Binary Synthesis from RISC-V Executables

Shoki HAMANA   Nagisa ISHIURA
School of Science and Technology, Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, 669–1337, Japan

Abstract—This paper presents an implementation of a binary synthesizer which converts a given executable binary code of RISC-V into hardware functionally equivalent to a RISC-V core executing the code. A CPU core and an instruction memory are replaced by the synthesized hardware, which reduces execution time and hardware size for small scale programs. A given binary code is disassembled and parsed to build a control dataflow graph (CDFG), then traditional high-level synthesis techniques are applied to generate RT level Verilog HDL. For a small example program consisting of 34 through 160 instructions, synthesized hardware on Xilinx FPGA Artix-7 took about 74.5% less cycles than on RISC-V Rocket core, with smaller number of LUTs.

I. Introduction

RISC-V [1] is an open-source instruction set architecture. Publicly available RTL source codes and tool chains including compilers, simulators, debuggers, are making it easy to develop hardware and software for embedded systems.

Though the most popular usage of RISC-V cores may be as host processors for high-performance custom hardware engines, they may also be used as main processors for low cost IoT devices. Since RISC-V is originally not intended for high performance processors, instruction extensions or hardware support become options, when a RISC-V core needs more efficiency in computation speed or in power consumption. If the bottleneck is reduced to some core operations, customized instructions will be effective. Otherwise, migration of the loads from software to hardware should be considered.

High-level synthesis [2] or binary synthesis [3, 4, 5] is one of the most prevailing techniques for this purpose, where a program code written in high-level languages or an executable binary code are automatically compiled into a register transfer level hardware model.

Although the performance of the hardware generated by binary synthesis is generally lower than those by high-level synthesis, due to lack of high-level information in the input programs, binary synthesis can handle wider range of software codes. The input programs may contain inline assembly or interrupt handlers in hand written assembly.

This paper presents binary synthesizer that translates a RISC-V binary code into logic synthesizable Verilog HDL description. A whole binary code is synthesized into a hardware module functionally equivalent to the RISC-V core executing the code. It aims at replacing the CPU core and the instruction memory by the hardware and reducing the execution cycles and hardware cost for small scale programs.

II. Binary Synthesis

Binary synthesis differs from high-level synthesis in the front-end part; a CDFG (control dataflow graph) is constructed from a binary code instead of a program written in a high-level language. Almost all of the back-end techniques are common with high-level synthesis.

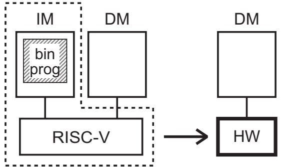


Fig. 1. Binary synthesis of CPU compatible hardware.

The merit of binary synthesis is that it has less restrictions on input behavior specifications. For example, C programs with pointers or complicated control structures can be synthesized into hardware. Binary synthesis can handle programs in hand-written assembly programs or those containing inline assembly codes intended to communicate with hardware. it can even deal with programs containing interrupt handlers [6].

Mittal et al. [3] developed a binary synthesizer to translate DSP binaries into FPGA hardware. It accepted programs in C/C++, Matlab, and Simulink as well as hand-written assembly. The binary synthesizer developed by Stitt et al. [4] synthesized selected sections of binary codes of MIPS, ARM, and MicroBlaze into coprocessors, or hardware accelerators. ACAP [5] converted MIPS binary codes into hardware. It allows either generation of coprocessors from code sections or compilation of a whole binary code into a hardware module that replaces the CPU and the instruction memory.

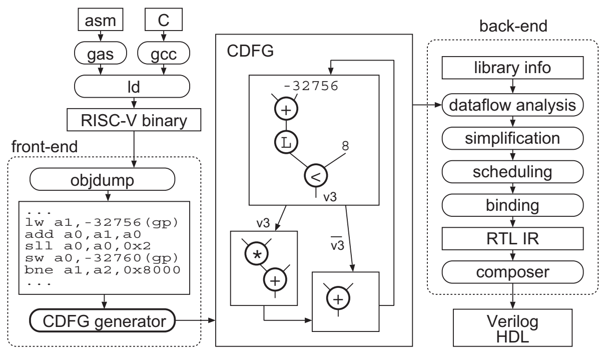
However, to the best of our knowledge, there is no binary synthesizer that takes RISC-V binaries.

III. Binary Synthesis from RISC-V Executables

The input to our binary synthesizer is a binary executable code for RISC-V. As for the instruction set, we assume RV32IM, base integer instruction set with standard extension for integer multiplication and division. Then, as shown in Fig. 1, a hardware module which is compatible with the CPU executing the binary program is synthesized. The elimination of the instruction memory might be useful in protecting the software from analysis.

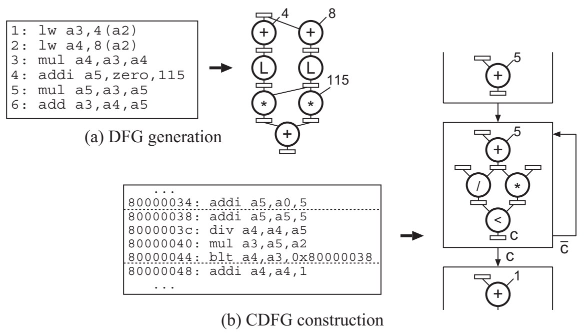
Fig. 2 shows the flow of synthesis in our method. Input programs may be written in C language or in assembly language. Inline assembly is also allowed. They are compiled (by gcc) or assembled (by gas) and then linked to a executable binary code (by ld). The front-end of our binary synthesizer scans the disassembled code to generate a CDFG (control dataflow graph), a popular data structure for high-level synthesis. The back-end of the synthesizer takes the CDFG to perform a standard sequence of high-level synthesis tasks to generate a hardware model in Verilog HDL.

Fig. 3 (a) shows how DFG is generated from a RISC-V in-



```
...
lw a1,-32756(gp)
add a0,a1,a0
sll a0,a0,0x2
sw a0,-32760(gp)
bne a1,a2,0x8000
...
```

Fig. 2. Flow of binary synthesis.



```
1: lw a3,4(a2)
2: lw a4,8(a2)
3: mul a4,a3,a4
4: addi a5,zero,115
5: mul a5,a3,a5
6: add a3,a4,a5
```

(a) DFG generation

```
   ...
80000034: addi a5,a0,5
80000038: addi a5,a5,5
8000003c: div a4,a4,a5
80000040: mul a3,a5,a2
80000044: blt a4,a3,0x80000038
80000048: addi a4,a4,1
   ...
```

(b) CDFG construction

Fig. 3. CDFG generation.

struction sequence. A load instruction (such as `lw` in line 1) is converted into an add operation to calculate its effective address and a load operation. An arithmetic instruction such as `mul` (multiplication) in line 3 is translated into a corresponding operation node. When the value in a register is updated, a new value node is generated. The value node is later assigned to a hardware register in the binding phase. An operation to load a constant into a register (such as `addi` in line 4) reduced to a constant value node.

Fig. 3 (b) illustrates how CDFG is constructed. An instruction sequence is dissected at branch instructions and branch targets to yield a set of basic blocks. Since there is no delayed branch in RISC-V, this dissection is straightforward. Each basic block becomes a DFG (dataflow graph). A conditional branch instruction is translated into an operation to compute the branch condition and transition edges to the next DFGs.

One hard issue in binary synthesis is how to handle register jump instructions (such as `jr`) whose target addresses are determined only at run time. We adopt the same strategy as ACAP [5]. A module named "PC2state" is generated which translates the instruction addresses to the corresponding state encodings of the synthesized hardware. A register jump instruction, which writes its target address into PC (program counter), is converted into an operation to update the state register by the output of the PC2state module. The PC2state module is generated by gathering the head states of all the DFGs.

IV. Preliminary Experiment

A binary synthesizer "ACAP-R" has been implemented based on the method described in the previous section. The front-end for RISC-V has been newly developed and the back-end of ACAP is used as it is.

Table I summarizes the result of an experiment where small C programs were synthesized targeting an FPGA. "#insn"

TABLE I
Experimental result

program	#insn	RISC-V #cycle	RISC-V #LUT	RISC-V delay [ns]	ACAP-R A,M	ACAP-R #cycle	ACAP-R #LUT	ACAP-R delay [ns]
fibonacci	34	1,483			3,0	274	1,533	7.2
arith-test	70	642	3,202	14.7	2,1	273	2,665	11.6
FSM	160	777			2,2	164	2,362	13.3

Synthesizer: Xilinx Vivado (2017.4)
Target: Xilinx Artix-7 (XC7Z020-1CLG400I)

indicates the instruction counts in the programs. "#cycle", "#LUT", "delay" show the execution cycle counts, the LUT counts, and the critical path delays, respectively. The target FPGA was Xilinx Artix-7 (XC7Z020-1CLG400I). "A,M" indicates the numbers of ALUs and multipliers allocated for synthesis. The RISC-V core used in the experiment is generated by the Rocket Chip Generator[1] with the DefaultRV32Config mode. The hardware generated by ACAP-R took 74.5% less cycles on average. The LUT counts of the hardware are supposed to grow in proportional to the instruction counts, but were smaller than the core when the programs were of less than 160 instructions.

V. Conclusion

A binary synthesizer ACAP-R which generates hardware from a RISC-V binary code has been presented.

The original version of ACAP has another synthesis mode that compiles specified code segments into coprocessors which is tightly coupled with the CPU core. Our future work includes extending ACAP-R to operate in this mode.

Acknowledgments—Authors would like to express their appreciation to Dr. H. Kanbara (ASTEM/RI), Prof. H. Tomiyama (Ritsumeikan Univ.), and Mr. T. Nakatani (formerly with Ritsumeikan Univ.) for their valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin Univ. for their cooperation.

References

[1] D. Patterson and A. Waterman: *The RISC-V Reader: An Open Architecture Atlas*, Strawberry Canyon (Nov. 2017).

[2] 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).

[3] G. Mittal, D. C. Zaretsky, X. Tang, and P. Banerjee: "Automatic translation of software binaries onto FPGAs," in *Proc. DAC 2004*, pp. 389–394 (June 2004).

[4] G. Stitt and F. Vahid: "Binary synthesis," *ACM TODAES*, vol. 12, no. 3, article 34 (Aug. 2007).

[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] 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).

[1] `https://github.com/freechipsproject/rocket-chip` (accessed 2019-06-10).