Bitwise 2 / ISAs (start)

Changelog

Changes made in this version not seen in first lecture:

- 4 Feb 2019: exercise (near end): replace abcdef with uvwxyz in exercise + explanation to make it more obviously not hexadecimal

last time

C nits: structs, typedef, malloc/free, short-circuiting

C traps: signed v unsigned; undefined behavior

bitshifting: exposing bit operations to software

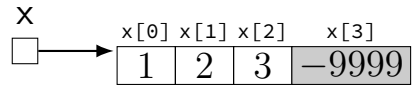
- in hardware: one wire per bit — just ignore some wires?

right shifts, arithmetic and logical

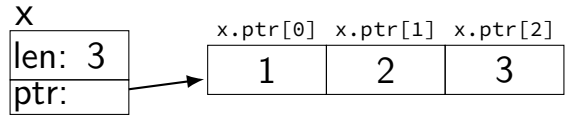
left shifts (briefly)

lists from lists HW

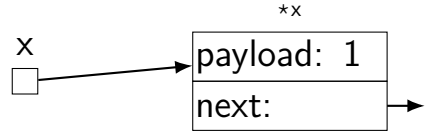
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```




```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```




```
typedef struct node_t {
    short payload;
    struct node_t *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```

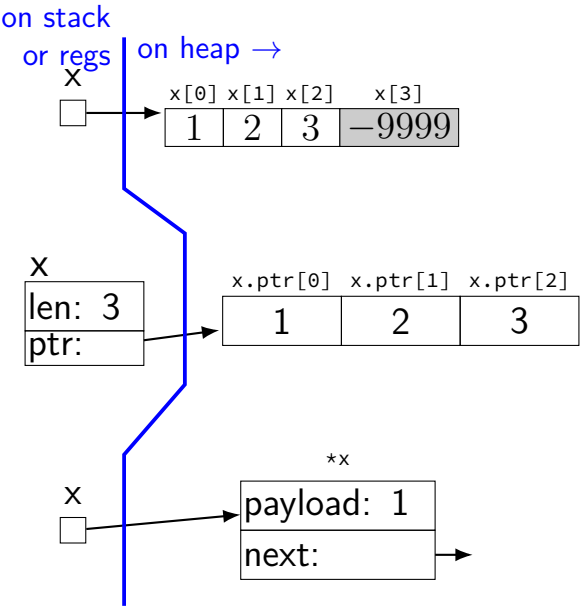


lists from lists HW

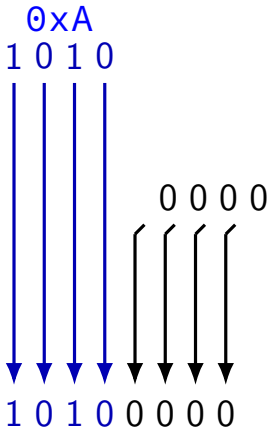
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```

```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```

```
typedef struct node_t {
    short payload;
    struct node_t *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```



multiplying by 16



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

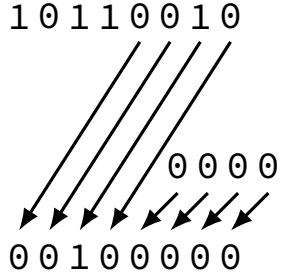
shift left

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

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

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

instead: `value << 4`



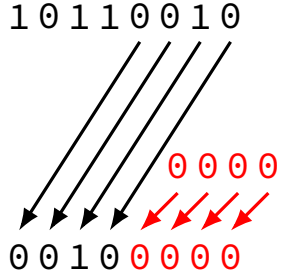
shift left

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

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

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

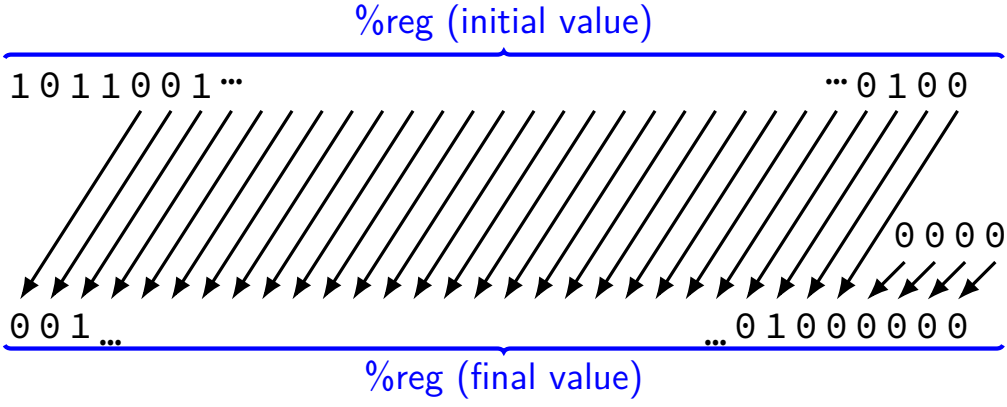
instead: `value << 4`



shift left

x86 instruction: **`shl`** — shift left

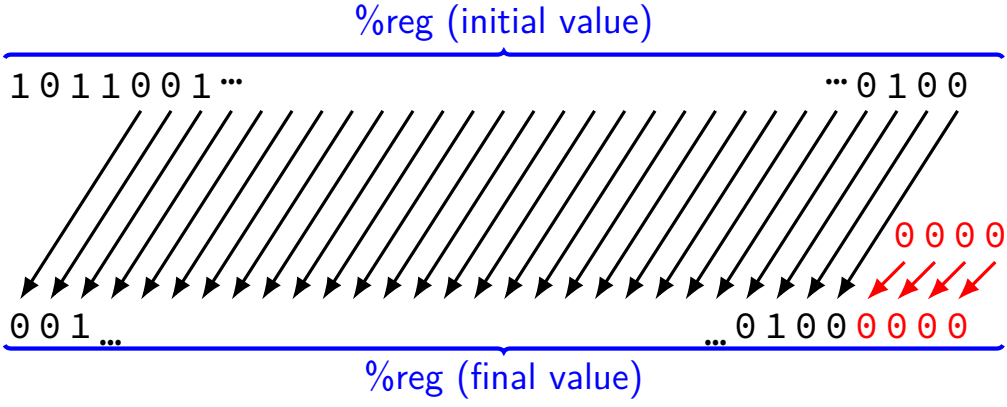
`shl` `$`*`amount`*`, %reg` (or variable: **`shl`** `%cl, %reg`)



shift left

x86 instruction: **`shl`** — shift left

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



left shift in math

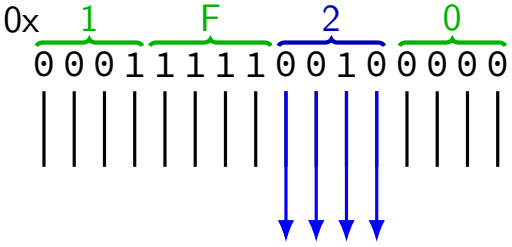
```
1 << 0 == 1                    0000 0001
1 << 1 == 2                    0000 0010
1 << 2 == 4                    0000 0100

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`

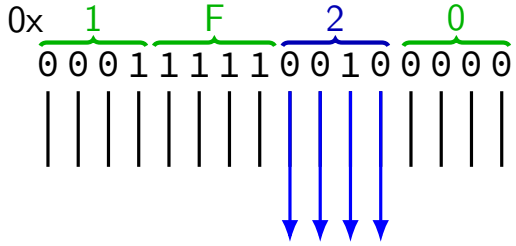
$$x \texttt{ << } y = x \times 2^y$$

extracting nibble from more




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

extracting nibble from more




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

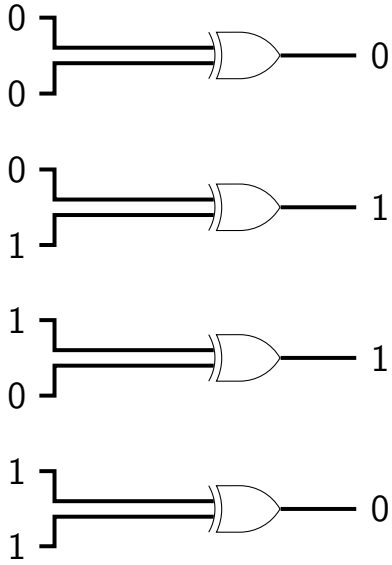
```
// % -- remainder
unsigned extract_second_nibble(unsigned value) {
    return (value / 16) % 16;
}

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
```

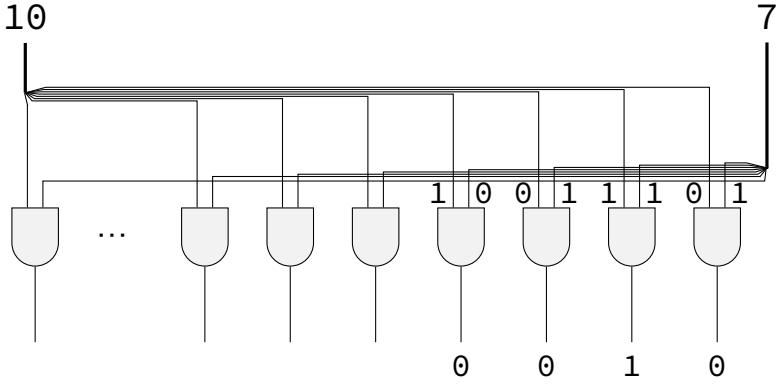
```
    … 1 0 1 0
&   … 0 1 1 1
  ---------
    … 0 0 1 0
```

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
```

and/or/xor

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 — ^

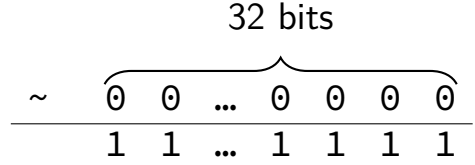
```
1 ^ 1 == 0

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 `!`:

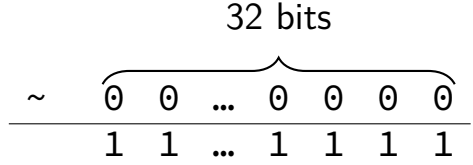
```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)
```



negation / not — ~

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

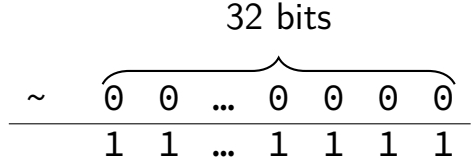
```
!0 == 1

!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; }
```

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: want `1111111111…1`

if x = 0: want `0000000000…0`

mask: x

constructing other masks

constraint: x *is 0 or 1*

`(x ? 0 : z)`

if x = 0: want `1111111111…1`

if x = 1: want `0000000000…0`

mask: `-(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`

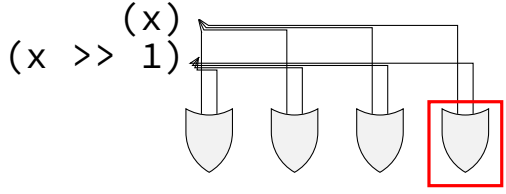
```
(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1
```

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



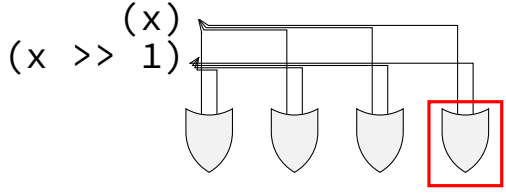
wasted work (2)

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

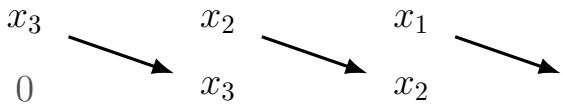
performing 3 bitwise ors

...each bitwise or does 4 OR operations

but only result of one of the 4!

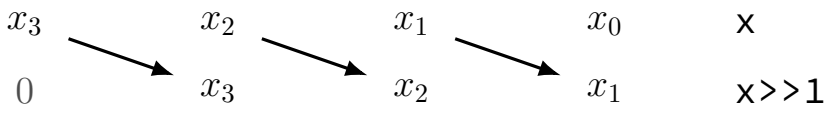


any-bit: looking at wasted work



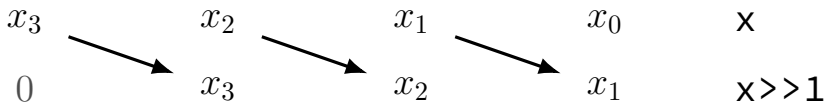

```
y=(x|x>>1)
```

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: $x_3|x_2|x_1|x_0$

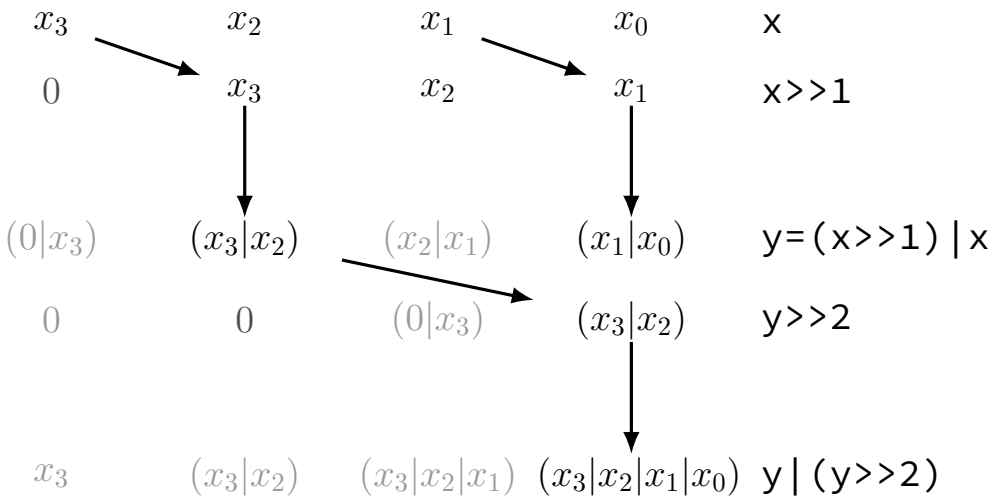
previously:

compute x | (x>>1) for $x_1|x_0$;

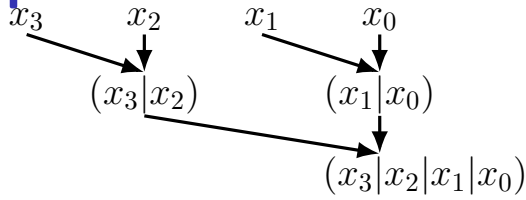
(x>>2) | (x>>3) for $x_3|x_2$

observation: got both parts with just x | (x>>1)

any-bit: divide and conquer



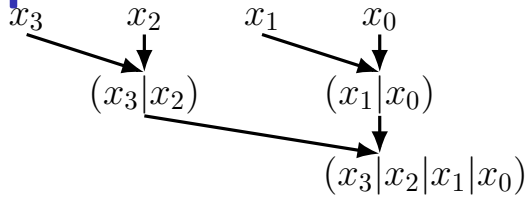
any-bit: divide and conquer



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



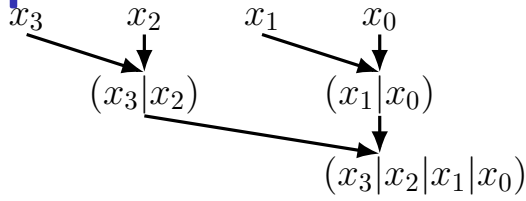
four-bit input $x = x_3x_2x_1x_0$

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 $x = x_3x_2x_1x_0$

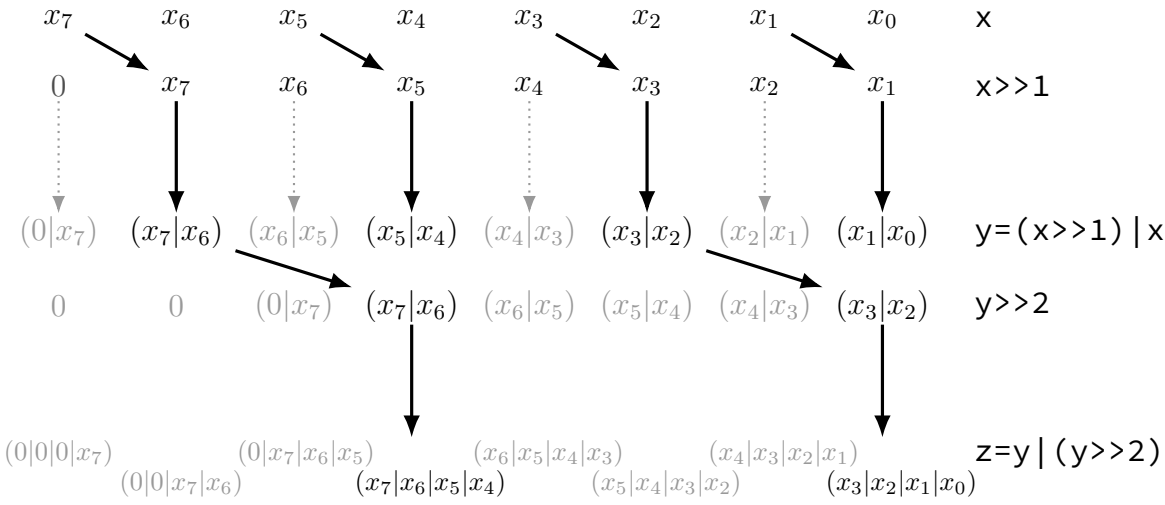
`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

`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;
}
```

parallel operations

key observation: bitwise and, or, etc. do many things in parallel

can have single instruction do work of a loop

more than just bitwise operations:

e.g. "add four pairs of values together"

later: single-instruction, multiple data (SIMD)

base-10 parallelism

compute 14 + 23 and 13 + 99 in parallel?

```
  000014000013
+ 000023000099
----------------
  000037000114
```

14+23 = 37 and 13 + 99 = 114 — one add!

apply same principle in binary?

base-2 parallelism

compute $110_{\text{TWO}} + 011_{\text{TWO}}$ and $010_{\text{TWO}} + 101_{\text{TWO}}$ in parallel?

```
  000110000010 (base 2)
+ 000011000101
----------------
  001001000111
```

$110_{\text{TWO}} + 011_{\text{TWO}} = 1001_{\text{TWO}}$; $010_{\text{TWO}} + 101_{\text{TWO}} = 111_{\text{TWO}}$

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? (bits $uvwxyz$ become $uvwxzy$)

```
/* 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));
    //      ^^^^^^^^^^^^^^
    //       uvwxyz --> 0uvwxy -> 00000y

    //                        ^^^^^^^^^^
    //       uvwxyz --> uvwxy0

    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //       00000y | uvwxy0 = uvwxyy
```

version B

```
/* version B */
    return ((x >> 1) & 1) | ((x << 1) & (~2)) | (x & (~3));
    //      ^^^^^^^^^^^^^^^
    //       uvwxyz --> 0uvwxy --> 00000y

    //                          ^^^^^^^^^^^^^^^
    //       uvwxyz --> vwxyz0 --> vwxy00

    //                                            ^^^^^^^^^
    //       uvwxyz -->             uvwx00
```

version C

```
/* version C */
    return (x & (~3)) | ((x & 1) << 1) | ((x >> 1) & 1);
    //      ^^^^^^^^^^
    //       uvwxyz -->             uvwx00

    //                    ^^^^^^^^^^^^^^
    //       uvwxyz --> 00000z --> 0000z0

    //                                    ^^^^^^^^^^^^
    //       uvwxyz --> 0uvwxy --> 00000y
```

version D

```
/* version D */
    return (((x & 1) << 1) | ((x & 3) >> 1)) ^ x;
    //      ^^^^^^^^^^^^^^^
    //       uvwxyz --> 00000z --> 0000z0

    //                          ^^^^^^^^^^^^^^
    //       uvwxyz --> 0000yz --> 00000y

    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //      0000zy ^ uvwxyz --> uvwx(z XOR y)(y XOR z)
```

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;
```

exercise

Which of these are true only if x has all of bit 0, 3, 6, and 9 set (where bit 0 = least significant bit)?

```
/* version A */
    x = (x >> 6) & x;
    x = (x >> 3) & x;
    return x & 1;

/* version B */
    return ((x >> 9) & 1) & ((x >> 6) & 1) & ((x >> 3) & 1) &

/* version C */
    return (x & 0x100) & (x & 0x40) & (x & 0x04) & (x & 0x01);

/* version D */
    return (x & 0x145) == 0x145;
```