Changelog

Changes made in this version not seen in first lecture:

- 7 September 2017: slide 37: correct text about division speed: four-byte division is weirdly not much slower than 1-byte division on Skylake (but 64-bit division is much slower)
- 7 September 2017: slide 32: was missing rrmovq near end of decoded instructions

Y86 / Binary Ops

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

```
start_loop:
  cmpq $10, %rbx
    # rbx >= 10?
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
    ...
    ...
    ...
    ...
```

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

```
start_loop:
  cmpq $10, %rbx
    # rbx >= 10?
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
    ...
    ...
    ...
    ...
```

```
  cmpq $10, %rbx
    # rbx >= 10?
  jge end_loop
start_loop:
  call foo
  addq $1, %rbx
  cmpq $10, %rbx
    # rbx != 10?
  jne start_loop
end_loop:
    ...
    ...
    ...
```

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

```
start_loop:
  cmpq $10, %rbx
    # rbx >= 10?
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
    ...
    ...
    ...
    ...
```

```
  cmpq $10, %rbx
    # rbx >= 10?
  jge end_loop
start_loop:
  call foo
  addq $1, %rbx
  cmpq $10, %rbx
    # rbx != 10?
  jne start_loop
end_loop:
    ...
    ...
    ...
```

```
  cmpq $10, %rbx
    # rbx >= 10
  jge end_loop
  movq $10, %rax
  subq %rbx, %rax
  movq %rax, %rbx
start_loop:
  call foo
  decq %rbx
    # rbx != 0
  jne start_loop
  movq $10, %rbx
end_loop:
```

last time

condition codes: ZF (zero), SF (sign), OF (overflow), CF (carry)

jump tables: `jmp *table(%rax)`

read address of next instruction from table

microarchitecture vs. instruction set architecutre (ISA)

`cmov`*CC*: conditional move

Y86: `movq` → {`rrmovq`, `irmovq`, `mrmovq`, `rmmovq`}

pre-quiz next week

textbooks are definitely available

quiz on reading for next week

get a textbook if you don't have one

bomb HW grades

are on the gradebook

please check: possible you registered a bomb with an invalid computing ID

some transient weirdness with gradebook if you had used multiple bombs, now fixed

strlen/strsep lab

next week: in-lab quiz to write two functions:

`strlen` — length of nul-terminated string

`strsep` (simplified) — divide string into 'tokens'

strsep (1)

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

strsep (2)

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

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: 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 ← memory[10 + r11] + r12

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

Y86-64: accessing memory (1)

r12 ← 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 ← memory[10 + 8 * r11] + r12

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

Y86-64 constants (1)

`irmovq $100, %r11`

only instruction with non-address constant operand

Y86-64 constants (2)

r12 ← r12 + 1

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

Y86-64 constants (2)

r12 ← 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 ≤ 0
`l`	$SF = 1$	value < 0
`e`	$ZF = 1$	value $= 0$
`ne`	$ZF = 0$	value $\neq 0$
`ge`	$SF = 0$	value ≥ 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)

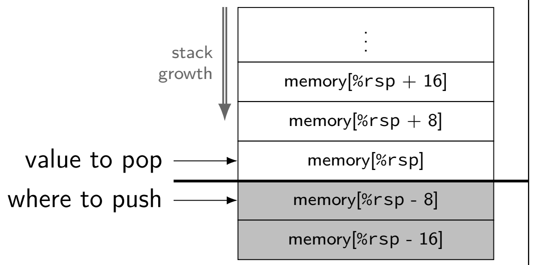
push/pop

`pushq %rbx`

- %rsp $\leftarrow$ %rsp $- 8$
- memory[%rsp] $\leftarrow$ %rbx

`popq %rbx`

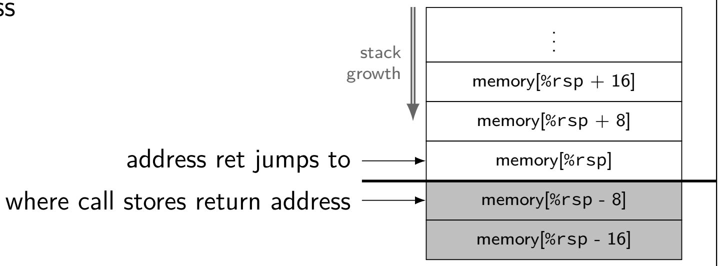
- %rbx $\leftarrow$ memory[%rsp]
- %rsp $\leftarrow$ %rsp $+ 8$



call/ret

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

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



Y86-64 state

`%r`*XX* — 15 registers
- ~~`%r15`~~ missing — replaced with “no register”
- smaller parts of registers missing

`ZF` (zero), `SF` (sign), ~~`OF` (overflow)~~
- book has `OF`, we'll not use it
- ~~`CF`~~ (carry) missing (no unsigned jumps)

`Stat` — processor status — halted?

PC — **p**rogram **c**ounter (AKA instruction pointer)

main memory

typical RISC ISA properties

fewer, simpler instructions

seperate instructions to access memory

~~fixed-length instructions~~

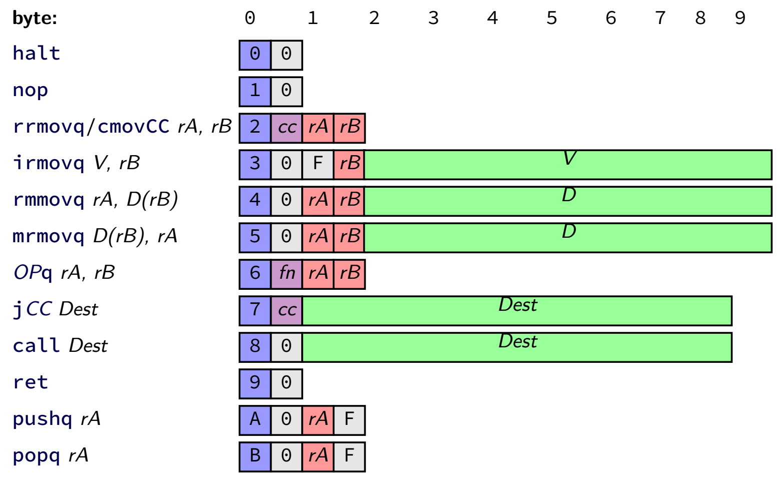
more registers

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` *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*							
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*								
`j`*CC Dest*	7	*cc*	*Dest*									
`call` *Dest*	8	0	*Dest*									
`ret`	9	0										
`pushq` *rA*	A	0	*rA*	F								
`popq` *rA*	B	0	*rA*	F								

secondary opcodes: cmov*cc*/j*cc*

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*							
*OP*q *rA*, *rB*	6 *fn*	*rA* *rB*								
j*CC* *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	*always* (jmp/rrmovq)
1	le
2	l
3	e
4	ne
5	ge
6	g

secondary opcodes: *OP*q

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*							
*OP*q *rA*, *rB*	6 *fn*	*rA* *rB*								
j*CC* *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 *rA*, *rB*	2 *cc*	*rA* *rB*	
irmovq *V*, *rB*	3 0	F *rB*	
rmmovq *rA*, *D*(*rB*)	4 0	*rA* *rB*	
mrmovq *D*(*rB*), *rA*	5 0	*rA* *rB*	
*OP*q *rA*, *rB*	6 *fn*	*rA* *rB*	
j*CC* *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*

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*	
mrmovq *D*(*rB*), *rA*	5 0	*rA* *rB*	
*OP*q *rA*, *rB*	6 *fn*	*rA* *rB*	
j*CC* *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*							
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*							
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*								
`j`*CC 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*							
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*								
`j`*CC Dest*	7	*cc*	*Dest*									
`call` *Dest*	8	0	*Dest*									
`ret`	9	0										
`pushq` *rA*	A	0	*rA*	F								
`popq` *rA*	B	0	*rA*	F								

28

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
```

```
  3 0   F     0     01 00 00 00 00 00 00 00
★ 6 add %rdi %rax
```

Y86-64 encoding (2)

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

```
  3 0 F 0 01 00 00 00 00 00 00 00
★ 6 0 7 0
```

Y86-64 encoding (2)

```
addOne:
  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
```

Y86-64 encoding (2)

```
addOne:
  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
```

```
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
```

```
  6  0   0  0
⋆ 7  ge  23 01 00 00 00 00 00 00
```

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 00
```

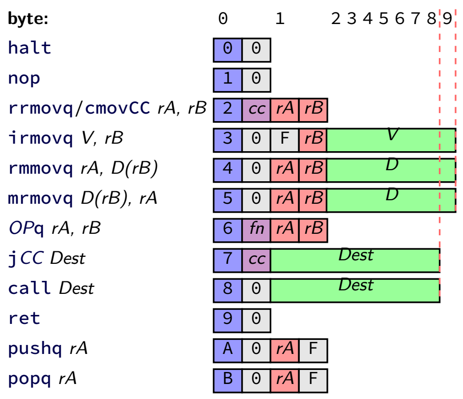
Y86-64 encoding (3)

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

```
  6  0  0  0
  7  5  23 01 00 00 00 00 00 00
```

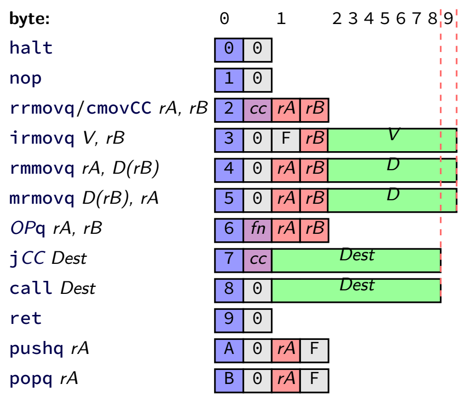
Y86-64 decoding

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



Y86-64 decoding

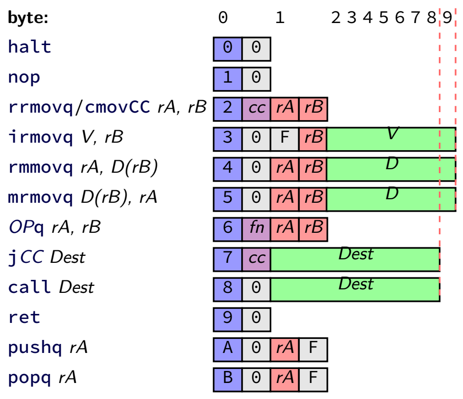
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00



Y86-64 decoding

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

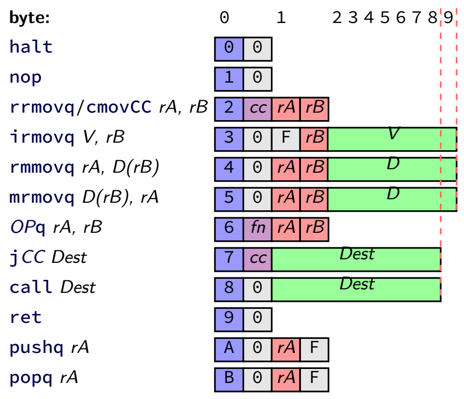
```
rrmovq %rcx, %rax
```
- 0 as cc: always
- 1 as reg: `%rcx`
- 0 as reg: `%rax`



Y86-64 decoding

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

```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
```
- 0 as fn: `add`
- 1 as fn: `sub`

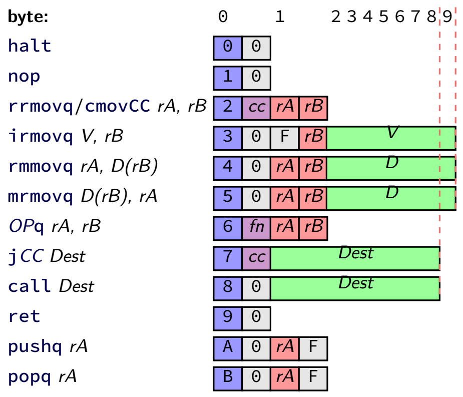


Y86-64 decoding

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

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

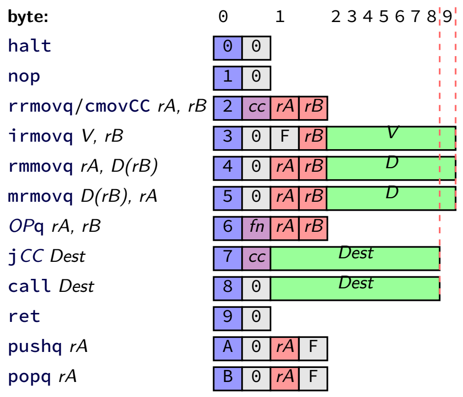
- 2 as cc: `l` (less than)
- hex 84 00… as little endian *Dest*: *0x84*



Y86-64 decoding

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

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



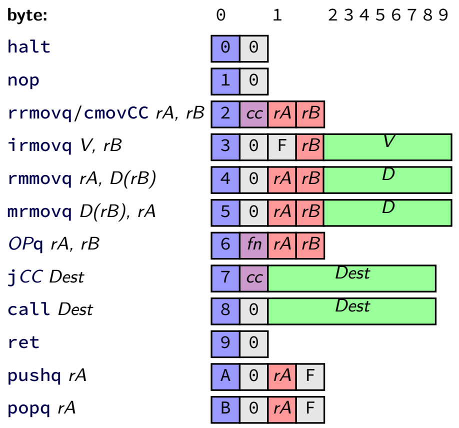
Y86-64: convenience for hardware

4 bits to decode instruction size/layout

(mostly) uniform placement of operands ("uniform decode")

jumping to zeroes (uninitialized?) by accident halts

no attempt to fit (parts of) multiple instructions in a byte



Y86-64

Y86-64: simplified, more RISC-y version of X86-64

minimal set of arithmetic

only movs touch memory

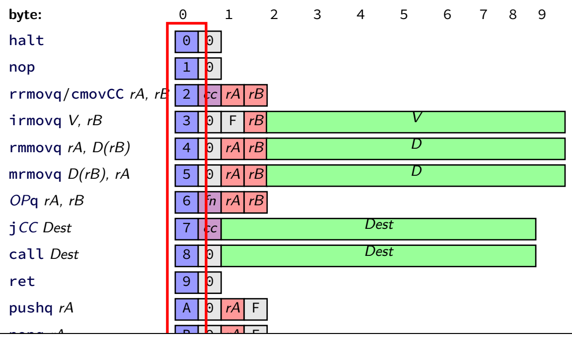
only jumps, calls, and movs take immediates

simple variable-length encoding

later: implementing with circuits

extracting opcodes (1)

```
typedef unsigned char byte;
int get_opcode(byte *instr) {
    return ???;
}
```



extracing opcodes (2)

```
typedef unsigned char byte;
int get_opcode_and_function(byte *instr) {
    return instr[0];
}
/* first byte = opcode * 16 + fn/cc code */
int get_opcode(byte *instr) {
    return instr[0] / 16;
}
```

aside: division

division is really slow

Intel "Skylake" microarchitecture:

- about six cycles per division
- ...and much worse for eight-byte division
- versus: four additions per cycle

aside: division

division is really slow

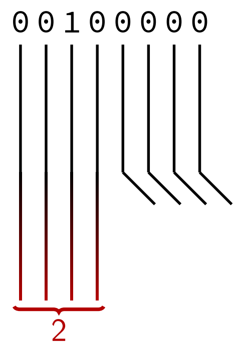
Intel "Skylake" microarchitecture:

- about six cycles per division
- ...and much worse for eight-byte division
- versus: four additions per cycle

but this case: it's just extracting 'top wires' — simpler?

extracting opcode in hardware

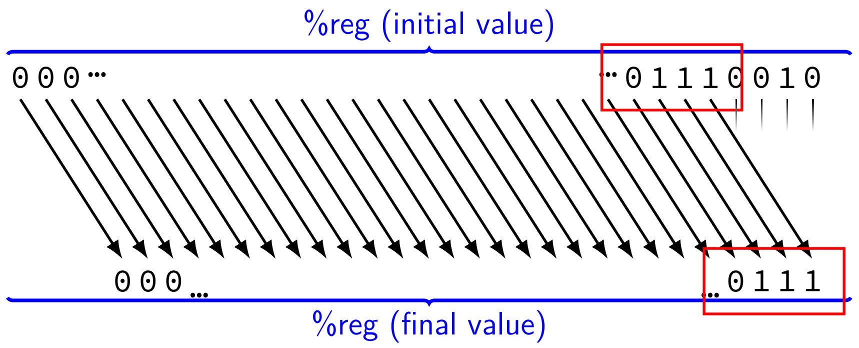
`0111 0010` = 0x72 (first byte of jl)



exposing wire selection

x86 instruction: `shr` — shift right

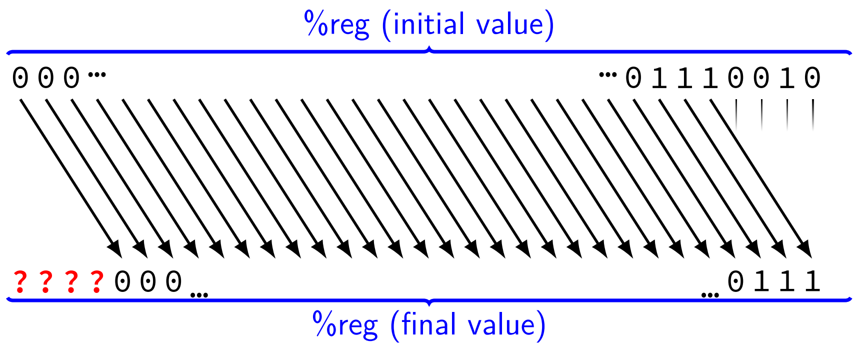
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

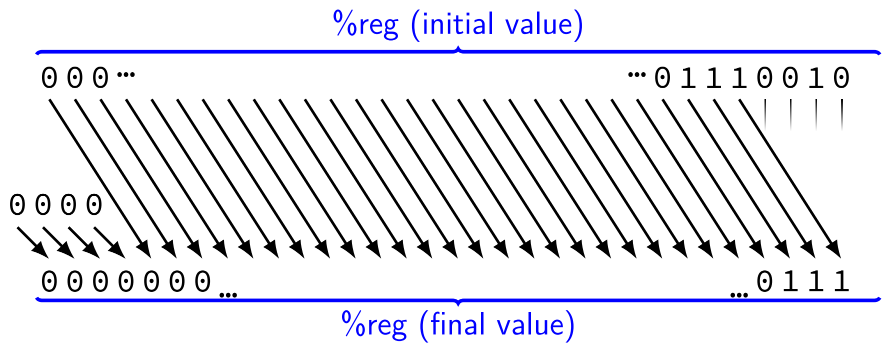
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

`shr $amount, %reg` (or variable: `shr %cl, %reg`)



shift right

x86 instruction: `shr` — shift right

`shr $`*amount*`, %reg`

(or variable: `shr %cl, %reg`)

```
get_opcode:
    // eax ← byte at memory[rdi] with zero padding
    // intel syntax: movzx eax, byte ptr [rdi]
    movzbl (%rdi), %eax
    shrl $4, %eax
    ret
```

shift right

x86 instruction: `shr` — shift right

`shr $`*amount*`, %reg`

(or variable: `shr %cl, %reg`)

```
get_opcode:
    // eax ← byte at memory[rdi] with zero padding
    // intel syntax: movzx eax, byte ptr [rdi]
    movzbl (%rdi), %eax
    shrl $4, %eax
    ret
```

right shift in C

```
get_opcode: // %rdi -- instruction address
    // eax ← one byte of memory[rdi] with zero padd
    // intel syntax: movzx eax, byte ptr [rdi]
    movzbl (%rdi), %eax
    shrl $4, %eax
    ret

typedef unsigned char byte;
int get_opcode(byte *instr) {
    return instr[0] >> 4;
}
```

right shift in C

```
typedef unsigned char byte;
int get_opcode1(byte *instr) { return instr[0] >> 4; }
int get_opcode2(byte *instr) { return instr[0] / 16; }
```

right shift in C

```
typedef unsigned char byte;
int get_opcode1(byte *instr) { return instr[0] >> 4; }
int get_opcode2(byte *instr) { return instr[0] / 16; }
```

example output from optimizing compiler:

```
get_opcode1:
    movzbl (%rdi), %eax
    shrl $4, %eax
    ret

get_opcode2:
    movb (%rdi), %al
    shrb $4, %al
    movzbl %al, %eax
    ret
```

right shift in math

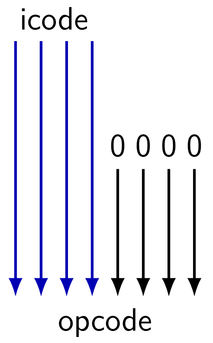
```
1 >> 0 == 1              0000 0001
1 >> 1 == 0              0000 0000
1 >> 2 == 0              0000 0000

10 >> 0 == 10            0000 1010
10 >> 1 == 5             0000 0101
10 >> 2 == 2             0000 0010
```

$$x >> y = \lfloor x \times 2^{-y} \rfloor$$

constructing instructions

```
typedef unsigned char byte;
byte make_simple_opcode(byte icode) {
    // function code is fixed as 0 for now
    return opcode * 16;
}
```

constructing instructions in hardware



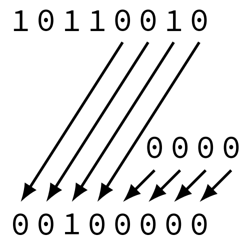
shift left

~~shr $-4, %reg~~

instead: `shl $4, %reg` ("**sh**ift **l**eft")

~~opcode >> (-4)~~

instead: `opcode << 4`



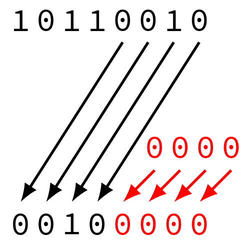
shift left

~~shr $-4, %reg~~

instead: `shl $4, %reg` ("**sh**ift **l**eft")

~~opcode >> (-4)~~

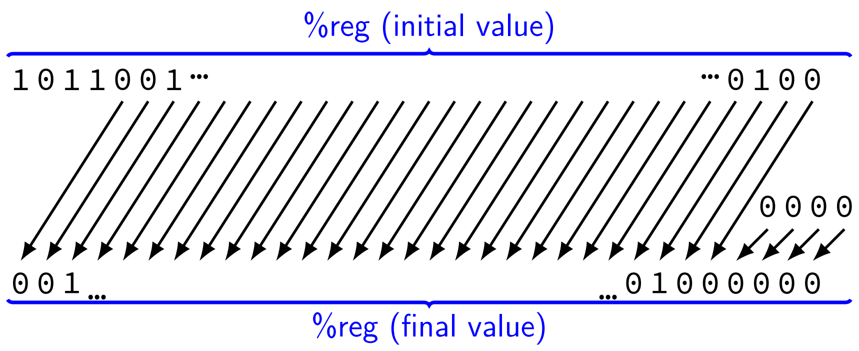
instead: `opcode << 4`



shift left

x86 instruction: `shl` — shift left

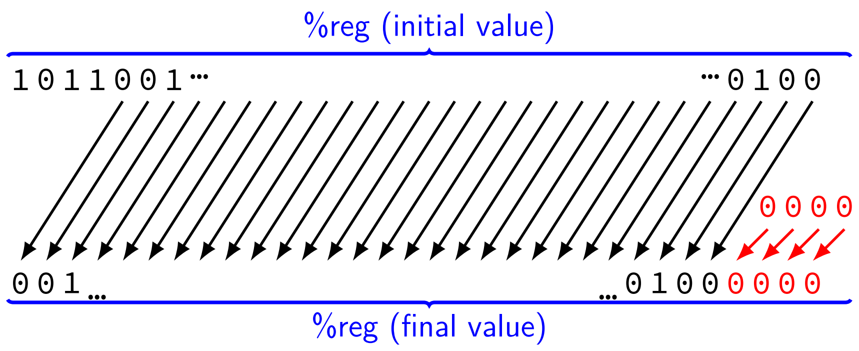
`shl $`*`amount`*`, %reg` (or variable: `shr %cl, %reg`)



shift left

x86 instruction: `shl` — shift left

`shl $`*`amount`*`, %reg` (or variable: `shr %cl, %reg`)



left shift in math

```
1 << 0 == 1            0000 0001
1 << 1 == 2            0000 0010
1 << 2 == 4            0000 0100

10 << 0 == 10          0000 1010
10 << 1 == 20          0001 0100
10 << 2 == 40          0010 1000
```

left shift in math

```
1 << 0 == 1            0000 0001
1 << 1 == 2            0000 0010
1 << 2 == 4            0000 0100

10 << 0 == 10          0000 1010
10 << 1 == 20          0001 0100
10 << 2 == 40          0010 1000
```

$$x \ll y = x \times 2^y$$

backup slides

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

destination

source

SD`movq`

`i` — immediate
`r` — register
`m` — memory

Y86-64: movq

destination

source

SD`movq`

`i` — immediate
`r` — register
`m` — memory

```
irmovq  ~~immovq~~  ~~iimovq~~
rrmovq  rmmovq      ~~rimovq~~
mrmovq  ~~mmmovq~~  ~~mimovq~~
```

Y86-64: movq

destination

source

SD`movq`

`i` — immediate
`r` — register
`m` — memory

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

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)

Y86-64: register-to-register only

instead of:

```
    jle skip_move
    rrmovq %rax, %rbx
skip_move:
    // ...
```

can do:

```
    cmovg %rax, %rbx
```

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

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

much, much simpler encoding

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: condition codes with OF

`subq SECOND, FIRST` (value = FIRST - SECOND)

`j__` or `cmov__`	condition code bit test	value test
`le`	SF $\neq$ OF or ZF $= 0$	value ≤ 0
`l`	SF $\neq$ OF	value < 0
`e`	ZF $= 1$	value $= 0$
`ne`	ZF $= 0$	value $\neq 0$
`ge`	SF $=$ OF or ZF $= 1$	value ≥ 0
`g`	SF $\neq$ OF	value > 0