Homework

- Reading and homework:
  - Chapter 13
  - Homework due 10/15 (later than usual – you're welcome!)
- Homework keys posted soon.
- **Don't forget: Prelim I on 10/12!**

COS 140: Foundations of Computer Science

Booth's Algorithm

Fall 2018

The problem

- □ We know how to do addition in the computer…
- □ …but what about multiplication?
- □ For $n \times m$, could just add n to itself m times
- □ But can $\longrightarrow$ lot of additions!

  E.g.: $2,999,111 \times 1,999,999,999$
- □ Can we do better?

The problem

- ☐ *Booth's algorithm: algorithm* for multiplication that:
  - – Uses mathematics insights $\Rightarrow \downarrow\downarrow$ # additions
  - – Can be implemented in hardware
- ☐ First: need to understand how to represent numbers in the computer

Number Representation

Numbers

- ☐ Here: focus only on integers – floating point numbers in later class/courses
- ☐ Many different ways have been tried
  - – E.g., binary coded decimal (BCD)

$$1346 = 0001\ 0011\ 0100\ 0110$$

- ☐ This class: look at most common:
  - – Sign-magnitude representation
  - – Two's complement representation

Sign-Magnitude Representation of Numbers

- Two parts to represent number n:
  - *Sign bit*:
    - leftmost (high-order) bit
    - $1 =$ negative, $0 =$ positive
  - *Magnitude*:
    - Remaining bits
    - $= |n|$

Example: Sign-Magnitude Representation

Represent 12 in 8-bit sign-magnitude representation:

- 12 in binary is 1100
- The sign of 12 is positive, so represented as 0.
- Representation of 12: 0000 1100

Example: Sign-Magnitude Representation

Represent -12 in 8-bit sign-magnitude representation:

- 12 in binary is 1100 (0000 1100 in 8 bits)
- The sign of -12 is negative, so represented as 1.
- Representation of -12: 1000 1100

Number of Bits in Representation

- Each computer/OS/language: several integer representations
- Differ by length (# of bits)
- Need to know how to change size of representation

Changing size of representation

- Smaller $s \rightarrow$ larger l:
  - Sign bit of $l =$ sign bit of s
  - Magnitude of $l =$ magnitude of s
  - Will need to *pad* with 0s to left
  - E.g., extend $1001\ 1010_2$ (-26_{10}) to 16 bits:
    $1000\ 0000\ 0001\ 1010$
- Larger $l \rightarrow$ smaller s:
  - Same idea
  - Will have to *truncate* bits on left
  - What if number too large?

Problems with Sign Magnitude Representation

- *Two* ways to represent 0!
- Operations need to take sign bit into account
- Need *both* addition and subtraction logic

Two's Complement Representation

- Positive numbers: same as sign-magnitude representation
- Negative numbers: use the number's *two's complement*

  The 2's complement of an $b-$bit binary number n is the number n' such that the $b-$bit sum $s = n + n' = 0$.

- Since $n + n' = 0 \;\Rightarrow\; n' = -n$
- Analogy – an *odometer*:

  999999 + 000001 = 000000

  999998 + 000002 = 000000

- So for 6 digits: 999999 represents -1, 999998 is -2, etc.
- For binary:

  111111 + 000001 = 000000

  111110 + 000010 = 000000

Finding 2's complement

- One way (e.g., 4-digit numbers):
  - $n + n' = (1)0000$
  - $\Rightarrow n' = (1)0000 - n$
  - E.g., 10's complement of $0004 = 10000 - 4 = 9996$
  - E.g., 2's complement of $0010_2 = 10000 - 10 = 1110$
- Want a more efficient way
- For 4-digit 10's complement:
  - What number can I add to n to get 9999?
  - Find that, add 1, should be the 10's complement
  - E.g., 10's complement of 0235
    = 9999 - 235 + 1 = 9764 + 1 = 9765
  - Does this work?
    Yes: $0235 + 9765 = 10000$ which is a 4-digit 0.
- Will it work for binary? And can we do it efficiently?

Finding 2's complement

- Problem: How to *efficiently* find the 2's complement?
- Example: find 8-bit 2's complement of $1000\,1100$
  - First find number I can add to give $1111\,1111$, then add 1
  - $1000\,1100 + 0111\,0011 = 1111\,1111$, so...
  - ...$0111\,0011 + 1 = 0111\,0100 =$ 2's complement of $1000\,1100$
- In example, $0111\,0011$ is the *1's complement* of $1000\,1100$
- Easy (efficient) to find: *bitwise negation* of number
- So to find 2's complement of n:
  1. Do bitwise negation of n.
  2. Add 1.
- Both are easy for hardware or software
- Note that leftmost bit still denotes sign

Examples

- Problem: Represent 37 in 8-bit 2's complement form
  - Convert to binary: $0010\,0101$
  - It's positive $\Rightarrow$ done.
- Problem: Represent -37 in 8-bit 2's complement form
  - Negative $\Rightarrow$ find 8-bit 2's complement of 37
  - Convert 37 to binary: $0010\,0101$
  - Find 1's complement: $1101\,1010$
  - Add 1: $1101\,1010 + 1 = 1101\,1011$
- Is this correct?
  - If $1101\,1011$ is $-n$, then $n = -(-n) = -(1101\,1011)$
  - So find 2's complement of $1101\,1011$
  - 2's complement $= 0010\,0100 + 1 = 0010\,0101 = 37_{10}$
  - So $1101\,1011$ represents -37_{10}.

What about 0?

- Let's look at 8-bit 2's complement 0:
  - 1's complement: $1111\,1111$
  - 8-bit 2's complement: $1111\,1111 + 0000\,0001 = 0000\,0000$
- $\therefore$ only one representation for 0.

Extending Two's Complement to More Bits

- Pad highest order bits with the sign bit.
  - Extend our representation of 12 to 16 bits:

$$0000\ 1100 \Rightarrow 0000\ 0000\ 0000\ 1100$$

  - Extend our representation of -12 to 16 bits:

$$1111\ 0100 \Rightarrow 1111\ 1111\ 1111\ 0100$$

    Check it:
    $0000\ 0000\ 0000\ 1011$ – one's complement
    $0000\ 0000\ 0000\ 1100 = 12$
- When create initial representation, make sure have enough bits to have correct sign bit.

Two's complement addition

- Simple: just add the two number, whether they're positive *or* negative!
- E.g., two positive numbers, 4-bit representation: 4 + 3

$$\begin{array}{r} 0100 \\ \underline{+0011} \\ (0)\ 0111 \end{array}$$

- E.g., positive and negative, 4-bit representation: 4 + -3

$$\begin{array}{r} 0100 \\ \underline{+1101} \\ (1)\ 0001 \end{array}$$

- E.g., Two negative numbers, 4-bit representation: -4 + -3

$$\begin{array}{r} 1100 \\ \underline{+1101} \\ (1)\ 1001 \end{array}$$

Two's complement subtraction

- Simple: just negate the *subtrahend* and add to the *minuend*
- E.g., what is $4 - 3$ in 4-bit 2's complement arithmetic?

$$\begin{array}{r} 0100 \\ \underline{+1101} \\ (1)\ 0001 \end{array}$$

What about overflow?

- *Overflow:* when result of computation can't be stored in representation
- E.g., 255 + 255 in 8-bit representation
- How to detect in 2's complement?
- If differ in sign: no overflow possible
- If both positive:
  - Overflow will $\rightarrow$ negative result
  - E.g., 4-bit 2's complement: $7 + 7$

$$\begin{array}{r} 0111 \\ \underline{+0111} \\ (0)\ 1110 \end{array}$$

- If both negative:
  - Overflow will $\rightarrow$ postive result
  - E.g., 4-bit 2's complement: $-7 + -7$

$$\begin{array}{r} 1001 \\ \underline{+1001} \\ (1)\ 0010 \end{array}$$

Basis of Booth's Algorithm

Long-hand Multiplication

- From elementary school...
- For each digit in the multiplier:
  - Start creating partial product in the proper column.
  - Multiply each digit in the multiplicand to form a partial product.
  - Add all the partial products together (with each being in its proper columns).
- Intuition for multiplication of unsigned numbers. Sped up by fact that can only use 1's (add the multiplicand and shift to next column) and 0's (shift to next column).
- We would like to not do so many additions!

Insight Behind Booth's Algorithm

- A block of k 1's in a number is equal to
$$2^n + 2^{n-1} \ldots + 2^{n-k+1}$$
where n is determined by where the block appears.
- E.g., $0001\ 1000_2 (= 24_{10})$; $n = 4, k = 2$:
$$0001\ 1000_2 = 2^4 + 2^3 = 2^4 + 2^{4-2+1} = 2^n + 2^{n-k+1}$$
- Insight: The same block of 1's is also equal to:
$$2^{n+1} - 2^{n-k+1}$$
- E.g., $0001\ 1000_2$:
$$0001\ 1000_2 = 2^5 - 2^3 = 32 - 8 = 24$$
- To find value of a number, simply perform this operation when going in or out of blocks of 1's – saves additions

Why Does This Work?

- If you think about it, adding 2^{n-k+1} to the number is the same as adding a number with only a 1 in that position, which is guaranteed to give a new number with a 1 in 2^{n+1} and 0's where the 1's were:
$$\begin{array}{r} 011100 \\ +000100 \\ \hline 100000 \end{array}$$
- Let the first number be x, with $k = 3$ 1s and $n = 4$
- The second number is 2^{n-k+1}
- The sum is 2^{n+1}
- So: $x + 2^{n-k+1} = 2^{n+1}$
- So: $x = 2^{n+1} - 2^{n-k+1}$

Example of Insight

Convert 11011110 to decimal.

- Standard way
  - $2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1$
  - $128 + 64 + 16 + 8 + 4 + 2$
  - 222
- Using the insight:
  - $(2^8 - 2^6) + (2^5 - 2^1)$
  - $(256 - 64) + (32 - 2)$
  - $192 + 30$
  - 222

Multiplication using the insight

- Suppose we have a number such as 0110_2 we wish to multiply by another number, say 0010_2
- We know that:

$$0110_2 = 2^3 - 2^1$$

- So, multiplying both sides by 0010_2 gives:

$$0110_2 \times 0010_2 = (2^3 - 2^1) \times 0010_2$$

- Which can be rewritten as: $2^3 \times 0010_2 - 2^1 \times 0010_2$
- This can save additions (subtractions):
  - Old way of multiplication: addition for each 1 in multiplier
  - This way: need 1 subtraction for each *group* of 1s, addition for the partial sums (differences)
  - Also need way to multiply by powers of two: just *shifting*

Booth's Algorithm

- Booth's algorithm is just the implementation of this insight, with some clever optimizations
- Requires:
  - way to keep track of which bit we're on
  - way to keep track of beginning, end of sequence of 1's
  - way to form 2's complement
  - way to *shift* over multiplicand for adding
  - way to add
  - holder for product

Registers Used by Booth's Algorithm

Assuming n-bit numbers:

Register	Size	Description
Q	n	Initially holds multiplier, ultimately holds low-order n bits of product
Q-1	1	Holds previous low-order bit of Q – lets us tell if block of 1's has started/stopped
M	n	Multiplicand
A	n	Holds high-order portion of result
Count	–	Holds the number of bits in the multiplicand/multiplier

Operations Used by Booth's Algorithm

Arithmetic Shift: a fast machine operation which moves all bits over one position and repeats the sign bit (most significant bit) in the newly open position.
Compare: a fast machine instruction that checks to see if two bytes or words are the same
Add: a fast machine instruction that adds two numbers together
Complement: a fast machine instruction that gives the complement of all bits (some machines may have a machine instruction for two's complement)

Booth's Algorithm Overview

- Will start with low-order bits of product in A (0s), multiplier in Q
- For each digit seen, regardless of what it is or what was seen before, shift once it has been handled.
  - This is equivalant to moving partial products over in long multiplication.
  - Instead of shifting multiplicand left before adding, we'll shift product (and multiplier) right – product shifts into Q over time, multiplier shifts out.
- When enter a group of 1's from the right, subtract the multiplicand from the accumulating product.
- When leave a group of 1's from the right, add the multiplicand to the accumulating product.
- The last two steps apply the basic insight, multiplied by the multiplicand.

Booth's Algorithm

1. Initialize registers with proper information. Count is the number of bits, A is 0, and Q-1 is 0. Q is the multiplier.
2. Compare the least significant bit of Q and Q-1 to see if entering or leaving a group of 1's:
   (a) If the least significant bit of Q is 1 and Q-1 is 0, subtract M from A.
   (b) If the least significant bit of Q is 0 and Q-1 is 1, add M to A.
   (c) Otherwise, do nothing.
3. Prepare for next bit.
   (a) Arithmetic shift right A, Q, Q-1. (Shift along these registers as though they were one continuous register.)
   (b) Reduce the count by 1.
   (c) If count is 0 end. Result is in AQ. Otherwise, go to step 2.

Example

2 times 7, using 4 bit numbers. Multiplier (Q) is 0111. Multiplicand (M) is 0010. Two's complement of multiplicand: 1110.

A	Q	Q-1	C	
0000	0111	0	4	Initialize; 1-0...
+1110				Subtract M from A (add -2)
1110	0111	0	4	Now shift
1111	0011	1	3	Now shift
1111	1001	1	2	Now shift
1111	1100	1	1	0-1...
+0010				Add M to A (add 2)
0001	1100	1	1	Now shift
0000	1110	0	0	Done: answer = 14

Multiply: -63 x 110

63 = 00111111
1's = 11000000
-63 = 11000001

M	-M
11000001	00111111

110 = 01101110

A	Q	Q-1	Count	
00000000	01101110	0	1000	Initial; just shift
00000000	00110111	0	0111	Entering block; add -M
00111111	00110111	0	0111	Shift
00011111	10011011	1	0110	In block; just shift
00001111	11001101	1	0101	In block; just shift
00000111	11100110	1	0100	Exiting block; add M
11001000	11100110	1	0100	Shift
11100100	01110011	0	0011	Entering block; add -M
00100011	01110011	0	0011	Shift
00010001	10111001	1	0010	In block; just shift
00001000	11011100	1	0001	Exiting block; add M
11001001	11011100	1	0001	Shift
11100100	11101110	0	0000	Done

-6930