Tristan Niewenhuis Due: 10/27/2025 Ceng 242: Dr. Pyeatt

Lab 4: Load and Store

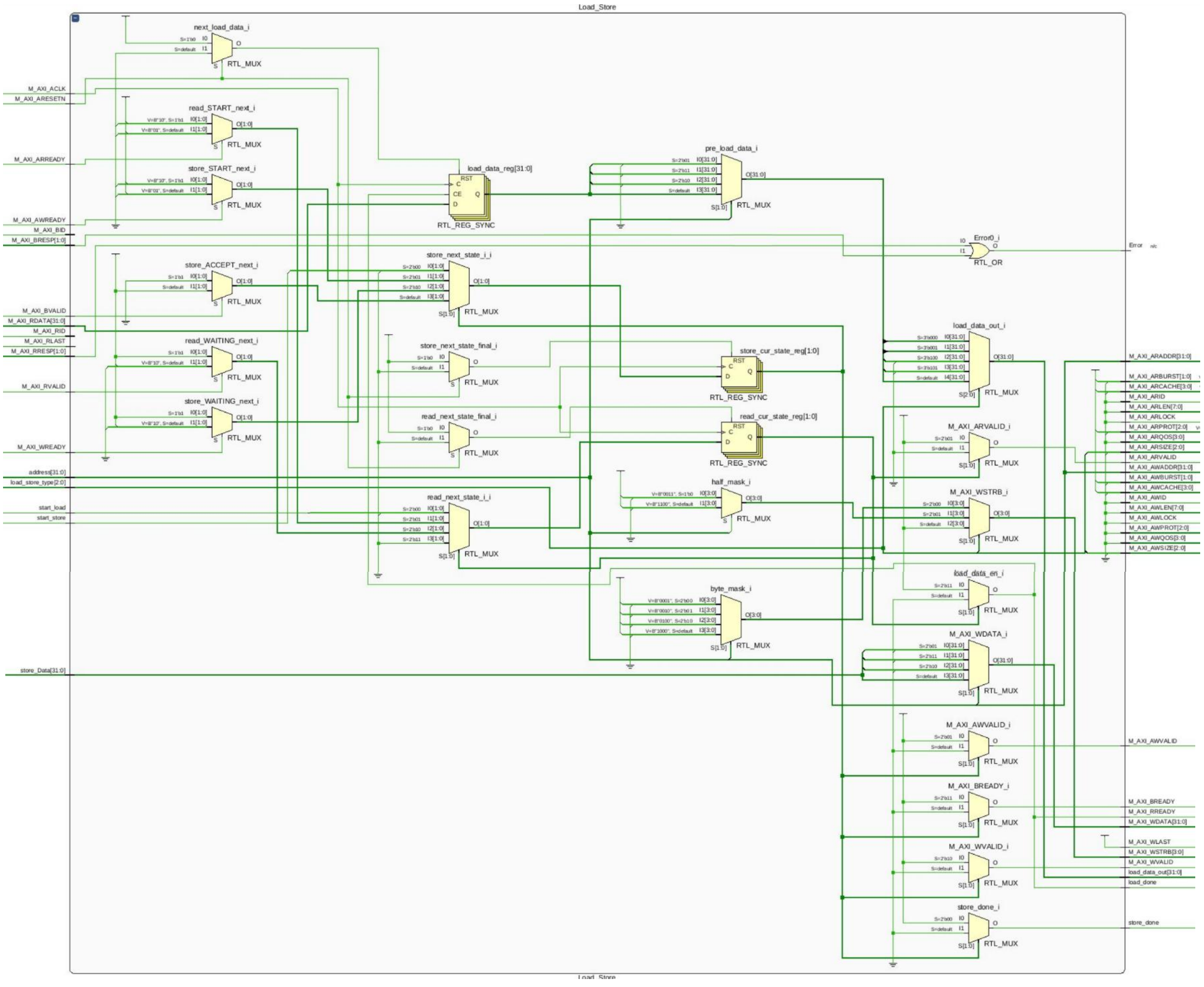
-Overview-

In this lab, we will design and integrate a load/store unit into our existing RISC-V CPU. The load/store unit will use an AXI4 bus manager interface to access memory, enabling full support for load and store instructions. We will update the datapath and sequencer so that addresses and data can be sent to and from the load/store unit. Once integrated, our CPU should be able to execute all standard RISC-V instructions except fence and system calls. We will verify our design through simulation, resulting in a fully functional core without exception handling.

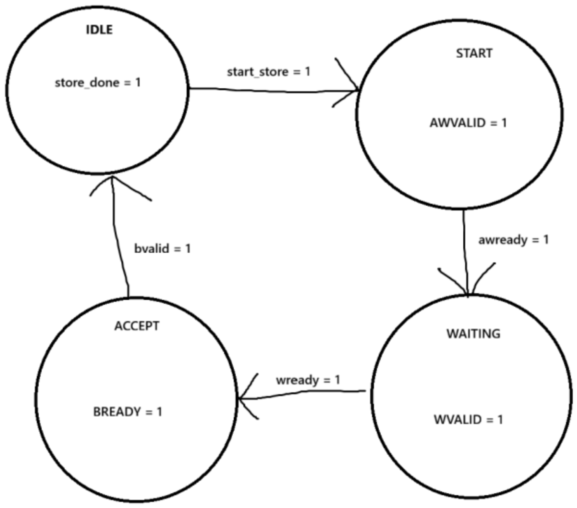
-Design-

The load part of the unit is basically just fetch unit with some additional logic for byte and half word loads. I found the memory only fetches 32 bits, so if I want a byte or half word, I have to shift the appropriate bytes into the low byte(s) (handled by “pre_load_data_i” mux). Then with the “load_data_out_i” mux, I can appropriately sign or zero extend.

For store, it has 4 states IDLE, START, WAITING, ACCEPT (state diagram below). To store the right values, I first change the data out to the memory to align the byte(s) being written, so it is aligned to its proper spot on the 32-bit bus. Then I select the appropriate strobe mask depending on if it is a byte or half word and what the last bits of the address are.

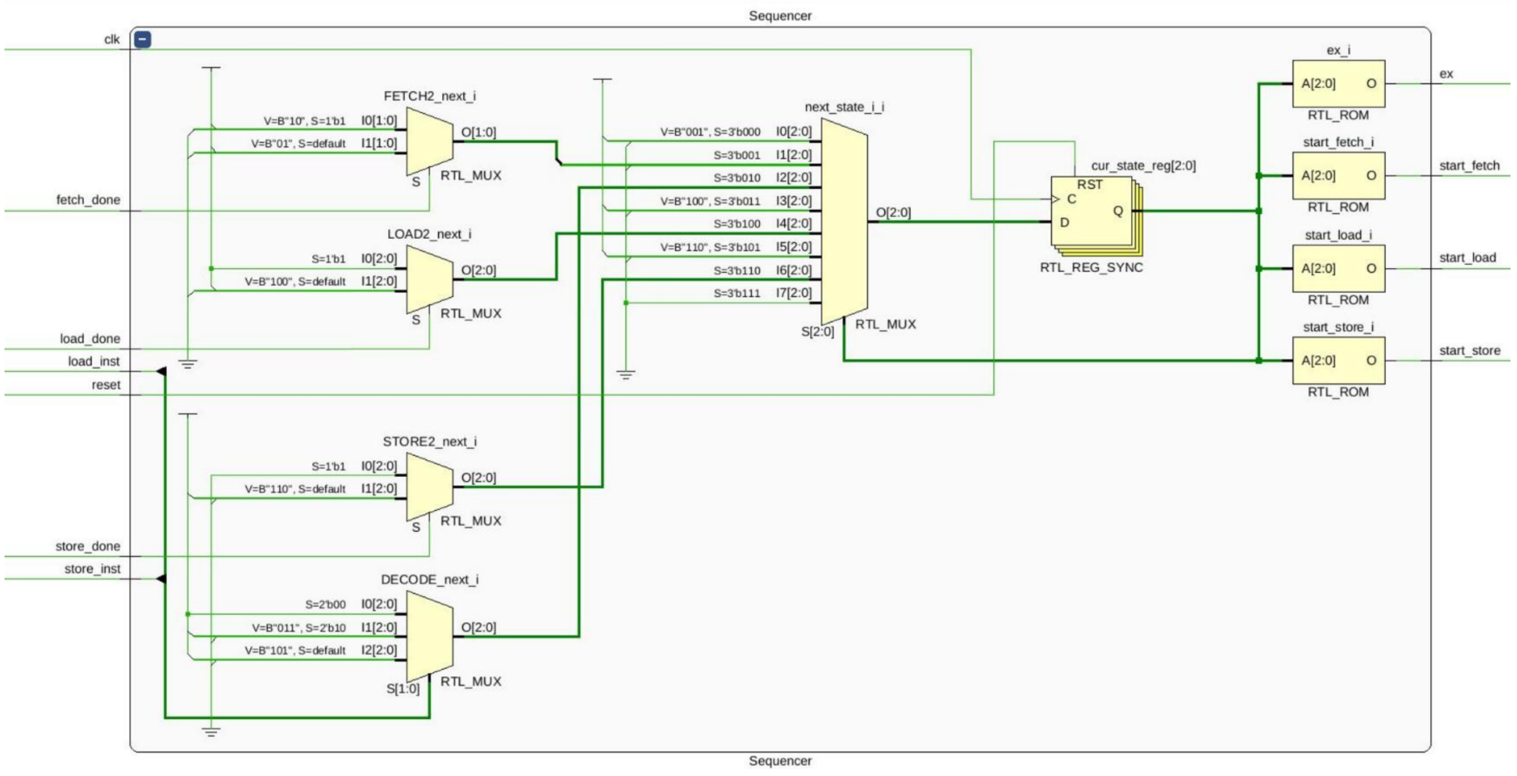


Store State Machine

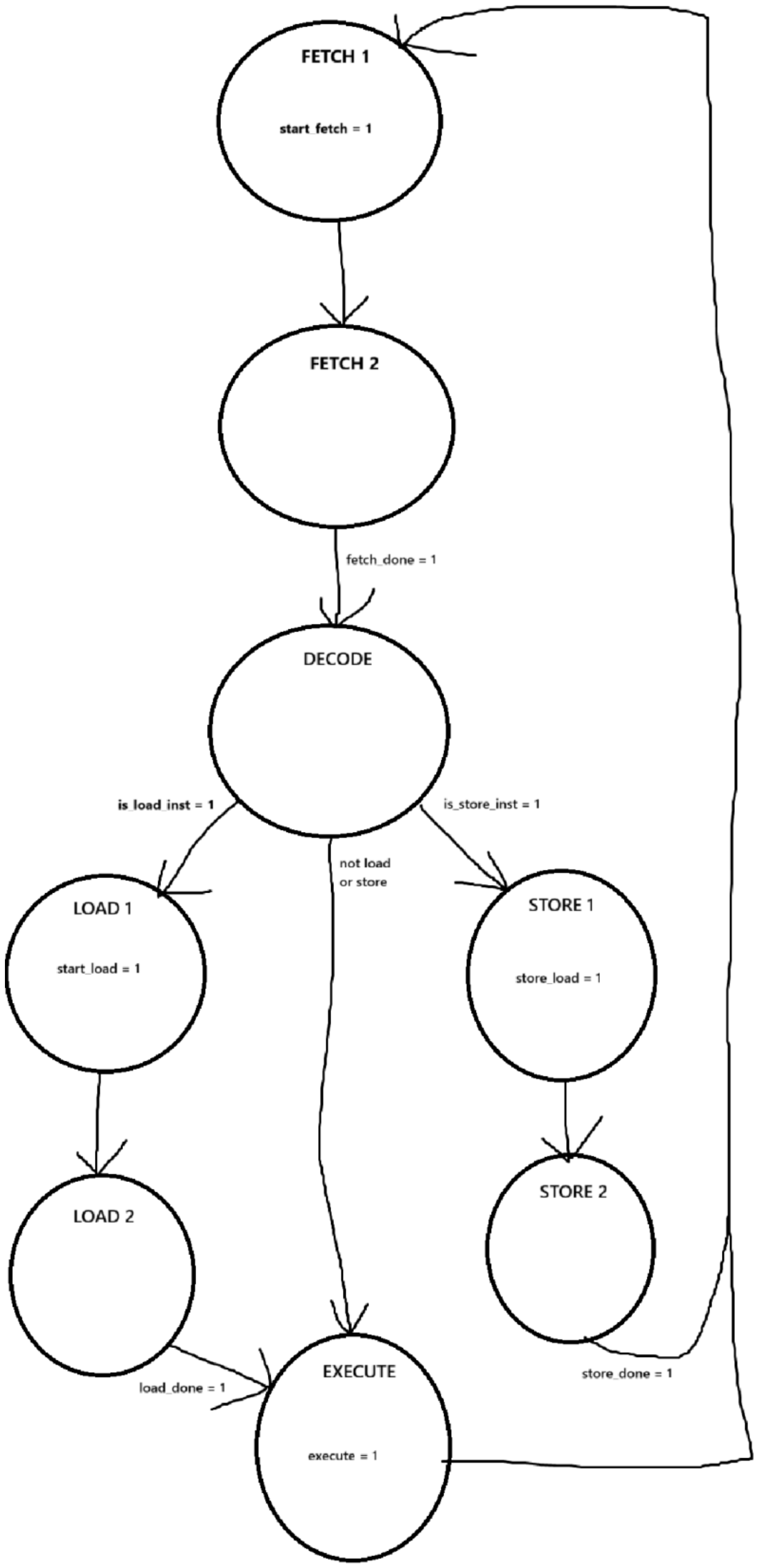


Sequencer Update

The Sequencer was updated to add 2 states for store, 2 states for load, and 1 state to decode if it is a store or load. State machine is below.

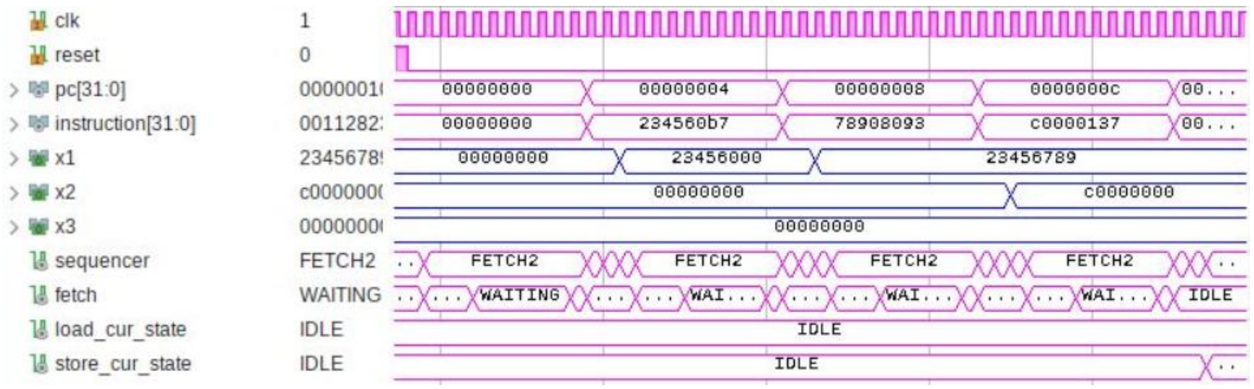


Sequencer State Machine



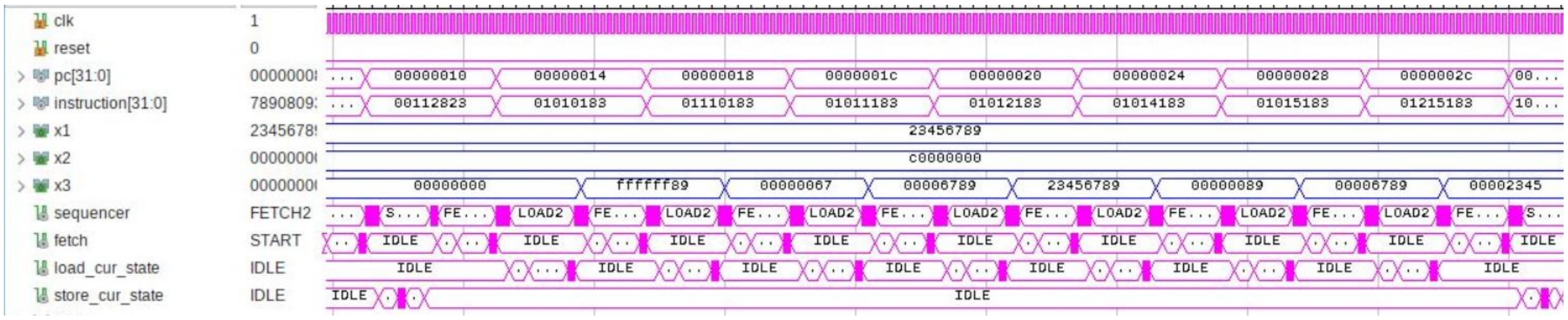
-Simulation-

This first part of the simulation is setting up the values.
x1 = 0x23456789 (value that going to be used to store/load)
x2 = 0xC0000000 (value that is offset into the RAM section of memory)
x3 is where the loads will be stored

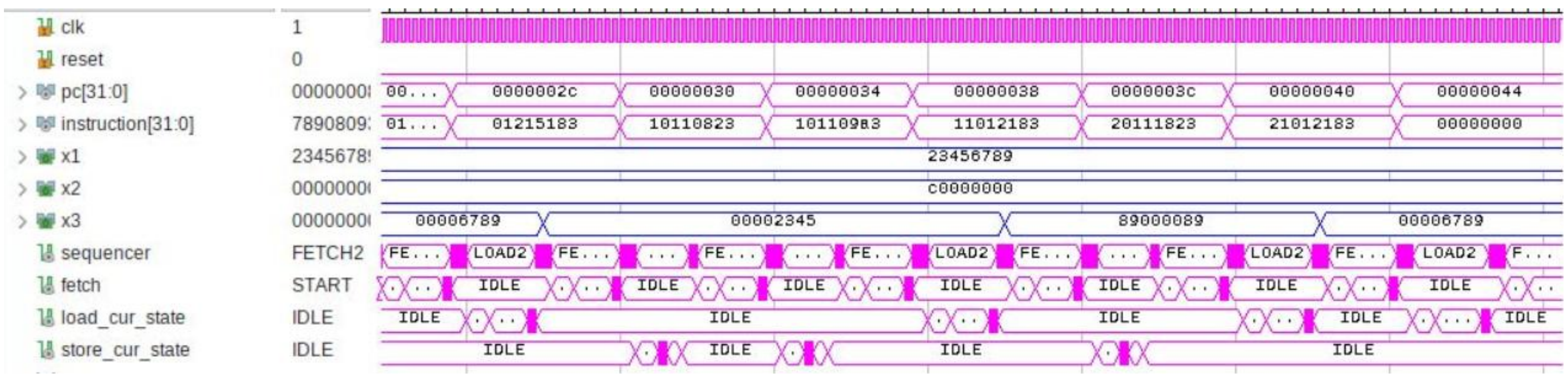

clk 1
reset 0
pc[31:0] 0000001(00000000 00000004 00000008 0000000c 00...
instruction[31:0] 0011282: 00000000 234560b7 78908093 c0000137 00...
x1 2345678! 00000000 23456000 23456789
x2 c000000(00000000 c0000000
x3 0000000(00000000
sequencer FETCH2 FETCH2 FETCH2 FETCH2 FETCH2
fetch WAITING WAITING WAI... WAI... WAI... IDLE
load_cur_state IDLE IDLE
store_cur_state IDLE IDLE

We start by storing our word (0x2345789) into RAM at address 0xC0000010. At each instruction, the following loads are performed. (I'm aligning my labels to the PC value above where the load value is put in the register even though that is the address of the next instruction it makes it easier)

0x14: Sign extended byte load at 0xC0000010. This is just the lower byte of the 32-bit bus, so it is the easy case
0x18: Sign extended byte load at 0xC0000011. Not aligned to the bottom of the 32-bit bus, so it must be shifted into the bottom byte then extended.
0x1C: Sign extended half-word load at 0xC0000010.
0x20: Sign extended word load at 0xC0000010. Simplest case
0x24: Zero extended byte load at 0xC0000010.
0x28: Zero extended half-word load at 0xC0000010.
0x2C: Zero extended half-word load at 0xC0000012. Not aligned to the 32-bit bus so also had to be shifted in, then extended.


clk 1
reset 0
pc[31:0] 0000000! 00000010 00000014 00000018 0000001c 00000020 00000024 00000028 0000002c 00...
instruction[31:0] 7890809: 00112823 01010183 01110183 01011183 01012183 01014183 01015183 01215183 10...
x1 2345678! 23456789
x2 0000000(c0000000
x3 0000000(00000000 ffffff89 00000067 00006789 23456789 00000089 00006789 00002345
sequencer FETCH2 S... FE... LOAD2 FE... LOAD2 FE... LOAD2 FE... LOAD2 FE... LOAD2 FE... LOAD2 FE... LOAD2 FE... S...
fetch START IDLE IDLE IDLE IDLE IDLE IDLE IDLE IDLE IDLE
load_cur_state IDLE IDLE IDLE IDLE IDLE IDLE IDLE IDLE IDLE
store_cur_state IDLE IDLE IDLE

To test the stores, the following tests were performed at the following instructions.
0x30: Store byte into 0xC0000110. Aligned to the 32-bit bus.
0x34: Store byte into 0xC0000113. Need to move bottom byte of register to top of AXI data bus and assign appropriate strobe byte mask.
0x38: Load word at 0xC0000110 and see 0x89 has been written to both spots of memory.
0x3C: Store half word into 0xC0000210. Align to the 32-bus.
0x40: Load word at 0xC0000210 and see it has been written properly.

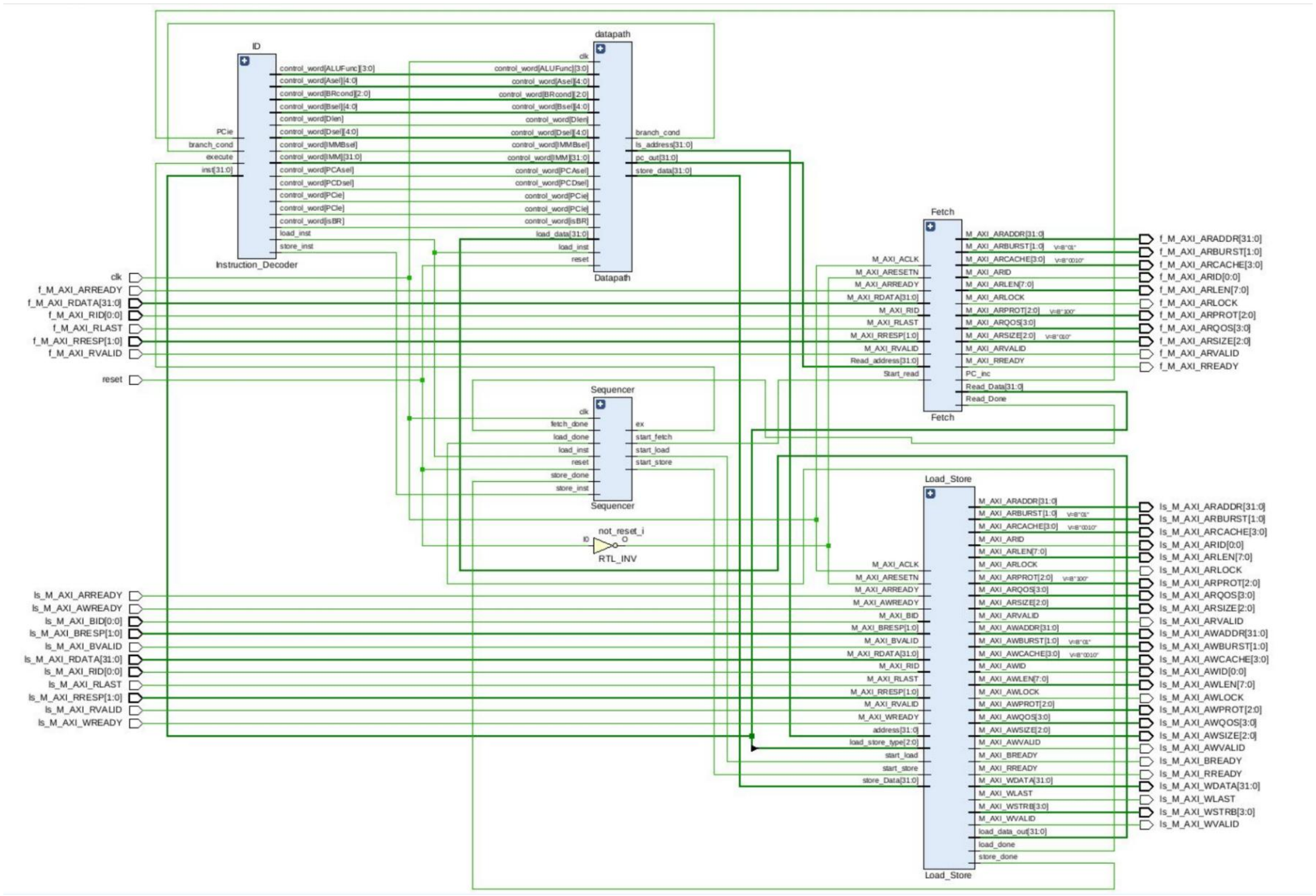

clk 1
reset 0
pc[31:0] 0000000: 00... 0000002c 00000030 00000034 00000038 0000003c 00000040 00000044
instruction[31:0] 7890809: 01... 01215183 10110823 101109a3 11012183 20111823 21012183 00000000
x1 23456789 23456789
x2 0000000 c0000000
x3 0000000 00006789 00002345 89000089 00006789
sequencer FETCH2 FE... LOAD2 FE... ... FE... ... FE... LOAD2 FE... ... FE... LOAD2 FE... LOAD2 F...
fetch START IDLE IDLE IDLE IDLE IDLE IDLE IDLE
load_cur_state IDLE IDLE IDLE IDLE IDLE IDLE
store_cur_state IDLE IDLE IDLE IDLE IDLE

-Conclusion-

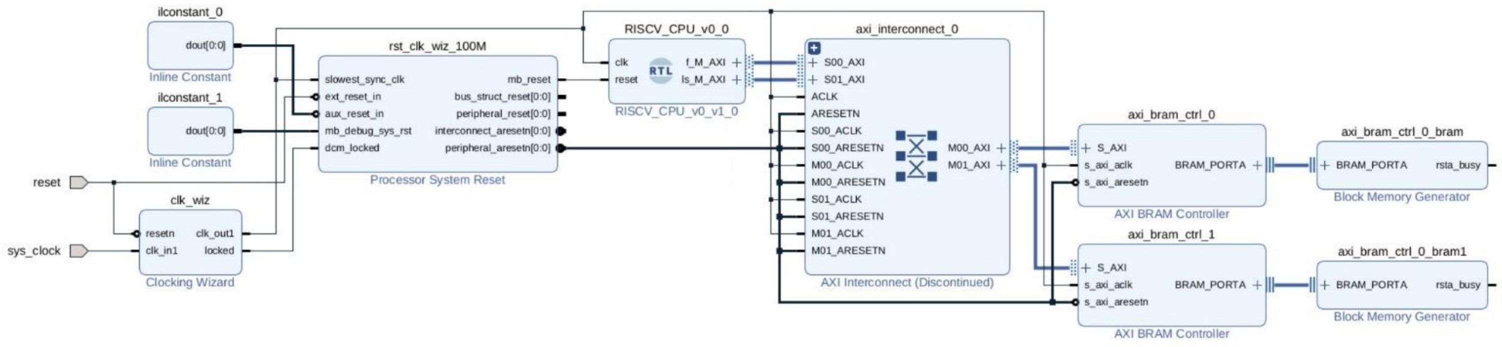
This lab proved to be harder what I thought it would be. It’s hard to debug something while you are sending signals into a black box that is the memory and hoping for a signal back. Once I dialed in the value that I needed to send the memory module, then it was smooth sailing from there. I had kind of mangle in the DECODE state into my sequencer to make sure my is_load/is_store values were getting set properly. I could probably optimize that to not waste a clock cycle on every single instruction, but that’s a good future improvement.

-Appendix-

Overview of Processor



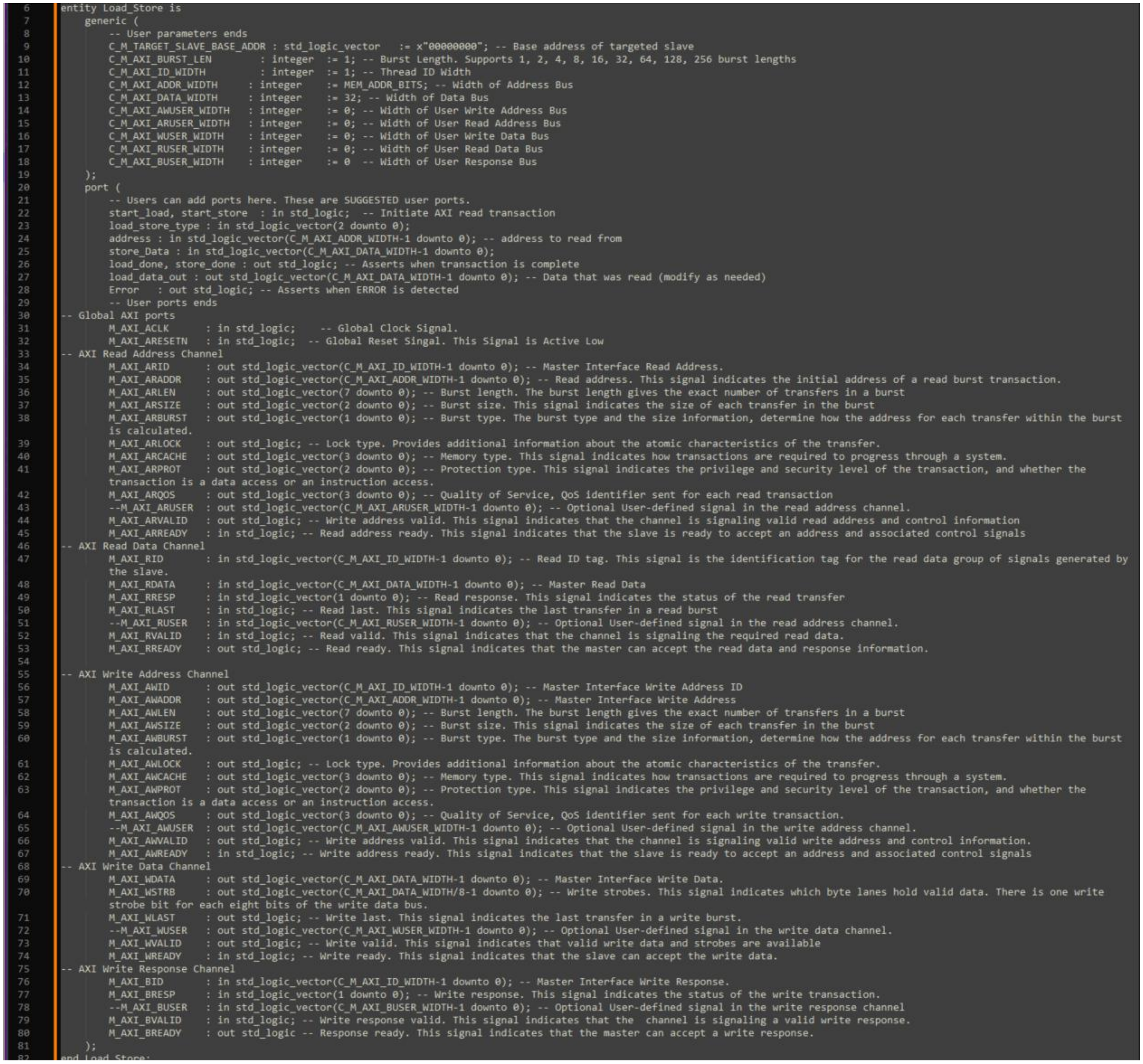
Test rig with both ROM and RAM.



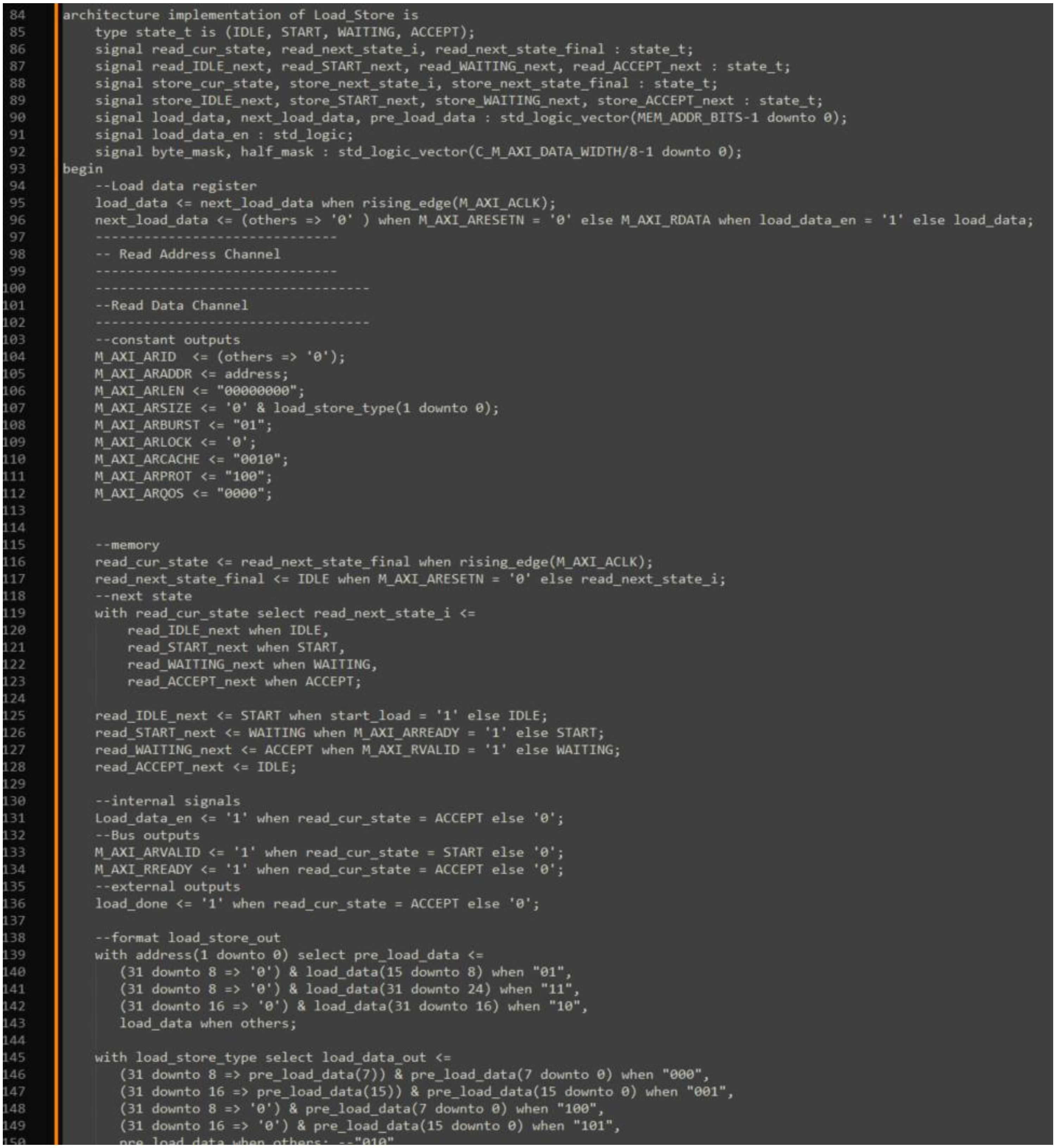
Test Code

```
234560b7 ; lui x1 #0x23456
78908093 ; addi x1 x1 #0x789
C0000137 ; lui x2 #0xC0000
00112823 ; SW x1, x2, #0x010
01010183 ; LB x3, x2, #0x010
01110183 ; LB x3, x2, #0x011
01011183 ; LH x3, x2, #0x010
01012183 ; LW x3, x2, #0x010
01014183 ; LBU x3, x2, #0x010
01015183 ; LHU x3, x2, #0x010
01215183 ; LHU x3, x2, #0x012
10110823 ; SB x1, x2, #0x110
101109a3 ; SB x1, x2, #0x113
11012183 ; LB x3, x2, #0x110
20111823 ; SH x1, x2, #0x210
21012183 ; LB x3, x2, #0x210
```

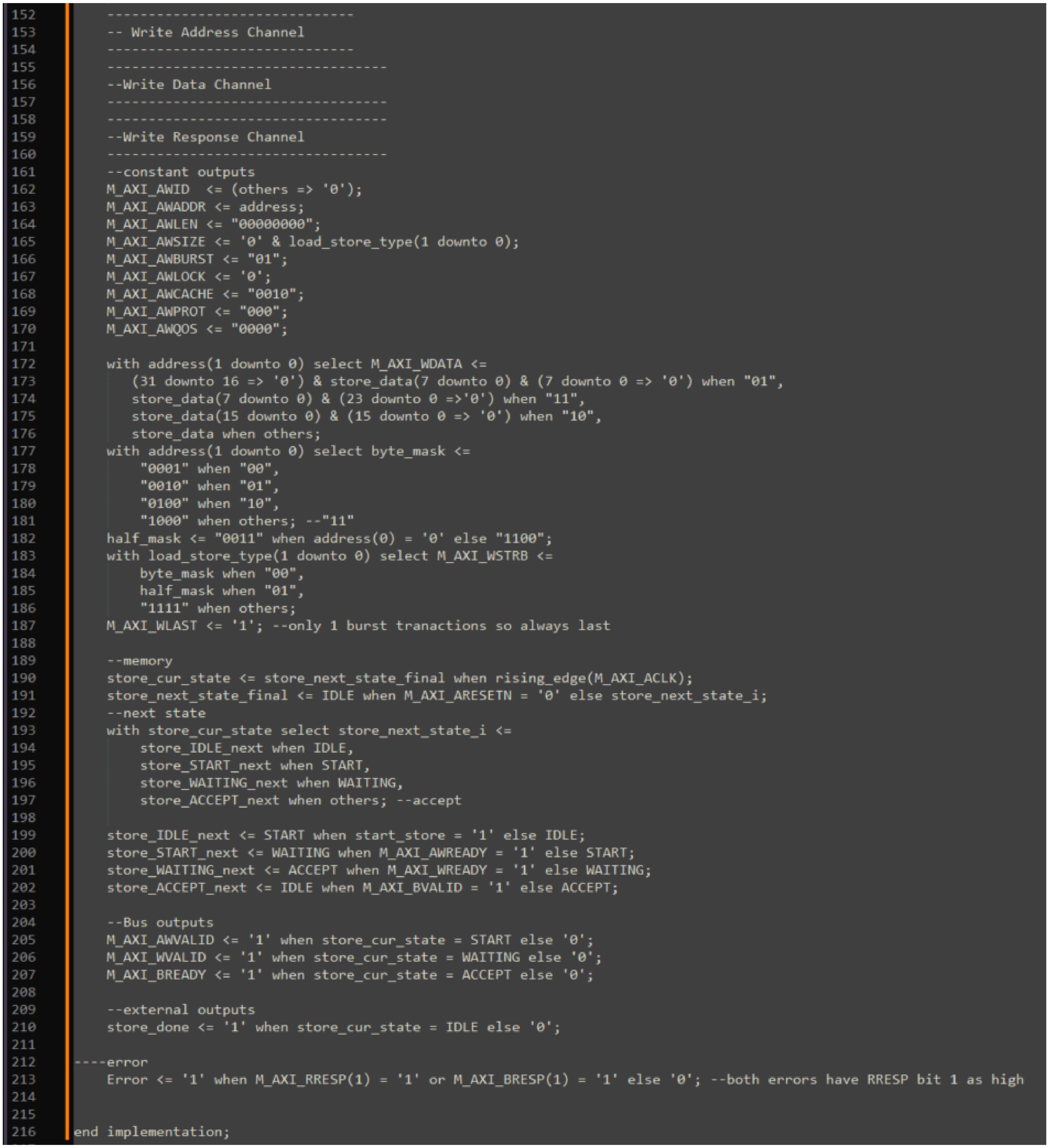
Load Store VHDL



```vhdl
entity Load_Store is
    generic (
        -- User parameters ends
        C_M_TARGET_SLAVE_BASE_ADDR : std_logic_vector   := x"00000000"; -- Base address of targeted slave
        C_M_AXI_BURST_LEN        : integer  := 1; -- Burst Length. Supports 1, 2, 4, 8, 16, 32, 64, 128, 256 burst lengths
        C_M_AXI_ID_WIDTH         : integer  := 1; -- Thread ID Width
        C_M_AXI_ADDR_WIDTH     : integer    := MEM_ADDR_BITS; -- Width of Address Bus
        C_M_AXI_DATA_WIDTH     : integer    := 32; -- Width of Data Bus
        C_M_AXI_AWUSER_WIDTH   : integer    := 0; -- Width of User Write Address Bus
        C_M_AXI_ARUSER_WIDTH   : integer    := 0; -- Width of User Read Address Bus
        C_M_AXI_WUSER_WIDTH    : integer    := 0; -- Width of User Write Data Bus
        C_M_AXI_RUSER_WIDTH    : integer    := 0; -- Width of User Read Data Bus
        C_M_AXI_BUSER_WIDTH    : integer    := 0  -- Width of User Response Bus
    );
    port (
        -- Users can add ports here. These are SUGGESTED user ports.
        start_load, start_store  : in std_logic;  -- Initiate AXI read transaction
        load_store_type : in std_logic_vector(2 downto 0);
        address : in std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0); -- address to read from
        store_Data : in std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0);
        load_done, store_done : out std_logic; -- Asserts when transaction is complete
        load_data_out : out std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0); -- Data that was read (modify as needed)
        Error   : out std_logic; -- Asserts when ERROR is detected
        -- User ports ends
-- Global AXI ports
        M_AXI_ACLK      : in std_logic;    -- Global Clock Signal.
        M_AXI_ARESETN   : in std_logic;  -- Global Reset Singal. This Signal is Active Low
-- AXI Read Address Channel
        M_AXI_ARID      : out std_logic_vector(C_M_AXI_ID_WIDTH-1 downto 0); -- Master Interface Read Address.
        M_AXI_ARADDR    : out std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0); -- Read address. This signal indicates the initial address of a read burst transaction.
        M_AXI_ARLEN     : out std_logic_vector(7 downto 0); -- Burst length. The burst length gives the exact number of transfers in a burst
        M_AXI_ARSIZE    : out std_logic_vector(2 downto 0); -- Burst size. This signal indicates the size of each transfer in the burst
        M_AXI_ARBURST   : out std_logic_vector(1 downto 0); -- Burst type. The burst type and the size information, determine how the address for each transfer within the burst is calculated.
        M_AXI_ARLOCK    : out std_logic; -- Lock type. Provides additional information about the atomic characteristics of the transfer.
        M_AXI_ARCACHE   : out std_logic_vector(3 downto 0); -- Memory type. This signal indicates how transactions are required to progress through a system.
        M_AXI_ARPROT    : out std_logic_vector(2 downto 0); -- 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.
        M_AXI_ARQOS     : out std_logic_vector(3 downto 0); -- Quality of Service, QoS identifier sent for each read transaction
        --M_AXI_ARUSER  : out std_logic_vector(C_M_AXI_ARUSER_WIDTH-1 downto 0); -- Optional User-defined signal in the read address channel.
        M_AXI_ARVALID   : out std_logic; -- Write address valid. This signal indicates that the channel is signaling valid read address and control information
        M_AXI_ARREADY   : in std_logic; -- Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals
-- AXI Read Data Channel
        M_AXI_RID       : in std_logic_vector(C_M_AXI_ID_WIDTH-1 downto 0); -- Read ID tag. This signal is the identification tag for the read data group of signals generated by the slave.
        M_AXI_RDATA     : in std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0); -- Master Read Data
        M_AXI_RRESP     : in std_logic_vector(1 downto 0); -- Read response. This signal indicates the status of the read transfer
        M_AXI_RLAST     : in std_logic; -- Read last. This signal indicates the last transfer in a read burst
        --M_AXI_RUSER   : in std_logic_vector(C_M_AXI_RUSER_WIDTH-1 downto 0); -- Optional User-defined signal in the read address channel.
        M_AXI_RVALID    : in std_logic; -- Read valid. This signal indicates that the channel is signaling the required read data.
        M_AXI_RREADY    : out std_logic; -- Read ready. This signal indicates that the master can accept the read data and response information.

-- AXI Write Address Channel
        M_AXI_AWID      : out std_logic_vector(C_M_AXI_ID_WIDTH-1 downto 0); -- Master Interface Write Address ID
        M_AXI_AWADDR    : out std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0); -- Master Interface Write Address
        M_AXI_AWLEN     : out std_logic_vector(7 downto 0); -- Burst length. The burst length gives the exact number of transfers in a burst
        M_AXI_AWSIZE    : out std_logic_vector(2 downto 0); -- Burst size. This signal indicates the size of each transfer in the burst
        M_AXI_AWBURST   : out std_logic_vector(1 downto 0); -- Burst type. The burst type and the size information, determine how the address for each transfer within the burst is calculated.
        M_AXI_AWLOCK    : out std_logic; -- Lock type. Provides additional information about the atomic characteristics of the transfer.
        M_AXI_AWCACHE   : out std_logic_vector(3 downto 0); -- Memory type. This signal indicates how transactions are required to progress through a system.
        M_AXI_AWPROT    : out std_logic_vector(2 downto 0); -- 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.
        M_AXI_AWQOS     : out std_logic_vector(3 downto 0); -- Quality of Service, QoS identifier sent for each write transaction.
        --M_AXI_AWUSER  : out std_logic_vector(C_M_AXI_AWUSER_WIDTH-1 downto 0); -- Optional User-defined signal in the write address channel.
        M_AXI_AWVALID   : out std_logic; -- Write address valid. This signal indicates that the channel is signaling valid write address and control information.
        M_AXI_AWREADY   : in std_logic; -- Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals
-- AXI Write Data Channel
        M_AXI_WDATA     : out std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0); -- Master Interface Write Data.
        M_AXI_WSTRB     : out std_logic_vector(C_M_AXI_DATA_WIDTH/8-1 downto 0); -- 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.
        M_AXI_WLAST     : out std_logic; -- Write last. This signal indicates the last transfer in a write burst.
        --M_AXI_WUSER   : out std_logic_vector(C_M_AXI_WUSER_WIDTH-1 downto 0); -- Optional User-defined signal in the write data channel.
        M_AXI_WVALID    : out std_logic; -- Write valid. This signal indicates that valid write data and strobes are available
        M_AXI_WREADY    : in std_logic; -- Write ready. This signal indicates that the slave can accept the write data.
-- AXI Write Response Channel
        M_AXI_BID       : in std_logic_vector(C_M_AXI_ID_WIDTH-1 downto 0); -- Master Interface Write Response.
        M_AXI_BRESP     : in std_logic_vector(1 downto 0); -- Write response. This signal indicates the status of the write transaction.
        --M_AXI_BUSER   : in std_logic_vector(C_M_AXI_BUSER_WIDTH-1 downto 0); -- Optional User-defined signal in the write response channel
        M_AXI_BVALID    : in std_logic; -- Write response valid. This signal indicates that the  channel is signaling a valid write response.
        M_AXI_BREADY    : out std_logic -- Response ready. This signal indicates that the master can accept a write response.
    );
end Load_Store;
```

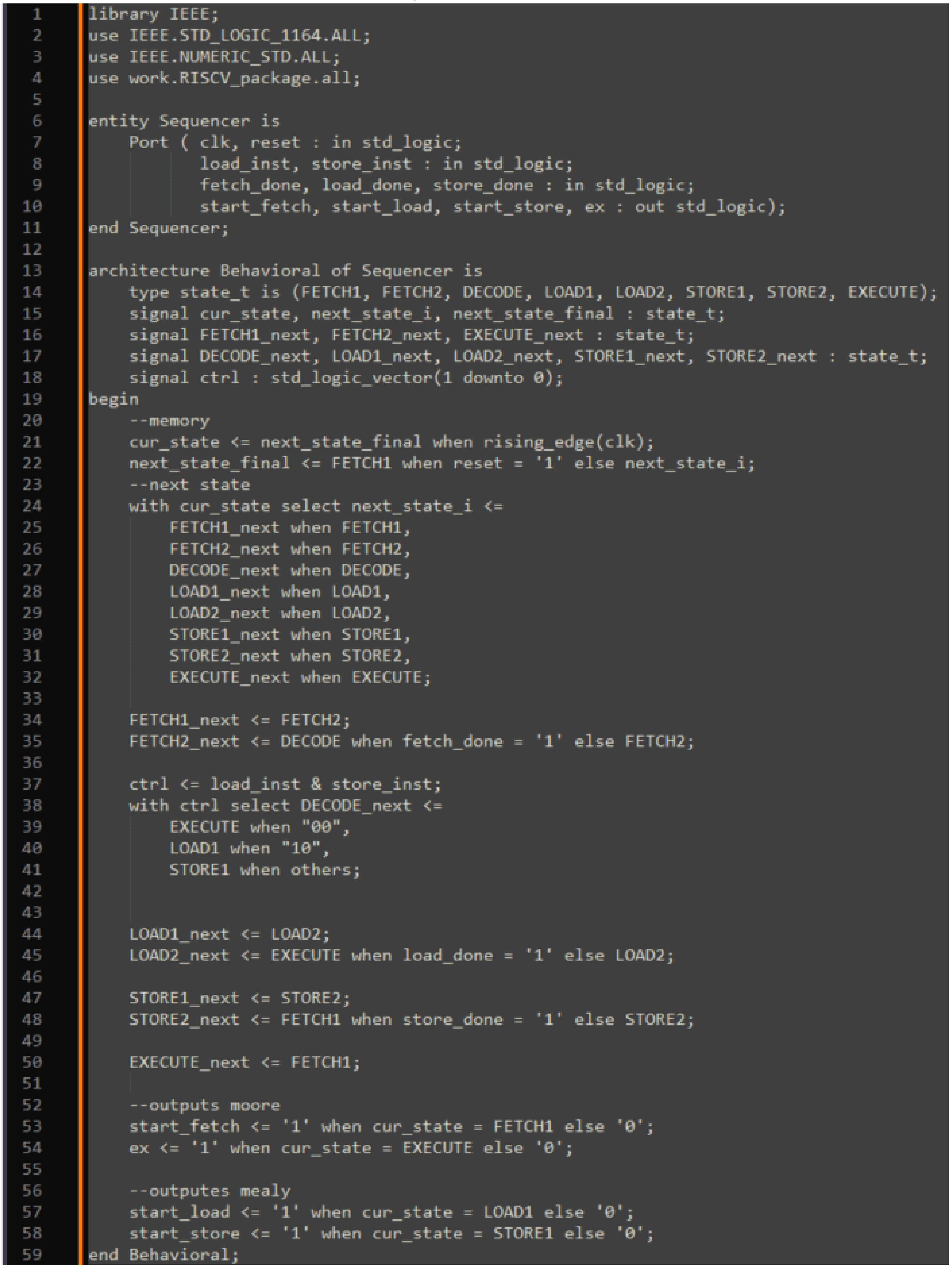


```
architecture implementation of Load_Store is
    type state_t is (IDLE, START, WAITING, ACCEPT);
    signal read_cur_state, read_next_state_i, read_next_state_final : state_t;
    signal read_IDLE_next, read_START_next, read_WAITING_next, read_ACCEPT_next : state_t;
    signal store_cur_state, store_next_state_i, store_next_state_final : state_t;
    signal store_IDLE_next, store_START_next, store_WAITING_next, store_ACCEPT_next : state_t;
    signal load_data, next_load_data, pre_load_data : std_logic_vector(MEM_ADDR_BITS-1 downto 0);
    signal load_data_en : std_logic;
    signal byte_mask, half_mask : std_logic_vector(C_M_AXI_DATA_WIDTH/8-1 downto 0);
begin
    --Load data register
    load_data <= next_load_data when rising_edge(M_AXI_ACLK);
    next_load_data <= (others => '0' ) when M_AXI_ARESETN = '0' else M_AXI_RDATA when load_data_en = '1' else load_data;
    ------------------------------
    -- Read Address Channel
    ------------------------------
    ----------------------------------
    --Read Data Channel
    ----------------------------------
    --constant outputs
    M_AXI_ARID  <= (others => '0');
    M_AXI_ARADDR <= address;
    M_AXI_ARLEN <= "00000000";
    M_AXI_ARSIZE <= '0' & load_store_type(1 downto 0);
    M_AXI_ARBURST <= "01";
    M_AXI_ARLOCK <= '0';
    M_AXI_ARCACHE <= "0010";
    M_AXI_ARPROT <= "100";
    M_AXI_ARQOS <= "0000";


    --memory
    read_cur_state <= read_next_state_final when rising_edge(M_AXI_ACLK);
    read_next_state_final <= IDLE when M_AXI_ARESETN = '0' else read_next_state_i;
    --next state
    with read_cur_state select read_next_state_i <=
        read_IDLE_next when IDLE,
        read_START_next when START,
        read_WAITING_next when WAITING,
        read_ACCEPT_next when ACCEPT;

    read_IDLE_next <= START when start_load = '1' else IDLE;
    read_START_next <= WAITING when M_AXI_ARREADY = '1' else START;
    read_WAITING_next <= ACCEPT when M_AXI_RVALID = '1' else WAITING;
    read_ACCEPT_next <= IDLE;

    --internal signals
    Load_data_en <= '1' when read_cur_state = ACCEPT else '0';
    --Bus outputs
    M_AXI_ARVALID <= '1' when read_cur_state = START else '0';
    M_AXI_RREADY <= '1' when read_cur_state = ACCEPT else '0';
    --external outputs
    load_done <= '1' when read_cur_state = ACCEPT else '0';

    --format load_store_out
    with address(1 downto 0) select pre_load_data <=
       (31 downto 8 => '0') & load_data(15 downto 8) when "01",
       (31 downto 8 => '0') & load_data(31 downto 24) when "11",
       (31 downto 16 => '0') & load_data(31 downto 16) when "10",
       load_data when others;

    with load_store_type select load_data_out <=
       (31 downto 8 => pre_load_data(7)) & pre_load_data(7 downto 0) when "000",
       (31 downto 16 => pre_load_data(15)) & pre_load_data(15 downto 0) when "001",
       (31 downto 8 => '0') & pre_load_data(7 downto 0) when "100",
       (31 downto 16 => '0') & pre_load_data(15 downto 0) when "101",
       pre_load_data when others; --"010"
```



```
    ------------------------------
    -- Write Address Channel
    ------------------------------
    ----------------------------------
    --Write Data Channel
    ----------------------------------
    ----------------------------------
    --Write Response Channel
    ----------------------------------
    --constant outputs
    M_AXI_AWID  <= (others => '0');
    M_AXI_AWADDR <= address;
    M_AXI_AWLEN <= "00000000";
    M_AXI_AWSIZE <= '0' & load_store_type(1 downto 0);
    M_AXI_AWBURST <= "01";
    M_AXI_AWLOCK <= '0';
    M_AXI_AWCACHE <= "0010";
    M_AXI_AWPROT <= "000";
    M_AXI_AWQOS <= "0000";

    with address(1 downto 0) select M_AXI_WDATA <=
       (31 downto 16 => '0') & store_data(7 downto 0) & (7 downto 0 => '0') when "01",
       store_data(7 downto 0) & (23 downto 0 =>'0') when "11",
       store_data(15 downto 0) & (15 downto 0 => '0') when "10",
       store_data when others;
    with address(1 downto 0) select byte_mask <=
        "0001" when "00",
        "0010" when "01",
        "0100" when "10",
        "1000" when others; --"11"
    half_mask <= "0011" when address(0) = '0' else "1100";
    with load_store_type(1 downto 0) select M_AXI_WSTRB <=
        byte_mask when "00",
        half_mask when "01",
        "1111" when others;
    M_AXI_WLAST <= '1'; --only 1 burst tranactions so always last

    --memory
    store_cur_state <= store_next_state_final when rising_edge(M_AXI_ACLK);
    store_next_state_final <= IDLE when M_AXI_ARESETN = '0' else store_next_state_i;
    --next state
    with store_cur_state select store_next_state_i <=
        store_IDLE_next when IDLE,
        store_START_next when START,
        store_WAITING_next when WAITING,
        store_ACCEPT_next when others; --accept

    store_IDLE_next <= START when start_store = '1' else IDLE;
    store_START_next <= WAITING when M_AXI_AWREADY = '1' else START;
    store_WAITING_next <= ACCEPT when M_AXI_WREADY = '1' else WAITING;
    store_ACCEPT_next <= IDLE when M_AXI_BVALID = '1' else ACCEPT;

    --Bus outputs
    M_AXI_AWVALID <= '1' when store_cur_state = START else '0';
    M_AXI_WVALID <= '1' when store_cur_state = WAITING else '0';
    M_AXI_BREADY <= '1' when store_cur_state = ACCEPT else '0';

    --external outputs
    store_done <= '1' when store_cur_state = IDLE else '0';

----error
    Error <= '1' when M_AXI_RRESP(1) = '1' or M_AXI_BRESP(1) = '1' else '0'; --both errors have RRESP bit 1 as high


end implementation;
```

Sequencer VHDL



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.RISCV_package.all;

entity Sequencer is
    Port ( clk, reset : in std_logic;
           load_inst, store_inst : in std_logic;
           fetch_done, load_done, store_done : in std_logic;
           start_fetch, start_load, start_store, ex : out std_logic);
end Sequencer;

architecture Behavioral of Sequencer is
    type state_t is (FETCH1, FETCH2, DECODE, LOAD1, LOAD2, STORE1, STORE2, EXECUTE);
    signal cur_state, next_state_i, next_state_final : state_t;
    signal FETCH1_next, FETCH2_next, EXECUTE_next : state_t;
    signal DECODE_next, LOAD1_next, LOAD2_next, STORE1_next, STORE2_next : state_t;
    signal ctrl : std_logic_vector(1 downto 0);
begin
    --memory
    cur_state <= next_state_final when rising_edge(clk);
    next_state_final <= FETCH1 when reset = '1' else next_state_i;
    --next state
    with cur_state select next_state_i <=
        FETCH1_next when FETCH1,
        FETCH2_next when FETCH2,
        DECODE_next when DECODE,
        LOAD1_next when LOAD1,
        LOAD2_next when LOAD2,
        STORE1_next when STORE1,
        STORE2_next when STORE2,
        EXECUTE_next when EXECUTE;

    FETCH1_next <= FETCH2;
    FETCH2_next <= DECODE when fetch_done = '1' else FETCH2;

    ctrl <= load_inst & store_inst;
    with ctrl select DECODE_next <=
        EXECUTE when "00",
        LOAD1 when "10",
        STORE1 when others;


    LOAD1_next <= LOAD2;
    LOAD2_next <= EXECUTE when load_done = '1' else LOAD2;

    STORE1_next <= STORE2;
    STORE2_next <= FETCH1 when store_done = '1' else STORE2;

    EXECUTE_next <= FETCH1;

    --outputs moore
    start_fetch <= '1' when cur_state = FETCH1 else '0';
    ex <= '1' when cur_state = EXECUTE else '0';

    --outputes mealy
    start_load <= '1' when cur_state = LOAD1 else '0';
    start_store <= '1' when cur_state = STORE1 else '0';
end Behavioral;
```