

Programming Assignment 1: Non-pipelined Simulator

Due October 3, 2022 (Monday) @ 11:59 PM on HuskyCT

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:

<code>src/</code>	Simulator source code
<code>unittests/</code>	Simulator unit tests (test programs)
<code>README.md</code>	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 `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 (see section 5), 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. These review meetings will be held independently from office hours, and you will hear more about scheduling appointments as the due date approaches.**

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.
- Section 5 describes the procedure of implementing your own unit tests written in assembly code.

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_stage()`, 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_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 a `State` structure. The `State` structure 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
<code>inst</code>	fetched instruction
<code>opcode</code>	opcode field
<code>func</code>	function field
<code>rs</code>	rs register specifier
<code>rt</code>	rt register specifier
<code>rd</code>	rd register specifier
<code>sa</code>	shift amount (shamt)
<code>imm</code>	immediate value
<code>mem_addr</code>	memory address for LW/SW instruction
<code>mem_out</code>	memory output
<code>jmp_out_31</code>	return address for JAL instruction
<code>br_addr</code>	target address for BEQ/BNE instruction
<code>alu_in1</code>	first ALU operand
<code>alu_in2</code>	second ALU operand
<code>alu_out</code>	ALU output

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. Make sure to pay special attention to where control-flow instructions resolve.

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`, `$v0-1`, `$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).

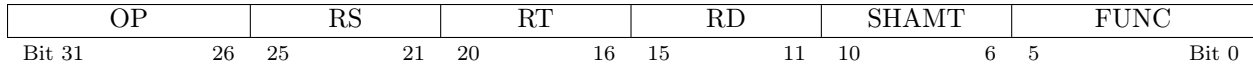


Figure 2.1: R-Type instruction format

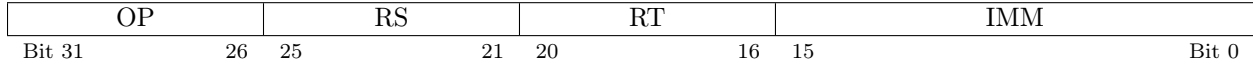


Figure 2.2: I-Type instruction format



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).

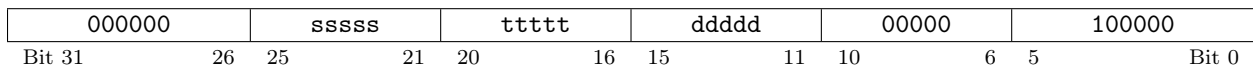
The instructions supported by the *riscy-uconn* machine are specified in `instruction_map.h`. **You are expected to modify `sim_stages.c` to support all of the instructions identified in this file.**

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:	<code>add \$d, \$s, \$t</code>
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. <code>registers[RS]</code> and <code>registers[RT]</code> are read as the two ALU operands. The PC is advanced by 4 bytes using the <code>advance_pc(4)</code> function call.
Execute Stage:	The two ALU operands are added using the <code>+</code> operator to compute the output value.
Memory Stage:	Nothing is done for this instruction.
Writeback Stage:	<code>registers[RD]</code> is updated with the output value.
Encoding:	



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:

000000		sssss		ttttt		dddd		00000		100001	
Bit 31	26	25	21	20	16	15	11	10	6	5	Bit 0

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:

000000		sssss		ttttt		dddd		00000		100100	
Bit 31	26	25	21	20	16	15	11	10	6	5	Bit 0

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:

000000		sssss		ttttt		dddd		00000		100101	
Bit 31	26	25	21	20	16	15	11	10	6	5	Bit 0

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:

000000		00000		ttttt		ddddd		hhhhh		000000	
Bit 31	26	25	21	20	16	15	11	10	6	5	Bit 0

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:

000000		00000		ttttt		ddddd		hhhhh		000010	
Bit 31	26	25	21	20	16	15	11	10	6	5	Bit 0

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:

000000		sssss		ttttt		ddddd		00000		101010	
Bit 31	26	25	21	20	16	15	11	10	6	5	Bit 0

JR

Full Name:	Jump Register
Description:	Jump unconditionally to the address stored in a register. Resolves in the decode stage.
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:	

000000		sssss		00000		00000		00000		001000	
Bit 31	26	25	21	20	16	15	11	10	6	5	Bit 0

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 \$t = memory[mem_addr]
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		iiiiiiiiiiiiiiii	
Bit 31	26	25	21	20	16	15	Bit 0

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 memory[mem_addr] = \$t
Decode Stage:	Extract 6-bit OP field to identify this operation. Extract 5-bit RS and RT register specifiers. <code>registers[RS]</code> is read to determine the base for the address calculation. Extract 16-bit IMM field to determine the offset for address calculation. <code>mem_flag</code> is set for later stages. <code>mem_out</code> is set to <code>registers[RT]</code> to propagate the value to be stored to memory. The PC is advanced by 4 bytes using the <code>advance_pc(4)</code> function call.
Execute Stage:	The memory address is calculated using the + operator and stored in <code>mem_addr</code> . The address is calculated by sign-extending the 16-bit offset to the register length (32-bits), and then adding <code>registers[RS]</code> to the sign-extended offset.
Memory Stage:	<code>mem_out</code> is written to <code>memory[mem_addr]</code> .
Writeback Stage:	Nothing is done for this instruction.
Encoding:	

101011		sssss		ttttt		iiiiiiiiiiiiiiii	
Bit 31	26	25	21	20	16	15	Bit 0

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. <code>registers[RS]</code> 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 <code>advance_pc(4)</code> 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:	<code>registers[RT]</code> is updated with the output value.
Encoding:	

001100		sssss		ttttt		iiiiiiiiiiiiiiii	
Bit 31	26	25	21	20	16	15	Bit 0

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: $\$t = \$s + \text{imm}$

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

Encoding:

001000	sssss	ttttt	iiiiiiiiiiiiiiii
Bit 31	26 25	21 20	16 15 Bit 0

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: $\$t = \$s | \text{imm}$

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

Encoding:

001101	sssss	ttttt	iiiiiiiiiiiiiiii
Bit 31	26 25	21 20	16 15 Bit 0

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 $\$s < \text{imm}$, 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:

001010	sssss	ttttt	iiiiiiiiiiiiiiii
Bit 31	26 25	21 20	16 15 Bit 0

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 <code>advance_pc(4)</code> 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:	<code>registers[RT]</code> is updated with the output value.
Encoding:	

001111		00000		ttttt		iiiiiiiiiiiiiiii	
Bit 31	26	25	21	20	16	15	Bit 0

BEQ

Full Name:	Branch on Equal
Description:	Branches if the contents of two registers are equal. Resolves in the execute stage.
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. <code>registers[RS]</code> and <code>registers[RT]</code> are read as the two ALU operands. Extract 16-bit IMM field. First, <code>br_addr</code> is set to PC+4+(sign-extended IMM). Then, the PC is advanced by 4 bytes using the <code>advance_pc(4)</code> function call ("branch not taken" predicted).
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 the PC is set to <code>br_addr</code> . Otherwise, nothing is done here.
Memory Stage:	Nothing is done for this instruction.
Writeback Stage:	Nothing is done for this instruction.
Encoding:	

000100		sssss		ttttt		iiiiiiiiiiiiiiii	
Bit 31	26	25	21	20	16	15	Bit 0

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 `0xfffffe0` or -32. The branch address will then be calculated as PC+4-32 or PC-28, which will allow the program to loop back seven instructions. (refer to `fetch()`, where a PC/4 is used as the index for instruction memory). 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 branch address 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. Resolves in the execute stage.

Assembler Syntax: bne \$s, \$t, offset

Operation: if $\$s \neq \t , then $pc = pc + 4 + (\text{offset})$; else $pc = \text{advance_pc}(4)$

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

Encoding:

000101		sssss		ttttt		iiiiiiiiiiiiiiii	
Bit 31	26	25	21	20	16	15	Bit 0

2.3.3 J-Type Instructions

J

Full Name: Jump

Description: Jumps to the calculated address. Resolves in the decode stage.

Assembler Syntax: j target

Operation: $PC = 26\text{-bit target address appended with six upper zero bits}$

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:

000010		iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii															
Bit 31	26	25															Bit 0

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). Resolves in the decode stage.

Assembler Syntax: jal target

Operation: $\$31 = PC + 4$

$PC = 26\text{-bit target address appended with six upper zero bits}$

Decode 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.

Execute Stage: Nothing is done for this instruction.

Memory Stage: Nothing is done for this instruction.

Writeback Stage: `registers[31]` is set to `jmp_out_31`.

Encoding:

000011		iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii															
Bit 31	26	25															Bit 0

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>

5 Unit Tests

In addition to the provided test cases within the `unittests` 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. You should refer to section 3 to learn about what the different directives mean, and use existing unit tests as a sample to guide you. When you are doing your own debugging, you can use custom tests to try and focus on certain functionalities. Additionally, make sure that your unit tests terminate if using control flow instructions, and that your unit test is inside the `unittests` directory.

When trying to assemble your test, you must ensure that the assembly file is in the `unix` format. To do so, a tool called `dos2unix` should be used, as files in the DOS format will give you segmentation faults upon using the assembler. The tool can be installed with the following command:

```
$ sudo apt install dos2unix
```

after which, you convert the user-made test with the following command:

```
$ dos2unix my_test.asm
```

where `my_test` is whatever you named your file.

After this step, you should be able to run the assembler just like the other unit tests (refer to `README.md`) to produce the corresponding `.out` file.