Computer Science 281
A MIPS Simulator

In this project, you will design and implement, in MIPS assembly language, a simulator that is capable of executing MIPS (machine language) programs.

1 Memory abstractions

The first step in this project is to design an abstraction for memory. To simplify matters, we will separate instruction memory, data memory, and the stack in the simulation. The three can be modeled as arrays of bytes with sizes 8192, 16384, and 4096, respectively. You will also need space in memory to store the register file (128 bytes) and a few additional special purpose registers. Keep in mind that the MIPS programs that you are executing do not know about this arrangement; you will need to detect to which of these memories a given memory address refers.

Your simulator will begin by prompting the user for the names of two files: one containing the machine language instructions to execute and the other containing the initial contents of the data segment. The simulator will then open each of these files and load their contents into the appropriate memory. To interact with the user and work with files, you will need to use MARS system calls. You can find good documentation on these system calls in the MARS Help menu. The first 17 system calls allow for basic I/O and other system services. MARS also provides a number of more sophisticated system calls; of these, you may be most interested in numbers 51–54 which bring up dialog boxes to prompt for user input.

2 Menu

Once the files are loaded, the simulator should repeatedly present the user with a menu containing 4 options:

1. Display the contents of the machine's registers.
2. Display the contents of a portion of the machine's memory.
3. Execute a single step in the loaded program.
4. Execute the loaded program to completion.

3 Executing instructions

To execute instructions in your simulator, you will follow the standard fetch-decode-execute cycle:

1. Load the instruction at the address in the PC into the IR.
2. Increment the PC.
3. Break the instruction down into its component parts to decode it.

4. Based on the results of the previous step, execute the instruction and update the contents of the appropriate registers and/or memory.

5. Repeat step 1.

Each instruction should be implemented in a separate function in your simulator code. Rather than use a long and complicated sequence of comparisons to decide to which function to jump in each decode step, you will use *jump tables*. A jump table is simply an array containing the addresses of each of these functions, indexed by the opcode. We will spend some more time in class talking about this.

4 Phases

We will develop the simulator in a sequence of three phases:

1. Implement the register and memory abstractions, the code to load instructions and data from files, and the main runtime menu. Also implement the code that performs the register and memory dumps.

2. Implement an initial set of basic instructions:

   - `lui`
   - `or` and `ori`
   - `and` and `andi`
   - `addi`
   - `add` and `addu`
   - `sub` and `subu`
   - `lw` (from data memory only)
   - `sw` (to data memory only)

3. Implement more instructions:

   - `lw` (from data memory or stack)
   - `sw` (to data memory or stack)
   - `lb` and `lbu`
   - `sb`
   - `mult` and `div`
   - `mflo` and `mfhi`
   - `j`, `jal`, and `jr`
   - `slt`
   - `beq` and `bne`
   - `syscall` (and at least syscall codes 1, 4, 5, 8, 11, 12)