Milestone 1: Design

Team Members

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

System Description

Summary

We plan to create 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 plan to: - Formally define and document a new functional hardware description language - Write an OCaml interpreter for our new language that simulates the behavior of the defined circuit - Build a GUI based application that allows the user to interact with the simulation - The GUI will allow the user to view a visualization of their circuit - The GUI will allow the user to step and run the clock - The GUI will allow 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.

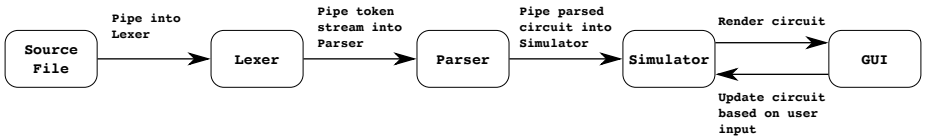

Figure 1: C&C Diagram

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 `0b0010`). 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] & 0x1` would be represented as

```
Gate (Or, Nth (2, Var "A"), Gate (And, Nth (1, Var "B"), Const 0x1))
```

Circuit

This module 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.

Base_conversions

This module provides utility functions for converting between string and integer representations of decimal, binary and hexadecimal bases.

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 Ocamlyacc parser, based in part off the one provided in A4: Ocalf. It provides parsing functions both for pure combinational expressions and for circuit definitions. For a description of the syntax involved, see the **Language Definition** section.

Parse

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

Formatter

This module 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.

The formatter also assigns wiring positions between circuits and gate placement. All placements are given in percents rather than pixels, etc to make for a more flexible display.

Renderer

This module drives the GUI. It uses the intermediate data structures defined in the Formatter module to render circuits. When neccesary, it passes along user input to the simulator, before reformatting and rerendering the result.

Module Dependency Diagram

Module Design

See `interfaces.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 a `list`.

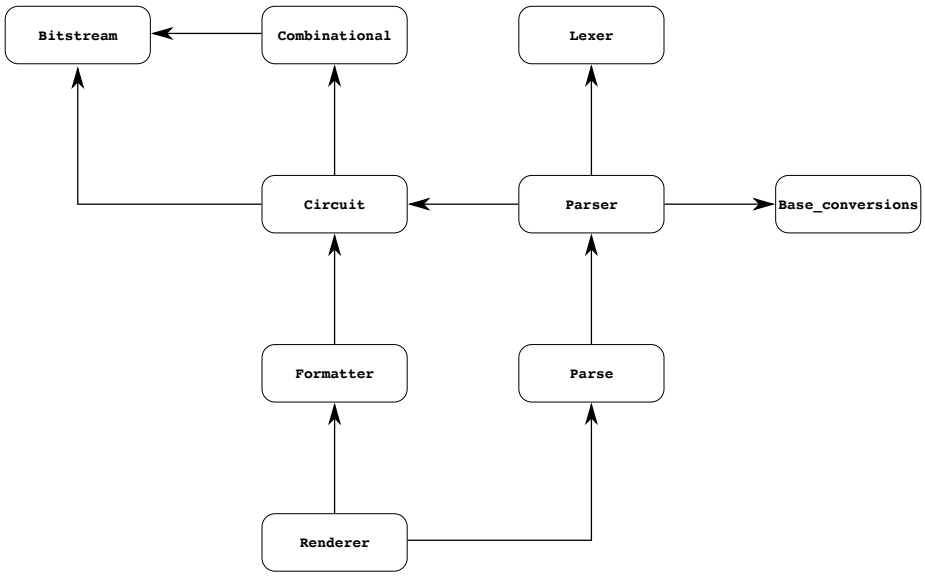

Figure 2: MDD

Combinational

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

```
(* the type of the keys to which registers are bound *)
type id = string

(* the digital gates *)
type gate =
  | And | Or | Xor | Nand | Nor | Nxor

(* the types of negation *)
type negation =
  | Neg_bitwise | Neg_logical | Neg_arithmetic

(* the types of comparison *)
type comparison =
  | Lt | Gt | Eq | Lte | Gte | Neq

(* the types of supported arithmetic *)
type arithmetic =
  | Add | Subtract | Sll | Srl | Sra

(* values of type [comb] represent combinational logic circuits.
```

```
 * - [Const b] represents a constant value containing bitstream [b]
 * - [Reg id] represents the value of the register with id [id]
 * - [Sub_seq from to b] represents the subsequence of bitstream [b] from index
 * - [Nth n b] represents the [n]th bit of [b]
 *   [from] to index [to]
 * - [Gate g b1 b2] represents gate [g] applied bitwise to [b1] and [b2]
 * - [Logical g b1 b2] represents gate [g] applied logically to [b1] and [b2]
 * - [Reduce g b] represents [b] reduced with gate [g]
 * - [Neg negation b] represents [negation] applied to [b]
 * - [Comp comp b1 b2] represents [comp] applied to [b1] and [b2]
 * - [Arith op b1 b2] represents [op] applied to [b1] and [b2]
 * - [Concat b1 b2] represents [b1] concatenated to [b2]
 * - [In] represents a value that is controlled by a user
 *)
type comb =
  | Const     of bitstream
  | Reg       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 * comb
  | In
```

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

(* a digital state component *)
type register = {
  reg_type : reg_type;
  length : int;
  value : bitstream;
  next : comb;
}
```

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 from `id`s to `register`s. 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 (eg locations) added in to make rendering easy.

Language Definition

The following is a formal definition of our language in Backus-Naur form. A source file consists of a sequence of variable definitions

```
definition := (* a variable definition *)
  | [rising | falling] register var[number] = comb;
  | output var[number] = comb;
  | input var[number];

comb := (* combinational logic *)
  | add_expr

add_expr := (* addition expression *)
  | shift_expr
  | add_expr + shift_expr (* addition*)
  | add_expr - shift_expr (* subtraction *)

shift_expr := (* shift expression *)
  | or_expr
  | shift_expr << or_expr  (* shift left logical *)
  | shift_expr >> or_expr  (* shift right logical *)
  | shift_expr >>> or_expr (* shift right arithmetic*)

or_expr := (* or expression *)
  | and_expr
  | or_expr | and_expr  (* bitwise or *)
  | or_expr ^ and_expr  (* bitwise xor *)
  | or_expr ~| and_expr (* bitwise nor *)
  | or_expr ~^ and_expr (* bitwise nxor *)
  | or_expr || and_expr (* logical or *)
```

```
and_expr (* and expression *)
  | comp_expr
  | and_expr & comp_expr  (* bitwise and *)
  | and_expr ~& comp_expr (* bitwise nand *)
  | and_expr && comp_expr (* logical and *)

comp_expr := (* comparison expression *)
  | unary
  | comp_expr == unary (* equals *)
  | comp_expr != unary (* not equals *)
  | comp_expr > unary  (* greater than *)
  | comp_expr < unary  (* less than *)
  | comp_expr >= unary (* greater than or equal to *)
  | comp_expr <= unary (* less than or equal to *)

unary := (* unary expressions *)
  | array_access
  | ~unary  (* bitwise negation *)
  | !unary  (* logical negation *)
  | -unary  (* arithmetic negation *)
  | &unary  (* reduction with and *)
  | |unary  (* reduction with or *)
  | ^unary  (* reduction with xor *)
  | ~&unary (* reduction with nand *)
  | ~|unary (* reduction with nor *)
  | ~^unary (* reduction with nxor *)

array_access := (* array access expressions *)
  | primary
  | primary[number]          (* access nth bit *)
  | primary[number - number] (* access subsequence *)

primary := (* a primary expression *)
  | var          (* a variable reference *)
  | number       (* a constant *)
  | (comb)       (* an expression in parentheses *)
  | {comb, comb} (* concatenation *)
```

External Dependencies

We will be using Js_of_ocaml, OcamlLex, OcamlYacc. 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 Plan

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

Our `bitstream` and `base_conversion` modules will be unit-tested to ensure that their representations of bitstreams and input values are well-constructed and correct. The functions in these modules will be easy to unit-test on a per-function basis, and will ensure that these utilities are correct for use in other modules.

Our `combinational` module primarily deals with the representation of the AST nodes, so it will be involved in the full-system, integration-tests of our project.

Our `circuit` module will be involved in the evaluation of our circuits / the persistance of results in registers based on clock-ups and and clock-downs. This module can be involved in the full-system, integration-tests of our system. We will be able to check register values on clock changes to detect proper and expected processing of circuit logic.

Our `lexer` and `parser` modules will help build the AST that is eventually evaluated in our integration-tests. These modules will also be unit-tested in terms of the tokenization of elements of our language, as well as the construction of the physical AST. We can probe our AST to ensure that it's physically constructed properly according to the expected parsing of our language.

Our `formatter` will be tested on circuits that we are sure are constructed properly according to our AST. We can probe 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 plan on hand-testing our GUI (the product of our `render` module) as well as possibly involving an automation / browser technology like Selenium to automated the testing of interactions and HTML components involved in our GUI representation.