

Computer Organization Project

Sujay Deb, Tammam Tillo

Feb 14, 2023

1 Introduction and Instructions

1. This will be a group assignment in groups of 4 students. Each student in the group will be marked separately. Therefore, try to ensure that work is roughly divided equally among all the group members.
2. In this assignment, you will have to design and implement a custom assembler and a custom simulator for a given ISA.
3. You are not restricted to any programming language. However, your program must read from the specified file and write to the specified file.
4. You must use GitHub to collaborate. You must track your progress via git.
5. The automated testing infrastructure assumes that you have a working Linux-based shell. For those using Windows, you can use a VM or WSL.
6. Start the assignment early and ask the queries well in advance. Do not expect any reply on weekends and 10 PM - 6 AM on working days. Do not escalate your query to instructors directly. **Write any queries you have in the comments section.** Wait at least 24 hours for any reply to your comment. If there is no reply, then you can mail it to the respective TAs if still there is no response, then mail the TFs, and if still there is no response, then mail the instructors for clarification.
7. No last-minute deadline extensions will be considered whatsoever. This includes but is not limited to connectivity issues, one group member not working or not cooperating, etc. The duration of the deadline is sufficient enough to complete the assignment.
8. Commit your code to the repository periodically to prevent any loss of your code due to system failures or any other issues. In case of system failures of all the members of the group, your last committed code on Git Hub before the deadline will be considered for evaluation.

2 Question Description

There are a total of three questions in this assignment.

1. Designing and Implementing the assembler.
2. Designing and Implementing the simulator.
3. A bonus question based on the assembler and simulator.

We will release some test cases with the assignment so that you can test your implementations. During the evaluations, a superset of these test cases will be provided to you, on which basis your project will be graded.

3 Deadlines

The assignment consists of two deadlines:

3.1 Mid Evaluation

1. By this deadline, you must have the assembler ready.
2. You will be tested mostly on the test cases already provided to you with the assignment.
3. However, we might add some other test cases as well.
4. You will only be evaluated on the assembler. (20%)

3.2 Final Evaluation

1. By this deadline, you must have both the assembler and the simulator ready(70%).
2. You should also have completed Q3(10%).
3. You will be evaluated on a much larger set of test cases this time.
4. You will also be evaluated on the bonus question at this stage.

The mid-evaluation will be worth 20% of your final assignment grade. The final evaluation will be worth the remaining 80% of your final assignment grade. The bonus will be worth 10%, making the total 110%.

4 Grading

Q1 and Q2 are mandatory questions. In Q1, you will have to make an assembler. In Q2, you have to make a simulator for which detailed information is mentioned in the respective questions. Grading will be based on the number of test cases that your program passes. Q3 (**bonus**) is an extension of the functionality of the assembler-simulator set-up that you build.

4.1 Q1: Assembler

The test cases are divided into three sets:

1. ErrorGen: These tests are supposed to generate errors
2. simpleBin: These are simple test cases that are supposed to generate a binary.
3. hardGen: These are hard test cases that are supposed to generate a binary.

4.2 Q2: Simulator

The test cases are divided into two sets:

1. simpleBin: These are simple test cases that are supposed to generate a trace.
2. hardGen: These are hard test cases that are supposed to generate a trace.

The TA will grade the errorGen cases manually.

4.3 Q3: Bonus

In this question, you need to advance the designed assembler and simulator for the additional instructions mentioned in red color in Table 14. Furthermore, you must design one test case for each new instruction implemented. And one more test case program containing all new implemented instructions.

5 Evaluation Procedure

1. On your demo day, a compressed archive of all tests will be shared with you on Google Classroom. This archive will also include other test cases, which will not be provided to you beforehand.
2. On the day of evaluation, you must
 - (a) Prove that you are not running code written after the deadline by running “git log HEAD,” which prints the date and time of the commit pointed to by the HEAD. You must also run “git status” to show that you don’t have any uncommitted changes.
 - (b) Prove the integrity of the tests archive by computing the sha256sum of the archive. To compute the checksum, you can run “sha256sum path/to/the/archive”. The TA will then match the checksum to verify the integrity.
3. Then you can extract the archive and replace the “automatedTesting/tests” directory.
4. Then you need to execute the automated testing infrastructure, which will run all the tests and finally print your score.
5. The TA will verify the correctness of the test cases which are supposed to generate errors. An automated test-case infrastructure will be provided.

6 Plagiarism

1. Any copying of code from your peers or from the internet will invoke the institute’s Plagiarism policy.
2. Provide proper references if you’re taking your code from another resource. If the said code is the main part of the assignment, You will be awarded 0 marks.
3. If you are found indulging in any bad practice to circumvent the above-mentioned evaluation procedure, you will be awarded 0 marks, and the institute plagiarism policy will be applied.

7 Assignment Description

In this project, you will implement a subset of RV32I (RISC-V 32-bit integer) instruction set. RISC-V, an open-source ISA, is increasingly used for open-source hardware development.

7.1 ISA Explanation

RISC V is a load-store type ISA. This implies that the data is brought to the register before being processed. Hence, even the instructions that perform memory access, have the memory address present in the register. Table 1 shows the various instructions format used in RV32I ISA. The following definitions should be used while reading it.

1. **rs**: Source register.
2. **rd**: Destination register.
3. **rt**: Temporary register.
4. **imm**: Immediate.
5. **PC**: Program Counter.
6. **sp**: Stack Pointer.

Further details on each of the instruction types can also be found on:

<https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html>

Some available simulators can be accessed at:

<https://ascslab.org/research/briscv/simulator/simulator.html>

<https://github.com/TheThirdOne/rars>

<https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/>

7.2 Instruction and Register Encoding

Each instruction in RISC V can be represented uniquely in 32 bits. For each format type tables are listed to convert the instructions from assembly to binary.

Table 17 shows the binary encoding of the various registers used by RISC V. Note that the register x0 always contains the value 0. The value is not impacted by any writes done on it. While writing the program code, it is recommended to use saved registers as much as possible.

| | | | | | | Format |
|---------------------------|---------|---------|---------|---------------|--------|--------|
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | |
| funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type |
| [31:20] | | [19:15] | [14:12] | [11:7] | [6:0] | |
| imm[11 : 0] | | rs1 | funct3 | rd | opcode | I-type |
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | |
| imm[11 : 5] | rs2 | rs1 | funct3 | imm[4 : 0] | opcode | S-type |
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | |
| imm[12 10 : 5] | rs2 | rs1 | funct3 | imm[4 : 1 11] | opcode | B-type |
| [31:12] | | | | [11:7] | [6:0] | |
| imm[31:12] | | | | rd | opcode | U-type |
| [31:12] | | | | [11:7] | [6:0] | |
| imm[20 10 : 1 11 19 : 12] | | | | rd | opcode | J-type |

Table 1: Type of Instructions in RISC-V

Note: The abbreviation "sext" stands for sign extension.

7.2.1 R Type Instructions

| Base Instruction(s) | Explanation |
|--|---|
| add rd, rs1, rs2 | rd = sext(rs1) + sext(rs2) (Overflow are ignored) |
| sub rd, x0, rs | rd = 0 - rs. (Two's complement) |
| sub rd, rs1, rs2 | rd = signed(rs1) - signed(rs2) |
| slt rd, rs1, rs2 | rd = 1. If sext(rs1) < sext(rs2) |
| sltu rd, rs1, rs2 | rd = 1. If unsigned(rs1) < unsigned(rs2) |
| xor rd, rs1, rs2 | rd = rs1 ⊕ rs2 (Bitwise Exor) |
| sll rd, rs1, rs2 | rd = rs1 << unsigned(rs2[4:0]) |
| Left shift rs1 by the value in lower 5 bits of rs2. | |
| srl rd, rs1, rs2 | rd = rs1 >> unsigned(rs2[4:0]) |
| Right shift rs1 by the value in lower 5 bits of rs2. | |
| or rd, rs1, rs2 | rd = rs1 rs2 (Bitwise logical or.) |
| and rd, rs1, rs2 | rd = rs1 & rs2 (Bitwise logical and.) |

Table 2: Register type (R-type) instruction in RISC-V.

| | | | | | | |
|---------------|------------|------------|---------------|-----------|---------------|-------------|
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | add |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | sub |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | sll |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | slt |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | sltu |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | xor |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | srl |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | or |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | and |

Table 3: Binary encoding of Register type (R-type) instruction in RISC-V.

7.2.2 I Type Instructions

| Base Instruction(s) | Explanation |
|--|---|
| $l\{b h w d\} \text{ rd, imm}[11:0](\text{rs1})$ | $\text{rd} = \text{mem}(\text{rs1} + \text{sext}(\text{imm}[11:0]))$ |
| addi rd, rs, imm[11:0] | $\text{rd} = \text{rs} + \text{sext}(\text{imm}[11:0])$ |
| sltiu rd, rs, imm[11:0] | $\text{rd} = 1$. If $\text{unsigned}(\text{rs}) < \text{unsigned}(\text{imm})$ |
| jalr rd, x6, offset[11:0] | $\text{rd} = \text{PC} + 4$. |
| | And store(link) the return address in (rd). |
| | $\text{PC} = \text{x6} + \text{sext}(\text{imm}[11:0])$ |
| | Before jumping make the LSB=0 for PC. |

Table 4: Immediate type (I-type) instructions in RISC-V.

| | | | | | |
|--------------------|------------|---------------|-----------|---------------|-------------|
| [31:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| imm[11 : 0] | rs1 | funct3 | rd | opcode | |
| imm[11 : 0] | rs1 | 010 | rd | 0000011 | lw |
| imm[11 : 0] | rs1 | 000 | rd | 0010011 | addi |
| imm[11 : 0] | rs1 | 011 | rd | 0010011 | sltiu |
| imm[11 : 0] | rs1 | 000 | rd | 1100111 | jalr |

Table 5: Binary encoding of Immediate type (I-type) instructions in RISC-V.

7.2.3 S Type Instructions

| Base Instruction(s) | Explanation |
|---|---|
| $s\{b h w d\} \text{ rd, imm}[11:0](\text{rt})$ | $\text{mem}(\text{rt} + \text{sext}(\text{imm}[11:0])) = \text{rd}$ |

Table 6: (S-type) instructions in RISC-V.

| | | | | | | |
|--------------------|------------|------------|---------------|-------------------|---------------|-------------|
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| imm[11 : 5] | rs2 | rs1 | funct3 | imm[4 : 0] | opcode | |
| imm[11 : 5] | rs2 | rs1 | 010 | imm[4 : 0] | 0100011 | sw |

Table 7: Binary encoding of S-type instructions in RISC-V.

7.2.4 B Type Instructions

| Base Instruction(s) | Explanation |
|--------------------------|--|
| beq rs1, rs2, imm[12:1] | Branch or $PC = PC + sext(\{imm[12:1], 1'b0\})$. If $sext(rs1) == sext(rs2)$. |
| bne rs1, rs2, imm[12:1] | Branch or $PC = PC + sext(\{imm[12:1], 1'b0\})$. If $sext(rs1) \neq sext(rs2)$. |
| bge rs1, rs2, imm[12:1] | Branch or $PC = PC + sext(\{imm[12:1], 1'b0\})$. If $sext(rs1) \geq sext(rs2)$. |
| bgeu rs1, rs2, imm[12:1] | Branch or $PC = PC + signed(\{imm[12:1], 1'b0\})$. If $unsigned(rs1) \geq unsigned(rs2)$. |
| blt rs1, rs2, imm[12:1] | Branch or $PC = PC + sext(\{imm[12:1], 1'b0\})$. If $sext(rs1) < sext(rs2)$. |
| bltu rs1, rs2, imm[12:1] | Branch or $PC = PC + sext(\{imm[12:1], 1'b0\})$. If $unsigned(rs1) < unsigned(rs2)$. |

Table 8: Type of Branch type (B-type) instruction in RISC-V.

| | | | | | | |
|----------------|---------|---------|---------|---------------|---------|-------------|
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| imm[12 10 : 5] | rs2 | rs1 | funct3 | imm[4 : 1 11] | opcode | |
| imm[12 10 : 5] | rs2 | rs1 | 000 | imm[4 : 1 11] | 1100011 | beq |
| imm[12 10 : 5] | rs2 | rs1 | 001 | imm[4 : 1 11] | 1100011 | bne |
| imm[12 10 : 5] | rs2 | rs1 | 100 | imm[4 : 1 11] | 1100011 | blt |
| imm[12 10 : 5] | rs2 | rs1 | 101 | imm[4 : 1 11] | 1100011 | bge |
| imm[12 10 : 5] | rs2 | rs1 | 110 | imm[4 : 1 11] | 1100011 | bltu |
| imm[12 10 : 5] | rs2 | rs1 | 111 | imm[4 : 1 11] | 1100011 | bgeu |

Table 9: Binary encoding of Branch type (B-type) instruction in RISC-V.

7.2.5 U Type Instructions

| Base Instruction(s) | Explanation |
|----------------------|---|
| auipc rd, imm[31:12] | $rd = PC + sext(\{imm[31:12], 12'h000\})$ |
| lui rd, immediate | $rd = sext(\{imm[31:12], 12'h000\})$ |

Table 10: Unsigned type (U-type) instruction in RISC-V.

| | | | |
|------------|--------|---------|-------------|
| [31:12] | [11:7] | [6:0] | Instruction |
| imm[31:12] | rd | opcode | |
| imm[31:12] | rd | 0110111 | lui |
| imm[31:12] | rd | 0010111 | auipc |

Table 11: Binary encoding of Unsigned type (U-type) instruction in RISC-V.

7.2.6 J Type Instructions

| Base Instruction(s) | Explanation |
|---------------------|--|
| jal rd, imm[20:1] | rd = PC + 4. And store(link) the return address in (rd). PC = PC + sext({imm[20:1],1'b0}) Before jumping make the LSB=0 for PC. |

Table 12: J-type instruction in RISC-V.

| | | | |
|---------------------------|--------|---------|-------------|
| [31:12] | [11:7] | [6:0] | Instruction |
| imm[20 10 : 1 11 19 : 12] | rd | opcode | |
| imm[20 10 : 1 11 19 : 12] | rd | 0010111 | jal |

Table 13: J-type instruction in RISC-V.

7.2.7 Instructions for Bonus part

| Base Instruction(s) | Explanation |
|---------------------|---|
| mul rd,rs1,rs2 | rd = rs1 * rs2. Ignore overflow. |
| rst | Reset all registers except Program counter(PC). |
| halt | Stops the further execution. Or halt the processor. |
| rvrs rd,rs | Reverse the content of (rs) and store in (rd). |

Table 14: Bonus instructions to be implemented in the project.

| | | | | | | |
|---------|---------|---------|---------|--------|-------|------|
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | |
| | rs2 | rs1 | | rd | | mul |
| | | | | | | rst |
| | | | | | | halt |
| | | rs1 | | rd | | rvrs |

Table 15: Mnemonics for bonus instructions.

| Base Instruction(s) | Explanation |
|--------------------------|-----------------------------------|
| addi x0, x0, 0 | Perform no operation and advance. |
| beq zero,zero,0x00000000 | Virtual Halt. PC = PC. |

Table 16: Additional functions after instruction manipulation.

7.2.8 Register Encoding

| Address | Register | ABI Name | Description | Saver |
|-----------------------------|----------|----------|-----------------------------------|--------|
| 5'b0000_0 | x0 | zero | Hard-wired zero | --- |
| 5'b0000_1 | x1 | ra | Return address | Caller |
| 5'b0001_0 | x2 | sp | Stack Pointer | Callee |
| 5'b0001_1 | x3 | gp | Global Pointer | ---- |
| 5'b0010_0 | x4 | tp | Thread Pointer | ---- |
| 5'b0010_1 | x5 | t0 | Temporary/alternate link register | Caller |
| 5'b00_1{110,111} | x6-7 | t1-2 | Temporaries | Caller |
| 5'b0100_0 | x8 | s0/fp | Saved register/frame pointer | Callee |
| 5'b0100_1 | x9 | s1 | Saved Register | Callee |
| 5'b0101_0,1 | x10-11 | a0-1 | Function arguments/ return values | Caller |
| (5'b011_00-11), (5'b1000_1) | x12-17 | a2-7 | Function arguments | Caller |
| 5'b1_0010-1101 | x18-27 | s2-11 | Saved registers | Caller |
| 5'b111_00-11 | x28-31 | t3-6 | Temporaries | Caller |

Table 17: Encoding of the registers used by RISC V ISA

Note: The "_" in the above table is used just to improve the visualization. The symbol itself carries no meaning.

Additional information about caller and callee can be found at https://ocw.mit.edu/courses/6-004-computation-structures-spring-2009/9051d6b950cac5104c4be3edf9412938_MIT6_004s09_lec13.pdf.

8 Questions

8.1 Assembler

Program an assembler for the aforementioned ISA and assembly. The assembler must read the assembly program as an input text file (stdin) and must generate the binary (if there are no errors) as an output text file (stdout). If there are errors, the assembler must generate the error notifications along with the line number on which the error was encountered as an output text file (stdout). In case of multiple errors, the assembler may print any one of the errors.

The input to the assembler is a text file containing the assembly instructions. Each line of the text file may be of one of two types:

1. Empty line: Ignore these lines
2. An instruction
3. A label

8.1.1 Here is the meaning corresponding to these entities.

1. An instruction can be of the following:

The opcode from the mentioned mnemonics.

A register can be zero, ra, sp, gp, etc. as per their ABI Name.

A immediate within bounds as per specified instruction.

A label that will be utilized at *jump and branch* type instructions.

2. If an instruction is labeled. Then, the **label** must be at the beginning of the instruction. The label is followed by a colon with no space in between the label and the colon. The branch instructions in the assembly code will utilize the labels to jump to the specific location. While converting assembly into binary, the label will be converted into an immediate by subtracting the absolute instruction address (pointed out by the label) from the current instruction address (Program Counter). The arithmetic operation is signed as the jump can be upward or downward. And, the converted immediate is signed (2's complement representation) and is of 12 bits.

All the programs should terminate with the **Virtual Halt** instruction (beq zero,zero,0x00000000). Note that the immediate is signed. Here (0x00000000) represents (0) of decimal. This instruction can be used to halt the processor. The assembler should be capable of:

1. Handling all supported instructions
2. Making sure that any illegal instruction (any instruction (or instruction usage) that is not supported) results in a syntax error. In particular, you must handle:
 - (a) Typos in instruction or register name
 - (b) Illegal immediate values (more than 12 bits)
 - (c) Missing **Virtual Halt** instruction
 - (d) **Virtual Halt** not being used as the last instruction
3. The corresponding binary is generated if the code is error-free. The binary file is a text file in which each line is a 32-bit binary number written using 0s and 1s in ASCII.

Note: ABI stands for Application Binary Interface.

8.2 Assembly Instruction encoding examples

1. R-type instruction encoding.

{Instruction_code}{Space}{Destination_Register(ABI)}{,}{Source_Register1(ABI)}{,}{Source_Register2(ABI)}

Example: add s1,s2,s3

| | | | | | | |
|---------|---------|---------|---------|--------|---------|-------------|
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| funct7 | s3 | s2 | add | s1 | opcode | add |
| 0000000 | 10011 | 10010 | 000 | 01001 | 0110011 | |

2. I-type instruction encoding.

{Instruction_code}{Space}{Return_Address_Register(ABI)}{,}{Source_Register1(ABI)}{,}{Immediate}

Example: jalr ra,a5,-07

| | | | | | |
|-------------|---------|---------|--------|---------|-------------|
| [31:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| -07 | a5 | funct3 | ra | opcode | jalr |
| 11111111001 | 01111 | 000 | 00001 | 1100111 | |

{Instruction_code}{Space}{Return_Address_Register(ABI)}{,}{Source_Register1(ABI)}{,}{Immediate}

Example: lw a5,20(s1)

| | | | | | |
|--------------|---------|---------|--------|---------|-------------|
| [31:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| 20 | s1 | funct3 | a4 | opcode | lw |
| 000000010100 | 01001 | 010 | 01110 | 0000011 | |

3. S-type instruction encoding.

{Instruction_code}{Space}{Return_Address_Register(ABI)}{,}{Source_Register1(ABI)}{,}{Immediate[11:0]}

Example: sw ra,32(sp)

| | | | | | | |
|-------------|---------|---------|---------|------------|---------|-------------|
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| imm[11 : 5] | ra | sp | funct3 | imm[4 : 0] | opcode | sw |
| 0000001 | 00001 | 00010 | 010 | 00000 | 0100011 | |

4. B-type instruction encoding.

{Instruction_code}{Space}{Source_register1}{Source_register2}{Immediate[12:1]}

Example: blt a4,a5,label

Suppose label = 200

blt a4,a5,200

200 => binary(0000.0000.1100.1000)

| | | | | | | |
|----------------|---------|---------|---------|---------------|---------|-------------|
| [31:25] | [24:20] | [19:15] | [14:12] | [11:7] | [6:0] | Instruction |
| imm[12 10 : 5] | a5 | a4 | funct3 | imm[4 : 1 11] | opcode | blt |
| 0000110 | 01111 | 01101 | 100 | 01000 | 1100011 | |

5. U-type instruction encoding.

{Instruction_code}{Space}{Destination_Register}{Immediate[31:12]}

Example: auipc s2,-30

| | | | |
|----------------------|--------|---------|-------------|
| [31:12] | [11:7] | [6:0] | Instruction |
| imm[31:12] | s2 | opcode | auipc |
| 11111111111111111111 | 10010 | 0010111 | |

6. J-type instruction encoding.

{Instruction_code}{Space}{Destination_Register}{Immediate[20:1]}

Example: jal ra,label

Suppose label = -1024

jal ra,-1024

-1024 => binary(1111.1111.1100.0000.0000)

| | | | |
|---------------------------|--------|---------|-------------|
| [31:12] | [11:7] | [6:0] | Instruction |
| imm[20 10 : 1 11 19 : 12] | ra | opcode | jal |
| 11000000000111111111 | 00001 | 0010111 | |

8.3 Memory size and addressing

- Program Memory: The size of program memory is 256 bytes, and each location can store 32 bits. Eventually, we have 64 locations to store our instructions. The instruction memory ranges from {0x0000_0000, 0x0000_00FF}.
- Stack Memory: The size of stack memory is 128 bytes, and each location can store 32 bits. Eventually, we have 32 locations to stack our register values. The stack grows downwards or we need to decrement the stack address before storing any register content. The stack memory ranges from {0x0000_0100, 0x0000_017F}.
- Data Memory: The size of data memory is 128 bytes, and each location can store 32 bits. Eventually, we have 32 locations in our data memory. The data memory ranges from {0x0001_0000, 0x0001_007F}.

8.4 Assembly Program encoding example

Here an assembly program is written to evaluate whether a given number is even or odd. If even then, "0" is stored at the specified (0x0001_0000) data memory location. If odd "1" is stored at the specified (0x0001_0000) data memory location. The given number is loaded as an immediate value "21". The data memory location where the corresponding result needs to be stored is $(65536)_{10} = (0x0001_0000)_2$. In the program, a label is replaced in **jal** instruction. The immediate is calculated as the address label at store_value (0x0000_002C) minus the program counter of the **jal** instruction (0x0000_0024).

| Address | Pseudo Assembly Program | Explanation Converted Instruction |
|-------------|-------------------------------------|--|
| 0x0000_0000 | addi sp,zero,380 | Initialize the stack pointer |
| 0x0000_0004 | lui s0,65536 | Store data memory address into register. |
| 0x0000_0008 | addi s0,s0,65536 | |
| 0x0000_000C | addi s2,zero,21 | Load number to be evaluated |
| 0x0000_0010 | addi s3,zero,01 | |
| 0x0000_0014 | xor t0,s2,s3 | |
| 0x0000_0018 | add s1,zero,t0 | |
| 0x0000_001C | addi sp,sp,-8 | Decrement stack pointer before storing values |
| 0x0000_0020 | sw ra,4(sp) | Store the return address before new subroutine jump. |
| 0x0000_0024 | jal ra,store_value jal ra,08 | jump to subroutine at label store_value |
| 0x0000_0028 | beq zero,zero,0 | Virtual Halt |
| 0x0000_002C | store_value: addi sp,sp,-4 | decrement stack before storing any value |
| 0x0000_0030 | sw ra,0(sp) | Store the return address in stack. |
| 0x0000_0034 | sw s1,0(s0) | store the result in data memory. |
| 0x0000_0038 | lw ra,0(sp) | load return address back |
| 0x0000_003C | addi sp,sp,4 | Get stack pointer to original state |
| 0x0000_0040 | jal zero,0(ra) | Return from this subroutine. |

Note: The instruction **jal ra,08** is the instruction compatible to RISC-V ISA. This instruction is the converted form of **jal ra,store_value**. The labels are used for the ease of assembly programmers for subroutine jump.

8.5 Simulator

8.5.1 Format of output from Simulator

The input for the simulator is the binary file converted by the assembler. The simulator, after execution of every instruction, prints the register values in any chosen file. The simulator will print the memory stats after the execution of the mentioned **Virtual Halt** instruction.

Output format to be stored in the chosen file after execution of every instruction.

{Program.Counter}{Space}{x0}{Space}{x1}{Space}{x2}{Space}.....{Space}{x31}

The memory at the simulator side should be of size (32X32-bit). The output will have all the memory content (32 lines) printed starting from address (0x0000_0000). Output format of memory after the execution of **Virtual Halt**.

32-bit_binary_data

32-bit_binary_data

32-bit_binary_data

32-bit_binary_data

.

.

.

32-bit_binary_data

Note: Both the stats after each instruction execution and the memory stats after the virtual halt will be in a single file only. Firstly, each instruction's stats will be stored, followed by memory stats.

8.6 (Bonus) Implementing new instructions

As a part of the bonus, you need to define the instruction encoding for supporting four new instructions: `mul`, `rst`, `halt`, and `rvrs` as mentioned in Table 14, Table 15. Please take note that you can format these instructions in whatever you choose as long as the current instructions remain unchanged. You will extend the assembler and simulator designed in the last two questions to support these instructions. As these instructions are custom-defined, you would have to define the test cases for the evaluation of these instructions.

Note: If anyone finds any kind of typographical mistake, any ambiguity or any incorrect technical detail please notify us as soon as possible.
