Selected C Topics
Bitwise

February 2, 2023

last lecture topics

while/switch-to-assembly
- levels of optimization
- various implementations

object files
- text: machine code with things missing
- data, relocation table (directives how to fill in)
- symbol table: what other files might need

executable files machine code

compilation pipeline: compile, assemble, linking

lab+HW

Labs started, fun!

Homework due February 8 4:59 pm (Wednesday afternoon)

C Data Types

Varies between machines(!). For this course:

type	size (bytes)
`char`	1
`short`	2
`int`	4
`long`	8

C Data Types

Varies between machines(!). For this course:

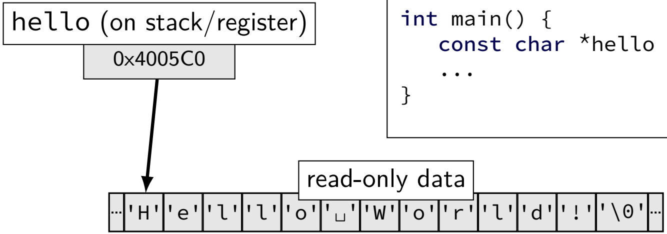
type	size (bytes)
`char`	1
`short`	2
`int`	4
`long`	8
`float`	4
`double`	8

C Data Types

Varies between machines(!). For this course:

type	size (bytes)
`char`	1
`short`	2
`int`	4
`long`	8
`float`	4
`double`	8
`void *`	8
`anything` `*`	8

strings in C



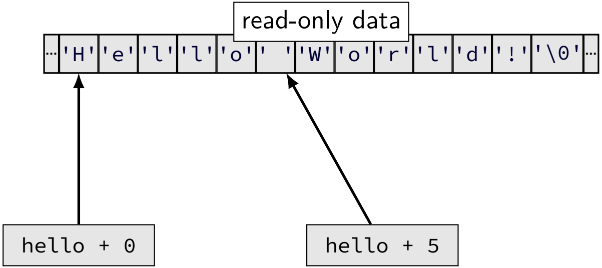

```c
int main() {
    const char *hello = "Hello World!";
    ...
}
```

char arrays

null terminated

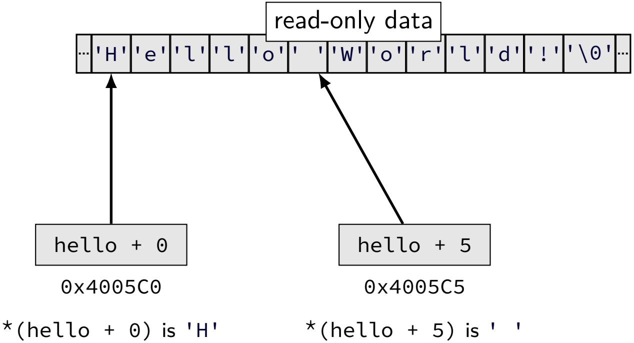
pointer stores beginning of array

pointer arithmetic



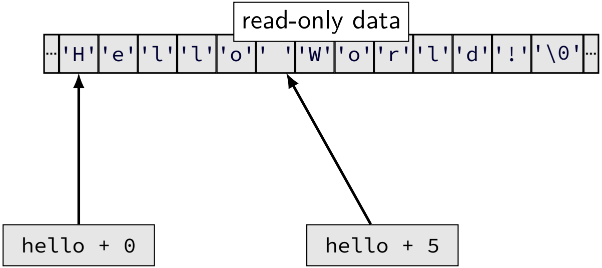
two different ways to access array indexes

pointer arithmetic



two different ways to access array indexes

pointer arithmetic



0x4005C0

*(hello + 0) is 'H'

hello[0] is 'H'

0x4005C5

*(hello + 5) is ' '

hello[5] is ' '

two different ways to access array indexes

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}
```

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

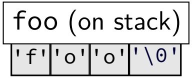
```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

Final value of `foo`?

A. `"fao"`
B. `"zao"`
C. `"baz"`
D. `"bao"`
E. something else/crash

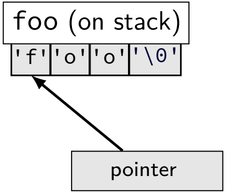
exercise explanation

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



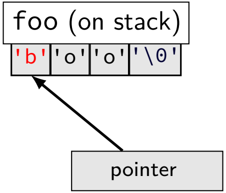
exercise explanation

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



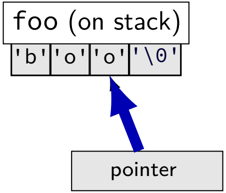
exercise explanation

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



exercise explanation

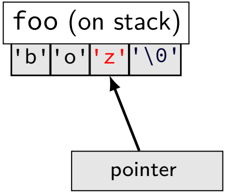
```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



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';`

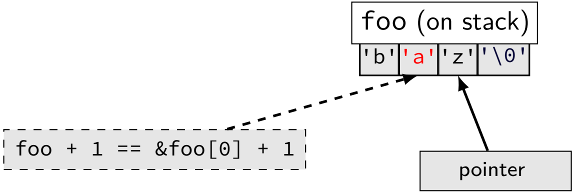


exercise explanation

```
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';`



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])`)

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — “K&R C”

- very different from modern C

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — “K&R C”

- very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

- compiler option: `-ansi`, `-std=c90`
- looks mostly like modern C

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — “K&R C”

- very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

- compiler option: `-ansi`, `-std=c90`
- looks mostly like modern C

1999: ISO (and ANSI) update C standard — C99

- compiler option: `-std=c99`
- adds: declare variables in middle of block
- adds: `//` comments

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — "K&R C"

very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

compiler option: `-ansi`, `-std=c90`

looks mostly like modern C

1999: ISO (and ANSI) update C standard — C99

compiler option: `-std=c99`

adds: declare variables in middle of block

adds: // comments

2011, 2017: Second/Third ISO update — C11, C17

undefined behavior example (1)

```
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX)); // INT_MAX+1
}
```

undefined behavior example (1)

```
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX)); // INT_MAX+1
}
```

without optimizations: `0`

undefined behavior example (1)

```
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX)); // INT_MAX+1
}
```

without optimizations: `0`

with optimizations: `1`

undefined behavior example (2)

```
int test(int number) {
    return (number + 1) > number;
}
```

Optimized:

```
test:
    movl    $1, %eax        # eax <- 1
    ret
```

Less optimized:

```
test:
    leal    1(%rdi), %eax # eax <- rdi + 1
    cmpl    %eax, %edi
    setl    %al            # al <- eax < edi
    movzbl  %al, %eax      # eax <- al (pad with zeros)
    ret
```

undefined behavior

compilers can do whatever they want

- what you expect
- crash your program
- ...

common types:

- *signed* integer overflow/underflow
- out-of-bounds pointers
- integer divide-by-zero
- writing read-only data
- out-of-bounds shift

undefined behavior

why undefined behavior?

different architectures work differently

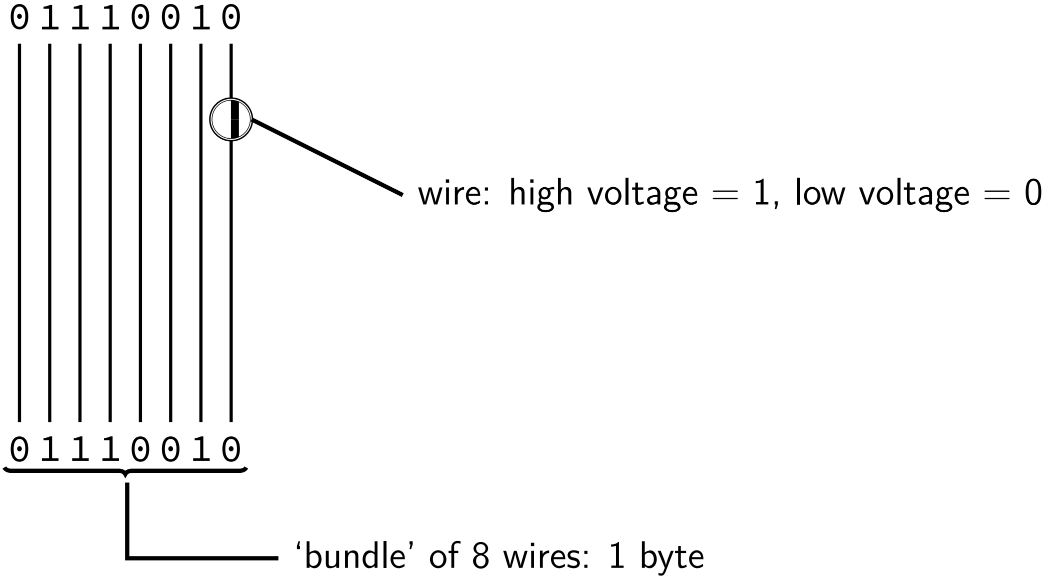
- allow compilers to expose whatever processor does “naturally”
- don’t encode any particular machine in the standard

flexibility for optimizations

Today's topic

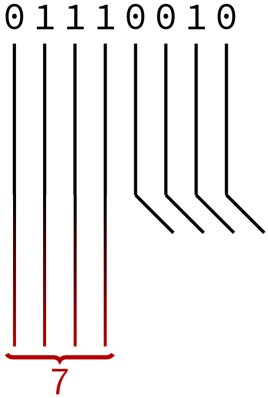
Bitwise operations

moving bits in hardware (one way)



extracting bits in hardware

`0111 0010` = 0x72



simple to do in hardware

extracting hexadecimal nibble (1)

problem: given 0xAB extract 0xA

(hexadecimal digits called “nibbles”)

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

How to extract nibble in software?

- math operation
- bitwise operation

extracing hexadecimal nibbles (2)

```
typedef unsigned char byte;
int get_top_nibble(byte value) {
    return value / 16;
}
```

More efficient: software or hardware?

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

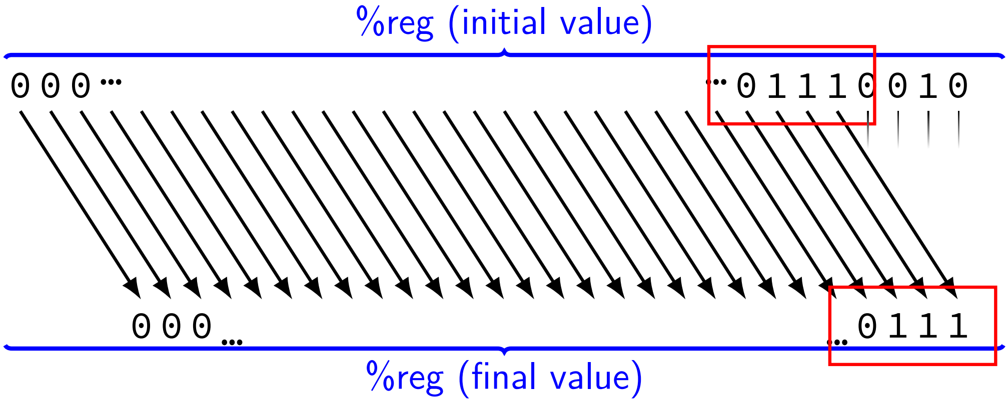
general purpose division is complicated

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

exposing wire selection

x86 instruction: `shr` — shift right

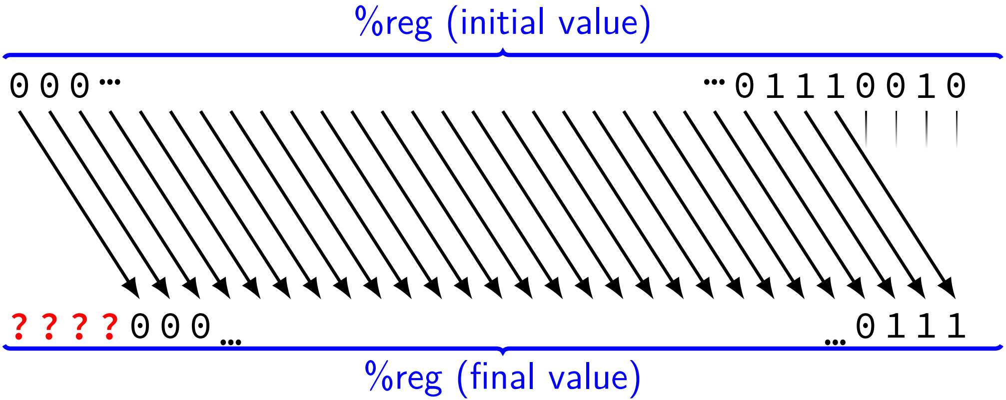
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

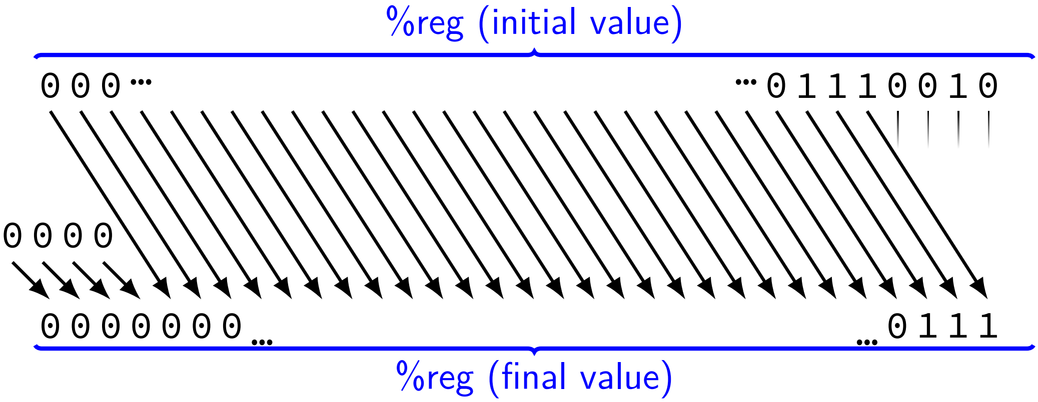
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



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 // move zero-ext. byte to long
 shrl $4, %eax // shift right by 4 bits
 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 // move zero-ext. byte to long
 shrl $4, %eax // shift right by 4 bits
 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 // move zero-ext. byte to long
 shrl $4, %eax // shift right by 4 bits
 ret
```

right shift in C

```
get_top_nibble:
   // eax <- dil (low byte of rdi) w/ zero padding
   movzbl %dil, %eax // move zero-ext. byte to long
   shrl $4, %eax // shift right by 4 bits
   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 vs division by 2^y in C?

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 vs division by 2^y in C?

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

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

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

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

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

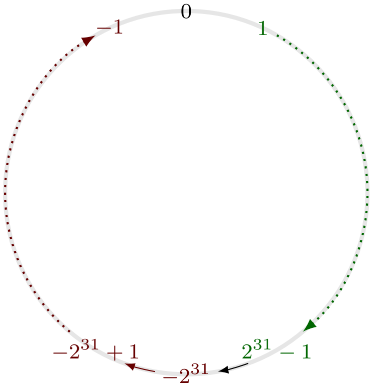
G. a large negative number H. something else

two's complement refresher

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

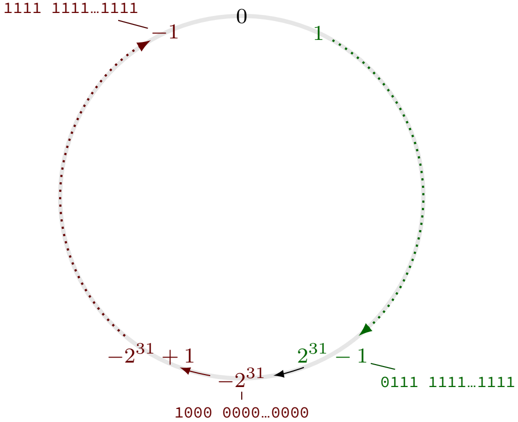
two's complement refresher

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



two's complement refresher

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



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 $-x/2$

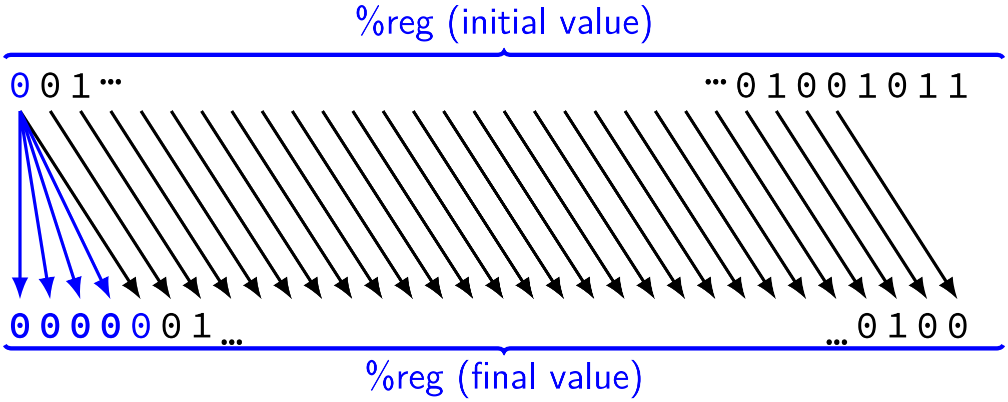
same as right shift by one, adding `1`s instead of `0`s
(except for rounding)

useful operation!

arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

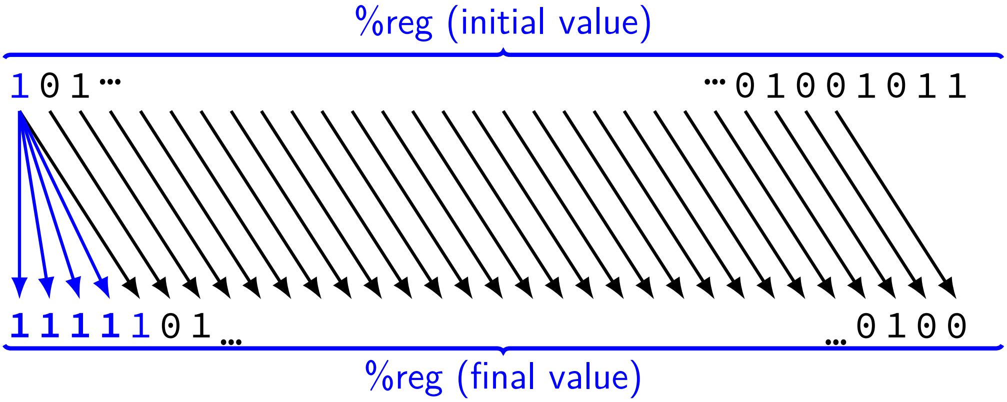
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

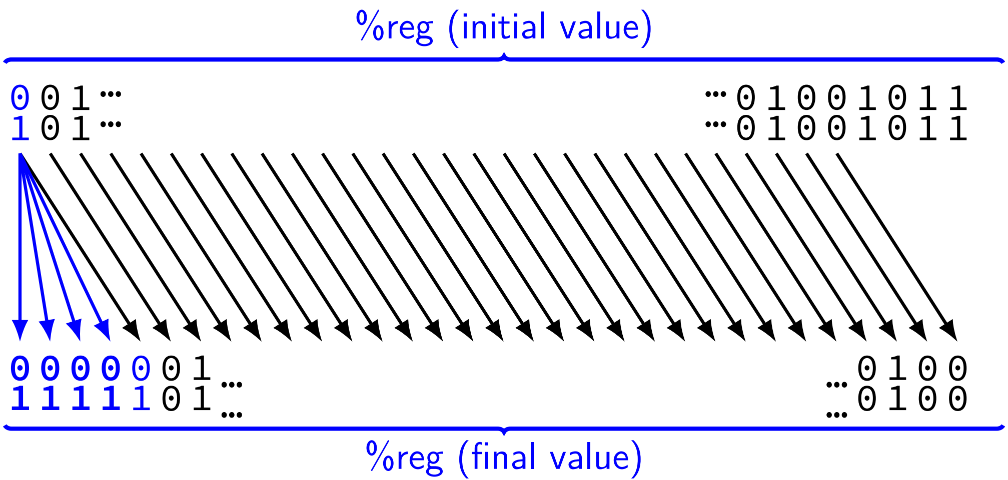
`sar $`*`amount`*`, %reg` (or variable: `sar %cl, %reg`)



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)

$-6/4 = -1.5$ which rounds to -1

however, arithmetic right shift gives different result

divide with proper rounding

C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: "bias" adjustments — described in textbook

example: $-38/8 = -4.75 = -4$ but will round up to -5 with ras

correction: $(-38 + 7)/8 = -31/8 = -3.875$ will round up to -4

divide with proper rounding

C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

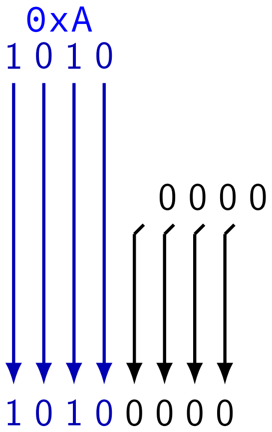
solution: "bias" adjustments — described in textbook

```
// int %eax = int divideBy8(int %edi)
divideBy8: // GCC generated code
    leal    7(%rdi), %eax // %eax <- %edi + 7 (=8-1)
    testl   %edi, %edi    // set cond. codes based on %edi
                          // set SF to 1 if %edi < 0
    cmovns  %edi, %eax    // if (SF == 0) %eax <- %edi
                          // conditional move offset value
    sarl    $3, %eax      // arithmetic shift
    ret
```

example: $-38/8 = -4.75 = -4$ but will round up to -5 with ras

correction: $(-38+7)/8 = -31/8 = -3.875$ will round up to -4

multiplying by 16



0xA × 16 = 0xA0

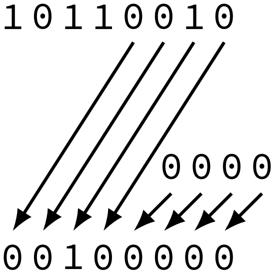
shift left

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

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

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

instead: `value << 4`



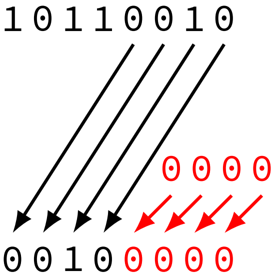
shift left

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

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

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

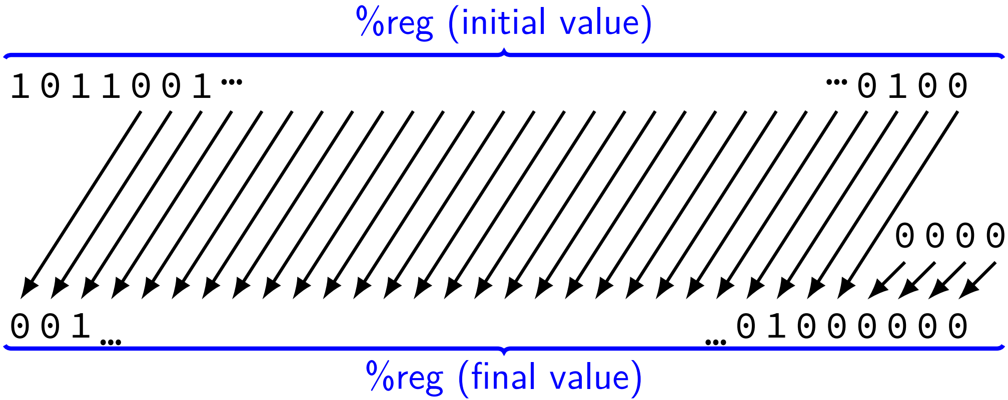
instead: `value << 4`



shift left

x86 instruction: `shl` — shift left

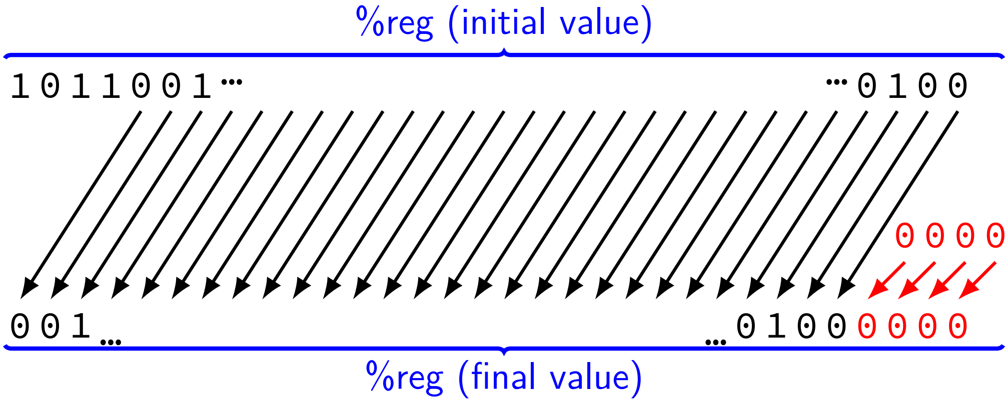
`shl $amount, %reg` (or variable: `shl %cl, %reg`)



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

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

left shift in math

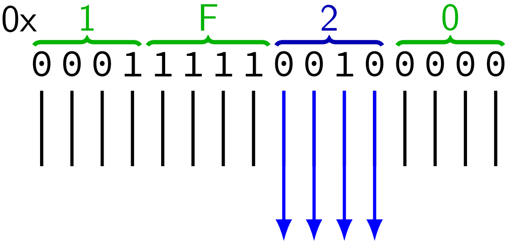
```
1 << 0 == 1               0000 0001
1 << 1 == 2               0000 0010
1 << 2 == 4               0000 0100

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
```

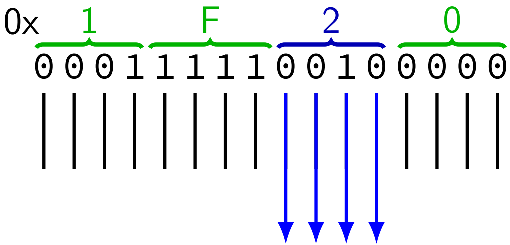
$$x \texttt{<<} y = x \times 2^y$$

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

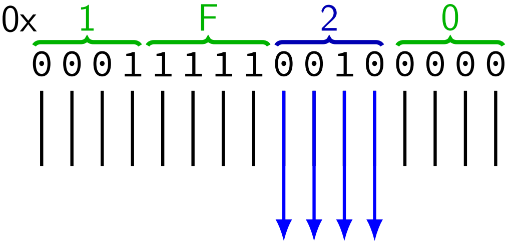
A. `value >> 16` **B.** `value >> 4` **C.** `value << 2` **D.** `value << 4`

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;
} // both use modulo, comes from divide, is slow
```

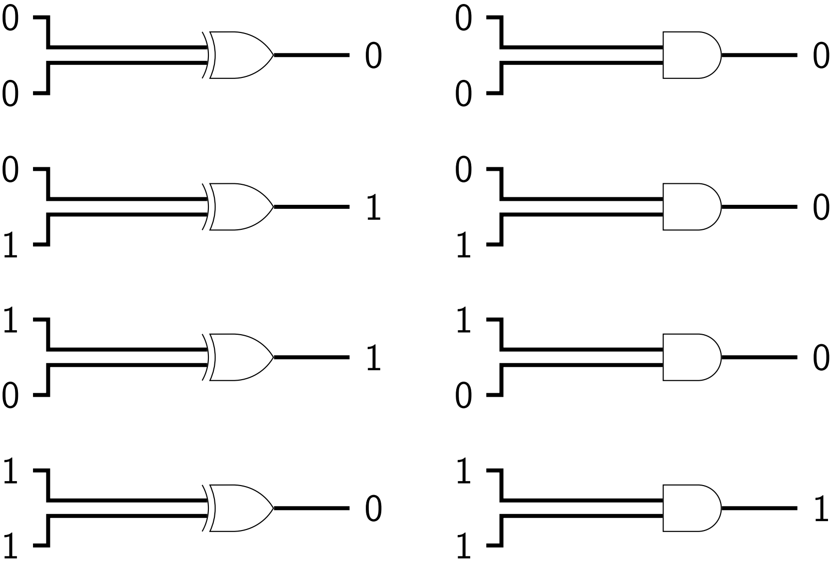
manipulating bits?

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

0xABCD & 0x0F0F == 0x0B0D
```

bitwise AND — &

Treat value as array of bits

```
1 & 1 == 1

1 & 0 == 0

0 & 0 == 0

2 & 4 == 0

10 & 7 == 2

0xABCD & 0x0F0F == 0x0B0D
```

	...	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`

`0xABCD & 0x0F0F == 0x0B0D`

	…	0	0	1	0
&	…	0	1	0	0
	…	0	0	0	0

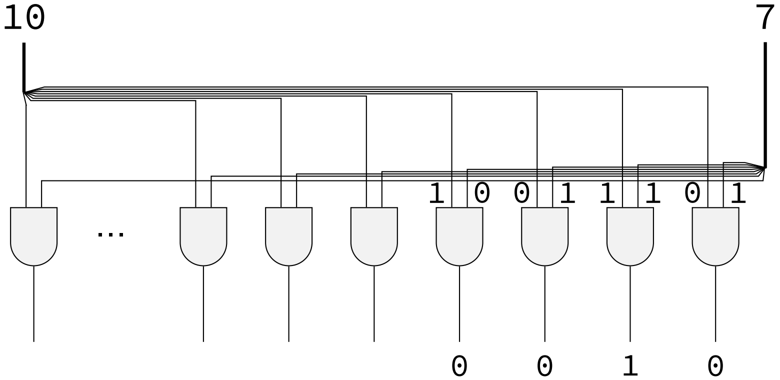
	…	1	0	1	0
&	…	0	1	1	1
	…	0	0	1	0

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
}
```

Bits: aaaabbbbccccdddd → aaaabbbbcccc → 00000000cccc

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

Bits: 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

0xABCD | 0x0F0F == 0xAFCF
```

```
    … 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

0xABCD ^ 0x0F0F == 0xA4C2
```

	…	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 — ~

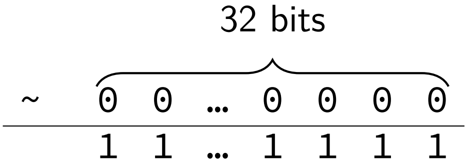
~ (‘complement’) is bitwise version of `!`:

```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF
```
(aka −1)

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



negation / not — ~

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

```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)

~2 == (int) 0xFFFFFFFD (aka −3)

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

32 bits

~ 0	0	…	0	0	0	0
1	1	…	1	1	1	1

bit-puzzles

lab and hw 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)` = if (x) y else 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)` = if (x) y else 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)` = if x then y else z

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

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

one-bit ternary: evaluating example (1)

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

`(x ? y : z)` = if x then y else z

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

x = 1, y = 0, z = 1

`(1 & 0) | ((~1) & 1) =`

`(1 & 0) | (11...1110 & 00...0001) = 0`

one-bit ternary: not general yet

if (x) y else z

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

DOES NOT WORK: $x = 1$, $y = 4$, $z = 2$

`(1 & 4) | ((~1) & 2) =`

`(..0001 & ...0100) | (11...110 & 00...0010) =`

`(0) | (000...0010) = 2` (expected y, which is 4)

multibit ternary

constraint: `x` *is 0 or 1*

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

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

multibit ternary

constraint: `x` *is 0 or 1*

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

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

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

constructing masks

constraint: `x` *is 0 or 1*

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

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

"keep certain bits"

constructing masks

constraint: `x` *is 0 or 1*

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