ECE 449
Project 4: Logic Simulation
Linghan Zhao A20313766
Dec 7, 2014

Logic Simulation

1. Overview

In project 4, Logic simulation of EasyVL language has been designed and implemented based on the previous project netlist construction. Recall about the project 3, we have built a netlist data structure includes nets and gates that are constructed from wires and components, also pins model the connections between nets and gates. In this project, we will simulate the corresponding circuits and implement the algorithms for them.

This project should be designed to implement different gates algorithm, for this function, the program will use derived class to overwritten key function to compute each gates output. In addition, the program can output simulation result file and input data as required gates. The critical section of this project is to implement combinational gates and input, output gates. Finally, lut and tris gates are the most difficult part of this project.

This project is developed in eclipse C++ environment and executed both in windows and CI system.

2. Features Design

Initial Release Features

A logic simulation of EasyVL is based on the netlist contracture, each net connects to different type's gates, and each gate has its own logic computation. In the initial release, as professor's suggestions, I provide support for *not*, *evl_dff* and *evl_output* gate, in addition to *evl_one* and *evl_zero* gate, and then design several test cases to test results.

To make this design implementation, as professor suggested in class, I refractor several class from last project, especially class net and gate.

For class *net*, we may have to compute the value of each net when simulation, so I add a variable to record the value of net and a flag to record if this net has been set logic value. And another function added is *bool compute()*, it will return the computational result of logic value to this net.

```
class net {
    ……
    net(std::string name, int v, bool f){
        net_name = name;
        value = v;
```

```
        flag = f;
    }
    ~net(){}
    ……
    int value; //true is 1 and false is 0
    bool flag;
public:
    …
    bool compute(); // compute logic value
     }; // class net
```

For class *gate*, as the features of this project, gates are the most important part and overwriting part. Therefore, using the inheritance method for different type gate is the best. At first, class *gate* should be the base class. It contains two virtual functions that are for other derived class to overwriting. Protected members allow derived class to access but not for all general users.

```
class gate {
    ……
    virtual bool validate_structural_semantics() = 0;
    virtual bool compute_output() {return true;};

protected:
    std::vector<pin *> pins_;
    ……
    virtual ~gate(){};
}; // class gate
```

After these processes, I can start to write derived different type gate class and implement their computation. What important functions or classes I implemented in initial release list below

Name	Type	Functions
netlist::simulate(int cycles)	bool	Do simulation of all the netlist and save result to specific file.
net::compute()	int	Compute logic value of each net, when computing, it will call the overwriting function of corresponding gate
class flip_flop: public gate	class	Class evl_dff gate, do validation and computation of the nets in this gate
class output_gate: public gate	class	Class out_put gate, do validation and computation of the nets in this gate, for initial nets size is only 1.
class not_gate: public gate	class	Class not_gate, do validation and computation of the nets in this gate

In every derived class, I derived virtual functions from base gate class. Therefore, I can overwriting function *validate_structural_semantics()* and *compute_output()*. In the former function, I will set a validation to judge if the gate has the right number of pins and correct width of nets as specific gate requirements. In the later function, the logic computation of gate happens here. In this function, accurate output should come out from here.

In addition, I also implement *evl_one* and *evl_zero* in the initial release for more selected test cases.

Final Release Features

In the final version, I finish rest gates and logic computation. At first, as suggestion of professor, I add width of nets for output gate computation so it can output bus.

```
bool output_gate::compute_output(){
……
    for(int i = 0; i < pin_size; i++){
        ……
        if(width == 1){
            pin_value = pins_[i]->nets_[0]->get_logic_value();
            outputfile << pin_value << " ";
        }
        // width != 1
        else{
            pin_value = 0;
            nets_size = pins_[i]->nets_.size();
            w = width/4;
            //std::cout << w << std::endl ;
            if(width%4 != 0){
                w = w+1;
            }
            ……
            for(int j = 0; j < nets_size; j++){
                ……
                //std::cout << (int)pins_[i]->nets_[j]->get_logic_value() ;
            }
            ……
}
```

Then, completing other derived combination gates and validation or computation of them is relatively simple. These gates include and, or, xor, buf, clock and so on. After these gates implemented, I have passed 5 golden tests. Then, I try to finish input gate, tris gate and lut gate which I will discuss design of them in next part. The final core implement class of gate is listed below.

Name	Type	Functions
hex_to_2(char hex)	string	Change from 1 bit hexadecimal to 4 bits binary.
class output_gate: public gate	class	Finish bus part of nets
class buf: public gate	class	Class buf gate, do validation and computation, If the input in is 1, then the output out is 1; otherwise out is 0. It can pass Z value
class input_gate: public gate	class	Class evl_input gate, do validation and computation, input file from outside to set value to corresponding nets
Class tris_gate: public gate	class	Class tris gate, do validation and computation (discuss in next section)
Class lut_gate: public gate	class	Class lut gate, do validation and computation, look up table to find corresponding value by address and set to specific nets.

The final release is completed step by step as project suggestion by professor, completing output and dff gate in first release will be helpful to understand how cycle simulation works. In final release, I complete rest gates step by step, from bus of output, to input gate, finally to tris and lut.

3. Key Implement

This project focuses on the logic computation of evl file. There are several type gates that are complexity and hard to implement. I would like to talk about how I design and implement them.

3.1 flip-flop gate

In flip-flop gate class, I set two values: state_ and next_state_ to store statements.

```
class flip_flop: public gate{
    int state_;
    int next_state_;
    ……
        };// class flip_flop
```

In flip-flop gate, the state value of evl_dff is set to be next value of last cycle and then output to file. The dff will ask value from its net of input pin. Then find next gate connects to the specific net and so on, repeating this till the next state value of output gate is obtained.

3.2 input gate

In input gate class, number of transitions should be an important value to decide which value to be input to specific output pins.

```
class input_gate: public gate{
    ……
    int number_of_transitions;
    std::ifstream read;
   public:
            input_gate( std::string name_ ):gate( "evl_input", name_){
                number_of_transitions = -1;
                std::string filename = std::string(file) + "." + name + ".evl_input";
                ……
            if(!read)
                ……
            else
                read.open(filename.c_str(),std::ios::in);
            }
            ……
};
```

In the logic part, as number of transition changes, program will decide which value should be assigned to output pins. I also set a vector to store input values in each line.

3.3 lut gate

In lut gate class, it is similar like the input gate class, this logic computation also needs to read a lut file from outside and find the set value based on the address.

```
class lut_gate: public gate{
    ……
    std::ifstream read;
    int first_line_flag;
    std::vector<std::string> lut_values_;
   public:
            ……
};
```

The major difference from input gate is that when it read first line of lut file, I set a vector to store all the lut values. So, when next time program encounter lut gate, it can look up the vector and use index of address to find the corresponding value.

3.4 tris gate

No doubt it is the hardest logic of this program. I try to set a Z value to each net. When en of tris is 0, I set output pins as Z. Also, if value of in is Z and en is 1, set output pins as Z, too. Otherwise, set the normal value as before. Gate of buf is the same as tris.

After this, in function `net::compute()`, the logic part is that if one output of nets return Z value, then continue to find next connections.

4. Test Cases design

In the initial release, I create 5 tests only contains 1-bit nets in gate and 5 different gates type including evl_dff, output, evl_one, evl_zero and not.

However, in the final release, I modify some tests included rest required parts. Also, I use the golden tests to help test reliability and robustness.

Case 1: test.evl

This file will test correctness of 1-bit nets and several gates type (evl_zero, evl_output) of the programming which implemented in initial release. Result is the same as expectation and the test is passed.

Case 2: test2.evl

This file will test rightness about complexity of gates and nets combination. The generated *.output* file shows the right display format. Result is the same as expectation and the test is passed.

Case 3: cpu8_flat.evl

This file is from golden test that will test combination gates and all logic part in this project. It should be the most comprehensive test for the program. Result is the same as expectation and the test is passed.

Case 4: io.evl

This file is from golden test that will test rightness of input gate. The program will read the .input file to set value of specific pins and affect the output file values. Result is the same as expectation and the test is passed.

Case 5: tris_lut.evl

This file is from golden test that will test rightness of tris and lut gate, as well as the buf gate that has modified to fit for the Z value conditions. Result is the same as expectation and the test is passed.