1 Introduction

For this lab, you are going to complete the implementation of your single cycle CPU. With the VHDL that is turned in with this assignment you will be able to fully simulate a subset of the MIPS ISA. The CPU will run with a single clock cycle for each instruction and should use all the components you've developed through the semester.

2 Requirements

The following requirements must be met by your CPU:

1. The assignment will be written in VHDL. The exact signature of the CPU should conform to the following definition:

   ENTITY CPU IS
   END CPU;

   In other words, there will be no inputs or outputs for your CPU.

2. The CPU must support the following MIPS instructions:

   - `add` is an R-type instruction that places $\mathcal{R}[R_s] + \mathcal{R}[R_t]$ into $\mathcal{R}[R_d]$.
   - `addu` is an R-type instruction that does the same as `add`, but treats the operands as unsigned values instead of signed values.
   - `addi` is an I-type instruction that places $\mathcal{R}[R_s] + \mathcal{S}$ into $\mathcal{R}[R_t]$.
   - `addui` is an R-type instruction that does the same as `addi`, but treats the operands as unsigned values instead of signed values.
   - `beq` is an I-type instruction that places PC + 4 + (S ¡¡ 2) into PC if $\mathcal{R}[R_s]$ equals $\mathcal{R}[R_t]$.
   - `halt` is a new J-type instruction that is encoded as all 1 bits in the opcode. This instruction stops the execution of the processor. No other instruction should execute after this instruction. Your simulation should stop executing on its own after it encounters this instruction.
   - `j` is a J-type instruction that replaces the least significant 28 bits of the PC with (A ¡¡ 2).
   - `jal` is a J-type instruction that does the same as `j`, plus it stores PC + 4 into $\mathcal{R}[31]$.
   - `jr` is an R-type instruction that stores $\mathcal{R}[R_s]$ in the PC.
   - `lw` is an I-type instruction that loads $\mathcal{M}[\mathcal{R}[R_s] + \mathcal{S}]$ into $\mathcal{R}[R_t]$.
   - `nor` is an R-type instruction that places $\overline{\mathcal{R}[R_s] + \mathcal{R}[R_t]}$ into $\mathcal{R}[R_d]$.
   - `sll` is an R-type instruction that places $\mathcal{R}[R_s] << \mathcal{F}$ into $\mathcal{R}[R_d]$.
   - `srl` is an R-type instruction that places $\mathcal{R}[R_s] >> \mathcal{F}$ into $\mathcal{R}[R_d]$.
   - `slt` is an R-type instruction that places $(\mathcal{R}[R_s] < \mathcal{R}[R_t])$: 1 ? 0 into $\mathcal{R}[R_d]$.
   - `sub` is an R-type instruction that places $\mathcal{R}[R_s] - \mathcal{R}[R_t]$ into $\mathcal{R}[R_d]$.
   - `sw` is an I-type instruction that stores $\mathcal{R}[R_d]$ into $\mathcal{M}[\mathcal{R}[R_s] + \mathcal{S}]$.
   - `ori` is an I-type instruction that stores $\mathcal{R}[R_s]|\mathcal{Z}$ into $\mathcal{R}[R_t]$.
   - `xor` is an R-type instruction that places $\mathcal{R}[R_s] \oplus \mathcal{R}[R_t]$ into $\mathcal{R}[R_d]$.

   where $\mathcal{R}$ represents the register file, $\mathcal{S}$ represents the immediate portion of an I-type instruction that is sign extended, $\mathcal{Z}$ represents the immediate portion of an I-type instruction that is zero extended, $\mathcal{A}$ represents the address portion of a J-type instruction, $\mathcal{F}$ represents the shift amount of an R-type instruction, and $\mathcal{M}$ represents the memory.

3. All gates that are used must have four or fewer inputs.

4. Each gate should have a 5ps delay.

5. If you have a behaviorally defined control unit, it should exhibit a 20ps delay.

6. The instruction and data memories should be separate entities. This means that any modifications to the data memory will *NOT* be reflected in the instruction memory. This will prevent the architecture from supporting self-modifying code.

   You should use the freely available memory located on Moodle. It provides a 64KB memory that is accessible a word at a time. You can look at the simple testbench that is also linked to from Moodle. It contains examples on how to read data from and write data to the memory unit. A quick explanation follows.

   The memory unit takes 4 inputs, plus the data lines that can be either input or output depending on the memory operation. The `ncs` input is the chip select that is activated when low. The memory doesn't do anything if this value is not low. The `addr` is the 32-bit address of the memory location that is being used. Only the least significant 16 bits are used by the memory. The `nwe` input is the write enable that is activated when low. Whenever this value goes low, the memory will store the value in `data` at the specified address. The `noe` input is the output enable that is activated when low. Whenever `nwe` is not activated and this value goes low, the memory will place the value in memory at the specified address onto the `data` output. Notice that the memory only responds to the falling edge on `noe` and `nwe`, so you need to make sure you cycle those values if you want to repeat a read or write operation on two consecutive instructions.

   Since `data` can be used as both input and output, the memory and CPU need to work together on those lines ensuring that they don't both try to send a value to `data` at the same time. The memory is properly programmed to keep from writing to `data` when `noe` is not activated. It is the responsibility of the CPU to do the same when `noe` is activated. The easiest way to do this is to set the value of each data element to 'Z', which is how VHDL indicates that the element is not being set high *or* low. See the testbench for good examples of this practice.

   The memory is initialized through a file whose name defaults to sram64kx8.dat. The format of the file is a set of lines with two hexadecimal values[1] separated by a single space. The first value is a memory location and the second value is a data value that should be placed in the memory location. Since the memory is a 32-bit, but byte addressable unit, the data values are expected to be 32-bit values as well. Make sure you don't initialize one value on top of another. The initialization code can be sparse, so you do not have to fill every memory location. The default value that memory contains is 0.

   The memory has two special addresses: 0xFFFC and 0xFFF8. These addresses act like the screen and keyboard, respectively. In other words, when you write a word to 0xFFFC, the least significant byte is displayed as a character on the screen; when you read a word from 0xFFF8, you will receive a word whose least significant byte is the ASCII representation of the next character in the keyboard buffer. You can use these two memory addresses to perform simple I/O.

3   Implementation ideas

Note that one new instruction was added since the control assignment: `halt`. The halt instruction needs to stop the control from trying to fetch the next instruction and so no new VHDL processes are triggered. Your program should run to completion and not need to be forced to quit.

---

[1]These values are temperamental and only accept lowercase characters

4   Deliverables

You should turn in an electronic copy of your VHDL, including all the components you created to construct the control units hierarchically. Additionally, you must demonstrate your lab to me at a scheduled time.

The project is worth 100 points as follows:

23 points overall structure

8 points good hierarchical design

15 points correct connections between components

72 points for CPU performing instructions correctly

4 points for each instruction

5 points for using the memory unit correctly