LABORATION

TSEA28

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

Version: 2.2
2013 (OS,AE) 2014 (AE) 2025 (KP)

Namn och personnummer	Godkänd

blank sida

Innehåll

blank sida

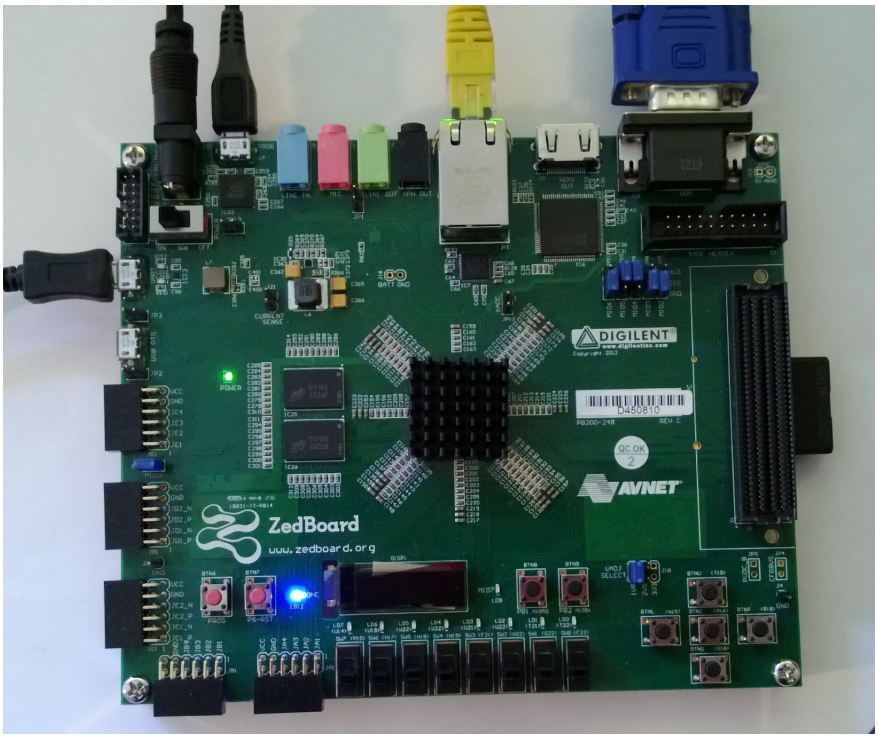
1 Inledning

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 fulla manualen för den krets vi använder (Zynq-7000) finns på `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf` för den som är nyfiken, men tanken är att ni ej ska behöva läsa alla dessa sidor...

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:`. Undvik sökvägar med mellanslag eller konstiga tecken.

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 söka efter programmet **Vitis HLS 2023.2 Command Prompt**. Ge kommandot `settings64` för att ge tillgång till programmen som ska användas.
- Förflytta dig nu 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 `Tera Term` med från start-menyn. Där behöver några inställningar göras först: Ett nytt fönster öppnas först. Välj där **Serial** och **Port** sätts till USB Serial Device. Därefter väljs meny **Setup** → **Serial port...** Välj **Speed:** 230400.
- 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]
Misc:
  p <val> [Set parport]
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: 0x00400a40`.
- Programmet `hello` kan nu startas med kommandot `g` i `Tera Term`.
- Nu ska programmet skriva ut texten `Hello world` och sedan ska du få en ny prompt i Tera Term.

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 Vivado

Vivado är ett stort utvecklingssystem för hårdvara, och bland annat innehåller den en logikanalysator som ska användas i denna laboration. För att starta denna startar du först **Vitis HLS 2023.2 Command Prompt**. Om en varning dyker upp om windows brandvägg eller access till nätverket kan du välja **Cancel**.

När **Vitis HLS 2023.2 Command Prompt** startat kör du först programmet `settings64`. Därefter startas Vivado med kommandot `vivado -script k:\tsea28\lab5\labsetup.tcl`. Detta startar chipscope och ställer in programmet så det visar vad som händer på bussen mellan processor och minne.

Nu kan du i vivado trycka på "play"-knappen för att starta en mätning. Vivado 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 `Tera Term`:

```
m 9fff0000 1234abcd
```

Efter ett litet tag ska du nu i vivado-fönstret få upp den busstrafik som inträffade på `AW, W, B Channel` i samband med detta. Du behöver antagligen zooma in runt tiden 0 klockcykler för att du ska kunna se detta ordentligt. För att zooma in kan du antingen klicka på zoomknapparna eller i signalfönstret trycka ned vänsterknappen och dra snett nedåt höger för att zooma in ett speciellt område. Alla tider anges i klockcykler.

Om det inte dyker upp några värden i waveform-fönstret bör det räcka att ge kommandot i `Tera Term` en gång till.

2.4.1 Att mäta tiden mellan två händelser i vivado

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 läsa av tidsskalan kan man använda den gula cursor som finns, vilken visar i den gula rutan vilken klockcykel den markerar. För att mäta avstånd mellan den gula cursorn och andra händelser kan man lägga till en markör. Om markören placeras på en tidpunkt så visas en tidsskala längst ned i fönstret som har markörens position som tid 0, och det visas även hur många klockcykler det är till cursorns position (eller andra markörer).

För att lätt räkna ut tiden mellan två händelser i vivado kan du lägga lägga till en markör och placera den på den ena tidpunkten, vilket ger en gul tidsskala längst ned i fönstret med tidpunkt 0 på tiden som markören placerats på. Flytta sedan den gula markören till den andra tidpunkten. Skalan längst ned visar då hur långt det är mellan den och välja place O cursor. Tidsdifferensen (mätt i klockcykler) visas längst ner till höger. Notera att Vivado kan spara data ifrån maximalt 4192 klockcykler i den konfiguration vi använder just nu. Längre tidsperioder än så går alltså inte att mäta i vivado.

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

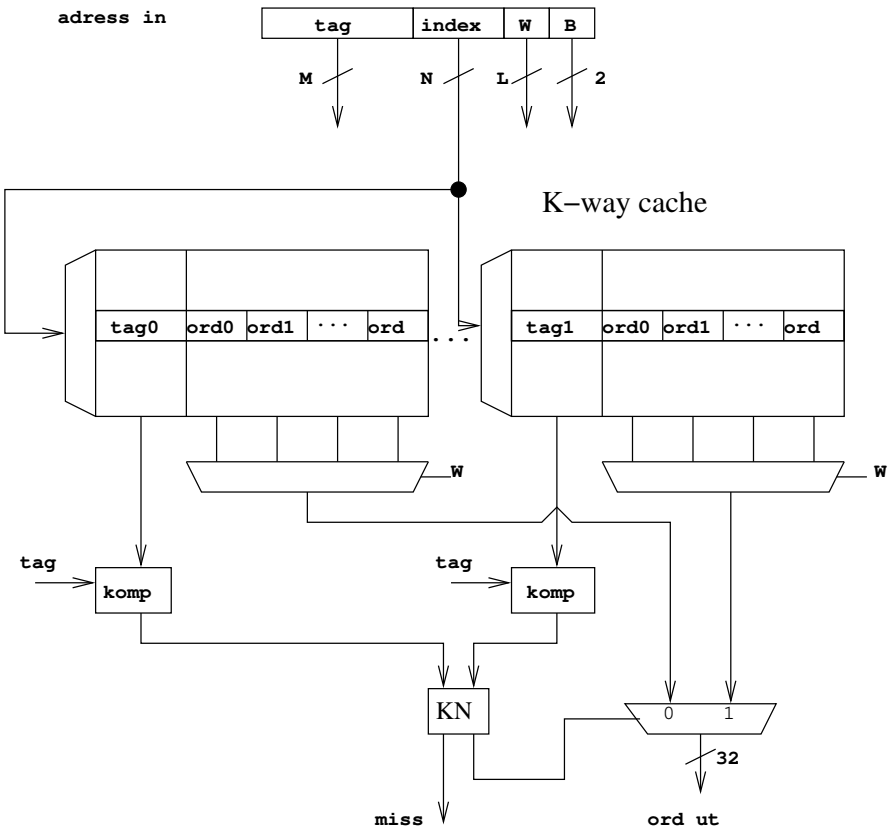
Figur 2 visar ett gruppassociativt (*set-associative*) cacheminne (CM) med K vägar. Detta CM består alltså av K st identiska del-CM. Lägg märke till att detaljer för skrivning i CM inte är utritade.

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. Bytepositionen i en cacheline blir alltså kobminationen av W och B fälten.
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.



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

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

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

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

uppdelad i tag index och W enligt 1000 1001 1010 101 $\|$ 1 1100 1101 111 $\|$ 0 11 $\|$ 00. Alltså, tag = 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[1]. 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

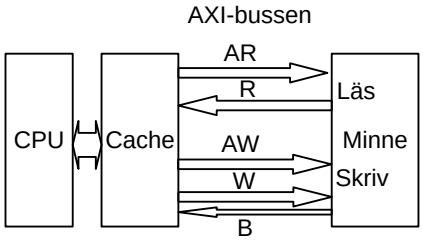
AXI3 är den systembuss som används för att ansluta Cortex-A9 till resten av systemet. Detta är en modern buss som är anpassad för att exempelvis cachemissar ska kunna hanteras snabbt och effektivt. Detta innebär att den exempelvis har stöd för burst-läsningar och burst-skrivningar. Bussen är även pipelinead 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 (AR)
- Kanal för läsdata och läsbekräftelser (R)
- Kanal för skrivadresser (AW)
- Kanal för skrivdata (W)
- Kanal för Skrivbekräftelse (B)

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

Den här signalen kommer oftast att vara 1, men det är 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 identifikationsnummer 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` (OKAY) 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 AR

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 R

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

- RDATA[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 AW

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 W

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

3.2.11 Write Response Channel B

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 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.
- `0x82000000-0x9ffeffff`: Cachebart DDR-SDRAM som är anslutet via den AXI-buss som vivado kan läsa av. 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`):

- `0x41200000`: Hit kan du skriva för att tända/släcka de 8 lysdioder som finns på kortet
- `0xe0001000-0xe00010ff`: UART

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.

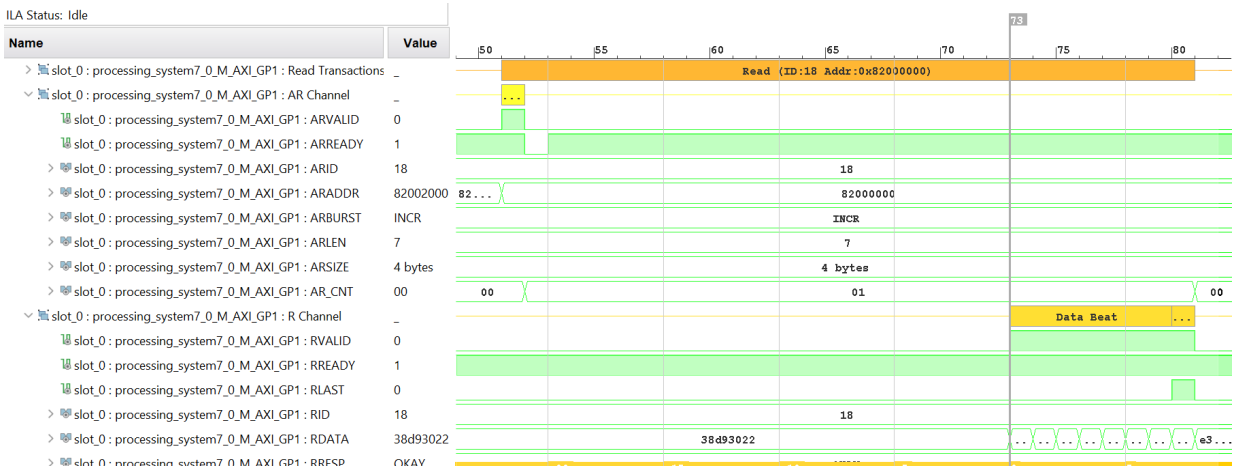
4.1 Förberedelseuppgift F.1

Antag att cachens parametrar är $A = 4$,$CL = 16$ och $CM = 256$ kB, precis som i avsnitt 3.1.1 och att processorn har fått en cachemiss vid läsning på $M(0x12345678) = 0xDEADBEEF$. (OBS: labutrustningen har en annan uppsättning parametrar!) 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 4 finns ett exempel på hur det ser ut i vivado när en cachemiss inträffar i samband med en läsning på adress `0x82000000`. 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 4: En typisk läsburst på AXI-bussen i samband med en cachemiss.

Givet busstrafiken i figur 4 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.

Programmet insnfetch anropar funktionen fetchtest med ett addressargument med värde `0x85123456`, och funktionen fetchtest adderar sedan värdet på de 15 adresserna från `0x85123456` och framåt.

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 samt transaktionen som läser adress `0x85123456`. Förklara varför just dessa ord har lästs in (tips: Valid-signaler är aktiva 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 5 hur läsningen i figur 4 beskrivs i tabellen.

tid	arid	address	arburst	arlen	rid	rdata	kommentar
51	18	82000000	INCR	7			läsadress
73					18	38d93022	1:a data
74					18	42345678	2:a data (exakt värde framgår inte i figur)
75					18	0ABCDEF0	3:e data - ' ' -
76					18	E468ACE0	4:e data - ' ' -
77					18	23579BDF	5:e data - ' ' -
78					18	CEDCBA98	6:e data - ' ' -
79					18	06543210	7:e data - ' ' -
80					18	3EADBEEF	8:e data - ' ' -

Figur 5: Beskrivning av läsning i figur 4.

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`?

Vid vilken tidpunkt skickar minnet tillbaks värdet på minnesadress `0x85123456`?

Hur många klockcykler tar det från det att första instruktionen i funktionen `fetchtest()` ska hämtas (dvs adressen till första instruktionen skickas till minnet) tills processorn fått data från minnesläsning från den plats som pekaren `test` pekar på (dvs adress `0x85123456` som specificeras i insnfetch.c)?

Hur många instruktioner har körts av funktionen `fetchtest()` innan programmet försöker läsa adress `0x85123456`? Se insnfetch.dis för att se fetchtest-subrutinen.

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 ändra ANTAL respektive STEGLANGD till lämpliga siffervärden). 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(0x82F08000 + 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=0x10` och `nr=3` kommer `read_mem32()` anropas med adresssekvensen 0x82F08000, 0x82F08010, 0x82F08020, 0x82F080000, 0x82F08010, 0x82F08020,0x82F08000, 0x82F08010, 0x82F08020, 0x82F08000, 0x82F08010, 0x82F08020. 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 `0x82F08000` 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. [2]

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

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

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 **??** 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
v2.0	Design byggd med vivado
v2.1	Ljust tema på vågform, omformulerad introduktion uppgift 1
v2.2	Search for Vitis 2023.1 Command prompt, change base address in task 2