MLab1

Modeling and simulate with Verilog HDL

In this part of the book we show how to build and use **Verilog HDL** language to model simple RISC-V architectures at RTL (Register Transfer Level) and how to copile and run these models on our RISC-V based boards.

Verilog HDL is widely used to model RISC-V architectures because it is a **hardware-centric language** with excellent tool support, **simulation capability**, and fine-grained control of **RTL** structures. It enables accurate modeling of pipelined, parallel architectures like RISC-V cores and fits seamlessly into ASIC/FPGA design flows.

Before the introduction of RISC-V models written in Verilog we provide an initial modeling lab to introduce the essential elements of Verilog HDL. We also show how to compile the hardware description into simulation model and how to run the model with the associated output information for graphic waveforms presentation.

Key Features of Verilog HDL

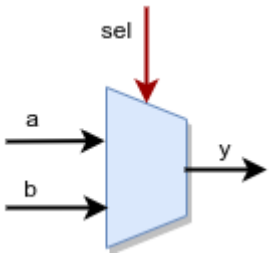
1. **Hardware Modeling**:
    - Verilog allows you to describe circuits at various abstraction levels:
        - **Gate level**: Describes the digital circuit using basic logic gates (AND, OR, etc.).
        - **Register-transfer level (RTL)**: Describes how data is transferred between registers and how operations are performed on data.
        - **Behavioral level**: Focuses on the behavior of the circuit, often resembling high-level programming.
2. **Concurrency**:
    - Unlike traditional programming languages that execute instructions sequentially, **Verilog** models the **concurrent nature of hardware**, where multiple parts of the circuit can operate simultaneously.
    -
3. **Simulation and Synthesis**:
    - **Simulation**: Verilog can be used to simulate how a digital circuit behaves before physically implementing it. You can create **testbenches** to test the functionality of your design.
    - **Synthesis**: Verilog descriptions can be synthesized into gate-level representations and eventually into hardware, such as FPGAs or ASICs.

4. **Modules**:
    - A Verilog design is composed of **modules**, which are **basic building blocks**. Each module can represent a component or an entire system. Modules have inputs, outputs, and internal logic that define the behavior of the system.

5. **Structural and Behavioral Descriptions**:
    - **Structural modeling**: Describes the hardware design by connecting components (e.g., gates or submodules) together.
    - **Behavioral modeling**: Describes the functionality of the system without worrying about how it is connected at a low level. It focuses on "what" the circuit does rather than "how" it's done.

Example Verilog Code

Simple 2-to-1 Multiplexer (MUX):




```
-------------------------------------------------------------------------------------------
module mux2to1 (
    input wire a,       // Input A
    input wire b,       // Input B
    input wire sel,     // Select line
    output wire y       // Output Y
);
assign y = (sel) ? b : a;   // If sel = 1, choose B; otherwise, choose A
endmodule
-------------------------------------------------------------------------------------------
```

In this example:

- A simple **2-to-1 multiplexer** is described in Verilog.
- **`sel`** is the select signal that controls which input (**`a` or `b`**) is passed **permanently** (**`assign`**) to the output **`y`**.

The following is testbench module:

`tb_mux2to1.vt`

for the above `mux2to1.v` module

```
-------------------------------------------------------------------------------------------
module tb_mux2to1;
    reg a, b, sel;
    wire y;

    mux2to1 uut (             // module instantiation
        .a(a),
        .b(b),
        .sel(sel),
        .y(y)
    );
    initial begin             // initial process
        // Test case 1: sel = 0, a = 0, b = 1
        a = 0; b = 1; sel = 0;
        #10; // Wait 10 time units
        $display("Test 1: y = %b", y);  // Should print y = 0
        // Test case 2: sel = 1, a = 0, b = 1
        sel = 1;
        #10;
        $display("Test 2: y = %b", y);  // Should print y = 1
        $finish; // End simulation
    end
endmodule
-------------------------------------------------------------------------------
```

In this **testbench**:

- The multiplexer is instantiated and tested with different input combinations.
- The sequential steps are separated by 10 time units (**`#10`**)
- The results (register contents) are prited with **`$display`** instruction

---

Installing compilation and simulation tools

1. Install Icarus Verilog:

Icarus Verilog (**`iverilog`**) is a Verilog **simulation** and **synthesis** tool.

- On **Linux (Ubuntu)**, you can install it using:

```
sudo apt-get install iverilog
```

2. Install GTKWave:

GTKWave is a **waveform viewer** to visualize simulation results.

```
sudo apt-get install gtkwave
```

To do

Analyze the presented modules **`uut`** (**unit under test**) and the test bench codes.

You can **compile** both modules together to get **simulation model** with **`iverilog`** and run the model with **`vvp`** command as presented below:

```
musepi@musepipro:~/RVLabs/RVMLabs/lab1$ iverilog mux2to1.v tb_mux2to1.v -o tb_mux2to1
musepi@musepipro:~/RVLabs/RVMLabs/lab1$ vvp tb_mux2to1
Test 1: y = 0
Test 2: y = 1
tb_mux2to1.v:20: $finish called at 20 (1s)
```

Note that **`iverilog`** can be installed with simple command

```
sudo apt install iverilog
```

1.1 Unit Under Test and the corresponding testbench

```
--------------------------------------------------------------------------------
// simple_module.v
module simple_module(input a, input b, output y);
    assign y = a & b;  // AND gate
endmodule
--------------------------------------------------------------------------------
```

Testbench file: `tb_simple_module.v` :

```
--------------------------------------------------------------------------------
// tb_simple_module.v
`timescale 1ns/1ps
module tb_simple_module;
    reg a;              // Inputs
    reg b;
    wire y;         // Output
    // Instantiate the design under test (DUT)
    simple_module dut (.a(a), .b(b), .y(y));
    // Testbench process
    initial begin
        // Initialize Inputs
        a = 0;
        b = 0;
        // Simulation sequence
        #10 a = 1;
        #10 b = 1;
        #10 a = 0;
        #10 b = 0;
        #10 a = 1; b = 1;
        #10;
        // End the simulation
        $finish;
    end
    // Dumping the waveform for GTKWave
    initial begin
        $dumpfile("tb_simple_module.vcd");  // Create the VCD file
        $dumpvars(0, tb_simple_module);     // Dump variables
    end
endmodule
--------------------------------------------------------------------------------
```

To do

Compile and **Simulate** with Icarus Verilog and GTKWave

Once you have your Verilog module and **testbench** ready, follow these steps to simulate:

Compile the Verilog files:

```
$ iverilog simple_module.v tb_simple_module.v -o tb_simple_module
```

This will compile the Verilog files and create an output binary named **`simple_tb`**.

Run the simulation:

```
vvp tb_simple_module
```

This command **runs** the compiled Verilog simulation model using **`vvp`**, the Icarus Verilog simulation runtime.

If the **testbench** has **waveform dumping enabled** (**`$dumpfile`** and **`$dumpvars`**), this will create a **`tb_simple_module.vcd`** file.

Use GTKWave:

- Once GTKWave opens, you should see the hierarchy of your Verilog module and signals in the **SST (Signal Selection Tree) panel on the left**.
- Select the signals (e.g., **`a, b,`** and **`y`**) and **drag them into the waveform panel** to visualize their behavior over time.
- The waveform display will show how the inputs (**`a`** and **`b`**) and output (**`y`**) change over time as per the simulation.

```
--------------------------------------------------------------------------------
$ iverilog simple_module.v tb_simple_module.v -o tb_simple_module
$ vvp tb_simple_module
VCD info: dumpfile tb_simple_module.vcd opened for output.
tb_simple_module.v:22: $finish called at 60000 (1ps)
musepi@musepipro:~/RVLabs/RVMLabs/lab1$ ls -l
```

```
total 28
-rw-rw-r-- 1 bako bako  245 11月 13 10:20 mux2to1.v
-rw-rw-r-- 1 bako bako  115 11月 13 10:31 simple_module.v
-rwxr-xr-x 1 bako bako 1875 11月 13 10:23 tb_mux2to1
-rw-rw-r-- 1 bako bako  523 11月 13 10:21 tb_mux2to1.v
-rwxr-xr-x 1 bako bako 2189 11月 13 10:39 tb_simple_module
-rw-rw-r-- 1 bako bako  726 11月 13 10:39 tb_simple_module.v
-rw-rw-r-- 1 bako bako  471 11月 13 10:39 tb_simple_module.vcd
--------------------------------------------------------------------------------
```

To do

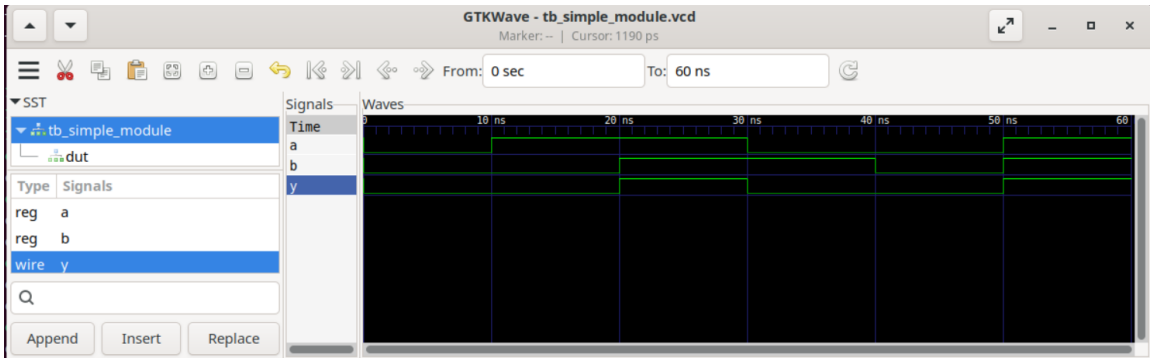
Analyze the presented modules **`uut`** (unit under test) and the test bench codes. You can compile both modules together to get simulation model with **`iverilog`** and run the model with **`vvp`** command .

```
$ iverilog simple_module.v tb_simple_module.v -o tb_simple_module
$ vvp tb_simple_module
```

Analyze the waveforms generated in **`tb_simple_module.vcd`**

```
gtkwave tb_simple_module.vcd
```

Fig 1.1



GTKWave diagram for simple **`simple_module.v`**

1.2 Simple ALU module and testbench

Below is a simple **32-bit ALU (Arithmetic Logic Unit)** model written in Verilog HDL. This ALU can perform basic arithmetic and logic operations on 32-bit inputs, such as addition, subtraction, AND, OR, XOR, and shift operations.

```
-----------------------------------------------------------------------------------------
ALU Model in Verilog HDL

module ALU(
    input [31:0] A,         // First 32-bit operand
    input [31:0] B,         // Second 32-bit operand
    input [3:0] ALUOp,      // 4-bit control signal to choose operation
    output reg [31:0] Result, // 32-bit output result
    output Zero             // Zero flag to indicate if result is zero
);

    always @(*) begin
        case (ALUOp)
            4'b0000: Result = A + B;          // Addition
            4'b0001: Result = A - B;          // Subtraction
            4'b0010: Result = A & B;          // Bitwise AND
            4'b0011: Result = A | B;          // Bitwise OR
            4'b0100: Result = A ^ B;          // Bitwise XOR
            4'b0101: Result = A << B[4:0];    // Logical shift left
            4'b0110: Result = A >> B[4:0];    // Logical shift right
            4'b0111: Result = $signed(A) >>> B[4:0]; // Arithmetic shift right
            default: Result = 32'b0;          // Default: Zero result
        endcase
    end
    // Zero flag: Set if result is zero
    assign Zero = (Result == 32'b0) ? 1'b1 : 1'b0;

endmodule
-----------------------------------------------------------------------------------------
```

Explanation

- **Inputs:**
  - **`A`** and **`B`**: The two 32-bit input operands.
  - **`ALUOp`**: A 4-bit control signal to select the operation. This control signal determines which operation the ALU will perform.
- **Outputs:**
  - **`Result`**: The 32-bit output result of the ALU operation.
  - **`Zero`**: A flag indicating if the result of the operation is zero (`1` if **`Result`** is **`zero`**, otherwise `0`).
- **Operations:**
  - **Addition (`ALUOp = 4'b0000`)**: Adds the two operands.
  - **Subtraction (`ALUOp = 4'b0001`)**: Subtracts the second operand from the first.
  - **Bitwise AND (`ALUOp = 4'b0010`)**: Performs a bitwise AND between the two operands.
  - **Bitwise OR (`ALUOp = 4'b0011`)**: Performs a bitwise OR between the two operands.
  - **Bitwise XOR (`ALUOp = 4'b0100`)**: Performs a bitwise XOR between the two operands.
  - **Logical Shift Left (`ALUOp = 4'b0101`)**: Shifts the first operand left by the number of positions specified in the lower 5 bits of the second operand.
  - **Logical Shift Right (`ALUOp = 4'b0110`)**: Shifts the first operand right logically by the number of positions specified in the lower 5 bits of the second operand.
  - **Arithmetic Shift Right (`ALUOp = 4'b0111`)**: Shifts the first operand right arithmetically (**preserving the sign bit**).
- **Zero Flag**: The **`Zero`** output flag is set to `1` if the result is zero, and `0` otherwise. This flag is typically **used in conditional branching to detect zero results**.

Example Simulation with testbench

You can simulate this **ALU** with a simple **testbench** to see the results for various operations.

```
-----------------------------------------------------------------------------------------
module tb_ALU; // Testbench module
    reg [31:0] A, B;
    reg [3:0] ALUOp;
    wire [31:0] Result;
    wire Zero;
    // Instantiate the ALU
    ALU uut (
        .A(A),
        .B(B),
        .ALUOp(ALUOp),
        .Result(Result),
```

```
        .Zero(Zero)
    );

    initial begin
        // Test addition
        A = 32'h00000005; B = 32'h00000003; ALUOp = 4'b0000;
        #10;
        $display("Addition: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test subtraction
        A = 32'h00000005; B = 32'h00000005; ALUOp = 4'b0001;
        #10;
        $display("Subtraction: A= %h, B= %h,Result= %h, Zero= %b", A, B, Result, Zero);
        // Test AND
        A = 32'h0000000F; B = 32'h000000F0; ALUOp = 4'b0010;
        #10;
        $display("AND: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test OR
        A = 32'h0000000F; B = 32'h000000F0; ALUOp = 4'b0011;
        #10;
        $display("OR: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test XOR
        A = 32'h0000000F; B = 32'h000000F0; ALUOp = 4'b0100;
        #10;
        $display("XOR: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test shift left
        A = 32'h00000001; B = 32'h00000004; ALUOp = 4'b0101;
        #10;
        $display("Shift Left: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test shift right
        A = 32'h00000010; B = 32'h00000002; ALUOp = 4'b0110;
        #10;
        $display("Shift Right: A= %h, B= %h, Result= %h, Zero = %b", A, B, Result, Zero);
        // Test arithmetic shift right
        A = 32'h80000000; B = 32'h00000002; ALUOp = 4'b0111;
        #10;
        $display("Arithmetic Shift Right:A=%h,B=%h,Result=%h,Zero= %b",A,B,Result,Zero);
        $finish;
    end
    initial begin
        $dumpfile("tb_ALU.vcd");  // Create the VCD
        $dumpvars(0, tb_ALU);     // Dump variables
    end
endmodule
```

Explanation of the Testbench

- The **testbench** drives different values to the inputs of the ALU and selects various operations using the **`ALUOp`** signal.
- The input values, the **result** and the **zero flag** are displayed for each test case using **`$display()`**.
- The testbench checks basic operations such as addition, subtraction, logical operations (AND, OR, XOR), and shift operations.

To do

Compile (**`iverilog`**) and execute (**`vvp`**):

```
$ vi ALU.v
$ vi tb_ALU.v
$ iverilog ALU.v tb_ALU.v -o tb_ALU
$ vvp tb_ALU
Addition: A = 00000005, B = 00000003, Result = 00000008, Zero = 0
Subtraction: A = 00000005, B = 00000005, Result = 00000000, Zero = 1
AND: A = 0000000f, B = 000000f0, Result = 00000000, Zero = 1
OR: A = 0000000f, B = 000000f0, Result = 000000ff, Zero = 0
XOR: A = 0000000f, B = 000000f0, Result = 000000ff, Zero = 0
Shift Left: A = 00000001, B = 00000004, Result = 00000010, Zero = 0
Shift Right: A = 00000010, B = 00000002, Result = 00000004, Zero = 0
Arithmetic Shift Right:A = 80000000,B = 00000002,Result = e0000000,Zero = 0
```

$gtkwave tb_ALU.vcd

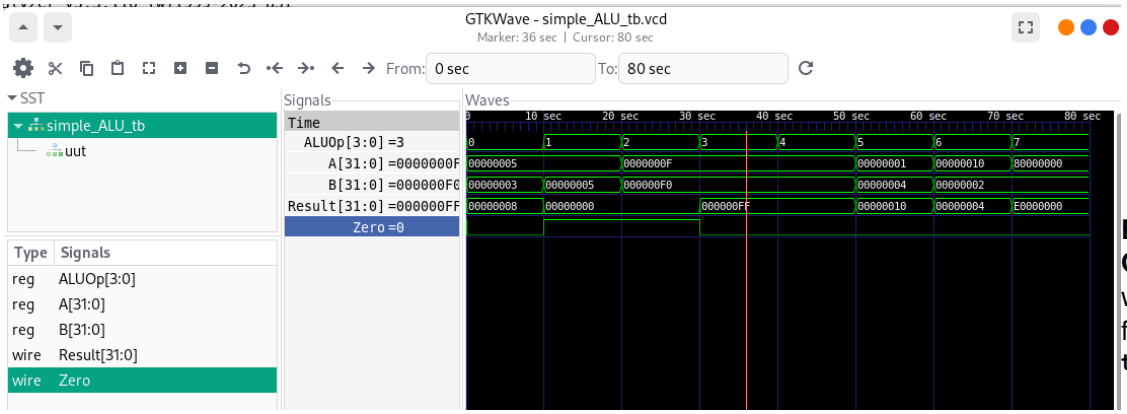


Fig 1.2 GTKWave waveforms for **tb_ALU.v**

1.3 Simple RAM module and testbench

The following is a simple **RAM memory model** written in **Verilog HDL**. This RAM model uses **32-bit words** and allows you to perform **read** and **write** operations.

```
module RAM (
    input clk,                    // Clock signal
    input we,                     // Write enable (1 for write, 0 for read)
    input [7:0] addr,             // 8-bit address (for 256 words)
    input [31:0] data_in,         // 32-bit input data for writing
    output reg [31:0] data_out    // 32-bit output data for reading
);

    // Declare the RAM memory (256 words of 32-bit data)
    reg [31:0] memory [255:0];
    // Read/Write logic
    always @(posedge clk) begin
        if (we) begin
            // Write operation: If write enable is high, store data_in at addr
            memory[addr] <= data_in;
        end else begin
            // Read operation: If write enable is low, output data from addr
            data_out <= memory[addr];
        end
    end

endmodule
```

Explanation of the Model

- **Inputs:**
  - **`clk`**: Clock signal used to synchronize the memory operations.
  - **`we`**: Write enable signal. When **`we`** is **high** (1), the memory performs a **write operation**. When **`we`** is **low** (0), the memory performs a **read operation**.
  - **`addr`**: The 8-bit address input, allowing the RAM to access up to 256 memory locations.
  - **`data_in:`** The 32-bit input data, used when writing data to memory.
- **Output:**
  - **`data_out`**: The 32-bit output data, used when reading data from memory.
- **Memory Array:**
  - The memory array `memory` is a **256-word** RAM, where each word is **32 bits wide**. This is modeled using **`reg[31:0] memory[255:0]`**.
- **Behavior:**
  - On the **positive edge of the clock**, the RAM performs either a write or a read operation depending on the value of the we signal.
  - If we is high, the **`data_in`** is written to the memory location specified by **`addr`**.
  - If we is low, the **`data_out`** will hold the value stored in the memory location specified by **`addr`**.

Testbench for the RAM Model

Below is a simple testbench that simulates read and write operations on the RAM model.

```
module tb_RAM;
    reg clk;                // Clock signal
    reg we;                 // Write enable signal
```

```
    reg [7:0] addr;         // 8-bit address for memory
    reg [31:0] data_in;     // 32-bit input data for write
    wire [31:0] data_out;   // 32-bit output data for read
    // Instantiate the RAM module
    RAM uut (
        .clk(clk),
        .we(we),
        .addr(addr),
        .data_in(data_in),
        .data_out(data_out)
    );
    // Clock generator: Toggle clk every 5 time units
    always #5 clk = ~clk;
    initial begin
        // Initialize signals
        clk = 0;
        we = 0;
        addr = 0;
        data_in = 0;
        // Write data to address 0
        #10;
        we = 1;
        addr = 8'h00;       // Address 0
        data_in = 32'hDEADBEEF; // Data to write
        #10;
        // Write data to address 1
        we = 1;
        addr = 8'h01;       // Address 1
        data_in = 32'hCAFEBABE; // Data to write
        #10;
        // Read data from address 0
        we = 0;
        addr = 8'h00;       // Address 0
        #10;
        $display("Read Address 0: Data = %h", data_out);
        // Read data from address 1
        we = 0;
        addr = 8'h01;       // Address 1
        #10;
        $display("Read Address 1: Data = %h", data_out);
        // Finish simulation
        $finish;
    end
    initial begin
        $dumpfile("tb_RAM.vcd");  // Create the VCD file
        $dumpvars(0, tb_RAM);     // Dump variables
    end
endmodule
```

Explanation of the Testbench

- The **testbench** initializes the clock (`clk`) and sets it to toggle every 5 time units.
- It performs the following operations:
    1. Writes `32'hDEADBEEF` to address `0x00`.
    2. Writes `32'hCAFEBABE` to address `0x01`.
    3. Reads from address `0x00` and prints the value to the console.
    4. Reads from address `0x01` and prints the value to the console.
- After these operations, the simulation stops.

Expected Output

The **testbench** will print the following output after simulating the read operations:

```
Read Address 0: Data = DEADBEEF
Read Address 1: Data = CAFEBABE
```

To do

Compilation and execution:

```
musepi@musepipro:~/Design/labgen$ iverilog RAM.v tb_RAM.v -o tb_RAM
musepi@musepipro:~/Design/labgen$ vvp tb_RAM
VCD info: dumpfile tb_RAM.vcd opened for output.
Read Address 0: Data = deadbeef
Read Address 1: Data = cafebabe
simple_RAM_tb.v:54: $finish called at 50 (1s)
gtkwave tb_RAM.vcd
```

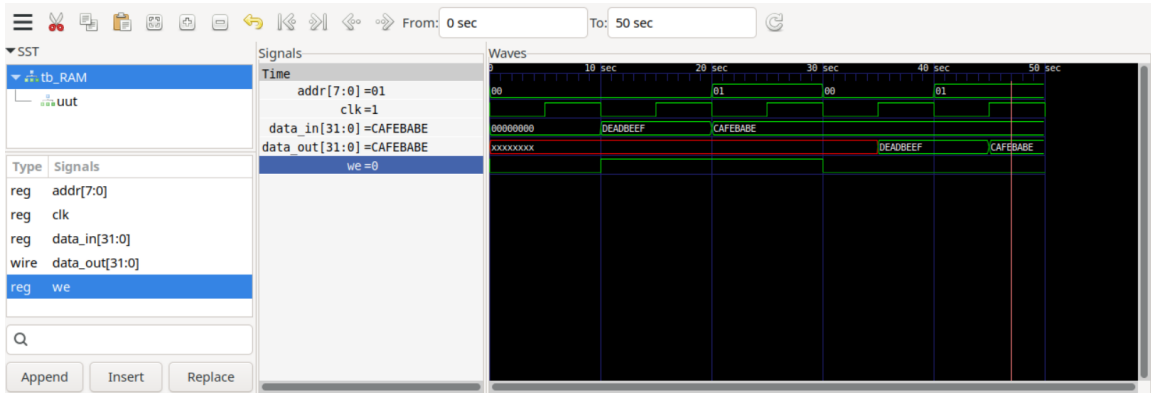


Fig 2.3 GTKWave diagram for simple **RAM** memory run.

Summary

This **32-bit RAM model** allows for read and write operations with a 32-bit data width and an 8-bit address space, allowing for up to 256 memory locations.

- The **testbench** demonstrates basic read and write operations on the RAM, showing how to interact with the memory and verify the results.
- You can extend or modify this RAM model for larger or more complex systems by adjusting the address width or adding additional control logic.