RTL回路に対するファジングを用いたバグ検出の有効性評価

伊沢 亮一[1]　藤原 吉唯[1]　金谷 延幸[1]　井上 大介[1]

概要：本研究では RTL（Register Transfer Level）回路に対してフィードバック・ドリブン・ファジングを行い，バグ検出におけるファジングの有効性を評価する．RTL 回路にバグ検出のためのアサーションを埋め込み，シミュレータで動作させながらファジングツール AFL（American Fuzzy Lop）を使用する．既存研究では AFL を用いた RFUZZ が提案されているが，サイクル数とコードカバレッジのみが評価されている．本研究の貢献はバグ検出に要したサイクル数まで含めて AFL の有効性を評価する点にある．一般に公開されている IP（Intellectual Property）コア 10 個に対して AFL とランダム検証を評価したところ，ともに同じ 7 個の IP コアのバグが検出できたが，ランダム検証の方が要したサイクル数は少ない結果となった．

Evaluation of Fuzzing for Finding Bugs in RTL Circuits

Ryoichi Isawa[1]　Yoshitada Fujiwara[1]　Nobuyuki Kanaya[1]　Daisuke Inoue[1]

Abstract: We evaluate how well feeedback-driven fuzzing can find bugs in RTL circuits. In this evaluation, we insert assertions to RTL circuits and run those circuits in simulation, while fuzzing them with AFL (American Fuzzy Lop), a fuzzing tool. Although Kevin Laeufer et al. propose a fuzzing system using AFL named RFUZZ, they only evaluate how efficiently it can raise code coverage of RTL circuits. In addition to this, our contributions is to measure how many simulation cycles AFL takes untils bugs are found. We also evaluate that of a random test pattern generater as a benchmark. With an evaluation using 10 open-sourced IP (Intellectual Property) cores, we confirm that both AFL and a random test pattern generater were able to found a bug in seven IP cores whereas a random test pattern takes less simulation cycles than AFL for finding bugs.

1. はじめに

FPGA（Field Programmable Gate Arrays）はロボット操作や核関連施設など幅広い用途で活用されている [1]．一方, 不具合や不正な機能, ハードウェアトロジャン [2], [3], [4] などが埋め込まれていることが懸念されており，不正な動作をすると重大なインシデントが起こりえるため，FPGA を検証する技術が求められている．

FPGA へのプログラミングに用いられる RTL（Register Transfer Level）で設計される論理回路（以下，RTL 回路と呼ぶ）は Verilog や VHDL などの HDL（Hardware Description Language）で記述できる．ある検証対象の RTL 回路（以下，DUT（Device Under Test）と呼ぶ）を Vivado[5] などのシミュレータで動作させながら検証するとき，DUT に入力（例: Verilog で記述されたモジュールの input）を与える．このとき，人手で作成した入力パターンだけの場合, DUT のある条件分岐先のコードブロックが実行されず意図しない入力に対する挙動が未検証となる懸念がある．

RTL 回路に対する一般的な検証方法としてランダム検証がある．これは DUT に対してランダムな入力を与えて挙動を観測する検証方法で，自動化することで様々な入力パターンで DUT を検証することが容易になる．しかしながら，DUT の入力のビット幅が多い場合は入力空間が大きくなるため，ランダムに入力を生成する方法では確率的に未検証のコードブロックを実行するための条件を満たすことが難しいことがある．

RFUZZ[6] はソフトウェアの分野ではよく使用されているファジングツール AFL（American Fuzzy Lop）[7] を

[1] 国立研究開発法人情報通信研究機構
NICT, Koganei, Tokyo 184–8795, Japan

RTL 回路の検証に用いるシステムである．AFL はある入力を DUT に与えて，新しい条件分岐先が実行されたとき，その入力をもとに僅かに値を変更することで次の入力を生成する．これはある条件を満たす入力を少しだけ変えることでコードカバレッジ（実行されたコードの割合）を上げるなどの利点があるという考え方に基づく．例えば，その条件にネストされている条件を満たすことができるということを仮定している．RFUZZ の文献 [6] ではシミュレーションサイクル数と各サイクル数におけるコードカバレッジを調べるだけに留まっており，不正な動作の検出までは行っていない．

本研究ではオープンソースとして一般に公開されている IP（Intellectual Property）コアに着目し，効率的なバグ検出という観点で AFL を評価する．本評価では OpenCores と Github から合計 10 個の Verilog で記述された IP コアを用いる．これらの IP コアのバグのあるリビジョンを使用して，AFL によるファジングを行いバグの検出を試みる．評価の指標は 3 つあり，バグ検出の可否，何サイクルでバグ検出ができたか，コードカバレッジを効率的に上昇させることができたかである．比較対象としてランダム検証も用いる．評価結果としては AFL は 7 個の IP コアに対してバグ検出ができ，ランダム検証でも AFL と同じ IP コアのみバグ検出ができた．バグ検出のサイクル数やカバレッジの上昇率に関しては，少なくとも本稿で用いた IP コアに対してはランダム検証の方が良いという結果になった．

2. 基礎知識

2.1 ランダム検証

図 1 に Verilog のサンプルーコードを示す．このコードのモジュール `example` は入力ポート `clock` と `reset`，`in_a` を有し，出力ポート `out_c` を有する．また，レジスタ `reg_state` と `reg_tmpout` を有する．このモジュールの検証方法として，入力ポートに任意の値を入力し，Vivado[5] や Verilator[8] などのシミュレータや FPGA（Field Programmable Gate Array）により動作させることで，レジスタの値や出力ポートの値が仕様外であればエラーとして検出する方法があげられる．図 1 のコードでは，19 行目に不具合があり，19 行目にアサーションを記述しておくなどしてその不具合を検出するものとする．

ランダム検証とはモジュールの入力ポートに対し，ランダムな値を与える検証方法である．モジュール `example` の `in_a` は 16 ビット幅の値を入力とするため, $0 <= \texttt{in\_a} < 2^{16}$ の範囲でランダムに値を選択する．ただし，モジュールの同期信号 `clock` とレジスタのリセット信号 `reset` は除く．手動でモジュールに適した固有の入力パターンを作成する方法と比べ，ランダム検証はモジュールに依存せず入力を機械的に決定できるため自動化が容易であり，またモジュールの設計者が意図しない入力により発生する不具合を発見できる可能性がある．

```
 1: module example(clock,reset,in_a,out_c);
 2:   input clock, reset;
 3:   input [15:0] in_a;
 4:   output out_c;
 5:   reg [1:0] reg_state;
 6:   reg reg_tmpout;
 7:
 8:   always@(posedge clock) begin
 9:     if (reset == 1) begin
10:       reg_state <= 0;      /* Branch 1 */
11:       reg_tmpout <= 0;     /* Branch 1 */
12:     end
13:     else if (in_a == 16'h1234)
14:      reg_state <= 1;       /* Branch 2 */
15:     else if (reg_state == 1 && in_a == 16'h5678)
16:       reg_state <= 2;      /* Branch 3 */
17:     else if (reg_state == 2)
18:       reg_tmpout <= 1;     /* Branch 4 */
19:     ... /* <- Suppose some bugs are found here.*/
20:     end
21:     assign out_c = reg_tmpout;
22: endmodule
```

図 1 Verilog サンプルコード

2.2 コードカバレッジ

Verilog コードが仕様通りの動作をするかを検証する上で，検証の網羅性を示す指標としてコードカバレッジ（網羅率）があり，コードカバレッジの種類を表 1 に示す [9]．ここで表中のステートメントとは Verilog コードの実効行（コメントを除く行）を意味し，ブランチとは条件分岐においてある条件が満たされたときに実行されるステートメントである．図 1 のサンプルコードであれば Branch 1〜Branch 4 がブランチである．カバレッジが高いほど，実行されたステートメントやステートメントの組み合わせは多く，未検証の箇所が少ないことを意味する．

本稿ではブランチカバレッジを採用し，ブランチカバレッジを「トグルした条件の数 / 条件の総数」と定義する．ここでトグルした条件とはある時点においてこれまでに True および False に変化した条件を意味する．モジュール `example` の `(reset == 1)` であれば，`reset` に 0 および 1 が入力されたことがあれば，`(reset == 1)` はトグルした条件となる．また，モジュール `example` に含まれる条件の総数は 4 であり，ある時点までに `(reset == 1)` と `(in_a == 16'h1234)` のみトグルしていた場合は，その時点でのブランチカバレッジは 0.5 (50%) となる．

表 1　カバレッジの種類 [9]

カバレッジの種類	内容
ステートメント カバレッジ	各ステートメントの実行の有無
ブランチ カバレッジ	条件分岐の各ブランチの実行の有無
コンディショナル カバレッジ	条件分岐の条件が複数ある場合， それぞれの条件の True/False
パスカバレッジ	複数の条件分岐がある場合，選択 されたブランチの組み合わせ

2.3　ステートの遷移および問題定義

ステートの遷移: 図 1 のモジュール `example` はモジュールのステートを保持するためのレジスタ `reg_state` を有し，always 文によりステート遷移が記述されている．その always 文には 4 つの条件分岐が含まれ，各分岐はあるステートに対応し，入力に応じて次にどのステートに遷移するかを表している．例えば，15 行目の else if 文は「`reg_state` が 1 のステートにおいて，入力 `in_a` が 5678（16 進数）であれば，`reg_state` が 2 のステートに遷移する」ことを表している．このように HDL コードで記述されたモジュールは「レジスタによってステートを実現し，あるステートにおける入力によって次のステートを決定するステートマシンをもつ順序回路」[10], [11] として記述される．

問題定義: 一般的に入力幅が Nbit のモジュールでは，あるステートに遷移する適切な入力を得る確率 X は $X = l(\prod_{i=1}^{N} 1/2)$ $(= l/2^N)$ となる．ここで l はあるステートに遷移させることができる入力値の数である．l が小さいステートかつ N が大きいモジュールにおいて，ランダム検証ではそのステートに遷移する確率が低くなり，カバレッジをあげるための試行回数が多くなってしまう．

3.　ファジングシステムの実装

本章ではファジング対象の Verilog コードを入力とするファジングシステムの実装について述べる．本実装には RFUZZ[6] を組み込む．

3.1　RFUZZ

RFUZZ は AFL（American Fuzzy Lop）[7] を用いて RTL 回路に対してファジングをするシステムである．AFL のフィードバック・ドリブン・ファジングのアルゴリズムに従い，ある入力によって新規ブランチが実行されると，その入力をもとに次の入力を生成することで，カバレッジを効率的に上昇させることを目的にしている．

RFUZZ は FIRRTL[*1]形式の RTL 回路を読み込み，カバレッジ計測と AFL へのフィードバック（どのブランチが実行されたか）の仕組みを DUT に埋め込んだ上で，Verilog コードを生成する．その後，Verilog コードを Verilator[8] により C++ソースコードに変換して実行可能なバイナリファイルを生成する．このバイナリファイルを実行し，AFL が生成した入力を DUT に与えながら，1 シミュレーションサイクル動作させた後のフィードバックを受けとってから AFL に渡し，AFL が次の入力を生成する．

3.2　Verilog コードを扱うための実装

RFUZZ は Verilog や VHDL で記述された DUT をファジング対象にすることができない．Verilog を FIRRTL を変換する機能を持つツールとして yosys[*2]があるが，我々が試したところ Verilog から機能的に等価な FIRRTL 形式が生成されない例が散見された．オープンソースで公開されている IP コアには Verilog や VHDL で記述されているものが多いため，Verilog コードや VHDL に対応することで利便性が向上する．本稿では Verilog で記述された IP コアを DUT として扱えるようにファジングシステムを実装する．

図 2 にファジングシステムの構成と処理の流れを示す．コード埋め込み処理（Instrumentation-code injection）では，まず DUT の Verilog コードを直接読み込み，python パッケージである Pyverilog[12] を用いて AST（Abstraction Syntax Tree）を構築し，if 文や case 文などの条件分岐のノードを探索する．条件分岐のノードがあれば，その条件の真/偽（1/0）を出力する output のノードを生成して AST に挿入する．例えば，DUT のモジュールに if (in_a == 16'h1234) という条件があれば，出力ポート（`output covarage_out_01`）の宣言に対応するノードとその出力ポートに条件の真/偽を出力するためのステートメント（例: `assign covarage_out_01 == (in_a == 16'h1234);`）のノードを AST に挿入する．シミュレーション時にはその出力ポートの値を見ることで，対応する条件の真偽が分かることから実行されたブランチがわかる．全ての条件に対応する出力ポートのノードを挿入した AST から Verilog コード（Verilog code with instrumentation）を pyverilog で生成する．

Verilator により DUT の Verilog コードを C++言語に変換し，そのモジュールをインスタンスとして操作するテストベンチの C++コードを生成する．このテストベンチはファザー（AFL）が生成した入力を DUT に渡し，出力ポートから得た値をファザーに返す役割を担う．また，シミュレーション開始からのシミュレーションサイクル数と各サイクルにおけるカバレッジをログに残す．テストベンチと DUT を Verilator で実行可能なバイナリに変換し，そのバイナリを Linux 上で実行するとシミュレーションが開始される．なお，ランダム検証を実施するときにはファザーがランダムな入力を生成してテストベンチに渡す．

[*1] https://bar.eecs.berkeley.edu/projects/firrtl.html

[*2] https://github.com/YosysHQ/yosys

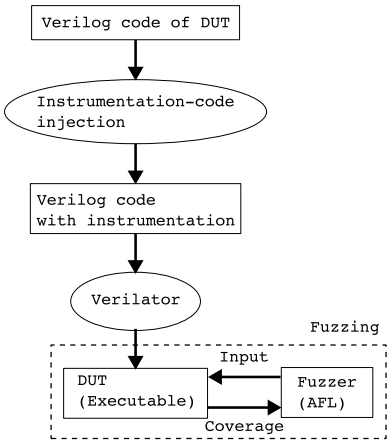

図 2 ファジングシステムの構成と流れ

表 2 評価用データセット（数字・アルファベット順）

IP コア名	入力ビット幅	条件数	コード行数
8/16/32 bit SDRAM Controller	48	36	411
Adjustable Frequency Divider	7	14	152
AltOr32 - Alternative Lightweight OpenRisc CPU	217	175	2869
Amber ARM-compatible core	207	337	2045
Another Wishbone Controlled UART	10	39	407
USB 1.1 PHY	4	46	476
USB 2.0 Function Core	27	71	8048
USB CDC Device	25	359	3173
Srdy-Drdy Library	35	16	186
Wishbone LPC Host and Peripheral Bridge	81	79	1728

4. ファジングの有効性評価

4.1 評価の方針

本評価ではバグがある IP コアにアサーションによるバグ検出の仕組みを入れ，その IP コアに対して AFL を使用する．カバレッジをどの程度効率的に上げることができるかを計測するとともに，どの程度のシミュレーションサイクルでバグが検出できるかを評価する．比較対象としてランダム検証も使用する．

4.2 データセット

表 2 に本評価に使用する IP コアを示す．表中の入力ビット幅は IP コアのトップモジュールに与える各入力のビット幅を合計したものである．条件数はトップモジュール内の条件数およびトップモジュールに接続されているサブモジュールの条件数の合計である．コード行数はトップモジュールとサブモジュールの Verilog コード（*.v および*.vh）の行数の合計である．それぞれ IP コアの規模を示すために記載している．図 1 の example モジュールであれば入力ビット幅は 18 ビット，条件数は 4，行数は 22 である．これらはオープンソースとして一般に公開されており，OpenCores もしくは Github から取得した．入手先 URL および IP コアのリビジョンは付録 A.1 に記載する．

各 IP コアのバグ履歴を参照し，バグを含むリビジョンを選択した．その Verilog ソースコードに対して，本来であれば満たされるべき条件をアサーションとして記述した．その条件が満たされないとき，バグとして検出する．各 IP コアのアサーションの条件は付録 A.2 に記載している．

4.3 評価方法

AFL およびランダム検証にて，各 IP コアをシミュレーションし，各シミュレーションサイクル数におけるブランチカバレッジとバグ検出に要したシミレーションサイクル数を計測した．なお，1 シミュレーションサイクルは「モジュール同期用クロック（`clock`）が立ち上がり，その次に 0 に立ち下がる直前まで」と定義する．また，シミュレーション開始時に DUT のリセット（リセット信号（rst）に 1 を入力した後，0 を入力する）を 1 度だけ実施するものとする．

8/16/32 bit SDRAM Controller と Adjustable Frequency Divider を除いて，各 IP コアに対するシミュレーションは AFL で 3 分間，ランダム検証で 3 分間実施した．8/16/32 bit SDRAM Controller と Adjustable Frequency Divider に関しては AFL で 3 分間実施して，アサーションによるバグ検出ができたが，ランダム検証の方ではバグ検出ができなかった．そのため，これら 2 つの IP コアのランダム検証のみ 30 分間実施した．シミュレーションに用いた PC の CPU は Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz，物理メモリの容量は 16GB である．

4.4 評価結果

4.4.1 バグ検出に成功した IP コア

ランダム検証と AFL ともにバグの検出に成功した IP コアは 8/16/32 bit SDRAM Controller，Adjustable Frequency Divider，Amber ARM-compatible core，Another Wishbone Controlled UART，Srdy-Drdy Library，Wishbone LPC Host and Peripheral Bridge だった．それぞれのシミュレーションサイクル数と各サイクルにおけるカバレッジ，バグが検出されたときのサイクル数を図 3〜図 8 に示す．各図の縦軸はブランチカバレッジを示しており，横軸はシミュレーションサイクル数を示しており対数目盛にしている．ランダム検証と AFL でバグが検出されたシミュレーションサイクル数にそれぞれ ‘●’ と ‘▲’ をプロットしている．

8/16/32 bit SDRAM Controller（図 3）に対してランダム検証では 6 サイクルでカバレッジ 0.30 のときにバグ検

出をしている．一方，AFL では 231 サイクルでカバレッジ 0.25 のときにバグ検出をしている．本 IP コアのアサーションの条件が 2 ビットの input（sdr_width）に依存しており，取り得る値の種類が少なく，確率的に検出しやすいバグと考えられる．そのため，ランダム検証と AFL ともに比較的短いシミュレーションサイクル数でバグを検出できている．また，ランダム検証は 52 サイクルでカバレッジが 0.88 に到達し，AFL ではカバレッジが 0.88 になるまでに 81,314 サイクルを要している．AFL は基本的には前の入力を 1 ビット変化させて次の入力を生成しているため，入力を少しずつ変化させながら一つのブランチを検証するような動きになっている．そのため，新しいブランチの条件が満たされず，カバレッジが上がらなかった．

Adjustable Frequency Divider（図 4）に対してランダム検証では 1,109 サイクルでカバレッジ 0.57，AFL では 81,294 サイクルのとき 0.50 のときにそれぞれバグを検出している．また，ランダム検証は 548,683 サイクルでカバレッジ 0.78 に到達し，AFL は 90,018 サイクルでカバレッジ 0.78 に到達している．ランダム検証の方がカバレッジ 0.78 までのサイクル数が多い要因として，本 IP コアのモジュール odd(clk, out, N, reset, enable, o_assert）の input N に 1 サイクル前と異なる値を入力するとモジュールにリセット後の状態に戻る仕様があげられる．ランダム検証ではランダムに値を入力するためリセット後の状態に頻繁に戻り，実行されていないブランチがあった．

Amber ARM-compatible core（図 5）に対してランダム検証では 34 サイクルでカバレッジ 0.72，AFL では 334,594 サイクルでカバレッジ 0.68 のときにバグを検出している．AFL の方がサイクル数を要しているのは上述の通り入力を 1 ビットずつ変更する方法が要因であると思われる．AFL のサイクル数が 10^3 付近からカバレッジが上がっている．これは新しいブランチを実行するための条件を発見した際にはその入力を少しずつ変化させながら以降の入力を生成するため，同じような入力で実行されるブランチがあった場合はカバレッジが上がりやすくなるためだと考えられる．

Another Wishbone Controlled UART（図 6）はランダム検証 9 サイクルでカバレッジ 0.61，AFL は 77 サイクルでカバレッジ 0.23 のときにバグを検出しており，Srdy-Drdy Library（図 7）はランダム検証で 88 サイクルでカバレッジ 0.5，AFL は 81,327 サイクルでカバレッジ 0.50 のときにバグ検出している．カバレッジの上がり方は図 5 と傾向が似ている．

Wishbone LPC Host and Peripheral Bridge（図 8）ランダム検証では 7 サイクルでカバレッジ 0.10，AFL では 386 サイクルでカバレッジ 0.03 のときにバグを検出している．カバレッジに関してはランダム検証がカバレッジ 0.46 から上昇していないのに対して，AFL では 0.89 まで上昇させることができている．これは本 IP コアのモジュール wb_lpc_host が 13 種類のステート（LPC_ST_IDLE や LPC_ST_START，LPC_ST_CYCTYP など）があり，ランダム検証では一部のステートに遷移させることができなかった．本 IP コアのカバレッジに関して，AFL の入力を少しずつ変化させる方法が有効だった事例といえる．

本データセットに対するアサーションの結果としてはランダム検証の方が有効な結果となった．これは AFL があるブランチの条件を満たすようなモジュールへの入力をもとにして，その入力の値少しずつ変化させながら何度も検証する方針が影響しているものと考えられる．一方，Wishbone LPC Host and Peripheral Bridge（図 8）のようにステートが多い IP コアに対してはカバレッジを上げるという点では AFL のほうが有効な事例も見られた．

4.4.2 バグ検出に失敗した IP コア

AltOr32 - Alternative Lightweight OpenRisc CPU（以下，AltOr32 と呼ぶ）と USB 1.1 PHY，USB 2.0 Function Core，USB CDC Device に対してはランダム検証と AFL ともにバグの検出できなかった．それぞれのシミュレーションサイクル数と各サイクルにおけるカバレッジは図 9〜図 12 に示す．

AltOr32（図 9）では ALU（Arithmetic Logic Unit）に対して仕様として正しい CPU 命令が入力されて，割り込み処理（Syscall/Break）が実行されていないとき，SR（Supervision register）のコントロールフラグをクリアするバグがあった．しかしながら，ランダム検証と AFL ともに，正しい CPU 命令の入力した上で割り込み処理無しの状態にすることができず，そのバグの箇所を実行できなかった．CPU 命令（opcode_i）は 32 ビット幅であるため確率的に正しい仕様になり辛いことも要因としてあげられる．

USB 1.1 PHY（図 10）や USB 2.0 Function Core（図 11），USB CDC Device（図 12）はいずれも USB 関連の IP コアである．USB 1.1 PHY であれば DPLL（Digital Phase-Locked Loop）や USB 2.0 Function Core であれば TX/RX（送受信）などの USB 関連のプロトコルに従い，モジュールに入力を与えなければならない．ランダム検証や AFL ではそのプロトコルに従う入力を与えられないため，カバレッジがあがらず，またバグを含むブランチを実行・検出することができなかった．

AltOr32 のようにモジュールへの入力のビット幅が大きく，入力の仕様が決まっているものや USB のようにプロトコルに従った入力を与えないとステートが変化しない IP コアについてはランダム検証と AFL は有効ではないという結果となった．

5. 関連研究

5.1 カバレッジ向上を目的とした検証

本稿では RFUZZ を実装に用いることで AFL の評価を

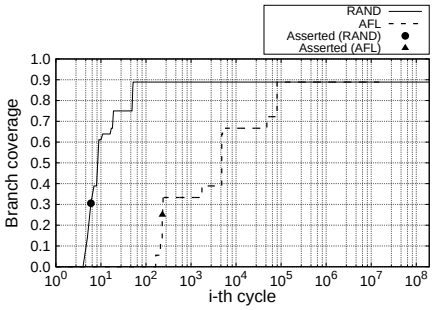

図 3 8/16/32 bit SDRAM Controller

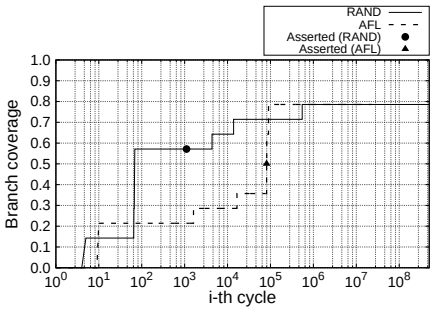

図 4 Adjustable Frequency Divider

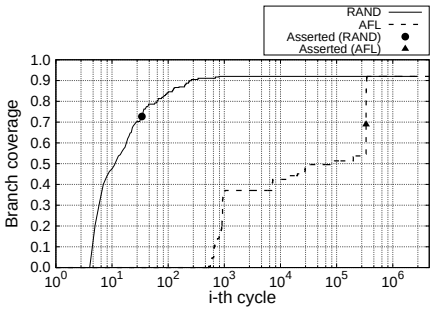

図 5 Amber ARM-compatible core

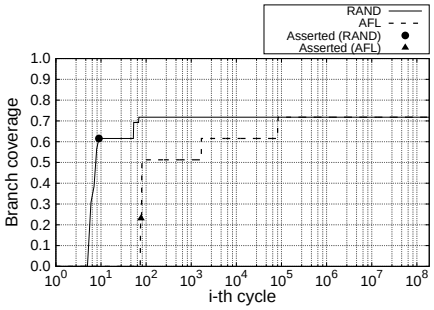

図 6 Another Wishbone Controlled UART

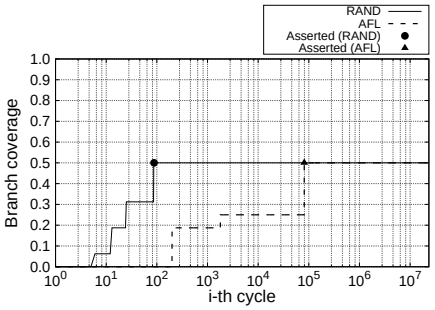

図 7 Srdy-Drdy Library

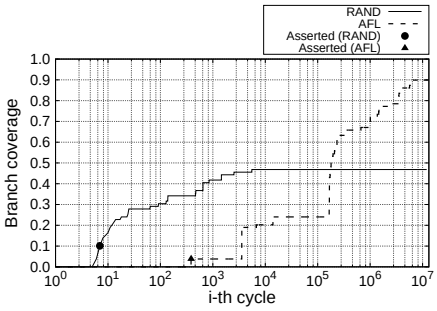

図 8 Wishbone LPC Host and Peripheral Bridge

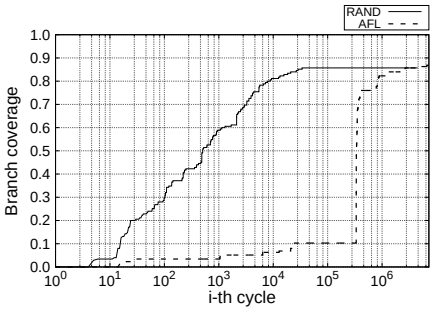

図 9 AltOr32 - Alternative Lightweight OpenRisc CPU

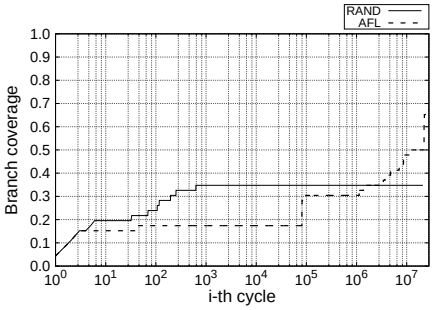

図 10 USB 1.1 PHY

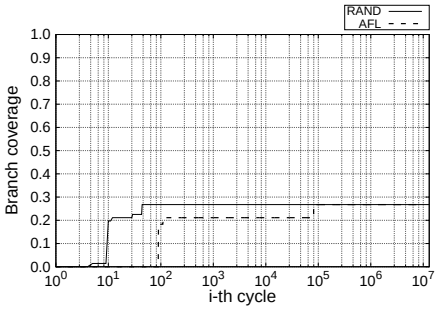

図 11 USB 2.0 Function Core

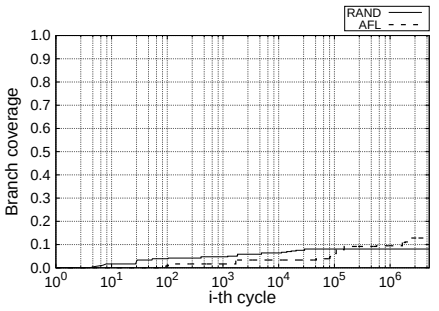

図 12 USB CDC Device

行ったが，他にもカバレッジ向上を目的とした既存の検証方法が提案されている．

STAR[13] は効率的にカバレッジを向上させることを目的とし，ランダム検証に SAT（boolean SATisfiability problem）ソルバを組み合わせて RTL コードの検証を行う．あるサイクルのシミュレーションにおける，各条件の一部を反転させて，次のサイクルでは別のブランチを選択する．STAR では SAT ソルバを利用しているため，2 値をとる入力（1 ビットの input）しか扱えない．

Ghosh らの手法 [14] はモジュール（HDL コード）中のあるステートメントに指定し，そのステートメントがあるブランチを実行するためのモジュールへの入力を自動生成することを目的とする．Ghosh らの手法では VHDL を ADD（Assignment Decision Diagram）に変換して，ある選択したステートメントに関連するステートメントや条件を抽出する．Ghosh らの手法ではステートメントを 1 つ指定し，そこから遡って，そのステートメントを実行している．

5.2 エラー検出

本節では回路に含まれるエラーを検出することを目的とした研究について述べる．

Sheeran らの手法 [15] はモジュールの各出力ポートの値の組をステートとして表し，各ステートが保持する値およびステート間の遷移は仕様通りかを検証することを目的としている．例えば，モジュールの 1 ビットの出力ポート (a，b，c) の値と遷移を $(1,0,0) \rightarrow (0,0,1)$ のように表し，あるステート間での遷移が本来起こり得ない場合，または本来取り得ない値（例: a と b が 0 のとき，c は 0 にならない）のとき，エラーとして検出する．

Ciesielski らの手法 [16] は多項式の算術演算回路の入出力が仕様通りかどうかを検証することを目的としている．本手法では回路（ゲートレベル）の出力から回路の入力まで遡りながら演算式を自動生成することを特徴とし，事前に与えられる出力の仕様と演算式を比較してエラーがないかを検出する．

6. まとめ

本稿では RTL 回路の検証に AFL がどの程度有効かをオープンソースで公開されている 10 個の IP コアを用いて評価した．7 個に対してはバグ検出は可能であったが，ランダム検証と比べてバグ検出までに要するシミュレーションサイクル数が多いことが分かった．これは本評価で用いた IP コアに対してはカバレッジを効率的に上げることが難しく，バグを含む箇所が実行されるまでサイクル数を要したためである．

バグを含む箇所に検討がついている場合は，特定の箇所を実行することに焦点をおいた DirectFuzz[17] などのファジングシステムの活用が期待できる．今後，その有効性を評価する予定である．

参考文献

[1] Melanie Berg, Kenneth LaBel, and Jonathan Pellish. New developments in fpga: Seus and fail-safe strategies from the nasa goddard perspective, 2016. https://ntrs.nasa.gov/api/citations/20160014713/downloads/20160014713.pdf.

[2] Oliver Söll, Thomas Korak, Michael Muehlberghuber, and Michael Hutter. Em-based detection of hardware trojans on fpgas. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 84–87, 2014.

[3] Apostolos P. Fournaris, Lampros Pyrgas, and Paris Kitsos. An fpga hardware trojan detection approach based on multiple parameter analysis. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 516–522, 2018.

[4] Sanchita Mal-Sarkar, Robert Karam, Seetharam Narasimhan, Anandaroop Ghosh, Aswin Krishna, and Swarup Bhunia. Design and validation for fpga trust under hardware trojan attacks. *IEEE Transactions on Multi-Scale Computing Systems*, Vol. 2, No. 3, pp. 186–198, 2016.

[5] Inc. Xilinx. Vivado. https://www.xilinx.com/products/design-tools/vivado.html.

[6] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, 2018.

[7] Michał Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.
[8] W. Snyder. Veripool. https://www.veripool.org/verilator/.
[9] 兼平靖夫. Rtl コード・カバレッジ解析入門. In *Design Wave Magazine*, pp. 104–111. CQ 出版, 2002.
[10] Edward F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, pp. 129–153. Princeton University Press, 1956.
[11] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, Vol. 34, No. 5, pp. 1045–1079, 1955.
[12] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, Vol. 9040, pp. 451–460, Apr 2015.
[13] Lingyi Liu and Shabha Vasudevan. Star: Generating input vectors for design validation by static analysis of rtl. In *2009 IEEE International High Level Design Validation and Test Workshop*, pp. 32–37, 2009.
[14] Indradeep Ghosh and Masahiro Fujita. Automatic test pattern generation for functional rtl circuits using assignment decision diagrams. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, p. 43–48, 2000.
[15] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pp. 127–144, 2000.
[16] Maciej Ciesielski, Cunxi Yu, Walter Brown, Duo Liu, and André Rossi. Verification of gate-level arithmetic circuits by function extraction. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, 2015.
[17] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 529–534, 2021.

付　　録

A.1　IP コア入手先 URL 一覧

- 8/16/32 bit SDRAM Controller (REV 73)
  https://opencores.org/websvn/listing/sdr_ctrl
- Adjustable Frequency Divider (REV 5)
  https://opencores.org/websvn/listing/sdr_ctrl
- AltOr32 - Alternative Lightweight OpenRisc CPU (REV 40)
  https://opencores.org/websvn/listing/altor32
- Amber ARM-compatible core (REV 82)
  https://opencores.org/websvn/listing/amber
- Another Wishbone Controlled UART (REV 24)
  https://opencores.org/projects/wbuart32
- USB 1.1 PHY (REV 10)
  https://opencores.org/projects/usb_phy
- USB 2.0 Function Core (REV 14)
  https://opencores.org/projects/usb
- USB CDC Device (SHA1: fb8e64843852abd3718aa28e6fb558f15c19a50d)
  https://github.com/ultraembedded/core_usb_cdc
- Srdy-Drdy Library (REV 19)
  https://opencores.org/websvn/listing/srdydrdy_lib
- Wishbone LPC Host and Peripheral Bridge (REV 15)
  https://opencores.org/projects/wb_lpc

A.2　アサーションの条件

- 8/16/32 bit SDRAM Controller
  sdrc_req_gen.v の sdrc_req_gen にて `(sdr_width == 2'b00)||(sdr_width==2'b01 && req_addr_int[0] != 1'b0) || (req_addr_int[1:0] != 2'b0)` が偽のときバグとして検出.
- Adjustable Frequency Divider
  odd.v の odd モジュールにて `(assert_counter == 1'b0)` が偽のときバグとして検出.
- AltOr32 - Alternative Lightweight OpenRisc CPU
  altor32_exec.v の altor32_exec モジュールにて`!(execute_inst_r & ~stall_inst_r) || !(!inst_sys_w && !inst_trap_w) || (next_sr_r[`SR_STEP] == 1'b0)` が偽のときバグとして検出.
- Amber ARM-compatible core
  a23_execute.v の a23_execute モジュールにて `status_bits_out == BASE_status_bits_out` のが偽のときバグとして検出.
- Another Wishbone Controlled UART
  ufifo.v の ufifo モジュールにて `(o_status[0] == (!w_full_n))` が偽のときバグとして検出.
- Srdy-Drdy Library
  sd_scoreboard_fsm.v の sd_scoreboard_fsm にて `(!(state[s_read] & !ic_drdy) || (rd_en==0 && wr_en==0) )` が偽のときバグとして検出.
- USB 1.1 PHY
  usb_rx_phy.v の usb_rx_phy モジュールにて `(sd_nrzi==1'b1)` が偽のときバグとして検出.
- USB 2.0 Function Core
  usbf_utmi_if.v の usbf_utmi_if モジュールにて`!drive_k || (DataOut != 8'h00)` が偽のときバグとして検出.
- USB CDC Device
  usb_cdc_core.v の usb_cdc_core モジュールにて`!data_status_zlp_w || (set_with_data_r !=1'b0)` が偽のときバグとして検出.
- Wishbone LPC Host and Peripheral Bridge
  wb_lpc_host.v の wb_lpc_host モジュールにて `(fw_xfr==0) || state==`LPC_ST_P_DATA が偽のときバグとして検出.