PROJECT REPORT

TEAM MEMBERS:

- Divyanshi Singh Bora (B20EE018)
- Bhavya Manish Sharma (B20EE013)

OBJECTIVE:

- Verilog development of a simple embedded system (CPU + Memory + Peripherals)
- To create a straightforward embedded system (consisting of a CPU, memory, and peripherals) using Verilog. This can be done either by writing the code for these modules from scratch or by sourcing them from online repositories. Our aim is to simulate the entire embedded system.

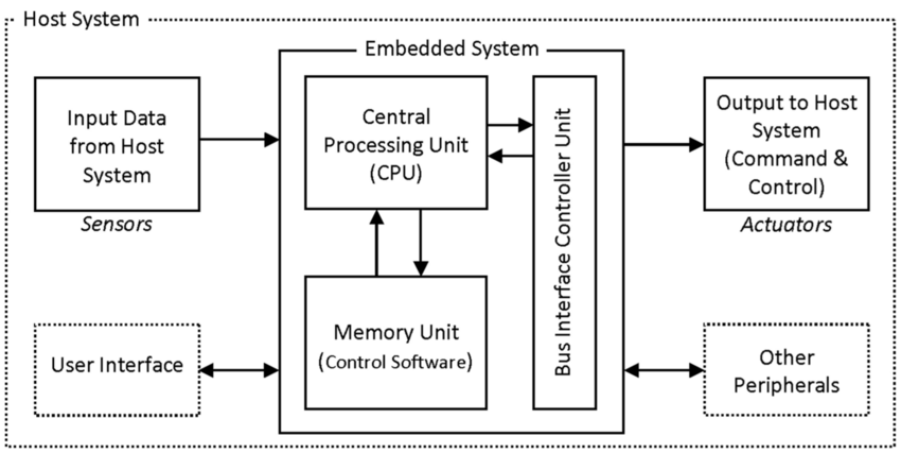
ABSTRACT:

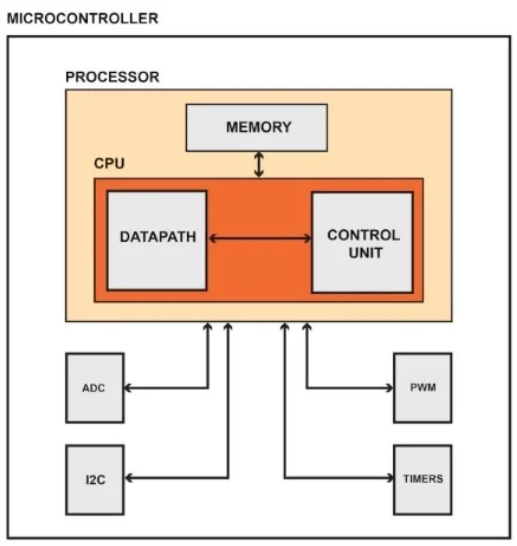
Embedded means something that is attached to another thing. An embedded system can be thought of as a computer hardware system having software embedded in it. An embedded system can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor-based system designed to perform a specific task.

Memory is responsible for storing program code and data that the system needs to operate. The amount and type of memory required for an embedded system depends on the system's complexity, processing requirements, and the size and complexity of the software running on the system.

Peripherals are a crucial component of an embedded system as they enable the system to interface with the external world. Peripherals are hardware devices that connect to the embedded system and provide additional functionality beyond the core system.

This project focuses on the Verilog development of a simple embedded system comprising a CPU, memory, and peripherals. The peripherals used in the project are UART and timer. A memory subsystem is then added to the system, including both instruction and data memory. The development process involves designing and testing individual components using testbenches, integrating them into the full system, and testing the system using simulation.





COMPONENTS USED:

- CPU
- memory
- peripherals
  - UART
  - timer

CODE:

- Top-level module

```
// Top-level module
module Top (
input clk,                      // input clock signal
input reset,                    // input reset signal
```

```
input [31:0] cpu_input1,        // input data for the first operand of CPU
input [31:0] cpu_input2,        // input data for the second operand of CPU
input [3:0] cpu_operation,      // input operation code for CPU
input [31:0] memory_address,    // input address for memory
input [31:0] memory_data_in,    // input data for memory to be written
input memory_write_enable,      // input write enable signal for memory
input [7:0] uart_data_in,       // input data for UART
input uart_tx_enable,           // input transmit enable signal for UART
input timer_enable,             // input enable signal for timer
output [31:0] cpu_result,       // output result of CPU operation
output [31:0] memory_data_out,  // output data read from memory
output [7:0] uart_data_out,     // output data to be transmitted by UART
output uart_tx_busy,            // output transmit busy signal for UART
output timer_trigger,           // output trigger signal for timer
output reg zero_flag            // output zero flag
);

// Instantiate CPU module
CPU cpu_inst (
.input1(cpu_input1),
.input2(cpu_input2),
.operation(cpu_operation),
.result(cpu_result)
);

// Instantiate Memory module
Memory memory_inst (
.address(memory_address),
.data_in(memory_data_in),
.write_enable(memory_write_enable),
.data_out(memory_data_out)
);

// Instantiate UART module
UART uart_inst (
.clk(clk),
.reset(reset),
.data_in(uart_data_in),
.tx_enable(uart_tx_enable),
.data_out(uart_data_out),
.tx_busy(uart_tx_busy)
);

// Instantiate Timer module
Timer timer_inst (
.clk(clk),
.reset(reset),
.enable(timer_enable),
.trigger(timer_trigger)
);

endmodule

// Comments:
// This module instantiates all the other modules and connects their inputs and outputs to the top-level inputs and outputs.
// It uses the dot notation to connect each input/output signal to the corresponding module port.
```

- CPU

```
// CPU module
module CPU (
  input [31:0] input1,                    //defines the input1 as a 32-bit signal
  input [31:0] input2,                    //defines the input2 as a 32-bit signal
  input [3:0] operation,                  //defines the input as a 4-bit signal
  output reg [31:0] result                //defines the output as a 32-bit signal which is register type
);
  always @(*) begin
    case (operation).                     //switch control which compares and sends the control to different cases
      4'b0000: result <= input1 + input2;   // Addition
      4'b0001: result <= input1 - input2;   // Subtraction
      4'b0010: result <= input1 & input2;   // Bitwise AND
      4'b0011: result <= input1 | input2;   // Bitwise OR
      4'b0100: result <= input1 ^ input2;   // Bitwise XOR
      default: result <= 0;                 // Default to 0
    endcase
  end
endmodule
```

- Memory

```
// Memory module
module Memory (
  input [31:0] address,             //input 32-bit signal in address to store the address
  input [31:0] data_in,             //input 32-bit signal in data_in to store the data input
  input write_enable,               //input single-bit signal in write_enable to write in data input if enable is 1
  output reg [31:0] data_out.       //output 32-bit signal in data_out to store the output data
);
  reg [31:0] memory [0:1023];       //defines a 32-bit register array called "memory" with 1024 locations indexed from 0 to 1023
                                    //represents the memory space of the module and stores the values written to it when "write_enable" is 1
```

```
  always @(*) begin
    if (write_enable) begin          //checks if the write_enable input signal is 1
      memory[address] <= data_in;    //writes the value of data_in to the memory array at the location given by address
    end
    data_out <= memory[address];     //sets the value of the data_out to value stored in the memory at the location given by address
  end
endmodule
```

- Timer

```
//Timer module
module Timer (
  input clk,                          // input clock signal
  input reset,                        // input reset signal
  input enable,                       // input enable signal
  output reg trigger                  // output trigger signal
);
  reg [31:0] count;                   // 32-bit register to keep track of the count value

  always @(posedge clk) begin         // always block triggered on positive edge of the clock signal
    if (reset) begin                  // if reset signal is high, reset the count and trigger signal
      count <= 0;
      trigger <= 0;
    end else begin                    // otherwise, execute the following statements
      if (enable) begin               // if enable signal is high, start counting
        count <= count + 1;           // increment the count value by 1
        if (count == 1000000) begin   // if the count reaches a million, trigger the trigger signal
          trigger <= 1;
          count <= 0;                 // reset the count value to zero
        end
      end else begin                  // if enable signal is low, reset the count and trigger signals
        trigger <= 0;
        count <= 0;
      end
    end
  end
endmodule
```

- UART

```
// UART module
module UART (
  input clk,                                // input clock signal
  input reset,                              // input reset signal
  input [7:0] data_in,                      // input data to be transmitted
  input tx_enable,                          // input signal to enable transmission
  output reg [7:0] data_out,                // output data received
  output reg tx_busy                        // output signal indicating if transmission is ongoing
);

  reg [7:0] tx_reg;                         // 8-bit register to store the data to be transmitted
  reg tx_empty, tx_shift_reg;               // 1-bit registers to indicate whether the shift register is empty and whether data is being shifted out


  always @(posedge clk) begin               // always block triggered on positive edge of the clock signal
    if (reset) begin                        // if reset signal is high, reset the transmit registers
      tx_empty <= 1;
      tx_shift_reg <= 0;
      tx_reg <= 0;
    end else begin                          // otherwise, execute the following statements
      if (tx_enable) begin                  // if transmission is enabled
        if (tx_empty) begin                 // if the shift register is empty
          tx_reg <= data_in;                // store the input data in the transmit register
          tx_shift_reg <= 1;                // set the shift register to shift the data out
          tx_empty <= 0;                    // reset the tx_empty signal
        end
      end else begin                        // if transmission is disabled
        if (!tx_empty) begin                // if the shift register is not empty
          tx_shift_reg <= 1;                // shift the remaining data out
        end
      end
    end
  end
end


always @(posedge clk) begin                 // always block triggered on positive edge of the clock signal
  if (tx_shift_reg) begin                   // if data is being shifted out
    if (tx_busy) begin                      // if transmission is ongoing
      tx_shift_reg <= 0;                    // stop shifting data out
      tx_empty <= 1;                        // set the tx_empty signal to indicate shift register is empty
    end else begin
      data_out <= tx_reg;                   // output the data in the transmit register
      tx_busy <= 1;                         // set the tx_busy signal to indicate transmission is ongoing
      tx_shift_reg <= 0;                    // stop shifting data out
    end
  end else begin
    data_out <= 0;                          // if no data is being shifted out, output zero
  end
```

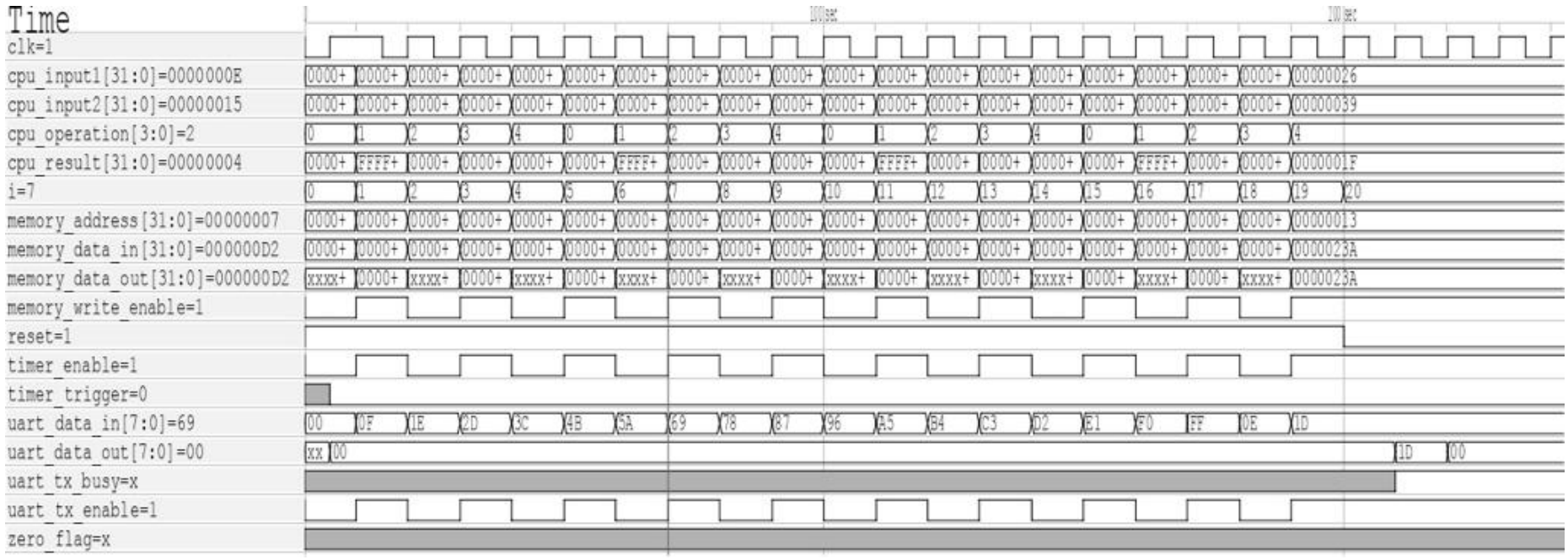
```
  end
endmodule
```

- Testbench

```
                                                    // Testbench module to test the Top-level module
module Testbench ();

                                                    // Declare inputs
reg clk, reset;
reg [31:0] cpu_input1, cpu_input2, memory_address, memory_data_in;
reg [3:0] cpu_operation;
reg memory_write_enable, uart_tx_enable, timer_enable;
reg [7:0] uart_data_in;

                                                    // Declare outputs
wire [31:0] cpu_result, memory_data_out;
wire [7:0] uart_data_out;
wire uart_tx_busy, timer_trigger;
wire zero_flag;

                                                    // Instantiate Top-level module
Top top_inst (
.clk(clk),
.reset(reset),
.cpu_input1(cpu_input1),
.cpu_input2(cpu_input2),
.cpu_operation(cpu_operation),
.memory_address(memory_address),
.memory_data_in(memory_data_in),
.memory_write_enable(memory_write_enable),
.uart_data_in(uart_data_in),
.uart_tx_enable(uart_tx_enable),
.timer_enable(timer_enable),
.cpu_result(cpu_result),
.memory_data_out(memory_data_out),
.uart_data_out(uart_data_out),
.uart_tx_busy(uart_tx_busy),
.timer_trigger(timer_trigger),
.zero_flag(zero_flag)
);

                                                    // Initialize and run simulation
integer i;
initial begin
// Set inputs for 20 clock cycles
for(i=0;i<20;i=i+1) begin
clk = 0+ 10i;
reset = 1;
cpu_input1 = 2i;
cpu_input2 = 0 + 3i;
cpu_operation = i%5;
memory_address = i;
memory_data_in = 30i;
memory_write_enable = i%2;
uart_data_in = 15*i;
uart_tx_enable = i%2;
timer_enable = i%2;
#10 reset = 0;
end

                                                    // Wait for simulation to complete
#100;
$finish;
end


                                                    // Toggle clock every 5 time units
always #5 clk = ~clk;


                                                    // Dump waveform to output.vcd file
initial begin
$dumpfile("output.vcd");
$dumpvars(1);
end

endmodule
```

SIMULATION:

- *Waveforms:*



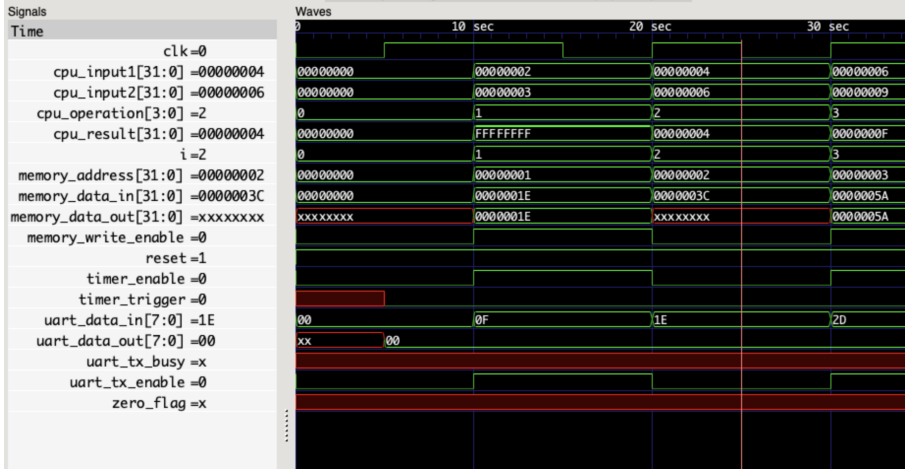


fig01: at t= 20sec

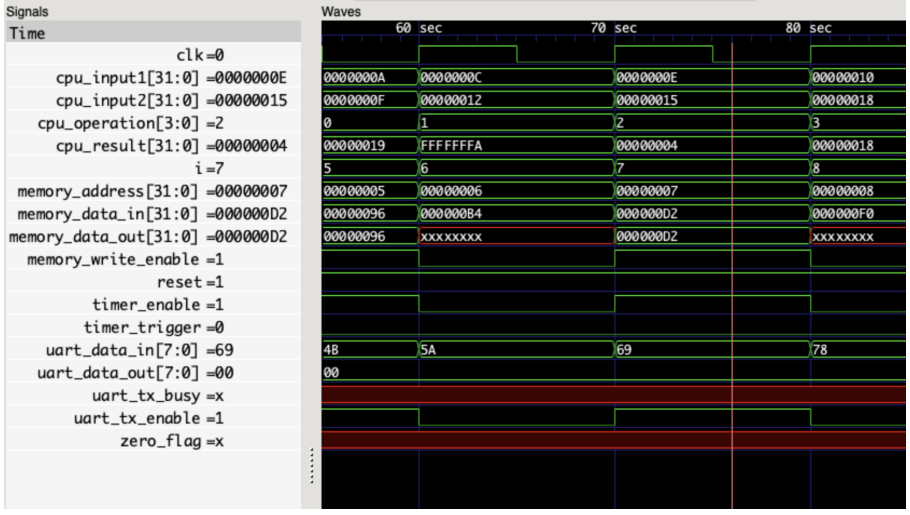
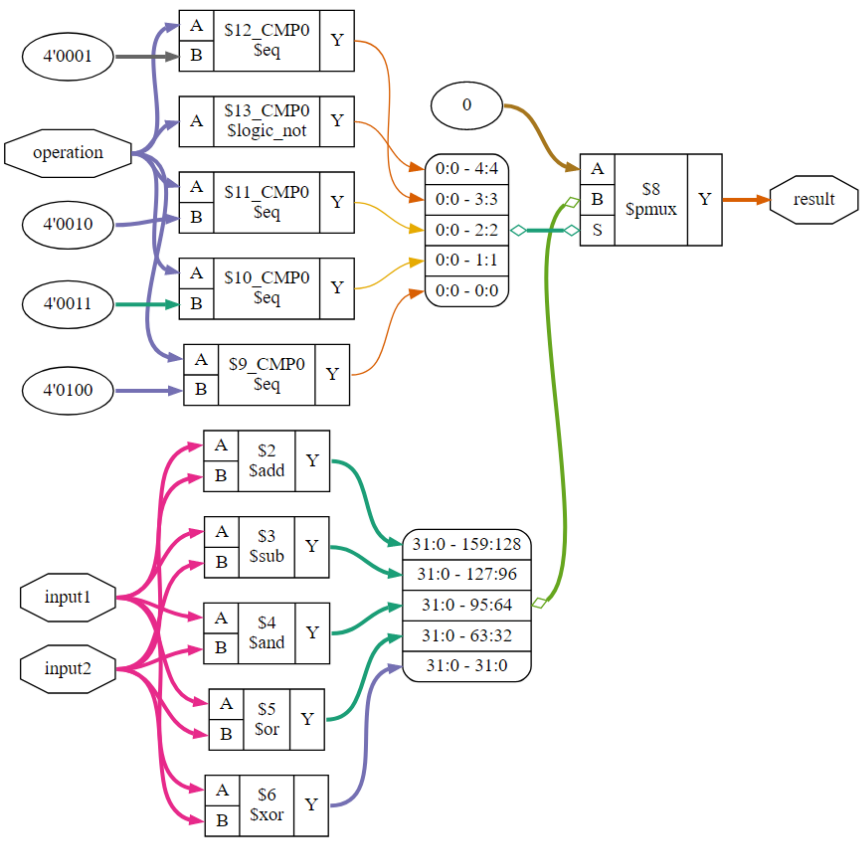


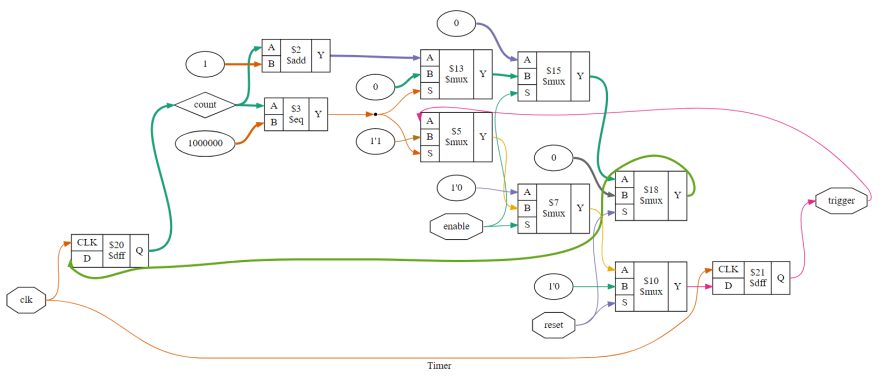
fig02: at t= 70sec

CPU block diagram:

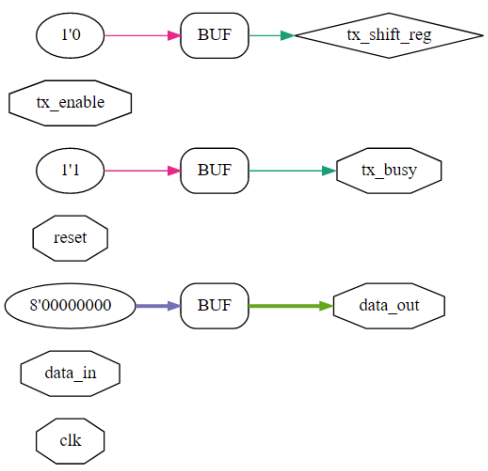


CPU

TIMER block diagram:



UART block diagram:



UART

DISCUSSION:

The code consists of six files. The first file top module instantiates the other four modules namely CPU, UART, timer and memory. The testbench is used to test the code by giving user input and checking the waveforms of the inputs and outputs. All the files were written in verilog.

The code was run on VS code. The requirements are Icarus verilog, gtkwave must be installed on the computer . Extension Verilog-HDL/SystemVerilog needs to installed on verilog.

In the terminal run the following commands:

```
$ iverilog -o out.vvp testbench.v cpu.v memory.v timer.v top.v uart.v
$ vvp out.vvp
$ gtkwave output.vcd
```

To visualize the peripherals EDAplayground was used where the specific for the peripheral module was added and the b;ock diagrams were obtained.

memory_data_out shows the memory used at that instant.

REFERENCES:

- https://opencores.org/
- https://verilogguide.readthedocs.io/en/latest/
- Quick Reference for Verilog HDL- Rajeev Madhavan AMBIT Design Systems, Inc.
- https://chat.openai.com/
- Verilog HDL Reference Manual- Version 1999.05, May 1999
- https://edaplayground.com/