Lösningsförslag till Tenta i Mikrodator 050113

1. Vilka register finns det i processorn och vad används dessa till?

D0 till D7: Dataregister som används för beräkningar
A0 till A6: Adressregister som används till att lagra pekare till minnet
A7: pekar på stacken och är dubblerad till USP och SSP som används beroende på vilken mod processorn befinner sig i, *user mode* eller *supervisor mode*.
PC: Programräknaren pekar på den adress i minnet som nästa instruktion skall hämtas ifrån.
SR: statusregistret styr några av processorns funktioner, avbrottsmask, trace och användarmod. Status från aritmetiska operationer indikeras i ett antal statusbitar.

2. Nämn och beskriv kortfattat två sätt att adressera data i ett cacheminne.

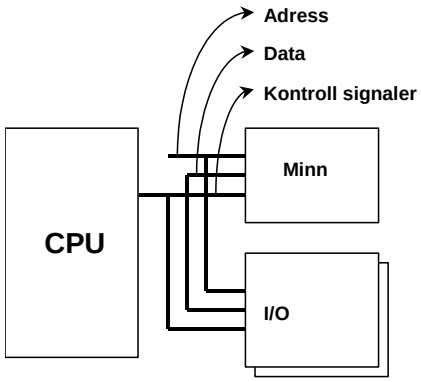
Direktmappad cache: Cacheminnet är indelat i N stycken lika stora linjer. Primärminnet är indelat i K stycken block. Ett sådant block består i sin tur av N stycken linjer. En linje n i primärminnet kan endast lagras i motsvarande linje n i cacheminnet. Därav direktmappat.

Associativ cache: En fullt associativ cache är precis som för den direktmappade indelad i N stycken linjer. En linje i primärminnet kan för en associativ adressering lagras i godtycklig linje i cacheminnet. Denna flexibla adressering gör också att sökningen i det associativa cacheminnet blir mer komplext och dyrare att implementera.

3. Vilken uppgift har UDS* och LDS* ?

UDS* och LDS* är övre respektive nedre datastrobe. LDS indikerar att den minst signifikanta byten skall adresseras. UDS indikerar att den mest signifikanta byten skall adresseras.

4. Vad står DMA för samt beskriv hur DMA fungerar?



DMA står för Direct Memory Acess och är en metod för processorn att synkronisera mot I/O-enheter. När en I/O-enhet behöver överföra data till minnet så begär den att få ta över kontrollen över systembussen, dvs bli bussmästare. Processorn lämnar då över kontrollen till I/O som nu kan direkt överföra data till minnet utan att gå via processorn. Efter avslutad dataöverföring lämnas kontrollen över systembussen tillbaka till processorn.

5. Antag att register D1 innehåller $00000099. Ange D1, C-flaggans samt Z- flaggans värde efter att instruktionerna nedan har exekverats. Observera att a till f är enskilda instruktioner och inte ett program.

a) **EORI.B #$66,D1**

$99 = 1001 1001
$66 = 0110 0110
XOR
$FF 1111 1111

→ C=0 , Z = 0, D1=$000000FF

b) **ADDI.B #$66,D1**

$99 = 1001 1001
$66 = 0110 0110
ADD
$FF 0 1111 1111

→ C=0 , Z = 0, D1=$000000FF

c) **ADDI.B #$67,D1**

$99 = 1001 1001
$67 = 0110 0111
ADD
$100 = 1 0000 0000

→ C=1 , Z = 1, D1=$00000000

d) **SUBI.B #$66,D1**

$99 = 1001 1001
$66 = 0110 0110
SUB
$033 = 0 0011 0011

→ C=0 , Z = 0, D1=$00000033

e) **ASR.B #2,D1**

$99 = 1001 1001
ASR 2
$E6 = 1110 0110

→ C=0 , Z = 0, D1=$000000E6

f) **ROL.B #4,D1**

$99 = 1001 1001
ROL 4
$99 = 1001 1001

→ C=1 , Z = 0, D1=$00000099

6. Funktionsanrop i assembler. En funktion i C är deklarerade som följande:

```
void printChar(int a, int b);
```

Parametrarna a och b ska skickas till funktionen via stacken.

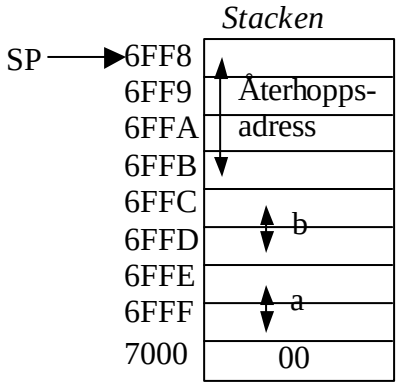
a) **Rita stackens innehåll efter hoppet till subrutinen printChar, precis innan första instruktionen i subrutinen exekverats. Stackpekaren pekar på adress $7000 innan anropet. Anropet till funktionen i assembler görs enligt följande:**

Funktionsanrop i C

```
PrintChar(a,b);
```

Funktionsanrop i assembler

```
MOVE.W a,-(A7)
MOVE.W b,-(A7)
BSR printChar
ADDA.L #4,A7
```



b) Vilken funktion har instruktionen ADDA.L #4,A7 ?

Instruktionen ”tar bort” parametrarna a och b från stacken. (Med ”tar bort” i detta samanhang menas att stackpekaren justeras +4.)

7. Adresseringsmetoder: Vad kommer D2 och A3 att innehålla efter att instruktionen har exekverats. Innehållet i minnet är specificerat av tabellen nedan. Instruktionerna är oberoende av varandra (inget program). D2 har värdet $00000000 och A3 $00004004 innan varje instruktion exekveras.

a) MOVE.L $4004, D2
b) MOVE.W 4(A3), D2
c) MOVE.L -(A3), D2
d) MOVE.B -(A3), D2
e) MOVE.W -2(A3), D2
f) MOVE.W #$4004, D2
g) MOVE.B (A3)+,D2
h) **MOVE.L (A3)+,D2**

Adress (Hexadecimal)	Innehåll (Hexadecimal)
4000	77
4001	21
4002	C3
4003	FD
4004	00
4005	14
4006	99
4007	74
4008	A6
4009	AA

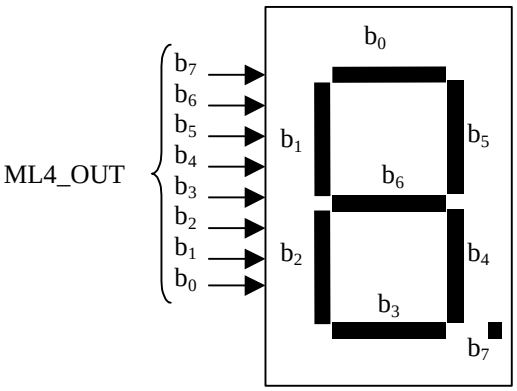
Deluppg.	**Instruktion**	**D2**	**A3**
A	MOVE.L $4004, D2	$00149974	$00004004
B	MOVE.W 4(A3), D2	$0000A6AA	$00004004
C	MOVE.L -(A3), D2	$7721C3FD	$00004000
D	MOVE.B -(A3), D2	$000000FD	$00004003
E	MOVE.W -2(A3), D2	$0000C3FD	$00004004
F	MOVE.W #$4004, D2	$00004004	$00004004
G	MOVE.B (A3)+,D2	$00000000	$00004005
H	MOVE.L (A3)+,D2	$00149974	$00004008

8. Assemblerprogrammering: Skriv en subrutin i 68000-assembler som skriver ut ett tal
(0-15) till en display. På displayen presenteras talet som en hexadecimal siffra (0-F). Displayen är av samma typ som på labbkorten, dvs. en 7-segment display.

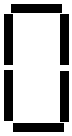
Displayen är ansluten till ML4_OUT (porten är definierad). Varje bit i ML4_OUT är ansluten till en lysdiod i 7-segment displayen. Kopplingen mellan bitarna i ML4_OUT och lysdioderna är definierad i figuren nedan (T ex. b_0 är kopplad till den översta vågräta dioden). b_7 är kopplad till punkten på 7-segmentdiesplayen, så den sätter ni alltid till noll.

In: ML4_OUT = Talet som ska skrivas ut till 7-segmentdisplayen (tal mellan 0-16)
Ut: Inget ska returneras från subrutinen.

Exempel: Nedre byten på D0 = %00110000 ska ge en etta på displayen och ML4_OUT = %00111111 ger en nolla.



Ex. för en nolla
ML4_OUT = %00111111

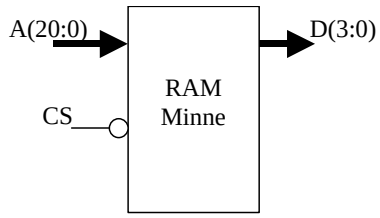


b_7=0
b_6=0
b_5=1
b_4=1
b_3=1
b_2=1
b_1=1
b_0=1

```
**********************************************
* Subroutine:  writeToSevenSegment
*
* Input parameters: D0.B number to be written
* Output: none
**********************************************

writeToSevenSegment:
                MOVEM.L D0/A0,-(SP)         *A0, D0 will be used and copied to the stack
                BRA startOfSub

segmentLUT:
**SEGMENTS**76543210*****
OUT_0 DC.B %00111111
OUT_1 DC.B %00110000
OUT_2 DC.B %01101101
OUT_3 DC.B %11111001
OUT_4 DC.B %01110010
OUT_5 DC.B %01011011
OUT_6 DC.B %01011111
OUT_7 DC.B %00110001
OUT_8 DC.B %01111111
OUT_9 DC.B %01111011
OUT_A DC.B %01110111
OUT_B DC.B %01011110
OUT_C DC.B %00001111
OUT_D DC.B %01111100
OUT_E DC.B %01001111
OUT_F DC.B %01000111

                ALIGN
startOfSub:
                ANDI.L #$F,D0                       * Only four bit parameter
                LEA             segmentLUT,A0       * A0 points at start of table
                ADDA.L D0,A0                        * Add parameter to start of table
                MOVE.B (A0),ML4_OUT                 * Copy the 7-seg code that A0 points at

                MOVEM.L (SP)+,D0/A0                 * Retrieve old value of A0,D0
                RTS
```

9. Sätt samman en minnesbank som är konstruerad av flera mindre minnesobjekt (se komponenten i figuren). Signaler till minnesbanken ska vara A(adress), D(data) samt CS*. Minneskapslarna som används för att konstruera minnesbanken har samma signaler som den färdiga minnesbanken har. Där CS* signalen är aktivt låg. Du ska indikera vilka adress/data-signaler som ska kopplas till de olika minneskapslarna.

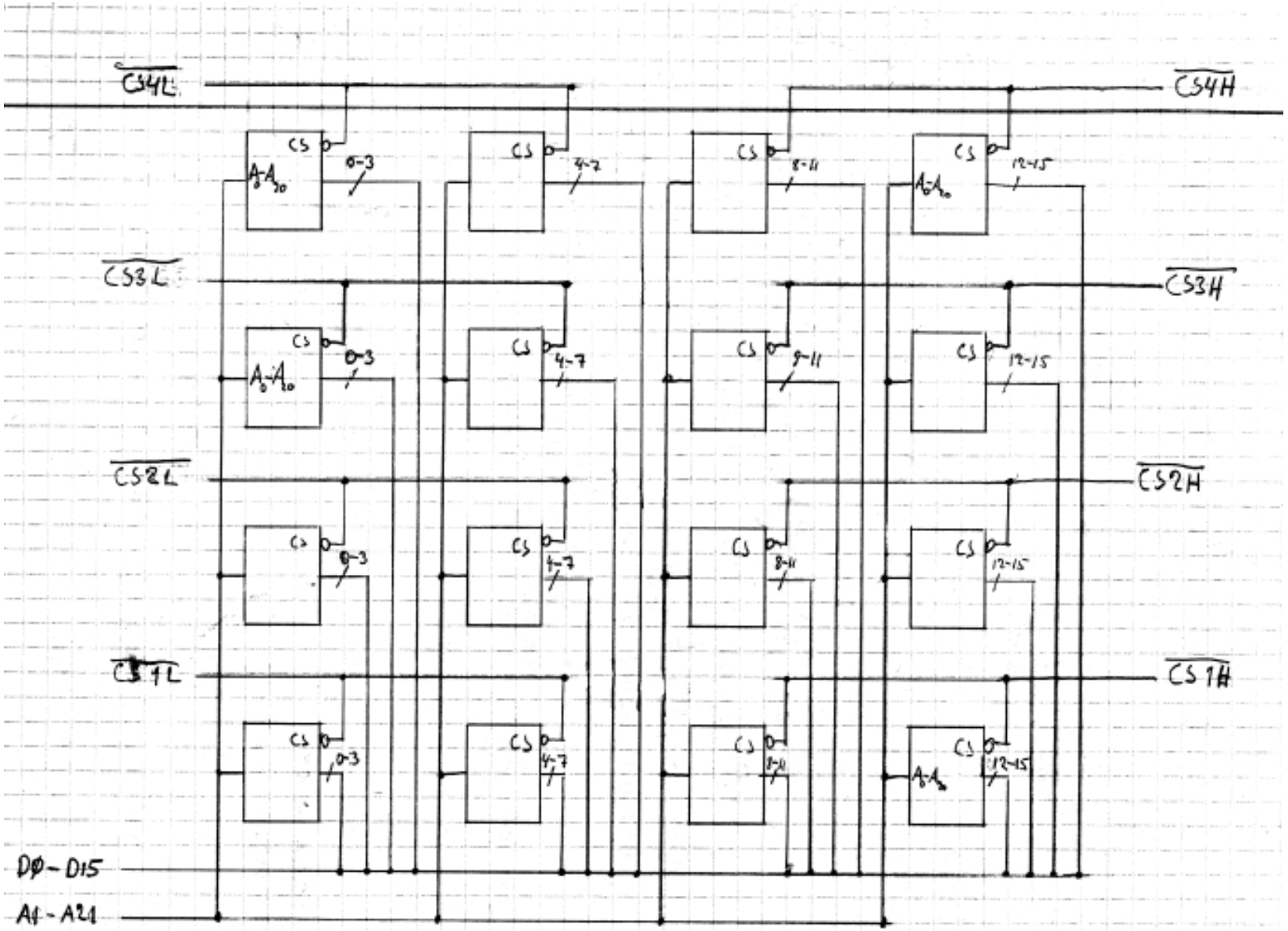
Minnesbanken ska vara 16Mbytes stor och organiserad med en 16 bitars databuss. Minnesbanken sätts samman med hjälp av minneskomponenter som har 4-bitars databuss och 21 bitars adressbuss.



$$m_{comp} = \frac{2^{24}}{2^{21} \cdot \frac{4}{8}} = \frac{2^3}{1/2} = 2^4 = 16 \text{ kapslar}$$

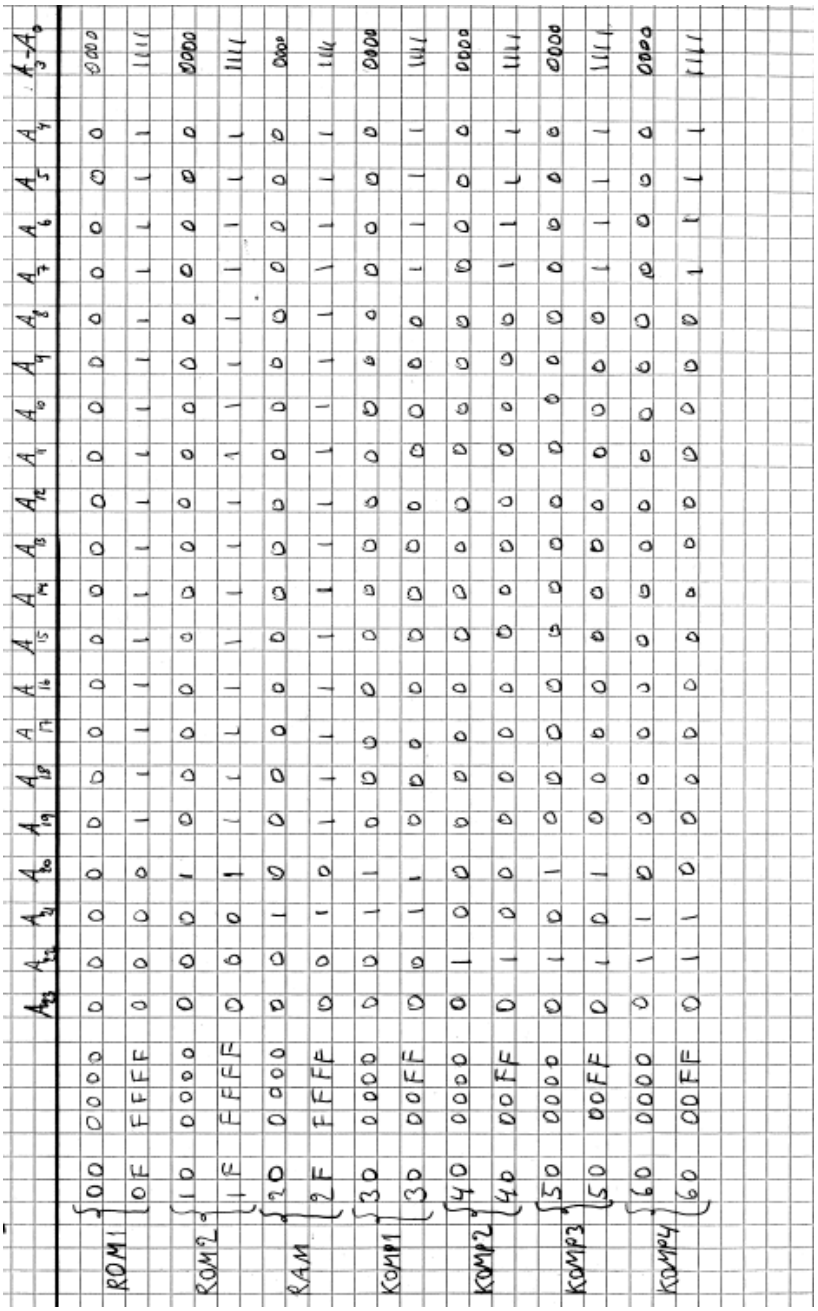
antal kapslar per minnesbank $= m_{kpb} = \frac{16}{4} = 4$

antal minnesbanker $m_{bank} = \frac{m_{comp}}{m_{kpb}} = \frac{16}{4} = 4$

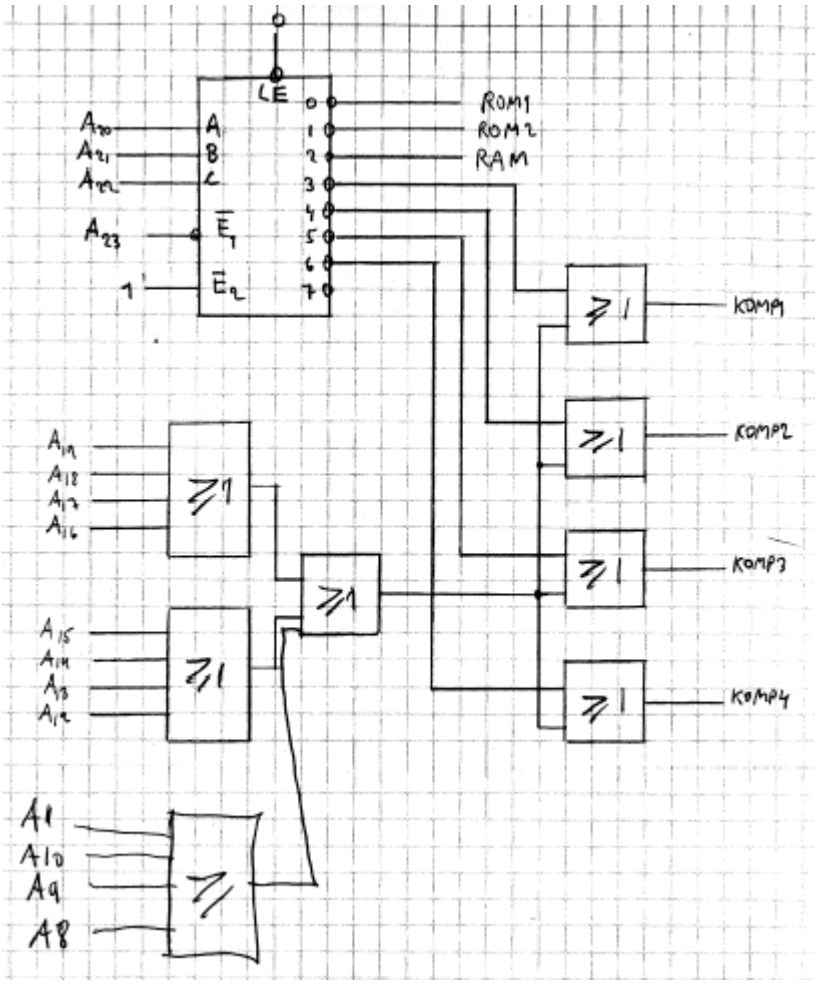

CS4L
CS4H
CS3L
CS3H
CS2L
CS2H
CS1L
CS1H
CS
0-3
4-7
8-11
12-15
D0 - D15
A1 - A21

10. A) Konstruera en fullständig adressavkodare (fullständig adressavkodning) som realiserar adresskartan definierad i uppgiften. Implementera med hjälp av logiska grindar och avkodare (specificerad i Appendix)

Komponent	**Adressområde**
ROM1	00 0000 - 0F FFFF
ROM2	10 0000 – 1F FFFF
RAM	20 0000 - 2F FFFF
KOMP_1	30 0000 - 30 00FF
KOMP_2	40 0000 - 40 00FF
KOMP_3	50 0000 - 50 00FF
KOMP_4	60 0000 - 60 00FF



	A_{23}	A_{22}	A_{21}	A_{20}	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_7	A_6	A_5	A_4	$A_3 - A_0$
ROM1 00 0000 - 0F FFFF	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X
ROM2 10 0000 - 1F FFFF	0	0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X
RAM 20 0000 - 2F FFFF	0	0	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X
KOMP1 30 0000 - 30 00FF	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	X
KOMP2 40 0000 - 40 00FF	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
KOMP3 50 0000 - 50 00FF	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	X
KOMP4 60 0000 - 60 00FF	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	X



B) Hur många bytes är definierat för varje komponent i adresskartan?

Komponent	Adressbitar som omfattas av komponent	Storlek
ROM1	A_{19}- A_0	2^{20} = 1 Mbyte
ROM2	A_{19}- A_0	2^{20} = 1 Mbyte
RAM	A_{19}- A_0	2^{20}= 1 Mbyte
KOMP_1	A_7- A_0	2^8 byte = 256 byte
KOMP_2	A_7- A_0	2^8 byte = 256 byte
KOMP_3	A_7- A_0	2^8 byte = 256 byte
KOMP_4	A_7- A_0	2^8 byte = 256 byte

11. A) Implementera följande IF-sats i 68000-assembler. Alla variabler är av 16-bit storlek och kan ersättas med register.

```
if (tal1 > tal2)
   max = tal1;
else
   max = tal2;
```

```
            move.w      tal1,d0
            cmp.w       tal2,d0
            bls         condElse  // Jump if tal1 <= tal2
            move.w      tal1,max
            bra         condDone
condElse:
            move.w      tal2,max
condDone:
```

B) Implementera följande loop

```
while(a<5){
  kalle();
  a++;
}
```

i assemblerkod. Där `kalle` är en subrutin och `a` är ett 16-bits tal i minnet.

```
Loop:
            CMPI.W      #5,a
            BCC         LoopExit    // Jump if a >= 5
            BSR         kalle
            ADDQ.W      #1,a
            BRA         Loop
LoopExit:
```