Project 1.2: RVC instructions to RISC-V instructions in RISC-V

Computer Architecture I ShanghaiTech University
Project 1.1 Project 1.2 Project 2.1

IMPORTANT INFO - PLEASE READ

The projects are part of your design project worth 2 credit points. As such they run in parallel to the actual course. So be aware that the due date for project and homework might be very close to each other! Start early and do not procrastinate.

Introduction

In project 1.2, you will implement a translator that converts 16-bit RISC-V Compressed (RVC) instructions to equivalent 32-bit RISC-V instructions. Instructions are provided and output in binary form. This project is easy if you gain sufficient understanding of RISC-V from the courses, Lab 3, Lab 4 and its compressed instruction set extension from Project1.1.

To simplify the implementation, all RVC instructions map to a single existing RISC-V instruction. In this project, you only need to translate part of RVC instructions that are mentioned in Project 1.1. The instruction set is listed below.

The Instruction Set

RVC INSTR	FORMAT	RISC-V EQUIVALENT	FORMAT	OP	FUNCT3	FUNCT7
c.add	CR	add rd, rd, rs2	R	0110011	000	0000000
c.mv	CR	add rd, x0, rs2	R	0110011	000	0000000
c.jr	CR	jalr x0, 0 (rs1)	I	1100111	000	
c.jalr	CR	jalr x1, 0 (rs1)	I	1100111	000	
c.li	CI	addi rd, x0, imm	I	0010011	000	
c.lui	CI	lui rd, nzimm	U	0110111		
c.addi	CI	addi rd, rd, nzimm	I	0010011	000	
c.slli	CI	slli rd, rd, shamt	I	0010011	001	0000000
c.lw	CL	lw rd', offset (rs1')	I	0000011	010	
c.sw	CS	sw rs2', offset (rs1')	S	0100011	010	
c.and	CA	and rd', rd', rs2'	R	0110011	111	0000000
c.or	CA	or rd', rd', rs2'	R	0110011	110	0000000
c.xor	CA	xor rd', rd', rs2'	R	0110011	100	0000000
c.sub	CA	sub rd', rd', rs2'	R	0110011	000	0100000
c.beqz	CB	beq rs1', x0, offset	SB	1100011	000	

RVC INSTR	FORMAT	RISC-V EQUIVALENT	FORMAT	OP	FUNCT3	FUNCT7
c.bnez	CB	bne rs1', x0, offset	SB	1100011	001	
c.srli	CB	srli rd', rd', shamt	I	0010011	101	0000000
c.srai	CB	srai rd', rd', shamt	I	0010011	101	0100000
c.andi	CB	andi rd', rd', imm	I	0010011	111	
c.j	CJ	jal x0, offset	UJ	1101111		
c.jal	CJ	jal x1, offset	UJ	1101111		

Although some of the RVC instruction can be expanded in different ways, we suggest you to follow the above translation or you will fail some testcases.

Instruction Formats

A detailed description of compressed instruction formats is provided for you. You can either refer to The RISC-V Instruction Set Manual, Chapter 16, or The RISC-V Compressed Instruction Set Manual Version 1.9. If there are any mistakes below, the document would prevail.

CR Format

CR	FUNCT4	RD/RS1	RS2	OPCODE
Bits	4	5	5	2
C.ADD	1001	dest≠ 0	src≠ 0	10
C.MV	1000	dest≠ 0	src≠ 0	10
C.JR	1000	src≠ 0	0	10
C.JALR	1001	src≠ 0	0	10

CI Format

CI	FUNCT3	IMM	RD/RS1	IMM	OPCODE
Bits	3	1	5	5	2
C.LI	010	imm[5]	dest≠ 0	imm[4:0]	01
C.LUI	011	nzimm[17]	dest≠{0,2}	nzimm[16:12]	01
C.ADDI	000	nzimm[5]	dest≠ 0	nzimm[4:0]	01
C.SLLI	000	shamt[5]	dest≠ 0	shamt[4:0]	10

CL Format

CL	FUNCT3	IMM	RS1'	IMM	RD'	OPCODE
Bits	3	3	3	2	3	2
C.LW	010	offset[5:3]	base	offset[2|6]	dest	00

CS Format

CS	FUNCT3	IMM	RS1'	IMM	RS2'	OPCODE
Bits	3	3	3	2	3	2
C.SW	110	offset[5:3]	base	offset[2|6]	src	00

CA Format

CA	FUNCT6	RD'/RS1'	FUNCT2	RS2'	OPCODE
Bits	6	3	2	3	2
C.AND	100011	dest	11	src	01
C.OR	100011	dest	10	src	01
C.XOR	100011	dest	01	src	01
C.SUB	100011	dest	00	src	01

CB Format

CB-TYPE1	FUNCT3	IMM	RD'/RS1'	IMM	OPCODE
Bits	3	3	3	5	2
C.BEQZ	110	offset[8|4:3]	src	offset[7:6|2:1|5]	01
C.BNEZ	111	offset[8|4:3]	src	offset[7:6|2:1|5]	01

CB-TYPE2	FUNCT3	IMM	FUNCT2	RD'/RS1'	IMM	OPCODE
Bits	3	1	2	3	5	2
C.SRLI	100	shamt[5]	00	dest	shamt[4:0]	01
C.SRAI	100	shamt[5]	01	dest	shamt[4:0]	01
C.ANDI	100	imm[5]	10	dest	imm[4:0]	01

CJ Format

CJ	FUNCT3	JUMP TARGET	OPCODE
Bits	3	11	2
C.J	101	offset[11|4|9:8|10|6|7|3:1|5]	01
C.JAL	001	offset[11|4|9:8|10|6|7|3:1|5]	01

And recall the RV32 instrcution formats which has been covered in the lecture. The necessary knowledge for the project is provided below. You can either consult The RISC-V Instruction Set Manual, Chapter 2, or RISC-V Green Card for further information.

R Format

R	FUNCT7	RS2	RS1	FUNCT3	RD	OPCODE
Bits	7	5	5	3	5	7

I Format

I	IMM[11:0]	RS1	FUNCT3	RD	OPCODE
Bits	12	5	3	5	7

S Format

S	IMM[11:5]	RS2	RS1	FUNCT3	IMM[4:0]	OPCODE
Bits	7	5	5	3	5	7

SB Format

SB	IMM[12]	IMM[10:5]	RS2	RS1	FUNCT3	IMM[4:1]	IMM[11]	OPCODE
Bits	1	6	5	5	3	4	1	7

U Format

U	IMM[31:12]	RD	OPCODE
Bits	20	5	7

UJ Format

UJ	IMM[20]	IMM[10:1]	IMM[11]	IMM[19:12]	RD	OPCODE
Bits	1	10	1	8	5	7

Registers

In compressed RISC-V, Format CR, CI, and CSS can use any of the 32 RVI registers, but CIW, CL, CS, CA, and CB are limited to just 8 of them. The following table lists these popular registers specified by the three-bit `rs1'`,`rs2'`, and `rd'` fields of these formats, which correspond to RISC-V integer registers x8 to x15.

RVC REGISTER NUMBER	000	001	010	011	100	101	110	111
INTEGER REGISTER NUMBER	x8	x9	x10	x11	x12	x13	x14	x15

Implementation

Here is a simple template to begin with.

Input

A reference input is already provided to you in the `input.S` file. The final tests's input have the same format as the provided input except the number and contents of RVC codes.

```
.data

# Constant integer specifying the lines of RVC codes

# DO NOT MODIFY THIS VARIABLE
.globl lines_of_rvc_codes
lines_of_rvc_codes:
    .word 7
```

```
# RVC codes, 16-bits instructions mixed with 32-bits instructions
# A 16/32-bits binary number represents one line of rvc code.
# You can suppose all of the input codes are valid.

# DO NOT MODIFY THIS VARIABLE
.globl rvc_codes
rvc_codes:
    .word 0b00000000000000000000001010110011
    .half 0b0001010111111101
    .half 0b1001001010101010
    .half 0b0001010111111101
    .word 0b11111111000000101110111110111100011
    .half 0b1000010100010110
    .half 0b1000000010000010
```

Hint

- You may assume that all the machine codes are valid.
- To interact with the input file, you can use the global labels defined in `input.S`. For example, you can get the `lines_of_rvc_codes` with the following code:

```
la   a0, lines_of_rvc_codes
```

- The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary, i.e., IALIGN=16. You should be able to handle the intermixed codes, 32-bit-only codes as well as 16-bit-only codes, converting them to all-32-bit codes.
- Same as in Project 1.1, think about how the jump amount will need to be adjusted.

Output

In this project, you need to output the results in binary form. For `input.S`, the output should be:

```
00000000000000000000001010110011
11111111111101011000010110010011
00000000101000101000001010110011
11111111111101011000010110010011
11111110000001011101110011100011
00000000101000000000010100110011
00000000000000001000000001100111

Exited with error code 0
```

It's usually the duty of the supervisor (operating system) to deal with input/output and halting program execution. Venus, being a simple emulator, does not offer us such luxury, but supports a list of primitive environmental calls. Since Venus does not provide an environmental call to directly print binary, you may need to combine some of the environmental calls to achieve the above output. The following environmental calls could be helpful.

ID (A0)	NAME	DESCRIPTION
1	print_int	prints integer in a1
11	print_character	prints ASCII character in a1
17	exit2	ends the program with return code in a1

- Each line of instruction ends with a `\n` (ASCII: 10).
- A snippet of assembly of how to exit with error code 0 is already provided to you in `translator.S`. The output line 'Exited with error code 0' is neccessary. So please use ID17(exit2) environmental environmental call instead of ID10(exit) environmental call.

The command that we use to test your program's correctness is

```
diff <your_transformed_output> <reference_output>
```

You can also test your result using this command.

Execution

Make sure that `venus-jvm-latest.jar`, `translator.S` and `input.S` reside in the same directory. To run your program locally and write the output to `translator.output`, use the following command.

```
java -jar venus-jvm-latest.jar translator.S >> translator.output
```

To debug your program online, you might want to replace `.import input.S` in `translator.S` with the content of `input.S`.

Tips

- You can use any risc-v instructions as long as the venus can recognize them.
- Handwritten assembly are postfixed with extension `.S` to distinguish from compiler generated assembly `.s`
- You can learn how to output a string, int or char from Lab 3 and Lab 4.
- We will test your program using RISC-V emulator venus. Actually almost all things you need can be learnt from venus Wiki.
- Learn save and load from memory using RISC-V.
- Be careful about the calling convention, it will make life easier.
- Write comments.
- The test cases are very friendly! Don't focus too much on the edge cases, focus on the correctness on the common cases.

Submission

- You need to follow the RISC-V integer register convention and the RISC-V integer calling convention.
- You need to have **meaningful** comments not less than 25% of the total lines of code you wrote.

- A comment is defined by a sentence followed by `#`.
- We will use gitlab with the according `autolab.txt`, as in Project 1.1. From your gitlab we will only use `translator.S`. Do NOT include venus or any other quite big files into the git - you are welcome to use the git to also share your test input and other small support files with your teammate.