Rozszerzenie symulatora logiki dla języka Verilog poprzez dodanie bezpośredniej obsługi FIRRTL

(Extending logic simulator for verilog through adding direct support for FIRRTL)

Jakub Szczerbiński

Praca licencjacka

Promotor: dr Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

11 lutego 2021

Streszczenie

Celem tej pracy jest stworzenie narzędzia do konwersji kodu w językach Chisel oraz FIRRTL do formatu używanego przez edukacyjny symulator układów logicznych DigitalJS. Stworzenie takiego narzędzia pozwoli rozszerzyć możliwości symulatora DigitalJS oraz umożliwi łatwiejsze nauczanie języka Chisel, dzięki możliwości skorzystania przez jego użytkowników z tego symulatora. Praca skupia się na stworzeniu poprawnego algorytmu tłumaczenia razem z zestawem testów, który w przyszłości będzie mógł być łatwo rozwijany w celu bardziej optymalnego zrealizowania celów dydaktycznych.

The purpose of this work is to create a tool that allows conversion from Chisel and FIRRTL languages to the format used by educational circuit simulator DigitalJS. Creation of such tool extends capabilities of the DigitalJS simulator and makes it easier for Chisel users to learn the language through the usage of that simulator. This work focuses on creating a correct translation algorithm along with a testsuite, which could be easily developed further in order to better suit educational needs.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Języki opisu sprzętu

Języki opisu sprzętu to rodzina języków komputerowych służących do opisu działania obwodów elektronicznych, najczęściej układów logiki cyfrowej. Używane są na różnych etapach procesu tworzenia elektroniki: projektowania, weryfikacji oraz syntezy.

Historia języków opisu sprzętu sięga roku 1938. W pracy magisterskiej Claude'a Shannona z tamtego roku można znaleźć prawdopodobnie najwcześniejsze odniesienie do języków opisów sprzętów. Zaproponował on w niej metodę opisu obwodów za pomocą algebry Boole'a [1]. W latach pięćdziesiątych Irving S. Reed pokazał, że wyrażenia algebry Boole'a mogą być zrealizowane za pomocą obwodów logicznych [2]. Języki opisu sprzętu były dalej rozwijane w latach sześćdziesiątych do tego stopnia, że w 1973 roku odbyła się pierwsza konferencja poświęcona tej dziedzinie, a w ankiecie z 1974 roku wyróżniono ponad 50 języków opisów sprzętów z całego świata. Języki tworzone w latach siedemdziesiątych różniły się od tych wykorzystywanych współcześnie. Większość z nich posiadała tylko proste niskopoziomowe wyrażenia i służyły jedynie do opisania i weryfikacji projektu układu, ale nie do jego syntezy [3].

Techniki tworzenia układów (w tym VLSI[1]), które pojawiły się w latach osiemdziesiątych, sprawiły, że pojawiło się zapotrzebowanie na potężniejsze narzędzia wspomagające tworzenie układów cyfrowych. To zapotrzebowanie sprawiło, że powstało wiele nowych symulatorów, narzędzi do weryfikacji, języków specyfikacji, oraz pierwsze narzędzia do syntezy. Jednym z języków specyfikacji był VHDL[2] stworzony na potrzeby Departamentu Obrony Stanów Zjednoczonych w celu specyfikacji specjalizowanych układów scalonych. Jednym z symulatorów był, stworzony przez firmę

[1] Very Large Scale Integration – proces tworzenia układu scalonego poprzez połączenie milionów tranzystorów MOS w jeden układ scalony

[2] VHSIC (Very high speed integrated circuits) Hardware Description Language

Gateway Design Automation, Verilog-XL, który zdefiniował na swoje potrzeby język opisu sprzętu Verilog. VHDL i Verilog przez lata pokonały lub wchłonęły konkurencje i dominują dziś rynek języków opisów sprzętu. [3]

1.2. Verilog i SystemVerilog

Verilog został zaprojektowany w latach 1983-84 przez Phila Moorbiego i Chi-lai Huanga. Phil Moorby w latach siedemdziesiątych pracował nad językiem i symulatorem HILO [3]. Chi-lai Huang natomiast opublikował swoją pracę magisterską na temat syntezy języka LALSD II [4]. Verilog w swojej wczesnej formie był językiem mocno wzorowanym na HILO, ale w przeciwieństwie do HILO był projektowany również z myślą o syntezie. Verilog-XL szybko zdobył silną pozycję na rynku symulatorów i w 1987 roku firma Synsopsys nawiązała współpracę z firmą Gateway, dzięki której powstał syntezator języka Verilog - Design Compiler. Do końca lat 80. język Verilog był własnościowy, jednak w 1990 roku, aby konkurować z otwartym standardem VHDLa, został przekazany do domeny publicznej, w ręce nowo utworzonej organizacji OVI (Open Verilog International). Prace w ramach OVI i jej spadkobierców doprowadziły do postania standardów IEEE Verilog: IEEE 1364-1995, IEEE 1364-2001, IEEE 1364-2005. W 2005 roku powstał także standard nowego języka SystemVerilog, który był nadzbiorem Veriloga. W 2009 standard 1800-2009 połączył SystemVeriloga i Veriloga w jeden język. [3]

O sukcesie Veriloga zadecydowały najprawdopodobniej dwie zalety: semantyka dająca dobry kompromis pomiędzy dokładnością a szybkością symulacji oraz współistniejące konstrukcje pozwalające opisać układ na wielu poziomach abstrakcji. Niestety przez lata stał się bardzo rozległym i skomplikowanym językiem. Przez to późniejsze standardy rzadko są wspierane w całości przez jakiekolwiek narzędzia, a początkujący użytkownicy mają trudności z jego opanowaniem i zrozumieniem.

1.3. DigitalJS

DigitalJS jest symulatorem układów cyfrowych napisanym w JavaScripcie stworzonym przez Marka Materzoka [5]. Projekt DigitalJS jest zmotywowany zastosowaniami dydaktycznymi, dlatego ważnymi aspektami tego narzędzia są przedstawienie graficzne układu, możliwości inspekcji wartości sygnałów pomiędzy komponentami układu oraz prostota użycia.

Korzystanie z takiego symulatora podczas nauki logiki cyfrowej czy języka Verilog jest bardzo pomocne, ponieważ reprezentacja graficzna dostarczona przez symulator pozwala budować mentalny model tego jak mogą wyglądać układy logiczne, zakodowane przez poszczególne konstrukcje w Verilogu. Jest to szczególnie przydatne

dla osób, które wcześniej poznały języki programowania. Poprzez podobieństwo języków opisów sprzętów do języków programowania łatwo popełnić błąd myślenia o działaniu układów logicznych w podobny sposób w jaki myśli się o działaniu programów komputerowych. Dzięki interaktywnej reprezentacji graficznej łatwo uniknąć takiej pułapki.

DigitalJS ma wiele zalet i dobrze spełnia swoje zadanie. Niestety narzędzie to wspiera jedynie translacje z Veriloga do swojego wewnętrznego formatu korzystając z kompilatora yosys [6]. Verilog jest dominującym językiem opisu sprzętu i ma wiele przydatnych funkcjonalności, ale jest również został zaprojektowany w czasach, kiedy nie rozumiano tak dobrze ergonomii języków komputerowych, z konstrukcjami, które są nieintuicyjne w użyciu lub łatwo prowadzą do błędów, co czyni go trudnym do nauki. Sam yosys jest kompilatorem, którego wewnętrzna reprezentacja operuje na niskim poziomie abstrakcji, co może uniemożliwiać rozszerzenie go o transformacje przydatne z perpektywy DigitalJS (np. takie które zwiększają czytelność reprezentacji graficznej układu).

1.4. Chisel i FIRRTL

Od lat na rynku urządzeń komputerowych dominują procesory ogólnego przeznaczenia, jednak z powodu ograniczeń fizycznych wielkości tranzystorów wiemy, że będzie trudno zwiększać ich wydajność poprzez ulepszenie procesu technologicznego. Z tego powodu rozważa się alternatywne sposoby zwiększenia wydajności procesorów. Jedną z nich jest ich specjalizacja. [7]

Układy typu ASIC (application specific integrated circuit), które osiągają swoją wydajność w ten sposób, są niszą rynkową z powodu wysokich kosztów projektowania układów. Jednym ze sposobów na ich obniżenie jest powtórne używanie części wcześniejszych projektów układów, co okazuje się trudne do osiągnięcia za pomocą języków opisów sprzętów dominujących dziś rynek. Z tego powodu powstały nowe języki opisu sprzętu próbujące rozwiązać ten problem. Jednym z nich jest Chisel[8]/FIRRTL[9].

Chisel jest biblioteką w Scali, która definiuje DSL [3] pozwalający tworzyć generatory układów, z kolei FIRRTL jest językiem opisu sprzętu w którym są zdefiniowane układy zwracane przez generatory napisane w Chiselu. Zaprojektowany w ten sposób ekosystem, składający się z języka konstrukcji sprzętu współgrającego z nim języka opisu sprzętu ma wiele zalet. Sam język opisu sprzętu może być prosty, ponieważ nie musi być rozszerzany o elementy dostępne w językach programowania (tak jak w Verilogu czy VHDLu), ponieważ przy pisaniu generatora w Chiselu użytkownik ma do dyspozycji wszystkie konstrukcje dostępne w Scali. Dzieki temu, że Chisel służy do pisania generatorów układów, łatwo możemy sparametryzować projekt w taki

[3] Domain Specific Language – język programowania przystosowany do rozwiązywania określonej dziedziny problemów

sposób, że stworzony generator będzie przydatny przy tworzeniu innych układów. Przez to, że dużo funkcjonalności jest realizowanych przez Chisela i Scalę, FIRRTL jest małym językiem i może służyć jako pośrednia reprezentacja dla kompilatora. Kompilator FIRRTLa o takiej samej nazwie jak język korzysta z niego jak z postaci pośredniej. Udostępnia on możliwości rozszerzenia i konfiguracji procesu kompilacji, które pozwalają na niestandardową optymalizację układu dostosowaną do technologii jego realizacji. Oczywiście integracja FIRRTLa z istniejącymi narzędziami byłaby trudna, dlatego kompilator wspiera emisję układu do języka Verilog.

1.5. Motywacja

Obecnie istnieje sposób na konwersje układów w Chiselu/FIRRTLu do formatu DigitalJS. Można najpierw wygenerować układ w FIRRTLu, skonwertować go do Veriloga, a następnie użyć narzędzia `yosys2digitaljs`[4] by uzyskać układ w formacie DigitalJS. To rozwiązanie działa poprawnie i dzieki niemu użytkownicy Chisela mogą korzystać z symulatora DigitalJS. Niestety ten sposób tłumaczenia układu zatraca jego początkową strukturę. Powoduje to, że korzyści z korzystania symulatora DigitalJS są ograniczone, ponieważ połączenie pierwotnego opisu układu w Chiselu/FIRRTLu z wynikowym diagramem w symulatorze będzie zbyt trudne. Rozwiązaniem tego problemu jest napisanie translatora, który przetłumaczy układ w taki sposób, aby charakterystyka początkowego układu została zachowana.

Samo narzędzie DigitalJS także może skorzystać z takiej translacji. Obecnie wachlarz jego możliwości jest ograniczony przez opis układu uzyskany z yosysa. FIRRTL obecnie posiada wiele konstruktów, których nie da się wyrazić wprost w DigitalJS. Dzięki napisaniu niestandardowego narzędzia, które tłumaczy FIRRTLa do formatu DigitalJS, można zdecydować o które z tych konstruktów warto rozszerzyć DigitalJS, a które z nich należy wyrazić za pomocą istniejących konstruktów. Otwiera to nowe możliwości rozszerzenia symulatora DigitalJS.

Celem tej pracy jest zaimplementowanie poprawnego, w sensie równoważności symulacji, translatora FIRRTLa do formatu DigitalJS oraz stworzenie zestawu testów zapewniających jego poprawność. Stworzenie takiego symulatora i zestawu testów tworzy podstawę, dzięki której będzie można dążyć do bliższej integracji FIRRTLa z symulatorem DigitalJS oraz rozwinięcia i doskonalenia narzędzia DigitalJS, a także samego translatora. Celem tej pracy nie jest rozszerzenie narzędzia DigitalJS ani też eksperymentowanie ze sposobem translacji, który poprawia czytelność układu w symulatorze DigitalJS.

[4]Narzędzie korzystające z kompilatora yosys pozwalające na konwersję pomiędzy Verilogiem a formatem DigitalJS

Rozdział 2.

Chisel, FIRRTL i DigitalJS

W tym rozdziale, krótko opiszę języki opisu sprzętu których dotyczy moja praca.

2.1. Chisel oraz FIRRTL

Chisel jest biblioteką w Scali służącą do tworzenia generatorów układów w FIRRTLu. Zwykle takie układy składają się z wielu modułów, które są podstawową jednostką abstakcji w Chiselu. Moduły są tworzone poprzez rozszerzenie klasy `chisel3.Module`. Korzystając z elastyczności Scali, Chisel definiuje DSL wykorzystywany do wyrażenia logiki modułów w ciele takiej klasy.

```
import chisel3._

class MuxExample extends Module {
    val io = IO(new Bundle {
        val sel =   Input(UInt(2.W))
        val in  =   Input(Vec(4, UInt(1.W)))
        val out =   Input(UInt(1.W))
    })

    io.out := io.in(io.sel)
}
```

Powyższa klasa definiuje generator, który emituje opis układu w FIRRTLu. Wyemitowany opis przed jakimikolwiek przekształceniami odzwierciedla mocno generator, ponieważ dla każdej konstrukcji wyrażonej w DSLu Chisela moduł generuje ściśle odpowiadająca jej konstrukcje w FIRRTLu.

```
circuit MuxExample :
  module MuxExample :
    input clock : Clock
    input reset : UInt<1>
    output io : {flip sel : UInt<2>, flip in : UInt<1>[4], out : UInt<1>}

    io.out <= io.in[io.sel] @[MuxExample.scala 16:12]
```

2.1.1. Typy w FIRRTLu

Ważną cechą FIRRTLa jest jego system typów. Pozwala on zapobiegać tworzeniu nieprawidłowych układów poprzez inferencje typów i wykrywanie nieprawidłowych wyrażeń. Zawiera on:

- Typy proste
  - `UInt<n>` – typ liczby całkowita bez znaku o długości `n` bitów
  - `SInt<n>` – typ liczby całkowita ze znakiem o długości `n` bitów
  - `Fixed<n><<m>>` – typ liczby stałoprzecinkowej o długości `n` bitów z `m` miejscami po przecinku
  - `Clock` – typ dla zegara
  - `Analog<n>` – typ analogowy
- Typy złożone
  - `T[n]`– typ wektora elementów typu `T` o długości `n`
  - `{f0:T0,f1:T1,...,fn:Tn}` – typ rekordu o polach `f1` `f2`, ..., `fn` o typach odpowiednio `T0`, `T1`, ..., `Tn`

Aby zweryfikować prawidłowość wyrażeń w FIRRTLu stosuje się także wykrywanie i sprawdzanie poprawności kierunku przepływu sygnałów. Każdemu wyrażeniu przypisuje się kierunek przepływu, a następnie sprawdza się czy cały układ ma spójny przepływ.

2.1.2. Formy FIRRTLa

Układy zaraz po wygenerowaniu są opisane w za pomocą pełnego języka FIRRTL. Na potrzeby kompilacji układów w specyfikacji zostały wyróżnione dwa podzbiory FIRRTLa:

- MidFIRRTL, która zacieśnia FIRRTLa następującymi warunkami:
  - Wszystkie szerokości (połączeń, portów) muszą być wyraźnie określone.
  - Wyrażenie warunkowe nie może być używane.
  - Dynamiczny dostęp do pól jest zabroniony.
  - Każdy komponent musi być podłączony dokładnie jeden raz.
- LoFIRRTL, rozszerza obostrzenia MidFIRRTLa o następujące warunki:
  - Wszystkie komponenty muszą być zadeklarowane z użyciem jedynie typów prostych.

– Instrukcja częściowego podłączenia nie może być używana.

```
circuit MuxExample :
  module MuxExample :
    input clock : Clock
    input reset : UInt<1>
    input io_sel : UInt<2>
    input io_in_0 : UInt<1>
    input io_in_1 : UInt<1>
    input io_in_2 : UInt<1>
    input io_in_3 : UInt<1>
    output io_out : UInt<1>

    node _GEN_0 = validif(eq(UInt<1>("h0"), io_sel), io_in_0)
    node _GEN_1 = mux(eq(UInt<1>("h1"), io_sel), io_in_1, _GEN_0)
    node _GEN_2 = mux(eq(UInt<2>("h2"), io_sel), io_in_2, _GEN_1)
    node _GEN_3 = mux(eq(UInt<2>("h3"), io_sel), io_in_3, _GEN_2)
    node _io_in_io_sel = _GEN_3
```

Najczęstszym przypadkiem użycia Chisela i kompilatora FIRRTLa jest wygenerowanie FIRRTLa przez generator napisany w Chiselu, następnie nałożenie na niego szeregu transformacji, które najpierw go sprowadzają do formy MidFIRRTL, a potem LoFIRRTLa. Na końcu taki układ w LoFIRRTLu zostaje przetłumaczony na układ w innym języku sprzętu, najczęściej Verilogu.

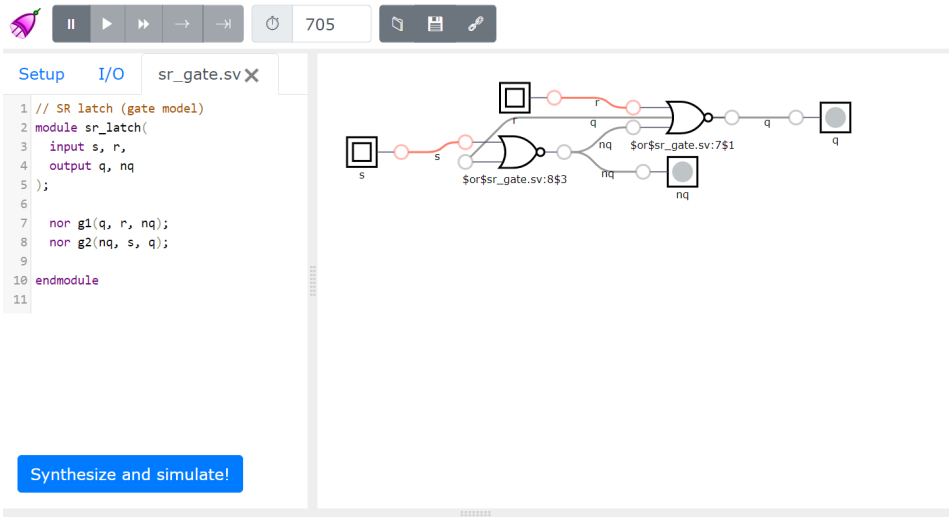
Infrastruktura kompilatora FIRRTLa jest elastyczna i pozwala na więcej niż taki przypadek uzycia. Formy, takie jak MidFIRRTL czy LoFIRRTL są zdefiniowane za pomocą zestawów transformacji, który układ musi przejść, aby na pewno mieć daną formę. Dzięki temu można zdefiniować własne formy FIRRTLa, które są lepiej przystosowane to przypadku uzycia. Tak robi to część kompilatora tłumacząca LoFIRRTLa na Veriloga. Zamiast tłumaczyć wprost, definiuje ona zoptymalizowaną pod tłumaczenie na Veriloga formę, i tłumaczy uzywając lepiej przystosowanej do tego postaci FIRRTLa.

Translator FIRRTLa do formatu DigitalJS będzie używa formy LoFIRRTL i dodatkowych transformacji do tłumaczenia, ponieważ taki sposób translacji jest najprostszy i pomaga w uniknięciu błędów. Jednak kompilator FIRRTLa pozwala na uważniejsze dobrane postaci FIRRTLa, którą chcemy tłumaczyć. Potencjalnie daje to możliwość dostrojenia translatora w taki sposób, by użycie go w parze symulatorem DigitalJS dawało lepsze efekty dydaktyczne.

2.2. DigitalJS

Symulator DigitalJS jest przeglądarkową biblioteką napisaną w Javascripcie, która wykorzystuję bibliotekę JointJS do wizualizacji symulowanego układu. Symulator można osadzić w aplikacji webowej i sterować nim programistycznie. Symulator

jest wykorzystywany w przeglądarkowym IDE DigitalJS Online w którym można pisać i uruchamiać układy stworzone w Verilogu oraz testbenche[1] zakodowane w Lua działające na symulatorze DigitalJS.



Aby obsługiwać Veriloga używane jest narzędzie yosys2digitaljs, które konwertuje Veriloga do formatu DigitalJS za pomocą kompilatora yosys, z użyciem backendu, który emituje netlistę układu opisanego w JSONie. Ten opis układu jest lekko modyfikowany zanim staje się układem w formacie DigitalJS.

Format DigitalJS jest bazowany na formacie JSON. Układ jest opisany przez obiekt zawierający następujące pola:

- devices – obiekt zawierający urządzenia będących częścią głównego układu
- connectors – tablica zawierająca obiekty opisujące pojedyncze połączenia między urządzeniami w głównym układzie
- subcircuits – tablica podukładów. Każdy z nich jest opisany za pomocą obiektu polami devices i connectors zawierającymi analogiczny opis urządzeń i połączeń podukładów.

Każde urządzenie ma 0 lub więcej portów wyjściowych i wejściowych. Ich ilość i nazwy są zdefiniowane na podstawie ich typów i parametrów urządzeń. Na przykład urządzenie typu `And`, posiada dwa porty wejściowe `in0`, `in1` oraz port wyjściowy `out`. Porty identyfikowane są w układzie za pomocą identyfikatora urządzenia i nazwy portu. Połączenia są kodowane za pomocą obiektu zawierającego dwa pola `in` i `out`. Każde z nich ma przypisany obiekt zawierający identyfikator układu oraz nazwe portu.

Ponieważ DigitalJS jest definiowany jako JSON, a symulator DigitalJS nie waliduje układów, niezgodności z formatem często skutkują w niełatwych do wykrycia

[1] Programy, które służą do uruchamiania i testowania układów

błędów działania układu w symulatorze, często objawiającymi się błędami aplikacji. Jest to problematyczne podczas budowania narzędzi które tworzą układ przeznaczony do użycia z symulatorem. Każdy błąd w narzędziu, który wprowadza nieprawidłowości w wynikowym układzie, może łatwo zostać niezauważony.

Rozwiązaniem tych problemów jest walidator układów. Pozwala on wcześniej wyłapywać błędy w narzędziach, ułatwia testowanie i jeśli jest odpowiednio napisany, dokumentuje format DigitalJS. Dlatego w przygotowaniu do tłumaczenia FIRRTLa na DigitalJS stworzyłem validator – `digitaljs_schema`.Walidator składa się z:

- Opisu formatu DigitalJS za pomocą typów w TypeScript używanego z biblioteką która sprawdza zgodność JSONa z typem TypeScriptowym. Pozwala on na zweryfikowanie czy istnieją wszystkie pola w obiektach, które są wymagane przez format.
- Sprawdzaczki połączeń pomiędzy urządzeniami. Dzięki niej można uniknąć sytuacji, w których mamy połączenia pomiędzy nieistniejącymi portami lub urządzeniami, a także wielokrotnych połączeń do jednego wejścia.

Rozdział 3.

FIRRTL2DigitalJS

Kompilator FIRRTLa oraz Chisel są napisane w Scali. FIRRTL2DigitalJS także został napisany w Scali, ponieważ dzięki zastosowaniu tej samej technologii z której korzysta Chisel/FIRRTL można korzystać z kodu bibliotecznego tych projektów. Kompilator FIRRTLa jest zaprojektowany z myślą o rozszerzalności, dzięki czemu z łatwością można skorzystać z jego parsera i transformacji.

Kompilator FIRRTLa udostępnia nam narzędzia, pozwalające na dostosowanie wejściowej postaci FIRRTLa do translatora. Warto zatem zidentyfikować dokładnie jaka postać jest najwygodniejsza do użycia. Najkorzystniejsze z perspektywy translatora jest gdy struktura postaci wejściowej jest jak najbardziej zbliżona do postaci wyjściowej. Układy w DigitalJS mają płaską strukturę, składają się z listy urządzeń oraz listy połączeń, więc chcemy zadbać, aby FIRRTL na wejściu miał odpowiednio płaską strukturę.

Transformacja do LoFIRRTLa zapewnia nam, pożądaną przez translator, płaską strukturę. Niestety, kiedy przyjrzymy się bliżej układom w LoFIRRTLu znajdziemy dwie znaczące różnice pomiędzy tą reprezentacją a DigitalJSową:

- W LoFIRRTLu można czasem używać portów jako źródeł, ale nie zawsze można w ich taki sam sposób używać w DigitalJS.
- Interfejs pamięci jest różny, oraz konwencja nazw portów pamięci jest inna.

Te różnice są dość niewygodne do obsługi przy tłumaczeniu, dlatego korzystam z infrastruktury kompilatora i napisałem dwa dodatkowe przekształcenia, które zmienią układ w taki sposób by pozbyć się problematycznych konstrukcji.

3.1. Zastępowanie portów używanych jako źródła

```
circuit clone :
  module clone :
    input io_in : UInt<1>
    output io_out0 : UInt<1>
    output io_out1 : UInt<1>
    io_out0 <= io_in;
    io_out1 <= io_out0;
```

Listing 1: Port wyjściowy modułu używany jako źródło

```
circuit main :
  module main :
    input io_in : UInt<1>
    output io_out : UInt<1>
    inst s of sub;
    s.io_in <= io_in;
    io_out <= s.io_in;
  module sub :
    input io_in : UInt<1>
```

Listing 2: Port wejściowy instancji modułu używany jako źródło

Istnieją dwa przypadki użycia portów jako źródeł w sposób niekompatybilny z formatem DigitalJS.

- Moduł może używać swojego portu wyjściowego jako źródła. Najprostszy przykład to 1. Do port `io_out0` zostaje połączony z portem `io_in`, natomiast `io_out1` z `io_out0`. W efekcie tych przypisań powstaje układ, który posiada jedno wejście i dwa wyjścia, które mają wartość wejścia. W formacie DigitalJS niedozwolone połączenie jest analogiczne do drugiego przypisania. Porty wyjściowe z FIRRTLa najłatwiej są reprezentowane jako urządzenia typu `Output`. Urządzenia typu `Output` posiadają tylko jeden port wejściowy i żadnych wyjściowych, więc nie nadają się do użycia jako źródła sygnałów.
- Porty wejściowe instancji modułów również mogą być używane jako źródła. Taki przypadek obrazuje układ z 2. Port `io_in` z instancji modułu sub zostaje połączony z portem `io_in`, a do portu `io_out` zostaje przypisana port `io_in` należący do instancji modułu. W DigitalJS FIRRTLowe instancje modułów mogą być wprost zareprezentowane jako urządzenia typu `Subcircuit`, a porty instancji jako porty tych urządzeń. W przeciwieństwie do FIRRTLa, DigitalJS nie pozwala jednak na korzystanie z portów wejściowych podukładów jako ze źródeł.

```
circuit mult :
  module mult :
    input io_in : UInt<1>
    output io_out0 : UInt<1>
```

```
    output io_out1 : UInt<1>
    output io_out2 : UInt<1>
    output io_out3 : UInt<1>

    io_out0 <= io_in;
    io_out1 <= io_out0;
    io_out2 <= io_out1;
    io_out3 <= io_out2;
```

Listing 3: Ciąg przypisań zawierający nieporządane użycie portów

W tym celu napisałem transformacje, która wymienia w wszystkie takie wystąpienia portów w wyrażeniach na wyrażenia do nich przypisane. Transformacja na module najpierw szuka wszystkich zadeklarowane portów wyjściowe oraz instancje modułów. Następnie dla każdego z portów wyjściowych i portów wejściowych instancji modułów wylicza wyrażenie które może zostać podstawione za te porty.

Dla wielu z nich wyliczonym wyrażeniem jest prawa strona przypisania. Tak jest w przykładach 2 oraz 1, gdzie za porty `s.io_in` i `s.io_in` wystarczy podstawić `io_in`. Jednak czasem może się zdążyć, że istnieje ciąg przypisań, który zawiera niepożądane użycie portów, tak jak to jest pokazane w przykładzie 3. Dlatego algorytm musi wyliczać wyrażenia inaczej. Zaczyna od on wyrażenia po prawej stronie przypisania, a następnie rekurencyjnie podstawia niepożądane porty za wyrażenia po prawej stronie ich przypisań.

```
circuit exp :
  module exp :
    input io_in : UInt<1>
    output io_out0 : UInt<1>
    output io_out1 : UInt<1>
    output io_out2 : UInt<1>

    io_out0 <= and(io_in, io_in);
    io_out1 <= and(io_out0, io_out0);
    io_out2 <= and(io_out1, io_out1);
```

Listing 4: Ciąg przypisań

```
circuit exp :
  module exp :
    input io_in : UInt<1>
    output io_out0 : UInt<1>
    output io_out1 : UInt<1>
    output io_out2 : UInt<1>

    io_out0 <= and(io_in, io_in);
    io_out1 <= and(and(io_in, io_in),  and(io_in, io_in));
    io_out2 <= and(and(and(io_in, io_in),  and(io_in, io_in)),
                   and(and(io_in, io_in),  and(io_in, io_in)));
```

Listing 5: Ciąg przypisań po transformacji

Przy zastosowaniu tej metody problemem może być to, że wielkość wyrażeń po prawej stronie może urosnąć wykładniczo w stosunku do długości ciągu przypisań oraz stwarza okazje do duplikacji wyrażeń, tak jak to widać na przykładach 4 i 5. Kompilator FIRRTLa pozwala określić na jakiej formie FIRRTLa powinna być uruchamiana transformacja. Jeśli będziemy wymagać, aby transformacja została zaaplikowana na formie LoFIRRTL to nie musimy się martwić o te problemy, ponieważ w tej postaci po prawej stronie przypisań występują tylko bardzo proste wyrażenia, które albo nie muszą zostać zastąpione albo zostają zastąpione w całości.

3.2. Zastąpienie pamięci modułami zewnętrznymi

Oba języki opisu sprzętu oferują abstrakcje dla pamięci. Niestety interfejsy tych abstrakcji i konwencje nazewnictwa portów nie są ze sobą zgodne.

```
case class DefMemory(
    info: Info,
    name: String,
    dataType: Type,
    depth: BigInt,
    writeLatency: Int,
    readLatency: Int,
    readers: Seq[String],
    writers: Seq[String],
    readwriters: Seq[String],
    readUnderWrite: ReadUnderWrite.Value = ReadUnderWrite.Undefined)
```

Listing 6: Typ dla pamięci w drzewie składniowym FIRRTLa

Lista parametrów dla deklaracji pamięci FIRRTLa pokazana jest w 6. Urządzenie typu `Memory` w DigitalJS ma następujące własności:

- Atrybuty:
  - `bits` – Ilość bitów dla słowa
  - `abits` – Ilość bitów dla adresu
  - `words` – Ilość słów
  - `offset` – Offset adresu, w którym znajduje się pierwsze słowo
  - `rdports` – Opisy portów odczytu
  - `wrports` – Opisy portów zapisu
  - `memdata` – Opis początkowego stanu pamięci
- Wejścia i wyjścia dla każdego portu odczytującego (o numerze K):
  - `rdKaddr` – wejście o `abits` bitów długości
  - `rdKdata` – wyjście o `bits` bitów długości

  - `rdKen` – wejście 1-bitowe, obecne jeśli `enable_polarity` jest w opisie portu
  - `rdKclk` – wejście 1-bitowe, obecne jeśli `clock_polarity` jest w opisie portu

- Wejścia i wyjścia dla każdego portu piszącego (o numerze K):
  - `wrKaddr` – wejście o `abits` bitów długości
  - `wrKdata`– wejście o `bits` bitów długości
  - `wrKen` – wejście 1-bitowe, obecne jeśli `enable_polarity` jest w opisie portu
  - `wrKclk` – wejście 1-bitowe, obecne jeśli `clock_polarity` jest w opisie portu

Pamięć w FIRRTLu ma pola, które nie mają odzwierciedlenia w DigitalJS, a nazwy portów są nadawane przez użytkownika, a nie są tworzone na podstawie indeksu portu. Aby zniwelować te rozbieżności, zdecydowałem podmienić instancje pamięci na instancję modułu, który będzie miał interfejs zgodny z interfejsem instancji pamięci, a jego implementacja będzie symulowała zachowanie pamięci wykorzystując dowolne wyrażenia FIRRTLa oprócz instancji pamięci oraz zewnętrzny moduł[1] reprezentujący urządzenie pamięci z DigitalJSa. Ten zewnętrzny moduł zostanie przetłumaczony wprost na urządzenie `Memory` w fazie translacji. Rozważmy jak można symulować zachowanie aspektów pamięci w FIRRTLu, których nie można obsłużyć za pomocą urządzenia `Memory`:

- `write_latency`, `read_latency`,które określają opóźnienie w liczbie cykli odczytu i zapisu do pamięci. W celu symulacji zachowania możemy opóźnić sygnały poprowadzone do portów za pomocą kolejek o różnej długości, a także manipulować atrybutem `transparency`[2] w opisie portu odczytującego.
- `readwriters` określa ile jest portów, które służą do zapisu i odczytu. Można stworzyć po jednym porcie do czytania i pisania w zewnętrznym module i użyć ich do symulacji zachowania portu do odczytu i zapisu.
- `readUnderWrite` zachowanie odczytu podczas zapisu pod ten sam adres. Ma trzy możliwe wartości: `undefined` – niezdefiniowane zachowanie, `new` – zwracana jest wartość, która jest w momencie zakończenia odczytu (równoważne z opóźnieniem sygnałów związanych z rozpoczęciem odczytu `addr`, `en`), `old` – zwracana jest wartość, która jest w pamięci w momencie rozpoczęcia odczytu (równoważne z opóźnieniem sygnału na wyjściu `data`). W zależności

[1] FIRRTL pozwala definiować zewnętrzne moduły, które są interfejsami bez implementacji. Zwykle używane są do zrealizowania interoperacyjności między FIRRTLem a innym językiem opisu sprzętu.

[2] Atrybut transparency decyduje czy port pamięci jest synchroniczny czy asynchroniczny. Asynchroniczne porty w DigitalJS odpowiadają portom z opóźnieniem 0 w FIRRTLu

od tej wartości zmieniany jest sposób realizowania opóźnienia dla parametru `read_latency`.

Obecnie FIRRTL2DigitalJS obsługuje wartość 1 dla `write_latency` oraz wartości 0 i 1 dla `read_latency`, ponieważ pamięci tylko z takimi wartościami są emitowane przez Chisela, a także wyłącznie te wartości są wspierane przez symulator FIRRTLa. Pozostałe wartości są również obsługiwane, jednak ich działanie nie zostało do końca zweryfikowane za pomocą technik testowania opisanych w następnym rozdziale, a jedynie manualnie z powodu braku wsparcia ze strony symulatora FIRRTLa.

Obsługa portów z jednoczesną możliwością odczytu i zapisu została zaimplementowana w wyżej wymieniony sposób i przetestowana dla wyżej wymienionych wartości `write_latency` i `read_latency`.

Dzięki zastosowaniu takiego przekształcenia zyskujemy:

- Zachowanie nazw portów w wynikowym układzie w DigitalJS (poprzez użycie ich w nazwach portów modułu pamięci).
- Ukrycie implementacji obsługi portów do odczytu i zapisu wewnątrz modułu.
- Ukrycie implementacji obsługi `write_latency`, `read_latency`, `readUnderWrite` wewnątrz modułu.

3.3. Translacja

Układ po przejściu na postać LoFIRRTL rozszerzoną o wyżej opisane transformacje jest gotowy do translacji. Jest ona dość prosta, ponieważ układ w takiej postaci jest już całkiem podobny do odpowiadającego układu w formacie DigitalJS. Rozważmy wyrażenia w FIRRTLu i jak zostają przetłumaczone na urządzenia i połączenia.

1. Porty będą przetłumaczone do urządzenia `Input/Output` w zależności czy jest wejściowy czy wyjściowy.
2. Instrukcje:
    - Połączenie – tworzy połączenie między dwoma portami urządzeń: wejściowym wyliczonym z wyrażenia po lewej strony operatora `<=`, wyjściowym wyliczonym z wyrażenie po prawej stronie.
    - Definicja węzła – tworzy urządzenie z nazwą taką samą jak nazwa węzła na podstawie przypisanego do niego wyrażenia.
    - Definicja rejestru – tworzy urządzenie `Dff`

- Definicja instancji modułu – tworzy urządzenie `Subcircuit` z opowiadającym podukładem.

3. Wyrażenia – tłumaczymy na port oraz opcjonalnie na urządzenia i połączenia.

   - Referencje i dostęp do pól – każda instrukcja oprócz połączenia opisu część układu, do której może odnosić się wyrażenie. Referencje do nich i dostęp do ich pól (w przypadku instancji modułu) tłumacze na porty odpowiednich urządzeń. Ponieważ nazwy referencji odpowiadają nazwom urządzeń, a nazwy pól odpowiadają nazwom portów, translacja jest prosta.
   - Operatory – dla każdego operatora jest dobierane odpowiednie urządzenie z DigitalJS, jego argumenty zostają przetłumaczone, a wynikające z tłumaczenia argumentów porty wyjściowe zostają połączone do portów wejściowych. Wszystkie operatory tłumaczą się na urządzenia, które mają jeden port wyjściowy `out`, na którym podawana jest wyliczona wartość operatora.
   - Stałe - tłumaczone są na urządzenie `Constant`, a ich wartość jest dostępna za pomocą portu `out`.

Każdy moduł zostaje przetłumaczony na podukład w wyżej wymieniony sposób. Następnie zostaje wybrany główny układ i z niego oraz reszty podukładów zostaje utworzony układ w DigitalJS.

Rozdział 4.

Testowanie rozwiązania

W wyniku translacji otrzymujemy układ, który powinien w jakimś sensie być poprawnie przetłumaczonym układem na wejściu. Aby zrozumieć na czym polega ta poprawność, trzeba spojrzeć w jaki sposób zdefiniowana jest semantyka obu języków sprzętu. Dla FIRRTLa istnieje specyfikacja, która mówi o zachowaniu układów, jednak dla DigitalJSa znaczenie układów jest definiowane przez implementacje symulatora. Istnieje zgodny ze specyfikacją symulator FIRRTLa. Skoro układy w obu językach posiadają symulatory, które działają zgodnie z semantyką języka, definicja poprawności może być bazowana na zachowaniu układów podczas symulacji.

4.1. Peek, poke, step

Do testowania układów Chiselu/FIRRTLu często używany jest interpreter FIRRTLa, schowany za interfejsem, który pozwala wykonać trzy podstawowe operacje na układzie:

- peek – sprawdza wartość jednego z portów wyjściowych układu.
- poke – ustawia wartość dla jednego z portów wejściowych układu i propaguje wartość do momentu ustabilizowania układu.
- step – wykonuje krok zegara i propaguje wartości do momentu ustabilizowania układu.

Klasa w Scali implementująca ten interfejs nazywa się `InterpretiveTester`. Testy układów są programami w Scali, wykorzystują tą klasę. Możemy wykorzystać ten interfejs do weryfikacji układów DigitalJS, jeśli zrealizujemy te trzy operacje za pomocą symulatora DigitalJS. Implementacja takiego interfejsu w Scali (o nazwie `DigitalJSTester`)może być używana wymiennie z `InterpretiveTester`, a tym samym korzystać ze stworzonych dla niego testów układów.

Mając takie mechanizmy uruchamiania testów układów możemy mówić, że układ jest poprawnie przetłumaczony wtedy gdy dla dowolnego testu jeśli układ w FIRRTLu przechodzi test to układ w DigitalJS także przechodzi test. Wydaję się, że taka definicja relacji poprawnego tłumaczenia mogłaby zawierać kolejny warunek, że jeśli test przechodzi dla układu w DigitalJS to przechodzi też dla układu w FIRRTLu. Jednak interpreter FIRRTLa symuluje układy mniej dokładnie niż symulator DigitalJS, ponieważ ten pierwszy posługuje się logiką dwuwartościową wspólnie z mechanizmem zatruwania podczas symulacji, a ten drugi logiką trójwartościową, dzięki której czasami nieustalona wartość nie propaguje się dalej.

Aby weryfikować poprawność tłumaczenia, należy mieć zestaw układów w FIRRTLu/Chiselu z dołączonymi testami używającymi `InterpretiveTester`. Po upewnieniu się, że wszystkie testy przechodzą, można podmienić użycie `InterpretiveTester` na `DigitalJSTester` i efekcie otrzymamy zestaw testów, który weryfikuje poprawność translacji tych układów.

Używam istniejących układów i testów (np. tworzonych w projektach open source) do stworzenia testów weryfikujących translację. Takie podejście prowadzi do dobrego testowania przypadków translacji praktycznych układów, które tworzą użytkownicy Chisela. Daje to najlepsze pokrycie testami dla najczęściej używanych konstrukcji i wyrażeń FIRRTLa.

Oczywiście testy całych, dużych układów to nie wszystko. Warto zweryfikować dokładnie tłumaczenie pojedynczych konstrukcji. Takie testy są w projekcie i zwykle składają się z:

- Bardzo prostego układu zawierającego tylko porty wejściowe i wyjściowe oraz testowaną konstrukcję.
- Testu sprawdzający poprawność wartości wyjścia na pełnym zakresie wartości wejściowych.

Przy tworzeniu takiego testu najpierw można sprawdzić poprawność testu używając `InterpretiveTester`, a potem podobnie jak dla istniejących testów zamienić go na `DigitalJSTester`.

4.2. Assert, printf, cover

Oprócz konstrukcji służących do zdefiniowania układu, języki opisy sprzętu mają również takie pozwalające na weryfikację i debugowanie układu. FIRRTL zawiera trzy instrukcję zapewniające taką funkcjonalność: assert, printf oraz cover. Pominąłem je podczas tłumaczenia, ponieważ DigitalJS nie wspiera tego typu konstrukcji oraz nie są niezbędne do poprawnego działania układu.

Symulator DigitalJS daje możliwość na pisania testbenchy w skryptowym języku Lua za pomocą pakietu `digitaljs_lua`. W planach jest wsparcie dla konstrukcji assert, printf, cover poprzez generowanie testbenchy symulujących ich zachowanie.

Rozdział 5.

Podręcznik użytkownika

Instrukcje instalacji zależności, kompilowania oraz uruchamiania zostały zweryfikowane na systemie operacyjnym Ubuntu 18.04.3 LTS, ale te same lub analogiczne instrukcje instalacji powinny działać na innych dystrybucjach systemu GNU/Linux.

5.1. Instalacja zależności

Aby projekt mógł zostać uruchomiony poprawnie wymaga on zainstalowanych programów Java JDK/JRE w wersji 11 lub wyższej, sbt w wersji 1.3.8 lub wyższej, nodejs/npm w wersji 14 lub wyższej. Można to osiągnąć za pomocą poniższej sekwencji komend:

```
> sudo apt install curl
> echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list
    .d/sbt.list
> curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0
    x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add
> curl -sL https://deb.nodesource.com/setup_14.x | sudo -E bash -
> sudo apt-get update
> sudo apt-get install sbt nodejs default-jdk
```

Następnie należy wykonać skrypt `make_deps.sh` znajdujący się w katalogu głównym projektu. Pobierze on i zainstaluje zależności biblioteczne projektu oraz zainstaluje pakiet `digitaljs_peek_poke_tester` wykorzystywany do uruchamiania testów korzystających z interpretera DigitalJS.

5.2. Kompilacja i uruchamianie

Po zainstalowaniu zależności można skorzytać z komend narzędzia sbt:

- `sbt compile` – kompiluje projekt.
- `sbt test` – uruchamia testy.
- `sbt "run <args>"` – uruchamia program z argumentami `args`.
- `sbt jacoco` – uruchamia testy oraz generuje raport pokrycia testów dostępny pod ścieżką `target/scala-2.12/jacoco/report/`.
- `sbt assembly` – kompiluje projekt oraz generuje plik JAR, który zawiera spakowany program z zależnościami. Można go uruchomić następnie za pomocą polecenia `java -jar ./target/scala-2.12/firrtl2digitaljs-assembly-1.0.jar`

Program uruchomiony bez argumentów wyświetla swoją instrukcje użycia na terminalu. Aby zaprezentować działanie program w katalogu `examples` przygotowałem parę przykładowych układów w FIRRTLu, które pokazują działanie programu. Polecenie `sbt "run /examples/<nazwa_przykładu>.firrtl"` uruchamia konwersję, która stworzy plik w formacie DigitalJS o nazwie `<nazwa_przykładu>.json`. W celu przetestowania działania wynikowego układu można skorzystać z narzędzia DigitalJS Online.

Rozdział 6.

Podsumowanie

FIRRTL2DigitalJS tłumaczy FIRRTLa na format DigitalJS poprawnie, co zweryfikowałem zestawem testów. Dzięki temu projekty DigitalJS i DigitalJS Online zyskują bezpośrednie wsparcie dla FIRRTLa i Chisela. Zestaw testów dołączony do translatora jest dość niezależny od implementacji. Pozwala na prawie dowolne zmiany w algorytmie translacji. które zachowują poprawność tłumaczenia. Jest to bardzo ważne, ponieważ translator w obecnej formie z założenia miał być prosty i poprawny jednocześnie dając możliwości rozbudowy i eksperymentowania.

FIRRTL2DigitalJS będzie używany przez DigitalJS Online, aby tłumaczyć FIRRTLa, którego używanie w tym webowym IDE będzie podobne do Veriloga. Niestety prawdopodobnie Chisel nie dostanie takiego samego wsparcia, ponieważ kompilacja i uruchamianie programów w Scali (czego wymaga Chisel) jest zbyt kosztowne dla serwera. Oznacza to, że programista Chisela korzystający z DigitalJS Online będzie kopiować wygenerowany układ za każdem razy gdy ten się zmieni. Pomysłem na rozwiązanie tego problemu z przepływem pracy jest stworzenie wtyczki, która będzie udostępniała możliwości symulatora DigitalJS, FIRRTL2DigitalJS oraz yosys2digitaljs w rozszerzalnym edytorze jakim jest Visual Studio Code.

Bibliografia

[1] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723, 1938.

[2] Irving S. Reed. Symbolic synthesis of digital computers. In *Proceedings of the 1952 ACM National Meeting (Toronto)*, ACM '52, page 90–94, New York, NY, USA, 1952. Association for Computing Machinery.

[3] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. Verilog hdl and its ancestors and descendants. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[4] Chi-Lai Huang. *Computer-Aided Logic Synthesis Based on a New Multi-Level Hardware Design Language—Lalsd Ii.* PhD thesis, USA, 1982. AAI8208556.

[5] Marek Materzok. Digitaljs: A visual Verilog simulator for teaching. In *Proceedings of the 8th Computer Science Education Research Conference*, CSERC '19, page 110–115, New York, NY, USA, 2019. Association for Computing Machinery.

[6] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free Verilog synthesis suite. 2013.

[7] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. *SIGARCH Comput. Archit. News*, 38(3):37–47, June 2010.

[8] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[9] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216, 2017.