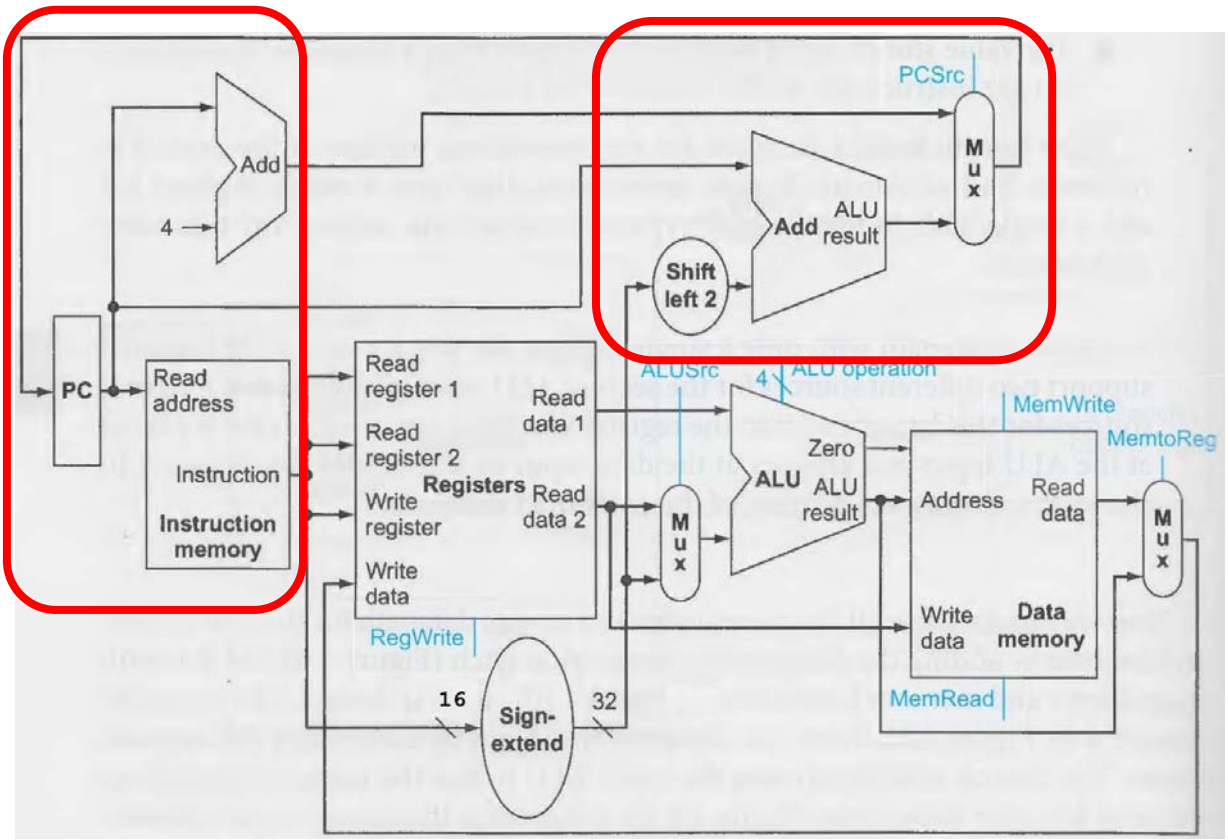
Tentamen DV1464/DV1493 Datorteknik

Datum 2019-06-05
Tid: 15:00-20:00
Hjälpmedel: Räknedosa (tömd)

V I K T I G T
- Endast en uppgift per blad - skriv INTE på baksidan.
- Uppgifterna skall lösas så utförligt att din tankegång går att följa.
- Det räcker inte med enbart svar till räkneuppgifter.
- Använd inte RÖD penna!!!!
- Det är en klar fördel om du skriver läsligt!

KOM IHÅG. Om du kör fast på en uppgift – lämna denna och gå vidare till nästa. Man behöver inte göra uppgifterna i ordning.

Uppgift 1
Nedan visas en enkel bild av de olika processorblocken där man kan se de olika datavägarna i en processor.
a) Förklara vad som händer i den vänstra inringningen och vad syftet är. Förklara speciellt varför det står ”4”.
b) Förklara vad som händer i den högra inringningen och vad syftet är. Förklara speciellt varför det står ”Shift left 2”.
c) Vilken funktion har den blåa parametern ”PCSrc” i den högra inringningen.



Lösning:
a) Vänster modul: <u>Instruktionshämtning och uppräkning av programräknarregister PC</u>
- PC pekar på nästa instruktion i instruktionsminnet.
- Beroende på vad instruktionen i instruktionsminnet säger, skall PC antingen:

1. **stegas upp med fyra steg (pga 32-bits instr.) till nästa instruktion i instruktions-minnet, vilket görs av ”Add” i denna vänstra modul och skickas därefter via Mux i höger modul så att PC uppdateras.**
2. **ges ett helt nytt värde (pga av relativt hopp i instr.) vilket görs i höger modul.**

b) Höger modul: <u>Branch (relativt hopp i programmet)</u>
- Om PC skall ges ett helt nytt värde (dvs man skall lägga till en relativ hoppadress till nuvarande PC) måste värdet (som är max 16 bitar i instruktionsformatet) konverteras till 32-bit relativ hoppadress. Detta görs i ”Sign-extend” utanför boxen.
- Därefter multipliceras värdet med 4 (pga 32-bits instruktioner) så att den relativa hoppadressen konverteras till rätt relativt hopp i instruktionsminnet. Detta görs i ”Shift left 2”.
- Till sist adderas det relativa hoppet i instruktionsminnet med nuvarande PC så att PC pekar på nästa instruktion, dvs den vi skulle hoppa till.

c) PCSrc=0: PC uppdateras enligt vänster modul, dvs uppstegning till nästa instruktion.
PCSrc=1 : PC uppdateras enligt höger modul, dvs ett relativt hopp till nästa instruktion.

(3 p)

Uppgift 2
Använd bitskift för att utföra följande beräkningar:
a) Dividera 0x10 med 4
b) Dividera 0xF0 med 16
c) Multiplicera 0x10 med 2
(Anmärkning: Talen är på tvåkomplementsform.)
Lösning:
a) $0001\ 0000_2$ skiftas 2 steg åt höger och resultatet blir $0000\ 0100_2 = 0x04$
b) $1111\ 0000_2$ skiftas 4 steg åt höger med ”aritmetiskt högerskift” (dvs kopia av MSB skiftas in för varje högerskift) och resultatet blir $1111\ 1111_2 = 0xFF$
[Kontroll: $1111\ 0000_2 = -\ 0001\ 0000_2 = -16_{10}$ och $1111\ 1111_2 = -0000\ 0001_2 = -1_{10}$
-16_{10} dividerat med 16 blir just -1_{10} dvs beräkningen stämmer.]
c) $0001\ 0000_2$ skiftas 1 steg åt vänster och resultatet blir $0010\ 0000_2 = 0x20$

(3 p)

Uppgift 3
Förklara vad
a) en avbrottstabell (eller avbrottsvektor) är. Förklara också vad som händer efter att tabellen har använts.
b) en avbrottsmask är och hur den används.
Lösning:
a) Varje typ av avbrott pekar på en rad i avbrottstabellen. På respektive plats i avbrottstabellen ligger en hoppadress lagrad som pekar på den avbrottsrutin som skall köras. När avbrottsrutinen har körts, hoppar PC tillbaka till instruktionen den skulle utfört när avbrottet kom.
b) I t.ex. ARM finns ett statusregister CPSR. Två av bitarna i detta register talar om ifall IRQ/FIQ är tillåtet. Med avbrottsmasken $00000000\ 00000000\ 00000000\ 11000000_2$ maskas dessa två bitar ut.

(2 p)

Uppgift 4
Förklara följande begrepp (som alla handlar om ”hazard”) :
a) ”Strukturella hazards”
b) ”Data hazards” (kallas även ”Pipeline data hazards”)
c) ”Data forwarding”
d) ”Instruction reordering”

Lösning:
a) Uppstår vid gemensam instruktions- och datacache. Man kan inte läsa instruktion och läsa/skriva data samtidigt.

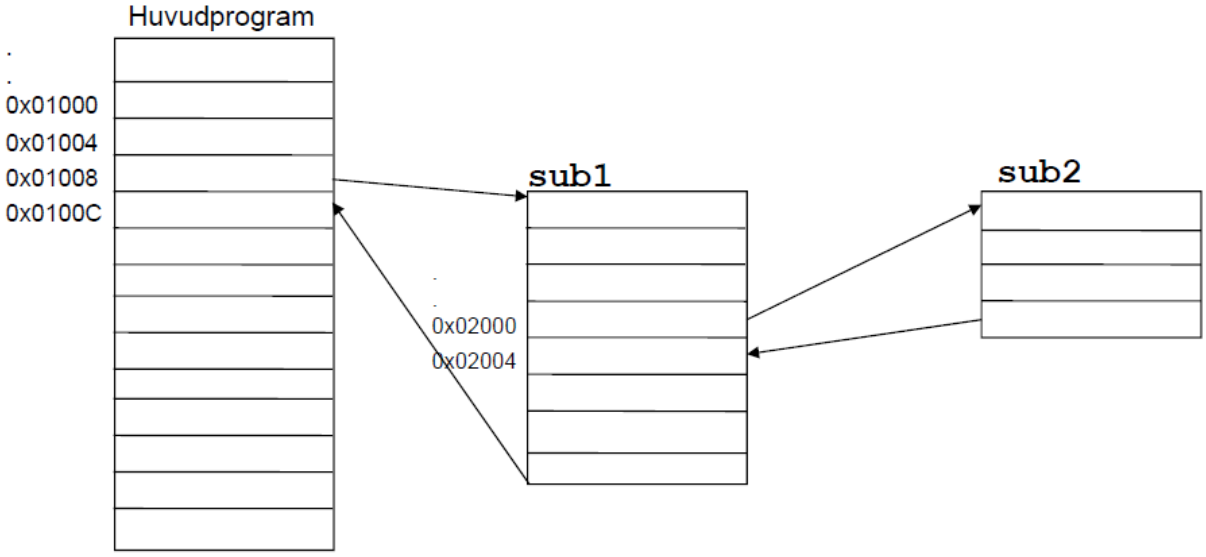
b) Uppstår om en instruktion behöver resultat från en föregående instruktion, som ännu inte är klart (databeroende).

c) Gör ett resultat tillgängligt vid ALU:ns ingångar innan det skrivits till register.

d) Assemblern försöker undvika ”Data hazards” genom att möblera om instruktionerna.

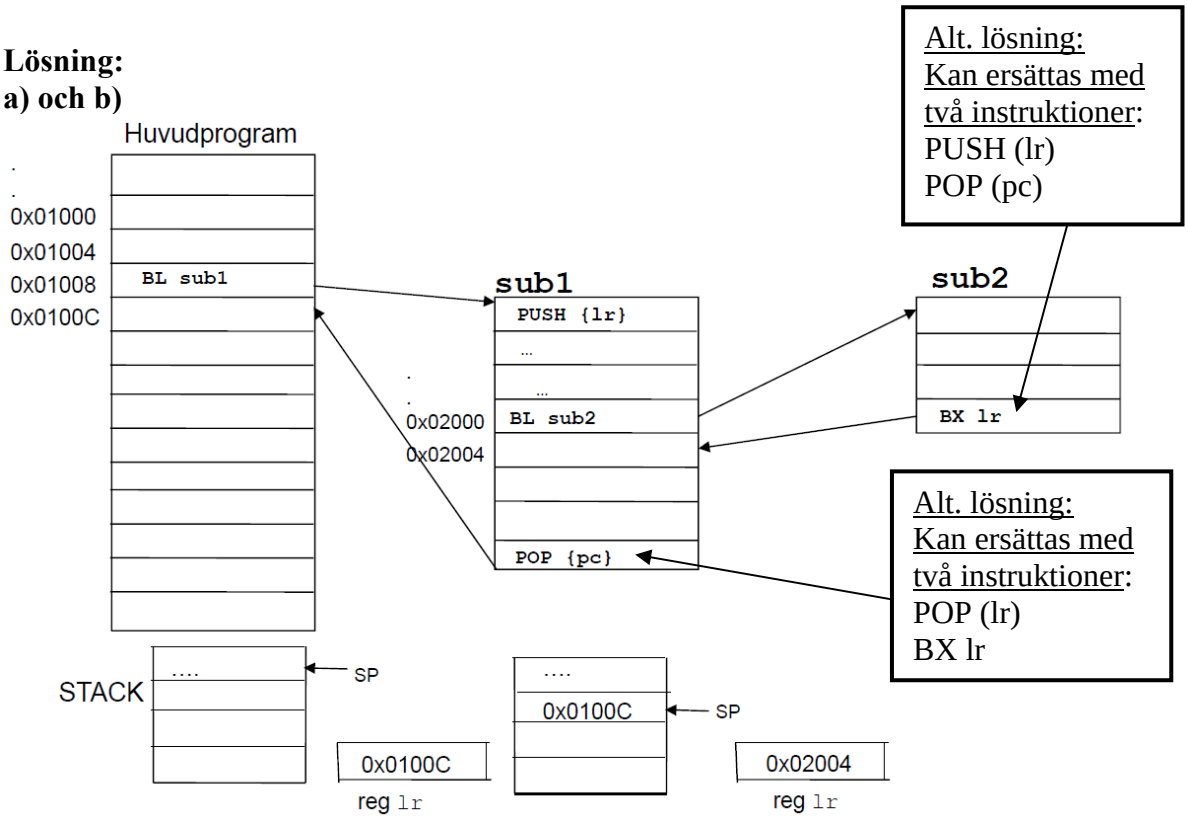
(4 p)

Uppgift 5
I bilden nedan visas ett ARM-assemblerprogram som anropar en subrutin sub1 som i sin tur anropar ytterligare en subrutin sub2.



a) **<u>Rita av bilden</u>** och fyll i de ARM-assembler som behövs för att det skall fungera.
b) Komplettera bilden med boxar för exempelvis register, minne, heap och stack. (Kanske inte allt behövs.) Ange framför varje box vad som lagras i den.
c) Förklara vad du gjort **<u>i detalj</u>** steg för steg!

Lösning:
a) och b)



c)

1. När programpekaren PC når adress 0x01008 i huvudprogrammet körs instruktionen BL sub1, dvs:
   a) Återhoppsadressen (0x0100C) sparas automatiskt undan i länkregister lr.
   b) Programpekaren hoppar till första adressen i sub1

2. För att inte återhoppsadressen i länkregistret lr skall skrivas över vid hopp till sub2 sparas den på stacken med kommandot PUSH {lr}.

3. När programpekaren PC når adress 0x02000 i sub1 körs instruktionen BL sub2, dvs:
   a) Återhoppsadressen (0x02004) sparas automatiskt undan i länkregister lr.
   b) Programpekaren hoppar till första adressen i sub2

4. När programpekaren PC når sista adressen i sub2 körs instruktionen BX lr, dvs:
   a) Programpekaren hoppar till den lagrade adressen i register lr, dvs 0x02004

5. När programpekaren PC når sista adressen i sub1 körs instruktionen POP {PC} , dvs:
   a) Programpekaren hoppar till den lagrade adressen som ligger överst i stacken, dvs 0x0100C.

6. Resten av huvudprogrammet körs.

(3 p)

Uppgift 6

Beskriv skillnaden mellan dynamisk länkning och statisk länkning. Ange fördelar och nackdelar för respektive metod. OBS Det finns flera fördelar/nackdelar!
Förklara speciellt begreppet ”Lazy linkage” och vilka fördelar som detta har.

Lösning:

- **Vid statisk länkning länkas hela biblioteket med moduler in, även om bara ett fåtal moduler skall användas.**

- **Vid dynamisk länkning länkas och laddas modulerna först när programmet ska köras och då länkas och laddas bara de moduler som ska användas.**

- **Fördelar/nackdelar med dynamisk länkning jämfört med statisk länkning**
  + Ingen onödig kod laddas in i minnet (liten EXE.fil).
  + Flera processer kan dela på samma kod.
  + Uppdaterade kodmoduler kan användas utan att kompilera om hela programmet.
  - Det tar tid att ladda in koden första gången.
  - Mindre snabb exekvering.

- **Vid lazy linkage laddas rutinerna in först då de anropas. (Jämfört med dynamisk länkning laddas endast de rutiner in som verkligen används – inte bara de som kanske behöver användas.)**

(1+1+1=3 p)

Uppgift 7

Man skall beräkna uttrycket **`a = b + c + d`** med assemblerinstruktioner till ARMv6 där a, b, c, d refererar till minnesadresser d.v.s. vi skall läsa in tre värden från tre olika minnesadresser och lagra summan på en fjärde minnesadress. Man kan göra detta på två sätt:

Metod 1:
```
ADD r1, b, c
ADD a, r1, d
```
Metod 2:
```
ADD a, b, c
ADD a, a, d
```

Tidsåtgång för skrivning/läsning till/från en minnesadress är 100 ns. Tidsåtgång för skrivning/läsning till/från ett register är 1 ns.

a) Vilken metod är snabbast?
b) Hur många procent snabbare är den snabba metoden?

Lösning:

a) Antal minnesaccesser i metod 1 = 2 + 2 = 4 st => Tid 4 * 100 ns = 400 ns
Antal registeraccesser i metod 1 = 1 + 1 = 2 st => Tid 2 * 1 ns = 2 ns
=> Total tid = 402 ns

Antal minnesaccesser i metod 2 = 3 +3 = 6 st => Tid 6 * 100 ns = 600 ns
Antal registeraccesser i metod 2 = 0 st => Tid 0 ns
=> Total tid = 600 ns

Alltså är metod 1 snabbast

b) Metod 1 är (600-402)/600 = 33 % snabbare

(2+1=3 p)

Uppgift 8

Några småuppgifter med ARMv6 (beskriv med ord vad som händer).

a) Vad gör LDR r4, [r3,#4] och vad kallas denna typ av adressering?
b) Vad gör ADD r4, r4, #3 och vad kallas denna typ av adressering?
c) Vad gör ADD r4, r4, r3 och vad kallas denna typ av adressering?
d) En vektor A består av 15 st 32-bitars integer som är numrerade från A[0] upp till A[14] och ligger lagrad i arbetsminnet. Skriv ett kort assemblerprogram som beräknar A[12] = h + A[8]. Vid start innehåller register r2 talet h och register r3 pekar på minnesadressen till första byten i första vektorelementet A[0]. När programmet har körts skall svaret alltså ligga i vektorelementet A[12], dvs i arbetsminnet. Register r5 skall då peka på A[12].

Lösning:

a) Adressering med basadress och förflyttning (immediate offset): Register r3 pekar på en minnesadress (som är innehållet i r3 plus fyra steg). Innehållet där läggs i register r4.

b) Omedelbar adressering (immediate): Innehållet i register r4 ökas med 3.

c) Registeradressering: Innehållet i register r4 ökas med innehållet i register r3.

```
d) LDR r5, [r3, #32]     /* Lägg innehållet i A[8] i r5. OBS 8*4 = 32 steg */
   ADD r5, r2, r5        /* Beräkna A[8]+h och lägg i r5                   */
   STR r5, [r3, #48]     /* Spara r5 i A[12]. OBS 12*4 = 48 steg           */
   ADD r5, r3, #48       /* Nu pekar r5 på A[12]                           */
```

Några kommentarer till uppgift d):
- Det finns många varianter, t.ex. kan man välja andra register så att r5 kan beräknas innan STR och då använda [r5] som pekare.
- ADD fungerar inte på minnesadresser, dvs ADD r1, r2, [r3,#32] fungerar inte. Gör först load (LDR)
- Notera ordningen på variablerna i STR

(4 p)

Uppgift 9

Vid dynamisk bransch prediktering förekommer begreppet ”bitar” där t.ex. processorn i7 har 18 ”bitar”. Förklara på vilket sätt antalet bitar påverkar prestandan samt varför man använder ”bitar” överhuvudtaget som ett hjälpmedel vid dynamisk bransch prediktering.

Lösning:

<u>En bit för prediktion (gissning) av nästa hopp</u>:
Sparar ifall hoppet gjordes förra gången hoppinstruktionen användes.
Predikterar att samma sak ska hända som förra gången. Om hoppet inte togs förra gången gissar man det inte ska tas nu heller och vice versa.

<u>Två bitar för prediktion (gissning) av nästa hopp</u>:
Sparar ifall hoppet gjordes de två senaste gångerna hoppinstruktionen användes.
En prediktion skall vara fel två gånger för att den skall ändras”

<u>Flera bitar (t.ex. 18 bitar) för prediktion (gissning) av nästa hopp</u>:
Sparar ifall hoppet gjordes under många gånger. 18 bitar innebär att man kan spara mycket information. En prediktion baserad på så stor mängd information blir betydligt säkrare.

Anmärkning: Notera att felstavningen på ”Branch” (”Bransch” är fel)

(2 p)

Uppgift 10
Assemblerprogrammet nedan är skrivet för en ARMv6-processor och skall göra beräkningar på en vektor A som består av 15 st 32-bitars integer. Elementen i vektorn är numrerade från A[0] upp till A[14] och ligger lagrade i arbetsminnet. Vid start innehåller register r2 talet h och register r3 pekar på minnesadressen till första byten i första vektorelementet A[0].

```
LDR r5, [r3, #32]
ADD r5, r2, r5
STR r5, [r3, #48]
ADD r5, r3, #48
```

a) Beskriv vad som händer när programmet körs. Beskrivningen skall vara av typen ”A[3] subtraheras från A[11] och läggs samman med h. etc” (Detta var naturligtvis inte svaret).
Ibland kan man göra slarvfel. För att få delpoäng kan det vara ide att även beskriva vad som händer för varje programrad och inte bara ange slutresultatet.

b) Vad för mening finns det med sista satsen i programmet? Beskriv nyttan med denna sats.

c) I programmet finns det olika typer av adresseringar, nämligen
   1. Registeradressering,
   2. Adressering med basadress och förflyttning (immediate offset)
   3. Omedelbar adressering (immediate):

Rita av och fyll i nedanstående tabell:

Programrad	Adresseringsmetod (ange siffran framför resp. metod)
Rad 1	
Rad 2	
Rad 3	
Rad 4	

Lösning:
a) Rad 1: A[8] läggs i r5
Rad 2: r5 ökas med talet h
Rad 3: r5 sparas i A[12]
Rad 4: r5 pekar på A[12]
Alltså: A[12] = h + A[8] och r5 pekar på A[12] när programmet körts.

b) r5 pekar på A[12] vilket gör att den som använder detta program lätta kan komma åt A[12]

c)

Programrad	**Adresseringsmetod (ange siffran framför resp. metod)**
Rad 1	**2. (Adressering med basadress och förflyttning (immediate offset))**
Rad 2	**1. (Registeradressering)**
Rad 3	**2. (Adressering med basadress och förflyttning (immediate offset))**
Rad 4	**3. (Omedelbar adressering (immediate))**

(1+1+1 = 3 p)

Uppgift 11
I nedanstående exempel använder vi ”Direktmappad cache”.
Vi vill nå minnescellerna i ordningsföljden: 0, 1, 2, 3, 4, 3, 4 resp. 15.
Motsvarande binär adresser blir: 0000, 0001, 0010, 0011, 0100, 0011, 0100 resp. 1111
Anm: Den understrukna biten är ”Index” medan den icke-understrukna biten är ”Tag”
Vi startar med en tom cache.
Kommentar: Mem(0) betecknar innehållet på adressen 0, dvs på den binära minnesadresen 0000).

Fullfölj de resterande sju tabellerna nedan och ange för vart och ett av de åtta fallen om det blir en cachemiss eller inte. (Varje tabell visar innehållet i cacheminnet efter resp. läsning.)

0

Index	Tag	Data
00	00	Mem(0)
01		
10		
11		

1

Index	Tag	Data
00		
01		
10		
11		

2

Index	Tag	Data
00		
01		
10		
11		

3

Index	Tag	Data
00		
01		
10		
11		

4

Index	Tag	Data
00		
01		
10		
11		

3

Index	Tag	Data
00		
01		
10		
11		

4

Index	Tag	Data
00		
01		
10		
11		

15

Index	Tag	Data
00		
01		
10		
11		

Lösning:

0 miss

Index	Tag	Data
00	00	**Mem(0)**
01		
10		
11		

1 miss

00	00	Mem(0)
01	00	**Mem(1)**
10		
11		

2 miss

00	00	Mem(0)
01	00	Mem(1)
10	00	**Mem(2)**
11		

3 miss

00	00	Mem(0)
01	00	Mem(1)
10	00	Mem(2)
11	00	**Mem(3)**

4 miss (01 → 4)

00	00	**Mem(0)**
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

3 hit

00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	**Mem(3)**

4 hit

00	01	**Mem(4)**
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

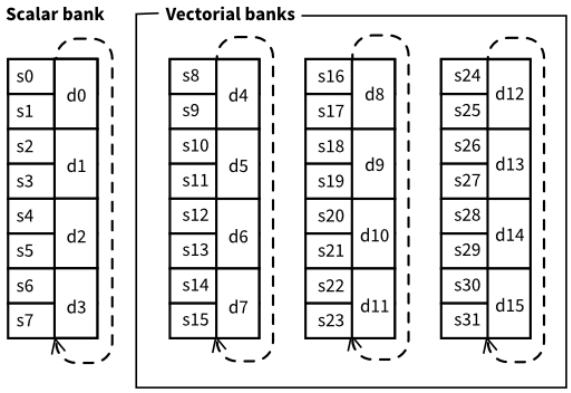
15 miss (11 → 15)

00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	**Mem(3)**

- 8 förfrågningar, 6 missar

(3 p)

Uppgift 12
En ARM-coprocessor har följande register:



a) Om stride=2 och len=4, vilka multiplikationer kommer att utföras då kommandot **VADD.F32 s8, s16, s24** ges? Ange var samtliga värden hämtas och var samtliga resultat lagras.
b) Samma men om stride=1 och len=8

Lösning:
a) **s8 ← s16 + s24**
s10 ← s18 + s26
s12 ← s20 + s28
s14 ← s22 + s30
Kompakt skrivsätt: {s8,s10,s12,s14} ← {s16,s18,s20,s22} + {s24,s26,s28,s30}

b) **s8 ← s16 + s24**
s9 ← s17 + s25
s10 ← s18 + s26
s11 ← s19 + s27
s12 ← s20 + s28
s13 ← s21 + s29
s14 ← s22 + s30
s15 ← s23 + s31
Kompakt skrivsätt: {s8,s9,s10,s11,s12,s13,s14,s15} ← s16,s17,s18,s19,s20,s21,s22,s23}
+ {s24,s25,s26,s27,s28,s29,s30,s31} **(2 p)**

Uppgift 13
Beräkna Y = 0x42e48000 + 0x3f880000 där 0x betecknar hexadecimala tal. Talen är på 32 bitars IEEE-754-form, dvs 1 teckenbit följt av 8 exponentbitar samt följt av 23 signifikandbitar. Exponenten använder excess 127. Svaret skall också vara på IEEE-754-form.

Lösning:

STEG 0: Avkoda resp. tal i tecken, exponent och signifikand
Tal A = 0x42e48000 = 0100 0010 1110 0100 1000 0000 0000 0000$_2$ =
$= (-1)^0 \cdot 2^{(133-127)} \cdot 1.110\ 0100\ 1000\ 0000\ 0000\ 0000_2$
Tal B = 0x3f880000 = 0011 1111 1000 1000 0000 0000 0000 0000$_2$ =
$= (-1)^0 \cdot 2^{(127-127)} \cdot 1.\ 000\ 1000\ 0000\ 0000\ 0000\ 0000_2$

(forts)

(forts.)

STEG 1: Bestäm skillnaden mellan de båda talens exponenter.

Tal A: Exp_A -127 = 133_{10} -127_{10} = 6_{10} (dvs störst tal)

Tal B: Exp_B -127 = 127_{10} -127_{10} = 0_{10} (dvs minst tal)

STEG 2: Skifta minsta talets signifikand (inkl. implicit etta) till höger så båda talens exponent blir lika stora.

Tal A: Ingen förändring av exponent/signifikand

Tal B: Öka exponent med 6: Exp_B -127 = 0_{10} + 6_{10} = 6_{10}

Minska signifikand med 2^6 genom att skifta 6 steg höger.

1. 000 1000 0000 0000 0000 0000_2 -> 0.000001000 1000 0000 0000 0000 0000_2

Alltså:

$(-1)^0 \cdot 2^{(127-127)} \cdot$ 1.000 1000 0000 0000 0000 0000_2 =

$(-1)^0 \cdot 2^{(133-127)} \cdot$ 0.000 0010 0010 0000 0000 0000 00 0000_2 =

$(-1)^0 \cdot 2^{(133-127)} \cdot$ 0.000 0010 0010 0000 0000 0000_2 där Guard=0, Round=0 och 0000 kastas (Sticky=0).

STEG 3: Addera/subtrahera signifikanderna.

Tal A: $(-1)^0 \cdot 2^{(133-127)} \cdot$ 1.110 0100 1000 0000 0000 0000_2

Tal B: $(-1)^0 \cdot 2^{(133-127)} \cdot$ 0.000 0010 0010 0000 0000 0000_2

1.110 0100 1000 0000 0000 0000_2

+ 0.000 0010 0010 0000 0000 0000_2 (Guard=0, Round=0, Sticky=0)

1.110 0110 1010 0000 0000 0000_2

Kontroll (omständlig och rekommenderas inte):
Tal A: 2^6 * 1.11001001 = 1110010.01 =
= 64 + 32 + 16 + 2 + 1/4 = 114,25
Tal B: 2^0 * 1.0001 = 1 + 1/16 = 1,0625
Summa = 114.25 + 1,0625 = 115,3125

Svar: 2^6 * 1.1100110101 = 1110011.0101 =
= 64 + 32 + 16 + 2 + 1 + 1/4 + 1/16 =
= 115,3125 (samma som ovan)

STEG 4: Normalisera summan (skifta, ändra exponent)

Redan normaliserad

STEG 5: Avrunda signifikanden om det behövs

Guard=0, Round=0, Sticky=0 ger inget bidrag

STEG 6: Om avrundningen innebär att det blir onormaliserat så gå tillbaka till steg 4.

(Är normaliserat => ingen åtgärd)

STEG 7: Sammanställ:

$(-1)^0 \cdot 2^{(133-127)} \cdot$ 0000 0000 0000_2 = 0 1000 0101 110 0110 1010 0000 0000 0000_2

STEG 8: Snygga till:

Svar: 0100 0010 1110 0110 1010 0000 0000 0000_2 = 0x42e6a000

Kommentarer:

Att gå via det decimala talsystemet och addera 1/2 + 1/4 + 1/32 + 1/256 är onödigt jobbigt samt bäddar för slarvfel. Dessutom arbetar en dator inte så. **(3 p)**

Uppgift	1	2	3	4	5	6	7	8	9	10	11	12	13	Summa
Poäng	3	3	2	4	3	3	3	4	2	3	3	2	3	38

Betyg	E	D	C	B	A
Gräns	19	23	26	29	32

Lycka till
Mikael