CDA3103 – Computer Logic and Organization

Project Description

Due Date: Submission through Webcourses, December 3rd, 11:59pm

1. Introduction

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

2. Specification of the simulator

2.1. Instructions to be simulated

You must simulate the 14 instructions listed in Figure 1 below. Please refer to section B.10 of the textbook (or the MIPS manual provided on Webcourses) for the ISA encoding 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 to be handled

MySPIM should handle the 32 general purpose registers. Floating point instructions need not be handled.

2.3. Memory usage

- The size of memory of MySPIM is 64kB (Addresses 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 multiples 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 is initialized to zero, except that specified in the "-asc" input file, as shown in the provided code.
- The memory is in *big-endian* byte order, so therefore is in the following format: e.g. Store a 32-bit number 0x*aabbccdd* to memory address 0x0 – 0x3.

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

2.4. Conditions under which MySPIM should halt

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

- An illegal instruction is encountered.
- Jumping to an address that is not word-aligned (not 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 end of memory.

Note: Any instructions other than those in the list of instructions in Figure 1 are illegal.

2.5. Format of the input machine code file

MySPIM takes hexadecimal formatted machine codes, with filename *xxx.asc*, as input. An example of a *.asc* file is shown below (and one is also given to you as an example on webcourses). Text after "#" on any line is treated as a comment.

```
20010000    # addi $1, $0, 0
200200c8    # addi $2, $0, 200
10220003    # beq $1, $2, 3
00000020    # delay slot (nop)
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

```
        beq  $1, $2, label
        beq  $3, $4, label
label:  beq  $5, $6, label
```

Machine Codes

Opcode 6 bits	Rs 5 bits	Rt 5 bits	offset 16 bits
4	1	2	0x0001
4	3	4	0x0000
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. The code should be self-explanatory. 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. Otherwise, your program may not be properly graded. **You are not allowed to modify spimcore.c and spimcore.h. All your work should be placed in project.c only.**

The details are described in Section 4 below.

3.2. MIPS assembler

This is optional. However, you may wish to write a simple assembler to test your MySPIM. The assembler should take a file with extension .asm and output a file with extension .asc containing the machine code in hexadecimal format. A correct version of this assembler is worth 20 points of extra credit on the final project.

3.3. MIPS assembly language

An introduction to the MIPS assembly language accepted by the SPIM assembler/simulator can be found from the textbook Appendix. Examples of programs written in MIPS assembly language can also be found in the lecture and lab notes.

4. The functions to be filled 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 (*Z*) to *ALUresult*.
    3. Assign *Zero* to 1 if the result is zero; otherwise, set it to 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 OR B
110	Shift left B by 16 bits
111	Z = NOT A

In the second part, you are required to fill in 9 functions in project.c. Each function simulates the operations of a section of the datapath. Figure 2 in the appendix below shows the datapath and the sections of the datapath you need to simulate.

In spimcore.c, the function Step() is the core function of MySPIM. 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 to understand the signals and data passing, and 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 line 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 the spimcore.h file).
   The meanings of the values of the control signals:
   For *MemRead*, *MemWrite* or *RegWrite*, the value 1 means enabled, 0 means 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 less than" operation
011	ALU will do "set less than unsigned" operation
100	ALU will do "AND" operation
101	ALU will do "OR" 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. Based on the value of MemWrite or MemRead to determine memory write operation or memory read operation.
  2. Read the content of the memory location addressed by *ALUresult* to *memdata*.
  3. 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 that contains the definition of the structure storing the control signals and the prototypes of the above 10 functions. The functions may contain some parameters. Read spimcore.h for more information.

Hint: Some instructions may try to write to register $zero and this is perfectly fine.
However, your simulator should ***always*** keep the value of $zero equal to 0.

NOTE: You should not do any "print" operations in your final version of project.c. Otherwise, the operation will disturb the grading process and you will be penalized.

5. Operation of the spimcore

For your convenience, here is how you could develop this project in a UNIX environment. First compile:

$ gcc -o spimcore spimcore.c project.c

After compilation, to use MySPIM, you would type the following command in UNIX:

$ spimcore <filename>.asc

The command prompt

cmd:

should appear. spimcore works like a simple debugger with the following commands:

r	Dump register contents
m	Dump memory contents (in Hexadecimal format)
s[n]	Step n instructions (simulate the next n instructions). If n is not typed, 1 is assumed
c	Continue (carry on the simulation until the program halts (with illegal instruction))
H	Check if the program has halted
d	ads1 ads2 Hexadecimal dump from address ads1 to ads2
I	Inquire memory size
P	Print the input file
g	Display all control signals
X, X, q, Q	Quit

6. Submission Guideline

Make sure that your program can be compiled and works properly.
Submit project.c online through WebCourses.

You are only required to submit project.c (Additional report to summarize your work is not required. Therefore, you should provide detailed explanations & comments in your project.c file for any partial credit).

You must work in groups of 2. Specify in project.c who the group members are, and the contribution of each member in detail. Only one group member needs to submit on webcourses.

Appendix

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
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	3 operands; logical OR
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 less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1 else $s1 =0	compare < constant; two's complement
	set less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1 else $s1 =0	compare less than; natural number
	set 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

Figure 1: Instructions to be implemented in this project.

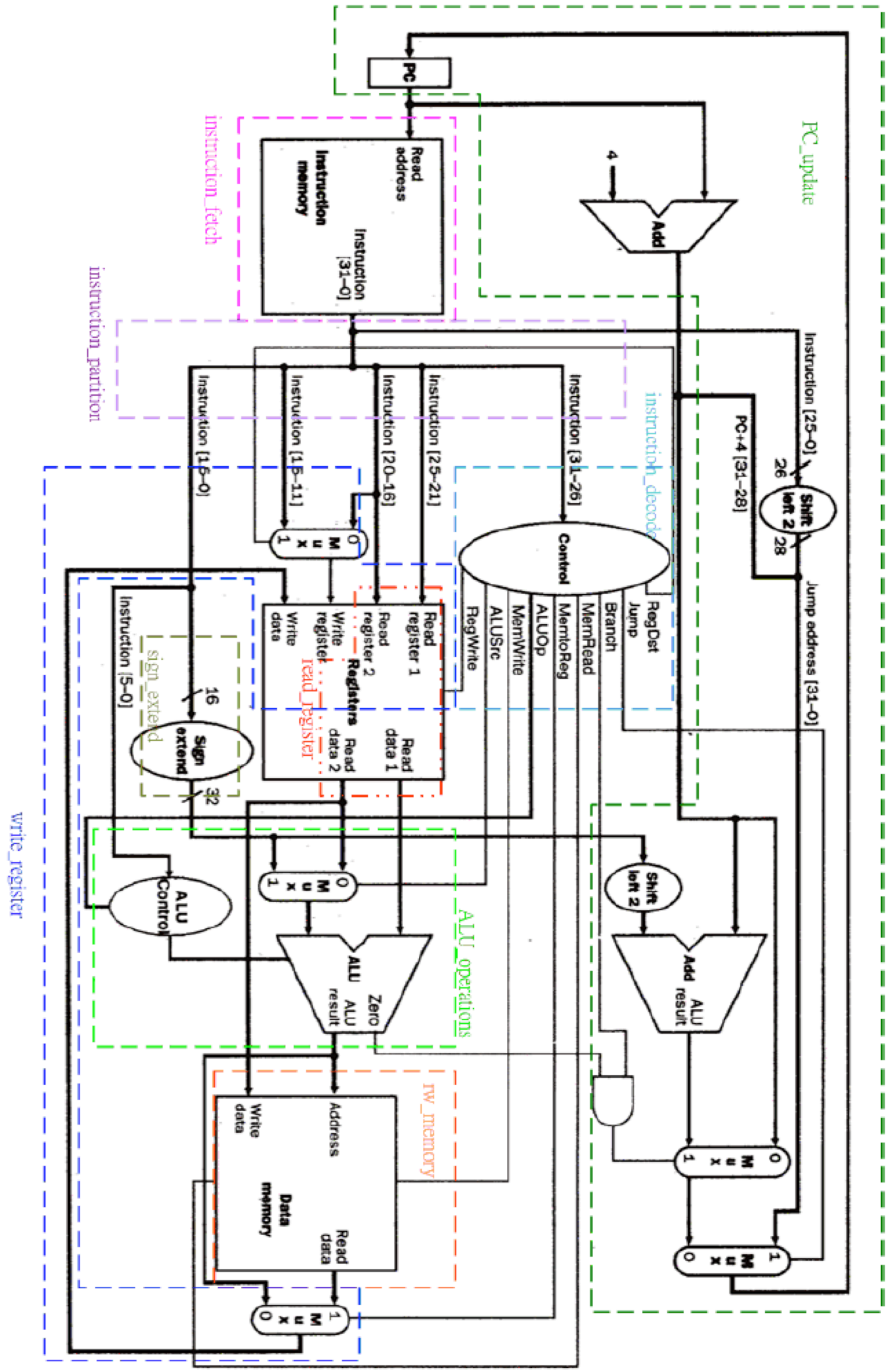

Figure 2: The single-cycle datapath to be implemented.