CS 31: Introduction to Computer Systems

03: Binary Arithmetic
January 28
Clicker Question: Have you registered your clicker?

A. Yes
B. No
C. What’s a clicker?

Check your frequency:

- Iclicker2: frequency AA
- Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

Reset:
1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA
vote status light will indicate success
Reading Quiz
Abstraction

- User / Programmer
  - Wants low complexity

- Applications
  - Specific functionality

- Software library
  - Reusable functionality

- Operating system
  - Manage resources

- Complex devices
  - Compute & I/O
Abstraction

Complex devices
Compute & I/O

100000000
101010100
Today

• Binary Arithmetic
  – Unsigned addition
  – Subtraction
• Representation
  – Signed magnitude
  – Two’s complement
  – Signed overflow
• Bit operations
Last Class: Binary Digits: (BITS)

Representation:  
\[ 1 \times 2^7 + 0 \times 2^6 + \ldots + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]

10001111 = 143

one byte is the smallest addressable unit - contains 8 bits
Last Class: Unsigned Integers

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

- one byte
  - $2^8$ (256) values
  - unsigned: 0 – 255

\begin{align*}
0 &= 00000000 \\
1 &= 00000001 \\
2 &= 00000010 \\
\vdots \\
254 &= 11111110 \\
255 &= 11111111
\end{align*}

Number line:

- $0 = 00000000$
- $1 = 00000001$
- $2 = 00000010$
- $\ldots$
- $254 = 11111110$
- $255 = 11111111$

Addition:

\[
255 = 1 \times 2^7 + \ldots + 1 \times 2^0
\]
Last Class: Unsigned Arithmetic (one byte)

- one byte
  - $2^8$ (256) values
  - unsigned: 0 – 255

$252 = \text{11111100}_2$  
$253 = \text{11111101}_2$  
$254 = \text{11111110}_2$  
$255 = \text{11111111}_2$

What if we add one more?  
$255 + 1$ is?

We cannot represent an infinite number of values in a finite storage space.

Car odometer “rolls over”.

Slide 16
Last Class: Arithmetic and Fixed Storage

• Fixed Storage: finite set of values
  – 1 byte: $2^8$ (256) values
  – unsigned values: 0 – 255

• Yields Modular Arithmetic
  – All operations are $\%$ 256
    (eg) $255 + 4 = 259 \% 256 = 3$

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

Addition works like grade school addition:

```
  1
+ 0110   6
+ 0100   + 4
+ 1010   + 10
```

Four bits give us range: 0 - 15
Addition works like grade school addition:

\[
\begin{array}{cccc}
    & 1 & \quad & \\
0110 & 6 & + & 1100 & 12 \\
+ 0100 & + 4 & + 1010 & +10 \\
\hline
1010 & 10 & 10110 & 6 \\
\hline
\end{array}
\]

^carry out

Four bits give us range: 0 - 15

Overflow!
Last Class: Arithmetic and Fixed Storage

- Fixed Storage: finite set of values
  - 1 byte: $2^8$ (256) values
  - unsigned values: 0 – 255
Not Used: Signed Magnitude Representation (for 4 bit values)

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

For one byte:
1 = 00000001
-1 = 10000001

Pros: Negation (negative value of a number) is very simple!

Major con: Two ways to represent zero.

For one byte:
0 = 00000000
-0? = 10000000
Used Today: Two’s Complement Representation (for four bit values)

- Borrow nice property from number line:

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 bit still designates
  - 0: positive
  - 1: negative

- Negating a value is slightly more complicated:
  - $1 = 00000001$, $-1 = 11111111$

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 Complement

• Each two’s complement 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 most significant bit. This is why first bit tells us whether the value is negative vs. positive.
If we interpret 11001 as a two’s complement number, what is the value in decimal?

Each two’s complement number is now:

\[-2^{n-1}d_{n-1} + 2^{n-2}d_{n-2} + \ldots + 2^1d_1 + 2^0d_0\]

A. -2
B. -7
C. -9
D. -25
If we interpret 11001 as a two’s complement number, what is the value in decimal?

Each two’s complement number is now:

\[-2^{n-1}d_{n-1} + 2^{n-2}d_{n-2} + ... + 2^1d_1 + 2^0d_0\]

A. -2

B. -7  \(-16 + 8 + 1 = -7\)

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 comp), 4-bit value: -4 (%d)
• ...as an 8-bit value: 12. (i.e., 0000 1100)
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 the value (2) 00000010
- \( 2^8 - 2 = 256 - 2 = 254 \)

- 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)

• Formally:
  – $2^8 - 46 = 256 - 46 = 210$
  – 210 in binary is 11010010

<table>
<thead>
<tr>
<th>46:</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flip the bits:</td>
<td>11010001</td>
</tr>
<tr>
<td>Add 1</td>
<td>+ 1</td>
</tr>
<tr>
<td>-46:</td>
<td>11010010</td>
</tr>
</tbody>
</table>
Negation Two Ways

<table>
<thead>
<tr>
<th>x</th>
<th>-x</th>
<th>$2^4 - x$</th>
<th>Bit flip + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>10000 − 0000 = 0000</td>
<td>1111 + 1 = 0000</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>10000 − 0001 = 1111</td>
<td>1110 + 1 = 1111</td>
</tr>
<tr>
<td>0010</td>
<td>110</td>
<td>10000 − 0010 = 1110</td>
<td>1101 + 1 = 1110</td>
</tr>
<tr>
<td>0111</td>
<td>1001</td>
<td>10000 − 0111 = 1001</td>
<td>1000 + 1 = 1001</td>
</tr>
</tbody>
</table>
Decimal to Two’s Complement with 8 bit values  
(high-order bit is the sign bit)

for positive values, use same algorithm as for unsigned

• (E.g.) $6 - 4 = 2 \ (4:2^2)$
• $2 - 2 = 0 \ (2:2^1) : \ 00000110$

for negative values:

• convert negation (positive) to binary
• then negate binary to get negative

E.g.: $-3$
• $3: \ 00000011$
• negate: $11111100+1 = 11111101 = -3$
Decimal to Two’s Complement with 8 bit values (high-order bit is the sign bit)

for negative values:
• convert negation (positive) to binary
• then negate binary to get negative

Try converting -7 to Two’s Complement representation

A. 11111001
B. 00000111
C. 11111000
D. 11110011
Decimal to Two’s Complement with 8 bit values (high-order bit is the sign bit)

for negative values:

• convert negation (positive) to binary
• then negate binary to get negative

Try converting -7 to Two’s Complement representation

A. 11111001  
B. 00000111  
C. 11111000  
D. 11110011

-7 = (1) 7: 00000111  
   (2) negate: 11111000 + 1 = 11111001
Addition & Subtraction for Integers

• 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...

• $6 - 7 = 6 + (-7) = 6 + (\sim 7 + 1)$
  – $\sim 7$ is shorthand for “flip the bits of 7”
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)
4-bit signed Examples:

Subtraction via Addition:
- a - b is same as a + ~b + 1

Subtraction: flip bits and add 1

3 - 6 = 0011
1001 (6: 0110 ~6: 1001)
+ 1
1101 = -3

Addition: don’t flip bits or add 1

3 + -6 = 0011
+ 1010
1101 = -3
Signed & Unsigned 4-bit Subtraction:

Unsigned subtraction: flip bits and add 1

13 - 1 =

Signed subtraction: flip bits and add 1

-3 - 1 =

A. 1100 & 1100
B. 1100 & 1010
C. 1010 & 1010
D. 1001 & 1100
Signed & Unsigned 4-bit Subtraction:

Unsigned subtraction: flip bits and add 1

\[ 13 - 1 = 1101 \]
\[ 1110 \quad (1: \ 0001 \ \sim 1: \ 1110) \]
\[ + \quad 1 \]
\[ \underline{1100} = 12 \]

Signed subtraction: flip bits and add 1

\[ -3 - 1 = 1101 \]
\[ 1110 \]
\[ + \quad 1 \]
\[ \underline{1100} = -4 \]
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.
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.

This is an issue we need to be aware of when adding and subtracting!
Overflow, Revisited

- **Unsigned**
  - Range: 0 to 255
  - Danger Zone: 255 to 0

- **Signed**
  - Range: -128 to 127
  - Danger Zone: -128 to 0

Examples:

- **UnSigned**:
  - 192
  - 64
  - 128

- **Signed**:
  - -1
  - 0
  - 1
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
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
Signed (Two’s Complement) Overflow

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

Signed addition (and subtraction):

\[
\begin{align*}
2 + (-1) &= 1 \\
2 + (-2) &= 0 \\
2 + (-4) &= -2 \\
0010 &+ 1111 \\
0010 &+ 1110 \\
0010 &+ 1100 \\
10001 &+ 0000 \\
1110 &+ 0000 \\
0001 &+ 1001 \\
\end{align*}
\]

No chance of overflow here - signs of operands are different!
Signed (Two’s Complement) Overflow

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

Signed addition (and subtraction):

\[
\begin{align*}
2^+ & -1 = 1 & 2^+ & -2 = 0 & 2^+ & -4 = -2 & 2^+ & 7 = -7 & -2^+ & -7 = 7 \\
0010 & 0010 & 0010 & 0010 & 1110 \\
+1111 & +1110 & +1100 & +0111 & +1001 \\
1 & 0001 & 1 & 0000 & 1110 & 1001 & 1 & 0111
\end{align*}
\]

Overflow here! Operand signs are the same, and they don’t match output sign!
Overflow Rules

• Signed (Two’s Complement):
  – Addition:
    • The sign bits of operands are the same, but the sign bit of result is different.
  – Subtraction: turn into and Add
    • negate 2nd operand and add
    • Apply overflow rule to negation of 2nd operand

• Can we formalize unsigned overflow?
  – Need to include subtraction too, skipped it before.
Recall Subtraction Hardware

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

input 1 ----------------------------->
in 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?

4 bit unsigned values (range 0 to 15):

Addition (carry-in = 0)

- $9 + 11 = 1001 + 1011 + 0 = 10100$
- $9 + 6 = 1001 + 0110 + 0 = 1111$
- $3 + 6 = 0011 + 0110 + 0 = 1001$

Subtraction (carry-in = 1)

- $6 - 3 = 0110 + 1100 + 1 = 10011$ (-3)
- $3 - 6 = 0011 + 1010 + 1 = 1101$ (-6)

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

4 bit unsigned values (range 0 to 15):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Carry-in</th>
<th>Carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition (carry-in = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 + 11 = 1001 + 1011 + 0 = 1 0100 = 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 + 6 = 1001 + 0110 + 0 = 0 1111 = 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 + 6 = 0011 + 0110 + 0 = 0 1001 = 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Carry-in</th>
<th>Carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtraction (carry-in = 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 - 3 = 0110 + 1100 + 1 = 1 0011 = 3 (-3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 - 6 = 0011 + 1010 + 1 = 0 1101 = 13 (-6)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

4 bit unsigned values (range 0 to 15):

**Addition (carry-in = 0)**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Carry-in</th>
<th>Carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>$9 + 11$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$9 + 6$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$3 + 6$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

$9 + 11 = 1001 + 1011 + 0 = 10100 = 4$

Subtraction (carry-in = 1)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Carry-in</th>
<th>Carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6 - 3$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$3 - 6$</td>
<td>$1$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

$6 - 3 = 0110 + 1100 + 1 = 0011 = 3$

$3 - 6 = 0011 + 1010 + 1 = 1101 = 13$

A. 1  
B. 2  **Pattern?**  
C. 3  
D. 4  
E. 5
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>$C_{\text{in}}$</th>
<th>$C_{\text{out}}$</th>
<th>$C_{\text{in}} \text{ XOR} C_{\text{out}}$</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 to equivalent larger size:

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

\[
\begin{align*}
\text{z} &= \text{z} + \text{y}; \\
000000000001010 + 00000010 &= 000000000000101 \\
0000000000000100 + 11110011 &= 1111111111100111
\end{align*}
\]

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, 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 you please)

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

<table>
<thead>
<tr>
<th>01010101</th>
<th>01101010</th>
<th>10101010</th>
<th>~10101111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00100001</td>
<td>&amp; 10111011</td>
<td>^ 01101001</td>
</tr>
<tr>
<td>01110101</td>
<td>00101010</td>
<td>11000011</td>
<td>01010000</td>
</tr>
</tbody>
</table>
More Operations on Bits

- Bit-shift operators: \(<<\) left shift, \(>>\) right shift

\[
\begin{align*}
01010101 \; \ll\! \ll \; 2 & \; \text{is} \; \; 01010100 \\
& \quad \text{2 high-order bits shifted out} \\
& \quad \text{2 low-order bits filled with 0} \\
01101010 \; \ll\! \ll \; 4 & \; \text{is} \; \; 10100000 \\
01010101 \; \gg \! \gg \; 2 & \; \text{is} \; \; 00010101 \\
01101010 \; \gg \! \gg \; 4 & \; \text{is} \; \; 00000110 \\
10101100 \; \gg \! \gg \; 2 & \; \text{is} \; \; 00101011 \; \text{(logical shift)} \\
& \quad \text{or} \; \; 11101011 \; \text{(arithmetic shift)}
\end{align*}
\]

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

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

Logical (unsigned) bit shift:
• 1010 << 2 = 1000
• 1010 >> 2 = 0010

Arithmetic (signed) bit shift:
• 1010 << 2 = 1000
• 1010 >> 2 = 1110
How do we represent fractions in binary?
Additional Info: Representing Signed Float Values

• One option (used for floats, **NOT integers**)
  – Let the first bit represent the sign
  – 0 means positive
  – 1 means negative

• For example:
  – 0101  ->  5
  – 1101  -> -5

• Problem with this scheme?
Additional Info: Floating Point Representation

1 bit for sign   sign | exponent | fraction |
8 bits for exponent
23 bits for precision

\[ \text{value} = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{(\text{exponent}-127)} \]

let's just plug in some values and try it out

0x40ac49ba: 0 10000001   01011000100100110111010

sign = 0 exp = 129 fraction = 2902458

\[ = 1 \times 1.2902458 \times 2^2 = 5.16098 \]

I don't expect you to memorize this
Up Next

• C programming