Binary Integers

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Binary integers are one of the basic ways to store information in a modern computer.

B. Outcomes

At the end of today, you should:

- Know the three ways to represent signed binary integers.
- Know the pros and cons of each system.
- Know how to take the negative of a binary number in each system.
- Know how to do subtraction in two's complement.
- Know what overflow is, what it looks like, and when it occurs.

⇒ Apologies: For some of you, this will be a review topic

C. Unsigned Binary Integers

- Our hardware represents data using bits; we use bits to represent binary numbers.
- Given n bits, there are 2^n possible bit patterns.
- We can use them to name 2^n possible items.
- For unsigned binary integers, we read the n-bit string as a base 2 number that's ≥ 0.
  - E.g. for 3 bits: 000, 001, 010, 011, 100, 101, 110, 111 are 0 through 7.
  - The bit positions (left to right) are $2^{n-1}, \ldots, 2^2, 2^1, 2^0$.
  - The largest unsigned n-bit number (has n 1 bits), represents $2^{n-1}+\ldots+2^1+2^0 = 2^n - 1$.

Unsigned Binary Addition

- Unsigned binary addition is like decimal addition: Add column-by-column from right-to-left. If a column yields a result of $2_{10} = 10_2$ or $3_{10} = 11_2$, then carry the left bit (the 1 in $\underline{1}0_2$ or $\underline{1}1_2$) to the next column leftward and keep the right bit (the 0 in $1\underline{0}_2$ or 1 in $1\underline{1}_2$) in the current column.

Unsigned Addition Example

- Decimal $60 + 26 = 86$; binary $111100 + 11010 = 1010110$

+		1	1	1	1	0	0
		0	1	1	0	1	0
							0

In the 1's column, $0 + 0 = 0$

+		1	1	1	1	0	0
		0	1	1	0	1	0
						1	0

In the 2's column, $0 + 1 = 1$

+		1	1	1	1	0	0
		0	1	1	0	1	0
					1	1	0

In the 4's column, $1 + 0 = 1$

			1				
+		1	1	1	1	0	0
		0	1	1	0	1	0
				0	1	1	0

In the 8's column $1 + 1 = 10_2 = 1$ sixteen $+ 0$ eight; write the 0, carry the 1

		1	1				
+		1	1	1	1	0	0
		0	1	1	0	1	0
			1	0	1	1	0

In the 16's column, $1 + 1 + 1 = 11_2 =$ 1 thirty-two and 1 sixteen

		1	1				
+		1	1	1	1	0	0
		0	1	1	0	1	0
	1	0	1	0	1	1	0

In the 32'nds column, $1 + 1 = 10_2$, so there's a carry into the 64's column

Unsigned Binary Subtraction

- Unsigned binary subtraction is like decimal subtraction; again, we go column-by-column, right-to-left. We can't subtract 0 – 1, so we borrow a 1 from the column to the upper left so that we can (in effect) subtract $10_2 - 1 = 01_2$.
- **Repeated borrowing**: It's possible for the column to the upper left to be 0, in which case we have to borrow a 1 from the *column to its left* to get $100_2 - 1 = 011_2$. We continue borrowing from the left until we find a 1 and get $100\ldots0_2 - 1 = 011\ldots1_2$.
- **Fall off left end**: If we fall off the left end of the upper bitstring looking for a 1 to borrow, then either (a) we can declare an error because we're trying to get a negative result (as in 000 – 1 = ???) or (b) pretend there's a 1 to the left of the leftmost upper bit (so that 000 – 1 becomes $1000_2 - 1 = 111_2$). [In this second case, we're basically implementing arithmetic modulo 2^n where n is the number of bits.]

Unsigned Subtraction Example

- Decimal 41 – 22 = 19 = 6-bit binary 101001 - 010110 = 010011

–	1	0	1	0	0	1
	0	1	0	1	1	0
						1

In the 1's column, 1 one – 0 ones = 1 one

–	1	0	1	0	0	1
	0	1	0	1	1	0
					?	1

In the 2's column, we can't calculate 0 – 1 so we'll have to borrow from the 4's column.

				? ?		
–	1	0	1	0	0	1
	0	1	0	1	1	0
					?	1

But the 4's column has 0, so we can't borrow from it

			0	1	?	
−	1	0	~~1~~	0	0	1
	0	1	0	1	1	0
					?	1

But we can borrow 1 eight and get 10_2 fours

			0	~~1~~ 1	1	
−	1	0	~~1~~	~~0~~	0	1
	0	1	0	1	1	0
					?	1

Now we can borrow 1 four from the 10_2 fours, leaving 1 four and 10_2 twos

			0	~~1~~ 1	1	
−	1	0	~~1~~	~~0~~	0	1
	0	1	0	1	1	0
					1	1

And finally, 10_2 twos – 1 two = 1 two

			0	~~1~~ 1	1	
−	1	0	~~1~~	~~0~~	0	1
	0	1	0	1	1	0
				0	1	1

And then 1 four – 1 four = 0 fours

			0	~~1~~ 1	1	
−	1	0	~~1~~	~~0~~	0	1
	0	1	0	1	1	0
			0	0	1	1

And 0 eights – 0 eights = 0 eights

	0	1	0	~~1~~ 1	1	
−	~~1~~	0	~~1~~	~~0~~	0	1
	0	1	0	1	1	0
		?	0	0	1	1

For the 16’s column, we have to borrow from the 32’s column

	0	1	0	~~1~~ 1	1	
−	~~1~~	0	~~1~~	~~0~~	0	1
	0	1	0	1	1	0
		1	0	0	1	1

After borrowing, 10_2 sixteens – 1 sixteen = 1 sixteen

$$
\begin{array}{ccccccc}
 & {}^{0} & {}^{1} & {}^{0} & {}^{\cancel{1}\ 1} & {}^{1} & \\
- & \cancel{1} & 0 & \cancel{1} & \cancel{0} & 0 & 1 \\
 & 0 & 1 & 0 & 1 & 1 & 0 \\ \hline
 & 0 & 1 & 0 & 0 & 1 & 1
\end{array}
$$

And finally in the 32's column, 0 – 0 = 0, and we're done

D. Signed Binary Integers

- We're going to study 3 systems for representing signed binary integers. They all use the **leftmost bit as the sign bit** (tells you if you have a positive or negative number.)
- **Unique reading for a leading 0 bit**: All three systems interpret a bitstring with a leading 0 in the same way, as the same nonnegative number we'd get if we read the bitstring as unsigned. E.g., 011 always means 3.
- **Leading 1 bit means "negative" number**: The 3 systems all interpret a bitstring with a leading 1 as a "negative" number*. **But they'll differ on which negative number**, so a question like "What does 101 mean?" can't be answered without a context. For now, we'll always make the context explicit; at some point, we'll start always using 2's complement.
- **Mismatch of desired features**: There are two features we'd like our systems to have; unfortunately,we can't have both simultaneously
  - **Symmetry**: It would be nice for there to the same number of numbers > and < 0. For some k, we'd like to represent the $2k+1$ numbers $-k$, $-k+1$, $-k+2$, …, -1, 0, 1, 2, …, $k-1$, k.
  - **Power of 2 number of bitstrings**: If we have n bits, then we can write 2^n bitstrings, with half of them (2^{n-1} bitstrings) beginning with 0 and the other half beginning with 1. But 2^n is an even number and $2k + 1$ is an odd number, so they can't possibly be equal.

---

* I put "negative" in quotes because of the weird situation with "negative 0" that we'll see in 2 of the systems.

- **Irregularities in all 3 systems**: Each of the 3 schemes for representing negative numbers has some irregular feature:
  - (In 2's complement): Unequal numbers of integers strictly > and < zero.
  - (In Sign-Magnitude and 1's Complement): Multiple representations of zero.

E. Sign-Magnitude Negative Numbers

- To take the negative of a number, flip its sign bit. (Easy!)
  - (This is what we do at the blackboard except in decimal, and instead of writing leading 0 and 1, we write + and –.)
- **Examples**: 0111 represents 7; In sign-magnitude, 1111 represents –7.
- Note you can take the negative of a negative number: Flipping the sign bit of 1111 takes us to 0111: $-(-7) = 7$.
- **Two representations of zero**: The negative of 0000 is 1000 (because we just flip the sign bit). But arithmetically, the negative of zero should equal zero, so 0000 and 1000 both represent zero: 0000 is "positive" zero and 1000 is "negative" zero.
- **Sign-Magnitude Addition and Subtraction**: Addition and subtraction of sign-magnitude numbers involves multiple algorithms. Let P and N represent positive and negative numbers respectively. Sign-magnitude $P_0 + P_2$ looks like unsigned addition on the same bitstrings; $N_0 + N_1$ equals $-(-N_0 + -N_1)$, so we can do it using unsigned addition and some sign bit flips. But $P + N$ (and $N + P$) are more complicated: If $P \geq -N$, then $P + N$ equals $P - (-N)$, but if $P < -N$, then $P + N$ equals $-((-N) - P)$. Plus, we have to figure out if/when negative 0 comes into play.
- **Example**: As unsigned bitstrings, 1011 + 1001 = 10100 (in decimal, 11 + 9 = 20) In sign-magnitude, 1011 + 1001 = –(0011 + 0001) = –0100 = 1000 (in decimal, $-3 + -1 = -(3 + 1) = -(4) = -4$).

F. One's Complement

- In one's complement, to take the negative of a number, flip all its bits. (Again, a pretty easy algorithm.)
- **Notation**: Let's write $\sim x$ (tilde x) to mean the result of flipping all of x's bits (also known as the bitwise negation of x). (This is what tilde means in C, by the way.)

   - **Example**: In 4-bit 1's complement, $-7 = \sim 0111 = 1000$.
- You can also think of negative as being like unsigned subtraction from 11…1.
   - **Example**: Here $-7 = 1111 - 0111 = 1000$.
- The decimal analog to binary 1's complement is "9's complement" — you subtract each digit from 9. E.g., $-1234 = 9999 - 1234 = 8765$.
- **Irregularity in 1's complement**: Like sign-magnitude, 1's complement has a "negative" zero, though it's written differently: $1111 = \sim 0000$ is negative zero. (In sign-magnitude, $-0000 = 1000$ and $1111 = -0111 = -7$.)
- As in sign-magnitude, $P + N$ and $N_0 + N_1$ involve taking the negative and using subtraction or addition.

G. Signed Subtraction as Adding the Negative

- It's a law of algebra that $x - y = x + (-y)$.
- It would simplify the circuitry if we could treat subtraction as adding the negative.
- Let's look at subtracting when $y = 1$. We want $1 - 1 = 1 + (-1) = 0$. In 4 bits, we want 0001 + ???? = 0000 for some bitstring ????. If you think about it, you'll see that ???? = 1111 because $0001 + 1111 = {}^{1}0000$ where the tiny leading 1 indicates a carry out of 1 from the sign bit position.
   - Note $0 - 1 = 0 + (-1) = -1$: In binary, $0000 - 0001 = 0000 + 1111 = 1111$.
   - Just more generally, $x + \sim x = 11\ldots1_2 = -1_{10}$. But since $-1+1 = 0$, we get $x + (\sim x + 1) = 0$, so $-x = \sim x + 1$.
   - This scheme for taking the negative is called **2's complement**.

H. Two's Complement

- Repeating, in 2's complement, $-x = \sim x + 1$: to take the negative of a number, take the one's complement and add 1 (as unsigned addition).
- 2's complement has the pleasant property that $x - y = x + (-y)$ where we perform the plus operation in the same way as unsigned addition (and ignore any carry out from the sign bit position).

- **Another characterization of 2's complement**: Recall that $\sim x = 11\ldots1 - x$ (using unsigned subtraction). If we have n bit binary numbers then $11\ldots1 = 2^n - 1$, so in 2's complement, $-x = \sim x + 1 = (2^n - 1) - x + 1 = 2^n - x$ (where all the binary + and – are done as in unsigned + and –). E.g., $-0001 = 10000 - 1 = 1111$.
  - It's the power of 2 in $-x = 2^n - x$ that makes it "2's" complement.
- **Shortcut for taking 2's complement negative**: If you think about it, you'll realize that $-100\ldots0$ = itself because $-100\ldots0 = 011\ldots1 + 1 = 100\ldots0$.
  - The generalization of this is that to take -x, if you're going through the bitstring left-to-right, you flip all the bits of x and stop short of the rightmost 1 bit.
  - **Example**: −101001000 breaks up into −10100 1000 (adding a space for visibility) = ~10100 concatenated with 1000 = 01011 1000.
  - If you process the bitstring right-to-left, you skip over any rightmost 0 bits to find the rightmost 1 bit and then flip all the rest of the bits.
  - For both algorithms, remember that there may not be any rightmost 0 bits.
    - **Example**: $-1_{10} = -0001$ = (~000) 1 = 111 1
  - And as a kind-of-special cases, you don't flip any bits of 00…0 or 100…0.
  - Having −0 = −0000 = 0000 makes sense and it tells us we only have one version of zero.
- **Irregularity of 2's complement**:
  - Having −1000 = 1000 can't be arithmetically correct: 1000 + 0111 = 1111. I.e., $1000_2 + 7_{10} = -1$, so $1000_2 = -1 - 7 = -8$.
  - More generally, with n bits, a 1 followed by n – 1 zero bits represents -2^{n-1}.
  - The problem with two's complement is that it has one more negative number than positive number, and the negative of the most negative number is itself.

I. *Overflow*

- Overflow occurs when you try to go too far from zero. (The result of some operation on n bits doesn't fit into n bits.)
- Overflow occurs

- When adding two positive or negative numbers if the result has the wrong sign: $P_0 + P_1 = N$ or $N_0 + N_1 = P$. One way to detect this: The carry in to the sign bit position ≠ the carry out from the sign bit position.
- In 2's complement when taking the negative of the most negative number.

- Overflow does not occur
  - When adding a positive and negative number (the result is closer to zero).
  - When taking the negative of the most positive number (the result exists in all three systems).
  - When taking the negative of the most negative number in sign-magnitude or 1's complement (they have same number of numbers > 0 and < 0).

J. A Fourth Representation

- For integers, computer hardware typically uses 2's complement.
- When we look at the IEEE representation of floating-point numbers, we'll see that it uses a "*k*-offset" representation for exponents ($k \geq 0$).
  - The unsigned bitstrings for 0, 1, 2, …, $(2^{n-1} - 1)$ represent $-k$, $(-k + 1)$, $(-k+2)$, …, $(-k + 2^{n-1} - 1)$.
  - In general, for $0 \leq m < 2^n$, we use the bitstring for m to represent $m - k$.
  - In particular, if k is 2^{n-2}, we can represent -2^{n-2} through $(2^{n-2} - 1)$.

Binary Integers

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Binary integers are one of the basic ways to store information in a modern computer.

B. Outcomes

After this activity, you should be able to

- Represent signed binary integers in sign-magnitude, and 1's- and 2's-complement
- Take the negative of a binary number in each of the 3 systems.
- Do subtraction in two's complement.
- Recognize overflow and know when and why it occurs.

C. Problems

1. Complete the following table of representations of 4-bit bitstrings as unsigned, sign-magnitude, and 1's and 2's complement binary numbers: Fill in each entry with the decimal number represented by the bitstring under the given representation.

Bitstring	*Unsigned*	*Sign-Magnitude*	*1's Complement*	*2's Complement*
0111	7	7	7	7
0110	6	6	6	6
0101	5	5	5	5
0100	4	4	4	4
0011	3	3	3	3
0010	2	2	2	2
0001	1	1	1	1
0000	0	0	0	0
1111				
1110				
1101				
1100				
1011				
1010				
1001				
1000				

2. Use the table from the previous problem to answer the following questions:
   a. What decimal number does 0010 represent in sign-magnitude, 1's complement, and 2's complement?
   b. What is ~0010 (i.e., the bitstring that results from flipping the bits of 0010)?
   c. What (decimal number) does the answer from (b) represent in sign-magnitude?
   d. … in 1's complement?
   e. … in 2's complement?
3. Rewrite the following subtractions in binary, using 5-bit 2's complement. (E.g., 3–1 is rewritten as 00011 – 00001 = 00011 + (–00001) = 00011 + (11110 + 1) = 00011 + 11111 = 00010.)
   a. 7 – 5 = 2
   b. 6 – 8 = –2
   c. –5 – 10 = –15
   d. –12 – 4 = –16
   e. –9 – 9 = ????
4. With an 8-bit number
   a. What is the largest positive number that can be represented? (unsigned? signed?)
   b. What is the largest negative number (i.e., most negative) that can be represented in sign-magnitude?
   c. … in 1's complement?
   d. … in 2's complement?
5. Repeat Problem 4 using *n*-bit numbers.

Activity 2 Solution

Bitstring	*Unsigned*	*Sign-Magnitude*	*1's Complement*	*2's Complement*
0111	7	7	7	7
0110	6	6	6	6
0101	5	5	5	5
0100	4	4	4	4
0011	3	3	3	3
0010	2	2	2	2
0001	1	1	1	1
0000	0	0	0	0
1111	15	−7	−0	−1
1110	14	−6	−1	−2
1101	13	−5	−2	−3
1100	12	−4	−3	−4
1011	11	−3	−4	−5
1010	10	−2	−5	−6
1001	9	−1	−6	−7
1000	8	−0	−7	−8

2 (a) 2, 2, 2; (b) 1101; (c) −5, −2, −3

3a.
```
   7      00111
 - 5     +11011 = -00101
 ----    ---------------
   2      00010 = 2
```

3b.
```
   6      00110
 - 8     +11000 = -01000
 ----    ---------------
 - 2      11110 = -00010 = -2
```

3c.
```
  -5     11011 = -00101 = -5
 -10    +10110 = -01010 = -10
----    --------------------
 -15     10001 = -01111 = -15
```

3d.
```
 -12     10100 = -01100 = -12
 - 4    +11100 = -00100 = -4
----    --------------------
 -16     10000 = -16
```

3e.
```
  -9     10111 = -01001 = -9
 - 9    +10111 = -01001 = -9
----    --------------------
  14     01110 = 14 (overflow!)
```

4a. unsigned: 1111 1111 = 255; signed: 0111 1111 = 127

4b. 1111 1111 = –127

4c. 1000 0000 = –127

4d. 1000 0000 = –128

5a. unsigned: $2^n - 1$; signed: $2^{n-1} - 1$

5b. $11\ldots11 = -(2^{n-1} - 1) = -2^{n-1} + 1$

5c. $100\ldots00 = -(2^{n-1} - 1) = -2^{n-1} + 1$

5d. $100\ldots00 = -2^{n-1}$

