Programming Assignment 1:
Non-Pipelined RISC-V based riscv-uconn Simulator

Due September 29, 2023 @ 11:59 PM on HuskyCT

Introduction

In this programming assignment, you will build a non-pipelined simulator implementing the RISC-V riscv-uconn Instruction Set Architecture (ISA).

First, ensure that your git repository is up-to-date by executing git pull within the cse4302 directory. This will create a new pa1 directory in the repository root that contains the materials for this programming assignment.

Next, navigate to the assembler and make it again. Periodically, the assembler may be updated with new functionality, so it is important to ensure it is up to date between assignments.

The following is a brief description of the relevant materials:

- `src/`: Simulator source code
- `unittests/`: Simulator unit tests (test programs)
- `README.md`: Simulator and unit test build instructions
- `assemble_all.sh`: Bash script to automatically assemble the tests in `unittests/`

There are several source code files in the `src` directory, but you will only modify `sim_stages.c` for this programming assignment; you may not modify any other files in this directory.

The objective of this programming assignment is to modify `sim_stages.c` to implement a fully functional 5-stage non-pipelined (5 cycles per instruction) CPU simulator for the riscv-uconn ISA described in this document. The following sections provide a detailed description of the simulator implementation and riscv-uconn ISA as well as other helpful information.

To receive full credit for this assignment, your simulator implementation must be fully functional and correct for all 13 unit tests in the `unittests` directory. Instructions for assembling and running the unit tests can be found in `README.md`. You are encouraged to write and test your own unit tests, but they will not contribute to your grade for PA1.

When you have completed the programming assignment, submit your `sim_stages.c` file via HuskyCT by the posted deadline.

The remaining sections in this document are as follows:

- Section 1 describes the riscv-uconn assembler. This section is most relevant to those writing their own riscv-uconn assembly programs (such as unit tests).
- Section 2 describes the unit tests for this assignment.
- Section 3 describes the simulator structure.
- Section 4 describes the riscv-uconn ISA that must be implemented for this programming assignment.
- Section 5 provides helpful debugging tips.
1 The riscv-uconn Assembler

The riscv-uconn assembler is provided to you and will not be modified by you in this course. However, you will need to compile it by following the build instructions in the assembler's README.md (this was done in PA0). Instructions for using the assembler are also available in the README.md file. Note that a bash script assemble_all.sh is provided to you. Running this script will create a directory assembled_tests, and populate it with the assembled unit tests found in the unitests directory.

1.1 Assembler Labels

The assembler converts instructions to machine code. The assembler directives .text and .data direct the assembler to the start of instruction and data memory respectively. For example, instructions following .text are converted into 32-bit machine code starting at address 0. The .data assembler directive identifies the start of data memory. Each data word (defined with the .word) following the directive will be loaded into memory starting at address 256. For example, the third word after .data will have a memory address of 258. The specific syntax for each supported instruction is given in the later sections.

2 Unit Tests

In addition to the provided test cases within the unitests directory, you are also encouraged to create your own unit tests. Like those already created for you, the file extension is *.asm and is assembled in the same way as the others. Make sure that your unit tests terminate, and that your unit test is inside the unitests directory if you wish to assemble it using the assemble_all.sh script.

When writing unit tests, you use either the register itself (i.e., x0, x1, etc) or the register ABI name (i.e., zero, ra, etc). Additionally, immediate can be represented in both decimal and hexadecimal form. The assembler knows how to interpret either.

3 Simulator Structure

The simulator source code is located in the src directory. sim_core.c contains the simulator initialization functions and the main simulation loop as well as the machine’s registers and memory. sim_stages.c contains the functions corresponding to the individual CPU stages that you will implement for this programming assignment. You may only modify sim_stages.c.

sim_core.c contains the simulator’s entry point main(), initialization function initialize(), main simulation loop process_instructions(), registers, and memory.

main() simply invokes the initialization function and main simulation loop, and prints state information (committed instructions, simulated cycles, register contents, memory contents, etc.) after the simulation terminates.

initialize() clears the machine’s registers and memory, and loads the assembled *.out file into the machine’s memory beginning with the .text (code) section. Each row (instruction) in the *.out file is read one by one and loaded into memory starting at address 0. The row containing 11111111111111111111111111111111 indicates the end of the code section, and is not loaded into memory. The following rows contain the data section and are loaded into memory starting at address 256.

process_instructions() contains the main simulation loop responsible for executing instructions. The simulation loop invokes the functions corresponding to the 5 CPU stages (fetch, decode, execute, memory, and writeback) and handles the passing of state information between stages. Additionally, the simulator current program counter, pc, is updated with the next program counter, pc_n, which is used to fetch the next instruction. The simulation loop also checks for the simulation termination condition: that is, when an instruction has written a 1 to the x0 register, such as in addi x0, x0, 1, the simulator terminates. Do
note that the termination condition will not trigger until you properly implement the CPU stages!

The implementations for the CPU stages (fetch(), decode(), execute(), memory_stage(), and writeback()) are in sim_stages.c. The fetch() function is provided to you, and you are not allowed to modify it. When an instruction is fetched, i) the instruction located at memory[pc/4] and ii) the instruction address, i.e. the value of pc, is stored inside a State structure. Then, the simulator calls advance_pc(4), which sets pc_n = pc + 4 (Note: It is possible for control flow instructions to overwrite the value of pc_n in later stages when a branch/jump is taken. By default, conditional control flow instructions predict “branch not taken”, and will only overwrite the value of pc_n when the branch conditional evaluates to true). Finally, the State structure is returned and and gets forwarded to the input of decode(). The output of decode() is then forwarded to execute(), and so on and so forth. **You will implement the decode(), execute(), memory Stage(), and writeback() functions for this assignment.** The implementation details of every instruction for each stage is provided in the following section.

State information is passed between CPU stages using the State structure. The State structures contains dynamic information about each instruction. **You must ensure the State structure is correctly populated in each stage.** The definition of the State structure can be found in sim_core.h, and is described in Figure 3.1.

<table>
<thead>
<tr>
<th>Struct Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst</td>
<td>fetched instruction</td>
</tr>
<tr>
<td>inst_addr</td>
<td>address of instruction</td>
</tr>
<tr>
<td>opcode</td>
<td>opcode field</td>
</tr>
<tr>
<td>funct3</td>
<td>3-bit function field</td>
</tr>
<tr>
<td>funct7</td>
<td>7-bit function field</td>
</tr>
<tr>
<td>rd</td>
<td>destination register specifier</td>
</tr>
<tr>
<td>rs1</td>
<td>source 1 register specifier</td>
</tr>
<tr>
<td>rs2</td>
<td>source 2 register specifier</td>
</tr>
<tr>
<td>imm</td>
<td>immediate value</td>
</tr>
<tr>
<td>mem_buffer</td>
<td>memory data for LW/SW instructions</td>
</tr>
<tr>
<td>br_addr</td>
<td>target address for B-Type instructions</td>
</tr>
<tr>
<td>link_addr</td>
<td>return address for JAL/JALR instructions</td>
</tr>
<tr>
<td>alu_in1</td>
<td>first ALU operand</td>
</tr>
<tr>
<td>alu_in2</td>
<td>second ALU operand</td>
</tr>
<tr>
<td>alu_out</td>
<td>ALU output</td>
</tr>
</tbody>
</table>

**Figure 3.1:** Fields of the State struct. The fields contain information about the decoded instruction, ALU operands, and other (micro)architectural state.

sim_stages.c also provides the advance_pc() function, and should not be modified.

The machine’s thirty two 32-bit registers are stored in the registers[] array. Register indices and their corresponding names are specified in the following section (see table 1). The machine’s memory (both instructions and data) is stored in the memory[] array. Each element of memory[] corresponds to a single word (4 bytes, or 32-bits). More details regarding the memory model are provided in the following section.
4 The \texttt{riscv-uconn} Instruction Set Architecture

4.1 Memory and Execution Model

Memory

\texttt{riscv-uconn} memory is partitioned into instructions and data, and its total size is limited to 16,384 addresses. A word (4 bytes, or 32-bits) is stored at each memory address, leading to a total memory capacity of 65,538 bytes. The machine only supports word addressable memory.

Instructions reside in the first 256 locations of memory, starting from address 0. Each instruction is one word. A total of 256 instructions (1,024 bytes) can be stored in memory. Each instruction word is read from right to left.

Data resides in addresses 256 through 16,383. Each address contains a single word of data.

Execution

The machine's program counter register (\texttt{pc}) initially points at address 0, and addresses the first instruction word (4 bytes). The next instruction word is at address 4, and so on and so forth. The index into the memory[] array is always computed by dividing \texttt{pc} by 4. For example, if the \texttt{pc} is 32, the index containing the corresponding instruction word is calculated as $32/4 = 8$. Most instructions increment the program counter by 4 bytes, as is performed in the \texttt{fetch()} stage by calling \texttt{advance_pc(4)}, reading instructions from memory[] sequentially. However, control flow instructions (\texttt{BNE}, \texttt{BEQ}, \texttt{BLT}, \texttt{BGE}, \texttt{JAL}, and \texttt{JALR}), may modify the next program counter to a non-sequential instruction address. Make sure to pay special attention to where control-flow instructions resolve. \textbf{It should be noted that \texttt{pc} is never to be directly written to.} \texttt{pc\_n}, which holds the program counter value of the next instruction, is modified instead. At the end of the cycle, $\texttt{pc} \leftarrow \texttt{pc\_n}$, so that the next instruction is correctly fetched.

4.2 Registers

The machine implements a RISC-V ISA with 32 registers, where each register is 32-bits (or one word). These registers are named by the ISA as seen in

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
Register Number & ABI Name & Description \\
\hline
x0 & zero & hardwired 0x00000000 \\
1x & ra & return address \\
x2 & sp & stack pointer \\
x3 & gp & global pointer \\
x4 & tp & thread pointer \\
x5-7 & t0-2 & temporary registers \\
x8-9 & a0-1 & saved registers \\
x10-11 & a0-1 & function arguments / return values \\
x12-17 & a2-7 & function arguments \\
x18-27 & s2-11 & saved registers \\
x28-31 & t3-6 & temporary registers \\
\hline
\end{tabular}
\end{table}

\textbf{Table 1: riscv-uconn registers and their purposes.}

The \texttt{zero} register normally contains a value of 0, but can be set to 1 to trigger program termination. The mapping from register indices (0–31) to register names can be found in \texttt{register_map.c}.

4.3 Instructions

The \texttt{riscv-uconn} instruction format is the same as the standard RISC-V 32-bit integer instruction set. A 32-bit instruction is broken down into six formats: R-Type (figure 4.1), I-Type (figure 4.2), S-Type (figure 4.3),
The 7-bit opcode field, 3-bit funct3, and 7-bit funct7 fields are used to differentiate between instruction types. The 5-bit rd, rs1, and rs2 fields encode the indices of the destination, source 1, and source 2 registers, respectively. The imm field encodes the immediate/offset value used by various instruction types. The size and encoding of the immediate varies depending on the instruction type. In the following sections, \( i \) represents the numerical value of the immediate, with the subscript representing the specific bit of the immediate. For example, \( i_5 \) means the 5th bit of the immediate.

The binary values for each opcode, funct3, and funct7 field for different instructions and types are already defined in instruction_map.h. You will also find some helpful #define statements to help you with decoding in this file. **You are expected to modify sim_stages.c to support all of the following instructions.**

The implementation details of each instruction for each CPU stage are specified in the following sections:

1. **R-Type Instructions:** ADD, SUB, AND, OR, XOR, SLT, SLL, SRL
2. **I-Type Instructions:** LW, JALR, ADDI, ANDI, ORI, XORI, SLTI, SLLI, SRLI
3. **S-Type Instructions:** SW
4. **B-Type Instructions:** BEQ, BNE, BLT, BGE
5. **U-Type Instructions:** LUI
6. **J-Type Instructions:** JAL
4.3.1 R-Type Instructions

ADD

Full Name: Addition
Description: Add the contents of two registers and store the result in a register.
Assembler Syntax: add rd, rs1, rs2
Operation: rd = rs1 + rs2
Decode Stage: Extract 7-bit opcode, 3-bit funct3, and 7-bit funct7 fields to identify this operation. Extract 5-bit rd, rs1, and rs2 register specifiers. registers[rs1] and registers[rs2] are read as the two ALU operands.
Execute Stage: The two ALU operands are added using the + operator to compute the output value.
Memory Stage: Nothing is done for this instruction.
Writeback Stage: registers[rd] is updated with the output value.
Encoding:

<table>
<thead>
<tr>
<th>Bit</th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>19</td>
<td>15</td>
</tr>
</tbody>
</table>

SUB

Full Name: Subtraction
Description: Subtract the contents of two registers and store the result in a register.
Assembler Syntax: sub rd, rs1, rs2
Operation: rd = rs1 − rs2
Implementation is the same as ADD except that the − operator is used to compute the output value in the execute stage.
Encoding:

<table>
<thead>
<tr>
<th>Bit</th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>19</td>
<td>15</td>
</tr>
</tbody>
</table>

AND

Full Name: Bitwise AND
Description: Bitwise AND the contents of two registers and store the result in a register.
Assembler Syntax: and rd, rs1, rs2
Operation: rd = rs1 & rs2
Implementation is the same as ADD except that the & operator is used to compute the output value in the execute stage.
Encoding:

<table>
<thead>
<tr>
<th>Bit</th>
<th>rs2</th>
<th>rs1</th>
<th>111</th>
<th>rd</th>
<th>0110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>19</td>
<td>15</td>
</tr>
</tbody>
</table>
**OR**

**Full Name:** Bitwise OR  
**Description:** Bitwise OR the contents of two registers and store the result in a register.  
**Assembler Syntax:** `or rd, rs1, rs2`  
**Operation:** `rd = rs1 | rs2`  
Implementation is the same as ADD except that the `|` operator is used to compute the output value in the execute stage.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**XOR**

**Full Name:** Bitwise Exclusive OR  
**Description:** Bitwise XOR the contents of two registers and store the result in a register.  
**Assembler Syntax:** `xor rd, rs1, rs2`  
**Operation:** `rd = rs1 ^ rs2`  
Implementation is the same as ADD except that the `^` operator is used to compute the output value in the execute stage.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SLT**

**Full Name:** Set on Less Than  
**Description:** If `rs1` is less `rs2`, then `rd` is set to one. Otherwise, `rd` is set to zero.  
**Assembler Syntax:** `slt rd, rs1, rs2`  
**Operation:** `rd = rs1 < rs2`  
Implementation is the same as ADD except that the `<` operator is used to compute the output value in the execute stage.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SLL**

**Full Name:** Shift Left Logical  
**Description:** Shift the contents of `rs1` left by `rs2` positions and store the result in `rd` (Note: whenever `rs2 > 32`, shifting has the same effect as when `rs2 = 32`).  
**Assembler Syntax:** `sll rd, rs1, rs2`  
**Operation:** `rd = rs1 << rs2`  
Implementation is the same as ADD except that the `<<` operator is used to compute the output value in the execute stage.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SRL

Full Name: Shift Right Logical
Description: Shift the contents of rs1 right by rs2 positions and store the result in rd (Note: whenever rs2 > 32, shifting has the same effect as when rs2 = 32).

Assembler Syntax: srl rd, rs1, rs2
Operation: rd = rs1 >> rs2
Implementation is the same as ADD except that the >> operator is used to compute the output value in the execute stage.

Encoding:

<table>
<thead>
<tr>
<th>Bit</th>
<th>0000000</th>
<th>rs2</th>
<th>rs1</th>
<th>101</th>
<th>rd</th>
<th>0110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

4.3.2 I-Type Instructions

LW

Full Name: Load Word
Description: A word is loaded into a register from the specified memory address.

Assembler Syntax: lw rd, offset(rs1)
Operation: alu_out = rs1 + offset
rd = memory[alu_out]

Decode Stage: Extract 7-bit opcode and 3-bit func3 fields to identify this operation. Extract 5-bit rd and rs1 register specifiers. registers[rs1] is read to determine the base for the address calculation and is set as the first ALU operand. The second ALU operand, the address offset, is calculated by sign-extending the 12-bit immediate field to the register length (32-bits).

Execute Stage: The memory address is calculated by adding the two ALU operands with the + operator and is stored in alu_out.

Memory Stage: mem_buffer is set to memory[alu_out].
Writeback Stage: mem_buffer is stored in registers[rd].

Encoding:

<table>
<thead>
<tr>
<th>Bit</th>
<th>11i10i9i8i7i6i5i4i3i2i1i0</th>
<th>rs1</th>
<th>010</th>
<th>rd</th>
<th>0000011</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>
JALR

Full Name: Jump and Link Register
Description: Jumps unconditionally to the calculated address. Stores the return address in rd if it is not the zero register (x0), otherwise a jump with no link is performed. Resolves in the decode stage.
Assembler Syntax: jalr rd, rs1, offset
Operation: pc_n = rs1 + offset
Decode Stage: rd = inst_addr + 4 if rd is not the zero register (x0); else perform no writeback
Execute Stage: Nothing is done for this instruction.
Memory Stage: Nothing is done for this instruction.
Writeback Stage: registers[rd] is set to link_addr if rd \neq 0. Otherwise, does nothing.
Encoding:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_11</td>
<td>i_10</td>
<td>i_9</td>
<td>i_18</td>
<td>i_17</td>
<td>i_16</td>
<td>i_15</td>
<td>i_14</td>
<td>i_13</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>000</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADDI

Full Name: Add Immediate
Description: Add the contents of a register to a sign-extended immediate value and store the result in a register.
Assembler Syntax: addi rd, rs1, immediate
Operation: rd = rs1 + immediate
Decode Stage: Extract 7-bit opcode and 3-bit func3 fields to identify this operation. Extract 5-bit rd and rs1 register specifiers. Extract and sign-extend the 12-bit immediate to the register length (32-bits). pc_n is set to registers[rs1] + (sign-extended) immediate.
Execute Stage: The output value is computed as the addition of the two operands using the + operator.
Memory Stage: Nothing is done for this instruction.
Writeback Stage: registers[rd] is updated with the output value.
Encoding:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_11</td>
<td>i_10</td>
<td>i_9</td>
<td>i_8</td>
<td>i_17</td>
<td>i_16</td>
<td>i_15</td>
<td>i_14</td>
<td>i_13</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>000</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The encoding for a “nop” (no operation, or an instruction that does nothing) is represented by the instruction addi x0, x0, 0 which has no side effects on the register and memory state of the machine.
ANDI

Full Name: Bitwise AND Immediate
Description: Bitwise AND the contents of a register to a sign-extended immediate value and store the result in a register.
Assembler Syntax: andi rd, rs1, immediate
Operation: rd = rs1 & immediate
Implementation is the same as ADDI except that the & operator is used to compute the output value in the execute stage.
Encoding:

<table>
<thead>
<tr>
<th>i11i10i9i8i7i6i5i4i3i2i1i0</th>
<th>rs1</th>
<th>111</th>
<th>rd</th>
<th>0010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

ORI

Full Name: Bitwise OR Immediate
Description: Bitwise OR the contents of a register to a sign-extended immediate value and store the result in a register.
Assembler Syntax: ori rd, rs1, immediate
Operation: rd = rs1 | immediate
Implementation is the same as ADDI except that the | operator is used to compute the output value in the execute stage.
Encoding:

<table>
<thead>
<tr>
<th>i11i10i9i8i7i6i5i4i3i2i1i0</th>
<th>rs1</th>
<th>110</th>
<th>rd</th>
<th>0010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

XORI

Full Name: Bitwise Exclusive OR Immediate
Description: Bitwise XOR the contents of a register to a sign-extended immediate value and store the result in a register.
Assembler Syntax: xori rd, rs1, immediate
Operation: rd = rs1 ^ immediate
Implementation is the same as ADDI except that the ^ operator is used to compute the output value in the execute stage.
Encoding:

<table>
<thead>
<tr>
<th>i11i10i9i8i7i6i5i4i3i2i1i0</th>
<th>rs1</th>
<th>100</th>
<th>rd</th>
<th>0010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

SLTI

Full Name: Set on Less Than Immediate
Description: If rs1 is less than the sign-extended immediate value, rd is set to one. Otherwise, rd is set to zero.
Assembler Syntax: slti rd, rs1, immediate
Operation: rd = rs1 < immediate
Implementation is the same as ADDI except that the < operator is used to compute the output value in the execute stage.
Encoding:
SLLI

**Full Name:** Shift Left Logical Immediate

**Description:** Shift the contents of rs1 left by the least 5-bit immediate positions and store the result in rd.

**Assembler Syntax:** `slli rd, rs1, immediate`

**Operation:** `rd = rs1 << immediate`

Implementation is the same as ADDI except that the `<<` operator is used to compute the output value in the execute stage.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SRLI

**Full Name:** Shift Right Logical Immediate

**Description:** Shift the contents of rs1 right by the least 5-bit immediate positions and store the result in rd.

**Assembler Syntax:** `srli rd, rs1, immediate`

**Operation:** `rd = rs1 >> immediate`

Implementation is the same as ADDI except that the `>>` operator is used to compute the output value in the execute stage.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3.3 S-Type Instructions

SW

**Full Name:** Store Word

**Description:** The contents of a register is stored at the specified memory address.

**Assembler Syntax:** `sw rs2, offset(rs1)`

**Operation:**

- `alu_out = rs1 + offset`
- `memory[alu_out] = rs2`

**Decode Stage:** Extract 7-bit `opcode` and 3-bit `funct3` fields to identify this operation. Extract 5-bit `rs1` and `rs2` register specifiers. `registers[rs1]` is read to determine the base for the address calculation and is set as the first ALU operand. The second ALU operand, the address offset, is calculated by sign-extending the 12-bit immediate field to the register length (32-bits). `mem_buffer` is set to `registers[rs2]` to propagate the value to be stored to memory.

**Execute Stage:** The memory address is calculated by adding the two ALU operands with the `+` operator and is stored in `alu_out`.

**Memory Stage:** `mem_buffer` is written to `memory[alu_out]`.

**Writeback Stage:** Nothing is done for this instruction.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11
4.3.4 B-Type Instructions

**BEQ**

**Full Name:** Branch on Equal

**Description:** Branches if the contents of two registers are equal. Resolves in the execute stage.

**Assembler Syntax:** `beq rs1, rs2, offset`

**Operation:**
- If `rs1 = rs2`, then `pc_n = inst_addr + offset`; else do nothing (normal `pc_n = pc + 4` is carried out).

**Decode Stage:**
- Extract 7-bit opcode and 3-bit funct3 fields to identify this operation. Extract 5-bit `rs1` and `rs2` register specifiers. `registers[rs1]` and `registers[rs2]` are read as the two ALU operands. Extract and sign-extend the 12-bit immediate field. `br_addr` is set to `inst_addr + (sign-extended) immediate` (Note: because instructions are always aligned to at least 2-bytes in RISC-V, the least significant bit (i0) is always zero, and is not encoded).

**Execute Stage:**
- The two ALU operands are compared to determine if the branch will be taken or not.
- If the branch is taken (i.e., the two ALU operands are equal), then `pc_n` is set to `br_addr`, overwriting the `advance_pc(4)` call performed in the fetch stage. Otherwise, nothing is done here.

**Memory Stage:** Nothing is done for this instruction.

**Writeback Stage:** Nothing is done for this instruction.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i</td>
<td>i</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**BNE**

**Full Name:** Branch on Not Equal

**Description:** Branches if the contents of two registers are not equal. Resolves in the execute stage.

**Assembler Syntax:** `bne rs1, rs2, offset`

**Operation:**
- If `rs1 ≠ rs2`, then `pc_n = inst_addr + offset`; else do nothing (normal `pc_n = pc + 4` is carried out).

**Implementation** is the same as BEQ except that the branch condition tests for non-equality in the execute stage.

**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i</td>
<td>i</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
BLT

Full Name: Branch on Less Than
Description: Branches if the contents of one register is less than another. Resolves in the execute stage.
Assembler Syntax: `blt rs1, rs2, offset`
Operation: if `rs1 < rs2`, then `pc_n = inst_addr + offset`; else do nothing (normal `pc_n = pc + 4` is carried out).
Implementation is the same as BEQ except that the branch condition tests for a less than condition in the execute stage.

Encoding:

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>1100011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

BGE

Full Name: Branch on Greater Than or Equal To
Description: Branches if the contents of one register is greater than or equal to another. Resolves in the execute stage.
Assembler Syntax: `bge rs1, rs2, offset`
Operation: if `rs1 ≥ rs2`, then `pc_n = inst_addr + offset`; else do nothing (normal `pc_n = pc + 4` is carried out).
Implementation is the same as BEQ except that the branch condition tests for a greater than or equal to condition in the execute stage.

Encoding:

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>1100011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

Note 1: The 16-bit offset in B-Type instructions is calculated by the assembler using the difference between the 32-bit address of the instruction and the address of the label. For example, if a program wants to loop back four instructions, then the offset will be stored as `0xFFFFFFFFF0` or `-16`. The branch address will then be calculated as `pc+(-16)`, which will allow the program to loop back four instructions. Similarly, if the program wants to loop forward 4 instructions, then the offset will be stored as `0x10` or `16`. When the condition is checked depending on if the instruction is BEQ, BNE, BLT, or BGE, the branch address will be calculated as `pc+16`.

Note 2: For the B-Type instructions, the offset immediate is not encoded contiguously or with the bits in-order. You must extract the proper bit-fields and re-order the bits within the instruction according to the encoding indices to correctly calculate the branch address.
4.3.5 U-Type Instructions

LUI

<table>
<thead>
<tr>
<th>Full Name:</th>
<th>Load Upper Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The 20-bit immediate value is extracted and stored in the upper 20 bits of a register. The lower 12 bits are cleared to zero.</td>
</tr>
<tr>
<td>Assembler Syntax:</td>
<td>lui rd, immediate</td>
</tr>
<tr>
<td>Operation:</td>
<td>rd = immediate[31:12]</td>
</tr>
<tr>
<td>Decode Stage:</td>
<td>Extract 7-bit opcode field to identify this operation.</td>
</tr>
<tr>
<td>Execute Stage:</td>
<td>The upper 20 bits of the instruction are set to the upper 20 bits of the ALU output. The lower 12 bits of the ALU are set to 0.</td>
</tr>
<tr>
<td>Writeback Stage:</td>
<td>registers[rd] is updated with the output value.</td>
</tr>
<tr>
<td>Encoding:</td>
<td>![Encoding Table]</td>
</tr>
</tbody>
</table>

4.3.6 J-Type Instructions

JAL

<table>
<thead>
<tr>
<th>Full Name:</th>
<th>Jump and Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Jumps unconditionally to the calculated address. Stores the return address in rd if it is not the zero register (x0), otherwise a jump with no link is performed. Resolves in the decode stage.</td>
</tr>
<tr>
<td>Assembler Syntax:</td>
<td>jal rd, offset</td>
</tr>
<tr>
<td>Operation:</td>
<td>pc_n = inst_addr + offset</td>
</tr>
<tr>
<td></td>
<td>rd = inst_addr + 4 if rd is not the zero register (x0); else perform no writeback.</td>
</tr>
<tr>
<td>Decode Stage:</td>
<td>Extract 7-bit opcode field to identify this operation. link_addr is set to inst_addr+4. Extract and sign-extend the 20-bit immediate field to the register length (32-bits). pc_n is set to inst_addr + (sign-extended) immediate (Note: because instructions are always aligned to at least 2-bytes in RISC-V, the least significant bit (i_0) is always zero, and is not encoded).</td>
</tr>
<tr>
<td>Execute Stage:</td>
<td>Nothing is done for this instruction.</td>
</tr>
<tr>
<td>Memory Stage:</td>
<td>Nothing is done for this instruction.</td>
</tr>
<tr>
<td>Writeback Stage:</td>
<td>registers[rd] is set to link_addr if rd (\neq 0). Otherwise, do nothing.</td>
</tr>
<tr>
<td>Encoding:</td>
<td>![Encoding Table]</td>
</tr>
</tbody>
</table>

Note: For the J-Type instructions, the offset immediate is not encoded with the bits in-order. You must must re-order the bits within the instruction according to the encoding indices to correctly calculate the target address.

5 Debugging

You have several options for debugging your simulator implementation. printf statements can be added anywhere in sim_stages.c so long as they are properly gated by the debug flag variable at the top of the file. util.c provides some helpful debugging functions that output the
register \texttt{(rdump())} and memory \texttt{(mdump())} contents. Several of these debugging functions are used in the core simulator implementation by default. You may use these functions so long as they are properly gated by the \texttt{debug} flag.

A facility called pipe trace is added to the simulator to support visualization of instruction processing across cycles. The file \texttt{pipe_trace.txt} will be created whenever the simulator is executed. The \texttt{pipe_trace} flag variable in \texttt{sim_stages.c} toggles whether pipe tracing is enabled or not. You may insert debugging information into the pipe trace file so long as it is properly gated with the \texttt{debug} flag. Refer to \texttt{sim_core.c} for examples of writing to the pipe trace. Additionally, when using the \texttt{pipe_trace} output, you may configure the display mode of the trace. You may toggle between register numbers and ABI names, and between hexadecimal and decimal values for immediate. This is toggled by changing the \texttt{pipe_trace_mode} variable in \texttt{sim_stages.c} according to the below table:

<table>
<thead>
<tr>
<th>Value of \texttt{pipe_trace_mode}</th>
<th>Register Display Setting</th>
<th>Immediate Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Register No.</td>
<td>Hexidecimal</td>
</tr>
<tr>
<td>1</td>
<td>ABI Name</td>
<td>Hexidecimal</td>
</tr>
<tr>
<td>2</td>
<td>Register No.</td>
<td>Decimal</td>
</tr>
<tr>
<td>3</td>
<td>ABI Name</td>
<td>Decimal</td>
</tr>
</tbody>
</table>

Finally, you may use the GDB debugger (see the guide here: \url{https://condor.depaul.edu/glancast/373class/docs/gdb.html}). You can run the simulator with a unit test under GDB using the following:

\begin{verbatim}
$ gdb ./simulator unit_test.out
\end{verbatim}