Report lab 1

TSEA44

Jonathan Karlsson, Niclas Olofsson, Paul Nedstrand
jonka293, nicol271, paune415
Grupp 2

27 januari 2014

Innehåll

1 Lab 1 **3**
1.1 Introduction 3

2 Implementation **3**
2.1 Interface 3
2.2 How we tested our implementation 4
2.3 result of the interface implementation 4

3 Performance counter **4**
3.1 Result of performance counter 5

4 Conclusion **5**

5 Files **6**

Appendix **7**

A lab1uarttop.sv **7**

B perftop.sv **10**

C uarttb.sv **12**

D dctsw.c **14**

1 Lab 1

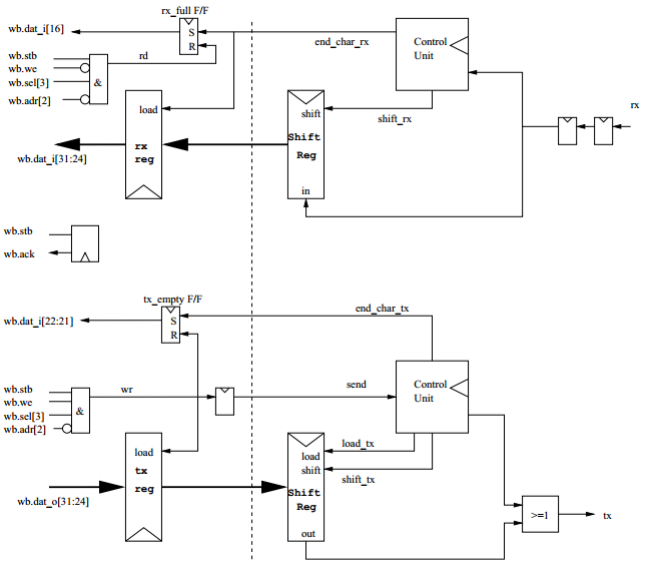
1.1 Introduction

In this lab we created an interface in System Verilog between a UART (The one we did in lab0) and the wishbone bus, so that we are able to communicate with the computer. The wishbone bus is a communication link between the OR1200, the boot monitor, The UART and the parallel port.
When we were done with the interface we did performance counters in System Verilog to measure the performance of our implementation with different cache settings.

2 Implementation

2.1 Interface

Our implementation of the interface is like the one in the lab compendium. We receive data from the rx channel. This is 8 bits plus a stop bit. We want to make the serial data to parallel which is the data that the Wish bone wants. We have a shift register and when we get a new bit, we shift that in the register. When we get the final bit, the stop bit, we push the data on the wishbone bus i.e we load the rx reg.
The same principle is made when transmitting (reading from the wb), then write signal (wr) = '1', i.e. when wb.stb = 1, wb.we = '1' wb.sel[3] = '1' and wb.adr[2] = '0'. The shift reg is loaded from the tx reg and then the bits are shifted out one by one in the specified data rate.



Figur 1: The block diagram of our architecture

2.2 How we tested our implementation

First we simulated our design with a test bench and when that was validated we tested our design by running a test program given to us called dctsw.c. The input we gave was a 64 long array, with the numbers 1-64. We took these because we knew what the output of this would be. The result is given at the home page.

2.3 result of the interface implementation

The following shows the result when a is [1 2 3 4 ... 63 64] as input to the DCT.
a=

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

8xDCT[a-128]=

-6112	-152	0	-16	0	-8	0	-8
-1167	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-122	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-37	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-10	0	0	0	0	0	0	0

RND(8xDCT[a-128]/(8xQx1/2)) =

-96	-3	0	0	0	0	0	0
-24	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

3 Performance counter

In this lab we also implemented four performance counters that are measuring the traffic on the WB. There are two counters that are counting the number of m0.ack and m1.ack we get on WB and two counters are counting how many (m0.ack and m0.stb) and (m1.ack and m1.stb) we get on the bus. These counters are saved at address x'99000000, x'99000004, x'99000008 and x'9900000C

respectively in the memory.
We looked at these addresses when running the test program with different cache settings. We ran it without any caches selected, with I-cache selected, with D-cache selected and with both selected. We saw that the performance was increasing with the caches selected i.e. the counters value decreased when caches were used i.e. there was less data traffic on the wishbone bus. We got the best result when both caches were selected. We did three runs with all the alternatives

3.1 Result of performance counter

ICache enabled and DCache enabled
Clk cycles for ctr1: 797, ctr2: 155, ctr3: 964, ctr4: 271
Clk cycles for ctr1: 367, ctr2: 76, ctr3: 480, ctr4: 159
Clk cycles for ctr1: 367, ctr2: 76, ctr3: 480, ctr4: 159

ICache enabled and DCache disabled
Clk cycles for ctr1: 802, ctr2: 162, ctr3: 989, ctr4: 309
Clk cycles for ctr1: 447, ctr2: 80, ctr3: 954, ctr4: 309
Clk cycles for ctr1: 446, ctr2: 80, ctr3: 955, ctr4: 309

ICache disabled and DCache enabled
Clk cycles for ctr1: 8639, ctr2: 1764, ctr3: 2254, ctr4: 270
Clk cycles for ctr1: 7887, ctr2: 1748, ctr3: 1127, ctr4: 158
Clk cycles for ctr1: 7887, ctr2: 1748, ctr3: 1127, ctr4: 158

ICache disabled and DCache disabled
Clk cycles for ctr1: 8728, ctr2: 1751, ctr3: 2533, ctr4: 309
Clk cycles for ctr1: 8728, ctr2: 1751, ctr3: 2533, ctr4: 309
Clk cycles for ctr1: 8728, ctr2: 1751, ctr3: 2533, ctr4: 309

4 Conclusion

In this lab we learned how to implement an interface to the wishbone bus and performance counters that measured the data traffic on the wishbone bus. We tested the interface with both a test bench and given test program called dctsw, with the image [1 2 3 4 ... 63 64]. When we validated the result, we created performance counters to count the amount of traffic we got on the bus with different cache settings. We saw that if we had no caches selected we got high traffic on the bus. When we had one cache selected the traffic on the bus decreased. We saw that when we enabled the I-cache the perforamance increased (avservärt) compared to when the D-cache was enabled. The best result (least amount of traffic) we got when we had both D-cache and I-cache was selected.

5 Files

lab1uarttop.sv In this file we implemented our interface between the WB and the UART.

perftop.sv In this file we implemented our performance counter.

uarttb.sv In this file we tested our hardware.

dctsw.c In this file we ran a software DCT to measure the traffic on the wish bone bus.

Appendix

A lab1uarttop.sv

Listing 1: Code lab1uarttop

```
`include "include/timescale.v"


module lab1_uart_top (
    wishbone.slave wb,
    output wire int_o,
    input wire  srx_pad_i,
    output wire stx_pad_o);

    assign int_o = 1'b0;  // Interrupt, not used in this lab
    assign wb.err = 1'b0; // Error, not used in this lab
    assign wb.rty = 1'b0; // Retry, not used in this course

    // Here you must instantiate lab0_uart or cut and paste
    // You will also have to change the interface of lab0_uart to make this work.


    reg rd;
    reg wr;
    reg wr_o;
    reg end_char;
    reg ack;
    reg uart_ack;
    reg rx_full;
    reg [7:0] tx_reg;
    reg [1:0] tx_empty;
    reg [7:0] last_8_rx_bits;
    reg [7:0] rx_reg;

    //the uart implemented in lab 0
    lab0 uart(wb.clk, wb.rst, srx_pad_i, stx_pad_o, last_8_rx_bits, tx_reg, wr_o, end_char, u


    //set the rd and wr accourdlingly to the block diagram
    always_comb
    begin
        if (wb.rst) begin
            rd <= 1'b0;
            wr <= 1'b0;
        end else begin
            rd <= wb.stb && ~wb.we && wb.sel[3] && ~wb.adr[2];
            wr <= wb.stb && wb.we && wb.sel[3] && ~wb.adr[2];
        end
```

```
    end

   //the rx_full and tx_empty F/F
   always @(posedge wb.clk)
    begin
        wr_o <= wr;
        //rx_full
        if (wb.rst) begin
            rx_full <= 1'b0;
            tx_empty <= 2'b11;
            tx_reg <= 8'b0;
        end else if (rd)
            rx_full <= 1'b0;
        else if (end_char)
            rx_full <= 1'b1;

        //tx_empty
        if (wr) begin
            tx_empty <= 2'b0;
            tx_reg <= wb.dat_o[31:24];
        end else if (uart_ack)
            tx_empty <= 2'b11;
    end

    //put data out to wishbone bus when we get an end_char
    always @(posedge wb.clk)
    begin
        if(end_char)
            rx_reg <= last_8_rx_bits;
    end

    //set ack
    always @(posedge wb.clk)
    begin
        if (wb.rst)
            ack <= 1'b0;
        else if (wb.stb)
            ack <= 1'b1;
        else
            ack <= 1'b0;
    end

    assign wb.ack = ack;
    assign wb.dat_i[31:24] = rx_reg;
    assign wb.dat_i[16] = rx_full;
    assign wb.dat_i[22:21] = tx_empty;

endmodule
```

```
// Local Variables:
// verilog-library-directories:("." ".." "../or1200" "../jpeg" "../pkmc" "../dvga" "../uart"
// End:
```

B perftop.sv

Listing 2: Code perftop

```
`include "include/timescale.v"


module perf_top (
    wishbone.slave wb,
    wishbone.monitor m0, m1);

    reg [31:0] counter1, counter2, counter3, counter4;
    reg [31:0] rx_reg;

    reg rd, wr, ack;

    always @(posedge wb.clk) begin
        if (wb.rst) begin
            //reset counters
            counter1 <= 32'b0;
            counter2 <= 32'b0;
            counter3 <= 32'b0;
            counter4 <= 32'b0;

        //set the counters accordinly to which address is pointed out.
        end else if (wr) begin

            if (wb.adr == 32'h99000000)
                counter1 <= wb.dat_o;

            else if (wb.adr == 32'h99000004)
                counter2 <= wb.dat_o;

            else if (wb.adr == 32'h99000008)
                counter3 <= wb.dat_o;

            else if (wb.adr == 32'h9900000c)
                counter4 <= wb.dat_o;

        end else begin
            //increase the counters if have m0.cyc && m0.stb or m1.cyc && m1.stb or m0.ack or
            if (m0.cyc && m0.stb)
                counter1 <= counter1 + 1;

            if (m0.ack)
                counter2 <= counter2 + 1;

            if (m1.cyc && m1.stb)
                counter3 <= counter3 + 1;
```

```
            if (m1.ack)
                counter4 <= counter4 + 1;
        end
    end

    always_comb begin
        if (wb.rst) begin
            rd <= 1'b0;
            wr <= 1'b0;
        end else begin
            rd <= wb.stb && ~wb.we;
            wr <= wb.stb && wb.we;
        end
    end

    always_comb begin
         //assign rx reg to the corresponding counter depending on which address is selected.
        case (wb.adr[3:2])
            2'b00:  rx_reg = counter1;
            2'b01:  rx_reg = counter2;
            2'b10:  rx_reg = counter3;
            2'b11:  rx_reg = counter4;
        endcase
    end


    //set ack
    always @(posedge wb.clk)
    begin
        if (wb.rst)
            ack <= 1'b0;
        else if (wb.stb)
            ack <= 1'b1;
        else
            ack <= 1'b0;
    end

    assign wb.ack = ack;
    assign wb.dat_i = rx_reg;
    assign wb.rty = 1'b0;
    assign wb.err = 1'b0;


endmodule

// Local Variables:
// verilog-library-directories:("." "or1200" "jpeg" "pkmc" "dvga" "uart" "monitor" "lab1" "da
// End:
```

C uarttb.sv

Listing 3: Code uarttb

```
`include "include/timescale.v"

module suart_tb();
    logic        clk = 1'b0;
    logic        rst = 1'b1;
    logic        int_o, rx_tx;

    wishbone wb(clk,rst);

    initial begin
       #75 rst = 1'b0;
    end

    always #20 clk = ~clk;

    // Instantiate the DUT
    lab1_uart_top dut(wb, int_o, rx_tx, rx_tx);

    wishbone_tasks wb0(wb);

    // Instantiate the tester
    test_uart test_uart0();
endmodule // jpeg_top_tb

program test_uart();
    int A = 32'h41000000;
    int result = 0;
    int i;

    initial begin
       for (i=0;i<25;i++) begin
          suart_tb.wb0.m_write(32'h90000000, A);
          #400;
          //wait untill full
          while (result != 32'h00010000) begin
             suart_tb.wb0.m_read(32'h90000004, result);
             result = result & 32'h00010000;
             #400;
          end
          suart_tb.wb0.m_read(32'h90000000, result);
          #400;
          result = 0;
          A = A + 32'h01000000;
       end
    end
```

```
endprogram // test_uart
```

D dctsw.c

Listing 4: Code dctsw

```
#include "printf.h"
#include <common.h>


/* Precalculated constants */
#define FIX_0_298631336  ((int)  2446)       /* FIX(0.298631336) */
#define FIX_0_390180644  ((int)  3196)       /* FIX(0.390180644) */
#define FIX_0_541196100  ((int)  4433)       /* FIX(0.541196100) */
#define FIX_0_765366865  ((int)  6270)       /* FIX(0.765366865) */
#define FIX_0_899976223  ((int)  7373)       /* FIX(0.899976223) */
#define FIX_1_175875602  ((int)  9633)       /* FIX(1.175875602) */
#define FIX_1_501321110  ((int)  12299)       /* FIX(1.501321110) */
#define FIX_1_847759065  ((int)  15137)       /* FIX(1.847759065) */
#define FIX_1_961570560  ((int)  16069)       /* FIX(1.961570560) */
#define FIX_2_053119869  ((int)  16819)       /* FIX(2.053119869) */
#define FIX_2_562915447  ((int)  20995)       /* FIX(2.562915447) */
#define FIX_3_072711026  ((int)  25172)       /* FIX(3.072711026) */
#define FIX_C6           ((int)  4433)       /* sqrt(2) * (c6) */
#define FIX_S6           ((int)  10703)       /* sqrt(2) * (s6) */


#define MULTIPLY(var,const)  (((short) (var)) * ((short) (const)))
#define DESCALE(x,n)  ((x) >> (n)) /* no rounding in our HW */
#define CONST_BITS  13

int image[64];
/* Quantization matrix, Matlab notation
Q = [16 11 10 16 24 40 51 61;
     12 12 14 19 26 58 60 55;
     14 13 16 24 40 57 69 56;
     14 17 22 29 51 87 80 62;
     18 22 37 56 68 109 103 77;
     24 35 55 64 81 104 113 92;
     49 64 78 87 103 121 120 101;
     72 92 95 98 112 100 103 99];

reciprocals = round(2^15 ./ Q);
*/

static const int reciprocals[] = {2048, 2979, 3277, 2048, 1365, 819, 643, 537,
                                  2731, 2731, 2341, 1725, 1260, 565, 546, 596,
                                  2341, 2521, 2048, 1365,  819, 575, 475, 585,
                                  2341, 1928, 1489, 1130,  643, 377, 410, 529,
                                  1820, 1489,  886,  585,  482, 301, 318, 426,
                                  1365,  936,  596,  512,  405, 315, 290, 356,
                                  669,   512,  420,  377,  318, 271, 273, 324,
```

```
                                455,   356,  345,  334,  293, 328, 318, 331};


void dct2(int *data);
void dct1(int *a, int *p);

int main()
{
  int i, j, temp, rval, rnd, bits, pos, ctr1, ctr2, ctr3, ctr4;

  printf("\na=\n");
  for (i=0; i<8; i++) {
    for (j=0; j<8; j++)
      printf("%5d ", image[j+8*i]=j+8*i+1);
    printf("\n");
  }
  //clear registers
  REG32(0x99000000) = 0;
  REG32(0x99000004) = 0;
  REG32(0x99000008) = 0;
  REG32(0x9900000C) = 0;
  dct2(image);
  //read data from memory
  ctr1 = REG32(0x99000000);
  ctr2 = REG32(0x99000004);
  ctr3 = REG32(0x99000008);
  ctr4 = REG32(0x9900000C);

  image[0] -= 8192;

  printf("\n8xDCT[a-128]=\n");
  for (i=0; i<8; i++) {
    for (j=0; j<8; j++)
      printf("%5d ", image[j+8*i]);
    printf("\n");
  }


  // Quantization
  printf("\nRND(8xDCT[a-128]/(8xQx1/2))=\n");
  for (i=0; i<8; i++) {
    for (j=0; j<8; j++) {
      temp = image[j+8*i];
      rval = reciprocals[j+8*i];

      temp = temp*rval;

      rnd = (temp & 0x10000) != 0 ;
      bits = (temp & 0xffff) != 0;
      pos = (temp & 0x80000000) == 0;
```

```
      temp = temp >> 17;
      temp += rnd && (pos || bits);


      printf("%5d ", temp);

    }
    printf("\n");
  }

  printf("\Clk cycles for ctr1: %d, ctr2: %d, ctr3: %d, ctr4: %d\n", ctr1, ctr2, ctr3, ctr4);



  return(0);
}

void dct2(int *data)
{
  int tmp[64];

  dct1(data, tmp);
  dct1(tmp, data);
}

/*
 * Perform the forward DCT on one block of samples.
 */

void dct1(int *a, int *b)
{
  int tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
  int tmp10, tmp11, tmp12, tmp13;
  int z1, z2, z3, z4, z5;
  int *pa, *pb;
  int row;

  /* process rows. */
  /* Note results are scaled up by sqrt(8) compared to a true DCT; */

  pa = a;
  pb = b;

  for (row = 7; row >= 0; row--) {
    tmp0 = pa[0] + pa[7];
    tmp7 = pa[0] - pa[7];
    tmp1 = pa[1] + pa[6];
    tmp6 = pa[1] - pa[6];
    tmp2 = pa[2] + pa[5];
    tmp5 = pa[2] - pa[5];
```

```
    tmp3 = pa[3] + pa[4];
    tmp4 = pa[3] - pa[4];

    /* Even part */

    tmp10 = tmp0 + tmp3;
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    pb[0] = tmp10 + tmp11;          /* 0 */
    pb[32] = tmp10 - tmp11;         /* 4 */
    pb[16] = DESCALE(MULTIPLY(tmp12,FIX_C6) + MULTIPLY(tmp13,FIX_S6), CONST_BITS); /* 2 */
    pb[48] = DESCALE(-MULTIPLY(tmp12,FIX_S6) + MULTIPLY(tmp13,FIX_C6), CONST_BITS); /* 6 */

    /* Odd part */

    z1 = tmp4 + tmp7;
    z2 = tmp5 + tmp6;
    z3 = tmp4 + tmp6;
    z4 = tmp5 + tmp7;
    z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3 */

    tmp4 = MULTIPLY(tmp4, FIX_0_298631336); /* sqrt(2) * (-c1+c3+c5-c7) */
    tmp5 = MULTIPLY(tmp5, FIX_2_053119869); /* sqrt(2) * ( c1+c3-c5+c7) */
    tmp6 = MULTIPLY(tmp6, FIX_3_072711026); /* sqrt(2) * ( c1+c3+c5-c7) */
    tmp7 = MULTIPLY(tmp7, FIX_1_501321110); /* sqrt(2) * ( c1+c3-c5-c7) */
    z1 = MULTIPLY(z1, - FIX_0_899976223); /* sqrt(2) * (c7-c3) */
    z2 = MULTIPLY(z2, - FIX_2_562915447); /* sqrt(2) * (-c1-c3) */
    z3 = MULTIPLY(z3, - FIX_1_961570560); /* sqrt(2) * (-c3-c5) */
    z4 = MULTIPLY(z4, - FIX_0_390180644); /* sqrt(2) * (c5-c3) */

    z3 += z5;
    z4 += z5;

    pb[56] = DESCALE(tmp4 + z1 + z3, CONST_BITS); /* 7 */
    pb[40] = DESCALE(tmp5 + z2 + z4, CONST_BITS); /* 5 */
    pb[24] = DESCALE(tmp6 + z2 + z3, CONST_BITS); /* 3 */
    pb[8] = DESCALE(tmp7 + z1 + z4, CONST_BITS);  /* 1 */

    pa += 8;                    /* advance in pointer to next row */
    pb++;                     /* advance out pointer to next column */
  }
}
```