ECE 449
Project 3: Netlist Construction
Linghan Zhao A20313766
Nov 7, 2014

Netlist Construction

1. Overview

In project 3, Netlist Construction has been designed and implemented based on the previous project syntax analysis. In project 2, we moved tokens to statements and use finite statement to extract wire and component. In this project, I build a data structure named Netlist from wires and components. A Netlist data structure includes nets and gates that are constructed from wires and components. Pins model the connection between nets and gates.

This project is designed by an OOD (object-oriented design) method. In Netlist object, each net, gate and pin is an object. Each of them has functions or data structures and holds pointer to related objects.

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

2. Features Design

Initial Release Features

In the initial release, I refactor code, using header files and translation units, to separate one single file to different understandable parts based on the previous project. The *syn.cpp* and *syn.h* file contains the project 2 syntax analyze process. In the *main.cpp*, we claim some functions to receive results of previous project and use them in the later program. In the *main.cpp*, I use functions: *parse_evl_file*, *make_wires_table* and other important functions implemented in *netlist.h* I will discuss later. The main functions are showing blow.

Name	Type	Functions
parse_evl_file	bool	Parse evl_file and process it to store them to wires and components data structures (project 2 main function)
make_wires_table	std::map<std::string, int>	Transform wires data structure to map, first element is name, second element is width.

To create circuit netlist, a data structure to represent circuits in programs, I will create nets and gates separately in netlist.h file. At first, it generates nets from wires, then it generates gates from components and creates pins from components pins. At last, it connects each pin to related nets and saves to *.netlist* file based on requirements. The simple flow diagram of this project shows in Figure 1.

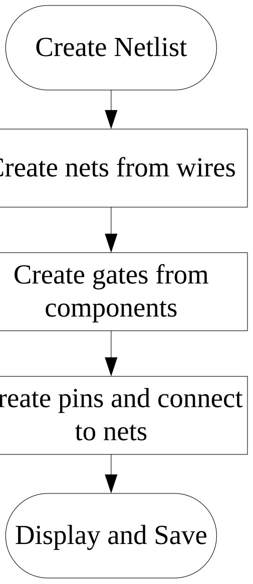


Figure 1. Simple flow diagram

I will talk about the important OOD implementation in next major part. And some main functions implemented in *netlist.h* in initial release shows blow.

Name	Type	Functions
make_net_name	std::string	Make net name for wires in every bit
netlist::create	bool	Main function to create netlist including nets and gates creation
netlist::create_nets	bool	Create nets from wires, use a map data structure to handle and store them to net list.
netlist::create_gates	bool	Create gates from components, and store them to gate list.
gate::create_pin	bool	Create pins from each gates and store them to pins data structure in gate class
pin::create	bool	Decide pins' possibilities depend on bus value and append them to related nets.

The reason why I choose these features because refactoring codes of second project will be more readable and extendable for further functions implementation. And the *netlist.h* file functions are the basic parts to implement. For the initial release, the programming can support 1-bit wires for different gate types. I will make up bus for rest parts in final release.

Final Release Features

In the final version, I finish all the rest parts including nets, gates and pins bus conditions. At first, in *gate::create_pin* function, I discuss different situations about 1-bit wire and bus wires. I introduce another net vector data structure to bus wires. If the pins in gates are bus wires, it will be pushed to net vector by every bit, and append to related net. After completing the bus parts, using corresponding access ways to display them in the right format and store to *.netlist* file. The modified main functions show below.

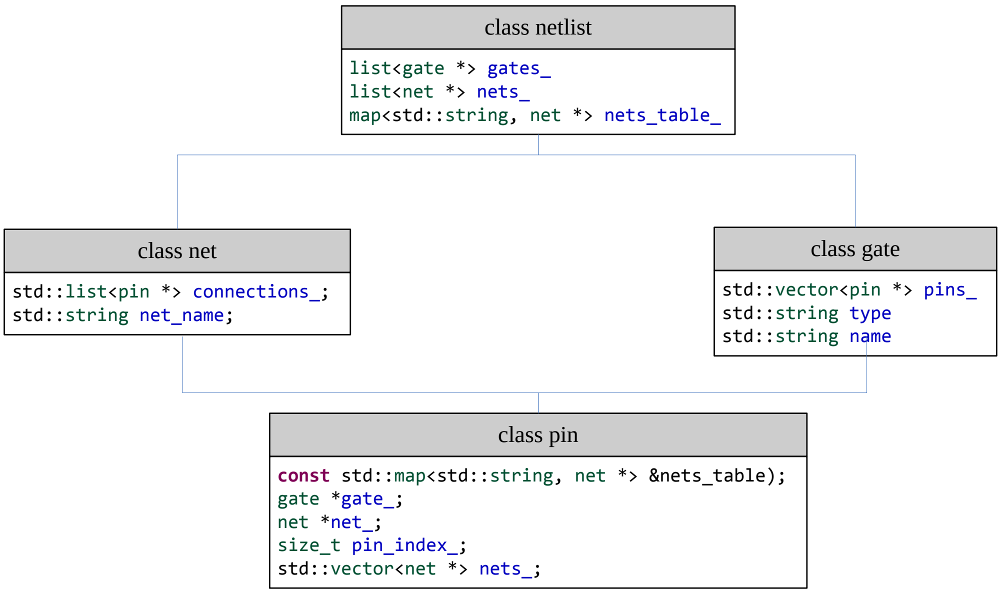
Name	Type	Functions
pin::create	bool	Add bus part condition to pins and connect to related net.
netlist::display	void	Finish rest part of pins display and display all in right format.
netlist::save	bool	Store display to .netlist fle

3. Key Implement

This project focuses on the *netlist* construction of EasyVL language. And what I should do based on previous project is to use OOD mechanism to store and process existed wires and component.

In this project, as requirements and guides by professor, I create four classes for object *netlist*, *gate*, *net* and *pin*. In general, the *netlist* will access gates and nets, each gate will access its pins, each net will access pins it is connected to and pin will access the gate it belongs to and the nets connected to it.

The simple connection among class shows below in Figure 2, only components of class list here.



```
list<gate *> gates_
list<net *> nets_
map<std::string, net *> nets_table_
```

```
std::list<pin *> connections_;
std::string net_name;
```

```
std::vector<pin *> pins_
std::string type
std::string name
```

```
const std::map<std::string, net *> &nets_table);
gate *gate_;
net *net_;
size_t pin_index_;
std::vector<net *> nets_;
```

Figure 2. Simple Connection among 4 Classes

We can figure out that the net object holds a pointer to record the pins connect to net, which implemented by its own function *void net::append_pin*; the pin object will record the nets and gates it connect to by its own *bool pin::create* functions and data structures; the gate object holds a pointer to record the pins connect to gate, which implemented by its own *bool gate::create* and *bool* gate::*create_pin* functions.

And in the core class *netlist*, it will use its own member functions to create nets and gates, then display and store right format to specific *.netlist* file. The simple main functions in *netlist* are

bool netlist::create
bool netlist::create_nets
void netlist::create_net
bool netlist::create_gates
bool netlist::create_gate
void netlist::display
bool netlist::save

4. Test Cases design

In the initial release, I create 5 tests only has 1-bit wire and different gates type.

However, in the final release, I modify some tests included all 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 wire and several gates type 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 display of gates and nets. The generated *.netlist* file shows the right display format. Result is the same as expectation and the test is passed.

Case 3: test3.evl

This file will test bus of wires which implemented in final release. The language is *wire [5:0]s0, s1, clk; wire [1:0]a;* which will display right format of related gates, nets and pins. Result is the same as expectation and the test is passed.

Case 4: test4.evl

This file will test 1-bit wire and bus wires together, as well as different types of gates. Result is the same as expectation and the test is passed.

Case 5: test5.evl

This file content is the same as the one in example in project 3 requirements. It will test all parts included in this project. Result is the same as display in project requirement and the test is passed.