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
VLD2017-75,CPSY2017-119,RECONF2017-63(2018-01)

Erlangからの高位合成のためのメモリ分散アーキテクチャ

東　香実† 浜名　将輝† 若林　秀和† 石浦菜岐佐† 吉田　信明††
神原　弘之††

† 関西学院大学 理工学部 〒669–1337 兵庫県三田市学園 2-1
†† 京都高度技術研究所 〒600–8813 京都市下京区中堂寺南町 134 番地

あらまし　本稿では, Erlang からの高位合成のためのメモリ分散アーキテクチャを提案する. 竹林らが提案した Erlang サブセットからの高位合成手法では, Erlang プロセスを並列動作可能なハードウェアモジュールに合成している. しかし, 全てのプロセスモジュールの記憶領域を一つの共有メモリに格納しているため, プロセスモジュールを並列に動作させるためにはその数に比例したメモリポートが必要となってしまう. この課題を解決するため本稿では, 各プロセスモジュールがローカルなメモリを保持するアーキテクチャへの合成を提案する. 各プロセスは自身のローカルメモリに対して他のプロセスとは独立にアクセスを行えるため, 全プロセスモジュールが並列に動作可能である. プロセス間のメッセージ送信やガーベジコレクションの際には, 他のプロセスモジュールのローカルメモリへのアクセスが必要になるが, ローカルメモリ間の接続の複雑化を避けるため, バスアーキテクチャを採用する. 同時に実行可能なメッセージ送信とガーベジコレクションはそれぞれ一つに限定し, その調停はアービタにより行う. 提案手法に基づき, 2 プロセスからなる簡単な Erlang プログラムから論理合成可能な Verilog HDL を生成し, RT レベルシミュレーションによる動作確認を行った.

キーワード　高位合成, ハードウェア/ソフトウェア協調設計, 組込みシステム, Erlang, ドメイン特化言語

Distributed Memory Architecture for High-Level Synthesis from Erlang

Kagumi AZUMA†, Shoki HAMANA†, Hidekazu WAKABAYASHI†, Nagisa ISHIURA†, Nobuaki YOSHIDA††, and Hiroyuki KANBARA††

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan
†† ASTEM RI/KYOTO, 134 Chudoji Minamimachi, Shimogyo-ku, Kyoto, 600–8813, Japan

Abstract　This paper presents a distributed memory architecture for dedicated hardware automatically synthesized from Erlang programs. Takebayashi et al. had developed a framework for generating embedded systems controllers whose behavior was specified by a subset of Erlang, where each process was mapped onto a hardware module running independently of those for the other processes. However, the resulting hardware was not of practical use because it shared a single main memory potentially accessed by all the process modules simultaneously. To address this issue, in this paper, the main memory is partitioned into banks so that each process can access its own memory independently of the other processes. In order to keep the interconnections for message passing and garbage collection to a practical size, a bus architecture is employed where requests for send and garbage collection are arbitrated by an arbiter module. From a simple Erlang specification consisting of 2 processes, a synthesizable Verilog HDL code has been generated whose behavior was confirmed by RTL simulation.

Key words　high-level synthesis, hardware/software codesign, embedded systems, Erlang, domain-specific language

1. はじめに

組込みシステムは, 家電製品, 車載機器, 医療用機器, 産業用機器等, 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ニーズに対応するため, 組込みシステムには益々高い機能や性能とともに, 小型化や低消費電力化が求められるようになっている. 組込みシステムはソフトウェアおよびハードウェアの組み合わせで実装されるが, 性能と消費電力の両立が厳しい場合にはソフトウェアをハードウェアで実装することにより電力性能比の向上が図られる. システムの効率的な設計手法として, ソフトウェアによる仕様記述からハードウェアを自動生成する高位合成技術 [1] を利用した設計手法が提案されて

いる [2], [3].

近年の IoT サービスの普及により, 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いる. 多種のイベント入力に対応した複雑な制御の効率的な実装には, 並行プロセスや割り込み処理の記述が必要であり, このようなシステムの仕様記述や効率的な設計手法が今後重要な課題になると考えられる. リアルタイム OS はシステム実装の複雑さを軽減するが, 割り込み処理や排他制御の記述には高度な技術が必要となる.

この問題の解決策の一つとして, Elrang [4] 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によるシステムの制御の記述が挙げられる. Erlang は, 交換機の実装言語として開発されたものであり, 近年では主として大量のメッセージをさばく Web サービスの実装に用いられている [5]. その一方で, 並行プロセスとメッセージ通信によってイベント処理が簡潔に記述できるという点に着目し, これを組込みシステムの実装に用いる試みも行われている [6]. Erlang によるシステムの記述をハードウェアに合成できれば, 高度化するシステムの機能をより容易に効率的なデバイスに実装することが可能となる.

文献 [7], [8] では, Erlang のサブセットからハードウェアを自動合成する手法を提案している. Erlang の 1 プロセスは独立した一つのハードウェアモジュールに合成され, 並列動作が可能である. しかし, この手法では全プロセスの記憶領域を一つのメモリ内に格納するため, プロセスを並列に動作させるためには, プロセス数に比例した数のメモリポートが必要になる. 組込みシステムの実装を考えると, メモリのポート数は 2 〜 3 程度に制限されるため, プロセスの処理を行うモジュール数を増やしても性能向上は見込めない.

そこで本稿では, Erlang からの高位合成のためのメモリ分散アーキテクチャを提案する. これは, 各プロセスが独立した記憶領域を所有するアーキテクチャであり, これによって全プロセスが並列に動作することが可能となる. メッセージ送信時には他プロセスへのメモリアクセスが発生するが, 任意のプロセス間のメッセージ送信を実現するためには膨大な結合網が必要になる. そこで本稿ではバスアーキテクチャを採用し, 他プロセスへのメモリアクセスはバスを介して行う. バスの衝突を防ぐためプロセス間の同時メッセージ送信は一組のみに制限し, その調停はアービタモジュールにより行う. 提案手法に基づき, 2 プロセスと 2 ポートからなる簡単な Erlang プログラムから, 論理合成可能な Verilog HDL 記述を生成し, RT レベルシミュレーションによる動作確認を行うことができた.

2. Erlang からの高位合成

2.1 組込みシステム記述のための Erlang サブセット

Erlang 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.

文献 [7], [8] では, Erlang のサブセットにより組込みシステムの制御を記述し, そこからハードウェアを自動合成する手法を提案している. この手法では, ハードウェアを合成可能とするために, 入力となる Erlang プログラムに次の制限を課している.

- すべてのプロセスはシステムの起動時に生成され, プロセスの動的生成や削除は行わない.
- システムの入出力は Erlang のポートを介して行い, ポートはバイト系列の入出力を行う.
- TCP/IP による外部の Erlang プロセスとの通信は行わない.
- 利用するデータ型は 28 ビットの符号付き整数, およびリストとタプルに限定し, 関数型は扱わない.

Erlang サブセットによって記述した簡単なコントローラの例を図 1 に示す. このコントローラは, 進行方向を指示するボタン入力があると, それを駆動系への制御信号に変換して出力する. ここでは, このコントローラを図 1 (a) のように二つのプロセスと二つのポートでモデル化している. port0 はボタンからの入力を受信し, プロセス proc0 に送信する. proc0 は受け取ったデータをプロセス proc1 に転送する. proc1 ではデータの変換を行い, 変換後のデータを proc0 に送信し, 最終的に port1 へ出力する. 図 1 (b) が Erlang による記述例である. `start` 関数 (4 – 14 行目) で `proc0`, `proc1`, `Port0`, `Port1` を生成する. `proc0` は関数 `loop0` (19–32 行目) を実行し, 処理する. また `proc1` は関数 `loop1` (34–44 行目) を実行し, 処理する.

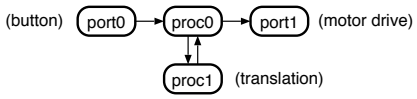
2.2 Erlang の実行系

Erlang のプログラムは, 仮想マシン BEAM のバイトコードにコンパイルされ, インタプリタで実行される. BEAM は, 1024 ワードの x レジスタ, 実行中の BEAM 命令の番地を保持する PC レジスタ, 関数の戻り番地を保持する CP レジスタ, およびスタックポインタ SP を持つ. プロセスの実行で必要になるデータのうち, x レジスタに格納できないリストやタプルのデータは, プロセス毎に用意される独立した記憶領域の先頭から格納され (ヒープ), 関数のフレームデータはその記憶領域の末尾から格納される (スタック). プロセスの記憶領域が不足するとガーベジコレクションが行われ, 不足量に応じて記憶領域の拡張が行われる. BEAM インタプリタは C 言語で実装されている.

各プロセスは線形リストで実装される一つのメッセージキューを持ち, プロセス宛に送られたメッセージはこのキューの末尾につながれる. メッセージがヒープ中のデータを参照している場合は, ヒープデータのコピーが必要になる. このコピーは送信側から受信側の記憶領域に直接行うのではなく, メッセージにミニヒープを割り当てて一旦そこにコピーし, メッセージ受信時に受信側が自身の記憶領域にコピーする. プロセスは, 現在参照している「カレントメッセージ」へのポインタを持っており, メッセージ受信時にカレントメッセージが x[0] にコピーされる. パターンマッチが成功すると, カレントメッセージはキューから削除される. パターンマッチが成功しなかった場合は, カレントメッセージのポインタが 1 つ進められ, 次のメッセージとのパターンマッチが試みられる.

2.3 高位合成の流れ

文献 [7], [8] の Erlang からの高位合成の流れは図 2 の通りである. Erlang コンパイラ erlc で Erlang のプログラムから BEAM のアセンブリコードを生成し, ここから CDFG (control dataflow graph) を生成する. これを高位合成システム ACAP [9] のバックエンドに入力して Verilog HDL を得る. Erlang の各プロセスは一つのハードウェアモジュールに合成する. 各関数はラベル, 分岐命令に基づき基本ブロックに分割し, 基本ブロック中の各 BEAM 命令を演算に変換することにより DFG (dataflow graph) を構築する. 例えば, 算術演算, ビット演算は組込み関数を実行する BEAM 命令にコンパイルされており, それを図 3 (a) のような DFG の演算に変換する. 28 ビットデータは, 下位 4 ビットのタグとともに 32 ビットのレジスタに格納されているため, これらのデータに対する演算は, データの上位 28 ビットを操作す



(a) プロセス/ポート間の接続関係

```
01: -module(roomba).
02: -export([start/0]).
03:
04: start() ->
05:   spawn(fun() ->
06:     register(proc1, self()),
07:     loop1(0, 0)
08:   end),
09:   spawn(fun() ->
10:     register(proc0, self()),
11:     Port0 = open_port({spawn, "stdbuf -i0 -o0 -e0 od
    -h -w8 /dev/input/js0 | ./controller"}, [{packet, 2}]),
12:     Port1 = open_port({spawn, "./roomba"}, [{packet, 2}]),
13:     loop0(Port0, Port1)
14:   end).
15:
16: decode([Dt,Dh,Et,Eh]) ->
      {((Dh bsl 8) bor Dt), ((Eh bsl 8) bor Et)};
17: decode(X) -> X.
18:
19: loop0(Port0, Port1) ->
20:   receive
21:     {Port0, {data, Data}} ->
22:       Data2 = decode(Data),
23:       proc1 ! {proc0, data, Data2},
24:       loop0(Port0, Port1);
25:     {proc1, Data3} ->
26:       Port1 ! {proc0, {command, Data3}},
27:       loop0(Port0, Port1);
28:     {Port1, _} ->
29:       loop0(Port0, Port1);
30:     _ ->
31:       loop0(Port0, Port1)
32:   end.
33:
34: loop1(D, T) ->
35:   receive
36:     {proc0, data, Data} ->
37:       {Drive, Turn} = calc(Data, D, T),
38:       Cmd = encode(Drive, Turn),
39:       proc0 ! {proc1, Cmd},
40:       loop1(Drive, Turn);
41:     X ->
42:       proc0 ! X,
43:       loop1(D, T)
44:   end.
45:
46: calc({Para, X}, Drive, Turn) ->
47:   if
48:     X == 258  -> {Para, Turn};
49:     X == 1026 -> {Para, Turn};
50:     X == 2    -> {Drive, Para};
51:     X == 770  -> {Drive, Para};
52:     true      -> {0, 0}
53:   end.
54:
55: encode(Drive, Turn) ->
56:   if
57:     Drive =< 57343, Drive >= 32768 ->
58:       if
59:         Turn =< 57343, Turn >= 32768 -> {146, 0, 127, 0, 63};
60:         Turn =< 32767, Turn >= 12288 -> {146, 0, 63, 0, 127};
61:         true                         -> {146, 0, 127, 0, 127}
62:       end;
63:     Drive =< 32767, Drive >= 8192 ->
64:       if
65:         Turn =< 57343, Turn >= 32768 -> {146,255,127,255,63};
66:         Turn =< 32767, Turn >= 12288 -> {146,255,63,255,127};
67:         true                         -> {146,255,127,255,127}
68:       end;
69:     true ->
70:       if
71:         Turn =< 57343, Turn >= 32768 -> {146,255,127,0,127};
72:         Turn =< 32767, Turn >= 12288 -> {146,0,127,255,127};
73:         true                         -> {146,0,0,0,0}
74:       end
75:   end.
```

(b) Erlang による記述

図 1　コントローラの記述例 [7], [8]

る演算系列に変換する. リストやタプルの操作はヒープに対するロード/ストア演算に変換する. リスト x[0] (上位 30 ビットがリストの先頭要素のアドレスで, 下位 2 ビットが b'01) の先頭要素のデータ (car) を x[1] に, 次要素 (cdr) を x[2] に格納する命令では, 図 3 (b) のようにヒープからのロードを行う DFG に変換する.

BEAM の命令の中には, メッセージの送受信, ガーベジコレクション等, DFG が大規模になるものが存在する. これらの処理は, 別途独立したハードウェア (以下, ライブラリモジュールと呼ぶ) として実装し, プロセスを実行するハードウェアからサブルーチンのように呼び出せるようにする. ライブラリモジュールは, C 言語で実装された BEAM のインタプリタを縮約したものから ACAP で合成する.

なお, 高位合成系は ACAP に限定されないが, C 言語が処理できることと, バックエンドに CDFG が入力できることが必要になる.

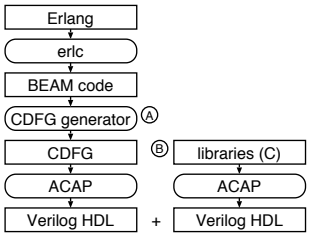
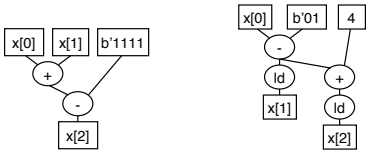


図 2　Erlang からの高位合成の流れ [7], [8]



(a) 加算　　(b) get_list

図 3　BEAM の命令から DFG への変換例 [7], [8]

2.4　ライブラリモジュール

ライブラリモジュールは各プロセスおよびポートに対して一つずつ設けられ, 記憶領域の操作やメッセージの送受信等の複雑な処理を実行する. ライブラリモジュールが実行する機能は以下の 7 つである.

(1) test_heap

ヒープに要求されたワード分の領域を確保する. プロセスの記憶領域に空きがなければガーベジコレクションを行う.

(2) allocate

スタック領域を要求されたワード分増やす. 記憶領域に空きがなければガーベジコレクションを行う.

(3) send

プロセスまたはポート x[0] にメッセージ x[1] を送信する.

(4) receive

カレントメッセージをキューから x[0] にコピーし, メッセージに付随データがあればそれをプロセスのヒープにコピーする. ヒープに十分な空きがなければその前にガーベジコレクションを行う.

(5) remove_message

カレントメッセージをキューから削除する.

(6) save_message

カレントメッセージポインタを一つ進める.

(7) wait_timeout

新しいメッセージを待つ. 指定時間以内に新しいメッセージが来なければ待ち状態を解除する.

ライブラリモジュールは制御変数によってプロセスモジュールから制御される. プロセス *i* のライブラリモジュールの制御変数を RUN_*i* とすると, RUN_*i* が 0 のときはライブラリモジュールは待機状態にある. プロセスモジュールが RUN_*i* に要望する機能の番号を書き込むことによって, ライブラリモジュールは起動し, その処理を行う. 処理が終わると, ライブラリモジュールは RUN_*i* をリセットして待機状態に戻り, プロセスモジュールは続きの処理を行う.

ライブラリモジュールの C プログラム (図 2 中の Ⓑ) は Erlang OTP 18.1.3 の BEAM インタプリタのソースコードを縮約したものである. メッセージキューに関する処理とヒープのコピーに関する処理は BEAM コードを基に, 不要なコードを削除し, 動的割り当てを静的割り当てに変更している. ガーベジコレクションは 2 つの動的領域のマーク・アンド・スイープ方式を 2 つの静的領域割り当てを使う簡単な方法に変更している. プロセスを管理するためのデータ構造やメッセージの送受信処理は新たに実装している. ライブラリモジュールの Verilog HDL 記述はこれらの C プログラムから高位合成により得ている.

2.5 課　　題

文献 [7], [8] の手法で合成されるハードウェアは, 図 4 に示すように, 一つの共有メモリ内にすべての Erlang プロセスの記憶領域を格納している. この構成で本稿で想定する Erlang プロセスの数は 10 程度であるが, 組込みシステムで利用できるメモリのポート数を考えれば非現実的である. また, ハードウェアサイズが大きすぎることも課題としてあげられる. 特に, 高位合成で生成されるライブラリモジュールのサイズが非常に大きくなる.

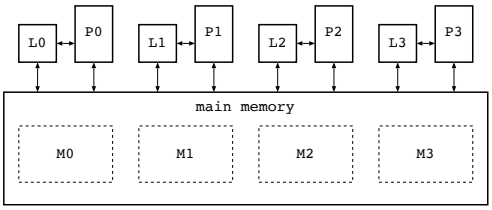


図 4　メモリ共有アーキテクチャ

3. メモリ分散アーキテクチャ

3.1 概　　要

この課題に対し, 本稿ではメモリ分散アーキテクチャを提案する. Erlang のプロセスは, メッセージ送信時以外は各自のメモリしかアクセスしないため, プロセスモジュール毎にローカルなメモリを持たせるのが理にかなっている. ただし, その場合でも, 任意のプロセス間のメッセージ送信を実現するためには膨大な結合網が必要になる. そこで本稿では, 図 5 に示す以下のようなバス結合型のハードウェア構成を採用する.

(1) メモリ分散アーキテクチャ

本手法では, 各プロセス, ライブラリモジュール (P*i*, L*i*) はプロセス毎に独立したローカルなメモリを所有する. そのメモリはさらにヒープ/スタック用の領域 (H*i*) とメッセージキュー/ミニヒープ用の領域 (Q*i*) に分割される. H*i* と Q*i* のメモリポートは一つだけで良い. メッセージ通信やガーベジコレクションの時を除き, すべての P*i* と L*i* は並列にメモリにアクセスできる.

(2) ガーベジコレクションモジュールの共用

本稿では, 文献 [10] の手法に基づき, ガーベジコレクション機能をライブラリモジュールから独立させたガーベジコレクションモジュール (GC) を利用する. ガーベジコレクションを同時に実行出来るのは一つのプロセスのみと制限することにより, すべてのプロセスで一つの GC を共有して, 回路規模削減を図る.

(3) 2 バス構成

他プロセスへのローカルメモリへのアクセスは, Q-bus, H-bus という二本のバスを介して行う. Q-bus は Q*i* へのアクセスを提供し, メッセージ送信に使用する. H-bus は H*i* へのアクセスを提供し, ガーベジコレクションに使用する. バス使用の衝突を避けるため, メッセージ送信とガーベジコレクションはそれぞれシステム中で一組のみ同時に実行できると制限する.

(4) アービタ

アービタモジュール (arbiter) がライブラリモジュールからメッセージ送信とガーベジコレクションのリクエストを受け, これらを優先度に基づいて調停する.

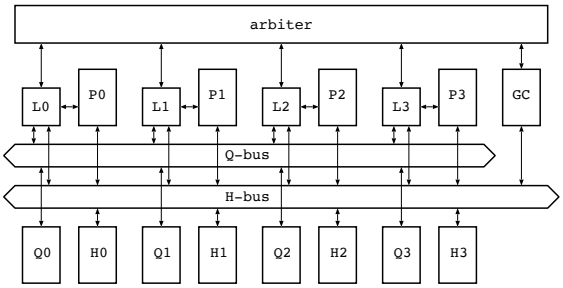


図 5　メモリ分散アーキテクチャ

3.2 メモリ分散とバスモジュール

本システムは, 以下のように動作する.

(1) ローカル記憶への並列アクセス

各プロセスモジュール P*i* は自分のワーキングメモリ H*i* にアクセス可能である. 図 6 (a) では, P0 と P1 はそれぞれ H0 と H1 に同時にアクセスしてローカルな計算を並列に実行している. 各ライブラリモジュール L*i* はメッセージキュー Q*i* と H*i* にアクセス可能である. L2, L3 は receive 命令を処理しているが, これは P0, P1 のローカルな計算と並列に実行できる.

(2) メッセージ送信

プロセスモジュール P*i* から P*j* にメッセージを送るとき, ライブラリモジュール L*i* はワーキングメモリ H*i* からメッセージに関連するデータを読み, 送信先のメッセージキュー Q*j* に書き込みを行う. このとき, データは Q-bus を介して送信する. 図 6 (b) は, P2 から P1 へのメッセージ送信を表している. L2 は H2 内のデータを読み, 送信先のメッセージキューがある Q1 に書き込みを行う. この間も, P0 と P3 のローカルな計算は並列に処理が可能である. さらに, P1 は Q1 にアクセスしなければローカルな処理を同時に実行することができる.

(3) ガーベジコレクション

P*i* がガーベジコレクションをリクエストすると, ガーベジコレクションモジュールが H-bus を通して H*i* のガーベジコレクションを行う. 図 6 (c) では, GC が P1 のガーベジコレクションを行っている. このとき, P1 以外のプロセスは同時に処理を行うことができる. また, P2 から P1 のメッセージ送信も可能である.

本手法では, システム全体に一つの 32 ビット アドレス空間を割り当て, 各 H*i* や Q*i* はアドレス空間のセグメントに配置する. アドレスの下位 *m* ビットはローカルメモリのアドレス, 上位 32 – *m* ビットはセグメント番号を表すものとし, 他のプロセ

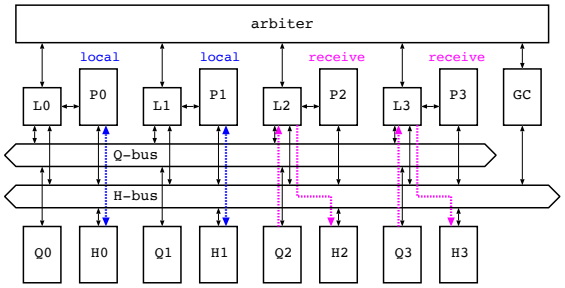
スのメモリへのアクセスはバスモジュールが担当する．

Li から Qi と Qj ($j \neq i$) への read アクセスは以下のように扱う．

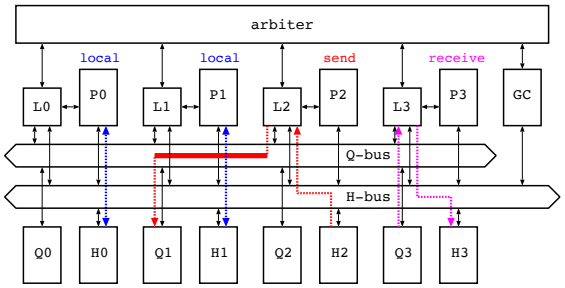
- Li からアクセスがあり，アドレスのセグメント番号が i のとき，Qi にアドレスの下位 m ビットを送り，読み出されたデータを Li に返す．
- Li からアクセスがあり，アドレスのセグメント番号が j のとき，リクエストは Q-bus に流され，Qj からのデータが Li に返される．
- Q-bus にセグメント番号が i のアクセスが流されると，リクエストは Qi が受け取り，Qi の読み出し結果はバスに流される．

2 つのライブラリモジュールが Q-bus に同時にアクセスすることはなく，さらに，Li と Lj が同時に Qi にアクセスすることはない．これらはアービタの調停により保証されている．

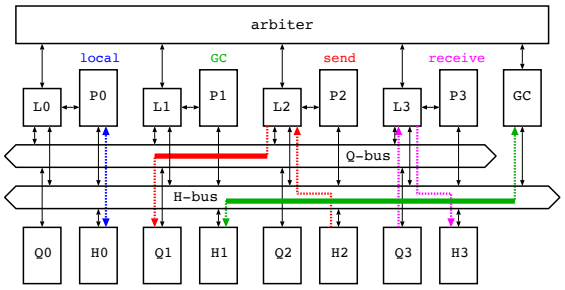
Hi へのアクセスも Qi と同様の方法で処理される．



(a) ローカルな計算



(b) メッセージ送信命令



(c) ガーベジコレクション命令

図 6　メモリ分散アーキテクチャの動作例

3.3　アービタ

アービタモジュールは，プロセスから送られるメッセージ送信リクエストとガーベジコレクションリクエストの調停を行う．複数のリクエストが同時に来たときには，優先度に従って実行許可を与えるリクエストを決定する．メッセージ送信やガーベジコレクションの処理中に新しいリクエストが来たときは，現在の処理が終わってから新しいリクエストを含む待機中のリクエストの中で最も優先順位が高いものに実行許可を与える．

ガーベジコレクションリクエストの処理手順は以下の通りである．

(1) ライブラリモジュール Li がガーベジコレクションリクエストを送信するには，ガーベジコレクションリクエスト信号 $\mathrm{GC\_req}_i$ を 1 にする．

(2) $\mathrm{GC\_req}_i$ が 1 になると，アービタが GC_req を 1 にし，リクエスト元のプロセス番号を表す GC_process を i にする．複数プロセスから同時にリクエストが来た場合には，優先度に応じてそのうちの一つを選択する．

(3) ガーベジコレクションモジュール GC は Hi に対してガーベジコレクションを実行する．処理が終了したら GC_req を 0 にする．

(4) アービタは $\mathrm{GC\_req}_i$ を 0 にすることにより Li に処理終了を通知する．

(5) Li は $\mathrm{GC\_req}_i$ が 0 になってると確認でき次第，続きの処理を行う．

メッセージ送信リクエストの処理は，送信先プロセスがメッセージキューに対する処理を行っているときにはメッセージを送信できないため，ガーベジコレクションリクエストの処理より少し煩雑になる．メッセージ送信リクエストの処理手順は以下の通りである．

(1) Li はメッセージ送信リクエスト信号である $\mathrm{send\_req}_i$ に 1，送信先の宛先を表す信号である send_to に j を書き込むことにより，プロセス j へのメッセージ送信リクエストを送信する．

(2) アービタはプロセス j に対し，エンキューリクエスト信号 $\mathrm{enq\_req}_j$ を 1 にし，エンキューリクエストを送る．このとき複数のメッセージ送信リクエストがあれば優先度に従って一つを選択する．

(3) プロセス j が receive_message，remove_message，save_message を実行していればその完了を待って，していなければ直ちに，新しいメッセージのキューへのつなぎこみを許可するため，$\mathrm{enq\_ready}_j$ に 1 を書き込む．

(4) アービタは $\mathrm{send\_req}_i$ を 1 にすることにより Li にメッセージ送信許可を与える．

(5) Li は $\mathrm{send\_req}_i$ が 1 になったことを確認し，データを Qj に送信する．

(6) Li の処理が終了したら $\mathrm{send\_req}_i$ を 0 にし，リクエストを取り下げる．

(7) アービタは $\mathrm{send\_req}_i$ が 0 になったことを確認したら $\mathrm{enq\_req}_j$ を 0 にする．

ただし，Lj は send リクエストの許可を待ってる間にもエンキューのリクエストを許可できるように設計する必要がある．

3.4　I/O モジュール

$\mathrm{GC\_req}_i$ や $\mathrm{enq\_ready}_j$ 等の信号はアドレス空間内に配置し，メモリマップト I/O によって，各プロセス，ライブラリ，GC モジュールからアクセスできるようにする．このアクセスを制御するのが I/O モジュールである．図 5 に I/O モジュールを追加したハードウェア構成が図 7 である．各モジュールは I/O モジュールを介してバスに接続する．Li からのアクセスが Hi へのアクセスか Qi へのアクセスかは I/O モジュールが判断し，出力先を切り替える．

4.　実装と実験

提案手法に基づき高位合成系のプロトタイプを実装した．処理系は Ubuntu Linux と Mac OS X 上で動作する．BEAM アセンブリから CDFG を生成する処理系 (図 2 中の Ⓐ) は Perl5 で実装し，CDFG 内の命令は ACAP の 32 ビット演算器を想定して

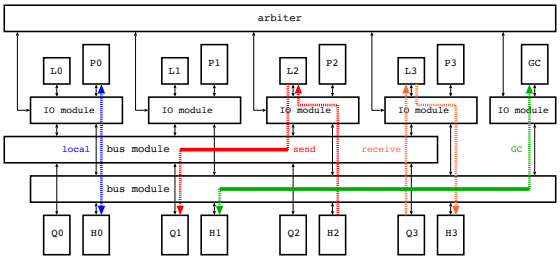


図 7　I/O モジュールを含むメモリ分散アーキテクチャ

いる.

図 1 のシステムを合成すると図 8 のようなハードウェア構成となる. iport0 は図 1 (a) の port0 のライブラリモジュールである. oport1 は図 1 (a) の port1 のライブラリモジュールであり, P0 からのメッセージ送信リクエストがあったときにキューへのつなぎこみを許可する信号を与える機能のみを有する.

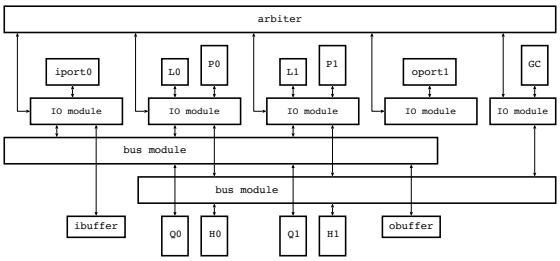


図 8　合成対象となる Erlang プログラムのハードウェア構成

本システムで Verilog HDL を生成し, FPGA を対象に論理合成した結果を表 1 に示す. “LUTs” は LUT 数, “FFs” はフリップフロップ数, “delay” はクリティカルパス遅延を表す. 論理合成は Artix-7 (xc7a100tcsg324-3) をターゲットに Vivado (2015.4) で面積最適化オプションを指定して行った. 参考のため, MIPS R3000 ソフトコア [11] の合成結果を “MIPS” 行に記載している. バスモジュール, アービタモジュール, I/O モジュールは手設計したものであり, いずれも小さく高速である. ライブラリモジュールは MIPS に比べると非常に大きい. これは, 現時点ではまだ十分なアーキテクチャの最適化を行っていないためであり, 回路規模の削減は今後の課題である.

表 1　合 成 結 果

	LUTs	FFs	delay [ns]
bus	245	0	5.030
arbiter	169	180	3.460
I/O	415	128	7.235
P0	4,520	1,010	10.757
P1	6,007	888	12.171
GC	5,968	1,349	12.003
L0	9,163	1,323	14.441
L1	8,994	1,386	13.846
iport0	3,646	683	11.133
oport1	539	138	5.688
MIPS	3,070	1,771	13.037

論理合成: Vivado 2015.4, ターゲット: Artix-7

ライブラリモジュールは C 言語のレベルでコンピュータ上で動作確認後に高位合成を行った. 手設計のモジュールは Verilog HDL で記述後, 単体テストを実施した. GC モジュールは, C 言語のレベルでコンピュータ上で動作確認後に高位合成を行ったが, 合成後のハードウェアの動作確認はまだできていない.

RT レベルシミュレーションで確認した本システムの処理時間を表 2 に示す. “cycle” は入力ポートがデータを受け取ってから出力ポートに値を出力するまでのサイクル数であり, “time” はそれにクリティカルパス遅延を乗じた値である. ただし, シミュレーションに用いたメモリは非同期型であり, メモリアクセスは 1 クロックで出力を完了するモデルである. またこのシミュレーション中にガーベジコレクションは発生していない.

表 2　処 理 時 間

cycle	time [μs]
2,471	35.684

5. む　す　び

本稿では, Erlang からの高位合成のためのメモリ分散アーキテクチャを提案した. 簡単な Erlang プログラムから Verilog HDL コードを生成する合成系のプロトタイプを実装し, 合成により得られた回路の動作確認をシミュレーション上で行った. 今後の課題は, GC モジュールのシミュレーション, 回路規模の削減, および回路構成の最適化が挙げられる.

謝　　辞

本稿の研究にあたり, 有益な御助言を頂いた立命館大学の冨山宏之教授, 元立命館大学の中谷嵩之氏, 元関西学院大学の竹林陽氏に感謝いたします. また, 本稿に関してご協力, ご討議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. 本研究は一部 JSPS 科研費 16K00088 および 16K01207 の助成による.

文　　献

[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers (1992).

[2] 本田晋也, 冨山宏之, 高田広章, “システムレベル設計環境：SystemBuilder,” 信学論, Vol. J88-D-I, No. 2, pp. 163–174 (Feb. 2005).

[3] 田村真平, 石浦菜岐佐, 神原弘之, 冨山宏之: “CPU 密結合型アクセラレータの機械語プログラムからの自動合成,” 信学技報, VLD2013-133 (Jan. 2014).

[4] Joe Armstrong 著, 榊原一矢訳: プログラミング Erlang, オーム社 (2008).

[5] 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).

[6] Brian Chamberlain: Using Erlang on the RaspberryPi to interact with the physical world (online), `http://www.slideshare.net/breakpointer/using-erlang-on-the-raspberrypi` (accessed 2016-02-04).

[7] 竹林陽, 石浦菜岐佐, 東香実, 吉田信明, 神原弘之: “Erlang による組込みシステムの制御記述からの高位合成,” 信学技報, VLD2015-114 (Feb. 2016).

[8] H. Takebayashi, N. Ishiura, K. Azuma, N. Yoshida, and H. Kanbara: “High-Level Synthesis of Embedded Systems Controller from Erlang,” in *Proc. SASIMI 2016*, R4-2, pp. 285–290 (Oct. 2016).

[9] N. Ishiura, H. Kanbara, and H. Tomiyama: “ACAP: Binary Synthesizer Based on MIPS Object Codes,” in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).

[10] 浜名将輝, 石浦菜岐佐, 吉田信明, 神原弘之: “Erlang からの高位合成のためのライブラリモジュールの回路規模削減,” 信学ソ大, A-6-2 (Sept. 2017).

[11] 神原弘之, 金城良太, 戸田勇希, 矢野正治, 小柳滋: “パイプラインプロセッサを理解するための教材 RUE-CHIP1 プロセッサ,” 情処関西支部大会, A-09 (Sept. 2009).