ELEC473, ASSIGNMENT 3 MIPS PROCESSOR

Ben Hague - 201146260

March 13, 2020

The University of Liverpool

CONTENTS

Introduction 2
1 Part A 2
1.1 Program: 2
1.2 mips data file 3
1.3 Results and testing 4
2 Part B 4
2.1 Adding the XOR and xori instruction 4
2.2 ASM 6
2.3 VERILOG 9
2.3.1 alu decoder 9
2.3.2 ALU 9
2.3.3 Main Decoder 10
2.4 Testing the XOR instruction 10
2.4.1 Program 10
2.4.2 Results 11
2.5 Testing the XORI instruction 11
2.5.1 Program 11
2.5.2 Results 12
2.6 Adding the LBU instruction 12
2.6.1 Verilog 14
2.6.2 Program 14
2.6.3 Results 15
3 Part C 16

INTRODUCTION

This task is designed to introduce the single cycle MIPS Processor, writing programs for it and implementing additional instructions.

Part A introduces creating a program to display output on the Hex displays on the DE2 board

Part B introduces 3 additional instructions to the processor

Part C introduces hardware defined PWM.

1 PART A

The methodology for this part is simple, Calculate the value written to memory for each hex value to be displayed. Sort them from smallest to largest and assign the values to each hex display in order of size.

To begin we must calculate the encoded hex value for each number in the student id (01146260) and the memory offset (where in the id it is) this is shown in Table 1.

TABLE 1, STUDENT ID MEMORY OFFSET AND DISPLAY VALUES

Number	Hex Memory Offset	Hex Display Value
0	0, 28 (0x0, 0x1C	79
1	20, 24 (0x14, 0x18)	40
2	8 (0x8)	24
4	16 (0x10)	19
6	4, 12 (0x4, 0xC)	02

1.1 PROGRAM:

```
# Ensure registers read 0 at the start
lui $2, 0x0000
lui $1, 0x0000
# Generate Start address
lui $2, 0xFFFF
addiu $2, $2, 0x2010

# Generate 6s (02) 0 + 02
addiu $1, $1, 0x02
# Place 6s
sw  $1, 0x4($2)
sw  $1, 0xC($2)

# Generate 4s (19) 02+17
addiu $1, $1, 0x17
# Place 4s
sw  $1, 0x10($2)

# Generate 2s (24) 19+b
addiu $1, $1, 0x0b
# Place 2s
sw  $1, 0x8($2)

# Generate 0s (40) 24+1C
addiu $1, $1, 0x1C # start bit
```

```
# Place 0s
sw  $1, 0x0($2)
sw  $1, 0x1C($2)

# Generate 1s (79) 40+39
addiu $1, $1, 0x39
# Place 1s
sw  $1, 0x18($2) # Place digit 3
sw  $1, 0x14($2) # Place digit 4

# Define loop to maintain Register Values
end:
j end
```

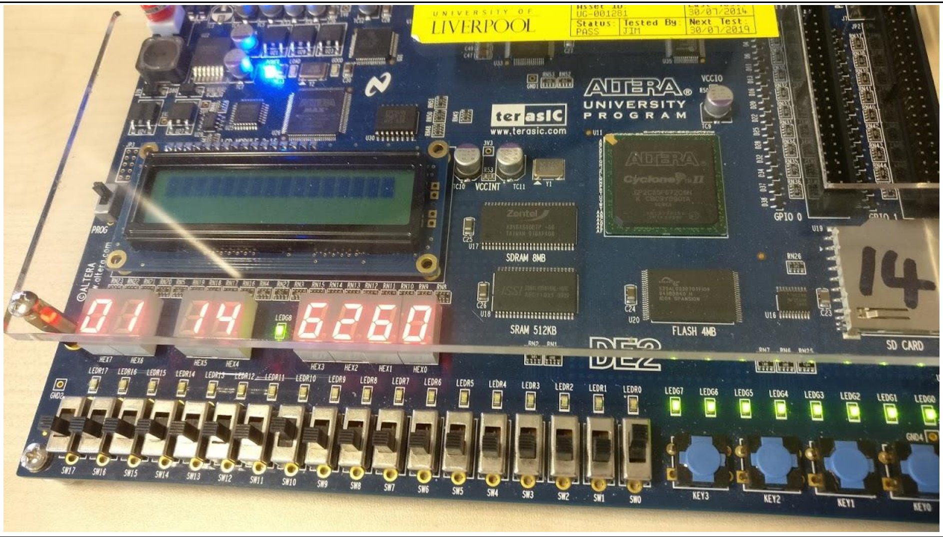
1.2 MIPS DATA FILE

```
-- Copyright (C) 1991-2013 Altera Corporation
-- Your use of Altera Corporation's design tools, logic functions
-- and other software and tools, and its AMPP partner logic
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Altera Program License
-- Subscription Agreement, Altera MegaCore Function License
-- Agreement, or other applicable license agreement, including,
-- without limitation, that your use is for the sole purpose of
-- programming logic devices manufactured by Altera and sold by
-- Altera or its authorized distributors.  Please refer to the
-- applicable agreement for further details.

-- Quartus II generated Memory Initialization File (.mif)

WIDTH=32;
DEPTH=2048;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
	000 : 3c020000;
	001 : 3c010000;
	002 : 3c02ffff;
	003 : 24422010;
	004 : 24210002;
	005 : ac410004;
	006 : ac41000c;
	007 : 24210017;
	008 : ac410010;
	009 : 2421000b;
	00a : ac410008;
	00b : 2421001c;
	00c : ac410000;
	00d : ac41001c;
	00e : 24210039;
	00f : ac410018;
	010 : ac410014;
	011 : 08100011;
	[012..7FF]  :   00000000;
```

```
END;
```

1.3 RESULTS AND TESTING

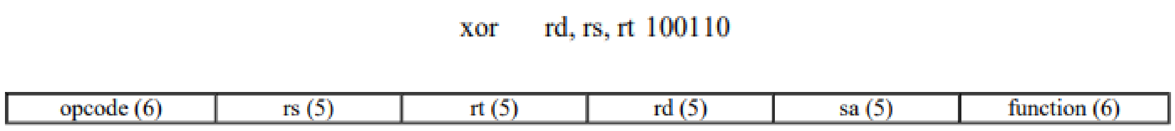
Whilst I don't have the signal tap output available for this module, Below shows the result as executed on the DE2 board.



2 PART B

2.1 ADDING THE XOR AND XORI INSTRUCTION

The details for the XOR instruction are shown in Figure 1.

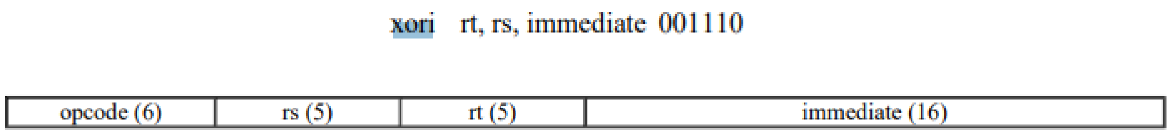


xor rd, rs, rt 100110

opcode (6)	rs (5)	rt (5)	rd (5)	sa (5)	function (6)

FIGURE 1, INSTRUCTION LAYOUT FOR XOR

The details for the XORI instruction are shown below



xori rt, rs, immediate 001110

opcode (6)	rs (5)	rt (5)	immediate (16)

FIGURE 2, INSTRUCTION LAYOUT FOR XORI

The XOR instruction is an R type instruction, This means it only operates on registers.

The ALU must be expanded to include an XOR instruction. The control signal for the ALU is currently 2 bits which are fully satisfied, This must be expanded to 3 bits.

The ALU decoder must be expanded to include the function required by the R type instruction.

For the XORI instruction we must then expand the Main decoder to give the ability to interpret the XORI opcode. As shown below

```
 Signext      = 0
 shiftl16     = 0
 regwrite     = 1
 regdst       = 0
 alusrc       = 1
 branch       = 0
 memwrite     = 0
 memtoreg     = 0
 jump         = 0
aluop        = 011
```

The control signals can then be combined to form the 12 bit signal 001010000011. The opcode for Xori must set the control signal to these values. Note that the step to expand XOR have already taken place before this is implemented.

The ALUOP signal must be expanded from 2 -> 3 bits to allow selection of the XOR operation in the ALU this change is implemented within the alu decoder.

2.2 ASM

Figure 3 and Figure 4 show the ASMs for the ALU and ALU controller

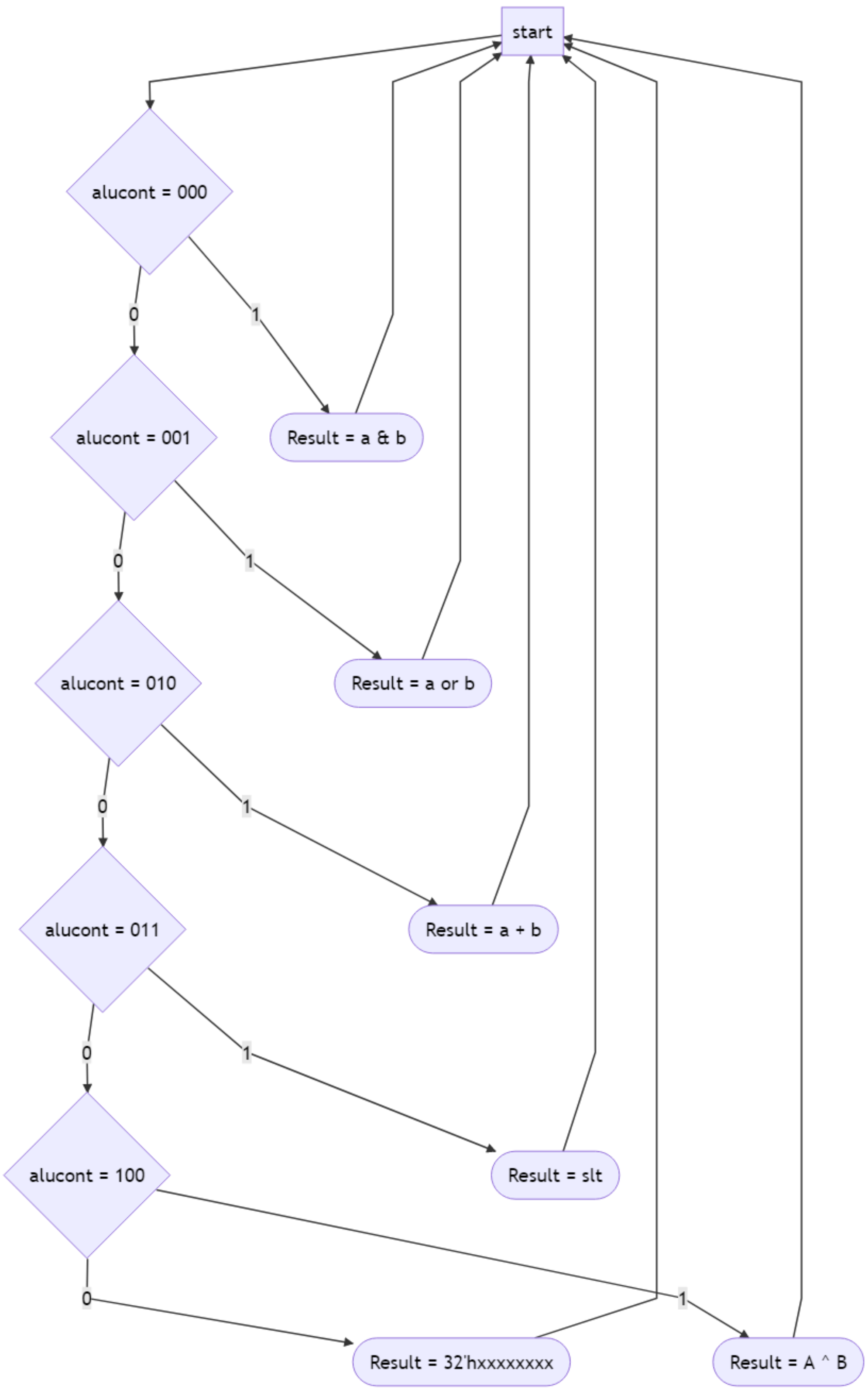


FIGURE 3, ALU ASM

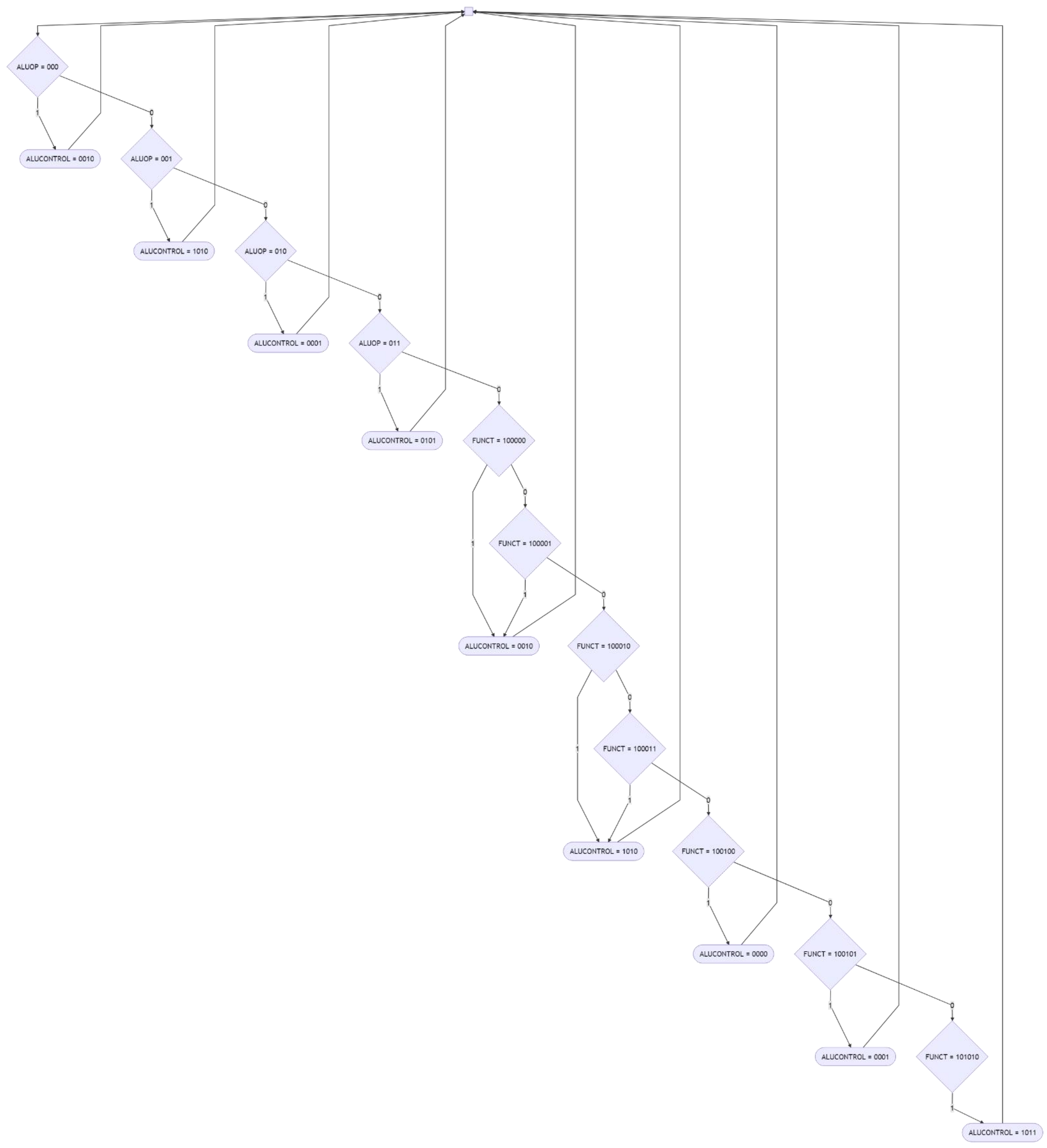


FIGURE 4, ALU DECODER ASM

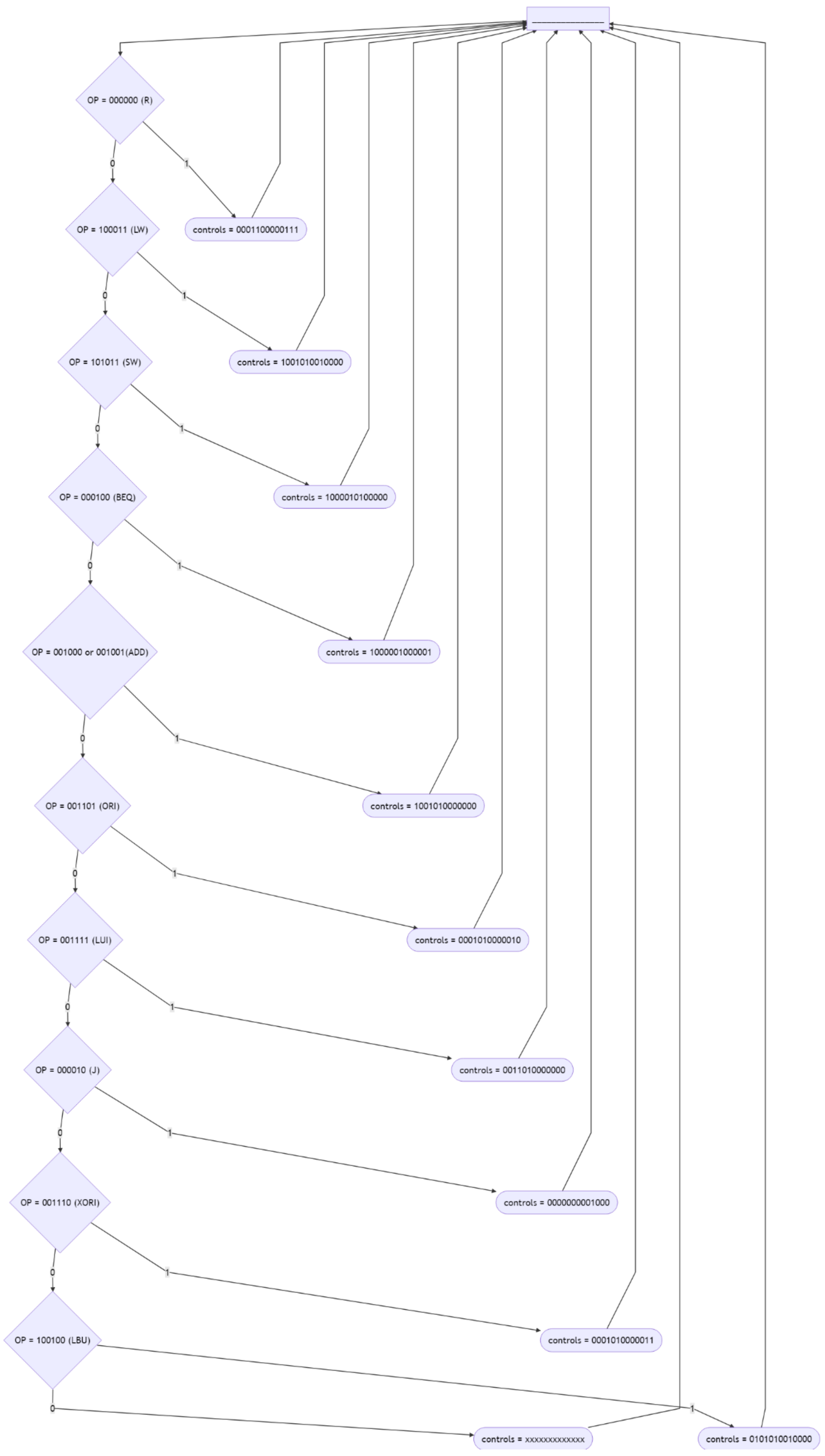


FIGURE 5, MAIN DECODER ASM

2.3 VERILOG

2.3.1 ALU DECODER

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

  always @(*)
    case(aluop)
             //BH2020 aluop expanded from 2->3
      3'b000: alucontrol <= 4'b0010;  // add
      3'b001: alucontrol <= 4'b1010;  // sub
      3'b010: alucontrol <= 4'b0001;  // or
      3'b011: alucontrol <= 4'b0101;  // xor BH2020
      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
          // Here we add the additional instructions BH2020
          // Used commands (010, 110, 000, 001, 111)
          // Commands Used By me (101 xor),
          // Unused commands (100, 011)
          // Commands are 3 least significant bits!
          6'b100110: alucontrol <= 4'b0101;
          default:   alucontrol <= 4'bxxxx; // Define the defauly and prevent
latches
        endcase
    endcase
endmodule
```

2.3.2 ALU

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

  wire [31:0] b2, sum, slt;

  assign b2 = alucont[3] ? ~b:b;  //. add or subtract is in MSB
  assign sum = a + b2 + alucont[3];
  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;
             // Expand alucont from 3 width to 4 BH2020
             // Add xor to commands
             3'b100: result <= a^b;
```

```
            default: result <= 32'hxxxxxxxx;
    endcase

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

endmodule
```

2.3.3 Main Decoder

```
module maindec(input  [5:0] op,
               output       signext,
               output       loadb, //BH2020
               output       shiftl16,
               output       memtoreg, memwrite,
               output       branch, alusrc,
               output       regdst, regwrite,
               output       jump,
               output [2:0] aluop);

  reg [12:0] controls;

  assign {signext,loadb, shiftl16, regwrite, regdst,
                     alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls; // BH2020

  always @(*)
    case(op)
      6'b000000: controls <= 13'b0001100000111; // Rtype
      6'b100011: controls <= 13'b1001010010000; // LW
      6'b101011: controls <= 13'b1000010100000; // SW
      6'b000100: controls <= 13'b1000001000001; // BEQ
      6'b001000,
      6'b001001: controls <= 13'b1001010000000; // ADDI, ADDIU: only difference is
exception
      6'b001101: controls <= 13'b0001010000010; // ORI
      6'b001111: controls <= 13'b0011010000000; // LUI
      6'b000010: controls <= 13'b0000000001000; // J
       // BH2020
       // Add Xori command
       6'b001110: controls <= 13'b0001010000011; // XORI
       // Add command for LBU
       6'b100100: controls <= 13'b0101010010000; // LBU
       default:   controls <= 13'bxxxxxxxxxxxxx; // prevent latches
    endcase

endmodule
```

2.4 TESTING THE XOR INSTRUCTION

2.4.1 Program

To test this instruction, we load data into the upper bits of register 1 and 2, The result is then loaded into register 3. This is shown below

```
# Ensure registers read known values at the start
lui    $2     0x5556
lui    $1,    0x9AAA
lui    $3,    0x9AAA
# Test the XOR function
```

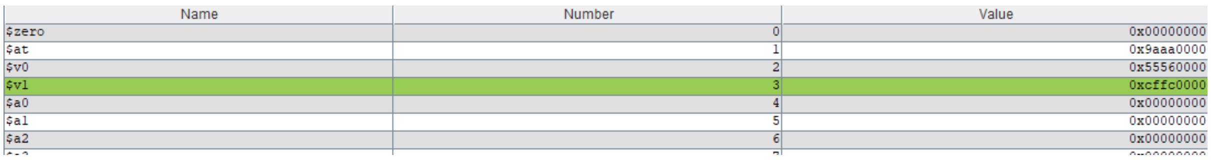
```
xor    $3     $1     $2
# Define loop to maintain Register Values
end:
j end
```

2.4.1.1 HEX FILE

```
-- Quartus II generated Memory Initialization File (.mif)

WIDTH=32;
DEPTH=2048;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
	000 : 3c025556;
	001 : 3c019aaa;
	002 : 3c039aaa;
	003 : 00221826;
	004 : 08100004;
	[005..7FF]  :   00000000;
END;
```

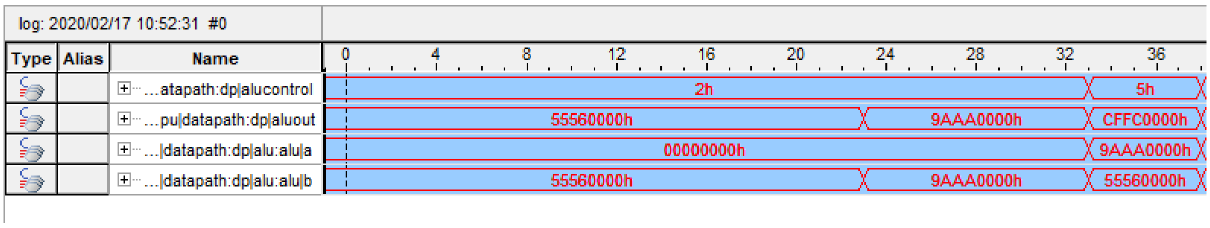
2.4.2 RESULTS

The outcome of stepping through the instructions individually is shown here



Name	Number	Value
$zero	0	0x00000000
$at	1	0x9aaa0000
$v0	2	0x55560000
$v1	3	0xcffc0000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000

The ALU inputs a and b are shown here, alongside ALU control and ALU output


log: 2020/02/17 10:52:31 #0
Type
Alias
Name
0 4 8 12 16 20 24 28 32 36
...atapath:dp|alucontrol 2h 5h
...pu|datapath:dp|aluout 55560000h 9AAA0000h CFFC0000h
...|datapath:dp|alu:alu|a 00000000h 9AAA0000h
...|datapath:dp|alu:alu|b 55560000h 9AAA0000h 55560000h

We can observe that this works as intended.

0x9AAA =	*1001101010101010*
0x5556 =	*0101010101010110*
0xCFFC =	*1100111111111100*

2.5 TESTING THE XORI INSTRUCTION

2.5.1 PROGRAM

```
# Ensure registers read 0 at the start
lui    $1,    0x0000
lui    $3,    0x0000
addiu  $1,    $3     0x9AAA
# Test the XOR function
xori   $3     $1     0x5556
# Define loop to maintain Register Values
end:
```

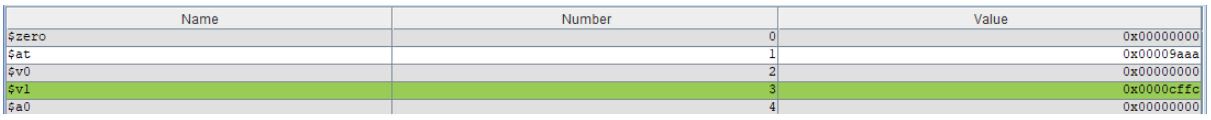
```
j end
```

2.5.1.1 HEX FILE

```
-- Quartus II generated Memory Initialization File (.mif)

WIDTH=32;
DEPTH=2048;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
	000 : 3c010000;
	001 : 3c030000;
	002 : 3c010000;
	003 : 34219aaa;
	004 : 00610821;
	005 : 38235556;
	006 : 08100006;
	[007..7FF]  :   00000000;
END;
```

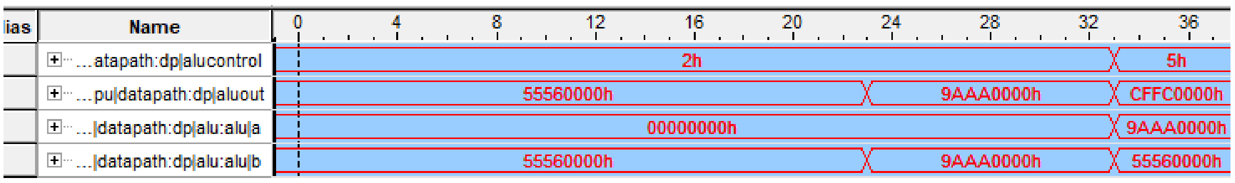
2.5.2 RESULTS

The outcome of stepping through the instructions individually is shown here



Name	Number	Value
$zero	0	0x00000000
$at	1	0x00009aaa
$v0	2	0x00000000
$v1	3	0x0000cffc
$a0	4	0x00000000

The ALU inputs a and b are shown here, alongside ALU control and ALU output

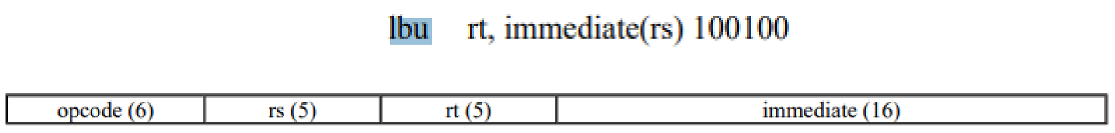


We can observe that this works as intended.

0x9AAA=	*1001101010101010*
0x5556 =	*0101010101010110*
0xCFFC =	*1100111111111100*

2.6 ADDING THE LBU INSTRUCTION

The details for the LBU instruction are shown below.



lbu rt, immediate(rs) 100100

opcode (6)	rs (5)	rt (5)	immediate (16)

FIGURE 6, INSTRUCTION LAYOUT FOR LBU

The load byte signal requires additional logic to determine which byte is selected, and to extend the byte to 32 bits matching the rest of the system. The selector takes the form of a 4 way MUX.

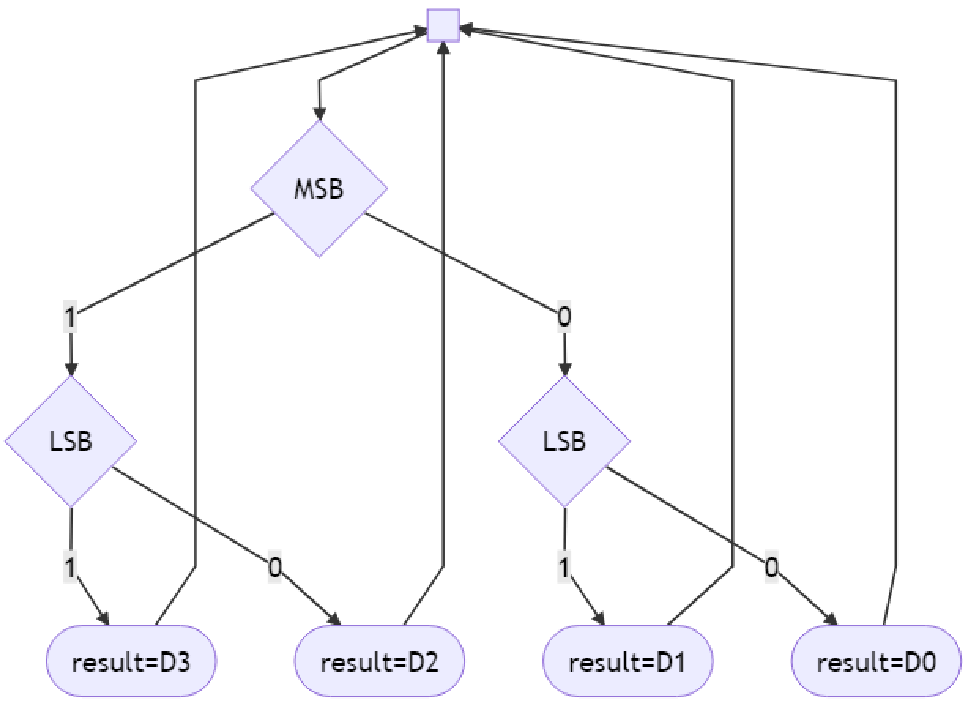


FIGURE 7, 4 WAY MUX ASM

The Extender is simple matter of assigning location on the result of the 4 way MUX

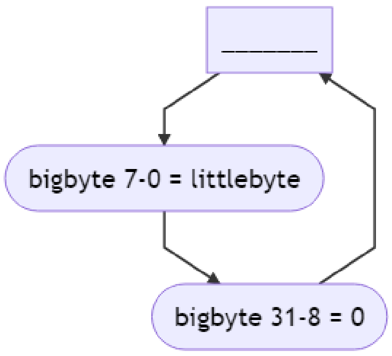


FIGURE 8, 32 BIT EXTENDER ASM

An Additional 2 way multiplexer is needed to select if the output from this logic path is used. The control signal from this is introduced above as a loadb signal (This is shown in the earlier ASMs)

The Datapath with these amendments is shown in Figure 9.

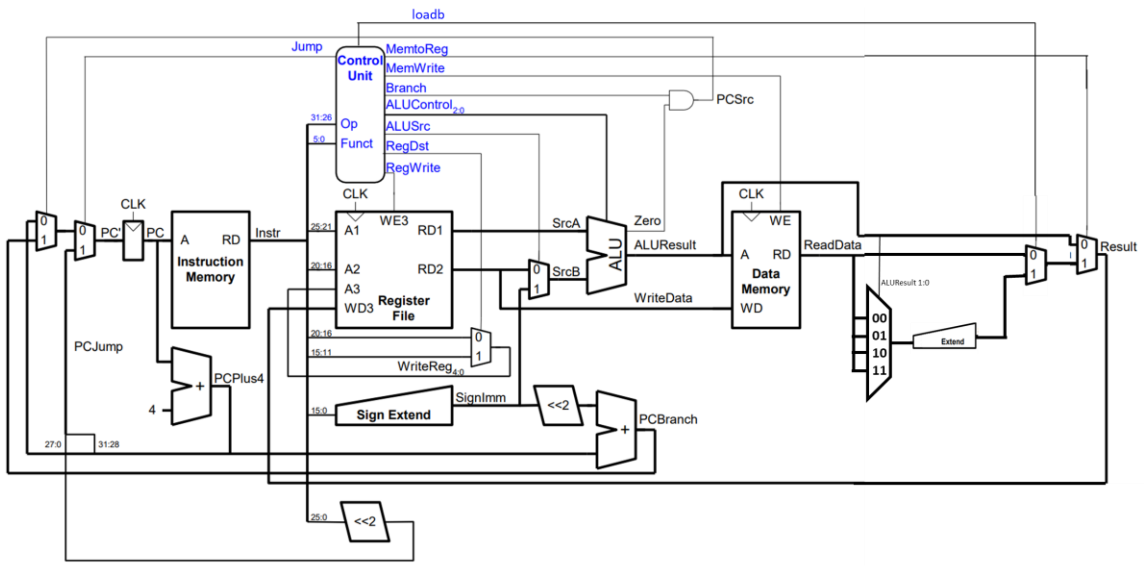


FIGURE 9, DATAPATH WITH LBU INSTRUCTION

2.6.1 VERILOG

2.6.1.1 MUX 4

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

  assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);

endmodule
```

2.6.1.2 EXTENDER

```
module extend(input      [7:0] littlebyte,
              output     [31:0] bigbyte);

   assign bigbyte = {{24{1'b0}}, littlebyte[7:0]};

endmodule
```

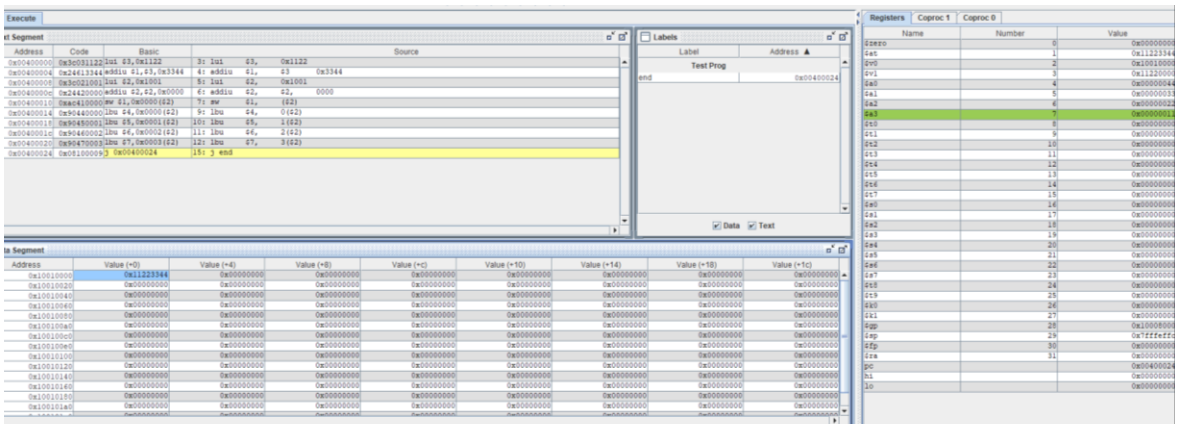
2.6.2 PROGRAM

```
# Load 11223344 to a known memory location
lui    $3,    0x1122
addiu  $1,    $3     0x3344
lui    $2,    0x2008
addiu  $2,    $2,    0000
sw     $1,    ($2)
# Test the LBU function
lbu    $4,    0($2)
lbu    $5,    1($2)
lbu    $6,    2($2)
lbu    $7,    3($2)
# Define loop to maintain Register Values
end:
j end
```

2.6.2.1 HEX FILE

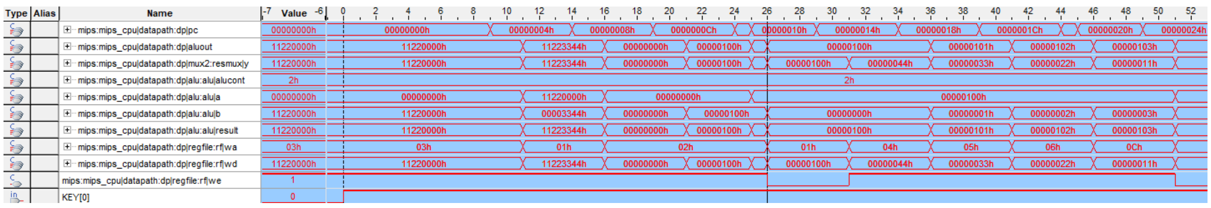
```
-- Quartus II generated Memory Initialization File (.mif)

WIDTH=32;
DEPTH=2048;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
       000 : 3c031122;
       001 : 24613344;
       002 : 3c022008;
       003 : 24420000;
       004 : ac410000;
       005 : 90440000;
       006 : 90450001;
       007 : 90460002;
       008 : 90470003;
       009 : 08100009;
       [00A..7FF]  :   00000000;
END;
```

2.6.3 RESULTS



```
Address     Code        Basic                      Source
0x00400000  0x3c031122  lui $3,0x1122              3: lui    $3,   0x1122
0x00400004  0x24613344  addiu $1,$3,0x3344         4: addiu  $1,   $3      0x3344
0x00400008  0x3c021001  lui $2,0x1001              5: lui    $2,   0x1001
0x0040000c  0x24420000  addiu $2,$2,0x0000         6: addiu  $2,   $2,     0000
0x00400010  0xac410000  sw $1,0x0000($2)           7: sw     $1,   ($2)
0x00400014  0x90440000  lbu $4,0x0000($2)          9: lbu    $4,   0($2)
0x00400018  0x90450001  lbu $5,0x0001($2)         10: lbu    $5,   1($2)
0x0040001c  0x90460002  lbu $6,0x0002($2)         11: lbu    $6,   2($2)
0x00400020  0x90470003  lbu $7,0x0003($2)         12: lbu    $7,   3($2)
0x00400024  0x08100009  j 0x00400024              15: j end
```

Simulated step through of program showing register values etc.



```
Name                                         Value
mips:mips_cpu|datapath:dp|pc                 00000000h  00000000h 00000004h 00000008h 0000000Ch 00000010h 00000014h 00000018h 0000001Ch 00000020h 00000024h
mips:mips_cpu|datapath:dp|aluout             11220000h  11220000h 11223344h 00000000h 00000100h 00000100h 00000101h 00000102h 00000103h
mips:mips_cpu|datapath:dp|mux2:resmux|y      11220000h  11220000h 11223344h 00000000h 00000100h 00000100h 00000044h 00000033h 00000022h 00000011h
mips:mips_cpu|datapath:dp|alu:alu|alucont    2h         2h
mips:mips_cpu|datapath:dp|alu:alu|a          00000000h  00000000h 11220000h 00000000h 00000100h
mips:mips_cpu|datapath:dp|alu:alu|b          11220000h  11220000h 00003344h 00000000h 00000100h 00000000h 00000001h 00000002h 00000003h
mips:mips_cpu|datapath:dp|alu:alu|result     11220000h  11220000h 11223344h 00000000h 00000100h 00000100h 00000101h 00000102h 00000103h
mips:mips_cpu|datapath:dp|regfile:rf|wa      03h        03h 01h 02h 01h 04h 05h 06h 0Ch
mips:mips_cpu|datapath:dp|regfile:rf|wd      11220000h  11220000h 11223344h 00000000h 00000100h 00000100h 00000044h 00000033h 00000022h 00000011h
mips:mips_cpu|datapath:dp|regfile:rf|we      1
KEY[0]                                       0
```

We can observe the loading of bytes in order 44 33 22 11, therefore proving the correctness of this instruction.

3 PART C

Part C has been tested, and confirmed working on the DE2 board, however due to the issues relating to COVID 19 no evidence for this exists. Over the time period it has proved that using modelsim has been a difficult challenge. It is extremely difficult to fully verify the function of this module without the visual verification of the fading LED it is also difficult to simulate this module at a suitable speed which allows it to function fully. Whilst demonstrated working on the DE2 board pre COVID, it has not been functioning in simulation.

The methodology for this implementation follows the structure introduced by the timer module. The PWM module will occupy memory addresses beginning 0XFFF3 . To implement this, a control signal has been added to the memory decoder.

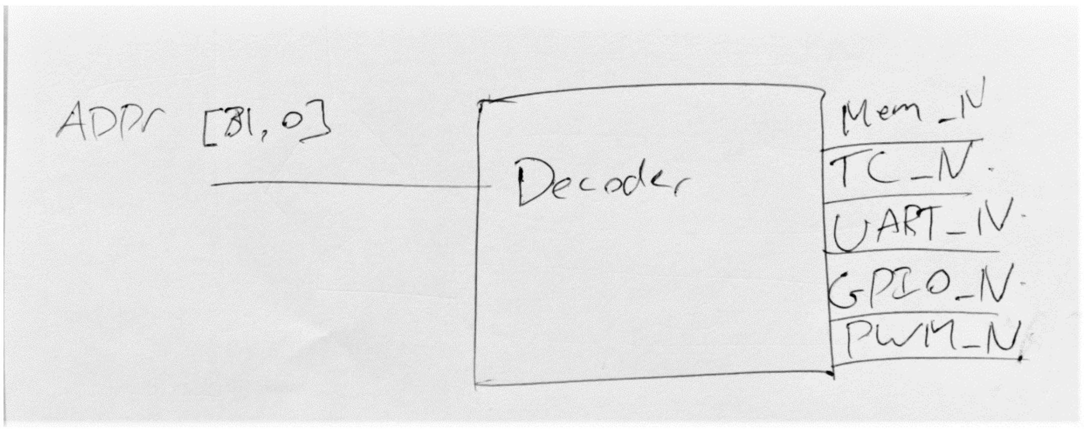


FIGURE 10, DECODER BLOCK DIAGRAM

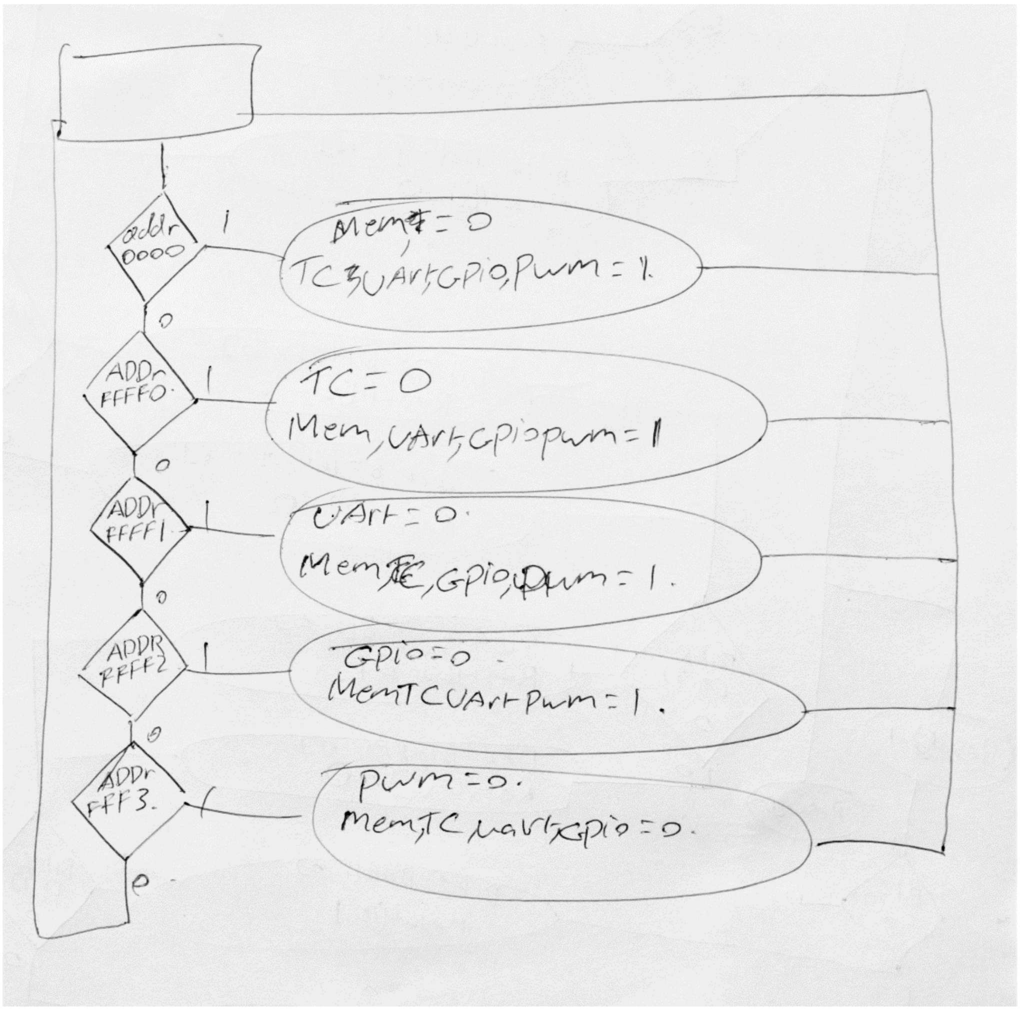


FIGURE 11, DECODER ASM

```
module Addr_Decoder (input [31:0] Addr,
                     output reg CS_MEM_N,
                     output reg CS_TC_N,
                     output reg CS_UART_N,
                     output reg CS_GPIO_N,
                                              output reg CS_PWM_N); //BH2020
CS_PWM_N

//======================================================================
//  Address      Peripheral     Peripheral Name              Size
// OxFFFF_FFFF  -------------
//
//                Reserved
//
// OxFFFF_3000  -------------
//                 GPIO         General Purpose IO            4KB
// OxFFFF_2000  -------------
//                              Universal
//                  UART        Asynchronous                  4KB
//                              Receive/ Transmitter
// OxFFFF_1000  -------------
//                  TC          Timer Conter                  4KB
// OxFFFF_0000  -------------
//
```

```
//                 Reserved
//
// Ox0000_2000  -------------
//                   mem        Instruction & Data Memory      8KB
// Ox0000_0000  -------------
//========================================================================

always @(*)
       begin
              if            (Addr[31:13] == 19'h0000)        // Instruction & Data
Memory
                     begin
                            CS_MEM_N            <=0;
                            CS_TC_N             <=1;
                            CS_UART_N    <=1;
                            CS_GPIO_N    <=1;
                            CS_PWM_N            <=1;
                     end

              else if       (Addr[31:12] == 20'hFFFF0)// Timer
                     begin
                            CS_MEM_N            <=1;
                            CS_TC_N             <=0;
                            CS_UART_N    <=1;
                            CS_GPIO_N    <=1;
                            CS_PWM_N            <=1;
                     end

              else if       (Addr[31:12] == 20'hFFFF1)// UART
                     begin
                            CS_MEM_N    <=1;
                            CS_TC_N     <=1;
                            CS_UART_N   <=0;
                            CS_GPIO_N   <=1;
                            CS_PWM_N            <=1;
                     end

              else if  (Addr[31:12] == 20'hFFFF2)     // GPIO
                     begin
                            CS_MEM_N    <=1;
                            CS_TC_N     <=1;
                            CS_UART_N   <=1;
                            CS_GPIO_N    <=0;
                            CS_PWM_N            <=1;
                     end
              else if  (Addr[31:12] == 20'hFFFF3)     // PWM
                     begin
                            CS_MEM_N    <=1;
                            CS_TC_N     <=1;
                            CS_UART_N   <=1;
                            CS_GPIO_N    <=1;
                            CS_PWM_N            <=0;
                     end

              else
       // Nothing selected
                     begin
                            CS_MEM_N     <=1;
                            CS_TC_N      <=1;
```

```
                                CS_UART_N    <=1;
                                CS_GPIO_N    <=1;
                                CS_PWM_N            <=1;
                        end
        end
endmodule
```

The next step is a PWM controller implementation. The PWM controller is a simple operation providing a wire with a modulated signal, it would be quick to expand this functionality to include a mask defining which pins are connected to the PWM module. In this example LEDR[5] is connected to the PWM signal. The Block diagram and ASM for the PWM module is shown below:

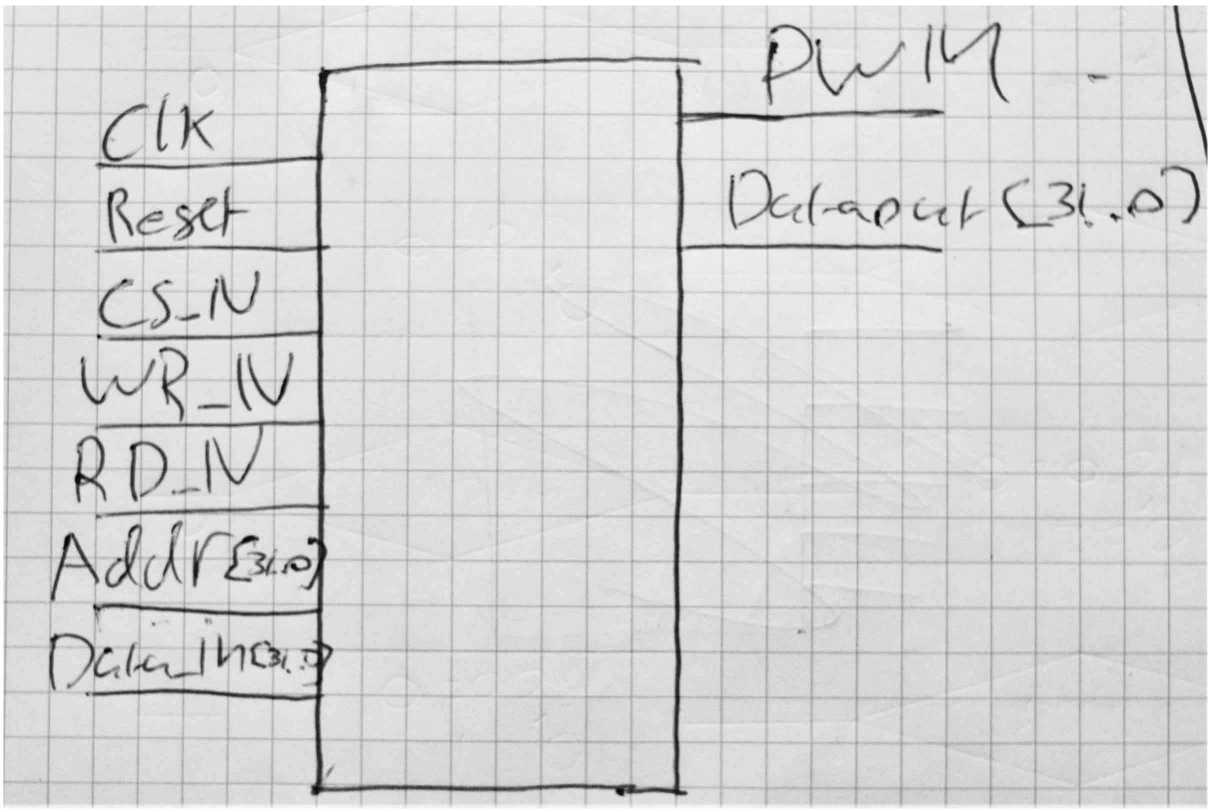


FIGURE 12, PWM BLOCK DIAGRAM

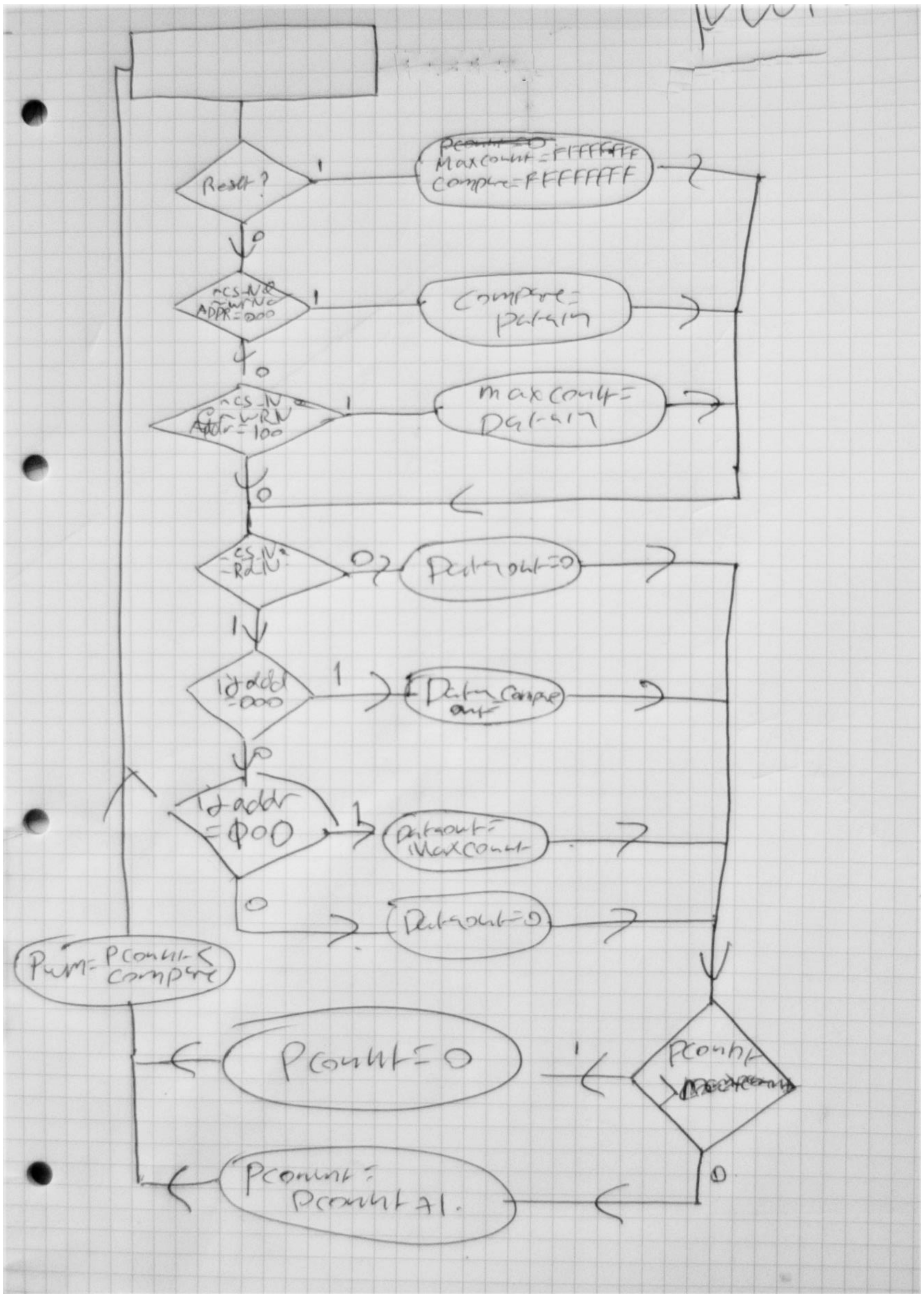


FIGURE 13, PWM ASM

The Verilog implementation is shown below:

```
module PWMcontrol(
  input clk,
  input reset,
  input CS_N,
  input WR_N,
  input RD_N,
  input [11:0] Addr,
  input [31:0] DataIn,
  output reg [31:0] DataOut,
  output PWM);
  // Variables
  reg [31:0] Compare;
  reg [31:0] MaxCount;
  // Counter status holders
  reg [31:0] pcount, ncount;

//  ==============================
//  PWM registers
//  ==============================
//  FFFF_0100 Frequency register   (MaxCount)(Write)
//  ------------------------------
//  FFFF_0000 Pulse Width register (Compare)(Write)
//  ==============================

  assign PWM = (Compare>pcount);
  // Compare Register Write
  always @(posedge clk)
  begin
    if(reset)
             begin
          Compare <=32'hFFFF_FFFF;
               MaxCount <=32'hFFFF_FFFF;
             end
      else if(~CS_N && ~WR_N && (Addr[11:0]==12'h000))
             begin
          Compare <= DataIn;
             end
      else if(~CS_N && ~WR_N && (Addr[11:0]==12'h100))
          MaxCount <= DataIn;
  end



  // Increment Counter in the Counter Register
  // Reset conditions: 1. reset 2. when the counter value is equal to the compare
register
  always @(posedge clk) // Reset the counter or increment by one on the clock.
  begin
    if(reset | pcount>MaxCount)   pcount <= 32'b0;
    else                      pcount <= ncount;
  end


  // Register Read
  always @(clk)
  begin
      ncount= pcount + 1;
    if(~CS_N && ~RD_N) // if read is high (not high (_N)) and CS is high
    begin
```

```
      if      (Addr[11:0] == 12'h000) DataOut <= Compare;
      else if (Addr[11:0] == 12'h100) DataOut <= MaxCount;
      else                            DataOut <= 32'b0;
     end
     else                             DataOut <= 32'b0;
  end

endmodule
```

Following this lead there are 2 variables for this module the maximum count register, and the compare register. The maximum count register defines the frequency, and the compare defines the pulse width.

The duty cycle of the module can be calculated with:

$$Duty\ cycle = \frac{Compare}{Max\ Count}$$

The frequency of the module (with a 50mhz clock as on the DE2 board) is defined with:

$$Frequency = \frac{\text{Max Count}}{50{,}000{,}000}$$

To perform preliminary testing of this module we will set the frequency to 20khz and the duty cycle to 50%. For this the Max count register will be set to 2500 and the compare to 1250 This is shown below:

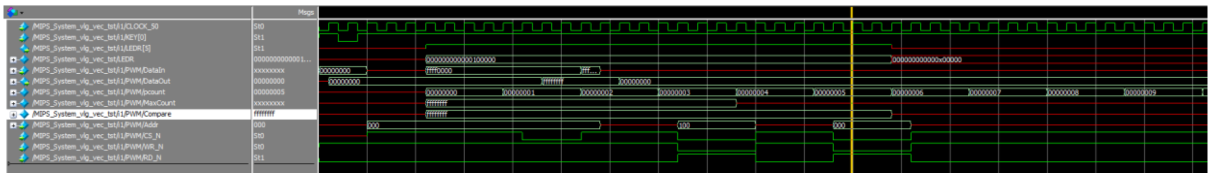
```
# Ben Hague, dimmer
# Reg 1 contains PWM address
# Reg 2 contains max counter
# Reg 3 contains Counter
# Reg 4 is the delay counter
# Reg 5 is the delay compare

# Reg 1 - PWM address
# Generate PWM address ffff3000
lui    $1,    0xFFFF
addiu  $1,    $1,    0x3000

# Reg 2 - Generate Max Counter
# max counter is 10000
addiu  $2,    $2,    500
# Store in the memory locaton
sw     $2,    0x100($1)

# Reg 5, store the delay count
addiu  $5,    $5,    250
sw     $5,    0x000($1)
holt:
```
j holt

The final code where the light slowly fades on and off:

```
# Ben Hague, dimmer
# Reg 1 contains PWM address
# Reg 2 contains max counter
# Reg 3 contains Counter
# Reg 4 is the delay counter
# Reg 5 is the delay compare

# Reg 1 - PWM address
# Generate PWM address ffff3000
lui    $1,    0xFFFF
addiu  $1,    $1,    0x3000

# Reg 2 - Generate Max Counter
# max counter is 10000
addiu  $2,    $2,    0xFFFF
# Store in the memory locaton
sw     $2,    0x100($1)

# Reg 5, store the delay count
addiu  $5,    $5,    0xFFF


Increment:
li     $4,    0
addiu  $3,    $3,    0x8
# Store in the memory location for pulse
sw     $3,    0x000($1)
beq    $2,    $3,    Decrement
j      IncrementDelay

Decrement:
li     $4,    0
# max counter is 10000
addi   $3,    $3,    -8
# Store in the memory locaton for pulse
sw     $3,    0x000($1)
beqz   $3,    Increment
j      DecrementDelay

IncrementDelay:
addiu  $4,    $4,    1
beq    $4,    $5,    Increment
j IncrementDelay

DecrementDelay:
addiu  $4,    $4,    1
beq    $4,    $5,    Decrement
j DecrementDelay
```

Therefore to verify the module works we must observe at least 2ms of execution time, during this time we would expect to see 2 periods of PWM modulation.

We can observe below how the module fails to successfully simulate



```
                                              Msgs
/MIPS_System_vlg_vec_tst/i1/CLOCK_50          St0
/MIPS_System_vlg_vec_tst/i1/KEY[0]            St1
/MIPS_System_vlg_vec_tst/i1/LEDR[5]           St1
/MIPS_System_vlg_vec_tst/LEDR                 0000000000001...
/MIPS_System_vlg_vec_tst/i1/PWM/DataIn        xxxxxxxx
/MIPS_System_vlg_vec_tst/i1/PWM/DataOut       00000000
/MIPS_System_vlg_vec_tst/i1/PWM/pcount        00000005
/MIPS_System_vlg_vec_tst/i1/PWM/MaxCount      xxxxxxxx
/MIPS_System_vlg_vec_tst/i1/PWM/Compare       ffffffff
/MIPS_System_vlg_vec_tst/i1/PWM/Addr          000
/MIPS_System_vlg_vec_tst/i1/PWM/CS_N          St0
/MIPS_System_vlg_vec_tst/i1/PWM/WR_N          St0
/MIPS_System_vlg_vec_tst/i1/PWM/RD_N          St1
```