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)

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)

```c
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)

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

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)

$\text{r12} \leftarrow \text{r12} + 1$

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`	$\mathrm{SF} = 1$ or $\mathrm{ZF} = 1$	value ≤ 0
`l`	$\mathrm{SF} = 1$	value < 0
`e`	$\mathrm{ZF} = 1$	value $= 0$
`ne`	$\mathrm{ZF} = 0$	value $\neq 0$
`ge`	$\mathrm{SF} = 0$	value ≥ 0
`g`	$\mathrm{SF} = 0$ and $\mathrm{ZF} = 0$	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`

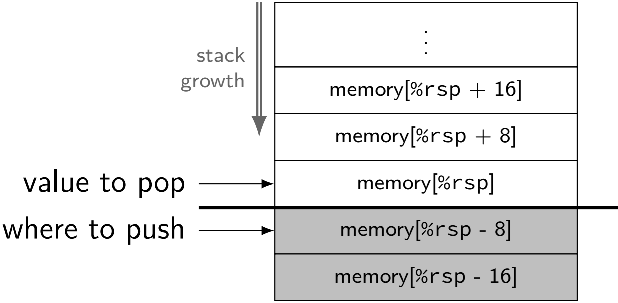
$\texttt{\%rsp} \leftarrow \texttt{\%rsp} - 8$

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

`popq %rbx`

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

$\texttt{\%rsp} \leftarrow \texttt{\%rsp} + 8$



call/ret

call LABEL

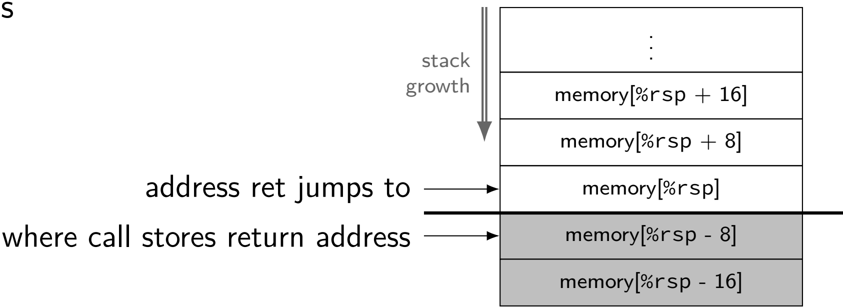
push PC (next instruction address) on stack

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: *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*								
`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								

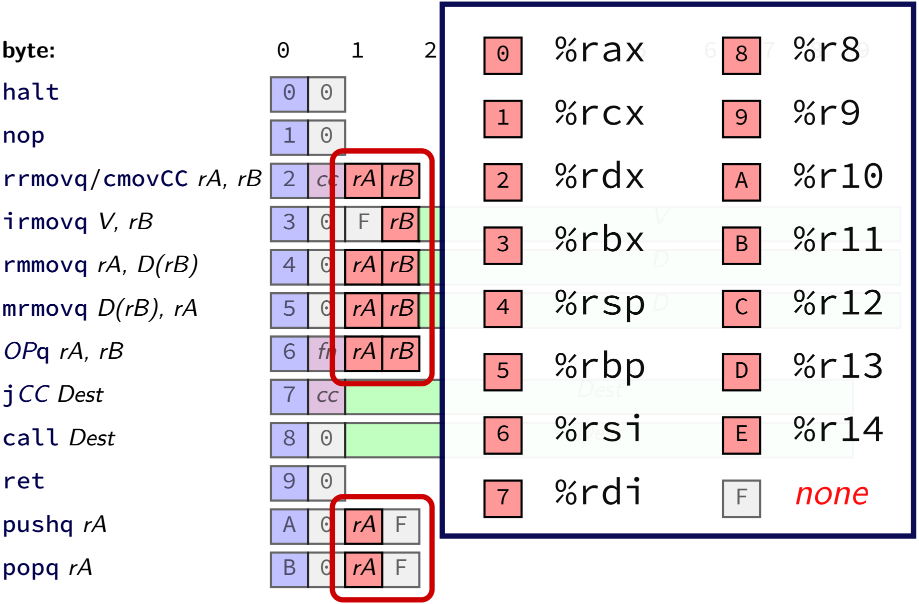
0	add
1	sub
2	and
3	xor

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*

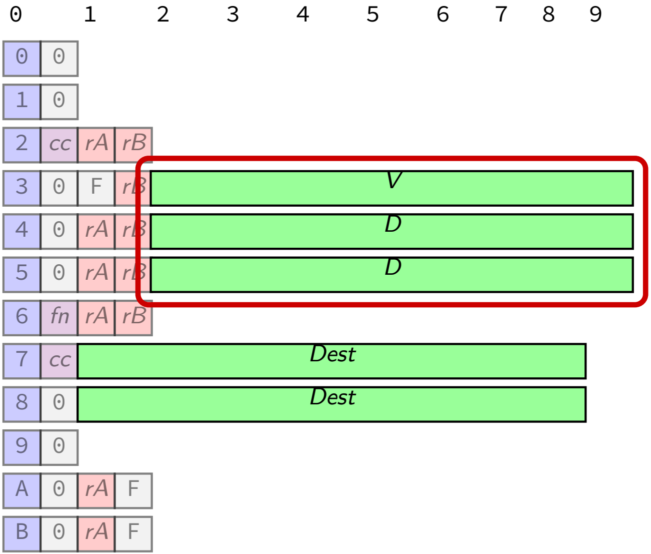
Registers: *rA*, *rB*



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	

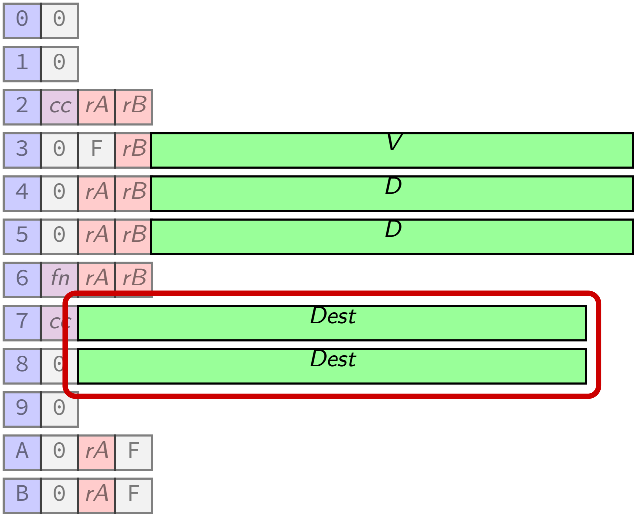
0	`%rax`	8	`%r8`
1	`%rcx`	9	`%r9`
2	`%rdx`	A	`%r10`
3	`%rbx`	B	`%r11`
4	`%rsp`	C	`%r12`
5	`%rbp`	D	`%r13`
6	`%rsi`	E	`%r14`
7	`%rdi`	F	*none*

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								

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:

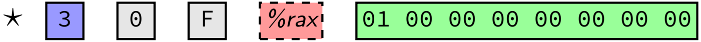
```
    movq %rdi, %rax
    addq $1, %rax
    ret
```

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

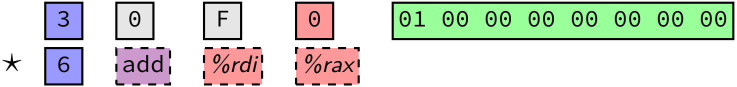
Y86-64 encoding (2)

```
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
```

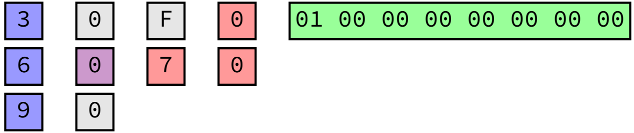
Y86-64 encoding (2)

```
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
```

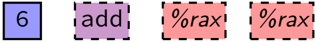


```
3 0 F 0 01 00 00 00 00 00 00 00
6 0 7 0
9 0
```

```
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

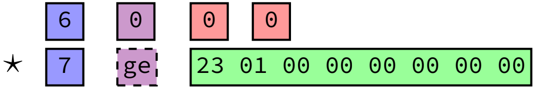
Y86-64 encoding (3)

```
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
```

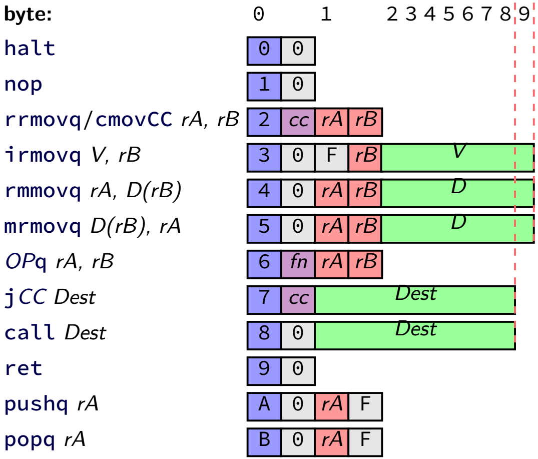
Y86-64 encoding (3)

```
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

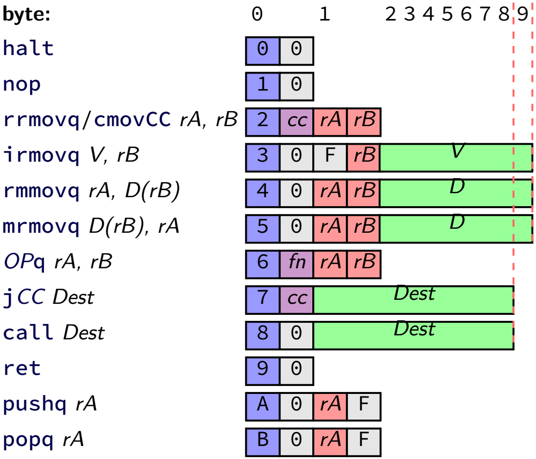
```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```



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	

Y86-64 decoding

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



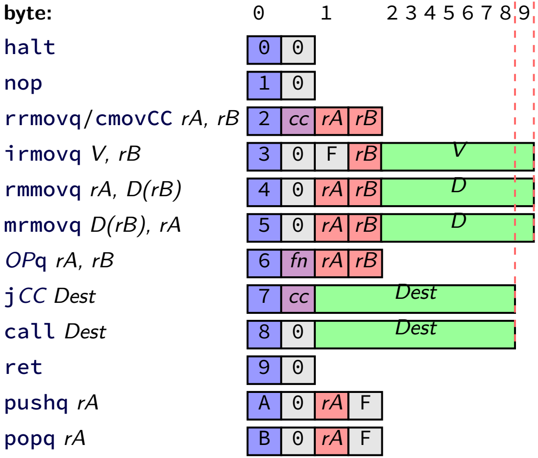
byte:	0		1		2 3 4 5 6 7 8 9
halt	0	0			
nop	1	0			
rrmovq/cmovCC *rA*, *rB*	2	*cc*	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
rmmovq *rA*, *D*(*rB*)	4	0	*rA*	*rB*	*D*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*
*OP*q *rA*, *rB*	6	*fn*	*rA*	*rB*	
j*CC* *Dest*	7	*cc*	*Dest*		
call *Dest*	8	0	*Dest*		
ret	9	0			
pushq *rA*	A	0	*rA*	F	
popq *rA*	B	0	*rA*	F	

Y86-64 decoding

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

`rrmovq %rcx, %rax`

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



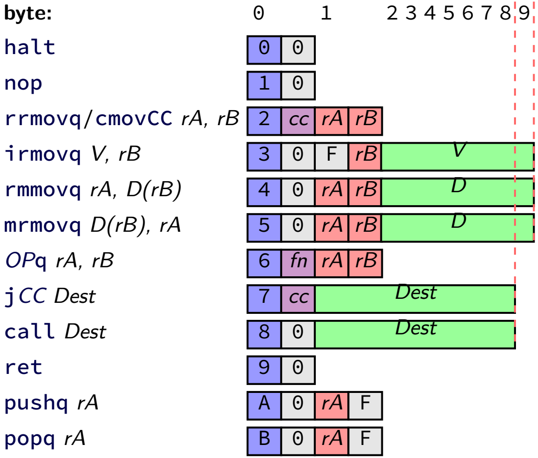
byte:	0	1	2 3 4 5 6 7 8 9
`halt`	0	0	
`nop`	1	0	
`rrmovq/cmovCC` *rA*, *rB*	2	*cc*	*rA* *rB*
`irmovq` *V*, *rB*	3	0	F *rB* *V*
`rmmovq` *rA*, *D*(*rB*)	4	0	*rA* *rB* *D*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA* *rB* *D*
OP`q` *rA*, *rB*	6	*fn*	*rA* *rB*
`j`*CC* *Dest*	7	*cc*	*Dest*
`call` *Dest*	8	0	*Dest*
`ret`	9	0	
`pushq` *rA*	A	0	*rA* F
`popq` *rA*	B	0	*rA* F

Y86-64 decoding

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

```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
```

- 0 as fn: `add`
- 1 as fn: `sub`



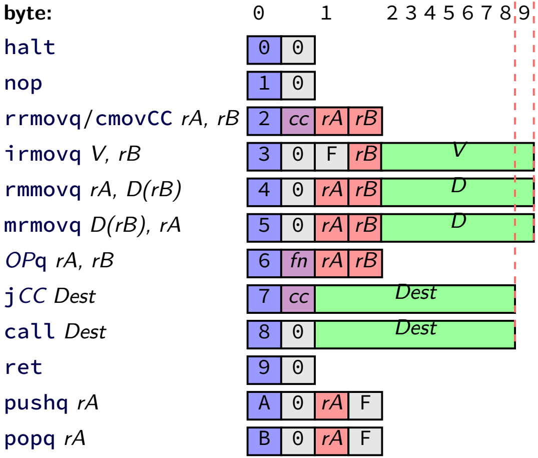
byte:	0		1		2 3 4 5 6 7 8 9
`halt`	0	0			
`nop`	1	0			
`rrmovq/cmovCC` *rA, rB*	2	*cc*	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*	
`j`*CC Dest*	7	*cc*	*Dest*		
`call` *Dest*	8	0	*Dest*		
`ret`	9	0			
`pushq` *rA*	A	0	*rA*	F	
`popq` *rA*	B	0	*rA*	F	

Y86-64 decoding

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

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

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

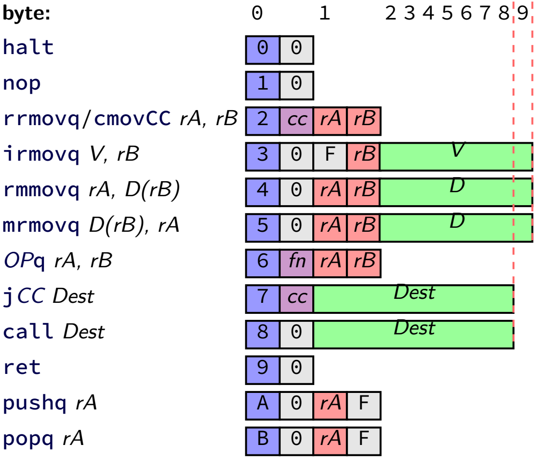


byte:	0		1		2 3 4 5 6 7 8 9
`halt`	0	0			
`nop`	1	0			
`rrmovq/cmovCC` *rA, rB*	2	*cc*	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
OP`q` *rA, rB*	6	*fn*	*rA*	*rB*	
`j`*CC Dest*	7	*cc*	*Dest*		
`call` *Dest*	8	0	*Dest*		
`ret`	9	0			
`pushq` *rA*	A	0	*rA*	F	
`popq` *rA*	B	0	*rA*	F	

Y86-64 decoding

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

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



byte:	0		1		2 3 4 5 6 7 8 9
halt	0	0			
nop	1	0			
rrmovq/cmovCC *rA*, *rB*	2	*cc*	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
rmmovq *rA*, *D*(*rB*)	4	0	*rA*	*rB*	*D*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*
*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	

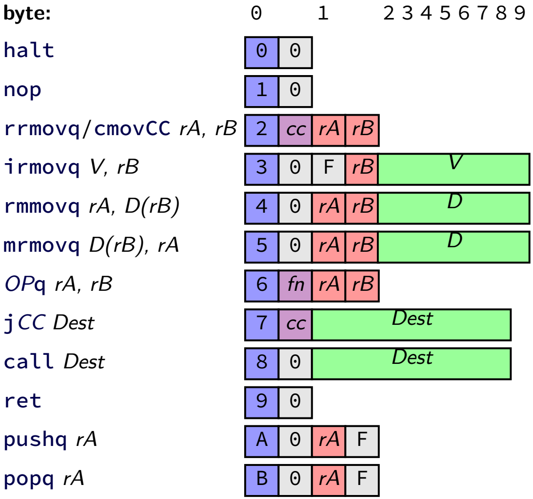
Y86-64: convenience for hardware

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



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	

Y86-64

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

minimal set of arithmetic

only movs touch memory

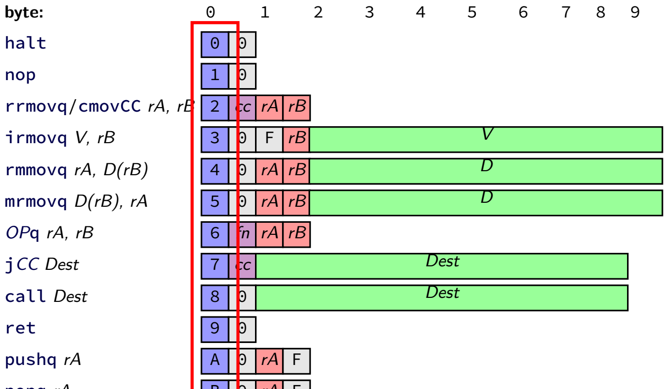
only jumps, calls, and movs take immediates

simple variable-length encoding

later: implementing with circuits

extracting opcodes (1)

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



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								

extracing opcodes (2)

```c
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

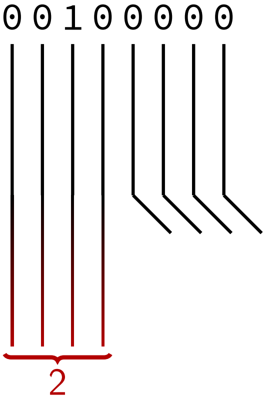
Intel “Skylake” microarchitecture:

- 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

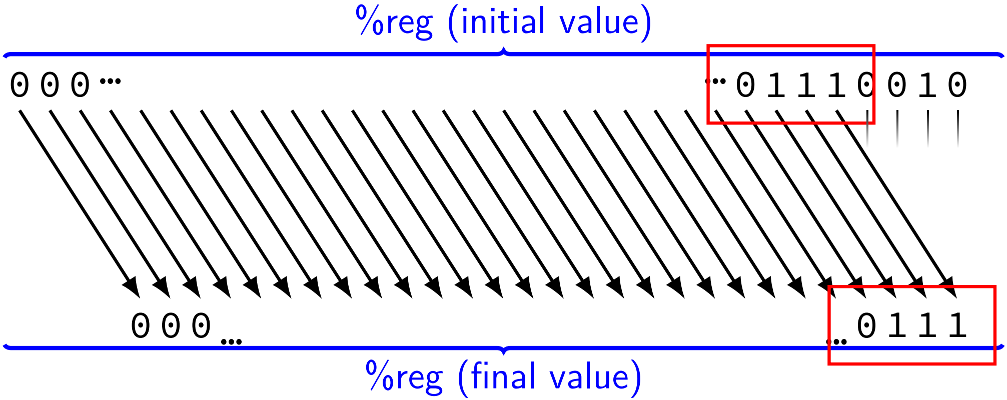
`0111 0010` = 0x72 (first byte of jl)



exposing wire selection

x86 instruction: `shr` — shift right

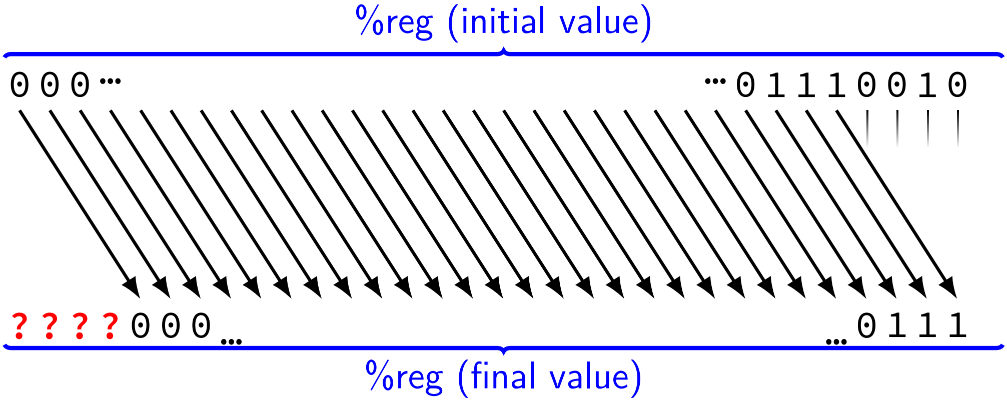
`shr $`*`amount`*`, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

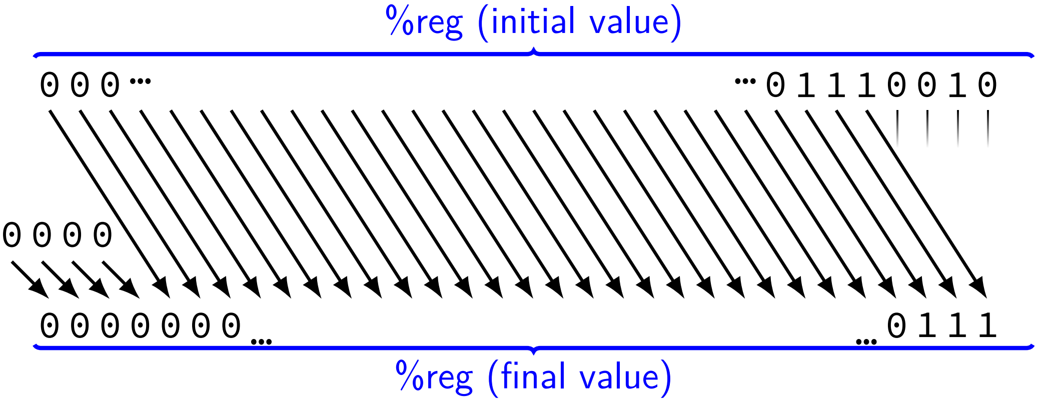
`shr $`*`amount`*`, %reg` (or variable: `shr %cl, %reg`)



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

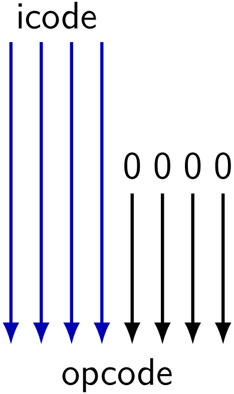
```
1 >> 0 == 1                 0000 0001
1 >> 1 == 0                 0000 0000
1 >> 2 == 0                 0000 0000

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

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

constructing instructions

```c
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



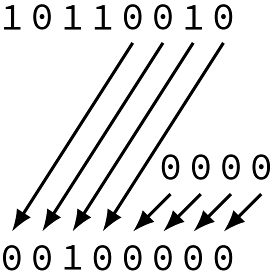
shift left

~~`shr $-4, %reg`~~

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

~~`opcode >> (-4)`~~

instead: `opcode << 4`



```
1 0 1 1 0 0 1 0
        0 0 0 0
0 0 1 0 0 0 0 0
```

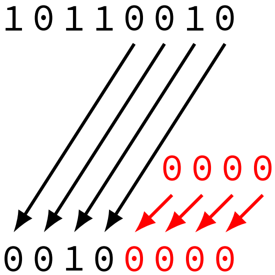
shift left

~~`shr $-4, %reg`~~

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

~~`opcode >> (-4)`~~

instead: `opcode << 4`

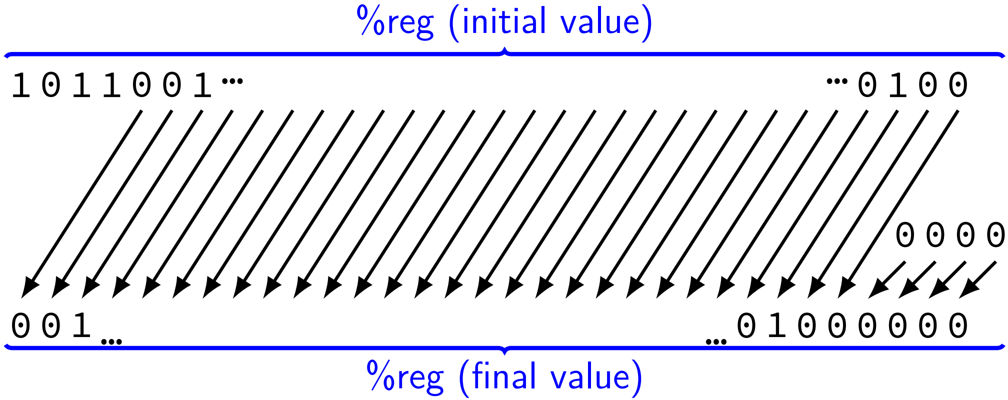


```
1 0 1 1 0 0 1 0
        0 0 0 0
0 0 1 0 0 0 0 0
```

shift left

x86 instruction: `shl` — shift left

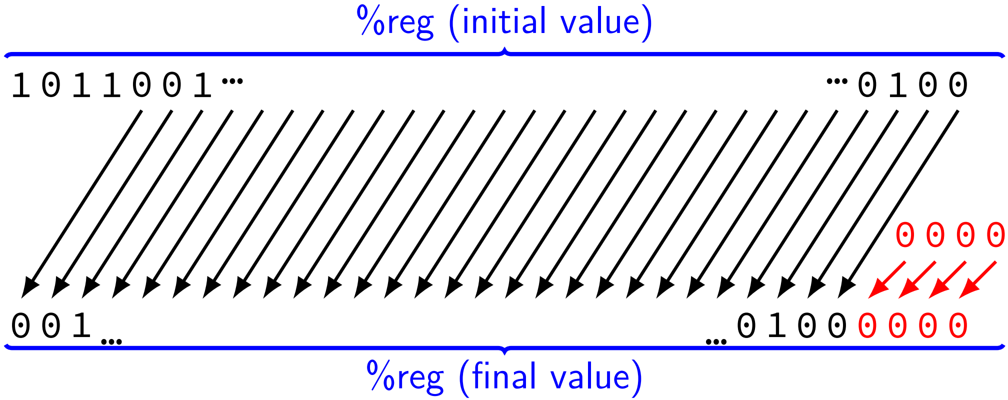
`shl $amount, %reg` (or variable: `shr %cl, %reg`)



shift left

x86 instruction: `shl` — shift left

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



left shift in math

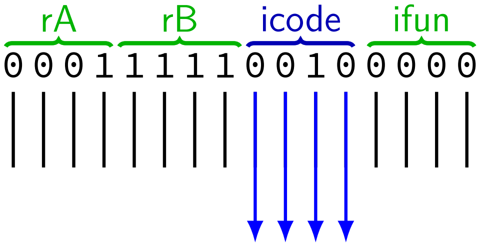
```
1 << 0 == 1                 0000 0001
1 << 1 == 2                 0000 0010
1 << 2 == 4                 0000 0100

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

left shift in math

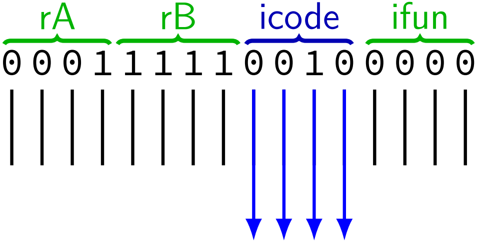
```
1 << 0 == 1                  0000 0001
1 << 1 == 2                  0000 0010
1 << 2 == 4                  0000 0100

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

$$x \texttt{ << } y = x \times 2^y$$

extracting icode from more



extracting icode from more



```
// % -- remainder
unsigned extract_opcode1(unsigned value) {
    return (value / 16) % 16;
}

unsigned extract_opcode2(unsigned value) {
    return (value % 256) / 16;
}
```

manipulating bits?

easy to manipulate individual bits in HW

how do we expose that to software?

interlude: a truth table

AND	**0**	**1**
0	0	0
1	0	1

interlude: a truth table

AND	**0**	**1**
0	0	0
1	0	1

AND with 1: keep a bit the same

interlude: a truth table

AND	**0**	**1**
0	0	0
1	0	1

AND with 1: keep a bit the same

AND with 0: clear a bit

interlude: a truth table

AND	**0**	**1**
0	0	0
1	0	1

AND with 1: keep a bit the same

AND with 0: clear a bit

method: construct "mask" of what to keep/remove

bitwise AND — &

Treat value as array of bits

```
1 & 1 == 1
```

```
1 & 0 == 0
```

```
0 & 0 == 0
```

```
2 & 4 == 0
```

```
10 & 7 == 2
```

bitwise AND — &

Treat value as array of bits

```
1 & 1 == 1

1 & 0 == 0

0 & 0 == 0

2 & 4 == 0

10 & 7 == 2
```

```
        … 0 0 1 0
&       … 0 1 0 0
-----------------
        … 0 0 0 0
```

bitwise AND — &

Treat value as array of bits

```
1 & 1 == 1

1 & 0 == 0

0 & 0 == 0

2 & 4 == 0

10 & 7 == 2
```

$$
\begin{array}{llcccc}
 & \ldots & 0 & 0 & 1 & 0 \\
\& & \ldots & 0 & 1 & 0 & 0 \\
\hline
 & \ldots & 0 & 0 & 0 & 0
\end{array}
$$

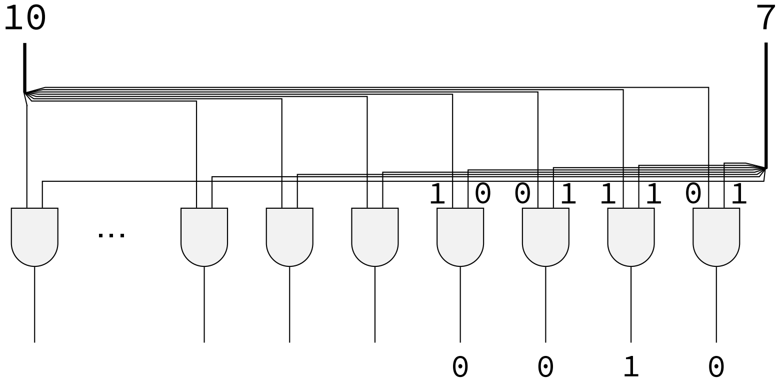
$$
\begin{array}{llcccc}
 & \ldots & 1 & 0 & 1 & 0 \\
\& & \ldots & 0 & 1 & 1 & 1 \\
\hline
 & \ldots & 0 & 0 & 1 & 0
\end{array}
$$

bitwise AND — C/assembly

x86: `and %reg, %reg`

C: `foo & bar`

bitwise hardware (10 & 7 == 2)



extract opcode from larger

```
unsigned extract_opcode1_bitwise(unsigned value) {
    return (value >> 4) & 0xF; // 0xF: 00001111
    // like (value / 16) % 16
}

unsigned extract_opcode2_bitwise(unsigned value) {
    return (value & 0xF0) >> 4; // 0xF0: 11110000
    // like (value % 256) / 16;
}
```

extract opcode from larger

```
extract_opcode1_bitwise:
    movl %edi, %eax
    shrl $4, %eax
    andl $0xF, %eax
    ret

extract_opcode2_bitwise:
    movl %edi, %eax
    andl $0xF0, %eax
    shrl $4, %eax
    ret
```

more truth tables

AND	**0**	**1**
0	0	0
1	0	1

&

conditionally clear bit
conditionally keep bit

OR	**0**	**1**
0	0	1
1	1	1

|

conditionally set bit

XOR	**0**	**1**
0	0	1
1	1	0

^

conditionally flip bit

bitwise OR — |

```
1 | 1 == 1

1 | 0 == 1

0 | 0 == 0

2 | 4 == 6

10 | 7 == 15
```

bitwise OR — |

```
1 | 1 == 1

1 | 0 == 1

0 | 0 == 0

2 | 4 == 6

10 | 7 == 15
```

```
       … 0 0 1 0
|      … 0 1 0 0
----------------
       … 0 1 1 0
```

bitwise OR — |

```
1 | 1 == 1

1 | 0 == 1

0 | 0 == 0

2 | 4 == 6

10 | 7 == 15
```

```
      … 0 0 1 0
|     … 0 1 0 0
---------------
      … 0 1 1 0
```

```
      … 1 0 1 0
|     … 0 1 1 1
---------------
      … 1 1 1 1
```

bitwise xor — ^

```
1 ^ 1 == 0

1 ^ 0 == 1

0 ^ 0 == 0

2 ^ 4 == 6

10 ^ 7 == 13
```

```
      … 0 0 1 0
^     … 0 1 0 0
---------------
      … 0 1 1 0
```

```
      … 1 0 1 0
^     … 0 1 1 1
---------------
      … 1 1 0 1
```

negation / not — ~

~ (‘complement’) is bitwise version of `!`:

`!0 == 1`

`!notZero == 0`

`~0 == (int) 0xFFFFFFFF` (aka -1)

```
~    0  0  …  0  0  0  0
------------------------
     1  1  …  1  1  1  1
```

2 bits

negation / not — ~

~ (‘complement’) is bitwise version of `!`:

`!0 == 1`

`!notZero == 0`

`~0 == (int) 0xFFFFFFFF` (aka -1)

$$\begin{array}{c|cccccccc} \sim & & \texttt{0} & \texttt{0} & \ldots & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{0} \\ \hline & & \texttt{1} & \texttt{1} & \ldots & \texttt{1} & \texttt{1} & \texttt{1} & \texttt{1} \end{array}$$

`~2 == (int) 0xFFFFFFFD` (aka -3)

2 bits

negation / not — ~

~ (‘complement’) is bitwise version of `!`:

```
!0 == 1
```

```
!notZero == 0
```

`~0 == (int) 0xFFFFFFFF` (aka −1)

```
~   0 0 … 0 0 0 0
-----------------
    1 1 … 1 1 1 1
```

`~2 == (int) 0xFFFFFFFD` (aka −3)

2 bits

```
~((unsigned) 2) == 0xFFFFFFFD
```

strategy: mask and shift

construct mask — bits we care about are 1

extract bits with **&**

  or flip with **^**, …

relocate with **<<** or **>>**

combine parts with **|**

note: ternary operator

```
w = (x ? y : z)

if (x) { w = y; } else { w = z; }
```

one-bit ternary

```
(x ? y : z)
```

constraint: everything is 0 or 1

exercise: implement in C without ternary operator or if/else

one-bit ternary

```
(x ? y : z)
```

constraint: everything is 0 or 1

exercise: implement in C without ternary operator or if/else

divide-and-conquer:

```
    (x ? y : 0)
    (x ? 0 : z)
```

one-bit ternary parts (1)

constraint: everything is 0 or 1

```
(x ? y : 0)
```

that's just `(x & y)`

	y=0	**y=1**
x=0	0	0
x=1	0	1

systematically: write out truth table — we've seen it before

one-bit ternary parts (2)

`(x ? y : 0)` $=$ `(x & y)`

one-bit ternary parts (2)

`(x ? y : 0)` = `(x & y)`

`(x ? 0 : z)`

opposite x: `~x`

`((~x) & y)`

one-bit ternary

constraint: everything is 0 or 1 — but y, z is any integer

```
(x ? y : z)
```

```
(x & y) | ((~x) & z)
```

multibit ternary

constraint: x is 0 or 1

```
(x ? y : z)
```

multibit ternary

constraint: x is 0 or 1

```
(x ? y : z)
```

```
(x ? y : 0) | (x ? 0 : z)
```

constructing masks

constraint: x is 0 or 1

`(x ? y : 0)`

if x = 1: want `1111111111`...`1`

if x = 0: want `0000000000`...`0`

one idea: `x | (x << 1) | (x << 2) | ...`

constructing masks

constraint: x is 0 or 1

`(x ? y : 0)`

if x = 1: want `1111111111`…`1`

if x = 0: want `0000000000`…`0`

one idea: `x | (x << 1) | (x << 2) | ...`

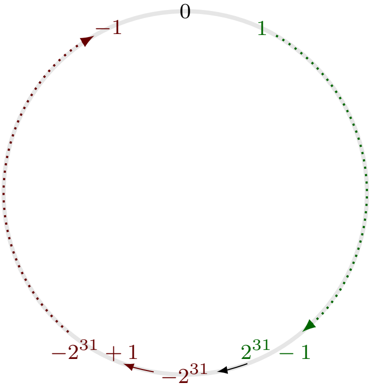
a trick: $-x$

two's complement refresher

$$-1 = \begin{array}{ccccccc} -2^{31} & +2^{30} & +2^{29} & & +2^{2} & +2^{1} & +2^{0} \\ 1 & 1 & 1 & \dots & 1 & 1 & 1 \end{array}$$

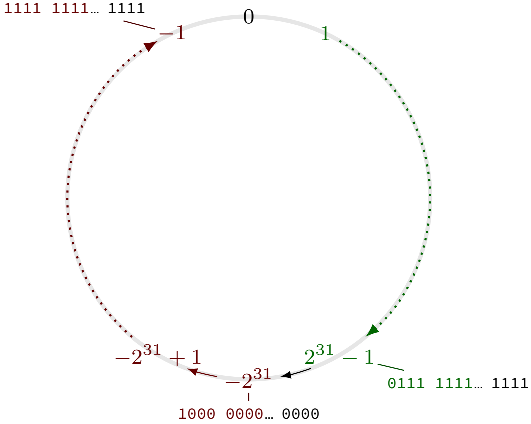
two's complement refresher

$$-1 = \begin{array}{ccccccc} -2^{31} & +2^{30} & +2^{29} & & +2^{2} & +2^{1} & +2^{0} \\ 1 & 1 & 1 & \ldots & 1 & 1 & 1 \end{array}$$



two’s complement refresher

$$\begin{array}{lccccccc} & -2^{31} & +2^{30} & +2^{29} & & +2^{2} & +2^{1} & +2^{0} \\ -1 = & 1 & 1 & 1 & \ldots & 1 & 1 & 1 \end{array}$$



constructing masks

constraint: x is 0 or 1

`(x ? y : 0)`

if x = 1: want `1111111111`…`1`

if x = 0: want `0000000000`…`0`

one idea: `x | (x << 1) | (x << 2) | ...`

a trick: −x

`((-x) & y)`

constructing other masks

constraint: x is 0 or 1

`(x ? 0 : z)`

if x = ~~1~~ 0: want `1111111111`…`1`

if x = ~~0~~ 1: want `0000000000`…`0`

constructing other masks

constraint: x is 0 or 1

`(x ? 0 : z)`

if x = ~~1~~ 0: want `1111111111`...`1`

if x = ~~0~~ 1: want `0000000000`...`0`

flip x first: `(x ^ 1)`

`−(x ^ 1)`

multibit ternary

constraint: x is 0 or 1

```
(x ? y : z)

(x ? y : 0) | (x ? 0 : z)

((-x) & y) | ((-(x ^ 1)) & z)
```

ternary multibit

~~constraint: x is 0 or 1~~

`(x ? y : z)`

trick: `!x` = 0 or 1, `!!x` = 0 or 1

- x86 assembly: `testq %rax, %rax` then `sete`/`setne`

ternary multibit

~~constraint: x is 0 or 1~~

`(x ? y : z)`

trick: `!x` = 0 or 1, `!!x` = 0 or 1

- x86 assembly: `testq %rax, %rax` then `sete/setne`

`((-!!x) & y) | ((-!x) & z)`

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

what if we don't have `!`?

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

what if we don't have `!`?

how do we solve is x is two bits? four bits?

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

what if we don't have `!`?

how do we solve is x is two bits? four bits?

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

wasted work (1)

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

in general: `(x & 1) | (y & 1) == (x | y) & 1`

wasted work (1)

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

in general: `(x & 1) | (y & 1) == (x | y) & 1`

```
(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1
```

wasted work (2)

4-bit any set: `(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1`

performing 4 bitwise ors

…each bitwise or does 4 OR operations

3/4 of bitwise ORs useless — don't use upper bits

any-bit: divide and conquer

four-bit input $x_1x_2x_3x_4$

`(x >> 1) | x` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1y_2y_3y_4$

$y_2 =$ any-of$(x_1x_2) = x_1|x_2$, $y_4 =$ any-of$(x_3x_4) = x_3|x_4$

```
unsigned int any_of_four(unsigned int x) {
    int part_bits = (x >> 1) | x;
    return ((part_bits >> 2) | part_bits) & 1;
}
```

strategy: divide and conquer

two or more calculations in parallel — different parts of integer

use bit shifts + masks to extract each part later

e.g. bitwise OR/AND/XOR — can compute multiple bits

can also apply to addition

any-bit-set: 32 bits

```
unsigned int any_of_four(unsigned int x) {
    x = (x >> 1) | x;
    x = (x >> 2) | x;
    x = (x >> 4) | x;
    x = (x >> 8) | x;
    x = (x >> 16) | x;
    return x & 1;
}
```

bitwise strategies

use paper, etc.

mask and shift

```
    (x & 0xF0) >> 4
```

factor/distribute

```
    (x & 1) | (y & 1) == (x | y) & 1
```

divide and conquer

common subexpression elimination

```
    ((-!!x) & y) | ((-!!x) & z)
    d = !x; return ((-!d) & y) | ((-d) & z)
```

non-power of two arithmetic

```
unsigned times130(unsigned x) {
    return x * 130;
}
```

non-power of two arithmetic

```
unsigned times130(unsigned x) {
    return x * 130;
}
```

```
unsigned times130(unsigned x) {
    return (x << 7) + (x << 1); // x * 128 + x * 2
}
```

non-power of two arithmetic

```
unsigned times130(unsigned x) {
    return x * 130;
}
```

```
unsigned times130(unsigned x) {
    return (x << 7) + (x << 1); // x * 128 + x * 2
}
```

```
times130:
    movl %edi, %eax
    shll $7, %eax
    leal (%rax, %rdi, 2), %eax
    ret
```

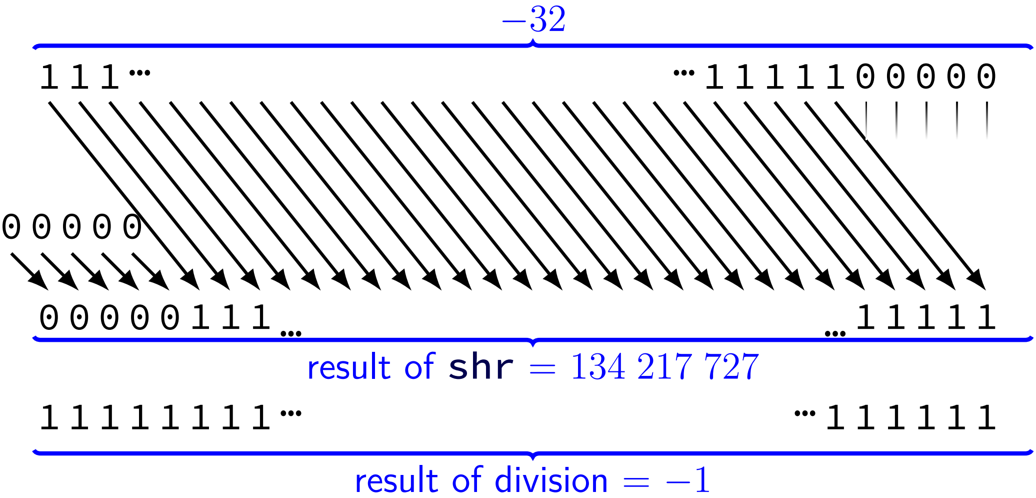
more division

```
int divide_by_32(int x) {
    return x / 32;
}

// INCORRECT generated code
divide_by_32:
    shrl $5, %edi // ← this is WRONG
    mov %edi, %eax
```

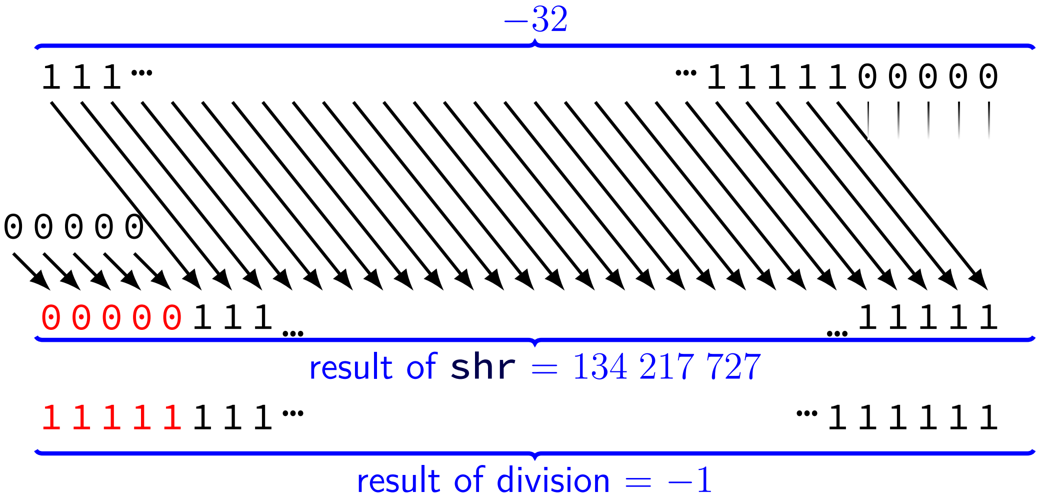
example input with wrong output: -32

exercise: what does this asm output? what is the correct output?

wrong division



wrong division



-32

1 1 1 ··· ··· 1 1 1 1 1 0 0 0 0 0

0 0 0 0 0

0 0 0 0 0 1 1 1 ··· ··· 1 1 1 1 1

result of `shr` $= 134\,217\,727$

1 1 1 1 1 1 1 1 ··· ··· 1 1 1 1 1 1

result of division $= -1$

dividing negative by two

start with $-x$

flip all bits and add one to get x

right shift by one to get $x/2$

flip all bits and add one to get $-x/2$

dividing negative by two

start with $-x$

flip all bits and add one to get x

right shift by one to get $x/2$

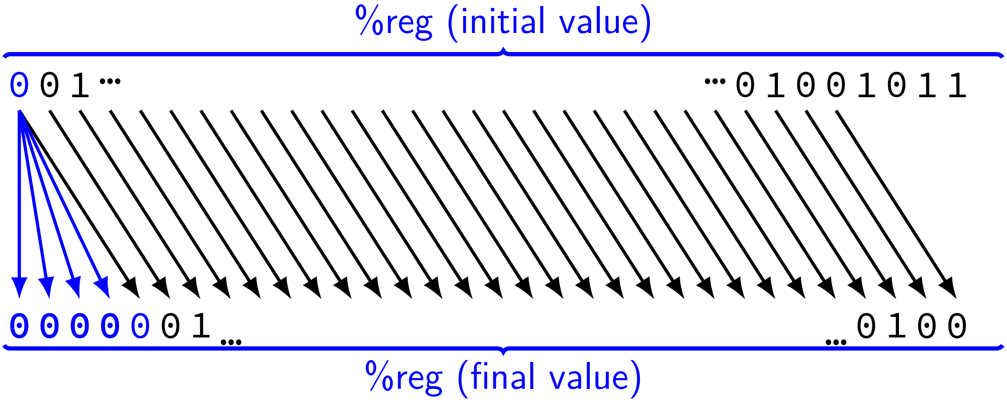
flip all bits and add one to get $-x/2$

same as right shift by one, adding `1`s instead of `0`s
(except for rounding)

arithmetic right shift

x86 instruction: `sra` — arithmetic shift right

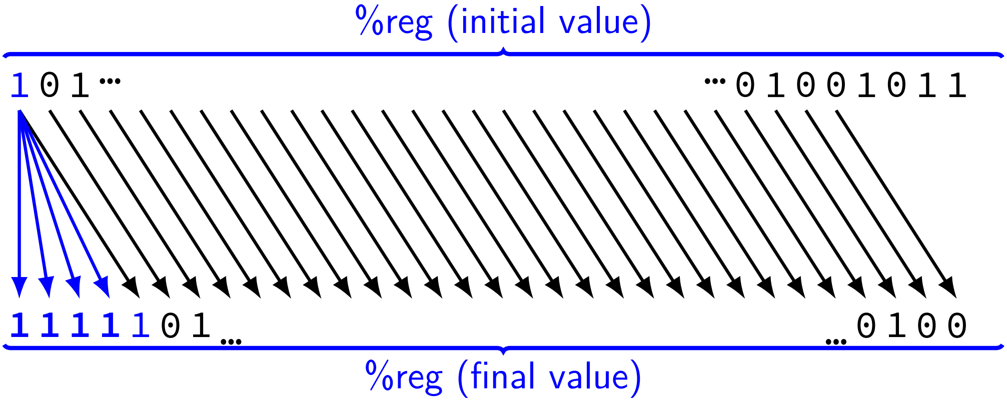
`sra $amount, %reg` (or variable: `sra %cl, %reg`)



arithmetic right shift

x86 instruction: `sra` — arithmetic shift right

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



right shift in C

```c
int divide_32_signed(int x) {
    return x >> 5;
}
unsigned divide_32_unsigned(unsigned x) {
    return x >> 5;
}
```

```
divide_32_signed:   divide_32_unsigned:
    movl %edi, %eax     movl %edi, %eax
    sral $5, %eax       shrl $5, eax
    ret                 ret
```

dividing negative by two

start with $-x$

flip all bits and add one to get x

right shift by one to get $x/2$

flip all bits and add one to get $-x/2$

same as right shift by one, adding `1`s instead of `0`s
(except for rounding)

divide with proper rounding

C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: “bias” adjustments — described in textbook

divide with proper rounding

C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: “bias” adjustments — described in textbook

```
divideBy8: // GCC generated code
    leal   7(%rdi), %eax   // eax ← edi + 7
    testl  %edi, %edi      // set cond. codes based c
    cmovns %edi, %eax      // if (edi sign bit = 0) e
    sarl   $3, %eax        // arithmetic shift
```

standards and shifts in C

signed right shift is implementation-defined

- standard lets compilers choose which type of shift to do
- all x86 compilers I know of — arithmetic

shift amount $\geq$ width of type: undefined

- x86 assembly: only uses lower bits of shift amount

miscellaneous bit manipulation

common bit manipulation instructions are not in C:

rotate (x86: `ror`, `rol`) — like shift, but wrap around

first/last bit set (x86: `bsf`, `bsr`)

population count (some x86: `popcnt`) — number of bits set

bitwise strategies

use paper, etc.

mask and shift

```
(x & 0xF0) >> 4
```

factor/distribute

```
(x & 1) | (y & 1) == (x | y) & 1
```

divide and conquer

common subexpression elimination

```
((-!!x) & y) | ((-!x) & z)
d = !x; return ((-!d) & y) | ((-d) & z)
```

backup slides

Y86-64 instruction set

based on x86

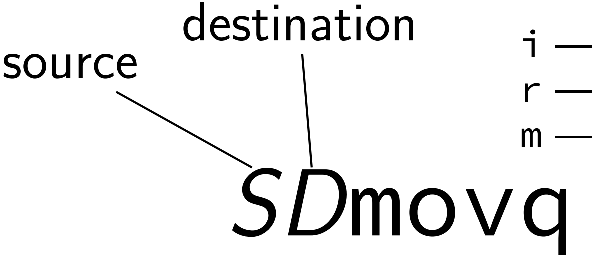
omits most of the 1000+ instructions

leaves

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

much, much simpler encoding

Y86-64: movq



source

destination

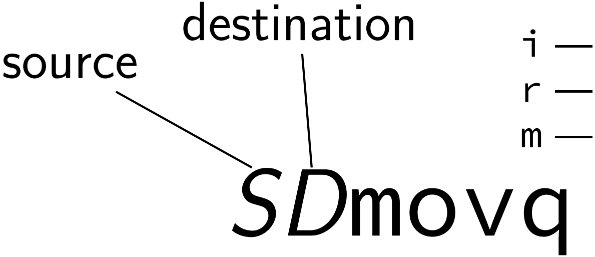
*SD*movq

`i` — immediate

`r` — register

`m` — memory

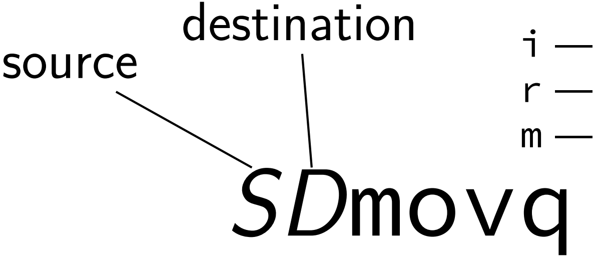
Y86-64: movq



i — immediate
r — register
m — memory

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

Y86-64: movq



source

destination

*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 jCC     popq
andq cmovCC  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