Lab 3: Fetch and Sequencer

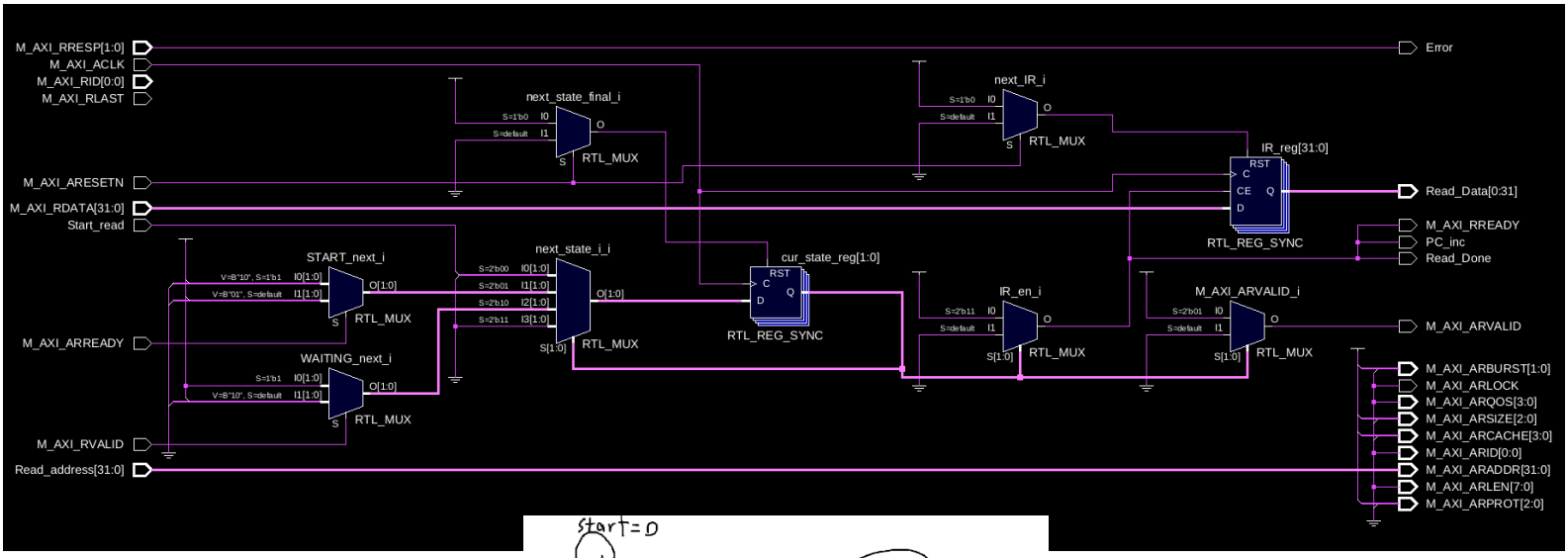
-Overview-

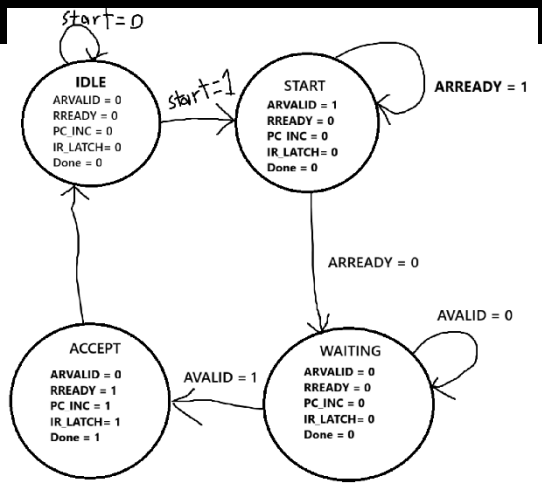
The lab involves building the fetch unit and basic sequencer for an RV32I RISC-V core in VHDL, verified through simulation. The unit should fetch and execute all instructions except loads/stores (a load/store unit will be added later). It must use an AXI4 bus manager interface, specifically the Read Address and Read Data channels.

-Design-

Fetch Unit

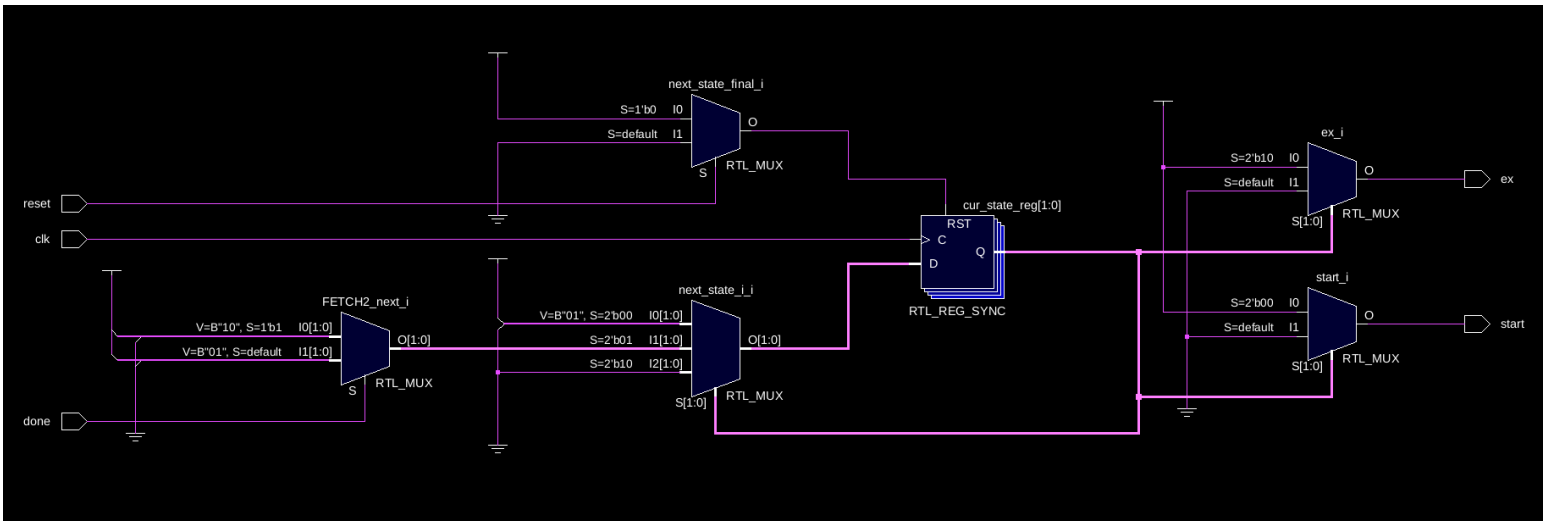
For this state machine, it's linear in the sense every state just goes to the next state, as long as a condition is reached, else it just stays in that state. In the bottom right, you can see most the signals for the AXI bus are just tied to constants, so the message is always a 1 burst 32 bit INCR type transaction. Both the RREADY and ARVALID are selected based on the current state register. So are the PC_inc, Read_Done, and latch IR which is the last state ACCEPT. The reset for the IR and Current state are tied to be active low reset (hence the top muxes which are just inverters). The address is always going to be hardwired to the PC.

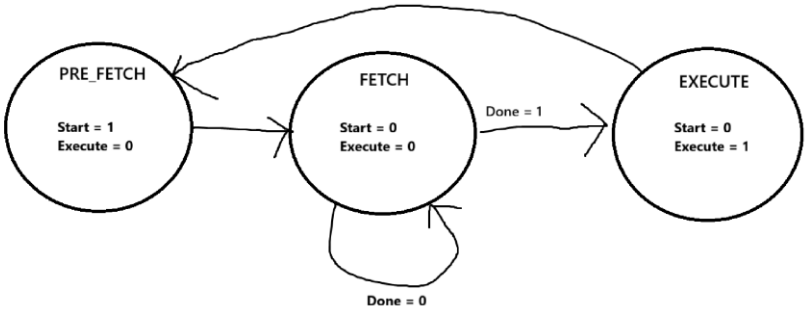




Sequencer Unit

The sequencer is even simpler. It only has one waiting condition and that is waiting on the fetch unit to be done. Once it gets a response, it signals the execute stage, and immediately signals the fetch unit for another instruction.





-Simulation-

Fetch Unit by itself

The fetch unit doesn't do anything until it is signaled on the Start_read line for at least one clock cycle.

1420ns: It signals ARVALID to say that the address (0x00000000) is valid

1430ns: The memory unit responds with a ARREADY to signal it it going to take the address on the next clk edge

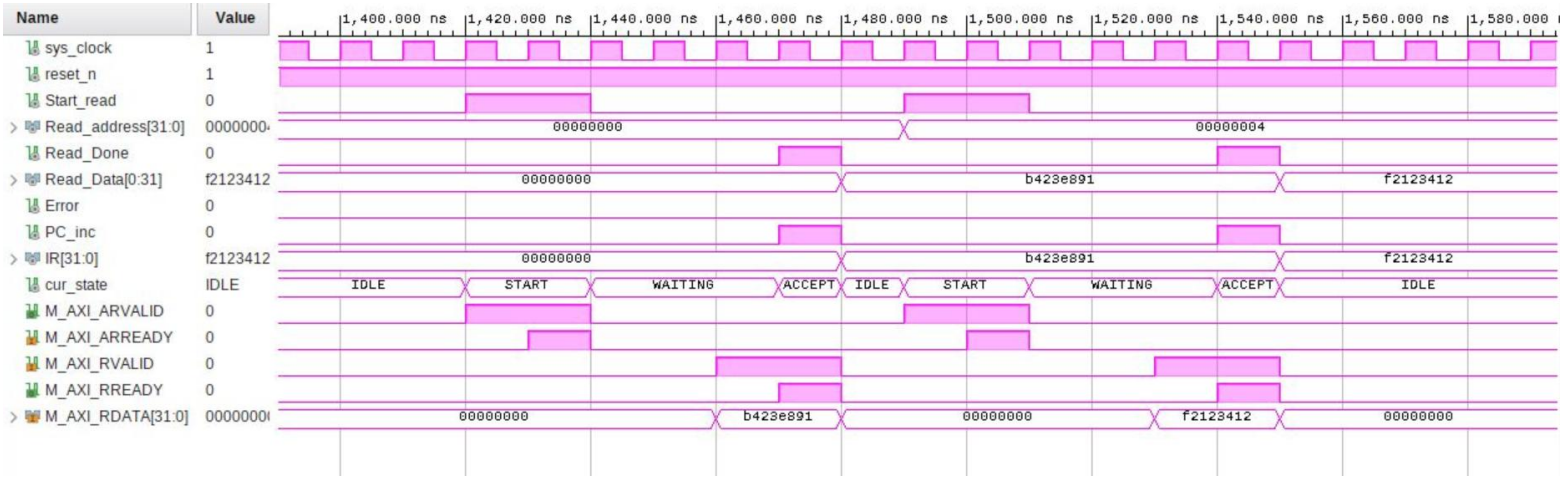
1440ns: Now the memory unit has the address, it is going to fetch the data, and we are going to wait for RVALID

1460ns: RVALID gets asserted high by the memory unity

1470ns: We respond with a RREADY signals to say that the next clk edge we are going to latch it

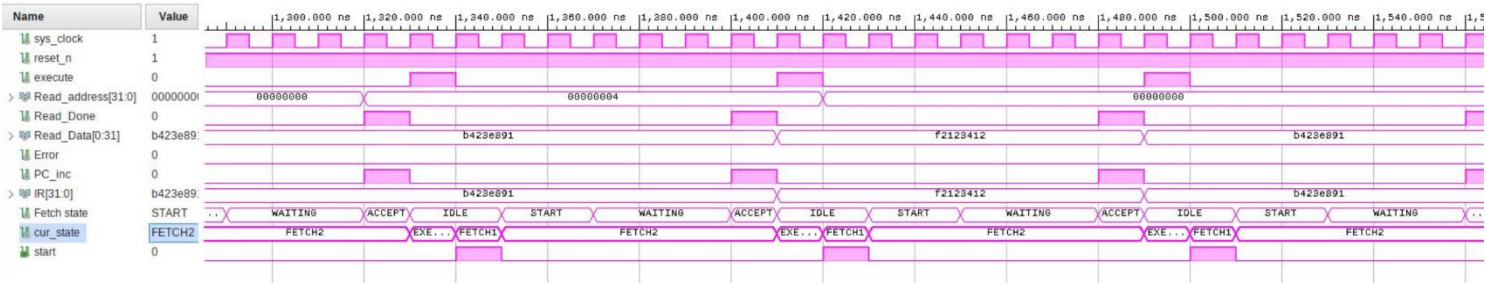
1480ns: PC is incremented, IR latches the data (I put data in memory ahead of time), and we go back to IDLE waiting for another start_read. The instruction (Read_Data) is out for the decoder to read.

1490ns-1560ns: Another fetch at address 4.



Fetch Unit plus Sequencer

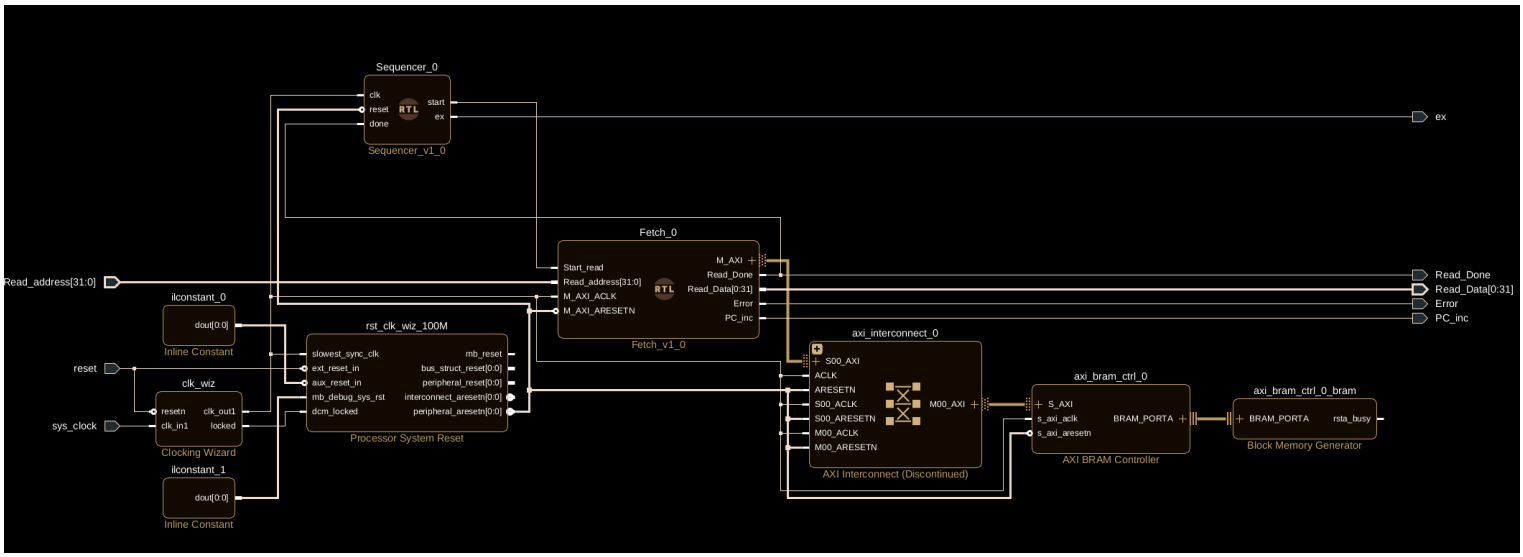
The sequencer is added here. It signals the start automatically when the fetch is done. Here I am setting the read_address which will be the PC, and when the sequencer get around to calling the Fetch unit, the data from that address is eventually grabbed. Once it executes and signals for the fetch unit, it waits for the fetch unit to be done again.



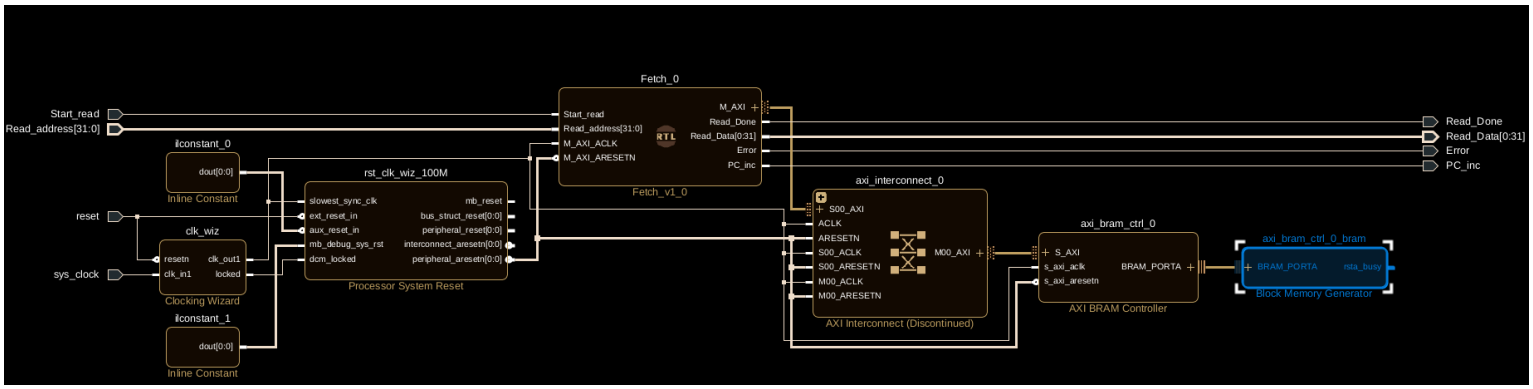
-Conclusion-

The biggest problem I ran into was that I wasn't giving time in my simulation to properly allow the clk wizard and reset unit to properly set. The reset unit and the clk wizard both take some time to rest on startup, so my fetch and sequencer units were not going out of reset. After I figured it out, I had to give some time for it to go out of reset, my tests worked properly. Overall, actual design of the fetch and sequencer was successful.

-Appendix-
Just Fetch Unit Design Overview



Fetch and Sequencer Overview



Fetch VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.RISCV_package.all;

entity Fetch is
    generic (
        -- Users can add generic parameters here

        --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_read   : in std_logic;  -- Initiate AXI read transaction
        Read_address : in std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0); -- address to read from
        Read_Done    : out std_logic; -- Asserts when transaction is complete
        Read_Data : out std_logic_vector(0 to C_M_AXI_DATA_WIDTH*C_M_AXI_BURST_LEN - 1); -- Data that was read (modify as needed)
        Error    : out std_logic; -- Asserts when ERROR is detected
        PC_inc : out std_logic;
        -- 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
        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 transa
        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
        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.
    );
end Fetch;
```

```vhdl
architecture implementation of Fetch is
    type state_t is (IDLE, START, WAITING, ACCEPT);
    signal cur_state, next_state_i, next_state_final : state_t;
    signal IDLE_next, START_next, WAITING_next, ACCEPT_next : state_t;
    signal IR, next_IR : std_logic_vector(MEM_ADDR_BITS-1 downto 0);
    signal IR_en : std_logic;
begin
    --Instruction register
    IR <= next_IR when rising_edge(M_AXI_ACLK);
    next_IR <= (others => '0' ) when M_AXI_ARESETN = '0' else M_AXI_RDATA when IR_en = '1' else IR;
    ------------------------------
    -- Read Address Channel
    ------------------------------
    ----------------------------------
    --Read Data Channel
    ----------------------------------
    --constant outputs
    M_AXI_ARID  <= (others => '0');
    M_AXI_ARADDR <= Read_address;
    M_AXI_ARLEN <= "00000000";
    M_AXI_ARSIZE <= "010";
    M_AXI_ARBURST <= "01";
    M_AXI_ARLOCK <= '0';
    M_AXI_ARCACHE <= "0010";
    M_AXI_ARPROT <= "100";
    M_AXI_ARQOS <= "0000";

    --memory
    cur_state <= next_state_final when rising_edge(M_AXI_ACLK);
    next_state_final <= IDLE when M_AXI_ARESETN = '0' else next_state_i;
    --next state
    with cur_state select next_state_i <=
        IDLE_next when IDLE,
        START_next when START,
        WAITING_next when WAITING,
        ACCEPT_next when ACCEPT;

    IDLE_next <= START when Start_read = '1' else IDLE;
    START_next <= WAITING when M_AXI_ARREADY = '1' else START;
    WAITING_next <= ACCEPT when M_AXI_RVALID = '1' else WAITING;
    ACCEPT_next <= IDLE;

    --internal signals
    IR_en <= '1' when cur_state = ACCEPT else '0';
    --Bus outputs
    M_AXI_ARVALID <= '1' when cur_state = START else '0';
    M_AXI_RREADY <= '1' when cur_state = ACCEPT else '0';
    --external outputs
    Read_Done <= '1' when cur_state = ACCEPT else '0';
    Error <= '1' when M_AXI_RRESP(1) = '1' else '0'; --both errors have RRESP bit 1 as high
    PC_inc <= '1' when cur_state = ACCEPT else '0';
    Read_Data <= IR;


end implementation;
```

Sequencer VHDL

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use work.RISCV_package.all;
5
6  entity Sequencer is
7      Port ( clk, reset : in std_logic;
8             done : in std_logic;
9             start, ex : out std_logic);
10 end Sequencer;
11
12 architecture Behavioral of Sequencer is
13     type state_t is (FETCH1, FETCH2, EXECUTE);
14     signal cur_state, next_state_i, next_state_final : state_t;
15     signal FETCH1_next, FETCH2_next, EXECUTE_next : state_t;
16 begin
17     --memory
18     cur_state <= next_state_final when rising_edge(clk);
19     next_state_final <= FETCH1 when reset = '0' else next_state_i;
20     --next state
21     with cur_state select next_state_i <=
22         FETCH1_next when FETCH1,
23         FETCH2_next when FETCH2,
24         EXECUTE_next when EXECUTE;
25
26     FETCH1_next <= FETCH2;
27     FETCH2_next <= EXECUTE when done = '1' else FETCH2;
28     EXECUTE_next <= FETCH1;
29
30     --outputs
31     start <= '1' when cur_state = FETCH1 else '0';
32     ex <= '1' when cur_state = EXECUTE else '0';
33 end Behavioral;
34
```

Fetch Test Bench

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


entity Fetch_tb is
end Fetch_tb;

architecture Behavioral of Fetch_tb is
    signal sys_clock : STD_LOGIC := '1';
    signal reset_n : STD_LOGIC := '1';
    signal Start_read : STD_LOGIC := '0';
    signal Read_address : STD_LOGIC_VECTOR ( 31 downto 0 ) := (others => '0');
    signal Read_Done : STD_LOGIC := '0';
    signal Read_Data : STD_LOGIC_VECTOR ( 0 to 31 ) := (others => '0');
    signal Error : STD_LOGIC := '0';
    signal PC_inc : std_logic := '0';
begin
    uut: entity work.Whole_system_wrapper (STRUCTURE)
        port map ( sys_clock => sys_clock,
                   reset => reset_n,
                   Start_read => Start_read,
                   Read_address => Read_address,
                   Read_Done => Read_Done,
                   Read_Data => Read_Data,
                   Error => Error,
                   PC_inc => PC_inc);

     --clk
  sys_clock <= not sys_clock after 5 ns;

  test: process
    begin
     wait for 10 ns;
      --reset in beginning
      reset_n <= '0';
      wait for 10 ns;
      reset_n <= '1';
      wait for 1400ns;

      Read_address <= std_logic_vector(to_unsigned(0, Read_address'length));
      Start_read <= '1';
      wait for 20 ns;
      Start_read <= '0';
      wait for 50 ns;
      Read_address <= std_logic_vector(to_unsigned(4, Read_address'length));
      Start_read <= '1';
      wait for 20 ns;
      Start_read <= '0';


      wait;
  end process;
end Behavioral;

```

Fetch and Sequencer

```
 7
 8    entity Fetch_plus_Sequencer_tb is
 9    end Fetch_plus_Sequencer_tb;
10
11    architecture Behavioral of Fetch_plus_Sequencer_tb is
12        signal sys_clock : STD_LOGIC := '1';
13        signal reset_n : STD_LOGIC := '1';
14        signal execute : STD_LOGIC := '0';
15        signal Read_address : STD_LOGIC_VECTOR ( 31 downto 0 ) := (others => '0');
16        signal Read_Done : STD_LOGIC := '0';
17        signal Read_Data : STD_LOGIC_VECTOR ( 0 to 31 ) := (others => '0');
18        signal Error : STD_LOGIC := '0';
19        signal PC_inc : std_logic := '0';
20    begin
21        uut1: entity work.Fetch_sequncer_wrapper (STRUCTURE)
22            port map ( sys_clock => sys_clock,
23                       reset => reset_n,
24                       ex => execute,
25                       Read_address => Read_address,
26                       Read_Done => Read_Done,
27                       Read_Data => Read_Data,
28                       Error => Error,
29                       PC_inc => PC_inc);
30
31          --clk
32      sys_clock <= not sys_clock after 5 ns;
33
34      test: process
35        begin
36         wait for 10 ns;
37          --reset in beginning
38          reset_n <= '0';
39          wait for 10 ns;
40          reset_n <= '1';
41
42          wait for 1300ns;
43
44          Read_address <= std_logic_vector(to_unsigned(4, Read_address'length));
45          wait for 100 ns;
46          Read_address <= std_logic_vector(to_unsigned(0, Read_address'length));
47
48
49
50          wait for 20 ns;
51
52          wait;
53      end process;
54    end Behavioral;
55
```