Programming Assignment - Extra Credit

Rutgers CS211: Fall 2024

Programs must run on iLab machines, and your code must strictly follow input-output guidelines in each section. Please see the CS department's academic integrity policy at: http://nbacademicintegrity.rutgers.edu

Circuit Description

One of the inputs to your program will be a circuit description file that will describe a circuit using various directives. We will now describe the various directives. The input variables used in the circuit are provided using the INPUTVAR directive. The INPUTVAR directive is followed by the number of input variables and the names of the input variables. All the input variables will be named with capitalized identifiers. An identifier consists of at least one character (A-Z) followed by a series of zero or many characters (A-Z) or digits (0-9). For example, some identifiers are IN1 and IN2. An example specification of the inputs for a circuit with two input variables: IN1, IN2 is as follows:

```
INPUTVAR 2 IN1 IN2
```

The outputs produced by the circuit is specified using the OUTPUTVAR directive. The OUTPUTVAR directive is followed by the number of outputs and the names of the outputs. An example specification of the circuit with two outputs OUT1 and OUT2 is as follows: `OUTPUTVAR 2 OUT1 OUT2` The output values produced by the circuit is specified using the OUTPUTVAL directive. The OUTPUTVAL directive describes the output values of the circuit each on its own subsequent line. Each line begins with the name of the OUTPUTVAR followed by the values {0, 1} of the variable for each permutation of the input. OUTPUTVAL will be followed by the same number of lines as their are variables as described in OUTPUTVAR and they will be described in the same order. An example specification for the OUTPUTVAL directive with the OUTPUTVAR described above is as follows:

```
OUTPUTVAL
OUT1 0 0 0 1
OUT2 1 0 1 0
```

In this example, we can deduce that there are 2 inputs. As the input is binary, there are 4 permutations of these inputs {0, 0}, {0, 1}, {1, 0}, and {1, 1}. For these permutations, OUT1 has the values 0, 0, 0, 1 and OUT2 has the values 1, 0, 1, 0, respectively. The order of these permutations will be detailed later. Logic gates are the basic building blocks of digital systems and circuits. A circuit is one or more logic gates creating a logical relationship between the input and output. The circuits used in this assignment will be using the following building blocks: OR, AND, XOR, NOT, DECODER, and MULTIPLEXER. The specifications of each building block is as follows:

- OR: This directive represents the or gate in logic design. The directive is followed by the names of the two inputs and the name of the output. An example circuit for an OR gate (OUT1 = IN1 + IN2) is as follows:

```
OR IN1 IN2 OUT1
```

- AND: This directive represents the and gate in logic design. The directive is followed by the names of the two inputs and the name of the output. An example circuit for an AND gate (OUT1 = IN1.IN2) is as follows:

```
AND IN1 IN2 OUT1
```

- XOR: This directive represents the xor gate in logic design. The directive is followed by the names of the two inputs and the name of the output. An example circuit for an XOR gate ($OUT1 = IN1 \oplus IN2$) is as follows:

```
XOR IN1 IN2 OUT1
```

- NOT: This directive represents the not gate in logic design. The directive is followed by the name of the input and the name of the output. An example circuit for a NOT gate (OUT1 = TN1) is as follows:

```
NOT IN1 OUT1
```

- DECODER: This directive represents the decoder in logic design. The directive is followed by the number of inputs, names of the inputs, and the names of the outputs. An example decoder with two inputs IN1 and IN2 is specified as follows:

```
DECODER 2 IN1 IN2 OUT1 DUT2 OUT3 OUT4
```

Here, OUT1 represents the $\overline{IN1}.\overline{IN2}$ output of the decoder, OUT2 represents the $\overline{IN1}.IN2$ output of the decoder, OUT3 represents the IN1.$\overline{IN2}$ output of the decoder, OUT4 represents the IN1.IN2 output of the decoder. Note that the output of the decoder (i.e, OUT1, OUT2, OUT3, and OUT4) are based on unsigned binary value. They will be in gray code sequence beginning in the second part of this assignment.

- Multiplexer: This directive represents the multiplexer in logic design. The directive is followed by the number of inputs, names of the inputs, names of the selectors and the name of the output. An example of a 4:1 multiplexer is specified as follows:

```
MULTIPLEXER 4 0 0 1 0 IN1 IN2 OUT1
```

The above description states that there are 4 inputs to the multiplexer. The four inputs to the multiplexer are 0 0 1 0 respectively. The two selector input signals are IN1 and IN2. The name of the output is OUT1.

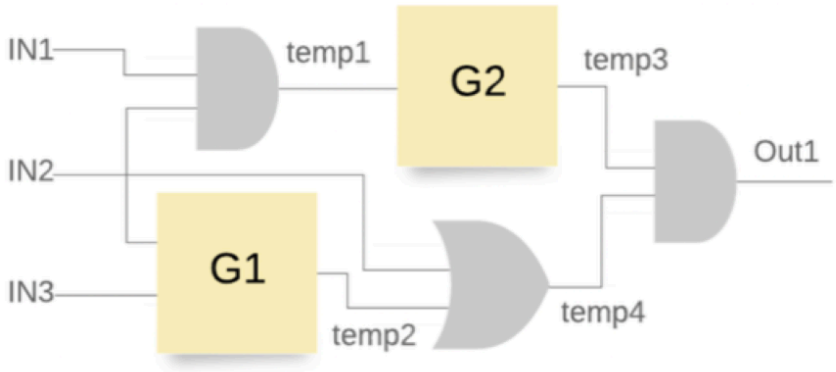
Describing Circuits Using these Directives

It is possible to describe any combinational circuit using the above set of directives. For example, the circuit OUT1 = IN1.IN2 + IN1.IN3 can be described as follows:

```
INPUTVAR 3 IN1 IN2 IN3
OUTPUTVAR 1 OUT1
OUTPUTVAL
OUT1 0 0 0 0 0 1 1 1
AND IN1 IN2 temp1
AND IN1 IN3 temp2
OR temp1 temp2 OUT1
```

Note that OUT1 is the output variable. IN1, IN2, and IN3 are input variables. Here, temp1 and temp2 are temporary variables. In the assignment, some gates will be described as unknown or variable gates. They will always be named in the format $Gnmx_1, x_2, ...x_m$ where n and m are both integers. n will always be the number of the variable gate in which it appears in the input i.e. starting at 1 and incrementing by 1 per variable gate. Here, m will be the number of variables, including inputs and outputs, for the gate. Lastly $x_1, x_2, ...x_m$ will be the name of the variables of the gate. Here is an example of this:

```
INPUTVAR 2 IN1 TH2
OUTPUTVAR 1 OUTI
OUTPUTVAL.
OUT1 0 0 1 1
G1 2 IN1 temp1
AND IN1 IN2 temp2
MULTIPLEXER 4 1 1 1 0 0 temp2 temp1 OUT1
```



As seen above, a circuit description is a sequence of directives. G1 is an unknown gate which seemingly takes one input IN1 and output temp1. If every temporary variable occurs as an output variable in the sequence before occurring as an input variable, we say that the circuit description is sorted. You can assume that the

circuit description files will all be sorted. Note: A temporary variable can occur as an output variable in at most one directive.

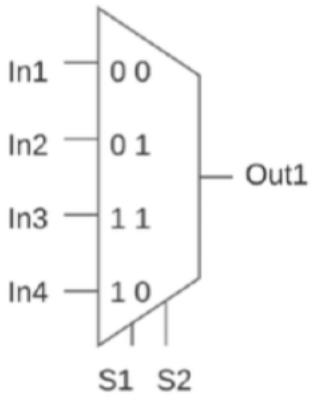
Part I: Circuit Completer

You will have to write a program that will complete a circuit by choosing the gates that fill the circuit and fulfill the constraints. In this section, input permutations and the ordering for decoders and multiplexers will be in standard binary order. In this section, we will choose gates based on a DFS strategy where each gate will be temporarily designated as a logic gate, chosen in the order of the gates (OR, AND, XOR, NOT, DECODER, MULTIPLEXER). Using this strategy, circuits with multiple solutions should still return one solution with the gates chosen in the DFS fashion.

For example, say that the circuit description file contains the following:

```
INPUTVAR 3 IN1 IN2 IN3
OUTPUTVAR 1 Out1
OUTPUTVAL
Out1 0 1 1 1 0 1 0 0
AND IN1 IN2 temp1
G1 3 IN3 IN2 temp2
G2 2 temp1 temp3
OR temp2 IN2 temp4
AND temp3 temp4 Out1
```

The circuit described by these directives is shown visually in Figure 1. G1 will first be designated as an OR gate then G2 will be designated as an OR gate. If the output of the circuit matches OUTPUTVAL then the output will be two OR gates. If they do not match, then G2 will be sequentially designated as the other gates until a match is found. If no match is found then G1 will be designated as an AND gate. G2 will be reset to an OR gate and the algorithm will repeat



Input format: Your program will take the file name as input. The file will contain the description of a circuit using the directives described above.

Output format: Your output will be the name of the variable gate followed by its assigned type, listed one per line, that complete the circuit. The gates must be in the same order as they were given. If no permutations of logic gates can fulfill the circuit then the program must print: **INVALID**. For the example in Figure 1, the output would be:

```
G1 OR
G2 NOT
```

Part II: Circuit Completer for Gray Code Inputs

For this section, you will have to write a program that will complete a circuit by choosing the gates that fill the circuit and fulfill the constraints as in the first part. However, in this section input permutations and the ordering of multiplexer and decoders will be in Gray Code. We will use Reflective Gray Code to create the permutations. You must still use the DFS methodology for choosing gates as in the first part. An example of a multiplexer utilizing grey code ordering is visualized in Figure 2.

Input format: Your program will take the file name as input. The file will contain the description of a circuit using the directives described above.

Output format: Your output will be the name of the variable gate followed by its assigned type, listed one per line, that complete the circuit. The gates must be in the same order as they were given. If no permutations of logic gates can fulfill the circuit then the program must print: **INVALID**.

Part III - Circuit Reducer

In this part, you have to determine whether a circuit can be reduced to fewer logic gates. For example, for the solution in Figure 1, G1 is an OR gate. This implies that part of the circuit is:

```
IN2 OR (IN2 OR IN3)
```

This can be simplified to simply IN2 OR IN3. This is an example of the Associative Law and a basic rule of Boolean Algebra where anything ORed with itself is equal to itself. We will simplify circuits based on 2 combinations of boolean algebra laws and basic rules of boolean algebra. They are listed below.

- **Associative law** and OR rule: The Associativity rule states that when ORing more than two variables, the result is the same regardless of the grouping of the variables. Again, a basic rule of Boolean Algebra is that anything ORed with itself is equal to itself. Thus, a sequence of two OR gates with the same variable in both can be reduced to just one OR gate. An example of this is that the directives:

```
OR IN1 IN2 temp1
OR IN1 temp1 temp2
```

can be reduced to

```
OR IN1 IN2 temp2
```

For simplicity, we will only include cases such as this example where the pair of gates to be reduced will have the output of gate as the input to the other.

- **Distributivity rule**: The distributivity rule is the factoring law. This states that a common variable can be factored from an expression just as in ordinary algebra. Thus, a sequence of two AND gates with the same variable in both followed by an OR gate can be reduced to just one AND and one OR gate.

An example of this is that the directives:

```
AND IN1 IN2 temp1
AND IN1 IN3 temp2
OR temp1 temp2 temp3
```

can be reduced to

```
OR IN2 IN3 temp2
AND IN1 temp2 temp3
```

Another example of this is that the directives:

```
OR IN1 IN2 temp1
OR IN1 IN3 temp2
AND temp1 temp2 temp3
```

can be reduced to

```
AND IN2 IN3 temp2
OR IN1 temp2 temp3
```

As seen in the previous examples, when reducing gates, the resulting gates should output the variables used by the last-most gates. i.e. when reducing 2 gates that output temp1 and temp2 respectively, the output of the single reduced gate should be temp2. The input variables of the gates that resulted from a reduction should be sorted in lexicographical order. An example of these properties is given under the Output format section. In this section we will continue to use gray code ordering so your implementation for third should be used.

Input format: Your program will take the file name as input. The file will contain the description of a circuit using the directives described above. The directives may still contain the variables gates as in first and second.

Output format: Your output will be the list of reduced directives that represent the solved original circuit described in the input. If no permutations of logic gates can fulfill the circuit then the program must print: INVALID. The reduced gates should take the place of the last-most gates that they replaced in the sequence. Namely, if two gates replaced three gates, the first new gate should take the place of the second original gate and the second new gate should take the place of the third original gate in the output sequence.

For example, for the example in Figure 1 the reduced output would be

```
INPUTVAR 3 IN1 IN2 IN3
OUTPUTVAR 1 Out1
OUTPUTVAL
Out1 0 1 1 1 0 1 0 0
AND IN1 IN2 temp1
NOT temp1 temp3
OR IN2 IN3 temp4
AND temp3 temp4 Out1
```

Part IV: Conversion to Unsigned Binary Representation

In this part, your task is to write a C program that prints the unsigned binary representation of a number with a specific number of bits. The argument to the program is an input file, whose format is described in the input format. If a given number is representable with a given number of bits (w), then you should print the binary representation of the number. Otherwise, you print the binary representation of the remainder when divided by 2^w.

Input-Output format: Your program will take one file name as its command-line input. Each line inthe input file will have two positive integers separated by a space: an integer that you want to represent in binary and the number of

bits to use for the representation. For each line in the input, you should print out the binary representation of the number followed by a newline character.

Example Execution

Let's assume we have the following input file:

```
input.txt
42 12
27 3
```

Then the result will be:

```
$ ./third input.txt
000000101010
011
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

Part V: Effect of Signed/Unsigned Casts

In this part, your task is to write a C program that prints the value of the number when it is cast from an unsigned number to a signed number and vice-versa. The argument to the program is an input file, whose format is described in the input format.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have two integers and two characters separated by a space: an integer that is being represented, the number of bits to use for the representation, the source representation, and the destination representation. Here, u represents unsigned representation and s represents signed representation. For each line in the input, you should print out the value of the number (in decimal) in the destination representation followed by a newline character. You can assume that the unsigned value of the number is representable with the given number of bits.

Example Execution

Let's assume we have the following input file:

```
input.txt
7 3 u s
-2 4 s u
```

Then the result will be:

```
$ ./fourth input.txt
-1
14
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

Part VI: Decimal Fraction to Canonical Binary Fraction

You will write a program to convert a decimal fraction to a binary fraction in the canonical representation (i.e., $(-1)^s \times 1.M \times 2^E$).

For this program:

- (M) lies between ([1, 2)).
- You do not have to perform any rounding for this part.

You are required to print as many digits after the decimal point as specified by the input.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have a decimal fraction (use a double type to read it) and the number of bits to show in the canonical binary representation separated by a space. For each line in the input, you should print the M value and E value in the canonical representation separated by space. Add a newline character after printing the output for each input.

Example Execution:

Let's assume we have the following input file:

```
input.txt
6.25 3
12.53 4
```

Then the result will be:

```
$ ./first
1.100 2
1.1001 3
```

explanation

- 6.25 in fractional binary is $4 + 2 + \frac{1}{2^2} = 110.01$
- We shift the decimal to the left by two bits, so we have 1.1001×2^2.
- Since we only print 3 digits, we print "1.100 2", where 2 is E.

Part VII - Decimal to IEEE-FP with Rounding

Your task is to write a program to convert a decimal fraction to IEEE-754 FP representation in a given configuration with the rounding to nearest with ties-to-even rounding mode.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have a decimal fraction (use a double

type to read it), the number of the bits (n) in the IEEE-754 FP representation, number of bits for the exponent, and number of fraction bits. These numbers on a given line are separated by a space. For each line in the input, you should the IEEE-754 representation with n-bits followed by a new line.

Example Execution:

let's assume we have the following input file:

```
input.txt
7.5 32 8 23
0.125 6 3 2
0.1875 6 3 2
```

Here, we want to convert 7.5 to a 32 bit IEEE754 FP representation with an 8 bit exponent and 23 bit mantissa, and the 1 bit sign (obviously).

The expected output for this program is:

```
01000000111100000000000000000000
000010
000011
```

Explanation: let's walk through 7.5, and how we got our answer:

- First, we convert our solution to fractional binary: $7.5 = 111.1$
- Next, we shift the decimal point to the left until there's only one 1:
  - $111.1 \rightarrow 1.111 \times 2^2$.
    - Here, M is 111, and E is 2
- Next, we need to figure out the exponent:
  - We need to add a bias to E. $Bias = 2^{k-1} - 1$, where k is the number of exponent bits.
  - Exponent = 2 + 127 = 129
    - $Bias = 2^{8-1} - 1 = 2^7 - 1 = 127$
  - 129 in binary is 10000001.
- Finally, we put the pieces together
  - 7.5 is a positive number, so the sign bit is 0.
  - Our exponent is 10000001
  - our mantissa is 111000…
- Hence, our final answer is 01000000111100000000000000000000

round to even

- You're expected to implement round to even:

let's assume we're rounding to the nearest whole number in the table below

number	round
1.4	1.0
1.6	2.0
1.5	2.0
2.5	2.0

- If we're "in the middle", we round to the nearest even number.

assumptions

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above. You can assume that input will not have NaNs and any value will not round up or down to infinities

Part VIII: Hexadecimal Bit-pattern in the IEEE-FP format to decimal fraction

Your task is to write a program that takes a hexadecimal bit-pattern and prints the decimal fractional value of the number.

Input-Output format: Your program will take one file name as its command-line input. Each line in the input file will have the total number of bits, the number of bits for the exponent, the number of bits for the fraction, the hexadecimal bit-pattern, and the number of precision bits after the decimal point in the decimal fraction. These numbers on a given line are separated by a space. For each line in the input, you should print out the decimal fraction value with the specified number of precision bits followed by a new line.

example execution

Example Execution: Let's assume we have the following input file:

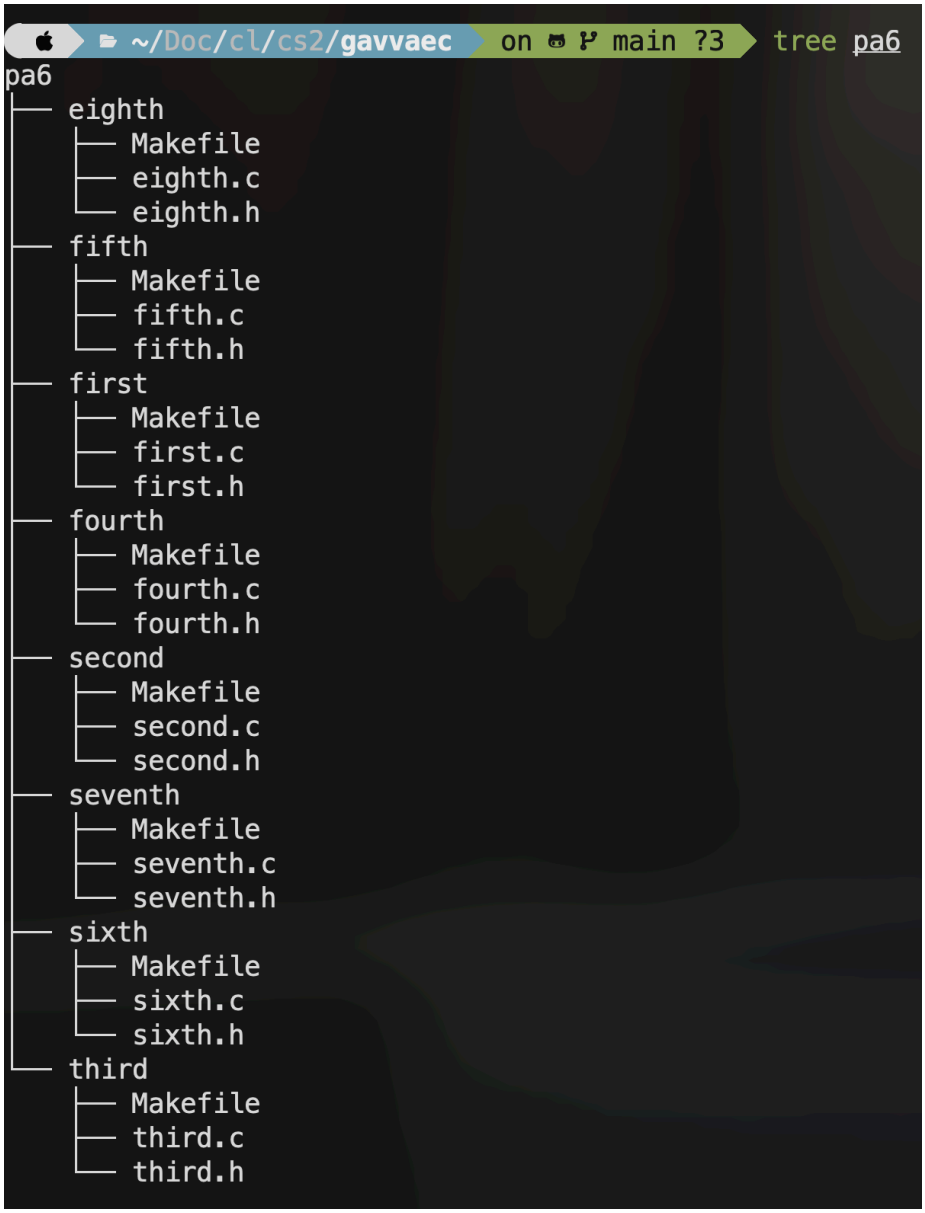
```
input.txt
8 4 3 0x4d 2
8 4 3 0x16 7
```

Then the result will be:

```
$./first input.txt
6.50
.0546875
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format, as stated above.

Submission



- Compress the pa6 folder in a tar file named `pa6.tar`
- to do this, you can run the command: `tar -cvf pa6.tar pa6`
- to extract (test it), you can do `tar -xvf pa6.tar`

Grading Guidelines

This is a large class, so the most significant part of your grade will be based on programmatic checking of your program. That is, we will build the binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should not see or use your friend's code either partially or fully. We will run state-of-the-art plagiarism detectors. We will report everything caught by the tool to the Office of Student Conduct.
- You should make sure that we can build your program by just running make.
- Your compilation command with gcc should include the following flags: `-fsanitize=address,undefined -Og -g -std=c11 -Wall -Werror`
- You should test your code as thoroughly as you can. For example, programs should not crash with memory errors.
- Your program should produce output following the example format shown in previous sections. Any variation in the output format can result in up to 100% penalty. Be especially careful not to add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.
- Your folder names in the path should not have any spaces.