GF2 Interim Report 1

Joseph Roberts jr592
Daniel Potter djp73
Duncan Barber dab67

May 19, 2015

1 Summary

This project aims to build a piece of software to meet a 'client' specification describing a logic simulator. Notable features of the specification are:

- Simple and unambiguous syntax for defining the logic network
- Robust and informative error messages available at all stages
- Both a GUI and a command line interface to the simulator

The following report contains details of our intended approach to building the software, as well as our specification for the logic definition format.

2 General approach

We have chosen not to use the provided simulator source, in favour of our own. This provides several advantages, most prominent of which is that a network of components is now polymorphic with the base component, such that they can be nested within each other to an arbitrary depth. This is a key feature of the simulator and related file format, as repetitive blocks of logic can be written once and included many times.

We have also chosen to break up the file parsing into two distinct stages; one which interprets the file into an internal representation but extracts no meaning, and one which takes this internal representation and builds a network. This is a good idea since it keeps the logically separate tasks of parsing and construction apart. It also allows the builder to manage recursive calls to the parser to process included files correctly.

For unit testing, we are using a framework called Catch[1]. Catch is a header only library which supports a Behavior Driven Development style (Given-When-Then).

3 Teamwork planning

The work is roughly divided as follows:

Joe Roberts Simulator and network builder

Duncan Barber Definition parser

Daniel Potter GUI

[1]https://github.com/philsquared/Catch

4 File format

4.1 Overview and motivations

We reasoned that a logic simulator will primarily be used to simulate logic circuits of a reasonably large complexity, and so it would be prudent to design a file format which supports this use case well.

The format is a simple recursive key-value map, such that the 'value' can be another nested map. This allows for arbitrarily complex data to be represented, while keeping the format exceptionally easy to parse. Since the syntax doesn't encode any meaning, it should be read alongside a schema representing recognised fields.

The only root key required is the one defining components, all others are optional. The schema includes an 'includes' field where other definitions can be included and aliased, to be used in the definition just as though they were base components.

4.2 EBNF description

```
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g"
       | "h" | "i" | "j" | "k" | "l" | "m" | "n"
       | "o" | "p" | "q" | "r" | "s" | "t" | "u"
       | "v" | "w" | "x" | "y" | "z"
       | "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
character = letter | digit | "_" ;

identifier = character, { character } ;
value = identifier, [ ".", identifier ] ;
string = "\"", { character | whitespace | punctuation - "\""}, "\"";

pair  = (identifier | string), ":", (value | dict | string);
dict = "{", [pair, {",", pair}], "}";

comment = "/*", { character | whitespace | punctuation }, "*/" ;

grammar = dict; (* Definition file has exactly one root dict *)
```

4.3 Format schema

The schema takes the form of a pseudo-definition file. Square brackets indicate that a field is optional, and regular parentheses contain possibilities for a field.

```
{
  description : "A description of the network",

  includes : {
    "/absolute/path/to/include" : include_nickname,
    "relative/path/to/include" : include_nickname
  },

  inputs : {
    input_nickname : initial_value (true|false|t|f|1|0)
  },
  outputs : {
    output_nickname : dest_component_nickname[.dest_output_name]
  },
  monitor : {
    monitor_point_nickname : dest_component_nickname[.dest_output_name]
  },

  components : {
    component_nickname : {
      type : component_type (and|or|nand|nor|xor|dtype|siggen|includes.include_nickname),
      { config_param : config_data, }
      inputs : {
        input_name : dest_component_nickname[.dest_output_name]
      }
    }
    component_nickname : {
      type : includes.include_nickname,
      inputs : {
        input_name : dest_component_nickname[.dest_output_name]
      }
    }
  }
}
```

4.4 Possible syntax errors

The table below gives some examples of syntax errors which will have to be safely handled by the programme. All syntax errors will be unrecoverable, and hence the user will have to be informed of the error. The possible syntax errors are contained in Table 1.

#	Description
1	Mismatched delimiters
2	Invalid character in an identifier
3	Colon missing from a pair definition
4	Equals sign used in place of colon in a pair definition
5	Comma missing after non-final item in a dict
6	Comma after final item in a dict
7	Identifier missing from left hand side of a pair
8	More than one full-stop in a value

Table 1: Possible syntax errors

4.5 Possible semantic errors

Due to the simplicity of the syntax, most of the possible errors which can occur are semantic errors. The errors which follow are divided roughly into two catagories, recoverable and unrecoverable. Unrecoverable here means that there is resulting ambiguity in the definition, or that the definition does not have meaning. Recoverable errors are still technically errors, though it is still possible to extract meaning from the definition and simulate it. All semantic errors present will be reported where possible. All semantic errors will be discovered during the Network construction, after the file is successfully parsed. The possible semantic errors are contained in Table 2.

The handling of many of these errors is built into the rewritten simulator, such as unconnected inputs and invalid input/output/component names. Of the above the most difficult to detect will be an include loop, though this shouldn't be too difficult with a depth first search through the include tree.

#	Description	Type
1	Root components field missing	Unrecoverable
2	Any of the other root field missing	Recoverable (since no ambiguity)
3	An included file does not exist/cannot be accessed	Unrecoverable, unless the include is not used
4	Included definitions form a directed cycle	Unrecoverable
5	Initial input value not valid	Recoverable, treat as false
6	input_name doesn't exist	Recoverable, ignore it
7	Input of a gate left unconnected	Recoverable, treat as unconnected/connected to false
8	same input_name is connected more than once in the same component block	Unrecoverable
9	dest_component_nickname doesn't exist	Recoverable, treat as unconnected/connected to false
10	dest_output_name is not an output of dest_component_name	Recoverable, treat as unconnected/connected to false
11	Not appending [.dest_output_name] where there are multiple outputs	Recoverable, treat as unconnected/connected to false
12	component_type is not a valid component or include	Unrecoverable
13	Same component_nickname appears twice	Unrecoverable
14	Same input_nickname appears twice	Unrecoverable
15	Same output_nickname appears twice	Unrecoverable
16	Same monitor_point_nickname appears twice	Recoverable, just rename one of them
17	A config_param is not included in a component which takes it	Depends on the specific component.
18	An unrecognised config_param is included	Recoverable - ignore it

Table 2: Possible semantic errors

5 Example definition files

5.1 JK flip flop

```
{
  description : "A standard JK Flip Flop made from NAND gates",
  inputs : {
    J : false,
    K : false,
    clock : false
  }
  outputs : {
    Q : nand3,
    Qbar : nand4,
  }
  components : {
    nand1 : {
      type : NAND,
      inputs : {
        i1 : inputs.J,
        i2 : inputs.clock,
        i3 : nand3
      }
    }
    nand2 : {
      type : NAND,
      inputs : {
        i1 : inputs.K,
        i2 : inputs.clock,
        i3 : nand4
      }
    }
    nand3 : {
      type : NAND,
      inputs : {
        i1 : nand4,
        i2 : nand1,
      }
    }
    nand4 : {
      type : NAND,
      inputs : {
        i1 : nand3,
        i2 : nand2,
      }
    }
  }
}
```

5.2 Synchronous counter

```
{
  description : "A 4-bit synchronous counter, which avoids the ripple effect.",
  includes : {
    jkflipflop.def : JK
  },
  inputs : {
    clock : false
  },
  outputs : {
    q0 : jk1.Q,
    q1 : jk2.Q,
    q2 : jk3.Q,
    q3 : jk4.Q
  },
  components : {
    jk1 : {
      type : includes.JK,
      inputs : {
        J : constants.true,
        K : constants.true,
        clock : inputs.clock
      }
    }
    jk2 : {
      type : includes.JK,
      inputs : {
        J : jk1.q,
        K : jk1.q,
        clock : inputs.clock
      }
    }
    jk3 : {
      type : includes.JK,
      inputs : {
        J : and1,
        K : and1,
        clock : inputs.clock
      }
    }
    jk4 : {
      type : includes.JK,
      inputs : {
        J : and2,
        K : and2,
        clock : inputs.clock
      }
    }
    and1 : {
      type : and,
      inputs : {
        in1 : jk1.q,
        in2 : jk2.q
      }
    }
    and2 : {
      type : and,
      inputs : {
        J : and1,
        K : jk3.q,
        clock : inputs.clock
      }
    }
  }
}
```

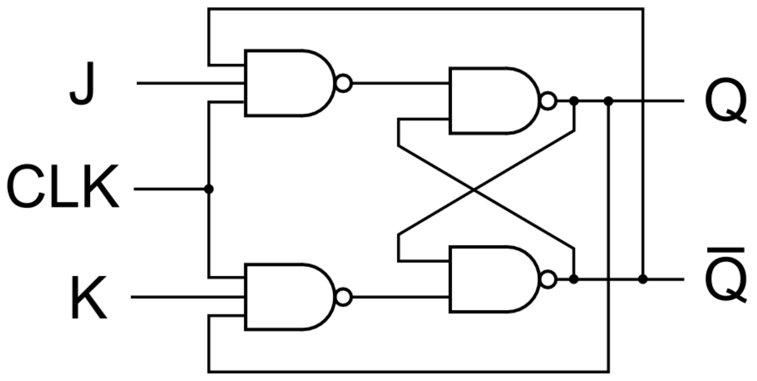

Figure 1: J-K Flip-flop Schematic

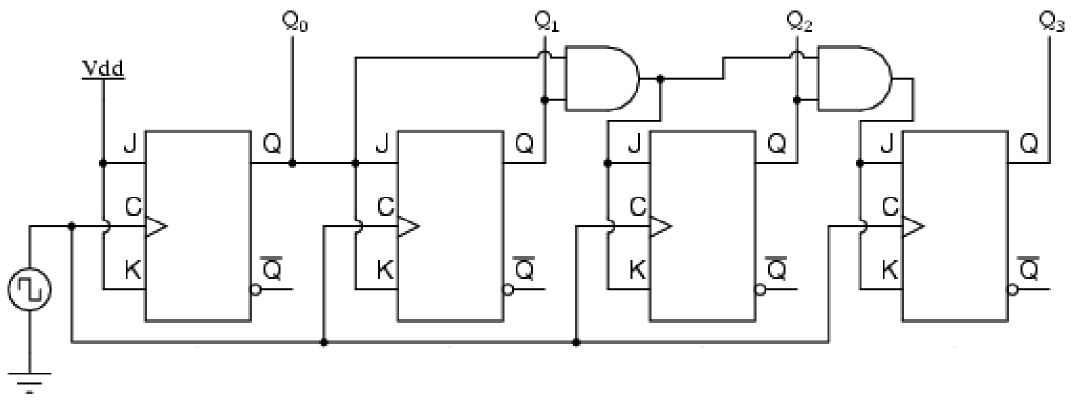

Figure 2: Synchronous Counter Schematic