Front. Comput. Sci., 2025, 19(1): 191103
https://doi.org/10.1007/s11704-023-3239-x

RESEARCH ARTICLE

RVAM16: a low-cost multiple-ISA processor based on RISC-V and ARM Thumb

Libo HUANG, Jing ZHANG, Ling YANG, Sheng MA, Yongwen WANG, Yuanhu CHENG (✉)
College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China

© Higher Education Press 2025

Abstract The rapid development of ISAs has brought the issue of software compatibility to the forefront in the embedded field. To address this challenge, one of the promising solutions is the adoption of a multiple-ISA processor that supports multiple different ISAs. However, due to constraints in cost and performance, the architecture of a multiple-ISA processor must be carefully optimized to meet the specific requirements of embedded systems. By exploring the RISC-V and ARM Thumb ISAs, this paper proposes RVAM16, which is an optimized multiple-ISA processor microarchitecture for embedded devices based on hardware binary translation technique. The results show that, when running non-native ARM Thumb programs, RVAM16 achieves a significant speedup of over 2.73× with less area and energy consumption compared to using hardware binary translation alone, reaching more than 70% of the performance of native RISC-V programs.

Keywords multiple-ISA processor, architecture, binary translation, RISC-V, embedded

1 Introduction

With the increasing demand in the embedded field, numerous impressive instruction set architectures (ISAs) have emerged, intensifying the competition [1–3]. However, when processors migrate to a new ISA, legacy binary codes without source codes become incapable of execution. Consequently, redeveloping these codes on the new ISA incurs additional software and time costs. Simultaneously, the software ecosystem poses a significant challenge for emerging ISAs. Allowing existing programs, designed for other ISAs, to run on processors based on a newer ISA could assist in sustaining the newer ISA during the period of limited software ecosystem support.

In desktop and server systems, software binary translation systems [4], including dynamic binary translation (DBT) and static binary translation (SBT), are the most common methods to overcome software compatibility issues. These technologies simply involve installing a binary translation system software, allowing programs based on other ISAs to be directly executed through this system. For instance, Apple's latest Rosetta [5] system can efficiently support X86 programs to execute on the ARM-based processor.

Nevertheless, the software binary translation system does have its limitations, particularly in low-cost embedded systems. Firstly, due to the on-the-fly translation process, software DBT necessitates running concurrently with the application program, leading to increased runtime and a lack of optimization. Consequently, its performance when running non-native programs often falls far behind that of native programs [6–9].

Secondly, to comply with area and power constraints, embedded systems typically have limited memory sizes specified in kilobytes to only support running kernel applications [10]. Consequently, there is often insufficient storage space and environment to accommodate a DBT system. This scarcity of resources further hinders the practical implementation of software DBT in the context of embedded systems.

Indeed, a software SBT system does offer advantages by completing the translation offline before execution begins. This approach allows for maximizing the optimization of the translated code, leading to improved performance compared to dynamic binary translation. However, SBT comes with its own set of limitations. Problems such as code discovery, self-modifying code, and indirect branches pose significant challenges for SBT systems [11,12], which limits its applicability in certain scenarios. Addressing these issues requires dynamic analysis and adaptation during program execution, which is not possible with SBT's offline translation approach.

This paper attempts to take a multiple-ISA processor to solve the problem of software compatibility in the embedded field, which can avoid the startup and additional run time of software DBT systems and the problems encountered with SBT. Given that area and power are limited resources in embedded systems, the key focus is on developing a multiple-ISA processor that minimizes resource consumption. To achieve this goal, the paper draws inspiration from two prominent ISAs: RISC-V, an emerging open-source ISA highly regarded by developers, and ARM Thumb, currently the most widely used ISA in the embedded domain. By

Received March 23, 2023; accepted October 30, 2023

E-mail: chengyh105@gmail.com

leveraging the research insights from these ISAs, the paper seeks to devise an efficient and cost-effective multiple-ISA processor architecture and create a processor that not only offers robust software compatibility but also optimizes resource utilization for low-cost embedded systems.

In the subsequent sections of the paper, we delve into the details of the proposed 32-bit low-cost multiple-ISA processor core, called RVAM16, which is built using the hardware binary translation technique. RVAM16 is specifically designed to support both RISC-V and ARM Thumb ISAs, achieved by translating ARM Thumb instructions into RISC-V instructions. Our approach enhances the performance of ARM Thumb programs by minimizing the translation ratio of ARM Thumb instructions to RISC-V instructions. Moreover, the hardware cost of RVAM16 is effectively managed by leveraging 16-bit units and data paths. The evaluation results provide insights into the performance, area, power and energy characteristics of RVAM16, which may be valuable for developers and researchers working in the embedded systems domain.

The following sections are organized as follows. The multiple-ISA processor technique is introduced in Section 2. Section 3 focuses on the microarchitecture and optimization methods for improving the energy efficiency of RVAM16. In Section 4, the area, performance, power and energy consumption of RVAM16 are evaluated by running Dhrystone, CoreMark, Embench benchmark suite, and some real applications. Finally, Section 5 concludes this paper.

2 Multiple-ISA processor

A multiple-ISA processor is designed to enable the direct execution of binary codes from different ISAs by incorporating additional hardware units into the traditional single-ISA processor. Currently, multiple-ISA processors are predominantly utilized to support two ISAs with different instruction encoding lengths that exhibit an inclusive relationship, such as ARM and Thumb. In low-cost and mobile processors, this approach is particularly beneficial in enhancing binary code density and saving memory space.

An ISA-level heterogeneous multi-core processor can also be regarded as a form of the multiple-ISA processor, which runs different program segments on different cores that are based on different ISAs or sub-ISAs to enhance energy efficiency [13–16]. Solutions in this domain must address the critical challenge of effectively migrating programs between cores based on different ISAs [13]. On the other hand, the multiple-ISA processor for solving software compatibility focuses on how to efficiently run the program compiled based on different ISAs on a processor. Obviously, the solution of implementing an additional core like a heterogeneous multi-core processor is too expensive to solve the software compatibility problem.

There are two methods to implement a multiple-ISA processor alongside the multi-core approach. The first way is that the core is equipped with multiple decoders and shares a pipeline backend containing superset features of all ISAs. In this method, instructions from different ISAs will be decoded by different decoders. And another way is translating all instructions into native ISA through hardware binary translation (HBT), and then decoding and executing the translated instructions in a single-ISA pipeline [17,18]. Obviously, the technique of based on HBT has less impact on the microarchitecture and hardware overhead of the traditional single-ISA processor.

Some works focus on accelerating the binary translation process by hardware to reduce the runtime overhead of software DBT [19,20]. However, a significant challenge inherent to DBT systems is the expansion of the instruction count following translation from non-native instructions to native. This phenomenon impacts the performance of executing non-native ISA programs, leading to a substantial performance gap compared to native ISA programs. The multiple-ISA based on HBT also suffers from the same dilemma. For example, when translating ARM into MIPS without optimization, the number of instructions will increase to more than 3 times [17].

We can summarize the advantages and disadvantages of the three techniques to implement a multiple-ISA processor in Table 1. Among these methods, the HBT-based technique is more suitable to support multiple ISAs for the embedded processors because embedded devices are sensitive to the cost, area, and energy consumption. A key consideration for the successful application of the HBT-based approach lies in its ability to enhance the performance of running non-native ISA programs.

Figure 1 shows the multiple-ISA processor microarchitecture based on HBT. It introduces a binary translator into the traditional pipeline, serving as a critical addition when compared to a single-ISA processor. The binary translator's primary function is to seamlessly translate non-native instructions into corresponding native instructions. The binary translator crosses the boundary of the core in Fig. 1 because it can be placed outside or inside the core pipeline.

When situated outside the core, all instructions obtained by the fetch unit are native instructions, thus the pipeline of the multiple-ISA processor is the same as the native single-ISA

Table 1 The features of different multiple-ISA processor techniques

Technique	Pros	Cons
Multi-core	Independent performance for different ISAs	High hardware cost
Multi-decoder	Same performance for different ISAs	Complex pipeline
HBT	Low hardware cost	Lower performance for non-native ISA

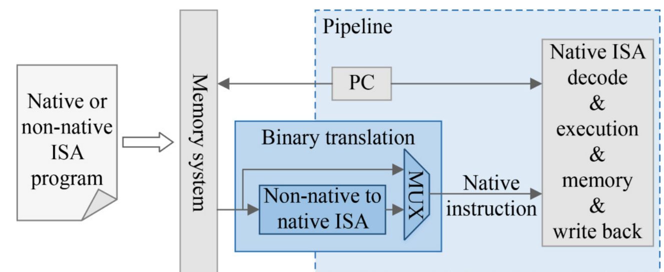


Fig. 1 Multiple-ISA processor microarchitecture based on binary translation

processor except for the control complexity of the Program Counter (PC). The reason is that the translation of a non-native instruction into potentially multiple native instructions, and the PC must remain until all translated instructions are executed. However, the outside binary translator is difficult to optimize the native pipeline for improving the performance of running non-native ISA programs. As a result, this external binary translator setup is more suitable to support a smaller subset ISA on a processor with a larger ISA. In such cases, all instructions from the subset ISA can be effectively translated into a single larger ISA instruction.

When placed inside the core, the ideal location of the binary translator is between the fetch unit and decoder. A non-native instruction translated into multiple native instructions can be treated as a multiple-cycle instruction, necessitating only several stalls in the fetch unit. Obviously, inside binary translator is necessary to modify the structure of the traditional single-ISA pipeline, primarily to accommodate the integration of the translator. Because it is a part of the pipeline, the optimization for binary translation is easy to achieve: the pipeline can support some unique features of non-native ISA controlled by the signals pre-decoded in the binary translator. Therefore, this approach is particularly well-suited for supporting two distinct and independent ISAs, where direct one-to-one translation is challenging.

An essential consideration of a multiple-ISA processor is the method for distinguishing instructions from different ISAs. This can be achieved by manually selecting the running ISA using an external switch. Alternatively, a more sophisticated approach involves automatic identification by allocating the codes of different ISAs to separate address spaces. The mapping between ISAs and address spaces can also be dynamically altered through manipulation of control and status registers (CSRs).

3 RVAM16 architecture

RVAM16 multiple-ISA processor includes two pipeline stages based on RV32I ISA without branch prediction, whose microarchitecture is shown in Fig. 2. When executing an ARM Thumb program, RVAM16 will translate ARM Thumb instruction into RISC-V instruction following alignment by the binary translator. In this section, we will focus on hardware binary translation and the low-cost optimization technology rather than compress instructions decoder (C2I in Fig. 2) or execution unit that are similar to traditional single RISC-V ISA processors.

3.1 Binary translator

The main function of the binary translator is the translation of non-native ISA instructions to native, which includes two categories: Register mapping and instruction mapping.

3.1.1 Register mapping

The process of register mapping entails the establishment of logical registers correspondence between the non-native and the native ISA. The ease of implementation depends on whether the native ISA includes more logical registers than the non-native ISA. If this is the case, the mapping can be straightforwardly accomplished. However, if the non-native ISA contains more logical registers, achieving this mapping might necessitate external memory or supplementary hardware logic.

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). The RISC-V RV32I based ISA includes 32 general-purpose registers, in which two general-purpose registers are regarded as SP and LR, while the PC is an independent special register. Because the R0 Register in 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. All register mapping relationships are shown in Table 2.

3.1.2 Instruction mapping

The objective of instruction mapping is to realize the functionalities of all non-native instructions by leveraging native ISA instructions. As an illustration, consider Listing 1,

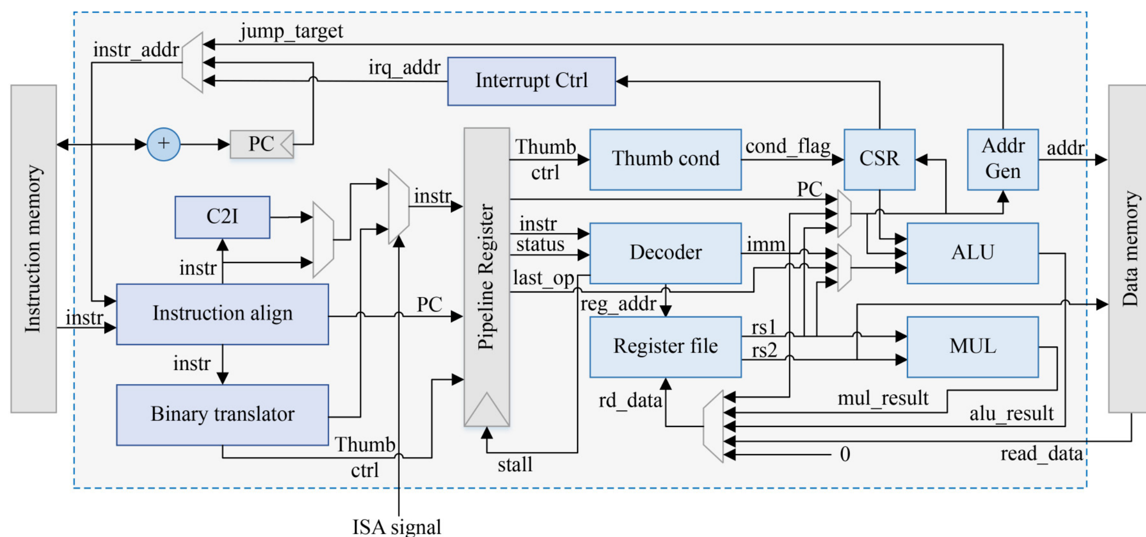


Fig. 2 Microarchitecture of RVAM16 processor core

Table 2 Register mapping from ARM Thumb to RISC-V

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

where RISC-V instructions are harnessed to emulate the ARM Thumb ADDS instruction without necessitating dedicated hardware support.

In this context, the first ADD instruction in Listing 1 serves the purpose of performing the addition operation. The subsequent seven instructions in the sequence cater to supporting the ARM Thumb condition flag bits [21]. The final instruction concludes the sequence by transferring the addition result to the Rd register. This step becomes pertinent due to the potential scenario wherein Rd might be identical to either Rn or Rm in the ADDS instruction.

In the previous subsection, R0−R15 registers of RISC-V are not used in the register mapping, thus, in Listing 1, this allows us to utilize R1−R4 to hold the ARM Thumb condition flag bits and take the remaining registers as temporary registers. Obviously, the quality of instruction mapping determines the performance of the multiple-ISA processor based on binary translation running the non-native ISA program.

It's important to acknowledge that some arithmetic instructions in ARM Thumb have the potential to alter the value of the PC (such as MOV PC, Rm and ADD PC, PC, Rm). In contrast, only specific jump and branch instructions are designed to modify the PC in the RISC-V ISA. In the register mapping, the PC in ARM Thumb architecture is mapped to R31 in RISC-V. Therefore, a careful approach is required to ensure the accurate translation of these types of instructions. In RVAM16, if an ARM Thumb arithmetic instruction requires to modify the PC, the RISC-V AUIPC R31,0 (PC plus an immediate) instruction will be used first to save the PC to R31. After the requisite calculations are performed, the JALR (jump and link register) instruction is executed to change the value of the PC and jump to the target address.

Listing 1 RISC-V instructions for achieving Thumb ADDS

```
//Basic addition
ADD R15, Rn, Rm
//Judge and save sign flag
SLTI R1, R15, 0
//Judge and save zero flag
SLTU R2, R0, R15
XORI R2, R2, 1
//Judge and save carry flag
SLTU R3, R15, Rn
//Judge and save overflow flag
SLTI R5, Rn, 0
SLT R6, R15, Rm
XOR R4, R5, R6
//Save addition result to the Rd
ADDI Rd, R15, 0
```

3.2 Binary translation optimization

As mentioned earlier, the proliferation of instruction count resulting from binary translation is a primary factor contributing to the suboptimal performance when executing non-native ISA programs. We employ the concept of a translation ratio, which succinctly illustrates the alteration in instruction count before and after the translation of a non-native program. Mathematically, the translation ratio is expressed as follows:

$$BT_{ratio} = \frac{Instr_{native}}{Instr_{non-native}}, \tag{1}$$

in which $Instr_{native}$ is the number of native instructions generated by translating, and $Instr_{non-native}$ is the number of non-native instructions before translating. For example, the translation ratio of the aforementioned ARM Thumb ADDS instruction is equal to 9.

Evidently, one avenue to enhance the performance of a multiple-ISA processor based on binary translation running non-native ISA programs involves the reduction of the translation ratio. Furthermore, to cater to the demands of embedded systems, it's imperative to explore optimization strategies that yield significant performance enhancements while introducing minimal hardware overhead. Targeting RISC-V and ARM Thumb, we propose hardware optimization techniques for the ARM Thumb conditional flags, branch instructions, and conditional execution instructions in the RISC-V pipeline. The goal of these optimizations is to curtail the translation ratio between Thumb and RISC-V instructions, thereby facilitating more efficient execution of non-native ARM Thumb programs on the RISC-V processor.

3.2.1 Condition flags

In 9 RISC-V instructions required by ARM Thumb ADDS instruction, 7 instructions are dedicated to altering the condition flag bits of N (Sign), Z (Zero), C (Carry), and V (Overflow). The processing method of flags has a vital impact on the performance of translating and running ARM Thumb programs. To optimize this aspect, RVAM16 incorporates specialized hardware logic and registers in the pipeline (denoted as Thumb cond in Fig. 2). This augmentation empowers the processor to efficiently generate and store the ARM Thumb condition flag bits. By implementing this optimization, the ARM Thumb ADDS instruction can be accomplished through only one RISC-V ADD instruction.

3.2.2 Branch instructions

Diverse flag bit implementations lead to different implementations of conditional branch instructions. For an ISA like RISC-V without condition flags, it determines the branch direction by directly comparing two operands. But for an ISA like ARM Thumb with condition flags, it decides the branch by judging the flag bits generated by preceding instruction. In the instruction mapping, the maximum translation ratio of ARMv6-M (the ISA is based on ARM Thumb and implemented by Cortex-M0) branch instruction is equal to 4.

For reducing the translation ratio of the branch instruction,

RVAM16 integrates the branch condition judgment logic of ARM Thumb in its pipeline, leveraging the aforementioned hardware flag logic and register. All ARM Thumb branch instructions will be translated into a RISC-V BEQ instruction (compare the values of two registers, the branch takes if they are equal). The role of the RS1 field of BEQ instruction is modified to represent the condition code of ARM Thumb when running ARM Thumb programs. Subsequently, the condition code and flag bits collaborate to determine whether a branch is taken in the execution unit.

3.2.3 Condition execution

Another significant difference between ARM Thumb and RISC-V ISAs is that ARM Thumb supports conditional execution, while RISC-V does not. In ARM Thumb, a conditional execution instruction will be treated as a no-operation instruction (NOP) if its condition is unmet. In the context of binary translation, if the execution condition is judged in the decoder, these instructions with a translation ratio exceeding 1 necessitate additional idle cycles, which impacts the overall execution efficiency of ARM Thumb programs.

In order to mitigate such wastage of cycles, an alternative tactic involves shifting the execution condition judgment logic into the binary translator itself. Before an ARM Thumb instruction is translated into RISC-V instruction(s), the binary translator first evaluates the execution condition. Only the instruction that its execution condition is met can be translated, otherwise, it will be processed as a NOP instruction. This approach ensures that instructions failing the execution condition only result in one idle cycle, minimizing pipeline cycle wastage.

Figure 3 depicts the binary translator responsible for translating ARM Thumb instructions into RISC-V instructions while supporting hardware optimization. Beyond translation, this binary translator generates supplementary control signals to regulate flag bit manipulation. When dealing with instructions situated within an IT block, the binary translator additionally assesses whether the execution condition is met. Given that RVAM16 operates with a mere two pipeline stages, the binary translator can always get the latest execution conditions without necessitating the insertion of stall cycles.

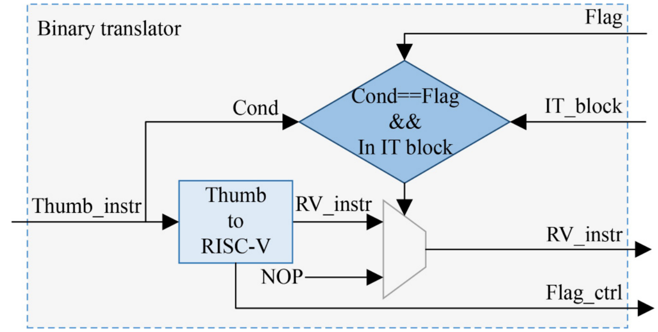


Fig. 3 Architecture of hardware binary translator in RVAM16

3.3 Design for low-cost

To manage the inevitable hardware overhead introduced by the binary translator while ensuring cost-effectiveness, RVAM16 employs a unique low-cost design approach. This approach implements a 32-bit processor based on the 16-bit data path by increasing the execution cycle to reuse the 16-bit logic units. The 16-bit data path can reduce the hardware overhead for a low-cost processor because the 16-bit ALU (arithmetic logic unit) and multiplier take less cost than 32-bit and they are dominating hardware functional units in the low-cost processor. For supporting the 16-bit internal data path, in RVAM16, 32 32-bit registers in RV32I are implemented by the register file containing 64 16-bit registers. This strategy allows RVAM16 to harness the efficiency of a 16-bit data path while accommodating the overall design requirements of RISC-V ISA.

The multiplier integrated into RVAM16 is implemented by a 16-bit single-cycle multiplier, which requires 3 or 4 cycles to get a 32-bit multiplication result. And its divider is a 32-bit divider with operands transfer taking two cycles. Figure 4 shows the structure of ALU and the details of the shifter in RVAM16. The carry bit is required to complete a 32-bit addition operation, thus the adder in RVAM16 must include a carry logic. An advantageous aspect is that this carry logic

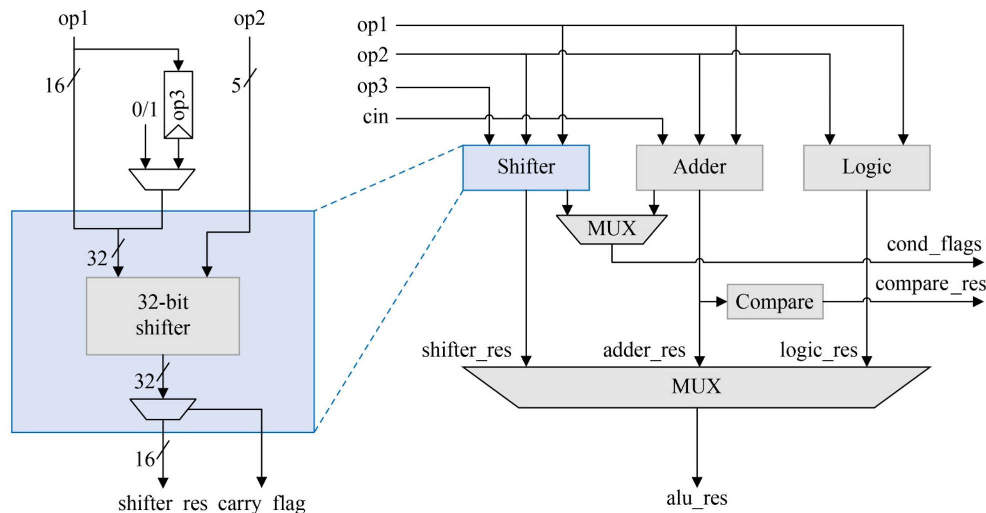


Fig. 4 Structure of RVAM16 ALU

component can be shared with the ARM Thumb condition flags.

The basis of the RVAM16 shifter is a 32-bit shifter because two 16-bit shift operations for achieving a 32-bit operation will affect each other. The 32-bit shifter operand is spliced by two 16-bit operands (op1 and op3 in Fig. 4), and they are organized as follows. In the first execution cycle, op1 is set to the lower (shift left) or higher (shift right) 16-bit data of the 32-bit source operand, and every bit in op3 is set to 0 or 1 according to the shift mode. In the second execution cycle, op1 is set to another 16-bit data and op3 is set to op1 of the previous cycle. In the composition of the 32-bit shift operand, op1 is in the upper half-word and op3 is in the lower half-word if this is a shift left operation, otherwise op1 is in the lower half-word. Finally, a 16-bit result can be obtained at the corresponding op1 position of the 32-bit result per cycle and the carry flag can be obtained at the second cycle.

Generally, it will take two cycles to process a 32-bit operation (excluding multiplication and division) by the 16-bit data path, which has a significant impact on performance. For minimizing the impact of the 16-bit path, we optimize branch instruction by operating upper 16-bit data first to save one cycle when the upper 16-bit of two operands are not equal, and optimize load instruction by writing upper 16-bit data first to save one execution cycle for load byte and load half-word. These optimizations collectively aim to ameliorate the performance implications of utilizing a 16-bit data path in RVAM16, effectively optimizing critical operations such as branches and loads for enhanced overall efficiency.

4 Experiment evaluation

In this section, we introduce the implementation of an RVAM16 prototype multiple-ISA processor, emphasizing its ability to support both ARMv6-M and RISC-V RV32IMC ISAs. We proceed to assess its performance, area utilization, power and energy consumption through rigorous testing using benchmarks and real-world applications that are commonly employed in the embedded domain. This evaluation provides a comprehensive understanding of the RVAM16's capabilities and its potential impact on the multiple-ISA processors to embedded systems.

Furthermore, to provide a meaningful context for our evaluation, we include the Cortex-M0 processor as a reference processor core for comparison. The Cortex-M0 soft core used in the experiment is obtained from the Cortex-M0 DesignStart (DS) Design Kit [22], which natively supports the ARMv6-M ISA. This allows us to benchmark our RVAM16 multiple-ISA processor against a native ARMv6-M supporting processor, thus facilitating a robust assessment of the RVAM16's performance and efficiency in comparison to an established industry-standard solution.

4.1 Translation ratio

Table 3 shows the practical translation ratio of 13 categories of instructions in ARM Thumb, all computation, memory, and branch instructions included in ARMv6-M ISA, by offering a comparison of without and with hardware optimization techniques. For instance, the hardware optimization for the N

Table 3 Instruction translation ratio in ARMv6-M without and with optimization

Instruction category	Instruction	Translation ratio	
		Without optimization	With optimization
Move	MOV Rd, Rm	1	1
	MOV PC, Rm	1	1
	MOV Rd, PC	2	2
	MOVS Rd, Rm	**4**	**1**
	MOVS Rd, imm	**4**	**1**
Add	ADD Rd, SP, imm	1	1
	ADD Rd, Rd, Rm	1	1
	ADD Rd, Rd, PC	2	2
	ADD PC, PC, Rm	3	3
	ADR Rd, <label>	3	3
	ADDS Rd, Rd, imm	**8**	**1**
	ADDS Rd, Rm, Rn	**9**	**1**
	ADDS Rd, Rm, imm	**9**	**1**
	ADCS Rd, Rd, Rm	**16**	**1**
Subtract	SUB SP, SP, imm	2	2
	SUBS Rd, Rm, Rn	**17**	**1**
	SBCS Rd, Rd, Rm	**17**	**1**
	RSBS Rd, Rm, 0	**17**	**1**
	SUBS Rd, Rm, imm	**18**	**1**
	SUBS Rd, Rd, imm	**18**	**1**
Multiply	**MUL Rd, Rm, Rd**	**4**	**1**
Compare	**CMN Rm, imm**	**8**	**1**
	CMP Rm, Rn	**16**	**1**
	CMP Rn, imm	**17**	**2**
Logic	**LOP[a]Rd, Rd, Rm**	**4**	**1**
	BICS Rd, Rd, Rm	**5**	**2**
Shift	**SOP[b]Rd, Rd, Rm**	**8**	**1**
	SOP[b]Rd, Rm, imm	**9**	**1**
	RORS Rd, Rd, Rm	**10**	**6**
Load	LDRI[c] Rd, Rm, imm	1	1
	LDRR[d] Rd, Rm, Rn	2	2
	LDR Rd, <label>	3	3
	LDM Rm, registers	n[e]	n[e]
Store	STRX[f] Rd, Rm, imm	1	1
	STRX[f] Rd, Rm, Rn	2	2
	STM Rm, registers	n[e]	n[e]
Stack	POP registers	n[e]	n[e]
	PUSH registers	n[e]	n[e]
Branch	BXX[g]	1	1
	BGT	**4**	**1**
	BLE	**4**	**1**
Extend	EOP Rd, Rm	2	2
Reverse	REVSH Rd, Rm	5	5
	REV Rd, Rm	11	11
	REV16 Rd, Rm	11	11

[a] LOP(Logic operation) instructions include ANDS, EORS, ORRS, MVNS, TST.
[b] SOP(Shift operation) instructions include LSLS, LSRS, ASRS.
[c] LDRI(Load data with immediate offset) instructions include LDR, LDRH, LDRB.
[d] LDRR(Load data with register offset) instructions include LDR, LDRH, LDRB, LDRSH, LDRSB.
[e] The number of RISC-V instructions translated by LDM, STM, POP and PUSH depends on the number of registers that need to be read and written.
[f] STRX(Store data with immediate or register offset) instructions include STR, STRH, STR.
[g] BXX(Branch with condition) instructions include BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC, BHI, BLS, BGE, BLT.

and Z condition flag bits reduces the translation ratio of instructions in the move, multiply and logic categories by about 3. Furthermore, shift instructions benefit from optimization applied to the N, Z, and C condition flags.

The translation ratio for addition, subtraction, and comparison instructions sees a significant decrease, which thanks to the optimization for all condition flags. Lastly, the translation ratios of BGT (Branch if Greater Than) and BLE (Branch if Less than or Equal) instructions decreases from a maximum of 4 to 1 subsequent to the implementation of condition flags and branch judgment logic. All of these optimizations collectively contribute to improving the performance of running non-native ARM Thumb codes on our RVAM16 multiple-ISA processor.

4.2 Performance

The performance of the RVAM16 prototype Multiple-ISA processor is evaluated by running different benchmarks in the Verilog simulator tool. For native RISC-V codes execution, RVAM16 exhibits commendable performance, achieving a peak performance of 0.92 DMIPS/MHz and 1.51 CoreMark/MHz for running Dhrystone and CoreMark compiled by GCC for RISC-V with the -O3 option. Comparatively, RVAM16 reaches about 71% and 69% of the performance of Ibex processor core, a traditional 32-bit RISC-V processor featuring a 2-stage pipeline whose peak performance is 1.29 DMIPS/MHz and 2.20 CoreMark/MHz. This means that the narrower 16-bit data path in RVAM16 is responsible for about 30% of the performance reduction.

Illustrated in Fig. 5, within the Cortex-M0 DS system environment, the performance of RVAM16 running non-native ARM Thumb codes of Dhrystone and CoreMark achieves 71% and 76% of the performance running native RISC-V codes. By integrating hardware optimizations in addition to HBT, RVAM16 demonstrates a notable 2.73× speed enhancement during the execution of ARM Thumb codes in Dhrystone, and a substantial 4.31× speedup when executing CoreMark. Furthermore, when running the same ARM Thumb codes, RVAM16 attains 65% of the performance level achieved by the Cortex-M0 processor.

These outcomes highlight the effectiveness of RVAM16's hardware optimizations and its ability to significantly enhance the performance of executing ARM Thumb code when

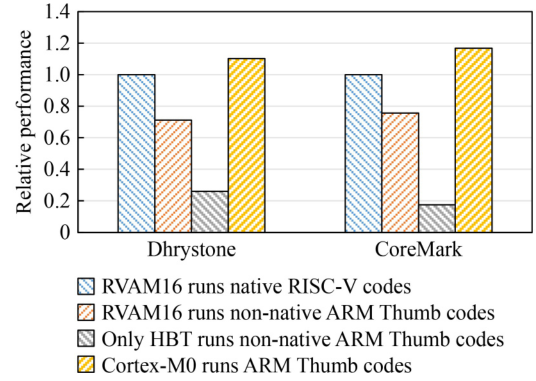


Fig. 5 Performance of running Dhrystone and CoreMark benchmarks

compared to the sole HBT. Moreover, the processor's performance in executing ARM Thumb code demonstrates competitive prowess, marking it as a promising solution for addressing software compatibility challenges in embedded systems.

We additionally investigate RVAM16's performance characteristics while executing various programs in Fig. 6, utilizing the Embench benchmark suite. On average, when executing non-native ARM Thumb codes, RVAM16 showcases a 3.86× performance increase compared to employing only HBT, which is 68% of running native RISC-V codes.

Certainly, a notable aspect is RVAM16's exceptional performance improvement in certain benchmarks from the Embench suite, particularly in cases such as *Nbody*, *Minver*, and *St*. These benchmarks prominently consist of computation and control instructions, a category with initially high translation ratios that benefit significantly from the applied hardware optimizations. In these specific scenarios, RVAM16 achieves an impressive maximum speedup of 5.86× compared to HBT.

The performance result of the *Matmult-int* benchmark provides insight into the trade-off made by RVAM16 between multiplication performance and hardware overhead. In the case of executing ARM Thumb code for the *Matmult-int* benchmark, the performance of RVAM16 is only 33% that of Cortex-M0. The reason is that Cortex-M0 includes a 32-bit single-cycle multiplier, which significantly contributes to its

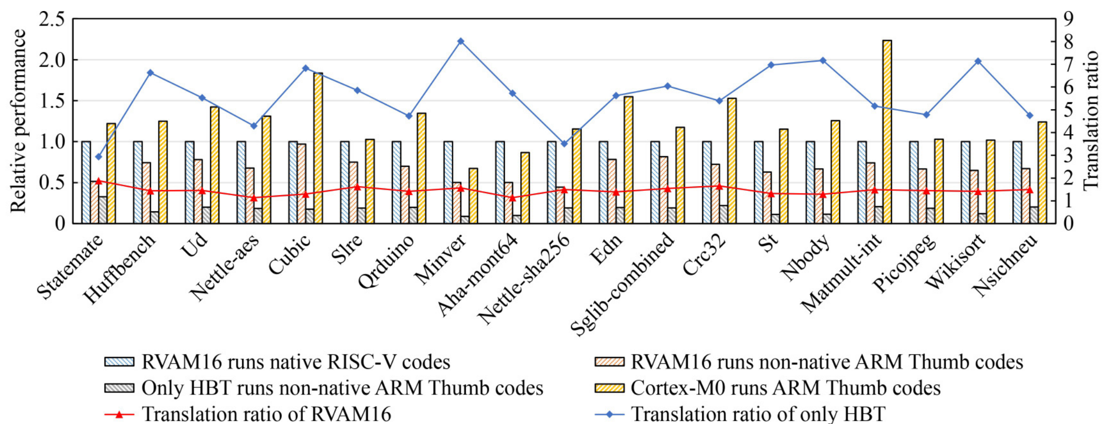


Fig. 6 Performance of running Embench benchmark suite

multiplication capabilities.

The point line depicted in Fig. 6 represents the average translation ratio for each benchmark within the Embench suite. Notably, in HBT without optimization, the *Minver* records the highest translation ratio, reaching 8.02 while the *Statemate* exhibits the lowest translation ratio. This discrepancy can be attributed to involving more memory instructions in *Statemate* but more branch and integer arithmetic instructions to implement float point operations in *Minver*.

Since the hardware optimization strategies in RVAM16 primarily target arithmetic operation instructions, a notable difference is observed on the translation ratios of various benchmarks. Specifically, for the *Minver* benchmark, the translation ratio experiences a substantial reduction of 6.45, marking the most significant decline.

When considering the entire Embench benchmark suite, the hardware optimizations yield a considerable reduction in translation ratios. On average, the translation ratio decreases from 5.63 to 1.45. This reduction in translation ratios further results in a remarkable 74% decrease in the total number of instructions generated through binary translation.

In order to evaluate the performance of RVAM16 in real application scenarios, Table 4 shows the relative execution time (compared to Cortex-M0) of running kernel program snippets from different real typical general-purpose applications in the embedded field. These real applications include 1) communication program, 2) temperature control system, and 3) Flash control system.

Table 4 Relative execution time of RVAM16 running real applications

Real case	Kernel program	Execution time	
		RISC-V	Thumb
Communication program	UART	1.10	1.23
Temperature control system	Sensor access	1.19	1.38
Flash control system	Flash access	1.50	1.53

Overall, RVAM16 is only 1.26× slower than Cortex-M0 when running RISC-V codes and 1.38× slower when running ARM Thumb codes. RVAM16 performs better in the communication program because there is more byte-based memory access in UART (Universal Asynchronous Receiver/Transmitter) communication. On the contrary, the Flash access program includes more normal word-based operations, contributing to a more pronounced performance gap between RVAM16 and Cortex-M0.

Lastly, Table 5 lists the performance slowdown of running non-native ISA programs (compared to native programs) in the state-of-the-art software binary translation systems and HBT-based multiple-ISA processors. RVAM16 showcases a performance that outperforms software DBT systems, positioning itself in proximity to the performance levels achievable with software SBT. HBT-based RVAM16 also does not suffer from the problems of the software SBT and can be used in all applications. Considering these insights, the efficacy of a multiple-ISA processor like RVAM16 in addressing software compatibility issues within the embedded domain is underscored.

4.3 Area, power, and energy consumption

The second column in Table 6 presents the area occupied by the processor cores after synthesis at 100 MHz and 0.81 V with 28 nm process technology. Compared with the base core that only supports RV32IMC ISA with the 16-bit data path,

Table 5 Compareison between state-of-the-art software binary translation system and HBT-based multiple-ISA processors

	Name	Technique	Native ISA	Non-native ISA	Slowdown
Software	QEMU [6]	DBT	X86/ARM/Other	RISC-V	7.07× [12]
	QEMU	DBT	RISC-V	X86	more than 3.50× [23]
	RV8 [9]	DBT	X86	RISC-V	2.60×
	Lupori et al. [12]	SBT	X86/ARM	RISC-V	1.12×/1.35×
Hardware	Capella et al. [18]	HBT + extended MIPS	MIPS	X86/ARM/PowerPC	about 1.57×/2.03×/2.22×
	HBT	HBT	RISC-V	ARM	4.89×
	RVAM16	HBT + hardware optimization	RISC-V	ARM	1.40×

Table 6 Area, power and energy consumption of processors at 100 MHz and 0.81 V

Core	Area (μm^2)	Benchmark (Target ISA)	Power (mW)			Relative energy
			Static	Dynamic	Total	
RVAM16	24269.28	Dhrystone (RISC-V)	1.69	1.51	3.20	0.84
		CoreMark (RISC-V)	1.69	1.59	3.28	0.90
		Dhrystone (ARM Thumb)	1.71	1.70	3.41	1.25
		CoreMark (ARM Thumb)	1.70	1.78	3.48	1.26
RISC-V + HBT	26658.41	Dhrystone (RISC-V)	1.86	1.48	3.34	0.87
		CoreMark (RISC-V)	1.86	1.55	3.41	0.94
		Dhrystone (ARM Thumb)	1.87	1.77	3.64	3.65
		CoreMark (ARM Thumb)	1.87	1.71	3.58	5.59
Base RISC-V Core	20757.91	Dhrystone (RISC-V)	1.37	1.42	2.79	0.73
		CoreMark (RISC-V)	1.36	1.53	2.89	0.79
Ibex	32802.67	Dhrystone (RISC-V)	2.29	2.70	4.99	0.93
		CoreMark (RISC-V)	2.29	3.26	5.55	1.04
Cortex-M0	26264.45	Dhrystone (ARM Thumb)	2.41	1.80	4.21	1.00
		CoreMark (ARM Thumb)	2.41	1.85	4.26	1.00

the area requirement for RVAM16 increases by 3511.37 μm^2 to support the ARMv6-M ISA, which is about 13% of the total area of Cortex-M0 core. Overall, the area of RVAM16 remains slightly lower than that of the Cortex-M0. As a result, the integration of RVAM16 into existing Cortex-M0-based embedded systems will not impose any additional hardware overhead.

The results also show that the area advantage of 16-bit functional units and data paths is significant in such low-cost processors. Compared with Ibex processor, the area of our foundational RISC-V processor experiences a reduction of approximately 37%. Furthermore, the area required by RVAM16 is even smaller than that resulting from solely implementing HBT. This phenomenon can be attributed to the reduction in translation ratio, which in turn decreases the complexity of the binary translator for instruction mapping. The benefits stemming from this optimization outweigh all costs introduced by the additional hardware logic.

The first column in Fig. 7 shows the ratio of the area of each unit in RVAM16. In this distribution, the dominant contributors to the total area include the register file (RF), accounting for 36%, and the multiplier and divider unit (MULDIV), constituting 34%. This distribution aligns with patterns observed in conventional low-cost processors, primarily due to the fact that these specific units derive limited benefits from the 16-bit data path. The binary translator occupies only 8% of RVAM16 area since the hardware optimization logic offloads some of the cost to the control logic.

Based on the netlist file generated by synthesis and switch activities file extracted by the simulation running Dhrystone and CoreMark, we employ the power analysis tool to comprehensively assess the power consumption of RVAM16 running different programs. The static and dynamic power results are depicted in the fourth and fifth columns of Table 6. The static power in the same processor is similar when running different programs, while the dynamic power is somewhat variable because of the different switch activities generated by different programs. In aggregate, RVAM16's power consumption remains advantageous. Its fewer logic cells contribute to lower total power consumption compared to both the Ibex and Cortex-M0.

In Fig. 7, a parallel observation can be made regarding power distribution within RVAM16. Analogous to the area distribution, the dominant contributors to power consumption are the register file and the multiplier and divider unit. Furthermore, when comparing the power consumption associated with running ARM Thumb codes against RISC-V codes, a substantial portion of the additional power can be attributed to the dynamic power of the binary translator. This difference arises primarily due to the fact that the binary translator remains inactive when executing RISC-V codes.

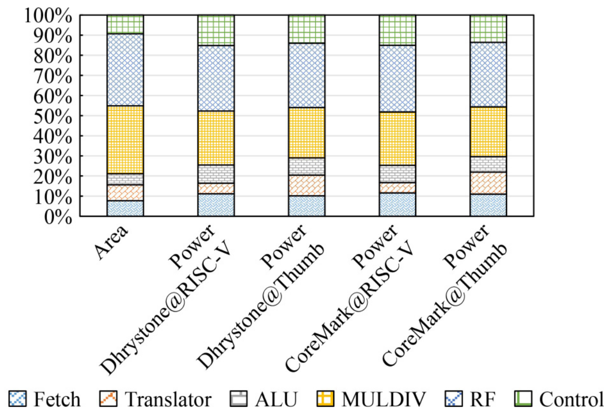


Fig. 7 Area and power distribution of RVAM16

Finally, we get the relative energy consumption by multiplying the total power by the relative execution time of running Dhrystone and CoreMark, which is listed in the last column of Table 6. Energy consumption comprehensively reflects the power and performance of a processor running a certain program. According to the results, the energy of HBT without hardware optimization running ARM Thumb code is far more than Cortex-M0 because of the performance gap (as shown in Fig. 5). With improved performance, the energy of RVAM16 running ARM Thumb code is only 25% more than Cortex-M0. Moreover, because of lower power and comparable performance, RVAM16 running RISC-V code consumes less energy than Ibex and Cortex-M0.

5 Conclusion

Based on the RISC-V and ARM Thumb ISAs, this paper proposes RVAM16 multiple-ISA processor microarchitecture and implements a prototype processor that supports both RISC-V RV32IMC and ARMv6-M ISAs. By optimizing the flag bits, branch instructions, and conditional execution in the ARM Thumb architecture, RVAM16 significantly reduces the performance gap between running native ISA programs and non-native ISA programs in HBT-based multiple-ISA processors, which is far better than that of software DBT and close to that of software SBT. Without suffering from SBT's dilemma, RVAM16 can directly execute all non-native ISA programs. The performance of the RVAM16 prototype processor running ARM Thumb programs reaches 65% of the Cortex-M0 (natively supporting ARMv6-M ISA) running the same programs with similar hardware cost and about 25% extra energy consumption. As a result, integrating RVAM16 into the existing Cortex-M0-based embedded system will not add additional hardware overhead. Moreover, RVAM16 retains the potential to extend its support to additional ARM Thumb and RISC-V ISAs by expanding its instructions mapping capabilities. By relinquishing certain low-cost optimizations, the processor can achieve enhanced performance. Furthermore, with suitable adaptations to the binary translator and related hardware optimization units, the proposed architecture can be effectively harnessed to accommodate any pair of distinct ISAs. This flexibility positions the RVAM16 microarchitecture as a compelling and versatile solution for addressing the challenges of software compatibility stemming from diverse ISAs.

Acknowledgements This work was supported in part by the National Natural Science Foundation of China (Grant Nos. 62272475, 62090023, and 62172430), the National Key R&D Program of China (No. 2021YFB0300300), the Natural Science Foundation of Hunan Province of China (Nos. 2022JJ10064 and 2021JJ10052), the STIP of Hunan Province (No. 2022RC3065), and the Key Laboratory of Advanced Microprocessor Chips and Systems.

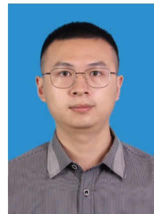
Competing interests The authors declare that they have no competing interests or financial conflicts to disclose.

References

1. Adegbija T, Rogacs A, Patel C, Gordon-Ross A. Microprocessor optimizations for the internet of things: a survey. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018, 37(1): 7–20
2. Saso K, Hara-Azumi Y. Simple instruction-set computer for area and energy-sensitive IoT edge devices. In: Proceedings of the 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP). 2018, 1−4
3. Saso K, Hara-Azumi Y. Revisiting simple and energy efficient embedded processor designs toward the edge computing. IEEE Embedded Systems Letters, 2020, 12(2): 45–49
4. Sites R L, Chernoff A, Kirk M B, Marks M P, Robinson S G. Binary translation. Communications of the ACM, 1993, 36(2): 69–81
5. Apple Inc. About the Rosetta translation environment. See developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment website, Accessed: 2023
6. Bellard F. QEMU, a fast and portable dynamic translator. In: Proceedings of the USENIX Annual Technical Conference. 2005, 41−46
7. Hong D Y, Hsu C C, Yew P C, Wu J J, Hsu W C, Liu P, Wang C M, Chung Y C. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In: Proceedings of the 10th International Symposium on Code Generation and Optimization. 2012, 104−113
8. Ilbeyi B, Lockhart D, Batten C. Pydgin for RISC-V: a fast and productive instruction-set simulator. In: Proceedings of the 3rd RISC-V Workshop. 2016
9. Clark M, Hoult B. rv8: a high performance RISC-V to x86 binary translator. In: Proceedings of the 1st Workshop on Computer Architecture Research with RISC-V (CARRV). 2017
10. Sabri C, Kriaa L, Azzouz S L. Comparison of IoT constrained devices operating systems: a survey. In: Proceedings of the 14th International Conference on Computer Systems and Applications (AICCSA). 2017, 369−375
11. Shen B Y, Chen J Y, Hsu W C, Yang W. LLBT: an LLVM-based static binary translator. In: Proceedings of 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems. 2012, 51−60
12. Lupori L, Rosario V, Borin E. Towards a high-performance RISC-V emulator. In: Proceedings of 2018 Symposium on High Performance Computing Systems (WSCAD). 2018, 213−220
13. Venkat A, Tullsen D M. Harnessing ISA diversity: design of a heterogeneous-ISA chip multiprocessor. In: Proceedings of the 41st International Symposium on Computer Architecture (ISCA). 2014, 121−132
14. Venkat A, Basavaraj H, Tullsen D M. Composite-ISA cores: enabling multi-ISA heterogeneity using a single ISA. In: Proceedings of 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). 2019, 42−55
15. Balkind J, Lim K, Schaffner M, Gao F, Chirkov G, Li A, Lavrov A, Nguyen T M, Fu Y, Zaruba F, Gulati K, Benini L, Wentzlaff D. BYOC: a ”bring your own core” framework for heterogeneous-ISA research. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems. 2020, 699−714
16. Rokicki S, Rohou E, Derrien S. Hybrid-DBT: hardware/software dynamic binary translation targeting VLIW. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019, 38(10): 1872–1885
17. Capella F M, Brandalero M, Carro L, Beck A C S. A multiple-ISA reconfigurable architecture. Design Automation for Embedded Systems, 2015, 19(4): 329–344
18. Fajardo J, Rutzig M B, Carro L, Beck A C S. Towards a multiple-ISA embedded system. Journal of Systems Architecture, 2013, 59(2): 103–119
19. Chai K, Wolff F, Papachristou C. XBT: FPGA accelerated binary translation. In: Proceedings of IEEE National Aerospace and Electronics Conference. 2021, 365−372
20. Rokicki S, Rohou E, Derrien S. Hardware-accelerated dynamic binary translation. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE). 2017, 1062−1067
21. Waterman A, Asanovic K. The RISC-V instruction set manual: volume I: unprivileged ISA. 2019
22. ARM. ARM® Cortex®-M0 DesignStart™ RTL Testbench: user guide. 2015
23. Wang W, Liu X, Yu J, Li J, Mao Z, Li Z, Ding C, Zhang C. The design and building of openKylin on RISC-V architecture. In: Proceedings of the 15th International Conference on Advanced Computer Theory and Engineering (ICACTE). 2022, 88−91

Libo Huang received his BS and PhD degree in computer engineering from National University of Defense Technology, China in 2005 and 2010, respectively. He is a professor at College of Computer Science and Technology, National University of Defense Technology, China. His research interests include computer architecture, hardware/software co-design, VLSI design, and on-chip communication.

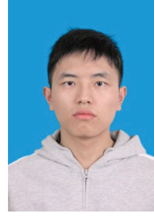
Jing Zhang received his BS degree in electronic commerce from Northwest Agriculture and Forestry University, China in 2020. He is a Master student at College of Computer Science and Technology, National University of Defense Technology, China. His research interests include microprocessor architecture and AI accelerator.



Ling Yang received his BS degree in integrated circuit design and integrated systems from Chongqing University, China in 2020, and MS degree in electronic science and technology from National University of Defense Technology, China in 2022. He is a PhD candidate at College of Computer Science and Technology, National University of Defense Technology, China. His research interests include microprocessor architecture.

Sheng Ma received his BS and PhD degree in computer science and technology from National University of Defense Technology, China in 2007 and 2012, respectively. He is a professor at College of Computer Science and Technology, National University of Defense Technology, China. His research interests include on-chip networks and SIMD architecture.

Yongwen Wang received his PhD degree in computer science from National University of Defense Technology, China in 2004. He is a professor at College of Computer Science and Technology, National University of Defense Technology, China. His research interests include computer architecture and high performance computing.



Yuanhu Cheng received his BS degree in computer science and technology from Sichuan University, China in 2018, and MS degree in computer science and technology from National University of Defense Technology, China in 2021. His research interests include microprocessor architecture.