Legend Of Bluespec

kr469

October 2021

Contents

Chapter 1

Introduction

Making hardware is hard, and similarly to other problems in computer science, we make it easier by adding layers of abstraction. There are many layers involved in making hardware from that abstract transistors to gates to logical operations to functions to modules to chips to multi-chip modules to whole devices. The layer this project focuses on is going from modules to chips(TODO: this probably need renaming). At this layer we already have code for modules like memory, CPU-cores, interconnects and so on. Now what we need is an ability to compose them in such a way that they create a whole device. On a paper this should be a simple task but in reality it's a lot more complicated.

1.1 Reality

Creating custom hardware is getting more expensive both in therms of money but also in terms of man-hours needed to create it. This creates barriers to entry for new players, and also reinforces strong positions of already adopted standards. Two monoliths that I will focus on are language Verilog and design software Intel Quartus Prime(IQP). In my work I will work with Bluespec language that can be compiled down to Verilog.

Market share and justification for focusing entirely on comparisons with Intel Quartus Prime

It's a bit difficult to accurately judge market share of IQP, but It is produced by one of the biggest chip manufacturer and designer in the world, and it's also used by Qualcomm according to HG Insights. Using Google trends tool we can see that while in recent years(since 2016) competitor Xilinx Vivado has overtaken, IQP in interest at the global scale, in most developed countries like US, Europe, and parts of Asia. There is roughly 50/50 split between IQP and Vivado. My personal observations suggest that Xilinx Vivado has been more popular in India which is a large country that is currently developing rapidly. Therefore, while I'm going to assume that IQP is still one of two biggest hardware design platforms, and it's fair to not investigate Xilinx's software during evaluation(TODO: check if this has changed).

My project will try to tackle a subset of functionality provided by a tool called Platform Designer that is a part of Intel Quartus Prime package. Platform Designer is a GUI tool for connecting modules, it is capable of saving and loading designs, from proprietary plain text file format. Unfortunately, those files are not exactly what one might call "Human readable" as they have a tendency to be megabytes long(millions of characters). This tool also have some other pains that will be explained later.

1.2 Solution

During my project I will try to create an alternative file format that is much simpler and allows for editing by a human. To do this I will harness power of types in Bluespec language. My tool will be able to operate

on Bluespec packages, but thanks to other tools that allow for interoperability of Bluespec and Verilog, it should be still technically possible to use my tool with wider Verilog ecosystem.

Chapter 2

Preparation

2.1 Understanding Bluespec

To begin working with Bluespec we first need to understand the language.

2.1.1 Rules

In Bluespec all computation is done in form of rules. Each cycle we will take a subset of all rules that we are going to execute in this cycle, rule is fired (executed) in cycle only if it's ready (or will be ready) and it's not conflicting with other rules (If this happens compiler must issue a warning, and picks arbitrary rule to fire from subset of conflicting rules). Each rule can fire at most one time per cycle. For rule to be ready to fire it needs it's implicit and explicit conditions to be true. Rule can be fired in a cycle even if it's not ready at the start of cycle, for example if you add item on an empty queue and then pop can happen in same cycle.

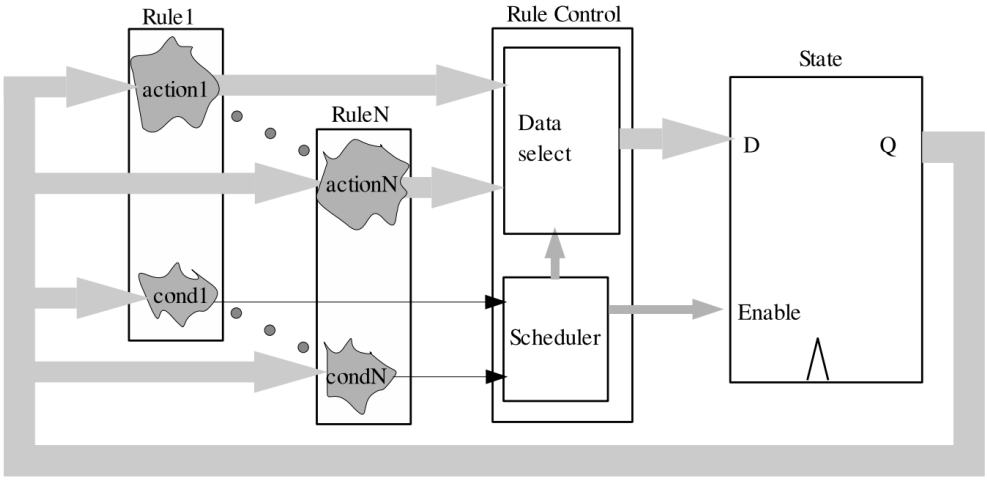


Figure 1: A general scheme for mapping an N-rule system into clocked synchronous hardware.

(TODO: add reference to the bsv reference document from which this image was taken)

```
TODO: piece of Bluespec with module using fifo and other rule
 explaining types of conditions.
```

2.1.2 Modules and interfaces

Module is something (TODO: I have no Idea how to call it). Modules don't have types, instead they implement some interface. This means that you can have multiple different modules that implement the same interface, this makes interoperability much easier. Interfaces are made up out of two things:

- Methods - that allow for interaction with the module.
- Subinterfaces - That allow for more generalization, for example you need to connect one more thing you don't need to change an interface used by other modules, you can just create a new interface that contains two subinterfaces and pass them accordingly.

2.1.3 Types

Welp, there is a ton of them.

2.1.4 Typeclasses

Types classes in Bluespec are used to group types for which specific functions are implemented. TODO:

2.2 Creating a grammar

2.2.1 Why grammar is needed ?

I had effectively 3 choices for a language to write this grammar in:

- Haskell - This would require me to directly tap into complier for information about modules and ect. in packages, and I would also need to learn Haskell effectively from scratch. I hope that I don't need to explain why learning Haskell while trying to understand complier of another language is a bad idea.
- Tcl (pronounced "tickle") - This language is used as a scripting language in both Inter and Xilinx tools, I will be reading packages using Tcl scripts provided by the creators of the Bluespec complier (BSC), but my understanding is that those scripts are just handy wrappers for some Haskell code. This is also foreign language to me with minimal presence, and negligible learning resources.
- Python - Firstly this is a language I have experience working with, secondly it's widely supported, and there is extensive tooling for it. It's flexible typing system allows for rapid experimenting.

I have chosen Python for this project as I didn't want to dabble in the BSC as I was advised that this is dangerous for part II project as compliers are overwhelmingly complex and difficult to understand. This meant that I will need to parse output of Bluetcl (complier script for inspecting packages), and to do this I'm going to need grammar.

2.2.2 What grammar is needed ?

Bluetcl produces many, outputs but two of them that we are going to focus on are: descriptions of functions and descriptions of types. To simplify parsing of those I will have one grammar capable of parsing both outputs at the same time, as some grammar structures are reued in both outputs.

2.2.3 Where to find this grammar ?

Unfortunately this grammar is not documented anywhere, so I needed to reverse engineer it. This grammar might not be perfect and might not cover every input, but if created carfuly to allow for as much flexibility as possible, and use of large body of test input in from of standard library during creation of it we should be exposed to enough examples to be able to parse decently large subset of future inputs. Here are other reasons to justify this approach:

- Heaps' law suggests that number of unique words in given body of text is proportional to roughly square root of number of words in the text. I think it's fair to assume that something similar will be true if we consider number of unique grammar rules.
- This grammar while different from grammar of Bluespec language maps subset of Bluespec grammar, so we can supplement our deductions with cases that we expect to arise.
- We don't need to understand everything to just connect modules, and because at the module level, things need to be less abstract as they need to be synthesizable, we don't expect highly exotic things to appear in higher level modules.
- This approach is probably the best way for me anyway.

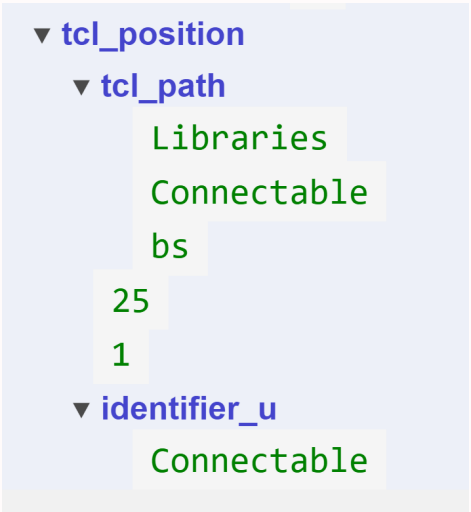
2.2.4 Technical aspects

This is EBNF grammar, I parse it using Lark library for Python, and I'm using Earley parser, as it is capable of arbitrary length lookahead. Grammar I created contains roughly 90 rules, and I won't include all of them here, but I will show few examples to give a feel of what is happening.

Parsing position TODO maybe find a better example with shorter line

```
tcl_position: "{" "position" "{" tcl_path NUMBER NUMBER ["{" "Library" identifier_u "}" ]"}""}"
// todo check paths with spaces
tcl_path: ["%/"] /(([.]{1,2})|(\w+))/ ["/" /(([.][.])|([.])|(\w+))/]* "." /\w+/

-------- Text to parse ---------
{position {%/Libraries/Connectable.bs 25 1 {Library Connectable}}}
```



A nice feature supported by Lark is ability to have regular expressions in the grammar, I'm mentioning this as is effectively having parser in side of parser. A handy tool for debuging and creating grammar was this website: https://www.lark-parser.org/ide/, it can run parser online, and show output tree.

Chapter 3

Implementation

3.1 Reading packages

3.1.1 Bluetcl

As mentioned before, bluetcl is a tool written in Tcl that behaves like a library. To interact with this tool I have written a script using pexpect library. This script works by creating subprocess of bluetcl, and gives works like a library with functions that allow for performing certain quarries. Core of this script is a function called `fancy_call` that takes as input a string that is a command and returns output of stripped of warnings or raises an error if such occurred(for example in case where package was not found). To remove warnings I make some assumptions.

- I only care about supporting fixed set of commands.
- For those commands output that I care about is always equal to the last line. (this was checked empirically)
- Output of a command is always followed by % a character that never occurs in the rest of the output and marks the end of the output.

This simple script allows me for:

- Initialize subprocess
- Add folder to search path of bluetcl
- Load package (bluetcl takes care of loading dependencies)
- Get list of loaded packages
- List functions in package
- List types in package
- Get information about types and function in the package

3.2 Parsing

TODO, I might want to clean this up a bit before I write about it.

3.3 Synthesizing

The goal of this project is to synthesize a