IT5002 Tutorial 1

AY 2025/26 Semester 1

Slides adapted from Theodore Leebrant and Prof. Colin

About Me

- Third Year Computer Science Major
  - Specializing in Algorithms & Theory and Parallel Computing
- Previously TA'd for CS1101S in AY 2024/25 Sem 1
  - Feedback Rating: 4.5/5 (Faculty Average 4.2/5)
- Likes: exercising, Taekwondo

Attendance Policy

- Just show up for the tutorial

Q1. Sign Extension in 2's Complement

Use more bits (m) to represent an n-bit number, where m > n

How? Copy the **M**ost **S**ignificant **B**it (MSB) (m-n) times to the front

For example, 0110 sign extended to 8 bits would be 0000 0110

Show that in general, sign extension is **value-preserving**, i.e. it still represents the same value after extension

For positive values, append 0's to the front, it still has the same value

E.g. $(0100)_{2s} = (0000\ 0100)_{2s} = 4$

For negative values, e.g. $(-3)_{10} = (1101)_{2s} = (1111\ 1101)_{2s}$

Treat the value of the MSB as -2^{n-1}, the other bits are added normally

$-8 + 4 + 0 + 1 = -128 + 64 + 32 + 16 + 8 + 4 + 0 + 1$

- Why can we do this?
- It's the definition of 2's complement

For a number represented in 2's complement using n bits

$b_{n-1}b_{n-2}\ldots b_0$

The value it represents is

$v = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$

If we extend a negative number by adding 1's to the front (to a length of m bits), the new value becomes

$$v' = -b_{m-1}2^{m-1} + b_{m-2}2^{m-2} + \ldots + b_n 2^n + b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_0 2^0$$
$$= -2^{m-1} + 2^{m-2} + \ldots + 2^n + 2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_0 2^0$$

$$
\begin{aligned}
v' &= -2^{m-1} + 2^{m-2} + \ldots + 2^{n} + 2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_0 2^0 \\
&= -2^{m-1} + 2^{n-1}(2^{m-n} - 1) / (2 - 1) + b_{n-2}2^{n-2} + \ldots + b_0 2^0 \qquad \text{(sum of G.P.)} \\
&= -2^{m-1} + 2^{n-1+m-n} - 2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_0 2^0 \\
&= -2^{m-1} + 2^{m-1} - 2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_0 2^0 \\
&= -2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_0 2^0 \\
&= v \text{ (the initial value)}
\end{aligned}
$$

∴ Sign extension in 2's complement is value-preserving

You can check using the 'trick' that is taught: flip all the bits, then add 1

You can also check using the formula: $-x = 2^n - x$

Extra

- Does signed extension work for sign and magnitude? (No)

- Does it work for 1's complement? (Yes)

- There is a general formula for n-bit base-R, R's complement
  - $-x = R^n - x$

- For n-bit base-R, the (R-1)'s complement is
  - $-x = R^n – x - 1$

Q2. Subtraction in 1's complement

1's and 2's complement can be used for fractions as well

The rule for finding the complement is exactly the same

- 1's complement: flip all bits
- 2's complement: flip add bits and add 1 (to LSB)

Formula for (r-1)'s complement of N: $r^n – r^{-m} - N$

where n is the number of bits used for the whole numbers

m is the number of bits used for the fractional part

Recall: 'Algorithm' for addition in 1's complement

1. Perform binary addition on the two numbers
2. If there is a carry out of the MSB, add 1 to the result
    - Note: If there is a fractional part, you don't literally add the decimal value 1
    - You add 1 to the least significant bit of the fractional part, NOT the decimal
3. Check for overflow
    - Overflow only occurs if the sign changes (check the MSB)
    - E.g. positive + positive = negative
    - Or negative + negative = positive
    - Overflow will **not** occur for positive + negative

Algorithm for subtraction in 1's complement:

Convert value being subtracted to its complement, perform addition

(a) $(0101.11)_{1s}$ - $(010.0101)_{1s}$

Step 0: Make sure both numbers are the same no. of bits wide

$(0101.1100)_{1s}$ - $(0010.0101)_{1s}$

Step 1: Convert to complement

$(0101.1100)_{1s}$ + $(1101.1010)_{1s}$

Step 2: Perform addition

```
     0101.1100
+    1101.1010
--------------------
    10011.0110
+   00000.0001 (because of carry out from MSB)
--------------------
     0011.0111
```

(a) $(0101.11)_{1s} - (010.0101)_{1s}$

Check: $(0101.11)_{1s} - (010.0101)_{1s} = 5.75 - 2.3125 = 3.4375$

$(0011.0111)_{1s} = 2^1 + 2^0 + 2^{-2} + 2^{-3} + 2^{-4} = 2 + 1 + 0.25 + 0.125 + 0.0625$

$= 3 + 0.375 + 0.0625$

$= 3.4375$

(b) $(010111.101)_{1s} - (0111010.11)_{1s}$

$= (0010111.101)_{1s} - (0111010.110)_{1s}$ (make both side equal bit width)

$= (0010111.101)_{1s} + (1000101.001)_{1s}$ (take complement)

```
        0010111.101
+       1000101.001
-----------------------
        1011100.110
```

Check: $(0010111.101)_{1s} - (0111010.110)_{1s} = 23.625 - 58.75 = -35.125$
$(1011100.110)_{1s} = -(0100011.001)_{1s} = -(2^5 + 2^1 + 2^0 + 2^{-3})$
$= -(32 + 2 + 1 + 0.125) = -35.125$

Extra

- Why not subtract two numbers directly?
  - A: You would have to build a separate circuit just to handle subtraction
  - Early computer engineers realized this was redundant, and that subtraction was just addition but with complements
  - It is a lot cheaper and faster to just use an adder circuit for both addition and subtraction operations

Q3. Convert Decimal to Fixed Point Binary

Convert the numbers to fixed-point binary in 2's complement, with 4 bits for the integer portion and 3 bits for the fraction portion

(a) 1.75

Recall: to convert a decimal number to base-N,

- For the integer part, do repeated division by N until the quotient becomes 0
- For the fractional part, do repeated multiplication by N until the fractional part becomes 0

Decimal part:

1 / 2 = 0 R1 (quotient is 0, end)

Fractional part:

0.75 x 2 = 1.5

0.50 x 2 = 1.0 (fractional part reaches 0, end)

$1.75 = (1.11)_{2s}$

But we want it in <u>fixed-point</u> binary

$1.75 = (0001.110)_{2s}$

(b) -2.5

First find the fixed-point binary representation of 2.5, then take its complement

Decimal part:

2 / 2 = 1 R0

1 / 2 = 0 R1

Fractional part:

0.5 x 2 = 1.0

2.5 = $(0010.100)_{2s}$

-2.5 = $(1101.100)_{2s}$ (flip all bits, add 1 to LSB)

(c) 3.876

Decimal part:

3 / 2 = 1 R1

1 / 2 = 0 R1

Fractional part:

0.876 x 2 = 1.752

0.752 x 2 = 1.504

0.504 x 2 = 1.008

0.008 x 2 = 0.016

$3.876 \approx (0011.1110)_{2s} = (0011.111)_{2s}$

We perform an additional step so that we know whether to round the representation up or down

Converting back, $(0011.111)_{2s} = 3.875$.

The downside of fixed-point binary representation is that we may lose accuracy

(d) 2.1

Decimal part:

2 / 2 = 1 R0

1 / 2 = 0 R1

Fractional part:

0.1 x 2 = 0.2

0.2 x 2 = 0.4

0.4 x 2 = 0.8

0.8 x 2 = 1.6

2.1 ≈ $(0010.0001)_{2s}$ = $(0010.001)_{2s}$ (again, do one more step to round off)

$(0010.001)_{2s} = 2^1 + 2^{-3} = 2 + 0.125 = 2.125$

In this case, the smallest precision we can have is 0.125

So we cannot represent 0.1 exactly

Q4. IEEE752 single-precision repr.

Represent –0.078125 in IEEE 752 single-precision representation.
Express your answer in hexadecimal.

Step 1: Convert it to binary, then write it in normalized form.

$0.078125 \times 2 = 0.15625$

$0.15625 \times 2 = 0.3125$

$0.3125 \times 2 = 0.625$

$0.625 \times 2 = 1.25$

$0.25 \times 2 = 0.5$

$0.5 \times 2 = 1.0$ (done)

$-0.078125 = -(0.000101)_2 = -(1.01)_2 \times 2^{-4}$

<u>Step 2</u>: Convert the normalized form into its sign/exponent/mantissa representation, in binary

$-(1.01)_2 \times 2^{-4}$

Sign = -1, so the sign bit is 1

Exponent = (127 - 4) = 123 = $(0111\ 1011)_2$ (recall how excess-127 works!)

Mantissa = 010 0000 0000 0000 0000 0000

1	0111 1011	010 0000 0000 0000 0000 0000
Sign (1 bit)	Exponent (8 bits)	Mantissa (23 bits)

Step 3: Convert to hexadecimal

(Group into 4 bits for easier conversion)

1	0111 1011	010 0000 0000 0000 0000 0000
Sign (1 bit)	Exponent (8 bits)	Mantissa (23 bits)

1011 1101 1010 0000 0000 0000 0000 0000

B D A 0 0 0 0 0

Answer: $(\text{BDA0 0000})_{16}$

Q5. MIPS Arithmetic

Write the statements in MIPS using as little instructions as possible.

You may rewrite the equation if necessary to minimize instructions.

Assume register \$s0 is mapped to a, \$s1 to b, \$s2 to c, \$s3 to d

(a) c = a + b

```
add $s2, $s0, $s1
```

(b) d = a + b – c

```
add $s3, $s0, $s1        # d = a + b
sub $s3, $s3, $s2        # d = d – c = (a + b) - c
```

(c) c = 2b + (a – 2)

```
add $s2, $s1, $s1      # c = 2b (alternatively, do shift left by 1 bit)
addi $t0, $s0, -2      # $t0 = a – 2
add $s2, $s2, $t0      # c = 2b + (a – 2)
```

(d) $d = 6a + 3(b - 2c)$

Rewrite:

$d = 6a + 3b - 6c = 3(2a + b - 2c) = 3(2(a - c) + b)$

```
sub $t0, $s0, $s2      # $t0 = a – c
sll $t0, $t0, 1        # $t0 = 2(a – c)
add $t0, $t0, $s1      # $t0 = 2(a – c) + b
sll $t1, $t0, 2        # $t1 = 4(2(a – c) + b)
sub $s3, $t1, $t0      # d = 3(2(a – c) + b)
```

Slides uploaded to https://github.com/michaelyql/IT5002

Email: e1121035@u.nus.edu

Anonymous feedback: https://bit.ly/feedback-michael

(or scan the QR below)