Multi-Cycle MIPS Implementation in VHDL

**Authors:** Austin Brown - Datapath, Registers, Memory

Daniel Mills - ALU, ALU Control, Control

**Course:** CPE 526

**Date:** Apr 23, 2021

**Table of Contents:**

1. Executive Summary
2. Introduction
3. Version Control
4. Design
   1. Top-Level MIPS Module
   2. Control
   3. ALU Control
   4. Datapath
   5. ALU
   6. Memory
   7. Program Counter
   8. Sign Extend
5. Verification
   1. ALU and ALU Control
   2. Control
   3. Sign Extend
   4. Memory
   5. Program Counter
   6. Assembly Code
   7. Top-Level File Testbench Results
6. Synthesis
   1. Design and Overcoming Problems
   2. Compilation Report
   3. Design Results
7. Conclusion
8. Appendix
   1. Models
      1. DE10\_LITE\_Golden\_Top.vhd
      2. Mips.vhd
      3. Control.vhd
      4. Alu\_control.vhd
      5. Datapath.vhd
      6. Memory.vhd
      7. Program\_counter.vhd
      8. Sign\_extend.vhd
      9. Alu.vhd
      10. display\_driver.vhd
   2. Test Benches
      1. Mips\_tb.vhd
      2. Alu\_tb.vhd
      3. Control\_tb.vhd
      4. Datapath\_tb.vhd
      5. Program\_counter\_tb.vhd
      6. sign\_extend.vhd

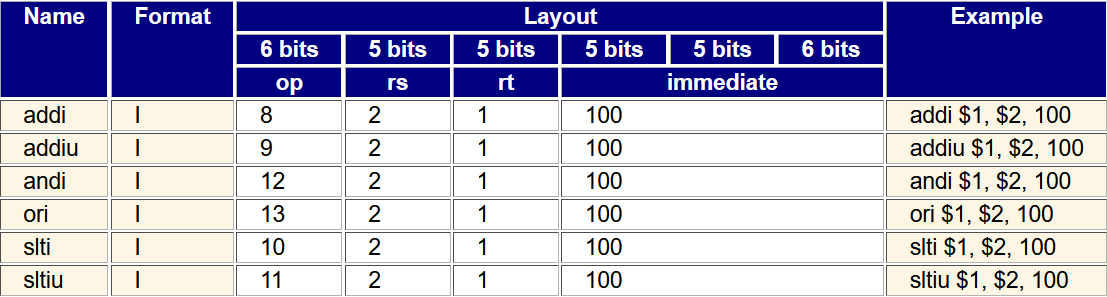
**Executive Summary:**

Our team chose to implement a Microprocessor without Interlocked Pipelined Stages, or MIPS, microcontroller on the DE10-Lite board. We utilized our VHDL knowledge and the tools Quartus and Modelsim in order to design, verify, and synthesize this microcontroller. We found that we were able to implement the ADD, SUB, AND, OR, LW, SW, and SLT instructions. We failed to implement the BEQ instruction successfully, although we worked towards it. We successfully verified the degree to which our design works theoretically. Our synthesis did not turn out as expected. Although our design synthesized to the board, we only occasionally received a valid output that we expected from our simulations. In short, we are proud of what we were able to accomplish thus far in our project, though we did not meet all of our design goals.

**Introduction:**

MIPS is a Reduced Instruction Set Computer (RISC) architecture. This means that the instruction set is small and heavily optimized. This is the opposite of a Complex Instruction Set Computer (CISC), which has a large number of highly specialized instructions. MIPS is also a load store architecture. This means that all instructions read, operate on, and write to registers. The only exceptions are load and store operations which read and write from memory respectively.

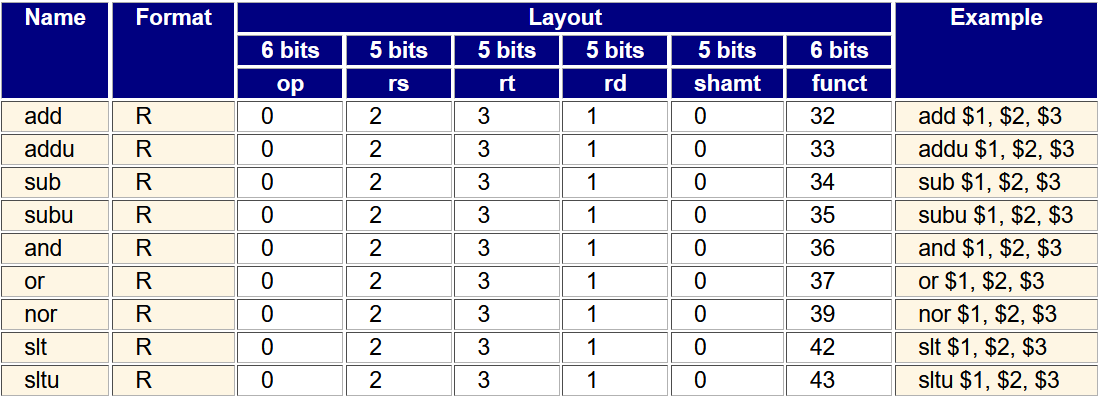
MIPS has three instruction formats. They J, I, and R. The format for I-type instructions is shown below.



**Figure 1.** I-type Instruction Diagram

The op field determines what instruction is being executed. Rt is the destination register. This is where the result will be stored. The immediate value is a 16-bit value that will be used in computation. This value will be sign-extended to 32 bits in the datapath. Rs is the source register. This is what the immediate value will operate on.

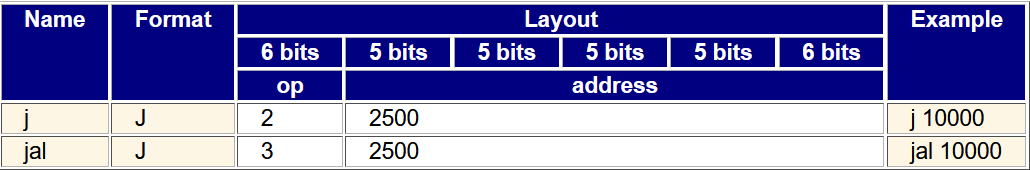
R-type instructions have registers for all three operands. The format is shown below.



**Figure 2.** R-type Instruction Diagram

R-type instructions have an op field that is always 0. The funct field is used to tell the difference between instructions. Rd designates the destination register - where the result will be stored. Rs and rt are the operands.

The final instruction type is the J-type. The format is shown below.



**Figure 3.** J-type Instruction Diagram

J type instructions have an op field and an address field. The address field is 26 bits long. Concatenating the 4 high order bits of the program counter, the address field, and two zero bits creates this 32-bit address.

At first glance, a multi-cycle implementation might seem less efficient than a single-cycle implementation. This is not true because the single-cycle implementation is held back by the amount of time it takes to read and write to memory. In a multi-cycle implementation, load and store instructions take more cycles to execute while instructions that only operate on registers take fewer cycles. This allows for higher clock frequencies.

This works by implementing the control unit as a state machine. There are a total of 5 states. The are instruction fetch, instruction decode, execute, read or write to memory, and write back to the register file. Not all instructions will use every state. This is what makes the multi cycle implementation work.

**Version Control:**

Our team utilized the version control software known as Git to manage our code and revision history. It allowed us to work on modules asynchronously and to integrate our changes throughout development. A link to our source code is provided here:

https://github.com/dandeto/vhdl-mips

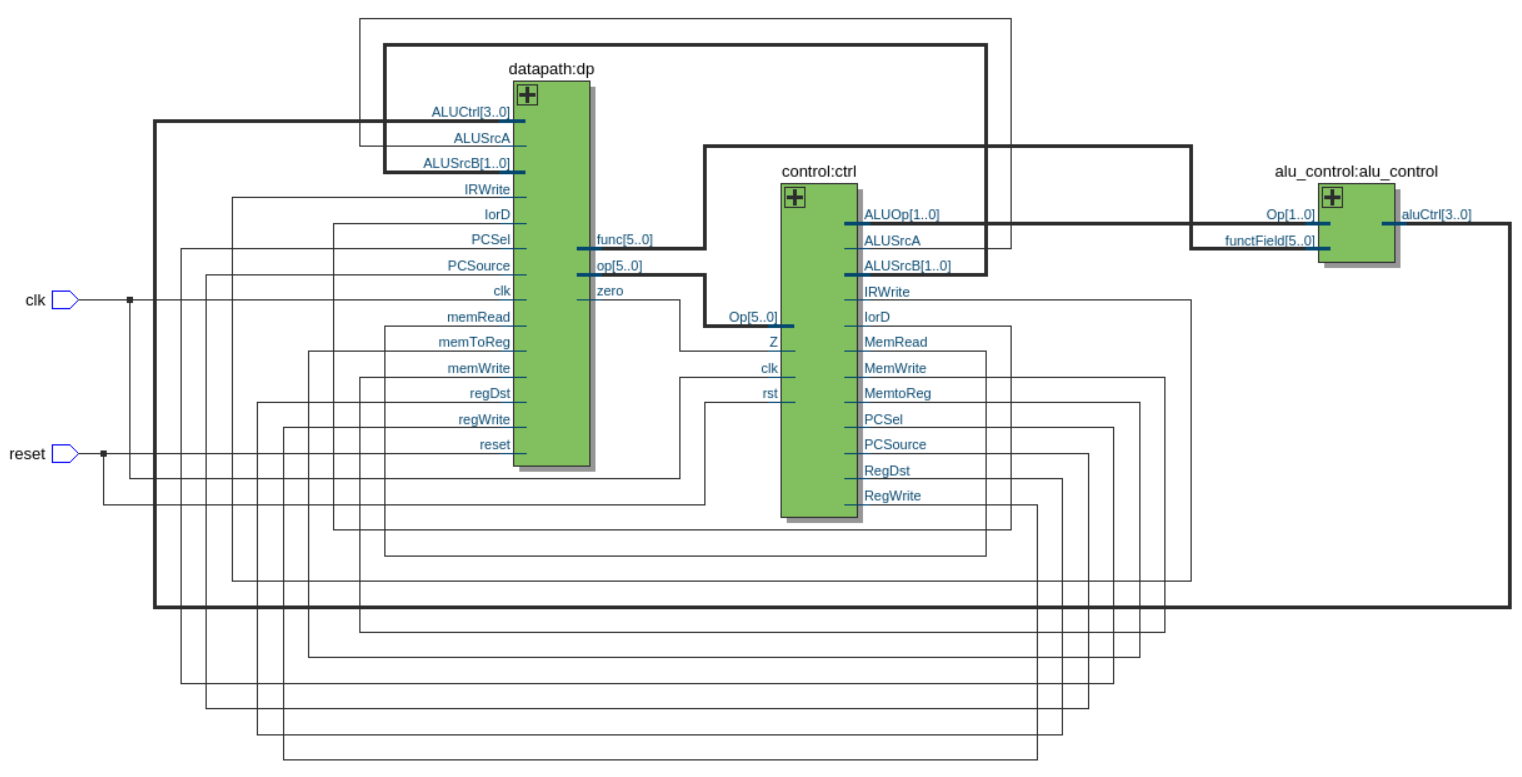
**Design**

**1. Top-Level MIPS Module**

The mips module is the top level of our design. Its purpose is to tie other larger modules together. The entity of this module is shown below. The two inputs are the clock signal and the reset signal.

|  |
| --- |
| entity mips is  port  (  clk : in *std\_logic*;  reset : in *std\_logic*  );  end mips; |

Below is the diagram of the MIPS module. This diagram was generated by Quartus post-synthesis. The three modules shown below are the datapath, the control, and the alu control. The datapath takes signals from the control and the ALU control units and performs instruction execution. The datapath contains other sub-modules inside of it.



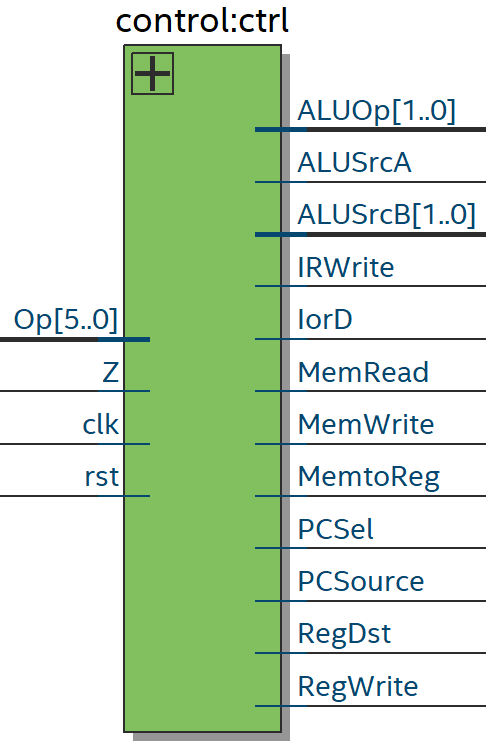
**Figure 4.** ALU Control Block Diagram

**2. Control**

The control module is a Finite State Machine (FSM) that determines which stage of instruction processing the microcontroller is in. Our state machine had the following 9 states:

1. Instruction Fetch
2. Instruction Decode / Register Fetch
3. Memory Address Computation
4. Memory Access 'load word'
5. Memory Access 'store word'
6. Memory Access Finished
7. Execution
8. R-type completion
9. Branch completion

The specific inner-workings of the control unit can be found in the source code. A high-level description is presented here. The control unit accepts two input signals from the datapath, while *rst* and *clk* are tied directly to the board. The zero flag, *Z*, originates in any arithmetic operation, while the 6-bit *Op* signal is the Op field from whatever instruction that is currently being processed. The control considers the *Op* signal when advancing states. The Z flag is used to set the *PcSel* output. All other output signals are a result of the current state.



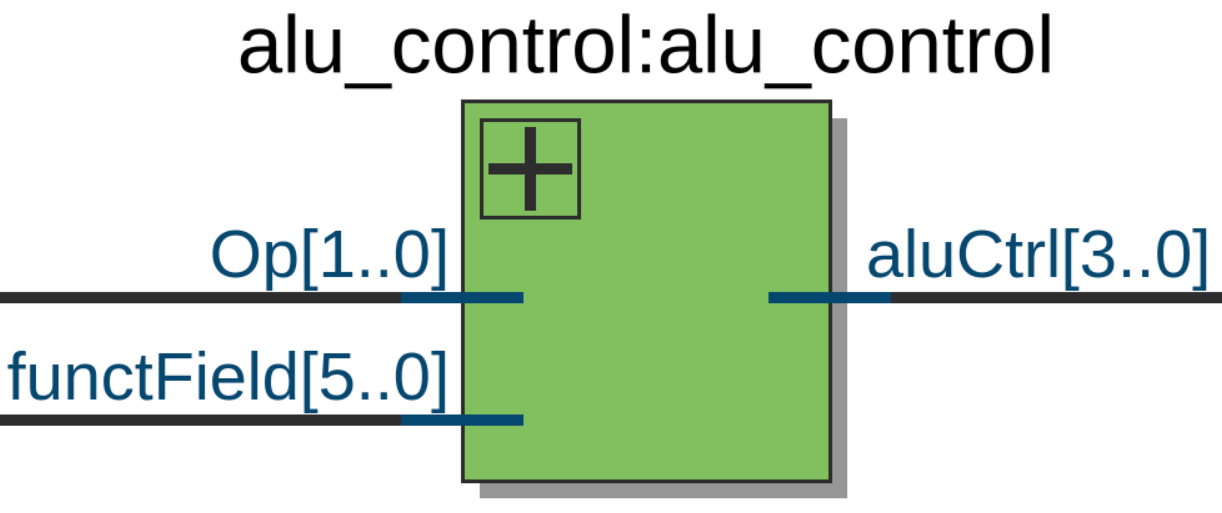
**Figure 5.** ALU Control Block Diagram

|  |  |
| --- | --- |
| **Output Signal** | **Description** |
| ALUOp | Sent to ALU Control  00: Load or store word  01: Branch  10, 11: R-type instructions |
| ALUSrcA | 0: ALU operand A is PC output  1: ALU operand A is from register file |
| ALUSrcB | 00: ALU operand B is from register file  01: ALU operand B is 1 for word aligned PC increments  10: ALU operand B is sign extended  11: ALU operand B is sign extended and shifted left by 2 |
| IRWrite | Enable writing instruction to Instruction Register |
| IorD | Reads instruction from PC when HIGH  Reads data from memory (through ALU) when LOW |
| MemRead | Sets memory module to read mode when HIGH |
| MemWrite | Sets memory module to write mode when HIGH |
| MemtoReg | 0: Read Memory Data Register into register  1: Read ALU result into register |
| PCSel | PC enable bit |
| PCSource | 0: PC gets ALUResult  1: PC gets ALUOut |
| RegDst | 0: Get bits 20 to 16 from instruction  1: Get bits 15 to 11 from instruction |
| RegWrite | Enable writing to register |

**Table 1.** Control Unit Output Signals

**3. ALU Control**

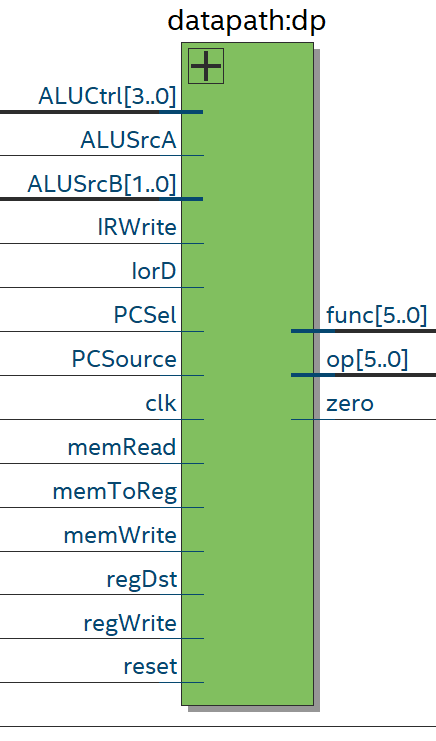
The Arithmetic Logic Unit (ALU) control module is responsible for selecting the operation that the ALU will perform. It takes two input signals. *Op* is a 2-bit signal sent from the control module. It controls which format of instruction the ALU will perform - I, R, or J type. The signal *functField* designates which R type instruction is to be performed. We used a simple VHDL **case?** statement to implement the ALU control’s functionality. This aforementioned code is present in the Appendix. The output signal *aluCtrl* is sent to the datapath module in order to properly control the ALU component which resides within.

****

**Figure 6.** ALU Control Block Diagram

**4. Datapath**

The datapath accepts control signals from the control and ALU control modules. From there, it performs code execution. The datapath contains many smaller modules such as the ALU, memory, program counter, registers, and sign extend. All of these are connected together in that datapath. The datapath interface can be seen below.

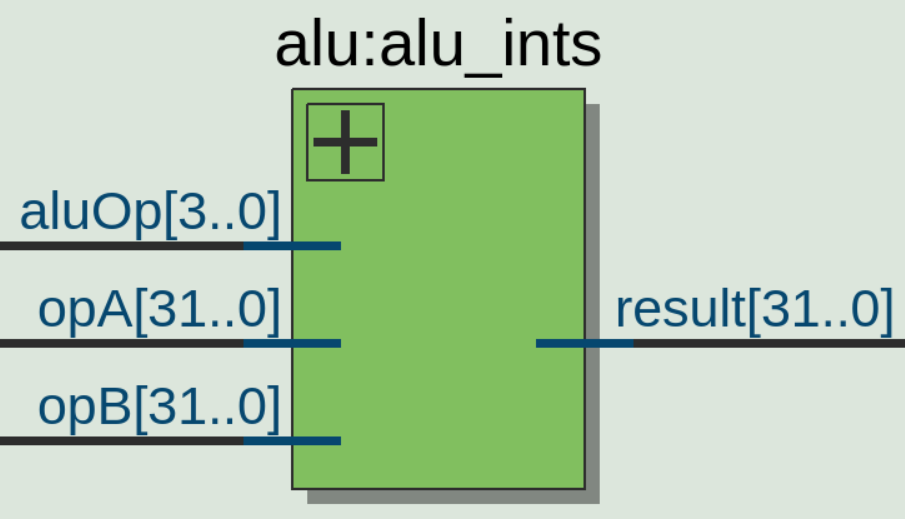


**Figure 7.** Block Diagram

Most of the inputs are control signals, which were discussed in the control section. There is also an input for a clock. This is because various memory units are used for the registers, memory block, and program counter. As for the outputs, they are func, op, and zero. The func field goes straight to the ALU control module. Its purpose is to differentiate between R-type instruction as they have the same op field. The op output is sent to the datapath. It is used for ALU operations. The zero flag is sent to the control unit. The zero flag is set to high if the ALU result is 0.

**5. Arithmetic Logic Unit (ALU)**

The purpose of the ALU is to perform arithmetic operations. It supports logical and, logical or, addition, subtraction, set on less than, and branching.



**Figure 8.** ALU Block Diagram

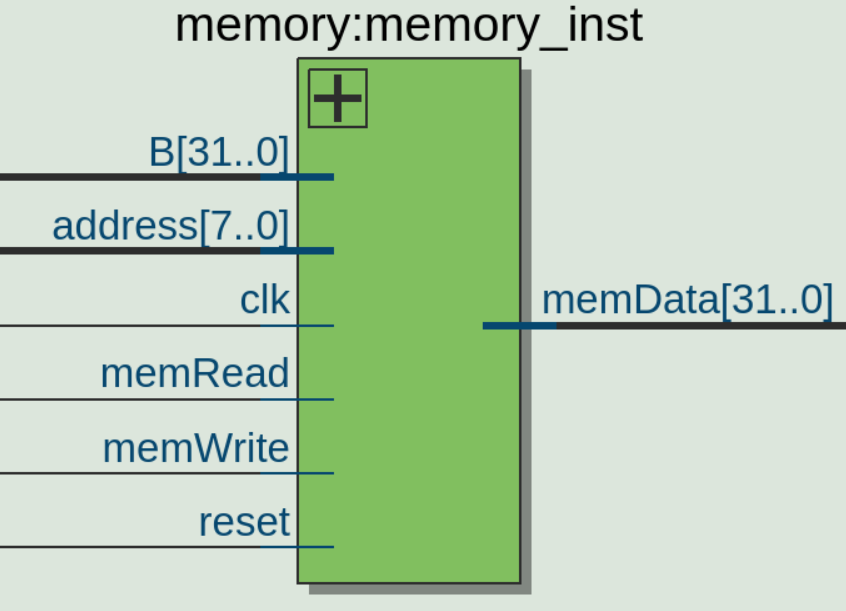
The ALU takes three inputs: *aluOp*, *opA*, and *opB*. *OpA* can either come from the program counter or from the registers. There is a multiplexer in datapath that controls which one gets sent to the ALU. The control signal used is *ALUSrcA*. *OpB* can come from several different sources. It can come directly from the register, it can come from the sign extend module, or it can be shifted left by 2. *OpB* is controlled by *ALUSrcB*. The *aluOp* input controls what operation is performed. The table below shows what aluOp code corresponds to what operation.

|  |  |
| --- | --- |
| Operation | Code |
| and | 0000 |
| or | 0001 |
| addition | 0010 |
| subtraction | 0110 |
| Set on less than | 0111 |
| branch | 1100 |

**Table 2.** ALU Operation Codes

**6. Memory**

The data and instruction memory were implemented as a combined module. The inputs and outputs are shown below.



**Figure 9.** Memory Block Diagram

Address points to an element in memory. *MemRead* and *memWrite* are both signals from the control module. When *memRead* is high, the location pointed to by address is written to *memData*. When *memWrite* is high, the location pointed to by *address* is written to. Its value will be set to *B*. The *reset* switch forces the memory back into an initialized state. This was necessary because Quartus would not allow us to simply initialize instruction memory. This is known as a power on reset.

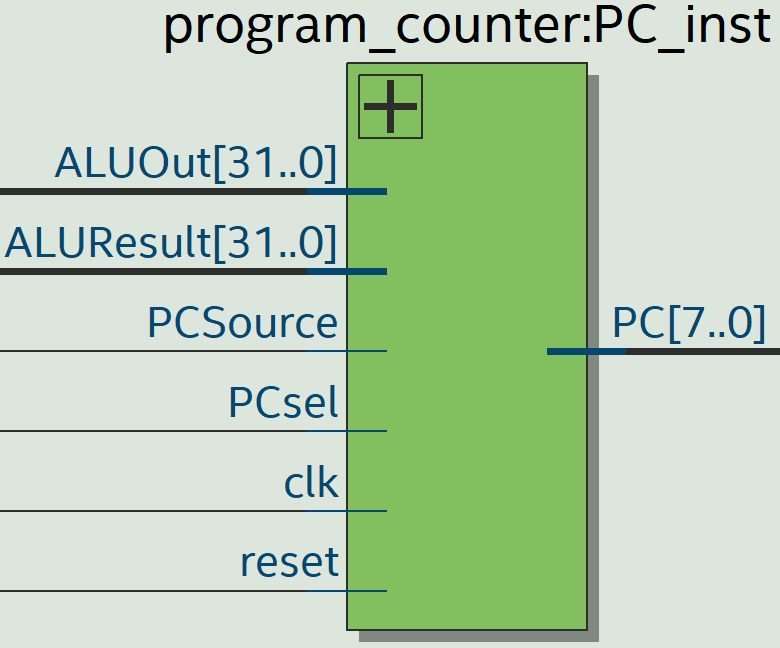
The memory block was created as an array of standard logic vectors. This declaration is shown below.



Memory\_block is a user defined type. Mem is an instantiation of that type.

**7. Program Counter**

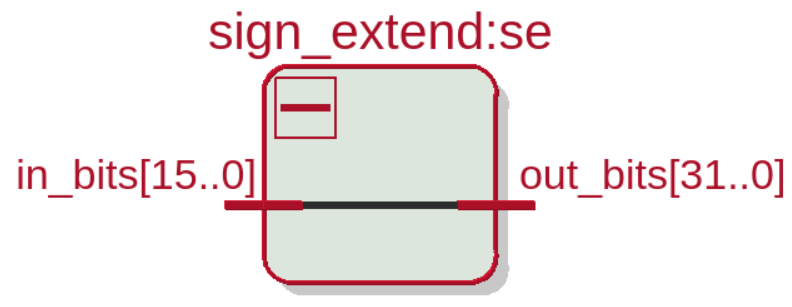
The Program Counter (PC) is a pointer to the current instruction in memory. Upon reset, the PC will be set to the value of 128 in decimal, since that is where instruction memory begins. The control module will send the PCWrite signal to the PC when it is in the instruction fetch state and it needs to be incremented to the next location in memory. The *clk* and *rst* signals come directly from the board. The *ALUOut* signal is sent from an intermediate register, while the *ALUResult* signal comes directly from the ALU. The *PCSource* signal determines which signal to use. *PCsel* is an enable bit. The output signal *PC* is the 8 least-significant-bits from whichever ALU source has been chosen.



**Figure 10.** PC Block Diagram

**8. Sign Extend**

The Sign Extend module is quite simple. The 16-bit input is converted into a 16-bit output for use in I-type instructions. The 16 extra bits to be prepended are determined by the input’s most significant bit.



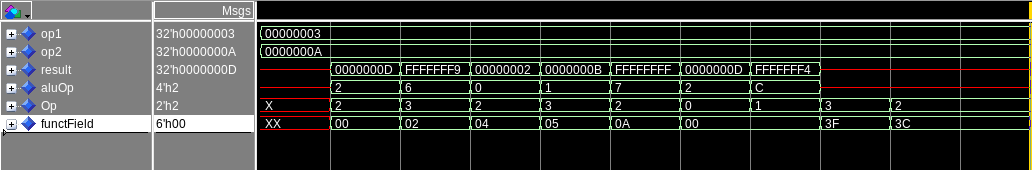
**Figure 11.** Sign Extend Block Diagram

**Verification:**

**Module Testbench Results**

**1. ALU and ALU Control**

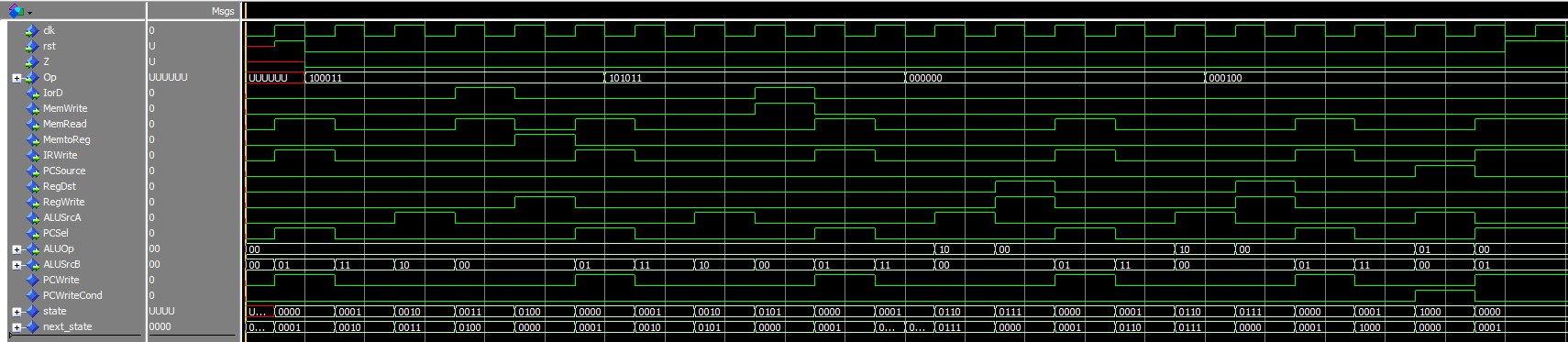
The following image is evidence that the ALU Control module produces the expected *aluOp* control signal upon stimulus from changing *Op* and *functField* signals. The first input of *Op*: 2 and *functField:* 0 correctly produced the *aluOp* of 2, and prompted the ALU to add *op1* and *op2*, producing a *result* of 0xD. The next operations are listed in order: subtraction, logical AND, logical OR, set on less than, load word, branch if equal, undefined, and undefined. The load word control signal is the same as the addition control signal, which is why the ALU produces the result 0xD when load word is inferred from the *Op* and *functField*. All results pictured here were expected and valid.



**Figure 12.** ALU & ALU Control Testbench Waveform

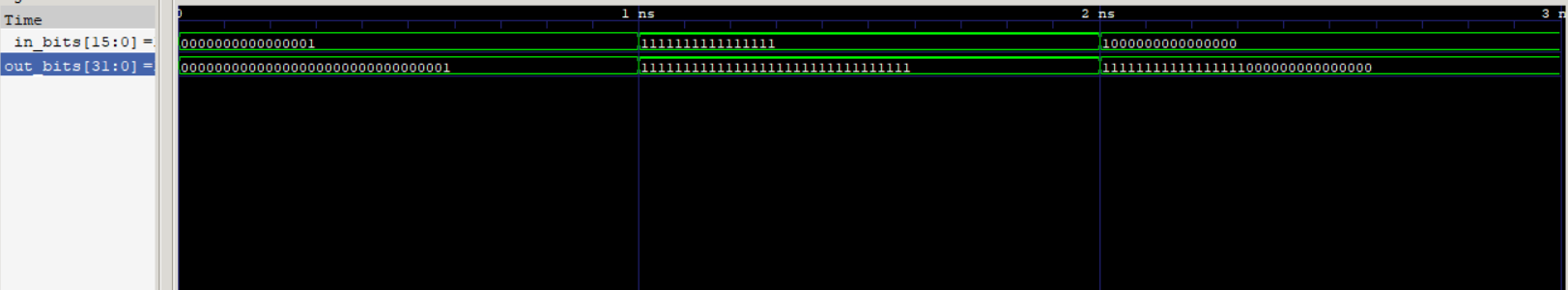
**2. Control**

This testbench was stimulated with varying *Z* and *Op* inputs in order to produce this range of valid states and outputs. All 4 types of instructions were simulated here. By name, they are load word, store word, R-type, and branch. Each type of instruction was given 5 cycles to run to completion, no matter what type it was. This was in order to make sure that each instruction would execute properly and would not be corrupted by a late change in *Op*. Changing *Op* during or before FETCH or DECODE, or after the EXECUTE state will result in correct operation. We tested both of those conditions and achieved favorable results. All other signals are set based off of the current state. They were set to the expected values.

****

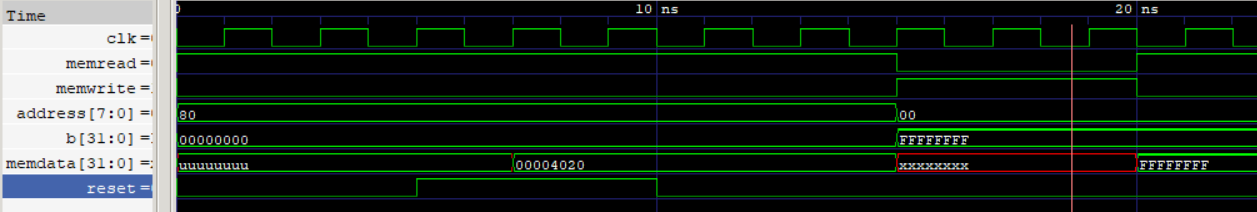
**Figure 13.** Control Testbench Waveform

**3. Sign-Extend**

A picture of this very simple test bench follows. The most significant bit of the input signal is extended from its original 16-bit length to the output width of 32. We determined that testing any other values than the following would be unnecessary, and the result was satisfactory.

**Figure 14.** Sign Extend Testbench Waveform

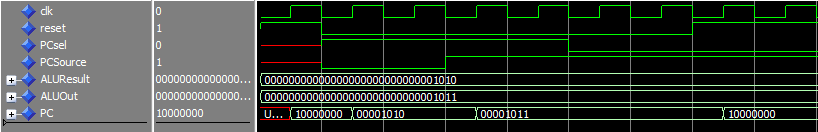
**4. Memory**



**Figure 15.** Memory Testbench Waveform

**5. Program Counter**

This testbench was crafted in order to determine whether or not the output signal *PC* would select from the intermediate register *ALUOut* or the direct line from the ALU: *ALUResult*. When *reset* is high at the beginning and end of the testbench, the default value of 128 is loaded into the PC. While *PCsel* is enabled and *PCSource* changes, the PC will load the proper value. We deemed this module to be validated.



**Figure 16.** PC Testbench Waveform

**6. Assembly Code**

The following instructions perform all operations and store the result to memory location 72 (0x0048). I attached a debug signal to watch this specific memory location, so the proper operations could be easily verified. This is shown in the section to follow. This program is verified in the top level test bench, but it failed in synthesis because it only partially works on the physical board.

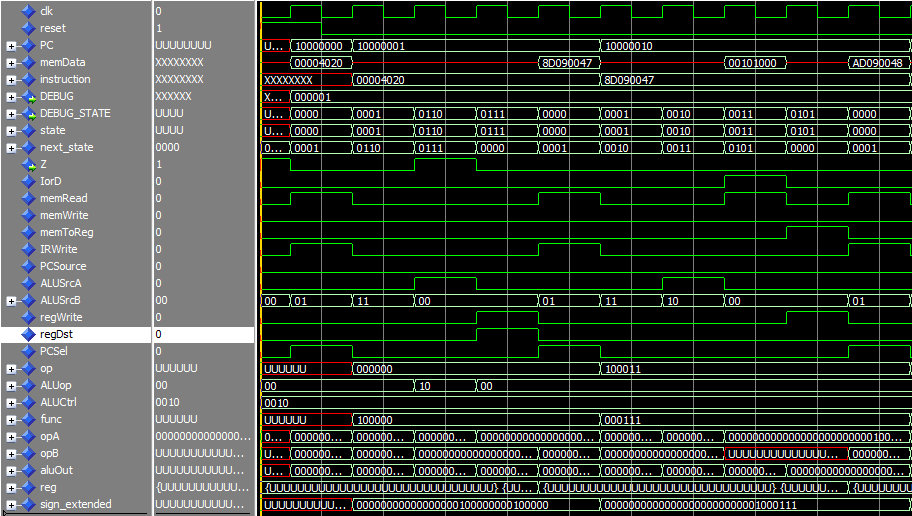
|  |  |  |
| --- | --- | --- |
| **MIPS Instruction** | **Hex OpCode** | **description** |
| ADD $t0 $zero $zero | 00004020 | t0 is pointer to base adr |
| LW $t1 0x0047 $t0 | 8D090047 | t1 <- mem[0x47] |
| SW $t1 0x0048 $t0 | AD090048 | Mem[0x48] <- t1 |
| LW $t2 0x0046 $t0 | 8D0A0046 | t2 <- mem[0x46] |
| SW $t2 0x0048 $t0 | AD0A0048 | Mem[0x48] <- t2 |
| ADD $t3 $t1 $t2 | 012A5820 | t3 <- t1 + t2 |
| SW $t3 0x0048 $t0 | AD0B0048 | Mem[0x48] <- t3 |
| SUB $t3 $t1 $t2 | 012A5822 | t3 <- t1 - t2 |
| SW $t3 0x0048 $t0 | AD0B0048 | Mem[0x48] <- t3 |
| AND $t3 $t1 $t2 | 012A5824 | t3 <- t1 & t2 |
| SW $t3 0x0048 $t0 | AD0B0048 | Mem[0x48] <- t3 |
| OR $t3 $t1 $t2 | 012A5825 | t3 <- t1 | t2 |
| SW $t3 0x0048 $t0 | AD0B0048 | Mem[0x48] <- t3 |
| SLT $t3 $t1 $t2 | 012A582A | t3 <- t1 < t2 |
| SW $t3 0x0048 $t0 | AD0B0048 | Mem[0x48] <- t3 |
| LW $t1 0x0045 $t0 | 8D090045 | t1 <- mem[0x45] |
| LW $t2 0x0044 $t0 | 8D0A0044 | t2 <- mem[0x44] |
| SW $t1 0x0048 $t0 | AD090048 | Mem[0x48] <- t1 |
| SW $t2 0x0048 $t0 | AD0A0048 | t1 <- mem[0x48] |
| ADD $t2 $t2 $t1 | 01495020 | t2 <- t2 + t1 |
| SW $t2 0x0048 $t0 | AD0A0048 | Mem[0x48] <- t2 |
| BEQ $t2 $t1 0xFFEB | 1149FFEB | goto add until t2 = t1 |

**Table 3.** MIPS Assembly Program for Testing

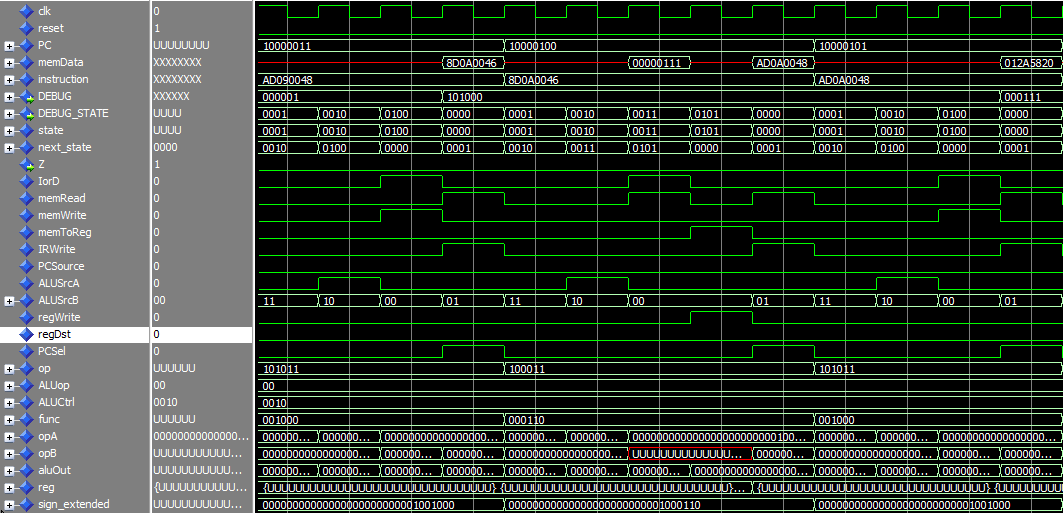
**8. Top-Level File Testbench Results**

This top-level testbench validated all operations except for BEQ. Unfortunately, we were unable to complete this instruction’s correct operation. When looking at the following waveforms, we ask the reader to pay special attention to the DEBUG signal while referencing the above table of instructions. A much easier-to-read table summarizing the results will follow images of the testbench’s waveform diagram. In order to fit the entire simulation results into this document in a readable format, we had to split the results into several smaller images. The full waveform in high resolution is available for download here:

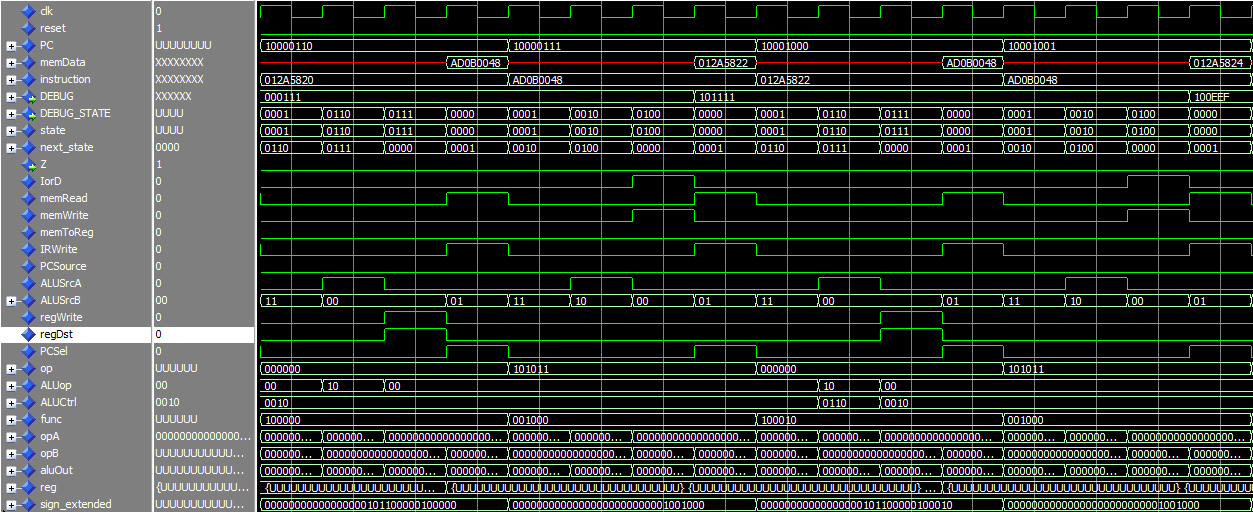
https://drive.google.com/file/d/16AxCjvTToWbO1xX3ZeF0j1Qk2hEkOobe/view?usp=sharing

****

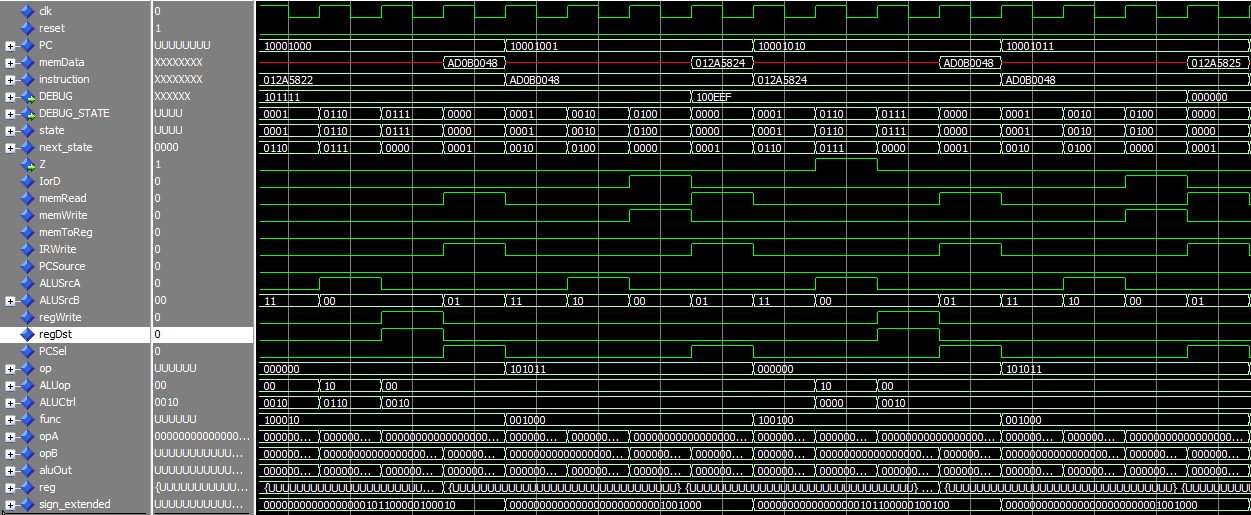
**Figure 17.** Top-Level MIPS Testbench Waveform (1/8)

****

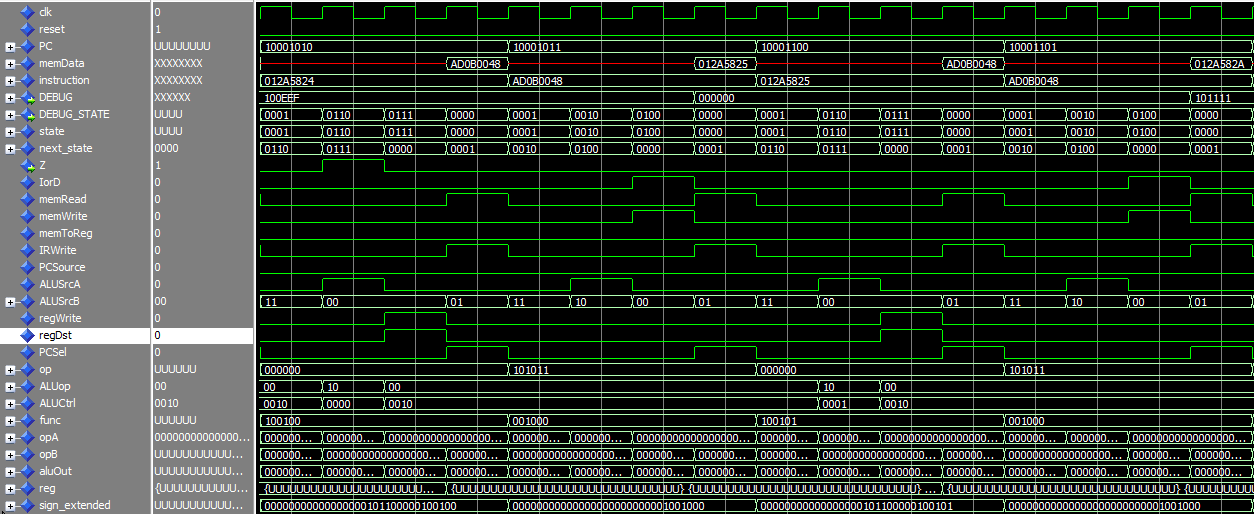
**Figure 18.** Top-Level MIPS Testbench Waveform (2/8)

****

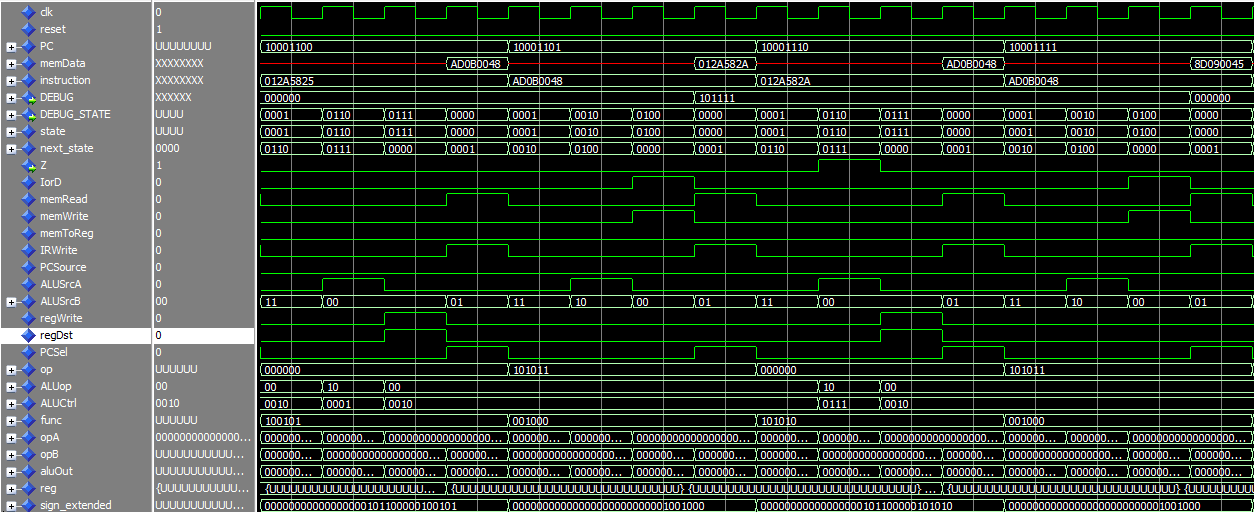
**Figure 19.** Top-Level MIPS Testbench Waveform (3/8)

****

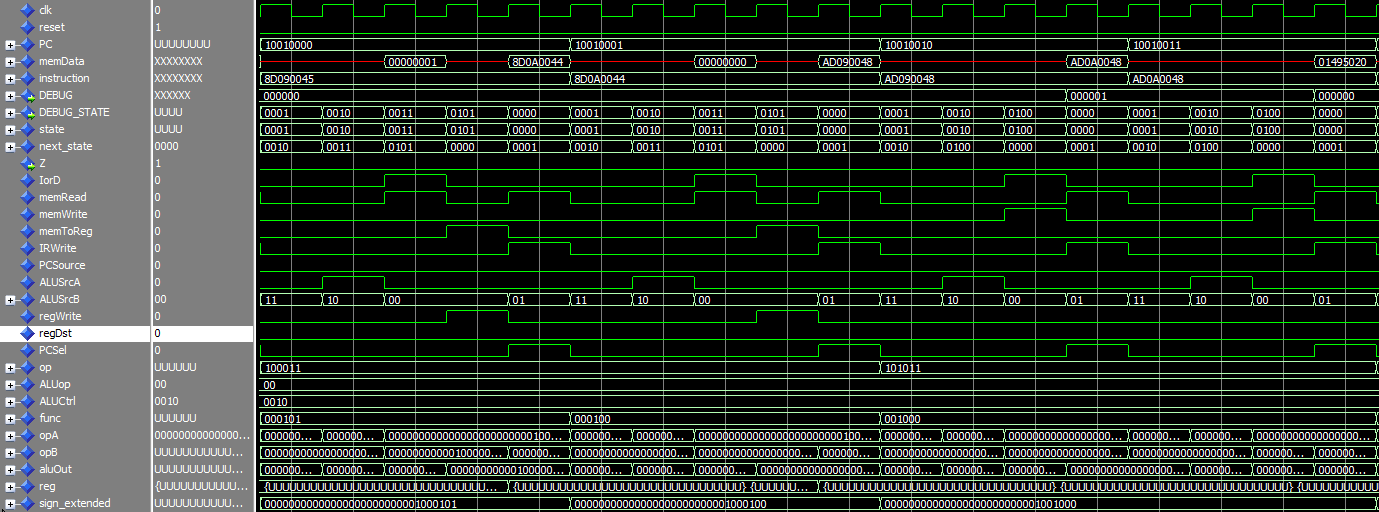
**Figure 20.** Top-Level MIPS Testbench Waveform (4/8)

****

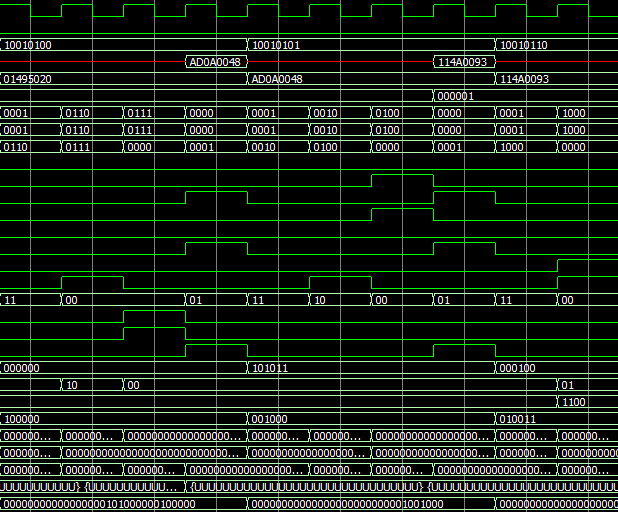
**Figure 21.** Top-Level MIPS Testbench Waveform (5/8)

****

**Figure 22.** Top-Level MIPS Testbench Waveform (6/8)

****

**Figure 23.** Top-Level MIPS Testbench Waveform (7/8)

****

**Figure 24.** Top-Level MIPS Testbench Waveform (8/8)

The following is an easier-to-read table view of the operations and their results. We excluded all other signals and states for brevity. Taking the first seven instructions as an example, one can clearly see that $t1 has the value 101000 and that $t2 has the value 000111. When the add instruction is performed and the result is stored to memory, the value in memory cell 0x0048 is the expected value of 101111.

|  |  |
| --- | --- |
| **MIPS Instruction** | **Hex Value in Memory Cell 0x0048** |
| ADD $t0 $zero $zero | 000001 |
| LW $t1 0x0047 $t0 | 000001 |
| SW $t1 0x0048 $t0 | 101000 |
| LW $t2 0x0046 $t0 | 101000 |
| SW $t2 0x0048 $t0 | 000111 |
| ADD $t3 $t1 $t2 | 000111 |
| SW $t3 0x0048 $t0 | 101111 |
| SUB $t3 $t1 $t2 | 101111 |
| SW $t3 0x0048 $t0 | 100EEF |
| AND $t3 $t1 $t2 | 100EEF |
| SW $t3 0x0048 $t0 | 000000 |
| OR $t3 $t1 $t2 | 000000 |
| SW $t3 0x0048 $t0 | 101111 |
| SLT $t3 $t1 $t2 | 101111 |
| SW $t3 0x0048 $t0 | 000000 |
| LW $t1 0x0045 $t0 | 000000 |
| LW $t2 0x0044 $t0 | 000000 |
| SW $t1 0x0048 $t0 | 000001 |
| SW $t2 0x0048 $t0 | 000000 |
| ADD $t2 $t2 $t1 | 000000 |
| SW $t2 0x0048 $t0 | 000001 |
| BEQ $t2 $t1 0xFFEB | Error |

**Table 4.** MIPS Instructions and Execution Results

**Synthesis:**

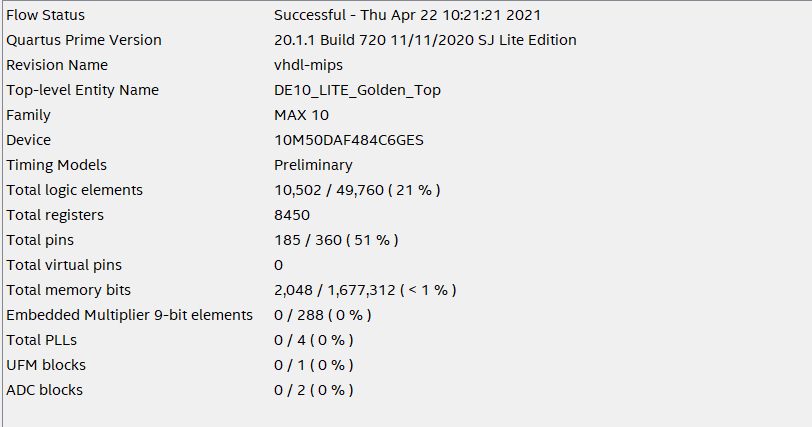
**1. Design and Overcoming Problems**

The synthesis phase was both challenging and rewarding. This is when some of our previous errors came to light and when novel issues were introduced. We had few trivial warnings and errors to solve before synthesizing. After they were patched, we quickly discovered that our synthesis was resulting in utilization of a single logic element. After research, we found that Quartus was optimizing our design down to nothing since we had only inputs and no outputs. To provide some output, we designated some of the onboard LEDs to display the *Z* flag and the current state of the control unit. This worked to our favor, since our design synthesized soon thereafter.

In addition to the 10 LEDs, the DE10-Lite board has six 7-segment LEDs. We decided that this would be the perfect display for some of our data as we are using the board. Our initial synthesis design placed the current state as 4 binary digits at the rightmost LEDs, the zero flag as the leftmost LED, and the state displayed in hexadecimal on the seven segment display. The reset signal is generated from switch zero. The clock signal is generated from switch one so that the user can make the design run at whatever pace he desires.

**2. Compilation Report**

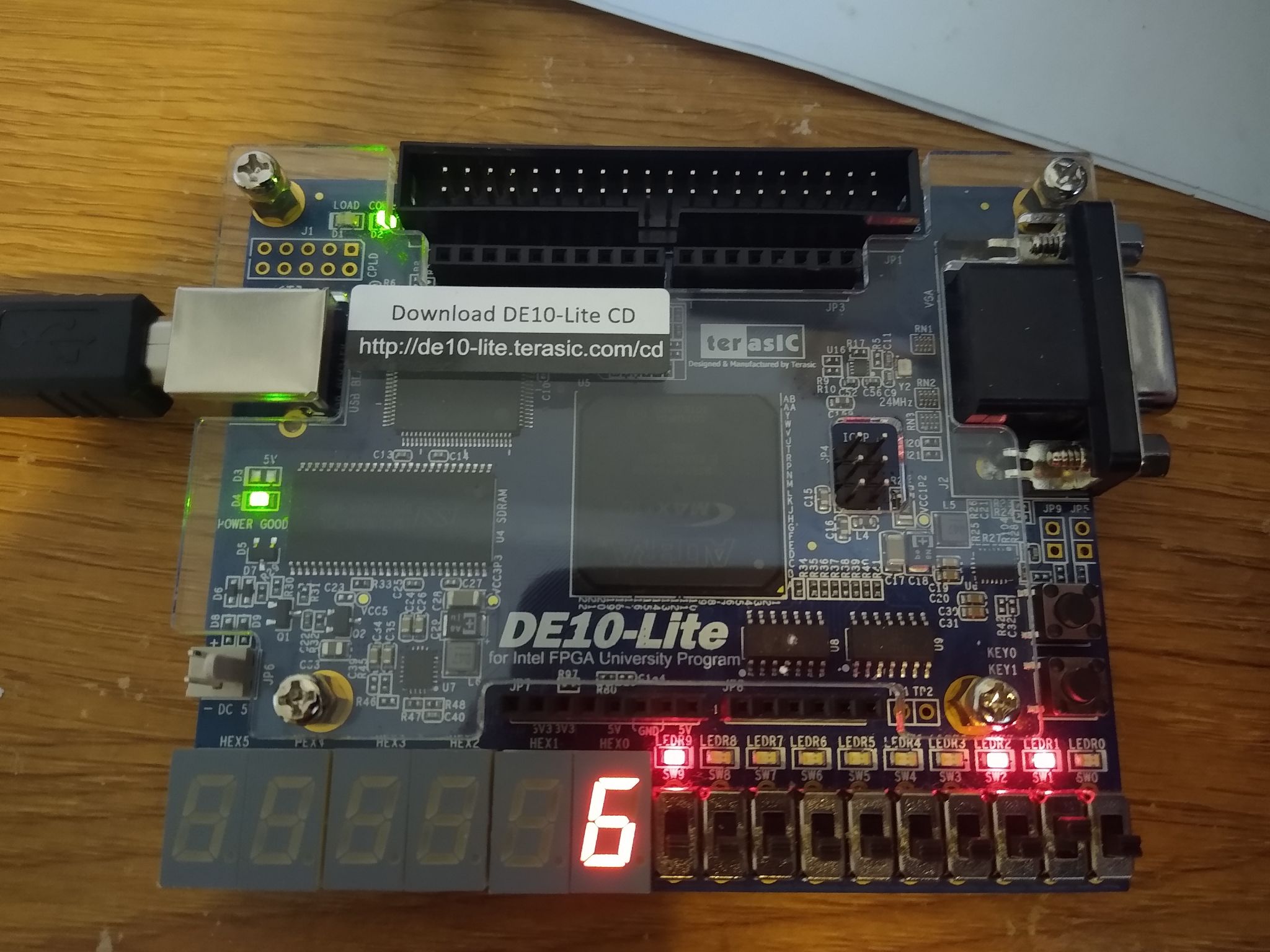
Our design used 21% of total logic elements available on the board but not even 1% of the total memory bits. This disparity was to be expected since only a small portion of our design holds any state. Quartus reported that we used 185 pins because we did not change any default settings related to the pin assignments. In reality, we used much less.

****

**Figure 25.** Compilation Report

**3. Design Results**

The initial design did not produce valid results, but it handled inputs and displayed the outputs as expected.

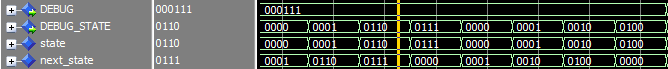


**Figure 26.** Initial Output Design. Outputs Shown are Execute State and Z flag set

Our second synthesis design placed the memory location 0x0048 on the seven segment display. Although we got our design to synthesize to the board properly, we cannot verify that it works the same on the board as it does in the simulation. This is because some of the output matched what we got in the simulation, but the majority of the output did not.



**Figure 27.** One Instance of Correct State and Memory Value



**Figure 28.** Corresponding Simulation Result

**Conclusion:**

Our team learned that perseverance is what kept our project from becoming a disaster. Initially, our project was going to be different. We quickly realized that we were in over our heads and had to change to our current project. This caused us to lose time. Without consistent testing and repeated searching for bugs, our design would not be functioning even in its current crippled state. We were able to successfully produce the expected results for ADD, SUB, AND, OR, SLT, LW, and SW instructions. Our design would clearly be remiss without implementing some form of a conditional branch instruction; therefore, we set out to create code to handle the Branch if Equal instruction. Although we wrote a lot of code for it, we were unable to successfully implement it. In addition to solving the remainder of our synthesis errors, we would dedicate more effort towards processing the BEQ instruction if we had more time to work on this project.

**References:**

Brady, A., n.d. *MIPS Instruction Formats*. [online] Max.cs.kzoo.edu. Available at: <https://max.cs.kzoo.edu/cs230/Resources/MIPS/MachineXL/InstructionFormats.html> [Accessed 28 April 2021].

Brady, A., n.d. *MIPS Instruction Formats*. [online] Max.cs.kzoo.edu. Available at: <https://max.cs.kzoo.edu/cs230/Resources/MIPS/MachineXL/InstructionFormats.html> [Accessed 29 April 2021].

**Appendix:**

**Models**

**Appendix 1.1:** DE10\_LITE\_Golden\_Top.vhd

|  |
| --- |
| *-- Copyright (C) 2017 Intel Corporation. All rights reserved.*  *-- Your use of Intel Corporation's design tools, logic functions*  *-- and other software and tools, and its AMPP partner logic*  *-- functions, and any output files from any of the foregoing*  *-- (including device programming or simulation files), and any*  *-- associated documentation or information are expressly subject*  *-- to the terms and conditions of the Intel Program License*  *-- Subscription Agreement, the Intel Quartus Prime License Agreement,*  *-- the Intel FPGA IP License Agreement, or other applicable license*  *-- agreement, including, without limitation, that your use is for*  *-- the sole purpose of programming logic devices manufactured by*  *-- Intel and sold by Intel or its authorized distributors. Please*  *-- refer to the applicable agreement for further details.*  library ieee;  use ieee.std\_logic\_1164.all;  library altera;  use altera.altera\_syn\_attributes.all;  entity DE10\_LITE\_Golden\_Top is  port  (  *-- {ALTERA\_IO\_BEGIN} DO NOT REMOVE THIS LINE!*  ADC\_CLK\_10 : in std\_logic;  MAX10\_CLK1\_50 : in std\_logic; *-- use clk*  MAX10\_CLK2\_50 : in std\_logic;  DRAM\_CAS\_N : in std\_logic;  DRAM\_CKE : in std\_logic;  DRAM\_CLK : in std\_logic;  DRAM\_CS\_N : in std\_logic;  DRAM\_LDQM : in std\_logic;  DRAM\_RAS\_N : in std\_logic;  DRAM\_UDQM : in std\_logic;  DRAM\_WE\_N : in std\_logic;  VGA\_HS : in std\_logic;  VGA\_VS : in std\_logic;  GSENSOR\_CS\_N : in std\_logic;  GSENSOR\_SCLK : in std\_logic;  GSENSOR\_SDI : in std\_logic;  GSENSOR\_SDO : in std\_logic;  ARDUINO\_RESET\_N : in std\_logic;  DRAM\_ADDR : in std\_logic\_vector(0 to 12);  DRAM\_BA : in std\_logic\_vector(0 to 1);  DRAM\_DQ : in std\_logic\_vector(0 to 15);  HEX0 : out std\_logic\_vector(7 downto 0);  HEX1 : out std\_logic\_vector(7 downto 0);  HEX2 : out std\_logic\_vector(7 downto 0);  HEX3 : out std\_logic\_vector(7 downto 0);  HEX4 : out std\_logic\_vector(7 downto 0);  HEX5 : out std\_logic\_vector(7 downto 0);  KEY : in std\_logic\_vector(0 to 1);  LEDR : inout std\_logic\_vector(0 to 9);  SW : in std\_logic\_vector(0 to 9); *-- use sw 0 as reset*  VGA\_B : in std\_logic\_vector(0 to 3);  VGA\_G : in std\_logic\_vector(0 to 3);  VGA\_R : in std\_logic\_vector(0 to 3);  GSENSOR\_INT : in std\_logic\_vector(1 to 2);  ARDUINO\_IO : in std\_logic\_vector(0 to 15);  GPIO : in std\_logic\_vector(0 to 35)  *-- {ALTERA\_IO\_END} DO NOT REMOVE THIS LINE!*  );  *-- {ALTERA\_ATTRIBUTE\_BEGIN} DO NOT REMOVE THIS LINE!*  *-- {ALTERA\_ATTRIBUTE\_END} DO NOT REMOVE THIS LINE!*  end DE10\_LITE\_Golden\_Top;  architecture ppl\_type of DE10\_LITE\_Golden\_Top is  *-- {ALTERA\_COMPONENTS\_BEGIN} DO NOT REMOVE THIS LINE!*  *-- {ALTERA\_COMPONENTS\_END} DO NOT REMOVE THIS LINE!*  signal outputtest : std\_logic\_vector(3 downto 0);  signal DEBUG\_STATE : std\_logic\_vector(3 downto 0);  signal DEBUG : std\_logic\_vector(23 downto 0);  begin  *-- {ALTERA\_INSTANTIATION\_BEGIN} DO NOT REMOVE THIS LINE!*  *-- {ALTERA\_INSTANTIATION\_END} DO NOT REMOVE THIS LINE!*  *--LEDR(0) <= outputtest(0);*  *--LEDR(1) <= outputtest(1);*  *--LEDR(2) <= outputtest(2);*  *--LEDR(3) <= outputtest(3);*  LEDR(0) <= DEBUG\_STATE(0);  LEDR(1) <= DEBUG\_STATE(1);  LEDR(2) <= DEBUG\_STATE(2);  LEDR(3) <= DEBUG\_STATE(3);  mips: entity work.mips(behave)  port map (SW(1), SW(0), outputtest, LEDR(9), DEBUG\_STATE, DEBUG);  display1: entity work.display\_driver(behave)  port map (DEBUG(3 downto 0), HEX0);  display2: entity work.display\_driver(behave)  port map (DEBUG(7 downto 4), HEX1);  display3: entity work.display\_driver(behave)  port map (DEBUG(11 downto 8), HEX2);  display4: entity work.display\_driver(behave)  port map (DEBUG(15 downto 12), HEX3);  display5: entity work.display\_driver(behave)  port map (DEBUG(19 downto 16), HEX4);  display6: entity work.display\_driver(behave)  port map (DEBUG(23 downto 20), HEX5);  end; |

**Appendix 1.2:** mips.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  entity mips is  port  (  clk : in std\_logic;  reset : in std\_logic;  ALUMirror : out std\_logic\_vector(3 downto 0);  Z : out std\_logic;  DEBUG\_STATE : out std\_logic\_vector(3 downto 0);  DEBUG : out std\_logic\_vector(23 downto 0)  );  end mips;  architecture behave of mips is  signal zero : std\_logic;  signal IorD : std\_logic;  signal memRead : std\_logic;  signal memWrite : std\_logic;  signal memToReg : std\_logic;  signal IRWrite : std\_logic;  signal PCSource : std\_logic;  signal ALUSrcA : std\_logic;  signal ALUSrcB : std\_logic\_vector(1 downto 0);  signal regWrite : std\_logic;  signal regDst : std\_logic;  signal PCSel : std\_logic;  signal op : std\_logic\_vector(5 downto 0);  signal ALUop : std\_logic\_vector(1 downto 0);  signal ALUCtrl : std\_logic\_vector(3 downto 0);  signal func : std\_logic\_vector(5 downto 0);  component datapath  port  (  clk : in std\_logic;  reset : in std\_logic;  IorD : in std\_logic;  memWrite : in std\_logic;  memRead : in std\_logic;  memToReg : in std\_logic;  IRWrite : in std\_logic;  PCSource : in std\_logic;  regDst : in std\_logic;  regWrite : in std\_logic;  PCSel : in std\_logic;  ALUSrcA : in std\_logic;  ALUSrcB : in std\_logic\_vector(1 downto 0);  ALUCtrl : in std\_logic\_vector(3 downto 0);  op : out std\_logic\_vector(5 downto 0);  zero : out std\_logic;  func : out std\_logic\_vector(5 downto 0);  DEBUG : out std\_logic\_vector(23 downto 0)  );  end component;  begin  dp : datapath  port map  (  clk=>clk, reset=>reset, IorD=>IorD, memWrite=>memWrite, memRead=>memRead, memToReg=>memToReg,  IRWrite=>IRWrite, PCSource=>PCSOurce, regDst=>regDst, regWrite=>regWrite, PCSel=>PCSel,  ALUSrcA=>ALUSrcA, ALUSrcB=>ALUSrcB, ALUCtrl=>ALUCtrl, op=>op, zero=>zero, func=>func, DEBUG=>DEBUG  );  ctrl : entity work.control(behave)  port map (  clk => clk,  rst => reset,  Z => zero,  Op => Op,  IorD => IorD,  MemWrite => MemWrite,  MemRead => MemRead,  MemtoReg => MemtoReg,  IRWrite => IRWrite,  PCSource => PCSource,  RegDst => RegDst,  RegWrite => RegWrite,  ALUSrcA => ALUSrcA,  PCSel => PCSel,  ALUOp => ALUOp,  ALUSrcB => ALUSrcB,  DEBUG\_STATE => DEBUG\_STATE  );  alu\_control: entity work.alu\_control(behave)  port map (ALUop, func, ALUCtrl);    ALUMirror <= ALUCtrl;  Z <= zero;  end architecture; |

**Appendix 1.3** control.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity control is  port  (  clk, rst, Z : in std\_logic;  Op : in std\_logic\_vector(5 downto 0);  IorD, MemWrite, MemRead,  MemtoReg, IRWrite, PCSource,  RegDst, RegWrite, ALUSrcA,  PCSel : out std\_logic;  ALUOp, ALUSrcB : out std\_logic\_vector(1 downto 0);  DEBUG\_STATE : out std\_logic\_vector(3 downto 0)  );  end control;  architecture behave of control is  *-- Local Variables*  signal PCWrite, PCWriteCond : std\_logic;  signal state, next\_state : std\_logic\_vector(3 downto 0);  *-- Control Constants*  constant IFETCH : std\_logic\_vector(3 downto 0) := "0000"; *-- instruction fetch*  constant IDECODE : std\_logic\_vector(3 downto 0) := "0001"; *-- instruction decode / register fetch*  constant ADDRESSCMP : std\_logic\_vector(3 downto 0) := "0010"; *-- memory address computation*  constant MEMLOAD : std\_logic\_vector(3 downto 0) := "0011"; *-- Memory Access 'load word'*  constant MEMSTORE : std\_logic\_vector(3 downto 0) := "0100"; *-- Memory Access 'store word'*  constant WRITEBACK : std\_logic\_vector(3 downto 0) := "0101"; *-- Memory Access finished*  constant EXECUTE : std\_logic\_vector(3 downto 0) := "0110";  constant RCOMPLETE : std\_logic\_vector(3 downto 0) := "0111"; *-- R-type completion*  constant BCOMPLETE : std\_logic\_vector(3 downto 0) := "1000"; *-- Branch completion*  begin  *-- advance PC*  PCSel <= (PCWrite or (PCWriteCond and Z));  DEBUG\_STATE <= state; *-- output to hex display*  *-- get next state*  process (clk) begin  if rising\_edge(clk) then  if(rst = '1') then  state <= IFETCH;  else  state <= next\_state;  end if;  end if;  end process;  *-- FSM next state*  process(state, Op) begin  case(state) is  when IFETCH => next\_state <= IDECODE;  when IDECODE =>  case(Op) is  when "000000" => next\_state <= EXECUTE;  when "000100" => next\_state <= BCOMPLETE;  when "100011" => next\_state <= ADDRESSCMP;  when "101011" => next\_state <= ADDRESSCMP;  when others => next\_state <= IFETCH;  end case;  when ADDRESSCMP =>  case(Op) is  when "100011" => next\_state <= MEMLOAD;  when "101011" => next\_state <= MEMSTORE;  when others => next\_state <= IFETCH;  end case;  when MEMLOAD => next\_state <= WRITEBACK;  when MEMSTORE => next\_state <= IFETCH;  when WRITEBACK => next\_state <= IFETCH;  when EXECUTE => next\_state <= RCOMPLETE;  when RCOMPLETE => next\_state <= IFETCH;  when BCOMPLETE => next\_state <= IFETCH;  when others => next\_state <= IFETCH; *-- when in doubt: IFETCH*  end case;  end process;  process(state) begin  *-- init to zero - eliminate carry-over errors*  IorD <= '0'; *-- instruction mode by default*  MemRead <= '0';  MemWrite <= '0';  MemtoReg <= '0';  IRWrite <= '0';  RegWrite <= '0';  RegDst <= '0';  PCSource <= '0';  PCWrite <= '0';  PCWriteCond <= '0';  ALUSrcB <= "00";  ALUSrcA <= '0';  ALUOp <= "00";  *-- FSM output*  case (state) is  when IFETCH =>  MemRead <= '1';  IRWrite <= '1';  PCWrite <= '1';  ALUSrcB <= "01";  when IDECODE => ALUSrcB <= "11";  when ADDRESSCMP =>  ALUSrcA <= '1';  ALUSrcB <= "10";  when MEMLOAD =>  MemRead <= '1';  IorD <= '1'; *-- switch to data*  when MEMSTORE =>  MemWrite <= '1';  IorD <= '1'; *-- switch to data*  when WRITEBACK =>  RegDst <= '0';  RegWrite <= '1';  MemtoReg <= '1';  when EXECUTE =>  ALUSrcA <= '1';  ALUOp <= "10";  when RCOMPLETE =>  RegDst <= '1';  RegWrite <= '1';  when BCOMPLETE =>  PCWriteCond <= '1';  PCSource <= '1';  ALUSrcA <= '1';  ALUOp <= "01";  when others => null;  end case;  end process;  end behave; |

**Appendix 1.4:** alu\_control.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity alu\_control is  port (  Op : in std\_logic\_vector(1 downto 0);  functField : in std\_logic\_vector(5 downto 0);  aluCtrl : out std\_logic\_vector(3 downto 0)  );  end alu\_control;  architecture behave of alu\_control is begin  process(Op, functField)  variable Op\_Field : std\_logic\_vector(7 downto 0);  begin  Op\_Field := Op & functField;  case? Op\_Field is  when "1---0000" => aluCtrl <= "0010"; *-- add*  when "1---0010" => aluCtrl <= "0110"; *-- sub*  when "1---0100" => aluCtrl <= "0000"; *-- and*  when "1---0101" => aluCtrl <= "0001"; *-- or*  when "1---1010" => aluCtrl <= "0111"; *-- slt*  when "00------" => aluCtrl <= "0010"; *-- lw and sw (does not go to alu)*  when "01------" => aluCtrl <= "1100"; *-- beq*  when others => aluCtrl <= "XXXX"; *-- 0101 ?*  end case?;  end process;  end behave; |

**Appendix 1.5:** datapath.vhd

|  |
| --- |
| use work.all;  library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.numeric\_std.all;  entity datapath is  port  (  clk : in std\_logic; *-- input clock*  reset : in std\_logic; *-- reset switch*  IorD : in std\_logic; *-- determine if you are reading from instruction or data memory*  memWrite : in std\_logic; *-- signal to write to registers*  memRead : in std\_logic; *-- signal to read from registers*  memToReg : in std\_logic; *-- determines where the value to be written comes from*  IRWrite : in std\_logic;  PCSource : in std\_logic;  regDst : in std\_logic; *-- determines how the destination reg is specified*  regWrite : in std\_logic; *-- enables writing to a register*  PCSel : in std\_logic;  ALUSrcA : in std\_logic;  ALUSrcB : in std\_logic\_vector(1 downto 0);  ALUCtrl : in std\_logic\_vector(3 downto 0);  op : out std\_logic\_vector(5 downto 0);  zero : out std\_logic;  func : out std\_logic\_vector(5 downto 0);  DEBUG : out std\_logic\_vector(23 downto 0) *-- for hex display*  );  end datapath;  architecture behave of datapath is  signal instruction : std\_logic\_vector(31 downto 0); *-- instruction that is being executed*  signal PC : std\_logic\_vector(7 downto 0); *-- program counter*  signal aluOut : std\_logic\_vector(31 downto 0);  signal address : std\_logic\_vector(7 downto 0);  signal memData : std\_logic\_vector(31 downto 0); *-- holds data read from memory*  signal A : std\_logic\_vector(31 downto 0); *-- intermidiate register*  signal B : std\_logic\_vector(31 downto 0); *-- intermidiate register*  signal ALUResult : std\_logic\_vector(31 downto 0);  type register\_block is array (0 to 31) of std\_logic\_vector(31 downto 0); *-- define register block type*  signal reg : register\_block;  signal mdr : std\_logic\_vector(31 downto 0); *-- used for transfering data to instruction opcodes*  signal da : std\_logic\_vector(31 downto 0); *-- data read a*  signal db : std\_logic\_vector(31 downto 0); *-- data read b*  signal opA : std\_logic\_vector(31 downto 0);  signal opB : std\_logic\_vector(31 downto 0);  signal sign\_extended : std\_logic\_vector(31 downto 0);  component sign\_extend  port  (  in\_bits : in std\_logic\_vector(15 downto 0);  out\_bits : out std\_logic\_vector(31 downto 0)  );  end component;  component alu  port  (  opA : in std\_logic\_vector(31 downto 0);  opB : in std\_logic\_vector(31 downto 0);  aluOp : in std\_logic\_vector(3 downto 0);  result : out std\_logic\_vector(31 downto 0)  );  end component;  component memory  port  (  clk : in std\_logic;  reset : in std\_logic;  memWrite : in std\_logic;  memRead : in std\_logic;  address : in std\_logic\_vector(7 downto 0);  B : in std\_logic\_vector(31 downto 0);  memData : out std\_logic\_vector(31 downto 0);  DEBUG : out std\_logic\_vector(23 downto 0) *-- for hex display*  );  end component;  component program\_counter  generic(PCSTART : integer := 128);  port  (  clk : in std\_logic;  reset : in std\_logic;  PCsel : in std\_logic;  PCSource : in std\_logic;  ALUResult : in std\_logic\_vector(31 downto 0);  ALUOut : in std\_logic\_vector(31 downto 0);  PC : out std\_logic\_vector(7 downto 0)  );  end component;  begin  func <= instruction(5 downto 0); *-- assign func field*  op <= instruction(31 downto 26); *-- assign op field*  address <= aluOut(7 downto 0) when (IorD = '1') else PC; *-- needs verifying!!!*  memory\_inst : memory  port map  (  clk=>clk,  reset=>reset,  memWrite=>memWrite,  memRead=>memRead,  address=>address,  B=>B,  memData=>memData,  DEBUG=>DEBUG  );    *-- PC logic*  PC\_inst : program\_counter  port map  (  clk=>clk, reset=>reset, PCSel=>PCSel,  PCSource=>PCSource, ALUResult=>ALUResult,  ALUOut=>ALUOut, PC=>PC  );  instruction\_register\_write : process(clk) begin *-- get instruction*  if rising\_edge(clk) then  if(IRWrite = '1') then  instruction <= memData;  end if;  end if;  end process;  memory\_data\_to\_reg : process(clk) begin  if rising\_edge(clk) then  mdr <= memData;  end if;  end process;  da <= reg(to\_integer(unsigned(instruction(25 downto 21)))) when (to\_integer(unsigned(instruction(25 downto 21))) /= 0)  else x"00000000";  db <= reg(to\_integer(unsigned(instruction(20 downto 16)))) when (to\_integer(unsigned(instruction(20 downto 16))) /= 0)  else x"00000000";  reg\_dst : process(clk) begin  if rising\_edge(clk) then  if(regWrite = '1') then  if(regDst = '1') then  if(memToReg = '1') then  reg(to\_integer(unsigned(instruction(15 downto 11)))) <= mdr;  else  reg(to\_integer(unsigned(instruction(15 downto 11)))) <= ALUOut;  end if;  else  if(memToReg = '1') then  reg(to\_integer(unsigned(instruction(20 downto 16)))) <= mdr;  else  reg(to\_integer(unsigned(instruction(20 downto 16)))) <= ALUOut;  end if;  end if;  end if;  end if;  end process;  a\_reg : process(clk) begin  if rising\_edge(clk) then  A <= da;  end if;  end process;  b\_reg : process(clk) begin  if rising\_edge(clk) then  B <= db;  end if;  end process;  opA <= A when (ALUSrcA = '1') else std\_logic\_vector(resize(unsigned(PC), opA'length));  *-- sign extension*  se : sign\_extend  port map  (  in\_bits => instruction(15 downto 0),  out\_bits => sign\_extended  );  opB\_process : process(ALUSrcB, B, instruction(15 downto 0), sign\_extended)  begin  case? ALUSrcB is  when "00" => opB <= B; *-- add*  when "01" => opB <= x"00000001"; *-- sub*  when "1-" => opB <= sign\_extended; *-- and*  when others => opB <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"; *-- 0101 ?*  end case?;  end process;  zero <= '1' when (ALUResult = x"00000000") else '0';    *-- alu logic*  alu\_ints : alu  port map  (  opA => opA,  opB => opB,  aluOp => aluCtrl,  result => ALUResult  );  ALUOut\_to\_ALUResult : process(clk)  begin  if rising\_edge(clk) then  ALUOut <= ALUResult;  end if;  end process;  end architecture; |

**Appendix 1.6:** memory.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.numeric\_std.all;  entity memory is  port  (  clk : in std\_logic;  reset : in std\_logic;  memWrite : in std\_logic;  memRead : in std\_logic;  address : in std\_logic\_vector(7 downto 0);  B : in std\_logic\_vector(31 downto 0);  memData : out std\_logic\_vector(31 downto 0);  DEBUG : out std\_logic\_vector(23 downto 0) *-- for hex display*  );  end memory;  architecture behave of memory is  type memory\_block is array (0 to 255) of std\_logic\_vector(31 downto 0); *-- define memory block type*  signal mem : memory\_block; *-- data and instruction memory*  begin  *-- send debug to display*  DEBUG <= mem(72)(23 downto 0);  write\_to\_mem : process(clk, reset) begin *-- handle writes to memory*  if rising\_edge(clk) then  if reset = '1' then *-- power on reset*  mem(0) <= x"00000001"; *-- data memory*  mem(1) <= x"00000000";  mem(2) <= x"00000000";  mem(3) <= x"00000000";  mem(4) <= x"00000001";  mem(5) <= x"00000000";  mem(6) <= x"00000000";  mem(7) <= x"00000000";  mem(8) <= x"00000001";  mem(9) <= x"00000000";  mem(10) <= x"00000000";  mem(11) <= x"00000000";  mem(12) <= x"00000001";  mem(13) <= x"00000000";  mem(14) <= x"00000000";  mem(15) <= x"00000000";  mem(68) <= x"00000000";  mem(69) <= x"00000001";  mem(70) <= x"00000111";  mem(71) <= x"00101000";  mem(72) <= x"00000001";  mem(128) <= x"00004020"; *-- instruction memory*  mem(129) <= x"8D090047";  mem(130) <= x"AD090048";  mem(131) <= x"8D0A0046";  mem(132) <= x"AD0A0048";  mem(133) <= x"012A5820";  mem(134) <= x"AD0B0048";  mem(135) <= x"012A5822";  mem(136) <= x"AD0B0048";  mem(137) <= x"012A5824";  mem(138) <= x"AD0B0048";  mem(139) <= x"012A5825";  mem(140) <= x"AD0B0048";  mem(141) <= x"012A582A";  mem(142) <= x"AD0B0048";  mem(143) <= x"8D090045";  mem(144) <= x"8D0A0044";  mem(145) <= x"AD090048";  mem(146) <= x"AD0A0048";  mem(147) <= x"01495020";  mem(148) <= x"AD0A0048";  mem(149) <= x"114A0093";  elsif(memWrite = '1') then  mem(to\_integer(unsigned(address))) <= B;  end if;  end if;  end process;  memData <= mem(to\_integer(unsigned(address))) when (memRead = '1') else "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"; *-- read from memory*  end architecture; |

**Appendix 1.7:** program\_counter.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.numeric\_std.all;  entity program\_counter is  generic(PCSTART : integer := 128);  port  (  clk : in std\_logic;  reset : in std\_logic;  PCsel : in std\_logic;  PCSource : in std\_logic;  ALUResult : in std\_logic\_vector(31 downto 0);  ALUOut : in std\_logic\_vector(31 downto 0);  PC : out std\_logic\_vector(7 downto 0)  );  end program\_counter;  architecture behave of program\_counter is  begin  process(clk) begin  if rising\_edge(clk) then  if(reset = '1') then  PC <= std\_logic\_vector(to\_unsigned(PCSTART, PC'length)); *-- go back to the original instruction*  else  if(PCsel = '1') then  case(PCSource) is  when '0' => PC <= ALUResult(7 downto 0); *-- direct line from alu*  when '1' => PC <= ALUOut(7 downto 0); *-- intermediate register*  when others => PC <= "XXXXXXXX";  end case;  end if;  end if;  end if;  end process;  end architecture; |

**Appendix 1.8:** sign\_extend.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.numeric\_std.all;  entity sign\_extend is *-- sign extend for 16 to 32 bits*  port  (  in\_bits : in std\_logic\_vector(15 downto 0);  out\_bits : out std\_logic\_vector(31 downto 0)  );  end sign\_extend;  architecture behav of sign\_extend is  begin  out\_bits <= std\_logic\_vector(resize(signed(in\_bits), out\_bits'length));  end architecture; |

**Appendix 1.9:** alu.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity alu is  port  (  opA, opB : in std\_logic\_vector(31 downto 0); *-- the inputs*  aluOp : in std\_logic\_vector(3 downto 0); *-- what operation to perform*  result : out std\_logic\_vector(31 downto 0) *-- the end result*  *-- maybe add zero flag*  );  end entity alu;  architecture behave of alu is begin  process(aluOp, opA, opB) begin  case aluOp is  when "0000" => result <= opA and opB; *-- and*  when "0001" => result <= opA or opB; *-- or*  when "0010" => result <= std\_logic\_vector(to\_signed(to\_integer(signed(opA)) + to\_integer(signed(opB)), 32)); *-- add*  when "0110" => result <= std\_logic\_vector(to\_signed(to\_integer(signed(opA)) - to\_integer(signed(opB)), 32)); *-- subtract*  when "0111" =>  if(opA < opB) then  result <= "11111111111111111111111111111111"; *-- set on less than*  else result <= "00000000000000000000000000000000";  end if;  when "1100" => result <= not (opA or opB); *-- branch if equal*  when others => result <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";  end case;  end process;  end behave; |

**Appendix 1.10:** display\_driver.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  entity display\_driver is  port  (  Input : in std\_logic\_vector(3 downto 0);  Output : out std\_logic\_vector(7 downto 0));  end display\_driver;  *-- convert byte to display on the 7 segment LEDs*  architecture behave of display\_driver is  begin  process (Input) begin  case Input is  when "0000" => Output <= x"C0";  when "0001" => Output <= x"F9";  when "0010" => Output <= x"A4";  when "0011" => Output <= x"B0";  when "0100" => Output <= x"99";  when "0101" => Output <= x"92";  when "0110" => Output <= x"82";  when "0111" => Output <= x"F8";  when "1000" => Output <= x"80";  when "1001" => Output <= x"90";  when "1010" => Output <= x"88";  when "1011" => Output <= x"83";  when "1100" => Output <= x"C6";  when "1101" => Output <= x"A1";  when "1110" => Output <= x"86";  when "1111" => Output <= x"8E";  when others => Output <= x"FF"; *-- all high means all off*  end case;  end process;  end behave; |

**Test Benches**

**Appendix 2.1:** mips\_tb.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  entity mips\_tb is  end mips\_tb;  architecture testbench of mips\_tb is  signal clk : std\_logic := '0';  signal reset : std\_logic;  component mips  port  (  clk : in std\_logic;  reset : in std\_logic  );  end component;  begin  dut : mips port map (clk=>clk, reset=>reset);  process(clk)  begin  clk <= not clk after 5 ns;  end process;    stimuli : process  begin  reset <= '1';  wait for 10 ns;  reset <= '0';  wait;  end process;  end testbench; |

**Appendix 2.2:** alu\_tb.vhd

|  |
| --- |
| use WORK.all;  library ieee;  use IEEE.std\_logic\_1164.all;  entity ALU\_TEST\_BENCH is  end ALU\_TEST\_BENCH;  architecture TEST of ALU\_TEST\_BENCH is  *-- alu*  signal op1, op2 : std\_logic\_vector(31 downto 0);  signal aluOp : std\_logic\_vector(3 downto 0);  signal result : std\_logic\_vector(31 downto 0);  signal zeroFlag : std\_logic;  *-- alu control*  signal Op : std\_logic\_vector(1 downto 0);  signal functField : std\_logic\_vector(5 downto 0);  begin  alu : entity work.alu(behave)  port map (op1, op2, aluOp, result);  control: entity work.alu\_control(behave)  port map (Op, functField, aluOp);  process begin  *-- initialize alu operations*  op1 <= "00000000000000000000000000000011";  op2 <= "00000000000000000000000000001010";  *-- test alu control ( run for 11 ns )*  wait for 1 ns;  Op <= "10";  functField <= "000000";  wait for 1 ns;  Op <= "11";  functField <= "000010";  wait for 1 ns;  Op <= "10";  functField <= "000100";  wait for 1 ns;  Op <= "11";  functField <= "000101";  wait for 1 ns;  Op <= "10";  functField <= "001010";  wait for 1 ns;  Op <= "00";  functField <= "000000";  wait for 1 ns;  Op <= "01";  functField <= "000000";  wait for 1 ns;  *-- when others*  Op <= "11";  functField <= "111111";  wait for 1 ns;  Op <= "10";  functField <= "111100";  wait for 1 ns;  end process;  end TEST; |

**Appendix 2.3:** control\_tb.vhd

|  |
| --- |
| use WORK.all;  library ieee;  use IEEE.std\_logic\_1164.all;  entity CONTROL\_TEST\_BENCH is  end CONTROL\_TEST\_BENCH;  architecture TEST of CONTROL\_TEST\_BENCH is  signal IorD, MemWrite, MemRead,  MemtoReg, IRWrite, PCSource,  RegDst, RegWrite, ALUSrcA,  PCSel : std\_logic;  signal clk : std\_logic := '0';  signal rst, Z : std\_logic;  signal Op : std\_logic\_vector(5 downto 0);  signal ALUOp, ALUSrcB : std\_logic\_vector(1 downto 0);  begin  control : entity work.control(behave)  port map (  clk => clk,  rst => rst,  Z => Z,  Op => Op,  IorD => IorD,  MemWrite => MemWrite,  MemRead => MemRead,  MemtoReg => MemtoReg,  IRWrite => IRWrite,  PCSource => PCSource,  RegDst => RegDst,  RegWrite => RegWrite,  ALUSrcA => ALUSrcA,  PCSel => PCSel,  ALUOp => ALUOp,  ALUSrcB => ALUSrcB  );  process(clk) begin  clk <= not clk after 1 ns;  end process;  *-- run for 52 ns*  process begin  wait for 1 ns;  rst <= '1';  wait for 1 ns;  rst <= '0';  Op <= "100011";  Z <= '0';  wait for 10 ns; *-- 5 clk pulses - max*  Op <= "101011";  wait for 10 ns;  Op <= "000000";  wait for 10 ns;  Op <= "000100";  wait for 10 ns;  rst <= '1';  wait for 10 ns;  wait;  end process;  end TEST; |

**Appendix 2.4:** datapath\_tb.vhd

|  |
| --- |
| use WORK.all;  library ieee;  use IEEE.std\_logic\_1164.all;  entity DATAPATH\_TEST\_BENCH is  end DATAPATH\_TEST\_BENCH;  architecture TEST of DATAPATH\_TEST\_BENCH is  signal IorD, MemWrite, MemRead,  MemtoReg, IRWrite, PCSource,  RegDst, RegWrite, ALUSrcA,  PCSel : std\_logic;  signal clk : std\_logic := '0';  signal rst, Z : std\_logic;  signal Op, func : std\_logic\_vector(5 downto 0);  signal ALUOp, ALUSrcB : std\_logic\_vector(1 downto 0);  signal ALUCtrl : std\_logic\_vector(3 downto 0);1  begin  *--control: entity work.alu\_control(behave)*  *-- port map (Op, functField, aluOp);*  dp : entity work.datapath(behave)  port map (  clk=>clk, reset=>rst, IorD=>IorD, memWrite=>memWrite, memRead=>memRead, memToReg=>memToReg,  IRWrite=>IRWrite, PCSource=>PCSOurce, regDst=>regDst, regWrite=>regWrite, PCSel=>PCSel,  ALUSrcA=>ALUSrcA, ALUSrcB=>ALUSrcB, ALUCtrl=>ALUCtrl, op=>Op, zero=>Z, func=>func  );  *-- instantiate clk*  process(clk) begin  clk <= not clk after 1 ns;  end process;  *-- run for 52 ns*  process begin  wait for 1 ns;  rst <= '1';  wait for 1 ns;  rst <= '0';  IorD = '1'; *-- address gets aluOut*  IRWrite <= '1';    wait for 2 ns; *-- 5 clk pulses - max*  Op <= "101011";  wait for 10 ns;  Op <= "000000";  wait for 10 ns;  Op <= "000100";  wait for 10 ns;  rst <= '1';  wait for 10 ns;  wait;  end process;  end TEST; |

**Appendix 2.5:** program\_counter.vhd

|  |
| --- |
| use WORK.all;  library ieee;  use IEEE.std\_logic\_1164.all;  entity PROGRAM\_COUNTER\_TEST\_BENCH is  end PROGRAM\_COUNTER\_TEST\_BENCH;  architecture TEST of PROGRAM\_COUNTER\_TEST\_BENCH is  signal clk : std\_logic := '0';  signal reset : std\_logic;  signal PCsel : std\_logic;  signal PCSource : std\_logic;  signal ALUResult : std\_logic\_vector(31 downto 0);  signal ALUOut : std\_logic\_vector(31 downto 0);  signal PC : std\_logic\_vector(7 downto 0);  begin  *--control: entity work.alu\_control(behave)*  *-- port map (Op, functField, aluOp);*  pc\_inst : entity work.program\_counter(behave)  port map (  clk=>clk, reset=>reset, PCSel=>PCSel,  PCSource=>PCSource, ALUResult=>ALUResult,  ALUOut=>ALUOut, PC=>PC  );  *-- instantiate clk*  process(clk) begin  clk <= not clk after 1 ns;  end process;  ALUResult <= x"0000000a";  ALUOut <= x"0000000b";  process begin  reset <= '1';  wait for 2 ns;  reset <= '0';  PCsel <= '1';  PCSource <= '0';  wait for 4 ns;  PCsel <= '1'; *-- address gets aluOut*  PCSource <= '1';  wait for 4 ns;  PCsel <= '0';  wait for 4 ns;  reset <= '1';  wait;  end process;  end TEST; |

**Appendix 2.6:** sign\_extend\_tb.vhd

|  |
| --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.numeric\_std.all;  entity sign\_extend\_tb is  end sign\_extend\_tb;  architecture test of sign\_extend\_tb is  component sign\_extend  port  (  in\_bits : in std\_logic\_vector(15 downto 0);  out\_bits : out std\_logic\_vector(31 downto 0)  );  end component;  signal in\_bits : std\_logic\_vector(15 downto 0);  signal out\_bits : std\_logic\_vector(31 downto 0);  begin  uut : sign\_extend  port map  (  in\_bits => in\_bits,  out\_bits => out\_bits  );  process begin  in\_bits <= x"0001";  wait for 1 ns;  in\_bits <= x"FFFF";  wait for 1 ns;  in\_bits <= "1000000000000000";  wait for 1 ns;    wait;  end process;  end test; |