Project 1.1: RISC-V instructions to RVC instructions in C

Computer Architecture I ShanghaiTech University
Project 1.1 Project 1.2

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.

Getting started

Make sure you read through the entire specification before starting the project.

The whole project is split into two parts. Project 1.1 and Project 1.2. Those will be autograded separately and have their own deadlines.

You will be using gitlab to collaborate with your group partner. Autolab will use the files from gitlab. Make sure that you have access to gitlab. In the group CS110_22s_Projects you should have access to your project 1.1 project. Also, in the group CS110_22s, you should have access to the p1.1_framework.

Obtain your files

1. Clone your p1.1 repository from gitlab. You may want to change `http` to `https`.
   `git clone https://autolab.sist.shanghaitech.edu.cn/gitlab/cs110_22s_projects/p1.1_xxx_xxx.git`
   (replace xxx to your project name)
2. In the repository add a remote repo that contains the framework files:
   `git remote add framework`
   `https://autolab.sist.shanghaitech.edu.cn/gitlab/cs110_22s/p1.1_framework.git` (or change to http)
3. Go and fetch the files:
   `git fetch framework`
4. Now merge those files with your master branch:
   `git merge framework/master`
5. The rest of the git commands work as usual.

How to Autolab

1. Edit the text file `autolab.txt`. The first line has to be the name of your p1.1 project in gitlab. So `p1.1_email1_email2`.
2. The following lines have to contain a long, random secret. Commit and push to gitlab. We will test the length and randomness of this secret by running `tar -cjf size.tar.bz2 autolab.txt`.

3. When you want to run the autograder in autolab, you have to upload your autolab.txt. Autolab will clone, from gitlab, the master branch of the repo specified in the `autolab.txt` you uploaded and then continue grading only if all of these conditions are met:
    1. The `autolab.txt` you uploaded and the one in the clone repo are identical.
    2. The size of the generated `size.tar.bz2` is at least 1000B.
    3. Only the files from the framework are present in the cloned repo.

Collaborative Coding and Frequent Pushing

You have to work at this project as a team. We invite you to use all of the features of gitlab for your project, for example branches, issues, wiki, milestones, etc.

We require you to push very frequently to gitlab. In your commits we want to see how the code evolved. We do NOT want to see the working code suddenly appear - this will make us suspicious.

We also require that all group members do substantial contributions to the project. This also means that one group member should not finish the project all by himself, but distribute the work among all group members!
Gitlab has excellent tools to track that (see "Repository : Contributors"). At the end of Project 1 we will interview all group members and discuss their contributions, to see if we need to modify the score for certain group members.

When your project is done, please submit all the code including the framework to your remote GitLab repo by running the following commands.

```
$ git commit -a
$ git push origin master:master
```

Details of the files that you need to modify, and how to submit can be found in the Submission section.

So What Is This About?

In this part of the project, you will be writing an translator that translates RISC-V instructions to RISC-V compressed instructions. The RISC-V standard compressed instruction set extension (called RVC for shorthand) offers 16-bit versions of common 32-bit RISC-V instructions, thus reducing the binary code size. Note that not all RISC-V instructions can be compressed. RVC scheme offers compression to a 32-bit RISC-V instruction when:

- The immediate or address offset is small, or
- one of the registers is the zero register (x0), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

In this project this process is a part of the assembler's job - even thogh in practical systems a good compression ratio can only be achived if the compiler is aware of the compression. To simplify the task, assume each RISC-V instruction from compiler is already translated into 32-bit RISC-V binary instruction. So your job is to take an input file consisting of 32-bit RISC-V binary instructions, and output the compressed machine code. Since not all RISC-V instructions can be compressed, your output will be a mix of 32-bit and 16-bit instructions. This is also the reason why the offsets in the jumps we saw in class are scaled by two rather than four.

In general, the compression process has the following steps:

1. Parse the input 32-bit RISC-V instructions line by line.
2. Check if it can be compressed. If not, you should write the 32-bit without compression to the output file. If it can be compressed, compress it to the corresponding 16-bit RVC instruction and write it to the output file.
3. However, since some lines are compressed into 16-bits, original jump offsets need to be updated. You can use another pass to deal with it.

The Instruction Set

Remember that not all RISC-V instructions can be compressed, here we list all forms of RISC-V instructions that are compressible, which means they have their corresponding RVC instructions (if they meet the constraints listed in the table). For all other forms of RISC-V instructions, you should consider them not compressible.

RISC-V INSTR	FORMAT	RVC INSTR	FORMAT	CONSTRAINTS
add rd, rd, rs2	R	c.add	CR	rd≠x0; rs2≠x0
add rd, x0, rs2	R	c.mv	CR	rd≠x0; rs2≠x0
jalr x0, 0 (rs1)	I	c.jr	CR	rs1≠X0
jalr x1, 0 (rs1)	I	c.jalr	CR	rs1≠X0
addi rd, x0, imm	I	c.li	CI	rd≠x0
lui rd, nzimm	U	c.lui	CI	rd≠{x0,x2}; immediate is nonzero
addi rd, rd, nzimm	I	c.addi	CI	rd≠x0; immediate is nonzero
slli rd, rd, shamt	I	c.slli	CI	rd≠x0; shamt[5] must be zero
lw rd', offset (rs1')	I	c.lw	CL	
sw rs2, offset (rs1')	S	c.sw	CS	
and rd', rd', rs2'	R	c.and	CS	
or rd', rd', rs2'	R	c.or	CS	
xor rd', rd', rs2'	R	c.xor	CS	
sub rd', rd', rs2'	R	c.sub	CS	
beq rs1', x0, offset	SB	c.beqz	CB	
bne rs1', x0, offset	SB	c.bnez	CB	
srli rd', rd', shamt	I	c.srli	CB	shamt[5] must be zero
srai rd', rd', shamt	I	c.srai	CB	shamt[5] must be zero
andi rd', rd', imm	I	c.andi	CB	

RISC-V INSTR	FORMAT	RVC INSTR	FORMAT	CONSTRAINTS
jal x0, offset	UJ	c.j	CJ	
jal x1, offset	UJ	c.jal	CJ	

In addition, a detailed description of RVC instruction formats is provided for you below to help you map RVC instructions into binary code. If you understand RISC-V instruction formats taught in class, the following RVC formats should be easy to you.

CR Format

CR FORMAT	FUNCT4	RD/RS1	RS2	OPCODE
# of 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 FORMAT	FUNCT3	IMM	RD/RS1	IMM	OPCODE
# of 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 FORMAT	FUNCT3	IMM	RS1’	IMM	RD’	OPCODE
# of Bits	3	3	3	2	3	2
C.LW	010	offset[5:3]	base	offset[2|6]	dest	00

CS Format

CS-TYPE1	FUNCT3	IMM	RS1’	IMM	RS2’	OPCODE
# of Bits	3	3	3	2	3	2
C.SW	110	offset[5:3]	base	offset[2|6]	src	00

CS-TYPE2	FUNCT6	RD’/RS1’	FUNCT2	RS2’	OPCODE
# of 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
# of 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
# of 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 FORMAT	FUNCT3	JUMP TARGET	OPCODE
# of 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

In these RVC formats, you will find CL, CS and CB formats use rs1', rs2' and rd' to indicate their register fields. To save space, rs1', rs2' and rd' are restricted to 8 popular registers x8-x15. Only registers x8-x15 can be used in these formats. When translating these formats, you will need to map RISC-V register to 3-bit RVC register number:

RISC-V REGISTER NAME	X8	X9	X10	X11	X12	X13	X14	X15
RVC Register Number	000	001	010	011	100	101	110	111

Input & Output

Input: RISC-V machine code (assuming no empty lines)

Output: compressed RISC-V machine code

Samples: See files under `./test`

Testing

Manually testing

1. Run `make` to compile the code and create the translator executable.
2. To run the translator, type `./translator <input file> <output file>`. Then you can use diff to check the difference between output file and reference file: `diff <output file> <reference file>`. To see how to interpret diff results, click here.
3. Additionally, you should use Valgrind to check whether your code has any memory leaks by running: `valgrind --tool=memcheck --leak-check=full --track-origin=yes ./translator <input file> <output file>`

Using testing script

We have provided a few test cases under `test/in`. There are 7 types of test cases. Each type corresponds to one RISC-V format. For example, test cases under `test/in/rtype` will check rtype RISC-V instructions. "full" type test cases will cover instructions of mixed types. Note that these testcases are by no means comprehensive (in terms of checking range and testcase numbers).

We've also provided a simple testing script. Run `make grade`, and it will automatically check your outputs with the reference outputs and detect if there are any memory leak errors, and print out a score list to the terminal. The output files will be under `test/out`.

To add your own test cases, for example there are already 2 test cases under `test/in/full` and you want to add another test case, you should:

1. Name your test case as `input_3.s`
2. Put `input_3.s` under `test/in/full`
3. Put your desired output `ref_3.s` under `test/ref/full`
4. Modify `Makefile` under `test/`, append a number `3` in variable `full_TESTS`
5. Modify `test.py` under `test/`, change the dictionary `TESTS` according to the comment
6. Then you can run `make grade` again.

Some Things to Note

- For RISC-V jump and branch instructions, you can assume that they will not jump outside of the program.
- For all RISC-V instructions with imm field, they are not compressible when the immediate value is too large. Think carefully about the boundaries when dealing with each type of the RVC instruction. However, for simplification you don't need to consider imm boundary for jump or branch instructions.
- For c.lw and c.sw, their offsets are scaled by 4 to save the space of imm field, which is different from lw and sw. This means the least two significant bits of the offsets of lw and sw must be 0, otherwise you shouldn't compress them. Think about why. Furthermore, offsets in c.lw and c.sw are zero-extended (again different from lw and sw), which means you shouldn't compress a lw or sw instruction with negative offset.
- You may need to use some string processing functions in C standard library like strcmp, strcat, strtol or strcpy.
- You need to correctly consider ALL the edge cases mentioned in this description, otherwise you may fail some test cases.

Notes regarding grading

- The Autolab will enforce a proper amount of comments. Make sure you add proper comments - about one every four lines of code that you ADD.
- The Autolab will use `-Wpedantic -Wall -Wextra -Werror -std=c89` for compilation (removing `-g` from the Makefile in the template). You should not edit the Makefile - Autolab will replace the Makefile when testing your code.
- Memory check will be done all the testcases. If you pass the memory check on ALL testcases, you will get the point for memory check.

Submission

You should submit the same autolab.txt in your gitlab repo to to Autolab.

The directory tree of your gitlab repo should like the following:

You can leave the test folder and the Makefile. Autolab will replace them with the real testcases and the Makefile.

```
|--- src
    |      |-- compression.c
    |      |-- compression.h
    |      |-- utils.c
    |      |-- utils.h
    |--- test (optional)
    |--- translator.c
    |--- translator.h
    |--- Makefile (optional)
    |--- autolab.txt
```

Resources

- For RISC-V instructions, you can consult the RISC-V Green Sheet for details.
- For RVC instructions, you can consult The RISC-V Compressed Instruction Set Manual Version 1.9 for details. Please make a post on Piazza if you find anything inconsistent with the official manual.

Last modified: 2022-03-08