Project GF2: Software
First Interim Report
Software Design Team 1

George Ayris	James Glanville	Andrew Holt
gdwa2	jg597	ah635
Emmanuel	Emmanuel	Emmanuel

21 May 2013

1 Introduction

This project requires the development of a logic simulation program, implemented in C++. When completed, the application will read a definition file containing a list of devices and the connections between them. It will then graphically display the values of specified monitor points in the circuit as the simulation is run. Some legacy code has been provided, but lacks a scanner, parser and GUI which will be developed in this project.

1.1 Teamwork Planning

The GUI development requires learning and reference to the wxWidgets and OpenGL packages, so makes sense for one team member to focus efforts here. The scanner and parser will require robust coding and testing, so it was decided to combine the two as a peer programming project to allow implementation and testing to be carried out by different team members.

The work is to be distributed as follows: Andrew will code the GUI, and James and George will write and test the scanner and parser. The time frame for the development is that the software should be designed by the end of Tuesday 21st May; implemented and unit tested by Tuesday 28th; allowing time for system integration and testing by 11.00am on Friday 31st May.

2 Syntax Specification

The syntax for the circuit definition files was defined and described using the following EBNF grammar:

```
DEFINITION = '{' DEVICES INIT CONNECTIONS MONITORS '}'

DEVICES =  'DEVICES' '{' device {device} '}'
device = devicename '=' devicetype [ '(' digit {digit} ')' ] ';'
devicename = letter {letter | digit}
devicetype = 'AND' | 'NAND' | 'OR' | 'NOR' | 'XOR' | 'DTYPE' | 'CLK' | 'SW'

INIT = 'INIT' '{' {init} '}'
init = devicename '=' digit{digit} ';'

CONNECTIONS = 'CONNECTIONS' '{' connection {connection} '}'
```

```
connection = input '<=' output ';'
input = letter {letter|digit} ['.'letter|digit{letter|digit}]
output = letter {letter|digit} ['.'letter|digit{letter|digit}]

MONITORS = 'MONITORS' '{' monitor {monitor} '}'
monitor = monitorname '<=' output ';'
monitorname = letter{letter|digit}
```

3 Semantics Specification

The following set of semantic descriptions was created to describe the semantics.

- All names and keywords are case-insensitive, i.e. `GATE1.O` and `gate1.o` are the same, and must be unique.
- If the device has a variable number of inputs then the number required for the device must be specified in parentheses after the device type. The number of inputs specified must lie within the allowable range for that device type. A device that does not have a variable number of inputs should not have an argument in parentheses.
- Within "`CONNECTIONS`" every input (of defined devices) must be connected to a single device output.
- When specifying a connection, the input and output must both correspond to a defined device and a legal input/output symbol for that device.
- The clock, switch and gate outputs are accessed with "`.o`".
- The gate inputs are accessed with "`.n`" where `n` is 1 or 2 for XOR gates and 1-16 for all the other gates.
- The `DTYPE` inputs are accessed with "`.d`", "`.c`", "`.s`" and "`.r`", which represent `DATA`, `CLK`, `SET` and `RESET` (rather than `CLEAR` to avoid confusion with `CLK`).
- The `DTYPE` outputs are accessed with "`.o`" and "`.no`" for normal output (`Q`) and inverting output (`QBAR`).
- Within "`INIT`" all defined switches and clocks must have an initialisation and any initialised device must be specified in `Devices`.
- Clocks are initialised with a period of `n` simulation cycles, where `n` is a positive integer.
- Switches are initialised with a state that is either 1 or 0, where 1 is a logic high state on the output and 0 is a logic low state.
- Comments in the definition file are defined as any data between an opening "`/*`" and a closing "`*/`". Comments may be nested.

4 Error Handling

The scanner will pass symbols to the parser, one at a time. The symbols will also be stored in a private list in the scanner class. The parser will check that the syntax and semantic rules are followed as each symbol comes in from the scanner. If an error is found, the parser will

pass an error message to the scanner. The scanner will know which symbol was last passed, and hence which generated the error. The line will then be printed with a carat (^) pointing at the offending symbol, along with the error message that will give a brief description of what is wrong, such as "`expected ')' instead of ';'`", so as to inform the user how to fix the definition file.

When an error is found, the system will not continue to process the file beyond the end of that line, as one error is likely to cause further errors. This will allow just the first error to be shown, allowing easy debugging by the user, as they know exactly where the error has occurred without being distracted by other errors that resulted. The downside of this is that only one error will be shown at a time, so independent errors cannot be debugged at the same time.

The parser class is directly translated from the EBNF, and when the parsing routine encounters a symbol that does not correlate with the EBNF then an error will be reported. The EBNF specifies semi colons at the end of most lines, so parsing can stop as soon as it reaches the next semi colon.

Some of the semantic errors will be checked on a line by line basis, such as ensuring that the parameters in the expression are legal and defined. Others will require waiting until the section is completed, such as ensuring that all device inputs have a specified connection. This will be checked when the closing brace (}) denoting the end of the section is parsed.

5 Example Circuits and Definition Files

The following two examples show how to define logic circuits using our definition language. Both are common logic applications, the first is making a more complex logic device from the easiest to manufacture device, the NAND gate. The other shows how to define a common sequential logic, a 3 bit shift register, implemented in D-type bistables.

5.1 XOR Circuit Composed of NAND Gates

The XOR circuit is shown in figure 1. The specification file is shown below.

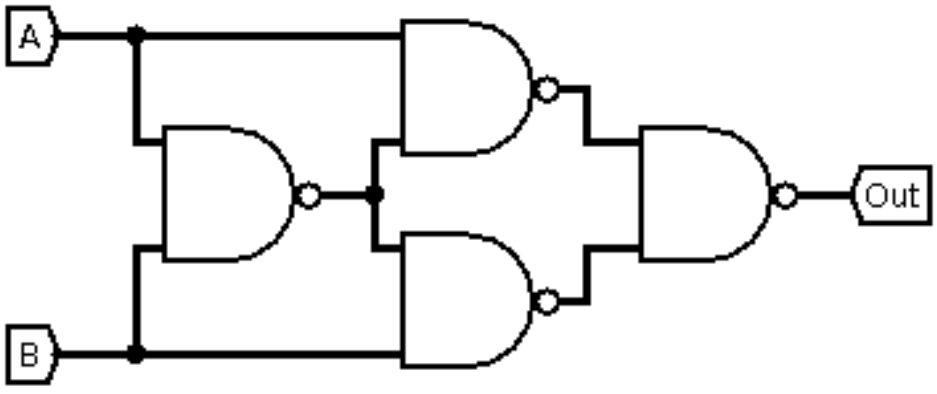

Figure 1: XOR circuit composed of NAND gates.

```
{
  DEVICES
  {
    N1 = NAND(2);
```

```
    N2 = NAND(2);
    N3 = NAND(2);
    N4 = NAND(2);
   SWA = SW;
   SWB = SW;
  }

  INIT
  {
    SWA.O=1;
    SWB.O=0;
  }

  CONNECTIONS
  {
    N1.1 <= A.O;
    N1.2 <= B.O;
    N2.1 <= A.O;
    N2.2 <= N1.O;
    N3.1 <= N1.O;
    N3.2 <= B.O;
    N4.1 <= N2.O;
    N4.2 <= N3.O;
  }

  MONITORS
  {
    OUT1 <= N4.O;
  }
}
```

5.2 3-bit Shift Register

The 3-bit shift register is shown in figure 2, and specified below.

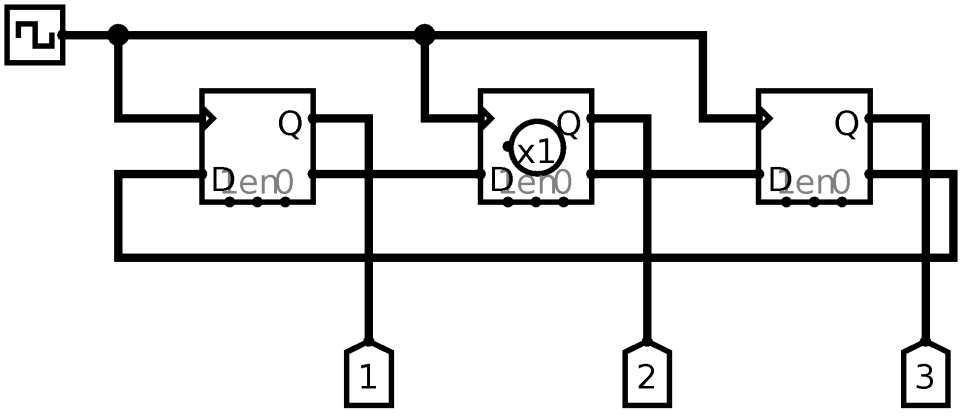

Figure 2: 3 bit shift register from D-type bistables

```
{
  DEVICES
  {
```

```
    D1 = DTYPE;
    D2 = DTYPE;
    D3 = DTYPE;
    CLK1 = CLK;
  }

  INIT
  {
    CLK1 = 1;
  }

  CONNECTIONS
  {
    D1.C <= CLK1.O;
    D2.C <= CLK1.O;
    D3.C <= CLK1.O;
    D1.D  <= D3.NO;
    D2.D  <= D1.NO;
    D3.D  <= D2.NO;
  }

  MONITORS
  {
    OUT1 <= D1.O;
    OUT2 <= D2.O;
    OUT3 <= D3.O;
  }
}
```