High-Level Synthesis from Programs with External Interrupt Handling

Naoya Ito [1] Nagisa Ishiura [1] Hiroyuki Tomiyama [2] Hiroyuki Kanbara [3]

[1] School of Science and Technology, Kwansei Gakuin University, Sanda, Hyogo, Japan
[2] College of Science and Engineering, Ritsumeikan University, Kusatsu, Shiga, Japan
[3] Advanced Scientific Technology & Management Research Institute of KYOTO, Kyoto, Kyoto, Japan

Abstract—This paper presents a method of synthesizing a given binary program, which contains external interrupt handling, into hardware whose behavior 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. Instructions for accessing coprocessor registers, returning from interrupt handling, and making system calls are scheduled as operations, and bound to the coprocessor. Jump register instructions for calling and returning from interrupt service routines are synthesized using operations that convert instruction 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. A program of about 40 lines with an external interrupt service routine was synthesized into hardware, and it was confirmed that interrupt handling works correctly. The execution cycles and the delay were reduced by 14% and 26% respectively, at the cost of 1.1 times increase in hardware size.

I. Introduction

While embedded systems are getting increasingly rich in functionalities, 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 [2, 3].

While high-level synthesis generates hardware from high-level behavioral languages such as C, *binary synthesis* [4] generates hardware from low-level programs written in assembly or machine languages. 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. It is even possible to replace a processor and a binary program by a faster or smaller hardware module by applying binary synthesis to the whole binary code.

However, for applications where processors control external devices, the programs contain interrupt processing. Since traditional high-level/binary synthesis methods have not assumed interrupts in behavior specification, such programs can not be automatically translated into hardware.

This paper presents a first attempt to extend the ability of binary synthesis to handle interrupts, where binary programs with interrupt handlers are synthesized into hardware without modification. In our method, a system control coprocessor which handles interrupts is incorporated into the hardware as one of the functional units. The instructions for accessing coprocessor registers, for returning from interrupt handling, and for making system calls are translated into operations which are scheduled and executed by the coprocessor. Jump register instructions to branch to interrupt service routines and to return from interrupt handling are also synthesized by using an address-to-state translation function. The proposed method has been implemented on top of a binary synthesizer ACAP [3]. A program of about 40 lines with an external interrupt service routine was successfully synthesized into hardware. The execution speed was accelerated by 1.6 times at the cost of 1.1 times increases in the hardware size.

II. Binary synthesis

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 high-level synthesis techniques are called binary synthesis.

It can handle wider range of software programs than conventional high-level synthesis, although back-end technologies to generate hardware are common. By using the assembly/machine languages as intermediate representation, it can accept multiple high-level languages for which compilers are available. It can also handle pointers and library calls in a natural way, though this may involve some loss in performance. While binary synthesis can be used to convert some computationally intensive parts of given software programs, it can also generate a hardware module which is functionally equivalent to the processor running a program, by synthesizing the binary code as a whole.

ACAP [3] is a binary synthesizer which generates hardware from MIPS R3000 binary codes. As shown in Fig. 1, it disassembles a binary code to recover an assembly program, and converts it into CDFG. It then performs scheduling and binding to generate a Verilog HDL code based on typical high-level synthesis algorithms. ACAP has the following three synthesis modes.

1. Separate compilation mode: Selected subprograms of a given program are converted into hardware modules which are callable from the software part.

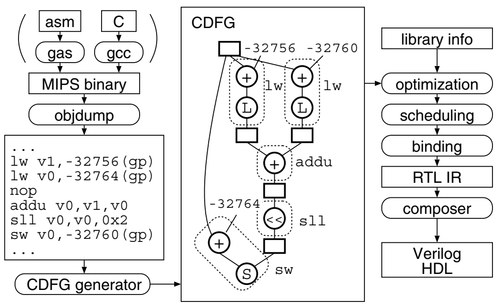


Fig. 1: Flow of synthesis in ACAP

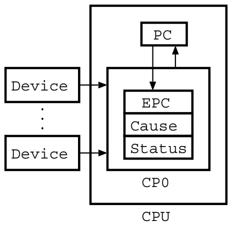


Fig. 2: CPU and system control coprocessor (CP0)

2. Full synthesis mode: A whole linked executable code is synthesized into a hardware module.
3. Accelerator synthesis mode: User specified sections of linked executable code are transformed into a hardware accelerator which is tightly coupled with the CPU.

This paper focuses on the second 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, owing to instruction level parallelism and operation chaining. The hardware size is also expected to be smaller, for the instruction memory is no longer necessary.

III. 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 interrput, 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.

The interrupt signals from external devices are sent to CP0, and then each of them is handled in the following way. Note that MIPS R3000 doesn't handle multilevel interrupt and all the other interrupts happened during one interrupt handling are ignored.

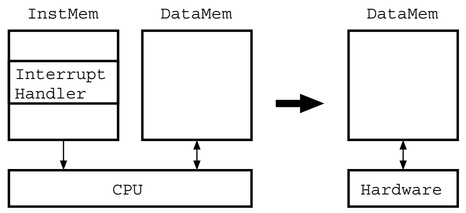


Fig. 3: Synthesis from a code with an interrupt handler

- CP0 saves the PC value in the `EPC` register and the cause of the interrupt in the Cause register, and updates the `Status` register so that the system may run in the kernel mode and interrupts will be prohibited.
- 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.
- 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.
- After the routine finishes its tasks, the handler restores the general purpose registers.
- 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)
  `mfc0 rt,rd` moves the value in a CP0 register `rd` to a general purpose register `rt`, and `mtc0 rt,rd` does the move 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.

IV. High level synthesis of external interrupt handling

A. *Overview*

This paper proposes a high-level synthesis method which can generate hardware from programs containing external interrupt handlers. As illustrated in Fig. 3, the proposed method translates a linked executable code including an interrupt handler to a hardware module, which replaces the CPU and the instruction memory.

In the following sections, we assume MIPS R3000 as a CPU. We also base our method on the following assumptions.

- When the hardware accepts an interrupt, the control is transfered to the interrupt handler at the end of the basic block where the interrupt is accepted.

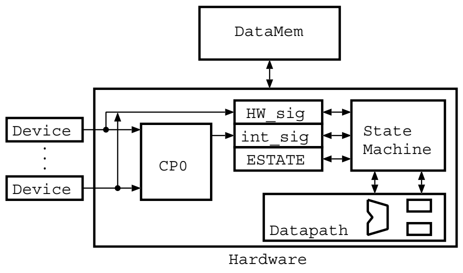


Fig. 4: The structure of synthesized hardware

- Multilevel interrupts are not considered. Any other interrupts during interrupt handling are ignored.
- The interrupt handler cannot refer to the instruction address where the interrupt occurred nor access to the instruction memory.

Although internal interrupts are not handled, system calls can be synthesized. This is to enable mutual exclusion of the accesses to shared resources.

In our method, the synthesized hardware module includes the CP0 coprocessor of MIPS R3000 as one of the functional units. Instructions, `mfc0`, `mtc0` and `rfe`, are synthesized and bound to CP0. Register jumps used to branch to interrupt service routines and to return from interrupts are synthesized by using an operation to translate instruction addresses to the corresponding states of the hardware.

B. Handling of external interrupts

The structure of the hardware generated by this method is shown in Fig. 4. The hardware has CP0 of MIPS R3000 as a functional unit and three registers `HW_sig`, `int_sig` and `ESTATE`.

The flow of interrupt handling is as shown in Fig. 5. DFG1 through DFG4 are DFGs (data flow graphs) corresponding to the basic blocks of a given program, and s0 through s28 are states (control steps). An external interrupt triggered during the execution of DFG1 is processed in the following way.

1. The interrupt signal is sent to CP0 (a).
2. The interrupt execution signal from CP0 is latched in the `int_sig` register (b).
3. At the last state (s3) of DFG1, `int_sig` is tested (c). 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) (e).
4. At the initial state of the handler, `int_sig` is cleared (f).
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.

C. Synthesis of instructions for interrupt handling

In the stage of high-level synthesis, `mfc0`, `mtc0`, `rfe`, and `syscall` instructions are dealt with as operations executed by CP0, so that they are scheduled and bound to CP0. They are processed as follows.

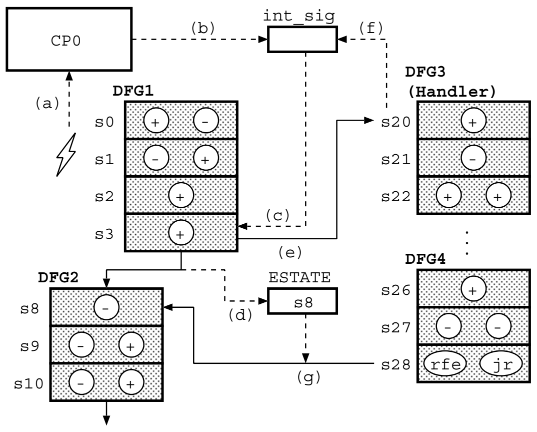
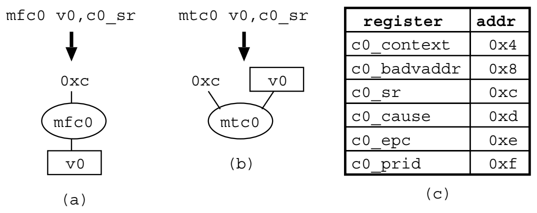


Fig. 5: Flow of interrupt handling by hardware



register	addr
c0_context	0x4
c0_badvaddr	0x8
c0_sr	0xc
c0_cause	0xd
c0_epc	0xe
c0_prid	0xf

Fig. 6: Operations corresponding to mfc0 and mtc0

- `mfc0`, `mtc0`
  Each of the `mfc0` and `mtc0` instructions is translated to an operations, as shown in Fig. 6 (a) and (b) respectively, where the source/destination register is selected by the numbers shown in Fig. 6 (c).
- `rfe`
  The `rfe` instruction is translated to an operation without input nor output and is scheduled at the last state of the DFG.
- `syscall`
  Before the scheduling phase, the basic block is divided just after the `syscall` instruction. The `syscall` instruction is translated to an operation without input nor output and is scheduled at the last state of the DFG so that it is executed independently from the other operations.

The `syscall` operation triggers a transition to the interrupt handler, just like external interrupts, by sending an execution signal of `syscall` to CP0. Since we assume the transitions to the interrupt handler occur only at the end of the basic blocks, an external interrupt and a system call may happen in a same basic block. In this case, the system call is ignored, since CP0 stores the information of an external interrupt in `Cause` register. In order to avoid this situation, we first handle external interrupts, and then process the system call. The revised flow of the process is shown in Fig. 7, where an external interrupt occurs during the execution of DFG1 which contains `syscall`.

- The external interrupt signal is sent to CP0 (a) and stored in the `int_sig` register as well as (c) in `HW_sig` register (b).
- At the last state (s3) of DFG1, `int_sig` and `HW_sig` are tested (d). If only one of them is 1, it is handled

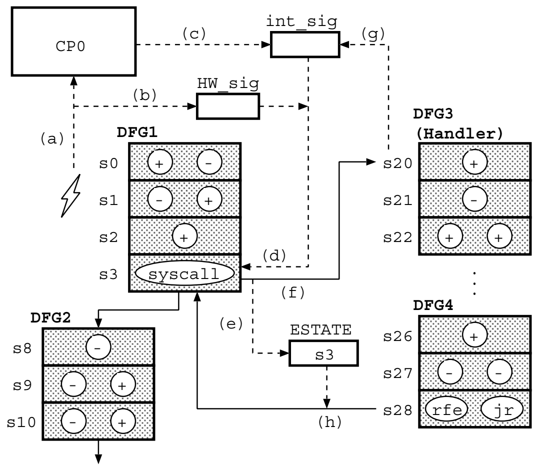


Fig. 7: Handling of syscall overlapping with an external interrupt

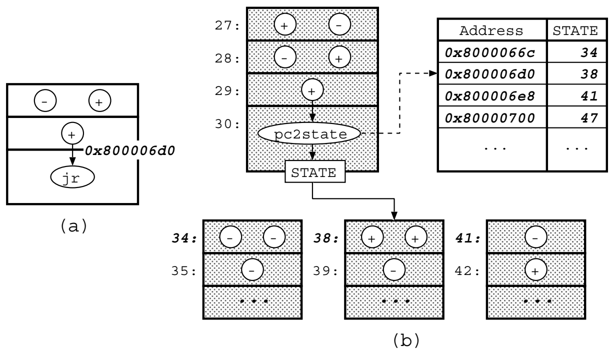


Fig. 8: Converting addresses into states

as described before. If both of them are 1, the current state (s3), instead of the next state (s8), is stored in ESTATE (e) and the hardware jumps to the interrupt handler (Handler) (f).

- At the first state (s20) of Handler, int_sig and HW_sig are cleared (g).
- At the last state (s28) of the handler, the hardware jumps to the state saved in ESTATE (h). If both of the external interrupt and the system call are present, the hardware returns to s3 and executes the system call.

D. *Synthesis of register jump instructions*

For the jalr and jr instructions used to select interrupt service routines and return from interrupt, their branch destinations cannot be determined by static analysis, because they depend on operand values computed at runtime. In this paper, these instructions are synthesized into hardware by using operations which convert instruction addresses into the corresponding states of the hardware.

For example, the jr instruction is translated into an operation whose input is a target address and is scheduled at the last state of the basic block as shown Fig. 8 (a). It is then further translated into a "pc2state" operation which converts a given program address to the corresponding state of the state machine, and writes the results to the state register as shown Fig. 8 (b). This conversion will be implemented as a combinational circuit.

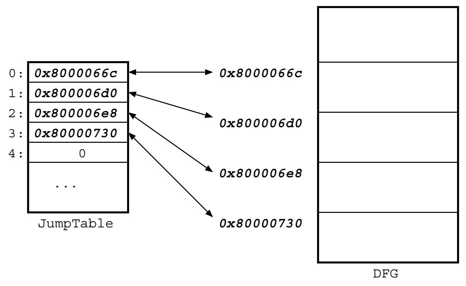


Fig. 9: Deviding basic blocks according to a jump table

Basically, the addresses that the pc2state operation has to convert are those of the entry points of the subroutines and the return addresses. However, jr instructions may be also used to implement switch-case statements, so their target addresses must be considered. Since the jump addresses of the switch-case statements are listed in a jump table, these addresses can be extracted from the table by scanning the data section of the binary code and enumerate the values which are multiples of 4 and within a range of the instruction addresses.

V. Experimental results

A. *Simulation results*

A binary synthesizer based on the proposed method has been implemented on top of ACAP. It is written in Perl5 and runs on Linux, Mac OS X, Cygwin on Windows, etc. The synthesized hardware is in the form of Verilog HDL.

A test program with external interrupt handling was written in C and assembly languages. After its behavior is verified on a MIPS R3000 compatible processor, the program is synthesized into hardware. The behavior of the resulting hardware was confirmed to be correct on the same test stimulus. The simulation is performed by Xilinx ISim 14.3.

Programs used in the experiment are shown in Fig. 10, 11, and 12. The program shown in Fig. 10 gets integer data by interrupts, stores them in a buffer, and calculates the sum of squares of the input data in the main routine. Upon the interrupt function int_routine (lines 33–39) is called and values in variable input, which is mapped to the external input, are moved to array buffer. The main routine (lines 14–31) calculates the sum of squares in the loop in lines 22–24. In lines 26–28, buffer is initialized under an exclusive control between interrupt prohibition function int_prohibition and an interrupt permission function int_permission. A registration of int_routine is performed in line 19. In the program, int_routine is registered to be called for the zeroth external interrupt (EXC_Int0). Also, function init_interrupt in line 18 performs initial setting to handle interrupts.

The program shown in Fig. 11 is the interrupt handler. 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 Fig. 12 consists of library functions to handle interrupts which is written in

```
#include "interrupt_lib.h"

#define N 10
#define LOOP_NUM 30

void int_routine(void);

volatile int input;

int sp = 5;
int buffer[N] = {10, 20, 30, 40, 50};
int output[LOOP_NUM];

int main(void)
{
  int i, j;

  init_interrupt();
  register_exc_handler(EXC_Int0, int_routine);

  for (i = 0; i < LOOP_NUM; i++) {
    for (j = 0; j < sp; j++) {
      output[i] += buffer[j] * buffer[j];
    }

    int_prohibition();
    sp = 0;
    int_permission();
  }
  return 0;
}

void int_routine(void)
{
  if (sp < N) {
    buffer[sp] = input;
    sp++;
  }
}
```

Fig. 10: A test program

```
; 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) Storeing the teturn address
80000104:  mfc0 ra,c0_epc
80000108:  sll zero,zero,0x0
8000010c:  sw ra,0(k0)

; 3) Calling the interrupt servece 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
```

Fig. 11: Interrupt handler

C and inline assembly. Function `init_interrupt` (lines 7–11) registers interrupt service routine execution function `run_exc_handler` (lines 20–41), which are called from the interrupt handler, and a routine `run_syscall_handler` (lines 43–53) for system call (`EXC_Sys`). Function `int_prohibition` (lines 55–59) is an interface function to set interrupt prohibition by calling `run_syscall_handler` through system call and then, calling `int_prohibiton_func` (lines 61–68). Function `int_permission` (lines 70–75) sets interrupt permission.

It was confirmed by simulation that the synthesized hardware switched to the interrupt handler at the end of the basic block in which the external interrupt signal was accepted, and that after the handler finished it resumed the original process. It is also confirmed that the CPU and the hardware produced exactly the same memory dump after the simulation, when the same numbers of interrupts were requested at the same timing. However, the CPU and the hardware behaved differently in terms of timing; the hardware branched to the handler at the end of the DFG where interrupt was accepted while the CPU switched to the handler just after the instruction at which the interrupt was signaled.

B. *Synthesis results*

The Verilog code generated from the test program was synthesized by Xilinx ISE 14.3 targeting the Xilinx Spartan-3E FPGA. Three ALUs were used but ALU chaining was not applied.

The result is summarized in TABLE. I. Rows “SW” and “HW” show the results of the software running on MIPS and synthesized hardware, respectively. “Slices”, “FFs”, “LUTs” and “Delay” are the numbers of slices, flip-flops, LUTs and delay times, respectively, and ”Cycles” indicates the number of cycles spent on the main process and the external interrupt handler. The presented method reduced the delay and execution cycles by 14% and 26%, respectively which results in a speed up by about 1.6 times, at the cost of hardware increase by about 1.1 times. The increase is due to the increase in binary code size; the interrupt handler and `interrupt-lib.c` compiled to 84 and 85 instructions, respectively, where codes for normal routines consisted of 94 instructions.

VI. Discussion

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 will not be saved nor restored on interrupt handling. ACAP guarantees that such intermediate registers don’t appear at the end of basic blocks, so we can just let the synthesized hardware save and restore the registers, as long as the jumps occur only at the end of the DFG.

If we can completely separate the codes for interrupt handling from those for normal processing and can synthesize them into independent hardware modules, we may be able to get rid of this restriction. We are now working on this issue.

TABLE I: Result of synthesis

Target	Slices	FFs	LUTs	Delay [ns]	Cycles
SW	3144 (1.00)	1771 (1.00)	5632 (1.00)	25.24 (1.00)	5192 (1.00)
HW	3563 (1.13)	2003 (1.13)	6801 (1.21)	21.57 (0.85)	3816 (0.74)

```
#include "interrupt_lib.h"

extern void *interrupt_call;
extern void (*exc_handler[24])();
extern void *_reg_store;

void init_interrupt(void)
{
  (*((unsigned int*)&interrupt_call))
            = (unsigned int)run_exc_handler;
  register_exc_handler(EXC_Sys, run_syscall_handler);
}

void register_exc_handler(unsigned int exc, void (*f)(void))
{
  if (EXC_MOD <= exc && exc <= EXC_Int5) {
    exc_handler[exc] = f;
  }
}

void run_exc_handler(void)
{
  int i;
  unsigned int cause_reg, exc_code, int_field;
  void (*handler)() = 0x00000000;

  asm("mfc0 %0, $13" :  "=r" (cause_reg));
  exc_code = (cause_reg >> 2) & 0xf;
  int_field = (cause_reg >> 8) & 0xff;

  if (!exc_code) {
    for (i = EXC_Sw0; i <= EXC_Int5; i++) {
      if (int_field & 0x1) {
        handler = exc_handler[i]; break;
      }
      int_field = int_field >> 1;
    }
  } else {
    handler = exc_handler[exc_code];
  }
  if (*handler) {(*handler)();}
}

void run_syscall_handler(void)
{
  unsigned int reg_k1;

  asm("add %0, $0, $27" :  "=r" (reg_k1));

  if (reg_k1 == 1) {
    int_prohibition_func();
    ((unsigned int *)&_reg_store)[0] += 4;
  }
}

void int_prohibition(void)
{
  asm("addiu $27,$0,1");
  asm("syscall");
}

void int_prohibition_func(void)
{
  asm("lui $8, 0xffff");
  asm("ori $8, 0xfffb");
  asm("mfc0 $9, $12");
  asm("and $9, $9, $8");
  asm("mtc0 $9, $12");
}

void int_permission(void)
{
  asm("mfc0 $8, $12");
  asm("ori $8, $8, 0x1");
  asm("mtc0 $8, $12");
}
```

Fig. 12: interrupt_lib.c

In this paper, internal interrupts (software interrupts) are not considered. This is partly because of the register save/restore problem and also because architecture dependent codes are often written in internal interrupt handlers which are very difficult synthesize into equivalent hardware. If they are written as architecture independent or rewritten to work on synthesized hardware, it should not be difficult to synthesize the handlers.

VII. Conclusion

We have presented a high-level synthesis for programs with an external interrupt handling. The presented method has been implemented and it was confirmed that the interrupt handler runs correctly.

We are now working on response improvement discussed in Section IV, as well as trying to reduce the area and to improve the execution time of generated hardware.

Acknowledgements

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.

References

[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).