

# CS 3330: ISAs and Y86-64

8 September 2016

# Changelog

Corrections made in this version not in first posting:

10 Sep 2016: slide 24: jle looks for  $ZF = 1$ , not  $ZF = 0$

# ISAs being manufactured today

x86 — dominant in desktops, servers

ARM — dominant in mobile devices

POWER — Wii U, IBM supercomputers

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

...

# Instruction Set Architecture

the instructions

types of operations?

registers/storage?

how their machine code looks

# 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

# Other choices: condition codes?

instead of:

```
testq %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** dest, src1, src2

ARM, POWER, MIPS, SPARC, ...

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

x86, AVR, Z80, ...

VAX: both



# CISC and RISC

RISC — Reduced Instruction Set Computer

Reduced from what?

# Other choices: instruction complexity

instructions that write multiple values?

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

more?

# 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*.

# 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

# Why RISC?

complex instructions were not faster

complex instructions were harder to implement

compilers, not hand-written assembly

# 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 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

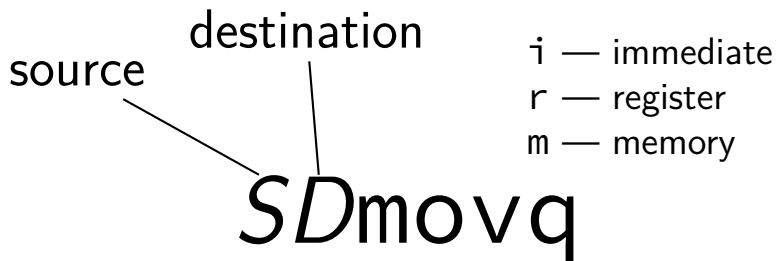
leaves

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

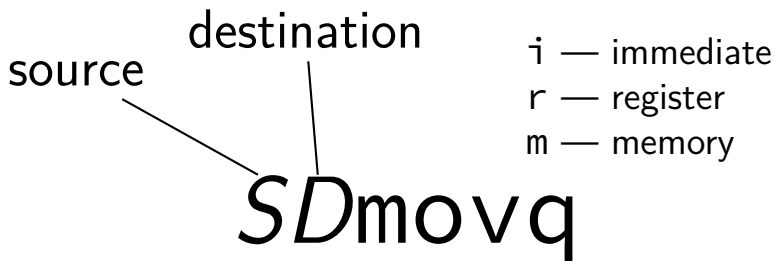
much, much simpler encoding



## Y86-64: `movq`

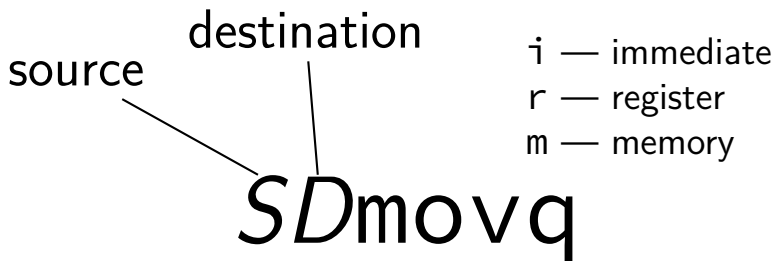


# Y86-64: `movq`



irmovq	<del>imovq</del>	<del>imovq</del>
rrmovq	rmmovq	<del>rimovq</del>
rrmovq	<del>mmmovq</del>	<del>mimovq</del>

# Y86-64: `movq`



irmovq    ~~imovq~~

rrmovq    rmmovq

mrmmovq    ~~mmovq~~

# 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?

set by **all** arithmetic

# 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)

# 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



# 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)

## 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)

# push/pop

**pushq %rbx**

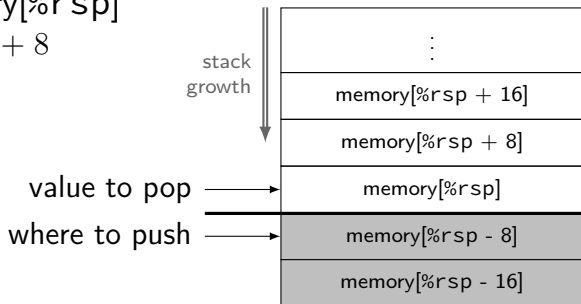
$\%rsp \leftarrow \%rsp - 8$

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

**popq %rbx**

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

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



# 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>					
mrmovq <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
<code>halt</code>	0	0								
<code>nop</code>	1	0								
<code>rrmovq/cmovCC rA, rB</code>	2	cc	rA	rB						
<code>irmovq V, rB</code>	3	0	F	rB						
<code>rmmovq rA, D(rB)</code>	4	0	rA	rB						
<code>mrmovq D(rB), rA</code>	5	0	rA	rB						
<code>OPq rA, rB</code>	6	fn	rA	rB						
<code>jCC Dest</code>	7	cc								
<code>call Dest</code>	8	0								
<code>ret</code>	9	0								
<code>pushq rA</code>	A	0	rA	F						
<code>popq rA</code>	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 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						

0 add

1 sub

2 and

3 xor

# Registers: *rA*, *rB*

byte:	0	1	2
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>
rmmovq <i>rA</i> , <i>D(rB)</i>	4	0	<i>rA</i> <i>rB</i>
mrmovq <i>D(rB)</i> , <i>rA</i>	5	0	<i>rA</i> <i>rB</i>
OPq <i>rA</i> , <i>rB</i>	6	ff	<i>rA</i> <i>rB</i>
jCC <i>Dest</i>	7	cc	
call <i>Dest</i>	8	0	
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 $rA$ , $rB$	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						

# 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	$rA$	$rB$	$D$					
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 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 example (2)

addOne:

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

---

★ 

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

---



## Y86-64 example (2)

addOne:

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

---

★ 

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

---

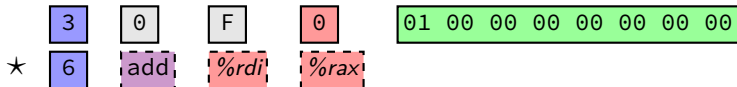
## Y86-64 example (2)

addOne:

`irmovq $1, %rax`

`addq %rdi, %rax`

`ret`



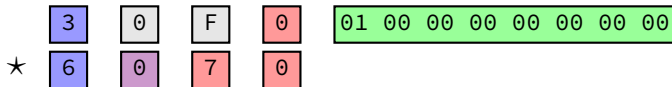
## Y86-64 example (2)

addOne:

**irmovq** \$1, %rax

**addq** %rdi, %rax

**ret**



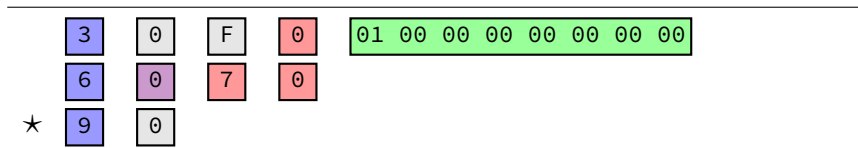
# Y86-64 example (2)

addOne:

**irmovq** \$1, %rax

**addq** %rdi, %rax

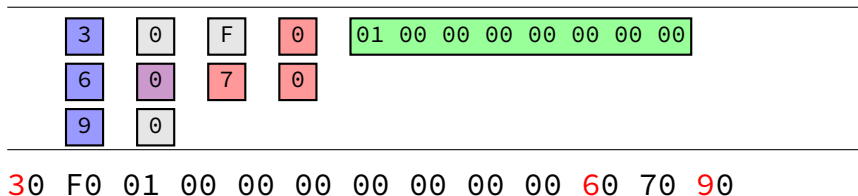
**ret**



# Y86-64 example (2)

addOne:

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



# Y86-64 decoding

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

# Y86-64 decoding

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

# Y86-64 decoding

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

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

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

**rrmovq** %rcx, %rax

**addq** %rdx, %rax

**subq** %rbx, %rdi

**j1** 0x84

► 2 as cc: 1 (less than)

► hex 84 00... as little endian

*Dest: 0x84*

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 decoding

20 10 60 20 61 37 72 84 00 00 00 00 00 00 00 00 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					
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: Convenience for hardware

one nibble 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					
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

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