3.2 Addition and Subtraction

Binary Addition:

Grade-school method for adding non-negative decimal numbers

```
      123        same as ...00000123
+      49        same as ...00000049
---------
        2        3+9   = 12        carry = 1
       7         2+4+1 = 07        carry = 0
+     1          1+0+0 = 01        carry out = 0
---------
=     172
```

Same method for adding unsigned binary numbers (3-bit representation)

```
      101        same as ...00000101 (5 decimal)
+     011        same as ...00000011 (3 decimal)
---------
        0        1+1    = 10       carry = 1
       0         0+1+1 = 10        carry = 1
+     0          1+0+1  = 10       carry out = 1
---------
=     000         (actual answer is ...00001000 ->
                 unsigned value too large)
```

Same method for adding 2's-complement signed binary numbers (3-bit)

```
      101        same as ...11111101 (-3 decimal)
+     011        same as ...00000011 (3 decimal)
---------
        0        1+1    = 10       carry = 1
       0         0+1+1 = 10        carry = 1
+     0          1+0+1  = 10       carry out = 1
---------
=     000         (continuing process give ...0000000
                 due to infinite string of carries)
```

Signed (2’s-complement) Overflow:

1) Requires addition of two negative numbers or two positive numbers

    Magnitude of sum of a negative and a positive is less than magnitude of either number added

2) Occurs when sign bit of result is different than sign bit of two numbers added

    Adding two positive numbers always results in carry out of 0 and adding two negative numbers always results in carry out of 1 <u>whether there is signed overflow or not</u>

Binary Subtraction:

Normally accomplished by inverting all bits and adding 1 to the number to be subtracted (with a dedicated hardware unit) and sending the result to the adder

Overflow detection hardware needs to be slightly modified

3.3 Multiplication

Binary Multiplication:

Grade-school method for multiplying non-negative decimal numbers

```
      123       same as ...00000123
X      49       same as ...00000049
---------
     1107       9X123           = 1107
+    492        4X123           = 492
---------
=    6027       1107 + 4920     = 6027
```

Top number (that gets multiplied in its entirety) called ***multiplicand***

Bottom number (that gets broken into 1 digit pieces) called ***multiplier***

Result called ***product***

Same method for multiplying unsigned binary numbers (3-bit representation)

```
       101      same as ...00000101 (5 decimal)
X      011      same as ...00000011 (3 decimal)
-----------
       101      1X101                 = 101
      101       1X101                 = 101
+    000        0X101                 = 000
-----------
=    01111      101 + 1010 + 00000    = 01111
                                      (decimal 15)
```

Since each bit of the multiplier can be only either 0 or 1, each term in the sum can only be either 0 or a replicate of the multiplicand

111 X 111 = 110001 (decimal 7 X 7 = 49) -> multiplying the largest two 3-bit unsigned numbers requires 6 bits to represent

In general, the largest possible product requires twice the number of bits to represent as the multiplier/multiplicand

Signed *Binary Multiplication:*

1) Negate multiplier and/or multiplicand (invert bits and add 1) if sign bit = 1

2) Perform unsigned multiplication

3) Negate product if S_1 XOR S_2 = 1 (where the S_n are the sign bits)

Sequential Multiplier Hardware (un-optimized but conceptually simple version, see Figure 3.3):

One sequential step for each term in sum (as many step as bits in multiplier/multiplicand)

Multiplier Register (same size as multiplier/multiplicand):

LSb determines whether to add multiplicand or not

Shifted right one position each step

Multiplicand Register (twice as large as multiplier/multiplicand):

Contains value to be added or not added to sum in current step

Shifted left one position each step

Product Register (twice as large as multiplier/multiplicand):

Initialized to 0

Contains running sum at each step

Adder (ALU) always generates new running sum using multiplicand, but LSb of multiplier register controls whether product register latches new value (LSb = 1) or retains old value (LSb = 0) via the ***write*** signal input to product register

Sequential Multiplier Hardware (optimized version, see Figure 3.5):

Uses 32-bit instead of 64-bit adder for 32-bit multiplier/multiplicand

Reduces register bits from 160 to 96 for 32-bit multiplier/multiplicand

Multiplicand Register (same size as multiplier/multiplicand):

Contains value to be added or not added to sum in current step

Not shifted

Product/Multiplier Register (twice as large as multiplier/multiplicand):

Upper 32 bits initialized to 0

<u>Lower 32 bits initialized with multiplier value</u>

<u>Shifted right one position each step</u>

Contains running sum at each step in the upper 32+n bits, where n is the step number n = 0 ... 31

Contains the remaining multiplier value in the lower 32-n bits

Adder (ALU) always generates a new value for the most-significant 32 bits of the running sum (less significant bits are always unchanged since 0 is added)

LSb of Product/Multiplier Register determines whether most-significant 32 bits are replaced by adder value or ignored

Parallel Multiplier Hardware (see Figure 3.7):

No registers - purely combinational logic

Terms in the sum can be generated in parallel (MUXs select 00...000 or multiplicand based using multiplier bits as select signal)

31 adders generate sum in parallel (lots more hardware than sequential)

Propagation delay through adders reduced by rearranging add order