ELEC473
Digital Systems Design

Assignment 3
MIPS Processor

Khanthapak Thaipakdee

Department of Electrical Engineering and Electronics,
University of Liverpool,
Brownlow Hill, Liverpool L69 3GJ, UK

Table of Contents

Contents **Page**

1. Introduction 1
2. Description of each module: Part A 1
   2.1 gpio 1
3. Results of Part A 2
4. Overview of Part B 4
5. Description of each module: Part B 6
   5.1 maindec 6
   5.2 aludec 8
   5.3 alu 10
   5.4 mux4 12
   5.5 zero_byte_extension 13
   5.6 mux2 13
   5.7 datapath 14
   5.8 controller 16
   5.9 mips 17
6. Results of Part B 18
7. Description of each module: Part C 22
   7.1 counter 23
   7.2 PWM 24
   7.3 MIPS_System 28
8. Results of Part C 29
Appendix A: insts_data.mif file of Part A 34
Appendix B: Verilog code of Part B 35
Appendix C: Verilog code of Part C 42

Table of Figures

Figure **Page**

Figure 1 Screenshot of 7 Segments Display on DE2 board 2
Figure 2 Signal Tap logic analyser screenshot (7 Segments Display) 3
Figure 3 Block diagram of Part B 5
Figure 4 maindec ASM chart 7
Figure 5 aludec ASM chart 9
Figure 6 alu ASM chart 11
Figure 7 mux4 ASM chart 12
Figure 8 mux4 RTL view 12
Figure 9 zero_byte_ext ASM chart 13
Figure 10 mux2 ASM chart 13
Figure 11 datapath RTL view 15
Figure 12 Signal Tap logic analyser screenshot (XOR and XORI instructions) 20
Figure 13 Signal Tap logic analyser screenshot (LBU instruction) 21
Figure 14 Block Diagram of PWM module. 22
Figure 15 8 bits counter ASM chart 23
Figure 16 PWM module ASM chart (left) Register ASM chart (right) Comparator ASM chart. 25
Figure 17 (a) PWM RTL view and (b) counter RTL view 26
Figure 18 (a) PWM simulation (0 – 1 μs) (b) PWM simulation (4.8 – 5.6 μs) 27
Figure 19 MIPS_System RTL view 28
Figure 20 Brightening and Dimming LED Instructions Flowchart 31
Figure 21 Signal Tap logic analyser Screenshot to test PWM unit
(Instructions from Table 7, Table 8) 32
Figure 22 Signal Tap logic analyser Screenshot for PWM unit testing
(Instructions from Table 9, Duration = sample 254 to 270) 33
Figure 23 Signal Tap logic analyser Screenshot for PWM unit testing
(Instructions from Table 9) 33

1. Introduction

In the assignment details provided involves designing a MIPS Processor Module, implementing digital systems using Verilog HDL, and understanding the internal operation of a MIPS processor. The assignment is divided into 3 parts: Part A focuses on writing a program to control the 7 Segments, Part B extends the processor to implement additional instructions, and Part C involves designing a Pulse Width Modulation (PWM) unit to drive an LED.

For Part A, the task is to modify the MIPS assembly language program to display the lowest 8 digits of the student's university ID on the DE2 board 7-segment display. This modification should be demonstrated with screenshots of the SignalTap Logic analyzer showing the instruction and program counter

2. Description of each module: Part A

2.1 gpio

The Verilog snippet writes output registers on the clock's positive edge, initializing them upon reset. CS_N and WR_N signals validate write operations, with the output registers (LEDR_R, LEDG_R, HEX0_R through HEX7_R) updating based on the address (Addr) and incoming DataIn.

The code (Table 1) initializes register $1 with 0xFFFF0000 using the "lui" instruction. Subsequent addiu" instructions incrementally add immediate values to register $0 and store the results in registers $2 through $6. These values are then written to specific memory addresses using "sw" instructions, updating the values stored in memory corresponding to HEX6_R, HEX5_R, HEX4_R, HEX2_R, and HEX0_R. This results in the board displaying hexadecimal values 0x79, 0x78, 0x12, 0x19, and 0x02, respectively.

```
// Register
// FFFF_202C   HEX7_R
// FFFF_2028   HEX6_R
// FFFF_2024   HEX5_R
// FFFF_2020   HEX4_R
// FFFF_201C   HEX3_R
// FFFF_2018   HEX2_R
// FFFF_2014   HEX1_R
// FFFF_2010   HEX0_R

//write output Register
always @(posedge clk)
    begin
        if(reset)
            begin
                LEDR_R[17:0] <=18'b0;
                LEDG_R[8:0]  <=9'h1FF;
                HEX0_R <= 7'b1000000;
                HEX1_R <= 7'b1000000;
                HEX2_R <= 7'b1000000;
                HEX3_R <= 7'b1000000;
                HEX4_R <= 7'b1000000;
                HEX5_R <= 7'b1000000;
                HEX6_R <= 7'b1000000;
                HEX7_R <= 7'b1000000;
            end

        else if(~CS_N && ~WR_N)
            begin
                if  (Addr[11:0] == 12'h008)
                    LEDR_R      <= DataIn;
                else if  (Addr[11:0] == 12'h00C)
                    LEDG_R      <= DataIn;
                else if  (Addr[11:0] == 12'h010)
                    HEX0_R      <= DataIn;
                else if  (Addr[11:0] == 12'h014)
                    HEX1_R      <= DataIn;
                else if  (Addr[11:0] == 12'h018)
                    HEX2_R      <= DataIn;
                else if  (Addr[11:0] == 12'h01C)
                    HEX3_R      <= DataIn;
                else if  (Addr[11:0] == 12'h020)
                    HEX4_R      <= DataIn;
                else if  (Addr[11:0] == 12'h024)
                    HEX5_R      <= DataIn;
                else if  (Addr[11:0] == 12'h028)
                    HEX6_R      <= DataIn;
                else if  (Addr[11:0] == 12'h02C)
                    HEX7_R      <= DataIn;
            end
    end
```

3. Results of Part A

The memory mapping indicates the locations of specific registers within the system. Register HEX7_R resides at memory address 0xFFFF202C, followed by HEX6_R at 0xFFFF2028, HEX5_R at 0xFFFF2024, and HEX4_R at 0xFFFF2020. Subsequently, HEX3_R is located at 0xFFFF201C, HEX2_R at 0xFFFF2018, HEX1_R at 0xFFFF2014, and HEX0_R at 0xFFFF2010.

According to the provided sequence of instructions, specific hexadecimal values will be displayed on the board's output registers. HEX6_R will showcase the hexadecimal value 0x79 (equals to 1), extracted from the third instruction. Subsequently, HEX5_R will exhibit the value 0x78 (equals to 7), sourced from the fifth instruction. HEX4_R will portray the value 0x12 (equals to 5), acquired from the seventh instruction. Moving forward, HEX2_R will showcase the value 0x19 (equals 4), retrieved from the ninth instruction. Lastly, HEX0_R will display the hexadecimal value 0x02 (equals 6), obtained from the eleventh instruction.

Table 1 Instruction set for displaying the last 8 digits of the student ID number.

	Code	Basic
1	0x3C01FFFF	lui $1, 0xFFFF
2	0x24020079	addiu $2, $0, 0x0079
3	0xAC222028	sw $2, 0x2028($1)
4	0x24030078	addiu $3, $0, 0x0078
5	0xAC232024	sw $3, 0x2024($1)
6	0x24040012	addiu $4, $0, 0x0012
7	0xAC242020	sw $4, 0x2020($1)
8	0x24050019	addiu $5, $0, 0x0019
9	0xAC252018	sw $5, 0x2018($1)
10	0x24060002	addiu $6, $0, 0x0002
11	0xAC262010	sw $6, 0x2010($1)
12	0x08000000B	lab1: j lab1

The logic analyser (Figure 2) illustrates the correct execution of instructions, as outlined in the instruction set provided in the insts_data.mif file (Table 1). The blue arrows indicate store word (“sw”) operations, signifying the storing of values from the HEX registers into memory locations associated with the 7-segment displays. Conversely, the green arrows indicate the desired values on the 7-segment displays, corresponding to the student ID. Notably, when writing to these memory addresses, the chip select (CS_N) and write enable (WR_N) signals are low, as indicated in the code snippet. Additionally, Figure 1 provides a screenshot of the 7-segment displays on the DE2 board, showcasing the correct and matching values being displayed. This overview confirms accurate instruction execution and proper 7-segment display functionality, ensuring correct student ID representation.

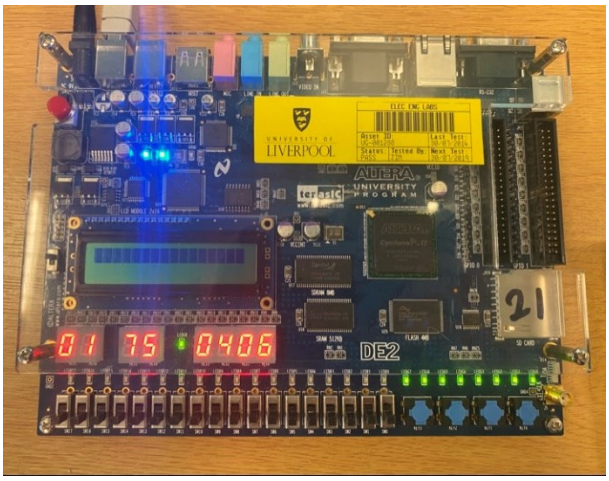

Figure 1 Screenshot of 7 Segments Display on DE2 board

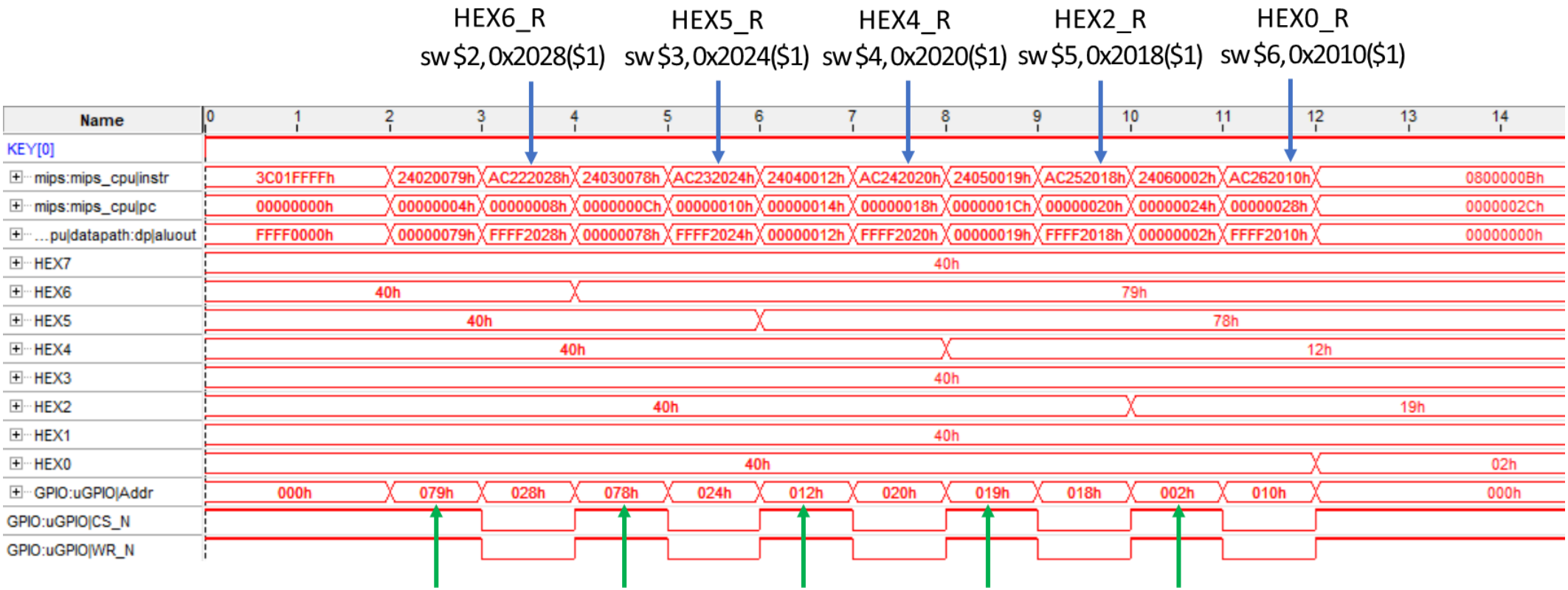

Figure 2 Signal Tap logic analyser screenshot
(7 Segments Display)

4. Overview of Part B

The MIPS design outlined in MIPS_System only supports a limited set of MIPS instructions, including ADD, ADDU, SUB, SUBU, AND, OR, and SLT within the R-Type instructions category. To incorporate the XOR instruction into the design, no additional hardware blocks are required since XOR operations can be accommodated within the existing ALU functionality. However, adjustments need to be made in the controller module's “aludec” and in the datapath's “alu” module. Specifically, the “aludec” module in the controller must be modified to recognize the XOR function, and the logic in the “alu” module within the datapath needs to be updated to handle XOR operations appropriately.

To integrate the XORI instruction effectively, adjustments are necessary in both the “maindec” and “aludec” modules. Firstly, within the “maindec” module responsible for opcode decoding, modifications are required to recognize the XORI opcode. In “maindec”, which generates control signals based on the opcode, the output should incorporate signals for “regwrite” and “alusrc”. These signals must be activated for the XORI instruction to ensure proper register writing and selection of input values for ALU operations. This guarantees the correct functionality of the XORI instruction within the system architecture. Furthermore, in the “aludec” module, the “aluop” values should align with XOR logic to properly coordinate with the XORI instruction's operation.

To modify LBU, two additional multiplexers are required. One multiplexer selects which byte from byte 0 to byte 3, and another multiplexer selects between the whole word or a byte. The signal to select which byte is used comes from aluresult [1:0], while the byte enable signal comes from the controller, which is the new signal that needs modification. Additionally, for unsigned integers, zero extension is necessary to transform an 8-bit value to a 32-bit value. The added blocks and signals are shown in Figure 2 (Red box).

Similar to the XORI instruction, the LBU instruction necessitates specific signals to be activated to ensure proper functionality. These signals include regwrite, alusrc, signext, memtoreg, and lb_en. Activation of these signals is crucial for enabling proper register writing and memory address register handling during the LBU operation. The lb_en signal specifically ensures that bytes are loaded instead of words. Additionally, for aluop, the value will be 000 to correspond to the ADD logic.

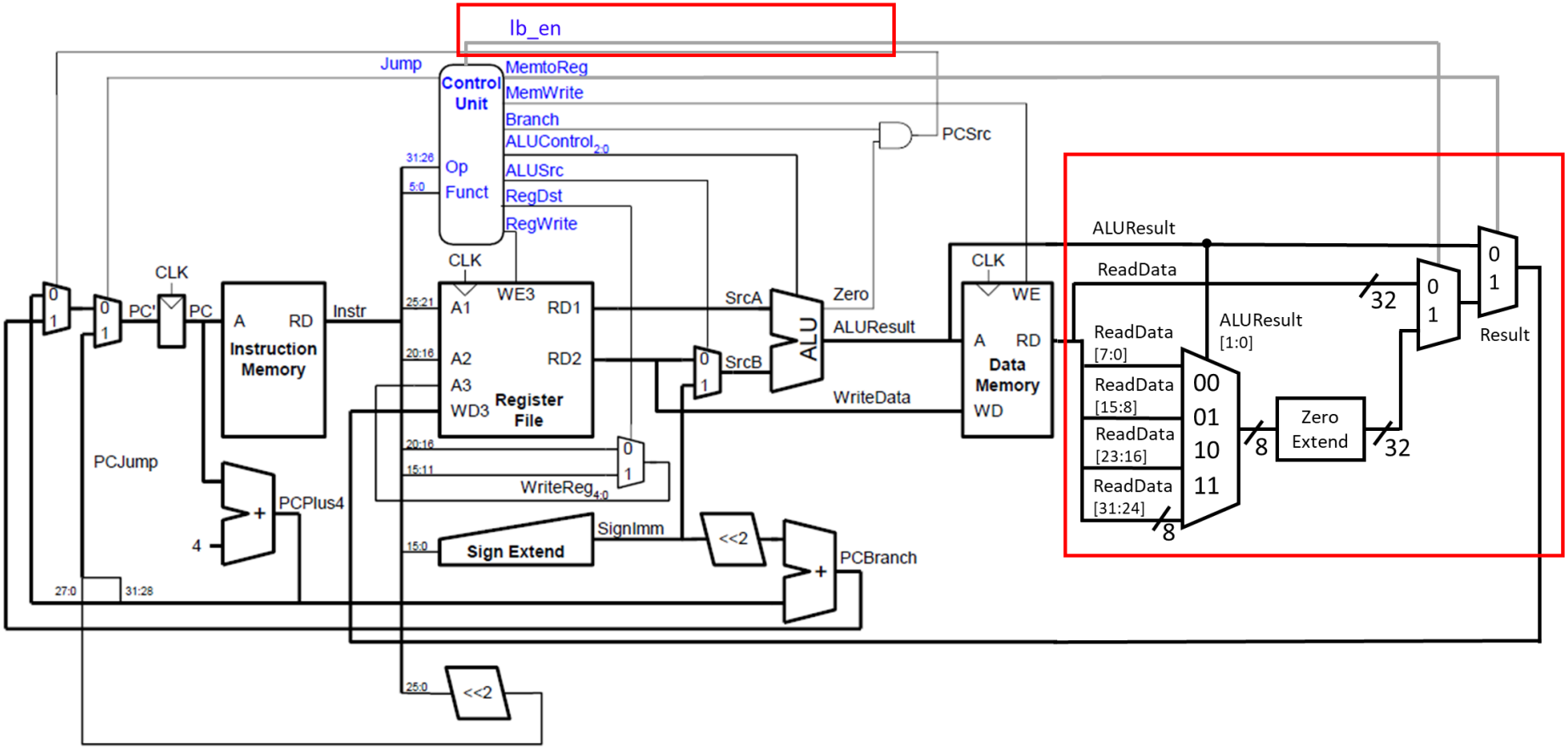

Figure 3 Block diagram of Part B

5. Description of each module: Part B

5.1 maindec

For the XOR instruction, its opcode remains the same as other R-Type instructions; hence, no modifications are necessary. However, the XORI instruction possesses a different opcode, thus requiring adjustments in the opcode decoder. Regarding the controls for XORI, they are similar to ORI (regwrite = 1, alusrc = 1) except for the "aluop" signal, as different ALU logic is utilized. However, since the previous 2-bit "aluop" is fully utilised, it is necessary to expand it from 2 bits to 3 bits. Table 2 displays the changes in "aluop".

The opcode decoder must be modified to recognize the distinct opcode associated with the LBU instruction. Additionally, according to the previous block diagram, the LBU instruction necessitates a new control signal called "lb_en", which determines whether the word or byte is used. However, the other control signals remain similar to those of LW, including signext, regwrite, alusrc, and memtoreg. Figure 4 visually illustrates the aluop ASM chart, decoding "op" into control signal outputs for different instructions

Table 2 Modification of aluop signal.

instructions	old aluop	new aluop
R-Type	11	011
LW	00	000
SW	00	000
BEQ	01	001
ADDI, ADDIU	00	000
ORI	10	010
LUI	00	000
J	00	000
XORI		100
LBU		000
Default	xxx	xxxx

```
module maindec (  input [5:0] op,
                  output      signext, lb_en,
                  output      shiftl16,
                  output      memtoreg, memwrite,
                  output      branch, alusrc,
                  output      regdst, regwrite,
                  output      jump,
                  output [2:0] aluop);
                  //modify from output [1:0] aluop

   reg [12:0] controls; //modify from reg [10:0] controls

   assign {  lb_en, signext, shiftl16, regwrite, regdst,
             alusrc, branch, memwrite,
             memtoreg, jump, aluop    } = controls;

   always @(*)
      case(op)
         // Rtype
         6'b000000: controls <= 13'b0001100000011;

         // LW
         6'b100011: controls <= 13'b0101010010000;

         // SW
         6'b101011: controls <= 13'b0100010100000;

         // BEQ
         6'b000100: controls <= 13'b0100001000001;

         // ADDI, ADDIU: only difference is exception
         6'b001000,
         6'b001001: controls <= 13'b0101010000000;

         // ORI
         6'b001101: controls <= 13'b0001010000010;

         // LUI
         6'b001111: controls <= 13'b0011010000000;

         // J
         6'b000010: controls <= 13'b0000000001000;

         // XORI
         6'b001110: controls <= 13'b0001010000100;

         // LBU
         6'b100100: controls <= 13'b1101010010000;

         // ???
         default:   controls <= 13'bxxxxxxxxxxxxx;
      endcase
endmodule
```

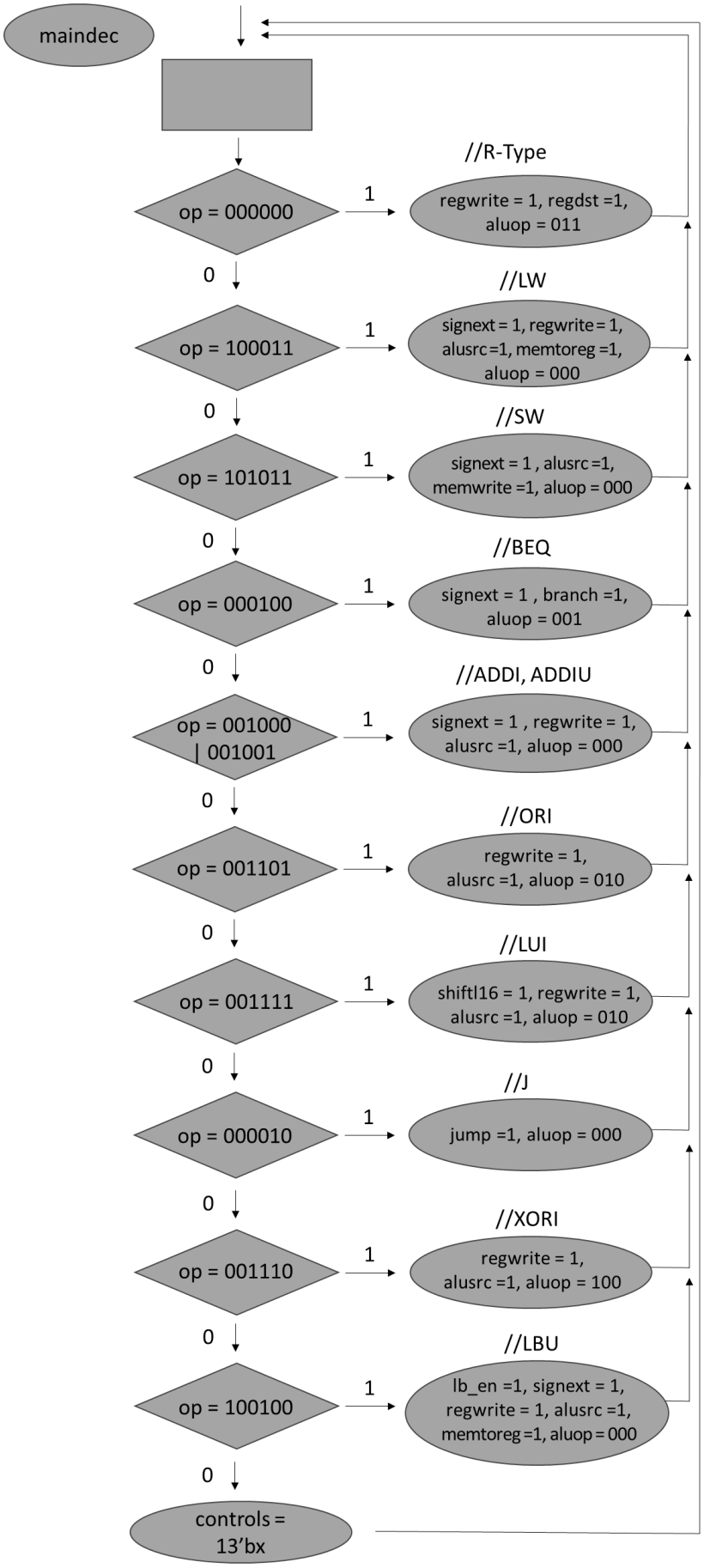

Figure 4 maindec ASM chart

5.2 aludec

The aludec module is responsible for decoding the alucontrol signal from aluop and the function of instructions. Since R-Type instructions share the same opcode, they will have the same aluop but different “funct” signals (function) that determine which ALU logic will be used for computation. However, I-Type instructions have different opcodes and may have the same or different aluop values depending on the previous module, so aluop will be used to determine the alucontrol signal. Additionally, the “alucontrol” signal in this module will correspond to the “alucont” signal in the alu module for the same logic operation. Table 3 represents the relationship between aluop, funct, and alucontrol. The logic of alucontrol will be explained in the alu module section. Figure 5 (aludec ASM chart) visually depicts the alucontrol output signal, illustrating the correlation between “aluop” and “funct” cases.

Table 3 Modification of alucontrol signal.

aluop	func	alucontrol
000		010 → 0010 //add
001		110 → 1010 //sub
010		001 → 0001 //or
100		0100 //xor
xxx (R-TYPE)	100000	010 → 0010 //ADD, ADDU
	100001	
	100010	110 → 1010 //SUB, SUBU
	100011	
	100100	000 → 0000 //AND
	100101	001 → 0001 //OR
	101010	111 → 1011 //SLT
	100110	0100 //XOR
Default		xxxx

```
module aludec (input [5:0] funct,
               input [2:0] aluop,
               //modify from [1:0] aluop
               output reg [3:0] alucontrol);
               //modify from [2:0] alucontrol

    always @(*)
        case(aluop)
            3'b000: alucontrol <= 4'b0010;  // add
            3'b001: alucontrol <= 4'b1010;  // sub
            3'b010: alucontrol <= 4'b0001;  // or
            3'b100: alucontrol <= 4'b0100;  // xor
            default: case(funct)          // RTYPE
                // ADD, ADDU
                6'b100000,
                6'b100001: alucontrol <= 4'b0010;

                // SUB, SUBU
                6'b100010,
                6'b100011: alucontrol <= 4'b1010;

                // AND
                6'b100100: alucontrol <= 4'b0000;

                // OR
                6'b100101: alucontrol <= 4'b0001;

                // SLT
                6'b101010: alucontrol <= 4'b1011;

                // XOR
                6'b100110: alucontrol <= 4'b0100;

                // ???
                default:      alucontrol <= 4'bxxxx;
            endcase
        endcase
endmodule
```

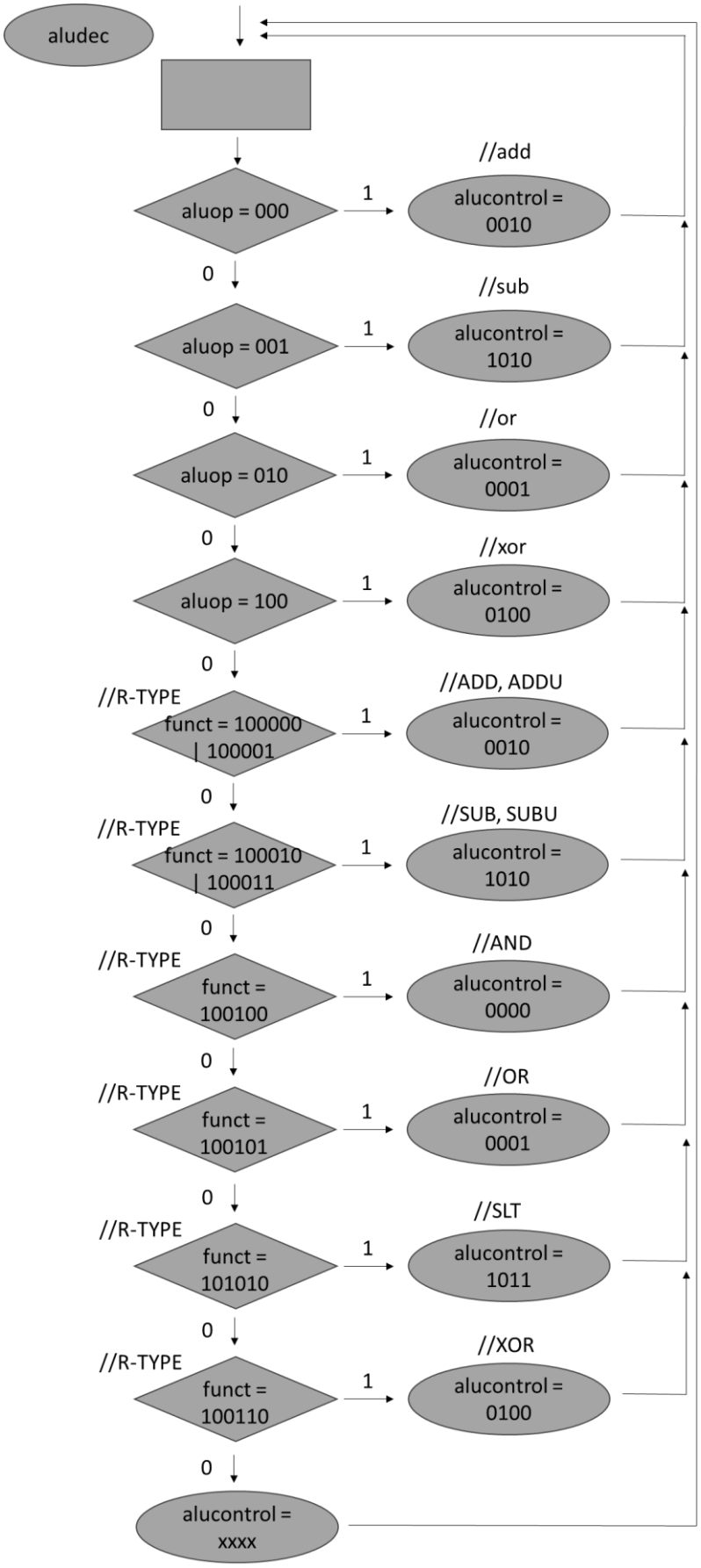

Figure 5 aludec ASM chart

5.3 alu

The previous “alucont” signal is a 3-bit signal, with the most significant bit (MSB) determining the use of the negative b. For the other 2 bits, “00” represents AND logic, “01” represents OR logic, “10” represents addition (SUM) logic, and “11” represents SLT logic.

To incorporate XOR logic, we extend the “alucont” signal from 3 bits to 4 bits. The new XOR case is represented by “100”, with the most significant bit set to “0” since XOR doesn't involve negating “b”. Furthermore, the existing cases are expanded from 3 bits to 4 bits as outlined in Table 4. The code snippet states the logic for determining the result output based on the alucont [2:0] signal and 32-bit inputs a and b. For addition (ADD), signalled by alucont value 4'b0010, result is the sum of a and b, with b3 retaining b. Conversely, for subtraction (SUB), indicated by alucont value 4'b1010, result is a minus b, facilitated by negating b and assigning it to b3 for two's complement arithmetic.

The ASM chart (Figure 6) illustrates the operation for two input signals to produce a result. Specifically, when alucont [2:0] equals 010, representing the case for addition (a+b) or subtraction (a-b), the behavior changes based on the most significant bit (MSB) of alucont. If the MSB is high, indicating a negative operation, the result will be a-b. Conversely, when alucont[3] is 0, the result will be a+b. This distinction in behavior based on the value of alucont[3] ensures proper handling of addition and subtraction operations within the ASM chart.

Furthermore, certain logic cases, like when alucont equals 1000 (representing a & ~b), remain available within the design. However, they are not utilized since no instructions in the design require this specific logic operation.

Moreover, in the “maindec” and “aludec” modules, there's a significant similarity between the I-Type instruction ADDI and the R-Type instruction ADD, despite their distinct opcodes. Both generate the same “alucont” signal, “010”. Similarly, for XOR and XORI instructions, despite their differing opcodes, they produce the same “alucont” signal, “100”.

```
module alu (input [31:0] a, b,
            input [3:0] alucont,
            // modify from input [2:0] alucont,
            output reg [31:0] result,
            output zero);

    wire [31:0] b3, sum, slt; // modify b3

    assign b3 = alucont[3] ? ~b:b; // negative b
    assign sum = a + b3 + alucont[3]; // a+b / a-b
    assign slt = sum[31];

    always@(*)
        case(alucont[2:0])
            3'b000: result <= a & b;
            3'b001: result <= a | b;
            3'b010: result <= sum;
            3'b011: result <= slt;
            3'b100: result <= a ^ b; //xor
            default: result <= 32'bx;
        endcase

    assign zero = (result == 32'b0);

endmodule
```

Table 4 ALU result logic

old alucont	new alucont	ALU logic
0 00	0 000	// a & b
0 01	0 001	// a | b
0 10	0 010	// a + b
1 10	1 010	// a - b
1 11	1 011	// SLT
	0 100	// a ^ b
Default		32’bx

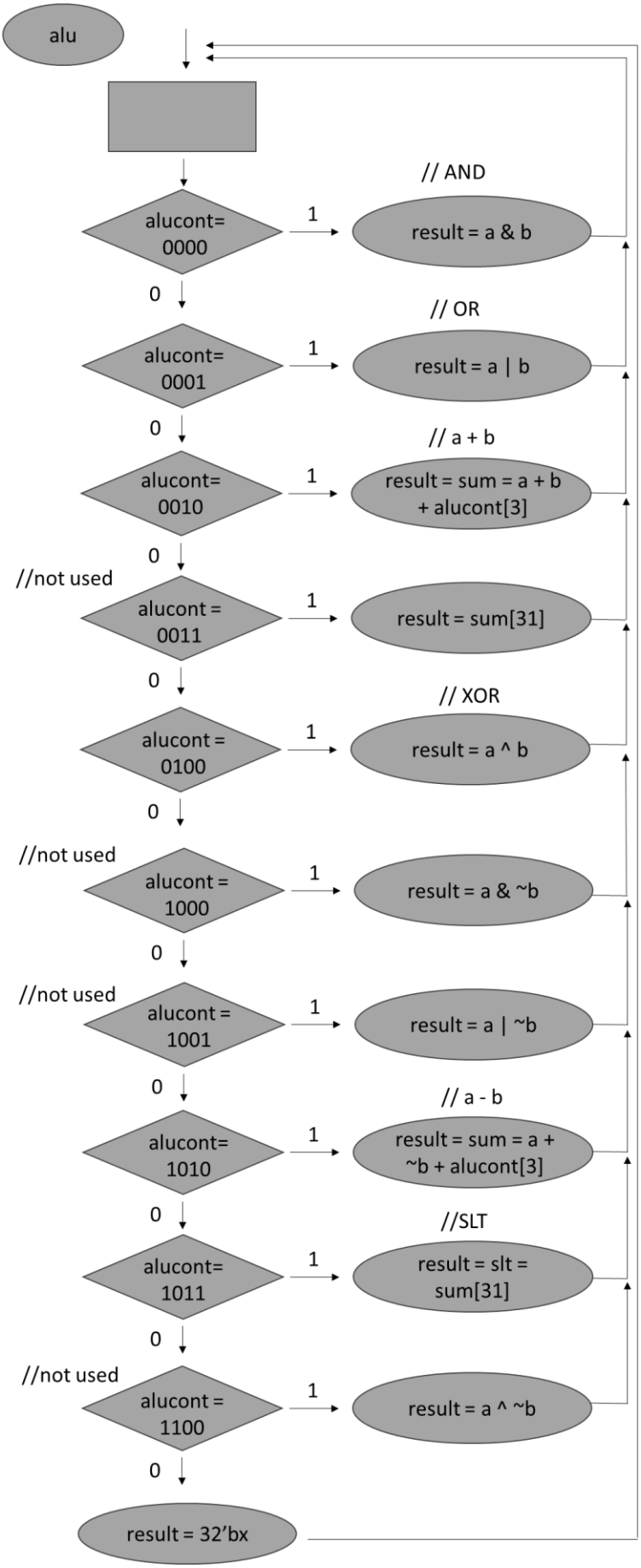

Figure 6 alu ASM chart

5.4 mux4

The “mux4” module is designed as a 32-bit to 8-bit converter, where it selects which byte is used based on the "s" signal (select). When s = 0, the output y corresponds to byte 0; when s = 1, y corresponds to byte 1; when s = 2, y corresponds to byte 2; and when s = 3, y corresponds to byte 3. The ASM chart of the mux4 module, depicted in Figure 7, shows “d0”, “d1”, “d2”, and “d3” representing different bytes of the input, while “y” refers to the output. Additionally, Figure 8 illustrates the RTL view of the “mux4” module, ensuring that there are no latches or flip-flops in the design.

```
mux4 #(8) selectbytemux(    .d0 (readdata[7:0]),
                            .d1 (readdata[15:8]),
                            .d2 (readdata[23:16]),
                            .d3 (readdata[31:24]),
                            .s  (aluout[1:0]),
                            .y  (lb_data));
```

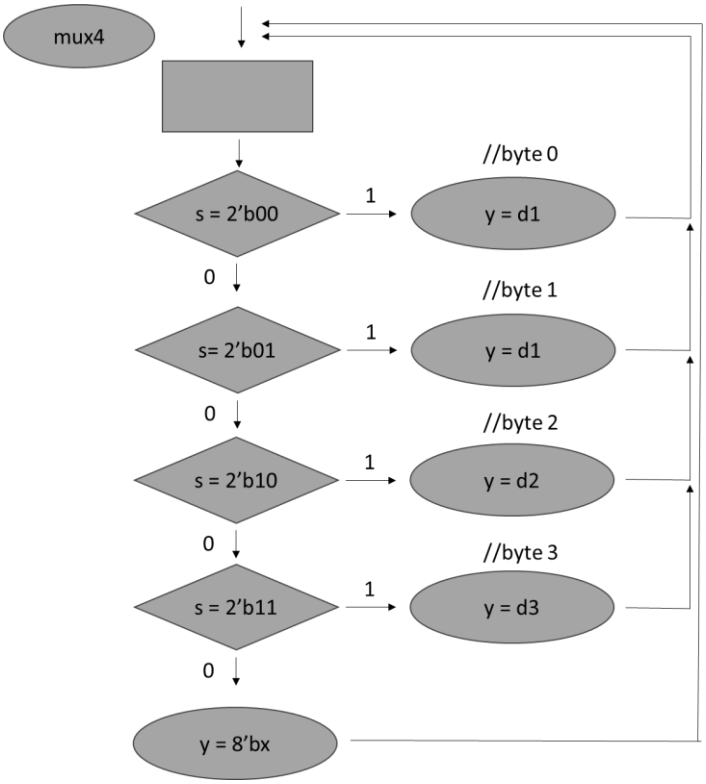

Figure 7 mux4 ASM chart

```
module mux4 # (parameter WIDTH = 8)
                (input  [WIDTH-1:0] d0, d1, d2, d3,
                input  [1:0] s,
                output [WIDTH-1:0] y);

    reg [WIDTH-1:0] mux_int;

    assign y =  (s == 2'b00) ? d0:
                (s == 2'b01) ? d1:
                (s == 2'b10) ? d2:
                (s == 2'b11) ? d3:
                8'bx;

endmodule
```

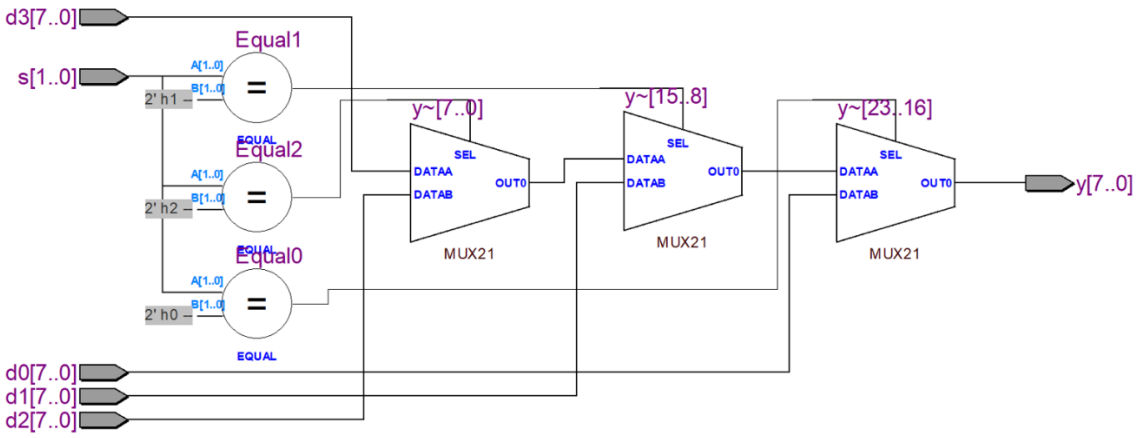

Figure 8 mux4 RTL view

5.5 zero_byte_extension

The “zero_byte_ext” module extends an 8-bit input to a 32-bit output by adding 24 zeros in front of the input. This module does not require any control signal because, for the LBU instruction, regardless of the selected byte, the input needs to pass through this module to ensure a 32-bit output. In the ASM chart of this module illustrated in Figure 9, “a” represents the input, while “y” refers to the output.

```
module zero_byte_ext(input [7:0] a,
                     output reg [31:0] y);

  always @(*)
       y <= {24'b0, a[7:0]};

endmodule
```

```
zero_byte_ext  byteex(.a (lb_data),
                      .y  (lb_data_ext));
```

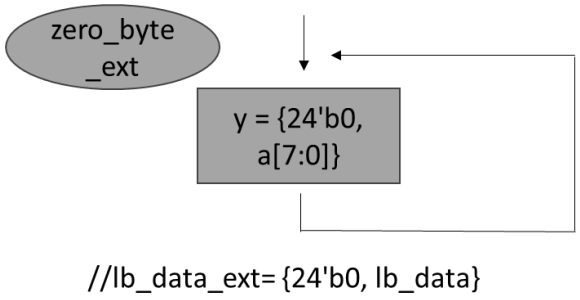

Figure 9 zero_byte_ext ASM chart

5.6 mux2

The mux2 module functions as a 2-to-1 multiplexer with customizable data width, handling two input signals (d0 and d1) and a select signal (s). Based on the select signal, it outputs either d1 or d0. Utilized in the datapath for lbumux, it selects between word (“readdata”) and byte (“lb_data_ext”) inputs depending on “lb_en”, with the output labeled “lw_lb_data” in the ASM chart (Figure 10).

```
module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
             input            s,
             output [WIDTH-1:0] y);

    assign y = s ? d1 : d0;

endmodule
```

```
mux2 #(32)  lbumux(  .d0 (readdata),
                     .d1 (lb_data_ext),
                     .s   (lb_en),
                     .y  (lw_lb_data));
```

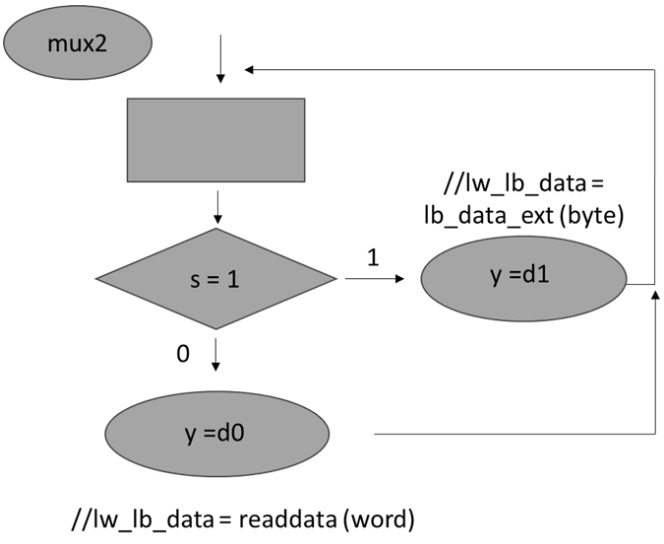

Figure 10 mux2 ASM chart

5.7 datapath

The datapath module acts as the central hub connecting the alu, regfile, mux2, mux4, and zero_byte_ext modules, enabling seamless data transmission between each component. In the modified design of Part B, two additional multiplexers are needed: “selectbytemux” (mux4 #(8)), which chooses between bytes 0 to 3, and “lbumux” (mux2 #(32)), determining whether to use the whole word or a byte.

As depicted in Figure 11, the readdata input enters the datapath module and is directed to selectbytemux. In selectbytemux, readdata[7:0] corresponds to byte 0, readdata[15:8] corresponds to byte 1, readdata[23:16] corresponds to byte 2, and readdata[31:24] corresponds to byte 3, as observed in the code snippet. Additionally, the select signal (“s”) is derived from aluout[1:0], representing the user-input instruction.

A wire named lb_data is created to connect the output from selectbytemux as an input to the byteex module. In the byteex module, lb_data, representing an 8-bit data (representing a byte of the word), is extended to a 32-bit output named lb_data_ext. This lb_data_ext is then directed to lbumux, where the selection between the whole word (readdata) or a byte (lb_data_ext) is determined by the select signal (lb_en). Furthermore, the output, lw_lb_data (referring to word or byte), is sent to resmux for transmission to the regfile module.

It is essential to clarify the inputs “lb_en” and “[3:0] alucontrol” in the modification of the code. “lb_en” is crucial for the LBU instruction, determining whether to select the whole word or a byte. On the other hand, the 4-bit “alucontrol” is critical for both XOR and XORI instructions, determining the specific ALU operation to perform.

```
module datapath (  input  clk, reset,
                   input  signext, lb_en, // add lb_en
                   input  shiftl16,
                   input  memtoreg, pcsrc,
                   input  alusrc, regdst,
                   input  regwrite, jump,
                // modify from input [2:0] alucontrol
                   input [3:0] alucontrol,
                   input [31:0] instr,
                   input [31:0] readdata,
                   output zero,
                   output [31:0] pc,
                   output [31:0] aluout, writedata);

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh, shiftedimm;
  wire [31:0] srca, srcb;
  wire [31:0] lw_lb_data, lb_data_ext;  // added
  wire [7:0]  lb_data;
  wire [31:0] result;
  wire        shift;

    // register file logic
      regfile     rf (     .clk     (clk),
                           .we      (regwrite),
                           .ra1     (instr[25:21]),
                           .ra2     (instr[20:16]),
                           .wa      (writereg),
                           .wd      (result),
                           .rd1     (srca),
                           .rd2     (writedata));

      mux2 #(32)  resmux(  .d0 (aluout),
                           .d1 (lw_lb_data),
                           .s  (memtoreg),
                           .y  (result));
      // LBU
      mux2 #(32)  lbumux(  .d0 (readdata),
                           .d1 (lb_data_ext),
                           .s  (lb_en),
                           .y  (lw_lb_data));

      mux4 #(8) selectbytemux(.d0 (readdata[7:0]),
                              .d1 (readdata[15:8]),
                              .d2 (readdata[23:16]),
                              .d3 (readdata[31:24]),
                              .s  (aluout[1:0]),
                              .y  (lb_data));

      zero_byte_ext  byteex(.a (lb_data),
                           .y  (lb_data_ext));
```

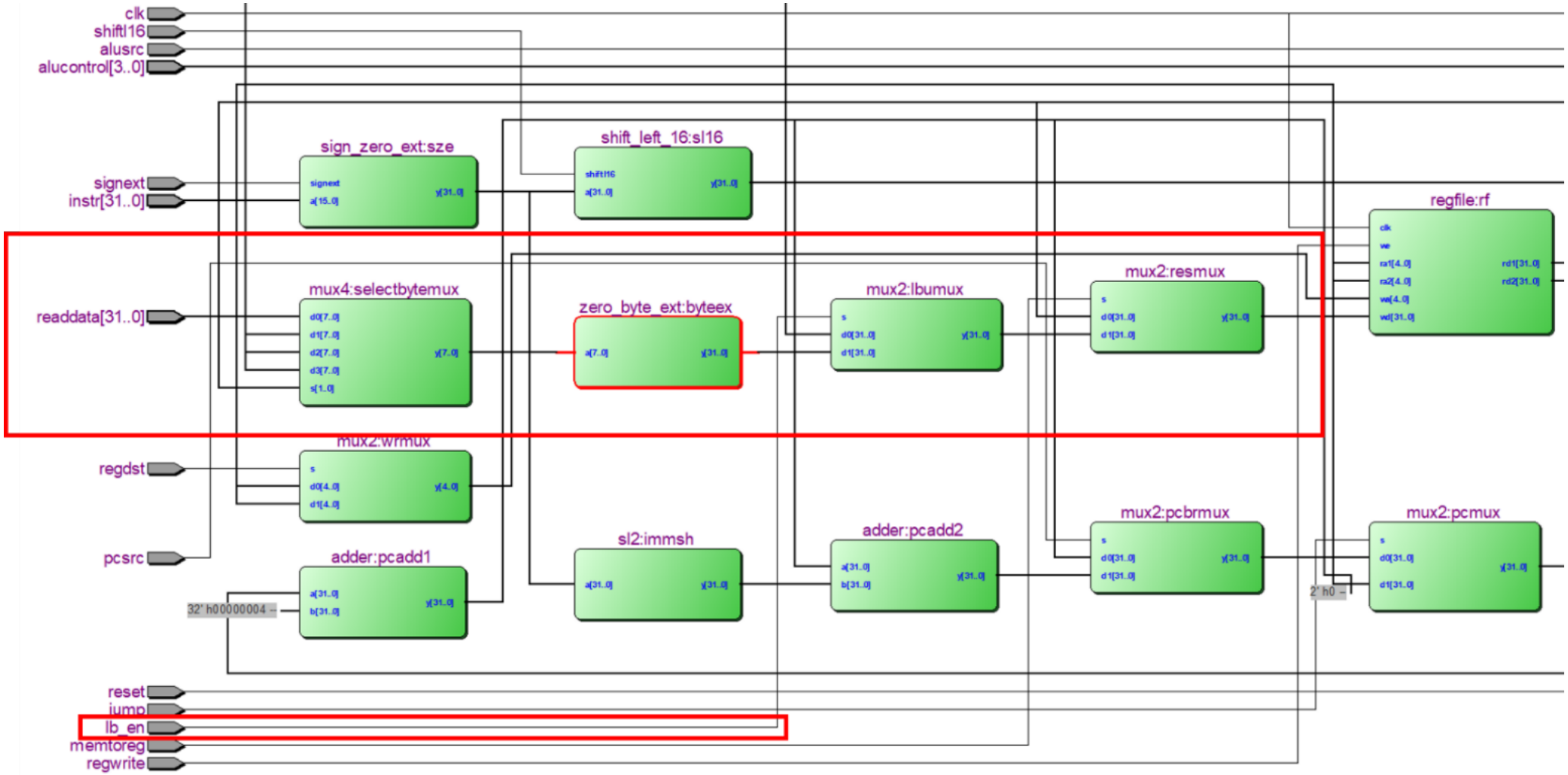

Figure 11 datapath RTL view

5.8 controller

The controller module serves as the controller unit for a processor. It takes opcode (op), function code (funct) as inputs and generates various control signals required for instruction execution.

The module includes a “maindec” submodule to decode opcode and generate control signals based on the instruction type. Additionally, it includes an “aludec” submodule to decode function code and generate ALU control signals.

In the previously mentioned modification, the “alucontrol” signal is changed from 3 bits to 4 bits for the XORI instruction, and the “aluop” signal is changed from 2 bits to 3 bits for XOR logic in the ALU unit. Additionally, the “lb_en” signal is added for the LBU instruction, as observed from the code snippet.

```
module controller (input  [5:0] op, funct,
                   input        zero,
                   output       signext, lb_en,
                   output       shiftl16,
                   output       memtoreg, memwrite,
                   output       pcsrc, alusrc,
                   output       regdst, regwrite,
                   output       jump,
      //modify from output [2:0] alucontrol
                   output [3:0] alucontrol);


    wire [2:0] aluop; //modify from wire [1:0] aluop
     wire       branch`;

    maindec md(.op    (op),
               .lb_en     (lb_en), // add lb_en
               .signext   (signext),
               .shiftl16  (shiftl16),
               .memtoreg  (memtoreg),
               .memwrite  (memwrite),
               .branch    (branch),
               .alusrc    (alusrc),
               .regdst    (regdst),
               .regwrite  (regwrite),
               .jump      (jump),
               .aluop     (aluop));

    aludec  ad (  .funct      (funct),
               .aluop      (aluop),
               .alucontrol (alucontrol));

    assign pcsrc = branch & zero;

endmodule
```

5.9 mips

The Verilog code represents a MIPS processor module named “mips” comprising a controller and datapath. The controller decodes opcodes and function codes to generate control signals, including alucontrol, lb_en, and various others. Additionally, the datapath processes instructions based on the received control signals, including alucontrol, lb_en, and others, and performs data operations accordingly. Modifications include changing the width of “alucontrol” from 3 bits to 4 bits and adding a new input “lb_en” to both the controller and datapath modules to accommodate the XOR, XORI, and LBU instructions.

```
module mips (  input              clk, reset,
               input  [31:0]      instr,
               input  [31:0]      memreaddata,
               output             memwrite,
               output [31:0]      memaddr,
               output [31:0]      memwritedata,
               output [31:0]      pc);

   wire       signext, shiftl16, memtoreg, branch;
   wire       pcsrc, zero;
   wire       alusrc, regdst, regwrite, jump;
   wire [3:0] alucontrol; //modify from wire [2:0]
   wire       lb_en; // add lb_en

 // Instantiate Controller
 controller c(   .op        (instr[31:26]),
                 .funct     (instr[5:0]),
                 .zero      (zero),
                 .lb_en     (lb_en), // add lb_en
                 .signext   (signext),
                 .shiftl16  (shiftl16),
                 .memtoreg  (memtoreg),
                 .memwrite  (memwrite),
                 .pcsrc     (pcsrc),
                 .alusrc    (alusrc),
                 .regdst    (regdst),
                 .regwrite  (regwrite),
                 .jump      (jump),
                 .alucontrol (alucontrol));

 // Instantiate Datapath
 datapath dp(    .clk       (clk),
                 .reset     (reset),
                 .lb_en     (lb_en), // add lb_en
                 .signext   (signext),
                 .shiftl16  (shiftl16),
                 .memtoreg  (memtoreg),
                 .pcsrc     (pcsrc),
                 .alusrc    (alusrc),
                 .regdst    (regdst),
                 .regwrite  (regwrite),
                 .jump      (jump),
                 .alucontrol (alucontrol),
                 .zero      (zero),
                 .pc        (pc),
                 .instr     (instr),
                 .aluout    (memaddr),
                 .writedata (memwritedata),
                 .readdata  (memreaddata));

endmodule
```

6. Results of Part B

Figure 12 represents a screenshot of the Signal Tap logic analyser to test XOR and XORI. The input instructions (Table 5) include lui$1 with 0xFFFF, lui$2 with 0x00FF, addiu$2 with $2 and 0x00FF, xor$3 with $1 and $2, xori$4 with $2 and 0xFFFF, and lab1: j lab1. From these instructions, register 1 contains the value 0xFFFF_0000, while register 2 contains the value 0x00FF_00FF.

For the XOR operation between register 1 and register 2, the expected result is 0xFF00_00FF. It can be observed in Figure 12 (highlighted in a blue block) that "a" represents the value of register 1 and "b" represents the value of register 2, both of which are correct. Additionally, the result in the ALU module gives the correct answer as XOR logic computation. Moreover, the "funct" signal in the logic analyser is 26h, matching the function field of the XOR operation, and the "op" signal represents the correct opcode of XOR.

Table 5 Input instructions to test XOR and XORI.

1	lui$1, 0xFFFF
2	lui$2, 0x00FF
3	addiu$2, $2, 0x00FF
4	xor$3, $1, $2
5	xori$4, $2, 0xFFFF
6	lab1: j lab1

For the XORI operation between register 2 and value 0xFFFF, the expected result is 0x00FF_FF00. From the logic analyser (refers to green box in Figure 12) that "a" represents the value of register 2 and "b" represents the input value 0xFFFF, both of which are correct. Additionally, the result in the ALU module gives the correct answer as XOR logic computation. Moreover, the "op" signal in the logic analyser is 0Eh, matching the opcode field of the XORI operation, and the "alucontrol" signal represents the correct value for both XOR and XORI instructions.

This case examines all possible combinations of 00 and FF for both the XOR and XORI instructions, specifically testing 00 with 00, 00 with FF, FF with 00, and FF with FF. The XOR instruction (xor $3, $1, $2) computes the bitwise XOR of registers $1 and $2, while the XORI instruction (xori $4, $2, 0xFFFF) performs the XOR operation between register $2 and the immediate value 0xFFFF. The expected values for registers $3 and $4 after executing these instructions are presented in Table 6, reflecting the outcome of each XOR operation.

Table 6 Register values after executing XOR and XORI instructions.

	xor$3, $1, $2	**xori$4, $2, 0xFFFF**
$1	0xFFFF 0000	
$2	0x00FF 00FF	0x00FF 00FF
$3	**0xFF00 00FF**	
		0x0000 FFFF
$4		**0x00FF FF00**

The input instructions listed in Table 7 are used to test the LBU instruction. They include the following sequence: lui $2, 0x0FFF; ori $2, $2, 0xF000; addiu $3, $0, 0x0200; sw $2, 0x0000 ($3); lw $4, 0x0000 ($3); lbu $2, 0x0000 ($3); lbu $2, 0x0001 ($3); lbu $2, 0x0002 ($3); lbu $2, 0x0003 ($3); and lab1: j lab1. From these instructions, it is expected that register 2 contains the value 0x0FFF_F000, and at memory location 0x200, there must be a word with the value 0x0FFF_F000. When performing the lbu instruction with the offset 0x0000 ($3), the result should be the least significant byte (LSB) of the word, which is "00". Similarly, for lbu with offsets 0x0001 ($3), 0x0002 ($3), and 0x0003 ($3), the results should correspond to byte 1 (F0), byte 2 (FF), and byte 3 (0F) of the word, respectively.

In Figure 13, the lbu instruction results for each byte align with the "wd" signal in the regfile module (green box). The lw instruction correctly displays the entire word (black box). The select signal of the lbumux ("lb_en") is low for lw and high for lbu instructions (blue box). The "we" signal (regwrite) is low during word storage and high during word or byte loading (green arrow).

Table 7 Input instructions to test LBU.

1	lui$2, 0x0FFF
2	ori$2, $2, 0xF000
3	addiu$3, $0, 0x0200
4	sw$2, 0x0000 ($3)
5	lw$4, 0x0000 ($3)
6	lbu$2, 0x0000 ($3)
7	lbu$2, 0x0001 ($3)
8	lbu$2, 0x0002 ($3)
9	lbu$2, 0x0003 ($3)
10	lab1: j lab1

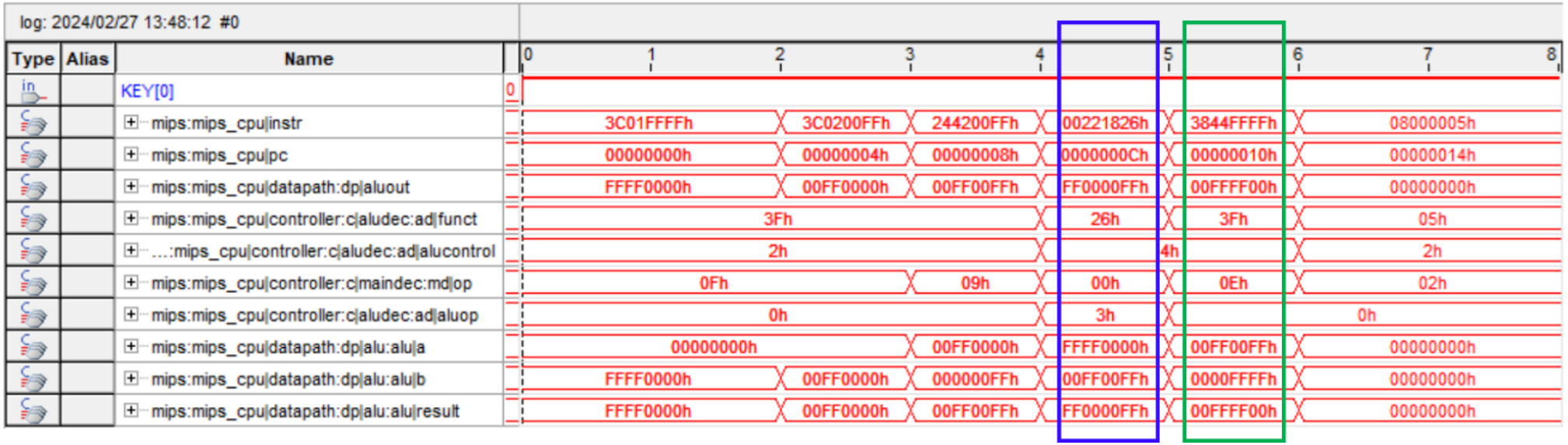

Figure 12 Signal Tap logic analyser screenshot
(XOR and XORI instructions)

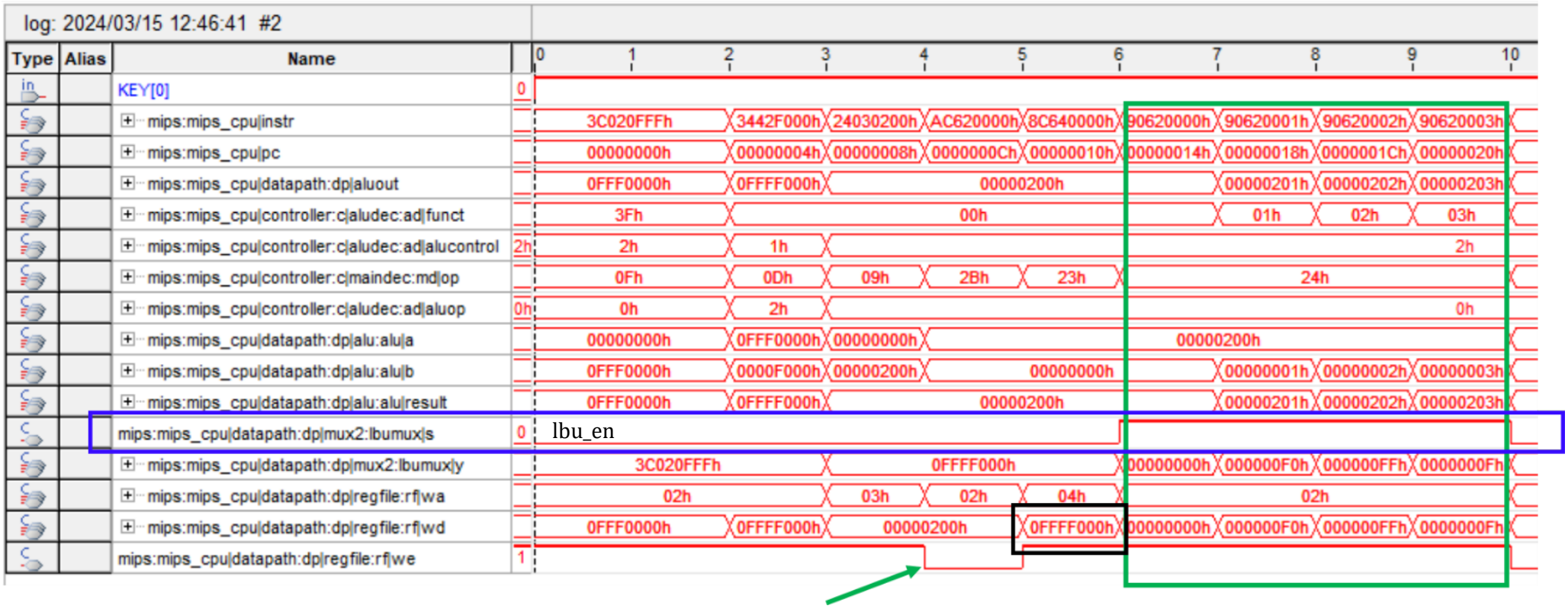

Figure 13 Signal Tap logic analyser screenshot
(LBU instruction)

7. Description of each module: Part C

In Part C, we will design a PWM (Pulse Width Modulation) unit intended to control the LEDR5 using its output. This PWM unit will be accessible to the MIPS Processor via a memory location. It will consist of a free-running 8-bit counter with a comparator, where one of the comparator inputs can be set by the MIPS Processor through a register write operation. When the counter value is less than the register value, the output will be high, and when the counter value exceeds the register value, the output will be low. It's important to note that the PWM unit is distinct from the existing GPIO unit. Therefore, we need to adjust the instantiation of the GPIO module so that the LEDR5 is allocated to the PWM unit instead of the GPIO unit.

Figure 14 represents the block diagram of the PWM module. The reset signal, chip select signal, write enable signal, location address, and DataIn are used as inputs to the register block in the PWM module. The counter is an 8-bit free-running counter, and the tick_signal (output from the counter) and pwm_register (input register value from the user) are compared in the comparator to produce the PWM output, named LEDR5. This output is used to control the brightness of the LEDR5 on the DE2 board, allowing it to light up or dim.

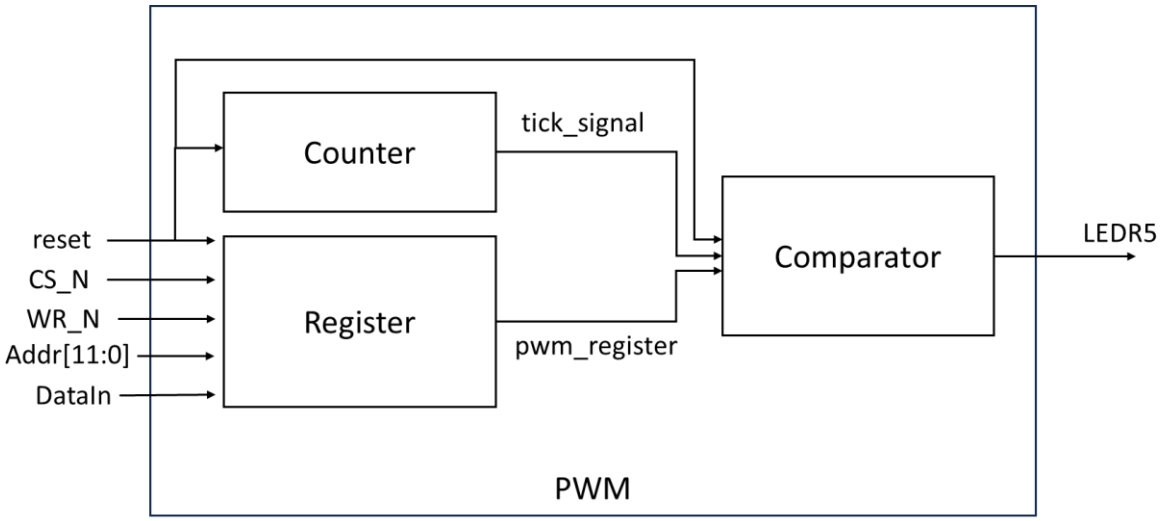

Figure 14 Block Diagram of PWM module.

7.1 counter

The “counter” module provided implements an 8-bit free-running counter with clock and reset inputs, and an 8-bit output "tick". It utilizes two internal 8-bit registers, p_counter and n_counter, with the "tick" output assigned to the value of p_counter. The counter operates based on two always blocks: one triggered on the positive clock edge handles reset logic, setting p_counter to 0 when reset is asserted, otherwise maintaining its value; the second block updates p_counter based on its current value, resetting it to 0 when it reaches its maximum value of 255 (specified by CLOCK_DIVISOR), otherwise incrementing it by 1. The ASM chart (Figure 15) illustrates that the counter increments when its value is not 255 and resets to 0 when it reaches 255.

```
module counter(
    input wire clock,
    input wire reset,
    output [7:0] tick
);


  // Assign the clock divisor
    parameter CLOCK_DIVISOR = 256;

  // Internal counter for free running 8 bits counter
    reg [7:0] p_counter,n_counter;   //8 bits

    assign tick = p_counter;

    always @(posedge clock)
        begin
            if (reset)
                p_counter <= 0;
            else
                p_counter <= n_counter;
        end

    always @( p_counter)
        begin
            n_counter = p_counter;
            if (p_counter == (CLOCK_DIVISOR - 1))
                n_counter = 8'd0;
            else
                n_counter = p_counter + 8'd1;
        end
endmodule
```

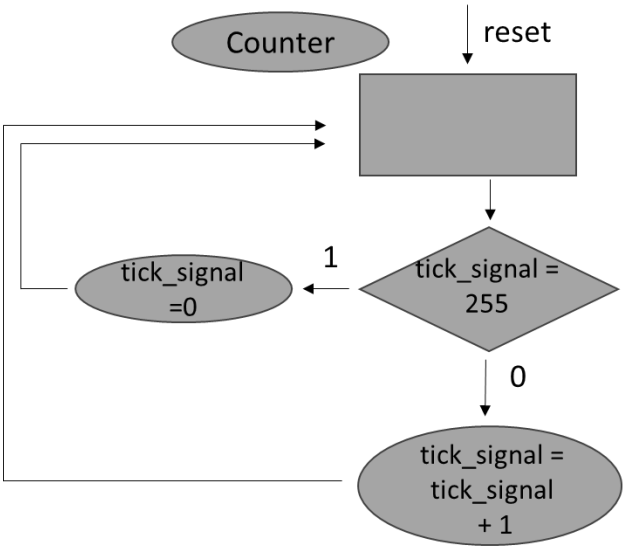

Figure 15 8 bits counter ASM chart

7.2 PWM

This Verilog module implements a Pulse Width Modulation (PWM) controller for driving an LED (LEDR5). It utilizes a pwm_register to store the duty cycle value and a tick_signal generated from an 8-bit free-running counter. The LEDR_R wire is assigned based on a comparison between the pwm_register and tick_signal, determining the LED's state. This signal passes through a flip-flop to produce the final output signal LEDR5. The module handles chip select and write enable signals (CS_N and WR_N, respectively) to update the pwm_register when necessary, provided the address of the register is 0x200. The PWM ASM chart (Figure 16) illustrates the logic of the output LEDR5 based on the comparison of pwm_register and tick_signal.

```
module PWM(
    input clk,
    input reset,
    input CS_N,
    input WR_N,
    input [11:0] Addr,
    input [7:0] DataIn,
    output wire LEDR5
 );

    reg [7:0] pwm_register;
    wire [7:0] tick_signal;
    wire LEDR_R;


    // Instantiate Couter
  counter c ( .clock(clk),
              .reset(reset),
              .tick(tick_signal));

    //flip-flop
    always @(posedge clk, posedge reset)
        if (reset) LEDR5 <= 0;
        else       LEDR5 <= LEDR_R;

      //write output Register
    always @( posedge clk, posedge reset)
        begin
            if(reset)
                begin
                    pwm_register <= 0;
                end

            else if(~CS_N && ~WR_N)
                begin
                    if  (Addr[11:0] == 12'h200)
                        pwm_register    <= DataIn;
                end
        end

    //output
    assign LEDR_R = (pwm_register >= tick_signal);


endmodule
```

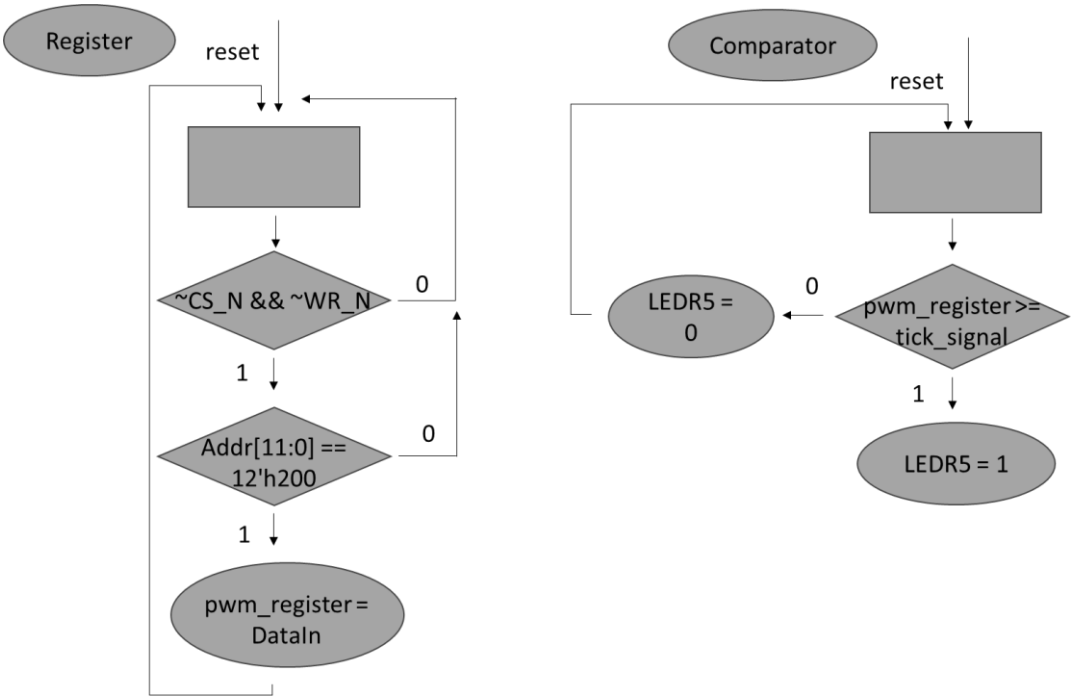

Figure 16 PWM module ASM chart
(left) Register ASM chart (right) Comparator ASM chart.

The RTL of both the PWM and counter (depicted in Figure 17) ensures that no unnecessary D-flipflops or latches are produced. In the PWM module, the last D-flipflop is utilized to prevent unsynchronized signals for the output. Figure 18(a) illustrates the PWM simulation from 0 to 1 microsecond, while Figure 18(b) showcases the simulation from 4.8 to 5.6 microseconds. During a reset (high) state, the counter resets to 0, as indicated by the red arrow in Figure 18(a). Additionally, the p_counter represents a free-running 8-bit value ranging from 0 to 255; when it reaches 255, it resets back to 0 and resumes counting, as highlighted by the yellow arrow in Figure 18(b).

Moreover, when both CS_N and WR_N are low and the address is 0x200, the pwm_register updates its value from DataIn. However, if only CS_N is low, or only WR_N is low, or if both signals are high (as indicated by the orange arrows in Figure 18(a)), the value of pwm_register does not update to be FF as DataIn; this can be observed in the simulation where the value of pwm_register remains 10. Similarly, when both CS_N and WR_N are low but the address is not 0x200, the value of pwm_register does not update to be FF as DataIn. The blue arrow signifies that LEDR5 turns low when the counter surpasses the pwm_register value of 16 (10h in hexadecimal). This implies a duty cycle of 16 clock cycles, as indicated by the blue bracket.

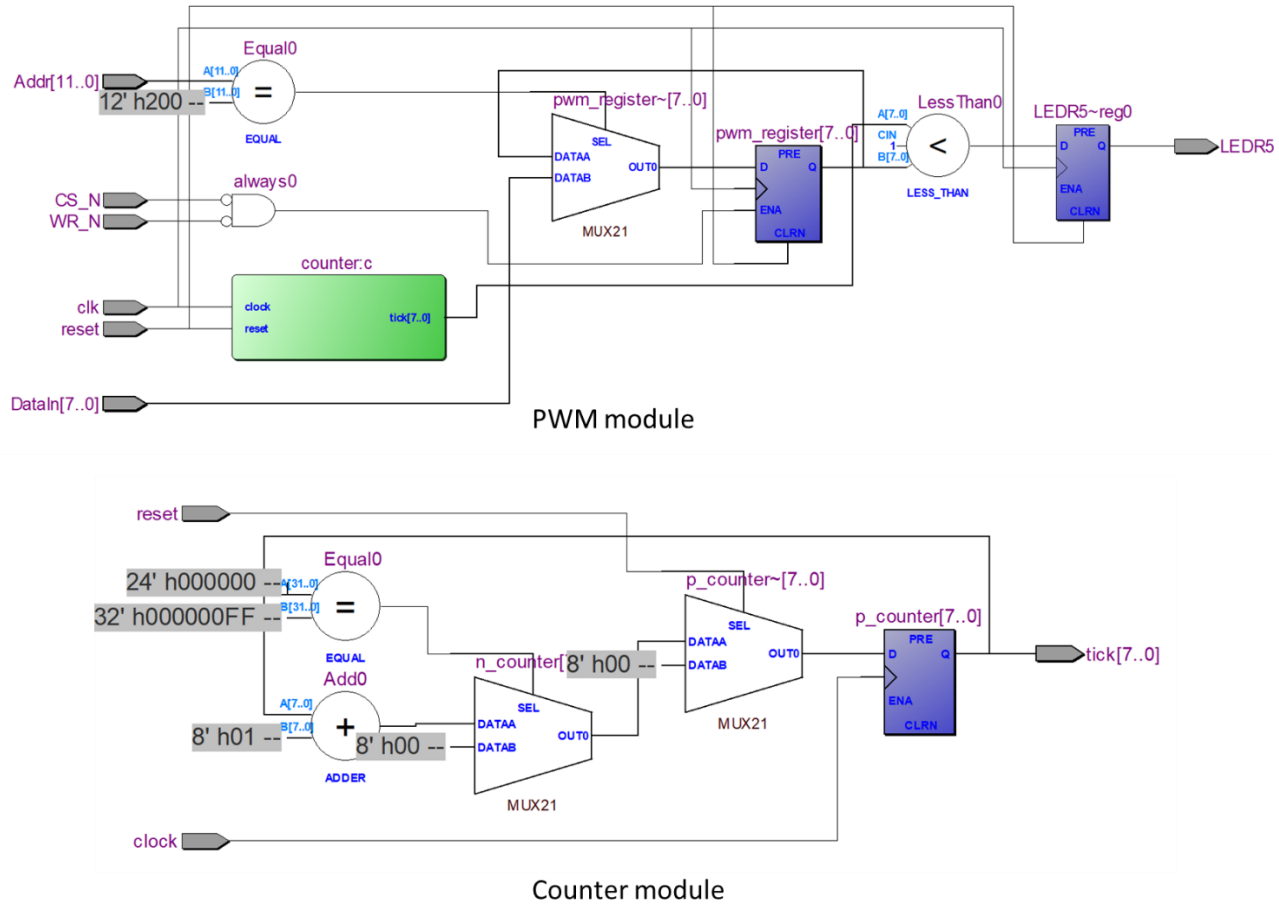

Figure 17 (a) PWM RTL view and (b) counter RTL view

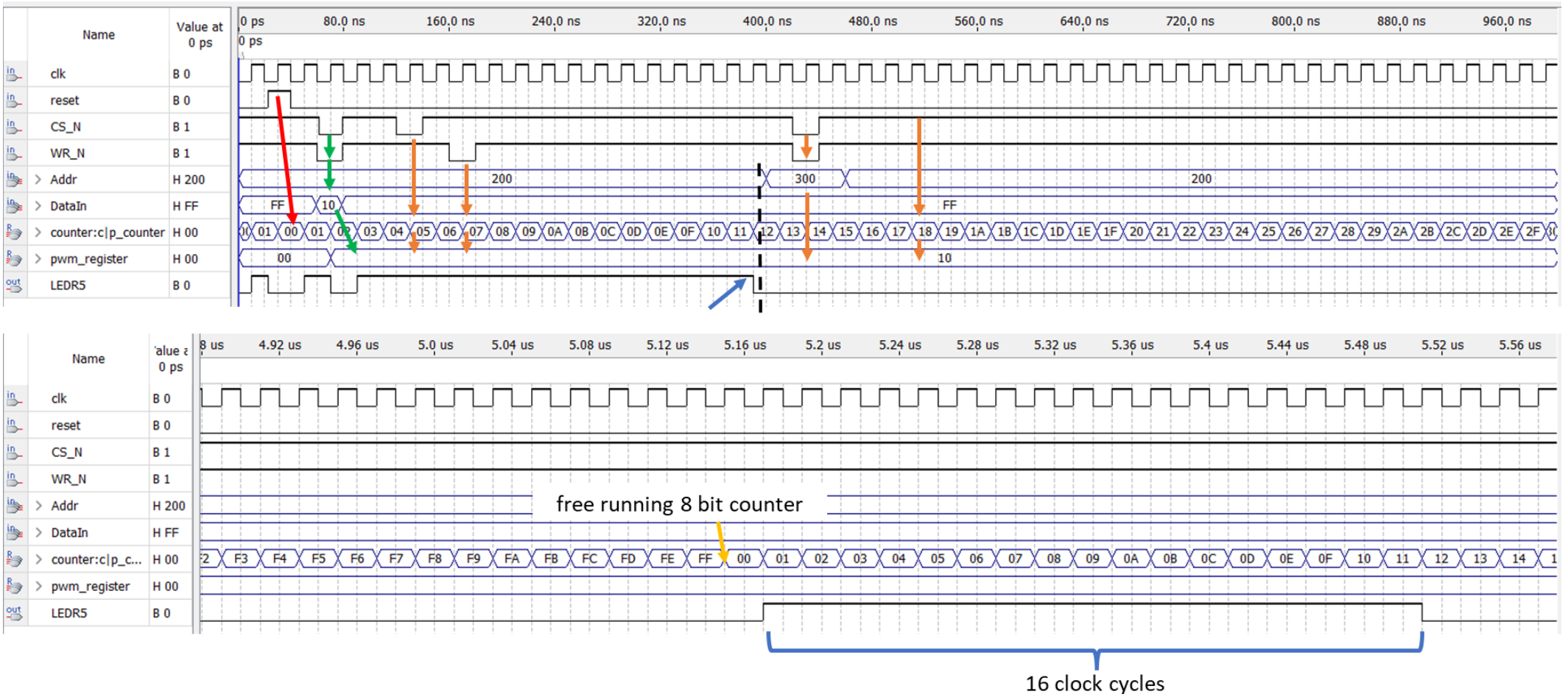

Figure 18 (a) PWM simulation (0 – 1 µs) (b) PWM simulation (4.8 – 5.6 µs)

7.3 MIPS_System

The MIPS_System module integrates ALTPLL_clkgen, mips, ram2port_inst_data, Addr_Decoder, TimerCounter, GPIO, and PWM modules. It interfaces with CLOCK_50, [3:0] KEY, [17:0] SW inputs, and produces outputs including [6:0] HEX7 through HEX0, [17:0] LEDR, and [8:0] LEDG.

To modify the instantiation of the GPIO module such that LEDR5 is allocated to the PWM unit instead of the GPIO unit, the LEDR signal should be concatenated without LEDR5 as follows: {LEDR [17:6], LEDR [4:0]}. This concatenation ensures that LEDR [5] is excluded from GPIO unit. Figure 19 shows RTL view of GPIO unit and PWM unit.

In the PWM module, the clock signal used is clk0, and the reset signal is ~reset_ff, consistent with the GPIO module. However, the chip select signal (CS_N) is derived from the cs_mem_n signal, indicating the selected location 0x200, which resides in the Instruction & Data Memory peripheral. The output of the PWM module controls LEDR [5], as designated for PWM unit control. Figure 18 shows that LEDR5 comes from PWM, while the other LEDs come from GPIO.

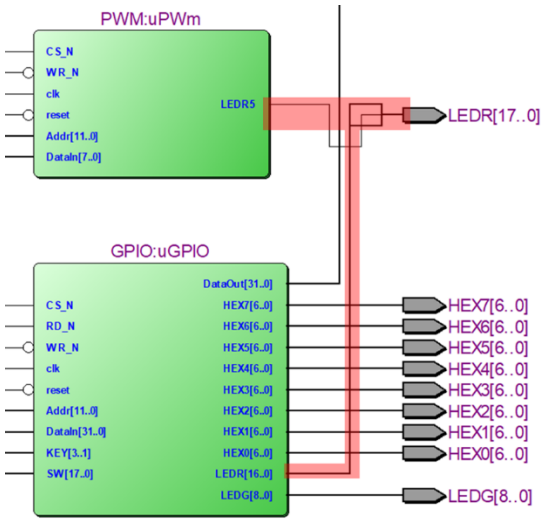

Figure 19 MIPS_System RTL view

```
module MIPS_System(
    input           CLOCK_50,
    …
    output [17:0]   LEDR,
    …
);

    wire        clk0;
    …


    GPIO uGPIO (
            .clk        (clk0),
            .reset      (~reset_ff),
            .CS_N       (cs_gpio_n),
            .RD_N       (~data_re),
            .WR_N       (~data_we),
            .Addr       (data_addr[11:0]),
            .DataIn     (write_data),
            .DataOut    (read_data_gpio),
            .Intr       (),

            .KEY        (KEY[3:1]),
            .SW         (SW),
            .HEX7       (HEX7),
            .HEX6       (HEX6),
            .HEX5       (HEX5),
            .HEX4       (HEX4),
            .HEX3       (HEX3),
            .HEX2       (HEX2),
            .HEX1       (HEX1),
            .HEX0       (HEX0),
            .LEDR       ({LEDR[17:6],LEDR[4:0]}),
            .LEDG       (LEDG));

    PWM uPWm(
            .clk        (clk0),
            .reset      (~reset_ff),
            .CS_N       (cs_mem_n),
            .WR_N       (~data_we),
            .Addr       (data_addr[11:0]),
            .DataIn     (write_data),

            .LEDR5      (LEDR[5]));

endmodule
```

8. Results of Part C

Figure 20 illustrates the instruction sequence, highlighting MIPS design constraints that permit only basic instructions and Figure 21 showcases a screenshot from the Signal Tap logic analyser, providing insight into the testing process. The instructions outlined in Table 8 encompass various operations, including adding an immediate value to register $1, storing the result at memory location 0x0200, loading the stored value into register $3, and establishing an infinite loop for continuous execution. Similarly, Table 9 instructions, resembling those in Table 8 but involving the storage of 0x000A to location 0x0200, are pivotal in PWM unit testing.

Within Figure 21(a) and (b), observations of DataIn equals 0x0A and 0xFF respectively, subsequent to store instructions, illustrate the update of pwm_register to reflect the corresponding DataIn value, clearly depicted by green lines and arrows. The accurate storage of input values at the designated location (0x200) is corroborated by the load instruction, where wd signals align with 0x0A (highlighted by the black arrow in 21a) and 0xFF (highlighted by the black arrow in 19b). Additionally, the address signal accurately matches the expected 0x200, as indicated by the Addr signal (blue arrows in 21(a), 21(b)), along with the low values of WR_N and CS_N signals, as highlighted by the yellow arrows. After updating pwm_register to 0x0A and 0xFF, LEDR5 goes high when the input register exceeds the tick value. In Figure 21(a), LEDR5 goes low when the tick value becomes 0x0B, surpassing the register value of 0x0A. However, in Figure 21(b), LEDR5 remains high despite the tick value reaching 0x0B, as the register value remains at 0xFF (orange arrows in Figure 21).

Table 6 Input instructions to test PWM output

1	addiu $1, $0, 0x000A
2	sw $1, 0x0200
3	lw $3, 0x0200
4	lab1: j lab1

Table 7 Input instructions to test PWM output

1	addiu $1, $0, 0x00FF
2	sw $1, 0x0200
3	lw $3, 0x0200
4	lab1: j lab1

Table 10 outlines software instructions to gradually adjust the PWM register, brightening and dimming the LED in a repeating cycle.

Initially, register 4 is set to 0x000F using the instruction ori $4 by $0 (zero), which is used as the delay value. Register 6 is then assigned the value 0x00FF with the addiu $6 with $0 instruction, serving as a comparison value for the input register. Plus, register 7 is set to 0x0001 using addiu $7 with $0, utilized as the comparison output for the TRUE case in the beq instruction.

Table 8 Brightening and Dimming LED instructions

1		ori $4, $0, 0x000F
2		addiu $6, $0, 0x00FF
3		addiu $7, $0, 0x0001
4	again:	addiu $1, $0, 0x0001
5	loop:	sw $1, 0x0200
6		add $2, $0 $0
7	delay:	addiu $2, $2, 0x0001
8		slt $3, $2, $4
9		beq $3, $7, delay
10		addiu $1, $1, 0x0002
11		slt $5, $1, $6
12		beq $5, $7, loop
13		j again
14	lab1:	j lab1

Then, register 1 is set to 0x0001 using the instruction addiu $1, $0, 0x0001, which serves as the input register value to compare to the tick value. Subsequently, register 1 is stored at address 0x0200 with sw $1, 0x0200, facilitating the writing process. Additionally, a delay is introduced before changing the value of register 1.

To implement a delay function, register 2 is initialized to 0x0000 as the delay value. Subsequently, register 2 is incremented by 0x0001 using addiu $2, $2, 0x0001, and compared to register 4 (0x000F). The result of this comparison is stored in register 3 using slt $3, $2, $4. Then, the output from the previous instruction ($3) is checked to determine if it equals $7 (0x0001, TRUE). If $3 equals $7, indicating that the delay condition is met, the program returns to the delay stage and increments the value of register 2 further. Once $2 surpasses $4, $3 becomes 0x0001, signalling that the program should proceed to increment the value of register 1. Thus, the delay function is achieved by incrementing the value of register 2 until it reaches the set value ($4). Adjusting the value of $4 can effectively control the duration of the delay, allowing for longer or shorter delays as needed.

Continuing from the previous description, the program proceeds to increment the value of register 1 by 0x0002 using the addiu $1, $1, 0x0002 instruction. This incremented value corresponds to the pwm_register, which determines the high or low state of LEDR5. After incrementing the value of $1, the program checks if $1 is less than $6 (which is 0xFF) and stores the output in register $5. It then compares whether $5 equals $7 (0x0001). If this condition is true, the program returns to the loop again to increase the value of register 1. Before each increment, the program undergoes a delay using the same process. When $1 exceeds $6, indicating that LEDR5 is at the brightness stage, the program resets register 1 to 0x0001 for the dimmer stage, and the cycle repeats. This iterative process effectively controls the brightness of LEDR5 in a gradual manner.

Figure 22 depicts a screenshot of the Signal Tap logic analyser for testing the PWM unit using instructions from Table 9 (Duration = sample 254 to 270). The “instr” signal displays the correct sequence of instructions as described in the flowchart. Moreover, the "tick" signal functions effectively as an 8-bit free-running counter, resetting from 0xFF back to 0x00 as expected. When both CS_N and WR_N are low, as indicated by the yellow arrows, the pwm_register updates to 0x0B in accordance with the store word instruction. Furthermore, when the tick reaches 0x0C, highlighted by the red arrow, the output LEDR5 goes low because the pwm_register is less than the tick value, indicating that the PWM unit is functioning correctly.

Figure 23 provides a zoomed-out view of Figure 22, encompassing the entire program execution from start to finish. In this overview, it is evident that when a store instruction occurs, both CS_N and WR_N signals are low, as indicated by the yellow arrows, resulting in the update of the pwm_register value (which increments by 2). Furthermore, the duty cycle of LEDR5 corresponds directly with the pwm_register value, as highlighted by the green circles. For instance, when the pwm_register is 0x0B, LEDR5 remains high for 11 clock cycles, and when it is 0x15, LEDR5 stays high for 21 clock cycles, demonstrating the correlation between pwm_register and LEDR5 duty cycle. To ensure clear visibility on the board, it is recommended to update the delay value ($4) to 0xFFFF.

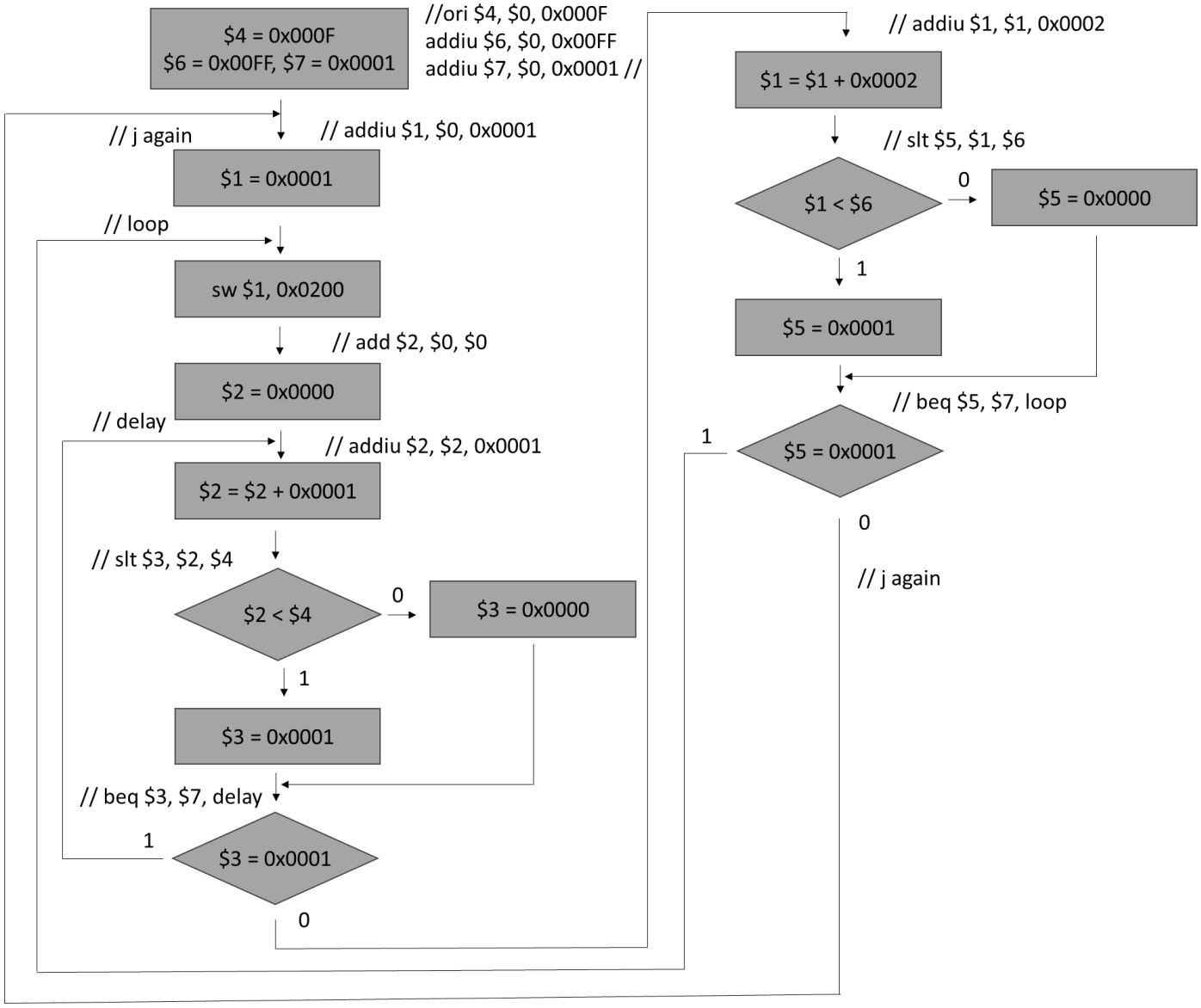

Figure 20 Brightening and Dimming LED Instructions Flowchart

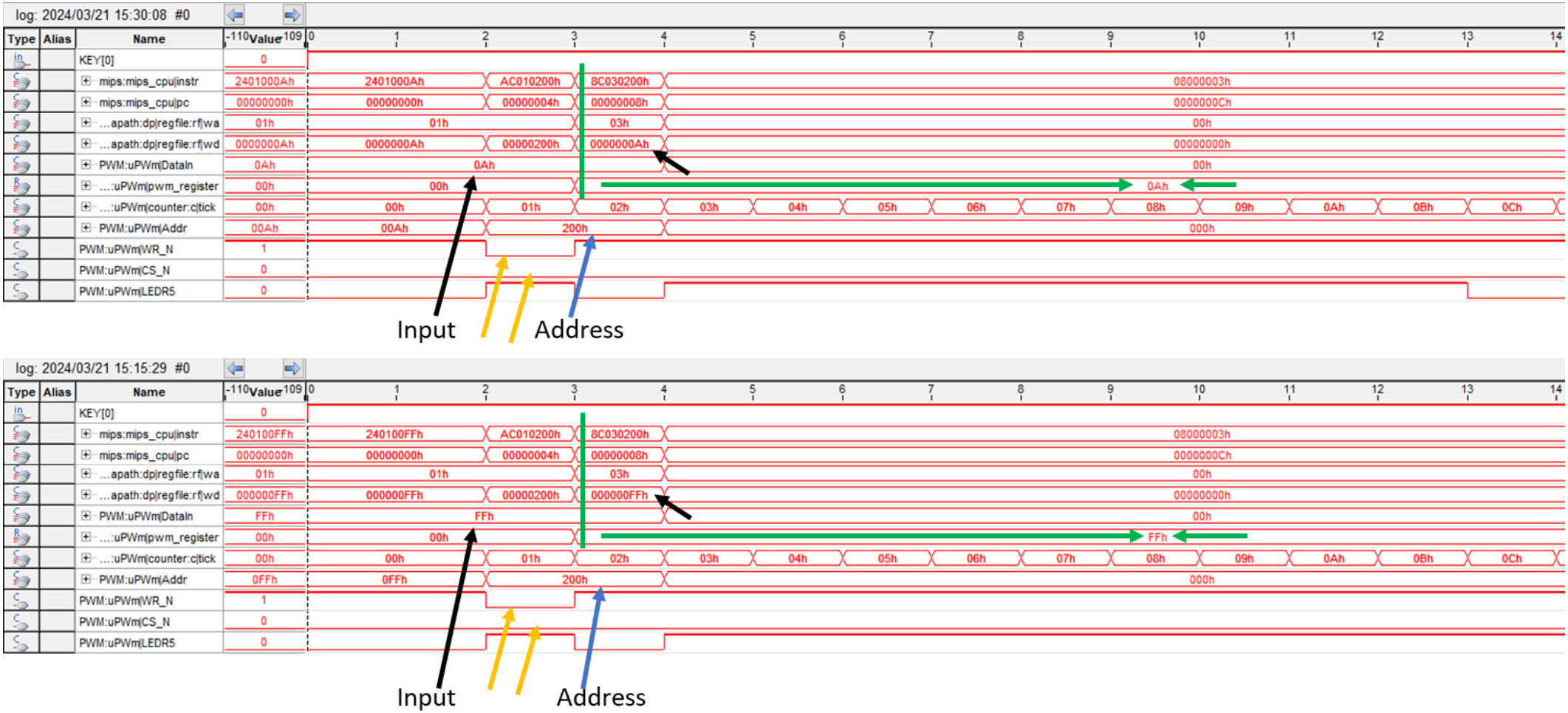

Figure 21 Signal Tap logic analyser Screenshot to test PWM unit (Instructions from Table 7, Table 8)

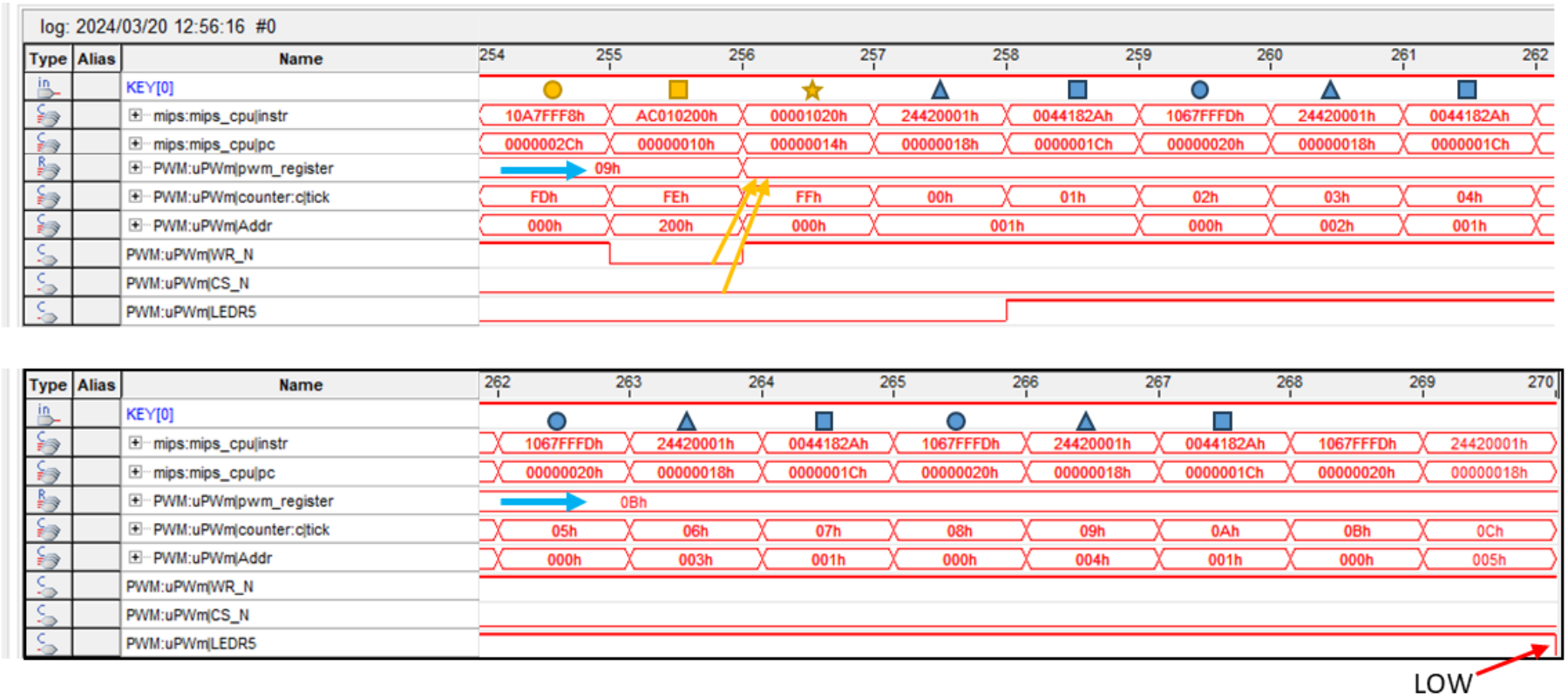

Figure 22 Signal Tap logic analyser Screenshot for PWM unit testing (Instructions from Table 9, Duration = sample 254 to 270)

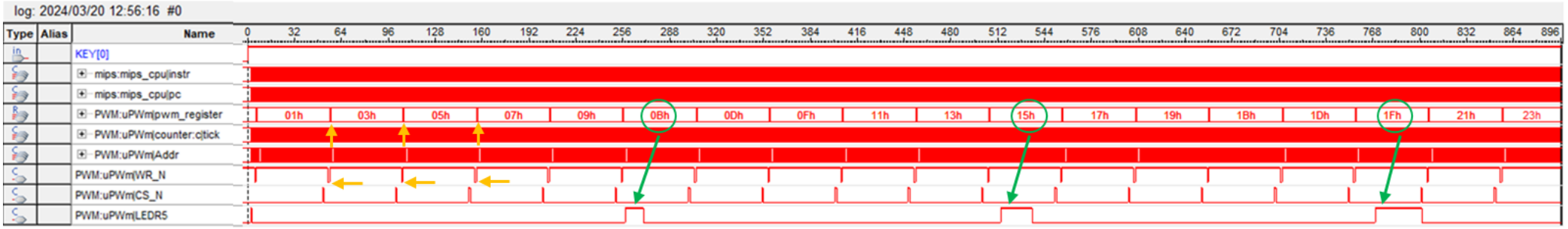

Figure 23 Signal Tap logic analyser Screenshot for PWM unit testing (Instructions from Table 9)

Appendix A: insts_data.mif file of Part A

```
3c01ffff
24020079
ac222028
24030078
ac232024
24040012
ac242020
24050019
ac252018
24060002
ac262010
0810000b
```

Appendix B: Verilog code of Part B

```
// single-cycle MIPS processor
module mips(input      clk, reset,
            output [31:0]   pc,
            input  [31:0]   instr,
            output          memwrite,
            output [31:0]   memaddr,
            output [31:0]   memwritedata,
            input  [31:0]   memreaddata);

  wire          signext, shiftl16, memtoreg, branch;
  wire          pcsrc, zero;
  wire          alusrc, regdst, regwrite, jump;
  wire [3:0]  alucontrol; //modify from wire [2:0]
  wire          lb_en; // add lb_en

  // Instantiate Controller
  controller c(    .op         (instr[31:26]),
                   .funct      (instr[5:0]),
                   .zero       (zero),
                   .lb_en      (lb_en), // add lb_en
                   .signext    (signext),
                    .shiftl16   (shiftl16),
                   .memtoreg   (memtoreg),
                   .memwrite   (memwrite),
                   .pcsrc      (pcsrc),
                   .alusrc     (alusrc),
                   .regdst     (regdst),
                   .regwrite   (regwrite),
                   .jump       (jump),
                   .alucontrol (alucontrol));

  // Instantiate Datapath
  datapath dp(     .clk        (clk),
                   .reset      (reset),
                   .lb_en      (lb_en), // add lb_en
                   .signext    (signext),
                   .shiftl16   (shiftl16),
                   .memtoreg   (memtoreg),
                   .pcsrc      (pcsrc),
                   .alusrc     (alusrc),
                   .regdst     (regdst),
                   .regwrite   (regwrite),
                   .jump       (jump),
                   .alucontrol (alucontrol),
                   .zero       (zero),
                   .pc         (pc),
                   .instr      (instr),
                   .aluout     (memaddr),
                   .writedata  (memwritedata),
                   .readdata   (memreaddata));
endmodule
```

```
module controller( input  [5:0] op, funct,
                   input        zero,
                   output       signext, lb_en,
                   output       shiftl16,
                   output       memtoreg, memwrite,
                   output       pcsrc, alusrc,
                   output       regdst, regwrite,
                   output       jump,
                   output [3:0] alucontrol); //modify from output [2:0] alucontrol

  wire [2:0] aluop; //modify from wire [1:0] aluop
  wire       branch;

  maindec md(   .op    (op),
                .lb_en     (lb_en), // add lb_en
                .signext   (signext),
                .shiftl16  (shiftl16),
                .memtoreg  (memtoreg),
                .memwrite  (memwrite),
                .branch    (branch),
                .alusrc    (alusrc),
                .regdst    (regdst),
                .regwrite  (regwrite),
                .jump      (jump),
                .aluop     (aluop));

  aludec  ad(       .funct      (funct),
                .aluop      (aluop),
                .alucontrol (alucontrol));

  assign pcsrc = branch & zero;

endmodule
```

```
module maindec(   input  [5:0]   op,
                  output         signext, lb_en, // add lb_en
                  output         shiftl16,
                  output         memtoreg, memwrite,
                  output         branch, alusrc,
                  output         regdst, regwrite,
                  output         jump,
                  output [2:0]   aluop); //modify from output [1:0] aluop

  reg [12:0] controls; //modify from reg [10:0] controls

  assign {     lb_en, signext, shiftl16, regwrite, regdst,
               alusrc, branch, memwrite,
               memtoreg, jump, aluop       } = controls;

  always @(*)
    case(op)
        6'b000000: controls <= 13'b0001100000011; // Rtype
        6'b100011: controls <= 13'b0101010010000; // LW
        6'b101011: controls <= 13'b0100010100000; // SW
        6'b000100: controls <= 13'b0100001000001; // BEQ
        6'b001000,
        6'b001001: controls <= 13'b0101010000000; // ADDI, ADDIU: only difference is exception
        6'b001101: controls <= 13'b0001010000010; // ORI
        6'b001111: controls <= 13'b0011010000000; // LUI
        6'b000010: controls <= 13'b0000000001000; // J
        6'b001110: controls <= 13'b0001010000100; // XORI
        6'b100100: controls <= 13'b1101010010000; // LBU
        default:   controls <= 13'bxxxxxxxxxxxxx; // ???
    endcase
endmodule

module aludec( input      [5:0]    funct,
               input      [2:0]    aluop, //modify from input [1:0] aluop,
               output reg [3:0]    alucontrol);
  always @(*)
    case(aluop)
        3'b000: alucontrol <= 4'b0010;  // add
        3'b001: alucontrol <= 4'b1010;  // sub
        3'b010: alucontrol <= 4'b0001;  // or
        3'b100: alucontrol <= 4'b0100;  // xor
        default: case(funct)          // RTYPE
            6'b100000,
            6'b100001: alucontrol <= 4'b0010; // ADD, ADDU: only difference is exception
            6'b100010,
            6'b100011: alucontrol <= 4'b1010; // SUB, SUBU: only difference is exception
            6'b100100: alucontrol <= 4'b0000; // AND
            6'b100101: alucontrol <= 4'b0001; // OR
            6'b101010: alucontrol <= 4'b1011; // SLT
            6'b100110: alucontrol <= 4'b0100; // XOR
            default:   alucontrol <= 4'bxxxx; // ???
            endcase
    endcase
endmodule
```

```
module datapath(  input        clk, reset,
                  input        signext, lb_en, // add lb_en (load byte enable)
                  input        shiftl16,
                  input        memtoreg, pcsrc,
                  input        alusrc, regdst,
                  input        regwrite, jump,
                  input  [3:0]  alucontrol, //// modify from  input  [2:0]  alucontrol
                  output       zero,
                  output [31:0] pc,
                  input  [31:0] instr,
                  output [31:0] aluout, writedata,
                  input  [31:0] readdata);

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh, shiftedimm;
  wire [31:0] srca, srcb;
  wire [31:0] lw_lb_data, lb_data_ext; // add load word_load byte data
  wire [7:0]  lb_data;
  wire [31:0] result;
  wire        shift;

  // next PC logic
  flopr #(32) pcreg (    .clk   (clk),
                         .reset (reset),
                         .d     (pcnext),
                         .q     (pc));

  adder       pcadd1 (   .a (pc),
                         .b (32'b100),
                         .y (pcplus4));

  sl2         immsh (    .a (signimm),
                         .y (signimmsh));

  adder       pcadd2 (   .a (pcplus4),
                         .b (signimmsh),
                         .y (pcbranch));

  mux2 #(32)  pcbrmux(.  d0  (pcplus4),
                         .d1  (pcbranch),
                         .s   (pcsrc),
                         .y   (pcnextbr));

  mux2 #(32)  pcmux (    .d0   (pcnextbr),
                         .d1   ({pcplus4[31:28], instr[25:0], 2'b00}),
                         .s    (jump),
                         .y    (pcnext));
```

```
// register file logic
    regfile     rf(          .clk     (clk),
                             .we      (regwrite),
                             .ra1     (instr[25:21]),
                             .ra2     (instr[20:16]),
                             .wa      (writereg),
                             .wd      (result),
                             .rd1     (srca),
                             .rd2     (writedata));

    mux2 #(5)   wrmux(    .d0  (instr[20:16]),
                          .d1  (instr[15:11]),
                          .s   (regdst),
                          .y   (writereg));

    mux2 #(32)  resmux(   .d0 (aluout),
                          .d1 (lw_lb_data),
                          .s  (memtoreg),
                        .   y  (result));

    sign_zero_ext  sze(.a       (instr[15:0]),
                       .signext (signext),
                       .y       (signimm[31:0]));

    shift_left_16 sl16( .a         (signimm[31:0]),
                       .shiftl16  (shiftl16),
                       .y         (shiftedimm[31:0]));

  // LBU

    mux2 #(32)  lbumux(  .d0 (readdata),
                         .d1 (lb_data_ext),
                         .s  (lb_en),
                         .y  (lw_lb_data));

    mux4 #(8) selectbytemux(     .d0 (readdata[7:0]),
                                 .d1 (readdata[15:8]),
                                 .d2 (readdata[23:16]),
                                 .d3 (readdata[31:24]),
                                 .s  (aluout[1:0]),
                                 .y  (lb_data));

    sign_zero_byte_ext  byteex(  .a (lb_data),
                                 .y  (lb_data_ext));
```

```
 // ALU logic
 mux2 #(32)  srcbmux(    .d0 (writedata),
                         .d1 (shiftedimm[31:0]),
                          .s  (alusrc),
                         .y  (srcb));

 alu        alu(        .a      (srca),
                        .b      (srcb),
                        .alucont (alucontrol),
                        .result  (aluout),
                        .zero    (zero));
endmodule
```

```
module alu( input      [31:0]       a, b,
            input      [3:0]        alucont,
            output reg [31:0]       result,
            output                  zero);

  wire [31:0] b3, sum, slt; // modify b3

  assign b3 = alucont[3] ? ~b:b; // negative b
  assign sum = a + b3 + alucont[3]; // a+b / a-b
  assign slt = sum[31];

  always@(*)
    case(alucont[2:0])
        3'b000: result <= a & b;
        3'b001: result <= a | b;
        3'b010: result <= sum;
        3'b011: result <= slt;
        3'b100: result <= a ^ b; //xor
        default: result <= 32'bx;
    endcase

  assign zero = (result == 32'b0);

endmodule
```

```
module mux4 #(parameter WIDTH = 8)
            (   input  [WIDTH-1:0]   d0, d1, d2, d3,
                input  [1:0]         s,
                output [WIDTH-1:0]   y);

    reg [WIDTH-1:0] mux_int;

    assign y =    (s == 2'b00) ? d0:
                  (s == 2'b01) ? d1:
                  (s == 2'b10) ? d2:
                  (s == 2'b11) ? d3:
                  8'bx;

endmodule

module sign_zero_byte_ext(     input [7:0]          a,
                               output reg [31:0]    y);

    always @(*)
        y <= {24'b0, a[7:0]};

endmodule
```

Appendix C: Verilog code of Part C

```
module PWM(
    input clk,
    input reset,
    input CS_N,
    input WR_N,
    input [11:0] Addr,
    input [7:0] DataIn,
    output reg LEDR5
 );

    reg [7:0] pwm_register;
    wire [7:0] tick_signal;
    wire LEDR_R;

   // Assign the clock divisor
    parameter CLOCK_DIVISOR = 256;

 // Internal counter for free running 8 bits counter
    reg [7:0] p_counter,n_counter;   //8 bits

    // Instantiate Couter
  counter c(      .clock(clk),
                  .reset(reset),
                  .tick(tick_signal));

 //write output Register
 always @(posedge clk, posedge reset)
    begin
        if(reset)
            begin
                pwm_register <= 0;
            end
        else if(~CS_N && ~WR_N)
            begin
                if     (Addr[11:0] == 12'h200)  pwm_register          <= DataIn;
            end
 end


    //output
    assign LEDR_R = (pwm_register >= tick_signal);

    //flip-flop
    always @(posedge clk, posedge reset)
        if (reset) LEDR5 <= 0;
        else       LEDR5 <= LEDR_R;

endmodule
```

```
module counter(
    input wire clock,
    input wire reset,
    output [7:0] tick
);

  // Assign the clock divisor
    parameter CLOCK_DIVISOR = 256;

  // Internal counter for free running 8 bits counter
    reg [7:0] p_counter,n_counter;   //8 bits

    assign tick = p_counter;

    always @(posedge clock)
        begin
            if (reset)
                p_counter <= 0;
            else
                p_counter <= n_counter;
        end

    always @( p_counter)
        begin
            n_counter = p_counter;
            if (p_counter == (CLOCK_DIVISOR - 1))
                n_counter = 8'd0;
            else
                n_counter = p_counter + 8'd1;
        end
endmodule
```

```
module GPIO(
  input clk,
  input reset,
  input CS_N,
  input RD_N,
  input WR_N,
  input [11:0] Addr,
  input [31:0] DataIn,
  input [3:1] KEY,
  input [17:0] SW,

  output reg [31:0] DataOut,
  output Intr,
  output [6:0] HEX7,
  output [6:0] HEX6,
  output [6:0] HEX5,
  output [6:0] HEX4,
  output [6:0] HEX3,
  output [6:0] HEX2,
  output [6:0] HEX1,
  output [6:0] HEX0,
  output [16:0] LEDR,
  output [8:0] LEDG
  );

  reg [31:0] SW_StatusR;
  reg [31:0] KEY_StatusR;
  reg [31:0] LEDG_R;
  reg [31:0] LEDR_R;
  reg [31:0] HEX0_R;
  reg [31:0] HEX1_R;
  reg [31:0] HEX2_R;
  reg [31:0] HEX3_R;
  reg [31:0] HEX4_R;
  reg [31:0] HEX5_R;
  reg [31:0] HEX6_R;
  reg [31:0] HEX7_R;

  wire key1_pressed;
  wire key2_pressed;
  wire key3_pressed;

  wire sw0_pressed;
  wire sw1_pressed;
  wire sw2_pressed;
  wire sw3_pressed;
  wire sw4_pressed;
  wire sw5_pressed;
  wire sw6_pressed;
  wire sw7_pressed;
  wire sw8_pressed;
```

```
wire sw9_pressed;
wire sw10_pressed;
wire sw11_pressed;
wire sw12_pressed;
wire sw13_pressed;
wire sw14_pressed;
wire sw15_pressed;
wire sw16_pressed;
wire sw17_pressed;

 key_detect  key1 (clk, reset, KEY[1], key1_pressed);
 key_detect  key2 (clk, reset, KEY[2], key2_pressed);
 key_detect  key3 (clk, reset, KEY[3], key3_pressed);

 key_detect  sw0  (clk, reset, ~SW[0],  sw0_pressed);
 key_detect  sw1  (clk, reset, ~SW[1],  sw1_pressed);
 key_detect  sw2  (clk, reset, ~SW[2],  sw2_pressed);
 key_detect  sw3  (clk, reset, ~SW[3],  sw3_pressed);
 key_detect  sw4  (clk, reset, ~SW[4],  sw4_pressed);
 key_detect  sw5  (clk, reset, ~SW[5],  sw5_pressed);
 key_detect  sw6  (clk, reset, ~SW[6],  sw6_pressed);
 key_detect  sw7  (clk, reset, ~SW[7],  sw7_pressed);
 key_detect  sw8  (clk, reset, ~SW[8],  sw8_pressed);
 key_detect  sw9  (clk, reset, ~SW[9],  sw9_pressed);
 key_detect  sw10 (clk, reset, ~SW[10], sw10_pressed);
 key_detect  sw11 (clk, reset, ~SW[11], sw11_pressed);
 key_detect  sw12 (clk, reset, ~SW[12], sw12_pressed);
 key_detect  sw13 (clk, reset, ~SW[13], sw13_pressed);
 key_detect  sw14 (clk, reset, ~SW[14], sw14_pressed);
 key_detect  sw15 (clk, reset, ~SW[15], sw15_pressed);
 key_detect  sw16 (clk, reset, ~SW[16], sw16_pressed);
 key_detect  sw17 (clk, reset, ~SW[17], sw17_pressed);

//KEY Status Register Write
always @(posedge clk)
  begin
       if(reset)    KEY_StatusR  <= 0;
      else
           begin
          //KEY
           if (~CS_N && ~RD_N && Addr[11:0] == 12'h000)
               KEY_StatusR  <= 0;
          else
              begin
                  if(key1_pressed)   KEY_StatusR[1] <= 1'b1;
                  if(key2_pressed)   KEY_StatusR[2] <= 1'b1;
                  if(key3_pressed)   KEY_StatusR[3] <= 1'b1;
              end
          end
   end
```

```
//write output Register
  always @(posedge clk)
     begin
        if(reset)
            begin
                LEDR_R[16:0] <=17'b0;
                LEDG_R[8:0]  <=9'h1FF;
                HEX0_R <= 7'b1000000;
                HEX1_R <= 7'b1000000;
                HEX2_R <= 7'b1000000;
                HEX3_R <= 7'b1000000;
                HEX4_R <= 7'b1000000;
                HEX5_R <= 7'b1000000;
                HEX6_R <= 7'b1000000;
                 HEX7_R <= 7'b1000000;
            end

        else if(~CS_N && ~WR_N)
            begin
                if      (Addr[11:0] == 12'h008)  LEDR_R        <= DataIn;
                else if  (Addr[11:0] == 12'h00C)  LEDG_R       <= DataIn;
                else if  (Addr[11:0] == 12'h010)  HEX0_R       <= DataIn;
                else if  (Addr[11:0] == 12'h014)  HEX1_R       <= DataIn;
                else if  (Addr[11:0] == 12'h018)  HEX2_R       <= DataIn;
                else if  (Addr[11:0] == 12'h01C)  HEX3_R       <= DataIn;
                else if  (Addr[11:0] == 12'h020)  HEX4_R       <= DataIn;
                else if  (Addr[11:0] == 12'h024)  HEX5_R       <= DataIn;
                else if  (Addr[11:0] == 12'h028)  HEX6_R       <= DataIn;
                else if  (Addr[11:0] == 12'h02C)  HEX7_R       <= DataIn;
            end
    end

    //output
    assign LEDR[16:0] =  LEDR_R[16:0];
    assign LEDG[8:0]  =  LEDG_R[8:0];
    assign HEX0       =  HEX0_R[6:0];
    assign HEX1       =  HEX1_R[6:0];
    assign HEX2       =  HEX2_R[6:0];
    assign HEX3       =  HEX3_R[6:0];
    assign HEX4       =  HEX4_R[6:0];
    assign HEX5       =  HEX5_R[6:0];
    assign HEX6       =  HEX6_R[6:0];
    assign HEX7       =  HEX7_R[6:0];
```

```
module MIPS_System(
  input             CLOCK_50,
  input[3:0]        KEY,
  input[17:0]       SW,
  output [6:0]      HEX7,
  output [6:0]      HEX6,
  output [6:0]      HEX5,
  output [6:0]      HEX4,
  output [6:0]      HEX3,
  output [6:0]      HEX2,
  output [6:0]      HEX1,
  output [6:0]      HEX0,
  output [17:0]     LEDR,
  output [8:0]      LEDG);

  wire              reset;
  wire              reset_poweron;
  reg               reset_ff;
  wire              clk0;
  wire              locked;
  wire [31:0]           inst_addr;
  wire [31:0]           inst;
  wire [31:0]           data_addr;
  wire [31:0]           write_data;
  wire [31:0]           read_data_timer;
  wire [31:0]           read_data_uart;
  wire [31:0]           read_data_gpio;
  wire [31:0]           read_data_mem;
  reg  [31:0]       read_data;
  wire              cs_mem_n;
  wire              cs_timer_n;
  wire              cs_gpio_n;
  wire              data_we;

  wire              clk90;
  wire              clk180;
  wire              data_re;

  // reset =  KEY[0]
  // if KEY[0] is pressed, the reset goes down to "0"
  // reset is a low-active signal
  assign  reset_poweron = KEY[0];
  assign  reset = reset_poweron & locked;

  always @(posedge clk0)  reset_ff <= reset;

  ALTPLL_clkgen pll0(
              .inclk0   (CLOCK_50),
              .c0       (clk0),
              .c1       (clk90),
              .c2       (clk180),
              .locked   (locked));
```

```
always @(*)
begin
    if     (~cs_timer_n) read_data <= read_data_timer;
    else if (~cs_gpio_n)  read_data <= read_data_gpio;
    else                  read_data <= read_data_mem;
end


mips mips_cpu (
       .clk              (clk0),
       .reset            (~reset_ff),
       .pc               (inst_addr),
       .instr            (inst),
       .memwrite         (data_we),  // data_we: active high
       .memaddr          (data_addr),
       .memwritedata     (write_data),
       .memreaddata      (read_data));

  assign data_re = ~data_we;



  // Port A: Instruction
  // Port B: Data
 ram2port_inst_data Inst_Data_Mem (
       .address_a   (inst_addr[12:2]),
       .address_b   (data_addr[12:2]),
       .byteena_b   (4'b1111),
       .clock_a     (clk90),        //was clk90
       .clock_b     (clk180),       //was clk180
       .data_a      (),
       .data_b      (write_data),
       .enable_a    (1'b1),
       .enable_b    (~cs_mem_n),
       .wren_a      (1'b0),
       .wren_b      (data_we),
       .q_a         (inst),
       .q_b         (read_data_mem));


Addr_Decoder Decoder (
        .Addr          (data_addr),
    .CS_MEM_N      (cs_mem_n) ,
    .CS_TC_N       (cs_timer_n),
    .CS_UART_N     (),
    .CS_GPIO_N     (cs_gpio_n));
```

```
  TimerCounter Timer (
        .clk        (clk0),
        .reset      (~reset_ff),
        .CS_N     (cs_timer_n),
        .RD_N     (~data_re),
        .WR_N     (~data_we),
        .Addr       (data_addr[11:0]),
        .DataIn     (write_data),
        .DataOut (read_data_timer),
        .Intr       ());


    GPIO uGPIO (
        .clk        (clk0),
        .reset      (~reset_ff),
        .CS_N      (cs_gpio_n),
        .RD_N     (~data_re),
        .WR_N    (~data_we),
        .Addr       (data_addr[11:0]),
        .DataIn     (write_data),
        .DataOut (read_data_gpio),
        .Intr       (),

        .KEY        (KEY[3:1]),
        .SW         (SW),
        .HEX7     (HEX7),
        .HEX6     (HEX6),
        .HEX5     (HEX5),
        .HEX4     (HEX4),
        .HEX3     (HEX3),
        .HEX2     (HEX2),
        .HEX1     (HEX1),
        .HEX0     (HEX0),
        .LEDR     ({LEDR[17:6],LEDR[4:0]}),
        .LEDG     (LEDG));


      PWM uPWM(
        .clk        (clk0),
        .reset      (~reset_ff),
        .CS_N     (cs_mem_n),
        .WR_N    (~data_we),
        .Addr       (data_addr[11:0]),
        .DataIn     (write_data),

        .LEDR5   (LEDR[5]));

endmodule
```