bitwise (part 1)

last time

- object files
  - contain machine code with things missing...
  - relocation table: what to fill in
  - symbol table: what other files might need
- C pointer arithmetic and arrays
  - `foo[x] = *(x+foo)`
  - foo + X uses size of what foo points to
- typedef struct
- undefined behavior
  - C specification says compilers can do whatever
  - signed integer overflow
  - divide-by-zero
  - out-of-bounds pointers

quiz 2 Q1

suppose p is `0x10000`, x is 0x3, *p (memory @ 0x10000) is `0x1`

`*p = x * 2`

result: memory @ 0x10000 becomes 0x3 * 2 = 0x6; p, x unchanged

A. `movq %rax, (%r8); leaq (%r8, 2), %r8`

memory @ 0x10000 (*p) $\leftarrow$ x $=$ 0x3
p $\leftarrow$ p $\times 2 =$ 0x20000
actual result: memory @ 0x10000 becomes 0x3; p also changed

B. `leaq (%rax, 2), %r8; leaq (%r8, 2), %r8`

p $\leftarrow$ x $\times 2 =$ 0x6
p $\leftarrow$ p $\times 2 =$ 0xc
actual result: memory @ 0x10000 unchanged; p changed

quiz 2 Q2

```
start:
    addq %r8, %r9
    cmpq %r9, %r10
    jle start
end:
```

to get to `end`, need `jle` not to jump back to start

when `jle` jumps back to start, last result was ≤ 0, so
(ignoring overflow)

when `jle` does not jump to start, last result was > 0
(ignoring overflow)

ZF $= 1$ if+only if last result $= 0$ ✘

SF $= 1$ if+only if last result is negative ✘

$\implies$ ZF $= 0$, SF $= 0$

quiz 2 Q3

```
start:
    addq %r8, %r9
    cmpq %r9, %r10
    jle start
end:
```

cmpq+jle: `if (r10 - r9 <= 0) goto start`

- sub is subtract from

`do ... while (r10 - r9 <= 0)`

`do ... while (r10 <= r9)`

quiz 2 Q4/5

```
foo:      // <- turns into symbol table entry
          //      pointing to next instruction
    ...
    jmp bar // <- turns into relocation pointing
            //    to jmp + naming bar
    mov $bar, %rdi // <- turns into relocation pointing
                   //    to mov + naming bar
    ...
quux:
    ...
```

symbol table entry:

- there is something called `foo`
- you can find the machine code/data for it in this file at X

relocation table entry:

- please find where `foo` is in memory
- and write that location over the machine code/data in this file at X

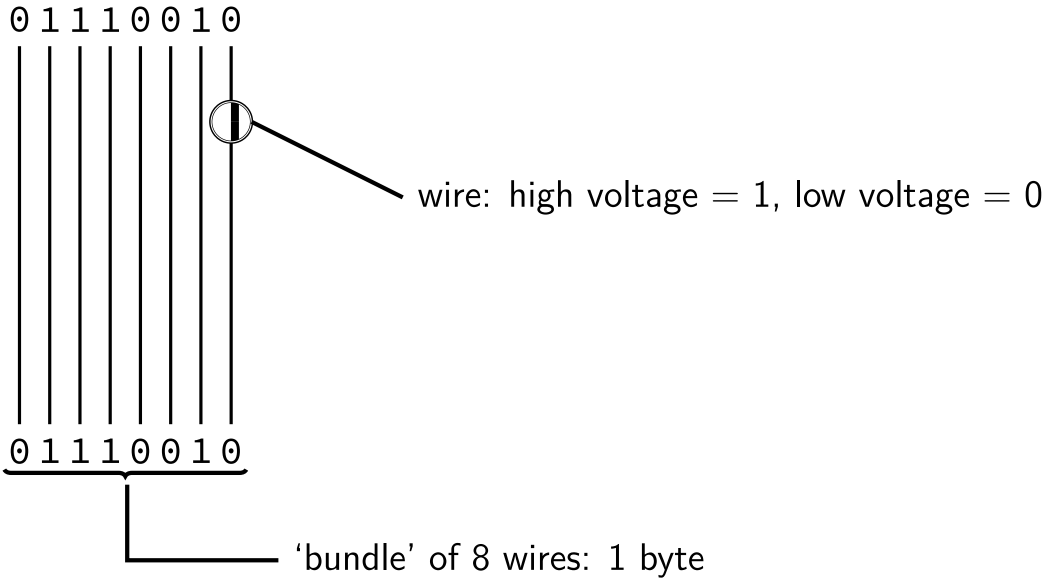
quiz 2 Q4/5 aside

in lecture: showed symbol table showing what labels are defined

often (including on Linux (ELF format .o files)):

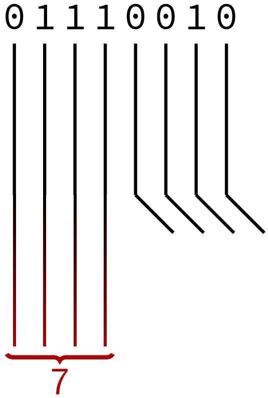
- “blank” symbol table entry for used symbols
- probably to make it easier for linker to know what symbols to look for elsewhere

moving bits in hardware (one way)



extracting bits in hardware

`0111 0010` = 0x72



extracting hexadecimal nibble (1)

problem: given `0xAB` extract `0xA`

(hexadecimal digits called "nibbles")

```
typedef unsigned char byte;
int get_top_nibble(byte value) {
    return ???;
}
```

extracing hexadecimal nibbles (2)

```
typedef unsigned char byte;
int get_top_nibble(byte value) {
    return value / 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:

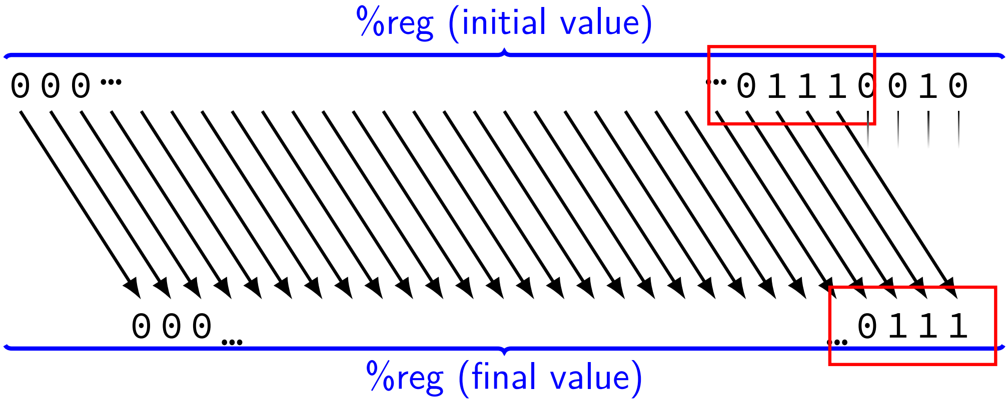
- about six cycles per division
- ...and much worse for eight-byte division
- versus: four additions per cycle

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

exposing wire selection

x86 instruction: `shr` — shift right

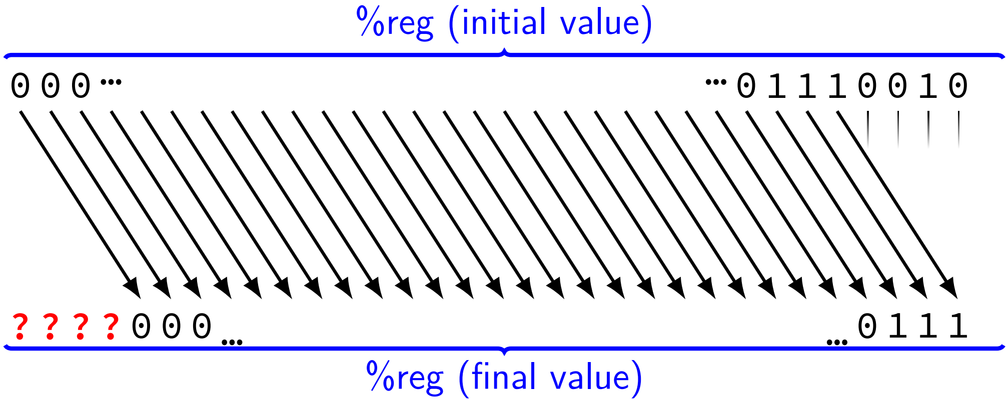
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

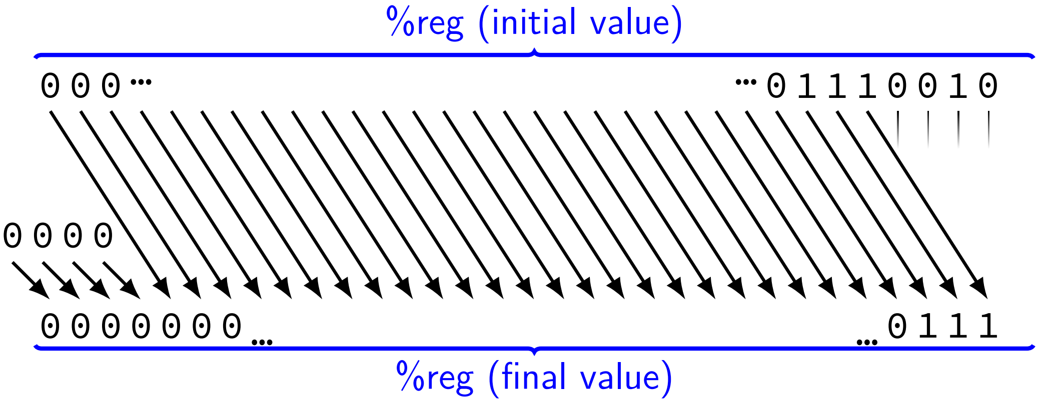
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



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_top_nibble:
  // eax <- dil (low byte of rdi) w/ zero padding
  movzbl %dil, %eax
  shrl $4, %eax
  ret
```

shift right

x86 instruction: `shr` — shift right

`shr $amount, %reg`

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

```
get_top_nibble:
 // eax <- dil (low byte of rdi) w/ zero padding
 movzbl %dil, %eax
 shrl $4, %eax
 ret
```

shift right

x86 instruction: `shr` — shift right

`shr $amount, %reg`

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

```
get_top_nibble:
 // eax <- dil (low byte of rdi) w/ zero padding
 movzbl %dil, %eax
 shrl $4, %eax
 ret
```

right shift in C

```
get_top_nibble:
  // eax <- dil (low byte of rdi) w/ zero padding
  movzbl %dil, %eax
  shrl $4, %eax
  ret
```

```
typedef unsigned char byte;
int get_top_nibble(byte value) {
    return value >> 4;
}
```

right shift in C

```
typedef unsigned char byte;
int get_top_nibble1(byte value) { return value >> 4; }
int get_top_nibble2(byte value) { return value / 16; }
```

right shift in C

```
typedef unsigned char byte;
int get_top_nibble1(byte value) { return value >> 4; }
int get_top_nibble2(byte value) { return value / 16; }
```

example output from optimizing compiler:

```
get_top_nibble1:
    shrb $4, %dil
    movzbl %dil, %eax
    ret

get_top_nibble2:
    shrb $4, %dil
    movzbl %dil, %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
```

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

exercise

```
int foo(int)
```

```
foo:
        movl %edi, %eax
        shrl $1, %eax
        ret
```

what is the value of `foo(-2)`?

A. -4 B. -2 C. -1 D. 0

E. a small positive number F. a large positive number

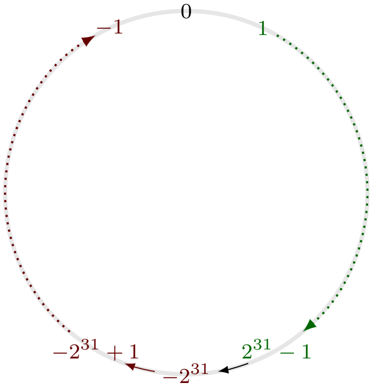
G. a large negative number H. something else

two's complement refresher

	-2^{31}	$+2^{30}$	$+2^{29}$		$+2^{2}$	$+2^{1}$	$+2^{0}$
$-1 =$	1	1	1	...	1	1	1

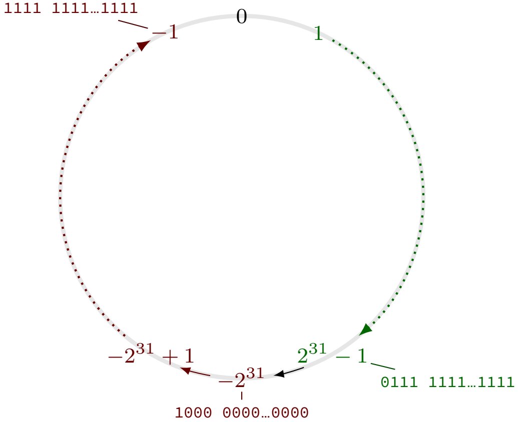
two's complement refresher

	-2^{31}	$+2^{30}$	$+2^{29}$		$+2^{2}$	$+2^{1}$	$+2^{0}$
$-1 =$	1	1	1	...	1	1	1



two's complement refresher

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



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 $x/2$

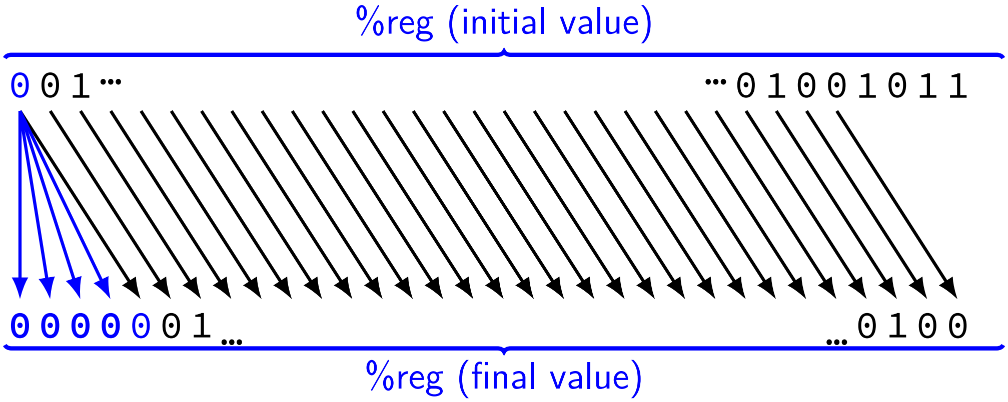
flip all bits and add one to get $-x/2$

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

arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

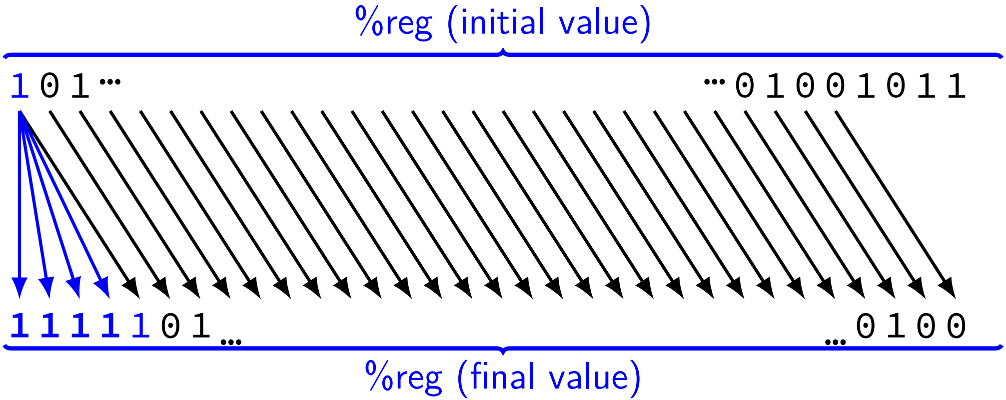
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

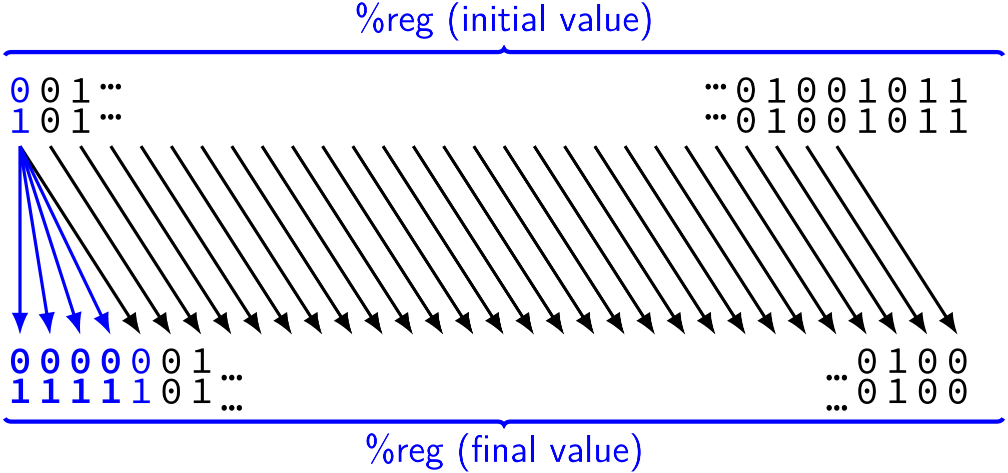
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

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



right shift in C

```
int shift_signed(int x) {
    return x >> 5;
}
unsigned shift_unsigned(unsigned x) {
    return x >> 5;
}
```

```
shift_signed:
    movl %edi, %eax
    sarl $5, %eax
    ret
```

```
shift_unsigned:
    movl %edi, %eax
    shrl $5, eax
    ret
```

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

we'll assume compiler decides arithmetic in this class

shift amount $\geq$ width of type: undefined

- x86 assembly: only uses lower bits of shift amount

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

we'll assume compiler decides arithmetic in this class

shift amount $\geq$ width of type: undefined

- x86 assembly: only uses lower bits of shift amount

exercise

```
int shiftTwo(int x) {
    return x >> 2;
}
```

`shiftTwo(-6) = ???`

A. -4 B. -3 C. -2 D. -1 E. 0

F. some positive number G. something else

explanation

6 =	`000…00000110`
flip bits	`111…11111001`
add one	
-6 =	`111…11111010`
arithmetic shift by 2	**11**111…11111110~~10~~
	`111…111110` (-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)

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

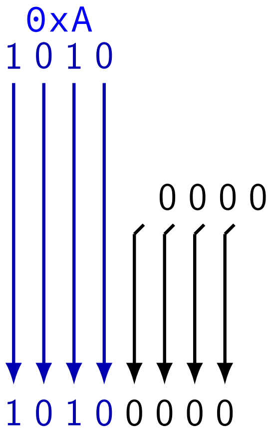
C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: “bias” adjustments — described in textbook

```
// %eax = int divideBy8(int %edi)
divideBy8: // GCC generated code
    leal   7(%rdi), %eax // %eax <- %edi + 7
    testl  %edi, %edi    // set cond. codes based on %edi
    cmovns %edi, %eax    // if (SF == 0) %eax <- %edi
    sarl   $3, %eax      // arithmetic shift
    ret
```

multiplying by 16



$\texttt{0xA} \times 16 = \texttt{0xA0}$

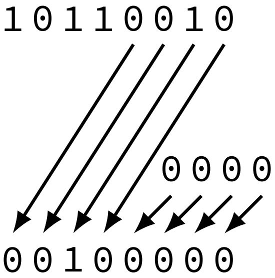
shift left

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

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

~~`value >> (-4)`~~

instead: `value << 4`



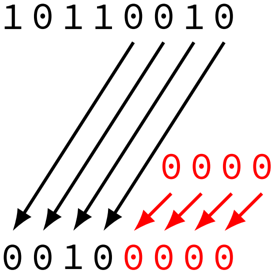
shift left

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

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

~~`value >> (-4)`~~

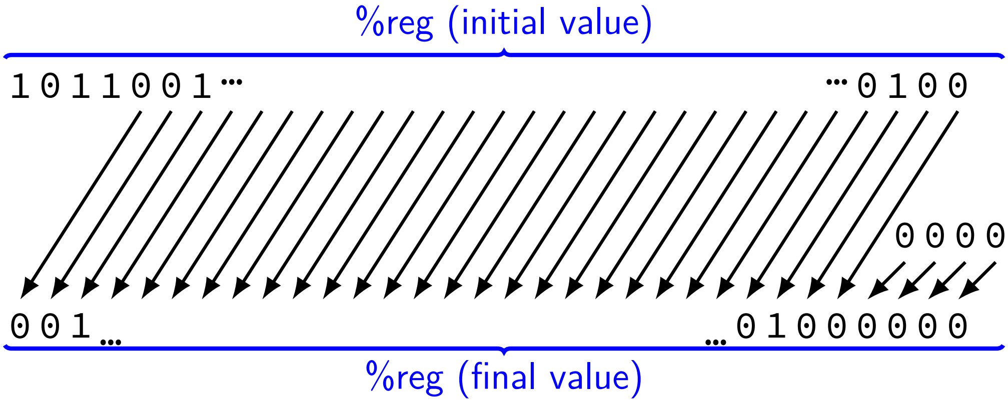
instead: `value << 4`



shift left

x86 instruction: `shl` — shift left

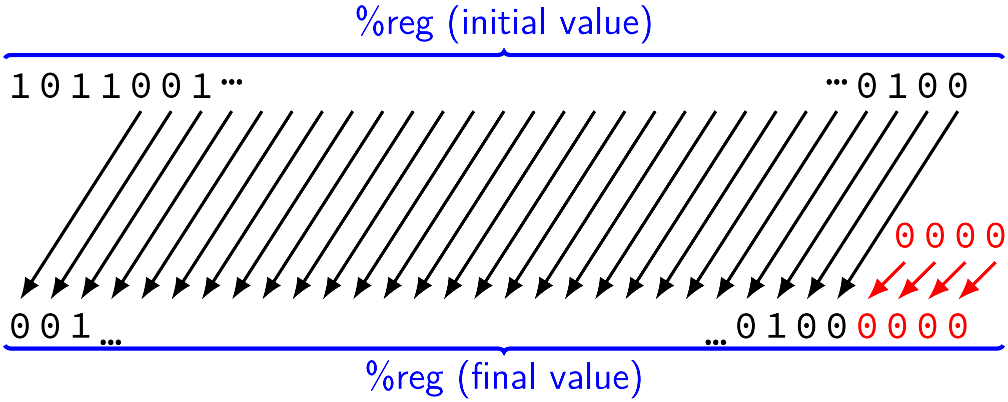
`shl $amount, %reg` (or variable: `shl %cl, %reg`)



shift left

x86 instruction: `shl` — shift left

`shl $amount, %reg` (or variable: `shl %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

-10 << 0 == -10        1111 0110
-10 << 1 == -20        1110 1100
-10 << 2 == -40        1101 1000
```

left shift in math

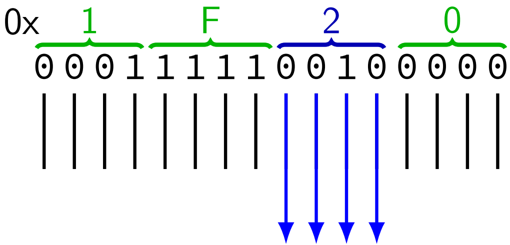
```
1 << 0 == 1            0000 0001
1 << 1 == 2            0000 0010
1 << 2 == 4            0000 0100

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

-10 << 0 == -10        1111 0110
-10 << 1 == -20        1110 1100
-10 << 2 == -40        1101 1000
```

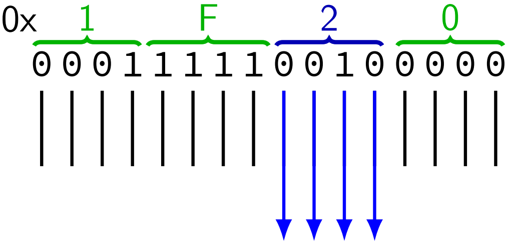
$$x \texttt{<<} y = x \times 2^y$$

extracting nibble from more




```
unsigned
extract_2nd(unsigned value) {
    return ???;
}
```

exercise




```
unsigned
extract_2nd(unsigned value) {
    return ???;
}
```

One idea: `0x1F20` → `0x1F2` → `0x2`.
How can we do each step?

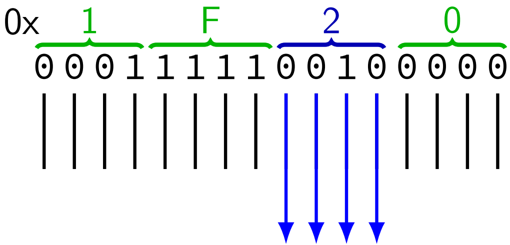
value=0x1F20 → 0x1F2

A. `value >> 16` **B.** `value >> 4` **C.** `value << 2` **D.** `value << 4`

result=0x1F2 → 0x2

A. `result / 256` **B.** `result % 256` **C.** `result / 16`
D. `result % 16` **E.** `result << 4` **F.** `result % 4` **G.** `result / 4`

extracting nibble from more




```
unsigned
extract_2nd(unsigned value) {
    return ???;
}
```

```
// % -- remainder
unsigned extract_second_nibble(unsigned value) {
    return (value >> 4) % 16;
}

unsigned extract_second_nibble(unsigned value) {
    return (value % 256) >> 4;
}
```

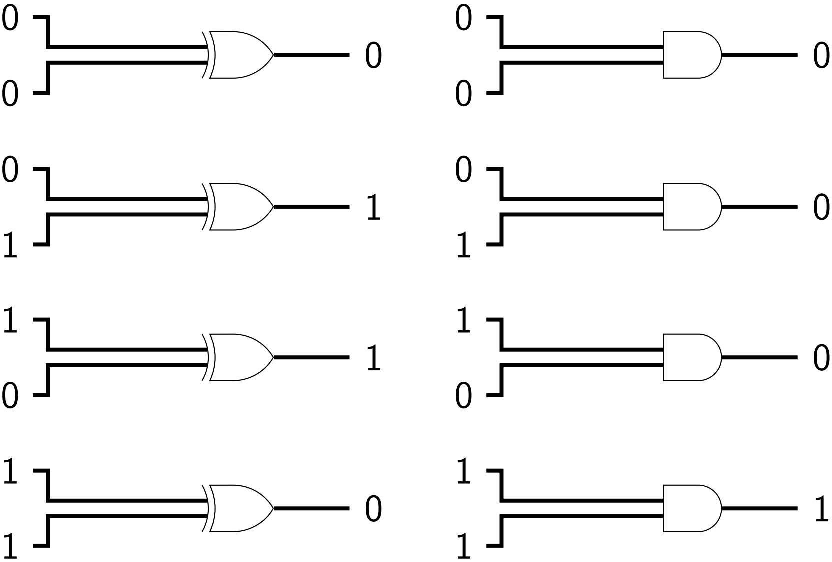
manipulating bits?

easy to manipulate individual bits in HW

- separate wire for each bit
- just ignore/select wires you care about

how do we expose that to software?

circuits: gates



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`

	…	0	0	1	0
&	…	0	1	0	0
	…	0	0	0	0

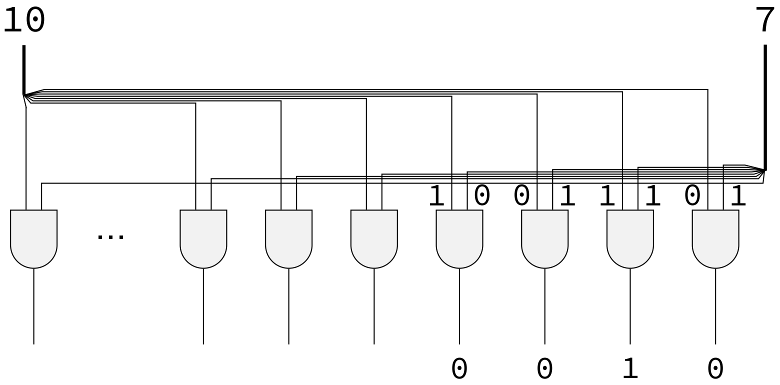
	…	1	0	1	0
&	…	0	1	1	1
	…	0	0	1	0

bitwise AND — C/assembly

x86: `and %reg, %reg`

C: `foo & bar`

bitwise hardware (`10 & 7 == 2`)



extract 0x3 from 0x1234

```
unsigned get_second_nibble1(unsigned value) {
    return (value >> 4) & 0xF; // 0xF: 00001111
    // like (value / 16) % 16
}
```

aaaabbbbccccdddd → aaaabbbbcccc → 00000000cccc

```
unsigned get_second_nibble2(unsigned value) {
    return (value & 0xF0) >> 4; // 0xF0: 11110000
         //     "mask and shift"
    // like (value % 256) / 16;
}
```

aaaabbbbccccdddd → 00000000cccc0000 → 00000000cccc

extract 0x3 from 0x1234

```
get_second_nibble1_bitwise:
    movl %edi, %eax
    shrl $4, %eax
    andl $0xF, %eax
    ret

get_second_nibble2_bitwise:
    movl %edi, %eax
    andl $0xF0, %eax
    shrl $4, %eax
    ret
```

and/or/xor

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

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

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

&

conditionally clear bit

conditionally keep bit

mask: 0s = clear; 1s = keep
e.g. 101010101…=
clear every other bit

|

conditionally set bit

mask: 1s = set; 0s = keep same
e.g. 101010101…=
set every other bit

^

conditionally flip bit

mask: 1s = flip; 0s = keep sam

bitwise OR — |

```
1 | 1 == 1

1 | 0 == 1

0 | 0 == 0

2 | 4 == 6

10 | 7 == 15
```

```
     … 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
```

	…	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)

32 bits

~	0	0	...	0	0	0	0
	1	1	...	1	1	1	1

negation / not — ~

~ ('complement') is bitwise version of `!`:

`!0 == 1`

`!notZero == 0`

`~0 == (int) 0xFFFFFFFF` (aka −1)

`~2 == (int) 0xFFFFFFFD` (aka −3)

32 bits

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

negation / not — ~

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

`!0 == 1`

`!notZero == 0`

`~0 == (int) 0xFFFFFFFF` (aka −1)

`~2 == (int) 0xFFFFFFFD` (aka −3)

32 bits

~	0	0	…	0	0	0	0
	1	1	…	1	1	1	1

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

bit-puzzles

assignments: bit manipulation puzzles

solve some problem with bitwise ops

- maybe that you could do with normal arithmetic, comparisons, etc.

why?

- good for thinking about HW design
- good for understanding bitwise ops
- unreasonably common interview question type

note: ternary operator

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

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

ternary as bitwise: simplifying

`(x ? y : z)` `if (x) return y; else return z;`

task: turn into non-if/else/etc. operators

- assembly: no jumps probably

strategy today: build a solution from simpler subproblems

- (1) with x, y, z 1 bit: `(x ? y : 0)` and `(x ? 0 : z)`
- (2) with x, y, z 1 bit: `(x ? y : z)`
- (3) with x 1 bit: `(x ? y : z)`
- (4) `(x ? y : z)`

one-bit ternary

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

constraint: *x, y, and z are 0 or 1*

now: reimplement in C without if/else/||/etc.

(assembly: no jumps probably)

one-bit ternary

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

constraint: *x, y, and z are 0 or 1*

now: reimplement in C without if/else/||/etc.

(assembly: no jumps probably)

divide-and-conquer:

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

one-bit ternary parts (1)

constraint: *x, y, and z are 0 or 1*

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

one-bit ternary parts (1)

constraint: *x, y, and z are 0 or 1*

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

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

$\rightarrow$ `(x & y)`

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) & z)`

one-bit ternary

constraint: *x, y, and z are 0 or 1*

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

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

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

multibit ternary

constraint: `x` *is 0 or 1*

old solution `((x & y) | (~x) & z)` only gets least sig. bit

`(x ? y : z)`

multibit ternary

constraint: `x` *is 0 or 1*

old solution `((x & y) | (~x) & z)` only gets least sig. bit

`(x ? y : z)`

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

constructing masks

constraint: `x` *is 0 or 1*

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

turn into `y & MASK`, where MASK = ???

"keep certain bits"

constructing masks

constraint: `x` *is 0 or 1*

`(x ? y : 0)`

turn into `y & MASK`, where MASK = ???

"keep certain bits"

if x = 1: want `1111111111`...1 (keep `y`)

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

constructing masks

constraint: x *is 0 or 1*

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

turn into `y & MASK`, where MASK = ???

"keep certain bits"

if x = 1: want `1111111111`...`1` (keep y)

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

a trick: −x (`-1` is `1111`...`1`)

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`

mask: ~~`-x`~~

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`

mask: ~~`-x`~~ `-(x^1)`

multibit ternary

constraint: `x` *is 0 or 1*

old solution `((x & y) | (~x) & z)` only gets least sig. bit

`(x ? y : z)`

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

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

fully multibit

~~constraint: x is 0 or 1~~

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

fully multibit

~~constraint: x is 0 or 1~~

`(x ? y : z)`

easy C way: `!x` $=$ 1 (if $x = 0$) or 0, `!(!x)` $=$ 0 or 1

- x86 assembly: `testq %rax, %rax` then `sete/setne` (copy from ZF)

fully multibit

~~constraint: x is 0 or 1~~

`(x ? y : z)`

easy C way: `!x` = 1 (if $x = 0$) or 0, `!(!x)` = 0 or 1

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

(copy from ZF)

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

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

backup slides

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub   $8, %rsp
    mov   $.Lstr, %rdi
    call  puts
    xor   %eax, %eax
    add   $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.s (Intel syntax)

```
.text
main:
  sub RSP, 8
  mov RDI, .Lstr
  call puts
  xor EAX, EAX
  add RSP, 8
  ret

.data
.Lstr: .string "Hello, Wor
```

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

Linux x86-64 calling convention: stack addr. must be multiple of 16

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

sets eax to 0 (shorter machine code than mov)

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

mark used by other files

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub   $8, %rsp
    mov   $.Lstr, %rdi
    call  puts
    xor   %eax, %eax
    add   $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
     .text
     .global main
main:
     sub  $8, %rsp
     mov  $.Lstr, %rdi
     call puts
     xor  %eax, %eax
     add  $8, %rsp
     ret

     .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub   $8, %rsp
    mov   $.Lstr, %rdi
    call  puts
    xor   %eax, %eax
    add   $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at	*and replace with addr. of*
text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	`puts`

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
     .text
     .global main
main:
     sub   $8, %rsp
     mov   $.Lstr, %rdi
     call  puts
     xor   %eax, %eax
     add   $8, %rsp
     ret

     .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3

data segment:

48 65 6C 6C 6F 2C 20 57 6F 72 6C 00

relocations:

take 0s at	*and replace with addr. of*
text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	puts

symbol table:

main	text byte 0

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
        .text
        .global main
main:
        sub   $8, %rsp
        mov   $.Lstr, %rdi
        call  puts
        xor   %eax, %eax
        add   $8, %rsp
        ret

        .data
.Lstr:  .string "Hello, World!"
```

hello.o

text (code) segment:

48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3

data segment:

48 65 6C 6C 6F 2C 20 57 6F 72 6C 00

relocations:

take 0s at	*and replace with addr. of*
text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	puts

symbol table:

main	text byte 0

.Lstr location specified w/o name
and not usable by other files
so no symbol table entry needed

(convention: .L...labels always local)

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
     .text
     .global main
main:
     sub   $8, %rsp
     mov   $.Lstr, %rdi
     call  puts
     xor   %eax, %eax
     add   $8, %rsp
     ret

     .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at	*and replace with addr. of*
text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	puts

symbol table:

main	text byte 0

+ stdio.o

hello.exe

(actually binary, but shown as hexadecimal) …

```
48 83 EC 08 BF A7 02 04 00
E8 08 4A 00 00 31 C0 48
83 C4 08 C3 …
…(code from stdio.o) …
48 65 6C 6C 6F 2C 20 57 6F
72 6C 00 …
…(data from stdio.o) …
```

hello.s

```
        .section        .rodata.str1.1,"aMS",@progb
.LC0:
        .string "Hello, World!"
        .text
        .globl  main
main:
        subq    $8, %rsp
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        addq    $8, %rsp
        ret
```