CS211 Fall 2019
Programming Assignment IV

David Menendez

Due: Monday, November 25, 2018
Submit by November 26 at 3:00 AM

This assignment will provide more practice programming in C and working with circuits and digital logic. You will write a program that generates a truth table for a circuit specified using a specification language.

Tests will be broken into three groups, based on the number of language features used. In the first part, all AND, OR, NAND, and NOR gates will have exactly two inputs and temporary variables will appear as the output of a gate before they are used as the input to another gate. In the second part, gates may have two or more inputs. In the third part, gates may be given in any order. See section 1 for details about the specification language.

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 3 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

A circuit specification file describes (a) a sequence of *input variables*, (b) a sequence *output variables*, and (c) a set of *gates*. Each gate has a sequence of *parameters*, which give the inputs and outputs for that gate.

The file itself is a sequence of tokens, where each token is a gate type, a variable name, or one of a few special keywords. These tokens are organized into *directives*, consisting of a gate type followed by variables representing its parameters. This sequence also includes one or two colon tokens (`:`) separating the input and output variables.

For example, `AND X Y : Z` is a directive specifying an AND gate with inputs X and Y and output Z.

Table 1: Gate inputs and outputs

Gate	Inputs	Outputs
PASS	1	1
NOT	1	1
AND	n	1
OR	n	1
NAND	n	1
NOR	n	1
XOR	2	1
DECODER	n	2^n
MULTIPLEXER	$n + 2^n$	1

1.1 Lexical structure

The specification file is a sequence of tokens, forming several directives. Each token is a sequence of up to 16 non-whitespace characters. Tokens are separated by whitespace.

Typically, the tokens that make up a directive will occur on a line by themselves, but it is not necessary to require this.

Several tokens have special meaning: `INPUT`, `OUTPUT`, `PASS`, `NOT`, `AND`, `OR`, `NAND`, `NOR`, `XOR`, `DECODER`, `MULTIPLEXER`, `:`, `0`, `1`, `_`. All other tokens are variable names.

You may assume that variable names begin with a letter and contain only letters and digits, but you are not required to check this.

Tokens are case-sensitive: `X` and `x` are different tokens.

1.2 Directives

A directive consists of a keyword followed by a sequence of parameters and colons.

The INPUT directive always occurs first. It is followed by one or more variables, which will be considered input variables.

The OUTPUT directive always occurs after the INPUT directive. It is followed by one or more variables, which will be considered output variables.

The remaining directives specify gates. Gates take a different number of parameters, which can be grouped into inputs and outputs. Some gates take variable number of parameters. Table 1 shows the number of parameters for each gate kind.

The parameter list for each gate kind except MULTIPLEXER is given in two parts. The first part gives the inputs for the gate and the second part gives the outputs. The inputs and outputs are separated by a colon.

The parameters for a MULTIPLEXER are given in three parts. The first part gives the n selector inputs, the second part gives the 2^n input lines that the multiplexer selects from, and the third part gives the output. The three parts are separated by colons.

1.3 Parameters

Each variable name in a circuit description has one of three roles: input variable, output variable, or temporary variable. The input and output variables are exactly those given by the INPUT and

OUTPUT directives, respectively. All other variables are temporary variables.

The inputs of a gate must be input variables, temporary variables, `0`, or `1`.

The outputs of a gate must be output variables, temporary variables, or `_`.

No variable may occur as an output more than once. Input variables can only be used as inputs, and output variables can only be used as outputs. Temporary variables must occur once as an output, and may occur zero or more times as inputs.

The special inputs `0` and `1` are not variables, but represent constant inputs.

The special output `_` is a "don't care" value, indicating an output line that is not connected to anything.

1.4 Gate kinds

Each gate has one of the following kinds:

- PASS – output is equal to input
- NOT – output is inverse of input
- ADD – output is true if all of its inputs are true
- OR – output is true if at least one input is true
- NAND – item is true unless all of its inputs are true
- NOR – item is true if none of its inputs are true
- XOR – item is true if exactly one input is true
- DECODER – the n inputs are interpreted as a binary integer in the range 0 to $2^n - 1$; the outputs are numbered 0 to $2^n - 1$, and the output with the number given by the inputs will be 1 and all others 0
- MULTIPLEXER – the n selectors are interpreted as a binary integer in the range 0 to $2^n - 1$; the remaining inputs are numbered 0 to $2^n - 1$; the output will be equal to the input with the number given by the selectors

The number of input and output parameters for each gate is given in table 1.

1.5 Examples

This circuit describes a half-adder, where S is the sum and C is the carry-out.

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

This circuit computes $z = ab + ac$

```
INPUT a b c
OUTPUT 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 (`OUT1`) as an input parameter:

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

This can be fixed by using `PASS`:

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

This circuit demonstrates the use of `MULTIPLEXER`:

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

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

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

This circuit demonstrates DECODER and the use of `_`.

```
INPUT a b c
OUTPUT d
DECODER a b c : _ _ _ e f _ g h
OR e f g h : d
```

2 Program (100 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 (30 points) In this part, there will be at most 5 input variables and at most 8 output variables. AND, OR, NAND, and NOR gates will always have exactly 2 input parameters. No DECODER or MULTIPLEXER gates will occur.

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 (50 points) In this part, there are no limits on the number of input or output variables, and any gate may be used with any number of parameters consistent with table 1.

Part 3 (20 points) 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 In part 1, circuits will have a maximum number of input and output variables, and gates will have fixed numbers of parameters.

```
$ cat circuit1.txt
INPUT X Y
OUTPUT Z W
XOR X Y : T
PASS T : Z
NOT T W
$ ./truthtable circuit1.txt
0 0 | 0 1
0 1 | 1 0
1 0 | 1 0
1 1 | 0 1
```

In part 2, circuits may have arbitrarily many inputs and outputs, and gates may allow variably many parameters.

```
$ cat circuit2.txt
INPUT IN1 IN3 IN4
OUTPUT OUT1
MULTIPLEXER IN3 IN4 : 1 0 1 0 : temp1
MULTIPLEXER IN1 : temp1 1 : OUT1
$ ./truthtable circuit2.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
```

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

```
$ cat circuit3.txt
INPUT A B C D
OUTPUT Z
AND A B : E
MULTIPLEXER E F : 0 1 1 0 : 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
```

3 Implementation Suggestions

This project can be implemented in several ways, but a key to making an efficient solution is to choose data types wisely. Remember: searching is slow, string comparison is slow, and searching while using string comparisons is very slow. Your program will need to do some string comparison–based searches, in order to tell whether a variable name is new, but it is best to do this a few times as possible.

One good approach is to read the circuit description file once to create a data structure that represents the circuit. Next, generate the truth table one row at a time by selecting new values for the input variables and determining the values of the output variables.

To use this approach, you will want to design a data structure that allows for efficiently getting the current value of each variable and setting new values. One possibility is to assign each variable a number and store the values for the variables in an array, indexed by the variable number. An array with integer indices is both faster and easier to code than a hashtable or binary tree keyed by variable name.

Note that the number of gates and the number of inputs for many gates is not known in advance. It is not recommended to guess a maximum size and use an array; instead use a sequence structure that can be easily expanded, such as linked lists.

Give yourself time to design your program before you start coding. Think about the problems you will need to handle and how you can approach them. You may find it helpful to make diagrams.

Note that tokens are at most 16 characters. This means you can use a fixed-length string buffer for reading individual tokens, and that you can use the format code `%16s` with `fscanf` to read a string of at most 16 characters.

Note also that variable names are guaranteed not to clash with the specified keywords. Thus, when reading the input parameters, you may keep reading until you find the token `OUTPUT`. For gates, the first `:` encountered signals the end of the inputs, at which point it is possible to compute the expected number of remaining parameters.

While your program may assume that its input is correctly formatted, you may want to check for some errors in order to catch problems while testing your program.

4 Grading

This assignment is worth 100 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 may be reduced.

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 5.5.

4.1 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.

5 Submission

Your solution to the assignment will be submitted through Sakai. You will submit a Tar archive file containing 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.

5.1 Directory structure

Your project should be stored in a directory named `src`. This directory will contain a makefile and any source files needed to compile `truthtable`. It may also contain subdirectory `tests` containing additional circuits and their reference outputs (see section 5.5.

This diagram shows the layout of a typical project:

```
src
+- Makefile
+- truthtable.c
+- tests
   +- test.1.ref.txt
   +- test.1.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 5.4).

5.2 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.

5.3 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.

5.4 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 5.1.

By default, the auto-grader will test parts 1, 2, and 3 of the project. To grade only some parts, give their names as arguments. For example:

```
python grader.py truthtable:1 truthtable:2
```

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
```

5.5 User-provided tests

It is recommended that you create additional tests. If you would like the auto-grader to run these tests in addition to the provided tests, create a directory `tests` in your `src` directory.

Each test consists of two files, `test.`*X*`.txt` and `test.`*X*`.ref.txt`, containing the file provided to `truthtable` and the expected output of `truthtable`, respectively.

When both files exist, the auto-grader will give the input file to `truthtable` and compare the output of `truthtable` to the reference file.

This testing capacity is provided for your convenience. Any tests you provide will not directly affect your grade.