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
VLD2015-114(2016-02)

Erlang による組込みシステムの制御記述からの高位合成

竹林　陽[†]　石浦菜岐佐[†]　東　香実[†]　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 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する. 本手法では, 複数の Erlang プロセスとその間のメッセージ通信に基づいてシステムの動作を記述し, Erlang の 1 プロセスを 1 つのハードウェアモジュールで実行するハードウェア構成を想定して合成を行う. Erlang のプログラムをコンパイルして得られる仮想マシン BEAM のアセンブリを CDFG (control dataflow graph) に変換し, これを高位合成システム ACAP のバックエンドに入力して Verilog HDL によるレジスタ転送レベルの設計記述を生成する. メッセージの送受信処理やガーベジコレクションの処理は, BEAM インタプリタの C 言語による実装を簡略化したものから ACAP で合成する. 提案手法に基づく合成の処理系を Perl で実装し, 2 プロセスからなる簡単な制御記述から論理合成可能な Verilog 記述を生成することができた.

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

High-Level Synthesis of Embedded Systems Controller from Erlang

Hinata TAKEBAYASHI[†], Nagisa ISHIURA[†], Kagumi AZUMA[†], 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 article presents a method of specifying the behavior of embedded systems' control by a subset of Erlang and synthesizing hardware from the specification. In this method, the systems' behavior is modeld so that events are processed by concurrent processes with message passing. Assembly codes of the BEAM virtual machine compiled from Erlang programs are converted into CDFGs (control dataflow graphs), which are synthesized into Verilog HDL by the backend of the high-level synthesizer ACAP . Complex routines to handle message passing and garbage collection are synthesized by the ACAP from reduced C implementation of the BEAM interpreter. A prototype system based on the proposed method implemented in Perl has successfuly synthesized a simple two-process Erlang description into logic-synthesizable Verilog codes.

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, 自動車, 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, 一般にプロセッサ上で動作する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, ソフトウェアで実装されている機能の全部あるいは一部をハードウェアで実装することによって電力性能比の向上が図られる. このようなシステムを効率的に設計するために, 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種々提案されている [2], [3].

近年の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広がりに伴い, 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が今後重要な課題になると考えられる.

組込みシステムは, 外部からのイベント入力に対応して処理を実行するリアクティブシステム (reactive system) の一種と見なすことができる. 多種のイベント入力に対応した複雑な制御を効率的に実装するためには, 並行プロセスや割り込みを活用した機能の記述が必要になる. リアルタイム OS はこのようなシステムの実装の複雑さを低減するが, 割り込み処理の記述や応答時間の保証には高度な技能が必要となる.

この問題を解決する一つのアプローチとして, Erlang [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を用いてシステムの制御を記述することが考えられる. 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が [5] , 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になると考えられる.

そこで本稿では, 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する. 本手法では, 組込みシステムへのイベント入出力は Erlang のポートを介して行い, 複数の Erlang プロセスの連携によりイベントの処理を行うというモデルで制御を記述する. Erlang の 1 プロセスは独立した 1 つのハードウェアモジュールに合成する. Erlang のプログラムをコンパイルして得られる仮想マシン BEAM のアセンブリを CDFG (control dataflow graph) に変換し, これを高位合成システム ACAP [7] のバックエンドに入力して Verilog HDL によるレジスタ転送レベルの設計記述を生成する. メッセージの送受信処理やガーベジコレクションの処理は, BEAM インタプリタの C 言語記述を簡略化したものから ACAP で合成する.

提案手法に基づく合成の処理系を Perl で実装し, 2 プロセスからなる簡単な制御記述から論理合成可能な Verilog 記述を生成することができた.

2. Erlang と高位合成

2.1 Erlang

Erlang は, 並行処理指向の関数型言語である. Erlang では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.

Erlang プログラムの例を 図 1(a) に示す. このプログラムでは, 2 つのリストの内積を求める関数 `iprod` を定義している. 小文字から始まる英数字列は関数名またはアトムと呼ばれる不変定数を表し, 大文字から始まる英数字列は変数を表す. "->" は関数の定義を表す. Erlang の変数は, 代入が 1 回限りの単一代入変数である. Erlang ではデータ型として整数 (多倍長) と浮動小数点数, およびそのタプル (要素数が固定; { } で囲まれている) やリスト (要素数が可変; [ ] で囲まれている) が用意されている.

プロセスの生成は `spawn`, メッセージの送受信は ！ 演算と `receive` により記述する. 図 1(b) は, 円または長方形の面積を計算して返すプロセスの記述例である [4]. `start` を呼び出すと, 5 行目の `spawn` により 7–18 行目の `loop` を実行するプロセスを生成し, そのプロセスの ID を返す. 20 行目の `area` は, このプロセスの ID (`Pid`) とデータ (`Request`) を受け取って面積を返す. 21 行目の！演算はメッセージの送信であり, プロセス `Pid` にタプル {`self()`, `Request`} を送ることを意味する. ここでは, 計算結果を返送してもらうために, 自身のプロセス ID (self()) をデータと一緒に面積計算プロセスに送っている. メッセージの受信は, 9–17 行目および 22–24 行目のように `receive` で記述し, 届いたメッセージのパターンにマッチする処理を行う. 受信にタイムアウトを指定することもできる.

メッセージの送信は図 1(b) の例のようにプロセスを指定して行われる. 各プロセスは 1 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につながれる. プロセスは, パターンマッチに従って選択的にキューからメッセージを取り出す.

```
1: -module(iprod).
2: -export([iprod/2]).
3:
4: iprod(A,B) -> iprod3(A,B,0).
5:
6: iprod3([],[],X) -> X;
7: iprod3([AH|AT],[BH|BT],X) -> iprod3(AT,BT,AH*BH+X).
```

(a) リストの内積計算

```
01: -module(area_server).
02: -export([start/0, area/2]).
03:
04: start() ->
05:   spawn(fun loop/0).
06:
07: loop() ->
08:   receive
09:     {From, {rectangle, Width, Ht}} ->
10:       From ! {self(), Width * Ht},
11:       loop();
12:     {From, {circle, R}} ->
13:       From ! {self(), 3.14159 * R * R},
14:       loop();
15:     {From, Other} ->
16:       From ! {self(), {error, Other}},
17:       loop()
18:   end.
19:
20: area(Pid, Request) ->
21:   Pid ! {self(), Request},
22:   receive
23:     {Pid, Response} ->
24:       Response
25:   end.
```

(b) 面積を計算するサーバ [4]

図 1　Erlang プログラムの例 [4]

```
01: {function, iprod, 2, 2}.
02:   {label,1}.
03:     {func_info,{atom,iprod},{atom,iprod},2}.
04:   {label,2}.
05:     {move,{integer,0},{x,2}}.
06:     {call_only,3,{f,4}}.
07:
08: {function, iprod3, 3, 4}.
09:   {label,3}.
10:     {func_info,{atom,iprod},{atom,iprod3},3}.
11:   {label,4}.
12:     {test,is_nonempty_list,{f,5},[{x,0}]}.
13:     {get_list,{x,0},{x,3},{x,4}}.
14:     {test,is_nonempty_list,{f,3},[{x,1}]}.
15:     {get_list,{x,1},{x,5},{x,6}}.
16:     {gc_bif,'*',{f,0},7,[{x,3},{x,5}],{x,0}}.
17:     {gc_bif,'+',{f,0},7,[{x,0},{x,2}],{x,2}}.
18:     {move,{x,6},{x,1}}.
19:     {move,{x,4},{x,0}}.
20:     {call_only,3,{f,4}}.
21:   {label,5}.
22:     {test,is_nil,{f,3},[{x,0}]}.
23:     {test,is_nil,{f,3},[{x,1}]}.
24:     {move,{x,2},{x,0}}.
25:     return.
```

図 2　図 1(b) の BEAM アセンブリコード

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 (注1). 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 を持ち, そのインタプリタは C 言語で実装されている. 図 2 は図 1(a) をコンパイルして得られる BEAM のアセンブリコードである.

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

(注1)：ネイティブコードへのコンパイラ HiPE も存在するが, 対応するプラットフォームは限定される.

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るが, この際, 不足量に応じて記憶領域の拡張が行われる.

BEAM の命令は約 150 種類あり, 次の 5 つに大別できる.

- 算術演算やビット演算を行う組込み関数を呼び出す命令
- リストおよびタプルの操作を行う命令
- ジャンプ命令および関数呼び出し命令
- ヒープおよびスタックの領域確保やガーベジコレクションを行う命令
- メッセージの送受信を行う命令

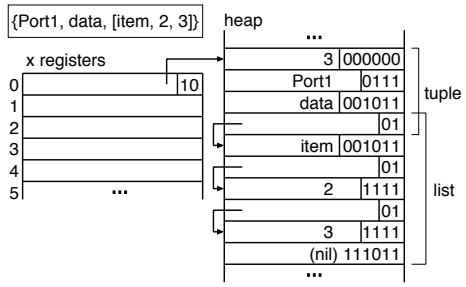


図 3　リストとタプルの表現

Erlang のデータは 1 ワード 32 ビットの「Eterm」によって表現される. 図 3 は, x[0] レジスタに {Port1, data, [item, 2, 3]} というデータが格納されている場合のヒープの状態である. Eterm の下位ビットにはそのワードの型を表すタグが格納されている. 3 要素のタプルは, そのサイズと各要素の Eterm を格納する 4 ワードで表現される. リストの各要素は car と cdr の 2 ワードで表現される. 28 ビットまでの整数は, 下位 4 ビットを b'1111 とする 1 ワードで表現される.

各プロセスのメッセージキューは線形リストで実装されている. 1 ワードの Eterm をメッセージとして送信する場合は, メッセージの構造体を一つ割り当て, 受信側のプロセスのキューの末尾に接続する. メッセージデータ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. ただし, 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「heap fragment」と呼ばれるミニヒープを割り当てて一旦そこにコピーし, 受信側が receive を実行するタイミングでこれを受信側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, receive を実行すると, 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.

ポートに入力された n バイトのデータは, n 要素のリストとしてプロセスに送信する.

2.3　高位合成システム ACAP

ACAP は C プログラムおよび MIPS R3000 の機械語プログラムを入力として, レジスタ転送レベルのハードウェアを合成する高位合成システムである. ACAP の合成処理の流れを図 4 に示す. ACAP は GCC または GAS (アセンブラ) によって生成される機械語を逆アセンブルして得られるアセンブリコードに対して字句解析や構文解析を行い, CDFG (control data flow graph) を生成する. CDFG に対して, 最適化, スケジューリング, バインディング処理を行い, Verilog HDL を出力する.

3. Erlang による組込みシステム制御の記述

本手法では, システムの動作を複数の Erlang プロセスを用いて記述する. ただし, 全てのプロセスはシステムの起動時に生成し, 実行時に新たなプロセスの生成やプロセスの削除は行わないものとする.

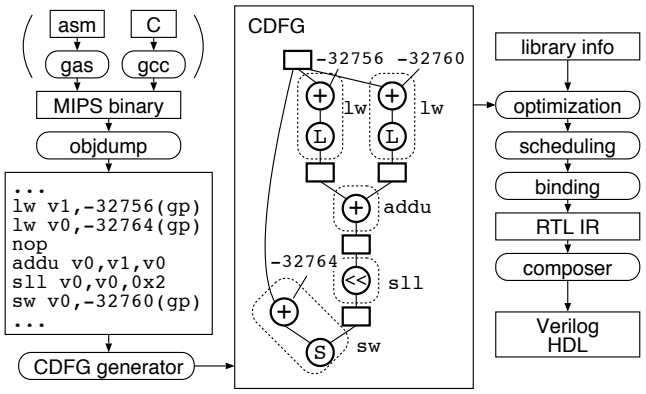


図 4　高位合成システム ACAP の処理の流れ [7]

システムへの入出力は 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ものとする. イベントに対する処理はメッセージの授受に基づく計算により表現する. なお, 本稿では, メッセージによる通信はシステム内のプロセスとポート間でのみ行われるものとする. すなわち, 本稿では, 記述対象のシステムが, TCP/IP によって外部の Erlang プロセスと通信することは想定しない. メッセージの送信先 (! 演算の左辺) は, 実行時の式の評価結果により決定されても良いものとする. 本稿で使用するデータ型は 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.

以上の Erlang サブセットによってごく簡単なコントローラを記述した例を図 5 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ものである.

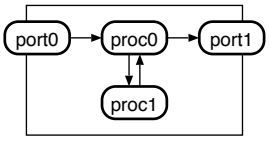
コントローラは図 5(a) のようなポートとプロセスでモデル化するものとする. `port0` がボタンからの入力を受信し, 制御信号を `port1` に出力する. ボタン入力データの駆動制御信号データへの変換はプロセス `proc1` が行う. プロセス `proc0` は, `port0` に入力があると, そのデータを `proc1` に送って制御信号に変換したものを受け取り, それを `port1` に出力する.

図 5(b) 中, 4 行目の `start` がシステムを起動する関数であり, 5 行目でプロセス `proc1` を, 9 行目でプロセス `proc0` を生成している. `proc1` は本体 `loop1` (34–44 行目) を実行する. `proc0` は, 11 行目と 12 行目でそれぞれポート `Port0` と `Port1` を開き, 本体 `loop0` (19–32 行目) を実行する. `proc0` は `Port0` から制御の変換結果をメッセージとして受け取るとデータを整形して `proc1` に転送する (21–24 行目). また, `proc0` は `proc1` からメッセージを受け取ると, それを `Port1` に転送する (25–27 行目). それ以外のプロセスおよびポートからのメッセージは削除する. `proc1` は `proc0` からデータを受け取ると制御データの変換を行い, `proc0` に返送する. それ以外のプロセス及びポートからのメッセージは, そのまま `proc0` に転送する.

4. Erlang からの高位合成

4.1　概　　要

本稿では, 前章で述べた Erlang のサブセットによるシステムの制御記述から, レジスタ転送レベルのハードウェア記述を合成する手法を提案する. 本手法では, Erlang の 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の実行の影響を受けずに実行できる. 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



(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 5　コントローラの記述例

バヘッドもなくなる. ただし本稿では, ハードウェアが使用するデータはすべて 1 つのメモリに置かれるハードウェア構成を想定する.

Erlang のプロセスは 1 つ以上の関数を実行する. 本稿では, 各プロセスに対し, そのプロセスが実行する可能性のある全ての関数 (静的解析により決定する) を 1 つのハードウェアモジュールに合成する. ある関数が複数のプロセスにより実行される可能性がある場合は, その関数を複製する. 例えば, 図 6 では, 起動時に 2 つのプロセス (proc0, proc1 とする) が生成され, それぞれ関数 f0, f1 を呼び出している. f0 からは f2 が, f1 からは f2, f3, f4 が呼び出されている. この場合, proc0 に対するハードウェアモジュール HW0 は f0 と f2 を, proc1 に対するハードウェアモジュール HW1 は f1, f2, f3, f4 を実行できるようにする. すなわち, 関数 f2 は両方のハードウェアで実行できるようにする.

なお, 最初にプロセスを起動するプロセス (図 6 の start) は合成対象とはしない.

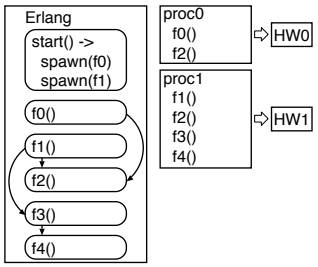


図 6　Erlang プロセスのハードウェア化

本稿の合成手法の流れを図 7 に示す. 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 のバックエンドに入力して Verilog HDL を得る. ただし, 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, データフローグラフが大規模になるものが存在する. 本稿では, 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.

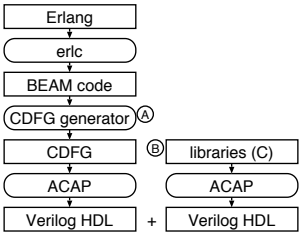


図 7　本手法の合成処理の流れ

4.2　BEAM アセンブリから CDFG への変換

コンパイルの結果得られる BEAM アセンブリを解析し, プロセス毎に CDFG を 1 つ生成する. まず, システム全体を起動する関数を解析して, 全てのプロセスを同定する. 次に, 関数の呼び出し関係を解析して, 各プロセスが実行する可能性のある関数を列挙する. 複数のプロセスから呼び出される可能性のある関数は複製する. 各関数はラベルや分岐命令に基づいて基本ブロックに分割する. 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) を構築した後, 基本ブロック間の制御の流れに基づいて CDFG を構築する.

BEAM の命令は以下のように CDFG の演算に変換する.

- 算術演算, ビット演算

Erlang の 算術演算, ビット演算は, 組込み関数を実行する BEAM 命令 (`gc_bif` 命令) にコンパイルされているので, それを CDFG の演算に変換する. 本稿が対象とする 28 ビットデータは, 下位 4 ビットのタグ (b'1111) とともに 32 ビットのレジス

タに格納されているので, これらのデータに対する演算は, データの上位 28 ビットを操作する演算系列に変換する. 例えば,

```
{gc_bif, '+', {f,0}, 0, [{x,0}, {x,1}], {x,2}}.
```

は, レジスタ x[0] と x[1] の加算結果を x[2] に格納する演算であるが, これは, 図 8(a) のような DFG に変換する. DFG の演算はすべて 32 ビットデータに対するものである. 加算に関しては, x[0] と x[1] の 32 ビット加算結果から下位 4 ビットのタグ b'1111 を減じれば, 上位 28 ビットの加算結果とタグが x[2] に書き込まれる.

なお, 本稿では ACAP の 32 ビット演算器をそのまま使うことを想定しているため, このようなビット操作が必要になるが, BEAM の 28 ビットデータを直接扱う演算器を準備すれば, その必要はなくなる.

- リスト, タプル操作演算

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. 例えば,

```
{get_list, {x,0}, {x,1}, {x,2}}.
```

は, 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] に格納する命令であるが, これは, 図 8(b) のようにヒープからのロードを行う DFG に変換できる.

- ジャンプ, コール

分岐命令は, DFG 間の遷移関係に変換する. 例えば,

```
{test, is_noempty_list, {f,4}, {x,0}}.
```

は, リスト x[0] が空であれば次の命令を実行し, 偽であればラベル f4 にジャンプする命令であるが, これは, 図 8(c) のように条件付きジャンプを行う DFG に変換する. また,

```
{call, 1, {f,2}}.
```

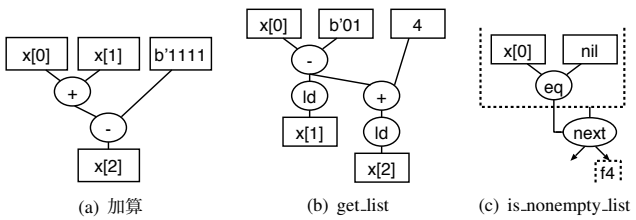
は, 戻り先番地を CP レジスタに格納し, ラベル f2 にジャンプする命令であるが, 命令番号を CP レジスタに格納し, ラベル f2 にジャンプする DFG に変換する. なお, return 命令では, 命令番号と状態の表を作っておき, CP レジスタの値を用いて呼び出し元へと遷移する.

- ヒープ, スタックの操作関係

ヒープやスタックに領域を確保する命令は, 領域が確保できなかった場合にガーベジコレクションが必要になるため, 単純な DFG には変換できない. これらの命令は, 次節で述べるライブラリモジュールを起動して処理するので, 1) 引数のストア, 2) ライブラリモジュールを起動するための変数へのストア, 3) ライブラリモジュールの終了を待つポーリング, および 4) 結果のロードを行う DFG に変換する.

- メッセージの送受信

プロセスの動的な生成・削除がないため, プロセスの ID は静的に決定している. また, 外部の Erlang プロセスとの通信がないため, 全てのアトムも ID が静的に決定している. ただし, メッセージ送受信も, 複雑な処理やガーベジコレクションを必要とするため, ライブラリモジュールの起動を行う DFG に変換する.



(a) 加算 (b) get_list (c) is_nonempty_list

図 8 BEAM の命令からデータフローグラフへの変換例

プロセスの CDFG を生成する際には [8] と同様の手法で関数併合を行う. インタフェース DFG で関数起動用の変数を監視し, 書きこまれた値に対応する関数を実行する.

4.3 ライブラリモジュール

記憶領域の操作やメッセージの送受信等の複雑な処理は, ライブラリモジュールを呼び出して実行する.

本稿の合成手法では, 1 つのプロセスに対して 1 つのライブラリモジュールを, また, 1 つの出力ポートに対して 1 つのライブラリモジュールを設ける. プロセスのライブラリモジュールは, 当該プロセスの記憶領域を読み書きする他, そのプロセスにメッセージを送る全てのプロセスの記憶領域, および入力ポートのバッファを参照する. 出力ポートのライブラリモジュールは, そのポートにメッセージを送る全てのプロセスの記憶領域を参照し, 出力バッファへの書き込みを行う. 例えば, 図 5(a) の構成の記述から合成されるハードウェアの接続関係は, 図 9 のようになる. プロセス proc0 と proc1 はそれぞれの記憶領域 mem0, mem1 を持つ. 入力ポート port0 に入力されたデータは buff0 に, 出力ポート port1 から出力されるデータは buff1 に格納されるとする. plib0 と plib1 はそれぞれ proc0, proc1 のライブラリモジュールであり, tlib1 が port1 のライブラリモジュールである. plib0 は, mem0 を読み書きするとともに, proc0 にメッセージを送る proc1 の記憶領域 mem1 と, port0 のバッファ buff0 を参照している. tlib1 は, mem0 を参照して, buff1 に値を書き込む.

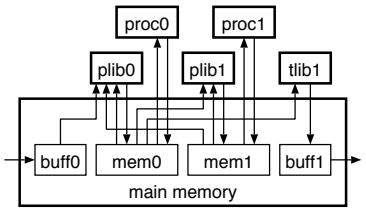


図 9 プロセスモジュールとライブラリモジュールの接続関係

表 1 ライブラリモジュールの機能一覧

no.	name	引数	返り値
1	send	x[0], x[1]	-
2	allocate	need, x[], n, htop, heap, sp	x[], n, htop, heap, sp
3	test_heap	need, x[], n, htop, heap, sp	x[], n, htop, heap, sp
4	receive	htop, heap, sp	x[], htop, heap, sp
5	remove_message	-	-
6	save_message	-	-

ライブラリモジュールが実行する機能の一覧を表 1 に示す. 詳細は下記の通りである.

(1) test_heap

ヒープに need ワードの領域を確保する. プロセスの記憶領域に need ワードの空きがあれば何もしないが, なければガーベジコレクションを行う. ガーベジコレクションのルートとなる x レジスタがあれば, その個数と値 n, x[0], x[1], ⋯, x[n-1] を授受する. ヒープのアドレスはガーベジコレクションにより変更されるので, ヒープの先頭, ヒープのトップ, スタックポインタの値をそれぞれ heap, htop, sp で授受する.

(2) allocate

スタック領域を need+1 ワード増やす (sp を更新する). test_heap と同様, 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

x[0] (プロセスまたはポート) にメッセージ x[1] を送信する.

ただし, 本稿の手法では, メッセージのキューへのつなぎ込みやヒープのコピーは受信側のライブラリモジュールが行う. こ

のため, send の処理では, 受信側のプロセス/ポートにメッセージがあったことを示す着信フラグのセットのみを行う. 着信フラグは, 送信プロセス/ポートと受信プロセス/ポート の組み合わせに対して 1 ビット準備する. 送信先は実行時に判明するので, x[0] の値に応じた着信フラグを選択してセットする.

受信側のライブラリモジュールは, 着信フラグをポーリングしており, 着信があると直ちにメッセージのキューへのつなぎ込みとデータのミニヒープへのコピーを行う. 複数のプロセス/ポートからメッセージを受信した場合は, 順次処理する.

受信側は処理が完了するとフラグをリセットする. 送信側はフラグがリセットされると, プロセスに制御を戻す.

(4) receive

カレントメッセージをキューから x[0] にコピーする. メッセージがヒープデータを持っていれば, それをプロセスのヒープにコピーするが, プロセスの記憶領域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, save_message

remove_message はパターンがマッチしたカレントメッセージをキューから削除する処理であり, save_message はパターンがマッチしなかった場合にカレントメッセージポインタを 1 つ進める処理である.

ライブラリモジュールは, ポーリングによりプロセスから制御する. プロセス i を制御する変数を RUN_i とする. RUN_i はメモリ上に置くか, メモリにマップされたレジスタに置く. RUN_i の値が 0 ときにはライブラリモジュールは待機中であり, プロセスは, 引数をメモリに書き込んだ後で, ここに表 1 の機能番号を書き込む. ライブラリモジュールは RUN_i に書き込まれた番号の機能を実行し, 処理が完了すると返り値をメモリに書き込んで, RUN_i に 0 を書き込む. プロセスは, RUN_i が 0 になると, 処理を再開する.

5. 実　　装

提案手法に基づく高位合成系のプロトタイプを実装した. 処理系は Linux 上で動作する.

BEAM アセンブリから CDFG を生成する処理系 (図 7 中の A) は Perl5 で実装した. 今回の実装では, CDFG 中の演算は ACAP の 32 ビット演算器を想定したものを生成した.

ライブラリモジュール (図 7 中の B) の C 言語記述は, Erlang OTP 18.1.3 中の BEAM インタープリタのソースコードから必要な部分を抽出して縮約することにより得た. メッセージキューの処理, およびヒープのコピーの処理は, BEAM のコードをほぼそのまま流用したが, 今回の実装に不要な部分は削除し, 動的割り当てはすべて静的割り当てに書き換えた. BEAM のガーベジコレクションは, 2 世代の領域を持つマークアンドスイープ方式で, ガーベジコレクションの度に新しいサイズの領域を割り当てる (古い領域は破棄する) 実装であるが, 本手法では, 世代管理を行わず, 静的に割り当てた 2 つの固定サイズの領域を交互に用いる実装に書き換えた. 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, 今回の仕様に沿った簡易なものを一から実装した. 入力ポートのデータは, unsigned char 型配列のバッファに置かれるものとし, ここにデータが書き込まれると, これを Erlang のオブジェクトデータに変換し, メッセージキューに接続する処理を C 言語で実装した. これらの C プログラムは, PC 上で動作確認を行った後, ACAP で Verilog HDL に合成した.

図 5(b) のプログラムから本システムで Verilog HDL を生成し, それを論理合成した結果を 2 に示す. 論理合成は Xilinx ISE 14.3 により, FPGA Xilinx Spartan6 をターゲットとして行った. 表の LUTs, FFs, Delay はそれぞれ合成結果の LUT 数, FF 数, クリティカルパス遅延である.

プロセスのモジュール (proc0 および proc1) の回路規模は, BEAM アセンブリの規模にほぼ比例すると考えられるので, 例題の処理内容を考えると回路規模は大きいと考えられる. ライブラリモジュール (plib0, plib1, tlib1) の回路規模は, プロセスの処理内容に関わらずほぼ一定であるが, 現状では大きいと考えられる. この回路規模の削減が今後の課題の一つである.

表 2　図 5(b) の記述の合成結果

	LUTs	FFs	Delay [ns]
proc0	3,835	686	18.929
proc1	4,380	451	18.929
plib0	24,643	1435	21.414
plib1	23,916	1384	21.468
tlib1	24,405	1403	21.541

また, 本稿の実装では, 全プロセスの記憶領域を一つのメモリに収容しているが, 並列に動作するプロセス数が増えると, メモリアクセスが処理速度のボトルネックになると考えられる. 各プロセスが独立したメモリを持つ構成のハードウェアを実装することが次の課題として挙げられる.

6. む　す　び

本稿では, Erlang のサブセットによる組込みシステムの制御記述からの高位合成手法を提案した. 提案手法に基づく合成の処理系を Perl で実装し, 2 プロセスからなる簡単な制御記述から論理合成可能な Verilog 記述を生成できることを示した. 今後の課題としては, 回路規模の削減およびメモリを分散化したハードウェア構成の検討が挙げられる.

謝　　辞

本研究に関して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に感謝いたします. また, 本研究に関してご協力, 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. 本研究の一部は, NEDO エネルギー・環境新技術先導プログラム「制御高度化により自動車等を省エネルギー化する低レイテンシコンピューティングの研究」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] 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).

[8] 高島史明, 石浦菜岐佐, 織野真琴, 冨山宏之, 神原弘之: “アセンブリコードを中間表現とする高位合成における関数の併合,” 信学技報, VLD2010-106 (Jan. 2012).