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
VLD2019-114 HWS2019-87 (2020-03)

# プログラマブル SoC における Erlang からのハードウェア制御

若林　秀和†　石浦菜岐佐†

† 関西学院大学 理工学部 〒 669–1337 兵庫県三田市学園 2-1

**あらまし** 本稿では Erlang プログラムからカスタムハードウェアを制御するための一手法を提案する．近年の組込みシステムには, IoT サービスのための通信を含む益々高い機能とともに, 複雑な計算をエッジで行うための高い性能が求められるようになってきている．本稿では, プログラマブル SoC において通信処理を Erlang で実装し, 処理量の大きな計算はハードウェアで実行する枠組みとして, Erlang プログラムから Erlang ポートを介してハードウェアの制御を行えるようにする．ハードウェアを制御するための Erlang メッセージはポートを介してバイト列として授受され, 外部プログラムでこれをハードウェアのレジスタの読み書きに変換する．ハードウェアに簡単なニューラルネットワークを実装して Erlang から制御を行った結果, 実行時間を大幅に短縮できることが確認できた.

**キーワード** Erlang, ハードウェア, ポート

# Hardware Control from Erlang Programs on Programmable SoC

Hidekazu WAKABAYASHI† and Nagisa ISHIURA†

† Kwansei Gakuin University, 2-1 Gakuen, Sanda, Hyogo, 669–1337, Japan

**Abstract** This article presents a method for controlling custom hardware from Erlang programs. Higher and higher functionality, including communication capability for IoT services, is required for embedded systems as well as high performance to shift sophisticated information processing from cloud to edge. With the view of implementing message processing in Erlang and offloading heavy computation tasks to custom hardware, our method enables activating hardware modules from Erlang programs through Erlang ports. Erlang messages to control hardware are sent and received via ports as byte sequences, which external programs translate into reads and writes to hardware registers. Through a preliminary experiment using a hardware implemented neural network, we confirmed that the execution time was substautially reduced by our offloaded scheme.

**Key words** Erlang, hardware, port

## 1. は じ め に

組込みシステムは, 家電製品, 自動車, 医療用機関, 産業機器等, 多岐に渡る製品に内蔵されている．これらの製品に対する多種多様な要求に対応するため, 組込みシステムには益々高い機能や性能が求められている.

近年ではネットワーク環境の普及およびこれを活用した新たなサービスの拡大に伴い, 組込みシステムには単体での動作だけでなく他のシステムとの連携が求められるようになってきている．組込みシステムの機能はこの点でも高度化しており, このようなシステムの仕様記述や効率的な設計手法も重要な課題となっている.

これ対する一アプローチとして, Erlang [1] や Elixir [2] 等の並行処理プロセスに基づいたメッセージ処理指向言語を用いてシステムの制御を記述する試みがなされている．Erlang は交換器の実装のために開発された言語で, 近年では大量のメッセージを捌く Web サービスの実装に用いられている [3]. また, 並行プロセスとメッセージ通信によってイベントを簡潔に記述できる点や, IoT デバイスの開発に必要なライブラリが豊富にある点に注目し, Elixir を利用して IoT デバイスを開発するフレームワーク Nerves の開発が行われている [4].

一方で, データ通信の帯域削減の観点から, システムにおける主な処理をクラウド側からエッジ側へと移行する動向がある．これにより, レスポンスやデータ秘匿性を向上させることができるが, 処理を高速に実行するだけの性能がエッジ側の端末に要求される．このため, 処理の大きい部分をハードウェアで実装し, 高速化を図る試みが多くされている.

文献 [5], [6] の高位合成手法では Eralng からの高位合成手法を提案している．この手法では, Erlang のサブセットにより記述された並行プロセスを並列に実行可能なハードウェアにできる．しかし, 合成可能な Erlang 文法に制限があり, 合成可能なプロセス数が 10 程度等 適用できる機器が限定されてしまう.



This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2020 by IEICE

そこで本稿では, Erlang プログラムをプロセッサ上で実行し, 処理量の大きな計算を専用ハードウェアに実行させるための枠組みとして, Erlang プログラムからポートを介してハードウェアを制御する方法を提案する. Erlang プログラムからポートにメッセージを送信して外部プログラムを実行し, 外部プログラムから MMIO (Memory Mapped I/O) でハードウェアを制御する. ポートを介することで, ハードウェアが何らかの理由で強制終了しても, Erlang プログラムは動作し続けることができる.

本手法を用いて, ニューラルネットワークの処理をハードウェア化して Erlang から実行できるようにした. その結果, 実行時間を大幅に短縮することができた.

## 2. Erlang

### 2.1 Erlang

Erlang [1] は並行処理指向の関数型言語であり, 動的に生成される複数のプロセスにより並行処理を記述する. プロセス間のデータ共有は, 共有メモリではなく非同期のメッセージ通信により行う. プロセスは非常に軽量であり, 多数のプロセスを生成して大量のメッセージを処理することができる. また, Eralng は共有メモリを持たず, 資源等の管理はプロセスで行う. 排他制御を実装する必要がないため, プログラムの並列化を容易に行える.

外部機器との通信は, TCP や UDP のソケットを利用して行う. Erlang では並列, 逐次, ブロッキング, ノンブロッキングなどの様々なサーバを構築することができる. 特に並列サーバでは, 新しいソケット接続ごとにプロセスを生成することにより, 同時に数十万のクライアントから来るメッセージを処理することができる.

また, Erlang では "ポート" を介した外部接続も可能である. ポートを利用すると, C や Python 等 で書かれた外部プログラムと Erlang を接続することができる. この時, 外部プログラムは Erlang ランタイムシステムとは別のオペレーティングシステムのプロセスで動作する.

### 2.2 Erlang からの高位合成

文献 [5], [6] では Erlang からの高位合成手法を提案している. この手法では, Erlang のサブセットで書かれた 1 つのプロセスを 1 つのハードウェアモジュールに合成する. プロセスはメッセージ通信を除いて他のプロセスを並行に実行できる. また, スケジューラやプロセスの管理等の実行時環境のオーバーヘッドもなくなる.

この手法では, Erlang のプログラムをコンパイルして得られる BEAM 仮想マシンのアセンブリコードから CDFG を生成し, これをバイナリ高位合成系 (機械語を入力とする高位合成系) ACAP [7] のバックエンドに入力して Verilog HDL を得る. BEAM の命令の中には, メッセージの送受信, ガベージコレクション等, データフローグラフが大規模になるものが存在するため, これらの処理は, 別途独立したハードウェア (ライブラリモジュール) として実装している.

しかしこの手法では, 合成できる Erlang の文法に制約が大きく, ソケットやポートを利用した Erlang プログラムは合成できない. また, 合成を想定している Erlang プロセス数は 10 程度であり, 動的なプロセス生成もできないため, 適用可能な機器は限定されてしまう.

## 3. Erlang からのカスタムハードウェア制御

本稿では外部との通信処理を Erlang で実装し, 処理量の大きな計算を専用ハードウェアで実行するというスキームを実現するため, Erlang からハードウェアを制御する一手法を提案する. Erlang プログラムからポートを介してメッセージを外部プログラムに送信し, 外部プログラムによって MMIO でハードウェアの制御を行う. ハードウェアでの計算結果は, 外部プログラムがポートを介して Erlang に返信する.

### 3.1 Erlang からのハードウェア制御法

Erlang プログラムからハードウェアを制御する方法としては以下の 3 通りが考えられる.

（1） 内部プロセスからのハードウェア制御

ハードウェアのレジスタやポートに直接値を書き込む関数を作り, その関数を内部プロセスから呼び出すことによりハードウェアを制御する. メッセージ通信によってハードウェアを直接制御することができるが, Erlang の実行環境である Erlang RunTime System (以下 ERTS) 自体を書き換える必要がある. また, 実装によってはハードウェアが強制終了すると内部プロセス自体も強制終了してしまう危険性がある.

（2） 外部プロセスからのハードウェア制御

ハードウェアを制御する外部プロセスを実装し, TCP または UDP のパケットを送信しハードウェアを制御する. ソケット通信によるハードウェアの制御が可能になるが, ハードウェアに TCP や UDP の通信とメッセージの解釈を行う機能を実装する必要がある.

（3） Erlang ポートを介したハードウェア制御

ERTS の外部のオペレーティングシステムのプロセスで外部プログラムを実行し, バイト型通信チャネルであるポートを通じてそのプロセスと通信をする. ERTS 自体を書き換える必要はない上, 外部プログラムが強制終了した場合でも ERTS は動作し続けることができる.

本稿では, 実装の容易さも考慮し, (3) の Erlang ポートを介したカスタムハードウェアの制御方法を提案する.

### 3.2 システム概要

システムの概要図を図 1 に示す. Erlang プログラムは複数の Erlang プロセスを起動し, 外部ノードと通信を行いながら機器の主要な処理を行う. ハードウェアによる計算を起動する場合には, ポートに対して必要な情報をメッセージとして送信する. 外部プログラムはプロトコル変換部と HW 制御部から成り, プロトコル変換部がポートから出力されるバイト列を解釈し, HW 制御部が MMIO でハードウェアのレジスタに値を書き込むことによりハードウェアを制御する. 計算結果がレジスタに書き込まれると, 外部プログラムはこれをバイト列に変換して Erlang ポートに返す.

これにより, Erlang プログラムからは特にハードウェアを意識することなく, メッセージ通信によって計算のオフローディングを行うことができる. また, ポートを通じてハードウェアを制御することによって, ハードウェアが動作している間も Erlang プログラムを動作させることができる.

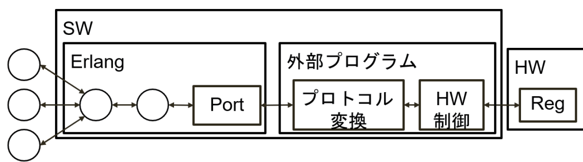



図 1　システムの概要

### 3.3　データの変換

外部プログラムとの通信は, Erlang ポートを管理するプロセスを用意し, そのプロセスとメッセージ通信を行うことにより実現する.

ハードウェア起動時には, 命令や入力値をまとめたメッセージを Erlang のライブラリである term_to_binary 関数でバイト列に変換し, Erlang ポートに送信する.

Erlang のタプルを変換しハードウェアに送信する様子を図 2 (a) に示す. この例では, Erlang プログラムから {add, 1, 2} を送信している. {add, 1, 2} を term_to_binary でバイト型に変換すると, ≪131, 104, 3, 100, 0, 3, 97, 100, 100, 97, 1, 97, 2≫ というバイト列が得られる. これをポートを通じてプロトコル変換部へと送信する. プロトコル変換部では受け取ったメッセージを復号し, 情報を読み取る. 今回のバイト列の例では, 131 から Erlang binary であることを読み取り, 104 でタプル, 次の 3 より要素数が 3 であること, 100, 0, 3 から第 1 要素は 3 文字のアトムであること, 97, 100, 100 から文字列は add であること, 97, 1 からタプルの第 2 要素は short int 型の整数で値は 1 であること, 最後に 97, 2 からタプルの第 3 要素は short int 型の整数で値が 2 であることを読み取る. これらの情報から, 1 と 2 の加算を行うという命令を生成し, MMIO でハードウェアに値を入力する.

ハードウェアから値を読み取り Erlang に返信する様子を図 2 (b) に示す. ハードウェアの計算結果が格納されているレジスタを MMIO で読み取り, 結果をバイト型に変換する. 今回の例では, 先頭に Erlang binary であることを示す 131 に, short int 型整数の 3 であることを示す 97, 3 を連結しバイト列 ≪131, 97, 3≫ に変換する. ポートを通じて送られたメッセージは Erlang のライブラリ binary_to_term で復号され計算結果である 3 が読み取られる.

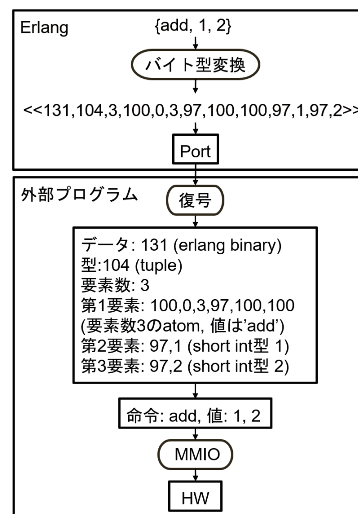
Erlang からのデータはバイト列で送られてくるためこれを復号する必要がある. 主要なものを抜粋した解釈表を表 1 に示す. Erlang のデータはタグとデータという組み合わせで表現されている.

表 1　バイト列の解釈表 (一部)

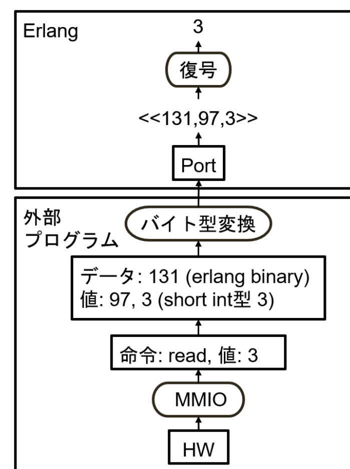
| バイト列 | 意味 |
|---|---|
| ≪131≫ | Erlang binary |
| ≪97, n≫ | short int 型 の n |
| ≪100, 0, n≫ | 要素数 n のアトム |
| ≪104, n≫ | 要素数 n のタプル |
| ≪107, 0, n≫ | 要素数 n のリスト |

### 3.4　外部プログラムからのハードウェア制御

Erlang から送られてきたメッセージを元にアドレス空間にマッピングしたハードウェアの入出力用のレジスタの読み書きを行うことによりハードウェアの制御を行う.




(a) Erlang からハードウェアへの送信




(b) ハードウェアから Erlang への返信

図 2　プロトコル変換の流れ

ハードウェア側では, これらの仕様に合わせて構成をカスタマイズする必要がある. ハードウェア構成の概要を図 3 に示す. ハードウェアには CPU とハードウェア間のやり取りを担うバスモジュールを用意する. バスモジュールはアクセスするレジスタ番号と, 書き込む値, 制御信号を受取り対応するレジスタに値を書き込む. どのレジスタがどのアドレスに対応するかは, ハードウェアを設計する際に決定する. CPU からバスモジュールにアクセスするためには, レジスタがマッピングされているアドレスに値を書き込むことでやり取りができる. 計算結果を読み取るときは, 計算が完了するまで待機し, 計算結果が格納されているレジスタの値を読み取る.

## 4.　実装と実験

### 4.1　プログラマブル SoC による実装

本稿で提案するハードウェア制御方式に基づく制御の雛形を Xilinx 社のプログラマブル SoC である PYNQ-Z1 上に実装した.

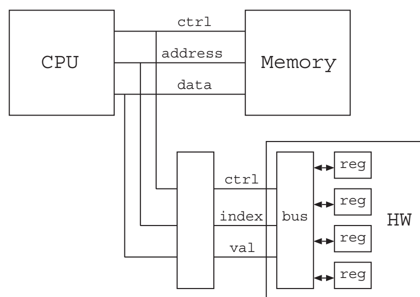



図 3　MMIO によるハードウェア制御

PYNQ-Z1 は Zynq を搭載した評価ボードであり, Python を利用した制御や Jupyter Notebook を利用した開発が可能である．Zynq では, プロセッサ部分を PS (Processing System), FPGA 部分を PL (Programable Logic) と呼んでおり, PS 部には ARM 搭載し, Linux を動作させることが可能である．PYNQ では FPGA の設定ファイルである bitstream を “Overlay” と呼んでおり, 複数の Overlay を用意しておくことにより動的に FPGA を再構成することも可能である．

#### 4.1.1　システムの構成

実装したシステムの構成を図 4 に示す．Erlang プログラムと外部プログラムは PS 部に, カスタムハードウェアとバスモジュールは PL 部に実装した．

Erlang プログラムは外部ノードと UDP によって通信を行う server プロセスと, ポートを介して外部プログラムと通信をする hw_port プロセスから成る．外部ノードとしては, 温度センサや水位センサのようなものを想定している．今回の実装では, server プロセスは外部ノードからデータを受け取ると hw_port プロセスに計算を依頼する．hw_port プロセスは外部プログラムを介してハードウェアで計算を行い, 結果を受け取ると, それを server プロセスに返信する．server プロセスは受け取った結果をノードへ送信する．

プロトコル変換部は, erlang-port-with-python [8] を参考に Python で実装した．プロトコル部からハードウェアへのアクセスは PYNQ Project で提供されているの MMIO クラスを利用した．

PS 部と PL 部のやり取りには AXI を利用し, AXI に Vivado HLS で高位合成した IP を接続した．

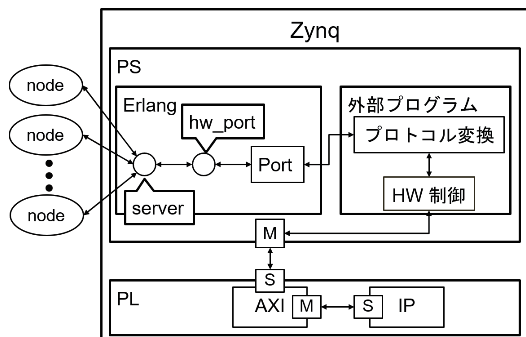



図 4　プログラマブル SoC による実装

#### 4.1.2　Erlang プログラム

Erlang プログラムのメイン関数を図 5 に示す．メイン関数ではサーバを起動する関数である start_server() と, ポートと通信をするプロセスを起動する start_port() を呼び出す．

サーバのソースコードを図 6 に示す．start_server 関数を実行すると, 4000 番ポートを開いてサーバを起動し (2 行目), サーバのソケットをオープンして待機状態に入る (4–7 行目). loop_for_server 関数はサーバの待機ループである．11–13 行目で外部ノードから UDP で受け取ったメッセージから必要な情報を取り出し, 14 行目で 2 数 X と Y の和を (ハードウェアで) 計算する add 関数を実行する．得られた結果は 15 行目で送信元に返信する．add 関数 (19 行目) は命令と引数をタプルにまとめ, call_port 関数を呼び出す．call_port 関数は, 22 行目で hw_port にメッセージを送信して計算を依頼し, 結果を add 関数に返す．

```
1 main() ->
2     start_server(),
3     start_port().
```

図 5　メイン関数

```
1  start_server() ->
2    spawn(fun() -> server(4000) end).
3
4  server(Port) ->
5    {ok, Socket} = gen_udp:open(Port, [binary
       ]),
6    io:format("server opened socket: ~p~n",[
       Socket]),
7    loop_for_server(Socket).
8
9  loop_for_sever(Socket) ->
10   receive
11     {udp, Socket, Host, Port, Bin} = Msg ->
12       io:format("server received: ~p~n", [
           Msg]),
13       {Ref,X,Y} = binary_to_term(Bin),
14       Fac = add(X,Y),
15       gen_udp:send(Socket, Host, Port,
           term_to_binary({Ref, Fac})),
16       loop1(Socket)
17   end.
18
19 add(X, Y) -> call_port({add, X, Y}).
20
21 call_port(Msg) ->
22     hw_port ! {call, self(), Msg},
23     receive
24       {hw_port, Result} ->
25       io:format("received: '~p' ~p ~n", [
           Result, self()])
26     end,
27     Result.
```

図 6　サーバの Erlang コード

Erlang ポートとの通信部を図 7 に示す．

start_port 関数を呼び出すと, 2–7 行目で接続プロセスを生成する．3 行目の register で接続プロセスを hw_port と命名し, 他の関数から呼び出しやすくする．4 行目の process_flag を呼び出すことで, 接続プロセスを補足が可能なシステムプロセスに変更する．5 行目の open_port で 外部プログラム my_protocol.py と接続するポートを開く．6 行目で 本体の loop 関数を呼び出す．

loop 関数は 14 行目で メッセージを受信すると, メッセージから必要な要素を取り出し, 15 行目でメッセージを term_to_binary でバイト列に変換して Port に送信する．16, 17 行目で Port

から返ってきたメッセージを受信し, 18 行目でメッセージを binary_to_term で復号して結果を送信元に送信する. 19–22 行目は例外処理であり, エラーメッセージを表示して例外を発生させる. メッセージの送受信が完了すると 23 行目で再度待機状態に入る.

```
1 start_port() ->
2     spawn(fun() ->
3     register(hw_port, self()),
4     process_flag(trap_exit, true),
5     Port = open_port({spawn, "./erlang_port
          /my_protocol.py"}, [{packet, 2},
          binary]),
6     loop(Port)
7    end).
8
9 stop() ->
10     hw_port ! stop.
11
12 loop(Port) ->
13     receive
14     {call, Caller, Msg} ->
15         Port ! {self(), {command,
              term_to_binary(Msg)}},
16         receive
17           {Port, {data, Data}} ->
18               Caller ! {hw_port,
                     binary_to_term(Data)};
19         X ->
20           io:format("unknown message: [~w]~
               n", [X]),
21           throw('Unknown message received
               .')
22         end,
23     loop(Port)
24     end.
```

図 7　ポートと通信を行う Erlang コード

**4.1.3　外部プログラム**

プロトコル変換部は Python3 で実装した. Python3 で実装することにより PYNQ の Overlay クラスを最大限利用することが可能となる. ハードウェアの制御を行うプログラムを図 8 に示す.

12 行目の MyProtocol クラス内に関数を handler_ というヘッダをつけて実装する. 13 行目の handler_write_hw(self, x) 関数は FPGA をコンフィグレーションする関数となっている. この関数が呼び出されると Overlay 以下に記述された bitstream などの情報をもとに FPGA をコンフィグレーションする. 成功した場合は引数をそのまま返し, 失敗した場合はエラーを出力し停止する.

17 行目の handler_neural_hw はハードウェアに値を入力し, 計算結果を読み込む関数となっている. 18 行目で IP 内の prop_cycle_0 にアクセスするためのドライバが作成され, mat_ip を通じてアクセスできるようにする. 19–21 行目の変数はオフセットとなっており, 入力と出力用のレジスタが基準から何番目にあるかを示している. これらの値は高位合成をした際に生成されるドライバのヘッダのソースを読むことで特定できる. 22, 23 行目でレジスタに書き込みを行う. 23 行目で pack 関数を利用しているのは入力値が小数の場合にも対応できるようにするためである. 24 行目で 0 番目のレジスタに 1 を書き込むことで FPGA が動作する. 25-27 行目で計算結果を読み取り, 26, 27 行目で unpack を利用することで, 計算結果が小数の場合でも正しく値を読み取ることができる. 28 行目で計算結果を Erlang に送信する. 30, 31 行目で Erlang とのやり取りを行うプログラムを実行する.

```
1 #! /usr/bin/env python3
2 # -*- coding:utf-8 -*-
3
4 from protocol import Protocol
5 from pynq import Overlay
6 from pynq import DefaultIP
7 import bnn
8 from PIL import Image
9 import numpy as np
10 import struct
11
12 class MyProtocol(Protocol):
13     def handler_write_hw(self, x):
14         self.overlay = Overlay('/home/
               xilinx/jupyter_notebooks/
               bitstreams/neural_hw/
               design_neural_hw_wrapper.bit')
15         return x
16 ,
17     def handler_neural_hw(self, x, y):
18         mat_ip = self.overlay.prop_cycle_0
19         in1 = 0x10
20         in2 = 0x18
21         out1 = 0x20
22         mat_ip.write(in1, x)
23         mat_ip.write(in2, struct.pack("<f",
               y))
24         mat_ip.write(0x00, 1)
25         a = mat_ip.read(out1)
26         b = struct.pack(">i", a)
27         res1 = struct.unpack(">f", b)[0]
28         return res1
29
30 if __name__ == '__main__':
31     MyProtocol().run()
```

図 8　ハードウェアの制御を行う Python コード

**4.1.4　ハードウェア構成**

プロセッサとのやり取りを仲介するバスモジュールとメインモジュールを搭載したハードウェアを実装した. メインモジュールは Vivado HLS を使って C++から高位合成をしたものを搭載し, バスモジュールとしては AXI を採用した.

高位合成を行う C++プログラムを図 9 に示す. a と b を加算し c に格納するだけの簡単なプログラムである. 2–5 行目の pragma 演算子で, 関数と引数を AXI のポートに接続するよう指定する. これにより AXI から容易に値を書き込むことができるようになる.

このプログラムを Vivado HLS で高位合成し, Xilinx が提供する統合開発環境である Vivado で FPGA の概要図であるブロック図を設計し, 設定ファイルである bitstream を生成する. ブロック図を図 10 に示す. 左上の rst_ps7_0_100M は 100Mhz の周期でリセット信号を出すモジュールとなっており, 左下の processing_system7_0 が PS 部となっている. また, 右の add_0 が図 9 のプログラムを高位合成して得られるモジュールであり, 中央の ps7_0_axi_periph が AXI Interconnect である.

```
1 void add(int a, int b, int& c) {
2 #pragma HLS INTERFACE ap_ctrl_none port=
      return
3 #pragma HLS INTERFACE s_axilite port=a
4 #pragma HLS INTERFACE s_axilite port=b
5 #pragma HLS INTERFACE s_axilite port=c
6
7     c = a + b;
8 }
```

図 9　高位合成される C++プログラム

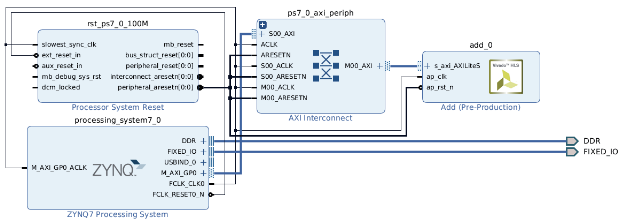



図 10　ブロック図

### 4.2 実　　験

Erlang プログラムからハードウェアを起動して計算を行い, 計算結果を返信するまでの処理時間を評価する実験を行った. 10 層の完全結合層からなる $n$ 入力 1 出力のニューラルネットワークを Python およびハードウェアで実装し, ポートを介して Erlang から起動した.

結果を表 2 に示す. 実行時間は起動から計算までを 1000 回繰り返すのに要した時間である. “Erlang + SW” は, Erlang からリクエストを送信して, ソフトウェアで計算が完了して結果を返信する実行方式であり, “Erlang + HW” は Erlang からリクエストを送信して, ハードウェアで計算を行い結果を返信する実行方式である. Erlang + SW の方式では $n$ の増加に応じて実行時間も増加しているのに対し, Erlang + HW の方式では $n$ が 1,024 の場合と 65,025 の場合を比較しても 0.1 秒程度しか増加していないのが分かる. これは HW 起動のオーバヘッドは大きいが, 計算量が大きい場合には処理時間の削減効果が大きいことを示している.

表 2　実行時間の比較

|  | 実行時間 [s] |  |
|---|---|---|
| $n$ | Erlang + SW | Erlang + HW |
| 1,024 | 5.01 | 4.14 |
| 4,096 | 6.83 | 4.13 |
| 16,384 | 12.39 | 4.24 |
| 65,025 | 35.38 | 4.28 |

(CPU: 650MHz dual-core Cortex-A9, FPGA: Artix-7)

## 5. む　す　び

本稿では, プログラマブル SoC における Erlang からのハードウェア制御方法を提案した. Erlang プログラムからポートを介して外部プログラムにメッセージを送信し MMIO でハードウェアを制御することにより, ハードウェアが強制終了しても, Erlang プログラムは動作し続けることが可能となった. 評価実験を行ったところ, 本手法の有効性を確認できた.

ハードウェア起動のオーバーヘッドが大きいため, Erlang プログラムから呼び出すを外部プログラムを C 言語などで実装することが今後の課題として挙げられる.

謝　　辞

本稿の研究にあたり, ご指導, ご助言を頂きました京都高度技術研究所の神原弘之氏, 吉田信明氏, 立命館大学の冨山宏之教授, 元立命館大学の中谷嵩之氏に感謝いたします. また, 本研究に関してご討論, ご協力頂いた関西学院大学石浦研究室の諸氏に感謝いたします. なお, 本研究は一部 JSPS 科研費 19H04081 の助成による.

### 文　　献

[1] Joe Armstrong 著, 榊原一矢訳: プログラミング Erlang, オーム社 (2008).
[2] Dave Thomas 著, 笹田耕一 訳, 鳥井雪 訳: プログラミング Elixir, オーム社 (2016).
[3] 力武健次: “Erlang で学ぶ並行プログラミング,” Software Design, 2015 年 4 月号, pp. 124–129 (Apr. 2015).
[4] Nerves が開拓する「Elixir で IoT」の新世界 - SlideShare (online), `https://www.slideshare.net/takasehideki/nerveselixiriot` (accessed 2020-01-20).
[5] H. Takebayashi, N. Ishiura, K. Azuma, N. Yoshida, and H. Kanbara: “High-Level Synthesis of Embedded Systems Controller from Erlang,” in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2016)*, pp. 285–290 (Oct. 2016).
[6] K. Azuma, N. Ishiura, N. Yoshida, and H. Kanbara: “Distribute Memory Architecture for High-Level Synthesis of Embedded Controllers from Erlang,” in *Proc. ACM SIGPLAN International Workshop on Erlang 2017*, pp. 13–19 (Sept. 2017).
[7] N. Ishiura, H. Kanbara, and H. Tomiyama: “ACAP: Binary Synthesizer Based on MIPS Object Codes,” in *Proc. International Technical Conference on Circuit/Systems, Computers and Communications (ITC-CSCC 2014)*, pp. 725–728 (July 2014).
[8] fujimisakari/erlang-port-with-python - GitHub (online), `https://github.com/fujimisakari/erlang-port-with-python` (accessed 2020-01-05).