Introduction

In this programming assignment, you will build a non-pipelined simulator implementing the MIPS-based riscy-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. The following is a brief description of the relevant materials:

```
src/       Simulator source code
unitests/  Simulator unit tests (test programs)
README.md  Simulator and unit test build instructions
```

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 riscy-uconn ISA described in this document. The following sections provide a detailed description of the simulator implementation and riscy-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 11 unit tests in the `unitests` 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.

When you have completed the programming assignment, submit your `sim_stages.c` file via HuskyCT by the posted deadline. To receive credit for this assignment, you must also schedule a 10–15 minute code review meeting with the TA. You have up until 2 weeks after the HuskyCT deadline to complete the code review.

The remaining sections in this document are as follows:

- Section 1 describes the simulator structure.
- Section 2 describes the riscy-uconn ISA that must be implemented for this programming assignment.
- Section 3 describes the riscy-uconn assembler. This section is most relevant to those writing their own riscy-uconn assembly programs (such as unit tests).
- Section 4 provides helpful debugging tips.
1 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 (e.g., nop.out, beq_test1.out, etc.) 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 remaining rows contain the data section and are loaded into memory starting at address 2,048.

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. The simulation loop also checks for the simulation termination condition: that is, when an instruction has written a 1 to the $0 register, such as in addi $0, $0, 1. 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(), and writeback()) are in sim_stages.c. The fetch() function is provided to you, and you are not allowed to modify it. fetch() returns the instruction from memory address PC/4 and forwards it to the decode() function. The output of decode() is then forwarded to execute(), and so on and so forth. You will implement the decode(), execute(), memory(), 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 a 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 1.1.

sim_stages.c also provides the advance_pc() function. You may not modify it, but you are free to use it in your implementations.

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. The machine’s memory 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.
### Struct Member Description

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst</td>
<td>fetched instruction</td>
</tr>
<tr>
<td>opcode</td>
<td>opcode field</td>
</tr>
<tr>
<td>func</td>
<td>function field</td>
</tr>
<tr>
<td>rs</td>
<td>rs register specifier</td>
</tr>
<tr>
<td>rt</td>
<td>rt register specifier</td>
</tr>
<tr>
<td>rd</td>
<td>rd register specifier</td>
</tr>
<tr>
<td>sa</td>
<td>shift amount (shamt)</td>
</tr>
<tr>
<td>imm</td>
<td>immediate value</td>
</tr>
<tr>
<td>mem_flag</td>
<td>flag indicating LW/SW instruction</td>
</tr>
<tr>
<td>mem_addr</td>
<td>memory address for LW/SW instruction</td>
</tr>
<tr>
<td>jmp_out_31</td>
<td>return address for JAL instruction</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>
<tr>
<td>mem_out</td>
<td>memory output</td>
</tr>
</tbody>
</table>

Figure 1.1: Fields of the State struct. The fields contain information about the decoded instruction, ALU operands, and other (micro)architectural state.

## 2 The riscy-uconn Instruction Set Architecture

### 2.1 Memory and Execution Model

#### Memory

riscy-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 2,048 locations of memory, starting from address 0. Each instruction is one word. A total of 2,048 instructions (8,192 bytes) can be stored in memory. Each instruction word is read from right to left.

Data resides in addresses 2,048 through 16,383. Each address contains a single word of data.

#### Execution

The machine’s program counter register (PC) initially points at address 0, and addresses the first instruction word (4 bytes). The next instruction word is stored at address 1, and so on and so forth. The address of the instruction memory is always computed by dividing PC by 4. For example, if the PC is calculated to be 32, the memory address containing the corresponding instruction word is calculated as $32 / 4 = 8$. Most instructions increment the program counter by 4 bytes. However, control flow instructions, such as BNE, BEQ, J, JAL and JR, may modify the PC to a non-sequential instruction address.

### 2.2 Registers

The machine implements a MIPS-like ISA with 32 registers, where each register is 32-bits (or one word). These registers are named by the ISA as $zero$, $at$, $v0$-$v1$, $a0$-$a3$, $t0$-$t9$, $s0$-$s7$, $k0$-$k1$, $gp$, $sp$, $fp$, and $ra$. The $zero$ 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 register_map.c.
2.3 Instructions

The riscy-uconn instruction format is similar to MIPS. A 32-bit instruction is broken down into three formats: R-Type (Figure 2.1), I-Type (Figure 2.2), and J-Type (Figure 2.3).

<table>
<thead>
<tr>
<th>OP</th>
<th>RS</th>
<th>RT</th>
<th>RD</th>
<th>SHAMT</th>
<th>FUNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Bit 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: R-Type instruction format

<table>
<thead>
<tr>
<th>OP</th>
<th>RS</th>
<th>RT</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>Bit 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2: I-Type instruction format

<table>
<thead>
<tr>
<th>OP</th>
<th>ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Bit 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.3: J-Type instruction format

The 6-bit OP and FUNC fields are used to differentiate between instruction types. The 5-bit RS, RT and RD fields encode the indices of the source and/or destination registers used by several instruction. The specific values of these fields for an instruction are referred to as $s$, $t$, and $d$ in the following sections. The 6-bit SHAMT field encodes the shift amount for the shift instructions SRL and SLL. The 16-bit IMM field encodes the immediate value used by I-Type instructions. Finally, the 26-bit ADDR field encodes the program counter address for J-Type instructions (unconditional jumps).

The instructions supported by the riscy-uconn machine are specified in instruction_map.h. You are expected to modify sim_stages.c to support all of the instructions identified in this file.

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

2.3.1 R-Type Instructions

ADD

**Full Name:** Addition

**Description:** Add the contents of two registers and store the result in a register.

**Assembler Syntax:** add $d, $s, $t

**Operation:** $d = s + t

**Decode Stage:** Extract 6-bit OP and FUNC fields to identify this operation. Extract 5-bit RS, RT, and RD register specifiers. registers[RS] and registers[RT] are read as the two ALU operands. The PC is advanced by 4 bytes using the advance_pc(4) function call.

**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>000000</th>
<th>ss</th>
<th>ttttt</th>
<th>ddd</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Bit 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 $d, $s, $t  
**Operation:** $d = $s − $t  

Implementation is the same as ADD except that the − operator is used to compute the output value in the execute stage.

**Encoding:**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>000000</td>
<td>sssss</td>
<td>ttttt</td>
<td>dddddd</td>
<td>00000</td>
<td>100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 $d, $s, $t  
**Operation:** $d = $s & $t  

Implementation is the same as ADD except that the & operator is used to compute the output value in the execute stage.

**Encoding:**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>000000</td>
<td>sssss</td>
<td>ttttt</td>
<td>dddddd</td>
<td>00000</td>
<td>100100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 $d, $s, $t  
**Operation:** $d = $s | $t  

Implementation is the same as ADD except that the | operator is used to compute the output value in the execute stage.

**Encoding:**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>000000</td>
<td>sssss</td>
<td>ttttt</td>
<td>dddddd</td>
<td>00000</td>
<td>100101</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 a register left by the shift amount and store the result in a register. Zeroes are shifted in.  
**Assembler Syntax:** `sll $d, $t, h`  
**Operation:** $d = $t $<< h  
Implementation is the same as ADD except that the 5-bit SHAMT field is extracted in the decode stage, and the output value is computed by shifting the contents of the RT register left by SHAMT using the $<<$ operator in the execute stage.  
**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>00000</td>
<td>ttttt</td>
<td>ddddd</td>
<td>hhhhh</td>
<td>00000</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) represents the instruction SLL $0, $0, 0, which has no side effects on the register and memory state of the machine.

### SRL

**Full Name:** Shift Right Logical  
**Description:** Shift the contents of a register right by the shift amount and store the result in a register. Zeroes are shifted in.  
**Assembler Syntax:** `srl $d, $t, h`  
**Operation:** $d = $t $>> h  
Implementation is the same as SRL except that the output value is computed by shifting the contents of the RT register right by SHAMT using the $>>$ operator in the execute stage.  
**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>00000</td>
<td>ttttt</td>
<td>ddddd</td>
<td>hhhhh</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SLT

**Full Name:** Set on Less Than  
**Description:** If $s$ is less than $t$, $d$ is set to one. $d$ is set to zero otherwise.  
**Assembler Syntax:** `slt $d, $s, $t`  
**Operation:** if $s < t$, then $d = 1$, else $d = 0$  
Implementation is the same as ADD except that an if-else check is used to compare the contents of the RS register and the RT register using the $<$ operator to compute the output value in the execute stage.  
**Encoding:**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>sssss</td>
<td>ttttt</td>
<td>ddddd</td>
<td>00000</td>
<td>101010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
JR

**Full Name:** Jump Register
**Description:** Jump unconditionally to the address stored in a register.
**Assembler Syntax:** `jr $s`
**Operation:** PC = $s

**Decode Stage:** Extract 6-bit OP and FUNC fields to identify this operation. Extract 5-bit RS register specifier. `registers[RS]` is read to determine the jump address. The PC is set directly to `registers[RS]`.

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

**Encoding:**
```
  00000  sssss  00000  00000  00000  00000  001000
```

**Note:** `jr` is generally used in combination with `jal` to call and return from a procedure call.

### 2.3.2 I-Type Instructions

**LW**

**Full Name:** Load Word
**Description:** A word is loaded into a register from the specified memory address.
**Assembler Syntax:** `lw $t, offset($s)`
**Operation:** `mem_addr = $s + offset`

**Decode Stage:** Extract 6-bit OP field to identify this operation. Extract 5-bit RS and RT register specifiers. `registers[RS]` is read to determine the base for the address calculation. Extract 16-bit IMM field to determine the offset for address calculation. `mem_flag` is set for later stages. The PC is advanced by 4 bytes using the `advance_pc(4)` function call.

**Execute Stage:** The memory address is calculated using the `+` operator and stored in `mem_addr`. The address is calculated by sign-extending the 16-bit offset to the register length (32-bits), and then adding `registers[RS]` to the sign-extended offset.

**Memory Stage:** `mem_out` is set to `memory[mem_addr]`.
**Writeback Stage:** `mem_out` is stored in `registers[RT]`.

**Encoding:**
```
  100011  sssss  ttttt  iiiiiiiiiiiiiii
```
SW

**Full Name:** Store Word  
**Description:** The contents of a register is stored at the specified memory address.  
**Assembler Syntax:** sw $t, offset($s)  
**Operation:** mem_addr = $s + offset  
**Decode Stage:** Extract 6-bit OP field to identify this operation. Extract 5-bit RS and RT register specifiers. registers[RS] is read to determine the base for the address calculation. Extract 16-bit IMM field to determine the offset for address calculation. mem_flag is set for later stages. The PC is advanced by 4 bytes using the advance_pc(4) function call.  
**Execute Stage:** The memory address is calculated using the + operator and stored in mem_addr. The address is calculated by sign-extending the 16-bit offset to the register length (32-bits), and then adding registers[RS] to the sign-extended offset.  
**Memory Stage:** registers[RT] is written to memory[mem_addr].  
**Writeback Stage:** Nothing is done for this instruction.  
**Encoding:**

<table>
<thead>
<tr>
<th>101011</th>
<th>sssss</th>
<th>ttttt</th>
<th>iiiiiiiiiiiiiii</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ANDI

**Full Name:** Bitwise AND Immediate  
**Description:** Bitwise AND the contents of a register with a sign-extended immediate value and store the result in a register.  
**Assembler Syntax:** andi $t, $s, imm  
**Operation:** $t = $s & imm  
**Decode Stage:** Extract 6-bit OP field to identify this operation. Extract 5-bit RS and RT register specifiers. Extract 16-bit IMM field. registers[RS] is read as the first ALU operand. The second ALU operand is calculated by sign-extending the IMM field to the register length (32-bits). The PC is advanced by 4 bytes using the advance_pc(4) function call.  
**Execute Stage:** The output value is computed as the bitwise AND of the two operands using the & operator.  
**Memory Stage:** Nothing is done for this instruction.  
**Writeback Stage:** registers[RT] is updated with the output value.  
**Encoding:**

<table>
<thead>
<tr>
<th>001100</th>
<th>sssss</th>
<th>ttttt</th>
<th>iiiiiiiiiiiiiii</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ADDI

**Full Name:** Addition Immediate

**Description:** Add the contents of a register to a sign-extended immediate value and store the result in a register.

**Assembler Syntax:** addi $t, $s, imm

**Operation:** $t = $s + imm

Implementation is the same as ANDI except that the + operator is used to compute the output value in the execute stage.

**Encoding:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>iiiiiiiiiiiiiii</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>sssss</td>
<td>ttttt</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

ORI

**Full Name:** Bitwise OR Immediate

**Description:** Bitwise OR the contents of a register with a sign-extended immediate value and store the result in a register.

**Assembler Syntax:** ori $t, $s, imm

**Operation:** $t = $s | imm

Implementation is the same as ANDI except that the | operator is used to compute the output value in the execute stage.

**Encoding:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>iiiiiiiiiiiiiii</th>
</tr>
</thead>
<tbody>
<tr>
<td>001101</td>
<td>sssss</td>
<td>ttttt</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

SLTI

**Full Name:** Set on Less Than Immediate

**Description:** If $s$ is less than sign-extended immediate value, $t$ is set to one. $t$ is set to zero otherwise.

**Assembler Syntax:** slti $t, $s, imm

**Operation:** if $s < imm$, then $t = 1$, else $t = 0$

Implementation is the same as ANDI except that an if-else check is used to compare the contents of the RS register and the sign-extended IMM field using the < operator to compute the output value in the execute stage.

**Encoding:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>iiiiiiiiiiiiiii</th>
</tr>
</thead>
<tbody>
<tr>
<td>001101</td>
<td>sssss</td>
<td>ttttt</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
LUI

**Full Name:** Load Upper Immediate

**Description:** The immediate value is shifted left by 16 bits and stored in a register. The lower 16 bits are cleared.

**Assembler Syntax:** `lui $t, imm`

**Operation:** `$t = imm << 16`

**Decode Stage:** Extract 6-bit **OP** field to identify this operation. Extract 5-bit **RT** register specifier. Extract 16-bit **IMM** field. The PC is advanced by 4 bytes using the `advance_pc(4)` function call.

**Execute Stage:** The 16-bit **IMM** value is shifted left by 16 bits to form a 32-bit output value whose lower 16 bits are cleared.

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

**Writeback Stage:** `registers[RT]` is updated with the output value.

**Encoding:**

<table>
<thead>
<tr>
<th>001111</th>
<th>00000</th>
<th>ttttt</th>
<th>iiiiiiiiiiiiiii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>26</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>Bit 0</td>
<td></td>
</tr>
</tbody>
</table>

BEQ

**Full Name:** Branch on Equal

**Description:** Branches if the contents of two registers are equal.

**Assembler Syntax:** `beq $s, $t, offset`

**Operation:** `if $s == $t, then pc = pc + 4 + (offset); else pc = advance_pc(4)`

**Decode Stage:** Extract 6-bit **OP** field to identify this operation. Extract 5-bit **RS** and **RT** register specifiers. Extract 16-bit **IMM** field. `registers[RS]` and `registers[RT]` are compared to determine if the branch will be taken or not. If the branch is taken (i.e., the two register values are equal), then the PC is advanced to `PC+4+(sign-extended IMM)`. Otherwise, the PC is advanced by 4 bytes using the `advance_pc(4)` function call.

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

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

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

**Encoding:**

<table>
<thead>
<tr>
<th>000100</th>
<th>sssss</th>
<th>ttttt</th>
<th>iiiiiiiiiiiiiii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31</td>
<td>26</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>Bit 0</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The 16-bit offset in this instruction is calculated by the assembler using the difference between the 32-bit address of the instruction following the BEQ (address of `PC+4` due to MIPS semantics), and the address of the label. For example, if a program wants to loop back seven instructions from BEQ, then the offset will be stored as `0xffffffff0` or `-32`. When `BEQ` tests positive for `$s == $t`, the new PC will be calculated as `PC+4-32` or `PC-28`, which will allow the program to loop back seven instructions (refer to `fetch()`). Similarly, if the program wants to loop forward seven instructions then the offset will be stored as `0x18` or `24`. When the BEQ tests positive for `$s == $t`, the new PC will be calculated as `PC+4+24` or `PC+28`, which will allow the program to loop forward seven instructions.
**BNE**

- **Full Name:** Branch on Not Equal
- **Description:** Branches if the contents of two registers are not equal.
- **Assembler Syntax:** `bne $s, $t, offset`
- **Operation:** if $s != $t, then pc = pc + 4 + (offset); else pc = advance_pc(4)

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

**Encoding:**
```
+---+---+---+---+-----+
|   | 26| 25| 21| 20  | 16  | 15  | iiii iiii iiii iiii iiii iiii iiii |
| 000101 | ss | sss | tttt | tttt | tttt |
```

**2.3.3 J-Type Instructions**

**J**

- **Full Name:** Jump
- **Description:** Jumps to the calculated address.
- **Assembler Syntax:** `j target`
- **Operation:** PC = 26-bit target address appended with six upper zero bits
- **Decode Stage:** Extract 6-bit OP field to identify this operation. The PC is set to the lower 26 bits of the instruction. The remaining bits are cleared.
- **Execute Stage:** Nothing is done for this instruction.
- **Memory Stage:** Nothing is done for this instruction.
- **Writeback Stage:** Nothing is done for this instruction.

**Encoding:**
```
+---+---+---+---+-----+
|   | 26| 25| 21| Bit 0 
| 000010 | iiii iiii iiii iiii iiii iiii iiii iiii |
```

**Note:** The target address will never use more than 11 lower bits since the instruction memory has a limit of 2,048 instructions.

**JAL**

- **Full Name:** Jump and Link
- **Description:** Jumps to the calculated address, and stores the return address in register $31 ($ra).
- **Assembler Syntax:** `jal target`
- **Operation:** $31 = PC + 4
- **Dec. Stage:** PC = 26-bit target address appended with six upper zero bits
- **Dec. Stage:** Extract 6-bit OP field to identify this operation. PC+4 is stored in jmp_out_31. The PC is set to the lower 26 bits of the instruction. The remaining bits are cleared.
- **Exec. Stage:** Nothing is done for this instruction.
- **Mem. Stage:** Nothing is done for this instruction.
- **Wr. Stage:** registers[31] is set to jmp_out_31.

**Encoding:**
```
+---+---+---+---+-----+
|   | 26| 25| 21| Bit 0 
| 000011 | iiii iiii iiii iiii iiii iiii iiii iiii |
```

11
3 The *riscy-uconn* Assembler

The *riscy-uconn* assembler is provided to you and will not be modified 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.

3.1 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 2,048. For example, the third word after .data will have a memory address of 2,050.

4 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 (rdump()) and memory (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 debug flag.

A facility called pipe trace is added to the simulator to support visualization of instruction processing across cycles. The file pipe_trace.txt will be created whenever the simulator is executed. The pipe_trace flag variable in 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 debug flag. Refer to sim_core.c for examples of writing to the pipe trace.

Finally, you may use the GDB debugger. You can invoke the simulator with a specified unit test under GDB with the following shell command:

```
$ gdb ./simulator unit_test.out
```

GDB is a complex tool with powerful functionality, but a complete guide on using it is beyond the scope of this course. A guide covering GDB functionality relevant to this course can be found on the following webpage:

[https://condor.depaul.edu/glancast/373class/docs/gdb.html](https://condor.depaul.edu/glancast/373class/docs/gdb.html)