CS201 – Lecture 4
Data Representation

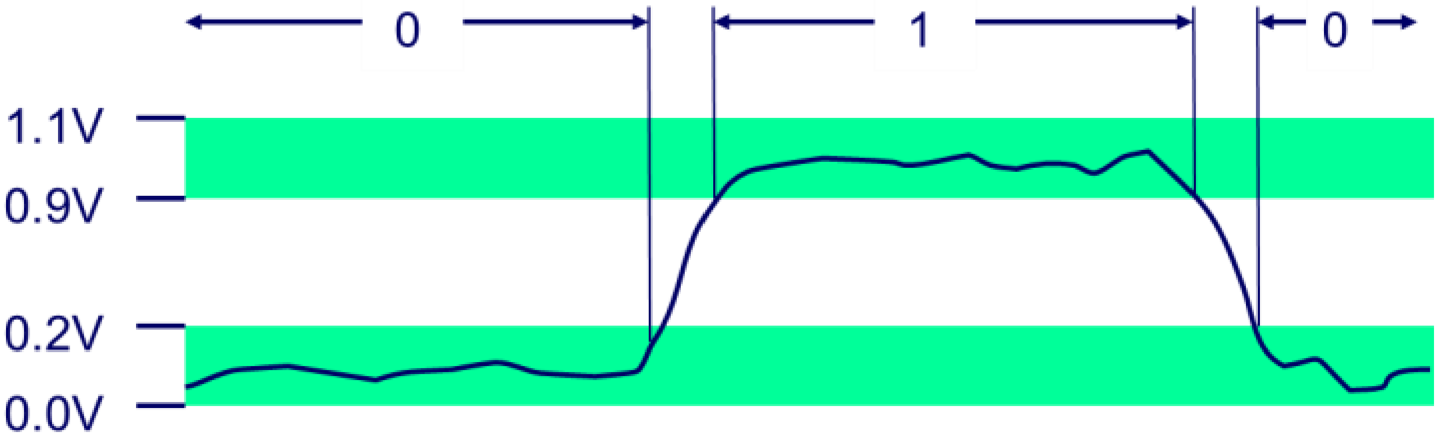
RAOUL RIVAS

PORTLAND STATE UNIVERSITY

Announcements

Bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Binary Numbers

- Base 2 Number Representation
  - Represent 15213_{10} in Binary
  - To convert we use a sequence of divisions by powers of 2:
    - $15213 - 2^{13} = 15213 - 8192 = 7021$
    - $7021 - 2^{12} = 7021 - 4096 = 2925$
    - $2925 - 2048 = 877$
    - $877 - 512 = 365$ (Note we can't divide by 1024)
    - $365 - 256 = 109$
    - $109 - 64 = 45$
    - $45 - 32 = 13$
    - $13 - 8 = 5$
    - $5 - 4 = 1$
    - $1 - 1 = 0$
    - **11101101101101_2**

1	2^0
2	2^1
4	2^2
8	2^3
16	2^4
32	2^5
64	2^6
128	2^7
256	2^8
512	2^9
1024	2^{10}
2048	2^{11}
4096	2^{12}
8192	2^{13}
16384	2^{14}

Binary to Decimal

- To convert to binary we add all the powers of 2 matching the position of all the ones
- **$11101101101101_2 = 2^{13} + 2^{12} + 2048 + 512 + 256 + 64 + 32 + 8 + 4 + 1 = 15213$**

1	2^0
2	2^1
4	2^2
8	2^3
16	2^4
32	2^5
64	2^6
128	2^7
256	2^8
512	2^9
1024	2^{10}
2048	2^{11}
4096	2^{12}
8192	2^{13}
16384	2^{14}

Hex Numbers

- Base 16 Number Representation:
  - Numerals: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Similar process to convert to Base 16 than to Base 2
- Represent 15213_{10} in Hex
  - $15213 – 3 * 16^3 = 2925$
  - $2925 – 11 * 16^2 = 109$
  - $109 – 6 * 16^1 = 13$
  - $13 – 13 = 0$
  - $3B6D_{16}$ (Also written as 0x3B6D or 3B6Dh)

1	16^0
16	16^1
256	16^2
4096	16^3
65535	16^4

- The process is the same for any base!

Hex to Decimal

- Represent $3B6D_{16}$ in Decimal
  - $3 * 16^3 + 11 * 16^2 + 6 * 16^1 + 13 = 15213_{10}$

1	16^0
16	16^1
256	16^2
4096	16^3
65536	16^4

Representing a BYTE

- 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 $7B_{16}$ in C as
      - 0x7B
      - 0x7b

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

- 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

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

Exclusive-Or (Xor)

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

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- 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

Basic Binary Arithmetic - Addition

- Binary addition by hand is similar to its base-10 addition ("grade-school algorithm")

```
  1      1
  01101001
+ 01010101
----------
  10111110
```

```
 1  1111
  11011111
+ 10000110
----------
 101100101
```

Basic Binary Arithmetic - Multiplication

- Binary multiplication by hand is similar to its base-10 multiplication (“grade-school algorithm”)

```
      1101001
×         101
-------------
      1101001
+   0000000
   1101001
-------------
   1000001101
```

```
       11011111
×         10000
---------------
   110111110000
```

The same trick of shifting left applies ☺

Representing Sets (Bitmasks)

- Representation
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$

    - 01101001 { 0, 3, 5, 6 }
    - *76543210*

    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- Operations
  - & Intersection 01000001 { 0, 6 }
  - | Union 01111101 { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference 00111100 { 2, 3, 4, 5 }
  - ~ Complement 10101010 { 1, 3, 5, 7 }
- Compact and Efficient way to represent Bitmasks (e.g flags or switches, etc)

Boolean vs Logic Operators in C

- Boolean Operators: &, |, ~, ^
- Logic Operators: &&, ||, !
  - View 0 as “False”
  - Anything nonzero as “True”
  - Lazy Evaluation
    - Stop evaluation as soon as TRUE or FALSE is determined

```
(p == 0 || p != 0 || q== 0 )
```

    - (q==0) never gets evaluated

    - How about this one: Are we accessing a NULL pointer?

```
(p != NULL && p->next !=NULL)
```

No, p -> next is not evaluated if p==NULL

Two's Complement Representation

- Signed Integer representation in modern computers
  - Suggested by Von Neumann in 1945
- Positive Integers are represented by themselves
- Negative Integers are represented by its Two's complement
- The two's complement TC(n) of an N-bit number n is defined as the complement with respect to 2^N:

$$\mathrm{TC(n)} = 2^N - n$$

- For a 16-bit Integer:
  - **15213 = 00111011 01101101_2**
  - **-15213 = TC(15213) = 2^{17} – 15213 = 1100010010010011_2**

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

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

Negation: Complement & Increment

- CLAIM: We can compute the Two's complement using the following formula:

~x + 1 = -x

  - The proof is beyond the scope of this class, but, note that:

```
~x + x = 1111…111 = -1
```

- Much easier to build circuitry to compute it
- Also faster manual computation

Complement & Increment Examples

x = 15213

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

x = 0

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

Unsigned & Signed Numeric Values

X	Unsigned	Signed
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

Integer Binary Subtraction

- Binary subtraction is done as an addition of the minuend plus the two's complement of the subtrahend
  - Ignore the carry over at the end! (Modular arithmetic)

```
  01101001
- 01010101
```

Two's Complement

```
 111 1 11
  01101001
+ 10101011
 100010100
```

105 – 85 = 276 mod 256 = 20

$$s \quad = \quad \mathrm{USub}_w(u\,,\,v) \quad = \quad u + (\sim v + 1) \;\; \mathrm{mod}\ 2^w$$

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 left
- ANSI C does not specify if Right Shift is Logical or Arithmetic
  - Usually Arithmetic Shifts

Numeric Ranges

Unsigned Values

- *UMin* = 0
  000…0
- *UMax* = $2^w - 1$
  111…1

Signed 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

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
  - $UMax = 2 * TMax + 1$

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

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 leaves bit pattern unchanged!

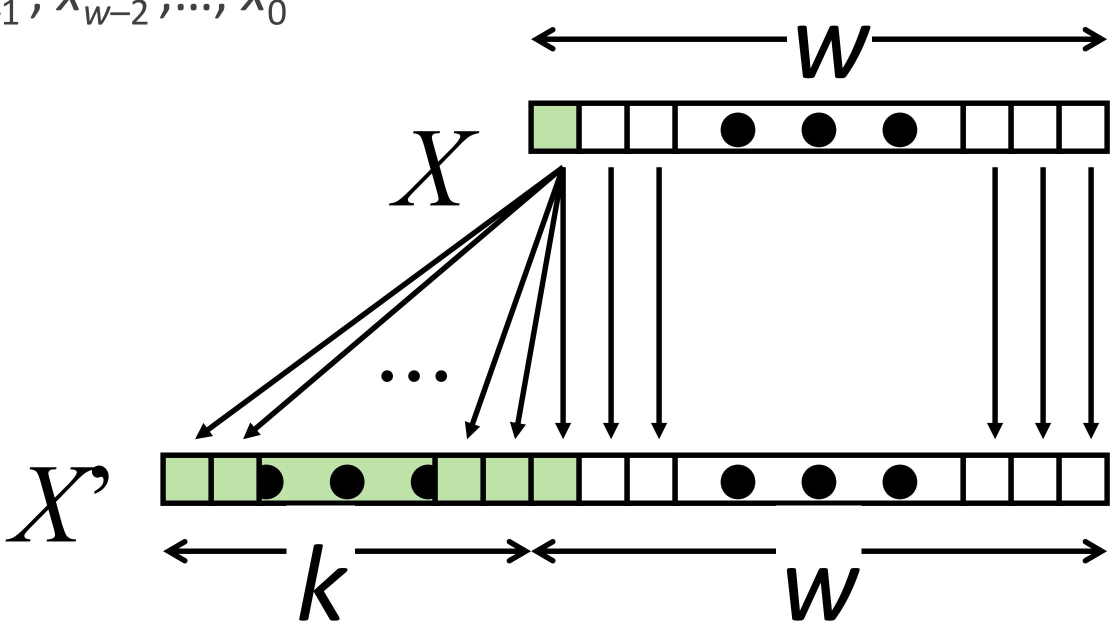
```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Sign Extension

- Given w-bit signed integer x
- Convert it to $w+k$-bit integer with same value
  - E.g. Convert a 16-bit integer to a 32-bit integer
- 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

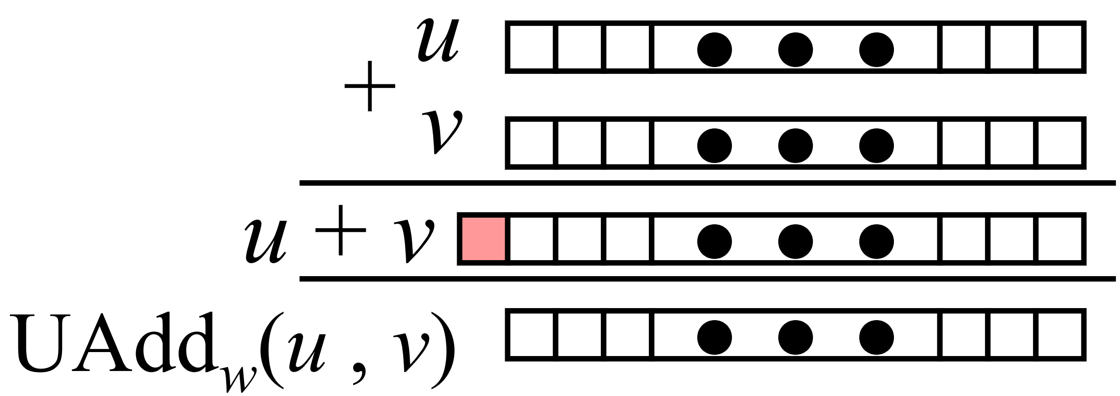
- Converting from smaller to larger integer data type
- C automatically performs sign extension

Unsigned Addition

Operands: *w* bits

True Sum Length : *w*+1 bits

Discard Carry: *w* bits



- Standard Addition Function
  - Ignores carry output
    - In assembly we can use the carry output to see if we overflow
- Implements Modular Arithmetic

$$s \quad = \quad \mathrm{UAdd}_w(u , v) \quad = \quad u + v \mod 2^w$$

Mathematical Properties

- Modular Addition Forms an *Abelian Group*
  - **Closed** under addition

    $0 \leq \mathrm{UAdd}_w(u, v) \leq 2^w - 1$
  - **Commutative**

    $\mathrm{UAdd}_w(u, v) = \mathrm{UAdd}_w(v, u)$
  - **Associative**

    $\mathrm{UAdd}_w(t, \mathrm{UAdd}_w(u, v)) = \mathrm{UAdd}_w(\mathrm{UAdd}_w(t, u), v)$
  - **0** is additive identity

    $\mathrm{UAdd}_w(u, 0) = u$
  - Every element has additive **inverse**
    - Let $\mathrm{UComp}_w(u) = 2^w - u$

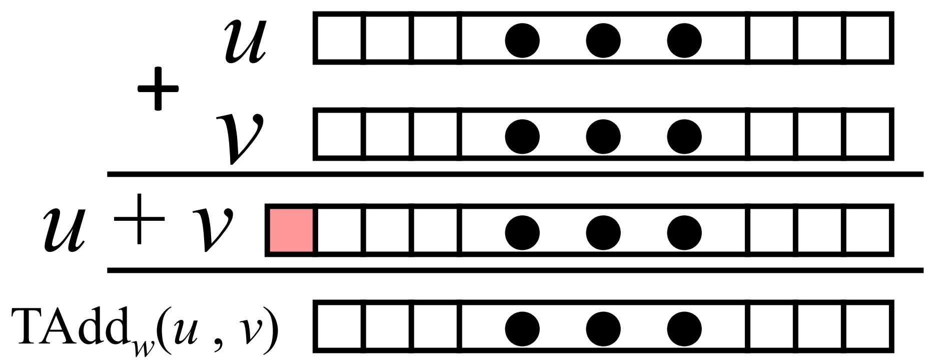
      $\mathrm{UAdd}_w(u, \mathrm{UComp}_w(u)) = 0$

Two’s Complement Addition

Operands: *w* bits

True Sum Length: *w*+1 bits

Discard Carry: *w* bits



- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

Mathematical Properties of TAdd

- Isomorphic Group to unsigneds with UAdd
  - $TAdd_w(u, v) = (\text{signed}) (UAdd_w((\text{unsigned})\ u, (\text{unsigned})\ v))$
    - Since both have identical bit patterns

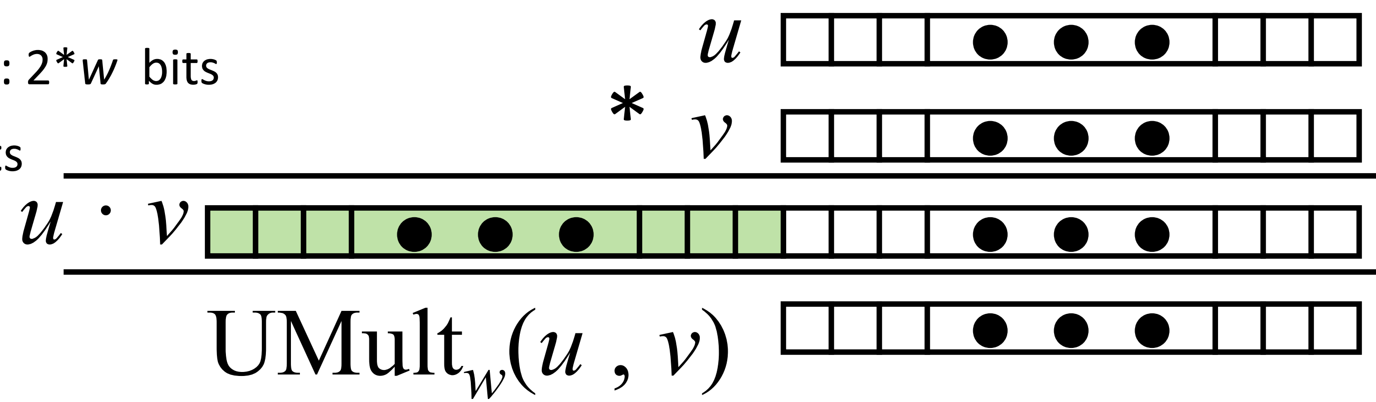
- Two's Complement Under TAdd Forms a Group
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

Unsigned Multiplication

Operands: *w* bits

True Product Length: 2**w* bits

Discard *w* bits: *w* bits



- Standard Multiplication Function in C
  - Ignores high order *w* bits
- IA-32’s 32-bit Multiplication returns a 64-bit integer
- Implements Modular Arithmetic

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

Properties of Unsigned Multiplication

- Unsigned Multiplication with Addition Forms Commutative Ring
  - Addition is commutative group
  - Closed under multiplication

    $0 \leq \text{UMult}_w(u, v) \leq 2^w - 1$
  - Multiplication Commutative

    $\text{UMult}_w(u, v) = \text{UMult}_w(v, u)$
  - Multiplication is Associative

    $\text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v)$
  - 1 is multiplicative identity

    $\text{UMult}_w(u, 1) = u$
  - Multiplication distributes over addtion

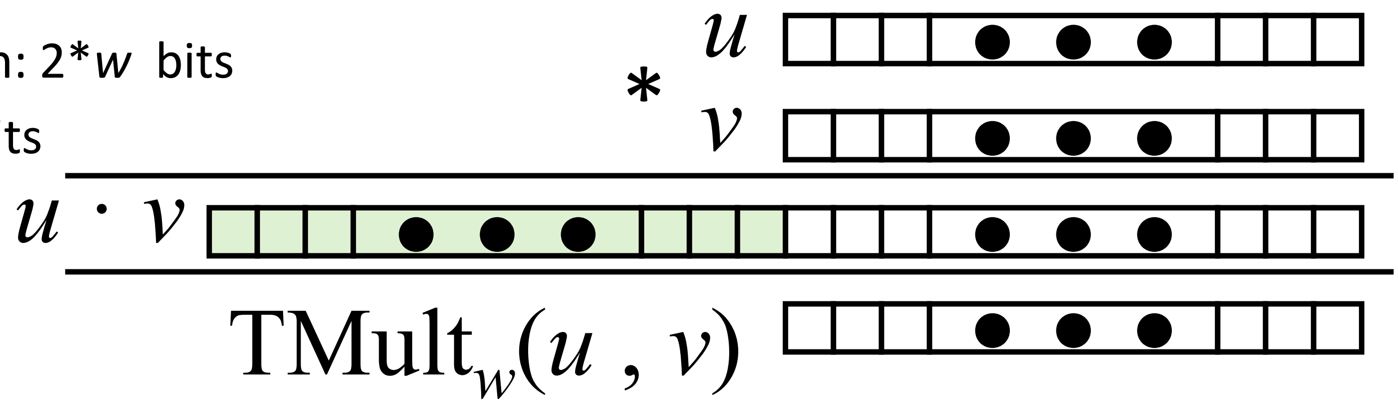
    $\text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v))$

Signed Multiplication

Operands: *w* bits

True Product Length: 2**w* bits

Discard *w* bits: *w* bits



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

Instruction Timings (Cycles)

	8086	286	386	486
16-bit Signed Multiplication	154	21	22	26
16-bit Unsigned Multiplication	118	21	22	26
16-bit Signed Division	184	25	27	27
16-bit Unsigned Division	162	22	22	24

- Value Processor Clock Frequency
  - 8086: 5 Mhz
  - 286: 6 Mhz
  - 386: 12 Mhz
  - 486: 16 Mhz

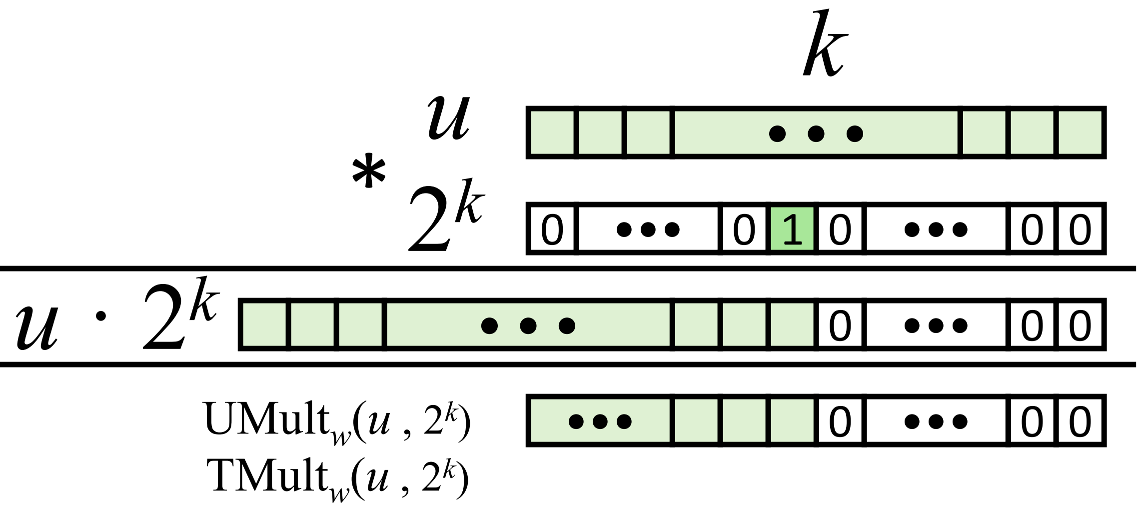
Power-of-2 Multiply with Shift

- 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 faster than multiply
    - Compiler generates this code automatically

Machine Words

- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
- Increasingly, machines have 64-bit word size
  - Potentially, could have 18 PB (petabytes) of addressable memory
  - That’s 18.4×10^{15}
- Machines still support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

Byte Ordering Example

- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

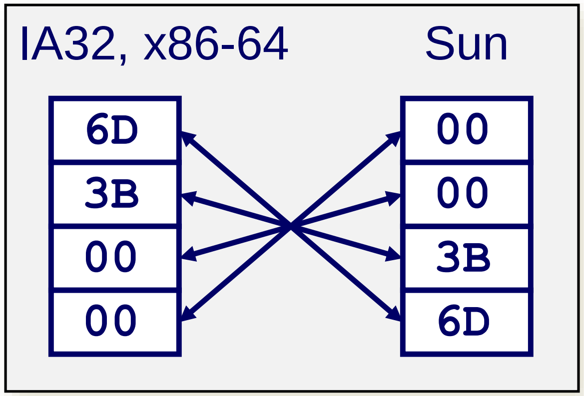
Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

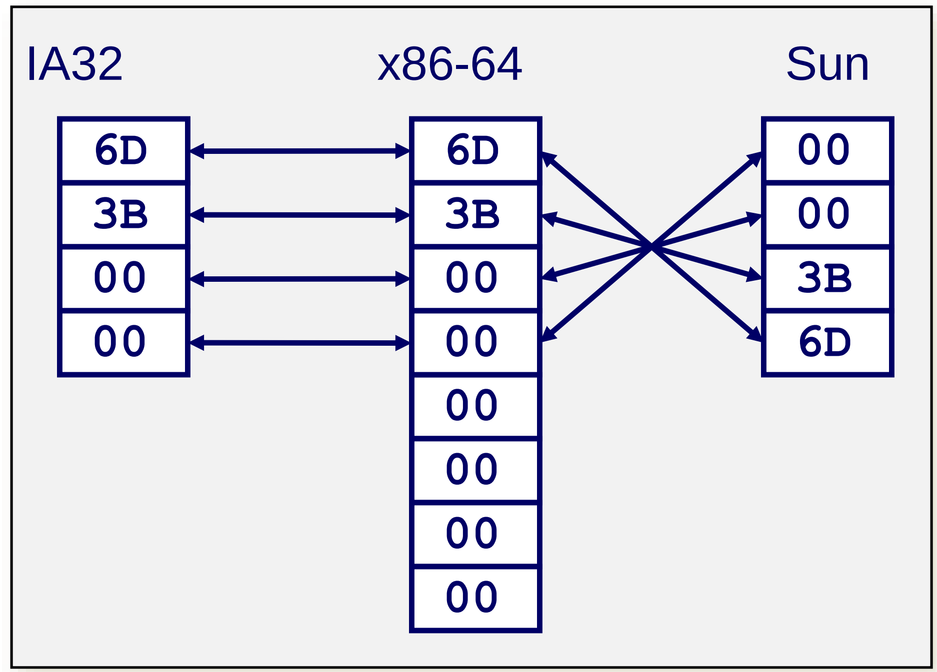
This is important when writing files or connecting to the network

Representing Integers

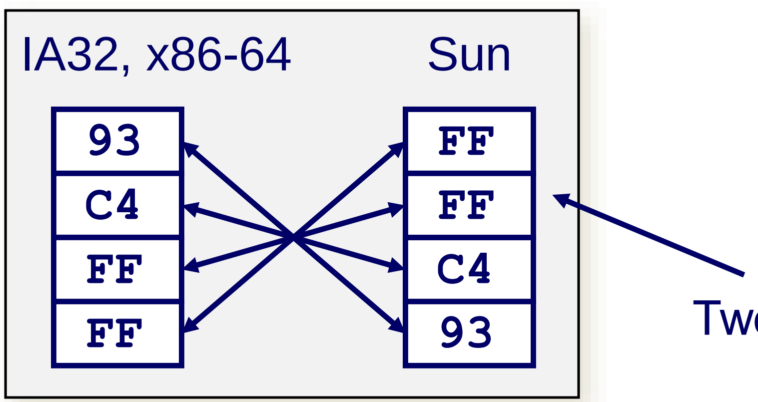
`int A = 15213;`



`long int C = 15213;`



`int B = -15213;`



Two’s complement representation

Summary

- Binary representation is very efficient in bistable components used in digital computers
- Boolean Algebra allows algebraic representation of logic
- Two’s Complement representation is used for signed Integers
  - Modular Integer Addition forms an Abelian Group
  - Modular Integer Addition and Multiplication form a Commutative Ring
  - Logical and Arithmetic Shifts are used to replace multiplication and addition of powers of 2.
- Integer Endianness is hardware dependent
  - Little Endian and Big Endian notations are used by different platforms