bitwise (finish) / ISAs

last time

bitshifting

- "move" bits left/right $\approx$ div/mult by power of two
- decision: what to put in extra bits
- arithmetic right (signed): copy sign bit
- logical right (unsigned): add zeroes

bitwise operators

- treat operands as arrays of bits, do operation on each pair of bits

mask = # with 1 (or 0) where we want to do something

- & (and): something = extract bit (or clear bit)
- | (or): something = set bit
- ^(xor): something = flip bit

divide and conquer to solve bit puzzles

aside: usually and not faster than add

- probably b/c processor designers thought add more important than and

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

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

- another solution if you have − or + (bang in lab)

what if we don't have ! or − or +

- more like what real hardware components to work with are

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

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

- another solution if you have − or + (bang in lab)

what if we don't have ! or − or +

- more like what real hardware components to work with are

how do we solve is x is, say, 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 solution if you have − or + (bang in lab)

what if we don't have ! or − or +

- more like what real hardware components to work with are

how do we solve is x is, say, 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`

distributive property

wasted work (1)

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

in general: `(x & 1) | (y & 1) == (x | y) & 1`

distributive property

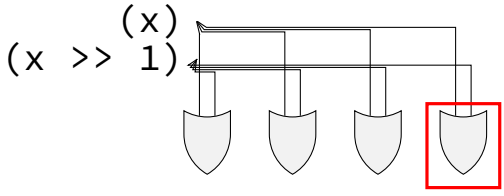
```
(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1
```

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



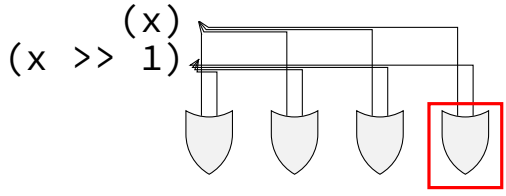
wasted work (2)

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

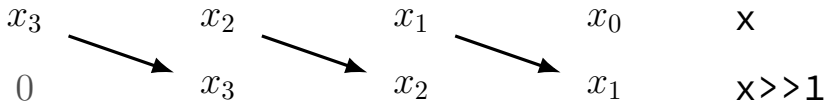
performing 3 bitwise ors

...each bitwise or does 4 OR operations

but only result of one of the 4!

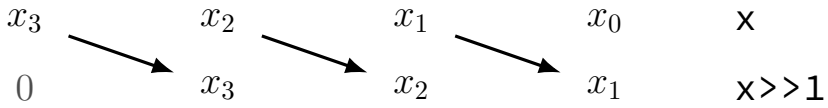


any-bit: looking at wasted work



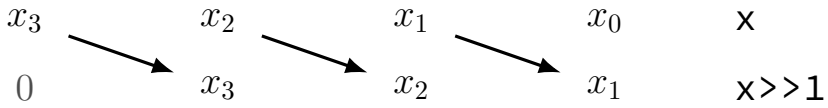

```
y=(x|x>>1)
```

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: $x_3|x_2|x_1|x_0$

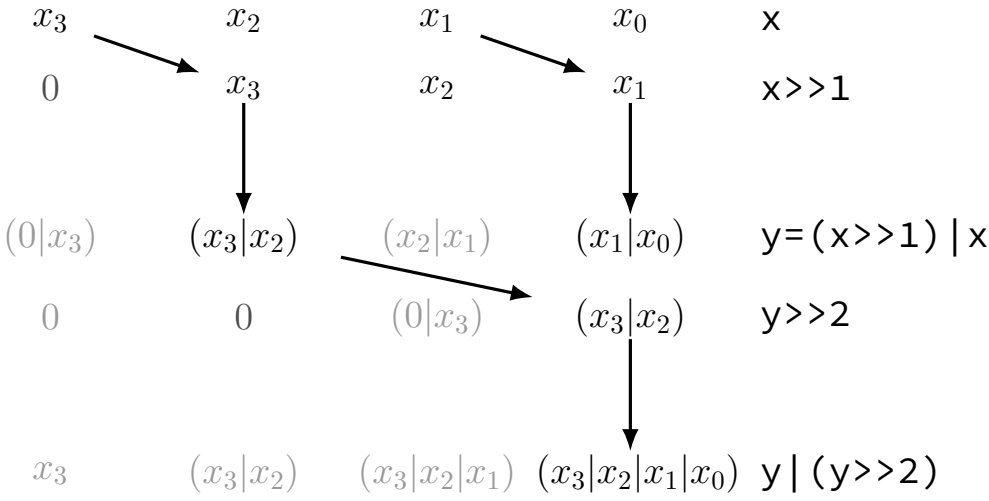
previously:

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

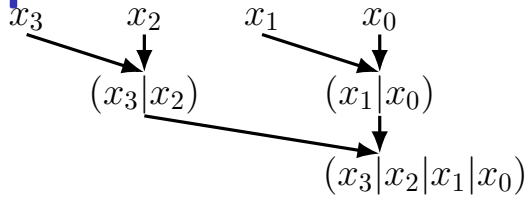
(x>>2)|(x>>3) for $x_3|x_2$

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

any-bit: divide and conquer



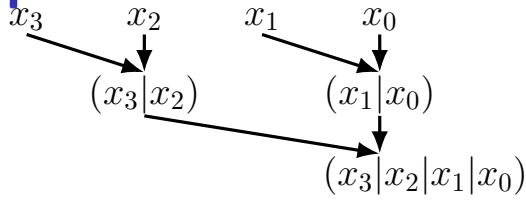
any-bit: divide and conquer



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



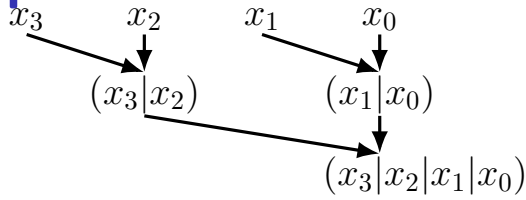
four-bit input $x = x_3x_2x_1x_0$

`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 $x = x_3x_2x_1x_0$

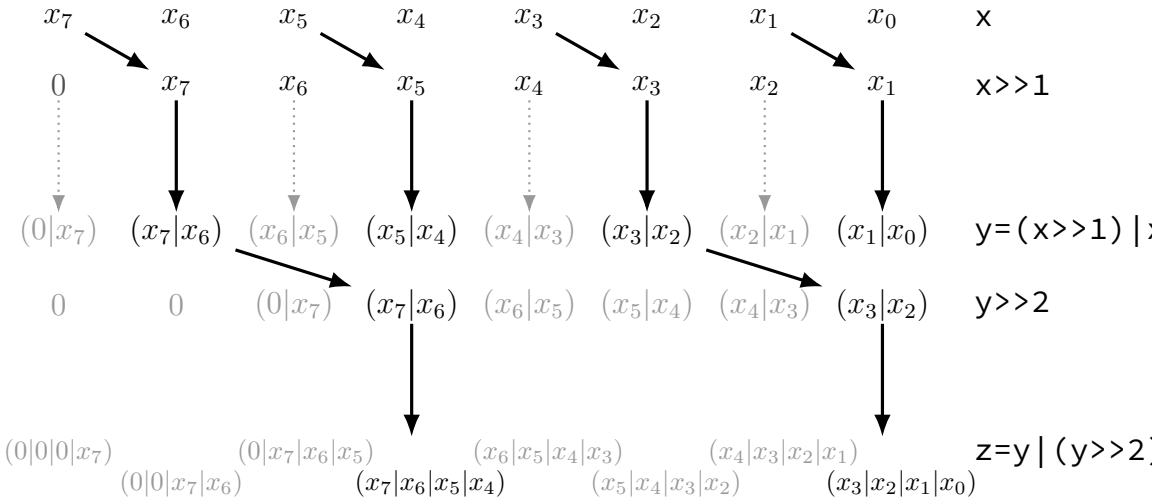
`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

`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)
```

exercise

Which of these will swap least significant and second least significant 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 2?

ISAs being manufactured today

(ISA = instruction set architecture)

x86 — dominant in desktops, servers

ARM — dominant in mobile devices

POWER — Wii U, IBM supercomputers and some servers

MIPS — common in consumer wifi access points

SPARC — some Oracle servers, Fujitsu supercomputers

z/Architecture — IBM mainframes

Z80 — TI calculators

SHARC — some digital signal processors

RISC V — some embedded

...

microarchitecture v. instruction set

microarchitecture — design of the hardware

- "generations" of Intel's x86 chips
- different microarchitectures for very low-power versus laptop/desktop
- changes in performance/efficiency

instruction set — interface visible by software

- what matters for software compatibility
- many ways to implement (but some might be easier)

exercise

which of the following changes to a processor are *instruction set* changes?

A. increasing the number of registers available in assembly

B. decreasing the runtime of the `add` instruction

C. making the machine code for `add` instructions shorter

D. removing a multiply instruction

E. allowing the add instruction to have two memory operands (instead of two register operands))

instruction set architecture goals

exercise: what are some goals to have when designing an *instruction set*?

ISA variation

instruction set	instr. length	# normal registers	*approx.* # instrs.
x86-64	1–15 byte	16	1500
Y86-64	1–10 byte	15	18
ARMv7	4 byte*	16	400
POWER8	4 byte	32	1400
MIPS32	4 byte	31	200
Itanium	41 bits*	128	300
Z80	1–4 byte	7	40
VAX	1–14 byte	8	150
z/Architecture	2–6 byte	16	1000
RISC V	4 byte*	31	500*

other choices: condition codes?

instead of:

```
cmpq %r11, %r12
je somewhere
```

could do:

```
/* _B_ranch if _EQ_ual */
beq  %r11, %r12, somewhere
```

other choices: addressing modes

ways of specifying operands. examples:

x86-64: `10(%r11,%r12,4)`

ARM: `%r11 << 3` (shift register value by constant)

VAX: `((%r11))` (register value is pointer to pointer)

other choices: number of operands

`add src1, src2, dest`

ARM, POWER, MIPS, SPARC, ...

`add src2, src1=dest`

x86, AVR, Z80, ...

VAX: both

CISC and RISC

RISC — Reduced Instruction Set Computer

reduced from what?

CISC and RISC

RISC — Reduced Instruction Set Computer

reduced from what?

CISC — Complex Instruction Set Computer

some VAX instructions

`MATCHC` *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

`POLY` *x*, *coefficientsLen*, *coefficientsPtr*
Evaluate the polynomial whose coefficients are pointed to by *coefficientPtr* at the value *x*.

`EDITPC` *sourceLen*, *sourcePtr*, *patternLen*, *patternPtr*
Edit the string pointed to by *sourcePtr* using the pattern string specified by *patternPtr*.

microcode

`MATCHC` *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

loop in hardware???

typically: lookup sequence of microinstructions (“microcode”)

secret simpler instruction set

Why RISC?

complex instructions were usually not faster

- (even though programs with simple instructions were bigger)

complex instructions were harder to implement

compilers were replacing hand-written assembly

- correct assumption: almost no one will write assembly anymore
- incorrect assumption: okay to recompile frequently

typical RISC ISA properties

fewer, simpler instructions

seperate instructions to access memory

fixed-length instructions

more registers

no “loops” within single instructions

no instructions with two memory operands

few addressing modes

typical RISC ISA properties

fewer, simpler instructions

seperate instructions to access memory

fixed-length instructions

more registers

no “loops” within single instructions

no instructions with two memory operands

few addressing modes

is CISC the winner?

well, can't get rid of x86 features

- backwards compatibility matters

more application-specific instructions

but...compilers tend to use more RISC-like subset of instructions

modern x86: often convert to RISC-like "microinstructions"

- sounds really expensive, but ...
- lots of instruction preprocessing used in 'fast' CPU designs (even for RISC ISAs)

ISAs: who does the work?

CISC-like (harder to make hardware, easier to use assembly)

choose instructions with particular assembly language in mind?
hardware designer provides operations assembly-writers wants

let the hardware worry about optimizing it?

RISC-like (easier to make hardware, harder to use assembly)

choose instructions with particular HW implementation in mind?
hardware designer exposes things it can do efficiently to assembly-writers

building blocks for compiler to make efficient programs?

note: *general* differences — no firm RISC v. CISC line

ISAs: who does the work?

CISC-like (harder to make hardware, easier to use assembly)

choose instructions with particular assembly language in mind?
hardware designer provides operations assembly-writers wants

let the hardware worry about optimizing it?

RISC-like (easier to make hardware, harder to use assembly)

choose instructions with particular HW implementation in mind?
hardware designer exposes things it can do efficiently to assembly-writers

building blocks for compiler to make efficient programs?

note: *general* differences — no firm RISC v. CISC line

ISAs: who does the work?

CISC-like (harder to make hardware, easier to use assembly)

choose instructions with particular assembly language in mind?
hardware designer provides operations assembly-writers wants

let the hardware worry about optimizing it?

RISC-like (easier to make hardware, harder to use assembly)

choose instructions with particular HW implementation in mind?
hardware designer exposes things it can do efficiently to assembly-writers

building blocks for compiler to make efficient programs?

note: *general* differences — no firm RISC v. CISC line

backup slides

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

byte swap: (x86: `bswap`)

other choices: instruction complexity

instructions that write multiple values?

- x86-64: `push`, `pop`, `movsb`, ...

more?

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;
```