Top Design Testbench

When designing our CPU, we wanted the CPU to work as a 'black box' from our testbench, requiring the user only to load their program, reset, and run the clk to operate. As a result, the overall testbench design is straightforward. We first instantiate the CPU module with required inputs and all output connections we would like to see. The only required inputs to run properly are CLK and RESET. In the initial block, we start by resetting the CPU. Our CPU is negedge reset, so we set RESET to 1, then 0, then 1. This ensures that the program count starts at zero. R0-R15 and the RAM initialize as don't care values. Next we load our RAM with the program using $readmemb. Ensure the filename provided to $readmemb is the same as the binary file you wish to load to memory.

NOTE: Our CPU runs correctly with Choice 1 programs (V1, V2, V3).

To run the code, all we do is provide the clock input signal. This is done by setting CLK to 1 and then 0 repeatedly in a loop. Results are displayed after each loop and also after the program is finished. Finally, we use $writememb to write the final RAM contents to "mem_results.txt".

Fetch-Decode-Execute Cycle

Our approach to implementing the fetch-decode-execute cycle was a focus on efficiency. It may be simpler to run all instructions at the same speed (ex. 3 clocks per instruction) but we lose plenty of performance doing that. We found that many instructions ran properly with only two clock edges if we implemented a small amount of optimization. For example, when we are finishing instruction # A, we can get started on processing instruction number # B. The rate at which this happens depends on the OPCODE. For fast instructions, the instructions executes with the CLK unimpeded. For longer instructions, we use a finite state machine to avoid starting the next instruction too early. Although there is room for improvement, we are happy with our design to fulfill the requirements of this project. The three speed categories we used are as follows:

For fast instructions (1 clk cycle) [ADD; SUB; MUL; ORR; AND; EOR; MOV; CMP; NOP; Undefined]:

1) At posedge clk: Disable ALU output write if not already. Disable Destination Reg write if not already. Load instruction # A to the instruction register. Increment PC from A to B.
    - Between posedge and negedge clk (always @*): Register bank provides correct source data for instruction # A. ALU performs condition check, shift/rotate of SRC2, and arithmetic.
2) At negedge clk: IF CONDITION SUCCEEDS: Set ALU output to result determined between clk edges. Set flags if required. Enable destination reg write for ADD, SUB, MUL, ORR, AND, EOR, MOV. DO NOT enable destination reg write if condition fails or if command is CMP, NOP, or undefined.

For LDR instruction (2 clk cycles):

1) At Posedge 1: Disable ALU output write if not already. Disable Destination Reg write if not already. Load instruction # A to the instruction register. Increment PC from A to B.
2) At Negedge 1: Enable destination reg write. Switch RAM Address MUX from Program Count mode to STR/LDR mode.
    - ie. Tell the memory controller to 'point' to the LDR source.
3) At Posedge 2: Disable destination reg write.
4) At Negedge 2: Switch RAM Address MUX from STR/LDR mode to program count mode. Reset state to zero.
    - ie. Tell memory controller to 'point' back to program count (instruction #B)

For STR instruction (3 clk cycles):

1) At Posedge 1: Disable ALU output write if not already. Disable Destination Reg write if not already. Load instruction # A to the instruction register. Increment PC from A to B.
2) At Negedge 1: Switch RAM Address MUX from Program Count mode to STR/LDR mode.
    - Ie. Tell the memory controller to 'point' to STR dest.
3) At Posedge 2: Enable RAM write.
4) At Negedge 2: Disable RAM write.
5) At Posedge 3: Switch RAM Address MUX from STR/LDR mode to Program Count mode.
    - Ie. Tell memory controller to 'point' back to program count (instruction #B)
6) At Negedge 3: Reset state to zero. Prepare for the next operation.

We hope that this page provides proper insight into our implementation of the fetch-decode-execute cycle and testbench. There are many other efficiency and ease of use considerations we made that we did not have time to go into (for example, our design allows us to read from and write to the same register in the same instruction, ie. ADD R1, R1, R1 works). If you have any questions or comments, feel free to contact us through canvas or by email.

Regards,
Drayton Monkman and Group 5