GAN を用いた新たなアバター生成

藤間裕大　小野佳祐　土居遼太郎　美谷佳寛
千葉大学院　融合理工学府　基幹工学専攻　修士 1 年

1　ハードウェア実装時の使用機器

本システムのハードウェア実装には、AMD Zynq™ UltraScale+™ MPSoC ZCU104 Evaluation Kitを用いた[1]。また、AMDの提供するPython productivity for Zynq(PYNQ)[2]と呼ばれるオープンソースプロジェクトを活用し、ブラウザ上からJupyter Notebookを通じてFPGAボードを動作させるシステムを構築した。本節では使用機器についての解説と仕様説明を行う。

1.1　Zynq UltraScale MPSoCの概要

Zynq UltraScale MPSoCとは、ARMプロセッサ（PS）とFPGA（PL）を1チップに統合したSoCである。PS部、PL部それぞれの仕様は表1, 2のようになっている。本ボードの最大の特徴は、ソフトウェアとハードウェアを同一チップ上で高速かつ低遅延に動作させられる点である。本回路ではPL部とPS部間の通信にAXIインターフェースの中でも扱いやすいAXI4-Liteを用いており、APUとFPGA間の通信が可能となっている。

表1　PL部の仕様

System Logic Cells	504,000
CLB Flip-Flops	460,800
CLB LUTs	230,400
Block RAM Blocks	312
Block RAM	11 [Mb]
DSP Slices	1,728
Max. HP I/O	416
Max. HD I/O	48

1.2　PYNQの概要

Python productivity for Zynq(PYNQ)は、Linux上で動作するPython APIを用いてZynq SoCのPL部を制御・利用するためのフレームワークである。本フレームワークを用いることにより、以下のような利点がある。

表2　PS部の仕様

APU	Dual-core Arm Cortex-A53
Architecture	Armv8-A
OS	PYNQ Linux
Framework	PYNQ
Language	Python

- Libraryが豊富なPythonを利用して、PL部の制御を行える
- Linux環境下で動作させられるため、柔軟性のある開発環境が実現可能
- Webブラウザ上でJupyter Notebookを利用した直感的な操作が可能

このように、PYNQを用いることで迅速な開発やPythonを利用できる幅広いユーザに対して製品を提供することが可能となる。また、評価ボードをインターネットに接続することで、別PCからJupyter Notebookを介してアクセスすることが可能であり、評価ボードの遠隔操作やファイル転送を容易に行える。そのため、本フレームワークはIoTシステムとしての実用化にも適している。このことから本システムにおいても、システムの実用化に向けてPYNQを利用した開発を行う。

2　システム構成と動作説明

本章では設計したシステムのアーキテクチャからPS部、PL部それぞれの構成、およびその動作について説明する。

2.1　システムのアーキテクチャ

本システムのアーキテクチャは図2のようになっている。Block RAM等のリソース量を考慮し、エミュレータで学習したパラメータを使用して生成部のみの実装を行った。本システムは大きく分けてPS部とPL部に分かれている。PS部は、保存された入力データおよ

び重みをPL部に送る役割を担う。またPL部の動作を制御する制御信号を送ることで動作モードを遷移させ、PL部の動作を処理の進行に合わせて適切に変化させる役割も担っている。一方、PL部はPSとPL間の通信を担うAXI-LITE、PS側から送られたデータをGeneratorに適切な形、タイミングで転送するモジュール、そしてGeneratorの大きく分けて３つの回路で構成されている。これらのシステム全体の動作概要は以下の通りである。

1. AXI-LITE形式でPS側からPL側に図中Input Vecotrで示される100次元入力を送信する
2. AXI-LITE形式でGeneratorの入力用RAMにデータ転送する開始信号（Control signal）を送る
3. AXI-LITE形式でPS側からPL側に推論に必要な重みのうち、１層分のさらに１出力要素を得るために必要な重み（Weight of 1-4 layersの一部）を送信する
4. AXI-LITE形式でGeneratorの重み用RAMにデータ転送する開始信号を送る、PSは演算が終了するまで待機する
5. １出力を得るために必要な入力xおよび重みwを取得した後、Generatorは計算を開始する
6. RAMに保存された重みを使い切ると、Generatorが1出力の演算終了信号を送信し、図中end_signalとしてAXI-LITE形式でPS側が受け取る
7. 3 - 6の手順を繰り返し、１層分の出力を得る。この時、１層分の演算終了信号がPS側に送信される
8. さらに3 - 7の手順を繰り返すことで4層分の演算を行う。この時、演算終了信号が送信される。
9. ４層分の演算終了後、AXI-LITE形式でPL側が出力したアバター画像をPS側に送信する

特に手順3-6については、図1で補足説明を行う。エミュレータで実装したCNNモデルのうち、特に１層目、２層目は図1のようになっている。１層目では入力された100次元の潜在ベクトルに対して4×4のカーネルを512×100要素掛け合わせることで出力を獲得し、これにReLU関数を適用したものを2層目の入力として用いる。このような試行を4層分繰り返すことでアバター画像を出力することができる。本回路では特に、それぞれの層の出力の１要素を求めるために必要な重みである図1中の1paked weight（1×100要素）のみをPS部からPL部に送信する。そして、この重みと入力の掛け算を行う試行をそれぞれの要素の出力要素分行うことで１層分の計算を終了する回路とした。本回路構成を採用することにより、FPGAボードのPL側に保存される重みのパラメータ数は常に図1で示される1paked weight部分のみとなるため、Block RAMの使用量を大幅に減少させることができるメリットがある。特に本構成が拡張性の高いシステムに繋がる厳密な議論については、10章の「FPGA実装の中で工夫した点」で示す。

2.2 PL 側の回路設計

AXI-LITE

本回路では、AXI-4という規格の中でもAXI-LITEを用いた通信方式を採用した。AXI規格とは、ARM社が使用を策定したバスプロトコルであり、開発者側は共通のインターフェースを用いてPSとPL間の通信を行う。特にAXI-LITE Moduleの回路構成を図3に示す。図3からわかるようにAXI-LITEはアドレス指定により異なる制御を行うこと

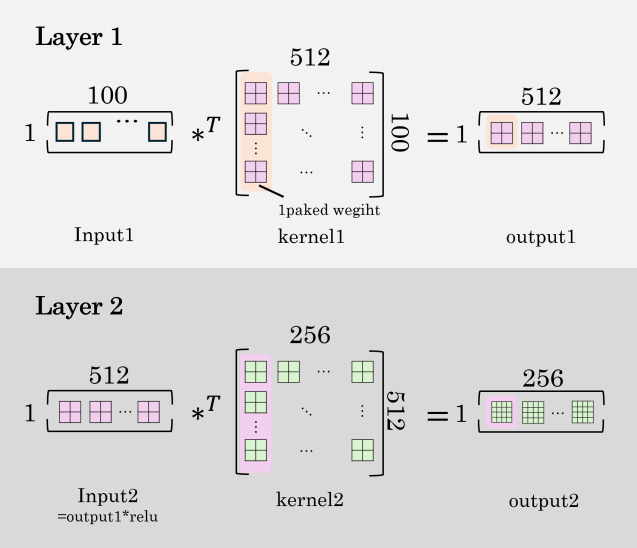

図 1　計算手順の概要

ができる利点がある。本回路では表3のようにアドレスマップを作成した。特にslv_reg0, slv_reg1を表4のように設定することで、PSからAXI-LITEを通じて細かくPLの制御を行えるような設計とした。この設計により、前述した複雑な計算手順をPythonを利用して極めて正確かつ簡単に行うことができるような設計となっている。具体的に 1 つの層を例にとって、実際の動作手順を説明する。

1. 100次元入力を送信するため、slv_reg0に00001をセットする（リセット信号はアクティブロー）
2. GeneratorのRAMに入力信号を読み込ませるため、計算開始信号を立てる（slv_reg0に00011）
3. 計算に必要な重みを送信するため、slv_reg0に00101をセットする。
4. GeneratorのRAMに重みを読み込ませるため、計算開始信号を立てる（slv_reg0に00111）
5. 重みを使い切るまでPS側は動作を止める（slv_reg1が100となるまで待機）
6. 1-5の操作を512回繰り返したとき、slv_reg1が110となり 2 層目の処理に移行する。

このように、PL側の動作をPS側から可視化できるようにすることでUIを用いた計算の進捗状況表示や計算手順の柔軟性が生まれるように設計を行った。本実装は、アドレス指定により制御に幅を持たせることができるAXI-LITEならではの制御法といえる。

表 3　AXI-LITE のアドレスマップ

Address (32bit)	data
0	slv_reg0
1	slv_reg1
2	Input or Weight DATA
3	Output DATA

表 4　AXI-LITE の制御信号

slv_reg0	値	slv_reg1	値
リセット信号	0	全計算終了	0
計算開始信号	1	1 層計算終了	1
FIFOA,C 選択	2	1 出力計算終了	2
計算層選択	3-4		

FIFO-A, FIFO-B, FIFO-C

FIFOとは、AXI-LITEからPL側に書き込まれた値、またはPS側に書き込む値を一時的に格納し、格納した順に出力するモジュールである。本システムでは、FIFOを 3 つ搭載し、FIFO-Aには入力zである100次元の潜在ベクトルの転送、FIFO-Cには重み$Weight$の転送、FIFO-BにはGeneratorからの出力の転送を割り当てている。FIFO-A, FIFO-Cにはどちらともに PS側がアドレス3を指定して書き込んだ値が入力されるが、slv_reg0(2)の値がLowの時はFIFO-Aのみに書き込みが有効、Highの時はFIFO-Cのみに書き込みが有効と

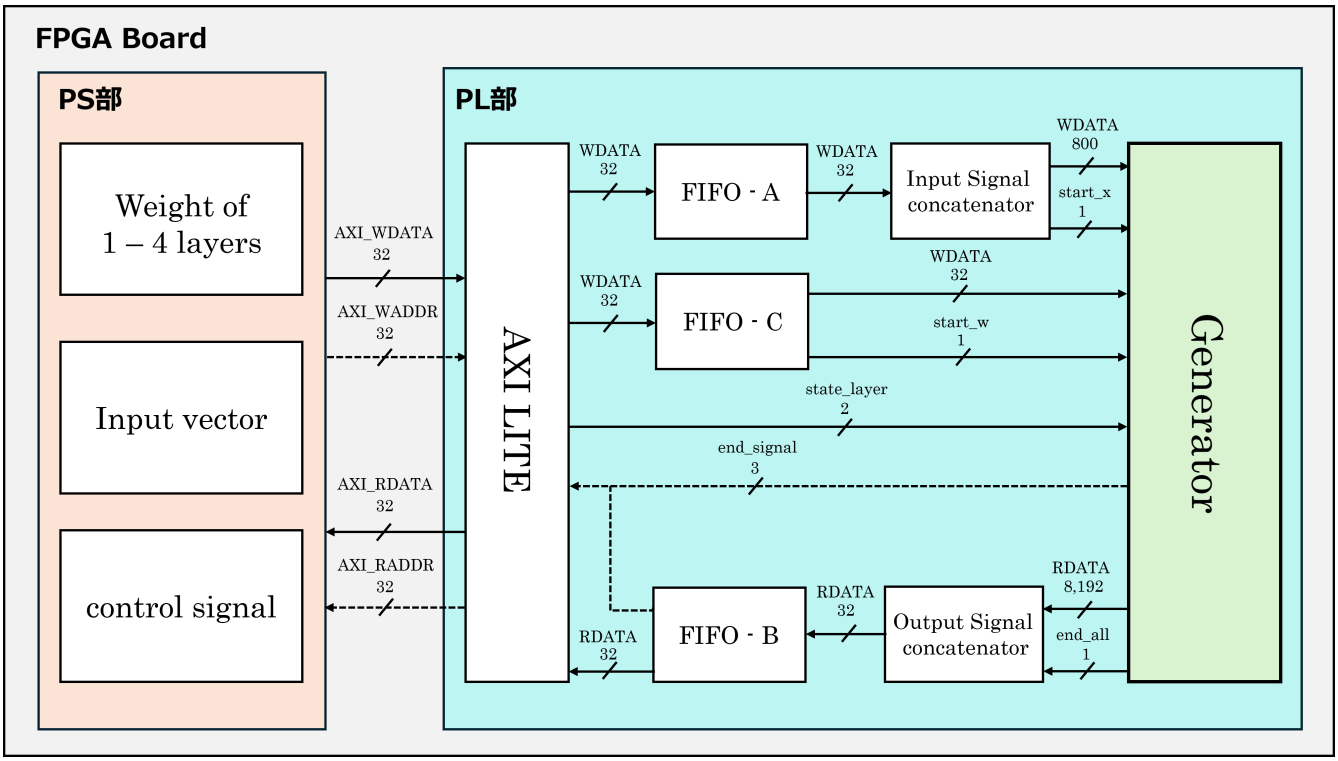

図 2　システムのアーキテクチャ

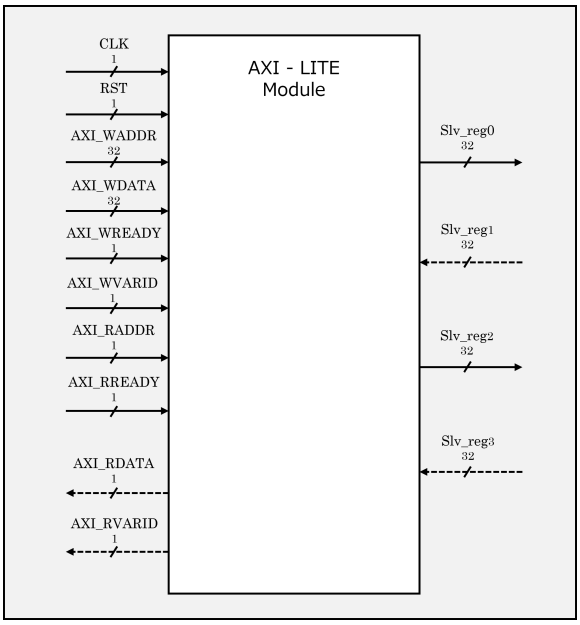

図 3　AXI-LITE module の構成

なるように設計することでFIFOの分岐動作を実現した。

Input, output Signal concatenator

Input signal concatenator, Output signal concatenatorはそれぞれ図4のようにあらわされる計算回路である。Input Signal cocatenatorでは、Generatorに対して必要な入力データを1CLKで渡すことを可能とするためにFIFO-Aから送られる32ビット幅の信号を繋げて１つの信号として送信する。一方、Output signal concatenatorではGeneratorからPS側に対して送られる$32 \times 32 \times 8$bitの信号をそれぞれ32ビットの信号に分けることで、本回路におけるAXI-LITE通信方式にあった形でPS側にデータ送信できるように加工する回路である。

Generator

エミュレータで学習を行ったパラメータと計算精度をもとにして、100次元入力から32×32のアバター画像を生成する計算部分である。次節にて詳しい説明を述べる。

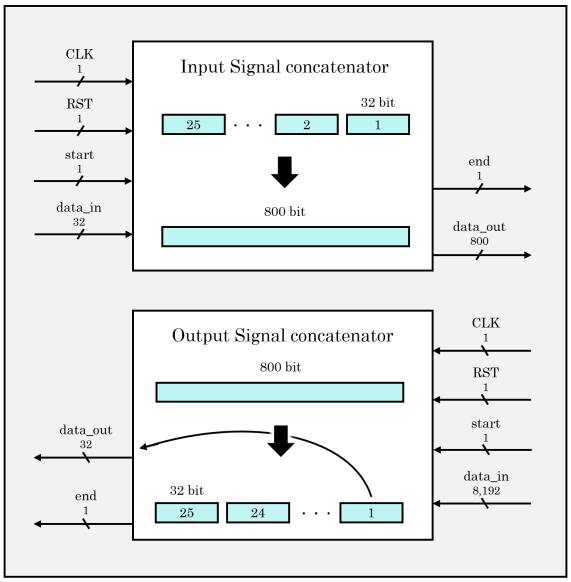

図 4　Input, output Signal concatenator の構成

3　計算回路の設計

本章では、提案するGANアバター生成システムの核となる計算回路のFPGA実装について、その詳細な設計内容を記述する。

3.1　計算回路の全体アーキテクチャ

本システムにおけるGenerator回路は、画像生成のための大規模な転置畳み込み演算を効率的に実行するため、明確な役割分担を持つ複数のモジュールによる階層構造を採用している。また、リソース削減の観点から、重み格納メモリWeight_RAMの定期更新と内部メモリの循環を採用している。各メモリ帯域および演算コア間の詳細なデータフローを図5に示す。

3.1.1　主要モジュールの機能

図5に示した全体のデータフローを解説する前に、本項ではシステムを構成する主要な8つのハードウェアモジュールの役割について定義する。

1. **Input_Deserializer**: 外部から1クロックで一括して供給される100次元のランダムデータ（16bit × 100要素）を受け取るインターフェースである。本モジュールは、1600bitの並列データを16bit単位に分割し、それぞれ演算用の16bit幅のFeature_RAMに順次転送する。
2. **Feature_RAM / Weight_RAM**: 演算に必要なデータを保持するメモリである。Feature_RAMは入力特徴マップ（16bit）を保持し、Weight_RAMは学習済みの重みパラメータを保持する。特にWeight_RAMでは、データ転送効率向上の観点から、4 × 4のカーネル内の横一列4要素（8bit × 4）を1つのアドレスに集約して保持する仕様にした。そのため、8bitの重みデータを4つ結合した32bitパケットとして管理する。
3. **deconv_layer**: 本システムの演算の中核を担うモジュールである。第1層から第4層までの転置畳み込み演算を担当し、メモリからデータを読み出して積和演算を実行する。
4. **Accumulator_RAM**: 積和演算の途中経過を保持する累積加算用メモリである。演算過程におけるオーバーフローを防止するため、32bit幅を採用した。
5. **Saturation**: Accumulator_RAMから読み出された32bitの演算結果に対し、飽和演算を行いながら16bit幅へ丸め処理を行うモジュールである。
6. **ReLU**: 中間層（第1〜3層）において使用される活性化関数モジュールである。負の値を0にする非線形処理を行う。
7. **Tanh_LUT**: 最終層（第4層）において使用される活性化関数モジュールである。双曲線正接関数の値を格納したルックアップテーブルを参照することで、複雑

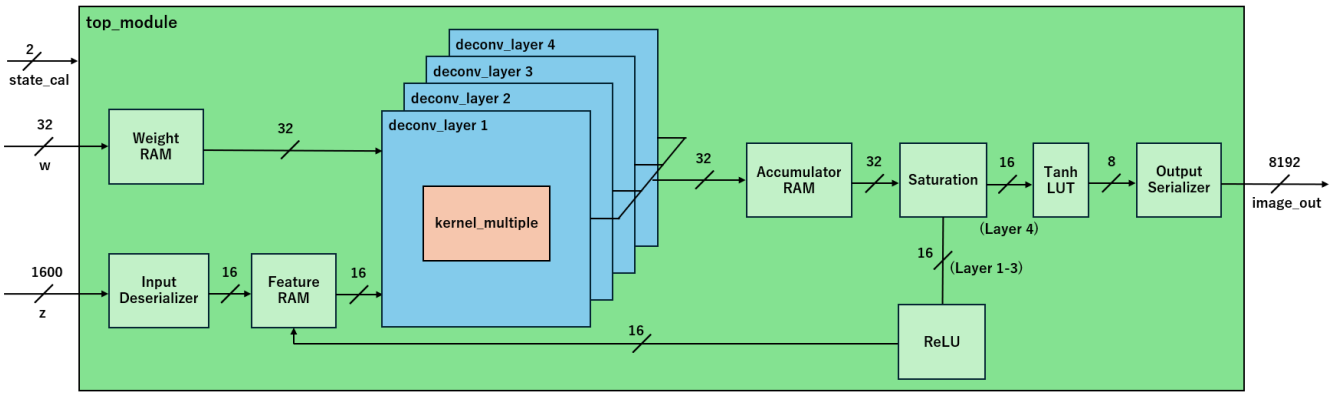

図 5　計算モジュールの階層構造と詳細データフロー図

な非線形計算を回避しつつ、データを画像の画素値に適した8bit範囲へ変換する。

8. **Output_Serializer**: 最終的な生成画像を外部へ出力するインターフェースである。並列データを1画素ずつ直列化し、image_out（8bit × 32 × 32画素 = 計8,192bit）として出力する。

3.1.2　全体のデータフロー

前項で定義したモジュール群は、最上位階層であるtop_moduleによって統合管理され、データは以下のフローに従って処理される。

まず、外部より供給されたランダム入力zは、Input_Deserializerによって個別に切り分けられ、Feature_RAMに格納される。その後、重み値wもWeight_RAMにロードされる。

演算フェーズでは、各層に対応するdeconv_layerが順次起動される。deconv_layerは、Feature_RAMおよびWeight_RAMから必要なデータを読み出し、内部演算コアkernel_multipleへ供給する。演算された積和結果は、Accumulator_RAMに対して順次加算され、畳み込み結果が形成される。

1層分の演算が完了すると、データはAccumulator_RAMから読み出され、Saturationで16bitに圧縮される。その後、中間層の場合は、データは通信部を介さず直接ReLUへ送られ、次層の入力特徴マップとして再びFeature_RAMへ書き戻される。一方、最終層の場合、データはTanh_LUTに入力され、画素値として整形された後にOutput_Serializerを介して画像データimage_outとして外部へ出力される。

このように、層によってデータの行き先を切り替える構成をとることで、限られた回路リソースを有効活用しながら、スムーズな処理を実現している。

3.1.3　全体制御ステートマシンの設計

本システムのtop_moduleは、入力から出力に至る一連の処理を効率的に制御するため、ステートマシーンによって管理されている。主要な状態遷移と制御信号の流れを図6に示す。

本ステートマシーンはアイドル状態（Idle）から始まり、Input_data_controlからの制御信号start_xの立ち上がりエッジを検出することで動作を開始する。動作は大きく分けて以下の手順で進行する。

1. **入力値の展開（Load_Input）**：start_xを受信すると、Input_Deserializerが起動する。ここでは一度に受け取った並列データを、クロックに同期して順番に切り分けてFeature_RAMへ書き込む。

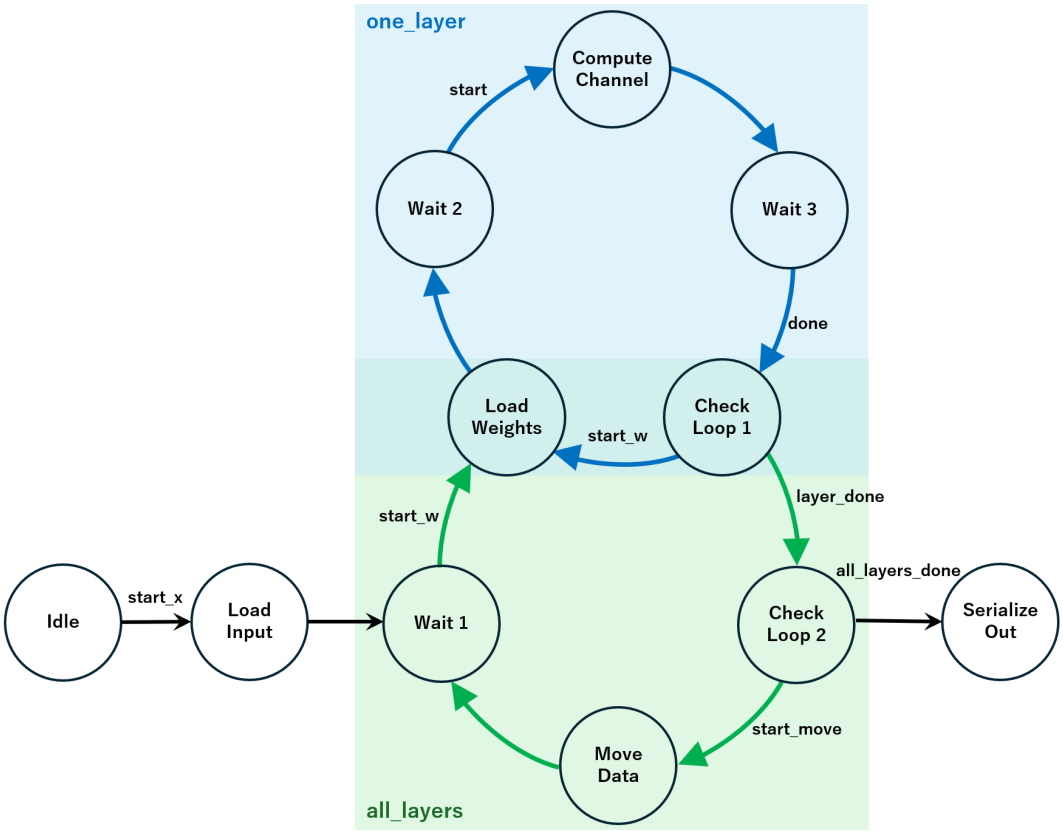

図 6　全体制御ステートマシンの簡略化状態遷移図

2. **入力値の格納待機（Wait 1）**：Load_Inputから来た場合は、全データの格納が完了するまで待機する。Move_Dataから来た場合は、Feature_RAMに対して全データの転送が完了するまで待機する。
3. **重み値のロード（Load_Weights）**：外部からのstart_w信号がHighになっている期間、ステートマシーンは毎クロック32bit幅の重み値をそのままWeight_RAMに格納する。本フェーズは1出力チャネルの演算が終わるたびに繰り返し実行される。
4. **重みロード待機（Wait 2）**：start_wがLowになり、データの転送期間が終了するまで待機する。
5. **チャネル演算実行（Compute_Channel）**：deconv_layerに対して開始信号startを発行し、1出力チャネル分の積和演算を実行する。
6. **演算待機（Wait 3）**：deconv_layerからの完了通知doneを受信するまで待機する。
7. **チャネルループ判定（Check_Loop 1）**：同一層内の演算継続可否を判定する。
   - **チャネル継続**: 未計算の出力チャネルが残っている場合、start_w信号に従い、次の重みをロードしてWeight_RAMを更新する。
   - **層完了**: 層内の全チャネル計算が完了している場合、完了信号layer_doneに従い、次の判定ステートへ移行する。
8. **レイヤーループ判定（Check_Loop 2）**：全層の進捗状況に応じた分岐を行う。
   - **データ更新（中間層）**：最終層でない場合、信号start_moveに従い、Move_Dataへ遷移する。

- **全完了（最終層）**：第4層の演算が完了している場合、完了信号all_layers_doneに従い、Serialize_Outへ遷移する。

9. **データ更新（Move_Data）**：Accumulator_RAMにある演算結果を1クロックごとに読み出し、SaturationおよびReLU活性化関数を通してFeature_RAMへ書き戻す。この時、外部は介さない。その後は、次層の重みをロードして演算を行う。
10. **最終出力（Serialize_Out）**：Accumulator_RAMから演算結果を読み出し、SaturationとTanh_LUT、Output_Serializerを通して、出力用レジスタimage_outに格納される。全画素の格納が完了した時点で、画像データ全体がまとめて外部へ出力される。

3.2 転置畳み込みユニット（deconv_layer）の実装

本システムにおける転置畳み込み演算の中核を担うのがdeconv_layerモジュールである。このモジュールは、上位システムtop_moduleと演算コアkernel_multipleの間に位置するメインコントローラとして機能する。本モジュールの概略図を図7に示す。

3.2.1 各サブモジュールの機能

本モジュールは、役割の異なる4つのサブモジュールによって構成されている。

1. **Controller**: ユニット全体の動作を統括する最上位コントローラ。初期化フェーズでは、次の演算結果が格納されるAccumulator_RAMの領域を順次走査し、クロック同期で0を書き込む。演算フェーズでは、loop_counterから供給される座標情報（ix, iy, kx, ky）を、読み出しアドレス（x_addr, w_addr）へと変換してFeature_RAMとWeight_RAMへ出力する。また、kernel_multiple演算結果を確定させるタイミングでAccumulator_RAMへの書き込み許可信号（acc_we）を発行する重要な役割を担う。1チャネルの演算が完了した際には、上位へ終了信号doneを発行する。
2. **loop_counter**: 1つの出力チャネルの演算に必要な4重ループ（入力画像の走査およびカーネルの展開）を管理する座標生成カウンタ。Controllerからのカウント許可信号cnt_enがHighの期間のみカウントを進める設計となっており、特定のタイミングで座標出力を保持することが可能。各ループの最大値は層のパラメータに応じて設定されており、全走査が完了すると完了信号loop_doneが自動で生成される。
3. **addr_generator**: loop_counterから供給される入力画像の座標（ix, iy）とカーネル座標（kx, ky）を基に、転置畳み込みのストライド（S）およびパディング（P）を考慮した出力座標を算出し、出力画像上の一次元アドレス（y_addr）に変換する。そして、そのアドレスをAccumulator_RAMへ出力する。
4. **Accumulator_RAM**: 積和演算の途中経過を保持する32bit精度の累積加算用メモリ。書き込み許可信号（acc_we）がHighの期間のみ書き込みが有効で、Lowの時にアドレスを送ると格納された蓄積値が読み出される。蓄積値に新しい乗算結果を足し合わせ、再び同じ場所へ上書きして保存することで、累積計算を実現する。
5. **kernel_multiple**: Feature_RAMから来る入力値x_dout、Weight_RAMから来る重み値w_dout、Accumulator_RAMから来る蓄積値y_rdataを基に、積和演算を行う演算コア。演算結果y_wdataは

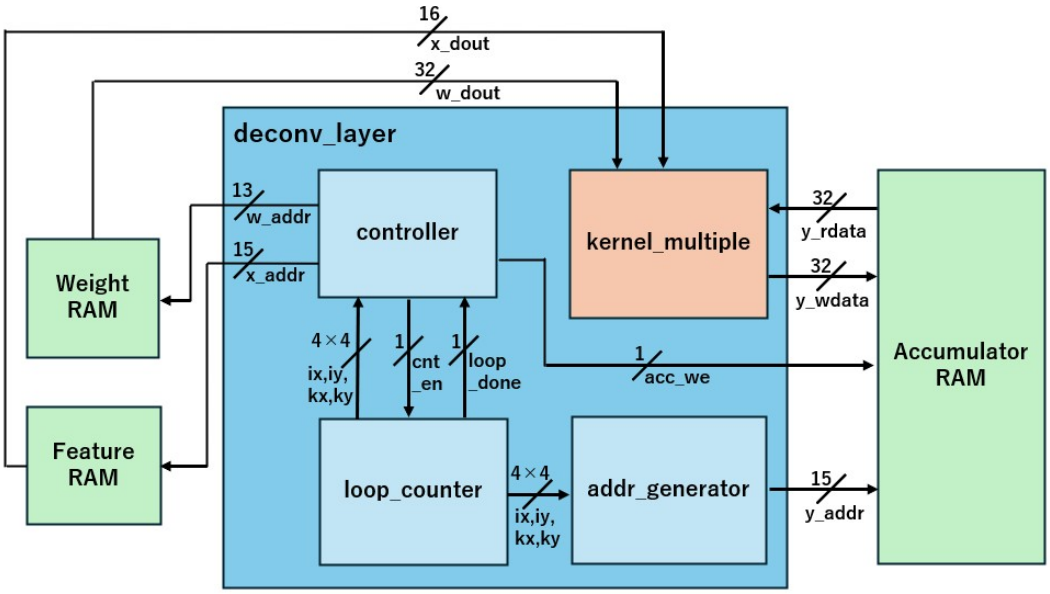

図 7　deconv_layer の内部構成

Accumulator_RAM に出力する。

3.2.2　deconv_layer 内におけるデータフロー

deconv_layer 内部では、Controller を中心に各モジュールが双方向に信号をやり取りすることで、一連の演算プロセスを進行させる。そのステートマシンを図8に示す。

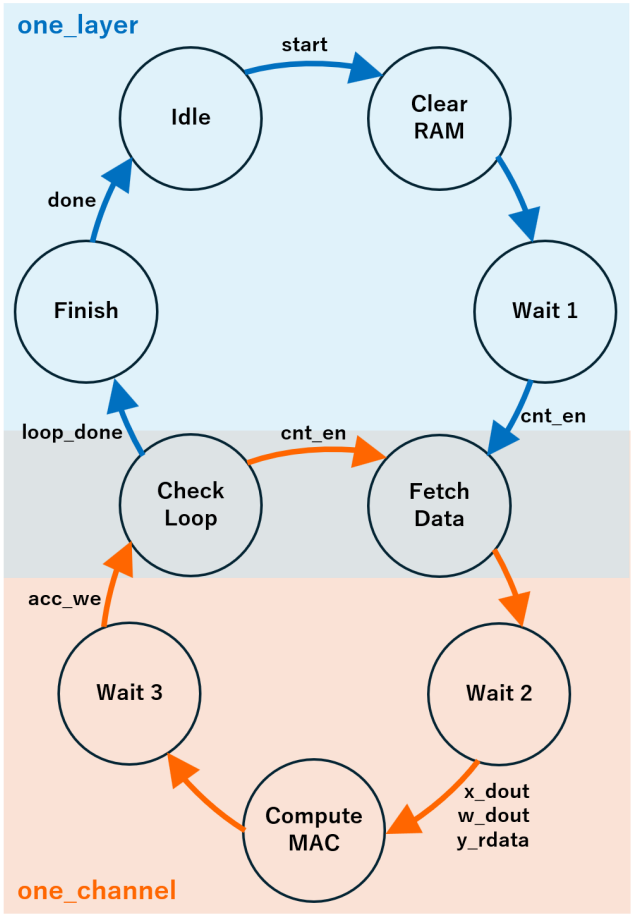

図 8　層内制御ステートマシンの状態遷移図

ステートマシンは以下の手順で動作する。

1. **待機（Idle）**：上位からの start 信号を待

機する。

2. **初期化（Clear_RAM）**：Controllerが主導してAccumulator_RAMの領域を初期化し、演算の準備を整える。
3. **初期化待機（Wait 1）**：メモリのリセットが完了するまで待機する。
4. **データ読み出し（Fetch_Data）**：Controllerがcnt_enを立ち上げ、loop_counterが座標信号を生成する。生成された座標信号は二手に分かれ、一方はControllerで読み出しアドレスに変換され、Feature_RAMとWeight_RAMに出力される。もう一方はaddr_generatorで演算結果の格納先アドレスに変化され、Accumulator_RAMに出力される。
5. **読み出し待機（Wait 2）**：kernel_multipleに対して、Feature_RAMとWeight_RAMからは入力値と重み値、Accumulator_RAMから蓄積値が読み出されるまで待機する。ここでcnt_enは立ち下げ、次のFetch_Dataまでloop_counterは停止する。
6. **積和演算実行（Compute_MAC）**：供給された入力値・重み値はkernel_multiple内で乗算され、蓄積値と加算される。このようにして1座標分の積和演算が実行される。
7. **演算完了待機（Wait 3）**：積和演算が終わると、加算結果y_wdataは、Controllerが制御する書き込み許可信号acc_weに同期して、再びAccumulator_RAMの同一アドレスへと書き戻される。
8. **ループ判定（Check_Loop）**：loop_counterから完了信号loop_doneを確認する。1チャネル内の全画素の走査が完了していればFinishへ遷移し、未走査要素があればFetch_Dataへ戻る。
9. **完了通知（Finish）**：上位へ終了信号doneを通知し、Idleへ復帰する。

3.3 演算コア（kernel_multiple）の実装

kernel_multipleは、deconv_layerの制御下で動作する演算モジュールである。その内部構成を図9に示す。

3.3.1 各サブモジュールの機能

本モジュールは内部に以下の主要なサブモジュールをインスタンス化している。

1. **multiple**: 入力値と重み値の乗算を行う。本設計では、メモリ帯域を有効活用するため、4×4のカーネル内の横一列4要素に相当する32bit幅（8bit $\times$ 4）の重み値が一括して供給される。そのため、座標信号に基づいたバイト選択ロジックにより、必要な8bitの重み値を抽出した上で16bitへの精度拡張を行い、入力値との乗算を実行する。
2. **adder**: 乗算結果をAccumulator_RAMの蓄積値に累積する。32bit幅の加算器を実装しており、読み出された蓄積データと現在の乗算結果を合算する際のオーバーフローを防ぎ、演算精度を維持する役割を持つ。

3.3.2 kernel_multiple 内におけるデータフロー

kernel_multiple内部では、上位のControllerによって送られてきた入力値と重み値と蓄積値を用い、1座標ごとに演算を実行する。

まずmultipleで32bit幅の重み値から、現在の演算座標に対応する特定のバイトデータが抽出される。この重み値と入力値による乗算結果productは、adderにおいて蓄積値と合算される。その合算値y_wdataは、新たな蓄積データとして即座に元の同一アドレスへと書き戻される。

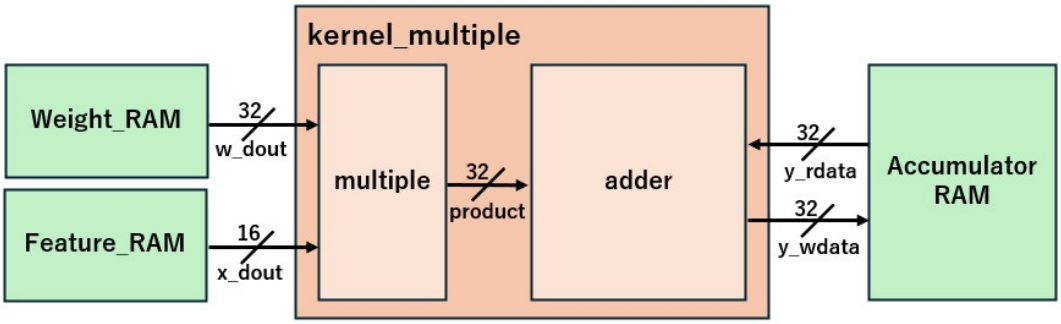

図 9　kernel_multiple の内部構成

この動作を繰り返すことで、1出力チャネル分の演算が完結する。

4　FPGA 実装の中で工夫した点

本章では特にFPGA実装を行うにあたって工夫した点についてまとめる。

4.1　RAM の再利用

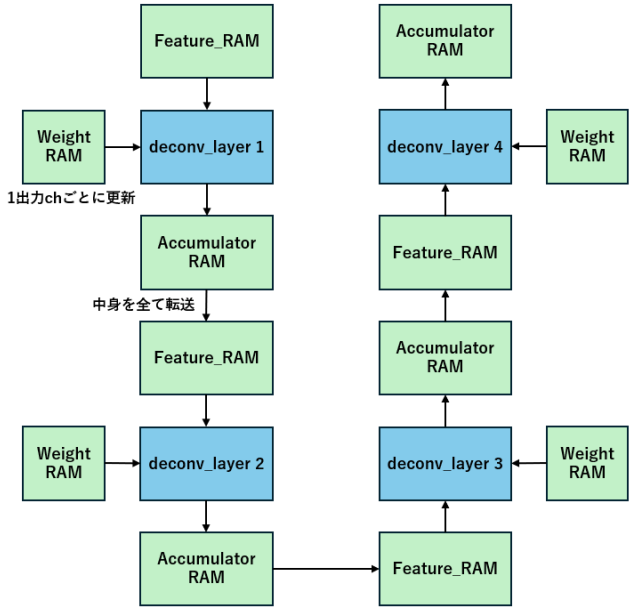

図 10　RAM 再利用に注目した全体アーキテクチャ

本システムでは、RAMを最大限に活用したアーキテクチャとなっている。図10に示す通り、入力データを保持していたFeature_RAMの出力は、演算コアを経てAccumulator_RAMへと蓄積される。1層分の演算が完了すると、Accumulator_RAM内のデータはSaturationとReLUを経て、次層の入力データとして再びFeature_RAMへ書き戻される。このように、入力特徴マップ用メモリと蓄積用メモリ間でデータを循環させることで、層の数に比例したメモリ消費を抑制している。Weight_RAMには、全層の重みパラメータを一括で保存しない。上位ステートマシンの制御に基づき、現在計算している「1出力チャネル分」の重みのみを随時外部メモリからロードして再利用する方式をとっている。

4.2　Block RAM の制限を考慮した回路構成

前述した計算手順をとった理由について詳しい説明を行う。仮に本システムの計算に用いるパラメータや入力要素をすべてFPGAボードのBlock RAMに保存する形式をとった場合のメモリ使用量の概算は表5のようになる。なお、使用したZCU104に搭載されるBlock RAM 1つにつき36kBのメモリ量を持つことをもとにして計算を行った。この時、実装に必要なBlock RAMの個数は7,943個となるため表1に示したZCU104に搭載されている312個のBlock RAMでは到底賄うことができないリソース量であることがわかる。そこで私たちのチームでは、PL部に1層の計算の中でも特に1出力要素に必要な重みのみを保存し、これを随時更新することで演算を行うアーキテクチャを設計した。このようなアーキテクチャをとることによって図1に示したweight-packed分の重みのみをPL側のRAMに

保存すればよいこととなるため、消費するBlock RAMは表6のようになる。表6より、Block RAMのリソース消費量は347個にまで削減することが可能である。しかしながら、表1に示したようにZCU104に搭載されたBlock RAMは312個であるため、依然としてPL部に実装できないことが推測された。そこで私たちは次に説明するRAMの再利用を行うことによって更なるメモリ消費量の削減を行った。

表 5　メモリの最大使用量

	入力要素 [Kb]	重み [Kb]	Block RAM
1 層目	65	6,553	1,839
2 層目	131	16,777	4,697
3 層目	262	4,194	1238
4 層目	524	16	169
総数	983	27,541	7,943

表 6　各層の 1 出力要素当たりのメモリ使用量

	入力要素 [Kb]	重み [Kb]	Block RAM
1 層目	65	12	24
2 層目	131	65	61
3 層目	262	32	92
4 層目	524	16	169
総数	983	127	347

4.3　RAM の再利用によるリソースの最適化

限られたBlock RAMリソースでGANを実装するためには、個々のメモリのビット幅およびアドレス幅を、格納するデータ量と必要精度に即して最小限に定義する必要がある。本システムにおける各RAMの仕様について以下のように工夫を加えた。

1. **Feature_RAM / Accumulator_RAM のビット幅**: Feature_RAMのビット幅は、外部から転送されるランダム入力 zの精度に適合する16bitを採用した。積和演算の結果を保持するAccumulator_RAMについては、累積加算過程におけるオーバーフローを防止するため、32bit幅を採用した。
2. **アドレス幅（深度）**：FPGAのBRAMは構成後に深度を動的に変更できないため、Feature_RAMのおよびAccumulator_RAMのアドレス幅は、全4層の中で最大となる特徴マップの総要素数（画素数 × チャネル数）から算出した。本ネットワークで最大のデータ量を持つのは第3層の出力時であり、そのデータサイズは以下の通りとなる。

   $$16\,(\text{px}) \times 16\,(\text{px}) \times 128\,(\text{ch}) = 32,768\,\text{要素}$$

   この 32,768 個の要素を一元的に管理するためには、$2^{15} = 32,768$ であることから、アドレス幅を15bitと定義し、メモリ配置の最適化を図った。
3. **Weight_RAMのビット幅**: 重みパラメータは学習済みの8bit精度の値を基本単位としている。本設計ではデータ転送効率向上の観点から、4×4のカーネル内の横一列4要素（8bit × 4）を1つのアドレスに集約して保持する仕様にした。 そこで、4つの重み要素（8bit × 4）を単一のアドレスに集約した32bit幅を採用した。
4. **Weight_RAMのアドレス幅**: 本システムでは全パラメータを一括で保持すると膨大なリソースを消費するため、出力を特定のチャネル数ごとに分割して演算を行う。本ネットワークにおいて入力チャネル数が最大となるのは第2層（512 ch）である。そこで、リソース制約と演算速度のバランスを考慮し、出力チャネルの分割単位を64とした。すると、重みメモリのアドレス幅は、入力チャネル数と出力チャネルの分割単位を掛け合わせた以下の式で導出できる。

   $$512\,(\text{入力 ch}) \times 64\,(\text{出力 ch}) = 32,768\,\text{カーネル}$$

   本設計では、演算効率化のためこれら4つ

の重み要素を1つのアドレスに集約して保持する。したがって、実際にWeight_RAMに割り当てる必要なアドレス数は以下の通りとなる。

$$32,768\,\text{要素} \div 4\,(\text{集約数}) = 8,192\,\text{アドレス}$$

この 8,192 個の要素を管理するためには $2^{13} = 8,192$ であることから、アドレス幅を 13bit と定義した。

以上の議論から、PLで用いられるBlock RAM数は表7のように表される。ZCU104に搭載されたBlock RAM数は312であるので、本構成によりハードウェア実装可能である設計に落とし込めているといえる。

表 7　最終的な Block RAM 使用量の概算

	サイズ [Kb]	Block RAM
Feature_RAM	524	16
Weight_RAM	262	8
Accumlator_RAM	1,048	32
総数	1,874	56

5　シミュレーション

ハードウェアはVHDLを用いて設計を行い、そのシミュレーションにはXilinx社のVivadoを使用した。本章ではVivadoを用いて作成したシミュレーションをもとにPL部の動作解説を行う。

5.1　FIFO の分岐動作

入力要素xを送信するFIFO-A、重みwを送信するFIFO-Cの切り替えについて着目する。本回路では、slv_reg0の3ビット目に示されるFIFO選択信号により、AXI-LITEを通じたデータ書き込みがどのFIFOに転送されるかを分岐する仕組みとなっている。この仕組みについて、図11に示したシミュレーション波形をもとに説明する。図11はFIFO-Aに100次元入力を送信した後、FIFO-Cに１つ目の重み$4 \times 4 \times 100$要素を送信した波形を表したものである。PS側からFIFO-Aへのデータ転送の様子を見ると、FIFO-A、FIFO-Cのどちらともものデータ書き込み信号dinに値が入力されていることがわかる。しかしながら、書き込み可能信号wr_enはFIFO-Aのもののみ立っており、この時slv_reg0は01（00001）から03（00011）に変化している。同様に、FIFO-Cを利用してる区間を確認するとPSからPLにデータ書き込みを行う際のwr_en信号はFIFO-Cのもののみ立っており、この時のslv_reg0は05（00101）である。このように、PSからの制御信号によってどのFIFOにデータを書き込むかの分岐が行えていることがわかる。

5.2　Generator の動作

１出力要素を得る手順

まず、１層目の中でも初めの出力要素を求める際の回路動作について説明を行う。図12に100次元入力xの受信から1層目の1出力要素の取得までのシミュレーション波形を示す。図12をもとに計算の動作について確認する。まず、本波形で確認する計算過程は次の通りである。

1. PS側から100次元入力xが送られるまで待機。取得後値をRAM-X（Feature_RAM）に格納
2. PS側から１つの出力要素を得るために必要な重みwが送られるまで待機
3. 取得後値をRAM-W（Weight_RAM）に格納し、計算開始
4. 計算終了時、計算終了信号を発信し、重み待機状態に遷移

まず手順１にて、演算に必要な100次元入力xをGeneratorに取り込む。実際に、ram_x_dinの動作からRAM-X（Feature_RAM）に入力データが書き込まれていることが確認でき

る。この時、動作モードはFIFO-Aを選択したうえで計算開始信号を立てることとなるため、slv_reg0は01から03に遷移する。
続いて手順２において、Generatorへの重み送信が開始するまでGeneratorは待機する。この時、PS側はPLに対して重みwを送信し、FIFO-Cはデータを蓄える。この際Generatorは計算を行わないため、計算回路の内部状態を表す信号stは、重みwを待つWAIT_W_REQとなっている。また、動作モードは計算開始信号が0かつFIFO-Cを選択するため05に遷移する。
手順3ではGeneratorが重みwを取得する。Generatorは計算に必要な入力xと重みwがそろい次第計算を開始する。図より、演算結果を保存するRAM-Y（Accumulator_RAM）に出力１要素分である$4 \times 4 = 16$要素以上の書き込みがあるが、これは計算回路の仕様による挙動である。計算回路ではRAM-X（Feature_RAM）から入力xのうち1変数、RAM-W（Weight_RAM）から得た重みwのうちの1変数を掛け合わせこれをRAM-Y（Accumulator_RAM）に書き込む動作を繰り返す。1出力要素(4×4)を得るためには、100次元の入力xと100次元の重みwの積を足し合わせる必要があるため、計算回路ではRAM-Y（Accumulator_RAM）への書き込みを行った後、次の計算結果とRAM-Y（Accumulator_RAM）の読み出しデータとの和を再び書き込む試行を繰り返す。このことから、シミュレーションのような波形となる。また、信号stは計算終了待機のWAIT_COREとなる。また、動作モードは計算開始信号が1となるため07に遷移する。
最後に、手順４では計算終了時に計算終了信号であるl1_doneを立てていることが確認できる。この信号はAXI-LITE modlueを通じてPS側に送信され、PS側は再び手順1-4を繰り返す。このことから、slv_reg0は再び05に戻る。このような操作をそれぞれの層の出力要素分行うことによって１層分の計算を完了することができる。

１層分の計算を終了した際の手順

図13に1層目の計算から2層目の計算に遷移する際のシミュレーション波形を示す。1層目の計算では、4×4のサイズを持つ512要素の出力を得る。このことから、出力要素を示すcur_ocは0-511の値を遷移することとなる。実際にシミュレーションは波形を確認するとcur_ocが511となる区間があり、１層目の計算が終了していることが確認できる。ここで、1層分の計算を終了した際の手順を、１層目から２層目に遷移する際を例に以下で説明する。

1. 1層目の最後である512番目の出力要素を計算し、計算終了時にl1_doneを立てる
2. 内部状態がMOVE_DATAに遷移し、RAM-Y（Accumulator_RAM）のデータをReLU関数にとした後、RAM-X（Feature_RAM）に格納される
3. RAM-X（Feature_RAM）への格納が終了した際に、end_moveが立つ。RAM-X（Feature_RAM）を入力とした計算を行うために、次の層の計算で用いる重みwのロードを開始する

まず、手順１について「１出力を得る手順」でも説明したように１出力要素を得る計算が終了した際にl1_done信号が立つ。この時、cur_ocが計算を行う層を指定するstate_calによって定まる出力要素と一致しているとき、重みwの受付を終了しデータ移行状態のMOVE_DATAに遷移する。実際にシミュレーション波形より、手順１までの計算は重みを格納したRAM-W（Weight_RAM）の出力および入力要素のRAM-X（Feature_RAM）の出力を用い

た演算結果を、RAM-Y（Accumulator_RAM）に書き込むことで完結している。
続いて、手順２では入力xと重みwの積が保存されたRAM-Y（Accumulator_RAM）の値にReLU関数を適用し、RAM-X（Feature_RAM）に保存する。実際に、シミュレーション波形では手順２においてram_x_dinが受け付けられていることがわかる。
手順３では、RAM-X（Feature_RAM）へのデータ移行が終了した際にend_move信号をPS側に送信する。PS側がこの信号を受け付けることによりPSは１層目の計算が終了したことを確認する。実際に、計算する層の層数を指定するstate_calはend_move信号が立つとともに1に遷移していることがわかる。また、slv_reg0の値は0d（01101）に遷移していることがわかるため、state_calも変化していることが裏付けられる。このように、１層分の計算が終了した際には出力結果をReLU関数を用いて整形して2層目の入力として用いるような回路動作が行えていることが確認できる。

計算終了時の手順

図14に計算終了時のGeneratorにおけるシミュレーション波形を示す。このシミュレーション波形は、３層目の計算が終了した後４層目の計算を行い、計算終了信号を送るまでを表した図である。実際に、計算される層を表すstate_cal信号は2から3に遷移しており、４層目の計算を行っていることが確認できる。計算終了までの手順を３層目から４層目の計算を行う部分を例に説明すると以下の通りとなる。

1. １層分の計算を終了した際の手順で示したように３層目の計算を終了し、RAM-X（Feature_RAM）に入力データを書き込む
2. 4層目の計算に必要な重みを受け取ったのち、計算を開始する
3. ４層目の計算終了した後、RAM-Y（Accumulator_RAM）に格納された32×32要素にTanhを適用し、１つの信号として繋げたものをend_allと共に出力として送る

まず、手順１についてシミュレーション波形からもend_moveが立っていることから正常動作していることが確認できる。続いて手順２について、今までと同様に重みの送信と計算開始信号start_wを送ることで計算開始していることがわかる。また、slv_reg0は1d（11101）で重みのデータ送信、1f（11111）で計算開始となっていることも確認できる。最後に手順３にて、RAM-Y（Accumulator_RAM）に格納した値にTanhとデータを連結する処理を加えた値をout_imgとして出力する。実際に、図14の拡大波形より、l4_doneから計算終了信号end_allが立つまでの間に出力信号out_imgの更新が行われており、この間にRAM-Y（Accumulator_RAM）の値に対するTanhの適用とデータ連結を行っていることがわかる。このことから、Generatorから出力される値は$32 \times 32 \times 8$bitであり、この信号が1clkで出力されることがわかる。

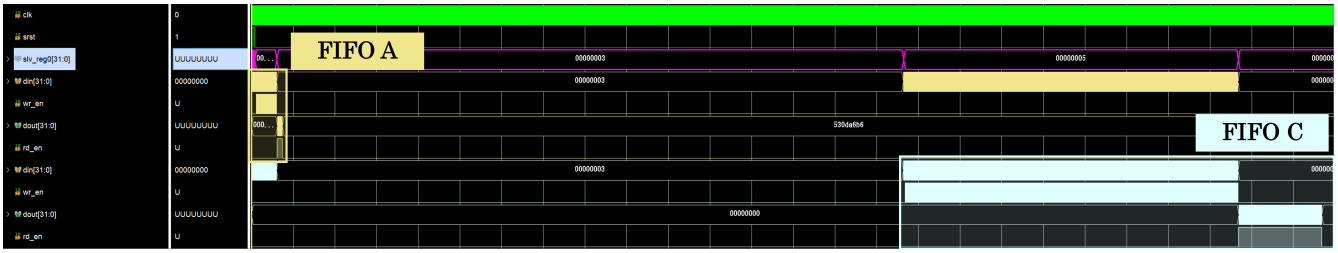

図 11　FIFO の分岐動作に関するシミュレーション波形

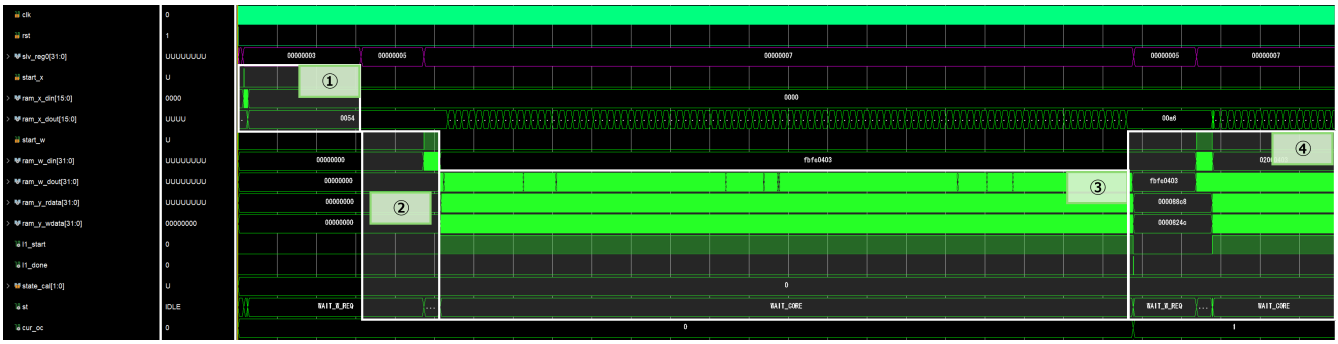

図 12　1 出力要素分のシミュレーション波形

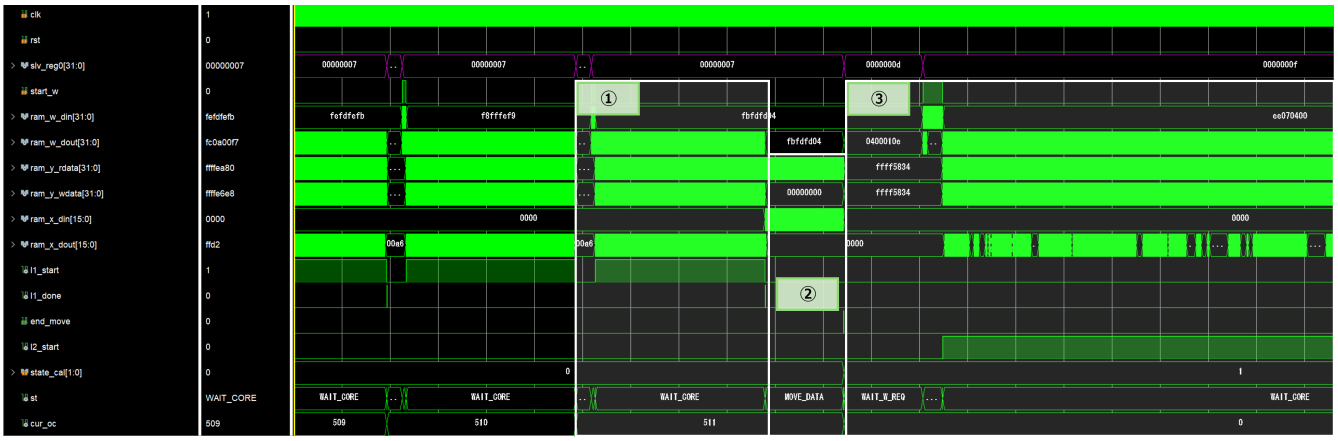

図 13　1 層目から 2 層目の計算に変化する際のシミュレーション波形

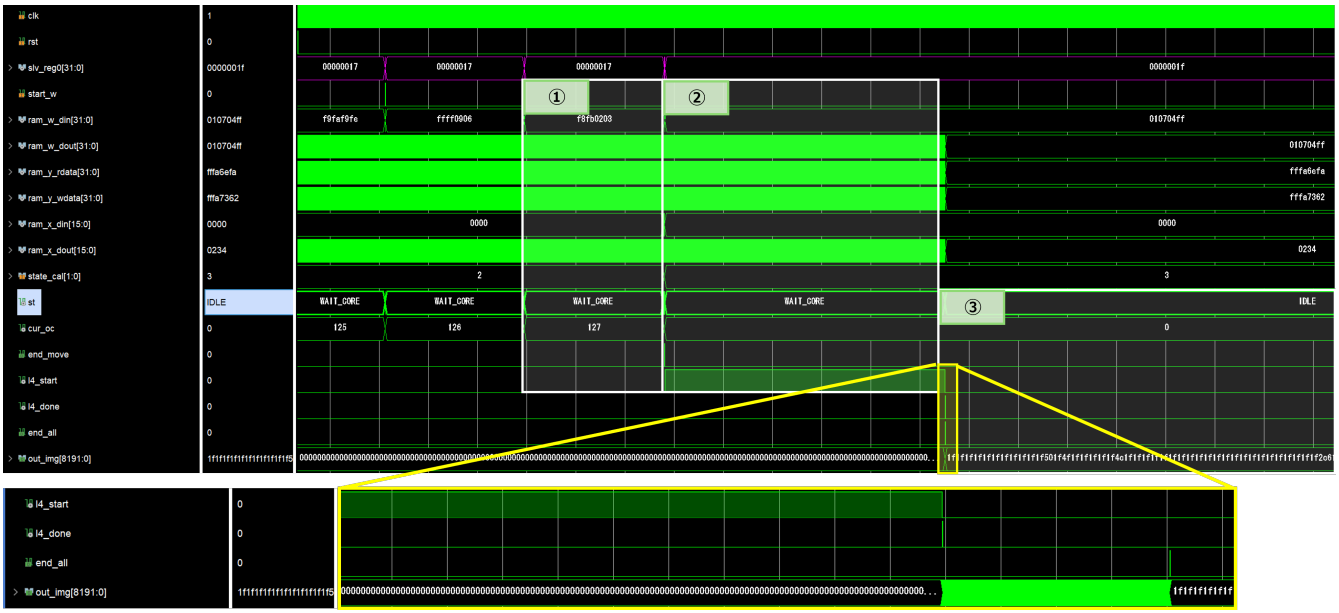

図 14　Generator 計算終了時のシミュレーション波形

6　実機動作

本章では、実際にシステムをFPGAボードで実機実装した際の結果をまとめる。

6.1　回路規模と動作周波数

本システムのうち、PL部が使用したリソースと最大周波数を表8に示す。なお、最大動作周波数f_{max}の計算には、Vivadoで配置配線を行った際に得たWNS（Worst Negative Slack）をもとに以下の式によって算出した。

$$f_{max} = \frac{1}{1/f - WNS} \tag{1}$$

現時点で実装を終えた回路では、1から4層目までのそれぞれの回路で1つの乗算器を利用している。このことから、使用リソースのうちDSP sliceは4つ利用していることがわかる。また、BRAMの使用量は「FPGA実装の中で工夫した点」で示したものとおおむね予測通りになっていることも確認できる。少し少なくなっている点に関しては、論理合成を行った際に1つのBRAMにtanhやFIFOといった容量の少ないRAMやROMを合わせて格納することでリソース消費量を削減していると考えられる。

表8　使用リソースと最大動作周波数

	Used resources	Utilization [%]
LUT	13454	5.84
LUTRAM	86	0.08
FF	19464	4.22
BRAM	52	16.67
URAM	1	1.04
DSP	4	0.23
BUFG	3	0.55
Maximum Frequency[MHz]	118.04	

6.2　CPU との性能比較

今回実装したシステムと同様のモデルを表9に示す環境で動作させた際の性能比較を行う。

表9　エミュレータ PC の仕様

CPU	Intel Core i7-1165G7 2.80GHz
RAM	32 GB
OS	Windows 11 home
Language	Python 3.11.14

100次元入力を入力し、1つのグレースケール画像を得るまでにかかった計算時間はCPU

とFPGAとでそれぞれ表10のように計測された。表10より、FPGAの計算時間はCPUのものに比べて2.28倍の高速化を達成していることが確認できる。

表 10　CPU と FPGA の計算時間比較

CPU	FPGA
304.26[s]	133.32[s]

また、ランダム入力に対する画像出力についてエミュレータから得られたものとFPGAから得られたものを比較したものを図15に示す。図15より、エミュレータとFPGAの出力結果は概ね同様になっていることがわかる。このことから、エミュレータ環境と同等のシステムをFPGAボードに実装することができたといえる。

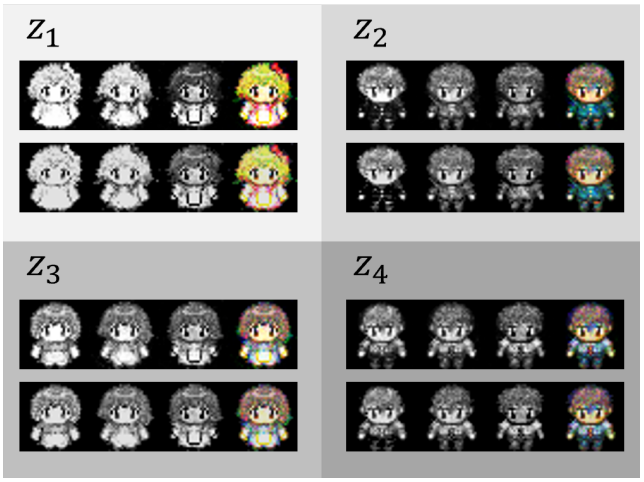

図 15　エミュレータと FPGA の出力比較
（上：エミュレータ、下：FPGA）

7　今後の展望

本研究で開発したシステムは、回路リソースの消費を抑えつつ、GANによる画像生成の基本動作をFPGA上で実現することに主眼を置いた。しかし、実用的なアバター生成を想定したリアルタイム処理を実現するためには、ソフトウェア、ハードウェア共にさらなる最適化が不可欠である。本章では、設計されたシステムについて現在実装中である要素について説明する。

7.1　学習法改善による高解像度化

今後の展望として、ソフトウェア上での学習手法を工夫することにより、より高画質な画像生成が実現できると考えられる。現状のFPGA実装では、リソース制約の観点から、出力可能な画像サイズには制限がある。しかしながら、本研究で用いた固定ベクトルによる新たな画像空間生成を、RGB成分ではなく分割画像に対して適用することで、この制約を回避できる可能性がある。具体的には、64×64画像を4分割して学習を行い、FPGA上では32×32画像を4枚出力する構成とすることで、実質的に高解像度画像の生成が可能になると考えられる。本手法に関する具体的な検証実験については、補足資料1に示す。

7.2　演算処理の並列化

図16は本システムのうち、3層目の計算の126番目の計算におけるシミュレーション波形を抜き出したものである。その中でも白枠で示した部分が重みのLOADにかかる区間、赤枠で示した部分が入力と重みを用いて1出力チャネルを得る計算部分となっている。図16からわかるように実際に計算を行う部分が実行時間の中でも多くの割合を占めていることから、現在のアーキテクチャは演算ボトルネックであるといえる。そこで、演算処理の並列化を行うことを目指す。

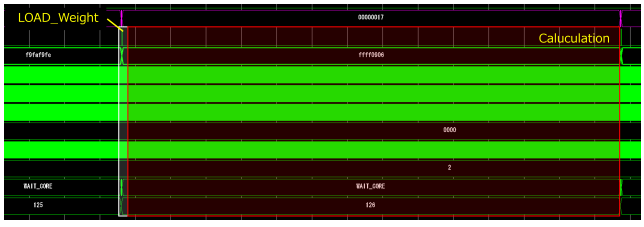

図 16　演算時間のボトルネック特定

図17に示すようにメモリ帯域を拡張し、か

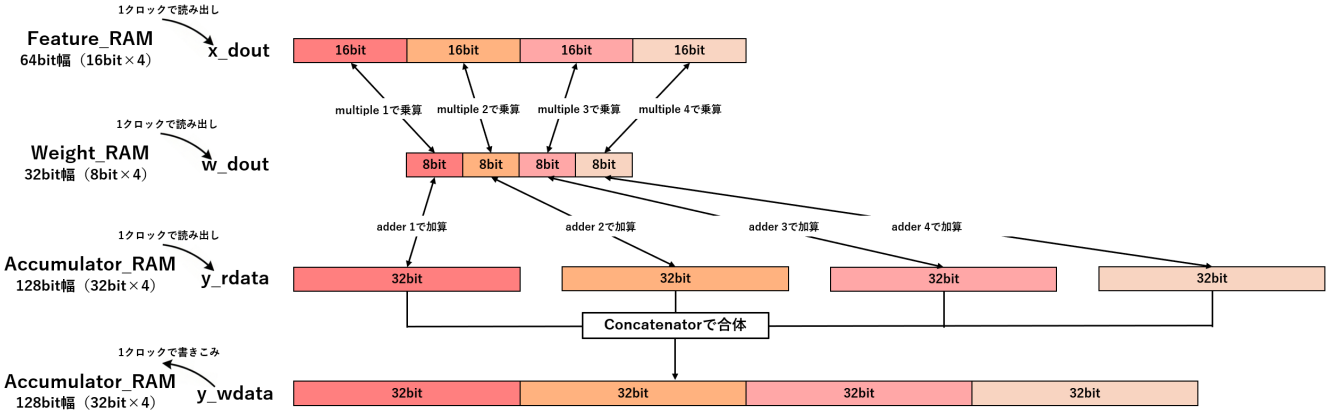

図 17　ビット帯域の分割による並列化の概念図

つ演算に用いていたkernel_miultiple1を各層で４つ準備することで並列演算を行うアーキテクチャを導入する。

現在の設計では、すでにWeight_RAMは4 × 4のカーネル内の横一列4要素を保持するため、32bit幅（8bit×4要素）を採用している。しかし、FPGAに実装されたブロックRAM構造上の制約により、同一クロック内で複数アドレスへの同時アクセスは不可能である。このため、従来の手法では1つのカーネル演算に対して4クロックを要していた。

この課題を解決するため、Feature_RAMを64bit（16bit×4要素）、Accumulator_RAMを128bit（32bit×4要素）へとそれぞれ拡張し、演算コアであるkernel_multipleを4系統並列に配置する設計変更を行う。これにより、1クロックの読み出しで、4要素分の入力値および蓄積値を一括して演算コアへ供給できる。

また、演算後の4つの積和結果を単一データに統合するため、新たに結合モジュールConcatenatorを導入する。これにより、複数のBRAMを消費することなく、4つの独立した演算結果を128bitの単一データとして1クロックでメモリへ書き戻すことが可能となる。特に図16のように、処理時間の大部分が計算時間になっているため、本並列化により計算時間は約4倍の33.33秒まで高速化できると予測できる。

7.2.1　4 並列化した演算コアの設計

図18に示す4並列化した演算コアでは、以下のステップで同期演算を実行する。

1. **一括データ供給**: Feature_RAMから読み出された64bitの入力値と、Accumulator_RAMからの128bitの蓄積値を、4基の演算ユニット（multiple nおよびadder n）へ同時に供給する。
2. **並列積和演算**: 各ユニットは、供給されたデータの中から自身が担当するビット範囲を直接抽出して演算を行う。
   - **multiple n**: 64bitの入力値から特定の16bitを選択し、同時に供給される32bit重み値のうち対応する8bit要素を抽出する。抽出された8bit重み値は、乗算前に16bitへと精度拡張され、選択された16bit入力値との乗算が実行される。
   - **adder n**: 128bitの蓄積値から対応する32bitを抽出し、同じkernelの乗算結果と合算する。
3. **出力の同期結合**: 各adderから出力された32bitずつの演算結果は、Concatenatorによって128bitデータへ合体され、

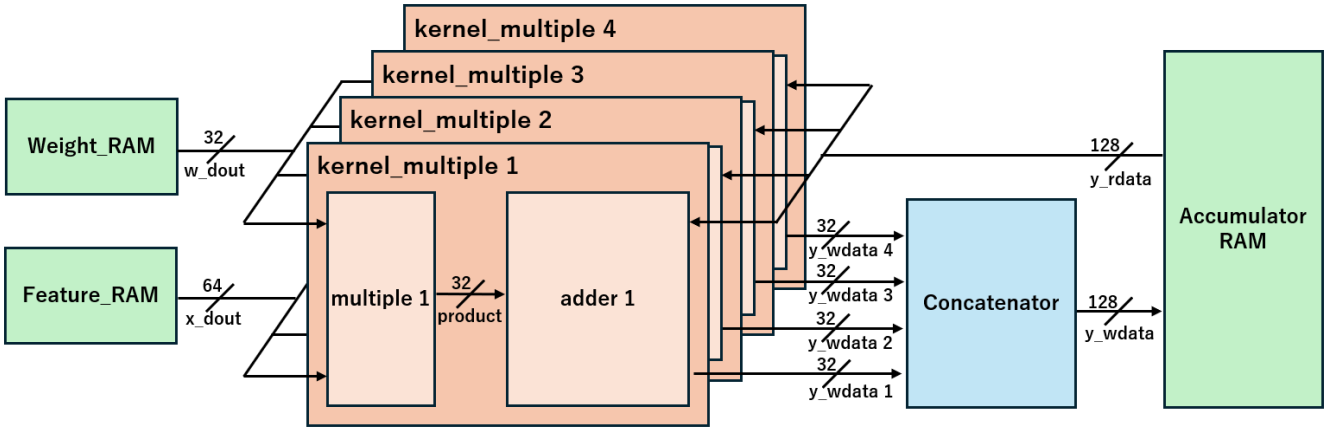

図 18　4 並列化した演算コア構成

Accumulator_RAMの同一アドレスへ一括して保存される。

7.3　フルカラー画像生成への対応（RGB 並列化）

本研究で開発したシステムは、1チャネル（モノクロ）の演算を順次繰り返す逐次実行方式を採用している。そのため、フルカラー画像を生成する際にはR・G・Bの各チャネルに対して個別に演算サイクルを回す必要があった。そこで、前述したカーネル演算の4並列化に加え、RGBの3色を同時に処理する並列を加えることで、合計12並列計算を計画している。

この設計の最大の特徴は、学習時に導入した固定ベクトルv_1, v_2を活用することで、単一の重みパラメータをRGBの全チャネルで共有し、外部からの入力値（ランダムノイズz）や重みの転送コストを一切増加させることなくフルカラー化を実現できることである。

7.3.1　12 並列演算コア (kernel_multiple) の設計

図 19 に示すように、演算の核となるkernel_multipleユニットは、合計12個（4要素並列×3チャネル並列）の構成をとる。

単一のWeight_RAMから読み出された32bitの重み信号は、RGBすべての演算コアへ同時に供給される。RGB各チャネルに配置された4つのkernel_multipleは、この共通の重みを用いながら、それぞれの色成分に対応する入力値に対して独立した積和演算を実行する。また、RGB各チャネルは同一座標を同時に処理するため、ステートマシンやアドレスは共通化できる。つまり、単一のControllerやloop_counter、addr_generatorで一括制御が可能である。この制御ロジックの共用化により、リソース消費を抑制しながら高密度な並列演算パスを構築している。

7.3.2　ハードウェア構成の拡張

12並列化に伴い、top_moduleの構成を図20のように拡張する。

1. **メモリ構成（計7基のRAM）**：重みパラメータを保持するWeight_RAMは1基のみとし、RGB全チャネルで共有する。一方で、各色独立した積和演算を行うため、入力値を保持するFeature_RAMおよび蓄積値を保持するAccumulator_RAMは、RGB各チャネル専用に3基ずつ配置する。
2. **Input_RGB_Gen の追加**: Input_Deserializer によって受け取られた100次元の入力ノイズzに対し、固定ベクトルv_1, v_2を加算して$z, z+v_1, z+v_2$の3系統を生成するモジュールInput_RGB_Gen

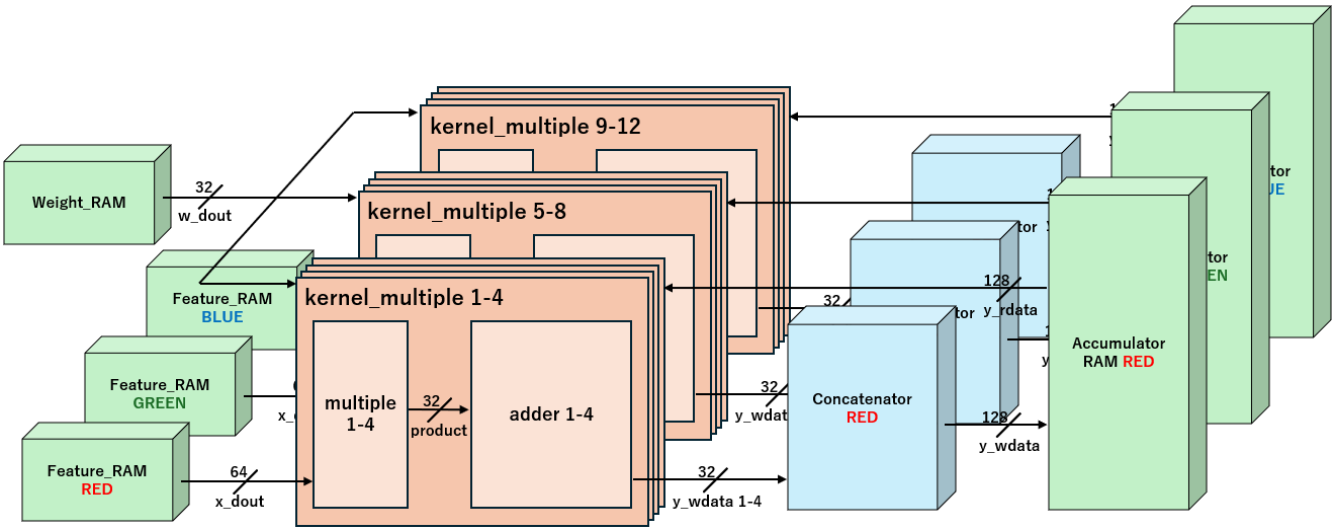

図 19　12 並列化した演算コア構成

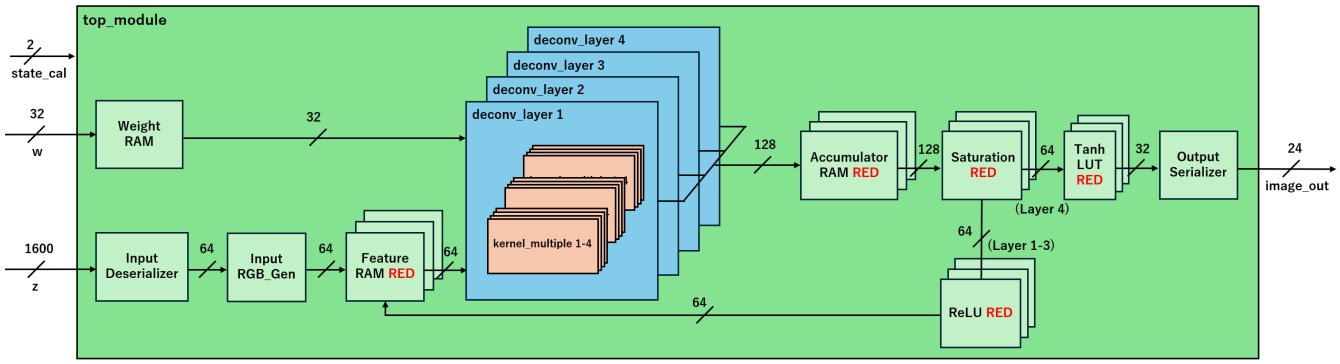

図 20　拡張された計算回路の構造

を追加する。本モジュールの導入により、外部からの転送部を変更することなく、内部でRGBの入力特徴マップを同時生成することが可能となった。

3. **演算部・活性化関数の三重化**: 各チャネルの独立性を保つため、Saturation、ReRU、Tanh_LUTの各サブモジュールをそれぞれ3系統配置する。これにより、RGBの全チャネルが最後まで同時に処理される。
4. **Output_Serializerの仕様変更**: 最終層の演算完了後、各チャネルのTanh_LUTから出力された3系統の8bit画素データは、Output_Serializer内で24bitの単一パケットimage_outへと集約され、一括して外部へ出力する。これにより、出力にかかる処理時間を大幅に短縮する。

本システムの実現可能性もハードウェアリソースの観点から確認する。kernel_multiple1回路の並列化数が12倍となることから、利用するDSP sliceも同様に12倍となる。このことから、表8より、48個のDSP sliceを消費する。また、Feature_RAMおよびWeight_RAMはカラー並列化により 3 倍のメモリ容量を占有する。このことから、予想されるメモリ使用量は表5と同様に考えると表11のようになる。ZCU104に搭載されたblock RAM総数は364であることからUtilizationは約47%となり、さらにFIFO等の通信で利用されるメモリを

さらに増やしたとしても十分実現可能であるといえる。

表 11　演算、カラー並列化時の Block RAM 使用量

	サイズ [Kb]	Block RAM
Feature_RAM	1,572	48
Weight_RAM	196	96
Accumlator_RAM	3,145	6
総数	4,913	148

8　まとめ

今回、アバター画像からGAMを用いて新たなアバターを生成するシステムの実装を行った。学習モデル作成に当たってはFPGA実装を踏まえ、１つのネットワークでRGB空間を表現できる学習モデルを構築した。回路設計を行うに当たっては、大規模なCNNモデルをFPGAの限られたリソース量で再現するために、PS,PL協同動作するシステムを設計するとともに、今後の拡張性を高めた設計に力を入れた。これらの設計が功を奏した結果、FPGAボードにシステムを実装することができ、CPUと比べての高速化や同様の出力結果を得ることができた。また、11章に示したような今後の展望についても深く考察し、実現可能性を考えることができた。一方で、今後の展望で示した部分は未だに設計段階であり実装途中のものとなる。これからは特に今後の展望に示した実装に力を入れ、より実運用に近いシステムへと昇華させることに取り組みたい。

9　最後に

今年の設計課題はGenerative Adversarial Networks（GAN）であり、新規画像を作成できるという強みに着目しました。この着眼点において、社会的インパクトやオリジナリティ、実現可能性の観点を考慮した結果今回のドット絵のアバター作成をテーマとして選択いたしました。私たちのチームは全員ともハードウェア設計初心者であったことから、テーマ策定の段階において実現可能性を考えることがとても難しかったです。このような状況の中でも、歴代の先輩方の実装された回路をもとにパラメータや回路リソースの目安を図りながらも、私たち独自のシステムを構築することを心掛けました。特に私たちの実装した「複数層で成り立つCNNモデル」は今までででも初の試みであり、どのようなアーキテクチャで実現するかはメンバー全員での度重なる議論と情報収集により成しえることができました。また、メンバーそれぞれが技術的強みを発揮し、ディープラーニングの観点からも、回路構成の観点からも面白いものを作成するよう努力いたしました。

この大会を通じて、深層学習やハードウェア設計に関する知見を深めることができたのはもちろんのこと、チームで１つのものを「モノ」を作り上げる難しさと楽しさを実感しました。メンバーそれぞれが「ユニークなシステムを作成する」という目標に向かい開発を行いましたが、やはり途中工程では認識のずれが生じるような場面もありました。このようなずれや様々な課題を乗り越え、本文で示しました実機実装まで行うことができた際は大きな達成感を得ることができました。このように、チームメンバーそれぞれの「ユニークさ」の主体性がアバター画像です。チームで取り組み１つの成果物を作り上げたこの経験を今後にも大いに生かしていきたいと思います。また、今後の展望にも示しましたように構想したすべての回路構成を実装することはいまだ完遂できておりませんので、引き続き実装を続けつつ、チームで納得のいく成果物を作り上げたいと思います。このような貴重な機会を与えてくださったLSI Design Contest

2026実行委員会の皆様、関係者の方々にチーム一同より心から感謝申し上げます。

参考文献

[1] Advanced Micro Devices, Inc. *ZCU104 Evaluation Board User Guide*. AMD.
[2] Advanced Micro Devices, Inc. *What is PYNQ*. AMD.