Programming Assignment 2

CS211 Section 10-12

March 11, 2025

Abstract

This Programming Assignments aims to fortify your understanding of how our computers represent data using 0s and 1s. In the first half, you will be implementing conversions of integer-based types. The second type will have you implement conversions of floating point types.

Contents

1 Setting up Autograder

The previous assignment had issues with the autograder: formatting inconsistencies, Makefiles, etc. Instead of using a boring old Python script (boo), this autograder uses Rust's **Foreign Function Interface (FFI)**, making your life easier.

1.1 Setting up Cargo

By default, the Instruction Lab machines **do not** have Rust installed. You will need to install it. To do this, run the following command **on the iLab machine**:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

You will be prompted with:

```
Current installation options:
 default host triple: x86_64-unknown-Linux-gnu
 default toolchain: stable (default)
 profile: default
 modify PATH variable: yes
1) Proceed with standard installation ...
2) Customize installation
3) Cancel installation
```

Press **Enter** to proceed with the standard installation (Option 1). Once complete, run the following command to update your `PATH` environment variables:

```
. "$HOME/.cargo/env"
```

Yes, include the dot at the beginning.

Finally, verify the installation by running:

```
cargo --version
```

1.2 Using the Autograder

With Cargo successfully installed, you can clone the assignment repository from GitHub using:

```
git clone -b PA2 github.com/blablabla
```

Unlike previous autograders, this one compiles your code as a **.so** file, which is a dynamically linked library. The autograder will call functions within this **.so** file and evaluate their return values.

1.2.1 Compiling Your Code

To compile your code, run:

```
cargo run
```

This will generate a `pa2.so` file.

1.2.2 Running Tests

To run all test cases, use:

```
cargo test
```

To run specific test groups, use:

```
cargo test partone      % Runs all partone tests
cargo test parttwo      % Runs all parttwo tests
cargo test partthree    % Runs all partthree tests
cargo test partfour     % Runs all partfour tests
```

To run a particular test case, such as `test_5` from `part1_1`, use:

```
cargo test partone::part1_1::test::test_5
```

The general format is:

```
cargo test <part_num>::part<part>::test::test_<test_num>
```

If you want to see your `print!()` statements, add the `-- --show-output` flag:

```
cargo test -- --show-output
cargo test partone::part1_1::test::test_5 -- --show-output
```

2 Converting Binary Strings to Decimal

This might look like a lot, but these functions are similar, with only a few differences.

2.1 Converting Binary to Unsigned Decimal

2.1.1 Function Definition

```
unsigned long binary_to_unsigned_decimal(const char *input_string);
```

In this part, you will implement a function that converts a binary string to its unsigned decimal value.

2.1.2 Arguments

- **input_string**: The binary string to be converted.

2.1.3 Approach

- Determine the number of bits in the `input_string`.
- Initialize a sum variable and set it to 0.
- Iterate through each bit in the string:
  - If the bit is 1, add $2^{numbits-1-i}$ to the sum.
  - If the bit is 0, do nothing.

2.1.4 Example

Let's say the input is `"11011"`:

- The binary string is 5 bits long.
- The first bit is 1, so we add $2^4 = 16$ to the sum.
- The next bit is 1, so we add $2^3 = 8$ to the sum.
- The next bit is 0, so we ignore it.
- The next bit is 1, so we add $2^1 = 2$ to the sum.
- The final bit is 1, so we add $2^0 = 1$ to the sum.

Calculation: $Sum = 16 + 8 + 2 + 1 = 27$
The function returns **27**.

2.1.5 Assumptions

- The maximum input will be 64 bits.
- The input format will be valid, containing only **0**s and **1**s.

2.2 Converting Binary to Signed Magnitude Decimal

In this part, you will convert a binary string into its signed magnitude decimal representation.

2.2.1 Function Definition

```
long binary_to_signed_magnitude_decimal(const char *input_string);
```

2.2.2 Arguments

- **input_string**: The binary string to be converted to a decimal value.

2.2.3 Explanation

The most significant bit (MSB) represents the sign of the number, while the remaining bits represent the magnitude.

Sign	Magnitude

2.2.4 Example

Let's convert 100011 to its decimal representation:

- The binary string is 6 bits long.
- The most significant bit (MSB) is **1**, indicating the value is negative.
- The remaining bits are 00011. Convert this to decimal: 00011 = 3.
- Since the MSB was 1, the final value is **-3**.

2.2.5 Assumptions

You can assume:

- The `input_string` will be at most 64 bits.
- The input will be valid and only contain **0**s and **1**s.
- **Hint:** This process is similar to the previous part, but you only consider the sign bit at the end.

2.3 Binary to One's Complement

In this part, you'll convert a binary string into its one's complement representation.

2.3.1 Function Definition

```
long binary_to_ones_complement_decimal(const char *input_string);
```

2.3.2 Arguments

- **input_string**: The binary string to be converted to its decimal equivalent.
- The function returns the converted value as a `long`.

2.3.3 Example

Let us convert 1110 to its one's complement representation.

- The most significant bit (MSB) is **1**, indicating the value is negative.
- Invert all the bits: $1110 \to 0001$.
- Convert the resulting binary string to decimal: $0001 = 1$.
- Since the original value was negative, the final result is **-1**.

2.3.4 Assumptions

You can assume:

- The input string will be a maximum of 64 bits.
- The input format will be valid, containing only **0** and **1**.

2.4 Binary to Two's Complement

Finally, you will convert a binary string into its twos complement representation.

2.4.1 Function Definition

```
long binary_to_twos_complement_decimal(const char *input_string);
```

2.4.2 Arguments

- **input_string**: The binary string to be converted to its decimal equivalent.
- The function returns the converted decimal value.

2.4.3 Example

Let us convert 11111110 into its two's complement representation.

- The binary string is 8 bits.
- The most significant bit (MSB) is **1**, indicating the value is negative. This contributes $-2^7 = -128$ to the sum.
- Iterating through the remaining bits and adding their contributions:
  - The 2nd MSB is 1: Add 64
  - The next bit is 1: Add 32
  - The next bit is 1: Add 16
  - The next bit is 1: Add 8
  - The next bit is 1: Add 4

    - The next bit is 1: Add 2
    - The final bit is 0: Add 0

- **Calculation:**

$$-128 + 64 + 32 + 16 + 8 + 4 + 2 = -128 + 126 = -2$$

- The decimal representation of 11111110 is **-2**.

2.4.4 Assumptions

You can assume:

- The input string will be 64 bits at max.
- The input format will be correct. There will only be 0 and 1 in the input.

2.5 Rules/hints

- You **cannot use strtoul**, or any similar functions.
- You should use strlen to get the number of bits your string has.

3 Converting Decimal to Binary

You converted Binary to decimal. Let us try switching things up.

3.1 Converting Decimal to Unsigned Binary

```
char* unsigned_decimal_to_binary(unsigned long input, unsigned int numbits)
```

3.1.1 Function Arguments

- **input**: The decimal number to be converted to Binary.
- **numbits**: The number of bits used to represent the binary number.

3.1.2 Example

Let's convert the decimal number 25 to Binary using a 5-bit representation.

- **Step 1: Find the Most Significant Bit (MSB)** Calculate $2^{numbits-1} = 2^4 = 16$.
    - Is $16 \leq 25$? Yes, so bit 4 is **1**. Subtract 16 from 25: $25 - 16 = 9$.
- **Step 2: Find the Next Bit** Calculate $2^3 = 8$.
    - Is $8 \leq 9$? Yes, so bit 3 is **1**. Subtract 8 from 9: $9 - 8 = 1$.

- **Step 3: Find the Next Bit** Calculate $2^2 = 4$.
  - Is $4 \leq 1$? No, so bit 2 is **0**.
- **Step 4: Find the Next Bit** Calculate $2^1 = 2$.
  - Is $2 \leq 1$? No, so bit 1 is **0**.
- **Step 5: Find the Least Significant Bit (LSB)** Calculate $2^0 = 1$.
  - Is $1 \leq 1$? Yes, so bit 0 is **1**. Subtract 1 from 1: $1 - 1 = 0$.
- **Result:** The binary representation 25 using 5 bits is **11001**.
- **Verification:**
  - $11001 = 16 + 8 + 1 = 25$. The calculation matches the original input.

3.2 Converting Decimal to Signed Magnitude

In this part, you'll be converting a decimal value into its signed magnitude representation.

```
char *signed_decimal_to_signed_magnitude(long input, unsigned int numbits);
```

3.2.1 Function Arguments

- input, the value you will be converting into signed magnitude.
- numbits, the number of bits used to represent the binary number.

3.2.2 Example

Let us say that we need to convert -30 into signed magnitude using 6-bits

- We can see that our input is negative, so we know the MSB is 1.
- We can convert 30 into Binary, using 5 bits with the same approach as the previous part.
  - $30 = 16 + 8 + 4 + 2 = 11110$, which is our magnitude
- We can finally put the sign and magnitude, giving us 111110

3.3 Converting Decimal to Ones Complement

```
char *signed_decimal_to_ones_complement(long input, unsigned int numbits);
```

3.3.1 Arguments

- input is the number we will be converting into one's complement.
- numbits is the number of bits used to represent the binary number.

3.3.2 Example

Let's say we want to convert -50 into ones complement using 7 bits.

- We can convert +50 as a 7-bit unsigned number:
  - 50 = 32 + 16 + 2 = 0110010
- Since 50 is negative, we flip the bits: 1001101

Hence, -50 in a 6-bit ones complement representation is 1001101.

3.3.3 Converting Decimal to Twos Complement

```
char *signed_decimal_to_twos_complement(long input, unsigned int numbits);
```

3.3.4 Arguments

- input: The number we will be converting to Binary.
- numbits: The number of bits representing the binary number.

3.3.5 example

let us convert -64 into two's complement using 8 bits.

- We know that the MSB is going to be $-2^{8-1} = -128$
  - Since our number is negative, we know bit 7 will be 1, and we can subtract -128 from -64, $-128 - 64 = 64$
- Now, we can convert 64 into Binary as we did before:
  - 64 in Binary is 1000000
- we add the MSB in front of 64, giving us 11000000
  - We can verify this by doing $11000000 = -128 + 64 = -64$.

3.4 Assumptions

- You can assume that the input number will fall within the 64-bit range for each representation.
- You can assume that the numbits argument will have enough bits for your representation.

4 Floating Point

In lecture, you learned about the IEEE 754 standard. In this section, you will convert a floating-point number (double) into its IEEE 754 representation. You will also handle varying bit-widths for the sign, exponent, and mantissa.

4.1 Converting IEEE 754 String into Decimal

```
double ieee754_to_decimal(char *input, int exp, int mantissa);
```

4.1.1 Arguments

- the input is the ieee754 you will be converting into its decimal representation.
- exp is the number of exponent bits your string will have.
- mantissa is the number of mantissa bits your string will have.
- You can assume there will always be **one** sign bit.

4.1.2 Example

Let us convert the IEEE 754 string `1100101` into its decimal value, assuming it uses a 3-bit exponent and a 3-bit mantissa.

- Let us first break our IEEE 754 string into three parts:
  - **Sign bit**: 1
  - **Exponent**: 100
  - **Mantissa**: 101

We can then plug these numbers into the following formula for IEEE 754 representation:

$$-1^1 \times 1.101 \times 2^E$$

- We need to figure out E by performing $exp - bias$:
  - The exponent (exp) is `100`, which is 4 in decimal.
  - The bias is calculated as $2^{k-1} - 1$, where k is the number of exponent bits. For a 3-bit exponent, the bias is:

$$2^{3-1} - 1 = 2^2 - 1 = 3$$

- We can calculate E as:

$$E = 4 - 3 = 1$$

Next, we must convert the mantissa (1.101) into its decimal value.

- There's a single 1 to the left of the decimal point, hence the value is 1.
- We now need to convert 0.101 into decimal:
  - Bit 1 is 1, so we add $2^{-1} = 0.5$ to 1: $1 + 0.5 = 1.5$

  - Bit 2 is 0, so we do not add anything.
  - Bit 3 is 1, so we add $2^{-3} = 0.125$ to 1.5: $1.5 + 0.125 = 1.625$

Now, we can plug all the values into the original formula:

$$-1^1 \times 1.625 \times 2^1 = -3.25$$

So, the IEEE 754 representation `1100101` equals -3.25 in decimal.

4.1.3 Denormalized and Special Case Values

- You will also have to handle denormalized and special case values.
  - The formula for denormalized values is $-1^s \times 0.M \times 2^{1-bias}$
- For Special Cases:
  - To return nan (**n**ot **a** **n**umber), you can use the NAN macro.
  - To return ∞, you can return INFINITY
  - To return $-\infty$, you can return -INFINITY

4.2 Decimal to IEEE 754 representation

```
char *decimal_to_ieee754_binary(double input, int exp_bits,
int mantissa_bits, int rounding_mode);
```

4.2.1 Arguments

- input is the decimal value we will be converting into IEEE 754.
- exp_bits specifies how many exponent bits we want in our representation.
- mantissa_bits specifies how many mantissa bits we want in our representation.
- rounding_mode specifies the rounding mode to use:
  - if 0 is passed in, you will always round up.
  - if 1 is passed in, you will always round down.
  - if 2 is passed in, you will **Round to even**

Three macros are included for your convenience:

```
#define ROUNDUP 0
#define ROUNDDOWN 1
#define ROUNDTOEVEN 2
```

4.2.2 Review

When working with base two notation, we can treat it similar to scientific notation:

$1.1 \times 2^0 = 0.11 \times 2^1 = 11 \times 2^{-1}$

We can also look at this from a base 10 perspective:

$1.5 \times 2^0 = 0.75 \times 2 = 3 \times 2^{-1}$

When, given a decimal number, x, we know that there is only a single 1 bit to the left of the point when $1.0 \leq |x| < 2$

We can also see, given the formula $x \times 2^E$, when we divide x by 2, we increment E by 1. When we multiply x by 2, we decrement E by 1.

4.2.3 Example

Let us convert -27 into its IEEE 754 representation. We will be representing this number using a 4-bit exp and 3-bit mantissa. We will also pass in 0, so we will always round up.

- We know this number is negative, so the sign bit will be 1; we can ignore the negative sign.
- Let us divide 27 by two until it is less than 2.0, and increment a variable E by one every time:
  - $\frac{27}{2} = 13.5$, E = 1
  - $\frac{13.5}{2} = 6.75$, E = 2
  - $\frac{6.75}{2} = 3.375$, E = 3
  - $\frac{3.375}{2} = 1.6875$, E = 4
  - We found a quotient less than 2.0, so we can express $27 = 1.6875 \times 2^4$
- If the number is less than one, we multiply it by 2 and decrement E until the number is greater than or equal to 1.0. We do not need to worry about that.
- We can then convert 0.6875 into fractional Binary (you ignore the 1 to the left of the decimal; it is implied for normalized values.)
  - You know that the input argument is a double; this means the mantissa is 52 bits so you can do the following steps 52 times:
    * multiply the number by two.
    * if the $result \geq 1.0$, set the bit to 1 and subtract one from the product, otherwise set it to 0
- Let us do this for 0.6875:
  - $0.6875 \times 2 = 1.375$
  - $0.375 \times 2 = 0.75$

- $0.75 \times 2 = 1.5$
- $0.5 \times 2 = 1.0$. Since we reached exactly one, that means we found the fractional Binary. For simplicity, you can iterate through this 52 times.

- We can finally say 0.6875 is 0.1011 is fractional Binary. Adding the 1 gives us 1.1011.
- We need to find the exponent section:
  - exp = E + bias
    - We found E to be 4
    - The bias is $2^{k-1}-1$, where k is the number of exp bits. $2^{4-1}-1 = 7$
    - exp = 4 + 7 = 11
    - We can convert this into binary, 1011
- We can put our mantissa, exp, and sign together: 110111011, which is an IEEE 754 representation but not what we want
  - Remember, we want a representation with a 4-bit exponent and a 3-bit mantissa. or representation has a 4-bit exp and 4-bit mantissa: 1 1011 1011
  - We need to round our number. As specified in the argument, we'll round up. Which is done by adding 1 to the mantissa:
    - 1 1011 1011 $\rightarrow$ 1 1011 110
- hence, our final ieee754 representation is 11011110

4.2.4 Rounding

We utilize rounding when the number we want to represent requires more bits than provided. There are three main rounding modes:

- Round-Up
- Round Down
- Round to Even

Let us say we want to represent a number, -14.75, using ieee754 with a 3-bit exponent, 3-bit mantissa, and the sign bit, which we can calculate is 1 110 110110 in it is ieee 754 representation. There's a problem with the representation having six mantissa bits, but our representation can only store 3 bits.

- **Round up** tells us to always round up, which is to add 1 and truncate the unnecessary bits.

  - We round up by adding 1 and truncated the remaining bits, giving us 1 110 111

- **Round down** tells us always to round down, which is to set the last bit to 0 and truncate the remaining bits:
  - Rounding down would give us 1 110 110
- **Round to even** tells us that we round based on whether we are the number above, below, or exactly halfway.
  - In other words, we want the last bit of our representation to be 0.

Round to Even

- Round to Even, unlike Roundup and Round down, requires you to look at the bits when deciding if you want to round up or down.

We can assign the bits as follows:

- Guard bit, the LSB of our IEEE754 representation.
- Round bit, first bit removed.
- Sticky bit, rest of the bits (if one of the extra bits is 1).

```
1110110110
SEEEMMM
GRSS
```

- We can see that the round bit is 1, which tells us we're in the middle.
- We look at the Sticky bits, there is a 1.
- Regardless of what G is, we round up, giving us 1 110 111

Let us look at an example when we would round down:

```
1110110010
SEEEMMM
GRSS
```

- Here, we can see that the Round bit is zero, so we are not halfway.
- Regardless of what the stick bits are, we round down to 1 110 110

Rounding to even:

```
1110110100
SEEEMMM
GRSS
```

- The Round bit is 1, which tells us we are in the middle
- The sticky bits are 0; this tells us we are exactly halfway, so we cannot decide whether to round up or down.
- In this case, we look at the Guard bit.
  - If it is 0, we round down.
  - If it is 1, we round up.
- Since the Guard bit is 0, we round down, giving us 1 110 110

5 Submission

5.1 Submitting your assignment

You'll just be submitting your pa2.c file on Canvas.

5.2 Compiling + Testing

Given testcases will make up 70% of the final grade. The other 30% will be randomized. The Autograder, for the fifth test on each part, tests your code with random inputs.

5.3 Help

- Office Hours
- Lecture/Recitation
- CSL/CAVE
- RLC