Milestone 2: Implementation

Team Members

- Joseph Antonakakis, jma353
- Natasha Armbrust, nka8
- Celine Brass, cjb327
- Reuben Rappaport, rbr76

System Description

Summary

We created a new functional hardware description language for digital logic with an OCaml interpreter that simulates the behavior of the circuit and an interactive GUI application that lets the user control the simulation.

Vision

The two most popular hardware description languages (programming languages used to describe the behavior of electrical circuitry) are Verilog and VHDL. Unfortunately, both of these are imperative languages, and the imperative paradigm lends itself poorly to describing hardware. Verilog in particular runs into serious issues. The syntax is such that the vast majority of possible Verilog programs are impossible to translate into real world circuitry! To successfully design in Verilog one must first picture the circuit one wants and then attempt to construct code that will generate it - obviously this results in a design process that's exactly backwards.

Although analog circuitry is a good deal more complicated, all digital circuitry is at its core just a state machine. That is, it consists of registers which hold the state and combinational logic connecting them which serve as a transition function every time the clock ticks. This setup is far better suited to being described by a functional language than an imperative one.

For this project we plan to define the formal syntax, and semantics of a new functional hardware description language for digital logic. Then, to make it a usable digital design tool we'll write an OCaml interpreter for the language that simulates the behavior of the described circuit. Finally, we'll build a GUI application to let users interact with the simulation by visualizing their circuit, controlling the values of inputs, and stepping and running the clock.

Featureset

For this project we: - Formally defined and documented a new functional hardware description language (found under documentation/Language.pdf) - Wrote an OCaml interpreter for our new language that simulates the behavior of the defined circuit - Built a GUI based application that allows the user to interact with the simulation - The GUI allows the user to view a visualization of their circuit - The GUI allows the user to step the circuit - The GUI allows the user to control the values of inputs to the circuit

Architecture

Since our project is essentially a compiler for hardware, we chose a pipe and filter architecture. Source files are first passed into the Lexer to create a token stream, and then into the parser to build the AST. From there, this is passed to the Simulator, which simulates circuit behavior and to the GUI which renders the circuit and reads user commands. When a user inputs a command, the GUI passes it back to the simulator to update the state, and then reads the new state and rerenders.

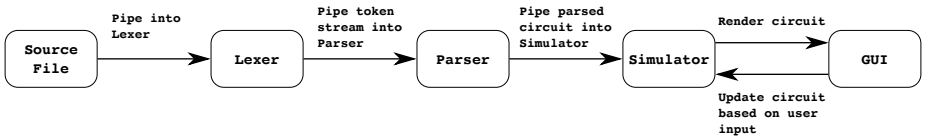

Figure 1: C&C Diagram

System Design

We split our design up into a number of coherent submodules:

Bitstream

This module contains a data type to represent an indexed collection of bits (ie something along the lines of `4'b0010`). It also provides a large collection of functions to create and manipulate bitstreams (arithmetic, bitwise operations, logical operations, concatenation, etc).

Combinational

This module contains a data type to represent a combinational logic expression as an abstract syntax tree. References to the values of inputs and registers are

done using their id - when the simulator goes to evaluate this AST it will simply look them up in its environment.

For example, the expression `A[2] | B[1] & 1'x1` would be represented as

```
Result (Gate (Or, Nth (2, Var "A"), Gate (And, Nth (1, Var "B"), Const 1'b1)))
```

Circuit

This module contains several submoduls that use the internals of the circuit data structure.

Simulator This submodule contains a data type to represent a circuit as a collection of registers storing state. Each register has an associated combinational logic expression which is used to compute its next value when the clock steps. In order to make matters simple, internally we represent inputs and outputs as circuits also. We define the following rules for updating registers:

- A falling register updates its value to the result of evaluating its AST when the clock falls
- A rising register updates its value to the result of evaluating its AST when the clock rises
- An output updates its value to the result of evaluating its AST when anything else changes
- An input changes its value whenever a user inputs a new value for it

Under this scheme, to change the value of an input, we first set the input's value to the new one, and then recompute all of the outputs.

This module provides functions for stepping the circuit, and for evaluating combinational expressions in the context of a circuit.

Analyzer This submodule contains static analysis functions which, given a circuit determine whether or not it is valid and if it is not output a log documenting the problems with it. Any circuit that passes static analysis should encounter no errors when stepping in the simulator.

Formatter This submodule provides an intermediate data structure for representing circuits that's easy to render as an SVG, and an array of functions for converting our internal circuit representation to this renderable format.

In a formatted_circuit, all registers are assigned a column depending on the other registers they depend on. For example, in the circuit `C = A & B`, Registers A and B would be assigned column 0 and Register C would be assigned column 1.

Lexer

This module is an OCamllex lexer, based in part off the one provided in A4: OCalf. We use it to convert source files into token streams.

Parser

This module is an Menhir parser, based in part off the one provided in A4: Ocalf. It provides parsing functions both for pure combinational expressions and for circuit definitions.

Parse

This module is an OCaml wrapper that makes it easy to pass strings and files through the Lexer -> Parser pipeline.

GUI

This module drives the GUI. It uses the intermediate data structures defined in the Formatter module to render circuits. It allows a user to compile a circuit, change the inputs of the circuit by clicking on the input-register's bit values, and allows the user to step-through the circuit via clock-changes. See GUIDoc.pdf for instructions on interacting with the GUI.

Model This module houses the entire state of that front-end application (current compiled circuit and clock value). It has functions to abstract away all state-changes.

View This module generates a circuit / the initial GUI and sets up the view to appropriately call controller methods (see **Controller**) to augment the state of the app and facilitate view-changes as necessary.

Controller This module connects view and model, calling model state-changing functions and initiating view changes based on view-altering functions that it is provided.

Components This module contains a suite of functions that generate SVG elements for circuit components (gates, registers, arithmetic, etc.)

Extensions This module extends the `D3` module that the front-end leverages, increasing the amount of OCaml-to-JavaScript bindings that the front-end can use.

Module Dependency Diagram

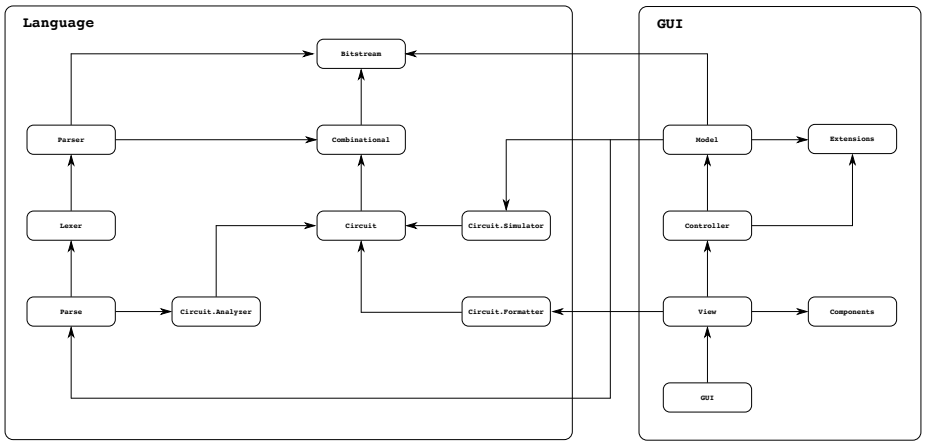

Figure 2: MDD

Module Design

See the interfaces for each module in `src.zip`

Data

We defined several data structures to use in our project.

Bitstream

A `bitstream` is a collection of boolean values indexed from least to most significant bit. Internally, this is implemented as an `array` to leverage the $O(1)$ indexed access time. The module maintains the representation invariant that once constructed, a bitstream can never be modified so although it is technically a mutable data structure, it is always treated as though it were immutable.

Combinational

A `comb` is an AST representing a combinational logic expression. We define it as:

```
type id = string

type gate =
| And | Or | Xor | Nand | Nor | Nxor

type negation =
| Neg_bitwise | Neg_logical | Neg_arithmetic

type comparison =
| Lt | Gt | Eq | Lte | Gte | Neq

type arithmetic =
| Add | Subtract | Sll | Srl | Sra

type comb =
| Const     of bitstream
| Var       of id
| Sub_seq   of int * int * comb
| Nth       of int * comb
| Gate      of gate * comb * comb
| Logical   of gate * comb * comb
| Reduce    of gate * comb
| Neg       of negation * comb
| Comp      of comparison * comb * comb
| Arith     of arithmetic * comb * comb
| Concat    of comb list
| Mux2      of comb * comb * comb
| Apply     of id * comb list
| Let       of id * comb * comb
```

Register

A `register` is a digital state component. It consists of a value paired with an AST to evaluate when the clock steps to compute the next value.

```
(* since we internally represent inputs and outputs as registers, we need a
* flag to specify their type *)
type reg_type =
| Rising | Falling | Input | Output

type register_input =
| User_input | AST of comb

(* a digital state component *)
    type register = {
```

```
    reg_type : reg_type;
    length : int;
    value : bitstream;
    next : register_input;
}
```

Subcircuit

A `subcircuit` is a digital state component. It consists of a value paired with an AST to evaluate when the clock steps to compute the next value.

```
type subcircuit = {
    ast : comb;
    length : int;
    args : (id * int) list;
}
```

Component

A `component` is a `register` or a `subcircuit`. It is used in our circuit map to represent the internal state of a circuit.

Circuit

A `circuit` is our top level data structure for representing state. It consists of a clock value (either high or low) and a map of components. From `ids` to `components`. When we step the circuit we compute the new value of each register by evaluating its AST in the context of the current circuit.

Formatted Circuit

A `formatted_circuit` is an intermediate representation of a circuit with extra information added in to make rendering easy.

```
type connection = Reg of id | Node of int | Let of id

type node =
    | B of gate * connection * connection
    | L of gate * connection * connection
    | A of arithmetic * connection * connection
    | N of negation * connection
    | C of comparison * connection * connection
```

```
    | Sub of int * int * connection
    | Nth of int * connection
    | Red of gate * connection
    | Concat of connection list
    | Mux of connection * connection * connection
    | Const of bitstream
    | Apply of id * connection list

type display_reg_type = Dis_rising | Dis_falling | Dis_input | Dis_output

type display_node = {
    n_id : int;
    n_x_coord: float;
    n_y_coord: float;
    node : node;
}

type display_let = {
    l_id : id;
    l_x_coord:float;
    l_y_coord: float;
    inputs: id list;
}

type display_register = {
    r_id : id;
    r_reg_type :  display_reg_type;
    r_x_coord : float;
    r_y_coord : float;
    input : connection;
}

type formatted_circuit = {
    registers : (id * display_register) list;
    nodes     : (int * display_node) list;
    lets      : (id * display_let) list
}
```

Language Definition

Please refer to a fully written Language documentation under `documentation/Language.pdf`. This document outlines our language and provides interesting example cases.

GUI behavior

Please refer to the GUI reference document under `documentation/GUIDoc.pdf`

External Dependencies

We will be using Js_of_ocaml, D3, Deriving, OcamlLex, and Menhir. The latter two, will allow us to easily lex and parse source files, the former will make building an interactive GUI significantly less challenging by binding our OCaml code to javascript.

Testing

Testing Plan

For testing, we used a mix of unit-, integration-, and hand-testing techniques to ensure our project works.

Our `Bitstream` module was unit-tested via the `Test_bitstream` module to ensure that the representations of bitstreams and input values are well-constructed and correct. The functions in this module were unit-tested on a per-function basis, and its testing ensured that this utility was correct for use in other modules.

Our simulator was tested via unit tests defined in `Test_simulator`. These unit tests integrated the `Parser`, `Lexer`, `Circuit`, and `Combinational` modules. In the tests we created different circuits in our language, changed the inputs of the circuit, and stepped the circuit in order to test the language generation and simulation. We probed register values in order to make sure all registers were being updated correctly based on our simulation.

Our `Formatter` module was tested via unit tests defined in `Test_formatter`. In these tests, we created circuits and tested against the expected values the formatter should have. We probed the resultant data-structure generated by the `Formatter` to ensure that the formatting is expected and sound based on the circuits we're creating.

We hand-tested our GUI (the product of our `Gui` module) with tests found in `test_gui`. These tests involved testing creating circuits on our gui, chaning inputs, and stepping them. The tests go through all features in our language including simple register values, basic gate, arithmatic, comparators, functions, substreams, concatanation, and lets as well as long examples and edge cases.

Results of Testing - Known Bugs and Limitations

- In particulary complicated circuits, wires occasionally overlap in a way that's difficult to read

- There's no way to view the internals of a subcircuit. We would have liked to add this but we ran out of time and the GUI is a fully functional top level view of the circuit without this feature.
- There's no way to define a mux that's larger than two inputs. We would have liked to add this feature to the language but we ran out of time. Luckily this can be approximated by a routing network of 2 to 1 muxes.
- Long register names result in the text overlapping the wires.
- The maximum bitstream length is 32 bits. This is a little shorter than is ideal.
- There's no way for a subcircuit to have multiple outputs.

Division of Labor

Reuben designed the language syntax and wrote up most of the language reference found in `documenation/language.pdf`. He wrote the lexer, the parser and the static analyzer as well as their test suites. He designed the data types in `Combinational` and `Circuit` When we decided to change the implementation of `Bitstream` to obtain $O(1)$ access time he rewrote the module and its test suite. He estimates he spent ~60 hours on the project.

Natasha wrote the Simulator and the unit tests for the simulator. She also wrote the GUI tests and helped with the language documentation and MS1/MS2 documentation. She wrote the `Circuit.Simulator` and `Bitstream`. However, `Bitstream`was re-written by Reuben after the group decided we wanted an $O(1)$ access time and a 32 bit maximum on the registers. She esitmates she put in ~40 hours on the project between planning, designing, implementing, and testing.

Celine wrote the Formatter and the tests for the Formatter. This module went through several iterations as the group learned the limitations the GUI would present. Celine worked with Joe extensively to design an appropriate intermediate representation of a circuit (the formatted_circuit) that took as much processing responsibility as possible from the GUI itself. This helped preserve the MVC design. She estimates she put in ~40 hours on the project between planning, designing, implementing, and testing.

Joe wrote the entire front-end at the top level of the project. He utilized an open-source project, `ocaml-d3` and extended its functionality to help bind OCaml to the JavaScript library `D3` (extensions found in `Extensions`). He wrote a suite of SVG elements in `Components` representing all essential components of the circuits that can be expressed via CAMLWare. He utilized the `Model-View-Controller` design pattern to establish a web interface, built-entirely in OCaml, that allows one to compile a written circuit in-browser, change inputs to that circuit, step through the circuit's progression, and recompile as necessary. Modules contributing to the `MVC` setup are `Model`, `View`, and `Controller`. `Model` handles state changes, `View` parses a clean intermediate representation of the circuit (involving fundamental circuit display info), generated by Celine, into a visible circuit,

and `Controller` binds the two together, calling view-augmenting functions after facilitating model state-changes. Joe, with Reuben's help, set up all compilation configurations at the top level, to transpile the entire project into a JavaScript file that can run in a browser. He estimates he spent ~60 hours on the project.