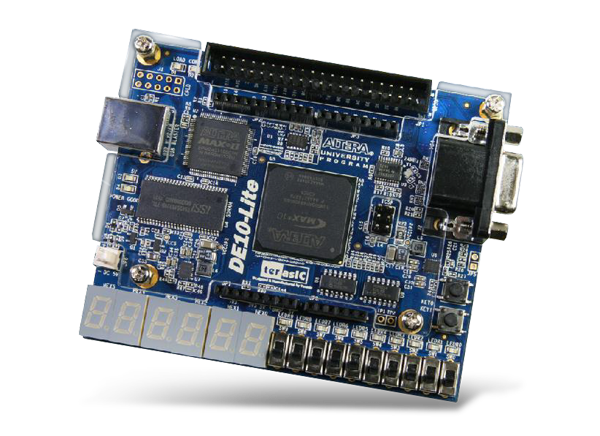
**16-BIT Single Cycle**

**CPU Analysis, Verification,**

**Synthesis, and Coverage DE10 LITE**



**Nolan Anderson**

**CPE 526 Project Report**

**April 22, 2022**

**Table of Contents**

* Executive Summary
* Introduction
* Chapter 1: Design
  + Section 1: CPU Design
  + Section 2: ISA Breakdown
  + Section 3: Top-Level Design
* Chapter 2: Verification
  + Section 1: Stimulus and Assertions
  + Section 2: RTL Analysis
  + Section 3: Coverage Report
* Chapter 3: Synthesis and Demonstration
  + Section 1: Utilization and Synthesis
  + Section 2: Demonstration Video
* Conclusion
* Appendix & Source Code

**Executive Summary**

It is common knowledge that CPU design and manufacturing is difficult and costly, but there are many methods and procedures that can help limit problems along the way. The design should be run through rigorous testing, verification, simulation, and synthesis on existing products. Tools such as ModelSim and Quartus give engineers the capability to perform these tasks. To simulate, engineers need to develop stimulus that drives every signal within the design. If done properly, a 100% coverage report paired with correct output provides reasonable proof that the design will function correctly. Synthesis is then needed to translate the proven digital design to a gate-level design. Once synthesized, a design can then be loaded and tested on an FPGA which will be the ultimate proof of the design’s functionality. These steps need to be performed throughout the entire design progress to limit costly errors. The following document outlines tests performed on a published CPU to confirm it performs as intended. Stimulus has been created which verifies the functionality of the designs components and covers every module of the CPU. A top-level design has additionally been created and synthesized to confirm the design works on a DE10 Lite board. The top-level is then demonstrated on the FPGA and the outputs are the same as simulation when performing the same stimulus.

**Introduction**

Very-large scale integration (VLSI) describes the process needed to create an integrated circuit. If you have specific hardware needs, you first create a design. This is usually completed using block diagrams so to ensure the design will function correctly at the highest lest. Then it is necessary to translate this design into a Hardware Description language (HDL) such as VHDL, Verilog, or SystemVerilog. During this phase, it is necessary to test the different components individually with stimulus to ensure they function correctly. This is done by running a test bench with stimulus through a simulation program like ModelSim. Programs like these allow you to simulate your design for correct functionality, making the process of error checking much faster and cheaper. They also usually have coverage tools to ensure that every part of the design is hit. Once all components and the overall design is simulated, covered, and checked for correct output, we then need to synthesize the design. Quartus is a good program for this and translates the HDL to a gate-level netlist. Once synthesized, we can check the designs utilization and then demonstrate the functionality of the design on a Field Programmable Gate Array (FPGA) such as a DE10 Lite. The following chapters provide an overview and testing of a Central Processing Unit (CPU) designed by *FPGA 4 Students*.

**Chapter 1: Design**

The CPU published by *FPGA 4 Students* is a 16-bit single cycle MIPS processor. The design consists of a program counter, register file, instruction memory, and data memory. The register file can hold 8 registers that are used to perform instructions. The instructions available to use in the CPU are add, sub, and, or, set less than, jump register, load and store word, branch on equal, add immediate, jump, jump and link, and finally set less than immediate. A more detailed breakdown of the ISA can be found in section two. The processor has been designed in Verilog, but this document focuses on its VHDL counterpart. Other than the language differences, the designs are identical.

**Section 1: CPU Design**

Diagram

Description automatically generatedFigure 1 depicts a high-level breakdown of the CPU whereas Figure 2 (next page) covers each module in much more detail. The 16-bit program counter is used to store the address of the next instruction in memory. While we add two to the program counter (to move onto the next instruction), we read the instruction from the instruction memory (IM). The IM can hold 16, 16-bit instructions. Instructions read out of the IM are then broken down and sent to different modules of the CPU. The control block is necessary for driving the state of the multiplexers in the CPU. For example, the RegDest signal drives which register we eventually write to in the register memory. The ALU is needed to perform the basic math operations while the data memory is reads and writes data to any 256 different 16-bit addresses. The MemToReg control block selects what register we write to from memory. Using this, we can store register values and pull them from memory later back into a register. The JR control block is used to select what register value we jump to. Lastly, the sign extension block is important so that we can modify registers and data without disturbing its positive or negative signature. The design can be found here at this [link](https://www.fpga4student.com/2017/09/vhdl-code-for-mips-processor.html), but two simple modifications have been made. To test the CPU on the board, the processor has an additional 16 std\_logic\_vector input to test input instructions. This input value is set to the instr signal in the behavioral architecture. This modification can be seen in the MIPS\_VHDL.vhd file. A final important note is all the entities of the CPU have behavioral architectures, including the top-level design. The files that make up the design are submitted alongside this document and include:

Figure 1

* MIPS\_VHDl.vhd is the main processor that ties everything together.
* ALU\_Control\_VHD.vhd is for driving the ALUControl signal.
* ALU\_VHDL.vhd holds and performs the ALU instructions.
* control\_unit\_VHDL.vhd is the control block design.
* Data\_Memory\_VHDL.vhd is the storage location for the CPU.
* register\_file\_VHDL.vhd holds register values. and the instruction memory is in
* Instruction\_Memory\_VHDL.vhd hold the stimulus that simulates the CPU.
* top\_level.vhd is the top-level design that will be described later.

**Diagram

Description automatically generated**

Figure 2: Processor Block Diagram

**Section 2: ISA Breakdown**

Table

Description automatically generatedThe Instruction Set Architecture is simple in nature like the CPU and consists of the instructions outline in the chapter heading. Figure 3 shows a more detailed breakdown of each instruction and the three different instruction formats. R-format is used for arithmetic instructions has an opcode (always 0), first and second registers (rs and rt), register destination (rd) and finally 4 bits for function select (funct). Curiously, the fourth bit of the function select is reserved exclusively for the JR, so the ALU could likely be expanded to hold more instructions. The I-type instructions are used for branches, transfer, and immediate instructions and uses the lower seven bits for addresses and immediate values. The lw and sw instructions allow for accessing and writing data memory. Addi is for adding a specific value to a register while slti checks a register against a value. Lastly, beq branches to a specific address if the first and second registers hold the same value. The J-type instructions are specifically for moving around the program counter. The jump and link (jal) instruction sets register 7 to the next program counter value and sets the current program counter equal to the address provided. This way, we can utilize the JR (jump to register) function to jump back to our place before the jal was called – cool!

Figure 3: ISA

Here is a more detailed breakdown of the different instructions:

* Add : R[rd] = R[rs] + R[rt]
* Subtract : R[rd] = R[rs] - R[rt]
* And: R[rd] = R[rs] & R[rt]
* Or : R[rd] = R[rs] | R[rt]
* SLT: R[rd] = 1 if R[rs] < R[rt] else 0
* Jr: PC=R[rs]
* Lw: R[rt] = M[R[rs]+SignExtImm]
* Sw : M[R[rs]+SignExtImm] = R[rt]
* Beq : if(R[rs]==R[rt]) PC=PC+1+BranchAddr
* Addi: R[rt] = R[rs] + SignExtImm
* J : PC=JumpAddr
* Jal : R[7]=PC+2;PC=JumpAddr
* SLTI: R[rt] = 1 if R[rs] < imm else 0
  + SignExtImm = { 9{immediate[6]}, imm
  + JumpAddr = { (PC+1)[15:13], address}
  + BranchAddr = { 7{immediate[6]}, immediate, 1’b0 }

**Section 3: Top-Level Design**

To test the CPU, we need a top-level design. This entity takes input from the 10 switches and 2 key buttons on the DE10-Lite and outputs results from the processor to the seven-segment display. The behavioral architecture instantiates the CPU and defines some signals to tie everything together. The SWAR (switch array) process block is checked every clock cycle and selects an instruction to execute. The instructions the switches select are hard coded and are outlined in section 2 of the next chapter. When key 0 is pressed on the DE10 Lite board, the selected instruction is driven to the CPU. Once the processor is finished, the seven-segment display shows the alu\_result or pc\_out signals. The alu\_result signal is the value driven out of the alu after a R-type or I-type instruction is performed. The pc\_out signal checks the result of a J-type instruction. There is a process block used to translate the pc\_out and alu\_result values to the seven-segment display and some additional signals to support this functionality. Lastly, the CPU is driven by a simple 10ns or 100Mhz clock and key 1 drives the CPU into reset state, effectively setting all values to 0. The clock definition is at the bottom of the top-level behavioral architecture. The top-level (top\_level.vhd) design is in the appendix. A demonstration of this top-level design can be found in Chapter 3 Section 2.

**Chapter 2: Verification Plan**

I verified the ISA by modifying the Instruction Memory with a set of 14 instructions. These instructions cover 98% of the design, save for a ‘when others’ statement not driven in the control block. Section one outlines the stimulus I provided to the instruction memory and section two is a breakdown of the simulated output. Section three is the output of the coverage report. **To properly run the stimulus, one needs to open the MIPS\_VHDL file and uncomment the instantiated Instruction Memory and comment out the instr <= Instruction line**. This will result in the processor using its own instruction memory rather than the instruction input.

**Section 1: Stimulus**

Figure 4 shows the instructions inserted into the Instruction Memory of the CPU. The branch on equal results in the PC driving to 0, which is its current state already. The following addi instruction stores a value in $1 so I can check the R-type instructions in the ALU. The result of these operations is very straight forward. The JR instruction jumps to register 3, so the PC gets 10. I then ensure the I-type instructions perform as designed starting with the set less than immediate (slti) instruction. After that I check addi again and then perform a load word and store word. Load word does not load any data since nothing has been stored in memory. Store word will put 20 into the data memory. Lastly, I jump to the next program counter and then finally perform a jump and link operation and sets $7 to the program counter.

Graphical user interface, application, table, Excel

Description automatically generated

Figure 4 ISA Stimulus

**Section 2: RTL Output Analysis**

Figure 5: Simulation Output

Graphical user interface, diagram

Description automatically generated

When analyzing the outputs of the instructions stored in the instruction memory, we can see that all operations in the CPU perform as intended. Figure 5 labels the output of each instruction outlined in Figure 4. As you can see, all operations are accounted for, and the CPU responds positively to the stimulus provided in the instruction memory. Each letter [A-N] corresponds to the instruction being executed. The instructions with their corresponding letters can be found in Figure 4. A higher resolution can be found on page two of the Block\_Diagram\_Simulation\_Results pdf in the submission file.

**Section 3: Coverage Report**

Text

Description automatically generatedFigure 6 is an overview of the coverage report after running through the stimulus outlined in figure four. My stimulus covered 98.02% of the design and was unable to cover a when others statement in the control block. The screenshot in figure 7 on the right shows this fact. While I was unable to hit this when others statement, the case statement covers all the OpCodes. Overall, the stimulus provided to the CPU generates a full and complete coverage report. In order to generate a coverage report, open modelsim and create a new project with the files uploaded. Then select all the files and open the compile settings. Make sure level 4 optimization is checked and all types of coverage are enabled. In the transcript window, run ‘vopt top\_level -o opttop +cover=sbecf’ followed by ‘vsim opttop -coverage’. Run the simulation with run -all and hit the break statement button to stop the CPU (otherwise it goes forever). To export the coverage, under verbosity select all the toggles under the details radial button. The coverage report I generated is in the file submission.

Figure 7: When Others

Graphical user interface, application

Description automatically generated

Figure 8: Coverage Report

**Chapter 3: Synthesis and Demonstration**

The top-level design outlined in Chapter 1 is used as an interface between the I/O of the DE10 Lite and the I/O of the MIPS processor design. Section one covers the utilization of the device on a 10M50DAF484C7G, and section two is a link to the demonstration of the top-level design on the board. **To demonstrate on the board, one needs to first comment out the Instruction Memory instantiation block and add in instr <= Instruction right below it.** This is necessary so the CPU can be driven with external instructions. The top\_level.vhd is designed to work with both and just the processor needs a slight modification.

**Section 1: Utilization, Synthesis and Timing Report**

Figure 9 is the utilization report of the top-level design from the Quartus synthesis. Figure 10 is the utilization of the MIPS Processor. The top\_level is naturally going to use slightly less resources because the instruction memory is never instantiated. The CPU certainly has room to grow on the DE10-Lite board. Likely you could enhance the design to a 32-bit architecture and still fit within the device. As mentioned in the top-level design, I ran the CPU on a 100Mhz clock because it was defined in the testbench provided by *FPGA 4 Students.* I created a clock in the timing quest analyzer, but the timing simply was not met. The clock was off by about 16ns. Since this project focuses more on the verification, synthesis, and coverage, I did not spend any time trying to meet timing. Figure 8 shows output of the timing report.

Figure 8 Timing Report

**Graphical user interface, text, application

Description automatically generatedGraphical user interface

Description automatically generated**

Figure 9: Top-Level Utilization Figure 10: MIPS Processor Utilization

**Section 2: Demonstration Video**

The demonstration video performs the stimulus I created in Figure 4 running on the board. All the instructions were performed in order. The seven-segment display shows the ALU result or PC in HEX. Make sure you don’t get confused! HEX 14 = 20 decimal. If you follow the demonstration, the outputs are the same as the wave form in Figure 5. It was difficult to get the program counter values to act correctly since the program continually runs on the board. As such some of the jump values may be incorrect, but they still jump to a location and output current program counter. Note that the initial BEQ instruction has an output of 0 in Figure 5 and in the demonstration because there is no ALU result for this instruction, and it jumps to PC 0. Overall, the output on the display matches with the waveform in figure 5 and demonstrates that the CPU design can be run on the DE10 Lite. The demonstration video can be found [here](https://drive.google.com/file/d/1iExcjPW3wHKOgWwB1FboFBoFp5bczmDz/view?usp=sharing).

**Conclusion**

Text, table

Description automatically generated The initial project proposal stated I wanted to challenge myself and design a CPU. While my time constraints limited my ability to create my own design, working to verify, simulate, and demonstrate an existing design taught me CPU design. I began work on the 17th of April, and Figure 11 shows breaks down tasks completed each day along with hours spent. While this may seem like a lot of hours, it’s what it took for me to complete the task at hand. The main bulk of the work was learning how to use top-level designs. I initially tried to add functionality to the processor definition, but this gave me lot of issues, specifically using clocks and driving output to the DE10 Lite. Once I understood a separate top-level file was recommended, this did not take me very long to complete. Generating stimulus by modifying the instruction memory that would reach 100% coverage also took some time of trial and error. While most of the instructions output was easy to verify, the jump instructions proved to be difficult. The project report forced me to check my work and gave me the confidence that all components were checked. The main things I learned were processor design, generating top-level designs, creating stimulus to drive all components of a processor, and the result of procrastination. If I was to start over, I would focus more on simulation at the start than trying to synthesize. A lot of time could had been saved if I verified the output before trying to create a functional top-level design. While this project was a challenge, I genuinely enjoying completing it.

If I was to continue with this project, I would add more functionality to the top-level design. The switches currently select hard coded instructions to demonstration proper functionality. A possible improvement would be to allow the board to create its own machine language instructions using the switches and key buttons. Another improvement to be made is a more comprehensive output to the seven-segment display. Right now, it only outputs the result of the program counter and ALU result. The CPU could also be drastically improved by adding in more instructions. The ALU certainly has space for more instructions. Overall, the CPU works well but definitely has room for improvement.

**Appendix**

**Google Drive Supporting Documents**

[Here](https://drive.google.com/drive/folders/1vsMjQa9iW9Fne6laON0Smhyen7fG4Px3?usp=sharing) is a link to the supporting documents for this project.

**Source Code is in the submission folder.**

**Top Level Design**

This file holds the instantiation of the processor and input output to the DE10 Lite board.

|  |
| --- |
| -- top\_level.vhd selects the instruction that we send to the processor.  -- The output from the processor is then sent to the HEX display on the board.  LIBRARY ieee;  USE ieee.std\_logic\_1164.ALL;  use IEEE.std\_logic\_signed.all;  -- fpga4student.com: FPGA projects, Verilog projects, VHDL projects  -- VHDL project: VHDL code for single-cycle MIPS Processor  -- VHDL testbench code for single-cycle MIPS Processor  ENTITY top\_level IS  port(  button1, button2 : IN std\_logic; -- Add in button presses:  HEX0, HEX1, HEX2, HEX3: out std\_logic\_vector(7 downto 0); -- Add in LED output  SWAR : IN std\_logic\_vector(9 downto 0) -- 10 switches on the board.  );  END top\_level;    ARCHITECTURE behavior OF top\_level IS  -- Component Declaration for the single-cycle MIPS Processor in VHDL  COMPONENT MIPS\_VHDL  PORT(  clk : IN std\_logic;  reset : IN std\_logic;  pc\_out : OUT std\_logic\_vector(15 downto 0);  alu\_result : OUT std\_logic\_vector(15 downto 0);  Instruction : IN std\_logic\_vector(15 downto 0)  );  END COMPONENT;  --Inputs  signal clk : std\_logic := '0';  signal reset : std\_logic := '1';  --Outputs  signal pc\_out : std\_logic\_vector(15 downto 0);  signal alu\_result : std\_logic\_vector(15 downto 0);  signal instr\_tp : std\_logic\_vector(15 downto 0) := "0000010010100000";  signal instr : std\_logic\_vector(15 downto 0) := "0000010010100000";  signal rji : std\_logic\_vector(1 downto 0) := "00";  -- Clock period definitions  constant clk\_period : time := 10 ns;    signal HX0, HX1, HX2, HX3, HX4, HX5: std\_logic\_vector(3 downto 0) := "0000"; -- Add in LED output  signal H0, H1, H2, H3, H4, H5 : std\_logic\_vector(7 downto 0) := "11111111";  BEGIN  -- Instantiate the for the single-cycle MIPS Processor in VHDL  uut: MIPS\_VHDL PORT MAP (  clk => clk,  reset => reset,  pc\_out => pc\_out,  alu\_result => alu\_result,  Instruction => instr  );    process(SWAR, clk)  begin  -- Switch arrangement case statement.  -- If button 1 is pressed, the instruction executes.  -- Output the result of the instruction to the HEX Display. Otherwise..  -- Output the initial register value that it intends to modify.  -- Initial register values are defined in register\_file\_VHDL.vhd.  -- For R-Type instructions, use Switch 9:  -- Add: Switch 9 and 0  -- Sub: Switch 9 and 1  -- And: Switch 9 and 2  -- Or: Switch 9 and 3  -- SLT: Switch 9 and 4  -- JR: Switch 9 and 5  -- For I and J type:  -- LW: Switch 0  -- SW: Switch 1  -- BEQ: Switch 2  -- Addi: Switch 3  -- J: Switch 4  -- JAL: Switch 5  -- STLI: Switch 6  -- JR: Switch 7  case(SWAR) is  when "0000000001" => -- LW  instr\_tp <= "1001010010000000"; -- lw $5, 7($6)  rji <= "10";  when "0000000010" => -- SW  instr\_tp <= "1011010100000000"; -- sw $5, 7($6)  rji <= "10";  when "0000000100" => -- BEQ  instr\_tp <= "1100010110000000"; -- beq $1, $2, 7  rji <= "10";  when "0000001000" => -- ADDI  instr\_tp <= "1110100010001010"; -- addi $1, $2, 10  rji <= "10";  when "0100000000" => -- ADDI2  instr\_tp <= "1110011100001010"; -- addi $6, $1, 10  rji <= "10";  when "0000010000" => -- SLTI  instr\_tp <= "0010011000001100"; -- slti $4, $1, 12  rji <= "10";  ---------------J-Type---------------  when "0000100000" => -- Jump  instr\_tp <= "0100000000001101"; -- j 13  rji <= "11";  when "0001000000" => -- JAL  instr\_tp <= "0110000000000000"; -- jal 4  rji <= "11";  ---------------R-Type---------------  when "1000000001" => -- ADD  instr\_tp <= "0000010010100000"; -- add $2, $1, $1  rji <= "01";  when "1000000010" => -- SUBTRACT  instr\_tp <= "0000100010110001"; -- sub $3, $2, $1  rji <= "01";  when "1000000100" => -- AND  instr\_tp <= "0000010101000010"; -- and $4, $1, $2  rji <= "01";  when "1000001000" => -- OR  instr\_tp <= "0000110011000011"; -- or $4, $3, $1  rji <= "01";  when "1000010000" => -- SLT  instr\_tp <= "0000100011000100"; -- slt $4, $2, $1  rji <= "01";  when "0010000000" => -- JR  instr\_tp <= "0000110000001000"; -- jr $1  rji <= "11";  ---------------OTHERS---------------  -- Output nothing to HEX display.  when "0110000000" => -- random function  --instr\_tp <=  --rji <= "01";  when others =>  instr\_tp <= "0000010010100000";  end case;  end process;    process(HX0, HX1, HX2, HX3)  begin  case HX0 is  when x"0" => H0 <= "11000000";  when x"1" => H0 <= "11111001";  when x"2" => H0 <= "10100100";  when x"3" => H0 <= "10110000";  when x"4" => H0 <= "10011001";  when x"5" => H0 <= "10010010";  when x"6" => H0 <= "10000010";  when x"7" => H0 <= "11111000";  when x"8" => H0 <= "10000000";  when x"9" => H0 <= "10011000";  when x"A" => H0 <= "10001000";  when x"B" => H0 <= "10000011";  when x"C" => H0 <= "11000110";  when x"D" => H0 <= "10100001";  when x"E" => H0 <= "10000110";  when x"F" => H0 <= "10001110";  when others => H0 <= "11000000";  end case;  case HX1 is  when x"0" => H1 <= "11000000";  when x"1" => H1 <= "11111001";  when x"2" => H1 <= "10100100";  when x"3" => H1 <= "10110000";  when x"4" => H1 <= "10011001";  when x"5" => H1 <= "10010010";  when x"6" => H1 <= "10000010";  when x"7" => H1 <= "11111000";  when x"8" => H1 <= "10000000";  when x"9" => H1 <= "10011000";  when x"A" => H1 <= "10001000";  when x"B" => H1 <= "10000011";  when x"C" => H1 <= "11000110";  when x"D" => H1 <= "10100001";  when x"E" => H1 <= "10000110";  when x"F" => H1 <= "10001110";  when others => H1 <= "11000000";  end case;  case HX2 is  when x"0" => H2 <= "11000000";  when x"1" => H2 <= "11111001";  when x"2" => H2 <= "10100100";  when x"3" => H2 <= "10110000";  when x"4" => H2 <= "10011001";  when x"5" => H2 <= "10010010";  when x"6" => H2 <= "10000010";  when x"7" => H2 <= "11111000";  when x"8" => H2 <= "10000000";  when x"9" => H2 <= "10011000";  when x"A" => H2 <= "10001000";  when x"B" => H2 <= "10000011";  when x"C" => H2 <= "11000110";  when x"D" => H2 <= "10100001";  when x"E" => H2 <= "10000110";  when x"F" => H2 <= "10001110";  when others => H2 <= "11000000";  end case;  case HX3 is  when x"0" => H3 <= "11000000";  when x"1" => H3 <= "11111001";  when x"2" => H3 <= "10100100";  when x"3" => H3 <= "10110000";  when x"4" => H3 <= "10011001";  when x"5" => H3 <= "10010010";  when x"6" => H3 <= "10000010";  when x"7" => H3 <= "11111000";  when x"8" => H3 <= "10000000";  when x"9" => H3 <= "10011000";  when x"A" => H3 <= "10001000";  when x"B" => H3 <= "10000011";  when x"C" => H3 <= "11000110";  when x"D" => H3 <= "10100001";  when x"E" => H3 <= "10000110";  when x"F" => H3 <= "10001110";  when others => H3 <= "11000000";  end case;  end process;    -- Update output hex values.  HEX0 <= H0;  HEX1 <= H1;  HEX2 <= H2;  HEX3 <= H3;    -- Process results that come out of the ALU.  process(alu\_result)  begin  case rji is  when "11" =>  HX0 <= pc\_out(3 downto 0);  HX1 <= pc\_out(7 downto 4);  HX2 <= pc\_out(11 downto 8);  HX3 <= pc\_out(15 downto 12);  when others =>  HX0 <= alu\_result(3 downto 0);  HX1 <= alu\_result(7 downto 4);  HX2 <= alu\_result(11 downto 8);  HX3 <= alu\_result(15 downto 12);  end case;  end process;  -- Clock process definitions  process(clk, button1, button2)  begin  clk <= not clk after 5ns;  reset <= '0' after clk\_period;  if(button2 = '0') then  instr <= instr\_tp;  end if;  if(button1 ='0') then  reset <= '1';  end if;  end process;  END; |