# CS 211: Computer Architecture, Fall 2020
# Extra Credit: Circuit Simulator (100 points)

Instructor: Prof. Santosh Nagakaratte

Due: Dec 12, 2020 at 5:00pm (EST).

## 1 Assignment Introduction

This assignment is designed to give you a better understanding of logic gates and circuits. Your program must follow the input-output guidelines listed in each section **exactly**, with no additional or missing output.

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

No cheating or copying will be tolerated in this class. Your assignments will be automatically checked with plagiarism detection tools that are pretty powerful. Hence, you should not look at your friend's code. See Rutgers academic integrity policy at:
`http://academicintegrity.rutgers.edu/`

## 2 Logic Gate Introduction

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 output OUT1 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}, {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 names of the two inputs and the name of the output.

  An example circuit for a NOT gate ($OUT1 = \overline{IN1}$) 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 OUT2 OUT3 OUT4
  ```

  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 outputs 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.

## 3 Describing Circuits using the 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. 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 $G\,n\,m\,x_1, x_2, ... x_m$ 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. $m$ 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 IN2
OUTPUTVAR 1 OUT1
```

```
OUTPUTVAL
OUT1 0 0 1 1
G1 2 IN1 temp1
AND IN1 IN2 temp2
MULTIPLEXER 4 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 inputs IN1 and outputs 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.

## 4 First: Circuit Completer (30 Points)

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 described in section 2 (OR, AND, XOR, NOT, DECODER, MULTIPLEXER). Using this strategy, circuits with multiple solutions should still return one solution with the gates chosen in 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 visualized in Figure 1 below.

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
```

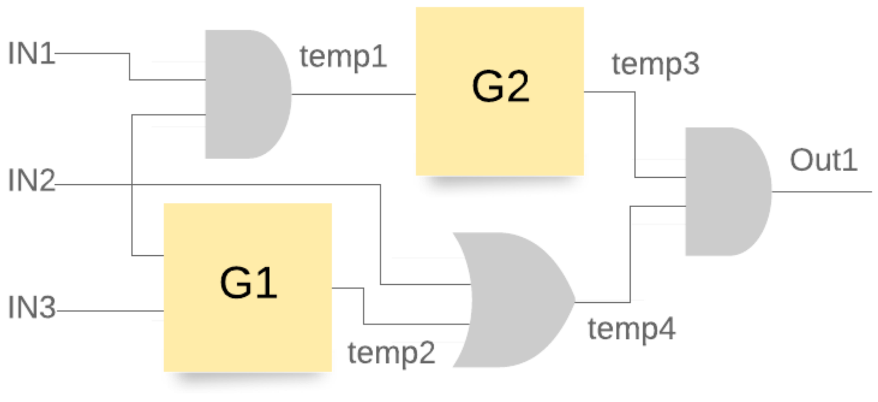

Figure 1: Visualization of a circuit with two unknown gates, G1 and G2

## 5 Second: Circuit Completer - Gray Code (10 points)

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 first. 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 first.

An example of a multiplexer utilizing grey code ordering is visualized in figure 2.

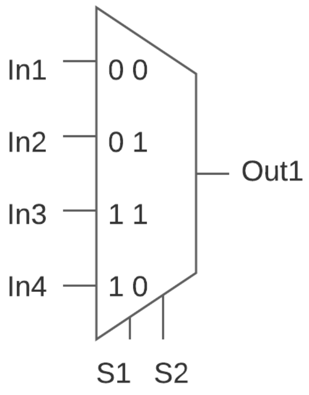

Figure 2: A visualization of a 4:1 multiplexer utilizing Gray Code ordering.

**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**.

## 6 Third: Circuit Completer - Optimized (25 points)

In this section, you must optimize your implementation of second where permutations are in Gray Code order. Namely, rather than sampling the possible gates using DFS, you must come up with a better strategy that reduces the number of sampled gates and/or run time. Your choice of strategy or combination of multiple strategies is left up to you. Describe your methodology, reasoning, and the way in which you implemented this, in at most two well written paragraphs, in a text file called **optimized.txt** which should be included in your third directory. In order to get full credit on this portion you must include this text file. You must also make a meaningful and beneficial change(s), i.e. changing DFS to BFS is not a valid approach.

**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**.

## 7 Fourth: Circuit Reducer (35 points)

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 Associative Law 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.

- **Distributive Law**: The Distributive Law 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 outputted 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
```

## Structure of your submission folder

All files must be included in the `pa6` folder. The `pa6` directory in your tar file must contain 4 subdirectories, one each for each of the parts. The name of the directories should be named first through fourth (in lower case). Each directory should contain a c source file, a header file (if you use it) and a Makefile. For example, the subdirectory first will contain, first.c, first.h (if you create one) and Makefile (the names are case sensitive). The third directory should also include your optimized.txt.

```
pa6
|- first
   |-- first.c
   |-- first.h (if used)
   |-- Makefile
|- second
   |-- second.c
   |-- second.h (if used)
   |-- Makefile
|- third
   |-- third.c
   |-- third.h (if used)
   |-- Makefile
   |-- optimized.txt
 |- fourth
   |-- fourth.c
   |-- fourth.h (if used)
   |-- Makefile
```

## Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa6.tar`. To create this file, put everything that you are submitting into a directory (folder) named `pa6`. Then, `cd` into the directory containing `pa6` (that is, `pa6`'s parent directory) and run the following command:

```
tar cvf pa6.tar pa6
```

To check that you have correctly created the tar file, you should copy it (`pa6.tar`) into an empty directory and run the following command:

```
tar xvf pa6.tar
```

This should create a directory named `pa6` in the (previously) empty directory.

The `pa6` directory in your tar file must contain 4 subdirectories, one each for each of the parts. The name of the directories should be named first through fourth (in lower case). Each directory should contain a c source file, a header file (if necessary) and a make file. For example, the subdirectory first will contain, first.c, first.h and Makefile (the names are case sensitive).

## AutoGrader

We provide the AutoGrader to test your assignment. AutoGrader is provided as autograder.tar. Executing the following command will create the autograder folder.

```
$tar xvf autograder.tar
```

There are two modes available for testing your assignment with the AutoGrader.

### First mode

Testing when you are writing code with a `pa6` folder

(1) Lets say you have a `pa6` folder with the directory structure as described in the assignment.

(2) Copy the folder to the directory of the autograder

(3) Run the autograder with the following command

$python auto_grader.py

It will run your programs and print your scores.

### Second mode

This mode is to test your final submission (i.e, pa6.tar)

(1) Copy pa6.tar to the auto_grader directory

(2) Run the auto_grader with pa6.tar as the argument.

The command line is

$python auto_grader.py pa6.tar

### Scoring

The autograder will print out information about the compilation and the testing process. At the end, if your assignment is completely correct, the score will something similar to what is given below.

```
You scored
5.0  in  second
17.5 in  fourth
12.5  in  third
15.0  in  first

Your TOTAL SCORE =  50.0 /50
Your assignment will be graded for another 50 points with test cases not given to you
```

## Grading Guidelines

This is a large class so that necessarily 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 Office of Student Conduct**.
- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can. For example, programs should *not* crash with memory errors.
- Your program should produce the 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 to not add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.

Be careful to follow all instructions. If something doesn't seem right, ask on discussion forum.