2)

addOne:

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

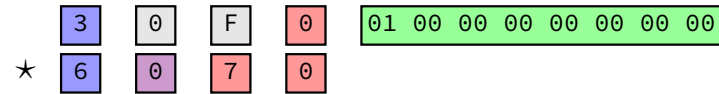


54

Y86-64 encoding (2)

addOne:

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

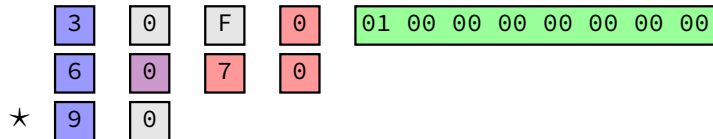


54

Y86-64 encoding (2)

addOne:

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

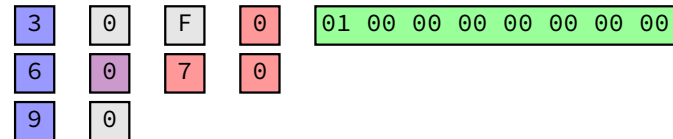


54

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

54

Y86-64 encoding (3)

doubleTillNegative:

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

6 add %rax %rax

55

Y86-64 encoding (3)

doubleTillNegative:

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

* 6 add %rax %rax

55

Y86-64 encoding (3)

doubleTillNegative:

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

* 6 0 0 0

55

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

55

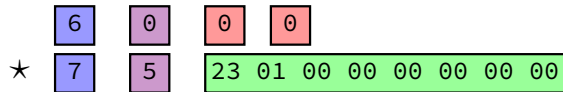
Y86-64 encoding (3)

doubleTillNegative:

/ suppose at address 0x123 */*

`addq %rax, %rax`

`jge doubleTillNegative`



55

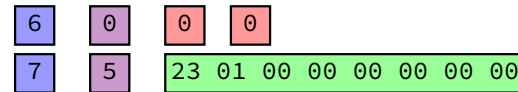
Y86-64 encoding (3)

doubleTillNegative:

/ suppose at address 0x123 */*

`addq %rax, %rax`

`jge doubleTillNegative`



55

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 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						

56

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 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						

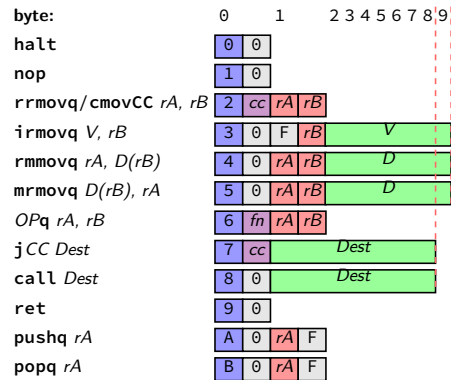
56

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 00
```

rrmovq %rcx, %rax

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



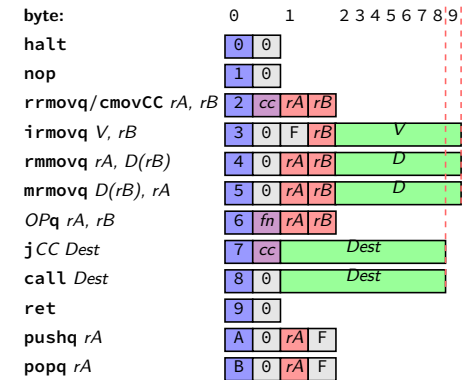
56

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 00
```

rrmovq %rcx, %rax

- addq %rdx, %rax**
- subq %rbx, %rdi**
- ▶ 0 as fn: add
- ▶ 1 as fn: sub



56

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 00
```

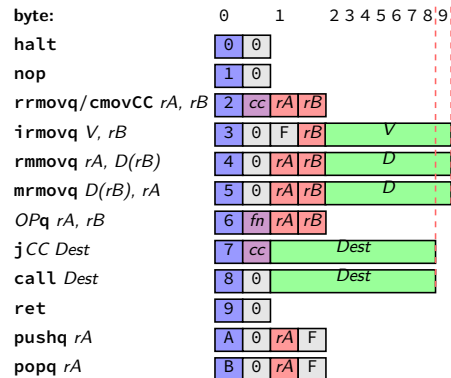
rrmovq %rcx, %rax

addq %rdx, %rax

subq %rbx, %rdi

jL 0x84

- ▶ 2 as cc: l (less than)
- ▶ hex 84 00... as little endian
Dest: 0x84



56

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 00
```

rrmovq %rcx, %rax

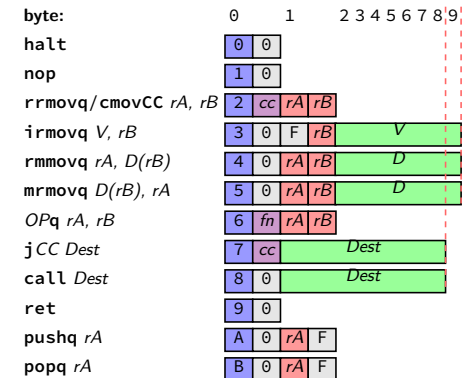
addq %rdx, %rax

subq %rbx, %rdi

jL 0x84

rrmovq %rax, %rcx

jmp 0x68



56

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					
rrmovq 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						

57

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

58