Vol.2019-ARC-235 No.31
Vol.2019-SLDM-187 No.31
Vol.2019-EMB-50 No.31
2019/3/18

関数型言語Elixirによるハードウェア設計環境の実現に向けたライブラリ関数のIPモジュール設計

松井 健太郎[1,a)]　上野 嘉大[2]　森 正和[3]　山崎 進[4]　高瀬 英希[1,5,b)]

概要：2012年に登場した関数型言語であるElixirは，言語仕様や概念がシンプルであるが応用が効くように設計されており，並列処理のプログラミングが容易に実現できるという特徴がある．我々は，Elixirを設計言語とするデータフロー型ハードウェアの設計環境の構想を検討している．本論文では，設計環境の実現に向けて，Elixirにおけるデータ並列処理のためのライブラリである`Enum`および`Flow`の各関数を，それらと機能等価なIPモジュールとして設計することに取り組む．加えて，Elixirのアプリケーションからハードウェア回路を透過的に駆動するための通信インタフェース機構を設計する．Xilinx社プログラマブルSoCの搭載された評価ボード上において本研究の成果物を実装し，それらの評価および動作検証を実施した．これにより，Elixirによるハードウェア設計環境の実現に向けた適用可能性および技術課題を明らかにした．

1. はじめに

定められた計算資源のもとで要求される性能を発揮することは，コンピュータシステム設計における主要な目標のひとつである．プロセッサの処理性能はムーアの法則に従って大きく発展してきたが，その向上は鈍化の兆しを見せ始めている．これを解決する方策のひとつとしてマルチプロセッサ構成の採用が挙げられる．しかし，本構成において性能向上を実現するためには，並列処理に対応できるように明示的にソフトウェアを設計する必要がある．また，求められる計算を全てのコアが協調して実行できるように，各コアに処理を割り当てる必要がある．しかし，現状ではこれを実現するプログラミング言語ならびにライブラリの整備は十分であるとはいえない．

このような現状を打破するため，近年ではコンピュータシステム設計におけるFPGAの活用が進んでいる．FPGAは再構成可能なハードウェアデバイスであり，任意の論理回路を自由に何度でも構成できる．プロセッサと比較した利点として，性能および省電力性に優れることが挙げられる．また，特定用途向けの専用ハードウェア回路であるASICと比較した場合でも，製造コストおよび設計柔軟性に優れている．FPGAを用いたシステム設計の課題としては，Verilog HDLやVHDLといったハードウェア記述言語（HDL: Hardware Description Language）によって，レジスタ転送レベル（RTL: Register Transfer Level）で回路の振る舞いを設計する必要があることが挙げられる．

FPGAを活用するシステム設計の生産性を向上させる方策として，高位合成技術が期待されている．高位合成とは，汎用プログラミング言語などの抽象度の高い動作記述からRTLを生成する技術のことを指す．ハードウェア設計の抽象度が高くなるため，設計の生産性と再利用性の向上に貢献する．高位合成ツールによるハードウェア設計では，ソフトウェア向けのプログラミング言語によってハードウェアの構造と振る舞いに注力して定義する．ただし，タイミング設計の最適化やモジュール並列化については，ツールが提供する合成オプションを適切に設定したり，言語仕様の拡張によって提供される最適化指示子を多用する必要がある．このため，ハードウェアの並列化による処理性能の向上を実現するためには，従来のソフトウェア向けプログラミング技術だけではなく，ハードウェア固有の知識を要求されることになる．

高水準言語からハードウェア回路を設計する場合，設計生産性の観点からは，習熟容易な言語を用いることが望ましいと考える．これは，開発者にとっては設計開発の移行に掛かる学習コストを抑えることに繋がる．さらに，並列処理について標準的な規定がある言語であることが望ましい．このような背景から，我々は，関数型言語Elixir[1]

[1] 京都大学
[2] デライトシステムズ
[3] カラビナテクノロジー
[4] 北九州市立大学
[5] JST さきがけ
[a)] emb@lab3.kuis.kyoto-u.ac.jp
[b)] takase@i.kyoto-u.ac.jp

の仕様をハードウェア設計の記述言語として活用することに着目する．Elixir におけるパイプライン演算子|>とMapReduce モデルに基づく並列処理のプログラミングモデルは，データフロー型のハードウェアアーキテクチャと親和性が高いものであると考える．Elixir によるハードウェア設計環境は，Cockatrice と名付けて検討を進めている．Elixir 記述を入力とするハードウェア設計環境を実現することで，Elixir アプリのコア処理における設計生産性および処理性能の向上が期待できる．

本研究では，Cockatrice を実現するための足掛かりとして，Elixir のライブラリ関数を機能等価な IP モジュールとして設計することに取り組む．設計対象は，Elixir のデータ並列プログラミングで頻用される `Enum` および `Flow` のライブラリ関数である．この際に，機能等価なハードウェア回路の最適化についても検討を行う．`Enum` ライブラリには，入力データ・コレクションの各要素に対して順次処理を行うものが存在する．このような処理について依存関係を分析し，可能であれば並列実行可能な回路として設計する．これらの関数を IP モジュールとして設計することで，パイプライン演算子を用いた Elixir の記述からハードウェア回路を合成できるようになることが期待される．

さらに本研究では，ハードウェア回路に対して Elixir アプリからデータ通信と実行制御を行うインタフェース機構の設計にも取り組む．SW/HW 間の通信方式については，Xilinx 社プログラマブル SoC で広く採用されている AXI（Advanced eXtensible Interface）プロトコルを採用して実装を行う．Elixir アプリ上からの FPGA 上のハードウェア回路の制御については，他言語で実装されたネイティブコードを Erlang/OTP を介して呼び出す NIF（Native Implemented Function）機能を用いて，C 言語実装のデバイスドライバを実行する形で実装する．これにより，ユーザは Elixir アプリ上から通常の関数呼び出しの形でハードウェア回路を駆動することが可能となる．

本論文の構成は次の通りである．2 章では，関数型言語 Elixir について解説する．3 章では，我々が検討を進めている設計環境 Cockatrice の基本的なアイデアを示し，これを実現するための技術要素を整理する．4 章では，Elixir のライブラリ関数と機能等価な IP モジュールの設計について述べる．5 章では，SW/HW 間の通信インタフェース機構の設計について述べる．6 章において本研究における実装成果の予備評価を示し，最後に 7 章で本論文のまとめと今後の展望を述べる．

2. 関数型言語 Elixir

Elixir は，2012 年に登場した Erlang VM 上で動作する関数型言語である．Elixir の特徴として，まず，開発生産性の高さが挙げられる．従来の関数型言語に比べて言語仕様や概念がシンプルであるが応用が効くように設計されており，処理の振る舞いではなくデータの扱いを直接的に操作するためのライブラリや記法が豊富に整備されている．図 1 に示す Elixir コードの例のように，パイプライン演算子|>と MapReduce モデル [2] に基づく Flow ライブラリ [3] によって，データを加工しながら計算が進行する．本コードでは，入力として 1 行目の `input_list` である数値のリストを受け取る．次に，各要素に対して 3 行目および 4 行目の関数 `foo` および `bar` の処理を適用する．これらの関数は，2 行目の `Flow` ライブラリおよび `Flow.map` 関数の記述によって 4 並列化される．各関数の処理後の要素は，4 行目の `Enum` ライブラリで提供されるリスト化処理の関数 `Enum.to_list` および 5 行目の昇順ソートの関数 `Enum.sort` によって，昇順ソートされたリストとして返される．それぞれの処理はパイプライン演算子|>によって順次加工されながら進行していく．このように Elixir は並列処理のプログラミングが容易に実現できる記述性を備えており，結果として生産性が向上する．

Elixir では，全ての変数はイミュータブルであるという特徴を備える．通常の言語ではマルチコアによる並列処理を行なってもコア間の同期や排他制御などが多々発生するため，コア数の増加に対し実行速度の向上が鈍化する傾向がある．しかし，Elixir ではイミュータブル特性により，マルチコアによる並列処理の際にはコア数に比例して実行速度が向上する [4]．そして，Elixir ではプロセス単位でガベッジコレクションを含む堅牢なメモリ管理がなされるという特徴があり，加えてレスポンス性が高く再起動を高速に行える．このため Elixir では，try/catch のような例外処理ではなく，障害が起こったらプロセスは再起動し，外部で動作している障害監視プロセスで例外処理を行うことが多い．これにより，例外処理の記述が簡素化でき，耐障害性の高いシステムを構築できる．

Elixir で実装されたウェブフレームワークである Phoenix[5] は，Ruby における Ruby on Rails と同等以上の生産性を誇りながら，レスポンス性が極めて高い．[6] によれば，Phoenix によって構築されたサーバは Ruby on Rails のそれよりも 10.63 倍のスループットを達成できたという評価結果が報告されている．このため，データベースをもとに複雑な計算をしてグラフィカルな表示をするようなウェブサイトを構築したときに，大量のアクセスに応えることができる．特に文献 [4] では，Elixir/Phoenix 性能向上に対する貢献が示されている．

Elixir/Phoenix に対する我々の取り組みとしては，まず文献 [7] において，Elixir/Phoenix の IoT 分野における可能性に着目し，IoT ボード上における基礎性能を評価した．ベンチマーク評価によって得られた基礎性能より，IoT システム開発における Elixir の有用性を定量的に議論した．文献 [8] では，Node プログラミングモデルに基づく軽量コールバックスレッド方式の並行プログラミング機構の

Vol.2019-ARC-235 No.31
Vol.2019-SLDM-187 No.31
Vol.2019-EMB-50 No.31
2019/3/18

```
input_list                                  # リストの入力を受け取る
  |> Flow.from_enumerable(stages: 4)        # 4並列のプロセスを生成しデータを分配する
  |> Flow.map(foo)                          # 分配されたデータを関数fooに渡す
  |> Flow.map(bar)                          # fooの結果をさらに関数barに渡す
  |> Enum.to_list                           # 各プロセスの出力をリストとしてマージする
  |> Enum.sort                              # リストのデータをソートする
```

図 1 Elixir コードの例

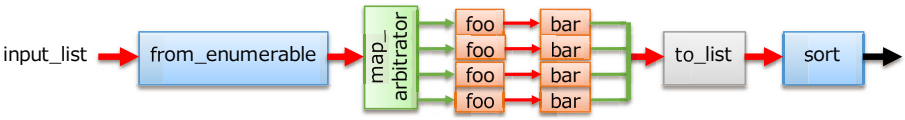

図 2 図 1 の Elixir コードから合成されるハードウェアの例

Eixir 実装を報告した．本方式はマルチプロセス方式やマルチスレッド方式に比べてプロセスの省メモリ性能を達成することができ，特にウェブサーバの同時セッション最大数やレイテンシを大幅に改善できる．さらに文献 [10] において，Elixir アプリにおいて GPGPU を活用できる超並列高速実行処理系である Hastega を提案した．Hastega のプロトタイプ実装は，Python の CuPy と比較して 3 倍以上の速度向上を達成できることを示した．

3. 設計環境の基本的なアイデア

パイプライン演算子 `|>` と `Enum` および `Flow` のライブラリで記述される Elixir の並列プログラミングモデルは，FPGA におけるデータフロー型のハードウェア設計との親和性が高いと考える．本章では，我々が検討を進めている設計環境 Cockatrice について，その基本アイデアを示す．

図 2 は，図 1 のコード片から合成されるデータフロー型ハードウェアのイメージを示している．まず，2 行目の `Flow.flow_enumerable` 関数で記述された引数 `stages:4` は，3 行目および 4 行目の `map` 関数に相当するハードウェアモジュール `foo` および `bar` が 4 並列に並ぶことを意味すると解釈できる．つまり，入力リストからデータを逐次取り出して各モジュールに分配するハードウェア，および，map 関数に相当する 4 個の並列なハードウェアモジュールを用意すればよいと解釈できる．その後段には，5 行目の `Enum.to_list` 関数および 6 行目の `Enum.sort` 関数に相当するハードウェアモジュールを配置すればよい．最後に，パイプライン演算子 `|>` による記述の順序に従って各モジュールの入出力を接続していき，データフロー型のハードウェア回路として合成していけばよい．このことから，Elixir を設計記述言語として活用するハードウェア設計環境が実現できることが考えられる．

我々が検討している設計環境では，IP ベース設計のアプローチを取る．設計検証済みの機能ブロックやハードウェアモジュールである IP（Intellectual Property）は，ソフトウェアにおけるライブラリに相当するとみなせる．本設計環境では，Elixir ライブラリコレクションに含まれる各関数を機能等価な IP モジュール・ライブラリ提供する．これらの IP モジュールを指定された情報に応じて構成し，パイプライン演算子の記述に応じてデータフローハードウェアとしてマッピングしていく．

設計環境 Cockatrice の実現に向けて必要となる研究課題および技術要素として，下記が挙げられる．

(1) Elixir による設計記述の解析による，使用するライブラリ関数のパラメータや並列度，および，パイプライン演算子によるデータフローの抽出
(2) Elixir のライブラリ関数に機能等価となるハードウェア IP モジュールの設計
(3) 抽出されたデータフローに基づく IP モジュール間のデータパス生成
(4) Elixir アプリからハードウェア回路を駆動するためのインタフェース機構の生成
(5) 合成ツールによるハードウェア回路の論理合成

本研究では，(2) および (4) の項目に重点的に取り組む．

4. IP モジュールの設計

本章では，Elixir のライブラリ関数をそれと機能等価なハードウェアの IP モジュールとして設計する．設計対象は，Elixir のデータプログラミングにおいて頻用される `Enum` ライブラリである．本ライブラリは，リストなどのコレクションを列挙していくために用いる一連のアルゴリズムであり，およそ 70 個以上の関数を含んでいる．これらの関数のうち，数値演算において特に有用と思われるものを選定し，表 1 に示す関数を対象として実装を行った．

4.1 Enum ライブラリ関数の処理パターンの分類

本節では `Enum` ライブラリ関数の機能仕様に着目し，それぞれの処理パターンの分類を整理する．

ここで，コレクションの要素に適用する関数を要素関数と呼ぶ．要素関数のうち，1 つの要素を引数として単一の結果を出力する処理を独立処理と呼び，それに対して複数

Vol.2019-ARC-235 No.31
Vol.2019-SLDM-187 No.31
Vol.2019-EMB-50 No.31
2019/3/18

の要素を引数として単一の結果を出力する処理を集約処理と呼ぶものとする．独立処理と集約処理の概念に従って Enum ライブラリ関数の機能仕様を表現すると，その処理パターンは次の 4 種類に分類できる．表 1 に関数名とそれらの機能仕様，および処理パターンを示す．

独立処理のみで機能仕様を表現できるもの

map 関数と filter 関数が該当する．各要素に対する演算は独立処理で表現され，結果を要素ごとに独立して出力する．

独立処理と集約処理の組合せで機能仕様を表現できるもの

all?関数と any?関数が該当する．各要素に対する演算は独立処理で表現され，演算結果をもとにコレクション全体で集約して出力するプロセスは集約処理を用いて表現される．

集約処理のみで機能仕様を表現できるもの

max 関数，min 関数，reduce 関数，および sum 関数が該当する．各要素における演算は集約処理により表現される．

独立処理と集約処理だけでは機能仕様を表現できないもの

sort 関数が該当する．sort 関数では，2 つの要素を交換する演算が必要であり，独立処理と集約処理だけでは表現することができない．

このように，Elixir の Enum ライブラリ関数には様々な処理パターンが存在する．FPGA は並列処理において高い処理性能を発揮できるため，IP モジュール生成においてもできるだけ並列処理で実現することが望ましい．処理パターン 1 については直観的に並列化が可能であると考えられるが，注意すべきは残りの処理パターンに該当する Enum ライブラリ関数である．処理パターン 2 および 3 に含まれる集約処理は，多段化した場合に実行順序を入れ替えた時の計算結果の正しさは保証されないため，基本的に順次処理で実現される．しかし，順次処理で実現すると FPGA の強みである並列性能を十分に発揮することができない．そこで，関数の集約処理を解析し，解析結果に応じて順次処理パターンの特定の構成を集約または並列化する必要がある．処理パターン 4 に含まれる関数についても同様に，要素関数を解析し機能仕様をなるべく並列処理で実現する．

表 1 実装対象とする Enum ライブラリの関数

関数名	機能仕様	パターン
all?	各要素の演算結果がすべて真ならば true を返す	2
any?	各要素の演算結果が 1 つでも真ならば true を返す	2
filter	要素関数の結果が真の要素のみを返す	1
map	各要素に対して演算を行う	1
max	最大要素を返す	3
min	最小要素を返す	3
reduce	先頭要素から前の要素の結果を引数として順次処理を行い，結果を集約して返す	3
sort	要素を昇順に並び変える	4
sum	全要素の合計値を求める	3

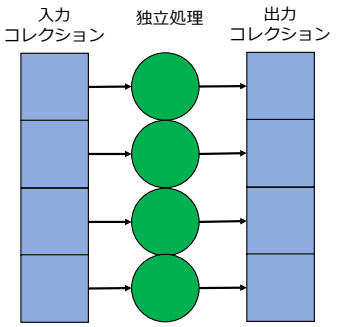

図 3 パターン 1 における IP モジュールの設計

4.2 各処理パターンのハードウェアによる実現方法

本節では，表 1 で述べた Enum ライブラリ関数の各処理パターンに対して，それらと機能等価なハードウェア回路を実現する方法を示す．各関数の要素関数を解析し，解析結果に応じて順次処理パターンの特定の構成を並列化または集約できるようにする．

4.2.1 パターン 1

パターン 1 に属する関数では入力コレクションのすべての要素に対して要素関数を並列に適用する．要素関数は独立処理であり，IP モジュールにおいて最大限の並列性がもたらされる．以上から，パターン 1 に属する関数の IP モジュールは，図 3 の設計で実現することができる．

4.2.2 パターン 2

パターン 2 に属する関数では，入力コレクションの各要素に対して独立処理により真偽値を評価し，その後全ての結果が真であるかを評価する．このような処理は，独立処理を実行するフェーズと集約処理を実行するフェーズからなると解釈できる．独立処理実行フェーズにおいては，コレクションのすべての要素に対して要素関数を並列に適用する．要素関数は独立処理であり，独立処理実行フェーズにおいては最大限の並列性がもたらされる．ただし，集約処理実行フェーズにおいては，図 4 に示すように計算結果の正しさを確実に保証するためには順次処理を行う必要がある．そこで，集約処理の順次処理を集約することを考える．順次処理が集約可能であるかどうかは，処理を構成する要素関数の結合性に依存する．結合性とは，二項演算子 $\circ$ が次の式を満たすような性質のことである．

$$(f \circ g) \circ h = f \circ (g \circ h) \tag{1}$$

集約処理が結合法則を満たせば，図 5 のように，隣接する要素が中間シーケンスである限り，ペア単位の組み合わせを任意の順序で使用できるため，順序処理を集約できる．

以上から，パターン 2 に属する関数については，合成フ

Vol.2019-ARC-235 No.31
Vol.2019-SLDM-187 No.31
Vol.2019-EMB-50 No.31
2019/3/18

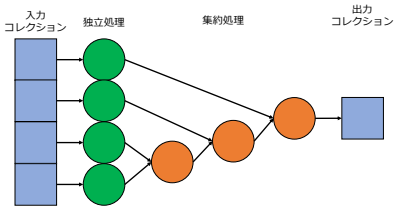

図 4　パターン 2 における順次処理の IP モジュールの設計

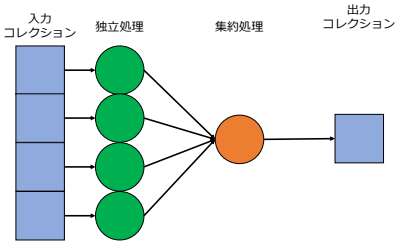

図 5　パターン 2 における集約された IP モジュールの設計

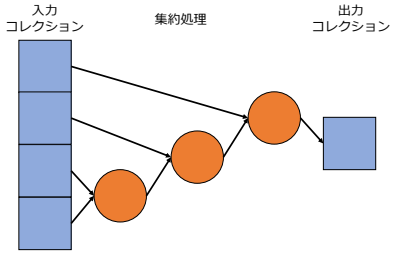

図 6　パターン 3 における順次処理の IP モジュールの設計

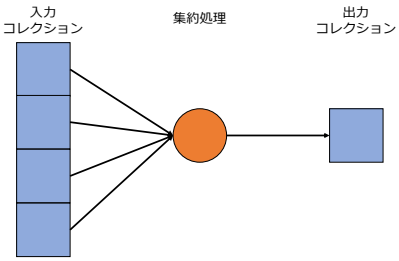

図 7　パターン 3 における集約された IP モジュールの設計

ローの AST 解析において集約処理が結合法則を満たすかどうかを判定し，満たす場合は集約された IP モジュールを合成する．パターン 2 に属する関数の集約処理はそれぞれの関数名で定まるため，関数名をもとにその集約処理が結合法則を満たすかどうかを判定できる．

例として `all?`の場合は，集約処理である論理積は結合法則を満たすため，図 5 のように集約された IP モジュールを合成することができる．

4.2.3　パターン 3

パターン 3 に属する関数では，処理は集約処理実行フェーズからなり，計算結果の正しさを確実に保証するためには順次処理を行う必要がある．本パターンにおいてもパターン 2 の場合と同様に集約処理の結合性に着目して順次処理を集約する．ただし，パターン 2 では集約処理は関数名（`all?`や `any?`など）で一意に定まるのに対してパターン 3 では集約処理は引数として与えられる．そのため，引数の集約処理が結合法則を満たすかを判定し，集約可能であれば図 7 の集約された IP モジュールを合成し，そうでなければ図 6 の順次処理の IP モジュールを合成する．

4.2.4　パターン 4

パターン 4 に属する `sort` 関数では，入力コレクションの各要素に対して，昇順にデータが並び変わるようにソート処理を行う必要がある．ソート手法については様々な手法が存在するが，IP モジュールとして実現する際は FPGA の並列処理性能を活用できるソートアルゴリズムであることが望ましい．そこで，本研究においては並列性能の高いソートの 1 つであるバイトニックソートを用いて IP モジュール合成を行う．

`sort` の IP モジュール処理は複数の交換処理実行フェーズからなる．各交換処理実行フェーズにおいては，パターン 1 の時と同様に関数の呼び出しセットを実行する．すなわち，入力コレクションのすべての要素に対して各フェーズにおいて交換処理を並列に適用する．各交換処理は，別々のデータ要素にアクセスする．交換処理の各フェーズにおいては，各要素に対して交換処理を独立して実行することができるため，パターン 1 と同様に最大限の並列性がもたらされる．

5.　インタフェース機構

本章では，合成したハードウェア回路のデータ通信と実行制御を行うインタフェース機構の設計を行う．インターフェース機構の実装においては，対象デバイスによってその機構は様々であるため，本研究においては CPU と FPGA を 1 チップに集積したプログラマブル SoC として代表的な，Xilinx 社の Zynq ファミリ [11], [12] のデバイスを対象に設計を行うこととした．まずはじめに，本研究がインタフェース機構を実装する対象の構成について述べる．次に，インタフェース機構の構造について述べる．その後，NIF 機能を用いたインタフェース回路の振る舞いについて述べる．

5.1　実装対象の構成

本研究では，実装対象のデバイスとして，XC7Z020-1CLG400C を搭載した Zynq ボードである Zybo Z7-20[13] を用いる．文献 [7] において，Zybo Z7-20 にて Elixir および Erlang/OTP 環境が動作することが確認されている．本デバイスでは，ARM Cortex-A9 dual-core processor と FPGA が ARM AMBA AXI4 バス [14] で密結合されている．Zynq ではソフトウェアとハードウェアの間の通信（SW/HW 間通信）において，4 種類のポートが用意されている．GP-AXI は汎用向けに利用できる 32 ビットバスのポートであり，SW と HW の双方が通信のマスタとなれる．AXI-HP は FPGA からのオンチップメモリまたは

Vol.2019-ARC-235 No.31
Vol.2019-SLDM-187 No.31
Vol.2019-EMB-50 No.31
2019/3/18

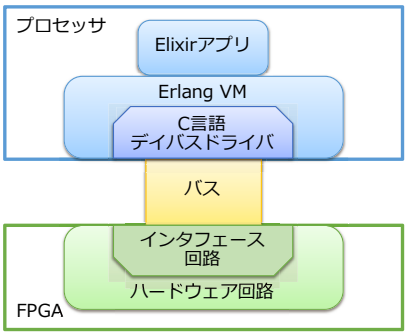

図 8　SW/HW 間の通信インタフェース機構

DDR メモリへのアクセス，AXI-ACP は FPGA からプロセッサの L2 キャッシュへのアクセスのための 64 ビットバスのポートである．AXI4 におけるプロトコルの実装は 3 種類が用意されている．そして，Zynq の AXI4 における通信プロトコルは，メモリマップ方式で最大 256 ワードのバースト転送が可能である AXI4，回路規模が小さい AXI4-Lite，アドレスフェーズを持たずストリーミングに特化した AXI4-Stream が実装されている．

5.2　インタフェース機構の構造

提案する設計環境 Cockatrice が生成するインタフェース機構の構造について説明する．図 8 は，Cockatrice が生成する SW/HW 間のインタフェース機構を示している．インタフェース機構は，ハードウェア回路内のインタフェース回路，AXI4 バス上のポートとプロトコル，および，Erlang VM 内にロードされている C 実装のデバイスドライバからなる．Xilinx 社の Vivado で合成される Zynq 向けのハードウェア回路は，AXI4 バスインタフェースを介して Linux カーネルのアドレス空間にメモリマップされる．Vivado では C 言語によるこのデバイスドライバの雛形も生成することもできる．ARM プロセッサからはこれを用いることで入出力およびハードウェア処理を制御することができる．`cockatrice_lib` ライブラリは，Elixir アプリからは Erlang/OTP の NIF 機能によってラッピングする．NIF（Native Implemented Function）とは，他言語で実装されたネイティブコードを Erlang VM から呼び出して実行するための仕組みである．Zynq 上の SW/HW 間通信インタフェース機構は入出力を配列として扱う必要があるため，Elixir のリストを C における配列に変換する処理，および IP モジュールからの演算結果の配列を Elixir のリスト形式に変換する処理を NIF 機能を介して実行する

次節では NIF 機能を介したデバイスドライバについて述べる．

5.3　インタフェース回路の振る舞い

実行時における SW/HW 間インタフェース機構の振る舞いは以下である．

(1) Elixir アプリからデータフロー型ハードウェア回路に変換された関数を呼び出す．
(2) NIF モジュールの初期化の後に，通信インタフェース機構のデバイスドライバの処理に入る．
(3) Elixir アプリから入力として与えられたリストのデータを，ハードウェア回路に入力するための配列形式に変換する．
(4) FPGA 上のハードウェア回路の処理を実行する．初期化ののちに入出力の配列を設定して処理を開始する．処理の正常終了は割込み信号と AXI4 インタフェースの状態レジスタで監視する．
(5) ハードウェア回路の処理結果として出力された配列を Elixir のリスト形式に変換し，呼び出し元の Elixir アプリに返す．

6.　予備評価

6.1　評価環境

Zybo Z7-20[34] を採用して，本研究で設計したハードウェア回路の有効性を評価した．本評価においては，FPGA の動作周波数は 50MHz とし，SW/HW 間通信プロトコルは AXI-Lite，通信ポートは GP-AXI を使用した．

実行時間の計測においては，Elixir から呼び出し可能な Erlang 標準ライブラリの時間計測関数である `:timer.tc` を使用した．

Zybo の OS は Ubuntu 16.04 LTS を使用した．実行評価は Zybo の Ubuntu 上で，NIF 機能を介して C 実装のデバイスドライバを用いて，Elixir アプリからハードウェア回路に相当する関数を呼び出す形で行った．

提案するハードウェアの設計手法で用いた主なツールのバージョンを表 2 に示す．Vivado はデータフロー合成の処理で得られた HDL 記述をもとに，FPGA デバイスに書込み可能なビットストリームを合成する論理合成ツールである．Petalinux は，Vivado から出力されたビットストリームをもとに zybo 上で動作可能な Linux カーネルとブートローダを生成するツールである．elixir と erlang/OTP のバージョンは，Zybo 上の Ubuntu でインストールしたバージョンである．

表 2　ツールのバージョン

ツール	バージョン
Vivado	2017.4
Peltalinux	2017.4
elixir	1.7.4
erlang/OTP	20.3

6.2　評価対象

本節では，提案する設計環境を用いて合成された HW 回路について評価を行う．Elixir アプリ上で，Enum ライ

ブラリの関数 map，all?，および reduce について，HW 回路を呼び出して実行，ソフトウェア上で実行，および，ソフトウェアで Flow ライブラリを使用して実行した時で比較評価を行った．Flow ライブラリでは Enum ライブラリの all?関数および reduce 関数には対応していないため，ソフトウェアで Flow ライブラリを使用して実行した時は map に対してのみ評価を行った．入力としては，1 から 1600,3200,6400,12800 までのそれぞれすべての整数を含む 4 つのリストとした．各関数における処理は，

- map：入力コレクション内の各要素を 2 倍する
- all?：入力コレクション内の各要素について，2 で割った余りがすべて 0 であれば true(IP モジュールでは 1) を返す
- reduce：入力コレクションの各要素を順次加算して返す

とした．

6.3 実行時間の評価

各方式について，実行時間を計測した．HW 回路の評価では入力としてリストを渡した後，

- NIF 機能呼び出し処理
- リスト/配列変換（リストから配列＋配列からリスト）
- AXI プロトコルを介した CPU/FPGA 間のデータ転送 (往復)
- FPGA 内の HW 回路で処理を実行

が行われる．そのため，評価した時間はこれらすべてを含めた時間になる．ハードウェア実行の際はすべてのパターンにおいて 32 並列のハードウェア上の回路で実行し，Flow を用いたソフトウェアでは Flow.enumerable の stage 数をデフォルトの値（コア数 2）に設定して実行するものとした．表 3 から表 5 に，それぞれ map，all?,reduce 関数に対して各実行パターンで実行した際の実行時間を示す．

表 3 map 関数の実行時間 [μs]

処理方式	要素数			
	1600	3200	6400	12800
HW	890652	3492932	13881233	55158058
SW	524	115	1975	4672
SW(Flow)	5398	9758	20498	36151

表 4 all?関数の実行時間 [μs]

処理方式	要素数			
	1600	3200	6400	12800
HW	901344	3492499	14948344	54906845
SW	6	6	4	7

評価結果から，各実行方式において大きな差があることがわかった．評価を行ったすべての関数およびデータ数において，ハードウェア上で実行したものはもっとも遅いという結果になった．これは，今回の実装では AXI-Lite プロトコルを用いているため，大規模なデータを連続して送受信する際に通信にかかるオーバヘッドが増大し，全体の処理時間に大きな影響を与えてしまっているためであると考えられる．そのため，高い並列度を維持しながら高速な通信を行うためには，バースト転送に対応した AXI-Full または AXI-Stream を用いた通信方式をサポートする必要があると考えられる．

表 5 reduce 関数の実行時間 [μs]

処理方式	要素数			
	1600	3200	6400	12800
HW	895984	3532384	15012232	55173891
SW	412	799	1599	3182

6.4 インターフェース機構の実行時間の評価

本項では HW 回路全体の処理時間のうち，HW 回路内の演算以外にかかる時間であるインターフェース機構の実行時間について調査を行う．インターフェース機構のうち，前項の評価で重い処理であることがわかった AXI 通信部を除いた残りの部分である，NIF 機能呼び出し処理およびリスト/配列変換処理について評価を行う．表 6 に，リスト/配列変換処理 (リストから配列+配列からリスト) の実行時間を示す．表 7 に NIF 機能単体の実行時間を示す．

表 6 リスト/配列変換処理の実行時間 [μs]

処理内容	要素数			
	1600	3200	6400	12800
リスト/配列変換処理	216	700	1028	3608

表 7 NIF 機能呼び出し処理の実行時間 [μs]

処理内容	要素数			
	1600	3200	6400	12800
NIF 機能呼び出し処理	7	10	8	8

評価結果から，以下の 2 つのことがわかる．1 つめは，NIF 機能を呼び出すのに約 10 μ 秒が必要ということである． 前節の結果を参考にすると，大規模なデータを扱う際には NIF 機能を呼び出す処理は影響を無視できるほど小さいものであることが分かる．2 つめは，リスト/配列変換処理にはかなりの時間がかかるということである．今回入力値として与えた要素数に対する結果は，それぞれソフトウェア実行時の結果に匹敵する．そのため，IP モジュール実行の高速化のためには，AXI 通信部の最適化だけではなく，リスト/配列変換処理についても高速化を図る必要がある．

6.5 Enum ライブラリ関数の IP モジュールの回路面積およびレイテンシの評価

32 並列の Enum ライブラリ関数の IP モジュールの回路面積およびレイテンシを，表 8 に示す．計測データは，論理合成ツール Vivado 上で合成したときのデータである．なお，各関数のデータは，プロセッサと通信するための AXI 制御回路部等を含んだデータである．

表 8 Enum ライブラリ関数の IP モジュールの回路面積およびレイテンシ

関数	LUT	LUTRAM	FF	レイテンシ [ns]
32 入力 all?	386	29	469	6.441
32 入力 map	797	59	1506	11.102
32 入力 reduce	1112	60	1444	14.075

今回の評価では FPGA は 50MHz で動作していることから，表 8 のすべての関数は 1 クロックで処理が完了することがわかる．

7. おわりに

本研究では，関数型 Elixir を FPGA のハードウェア設計における記述言語として活用する設計環境 Cockatrice を提案した．提案するハードウェア設計環境は，Enum および Flow のライブラリとパイプライン演算子を用いた Elixir の記述から，機能等価なハードウェア回路および Elixir ソフトウェアからのハードウェア回路の制御機構を提供する．Elixir プログラム上から関数呼び出しの形で使用することが可能となり，処理の一部として使用することができる．Elixir プログラムを実行するデバイスの処理性能のさらなる向上とシステム全体の省電力化が可能になる．

Elixir を設計記述言語として活用する Cockatrice を実現するための要素技術として，本研究では Flow および Enum ライブラリの関数を機能等価なハードウェアモジュールとして設計すること，および合成したハードウェア回路に対してデータ通信と実行制御を行うインタフェース機構の設計に取り組んだ．

評価の結果，実機上の Elixir アプリから関数呼び出しの形で IP モジュールを使用できることが確認できた．しかし，設計環境によって合成された IP モジュールは，Elixir 上でソフトウェアした時と，Elixir の並列処理ライブラリである Flow を用いて実行した時と比べて高速化を実現することはできなかった．主な原因として，AXI-Lite による SW/HW 間通信に多くの時間がかかってしまっていることが挙げられる．

今後の方針としては主に 2 つが挙げられる． 1 つは，AXI-Full や AXI-steam といった AXI-Lite 以外の通信プロトコルに対応し，実行時間のボトルネックとなっている SW/HW 間のインターフェース機構の高速化を行うことである． 2 つめは Elixir 記述からより最適な IP モジュールを合成する手法の検討である．そのためには，関数単位の最適化だけでなく，モジュール全体をまたがって最適化を行う必要があると考えられる．

謝辞 本研究は，JST さきがけ JPMJPR18M8 の支援を受けたものである．

参考文献

[1] Dave Thomas: Programming Elixir ≥ 1.6: Functional |> Concurrent |> Pragmatic |> Fun, Pragmatic Bookshelf, 2018.
[2] Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters, *Commun. ACM*, Vol. 51, No. 1, pp. 107–113, 2008.
[3] Flow: Computational parallel flows on top of GenStage (online), https://github.com/elixir-lang/flow (2018.06.07).
[4] Geovane Fedrecheski, et al.: Elixir programming language evaluation for IoT, *Proc. of ISCE*, pp. 105–106, 2016.
[5] Phoenix: Productive. Reliable. Fast. A productive web framework that does not compromise speed and maintainability (online), http://phoenixframework.org.
[6] Chris Mccord: Elixir vs Ruby Showdown - Phoenix vs Rails (online), https://littlelines.com/blog/2014/07/08/elixir-vs-ruby-showdown-phoenix-vs-rails.
[7] 高瀬英希，上野嘉大，山崎進: 関数型言語 Elixir の IoT システムへの導入に向けた基礎評価，情報処理学会研究報告，Vol. 2018-EMB-48，No. 5，pp. 1–8，2018.
[8] 山崎進，森正和，上野嘉大，高瀬英希: Node プログラミングモデルを活用した C++および Elixir の実行環境の実装，情報処理学会研究報告，Vol. 2018-OS-144，No. 1，pp. 1–6，2018.
[9] Jhdl - an hdl for reconfigurable systems., Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, pp. 175–184 (1998).
[10] 山崎進，森正和，上野嘉大，高瀬英希: Hastega: Elixir プログラミングにおける超並列化を実現するための GPGPU 活用手法，情報処理学会第 120 回プログラミング研究会，Vol. 2018，No. 2，pp. 8，2018.
[11] Xilinx Inc.: Zynq-7000 SoC (online), https://japan.xilinx.com/products/silicon-devices/soc/zynq-7000.html.
[12] Xilinx Inc.: Zynq UltraScale+ MPSoC (online), https://japan.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.
[13] Digilent Inc.: Zybo Z7 (online), https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/.
[14] Shaila S. Math, et al.: Data transactions on system-on-chip bus using AXI4 protocol, *Proc. of ICONRAEeCE*, pp. 423–427, 2011.