Binary Synthesis from ARM Machine Code Using a General-Purpose High-Level Synthesis System

Yuga SUGIMOTO † Nagisa ISHIURA ††

† Graduate School of Science and Technology †† School of Engineering
Kwansei Gakuin University
1 Gakuen Uegahara, Sanda, Hyogo, 669-1330, JAPAN

Abstract—This paper presents the implementation of a binary synthesis system that utilizes a general-purpose high-level synthesis (HLS) system as a back-end, starting from ARM Thumb machine code. Binary synthesis is a technique that generates a hardware design description functionally equivalent to the CPU executing the original machine code program. However, implementing such a system for each instruction set architecture incurs a high development cost. Nakamichi has proposed a method for easily implementing binary synthesis systems by converting machine code programs into C programs, which are then synthesized into hardware using a general-purpose HLS system. Based on this approach, we implement a binary synthesis system that generates hardware design descriptions from machine code programs using the ARMv6 instruction set. ARMv6 includes features such as program counter-relative addressing for immediate load instructions and conditional execution based on flags. Additionally, division and modulo operations are typically handled by runtime libraries, which increase circuit size and execution time; to address this, we convert runtime library calls directly into division/modulo instructions. The proposed binary synthesis system is implemented in Python. Experimental results using machine code programs obtained from several C programs show that, compared to hardware generated directly from the original C programs, the circuit size increased by a factor of approximately 1.3 to 5.5, while the delay remained nearly the same.

I. Introduction

In recent years, embedded systems are expected to deliver numerous functions and high performance, while also facing strict constraints such as miniaturization and power efficiency. When these demands cannot be met through software implementation alone, one possible approach is to migrate part or all of the system to hardware.

As a means of facilitating efficient migration from software to hardware, high-level synthesis (HLS) techniques have been proposed, which generate hardware design descriptions from programs written in C or other high-level languages [1]. However, programs that include special instructions, such as interrupt handlers, are often written in assembly or using inline assembly, making it difficult to directly apply HLS to such software.

Binary synthesis is a technique that generates hardware designs from machine code programs. For example, hardware synthesis from MIPS, ARM, and MicroBlaze machine codes has been demonstrated in [2], from MIPS in [3], and from RISC-V in [4]. Moreover, [5, 6] propose binary synthesis methods that target programs including MIPS interrupt handlers.

Binary synthesis offers advantages such as the ability to synthesize special instructions not directly representable in programming languages and memory accesses through pointers or global variables without code modification. Additionally, hardware synthesis helps prevent code or algorithm theft. On the other hand, binary synthesis systems must be implemented separately for each instruction set, which increases development cost. Even if existing HLS tools can be reused, analyzing data flow and control in the machine code program may still be necessary.

To address this issue, Nakamichi et al. have proposed a method for easily implementing binary synthesis systems using general-purpose HLS tools [7]. This method generates a high-level language program representing the behavior of the given machine code program and synthesizes it into hardware using commercial HLS tools. Nakamichi implemented a binary synthesis system for 32-bit RISC-V (RV32IM) machine code based on this method.

In this paper, we implement a binary synthesizer targeting ARMv6 Cortex-M1 machine code programs based on Nakamichi's method. In contrast to RISC-V, ARM presents two notable differences: (1) constants for immediate load instructions are not embedded in the instruction itself but are located elsewhere in the program and accessed relative to the program counter, and (2) instructions may be executed conditionally based on status flags. Moreover, since division and remainder operations executed by the runtime library become significantly inefficient when directly implemented in hardware, we also propose a method to convert such function calls into equivalent C code performing the division or remainder computation.

We conducted evaluation experiments using ARM machine code programs obtained by compiling C programs. Compared to circuits synthesized directly from the original C programs by HLS, the proposed system produced circuits with a hardware size increase of approximately 1.3 to 5.5 times, while maintaining nearly the same critical path delay.

The remainder of this paper is organized as follows. Section 2 reviews the proposed binary synthesis approach using a general-purpose high-level synthesizer as the back-end. Section 3 describes the technical details of applying the method to the ARM Thumb instruction set. Section 4 presents the implementation of the binary synthesizer and compares its performance with that of high-level synthesis from C programs. Finally, Section 5 concludes the paper and outlines future work.

II. Binary Synthesis Using General-Purpose High-Level Synthesis Systems

A. *High-level synthesis and binary synthesis*

The general flow of high-level synthesis is shown in Fig. 1. The input behavioral description written in a programming language is converted into an intermediate representation called a CDFG (control dataflow graph). Then, scheduling determines the execution timing of operations, and binding allocates operations and values to functional units and registers, respectively, finally generating a register-transfer level hardware description.

Binary synthesis differs from high-level synthesis only in the front-end; once the system analyzes machine code programs to generate a CDFG, the subsequent process is the same as in high-level synthesis.

Since most high-level synthesis systems do not accept CDFG as their input interface, the entire binary synthesis system must be implemented. If existing high-level synthesis or binary synthesis systems can be reused, or if an open-source HLS system is available, only the front-end needs to be implemented. Nevertheless, since generating a CDFG usually requires data-flow and control-flow analysis, developing a front-end for machine code programs involves significant implementation effort.

B. *Binary synthesis using general-purpose high-level synthesis systems*

As a simpler method for implementing binary synthesis systems, Nakamichi et al. proposed a method that utilizes a high-level synthesis system as a backend [7]. This method converts the given machine code program into a C program, which is then synthesized into hardware using an off-the-shelf HLS tool.

The flow of this process is shown in Fig. 2. The input is a linked executable machine code program. The disassembled assembly code is converted into a synthesizable

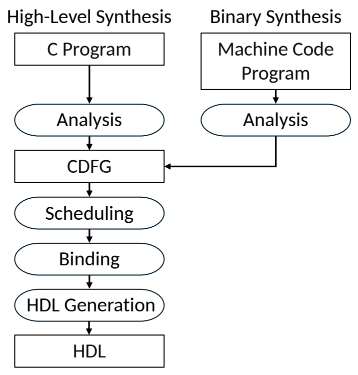


Fig. 1. Flow of high-level synthesis/binary synthesis [7]

C program that achieves the same functionality as the given program. As described later, this transformation is not a decompilation, but rather a method of converting each instruction one-by-one into corresponding C code. Because dataflow and control analysis are not required, development cost is reduced. Any HLS system that accepts C code as input can be used, including not only pre-developed systems but also commercial ones, though some system-specific adjustments may be needed. In addition, one can benefit from on-going improvements in the performance of HLS tools.

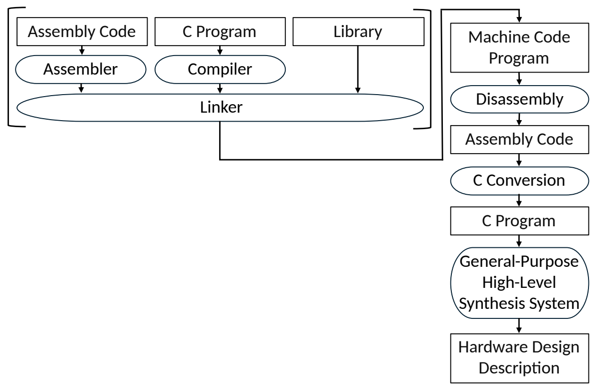
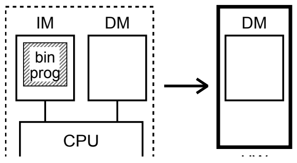


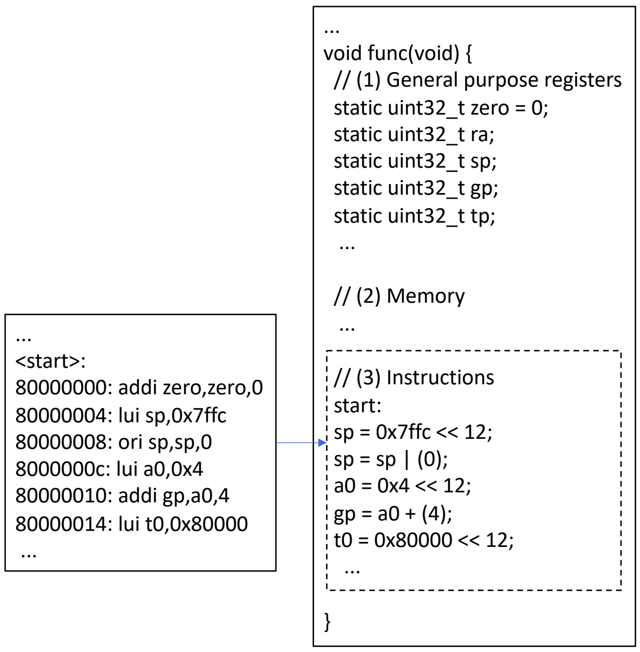
Fig. 2. Flow of binary synthesis proposed in [7]

As shown in Fig. 3, the binary synthesis presented in [7] converts a program stored in instruction memory, along with the CPU that executes it and other components such as data memory, into a single hardware module that is functionally equivalent to the original system.

Regarding the structure of the C program, one C function is generated for each assembly program in [7], as shown in Fig. 4[1]. The generated function consists of a part that declares registers and memory as local variables

[1]This is because the HLS tool used in the implementation synthesizes one function into one hardware module





```
...
<start>:
80000000: addi zero,zero,0
80000004: lui sp,0x7ffc
80000008: ori sp,sp,0
8000000c: lui a0,0x4
80000010: addi gp,a0,4
80000014: lui t0,0x80000
 ...
```

```
...
void func(void) {
  // (1) General purpose registers
  static uint32_t zero = 0;
  static uint32_t ra;
  static uint32_t sp;
  static uint32_t gp;
  static uint32_t tp;
   ...

  // (2) Memory
   ...

  // (3) Instructions
  start:
  sp = 0x7ffc << 12;
  sp = sp | (0);
  a0 = 0x4 << 12;
  gp = a0 + (4);
  t0 = 0x80000 << 12;
   ...
}
```

Fig. 4. Structure of generated C program in [7]

(sections (1) and (2) in the figure), and a part that consists of C code fragments each of which is converted from an instruction line (section (3) in the figure). In this example, the 32-bit integer registers in RISC-V are declared as 32-bit unsigned integer variables as shown in (1).

Examples of converting RISC-V instructions into C code are shown in TABLE I . For instance, the addition instruction `add a5,a4,a5` is converted into the C statement `a5 = a4 + a5;`.

TABLE I
Examples of RISC-V Instruction Conversion in [7]

Instruction	C Program
`add a5,a4,a5`	`a5 = a4 + a5;`
`addi sp,sp,-32`	`sp = sp + (-32);`
`and a4,a4,a5`	`a5 = a4 & a5;`
`sll a5,a5,a0`	`a5 = a5 << (a0 & 31);`
`sra a6,a4,a6`	`a6 = SINT32(a4) >> (a6 & 31);`
`srl a7,a4,a7`	`a7 = a4 >> (a7 & 31);`

Memory is represented as arrays within the function (Fig. 6), whose handling is shown in Fig. 5. The heap (global variables) is assumed to use GN words, and the stack (local variables) LN words. Macros like those in Fig. 7 convert memory addresses to array indices; the index for address `a` is given by `MA(a)`.

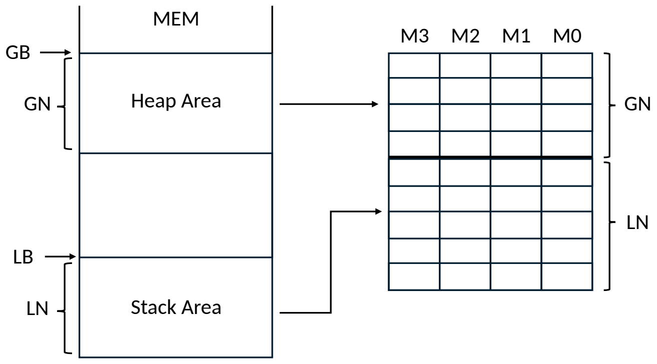


Fig. 5. Memory handling in [7]

```
static uint8_t MEM_3[GN+LN];
static uint8_t MEM_2[GN+LN];
static uint8_t MEM_1[GN+LN];
static uint8_t MEM_0[GN+LN];
```

Fig. 6. Declaration of memory array variables in [7]

```
#define LB 0x7ffbfec
#define GB 0xc0000000
#define MA_G(a) ( (((a) - GB) / 4) )
#define MA_L(a) ( (((a) - LB) / 4) + GN )
#define MA(a) ( ((a) >= GB) ? MA_G(a) :  MA_L(a) )
```

Fig. 7. Converting memory address to index in [7]

Branch instructions are implemented by attaching labels to the C code at target addresses and using `goto`. For register jumps, all possible targets and corresponding labels are enumerated, and a `switch` statement performs the jump.

III. Instruction Conversion Method for ARM

In this paper, we present an implementation of a binary synthesis system from ARM machine code based on the method proposed in [7]. Our target is the Cortex-M1 (Thumb of ARMv6).

The method of converting machine code programs into C programs is basically the same as for 32-bit RISC-V (RV32IM), but several modifications are made for ARM Thumb instructions. In Thumb, immediate load instructions use PC-relative addressing, so it is necessary to extract the corresponding values from the machine code. Moreover, since ARM allows conditional execution using a flag register, the conversion must handle this behavior appropriately. Additionally, in Cortex-M1, division and modulo operations are executed through runtime libraries. Directly converting these library calls into C and applying high-level synthesis results in significant inefficiency; therefore, we convert these calls directly into C code that performs division or modulo operations directly.

A. *Conversion of basic Thumb instructions*

As in [7], 32-bit general-purpose registers are declared as 32-bit unsigned integer local variables, and memory is handled as local arrays. Examples of basic Thumb instruction conversions are shown in TABLE ?? . For example, the addition instruction `add r0,r1` is converted into the C statement `r0 = r0 + r1;`. In Thumb, instructions with the same name may have different numbers of operands, so conversions are made accordingly. The conversion of load/store instructions (`ldr` and `str`) and branch instructions (`b.n`) is similar to that of RISC-V.

TABLE II
Conversion of basic Thumb instructions

Instruction	C Program
`add r0,r1`	`r0 = r0 + r1;`
`add r0,r1,r2`	`r0 = r1 + r2;`
`sub r3,#1`	`r3 = r3 - 1;`
`sub r3,r4,#1`	`r3 = r4 - 1;`
`mov r3,r1`	`r3 = r1;`
`neg r3,r3`	`r3 = -r3;`
`lsl r2,r2,#2`	`r2 = r2 << (2 & 31);`
`asr r2,r3,#1`	`r2 = (int32_t)r3 >> (1 & 31);`

`ldr r4,[r6,#4]`	`r4 = (MEM_3[MA(r6+4)] << 24)`
	`+ (MEM_2[MA(r6+4)] << 16)`
	`+ (MEM_1[MA(r6+4)] <<  8)`
	`+ (MEM_0[MA(r6+4)]       );`

`str r3,[r6,#4]`	`MEM_3[MA(r6+4)] = (r3 >> 24);`
	`MEM_2[MA(r6+4)] = (r3 >> 16);`
	`MEM_1[MA(r6+4)] = (r3 >>  8);`
	`MEM_0[MA(r6+4)] = (r3       );`

`b.n 80000182`	`goto L80000182;`

B. *Conversion of program counter-relative load instructions*

Since Thumb instructions are only 16 bits in length, there is insufficient space within the instruction to accommodate an immediate field large enough to load constants into registers. Therefore, the Thumb instruction set adopts a scheme where constants are placed within the program, and loaded into registers using program counter-relative load instructions (`ldr`).

For example, in Fig. 8, the instruction `ldr r6, [pc, #44]` uses program counter-relative addressing to load a 32-bit constant value `0xc0000000`, located at address `0x80000194` in instruction memory, into register `r6`. This instruction needs to be converted into C code like `r6 = 0xc0000000;`. However, two issues arise during this conversion.

The first issue is that the runtime address obtained by adding an offset to the value of `pc` may fall outside the

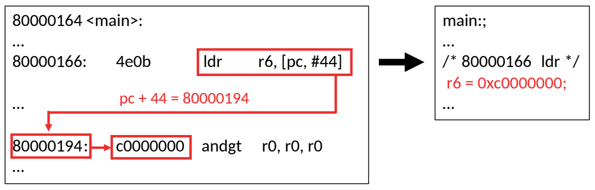


Fig. 8. Constant value reference via PC-relative addressing

binary range converted into an assembly program via disassembly. The second issue is that when the address falls within range, the data at that address may have been disassembled into unrelated instructions, which can result in invalid code.

In this paper, we resolve the first issue by extracting the constant data not from the assembly code, but from a separately dumped machine code file. The second issue is addressed by setting flags on instructions located at addresses identified as data in the assembly code, thereby excluding them from conversion into C code.

It should be noted that when calculating program counter-relative addresses, it is necessary to consider alignment to word boundaries and the data endianness.

C. *Handing of condition flags*

Unlike MIPS or RISC-V, ARM has mechanisms for setting flags through instructions and for conditional execution of instructions based on those flags. The flags include four types: N, Z, C, and V, which are set when the result of an instruction is negative, zero, when an unsigned overflow occurs, or when a signed overflow occurs, respectively.

In our approach, flags are declared as local variables just like general-purpose registers. Instructions that set flags are converted into C code that performs the computation and updates the flags according to their definitions. Conditionally executed instructions are converted into if-statements that reference these flags.

Fig. 9 shows examples of conversions for the addition instruction `adds`, which updates flags, and for conditional branching based on flag values. After executing the addition `r1 = r1 + r2`, the values of the four flags are computed. The variable `r1_org` stores the original value of `r1` prior to the update, which is needed to determine overflow. The `bne.n` instruction branches when the Z flag is cleared, and this is represented using an if-statement.

Although computing flags that are not referenced by subsequent instructions is redundant, such calculations are typically removed by general-purpose high-level synthesis tools through dead code elimination, which is a basic optimization performed by compilers.

Assembly	C Program
80000164 <main>:	main:;
...	...
80000182: adds r1, r2	/* 80000182 adds */
	r1_org = r1;
	r1 = r1 + r2;
	n = (r1 >> 31);
	z = (r1 == 0);
	c = (r1 < r1_org);
	v = ((r1_org > 0) && (r2 > 0)
	&& (r1 < 0)) || ((r1_org < 0)
	&& (r2 < 0) && (r1 > 0)) ;
80000184: bne.n 80000176	/* 80000184 bne.n */
	if (z == 0) {
	goto L80000176;
	}
...	...

Fig. 9. Flag update and reference

Assembly	C Program
...	...
80000140 <__aeabi_idivmod>:	
80000140: e3510000 cmp r1, #0	
80000144: 0afffff9 beq 80000130	
80000148: e92d4003 push r0, r1, lr	
8000014c: ebffffb3 bl 80000020	
80000150: e8bd4006 pop r1, r2, lr	
80000154: e0030092 mul r3, r2, r0	
80000158: e0411003 sub r1, r1, r3	
8000015c: e12fff1e bx lr	
...	...
80000164 <main>:	main:;
...	...
8000017a: f7ff ffe1 bl 80000140 <__aeabi_idivmod>	/* 8000017a bl */
	r0_org = r0;
	r0 = r0_org / r1;
	r1 = r0_org % r1;
...	...

Fig. 10. Conversion of runtime library calls for division/modulo

D. *Conversion of runtime library calls for division/modulo operations*

Since Cortex-M1 does not have divison/modulo instructions, these computations are carried out via calls to runtime libraries. In our method, although it is possible to synthesize hardware from machine code that includes linked runtime libraries, the division/modulo libraries involve a large number of both static and dynamic instructions. As a result, the generated hardware circuit size and the number of execution cycles tend to increase significantly.

To address this, our method directly converts runtime library calls for these operations into corresponding C code that performs division and modulo operations.

For example, in the assembly code shown in Fig. 10, the code starting at address `80000140` labeled `<__aeabi_idivmod>` corresponds to the division/modulo runtime library. By setting the dividend in `r0` and the divisor in `r1`, and invoking the runtime library with the instruction `bl 80000140` at address `8000017a`, the quotient is returned in `r0` and the remainder in `r1`. In our approach, the `bl 80000140` instruction is converted into C code that directly performs division and modulo operations. Additionally, the runtime library code itself is excluded from the conversion and removed.

This transformation is expected to significantly reduce both the size of the synthesized hardware circuit and the number of execution cycles.

IV. Implementation and Experiment

We have implemented a binary synthesis system that converts ARM Thumb machine code programs into hardware based on the proposed method.

We used 'arm-none-eabi-objdump' as the disassembler, and implemented the system that converts the assembly code into C programs using Python (version 3.11.7). The instruction sequence of the assembly program was extracted from the '.text' section, while constant data referred to by PC-relative `ldr` instructions was extracted separately from the '.data' section. Since immediate values extracted from machine code are stored in little-endian format, they were converted to big-endian during this process.

The machine code programs used in the experiments were generated by compiling C programs. We used 'gcc-11.4.0' targeting 'arm-none-eabi', with optimization option '-O3.' For high-level synthesis as the backend, we used Xilinx Vitis HLS (version 2024.1.2), targeting the Xilinx Artix-7 device.

The results are shown in TABLE III . The evaluated programs are as follows: "prime": primality testing, "heap_sort": heap sort, "binary_search": binary search on an integer array, "linear_search": linear search, and "best_first_search": best-first search.

The "proposed binary synthesis" column shows the results of our binary synthesis approach using ARM machine code, while "high-level synthesis" shows the results obtained by synthesizing the original C programs directly as a reference. (The purpose of this comparison is to evaluate the overhead incurred by the proposed binary synthesis system, if any, in comparison to high-level synthesis from a C program. It should be noted that the goal of this paper is not to develop a binary synthesis system that outperforms high-level synthesis.)

"#insn" indicates the number of static instructions in the assembly program, "#LUT" is the number of lookup tables used, "#cycle" is the number of execution cycles, and "delay" is the critical path delay in nanoseconds. Values in parentheses indicate the ratio relative to the "high-level synthesis" result, which is normalized to 1.00.

The execution results of the circuits generated by binary synthesis matched those of the high-level synthesis exactly. Compared to high-level synthesis, the binary synthesis circuits required: 1.29 to 8.66 times more LUTs and 1.29 to 3.83 times more execution cycles. The critical path

TABLE III
SYNTHESIS RESULT

program	proposed binary synthesis				high-level synthesis		
	#insn	#LUT	#cycle	delay [ns]	#LUT	#cycle	delay [ns]
prime	29	2,118 (1.29)	270 (1.29)	6.859 (0.98)	1,637	210	6.979
heap_sort	165	4,074 (5.47)	58,066 (3.65)	7.176 (0.99)	745	15,911	7.240
binary_search	38	851 (2.55)	92 (3.83)	7.043 (1.04)	334	24	6.755
linear_search	29	537 (8.66)	123 (3.73)	7.046 (1.18)	62	33	5.959
best_first_search	49	1,103 (5.52)	163 (2.96)	7.043 (1.00)	200	55	7.014

Synthesizer: Xilinx Vitis HLS (2024.1.2), Target: Xilinx Artix-7

delay remained nearly the same.

Because binary synthesis lacks structural information from source programs (unlike high-level synthesis), the resulting hardware is generally less efficient. ARM Thumb programs also include more instructions, affecting circuit size and execution cycles. These performance issues depend heavily on the backend high-level synthesis tool's optimization capabilities, and generating more optimization-friendly C code could help improve results.

V. Conclusion

In this paper, we presented an implementation of a binary synthesis system from ARM machine code that utilizes a general-purpose high-level synthesis (HLS) system as its backend. The key technical contributions include: PC-relative immediate value loading, flag updates and conditional execution handling, and direct conversion of division/modulo runtime library calls into corresponding arithmetic operations in C.

This method enables the implementation of a binary synthesizer at low cost. It is especially useful for hardware synthesis of ARM machine code in cases where no source code in a high-level language exists, when the program includes inline assembly, or when there is a need to protect the machine code program. Another advantage of this approach is that the performance of the binary synthesizer can benefit from ongoing improvements to the high-level synthesizer used as the backend.

The size of the circuits and the number of execution cycles generated by the binary synthesizer using this method tend to be larger than those obtained by directly performing high-level synthesis from a C program (if such a program exists). One possible approach to mitigate this issue is to convert the machine code program into a C program that is more suitable for optimization by the high-level synthesis system.

Future work also includes supporting 32-bit ARM and SIMD instructions.

Acknowledgements

Authors would like to express their appreciation to Dr. Hiroyuki Kanbara of ASTEM/RI, Prof. Hiroyuki Tomiyama of Ritsumeikan University, and Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their discussion and valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University. This work was partly supported by JSPS KAKENHI under Grant No. 24K14885.

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*, Springer (2012).

[2] G. Stitt and F. Vahid: "Binary synthesis," *ACM Trans. Design Automation of Electronic Systems (TODAES)*, Vol. 12, No. 3, pp. 1–30 (Aug. 2008).

[3] N. Ishiura, H. Kanbara and H. Tomiyama: "ACAP: Binary synthesizer based on MIPS object codes," in *Proc. International Technical Conf. on Circuit/Systems, Computers and Communications (ITC-CSCC 2014)*, pp. 725–728 (July 2014).

[4] S. Hamana and N. Ishiura: "Binary synthesis from RISC-V executables," in *Proc. Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI 2019)*, pp. 227–228 (Oct. 2019).

[5] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI 2015)*, pp. 10–15 (Mar. 2015).

[6] N. Ito, Y. Oosako, N. Ishiura, H. Kanbara and H. Tomiyama: "Binary synthesis implementing external interrupt handler as independent module," in *Proc. International Symposium on Rapid System Prototyping (RSP 2017)*, pp. 92–98 (Oct. 2017).

[7] R. Nakamichi, S. Kishimoto, N. Ishiura, and T. Kondo: "Binary synthesis using high-level synthesizer as its back-end," in *Proc. Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI 2022)*, pp. 121–126 (Oct. 2022).