外部割込みのハンドラを含むプログラムからの高位合成

伊藤直也 [1]　石浦菜岐佐 [1]　冨山宏之 [2]　神原弘之 [3]

[1] 関西学院大学 理工学部 〒 669–1337 兵庫県三田市学園 2-1
[2] 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
[3] 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

概要

本稿では, 外部割り込み処理が記述された機械語プログラムを入力として, これを実行する CPU と等価なハードウェアを合成する手法を提案する. 本手法では, CPU が割り込み処理に用いるシステム制御コプロセッサを演算器の一つとしてハードウェア内で利用する. 機械語プログラム中のコプロセッサレジスタへのアクセス命令, 割り込みからの復帰命令, およびシステムコール命令を合成の対象とし, コプロセッサにバインディングする. 割り込みサービスルーチンや割り込みからの復帰を行うためのレジスタジャンプ命令は, 命令番地をハードウェアの状態に変換する演算を用いて合成する. CPU として MIPS R3000 を想定して, 提案手法をバイナリ合成システム ACAP に実装し, 動作確認と性能評価を行った. 外部割り込みサービスルーチンを含む約 40 行の C プログラムをハードウェアに合成し, CPU 上で実行した場合と同じタイミングで外部割り込みを発生させた結果, 外部割り込みハンドラが実行され, CPU と同じ計算結果が得られることを確認した. また, CPU の約 1.1 倍の回路規模の増大で, 遅延を約 14%, 実行サイクル数を約 26% 削減し, 結果として約 1.6 倍の高速化を達成することができた.

High-Level Synthesis from Programs with External Interrupt Handling

Naoya ITO [1], Nagisa ISHIURA [1], Hiroyuki TOMIYAMA [2], and Hiroyuki KANBARA [3]

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

Abstract

This article presents a method of synthesizing a given binary program, which contain external interrupt handling, into hardware whose behaiver is equivalent to the CPU running the program. In our method, the system control coprocessor which CPU uses for interrupt handling is incorporated into the hardware as a functional unit. Instractions for accessing coprocessor registers, returning from interrupt handling, and making system calls are scheduled as operations, and bound to the coprocessor. Jump register instraction for calling and returning from interrupt service routines are synthesized using operations that convert instraction addresses into the corresponding states of the hardware. Assuming MIPS R3000 as a CPU, the proposed method has been implemented on top of binary synthesizer ACAP binary, synthesizer. A C program of about 40 lines with an external interrupt servece routine was synthesized into hardware, and it was confirmed that interrupt handling was correctly implemented. The execution cycles and the delay were reduced by 14% and 26% respectively, at the cost of 1.1 times increase in the hardware size.

1　はじめに

近年, 組み込みシステムの高機能化が進む一方で, システム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].

高位合成が, C 等の高水準言語からハードウェアを合成するのに対し, バイナリ合成 [4] は, アセンブリや機械語のプログラムからハードウェアを合成する. バイナリ合成では, 高位合成よりも広範なプログラムを扱うことが可能であり, ソフトウェアとして開発されたプログラムをハードウェア化するという目的に適している. また, この技術によって機械語プログラム全体を合成することにより, プロセッサを高速で小規模なハードウェアに置き換えることも可能となる.

しかし, プロセッサが外部機器を制御する用途では, プログラム中に割り込みの処理が書かれる. 従来の高位合成/バイナリ合成では割り込みの処理は想定していないため, このようなプログラムをそのままハードウェアに合成することはできなかった.

そこで本稿では, 外部割り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する. 本手法では, 割り込みを

処理するシステム制御コプロセッサをハードウェア内の演算器の一つとして用いる. コプロセッサ内のレジスタへのアクセス命令, 割り込みからの復帰命令, およびシステムコール命令を合成の対象とし, 割り込みサービスルーチンへの分岐や割り込みからの復帰のためのレジスタジャンプ命令は, 命令番地をハードウェアの状態に変換する演算を用いて合成する. 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. 割り込みサービスルーチンを含む約 40 行の C プログラムをハードウェアに合成した結果, CPU の約 1.1 倍の回路規模の増大で, 約 1.6 倍の高速化を達成できた.

2 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) [4] と呼ばれる. バイナリ合成により, C 以外の言語からでもハードウェアを合成することが可能となる. また, 合成されるハードウェアの性能に課題は残るが, 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も可能となる. 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, そのコード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 [6] の機械語を入力としてハードウェアを合成する. ACAP には, 次の 3 つの合成モードがある.

1. 分割コンパイルモード: 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.
2. 全体合成モード: 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.
3. アクセラレータ合成モード: 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.

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等価な処理を行うハードウェアに合成する. 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, 文字列処理等のライブラリがリンクされていてもよい.

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, 回路規模を大幅に削減することができる.

3 MIPS R3000 の外部割り込み処理

MIPS R3000 では, CPU に組み込まれたシステム制御コプロセッサ CP0 により外部割り込みの処理を行う. CP0 は外部割り込みの処理に, 以下の 3 つのレジスタを使用する.

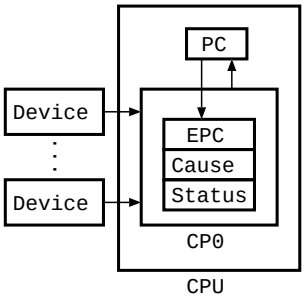

図 1 CPU とシステム制御コプロセッサ (CP0)

- EPC レジスタ
  割り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
  割り込みが発生したとき, その割り込みが, 外部割り込みか, 内部割り込みか, システムコールか等の判別や, 外部割り込みの場合は, どの外部割り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) や割り込みの許可, 割り込みマスク等の情報を保持する.

外部のデバイスからの割り込み信号は CP0 に入力され, 以下の手順により割り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, 割り込みの原因を 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り込み禁止を設定する.
2. CP0 は, CPU に割り込み実行信号を送り, PC に割り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 レジスタの割り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, 実行モードと割り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.

割り込みの処理には, 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 レジスタへの割り込み禁止, 許可の設定等に用いられる.
- rfe (return from exception) 命令
  割り込みハンドラの終端で実行され, 実行モードと割り込み許可設定を割り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.

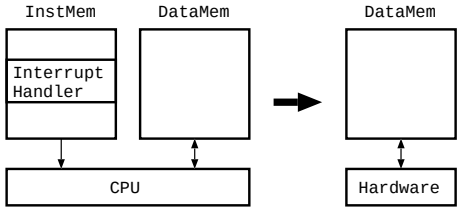

図 2　割り込みハンドラを含むコードのハードウェア化

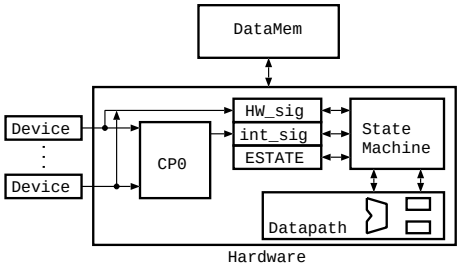

図 3　合成されるハードウェアの構成

なお, MIPS R3000 では多重割り込みは想定していない. 割り込み処理を実行している間, 他の一切の割り込みを禁止するため, この間に発生する割り込みはすべて無視される.

4　外部割り込みハンドラの高位合成

4.1　概要

本稿では, 外部割り込みハンドラを含むプログラムと等価な処理を行うハードウェアの高位合成法を提案する. 本手法では, 図 2 に示すように, 割り込みハンドラを含むリンク済み実行可能コードをハードウェアに変換し, データメモリと接続する.

以下本稿では, CPU として MIPS R3000 を想定する.

ハードウェアは, 外部割り込みを受け付けると, その基本ブロックの終端で割り込みハンドラに遷移するものとする. また, 多重割り込みの処理は行わないものとする. 割り込みハンドラの中では, 割り込みが発生した命令の番地やその前後の命令が何であるかを参照することはできないものとする. 本手法は内部割り込みには対応しないが, 共有資源に対するアクセスの排他制御を実現するために, システムコール命令を扱うことができる.

本稿で提案するハードウェアは, MIPS R3000 の CP0 を 1 つの演算器として組み込むことにより外部割り込みを処理する. mfc0, mtc0, rfe 命令を合成対象とし, CP0 にバインディングする. 割り込みサービスルーチンへの分岐や, 割り込みからの復帰のためのレジスタジャンプ命令は, 命令番地をハードウェアの状態に変換する演算を用いて合成する.

4.2　ハードウェアにおける外部割り込み受け付け

本手法により生成されるハードウェアの構成を図 3 に示す. ハードウェアは, MIPS R3000 の CP0 を演算器として内部に組み込み, 割り込みの制御用に 3 つのレジスタ HW_sig, int_sig, ESTATE を備える.

機械語から生成されるハードウェアの処理の流れを図 4 に示す. DFG1 〜 DFG4 はプログラムの基本ブロック

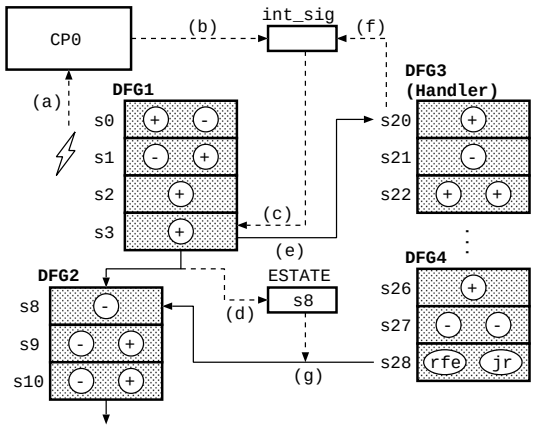

図 4　ハードウェアの外部割り込み処理

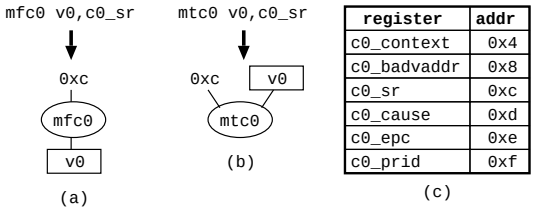

図 5　mfc0, mtc0 命令に対応する演算

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 の実行中に外部割り込みが発生した場合, 以下のように処理を行う.

1. 割り込み信号を CP0 に入力する (a).
2. CP0 からの割り込み実行信号を int_sig に保存する (b).
3. 基本ブロックの最終状態 (s3) で, int_sig を確認する (c).
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 (d), 割り込みハンドラ (Handler) の先頭 (s20) に遷移する (e).
5. Handler の先頭で, int_sig をクリアする (f).
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 (g).

4.3　割り込み関連命令の合成

提案手法では, mfc0, mtc0, rfe, syscall 命令をそれぞれ 1 演算としてスケジューリングし, CP0 にバインディングする. それぞれの変換手法を以下に示す.

- mfc0, mtc0 命令
  mfc0 命令および mtc0 命令は, それぞれ図 5 (a) (b) のように, CP0 のレジスタを図 5 (c) の番号で指定する演算に変換する.
- rfe 命令
  rfe 命令は, 入出力なしの演算として, 基本ブロックの最終状態にスケジューリングする. この演算は, CP0 に rfe 命令の実行信号を入力することにより, CP0 にこの命令の処理を行わせる.

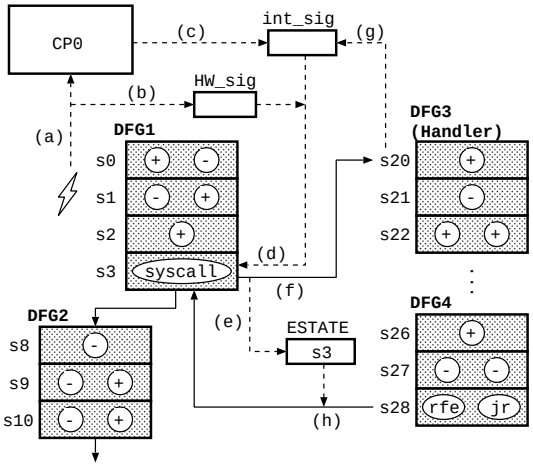

図 6　外部割り込みと重複するシステムコールの処理

- syscall 命令
  まず, syscall 命令の直後で基本ブロックを分割する. syscall 命令は, 入出力なしの演算として, 基本ブロックの最終状態に他の演算とは独立に実行されるようにスケジューリングする. この演算は, CP0 に syscall 命令の実行信号を入力することにより, 外部割り込みと同様に割り込みハンドラへの遷移を引き起こす. 基本ブロックの終端で割り込みハンドラに遷移するという仕様上, 外部割り込みとシステムコールが同じ基本ブロック中で発生する可能性がある. この場合, CP0 は外部割り込みの情報を Cause レジスタに保存するため, システムコールは無視されてしまう. そこで提案手法では, 先に外部割り込み処理を行い, それが終了した後に再度システムコールを実行するようにする. その処理の手順を図 6 に示す.
  1. システムコール命令のある DFG で外部割り込みが発生したとき, それを CP0 に入力するとともに (a), HW_sig レジスタに保存する (b).
  2. CP0 からの割り込み実行信号を, int_sig に保存する (c).
  3. システムコールが実行される基本ブロックの最終状態 (s3) で, int_sig と HW_sig を確認する (d).
  4. どちらか一方だけが 1 なら, これまでに示した方法により処理し, 両方が 1 なら, 現在の状態 (s3) を ESTATE に保存し (e), 割り込みハンドラ (Handler) に遷移する (f).
  5. Handler の先頭状態で, int_sig をクリアする (g).
  6. 外部割り込み処理が終了した後, ESTATE に保存した状態 (s3) に復帰する (h). 外部割り込みとシステムコールが両方起こっていた場合には, システムコールを実行することになり, 次の状態 (s8) を ESTATE に保存して Handler に遷移する.

4.4　レジスタジャンプ命令の合成

割り込みサービスルーチンの選択や割り込みからの復帰に用いられる jalr 命令や jr 命令では, 分岐先が実行時のオペランドの値により変わるため, 分岐先を静的に決定できない. 提案手法では, 命令番地をハードウェアの状態に変換する演算を用いてこれらの演算を合成する.

例えば, jr 命令は, 図 7 (a) に示すように, 遷移先番地を入力とする演算として, 基本ブロックの最終状態にスケジューリングする. その後, jr 命令は, 図 7 (b) に示すように, “pc2state” 演算とその結果の状態レジスタへの書き込みに変換する. この演算は, 入力された番地を対応する状態に変換するものであり, 組み合わせ回路に合成する.

pc2state 演算が変換の対象にする番地は, 基本的にはサブルーチンの先頭番地およびサブルーチンの復帰番地のみでよい. しかし, jr 命令は switch-case 文をコンパイルした場合にも出現するため, その命令番地も対象に加える必要がある. switch-case 文の遷移先番地は, ジャンプテーブルの中に列挙されるので, ジャンプテーブルに含まれる番地を抽出し, その番地で基本ブロックを分割する (図 8). 番地の抽出は, データメモリ中で命令メモリの領域に入る 4 の倍数の値を列挙することにより行う.

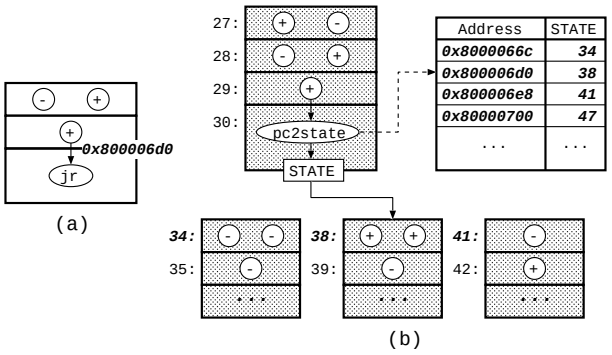

図 7　番地から状態への変換

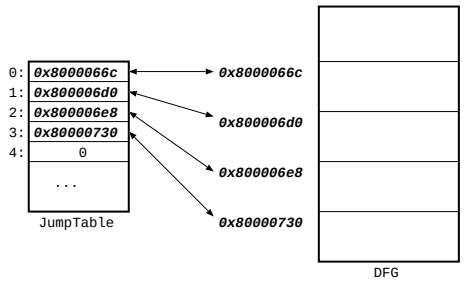

図 8　ジャンプテーブルによる基本ブロックの分割

5　実装と実験結果

5.1　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.

外部割り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 [7] 上で実行して動作確認を行った. そのプログラムをそのまま ACAP 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.

```
01: #include "interrupt_lib.h"
02:
03: #define N 10
04: #define LOOP_NUM 30
05:
06: void int_routine(void);
07:
08: volatile int input;
09:
10: int sp = 5;
11: int buffer[N] = {10, 20, 30, 40, 50};
12: int output[LOOP_NUM];
13:
14: int main(void)
15: {
16:   int i, j;
17:
18:   init_interrupt();
19:   register_exc_handler(EXC_Int0, int_routine);
20:
21:   for (i = 0; i < LOOP_NUM; i++) {
22:     for (j = 0; j < sp; j++) {
23:       output[i] += buffer[j] * buffer[j];
24:     }
25:
26:     int_prohibition();
27:     sp = 0;
28:     int_permission();
29:   }
30:   return 0;
31: }
32:
33: void int_routine(void)
34: {
35:   if (sp < N) {
36:     buffer[sp] = input;
37:     sp++;
38:   }
39: }
```

図 9　割り込みテストプログラム

実験に用いたプログラムを図 9, 10, 11 に示す．図 9 のプログラムは, 整数データの入力を割り込みによって受け付けてバッファに保存し, 入力されたデータの 2 乗和をメインルーチンで計算するというものである．当該割り込みが起こると, 33 ～ 39 行目の関数 int_routine が呼び出され, 外部入力にマップされた変数 input から読み込んだ値を配列 buffer に保存する．メインルーチン (14 ～ 31 行目) では, 22 ～ 24 行目のループで buffer の 2 乗和を求める．26 ～ 28 行目では, 割り込み禁止関数 int_prohibition と割り込み許可関数 int_permission で挟まれた排他制御の下で, buffer の初期化を行う．int_routine の登録は, 19 行目で行っている．ここでは, 0 番目の外部割り込み (EXC_Int0) に対して int_routine が呼び出されるように登録している．また, 18 行目の関数 init_interrupt は, 割り込み処理を行うための初期設定を行うものである．

図 10 のプログラムは割り込みハンドラである．外部割り込みが発生すると, CPU はまず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) 割り込みサービスルーチン実行関数の呼出し, 4) 汎用レジスタの復元, および 5) 割り込みからの復帰処理を行う．

図 11 のプログラムは, 割り込み処理を行うためのライブラリ関数であり, C とインラインアセンブリで記述している．関数 init_interrupt (7 ～ 11 行目) では, 初期設定として, 割り込みハンドラから割り込みサービスルーチン実行関数 run_exc_handler (20 ～ 41 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 (43 ～ 53 行目) を登録する．関数 register_exc_handler (13 ～ 18 行目) は, 割り込み原因 exc に対してルーチン f を登録

```
; 1) 汎用レジスタ退避
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) 復帰先番地の保存
80000104:  mfc0 ra,c0_epc
80000108:  sll zero,zero,0x0
8000010c:  sw ra,0(k0)

; 3) 割り込みサービスルーチン実行関数の呼出し
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) 汎用レジスタ復元
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) 割り込みからの復帰
800001bc:  lw k0,0(k0)
800001c0:  sll zero,zero,0x0
800001c4:  jr k0
800001c8:  rfe
800001cc:  sll zero,zero,0x0
```

図 10　割り込みハンドラ

する．関数 int_prohibition (55 ～ 59 行目) は, 割り込み禁止を設定するためのインターフェイス関数であり, 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_prohibiton_func (61 ～ 68 行目) を呼び出すことにより, 割り込み禁止を設定する．関数 int_permission (70 ～ 75 行目) では, 割り込み許可を設定する．

実験の結果, 外部割り込み信号を受け取った基本ブロックの終端から割り込みハンドラが実行され, その処理が終了した後, 通常の処理に復帰することを確認した．また, CPU で実行した場合と同じタイミングで, 同じ回数だけ割り込みハンドラを実行させた結果, 変数 output のメモリダンプが CPU と一致することを確認した．

5.2　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．ハードウェアの合成における ALU 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．Target の SW は, 機械語プログラムを MIPS により実行した結果であり, HW は 提案手法による合成結果である．Slices, Delay はそれぞれのスライス数, 遅延時間である．Cycles は, プログラム全体を実行している間に外部割り込み信号を 1 度だけ与え, 外部割り込みハンドラを実行させた際のサイクル数である．MIPS で実行した場合と比較すると, 提案手法では回路規模約 1.1 倍の増大で, 遅延を約 14%, 実行サイクル数を約 26% 削減することができ, 結果として約 1.6 倍の高速化を達成することができた．回路規模が増大したのは, 通常のプログラム (94 命令) と割り込みハ

```
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 11　interrupt_lib.c

表 1　割り込みテストプログラムの合成結果

Target	Slices	Delay[ns]	Cycles
SW	3144 (100.0%)	25.243 (100.0%)	5192 (100.0%)
HW	3563 (113.3%)	21.567 (85.4%)	3816 (73.5%)

ンドラ (84 命令) に加え, interrupt_lib.c (85 命令) の割り込み処理に必要な関数も合成に加えていることが原因であると考えられる.

6　考察

提案手法ではハンドラへの遷移タイミングを DFG の末尾からに限定しているが, これは, 割り込み処理に伴うレジスタの退避と復元をハンドラに書かれたコードに従って行うためである. 一般に, 基本ブロックから生成される DFG 中では, MIPS のレジスタに対応したレジスタ以外にも, 演算の中間結果を保存するためのレジスタが生成される. しかし, ACAP では, 基本ブロックの終端ではそれらのレジスタが出現しないことを保証しているため, ハンドラへの分岐を DFG の末尾からに限定すれば, ハンドラを修正なしに合成して用いることができる.

この限定は, 通常処理とハンドラがルーチンを共有している可能性を考慮して行っている. もし, この 2 つを完全に分離し, 独立したハードウェアモジュールとして合成することができれば, 分岐タイミングの制約をなくせるだけでなく, レジスタの退避/復元処理も削除できるため, 実行速度の高速化や回路規模の削減が可能となる.

本稿で内部割り込みを扱っていないのは, 上記のハンドラへの遷移タイミングの問題に加え, 内部割り込みのハンドラではアーキテクチャ依存のコードが書かれることが多く, コードを修正することなくハードウェアで等価な処理を実現することが困難なためである. 遷移タイミングの問題を解決し, ハンドラの処理を限定するか, 高位合成を想定したハンドラを書くことにすれば, 内部割り込みにも対応することができると考えられる.

7　むすび

本稿では, 外部割込みハンドラを含むプログラムの高位合成を提案した. 本手法をバイナリ合成システム ACAP に実装し, 生成したハードウェアに外部から割り込み信号を与えた結果, 割り込みハンドラの処理が実行されることを確認した.

今後の課題としては, 生成されたハードウェアの回路規模の削減や, 実行速度の改善などが挙げられる.

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.

参考文献

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

[5] J. L. Hennessy and D. A. Patterson: *Computer Architecture, Third Edition*, Morgan Kaufmann (Aug. 2004).

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

[7] 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).