



Representations

Prof. David August
COS 217



Goals of Today's Lecture

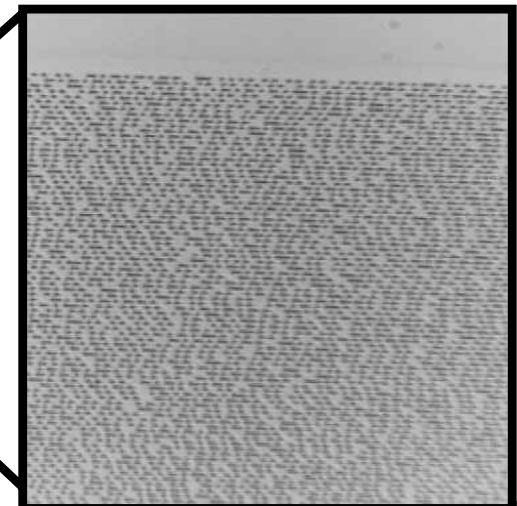
- Representations
 - Why binary?
 - Converting base 10 to base 2
 - Octal and hexadecimal
- Integers
 - Unsigned integers
 - Integer addition, subtraction
 - Signed integers
- C bit operators
 - And, or, not, and xor
 - Shift-left and shift-right
 - Function for counting the number of 1 bits
 - Function for XOR encryption of a message



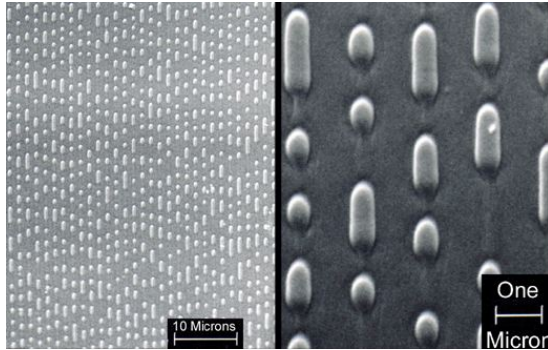
Radiohead - OK Computer CD



3 Miles of Music

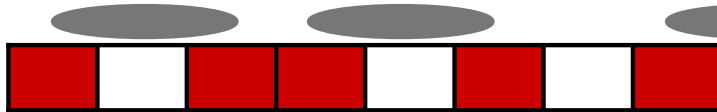


Pits and Lands

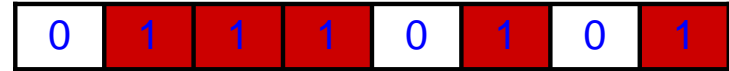


Transition represents a bit state (1/on/red/female/heads)

No change represents other state (0/off/white/male/tails)



Interpretation



As Music:

$01110101_2 = 117/256$ position of speaker

As Number:

$01110101_2 = 1 + 4 + 16 + 32 + 64 = 117_{10} = 75_{16}$

(Get comfortable with base 2, 8, 10, and 16.)

As Text:

$01110101_2 = 117^{\text{th}}$ character in the ASCII codes = "u"

Interpretation – ASCII



ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☉	SOH	033	!	065	A	097	a
002	☼	STX	034	"	066	B	098	b
003	▼	ETX	035	#	067	C	099	c
004	◆	EOT	036	\$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	■	BS	040	(072	H	104	h
009	(tab)	HT	041)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	☺	SO	046	.	078	N	110	n
015	☹	SI	047	/	079	O	111	o
016	▲	DLE	048	0	080	P	112	p
017	▼	DC1	049	1	081	Q	113	q
018	↺	DC2	050	2	082	R	114	r
019	↻	DC3	051	3	083	S	115	s
020	↷	DC4	052	4	084	T	116	t
021	↶	NAK	053	5	085	U	117	u
022	↵	SYN	054	6	086	V	118	v
023	↹	ETB	055	7	087	W	119	w
024	↻	CAN	056	8	088	X	120	x
025	↷	EM	057	9	089	Y	121	y
026	↶	SUB	058	:	090	Z	122	z
027	↵	ESC	059	;	091	[123	{
028	(cursor right)	FS	060	<	092	\	124	
029	(cursor left)	GS	061	=	093]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	␣

Computer Science Building West Wall



Interpretation: Code and Data (Hello World!)



- Programs consist of Code and Data
- Code and Data are Encoded in Bits

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF.....
...
00000260: 5002 0000 0000 0000 006c 6962 632e 736f  P.....libc.so
00000270: 2e36 2e31 0070 7269 6e74 6600 5f5f 6c69  .6.l.printf._li
00000280: 6263 5f73 7461 7274 5f6d 6169 6e00 474c  bc_start_main.GL
00000290: 4942 435f 322e 3200 0000 0200 0200 0000  IBC_2.2.....
...
00000860: 4865 6c6c 6f20 576f 726c 6421 0d00 0000  Hello world!...
...
40000000000000690 <main>:
40000000000000690: 00 10 15 08 80 05      [MII]    alloc r34=ar.pfs,5,4,0
40000000000000696: 30 02 30 00 42 20      mov r35=r12
4000000000000069c: 04 00 c4 00            mov r33=b0
400000000000006a0: 0a 20 81 03 00 24      [MMI]    addl r36=96,r1;;
400000000000006a6: 40 02 90 30 20 00      ld8 r36=[r36]
400000000000006ac: 04 08 00 84            mov r32=r1
400000000000006b0: 1d 00 00 00 01 00      [MFB]    nop.m 0x0
400000000000006b6: 00 00 00 02 00 00      nop.f 0x0
400000000000006bc: b8 fd ff 58            br.call.sptk.many b0=40000000000000460;;
400000000000006c0: 00 08 00 40 00 21      [MII]    mov r1=r32
400000000000006c6: 80 00 00 00 42 00      mov r8=r0
400000000000006cc: 20 02 aa 00            mov.i ar.pfs=r34
400000000000006d0: 00 00 00 00 01 00      [MII]    nop.m 0x0
400000000000006d6: 00 08 05 80 03 80      mov b0=r33
400000000000006dc: 01 18 01 84            mov r12=r35
400000000000006e0: 1d 00 00 00 01 00      [MFB]    nop.m 0x0
400000000000006e6: 00 00 00 02 00 80      nop.f 0x0
400000000000006ec: 08 00 84 00            br.ret.sptk.many b0;;
```

IA-64 Binary (objdump)

Interpretation: Numbers



- Base 10
 - Each digit represents a power of 10
 - $4173 = 4 \times 10^3 + 1 \times 10^2 + 7 \times 10^1 + 3 \times 10^0$
- Base 2
 - Each bit represents a power of 2
 - $10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22$

Divide repeatedly by 2 and keep remainders

$$\begin{aligned}
 12/2 &= 6 & R &= 0 \\
 6/2 &= 3 & R &= 0 \\
 3/2 &= 1 & R &= 1 \\
 1/2 &= 0 & R &= 1 \\
 \text{Result} &= 1100
 \end{aligned}$$

Writing Bits is Tedious for People



- Octal (base 8)
 - Digits 0, 1, ..., 7
 - In C: 00, 01, ..., 07
- Hexadecimal (base 16)
 - Digits 0, 1, ..., 9, A, B, C, D, E, F
 - In C: 0x0, 0x1, ..., 0xf

0000 = 0	1000 = 8
0001 = 1	1001 = 9
0010 = 2	1010 = A
0011 = 3	1011 = B
0100 = 4	1100 = C
0101 = 5	1101 = D
0110 = 6	1110 = E
0111 = 7	1111 = F

Thus the 16-bit binary number

1011 0010 1010 1001

converted to hex is

B2A9

Interpretation: Colors



- Three primary colors
 - Red
 - Green
 - Blue
- Strength
 - 8-bit number for each color (e.g., two hex digits)
 - So, 24 bits to specify a color

• In HTML, on the course Web page

- Red: `<i>Symbol Table Assignment Due</i>`
- Blue: `<i>Fall Recess</i>`

• Same thing in digital cameras

- Each pixel is a mixture of red, green, and blue

Binary Representation of Integers



- Fixed number of bits in memory

- char: 8 bits
- short: usually 16 bits
- int: 16 or 32 bits
- long: 32 bits
- long long: 64 bits

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
...	...
1{n}	2 ⁿ -1

- Unsigned integers

- Always positive or 0
- All arithmetic is modulo 2ⁿ
- unsigned char
- unsigned short
- unsigned int
- unsigned long
- unsigned long long

Size and Overflow in Unsigned Integers



Bits	Integer Range
8	0 - 255
16	0 - 65,535
32	0 - 4,294,967,295
64	0 - 18,446,744,073,709,551,615

Binary	Decimal
0	0
1	1
10	2
...	...
1{n}	2 ⁿ -1

Number of bits determines unsigned integer range

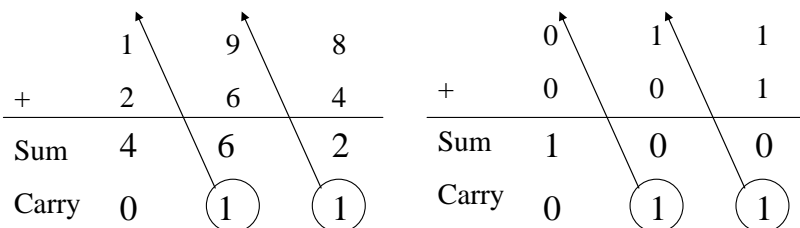
Overflow:

- 8-bit integer → 11111111₂ (255₁₀)
- Add 1
- What happens?

Adding Two Integers: Base 10



- From right to left, we add each pair of digits
- We write the sum, and add the carry to the next column



Binary Sums and Carries



a	b	Sum
0	0	0
0	1	1
1	0	1
1	1	0

a	b	Carry
0	0	0
0	1	0
1	0	0
1	1	1

XOR

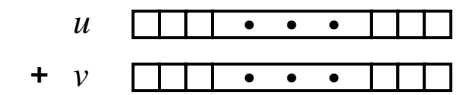
AND

$$\begin{array}{r}
 0100\ 0101 \leftarrow 69 \\
 + 0110\ 0111 \leftarrow 103 \\
 \hline
 1010\ 1100 \leftarrow 172
 \end{array}$$

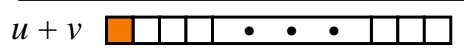
Overflow in Unsigned Addition



Operands: w bits



True Sum: $w + 1$ bits



Discard Carry: w bits

$UAdd_w(u, v)$



$$UAdd_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

Modulo Arithmetic: $UAdd_w(u, v) = u + v \bmod 2^w$

17

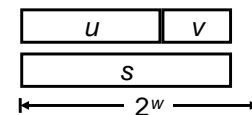
Detecting Unsigned Overflow



Task:

- Given $s = UAdd_w(u, v)$
- Determine if $s = u + v$

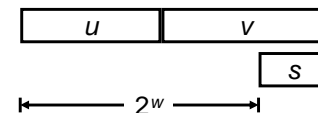
No Overflow:



Claim:

- Overflow iff $s < u$
- ovf = $(s < u)$
- By symmetry iff $s < v$

Overflow:



Proof:

- $0 \leq v < 2^w$
- No overflow $\Rightarrow s = u + v \geq u + 0 = u$
- Overflow $\Rightarrow s = u + v - 2^w < u + 0 = u$

18

Modulo Arithmetic



Consider only numbers in a range

- E.g., five-digit car odometer: 0, 1, ..., 99999
- E.g., eight-bit numbers 0, 1, ..., 255

Roll-over when you run out of space

- E.g., car odometer goes from 99999 to 0, 1, ...
- E.g., eight-bit number goes from 255 to 0, 1, ...

Adding 2^n doesn't change the answer

- For eight-bit number, $n=8$ and $2^n=256$
- E.g., $(37 + 256) \bmod 256$ is simply 37

This can help us do subtraction...

- Suppose you want to compute $a - b$
- Note that this equals $a + (256 - 1 - b) + 1$

19

Modulo Arithmetic



Modulo Addition Forms an Abelian Group

Closed under addition

- $0 \leq UAdd_w(u, v) \leq 2^w - 1$

Commutative

- $UAdd_w(u, v) = UAdd_w(v, u)$

Associative

- $UAdd_w(t, UAdd_w(u, v)) = UAdd_w(UAdd_w(t, u), v)$

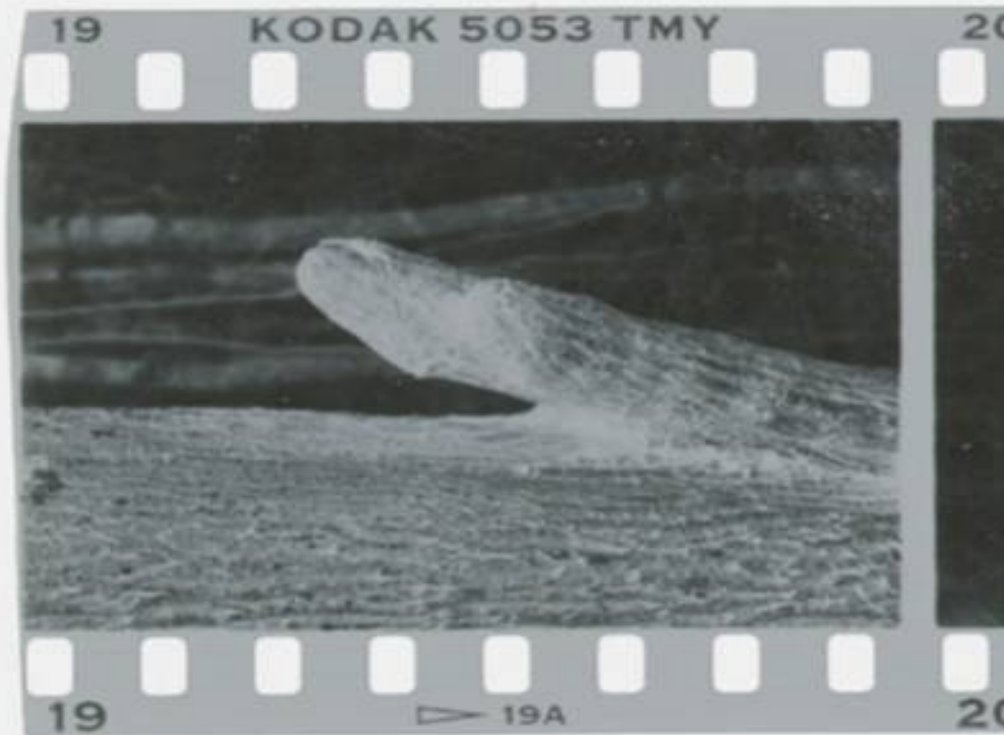
0 is additive identity

- $UAdd_w(u, 0) = u$

Every element has additive inverse

- Let $UComp_w(u) = 2^w - u$
- $UAdd_w(u, UComp_w(u)) = 0$

20



(negatives...)

What about Negative Numbers?



Bits	Patterns
8	256
16	65,536
32	4,294,967,296
64	18,446,744,073,709,551,616

Binary	Pattern
0	1
1	2
10	3
...	...
1{n}	2 ⁿ

- We have been looking at **unsigned numbers**
- What about negative or **signed numbers**?
- Need new interpretation of bits
- Some patterns interpreted as negative numbers

Key Standard Pattern Assignments



Bit Pattern	Sign Magnitude	One's Complement	Two's Complement
000	+0	+0	0
001	+1	+1	+1
010	+2	+2	+2
011	+3	+3	+3
100	-0	-3	-4
101	-1	-2	-3
110	-2	-1	-2
111	-3	-0	-1

- Which one is best?
 - Balance
 - Zeros
 - Ease of operations

Most Common: Two's Complement



Bit Pattern	Two's Complement
000	0
001	+1
010	+2
011	+3
100	-4
101	-3
110	-2
111	-1

- "Invert and Add 1" to negate
- Sign Bit
- Zeros, Range
- What about arithmetic?

Unsigned and Two's Complement



Unsigned Values

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

• UMin = 0

• UMax = $2^w - 1$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

• TMin = -2^{w-1}

• TMax = $2^{w-1} - 1$

Values for W = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000



Sizes and C Data Types

C Data Type	MIPS, x86	Alpha
char	8 bits	8 bits
short	16 bits	16 bits
int	32 bits	32 bits
long int	32 bits	64 bits

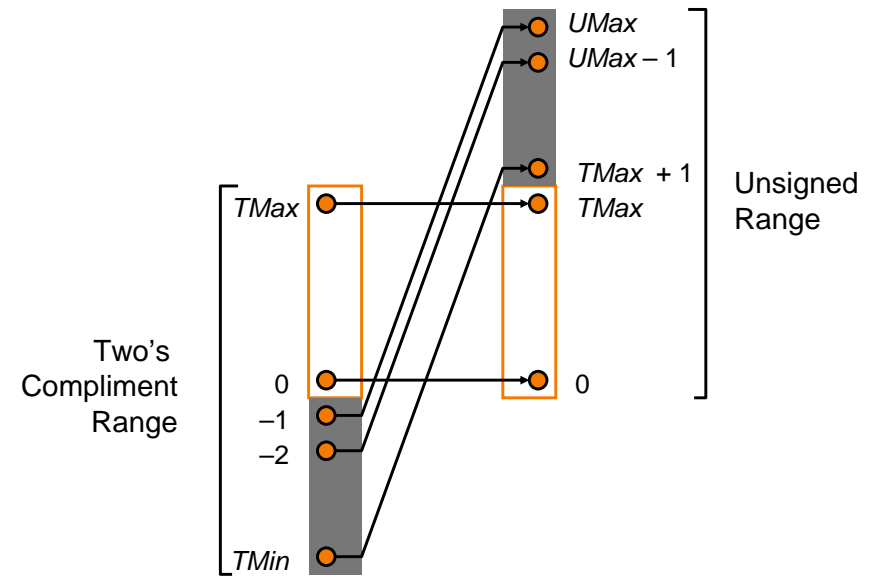
char, short, int, long int

- Refer to number of bits of integer
- Most machines: signed two's complement

unsigned <type>

- Same number of bits as signed counterparts
- Unsigned integer

Representation Relationship



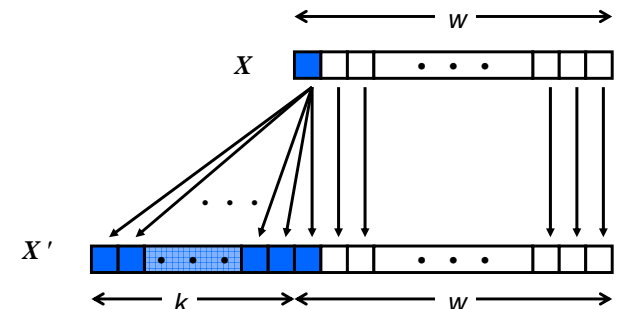
Sign Extension



```
char minusFour = -4;
short moreBits;
moreBits = (short) minusFour;
```

Given w bit signed integer, return equivalent w+k bit signed integer

Sign Extend:

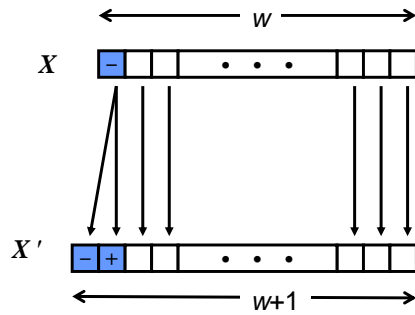


Sign Extension

Proof of Correctness Outline



- Prove Correctness by Induction on k
- Induction Step: extending by single bit maintains value

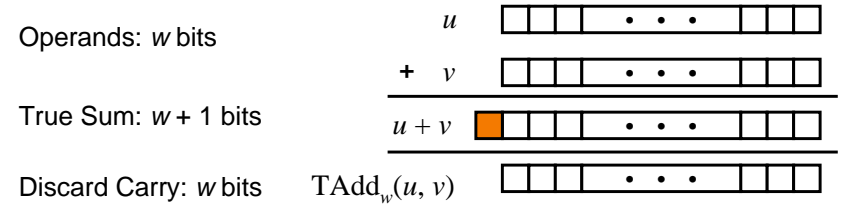


29

Two's Complement Addition



- TAdd and UAdd have identical Bit-Level Behavior!

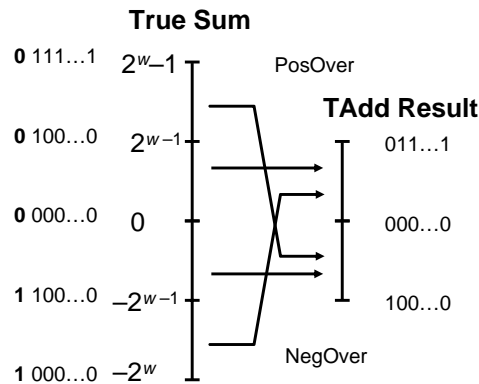
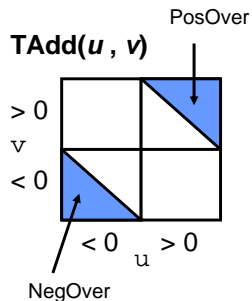


30

Characterizing TAdd



- True sum requires $w+1$ bits
- Drop MSB



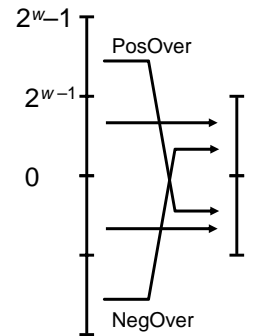
$$TAdd_w(u, v) = \begin{cases} u + v + 2^{w-1} & u + v < TMin_w \text{ (NegOver)} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^{w-1} & TMax_w < u + v \text{ (PosOver)} \end{cases}$$

31

Detecting Two's Complement Overflow



- Task:
 - Given $s = TAdd_w(u, v)$
 - Determine if $s = Add_w(u, v)$
- Claim:
 - Overflow iff either:
 - $u, v < 0, s \geq 0$ (NegOver)
 - $u, v \geq 0, s < 0$ (PosOver)
 - $ovf = (u < 0 == v < 0) \ \&\& \ (u < 0 != s < 0)$;



- Proof:
 - Obviously, if $u \geq 0$ and $v < 0$, then $TMin_w \leq u + v \leq TMax_w$
 - Symmetrically if $u < 0$ and $v \geq 0$
 - Other cases from analysis of TAdd

32

Negation vs. Inversion



Inversion:

- A bit-wise operation
- Flip all 0's to 1's and vice versa: 0011 => 1100
- What does this do to the two's complement value?

Negation:

- Two's complement: invert all bits and add 1
- Example:

$$3_{10} = 0011$$

$$\text{invert}(0011) + 1 \rightarrow 1100 + 1 \rightarrow 1101$$

$$1101 = -3_{10}$$

Two's Complement Negation



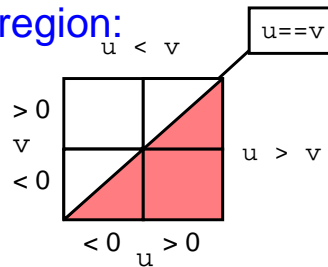
- Mostly like Integer Negation
 - $\text{TComp}(u) = -u$
- TMin is Special Case
 - $\text{TComp}(\text{TMin}) = \text{TMin}$
 - Note Also: $\text{TComp}(0) = 0$
- Negation in C ($x = -x;$) is Actually TComp

34

Comparing Two's Complements



- Given signed numbers u, v
- Determine whether or not $u > v$
- Return true for shaded region:



- **Bad Approach:**
 - Test $(u - v) > 0$
 - Problem: Thrown off by Overflow

35

Representation: A Collection of Bits



- Treat unsigned int as a collection 32 independent bits
- Good for tracking 32 individual binary conditions
 - True/False
 - Yes/No
 - Black/White
- Can also treat unsigned in as:
 - 16 2-bit values
 - 8 4-bit values
 - 4 8-bit values
 - 8 1-bit value, 4 2-bit values, 2 4-bit values, and 1 8-bit value

36

Bitwise Operators: AND and OR



• Bitwise AND (&)

&	0	1
0	0	0
1	0	1

• Bitwise OR (|)

	0	1
0	0	1
1	1	1

- Mod on the cheap!
 - E.g., $h = 53 \& 15$;

53

0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---

& 15

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

5

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

37

Bitwise Operators: Not and XOR



• One's complement (~)

- Turns 0 to 1, and 1 to 0
- E.g., set last three bits to 0
 - $x = x \& \sim 7$;

• XOR (^)

- 0 if both bits are the same
- 1 if the two bits are different

^	0	1
0	0	1
1	1	0

38

Bitwise Operators: Shift Left/Right



• Shift left (<<): Multiply by powers of 2

- Shift some # of bits to the left, filling the blanks with 0

53

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

53<<2

1	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

• Shift right (>>): Divide by powers of 2

- Shift some # of bits to the right
 - For unsigned integer, fill in blanks with 0
 - What about signed integers? Varies across machines...
 - Can vary from one machine to another!

53

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

53>>2

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

39

Count Number of 1s in an Integer



• Function bitcount(unsigned x)

- Input: unsigned integer
- Output: number of bits set to 1 in the binary representation of x

• Main idea

- Isolate the last bit and see if it is equal to 1
- Shift to the right by one bit, and repeat

```
int bitcount(unsigned int x) {
    int b;
    for (b = 0; x != 0; x >>= 1)
        if (x & 1)
            b++;
    return b;
}
```

40

XOR Encryption



- Program to encrypt text with a key
 - Input: original text in stdin
 - Output: encrypted text in stdout
- Use the same program to decrypt text with a key
 - Input: encrypted text in stdin
 - Output: original text in stdout
- Basic idea
 - Start with a key, some 8-bit number (e.g., 0110 0111)
 - Do an operation that can be inverted
 - E.g., XOR each character with the 8-bit number

0100 0101	0010 0010
^ 0110 0111	^ 0110 0111
-----	-----
0010 0010	0100 0101

41

XOR Encryption, Continued



- But, we have a problem
 - Some characters are control characters
 - These characters don't print
- So, let's play it safe
 - If the encrypted character would be a control character
 - ... just print the original, unencrypted character
 - Note: the same thing will happen when decrypting, so we're okay
- C function `isctr1()`
 - Returns true if the character is a control character

42

XOR Encryption, C Code



```
#define KEY '&'
int main(void) {
    int orig_char, new_char;

    while ((orig_char = getchar()) != EOF) {
        new_char = orig_char ^ KEY;
        if (isctr1(new_char))
            putchar(orig_char);
        else
            putchar(new_char);
    }
    return 0;
}
```

43

Stupid Programmer Tricks



- Where do I use bitwise & most?
 - Bit vectors
- What's a bit vector?
 - Lots of booleans packed into an int/long
 - Often used to indicate some condition(s)
 - Less storage space than lots of fields
 - More explicit storage than compiled-defined bit fields
- Your compiler can do this?

```
typedef struct Blah {
    int b_onoff:1;
    int b_temperature:7;
    char b_someChar;
}
```

44

Example From Real Code



```
• #define DONTCACHE_REQNOSTORE      0x000001
• #define DONTCACHE_AUTHORIZED      0x000002
• #define DONTCACHE_MISSINGVARIANTHDR 0x000004
• #define DONTCACHE_USERORPASS      0x000008
• #define DONTCACHE_BYPASSFILTER     0x000010
• #define DONTCACHE_NONCACHMETHOD   0x000020
• #define DONTCACHE_CTLPRIVATE      0x000040
• #define DONTCACHE_CTLNOSTORE      0x000080
• #define DONTCACHE_ISQUERY         0x000100
• #define DONTCACHE_EARLYEXPIRE     0x000200
• #define DONTCACHE_NOLASTMOD       0x000400
• #define DONTCACHE_NONEGCACHING    0x000800
• #define DONTCACHE_INSTANTEXPIRE   0x001000
• #define DONTCACHE_FILETOOBIG      0x002000
• #define DONTCACHE_FILEGREWTOOBIG  0x004000
• #define DONTCACHE_ICPPROXYONLY    0x008000
• #define DONTCACHE_LARGEFILEBLAST  0x010000
• #define DONTCACHE_PERSISTLOGLOADING 0x020000
• #define DONTCACHE_NEWERCOPYEXISTS  0x040000
• #define DONTCACHE_BADVARYFIELDS    0x080000
• #define DONTCACHE_SETCOOKIE       0x100000
• #define DONTCACHE_HTTPSTATUSCODE  0x200000
• #define DONTCACHE_OBJECTINCOMPLETE 0x400000
```

45

Conclusions



- Computer represents everything in binary
 - Integers, floating-point numbers, characters, addresses, ...
 - Pixels, sounds, colors, etc.
- Binary arithmetic through logic operations
 - Sum (XOR) and Carry (AND)
 - Two's complement for subtraction
- Binary operations in C
 - AND, OR, NOT, and XOR
 - Shift left and shift right
 - Useful for efficient and concise code, though sometimes cryptic

46