ECSE 318 Lab 5/6 Report

Audrey Michel (pjm173)

December 10, 2024

Abstract

In this project I designed a java program that interprets verilog circuits and simulates them according to a given vector file. The design uses polymorphism and class inheritance to simplify how each object within the program is simulated, and all data structures used have been optimized, using either hashmaps or linked lists. Thus, this program works for any verilog file of any size.

This program does differ from its expected format in the way how sched is used as a hashmap that contains all gates, ordered by level, as well as how wires are used to model the respective circuit in just one iteration of the verilog file.

User Guide

The program is run through VerilogParser.java. Be sure to also compile all java files in the backend folder. When VerilogParser is run, it takes two arguments: the verilog file and then the vector file (in that order). After simulation, the program will print the **time elapsed to the console**, not the output file. If the respective output file (named after the verilog file) has not yet been created, the program will create one and output all data to it. If this file already exists, the program will overwrite the previous data.

```
javac -cp bin -d bin .\backend\*.java
java .\backend\VerilogParser.java ./S385.v ./S385.vec
```

Average run times

S27: 403 ms
S359: 1194 ms
S385: 14737 ms

Part One:

Modeling the circuit

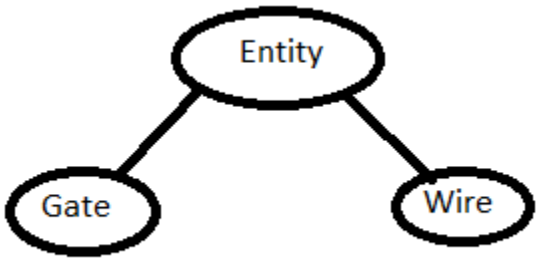

Figure: Representation of all polymorphism used in this project

The three objects shown above are used to model the parts of the circuit. Of course, Gate models gates and Wire models wires. The entity class is actually mainly used to describe objects that could be either a gate or a wire. An object will never be only an Entity and not a Gate or Wire.

Another class called Circuit is used to simulate the circuit. This class contains a hashmap used to store all wires (including inputs and outputs), alongside a linked list containing all the gates in the circuit. The linked list of gates- created such that all DFFs are at the beginning- is used to iterate through the circuit when printing the fan in and fan out of every gate.

```
public class Circuit {

    HashMap<String, Wire> wireList;
    HashMap<String, Wire> inputs;
    HashMap<String, Wire> outputs;
    HashMap<Integer, HashMap<String, Entity>> sched;
    Gate firstGate;
    Gate lastGate;
```

Figure: All local variables stored within the Circuit class

Linking Gates and Wires

In the first step of the circuit, all wires are added to the Circuit model before any gates are added. Once all wires have been introduced to the program, the code will iterate through each gate in the circuit, linking its fanin and fanout to the appropriate wires. When a gate adds a certain wire to its fanin, that wire will

add that gate to its “fanout,” and vice versa. This process allows for easy and quick modeling of all the gates and wires while only iterating through the input verilog file once.

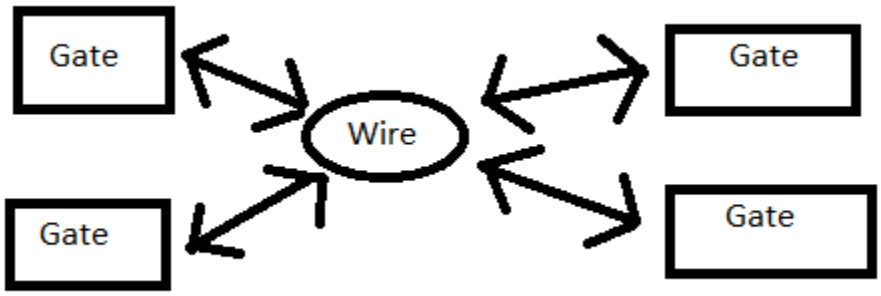

Figure: Example of how gates and wires point to each other

Buffer Creation

After the program has added all the gates to the circuit, it is time for step two: the creation of buffers. In this process, the program will then iterate through all non input/output wires in the wires hashmap. For each “fanin” of the wire, a connection with an intermediary buffer will be made between all of the wires “fanout” gates. Buffers are named BUF + fanin gate + fanout gate. Example below.

Gate Level Calculation

After the buffers are created, the circuit contents are printed out. Then it is time to calculate the gate levels. Since the end result of gates pointing to each other is much like a general tree (a binary tree where nodes can have 3+ children), I used a recursive algorithm to traverse the code.

```
void calculateLevels(int newLevel, HashMap<Integer, HashMap<String, Entity>> sched) {
    // If we reach a flip flop, stop and do not calibrate
    if (this.type == GateType.DFF)
        return;
    // If entity's max level is lower its new level, calibrate and traverse outputs
    if (this.level < newLevel) {
        int oldLevel = this.level;
        this.level = newLevel;
        recordLevel(oldLevel, newLevel, sched);
        DataWrapper<Entity> ptr = this.fanOut;
        while (ptr != null) {
            if (ptr.data != null)
                ptr.data.calculateLevels(newLevel + 1, sched);
            ptr = ptr.next;
        }
    }
}
```

Figure: the level calculation algorithm

This algorithm is run on every input wire with starting level zero, meaning the lowest level a non DFF gate can be is 1. After a level has been calculated, it will also be scheduled.

```
void recordLevel(int oldLevel, int newLevel, HashMap<Integer, HashMap<String, Entity>> sched) {
    // If the entity was previously logged into sched, remove it
    if (sched.containsKey(oldLevel) && sched.get(oldLevel).containsKey(this.name)) {
        sched.get(oldLevel).remove(this.name);
    }
    // If sched does not have a map for current level, create it
    if (!sched.containsKey(newLevel)) {
        sched.put(Integer.valueOf(newLevel), new HashMap<>(initialCapacity:100));
    }
    // Add entity to new spot in sched
    sched.get(newLevel).put(this.name, this);
}
```

Figure: the code to record each gates level

This recordLevel method will check to see if the gate's previous level has been recorded, then delete that old entry. Then it will record the gate under its new level. After this program has been run on all input wires and D flip flops, level calibration is complete.

Part Two:

Simulation

Simulation is fairly straightforward, but first I must explain my version of sched. It is best to explained with the diagram below. The sched (gate scheduler) hashmap holds multiple hashmaps all named after a

level. The overarching hashmap is like a list of levels, and each level has its own hashmap containing all of that level's gates. Each level-named hashmap then contains all the gates in that respective level

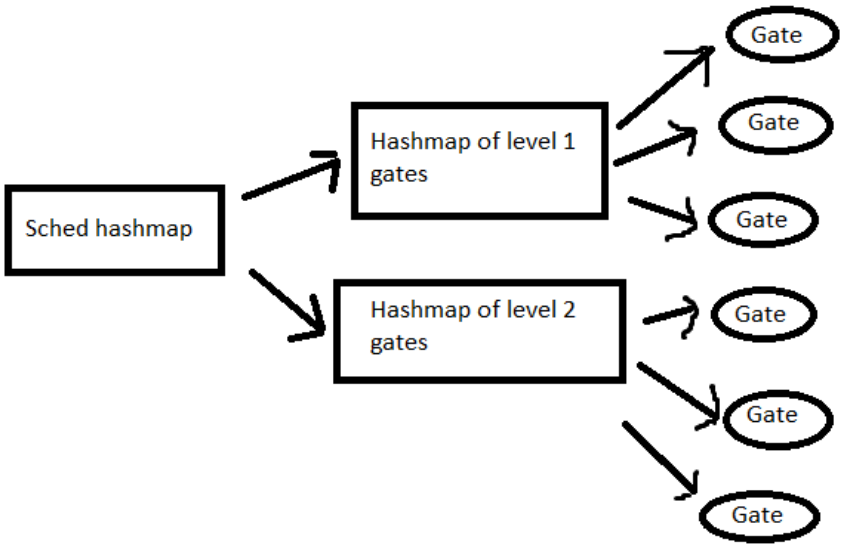

Figure: the organization of the sched and its nested hashmaps

This version of sched is more efficient than the proposed one as it is O(1) complexity to schedule gates. This is a huge contributor to why my simulation times are so fast (the largest circuit takes 15 seconds on average).

To simulate, first all inputs are set according to the vector file. Then the schedule evaluates each gate in order. Each gate then performs a fairly straightforward calculation based on the state of its inputs, and the process repeats until all gates have calculated their new values.