AHB-Lite Slave for RAM and ROM

Table of Contents

1. Introduction
2. RAM
    - RAM Signals
    - RAM Functionality
    - RAM Code
    - RAM Testbench
    - Testbench Output for RAM
3. ROM
    - ROM Signals
    - ROM Functionality
    - ROM Code
    - ROM Testbench
    - Testbench Output for ROM
4. Block Diagrams
5. Conclusion

Introduction

`Submitted by:` Amr Hossam
`GitHub Repo:` https://github.com/amrhossam9/ADI_Assignments

This document focuses on the **RAM** and **ROM** slave components for the AHB-Lite bus protocol. These components act as memory slaves that store and provide data to the master during bus transactions. This milestone primarily focuses on the designing of these memory components, while the remaining peripherals and decoders and integration will be in the next milestone.

RAM

RAM Signals

- `HSELx`: Select signal indicating the RAM is selected by the decoder.
- `HADDR`: Address signal from the master.
- `HWDATA`: Data to be written to RAM (during write operations).
- `HRDATA`: Data to be read from RAM (during read operations).
- `HWRITE`: Indicates read or write operation.
- `HREADYOUT`: Handshake signal indicating the completion of the transaction.
- `HCLK`: Clock signal.
- `HRESETn`: Active low reset signal.

RAM Functionality

- **Read Operation**: When `HWRITE` is low (`0`), the RAM reads the value stored at `HADDR` and sends it to `HRDATA`.
- **Write Operation**: When `HWRITE` is high (`1`), data from `HWDATA` is written to the memory location specified by `HADDR`.

RAM Code

```
module RAM (
    input  wire        HWRITE, HBURST, HSELx, HREADY,
    input  wire        HRESTn, HCLK,
    input  wire [2:0]  HSIZE,
    input  wire [1:0]  HTRANS,
    input  wire [31:0] HADDR, HWDATA,
    output reg  [31:0] HRDATA,
    output reg         HREADYOUT, HRESP
);

    typedef enum bit {
        IDLE = 1'b0,
        OPERATE = 1'b1
    } states;

    states PS, NS;

    typedef enum bit [2:0] {
        HSIZE_Byte = 3'b000,
        HSIZE_one_word = 3'b010
    } HSIZE_STATES;

    typedef enum bit [1:0] {
        HTRANS_IDLE = 2'b00,
        HTRANS_BUSY = 2'b01,
        HTRANS_NONSEQ = 2'b10,
        HTRANS_SEQ = 2'b11
    } HTRANS_STATES;

    typedef enum bit {
        HBURST_state_single = 0,
        HBURST_state_INCR = 1
    } HBURST_STATES;

    // Define a 32-bit wide memory with 8 words (size can be adjusted as needed)
    reg [31:0] memory [7:0];

    // Initialize memory to zero or some known state (optional)
    initial
    begin
        memory[0] = 32'b01;
        memory[1] = 32'b10;
        memory[2] = 32'b11;
```

```
    end

    // State Machine: Handles transitioning between IDLE and OPERATE
    always @(posedge HCLK or negedge HRESTn)
    begin
        if (!HRESTn)
            PS <= IDLE;
        else
            PS <= NS;
    end

    // Next State and Output Logic
    always @(*)
    begin
        case (PS)
            IDLE:
            begin
                HRDATA = 32'b0;
                HRESP = 1'b0;  // OKAY response

                // If selected and HREADY is high
                if (HREADY && HSELx && HTRANS != HTRANS_IDLE)
                begin
                    NS = OPERATE;
                    HREADYOUT = 0;  // Indicate the RAM is processing
                end
                else
                begin
                    NS = IDLE;
                    HREADYOUT = 1;  // Ready for a new transaction
                end
            end

            OPERATE:
            begin
                if (HWRITE && HSIZE == HSIZE_one_word && HTRANS != HTRANS_BUSY &&
HTRANS != HTRANS_IDLE && HREADY)
                begin
                    // Write to memory when HWRITE is asserted
                    memory[HADDR[2:0]] = HWDATA;  // Use lower address bits to
index memory
                    HRESP = 1'b0;  // OKAY response
                    HREADYOUT = 1; // Indicate transaction complete
                end
                else if (!HWRITE && HSIZE == HSIZE_one_word && HTRANS !=
HTRANS_BUSY && HTRANS != HTRANS_IDLE && HREADY)
                begin
                    // Read from memory when HWRITE is de-asserted
                    HRDATA = memory[HADDR[2:0]];  // Use lower address bits to
index memory
                    HRESP = 1'b0;  // OKAY response
                    HREADYOUT = 1; // Indicate transaction complete
                end
                else
```

```
                begin
                    HREADYOUT = 0; // RAM is busy
                end

                // Handle burst transactions
                if (HBURST == HBURST_state_INCR && HTRANS != HTRANS_IDLE)
                begin
                    NS = OPERATE; // Continue during burst
                end
                else
                begin
                    NS = IDLE;  // Return to IDLE after single transaction
                end
            end

            default:
                NS = IDLE;
        endcase
    end
endmodule
```

RAM Testbench

```
module RAM_TB ();
    reg         HWRITE_TB, HBURST_TB, HSELx_TB, HREADY_TB;
    reg         HRESTn_TB, HCLK_TB;
    reg  [2:0]  HSIZE_TB;
    reg  [1:0]  HTRANS_TB;
    reg  [31:0] HADDR_TB, HWDATA_TB;
    wire [31:0] HRDATA_TB;
    wire        HREADYOUT_TB, HRESP_TB;

    parameter CLK_PERIOD = 10;

    typedef enum bit {
        IDLE = 1'b0,
        SEND = 1'b1
    } states;

    states PS, NS;

    typedef enum bit [2:0] {
        HSIZE_Byte = 3'b000,
        HSIZE_one_word = 3'b010
    } HSIZE_STATES;

    typedef enum bit [1:0] {
        HTRANS_IDLE = 2'b00,
        HTRANS_BUSY = 2'b01,
        HTRANS_NONSEQ = 2'b10,
        HTRANS_SEQ = 2'b11
```

```
} HTRANS_STATES;

typedef enum bit {
    HBURST_state_single = 0,
    HBURST_state_INCR = 1
} HBURST_STATES;

initial
begin
    $dumpfile("RAM_Dump.vcd");
    $dumpvars;
    initialize();
    TEST_READ_INCR();
    #CLK_PERIOD;
    TEST_READ_single();
    #CLK_PERIOD;
    TEST_WRITE_INCR();
    #CLK_PERIOD;
    TEST_WRITE_single();
    #100;
    $finish;
end

task initialize;
begin
    HWRITE_TB = 0;
    HBURST_TB = HBURST_state_single;
    HSELx_TB = 0;
    HREADY_TB = 0;
    HRESTn_TB = 0;
    HCLK_TB = 0;
    HSIZE_TB = HSIZE_one_word;
    HTRANS_TB = HTRANS_IDLE;
    HADDR_TB = 0;
    HWDATA_TB = 0;
end
endtask

task rest;
begin
    HRESTn_TB = 'b1;
    #CLK_PERIOD;
    HRESTn_TB = 'b0;
    #CLK_PERIOD;
    HRESTn_TB = 'b1;
end
endtask

task TEST_READ_INCR;
begin
    $display("Test Increment read");
    rest();
    HWRITE_TB = 0;
    HSELx_TB = 1;
```

```
        HREADY_TB = 1;
        HTRANS_TB = HTRANS_NONSEQ;
        HADDR_TB = 32'b0000_0000;
        HSIZE_TB = HSIZE_one_word;
        HBURST_TB = HBURST_state_INCR;

        #CLK_PERIOD;
        wait (HREADYOUT_TB);
        if(HRDATA_TB == 32'b0000_0001)
        begin
            $display("operation 1 %h succeeded", HRDATA_TB);
        end
        else
        begin
            $display("operation 1 %h Failed", HRDATA_TB);
        end
        HTRANS_TB = HTRANS_SEQ;
        HADDR_TB = 32'b0000_0001;

        #CLK_PERIOD;
        wait (HREADYOUT_TB);
        if(HRDATA_TB == 32'b0000_0010)
        begin
            $display("operation 2 %h succeeded", HRDATA_TB);
        end
        else
        begin
            $display("operation 2 %h Failed", HRDATA_TB);
        end
        HADDR_TB = 32'b0000_0010;

        #CLK_PERIOD;
        wait (HREADYOUT_TB);
        if(HRDATA_TB == 32'b0000_0011)
        begin
            $display("operation 3 %h succeeded", HRDATA_TB);
        end
        else
        begin
            $display("operation 3 %h Failed", HRDATA_TB);
        end

        #CLK_PERIOD;
        HTRANS_TB = HTRANS_IDLE;
    end
    endtask

    task TEST_READ_single;
    begin
        $display("Test single read");
        // Test non-sequential read
        HTRANS_TB = HTRANS_NONSEQ;
        HADDR_TB = 32'b0000_0000;
        HSIZE_TB = HSIZE_one_word;
```

```
        HREADY_TB = 1;
        HSELx_TB = 1;
        #CLK_PERIOD;

        // Check read data
        if (HRDATA_TB == 32'b0000_0001)
            $display("Read operation %h succeeded", HRDATA_TB);
        else
            $display("Read operation %h failed", HRDATA_TB);
        HTRANS_TB = HTRANS_IDLE;
        #CLK_PERIOD;
    end
    endtask

    task TEST_WRITE_INCR;
    begin
        $display("Test Increment write");
        rest();
        HWRITE_TB = 1;
        HSELx_TB = 1;
        HREADY_TB = 1;
        HTRANS_TB = HTRANS_NONSEQ;
        HADDR_TB = 32'b0000_0000;
        HSIZE_TB = HSIZE_one_word;
        HBURST_TB = HBURST_state_INCR;
        HWDATA_TB = 32'b1000;

        #CLK_PERIOD;
        if(DUT.memory[0] == 32'b0000_1000)
        begin
            $display("operation 1 %h succeeded", DUT.memory[0]);
        end
        else
        begin
            $display("operation 1 %h Failed", DUT.memory[0]);
        end
        HTRANS_TB = HTRANS_SEQ;
        HADDR_TB = 32'b0000_0001;
        HWDATA_TB = 32'b1001;

        #CLK_PERIOD;
        if(DUT.memory[1] == 32'b0000_1001)
        begin
            $display("operation 2 %h succeeded", DUT.memory[1]);
        end
        else
        begin
            $display("operation 2 %h Failed", DUT.memory[1]);
        end
        HADDR_TB = 32'b0000_0010;
        HWDATA_TB = 32'b1101;

        #CLK_PERIOD;
        if(DUT.memory[2] == 32'b0000_1101)
```

```
        begin
            $display("operation 3 %h succeeded", DUT.memory[2]);
        end
        else
        begin
            $display("operation 3 %h Failed", DUT.memory[2]);
        end
        HADDR_TB = 32'b0000_0011;
        HWDATA_TB = 32'b1111;

        #CLK_PERIOD;
        if(DUT.memory[3] == 32'b0000_1111)
        begin
            $display("operation 4 %h succeeded", DUT.memory[3]);
        end
        else
        begin
            $display("operation 4 %h Failed", DUT.memory[3]);
        end
        HTRANS_TB = HTRANS_IDLE;
    end
    endtask

    task TEST_WRITE_single;
    begin
        $display("Test single write");
        // Test non-sequential read
        HTRANS_TB = HTRANS_NONSEQ;
        HADDR_TB = 32'b0000_0000;
        HREADY_TB = 1;
        HSELx_TB = 1;
        HWDATA_TB = 32'b0010;
        #CLK_PERIOD;

        // Check read data
        if (DUT.memory[0] == 32'b0000_0010)
            $display("Read operation %h succeeded", HRDATA_TB);
        else
            $display("Read operation %h failed", HRDATA_TB);
    end
    endtask

    always #(CLK_PERIOD/2) HCLK_TB = ~HCLK_TB;

    RAM DUT (
        .HWRITE(HWRITE_TB),
        .HBURST(HBURST_TB),
        .HSELx(HSELx_TB),
        .HREADY(HREADY_TB),
        .HRESTn(HRESTn_TB),
        .HCLK(HCLK_TB),
        .HSIZE(HSIZE_TB),
        .HTRANS(HTRANS_TB),
        .HADDR(HADDR_TB),
```

```
        .HWDATA(HWDATA_TB),
        .HRDATA(HRDATA_TB),
        .HREADYOUT(HREADYOUT_TB),
        .HRESP(HRESP_TB)
    );
endmodule
```

Testbench Output for RAM

```
# Test Increment read
# operation 1 00000001 succeeded
# operation 2 00000002 succeeded
# operation 3 00000003 succeeded
# Test single read
# Read operation 00000001 succeeded
# Test Increment write
# operation 1 00000008 succeeded
# operation 2 00000009 succeeded
# operation 3 0000000d succeeded
# operation 4 0000000f succeeded
# Test single write
# Read operation 00000000 succeeded
```

ROM

ROM Signals

- `HSEL`: Select signal indicating the ROM is selected by the decoder.
- `HADDR`: Address signal from the master.
- `HRDATA`: Data read from ROM.
- `HWRITE`: Should always be low (`0`) since ROM is read-only.
- `HREADYOUT`: Handshake signal indicating the completion of the transaction.
- `HCLK`: Clock signal.
- `HRESETn`: Active low reset signal.

ROM Functionality

- **Read Operation**: When `HWRITE` is low (`0`), the ROM provides data from the memory location specified by `HADDR` to `HRDATA`. ROM is read-only, so no write operations are allowed.

ROM Code

```
module ROM (
    input  wire        HWRITE, HBURST, HSELx, HREADY,
    input  wire        HRESTn, HCLK,
    input  wire [2:0]  HSIZE,
    input  wire [1:0]  HTRANS,
    input  wire [31:0] HADDR, HWDATA,
```

```
    output reg  [31:0] HRDATA,
    output reg         HREADYOUT, HRESP
);

    typedef enum bit {
        IDLE = 1'b0,
        OPERATE = 1'b1
    } states;

    states PS, NS;

    typedef enum bit [2:0] {
        HSIZE_Byte = 3'b000,
        HSIZE_one_word = 3'b010
    } HSIZE_STATES;

    typedef enum bit [1:0] {
        HTRANS_IDLE = 2'b00,
        HTRANS_BUSY = 2'b01,
        HTRANS_NONSEQ = 2'b10,
        HTRANS_SEQ = 2'b11
    } HTRANS_STATES;

    typedef enum bit {
        HBURST_state_single = 0,
        HBURST_state_INCR = 1
    } HBURST_STATES;

    // Define a 32-bit wide memory with 8 words
    reg [31:0] memory [7:0];

    // Initialize memory with some data
    initial
    begin
        memory[0] = 32'b0000_0010;
        memory[1] = 32'b1000_0001;
        memory[2] = 32'b1000_0010;
        memory[3] = 32'b1000_0011;
    end

    always @(posedge HCLK or negedge HRESTn)
    begin
        if (!HRESTn) begin
            PS <= IDLE;
        end
        else begin
            PS <= NS;
        end
    end

    always @(*)
    begin
        case (PS)
            IDLE:
```

```
            begin
                HRDATA = 32'b0;
                HRESP = 1'b0;  // OKAY response

                // If selected and HREADY is high
                if (HREADY && HSELx && HTRANS != HTRANS_IDLE)
                begin
                    NS = OPERATE;
                    HREADYOUT = 0;
                end
                else begin
                    NS = IDLE;
                    HREADYOUT = 1;
                end
            end

            OPERATE:
            begin
                // Read from memory when not busy, HWRITE is 0, and a byte
transfer
                if (HWRITE == 0 && HSIZE == HSIZE_Byte && HTRANS != HTRANS_BUSY &&
HTRANS != HTRANS_IDLE && HREADY)
                begin
                    HREADYOUT = 1;
                    HRESP = 1'b0;  // OKAY response
                    HRDATA = memory[HADDR[2:0]]; // Use lower address bits to
index memory
                end
                else
                begin
                    HREADYOUT = 0; // ROM is busy
                    HRESP = 1'b1;  // ERROR response for unsupported operation
                    HRDATA = 0;
                end

                // Handle burst transfers
                if (HBURST == HBURST_state_INCR && HTRANS != HTRANS_IDLE) begin
                    NS = OPERATE; // Continue sending during burst
                end
                else begin
                    NS = IDLE; // Go back to IDLE after single transfer
                end
            end

            default:
                NS = IDLE;
        endcase
    end
endmodule
```

ROM Testbench

```
module ROM_TB ();
    reg         HWRITE_TB, HBURST_TB, HSELx_TB, HREADY_TB;
    reg         HRESTn_TB, HCLK_TB;
    reg  [2:0]  HSIZE_TB;
    reg  [1:0]  HTRANS_TB;
    reg  [31:0] HADDR_TB, HWDATA_TB;
    wire [31:0] HRDATA_TB;
    wire        HREADYOUT_TB, HRESP_TB;

    parameter CLK_PERIOD = 10;

    typedef enum bit {
        IDLE = 1'b0,
        SEND = 1'b1
    } states;

    states PS, NS;

    typedef enum bit [2:0] {
        HSIZE_Byte = 3'b000,
        HSIZE_one_word = 3'b010
    } HSIZE_STATES;

    typedef enum bit [1:0] {
        HTRANS_IDLE = 2'b00,
        HTRANS_BUSY = 2'b01,
        HTRANS_NONSEQ = 2'b10,
        HTRANS_SEQ = 2'b11
    } HTRANS_STATES;

    typedef enum bit {
        HBURST_state_single = 0,
        HBURST_state_INCR = 1
    } HBURST_STATES;

    initial
    begin
        $dumpfile("ROM_Dump.vcd");
        $dumpvars;
        initialize();
        TEST_INCR();
        #CLK_PERIOD;
        TEST_single();
        #100;
        $finish;
    end

    task initialize;
    begin
        HWRITE_TB = 0;
```

```
        HBURST_TB = HBURST_state_single;
        HSELx_TB = 0;
        HREADY_TB = 0;
        HRESTn_TB = 0;
        HCLK_TB = 0;
        HSIZE_TB = HSIZE_one_word;
        HTRANS_TB = HTRANS_IDLE;
        HADDR_TB = 0;
        HWDATA_TB = 0;
    end
    endtask

    task rest;
    begin
        HRESTn_TB = 'b1;
        #CLK_PERIOD;
        HRESTn_TB = 'b0;
        #CLK_PERIOD;
        HRESTn_TB = 'b1;
    end
    endtask

    task TEST_INCR;
    begin
        rest();
        HWRITE_TB = 0;
        HSELx_TB = 1;
        HREADY_TB = 1;
        HTRANS_TB = HTRANS_NONSEQ;
        HADDR_TB = 32'b0000_0000;
        HSIZE_TB = HSIZE_Byte;
        HBURST_TB = HBURST_state_INCR;

        #CLK_PERIOD;
        wait (HREADYOUT_TB);
        if(HRDATA_TB == 32'b0000_0010)
        begin
            $display("instruction 1 %h succeeded", HRDATA_TB);
        end
        else
        begin
            $display("instruction 1 %h Failed", HRDATA_TB);
        end
        HTRANS_TB = HTRANS_SEQ;
        HADDR_TB = 32'b0000_0001;

        #CLK_PERIOD;
        wait (HREADYOUT_TB);
        if(HRDATA_TB == 32'b1000_0001)
        begin
            $display("instruction 2 %h succeeded", HRDATA_TB);
        end
        else
        begin
```

```
            $display("instruction 2 %h Failed", HRDATA_TB);
        end
        HADDR_TB = 32'b0000_0010;

        #CLK_PERIOD;
        wait (HREADYOUT_TB);
        if(HRDATA_TB == 32'b1000_0010)
        begin
            $display("instruction 3 %h succeeded", HRDATA_TB);
        end
        else
        begin
            $display("instruction 3 %h Failed", HRDATA_TB);
        end
        HADDR_TB = 32'b0000_0011;

        #CLK_PERIOD;
        wait (HREADYOUT_TB);
        if(HRDATA_TB == 32'b1000_0011)
        begin
            $display("instruction 4 %h succeeded", HRDATA_TB);
        end
        else
        begin
            $display("instruction 4 %h Failed", HRDATA_TB);
        end
        HTRANS_TB = HTRANS_IDLE;
    end
    endtask

    task TEST_single;
    begin
        // Test non-sequential read
        HTRANS_TB = HTRANS_NONSEQ;
        HADDR_TB = 32'b0000_0000;
        HREADY_TB = 1;
        HSELx_TB = 1;
        #CLK_PERIOD;

        // Check read data
        if (HRDATA_TB == 32'b0000_0010)
            $display("Read instruction %h succeeded", HRDATA_TB);
        else
            $display("Read instruction %h failed", HRDATA_TB);
        // Test idle transition
        HTRANS_TB = HTRANS_IDLE;
        #CLK_PERIOD;

        // Check no response during idle
        if (HREADYOUT_TB && HRDATA_TB == 32'h00000000)
            $display("Idle state %h succeeded", HRDATA_TB);
        else
            $display("Idle state failed %h", HRDATA_TB);
    end
```

```
    endtask

    always #(CLK_PERIOD/2) HCLK_TB = ~HCLK_TB;

    ROM DUT (
        .HWRITE(HWRITE_TB),
        .HBURST(HBURST_TB),
        .HSELx(HSELx_TB),
        .HREADY(HREADY_TB),
        .HRESTn(HRESTn_TB),
        .HCLK(HCLK_TB),
        .HSIZE(HSIZE_TB),
        .HTRANS(HTRANS_TB),
        .HADDR(HADDR_TB),
        .HWDATA(HWDATA_TB),
        .HRDATA(HRDATA_TB),
        .HREADYOUT(HREADYOUT_TB),
        .HRESP(HRESP_TB)
    );
endmodule
```

Testbench Output for ROM

```
# instruction 1 00000002 succeeded
# instruction 2 00000081 succeeded
# instruction 3 00000082 succeeded
# instruction 4 00000083 succeeded
# Read instruction 00000002 succeeded
# Idle state 00000000 succeeded
```

Block Diagrams

The RAM and ROM modules share a similar block structure, as they both function as memory slaves in the system.

Combined Memory Block Diagram

Conclusion

In **Milestone 2**, i developed the **RAM** and **ROM** modules as AHB-Lite slaves. These components will be integrated with additional peripherals in **Milestone 3**. The decoders and multiplexors required to select and route data from multiple memory blocks will also be added during the integration phase.