C-011

Rust による安全なハードウェア設計環境の検討
A Safety Hardware Design Environment in Rust

高野 恵輔† 小畑 正貴‡
Keiuske Takano Masaki Kohata

1. はじめに

近年、FPGA(Field Programmable Gate Array)は、電力当たりの性能に注目され多様な分野で利用され始めている。高い並列性を持つアプリケーションが実装できることから画像処理や深層学習などの分野で特に利用が盛んである。

一般的な計算機においてソフトウェア開発を行う場合、開発者の方針や稼働時の状況など、用途に応じてプログラミング言語やフレームワークを選択し開発を行う事が出来る。対して、ハードウェア開発において開発に利用できる言語は限られている。FPGA を利用したアプリケーションを実装するにあたり、ソフトウェアと同様に用途や趣向に応じて開発言語や手法の選択が可能であれば、より開発の幅が広がると考えられる。そこで我々は、言語レベルで安全性を考慮された言語 Rust を用いたハードウェア設計ドメイン固有言語を提案する。

本研究では、初期評価としてハードウェア設計ドメイン固有言語での記述および本ツールより出力されたソースコードの行数を比較した。その結果、Verilog HDL を用いたハードウェア記述のソースコード行数に比べ 2/3 程度のソースコード行数でレジスタ転送レベルの回路設計ができることが明らかになった。

2. Rust

Rust[1][2]は Mozilla が開発を支援するオープンソースのシステムプログラミング言語である。速度、安全性、並行性の 3 つを達成することを目標にして開発が進められている。現時点で、1)速度は、ソースコードからターゲットプラットフォームに最適化してバイナリの生成を行うので C/C++と同等、もしくはそれを上回る速度での速度で計算が可能である。2)安全性は、コンパイラが、リソースの静的検証を事前に行い不正なメモリ領域へのアクセスなどを防ぐことにより安全なバイナリを生成することで実現している。3)並列性は、言語の標準ライブラリに並列機能を内包することで達成している。

3. 関連研究

これまでにハードウェア設計における簡易化、高効率化を目標にした取り組みは盛んに行われている。中でも既存のプログラミング言語を対象にした開発が多く存在する。ここでは、主に RTL 程度の細粒度のハードウェア設計に関する研究を取り上げる。

以前より Java や Python などのプログラミング言語を用いたハードウェア設計ドメイン固有言語(DSL)が提案されている。Veriloggen[3]は、Python によるハードウェア設計 DSL である。Python の持つ ast モジュールを使用せず、階層化した Verilog HDL の AST として直接構築をしている点が特徴である。JHDL[4]は、Java によるハードウェア設計 DSL である。視覚的にテストが可能なツールとなっている。

† 岡山理科大学工学研究科システム科学専攻
‡ 岡山理科大学工学部情報工学科

```
1:  let mut m = VModule::new("LED");
2:  let clk = m.Input("CLK", 1);
3:  let rst  = m.Input("RST", 1);
4:  let btn1 = m.Input("BTN1", 1);
5:  let btn2 = m.Input("BTN2", 1);
6:  let mut led  = m.Output("LED", 8);
7:  let mut fsm = Clock_Reset(clk.clone(),rst.clone())
8:    .State("State")
9:    .AddState("IDLE").goto("RUN", F!(btn1 == 1))
10:   .AddState("RUN").goto("END", F!(btn2 == 1))
11:   .AddState("END").goto("IDLE", Brank!());
12: let run = fsm.Param("RUN");
13: let fstate = m.FSM(fsm);
14: m.Assign(led._e(_Branch(F!(fstate == run), _Num(8), _Num(0))));
15: m.endmodule();
```

図 1　LED 回路 Verugent 記述例

FSL[5]は、論理合成システム PARTHENON で使用する HDL である。SFL[6]の思想を引き継ぎつつ抽象度の高い記述を目標に開発が行われている。

4. Verugent

Verugent(Verilog from Rust : Generation Toolkit)とは、筆者の提案する Rust を用いたハードウェア設計 DSL である。Verilog の構文に対応させた構造体を定義することで Rust 上に Verilog 用の AST 実装を実現している。生成構文は Verilog2001 に準拠し、論理合成可能な構文を主としたライブラリとして作成を進めている。次項より図 1 に示す LED 回路の記述例より生成構文とコード生成ついて説明する。

4.1 モジュールの作成と信号線、パラメータの生成

Verugent では、最初に構文格納用のトップモジュール(VModule)を生成する必要がある。図 1、2 行目のように VModule::new()を使用する。引数にモジュール名を文字列で渡すことによりモジュールのオブジェクトが生成される。

パラメータ、入出力ポート、reg と wire の追加は、VModule のメソッドにより実現している。図 1 の 2-6 行目のように記述することで入出力の定義ができる。レジスタ定義された出力は Reg_Output メソッドを使用することで追加ができる。これらのメソッドは全て引数に名前を文字列、配線幅を i32 の型で要求する。

4.2 組み合わせ回路記述(assign, function)

配線の恒久的な信号代入は assign 構文や function 構文を用いることで実装でき、生成された回路は組み合わせ回路となる。図 1、15 行目のように Assing メソッドを使用する事で Verilog の assign 文と同等の組み合わせ回路記述ができる。Function と同様の記述を行う場合は、まず func 関数を使用し function 用の AST 構造体を生成する。ポートの追加および if, else や case などの分岐構文は AST 構造体のメソッドより追加する。その後、VModule の Function メソッドに読み込ませることで使用が可能となる。Assign への接続は Func!()マクロで行う。

```
1: m.Always(Posedge(clk.clone())
2:   .If(F!(rst != 1), Form(F!(done =0)))
3:   .Else_If(F!(btn0 == 1), Form(F!(data = 10)))
4:   .Else(Form(F!(data = 20))
5:         .Form(F!(done = 1)))
6:   .Case(select.clone())
7:         .S(_Num(1),Form(F!(o_1 = 1)))
8:         .S(_Num(2),Form(F!(o_1 = 2))));
```

図 2　順序回路記述

```
 1: module LED (
 2:     input  CLK,
 3:     input  RST,
 4:     input  BTN1,
 5:     input  BTN2,
 6:     output [7:0] LED
 7: );
 8:     localparam IDLE = 0;
 9:     localparam RUN = 1;
10:     localparam END = 2;
11:     reg [31:0] State;
12:     reg [31:0] State_Next;
13:     assign LED = (State==RUN)? 8: 0;
14:     always@(posedge CLK or posedge RST) begin
15:         if (RST == 1) begin
16:             State <= IDLE;
17:         end
18:         else begin
19:             State <= State_Next
20:         end
21:     end
22:     always@(posedge CLK) begin
23:         case(State)
24:             IDLE : begin
25:                 if(BTN1==1&&RST!=1)
26:                     State_Next <= RUN;
27:             end
28:             RUN : begin
29:                 if(BTN2==1)
30:                     State_Next <= END;
31:             end
32:             END : begin
33:                 State_Next <= IDLE;
34:             end
35:         endcase
36:     end
37: endmodule
```

図 3　LED 回路の Verilog 出力結果

4.3 順序回路記述(always)

VerilogHDL でクロックを基準に動作する回路を実装するには always 構文を記述することで可能となる。Verugent でも.Always()のメソッドを使用することで同等の記述が可能である。Always メソッドは、前述の Assign、Function メソッドと同様に、専用メソッドにより構築された Always_AST を引数としている。図 2 に順序回路の記述例を示す。

4.4 FSM(Finite State Machine)記述

本ツールの Always メソッドで FSM の記述は可能ではあるが Always メソッドで FSM 記述を行うと、Verilog の always 構文の使用時と同様にコードの増量から複雑になり、また管理に不便である。そこでより簡易な記述で FSM を記述できるように本ツールには、FSM を構築する機能を実装した。図 1 、7-13 行目に実装例を示す。この例は、IDLE、RUN、END の 3 ステートで実装された FSM 記述である。

4.5 Verilog コード出力

コードの出力には、AST の構築が完了した VModule から endmodule メソッドを呼び出すことで出力することが可能となっている。現時点では標準出力からのみの出力をサポートしている。ファイルへの出力はコンソールよりパイプなどを利用する。図 3 に、図 1 の LED 回路を出力した結果を示す。

表 1　コード行数と削減率

Circuit	Verugent(行数)	Verilog HDL(行数)	Verugen/Verilog (%)
LED	15	37	40.5
FIFO	32	44	72.7
UART	95	125	76.0

5. 評価

初期評価として、Verugent 記述よるソースコードの総行数と出力された Verilog HDL ソースコードの行数を比較し本ツールのコード生産効率を評価する。生成された HDL および Verugent 記述は、空行を行数として見なさず計測した。今回は、図 1 に示した LED 点灯回路に加え、FIFO、UART 送受信回路を Verugent により実装した。表 1 に Verugent および Verilog のソースコード行数と削減率を示す。結果から LED 点灯回路には FSM 実装機能を用いているため、コードの短縮がされており生産性が高いことが示される。FSM 構造体は、コード出力時にステートの定義パラメータなどを自動生成するため、より少量の記述で効率良く回路の実装ができると考えられる。FIFO や UART の回路も記述量は LED 回路ほどではないが、およそ 70%程度の削減ができている。これにより HDL の記述よりおよそ 2/3 に短縮した記述で回路の設計ができる事が示された。

6. おわりに

本稿では、Rust を用いたオープンソースのハードウェア設計ドメイン言語の Verugent を提案し、評価として記述コード行数と出力された Verilog のコード行数を比較した。その結果、Verilog に比べ 2/3 程度のコードの量で記述できることが明らかとなった。

今後は、様々なハードウェアを本ツールにより記述しコードの検証を進めていくとともに、Rust のコンパイラが持つ高い検証力を用いて、記述時に破綻の少ないコードを生成できるような検証機能、更に Rust そのものが持つ並列性を生かせるツールとしての実現を目指す。また、生成された回路の安全性についての検証も進める必要がある。

なお、今回提案した Verugent は Github[7]にて公開している。本稿内に示した例や各メソッドの説明を掲載しているので、詳細についてはそちらを参照願いたい。

参考文献

[1] Nicholas D. Matsakis, Felix S. Klock, II, “The rust language”, ACM SIGAda annual conference on High integrity language technology, Vol. 34, No.3 (2014).
[2] The Rust Programming Language, https://www.rust-lang.org/
[3] 高前田伸也, “Python を用いた高水準ハードウェア設計環境の検討”, 信学技報(RECONF), Vol.115, No.36 , pp.21-26(2015).
[4] P.Bellows, B. Hutchings, “Jhdl-an hdl for reconfigurable system”, FPGAs for Custom Computing Machines, IEEE Symposium (1998).
[5] 渡邊誠也, 名古屋彰, “オブジェクト指向/関数言語をベースとするハードウェア記述言語 FSL の設計”,信学技報, Vol.115, No.37, pp. 27-32, (2015).
[6] PARTHENON HOME PAGE, http://www.parthenon-society.com/archive/NTT/.
[7] “Verugent”, https://github.com/RuSys/Verugent/