Binary Synthesis Implementing External Interrupt Handler as Independent Module

Naoya Ito*
Kwansei Gakuin University
School of Science and Technology
Sanda, Hyogo, Japan

Yuuki Oosako
Kwansei Gakuin University
School of Science and Technology
Sanda, Hyogo, Japan

Nagisa Ishiura
Kwansei Gakuin University
School of Science and Technology
Sanda, Hyogo, Japan

Hiroyuki Kanbara
ASTEM RI Kyoto
Kyoto, Japan

Hiroyuki Tomiyama
Ritsumeikan University
College of Science and Engineering
Kusatsu, Shiga, Japan

ABSTRACT

This article presents a method of synthesizing hardware from a given executable binary code with an external interrupt handler, where the normal flow and the interrupt handling are executed by separate hardware modules. Our previous method synthesized the whole program into a single hardware module, in which register save/restore imposed limitations on the timing to start interrupt handling and also impaired efficiency of the synthesized hardware. By executing the two tasks on 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. 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. A motor controller driven by periodical interrupts from a timer has been successfully synthesized from C and assembly programs, which runs more than 20 times faster than the MIPS CPU.

CCS CONCEPTS

• **Hardware → Hardware-software codesign**; • **Computer systems organization → Embedded hardware**;

KEYWORDS

High-level synthesis, binary synthesis, interrupt handling

ACM Reference Format:
Naoya Ito, Yuuki Oosako, Nagisa Ishiura, Hiroyuki Kanbara, and Hiroyuki Tomiyama. 2017. Binary Synthesis Implementing External Interrupt Handler as Independent Module. In *Proceedings of RSP'17, Seoul, Republic of Korea, October 15–20, 2017,* 7 pages.
https://doi.org/10.1145/3130265.3130317

*Currently with Murata Manufacturing Co., Ltd., Nagaokakyo, Kyoto, Japan

RSP'17, October 15–20, 2017, Seoul, Republic of Korea

ACM ISBN 978-1-4503-5418-9/17/10…$15.00
https://doi.org/10.1145/3130265.3130317

1 INTRODUCTION

While embedded systems are getting increasingly rich in functionality, higher and higher performance is required within limited power consumption. To implement such systems efficiently within limited design periods, there have been a lot of attempts to utilize high-level synthesis technology [1] to automatically design hardware from existing software [3, 4].

Binary synthesis [2] is a kind of high-level synthesis which takes binary codes, instead of programs in high-level programming languages, as its inputs. Binary synthesis can handle wider range of programs than high-level synthesis and hence it is suitable for translating programs originally developed as software into hardware. However, applications where processors control external devices, the programs contain interrupt processing.

Ando et al. [5] proposed a method of synthesizing dedicated hardware driven by interrupts from an abstract system model that captured interrupts, interrupt handlers, and communication among devices. Each process may be implemented as either software or hardware according to the designer's choice. However, regardless of the choice, a CPU was always necessary in the resulting system.

On the other hand, in our team's previous work [6], a method was proposed which converted a given binary program containing an external interrupt handler into hardware whose behavior was equivalent to the CPU running the program. No rewriting was required on the binary program and the resulting hardware ran faster than the program on the CPU. However, in this method, jumps to the handler were allowed only at the end of the basic blocks, which hinders quick response to interrupts. Moreover, save/restore of registers took time even on the hardware for it resulted in sequential memory accesses.

We have found that the bottleneck was caused by register save/restore for context switching. Based on this observation, we propose in this paper a method of extending the previous method so that a main process and an interrupt handler are synthesized into separate hardware modules. Since the two modules hold their own context, operations for register save/restore will be eliminated. This allows the interrupt handler to start execution without waiting for the main process to reach at the end of the basic blocks. Furthermore, two modules may run in parallel, which drastically reduces the total execution time.

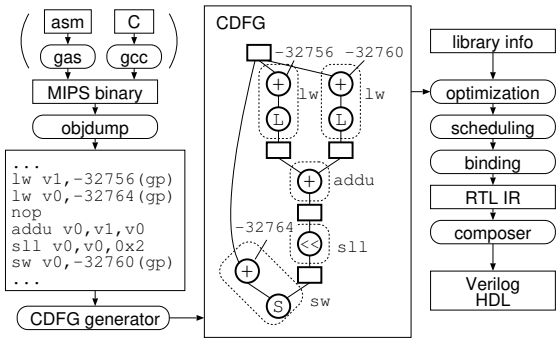


Figure 1: Flow of synthesis in ACAP.

The proposed method has been implemented on top of binary synthesizer ACAP [3]. 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. A motor controller driven by periodical interrupts from a timer has been successfully synthesized from C and assembly programs, which runs more than 20 times faster than the MIPS CPU.

2 BINARY SYNTHESIS OF MIPS INTERRUPT HANDLING

2.1 Binary Synthesis and ACAP

High-level synthesis takes behavioral specification as inputs which are usually written in high-level languages such as C. However, the behavior may be specified also in assembly or machine languages, and such synthesis techniques are called binary synthesis [2]. It can handle wider range of software programs than conventional high-level synthesis, although back-end technologies to generate hardware are common. While binary synthesis can be used to convert some computationally intensive parts of given software programs, it can also convert a whole binary code into a hardware module which is functionally equivalent to the processor running the program.

ACAP [3] is a binary synthesizer which generates hardware from MIPS R3000 binary codes. As shown in Figure 1, it takes a MIPS executable binary generated by GCC from C programs or by GAS from hand coded assembly programs. It disassembles (by objdump) a binary code to recover an assembly program, and converts it into a CDFG (control dataflow graph); an instruction sequence in each basic block is translated into a DFG (dataflow graph), and jumps and branches are compiled into transitions among the DFGs (delayed branches/jumps are properly recognized). Register jumps, used for multiway branches and return from subroutines/interrupt handlers, are handled by constructing a table that translates instruction address into the IDs of the corresponding DFGs. ACAP itself does not distinguish privilege modes; all the instructions might be executed as if it were in the kernel mode. It then performs scheduling and binding to generate a Verilog HDL code based on typical high-level synthesis algorithms.

While ACAP can transform user specified sections of linked executable code into a hardware accelerator which is tightly coupled with the CPU, it can also synthesize a whole linked executable code into a hardware module which is compatible with the CPU that runs the code. This paper utilizes the the latter mode where the synthesized hardware module can replace the CPU and the instruction memory. The source program may be written in C language or assembly/machine language, and may be linked with start-up routines and library codes for floating point emulation, string processing, etc. Synthesized hardware is expected to run faster than MIPS, due to instruction level parallelism and operation chaining. The hardware size is roughly proportional to the number of instructions in the input binary code, so trade-offs between speed-up and hardware cost need to be considered.

2.2 Interrupt Handling of MIPS R3000

MIPS R3000 handles interrupts by using CP0, the system control coprocessor, which is embedded in the CPU. CP0 has the following three registers to handle external interrupts.

- EPC register
  keeps the PC (Program Counter) value at which the interrupt is triggered.
- Cause register
  keeps the information regarding the cause of the interrupt, i.e., whether it is an external interrupt, an internal interrupt, or a system call, and also its detailed cause in the case of an external interrupt.
- Status register
  keeps the system's status information such as the execution mode (the user mode or the kernel mode), and the interrupt mask.

When an interrupt signal is sent from an external device to CP0, it is handled in the following way. The flow assumes only single-level interrupts; during interrupt handling, all the other interrupts are ignored.

(1) CP0 saves the PC value and the cause of the interrupt in the EPC register and the Cause register, respectively, and updates the Status register so that the system may run in the kernel mode and the other interrupts will be prohibited.
(2) CP0 sends an interrupt execution signal to the CPU, and writes the starting address of the interrupt handler into PC so that the CPU will jump to the handler.
(3) The handler saves the values of the general purpose registers to the main memory and calls the routine corresponding to the cause in the Cause register.
(4) After the routine finishes its task, the handler restores the general purpose registers.
(5) The handler restores the execution mode and clears interrupt prohibition, and the address in the EPC register is written back to PC so that the CPU can resume execution (or just jumps to the specified address for error handling).

To handle interrupts, the following instructions are used.

- mfc0 and mtc0 (move from/to CP0)
  Instruction mfc0 rt,rd moves the value in a CP0 register rd to a general purpose register rt, and mtc0 rt,rd does the move

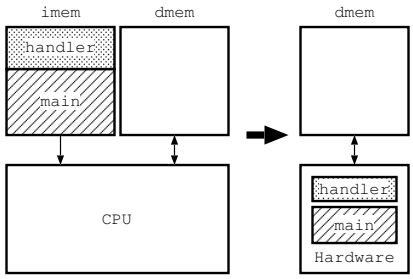


Figure 2: Synthesis from binary code with interrupt handler.

in the opposite way. Those instructions are used to access to the EPC, Cause, and Status registers.

- rfe (return from exception)
  It is executed at the end of the interrupt handler and restores the execution mode and the interrupt permission.

2.3 Synthesis from Binary Codes Containing Interrupt Handler [6]

In [6], a method of synthesizing hardware from programs containing external interrupt handlers was proposed. As illustrated in Figure 2, it translates a linked executable code including an interrupt handler to a hardware module, which replaces the CPU and the instruction memory.

The structure of the hardware generated by this method is shown in Figure 3. The hardware has CP0 of MIPS R3000 as a functional unit along with three registers HW_sig, int_sig, and ESTATE. The flow of interrupt handling is as shown in Figure 4. DFG1 through DFG4 are DFGs (dataflow graphs) corresponding to the basic blocks of a given program, and s0 through s28 are states (control steps). DFG1 and DFG2 are for the normal task and DFG3 and DFG4 are for the interrupt handler. An external interrupt triggered during the execution of DFG1 is processed in the following way.

(1) The interrupt signal is sent to CP0.
(2) The interrupt execution signal from CP0 is latched in the int_sig register.
(3) At the last state (s3) of DFG1, int_sig is tested. If it is 1, the next state of s3 (s8) is saved in the ESTATE register as the return state, and the hardware jumps to the starting state (s20) of the interrupt handler (Handler).
(4) At the initial state of the handler, int_sig is cleared.
(5) At the last state (s28) of the last DFG (DFG4) of the handler, the status is restored and the hardware returns to the state saved in the ESTATE register.

The proposed method allows jumps to the handler only at the end of the DFG. This is because register save and restore are done precisely as written in the handler code. However, high-level synthesizers generally generate extra registers to keep intermediate values, which are not saved nor restored in the interrupt handler. Therefore, if interrupts occur during execution of big basic blocks, start of interrupt handling will be delayed.

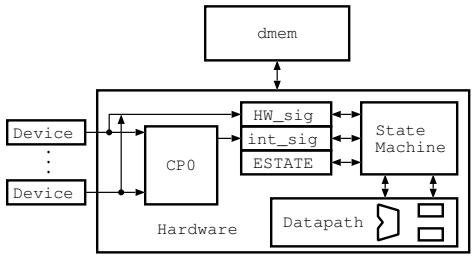


Figure 3: Configuration of the synthesized hardware in [6].

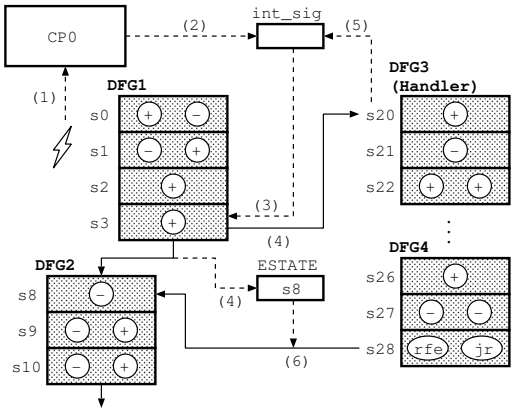


Figure 4: Interrupt handling in [6].

Moreover, in this method, synthesized hardware must save/restore registers, just as CPUs. Since this ends up with sequential memory accesses, even hardware needs almost as much cycles as the CPU.

3 SYNTHESIZING INTERRUPT HANDLER INTO SEPARATE HARDWARE

3.1 Overview

In order to solve the problem, we propose in this paper a binary synthesis method that synthesizes an interrupt handler into a separate hardware module. Henceforth, we will call a hardware module to compute normal execution flow the *main module*, and a module to handle interrupts the *handler module*. The both modules hold their contexts in their own register sets so that register save/restore operations will be eliminated and the handler module can start execution at an arbitrary timing. Moreover, the main module may continue its execution while interrupts are being handled.

In this paper, we assume binary programs for MIPS R3000, and make the following assumptions.

- Only external interrupts (hardware interrupts) are supported. Internal interrupts (software interrupts) are out of the scope of this paper. However, system calls can be synthesized to enable mutual exclusion.
- The interrupt handler cannot refer to the instruction address where the interrupt occurred nor access to the instruction memory.

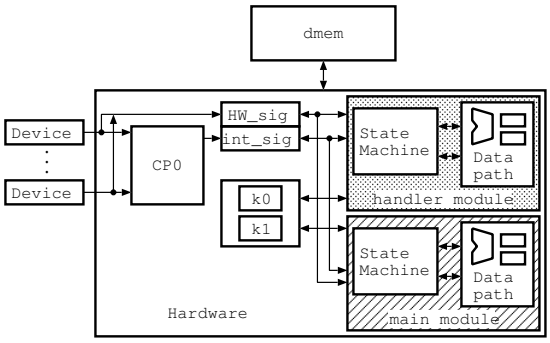


Figure 5: Configuration of the hardware synthesized in the proposed method.

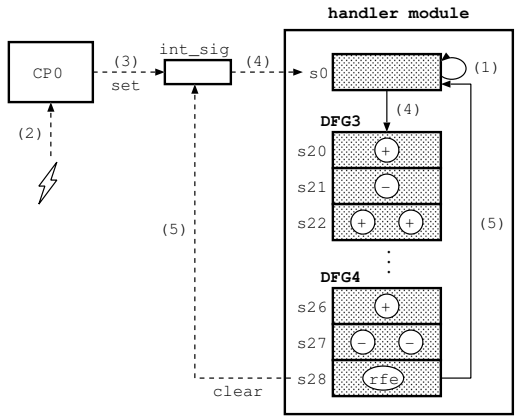


Figure 6: Interrupt handling in the proposed method.

- Multilevel interrupts are not supported. Any other interrupts during interrupt handling are ignored.

Basically, binary programs are synthesized into hardware without rewriting, but user must specify which parts of the input program are assigned to the handler module. This is specified by means of pragmas in the assembly code (obtained by disassembling the input binary).

3.2 Flow of Interrupt Processing

The configuration of the hardware synthesized by the proposed method is shown in Figure 5. Like in the previous configuration, it has CP0 along with the registers HW_sig and int_sig. Each of the main module and the handler module has an independent set of a finite state controller, a datapath, and registers. For the purpose of sharing data on system calls, two registers k0 and k1 are provided.

The flow of interrupt processing is shown in Figure 6, where DFG3 and DFG4 are of the interrupt handler and s0 ~ s28 are states (control steps). Basically, the handler module runs independently of the main module. Detailed steps for interrupt handling is as follows.

(1) The handler module waits at the initial state s0, polling the int_sig register.
(2) The interrupt signal is sent to CP0.
(3) The interrupt execution signal from CP0 is latched in the int_sig register.
(4) As soon as int_sig becomes 1, the handler module jumps to s20 and executes DFG3 and DFG4.
(5) When interrupt handling is finished (at s28), the handler module resets int_sig and returns to s0.

Unlike the method in [6], the main module may continue its execution while the handler module is running. If there is a need for serialization or mutual exclusion between the main module and the handler module, we assume that it is implemented in the machine program by using system calls, as explained in the next subsection.

3.3 Mutual Exclusion and System Call

In the MIPS R3000 convention, mutual exclusion is realized by disabling interrupts (CPU lock). This is done via the system call instruction (syscall). The syscall instruction switches the CPU state to the kernel mode and transfer control to the interrupt handler, where interrupts are disabled by updating the Status register.

In the case of single core execution, syscall will never be executed by the main routine during a certain external interrupt is handled by the interrupt handler (because the main routine is idle while the interrupt handler is running). However, in the case of our hardware configuration, this may happen, because two modules may run in parallel. In order to guarantee the equivalence of the behavior between the original machine program on the CPU and the synthesized hardware, we let the main module postpone execution of syscall until the handler module completes its task.

This is implemented by translating syscall into the three state operations.

(1) Test if the handler module is executing its task. If yes, goto (2). Otherwise goto (3).
(2) Wait until the handler module completes its task and then goto (1).
(3) Wait until the handler module to complete the task for syscall, and goto the next state. All the external interrupts occurring during this state are ignored.

We assume a calling convention that the kernel stack pointer is passed in register k0 and the syscall function is specified in k1. In our method, where the two modules have independent register sets, we somehow transfer data in the two registers. We do this via two external registers, depicted as k0 and k1 in Figure 5. After CDFG generation, operations to store k0 and k1 to the two external registers are inserted just before syscall operations in the main module, and operations to load the two registers are added in the head DFG of the handler module.

3.4 Deletion of Save/Restore Codes

In our method, the handler module does not have to save/restore registers. However, the save/restore codes are written in the given input binary codes, which must be deleted during the synthesis.

This is done just after the CDFG generation.

- Register save operations

(1) In the head DFG of the handler module, all the store operations that attempt to store undefined registers are recognized as save operations and are deleted.
(2) All the operations, whose destinations are temporary registers (created during CDFG generation) and are not referred to afterwards, are also deleted.

- Register restore operations

(1) In the DFGs that contain rfe operations, all the load operations that attempt to load to registers which will never be referred to are regarded as restore operations and are deleted.
(2) All the operations, whose destinations are temporary registers and are not referred to afterwards, are deleted.

4 EXPERIMENTAL RESULTS

4.1 Synthesis

The proposed synthesis method has been implemented on top of binary synthesizer ACAP. It is implemented by Perl5 and runs on Linux and Mac OS X, and outputs RTL description in Verilog HDL.

The test program is shown in Figure 7. Function main (lines 20–28) calls function filtering (lines 30–56) which performs Laplacian filtering (the function is called just once for the purpose of measuring execution cycles). Functions int_getpixel (lines 58–65) and int_average (lines 67–83) are service routines for external interrupts; int_getpixel returns the pixel value of a specified coordinate, and int_average the average pixel values in a specified rectangle region. The coordinates are given in variables in_x1, in_y1, in_x2, in_y2 and the result is stored in variable output.

The program shown in Figure 8 is the interrupt handler [6]. When an external interrupt occurs, CPU first jumps to the top of the handler. The handler 1) saves the general purpose registers, 2) saves the return address, 3) calls the interrupt service routine function, 4) restores the general purpose registers, and 5) returns from the interrupt.

The program shown in Figure 9 is a collection of library functions for interrupt handling which is written in C and inline assembly [6]. Function init_interrupt (lines 7–12) registers interrupt service routine execution function run_exc_handler (lines 21–42), which are called from the interrupt handler, and a routine run_syscall_handler (lines 44–54) for system call (EXC_Sys). Function int_prohibition (lines 56–60) is an interface function to set interrupt prohibition by calling run_syscall_handler through system call and then, calling int_prohibiton_func (lines 62–69). Function int_permission (lines 71–76) sets interrupt permission.

All the programs were compiled/assembled and linked into a single binary code executable on MIPS R3000 processor. After confirming the behavior of the program by running the processor in RTL simulation where external interrupts were given from the testbench, hardware is synthesized by extended ACAP. It was confirmed that the memory dumps of variable output obtained by the MIPS processor and by the hardware agreed. It was also confirmed that the main module continued its execution while the handler module was processing its tasks.

```
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 }
```

Figure 7: Test program.

4.2 Performance Evaluation

Logic synthesis was performed on the Verilog HDL description generated by ACAP. The logic synthesizer was Xilinx ISE 14.3 and the target was FPGA Xilinx Spartan-3E.

The result is shown in TABLE 1. "MIPS" indicates software execution on MIPS softcore processor, "HW [6]" and "HW (proposed)" stand for synthesized hardware by the method in [6] and the proposed method, respectively. "Slices" of MIPS includes estimated cost of the instruction memory; one instruction (32bit) is assumed to cost one slice. "Delay" is the critical path delay. Subcolumn "total (0)" under "Cycles" is the total number of execution cycles when the

```
; 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: nop
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: nop
80000120: beqz t0,80000134
80000124: nop
80000128: jalr t0
8000012c: nop
80000130: nop
; 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: nop
800001c4: jr k0
800001c8: rfe
800001cc: nop
```

Figure 8: Interrupt handler [6].

main routine in Figure 7 was executed without any interrupt. Subcolumns "int (1)" and "total(1)" are for the cases where interrupts for int_getpixel were raised 1,000 times; "int (1)" is the average number of cycles taken from the occurrence of an interrupt to finish the task for the interrupt, and "total (1)" the number of the total execution cycles. Subcolumns "int (2)" and "total(2)" are for the cases where interrupts for int_average for 40×30 pixels were raised 10 times.

As in "int (1)," our method took much less execution cycles than MIPS and the previous methods. This is considered mainly because register save/restore operations were eliminated and partly because interrupt handler starts at any time other than the end of DFGs. Our method is especially effective in reducing interrupt latencies of light and frequent interrupts.

The total execution cycles of our method in "total (1)" and "total (2)" are fewer than those of [6] and more than 80% fewer than software execution. This is because of the fast interrupt handling and also because of parallel execution of the main and the handler modules. Especially in the case of this experiment, where there was no critical section and there was no need for mutual exclusion, interrupt handling did not increase total execution time.

Hardware cost ("Slices") is higher than that of MIPS (which includes estimated memory size for 1,753 instructions) by 41%, but this would be reasonable considering the improvement of the speed performance.

4.3 Synthesis of Motor Controller

As a more practical example, a controller of a motor driven by a timer is synthesized from a C program. The overview of the controller is shown in Figure 10 (a). It controls the rotational speed of a brushed DC motor by changing supply voltage. The control is sensorless; it does not use a speed sensor but estimates the speed

```
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, $k1" : "=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 $k1,$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 }
```

Figure 9: Library for interrupt handling (interrupt_lib.c) [6].

by a motor simulator. The voltage is periodically adjusted upon interrupts from the timer.

The detailed block diagram of the transfer function for the PID control, with a motor model inclusive, is shown in Figure 10 (b) [7]. Three elements PID, Gain, and Limiter constitute the controller, which receives the target angular velocity $\omega*$ and decides the voltage $x1$ to drive the motor. The value of $x1$ as well as ω is computed by expressing the transfer function by an ordinary differential equation and numerically solving it by the Runge-Kutta method.

A C program is described in the same framework as in Figure 7 through Figure 9. In this example, the main routine does nothing (just busy-loops) after initialization, while the handler is activated by the periodic interrupts from the timer.

A synthesis result is shown in TABLE 2. The logic synthesizer and the target FPGA are the same as those in TABLE 1. Note that our MIPS core was equipped with only fixed point arithmetic units

Table 1: Experimental result.

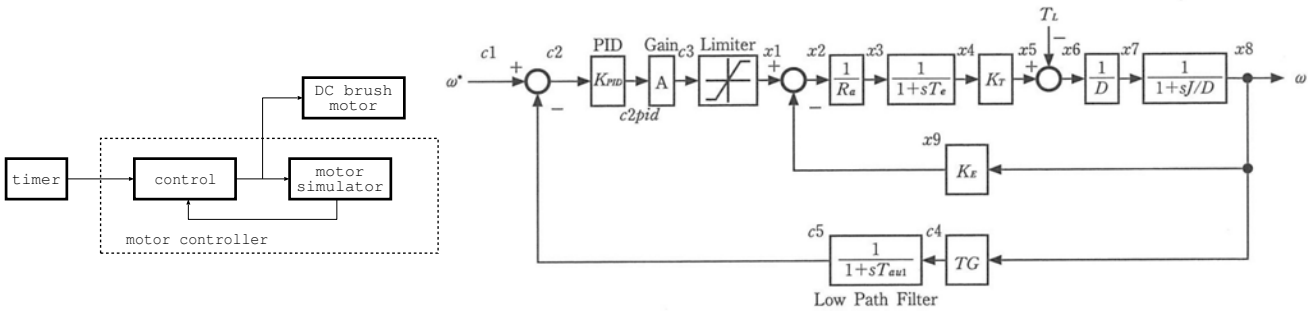
	Slices	Delay [ns]	Cycles				
			total (0)	int (1)	total (1)	int (2)	total (2)
MIPS	3,559 (1.00)	25.24 (1.00)	458,326 (1.00)	155 (1.00)	604,189 (1.00)	11,494 (1.00)	573,116 (1.00)
HW [6]	5,937 (1.67)	18.86 (0.75)	110,467 (0.22)	136 (0.88)	242,467 (0.40)	7,812 (0.68)	188,537 (0.33)
HW (proposed)	5,014 (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)

Logic synthesizer: Xilinx ISE 14.3, Target device: Xilinx Spartan-3E. Slices include estimated instruction memory size.

Table 2: Synthesis result of motor controller.

	Slices	Delay [ns]	Cycles	Period [μs]
MIPS	4,872 (1.00)	22.47 (1.00)	23,001 (1.00)	516.8 (1.00)
HW (proposed)	13,231 (2.72)	22.16 (0.99)	871 (0.04)	19.3 (0.04)

Logic synthesizer: Xilinx ISE 14.3, Target device: Xilinx Spartan-3E. Slices include estimated instruction memory size.



(a) Timer-driven sensorless control

(b) Block diagram of PID control [7]

Figure 10: Sensorless control of DC motor.

so floating point operations were carried out with GNU soft-float library, while the hardware had floating point units and ACAP converted the calls to the soft-float library by floating point arithmetic operations. As compared with software on the MIPS, the number of execution cycles ("Cycles") is drastically reduced, though circuit size ("Slices") is 2.72 times larger. The minimum period for the motor control has been reduced to 20.6 [μs], which enables extremely precise control.

5 CONCLUSION

This paper has proposed a method of synthesizing given binary programs with external interrupt handling into hardware where the normal task and the interrupt handler are implemented as separate modules. Efficient hardware compatible with the CPU running the binary code is generated without any rewriting on the binary code except for adding pragmas to specify which part should be synthesized as the handler module.

Future work includes extension of this method to multilevel interrupts and application of this method to other processors than MIPS.

ACKNOWLEDGMENTS

We would like to thank to Mr. Takayuki Nakatani who was with Ritsumeikan University, Mr. Masaharu Yano who was with Kyoto University, Mr. Shimpei Tamura who was with Kwansei Gakuin University and all the members of Ishiura Lab. of Kwansei Gakuin University for their discussion and advices on this research. This work is partly supported by JSPS KAKENHI under Grant Nos. 16K00088, 16K01207, and 15H02680.

REFERENCES

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

[2] Greg Stitt and Frank Vahid. 2007. Binary synthesis. *ACM Transactions on Design Automation of Electronic Systems* 12, 3, Article 34 (August 2007), 30 pages. DOI:http://doi.org/10.1145/1255456.1255471

[3] Nagisa Ishiura, Hiroyuki Kanbara, and Hiroyuki Tomiyama. ACAP: Binary Synthesizer Based on MIPS Object Codes. In *Proceedings of International Technical Conference on Circuit/Systems, Computers and Communications (ITC-CSCC 2014).* (July 2014), 725–728.

[4] Seiya Shibata, Shinya Honda, Hiroyuki Tomiyama, and Hiroaki Takada. 2010. Advanced System-Builder: A Tool Set for Multiprocessor Design Space Exploration. in *Proceedings of International SoC Design Conference (ISOCC 2010).* (November 2010), 79–82. DOI:http://doi.org/10.1109/SOCDC.2010.5682967

[5] Yuki Ando, Shinya Honda, Hiroaki Takada, and Masato Edahiro. 2015. System-level Design Method for Control Systems with Hardware-implemented Interrupt Handler. *IPSJ Journal of Information Processing*, 23, 5 (September 2015), 532–541. DOI:http://doi.org/10.2197/ipsjjip.23.532

[6] Naoya Ito, Nagisa Ishiura, Hiroyuki Tomiyama, and Hiroyuki Kanbara. High-Level Synthesis from Programs with External Interrupt Handling. in *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015).* (March 2015), 10–15.

[7] Hisashi Takahashi. 2017. *Introductory Lecture in Motor Control by the C Language* (in Japanese). Denpa Publications, Tokyo, Japan.