Running:

To compile the parser and run the program navigate to the Lab5 folder and compile the program with the following:

g++ -o main main.cpp Parser.cpp Gate.cpp
./main

The program will prompt you to enter the name of the file. We have already included S359.v, S385.v, and S27.v in the Tests folder; any additional tests should be moved into this folder for the compiler.

Enter the name of the .v file to parse: S27.v

The program will then print out the number of gates it created from the file and a text file with the same name as the file entered will be created in the tests folder. It will also ask you if you want to make a readableTXT file, this is only used for debugging purposes. To simulate the text file you will need to navigate to the simulator folder and run the following:

cd .\Simulator
g++ -o sim Simlulator.cpp Gate_SIM.cpp Main_SIM.cpp
./sim
Enter the name of the gatefile to parse: *..\Tests\S27.txt*
Enter the name of the testfile file to parse: *..\Tests\S27.vec*
Run an Input Scan simulation or a Table lookup Simulation? (I/T*): T*

Parser Logic:

The parser generates pointers to gate elements for easier access and stores these pointers in two parallel data structures: an unordered map and a vector, referred to as gateMap and gates, respectively. By using pointers, we can update the gate values in either data structure, ensuring that changes will be reflected in both, provided we do not alter the gate names after their creation, as this is how the gates are hashed. The gateMap is used for name matching (lining up the inputs and outputs in the module declaration with where the inputs and outputs array locations are and connecting the gate names to the actual gate instantiations for example) because this reduces the search time to O(1) from a potential O(N), which would be visiting every gate to find the correct gate a gate instantiation is referencing.

To create the text file in a way the simulator can use it the parser runs 4 commands, which are used sequentially in Main.cpp:

1. parse(): This is the initial .v intake, which features a specific exception for wire (buffer) declarations, as they are categorized differently from inputs, outputs, and other gate types. Excluding the module declaration and the wires themselves, the intake procedure directs all other lines to the function parseLine(). This function is responsible for the systematic instantiation of all other gates.
2. assignGateLevels(): Starting from all the inputs, this function utilizes a queue to evaluate all fanout gates at the current level, incrementing their level by one. Once the queue is exhausted, it takes all the gates that were just updated and adds them to the queue to repeat the process. This ensures that if a gate is originally designated as level 1 but has an input also at level 1, it will be automatically corrected to a level 2. The output level of all D flip-flops is set to level 1 to prevent infinite loops.

3. sortGates(): This function categorizes the gates by level in ascending order. It also assigns each gate's arrayLocation value as the spot it is in the vector, because the gateMap will access them at random but we need to know the order they are in the vector.
4. makeTXT(): This function processes the now sorted vector of gates and outputs the results into a .txt file that shares the same name as the vector. Moreover, during this operation, the function substitutes the gate names with their corresponding array indices. These indices are retained as attributes of the gate class, enabling efficient access to the gates without necessitating sequential retrieval.
5. makeReadableTXT(): An extra function that prints the gate names instead of their array locations so you can understand what the output means.

This file presents information about the gates in the following way for the simulator:

GATES{34} INPUTS{4} OUTPUTS{1} DFFS{3}
INPUTS{3, 0, 1, 2}
OUTPUTS{32}
DFFS{33, 28, 19}
GATETYPE{INPUT} OUTPUT{FALSE} GATELEVEL{0} FANIN{} FANOUT{6} GATENAME{G1}
GATETYPE{INPUT} OUTPUT{FALSE} GATELEVEL{0} FANIN{} FANOUT{5} GATENAME{G2}
…

This provides the simulator with all the necessary information to accurately create the array of gates. This is why the fanout and fanin return numbers (array addresses) rather than gate names. The inputs line indicates where each input stimulus connects in the array—the most significant bit (MSB) of the stimulus should link to array location 3 in this case. The same applies to outputs, while the DFFs specify which array locations to retrieve the DFF values from and the sequence in which they should be displayed.

There seems to be an issue with the CRC_OUT_1_31 in S359.v as it is declared as an output and a wire, which causes the level of the output to be set to level -1, so I removed it from being a wire.

Simulator:

The simulator is written in C++, and needs the Simulator.cpp, Simulator.hpp, Gate_SIM.cpp, Gate_SIM.hpp, and Main_SIM.cpp.

Compile the program, then run the executable. The gate file is the output from the parser. The stimulus file is the simulation input. You can preform a table lookup based simulation or an Inpit Scan simulation by pressing I or T when prompted.

Method:

I wrote the simulator the way it was specified in the lab document. The program creates a simulator object whose constructor reads in the data from the gate file and stimulus file. This is fairly straightforward, as all the logic about which gates go where and how the are linked is taken care of in the parser program. I can then read the gate file top to bottom and get a correctly initialized simulator. We had to add some extra information to the gate file because the lab document's specification for the gate file did not contain information on the order of the inputs, outputs, and dffs. We need information on that to be able to accurately assign stimulus to the right inputs, then print everything in the correct order.

The internal data structure of the simulator is a large array of ‘Gate’s. Each gate has integer fields holding data, and can use integers as a sort of pointer to other gates because all the gates have a unique index in this array. Each gate has a “level”, and levels are stored next to each other.

Simulation algorithm:
The simulation can be divided into a series things that need to happen at different scales of the program. There are program wide steps, cycle wide steps, and level wide steps.

Program wide steps: what I must do each program execution:

- Construct the Gates data structure
- Construct the stimulus data structure
- Run the simulation

Cycle Wide steps: A “cycle” is an application of new input values and evaluation of the all the combinational logic in the circuit for those values. Each cycle I must:

- Load schedule data from previous cycle dff evaluations to this cycle,
- Load in input data and schedule the appropriate gates
- Print the states of the inputs and dffs,
- Evaluate the circuit level by level (level wide) and schedule gates as I go
- Print the output

- The cycle is repeated for as many times as there are lines on the stimulus file.

Level Wide Steps: Each gate is assigned an integer “level” by the parser. That level means that the gate can be evaluated if all gates of the previous levels have been evaluated. Therefore we evaluate each level one by one till we hit “max level” at which point the cycle repeat. Each level I must:

- If the first gate of the level, Check the array “levels” for the pointer to the first scheduled gate on that level.
- Evaluate that gate and if it changed schedule it’s fanouts
- Go to the gate’s schedule pointer and make the gate it points to the active gate.
- Set the old gate’s schedule pointer to the placeholder “dummy gate”
- Repeat till the current gates schedule pointer points to the placeholder “last gate”

- The scheduling works as a linked list where the schedule pointer points the next gate that should be evaluated. We build this linked list during runtime so only gates with fan ins that change are evaluated

Scheduling: when a gate changes state (changes it’s output) its fanouts are scheduled. The schedule for each level is stored as a linked list with its head in levels. Each index in levels corresponds to the head of the scheduling list for that level. When a dff changes state, it’s fanout is not immediately scheduled, but rather built as a separate linked list with its head in an array called “nextLevels”. At the start of each cycle, nextLevels is loaded into levels, so the dffs fanouts are effectively scheduled for the next cycle.

Evaluation: Each gate can be evaluated with the table lookup or the input scanning algorithm. I won't go into detail here about how those work as it is very well explained in the lab document.