bitwise operators

Changelog

Changes made in this version not seen in first lecture:

6 Feb 2018: arithmetic right shift: x86 arith. shift instruction is `sar` to `sra`

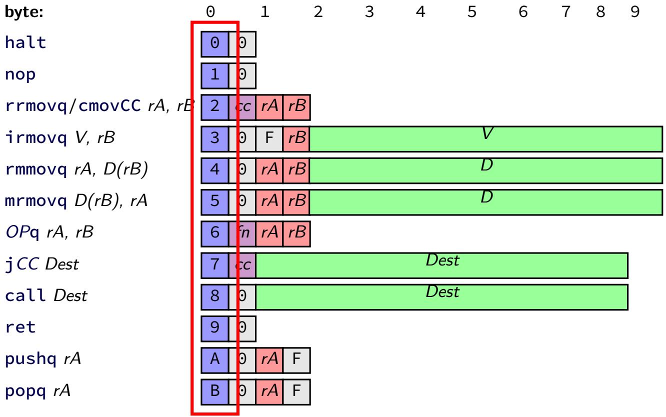
6 Feb 2018: logical left shift: use `shl` consistently

6 Feb 2018: exercise C explanation: correct bcde00 typo for abcd00

6 Feb

extracting opcodes (1)

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



extracing opcodes (2)

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

aside: division

division is really slow

Intel “Skylake” microarchitecture:

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

aside: division

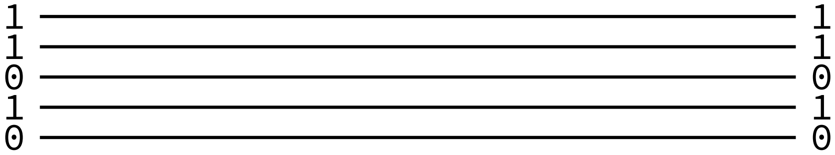
division is really slow

Intel "Skylake" microarchitecture:

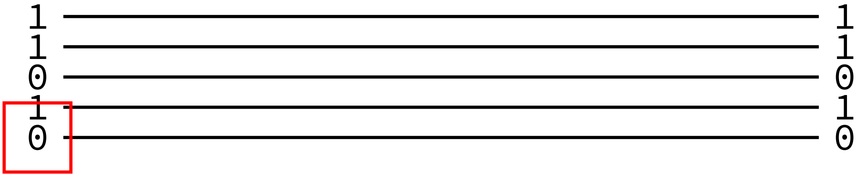
- about six cycles per division
- ...and much worse for eight-byte division
- versus: four additions per cycle

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

circuits: wires

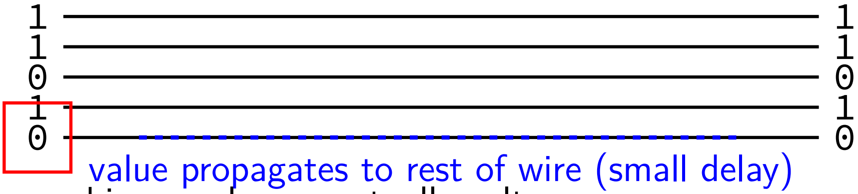


circuits: wires



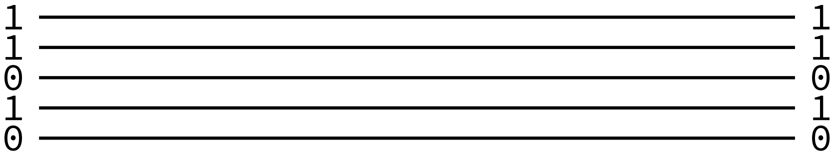
binary value — actually voltage

circuits: wires



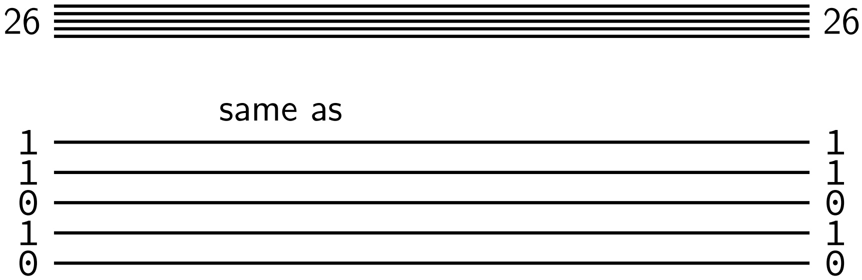
binary value — actually voltage

circuits: wire bundles



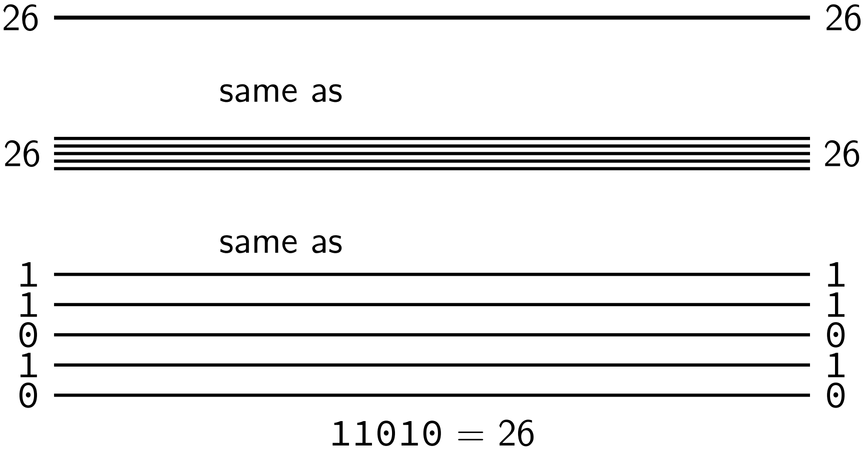
$\texttt{11010} = 26$

circuits: wire bundles



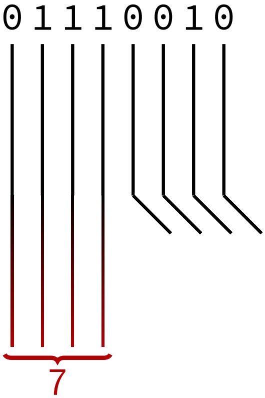
11010 = 26

circuits: wire bundles



extracting opcode in hardware

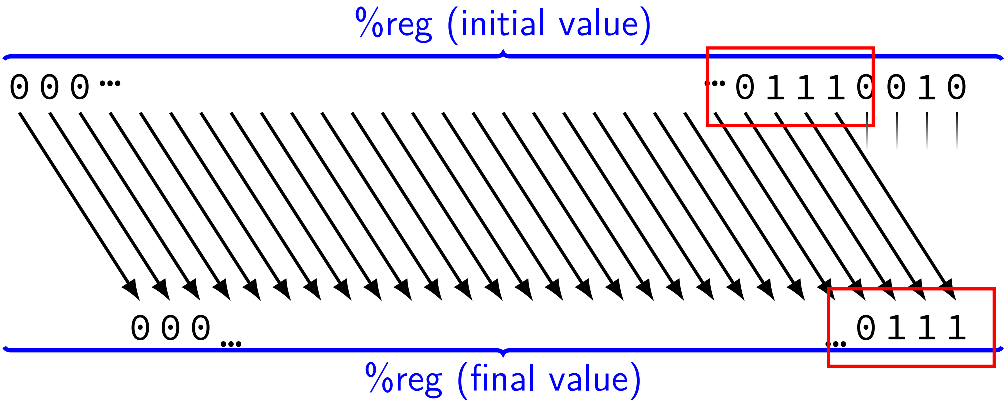
`0111 0010` = 0x72 (first byte of jl)



exposing wire selection

x86 instruction: `shr` — shift right

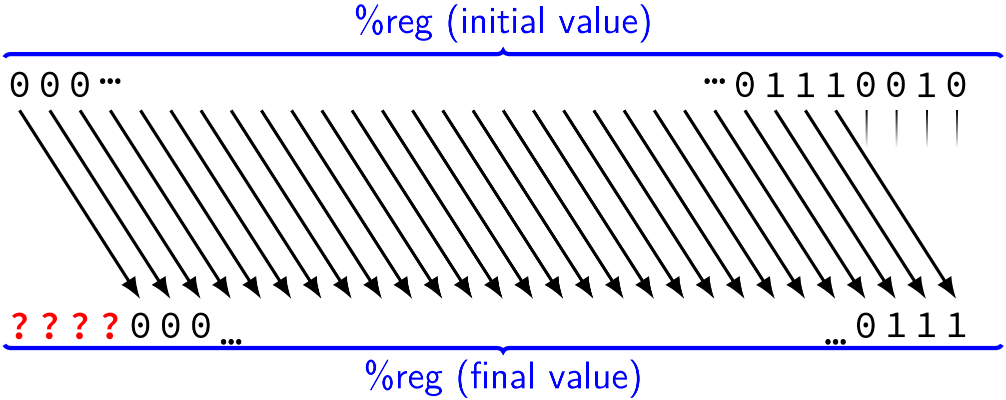
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

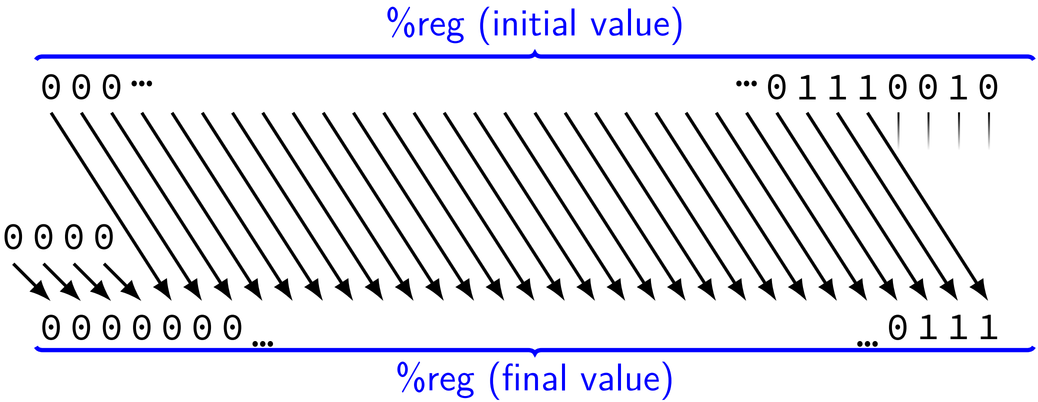
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



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

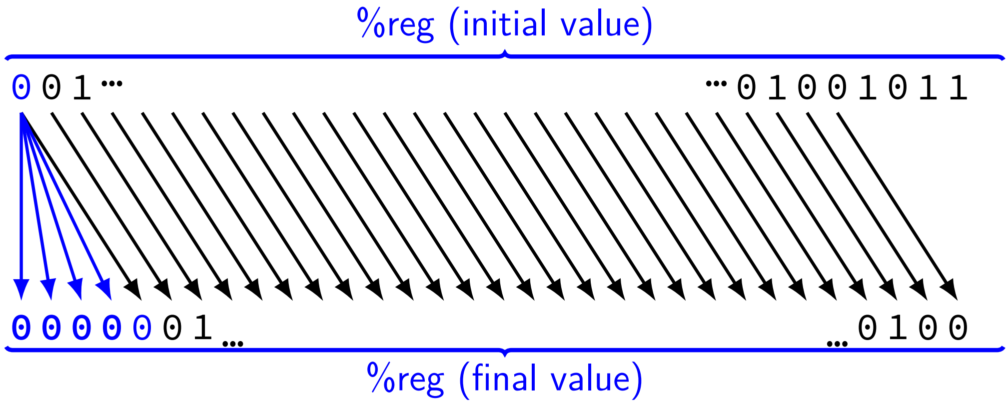
```
1 >> 0 == 1            0000 0001
1 >> 1 == 0            0000 0000
1 >> 2 == 0            0000 0000

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

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

arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

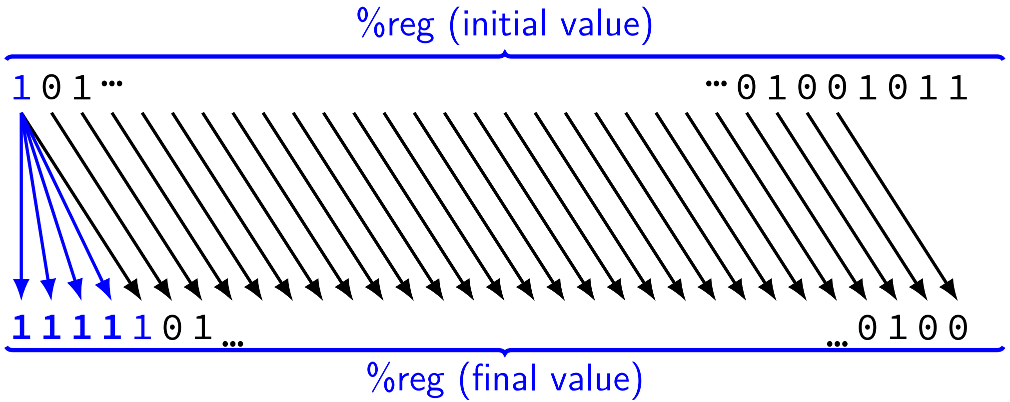
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

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



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)

right shift in C

```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

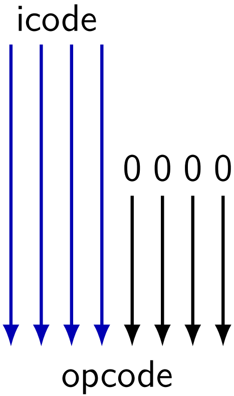
signed right shift is implementation-defined

- 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

constructing instructions in hardware



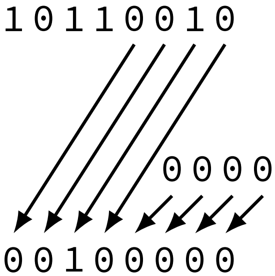
shift left

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

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

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

instead: `opcode << 4`



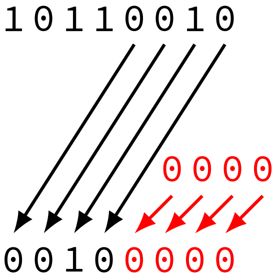
shift left

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

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

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

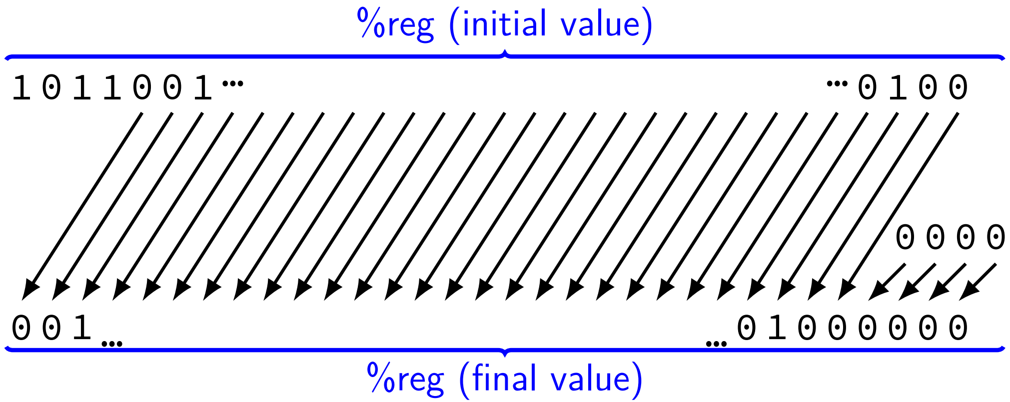
instead: `opcode << 4`



shift left

x86 instruction: `shl` — shift left

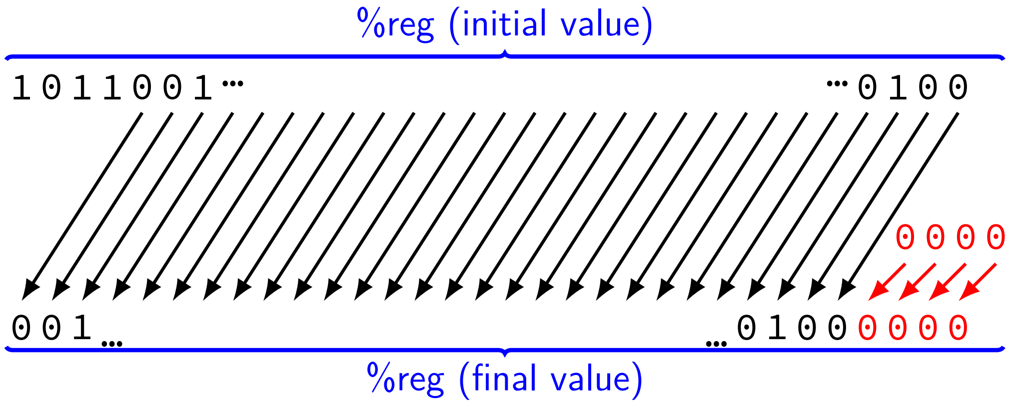
`shl $`*`amount`*`, %reg` (or variable: `shl %cl, %reg`)



shift left

x86 instruction: `shl` — shift left

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



left shift in math

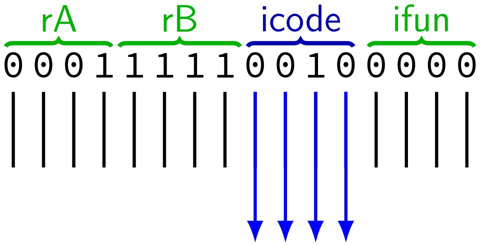
```
1 << 0 == 1              0000 0001
1 << 1 == 2              0000 0010
1 << 2 == 4              0000 0100

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

left shift in math

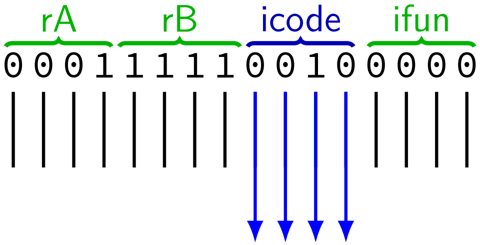
```
1 << 0 == 1            0000 0001
1 << 1 == 2            0000 0010
1 << 2 == 4            0000 0100

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

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

extracting icode from more



extracting icode from more



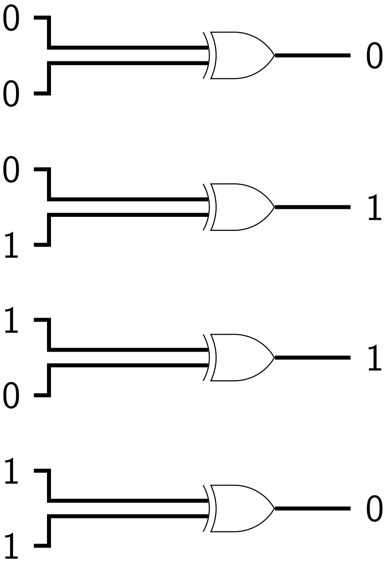

```
// % -- remainder
unsigned extract_opcode1(unsigned value) {
    return (value / 16) % 16;
}

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?

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

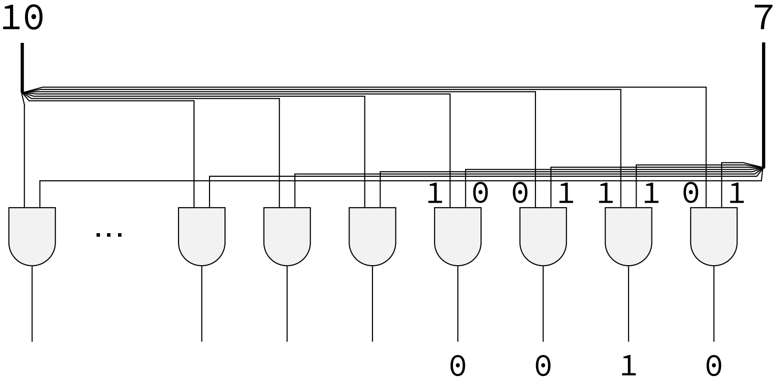
	…	1	0	1	0
&	…	0	1	1	1
	…	0	0	1	0

bitwise AND — C/assembly

x86: `and %reg, %reg`

C: `foo & bar`

bitwise hardware (10 & 7 == 2)



extract 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
```

```
      … 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
~2 == (int) 0xFFFFFFFD
```

(aka -1) for `~0`; (aka -3) for `~2`

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
```

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

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`

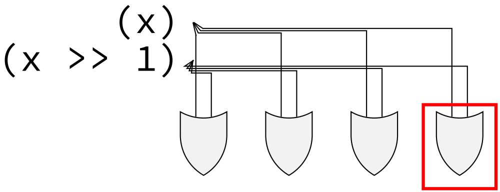
```
(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1
```

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



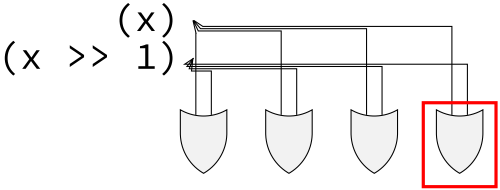
wasted work (2)

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: divide and conquer

four-bit input $x = x_1 x_2 x_3 x_4$

`x | (x >> 1)` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1 y_2 y_3 y_4$

any-bit: divide and conquer

four-bit input $x = x_1 x_2 x_3 x_4$

`x | (x >> 1)` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1 y_2 y_3 y_4$

`y | (y >> 2)` $= (y_1|0)(y_2|0)(y_3|y_1)(y_4|y_2) = z_1 z_2 z_3 z_4$

$z_4 = (y_4|y_2) = ((x_2|x_1)|(x_4|x_3)) = x_4|x_3|x_2|x_1$ "is any bit set?"

any-bit: divide and conquer

four-bit input $x = x_1x_2x_3x_4$

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

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

$z_4 = (y_4|y_2) = ((x_2|x_1)|(x_4|x_3)) = x_4|x_3|x_2|x_1$ "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-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)
```

exercise

Which of these will swap last and second-to-last bit of an `unsigned int` x? ($abcdef$ becomes $abcdfe$)

```
/* version A */
    return ((x >> 1) & 1) | (x & (~1));

/* version B */
    return ((x >> 1) & 1) | ((x << 1) & (~2)) | (x & (~3));

/* version C */
    return (x & (~3)) | ((x & 1) << 1) | ((x >> 1) & 1);

/* version D */
    return (((x & 1) << 1) | ((x & 3) >> 1)) ^ x;
```

version A

```
/* version A */
    return ((x >> 1) & 1) | (x & (~1));
    //     ^^^^^^^^^^^^^^
    //      abcdef --> 0abcde -> 00000e

    //                      ^^^^^^^^^^
    //      abcdef --> abcde0

    //     ^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //      00000e | abcde0 = abcdee
```

version B

```
/* version B */
    return ((x >> 1) & 1) | ((x << 1) & (~2)) | (x & (~3));
    //     ^^^^^^^^^^^^^^
    //      abcdef --> 0abcde --> 00000e

    //                       ^^^^^^^^^^^^^^^
    //      abcdef --> bcdef0 --> bcde00

    //                                          ^^^^^^^^^^
    //      abcdef -->            abcd00
```

version C

```
/* version C */
    return (x & (~3)) | ((x & 1) << 1) | ((x >> 1) & 1);
    //     ^^^^^^^^^^
    //      abcdef -->            abcd00

    //                 ^^^^^^^^^^^^^^
    //      abcdef --> 00000f --> 0000f0

    //                                    ^^^^^^^^^^^^^^
    //      abcdef --> 0abcde --> 00000e
```

version D

```
/* version D */
    return (((x & 1) << 1) | ((x & 3) >> 1)) ^ x;
    //     ^^^^^^^^^^^^^^^
    //      abcdef --> 00000f --> 0000f0

    //                        ^^^^^^^^^^^^^^
    //      abcdef --> 0000ef --> 00000e

    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //      0000fe ^ abcdef --> abcd(f XOR e)(e XOR f)
```

expanded code

```
int lastBit = x & 1;
int secondToLastBit = x & 2;
int rest = x & ~3;
int lastBitInPlace = lastBit << 1;
int secondToLastBitInPlace = secondToLastBit >> 1;
return rest | lastBitInPlace | secondToLastBitInPlace;
```

backup slides

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
```

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

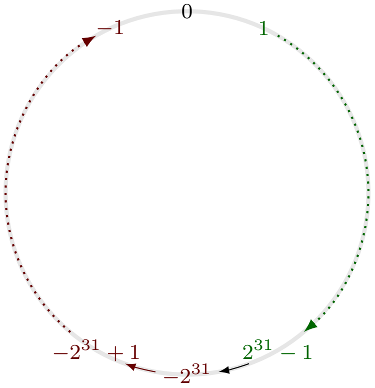
- e.g. x86-64 has a “multiply four pairs of floats” instruction

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

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