LABORATION

TSEA28

Analys av en ARM-processors cache genom mätning på AXI-bussen

Version: 1.22
2013 (OS,AE) 2014 (AE) 2024 (KP)

Namn och personnummer	Godkänd

blank sida

Innehåll

blank sida

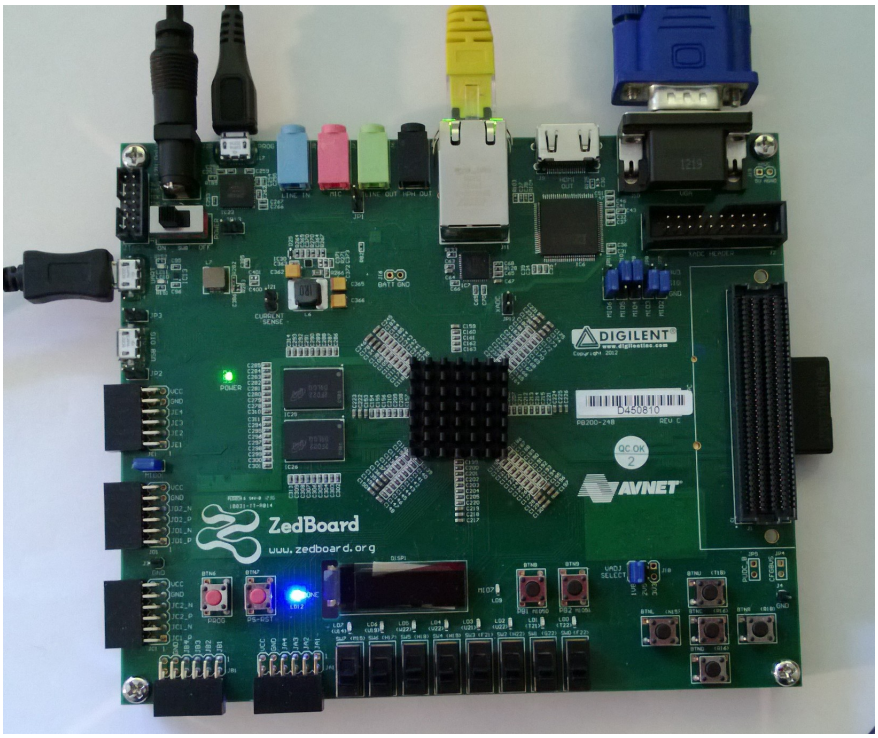
1 Inledning

1.1 Syfte

Det finns två syften med denna laboration. Dels ska du lära dig hur en cache fungerar och dels ska du lära dig hur en modern systembuss fungerar. När du är klar med laborationen kommer du att kunna förklara hur en cache beter sig i olika situationer och hur olika designparametrar påverkar cachens prestanda. Du kommer också att förstå hur en cache interagerar med systembussen och hur man ska tänka för att optimera ett program med avseende på cache och busstrafik.

1.2 Förberedelser

Innan du kommer till laborationen måste du vara väl förberedd. Alla uppgifter som är utmärkta med ett pekfinger ska redovisas på laborationstillfället. En del av dem kan med fördel utföras i förväg innan labbtillfället som förberedelseuppgifter.



Figur 1: Utvecklinsgsystemet Zedboard

2 Bruksanvisning

I denna laboration ska du använda ett utvecklingssystem som heter Zedboard. Zedboard innehåller en krets som heter Zynq-7000. Denna innehåller i sin tur två ARM-processorer av modell Cortex-A9 samt ett antal kringkretsar. För den som är intresserad finns mer information på `http://www.zedboard.org/`. Den kompletta manualen för den krets vi använder (Zynq-7000) finns på `https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM` för den som är nyfiken...

Viktig: placera filer endast i `X:` på windowsmaskinen. Filer placerade på andra ställen kommer försvinna när ni loggar ut.

2.1 Att göra första gången

Första gången måste lab-skelettet kopieras till ert hemkonto. Börja med att öppna file explorer, och kopiera mappen `K:\TSEA28\lab5` till någon lämplig plats i ert hemkonto under `X:`.

2.2 Kompilering och nedladdning av program

Alla program som används skrivs i C, och kompileras med hjälp av make. Detta kommando körs i ett speciellt terminalfönster.

- Öppna terminalfönstret genom att välja **Start** → **ISE Design Suite 14.7** → **ISE Design Suite 64-bit Command Prompt**. Förflytta dig till lab5-katalogen med hjälp av kommmandona **X:**, **cd**, och se att du är på rätt plats med **dir**.
- I katalogen `lab5` kan du nu ge kommandot `make hello` för att kompilera ett enkelt testprogram.

- Starta Zedboard genom att slå på spänningen. (Strömbrytaren i övre vänstra hörnet på bilden.) Efter cirka fem sekunder bör den blå lysdioden tändas och systemet är då klart att användas.
- Starta sedan programmet `TeraTerm` med från start-menyn. Där behöver några inställningar göras först: Ett nytt fönster öppnas först. Välj där **Serial** och **Port** sätts till USB Serial Port. Därefter väljs meny **Setup** → **Serial port...** Välj **Speed:** 115200 och sätt **transmit delay** till 1 msec/line.
- Nu ska du ha kontakt med monitorprogrammet som kör på Zedboard. Ge kommandot `h` och följande utskrift visas:

```
Memory manipulation:
  d <addr> [display mem]
  d [display mem (continued)]
  m <addr> <dat> [modify mem (32 bit at a time)]
  l [load hexfile]->
  c <src> <dst> <len> [copy]
  f [flush caches]
I/O manipulation:
  p <val> [Set parport]
  b [Create reference bitmap]
Execution:
  g <addr> [go to specified address]
  g [go to start address specified in hex-file]
```

- I monitorprogrammet ska du nu skriva kommandot `l`. Monitorprogrammet svarar med `Please send an intel hex file... (ctrl c to abort).`
- Välj meny **File** → **Send file....** och väljer sedan filen `hello.hex` som ligger i katalogen `lab5` (du kan ändra **File name** till ***.hex** för att bara se hex-filer). Tryck sedan **Open**. Filen laddas ner och placeras i minnet. Monitorprogrammet utskrift avslutas med (exempelvis) `Start address in hexfile: 0x00400a3c`.
- Programmet `hello` kan nu startas med kommandot `g` i `TeraTerm`.
- Nu ska programmet skriva ut texten `Hello world` och sedan ska du få en ny prompt i TeraTerm.

2.3 Nollställning av Zedboard

Om programmet som kör på ditt Zedboard kraschar kan du behöva nollställa kortet genom att trycka på resetknappen som är märkt PS-RST, till vänster om den blå lysdioden. Notera att knappen bredvid (märkt PROG) ej ska använda då den enbart nollställer delar av systemet. Har du tryckt på denna knapp behöver du nollställa systemet med hjälp av PS-RST.

2.4 ChipScope

Chipscope är en logikanalysator som ska användas i denna laboration. För att starta den väljer välja **Start** → **ISE Design Suite 14.7** → **Analyzer**. Om en varning dyker upp om windows brandvägg kan du välja **Cancel**.

När Chipscope har startat väljer du menyalternativet **JTAG Chain** → **Digilent USB JTAG Cable**. Du ska då få upp en ruta som (bland annat) innehåller serienumret (SN) för det Zedboard som du ska använda[1].

[1] Ibland fungerar detta tyvärr ej på första försöket av oklara anledningar. Försök igen om det inte fungerar på första försöket. Fungerar det inte efter tre försök får man ta bort USB-kabeln som sitter i PROG-porten på Zedboard och koppla i denna igen.

Tryck OK. Ytterligare en ruta kommer då upp som innehåller vilken ordning de olika JTAG-enheterna finns i. Här ska du också trycka OK.

Nu ska du ladda in rätt konfigurationsfil i ChipScope. Detta gör du genom att välja menyn **File→Open Project...** och öppna filen `chipscope_config_full.cpj` som finns i lab5-katalogen.

Nu kan du i ChipScope trycka på "play"-knappen för att starta en mätning. ChipScope väntar nu på att du ska göra en skrivning till adress 0x9fff0000. För att se att det fungerar kan du nu skriva exempelvis följande monitor-kommando i `TeraTerm`:

```
m 9fff0000 1234abcd
```

Efter ett litet tag ska du nu i ChipScope-fönstret få upp den busstrafik som inträffade på `AXI WRITE CHANNEL` i samband med detta. Du behöver antagligen zooma in runt 0 klockcykler för att du ska kunna se detta ordentligt. För att zooma in kan du antingen klicka på zoomknapparna eller rita upp en ruta med vänsterknappen i signalfönstret.

2.4.1 Att mäta tiden mellan två händelser i ChipScope

Varje gång som C-funktionen `trigger_logic_analyzer()` körs dyker det upp en skrivning på AXI-bussens skrivkanal. Som labskelettet ser ut när ni får det är det också enbart i samband med att denna funktion anropas som det ska dyka upp skrivningar på skrivbussen. För att mäta tiden mellan två händelser ska du alltså se till så att denna funktion körs så att det är lätt för dig att identifiera de tidpunkter du är intresserad av.

För att lätt räkna ut tiden mellan två händelser i ChipScope kan du högerklicka på den ena tidpunkten och välja place X cursor. Sedan kan du högerklicka på den andra tidpunkten och välja place O cursor. Tidsdifferensen (mätt i klockcykler) visas längst ner till höger. Notera att ChipScope kan spara data ifrån maximalt 8192 klockcykler i den konfiguration vi använder just nu. Längre tidsperioder än så går alltså inte att mäta i ChipScope.

För att tydligare se vilka signaler som är vilka kan ni dra i kolumnerna längst upp vid kolumnnamnen. För att läsa av värden vid en viss tidpukt är det enklast att placera en cursor (X eller O) vid denna tidpunkt och läsa av i motsvarande kolumn (X eller O).

3 Teori

I denna laboration kommer en del nya koncept att introduceras. Viktigast är cacheminnet och AXI-bussen, men det finns även några andra koncept som är viktiga att förstå för att kunna förstå programmet i uppgift 3 fullt ut. Dessa koncept introduceras nedan (avsnittet om cacheminne i läroboken rekommenderas givetvis också).

3.1 Cacheminne

När du är klar med den här laborationen förväntas du ha koll på både allmän cacheteori samt den specifika teori som du behöver för att klara av programmeringsupgifterna:

- Gruppassociativa cache:ar
- Förhämtning
- Ersättningsalgoritmer
- Tillbakaskrivningsalgoritmer
- Skrivbuffer
- Cachehierarki

Notera att det är mycket möjligt att du i samband med den muntliga examinationen kan få diskutera hur exempelvis prestandan för det system du använder i laborationen kommer att ändras om en annan typ av cache skulle användas. En exempel på en sådan fråga skulle exempelvis kunna vara hur systemets prestanda skulle förändras om en annan tillbakaskrivningsalgoritm användes.

3.1.1 Gruppassociativa cache:ar

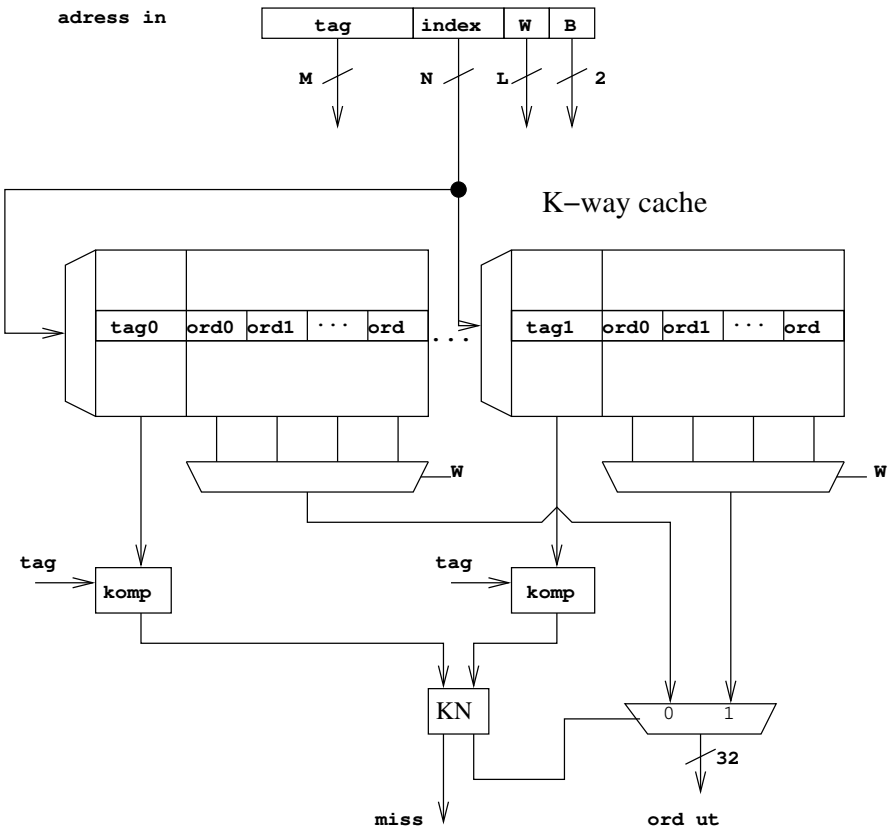
Figur 2 visar ett gruppassociativt (*set-associative*) cacheminne (CM) med K vägar. Detta CM består alltså av K st identiska del-CM. Lägg märke till att detaljer för skrivning i CM inte är utritade.

En läsning i CM går till på följande sätt:

1. Adressen, som kommer från CPU:n, delas in i fyra fält. I vårt fall gäller att $M + N + L + 2 = 32$.
   - *index*. Används för att välja en rad i CM. I varje del-CM består raden av ev en tag och en cacheline (CL). CL består av 2^L ord.
   - *W,B*. Fältet W väljer ett ord i CL och fältet B väljer en byte (B) i ordet. Här gäller alltid att ett ord är 4 bytes.
   - *tag*. Återstående mest signifikanta bitar kallas tag.
2. Indexfältet väljer samma rad i K del-CM. K jämförelser görs mha komparatorer mellan lagrad tag och tag-fältet i adressen.
3. Låt oss anta att $tag = tag_0$. Detta kallas för en cache-träff och medför att sökt ord muxas ut på ledningarna `ord ut` ifrån rätt del-CM.

Viktiga parametrar för vårt CM är

- *associativitet*. $A = K$.
- *cachelinens storlek*. $CL = 4 \cdot 2^L$ B.
- *cachens storlek*. $CM = K \cdot 2^N \cdot 4 \cdot 2^L$ B.



Figur 2: Ett K-vägs cacheminne. Detaljer för skrivning är ej utritade. KN är ett kombinatoriskt nät.

Följande parametrar håller vi fixa: ordlängd = 4 bytes (32 bitar) och adresslängd = 32 bitar.

3.1.2 Exempel

Ett exempel på hur man kan dela in en adress i en cache som kan lagra 256 kB kan ses nedan:

- *associativitet*. $A = 4$.
- *cacheline-storlek*. $CL = 16$ B.
- *cachens storlek*. $CM = 256$ kB.

Detta ger parametrarna $M = 16$, $N = 12$ och $L = 2$. En 32-bitars adress bestående av 8 hex-siffror kan i detta fall delas upp på följande sätt:

$$adress = XXXXYYYZ,$$

där $XXXX$ är tag och YYY är index.

Hur denna cache reagerar på en viss följd av läsningar synliggörs i tabell 1 under antagandet att cachen startar i tömt (flushed) tillstånd

I detta fall kommer cachen efter dessa transaktioner ha det innehåll som ses i tabell 2. Om valid-biten inte är satt i en viss cacheline indikeras detta genom “Invalid” i tabellen. Gå gärna igenom tabell 1 och 2 tillsammans för att övertyga dig själv om att du förstår detta till fullo.

OBS: Notera att om $A = 2$, $CL = 32$ B och $CM = 256$ kB blir parametrarna nu M = 15, N = 12 och L = 3. En läsning på adress 0x89abcdec = $1000\ 1001\ 1010\ 1011\ 1100\ 1101\ 1110\ 1100_2$ blir då uppdelad i tag index och W enligt $1000\ 1001\ 1010\ 101 \parallel 1\ 1100\ 1101\ 111 \parallel 0\ 11 \parallel 00$. Alltså, tag =

Operation	tag	index	W	Kommentar
Läsning på adress 0x82000000	0x8200	0x000	0x0	Cachemiss, läs in adress 0x82000000-0x8200000f till väg 0, index 0
Läsning på adress 0x82000004	0x8200	0x000	0x1	Cacheträff i väg 0, index 0
Läsning på adress 0x82000008	0x8200	0x000	0x2	Cacheträff i väg 0, index 0
Läsning på adress 0x8200000c	0x8200	0x000	0x3	Cacheträff i väg 0, index 0
Läsning på adress 0x82000010	0x8200	0x001	0x0	Cachemiss, läs in adress 0x82000010-0x8200001f till väg 0, index 1
Läsning på adress 0x83000000	0x8300	0x000	0x0	Cachemiss, läs in adress 0x83000000-0x8300000f till väg 1, index 0
Läsning på adress 0x83480008	0x8348	0x000	0x2	Cachemiss, läs in adress 0x83480000-0x8348000f till väg 2, index 0
Läsning på adress 0x91fc0000	0x91fc	0x000	0x0	Cachemiss, läs in adress 0x91fc0000-0x91fc000f till väg 3, index 0
Läsning på adress 0x89abcdec	0x89ab	0xcde	0x3	Cachemiss, läs in adress 0x89abcde0-0x89abcdef till väg 0, index 0xcde

Tabell 1: Exempel på hur en cache med parametrarna A=4, CL=16 B och CM=256 kB hanterar en viss minnesåtkomstsekvens (alla läsningar antas vara 32 bitar breda)

	Väg 0		Väg 1		Väg 2		Väg 3	
Index	Tag	Innehåll	Tag	Innehåll	Tag	Innehåll	Tag	Innehåll
0	0x8200	Från 0x82000000	0x8300	Från 0x83000000	0x8348	Från 0x83480000	0x91fc	Från 0x91fc0000
1	0x8200	Från 0x82000010		Invalid		Invalid		Invalid
2		Invalid		Invalid		Invalid		Invalid
...		...		...		...		...
0xcde	0x89ab	Från 0x89abcde0		Invalid		Invalid		Invalid
0xcdf		Invalid		Invalid		Invalid		Invalid
...		...		...		...		...
0xfff		Invalid		Invalid		Invalid		Invalid

Tabell 2: Innehållet i cachen efter att läsningarna i tabell 1 utförts

$100\ 0100\ 1101\ 0101_2$ = 0x44d5, index = $1110\ 0110\ 1111_2$ = 0xe6f och W = 011_2 = 0x3. Som synes kan alltså en hexadecimal siffra i adressen bli uppdelad mellan två olika fält i cachen (t ex mellan tag och index).

3.1.3 Ersättningsalgoritmer

I exemplet i avsnitt 3.1.2 fanns det alltid en tom cacheline ledig vid alla cachemissar. Om processorn försöker läsa i exempelvis adress 0x8acb000c gäller inte detta längre eftersom alla vägar vid index 0 redan är fulla. I detta läge behöver cachen välja ut en lämplig cacheline i index 0 att kasta för att ha möjlighet att ladda in datat som finns på adress 0x8acb0000-0x8acb000f. Några vanliga ersättningsalgoritmer som brukar användas kan ses nedan:

- Slumpmässig: En slumpmässig väg i valt index väljs ut
- LRU (Least Recently Used): Den väg i valt index som användes för längst tid sedan väljs ut
- FIFO (First-in First-out): Den väg i valt index som lästes in för längst tid sedan väljs ut

3.1.4 Förhämtning

För att öka prestandan är det vanligt att cachen använder så kallad förhämtning *prefetch*. Detta innebär att cachen försöker lista ut vilken nästa minnesaccess är och spekulativt hämta in denna innan den behövs. Detta är speciellt vanligt vid instruktionshämtning där det är mycket sannolikt att många cachelines kommer att läsas i sekvens, förutom i programkod som innehåller onormalt många hopp. (Detta är något du kommer att se i uppgift 1.)

Det finns dock ofta stöd för detta även vid datainläsning. Då försöker cachen upptäcka enkla mönster, exempelvis att varje, varannan, eller var fjärde cacheline läses i följd och fortsätter då spekulativt att hämta in cachelines. Det finns ofta även speciella assemblerinstruktioner som kan användas för att säga till cachen att en viss adress ska läsas in i cachen om den inte redan är inläst.

3.1.5 Tillbakaskrivningsalgoritm

Förutom läsningar måste en cache även kunna hantera skrivningar. Det finns i princip två huvudsätt att hantera detta, "write-through" och "write-back". Om en cache använder "write-through" innebär det att en skrivning uppdaterar både primärminnet och cache-minnet. Om en cache använder "write-back" så modifierar en skrivning enbart cachen och tillbakaskrivning till primärminnet sker antingen vid en cachemiss eller genom att programmet begär att så ska ske genom att den anger att en eller flera cachelines ska skrivas tillbaka (*flush*).

3.1.6 Skrivbuffer

För att få upp prestandan på write-through brukar man vanligtvis ha en skrivbuffer (*write buffer*) som ligger mellan cachen och primärminnet. Denna innehåller en buffer som lagrar ett fåtal skrivningar och ser till så att dessa skrivs ner i minnet. Den ser också till att en läsning ifrån en adress som tillhör en pågående skrivning inte läses ifrån primärminnet utan ifrån skrivbuffern. (Man kan likna skrivbuffern vid en mycket liten fullt associativ cache.)

Något annat en skrivbuffer ofta hanterar är att se till så att skrivningar till efterföljande adresser hanteras genom en bursttransaktion på bussen istället för enskilda skrivningar av varje ord. Detta sker genom att skrivbuffern väntar ett kort tag från det att en skrivning ankommer tills dess att en skrivning till primärminnet påbörjas. Kommer det under denna tid en skrivning till efterföljande adress kommer skrivbuffern att kunna kombinera dessa två transaktioner till en enda transaktion. Detta brukar kallas för *write combining*.

3.1.7 Hantering av skrivmissar

Vid en skrivning till en adress som inte är cache:ad kan man ofta välja på om cachen ska läsa in aktuell cacheline i samband med en skrivmiss eller om den enbart ska skicka skrivningen till primärminnet. Om cachen gör en sådan läsning kallas detta för *Write-Allocate*.

3.1.8 Hierarkiska cache:ar

Eftersom minnen blir långsammare desto mer data de kan innehålla brukar man ibland ha en hierarki av cache:ar. Närmast processorn sätter man vanligtvis två relativt små, men mycket snabba cache:ar, en för instruktioner och en för data. Dessa kallas för level-1 (L1) cache. Nästa cache, som finns mellan minnet och L1 minnet kallas för level-2 (L2) cache. För att få maximal prestanda brukar båda dessa cache:ar ligga på samma chip som processorn. Det är ännu så länge inte särskilt vanligt med fler än två nivåer,

men det finns processorer med hela fyra nivåer[2]. I denna laboration kommer vi dock inte att titta på cachehierarkin i detalj, utan vi kommer främst att inrikta oss på att undersöka beteendet hos L2-cachen.

3.1.9 Parametrar för cachen i Cortex-A9 på Zedboard

Den processor som används i laborationen har följande parametrar för cachen:

- L1 instruktions-cache på 32 KB
- L1 data-cache på 32 KB
- Associativitet på L1-cachen: 4-vägs
- L2 cache på 512 KB
- Cachelinestorlek i L1 och L2: ??? (Se förberedelseuppgift F.2)
- Associativitet på L2-cachen: ??? (Se labuppgift 2)
- I page-table har vi satt att alla sidor som är markerade som cachebara ska använda tillbakaskrivningspolicyn Write-back.
- Processorn har en write buffer som klarar av write-combining
- En slumpmässig ersättningspolicy används
- Klockfrekvens på AXI-bussen som är kopplad till ChipScope: 100 MHz

De parametrar som är markerade med frågetecken är tänkta att du själv ska lista ut och redovisa på laborationen. Vill du tjuvstarta på detta är det fullt tillåtet att ta reda på dessa parametrar i förväg genom att läsa i databladet för systemet, men du måste ändå kunna motivera ditt svar på laborationen genom att hänvisa till mätningar du gjort i ChipScope.

3.2 AXI-bussen

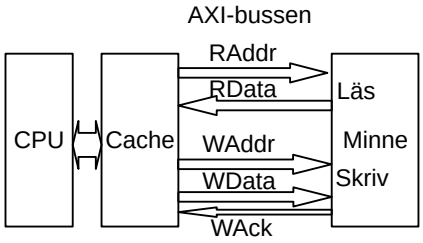
AXI3 är den systembuss som används för att ansluta Cortex-A9 till resten av systemet. Detta är en modern buss som är anpassad för att exempelvis cachemissar ska kunna hanteras snabbt och effektivt. Detta innebär att den exempelvis har stöd för burst-läsningar och burst-skrivningar. Bussen är även pipelinead vilket gör det möjligt att starta en ny transaktion utan att behöva vänta på att tidigare transaktioner blir klara.

3.2.1 Kanaler

AXI-bussen är också uppdelad i fem olika kanaler, varav två hanterar läsningar och tre hanterar skrivningar:

- Kanal för läsadresser
- Kanal för läsdata och läsbekräftelser
- Kanal för skrivadresser
- Kanal för skrivdata
- Kanal för Skrivbekräftelse

[2]Exempel: IBM zEC12, där en modul med 6 zEC12 delar på 384 MB level-4 cache.



Figur 3: Blockschema över kanaler i AXI-bussen.

3.2.2 Handskakningssignaler

För att ingen sändare på bussen ska kunna skicka mer information än mottagaren har plats för, har alla fem kanalerna handskakningssignaler. Innan mottagaren aktiverar signaler med namn som slutar på `READY` kommer inget att hända, även om sändaren aktiverar motsvarande `VALID`-signal.

3.2.3 Bursttyper

Signaler med namn som slutar på `BURST` markerar vilken typ av transaktion som ska ske:

- 0: En viss adress används hela transaktionen
- 1: Adressen ökar hela tiden
- 2: Cirkulär (Används vanligtvis för att fylla på en cacheline, med stöd för *critical word first*)

Den här signalen kommer oftast att vara 1, men i framförallt uppgift 1 och uppgift 2 är det viktigt att ni är uppmärksamma på om denna signal har ett annat värde.

3.2.4 Transaktionslängd

Du kommer antagligen ha stor nytta av signaler vars namn slutar med `LAST`, eftersom en etta här indikerar att datat som hör till en viss transaktion är färdigskickad. Det går dock också att bestämma längden på en transaktion genom att kolla på signaler med postfixen `LEN` och `SIZE`. `SIZE` berättar hur breda ord som används och kommer sannolikt att vara 2 för varenda transaktion du träffar på, vilket innebär att orden är 4 byte breda. `LEN` används för att berätta hur många ord som ingår i transaktionen. värdet noll innebär att ett ord ska överföras, värdet ett att två ord ska överföras, och så vidare.

3.2.5 Identifikationsnummer

Varje transaktion på AXI-bussen har ett identifikationsnumret i signalen `ID`. I denna laboration kommer du antagligen inte att behöva hålla reda på detta eftersom vi ännu ej sett en situation i labsystemet där en senare transaktion avslutats snabbare än en tidigare transaktion på samma buss.

3.2.6 Transaktionsstatus

Signaler med namn som slutar på `RESP` markerar om en transaktion har lyckats eller inte. I den här laborationen räcker det med att veta att värdet `00` innebär att transaktionen lyckades och att ett annat värde innebär att något gick fel. (I denna lab bör du dock aldrig se något annat än `00` här.)

3.2.7 Read Address Channel

Denna kanal används för att starta läsningar genom att önskad adress skickas till bussen. De signaler som finns i denna kanal återfinns här:

- ARADDR[31:0]: Adress vi vill läsa ifrån
- ARVALID: Master vill börja en lästransaktion
- ARREADY: Slave är redo att ta emot en lästransaktion
- ARLEN/ARSIZE: Specificerar antalet ord vi vill läsa
- ARBURST: Typ av burst-läsning
- ARID: Master skickar identifikationsnummer
- (Plus några till)

3.2.8 Read Data Channel

Denna kanal används för att bussen ska kunna skicka läsdata till processorn. Dessa signaler finns här:

- RXDATA[31:0]: Läsdata
- RVALID: Slave har giltig läsdata
- RREADY: Master är redo att ta emot denna
- RRESP: Status (lyckades transaktionen eller gick något snett?)
- RID: Slave returnerar identifikationsnummer
- RLAST: Markerar sista ordet i transaktionen

3.2.9 Write Address Channel

En läsning kan startas genom en skrivning av adressinformation till denna kanal. Följande signaler finns här:

- AWADDR[31:0]: Adress
- AWVALID: Master vill börja en skrivtransaktion
- AWREADY: Slave är redo att ta emot en skrivtransaktion
- AWLEN och AWSIZE: Transaktionens längd
- AWBURST: Typ av burst-skrivning (Cirkulär, linjär, fix)
- AWID: Master skickar identifikationsnummer

3.2.10 Write Data Channel

Denna kanal används för skrivdata och innehåller följande signaler:

- WVALID: Master har data tillgängligt
- WREADY: Slave har möjlighet att ta emot data

- WDATA: Data som ska skrivas
- WSTRB: Byte strobes (ett för de bytes i WDATA som ska skrivas till minnet)
- WLAST: Markerar sista ordet i transaktionen
- WID: Master skickar identifikationsnumret

3.2.11 Write Response Channel

Slutligen finns det en kanal som används om mastern på bussen vill ha reda på huruvida en skrivning lyckades eller inte.

- BVALID: Slave har ett svar på en skrivning
- BREADY: Master redo att ta emot svar på en skrivning
- BID: Slave returnerar identifikationsnumret
- BRESP: Status (Lyckades skrivningen eller gick något snett?)

3.3 Fixtalsaritmetik (för den intresserade)

I kursen har vi hittills lärt oss binär aritmetik med heltal. I denna laboration är det dock nödvändigt att använda rationella tal för att representera bildkoordinater i referensbilden. I datorsammanhang görs detta vanligtvis med flyttalsaritmetik eller fixtalsaritmetik (i engelsk litteratur vanligtvis kallad fixed point arithmetic). Processorn som används i denna laboration har stöd för flyttal, men tyvärr har vi inte tid att gå in på detta i detalj i denna kurs. Istället har vi valt att använda fixtalsaritmetik i denna laboration.

Idén med fixtalsaritmetik är enkel. Istället för att sätta decimalkommat till höger om LSB-biten sätter man det istället på godtycklig plats i talet. I denna laboration har vi valt att använda talformatet `sxxxxxxxxxxxxxxxxxxxxxxx,xxxxxxxx` (där s är teckenbiten). Dvs, vi kan representera tal mellan -2^{23} till $2^{23} - 2^{-8}$, i steg om 2^{-8}.

Tyvärr har de flesta processorer inget inbyggt stöd för fixtalsaritmetik, utan detta måste programmeraren själv hålla ordning på genom att utföra lämpliga heltalsberäkningar. För att exempelvis multiplicera två fixtal (a och b) multiplicerar man dessa två tal som heltal, samt skiftar resultatet ett lämpligt antal steg åt höger aritmetiskt för att kommat ska hamna på rätt ställe. I tabell 3 kan man se en sammanfattning på hur några olika operationer utförs på fixtal.

Önskad fixtalsoperation	Heltalsoperationer för att utföra detta
Addera a och b	$a + b$
Subtrahera b från a	$a - b$
Multiplikation av a och b	$ab/256$
Konvertera fixtalet a till heltal	$a/256$
Konvertera heltalet a til fixtal	$256a$

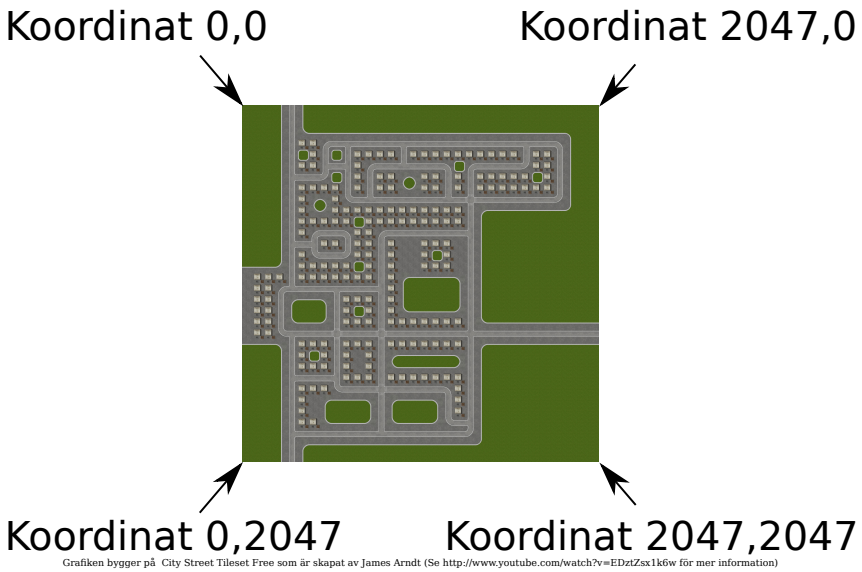
Tabell 3: Hur man med hjälp av heltalsaritmetik utför fixtalsberäkningar i formatet 24.8

3.4 Bildrepresentation

För att förstå hur `rotate.c` fungerar behöver du veta hur bilder lagras i denna laboration. Den referensbild som används i laborationen är 2048×2048 pixlar stor där varje pixel består av två bytes. (För den som är intresserad så återfinns det exakta pixelformatet i tabell 4.)

15 11	10 5	4 0
Röd färgkomponent	Grön färgkomponent	Blå färgkomponent

Tabell 4: Pixelformatet RGB-565



Figur 4: Den referensbild som ska roteras samt hur koordinatsystemet fungerar

I det koordinatsystem som vi använder i laborationen så befinner sig koordinaten (0,0) längst upp till vänster och koordinaten (2047,2047) längst ner till höger i bilden (se figur 4).

Den ordning som pixlarna lagras i minnet kan ses i figur 5. Dvs, pixeln med koordinaten (0,0) kommer först, sedan lagras pixlarna från vänster till höger fram till slutet på denna rad. Sedan börjar vi om på nästa rad och lagrar pixlarna från vänster till höger, och så vidare fram till och med slutet på rad 2047.

För att hitta förskjutningen till de två bytes som en given pixel består av kan följande ekvation användas (där x och y är heltalskoordinater).

$$\text{pixeladress} = 2(x + 2048y) + 0x81800000 \tag{1}$$

Bilden som visas upp på skärmen lagras på liknande sätt, förutom att upplösningen där är 640×480. Motsvarande ekvation för denna bild är

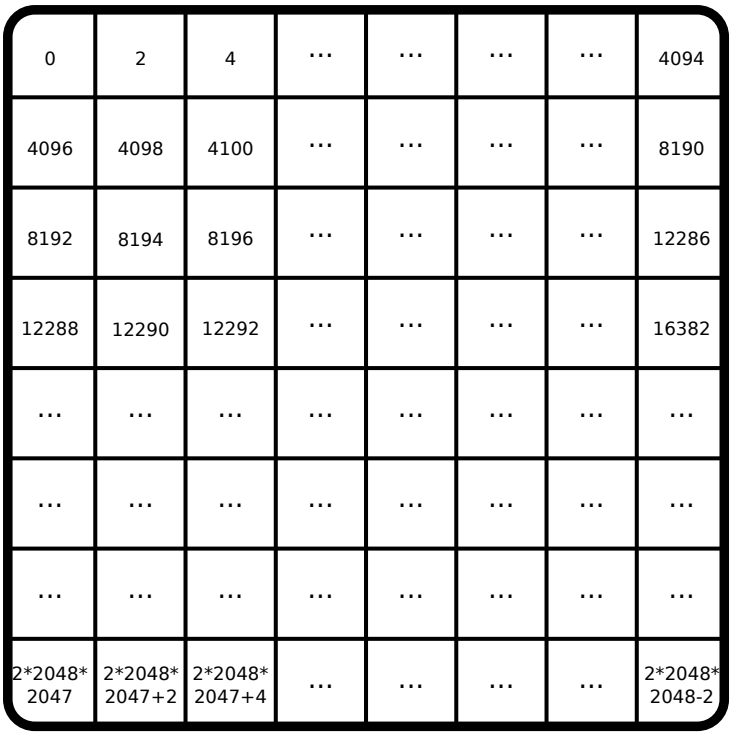
$$\text{pixeladress} = 2(x + 640y) + \text{framebufferadress}, \tag{2}$$

där framebufferadress är antingen `0x01000000` eller `0x01400000`. (Se avsnitt 3.5 för mer information om framebufferadressen.)

För att kunna rotera bilden på ett vettigt sätt behöver vi dock hålla koll på våra koordinater med högre precision än en pixel. Om vi använder formatet 24.8 får vi ändra (1) till

$$2048\lfloor y/256 \rfloor + \lfloor x/256 \rfloor \tag{3}$$

där $\lfloor z \rfloor$ betyder att z avrundas nedåt till närmaste heltal. I `rotate.c` finns det en funktion som gör just detta:



Figur 5: Byteoffset för pixlar i referensbilden. (Notera att basadressen för referensbilden är 0x81800000.)

```
static inline int fixed_point_xy_to_linear(int x,int y)
{
        return (y/256)*MAP_XSIZE + x/256;
}
```

3.5 Dubbelbuffring

Ett problem som är vanligt när en bild ska visas på en bildskärm är att det inte ser särskilt snyggt ut om bilden modifieras samtidigt som den läses ut för att visas upp på en skärm. Det sätt detta vanligtvis löses på är att det finns två stycken så kallade "frame buffers" istället för en. Den ena läses ut via DMA för att visas upp på skärmen medans programmet modifierar den andra. När bilden i den andra framebuffern ritas klart byter buffrarna roll och den andra visas upp på skärmen medans programmet modifierar den första.

I det system ni ska använda i laborationen kontrolleras allt detta genom adressen `0x40000000`. Denna adress har följande definition:

31	30 0
Aktivera utläsning	Adress till framebuffern

Om bit 31 är en etta aktiveras en DMA-enhet som läser ut den aktuella framebuffern för utläsning till bildskärm. För att starta utläsning från adress `0x01000000` behöver du alltså skriva värdet `0x81000000` till adress `0x40000000`.

För att undvika fula övergångar i bilden kommer grafikenheten också att vänta tills bilden ritats klart innan den växlar buffer. [3] Nackdelen är att processorn då också måste vänta på att grafikenheten växlar buffer innan den kan fortsätta att rita i bildminnet.

[3] Det är för övrigt detta som brukar kallas för exempelvis *"Wait for vertical sync"* i inställningarna i många datorspel.

All logik för att sköta detta är redan färdigskriven i `rotate.c`, se anropen till `framebuffer_swap()`. (Den som är intresserad kan se denna funktion i filen `util.c`).

3.6 Minneskarta

Detta är en förenkling av den minneskarta som finns i systemet.

- `0x00400000-0x004fffff`: Här laddas ditt program ned om inget annat angivits i labhäftet.
- `0x01000000-0x01096000`: Framebuffer 0
- `0x01400000-0x01496000`: Framebuffer 1
- `0x82000000-0x9ffeffff`: Cachebart DDR-SDRAM som är anslutet via den AXI-buss som ChipScope är inkopplat på Här får du göra vad du vill då detta område inte används till något annat.

Utöver minnet så finns det även några I/O-adresser som kan vara intressanta att känna till för den som vill ha en fullständig förståelse av de exempelprogram som finns med i labskelettet (se även `memorymap.h`):

- `0x40000000-0x40000007`: Kontrollregister för grafikenheten
- `0x41200000`: Hit kan du skriva för att tända/släcka de 8 lysdioder som finns på kortet
- `0xe0001000-0xe00010ff`: UART (tips: använd rutinerna i util.c för att komma åt denna)

4 Förberedelseuppgifter

Precis som i tidigare laborationer räknar vi med att ni har förberett i princip allting ni ska göra på laborationen. Däremot så lämpar sig uppgifterna nedan sig extra väl för att göra i förväg då de ej kräver tillgång till någon labhårdvara utan enbart är till för att bekanta dig lite med de koncept som används i laborationen.

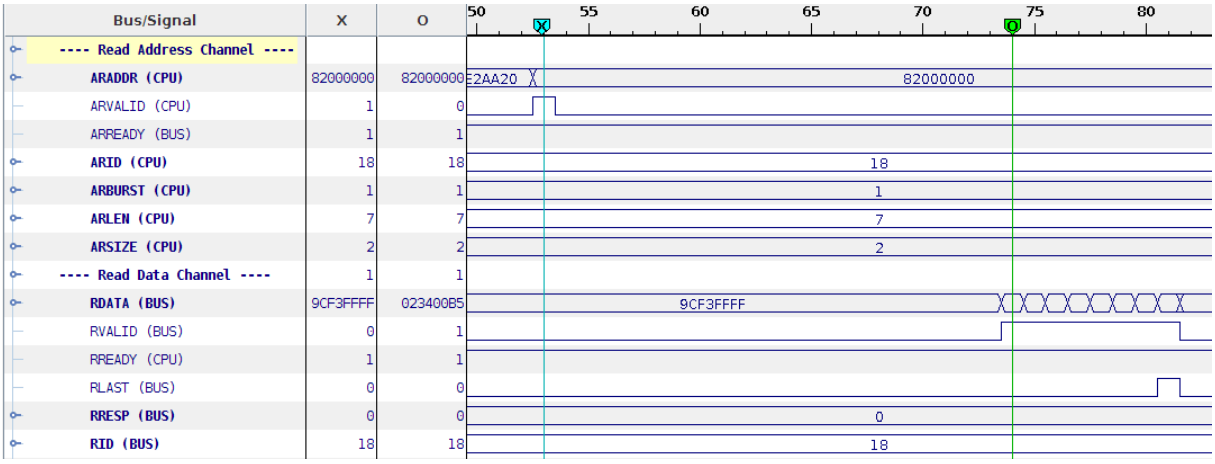
4.1 Förberedelseuppgift F.1

Antag att ett annat system har en cache med parametrarna $A = 4$,$CL = 16$ och $CM = 256$ kB, precis som i avsnitt 3.1.1 och att processorn har fått en cachemiss vid läsning på $M(0x12345678) = 0xDEADBEEF$. En CL kommer att hämtas och skrivas in i CM. Okända minnesdata kallas för X. Fyll i diagrammet nedan:

index	tag	byte0-3	byte4-7	byte8-11	byte12-15

4.2 Förberedelseuppgift F.2

I figur 6 finns ett exempel på hur det ser ut i ChipScope när en cachemiss inträffar i samband med en läsning på adress `0x82000000` i labbsystemet. Notera att detta, till skillnad ifrån F.1, är taget direkt ifrån det system ni ska använda i laborationen och alltså inte ett påhittat exempel.



Figur 6: En typisk läsburst på AXI-bussen i samband med en cachemiss

Givet busstrafiken i figur 6 så kan du bestämma några viktiga parametrar i det system du använder i laborationen; Hur lång fördröjning, mätt i klockcykler, från det att en läsning startar till att det första ordet i burstläsningen kommer tillbaka? Hur lång är en cacheline (antal byte)?

5 Uppgift 1 - Instruktionshämtning

Programmet insnfetch kan kompileras med hjälp av att köra kommandot `make insnfetch` i labskelettet. Detta program är gjort så att funktionen `fetchtest()` laddas ner till adress `0x82000010` vilket gör det möjligt att se alla minnesläsningar som görs via instruktionscachen. För att se vilken assemblerkod som programmet har kompilerats till, ta en titt i filen `insnfetch.dis`. (Jämför gärna med källkoden i `insnfetch.c` och `fetchtest.c`).

För att testköra detta program, skriv `make insnfetch`, ladda sedan ner hexfilen `insnfetch.hex` till monitorn (se avsnitt 2). Tryck sedan på "play"-knappen i ChipScope för att starta en mätning. Slutligen kan du starta programmet genom kommandot `g` i monitorn.

Analysera busstrafiken och identifiera varje ord som förekommer på läskanalen i de fyra första transaktionerna och förklara varför just dessa ord har lästs in (tips: Valid-signalerna är aktiva under de klockcykler som är av intresse). Notera nedan varje klockcykel där data överförs till/från CPU (både till och från). Notera bara data eller adress på varje rad.

Som exempel på hur tabellen fylls i visar figur 7 hur läsningen i figur 6 beskrivs i tabellen.

tid	arid	address	arburst	arlen	rid	rdata	kommentar
53	18	82000000	1	7			läsadress
74					18	023400B5	1:a data (t ex kod rad xxx eller data)
75					18	12345678	2:a data (värdet framgår inte i figur)
76					18	9ABCDEF0	3:e data - ' ' -
77					18	2468ACE0	4:e data - ' ' -
78					18	13579BDF	5:e data - ' ' -
79					18	FEDCBA98	6:e data - ' ' -
80					18	76543210	7:e data - ' ' -
81					18	DEADBEEF	8:e data - ' ' -

Figur 7: Beskrivning av läsning i figur 6. Notera att bara rader där Valid=1 på kanalen har angivits.

Kom ihåg att assemblerkoden för programmet som körs kan ses i filen `insnfetch.dis` vilket betyder att det går att bestämma exakt vid vilken tidpunkt processorn får reda på vilken instruktion som ska utföras.

När subrutinen fetchtest startas har main-programmet satt r0=0x85123456.

tid	arid	address	burst	len	rid	rdata	kommentar

Vid vilken tidpunkt skickades minnesadressen till minnet som skulle inkludera hämtning av 1:a instruktionen i funktionen `fetchtest()`?

Vid vilken tidpunkt skickades minnesadressen till minnet som skulle inkludera läsning av adress `0x85123456`? Tips: Tänk på att adressen som skickas motsvarar antingen starten på en cacheline eller mitten av en cacheline.

Vid vilken tidpunkt skickar minnet tillbaks värdet på minnesadress `0x85123456`? Tips: data från 4 adresser per klockcykel skickas från minnet.

Hur många klockcykler tar det från det att första instruktionen i funktionen `fetchtest()` börjar hämtas tills processorn fått data från minnesläsning från den plats som pekaren `test` pekar på (dvs från att adress till första instruktionen i fetchtest skickas tills data från adress `0x85123456` tas emot)?

Hur många instruktioner har körts av subrutinen fetchtest när data från `0x85123456` tagits emot? Se insnfetch.dis för att se fetchtest-subrutinen och vilka värden som motsvarar vilka instruktioner i koden.

Vilka av de fyra första transaktionerna beror på förhämtning? Tips: Inte alla läsningar beror på en cachemiss.

6 Uppgift 2 - Bestäm datacachens associativitet

I denna uppgift ska du bestämma cachens associativitet genom att göra ett lämpligt antal läsningar på lämpliga ställen i minnet. Se `assoc.c` och se till att anropa `find_associativity()` med lämpliga parametrar (dvs, du får i filen byta ut `ANTAL` respektive `STEGLANGD` till lämpliga siffervärden så `nr` och step i funktionen `find_associativity` får korrekta värden). Innan du kommer till laborationen är det en klar fördel om du har tittat på `assoc.c` så att du vet vad programmet gör och förstår hur du kan använda detta för att hitta cachens associativitet.

De intressanta raderna är:

```
j = 0;                    /* Kommentar: starta med j satt till 0 */
do {                      /* Loop över alla värden hos j (0,1,2,3)   */
  i = 0;                  /* För varje j starta med i=0 */
  do {                    /* Loop över alla värden 0 till (nr-1) */
    c = read_mem32(0x82000000 + i * step);
    i = i + 1;
  } while (i < nr);       /* Kör loop för i = 0,1 ... nr-1 */
  j = j + 1;
} while (j < 4);          /* Kör loop för j = 0,1,2,3 */
```

Dvs om `step=0x100` och `nr=3` kommer `read_mem32()` anropas med adresssekvensen 0x82000000, 0x82000100, 0x82000200, 0x820000000, 0x82000100, 0x82000200,0x82000000, 0x82000100, 0x82000200, 0x820000000, 0x82000100, 0x82000200. Tips: Vad skiljer en sekvens av läsningar som får plats i cachen mot en sekvens som inte får plats i cachen?

Bygg programmet (`make assoc`), ladda sedan ner `assoc.hex` i monitorn, starta en mätning i ChipScope och kör sedan programmet.

När programmet körs kommer först datacachen att tömmas (för att vår mätning inte ska störas av att du exempelvis kört `assoc.c`. Sedan skriver `trigger_logic_analyzer()` till `0x9fff0000` vilket startas en mätning i ChipScope. Slutligen körs `find_associativity()` som gör ett antal läsningar på adress `0x82000000` och framåt.

Notera att det kan komma någon enstaka extra läsning (1-2 styck) pga störningar orsakade av utskriften av värdet.

- Datacachens storlek är 512 kB. Föreslå ett lämpligt värde på parametern `step` för att läsningarna garanterat ska placeras på samma index (rad) i cachen. Ledning: Om du antar att cachen är direktmappad när du räknar ut detta värde så kommer du att täcka in det värsta fallet. [4]

- Ändra parametern `nr` och räkna minnesaccesser med `ChipScope`. Prova minst 2 och dokumentera i en tabell antal läsningar som syns för respektive värde på `nr`. Vilken är cachens associativitet? Rita upp en graf med antal cachemissar som funktion av antal olika adresser som läses.

[4] Notera att cachen som används i Zedboard inte nödvändigtvis har samma konfiguration som den cache som diskuteras i avsnitt 3.1.2.

Detta är ett bra tillfälle att redovisa dina labbresultat så här långt!

7 Uppgift 3 - Rotation av bild

Denna uppgift går ut på att snabba upp ett program genom att läsa minnet på ett sådant sätt att cachen utnyttjas mer effektivt.

Den applikation som du ska titta på är rotation av bild. Detta program ligger i `rotate.c` och är för tillfället implementerat på ett rättframt (men långsamt) sätt. På adress 0x81800000 och framåt finns det en referensbild som har upplösningen 2048×2048 pixlar. **(Denna referensbild skapas genom att du i monitorn använder kommandot `b`.)** Denna bild ska ritas upp i ett roterat skick i en framebuffer som en VGA-enhet sedan läser ut via DMA. Denna framebuffer har upplösningen 640×480 pixlar.

7.1 Användargränssnitt

När du laddar ner och kör programmet (kompilera med `make rotate` och ladda ner `rotate.hex`) kan du styra programmet genom att använda följande knappar:

- `a`: Rotera motsols
- `d`: Rotera medsols
- `w`: Öka hastigheten framåt
- `s`: Minska hastigheten framåt
- Mellanslag: Sätt hastighet till 0
- `D`: Demoläge där bilden kontinuerligt roterar. Det är antagligen detta läge du kommer att använda oftast när du utför denna uppgift.
- `q`: Avsluta programmet och hoppa tillbaka till monitorn

7.2 Prestandaanalys, ursprunglig version

När du kör programmet kommer du att märka att programmet går mycket långsammare när det ska rotera i närheten av 90° och 270°. Framförallt när du kör i demoläget kommer detta att märkas tydligt. Du kommer även se i `TeraTerm` att programmet skriver ut texten "LAG" när det inte hinner med att uppdatera bilden i 60 bildrutor per sekund.

Fallet 0°, omodifierad programkod

- Hur lång tid tar det att rita ut första raden i fallet 0°?[5]

- Hur många cachemissar får du?[6]

- Tar alla rader lika lång tid att rita ut? (Undersök ett par olika rader och mät tid och/eller antalet cachemissar.)

- Är svaren ovan rimliga? (Ledning: Fundera på bildens bredd samt hur lång en cacheline är. Jämför tiden för en rad med tiden programmet skriver ut att det tar att rita ut en hel frame (bussens klockfrekvens är 100 MHz)

[5] Tips: Adressen för starten av 2:a raden ligger 2048 pixlar längre bort än adressen för 1:a radens start.

[6] Ledning: Signalen "Read Transaction Counter" i ChipScope räknar antalet minnesläsningar, vilket i ert fall kommer att vara samma sak som antalet cachemissar.

Fallet $90°$, omodifierad programkod

- I detta fall tar första raden mer än 8192 klockcykler att rita upp vilket innebär att du inte kommer att få in alla cachemissar en rad åstadkommer på en och samma mätning. Mät istället hur många cachemissar du faktiskt får in på dessa 8192 klockcykler.

- Hur många cachemissar bör du få i samband med att den första raden ritas ut i fallet $90°$ (du behöver ej göra någon mätning i ChipScope för att svara på denna fråga)?

- Uppskatta hur lång tid den första raden tar att rita ut utifrån dessa värden.

- Bör du få lika många cachemissar på andra raden i detta fall? Ser detta ut att stämma om du gör om samma mätning på denna rad i ChipScope?

- Är dina mätvärden ovan rimliga? Ledning: Jämför dina mätvärden med hur lång tid det tar att rita ut en hel frame (skrivs ut på skärmen för varje bild).

Förklara varför det blir så många fler cachemissar vid $90°$ där skärmen läses ut kolumnvis jämfört med $0°$, där skärmen läses ut radvis. (Rent intuitivt borde det inte vara någon märkbar skillnad, eftersom 640 pixlar (som är 2 bytes stora) mer än väl borde rymmas i systemets cache som är på 512 KiB.) Tips: Titta på vilka adresser som läses när 1:a raden i framebuffern fylls. Räkna ut vilka platser i cachen som då används.

Använd ChipScope för att verifiera att ditt antagande stämmer. Tips: Se avsnitt 2.4.1 för information om hur du kan mäta tid i ChipScope.

7.3 Programmets upplägg

C-versionen är skriven för att vara så lättförståelig som möjlig för personer som inte är vana vid C eller C++. Inget standardbibliotek används, istället finns alla funktioner som anropas i programmet med i de filer som följer med labskelettet (`util.c`, `util.h` samt `sintable.c`). Vill du skriva ut något kan du använda en reducerad version av `printf()` som kallas för `small_printf()`[7].

`rotate.c` innehåller följande funktioner som kan vara bra att känna till.

- `fixed_point_xy_to_linear()`: Konverterar koordinater i fixtalsformat till en offset i framebuffern (se avsnitt 3.4)
- `copy_pixel()`: Kopiera en pixel från referensbilden till framebuffer, baserat på x och y koordinat i frambuffern.

[7] Som Makefile är skriven just nu är det i själva verket omöjligt att använda i princip alla funktioner ifrån standardbiblioteket i C, men du torde inte ha något behov av detta i vilket fall som helst.

- **`render_all_lines()`**: Får startkoordinat och information om avstånd mellan pixlar i x och y-led. Funktionen ska i tur och ordning ange alla x,y koordinater och anropa `copy_pixel()` för att kopiera alla pixlar från referensbilden till framebufferten.
- `rotate_image()`: Denna funktion beräknar, baserat på startposition och roteringsvinkel, startkoordinat samt dx och dy i referensbilden för alla linjer som ska ritas upp i framebuffern. Därefter anropar den den `render_all_lines()` för att kopiera alla pixlar
- `paintloop()`: Detta är huvudloopen som är ansvarig för användarinterfacet som presenteras i `TeraTerm` samt att hålla ordning på vilken framebuffer som ska skickas till bildskärmen och vilken som får modifieras av `rotate_image()`.
- `redraw_reference_image()`**:** Denna rutin är tänkt att användas om du på något sätt vill rita om referensbilden i början av programmet. I labskelettet du får är denna rutin dock tom.
- `main()`: Detta är den funktion som anropas när du startar ett nerladdat program med kommandot `g` i monitorn.

För att klara laborationen behöver du endast modifiera de funktioner som är markerade med fetstil. Övriga funktioner behöver du inte sätta dig in i närmre om du väljer att göra en av de lösningar som rekommenderas i detta labhäfte. Du kan eventuellt vilja ändra vissa parametrar i filen `config.h` också som innehåller definitionen av exempelvis `FB_XSIZE` och `FB_YSIZE` som håller reda på hur bred respektive hög framebuffern är. Det är dock fritt fram att ändra i andra funktioner samt andra filer om du känner för det.

7.3.1 Funktionen `render_all_lines()`

Denna funktion använder de globala variabler som beräknas av textttrotate_image(). Dessa globala variabler inkluderar startposition i referensbilden (för koordinat 0,0), hur långt bort nästa pixel (koordinat 1,0) ligger från startpositionen i referensbilden beskrivet som delta-x och delta-y, samt var någonstans framebuffern finns. Figur 8 exemplifierar hur detta går till.

Denna funktion ska räkna upp alla koordinater i framebufferten (640 pixlar på varje rad, 480 rader) och anropar `copy_pixel()` för varje sådan koordinat. Notera att koordinaterna avrundas ifrån fixtalsformat till heltalsformat genom att decimaldelen kapas bort. Se även `fixed_point_xy_to_linear()` i `rotate.c` samt avsnitt 3.4.

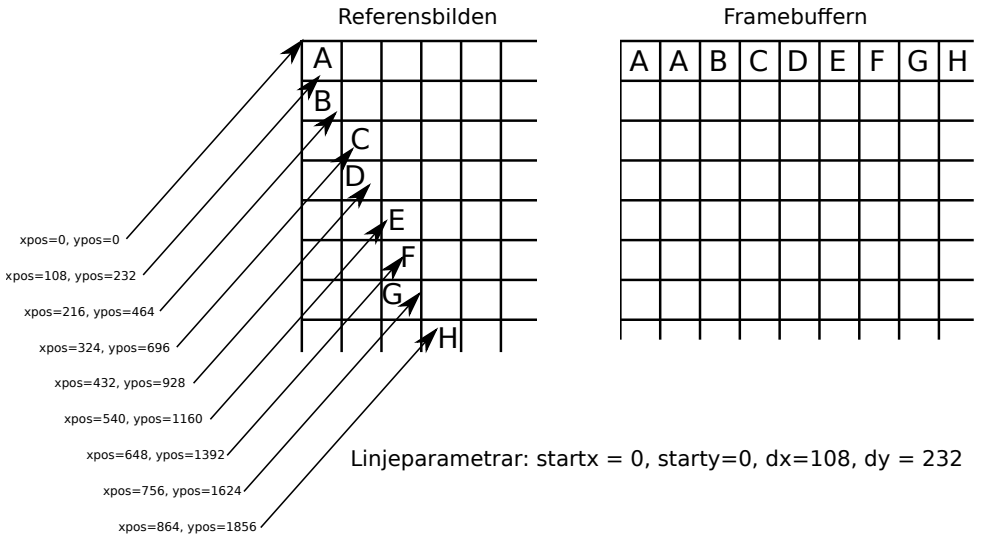
I koden finns också exempel på hur ni kan trigga ChipScope för en viss koordinat och även hur ni kan skriva ut värden på variabler med hjälp av `small_print`. Det finns även några extra variabler definierade (i,j,k,l) som ni kan använda om ni behöver.

De globala variablerna `image_dx` och `image_dy` har funktionen `rotate_image` bestämts med hjälp av elementär linjär algebra.

7.4 Problemformulering

Din uppgift är att snabba upp `rotate.c` genom att ändra på programmet så att referensbilden läses på ett sätt som är mer anpassat till hur cachen fungerar. Du får göra detta på valfritt sätt. Målet är att det ska gå såpass snabbt att köra rotationen så att det alltid går att rotera referensbilden i 60 bilder per sekund oavsett vinkel. (Men det är OK om det är något långsammare än så, så länge en markant förbättring har skett vid rotation $90°$ respektive $270°$.)

Du får egentligen lösa uppgiften på i princip valfritt sätt, så länge du kan förklara varför uppgiften går snabbare genom att använda ChipScope för att analysera busstrafiken. För att göra det enklare för dig



Figur 8: Ett exempel på hur `render_all_lines()` kopierar pixlar ifrån 1:a linjen med angiven vinkel i referensbilden till en horisontell linje i angiven framebuffer. Notera att bildkoordinater är angivna på fixtalsformen 24.8.

har vi dock två förslag på hur du kan lösa uppgiften. Innan du läser detta rekommenderas du dock att diskutera uppgiften med din labpartner för att se om ni själva har några tankar om hur detta ska gå till.

Analysera prestandan för ditt förbättrade program på samma sätt som du gjorde med originalprogrammet. **Notera dock att om du har delat upp en rad så att den ritas i olika segment kommer dina mätvärden nedan troligtvis handla om ett segment istället för en rad.**

Fallet $0°$, med dina modifikationer

- Hur lång tid tar det att rita ut första raden (i 1:a kolumnen) i fallet $0°$?
- Hur många cachemissar får du?[8]?

- Tar alla rader lika lång tid att rita ut? (Undersök ett par olika rader och mät tid och/eller antalet cachemissar.)

- Är svaren ovan rimliga? (Ledning: Fundera på bildens bredd samt hur lång en cacheline är. Jämför tiden för en rad med tiden det tar att rita ut en hel frame samt jämför tiderna med värdena du fick tidigare i laborationen när du körde den omodofierade versionen.)

Fallet $90°$, med dina modifikationer

- Hur många cachemissar bör du få i samband med att den första raden (i 1:a kolumnen) ritas ut i fallet $90°$ (du behöver ej göra någon mätning i ChipScope för att svara på denna fråga)?

- Använd ChipScope för att mäta hur lång tid den första raden tar att rita ut. . Hur många Cachemissar får du?

[8]Ledning: Signalen "Read Transaction Counter" i ChipScope räknar antalet minnesläsningar, vilket i ert fall kommer att vara samma sak som antalet cachemissar.

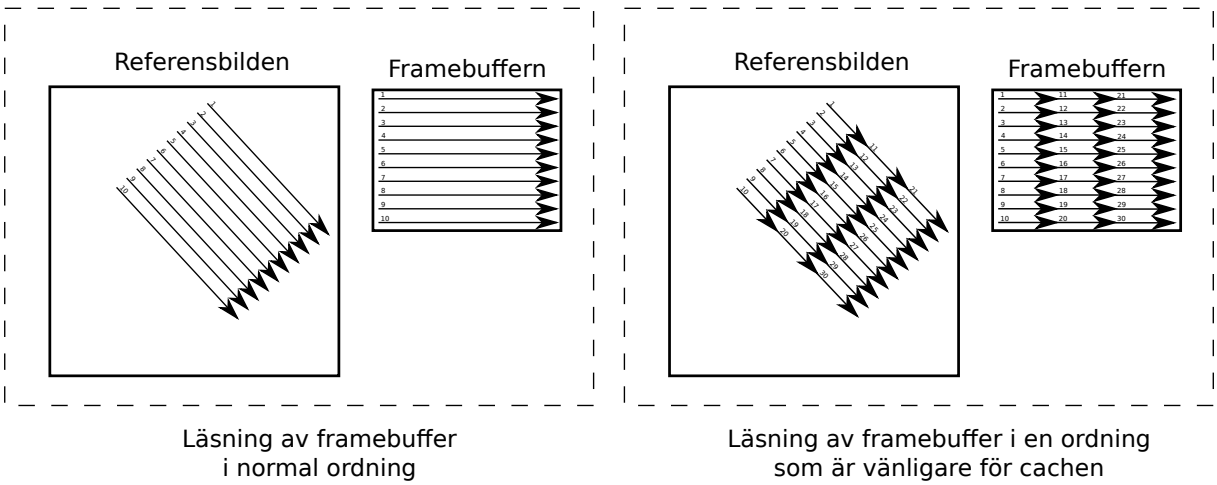
- Bör du få lika många cachemissar på andra raden i detta fall? Ser detta ut att stämma om du gör om samma mätning på denna rad i ChipScope?

- Är dina mätvärden ovan rimliga? Ledning: Jämför dina mätvärden med hur lång tid det tar att rita ut en hel frame.

Varning: Spoilers på nästa sida!

7.5 Parallel läsning/skrivning av flera linjer samtidigt

Just nu ritas raderna ut i den ordning som anges till vänster i figur 9. Om detta ändras till den ordning som ses till höger i figur 9 är det möjligt att få en signifikant uppsnabbning i fallet rotation nära $90°$. Tanken är helt enkelt att dela upp varje linje i flera delar och rita upp dessa var för sig.



Figur 9: Ett mer cachevänligt sätt att rita upp den roterade bilden på ett annat sätt

Denna modifikation görs antagligen enklast genom att `render_all_lines()` modifieras så att infoarrayen gås igenom flera gånger men att den varje gång enbart ritar upp kortare linjer. Detta kan antingen göras genom en yttre loop som räknar linjeavsnitt (samt att ni justerar start och slut på linjesegmenten korrekt), eller genom att duplicera koden (kopiera, klistra in) med olika start och slut i varje del av koden.

7.6 Omorganisation av referensbilden

Ett annat sätt att snabba upp programmet är att organisera om referensbilden. Enklast är antagligen att göra så att det tar mer än 4096 bytes att stega ner en rad i referensbilden genom att göra bilden lite bredare samt att lägga till en svart kant längst till höger. För att göra detta måste du dels ändra i `config.h` och dels ändra i `redraw_reference_image()`

8 Extrauppgifter

I detta appendix finns det förslag på extrauppgifter du kan göra i mån av tid. Ingen uppgift här är obligatorisk att faktiskt implementera på Zedboard, men du rekommenderas att försöka dig på åtminstone någon av dessa uppgifter om du har gott om tid över.

Det är också möjligt att du i samband med den muntliga examinationen av laborationen kommer att behöva resonera runt någon av dessa uppgifter, även om du inte behöver implementera den.

8.1 Implementera skalning av bilden

Just nu kan `rotate.c` enbart skala bilden. Ändra i `rotate.c` så att du även kan zooma in och zooma ut. Vad får denna modifikation för inverkan på antalet cachemissar per frame?

8.2 Bilinjär interpolation

Just nu avrundas koordinaterna till närmaste hela pixel. Detta lämnar dock en hel del att önska vad gäller bildkvalitet. Det är snyggare att använda ett viktat medelvärde av de fyra pixelvärden som omger den koordinat vi önskar läsa ut, där vikterna av de olika pixlarna beror på hur nära vi är. Ett sätt att lösa detta på som är vanligt i grafiksammanhang är så kallad bilinjär interpolation. Förändra `copy_pixel()` så att bilinjär interpolation används.

Vad får denna modifikation för inverkan på antalet cachemissar per frame?

8.3 Jämförelse av cachead och ocachead minnesbandbredd

Modifiera assoc.c så att du kan analysera skillnaden på att läsa ifrån minne som är cacheat respektive minne som ej är cacheat. (På adress 0x9fff0000 och fram till 0x9fffffff finns det DDR-minne som ej är cache:at som page-tables sätts upp av monitorn.)

8.4 Analys av skrivningar till framebuffern

Just nu tittar vi enbart på läsningar ifrån referensbilden. Modifiera `rotate.c` så att även skrivningar till framebuffers går via den buss som ChipScope kan använda. (Dvs ändra framebufferadressen så att den skriver till 0x81000000 och 0x81400000 istället för 0x01000000 och 0x01400000.)

Revisionshistorik

Versionsnummer	Ändringar ifrån tidigare version
v1.0	Första preliminära versionen
v1.1	Nästan total omskrivning
v1.2	Tog bort en spårutskrift i `rotate.c` som hamnat där av misstag Bytte namn på `render_lines` till `render_all_lines` för att undvika förvirring. Lade till en bild på hur referensbilden ser ut. Lade till en beskrivning av UI:t i rotate fungerar Allmän finputsning av labmanualen Allmän finputsning av labskelettet Lade till revisionshistoriken
v1.3	Smärre språkändringar assoc.c och insnfetch.c använder adress $82000000 inte $80800000 Ändrade om i labskelettet för att minska antalet C-finesser som används Lade till förslag på extrauppgifter för de som är klara snabbt
v1.4	Triggar nu logikanalysatorn även ifrån `line.S` i labskelettet Snyggade till avsnittet om ARM-instruktioner lite.
v1.5	Förtydligande av förberedelseuppgift F.1
v1.6	Förtydligande av frågor i uppgift 1 Litet förtydligande av uppgift 2 Assemblerversion av uppgift 1 i labskelettet Använde lite mindre finesser ifrån C i `rotate.c` Makefile kompilerar nu bara om de filer som faktiskt har förändrats sedan senaste körningen Gick över till ZIP-format för labskelettet
v1.7	Rättade figur 9 så att bredden är 2048 pixlar a 2 bytes styck Lade till information om `DUMP_REGISTERS`
v1.8	Rättade en bugg i labskelettet så att `fetchtest()` faktiskt anropas på adress $82000010 Adress $81800000 används till referensbilden, ej $80180000
v1.9	Bytte plats på uppgift 1 och 2. Flyttade om lite frågor. Lade till forms i PDF-filen.
v1.10	Förtydligande av framförallt vissa av mätuppgifterna.
v1.11	Förtydliga och förenkla
v1.12	Flytt till Windows, förenkla rotationsuppgift, ta bort assembler
v1.13	Korrigering andra exemplet 3.1.2
v1.14	Byte till C-stil på hexadecimala tal
v1.15	Ny OS-version
v1.16	Distansläge
v1.17	Bättre tabell och exempel uppgift 1
v1.22	Förtydliganden, omformuleringar av uppgifter