bitwise 2 / ISAs

last time

right shift

- logical (unsigned): shift in zeroes
- arithmetic (signed): shift in zero/one depending if negative
- equiv to division with different rounding

left shift $\approx$ multiplication by power of two

bitwise operators and masks

- mask "selects" certain bits
- bitwise and: certain bits to keep (1) / clear (0)
- bitwise or: certain bits to set (1) / leave unchanged (0)
- bitwise xor: certain bits to flip (1) / leave unchanged (0)

bitwise divide and conquer

- ternary operator example: x ? y : z
  - syntax for if (x) y else z in C/C++/Java
- result = result-or-zero | result-or-zero trick
- (9am) -1 –> 1111..1111 trick

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

→ `(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)
```

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

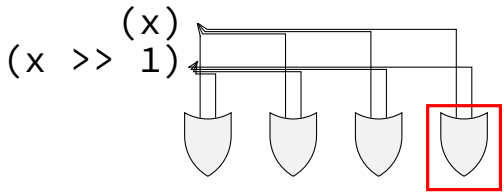
```
(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1
```

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



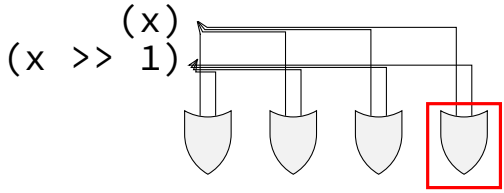
wasted work (2)

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

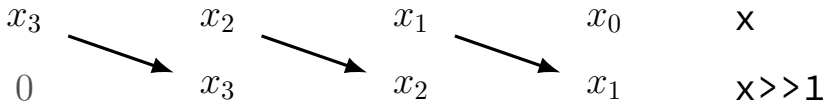
performing 3 bitwise ors

...each bitwise or does 4 OR operations

but only result of one of the 4!

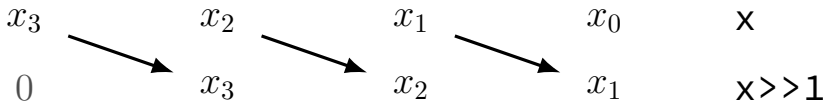


any-bit: looking at wasted work



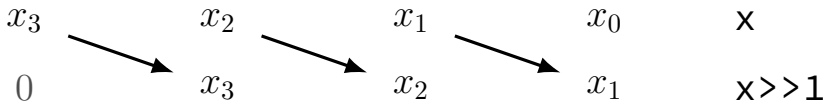

```
y=(x|x>>1)
```

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)`

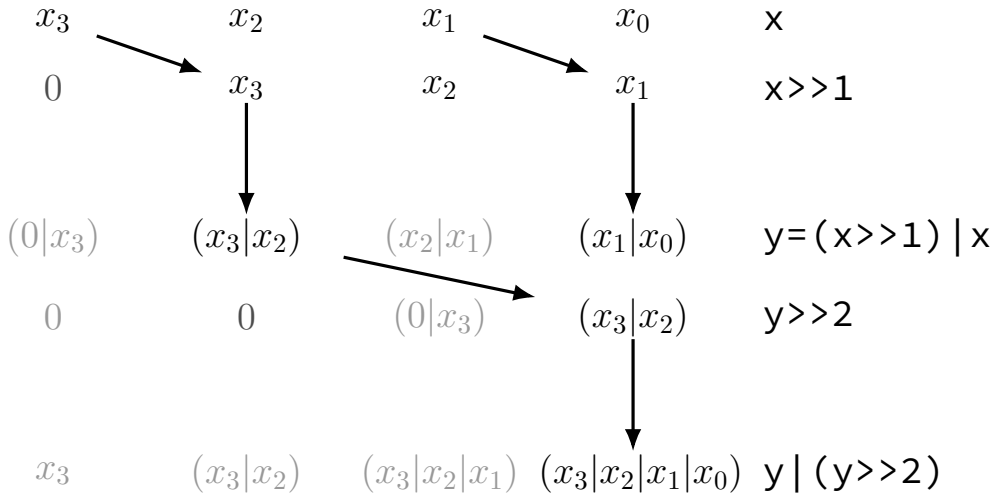
any-bit: looking at wasted work



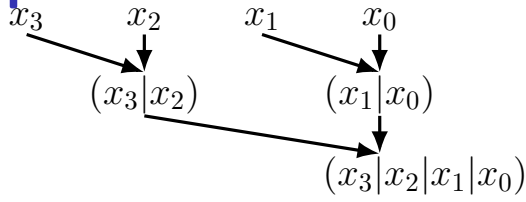
$(0|x_3)$ $(x_3|x_2)$ $(x_2|x_1)$ $(x_1|x_0)$ y=(x|x>>1)

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



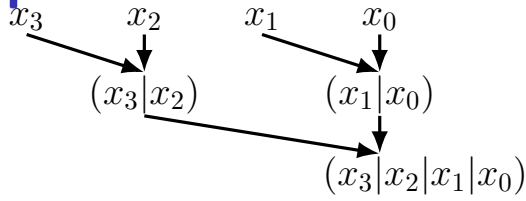
any-bit: divide and conquer



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



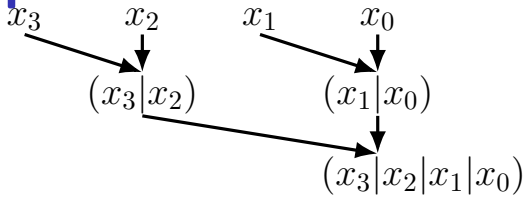
four-bit input $x = x_3x_2x_1x_0$

`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 $x = x_3x_2x_1x_0$

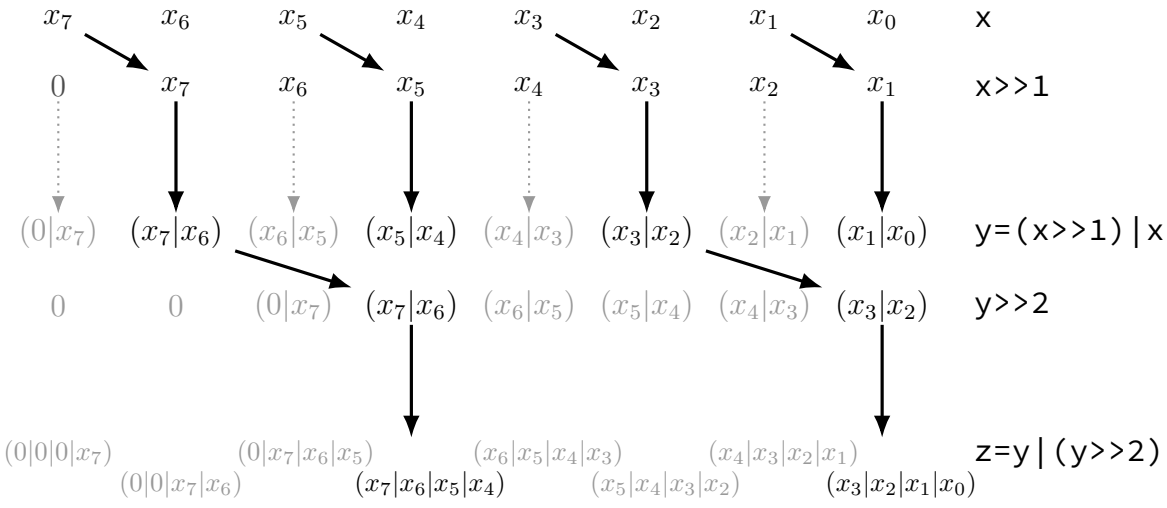
`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

`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

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;
```

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))

ISA "extensions"

I've been saying x86-64, ARM is an ISA

but there have been new instructions

(that weren't supported by original x86-64 or ARM processors)

really a bunch of variants of x86-64 (or ARM or ...), each of which is a different ISA

primary purpose of new processor designs usually to make non-ISA changes

ISA extensions won't improve performance of existing compiled code

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

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

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
```

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

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`)

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

other choices: instruction complexity

instructions that write multiple values?

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

more?