# Binary Synthesis Using High-Level Synthesizer as its Back-End

Ryo NAKAMICHI [†] Sho KISHIMOTO [†] Nagisa ISHIURA [††] Takumi KONDO [†]

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

**Abstract—This paper presents a facile way to implement binary synthesizers using existing high-level synthesizers as their back-ends. Binary synthesis is a variant of high-level synthesis which translates binary programs into register transfer level hardware models. In the proposed method, C programs in place of CDFGs (control dataflow graphs) are generated from binary programs, which are synthesized into hardware by high-level synthesis. Based on the proposed method, a binary synthesizer for RISC-V (RV32IM) has been implemented using Xilinx Vivado HLS as a back-end high-level synthesizer. The execution cycles and critical path delay of the synthesized circuits, generated from RV32IM binaries compiled from C programs, are almost the same as those of the circuits generated by the high-level synthesizer from the C programs, though the circuit size is 1.00 to 3.32 times larger.**

## I. Introduction

While increasingly rich functions are implemented in embedded systems, there is constant and severe requirement for size and power consumption of the embedded devices. One of the approaches to this issue is to implement some critical parts of the systems, or even the hole systems, as hardware.

The cost of hardware design is much higher than software design, for the level of abstraction is generally lower in hardware than in software. As a tool to improve efficiency of hardware design, high-level synthesis has been proposed [1]. Behavior specification in high-level programming languages such as C is automatically synthesized into hardware, which drastically reduces the cost of hardware design.

However, high-level synthesis can not be used to convert all the existing software programs into hardware. Especially, systems that control external devices must handle interrupts and use special instructions to access dedicated registers. When these programs are written in assembly languages or written using inline assembly, high-level synthesis is not directly applicable.

In such a case, *binary synthesis* may be used as an alternative. Binary synthesis generates hardware from binary programs, rather than programs in high-level languages. Stitt, et al. developed binary synthesizers for MIPS, ARM, and MicroBlaze [2]. Ito, et al. synthesized systems that include interrupt handlers written in MIPS assembly into hardware [3, 4] using a binary synthesizer ACAP [5]. Binary synthesizers may be used to replace a CPU running a control binary program without a source code by hardware, or to protect the binary codes from reverse engineering.

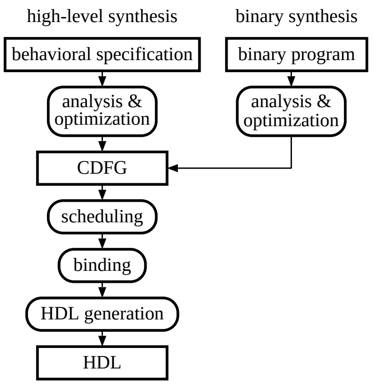


Fig. 1. Typical flow of high-level/binary synthesis

Unlike high-level synthesizers, binary synthesizers are machine independent. That means, a binary synthesizer must be developed for each instruction set architecture. An easy way to develop a binary synthesizer is to reuse existing high-level synthesizers or open source high-level synthesizers such as LegUp [6] and to implement only a front-end that generates intermediate data such as a CDFG (control dataflow graph). However, even this requires substantial effort, for it may involve dataflow analysis, program-level optimization, or construction of variable tables.

This paper proposes a facile method for developing binary synthesizers using high-level synthesizers as their back-ends. A C program that emulates the behavior of a given binary program is generated, which is fed to high-level synthesis. The binary to C translator is easy to develop, for its major task is one by one conversion of an instruction to C statements and there is no need for further analysis and optimization.

Based on the proposed method, a binary synthesizer for RISC-V (RV32IM) has been developed. An experiment on binary programs compiled from C programs has shown that synthesized circuits run as fast as the ones generated by high-level synthesizers, though the circuits become up to 3.32 times larger.

## II. High-level synthesis and binary synthesis

High-level synthesis [1] is a design automation technology which generates register transfer level description of hardware

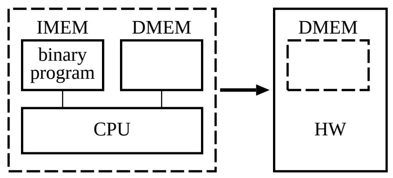


Fig. 2. Type of binary synthesis assumed in this paper

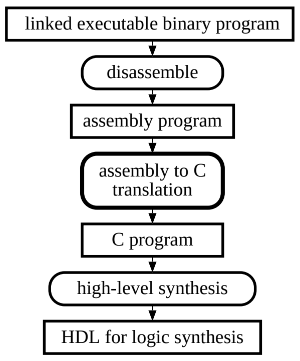
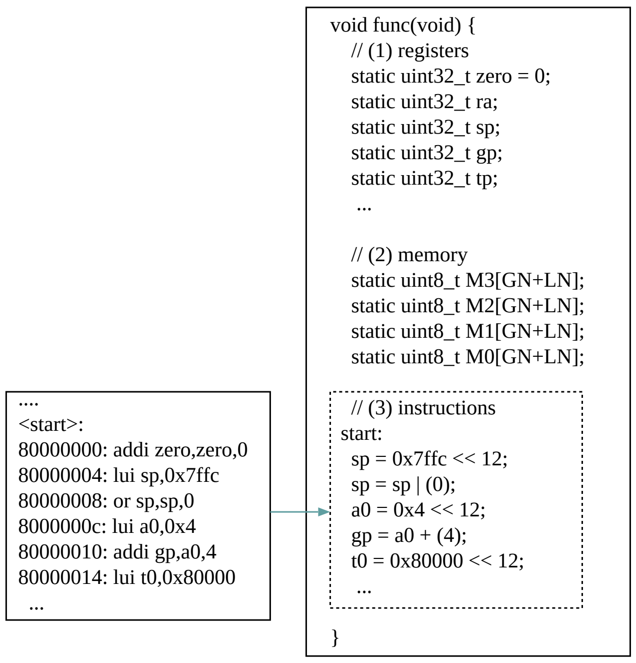


Fig. 3. Proposed flow of binary synthesis



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

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

   // (2) memory
   static uint8_t M3[GN+LN];
   static uint8_t M2[GN+LN];
   static uint8_t M1[GN+LN];
   static uint8_t M0[GN+LN];

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

}
```

Fig. 4. Configuration of generated C programs

| | |
|---|---|
| `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);` |

Fig. 5. Translation of ALU instructions

from behavioral specification written in procedural languages such as C.

Typical flow of high-level synthesis is shown in Fig. 1. A given behavioral specification is analyzed to generate an intermediate representation called a CDFG (control dataflow graph). This analysis may involve dataflow analysis and optimization using the information in the program. Scheduling and binding are performed on the CDFG, from which HDL (hardware design language) ready for logic synthesis is generated.

Programming languages such as C assume a computation model where variables are mapped in the single memory space and are accessible by their addresses. On the other hand, high-level synthesizers generate hardware of a different computation model where variables does not have explicit addresses. Thus, when functions or threads in input programs have global variables or share variables via pointers, these accesses must be rewritten to conform to the hardware model before high-level synthesis.

In the case of systems to control devices, programs may be composed of interrupt handlers or may use instructions to manipulate special registers, which are often written in assembly languages or using inline assembly. High-level synthesis is not directly applicable to those programs.

Binary synthesis is, in a sense, a variant of high-level synthesis. It takes binary programs instead of programs written in high-level languages. Stitt, et al. developed binary synthesizers for MIPS, ARM, and MicroBlaze, which convert critical parts of binary program into hardware [2]. ACAP is a binary synthesizer for MIPS, which can convert a whole executable binary program into a hardware module that is equivalent to a CPU running the program, or can compile specified sections of a binary program into coprocessors that are tightly coupled with the CPU running the program [5].

By generating hardware that emulates load/store instructions, binary synthesizers can handle accesses to global variables or accesses via pointers without rewriting the source program. Since binary synthesizers take binary programs as inputs, the languages of the source programs do not matter. In [3, 4], programs with interrupt handlers written in combination of C, MIPS assembly, and inline assembly are synthesized into hardware.

The flow of binary synthesis is shown in Fig. 1. Once a CDFG is constructed from a binary program, the subsequent tasks are the same as in high-level synthesis. If a source code of existing high-level synthesizer is available, or one has a good knowledge on open source high-level synthesizers [6], a binary synthesizer may be developed by implementing a translator from binary programs to CDFGs (control dataflow graphs). However, this may involves dataflow analysis, program-level optimization, and construction of variable tables, which still requires substantial effort.

## III. Binary synthesis using high-level synthesis

### A. *Overview*

A new flow of binary synthesis that allows low cost implementation is proposed in this paper. A C program, instead of a CDFG, is generated from a given binary program from which hardware design is generated by a high-level synthesizer.

In this paper a type of binary synthesis is assumed where a combination of a CPU and a binary program is converted to an equivalent hardware modules, as shown in Fig. 2. It is also as-

```
#define SINT32(u) ((int32_t)((u) & 0x7fffffff) + (((u) & 0x80000000) ?  -2147483648 : 0))
#define SINT16(u) ((int16_t)((u) & 0x7fff) + (((u) & 0x8000) ?  -32768 : 0))
#define SINT8(u)  ((int8_t)((u) & 0x7f) + (((u) & 0x80) ?  -128 : 0))
```

Fig. 6. Conversion from unsigned to signed integer

| | |
|---|---|
| `mul rd,rs1,rs2` | `rd = rs1 * rs2;` |
| `mulhu rd,rs1,rs2` | `rd = ((uint64_t)rs1 * (uint64_t)rs2) >> 32;` |
| `mulh rd,rs1,rs2` | `rd = ((int64_t)SINT32(rs1) * (int64_t)SINT32(rs2)) >> 32;` |
| `mulhsu rd,rs1,rs2` | `rd = ((int64_t)SINT32(rs1) * (int64_t)rs2) >> 32;` |

Fig. 7. Translation of multiply instructions

sumed in this paper that data memory is implemented logically inside the hardware module (physical implementation depends on high-level synthesizers).

By mapping general purpose registers of the CPU to C variables, each of the ALU instructions is translated to a C statement in a straightforward manner, with some attention to conversion between signed and unsigned integers and the precision of intermediate results. Load and store instructions are converted to accesses to an array variable with translation of addresses to indexes. Branch and jump instructions are translated to goto statements with some auxiliary statements. Register jump instructions, whose targets are decided only at run time, are converted to switch statements by listing all the possible targets.

### B. *Flow of synthesis*

The proposed flow of binary synthesis is shown in Fig. 3. A given linked executable binary program is disassembled and then translated to a C program of equivalent behavior. It is fed into high-level synthesis to generate an HDL code ready for logic synthesis. Since laborious tasks such as dataflow analysis, scheduling, biding are done by the high-level synthesizer, all we have to do is to develop an assembly to C translator.

Though various styles are possible for the C programs, this paper considers a C program consisting of a single function for the input assembly program, as shown in Fig. 4. The function consists of three parts; (1) declaration of the registers, (2) declaration of the memory, and (3) statements translated from the instructions.

In the following subsections, translation from RISC-V instructions is shown as examples, but note that the method itself is not restricted to the specific ISA.

### C. *ALU instructions and handling of signs*

Each register in the ISA is expressed as a local variable of unsigned integer type with the same number of bits. Fig. 4 (1) is an example of declaration for 32-bit registers. `uint32_t` is defined to be an integer type that the back-end high-level synthesizer deals with as a 32-bit unsigned integer.

Arithmetic, logical, and shift instructions are straightforwardly translated to equivalent C statements. Examples of translation of 32-bit ALU instructions of RISC-V are shown in Fig. 5. For example, an instruction "`add rd,rs1,rs2`" is translated to a statement "`rd = rs1 + rs2;`".

Since the variables for the registers are declared as unsigned, type conversion is necessary if an instruction assumes signed operands. However, in the case of CPUs based on 2's complement expression, there is no need for type conver-

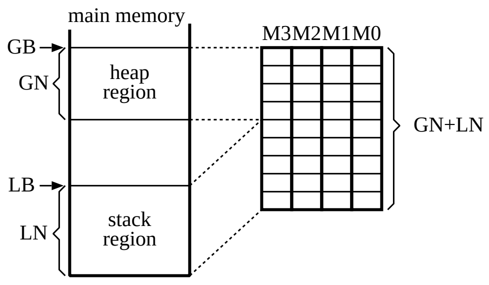


Fig. 8. Implementation of memory space

sion, for the unsigned arithmetic yields the same results as the signed arithmetic. For example, if we simply translate "`addi sp,sp,-32`" to "`sp = sp + (-32);`" assuming the unsigned arithmetic, −32 is interpreted as 4294967264 and the lower 32 bit of `sp + 4294967264` is assigned to `sp`, which yields the same results as in the signed arithmetic.

This does not work, however, for some instructions such as the shift right arithmetic instruction. In this case, strict type conversion is needed. Note that a simple cast operation is not always valid in the light of the C language specification; if we attempt to convert an unsigned integer whose MSB is set (a negative number if seen as a signed integer) to a signed integer by the cast operation, it causes signed overflow and incurs undefined behavior, which means that the program is invalid and compilers may generate any (erroneous) code. Thus, for a safety reason, macros `SINT32`, `SINT16`, and `SINT8` shown in Fig. 6 are defined which performs unsigned to signed conversion without triggering undefined behavior[1].

### D. *Precision*

When higher precision is needed in the computation for instructions, necessary type conversion is inserted. Examples for multiplication of RISC-V is shown in Fig. 7.

- `mul` instruction computes the product of unsigned 32-bit integers `rs1` and `rs2` and writes the lower 32 bits of the result to `rd`. Multiplication of 32-bit unsigned integer type in C only leave the lower 32-bit of the result, no type conversion is needed.
- `mulhu` instruction computes the product of unsigned 32-bit integers `rs1` and `rs2` and writes the upper 32 bits of the result to `rd`. This needs multiplication of the 64-bit precision, so the operands are cast to of 64-bit unsigned integers before multiplication. The upper half of the result

[1]Though they consist of several operations, GCC and LLVM of the latest versions reduce them to null.

| | |
|---|---|
| `lw rd,off(rs)` | `rd = (MEM_3[MA(rs+off)] << 24) + (MEM_2[MA(rs+off)] << 16) + (MEM_1[MA(rs+off)] << 8) + (MEM_0[MA(rs+off)]);` |
| `lh rd,off(rs)` | `rd = (MEM_1[MA(rs+off)] << 8) + (MEM_0[MA(rs+off)]);` |
| `lb rd,off(rs)` | `rd = (MEM_0[MA(rs+off)]);` |
| `sw rs2,off(rs1)` | `MEM_3[MA(rs1+off)] = (rs2 >> 24);` `MEM_2[MA(rs1+off)] = (rs2 >> 16);` `MEM_1[MA(rs1+off)] = (rs2 >> 8);` `MEM_0[MA(rs1+off)] = (rs2 & 255);` |
| `sh rs2,off(rs1)` | `MEM_1[MA(rs1+off)] = (rs2 >> 8);` `MEM_0[MA(rs1+off)] = (rs2 & 255);` |
| `sb rs2,off(rs1)` | `MEM_0[MA(rs1+off)] = (rs2 & 255);` |

Fig. 9. Translation of load/store instructions

```
#define LB 0x7ffbfe
#define GB 0xc0000000
#define MA_L(a) ( (((a) - LB) / 4) + GN )
#define MA_G(a) ( (((a) - GB) / 4) )
#define MA(a) ( ((a) >= LB) ? MA_G(a) : MA_L(a) )
```

Fig. 10. Macros for address translation

is captured by a shift operation.

- `mulh` instruction computes the product of *signed* 32-bit integers `rs1` and `rs2` and writes the upper 32 bits of the result to `rd`. In this case, `rs1` and `rs2` must be converted to 32-bit signed integers and then to 64 bit signed integers.
- `mulhsu` instruction computes the product of *signed* `rs1` and *unsigned* `rs2` and writes the upper 32 bits of the result to `rd`. The both operands must be extended to *signed* 64-bit integers, not unsigned 64-bit integers, before multiplication.

## *E. Load and store instructions*

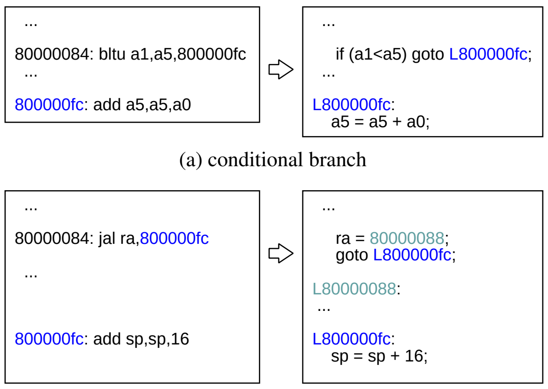
This paper assumes that the main memory is byte addressable but mainly accessed by the word consisting of $w$ bytes. Then the main memory is modeled by $w$ arrays of bytes. The effective addresses of load/store instructions are translated to the corresponding array indexes.

For example, when $w = 4$, and if the heap (global) region of size GN starts at address GB, and the stack region of size LN starts at address LB, the memory space is implemented as 4 arrays of bytes of size GN+LN, as shown in Fig. 8. A given effective address is first tested if it belongs to the heap region or the stack region, and then is converted to the array index. The macros for the address translation is shown in Fig. 10, where `MA(a)` in the bottom line gives the index corresponding to address `a`.

Using the `MA(a)`, load/store instructions are translated to accesses to the memory arrays, as the examples shown in Fig. 9. "`lw`", "`lh`", and "`lb`" are load instructions for 4, 2, and 1-byte data, respectively, while "`sw`", "`sh`", and "`sb`" are store instructions for 4, 2, and 1-byte data, respectively. Though they involve several operations, high-level synthesis will optimize the accesses to the arrays to be done in parallel and the shift and add operations into mere data transfers.

## *F. Branch and jump instructions*

A branch instruction can be converted to a conditional statement and a goto statement.



```
...
80000084: bltu a1,a5,800000fc
...
800000fc: add a5,a5,a0
```

```
...
  if (a1<a5) goto L800000fc;
...
L800000fc:
  a5 = a5 + a0;
```

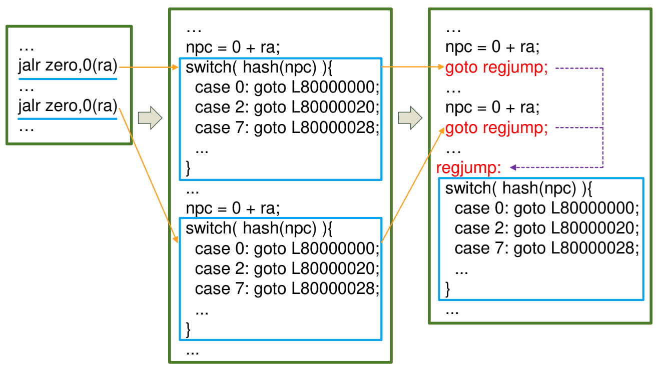
(a) conditional branch

```
...
80000084: jal ra,800000fc
...
800000fc: add sp,sp,16
```

```
...
  ra = 80000088;
  goto L800000fc;
L80000088:
...
L800000fc:
  sp = sp + 16;
```

(b) jump and link

Fig. 11. Translation of branch and jump instructions



```
...
jalr zero,0(ra)
...
jalr zero,0(ra)
...
```

```
...
npc = 0 + ra;
switch( hash(npc) ){
  case 0: goto L80000000;
  case 2: goto L80000020;
  case 7: goto L80000028;
  ...
}
...
npc = 0 + ra;
switch( hash(npc) ){
  case 0: goto L80000000;
  case 2: goto L80000020;
  case 7: goto L80000028;
  ...
}
...
```

```
...
npc = 0 + ra;
goto regjump;
...
npc = 0 + ra;
goto regjump;
...
regjump:
switch( hash(npc) ){
  case 0: goto L80000000;
  case 2: goto L80000020;
  case 7: goto L80000028;
  ...
}
...
```

Fig. 12. Translation of register jump instruction

An example of transformation of a "branch if less than unsigned" (bltu) instruction is shown Fig. 11 (a). In a preprocessing stage, all the head addresses of the functions and the target addresses of the branch instructions are enumerated and the labels are attached to the statements corresponding to the instructions at the addresses.

In our scheme where a whole assembly program is translated to a single function in the C program, call and return instructions in the assembly program can also be converted using goto statements. A jump with link instructions used for a function call is translated to statements to save the return address and to jump to the head of the function. An example of translation of a "jump and link" (jal) instruction is shown in Fig. 11 (b). The address of the next instruction is saved in `ra` as the return address and then jump is made to `L800000d8`.

Register jump instructions needs some elaboration, because the target of a register jump is dependent on the value in the specified register, which is determined only at run-time. This problem is solved by 1) including all the possible return addresses to the list of the branch targets and 2) generate a switch statement to jump to one of the labels determined at run-time. An example of translation of a register jump instruction is shown in Fig. 12. The hash function on the target address is to reduce the number of bits. Note that all the jalr instructions result in the same sentences. Thus, only a single switch statement is generated and all the register jumps are directed to the switch statement. This substantially reduces the resulting circuit size.

TABLE I
SYNTHESIS RESULT

| program | high-level synthesis | | | | | proposed binary synthesis | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | #FF | #LUT | #DSP | #cycle | delay [ns] | #insn | #FF | #LUT | #DSP | #cycle | delay [ns] |
| bin_search | 164 | 405 | 0 | 22 | 6.720 | 38 | 198 | 523 (1.29) | 0 | 21 (0.95) | 5.924 |
| bubble_sort | 112 | 190 | 0 | 90,300 | 6.514 | 37 | 170 | 631 (3.32) | 0 | 90,002 (1.00) | 6.514 |
| lcm | 1,419 | 1,363 | 3 | 200 | 8.470 | 39 | 1,654 | 1,801 (1.32) | 3 | 204 (1.02) | 8.470 |
| prime | 464 | 499 | 0 | 3,099 | 8.250 | 29 | 546 | 534 (1.07) | 0 | 2,491 (0.80) | 6.880 |
| fsm32 | 3,307 | 4,767 | 64 | 265 | 8.470 | 226 | 3,827 | 4,699 (0.99) | 66 | 268 (1.01) | 8.470 |

GCC optimization: –O3, Synthesizer: Xilinx Vivado HLS (2020.1), Target: Xilinx Artix-7

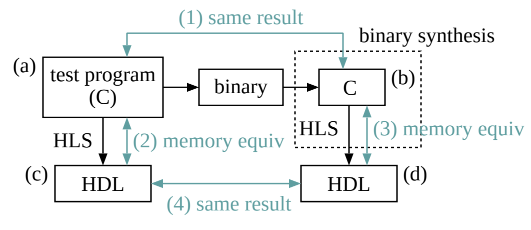


Fig. 13. Flow of test of binary synthesizer

## IV. EXPERIMENTAL RESULTS

Based on the proposed method, a binary synthesizer for RISC-V has been implemented. The assumed ISA is RV32IM. The synthesizer supports 45 instruction out of the 55 of RV32IM, excluding fence, ecall, and control status register related instructions. Vivado HLS (2020.01) is used as a back-end high-level synthesizer. The assembly to C translator is implemented in Python3.

The implemented synthesizer was tested by comparing the execution results of synthesized circuits by the binary synthesizer against those by the high-level synthesizer. The flow of test is shown in Fig. 13. First, test programs written in C are prepared (a). The programs are composed so that their final results will be written to the memory at specific addresses. The test programs are compiled and then translated to equivalent C programs (b). By running the programs, it is tested if (a) and (b) produce the same results. Then, both (a) and (b) are fed into the high-level synthesizer to generate HDLs ((c) and (d)). By co-simulation, it is tested if the C programs and HDLs yield the equivalent memory access sequences ((2) and (3)). After that, by simulation, it is tested if the pairs of HDLs produce the same results (4).

The implemented binary synthesizer were evaluated using 5 RISC-V binary programs, which were generated from 5 C programs by GCC. The GCC was of version 10.2.0 targeting riscv32-unknown-elf and the optimization option was –O3.

The result is summarized in Table I. Among them, "fsm32" is a randomly generated state machine of 32 states that repeats state transition 128 times using remainder operations to determine the next states. The columns "high-level synthesis" show the results of high-level synthesis from the original C programs (these are just for reference and we do not intend to beat the high-level synthesizer). The column "#insn" in the "proposed binary synthesis" section shows the number of the instructions of the binary programs.

The resulting circuit size is 3.32 times larger for bubble_sort but about the same for the other test programs. The execution cycles and the critical path delay are almost the same as high-level synthesis.

TABLE II
IMPACT OF PROGRAM SIZE

| program | #insn | #FF | #LUT | #DSP | #cycle | delay [ns] |
|---|---|---|---|---|---|---|
| fsm16 | 126 | 1,864 | 2,500 | 35 | 261 | 8.470 |
| fsm32 | 226 | 3,827 | 4,699 | 66 | 268 | 8.470 |
| fsm64 | 419 | 7,033 | 9,649 | 194 | 283 | 8.470 |
| fsm128 | 810 | 13,532 | 17,569 | 386 | 257 | 8.597 |
| fsm256 | 1,720 | 24,543 | 29,635 | 514 | 260 | 8.670 |
| fsm512 | 3,738 | 45,871 | 56,755 | 1,026 | 257 | 8.470 |

TABLE III
IMPACT OF COMPILER OPTION

| program | option | #insn | #FF | #LUT | #DSP | #cycle | delay [ns] |
|---|---|---|---|---|---|---|---|
| bin_search | –O0 | 55 | 375 | 1,530 | 0 | 101 | 8.661 |
| | –O1 | 37 | 232 | 673 | 0 | 29 | 7.557 |
| | –O2 | 42 | 166 | 530 | 0 | 20 | 7.113 |
| | –O3 | 38 | 198 | 523 | 0 | 21 | 5.924 |
| bubble_sort | –O0 | 86 | 794 | 1,985 | 0 | 362,107 | 8.660 |
| | –O1 | 43 | 162 | 649 | 0 | 90,002 | 6.514 |
| | –O2 | 37 | 170 | 631 | 0 | 90,002 | 6.514 |
| | –O3 | 37 | 170 | 631 | 0 | 90,002 | 6.514 |
| lcm | –O0 | 74 | 1,590 | 1,911 | 3 | 213 | 8.470 |
| | –O1 | 47 | 1,492 | 1,467 | 3 | 212 | 8.470 |
| | –O2 | 41 | 1,588 | 1,801 | 3 | 189 | 8.470 |
| | –O3 | 39 | 1,654 | 1,801 | 3 | 204 | 8.470 |
| prime | –O0 | 40 | 515 | 920 | 0 | 1,145 | 6.616 |
| | –O1 | 32 | 546 | 553 | 0 | 1,767 | 8.280 |
| | –O2 | 29 | 546 | 534 | 0 | 3,447 | 6.880 |
| | –O3 | 29 | 546 | 534 | 0 | 2,491 | 6.880 |
| fsm32 | –O0 | 704 | 2,671 | 16,511 | 64 | 1,105 | 8.470 |
| | –O1 | 220 | 4,013 | 5,059 | 98 | 266 | 8.470 |
| | –O2 | 226 | 3,827 | 4,699 | 66 | 268 | 8.470 |
| | –O3 | 226 | 3,827 | 4,699 | 66 | 268 | 8.470 |

In order to see an impact of program size on the resulting circuits, binary synthesis is run on different sizes of the state machine programs. The result is shown in Table II. "fsm*n*" is a state machine program of *n* states. For all the programs, the number of transition was fixed to 128. It can be observed that the circuit size grows in proportion to the number of instructions, but that the execution cycles and the critical path delay stay almost the same.

Table III shows the result of an experiment to see relation between the code quality in terms of the GCC's optimization options and the quality of the synthesized circuits. All the options –O1, –O2, and –O3 led to smaller circuits than –O0, which seems mainly due to the number of instructions. The cycle count varies, and it seems that we should better try plural options to get the best synthesis results.

## V. Conclusion

This paper has proposed an easy way to implement binary synthesizers utilizing existing general high-level synthesizers as their back-ends. A binary synthesizer for a new instruction set architecture can be developed by only implementing a binary to C translator. An experimental result shows that synthesized circuits from RISC-V binary programs are 1.00 to 3.32 times larger than those synthesized from original C programs, but the number of execution cycles are almost the same.

A strength of our method is seamless handling of pointers (addresses). For example, data objects can be shared among functions by passing pointers. However, our method handles only a single hardware module, so sharing of global objects among multiple modules is impossible. Moreover, memory mapped I/O is not supported either in the current framework. To meet these requirements, we are now working on translation of assembly codes to a different style of C programs that can access an external memory via address and data ports. It is also an interesting topic to generate C programs that lead to better HLS results.

### *Acknowledgments*

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. 19H04081.

## 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] G. Stitt and F. Vahid: "Binary synthesis," *ACM Trans. Design Automation of Electronic Systems*, vol. 12, no. 3, pp. 1–30 (Aug. 2007).

[3] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. SASIMI 2015*, R1-3, pp. 10–15 (Mar. 2015).

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

[5] 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).

[6] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, J. H. Anderson: "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," in *ACM Trans. on Embedded Computing Systems*, vol. 13, no. 2, pp. 1–27 (Sept. 2013).