# CDA 3103 – Computer Logic and Organization
# Project Description

**Due Date: Submission through WebCourses, August 1, 9:00 P.M.**

## 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/Linux or Windows 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 12 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 simulate the 32 general purpose integer registers. Floating point instructions need not be handled.

### 2.3. Memory usage

- The size of memory of MySPIM is 64KB (Address 0x0000 to 0xFFFF).
- The system assumes that all program starts 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 those specified in the ".asc" file, as shown in the provided code.
- The memory is in *big-endian* byte order.
- The memory is in the following format: e.g. Store a 32-bit number 0x*aabbccdd* in memory address 0x0 – 0x3.

### 2.4. Conditions where MySPIM should halt

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 (not a multiple of 4).
- The address of a lw or sw is not word-aligned, or a lh is not halfword 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 of the Appendix are illegal.

### 2.5. Format of the input machine code file

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

```
34080064    # ori $8, $0, 100       $t0 = 100
01084821    # addu $9, $8, $8       $t1 = 200
350a007f    # ori $10, $8, 127      $t2 = 127
112a0001    # beq $9, $10, next
01285023    # subu $10, $9, $8
8d49000c    # next: lw $9, 12($10)
012a802a    # slt $16, $9, $10
08001009    # j test
00000000    # nop                   illegal instruction
03e00008    # test: jr $31          illegal instruction
```

The simulation ends when an illegal instruction, such as 0x00000000, is encountered.
Note: The jump address for the above jump instruction is 0x1009 << 2, or 16420 in decimal. This is because the target address is at the label "test" which is word 9 from the beginning of the code and the code is to be loaded starting at address 0x4000 (or 16384 in decimal value).

### 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).

## 3. Resources
### 3.1. Files provided

Please download the following files from 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 marked. **You are not allowed to modify spimcore.c or spimcore.h. All your work should be placed in project.c only.**
The details are described in Section 4 below.

### 3.2. 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 the program 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 the 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. Set *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 | If A < 0, Z = 1; otherwise, Z = 0 |
| 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 in order 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 *struct_ 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 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 compare operand A with value 0 |
| 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. Base 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 the register $zero and we assume that they are valid.  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 operations will disturb the marking 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 registers contents |

| | |
|---|---|
| m | Dump memory contents (in Hexadecimal format) |
| s[n] | Step n instructions (simulate the next n instruction). 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, 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 explanation & comments in your project.c file for any partial credits).**

**You are allowed to work in a group of 2. Specify in a comment at the top of 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

## Add Unsigned

addu Rd, Rs, Rt # RF[Rd] = RF[Rs] + RF[Rt]

| Op-Code | Rs | Rt | Rd | | Function Code |
|---|---|---|---|---|---|
| 000000 | sssss | ttttt | ddddd | 00000 | 100001 |

Add contents of Reg.File[Rs] to Reg.File[Rt] and store result in Reg.File [Rd]. No overflow exception is generated.

## Subtract Unsigned

subu Rd, Rs, Rt # RF[Rd] = RF[Rs] − RF[Rt]

| Op-Code | Rs | Rt | Rd | | Function Code |
|---|---|---|---|---|---|
| 000000 | sssss | ttttt | ddddd | 00000 | 100011 |

Subtract contents of Reg.File[Rt] from Reg.File[Rs] and store result in Reg.File[Rd]. No overflow exception is generated.

## And

and Rd, Rs, Rt # RF[Rd] = RF[Rs] AND RF[Rt]

| Op-Code | Rs | Rt | Rd | | Function Code |
|---|---|---|---|---|---|
| 000000 | sssss | ttttt | ddddd | 00000 | 100100 |

Bitwise logically AND contents of Reg.File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

## OR Immediate

ori Rt, Rs, Imm # RF[Rt] = RF[Rs] OR Imm

| Op-Code | Rs | Rt | Imm |
|---|---|---|---|
| 001101 | sssss | ttttt | iiiiiiiiiiiiiiii |

Bitwise logically OR contents of Reg.File[Rs] with zero-extended Imm value and store result in Reg.File[Rt].

## Load Word

lw  Rt, offset(Rs)  # RF[Rt] = Mem[RF[Rs] + se Offset]

| Op-Code | Rs | Rt | Offset |
|---|---|---|---|
| 100011 | sssss | ttttt | iiiiiiiiiiiiiiii |

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. A 32-bit word is read from memory at the effective address and loaded into Reg.File[Rt]. If the least 2 significant bits of the effective address are not zero, an address error exception occurs. There are 4 bytes in a word, so word addresses must be binary numbers that are a multiple of four, otherwise an address error exception occurs.

## Store Word

sw  Rt, offset(Rs)  # Mem[RF[Rs] + se Offset] = RF[Rt]

| Op-Code | Rs | Rt | Offset |
|---|---|---|---|
| 101011 | sssss | ttttt | iiiiiiiiiiiiiiii |

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The contents of Reg.File[Rt ] are stored in memory at the effective address. If the least 2 significant bits of the effective address are not zero, an address error exception occurs. There are 4 bytes in a word, so word addresses must be binary numbers that are a multiple of 4, otherwise an address error exception occurs.

## Set on Less Than

slt  Rd, Rs, Rt # if (RF[Rs] < se RF[Rt] ) then RF[Rd] = 1 else RF[Rd] = 0

| Op-Code | Rs | Rt | Rd | | Function Code |
|---|---|---|---|---|---|
| 000000 | sssss | ttttt | ddddd | 00000 | 101010 |

If the contents of Reg.File[Rs] are less than the contents of Reg.File[Rt], then Reg.File[Rd] is set to one, otherwise Reg.File[Rd] is set to zero, assuming the two's complement number system representation (used in branch macro instructions).

## Set on Less Than Immediate

slti  Rt, Rs, Imm  # if (RF[Rs] < se Imm ) then RF[Rt] = 1 else RF[Rt] = 0

| Op-Code | Rs | Rt | Imm |
|---|---|---|---|
| 001010 | sssss | ttttt | iiiiiiiiiiiiiiii |

If the contents of Reg.File[Rs] are less than the sign-extended immediate value then Reg.File[Rt] is set to one; otherwise Reg.File[Rt] is set to zero, assuming the two's complement number system representation (used in branch macro instructions).

## Set on Less Than Unsigned

sltu Rd, Rs, Rt # if (RF[Rs] < RF[Rt] ) then RF[Rd] = 1 else RF[Rd] = 0

```
                                       Function
Op-Code  Rs     Rt     Rd              Code
000000sssssttttttdddddd00000           101011
```

If the contents of Reg.File[Rs] are less than the contents of Reg.File[Rt], then Reg.File[Rd] is set to one, otherwise Reg.File[Rd] is set to zero. This assumes an unsigned number representation (only positive values).

## Branch if Equal

beq Rs, Rt, Label # If (RF[Rs] == RF[Rt] )then PC = PC + se Imm<< 2

```
Op-Code  Rs     Rt          Imm
000100ssssstttttiiiiiiiiiiiiiiii
```

If Reg.File[Rs] is equal to Reg.File[Rt] then branch to label.

## Branch if Greater Than or Equal to Zero

bgez Rs, Label # If (RF[Rs] >= RF[0]) then PC = PC + se Imm<< 2

```
Op-Code  Rs   code        Imm
000001sssss00001iiiiiiiiiiiiiiii
```

If Reg.File[Rs] is greater than or equal to zero, then branch to label.

## Jump

j Label # PC = PC(31:28) | Imm << 2

```
Op-Code             Imm
000010iiiiiiiiiiiiiiiiiiiiiiiiii
```

Load the PC with an address formed by concatenating the first 4 bits of the current PC with the value in the 26-bit immediate field shifted left 2 bits.

Note on the jump address's translation. On a real MIPS processor the jump address is calculated based on the code given above, i.e., PC(31:28) | Imm << 2, but for this project since the memory is limited to 64KB (i.e., from 0x0000 to 0xFFFF), the leading 4 bits 31:28 of the PC will be all zeros.

**Figure 1: Instructions to be implemented in this project.**