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
VLD2022-22,ICD2022-39,DC2022-38,RECONF2022-45(2022-11)

多様な CGRA を実現する Diplomacy を活用した設計手法の検討

小島　拓也[†,††]　齋藤　　真[†]　中村　　宏[†]

† 東京大学　113–8656 東京都文京区本郷 7 丁目 3–1
†† JST さきがけ　102–0076 東京都千代田区五番町 7
E-mail: †{tkojima,saito,nakamura}@hal.ipc.i.u-tokyo.ac.jp

あらまし　Coarse-Grained Reconfigurable Architecture(CGRA) はエネルギー効率や計算性能に優れる再構成可能ハードウェアである．これまでに数多くのアーキテクチャや実行モデルが提案され，その有効性が確認されてきた．近年は，ユーザーが求める CGRA を容易に設計するためのフレームワークの提案が盛んに行われている．しかし，ハードウェア設計はソフトウェアほどパラメータ化が容易ではなく，既存のフレームワークでは特定の方式の CGRA しか想定されておらずスケーラビリティに乏しいという問題を抱えている．そこで，本研究では Chisel および Diplomacy を導入し，さまざまな方式の CGRA を同時にサポート可能な設計フレームワークの実現を目指し，その初期検討として演算モジュールのクラス設計と実装を行なった．指定するパラメータの変更を最小限に抑えつつ，生成されるハードウェア設計を柔軟に変更できることを確認した．

キーワード　再構成可能アーキテクチャ，CGRA，Chisel，Diplomacy

1. はじめに

「データ集約型科学」へのパラダイムシフトにより現代の計算機に対する要求は多様化している．特に，ビックデータ処理や AI 処理などをはじめとするこれからの社会で中核を担う処理は，計算スループットや低遅延性，消費エネルギーに関して厳しい要求を課す．一方で，近年は半導体微細加工技術の向上に陰りが見え始め，従来の汎用プロセッサではこれらの要求に応えるのは困難になっている．こうした背景から，近年は領域固有アーキテクチャ DSA(Domain Specific Architecture) に注目が集まっている [1]．これは汎用のプロセッサコアを用いるのではなく，対象アプリケーションに固有の計算処理に特化したコアを利用するというアプローチである．AI アプリケーションの爆発的な普及と需要の高さから DSA としての専用チップ化された実例は Google 社の TPU[2] や Cerebras 社の CS-2 [3] などの AI アクセラレータが大半を占める．

DSA を実現するもう一つの方法として再構成可能ハードウェアの活用が期待されている．特に，FPGA(Field-Programmable Gate Array) は市場にも広く普及している．FPGA は任意の論理関数を実現可能な LSI であり，ビットの単位で回路の再構成が可能である．このような柔軟性の高さから適用可能なアプリケーション領域も多岐に渡り，深層学習 [4] だけでなくデータベース処理 [5] などへも応用されている．しかし，ビット単位での再構成による電力，性能上のオーバーヘッドが大きいというのが問題となっている．

したがって，粗粒度再構成可能アーキテクチャ CGRA(Coarse-grained reconfigurable architecture) と総称される粒度の異なる再構成可能ハードウェアが DSA を実現するためのハードウェアプラットフォームとして期待されている．CGRA はデータフローレベルで再構成が可能で，FPGA と比較し消費電力と性能の両方に優れており，その効率は専用チップにも迫るものである [6]．Raghu Prabhakar ら [7] の報告によれば，FPGA と比較して CGRA は最大で約 95 倍の性能向上と約 76 倍のエネルギー効率を達成している．

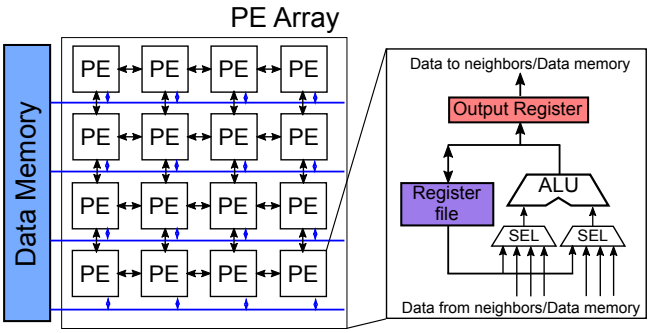

図 1: 一般的な CGRA の構成

CGRA の演算処理部は一般に図 1 に示すような構成をしている．小規模な演算処理ユニット (PE) を 2 次元のアレイにして持ち，隣接する PE は互いに相互接続網で結ばれている．PE で実行する演算の種類や接続網のルーティング経路をアプリケーションに応じて切り替えることができる．このようにして，対象とするアプリケーションが持つ計算データフローに特化したパイプラインを実現できる．さらに，FPGA と比較して CGRA の粗い再構成粒度は配置配線の最適化問題を緩和する．ゆえに，CGRA はコンパイル時間削減を目的として FPGA のソフトコアとしても利用される [8], [9]．

CGRA の研究に関する近年のトレンドは設計環境や設計探索のフレームワーク化である．例えば，パラメタライズした設計を提供し，ユーザーが与えたパラメータに応じて CGRA の設計として Verilog コードを生成する OpenCGRA[10] や CGRA-ME[11] などがある．しかし，これらのパラメタライズには限界があり，特定の実行モデルや制御手法を前提としている．一方で，CGRA は演算の実行モデルや制御モジュールなどさまざまな種類の方式が存在し，各々が一長一短である．ゆえに，既存のフレームワークを利用しても広域的な設計探索を行うことがで

きていない．本研究は, この問題を解決するために, より多様な方式を実現可能な設計手法およびそれに基づく設計フレームワークの実現を目指す．設計のパラメータ化には Chisel[12] と呼ばれる Scala 言語のドメイン固有言語 (DSL) として実装されているハードウェア記述言語を採用する．さらに, Chisel の拡張機能である *Diplomacy* を導入することでより柔軟なパラメータ化を図る．本稿では, Diplomacy に基づくその設計方針と初期実装の結果を報告する．以降の構成は, 2. 節にて CGRA の実行方式やその多様性を説明する．3. 節では本研究が活用する Chisel および Diplomacy によるハードウェア設計手法を説明する．4. 節において、本研究が目指す CGRA の設計環境の実現に向けた初期検討を行い, 5. 節でパラメータ化による効率的なハードウェア設計が可能であることを確認する．最後に, 6. 節にて本稿の結論を述べる．

2. CGRA における多様性

CGRA はアプリケーションの実行時間を支配する計算強度の高いループ部を効率的に処理するアクセラレータとして利用されるのが一般的である．前述の通り, CGRA の主要なモジュールは図 1 に示す PE アレイである．PE は演算器 ALU や中間データを保持するための小規模なレジスタなどで構成される．ALU は汎用プロセッサのように加算や乗算などいくつかの演算をサポートし, オペコードを指定し所望の演算を行う．PE 間には相互接続網を形成するスイッチが設けられ, PE 間で中間データを直接交換し合う．汎用プロセッサとは異なり、中間データのレジスタファイルやメモリへの書き戻しを最小限にすることができる．これまでに提案されてきた CGRA はいくつかの観点で分類することができる．本節ではまず, 再構成方式による分類を行い, それぞれの方式を簡単に説明する．次に, PE アレイの構成方法による分類について述べる．これらを踏まえて, 関連研究としてこれまでに提案されている設計フレームワークを紹介する．

2.1 再構成方式による分類

CGRA における “再構成” とは PE が内包する ALU へのオペコードや, 相互接続網におけるデータのルーティング経路, PE 間に分散したレジスタファイルへの書き込みアドレスの変更を指す．各種ハードウェアモジュールに対するこれらの指示は, 汎用プロセッサにおける機械語のようにいくつかフィールドに分割可能なビット表現を用いて行われる．このデータはコンフィギュレーションデータやビットストリームと呼び, PE アレイ上の専用メモリもしくはキャッシュで保持される．さらに, マルチコンテキスト型のアーキテクチャでは高速な回路切り替えを行うために, 複数のコンフィギュレーションデータを PE 上で保持する．多くの CGRA がホストプロセッサの存在を前提としたアクセラレータであり, データ転送や演算開始などの制御およびアプリケーション中の逐次処理部などはホストプロセッサが行う．ゆえに, 再構成の方式は PE アレイが自発的に変更を行うかどうかでパッシブ方式とアクティブ方式に分類できる．

2.1.1 パッシブ方式

パッシブ方式では回路の再構成や実行の制御など全て管理をホストプロセッサに一任する． MorphoSys[17], EGRA[18], DySER[19], Plasticine[7], SNAFU[15], CMA[20], Riken-CGRA[21] などがこれに該当する．特定のストリーム処理など頻繁な回路の切り替えが不要なアプリケーションにおいては十分な柔軟性であり, 再構成に関わる回路が簡略化できるためエネルギー効率などに優れる．複数のコンテキストを保持している場合は, ホストプロセッサが制御信号を送るだけで再構成が可能となる．1 つのコンテキストしかコンフィギュレーションデータを保持できない場合は, コンフィギュレーション用のメモリは小さくすることができるため小面積や省電力などの利点がある．しかし, 再構成するにはホストプロセッサがコンフィギュレーションデータを送り直す必要があり, その間に演算を行うことはできないためスループットが大きく低下する．

2.1.2 アクティブ方式

アクティブ方式においても演算開始に先立って, ホストプロセッサからコンフィギュレーションデータを CGRA 側に転送する処理は必要となる．しかし, PE には同時に複数コンテキストのコンフィギュレーションを格納させ, 一連のタスクを処理している間は PE が自発的に再構成を行う点がパッシブ方式とは異なる．動的再構成方式とも呼ばれ, 最短でクロックサイクル単位で再構成を行う．計算資源である PE をきめ細かく時分割的に利用していくため, 物理的な PE の数を超える演算数の計算カーネルでも実行ができる．ホストプロセッサからの制御を省略することができるため, 制御のオーバーヘッドが小さい．一方で, CGRA 部に設けるコントローラは複雑化し, エネルギーの面ではパッシブ方式に劣る．

アクティブ方式の CGRA はどのように自発的に再構成を行うかという点でさらに細分化が可能である．ADRES[22] や, FloRA[23], DRRA[24] はコンパイル時に各 PE の動作を固定のタイムスロットへ割り付ける静的スケジュール方式である．再構成にロジックとしてはカウンタがあればよく制御回路を簡略化できる．しかし, コンパイル時にサイクル単位のスケジューリングを行わなければならず, コンパイル処理の負荷が高い．また, 演算器がパイプライン化やマルチサイクル化された場合にスケジューリングはさらに複雑化するため, 計算資源の効率的な利用が難しくなる．対して, *Elastic* 方式と呼ばれる CGRA[25], [26] では, コンパイル時にはスケジューリングを行わず非同期的に各 PE が再構成を行う．PE は演算に必要なデータ (オペランド) が到着するのを待ち, 全てのデータが到着すると計算を行う．計算が完了すると自身の構成情報を次のコンテキストに変更し, 再び待ち状態になる．データの到着を検知するために PE アレイ上を流れるデータには簡単なタグを付与する．

2.2 PE アレイの構成方法による分類

CGRA の演算処理部は一般に PE を階層化したアレイとして構成する．この構成方法によってもいくつかの分類が可能である．

2.2.1 PE の構成

PE はアレイの最小単位である．前述の通り, PE は ALU を保持するがこの構成をアレイ上の全ての PE で共通にするホモジニアス構成と, PE ごとに異なる ALU を持つヘテロジニアス構成がある．ホモジニアス構成は設計が単純化し, コンパイラにおける演算配置の制約がないなどの利点がある．しかし, 乗算器や除算器, それ以外の高級な演算を全ての PE でサポートする場合, 回路面積は増大する．また, メモリバンド幅の制約から全ての PE にメモリアクセスを許すのは困難である．よって, 回路規模のトレードオフを考慮し, PE ごとに実行できる演算種類を最適化したヘテロジニアス構成が近年の主流となりつつ

表 1: CGRA 設計フレームワークの比較

フレームワーク	報告年	フロントエンド	バックエンド	抽象度	再構成, 実行方式	オープンソース化
CGRA-ME [11]	2017	XML	C++	低	アクティブ, 静的スケジュール	✓
OpenCGRA [10]	2020	Python-based DSL	Python, PyMTL	低	アクティブ, 静的スケジュール	✓
DSAGEN [13]	2020	Custom-DSL	C++	中	パッシブ, 非同期実行	✓
Pillars [14]	2020	Scala-based DSL	Chisel	低	アクティブ, 静的スケジュール	✓
SNAFU [15]	2021	N/A	SystemVerilog	高	パッシブ, 非同期実行	
Morpher [16]	2022	JSON like	Chisel	中	アクティブ, 静的スケジュール	✓
本研究の目標	-	script ベース DSL	Chisel+Diplomacy	高	任意	✓

ある [15].

2.2.2 PE アレイ

アレイの構成方法においてもさまざまな方式が存在する．最も単純なものはアレイの階層が 1 つだけのものである．Plasticine[7] は計算用 PE アレイと, メモリアクセスのためのアドレス計算を行う PE アレイを複数並べた 2 階層のアレイを持つ．ほかにも, ARENA[27] のように複数の PE アレイをリングネットワークで接続したアーキテクチャも存在する．

2.3 設計フレームワーク

このような CGRA の多様化や DSA のプラットフォームとしての期待が集まり, CGRA の設計を容易にするフレームワークの提案が盛んに行われている．表 1 にこれまでに提案されているフレームワークをまとめる．RISC-V 命令セットの普及によりハードウェア設計においてもオープンソース化の動きが盛んになっているように, これらのフレームワークもほとんどがオープンソース化されている．利用者に与えるカスタマイズ性の抽象度はさまざまであり, CGRA-ME,OpenCGRA,Pillars などはマルチプレクサやレジスタなどを明示的に指示するレベルでの記述ができる一方で, 抽象度は RTL に近い．DSAGEN や Morpher は中程度の抽象度で, PE のポートやそれに関連するいくつかのハードウェアモジュールの粒度でカスタムが可能であるが, 実行方式などは事前に設計済みのものが利用されることになる．SNAFU ではカスタムの演算を追加する程度のカスタマイズだけを許しており, PE のインターフェースなどを標準化してしまっているためそこに自由度はない．

これらの抽象化したハードウェア記述の表現形式は DSL が用いられていることが多い．いずれのフレームワークもバックエンドの実装が抽象化したハードウェア記述を Verilog などのデファクトスタンダードな HDL に変換する．これは, LSI 実装や FPGA 実装の CAD ツールへ入力することを想定している．CGRA-ME のように独自のソフトウェアとして実装しているケースもあれば, SNAFU のようにパラメタラズされた HDL を用いているものもある．Chisel はソフトウェアとしての Scala コードと HDL に対応する Chisel コードが共存可能であるため, SystemVerilog よりも強力なパラメタラズが可能である．しかし, いずれのフレームワークも抽象度が低い, もしくは高くても再構成方式や実行方式の垣根を越えられず, スケーラビリティに乏しい．また, 独自のバックエンドで実装されている場合, 提案者の想定外の CGRA を設計するにはバックエンド自体を変更する必要があるが, 一般にバックエンドのドキュメント化はされていないためその解読には時間を要し現実的な手段ではない．そこで, 本研究では, ユーザーに与える抽象度を高くする一方で, 任意の形式家の拡張が容易なスケーラビリティを有するバックエンドの実現を目指す．次節では、そのための設計手法についての検討を行う。

3. Chisel ベースの設計手法

3.1 Chisel HDL

Chisel[12] は Verilog,VHDL に続く第 3 の HDL として注目されている．Scala 言語を拡張した DSL として実装されているため, Scala 自体が持つオブジェクト指向型言語および関数型言語としての特徴を受け継ぐ．Chisel で記述されたハードウェアは FIRRTL(Flexible Intermediate Representation for RTL) と呼ばれる中間表現を経て Verilog に変換される．Chisel には FIRRTL の状態で RTL シミュレーションを行う Treadle というシミュレータを持つだけでなく, 生成した Verilog を iverilog, Verilator, Synopsys 社 VCS などのシミュレータを用いてシミュレーションを行う機能も有する．ハードウェアのパラメタラズだけでなく, 動作検証のテストなどにも Scala のソフトウェア実装が利用できる．また, カウンタやプライオリティエンコーダ, FIFO など頻出の実装済みモジュールが Chisel の標準ライブラリとして提供されている．基礎的な文法に関しての説明は入門書 [28] を参考されたい．

3.2 Diplomacy の概要

Chisel で実装された設計およびそのフレームワークとして Rocket Chip Generator[29] がある．これは RISC-V コアを持つ SoC の設計ジェネレータであり, プロセッサのコア数やキャッシュ構成, バスで接続される各種デバイスなどをカスタムできる．このような設計を生成する際に, あらゆるモジュールのパラメータをトップダウン的に指示することでも実現することは可能であるが, スケーラビリティに欠ける．そこで, Rocket Chip Generator では Chisel の拡張機能として Diplomacy[30] を導入している．Diplomacy はモジュール間で直接パラメータをネゴシエーションすることで自動的にパラメータを調整する．各モジュールは仮想的なノードをもち, ネゴシエーションをするモジュール同士の仮想ノードはエッジで結ばれる．仮想ノードは互いに自身の持つパラメータを交換し, 最終的にエッジのパラメータを決定する．エッジはモジュール間の配線に対応し, パラメータはどのような配線が必要であるか, またその配線幅などを定義する．これによって, モジュール間の柔軟な自動接続を行う．

3.3 LazyModule の必要性

Diplomacy と合わせて重要になるのが LazyModule である．純粋な Chisel で記述された HDL では一つのモジュールは Module クラスを継承したものに対応する．このモジュールをインスタンス化すると, 即座にハードウェアの設計が確定する．コード 1 に Module クラスによるインスタンス生成の例を示す．この例では Sub というモジュールが sub0 と sub1 としてインスタンス化されているが, モジュールがパラメタライズされて

いたとしても sub0 と sub1 の間でパラメータを直接やり取りすることはできず, sub1 のインスタンス化を行う頃には sub0 のハードウェアは既に固定されている.

ソースコード 1: Module クラスによる実装

```
class Sub extends Module {...}
class Main extends Module {
  val sub0 = Module(new Sub())
  val sub1 = Module(new Sub())
}
```

一方で, LazyModule は Scala において遅延評価を行う lazy の仕組みに似たモジュール生成が可能なクラスである. LazyModule クラスを継承したクラスは LazyModuleImp クラスを継承したクラスのインスタンスをメンバ module として持つ必要がある. これが実際のハードウェア実装にあたる. コード 2 に同様のクラスを LazyModule で実装した場合を示す. Module で実装した場合と異なり, sub0 のインスタンス化が完了した後も実際のハードウェアはまだ確定しておらず, そのメンバである module 変数が参照されたタイミングでハードウェアが確定する. これによって, sub0 と sub1 のインスタンスが生成された後でもパラメータの調整を行なうことができる.

ソースコード 2: LazyModule クラスによる実装

```
class Sub(implicit p: Parameters) extends
    extends LazyModule {
  // handling parameters
  lazy val module = new LazyModuleImp {
    // implementation
  }
}
class Main(implicit p: Parameters) extends
    extends LazyModule {...}
  val sub0 = LazyModule(new Sub())
  val sub1 = LazyModule(new Sub())
  lazy val module = ...
}
```

4. 提案する設計手法の初期検討

4.1 クラス設計

Diplomacy を用いて CGRA を構成する各種モジュールが互いにネゴシエーションを行い, モジュールのパラメータを決定する仕組みの考案を目指す. 本稿では, その初期検討として PE 内の ALU に対応するモジュール (以降 FunctionUnit と呼ぶ) とそれに関連するクラス設計を行なう. 図 2 に現時点でのクラス設計を示す. ただし, 青色で塗られたクラスは Rocket Chip Generator で実装されているものである. また, 灰色で塗られたクラスは今後実装を進めていくクラスである. FunctionUnit だけでなくレジスタファイルなどコンフィギュレーションデータによって動作が決まるモジュールは ReconfigurableModule という共通の抽象クラスを継承し, 後述するパラメータのネゴシエーションプロトコルを共通化する. FunctionUnit はメンバーとして Operator クラスを継承したインスタンスをリストとして保持する. 加算や乗算などのサポートする演算の一つ一つは Operator クラスを継承したクラスであり, FunctionUnit がどの演算を持つかに応じて機能の異なる計算モジュールが生成できる.

各演算器は入力データを受け取り, 計算を行った結果を一つ出力する. これに加えて, 実装や演算種によって内部状態の通知が必要なものもある. 例えば, 加算器ではオーバーフローが発生する可能性があり, それを検出しなくてはならないかもしれない. また, マルチサイクル化した演算器やパイプライン化されている演算器では busy 状態や stall 状態などを持つことが想定される. 図中には例外を発生するかを示す mayException という関数のみ示している. さらに, 例外を発生しうる演算については Operator クラスのみを継承するのではなく, MayExceptionPort トレイトをミックスインして実装する. ただし, 図中ではスペースの都合上実装が完了している演算モジュールの一部のみを示している.

4.2 Diplomacy によるパラメータ決定

このクラス設計に基づき, Diplomacy によるモジュール間パラメータネゴシエーションを行なう. 図 3 にパラメータネゴシエーションの様子を示す. 各種モジュールは仮想的なノードを持ち, 互いにパラメータを送り合う. FunctionUnit の内部ノードはまず, 保有する全ての演算器からオペランドの数や例外発生の有無などを受け取る. 例えば, 加算や算術シフトなどは 2 つの入力で良いが, 積和演算は 3 つの入力を必要とする. また, 加算や積和演算はオーバーフローが発生しうるが, 算術シフトでは発生しない. 内部ノードはこれらの情報をまとめ, FunctinUnit として例外発生の可能性があるのか, 幾つの入力データが必要であるか, また, コンフィギュレーションデータとして演算を切り替えるためには何ビットのオプコードが必要であるかを決定し, これを駆動するコントローラーや PE のインターフェースに対応するノードへ通知する. 一方で, 親モジュール側で決定したデータのビット幅などのパラメータは FunctionUnit を経由して演算モジュール側へ伝播する.

4.3 加算モジュールの実装例

コード 3 に AddModule(符号なし) の実装例を示す.

ソースコード 3: AddModule の実装例

```
class AddModule(implicit p: Parameters)
    extends Operator with MayExceptionModule
    {
    lazy val module = new OperatorImp(this)
        with HasExceptionPort {
        val w_result = in_channel.
            reduceTree(_ +& _)
        // check overflow
        val overflow = w_result.head(
            w_result.getWidth - in_channel
            (0).getWidth).orR
        exception := overflow
        out_channel := w_result

    }
}
```

I/O の宣言などはパラメータのネゴシエーションの結果により自動生成されるため, 不要である. 例外ポートに出力される

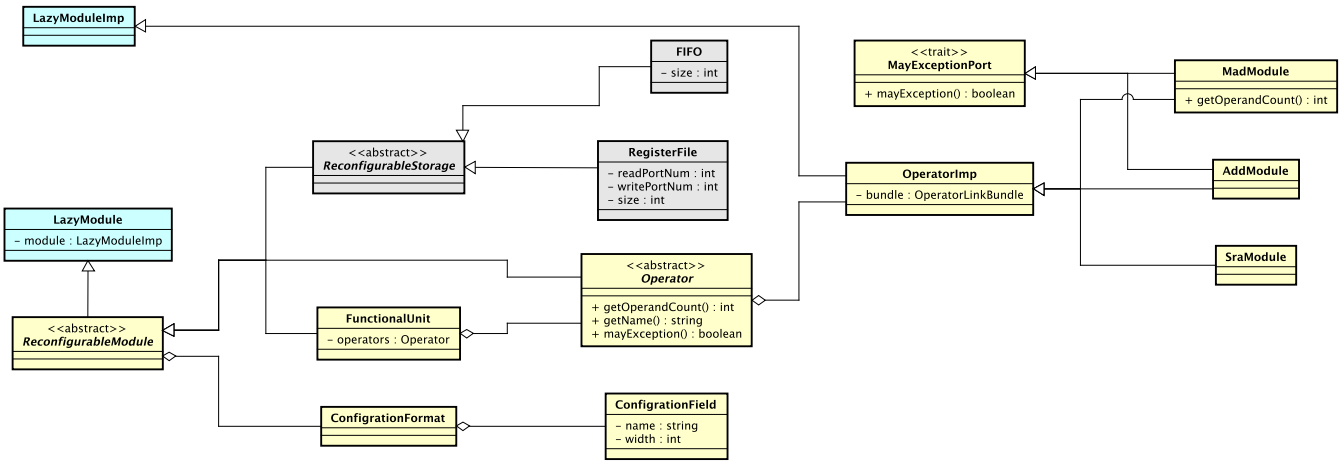

図 2: FunctionUnit に関連するクラス図

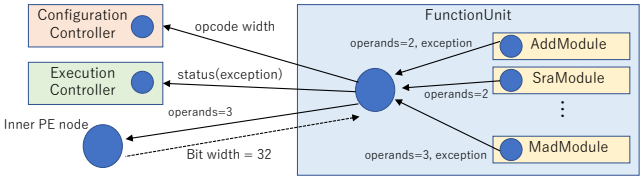

図 3: モジュール間ネゴシエーションによるパラメータ決定

```
module FunctionUnit(
  input  [3:0]  auto_sel_in_select_0,
  input  [31:0] auto_sel_in_channel_operands_0,
  input  [31:0] auto_sel_in_channel_operands_1,
  output [31:0] auto_sel_in_channel_result,
  output        auto_sel_in_channel_exception
);
```

図 4: 2 入力演算のみで構成される FunctionUnit

```
1 module AddModule(
2   input  [31:0] auto_operator_in_operands_0,
3   input  [31:0] auto_operator_in_operands_1,
4   output [31:0] auto_operator_in_result,
5   output        auto_operator_in_exception
6 );
7   wire [32:0] w_result = auto_operator_in_operands_0 + auto_operator_in_operands_1; // @
    [Operator.scala 141:48]
8   assign auto_operator_in_result = w_result[31:0]; // @[Nodes.scala 1210:84 Operator.scala
    145:21]
9   assign auto_operator_in_exception = |w_result[32]; // @[Operator.scala 143:82]
endmodule
```

図 5: 例外信号が外部から参照される構成時の AddModule

信号の決定は 5-6 行目で行われているが，この信号が他のどのモジュールからも参照されない場合は LazyModule の仕組みによって自動的に削除される．したがって，AddModule 側では上流側のモジュールでこの状態信号を必要としているかにかかわらずロジックを実装しておけばよく，if 文などが存在していないことに注意されたい．

5. 自動生成結果の確認

前節で実装したクラスに対してパラメータを変更して，生成される Verilog コードの結果を確認する．まずは，基本的な整数型の算術演算や論理演算，シフト演算など計 11 種を演算リストとして指定し，例外信号を外部から参照するモジュールが存在する構成を指定する．

この時の FunctionUnit の Verilog コードにおける I/O 宣言を図 4 に示す．想定通り演算器からの例外信号が外部へ引き出されていることがわかる．演算数は 11 なので選択には高々 4 ビットで，その幅で入力ポートが作られている．また，このケースにおける AddModule は図 5 のように生成される．符号なし加算におけるオーバーフロー検出のロジックが正しく生存している．

```
module FunctionUnit(
  input  [3:0]  auto_sel_in_select_0,
  input  [31:0] auto_sel_in_channel_operands_0,
  input  [31:0] auto_sel_in_channel_operands_1,
  input  [31:0] auto_sel_in_channel_operands_2,
  output [31:0] auto_sel_in_channel_result
);
```

図 6: 積和演算をリストに追加した FunctionUnit

```
module AddModule(
  input  [31:0] auto_operator_in_operands_0,
  input  [31:0] auto_operator_in_operands_1,
  output [31:0] auto_operator_in_result
);
  wire [32:0] w_result = auto_operator_in_operands_0 + auto_operator_in_operands_1; // @
  [Operator.scala 141:48]
  assign auto_operator_in_result = w_result[31:0]; // @[Nodes.scala 1210:84 Operator.scala
  145:21]
endmodule
```

図 7: 例外信号が外部から参照されない構成時の AddModule

次に，演算として積和演算を追加し，例外信号が外部のモジュールから参照されない構成で生成を行なう．この時の FunctionUnit の I/O 宣言と AddModule をそれぞれ図 6 と図 7 に示す PE アレイである．意図した通り，FunctionUnit の I/O でオペランド用の入力が 1 つ増え，例外用のポートが消えている．AddModule においてもそのロジックは生成されていない．これらの結果より，適切な Verilog コードが生成されていることがわかる．

6. おわりに

本稿では，多様化する CGRA の設計に柔軟に対応できる設計フレームワークの実現を目指し，Chisel と Diplomacy による実装を検討した．初期実装として PE 中の ALU に相当する FunctionUnit のクラス設計と Chisel による実装を行い，パラメータネゴシエーションによって適切な Verilog コードが生成できることを確認した．この仕組みを拡張することで，共通するサブモジュールを活用しながらさまざま再構成および制御方式の CGRA を実現できると期待される．今後は FunctionUnit 以外のモジュールについても実装を進め，異なる方式の CGRA を広域的に設計探索してより良い CGRA の構成方法の発見を行っていく．

謝　　辞

本研究は JSPS 科研費 22K17866 および JST さきがけ JPMJPR22P5 の支援を受けたものである.

文　　献

[1] J.L. Hennessy and D.A. Patterson, Computer Architecture, Sixth Edition: A Quantitative Approach, 6th edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017.

[2] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., "In-datacenter performance analysis of a tensor processing unit," Proceedings of the 44th Annual International Symposium on Computer Architecture, pp.1–12, 2017.

[3] J.-P. Fricker, "The Cerebras CS-2: Designing an AI Accelerator around the World's Largest 2.6 Trillion Transistor Chip," Proceedings of the 2022 International Symposium on Physical Design, pp.71–71, 2022.

[4] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[DL] A survey of FPGA-based neural network inference accelerators," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol.12, no.1, pp.1–26, 2019.

[5] P. Papaphilippou and W. Luk, "Accelerating Database Systems Using FPGAs: A Survey," 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pp.125–1255, 2018.

[6] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications," ACM Computing Surveys (CSUR), vol.52, no.6, pp.1–39, 2019.

[7] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)IEEE, pp.389–402 2017.

[8] A. Werner, F. Fricke, K. Shahin, F. Werner, and M. Hübner, "Automatic Toolflow for VCGRA Generation to Enable CGRA Evaluation for Arithmetic Algorithms," International Symposium on Applied Reconfigurable ComputingSpringer, pp.277–291 2019.

[9] I. Taras and J.H. Anderson, "Impact of FPGA Architecture on Area and Performance of CGRA Overlays," 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)IEEE, pp.87–95 2019.

[10] C. Tan, C. Xie, A. Li, K.J. Barker, and A. Tumeo, "OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs," 2020 IEEE 38th International Conference on Computer Design (ICCD)IEEE, pp.381–388 2020.

[11] S.A. Chin, N. Sakamoto, A. Rui, J. Zhao, J.H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A unified framework for CGRA modelling and exploration," Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference onIEEE, pp.184–189 2017.

[12] 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," DAC Design automation conference 2012IEEE, pp.1212–1221 2012.

[13] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)IEEE, pp.268–281 2020.

[14] Y. Guo and G. Luo, "Pillars: An Integrated CGRA Design Framework," Third Workshop on Open-Source EDA Technology (WOSET), 2020.

[15] Gobieski, Graham and Atli, Ahmet Oguz and Mai, Kenneth and Lucia, Brandon and Beckmann, Nathan, "Snafu: An ultra-low-power, energy-minimal cgra-generation framework and architecture," 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp.1027–1040, 2021.

[16] D. Wijerathne, Z. Li, M. Karunaratne, L.-S. Peh, and T. Mitra, "Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA," Fifth Workshop on Open-Source EDA Technology (WOSET), 2022.

[17] H. Singh, M.-H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," IEEE transactions on computers, vol.49, no.5, pp.465–481, 2000.

[18] G. Ansaloni, P. Bonzini, and L. Pozzi, "EGRA: A coarse grained reconfigurable architectural template," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.19, no.6, pp.1062–1074, 2011.

[19] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," IEEE Micro, vol.32, no.5, pp.38–51, 2012.

[20] T. Kojima, N. Ando, Y. Matshushita, H. Okuhara, N.A.V. Doan, and H. Amano, "Real chip evaluation of a low power CGRA with optimized application mapping," Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable TechnologiesACM, p.13 2018.

[21] A. Podobas, K. Sano, and S. Matsuoka, "A template-based framework for exploring coarse-grained reconfigurable architectures," 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)IEEE, pp.1–8 2020.

[22] B. Mei, F.-J. Veredas, and B. Masschelein, "Mapping an H. 264/AVC decoder onto the ADRES reconfigurable architecture," Field Programmable Logic and Applications, 2005. International Conference onIEEE, pp.622–625 2005.

[23] D. Lee, M. Jo, K. Han, and K. Choi, "FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability," Field-Programmable Technology, 2009. FPT 2009. International Conference onIEEE, pp.376–379 2009.

[24] M.A. Shami and A. Hemani, "Partially reconfigurable interconnection network for dynamically reprogrammable resource array," ASIC, 2009. ASICON'09. IEEE 8th International Conference onIEEE, pp.122–125 2009.

[25] O. Ragheb, T. Yu, R. Beidas, and J. Anderson, "Elastic Multi-Context CGRAs," 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)IEEE, pp.655–662 2022.

[26] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic cgras for irregular loop specialization," 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)IEEE, pp.412–425 2021.

[27] C. Tan, C. Xie, T. Geng, A. Marquez, A. Tumeo, K.J. Barker, and A. Li, "ARENA: Asynchronous Reconfigurable Accelerator Ring to Enable Data-Centric Parallel Computing," IEEE Transactions on Parallel and Distributed Systems, vol.32, no.12, pp.2880–2892, 2021.

[28] M. Schoeberl, Digital Design with Chisel, Kindle Direct Publishing, 2019. https://github.com/schoeberl/chisel-book

[29] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D.A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, April 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[30] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A TileLink case study," 1st Workshop on Computer Architecture Research with RISC-V, 2017.