Project 3 Integration of 16-bit Processor

Objectives

The project requires you to integrate the different modules making up the 16-bit processor and test it by running two assembly language programs on the processor. You will use the modules that you implemented in projects 1 and 2, along with two "memory modules" that will be provided to you, and a "program counter module" that you must first implement in this project.

Additional Modules in Project 3

1. ***Instruction Memory*** is used to hold the instructions of a program while ***Data Memory*** is used to read and write program variables. Both memory modules are byte-addressable (i.e. each address refers to an 8-bit quantity). Since instructions in this processor are 16-bit quantities, successive instructions have addresses separated by 2. Similarly, LOAD and STORE instructions (the only instructions that access Data Memory) in this processor read or write 16-bit words, and so access two consecutive memory addresses. Complete implementations of both these modules are provided to you in the files **instruction_memory.v** and **data_memory.v** respectively. It will be instructive to read and understand how they work. (**Note**: In actuality, a DRAM module is much more complex with its own decoding, access, and low-level signaling mechanisms; we have abstracted the functionality and provided a simple "register-like" implementation for purposes of this project.)

2. The ***Program Counter*** (PC) (sometimes also referred to as Instruction Pointer) holds the address, in ***Instruction Memory***, of the instruction currently in execution. The ***PC*** must be updated with the address of the next instruction to be executed at the active clock edge. In normal straight-line code, the next instruction is at the following 16-bit location in ***Instruction Memory*** (*i.e.* 2 more than the current ***PC*** value).

    a. For *taken branch* instructions (*i.e.* a conditional branch instruction that satisfies the branch condition) the address of the next instruction (the "branch target address") is obtained by adding the *branch offset* to the address of the next in-line instruction address. The branch offset is part of the immediate field of the branch instruction. Note that the branch offset is a signed (two's-complement) quantity and must be accounted for in the addition. A *branch that is not taken* proceeds with the default next in-line instruction.

    b. For *jump* instructions the next instruction is always obtained exactly as is done for a taken branch. The only difference is the size of the offset field is 12 bits rather than 6 for a branch. This allows the jump instruction to span a larger range (around 2048 in each direction) for the target address.

    You must implement the **program_counter** module by completing the stub provided in its Verilog file. The module must track the **PC** value in a 16-bit register that is updated at the positive clock edge of the input clock signal **clk_pi**. The *reset* signal must clear the **PC** synchronously; otherwise if the *clock enable* signal is asserted, **PC** must be updated with the address of the next instruction to execute.

3. The **processor** module can be found in the file **processor.v.** Its purpose is to create a working processor

by integrating the seven component modules: **program_counter**, **instruction_mem**, **decoder**, **regfile**, **alu**, **branch**, **data_mem**.

There are two inputs to the *processor* module, **CLK_pi** and **CPU_reset_pi**. The *system clock* signal (**CLK_pi**) must be distributed to the clock input ports of all the sequential-logic modules. These modules also receive *reset* and *clock enable* signals. The *reset* signal should be asserted if either the hardware reset signal **CPU_reset_pi** is true or the instruction that is currently executing is **RESET**. The *clock enable* signal should be asserted if the clock-divide signal **cpu_clk_en** is asserted and the current instruction is not the **HALT** instruction. The **cpu_clk_en** is generated by the module **clkdiv** in the same source file. Normally, the c**pu_clock_en** signal is a slowed down version of the system clock **CLK_pi**; since we are dealing with a virtual rather than a physical clock, in this implementation the clock frequency is not divided and **cpu_clk_en** is true for every clock cycle. (The module **seconds_clkdiv** shows how to generate a slow 1Hz **cpu_clk_enable** signal from a 100MHz system clock. It is not used in this project.)

You must instantiate a copy of each of the seven modules and connect the appropriate ports together to create the processor. Do not add any new modules and do not change the port definitions of any of the modules. All the signals needed to implement the processor are already available from the instantiated modules, the two input signals, and the **clkdiv** module.

It is strongly suggested that you use the verbose but more robust way of binding port variables to signals. You can see this style in the testbench programs of projects 1 and 2, and in the instantiation of **clkdiv** in the file **processor.v**. By explicitly stating both the port name in the instantiated module and the signal name in the instantiating module, the chances of error are greatly reduced, especially when changes need to be made.

Testing

Two assembly language programs have been "preloaded" into the instruction memory. The first **arraysum** initializes a 5-element array with the integers 1 through 5 in a loop, and then walks through the array in a second loop and computes the sum of the array elements. The second program **fib** computes the Fibonacci numbers until the value overflows the 16-bit register and then halts.

After completing the **program_counter** and **processor** modules, you can compile all the Verilog modules (using iVerilog) together with the provided testbench **tb_processor.v**. By default the first program **arraysum** will have been hardwired into instruction memory and will execute when you begin the simulation by running your compiled program. The output will show a trace of the instruction execution. You can also compare the expected output with the results file **results_arraysum**.

To run the Fibonacci sequence program, first edit the testbench file **tb_processor.v** and change the SIMULATION TIME from 550 to 1300. Also edit **instruction_memory.v** to comment out the instructions of **arraysum** and uncomment those for **fib**. Run the program and check the output trace. You can also check the expected output in the file **results_fib.**

Assignment

1. Complete the code for module **program_counter.**
2. Complete the code for module **processor.**
3. Compile the modules including those you created in projects 1 and 2, along with the *testbench* for this project, execute, and check the correctness of program **arraysum.**
4. Edit the *testbench* and *instruction memory* modules as described earlier, recompile, execute, and check the correctness of **fib.**
5. Submit all the Verilog modules that make up the working processor on Canvas. Your design may be stressed by checking its ability to execute other assembly programs correctly in addition to the two provided.
6. If something is not working as expected please document how far you have got and where it breaks down, in a separate text file called **README**.

Good Luck

Due Date: Friday, December 4, 11:59 pm