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
VLD2015-106,CPSY2015-138,RECONF2015-88 (2016-01)

割込みハンドラを独立したモジュールとして実装するバイナリ合成

伊藤　直也†　石浦菜岐佐†　冨山　宏之††　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
†† 立命館大学 理工学部 〒525–8577 滋賀県草津市野路東 1 丁目 1–1
††† 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, 外部割込み処理が記述された機械語プログラムを入力として, その通常処理と割込み処理を独立したモジュールとしてハードウェアに合成する手法を提案する. 以前に著者らが提案した手法は, 外部割込み処理を含む機械語プログラムをそのままハードウェアに合成できたが, すべてを単一のモジュールとして実装していたため, 割込み処理を開始できるタイミングが限定されるという問題や, レジスタ退避/復元の処理のために回路規模が増大する等の課題があった. 本稿では, プログラム中の通常処理と割込み処理を独立して動作するモジュールに合成することにより, 割込み処理を任意のタイミングで開始できるようにするとともに, 不要となるレジスタ退避/復帰コードを削除することにより回路規模と処理時間を改善する. また, 2 つの処理を並列に実行できるようにすることにより, 全体としての実行時間を短縮する. 外部割込みサービスルーチンを含む C プログラムをハードウェアに合成した結果, CPU の約 1.4 倍の回路規模で, 遅延時間を約 20%, 実行サイクル数を約 80% 削減することができた.
キーワード　高位合成, バイナリ合成, 機械語プログラム, 割込みハンドラ

Binary Synthesis Implementing External Interrupt Handler as Independent Module

Naoya ITO†, Nagisa ISHURA†, Hiroyuki TOMIYAMA††, and Hiroyuki KANBARA†††

† Kwansei Gakuin University, 2-1 Gakuen, Sanda, Hyogo, 669–1337, Japan
†† Ritsumeikan University, 1-1-1 Noji–Higashi Kusatsu, Shiga, 525–8577, Japan
††† ASTEM RI/KYOTO, 134 Chudoji Minamimachi, Shimogyo-ku, Kyoto, 600–8813, Japan

Abstract　This article presents a method of synthesizing a given machine program with external interrupt handling into hardware where the main process and the interrupt handler are implemented as separate modules. Our previous method synthesized the whole program into a single hardware module, in which save and restore of the registers imposed limitations on the timing to start interrupt handling and also impaired efficiency of the synthesized hardware. By executing the two processes by separate modules, register save/restore can be eliminated, which allows interrupt handler to start at arbitrary timing and reduces the response time and cost of the hardware. Moreover, by allowing two processes to run in parallel, total execution time is also reduced. An experiment with a simple program has shown that the execution cycles and the delay were reduced by about 80% and 20%, respectively, as compared with MIPS CPU, at the cost of 40% increase in the hardware cost.
Key words　High-Level Synthesis, binary synthesis, machine program, interrupt handler

1. はじめに

近年, 組込みシステムの高機能化が進む一方で, その低消費電力化や高性能化の要求はますます厳しくなりつつある. 限られた設計期間内に, これらの要求を満たすシステムを効率的に設計するための一手法として, 既存のソフトウェアを高位合成 [1] によりハードウェア化する技術の研究が行われている [2,3].

プロセッサが外部機器を制御する用途では, プログラム中に割込みの処理が書かれる. 文献 [4] では, 処理とデバイス間の通信, 割込み, 割込み処理を抽象化した制御システムモデルから, 割込みによって駆動される専用ハードウェアを自動合成する手法を提案している. システムのプロセス毎にそれをソフトウェアかハードウェアのどちらにマッピングするかを選択できるが, システムの実装において常にプロセッサが必要となる.

文献 [5] では, バイナリ合成によって, 外部割込みのハンドラを含む機械語プログラムを書き換えることなくそのままハードウェアに合成する手法を提案している. しかしこの手法では, 割込みハンドラへの遷移のタイミングが基本ブロックの末尾からに限られており, 割込みへの応答時間に課題があった. また, プログラム中の汎用レジスタの退避/復元はハードウェア化しても逐次的にしか実行できないため, 割込み処理があまり効率化されず, 回路規模も増大してしまうという課題もあった.

そこで本稿では, [5] の手法において, 割込みハンドラを独立したモジュールとして実装する手法を提案する. 本手法で提案するハードウェアは, 通常処理を行うモジュールと割込み処理を行うモジュールの 2 つを備え, それぞれを独立して動作させる. これにより, 割込み処理を任意のタイミングで開始できるようにするとともに, 不要となるレジスタ退避/復帰コードを削除することにより回路規模と処理時間を改善する. また, 2 つの処理を並列に実行できるようにすることにより, 全体としての実行時間を短縮する.

提案手法をバイナリ合成システム ACAP [3] に実装し, 動作確認と性能評価を行った. 割込みサービスルーチンを含む C プログラムをハードウェアに合成した結果, CPU の約 1.4 倍の回路規模で, 約 6.2 倍の高速化を達成することができた.

2. MIPS の外部割込み処理とそのバイナリ合成

2.1 バイナリ合成

高位合成の入力記述には, C 等の高水準言語が用いられるが, アセンブリや機械語を用いるものもあり, そのような合成技術はバイナリ合成 (binary synthesis) [6] と呼ばれる. バイナリ合成は, ポインタやライブラリ呼出し等を含む広範なプログラムを合成対象とすることができる. バイナリ合成では, 関数等の単位でソフトウェアの一部をハードウェア化することもできるが, 実行可能コード全体を処理対象とすることにより, それを搭載したプロセッサと等価なハードウェアを合成することもできる.

ACAP [3] は, 著者らが開発を進めているバイナリ合成システムであり, MIPS R3000 [7] の機械語を入力としてハードウェアを合成する. ACAP には, (1) C で書かれた関数群をソフトウェアから呼び出せるハードウェアに合成する「分割コンパイルモード」, (2) リンク済み実行可能コード全体をハードウェアに合成する「全体合成モード」, (3) リンク済み実行可能コードの選択された部分を CPU 密結合型アクセラレータに合成する「アクセラレータモード」がある. このうち (2) では, MIPS 上で実行されるリンク済み実行可能コードを機能等価なハードウェアに合成できる [8]. ソースプログラムは, C で書かれていてもアセンブリで書かれていてもよく, スタートアップルーチンや浮動小数点エミュレーション用の実行時ライブラリ等がリンクされていてもよい. ACAP では, 命令レベル並列性やチェイニングの利用により, MIPS よりも実行サイクル数の削減を図ることができる. また, 命令メモリが不要となるため, プログラムが小規模であれば, 回路規模を削減することができる.

2.2 MIPS R3000 での外部割込み処理

MIPS R3000 では, 図 1 に示すように, CPU の中のシステム制御コプロセッサ CP0 を用いて外部割込みの処理を行う. CP0 は外部割込みの処理に, 以下の 3 つのレジスタを使用する.

- EPC レジスタ: 割込みが発生したときの PC (Program Counter) の値を保存する.
- Cause レジスタ: 割込みが発生したとき, その割込みが, 外部割込みか, 内部割込みか, システムコールか等の判別や, 外部割込みの場合は, どの外部割込みが発生したか等の情報を保存する.
- Status レジスタ: 実行モード (ユーザモードかカーネルモード) や割込みの許可, 割込みマスク等の情報を保持する.

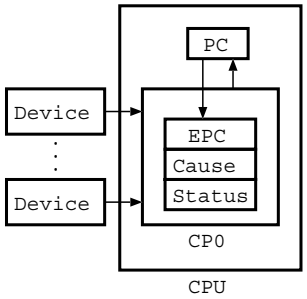

図 1 CPU とシステム制御コプロセッサ (CP0)

外部のデバイスからの割込み信号は CP0 に入力され, 以下の手順により割込み処理が行われる.

（1） CP0 は, PC の値を EPC レジスタに保存し, 割込みの原因を Cause レジスタに保存する. また, Status レジスタにカーネルモードと割込み禁止を設定する.

（2） CP0 は, CPU に割込み実行信号を送り, PC に割込みハンドラの先頭番地を設定する. これにより, CPU はハンドラに分岐する.

（3） ハンドラは, 汎用レジスタの値をメモリに退避し, Cause レジスタの割込み原因を確認して, その原因に対応するルーチンを呼び出す. ルーチンが終了したら, 汎用レジスタの値を復元する.

（4） ハンドラは, 実行モードと割込み許可設定を元に戻し, プログラムの実行を再開する場合は EPC レジスタに保存した番地に, エラー処理等を行う場合はその番地に分岐する.

割込みの処理には, 以下の命令が用いられる.

- mfc0 および mtc0 (move from/to CP0) 命令

`mfc0 rt, rd` は, CP0 のレジスタ `rd` の値を CPU の汎用レジスタ `rt` に転送し, `mtc0 rt, rd` はその逆の転送を行う. EPC レジスタからの復帰先番地の取得や, Status レジスタへの割込み禁止, 許可の設定等に用いられる.

- rfe (return from exception) 命令

割込みハンドラの終端で実行され, 実行モードと割込み許可設定を割込み前の状態に復元する.

- syscall 命令

システムコール例外を発生させる.

2.3 割込み処理を含む機械語のバイナリ合成 [5]

文献 [5] では, 外部割込みのハンドラを含む機械語プログラムをバイナリ合成によって等価なハードウェアに合成する手法を提案している. この手法では, 割込みハンドラを含むリンク済みの実行可能コードを単一のハードウェアに合成する.

この手法により合成されるハードウェアの構成を図 2 に示す. ハードウェアは, MIPS R3000 の CP0 を演算器として内部に備え, 割込みの制御用に 3 つのレジスタ HW_sig, int_sig, ESTATE を備える. ハードウェアの処理の流れを図 3 に示す. DFG1 ～ DFG4 はプログラムの基本ブロックに対応する DFG (Data Flow Graph) であり, s0 ～ s28 は状態 (制御ステップ) である. DFG1 の実行中に外部割込みが発生すると, 次のような処理が行われる.

（1） 割込み信号を CP0 に入力する.

（2） CP0 からの割込み実行信号を int_sig に保存する.

（3） 基本ブロックの最終状態 (s3) で, int_sig を確認する.

（4） int_sig が 1 であれば, s3 の次の状態 (s8) を復帰状態として ESTATE に保存し, 割込みハンドラ (Handler) の先頭 (s20) に遷移する.

（5） Handler の先頭で, int_sig をクリアする.

（6） Handler の処理が終了したら, ESTATE の状態, もしくは Handler の中で指定された番地に対応する状態に遷移する.

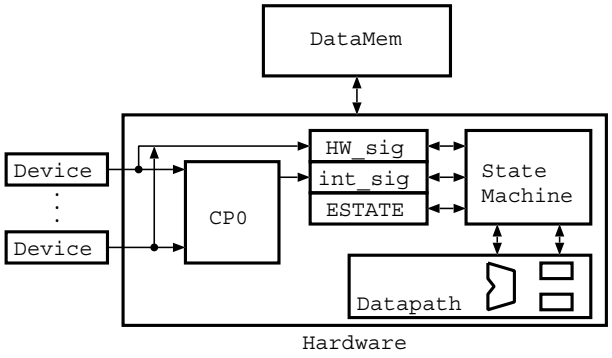

図 2 合成されるハードウェアの構成

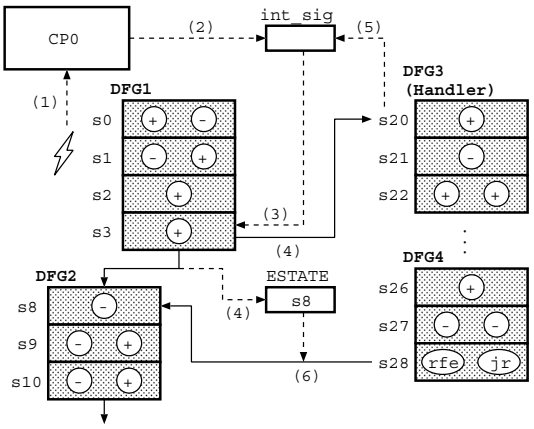

図 3 ハードウェアの外部割込み処理

この手法では, 割込みハンドラへの遷移のタイミングを基本ブロックの末尾からに限定している. これは, ハンドラに書かれているレジスタの退避/復元コードが, DFG 内で使用される中間レジスタまで対象としていないためである (DFG の終端では, 汎用レジスタに対応するレジスタのみしか使用されていないことが保証されている). 従って, サイズの大きい基本ブロックの実行中に割込みが発生した場合, それがすべて実行されるまで割込み処理を開始することができない.

また, この手法では, CPU における割込み処理と同様にレジスタの退避と復帰が必要になるが, この処理はハードウェアでも逐次的にしか行えないため, 高速化できない上, ハードウェアの状態数が増大してしまう.

3. 割込みハンドラの独立モジュールへの実装

3.1 概　　要

前述の問題を解決するため, 本稿ではバイナリ合成において, 割込みハンドラを独立したモジュールとして実装する手法を提案する. 本手法では, 図 4 に示すように, 通常処理 (main) と割込み処理 (int) から成るリンク済み実行可能コードをそれぞれ独立したモジュールとして合成することにより, 割込み処理を任意のタイミングで開始できるようにするとともに, 不要となるレジスタ退避/復帰コードを削除することにより回路規模と処理時間を改善する. また, 2 つのモジュールを並列に実行できるようにすることにより, 全体としての実行時間を短縮する.

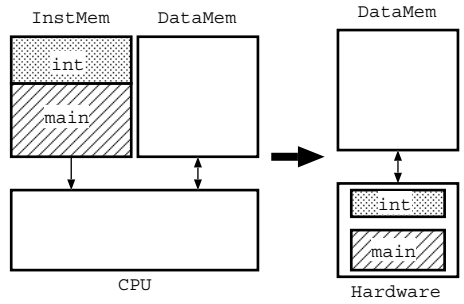

図 4 割込みハンドラを含むコードのハードウェア化

以下本稿では, 入力は MIPS R3000 の機械語プログラムを想定し, 通常処理および割込み処理の内容は一切変更せずに合成するものとする. ハードウェアは, 多重割込みの処理は行わないものとし, 割込み処理中に発生する別の割込みは無視するものとする. 割込み処理の中では, 割込みが発生した命令の番地やそれを用いた命令メモリの参照はできないものとする. 本手法は内部割込みには対応しないが, 排他制御を実現するために, システムコール命令は扱う. 通常処理と割込み処理の間のレジスタの値の共有はできないが, システムコールの処理を指定するためのレジスタの値は共有できるものとする.

ただし, ユーザは機械語プログラム中の各処理をどちらのモジュールに割り当てるかを定義する必要がある. この定義は, 機械語プログラムを逆アセンブルして得られるアセンブリプログラム中の各サブルーチンにプラグマを記述することにより行う.

3.2 ハードウェアによる外部割込みの受け付け

本手法により合成するハードウェアの構成を図 5 に示す. ハードウェアは, 通常処理 (main) と割込み処理 (int) を独立したモジュールとして持ち, それぞれが独立した状態制御機械, データパス, レジスタ群を備える. MIPS R3000 の CP0 を演算器として内部に組込み, 割込みの制御用に 2 つのレジスタ HW_sig, int_sig を備える. また, main module と int module がシステムコール時にレジスタの値を共有するために, 外部レジスタ (k0, k1) を備える.

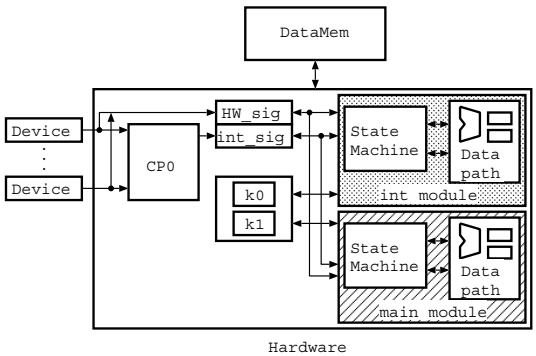

図 5 合成されるハードウェアの構成

外部割込みに対する処理の流れを図 6 に示す. 基本的に外部割込みの処理は int module が行い, main module は影響を受けない. DFG3, DFG4 は割込みハンドラの基本ブロックに対応する DFG (Data Flow Graph) であり, s0 ～ s28 は状態 (制御ステップ) である. 具体的な処理は次の通りである.

（1） int module は初期状態 s0 で待機する.

（2） 割込みが発生したら, 割込み信号を CP0 に入力する.

（3） CP0 からの割込み実行信号を int_sig に保存する.

（4） int_sig が 1 であれば, int module (DFG3, DFG4) の実行を開始する.

（5） int module の実行が終了したら (s28), int_sig をクリアし s0 に戻る.

main module と int module は独立しているため, int module が処理を行っている間も, main module は停止することなく処理を続けることができる. main module と int module の間の順序制御や共有資源へのアクセスの制御は, システムコールによる排他制御により実装されているものとする.

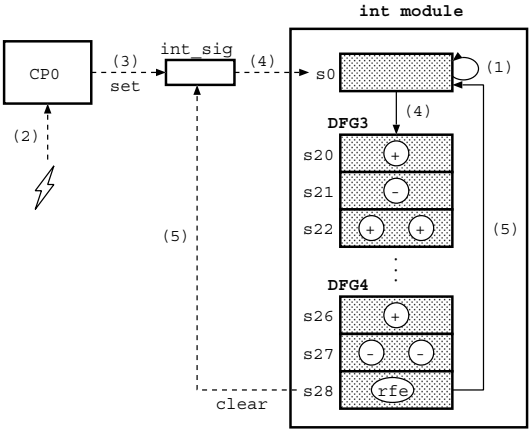

図 6 ハードウェアの外部割込み処理

3.3 排他制御とシステムコール

MIPS R3000 における排他制御は, システムコールを実行し, その処理の中で割込みを禁止することにより行える.

通常処理がクリティカルセクションを実行する際には, システムコールを行って割込みを禁止する. システムコールの処理は割込みハンドラで行われるため, その処理の間の外部割込みは無視される. MIPS で実行する場合には, 割込み処理中にシステムコールが実行されることはないが, 本手法のハードウェアではそれが起こり得る. このような場合, MIPS で実行した場合との動作の等価性を保証するため, 本手法では割込み処理が完了するまでシステムコールの実行を待たせる.

これは, syscall 命令を次の 3 つの状態に変換することにより実現する.

- syscall 状態

int module が外部割込みを処理していなければ, syscall 演算を実行して (int module で syscall の処理を実行し) syscall_wait 状態へ, 外部割込みを処理していれば int_wait 状態へ遷移する.

- int_wait 状態

int module の実行が終了するまでこの状態で待機し, 実行が終了したら syscall 状態へ戻る.

- syscall_wait 状態

syscall の処理 (int module の実行) が終了するのを待ち, 終了すれば次の状態へ遷移する. この間に発生する外部割込みはすべて無視される.

3.4 退避/復元コードの削除

提案手法では, main module と int module はそれぞれ個別に汎用レジスタを持つ. 従って, 割込みを実行する際に行う汎用レジスタの退避/復元処理が不要となる. 提案手法では, ハードウェアを合成する際, 機械語プログラム中の汎用レジスタの退避/復元コードをすべて削除する. それぞれの削除手順は次の通りである.

- レジスタ退避演算

（1） 割込みハンドラの先頭 DFG 内で, 値が定義されていないレジスタのストアを行っている演算があれば, これをレジスタ退避演算と見なして削除する.

（2） 同じ DFG 内で, 書き込み先が一時レジスタであり, かつその値が以降で参照されない演算があれば削除する.

- レジスタ復元演算

（1） 割込みハンドラの終端 DFG 内で値が参照されていないレジスタへのロード演算があれば, レジスタ復元演算と見なして削除する.

（2） 同じ DFG 内で, 書き込み先が一時レジスタであり, かつその値が以降で参照されない演算があれば削除する.

3.5 システムコールの処理を指定するレジスタの共有

MIPS R3000 におけるシステムコールは, 実行時に特定のレジスタ (k0, k1) に処理を特定する情報を格納し, 割込み処理内でそれを参照して処理を行う.

本手法では, main module と int module は独立しているため, 2 つのモジュール間でレジスタの値を渡す必要がある. そこで, 外部レジスタを介して, そのレジスタの値を main module から int module へ受け渡す. 各モジュールからの外部レジスタへのアクセスは, レジスタ書き込み, 読み込み演算を用いて行う. main module 内の syscall 演算の直前に, 対象のレジスタをオペランドとするレジスタ書き込み演算を挿入する. また, int module の先頭に, そのレジスタをオペランドとするレジスタ読み込み演算を挿入する.

4. 実装と実験結果

4.1 実　　験

本稿で提案するハードウェア合成手法を, バイナリ合成システム ACAP に実装した. ACAP は Perl5 で実装されており, Linux, Mac OS X 等の上で動作する. 合成されたハードウェアは, Verilog HDL として出力される.

外部割込みハンドラを含むプログラムを C とアセンブリで作成し, これをまず MIPS R3000 互換プロセッサ [9] 上で実行して動作確認を行った. 次に, そのプログラムを ACAP を用いて従来手法および本手法により合成し, 動作確認を行った. シミュレーションは, Xilinx ISim 14.3 で行った.

実験に用いたプログラムを図 7 に示す. メインルーチン (20 ～ 28 行目) では, 関数 `filtering` (30 ～ 56 行目) を呼出し, 縦 30 ピクセル, 横 40 ピクセルの画素の集合に対してラプラシアンフィルタリング処理を行っている. 関数 `int_getpixel` (58 ～ 65 行目) と `int_average` (67 ～ 83 行目) は, 外部割込みに対して呼び出される割込みサービスルーチンである. `int_getpixel` は, フィルタリング後の指定されたピクセルの値を返し, `int_average` は, フィルタリング前の指定されたピクセル区間の平均を返す. ピクセルの指定は, 変数 `in_x1`, `in_y1`, `in_x2`, `in_y2` を用いて行い, 出力は変数 `output` を用いて受け取る.

図 8 は割込みハンドラである [5]. 割込みが発生すると, CPU はハンドラの先頭にジャンプする. ハンドラは, 1) 汎用レジスタ退避, 2) 復帰先番地の保存, 3) 割込みサービスルーチン実行関数の呼出し, 4) 汎用レジスタ復元, 5) 割込みからの復帰を行う.

図 9 は割込み処理を行うためのライブラリである [5]. 関数 `init_interrupt` (7 ～ 12 行目) では, 初期設定として, 割込みハ

```
01 #include "interrupt_lib.h"
02
03 void filtering(int width, int height);
04 void int_getpixel(void);
05 void int_average(void);
06
07 #include "pixel.h"
08
09 int filter[3][3]={
10   {0, 1,0},
11   {1,-4,1},
12   {0, 1,0}
13 };
14
15 unsigned char pixel_out[MAX_Y][MAX_Y];
16
17 volatile int in_x1=-1,in_y1=-1,in_x2=-1,in_y2=-1;
18 volatile int output=-1;
19
20 int main(void) {
21   init_interrupt();
22   register_exc_handler(EXC_Int0,int_getpixel);
23   register_exc_handler(EXC_Int1,int_average);
24
25   filtering(MAX_X,MAX_Y);
26
27   return 0;
28 }
29
30 void filtering(int width, int height) {
31   int i,j,h,w,buf;
32   int filter_len=sizeof(filter[0])/sizeof(int);
33   int outer=filter_len/2;
34
35   for (i=0;i<height;i++) {
36     for (j=0;j<width;j++) {
37       if (
38         (outer<=i&&i<height-outer)&&
39         (outer<=j&&j<width-outer)
40       ) {
41         buf=0;
42         for (h=0;h<filter_len;h++) {
43           for (w=0;w<filter_len;w++) {
44             buf+=pixel[i+h-outer][j+w-outer]*filter[h][w];
45           }
46         }
47         pixel_out[i][j]=
48           (buf <   0) ?   0 :
49           (buf > 255) ? 255 :
50           (unsigned char)buf;
51       } else {
52         pixel_out[i][j]=pixel[i][j];
53       }
54     }
55   }
56 }
57
58 void int_getpixel(void) {
59   if (
60     !((0<=in_x1&&in_x1<MAX_X)&&
61       (0<=in_y1&&in_y1<MAX_Y))
62   ) {return;}
63
64   output=(int)pixel_out[in_y1][in_x1];
65 }
66
67 void int_average(void) {
68   int i,j;
69   int sum=0;
70   int count=0;
71   if (
72     !((0<=in_x1&&in_x1<=in_x2&&in_x2<MAX_X)&&
73       (0<=in_y1&&in_y1<=in_y2&&in_y2<MAX_Y))
74   ) {return;}
75
76   for (i=in_y1;i<=in_y2;i++) {
77     for (j=in_x1;j<=in_x2;j++) {
78       sum += pixel[i][j];
79       count++;
80     }
81   }
82   output=(int)(sum/count);
83 }
```

図 7　テストプログラム

ンドラから割込みサービスルーチン実行関数 `run_exc_handler` (21 ～ 42 行目) が実行されるよう登録し, システムコール例外 (EXC_Sys) に対してルーチン `run_syscall_handler` (44 ～ 54 行目) を登録する. 関数 `register_exc_handler` (14 ～ 19 行目) では, 割込み原因 exc に対してルーチン f を登録する. 関数 `int_prohibiton` (56 ～ 60 行目) は, 割込み禁止を設定するためのインタフェース関数であり, システムコールによりルーチン `run_syscall_handler` を呼出し, そこから `int_prohibition_func` (62 ～ 69 行目) を呼出すことにより, 割込み禁止を設定する. 関数 `int_permission` (71 ～ 76 行目)

```
; 1) Storing values of general registers
80000080:  lui k0,0xc000
80000084:  ori k0,k0,0x90
80000088:  sw at,4(k0)
8000008c:  sw v0,8(k0)
80000090:  sw v0,12(k0)
...
800000f8:  sw sp,116(k0)
800000fc:  sw s8,120(k0)
80000100:  sw ra,124(k0)

; 2) Storing the return address
80000104:  mfc0 ra,c0_epc
80000108:  sll zero,zero,0x0
8000010c:  sw ra,0(k0)

; 3) Calling the interrupt service routine execution function
80000110:  lui ra,0xc000
80000114:  ori ra,ra,0x2c
80000118:  lw t0,0(ra)
8000011c:  sll zero,zero,0x0
80000120:  beqz t0,80000134
80000124:  sll zero,zero,0x0
80000128:  jalr t0
8000012c:  sll zero,zero,0x0
80000130:  sll zero,zero,0x0

; 4) Recovering values of general registers
80000134:  lui k0,0xc000
80000138:  ori k0,k0,0x90
8000013c:  lw at,4(k0)
80000140:  lw v0,8(k0)
80000144:  lw v1,12(k0)
...
800001b0:  lw sp,116(k0)
800001b4:  lw s8,120(k0)
800001b8:  lw ra,124(k0)

; 5) Returning from the interrupt
800001bc:  lw k0,0(k0)
800001c0:  sll zero,zero,0x0
800001c4:  jr k0
800001c8:  rfe
800001cc:  sll zero,zero,0x0
```

図 8　割込みハンドラ [5]

では, 割込み許可を設定する.

実験の結果, 外部割込み信号を受け取ったその直後に割込み処理モジュールが実行を開始し, 処理が終了すると停止し, 割込み処理モジュールが実行している間も通常処理モジュールは実行を続けることを確認した. また, CPU および従来手法でのハードウェアで実行した場合と同じタイミングで, 同じ回数だけ割込みを発生させた結果, `int_getpixel` と `int_average` どちらを実行した場合においても, 変数 `output` のメモリダンプが一致することを確認した.

4.2 性能評価

前節のプログラムを, Xilinx ISE 14.3 を用い, FPGA Xilinx Spartan-3E をターゲットとして論理合成した. プログラムのコンパイルは GCC 4.8.2 で, 最適化オプション -O2 を指定して行った. ハードウェアの合成における ALU, 乗除算器の資源制約は 3 個に設定し, チェイニングは行っていない.

結果を表 1 に示す. 「MIPS」は, 機械語プログラムを MIPS プロセッサ上で実行した場合の結果, 「従来手法」は [5] の手法による結果, 「本手法」は本稿の提案手法による結果である. 「Slices」は回路規模を表し, 「Delay」は遅延時間を表す. ただし, 「MIPS」の回路規模は, 命令メモリの回路規模を 1 命令 (32 ビット) あたり 1 Slice として加えた結果である. 「Cycles」の「int (1)」, 「int (2)」は, それぞれ図 7 の `int_getpixel`, `int_average` を実行するための割込みを発生させてから, その処理が終了するまでの実行サイクル数を表す. 「total (0)」は, 図 7 を割込みを発生させずに実行した場合の, 「total (1)」は, 図 7 の `int_getpixel` を 1,000 回実行した場合の, 「total (2)」は, 図 7 の `int_average` を, 40 × 30 ピクセルすべてを指定し 10 回実行した場合のプログラム全体の実行サイクル数を表す.

「int (1)」において, MIPS の結果と比較すると, 従来手法では実行サイクル数を約 12% 削減したのに対し, 本手法では約 54% 削減することができた. これは本手法において, 汎用レジ

表 1 性 能 評 価

	Slices	Delay [ns]	Cycles				
			total (0)	int (1)	total (1)	int (2)	total (2)
MIPS	3,559 (1.00)	25.24 (1.00)	458,326 (1.00)	155 (1.00)	604,189 (1.00)	11,494 (1.00)	573,116 (1.00)
従来手法	5,937 (1.67)	18.86 (0.75)	110,467 (0.22)	135.7 (0.88)	242,467 (0.40)	7,812 (0.68)	188,537 (0.33)
本手法	5,014 (1.41)	20.16 (0.80)	110,467 (0.22)	72 (0.46)	100,467 (0.17)	7,747 (0.67)	100,467 (0.18)

```
01: #include "interrupt_lib.h"
02:
03: extern void *interrupt_call;
04: extern void (*exc_handler[24])();
05: extern void *_reg_store;
06:
07: void init_interrupt(void)
08: {
09:   (*((unsigned int*)&interrupt_call))
10:           = (unsigned int)run_exc_handler;
11:   register_exc_handler(EXC_Sys, run_syscall_handler);
12: }
13:
14: void register_exc_handler(unsigned int exc, void (*f)(void))
15: {
16:   if (EXC_MOD <= exc && exc <= EXC_Int5) {
17:     exc_handler[exc] = f;
18:   }
19: }
20:
21: void run_exc_handler(void)
22: {
23:   int i;
24:   unsigned int cause_reg, exc_code, int_field;
25:   void (*handler)() = 0x00000000;
26:
27:   asm("mfc0 %0, $13" :  "=r" (cause_reg));
28:   exc_code = (cause_reg >> 2) & 0xf;
29:   int_field = (cause_reg >> 8) & 0xff;
30:
31:   if (!exc_code) {
32:     for (i = EXC_Sw0; i <= EXC_Int5; i++) {
33:       if (int_field & 0x1) {
34:         handler = exc_handler[i]; break;
35:       }
36:       int_field = int_field >> 1;
37:     }
38:   } else {
39:     handler = exc_handler[exc_code];
40:   }
41:   if (*handler) {(*handler)();}
42: }
43:
44: void run_syscall_handler(void)
45: {
46:   unsigned int reg_k1;
47:
48:   asm("add %0, $0, $27" :  "=r" (reg_k1));
49:
50:   if (reg_k1 == 1) {
51:     int_prohibition_func();
52:     ((unsigned int *)&_reg_store)[0] += 4;
53:   }
54: }
55:
56: void int_prohibition(void)
57: {
58:   asm("addiu $27,$0,1");
59:   asm("syscall");
60: }
61:
62: void int_prohibition_func(void)
63: {
64:   asm("lui $8, 0xffff");
65:   asm("ori $8, 0xfffb");
66:   asm("mfc0 $9, $12");
67:   asm("and $9, $9, $8");
68:   asm("mtc0 $9, $12");
69: }
70:
71: void int_permission(void)
72: {
73:   asm("mfc0 $8, $12");
74:   asm("ori $8, $8, 0x1");
75:   asm("mtc0 $8, $12");
76: }
```

図 9 interrupt_lib.c [5]

スタの退避/復元コードを削除し, DFG の末尾以外からも割込みの受信が可能となったためであり, 「int (1)」のように比較的軽い処理の割込みを多数回発生させる場合, そのレイテンシを大幅に削減できることがわかる.

「total (1)」,「total(2)」において, MIPS の結果と比較すると, 従来手法では回路規模約 1.7 倍で, 遅延時間を約 25%, 実行サイクル数を約 60% 削減し, 約 3.3 倍高速化したのに対し, 本手法では, 回路規模約 1.4 倍で, 遅延時間を約 20%, 実行サイクル数を約 80% 削減し, 約 6.2 倍高速化することができた. また, 「本手法」における「total (1)」, 「total (2)」は, 割込み処理を行っているにも関わらず「total (0)」と同じ実行サイクルで実行することができた. これは本手法において, 通常処理と割込み処理を並列に実行できるためであり, 特に図 7 のように, 2 つの処理間で排他制御を行っていない場合, 完全に並列で実行することができるためである.

5. む す び

本稿では, バイナリ合成において, 割込みハンドラを独立したモジュールとして合成する手法を提案した. 今後の課題としては, ハードウェアの回路規模の削減や, 複数の割込みが発生した場合, それを並列で実行する処理の実装等が挙げられる.

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」により実施された. また, 本研究の一部は科研費 (15H02680) の支援による.

文 献

[1] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[2] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: "Advanced System-Builder: A Tool Set for Multiprocessor Design Space Exploration," in *Proc. ISOCC 2010*, pp. 79–82 (Nov. 2010).

[3] 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).

[4] Y. Ando, S. Honda, H. Takada, M. Edahiro: "System-level Design Method for Control Systems with Hardware-implemented Interrupt Handler," *IPSJ*, vol. 23, no. 5, pp. 532–541 (Sept. 2015)

[5] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: "High-Level Synthesis from Programs with External Interrupt Handling," in *Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015), R1-3*, pp. 10–15 (Mar. 2015).

[6] G. Stitt and F. Vahid: "Binary synthesis," *ACM TODAES*, vol. 12, no. 3, article 34, pp. 1–30 (Aug. 2007).

[7] G. Kane 著, 前川 訳: mips RISC アーキテクチャ—R2000/R3000, 共立出版 (1992).

[8] 伊藤, 竹林, 神原, 石浦: "多倍長整数演算ライブラリをリンクしたバイナリコードからの RSA 暗号回路の高位合成," 情処関西支部大会, A-04 (Sept. 2015).

[9] 神原, 金城, 矢野, 戸田, 小柳: "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).