Rapport lab 4

TSEA44

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

27 januari 2014

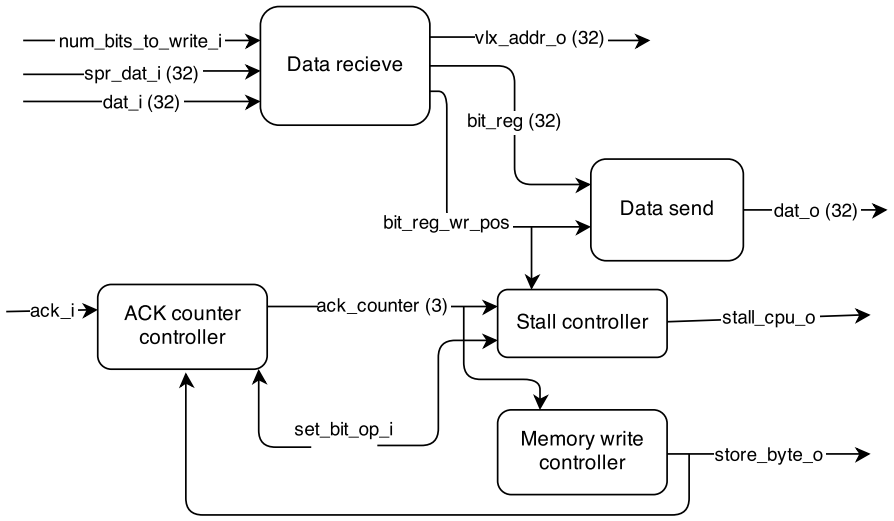
Innehåll

1 Inledning

Det fjärde och sista labben i kursen, lab 4, handlade om att skapa en egen assembler-instruktion (kallad sbit) för skrivning till minnet. Instruktionen tar en längd samt ett data som argument, och sparar all inkommen data i en buffer. När buffern är fylld skrivs denna till minnet. Syftet med denna instruktion är att snabba upp koden för att Huffman-koda den resulterade bilden i vår JPEG-accelerator.

2 Design

Av enkelhetsskäl valde vi att göra några modifikationer av den föreslagna hårdvaruarkitekturen. För det första valde vi att ha all hårdvara i en enda modul, istället för de föreslagna tre modulerna. Även om detta ger sämre modularitet, tyckte vi att det faktum att vi slapp bekymra oss om kommunikation mellan olika moduler gjorde detta till en enklare lösning. Vidare valde vi att bara använda SPR-registret för lagring av aktuell minnesadress, då detta var det enda register vi kunde se någon användning för. Vi upplevde att monitor-programmet gav minst lika bra debug-möjligheter som de övriga två föreslagna SPR-registren skulle ha gjort.



Figur 1: Ett något förenklat blockschema för vår sbit-hårdvara.

2.1 Data recieve

Detta block ansvarar för att lägga in mottagen data i buffern, och uppdatera pekaren för aktuell position. Adress- räknaren är även inkluderad i detta block. Så här i efterhand hade det kanske varit mer logiskt att placera den i ett separat block. Läsningen från data-bussen sker i en enda klockcykel, genom att vår buffer bit_reg skiftas åt vänster för att göra plats för den nya datan, och OR:as

med dat_i. Räknaren bit_reg_wr_pos, som fungerar som en pekare till nästa lediga element i buffern, räknas upp med så många bitar som just togs emot.

Eftersom att buffern skiftas åt vänster när den fylls på, befinner sig data som nyss skickats någonstans i mitten av buffern. Detta göra att denna modul även måste ha kod för att sätta de nyss skickade bitarna till noll.

2.2 Data send

Denna modul lägger ut de bitar som skall skrivas till minnet nästa gång i ett register som är kopplat till den utgående databussen. Detta sker bara om det finns minst en byte i buffern, för att undvika odefinierade signaler. Blocket ansvarar även för att kontrollera om vi är på väg att skriva 0xFF till minnet, och alltså måste skriva 0x00 till minnet härnäst.

2.3 ACK counter

Blocket sköter den ACK-räknare som räknas ner för varje ACK som tas emot, och som alltså håller koll på det kvarvarande antalet ACK-signaler som vi måste få från minnet innan vi är klara. Detta antal är givetvis detsamma som det antal skrivningar vi måste göra. Vi skriver alltid en byte åt gången, så fort vi fått ihop en hel byte, och får som mest två nya bytes varje gång. Med andra ord behöver vi i normalfallet skriva en eller två gånger, beroende på hur mycket data vi fått ihop i vår buffer. För specialfallet med 0xFF krävs dock en extra skrivning av 0x00 direkt efteråt, varför ACK-räknaren inte räknas ner i detta fall.

2.4 Memory write

Detta block hanterar styrsignalen för skrivning till minnet. Denna är hög så länge ACK-räknaren indikerar att vi ännu inte skickat all data till minnet, samt vi inte får någon ACK-signal. Write-signalen kommer annars att bli en klockpuls för lång, vilket leder till att vi i så fall skulle skriva två gånger till minnet istället för en gång.

2.5 Stall controller

Föga förvånande ansvarar detta block för att avgöra när en stall-signal skall skickas. Vi tar det säkra före det osäkra och skickar alltid en stall-signal samma klockpuls som set_bit_op_i-signalen kommer, oavsett om en skrivning till minnet behöver göras eller inte. Vi fortsätter skicka stall-signal tills ACK-räknaren indikerar att vi inte väntar på någon skrivning av data, d.v.s. efter en klockpuls om vi inte behöver skriva något, eller klockpulsen efter att den sista ACK-signalen från minnet har tagits emot, i annat fall.

3 Resultat

3.1 Verifiering av hårdvarans funktion

För att kontrollera att vår hårdvara fungerade, började vi med att anropa vår instruktion från det montor-program som körs när datorn startar. Efter en del felsökning och justering övergick vi till att testa koden på en FPGA med samma monitor-program. Vi använde monitorns inbyggda kommando för att visa minnesadresser för att verifiera att rätt data skrevs till minnet. Ganska snabbt insåg vi behovet av att kunna felsöka även i denna miljö utan att behöva syntetisera om koden varje gång, och skrev därför testprogrammet asm.c som vi kunne ladda in i minnet och köra via monitorn.

Det största problem vi hade under denna lab, och även det svåraste vi fått under labkursen, var att sista biten i varje byte vi skrev till minnet blev fel. Till skillnad från de tidigare fel vi fått under kursen så fick vi varken några varningar av värde vid syntetiseringen, konstiga odefinierade signaler eller märkliga läs/-skrivcykler vid simulering.

Efter noggrannt studerande av syntesrapporten upptäckte vi att en felaktig ihopslagning av två bitar i ett register orsakade problemet. Vi gjorde om koden för skrivning till det aktuella registret. Simuleringen blev fortfarande likadan som tidigare, men ingen konstig ihopslagning gjordes vid syntetiseringen, vilket löste vårt problem.

Till sist testades även hårdvaran genom att instruktionen användes av JPEG-acceleratorn. jchuff.c modifierades, för att använda set bit- instruktionen för skrivning till minnet. Efter en rejäl stunds felsökande genererades till sist en korrekt bild.

3.2 Prestanda

Vi använde vårt testprogram och den prestandaräknare som inkluderades i jpegtest-programmet för att mäta prestandan på några olika slags anrop till vår set bit-instruktion (Tabell 1). Dels varierade vi storleken, dels testade vi även att enbart skriva 0xFF, vilket i vår implementation gör två minnesskrivningar.

Storlek	Data	Antal klockcykler
2	0x02	10
8	0x33	13
8	0xFF	17
16	0x33	17

Tabell 1: Antal klockcykler per anrop av vår sbit-instruktion. Tabellen visar medelvärdet av 100 försök.

Vid användning i JPEG-acceleratorn jämförde vi utskriften av prestandamätningen för en version av programmet som kompilerades utan set bit-instruktionen

(Tabell 2), mot resultatet med denna instruktion påslagen (Tabell 3). Båda dessa versioner innehåller alla de tidigare förbättringar som gjorts; hårdvaru-DCT samt DMA.

Beskrivning	Antal klockcykler
Main program	25 847 242
Init	6 600 044
Encode_image	19 247 198
Forward_DCT	6 489 189
Copy	0
DCT kernel	0
Quantization	6 489 189
Huffman encoding	12 226 950
Emit_bits	4 983 905

Tabell 2: Prestanda för JPEG-acceleratorn utan sbit-instruktionen

Beskrivning	Antal klockcykler
Main program	21 754 287
Init	6 653 818
Encode_image	15 100 469
Forward_DCT	6 396 157
Copy	0
DCT kernel	0
Quantization	6 396 157
Huffman encoding	8 126 873
Emit_bits	1 306 129

Tabell 3: Prestanda för JPEG-acceleratorn med sbit-instruktionen

3.3 FPGA-användning

Flip Flops	7499 out of 46080	16%
4 input LUTs	12519 out of 46080	28%
MULT18X18s	19 out of 120	15%
RAMB16s	42 out of 120	35%

Tabell 4: De mest intressanta delarna ur syntes-rapporten för JPEG-acceleratorn med DCT, DMA samt sbit-instruktionen.

4 Slutsats

4.1 Analys av prestanda

Prestandan för set bit-instruktionen beror uteslutande på hur många minnesaccesser som behöver göras vid anropet. Vid upprepade anrop med storleken

2 görs en skrivning till minnet var fjärde anrop, vilket gör att snitt-tiden blir lägre än i övriga fall. För storleken 8 görs en skrivning vid varje anrop, vilket leder till värsta fallet på 13 klockcykler i medel. Vid skrivning av 0xFF görs internt två separata skrivningar, vilket gör att detta tar exakt lika lång tid som skrivningar av storleken 16, nämligen 17 cykler.

En intressant slutsats vi kan dra av detta är att det tar ganska mycket overhead bara att utföra vår instruktion. För storlek 2 görs fyra gånger färre minnesskrivningar jämfört med storlek 8, men tiden för dessa skiljer sig bara med en tredjedel. På samma sätt görs dubbelt så många skrivningar för storlek 16 som storlek 8 med samma data, men skillnaden i exekveringstid dem emellan är mindre än en fjärdedel.

Instruktionen gjorde en hel del skillnad när vi använde den i JPEG- acceleratorn. Som synes i Tabell 3 minskade tiden för att skriva data till minnet vid Huffman-kodningen (emit_bits) från 5,0 miljoner cykler till 1,3 miljoner cykler. Den totala tiden minskade därigenom med nästan lika mycket, vilket gav en total prestandaökning med 16%. Gruppen är dock enig om att prestandaförbättring-per-timme-spenderad-i-Muxen-indexet troligen är väldigt lågt.

4.2 Möjliga förbättringar

Ett problem med vår nuvarande hårdvara är att vi alltid skickar en stall-signal till CPU:n så fort vi får in en set bit-instruktion. Detta är i väldigt många fall inte alls nödvändigt (särskilt som vi bara skriver till minnet om vi har en hel byte att skicka). Detta är särskilt problematiskt i ett operativsystem med multitasking - det blir omöjligt att göra saker parallelt om vi säger åt resten av datorn att sluta arbeta så fort någon använder vår instruktion.

En möjlig förbättring skulle kunna vara att hårdvaran detekterar i vilka fall som vi behöver skicka en stall-signal och inte. Detta behöver i så fall ske med kombinatorik för att hinna skicka signalen tillräckligt fort, i de fall där detta behövs. På så sätt skulle vi kunna lösa problemen med dålig paralellism i operativsystem med multitaskning, samt få något bättre prestanda generellt, på bekostnad av ganska lite hårdvara.

Om målet istället skulle vara att hitta en billigare lösning på bekostnad av prestanda, hade vi kunnat använda ett skiftregister för att stegvis skifta in varje bit data i vårt register, istället för vår nuvarande lösning som troligen realiseras med en stor samling multiplexrar och OR-grindar för att lyckas göra detta på en klockcykel.

Appendix

A asm.c

Listing 1: Testprogram för mätning av prestanda för sbit-instruktionen

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

#include "perfctr.h"

int main() {
  unsigned int code, i, size, startcycle, perf;

  startcycle = gettimer();
  for (i = 0; i < 100; ++i) {
    asm volatile("l.sd 0x0(%0),%1" : : "r"(0x2), "r"(2));
  }

  perf = gettimer() - startcycle;
  printf("100x size 2 = %d\n", perf);
  startcycle = gettimer();

  for (i = 0; i < 100; ++i) {
    asm volatile("l.sd 0x0(%0),%1" : : "r"(0x33), "r"(8));
  }

  perf = gettimer() - startcycle;
  printf("100x size 8 = %d\n", perf);
  startcycle = gettimer();

  for (i = 0; i < 100; ++i) {
    asm volatile("l.sd 0x0(%0),%1" : : "r"(0x33), "r"(16));
  }

  perf = gettimer() - startcycle;
  printf("100x size 16 = %d\n", perf);
  startcycle = gettimer();

  for (i = 0; i < 100; ++i) {
    asm volatile("l.sd 0x0(%0),%1" : : "r"(0xFF), "r"(8));
  }

  perf = gettimer() - startcycle;
  printf("100x FF size 8 = %d\n", perf);

  return 0;
}
```

B or1200_vlx_top.sv

Listing 2: Hårdvaruimplementationen för sbit-instruktionen

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

module or1200_vlx_top(/*AUTOARG*/
    // Outputs
    spr_dat_o, stall_cpu_o, vlx_addr_o, dat_o, store_byte_o,
    // Inputs
    clk_i, rst_i, ack_i, dat_i, set_bit_op_i, num_bits_to_write_i,
    spr_cs, spr_write, spr_addr, spr_dat_i
    );
    input clk_i;
    input rst_i;

    input  ack_i; //ack
    input [31:0] dat_i; //data to be written

    input       set_bit_op_i; //high if a set bit operation is in progress
    input [4:0] num_bits_to_write_i; //.size
    input  spr_cs; //sprs chip select
    input  spr_write; //sprs write
    input [1:0] spr_addr; //sprs address
    input [31:0] spr_dat_i; //sprs data in

    output [31:0] spr_dat_o; //sprs data out
    output      stall_cpu_o; //if set high the cpu will be stalled
    output [31:0] vlx_addr_o; //the address to store vlx data
    output [31:0] dat_o; //data vlx data to be stored
    output       store_byte_o; //high when storing a byte

    wire         set_init_addr;
    wire         store_reg;
    wire [31:0]          su_data_in;
    wire [31:0]          spr_dp_dat_o;
    wire         write_dp_spr;

    reg      is_sending;
    reg [31:0]  bit_reg;
    reg [5:0]   bit_reg_wr_pos;
    reg [7:0] data_to_be_sent;
    reg [31:0] address_counter;

    reg [2:0] ack_counter;
    reg stall, next_stall;
    reg send_00;
    reg recieved_set_bit;
    reg ack_ff;
```

```
assign store_byte_o = is_sending;

// Signals for reading and writing SPR address register.
assign      set_init_addr = spr_cs & spr_addr[1] & spr_write;
assign      write_dp_spr = spr_cs & spr_write & ~spr_addr[1];
assign      su_data_in = set_init_addr ? spr_dat_i : bit_reg;
assign  spr_dat_o = spr_addr[1] ? vlx_addr_o : spr_dp_dat_o;
assign  vlx_addr_o = address_counter;

assign dat_o = {24'b0, data_to_be_sent};
assign stall_cpu_o = stall;

// Delayed ACK.
always @(posedge clk_i) begin
    ack_ff <= ack_i;
end

// Delayed start signal.
always @(posedge clk_i) begin
    recieved_set_bit <= set_bit_op_i;
end

// Data recieve controller
always_ff @(posedge clk_i or posedge rst_i) begin
if(rst_i) begin
    bit_reg         <= 0;
    bit_reg_wr_pos  <= 0;
    address_counter <= 0;
end else begin
    // If someone just wrote to address register, set the address counter to this value.
    if (set_init_addr) begin
        address_counter <= spr_dat_i;

    // If we just recieved the start signal.
    end else if(set_bit_op_i) begin
        if (bit_reg_wr_pos <= 7) begin
            // Insert num_bits_to_write bits from dat_i into bit_reg by ORing.
            bit_reg <= (bit_reg << num_bits_to_write_i) | dat_i;
            // Update write pointer.
            bit_reg_wr_pos <= bit_reg_wr_pos + num_bits_to_write_i;
        end

    // If we just recieved an ACK.
    end else if (ack_ff && (send_00 || bit_reg_wr_pos > 7)) begin
        // Reset the written bits in bit_reg.
        bit_reg[bit_reg_wr_pos-1] <= 0;
        bit_reg[bit_reg_wr_pos-2] <= 0;
        bit_reg[bit_reg_wr_pos-3] <= 0;
        bit_reg[bit_reg_wr_pos-4] <= 0;
        bit_reg[bit_reg_wr_pos-5] <= 0;
```

```
            bit_reg[bit_reg_wr_pos-6] <= 0;
            bit_reg[bit_reg_wr_pos-7] <= 0;
            bit_reg[bit_reg_wr_pos-8] <= 0;

            // Unless we just send the zero-padding for 0xFF, update bit_reg pointer.
            if (~send_00) begin
                bit_reg_wr_pos <= bit_reg_wr_pos - 8;
            end
            // Count up address.
            address_counter <= address_counter + 1;
        end
    end
end

// Data send controller
always @(posedge clk_i) begin
    if (rst_i) begin
        data_to_be_sent <= 0;
        send_00 <= 0;
    // Handling of zero-padding for 0xFF.
    end else if (ack_ff && send_00) begin
        data_to_be_sent <= 0;
        send_00 <= 0;
    // Handling of special cases which could give undefined signals.
    end else if (bit_reg_wr_pos < 8) begin
        data_to_be_sent <= 0;
    end else if (data_to_be_sent == 8'hff) begin
        send_00 <= 1;
    end else begin
        // Put data from the place pointed out by bit_reg pointer to send register.
        data_to_be_sent[7] <= bit_reg[bit_reg_wr_pos-1];
        data_to_be_sent[6] <= bit_reg[bit_reg_wr_pos-2];
        data_to_be_sent[5] <= bit_reg[bit_reg_wr_pos-3];
        data_to_be_sent[4] <= bit_reg[bit_reg_wr_pos-4];
        data_to_be_sent[3] <= bit_reg[bit_reg_wr_pos-5];
        data_to_be_sent[2] <= bit_reg[bit_reg_wr_pos-6];
        data_to_be_sent[1] <= bit_reg[bit_reg_wr_pos-7];
        data_to_be_sent[0] <= bit_reg[bit_reg_wr_pos-8];
    end
end

// ACK counter controller
always @(posedge clk_i) begin
    if (rst_i)
        ack_counter <= 0;

    // If we are currently sending something, and recieves an ACK.
    else if (ack_i == 1 && is_sending) begin
        // Unless we just sent the padding for 0xFF.
        if (data_to_be_sent != 8'hff) begin
```

```
            // Decrease the number of ACKs left to recieve.
            ack_counter <= ack_counter - 1;
        end
    // If we just recieved the start command.
    end else if (recieved_set_bit) begin
        // Determine number of ACKs to recieve depending on data length.
        if (bit_reg_wr_pos > 15)
            ack_counter <= 2;
        else if (bit_reg_wr_pos > 7)
            ack_counter <= 1;
    end
end

// Memory write signal
always @(posedge clk_i) begin
    if(rst_i) begin
        is_sending <= 0;
    end else if (~ack_i && ack_counter > 0) begin
        is_sending <= 1;
    end else begin
        is_sending <= 0;
    end
end

// Stall controller
always @(posedge clk_i) begin
    if (rst_i) begin
        next_stall <= 0;
    // If we just have recieved the start signal, begin to stall.
    end else if (set_bit_op_i == 1 || recieved_set_bit == 1) begin
        next_stall <= 1;
    // If we have recieved the required number of ACK-signals, end stall.
    end else if (ack_counter == 0) begin
        next_stall <= 0;
    end
end

always_comb begin
    stall = next_stall | set_bit_op_i;
end

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