CS 31: Intro to Systems
Binary Arithmetic

Vasanta Chaganti & Kevin Webb
Swarthmore College
September 14, 2023
Reading Quiz
So far: Unsigned Integers

• With N bits, we can represent values: 0 to $2^n - 1$

• We can always add 0’s to the front of a number without changing it:

\[ 10110 \quad = \quad 010110 \quad = \quad 00010110 \quad = \quad 0000010110 \]

• 1 byte: char, \underline{unsigned} char
• 2 bytes: short, \underline{unsigned} short
• 4 bytes: int, \underline{unsigned} int, float
• 8 bytes: long long, \underline{unsigned} long long, double
• 4 or 8 bytes: long, \underline{unsigned} long
Unsigned Integers

• Suppose we had one byte
  • Can represent $2^8$ (256) values
  • If unsigned (strictly non-negative): $0 – 255$

252 = 11111100
253 = 11111101
254 = 11111110
255 = 11111111
Unsigned Integers

• Suppose we had one byte
  • Can represent $2^8$ (256) values
  • If unsigned (strictly non-negative): 0 – 255

252 = 11111100
253 = 11111101
254 = 11111110
255 = 11111111

Car odometer “rolls over”.

Any time we are dealing with a finite storage space we cannot represent an infinite number of values!
Unsigned Integers

• Suppose we had one byte
  • Can represent $2^8$ (256) values
  • If unsigned (strictly non-negative): 0 – 255

252 = 11111100
253 = 11111101
254 = 11111110
255 = 11111111

What if we add one more?

Modular arithmetic: Here, all values are modulo 256.
Unsigned Addition (4-bit)

• Addition works like grade school addition:

\[
\begin{array}{c}
1 \\
0110 & 6 \\
+ 0100 & + 4 \\
\hline
1010 & 10 \\
\end{array}
\]

Four bits give us range: 0 - 15
Unsigned Addition (4-bit)

- Addition works like grade school addition:

\[
\begin{array}{cccc}
& 1 & & \\
 0110 & 6 & & 1100 & 12 \\
+ 0100 & + 4 & & + 1010 & + 10 \\
1010 & 10 & & 1 0110 & 6 \\
\end{array}
\]

^carry out

Four bits give us range: 0 - 15

Overflow!

*Carry out is indicative of something having gone wrong when adding unsigned values*
Suppose we want to support signed values too (positive and negative). Where should we put -1 and -127 on the circle? Why?

C: Put them somewhere else.
Not Used: Signed Magnitude

• One bit (usually left-most) signals:
  • 0 for positive
  • 1 for negative

For one byte:
  1 = 00000001
  -1 = 10000001

Pros: Negation is very simple!
Not Used: Signed Magnitude

- One bit (usually left-most) signals:
  - 0 for positive
  - 1 for negative

For one byte:

$0 = 00000000$

$-0? = 10000000$

Major con: Two ways to represent zero.
Used Today: Two’s Complement

• Borrow nice property from number line:

```
-1 0 1
```

Only one instance of zero!
Implies: -1 and 1 on either side of it.

• For an 8 bit range we can express 256 unique values:
  • 128 non-negative values (0 to 127)
  • 128 negative values (-1 to -128)
Two’s Complement

• Only one value for zero

• With N bits, can represent the range:
  • $-2^{N-1}$ to $2^{N-1} - 1$

• **Most significant** (first) bit still designates positive (0) /negative (1)

• Negating a value is slightly more complicated:
  
  \[
  1 = 00000001, \quad -1 = 11111111
  \]

From now on, unless we explicitly say otherwise, we’ll assume all integers are stored using two’s complement! This is the standard!
Two’s Compliment

• Each two’s compliment number is now:
  \[-2^{n-1}d_{n-1} + 2^{n-2}d_{n-2} + \ldots + 2^1d_1 + 2^0d_0\]

Note the **negative sign** on just the first digit.
This is why first digit tells us negative vs. positive.

(The other digits are unchanged and carry the same meaning as unsigned.)
If we interpret $11001$ as a two’s complement number, what is the value in decimal?

• Each two’s compliment number is now:
  \[-2^{n-1}d_{n-1} + 2^{n-2}d_{n-2} + \cdots + 2^1d_1 + 2^0d_0\]

A. $-2$

B. $-7$

C. $-9$

D. $-25$
“If we interpret…”

• What is the decimal value of 1100?

• ...as unsigned, 4-bit value: 12 (%u)
• ...as signed (two’s complement), 4-bit value: -4 (%d)

• ...as an 8-bit value: 12
  (i.e., 00001100)
Two’s Complement Negation

• To negate a value $x$, we want to find $y$ such that $x + y = 0$.

• For $N$ bits, $y = 2^N - x$
Negation Example (8 bits)

• For \( N \) bits, \( y = 2^N - x \)

• Negate 00000010 (2)
  • \( 2^8 - 2 = 256 - 2 = 254 \)

• Our wheel only goes to 127!
  • Put -2 where 254 would be if wheel was unsigned.
  • 254 in binary is 11111110

Given 11111110, it’s 254 if interpreted as unsigned and -2 interpreted as signed.
Negation Shortcut

• A much **easier, faster** way to negate:
  • Flip the bits (0’s become 1’s, 1’s become 0’s)
  • Add 1

• **Negate 00101110 (46)**
  • $2^8 - 46 = 256 - 46 = 210$
  • 210 in binary is 11010010

```
46:       00101110
Flip the bits: 11010001
Add 1
+ 1
-46:      11010010
```
Decimal to Two’s Complement with 8-bit values
(high-order bit is the sign bit)

For positive values, use same algorithm as unsigned
For example, 6:  
6 - 4 = 2 (4\cdot2^2)
2 - 2 = 0 (2\cdot2^1): 00000110

For negative values:
1. convert the equivalent positive value to binary
2. then negate binary to get the negative representation

For example, -3:  
3: 00000011
negate: 11111100+1 = 11111101 = -3
What is the 8-bit, two’s complement representation for -7?

For negative values:
1. convert the equivalent positive value to binary
2. then negate binary to get the negative representation

A. 11111001
B. 00000111
C. 11111000
D. 11110011
Addition & Subtraction

• Addition is the same as for unsigned
  • One exception: different rules for overflow
  • Can use the same hardware for both

• Subtraction is the same operation as addition
  • Just need to negate the second operand...

• 6 - 7 = 6 + (-7) = 6 + (~7 + 1)
  • ~7 is shorthand for “flip the bits of 7”
Subtraction Hardware

Negate and add 1 to second operand:
Can use the same circuit for add and subtract:
\[ 6 - 7 = 6 + \sim 7 + 1 \]

Let’s call this possible +1 input: “Carry in”
(0: on add, 1: on subtract)
4-Bit Subtraction Example

Subtraction via addition: $a - b$ is same as $a + \sim b + 1$

Subtraction: flip bits and add 1

\[
\begin{array}{c}
3 - 6 = \text{0011} \\
\text{1001} \quad \text{(6: 0110 \sim 6: 1001)} \\
+ \text{1} \\
\text{1101} = -3
\end{array}
\]

Equivalent addition: don’t flip bits or add 1

\[
\begin{array}{c}
3 + -6 = \text{0011} \\
+ \text{1010} \\
\text{1101} = -3
\end{array}
\]
By switching to two’s complement, have we solved this value “rolling over” (overflow) problem?

A. Yes, it’s gone.

B. Nope, it’s still there.

C. It’s even worse now.
Overflow, Revisited

- Signed
  -0
  -1
  0
  1
  127
  128
  -127
  -128

- Unsigned
  -1
  0
  1
  64
  128
  192

Danger Zone
If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always
B. Sometimes
C. Never
Two’s Complement Overflow For Addition

- **Addition Overflow**: IFF the sign bits of operands are the same, but the sign bit of result is different.
- Not enough bits to store result!

\[
\text{sign of operands} = \text{sign of result}
\]
Two’s Complement Overflow For Addition

- **Addition Overflow**: IFF the sign bits of operands are the same, but the sign bit of result is different.
- Not enough bits to store result!

<table>
<thead>
<tr>
<th>Sign of Operands = Sign of Result</th>
<th>Sign of Operands ≠ Sign of Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No Overflow</strong></td>
<td><strong>Overflow</strong></td>
</tr>
<tr>
<td>3 + 4 = 7</td>
<td>4 + 7 = 11</td>
</tr>
<tr>
<td>0111</td>
<td>0100</td>
</tr>
<tr>
<td>-2 + -3 = -5</td>
<td>-6 + -8 = -14</td>
</tr>
<tr>
<td>1110</td>
<td>1010</td>
</tr>
<tr>
<td>+ 0100 + 1101</td>
<td>+ 0111 + 1000</td>
</tr>
<tr>
<td>0111 1 1011</td>
<td>1011 1 0010</td>
</tr>
</tbody>
</table>

(-5) (2)
Two’s Complement Overflow For Subtraction

• Rule 1:

<table>
<thead>
<tr>
<th>Minuend</th>
<th>Subtrahend</th>
<th>Result</th>
</tr>
</thead>
</table>

- Positive operand - Negative operand = Positive Result: No Overflow
- Positive operand - Negative operand = Negative Result: Overflow
- **Intuition:** We know a positive – negative is equivalent to a positive + positive. If this sum does not result in a positive value we have an overflow

• Rule 2:

<table>
<thead>
<tr>
<th>Minuend</th>
<th>Subtrahend</th>
<th>Result</th>
</tr>
</thead>
</table>

- Negative operand - Positive operand = Negative Result: No Overflow
- Negative operand - Positive operand = Positive Result: Overflow
- **Intuition:** We know a negative – positive number is equivalent to a negative + negative number. If this sum does not result in a negative value we have an overflow
Two’s Complement Overflow For Subtraction

• Rule 1:
  - **Positive operand** - **Negative operand** = Positive Result: No Overflow
  - **Positive operand** - **Negative operand** = Negative Result: Overflow
  - **Intuition**: We know a positive – negative is equivalent to a positive + positive. If this sum does not result in a positive value we have an overflow

Subtrahend and Result have **different sign bits**
- **no overflow**
- \(2 - (-3) = 5\)
  - Minuend: 0010
  - Subtrahend: 1110
  - Result: 0010 + 0011 = 0101

Subtrahend and Result have the **same sign bits**
- **overflow**
- \(3 - (-7) = 10\)
  - Minuend: 0011
  - Subtrahend: 1001
  - Result: 0011 + 0111 = 1010

\(2 - (-6) = 8\)
- Minuend: 0010
- Subtrahend: 1010
- Result: 0010 + 0110 = 1000

\(3 - (-7) = 10\)
- Minuend: 0011
- Subtrahend: 1001
- Result: 0011 + 0111 = 1010

\(2 - (-6) = 8\)
- Minuend: 0010
- Subtrahend: 1010
- Result: 0010 + 0110 = 1000

\(3 - (-7) = 10\)
- Minuend: 0011
- Subtrahend: 1001
- Result: 0011 + 0111 = 1010
## Two’s Complement Overflow For Subtraction

**Rule 2:**

- Negative operand - Positive operand = Negative Result: No Overflow
- Negative operand - Positive operand = Positive Result: Overflow

**Intuition:** We know a negative – positive number is equivalent to a negative + negative number. If this sum does not result in a negative value we have an overflow.

<table>
<thead>
<tr>
<th>Subtrahend and Result have <strong>different sign bits</strong></th>
<th>Subtrahend and Result have the <strong>same sign bits</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>no overflow</strong></td>
<td><strong>overflow</strong></td>
</tr>
<tr>
<td><strong>-2 - (3) = -5</strong></td>
<td><strong>-2 - (7) = -9</strong></td>
</tr>
</tbody>
</table>
| \[ \begin{array}{c}
1110 \\
-0011
\end{array} \]                                    | \[ \begin{array}{c}
1110 \\
-0111
\end{array} \]                                    |
| **-3 - (4) = -7**                                | **-4 - (7) = -11**                              |
| \[ \begin{array}{c}
1110 \\
+1101
\end{array} \]                                    | \[ \begin{array}{c}
1110 \\
+0111
\end{array} \]                                    |
| **1 1011 (-5)**                                  | **1 0011 (-7)**                                 |
| **1 1001 (-7)**                                  | **1 0011 (-6)**                                 |
Two’s Complement Overflow For Subtraction

**Subtraction Overflow Rules Summarized:**

- Overflow occurs IFF the sign bits of the subtraction operands are different, and the sign bit of the Result and Subtrahend are the same as shown below:
  - Minuend - Subtrahend = Result
  - If positive – negative = negative (overflow)
  - If negative – positive = positive (overflow)

- Now that we have rules for two’s complement, let’s revisit unsigned numbers and formalize the overflow rules there!
Recall: Subtraction Hardware

Negate and add 1 to second operand:

Can use the same circuit for add and subtract:

\[ 6 - 7 = 6 + \neg 7 + 1 \]

input 1 -------------------------------------
input 2 --> possible bit flipper --> ADD CIRCUIT --> result
possible +1 input-------->

Let’s call this possible +1 input: “Carry in”
(0: on add, 1: on subtract)
How many of these unsigned operations have overflowed?

Interpret these as 4-bit unsigned values (valid range 0 to 15):

<table>
<thead>
<tr>
<th>Operation</th>
<th>carry-in</th>
<th>carry-out</th>
</tr>
</thead>
</table>
| Addition (carry-in = 0) | \[ \begin{array}{c}
9 + 11 = 1001 + 1011 + 0 = 10100 \\
9 + 6 = 1001 + 0110 + 0 = 01111 \\
3 + 6 = 0011 + 0110 + 0 = 01001 \\
\end{array} \right|
| Subtraction (carry-in = 1) | \[ \begin{array}{c}
6 - 3 = 0110 + 1100 + 1 = 10011 \\
3 - 6 = 0011 + 1001 + 1 = 01101 \\
\end{array} \right|

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
How many of these unsigned operations have overflowed?

Interpret these as 4-bit unsigned values (valid range 0 to 15):

<table>
<thead>
<tr>
<th>Addition (carry-in = 0)</th>
<th>carry-in</th>
<th>carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 + 11</td>
<td></td>
<td>1 0100 =  4</td>
</tr>
<tr>
<td>9 + 6</td>
<td></td>
<td>0 1111 = 15</td>
</tr>
<tr>
<td>3 + 6</td>
<td></td>
<td>0 1001 =  9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subtraction (carry-in = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 - 3</td>
</tr>
<tr>
<td>3 - 6</td>
</tr>
</tbody>
</table>

A. 1  
B. 2  
C. 3  
D. 4  
E. 5  

Pattern?
Overflow Rule Summary

- **Signed overflow:**
  - The sign bits of operands are the same, but the sign bit of result is different.

- **Unsigned: overflow**
  - The carry-in bit is different from the carry-out.

<table>
<thead>
<tr>
<th>$C_{in}$</th>
<th>$C_{out}$</th>
<th>$C_{in}$ XOR $C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

So far, all arithmetic on values that were the same size. What if they’re different?
Sign Extension

• When combining signed values of different sizes, expand the smaller value to equivalent larger size:

```c
char y = 2, x = -13;
short z = 10;

z = z + y;                z = z + x;
```

```
0000000000001010          0000000000000101
0000000000000101
+ 00000010 + 1110011
0000000000000010          1111111111110011
```

Fill in high-order bits with sign-bit value to get same numeric value in larger number of bytes.
Let’s verify that this works

4-bit signed value, sign extend to 8-bits, is it the same value?

0111 ---> 0000 0111 obviously still 7
1010 ---> 1111 1010 is this still -6?

-128 + 64 + 32 + 16 + 8 + 0 + 2 + 0 = -6 yes!
Operations on Bits

• For these, it doesn’t matter how the bits are interpreted (signed vs. unsigned)

• Bit-wise operators (AND, OR, NOT, XOR)

• Bit shifting
Bit-wise Operators

- Bit operands, Bit result (interpret as appropriate for the context)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>&amp; (AND)</th>
<th></th>
<th></th>
<th>(OR)</th>
<th></th>
<th>~ (NOT)</th>
<th></th>
<th>^ (XOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A &amp; B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>~ A</td>
<td>A ^ B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
01101010 & 01010101 & 10101010 & \sim 10101111 \\
\& 10111011 & | 00100001 & ^ 01101001 & 01010000 \\
00101010 & 01110101 & 11000011
\end{align*}
\]
More Operations on Bits (Shifting)

- Bit-shift operators: $\ll$ left shift, $\gg$ right shift

01010101 $\ll$ 2 is 01010100
   2 high-order bits shifted out
   2 low-order bits filled with 0

01101010 $\ll$ 4 is 10100000
01010101 $\gg$ 2 is 00010101
01101010 $\gg$ 4 is 00000110

10101100 $\gg$ 2 is 00101011 (logical shift)
   or 11101011 (arithmetic shift)

Arithmetic right shift: fills high-order bits w/sign bit
C automatically decides which to use based on type: signed: arithmetic, unsigned: logical
Try some 4-bit examples:

bit-wise operations:
• 0101 & 1101
• 0101 | 1101

Logical (unsigned) bit shift:
• 1010 << 2
• 1010 >> 2

Arithmetic (signed) bit shift:
• 1010 << 2
• 1010 >> 2
Up Next

• Circuits
  • How can we build hardware to perform all these operations on bits?