New Paltz
STATE UNIVERSITY OF NEW YORK

Implementation of 32-Bit MIPS CPU on PYNQ-Z1 FPGA

EGC 455 System-On-Chip

George Dagis (CE)

December 17th, 2018

Instructor: Yi-Chung Chen

Abstract

This project outlines the bottom-up design of a MIPS processor as well as its implementation onto a PYNQ-Z1 board. The MIPS processor discussed within the project is designed in verilog and has been simulated on EDA Playground [1] in order to test functionality prior to implementing onto the PYNQ board. After debugging the processor, it was then implemented onto the PYNQ board using Xilinx Vivado Design Suite 2018.3 [2]. The MIPS processor code was altered in order to light up LEDs corresponding to certain values outputted by the ALU. The group used the Software Development Kit (SDK) in order to create an First Stage Boot Loader (FSBL) which allowed for the code to run off an SD card rather than through the USB port.

Table of Contents

1. Introduction

The purpose of this project was to implement a 32-bit MIPS (Microprocessor without Interlocked Pipeline Stages) CPU on a PYNQ-Z1 FPGA board. The 32-bit MIPS CPU was built with the ability to perform addition, subtraction, and/or, lw/sw and beq functions. It is important to recall the architecture of a 32 bit MIPS CPU before attempting to build and implement one. The architecture of a standard MIPS CPU is depicted in the diagram seen in Figure 1, below.

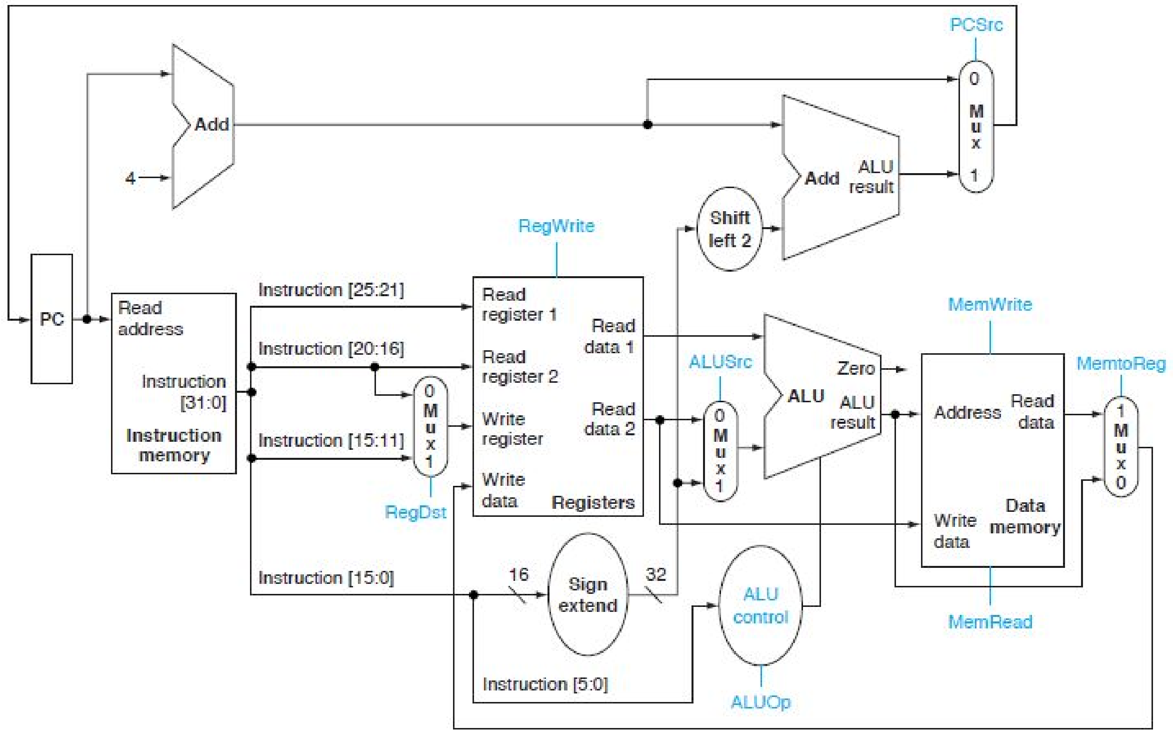


Figure 1: MIPS Architecture. Reprinted from https://stackoverflow.com/. Copyright 2018.

The PYNQ board used throughout the project is an FPGA designed for use with the PYNQ open-source framework. It comes fully equipped with a 650 MHz dual-core Cortex-A9 processor, 512 MB DDR3, HDMI ports, USB port, Ethernet capabilities, and many UART and I/O pins [3]. A picture of the PYNQ-Z1 board is shown in Figure 2, on the next page.

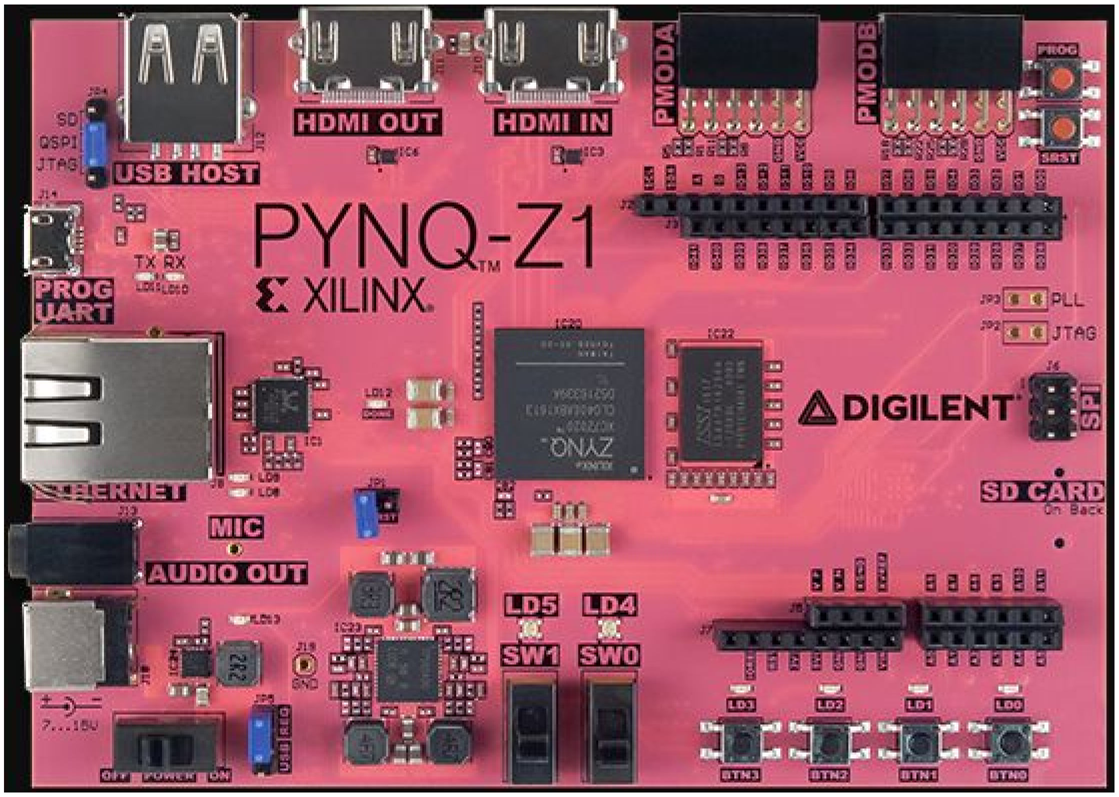


Figure 2: PYNQ-Z1 Board. Reprinted from https://reference.digilentinc.com/. Copyright 2018.

2. Design Procedure

2.1 Design Methodology

The overall design of the processor is organized in such a way that each block found within the MIPS Architecture diagram (Figure 1, above), has its own module.

2.2 Design of Program Counter

The group thought it would be best to start with the program counter as it's the first block sequentially when looking from left to right in the architecture diagram previously mentioned. The program counter is broken up into 3 separate modules. These modules are named "programcounter", "pcadder" and "PC_MUX."

The programcounter module is responsible for updating the program counter value, "currentPC" with the updated program counter value, appropriately called 'updatedPC' at every positive clock edge. The program counters' updated value is derived from the other two modules, pcadder and PC_MUX.

The pcadder module takes the original program counter value and increments it. This is done to point to the next instruction to be executed in the instruction memory. This updated value, "PC_plus4" is then used in the PC_MUX module.

The PC_MUX module takes PC_plus4 as well as the output of the "Add ALU (Figure 2, above)" as inputs. The select bit of this MUX is handled within a positive clock edge triggered always block which subtracts input_B from input_A. The zero and branch flags are then analyzed and output the proper output which is then fed back to the original programcounter module. The output is sent to both the pcadder and the register file where the process will repeat.

2.3 Design of Instruction Memory

The instruction memory was designed in two separate modules labeled "instructionmemory" and "instructionformat."

The instructionmemory module has input variables clk, currentPC and an output variable "instruction." A register called "instructionreg" exists to store 7 instructions. These instructions are add, subtract, and, or, beq, lw and sw. The instructions are separated by type in the code. The standard instruction format for MIPS seen in Figure 3, on the next page, was used as a reference, as well as the standard opcode & function values.

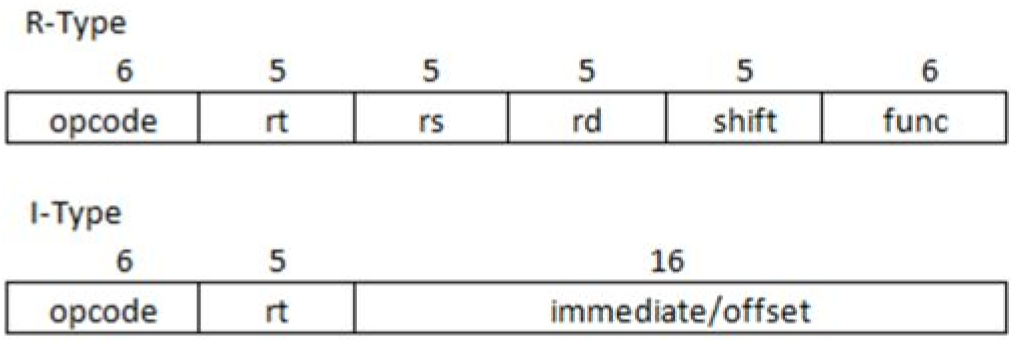
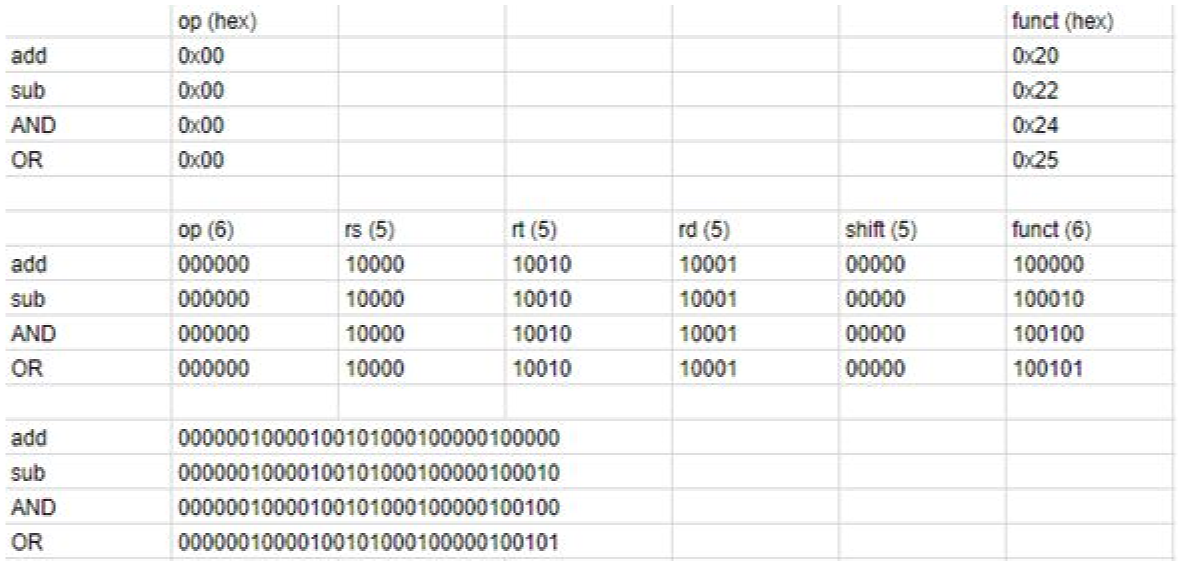


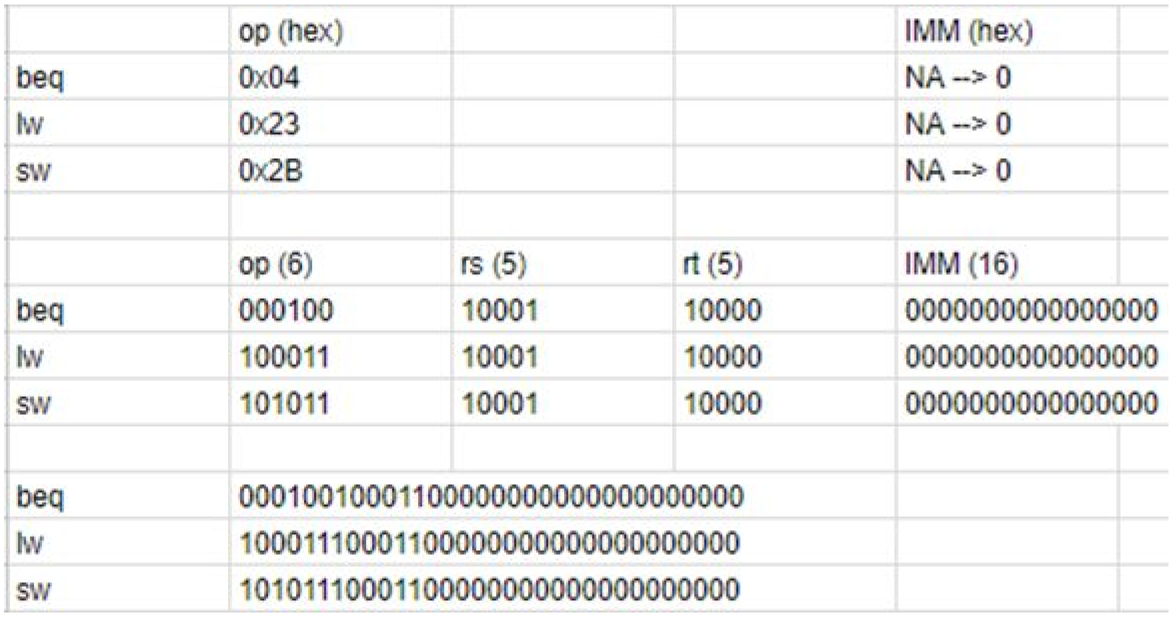
Figure 3: R and I-type instruction format for MIPS. Reprinted from www.researchgate.net/. Copyright 2018.

Using the above information, the group created corresponding 32 bit instructions piece by piece. The methodology can be seen for R-type and I-type instructions in Figures 4 and 5 respectively. The group decided to use the registers 16, 17 and 18 as they are the first three "save registers" in MIPS machines [5]. For R-type instructions we designated r16 to the source register, r17 for the destination register and r18 for the target register. For I-type instructions we designated r16 to the target register and r17 for the source register.



	op (hex)					funct (hex)
add	0x00					0x20
sub	0x00					0x22
AND	0x00					0x24
OR	0x00					0x25
	op (6)	rs (5)	rt (5)	rd (5)	shift (5)	funct (6)
add	000000	10000	10010	10001	00000	100000
sub	000000	10000	10010	10001	00000	100010
AND	000000	10000	10010	10001	00000	100100
OR	000000	10000	10010	10001	00000	100101
add	00000010000100101000100000100000					
sub	00000010000100101000100000100010					
AND	00000010000100101000100000100100					
OR	00000010000100101000100000100101					

Figure 4: Methodology to Create 32 bit R-type instructions.



	op (hex)			IMM (hex)	
beq	0x04			NA --> 0	
lw	0x23			NA --> 0	
sw	0x2B			NA --> 0	
	op (6)	rs (5)	rt (5)	IMM (16)	
beq	000100	10001	10000	0000000000000000	
lw	100011	10001	10000	0000000000000000	
sw	101011	10001	10000	0000000000000000	
beq	00010010001100000000000000000000				
lw	10001110001100000000000000000000				
sw	10101110001100000000000000000000				

Figure 5: Methodology to Create 32 bit I-type instructions.

After the instruction memory receives an address fed from the program counter, it points to its respective instruction. This is done in an always block triggered by the positive edge of a clock.

The second module used for instruction memory, "instructionformat" is used to separate the parts of each instruction in order for the ALU to process them better. The module takes the 32 bit instruction and outputs the opcode, rs, rt, rd, shift, function and immediate values as they are applicable. The module also has an always block which is triggered when a different instruction is read. The block is responsible for outputting rd/shift/function values in the cause of R instructions and an immediate value in the case of an I instruction. The block checks if the opcode is an R or I instruction by comparing it to 6 bits of 0. It then outputs the unique values specific to that type of instruction.

2.4 Design of Register File

The register file is mostly handled with one module simply named "registerfile." The module is initialized with 32 registers all 32 bits in size. All bits are initially set to 0 using a begin statement. The group created a reset variable which is handled within an if statement. The statement takes register memory from values 0 to 31 and replaces them all with 32 bits of 0. Another if statement exists within the registerfile module, in order to control overwrite privileges. The statement is triggered when input variable writeenable is logic 1. The new data is taken from the output of the WB_MUX module [See section 2.6 for more detail].

2.5 Design of ALU

The ALU is all handled within one module, 'ALU.' Despite this, the first module to be discussed in this section is the module 'signextend' as it is the most appropriate time to discuss it.

The signextend module simply takes a 16 bit value, 'unextended' and converts it to a new 'extended' 32-bit value. This is done by taking the sign bit of the original value and filling a 16 bit register with that value in every bit. This register is then concatenated with the original 16 bit unextended value to create a new 32 bit value. The original unextended value remains in the lower significant half. Extension is done this way to maintain sign.

The ALU module is where all functions are executed on the two 32-bit inputs. These two values are initialized here and called 'input_A' and 'input_B.' The 6 bit opcode and 6 bit function values, created in the instructionformat module, are listed as inputs for the ALU module and is used to inform the ALU of the type of instruction to perform. The opcode is initially compare to

6'b000000 within an if statement in order to determine whether it is an R instruction. If not equal to 6'b000000 it must be an I instruction. In both cases there are nested if statements in order to further specify the instruction by identifying the 6 bit function value. Lastly, an always statement was created in order to enable writing ability every time the aluresult value changes.

2.6 Design of Data Memory

Lastly, the data memory of the 32 bit MIPS CPU was created. This was done through the creation of two modules labeled "datamemory" and "WB_MUX."

The datamemory module was created in order to temporarily store results from the ALU through the use of 32 registers. All memory is initially filled with 0 to begin with. The datamemory module also contains a simple always block which is triggered on the positive edge of the clock. This statement checks to see if reset is logic 1, and then sets all memory in the registers to 0 with a for statement. The statement is similar to the one in the registermemory module. Lastly, this block checks to see the value of writeenable, and overwrites the memory with new data from the aluresult value if writeenable is logic 1.

The final module of the design is called WB_MUX and is responsible for taking input data from the ALU as well as from datamemory and passing the appropriate input to the registerfile. This logic is done within an always statement triggered by a change of instruction. It checks to see the type of instruction by comparing its opcode to 0. If an R-type instruction, the aluresult output is written to the registerfile. Otherwise the data from datamemory is passed.

Some changes were made in order to communicate with the I/O on the PYNQ board. The three modules added to are the instructionmemory, programcounter, and ALU modules.

The instruction memory module was modified in order to execute a series of add instructions which then correspond to a the linear sequence of the four LEDs. Secondly, the program counter was modified in order to reset the sequence after hitting the last LED. Lastly, within the ALU, the output_LEDs output was created and set equal to the result of the ALU which would strictly perform add instructions for this specific design.

The MIPS processor was simulated in order to prove functionality before implementing it on the board. The important waveforms are seen in Figure 6, below.

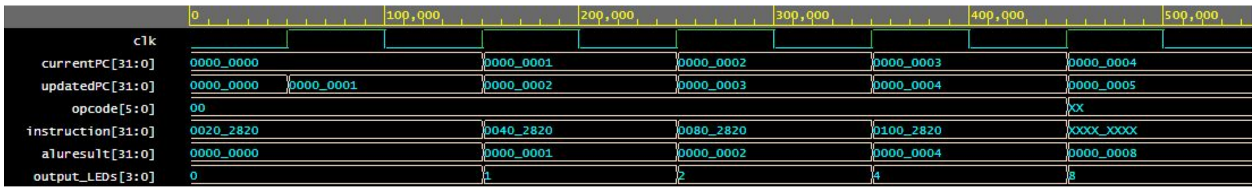
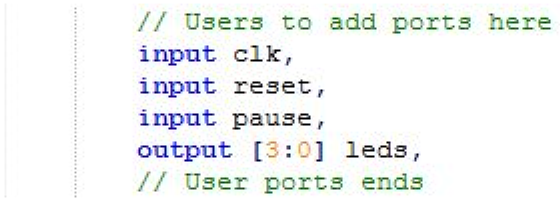


Figure 6: Waveforms for MIPS Processor

After designing the MIPS processor the group was ready to implement it on the PYNQ board through the use of Xilinx Vivado 2018.3 software.

3. Implementation

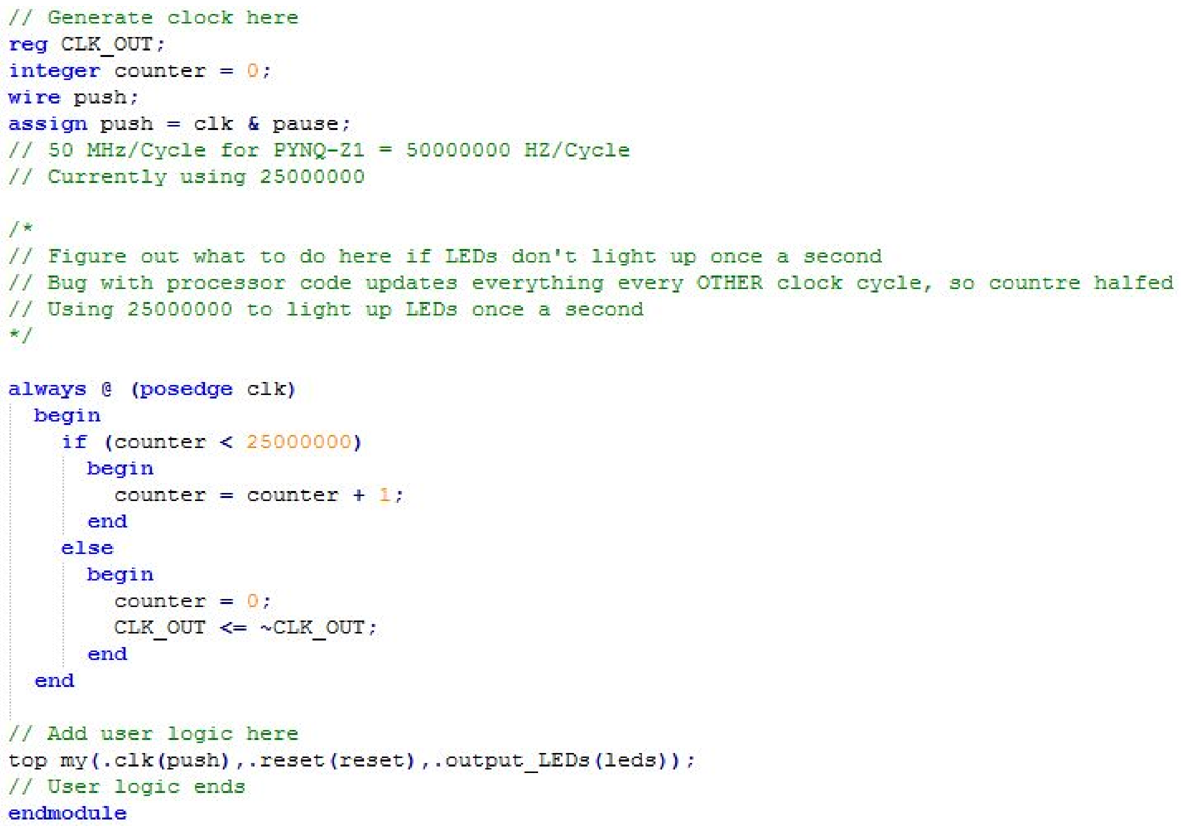
The next step in implementing the MIPS on the PYNQ board was to create IP in HDL. To make this process easier, Vivado has a set of AXI interface templates that can be customized to fit our specifications. We then packaged the IP through the Vivado tool which created the AXI4 IP peripherals and packages' existing source files into an IP package. In our AXI file we added our ports as shown below in Figure 7, on the next page.



```
// Users to add ports here
input clk,
input reset,
input pause,
output [3:0] leds,
// User ports ends
```

Figure 7: AXI User Ports

We added a clock divider to the bottom of our AXI file along with some user logic. The clock was generated using a simple counter. The counters' max boundary was set to 25000000. This is because the on board clock runs at 50 MHz. To ensure that the LEDs lit up quickly, 50000000 (corresponding to 50 MHz) was divided by 2 in order to speed up the sequence while maintaining visible functionality. This is all shown in Figure 8, below.



```
// Generate clock here
reg CLK_OUT;
integer counter = 0;
wire push;
assign push = clk & pause;
// 50 MHz/Cycle for PYNQ-Z1 = 50000000 HZ/Cycle
// Currently using 25000000

/*
// Figure out what to do here if LEDs don't light up once a second
// Bug with processor code updates everything every OTHER clock cycle, so countre halfed
// Using 25000000 to light up LEDs once a second
*/

always @ (posedge clk)
  begin
    if (counter < 25000000)
      begin
        counter = counter + 1;
      end
    else
      begin
        counter = 0;
        CLK_OUT <= ~CLK_OUT;
      end
  end

// Add user logic here
top my(.clk(push),.reset(reset),.output_LEDs(leds));
// User logic ends
endmodule
```

Figure 8: Clock Divider and User Logic for AXI File

The next step taken in implementation was to generate a block diagram. The LED outputs and the reset and pause inputs were set as external and then Vivado's block automation generated the block diagram shown in Figure 9, below.

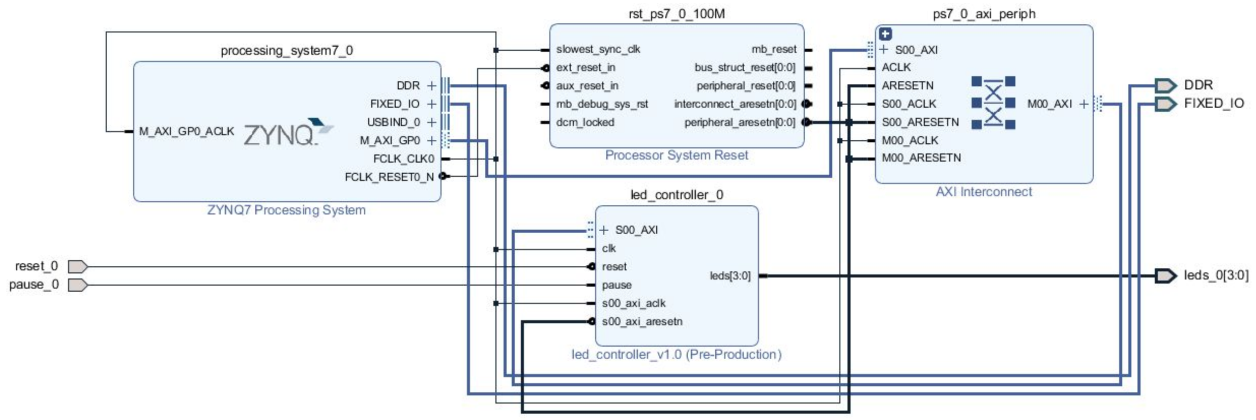
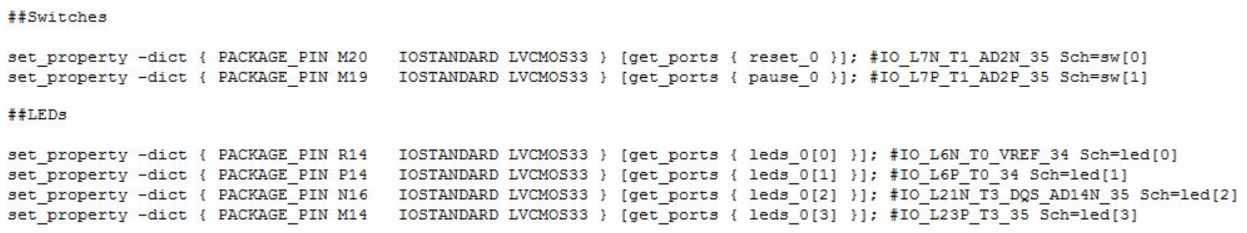


Figure 9: User generated block diagram

Next, an HDL wrapper was created for our file. This created a top-level HDL file for our design. In order to manage the I/O and interface the LEDs and switches for the MIPS code, constraints were specified within an XDC file. These constraints are shown below in figure 10.


```
##Switches

set_property -dict { PACKAGE_PIN M20   IOSTANDARD LVCMOS33 } [get_ports { reset_0 }]; #IO_L7N_T1_AD2N_35 Sch=sw[0]
set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 } [get_ports { pause_0 }]; #IO_L7P_T1_AD2P_35 Sch=sw[1]

##LEDs

set_property -dict { PACKAGE_PIN R14   IOSTANDARD LVCMOS33 } [get_ports { leds_0[0] }]; #IO_L6N_T0_VREF_34 Sch=led[0]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { leds_0[1] }]; #IO_L6P_T0_34 Sch=led[1]
set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { leds_0[2] }]; #IO_L21N_T3_DQS_AD14N_35 Sch=led[2]
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { leds_0[3] }]; #IO_L23P_T3_35 Sch=led[3]
```
Figure 10: I/O Pins from Constraints file

This concludes our design. The next step is to load the code onto the board by first generating a bitstream. This implements our embedded design elements and puts them into a file that gets downloaded to the PYNQ board. The final step taken was to export the design to the

SDK. Before attempting to run our design off a micro SD card, the designs' functionality was confirmed through a wired USB connection. This was as simple as selecting '*Launch on Hardware (GDB)'* within the SDK, which lit up the on-board LEDs in the expected sequence[4].

3.1 Booting off an SD Card

Within the SDK a new Application Project was created, as well as an FSBL. This file is used later in order to compile a bootable image called from the PYNQ board. The next step was to create a ZYNQ Boot Image. We needed to add several Boot Image Partitions to this file as well so we included our FSBL as a bootable file, and our bitstream *.elf* datafiles. The three files were combined and an OUTPUT.bin was generated, allowing for the program to be booted from the micro SD card.

Results

In Figure 11 on the next page, the PYNQ board is shown with the pause switch (SW1) activated. While this switch is activated, the program counter pauses and because of this so does the current LED. Releasing this switch causes the program counter to continue cycling through LEDs. In the top left of the picture you can also notice that our jumper is set to SD card mode, confirming functionality.

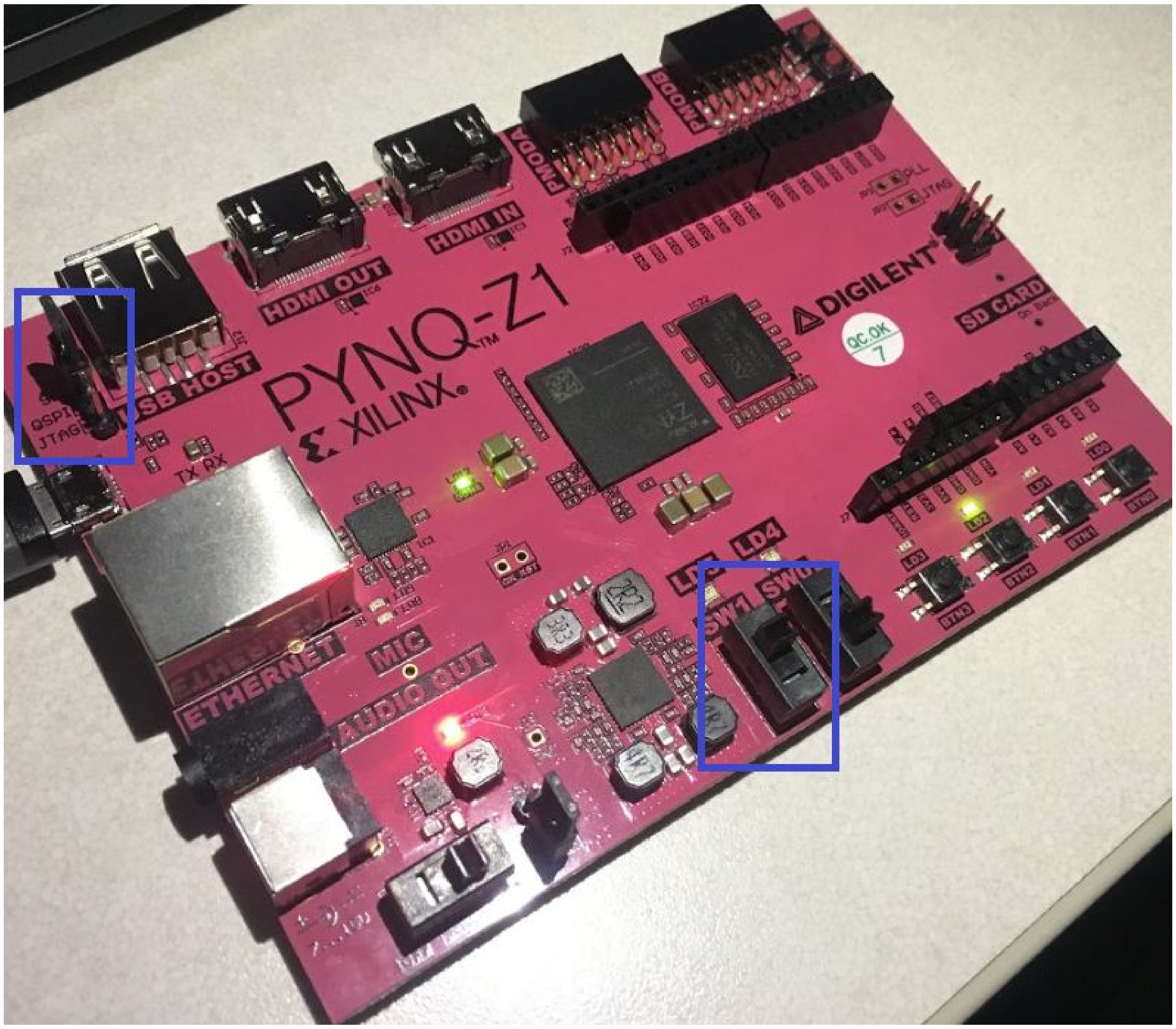


Figure 11: PYNQ Board with Pause Switch Active

On the next page, another feature of the design is depicted in Figure 12. The reset switch (SW0) is activate, meaning the last LED is the only LED on. This is because the reset switch resets the program counter to point to the first instruction.

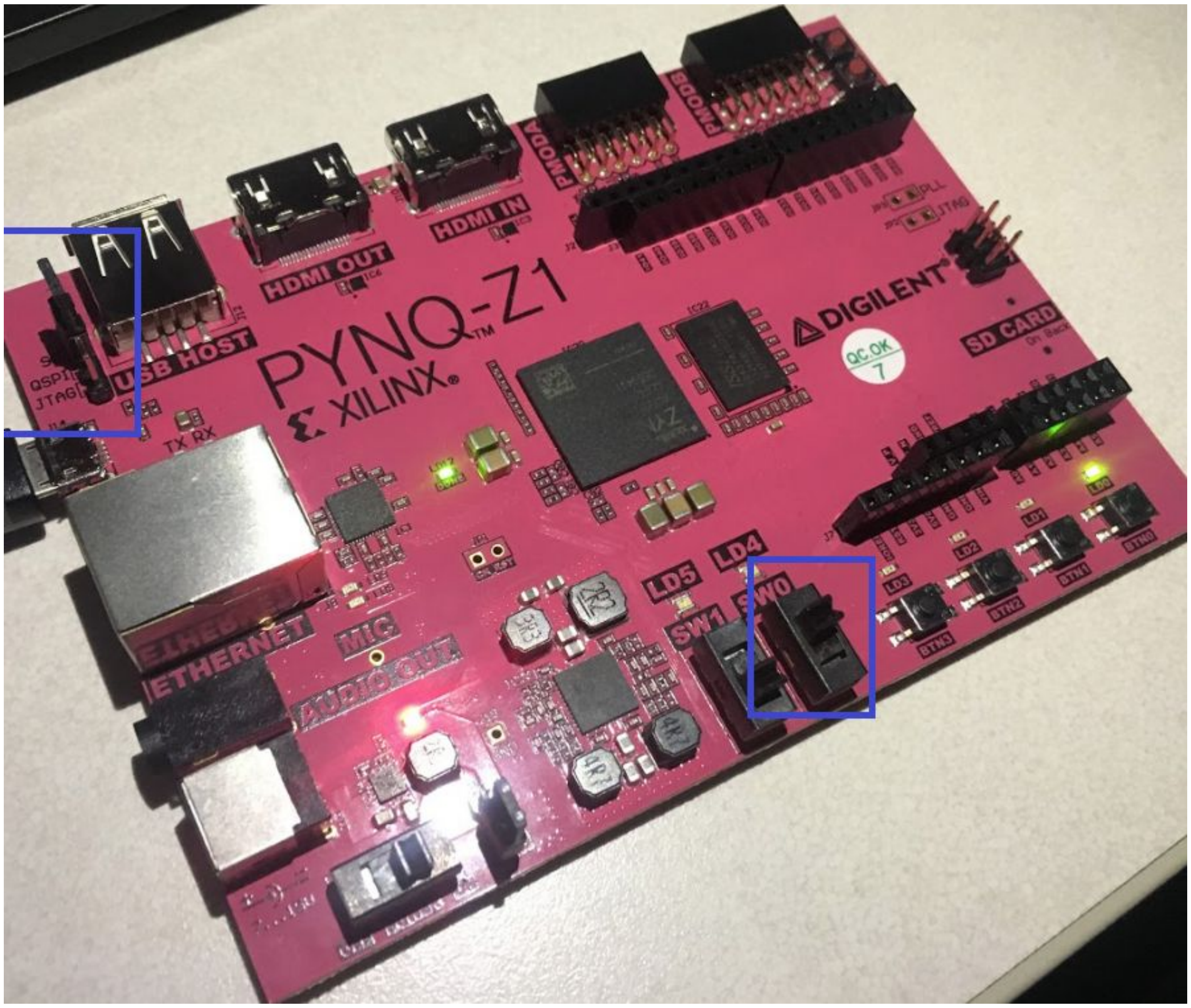


Figure 12: PYNQ Board with Reset Switch Active

Discussion

After finishing the design it's evident that many things about our design process could be improved. Some reflections can be made about our design and many things were learned throughout designing our project.

5.1 Problems Encountered

When testing our MIPS processor, we used EDAPlayground to run simulations through the use of a separate testbench file. After proper functionality, the group proceeded to move the code to Vivado. This lead the group to realize that the testbench included code with instantiations, which should have been included in the main verilog file, as they are necessary for any design with the processor.

After fixing these issues another wave of errors were raised after generating a bitstream. The reason for this was because certain parts of our code couldn't be implemented in hardware. The main case of this was in our use of for loops. The only place for loops can be implemented in hardware is if they are used to create several instantiations of a piece of hardware.

Another case of bad coding which managed to bypass the EDA Playground simulation was triggering always blocks from both positive and negative clock edges.

Another issue that we faced was every time we created an HDL wrapper it would make our code 'read-only.' This happens because when you make the file a wrapper it changes the code into a template. When this occurred, the project needed to be deleted and restarted.

5.2 Lessons Learned

This project improved the groups' proficiency in Verilog coding. The group has also become fairly familiar with using Xilinx Vivado, as this software has not been used by either group member prior to this project. Similarly, the group gained valuable knowledge through using the SDK. In projects prior, the group had simply created verilog code and synthesize it using a testbench. This project required us to interconnect this software to an external hardware FPGA, ensuring that the designed code must be synthesizable in hardware.

5.3 Future Iterations

Instead of setting the values for the LEDs from the ALU, accessing the values from datamemory would provide a cleaner design. We can also improve upon our power, performance and area (PPA) in future iterations as we didn't consider these factors while creating our design.

Appendices

In the three appendices below, the two AXI files as well as the constraints file used in this design can be found. The corresponding MIPS code can be found in the bottom of Appendix A.

Appendix A - AXI4

```
`timescale 1 ns / 1 ps

        module led_controller_v1_0_S00_AXI #
        (
           // Width of S_AXI data bus
           parameter integer C_S_AXI_DATA_WIDTH = 32,

           // Width of S_AXI address bus
           parameter integer C_S_AXI_ADDR_WIDTH = 4
        )

        (
           // Users to add ports here
           input clk,
           input reset,
           input pause,
           output [3:0] leds,

           // Global Clock Signal
           input wire  S_AXI_ACLK,

           // Global Reset Signal. This Signal is Active LOW
           input wire  S_AXI_ARESETN,

           // Write address (issued by master, acceped by Slave)
           input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
           // Write channel Protection type. This signal indicates the
           // privilege and security level of the transaction, and whether
           // the transaction is a data access or an instruction access.
           input wire [2 : 0] S_AXI_AWPROT,

           // Write address valid. This signal indicates that the master signaling
           // valid write address and control information.
           input wire  S_AXI_AWVALID,

           // Write address ready. This signal indicates that the slave is ready
           // to accept an address and associated control signals.
           output wire  S_AXI_AWREADY,

           // Write data (issued by master, acceped by Slave)
           input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,

           // Write strobes. This signal indicates which byte lanes hold
           // valid data. There is one write strobe bit for each eight
           // bits of the write data bus.
           input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,

           // Write valid. This signal indicates that valid write
           // data and strobes are available.
           input wire  S_AXI_WVALID,

           // Write ready. This signal indicates that the slave
           // can accept the write data.
           output wire  S_AXI_WREADY,

           // Write response. This signal indicates the status
           // of the write transaction.
           output wire [1 : 0] S_AXI_BRESP,

           // Write response valid. This signal indicates that the channel
           // is signaling a valid write response.
           output wire  S_AXI_BVALID,

           // Response ready. This signal indicates that the master
           // can accept a write response.
           input wire  S_AXI_BREADY,

           // Read address (issued by master, acceped by Slave)
           input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,

           // Protection type. This signal indicates the privilege
        // and security level of the transaction, and whether the
           // transaction is a data access or an instruction access.
           input wire [2 : 0] S_AXI_ARPROT,

           // Read address valid. This signal indicates that the channel
           // is signaling valid read address and control information.
           input wire  S_AXI_ARVALID,

           // Read address ready. This signal indicates that the slave is
           // ready to accept an address and associated control signals.
           output wire  S_AXI_ARREADY,

           // Read data (issued by slave)
           output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,

           // Read response. This signal indicates the status of the
           // read transfer.
           output wire [1 : 0] S_AXI_RRESP,

           // Read valid. This signal indicates that the channel is
           // signaling the required read data.
           output wire  S_AXI_RVALID,

           // Read ready. This signal indicates that the master can
           // accept the read data and response information.
           input wire  S_AXI_RREADY
);

// AXI4LITE signals
reg [C_S_AXI_ADDR_WIDTH-1 : 0]      axi_awaddr;
reg         axi_awready;
reg         axi_wready;
reg [1 : 0] axi_bresp;
reg         axi_bvalid;
reg [C_S_AXI_ADDR_WIDTH-1 : 0]      axi_araddr;
reg         axi_arready;
reg [C_S_AXI_DATA_WIDTH-1 : 0]      axi_rdata;
reg [1 : 0] axi_rresp;
reg         axi_rvalid;

// Example-specific design signals
// local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
// ADDR_LSB is used for addressing 32/64 bit registers/memories
// ADDR_LSB = 2 for 32 bits (n downto 2)
// ADDR_LSB = 3 for 64 bits (n downto 3)
localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
localparam integer OPT_MEM_ADDR_BITS = 1;
//----------------------------------------------
//-- Signals for user logic register space example
//------------------------------------------------
//-- Number of Slave Registers 4
reg [C_S_AXI_DATA_WIDTH-1:0]        slv_reg0;
reg [C_S_AXI_DATA_WIDTH-1:0]        slv_reg1;
reg [C_S_AXI_DATA_WIDTH-1:0]        slv_reg2;
reg [C_S_AXI_DATA_WIDTH-1:0]        slv_reg3;
wire        slv_reg_rden;
wire        slv_reg_wren;
reg [C_S_AXI_DATA_WIDTH-1:0]         reg_data_out;
integer     byte_index;
reg         aw_en;

// I/O Connections assignments

assign S_AXI_AWREADY     = axi_awready;
assign S_AXI_WREADY      = axi_wready;
assign S_AXI_BRESP       = axi_bresp;
assign S_AXI_BVALID      = axi_bvalid;
assign S_AXI_ARREADY     = axi_arready;
assign S_AXI_RDATA       = axi_rdata;
assign S_AXI_RRESP       = axi_rresp;
assign S_AXI_RVALID      = axi_rvalid;
// Implement axi_awready generation
// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_awready <= 1'b0;
      aw_en <= 1'b1;
    end
  else
    begin
      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
```

```
        begin
          // slave is ready to accept write address when
          // there is a valid write address and write data
          // on the write address and data bus. This design
          // expects no outstanding transactions.
          axi_awready <= 1'b1;
          aw_en <= 1'b0;
        end
        else if (S_AXI_BREADY && axi_bvalid)
            begin
              aw_en <= 1'b1;
              axi_awready <= 1'b0;
            end
      else
        begin
          axi_awready <= 1'b0;
        end
    end
end

// Implement axi_awaddr latching
// This process is used to latch the address when both
// S_AXI_AWVALID and S_AXI_WVALID are valid.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_awaddr <= 0;
    end
  else
    begin
      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
        begin
          // Write Address latching
          axi_awaddr <= S_AXI_AWADDR;
        end
    end
end

// Implement axi_wready generation
// axi_wready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_wready <= 1'b0;
    end
  else
    begin
      if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID && aw_en )
        begin
          // slave is ready to accept write data when
          // there is a valid write address and write data
          // on the write address and data bus. This design
          // expects no outstanding transactions.
          axi_wready <= 1'b1;
        end
      else
        begin
          axi_wready <= 1'b0;
        end
    end
```

```
end

// Implement memory mapped register select and write logic generation
// The write data is accepted and written to memory mapped registers when
// axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are used to
// select byte enables of slave registers while writing.
// These registers are cleared when reset (active low) is applied.
// Slave register write enable is asserted when valid address and data are available
// and the slave is ready to accept the write address and write data.
assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID;

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      slv_reg0 <= 0;
      slv_reg1 <= 0;
      slv_reg2 <= 0;
      slv_reg3 <= 0;
    end
  else begin
    if (slv_reg_wren)
      begin
        case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
          2'h0:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
              if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                // Respective byte enables are asserted as per write strobes
                // Slave register 0
                slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
              end
          2'h1:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
              if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                // Respective byte enables are asserted as per write strobes
                // Slave register 1
                slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
              end
          2'h2:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
              if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                // Respective byte enables are asserted as per write strobes
                // Slave register 2
                slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
              end
          2'h3:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
              if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                // Respective byte enables are asserted as per write strobes
                // Slave register 3
                slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
              end
          default : begin
                      slv_reg0 <= slv_reg0;
                      slv_reg1 <= slv_reg1;
                      slv_reg2 <= slv_reg2;
                      slv_reg3 <= slv_reg3;
                    end
        endcase
      end
  end
end

// Implement write response logic generation
// The write response and response valid signals are asserted by the slave
// when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
// This marks the acceptance of address and indicates the status of
// write transaction.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_bvalid  <= 0;
      axi_bresp   <= 2'b0;
    end
  else
    begin
      if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready && S_AXI_WVALID)
        begin
          // indicates a valid write response is available
          axi_bvalid <= 1'b1;
          axi_bresp  <= 2'b0; // 'OKAY' response
        end                   // work error responses in future
      else
        begin
          if (S_AXI_BREADY && axi_bvalid)
            //check if bready is asserted while bvalid is high)
            //(there is a possibility that bready is always asserted high)
            begin
              axi_bvalid <= 1'b0;
            end
        end
    end
end

// Implement axi_arready generation
// axi_arready is asserted for one S_AXI_ACLK clock cycle when
// S_AXI_ARVALID is asserted. axi_awready is
// de-asserted when reset (active low) is asserted.
// The read address is also latched when S_AXI_ARVALID is
// asserted. axi_araddr is reset to zero on reset assertion.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_arready <= 1'b0;
      axi_araddr  <= 32'b0;
    end
  else
    begin
      if (~axi_arready && S_AXI_ARVALID)
        begin
          // indicates that the slave has acceped the valid read address
          axi_arready <= 1'b1;
          // Read address latching
          axi_araddr  <= S_AXI_ARADDR;
        end
      else
        begin
          axi_arready <= 1'b0;
        end
    end
end

// Implement axi_arvalid generation
// axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_ARVALID and axi_arready are asserted. The slave registers
// data are available on the axi_rdata bus at this instance. The
// assertion of axi_rvalid marks the validity of read data on the
// bus and axi_rresp indicates the status of read transaction.axi_rvalid
```

```
      // is deasserted on reset (active low). axi_rresp and axi_rdata are
      // cleared to zero on reset (active low).
      always @( posedge S_AXI_ACLK )
      begin
        if ( S_AXI_ARESETN == 1'b0 )
          begin
            axi_rvalid <= 0;
            axi_rresp  <= 0;
          end
        else
          begin
            if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
              begin
                // Valid read data is available at the read data bus
                axi_rvalid <= 1'b1;
                axi_rresp  <= 2'b0; // 'OKAY' response
              end
            else if (axi_rvalid && S_AXI_RREADY)
              begin
                // Read data is accepted by the master
                axi_rvalid <= 1'b0;
              end
          end
      end

      // Implement memory mapped register select and read logic generation
      // Slave register read enable is asserted when valid address is available
      // and the slave is ready to accept the read address.
      assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
      always @(*)
      begin
            // Address decoding for reading registers
            case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
              2'h0   : reg_data_out <= slv_reg0;
              2'h1   : reg_data_out <= slv_reg1;
              2'h2   : reg_data_out <= slv_reg2;
              2'h3   : reg_data_out <= slv_reg3;
              default : reg_data_out <= 0;
            endcase
      end

      // Output register or memory read data
      always @( posedge S_AXI_ACLK )
      begin
        if ( S_AXI_ARESETN == 1'b0 )
          begin
            axi_rdata  <= 0;
          end
        else
          begin
            // When there is a valid read address (S_AXI_ARVALID) with
            // acceptance of read address by the slave (axi_arready),
            // output the read dada
            if (slv_reg_rden)
              begin
                axi_rdata <= reg_data_out;     // register read data
              end
          end
      end



// Generate clock here
reg CLK_OUT;
integer counter = 0;
```

```
    wire push;
    assign push = clk & pause;
    // 50 MHz/Cycle for PYNQ-Z1 = 50000000 HZ/Cycle

    always @ (posedge clk)
      begin
        if (counter < 25000000)
          begin
            counter = counter + 1;
          end
        else
          begin
            counter = 0;
            CLK_OUT <= ~CLK_OUT;
          end
      end

    // Add user logic here
    top my(.clk(push),.reset(reset),.output_LEDs(leds));
    // User logic ends
endmodule

// Time values read as 1ns and will be rounded to nearest 10ps
`timescale 1 ns / 10 ps

module top(clk, reset, output_LEDs);

    // Inputs
                input clk;
                input reset;
                output [3:0] output_LEDs;

                // Monitor Outputs
                wire[31:0] branchresult;
                wire[31:0] instruction;
                wire[31:0] input_A;
                wire[31:0] input_B;
                wire writeenable;
                wire bf;
                wire zf;
                wire[15:0] IMM;
                wire[15:0] signbitreg;
                wire[15:0] unextended;
                wire[31:0] PC_plus4;
                wire[31:0] address;
                wire[31:0] aluresult;
                wire[31:0] currentPC;
                wire[31:0] datamem;
                wire[31:0] extended;
                wire[31:0] readdata;
                wire[31:0] readdataA;
                wire[31:0] readdataB;
                wire[31:0] registermemory;
                wire[31:0] updatedPC;
                wire[31:0] writedata;
                wire[4:0] rd;
                wire[4:0] readregisterA;
                wire[4:0] readregisterB;
                wire[4:0] rs;
                wire[4:0] rt;
                wire[4:0] shift;
                wire[4:0] writeregister;
                wire[5:0] func;
                wire[5:0] opcode;
                wire[6:0] instructionreg;
```

```
programcounter pc_inst(
                            .clk            (clk),
                            .currentPC      (currentPC),
                            .updatedPC      (updatedPC)
                            );

pcadder pcadder_inst(
                            .clk            (clk),
                            .currentPC      (currentPC),
                            .PC_plus4       (PC_plus4)
                            );

instructionmemory im_inst(
                            .clk            (clk),
                            .currentPC      (currentPC),
                            .instruction    (instruction)
                            );

registerfile rf_inst(
                            .clk            (clk),
                            .reset          (reset),
                            .readregisterA  (readregisterA),
                            .readregisterB  (readregisterB),
                            .writeregister  (writeregister),
                            .writedata      (writedata),
                            .readdataA      (readdataA),
                            .readdataB      (readdataB),
                            .writeenable    (writeenable)
                            );

signextend se_inst(
                            .instruction    (instruction),
                            .extended       (extended)
                            );

PC_MUX pc_mux_inst(
                            .clk            (clk),
                            .input_A        (input_A),
                            .input_B        (input_B),
                            .PC_plus4       (PC_plus4),
                            .branchresult   (branchresult),
                            .updatedPC      (updatedPC),
                            .bf             (bf),
                            .zf             (zf)
                            );

branch_alu b_alu_inst(
                            .clk            (clk),
                            .PC_plus4       (PC_plus4),
                            .extended       (extended),
                            .branchresult   (branchresult)
                            );

datamemory dm_inst(
                            .clk            (clk),
                            .reset          (reset),
                            .writeenable    (writeenable),
                            .address        (address),
                            .aluresult      (aluresult),
                            .readdata       (readdata)
                            );

WB_MUX wb_mux_inst(
                            .instruction    (instruction),
                            .opcode         (opcode),
                            .aluresult      (aluresult),
                                        .readdata               (readdata),
                                        .writedata              (writedata)
                                        );

   instructionformat if_inst(
                                        .clk                     (clk),
                                        .instruction            (instruction),
                                        .opcode                 (opcode),
                                        .rs                      (rs),
                                        .rt                      (rt),
                                        .rd                      (rd),
                                        .shift                   (shift),
                                        .func                    (func),
                                        .IMM                     (IMM)
                                        );

   ALU alu_inst (
                                        .clk                    (clk),
                                        .input_A                (input_A),
                                        .input_B                (input_B),
                                        .rs                     (rs),
                                        .rt                     (rt),
                                        .opcode                 (opcode),
                                        .func                   (func),
                                        .extended               (extended),
                                        .PC_plus4               (PC_plus4),
                                        .readdataA              (readdataA),
                                        .readdataB              (readdataB),
                                        .aluresult              (aluresult),
                                        .writeenable            (writeenable),
                                        .output_LEDs            (output_LEDs),
                                        .bf                     (bf)
                                        );
endmodule



// Program counter is a register containing the location of the instruction being executed
module programcounter(clk, currentPC, updatedPC);

  input clk;
  //input reset;
  input wire [31:0] updatedPC;
  output reg[31:0] currentPC;

  initial begin
                currentPC = 32'h00000000;
  end

  // Set boundries for program counter
  always@(posedge clk)
  begin
    // If program reaches 16 (decimal), reset program counter.
    // This ensures LEDs loop
    if (updatedPC == 32'b010000)
               currentPC = 0;
    else
        // If boundry is not hit, proceed
        currentPC = updatedPC;
  end
endmodule
```

```
// Adder function used to increment by 4 bytes (32 bits)
module pcadder(clk, currentPC, PC_plus4);

  input clk;
  input[31:0] currentPC;
  output reg[31:0] PC_plus4;

  initial begin
		PC_plus4 = 32'h00000000;
  end

  always@(posedge clk)
  begin
    // Increment program counter
    // Typical MIPS architecture increments by 4 bytes
    PC_plus4 = currentPC+1;
  end
endmodule


// Instruction memory holds information which defines instructions using MIPS instruction format
module instructionmemory(clk, instruction, currentPC);

  input clk;
  input wire[31:0] currentPC;

  // 32bit instruction
  output wire[31:0] instruction;

  // Register containing 7 instructions, all of size 32
  reg [31:0] instructionreg [6:0];

  // Hardcode instructions using standard MIPS instruction format
  initial begin

    /*
		// R instructions (op, rs, rt, rd, shift, func. rs=r16, rt=r18, rd=r17)
    instructionreg[0] = 32'b00000010000100101000100000100000; //add
    instructionreg[1] = 32'b00000010000100101000100000100010; //sub
    instructionreg[2] = 32'b00000010000100101000100000100100; //and
    instructionreg[3] = 32'b00000010000100101000100000100101; //or

		// I instructions (op, rs, rt, IMM. rs=r17, rt=r16)
    instructionreg[4] = 32'b00010010001100000000000000000000; //beq
    instructionreg[5] = 32'b10001110001100000000000000000000; //lw
    instructionreg[6] = 32'b10101110001100000000000000000000; //sw
    */

    //https://www.eg.bucknell.edu/~csci320/mips_web/
    instructionreg[0] = 32'b00000000000100000001010000100000; //LED1 = 0+1
    instructionreg[1] = 32'b00000000001000000001010000100000; //LED2 = 0+2
    instructionreg[2] = 32'b00000000010000000001010000100000; //LED3 = 0+4
    instructionreg[3] = 32'b00000000100000000001010000100000; //LED4 = 0+8
  end

  //
  assign instruction = instructionreg[currentPC];

endmodule

// Definition of Instructions
// Separate parts of R and I instructions for ALU processes
// rs/rt/rd/shift/IMM written for testbench purposes
module instructionformat(clk, instruction, opcode, rs, rt, rd, shift, func, IMM);

  input clk;
  input wire [31:0] instruction;

  // Fields for instruction format
  output [5:0] opcode;
  output [4:0] rs;
  output [4:0] rt;
  output [4:0] rd;
  output [4:0] shift;
  output [5:0] func;
  output [15:0] IMM;

  //reg[5:0] opcode;
  //reg[4:0] rs;
  //reg[4:0] rt;
  reg[4:0] rd;
  reg[4:0] shift;
  reg[5:0] func;
  reg[15:0] IMM;

  // opcode, rs and rt all are same bits regardless of R or I instruction
  assign opcode = instruction[31:26];
  assign rs = instruction[25:21];
  assign rt = instruction[20:16];

  always@(posedge clk)
  begin
    // Check if opcode=000000 (R instruction)
    if(instruction[31:26]==6'b000000)
    begin
      // Format for R instructions
      rd = instruction[15:11];
      shift = instruction[10:6];
      func = instruction[5:0];
    end

    // If opcode!=000000, must be an I instruction
    else
                begin
        // IMM unique to I instructions
        IMM = instruction[15:0];
                end
    end
endmodule

// Set of general and special purpose storage cells inside CPU
module registerfile(clk, reset, readregisterA, readregisterB, writeregister, writedata, readdataA, readdataB, writeenable);

  input clk;
  input reset;

  input writeenable;

  // Address of first and second registers to be read
  input[4:0]  readregisterA;
  input[4:0]  readregisterB;

  // Write register
  input[4:0]  writeregister;

  // Data to write
  input[31:0] writedata;
```

```
// ALU inputs
output[31:0] readdataA;
output[31:0] readdataB;

// Create 32 registers 32 bits in size
reg [31:0] registermemory [31:0];

// Initialize all register memory to 0
initial begin
  registermemory[0] = 0;
  registermemory[1] = 0;
  registermemory[2] = 0;
  registermemory[3] = 0;
  registermemory[4] = 0;
  registermemory[5] = 0;
  registermemory[6] = 0;
  registermemory[7] = 0;
  registermemory[8] = 0;
  registermemory[9] = 0;
  registermemory[10] = 0;
  registermemory[11] = 0;
  registermemory[12] = 0;
  registermemory[13] = 0;
  registermemory[14] = 0;
  registermemory[15] = 0;
  registermemory[16] = 0;
  registermemory[17] = 0;
  registermemory[18] = 0;
  registermemory[19] = 0;
  registermemory[20] = 0;
  registermemory[21] = 0;
  registermemory[22] = 0;
  registermemory[23] = 0;
  registermemory[24] = 0;
  registermemory[25] = 0;
  registermemory[26] = 0;
  registermemory[27] = 0;
  registermemory[28] = 0;
  registermemory[29] = 0;
  registermemory[30] = 0;
  registermemory[31] = 0;
end

// Check for reset and write enable on positive clock edge
always@(posedge clk)

// If reset is active fill all registers with 0
begin if(reset==1)
  begin
          registermemory[0] = 0;
      registermemory[1] = 0;
      registermemory[2] = 0;
      registermemory[3] = 0;
      registermemory[4] = 0;
      registermemory[5] = 0;
      registermemory[6] = 0;
      registermemory[7] = 0;
      registermemory[8] = 0;
      registermemory[9] = 0;
      registermemory[10] = 0;
      registermemory[11] = 0;
      registermemory[12] = 0;
      registermemory[13] = 0;
      registermemory[14] = 0;
      registermemory[15] = 0;
      registermemory[16] = 0;
```

```
        registermemory[17] = 0;
        registermemory[18] = 0;
        registermemory[19] = 0;
        registermemory[20] = 0;
        registermemory[21] = 0;
        registermemory[22] = 0;
        registermemory[23] = 0;
        registermemory[24] = 0;
        registermemory[25] = 0;
        registermemory[26] = 0;
        registermemory[27] = 0;
        registermemory[28] = 0;
        registermemory[29] = 0;
        registermemory[30] = 0;
        registermemory[31] = 0;
            end

    // If write enable is active, write data
            if (writeenable==1)
        begin
            // Overwrite
                            registermemory[writeregister] = writedata;
        end
            end
endmodule

// Used for BEQ/LW/SQ in ALU
// Increase length of data while keeping sign (positive / negative)
module signextend(instruction, extended);

  input[31:0] instruction;
  output wire[31:0] extended;

  // 16 MSB's are filled with sign bit
  wire [15:0] signbitreg;

  // Fill all 16bits of signbitreg with sign bit of unextended number
  assign signbitreg = instruction[15];

            // Signbit for 16 MSB's. Original 16-bit number remains in bits of lowest significance.
            // Concatenate
            assign extended = {signbitreg, instruction[15:0]};
endmodule


// ALU performs actual functions (add/sub/and/or/beq/sw/lw)
module ALU(clk, input_A,  input_B, opcode, func, extended, PC_plus4, readdataA, readdataB, aluresult, writeenable, bf,
output_LEDs, rs, rt); //c_flag);

  input clk;

  // Inputs (numbers) to ALU
  input [31:0] input_A;
  input [31:0] input_B;

  // For arithmetic
  input wire [4:0] rs;
  input wire [4:0] rt;

  // Instruction specifiers
  input[5:0] opcode;
  input[5:0] func;

  // Factors to aid in I type instructions
  input[31:0] extended;
  input[31:0] PC_plus4;
```

```
input[31:0] readdataA;
input[31:0] readdataB;

// ALU outputs a result
// Result written to memory
// Result also controls LED signals on FPGA
output reg writeenable;
output reg bf;
output reg [31:0] aluresult;
output reg[3:0] output_LEDs;

initial begin
   writeenable = 0;
   bf = 0;
   aluresult = 0;
   output_LEDs = 0;
end

always@(posedge clk)
  begin
                // R instructions (add, sub, and, or)
    if(opcode == 6'b000000)
      begin
                                // ADD
        if(func == 6'b100000)
                                  begin
                // Result = A + B
            bf = 0;
                                    aluresult = rs+rt;
                                  end

                                // SUB
        if(func == 6'b100010)
                                  begin
            // Result = A - B
            bf = 0;
                                    aluresult = rs-rt;
                                  end

                                // AND
        if(func == 6'b100100)
                                  begin
            // Result = A and B
            bf = 0;
                                                aluresult = rs & rt;
                                  end

                                // OR
        if(func == 6'b100101)
                                  begin
            // Result = A or B
            bf = 0;
                                                aluresult=rs | rt;
                                  end
      end

    // Else instruction is not an R instruction
    // Check opcode again
                // BEQ
    else if(opcode == 6'b000100)
                              begin
        // Result (BEQ) = signextend + PCplus4
        bf = 1;
                    aluresult=extended+PC_plus4;
                              end
```

```
	    // LW
      else if(opcode==6'b100010)
			begin
        bf = 0;
		  aluresult=extended+readdataA;
			end

	    // SW
      else if(opcode==6'b101010)
			begin
        bf = 0;
		  aluresult=extended+readdataB;
       end

      // Set output_LEDs according to output of ALU
      output_LEDs = aluresult[3:0];
    end

  // When result changes, enable writing ability
  always@(aluresult)
    begin
	    writeenable=1;
    end
endmodule

module branch_alu(clk, PC_plus4, extended, branchresult);

  input clk;
  input[31:0] PC_plus4;
  input[31:0] extended;

  output reg[31:0] branchresult;

  // Add PC+4 and extended
  always@(posedge clk)
  begin
    branchresult = PC_plus4 + extended;
  end
endmodule

// MUX to feed next program counter value back to program counter
module PC_MUX(clk, input_A, input_B, PC_plus4, branchresult, updatedPC, bf, zf);

  input clk;

  // Inputs are fed from programcounter adder and ALU outputs
  input[31:0] PC_plus4;
  input[31:0] branchresult;

  input [31:0] input_A;
  input [31:0] input_B;
  output reg[31:0] updatedPC;

  input bf;
  output reg zf;

  // Start wth zero flag set to 0
  initial begin
	   zf = 0;
  end

  /*
  // This code is necssary for functionality on EDA Playground
  // Bug with clk only updates data every other clk cycle, so update twice as fast)
  // always@(posedge clk or negedge clk)
  */
  // Udate PC_MUX select
  always@(posedge clk)
    begin
      if (input_A - input_B == 0)
        zf = 1;
          if (zf && bf == 1)
            begin
                          updatedPC = branchresult;
            end
          else
            begin
              updatedPC = PC_plus4;
            end
    end
endmodule

// Temporarily store results
module datamemory(clk, reset, writeenable, address, aluresult, readdata);

  input clk;
  input reset;
  input writeenable;
  input[31:0] address;
  input[31:0] aluresult;

  // Can read data from datamemory (unused in program)
  output[31:0] readdata;

  // 32 bit memory with 64 entries
  reg[31:0] datamem[0:63];

  // Fill data with all 0's initially
  initial begin
                datamem[0] = 0;
                datamem[1] = 0;
                datamem[2] = 0;
                datamem[3] = 0;
                datamem[4] = 0;
                datamem[5] = 0;
                datamem[6] = 0;
                datamem[7] = 0;
                datamem[8] = 0;
                datamem[9] = 0;
                datamem[10] = 0;
                datamem[11] = 0;
                datamem[12] = 0;
                datamem[13] = 0;
                datamem[14] = 0;
                datamem[15] = 0;
                datamem[16] = 0;
                datamem[17] = 0;
                datamem[18] = 0;
                datamem[19] = 0;
                datamem[20] = 0;
                datamem[21] = 0;
                datamem[22] = 0;
                datamem[23] = 0;
                datamem[24] = 0;
                datamem[25] = 0;
                datamem[26] = 0;
                datamem[27] = 0;
                datamem[28] = 0;
                datamem[29] = 0;
                datamem[30] = 0;
                datamem[31] = 0;
  end

  // Check for reset and write enable on positive clock edge
  always@(posedge clk)

    // If reset is active fill all registers with 0
    begin if(reset==1)

      // Set memory in all data memory registers to 0
        begin
                datamem[0] = 0;
            datamem[1] = 0;
            datamem[2] = 0;
            datamem[3] = 0;
            datamem[4] = 0;
            datamem[5] = 0;
            datamem[6] = 0;
            datamem[7] = 0;
            datamem[8] = 0;
            datamem[9] = 0;
            datamem[10] = 0;
            datamem[11] = 0;
            datamem[12] = 0;
            datamem[13] = 0;
            datamem[14] = 0;
            datamem[15] = 0;
            datamem[16] = 0;
            datamem[17] = 0;
            datamem[18] = 0;
            datamem[19] = 0;
            datamem[20] = 0;
            datamem[21] = 0;
            datamem[22] = 0;
            datamem[23] = 0;
            datamem[24] = 0;
            datamem[25] = 0;
            datamem[26] = 0;
            datamem[27] = 0;
            datamem[28] = 0;
            datamem[29] = 0;
            datamem[30] = 0;
            datamem[31] = 0;
          end

      // If write enable is active, write data
      if (writeenable==1)
        begin
          // Overwrite (non blocking --> parallel change)
          datamem[address] = aluresult;
        end
              end
endmodule


// MUX feeds back to register file
module WB_MUX(instruction, opcode, aluresult, readdata, writedata);

  input [31:0] instruction;
  input [5:0]  opcode;

  // Inputs are fed from result of ALU and readdata (data memory)
  input[31:0] aluresult;
  input[31:0] readdata;

  // Output of MUX feeds back to register file
  output reg[31:0] writedata;
  // If instruction changes, check instruction type (possibly update WB_MUX select bit)
  // Read data is sent to registerfile when select is 1.  (I-type, opcode != 0)
  // ALU result is sent to register file when select is 0. (R-type, opcode = 0)
  always@(instruction)
          begin
    // If R-type instruction processed, send ALU result to register file
           if (opcode == 0)
                  writedata = aluresult;

    // Otherwise if an I-type instruction is processed, send read data to register file
           else
                  writedata = readdata;
          end
endmodule
```

Appendix B - AXI

```
`timescale 1 ns / 1 ps

        module led_controller_v1_0 #
        (
                // Parameters of Axi Slave Bus Interface S00_AXI
                parameter integer C_S00_AXI_DATA_WIDTH      = 32,
                parameter integer C_S00_AXI_ADDR_WIDTH      = 4
        )

        (
                // Users to add ports here
    input clk,
    input reset,
                input pause,
    output [3:0] leds,
                // User ports ends

                // Ports of Axi Slave Bus Interface S00_AXI
                input wire  s00_axi_aclk,
                input wire  s00_axi_aresetn,
                input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_awaddr,
                input wire [2 : 0] s00_axi_awprot,
                input wire  s00_axi_awvalid,
                output wire  s00_axi_awready,
                input wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_wdata,
                input wire [(C_S00_AXI_DATA_WIDTH/8)-1 : 0] s00_axi_wstrb,
                input wire  s00_axi_wvalid,
                output wire  s00_axi_wready,
                output wire [1 : 0] s00_axi_bresp,
                output wire  s00_axi_bvalid,
                input wire  s00_axi_bready,
                input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_araddr,
                input wire [2 : 0] s00_axi_arprot,
                input wire  s00_axi_arvalid,
                output wire  s00_axi_arready,
                output wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_rdata,
                output wire [1 : 0] s00_axi_rresp,
                output wire  s00_axi_rvalid,
                input wire  s00_axi_rready
        );

  // Instantiation of Axi Bus Interface S00_AXI
        led_controller_v1_0_S00_AXI # (
                .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
                .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
        )
```

```
led_controller_v1_0_S00_AXI_inst (
    .clk(clk),.reset(reset),.pause(pause),.leds(leds),
            .S_AXI_ACLK(s00_axi_aclk),
            .S_AXI_ARESETN(s00_axi_aresetn),
            .S_AXI_AWADDR(s00_axi_awaddr),
            .S_AXI_AWPROT(s00_axi_awprot),
            .S_AXI_AWVALID(s00_axi_awvalid),
            .S_AXI_AWREADY(s00_axi_awready),
            .S_AXI_WDATA(s00_axi_wdata),
            .S_AXI_WSTRB(s00_axi_wstrb),
            .S_AXI_WVALID(s00_axi_wvalid),
            .S_AXI_WREADY(s00_axi_wready),
            .S_AXI_BRESP(s00_axi_bresp),
            .S_AXI_BVALID(s00_axi_bvalid),
            .S_AXI_BREADY(s00_axi_bready),
            .S_AXI_ARADDR(s00_axi_araddr),
            .S_AXI_ARPROT(s00_axi_arprot),
            .S_AXI_ARVALID(s00_axi_arvalid),
            .S_AXI_ARREADY(s00_axi_arready),
            .S_AXI_RDATA(s00_axi_rdata),
            .S_AXI_RRESP(s00_axi_rresp),
            .S_AXI_RVALID(s00_axi_rvalid),
            .S_AXI_RREADY(s00_axi_rready)
);

endmodule
```

Appendix C - Constraints File

```
## This file is a general .xdc for the PYNQ-Z1 board Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

## Clock signal 125 MHz

#set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports { sysclk }]; #IO_L13P_T2_MRCC_35 Sch=sysclk
#create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { sysclk }];

##Switches

set_property -dict { PACKAGE_PIN M20   IOSTANDARD LVCMOS33 } [get_ports { reset_0 }]; #IO_L7N_T1_AD2N_35 Sch=sw[0]
set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 } [get_ports { pause_0 }]; #IO_L7P_T1_AD2P_35 Sch=sw[1]

##RGB LEDs

#set_property -dict { PACKAGE_PIN L15   IOSTANDARD LVCMOS33 } [get_ports { led4_b }]; #IO_L22N_T3_AD7N_35
Sch=led4_b
#set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports { led4_g }]; #IO_L16P_T2_35 Sch=led4_g
#set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { led4_r }]; #IO_L21P_T3_DQS_AD14P_35
Sch=led4_r
#set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { led5_b }]; #IO_0_35 Sch=led5_b
#set_property -dict { PACKAGE_PIN L14   IOSTANDARD LVCMOS33 } [get_ports { led5_g }]; #IO_L22P_T3_AD7P_35 Sch=led5_g
#set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [get_ports { led5_r }]; #IO_L23N_T3_35 Sch=led5_r

##LEDs

set_property -dict { PACKAGE_PIN R14   IOSTANDARD LVCMOS33 } [get_ports { leds_0[0] }]; #IO_L6N_T0_VREF_34 Sch=led[0]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { leds_0[1] }]; #IO_L6P_T0_34 Sch=led[1]
set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { leds_0[2] }]; #IO_L21N_T3_DQS_AD14N_35
Sch=led[2]
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { leds_0[3] }]; #IO_L23P_T3_35 Sch=led[3]

##Buttons
```

```
#set_property -dict { PACKAGE_PIN D19   IOSTANDARD LVCMOS33 } [get_ports { btn[0] }]; #IO_L4P_T0_35 Sch=btn[0]
#set_property -dict { PACKAGE_PIN D20   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }]; #IO_L4N_T0_35 Sch=btn[1]
#set_property -dict { PACKAGE_PIN L20   IOSTANDARD LVCMOS33 } [get_ports { btn[2] }]; #IO_L9N_T1_DQS_AD3N_35
Sch=btn[2]
#set_property -dict { PACKAGE_PIN L19   IOSTANDARD LVCMOS33 } [get_ports { btn[3] }]; #IO_L9P_T1_DQS_AD3P_35
Sch=btn[3]

##Pmod Header JA

#set_property -dict { PACKAGE_PIN Y18   IOSTANDARD LVCMOS33 } [get_ports { ja[0] }]; #IO_L17P_T2_34 Sch=ja_p[1]
#set_property -dict { PACKAGE_PIN Y19   IOSTANDARD LVCMOS33 } [get_ports { ja[1] }]; #IO_L17N_T2_34 Sch=ja_n[1]
#set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { ja[2] }]; #IO_L7P_T1_34 Sch=ja_p[2]
#set_property -dict { PACKAGE_PIN Y17   IOSTANDARD LVCMOS33 } [get_ports { ja[3] }]; #IO_L7N_T1_34 Sch=ja_n[2]
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { ja[4] }]; #IO_L12P_T1_MRCC_34 Sch=ja_p[3]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports { ja[5] }]; #IO_L12N_T1_MRCC_34 Sch=ja_n[3]
#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports { ja[6] }]; #IO_L22P_T3_34 Sch=ja_p[4]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports { ja[7] }]; #IO_L22N_T3_34 Sch=ja_n[4]

##Pmod Header JB

#set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports { jb[0] }]; #IO_L8P_T1_34 Sch=jb_p[1]
#set_property -dict { PACKAGE_PIN Y14   IOSTANDARD LVCMOS33 } [get_ports { jb[1] }]; #IO_L8N_T1_34 Sch=jb_n[1]
#set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { jb[2] }]; #IO_L1P_T0_34 Sch=jb_p[2]
#set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { jb[3] }]; #IO_L1N_T0_34 Sch=jb_n[2]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { jb[4] }]; #IO_L18P_T2_34 Sch=jb_p[3]
#set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports { jb[5] }]; #IO_L18N_T2_34 Sch=jb_n[3]
#set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { jb[6] }]; #IO_L4P_T0_34 Sch=jb_p[4]
#set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports { jb[7] }]; #IO_L4N_T0_34 Sch=jb_n[4]

##Audio Out

#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { aud_pwm }]; #IO_L20N_T3_34 Sch=aud_pwm
#set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports { aud_sd }]; #IO_L20P_T3_34 Sch=aud_sd

##Mic input

#set_property -dict { PACKAGE_PIN F17   IOSTANDARD LVCMOS33 } [get_ports { m_clk }]; #IO_L6N_T0_VREF_35 Sch=m_clk
#set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports { m_data }]; #IO_L16N_T2_35 Sch=m_data

##ChipKit Single Ended Analog Inputs
##NOTE: The ck_an_p pins can be used as single ended analog inputs with voltages from 0-3.3V (Chipkit Analog pins A0-A5).
##      These signals should only be connected to the XADC core. When using these pins as digital I/O, use pins ck_io[14-19].

#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { ck_an_n[0] }]; #IO_L3N_T0_DQS_AD1N_35
Sch=ck_an_n[0]
#set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports { ck_an_p[0] }]; #IO_L3P_T0_DQS_AD1P_35
Sch=ck_an_p[0]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports { ck_an_n[1] }]; #IO_L5N_T0_AD9N_35
Sch=ck_an_n[1]
#set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports { ck_an_p[1] }]; #IO_L5P_T0_AD9P_35
Sch=ck_an_p[1]
#set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { ck_an_n[2] }]; #IO_L20N_T3_AD6N_35
Sch=ck_an_n[2]
#set_property -dict { PACKAGE_PIN K14   IOSTANDARD LVCMOS33 } [get_ports { ck_an_p[2] }]; #IO_L20P_T3_AD6P_35
Sch=ck_an_p[2]
#set_property -dict { PACKAGE_PIN J16   IOSTANDARD LVCMOS33 } [get_ports { ck_an_n[3] }]; #IO_L24N_T3_AD15N_35
Sch=ck_an_n[3]
#set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { ck_an_p[3] }]; #IO_L24P_T3_AD15P_35
Sch=ck_an_p[3]
#set_property -dict { PACKAGE_PIN H20   IOSTANDARD LVCMOS33 } [get_ports { ck_an_n[4] }]; #IO_L17N_T2_AD5N_35
Sch=ck_an_n[4]
#set_property -dict { PACKAGE_PIN J20   IOSTANDARD LVCMOS33 } [get_ports { ck_an_p[4] }]; #IO_L17P_T2_AD5P_35
Sch=ck_an_p[4]
#set_property -dict { PACKAGE_PIN G20   IOSTANDARD LVCMOS33 } [get_ports { ck_an_n[5] }]; #IO_L18N_T2_AD13N_35
Sch=ck_an_n[5]
```

```
#set_property -dict { PACKAGE_PIN G19   IOSTANDARD LVCMOS33 } [get_ports { ck_an_p[5] }]; #IO_L18P_T2_AD13P_35
Sch=ck_an_p[5]

##ChipKit Digital I/O Low

#set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { ck_io[0] }]; #IO_L5P_T0_34 Sch=ck_io[0]
#set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { ck_io[1] }]; #IO_L2N_T0_34 Sch=ck_io[1]
#set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { ck_io[2] }]; #IO_L3P_T0_DQS_PUDC_B_34
Sch=ck_io[2]
#set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports { ck_io[3] }]; #IO_L3N_T0_DQS_34 Sch=ck_io[3]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { ck_io[4] }]; #IO_L10P_T1_34 Sch=ck_io[4]
#set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { ck_io[5] }]; #IO_L5N_T0_34 Sch=ck_io[5]
#set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { ck_io[6] }]; #IO_L19P_T3_34 Sch=ck_io[6]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { ck_io[7] }]; #IO_L9N_T1_DQS_34
Sch=ck_io[7]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { ck_io[8] }]; #IO_L21P_T3_DQS_34
Sch=ck_io[8]
#set_property -dict { PACKAGE_PIN V18   IOSTANDARD LVCMOS33 } [get_ports { ck_io[9] }]; #IO_L21N_T3_DQS_34
Sch=ck_io[9]
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { ck_io[10] }]; #IO_L9P_T1_DQS_34
Sch=ck_io[10]
#set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { ck_io[11] }]; #IO_L19N_T3_VREF_34
Sch=ck_io[11]
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { ck_io[12] }]; #IO_L23N_T3_34 Sch=ck_io[12]
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { ck_io[13] }]; #IO_L23P_T3_34 Sch=ck_io[13]

##ChipKit Digital I/O On Outer Analog Header
##NOTE: These pins should be used when using the analog header signals A0-A5 as digital I/O (Chipkit digital pins 14-19)

#set_property -dict { PACKAGE_PIN Y11   IOSTANDARD LVCMOS33 } [get_ports { ck_io[14] }]; #IO_L18N_T2_13 Sch=ck_a[0]
#set_property -dict { PACKAGE_PIN Y12   IOSTANDARD LVCMOS33 } [get_ports { ck_io[15] }]; #IO_L20P_T3_13 Sch=ck_a[1]
#set_property -dict { PACKAGE_PIN W11   IOSTANDARD LVCMOS33 } [get_ports { ck_io[16] }]; #IO_L18P_T2_13 Sch=ck_a[2]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { ck_io[17] }]; #IO_L21P_T3_DQS_13
Sch=ck_a[3]
#set_property -dict { PACKAGE_PIN T5    IOSTANDARD LVCMOS33 } [get_ports { ck_io[18] }]; #IO_L19P_T3_13 Sch=ck_a[4]
#set_property -dict { PACKAGE_PIN U10   IOSTANDARD LVCMOS33 } [get_ports { ck_io[19] }]; #IO_L12N_T1_MRCC_13
Sch=ck_a[5]

##ChipKit Digital I/O On Inner Analog Header
##NOTE: These pins will need to be connected to the XADC core when used as differential analog inputs (Chipkit analog pins
A6-A11)

#set_property -dict { PACKAGE_PIN B20   IOSTANDARD LVCMOS33 } [get_ports { ck_io[20] }]; #IO_L1N_T0_AD0N_35
Sch=ad_n[0]
#set_property -dict { PACKAGE_PIN C20   IOSTANDARD LVCMOS33 } [get_ports { ck_io[21] }]; #IO_L1P_T0_AD0P_35
Sch=ad_p[0]
#set_property -dict { PACKAGE_PIN F20   IOSTANDARD LVCMOS33 } [get_ports { ck_io[22] }]; #IO_L15N_T2_DQS_AD12N_35
Sch=ad_n[12]
#set_property -dict { PACKAGE_PIN F19   IOSTANDARD LVCMOS33 } [get_ports { ck_io[23] }]; #IO_L15P_T2_DQS_AD12P_35
Sch=ad_p[12]
#set_property -dict { PACKAGE_PIN A20   IOSTANDARD LVCMOS33 } [get_ports { ck_io[24] }]; #IO_L2N_T0_AD8N_35
Sch=ad_n[8]
#set_property -dict { PACKAGE_PIN B19   IOSTANDARD LVCMOS33 } [get_ports { ck_io[25] }]; #IO_L2P_T0_AD8P_35
Sch=ad_p[8]

##ChipKit Digital I/O High

#set_property -dict { PACKAGE_PIN U5    IOSTANDARD LVCMOS33 } [get_ports { ck_io[26] }]; #IO_L19N_T3_VREF_13
Sch=ck_io[26]
#set_property -dict { PACKAGE_PIN V5    IOSTANDARD LVCMOS33 } [get_ports { ck_io[27] }]; #IO_L6N_T0_VREF_13
Sch=ck_io[27]
#set_property -dict { PACKAGE_PIN V6    IOSTANDARD LVCMOS33 } [get_ports { ck_io[28] }]; #IO_L22P_T3_13 Sch=ck_io[28]
#set_property -dict { PACKAGE_PIN U7    IOSTANDARD LVCMOS33 } [get_ports { ck_io[29] }]; #IO_L11P_T1_SRCC_13
Sch=ck_io[29]
#set_property -dict { PACKAGE_PIN V7    IOSTANDARD LVCMOS33 } [get_ports { ck_io[30] }]; #IO_L11N_T1_SRCC_13
Sch=ck_io[30]
```

```
#set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS33 } [get_ports { ck_io[31] }]; #IO_L17N_T2_13 Sch=ck_io[31]
#set_property -dict { PACKAGE_PIN V8    IOSTANDARD LVCMOS33 } [get_ports { ck_io[32] }]; #IO_L15P_T2_DQS_13
Sch=ck_io[32]
#set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { ck_io[33] }]; #IO_L21N_T3_DQS_13
Sch=ck_io[33]
#set_property -dict { PACKAGE_PIN W10   IOSTANDARD LVCMOS33 } [get_ports { ck_io[34] }]; #IO_L16P_T2_13 Sch=ck_io[34]
#set_property -dict { PACKAGE_PIN W6    IOSTANDARD LVCMOS33 } [get_ports { ck_io[35] }]; #IO_L22N_T3_13 Sch=ck_io[35]
#set_property -dict { PACKAGE_PIN Y6    IOSTANDARD LVCMOS33 } [get_ports { ck_io[36] }]; #IO_L13N_T2_MRCC_13
Sch=ck_io[36]
#set_property -dict { PACKAGE_PIN Y7    IOSTANDARD LVCMOS33 } [get_ports { ck_io[37] }]; #IO_L13P_T2_MRCC_13
Sch=ck_io[37]
#set_property -dict { PACKAGE_PIN W8    IOSTANDARD LVCMOS33 } [get_ports { ck_io[38] }]; #IO_L15N_T2_DQS_13
Sch=ck_io[38]
#set_property -dict { PACKAGE_PIN Y8    IOSTANDARD LVCMOS33 } [get_ports { ck_io[39] }]; #IO_L14N_T2_SRCC_13
Sch=ck_io[39]
#set_property -dict { PACKAGE_PIN W9    IOSTANDARD LVCMOS33 } [get_ports { ck_io[40] }]; #IO_L16N_T2_13 Sch=ck_io[40]
#set_property -dict { PACKAGE_PIN Y9    IOSTANDARD LVCMOS33 } [get_ports { ck_io[41] }]; #IO_L14P_T2_SRCC_13
Sch=ck_io[41]
#set_property -dict { PACKAGE_PIN Y13   IOSTANDARD LVCMOS33 } [get_ports { ck_io[42] }]; #IO_L20N_T3_13 Sch=ck_ioa

## ChipKit SPI

#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports { ck_miso }]; #IO_L10N_T1_34 Sch=ck_miso
#set_property -dict { PACKAGE_PIN T12   IOSTANDARD LVCMOS33 } [get_ports { ck_mosi }]; #IO_L2P_T0_34 Sch=ck_mosi
#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { ck_sck }]; #IO_L19P_T3_35 Sch=ck_sck
#set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports { ck_ss }]; #IO_L6P_T0_35 Sch=ck_ss

## ChipKit I2C

#set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { ck_scl }]; #IO_L24N_T3_34 Sch=ck_scl
#set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { ck_sda }]; #IO_L24P_T3_34 Sch=ck_sda

##HDMI Rx

#set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { hdmi_rx_cec }]; #IO_L13N_T2_MRCC_35
Sch=hdmi_rx_cec
#set_property -dict { PACKAGE_PIN P19   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_clk_n }]; #IO_L13N_T2_MRCC_34
Sch=hdmi_rx_clk_n
#set_property -dict { PACKAGE_PIN N18   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_clk_p }]; #IO_L13P_T2_MRCC_34
Sch=hdmi_rx_clk_p
#set_property -dict { PACKAGE_PIN W20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_n[0] }]; #IO_L16N_T2_34
Sch=hdmi_rx_d_n[0]
#set_property -dict { PACKAGE_PIN V20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_p[0] }]; #IO_L16P_T2_34
Sch=hdmi_rx_d_p[0]
#set_property -dict { PACKAGE_PIN U20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_n[1] }]; #IO_L15N_T2_DQS_34
Sch=hdmi_rx_d_n[1]
#set_property -dict { PACKAGE_PIN T20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_p[1] }]; #IO_L15P_T2_DQS_34
Sch=hdmi_rx_d_p[1]
#set_property -dict { PACKAGE_PIN P20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_n[2] }]; #IO_L14N_T2_SRCC_34
Sch=hdmi_rx_d_n[2]
#set_property -dict { PACKAGE_PIN N20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_p[2] }]; #IO_L14P_T2_SRCC_34
Sch=hdmi_rx_d_p[2]
#set_property -dict { PACKAGE_PIN T19   IOSTANDARD LVCMOS33 } [get_ports { hdmi_rx_hpd }]; #IO_25_34 Sch=hdmi_rx_hpd
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { hdmi_rx_scl }]; #IO_L11P_T1_SRCC_34
Sch=hdmi_rx_scl
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports { hdmi_rx_sda }]; #IO_L11N_T1_SRCC_34
Sch=hdmi_rx_sda

##HDMI Tx

#set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports { hdmi_tx_cec }]; #IO_L19N_T3_VREF_35
Sch=hdmi_tx_cec
#set_property -dict { PACKAGE_PIN L17   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_clk_n }]; #IO_L11N_T1_SRCC_35
Sch=hdmi_tx_clk_n
#set_property -dict { PACKAGE_PIN L16   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_clk_p }]; #IO_L11P_T1_SRCC_35
Sch=hdmi_tx_clk_p
```

```
#set_property -dict { PACKAGE_PIN K18   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_n[0] }]; #IO_L12N_T1_MRCC_35
Sch=hdmi_tx_d_n[0]
#set_property -dict { PACKAGE_PIN K17   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_p[0] }]; #IO_L12P_T1_MRCC_35
Sch=hdmi_tx_d_p[0]
#set_property -dict { PACKAGE_PIN J19   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_n[1] }]; #IO_L10N_T1_AD11N_35
Sch=hdmi_tx_d_n[1]
#set_property -dict { PACKAGE_PIN K19   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_p[1] }]; #IO_L10P_T1_AD11P_35
Sch=hdmi_tx_d_p[1]
#set_property -dict { PACKAGE_PIN H18   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_n[2] }];
#IO_L14N_T2_AD4N_SRCC_35 Sch=hdmi_tx_d_n[2]
#set_property -dict { PACKAGE_PIN J18   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_p[2] }];
#IO_L14P_T2_AD4P_SRCC_35 Sch=hdmi_tx_d_p[2]
#set_property -dict { PACKAGE_PIN R19   IOSTANDARD LVCMOS33 } [get_ports { hdmi_tx_hpdn }]; #IO_0_34
Sch=hdmi_tx_hpdn
#set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { hdmi_tx_scl }]; #IO_L8P_T1_AD10P_35
Sch=hdmi_tx_scl
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { hdmi_tx_sda }]; #IO_L8N_T1_AD10N_35
Sch=hdmi_tx_sda

##Crypto SDA

#set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { crypto_sda }]; #IO_25_35 Sch=crypto_sda
```

Bibliography

[1] EDA Playground. EDA Playground. [online] Available at:
https://www.edaplayground.com/ [Last Accessed, 17 Dec, 2018]

[2] Xilinx. Vivado Design Suite - HLx Editions. [online] Available at:
https://www.xilinx.com/support/download.html [Last Accessed, 17 Dec, 2018]

[3] Digilent. PYNQ-Z1. [online] Available at:
https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start [Last Accessed, 17 Dec, 2018]

[4] The ZYNQ Book Tutorials. Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, Robert W. Stewart [online PDF] Available at:
http://www.zynqbook.com/download-tuts.html [Last Accessed, 17 Dec, 2018]