Bruno E. Gracia Villalobos & Bobby Reyes
EE 5193
Professor Eratne
Implementation of a 16-bit RISC architecture CPU on Nexys 4 DDR FPGA
August 5th, 2019

TABLE OF CONTENTS

ABSTRACT
This project report explores the design of a 16-bit processor to implement a CPU with a Reduced Instruction Set Computer (*RISC)* architecture on the Nexys-4 DDR FPGA Development Board. The processor can take 16-bit instruction codes from user or store instructions in main memory and perform the operation described in the instruction.

PROBLEM
RISC is a CPU design strategy based on the insight that a simplified instruction set gives higher performance when combined with a microprocessor architecture which has the ability to execute the instructions by using some microprocessor cycles per instruction.[1] These cycles vary but are typically based on 5 steps used to access, and execute an instruction. These steps make up what is known as the RISC data *pipeline* and are shown in Table 1 along with their respective purpose and associated hardware.

Pipeline Stage	Meaning
Fetch	CPU receives Instruction
Decode	CPU understands instruction
Execute	CPU performs computation
Access	CPU accesses required data from registers or memory
Write Back	CPU stores result to register or memory

Table 1- RISC pipeline stages and meaning

This purpose of this project is to implement the 16-bit RISC Processor architecture from Figure 1 on the Nexys-4 DDR FPGA. The custom RISC instruction set used is capable of performing operations based on Arithmetic, Logical, Memory, Branch, and Jump instruction codes.

[1] Agarwal, Tarun, et al. "What Is RISC and CISC Architecture ? Edgefxkits." *Edgefx Kits Official Blog*, 19 Feb. 2015, www.edgefxkits.com/blog/what-is-risc-and-cisc-architecture/.

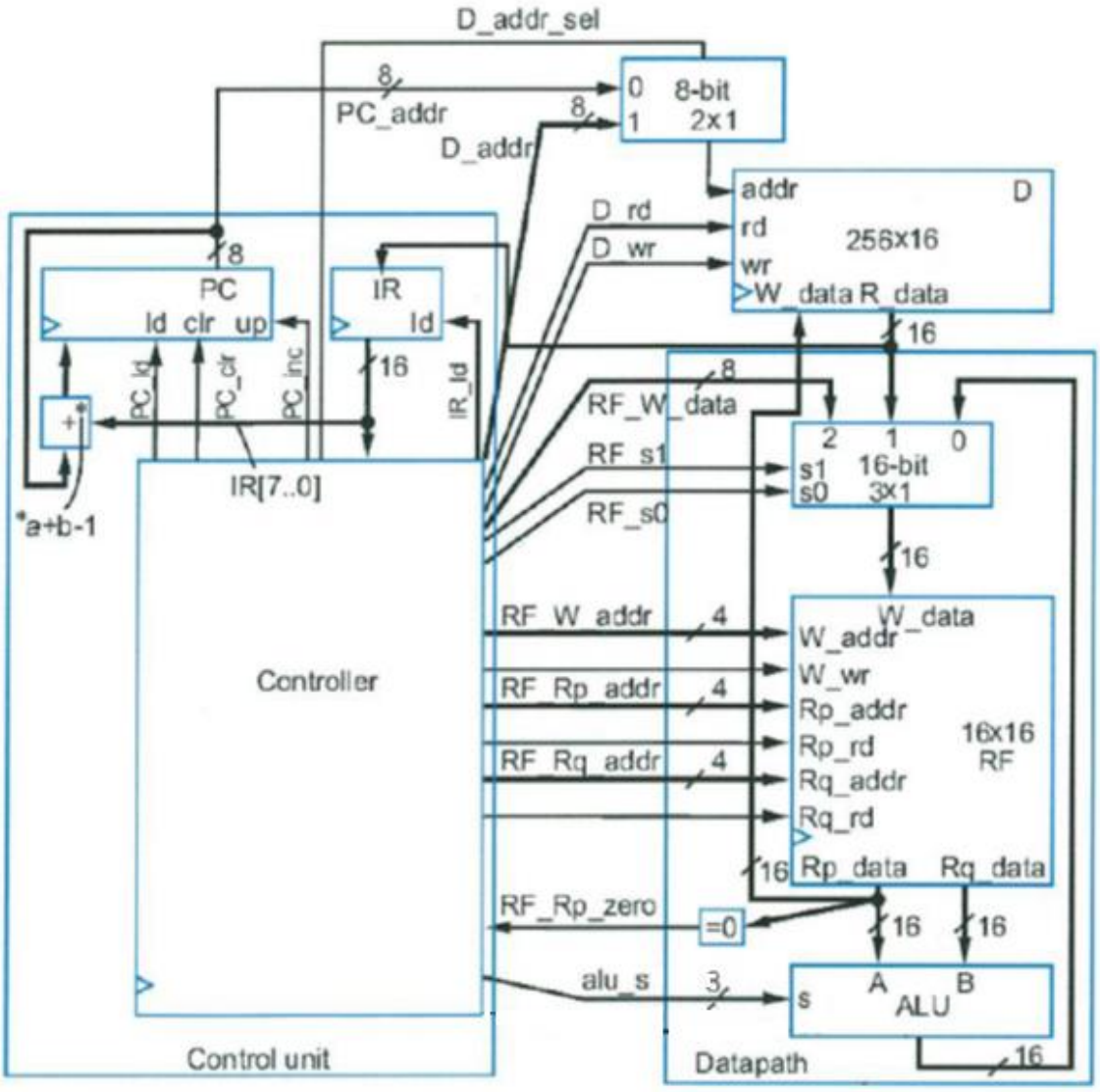


Figure 1 - 16-bit RISC Processor Architecture

SOLUTION

The approach for designing a CPU can be broken into two parts, the *controller*, and the *datapath*; these parts of the CPU are shown in Figure 1. By separating these two parts, the design of the system is simplified.This *application driven* design determines the Finite State Machine (FSM) that controls the datapath, by the instructions that must be executed. Therefore, the *control unit* can be designed as a FSM that drives the *pipeline* stages that send the signals to the *datapath unit* to perform the operations encoded in the instruction set.

RISC INSTRUCTION SET

Before explaining the design of the datapath unit and control unit, the RISC instruction set is presented to formulate an understanding of the processor. The design consists of a 16-bit instruction, 4-bit register address space, and an 8-bit memory address space; this maps to 16 registers and 256 memory locations for the control unit to work with. Additionally, there are 16 different instructions, both memory and register locations have a width of 16-bits, and memory accesses are restricted to a “load-store” method—memory needs to be loaded into a register before it can be altered.

Each 16-bit instruction contains fields that represent the opcode, destination, source, and target as subsets of the instruction. However, the use of the latter three fields varies depending on the different types of operations: arithmetic, logical, memory, conditional branch, and jump; the opcode field is static for all operations. The full instruction set is presented in Figures 2-6.

Operation	Opcode	Destination	Source	Target	Description
ADD	0000	Rd	Rs	Rt	**ADD**: Rd = Rs + Rt Operands A and B stored in register locations Rs and Rt are added and written to the destination register specified by Rd.
SUB	0001	Rd	Rs	Rt	**SUB**: Rd = Rs - Rt Operand B (Rt) is subtracted from Operand A (Rs) and written to Rd.

Figure 2 - Arithmetic ALU operations

Figure 2 showcases two instructions that are used to perform basic arithmetic calculations using the Arithmetic Logic Unit (ALU): addition and subtraction. For both of these instructions, the 16-bit address is subdivided into four different fields: the opcode, destination, source, and target. All of these fields are 4-bits, and they map to one of the 16 available registers. Essentially, the source and target fields are register addresses that each store a 16-bit operand, and the destination field is the desired register address to store the answer.

Operation	Opcode	Destination	Source	Target	Description
AND	0011	Rd	Rs	Rt	**AND**: Rd = Rs & Rt Operand A (Rs) is bitwise anded with Operand B (Rt) and written into Rd.
OR	0011	Rd	Rs	Rt	**OR**: Rd = Rs | Rt Operand A (Rs) is bitwise ored with Operand B (Rt) and written into Rd.
XOR	0100	Rd	Rs	Rt	**XOR**: Rd = Rs ^ Rt Operand A (Rs) is bitwise Xored with Operand B (Rt) and written into Rd.
NOT	0101	Rd	Rs		**NOT**: Rd = ~Rs Operand A (Rs) is bitwise inverted and written into Rd.
SLA	0110	Rd	Rs		**SLA**: Rd = Rs << 1 Operand A (Rs) is arithmetically shifted to the left by one bit and written into Rd.
SRA	0111	Rd	Rs		**SRA**: Rd = Rs >> 1 Operand A (Rs) is arithmetically shifted to the right by one bit and written into Rd. The MSB (sign bit) will be preserved for this operation.

Figure 3 - Logical ALU operations

Figure 3 showcases 5 logical operations performed by the ALU. The first three, AND, OR, and XOR, all make use of the same fields of the 16-bit instruction for destination, source, and target. However, operations NOT, SLA, and SRA discard the use of bits <3:0> for target field and only make use of the same destination and source fields as AND, OR, and XOR. This is because of the nature of their operation: only one 16-bit number from the source field is being altered.

Operation	Opcode	Destination	Source	Target	Description
LI	1000	Rd	Imm		**LI:** Rd = 8-bit Sign extended Immediate The 8-bit immediate in the Instruction word is sign-extended to 16-bits and written into the register specified by Rd.
LW	1001	Rd	Dir		**LW:** Rd = Mem[Dir] The memory word specified by the address Dir is loaded into register Rd.
SW	1010	Rt	Dir		**SW:** Mem[Dir] = Rt The data in register Rt is stored into the memory location Dir.

Figure 4 - Memory Access operations

Figure 4 contains two instructions that describe the load-store nature of the RISC instruction set: LW and SW. Given they require the need to specify a memory address, both source and target fields, each being 4-bit wide, are concatenated to create the required 8-bit length to select one of the 256 memory addresses. For the LW operation, bits <7:0> contain the memory address of the word to fetch, and bits <11:8> specify the destination register. On the other hand, the SW operation stores the contents of the register specified in bits<11:8> in the memory location specified by the lower 8 bits. The LI instruction is in this category, not because of being a load-store operation, but because of its need to specify an immediate—an 8-bit constant.

Operation	Opcode	Destination	Source	Target	Description
BIZ	1011	Rs	Offset		**BIZ:** PC = PC + 1 + Offset **if** Rs = 0 If all the bits in register Rs are zero then the current Program Count (PC + 1) is offset to PC + 1 + Offset. The count is offset from PC + 1 because it is incremented and stored during the Fetch cycle.
BNZ	1100	Rs	Offset		**BNZ:** PC = PC + 1 + Offset **if** Rs! = 0 If all the bits in register Rs are not zero than the current Program Count (PC + 1) is offset to PC + 1 + Offset.

Figure 5 - Conditional Branch operations

Figure 5 showcases the conditional branch operations BIZ (branch if zero), and BNZ (branch if not zero). These two instructions also make use of the concatenated source and target fields to create an 8-bit offset for the Program Counter. Furthermore, the Destination field is now used as source field, and is evaluated bitwise for the instruction's respective operation.

Operation	Opcode	Destination	Source	Target	Description
JAL	1101	Rd	Offset		**JAL**: Rd = PC + 1 and PC = PC + 1 + Offset Jump and Link instruction would write current Program Count in register Rd and offset the program count to PC + 1 + Offset
JMP	1110		Offset		**JMP**: PC = PC + 1 + Offset Unconditional jump instruction will offset the program count to PC + 1 + Offset.
JR	1111		Rs		**JR**: PC = Rs Jump Return instruction will set the Program Count to the one previously stored in JAL.

Figure 6 - Jump operations

Lastly, Figure 6 defines the three Jump operations for the instruction set: JAL, JMP, and JR. The former two instructions again make use of the lower 8 bits for an offset to the Program Counter, and the latter instruction only makes use of bits <7:4> as the source register. The jump operations are distinct since they are unconditional compared to the branch operations.

RISC INSTRUCTION EXAMPLE

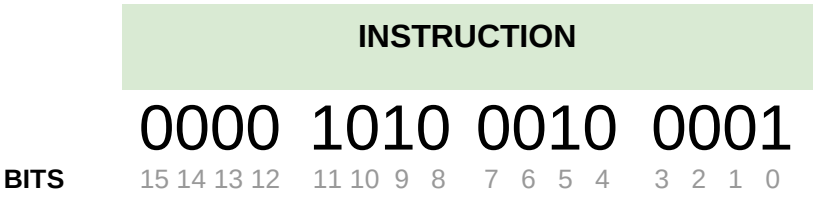


Figure 7 - General 16-Bit RISC Instruction

To better understand the instruction set, an example is presented in Figure 7. First, since the opcode field is static, bits<15:12> are decoded and found to represent the ADD operation. With this in mind, the other three fields can now be formulated as the destination, source, and target registers. Figure 8 showcases this simplified.

Figure 8 - ADD Instruction

Now that the instruction is decoded, the operation can be visualized easier. Here is the operation in algebraic form.

Destination = reg[Source] + reg[Target]

(Register 10) = data(Register 2) + data(Register 1)

The source and target registers contain the operands of the ADD operation, and they will be fetched and sent to the ALU for addition. Once the addition is complete, the result will be sent back to the Register Bank and stored in the destination register specified by bits<11:8>. Next, a design overview of the RISC Processor will be discussed.

RISC PROCESSOR DESIGN OVERVIEW

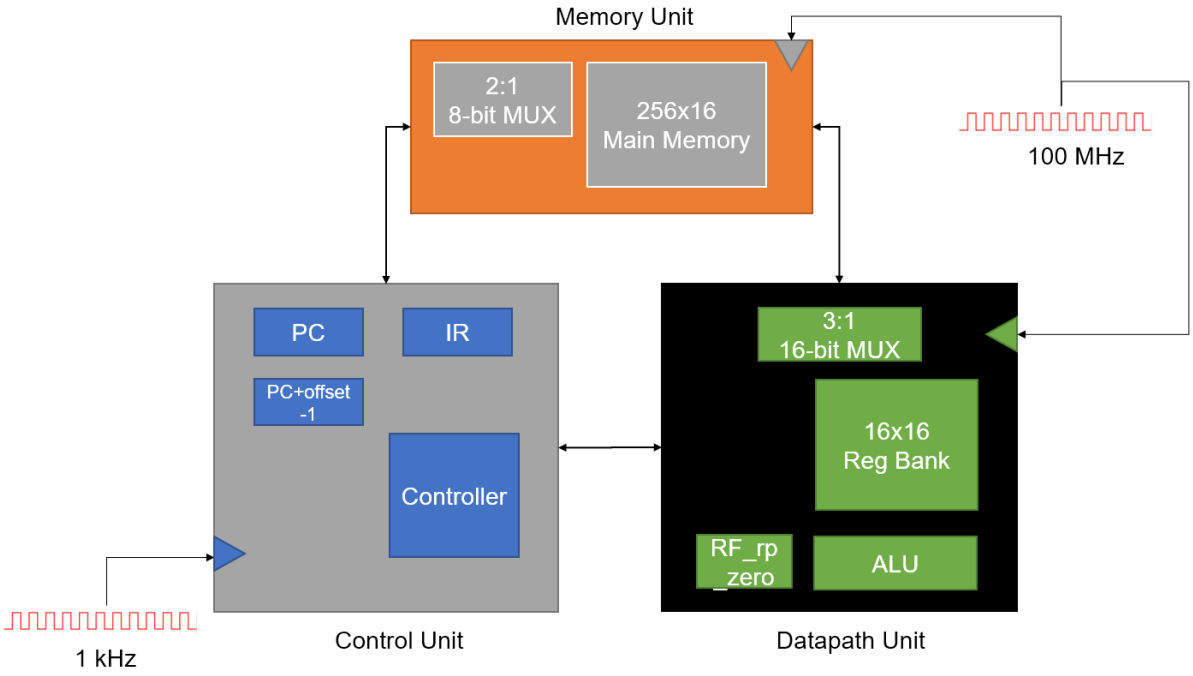


Figure 9 - Control Unit, Memory Unit, Datapath Unit

To simplify design, the processor is divided into three distinct units: the control, memory, and datapath units. In a high level, the control unit fetches the instruction, decodes it, and sends the appropriate signals to both memory and datapath units to execute the instruction. The datapath unit controls access to the register bank, does arithmetic and logical operations, and communicates with both control and memory units. The memory unit controls access to main memory, and communicates with both control and datapath units. This high level overview is showcased in Figure 9 and the HDL code reflects this hierarchy as it is shown—blocks inside the units are submodules to the unit module.

It is important to mention the Control Unit operates at a significantly lower clock speed—1:10,000—of the Datapath and Memory Units. This helps to avoid race conditions between control signals from the Controller and the outputs from both Datapath and Memory Units. The following sections explain each of the units' internal and external function.

CONTROL UNIT

```
PC pc1(
.ld(PC_ld),
.clr(clr),
.inc(PC_inc),
.CLK100MHZ(CLK100MHZ),
.load(instruction),
.PC_addr(PC_addr),
.flag(flag)
);

IR ir1(
.ins_in(R_data),
.IR_ld(IR_ld),
.CLK100MHZ(CLK100MHZ),
.ins_out(instruction)
);
```

The Control Unit consists of the Program Counter, the Instruction Register, and the Controller. The latter submodule is embedded into the Control Unit module for a simpler HDL implementation. Both the PC and IR are shown instantiated inside the Control Unit module in Figure 10-1. The following subsections will explain in greater detail the function and portlists of both of these submodules, and the Control Unit as a whole from the Controller's standpoint.

Figure 10-1 - Control Unit submodules

Program Counter

```
module PC(                          // PC current + DELTA - 1;
    input ld,                       // Control signal to load PC with offset
    input clr,                      // Control signal to clear PC to first location
    input inc,                      // Control signal to increment PC by 1
    input CLK100MHZ,                // Clock
    input  [7:0] load,              // Loads the PC with offset
    output [7:0] PC_addr,           // For use with branch/jmp
    output reg flag                 // To signal done
);
```

Figure 10-2 - Program Counter portlist

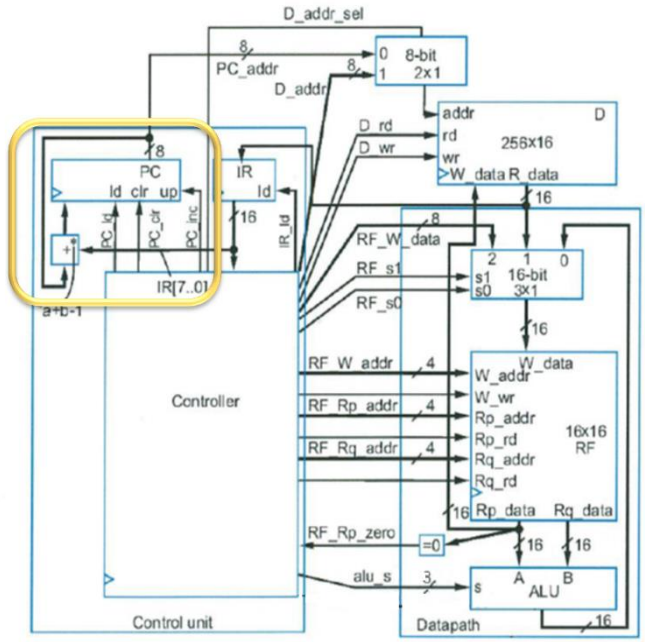


Figure 11 - Program Counter in the Overall Design

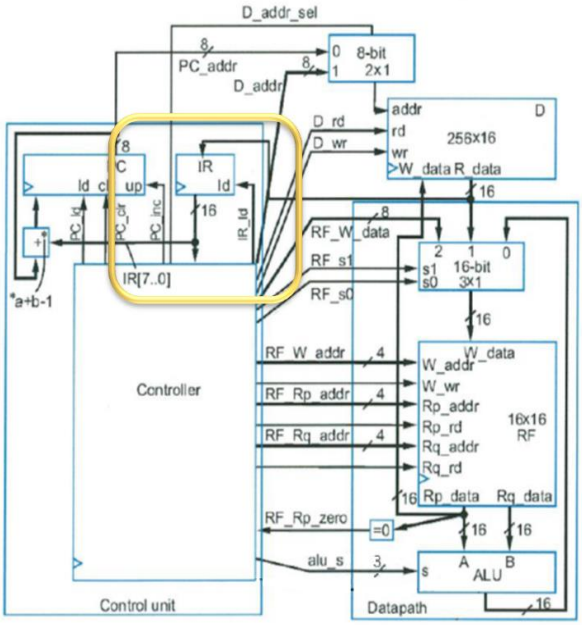
The program counter stores the program(s) that the processor is running and will run in the future. It stores the memory locations of each of the instructions to execute in a sequential manner, with exception to conditional branches and jumps. After the current instruction is executed, the Program Counter will load in the new address from the Memory Bank.

In the HDL code, Figure 10-2 showcases the portlist of the module and explains each of the inputs and outputs. Figure 11 shows the program counter inside the Control Unit and its connection to the Memory Unit, Control Unit, and Instruction Register.

Instruction Register

```
module IR(
    input [15:0] ins_in,            //Fetches the instruction from the Memory Unit
    input IR_ld,                    //Control signal from Control Unit to enable load
    input CLK100MHZ,                //Clock to control load of instruction
    output [15:0] ins_out           //Output for Control unit to decode
);
```

Figure 12 - Instruction Register portlist



Compared to the Program counter, the Instruction Register (IR) stores the current instruction, not the memory location. It's main purpose is to provide information about the instruction to the Control Unit. Its standalone submodule implementation helps provide a dedicated place to store the current instruction for easy access by the Controller. When a conditional branch or jump instruction is executing, the IR is in charge of sending the offset to the PC loader module to load the new address to the PC.

Figure 12 shows the portlist and explanation of the IR in the HDL code. Figure 13 shows the IR within the context of the design.

Figure 13 - Instruction Register in the Design

Controller (Control Unit)

```
module Controller(
    input BTNC,                     //For advancing the instruction in the FPGA
    input SLOWCLK,                  //The 1kHz Clock for the Control Unit
    input CLK100MHZ,                //100Mhz clock for the Memory and Datapath Units
    input [15:0] R_data,            //For fetching an instruction from Memory Unit
    input RF_Rp_zero,               //Flag for use with branch, jump instructions
    output wire [7:0] PC_addr,      //PC output signal
    output reg D_addr_sel,          //Control signal to select MUX in Memory Unit
    output reg [7:0] D_addr,        //Sending the address of the instruction to fetch
    output reg D_rd,                //Control signal to read mem
    output reg D_wr,                //Contorl signal to write to mem
    output reg [7:0] RF_W_data,     //For sending an immediate to Datapath unit
    output reg[1:0] RF,             //Selecting the MUX in datapath unit
    output reg [3:0] RF_W_addr,     //Send register address to write MUX output to
    output reg W_wr,                //Control signal to begin reg write
    output reg [3:0] RF_Rp_addr,    //Send reg address to READ from as Rp
    output reg Rp_rd,               //Control signal to begin read from Rp
    output reg [7:0] RF_Rq_addr,    //Send reg address to Read from as Rq
    output reg Rq_rd,               //Control signal to begin read from Rq
    output reg [4:0] alu_s          //Select ALU operation within Datapath Unit
);
```

Figure 14 - Controller module portlist

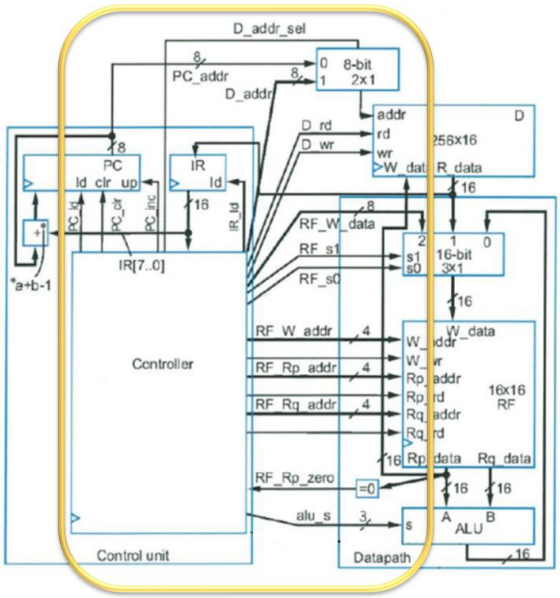


Figure 15 - Control Unit in the Design

The controller is the main operator in the design. It sends control signals to both the Datapath and Memory units to execute the instruction. The controller also controls the FSM of the design, and advances the instruction in the pipeline. In the HDL code, both PC and IR submodules are embedded within the Controller module for clarity, and therefore the Controller essentially serves as the entire Control Unit. Figure 14 showcases the extensive portlist of the Controller and explains each one with an adjacent comment. Figure 15 displays the Control Unit amidst the Datapath and Memory Units.

DATAPATH UNIT

```
regbank r1(
.CLK100MHZ(CLK100MHZ),
.W_data(W_data),
.W_addr(RF_W_addr),
.W_wr(W_wr),
.Rp_addr(RF_Rp_addr),
.Rp_rd(Rp_rd),
.Rq_addr(RF_Rq_addr),
.Rq_rd(Rq_rd),
.Rp_data(Rp_data),
.Rq_data(Rq_data)
);
mux3_1 m1(
.RF_W_data(RF_W_data),
.R_data(R_data),
.ALU_data(ALU_data),
.RF(RF),
.W_data(W_data)
);
ALU alu1(
.A(Rp_data),
.B(Rq_data),
.alu_s(alu_s),
.ALU_data(ALU_data)
);
Zero z1(
.Rp_data(Rp_data),
.RF_Rp_zero(RF_Rp_zero)
);
```

The Datapath Unit consists of a 3:1 16-bit MUX to control the incoming data to the Register Bank, the 16x16-bit Register Bank, and a combinational circuit RF_rp_zero for use with conditional branching instructions. Compared to the Control Unit, the Datapath Unit makes use of the 100MHz clock of the design. As mentioned on page 9, the objective of this approach is to have the fastest operation possible of the Control Unit's peripherals—Datapath and Memory Units—to prevent race conditions between the Control Unit's next signal execution in the ASM. Figure 16 showcases the instantiated submodules, and the following subsections explain these in greater detail.

Figure 16-1 - Datapath Unit HDL

3:1 16-bit MUX

```
module mux3_1(
    input [7:0] RF_W_data,      //Sel=2: 8-bit immediate incoming from Control Unit
    input [15:0] R_data,        //Sel=1: 16-bit data from Memory Unit
    input [15:0] ALU_data,      //Sel=0: 16-bit output from ALU
    input [1:0] RF,             //2 bit select line for the above
    output reg [15:0] W_data    //16-bit output of the 3:1 MUX
);
```

Figure 16-2 - 3:1 16-bit MUX portlist

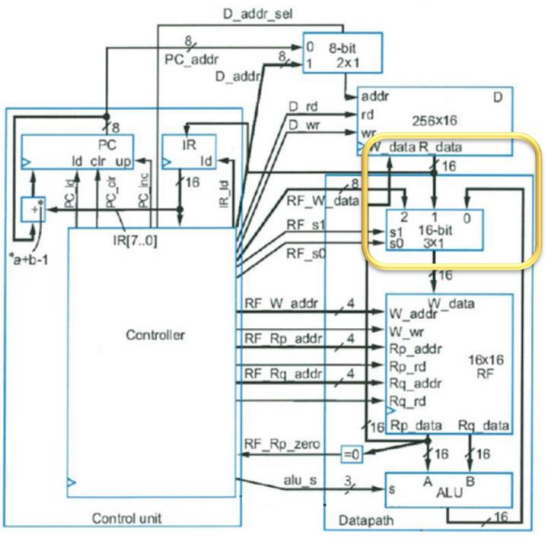


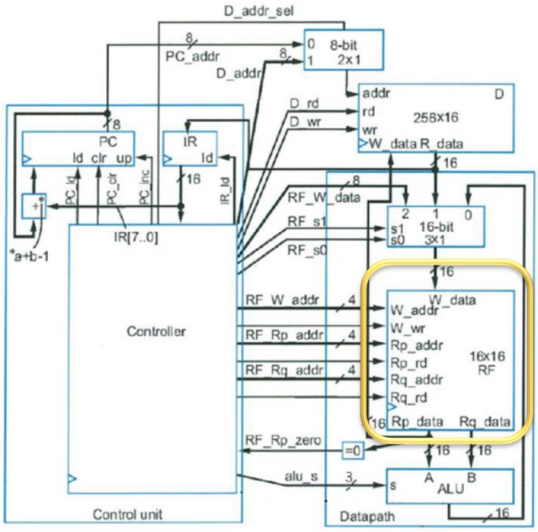
Figure 17 - 3:1 16-bit MUX in the Design

The MUX controls three different datalines into the Register Bank: the output from the ALU, Memory from the Memory Bank, and the 8-bit immediate from the Control Unit (when LI instruction is called). Given the difference in width of the immediate and MUX, the submodule sign extends the immediate to 16 bits. Although this approach might not be appropriate to have within a dedicated MUX module, it was chosen to simplify the design. A control signal from the Controller selects the line to be sent to the Register Bank. Figure 16-2 displays the portlist of the MUX submodule, and Figure 17 gives context of the submodule within the overall design.

16x16-bit Register Bank

```
module regbank(
    input CLK100MHZ,            //Controls writes and reads at 100MHz
    input [15:0] W_data,        //Data to write to register from 3:1 16-bit MUX
    input [3:0] W_addr,         //Address in register bank to write MUX data to
    input W_wr,                 //Control signal to enable MUX data write to reg
    input [3:0] Rp_addr,        //Address to read register output Rp from
    input Rp_rd,                //Control signal to enable Rp read
    input [3:0] Rq_addr,        //Address to read register output Rq from
    input Rq_rd,                //Control signal to enable Rq read
    output reg [15:0] Rp_data,  //Output contents of Rp_address read
    output reg [15:0] Rq_data   //Output contents of Rq_address read
);
```

Figure 18 - Register Bank portlist



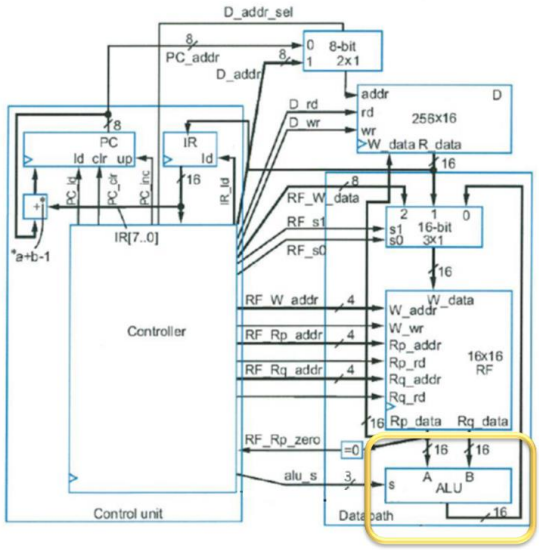
The Register Bank consists of 16, 16-bit registers for general purpose. Given the RISC processor's "load-store" approach to memory access, the register bank is extensively used for all instructions. Primarily, it receives signals from the Control Unit to read or write data, outputs data to the ALU for execution, outputs Rp to the RF_rp_zero flag submodule, and receives data from the 3:1 16-bit MUX (Figure 19). The Register Bank can only output two register's contents at the same time, which effectively gives it a 32-bit width data output bus. However, both Rp and Rq are split and sent individually as 16-bits each (Figure 18).

Figure 19 - Register Bank in the Design

16-bit ALU

```
module ALU(
    input [15:0] A,                 //16-bit A Operand from Rp_data output from Reg Bank
    input [15:0] B,                 //16-bit B Operand from Rq_data output from Reg Bank
    input [3:0] alu_s,              //4-bit control signal from CU to select operation
    output reg [15:0] ALU_data      //16-bit output from the ALU
);
```

Figure 20 - 16-bit ALU portlist



The 16-bit ALU is tasked with applying a logical or arithmetic operation on two operands that stem from the Register Bank. With a 4-bit select line from the Control Unit, the ALU operates up to 16 instructions, 8 of which have been implemented for the instructions specified in the *RISC INSTRUCTION SET* section in pages 4-7; The 4-bit select line is chosen over a 3-bit select line to allow for future expansion of the submodule. Essentially, the ALU outputs a 16-bit bus to the 3:1 16-bit MUX as explained in Figure 20. Figure 21 showcases the ALU amidst the Design.

Figure 21 - 16-bit ALU in the Design

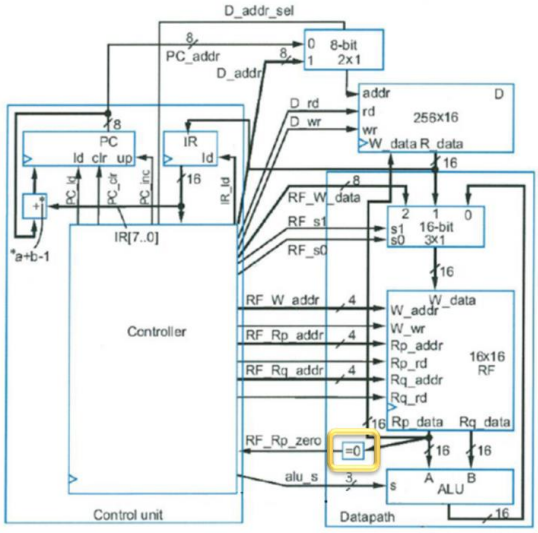
RF_rp_zero

```
module Zero(
    input [15:0] Rp_data,    //Get 15-bit Rp input from Register Bank
    output RF_Rp_zero        //Output flag = 1 if Rp is 16'h0000, 0 otherwise
);

    assign RF_Rp_zero = ~(Rp_data);

endmodule
```

Figure 22 - RF_rp_zero submodule portlist



The RF_rp_zero submodule serves as combinational circuit to check if the output Rp from the Register Bank is zero. This is useful for evaluating the BIZ and BNZ conditional branch instructions, and therefore is modularized for allowing the Control Unit to check with a dedicated line in its portlist. Figure 22 showcases the HDL code for the submodule: the reduction NOT operator is used to invert 16-bit Rp_data line from the Register Bank and assign the one bit answer to its output. Figure 23 displays the submodule in the Design.

Figure 23 - RF_rp_zero in the Design

MEMORY UNIT

```
mux2_1 m1(
.D_addr_sel(D_addr_sel),
.PC_addr(PC_addr),
.D_addr(D_addr),
.addr(addr)
);

main_mem m2(
.addr(addr),
.D_rd(D_rd),
.D_wr(D_wr),
.CLK100MHZ(CLK100MHZ),
.W_data(Rp_data),
.R_data(temp)
);
```

The Memory Unit consists of two submodules: a 256x16-bit Main Memory bank, and a 2:1 8-bit MUX to control the memory address to write or read from. Similar to the Datapath Unit, the Memory Unit also takes advantage of the 100MHz clock to operate as fast as possible to avoid race conditions with the Control Unit. Furthermore, its primary use is to store the programs to run, provide storage for register contents, and save PC values for conditional branch and jump instructions. Figure 24-1 displays the instantiated submodules which are explained in the following subsections.

Figure 24 - Memory Unit portlist

2:1 8-bit MUX

```
module mux2_1(
    input D_addr_sel,       //Selects the output of the 2:1 MUX to control input addr
    input [7:0] PC_addr,    //Sel 0: Bus from PC to fetch instruction from Memory
    input [7:0] D_addr,     //Sel 1: Dedicated bus from CU to address the Memory
    output [7:0] addr       //Outputs 8-bit address to Memory Module
);

assign addr = (D_addr_sel) ? D_addr : PC_addr;

endmodule
```

Figure 25 - 2:1 8-bit MUX module

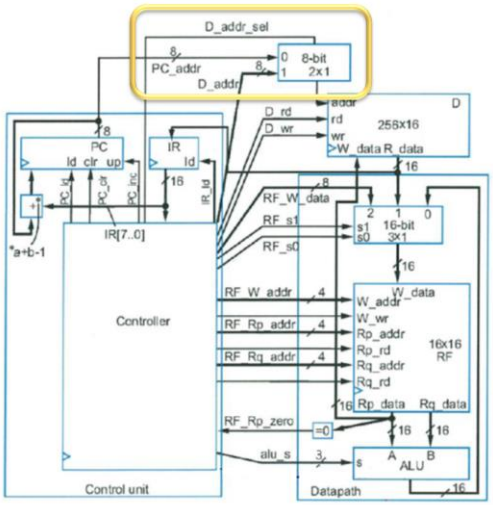


Figure 26 - 2:1 8-bit MUX in the Design

The main function of the MUX is to select the input address of the Memory submodule with the PC_addr from the PC or D_addr from the Control Unit. The Control Unit selects the output of the MUX with D_addr_sel. The HDL code is simple and the entire module is displayed in 25 alongside the portlist. An assign statement creates a MUX using the conditional operator in Verilog, and assigns D_addr if select is 1 and assigns PC_addr if select is 0. Figure 26 showcases the MUX in the Design.

256x16-bit Main Memory

```
module main_mem(
    input [7:0] addr,           //Address to read from or write to
    input D_rd,                 //Control signal to read
    input D_wr,                 //Control signal to write
    input CLK100MHZ,            //100MHz clock for sequential logic op.
    input  [15:0] W_data,       //For use to write register contents into memory
    output reg [15:0] R_data    //16-bit memory content output bus
);
```

Figure 27 - Main Memory portlist

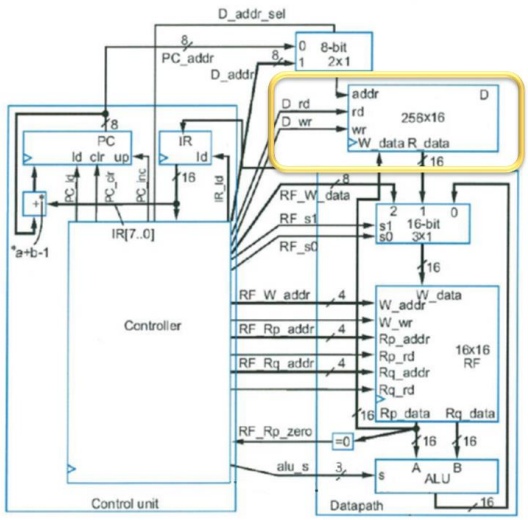


Figure 28 - Main Memory in the Design

This submodule contains the RISC Processor's onboard memory. With 256 rows, the 16-bit memory takes in an 8-bit address to select the address space. It receives two control signals, D_rd and D_wr, for reading and writing operations respectively. Unlike the MUX, this submodule contains sequential logic and operates in the fast 100 MHz clock that is an input to the Memory Unit module. Figure 27 displays the submodule's portlist with comments to each port's use. Figure 28 displays the submodule within the Memory Unit module and with respect to the overall design.

CONTROL UNIT ASM CHART

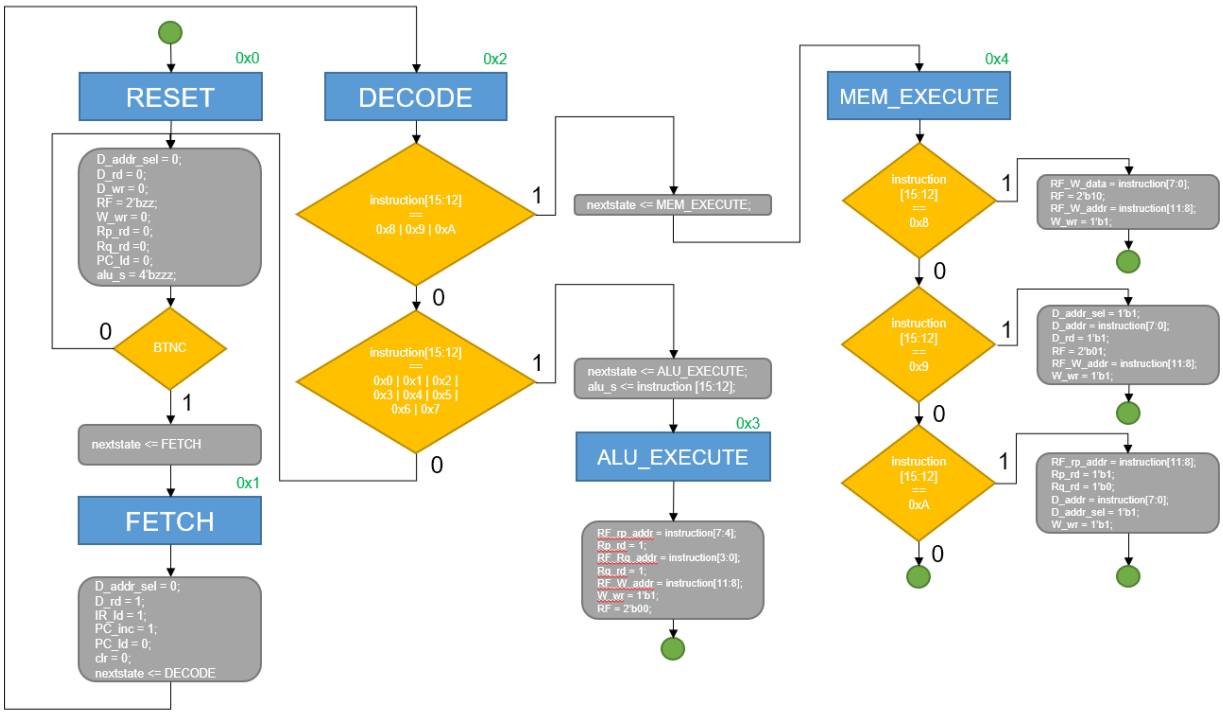


Figure 29 - Control Unit ASM Chart

The Control Unit ASM"s Chart is presented in this section and was chosen in this order to present the architecture before the behavior of the processor. Figure 29 displays the Control Unit ASM Chart. There are five states that the Control Unit cycles through: the Reset, Fetch, Decode, ALU_Execute and MEM_Execute. The Reset state sets the control signals to both the Memory Unit and Datapath Unit to 0. In this state, BTNC, a button in the FPGA, is checked for a press to advance to the next state to prevent the FPGA from cycling through all the instructions without the user noticing. Next, the Fetch state is present to fetch the address from the Memory Unit, and load it into the Instruction Register.

After fetching the address, the Control Unit goes to the Decode state, where the Instruction is analyzed for its opcode. Depending on the opcode, the Control Unit assigns the next state to either the MEM_Execute or ALU_Execute states, with the former used with load-store instructions and the latter with logical or arithmetic instructions. Once the Control Unit cycles through these two states, it goes back to the Reset state and awaits the user to press BTNC to fetch the next instruction.

FULL PROCESSOR SIMULATION SCREENSHOTS

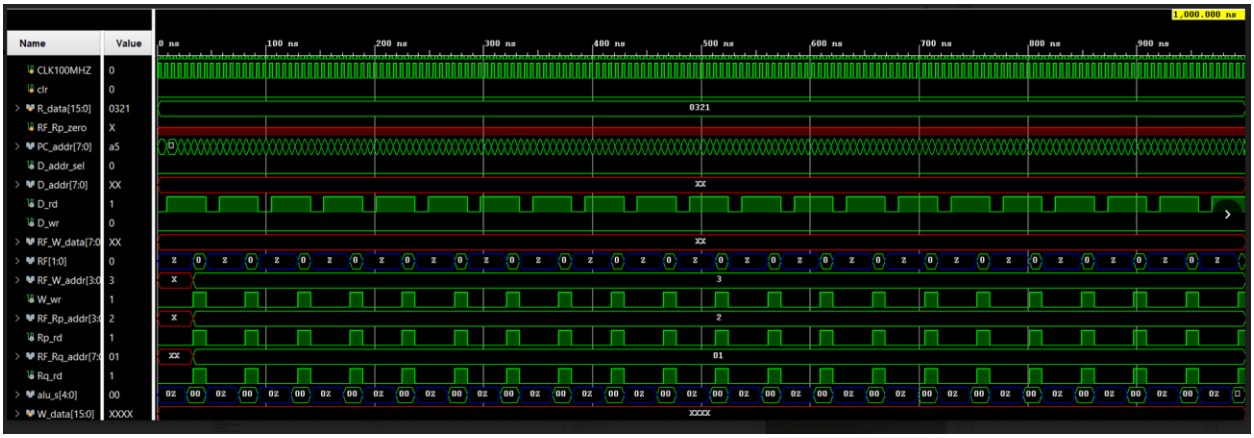


Figure 30 - Add Instruction 0x0321 Control Signals From Control Unit

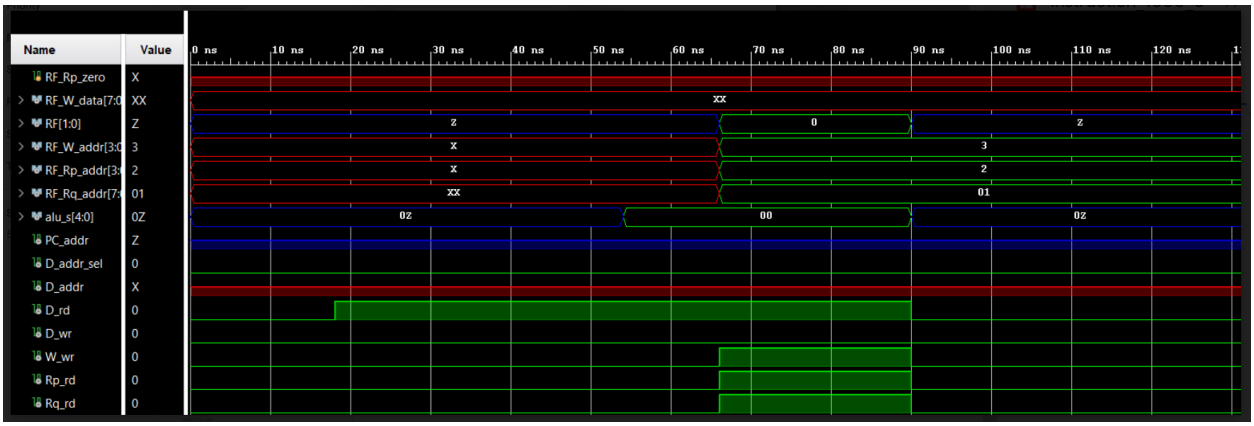


Figure 31 - Add Instruction 0x0321 Control Signals From Control Unit

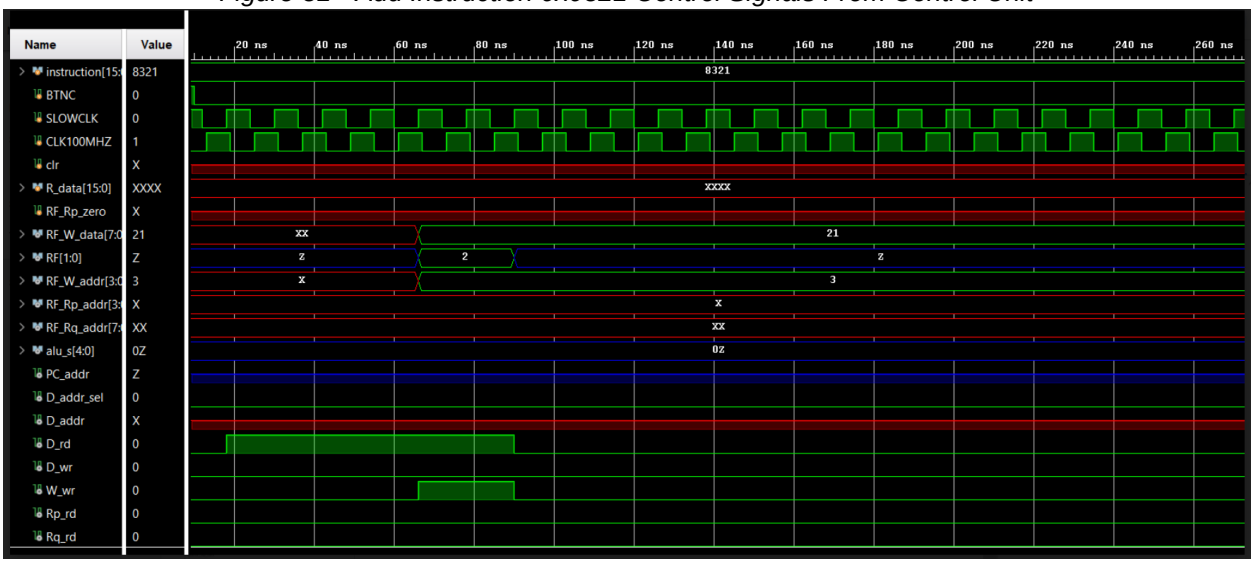


Figure 32 - LI Instruction 0x8321 Control Signals From Control Unit

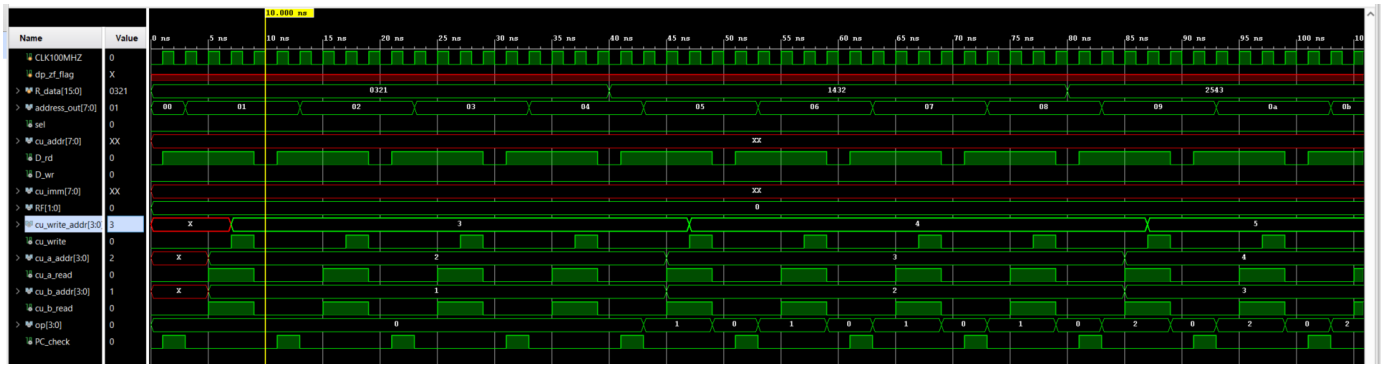


Figure 33 - Controller Unit outputting all control signals for ten instructions

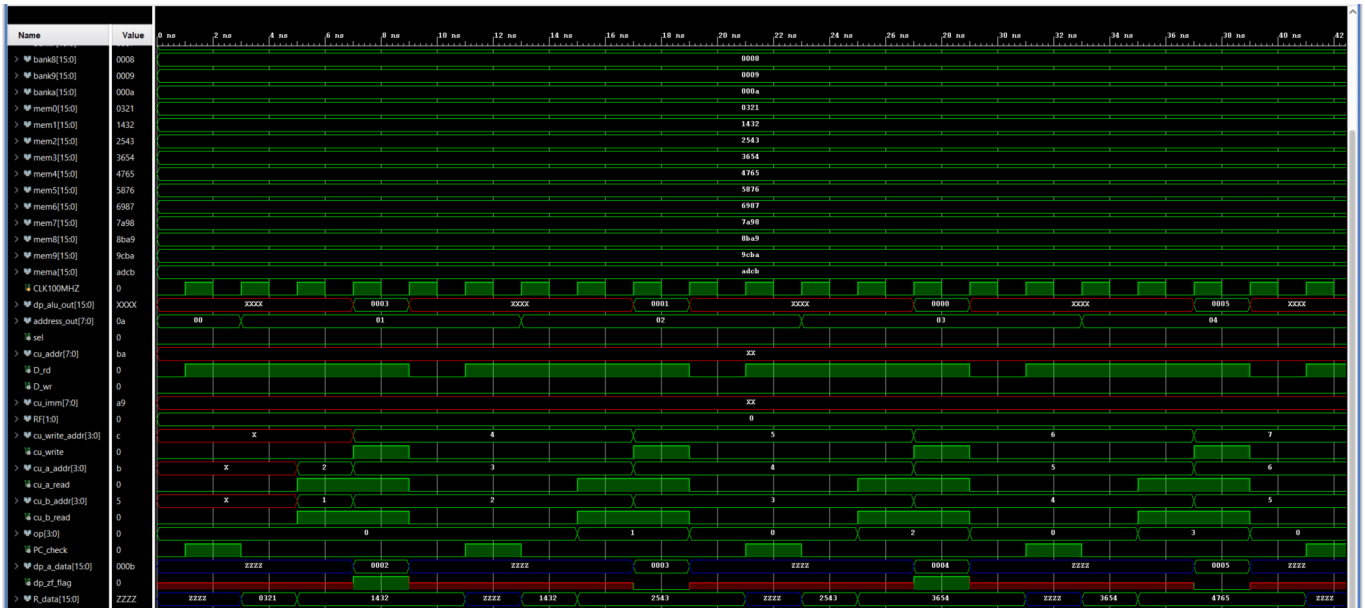


Figure 34 - Entire CPU simulation with initial register and memory values

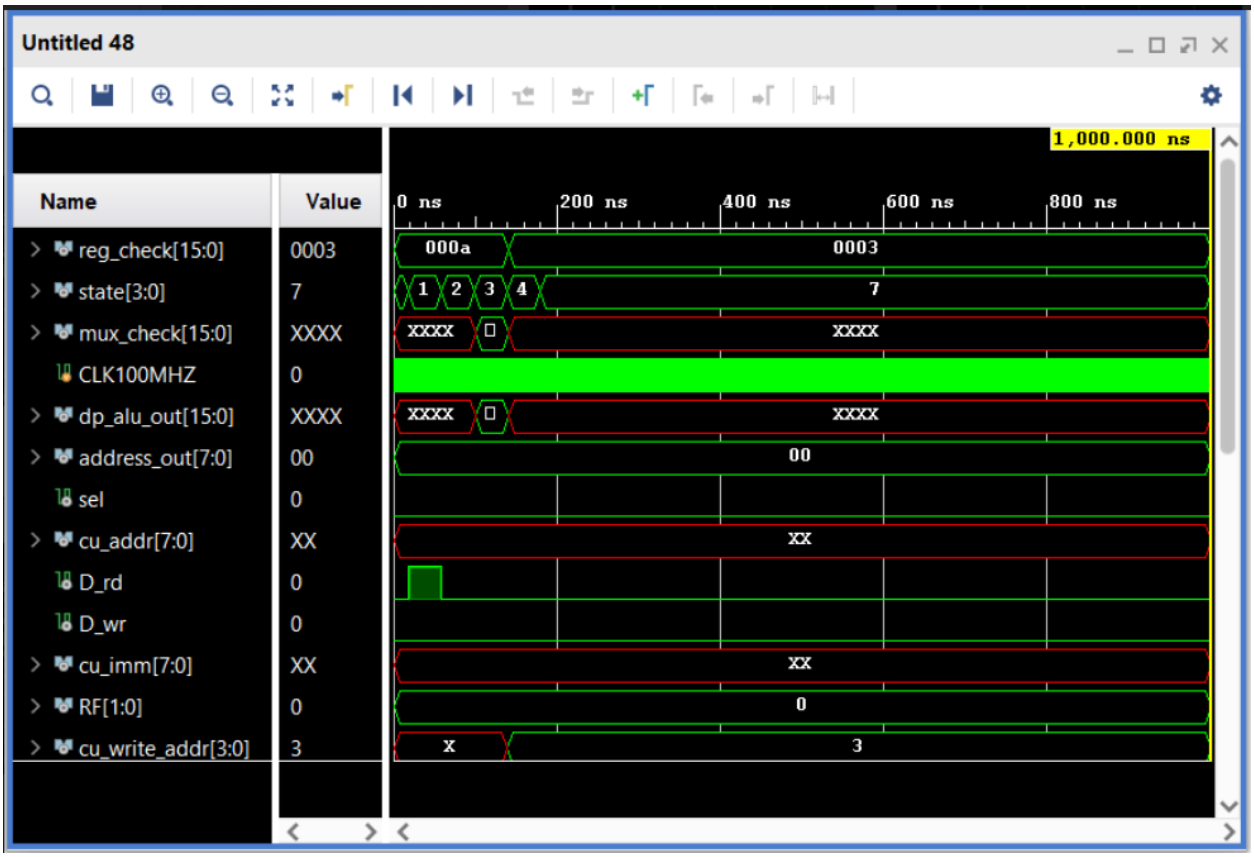


Figure 35 - Instruction 0

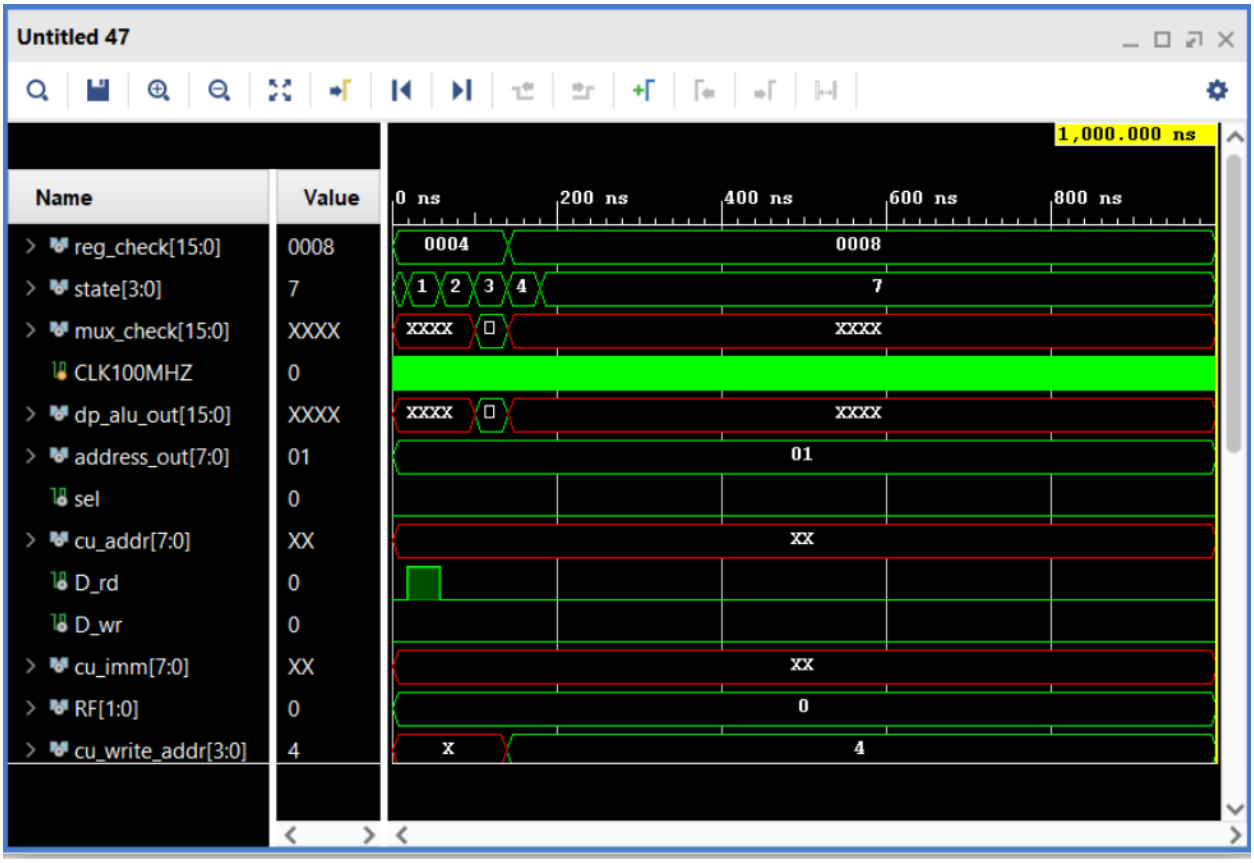


Figure 36: Instruction 1

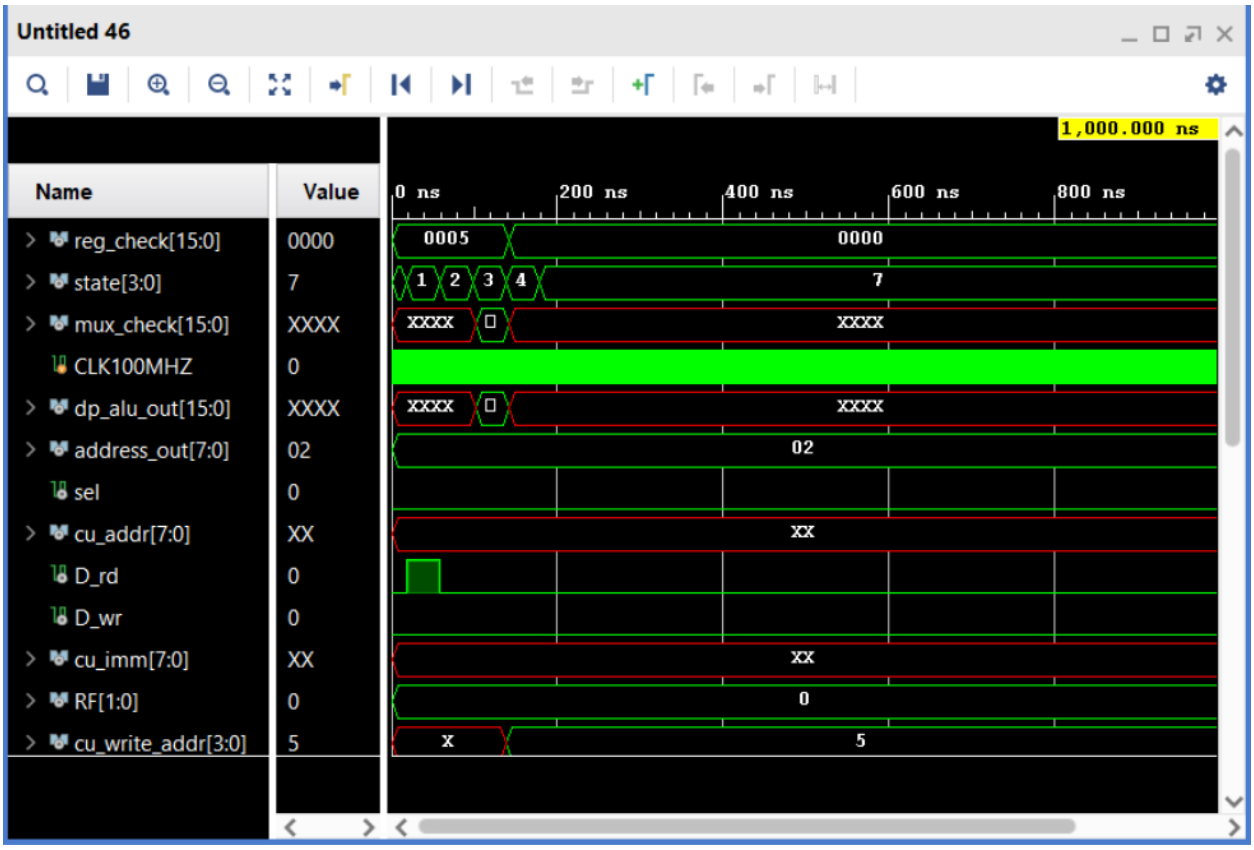


Figure 37: Instruction 2

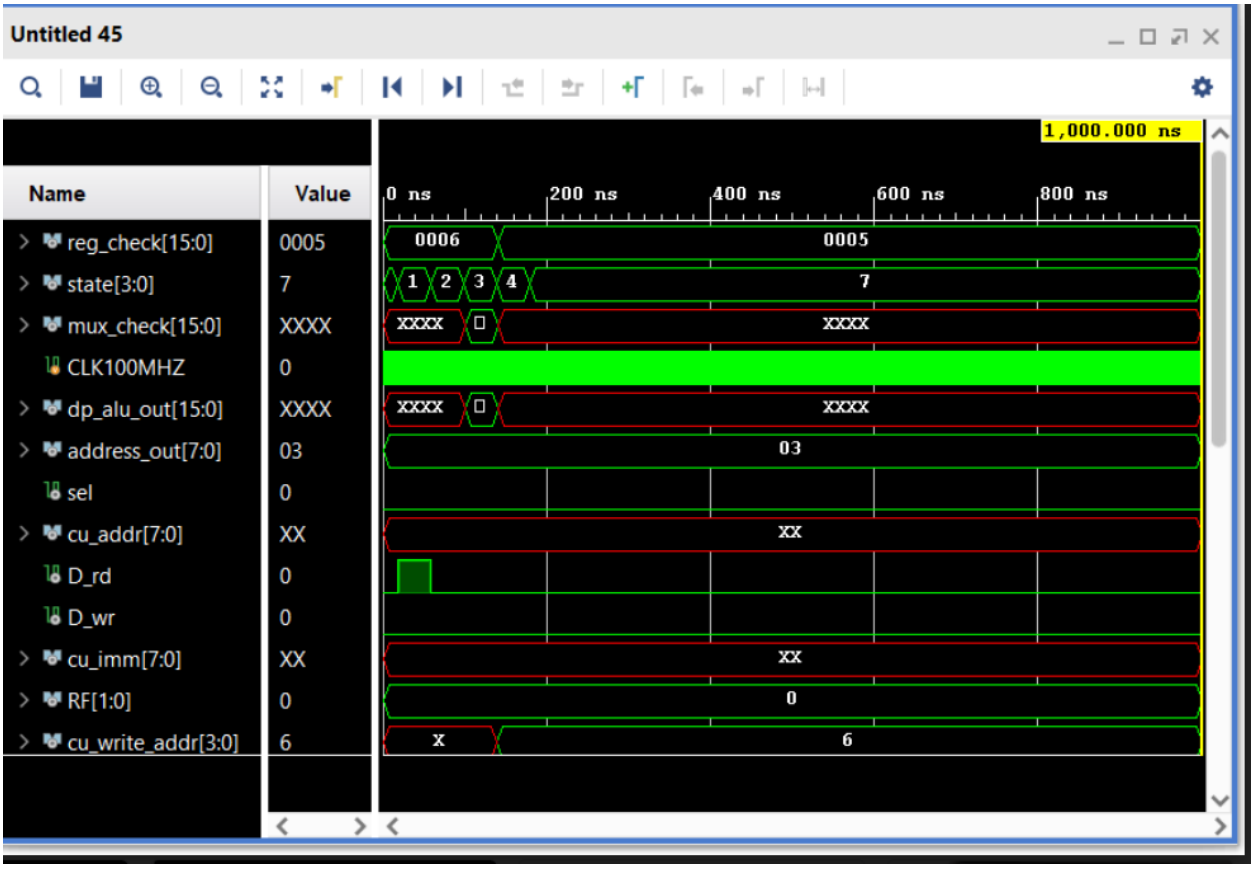


Figure 38: Instruction 3

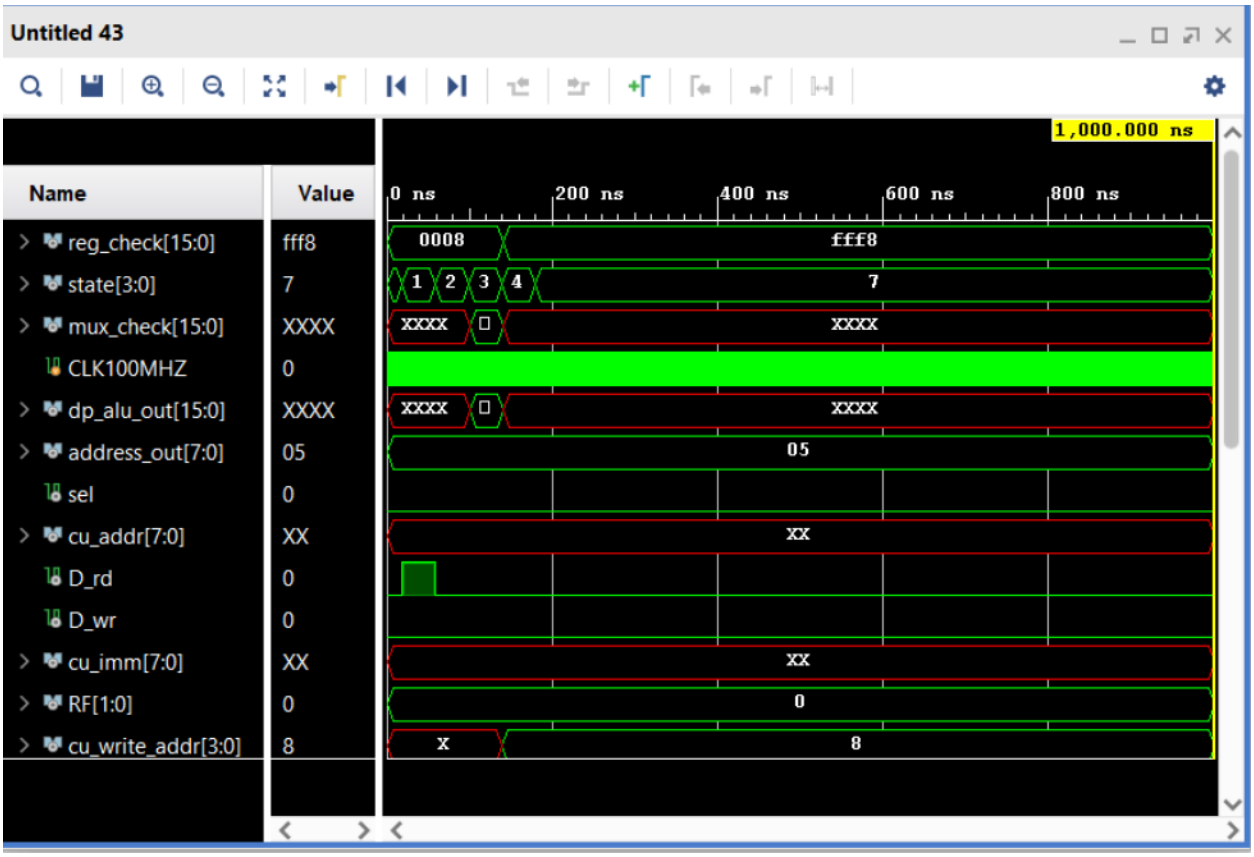


Figure 39: Instruction 4

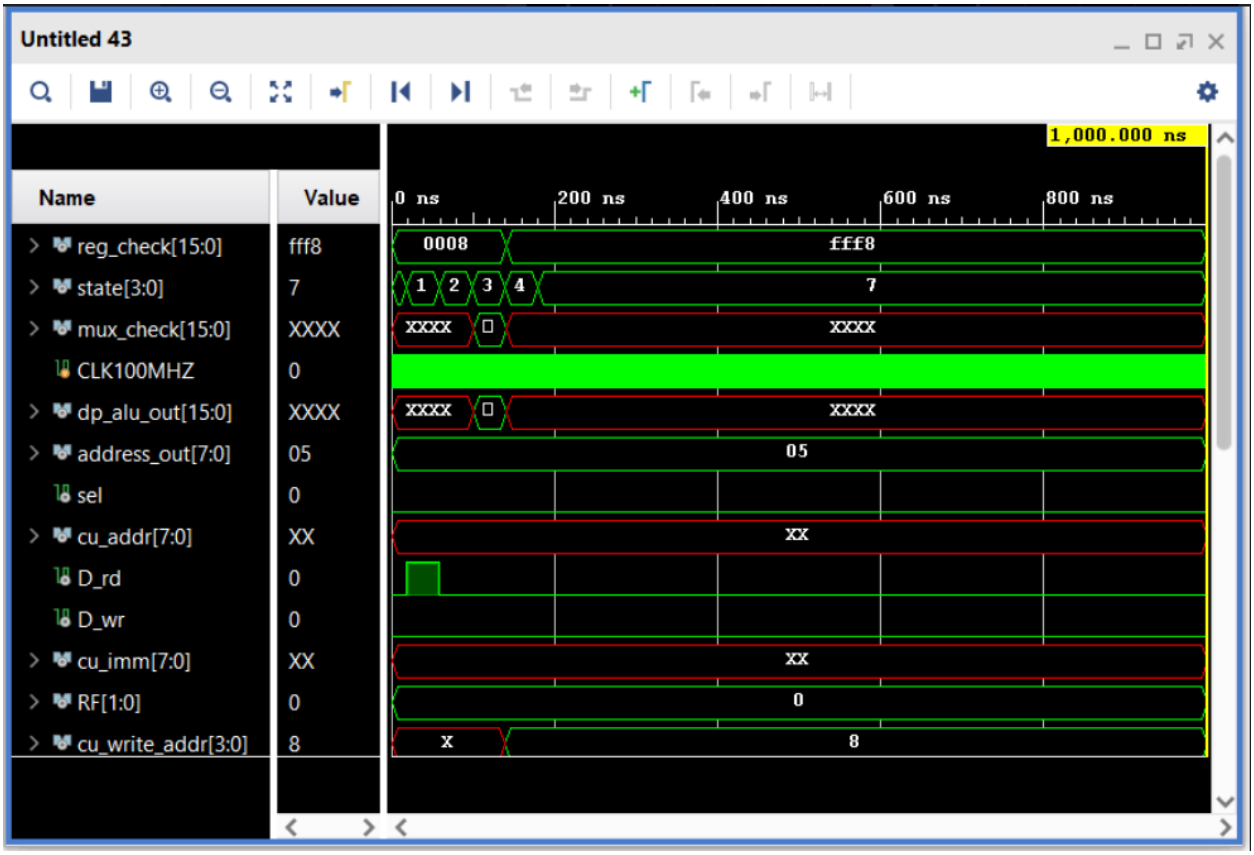


Figure 40: Instruction 5

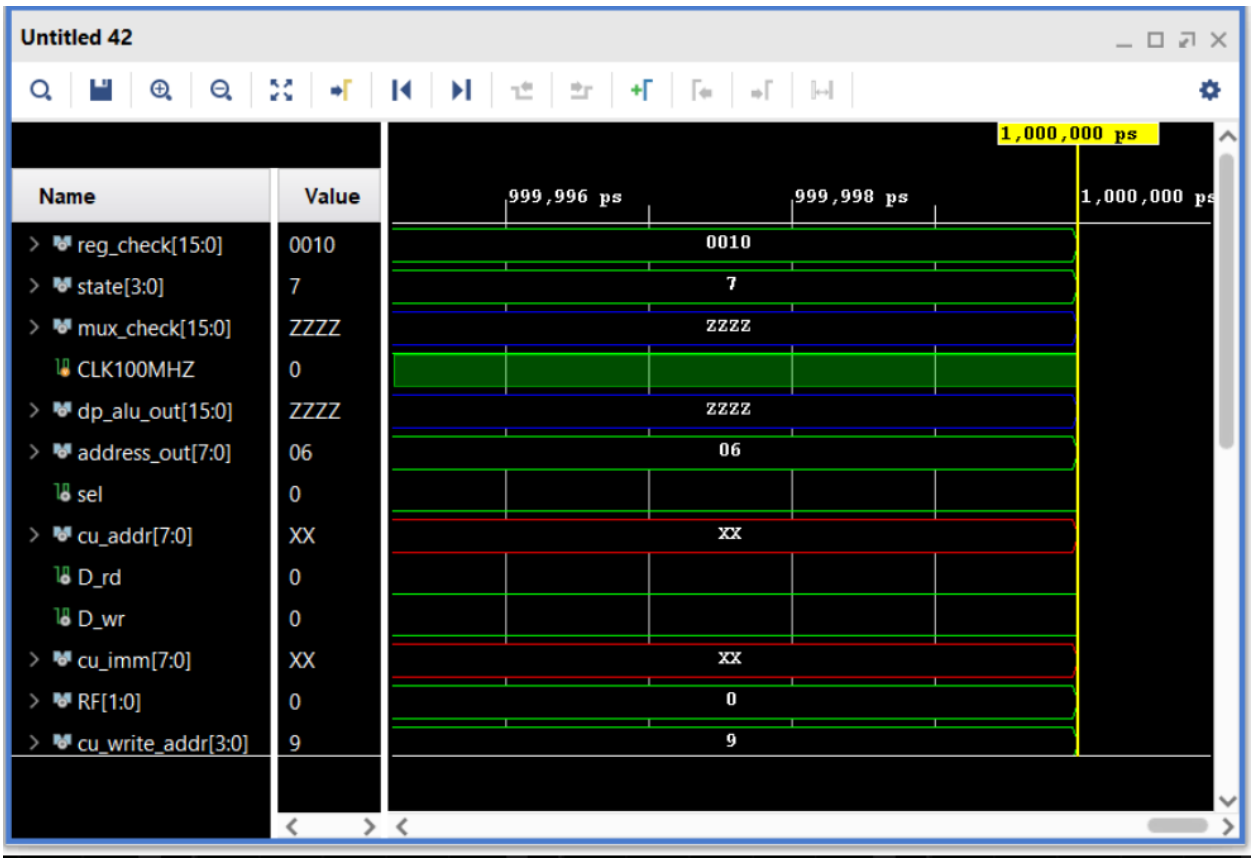


Figure 41: Instruction 6

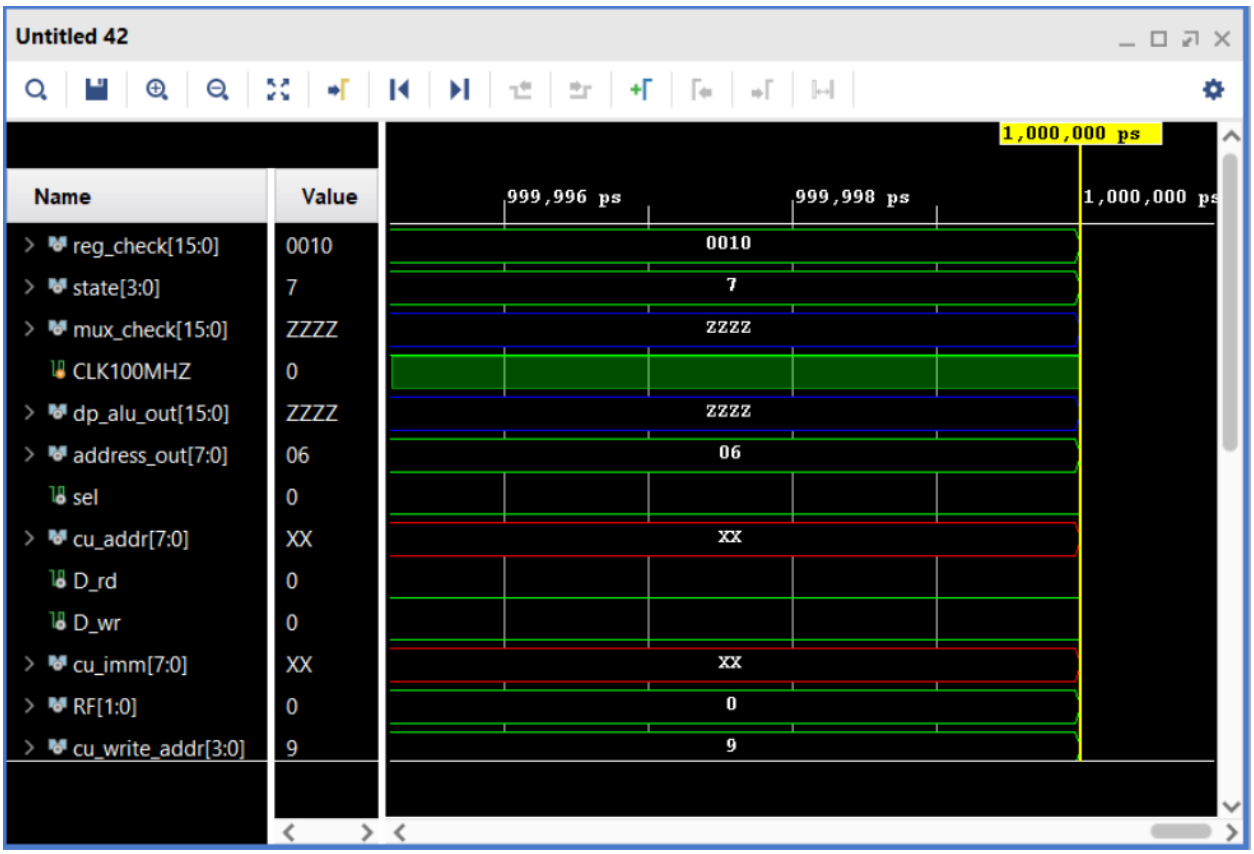


Figure 42: Instruction 7

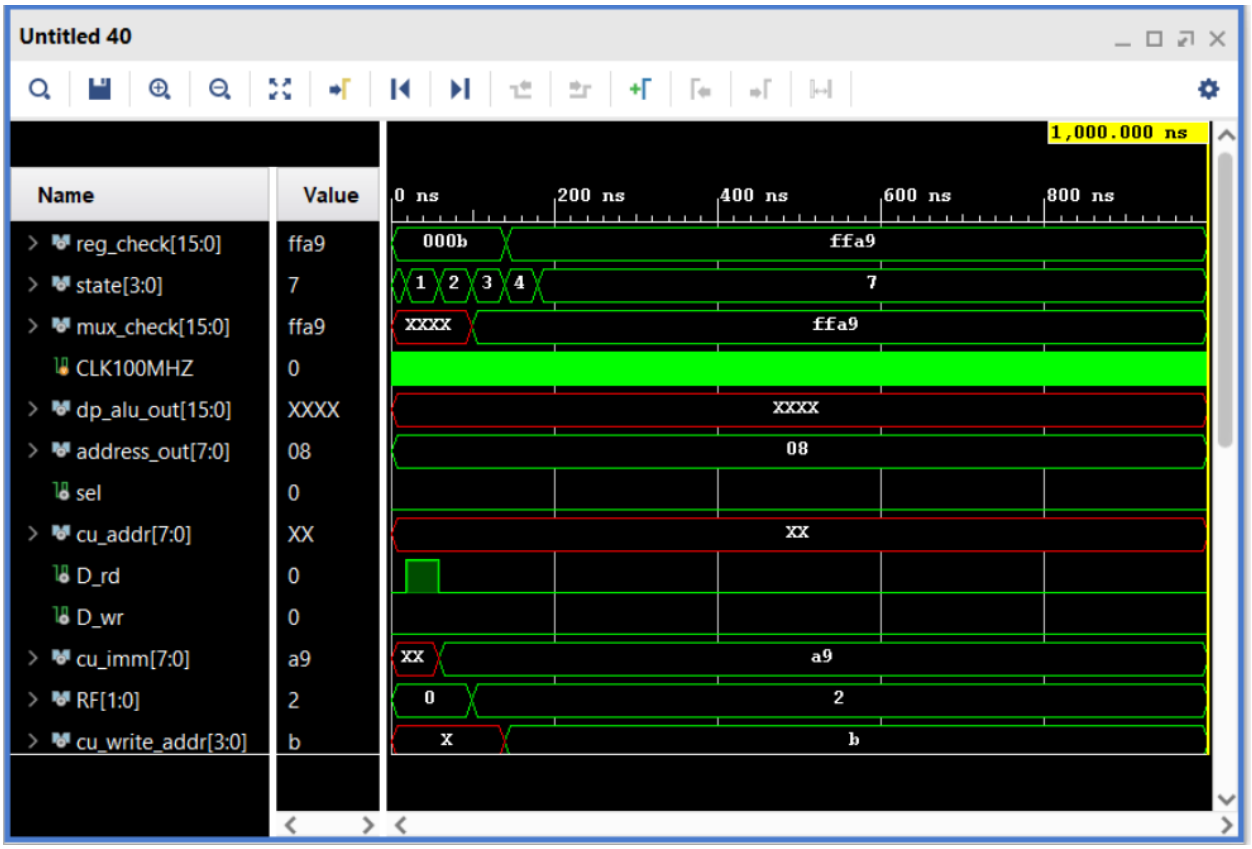


Figure 43: Instruction 8

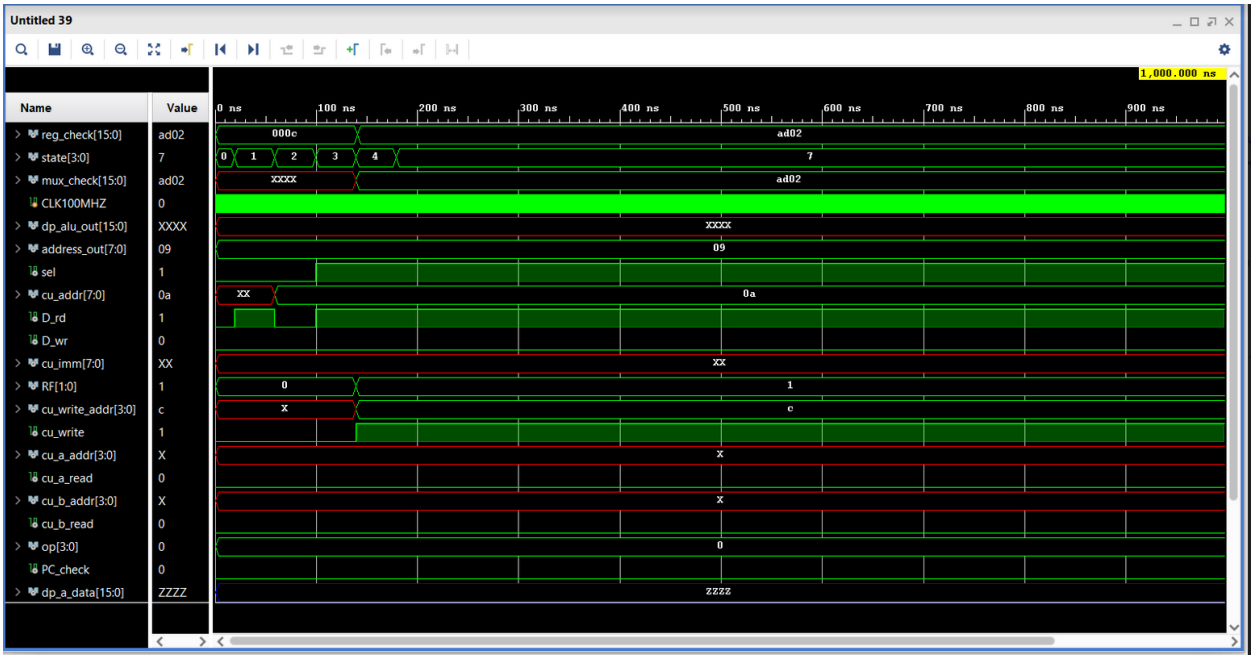


Figure 44: Instruction 9

MODULE SIMULATION SCREENSHOTS

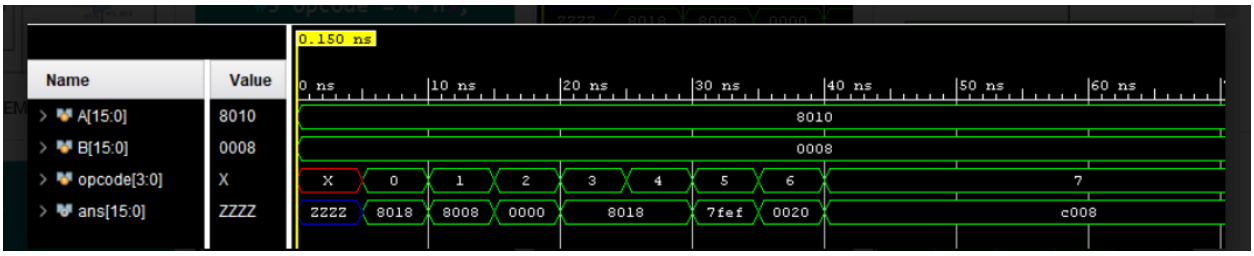


Figure 45: ALU Simulation

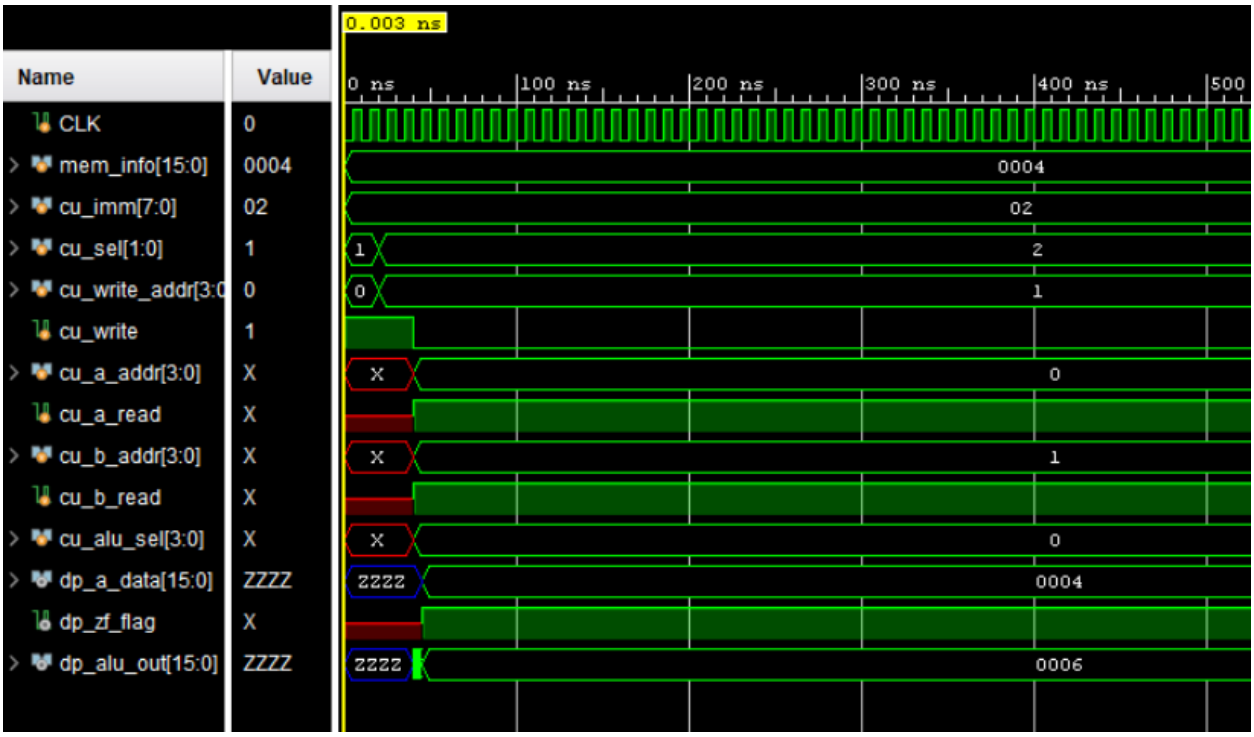


Figure 46: Datapath Simulation

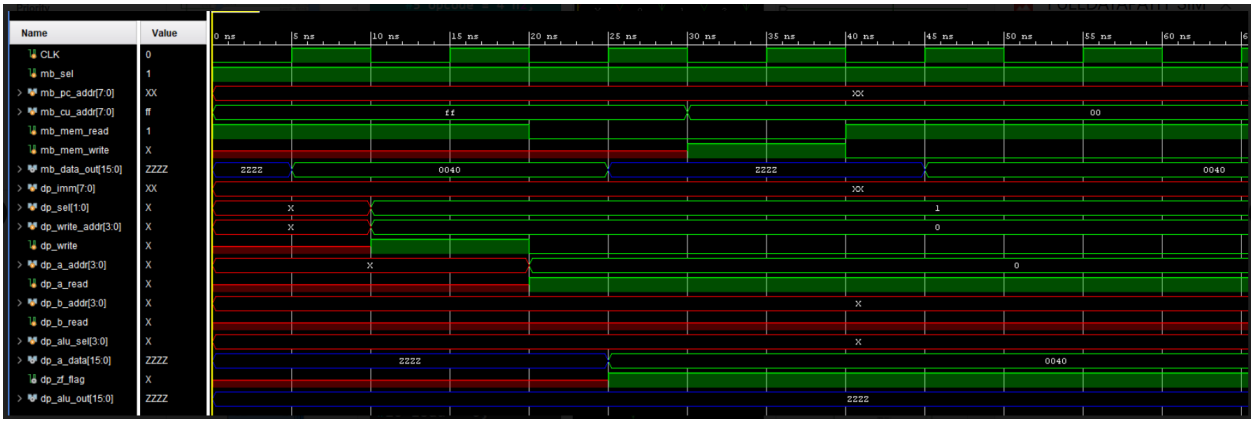


Figure 47: Memory and Datapath Simulation

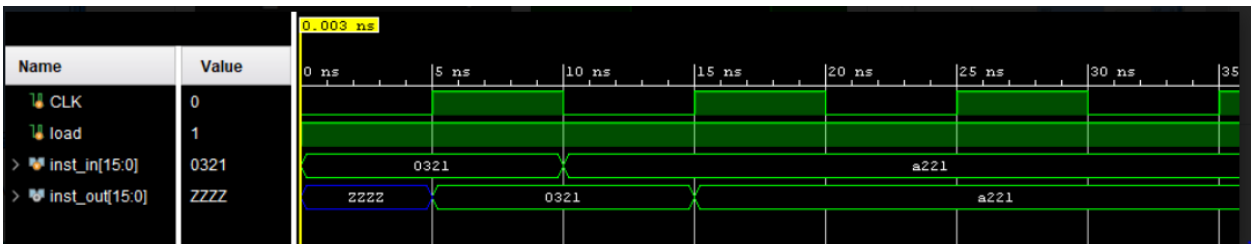


Figure 48: Instruction Register Simulation

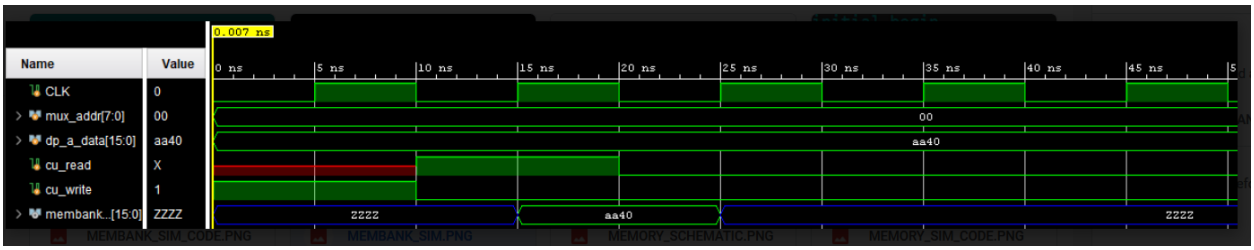


Figure 49: Memory bank Simulation

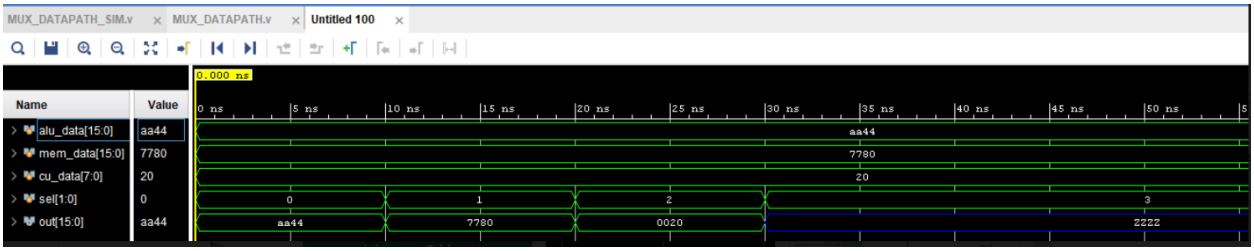


Figure 50: Memory Mux Simulation

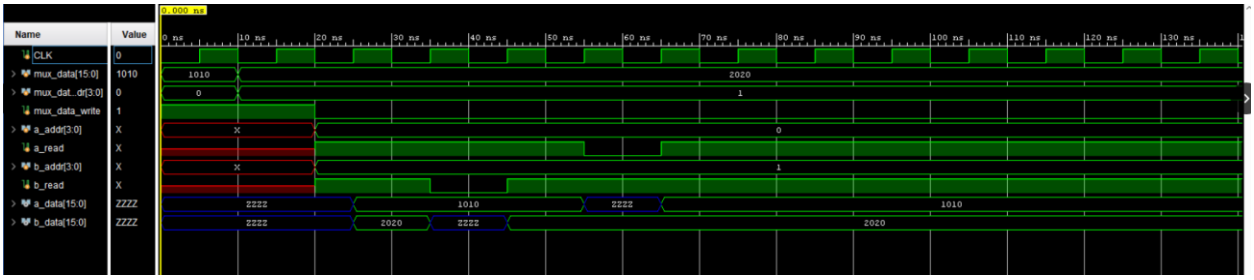


Figure 51: Reg bank Simulation

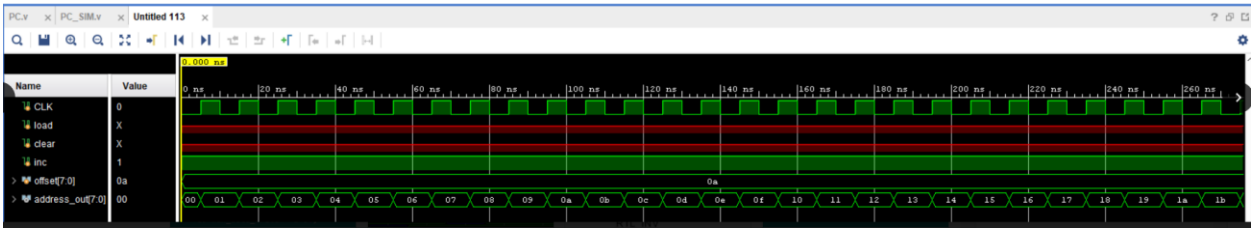


Figure 52: PC Bank Simulation

CONCLUSION

As far as complexity goes for designing and implemented the Control, Datapath, and Memory Units, the most difficult surely the Control Unit; the Memory Unit was the simplest and the Datapath Unit followed suit. First, all of the submodules except the Controller in the design were built and tested individually. After this, the design process for the Memory and Datapath Units were as follows: build and test individual submodules within a Unit, test the submodules in unison, and lastly develop and test the wrapper for the submodules as the Unit.

Once both the Datapath and Memory Units were built and tested individually, they were combined and tested in unison. After verifying the correct operation of both units, the Controller was designed with an ASM chart shown on page 22. Next, the Controller HDL was developed and simulated connected to the other two submodules in the Control Unit: the PC and IR.

With the Control Unit in working operation, the next step was to test it with the Memory Unit. Following the Memory Unit test came the verification of correct operation with the Datapath Unit. At last, the Processor was built as the top module containing the Control, Memory, and Datapath Units.

Above all, designing the 16-bit RISC Processor for the capstone to an introductory FPGA and HDL class was challenging. This project allowed us to better learn ASM's, Control vs. Datapath units, Verilog efficiency, Simulation, Synthesis, and overall Digital Design. It is not the same to learn about Computer Architecture in a pen and paper manner compared to an actual implementation, and the 16-bit RISC Processor implementation in an FPGA was a great way to learn strategies for a more efficient design. For future modifications, this project can include peripheral modules to test the processor's efficiency—for example, receiving data through UART and multiplying it with the processor.

SOURCE CODE

FULL DATAPATH

```
`timescale 1ns / 1ps
module FULL_DATAPATH(
        input CLK100MHZ,

        //FOR MEMORY
        input mb_sel,

        input [7:0] mb_pc_addr,
        input [7:0] mb_cu_addr,

        input mb_mem_read,
        input mb_mem_write,

        //FOR DATAPATH
        input [7:0] dp_imm,
        input [1:0] dp_sel,

        input [3:0] dp_write_addr,
        input    dp_write,

        input [3:0] dp_a_addr,
        input    dp_a_read,

        input [3:0] dp_b_addr,
        input    dp_b_read,

        input [3:0] dp_alu_sel,

        output [15:0] dp_a_data,
        output          dp_zf_flag,
        output [15:0] dp_alu_out,
```

```
        output [15:0] mb_data_out
);



//FOR MEMORY

//wire [15:0] mb_data_out;

//FOR DATAPATH

MEMORY MEM(
.CLK100MHZ  (CLK100MHZ),

.cu_mux_sel (mb_sel),
.pc_addr        (mb_pc_addr),
.cu_addr        (mb_cu_addr),

.mb_data_in (dp_a_data),

.cu_read        (mb_mem_read),
.cu_write   (mb_mem_write),

.mb_data_out(mb_data_out)
);



DATAPATH DP(
.CLK100MHZ    (CLK100MHZ),

.mem_info       (mb_data_out),

.cu_imm         (dp_imm),
```

```
.cu_sel          (dp_sel),

.cu_write_addr  (dp_write_addr),
.cu_write        (dp_write),

.cu_a_addr      (dp_a_addr),
.cu_a_read      (dp_a_read),

.cu_b_addr      (dp_b_addr),
.cu_b_read      (dp_b_read),

.cu_alu_sel     (dp_alu_sel),

.dp_a_data      (dp_a_data),
.dp_zf_flag     (dp_zf_flag),
.dp_alu_out     (dp_alu_out)
);

endmodule

MUX MEMORY
`timescale 1ns / 1ps
module MUX_MEMORY(
        input sel,
        input [7:0] pc_addr,
        input [7:0] cu_addr,

        output reg [7:0] out
);

//initial out = 8'hzz;
```

```
//Asterisk for combinational logic
always@(*) begin
        case(sel)
        1'b0: out       = pc_addr;
        1'b1: out       = cu_addr;
        default: out   = 8'hzz;   //disconnect if no select
        endcase
end

endmodule

MEMBANK
`timescale 1ns / 1ps
module MEMBANK(
        input CLK,

        input [7:0]     address,
        input [15:0]    data_in,
        input           read,
        input           write,

        output reg [15:0]   data_out
        //output reg status
);

initial data_out = 16'hzzzz;      //disconnect data_out initially

reg [15:0] memory [255:0];      //256x16bit memory

//Testing

initial memory[8'hFF] = 16'h0040;
initial memory[8'hFE] = 16'h0020;
```

```
initial begin



end
always@(posedge CLK) begin

        casez( {write, read} )
        2'b11: data_out = 16'hzzzz;
        2'b1?:  begin
                memory[address] = data_in;
                data_out = 16'hzzzz;
        end
        2'b?1: data_out = memory[address];
        default: data_out = 16'hzzzz;

        endcase

end




endmodule

PC
`timescale 1ns / 1ps
module PC(
input CLK,

input load,
input clear,
input inc,
```

```
input [7:0] offset,

output [7:0] address_out
);
reg [7:0] count;

assign address_out = count;

initial count = 8'h00;

always@(posedge CLK) begin

        casez({inc, load, clear})
        //If more than one signal is high, then keep the same count
        3'b111: count = count;
        3'b11?: count = count;
        3'b1?1: count = count;
        3'b?11: count = count;

        //Assess control signals
        3'b1??: count = count + 8'h01;
        3'b?1?: count = count + offset - 8'h01;
        3'b??1: count = 8'h00;

        //Keep count if any other cases are present (xzx, 0xz, 00x...)
        default: count = count;



        endcase
```

```
end



endmodule

`timescale 1ns / 1ps

module IR(
input CLK,

input load,
input    [15:0] inst_in,
output reg  [15:0] inst_out

);

initial inst_out = 16'hzzzz;
always@(posedge CLK) begin
        //load if clk is high
        if(load) inst_out = inst_in;
end


endmodule


`timescale 1ns / 1ps
module ZEROFLAG(
        input [15:0] num,
        output zeroflag
);
```

```
assign zeroflag = ~(num);


endmodule

`timescale 1ns / 1ps

module REGBANK(
        input CLK,

        input [15:0] mux_data,

        input [3:0] mux_data_addr,
        input    mux_data_write,

        input [3:0] a_addr,
        input    a_read,

        input [3:0] b_addr,
        input    b_read,

        output reg [15:0]  a_data,
        output reg [15:0]  b_data,

        //tell the outside world the status
        // z: BOTH a and b are read
        // 1: A is being read
        // 0: B is being read

        output reg idle
);
```

```
reg [15:0] bank [15:0];


//Disconnect a and b data values in beginning
initial begin
        a_data = 16'hzzzz;
        b_data = 16'hzzzz;
        //bank[4'hF] = 16'h9070; //FOR TESTING
end


always@(posedge CLK) begin
        casez( {mux_data_write, a_read, b_read} )
        //cannot write and read
        3'b111: begin
                a_data = 16'hzzzz;
                b_data = 16'hzzzz;
        end
        3'b11?: begin
                a_data = 16'hzzzz;
                b_data = 16'hzzzz;
        end
        3'b1?1: begin
                a_data = 16'hzzzz;
                b_data = 16'hzzzz;
        end
        3'b1??: begin   //now we can write
                bank[mux_data_addr] = mux_data;
                a_data = 16'hzzzz;
                b_data = 16'hzzzz;
        end
        3'b?11: begin   //show a and b if write is z or 0
                a_data = bank[a_addr];
```

```
                b_data = bank[b_addr];
        end
        3'b??1: begin
                a_data = 16'hzzzz;
                b_data = bank[b_addr];  //only b_read is valid
        end
        3'b?1?: begin
                a_data = bank[a_addr];  //only a_read is valid
                b_data = 16'hzzzz;
        end
        default: begin  //d/c if no control signals
                a_data = 16'hzzzz;
                b_data = 16'hzzzz;
        end

        endcase
end




endmodule

`timescale 1ns / 1ps

module MUX_DATAPATH(
        input [15:0] alu_data,
        input [15:0] mem_data,
        input [7:0]  cu_data,
        input [1:0] sel,

        output reg [15:0] out
```

```
);

//Disconnect output initially
initial out = 16'hzzzz;

//Asterisk for combinational logic
always@(*) begin
        case(sel)
        2'b00: out      = alu_data;
        2'b01: out      = mem_data;
        2'b10: out      = { {8{cu_data[7]}}, cu_data};  //sign extend to 16-bits
        default: out   = 16'hzzzz;                  //disconnect if no select
        endcase
end

endmodule

`timescale 1ns / 1ps

module ALU(
        input   [15:0] A,
        input   [15:0] B,
        input   [3:0] op,

        output reg [15:0] ans
);



/* Opcodes
  0: ADD
  1: SUB
  2: AND
  3: OR
```

```
  4: XOR
  5: NOT
  6: SLL/SLA
  7: SRA
*/

initial ans = 16'hzzzz;

always@(*) begin
        case(op)
        4'h0: ans = A+B;
        4'h1: ans = A-B;
        4'h2: ans = A&B;
        4'h3: ans = A|B;
        4'h4: ans = A^B;
        4'h5: ans = ~A;
        4'h6: ans = A << 1;
        4'h7: ans = $signed(A) >>> 1;
        default: ans = 16'hzzzz;
        endcase
end

Endmodule

//////////////////////////////////////////
`timescale 1ns / 1ps
module CONTROLLER(
input btnC,

input           SLOWCLK,

input                   CLK100MHZ,

input                   dp_zf_flag,   //RF_Rp_zero from DATAPATH - Zero Flag
input [15:0]        R_data,        //R_data from MEMORY - MEMBANK
```

```
// Control signals to MEMORY Unit
output     [7:0] address_out,  //PC_addr to MEMORY unit
output reg                     sel,          //D_addr_sel to MEMORY unit
output reg [7:0]         cu_addr,          //D_addr to MEMORY unit
output reg                     D_rd,              //D_rd to MEMORY unit
output reg                     D_wr,              //D_wr to MEMORY unit

// Control signals to DATAPATH unit
output reg [7:0]         cu_imm,                 //RF_W_data to DATAPATH   - Main Memory
output reg [1:0]         RF,                          //RF_s1-RF_s0 to DATAPATH - Main Memory

output reg [3:0]         cu_write_addr, //RF_W_addr to DATAPATH - - Register Bank
output reg                     cu_write,      //W_wr to DATAPATH - Register Bank

output reg [3:0]         cu_a_addr,       //RF_Rp_addr to DATAPATH - Register Bank
output reg                     cu_a_read,       //Rp_rd - Register Bank

output reg [3:0]    cu_b_addr,     //RF_Rq_addr to DATAPATH - Register Bank
output reg                     cu_b_read,       //Rq_rd - Register Bank

output reg [3:0]         op,                          //alu_s to DATAPATH - ALU

output              PC_check,

output [3:0] c_state
);



// Control signals for IR
reg IR_ld;
wire [15:0] inst_out;

// Control signals for PC
reg PC_ld, PC_clr, PC_inc;

assign PC_check = PC_inc; //implemented to check when the program counter increments

//////////////////////////////////////////////////////
IR IR_I(

.CLK      (CLK100MHZ),

.load      (IR_ld),    //
```

```
.inst_in  (R_data), //Instruction from MEMORY to IR

.inst_out (inst_out)  //Instruction into CONTROLLER

);

//////////////////////////////////////////////////////
PC PC_I(

.CLK                    (CLK100MHZ),

.load                   (PC_ld),

.clear                  (PC_clr),

.inc                    (PC_inc),

.offset                 (R_data[7:0]),

.address_out   (address_out)
);

//Registers for FSM states
reg [3:0] state, nextstate;

parameter START = 3'b000, Fetch = 3'b001, Decode = 3'b010, Access = 3'b011, Write_back =
3'b100, Dummy = 3'b111;
assign c_state = state;
initial state = START;

//Procedural block to update state
always @(posedge SLOWCLK) begin
        state <= nextstate;
end

//Procedural block to update nextstate
always @(state) begin
        case(state)
                START:          nextstate <= Fetch;
                Fetch:          nextstate <= Decode;
                Decode:         nextstate <= Access;
                Access:         nextstate <= Write_back;
                Write_back: nextstate <= Dummy;
```

```
                Dummy:      if(btnC)  nextstate <= START;
        endcase
end

//Procedural block to update the control signals
always @(state) begin
        case(state)

                START: begin //In this state initialize all signals

                        // Control signals to PC
                        PC_inc                  <= 1'b0;

                        // Control signals to IR
                        IR_ld           <= 1'b0;

                        // Control signals to MEMORY unit
                        sel             <= 1'b0;
                        D_rd            <= 1'b0;
                        D_wr            <= 1'b0;

                        // Control signals to DATAPATH unit
                        RF                      <= 2'b00;
                        cu_write        <= 1'b0;

                        cu_a_read       <= 1'b0;
                        cu_b_read       <= 1'b0;

                        op                      <= 4'h0;
                end

                Fetch: begin
                        // Control signals to MEMORY unit
                        sel             <= 1'b0;
                        D_rd            <= 1'b1;

                        // Control signals to IR
                        IR_ld           <= 1'b1;

                        // Control signals to PC
                        PC_inc          <= 1'b1;
                end

                Decode: begin
```

```
            D_rd        <= 1'b0;
                PC_inc                  <= 1'b0;
IR_ld       <= 1'b0;
                case(inst_out[15:12])

                        4'h0: op <= 4'h0;
                        4'h1: op <= 4'h1;
                        4'h2: op <= 4'h2;
                        4'h3: op <= 4'h3;
                        4'h4: op <= 4'h4;
                        4'h5: op <= 4'h5;
                        4'h6: op <= 4'h6;
                        4'h7: op <= 4'h7;
                        4'h8: cu_imm   <= inst_out[7:0];
                        4'h9: cu_addr  <= inst_out[7:0];
                        4'hA: cu_addr  <= inst_out[7:0];

                endcase
        end

        Access: begin
                case(inst_out[15:12])
                        4'h0: begin

                                cu_a_addr <= inst_out[7:4];
                                cu_b_addr <= inst_out[3:0];

                                cu_a_read <= 1'b1;
                                cu_b_read <= 1'b1;
                        end

                        4'h1: begin

                                cu_a_addr <= inst_out[7:4];
                                cu_b_addr <= inst_out[3:0];

                                cu_a_read <= 1'b1;
                                cu_b_read <= 1'b1;
                        end

                        4'h2:begin

                                cu_a_addr <= inst_out[7:4];
                                cu_b_addr <= inst_out[3:0];
```

```
        cu_a_read <= 1'b1;
        cu_b_read <= 1'b1;
end

4'h3: begin

        cu_a_addr <= inst_out[7:4];
        cu_b_addr <= inst_out[3:0];

        cu_a_read <= 1'b1;
        cu_b_read <= 1'b1;
end

4'h4: begin

        cu_a_addr <= inst_out[7:4];
        cu_b_addr <= inst_out[3:0];

        cu_a_read <= 1'b1;
        cu_b_read <= 1'b1;
end

4'h5: begin

        cu_a_addr <= inst_out[7:4];

        cu_a_read <= 1'b1;

end

4'h6: begin

        cu_a_addr <= inst_out[7:4];

        cu_a_read <= 1'b1;

end

4'h7: begin

        cu_a_addr <= inst_out[11:8];

        cu_a_read <= 1'b1;
```

```
                    end

                    4'h8: begin //LI

                            RF <= 2'b10;

                    end

                    4'h9: begin //LW

                            sel  <= 1'b1;
                            D_rd <= 1'b1;

                    end

                   4'hA: begin //SW

                            cu_a_addr       <= inst_out[3:0];
                            cu_a_read       <= 1'b1;
                            sel             <= 1'b1;

                    end
            endcase
    end

    Write_back: begin
            case(inst_out[15:12])

                    4'h0: begin
                      cu_a_read <= 1'b0;
                            cu_b_read <= 1'b0;
RF = 2'b00;
                            cu_write_addr <= inst_out[11:8];
                            cu_write <= 1'b1;

                    end

                    4'h1: begin
                      cu_a_read <= 1'b0;
                            cu_b_read <= 1'b0;
RF = 2'b00;
                            cu_write_addr <= inst_out[11:8];
                            cu_write <= 1'b1;
```

```
                    end

                    4'h2: begin
                      cu_a_read <= 1'b0;
                            cu_b_read <= 1'b0;
RF = 2'b00;
                            cu_write_addr <= inst_out[11:8];
                            cu_write <= 1'b1;

                    end

                    4'h3: begin
                      cu_a_read <= 1'b0;
                            cu_b_read <= 1'b0;
RF = 2'b00;
                            cu_write_addr <= inst_out[11:8];
                            cu_write <= 1'b1;

                    end

                    4'h4: begin
                      cu_a_read <= 1'b0;
                            cu_b_read <= 1'b0;
RF = 2'b00;
                            cu_write_addr <= inst_out[11:8];
                            cu_write <= 1'b1;

                    end

                    4'h5: begin
                      cu_a_read <= 1'b0;
                            cu_b_read <= 1'b0;
RF = 2'b00;
                            cu_write_addr <= inst_out[11:8];
                            cu_write <= 1'b1;

                    end

                    4'h6: begin
                      cu_a_read <= 1'b0;
                            cu_b_read <= 1'b0;
RF = 2'b00;
                            cu_write_addr <= inst_out[11:8];
```

```
                    cu_write <= 1'b1;

                end

                4'h7: begin
                  cu_a_read <= 1'b0;
                    cu_b_read <= 1'b0;
        RF = 2'b00;
                    cu_write_addr <= inst_out[11:8];
                    cu_write <= 1'b1;

                end

                4'h8: begin

                    cu_write_addr <= inst_out[11:8];
                    cu_write <= 1'b1;

                end

                4'h9: begin
                  D_wr <= 1'b0;
                    RF                      <= 2'b01;
                    cu_write_addr       <= inst_out[11:8];
                    cu_write            <= 1'b1;


                end

               4'hA: begin

                    D_wr          <= 1'b1;

                end
            endcase
      end
   endcase
end
endmodule
```