Data Representation

CSCI 2021: Machine Architecture and Organization

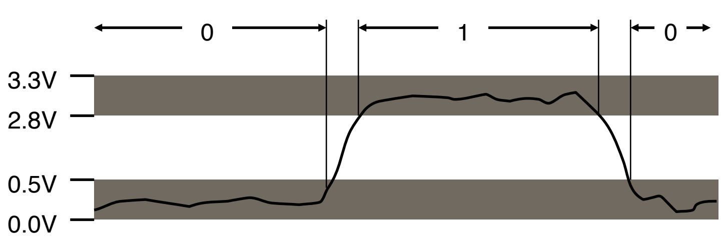
Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

University of Minnesota

A Two-State Approach

0 1



With Slides from Bryant and O'Hallaron

Counting

0 1 2 3 4

5 6 7 8 9

10 separate states!!

Why Don't Computers Use Base 10?

- Base 10 Number Representation
  - That's why fingers are known as "digits"
  - Natural representation for financial transactions
  - Even carries through in scientific notation
    - 1.5213×10^4
- Implementing Electronically
  - Hard to store
    - ENIAC (First electronic computer) used 10 vacuum tubes / digit
  - Hard to transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.

Boolean Algebra

Boolean Algebra

- Developed by George Boole in 19th Century
  Algebraic representation of logic: encode "True" as 1 and "False" as 0

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

&	0	1
0	0	0
1	0	1

Associativity and commutativity

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

|	0	1
0	0	1
1	1	1

Associativity and commutativity

Not: ~A = 1 when A=0

~	
0	1
1	0

The XOR Operation

Exclusive-Or (Xor):

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

^	0	1
0	0	1
1	1	0

What about:

A ^ A?

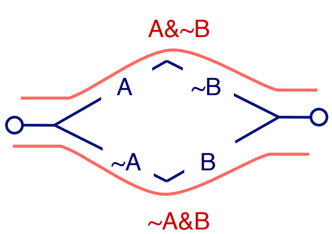
A ^ 0?

A ^ 1?

Associativity and commutativity

Application of Boolean Algebra

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master's Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0



Connection when

A&~B | ~A&B

= A^B

Relations Between Operations

- DeMorgan's Laws
  - Express & in terms of |, and vice-versa
    - A & B = ~(~A | ~B)
      - A and B are true if and only if neither A nor B is false
    - A | B = ~(~A & ~B)
      - A or B are true if and only if A and B are not both false
- Exclusive-Or
    - A ^ B = (~A & B) | (A & ~B)
      - Exactly one of A and B is true
    - A ^ B = (A | B) & ~(A & B)
      - Either A is true, or B is true, but not both

Bit-Level Operations in C

- Operations &, |, ~, ^ Available in C
  - Apply to any "integral" data type
    - `long, int, short, char`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - $\sim 01000001_2$ --> 10111110_2
  - $\sim 00000000_2$ --> 11111111_2
  - 01101001_2 & 01010101_2 --> 01000001_2
  - 01101001_2 | 01010101_2 --> 01111101_2

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - `&&`, `||`, `!`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- Examples (char data type)
  - ! 01000001_2 --> 00000000_2
  - ! 00000000_2 --> 00000001_2
  - !! 01000001_2 --> 00000001_2
  - 01010001_2 && 10101110_2 --> 00000001_2
  - 01010001_2 || 10101110_2 --> 00000001_2

Boolean Algebras with Bit Vector

- Operate on Bit Vectors
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- All of the Properties of Boolean Algebra Apply

Cool Stuff with Xor

(A ^ B) ^ B = A

(A^B)^B	A = 0	A = 1
B = 0	0	1
B = 1	0	1

```
void swap()
{
   int x = 00100111, y = 11100111;
   x = x ^ y;     /* #1 */
   y = x ^ y;     /* #2 */
   x = x ^ y;     /* #3 */
}
```

	*x	*y
Begin	A	B
1	A^B	B
2	A^B	(A^B)^B = A
3	(A^B)^A = B	A
End	B	A

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector `x` left `y` positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector `x` right `y` positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on right
    - Useful with two's complement integer representation

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Binary Numbers

Converting Between Decimal and Binary

Binary to Decimal

1001_2 ?

= $1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$

= 8 + 1

= 9

0000_2 ?

= 0

1111_2 ?

= 15

Converting Between Decimal and Binary

Decimal to Binary

Dividing the number repeatedly by 2 until the number becomes 0

49 ?

Divide by	Number	Remainder
2	49	
2		
2		
2		
2		
2		
2		

What about real numbers?

Binary to Decimal

101.11_2 ?

$= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2}$

$= 4 + 1 + 0.5 + 0.25$

$= 5.75$

0.1_2 ?

0.0001_2 ?

Converting Between Decimal and Binary

Decimal to Binary

multiply the number by 2 and register the integer portion

0.3125 ?

Multiply by	Number	Remainder
2	0.3125	
2		
2		
2		
2		

Converting Between Decimal and Binary

What about 0.4 ?

Multiply by	Number	Remainder
2	0.4	
2		
2		
2		
2		
2		
2		
2		
		...

$0.4_{10} = 0.[0110]_2$

Non-terminating repeating

Practice

Decimal	Binary
2	
6	
65	
63	
1025	
0.25	
43.16	
0.20	

Hex and Octal Number

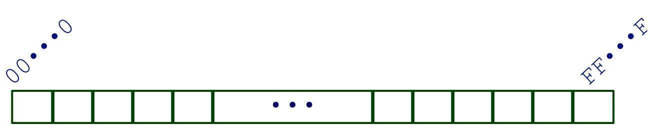
Binary numbers are long!
Binary-decimal conversion is non-trivial!

Binary	Decimal	Octal	Hexadecimal

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

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



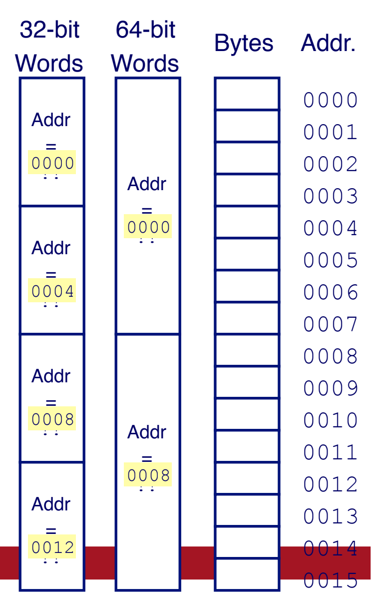
Byte-Oriented Memory Organization

Machine Words

- Machine Has "Word Size"
  - Nominal size of integer-valued data
    - Including addresses
  - Most current machines use 32 bits (4 bytes) words
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - High-end systems use 64 bits (8 bytes) words
    - Potential address space ≈ 1.8×10^{19} bytes
    - x86-64 machines support 48-bit addresses: 256 Terabytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



With Slides from Bryant and O'Hallaron

Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Variations of Data Format

Machines support multiple data formats

- Fractions or multiples of word size,
- but always integral number of bytes

```
int main ()   {
    printf("chars are %d bytes long\n", sizeof(char));
    printf("ints are %d bytes long\n", sizeof(int));
    printf("shorts are %d bytes long\n", sizeof(short));
    printf("floats are %d bytes long\n", sizeof(float));
}
```

chars are 1 bytes long
ints are 4 bytes long
shorts are 2 bytes long
floats are 4 bytes long

Byte Ordering Example

- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address
- Example
  - Variable `x` has 4-byte representation `0xDEADBEEF`
  - Address given by `&x` is `0x100`

Big Endian

		0x100	0x101	0x102	0x103		
		DE	AD	BE	EF		

Little Endian

		0x100	0x101	0x102	0x103		
		EF	BE	AD	DE		

Examining Data Representations

Code to Print Byte Representation of Data

- Casting pointer to `unsigned char *` creates byte array

```
int integer = 0xDEADBEEF;
int main()
{
  int i;
  unsigned char *pointer = &integer;
  for (i = 0; i < 4; i++)
    printf("Byte #%d: addr %p, value 0x%x\n", i, pointer+i, *(pointer+i));
}
```

Printf directives:
- **`%p`: Print pointer**
- **`%x`: Print Hexadecimal**

With Slides from Bryant and O'Hallaron

The Outcome is Different on Different Machine

On SUN work station:

Byte #0: addr 20944, value 0xde
Byte #1: addr 20945, value 0xad
Byte #2: addr 20946, value 0xbe
Byte #3: addr 20947, value 0xef

A big endian machine

On PC (linux):

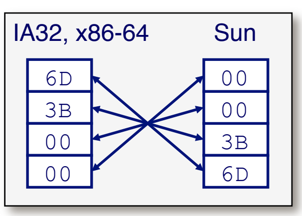
Byte #0: addr 0x80495f8, value 0xef
Byte #1: addr 0x80495f9, value 0xbe
Byte #2: addr 0x80495fa, value 0xad
Byte #3: addr 0x80495fb, value 0xde

A little endian machine

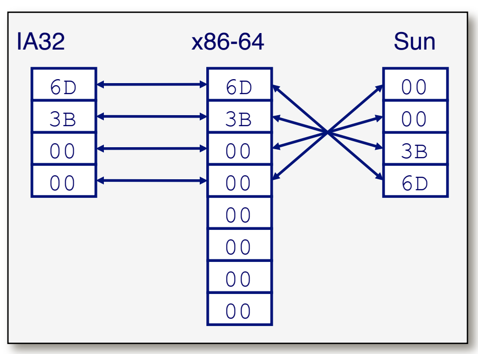
Representing Integers

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

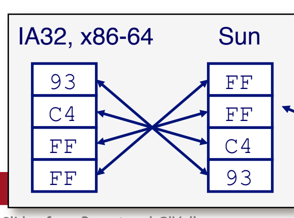
`int A = 15213;`



`long int C = 15213;`



`int B = -15213;`



Two's complement representation
(Covered later)

Representing Pointers

```
int B = -15213;
int *P = &B;
```

Sun	IA32	x86-64
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

Different compilers & machines assign different locations to objects

Representing Strings

```
char S[6] = "18243";
```

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character "0" has code 0x30
      - Digit i has code 0x30+i
  - String should be null-terminated
    - Final character = 0
- Compatibility
  - Byte ordering not an issue

Linux/Alpha	Sun
31	31
38	38
32	32
34	34
33	33
00	00

2's Complement Representation

Binary	Decimal	Octal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F

We need Negative Numbers!!!

Addition and Subtraction in Binary

Decimal: 10 + 2 = 12

Hexadecimal: A + 2 = C

Binary: 1010 + 0010 =1100

Decimal: 12 - 2 = 10

Hexadecimal: C - 2 = A

Binary: 1100 - 0010 =1010

Addition/Subtraction in Binary with Negative Number

Decimal: 0 - 1 = -1

Hexadecimal: 0 - 1 = -1

Binary: 0000 + 0001 =1111

Decimal: -1 + 1 = 0

Hexadecimal: -1 + 1 = 0

Binary: 1111 + 0001 =0000

Decimal: -1 - 1 = -2

Hexadecimal: -1 - 1 = -2

Binary: 1111 - 0001 =1110

Sign Bit

Using half of the numbers for negative

- The most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

Equivalence

- Same encodings for nonnegative values

Uniqueness

- Every bit pattern represents unique integer
- Each representable integer has a unique encoding

How to find the negative number?

1. Find the positive number
2. Inverse all bits
3. Add "1" to it

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

Encoding Integers

Unsigned

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

Two's Complement

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

Sign Bit

```
short int x =  15213;
short int y = -15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

Encoding Example (Cont.)

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 Range

Unsigned Values

- ***UMin*** = **0**

  000…0

- ***UMax*** = $2^w - 1$

  111…1

Two's Complement Values

- ***TMin*** = -2^{w-1}

  100…0

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

  011…1

Other Values

- **Minus 1**

  111…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

Unsigned & Signed Numeric Values

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 comp integer

Values for Different Word Sizes

	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

On A Real Machine

```
#include <stdio.h>
#include <limits.h>
main ()  {
   int uint_max = UINT_MAX;
   int int_max = INT_MAX;
   int int_min = INT_MIN;
   printf("uint_max = %u; int_max = %d; int_min = %d\n",
              uint_max, int_max, int_min);
}
```

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

uint_max = 4294967295; int_max = 2147483647; int_min = -2147483648

On A Real Machine

```
#include <stdio.h>
#include <limits.h>
main ()  {
   int ushrt_max = USHRT_MAX;
   int shrt_max = SHRT_MAX;
   int shrt_min = SHRT_MIN;
   printf("ushrt_max = %u; shrt_max = %d; shrt_min = %d\n",
              ushrt_max, shrt_max, shrt_min);
}
```

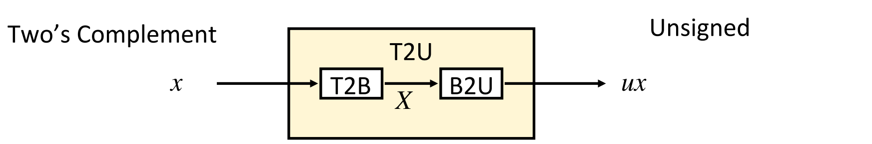
ushrt_max = 65535; shrt_max = 32767; shrt_min = -32768

Casting Signed to Unsigned

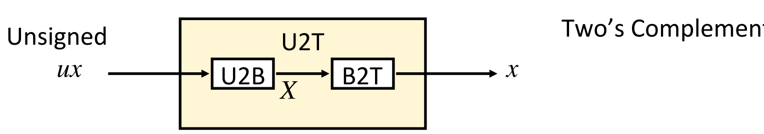
- C Allows signed and unsigned value, and the conversions from signed to unsigned, and vise versa

```
short int              x =  55455;
unsigned short int ux = (unsigned short) x;
short int              y  = -55455;
unsigned short int uy = (unsigned short) y;
```


Mapping Between Signed & Unsigned



Maintain Same Bit Pattern



Maintain Same Bit Pattern

- Mappings between unsigned and two's complement numbers: keep bit representations and reinterpret

Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix

```
0U, 4294967259U
```

- Casting
  - Explicit casting between signed & unsigned same as 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;
```


Conversion in C

```
#include <stdio.h>
main () {
  int          x =  55455;
  unsigned int ux = (unsigned int) x;
  int          y  = -55455;
  unsigned int uy = (unsigned int) y;
  printf("int 55455 = %d; int -55455 = %d\n", x, y);
  printf("unsigned 55455 = %u; unsigned -55455 = %u\n", ux, uy);
}
```

int 55455 = 55455; int -55455 = -55455
unsigned 55455 = 55455; unsigned -55455 = 4294911841

Mapping Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5	T2U →	5
0110	6	← U2T	6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

With Slides from Bryant and O'Hallaron

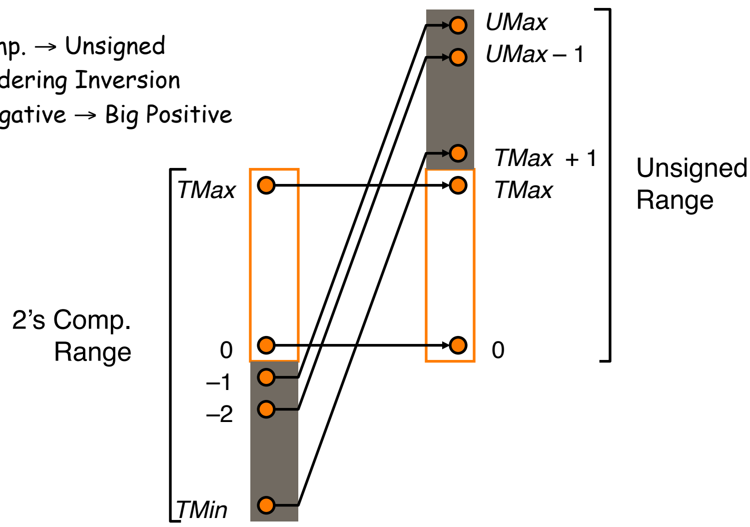
Mapping Signed ↔ Unsigned

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		11
1100	-4	↔ +/- 16	12
1101	-3		13
1110	-2		14
1111	-1		15

With Slides from Bryant and O'Hallaron

Explanation of Casting Surprises

- 2's Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



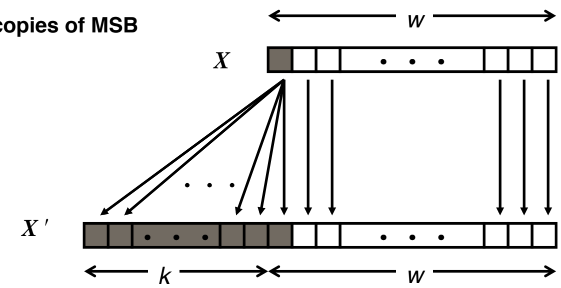
Conversion in C Answer

- Expression Evaluation
  - If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
  - Including comparison operations <, >, ==, <=, >=
  - Examples for $W = 32$
- Constant$_1$ Constant$_2$

Constant$_1$	Constant$_2$
`0`	`0U`
`-1`	`0`
`-1`	`0U`
`2147483647`	`-2147483648`
`2147483647U`	`-2147483648`
`-1`	`-2`
`(unsigned) -1`	`-2`
`2147483647`	`2147483648U`
`2147483647`	`(int) 2147483648U`

Sign Extension

- Task:
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- Rule:
  - Make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension Example

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Why Should I Use Unsigned?

- *Don't* Use Just Because Number Nonzero
  - C compilers on some machines generate less efficient code
  - Easy to make mistakes

```
unsigned int i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

- *Do* Use When Performing Modular Arithmetic
  - Multiprecision arithmetic
  - Other esoteric stuff
- *Do* Use When Need Extra Bit's Worth of Range
  - Working right up to limit of word size

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD's implementation of getpeername
- There are legions of smart people trying to find vulnerabilities in programs

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

With Slides from Bryant and O'Hallaron

Malicious Usage

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

With Slides from Bryant and O'Hallaron

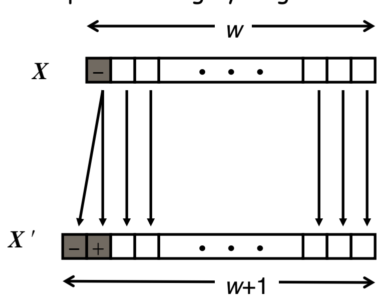
Summary
Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

- Expression containing signed and unsigned int
  - `int` is cast to `unsigned`!!

Justification For Sign Extension

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



  - Key observation: $-2^{w-1} = -2^w + 2^{w-1}$
  - Look at weight of upper bits:

$X \quad -2^{w-1}\, x_{w-1}$

$X' \quad -2^w\, x_{w-1} + 2^{w-1}\, x_{w-1} \quad = \quad -2^{w-1}\, x_{w-1}$

Summary: Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

Integer Arithmatics

Negating with Complement & Increment

- Claim: Following Holds for 2's Complement
  `~x + 1 == -x`
- Complement
  - Observation: `~x + x == 1111…11`$_2$ `== -1`

x	1 0 0 1 1 1 0 1
+ ~x	0 1 1 0 0 0 1 0
-1	1 1 1 1 1 1 1 1

- Increment
  `~x + x + (-x + 1)   ==   -1 + (-x + 1)`
  `~x + 1              ==   -x`
- Warning: Be cautious treating `int`'s as integers

Comp. & Incr. Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 1001001**1**
y	-15213	C4 93	11000100 10010011

0

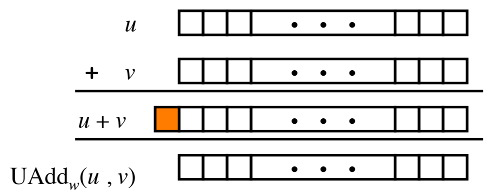
	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

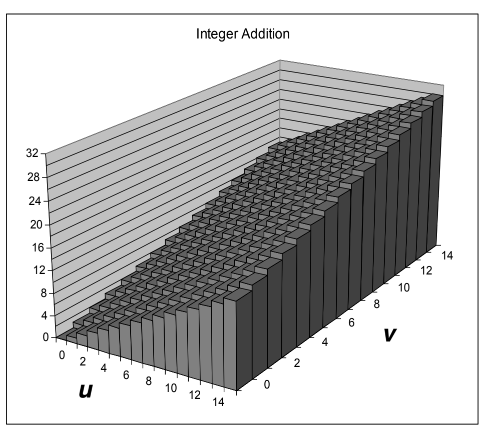
  s = UAdd$_w$(u , v) = u + v mod 2^w

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

Integer Addition

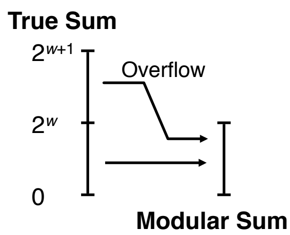
- Integer Addition
  - 4-bit integers *u, v*
  - Compute true sum Add$_4$(u , v)
  - Values increase linearly with *u* and *v*
  - Forms planar surface

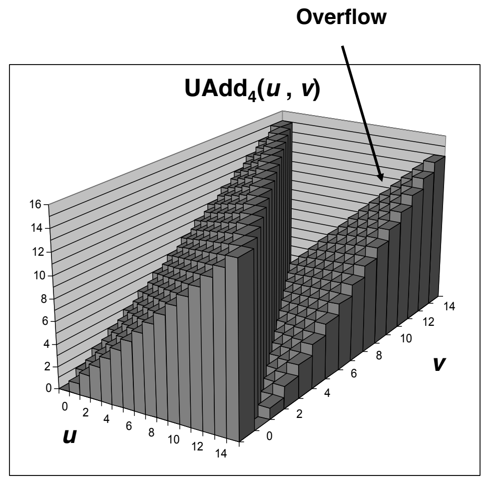
Add$_4$(u , v)



Unsigned Addition

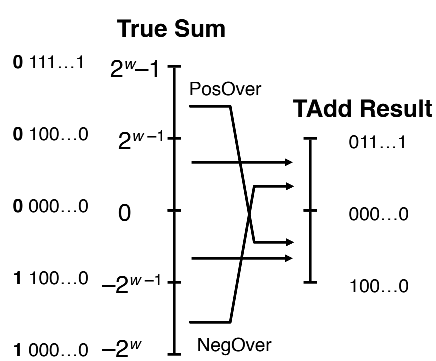
- Wraps Around
  - If true sum ≥ 2^w
  - At most once

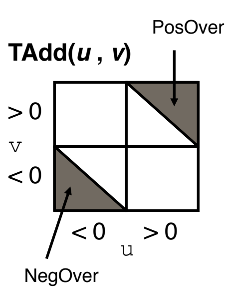




2's Complement Addition

- Functionality
  - True sum requires ***w* +1** bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer

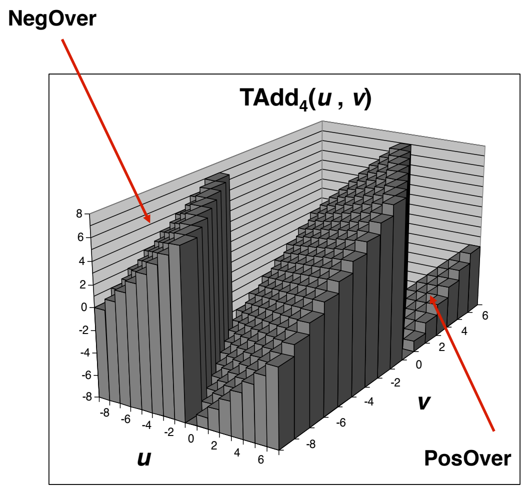




$$TAdd_w(u,v) = \begin{cases} u+v+2^{w-1} & u+v < TMin_w \quad \textbf{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^{w-1} & TMax_w < u+v \quad \textbf{(PosOver)} \end{cases}$$

2's Complement Addition

- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once



Detecting 2's Complement Overflow

- Task
  - Given s = $\mathrm{TAdd}_w(u , v)$
  - Determine if s = $\mathrm{Add}_w(u , v)$
  - Example

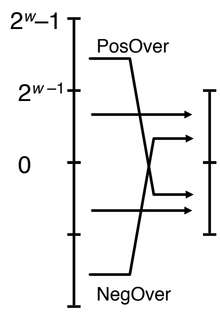
```
int s, u, v;
s = u + v;
```

- 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);
```



Mathematical Properties of TAdd

- Isomorphic Group to unsigneds with UAdd
  - $TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v)))$
    - Since both have identical bit patterns
- Two's Complement Under TAdd Forms a Group
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

$$TComp_w(u) \quad = \quad \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

With Slides from Bryant and O'Hallaron

C Puzzle Answer

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- Assume machine with 32 bit word size, two's comp. integers
- *TMin* makes a good counterexample in many cases

```
❑ x < 0                 ⇒   ((x*2) < 0)
❑ ux >= 0
❑ x & 7 == 7            ⇒   (x<<30) < 0
❑ ux > -1
❑ x > y                 ⇒   -x < -y
❑ x * x >= 0
❑ x > 0 && y > 0        ⇒   x + y > 0
❑ x >= 0                ⇒   -x <= 0
❑ x <= 0                ⇒   -x >= 0
```

Multiplication

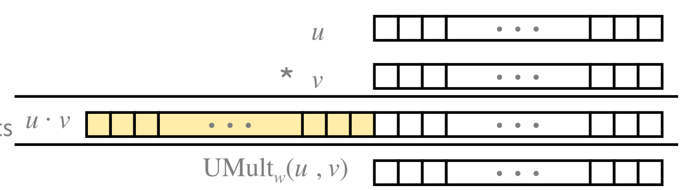
- Computing Exact Product of w-bit numbers x, y
  - Either signed or unsigned
- Ranges
  - Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two's complement min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only for $(TMin_w)^2$
- Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C

Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic

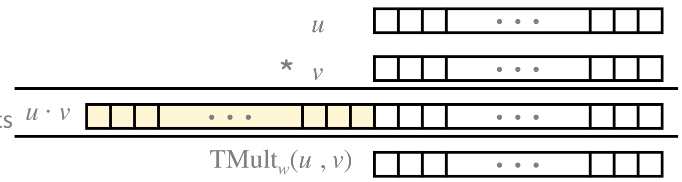
  $\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- Standard Multiplication Function
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

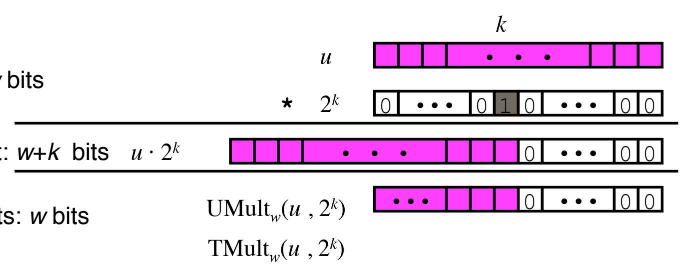
Power-of-2 Multiply with Shift

- Operation
  - `u << k` gives `u * ` ***2^k***
  - Both signed and unsigned

Operands: *w* bits

True Product: *w+k* bits

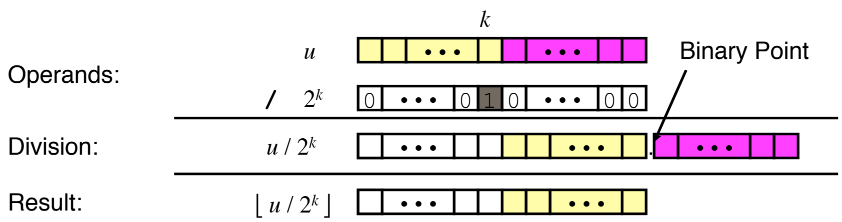
Discard *k* bits: *w* bits



- Examples
  - `u << 3            ==    u * 8`
  - `u << 5 - u << 3   ==    u * 24`
  - Most machines shift and add much faster than multiply
    - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

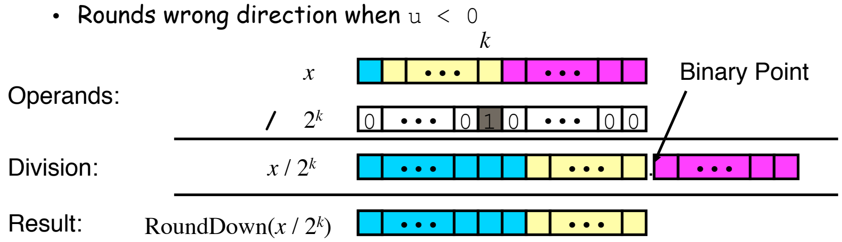
- Quotient of Unsigned by Power of 2
  - `u >> k` gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift



	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Signed Power-of-2 Divide with Shift

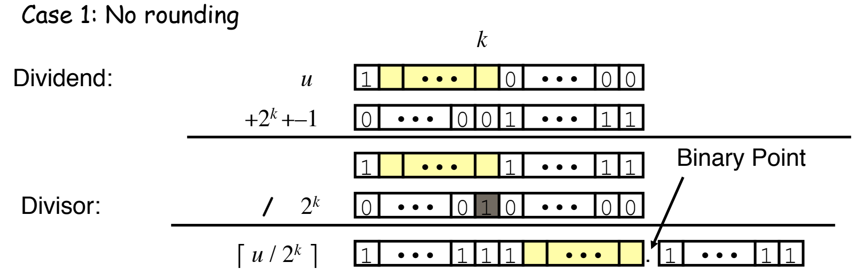
- Quotient of Signed by Power of 2
  - `x >> k` gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `u < 0`



	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

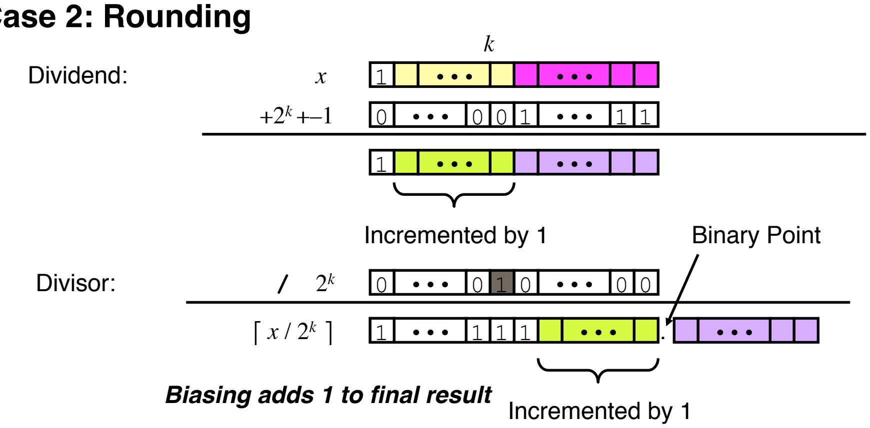
- Quotient of Negative Number by Power of 2
  - Want $\lceil$ `x / 2`^{k} $\rceil$ (Round Toward 0)
  - Compute as $\lfloor$ `(x+2`^{k}`-1)/ 2`^{k} $\rfloor$
    - In C: `(x + (1<<k)-1) >> k`
    - Biases dividend toward 0
- Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Arithmetic: Basic Rules

- Addition:
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2w

- Multiplication:
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

Arithmetic: Basic Rules

- Unsigned ints, 2's complement ints are isomorphic rings: isomorphism = casting

- Left shift
  - Unsigned/signed: multiplication by 2^k
  - Always logical shift

- Right shift
  - Unsigned: logical shift, div (division + round to zero) by 2^k
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by 2^k
    - Negative numbers: div (division + round away from zero) by 2^k
      Use biasing to fix

Properties of Unsigned Arithmetic

- Unsigned Multiplication with Addition Forms Commutative Ring
  - Addition is commutative group
  - Closed under multiplication
    $0 \leq UMult_w(u\,,\,v) \leq 2^w - 1$
  - Multiplication Commutative
    $UMult_w(u\,,\,v) = UMult_w(v\,,\,u)$
  - Multiplication is Associative
    $UMult_w(t, UMult_w(u\,,\,v)) = UMult_w(UMult_w(t, u), v)$
  - 1 is multiplicative identity
    $UMult_w(u\,,\,1) = u$
  - Multiplication distributes over addtion
    $UMult_w(t, UAdd_w(u\,,\,v)) = UAdd_w(UMult_w(t, u), UMult_w(t, v))$

Properties of Two's Comp. Arithmetic

- Isomorphic Algebras
  - Unsigned multiplication and addition
    - Truncating to w bits
  - Two's complement multiplication and addition
    - Truncating to w bits
- Both Form Rings
  - Isomorphic to ring of integers mod 2^w
- Comparison to (Mathematical) Integer Arithmetic
  - Both are rings
  - Integers obey ordering properties, e.g.,
    $u > 0 \quad \Rightarrow \quad u + v > v$
    $u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$
  - These properties are not obeyed by two's comp. arithmetic
    TMax `+ 1` `==` *TMin*
    `15213 * 30426 == -10030` (16-bit words)

Floating Point Representation

Floating Point

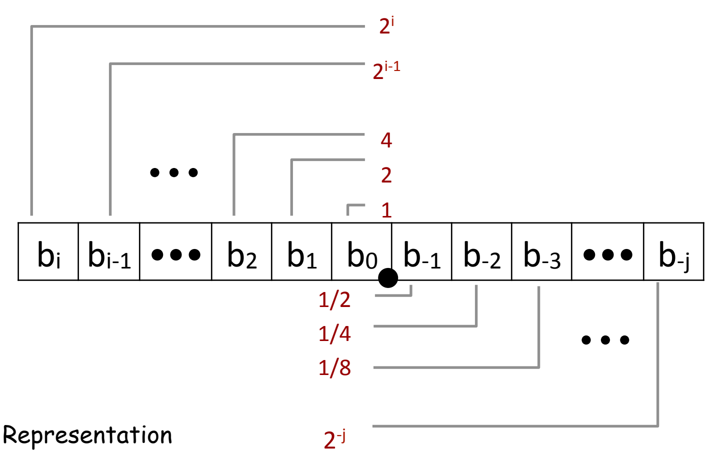
- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

With Slides from Bryant and O'Hallaron

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$

Fractional Binary Numbers: Examples

- Value Representation
  5 3/4 101.11_2
  2 7/8 10.111_2
  63/64 1.0111_2

- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- Limitation
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations

- Value Representation
  - 1/3 $0.0101010101[01]\ldots_2$
  - 1/5 $0.001100110011[0011]\ldots_2$
  - 1/10 $0.0001100110011[0011]\ldots_2$

Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Floating Point Representation

- Numerical Form:

  $(-1)^s\ M\ \ 2^E$

  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

- Encoding
  - MSB s is sign bit s
  - exp field encodes E (but is not equal to E)
  - frac field encodes M (but is not equal to M)

s	exp	frac

Precisions

- Single precision: 32 bits

s	exp	frac
1	8-bits	23-bits

- Double precision: 64 bits

s	exp	frac
1	11-bits	52-bits

- Extended precision: 80 bits (Intel only)

s	exp	frac
1	15-bits	63 or 64-bits

Normalized Values

- Condition: exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as biased value: E = Exp – Bias
  - Exp: unsigned value exp
  - Bias = 2^{k-1} - 1, where k is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)
- Significand coded with implied leading 1: M = $1.xxx…x_2$
  - xxx…x: bits of frac
  - Minimum when 000…0 (M = 1.0)
  - Maximum when 111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

Normalized Encoding Example

- Value: `Float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$
- Significand
  - M = `1.1101101101101`$_2$
  - `frac=` `11011011011010000000000`$_2$
- Exponent
  - E = 13
  - $Bias$ = 127
  - Exp = 140 = `10001100`$_2$
- Result:

s	exp	frac
0	10001100	11011011011010000000000

Denormalized Values

- Condition: exp = 000…0
- Exponent value: E = –Bias + 1 (instead of E = 0 – Bias)
- Significand coded with implied leading 0: M = $0.xxx…x_2$
  - xxx…x: bits of frac
- Cases
  - exp = 000…0, frac = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - exp = 000…0, frac ≠ 000…0
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

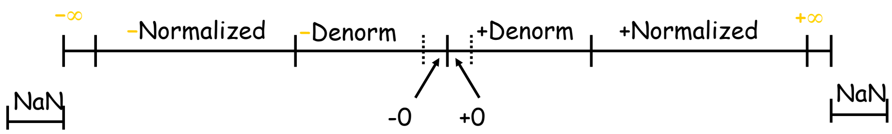
With Slides from Bryant and O'Hallaron

Special Values

- Condition: exp = 111…1
- Case: exp = 111…1, frac = 000…0
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞
- Case: exp = 111…1, frac ≠ 000…0
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

With Slides from Bryant and O'Hallaron

Visualization: Floating Point Encodings



Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Tiny Floating Point Example

s	exp	frac
1	4-bits	3-bits

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the `frac`
- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

With Slides from Bryant and O'Hallaron

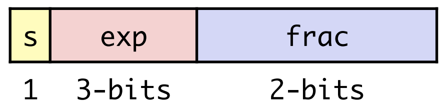
Dynamic Range (Positive Only)

	s exp frac	E	Value	
Denormalized numbers	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001	-6	9/8*1/64 = 9/512	
	…			
	0 0110 110	-1	14/8*1/2 = 14/16	
	0 0110 111	-1	15/8*1/2 = 15/16	closest to 1 below
	0 0111 000	0	8/8*1 = 1	
	0 0111 001	0	9/8*1 = 9/8	closest to 1 above
	0 0111 010	0	10/8*1 = 10/8	
	…			
	0 1110 110	7	14/8*128 = 224	
	0 1110 111	7	15/8*128 = 240	largest norm
	0 1111 000	n/a	inf	

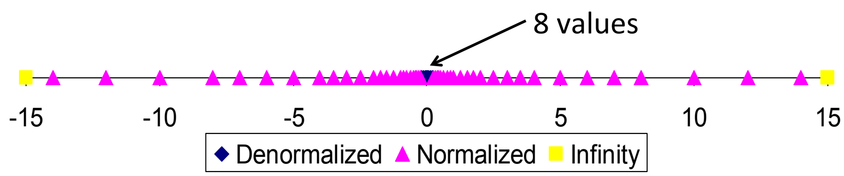
With Slides from Bryant and O'Hallaron

Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 23-1-1 = 3

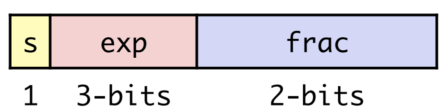


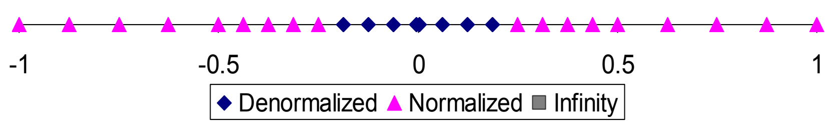
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3





Interesting Numbers{single,double}

Description	exp	frac	Numeric Value
• Zero	00…00	00…00	0.0
• Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
• Single ≈ 1.4×10^{-45}			
• Double ≈ 4.9×10^{-324}			
• Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
• Single ≈ 1.18×10^{-38}			
• Double ≈ 2.2×10^{-308}			
• Smallest Pos. Normalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
• Just larger than largest denormalized			
• One	01…11	00…00	1.0
• Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
• Single ≈ 3.4×10^{38}			
• Double ≈ 1.8×10^{308}			

Special Properties of Encoding

- FP Zero Same as Integer Zero
  - All bits = 0
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Floating Point Operations: Basic Idea

- `x +f y = Round(x + y)`
- `x ×f y = Round(x × y)`
- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`

Rounding

- Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	-$1.50
Towards zero	$1	$1	$1	$2	-$1
Round down (−∞)	$1	$1	$1	$2	-$2
Round up (+∞)	$2	$2	$2	$3	-$1
Nearest Even (default)	$1	$2	$2	$2	-$2

- What are the advantages of the modes?

Closer Look at Round-To-Even

- Default Rounding Mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100\ldots_2$

- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

FP Multiplication

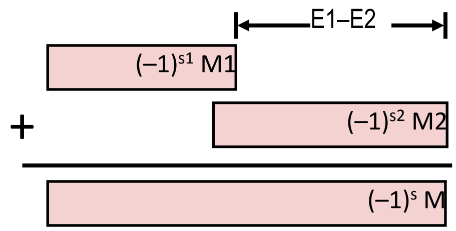
- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: s1 ^ s2
  - Significand M: M1 x M2
  - Exponent E: E1 + E2

- Fixing
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision

- Implementation
  - Biggest chore is multiplying significands

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume E1 > E2
- Exact Result: $(-1)^s\ M\ \ 2^E$
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E1



- Fixing
  - If M ≥ 2, shift M right, increment E
  - if M < 1, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit `frac` precision

Mathematical Properties of FP Add

- Compare to those of Abelian Group
  - Closed under addition?
    - But may generate infinity or NaN
  - Commutative?
  - Associative?
    - Overflow and inexactness of roun
  - 0 is additive identity?
  - Every element has additive inverse
    - Except for infinities & NaNs
- Monotonicity
  - a ≥ b ⇒ a+c ≥ b+c?
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- Compare to Commutative Ring
  - Closed under multiplication?
    - But may generate infinity or NaN
  - Multiplication Commutative?
  - Multiplication is Associative?
    - Possibility of overflow, inexactness of round
  - 1 is multiplicative identity?
  - Multiplication distributes over addition?
    - Possibility of overflow, inexactness of round

- Monotonicity
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$?
    - Except for infinities & NaNs

Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

With Slides from Bryant and O'Hallaron

Floating Point in C

- C Guarantees Two Levels
  - `float` single precision
  - `double` double precision
- Conversions/Casting
  - Casting between `int`, `float`, and `double` changes bit representation
  - `double`/`float` → `int`
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - `int` → `double`
    - Exact conversion, as long as `int` has ≤ 53 bit word size
  - `int` → `float`
    - Will round according to rounding mode

Floating Point Puzzle Answer

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form M x 2^E
- One can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers

Summary

- Binary algebra
  - &, |, ^, ~, <<, >>
- Binary numbers
  - Binary, decimal, hexadecimal
  - Conversions: binary ⇔ decimal; binary ⇔ hexadecimal
  - 2's Complement Representation
    - Signed and unsigned numbers
    - Addition and overflow
    - Multiplication & division
  - Floating point numbers