bitwise

Changelog

8 September 2020: and/or/xor: add explanation of what masks would contain

8 September 2020: and/or/xor: ...and make sure explanation for xor says “flip”, not “set”

last time

options for compiling switches (if/else; binary search; jump table)

program layout (briefly)

object files

- symbol table: labels other object files might need
- relocations: places to change once we know addresses of other labels
- linker handles relocations, combines object files into executable
- executable ready to be loaded into memory

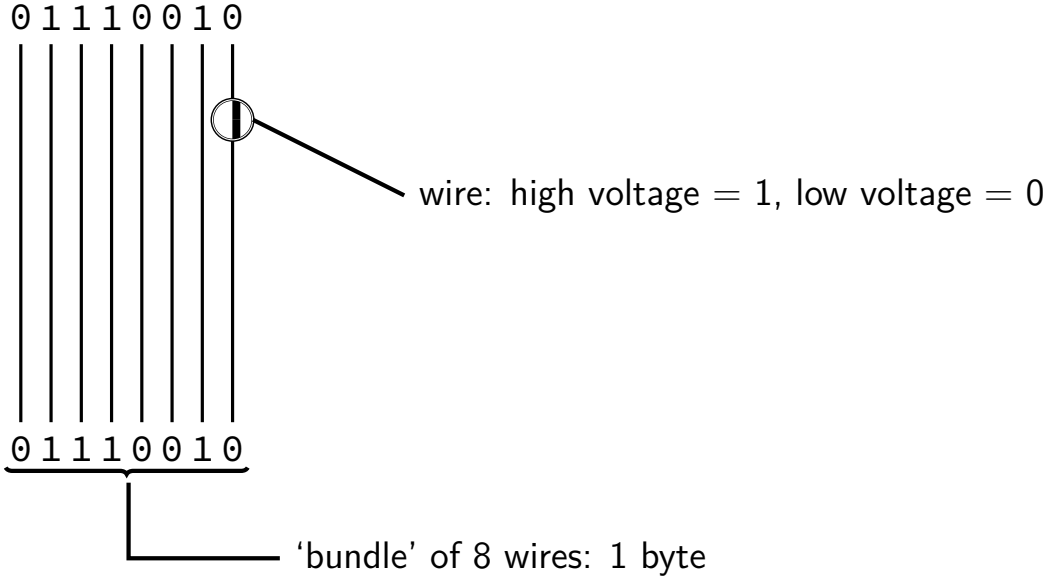
C pointer arithmetic

- `SomeType *p` $\implies$ `p += 1` adds `sizeof(SomeType)` to address

C undefined behavior

PM lecture while exericse error note

moving bits in hardware (one way)



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

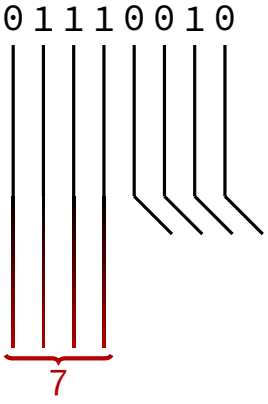
Intel "Skylake" microarchitecture:

- 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

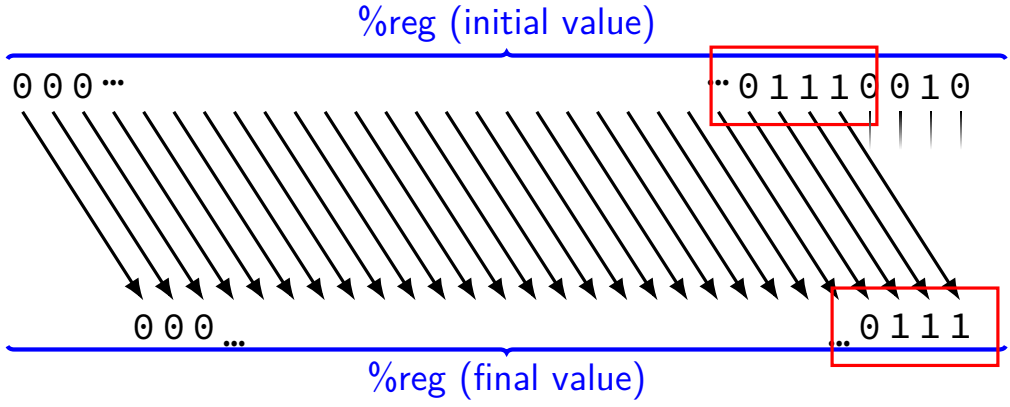
0111 0010 = 0x72



exposing wire selection

x86 instruction: `shr` — shift right

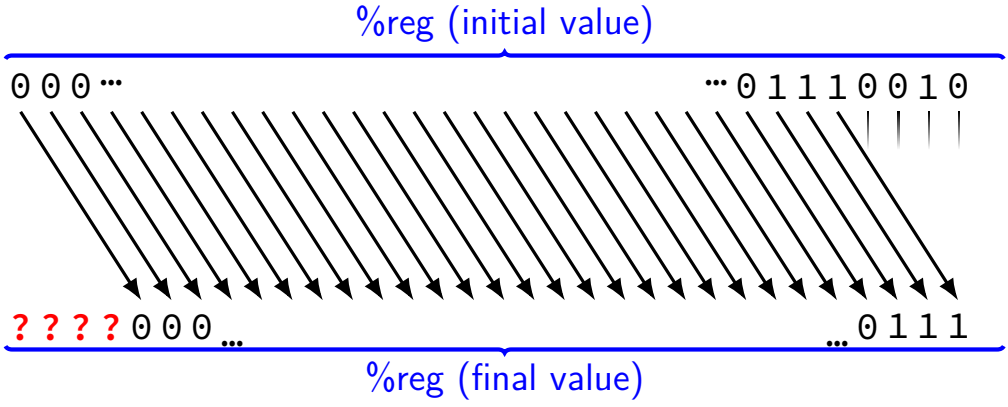
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

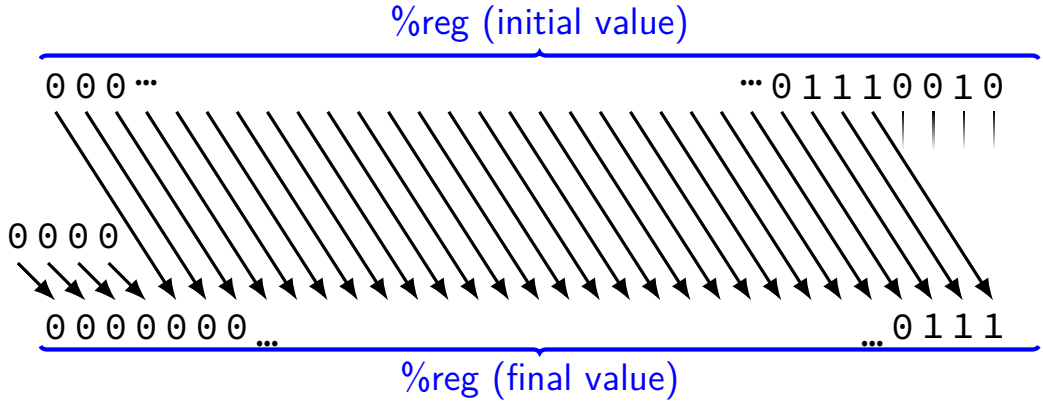
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



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) 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

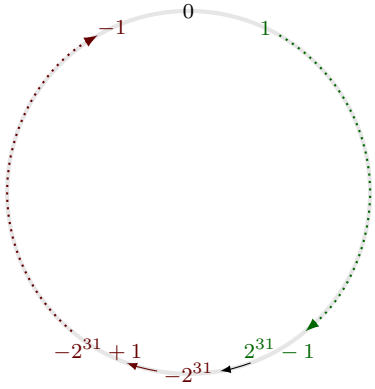
G. a large negative number H. something else

two's complement refresher

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

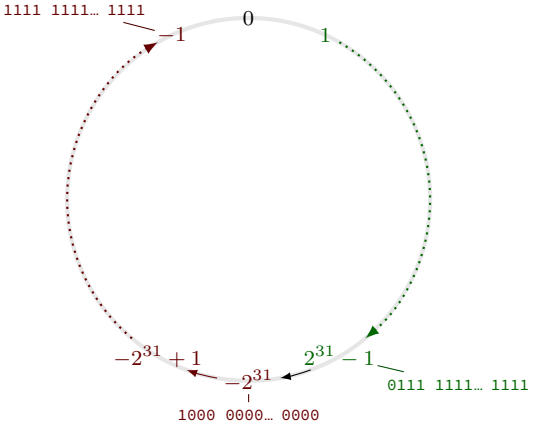
two's complement refresher

$$-1 = \underset{-2^{31}}{1}\ \underset{+2^{30}}{1}\ \underset{+2^{29}}{1}\ \ldots\ \underset{+2^{2}}{1}\ \underset{+2^{1}}{1}\ \underset{+2^{0}}{1}$$



two's complement refresher

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



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 $x/2$

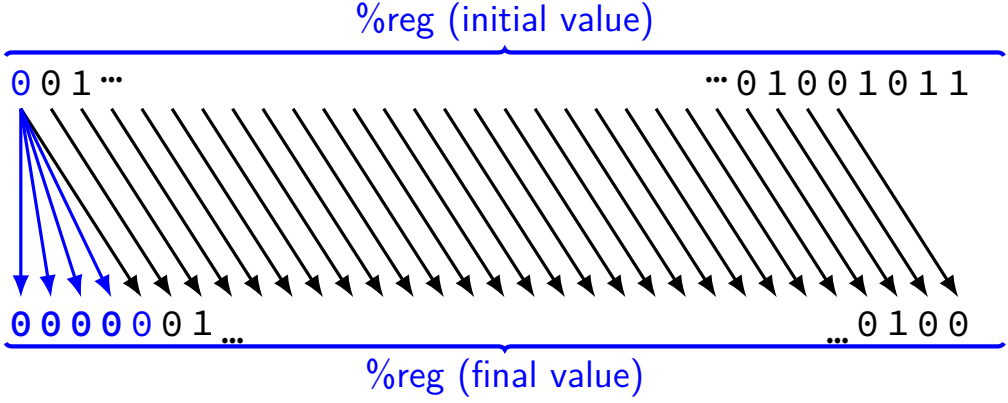
flip all bits and add one to get $-x/2$

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

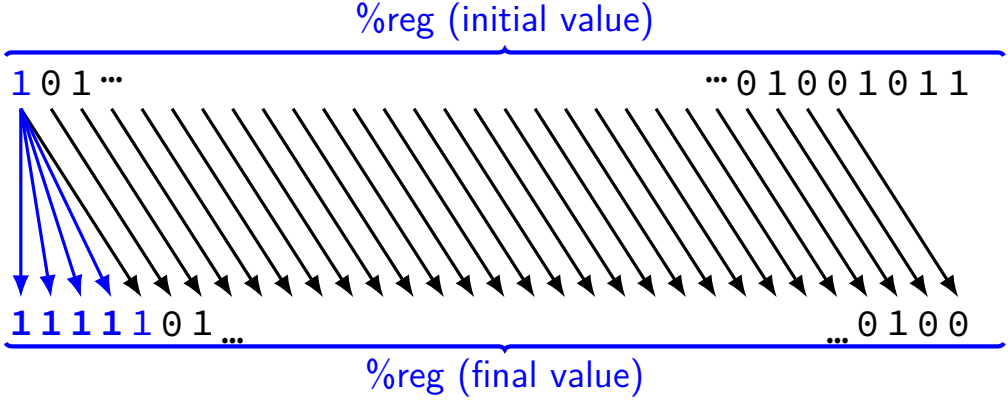
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

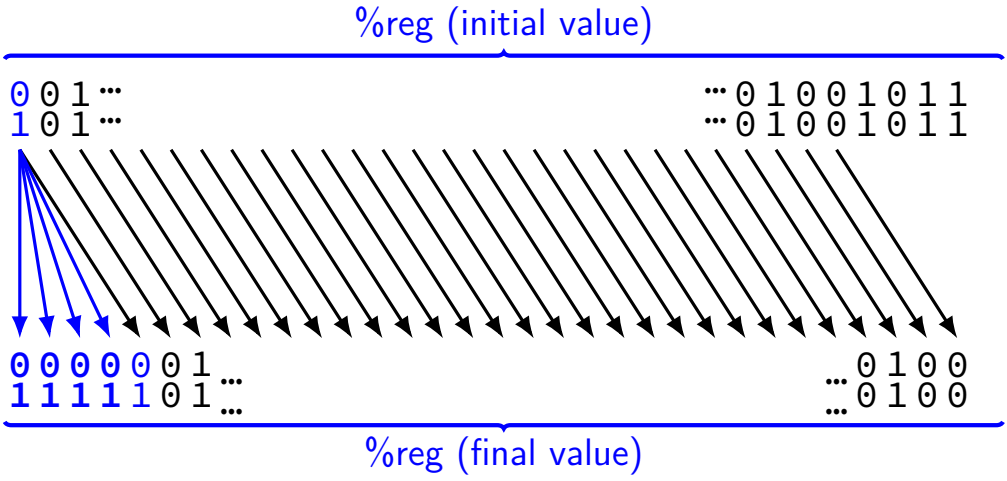
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



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…111111110`~~`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

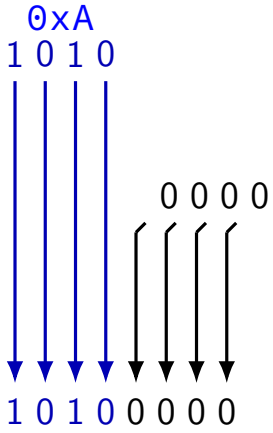
C division: rounds towards zero (truncate)

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



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

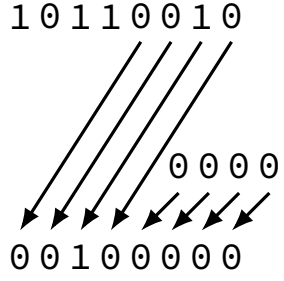
shift left

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

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

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

instead: `value << 4`



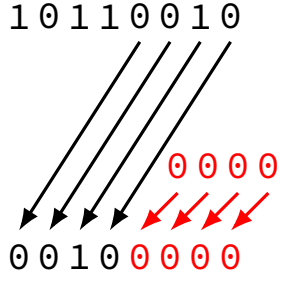
shift left

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

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

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

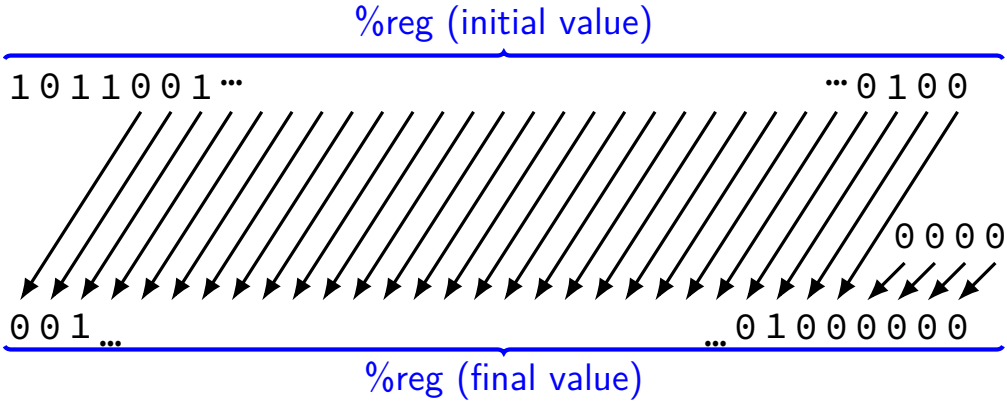
instead: `value << 4`



shift left

x86 instruction: `shl` — shift left

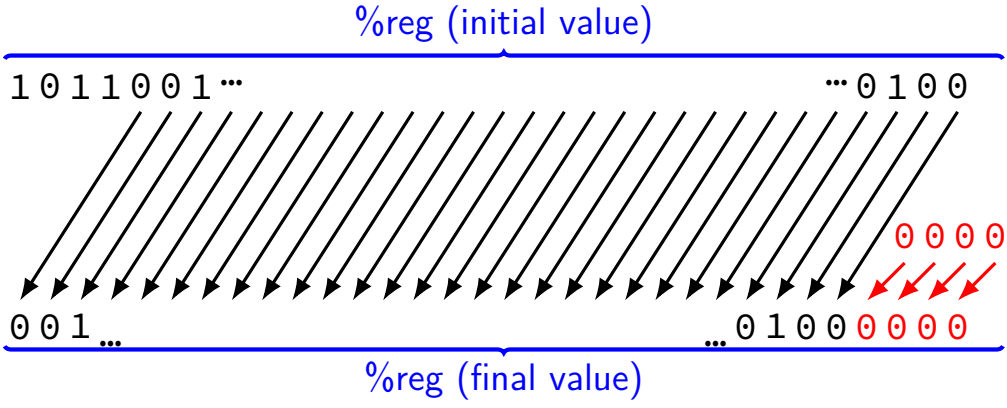
`shl` `$`*`amount`*`, %reg` (or variable: `shl %cl, %reg`)



shift left

x86 instruction: `shl` — shift left

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



left shift in math

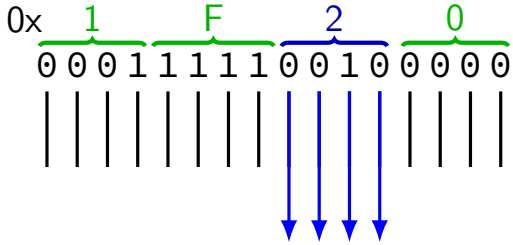
```
1 << 0 == 1                 0000 0001
1 << 1 == 2                 0000 0010
1 << 2 == 4                 0000 0100

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`

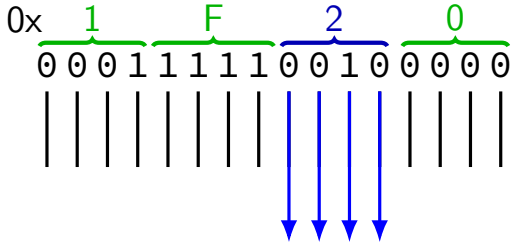
$$x \texttt{ << } y = x \times 2^y$$

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?

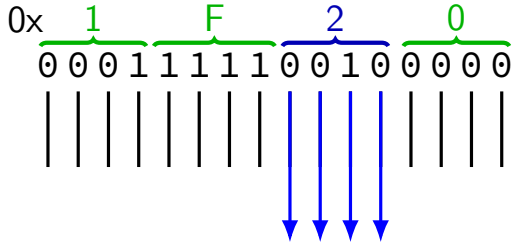
value=0x1F20 → 0x1F2

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;
}
```

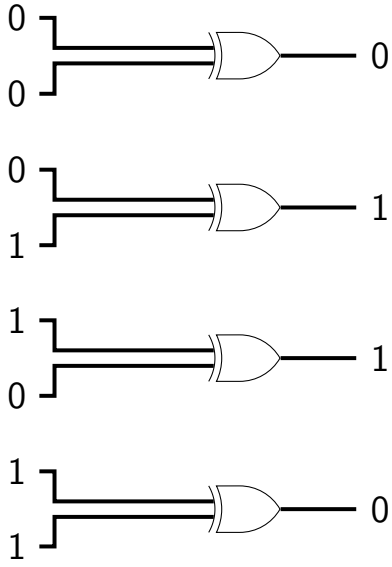
manipulating bits?

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
```

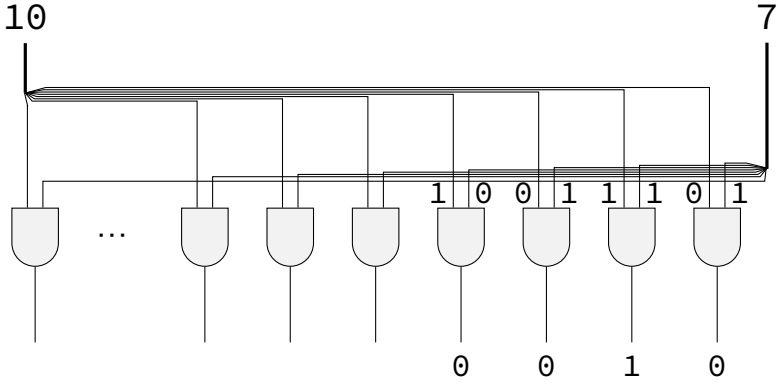
```
    … 1 0 1 0
&   … 0 1 1 1
  ---------
    … 0 0 1 0
```

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 → 0000000000cccc0000 → 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

&

conditionally clear bit
conditionally keep bit

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

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

|

conditionally set bit

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

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

^

conditionally flip bit

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

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 — ^

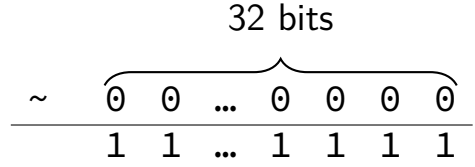
```
1 ^ 1 == 0

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 `!`:

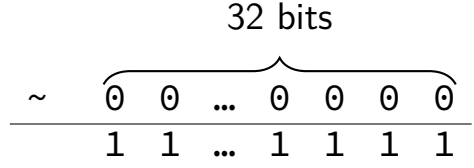
```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)
```



negation / not — ~

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

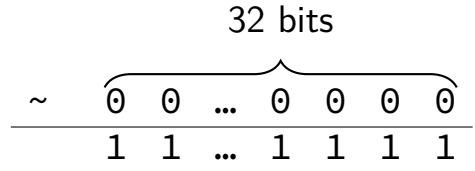
```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)

~2 == (int) 0xFFFFFFFD (aka −3)
```



negation / not — ~

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

```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)

~2 == (int) 0xFFFFFFFD (aka −3)

~((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; }
```

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

first question: let's find a simpler version

exercise: come up with a few options

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) & 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)`

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: want `1111111111…1`

if x = 0: want `0000000000…0`

mask: ~~$\sim$~~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` = 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

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

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

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

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

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

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))`

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

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`

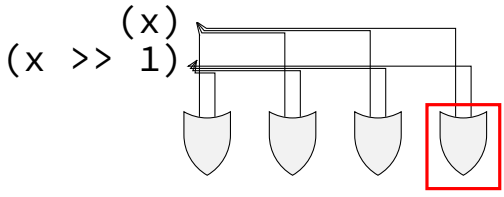
```
(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1
```

wasted work (2)

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

performing 3 bitwise ors

...each bitwise or does 4 OR operations



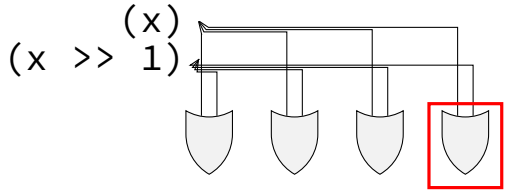
wasted work (2)

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

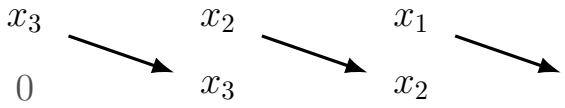
performing 3 bitwise ors

...each bitwise or does 4 OR operations

but only result of one of the 4!

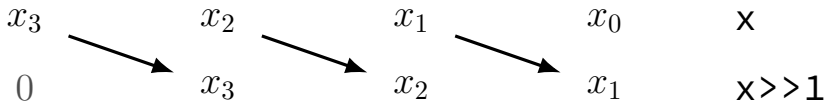


any-bit: looking at wasted work



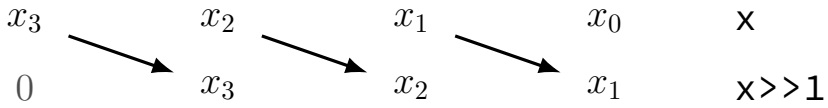

```
y=(x|x>>1)
```

any-bit: looking at wasted work



$(0|x_3)$ $(x_3|x_2)$ $(x_2|x_1)$ $(x_1|x_0)$ y=(x|x>>1)

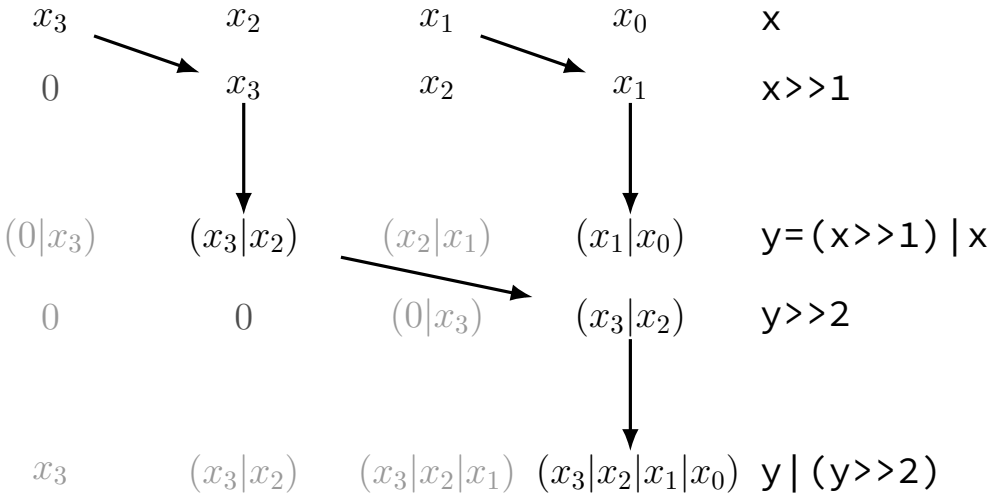
any-bit: looking at wasted work



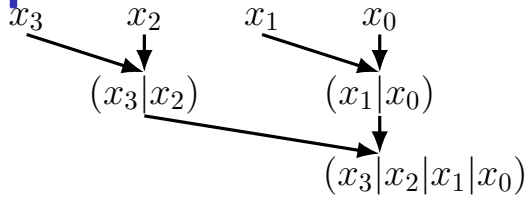
$(0|x_3)$ $(x_3|x_2)$ $(x_2|x_1)$ $(x_1|x_0)$ y=(x|x>>1)

final value wanted: $x_3|x_2|x_1|x_0$
previously:
compute x|(x>>1) for $x_1|x_0$;
(x>>2)|(x>>3) for $x_3|x_2$
observation: got both parts with just x|(x>>1)

any-bit: divide and conquer



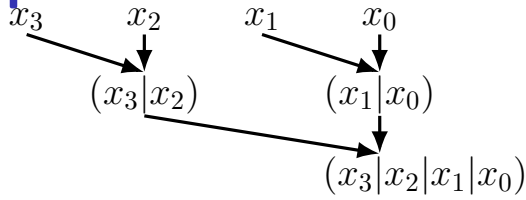
any-bit: divide and conquer



four-bit input $x = x_3x_2x_1x_0$

`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

any-bit: divide and conquer



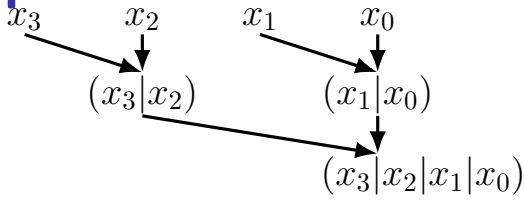
four-bit input $x = x_3x_2x_1x_0$

`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

`y | (y >> 2)` $= (y_1|0)(y_2|0)(y_3|y_1)(y_4|y_2) = z_1z_2z_3z_4$

$z_4 = (y_4|y_2) = ((x_2|x_3)|(x_0|x_1)) = x_0|x_1|x_2|x_3$ "is any bit set?"

any-bit: divide and conquer



four-bit input $x = x_3x_2x_1x_0$

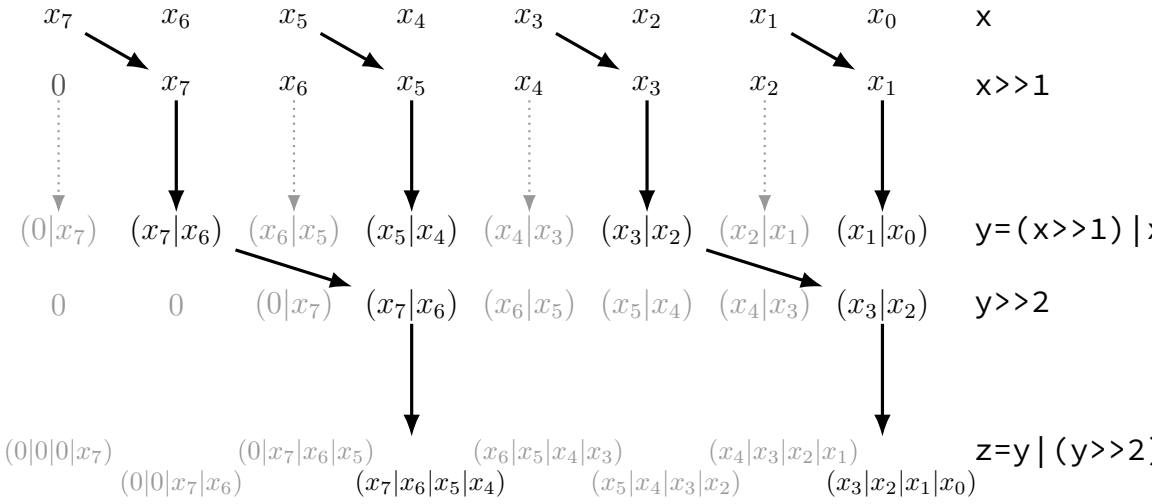
`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

`y | (y >> 2)` $= (y_1|0)(y_2|0)(y_3|y_1)(y_4|y_2) = z_1z_2z_3z_4$

$z_4 = (y_4|y_2) = ((x_2|x_3)|(x_0|x_1)) = x_0|x_1|x_2|x_3$ "is any bit set?"

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

any-bit: divide and conquer



any-bit-set: 32 bits

```
unsigned int any(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, find subproblems, etc.

mask and shift

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

factor/distribute

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

divide and conquer

common subexpression elimination

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

becomes

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

backup slides

short-circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	**false**	**true**
false	false	true
true	true	true

short-circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() || one());
6     printf("> %d\n", one() || zero());
7     return 0;
8 }
```

```
zero()
one()
> 1
one()
> 1
```

OR	**false**	**true**
false	false	true
true	true	true

short-circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	**false**	**true**
false	false	true
true	true	true

short-circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	**false**	**true**
false	false	true
true	true	true

short-circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	**false**	**true**
false	false	true
true	true	true

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**false**	**true**
false	false	false
true	false	true

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**false**	**true**
false	false	false
true	false	true

short-circuit (&&)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() && one());
    printf("> %d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**false**	**true**
false	false	false
true	false	true

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**false**	**true**
false	false	false
true	false	true

short-circuit (&&)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() && one());
    printf("> %d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**false**	**true**
false	false	false
true	false	true