CDA 3103 – Fall 2023 – Final Project

Version 1.1

Adapted from Sarah Angell

1 Introduction

In this project, you are asked to write the core part of a mini processor simulator called MySPIM using the C programming language on a Unix platform. Your MySPIM will demonstrate functions of the MIPS processor as well as the principle actions of the datapath and the control signals of a MIPS processor. The simulator should read in a file containing MIPS machine codes (in the format specified below) and simulate what the MIPS processor does cycle-by-cycle. You are required to implement the processor simulation as a single-cycle datapath. You are asked to fill in the body of several functions in the provided project file.

2 Simulator Specification

2.1 Instructions to be Simulated

The 14 instructions listed in Appendix A are to be simulated. Please refer to the tables in the MIPS Reference PowerPoint for the machine codes of these instructions. Note that you are NOT required to treat situations leading to exceptions, interrupts, or changes in the status register.

2.2 Registers

MySPIM should handle the 32 general-purpose registers.

2.3 Memory Usage

- The size of memory of MySPIM is 64 kB (Addresses range from 0x0000 to 0xFFFF).
- The system assumes that all programs start at memory location 0x4000.
- All instructions are word-aligned in the memory, i.e., the addresses of all instructions are a multiple of 4.
- The simulator (and the MIPS processor itself) treats the memory as one segment. (The division of memory into text, data, and stack segments is only done by the compiler/assembler.)
- At the start of the program, all memory locations are initialized to zero, except those specified in the ".asc" file, as shown in the provided codes.
- The memory is in *big-endian* byte order.
- The memory is in the following format: e.g. Store a 32-bit number 0xaabbccdd in the memory addresses 0x0 to 0x3.

	Mem[0]			
Address	0x0	0x1	0x2	0x3
Content	aa	bb	cc	dd

2.4 Conditions to Halt MySPIM

If one of the following situations is encountered, the global flag *Halt* is set to 1, and hence the simulation halts.

- An illegal instruction is encountered.
- Jumping to an address that is not word-aligned (being a multiple of 4).
- The address of lw or sw is not word-aligned.
- Accessing data or jumping to an address that is beyond the memory.

Note: The instructions beyond the list of instructions in Appendix A are illegal.

2.5 Format of the Input Machine Code File

MySPIM takes hexadecimal formatted machine codes, with file name ___.asc, as input. An example of a .asc file is shown below. Text after '#' on any line is treated as comments.

```
20010000 # addi $1, $0, 0
200200c8 # addi $2, $0, 200
10220003 # beq $1, $2, 3
00000020 # delay slot
20210001 # addi $1, $1, 1
00000020 # no operation
```

The simulation ends when an illegal instruction, such as 0x00000000, is encountered.

2.6 Note on Branch Addressing

The branch offset in MIPS, and hence in MySPIM, is relative to the next instruction, i.e., (PC + 4). For example:

Assembly Code	**Machine Code**			
	Opcode (6 bits)	rs (5 bits)	rt (5 bits)	offset (16 bits)
beq $1, $2, label	4	1	2	0x0001
beq $3, $4, label	4	3	4	0x0000
label: beq $5, $6, label	4	5	6	0xffff

3 Resources

3.1 Files Provided

Please download the following files from the Webcourses:

- spimcore.c
- spimcore.h
- project.c

These files contain the main program and the other supporting functions of the simulator. You are required to fill in the functions in project.c. You may also introduce new functions, but do not modify any other part of the files. **You are not allowed to modify spimcore.c and spimcore.h. All your work should be placed in project.c.** Please also avoid input and output statements as these will interrupt the test cases.

The details are described in Section 4 below.

4 The Functions to Fill In

The project is divided into 2 parts. In the first part, you are required to fill in a function ALU(...) in project.c that simulates the operations of an ALU.

- ALU(...)
  1. Implement the operations on input parameters *A* and *B* according to *ALUControl*.
  2. Output the result to *ALUresult*.
  3. Assign *Zero* to 1 if the result is zero; otherwise, assign 0.
  4. The following table shows the operations of the ALU.

ALU Control	Meaning
000	Z = A + B
001	Z = A - B
010	If A $<$ B, Z = 1; otherwise, Z = 0
011	If A $<$ B, Z = 1; otherwise, Z = 0 (A and B are unsigned integers)
100	Z = A AND B
101	Z = A XOR B
110	Z = Shift B left by 16 bits
111	Z = NOT A

In the second part, you are required to fill in 9 other functions in project.c. Each function simulates the operations of a section of the datapath. Appendix B below shows the datapath and the sections of the datapath you need to simulate color-coded to each of the 9 functions.

In spimcore.c, the function Step() is the core function of the simulator. This function invokes the 9 functions that you are required to implement to simulate the signals and data passing between the components of the datapath. **Read Step() thoroughly in order to understand the signals and data passing, in order to implement the 9 functions.**

The following shows the specifications of the 9 functions:

- instruction_fetch(...)
  1. Fetch the instruction addressed by *PC* from *Mem* and write it to *instruction*.
  2. Return 1 if a halt condition occurs; otherwise, return 0.
- instruction_partition(...)
  1. Partition *instruction* into several parts (op, r1, r2, r3, funct, offset, jsec).
  2. Read lines 41 to 47 of spimcore.c for more information.
- instruction_decode(...)
  1. Decode the instruction using the opcode (*op*).

2. Assign the values of the control signals to the variables in the structure *controls* (See spimcore.h file).
   The meanings of the values of the control signals:
   For *MemRead*, *MemWrite*, or *RegWrite*, the value 1 means that enabled, 0 means that disabled, 2 means "don't care."
   For *RegDst*, *Jump*, *Branch*, *MemtoReg*, or *ALUSrc*, the value 0 or 1 indicates the selected path of the multiplexer; 2 means "don't care."
   The following table shows the meaning of the values of *ALUOp*.

Value (binary)	Meaning
`000`	ALU will do addition or "don't care"
`001`	ALU will do subtraction
`010`	ALU will do "set on less than" operation
`011`	ALU will do "set on less than unsigned" operation
`100`	ALU will do "AND" operation
`101`	ALU will do "XOR" operation
`110`	ALU will shift left *extended_value* by 16 bits
`111`	The instruction is an R-type instruction

3. Return 1 if a halt condition occurs; otherwise, return 0.

- read_register(...)
  1. Read the registers addressed by *r1* and *r2* from *Reg*, and write the read values to *data1* and *data2*, respectively.
- sign_extend(...)
  1. Assign the sign-extended value of *offset* to *extended_value*.
- ALU_operations(...)
  1. Apply ALU operations on *data1*, and *data2* or *extended_value* (determined by *ALUSrc*).
  2. The operation performed is based on *ALUOp* and *funct*.
  3. Apply the function ALU(...).
  4. Output the result to *ALUresult*.
  5. Return 1 if a halt condition occurs; otherwise, return 0.
- rw_memory(...)
  1. Use the value of *MemWrite* or *MemRead* to determine if a memory write operation or memory read operation (or neither) is occurring.
  2. If reading from memory, read the content of the memory location addressed by *ALUresult* to *memdata*.
  3. If writing to memory, write the value of *data2* to the memory location addressed by *ALUresult*.
  4. Return 1 if a halt condition occurs; otherwise, return 0.
- write_register(...)
  1. Write the data (*ALUresult* or *memdata*) to a register (*Reg*) addressed by *r2* or *r3*.
- PC_update(...)
  1. Update the program counter (PC).

The file spimcore.h is the header file which contains the definition of a structure storing the control signals and the prototypes of the above 10 functions.

NOTE: You should avoid input and output operations in project.c. These operations are handled by spimcore.c.

5 Operation of the Simulator

The files spimcore.c and project.c should be compiled together. Here is an example of how that may be done in a Unix environment. First compile:

```
$ gcc -o spimcore spimcore.c project.c
```

After compilation, to begin the simulation, you can type the following command in Unix (replace <filename> with the correct name of the input file on your system):

```
$ ./spimcore <filename>.asc
```

The command prompt "`cmd:`" should appear. The simulation works with the following commands (both lowercase and uppercase letters are accepted):

Command	Mnemonic	Description
`r`	"Register"	Display register contents
`m`	"Memory"	Display memory contents
`s`	"Step"	Attempt to run the instruction located at the current PC
`c`	"Continue"	Attempt to run **all** instructions, beginning with PC
`h`	"Halt"	Check to see if the simulation has halted
`p`	"Print"	Prints a copy of the input file
`g`	"Controls"	Display the most recent control signals
`q`	"Quit"	Terminate the simulation

6 Submission Guidelines

Submit project.c online through Webcourses.

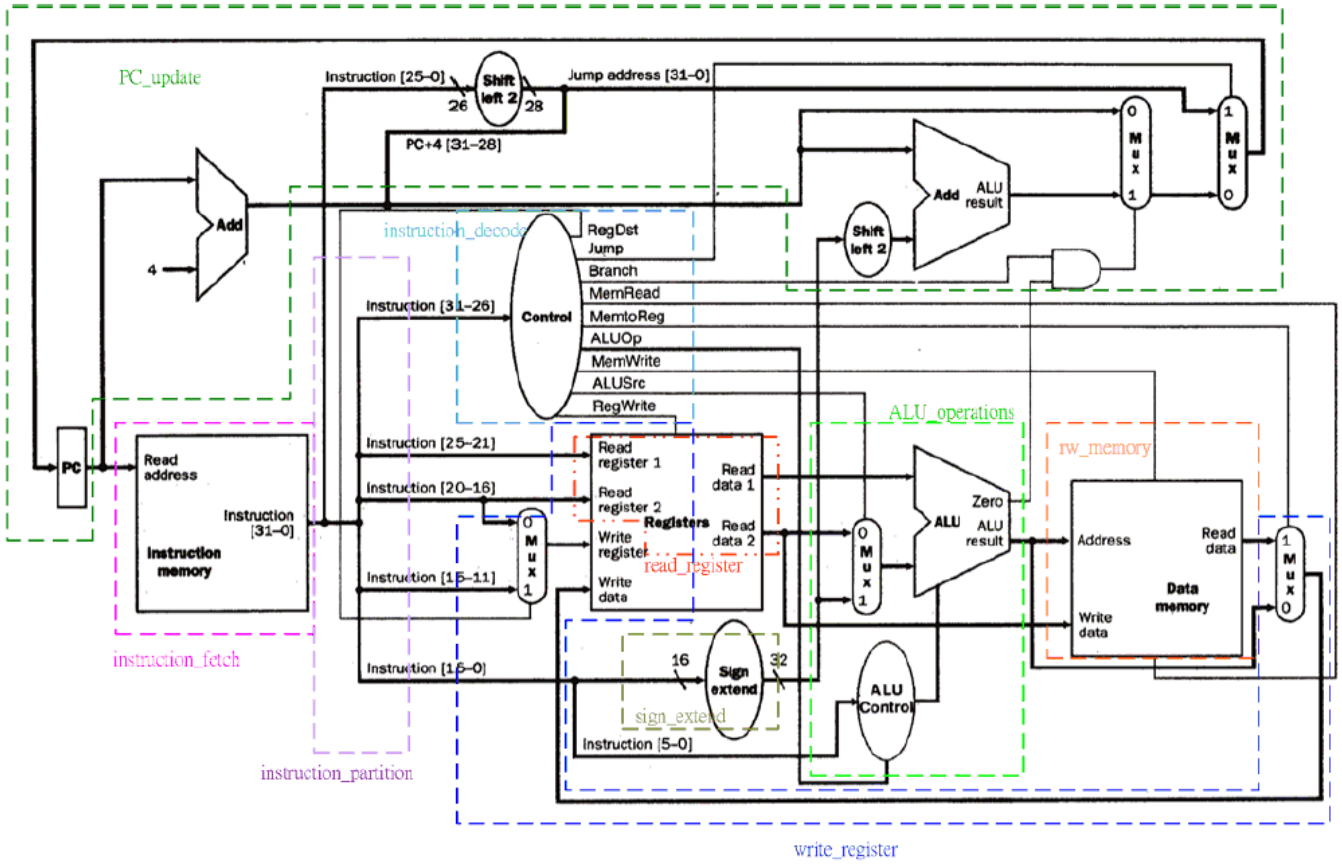
You are only required to submit project.c. No additional report to summarize your work is required. Therefore, you should provide any explanations via comments in your project.c file for potential partial credit.

Make sure that your program can be compiled and used with the commands in Section 5 on Eustis. Programs that do not compile on Eustis will be penalized a minimum of 20 points.

A Instructions to be Implemented

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	3 operands; overflow detected
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	3 operands; overflow detected
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	+ constant; overflow detected
Logic	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	3 operands; logical AND
	xor	xor $s1, $s2, $s3	$s1 = $s2 ^ $s3	3 operands; logical XOR
Data Transfer	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	word from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	loads constant in upper 16 bits
Conditional Branch	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) goto PC + 4 + 100	equal test; PC relative branch
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1 else $s1 = 0	compare less than; two's complement
	set on less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1 else $s1 = 0	compare < constant; two's compliment
	set on less than unsigned	sltu $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1 else $s1 = 0	compare less than; natural number
	set on less than immediate unsigned	sltiu $s1, $s2, 100	if ($s2 < 100) $s1 = 1 else $s1 = 0	compare < constant; natural number
Unconditional Branch	jump	j 2500	goto 10000	jump to target address

B Single-Cycle Datapath to be Implemented



C Frequently Asked Questions

Q. Is memory supposed to be an array of words, or an array of bytes?

As in, an array of words:
mem[0] = 32-bit word
mem[1] = 32-bit word
mem[2] = 32-bit word
...

Or, an array of bytes:
mem[0] = 8-bit byte
mem[1] = 8-bit byte
mem[2] = 8-bit byte
mem[3] = 8-bit byte
mem[4] = 8-bit byte (the first byte of a new word)

In the case of an array of bytes, if we want to read in an instruction from memory, we would need to read it in four 8-bit chunks.

A. Functions that require it are instruction_fetch and rw_memory and the array Mem is passed as an argument.

The little table from the first page of the project guidelines gives the answer:

Notice the "Mem[0]" in the first row, so the memory goes like this:
Mem[0] = 0x0000 - 0x0003
Mem[1] = 0x0004 - 0x0007
...
Mem[16383] = 0xFFFC - 0xFFFF (65532 - 65536)

Therefore, memory is an array of words.

It also says that all programs start at memory location 0x4000 (see definition of PCINIT in spimcore.c) which corresponds to Mem[4096] (4096 = 0x1000).

The trick is to test for word alignment while it is still in address form, then reference the proper index in the Mem array by shifting the address right twice.

Q. What are the inputs and outputs of the program?

A. Most of the functionality of this simulator is provided in spimcore.c. The only input that you need to provide to spimcore is a text file with the extension .asc, which should contain the 32-bit instructions as ASCII in hexadecimal format (see the example in the project description). You can write a sequence of instructions manually in your .asc file, or optionally write an assembler to do that for you (i.e., convert a program to its hex sequence).

There is no output. spimcore takes care of the simulation. But you need to make sure that the datapath/control are working correctly. For your convenience, the diagram in Appendix B of the project description color-codes each function to be implemented with dotted lines.

Q. How does *instruction_fetch()* work?

A. The data you are given is in Mem. Mem is an array that is populated be functions in spimcore.c. The data is supplied to the program by the .asc input file. PC is the index of Mem[], but with a little twist: You have to use

(PC >> 2) whenever you use it. The information that is in this location referenced by Mem[(PC >> 2)] is the decimal value of the instruction that was in hexadecimal in the file.

Note that 64 kB (bytes 0-65536) of memory is available, but again they are accessed by words (unsigned Mem[]). Therefore for a given address, you have to determine its word offset, which is the index to the array Mem[].

Q. What do "RegDst" and "ALUSrc" represent?

A. These are control signals established by the control unit.

RegDst defines which register is the destination register of an instruction. If the instruction is an R-type, then the destination register is given by bits 15-11 of the instruction, and if the instruction is an I-type or branch, then the destination register is given by bits 20-16 of the instruction. If you look at the diagram in Appendix B of the project description, then you can see that for the latter case RegDst=0 and for the former RegDst=1.

ALUSrc determines the source of the second input of the ALU. This can be determined from the diagram in Appendix B of the project description. For R-type instructions, the second input to the ALU is given by the register specified by bits 20-16, which appear directly at the output Read Data 2 of the register file. For most I-type instructions however, the second input to the ALU is coming from the Sign Extend unit. BEQ is an exception here – being an I-type instruction that relies on Read Data 2 to determine whether or not to branch. Therefore based on the diagram in Appendix B of the project description, R-types and Branch utilize ALUSrc=0 while other I-types use ALUSrc=1.

Q. What R-type instructions are we supposed to set when ALUOp is equal to "111"? It just says "instruction is an R-type."

A. When the ALUOp control signal is 7 or 111_2, this tells the ALU control unit that it needs to figure out what operation to tell the ALU what to do by looking at the funct field (bits 5-0) of the instruction. This is called "multiple levels of decoding": the instruction_decode(...) function sets the initial ALUOp value of 7 and the ALU_operations(...) function should update the ALUOp value based on the funct parameter.

Q. Should we halt in rw_memory() in the event that ((ALUresult % 4) != 0)?

A. Yes, but **only if** ALUresult represents an address. ALUresult should be an address if MemRead or MemWrite is asserted.

Q. Under the ALU_operations function, what are we supposed to do with the arguments, extended_value and ALUsrc?

A. If you look at the diagram in Appendix B of the project description, your ALUsrc is a control signal to a multiplexer, which chooses between a sign-extended value or data2 in order to send the outcome to the ALU for operation.

Q. What are we supposed to do with the Zero parameter being passed into many of the functions?

A. The Zero parameter indicates that the result of the operation performed by the ALU was zero. This is primarily used for conditional branching.

Q. How do we know when a control signal is a "don't care"?

A. A control signal is a "don't care" for an instruction if its value has no effect on the correct operation of the datapath for that instruction. For instance, the ALUOp signals have no impact on the correct operation of the datapath for the jump instruction. Also, since we will not write to a destination register, the value of RegDst would be irrelevant for jump.

Q. What are argc and argv for?

A. The argc stands for "argument count" and it is also the number of elements in the argv array ("argument vector"). char **argv could also be written as char *argv[].

In this project, spimcore.c handles the command line arguments and contains the main method. All project files will be tested with the original spimcore.c file.