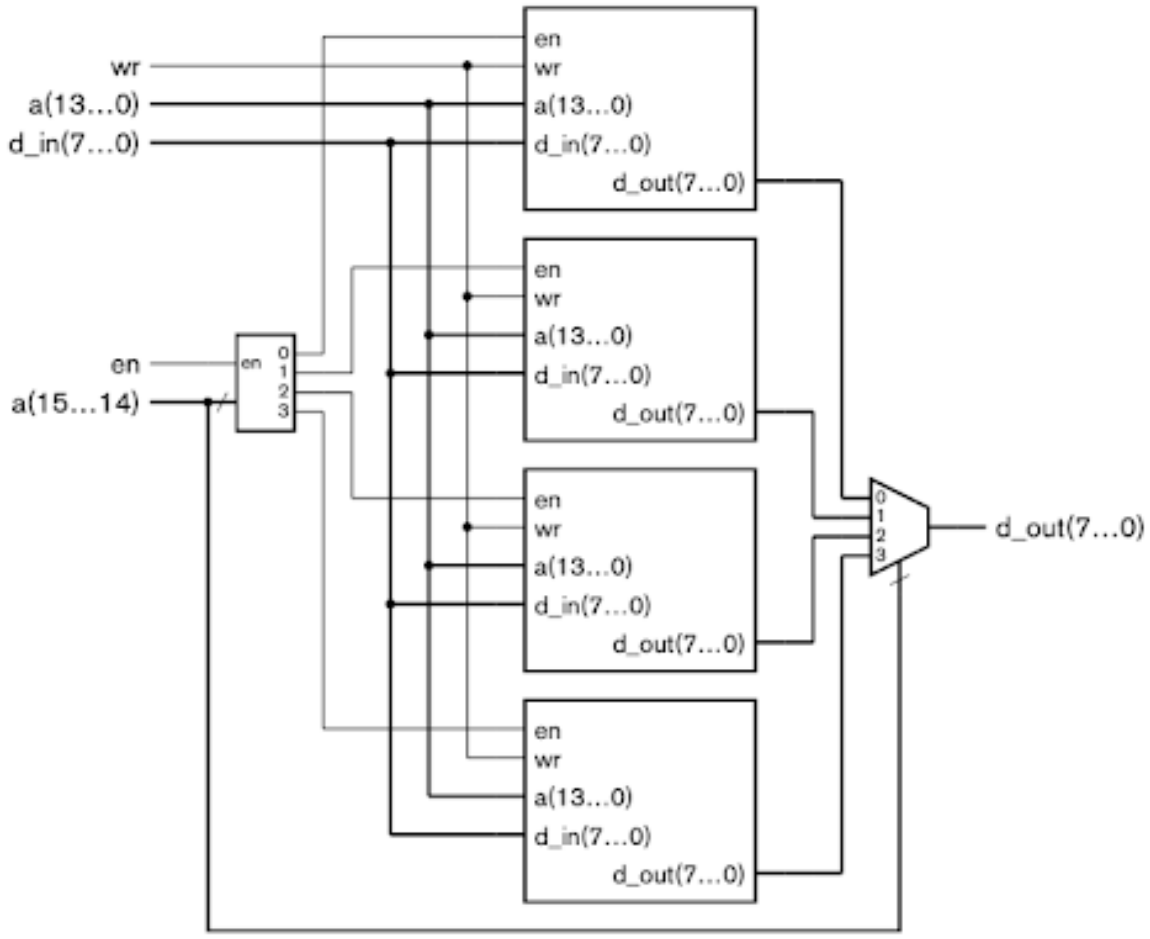
Assignment No.1

EE5530 : Principles of SoC Functional Verification

Nakul C - 122101024

Problem Statement

- **Implement the design. Introduce a bug in the decoder logic. Write a task based Verilog testbench to catch it.**



Simple Memory Module

```
1 // Simple Memory Module in Verilog
2 module simple_memory #(parameter ADDR_WIDTH = 12, DATA_WIDTH = 16) (
3     input clk, // Clock
```

```
    input we,  // Write Enable
    input [ADDR_WIDTH-1:0] addr,
    input [DATA_WIDTH-1:0] din,
    output reg [DATA_WIDTH-1:0] dout
);
    reg [DATA_WIDTH-1:0] mem [0:(1<<ADDR_WIDTH)-1];

    always @(posedge clk) begin
        if (we)
            mem[addr] <= din; // Write operation
        else
            dout <= mem[addr]; // Read operation
    end
endmodule
```

The given Verilog module implements a **synchronous memory unit** with configurable address and data widths, making it adaptable for various applications requiring temporary data storage.

Parameterized Address and Data Width

The module allows customization of the address (`ADDR_WIDTH`) and data (`DATA_WIDTH`) sizes using parameters. The memory size is determined as $2^{\text{ADDR\_WIDTH}}$ locations, each storing `DATA_WIDTH`-bit values.

Memory Array Declaration

A register array `mem` is declared to store data, where the number of memory locations is defined as $2^{\text{ADDR\_WIDTH}}$.

Clock-Synchronized Read and Write Operations

- **Write Operation** (`we = 1`): The input data (`din`) is stored at the memory location specified by `addr`.
- **Read Operation** (`we = 0`): The data from the memory location specified by `addr` is assigned to the output (`dout`).
- Both operations occur on the **rising edge of the clock** (`posedge clk`), ensuring synchronization.

Composite Memory

```
// Simple Memory Module in Verilog
module simple_memory #(parameter ADDR_WIDTH = 12, DATA_WIDTH = 16) (
    input clk, // Clock
    input we,  // Write Enable
    input [ADDR_WIDTH-1:0] addr,
    input [DATA_WIDTH-1:0] din,
    output reg [DATA_WIDTH-1:0] dout
);
```

```
9       reg [DATA_WIDTH-1:0] mem [0:(1<<ADDR_WIDTH)-1];
10
11      always @(posedge clk) begin
12          if (we)
13              mem[addr] <= din; // Write operation
14          else
15              dout <= mem[addr]; // Read operation
16      end
17  endmodule
18  // Composite Memory Module in Verilog
19  module composite_memory (
20      input clk,   // Clock
21      input we,    // Write Enable
22      input [15:0] addr,  // 16-bit address
23      input [7:0] din,    // 8-bit input data
24      output reg [7:0] dout  // 8-bit output data
25  );
26      // Memory bank selection (4 memory modules)
27      wire [1:0] bank_sel = addr[15:14]; // Upper 2 bits decide the bank
28      wire [13:0] local_addr = addr[13:0]; // Lower 14 bits for memory
            access
29
30      // Output data from each memory bank
31      wire [7:0] dout0, dout1, dout2, dout3;
32
33      // Memory Modules (16K x 8-bit each)
34      simple_memory #(14, 8) mem0 (.clk(clk), .we(we & (bank_sel == 2'b00)
            ), .addr(local_addr), .din(din), .dout(dout0));
35      simple_memory #(14, 8) mem1 (.clk(clk), .we(we & (bank_sel == 2'b01)
            ), .addr(local_addr), .din(din), .dout(dout1));
36      simple_memory #(14, 8) mem2 (.clk(clk), .we(we & (bank_sel == 2'b10)
            ), .addr(local_addr), .din(din), .dout(dout2));
37      simple_memory #(14, 8) mem3 (.clk(clk), .we(we & (bank_sel == 2'b11)
            ), .addr(local_addr), .din(din), .dout(dout3));
38
39      // Read MUX: Select the correct memory module's output
40      always @(*) begin
41          case (bank_sel)
42              2'b00: dout = dout0;
43              2'b01: dout = dout1;
44              2'b10: dout = dout2;
45              2'b11: dout = dout3;
46              default: dout = 8'b0;
47          endcase
48      end
49  endmodule
```

Memory Bank Selection

The composite memory is divided into 4 separate memory banks. Each memory bank is created using the `simple_memory` module. These banks are selected based on the upper 2 bits of the address (16-bit address). The selection mechanism is done through the `bank_sel` signal,

which is derived from the upper 2 bits of the address (`addr[15:14]`). This 2-bit selection chooses one of the four banks:

- 00 → Bank 0
- 01 → Bank 1
- 10 → Bank 2
- 11 → Bank 3

Memory Bank Structure

Each of the 4 memory banks is instantiated using the `simple_memory` module, with each module having a 16K x 8-bit memory configuration (`simple_memory #(14, 8)`). The `local_addr` (the lower 14 bits of the address) is used to access the memory locations within each of the four banks. This ensures that each memory module has access to the same address space, but only one bank can be accessed at a time based on the `bank_sel` signal.

Write Enable Control

The write enable (`we`) signal is controlled so that only the selected memory bank can perform write operations. This is achieved by using the condition `we & (bank_sel == <bank>)` for each memory bank:

- For example, `we & (bank_sel == 2'b00)` ensures that only the first bank performs a write operation when `we` is high and the `bank_sel` is 00.

This selective control prevents write conflicts between the memory banks, ensuring that each bank operates independently.

Read Operation

When reading data, the output data is selected based on the memory bank using a multiplexer (case statement). Depending on the value of `bank_sel`, the corresponding memory bank's output (`dout0`, `dout1`, `dout2`, `dout3`) is assigned to the output (`dout`) of the composite memory module. For example, when `bank_sel == 2'b00`, the output is taken from `dout0`, which is the output of `mem0`.

Composite Memory Testbench

```
1  // Task Based Testbench for the Composite Memory
2  module composite_memory_tb;
3      reg clk;
4      reg we;
5      reg [15:0] addr;
6      reg [7:0] din;
7      wire [7:0] dout;
8      reg [7:0] r_data;  // Declare r_data before use
9
10     // Instantiate the composite_memory module
11     composite_memory uut (
```

```
12          .clk(clk),
13          .we(we),
14          .addr(addr),
15          .din(din),
16          .dout(dout)
17      );
18
19      // Clock generation
20      always #5 clk = ~clk;
21
22      // Task for writing data
23      task write_mem;
24          input [15:0] t_addr;
25          input [7:0] t_din;
26          begin
27              @(posedge clk);
28              we = 1;
29              addr = t_addr;
30              din = t_din;
31              @(posedge clk);
32              we = 0; // Disable write after one cycle
33          end
34      endtask
35
36      // Task for reading data (fix: store output in `r_data`)
37      task read_mem;
38          input [15:0] t_addr;
39          begin
40              @(posedge clk);
41              addr = t_addr;
42              we = 0;  // Ensure write is disabled
43              @(posedge clk);
44              #1;  // Small delay to allow dout to stabilize
45              r_data = dout;
46          end
47      endtask
48
49
50      initial begin
51
52          $dumpfile("waveform.vcd");
53          $dumpvars(0, composite_memory_tb);
54
55          // Initialize signals
56          clk = 0;
57          we = 0;
58          addr = 0;
59          din = 0;
60
61          #10; // Wait for reset period
62
63          // Write and Read test
64          $display("Writing and Reading Test");
```

```
65          write_mem(16'h0000, 8'hA5); // Write to bank 0
66          write_mem(16'h4001, 8'h5A); // Write to bank 1
67          write_mem(16'h8002, 8'h3C); // Write to bank 2
68          write_mem(16'hC003, 8'h7E); // Write to bank 3
69
70          #10; // Wait some time
71
72          // Read back values
73          read_mem(16'h0000);
74          $display("Read from 0x0000: %h (Expected: A5)", r_data);
75          read_mem(16'h4001);
76          $display("Read from 0x4001: %h (Expected: 5A)", r_data);
77          read_mem(16'h8002);
78          $display("Read from 0x8002: %h (Expected: 3C)", r_data);
79          read_mem(16'hC003);
80          $display("Read from 0xC003: %h (Expected: 7E)", r_data);
81
82          #10;
83          $stop;
84      end
85  endmodule
```

This testbench verifies the functionality of the `composite_memory` module, which consists of multiple memory banks that are selected based on the address input. The testbench exercises both write and read operations, ensuring that the composite memory is functioning as intended.

Module Instantiation

The testbench begins by instantiating the `composite_memory` module (`uut` stands for Unit Under Test), passing the necessary inputs:

- **Clock** (`clk`)
- **Write Enable** (`we`)
- **Address** (`addr`)
- **Data Input** (`din`)
- **Data Output** (`dout`)

Clock Generation

The clock signal (`clk`) is generated using an `always` block, toggling every 5 time units (#5), which ensures a continuous clock signal for the entire simulation.

Task Definitions

Two tasks are defined within the testbench to simplify and modularize the test process: `write_mem` and `read_mem`.

Write Memory Task (`write_mem`)

This task writes data to the memory by setting the `we` signal to 1, assigning an address (`addr`) and data input (`din`), and then waiting for one clock cycle. After the data is written, the `we` signal is set to 0 to disable writing.

Read Memory Task (`read_mem`)

This task reads data from the memory by setting the `we` signal to 0 (to ensure the module is in read mode), assigning the address (`addr`), and then waiting for the next clock cycle. After the clock edge, a small delay (#1) ensures that the output data (`dout`) has stabilized, after which it is assigned to `r_data`.

Test Procedure

In the initial block, the signals are initialized, and a sequence of memory write and read operations is performed:

- **Write Operations**: Four different addresses are selected for writing data into the memory:
  - Address `16'h0000`: Data `8'hA5` (Bank 0)
  - Address `16'h4001`: Data `8'h5A` (Bank 1)
  - Address `16'h8002`: Data `8'h3C` (Bank 2)
  - Address `16'hC003`: Data `8'h7E` (Bank 3)

  The task `write_mem` is called for each address and corresponding data.
- **Read Operations**: After a short delay (#10), the memory is read back from the same addresses:
  - Address `16'h0000`: Expected data `A5`
  - Address `16'h4001`: Expected data `5A`
  - Address `16'h8002`: Expected data `3C`
  - Address `16'hC003`: Expected data `7E`

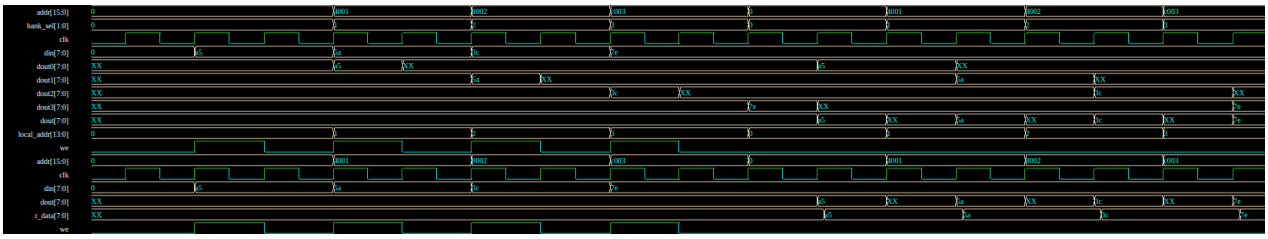
  The task `read_mem` is used to read from each address and store the result in `r_data`. The values of `r_data` are displayed using the `$display` system task.

The expected output in the `$display` statements would be:

- Reading from address `0x0000` should yield `A5`.
- Reading from address `0x4001` should yield `5A`.
- Reading from address `0x8002` should yield `3C`.
- Reading from address `0xC003` should yield `7E`.

Waveform Dumping

The testbench also includes functionality to dump the simulation waveform for visualization. This is done using the `$dumpfile` and `$dumpvars` system tasks. The generated waveform can be viewed using a waveform viewer, providing a visual representation of the signals over time.



Simulation Control

The simulation runs for a short period, waits for the write and read operations to complete, and then halts with the `$stop` command. This allows for an examination of the results at the end of the simulation.

```
# KERNEL: Writing and Reading Test
# KERNEL: Read from 0x0000: a5 (Expected: A5)
# KERNEL: Read from 0x4001: 5a (Expected: 5A)
# KERNEL: Read from 0x8002: 3c (Expected: 3C)
# KERNEL: Read from 0xC003: 7e (Expected: 7E)
# RUNTIME: Info: RUNTIME_0070 testbench.sv (83): $stop called.
```

Bug Detection

```
    // Memory bank selection (4 memory modules)
    wire [1:0] bank_sel = addr[14:13]; // Bug: using wrong bits for bank
        selection
    wire [1:0] bank_sel_read = addr[15:14];
    wire [13:0] local_addr = addr[13:0]; // Lower 14 bits for memory
       access
```

Incorrect Bank Selection Issue

The `bank_sel` signal in the composite memory module is responsible for selecting one of four memory banks based on the address. However, the selection is incorrectly based on the address bits `addr[14:13]`, which may not align with the intended bank selection logic.

Bank Selection Bug and Its Implications

The intended bank selection logic in the system relies on the upper two address bits, specifically `addr[15:14]`. However, due to a bug, the code is using `addr[14:13]` for determining the memory bank. This discrepancy has significant implications for the "Write and Read" test conducted on the composite memory module.

Correct Bank Selection Logic

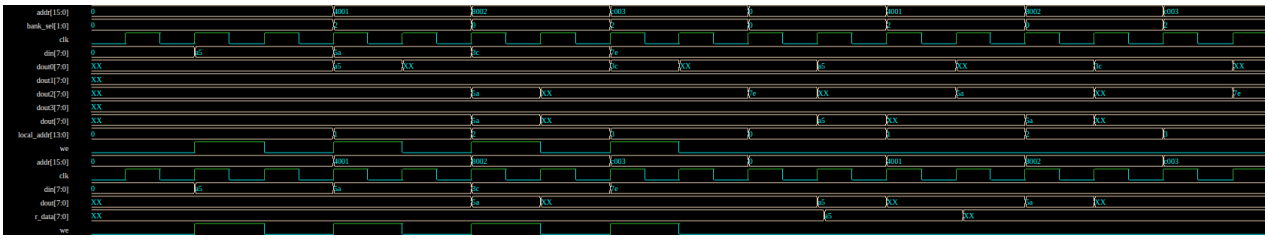
Under the correct bank selection logic, the upper two bits of the address (`addr[15:14]`) should dictate which memory bank is accessed. Each 16-bit address space is mapped to one of four banks as follows:

- 00 → Bank 0
- 01 → Bank 1
- 10 → Bank 2
- 11 → Bank 3

Impact of the Bug

Due to the use of `addr[14:13]` instead of `addr[15:14]`, the bank selection becomes misaligned. The effect on the test is as follows:

- The write operation `write_mem(16'h0000, 8'hA5)` correctly targets Bank 0, as the address 0x0000 has the upper 2 bits as 00.
- The write operation `write_mem(16'h4001, 8'h5A)` incorrectly targets Bank 3, as the address 0x4001 has the upper 2 bits as 10 but should have selected Bank 1.
- Similarly, the write operations `write_mem(16'h8002, 8'h3C)` and `write_mem(16'hC003, 8'h7E)` will mistakenly target Bank 0 and Bank 1, whereas they should access Bank 2 and Bank 3, respectively.



Effects on Read Operations

The incorrect bank selection also affects the reading process:

- During the read operation for `16'h4001`, data will be read from Bank 3 instead of the expected Bank 1.
- Likewise, for addresses `16'h8002` and `16'hC003`, data will be read from Bank 0 and Bank 1, despite them being intended for Banks 2 and 3.

```
# KERNEL: Writing and Reading Test
# KERNEL: Read from 0x0000: a5 (Expected: A5)
# KERNEL: Read from 0x4001: xx (Expected: 5A)
# KERNEL: Read from 0x8002: xx (Expected: 3C)
# KERNEL: Read from 0xC003: xx (Expected: 7E)
# RUNTIME: Info: RUNTIME_0070 testbench.sv (83): $stop called.
```

Conclusion

The issue arises from the incorrect mapping of address bits for bank selection, causing unintended memory banks to be accessed. As a result, this leads to data overwriting or incorrect data retrieval during read operations, causing test failures. The bug can be resolved by correcting the bank selection logic to use `addr[15:14]`, ensuring proper access to the intended memory banks.

EDAPLAYGROUND Link

You can view or run the Verilog code in EDAPLAYGROUND
Click here to visit EDAPLAYGROUND