一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2025-63,DC2025-135(2026-03)

CFU Proving Ground と高位合成を用いた ASIP 設計における高速な機能検証手法の検討

休石　大夢†　藤枝　直輝†

† 愛知工業大学　〒470–0392 愛知県豊田市八草町八千草 1247
E-mail: †nfujieda@aitech.ac.jp

あらまし　ASIP (Application-Specific Instruction-set Processor) の開発効率化に向け，CFU Proving Ground における高位合成を利用した設計フローに対して，新たな機能検証手法を提案する．CFU Proving Ground では，シミュレーションによるシステムの機能検証を Verilator を用いて行う．しかし，この検証は高位合成後の RTL モデルに行う，サイクル精度のものである．そのため，設計によってはシミュレーション時間が増大し，開発効率を悪化させる．本稿では，高位合成用の C++ モデルをシミュレーション環境に直接組み込み，カスタム命令に限り命令レベルでの検証を行うことで，高速な機能検証を実現する．適応ラインエンハンサをケーススタディとした評価から，本手法の ASIP 開発効率化に対する有用性を検討する．

キーワード　ASIP，CFU Proving Ground, Verilator, 高位合成，機能検証

Study on Fast Functional Verification Methodology for ASIP Design with CFU Proving Ground and High-Level Synthesis

Hiromu YASUISHI† and Naoki FUJIEDA†

† Aichi Institute of Technology,　1247 Yachigusa, Yakusa-cho, Toyota-shi, Aichi, 470–0392 Japan
E-mail: †nfujieda@aitech.ac.jp

Abstract　Toward more efficient ASIP (Application-Specific Instruction-set Processor) development, we propose a novel functional verification method for the high-level synthesis-based design flow of CFU Proving Ground. Its system-level functional verification is done by simulation using Verilator. However, it is a cycle-accurate verification with an RTL model generated by high-level synthesis. It may increase simulation time and degrade development efficiency. The proposed method enables partial instruction-level verification to custom instructions, by directly integrating the C++ model for high-level synthesis into the simulation environment, to achieve faster functional verification. Using an adaptive line enhancer as a case study, we evaluate the usefulness of the proposed method for improving ASIP development efficiency.

Key words　ASIP，CFU Proving Ground，Verilator，High-Level Synthesis，Functional Verification

1. はじめに

近年，あらゆる領域で IoT 化が加速し，端末側でデータを処理するエッジコンピューティングの重要性が高まっている [1]．エッジデバイスには，クラウドを介さないオンデバイスでのリアルタイム処理が求められるが，アプリケーションの高度化に伴い，従来の組込み向け汎用プロセッサでは処理能力が不足し，要求性能を満たせなくなってきている [2]．

この解決策として，Application-Specific Instruction-set Processor (ASIP) が有望視されている [3]．ASIP とは，対象アプリケーションの処理効率を最大化させるために，命令セットやマイクロアーキテクチャを最適化したプロセッサのことである．その実装の主な方法として，ベースとなる汎用プロセッサに対し，特定の処理を高速化するためのカスタム命令と，それを実行する専用ハードウェアの Custom Function Unit (CFU) を導入する手法が挙げられる [5]．これにより，汎用プロセッサの柔軟性を維持しつつ，演算負荷の高い処理のみを CFU で処理することで，処理効率の向上を実現する．

ASIP は高い処理効率を実現する一方で，その開発フローには大きな課題が存在する．対象アプリケーションを効率的に処理するための最適な ASIP 構成を特定するためには，しばしば相当な時間を要する点である [4]．ASIP の開発は，設計と性能評

価のループを繰り返して最適解を絞り込んでいくプロセスをとる．そもそもとして探索すべき設計空間が広いことに加え，シミュレーションや実機動作による性能評価は多くの時間を必要とする．こうした開発の効率化を支援するため，商用，オープンソース含め，現在まで多くの ASIP 開発を効率化するフレームワークが提案されてきた [6]˜ [9].

本研究では，これらのフレームワークの中でも．特に軽量かつシンプルな構造を持つ CFU Proving Ground に着目する [9]. これは，ASIP の設計から FPGA による実機での性能評価までを迅速に行うことに主眼をおくフレームワークである．5 段パイプラインの RISC-V プロセッサをベースとした SoC をひな型として提供しており，ユーザはプロセッサの全体設計を意識することなく CFU の設計に専念できる．また，構造や依存関係が明瞭で理解しやすいため，必要に応じた機能拡張が容易に可能である．CFU の設計には RTL 設計のほか，高位合成による C/C++ での CFU 設計もサポートされている．本研究では，設計効率を重視し，高位合成による CFU 設計を前提とする．

CFU Proving Ground の 1 つの問題点は，設計によっては，シミュレーションを用いたシステムレベルの機能検証に非常に多くの時間を要することである．高位合成による CFU 設計を行った場合，シミュレーションそのものが行えない場合もある．これは，検証にサイクル精度の，高位合成後の RTL モデルに対して行う方法しか提供されていないことに起因する．

そこで，本研究では，高位合成前の C++ モデルをシミュレーションに直接組み込み，部分的に命令レベルのシミュレーションを行う手法を提案する．すなわちカスタム命令に限り論理的な 1 サイクル処理として実行する．これにより，シミュレーション時間を短縮し，機能検証におけるソフトウェアとハードウェアの問題切り分けを容易にすることを目的とする．適応ラインエンハンサ (ALE) をケーススタディに，提案手法の有用性を検討する．

2. CFU Proving Ground

2.1 CFU Proving Ground の概要

CFU Proving Ground が提供するハードウェアシステムは，明確な包含関係を持つ階層構造として設計されている．最上位の `top` モジュールは，FPGA の外部インターフェース (クロック，リセット，GPIO 等) を管理し，その内部に `soc` モジュールを含む．`soc` モジュールは，RISC-V Processor (`cpu` モジュール)，メモリ (命令メモリおよびデータメモリ)，およびユーザが任意で追加する外部センサ用の通信モジュールから構成される．RISC-V Processor は 5 段パイプライン構造を持ち，その実行ステージ内に ALU や乗除算器と同等の演算器として，`cfu` モジュール (CFU Wrapper) が統合される．`cfu` モジュールは，高位合成で生成された CFU をカプセル化し，プロセッサとの標準インターフェースを提供する．これらのモジュールは，`top` モジュール → `soc` モジュール → `cpu` モジュール → `cfu` モジュールという入れ子構造として表現される．

図 1 に CFU Proving Ground の開発フローの概略を示す．開発フローは大きく分けて，CFU 設計，SoC 統合，ホストシミュレーション，FPGA 実装の 4 つのブロックから構成される．各ブロックにおける処理対象は，ユーザが設計するもの，フレームワークが提供するもの，およびフレームワークが自動生成するものに分類される．これらのブロックを反復的に利用することで，アジャイルな ASIP 開発を実現する．ユーザは，プログラムの初期実装から始まり，性能ボトルネックの特定，CFU による高速化，性能評価，そして最終的な FPGA 実装という一連のサイクルを繰り返すことで，最適な ASIP 構成を段階的に探索する．

開発の起点となるのは，ユーザが C/C++ で記述するアプリケーションプログラム (Program) である．実装されたプログラムは，RISC-V Compiler によってコンパイルされる．コンパイルにより生成される命令およびデータを格納するメモリイメージ (Memory instr/data) は，テキスト形式で SoC ブロックに受け渡される．その後ユーザは，ホストシミュレーションを実行することで，CFU を導入する前のベースライン性能を測定する．ベースライン性能を測定した後，ユーザはフレームワークが提供するプロファイリング機能を用いて，実行サイクル数を計測し，性能ボトルネックとなるコード区間を特定する．

性能ボトルネックが特定された後，ユーザは CFU を設計してこれを高速化する．高位合成による CFU 設計では，CFU の機能を図 2 の関数プロトタイプに従って記述する．この関数において，funct3_i および funct7_i は，RISC-V カスタム命令の機能指定子である．単一の CFU 内で複数の異なる演算を実装し，命令ごとに処理を切り替えることが可能となる．src1_i および src2_i は，RISC-V プロセッサのレジスタファイルから供給される 32 ビット入力オペランドである．カスタム命令のソースレジスタ rs1 および rs2 の値に対応する．rslt_o は，CFU の演算結果を格納するための 32 ビット出力ポインタである．ユーザは，特定されたボトルネックとなるコードの処理内容に基づいて，この cfu_hls 関数を実装する．実装された cfu_hls 関数は，図 1 の CFU design ブロック内の HLS design に相当する．HLS design は，Vitis HLS による高位合成によって RTL design に変換される．生成された Verilog コードは，図の SoC ブロック内の CFU Wrapper によってカプセル化され，RISC-V Processor との標準インターフェースが提供される．同時に，開発者は元の C/C++ プログラム内のボトルネックコードを，カスタム命令の呼び出しに置き換える．このとき，カスタム命令の呼び出しには，インラインアセンブリを利用する．この修正されたプログラムは，再び RISC-V Compiler によってコンパイルされ，新たなメモリイメージが生成される．

CFU の設計と統合が完了した後，ユーザは再びホストシミュレーションを実行し，CFU 導入による性能改善効果を定量的に確認する．改善が不十分な場合，ユーザは CFU 設計に戻り，cfu_hls 関数の実装を修正する．修正後，再び Vitis HLS による高位合成，CFU Wrapper への統合，ホストシミュレーションによる評価という一連の流れを繰り返す．この反復的な設計サイクルにより，ユーザは段階的に CFU の性能を向上させることができる．

十分な性能改善が確認された後，FPGA への実装に進む．SoC

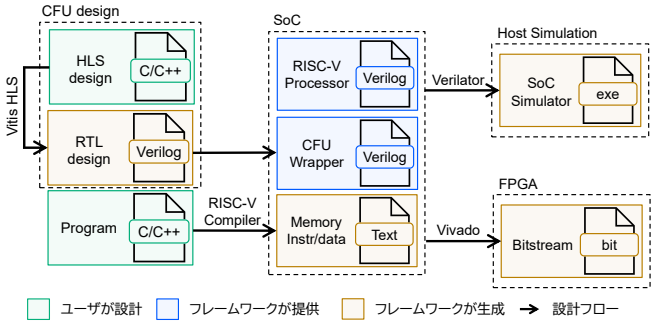

図 1: CFU Proving Ground の開発フローの概略 [9]

```
1  void cfu_hls (
2      char funct3_i,     // funct3 フィールド
3      char funct7_i,     // funct7 フィールド
4      int  src1_i,       // 入力オペランド1
5      int  src2_i,       // 入力オペランド2
6      int  *rslt_o       // 演算結果
7  );
```

図 2: CFU の関数プロトタイプ

全体が Vivado に入力され，ビットストリームファイルが生成される．このビットストリームをターゲット FPGA に配置することで，実機上でのアプリケーション実行および最終的な性能評価が可能となる．

2.2 CFU Proving Ground の問題点

CFU Proving Ground は，2.1 項の開発フローを用いることで，ASIP の効率的なプロトタイピングを可能にしている．しかし，カスタム命令を含むソフトウェアのアルゴリズム的な正当性を確認する機能検証の段階においては，依然として開発効率に大きな課題が残る．CFU Proving Ground は，ホストシミュレーション環境として Verilator を提供している．Verilator は，Verilog HDL で記述された回路デザインを高速な C++ クラスへと変換するサイクル精度シミュレータである．サイクル精度シミュレータは，ハードウェアの動作を正確にトレースするため，ハードウェアの規模に応じて，その実行時間は増加していく．CFU Proving Ground においては，ユーザが設計するハードウェア要素は主に CFU に集約される．そのため，CFU の規模や複雑さによっては，機能検証に数十分から数時間を要することとなり，デバッグの頻度を下げ，開発効率を著しく低下させる要因となる．加えて，サイクル精度での検証中に不具合が生じた際，その原因がソフトウェアの論理的なバグなのか，あるいは高位合成の最適化不足に起因するストールや処理落ちといったハードウェア側の不備なのかを切り分けることが極めて困難である．さらに，浮動小数点数を扱うプログラムを高位合成した場合など，外部の IP コアを含む回路が合成された場合，そもそも Verilator でのシミュレーションが行えないことも課題である．

こうした課題を解決するためには，CFU Proving Ground へ新たに命令レベルのシミュレータを統合する手法が考えられる．一般に ASIP 開発では，設計初期の機能検証には高速な命令レベルでのシミュレーションを用い，その後の詳細な性能評価においてサイクル精度のシミュレーションを行うという段階的なアプローチが取られる．しかし，新たなツールの追加は，外部ライブラリへの依存増加やフレームワークの複雑化を招き，CFU Proving Ground の特徴である軽量かつシンプルという利点を損なうおそれがある．

3. 提 案 手 法

本研究では，Verilator のシミュレーション環境に，高位合成前の CFU の C++ モデルを直接組み込む方法を提案する．Verilator の構造上，RTL モデルから変換された C++ クラスとそのテストベンチに対して，標準ライブラリや自作の C++ コードをそのまま組み込むことが可能である．これを利用して，CFU 以外の部分ではサイクル精度のシミュレーションを行いつつ，CFU でのみ命令レベルのシミュレーションを行う．つまり，CFU の起動をテストベンチで監視し，CFU が起動したら対応する C++ 関数を呼び出し，その結果を信号値として書き込む．これにより，CFU の処理を 1 クロックサイクルで完了させることで，シミュレーションにかかる時間の短縮をねらう．

本手法のポイントは，Verilator で RTL 記述から C++ に変換したコードを，元々 C++ で記述されている CFU のコードと組み合わせることにある．CFU Proving Ground 以外にも，Verilator を ASIP 開発の高速化に活用する先行研究は存在する [10]．しかし，いずれの研究も Verilator は RTL モデルのシミュレーション高速化のためにのみ使用している．Verilator の特徴をより積極的に活用しようとしている点に，本手法の新規性がある．なお，本手法で行う機能検証は，既存のサイクル精度の検証を置き換えるものではない．本手法は，サイクル精度の検証の手前に，追加の検証段階を提供するものであることに，注意されたい．

また本手法は，不具合が発生した際の問題切り分けにおいて有力な手段となり得ると考える．本手法において C++ 関数を用いた場合にソフトウェアが期待通り動作すれば，その時点でアルゴリズムのロジック自体に誤りがないことが証明される．したがって，後にこの C++ 関数を高位合成によって生成された RTL モデルに置き換えた際に不具合が発生するならば，その原因は高位合成の最適化設定に伴うパイプライン制御の不備といったハードウェア実装特有の問題であると明確に断定できる．このように，本手法は CFU Proving Ground の設計思想を損なうことなく，迅速かつ正確なデバッグプロセスを支援するものである．

図 3 に，CFU Proving Ground における SoC の最上位モジュールである top モジュールに対し，本手法による拡張を施したものの抜粋を示す．本来はプロセッサと CFU のラッパーモジュール間で完結している内部信号をシミュレーション環境の外部へ引き出す．これには，カスタム命令の実行開始を知らせるイネーブル信号 (cfu_en)，演算の対象となる 2 つの入力データ (src1, src2)，命令を識別するファンクションコード (funct3，funct7)，および演算結果を戻すためのデータライン (result) とプロセッサを一時停止させるストール信号 (stall) が含まれる．これにより，プロセッサの実行ステージにおける CFU の挙動を，テスト

ベンチから観測および制御することができる．

シミュレーションの実行主体となる C++ テストベンチの抜粋を図 4 に示す．本手法のために追加されたのは，このうち 10 から 18 行目である．毎サイクルの立ち上がりにおいて，拡張されたポートを通じて，CFU のイネーブル信号の状態を監視し，命令の発行を検知した瞬間にプロセッサ側のレジスタから出力された演算用データを読み出す．取得されたデータは，高位合成の入力として用いられる C++ 関数へと即座に引数として渡される．この関数内での演算は，シミュレータを実行している計算機上のソフトウェア処理として実行されるため，実機ハードウェアにおいて多サイクルを要する複雑なアルゴリズムであっても，シミュレーション上では論理的な 1 サイクルで結果が算出される．算出された演算結果は，直ちに SoC 側の入力ポートを介してプロセッサの入力ラインへと書き戻される．この際，テストベンチ側から SoC 側のストール信号を適切に操作し，データの注入が完了した直後にストールを解除して次の命令へ進めることで，プロセッサ側からはあたかもカスタム命令が 1 サイクルで正しい結果を返したかの用に振る舞わせる．この一連の操作を Verilator の eval() 関数による 1 回の更新ステップの中で完結させることにより，外部の命令レベルのシミュレータを統合することなく，既存の軽量かつシンプルな環境を維持したまま高速な機能検証環境が実現される．

4. ケーススタディ

4.1 適応ラインエンハンサ

提案した機能検証手法の有用性を検討するため，適応信号処理の一種である適応ラインエンハンサ (ALE) をケーススタディとして ASIP 開発を行った．ALE は，広帯域信号に埋もれた狭帯域信号を強調する適応フィルタである．ALE は，時刻 n における入力信号ベクトル $\boldsymbol{x}(n)$ を D サンプルだけ遅延させた参照信号ベクトル

$$\boldsymbol{x}(n-D) = [x(n-D), x(n-D-1), \ldots, x(n-D-M+1)]^T$$

を生成する．これを適応フィルタに適用することで，信号の予測値 $y(n)$ を算出する．フィルタの出力 $y(n)$ は，フィルタのタップ数を M としたとき，フィルタ係数ベクトル

$$\boldsymbol{h}(n) = [h_0(n), h_1(n), \ldots, h_{M-1}(n)]^T$$

と参照信号ベクトル $\boldsymbol{x}(n-D)$ の内積として，

$$y(n) = \boldsymbol{h}(n)^T \boldsymbol{x}(n)$$

のように定義される．この予測値 $y(n)$ と現在の入力信号 $x(n)$ との差分が誤差信号

$$e(n) = x(n) - y(n)$$

となり，これが最小化されるようにフィルタ係数が逐次更新される．

本研究では，この係数更新するための適応アルゴリズムとして，演算負荷の異なる以下の 3 種類を，それぞれフィルタの出

```
1  module top(
2      input wire clk_i,
3      input wire rst_i,
4
5      output wire cfu_en,
6      output wire [2:0] funct3,
7      output wire [6:0] funct7,
8      output wire [31:0] src1,
9      output wire [31:0] src2,
10     input  wire [31:0] rslt,
11     input  wire stall
12   );
13
14     wire clk,rst_n;
15     assign clk = clk_i;
16     assign rst_n = rst_i;
17
18     assign cfu_en = soc.cpu.cfu.en_i;
19     assign funct3 = soc.cpu.cfu.funct3_i;
20     assign funct7 = soc.cpu.cfu.funct7_i;
21     assign src1 = soc.cpu.cfu.src1_i;
22     assign src2 = soc.cpu.cfu.src2_i;
23     assign soc.cpu.cfu.stall_o = stall;
24     assign soc.cpu.cfu.rslt = rslt;
```

図 3: top モジュール (抜粋)

```
1   while (!contextp->gotFinish()) {
2     // clk toggle
3     top->clk_i = !top->clk_i;
4     // reset
5     if (!top->clk_i) {
6       top->rst_i = !(contextp->time() > 0
7                 && contextp->time() < 50);
8     }
9     // cfu
10    if (top->clk_i) {
11      if (top->cfu_en) {
12        top->stall = 1;
13        cfu_hls(top->funct3, top->funct7,
14                top->src1, top->src2, &rslt);
15        top->rslt = rslt;
16        top->stall = 0;
17      }
18    }
19    top->eval();
20    contextp->timeInc(5);
21  }
```

図 4: テストベンチ (抜粋)

力計算とともにカスタム命令として CFU に実装した．CFU の計算資源の削減とハードウェア実装の容易性を考慮し，演算には固定小数点数を用いた．数値表現形式は，演算精度を考慮し，整数部 16 ビット，小数部 32 ビットの計 48 ビット（Q16.32 形式）を採用した．

まず，最小二乗平均法 (LMS) である．最も基本的な適応アルゴリズムであり，

$$\boldsymbol{h}(n+1) = \boldsymbol{h}(n) + \mu e(n)\boldsymbol{x}(n-D)$$

に基づき係数を更新する．このとき，μ は収束速度を制御するステップサイズである．

次に，正規化最小二乗平均法 (NLMS) である．入力信号の電力でステップサイズを正規化することで，収束特性を安定させた手法である．NLMS では，

$$\boldsymbol{h}(n+1) = \boldsymbol{h}(n) + \frac{\mu}{\|\boldsymbol{x}(n-D)\|^2} e(n)$$

に基づき係数を更新する．

最後に，アフィン射影法 (APA) である．過去 p 個の参照信号ベクトルに対する拘束条件を課すことで，NLMS よりも高速な収束を実現する手法である．入力行列 $\boldsymbol{X}(n)$ と誤差信号ベクトル $\boldsymbol{e}(n)$ をそれぞれ，

$$\boldsymbol{X}(n) = [\boldsymbol{x}(n-D), \boldsymbol{x}(n-D-1), \ldots, \boldsymbol{x}(n-D-p+1)]^T$$

$$\boldsymbol{e}(n) = [e(n), e(n-1), \ldots, e(n-p+1)]^T$$

と定義する．これらを用いて，

$$\boldsymbol{h}(n+1) = \boldsymbol{h}(n) + \mu \boldsymbol{X}(n)(\boldsymbol{X}(n)^T \boldsymbol{X}(n))^{-1} e(n)$$

に基づき係数を更新する．本研究では，計算量の兼ね合いから $p = 2$ として実装した．

4.2 音声入力環境の構築

本研究では，Digilent 社製の PmodI2S2 モジュールを介したリアルタイムの音声信号処理を想定し，SoC 内に I2S 通信インターフェースを実装した．I2S 通信回路は，オーディオコーデックのサンプリング周波数に同期した専用のクロックドメインで動作する．一方，プロセッサはそれとは異なるシステムクロックドメインで動作している．これら異なる動作クロック間での安定したデータ転送を実現するため，SoC 内部には非同期 FIFO を導入した．モジュールの ADC 側ではアナログ音声信号を 24 ビットのディジタル音声信号に変換して SoC へ供給し，プロセッサが ALE の演算を実行した後，モジュールの DAC 側が再びアナログ信号へと復元して出力する．

5. 評　　価

5.1 評価環境・条件

カスタム命令の演算を C++ 関数として実行することで，従来の RTL モデルを利用したシミュレーションと比較し，シミュレーション時間がどの程度短縮されるかを定量的に評価する．また，機能検証後の出力信号を周波数領域で解析し，ALE としての妥当性を確認する．評価に用いる入力信号は，サンプリングレート 44.1 kHz でサンプリングされた 5 秒間の音声データ (総サンプル数: 220,500 点) である．この信号は，1 kHz の正弦波に対してホワイトノイズを重畳したものであり，信号対雑音比 (SNR) は 10 dB に設定した．また，ALE のパラメータとして，フィルタのタップ数 (M) を 512，遅延量 D を 44 サンプル，ステップサイズ μ を 0.001 とした．

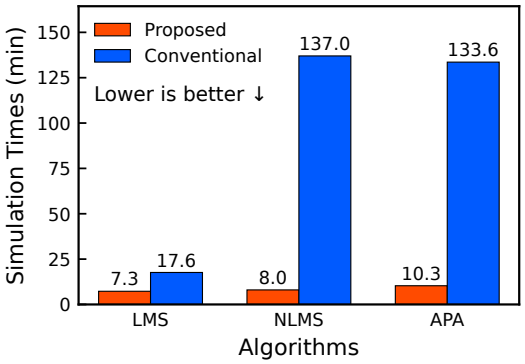

図 5: シミュレーション時間の比較

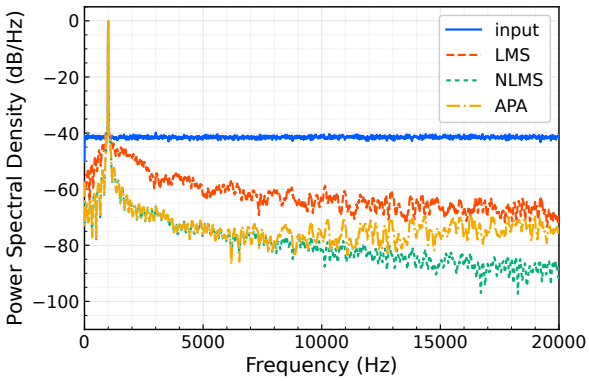

図 6: 各適応アルゴリズムにおけるパワースペクトル密度

シミュレーションの実行においては，テキストファイルを通じて入出力信号の読み出し，書き込みを行う．そのため，ディスク I/O がシミュレーション実行時間のボトルネックとなりうる．そこで，入力においては，あらかじめテキストファイルから全サンプルのデータを読み込み，ホスト計算機上のメモリに展開した状態でシミュレーションを開始した．また，出力においても，シミュレーション実行中に逐次ファイル書き出しを行うのではなく，すべてのサンプルに対する処理が完了した段階で，一括してファイルへ書き出す構成とした．

評価実験に用いた計算機は，CPU に Intel N100 の 16 GB DDR4 メモリを搭載したものを使用する．OS は Ubuntu 24.04 LTS である．また，ツール環境として，Verilator のバージョンは v5.040，高位合成ツールは Vitis HLS 2024.2 を使用した．

5.2 評価結果と考察

従来手法と提案手法におけるシミュレーション時間の比較結果を図 5 に示す．図 5 に示す通り，すべての適応アルゴリズムにおいて，提案手法は従来手法を大幅に短い実行時間を記録していることがわかる．LMS では 17.6 分から 7.3 分へと短縮され，演算負荷の高い NLMS では 137.0 分から 8.0 分へ，APA では 133.6 分から 10.3 分へと，約 13 倍から約 17 倍もの高速化を

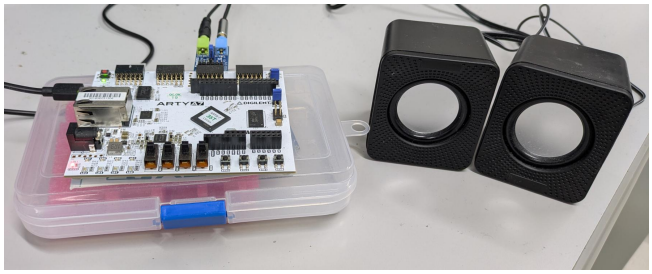

図 7: 実機での動作環境

達成した．また，従来手法にて，LMS と比較して NLMS，APA の実行時間は極端に長いことがわかる．これは，アルゴリズムに含まれる除算処理の影響であると考えられる．LMS は積和演算を主体とするに対して，NLMS や APA は除算を含んでいる．これを RTL シミュレーションする際には，複雑な除算器の動作を逐次計算する必要があるため，多大な時間を要する．一方，提案手法ではこれらの演算を C++ モデルとして実行するため，シミュレーションのオーバーヘッドを削減できているといえる．以上の結果から，除算器のような回路規模が大きくなる演算器を含む場合や，繰り返し処理を伴う複雑なアルゴリズムの実装において，提案手法の有用性が示唆されたといえる．

次に，ALE の妥当性をパワースペクトル密度 (PSD) に基づき評価した．図 6 から，約 −40dB/Hz のフロアノイズを持つ入力信号に対し，各アルゴリズムを適用した ALE は明確なノイズ抑制効果を示している．LMS は，約 −60dB/Hz 付近までの抑制に留まっているが，NLMS や APA は −80dB/Hz から −100dB/Hz 付近までノイズを低減させており，十分なノイズ抑制効果があることがわかる．以上の結果から，ALE として十分な性能であると言える．

5.3 実機での動作確認

作成された CFU を含む SoC を CFU Proving Ground の Vivado フローで論理合成し，FPGA ボードでの動作確認も行った．図 7 にその環境を示す．評価ボードには Digilent Arty A7-35T，その Pmod 端子に音声モジュール PmodI2S2 を接続した．音声信号の入出力系として，PC のイヤホンジャックから出力された音声を音声モジュールの入力端子へ接続し，FPGA 内部で信号処理を行った後の出力を，音声モジュールの出力端子に接続された外部スピーカへ接続する構成をとった．3 種類の各適応適応アルゴリズムにおいて，CFU による高速化をした場合 (ASIP) とそうでない場合 (汎用プロセッサ) での動作を確認した．

その結果，汎用プロセッサでは，最も演算負荷の低い LMS ですら正常に動作しなかった．スピーカからは低周波のうねりのような音が出力され，目的の信号とは明らかに異なる音であった．また，時折サンプル欠落による音の途切れが発生した．これは，サンプリング周期に対して処理が間に合っていないためであると考えられる．一方，ASIP では，いずれの適応アルゴリズムにおいても問題なく動作した．スピーカからは 1 kHz 正弦波が明瞭に出力され，ノイズが抑制されたクリアな音声を確認できた．これは，CFU への演算オフロードにより，実行サイクル数が削減されたためであると考えられる．

6. ま と め

本研究では，ASIP 開発における機能検証の効率化を目的として，CFU Proving Ground の高位合成フローをベースとした高速な機能検証手法を提案した．提案手法を用いて ALE の実装および評価を行った結果，従来の RTL シミュレーションと比較してシミュレーション時間を劇的に短縮できることを確認した．特に，除算器のような回路規模の大きい演算器を必要とする NLMS や APA アルゴリズムにおいて，最大で約 17 倍の高速化を達成した．また，PSD による評価では，ノイズレベルを −80 dB 以下まで抑制できており，ALE として十分な性能が得られていることを確認した．これにより，除算や繰り返し処理を含む複雑なカスタム命令を含む ASIP の開発において，本手法が有用な機能検証法となることが示唆された．

今後の課題として，ASIP 開発における本手法の適用指針を確立することが挙げられる．演算特性の異なる多様なアプリケーションでの評価を通じて，本手法が有効となる条件を明確化する必要がある．

文　献

[1] L. Kong, J. Tan, J. Huang, G. Chen, S. Wang, X. Jin, P. Zeng, M. Khan and S. K. Das, “Edge-computing-driven Internet of Things: A Survey,” ACM Computing Survey, vol.55, no.8, 2022.

[2] X. Wang, Z. Tang, J. Guo, T. Meng, C. Wang, T. Wang and W. Jia, “Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models,” ACM Computing Surveys, vol.57, no.9, 2025.

[3] H. W. Oh and S. E. Lee, “The Design of Optimized RISC Processor for Edge Artificial Intelligence Based on Custom Instruction Set Extension,” IEEE Access, vol.11, pp.49409–49421, 2023

[4] E. Hussein, B. Waschneck, C. Mayr, “Automating application-driven customization of ASIPs: A survey”, Journal of Systems Architecture, vol.148, p.103080, 2024.

[5] M. Damian, J. Oppermann, C. Spang, and A. Koch, “SCAIE-V: an open-source SCAlable interface for ISA extensions for RISC-V processors,“ In Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC ’22), pp.169–174, 2022.

[6] Synopsys, Inc., “ASIP Designer,” https://www.synopsys.com/designware-ip/processor-solutions/asip-designer.html, 2024. Accessed: january 26, 2026.

[7] K. Hepola, J. Multanen and P. Jääskeläinen, “OpenASIP 2.0: Co-Design Toolset for RISC-V Application-Specific Instruction-Set Processors,” 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp.161–165, 2022.

[8] S. Prakash, T. Callahan, J. Bushagour, C. Banbury, A. V. Green, P. Warden, T. Ansell, and V. J. Reddi, “CFU Playground: Full-Stack Open-source Framework for Tiny Machine Learning (tinyML) Acceleration on FPGAs,” 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp.157–167, 2023.

[9] A. Fujino and K. Kise, “CFU Proving Ground: a Hardware/Software Co-Design Framework for Leveraging a Custom Function Unit and RISC-V Custom Instructions,” 2025 IEEE 18th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp.199–206, 2025.

[10] Y. Chi, X. Lin, and X. Zheng, “Design of High-performance SoC Simulation Model Based on Verilator,” 5th International Conference on Algorithms, Computing, and Atrificial Intelligence, no. 92, pp. 1–6, 2022.