CS211 Fall 2018
Programming Assignment IV

David Menendez

Due: Wednesday, December 5, 2018
Submit by December 6 at 3:00 AM

This assignment will provide more practice programming in C and working with circuits and digital logic. It has two parts. In the first part, you will design several circuits using a simple specification language. In the second part, you will write a program that generates a truth table given such a circuit specification.

Advice Start working early. Before you do any coding, make sure you can run the auto-grader, create a Tar archive, and submit that archive to Sakai. You don't want to discover on the last day that `scp` doesn't work on your personal machine.

Decide your data structures and algorithms first. Section 4 describes one method for implementing the program that performs well, but you are free to design your own. Writing out pseudocode is not required, but it may be a good idea.

Start working early. You will almost certainly encounter problems you did not anticipate while writing this project. It's much better if you find them in the first week.

1 Specification Language

In this language, circuits are specified with a series of *directives*. These directives refer to various named *variables*, which correspond to wires in a circuit diagram. Many of the directives describe a logic gate or similar building block, indicating which varibles correspond to its input and output.

Each directive has the form of a *name* followed by one or more *parameters*, separated by whitespace. The name indicates which directive and determines the number of parameters. Some directives take a variable number of parameters; their first parameter will always be an integer which is used to determine the number of parameters. Depending on the directive, some parameters will be inputs and some will be outputs.

Variables in a circuit can be classified into three non-overlapping sets. *Input variables* must be declared by the `INPUT` directive, and may only occur as input parameters. *Output variables* must be declared by the `OUTPUT` directive and may occur exactly once in an output parameter. All other variables are *temporary variables* and must occur exactly once as an output parameter and zero or more times as an input parameter.

A variable name consists of a letter followed by zero or more letters or digits. You may assume that variable names are no longer than 16 characters.

In addition to variables, the constant inputs `0` and `1` may be used as input parameters. These are always false and always true, respectively.

Finally, directives may allow an optional colon (`:`) to occur between certain parameters. The presence or absence of this colon has no effect on semantics, but may make the specification more readable.

1.1 Directives

This section describes each of the directives used to describe a circuit. Each directive is followed by several parameters. A parameter n is always an integer and has a special meaning. Input parameters are indicated as i and output parameters are indicated as o. Ellipses ($\cdots$) are used to indicate a variable number of parameters. A colon (`:`) indicates an optional colon.

- `INPUT` n `:` $i_1 \cdots i_n$
  Declares n input variables. This directive must always occur first in a circuit description.

- `OUTPUT` n `:` $o_1 \cdots o_n$
  Declares n output variables. This directive must always occur second in a circuit description.

- `NOT` i `:` o
  Represents a *not* gate in logic design. Computes $o = \bar{i}$.

- `AND` $i_1\, i_2$ `:` o
  Represents an *and* gate in logic design. Computes $o = i_1 i_2$.

- `OR` $i_1\, i_2$ `:` o
  Represents an *or* gate in logic design. Computes $o = i_1 + i_2$.

- `NAND` $i_1\, i_2$ `:` o
  Represents a *nand* gate in logic design. Computes $o = \overline{i_1 i_2}$

- `NOR` $i_1\, i_2$ `:` o
  Represents a *nor* gate in logic design. Computes $o = \overline{i_1 + i_2}$

- `XOR` $i_1\, i_2$ `:` o
  Represents an *xor* gate in logic design. Computes $o = i_1 \oplus i_2$, where $\oplus$ indicates *exclusive or*.

- `DECODER` n `:` $i_1 \cdots i_n$ `:` $o_0 \cdots o_{2^n-1}$
  Represents an $n : 2^n$ *decoder* gate in logic design. The first argument gives the number of inputs, n. The next n parameters are the inputs, followed by 2^n parameters indicating the outputs.

  The inputs are interpreted as an n-bit binary number s in the range $0, \cdots, 2^n - 1$. The output o_s will be 1 and all others will be 0.

- `MULTIPLEXER` n `:` $i_0 \cdots i_{2^n-1}$ `:` $i'_1 \cdots i'_n$ `:` o
  Represents a $2^n : 1$ *multiplexer* gate in logic design. The inputs to a multiplexer are either regular inputs or selectors, indicated by i and i', respectively. The first parameter, n, gives the number of selectors. The next 2^n parameters give the regular inputs, followed by n selector inputs, and finally the output.

The selector inputs are interpreted as an n-bit binary number s in the range $0, \cdots, 2^n - 1$. The output is $o = i_s$.

- `PASS` i `:` o
  Represents the absence of a gate. Computes $o = i$. This may be used to convert a temporary variable into an output variable.

1.2 Examples

This circuit describes a half-adder, where s is the sum and c is the carry.

```
INPUT 2 A B
OUTPUT 2 C S
AND A B : C
XOR A B : S
```

This circuit computes $z = ab + ac$:

```
INPUT 3 a b c
OUTPUT 1 z
AND a b x
AND a c y
OR x y z
```

Note that x and y are temporary variables, since they were not declared in `INPUT` or `OUTPUT`.
This circuit description is invalid, becuase it uses an output variable as an input parameter:

```
INPUT 3 IN1 IN2 IN3
OUTPUT 2 OUT1 OUT2
AND IN1 IN2 : OUT1
OR IN3 OUT1 : OUT2
```

This can be rewritten using `PASS`:

```
INPUT 3 IN1 IN2 IN3
OUTPUT 2 OUT1 OUT2
AND IN1 IN2 : temp1
PASS temp1 : OUT1
OR IN3 temp1 : OUT2
```

This circuit demonstrates the user of `MULTIPLEXER`:

```
INPUT 3 A B C
OUTPUT 1 Z
MULTIPLEXER 3 : 0 0 0 1 1 0 1 1 : A B C : Z
```

As shown in class, this can be re-written to use a 4:1 multiplexer:

```
INPUT 3 A B C
OUTPUT 1 Z
NOT C NC
MULTIPLEXER 2 : 0 C NC 1 : A B : Z
```

2 Circuit Design (100 points)

For this part of the assignment, you will design circuits using the language described in section 1. These circuits will be delivered as part of your Tar archive, as described in section 6.5.

Your circuits must describe the correct truth tables. If explicit lists of input and output variables are given, you must use them as written.

1. A circuit that produces the following truth table:

0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

2. A (different) circuit using only `NAND` that produces the same truth table as above.

3. A two-bit full adder.

```
INPUT 5 X2 X1 Y2 Y1 Cin
OUTPUT 3 Cout Z2 Z1
```

This circuit should compute $C_{out}Z_2Z_1 = X_2X_1 + Y_2Y_1 + C_{in}$, where $C_{out}Z_2Z_1$ is a three-bit binary number, and + represents addition.

4. A 2:4 decoder using `AND`, `OR`, and `NOT`.

```
INPUT 2 I2 I1
OUTPUT 4 O0 O1 O2 O3
```

$O_n = 1$ if and only if $I_2I_1 = n$, where I_2I_1 is a two-bit binary number.

5. An 8:1 multiplexer. You may use multiplexers, but not an 8:1 multiplexer.

```
INPUT 11 I0 I1 I2 I3 I4 I5 I6 I7 S4 S2 S1
OUTPUT 1 O
```

$O = I_n$ if $S_4S_2S_1 = n$, where $S_4S_2S_1$ is a three-bit binary number.

3 Program (300 points)

You will write a program `truthtable` that reads a circuit description and prints a truth table showing all combinations of input variables.

`truthtable` takes a single argument, a file containing a circuit description.

Part 1 (Regular Credit) For this part, the circuit descriptions will be sorted so that each temporary variable appears as an output parameter before any appearances as an input variable.

Part 2 (Extra Credit) For this part, the circuit descriptions will *not* be sorted, meaning that a temporary variable may be used as an input parameter before its use as an output parameter.

Input The input to your program will be a single circuit description using the language described in section 1. The first argument to `truthtable` will identify a file containing this circuit description.

You MAY assume that the input is correctly formatted and that no variable depends on its own output.

Output The output of `truthtable` is a truth table showing each combination of inputs and the corresponding output for the specified circuit. Each column in the table corresponds to a specific input or output variable, which are given in the same order as their declaration in the `INPUT` and `OUTPUT` directives. Columns are separated by a single space, and a vertical bar (`|`) occurs between the input and output variables.

Note that no white space follows the final column.

Usage

```
$ cat circuit1.txt
INPUT 3 : IN1 IN3 IN4
OUTPUT 1 : OUT1
MULTIPLEXER 2 : 1 0 1 0 : IN3 IN4 : temp1
MULTIPLEXER 1 : temp1 1 : IN1 : OUT1
$ ./truthtable circuit1.txt
0 0 0 | 1
0 0 1 | 0
0 1 0 | 1
0 1 1 | 0
1 0 0 | 1
1 0 1 | 1
1 1 0 | 1
1 1 1 | 1
$ cat circuit2.txt
INPUT 2 X Y
OUTPUT 2 Z W
XOR X Y T
PASS T Z
NOT T W
$ ./truthtable circuit2.txt
0 0 | 0 1
0 1 | 1 0
1 0 | 1 0
1 1 | 0 1
```

For part 2, `truthtable` must handle unsorted circuit descriptions:

```
$ cat circuit3.txt
INPUT 4 A B C D
OUTPUT 1 Z
AND A B E
MULTIPLEXER 2 0 1 1 0 E F Z
OR C D F
$ ./truthtable circuit3.txt
0 0 0 0 | 0
0 0 0 1 | 1
0 0 1 0 | 1
0 0 1 1 | 1
0 1 0 0 | 0
0 1 0 1 | 1
0 1 1 0 | 1
0 1 1 1 | 1
1 0 0 0 | 0
1 0 0 1 | 1
1 0 1 0 | 1
1 0 1 1 | 1
1 1 0 0 | 1
1 1 0 1 | 0
1 1 1 0 | 0
1 1 1 1 | 0
```

4 Implementation Suggestions

While there are many ways to implement this project, the scheme outlined here is fairly simple to implement and is able to handle large truth tables without running out of time or space.

1. Avoid string comparisons by representing every variable as a number. For uniformity, you can represent 0 and 1 as 0 and 1, the m inputs as $2, 3 \cdots m+1$, the n outputs as $m+2, m+3 \cdots m+n+1$, and the p temporaries as $m+n+2 \cdots m+n+p+1$. This permits representing a complete set of variable assignments as an array.

   The file always begins with the lists of input and output vars, so m and n will be known before any temporary variables are found. Maintain a mapping of names to integers; each time you read a name not already in the mapping, give it the next smallest index and add it to the mapping.

2. Avoid repeatedly reading the input file by encoding each directive as a data structure. This data structure needs to describe

   - the type of gate
   - (optionally) information about the gate size (e.g., what size decoder)
   - the inputs
   - the outputs.

For the inputs and outputs, you simply need arrays of integers, since each variable is represented as an index.

3. To generate a particular row in the truth table, create an array of size $m + n + p + 2$. Initialize the constant and input variables. Then, for each gate, calculate the output value(s) and set the items in the array accordingly.

4. For each row of the truth table, note that every input value will be followed by a space, and every output value will be preceded by a space.

5. When reading the circuit description, you may assume that the file is correctly formatted. Thus, after reading a directive name, you may assume that it will be followed by the correct number of parameters. Thus, it is acceptable to use the whitespace-skipping tokenizer and ignore line breaks.

6. Since the behavior of `truthtable` is not determined for ill-formatted input files, it is not necessary to check that colons appear in the correct places. When reading the paramters to a directive, first try to read a colon, then try to read a variable name if that fails.

   Alternately, the format code `"%*[: ]%16s"` reads and discards any number of whitespace and colon characters before reading a string token of up to 16 characters (not including the terminator).

5 Grading

This assignment is worth 400 points. Designs for the circuits described in section 2 are worth 100 points. The program described in section 3 is worth 300 points. Your program MUST successfully compile and execute when using the auto-grader on an iLab machine in order to receive points. The auto-grader is provided so that you can confirm that your submission can be tested successfully. **It is your responsibility to ensure that your program can be tested by the auto-grader.**

Your program must be compiled with `-Wall -Werror -fsanitize=address`. If these options are not present, your score will be reduced by one fifth.

The auto-grader provided for students includes several test cases, but additional test cases will be used during grading. Make sure that your programs meet the specifications given, even if no test case explicitly checks it. It is advisable to perform additional tests of your own devising, including but not limited to the use of user tests as described in section 6.5.

5.1 Academic integrity

You must submit your own work. You should not copy or even see code for this project written by anyone else, nor should you look at code written for other classes. We will be using state of the art plagiarism detectors. Projects which the detectors deem similar will be reported to the Office of Student Conduct.

Do not post your code on-line or anywhere publically readable. If another student copies your code and submits it, both of you will be reported.

6 Submission

Your solution to the assignment will be submitted through Sakai. You will submit a Tar archive file containing your circuit designs, and the source code and makefile for your program. Your archive should not include any compiled code or object files.

The remainder of this section describes the directory structure, the requirements for your makefile, how to create the archive, and how to use the provided auto-grader.

6.1 Directory structure

Your project should be stored in a directory named `src`. This directory will contain (1) a makefile, (2) any source files needed to compile `truthtable`, and (3) a subdirectory `tests` containing your circuit designs from section 2 and their reference output (see section 6.5.

This diagram shows the layout of a typical project:

```
src
+- Makefile
+- truthtable.c
+- tests
   +- test.1.ref.txt
   +- test.1.txt
   +- test.2.ref.txt
   +- test.2.txt
   +- test.3.ref.txt
   +- test.3.txt
   +- test.4.ref.txt
   +- test.4.txt
   +- test.5.ref.txt
   +- test.5.txt
```

If you are using the auto-grader to check your program, it is easiest to create the `src` directory inside the `pa4` directory created when unpacking the auto-grader archive (see section 6.4).

6.2 Makefiles

We will use `make` to manage compilation. Each program directory will contain a file named `Makefile` that describes at least two targets. The first target (invoked when calling `make` with no arguments), MUST compile the program. An additional target, `clean`, MUST delete any files created when compiling the program (typically just the compiled program).

You may create this makefile using a text editor of your choice.

A typical makefile would be:

```
truthtable: truthtable.c
	gcc -g -Wall -Werror -fsanitize=address -o truthtable truthtable.c

clean:
	rm -f truthtable
```

The auto-grader will use both targets when testing your program. If either target cannot be made, or if `truthtable` is present after executing `make clean`, or if `truthtable` is not created after `make truthtable`, you will receive no points.

The command for compiling `truthtable` MUST include GCC warnings and AddressSanitizer. You will lose a fifth of your points if you do not include these.

You are not required to use `-g`. Including it will enable debugging using `gdb` and may improve AddressSanitizer error messages.

Note that the makefile format requires that lines be indented using a single tab, not spaces. Be aware that copying and pasting text from this document will "helpfully" convert the indentation to spaces. You will need to replace them with tabs (literal tab characters), or simply type the makefile yourself. You are advised to use make when compiling your program, as this will ensure (1) that your makefile works, and (2) that you are testing your program with the same compiler options that the auto-grader will use.

6.3 Creating the archive

We will use `tar` to create the archive file. To create the archive, first ensure that your `src` directory contains only the source code and makefile needed to compile your project. Any compiled programs, object files, or other additional files must be moved or removed.

Next, move to the directory containing `src` and execute this command:

```
tar czvf pa4.tar src
```

`tar` will create a file `pa4.tar` that contains all files in the directory `src`. This file can now be submitted through Sakai.

To verify that the archive contains the necessary files, you can print a list of the files contained in the archive with this command:

```
tar tf pa4.tar
```

You should also use the auto-grader to confirm that your archive is correctly structured.

On some operating systems, `tar` may find or create hidden files and include them in your archive. This is usually not a problem.

6.4 Using the auto-grader

We have provided a tool for checking the correctness of your project. The auto-grader will compile your programs and execute them several times with different arguments, comparing the results against the expected results.

Setup The auto-grader is distributed as an archive file `pa4_grader.tar`. To unpack the archive, move the archive to a directory and use this command:

```
tar xf pa4_grader.tar
```

This will create a directory `pa4` containing the auto-grader itself, `grader.py`, a library `autograde.py`, and a directory of test cases `data`.

Do not modify any of the files provided by the auto-grader. Doing so may prevent the auto-grader from correctly assessing your program.

You may create your `src` directory inside `pa4`. If you prefer to create `src` outside the `pa4` directory, you will need to provide a path to `grader.py` when invoking the auto-grader.

Usage While in the same directory as `grader.py`, use this command:

```
python grader.py
```

The auto-grader will compile and execute the program `truthtable` in a directory `src` contained in the current working directory, assuming `src` has the structure described in section 6.1.

By default, the auto-grader will test parts 1 and 2 of the project. To grade only a particular part, give its name as an argument. For example:

```
python grader.py truthtable:1
```

To obtain usage information, use the `-h` option.

Program output By default, the auto-grader will not print the output from your program, except for lines that are incorrect. To see all program output for unsuccessful tests, use the `-v` option:

```
python grader.py -v
```

To see program output for all tests, use `-vv`. To see no program output, use the `-q` option.

Checking your archive You SHOULD use the auto-grader to check an archive before (or just after) submitting. To do this, use the `-a` option with the archive file name. For example,

```
python grader.py -a pa4.tar
```

This will unpack the archive into a temporary directory, grade the programs, and then delete the temporary directory.

Specifying source directory If your `src` directory is not located in the same directory as `grader.py`, you may specify it using the `-s` option. For example,

```
python grader.py -s ../path/to/src
```

6.5 User-provided tests

Your implementations of the five circuits described in section 2 should be provided in the `tests` subdirectory of `src` and named `test.1.txt`, `test.2.txt`, and so forth. It is recommended that you also provide expected output for these circuits, named `test.1.ref.txt`, `test.2.ref.txt`, and so forth. If both circuits and reference outputs are provided, the auto-grader will automatically test `truthtable` using your circuits and compare its output to the provided references.

You may optionally include additional tests following the same naming convention, but these will not affect your grade.