Module 1

Examples to work out

Hexadecimal

$3F8_{16} =$

Hexadecimal

$3F8_{16} = 0011\ 1111\ 1000_2$

Base Conversion

Convert $139_{10} \longrightarrow N_8$:

$Q_0 =$

Base Conversion

Convert $139_{10} \longrightarrow N_8$:

$Q_0 = 139$

$Q_1 =$

Base Conversion

Convert $139_{10} \longrightarrow N_8$:

$Q_0 = 139$

$Q_1 = 139/8 = 17 \qquad R_1 = 3$

$Q_2 =$

Base Conversion

Convert $139_{10} \longrightarrow N_8$:

$Q_0 = 139$

$Q_1 = 139/8 = 17$ $R_1 = 3$

$Q_2 = 17/8 = 2$ $R_2 = 1$

$Q_3 =$

Base Conversion

Convert $139_{10} \longrightarrow N_8$:

$Q_0 = 139$

$Q_1 = 139/8 = 17$ $\quad R_1 = 3$

$Q_2 = 17/8 = 2$ $\quad R_2 = 1$

$Q_3 = 2/8 = 0$ $\quad R_3 = 2$

$Q_4 =$

Base Conversion

Convert $139_{10} \longrightarrow N_8$:

$Q_0 = 139$

$Q_1 = 139/8 = 17$ $\qquad R_1 = 3$

$Q_2 = 17/8 = 2$ $\qquad R_2 = 1$

$Q_3 = 2/8 = 0$ $\qquad R_3 = 2$

STOP

Answer:

Base Conversion

Convert $139_{10} \longrightarrow N_8$:

$Q_0 = 139$

$Q_1 = 139/8 = 17 \qquad R_1 = 3$

$Q_2 = 17/8 = 2 \qquad R_2 = 1$

$Q_3 = 2/8 = 0 \qquad R_3 = 2$

STOP

Answer: $139_{10} \longrightarrow 213_8$

Check: $2*8^2 + 1*8^1 + 3*8^0 = 128 + 8 + 3 = 139$

Fractions

$0.125_{10} = ?$

Fractions

$$0.125_{10} = (0.125 \times \frac{2}{2}) = 0.25 \times 2^{-1} = 0 \times 2^{-1} + 0.25 \times 2^{-1}$$

Fractions

$$0.125_{10} = (0.125 \times \frac{2}{2}) = 0.25 \times 2^{-1} = 0 \times 2^{-1} + 0.25 \times 2^{-1}$$

$$= 0 \times 2^{-1} + (0.25 \times \frac{2}{2}) \times 2^{-1} = 0 \times 2^{-1} + 0.5 \times 2^{-2} = 0 \times 2^{-1} + 0 \times 2^{-2} + 0.5 \times 2^{-2}$$

$$= 0 \times 2^{-1} + 0 \times 2^{-2} + (0.5 \times \frac{2}{2}) \times 2^{-2} = 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 0.001_2$$

http://www.easysurf.cc/cnver17.htm#b10tob2

Twos Complement

000...00011 = +3

000...00010 = +2

000...00001 = +1

000...00000 = 0

111...11111 = -1

111...11110 = -2

111...11101 = -3

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

if Remainder – D ≥ 0 {

q_i = 1

Remainder = Remainder – D}

else {

q_i = 0}

D = D/2

Example: 01101101 / 00010101

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

if Remainder – D ≥ 0 {

$q_i = 1$

Remainder = Remainder – D}

else {

$q_i = 0$}

D = D/2

Example: 01101101 / 00010101

Remainder = 01101101

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

  if Remainder – D ≥ 0 {

    q_i = 1

    Remainder = Remainder – D}

  else {

    q_i = 0}

D = D/2

Division Algorithm

$D = 2^{n-1} \times \text{divisor}$ Left shifting

Divisor = 00010101,

D = 2^2 x divisor 2 shifts to the left!

1 shift	00101010
2 shifts	01010100

D = 01010100

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

    if Remainder – D ≥ 0 {

        q_i = 1

        Remainder = Remainder – D}

    else {

        q_i = 0}

D = D/2

Remainder = 01101101

D = 01010100

i = 2

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

if Remainder – D ≥ 0 {

$q_i = 1$

Remainder = Remainder – D}

else {

$q_i = 0$}

D = D/2

Remainder = 01101101

D = 01010100

i = 2

Remainder – D

= 00011001 > 0

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

if Remainder – D ≥ 0 {

$q_i = 1$

Remainder = Remainder – D}

else {

$q_i = 0$}

D = D/2

Remainder = 01101101

D = 01010100

i = 2

Remainder – D

= 00011001 > 0

$q_i = 1$

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

    if Remainder – D ≥ 0 {

        $q_i = 1$

        Remainder = Remainder – D}

    else {

        $q_i = 0$}

D = D/2

Remainder = 01101101

D = 01010100

i = 2

Remainder – D

= 00011001 > 0

$q_i = 1$

Remainder = 00011001

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

  if Remainder – D ≥ 0 {

    $q_i = 1$

    Remainder = Remainder – D}

  else {

    $q_i = 0$}

D = D/2

Remainder = 01101101

D = 01010100

i = 2

Remainder – D

= 00011001 > 0

$q_i = 1$

Remainder = 00011001

D =

Division Algorithm

Remainder = Dividend

$$D = 2^{n-1} \times \text{divisor}$$

For i = n-1 to 0

    if Remainder – D ≥ 0 {

        q_i = 1

        Remainder = Remainder – D}

    else {

        q_i = 0}

D = D/2

Remainder = 01101101

D = 01010100

i = 2

Remainder – D

= 00011001 > 0

q_i = 1

Remainder = 00011001

D = 00101010 shift right

Mantissa-Exponent

- In general, FP are stored as:

  Sign S x (2 power E):

  S: Mantissa

  E: exponent
- Increasing the size of the S enhances its

Mantissa-Exponent

- In general, FP are stored as:

  Sign S x (2 power E):

  S: Mantissa

  E: exponent

- Increasing the size of the S enhances its accuracy, while increasing the size of the exponent increases the

Mantissa-Exponent

- In general, FP are stored as:

  Sign S x (2 power E):

  S: Mantissa

  E: exponent

- Increasing the size of the S enhances its accuracy, while increasing the size of the exponent increases the range of numbers that can be represented.

Hidden Bit Normalization

- Any floating-Point number can be expressed in many ways.
- Thus, the following are equivalent, where the S is expressed in binary form:

  0.110 x (2 power 5),
  1.100 x (2 power 4),
  0.0110 x (2 power 6)

Hidden Bit Normalization

- To simplify operation on floating-point numbers, it is typically required that they be normalized.
- A normalized floating-point number is one in the form

  Sign 1.bbbbb*….*(2 power *E*)

  where *b* is either binary digit (0 or 1).

- Note: There is ***a leading "1"*** in the normalized significand.
- Most floating point formats do not store that ***leading "1"****.*

Hidden Bit Normalization

- This results in having an additional bit of precision on the right of the number, due to removing the bit on the left.
- This missing bit is called the **hidden bit** (also known as a **hidden 1**).
- For example, if the significand in a given format is 1.1010 after normalization, then the bit pattern that is stored is 1010 - the leftmost bit is truncated (or hidden).

Hidden Bit Normalization

Mantissas are normalized so that the binary point falls to the right of the leading non-zero

Binary point is not stored

Leading digit is not stored

Excess-N

- No actual sign bit.
- Represent range of positive and negative numbers - “scale” the entire range so that it fits into the range of positive numbers.
- Ex. Want range [0,255] to map to [–128,128].

  How to do this?

Excess-N

- Choose N to be about half the range (2^{n-1}) and add to all numbers.

- For example, if we are using a 4-bit register, we can represent the unsigned numbers from 0 to 15;

- If we scale the numbers by adding 7 to any number we want to represent, then we can store the numbers from –7 to 8, that is:

Excess-N

Number:

-7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8

Representation:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

The binary representation is always 7 more than the value that is intended, so this would be called “excess-7” notation.

Excess-N

- In general, we would use:
  excess-$(2^{(n-1)} - 1)$ for an n-bit register.

- Addition and subtraction can be performed easily as long as we remember to scale the result back

Excess-N

- Thus, when adding two excess notation representations, we must subtract N to get the correct representation (e.g. -3+-3=-6: 4+4-7=1), and when subtracting we must add another N to get the correct answer (-2-(-3)=1:5-4+7=8).
- This is too cumbersome to use for the main representation for integers.

IEEE-754

- http://babbage.cs.qc.edu/IEEE-54/Decimal.html

- http://www.apropos-logic.com/nc/FPFormats.html