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

Yuanhu Cheng, Libo Huang, Yijun Cui, Sheng Ma, Yongwen Wang, Bincai Sui

National University of Defense Technology

Changsha, China

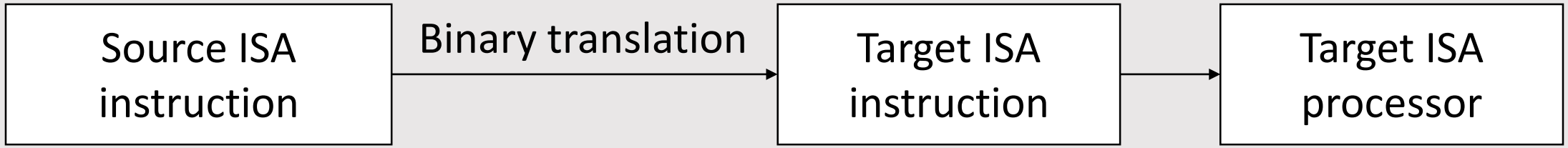
Contents

Software compatibility

- A lot of existing software is developed based on a specific ISA
- Software compatibility: Software based on one ISA cannot run directly on another ISA processor.
- Before the software ecosystem is perfected, the software cost caused by software compatibility will seriously hinder the development of RISC-V
- Solve software compatibility ——Run other ISA programs using RISC-V processor

How to solve software compatibility

- Software methods
  - Software binary translation system
- Hardware methods (multiple-ISA processor)
  - Hardware binary translation
  - Multiple decoders for multiple ISAs
  - Multi-core for multiple ISAs



Solve software compatibility for embedded

- Running environment and performance limit the use of existing software method.
- Hardware methods must meet the requirements of the embedded processor: Area and Power.
- Simplicity is essential —— we try to use hardware binary translation to achieve a multiple-ISA processor to solve the software compatibility problem that RISC-V faces in the embedded field

Supporting ARM Thumb with RISC-V

- Binary interpreter: Registers and instructions mapping
- Some optimizations to improve performance
  - Optimization 1: Condition flags
  - Optimization 2: Branch instruction
  - Optimization 3: Conditional execution

Instruction and register mapping

- Instruction mapping is to convert the ARM Thumb instruction into the corresponding RISC-V instruction(s).
- Register mapping can be achieved by adding a prefix in front of the ARM Thumb register number

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

Optimization 1: Condition flags

- ARM Thumb condition flags:
  - Negative flag (N)
  - Zero flag (Z)
  - Carry flag (C)
  - Overflow flag (V)
- 7 RISC-V instructions are needed to judge these flags

```
1  ;Judge and save N Flag
2  SLTI  R1 , Rd , 0
3  ;Judge and save Z Flag
4  SLTU  R2 , R0 , Rd
5  XORI  R2 , R2 , 1
6  ;Judge and save C flag
7  SLTU  R3 , Rd , Rn
8  ;Judge and save V flag
9  SLTI  R5 , Rn , 0
10 SLT   R6 , Rd , Rm
11 XOR   R4 , R5 , R6
```

Optimization 1: Condition flags

- Optimization: supporting condition flags by hardware in RISC-V processor
  - ALU
  - Flags register
  - Control signal

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

Optimization 2: Branch instruction

- Different condition flags implementations lead to different ways to implement branch instructions.
- In the worst case, 9 RISC-V instructions are needed to achieve an ARM Thumb branch instruction
- Optimization
  - The role of RS1 field of RISC-V BEQ instruction is modified to represent the condition code (named “ cond") of the ARM Thumb
  - Hardware logic is added to judge the flags according to the condition code in the execution stage

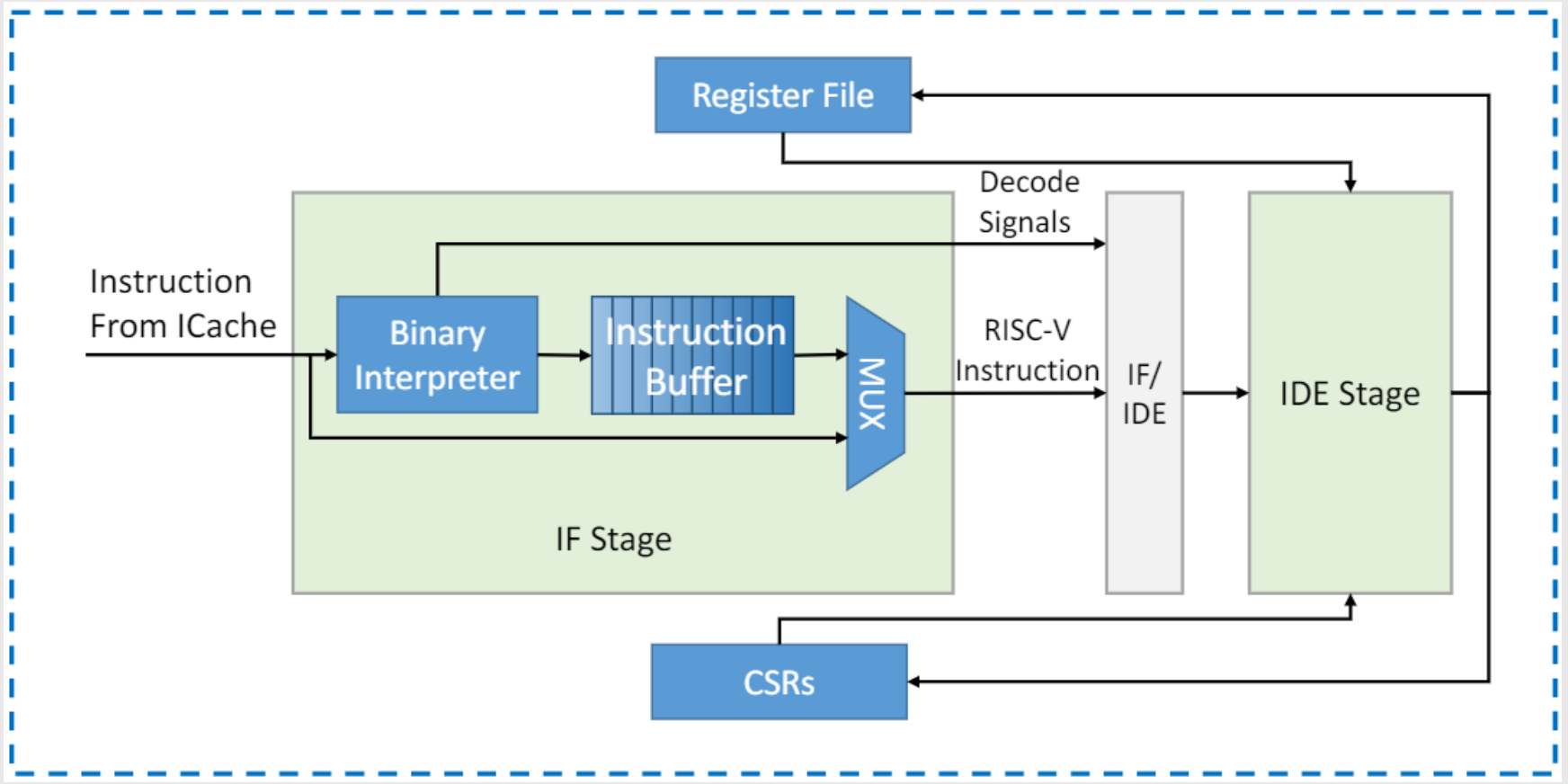
```
1  ;Load flags to general register
2  CSRRCI  R1 , 0 x20c , 0
3  ;Get V flag
4  ANDI  R2 , R1 , 0 0 0 1 b
5  ;Get N flag
6  SRLI  R3 , R1 , 3
7  ;N==V ? R5=0 : R5=1
8  SUB  R5 , R2 , R3
9  SLTU  R5 , R0 , R5
10 ;Get Z flag
11 ANDI  R4 , R1 , 0 1 0 0 b
12 SRLI  R4 , R4 , 2
13 ;(N==V and Z==0) ? R5=0 : R5!=0
14 ADD  R5 , R5 , R4
15 ;If R5==0 branch take
16 BEQ  R5 , R0 , imm
```

Optimization 3: Conditional execution

- ARM Thumb supports conditional execution
  - There is an IT block after each IT instruction
  - The instructions in the IT block are conditional execution
  - An 8-bits register named EPSR.IT is used to support conditional execution
- Judging the execution conditions in the execution stage will cause a large number of pipeline cycles to be wasted
- Optimization: Putting judgment logic of the execution condition into the binary interpreter

An example for ARMv6-M

- This example is based on the open-source core of PULPino, called Zero-riscy (Ibex).
- ARMv6-M ISA
- Microarchitecture



Benchmark

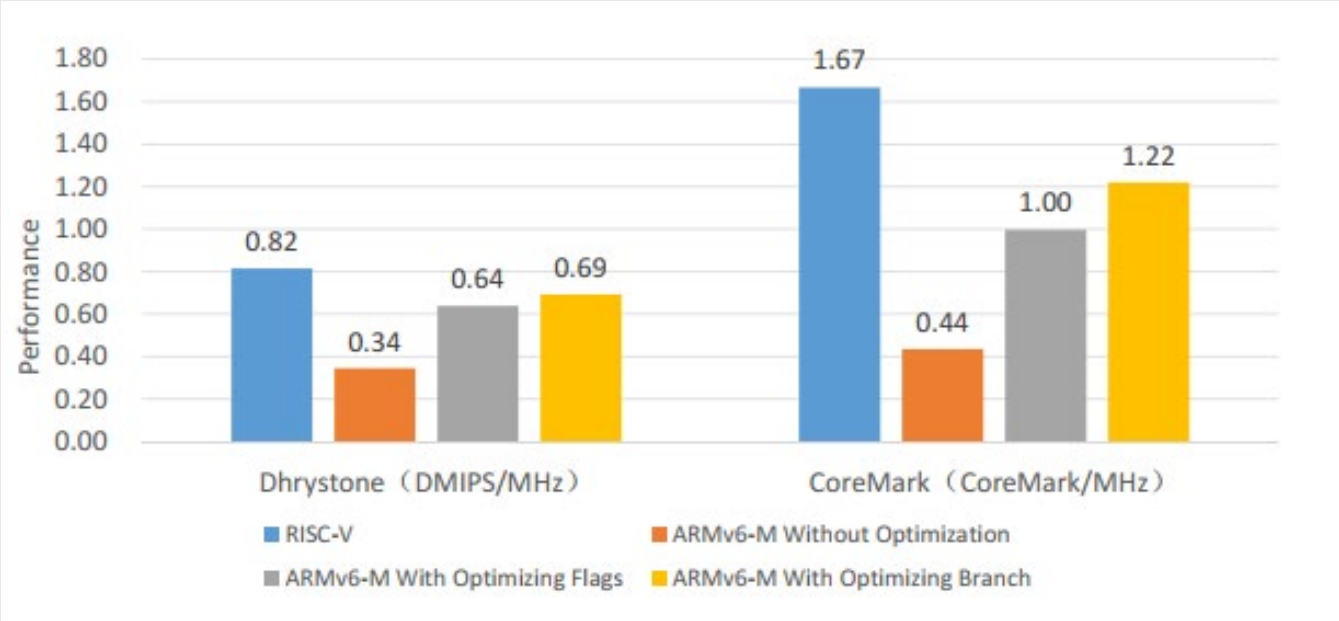
- Dhrystone and CoreMark
- Compiler
  - ARMCC for ARM Thumb
  - GNU GCC for RISC-V

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

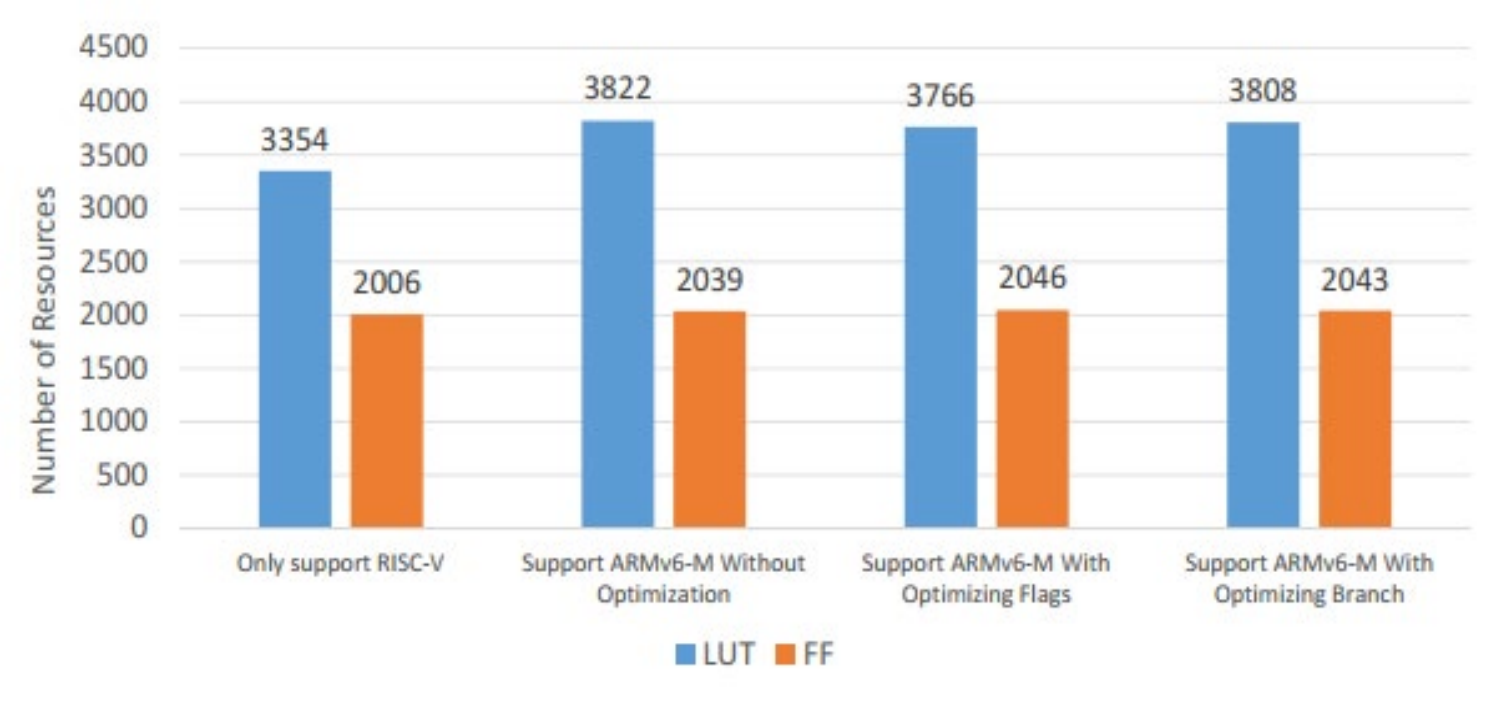
Performance

- Dhrystone
  - RISC-V: 0.82 DMIPS/MHz
  - ARMv6-M: 0.69 DMIPS/MHz
- CoreMark
  - RISC-V: 1.67 CoreMark/MHz
  - ARMv6-M: 1.22 CoreMark/MHz



FPGA resources

- LUT consumption increased by 454, 13.5% of Zero-riscy.
- FF consumption increased by 37, 1.8% of Zero-riscy



Conclusion

- Software ecosystem challenge of RISC-V and the methods for salving software compatibility
- Support ARM Thumb ISA based on binary translation
  - Instructions and registers mapping
  - Condition flags
  - Branch instruction
  - Condition execution
- An Example based on Zero-riscy
  - Dhrystone and CoreMark
  - 0.69 DMIPS/MHz and 1.22 CoreMark/MHz
  - FPGA resources increased by less than 13.5%

Thank You!

If you have any questions, please let us know!