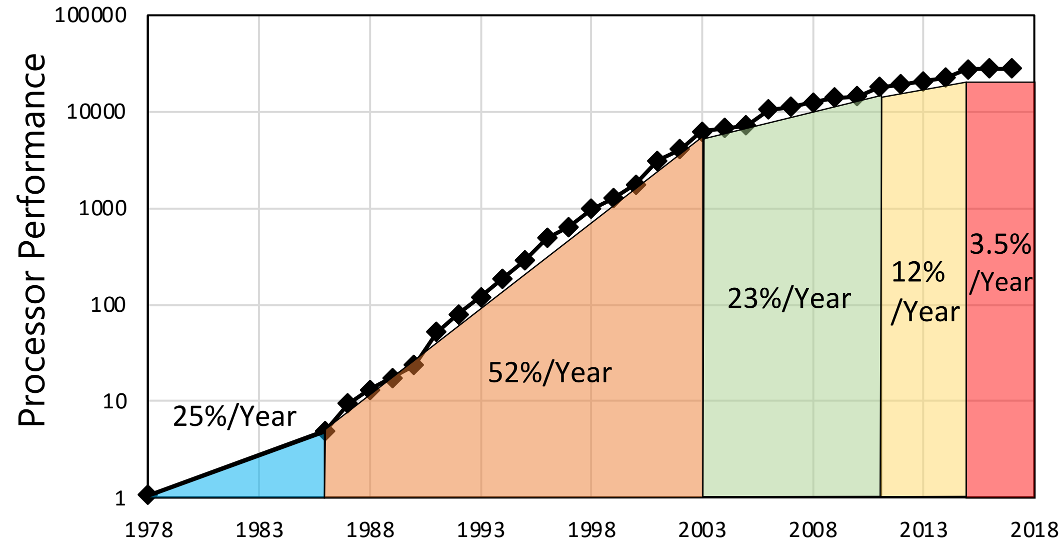
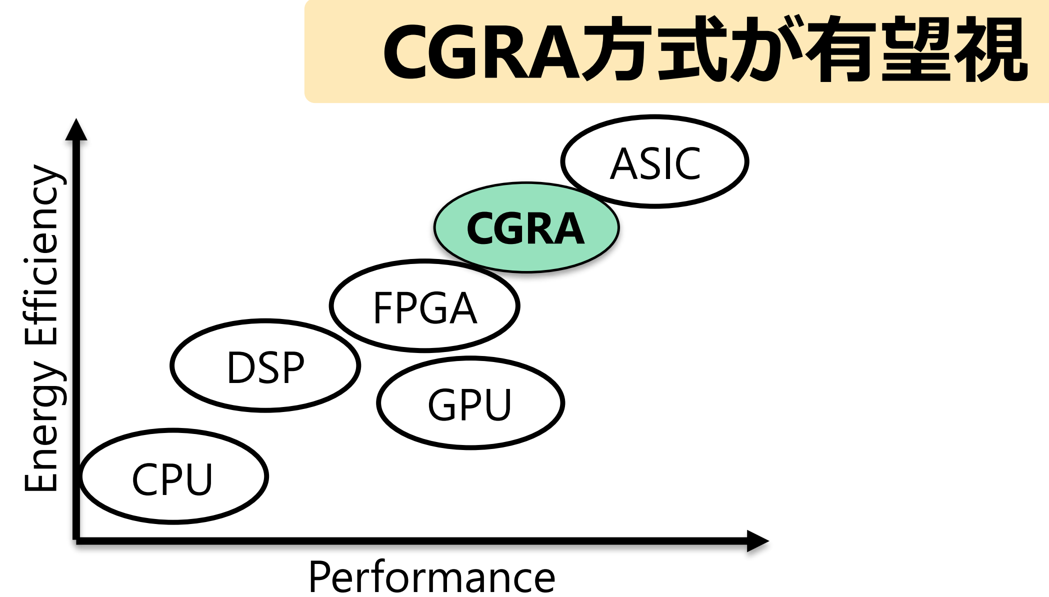
多様なCGRAを実現するDiplomacyを活用した設計手法の検討

小島　拓也†‡, 齋藤　真†, 中村　宏†
†東京大学
‡JSTさきがけ

ポストムーア時代の計算機アーキテクチャ



汎用プロセッサの性能スケーリング [1]



アーキテクチャ方式の比較 [2]

- CMOS微細化加工技術の限界が到来
  - 微細化に依存しない新たな計算機アーキテクチャの必要性
- 特にエネルギー効率の改善が急務
  - データセンターの消費電力は2030年には6倍と試算 [3]

CGRA: Coarse-grained reconfigurable architectures

- 再構成可能ハードウェアの一種
  - FPGAと比較して再構成の粒度が大きい (e.g., 32-bit)

特徴、比較 | **アプリケーション表現** | **ハードウェア構成**

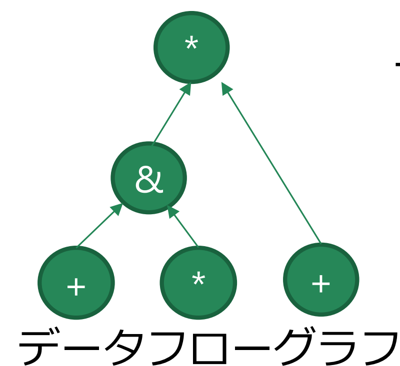
CGRA

- ワード単位の再構成
- 省面積、省電力、低遅延
- ns-usオーダーで切り替え

電力効率
77倍改善 [4]

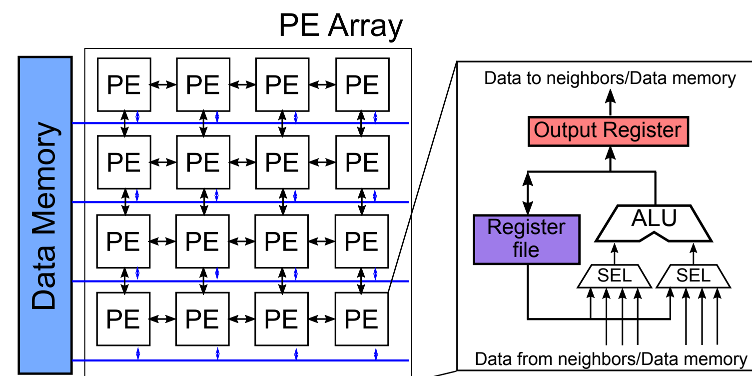
FPGA

- ビット単位の再構成
- 高いプログラマビリティ
- ms-sオーダーで切り替え

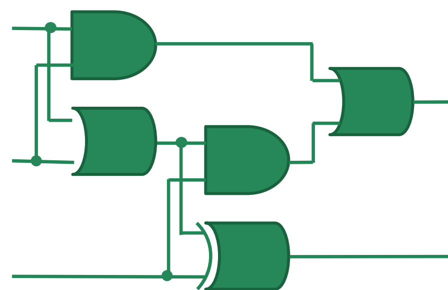


データフローグラフ

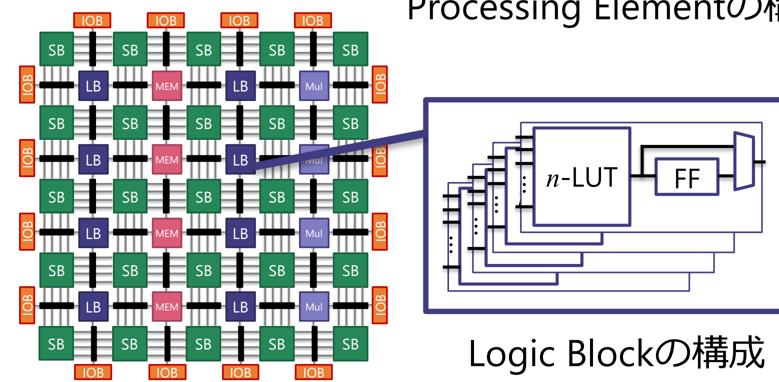
マッピング



Processing Elementの構成



ゲートレベルネットリスト



Logic Blockの構成

発表の流れ

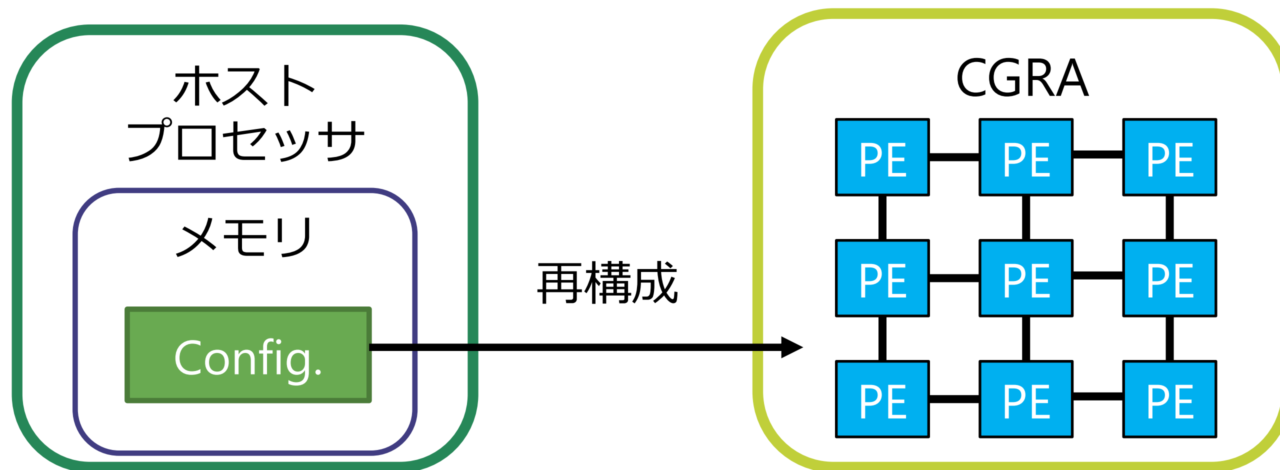
- CGRA設計における多様性
- 先行研究: 設計フレームワーク
- ChiselおよびDiplomacyによるハードウェア設計
- 提案フレームワークの初期設計
- 自動生成されるRTLの確認
- まとめ

発表の流れ

- CGRA設計における多様性
  - 再構成方式による分類
  - PEアレイの構成による分類
- 先行研究: 設計フレームワーク
- ChiselおよびDiplomacyによるハードウェア設計
- 提案フレームワークの初期設計
- 自動生成されるRTLの確認
- まとめ

多様化するCGRAの分類-その壱

- 再構成方式による分類
  - 一般にCGRAはアクセラレータとして利用
  - “再構成” = 構成情報(コンフィギュレーションデータ)の切り替え

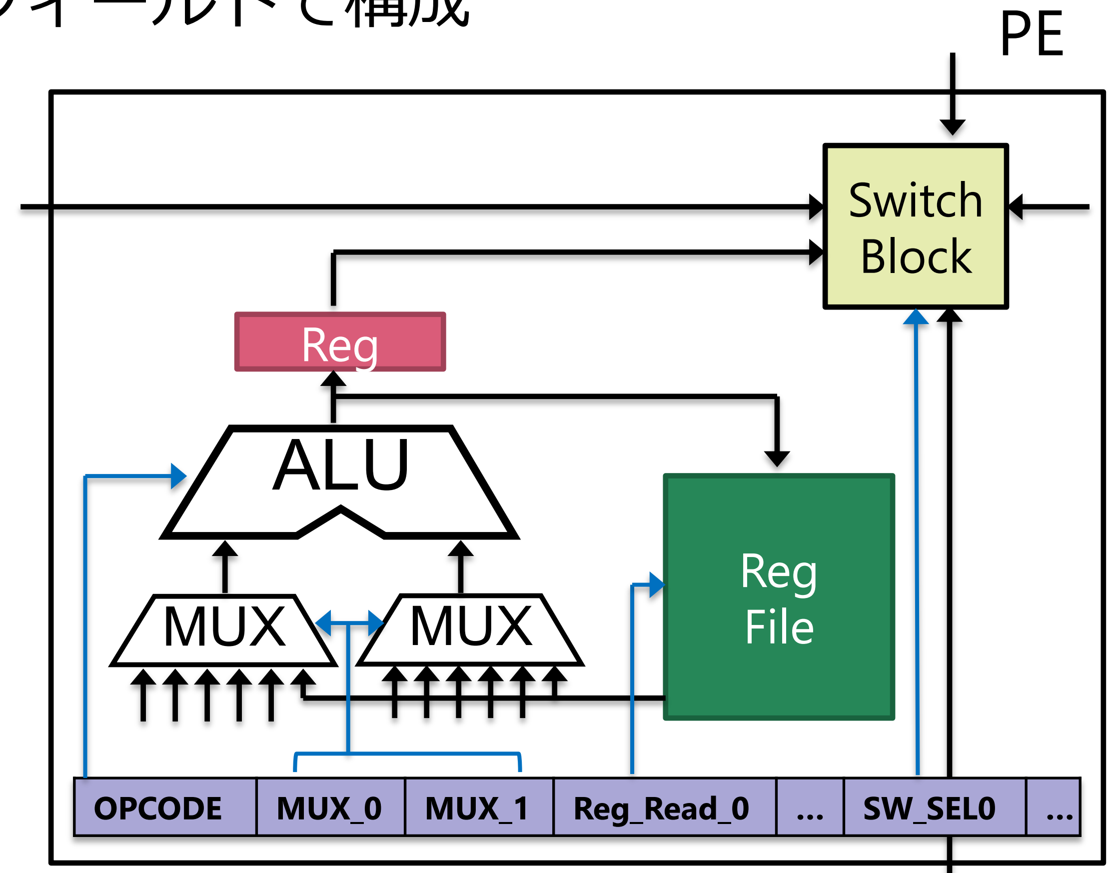


  - 再構成にホストプロセッサの介入をどの程度必要とするか

1. パッシブ方式 (集中制御型)
2. アクティブ方式 (分散制御型)

CGRAにおける構成情報

- 構成情報 (コンフィギュレーションデータ)
  - プロセッサの機械語に似たいくつかのフィールドで構成
  - 各構成モジュールの振る舞いを指定
- 主な再構成可能モジュール
  - ALU
  - マルチプレクサ
  - レジスタファイル or FIFOバッファ
  - 接続網用スイッチ
  - etc.



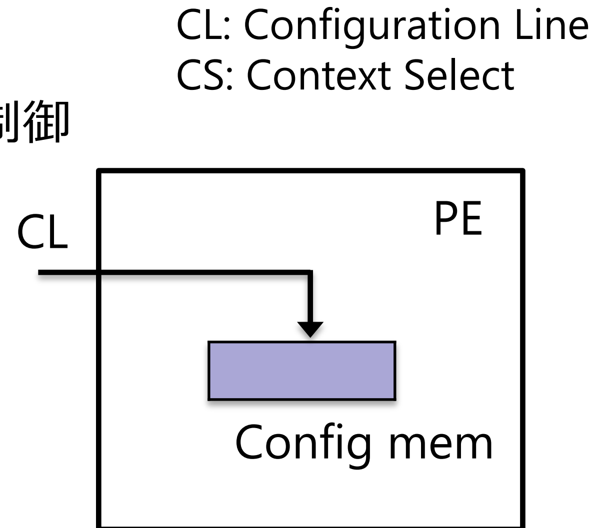
パッシブ方式 vs アクティブ方式

パッシブ方式

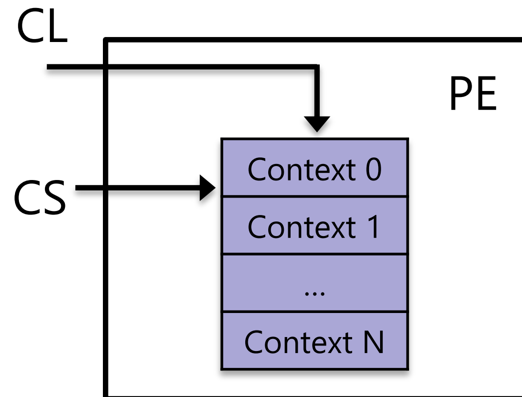
CL: Configuration Line
CS: Context Select

ホストプロセッサが集中制御

【単一構成情報のみを保持する方式】
- ■ FPGAに似た方式
- ☺ 省構成情報メモリ
- ☹ 再構成時間の増加



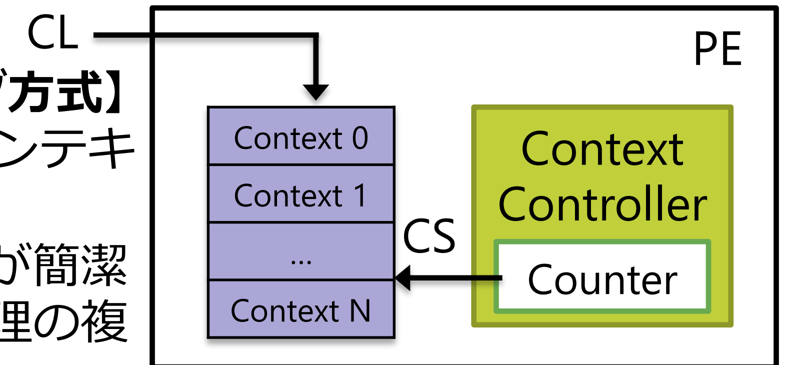
【マルチコンテキスト型】
- ■ 複数の構成情報を同時保持可能
- ☺ 構成情報の再利用
- ☺ 高速な再構成
- ☹ 構成情報メモリ増加



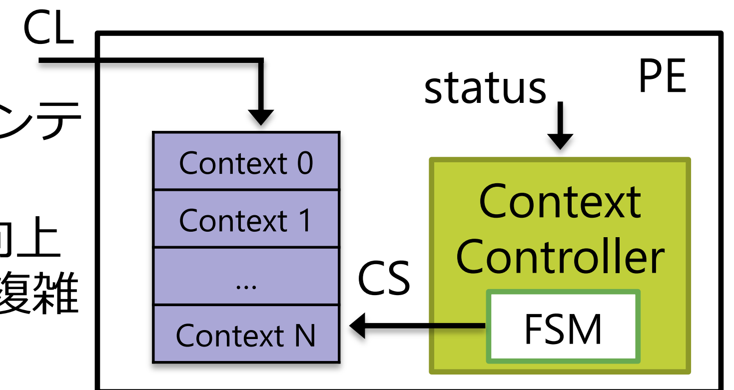
アクティブ方式

部分的な制御(コンテキスト切替)を自立的に行う

【静的スケジューリング方式】
- ■ 一定周期でコンテキストを切替
- ☺ コントローラが簡潔
- ☹ マッピング処理の複雑化



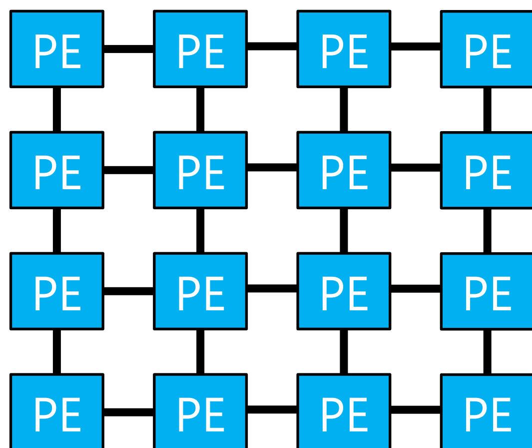
【非同期実行方式】
- ■ 各PEが独立にコンテキストを切替
- ☺ HW利用効率の向上
- ☹ コントローラが複雑化



多様化するCGRAの分類-その弐

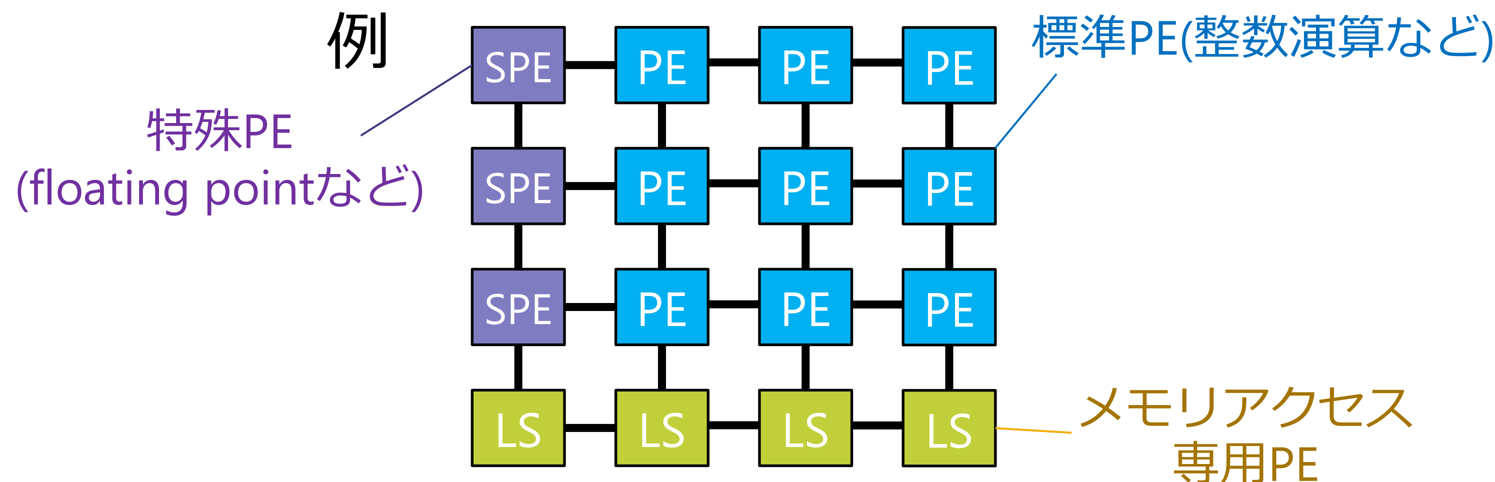
■ PEアレイの構成方法による分類

ホモジニアスPEアレイ



■ すべてのPEが同一の機能を有する
☹ 機能が過剰となる場合も
☺ マッピング処理が簡潔

ヘテロジニアスPEアレイ



■ 各PEが異なる機能を持ちうる
☺ 面積オーバーヘッドを最小化
☹ マッピング処理が複雑化

発表の流れ

- CGRA設計における多様性
- 先行研究: 設計フレームワーク
- ChiselおよびDiplomacyによるハードウェア設計
- 提案フレームワークの初期設計
- 自動生成されるRTLの確認
- まとめ

先行研究: 設計フレームワーク

- CGRAの多様化に合わせて設計探索を支援するフレームワークが多数提案
  - パラメタライズされた設計テンプレート
  - 用途に応じてパラメータを変更し、RTLを生成

フレームワーク	発表年	フロントエンド	バックエンド	抽象度	再構成方式	OSS化
CGRA-ME [5]	2017	XML	C++	低	アクティブ,静的スケジュール	✓
OpenCGRA [6]	2020	Python-based DSL	Python, PyMTL	低	アクティブ,静的スケジュール	✓
DSAGEN [7]	2020	Custom-DSL	C++	中	パッシブ, 単一コンテキスト	✓
Pillars [8]	2020	Scala-based DSL	Chisel	低	アクティブ,静的スケジュール	✓
SNAFU [9]	2021	N/A	SystemVerilog	高	パッシブ,マルチコンテキスト	
Riken CGRA [10]	2022	N/A	SystemVerilog	高	パッシブ、単一コンテキスト	
Morpher [11]	2022	JSON like	Chisel	中	アクティブ,静的スケジュール	✓

先行研究: 設計フレームワーク

- CGRAの多様化に合わせて設計探索を支援するフレームワークが多数提案
  - パラメタライズされた設計テンプレート
  - 用途に応じてパラメータを変更し、RTLを生成

フレームワーク	発表年	フロントエンド	バックエンド	抽象度	再構成方式	OSS化
CGRA-ME [5]	2017	XML	C++	低	アクティブ,静的スケジュール	✓
OpenCGRA [6]	2020	Python-based DSL	Python, PyMTL	低	アクティブ,静的スケジュール	✓
DSAGEN [7]	2020	Custom-DSL	C++	中	パッシブ, 単一コンテキスト	✓
Pillars [8]	2020	Scala-based DSL	Chisel	低	アクティブ,静的スケジュール	✓
SNAFU [9]	2021	N/A	SystemVerilog	高	パッシブ,マルチコンテキスト	
Riken CGRA [10]	2022	N/A	SystemVerilog	高	パッシブ、単一コンテキスト	
Morpher [11]	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ベースのDSL	Chisel+Diplomacy	高	**Any**	✓

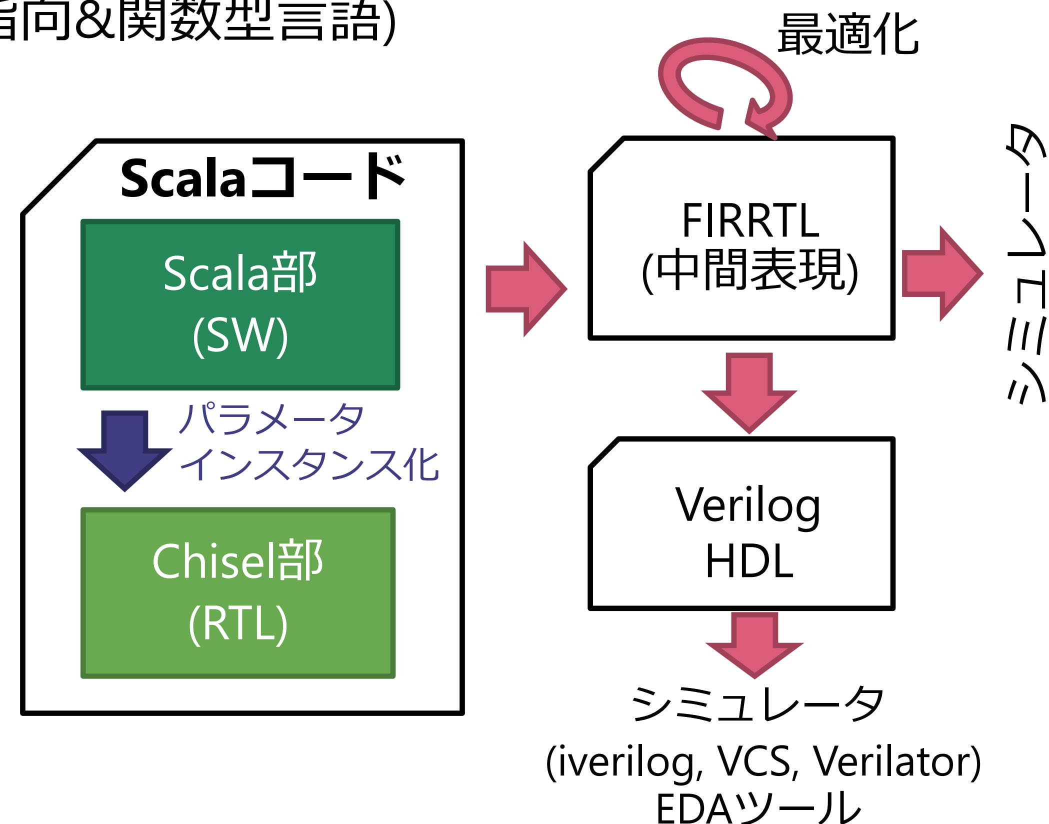
課題: 各方式における再利用性を活かしたスケーラブルな実装, 異なるモジュール間接続の取り扱い

発表の流れ

- CGRA設計における多様性
- 先行研究: 設計フレームワーク
- ChiselおよびDiplomacyによるハードウェア設計
  - Chisel
  - Diplomacyの必要性
- 提案フレームワークの初期設計
- 自動生成されるRTLの確認
- まとめ

Chisel[12]によるディジタル回路設計

- Scalaベースのハードウェア記述言語
  - ScalaおよびJavaのライブラリが利用できる
  - Scala言語のスケーラビリティを踏襲 (オブジェクト指向&関数型言語)
- オープンソース
- 複雑にパラメタライズされた設計が可能
  - 合成可能なVerilogを生成
- Chisel標準ライブラリ
  - One-Hotエンコーダ, カウンタ, アービタ, FIFO, etc.
- 代表的な利用例
  - Google TPU, SiFive Freedom, Apache TVM VTA



Chiselの例-ナイーブなALUの実装

- Verilogライクに記述した例

```
 1 object NaiveALU {
 2   val ALU_THA = 0.U
 3   val ALU_THB = 1.U
 4   val ALU_AND = 2.U
 5   val ALU_OR = 3.U
 6   val ALU_NOT = 4.U
 7   val ALU_XOR = 5.U
 8   val ALU_ADD = 6.U
 9   val ALU_SUB = 7.U
10 }
```

定数宣言
ヘッダファイルのdefine文に相当

```
 1 class NaiveALU(dataWidth : Int, selWidth : Int) extends Module {
 2   val io = IO(new Bundle {
 3     val a = Input(UInt(dataWidth.W))
 4     val b = Input(UInt(dataWidth.W))
 5     val s = Input(UInt(selWidth.W))
 6     val y = Output(UInt(dataWidth.W))
 7   })
 8   val a = io.a
 9   val b = io.b
10   val y = io.y
11   import NaiveALU._
12   y := 0.U
13   switch (io.s) {
14     is(ALU_THA) { y := a }
15     is(ALU_THB) { y := b }
16     is(ALU_AND) { y := a & b }
17     is(ALU_OR)  { y := a | b }
18     is(ALU_NOT)  { y:= ~a }
19     is(ALU_XOR)  { y:= a ^ b }
20     is(ALU_ADD) { y:= a + b }
21     is(ALU_SUB) { y:= a - b }
22   }
23 }
```

Chiselの例 – パラメタライズされたALU

インスタンス化
(ファクトリーメソッドは省略)

```
1 ALU(32) //全ての演算
2 ALU(List("ADD", "AND"), 32) //ADD,SUBのみのALU
```

```
1 object ALU {
2   val OpFuncsMap = ListMap[String, (UInt, UInt) => UInt](
3       "THA" -> ((a, _) => a),
4       "THB" -> ((_, b) => b),
5       "AND" -> (_ & _),
6       "OR" -> (_ | _),
7       "NOT" -> ((a, _) => ~a),
8       "XOR" -> (_ ^ _),
9       "ADD" -> (_ + _),
10      "SUB" -> (_ - _)
11   )
12 }
```

Opcodeに対応する
ロジックの連想配列

- 高い抽象度
  - 文字列のopcodeリストを渡して実体を作る
  - 可読性の悪いif-defは一切不要
  - 合成と継承を駆使すればもっとスケーラブルに

```
1 class ALU(opList : List[String], dataWidth : Int) extends Module {
2   val io = IO(new Bundle {
3     val a = Input(UInt(dataWidth.W))
4     val b = Input(UInt(dataWidth.W))
5     val s = Input(UInt(log2Ceil(opList.size).W))
6     val y = Output(UInt(dataWidth.W))
7   })
8   val a = io.a
9   val b = io.b
10  io.y := MuxCase(0.U, opList.map(ALU.OpFuncsMap(_)).zipWithIndex.map {
11      case (func, opcode) => (io.s === opcode.U) -> func(io.a, io.b)
12  })
13 }
```

Opcodeのリスト
を受け取りモ
ジュールを作る

Chiselにおけるパラメータ化の限界

- トップダウン的にパラメータを指定する必要性
  - インスタンス化された時点で設計がFix
  - パラメータを後から生成されるモジュールに依存できない
- スケーラビリティに欠ける
  - パターン(e.g., 再構成方式)ごとに微妙に異なる実装をし直すことに

```
1 class Sub extends Module {...}
2 class Main extends Module {
3     val sub0 = Module(new Sub())
4     val sub1 = Module(new Sub())
5 }
```

←この時点でsub0の設計が確定

Chiselにおけるパラメータ化の限界

- トップダウン的にパラメータを指定する必要性
  - インスタンス化された時点で設計がFix
  - パラメータを後から生成されるモジュールに依存できない
- スケーラビリティに欠ける
  - パターン(e.g., 再構成方式)ごとに微妙に異なる実装をし直すことに

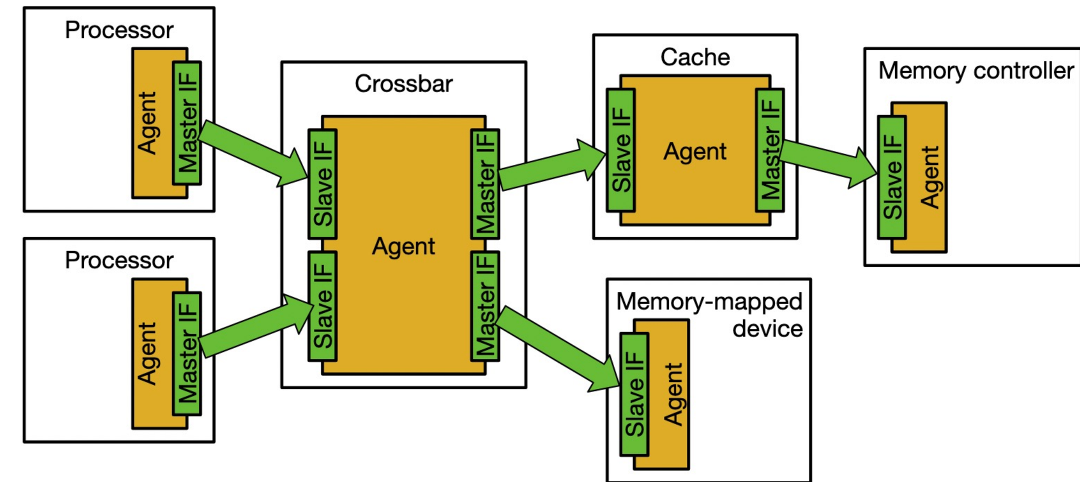
```
1 class Sub extends Module {...}
2 class Main extends Module {
3     val sub0 = Module(new Sub())
4     val sub1 = Module(new Sub())
5 }
```

←この時点でsub0の設計が確定

解決策: *Diplomacy*とLazyModuleの利用

Diplomacyによるパラメータネゴシエーション[13]

- Chiselの拡張機能
  - Rocket Chip Generator[14]に含まれる
  - 利用例) TileLink (RISC-V SoCのバス)
- モジュールは仮想的なノードを保有
- ハードウェア生成が2ステップ



TileLink[]設計時に構築されるグラフの例
([13]より借用)

STEP 1

- モジュール間接続のグラフ構築
- ノード間でパラメータの交換

STEP 2

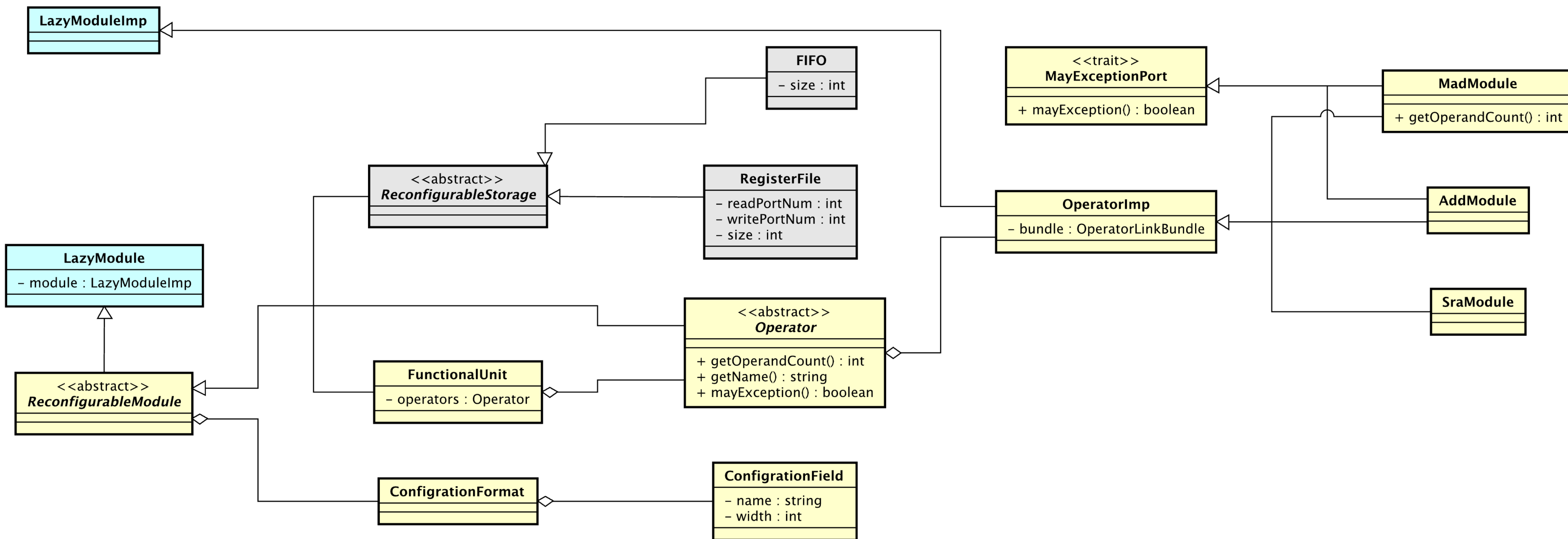
- ネゴシエーション後のパラメータを用いて
  モジュールのインスタンス化
- インスタンス化を遅延させる**LazyModule**を利用

発表の流れ

- CGRA設計における多様性
- 先行研究: 設計フレームワーク
- ChiselおよびDiplomacyによるハードウェア設計
- 提案フレームワークの初期設計
  - クラス設計
  - パラメータネゴシエーションの実装
- 自動生成されるRTLの確認
- まとめ

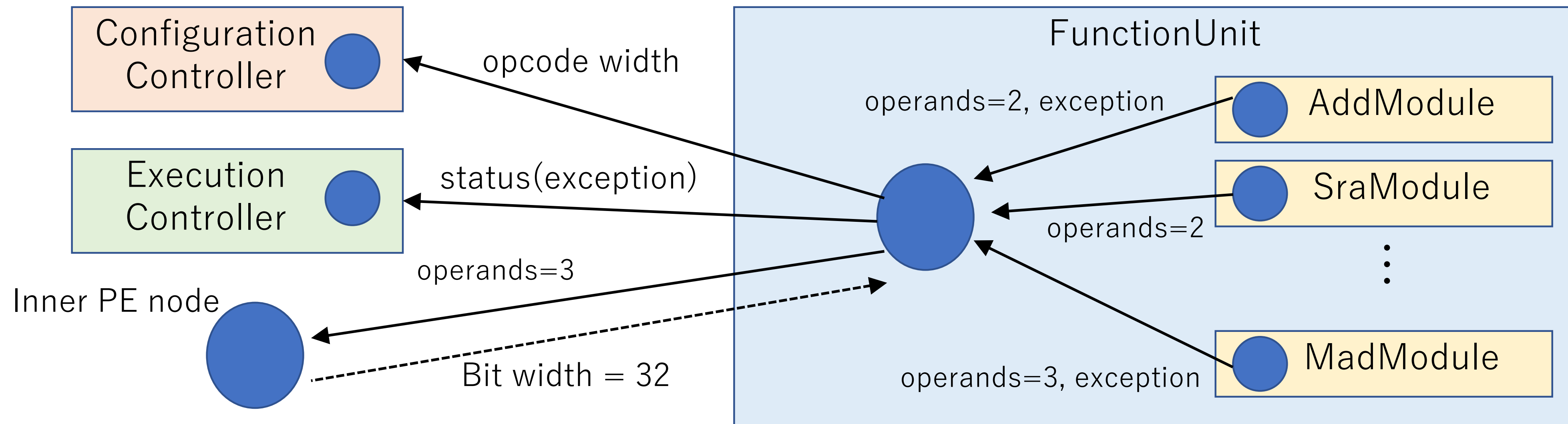
PE内モジュールのクラス設計

- 初期検討としてPE内のモジュールのクラス設計を実施
  - 演算ユニットFunctionUnit (ALU)に関連するモジュール(黄色)を今回は実装



パラメータネゴシエーションの実装

- FunctionUnitに関連するパラメータ
  - どのような演算をサポートするか (ユーザー指定)
  - データ幅: PE内ノードから伝播
  - オペランド数: 内包する演算器中で最大のもの
  - オプコードの幅: 演算種から決定
  - 内部ステータス用のポート(e.g., exception)の有無
    - 例外を起こしうる演算器があるか、コントローラがその情報を必要とするか



コード例) 加算器 AddModule

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

コード例) 加算器 AddModule

- 演算器の抽象クラス**Operator**を継承
- 例外を起こしうるモジュールの共通機能 **MayExceptionModule**トレイトを合成

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

コード例) 加算器 AddModule

- STEP2で評価されるパート
- 決定したパラメータをもとに設計を行う

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

コード例) 加算器 AddModule

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

- 符号なし整数のオーバーフローを検出
- 外部モジュールがexceptionのポートと接続されるかどうかの条件分岐は不要

発表の流れ

- CGRA設計における多様性
- 先行研究: 設計フレームワーク
- ChiselおよびDiplomacyによるハードウェア設計
- 提案フレームワークの初期設計
- 自動生成されるRTLの確認
- まとめ

ケース1 – 2項演算のみを有する

- 構成
  - FunctionUnitは2項演算の演算器を計11種を保有
  - 外部のコントローラから例外状態を観測される構成

```
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
);
```

生成されたVerilogコードのIO部 (FunctionUnit)

ケース1 – 2項演算のみを有する

- 構成
  - FunctionUnitは2項演算の演算器を計11種を保有
  - 外部のコントローラから例外状態を観測される構成

```
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
);
```

- Select信号の配線幅を自動決定

生成されたVerilogコードのIO部 (FunctionUnit)

ケース1 – 2項演算のみを有する

- 構成
  - FunctionUnitは2項演算の演算器を計11種を保有
  - 外部のコントローラから例外状態を観測される構成

```
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
);
```

- 例外発生通知用のポートが生成

生成されたVerilogコードのIO部 (FunctionUnit)

ケース1 – 2項演算のみを有する

- FunctionUnit
  - 2項演算の演算器を計11保有
  - 外部のコントローラから例外状態を観測される構成

```
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,
  output        auto_operator_in_exception
);
  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]
  assign auto_operator_in_exception = |w_result[32]; // @[Operator.scala 143:82]
endmodule
```

- FunctionUnit側へ引き伸ばすためのポート

- オーバーフロー検出のロジック

生成されたVerilogコード (AddModule)

ケース2 – 積和演算器を追加

- 先ほどの構成に積和演算器を追加
- 例外の観測が不要となる構成へ変更 (あくまで例として)

```
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
);
```

生成されたVerilogコードのIO部 (FunctionUnit)

ケース2 – 積和演算器を追加

- 先ほどの構成に積和演算器を追加
- 例外の観測が不要となる構成へ変更 (あくまで例として)

```
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
);
```

- 第3オペランド用のポートが出現

生成されたVerilogコードのIO部 (FunctionUnit)

ケース2 – 積和演算器を追加

- 先ほどの構成に積和演算器を追加
- 例外の観測が不要となる構成へ変更 (あくまで例として)

```
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
);
```

- 例外通知用ポートは削除

生成されたVerilogコードのIO部 (FunctionUnit)

ケース2 – 積和演算器を追加

- 先ほどの構成に積和演算器を追加
- 例外の観測が不要となる構成へ変更 (あくまで例として)

```
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
```

- 例外通知用ポートおよび
  オーバーフロー検出ロジックの消失

生成されたVerilogコード (AddModule)

発表の流れ

- CGRA設計における多様性
- 先行研究: 設計フレームワーク
- ChiselおよびDiplomacyによるハードウェア設計
- 提案フレームワークの初期設計
- 自動生成されるRTLの確認
- ➢ まとめ

まとめ

- ポストムーア時代のアーキテクチャとしてCGRAが有望
- CGRAは用途に応じて様々な方式
  - 設計パラメータも多岐にわたる
- 本研究では多様化するCGRAを単一のフレームワークから生成可能にすることを目標
- Chisel+Diplomacyを用いた設計の検討を実施
- 演算ユニットを例に柔軟に生成されるRTLを変更できることを確認
- 今後の展望:
他のモジュールやそのためのパラメータネゴシエーションプロトコルの実装

参考文献

- [1] Hennessy, John L., and David A. Patterson. "A new golden age for computer architecture." Communications of the ACM 62.2 (2019): 48-60.
- [2] Liu, Leibo, et al. "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications." ACM Computing Surveys (CSUR) 52.6 (2019): 1-39.
- [3]国立研究開発法人科学技術振興機構 低炭素社会戦略センター (LCS), "情報化社会の進展がエネルギー消費に与える影響（Vol.2）―データセンター消費エネルギーの現状と将来予測および技術的課題―", https://www.jst.go.jp/lcs/proposals/fy2020-pp-03.html (2022年11月25日参照)
- [4] Prabhakar, Raghu, et al. "Plasticine: A reconfigurable architecture for parallel patterns." 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.
- [5] Anderson, Jason, et al. "CGRA-ME: An Open-Source Framework for CGRA Architecture and CAD Research." 2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE, 2021.
- [6] Tan, Cheng, et al. "OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs." 2020 IEEE 38th International Conference on Computer Design (ICCD). IEEE, 2020.

参考文献

- [7] Weng, Jian, et al. "Dsagen: Synthesizing programmable spatial accelerators." 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.
- [8] Guo, Yijiang, and Guojie Luo. "Pillars: An Integrated CGRA Design Framework." Third Workshop on Open-Source EDA Technology (WOSET). 2020.m
- [9] Gobieski, Graham, et al. "Snafu: an ultra-low-power, energy-minimal CGRA-generation framework and architecture." 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021.
- [10] Adhi, Boma, et al. "Exploration Framework for Synthesizable CGRAs Targeting HPC: Initial Design and Evaluation." 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2022.
- [11] D. Wijerathne, Z. Li, M. Karunaratne, L.-S. Peh, and T. Mitra, "Morpher: An Open-Source Integrated Compilation and Simu- lation Framework for CGRA," Fifth Workshop on Open-Source EDA Technology (WOSET), 2022.
- [12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviˇzienis, J. Wawrzynek, and K. Asanovi ´c, "Chisel: construct- ing hardware in a scala embedded language," DAC Design au- tomation conference 2012IEEE, pp.1212–1221 2012.

参考文献

- [13] H. Cook, W. Terpstra, and Y. Lee, “Diplomatic design patterns: A TileLink case study,” 1st Workshop on Computer Architec- ture Research with RISC-V, 2017.
- [14] K. Asanovi ´c, 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 gen- erator,” Technical Report UCB/EECS-2016-17, EECS De- partment, University of California, Berkeley, April 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016- 17.html