Rapport lab 2-3

TSEA44

Jonathan Karlsson, Niclas Olofsson, Paul Nedstrand
jonka293, nicol, paune
Grupp 2

26 januari 2014

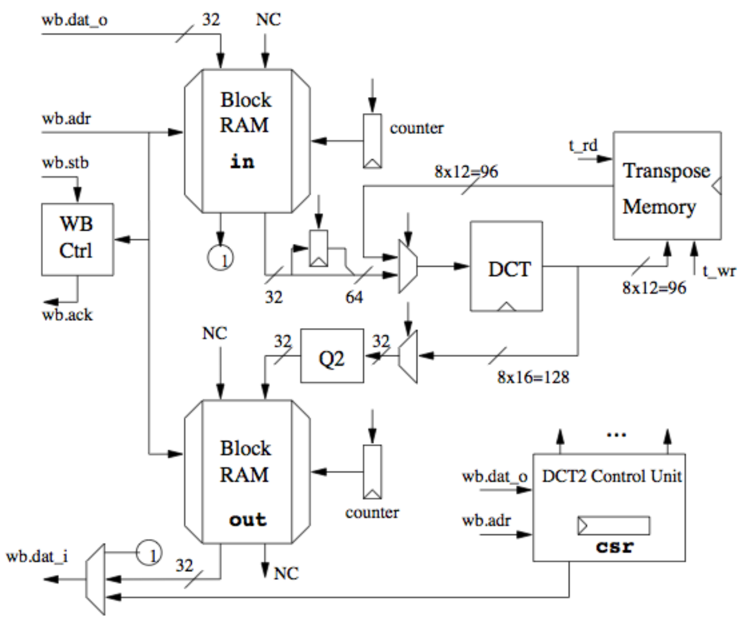
Innehåll

1 Lab 1

1.1 Introduction

Syftet med denna laboration var att konstruera en accelerator för jpeg-komprimering från raw-filer, i hårdvara. Detta löstes med hjälp av programmeringsspråket system verilog och genom att bygga vidare på det existerande skelettet som gavs med uppgiften. I första delen (labb2) så har vi en dator som styr själva acceleratorn och skickar all data via en bus för att sedan behandlas och lagras i ett blockminne där datorn sedan får läsa av resultatet. Ett register används för att starta grunkan och sedan läsa av att resultatet finns att hämta. Det jpeg-acceleratorn gör är att ta ett block med 8x8 pixlar, skicka in det i en givet DCT maskin som gör komprimeringar av pixlarna, sedan transponeras de 8x8 pixlarna i ett minne och skickas tillbaka till DCT'n för att gå igenom en gång till. Alla pixlar multipliceras nu med hårdkodade reciprokaler och läggs i ett utminne. Datorn kan nu läsa av minnet, transponera och spara ner till en jpeg fil som kan öppnas på vanligt sätt.

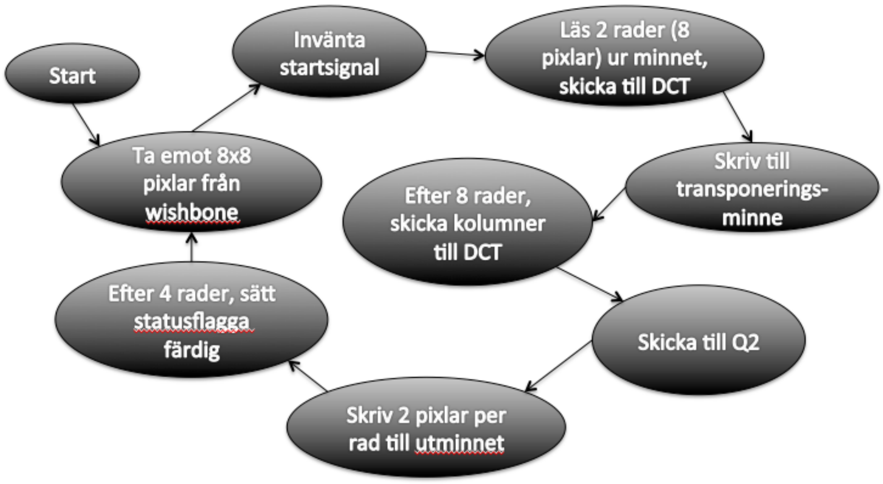
1.2 Implementation



Figur 1: Arkitektur för vår design

Figur 1 visar den arkitektur som konstruerades för att lösa uppgiften. All data som kommer in från bussen läggs in i minnet och vi väntar sedan på en startsignal från datorn. Då används räknaren tillsammans med en vippa för att skicka in data till DCT'n för att sedan skriva in 8x8 pixlar till transponeringsminnet, när vi skrivit klart där så läser vi av kolumnerna (och får därigenom transponeringen), skickar in till DCT'n igen.

Ett problem vi hade så här långt var klockningen, när vi skickar in data från inminnet så måste vi göra det 8 pixlar åt gången, men i minnet finns bara 4 pixlar per rad, så vi måste läsa två rader (därav vippan), vidare måste vi klocka ner DCT'n en klockcykel för att den ska "hinna med". Läsningen från transponeringen var inget problem, däremot när pixlarna kommer ut den andra gången måste klockningen anpassas till Q2-maskinen då den tar två pixlar per klockcykel så DCT'n måste gå ytterligare långsammare för att detta ska fungera.



Figur 2: Tillståndsgraf för vår design

Figur 2 visar en tillståndsgraf över maskinen och dess procedur för att komprimera raw-filer till jpeg. Tillståndsgrafen följer precis det schema som arkitekturen i figur 1 också visar.

1.3 Vidare frågor

2 Labb 3

2.1 Introduktion

Designen från labb 2 visas i figur 1 och innebär att datorn måste engagera sig en hel del och en stor flaskhals för prestandan ligger i bussen som används. Dels måste datorn läsa ur alla pixlar från minnet, via bussen, sedan skicka iväg pixlarna igen via bussen till jpeg-acceleratorn för att sedan skicka en startsignal för varje 8x8 pixlar. När acceleratorn är färdig måste datorn läsa av det i ett register varpå datorn läser av minnet i acceleratorn och skickar ännu mer pixlar genom bussen.

Den flaskhalsen skulle man kunna spara in en del på genom att acceleratorn själv hämtar sin data från minnet istället för att gå genom datorn, där gör den ändå ingen nytta. Så idén är att ha en ytterligare design som hämtar data och skickar in till acceleratorn, sätter igång den självmant. Datorn måste fortfarande läsa av registret för att kontrollera att acceleratorn är färdig samt hämta de färdiga pixlarna, men bussen besparas ändå en hel del data med denna lösning.

2.2 Design

3 Resultat

3.1 Prestanda

Beskrivning	Antal klockcykler
Main program	33 902 171
Init	9 850 924
Encode_image	24 051 247
Forward_DCT	8 317 191
Copy	1 518 825
DCT kernel	0
Quantization	6 798 366
Huffman encoding	15 246 407
Emit_bits	7 904 811

Tabell 1: Prestanda för JPEG-acceleratorn

3.2 FPGA-användning

Flip Flops	7 273 out of 46080	15%
4 input LUTs	11 485 out of 46080	24%
MULT18X18s	19 out of 120	15%
RAMB16s	42 out of 120	35%

Tabell 2: De mest intressanta delarna ur syntes-rapporten för JPEG-acceleratorn med DCT, DMA samt sbit-instruktionen.

4 Sammanfattning

4.1 Filer

jpegtop.sv Mestadels av designen skedde här, våra kontrollsignaler ligger här tillsammans med flera räknare och minnen.

transpose.sv Transponeringsminnet fick en egen fil med en kolumnräknare och en radräknare för att hålla koll på vart grunkan håller på att läsa respektive skriva.

q2.sv Denna sköter multipliceringen med reciprokalerna och skickar sedan resultatet till utminnet.

jpegdma.sv Designen för den andra biten där jpeg-acceleratorn själv läser av informationen i minnet.

5 What to Include in the Lab Report 2

- How did you verify that your 2D DCT hardware works correctly?
- Isthesizeofthe2DDCThardwarejustifiedbytheperformanceimprovements?
- What would be required in order to implement more functionality like zigzag addressing in the 2D DCT hardware module? Would it be difficult to modify jpegfiles to take advantage of such optimizations? And of course, the normal parts of a lab report such as a table of contents, an introduction, a conclusion, etc. The source code that you have written should be included in appendices and referred to from the main document.

6 What to Include in the Lab Report 3

The lab report should contain all source code that you have written. (The source code should of course be commented.) We would also like you to include a block diagram of your hardware. If you have written any FSM you should include a state diagram graph of the FSM. We would also like you to discuss the following questions in detail somewhere in your lab report1:

- How does your hardware work?
- How did you verify that your hardware worked?
- How did you modify the software?
- A timing diagram.
- What is the utilization of your accelerator?
- What is the performance of jpegtest with DMA enabled?
- How long does it take (on average) to read a macroblock into the DCT acceler- ator via DMA?
- How much is the wishbone bus used by the DMA unit and how much is the bus used by the CPU? And of course, the normal parts of a lab report such as a table of contents, an intro- duction, a conclusion, etc. The source code that you have written should be included in appendices and referred to from the main document.

Appendices

A jpegtop.sv

Listing 1: Kod för labb 2

```
//////////////////////////////////////////////////////////////////////
////                                                              ////
////  DAFK JPEG Accelerator top                                   ////
////                                                              ////
////  This file is part of the DAFK Lab Course                    ////
////  http://www.da.isy.liu.se/courses/tsea02                     ////
////                                                              ////
////  Description                                                 ////
////  DAFK JPEG Top Level SystemVerilog Version                   ////
////                                                              ////
////  To Do:                                                      ////
////   - make it smaller and faster                               ////
////                                                              ////
////  Author:                                                     ////
////      - Olle Seger, olles@isy.liu.se                          ////
////      - Andreas Ehliar, ehliar@isy.liu.se                     ////
////                                                              ////
//////////////////////////////////////////////////////////////////////
////                                                              ////
//// Copyright (C) 2005-2007 Authors                              ////
////                                                              ////
//// This source file may be used and distributed without         ////
//// restriction provided that this copyright statement is not    ////
//// removed from the file and that any derivative work contains  ////
//// the original copyright notice and the associated disclaimer. ////
////                                                              ////
//// This source file is free software; you can redistribute it   ////
//// and/or modify it under the terms of the GNU Lesser General   ////
//// Public License as published by the Free Software Foundation; ////
//// either version 2.1 of the License, or (at your option) any   ////
//// later version.                                               ////
////                                                              ////
//// This source is distributed in the hope that it will be       ////
//// useful, but WITHOUT ANY WARRANTY; without even the implied   ////
//// warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR      ////
//// PURPOSE.  See the GNU Lesser General Public License for more ////
//// details.                                                     ////
////                                                              ////
//// You should have received a copy of the GNU Lesser General    ////
//// Public License along with this source; if not, download it   ////
//// from http://www.opencores.org/lgpl.shtml                     ////
////                                                              ////
//////////////////////////////////////////////////////////////////////
```

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

typedef     enum {TST, CNT, RST} op_t;
typedef struct packed
  {op_t op;
   logic rden;
   logic reg1en;
   logic mux1;
   logic dcten;
   logic twr;
   logic trd;
   logic [1:0] mux2;
   logic wren;
   } mmem_t;

module jpeg_top(wishbone.slave wb, wishbone.master wbm);

logic [31:0]          dout_res;
logic                 ce_in, ce_ut;
logic [8:0]                   rdc;
logic [8:0]                   wrc;

logic [31:0]          dob, dob2, ut_doa;

logic [0:7][11:0]     x, ut;

logic [0:7][15:0]     y;

logic [31:0]          reg1;

logic [31:0]          q, dia;
logic [31:0]          doa;
logic                 csren;
logic [31:0]                  csr;
mmem_t        mmem;

logic                 dmaen, ctrl_control;
logic [15:0] rec_o2;
logic [15:0] rec_o1;
reg                   dct_busy;
logic                 dma_start_dct;
logic [7:0]  reciprocal_counter;
logic          dff1;
logic          dff2;
logic          dff1rst;
logic          dff2rst;
reg          ack;
// ****************************************
// *           Wishbone interface          *
```

```
// ******************************************

assign ce_in = wb.stb && (wb.adr[12:11]==2'b00); // Input mem
assign ce_ut = wb.stb && (wb.adr[12:11]==2'b01); // Output mem
assign csren = wb.stb && (wb.adr[12:11]==2'b10); // Control reg
assign dmaen = wb.stb && (wb.adr[12:11]==2'b11); // DMA control

//set ack
always @(posedge wb.clk)
begin
    if (wb.rst)
        ack <= 1'b0;
    else if (~dff1rst && dff2rst)
        ack <= 1'b0;
    else if (wb.stb)
        ack <= 1'b1;
    else
        ack <= 1'b0;
end

always @(posedge wb.clk)
begin
    if (wb.rst)
        dff1 <= 1'b0;
    else
        dff1 <= wb.we;
    dff2 <= dff1;
end

always @(posedge wb.clk)
begin
    dff1rst <= wb.rst;
    dff2rst <= dff1rst;
end

assign wb.ack = ack;

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

// Signals to the blockrams...
logic [31:0] dma_bram_data;
logic [8:0]  dma_bram_addr;
logic        dma_bram_we;

logic [31:0] bram_data;
logic [8:0]  bram_addr;
logic        bram_we;
logic        bram_ce;
```

```
    logic [31:0] wb_dma_dat;
    logic [8:0] clock_counter;

    reg [31:0] dflipflop;
    reg read_enable;
    reg [7:0] DC2_ctrl_counter;
    reg clk_div2;
    reg clk_div4;
    reg [1:0] divcounter;
    reg mux2_enable;
    reg [1:0] mux2_counter;
    reg [31:0] mux2_out;
    reg [31:0] mux2_flipflop;
    reg wren_flipflop;


    const reg [63:0][15:0] reciprocals = '{16'd2048,   16'd2731,   16'd2341,   16'd2341,
16'd1820,   16'd1365,   16'd669,   16'd455,
        16'd2979,   16'd2731,   16'd2521,   16'd1928,   16'd1489,   16'd936,   16'd512,
16'd356,
        16'd3277,   16'd2341,   16'd2048,   16'd1489,   16'd886,   16'd596,   16'd420,
16'd345,
        16'd2048,   16'd1725,   16'd1365,   16'd1130,   16'd585,   16'd512,   16'd377,
16'd334,
        16'd1365,   16'd1260,   16'd819,   16'd643,   16'd482,   16'd405,   16'd318,
16'd293,
        16'd819,   16'd565,   16'd575,   16'd377,   16'd301,   16'd315,   16'd271,
16'd328,
        16'd643,   16'd546,   16'd475,   16'd410,   16'd318,   16'd290,   16'd273,
16'd318,
        16'd537,   16'd596,   16'd585,   16'd529,   16'd426,   16'd356,   16'd324,
16'd331};


    always @(posedge wb.clk) begin
      if (wb.rst) begin
         read_enable <= 1'b0;
         rdc <= 9'b0;
         dflipflop <= 32'b0;
     end else if (csr == 32'h1 || dma_start_dct) begin
         read_enable <= 1'b1;
     // if write is finished and we are reading from the memory
     end else if (read_enable) begin
       // count up address memory on every second clock cycle
       if (clk_div2) begin
         rdc <= rdc + 4;
         dflipflop <= dob;
         if (rdc == 9'h40) begin
           rdc <= 9'd0;
```

```
        read_enable <= 1'b0;
      end
    end
  end
end

//mux till inmem
always_comb begin
   if (dma_bram_we) begin
     bram_we <= dma_bram_we;
     //bram_ce <= 1'b1;
     bram_data <= dma_bram_data;
     bram_addr <= dma_bram_addr;
  end else begin
     bram_we <= wb.we && ce_in;
     //bram_ce <= 1'b1;
     bram_data <= wb.dat_o;
     bram_addr <= wb.adr[8:0];
  end
end

//Mux till dct
always_comb begin
  if (mmem.mux1)
    x = ut;
  else begin
    x = {{4{dflipflop[31]}}, dflipflop[31:24],
         {4{dflipflop[23]}}, dflipflop[23:16],
         {4{dflipflop[15]}}, dflipflop[15:8],
         {4{dflipflop[7]}}, dflipflop[7:0],
         {4{dob[31]}}, dob[31:24],
         {4{dob[23]}}, dob[23:16],
         {4{dob[15]}}, dob[15:8],
         {4{dob[7]}}, dob[7:0]};
  end
end

//div2clk
always @(posedge wb.clk) begin
  if (wb.rst)
    clk_div2 <= 1'b0;
  else
    clk_div2 <= divcounter[0];
end

 //div4clk
always @(posedge wb.clk) begin
  if (wb.rst)
      clk_div4 <= 1'b0;
  else
```

```
    clk_div4 <= ~divcounter[1] && divcounter[0];
end

always @(posedge wb.clk) begin
    if(clock_counter == 2'h3)
        clock_counter <= 2'h0;
    else
        clock_counter <= clock_counter + 1;
end

jpeg_dma dma
 (
  .clk_i(wb.clk), .rst_i(wb.rst),

  .wb_adr_i  (wb.adr),
  .wb_dat_i  (wb.dat_o),
  .wb_we_i   (wb.we),
  .dmaen_i   (dmaen),
  .wb_dat_o  (wb_dma_dat),

  .wbm(wbm),

  .dma_bram_data           (dma_bram_data[31:0]),
  .dma_bram_addr           (dma_bram_addr[8:0]),
  .dma_bram_we             (dma_bram_we),

  .start_dct (dma_start_dct),
  .dct_busy  (dct_busy)
  );

//INMEM!!!!!!!!
RAMB16_S36_S36 #(.SIM_COLLISION_CHECK("NONE")) inmem
 (// WB read & write
  .CLKA(wb.clk), .SSRA(wb.rst),
  .ADDRA(bram_addr),
  .DIA(bram_data), .DIPA(4'h0),
  .ENA(1'b1), .WEA(bram_we),
  .DOA(doa), .DOPA(),
  // DCT read
  .CLKB(wb.clk), .SSRB(wb.rst),
  .ADDRB(rdc),
  .DIB(32'h0), .DIPB(4'h0),
  .ENB(1'b1),.WEB(1'b0),
  .DOB(dob2), .DOPB());

assign dob = dob2;
//UTMEM!!!!!!!
RAMB16_S36_S36 #(.SIM_COLLISION_CHECK("NONE")) utmem
 (// DCT write
  .CLKA(wb.clk), .SSRA(wb.rst),
```

```
  .ADDRA(wrc),
  .DIA(q), .DIPA(4'h0), .ENA(1'b1),
  .WEA(wren_flipflop), .DOA(ut_doa), .DOPA(),
  // WB read & write
  .CLKB(wb.clk), .SSRB(wb.rst),
  .ADDRB(wb.adr[10:2]),
  .DIB(wb.dat_o), .DIPB(4'h0), .ENB(ce_ut),
  .WEB(wb.we), .DOB(dout_res), .DOPB());

reg [31:0] toDatI;
// You must create the wb.dat_i signal somewhere...
assign wb.dat_i = toDatI;

//outmux
always_comb begin
  if(dmaen)
    toDatI = wb_dma_dat;
  else if (csren)
    toDatI = csr;
  else if (ce_in)
    toDatI = doa;
  else if (ce_ut)
    toDatI = dout_res;
end


always @(posedge wb.clk) begin
    if (wb.rst)
        ctrl_control <= 1'b0;
    else if (read_enable)
        ctrl_control <= 1'b1;
    else if (DC2_ctrl_counter == 8'd20)
        ctrl_control <= 1'b0;
end

// the control logic
always @(posedge wb.clk) begin
  if(wb.rst) begin
     csr <= 32'd0;
     mmem.rden <= 1'b0;
     mmem.reg1en <= 1'b0;
     mmem.mux1 <= 1'b0;
     mmem.dcten <= 1'b0;
     mmem.twr <= 1'b0;
     mmem.trd <= 1'b0;
     mmem.wren <= 1'b0;
     mux2_enable <= 1'b0;
     DC2_ctrl_counter <= 8'b0;
     divcounter <= 2'h0;
     dct_busy <= 1'b0;
```

```
end else if (dma_start_dct) begin
    dct_busy <= 1'b1;
end else if (ctrl_control) begin
   divcounter <= divcounter + 1;
    if (clk_div4)
        DC2_ctrl_counter <= DC2_ctrl_counter + 1;
    if(divcounter == 3'h2) begin
        // Enable DCT and get its input from block RAM
        if(DC2_ctrl_counter < 8'd18)
          mmem.dcten <= 1'b1;
        //fulhaxfulhaxheladan!!
        if(mmem.dcten == 1'b1 && DC2_ctrl_counter < 8'd8)
          mmem.twr <= 1'b1;



    // transpose write takes 8 cs
    end else if (DC2_ctrl_counter == 8'd10) begin
        // stop write, begin read
        mmem.twr <= 1'b0;
        mmem.trd <= 1'b1;
        // send transpose memory output to DCT
        mmem.mux1 <= 1'b1;

    // the first row transpose arrives out from DCT
    end else if (DC2_ctrl_counter == 8'd11) begin
        // begin writing result
        mux2_enable <= 1'b1;
        mmem.wren <= 1'b1;

    // 8 cc later, all rows are out of transpose memory
    end else if (DC2_ctrl_counter == 8'd19) begin
        // stop reading from transpose
        mmem.trd <= 1'b0;
        mmem.wren <= 1'b0;
    end else if (DC2_ctrl_counter == 8'd20) begin

        // turn off DCT
        //mmem.dcten <= 1'b0;
        mmem.wren <= 1'b0;
        dct_busy <= 1'b0;

        csr <= 32'd128;
    end
end else if (csren && wb.we) begin
    csr <= wb.dat_o;
end else if (csr == 32'h1) begin
    csr <= 32'h0;
end else begin
    mmem.rden <= 1'b0;
```

```
      mmem.reg1en <= 1'b0;
      mmem.mux1 <= 1'b0;
      mmem.dcten <= 1'b0;
      mmem.twr <= 1'b0;
      mmem.trd <= 1'b0;
      mmem.wren <= 1'b0;
      mux2_enable <= 1'b0;
      DC2_ctrl_counter <= 8'b0;
      divcounter <= 2'h0;
  end
end




//reciprocal_counter!!!
always @(posedge wb.clk) begin
  if (wb.rst)
    reciprocal_counter <= 8'd63;
  else if(mux2_flipflop)
    reciprocal_counter <= reciprocal_counter - 8'd2;
  else
    reciprocal_counter <= 8'd63;
end


always @(posedge wb.clk) begin
  mux2_flipflop <= mux2_enable;
  wren_flipflop <= mmem.wren;
end
//mux2
always_comb begin
  case(mux2_counter) //mmem.mux2
      2'h1:     mux2_out = y[2:3];
      2'h2:     mux2_out = y[4:5];
      2'h3:     mux2_out = y[6:7];
      default:  mux2_out = y[0:1];
  endcase
end


//count mux2 counter and output memory.
always @(posedge wb.clk) begin
  if(wb.rst) begin
      mux2_counter <= 2'd0;
      wrc <= 1'b0;
  end else if(mux2_flipflop) begin
      mux2_counter <= mux2_counter + 1;
      wrc <= wrc + 1;
  end else begin
```

```
        mux2_counter <= 2'd0;
        wrc <= 1'b0;
      end
    end

    //mux2 styrsignal
    /*always_comb begin
        case(mux2_counter)
        2'd1:      mmem.mux2 = 2'd1;
        2'd2:      mmem.mux2 = 2'd2;
        2'd3:      mmem.mux2 = 2'd3;
        default:  mmem.mux2 = 2'd0;
      endcase

    end*/

    //set reciprocals
    always_comb begin
        rec_o1 = reciprocals[reciprocal_counter];
        rec_o2 = reciprocals[reciprocal_counter - 8'd1];
    end

    // 8 point DCT
    // control: dcten
    dct dct0
     (.y(y), .x(x),
      .clk_i(wb.clk), .en(mmem.dcten)
    );

    q2 Q2 (
      .x_i(mux2_out), .x_o(q),
      .rec_i1(rec_o1),
      .rec_i2(rec_o2)
    );

    // transpose memory
    // control: trd, twr

    transpose tmem
     (.clk(wb.clk), .rst(wb.rst),
      .t_wr(mmem.twr)  , .t_rd(mmem.trd),
      .data_in({y[7][11:0],y[6][11:0],y[5][11:0],y[4][11:0],y[3][11:0],y[2][11:0],y[1][11:0],
      .data_out(ut));

endmodule

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

B transpose.sv

Listing 2: Kod för transponeringen

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

module transpose(
    input clk, rst,
    input t_rd,
    input t_wr,
    input [7:0][11:0] data_in,
    output [7:0][11:0] data_out
);
    reg [7:0][7:0][11:0] memory;
    reg [7:0][11:0] out_reg;
    reg [7:0] col_count;
    reg [7:0] row_count;
    reg [3:0] clk_counter;

    always @(posedge clk) begin
        if (rst) begin
            clk_counter <= 0;
        end else if(~t_rd && ~t_wr) begin
            clk_counter <= 0;
        end else begin
            clk_counter <= clk_counter + 1;

            if (clk_counter == 3'h3)
                clk_counter <= 3'h0;
        end
    end

    always @(posedge clk) begin
        if (rst) begin
            row_count[7:0] <= 8'd0;
        end else if(t_wr && clk_counter == 3'h1) begin
            memory[row_count] <= data_in;
            row_count <= row_count + 1;
            end else if (row_count == 8'd8) begin
                row_count <= 8'd0;
        end
    end

    always @(posedge clk) begin
        if (rst) begin
            col_count[7:0] <= 8'd8;
        end else if(t_rd && clk_counter == 3'h1) begin

        //({y[7][11:0],y[6][11:0],y[5][11:0],y[4][11:0],y[3][11:0],y[2][11:0],y[1][11:0],y[0]
          //  out_reg <= memory[7:0][col_count][11:0];
```

```
            col_count <= col_count - 1;
            if (col_count == 8'd0)
                col_count <= 8'd8;
        end
    end

    always_comb begin
        out_reg[7] = memory[0][7-col_count];
        out_reg[6] = memory[1][7-col_count];
        out_reg[5] = memory[2][7-col_count];
        out_reg[4] = memory[3][7-col_count];
        out_reg[3] = memory[4][7-col_count];
        out_reg[2] = memory[5][7-col_count];
        out_reg[1] = memory[6][7-col_count];
        out_reg[0] = memory[7][7-col_count];
    end

    assign data_out = out_reg;

endmodule

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

C Q2.sv

Listing 3: Kod för kvantiseringen

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

module q2(output [31:0] x_o,
          input [31:0] x_i,
          input [15:0] rec_i1,
          input [15:0] rec_i2);

    logic signed [15:0] a_signed;
    logic signed [15:0] b_signed;
    logic signed [15:0] rec_i1_signed;
    logic signed [15:0] rec_i2_signed;
    logic signed [31:0] r1;
    logic signed [31:0] r2;
    logic rnd1, bits1, pos1;
    logic rnd2, bits2, pos2;

    always_comb begin
        a_signed = x_i[31:16];
        rec_i1_signed = rec_i1;
```

```
        r1 = a_signed * rec_i1_signed;

        rnd1 = r1[16]; // (r1 & 0x10000) != 0 ;
        bits1 = r1[15:0] != 16'h0; // (r1 & 0xffff) != 0;
        pos1 = ~r1[31]; //(r1 & 0x80000000) == 0;

        r1[14:0] = r1[31:17];
        r1[31:15] = {17{r1[31]}};

        if (rnd1 && (pos1 || bits1))
            r1 = r1 + 1;
    end

    always_comb begin
        b_signed = x_i[15:0];
        rec_i2_signed = rec_i2;
        r2 = b_signed * rec_i2_signed;

        rnd2 = r2[16]; // (r2 & 0x10000) != 0 ;
        bits2 = r2[15:0] != 16'h0; // (r2 & 0xffff) != 0;
        pos2 = ~r2[31]; //(r2 & 0x80000000) == 0;

        r2[14:0] = r2[31:17];
        r2[31:15] = {17{r2[31]}};

        if (rnd2 && (pos2 || bits2))
            r2 = r2 + 1;
    end

    assign x_o[31:16] = r1[15:0];
    assign x_o[15:0] = r2[15:0];

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

D jpegdma.sv

Listing 4: Kod för labb 3

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

module jpeg_dma(
    input clk_i,
    input rst_i,

    input wire [31:0] wb_adr_i, // Slave interface port, this is not the complete wb bus
```

```
    input wire [31:0] wb_dat_i,
    input wire        wb_we_i,
    output reg [31:0] wb_dat_o,
    input wire        dmaen_i,

    wishbone.master wbm, // Master interface

    output wire [31:0] dma_bram_data,          // To the input block ram
    output reg [8:0]   dma_bram_addr,
    output reg         dma_bram_we,

    output reg         start_dct,              // DCT control signals
    input wire         dct_busy);


   reg [31:0]            dma_srcaddr;        // Start address of image
   reg [11:0]            dma_pitch;          // Width of image in bytes
   reg [7:0]             dma_endblock_x;     // Number of macroblocks in a row - 1
   reg [7:0]             dma_endblock_y;     // Number of macroblocks in a column - 1


   reg          incaddr;
   reg          resetaddr;

   wire                  startfsm;
   wire                  startnextblock;

   reg [3:0]             next_state;
   reg [3:0]             state;
   wire                  dma_is_running;

   reg [8:0]             next_dma_bram_addr;
   wire [3:0]            dma_bram_addr_plus1;

   wire                  endframe;
   wire                  endblock;
   logic                 endblock_reached;
   wire                  endline;
   reg [9:0]             ctr;
   reg nextStbCyc;

   reg [1:0]        goToRelease;

  //  reg [31:0] toDatI;
    // You must create the wb.dat_i signal somewhere...
//     assign wb_dat_i = toDatI;


    reg [31:0] jpeg_data;
```

```
assign   wbm.dat_o = 32'b0; // We never write from this module
assign   dma_bram_data = jpeg_data;

assign   dma_bram_addr_plus1 = dma_bram_addr + 1;

addrgen agen(
             .clk_i                 (clk_i),
             .rst_i                 (rst_i),

             // Control signals
             .resetaddr_i           (resetaddr),
             .incaddr_i             (incaddr),

             .address_o             (wbm.adr),

             .endframe_o            (endframe),
             .endblock_o            (endblock),
             .endline_o             (endline),

             // Parameters
             .dma_srcaddr           (dma_srcaddr),
             .dma_pitch             (dma_pitch),
             .dma_endblock_x        (dma_endblock_x),
             .dma_endblock_y        (dma_endblock_y)
             );

// Memory address registers
always_ff @(posedge clk_i) begin
   if(rst_i) begin
      dma_srcaddr <= 0;
      dma_pitch <= 0;
      dma_endblock_x <= 0;
      dma_endblock_y <= 0;
   end else if(dmaen_i && wb_we_i) begin
      case(wb_adr_i[4:2])
        3'b000: dma_srcaddr    <= wb_dat_i;
        3'b001: dma_pitch      <= wb_dat_i;
        3'b010: dma_endblock_x <= wb_dat_i;
        3'b011: dma_endblock_y <= wb_dat_i;
      endcase // case(wb_adr_i[4:2])
   end
end

always_ff @(posedge clk_i) begin
   if(rst_i)
     endblock_reached <= 1'b0;
   else
     endblock_reached <= endblock;
end
```

```
    // Decode the control bit that starts the DMA engine
    assign startfsm = dmaen_i && wb_we_i && (wb_adr_i[4:2] == 3'b100)
           && (wb_dat_i[0] == 1'b1);
    assign startnextblock = dmaen_i && wb_we_i && (wb_adr_i[4:2] == 3'b100)
           && (wb_dat_i[1] == 1'b1);

    reg fetch_ready;
    reg been_in_wait_ready;
    wire dct_ready;

    assign dct_ready = !dct_busy && fetch_ready;



always @(posedge clk_i) begin
if (incaddr == 1 || been_in_wait_ready)
     been_in_wait_ready = 1'b1;
     else
     been_in_wait_ready = 1'b0;
end



    always_comb begin
       wb_dat_o = 32'h00000000; // Default value
       case(wb_adr_i[4:2])
          3'b000: wb_dat_o = dma_srcaddr;
          3'b001: wb_dat_o = dma_pitch;
          3'b010: wb_dat_o = dma_endblock_x;
          3'b011: wb_dat_o = dma_endblock_y;
          3'b100: wb_dat_o = {ctr, 12'b0, dmaen_i, been_in_wait_ready, state, start_dct, dct_bu
          3'b101: wb_dat_o = next_dma_bram_addr;
          3'b110: wb_dat_o = dma_bram_data;
          3'b111: wb_dat_o = jpeg_data;
       endcase // case(wb_adr_i[4:2])
    end

    always_ff @(posedge clk_i) begin
       if(startfsm | startnextblock | rst_i)
          ctr <= 10'h0;
       else if ( wbm.ack )
          ctr <= ctr + 1;
    end


    localparam DMA_IDLE            = 4'd0;
    localparam DMA_GETBLOCK        = 4'd4;
    localparam DMA_RELEASEBUS      = 4'd5;
```

```
localparam DMA_WAITREADY          = 4'd6;
localparam DMA_WAITREADY_LAST     = 4'd7;

assign dma_is_running = (state != DMA_IDLE);

// Combinatorial part of the FSM
always_comb begin
   next_state = state;
   next_dma_bram_addr = dma_bram_addr;

   resetaddr = 0;
   incaddr = 0;

   // By default we don't try to access the WB bus

   start_dct = 0;
   dma_bram_we = 1;
   goToRelease = 0;

   wbm.sel = 4'b1111; // We always want to read all bytes
   wbm.we = 0; // We never write to the bus

   nextStbCyc = 1'b0;

   jpeg_data = 32'h73;
   fetch_ready = 1'b0;

   if(rst_i) begin
           next_state = DMA_IDLE;
     next_dma_bram_addr = 0;
     jpeg_data = 32'h0;
     start_dct = 0;
     dma_bram_we = 0;
     goToRelease = 0;
   end else begin
      case(state)
        DMA_IDLE: begin
            dma_bram_we = 0;
            if(startfsm) begin
                     next_state = DMA_GETBLOCK;
                     resetaddr = 1; // Start from the beginning of the frame
                     next_dma_bram_addr = 0;
                     jpeg_data = 32'h0;
        start_dct = 0;
     end
        end


        DMA_GETBLOCK: begin
            // Hint: look at endframe, endblock, endline and wbm_ack_i...
```

```
        if (endframe && wbm.ack) begin
    start_dct = 1;
                next_dma_bram_addr = -4;
    next_state = DMA_WAITREADY_LAST;

end else if (endblock && wbm.ack) begin
    start_dct = 1;
                next_dma_bram_addr = -4;
    next_state = DMA_WAITREADY;

end else if(endline && wbm.ack) begin
          next_state = DMA_RELEASEBUS;

        end else begin
          next_state = DMA_GETBLOCK;
        end


        if (wbm.ack) begin
            jpeg_data = wbm.dat_i ^ 32'h80808080;

     nextStbCyc = 1'b0;


            next_dma_bram_addr = next_dma_bram_addr + 4;
     incaddr = 1;

 end else begin
     nextStbCyc = 1'b1;
         end
    end

   DMA_RELEASEBUS: begin
        // Hint: Just wait a clock cycle so that someone else can access the bus if nee
            next_state = DMA_GETBLOCK;
nextStbCyc = 1'b0;
    end

   DMA_WAITREADY: begin
        // Hint: Need to tell the status register that we are waiting here...
        dma_bram_we = 0;
        if (!dct_busy) begin
    fetch_ready = 1;
end

if (startnextblock) begin
    next_state = DMA_GETBLOCK;
end else begin
    next_state = DMA_WAITREADY;
```

```
        end

            end

            DMA_WAITREADY_LAST: begin
                dma_bram_we = 0;
                // Hint: Need to tell the status register that we are waiting here...
                if (!dct_busy) begin
            fetch_ready = 1;
        end

        if (startnextblock) begin
            next_state = DMA_IDLE;
        end else begin
            next_state = DMA_WAITREADY_LAST;
        end
            end

         endcase // case(state)
      end
   end //

   // The flip flops for the FSM
   always_ff @(posedge clk_i) begin
      wbm.stb = nextStbCyc;
      wbm.cyc = nextStbCyc;
      state          <= next_state;
      dma_bram_addr <= next_dma_bram_addr;
   end


endmodule // jpeg_dma

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