Bitwise
ISAs

February 7, 2023

last lecture topics

C data types, strings
  - pointer stores beginning of array
  - pointer arithmetic: $*(\text{array} + 5) = \text{array}[5]$

C evolution and standards, undefined behavior

bitwise - extracting nibble

right shift
  - logical (unsigned) shr: shift in 0's
  - arithmetic (signed) sar: shift in sign bit
  - equivalent to division by 2^y, use bias for rounding

left shift, equivalent to multiplication by 2^y

last lecture topics

- bitwise operators and masks
  - bitwise and: keep (1) / clear (0)
  - bitwise or: set (1) / leave unchanged (0)
  - bitwise xor: flip (1) / leave unchanged (0)
  - bitwise not: negate each bit
  - efficient in hardware

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
```

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

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

example 1

Calculate $29 \;/\; 8 = 3$

0000 0000 0000 0000 0000 0000 0001 1101 $= 29$

0000 0000 0000 0000 0000 0000 0000 0011~~101~~ $= 29 >> 3 = 3$

x	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**
x/8	3	3.1	3.2	3.3	3.5	3.6	3.7	3.8	4	4.1	4.2	4.3
x div 8	3	3	3	3	3	3	3	3	4	4	4	4
x $>>$ 3	3	3	3	3	3	3	3	3	4	4	4	4

example 2

Calculate -29 / 8 = -3

with no bias adjustment:

1111 1111 1111 1111 1111 1111 1110 0011 = -29

1111 1111 1111 1111 1111 1111 1111 1100~~011~~ = -29 >> 3 = -4

with +7 bias adjustment:

1111 1111 1111 1111 1111 1111 1110 1010 = -29 + 7 = -22

1111 1111 1111 1111 1111 1111 1111 1101~~010~~ = -22 >> 3 = -3

x	**-24**	**-25**	**-26**	**-27**	**-28**	**-29**	**-30**	**-31**	**-32**	**-33**	**-3**
x/8	-3	-3.1	-3.2	-3.3	-3.5	-3.6	-3.7	-3.8	-4	-4.1	-4
x div 8	-3	-3	-3	-3	-3	-3	-3	-3	-4	-4	-4
x >> 3	-3	-4	-4	-4	-4	-4	-4	-4	-4	-5	-5
(x+7)>>3	-3	-3	-3	-3	-3	-3	-3	-3	-4	-4	-4

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
```

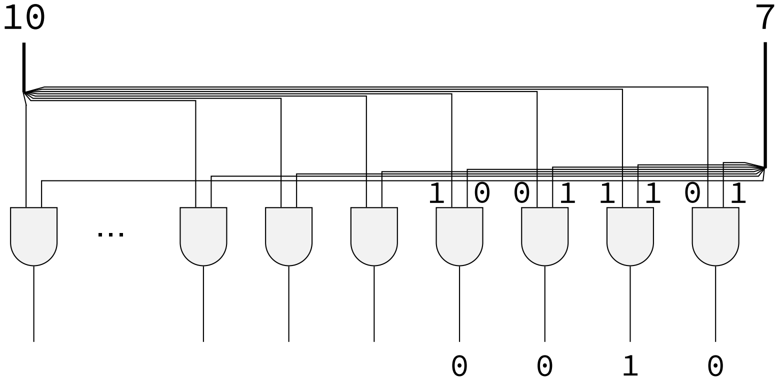
```
      … 1 0 1 0
&     … 0 1 1 1
---------------
      … 0 0 1 0
```

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

&

conditionally clear bit
conditionally keep bit

mask: 0s = clear; 1s = keep
e.g. 101010101…=
clear every other bit

OR	**0**	**1**
0	0	1
1	1	1

|

conditionally set bit

mask: 1s = set; 0s = keep same
e.g. 101010101…=
set every other bit

XOR	**0**	**1**
0	0	1
1	1	0

^

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)

$$\begin{array}{r}
\sim \; \overbrace{0 \; 0 \; \ldots \; 0 \; 0 \; 0 \; 0}^{\text{32 bits}} \\
\hline
1 \; 1 \; \ldots \; 1 \; 1 \; 1 \; 1
\end{array}$$

negation / not — ~

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

```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)

~2 == (int) 0xFFFFFFFD (aka −3)
```

$$\begin{array}{ll} \sim & \overbrace{0\quad 0\quad \ldots\quad 0\quad 0\quad 0\quad 0}^{\text{32 bits}} \\ \hline & 1\quad 1\quad \ldots\quad 1\quad 1\quad 1\quad 1 \end{array}$$

negation / not — ~

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

`!0 == 1`

`!notZero == 0`

`~0 == (int) 0xFFFFFFFF` (aka −1)

`~2 == (int) 0xFFFFFFFD` (aka −3)

$$\begin{array}{rcccccc} & \multicolumn{6}{c}{\overbrace{\hphantom{0\quad 0\quad \dots\quad 0\quad 0\quad 0\quad 0}}^{32\text{ bits}}} \\ \sim & 0 & 0 & \dots & 0 & 0 & 0 \; 0 \\ \hline & 1 & 1 & \dots & 1 & 1 & 1 \; 1 \end{array}$$

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

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)` or `(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 $x = \cancel{1}\ 0$: want `1111111111...1`

if $x = \cancel{0}\ 1$: want `0000000000...0`

mask: $\cancel{-x}$

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

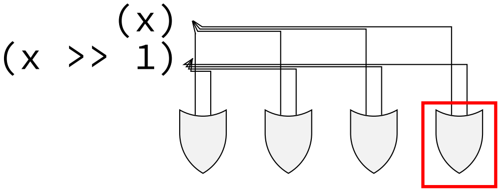
`(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1`

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



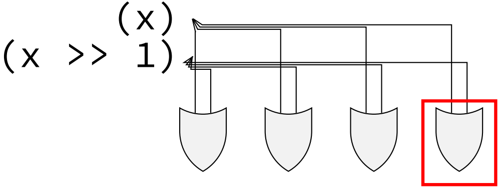
wasted work (2)

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

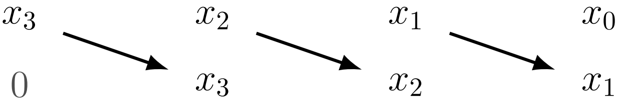
performing 3 bitwise ors

…each bitwise or does 4 OR operations

but only result of one of the 4!

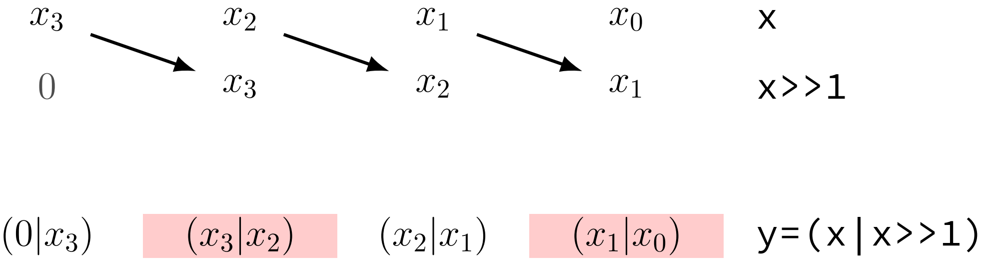


any-bit: looking at wasted work



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

any-bit: looking at wasted work

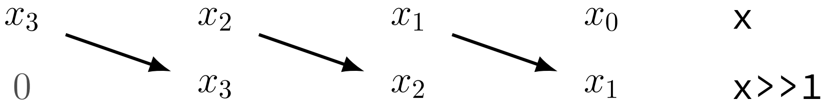


x_3 x_2 x_1 x_0 `x`

0 x_3 x_2 x_1 `x>>1`

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

any-bit: looking at wasted work



x_3 x_2 x_1 x_0 x

0 x_3 x_2 x_1 x>>1

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

final value wanted: $x_3|x_2|x_1|x_0$

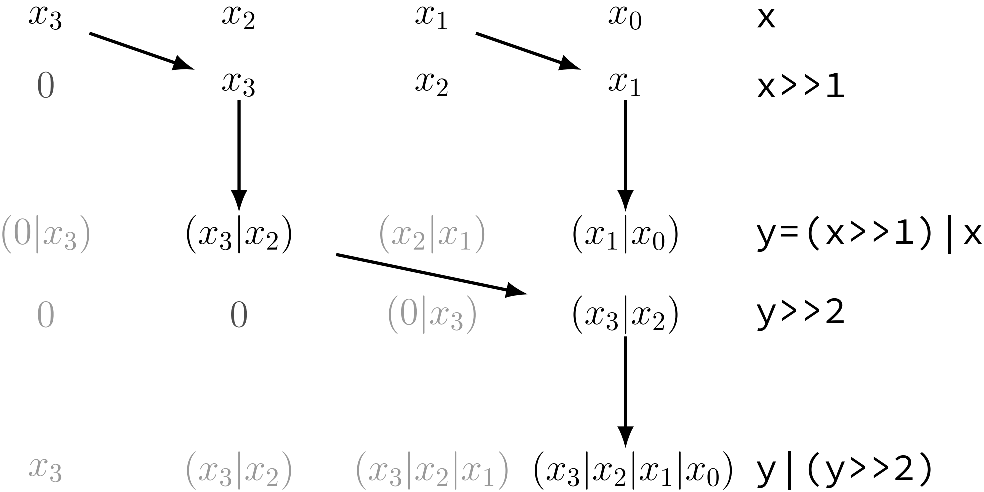
previously:

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



x_3 x_2 x_1 x_0 `x`

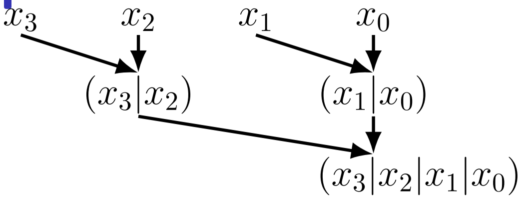
0 x_3 x_2 x_1 `x>>1`

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

0 0 $(0|x_3)$ $(x_3|x_2)$ `y>>2`

x_3 $(x_3|x_2)$ $(x_3|x_2|x_1)$ $(x_3|x_2|x_1|x_0)$ `y|(y>>2)`

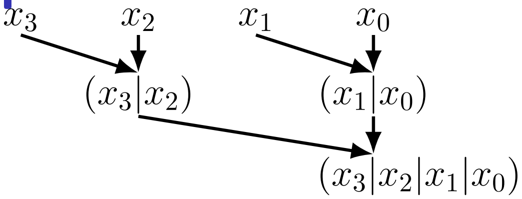
any-bit: divide and conquer



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



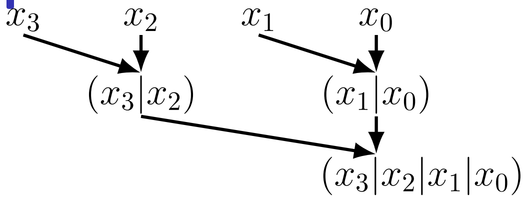
four-bit input $x = x_3x_2x_1x_0$

`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 $x = x_3x_2x_1x_0$

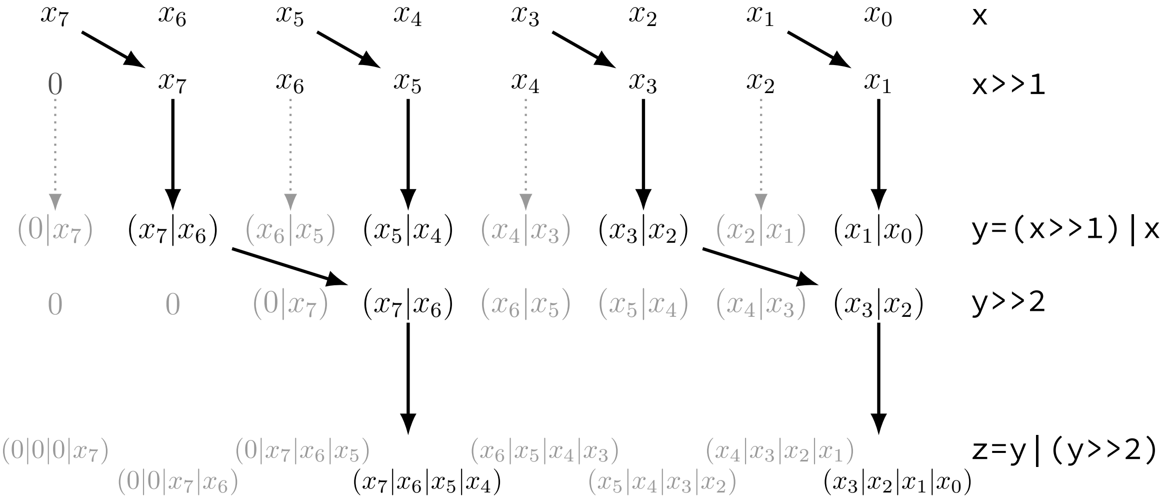
`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

`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



x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 x

0 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x>>1

$(0|x_7)$ $(x_7|x_6)$ $(x_6|x_5)$ $(x_5|x_4)$ $(x_4|x_3)$ $(x_3|x_2)$ $(x_2|x_1)$ $(x_1|x_0)$ y=(x>>1)|x

0 0 $(0|x_7)$ $(x_7|x_6)$ $(x_6|x_5)$ $(x_5|x_4)$ $(x_4|x_3)$ $(x_3|x_2)$ y>>2

$(0|0|0|x_7)$ $(0|0|x_7|x_6)$ $(0|x_7|x_6|x_5)$ $(x_7|x_6|x_5|x_4)$ $(x_6|x_5|x_4|x_3)$ $(x_5|x_4|x_3|x_2)$ $(x_4|x_3|x_2|x_1)$ $(x_3|x_2|x_1|x_0)$ z=y|(y>>2)

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)
```

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)

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

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

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