Project 1

In this project, you will create an instruction interpreter for a subset of MIPS code. Starting with the assembled instructions in memory, you will fetch, disassemble, decode, and execute MIPS machine instructions, simulating each stage in the computation. You're creating what is effectively a miniature version of MARS! There is one important difference, though—MARS takes in assembly language source files, not .dump files, so it contains an assembler, too.

You may work with a partner for this project. Each person will turn in the following for the project on UCMCrops:

- Completed `computer.c`
- Testing.txt/doc which outlines your testing strategy
- All the *.s and *.dump files you used to test your project in test.tgz
- Name of your partner in the submission box.

Project Specification

The files `sim.c`, `computer.h`, and `computer.c` comprise a framework for a MIPS simulator. Complete the program by adding code to `computer.c`. Your simulator must be able to simulate the machine code versions of the following MIPS machine instructions:

```
addu   Rdest, Rsrc1, Rsrc2
addiu  Rdest, Rsrc1, imm
subu   Rdest, Rsrc1, Rsrc2
sll    Rdest, Rsrc, shamt
srl    Rdest, Rsrc, shamt
and    Rdest, Rsrc1, Rsrc2
andi   Rdest, Rsrc, imm
or     Rdest, Rsrc1, Rsrc2
ori    Rdest, Rsrc, imm
lui    Rdest, imm
slt    Rdest, Rsrc1, Rsrc2
beq    Rsrc1, Rsrc2, raddr
bne    Rsrc1, Rsrc2, raddr
j      address
jal    address
jr     Rsrc
lw     Rdest, offset (Radd)
sw     Rsrc, offset (Radd)
```

Refer to the handout attached in the assignment page which contains the binary code of the MIPS instructions. Once complete, your solution program will be able to simulate real programs that do just about anything that can be done on a real MIPS, with the notable exceptions of floating-point math and interrupts.

The framework code

1. It reads the machine code into "memory", starting at "address" `0x00400000`. (In keeping with the MARS convention, addresses from `0x0000000` to `0x00400000` are unused.) We assume that the program will be no more than 1024 words long. The name of the file that contains the code is given as a command-line argument.

2. It initializes the stack pointer to `0x00404000`, it initializes all other registers to `0x00000000`, and it initializes the program counter to `0x00400000`.

3. It provides simulated data memory starting at address `0x00401000` and ending at address `0x00404000`. Internally, it stores instructions together with data in the same memory array.

4. It sets flags that govern how the program interacts with the user.

It then enters a loop that repeatedly fetches and executes instructions, printing information as it goes:

- the machine instruction being executed, along with its address and disassembled form (to be supplied by your `PrintInstruction` function);
- the new value of the program counter;
- information about the current state of the registers;
- information about the contents of memory.

The framework code supports several command line options:

-i	runs the program in "interactive mode". In this mode, the program prints a ">" prompt and waits for you to type a return before simulating each instruction. If you type a "q" (for "quit") followed by a return, the program exits. If this option isn't specified, the only way to terminate the program is to have it simulate an instruction that's not one of those listed on the previous page.
-r	prints all registers after the execution of an instruction. If this option isn't specified, only the register that was affected by the instruction should be printed; for instructions which don't write to any registers, the framework code prints a message saying that no registers were affected. (Your code needs to signal when a simulated instruction doesn't affect any registers by returning an appropriate value in the `changedReg` argument to `RegWrite`.)
-m	prints all data memory locations that contain nonzero values after the execution of an instruction. If this option isn't specified, only the memory location that was affected by the instruction should be printed; for any instruction that doesn't write to memory, the framework code prints a message saying that no memory locations were affected. (Your code needs to signal when a simulated instruction doesn't affect memory by returning an appropriate value in the `changedMem` argument to `Mem`.)
-d	is a debugging flag that you might find useful.

Implement

As discussed in lecture, `Fetch`, `Decode`, `Execute`, `Mem`, and `RegWrite` are the five processing stages of the MIPS archetecture. In our simulator, these steps involve completing the following tasks. The Fetch step has been implemented for you so you job is to fill in the remaining functions:

- `Decode` - Given an instruction, fill out the corresponding information in a DecodedInstr struct. Perform register reads and fill the RegVals struct. The addr_or_immed field of the IRegs struct should contain the properly extended version of the 16 bits of the immediate field.
- `Execute` - Perform any ALU computation associated with the instruction, and return the value. For a `lw` instruction, for example, this would involve computing the base + the offset address. For this project, branch comparisons also occur in this stage.
- `Mem` - Perform any memory reads or writes associated with the instruction. Note that as in the Fetch function, we map the MIPS address `0x00400000` to index 0 in our internal memory array, MIPS address `0x00400004` to index 1, and so forth. If an instruction accesses an invalid memory address (outside of our data memory range, `0x00401000` - `0x00403fff`, **or not word aligned for lw or sw**), your code must print the message, "Memory Access Exception at [PC val]: address [address]", where [PC val] is the current PC, and [address] is the offending address, both printed in hex (with leading 0x). Then you must call exit(0).
- `RegWrite` - Perform any register writes needed.

In the case of an unsupported instruction, make sure that you call exit(0) somewhere in your code, before PrintInfo and fetching the next instruction. Do not print any special error message in this case..

In the `UpdatePC` function, you should perform the PC update associated with the current instruction. For most instructions, this corresponds with an increment of 4 (which we have already added).

The `PrintInstruction` function prints the current instruction and its operands in text. Here are the details on the output format and sample.output file contains the expected output for sample.dump:

- The disassembled instruction must have the instruction name followed by a "tab" character (In C, this character is '\t'), followed by a comma-and-space separated list of the operations.
- For addiu, srl, sll, lw and sw, the immediate value must be printed as a decimal number (with the negative sign, if required) with no leading zeroes unless the value is exactly zero (printed as 0).
- For andi, ori, and lui, the immediate must be printed in hex, with a leading 0x and no leading zeroes unless the value is exactly zero (which is printed as 0x0).
- For the branch and jump instructions (except for jr), the target must be printed as a full 8-digit hex number, even if it has leading zeroes. (Note the difference between this format and the branch and jump assembly language instructions that you write.) Finally, the target of the branch or jump should be printed as an absolute address, rather than being PC relative.
- All hex values must use lower-case letters and have the leading 0x.
- Instruction arguments must be separated by a comma followed by a single space.
- Registers must be identified by number, with no leading zeroes (e.g. $10 and $3) and not by name (e.g. $t2).
- Terminate your output from the PrintInstruction function with a newline.
- As an example, for a store-byte instruction you might return "sb\t$10, -4($21)\n".
- Here are examples of good instructions printed by PrintInstruction:

```
addiu     $1, $0, -2
lw        $1, 8($3)
srl       $6, $7, 3
ori       $1, $1, 0x1234
lui       $10, 0x5678
j         0x0040002c
bne       $3, $4, 0x00400044
jr        $31
```

Here are examples of bad instructions:

```
# shouldn't print hex for addiu
addiu     $1, $0, 0xffffffff

# shouldn't print hex for sw
sw        $1, 0x8($3)

# should use reg numbers instead of names
sll       $a1, $a0, 3

# no spaces between arguments
srl       $6,$7,3

# forgot commas
ori       $1 $1 0x1234

# hex should be lowercase and not zero extended
lui       $t0, 0x0000ABCD


# address should be in hex
 j         54345

# forgot the leading 0x
 jal       00400548

# needs full target address in hex
 bne       $3, $4, 4
```

The files `sample.s` and `sample.output` provide an example output that you may use for a sanity check. We do not include any other test input files for this project. You must write the test cases in MIPS, use MARS to assemble them, and then dump the binary code. You will need to submit everything you used to test your project.

MARS places anything that follows the .data assembler directive sequentially in memory. However, this will not be reflected in the binary file that MARS dumps. That dump file only contains instructions. Therefore, instead of depending on MARS to load data memory for you, you should use instructions. For example, suppose you want to write a MIPS program that uses an array of 5 words called foo which is initialized with the integers 1, ..., 5. Normally, you would write something like:

```
        .data
foo:    .word 1,2,3,4,5
```

For this project, you would not use the .data section. Instead you would have your program initialize the array:

```
__start:
     lui     $t0, 0x1001
     ori     $t0, 0x0000
     addiu   $t1, 1
     sw      $t1, 0($t0)
     addiu   $t1, 2
     sw      $t1, 4($t0)
     addiu   $t1, 3
     sw      $t1, 8($t0)
     addiu   $t1, 4
     sw      $t1, 12($t0)
     addiu   $t1, 5
     sw      $t1, 16($t0)
```

You could also do this in a loop. This may seem a bit tedious and time consuming but it greatly simplifies the simulator.