Labrapport 4

TSEA44

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

25 januari 2014

Innehåll

1 Inledning **2**

2 Design **2**

3 Resultat **2**
3.1 Verifiering av hårdvarans funktion 2
3.2 Prestanda 3
3.3 FPGA-användning 4

4 Slutsats **4**
4.1 Analys av prestanda 4
4.2 Möjliga förbättringar 4

5 Appendix: Källkod **5**

1 Inledning

Det fjärde och sista miniprojektet i kursen, lab 4, handlar om att skapa en egen assembler-instruktion för skrivning till minnet. Syftet med denna instruktion är att snabba upp koden för att Huffman-koda den resulterade bilden från vår jpeg-accelerator.

Instruktionen tar en längd samt ett data som argument, och sparar all inkommen data i en buffer tills minst 8 bitar erhållits. I detta fall skrivs den buffrade datan till minnet; en byte åt gången och så många bytes som möjligt (maximalt två).

2 Design

- How does your hardware work?

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	0x00	10
8	0x00	13
8	0xFF	17
16	0x00	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 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 Huffmankodningen (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.

5 Appendix: Källkod