Efficient Multiple-ISA Embedded Processor Core Design Based on RISC-V

Yuanhu Cheng, Libo Huang*, Yijun Cui, Sheng Ma, Yongwen Wang, Bincai Sui
{chengyuanhu,libohuang,cuiyijun18,masheng,wyw,bingcaisui}@nudt.edu.cn
National University of Defense Technology
Changsha, China

ABSTRACT

RISC-V ISA is developing rapidly, and its target field highly overlaps with the ARM ISA. As a later ISA, RISC-V needs to solve the problem of software compatibility. In the embedded field, by a multiple-ISA processor based on binary interpretation, RISC-V supports for ARM Thumb can be implemented efficiently. However, binary interpretation may result in lower performance of running non-native ISA programs than native ISA. As a result, to improve the performance of running the ARM Thumb programs, we propose some optimization methods of hardware support to reduce the number of RISC-V instructions required to interpret ARM Thumb instructions. Based on the open-source Zero-riscy core, we implement a demo to support ARMv6-M based on RISC-V. The results show that running the ARMv6-M code of Dhrystone and CoreMark benchmarks compiled by ARMCC, Zero-riscy can achieve 85.0% and 73.1% of the performance of running RISC-V code compiled by GCC with less than 13.5% increase of FPGA resources.

KEYWORDS

RISC-V, ARM Thumb, Embedded, Multiple-ISA, Binary translation

1 INTRODUCTION

In recent years, the RISC-V instruction set architecture (ISA) has developed rapidly, and its target field highly overlaps with the ARM ISA. Compared to the traditional ISA, RISC-V has two obvious advantages [21]: 1) open-source, users can apply the RISC-V ISA for free to design processors, and even do secondary development based on open-source code from an open-source community; 2) the basic ISA is very simple and extensible. RISC-V has a series of standard extensions, so users can flexibly choose extended ISA to implement according to the scenario. These advantages of RISC-V can greatly shorten development costs and the period of the processor. However, a large number of existing programs are developed for traditional ISA (e.g. ARM and X86) and cannot run directly on RISC-V processors. As a later ISA, RISC-V needs to solve the problem of software compatibility.

In desktop and server systems, software binary translation [19] is the most common method to overcome software compatibility. This method only requires to install a binary translation system, and then the program based on other ISAs can be directly run through this system. The advantage of software binary translation systems is its convenience and flexibility. Unfortunately, it has limitations in the embedded system. First of all, due to the requirements of the chip area and power of the embedded system, there is often not enough storage space and environment to run a binary translation system. Second, most binary translation systems are still designed to complete functional simulation, but the performance has not received much attention.

Another method to solve the software compatibility problem is the multiple-ISA processor that adds dedicated hardware to the traditional processor to support more than one ISAs. Although the multiple-ISA processor also increases the area and power of the chip, it is a more appropriate method that solving the problem of software compatibility in the embedded field if it can be implemented with very low hardware resource consumption. However, at present, the most critical issue in using the hardware approach is the additional hardware overhead associated with supporting multiple ISAs. For example, in [7], a dynamic two-level binary translation system is implemented. The first level translates from non-native ISA into native ISA and the second level is used to optimize the generated code, in which the first level increases the hardware overhead by 29% for supporting ARM, while the second level consumes more than the first.

In this paper, we try to support RISC-V and ARM Thumb using multiple-ISA processors based on binary interpretation, because the method based on binary interpretation is the simplest so that it can achieve a multiple-ISA processor with lowest hardware resources. Firstly, we discuss the conversion of registers and instructions between ARM Thumb and RISC-V RV32I, as well as the optimization methods that can be taken to improve the performance of running ARM Thumb programs using RISC-V processors. Then, we develop a simple demo of the processor supporting the ARMv6-M (a subset of ARM Thumb) and RISC-V ISAs. The demo is implemented by adding a binary interpreter that converts ARMv6-M instructions into RISC-V instructions in the fetch stage based on PULPino's open-source RISC-V core, called Zero-riscy (Ibex) [1, 2]. We employe Dhrystone and CoreMark to evaluate the performance of Zero-riscy running ARMv6-M programs through simulation, and count the FPGA resources consumed after synthesis.

Finally, on optimized Zero-riscy, for the Dhrystone code compiled by ARMCC compiler, each ARM Thumb instruction is translated into 1.62 RISC-V instructions with a performance of 0.69 DMIPS/MHz, 85.0% of the performance of running RISC-V compiled by GNU GCC compiler. And for the CoreMark code, each ARM Thumb instruction is translated into 1.11 RISC-V instructions with a performance of 1.22 CoreMark/MHz, 73.1% of RISC-V. Compared to without any optimization, the performance is improved by 2.01 and 2.80 times respectively while the consumption of hardware resources is decreased slightly.

*Corresponding author

2 RELATED WORK

Binary translation is the most common way to overcome software compatibility. According to different implementations, binary translation can be divided into three types: binary interpretation (or emulator), dynamic translation, and static translation [3]. QEMU [5] is a typical software binary translation system, which uses dynamic binary translation to translate an ISA into the corresponding target ISA and supports to simulate the most popular ISAs. Similarly, DAISY [10] and FX!32 [8, 13] are binary translation systems that support specialized ISA. For embedded systems, LLBT [18] is a static binary translation system based on the LLVM compiler, and [4, 15–17] study the optimization methods of dynamic binary translation for embedded systems.

There are some works that implement binary translation system with combination of software and hardware. In these works, the major function of the hardware is that accelerating the binary translation process and optimizing the code to improve performance. GODSON-3 Processor [14] uses software to translate X86 instructions into MIPS instructions and inserts some units into hardware to support X86 instructions better. Furthermore, it adds some instructions to MIPS ISA so that one X86 instruction can be converted into fewer MIPS instructions. Crusoe processor [9] is a microprocessor that uses Code Morphing Software (CMS) to convert X86 instructions into VLIW instructions. CMS is a dynamic binary translation program and stored in the Read-Only Memory (ROM) of the motherboard and can be considered as part of the hardware. By modifying CMS, the processor can support any ISA. And hardware is also used to accelerate the speed of binary translation and optimize the code. And [22] improves the performance of MIPS microprocessors running X86 programs by hardware based on software dynamic binary translation.

Although there are some existing works that support multiple ISAs by hardware, the method of hardware still requires more research. [7, 11] implement a two-level binary translation system outside a MIPS core. The first level translates from non-native ISA into intermediate-level code (MIPS), and the second level is used to optimize the code generated at the first level. It evaluates the performance of running ARM, PowerPC, and X86 code by the system. [6] applies the 64-bit very long instruction word (VLIW) instruction, in which the upper n bits are used to identify the ISAs. If an instruction belongs to non-native ISA, the corresponding Dynamic Decode Units (DDU) are selected to convert it into native ISA instruction. The size of n determines how many non-native ISAs it can support, and each non-native ISA corresponds to a DDU unit. [12] allows the Intel X86 and PowerPC ISAs to run on a processor. Unlike binary translation, it implements two decode units for two ISAs and the processor's mode decides which one is used. [20] uses a multi-core processor to support multiple ISAs, in which each core supports one ISA. The system software is used to select the ISA and power on the corresponding core.

3 SUPPORT ARM THUMB WITH RISC-V

The multiple-ISA processor based on binary interpretation needs to implement the mapping of registers and instructions between two ISAs. And the most critical problem with binary interpretation is that the performance of running a non-native ISA program may be much lower than running a native ISA program, because many native instructions may be required to complete a non-native instruction. In this section, we will focus on how to interpret ARM Thumb instructions into RISC-V instructions, as well as optimization methods for some ARM Thumb instructions to improve performance.

3.1 Register Mapping

The ARM Thumb includes thirteen general-purpose registers (R0 ~ R12), one Stack Pointer (SP, R13), one Link Register (LR, R14) and one Program Counter (PC, R15), all of them are 32-bits. The RISC-V includes 32 general-purpose registers, and the PC is a special register. Because RISC-V has more registers than ARM Thumb, register mapping can be easily achieved. However, the R0 of RISC-V is always equal to 0 and cannot be modified, we cannot directly map the R0 of ARM Thumb to RISC-V.

Therefore, we can map the registers R0 ~ R12 of the ARM Thumb to R16 ~ R28 of RISC-V, SP to R29, LR to R30, and PC to R31. Since most ARM Thumb instructions take 3 bits to represent a register (R0 ~ R7), we can add a two-bits prefix '10' in front of the 3-bits ARM Thumb register number to map it to the RISC-V register (if the register number in the ARM Thumb instruction is represented by 4 bits, just add a one-bit prefix '1' in front of it). All register mapping relationships are shown in Table 1.

It is worth mentioning that many instructions can modify the value of PC in ARM Thumb ISA, while RISC-V cannot. For a RISC-V processor that supports ARM Thumb, it can still use the PC register defined in RISC-V to complete the instruction fetch. If an ARM Thumb instruction needs to modify the PC value, the RISC-V AUIPC instruction is used firstly to load the PC value into register R31. After performing related operations, the JALR instruction is used to change the PC value and jump to target.

Because RISC-V and ARM Thumb are both Reduced Instruction Set Computing (RISC) ISA, the instruction mapping between ARM Thumb and RISC-V is relatively simple. However, some subtle differences between them still make some ARM Thumb instructions require many RISC-V instructions to achieve, which will greatly reduce the performance of ARM Thumb programs.

3.2 Condition Flags

In ARM Thumb, most instructions need to change or use flag bits that composed of Negative flag (N), Zero flag (Z), Carry flag (C), and Overflow flag (V). There is a hardware logic and a special flags register used to judge and save the flags produced by the most recent instruction that needs to change the flags. In contrast, RISC-V deals with the flags by software, which means that there is no hardware logic and flags register inside the RISC-V processor. If a flag bit is needed (for example, the carry flag may be required for large integer addition), it will be produced by the software program, which is usually done by the compiler.

In fact, if the software method is used to complete the flags judgment, for an ADDS instruction of ARM Thumb (addition instruction, this instruction will change all flags of N, Z, C, and V), 7 additional RISC-V instructions will be needed to judge and save the flags [21]. A typical flags judgment instruction sequence is shown in Instruction List 1, in which N, Z, C, and V flags are stored in R1 ~

Table 1: Register mapping from ARM Thumb to RISC-V

ARM Thumb Register	RISC-V Register	Added Prefix
R0 ~ R7 (000 ~ 111)	R16 ~ R23 (10000 ~ 10111)	10
R8 ~ R12 (1000 ~ 1100)	R24 ~ R28 (11000 ~ 11100)	1
SP (1101)	R29 (11101)	1
LR (1110)	R30 (11110)	1
PC (1111)	PC/R31 (11111)	1

R4 respectively. So, how to process the flags will greatly affect the final performance of running ARM Thumb programs.

Instruction List 1: ARM Thumb Flags judgment by RISC-V instructions.

```
;Judge and save N Flag
SLTI  R1 , Rd , 0
;Judge and save Z Flag
SLTU  R2 , R0 , Rd
XORI  R2 , R2 , 1
;Judge and save C flag
SLTU  R3 , Rd , Rn
;Judge and save V flag
SLTI  R5 , Rn , 0
SLT  R6 , Rd , Rm
XOR  R4 , R5 , R6
```

If we can use hardware to support flags in the RISC-V processor, the number of RISC-V instructions required by each ARM Thumb instruction will be reduced greatly. To implement the hardware flags, some modifications to the RISC-V processor are required: 1) change the implementation of the Arithmetic and Logic Unit (ALU) so that it can generate the flags and the flags can be operated with operands; 2) add a flags register to save the flags, and 3) add a control signal to indicate whether an instruction changes the flags. In fact, all these modifications are very simple and do not incur significant hardware overhead.

Table 2 is the conversion of an ARM Thumb ADDS instruction to the RISC-V instructions with and without hardware flags. Since the source operand registers (Rm and Rn) and the destination operand register (Rd) may be the same and the original operands are required when determining the flags, the result cannot be directly written to the destination register, but to a temporary register (R15). So, additional instruction is used to move the result from the temporary register to the destination register. Finally, an ADDS requires 9 RISC-V instructions without hardware flags, while only one RISC-V instruction is required with hardware flags.

3.3 Branch Instructions

Besides, different flags implementations lead to different ways to implement branch instructions. The implementation of branch instructions refers to the way of determining whether a branch takes or not. For an ISA like RISC-V without hardware flags, the relationship between the two source operands determines the behavior of the branch. But for an ISA like ARM Thumb with hardware flags, the two source operands first take an operation, which will impact the flags. And the result of the branch is based on the flags generated by the operation.

If just RISC-V branch instructions are used to support the ARM Thumb branch, in the worst case, an ARM Thumb branch instruction will need 9 RISC-V instructions to complete (assuming that the flags are stored in a separate register). Instruction List 2 is an instruction sequence for implementing ARM Thumb Signed Greater Than (its condition is that the N flag is the same as the V flag, and the Z flag is equal to 0). Branch instructions are so frequent in a program that the performance will be impacted significantly if each branch instruction is completed using many instructions.

To use one RISC-V instruction to complete an ARM Thumb branch instruction without adding additional custom RISC-V instructions, the role of RS1 field of RISC-V BEQ (compare the values of two registers, the branch takes if they are equal) instruction is modified to represent the condition code (named "cond") of the ARM Thumb when running an ARM Thumb program. In the execution stage, the condition code and flags are used to determine whether a branch is taken. As a result, ARM Thumb branch instructions will be interpreted as RISC-V BEQ instruction by the binary interpreter. To support this optimization, the RISC-V processor needs to be modified: 1) add a hardware logic to judge the flags according to the cond in the execution stage; 2) pass the RS1 field of the RISC-V instruction to the execution stage.

Instruction List 2: RISC-V instruction sequence for implementing ARM Thumb Signed Greater Than.

```
;Load flags to general register
CSRRCI  R1 , 0x20c , 0
;Get V flag
ANDI  R2 , R1 , 0001b
;Get N flag
SRLI  R3 , R1 , 3
;N==V ? R5=0 : R5=1
SUB  R5 , R2 , R3
SLTU  R5 , R0 , R5
;Get Z flag
ANDI  R4 , R1 , 0100b
SRLI  R4 , R4 , 2
;(N==V and Z==0) ? R5=0 : R5!=0
ADD  R5 , R5 , R4
;If R5==0 branch take
BEQ  R5 , R0 , imm
```

Table 2: ARM Thumb ADDS instruction implementation using RISC-V with and without hardware flags

ARM Thumb instruction	RISC-V instruction	
	Without hardware flags	With hardware flags

ADDS Rd,Rn,Rm	ADD R15,Rn,Rm
	SLTI R1,R15,0
	SLTU R2,R0,R15
	XORI R2,R2,1
	SLTU R3,R15,Rn
	SLTI R5,Rn,0
	SLT R6,R15,Rm
	XOR R4,R5,R6
	ADDI Rd,R15,0
		ADD Rd,Rn,Rm

3.4 Conditional Execution

ARM Thumb supports conditional execution, while RISC-V does not. Unlike the ARM 32-bits ISA, there is a special conditional execution instruction in the ARM Thumb named IT (If Then). In the ARM Thumb code, there is an IT block after each IT instruction. Whether the instruction in the IT block is executed depends on whether the flags meets the condition set by the IT instruction, if it does not meet, then this instruction will not be executed but will be regarded as a NOP instruction.

In ARM Thumb, an 8-bits register named EPSR.IT is used to support conditional execution. Among them, EPSR.IT[7:5] is used to save the upper 3 bits of cond, and EPSR.IT[4:0] indicates the number of instructions in the IT block, up to 4 instructions, and the least significant bit of cond. More detailed content can refer to the user manual of ARM Thumb.

Based on the fact that the flags have been implemented by hardware, the conventional method of implementation conditional execution in the RISC-V processor is to judge the condition in the execution stage, and only the instructions that meet the condition will be executed. However, it will result in a problem, that is if an ARM Thumb instruction in the IT block requires many RISC-V instructions to implement and the execution condition of the instruction is not meet, these RISC-V instructions will not be executed but need to be issued to the execution unit, which will cause a large number of pipeline cycles to be wasted.

For overcoming the performance barrier, the judgment logic of the execution condition can be put into the binary interpreter. Like ARM Thumb, an EPSR.IT register is added in the binary interpreter to save the information of IT instruction. Before interpreting an ARM Thumb instruction into RISC-V instructions, the binary interpreter will judge whether the execution condition is met, and only the instruction that meets the condition will be interpreted. In this way, an instruction that does not be executed will only cause one-cycle idle. After an ARM instruction is interpreted, EPSR.IT advanced by shifting EPSR.IT[4:0] left by 1 bit. When EPSR.IT[4:0]=5'b00000, it means that the current instruction is not in an IT block, the instruction is always executed.

4 EXPERIMENT RESULTS

4.1 Supporting ARMv6-M

ARMv6-M is the ISA used in the Cortex-M0 processor, which is a subset of ARM Thumb but does not support conditional execution. We develop a demo that supports both ARMv6-M and RISC ISAs based on the open-source core of PULPino, called Zero-riscy (Ibex) [1, 2]. Zero-riscy is a 32-bit sequential core with two stages pipeline and supports I, M, C, and E standard extensions of RISC-V. We insert a binary interpreter and an instruction buffer in the fetch stage of Zero-riscy and modify the implementation of its ALU to support the flags of ARM Thumb. Figure 1 is the block diagram of the multiple-ISA processor core. The binary interpreter converts an ARMv6-M instruction into one or more RISC-V instructions, and the instruction buffer is used to temporarily save the instructions from the binary interpreter. The size of the instruction buffer is determined by the maximum number of RISC-V instructions required by an ARM Thumb instruction.

4.2 Benchmarks

In the embedded field, Dhrystone and CoreMark are the most common benchmarks, so we employe Dhrystone and CoreMark benchmarks to evaluate the performance of Zero-riscy running ARMv6-M programs. We use the ARMCC compiler to generate the ARMv6-M code and GNU GCC compiler to generate the RISC-V code. Table 3 shows the number of instructions executed by the two benchmarks within their main loop for different ISAs and optimizations.

For Dhrystone with 100 loops, 32711 RISC-V instructions are executed in the loop of the RISC-V program compiled by GNU GCC, while 28105 ARMv6-M instructions are executed in the program compiled by ARMCC. If the ARMv6-M instruction is directly interpreted as RISC-V instruction, 128608 RISC-V instructions are required, and on average, an ARMv6-M instruction requires 4.58 RISC-V instructions. But, after optimizing the flags, 28105 ARMv6-M instructions require 52005 RISC-V instructions, and on average each ARMv6-M instruction requires 1.85 RISC-V instructions. Finally, one ARM instruction requires 1.62 RISC-V instructions after optimizing the flags and branch instructions.

Similar to the Dhrystone benchmark, for CoreMark with 1 loop, the number of RISC-V instructions required to complete 442604 ARMv6-M instructions is reduced from 1963327 to 492766, and the

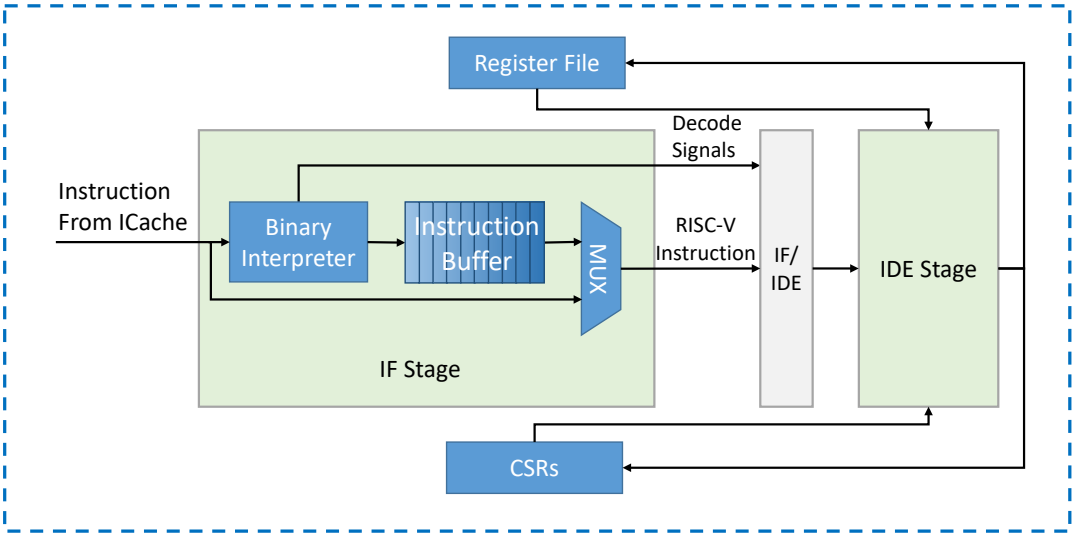

Figure 1: The block diagram of multiple-ISA processor core based on Zero-riscy, which supports RISC-V and ARMv6-M

Table 3: The number of instructions executed in the main loop of two benchmarks under different ISAs and optimizations.

ISA or Implementation	Benchmarks			
	Dhrystone (100 loops)		CoreMark (1 loop)	
	Instruction Counts	Conversion Ratio	Instruction Counts	Conversion Ratio
RISC-V Compiled by GCC	32711	\	306242	\
ARMv6-M Compiled by ARMCC	28105	\	442604	\
Only Binary Interpretation	128608	4.58	1963327	4.44
Optimize Flags	52005	1.85	654065	1.48
Optimize Flags and Branch	45605	1.62	492766	1.11

average number of RISC-V instructions required by each ARMv6-M instruction is reduced from 4.44 to 1.11.

4.3 Performance

Based on the binary file compiled above, our experiment results show that the performance of Zero-riscy running Dhrystone and CoreMark compiled for RISC-V is 0.82 DMIPS/MHz and 1.67 CoreMark/ MHz, respectively. And the performance of running the Dhrystone benchmark compiled by ARMCC is 0.69 DMIPS/MHz, and CoreMark is 1.22 CoreMark/MHz. Figure 2 shows the detailed performance of running Dhrystone and CoreMark on Zero-riscy with different optimizations.

In Figure 2, for the Dhrystone benchmark, the performance of the multiple-ISA processor running ARMv6-M code without optimization is only 42% of the performance of running RISC-V code, while after optimizing flags and branches, it can reach 85%. This means that the performance of running the ARMv6-M Dhrystone program increases by 2.01 times. The performance improvement for running CoreMark is even greater. The performance of running ARMv6-M CoreMark is increased from 26% of RISC-V performance to 73%, an increase of 2.80 times.

The result also shows that the optimized performance of CoreMark drops more than Dhrystone. The reason is that, as mentioned

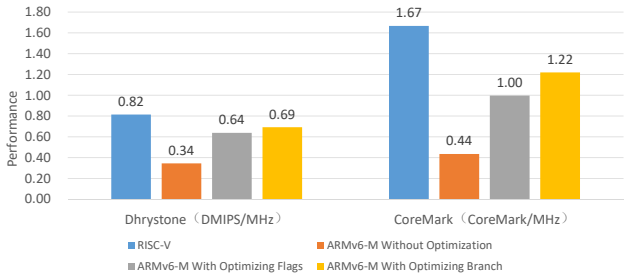

Figure 2: The performance of running Dhrystone and CoreMark on Zero-riscy.

in the previous section, the number of ARMv6-M instructions in CoreMark is significantly more than the RISC-V instructions, and Dhrystone is the opposite. On the other hand, due to the differences between DhryStone and CoreMark, the proportion of ARMv6-M instructions interpreted as RISC-V instructions is different. The conversion ratio of DhryStone is 1.62, while CoreMark is smaller, about 1.11, which causes the performance of DhryStone is still down by 15%, although the number of ARMv6-M instructions is less than RISC-V.

4.4 Hardware Resources

We use the Vivado 2018.3 tool to calculate the LUT and FF resources consumption of Zero-riscy after synthesis at the same frequency on the FPGA, and the result is shown in Figure 3. Without supporting ARMv6-M, the number of LUTs and FFs consumed by Zero-riscy is 3354 and 2006. If the multiple-ISA processor is implemented without any optimization, a binary interpreter and an instruction buffer will be added in the Zero-riscy, the number of LUTs and FFs will increase to 3822 and 2039. After optimizing flags by hardware logic, the number of LUTs drops to 3766, and FFs increases to 2046. In other words, the optimization of flags logic not only improves performance but also reduces hardware overhead. The reason is that more RISC-V instructions are required to complete an ARMv6-M instruction for the implementation without optimization, which will lead to the hardware logic of binary interpreter more complicated and require a larger buffer to store the instruction that translation from ARMv6-M. Relatively speaking, the hardware flags logic is much simpler. Of course, in order to optimize branch instructions, the consumption of hardware resources increased slightly: the number of LUTs increases by 42 while FFs decreases by 3.

Overall, there is a slight reduction in hardware resources compared to the implementation without any optimization. The number of LUTs and FFs are increased by 454 and 37 for supporting the ARMv6-M on Zero-riscy, less than 13.5% of Zero-riscy hardware resources.

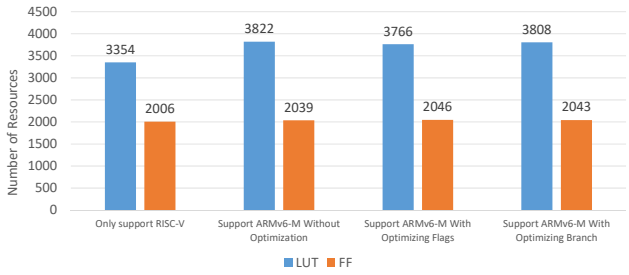

Figure 3: Hardware resources for Zero-riscy on FPGA.

5 CONCLUSION

In this paper, we study the possibility of running the ARM Thumb program on the RISC-V processor based on the multiple-ISA processor and also propose some optimization methods of hardware logic for flags, branch instruction, and conditional execution of ARM Thumb to improve the performance of running ARM Thumb programs. And a demo that supports both RISC-V and ARMv6-M ISAs is implemented based on the open-source RISC-V processor core named Zero-riscy, which proves the feasibility of these methods. In this example, flags and branch instructions are optimized to reduce the number of RISC-V instructions required to complete an ARMv6-M instruction. Finally, on Zero-riscy, for the Dhrystone code compiled by ARMCC compiler, each ARM Thumb instruction is translated into 1.62 RISC-V instructions with a performance of 0.69 DMIPS/MHz, 85.0% of the performance of running RISC-V compiled by GNU GCC compiler, And for the CoreMark code, each ARM Thumb instruction is translated into 1.11 RISC-V instructions with a performance of 1.22 CoreMark/MHz, 73.1% of RISC-V. Compared to without any optimization, the performance is improved by 2.01 and 2.80 times respectively while the consumption of hardware resources is decreased slightly.

6 ACKNOWLEDGEMENT

This work was supported by HGJ of China (under Grant 2017ZX01028-103-002), the NSF of China (under Grants 61872374 and 61672526).

REFERENCES

[1] 2019. https://pulp-platform.org.
[2] 2019. https://github.com/lowRISC/ibex.
[3] E. R. Altman, D. Kaeli, and Y. Sheffer. 2000. Welcome to the opportunities of binary translation. *Computer* 33, 3 (2000), 40–45.
[4] José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. 2013. Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory. *ACM Trans. Embed. Comput. Syst.* 11, 4, Article Article 89 (Jan. 2013), 33 pages. https://doi.org/10.1145/2362336.2399178
[5] F. BELLARD. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference, 2005.*
[6] BORRILL and PAUL. 2001. Method and apparatus for multiplatform instruction set architecture. EP1074910.
[7] Fernanda M. Capella, Marcelo Brandalero, Jair Fajardo Junior, Antonio C. S. Beck, and Luigi Carro. 2013. A Multiple-ISA Reconfigurable Architecture. In *2013 III Brazilian Symposium on Computing Systems Engineering (SBESC).*
[8] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates. 1998. FX!32 a profile-directed binary translator. *IEEE Micro* 18, 2 (1998), 0–64.
[9] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Er Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing™ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *International Symposium on Code Generation & Optimization.*
[10] Kemal Ebcioglu and Erik R. Altman. 1998. DAISY: Dynamic Compilation for 100% Architectural Compatibility. *Acm Sigarch Computer Architecture News* 25, 2 (1998).
[11] Jair Fajardo, Mateus B. Rutzig, Luigi Carro, and Antonio C.S. Beck. 2012. Towards a multiple-ISA embedded system. *Journal of Systems Architecture the Euromicro Journal* 59, 2 (2012), 103–119.
[12] JOHN W. GOETZ, STEPHEN W. MAHIN, and JOHN J. BERGKVIST. 1996. Processor capable of supporting two distinct instruction set architectures. EP0747808.
[13] Raymond J. Hookway and Mark A. Herdeg. 1997. *DIGITAL FX!32: combining emulation and binary translation.*
[14] Weiwu Hu, Wang Jian, Gao Xiang, Yunji Chen, and Guojie Li. 2009. Godson-3: A Scalable Multicore RISC Processor with x86 Emulation. *IEEE Micro* 29, 2 (2009), 17–29.
[15] Goh Kondoh and Hideaki Komatsu. 2010. Dynamic Binary Translation Specialized for Embedded Systems. *Sigplan Notices - SIGPLAN* 45, 157–166. https://doi.org/10.1145/1837854.1736019
[16] D. Richie and J. Ross. 2014. Cycle-accurate 8080 emulation using an ARM11 processor with dynamic binary translation. In *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE).* 186–189.
[17] F. Salgado, T. Gomes, J. Cabral, J. Monteiro, and A. Tavares. 2019. DBTOR: A Dynamic Binary Translation Architecture for Modern Embedded Systems. In *2019 IEEE International Conference on Industrial Technology (ICIT).* 1755–1760.
[18] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: an LLVM-based static binary translator. 51–60. https://doi.org/10.1145/2380403.2380419
[19] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993. Binary Translation. *Communications of the Acm* 36, 2 (1993), 69–81.
[20] RYMARCZYK JAMES WALTER, IGNATOWSKI MICHAEL, and HELLER THOMAS J. 2008. MULTIPLE-CORE PROCESSOR SUPPORTING MULTIPLE INSTRUCTION SET ARCHITECTURES. US2008059769.
[21] Andrew Waterman and Krste Asanović. 2017. *The RISC-V Instruction Set Manual Volume I: User-Level ISA* (2.2 ed.).
[22] Y. Yao, Z. Lu, Q. Shi, and W. Chen. 2013. FPGA based hardware-software co-designed dynamic binary translation system. In *2013 23rd International Conference on Field programmable Logic and Applications.* 1–4.