Chapter 5: Integer Arithmetic

Integer Arithmetic

We will cover the following operations on integers:

- Integer Addition
- Integer Subtraction
- Integer Multiplication (see Karen's notes)
- Integer Division (see Karen's notes)
- Logical Operations (`.words` only) (see Karen's notes)
- Shifting Operations (`.words` only) (see Karen's notes)

Integer Addition

Arithmetic on integers is different for each of the integer representations discussed in Chapter 4.

All arithmetic in computers is performed with a fixed precision. The number of bits in each operand and the result is fixed.

The addition of binary digits (bits) is the same as the addition of decimal digits. We just have to remember how each value in binary is represented.

$$0_2 + 0_2 = 0_2$$
$$0_2 + 1_2 = 1_2$$
$$1_2 + 0_2 = 1_2$$
$$1_2 + 1_2 = 10_2$$
$$10_2 + 1_2 = 11_2$$
$$1_2 + 1_2 + 1_2 = 11_2$$

carry in	a	b	sum	carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Remember when you're adding two values together, add each place starting from right to left. At each place, add the digits from each value. The *last* digit of that answer goes in the final result and the other digit(s) get carried over to the next place in the addition.

Decimal Example

```
   10011    the digits that were carried over
   849479
+   90349
---------
   939828   the final answer
```

This is the basic method for adding two values. It is used for each representation, but there are some differences as well.

- unsigned
- sign-magnitude
- one's complement
- two's complement

Unsigned Binary Addition

```
carry over bits→
                     11          111 1          1111
         a         0011         101101        00111010 (58)
       + b       + 0001       + 011101      + 00001110 (14)
       ---       ------       --------      ---------------
       sum         0100       1 001010        01001000 (72)
                               overflow
```

Overflow of unsigned binary

If there is a carry over from the addition of the most significant bits, that digit is ignored (thrown away), and OVERFLOW has occurred.

<u>Sign-magnitude Addition</u>

Rules for adding two Sign-Magnitude values

1. Only add integers of same sign.
   (add two positive values or two negative values only)
2. The sign of the result is the same as the sign of the operands (addends).
3. do unsigned addition on magnitudes only
   (do not carry into the sign bit)
4. throw away any carry out of the msb of the magnitude
   (be sure this is not recorded as the sign bit)

Sign-Magnitude Addition examples

two positives
`    111`
`  0 0101 (+5)`
`+ 0 0011 (+3)`
`--------`
`  0 1000 (+8)`
	two negatives
	
	`  1 1101 (-13)`
	`+ 1 0010 (-2)`
	`--------`
	`  1 1111 (-15)`
	don't add the sign bits!
		positive and negative
		
		`  0 01011 (+11)`
		`+ 1 01110 (-14)`
		`---------`
		don't add!
		must do subtraction!

OVERFLOW
`    111`
`  0 11011 (27)`
`+ 0 01110 (14)`
`---------`
`  0 01001`
`overflow`
don't carry over into the sign bit!
	OVERFLOW
	`    1`
	`  1 11011 (-27)`
	`+ 1 11000 (-24)`
	`---------`
	`  1 10011`
	`overflow`
	don't carry over into the sign bit!	

One's Complement Addition

Rules for adding two One's Complement values

1. Add integers of any sign.
2. Do unsigned addition.
3. Don't throw away a carry out of the msb.
   If there's a carry out of msg, add `1` to get the correct result.
   This is called "end-around carry" in hardware implementations.

One's Complement Addition examples

two positives

```
   00111  (+7)
 + 00101  (+5)
 -----------
   01100  (+12)
```

two negatives

```
   111
   111110  (-1)
 + 111101  (-2)
 ------------
 1 111011  (-4)
wrong!
      +  1
 ----------
   111100  (-3)
right!
```

positive and negative

```
   00001  (+1)
 + 11100  (-3)
 ------------
   11101  (-2)
right!
```

positive and negative

```
   01001  (+9)
 + 11100  (-3)
 ------------
 1 00101  (+5)
wrong!
      + 1
 ------------
   00110  (+6)
right!
```

OVERFLOW

```
      1 11
      01011(11)
  +   01001 (9)
  --------
      10100(-12)
    overflow
```

the sign of the result is opposite of the sign of the operands

OVERFLOW

```
          1
        11011  (-4)
   +    10010  (-13)
   ---------
      1 01101  (+13)
wrong!
          +  1
   ---------
        01110  (14)
still wrong!
     overflow
```

the sign of the result is opposite of the sign of the operands

NO OVERFLOW

```
      11
      01101  (+13)
  +   11110  (-1)
  ---------
    1 01011  (+11)  wrong!
        +  1
  ---------
      01100  (+12)  right!
```

NO OVERFLOW
there is no overflow if the two operands have different signs

Two's Complement Addition

Rules for adding two Two's Complement values

a. Add integers of any sign.
b. Do unsigned addition.
c. Don't throw away a carry out of the msb.
   If there's a carry out of msg, just ignore it.

Two's Complement Addition examples

two positives

```
  00111  (+7)
+ 00101  (+5)
 -----------
  01100  (+12)
```

two negatives

```
 1111
 111111(-1)
+111110(-2)
----------
 111101(-3)
right!
```

positive and negative

```
 00001  (+1)
+11101  (-3)
----------
 11110  (-2)
right!
```

positive and negative

```
    1
    01001  (+9)
  + 11101  (-3)
  ------------
    00110  (+6)
right!
```

OVERFLOW

```
    1 11
    01011  (11)
  + 01001  (9)
 ---------
    10100(-12)
wrong!
```

overflow

the sign of the result is opposite of the sign of the operands

OVERFLOW

```
    11100  (-4)
  + 10011  (-13)
   ---------
    01111  (+15)
wrong!
```

overflow

the sign of the result is opposite of the sign of the operands

NO OVERFLOW

```
       1111
       01101  (+13)
   +   11111  (-1)
   ---------
       01100  (+12)  right!
```

NO OVERFLOW

there is no overflow if the two operands have different signs

Biased Integer Addition

Rules for adding two Bias-B values

1. Add integers of any sign.
2. Do unsigned addition and keep the carry out bit (for now).
3. Subtract the bias. This is done as unsigned subtraction.
4. If there's still a carry out of msg, overflow has occurred.

An alternative for students on homeworks and exams,
is to convert to some other number system (like decimal)
do the addition and reconvert to Bias-7.

This alternative method on homeworks/exams is only acceptable for Bias-B numbers.

```
Examples with Bias-7 on 4-bit values

          11
         0011 (-4 in Bias-7)                      0101 (-2 in Bias-7)
       + 1011 (+4 in Bias-7)                    + 0011 (-4 in Bias-7)
       ------                                   ------
         1110 (+7 in Bias-7)                      1000 (1 in Bias-7)
       -  111 (7 in unsigned binary)            -  111 (7 unsigned)
       ------                                   ------
         0001 (-6 in Bias-7)                      0001 (1 in Bias-7)
```

Notice, that this system would not be easy to implement in computer hardware.

Integer Subtraction

The subtraction of binary digits (bits) is performed in the same way as the subtraction of decimal digits.
We just have to remember how each value in binary is represented.

$0_2 - 0_2 = 0_2$
$1_2 - 1_2 = 0_2$
$1_2 - 0_2 = 1_2$
$10_2 - 1_2 = 1_2$
$0_2 - 1_2 = 1_2$ if able to borrow from next place in the value

Remember when you're subtracting one value from another, subtract at each place starting from right to left. At each place, subtract the digits of the second value from the corresponding digit of the first value. It may be necessary to borrow from the place to the left.

Decimal Example

```
     must borrow at this place
      |
    849479
-    90349
---------
    759130   the final answer
```

This is the basic method for subtracting two values. It is used for each representation, but there are some differences as well.

- unsigned subtraction
- sign-magnitude subtraction
- one's complement subtraction
- two's complement subtraction

Unsigned Subtraction

It only makes sense to subtract a smaller number from a larger one

```
       must borrow here
        |
      1011 (11)
-     0111 (7)
------------
      0100 (4)
```

Sign-magnitude Subtraction

Rules for subtracting two Sign-Magnitude values

1. Only subtract integers of the same sign.
   (subtract two positive values or two negative values only)

2. If the signs are different, then change the problem to addition.

   This is okay, since the following identities are true.
   `a - b` is equal to `a + (-b)`
   `a + b` is equal to `a - (-b)`

3. Compare the magnitudes of each value.
   Subtract the smaller magnitude from the larger magnitude.
   (do not subtract the sign bit)

4. The sign is the same if the order of operands stayed the same.
   If the order was switched, the sign changes.

Sign-Magnitude Subtraction examples

If first operand has the larger magnitude:	If second operand has the larger magnitude:

<pre>24 - 7
     0 11000 (⁺24)
   - 0 00111 (⁺7)
   -------------- 	
     0 10001 (⁺17)</pre>
	<pre>6 - 28
	     0 00110 (⁺ 6) switch order, 0 11100 (⁺28)
	   - 0 11100 (⁺28) subtract and - 0 00110 (⁺ 6)
	   -------------- change sign ---------------
	                                        1 10110 (⁻22)</pre>

<pre>⁻24 - ⁻2
     1 11000 (⁻24)
   - 1 00010 (⁻2)
   -------------- 	
     1 10110 (⁻22)</pre>
	If second operand is the larger magnitude: ⁻6 - ⁻28<pre> 1 00110 (⁻ 6) switch order, 1 11100 (⁻28)
	   - 1 11100 (⁻28) subtract and - 1 00110 (⁻ 6)
	   -------------- change sign ---------------
	                                        0 10110 (⁺22)</pre>

One's Complement Subtraction

Try some examples on your own to figure this out!

Two's Complement Subtraction

Don't do subtraction at all.

Change the problem to an equivalent addition problem by taking the additive inverse (two's complement) of the second operand.

`a - b`   becomes   `a + (-b)`

```
        24 - 7 <==> ¯10 + ¯3
 011000 (24)                011000 (24)
-000111 ( 7)   ===>      + 111001 (¯7)
 -------------             --------
                             010001
(17)
```

```
          ¯10 - ⁺3 <==> ¯10 + ¯3

  10110 (-10)                 10110 (-10)
- 00011 (3)       ===>      + 11101 (- 3)
------------                -------
                              10011 (-13)
                         (throw away carry
out)
```

```
Second Example explained:


   10110 (-10)
 - 00011 (3)    -->       00011
 ------------               |
                        \|/
                       11100
                     +     1
                     -------
                       11101 (-3)
 so do

     10110 (-10)
   + 11101 (-3)
   ------------
     10011  (-13)
  (throw away carry out)
```

Biased Integer Subtraction

Rules for subtracting two Bias-B values

a. Subtract integers of any sign.
b. Do unsigned subtraction.
c. Add the bias back in. This is done as unsigned addition.

An alternative for students on homeworks and exams,
is to convert to some other number system (like decimal)
do the subtraction and reconvert to Bias-7.
This alternative method on homeworks/exams is only acceptable for Bias-B numbers.

Example with Bias-7 on 4-bit values

```
  0011 (-4 in Bias-7)              1001 ( 2)
- 1001 (+2 in Bias-7)   ==>      - 0011 (-4)
------                           ------
                                   0110 (-1)
                                 +  111 (add bias)
                                 ------
                                   1101 (6 inBias-7)
```

Notice, that is still hard to implement in computer hardware.

OVERFLOW

OVERFLOW DETECTION IN ADDITION

unsigned -- when there is a carry out of the msb

```
   1000  (8)
  +1001  (9)
  -----
 1 0001  (1)
```

sign magnitude -- when there is a carry out of the msb of the magnitude

```
   1 1000  (-8)
 + 1 1001  (-9)
    -----
   1 0001  (-1)   (carry out of msb of magnitude)
```

2's complement -- when the signs of the addends are the same, and the sign of the result is different

```
   0011  (3)
 + 0110  (6)
 ----------
   1001  (-7)   (note that a correct answer would be 9, but
               9 cannot be represented in 4-bit 2's complement)
```

Note: you will never get overflow when adding 2 numbers of opposite signs

OVERFLOW DETECTION IN SUBTRACTION

unsigned -- never
sign magnitude -- never happen when doing subtraction
2's complement -- we never do subtraction, so use the addition rule on the addition operation done.

MULTIPLICATION of integers

```
0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1
```

-- longhand, it looks just like decimal

-- the result can require 2x as many bits as the larger operand

-- in 2's complement, to always get the right answer without thinking about the problem, sign extend both integers to 2x as many bits (as the larger). Then take the correct number of result bits from the least significant portion of the result. Note that the HW that implements multiplication does not use this algorithm.

2's complement example:

```
          1111 1111          -1
        x 1111 1001       x  -7
     ----------------      ------
           11111111           7
          00000000
         00000000
           11111111
          11111111
         11111111
        11111111
  +    11111111
       ----------------
        1  00000000111
               --------   (correct answer underlined)
```

```
     WRONG !                  Sign extended:
      0011  (3)                0000 0011  (3)
     x 1011  (-5)             x 1111 1011  (-5)
     ------                   -----------
        0011                      00000011
       0011                      00000011
      0000                      00000000
   + 0011                      00000011
   ---------                  00000011
     0100001                 00000011
  not -15 in any            00000011
   representation!       + 00000011
                      ------------------
                             1011110001
                                  ---------
                        take the least significant 8 bits
                        11110001 = -15
```

more about integer multiplication.

```
  multiplicand
x   multiplier
----------------
   product
```

If we do NOT sign extend the operands (multiplier and multiplicand), before doing the multiplication, then the wrong answer sometimes results. To make this work, sign-extend the partial products to the correct number of bits.

To ease our work, classify which do work, and which don't.

```
   +          -          +          -      (muliplicands)
 x +        x +        x -        x -      (mulipiers)
 ----       ----       ----       ----
  OK         sign        |          |
           extend        | take     |
          partial        | additive |
          products       | inverses |

                         -          +
                       x +        x +
                       ----        ----
                        sign        OK
                      extend
                     partial
                     products
```

Example:

```
  without                                with correct
  sign extension                         sign extension

      11100 (-4)                               11100
    x 00011 (3)                              x 00011
    ------------                            ---------
      11100                               1111111100
     11100                               1111111100
    ------------                       ---------------
    1010100 (-36)                         1111110100(-12)
     WRONG!                                  RIGHT!
```

Another example:

```
    without adjustment                with correct adjustment

      11101 (-3)                             11101 (-3)
    x 11101 (-3)                           x 11101 (-3)
   -------------                           -------------
      11101                         (get additive inverse of both)
     11101
    11101                                    00011 (+3)
  +11101                                   x 00011 (+3)
  -----------                              -------------
  1101001001 (wrong!)                        00011
                                    + 00011
                                   ---------
                                      001001 (+9) (right!)
```

DIVISION of integers
unsigned only in this class!
(Don't worry, you'll do lots of division in CS/ECE 552!)

```
             quotient
          ----------------
 divisor | dividend


 written as fractions:      dividend
                            --------
                            divisor

 example:
  15 / 3         1111 / 011

            0101
          -----------
    011   | 1111
           -0
           --
            11
          - 11
          ----
             01
             -0
             --
              11
            - 11
            ----
               0  (remainder)
```

another example:

```
      20 / 3         010100 / 011


                000110 (quotient = 6)
              -----------
        011   | 010100
               -0
              ---
                01
                -0
               ---
                010
                 -0
                ---
                0101
               - 011
                ----
                00100
                - 011
                 ----
                  0010
                 -   0
                  ----
                    10 (remainder = 2)
```

Logical Operations

- All logical operations are done bitwise. This means corresponding bits are compared and the single bit result goes into the answer.
- All logical (and shift and rotate) operations are performed on .word variables only.

```
logical operations
(bitwise -
 each bit position)

         X = 0 0 1 1
         Y = 0 1 0 1
--------------------
   not   X = 1 1 0 0
 X or    Y = 0 1 1 1
 X and   Y = 0 0 0 1
--------------------
 X nor   Y = 1 0 0 0
 X nand  Y = 1 1 1 0
--------------------
 X xor   Y = 0 1 1 0
 X xnor  Y = 1 0 0 1
```

- Each logical operation exists in SAL even though all operations could be synthesized if only not & and were available.
- not z, x
- or z, x, y
- and z, x, y
- nor z, x, y
- nand z, x, y
- xor z, x, y
- xnor z, x, y

Shift Operations

- shift operations are used to change the positions of bits.
- There are three types of shifting
  - logical shifts of bits
  - arithmetic shifts of bits
  - rotate bits

Shift Left Logical (sll)

- bits are shifted to the left in the result
- ignore bit(s) that are shifted off left (msb)
- add a zero to the least significant bit

```
                              0  0 1 1 0 1 0 1
                              /  /  /  /  /  /  /
shift left logical by one      0 1 1 0 1 0 1 0
```

Shift Right Logical (srl)

- bits are shifted to the right in the result
- ignore bit(s) that are shifted off right (lsb)
- add a zero to the most significant bit

```
                              1 0 1 1 0 1 0 1
                               \ \ \ \ \ \ \
shift right logical by one  0  1 0 1 1 0 1 0
```

Shift Left Arithmetic (sll)

- same as shift left logical
- bits are shifted to the left in the result
- ignore bit(s) that are shifted off left (msb)
- add a zero to the least significant bit

```
                                0  0 1 1 0 1 0 1
                                /  /  /  /  /  /  /
shift left arithmetic by one     0 1 1 0 1 0 1 0
```

Shift Right Arithmetic (sra)

- bits are shifted to the right in the result
- ignore bit(s) that are shifted off right (lsb)
- sign extend the most significant bits

```
                                 1 0 1 1 0 1 0 1
                                  \ \ \ \ \ \ \
shift right arithmetic by one   1 1 0 1  1 0  1 0
```

Rotate Left (rol)

- bits are shifted to the left in the result
- save bit(s) that are shifted off (msb)
- place "**shifted**" bit(s) into the least significant bit(s)

```
                        1  0  1 1 0 1  0 1
                             /  /  / /  / /
                            /  /  / /  / /
                           /  /  / /  / /
rotate left by two         1 1 0 1 0 1 1 0
```

Rotate Right (ror)

- bits are shifted to the right in the result
- save bit(s) that are shifted off right (lsb)
- place "**shifted**" bits into the most significant bits

```
                          1 0 1  1  0 1 1 0
                           \  \  \  \  \
                            \  \  \  \  \
                             \  \  \  \  \
                              \  \  \  \  \
rotate right by three   1 1 0  1  0  1  1 0
```

What are logical, shift and rotate operations used for?

- arithmetic operations (mult & div)
- to set and clear bits of a word that are used as boolean (or other) values
- manipulate characters (bytes) in a string of characters
- and…

Store a set of boolean variables in a .word

- Number the bits so that we can refer to them
- We will use the little-endian bit numbering system
- Assign a meaning to each bit (or group of bits)

  bit #:7 6 5 4 3 2 1 0
  attrib: 1 1 0 1 0 0 0 1

  - Bit 0: filled ball = 1 and outline only = 0
  - Bit 1: bounces = 1 and doesn't bounce = 0
  - Bit 2,3,4: size is 0, 1, 2, 3 or 4 inches
  - Bit 5,6,7: which of eight colors

Using Masks to Set/Clear bits

- All logical operations in SAL and MAL can only use .word variables as operands.
- Masks are used with logical operations to set and/or clear specific bits within a word.
- A mask can clear or set some or all bits.
- Some common uses and their masks
  - or z, z, 0x43 (this will set bits 0,1, & 6 of z)
  - and z, z, 0xffffffbc (this will clear bits 0,1, & 6 of z)
  - or z, z, 0xffffffbc (this will set all bits except 0,1, & 6 of z)
  - and z, z, 0x43 (this will clear all bits except 0,1, & 6 of z)

Use masks and logical operations to interpret a specific portion of a .word

- Using some but not all bits of a word

attrib: 1 1 0 1 0 0 0 1

- What operation(s) and masks are needed to determine:
  - Is the ball filled or an outline?   bit 0=1, so it is filled
  - Does the ball bounce? bit 1=0, so it doesn't bounce
  - What size is the ball? bit 4,3,2=100, so size=4
  - What color is the ball? bit 7,6,5=110, so color is 6 *(or whatever color the code 6 represents).*
-

Manipulating characters in a string

- Characters of a string are stored next to each other in memory.
- We can insert, delete and modify individual characters by using logical and shift operations.
- This makes it possible to sort and perform other high level types of character and strings edits.

Store four characters in a .word of memory

- Can number bytes too. (Using big-endian this time)
- Each eight bits represent one character

  4 chars: 0101 0111 0100 1111 0100 1111 0100 0100
  - Byte 0: character 'W' = 0x57
  - Byte 1: character 'O' = 0x4f
  - Byte 2: character 'O' = 0x4f
  - Byte 3: character 'D' = 0x44

- How can we change this to "WORD"?

Finally!

- Why not just use a different variable for each attribute (or character), etc?
  - To save space in memory for programs with many such pieces of data.
  - To speed up execution time, since less data must be loaded from memory.
    (remember: memory accesses are very slow.)