More C / Bitwise

Changelog

Changes made in this version not seen in first lecture:

- 6 September: shr slides: correct typo 'valuaction' to instruction
- 6 September: switch to jump table: add 0 case to remove off-by-one error

last time

condition codes (ZF, SF, CF, OF)

- what jle, jge, etc. use
- set by all arithmetic, not just `cmp`
- CF, OF — wrong result as unsigned/signed?

`jmp *0x1234(%rax, %rbx, 8)`

- read memory[0x1234 + rax + rbx * 8]; set PC to read value

loops/if to C

pointer arithmetic in C

- `ptr + X` — add X times sizeof(thing ptr points to) to ptr
- `*(ptr + X) == ptr[X]`
- `ptr = array` same as `ptr = &array[0]`

switch to jump-table

```
switch (a) {
    case 0: ...; break;
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 100: ...; break;
    default: ...
}
```

```
// jump table
cmpq $100, %rax
jg code_for_default
cmpq $0, %rax
jl code_for_default
jmp *table(,%rax,8)
    // same jmp *0x1234(,%rax,8)
    // where 0x1234 = table addr.
```

```
table:
  // not instructions
  // .quad = 64-bit (4 x 16) c
  .quad code_for_0
  .quad code_for_1
  .quad code_for_2
  .quad code_for_3
  .quad code_for_4
    ...
```

quizzes

linked off course website (at top “Quizzes”)

90 minute time limit

box turns green when answer recorded

- no submit button

yellow box — network problem

quiz demo

https://archimedes.cs.virginia.edu/cs3330/
quizzes-demo/

exercise

```
char foo[4] = "foo";
     // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

Final value of foo?

A. "fao"

B. "zao"

C. "baz"

D. "bao"

E. something else/crash

exercise

```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

Final value of `foo`?

A. "fao"

B. "zao"

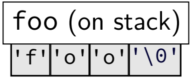
C. "baz"

D. "bao"

E. something else/crash

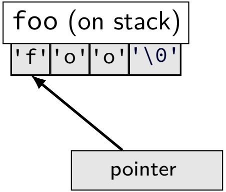
exercise explanation

```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



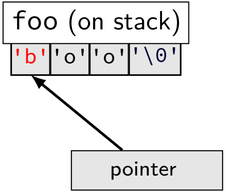
exercise explanation

```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



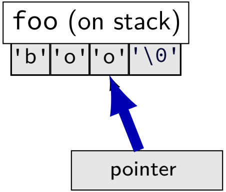
exercise explanation

```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



exercise explanation

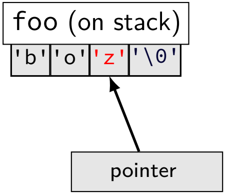
```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



exercise explanation

```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

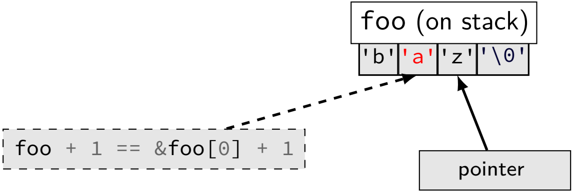


exercise explanation

```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`



a note on precedence

`&foo[1]` is the same as `&(foo[1])` (*not* `(&foo)[1]`)

`*foo[0]` is the same as `*(foo[0])` (*not* `(*foo)[0]`)

`*foo++` is the same as `*(foo++)` (*not* `(*foo)++`)

arrays: not quite pointers (1)

```
int array[100];
int *pointer;
```

Legal: `pointer = array;`

same as `pointer = &(array[0]);`

arrays: not quite pointers (1)

```
int array[100];
int *pointer;
```

Legal: `pointer = array;`

same as `pointer = &(array[0]);`

Illegal: ~~`array = pointer;`~~

arrays: not quite pointers (2)

```
int array[100];
int *pointer = array;
```

```
sizeof(array) == 400
```

size of all elements

arrays: not quite pointers (2)

```
int array[100];
int *pointer = array;
```

```
sizeof(array) == 400
```

size of all elements

```
sizeof(pointer) == 8
```

size of address

arrays: not quite pointers (2)

```
int array[100];
int *pointer = array;
```

```
sizeof(array) == 400
```

size of all elements

```
sizeof(pointer) == 8
```

size of address

```
sizeof(&array[0]) == ???
```

(`&array[0]` same as `&(array[0])`)

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

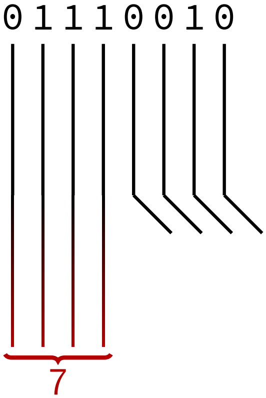
Intel "Skylake" microarchitecture:

- 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?

extracting bits in hardware

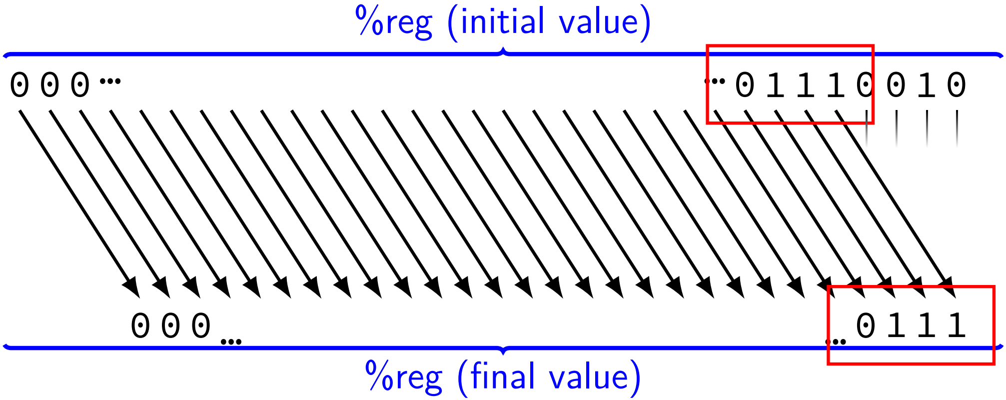
0111 0010 = 0x72



exposing wire selection

x86 instruction: `shr` — shift right

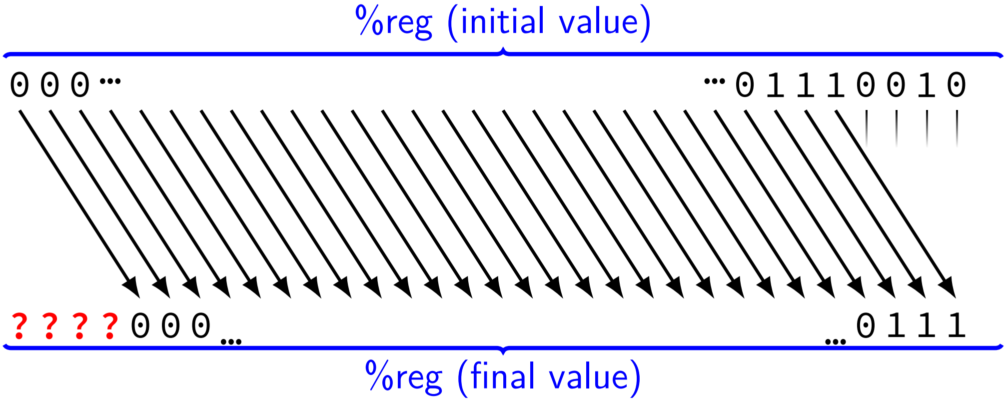
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

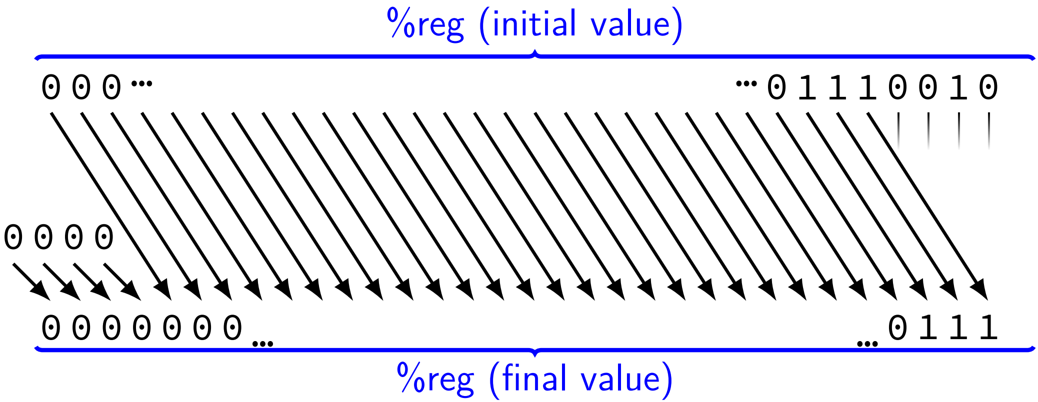
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



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) with zero paddir
    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) with zero paddir
    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) with zero paddir
    movzbl %dil, %eax
    shrl $4, %eax
    ret
```

right shift in C

```
get_top_nibble:
    // eax ← dil (low byte of rdi) with zero paddir
    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
```

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

exercise

```
int foo(int)
foo:
        shrl $1, %eax
        ret
```

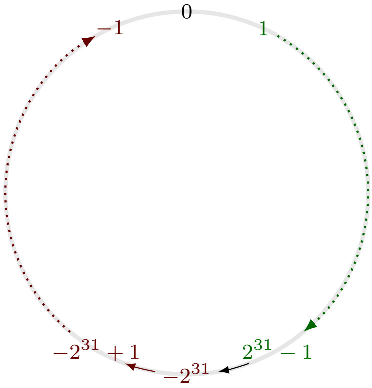
what is the value of `foo(-2)`?

two's complement refresher

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

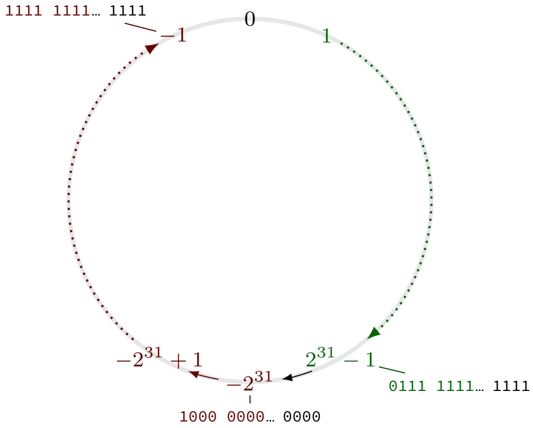
two's complement refresher

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



two's complement refresher

$$-1 = \overset{-2^{31}}{1}\ \overset{+2^{30}}{1}\ \overset{+2^{29}}{1}\ \dots\ \overset{+2^{2}}{1}\ \overset{+2^{1}}{1}\ \overset{+2^{0}}{1}$$



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 $x/2$

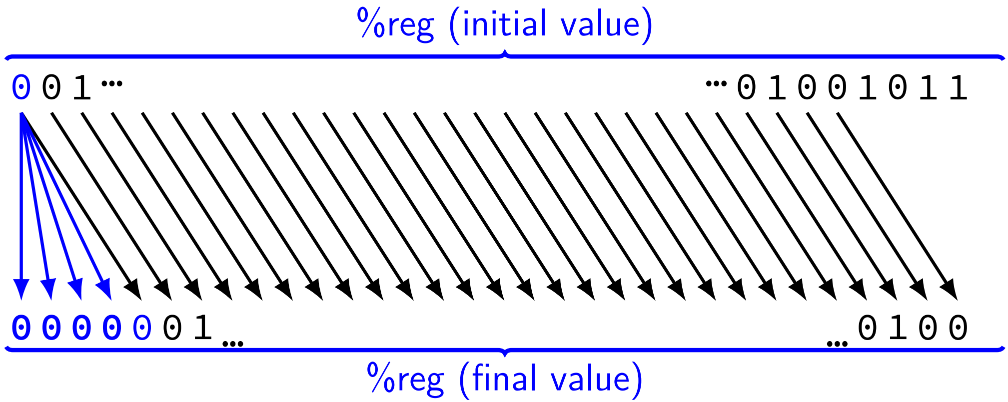
flip all bits and add one to get $-x/2$

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

arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

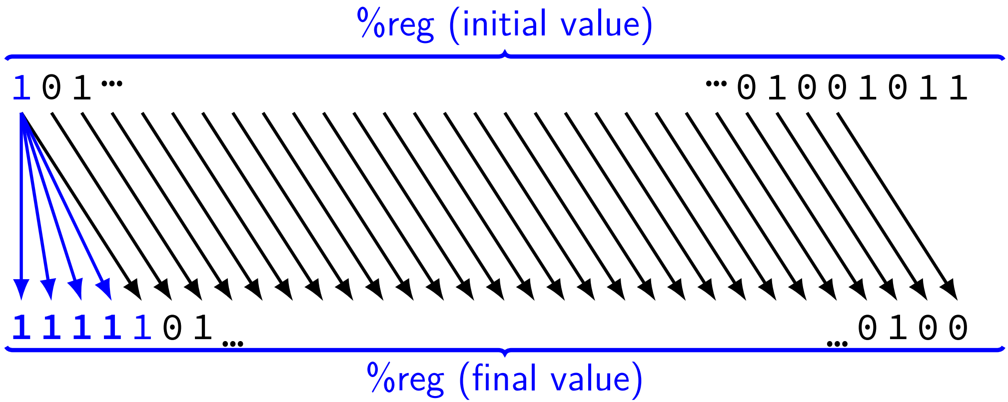
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



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

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) = ???
```

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

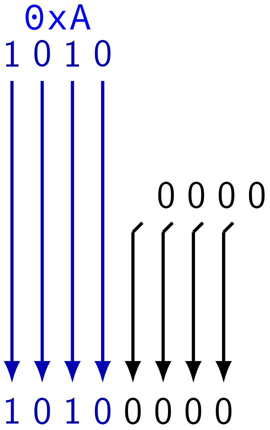
C division: rounds towards zero (truncate)

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 on %edi
    cmovns %edi, %eax     // if (edi sign bit = 0) eax ← edi
    sarl   $3, %eax       // arithmetic shift
```

multiplying by 16



$0xA \times 16 = 0xA0$

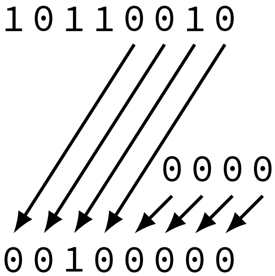
shift left

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

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

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

instead: `value << 4`



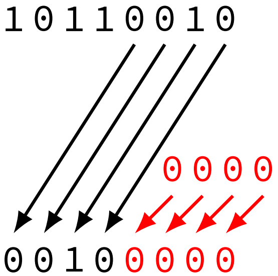
shift left

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

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

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

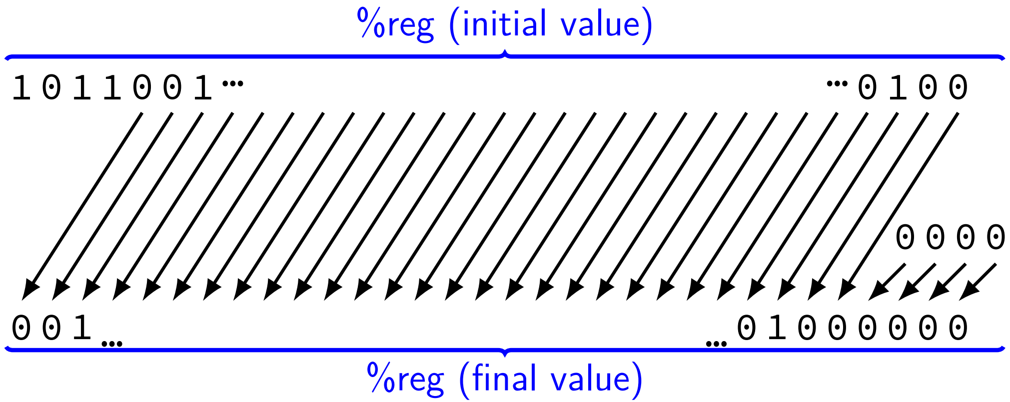
instead: `value << 4`



shift left

x86 instruction: `shl` — shift left

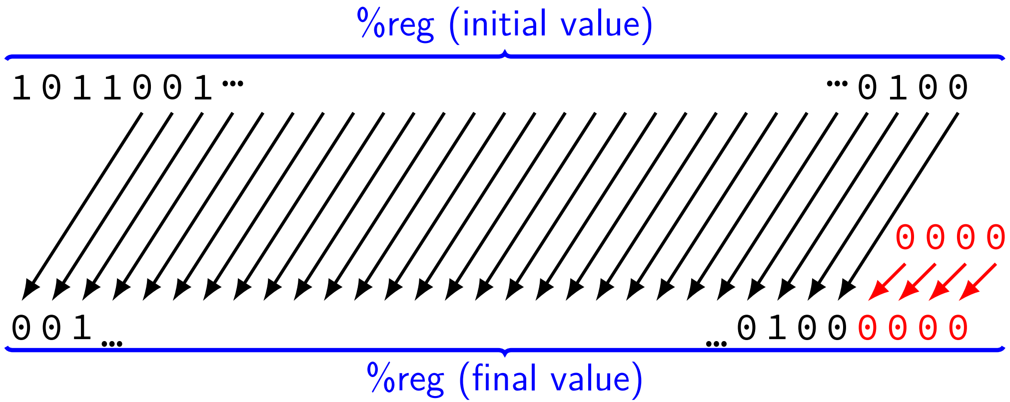
`shl $amount, %reg` (or variable: `shl %cl, %reg`)



shift left

x86 instruction: `shl` — shift left

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



left shift in math

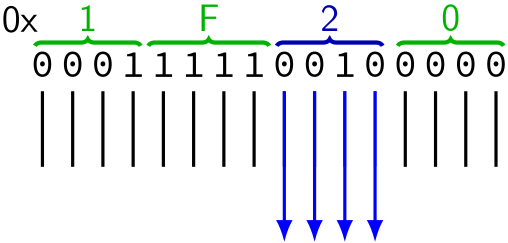
```
1 << 0 == 1                 0000 0001
1 << 1 == 2                 0000 0010
1 << 2 == 4                 0000 0100

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

left shift in math

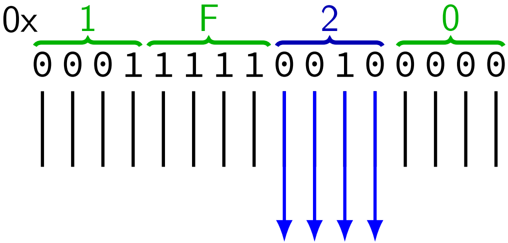
```
1 << 0 == 1            0000 0001
1 << 1 == 2            0000 0010
1 << 2 == 4            0000 0100

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

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

extracting nibble from more



extracting nibble from more



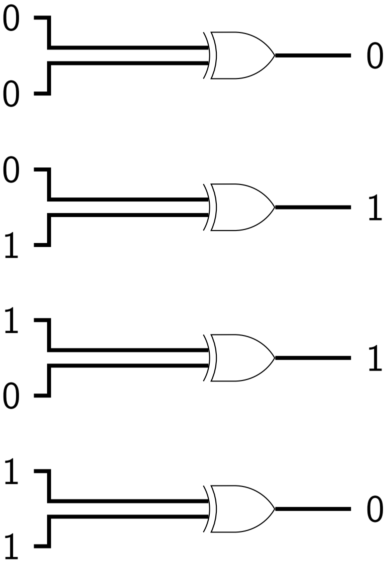

```
// % -- remainder
unsigned extract_second_nibble(unsigned value) {
    return (value / 16) % 16;
}

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

manipulating bits?

easy to manipulate individual bits in HW

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

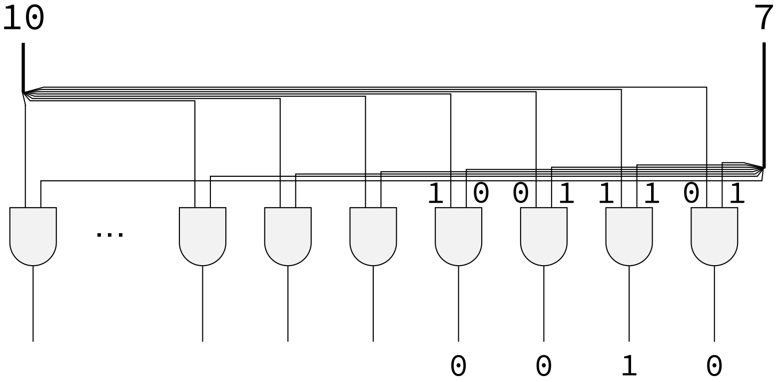
	…	1	0	1	0
&	…	0	1	1	1
	…	0	0	1	0

bitwise AND — C/assembly

x86: `and %reg, %reg`

C: `foo & bar`

bitwise hardware (10 & 7 == 2)



extract 0x3 from 0x1234

```
unsigned get_second_nibble1_bitwise(unsigned value)
    return (value >> 4) & 0xF; // 0xF: 00001111
    // like (value / 16) % 16
}

unsigned get_second_nibble2_bitwise(unsigned value)
    return (value & 0xF0) >> 4; // 0xF0: 11110000
    // like (value % 256) / 16;
}
```

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
```

more truth tables

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

|

conditionally set bit

^

conditionally flip bit

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

future assignment

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; }
```

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

→ `(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) & 1)` only gets least sig. bit

`(x ? y : z)`

multibit ternary

constraint: `x` *is 0 or 1*

old solution `((x & y) | (~x) & 1)` 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)`

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)`

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

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

a trick: $-$`x` (`-1` is `1111`…`1`)

constructing masks

constraint: `x` *is 0 or 1*

`(x ? y : 0)`

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

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

a trick: $-x$ (`-1` is `1111`...`1`)

`((-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`

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) & 1)` 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` = 0 or 1, `!!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` = 0 or 1, `!!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)`

simple operation performance

typical modern desktop processor:

- bitwise and/or/xor, shift, add, subtract, compare — $\sim$ 1 cycle
- integer multiply — $\sim$ 1-3 cycles
- integer divide — $\sim$ 10-150 cycles

(smaller/simpler/lower-power processors are different)

simple operation performance

typical modern desktop processor:

- bitwise and/or/xor, shift, add, subtract, compare — $\sim$ 1 cycle
- integer multiply — $\sim$ 1-3 cycles
- integer divide — $\sim$ 10-150 cycles

(smaller/simpler/lower-power processors are different)

add/subtract/compare are more complicated in hardware!

but *much* more important for typical applications

backup slides

switch to if-then

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    default: ...
}
```

```
    // same as if statement?
    cmpq $1, %rax
    je code_for_1
    cmpq $2, %rax
    je code_for_2
    cmpq $3, %rax
    je code_for_3
    ...
    jmp code_for_default
```

switch-to-binary-search

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 100: ...; break;
    default: ...
}
```

```
    // binary search
    cmpq $50, %rax
    jl code_for_less_than_50
    cmpq $75, %rax
    jl code_for_50_to_75
    ...
code_for_less_than_50:
    cmpq $25, %rax
    jl less_than_25_cases
    ...
```

example: C that is not C++

valid C and invalid C++:

```
char *str = malloc(100);
```

valid C and valid C++:

```
char *str = (char *) malloc(100);
```

valid C and invalid C++:

```
int class = 1;
```

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

parallelism

bitwise operations — each bit is seperate

parallelism

bitwise operations — each bit is seperate

same idea can apply to more interesting operations

$010 + 011 = 101; 001 + 010 = 011 \rightarrow$
$01000001 + 01100010 = 10100011$

parallelism

bitwise operations — each bit is seperate

same idea can apply to more interesting operations

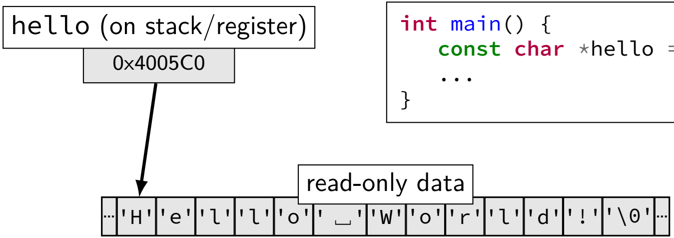
$010 + 011 = 101;\ 001 + 010 = 011\ \rightarrow$
$01000001 + 01100010 = 10100011$

sometimes specific HW support

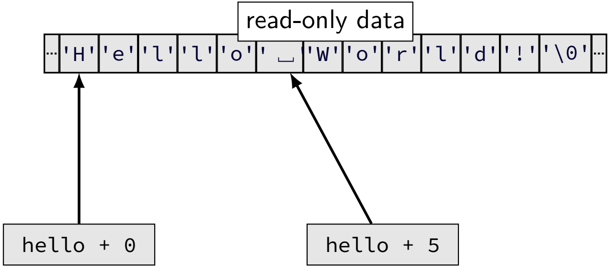
- e.g. x86-64 has a "multiply four pairs of floats" instruction

strings in C

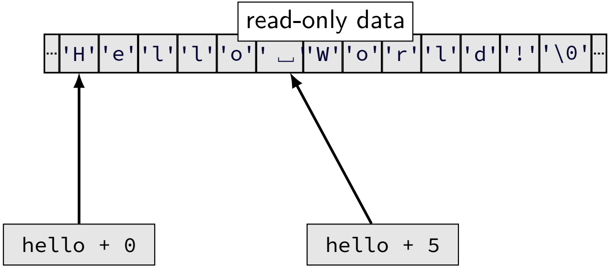
```c
int main() {
    const char *hello = "Hello World!";
    ...
}
```



pointer arithmetic



pointer arithmetic



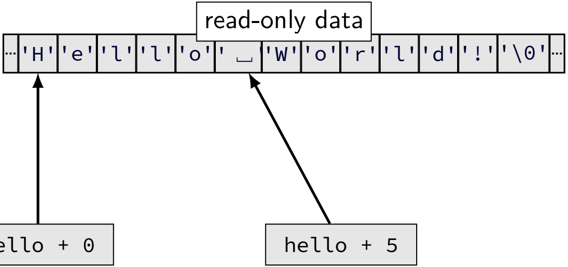
0x4005C0

0x4005C5

*(hello + 0) is 'H'

*(hello + 5) is '␣'

pointer arithmetic



0x4005C0

*(hello + 0) is 'H'

hello[0] is 'H'

0x4005C5

*(hello + 5) is '␣'

hello[5] is '␣'

arrays and pointers

`*(foo + bar)` exactly the same as `foo[bar]`

arrays ‘decay’ into pointers

arrays of non-bytes

`array[2]` and `*(array + 2)` still the same

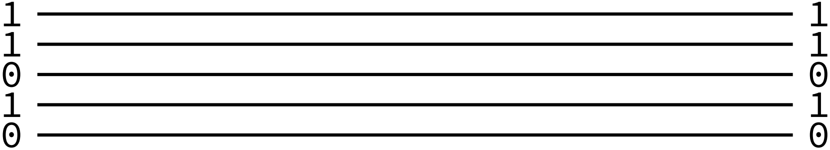
```
int numbers[4] = {10, 11, 12, 13};
int *pointer;
pointer = numbers;
*pointer = 20; // numbers[0] = 20;
pointer = pointer + 2;
/* adds 8 (2 ints) to address */
*pointer = 30; // numbers[2] = 30;
// numbers is 20, 11, 30, 13
```

arrays of non-bytes

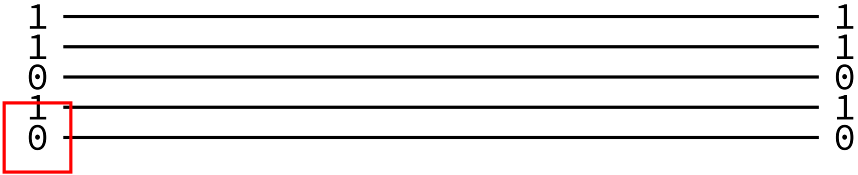
array[2] and *(array + 2) still the same

```
int numbers[4] = {10, 11, 12, 13};
int *pointer;
pointer = numbers;
*pointer = 20;  // numbers[0] = 20;
pointer = pointer + 2;
/* adds 8 (2 ints) to address */
*pointer = 30;  // numbers[2] = 30;
// numbers is 20, 11, 30, 13
```

circuits: wires

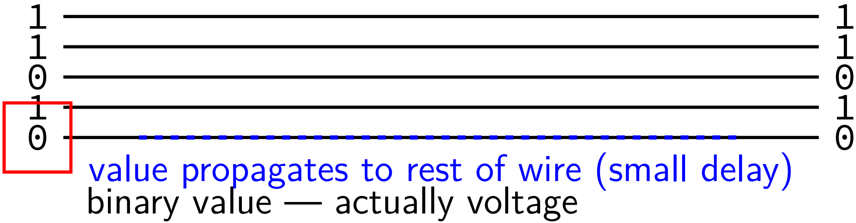


circuits: wires

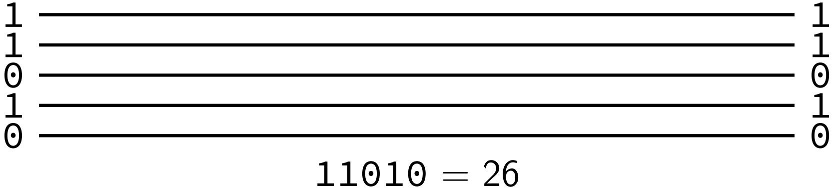


binary value — actually voltage

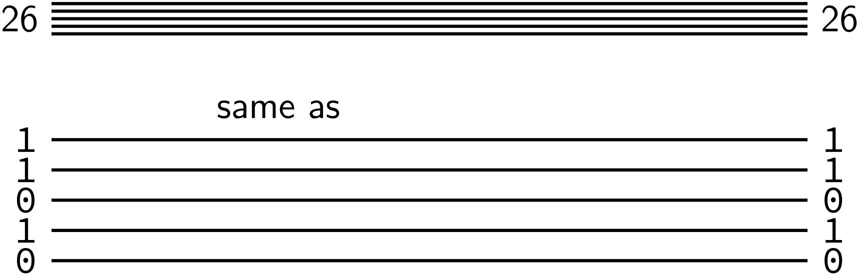
circuits: wires



circuits: wire bundles



circuits: wire bundles



`11010` = 26

circuits: wire bundles