Part IIA, GF2 (Software Project): Interim Report 1

Andy Marshall, James Robinson, and Oliver Lambert

27th May 2011

Contents

---

1 Introduction

The aim of this group software project is to develop a functioning logic simulation program in C++. The project will depend on a number of concepts such as a formal language theory, scanning and parsing, object oriented programming and GUI implementation.

2 General Approach

The skeleton of the project was provided in the form of a number of module files, with complete code responsible for the '`devices`', '`network`' and '`monitor`' aspects of the simulator. Using this skeleton code, the team was tasked to create a logic circuit simulator. This simulator would take inputs in the form of a text file (the syntax and semantics of which to be defined), and would output the waveforms from various monitor points to a graphical user interface (GUI).

The tasks for this project fell into the five stages of the software life cycle:

1. *Specification:* this was largely completed already, with the client's requirements being detailed in the handout. Some further work was necessary to complete the specification to a level which allowed the design to take place. The additional work consisted of defining the syntax and semantics to be used in the definition file, as well as how errors would be handled.

2. *Design:* this was carried out in a top-down fashion, with the team agreeing the overall design scheme of the program including the interfaces between the different modules. More detailed design was undertaken by each member in isolation to establish how each module was to meet the requirements of the overall design scheme.

3. *Implementation:* this will be carried out according to the design. Generally, this can be done by team members individually, except where the details of the implementation required a change in the design, where the team will be consulted.

4. *Testing:* this is a bottom-up approach — all the modules are tested independently as far as possible. When all these modules are working properly, they will be combined and the system as a whole will be tested.

5. *Maintenance:* the details of the maintenance are not yet known, so the approach will be adapted depending on the nature of the maintenance requirement.

In order to allow the members of the team to easily share their code and to provide a system for version control (to allow recovery of the code if necessary), the 'github' system was used. This stores a complete history of the code on remote servers and can be accessed from any of the three OSs which were being used to write the software (Linux, Windows and MacOS).

2.1 Deadlines

Item	**Person(s) responsible**	**Due date**
First interim report	All	27/05/2011
`names` class	James	27/05/2011
Scanner	James	31/05/2011
Parser	Andy	31/05/2011
GUI	Oliver	02/06/2011
Integrated system	All	04/06/2011
Second interim report	Individual	07/06/2011
Maintenance tasks	All	10/06/2011
Final report	Individual	11/06/2011

2.2 Allocation of work

Due to the time constraints on the project, it was important to use each member of the team as effectively as possible. This meant completing as many tasks as possible individually, whilst retaining a coherent approach with the efforts of all team members co-ordinated effectively. To this end, the main tasks were allocated as follows:

Andy Parser and `symbol` class

James Scanner, `symbol` class, and `names` class

Oliver GUI

The team has a brief meeting at the beginning of each session to review progress and discuss any issues. This prevents a fragmented approach.

3 Syntax Definition

The syntax was defined using the formal EBNF language, and designed to allow LL(1) parsing. The definition file is shown below.

```
LogicDefinitionGrammar = { section }
section                = ( 'DEVICES' '{' {devrule} '}' ) |
                         ( 'CONNECTIONS' '{' {connrule} '}' ) |
                         ( 'MONITOR' '{' {monrule} '}' )
devrule                = device '=' uname ';'
connrule               = uname [ '.' outname ] '>' uname '.' inname ';'
monrule                = uname [ '.' outname ] ';'
uname                  = '' alphachar { alnumchar } ''
device                 = ( 'CLOCK' '-period' uint ) |
                         ( 'SWITCH' '-initialvalue' bool ) |
                         ( 'AND' '-numinputs' uint ) |
                         ( 'NAND' '-numinputs' uint ) |
                         ( 'OR' '-numinputs' uint ) |
                         ( 'NOR' '-numinputs' uint ) |
                         ( 'DTYPE' ) |
                         ( 'XOR' )
outname                = 'Q' | 'QBAR'
inname                 = 'I1' | 'I2' | 'I3' | 'I4' |
                         'I5' | 'I6' | 'I7' | 'I8' |
                         'I9' | 'I10' | 'I11' | 'I12' |
                         'I13' | 'I14' | 'I15' | 'I16' |
                         'DATA' |
                         'SET' |
                         'CLEAR' |
                         'CLK'
```

4 Semantics Description

The semantics rules are as described

1. Sections must be in order (DEVICES, CONNECTIONS, MONITOR)
2. 0 or more rules in each section
3. `uname`s in `CONNECTIONS` and `MONITOR` must be defined in `DEVICES`
4. Each device name must have the correct device switch and quantity (`bool` or `uint` or none) associated with it
5. `outname` and `inname` must be valid for specified `uname`
6. `uname`s must be unique
7. A keyword (section name, device type, initial condition, `outname` or `inname`) may not be a `uname`
8. In `CONNECTIONS`, each input must be connected to exactly 1 output
9. A comment must begin and end with a single forward-slash ('/'), and may run across multiple lines. (NB this is filtered by the scanner when extracting symbols)

5 Error Handling

The definition file is scanned by the scanner in response to a '`getsymbol()`' call from the parser. Each symbol is identified by the scanner to be one of the symbols defined in '`symbol.h`' That symbol is then instantiated as a symbol object by the scanner, and returned to the parser.

5.1 Syntax errors

- Picked up by the scanner and parser.
- Errors such as invalid characters which do not conform to any of the allowed symbol types cannot be instantiated as symbol objects, and thus are picked up by the scanner.
- Any symbols which are technically valid are passed to the parser which then conducts syntax checking.
- Each rule in the EBNF grammar is represented by a function in the parser class.
- Each function repeatedly calls '`getsymbol()`' on the scanner, and either tests the symbol directly if a terminal symbol is expected (e.g. '`DEVICES`', '`{`', etc.) or if a non-terminal is expected, calls the parser function which tests that non-terminal in a similar fashion.
- Errors are dealt with by raising exceptions. Every function in the parser has its own exception class(es) which it can raise to describe the various errors.
- Errors are reported in the required way; when an exception is raised, a description of the error, and its location in the definition file is returned, and a running total of errors incremented.
- An error occurring anywhere in a 'rule' line causes the rest of that line to be skipped (by finding the terminating semicolon). A rule line is any line defining a device, a connection or a monitor point.
- Thus functions which parse smaller components than a rule line simply raise a suitable error which is caught by the rule function, which then advances to the next semicolon before returning.
- The only higher level in the grammar than a rule line is the 'section'. A syntax error here means a bad keyword and/or incorrect braces ('`{`', '`}`') have been used. The error recovery method here is to ignore that entire section, i.e. advance to the next section keyword.

5.2 Error conditions which will be detected

1. `Bad char`: character is not allowed in the definition file (e.g. '`@`', '`£`', etc.)
2. `Bad uint`: number is out of range (<1, >16, not integer)
3. `Bad uname`: the `uname` is not valid (does not start with alphabetic character, is not otherwise alphanumeric)
4. `Expected a '.'`: the grammar requires a '.' which was not found
5. `Expected a ';'`: the grammar requires a ';' which was not found
6. `Expected a '='`: the grammar requires a '=' which was not found
7. `Expected a '>'`: the grammar requires a '>' which was not found
8. `Expected a '{'`: the grammar requires a '{' which was not found

9. `Expected a '}'`: the grammar requires a '}' which was not found
10. `Expected an uint`: the grammar requires a `uint` which was not found
11. `Expected an bool`: the grammar requires a `bool` which was not found
12. `Expected an inname`: the grammar requires an `inname` which was not found
13. `Expected an outname`: the grammar requires an `outname` which was not found
14. `Expected a device name`: the grammar requires a device name ('`CLOCK`', '`AND`', etc.) which was not found
15. `Expected a device switch`: the grammar requires an device switch ('`numinputs`', etc.) which was not found
16. `Expected a device`: the grammar requires a device which was not found
17. `Expected a uname`: the grammar requires an `uname` which was not found
18. `Expected keyword 'DEVICES'`: the grammar requires a keyword '`DEVICES`' which was not found
19. `Expected keyword 'CONNECTIONS'`: the grammar requires a keyword '`CONNECTIONS`' which was not found
20. `Expected keyword 'MONITOR'`: the grammar requires a keyword '`MONITOR`' which was not found

5.3 Semantic errors

Picked up by logic in the parser and by calling methods of the network and devices classes. The semantics in Section 4, p.3 are a complete list of the semantic errors which will be looked for.

6 Example definition files

The following are two examples of how a logic circuit definition file could be created to comply with the syntax and semantics defined previously.

6.1 First example

The circuit that the file below describes can be seen in Figure 6.1, p.7.

```
DEVICES
{
        SWITCH -initialvalue 0 = switch1;
        SWITCH -initialvalue 1 = switch2;
        SWITCH -intialvalue 1 = switch3;
        AND -numinputs 2 = and1;
        OR -numinputs 2 = or1;
}

CONNECTIONS
{
        switch1 > and1.I1;
        switch2 > and1.I2;
```

```
        switch2 > or1.I1;
        switch3 > or1.I2;
}

MONITOR
{
        and1;
        or1;
}
```

6.2 Second example

The circuit that the file below describes can be seen in Figure 6.2, p.7.

```
DEVICES
{
        CLOCK -period 5 = clock;
        SWITCH -initialvalue 0 = switch1;
        SWITCH -initialvalue 1 = switch2;
        SWITCH -intialvalue 0 = switch3;
        DTYPE = dtype;
}

CONNECTIONS
{
        switch3 > dtype.DATA;
        switch1 > dtype.SET;
        switch2 > dtype.CLEAR;
        clock > dtype.CLOCK;
}

MONITOR
{
        dtype.Q;
        dtype.QBAR;
        clock;
}
```

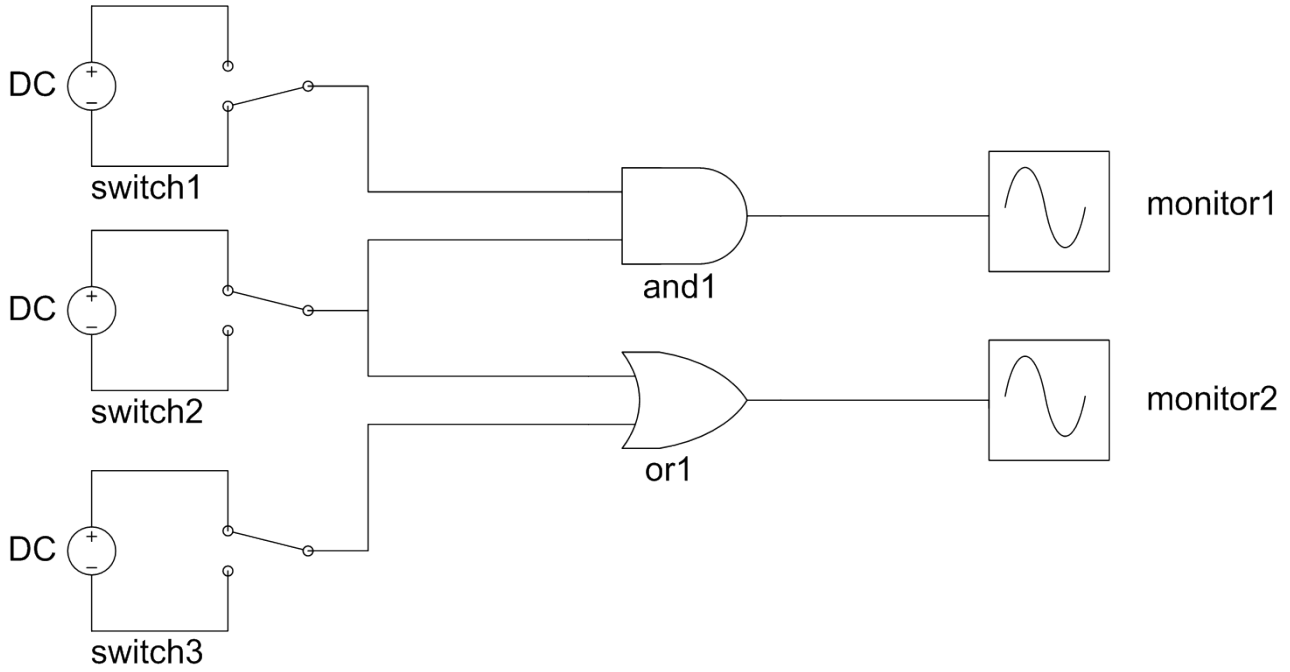

Figure 6.1: Example circuit 1

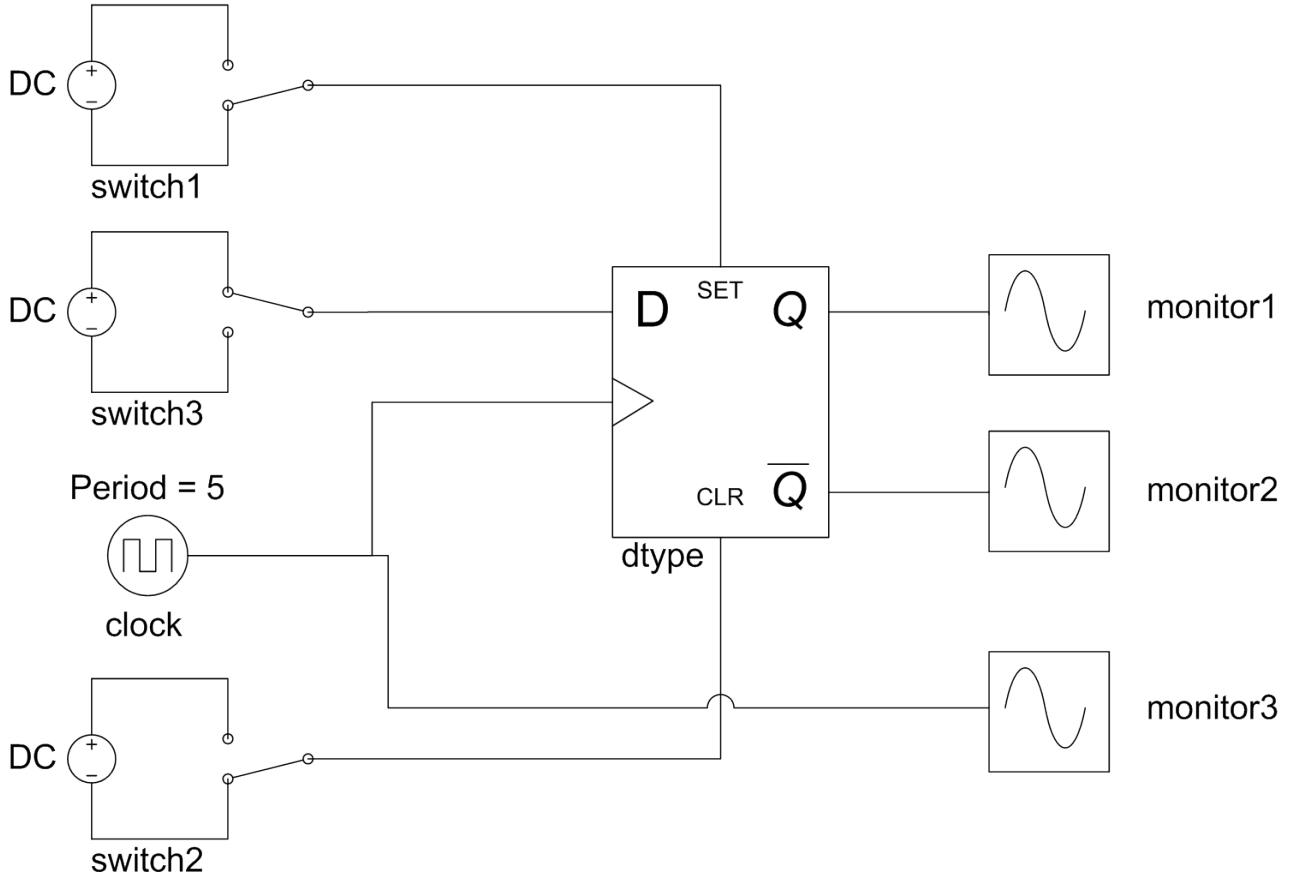

Figure 6.2: Example circuit 2