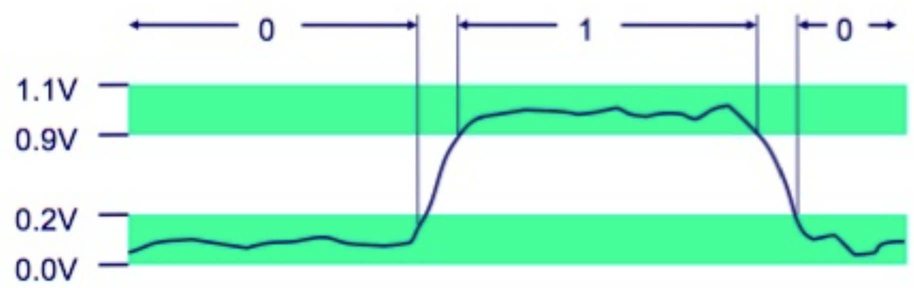
Bits & Integers

Everything is Bits

- each bit is 0 or 1
- by encoding/interpreting sets of bits in various ways
  - computers determine what to do (instructions)
  - and represent an manipulate numbers, sets, strings, etc.
- Why bits? Electronic implementation
  - easy to store with bistable elements
  - reliably transmitted on noisy and inaccurate wires
  - 

- 64-bit machines can address far more memory slots than can 32-bit machines

Encoding Byte Values

- byte = 8 bits
- **binary**: 0000 0000 to 1111 1111
- **decimal**: 0 to 255
- **hex**: 00 to FF

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Boolean Algebra

And

A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

~A = 1 when A=0

~	
0	1
1	0

Xor (exclusive or)

A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

Bit-Level Operations in C

- operations &, |, ~, ^ available in C

Examples (char type)

- ~0x41 ➤ 0xBE
  - `~01000001 ➤ 10111110`
- ~0x00 ➤ 0xFF
  - `~00000000 ➤ 11111111`
- 0x69 & 0x55 ➤ 0x41
  - `01101001 & 01010101 ➤ 01000001`
- 0x69 | 0x55 ➤ 0x7D
  - `01101001 | 01010101 ➤ 01111101`

Logic Operations in C (contrast)

- operations &&, ||, !
  - view 0 as false
  - anything else is true
  - **early termination** (short circuiting)
- examples (char data type)
  - `!0x41 ➤ 0x00`
  - `!0x00 ➤ 0x01`
  - `!!0x41 ➤ 0x01`
  - `0x69 && 0x55 ➤ 0x01`
  - `0x69 || 0x55 ➤ 0x01`

Shift Operations

Left Shift: x << y

- shift bit-vector x left y positions
- throw away extra bits on left
- fill with 0’s on left

Right Shift: x >> y

- shift bit-vector x right y positions
    - throw away extra bits on right
    - **l0gical shift**: fill with 0’s on left
    - **arithmetic shift**: replicate most significant bit on left

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Encoding Integers

Unsigned int

$$\sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

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

Two’s Complement Conversion

- for binary representation, flip all bits, then add 1
- `00111011 01101101` original
- `11000100 10010010` flipped
- `11000100 10010011` added 1
- flipped 15213 to -15213

Sign Bit

- for 2’s complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Encoding Example (+/- 15213)

`00111011 01101101` positive 15213
`11000100 10010011` negative 15213

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Numeric Ranges

Unsigned Values

UMin = 0
UMax = $2^w - 1$

Two's Complement Values

TMin = -2^{w-1}
TMax = $2^{w-1} - 1$

Other Values

Minus 1: 111...1

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Observations

- | TMin | = TMax + 1
  - asymmetric range
- UMax = 2 * TMax + 1

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

Equivalence

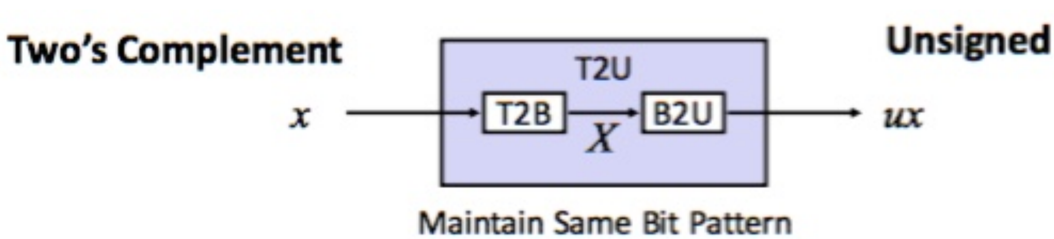
- same encodings for nonnegative values

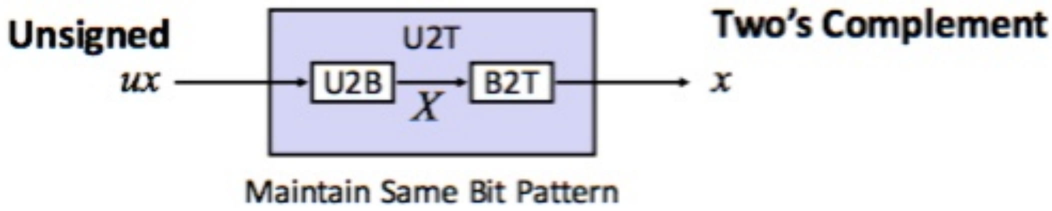
Uniqueness

- every bit pattern represents unique integer value
- each representable integer has unique bit encoding

Can Invert Mappings

- U2B(x) = $B2U^{-1}(x)$
  - bit pattern for unsigned integer
- T2B(x) = $B2T^{-1}(x)$
  - bit pattern for two's complement integer





Key

- **T**: Two’s Complement
- **B**: Bits
- **U**: Unsigned
- **2**: to

Mapping Signed ➤ Unsigned (and back)

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4	=	4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

- **Small negative becomes large positive**
- **Large negative becomes medium positive**

Signed vs. Unsigned in C

Constants

- by default, signed integers
- unsigned with “U” suffix

    - 0U, 4294U

Casting

- explicit casting between signed & unsigned uses U2T and T2U
    - `int tx, ty;`
    - `unsigned ux, uy;`
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
- implicit casting also occurs via assignments and procedure calls
    - `tx = ux;`
    - `uy = ty;`

Casting Surprises

Expression Evaluation

- if there is a mix of unsigned and signed in a single expression
    - **SIGNED VALUES IMPLICITLY CAST TO UNSIGNED**
    - **UNSIGNED HAS PRECEDENCE**

Examples for W = 32: TMin = -2,147,483,648, TMax = 2,147,483,647

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Byte Ordering

Conventions

Little Endian (basically backwards)

- least significant byte (right-most) has lowest (smallest) address

Big Endian

- least significant byte (right-most) has highest (largest) address

Example

- variable x has 4-byte value of **0x01234567** at address (&x) of **0x100**