Datorsystem
Övningshäfte

Senast uppdaterad: 30 oktober 2021
Version 2.0

Innehåll

1. Introduktion

Det här häftet används på kursen Datorsystem vid Institutionen för data- och systemvetenskap vid Stockholms universitet som komplement till kurslitteraturen. Häftet kommer att innehåller övningar till de flesta av kursens delar och förklarande texter till vissa av dem.

Häftet kommer att uppdateras kontinuerligt under kursens gång och kompletteras med fler kapitel. Det är mycket möjligt, för att inte säga troligt, att det finns stavfel, räknefel eller andra fel i häftet. Den som hittar ett fel ombeds skicka ett mail med information om felet till p_fgk @dsv.su.se så att felet kan korrigeras.

2. Datorns grunder

2.1 Övningsuppgifter

Uppgift 2.1.1
Inom datorsystem används ofta enheterna bitar och bytes.

a. Vad är en bit?

b. Hur många bitar är en byte?

Uppgift 2.1.2
När man använder enheter som bitar och bytes är det vanligt att man använder prefix för att enklare ange storleken på enheterna. Till skillnad från de vanliga SI-prefixen ger inte varje prefix en multipel av 1000. Hur många bytes anges nedan?[1]

a. Kilo: 1 kB

b. Mega: 1 MB

c. Giga: 1 GB

d. Peta: 1 TB

e. 3 kB

f. 4 MB

g. 2,5 GB

h. 5 MB

i. 0,5 GB

Uppgift 2.1.3
Om du ska köpa en dator i delar, vilka delar behöver du vanligtvis sett skaffa för att ha en fungerande dator?

Uppgift 2.1.4
När byggdes den första transistorn?

[1] Mer information finns på `http://sv.wikipedia.org/wiki/Bit`

3. Talsystem

3.1 Bitar och bytes

När man pratar om datorer kommer man på ett eller annat sätt att komma in på bitar och bytes. När man köper en ny dator kan man läsa att den har en 64-bitars processor, två gigabyte RAM-minne och kanske en hårddisk på en terabyte. När man ska beställa bredband kan man se att ADSL har hastigheter på upp till 24 megabit per sekund och att fiber kan ge hela en gigabit per sekund.

Bitar och bytes är alltså ganska vanliga, men vad är en bit och vad är en byte? Svaret är att en bit är den minsta mängden information en dator hanterar. En byte består av åtta bitar, och kallas ibland också för en oktett. Om man tittar på exemplen ovan för när man använder bitar och bytes ser vi att man ibland använder bitar och ibland bytes. Generellt använder man bitar när man pratar om processorer och nätverkshastighet medan man använder bytes när man pratar om minnesstorlekar.

Vad är det då för information man lagrar i en bit? I en bit brukar det finnas antingen en etta eller nolla, och dessa ettor och nollor representeras på olika sätt i olika enheter. I exempelvis en hårddisk, som använder sig av magnetism, representeras ettor och nollor av varsin magnetisk riktning.

3.2 Talsystem

Att en dator arbetar med bara ettor och nollor är av praktiska skäl. Arbetar man med elektricitet är det mycket lättare att veta om något har en laddning eller inte jämfört med att särskilja mellan olika storlekar på laddningen. Det betyder dock att vi behöver anpassa oss efter datorerna, vilket vi gjort genom att använda det binära talsystemet. Med hjälp av det kan en dator förstå och hantera de instruktioner som ges till den, eftersom det är anpassat till datorns sätt att lagra information.

Det decimala talsystemet

Vanligtvis brukar vi i Sverige och större delen av världen använda det decimala talsystemet. När man ser talet 4236 vet man att det är likvärdigt med 4*1000 + 2*100 + 3*10 + 6*1. Basen för det decimala talsystemet är 10, vilket ger att varje steg till vänster ökar en positions värde tiofaldigt. På samma sätt minskar värdet i varje position åt höger med faktor tio.

Den här typen av talsystem kallas för positionssystem. Det innebär att platsen för siffran är viktig; vi kan till exempel inte skriva 2364 och mena att det ska vara samma sak som 4236. För att göra det mer tydligt kan vi i tabell 1 se vad varje plats betyder, för ett femsiffrigt tal).

För att skriva decimala tal kan vi använda oss av siffrorna 0 till 9. Så fort vi behöver större tal än så måste vi använda oss av nästa position. När vi adderar 1 till 9 kan vi inte längre använda oss av bara en position eftersom de tillåtna siffrorna är 0-9. Behöver vi representera ett större tal måste vi använda oss av nästa position till vänster. Istället för $9 * 10^0$ får vi allts $1 * 10^1 + 0 * 10^0$ På samma sätt blir det om vi adderar 1 till 99 då vi får 100, vilket ger $1 * 10^2 + 0 * 10^1 + 0 * 10^0$ istället för $9 * 10^1 + 9 * 10^0$.

Det binära talsystemet

Det binära talsystemet fungerar precis likadant som det decimala, med skillnaden att basen är två istället för tio. När man i det decimala talsystemet använder sig av siffrorna 0 till 9 använder man sig i det binära

tiotusental	tusental	hundratal	tiotal	ental
10000	1000	100	10	1
10^4	10^3	10^2	10^1	10^0

Tabell 3.1: Värden för olika positioner i det decimala talsystemet

talsystemet av 0 till 1. I tabell 2 kan vi se värdet för olika positioner i ett femsiffrigt tal i det binära talsystemet.

16	8	4	2	1
2^4	2^3	2^2	2^1	2^0

Tabell 3.2: Värden för olika positioner i det binära talsystemet

Vi kan ta ett exempel för att visa hur det binära talsystemet fungerar. Om vi har det binära talet 11010, vilket är då motsvarande decimala tal? Ur tabellen ovan ser vi värdena på de olika positionerna, så det vi behöver göra är att lägga ihop dem och multiplicera med siffran på varje position: $1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 26$. Precis som i det decimala talsystemet ökar värdet på en position för varje steg till vänster, dock är ökningen tvåfaldig (med andra ord dubbleras värdet) eftersom vi har talbasen två.

Det hexadecimala talsystemet

Det finns ett tredje vanligt talsystem som används, nämligen det hexadecimala systemet, med talbasen 16. Precis som de två tidigare talsystemen är systemet positionsbaserat, och varje steg åt vänster ökas en positions värde. Varje steg här är ger en sextonfaldig ökning, något vi kan se i tabell 3.

65536	4096	256	16	1
16^4	16^3	16^2	16^1	16^0

Tabell 3.3: Värden för olika positioner i det hexadecimala talsystemet

Eftersom det hexadecimala talsystemet har talbasen 16 räcker inte våra siffror från det decimala talsystemet till, och vi kan ju inte börja skriva 10 eftersom vi då använder två positioner och därmed ökar värdet. Av den anledningen används tecknen 1-9 för de första nio siffrorna, och sedan används a-f för att markera värdena 10-15. En snabb översättningstabell kan ses i tabell 4.

10	11	12	13	14	15
a	b	c	d	e	f

Tabell 3.4: Siffrorna 10-15 i det hexadecimala talsystemet

Det hexadecimala talet 4FC ger då det decimala talet 1276. Även här ska vi se hur det ser genom att ta det i mindre steg. Vi tar hjälp av tabellen ovan för värdena för varje position, sen multiplicerar vi med siffrorna i vårt tal: $4 * 16^2 + F * 16^1 + C * 16^0 = 4 * 256 + 15 * 16 + 12 * 1 = 1276$.

Övriga talbaser

Egentligen kan man säga att vilket tal som helst kan vara talbas till ett talsystem, men i praktiken är det bara ett fåtal talsystem som används. För vår del bryr vi oss bara om de tre vi tagit upp här, men det händer att det oktala talsystemet används (talbas 8), och om Karl XII hade fått bestämma hade vi i Sverige använt ett system med talbas 64. Som tur var fick han aldrig igenom den idén och vi delar därför talbas med i stort sett hela världen

Talbasnotation

Som vi sett kan det ibland vara svårt att se skillnad på tal från de olika talbaserna. 1010 i det binära systemet blir 10 i det decimala systemet, men 10 i det binära systemet är 2 i det decimala. Hur kan man då se vilket talsystem det är som används? Det man gör är att man noterar de olika talsystemen genom att skriva talet och sedan nedsänkt skriva vilken talbas det är. Inom programmering kan man ibland också notera ett binärt tal genom att först skriva 0b, så att man till exempel kan skriva 0b10011. På

samma sätt skriver man hexadecimala tal genom att först skriva 0x så att man exempelvis får 0x45AC3. För decimala tal finns inget prefix eftersom det är det talsystemet som används som standard. I tabell 5 kan vi se exempel på hur man anger talbas för olika talsystem.

Binärt:	1011011_2	0b1011011
Decimalt:	1324_{10}	1324
Hexadecimalt:	$12F4C_{16}$	0x12FC4

Tabell 3.5: Notationer för olika talsystem

Binära prefix vs. SI-prefix

När man använder enheter som till exempel watt, Joule och gram brukar man också använda sig av olika SI-prefix (från franskans ”Système International d’Unités”, det vill säga ”Internationellt system för måttenheter”).

Prefixen används för att markera både stora och små tal, till exempel kan man säga att en person väger 67 kilogram (kg) och menar då att personen väger 67000 gram. Ett annat exempel är att kärnkraftverket Forsmark förra året producerade 21,9 terawatt-timmar (TWh), alltså 21 900 000 000 000 watt-timmar (eller, för att visa hur stort det är: 21,9 biljoner watt-timmar). Om man istället går åt andra hållet så byggs många av dagens processorer med 45 nanometers-teknik, det vill säga 0,000 000 045 meter.

Varje SI-prefix ökar eller minskar värdet tiofaldigt. Har man ett hektogram skinka har man 100 gram skinka. Om man ökar ett steg får man ett kilogram skinka och har därmed 1000 gram skinka.

För binära prefix, alltså de som används i samband med bitar och bytes, ökar man istället värdet för varje steg med en multipel av 1024. Anledningen till att det är just 1024 är för att det är en multipel av 2, och 1024 kan också skrivas som 2^{10}. För att man ska kunna se skillnad på ett binärt prefix och ett SI-prefix finns speciella binära prefix, men oftast används SI-prefixen även i samband med bitar och bytes. Den som är intresserad kan läsa mer på `http://sv.wikipedia.org/wiki/Bit`

Binära bråk

Det binära talsystemet kan även hantera bråktal. Precis som i det decimala talsystemet görs det genom att för varje position till höger om decimaltecknet minskas värdet med hälften (eller delas med tio i det decimala talsystemet.). Vi ser i tabell 6 hur det fungerar.

3.3 Omvandling mellan talsystem

Hittills har vi bara omvandlat från det binära och det hexadecimala talsystemen till det decimala talsystemet. Man kan naturligtvis också omvandla åt andra hållet och mellan de andra talsystemen.

Decimala tal till binära tal

Om man har ett decimalt tal och vill omvandla det till ett binärt tal kan man använda sig av en algoritm. Algoritmen har följande steg:

1. Skriv upp värdet på varje position tills du kommer till en position vars värde är större än det decimala talet du vill översätta.

16	8	4	2	1	1/2	1/4	1/8	1/16
2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
10000	1000	100	10	1	1/10	1/100	1/1000	1/10000
10^4	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}

Tabell 3.6: Bråk i olika talsystem

2. Sätt en etta i positionen för det näst största talet, samt skriv upp värdet på positionen.

3. Gå ett steg till höger och addera den positionens värde till summan du nyss skrev upp. Om den nya summan är mindre än eller lika med det decimala talet du ska omvandla sätter du en etta i den nya positionen. Om den nya summan istället är större sätter du en nolla och behåller den gamla summan.

4. Repetera steget ovan tills din summa matchar det decimala talet du vill omvandla. Om du fortfarande har positioner kvar till höger sätt dem till noll eftersom de inte ska ha något värde.

Vi kan illustrera algoritmen ovan med ett exempel. Vi kan ta det decimala talet 57 och omvandla det till ett binärt tal. Om vi gör stegen enligt algoritmen ovan får vi följande resultat:

1. Vi skriver upp värdet på positionerna (vi slutar med 64 då det är större 57.):
   64 32 16 8 4 2 1

2. Vi sätter en etta i positionen för det näst största talet, samt skriver upp värdet 32.
   ```
      1
   64 32 16 8 4 2 1
   ```

3. Vi går ett steg till höger och adderar 16 till 32. Vår summa blir 48, och eftersom det är mindre än 57 sätter vi en etta i den nya positionen.
   ```
      1  1
   64 32 16 8 4 2 1
   ```

4. Vi repeterar steget ovan och adderar 8 till 48 och får 56. 56 är mindre än 57 och vi sätter därmed en etta i vår nya position.
   ```
      1  1  1
   64 32 16 8 4 2 1
   ```

5. Här kan vi egentligen se att den enda position som kan adderas till 56 för att få 57 är den som är längst till höger, men vi gör nästa steg för att illustrera hur man får en nolla på en position. Vi adderar 4 till 56 och får 60, som alltså är större än 57. Därmed sätter vi en nolla på denna position.
   ```
      1  1  1  0
   64 32 16 8 4 2 1
   ```

6. Vi fortsätter och prova att addera 2 till 56 och får 58. Eftersom även det är större än 57 får vi en nolla i denna position.
   ```
      1  1  1  0  0
   64 32 16 8 4 2 1
   ```

7. I vårt sista steg provar vi att addera 1 till 56 och får 57. Eftersom det är lika med det decimala talet vi vill omvandla sätter vi en etta här.
   ```
      1  1  1  0  0  1
   64 32 16 8 4 2 1
   ```

Därmed har vi omvandlat det decimala talet 57 till det binära talsystemet och fick resultatet 111001.

Vi kan ta ett något större exempel genom att omvandla det decimala talet 112 till det binära talsystemet. Vi tar varje steg för sig:

1. Först skriver vi upp värdet på alla positioner och slutar när vi hittar ett tal större än 112.
   128 64 32 16 8 4 2 1

2. Vi börjar med att sätta en etta i den nästa största positionen samt skriva upp värdet där: 64.
   ```
     1
   128 64 32 16 8 4 2 1
   ```

3. Vi adderar 32 till 64 och får 96. 96 är mindre än 112 så vi sätter en etta här.
   ```
     1  1
   128 64 32 16 8 4 2 1
   ```

4. Nu adderar vi 16 till 96 och får 112, vilket är det tal vi skulle omvandla. Vi sätter en etta i vår nuvarande position, och eftersom vi har fått summan 112 kan vi sätta nollor i resterande positioner.

	1	1	1	0	0	0	0
128	64	32	16	8	4	2	1

Resultatet av att köra vår algoritm för att omvandla decimala tal till det binära talsystemet är talet $111\ 0000_2$.

Binära tal till hexadecimala tal

Att gå från binära till till hexadecimala tal, och vice versa, är enkelt eftersom varje hexadecimal siffra kan ha värdena 0-15, vilket är precis samma värden man kan representera med fyra bitar. Det man gör är då att man tar sitt binära tal, grupperar det så att man har fyra bitar (siffror) i varje grupp och sen översätter man varje grupp för sig. Som exempel kan vi kan översätta det binära talet 1101001100011110 till det hexadecimala talsystemet.

1. Först delar vi upp talet i grupper om fyra: 1101 0011 0001 1110.

2. Sedan översätter vi varje del för sig:

Bin	Dec	Hex
1101	13	D
0011	3	3
0001	1	1
1110	14	E

3. Vi sätter ihop resultatet och får det hexadecimala talet D31E.

Vi tar ännu ett exempel genom att omvandla det binära talet 1001010110100110 10 till det hexadecimala talsystemet.

1. Först grupperar vi talet (börja alltid från höger, vi kommer se varför): 10 0101 0110 1001 1010
   Här ser vi att det inte blir fyra siffror i varje grupp eftersom den sista gruppen bara har två siffror. Det man gör då är att man lägger på nollor eftersom de inte kommer att förändra värdet på talet. Vi får därmed 0010 0101 0110 1001 1010 och kan börja på steg 2.

2. Varje grupp omvandlas för sig:

Bin	Dec	Hex
0010	2	2
0101	5	5
0110	6	6
1001	9	9
1010	10	A

3. Sist lägger vi ihop resultatet och får ut det hexadecimala talet 2569A.

Hexadecimala tal till binära tal

Som vi såg i förra avsnittet var det väldigt enkelt att omvandla från binära tal till hexadecimala tal, och det är lika enkelt att omvandla åt andra hållet. Varje siffra i det hexadecimala talet kommer ge fyra siffror i det binära talet. Skulle man få för få siffror lägger man bara på nollor till vänster tills man har fyra siffror i varje grupp.

Vi omvandlar talet F3C9 som exempel:

1. Vi börjar med att omvandla varje siffra till ett fyra bitars binärtal:

Hex	Dec	Bin
F	15	1111
3	3	0011
C	12	1100
9	9	1001

2. Här lägger vi ihop resultaten så att vi får ett binärt tal: 1111001111001001.

Här ser vi att vi måste lägga på nollor ibland för att vi hela tiden ska få fyra bitar. Skulle man däremot få nollor längst till vänster efter att man satt ihop alla bitarna kan man ta bort dem eftersom de inte har något värde.

Decimala tal till hexadecimala tal

När man ska omvandla från det decimala talsystemet till det hexadecimala talsystemet skulle man kunna använda sig av precis samma algoritm som när man omvandlar decimala tal till det binära talsystemet. Det enda man behöver tänka på då är att värdena för positionerna är annorlunda efter vi arbetar med talbas 16 istället för talbas 2. Dock är det krångligare att göra så för hexadecimala tal eftersom det är mycket jobbigare huvudräkning. När man jobbar med hexadecimala tal räcker det inte med att addera värdet på en position och se om det blir större eller mindre utan man måste upprepa processen för varje möjlig siffra på den positionen. Eftersom det finns 16 (0-f) möjliga siffror blir det snabbt jobbigt att använda den algoritmen.

Ett enklare sätt att omvandla decimala tal till det hexadecimala talsystemet är att först omvandla till det binära talsystemet, och sen gå därifrån till hexadecimala tal.

Som exempel kan vi omvandla talet 13_{10} till det hexadecimala talsystemet:

1. Vi börjar med omvandlingen till det binära talsystemet och skriver därför upp värdena på de olika positionerna: 16 8 4 2 1

2. Sedan sätter vi en etta i den näst högsta positionen:
   ```
      1
   16 8 4 2 1
   ```

3. Vi adderar 4 till 8 och får 12. 12 är mindre än 13 så vi sätter en etta här. Vi inser också att det enda vi kan addera till 12 för att få 13 är 1, och kan därför sätta ut resterande ettor och nollor:
   ```
      1  1  0  1
   16 8  4  2  1
   ```

4. När vi nu har omvandlat till det binära talsystemet kan vi börja omvandla därifrån till det hexadecimala talsystemet genom att dela upp talet i grupper om fyra. I det här fallet blir det bara en grupp: 1101.
Bin	Dec	Hex
1101	13	D

Det här exemplet är egentligen för kort för att vara vettigt, eftersom vi redan vet att $13_{10} = D_{16}$, men det illustrerar ändå hur man går tillväga för att översätta mellan decimala tal och hexadecimala tal.

3.4 Teckenbit

När man har binära tal är det svårt att sätta ett minus framför för att ange att ett tal är negativt, en dator vet inte vad minustecknet innebär. Av den anledningen finns ofta en teckenbit som anger om ett tal är negativt eller positivt. Om man till exempel har åtta bitar för att spara ett heltal kommer biten längst till vänster vara 0 om talet är positivt och 1 om talet är negativt. Eftersom den vänstraste biten kommer att användas för att representera tecken kommer ett tal med teckenbit, en ”signed integer”, bara kunna använda resterande sju bitar för representation av talets värde. Det här sättet att notera tal med en teckenbit kallas på engelska också för ”sign-magnitude representation”.

När man programmerar är man inte alltid intresserad av att kunna representera negativa tal, utan vill hellre kunna använda samtliga bitar för att representera ett positivt heltal. Då kan man i många programmeringsspråk istället använda ”unsigned integers”, alltså heltal utan teckenbit. Ett sådant tal måste alltid vara positivt, vilket gör att man måste vara säker på att det inte kommer sparas några negativa tal eftersom de kommer tolkas som positiva och därmed bli fel.

3.5 Övningsuppgifter

Nedan finns ett flertal uppgifter för att öva på övergångar mellan olika talbaser. Prova gärna flera av de alternativa metoder som finns för att gå mellan talsystemen.

Uppgift 3.5.1
Översätt följande tal från det binära talsystemet till det decimala talsystemet

a. 11_2
b. 101_2
c. 1111_2
d. $1\ 0000_2$
e. $111\ 0101_2$
f. 1011_2
g. $1\ 1101_2$
h. $1001\ 1010_2$
i. $1111\ 0011_2$
j. $1\ 0001\ 0110_2$
k. $10\ 0000\ 0000_2$
l. $10\ 0110\ 1000_2$

Uppgift 3.5.2
Översätt följande tal från det decimala talsystemet till det binära talsystemet

a. 2_{10}
b. 13_{10}
c. 52_{10}
d. 172_{10}
e. 26_{10}
f. 35_{10}
g. 49_{10}
h. 4_{10}
i. 77_{10}
j. 143_{10}
k. 200_{10}
l. 556_{10}

Uppgift 3.5.3
Översätt följande tal från det binära talsystemet till det hexadecimala talsystemet

a. 1010_2
b. $1001\ 1111_2$
c. 1000_2
d. $0111\ 1010\ 0011_2$
e. 1110_2
f. $0101\ 1100_2$
g. $0111\ 0111_2$
h. $1001\ 1001\ 1001_2$
i. $1011\ 0000_2$
j. $1001\ 1111_2$
k. $1100\ 1010_2$
l. $0010\ 0101\ 0110_2$

Uppgift 3.5.4
Översätt följande tal från det hexadecimala talsystemet till det binära talsystemet

a. 3_{16}
b. 13_{16}
c. 127_{16}
d. $5B3A_{16}$
e. $2A_{16}$
f. 60_{16}
g. 120_{16}
h. 330_{16}
i. 572_{16}
j. $12AB_{16}$
k. $F3F4_{16}$
l. $7A69_{16}$

Uppgift 3.5.5
Översätt följande tal från det hexadecimala talsystemet till det decimala talsystemet

a. $2B_{16}$
b. C_{16}
c. BB_{16}
d. $31A_{16}$
e. 27_{16}
f. $3D_{16}$
g. $33C_{16}$
h. $142E_{16}$
i. $BEEF_{16}$
j. $FADE_{16}$
k. $DEAD_{16}$
l. $BABE_{16}$

Uppgift 3.5.6
Översätt följande tal från det decimala talsystemet till det hexadecimala talsystemet

a. 77_{10}
b. 255_{10}
c. 2572_{10}
d. 13_{10}
e. 1337_{10}
f. 2056_{10}
g. 3243_{10}
h. 734_{10}
i. 64206_{10}
j. 51966_{10}
k. 48813_{10}
l. 912559_{10}

Uppgift 3.5.7
Översätt följande tal från det binära talsystemet till det oktala talsystemet

a. 101_2
b. $011\ 001_2$
c. $101\ 111\ 100_2$
d. 111_2
e. 100_2
f. $111\ 000_2$
g. $001\ 101_2$
h. $100\ 000\ 110_2$
i. $101\ 101_2$
j. $110\ 100\ 000_2$
k. $111\ 111\ 111_2$
l. $011\ 011\ 100\ 001_2$

Uppgift 3.5.8
Översätt följande tal från talbas 10 (det decimala talsystemet) till talbas 5

a. 10_{10}
b. 17_{10}
c. 125_{10}
d. 337_{10}
e. 25_{10}
f. 52_{10}
g. 77_{10}
h. 256_{10}
i. 83_{10}
j. 95_{10}
k. 130_{10}
l. 515_{10}

Uppgift 3.5.9
Översätt följande tal från respektive talbaser till talbas 10 (det decimala talsystemet)

a. $1{,}1_2$
b. $10{,}01_2$
c. $1{,}11_2$
d. $0{,}1011_2$
e. $\mathrm{A{,}2}_{16}$
f. $\mathrm{A{,}B}_{16}$
g. $\mathrm{1{,}F}_{16}$
h. $0{,}01_{16}$
i. $1{,}1_8$
j. $10{,}01_8$
k. $1{,}11_8$
l. $0{,}001_8$

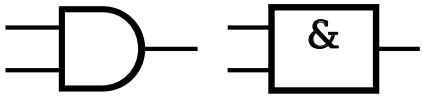
4. Digital logik

Digital logik handlar om utföra logiska operationer på binära uttryck. Mer information finns i föreläsningsbilderna, men en annan bra källa till information är `http://en.wikipedia.org/wiki/Logic_gate` och `http://www-inst.eecs.berkeley.edu/~cs61c/resources/dg-BOOL-handout.pdf`.

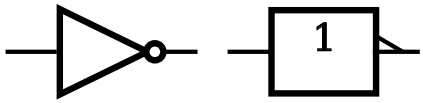
4.1 Övningsuppgifter

Uppgift 4.1.1
Nedan finns ett antal grindar enligt både amerikansk och europeisk notation. Vilka är grindarna, och hur noterar man dem matematiskt?

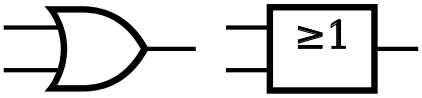
a.



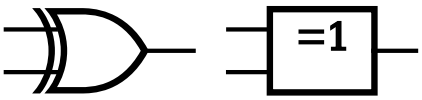
b.



c.



d.



Uppgift 4.1.2
Ange sanningstabellen för grindarna nedan:

a. OCH
b. ELLER
c. XOR
d. ICKE

Uppgift 4.1.3
Vad blir resultatet av följande bitvisa operationer?

a. 10 OCH 11
b. 111 OCH 101
c. 000 OCH 111
d. 1010 1001 OCH 1110 0010
e. 1001 1011 OCH 1000 0000
f. 1111 0000 OCH 1010 1010
g. 0111 1001 OCH 1111 1111
h. 0010 1011 OCH 1011 0010
i. 1110 1110 OCH 1010 0101

Uppgift 4.1.4
Vad blir resultatet av följande bitvisa operationer?

a. 10 ELLER 11
b. 111 ELLER 101
c. 000 ELLER 111
d. 1010 1001 ELLER 1110 0010
e. 1001 1011 ELLER 1000 0000
f. 1111 0000 ELLER 1010 1010
g. 0111 1001 ELLER 1111 1111
h. 0010 1011 ELLER 1011 0010
i. 1110 1110 ELLER 1010 0101

Uppgift 4.1.5
Vad blir resultatet av följande bitvisa operationer?

a. 10 XOR 11
b. 111 XOR 101
c. 000 XOR 111
d. 1010 1001 XOR 1110 0010
e. 1001 1011 XOR 1000 0000
f. 1111 0000 XOR 1010 1010
g. 0111 1001 XOR 1111 1111
h. 0010 1011 XOR 1011 0010
i. 1110 1110 XOR 1010 0101

Uppgift 4.1.6
Vad blir resultatet av följande bitvisa operationer?

a. ICKE 10
b. ICKE 111
c. ICKE 000
d. ICKE 1010 1001
e. ICKE 1001 1011
f. ICKE 1111 0000

Uppgift 4.1.7
Hur gör man om man vill veta om den femte biten i en bitsträng är ett eller noll?

Uppgift 4.1.8
Hur gör man om man vill sätta den femte biten i en bitsträng till ett?

Uppgift 4.1.9
Hur gör man om man vill sätta den femte biten i en bitsträng till noll?

Uppgift 4.1.10
Hur gör man om man vill toggla den femte biten i en bitsträng till noll? Med att toggla menas att man byter läge, så att en etta blir en nolla och en nolla blir en etta.

5. Aritmetik

Precis som med decimala tal kan man applicera de fyra räknesätten även på binära tal. I kursen Datorsystem kommer vi att fokusera på addition, subtraktion och multiplikation.

5.1 Addition

Addition av binära tal går till precis som addition av decimala tal. Vi kan börja med ett exempel på hur man adderar två decimala tal så att vi kan se likheterna senare. Vi har talen 123 och 358 och ska lägga ihop dem. Hur gör vi? Till att börja med finns det flera sätt, man kan till exempel använda sig av både mellanled och uppställning. Här kommer vi dock använda uppställning eftersom det är det enklaste sättet sen när vi kommer till binär addition.

1. Vi börjar med att ställa upp talen:

$$\begin{array}{cccc} & 1 & 2 & 3 \\ + & 3 & 5 & 8 \\ \hline \end{array}$$

2. Sedan lägger vi ihop varje kolumn för sig (det vill säga entalen, tiotalen och hundratalen var för sig):

$$\begin{array}{cccc} & & 1 & \\ & 1 & 2 & 3 \\ + & 3 & 5 & 8 \\ \hline & & & 1 \end{array}$$

   Eftersom 3+8=11 behöver vi flytta tiotalssiffran till rätt kolumn, därav att det hamnar en etta ovanför kolumnen med tiotalen.

3. Nu lägger vi ihop alla tiotalen:

$$\begin{array}{cccc} & & \cancel{1} & \\ & 1 & 2 & 3 \\ + & 3 & 5 & 8 \\ \hline & & 8 & 1 \end{array}$$

   Eftersom vi nu har räknat med ettan vi fick över förut kan vi stryka över den.

4. Till sist lägger vi ihop hundratalen:

$$\begin{array}{cccc} & & \cancel{1} & \\ & 1 & 2 & 3 \\ + & 3 & 5 & 8 \\ \hline & 4 & 8 & 1 \end{array}$$

Vi ser här att om man adderar de två decimala talen 123 och 358 är får man resultatet 481.

Om vi då tar två binära tal och adderar dem kan vi göra på samma sätt. Som exempel kan vi prova att addera de två talen binära talen 1101 och 1001:

1. Vi börjar med att ställa upp talen:

$$\begin{array}{cccccc} & 1 & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline \end{array}$$

2. Vi börjar med första positionen och lägger ihop de båda ettorna. Vi ser att 1_2+1_2 ger 10_2. Precis som i det decimala exemplet behöver vi se till att vi får vår nya etta i rätt kolumn för sin position:

$$\begin{array}{cccccc} & & & 1 & \\ & 1 & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline & & & & 0 \end{array}$$

3. Vi fortsätter med nästa kolumn:

$$\begin{array}{cccccc} & & & \cancel{1} & & \\ & 1 & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline & & & 1 & 0 \end{array}$$

   Eftersom vi har tagit hand om vår ”extra” etta kan vi stryka den.

4. Nästa steg är att addera nästa kolumn, vilket går utan bekymmer:

$$\begin{array}{cccccc} & & & \cancel{1} & & \\ & 1 & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline & & 1 & 1 & 0 \end{array}$$

5. I detta steg har vi återigen två ettor och måste flytta över en etta till nästa position:

$$\begin{array}{cccccc} 1 & & & \cancel{1} & & \\ & 1 & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

6. Eftersom vi fick en extra etta i förra steget får vi lägga till en extra position. Vårt nya tal blir alltså en bit längre än våra ursprungstal:

$$\begin{array}{cccccc} \cancel{1} & & & \cancel{1} & & \\ & 1 & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & 0 \end{array}$$

Att addera de två binära talen 1101 och 1001 ger resultatet 10110. Om vi lite snabbt omvandlar till decimalt kan vi kontrollera: $1101_2 = 13_{10}$, $1001_2 = 9_{10}$, $13 + 9 = 22$, $22_{10} = 10110_2$. Vi har alltså räknat rätt!

Vi kan ta ännu ett exempel, med lite större tal denna gång. Vi adderar talen 1011001 och 11001:

1. Precis som förra gången börjar vi med att ställa upp talen:

$$\begin{array}{ccccccccc} & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ + & & & 1 & 1 & 0 & 1 & 1 \\ \hline \end{array}$$

2. Vi börjar med att konstatera att det ena talet är större än det andra, men det kommer egentligen bara göra det enklare för oss. Vi adderar de först två ettorna och får en över, som vi flyttar till nästa kolumn:

$$\begin{array}{ccccccccc} & & & & & & 1 & \\ & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ + & & & 1 & 1 & 0 & 1 & 1 \\ \hline & & & & & & & 0 \end{array}$$

3. Här har vi återigen två ettor och kommer få föra över en till nästa kolumn:

$$\begin{array}{ccccccccc} & & & & & 1 & \cancel{1} & \\ & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ + & & & 1 & 1 & 0 & 1 & 1 \\ \hline & & & & & & 0 & 0 \end{array}$$

4. I detta steg har vi bara en etta, så vi får:

$$\begin{array}{ccccccccc} & & & & & \cancel{1} & \cancel{1} & \\ & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ + & & & 1 & 1 & 0 & 1 & 1 \\ \hline & & & & & 1 & 0 & 0 \end{array}$$

5. Här har vi återigen två ettor och får flytta över till nästa position:

$$\begin{array}{ccccccccc} & & & 1 & & \cancel{1} & \cancel{1} & \\ & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ + & & & 1 & 1 & 0 & 1 & 1 \\ \hline & & & & 0 & 1 & 0 & 0 \end{array}$$

6. Nu ser vi att vi har hela tre ettor, vilket ger $1_2 + 1_2 + 1_2 = 11_2$. Därmed får vi både en etta i vår nuvarande position och en överflyttad etta i positionen till höger:

		1	~~1~~		~~1~~	~~1~~	
	1	0	1	1	0	0	1
+			1	1	0	1	1
			1	0	1	0	0

7. Vi har nu kommit in på den bit där bara det ena talet har siffror, men eftersom vi fick en extra etta från förra positionen får vi addera även här:

		~~1~~	~~1~~		~~1~~	~~1~~	
	1	0	1	1	0	0	1
+			1	1	0	1	1
		1	1	0	1	0	0

8. I sista steget har vi bara en ensam etta och behöver därmed inte göra någon addition, utan vi får direkt ned ettan till slutsumman:

		~~1~~	~~1~~		~~1~~	~~1~~	
	1	0	1	1	0	0	1
+			1	1	0	1	1
	1	1	1	0	1	0	0

Resultatet av vår addition är det binära talet 1110100, vilket omvandlat till det decimala talsystemet blir 116. Våra två termer var $1011001_2 = 89_{10}$ och $11011_2 = 27_{10}$. 89 + 27 = 116, och vi kan därmed visa att vi har räknat rätt.

5.2 Subtraktion

När vi räknar subtraktion brukar vi använda oss av några olika metoder. Precis som med addition skulle man kunna använda sig av både mellanled och uppställning, och för en människa fungerar båda sätten ungefär lika bra. Dock fungerar båda sätten inte lika bra för en dator. Om man skulle tillämpa uppställning för subtraktion kan det hända att man behöver ”låna” från en siffra i en högre position, något vi som människor kan göra relativt enkelt eftersom vi ser var nästa siffra att låna ifrån finns. En dator däremot kommer behöva gå igenom varje position till vänster om den nuvarande för att se om det finns något att låna där. Att bygga en subtraherare för en dator som fungerar på det viset blir krångligt eftersom det inte finns något bra sätt att låna från steg man inte redan processat. Om man jämför med additionen är det mycket enklare att skicka en överbliven siffra vidare till nästa steg.

Av den anledningen används en annan metod för subtraktion för datorer. Istället för att subtrahera ett tal adderar man talets negation. Ett tals negation kan man får fram genom att ta fram **tvåkomplementet** för den term som ska dras bort. Tvåkomplementet är en negativa variant av ett tal. Om man har 12-7 kan man lika gärna se det som 12+(-7) eftersom det ger samma resultat, men istället för två positiva tal har vi ett positivt och ett negativt tal. På samma sätt kommer vi göra med binära tal. För att göra den term som ska dras bort negativ tar vi fram tvåkomplementsformen för termen.

Innan man tar fram tvåkomplementet för det tal som ska subtraheras måste vi se till att de två tal som ska subtraheras är lika långa. Det gör man genom att det tal som eventuellt är kortare fylls på med nollor från vänster tills båda talen har samma antal bitar. Den här operationen kallas att ”padda” talet, från engelskans ”padding”, stoppning. Det gör man eftersom formeln för att ta fram ett tvåkomplement är beroende av att veta hur många bitar ett tal är.

Båda talens teckenbitar måste också finnas med. Har man talet 5_{10} blir det 0101_2, eftersom den första nollan markerar talets tecken. Har man istället -5_{10} blir den binära varianten 1101_2. Även när man räknar addition tar man hänsyn till teckenbitarna, men eftersom samtliga exempel hittills har varit addition av två positiva tal har det inte funnits någon poäng med att ha med dem.

När båda talen är lika långa beräknar man tvåkomplementet med följande algoritm:

1. Invertera, det vill säga byt 1 mot 0 och 0 mot 1, samtliga bitar i det binära talet.

2. Addera 1_2 till det inverterade talet.

Vi kan ta fram tvåkomplementet för 011010_2 som exempel. Vi börjar med att invertera samtliga bitar och får 100101_2. Sedan adderar vi 1_2 och får 100110_2. Som ännu ett exempel kan vi ta fram tvåkomplementet till ett något större tal, 0101100110_2. Återigen börjar vi med att invertera talet och får 1010011001_2. Vi adderar 1_2 och det ger resultatet 1010011010_2, och därmed har vi fått fram tvåkomplementet. Notera att samtliga tvåkomplement har en etta som första siffra eftersom det är teckenbiten som noterar att talet är negativt.

Nu när vi vet hur man får fram tvåkomplementsformen för ett binärt tal kan vi också subtrahera. Som exempel tar vi talet 0100110_2 - 010101_2.

1. Vi börjar med att se till att båda talen är lika långa, så andra termen måste paddas: 0010101_2.
2. Vi börjar med att ta fram tvåkomplementet för 0010101_2. Först inverterar vi och får 1101010_2, och sedan adderar vi 1_2. Resultatet och vårt tvåkomplement blir därmed 1101011_2.
3. Vi ställer upp våra två termer:

	0	1	0	0	1	1	0
+	1	1	0	1	0	1	1

4. Vi adderar ihop de två termerna. Vid det här laget bör alla kunna addera binära tal, så vi sparar lite plats och går direkt till resultatet:

1	1		1	1	1		
	0	1	0	0	1	1	0
+	1	1	0	1	0	1	1
1	0	0	1	0	0	0	1

   Det vi noterar här är att allt som hamnar utanför våra ursprungliga bitar kommer kastas, annars skulle vi få ett större tal våra termer. Här markerar det vertikala strecket var gränsen går, allt till vänster om det kastas. Vi kan också se att resultatet är positivt, genom att teckenbiten är positiv.

Vi får alltså att 0100110_2 - 010101_2 = 010001_2. En snabbkoll via det decimala talsystemet ger 38 - 21 = 17, vilket överensstämmer med det resultat vi fick när vi adderade binärt. Därmed kan vi också se att vi räknat rätt genom att kontrollräkna i det decimala talsystemet.

Som ännu ett exempel kan vi ta 0110110_2 - 01111_2.

1. Återigen har det negativa talet färre bitar, så vi får förlänga: 0001111_2.
2. Vi tar fram tvåkomplementet till 0001111_2 genom att först invertera alla bitar så att vi får 1110000_2. Sist adderar vi 1_2 och får 1110001_2.
3. Här ställer vi upp våra termer:

	0	1	1	0	1	1	0
+	1	1	1	0	0	0	1

4. Sist lägger vi ihop de två termerna:

1	1	1					
	0	1	1	0	1	1	0
+	1	1	1	0	0	0	1
1	0	1	0	0	1	1	1

   Även här får vi en överflödig etta som ska plockas bort eftersom den går utanför bitlängden av de ursprungliga talen. Återigen ser vi också att talet är positivt med hjälp av teckenbiten.

Subtraktion med decimaler

När en av termerna innehåller decimaler blir det lite knepigare eftersom det kan vara svårt att veta hur man ska göra efter att man inverterat termen som ska subtraheras. Steget efter inverteringen är att addera 1, men ska man göra det före eller efter decimalkommat? Det enklaste svaret är att man lägger till ettan sist i talet, oavsett om talet innehåller decimaler eller inte. Man kan också strunta i decimalerna medan man tar bort tvåkomplementet.

- **Tumregel:** Addera alltid ettan till högraste positionen i talet vars tvåkomplement ska tas fram.

Vi tar ett exempel för att illustrera: $01101{,}1_2$ - 0101_2.

1. Först måste båda talen vara lika långa. I det här fallet måste vi därmed börja med att se till att värdet på den andra termen inte ändras när vi paddar, så vi börjar med att lägga till en decimal: $0101{,}0_2$. Därefter paddar vi och får $00101{,}0_2$.

2. Nu ska tvåkomplementet tas fram och då börjar vi med att invertera $00101{,}0_2$, vilket blir $11010{,}1_2$. Därefter adderar vi 1 till den högraste positionen och ignorerar decimalkommat:

					~~1~~	
	1	1	0	1	0	1
+						1
	1	1	0	1	1	0

3. Nu har vi tvåkomplementet och kan ställa upp våra termer:

~~1~~	~~1~~	~~1~~	~~1~~	~~1~~		
	0	1	1	0	1,	1
+	1	1	0	1	1,	0
1	0	1	0	0	0,	1

   Här har vi lagt tillbaka decimalkommat för att kunna få korrekt resultat. En snabb dubbelkoll ger följande: $01101{,}1_2$ - 0101_2 → $13{,}5_{10}$ - 5_{10} = $8{,}5_{10}$. $8{,}5_{10}$ = $01000{,}1_2$, och vi kan därmed se att vi fått ett korrekt resultat. Teckenbiten visar också att vårt resultat är positivt.

När resultatet blir negativt

Om den andra termen är större än den första kommer vi få ett negativt resultat. Det betyder att vi kommer att få svaret på tvåkomplementsform, och måste då vända tillbaka det för att se vad svaret blev.

Som exempel kan vi ta 5 - 10 = -5, som i binär form blir 0101_2 - 01010_2 med teckenbitar.

1. Vi tar fram tvåkomplementet till 01010_2 genom att först invertera alla bitar så att vi får 10101_2. Sist adderar vi 1_2 och får 10110_2.

2. Här ställer vi upp våra termer:

	0	0	1	0	1
+	1	0	1	1	0

3. Så adderar vi de två termerna:

		~~1~~			
	0	0	1	0	1
+	1	0	1	1	0
	1	1	0	1	1

   Svaret blir 11011_2, vilket är 27_{10}, och vi kan därmed se att något är fel med svaret eftersom det är större än båda våra termer. Svaret är nämligen negativt (vilket också ses i teckenbiten), och vi får vända tillbaka det från tvåkomplementsformen så att vi kan se den positiva representationen av vårt negativa tal.

4. Vi börjar med att invertera 11011_2 och får då 00100_2. Sedan adderar vi 1 och får 00101_2, vilket är 5_{10}. Svaret är alltså -5, eftersom vårt svar är negativt.

5.3 Multiplikation

Multiplikation brukar implementeras som upprepade steg av addition, vilket är samma metod som används när man ställer upp tal och multiplicerar. Man börjar med att ställa upp talen, och sedan adderar man multiplikatorn för varje etta i multiplikanden. Om multiplikanden innehåller nollor kommer den raden i additionen vara nollor. För varje steg i additionen skiftar man talen till vänster.

Vi tar ett exempel och multiplicerar 30*5=150. 30_{10}=11110_2 och 5_{10}=101_2.

1. Först ställer vi upp faktorerna:

	1	1	1	1	0
$*$			1	0	1

2. Den minst signifikanta biten i multiplikanden är 1, så vi adderar med hela multiplikatorn:

	1	1	1	1	0
$*$			1	0	~~1~~
	1	1	1	1	0

3. Nästa bit i multiplikanden är 0, så vi adderar bara nollor:

	1	1	1	1	0
$*$			1	~~0~~	~~1~~
	1	1	1	1	0
0	0	0	0	0	

4. Sista biten i multiplikanden är 1, så vi adderar hela multiplikatorn igen:

		1	1	1	1	0
	$*$			~~1~~	~~0~~	~~1~~
		1	1	1	1	0
	0	0	0	0	0	
1	1	1	1	0		

5. Vi har multiplicerat klart och kan börja addera:

			1	1	1	1	0
		$*$			~~1~~	~~0~~	~~1~~
			1	1	1	1	0
		0	0	0	0	0	
$+$	1	1	1	1	0		

6. Addera kan vi vid det här laget och tar allt i ett steg:

			1	1	1	1	0
		$*$			~~1~~	~~0~~	~~1~~
~~1~~	~~1~~	~~1~~	~~1~~				
			1	1	1	1	0
		0	0	0	0	0	
$+$	1	1	1	1	0		
1	0	0	1	0	1	1	0

$1001\ 0110_2 = 150_{10}$ och vi kan därmed se att vi fått rätt resultat.

Multiplikation med decimaler

När man multiplicerar två tal som innehåller decimaler, till exempel 5,5 eller 2,25, kan det vara svårt att veta hur många decimaler svaret ska ha. Regeln är att antalet decimaler i svaret är lika med summan av antalet decimaler i faktorerna. Regeln gäller för det decimala talsystemet såväl som det binära, så vi kan ta ett exempel från det decimala talsystemet: 5.5*2,25=12,375. Summan av antalet decimaler i faktorerna är tre, och svaret har också tre decimaler.

Som exempel kan vi multiplicera 10,5 med 5,5. $10{,}5_{10}$=$1010{,}1_2$ och $5{,}5_{10}$=$101{,}1_2$.

1. Först ställer vi upp faktorerna:

	1	0	1	0,	1
$*$		1	0	1,	1

2. Den minst signifikanta biten i multiplikanden är 1, så vi adderar med hela multiplikatorn:

	1	0	1	0,	1
$*$		1	0	1,	1
	1	0	1	0	1

3. Nästa bit i multiplikanden är 1, så vi adderar återigen hela multiplikatorn:

	1	0	1	0	1
$*$		1	0	~~1~~	~~1~~
	1	0	1	0	1
1	0	1	0	1	

4. Nästa bit i multiplikanden är 0, så vi adderar med 0:

		1	1	1	1	0
	$*$		1	~~0~~	~~1~~	~~1~~
		1	+	1	+	1
	1	0	1	0	1	
0	0	0	0	0		

5. Sista biten i multiplikanden är 1, så vi adderar multiplikanden igen:

			1	1	1	1	0
		$*$		~~1~~	~~0~~	~~1~~	~~1~~
			1	0	1	0	1
		1	0	1	0	1	
	0	0	0	0	0		
1	0	1	0	1			

6. Vi har multiplicerat klart och kan börja addera:

				1	1	1	1	0
			$*$		~~1~~	~~0~~	~~1~~	~~1~~
				1	0	1	0	1
			1	0	1	0	1	
		0	0	0	0	0		
$+$	1	0	1	0	1			

7. Resultatet av additionen:

				1	1	1	1	0
			$*$		~~1~~	~~0~~	~~1~~	~~1~~
		~~1~~	~~1~~	~~1~~				
				1	0	1	0	1
			1	0	1	0	1	
		0	0	0	0	0		
$+$	1	0	1	0	1			
	1	1	1	0	0	1	1	1

8. Vi räknar ut hur många decimaler vi ska ha i vårt svar genom att summera antalet decimaler i faktorerna. Båda faktorerna har en decimal var och vi ska därmed ha två decimaler i vårt svar. $1110\ 0111_2 \rightarrow 11\ 1001{,}11_2 = 57{,}75_{10}$. Vi kontrollräknar: 10,5*5,5=57,75, och vi kan se att svaret är rätt.

Multiplikation med teckenbit

Multiplikation måste inte alltid utföras med två positiva tal. En eller båda faktorerna kan också vara negativ, och vi kan använda följande regler:

- Multiplikation av två negativa tal ger alltid ett positivt resultat.
- Multiplikation av ett negativt och ett positivt tal ger alltid ett negativt resultat.

Vi tillämpar dessa regler genom att addera faktorernas teckenbitar och kasta eventuellt overflow:

Ingående teckenbitar:	Två positiva tal	Två negativa tal	Ett positivt och ett negativt tal
Teckenbit för svaret:	$0_2 + 0_2 = 0_2$	$1_2 + 1_2 = 1\|0_2$	$0_2 + 1_2 = 1_2$

5.4 Övningsuppgifter

Uppgift 5.4.1
Vad blir resultatet av följande tal?

a. $1_2 + 1_2$
b. $0_2 + 1_2$
c. $1_2 + 0_2$
d. $11_2 + 1_2$
e. $1010_2 + 101_2$
f. $1100_2 + 1100_2$
g. $1001_2 + 1011_2$
h. $1111_2 + 1_2$
i. $1100\ 1001_2 + 1000\ 1011_2$
j. $1011\ 1111_2 + 1111\ 1111_2$
k. $10\ 0010_2 + 1011\ 0100_2$
l. $1111\ 1010_2 + 1111\ 1110_2$

Uppgift 5.4.2
Vad blir resultatet av följande tal?

a. 1_2 - 1_2
b. 10_2 - 1_2
c. 1_2 - 0_2
d. 11_2 - 1_2
e. 1010_2 - 101_2
f. 1101_2 - 1100_2
g. 1111_2 - 1011_2
h. 1110_2 - 1_2
i. $1100\ 1001_2$ - $1000\ 1011_2$
j. $1011\ 1111_2$ - $111\ 1111_2$
k. $10\ 0010_2$ - 1100_2
l. $1111\ 1010_2$ - $1001\ 1110_2$

Uppgift 5.4.3
Vad blir resultatet av följande tal?

a. 1_2 * 1_2
b. 10_2 * 1_2
c. 1_2 * 0_2
d. 11_2 * 11_2
e. 1010_2 * 101_2
f. 1101_2 * 1100_2
g. 1111_2 * 1011_2
h. 10_2 * 101_2
i. $1100\ 1001_2$ * $1000\ 1011_2$
j. $1011\ 0101_2$ * $111\ 1111_2$
k. $10\ 0010_2$ * 1100_2
l. $1111\ 1010_2$ * $1001\ 1110_2$[1]

Uppgift 5.4.4
Vad blir resultatet av följande tal?

a. $10{,}1_2 + 1_2$
b. $1{,}1_2 + 1{,}1_2$
c. $10{,}01_2 + 10_2$
d. $1101{,}1_2 + 1001{,}11_2$
e. $10{,}1_2$ - 1_2
f. $11{,}1_2$ - $0{,}1_2$
g. $100{,}01_2$ - $1{,}1_2$
h. $1101{,}11_2$ - $101{,}01_2$
i. $11{,}1_2$ * 101_2
j. $10{,}01_2$ * 11_2
k. $1000{,}1_2$ * $10{,}1_2$
l. $1111{,}11_2$ * $10{,}01_2$

6. Flyttal

Namnet flyttal kommer från engelskans ”floating point numbers”, och att talen kallas just flytande är för att decimalmarkeringen kan flyttas för att markera hur stort talet är. Flyttal är det sätt som används i en dator för att representera reella tal[1]. Flyttal är väldigt användbara för att representera väldigt stora eller små tal, men också rationella tal[2]. Med stora tal menar man till exempel tal som $6 * 10^{23}$ eller $1,4 * 10^{32}$. Ett litet tal skulle kunna vara $1,6 * 10^{-35}$ eller $6,6 * 10^{-12}$, och med krångliga tal skulle man kunna mena π (pi) eftersom det har väldigt många decimaler.

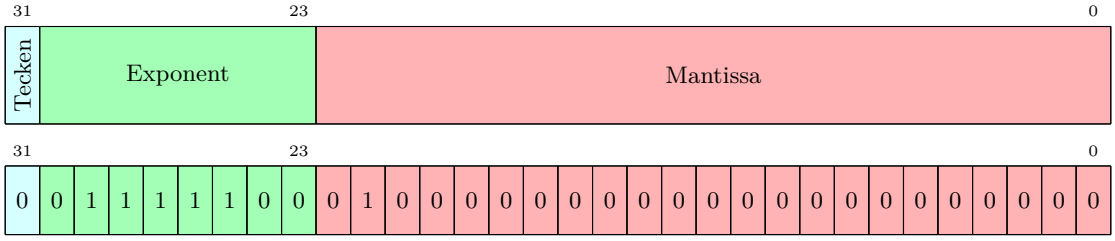
Problemet med att representera tal som till exempel π är att det är svårt att vara exakt när man bara har 32 eller 64 bitar till sin användning. Av den anledningen avrundas de flesta flyttal, och vissa är extra svåra att få rätt, bland andra just π men också det något mer oväntade talet 0,1. Den som är intresserad av att läsa mer om flyttal i allmänhet kan läsa mer på `http://en.wikipedia.org/wiki/Floating_point`.

6.1 IEEE 754

Idag finns en gemensam standard för hur en dator ska hantera flyttal, och denna standard kallas IEEE 754. IEEE står för ”Institute of Electrical and Electronics Engineers” och är en organisation som bland annat arbetar med att ta fram standarder inom ett flertal olika områden. Just standarden IEEE 754 publicerades första gången redan 1985, men fick en uppdatering 2008 för att bland annat inkludera hur man hanterar decimala flyttal. I standarden beskrivs hur man hanterar flyttal för olika antal bitar, nämligen 16, 32, 64 och 128 bitar. Ju fler bitar man använder desto större och mer precisa tal kan beskrivas.

32-bitars flyttal

I ett 32-bitars flyttal enligt standarden IEEE 754 finns tre fält. Det första fältet består av en bit, den mest signifikanta biten (biten längst till vänster), och innehåller teckenbiten. Nästa fält har 8 bitar och innehåller exponentens värde. Det tredje och sista fältet har 23 bitar och kallas för **mantissa**. I mantissan hittar man de siffror som kommer att ge talet dess värde efter att decimalpunkten hamnat på rätt ställe.



Figur 6.1: Ett 32-bitars flyttal enligt IEEE 754

För att få ut flyttalets värde (v) finns följande formel:

$$v = (-1)^{teckenbit} \times 2^{exponent-127} \times (1, mantissa_2)$$

där *teckenbit* är värdet på teckenbiten, *exponent* är värdet i exponentfältet och där *mantissa* är värdet i mantissa-fältet. Anledningen till att man tar exponenten minus 127 är för att man ska kunna representera tal som är både större och mindre än noll. Med åtta bitar skulle man kunna ha en exponent värd upp till och med 255, men då skulle man inte kunna ha några exponentvärden än noll. Därför delade man

[1] `http://sv.wikipedia.org/wiki/Reella_tal`
[2] `http://sv.wikipedia.org/wiki/Rationella_tal`

255 på två, vilket egentligen ger 127,5 men eftersom vi arbetar med heltal avrundades det till 127. På engelska används termen *bias* för det tal som subtraheras från exponentvärdet.

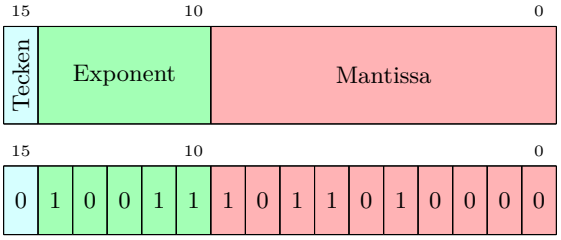
Vi kan använda siffrorna i figuren för att illustrera hur man räknar ut det decimala värdet av ett flyttal. Vi börjar med teckenbiten och ser att den är 0, vilket ger $-1^0 = 1$. I nästa steg räknar vi ut exponenten. 1111100_2 ger 124_{10}, och vi kommer då få $124 - 127 = -3$.

När man räknar ut mantissan börjar man alltid med att sätta ut en binär etta, vilket syns i formeln genom att det står $(1, mantissa_2)$. Vi får alltså $1_2 + 0{,}01_2 = 1{,}01_2$. Omvandlat till det decimala talsystemet blir det 1,25. När vi nu har räknat ut alla variabla delar i vår formel kan vi sätta ihop den och räkna ut vårt slutresultat: $v = 1 \times 2^{-3} \times 1,25 = 0,125 * 1,25 = 0,15625$.

Man kan också räkna ut resultatet genom att flytta decimaltecknet i mantissan det antal steg som exponenten anger. I vårt exempel har vi exponenten -3, vilket i formeln ger att 2^{-3} ska multipliceras med $1 + mantissan$. Om vi behåller mantissan som ett binärt tal kan vi helt enkelt flytta decimaltecknet tre steg åt vänster, och vi går därmed från $1{,}01_2$ till $0{,}00101_2$. Nu kan vi omvandla till det decimala talsystemet och får 1/8 + 1/32 vilket är 0,15625.

16-bitars flyttal

Precis som i ett 32-bitars flyttal enligt IEEE 754 har ett 16-bitars flyttal tre fält, med samma innebörd som i 32-bitars talet. Skillnaden är att vi bara har 16 bitar att tillgå, och varje fält får därmed färre bitar. Teckenfältet är fortfarande en bit långt, medan exponentfältet är fem bitar och mantissan 10 bitar långt.



Figur 6.2: Ett 16-bitars flyttal enligt IEEE 754

Eftersom det är en annan indelning av fälten behöver vi modifiera formeln en aning. Istället för att ta exponenten minus 127 tar vi exponenten minus 15. Anledningen är fortfarande samma, men eftersom exponenten bara har fem bitar måste vi ändra från 127 till 15. Formeln för få ut det decimala värdet ur ett 16-bitars flyttal blir då $v = (-1)^{teckenbit} \times 2^{exponent-15} \times (1, mantissa_2)$.

Om vi använder talet i figur 6.2 som exempel kan vi se hur man använder formeln för att få ut det decimala värdet. Teckenbiten är noll, så vi har ett positivt tal. Sedan beräknar vi värdet i exponentfältet och subtraherar med 15: 10011_2=19_{10} → 19-15=4.

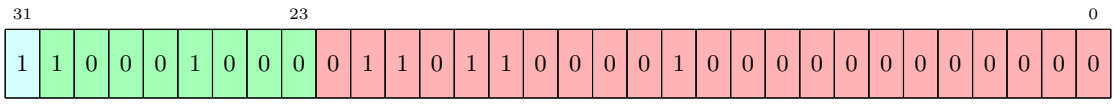
Nästa steg är att beräkna värdet i mantissan. Ur fältet får vi 1011010000, vilket ger $1{,}101101_2$. När vi nu har värdet på exponenten och mantissan kan vi flytta decimaltecknet rätt antal steg. Exponenten är fyra, så vi får följande när vi flyttar decimaltecknet fyra steg: $1{,}101101_2 \rightarrow 11011{,}01_2$=$27{,}25_{10}$.

Flyttalet 0100111011010000 enligt IEEE 754 är alltså 27,25 i det decimala talsystemet.

Förflyttning av decimaltecknet

Hur kommer det sig att vi kan flytta decimaltecknet som vi gjorde ovan? Eftersom mantissan sparas i det binära talsystemet är varje position värd en tvåpotens, exempelvis 2^3 eller 2^{-2}. Att flytta decimaltecknet ett steg åt höger är att multiplicera det nuvarande värdet med 2^1, att gå två steg är detsamma som att multiplicera med 2^2 och så vidare. Att gå åt vänster fungerar på samma sätt, ett steg är likvärdigt med att multiplicera med 2^{-1}, två steg är som att multiplicera med 2^{-2} och så vidare på samma sätt. Därför kan vi se att om vi ska multiplicera mantissan med 2^{-3} ska vi flytta decimaltecknet tre steg åt vänster. Skulle vi ha fått fram att exponenten hade gett oss 2^5 skulle vi ha kunnat flytta decimaltecknet fem steg åt höger.

Vi kan ta ett till exempel för att se hur det fungerar. Vi kan se i figur 6.3 hur våra 32 bitar är ordnade.



Figur 6.3: Ett annat 32-bitars flyttal enligt IEEE 754

Vi använder oss av samma formel som tidigare: $v = (-1)^{sign} \times 2^{exponent-127} \times (1, mantissa_2)$. Eftersom vår teckenbit den här gången är 1 får vi $(-1)^1 = -1$ och vi kommer därmed få ett negativt tal. Härnäst beräknar vi värdet i exponentfältet, vilket ger $10001000_2 = 136_{10}$. Vi beräknar $exponent - 127$, vilket ger 136-127=9.

Vi plockar ut värdet ur mantissan och adderar 1, vilket ger $1{,}01101100001_2$. Från exponenten kan vi dra slutsatsen att vi ska flytta decimaltecknet nio steg åt höger, vilket ger $1011011000{,}01_2$. Omvandlat till det decimala talsystemet blir det 728,25. Dock ska vi också multiplicera med -1, som kommer från vår teckenbit, och får som slutresultat det negativa talet -728,25.

Vi skulle också kunna göra som i föregående exempel och multiplicera de olika delarna istället för att flytta decimaltecknet. Vi får då följande ekvation: $-1 \times 2^9 \times (1_2 + 1, 01101100001_2) = -1 \times 512 \times (1 + \frac{865}{2048} = -1 \times 512 \times 1, 42236328125 = -728, 25)$

Som vi ser blir det snabbt enklare att bara flytta decimaltecknet för att slippa räkna med så här pass små bråk.

Att representera vissa tal

Ett problem med formatet för flyttal är att vissa tal är svåra att representera. Vissa tal är svåra att representera över huvud taget, som till exempel oändligheten, men även ett tal som noll är svårt att representera. Eftersom formeln alltid innehåller minst en etta i mantissan måste man ha ett specialfall för att representera noll, precis som man har specialfall för att representera oändligheten. Rent praktiskt finns en formel för väldigt små tal som skiljer sig från den formel som vanligtvis används, men vi kommer inte att ta upp den inom kursen.

I tabell 6.1 finns en lista över specialfall och deras representation i 16-bitars flyttal enligt IEEE 754.

Flyttalsrepresentation	**Specialfall**
0 00000 0000000000	0
1 00000 0000000000	-0
0 11111 0000000000	∞ (oändligheten)
1 11111 0000000000	$-\infty$ ($-$oändligheten)

Tabell 6.1: Specialfall för 16-bitars flyttal enligt IEEE 754

6.2 Aritmetik

Precis som med ”vanliga” binära tal kan man räkna med flyttal. Det är dock inte lika simpelt att räkna med flyttal, så vi kommer gå igenom hur man räknar addition, subtraktion och multiplikation med flyttal. Division med flyttal är snarlikt multiplikation, men av samma anledning som vi inte tar upp division av ”vanliga” binära tal kommer vi heller inte ta upp division av flyttal.

Addition

Vid addition av två flyttal måste man ta hänsyn till om exponenterna för de båda termerna är lika. Om de inte är det måste man skifta upp det minsta talet så att de blir lika stora. Anledningen till att man skiftar upp exponenten för det minsta talet är för att det är större risk att få overflow om man istället skiftar ned exponenten för det större talet.

Vi tar talet 0 11000 0111000000 + 0 11010 1011000000 som exempel, och börjar med att beräkna exponenterna i de båda termerna.

Första termen har exponentbitarna 11000_2, alltså 24_{10}. Enligt formeln ska vi subtrahera exponentvärdet med 15, vilket ger 24-15=9. Andra termen har exponentbitarna 11010_2, alltså 26_{10}. Återigen subtraherar vi exponentvärdet med 15 och får 26-15=11.

De båda talen har inte samma exponent, vilket innebär att vi måste öka exponenten med två (11-9=2) för tal A. För att talet inte ska ändra värde måste mantissan minskas när vi ökar exponenten, vilket ger $1,0111 >> 0,10111 >> 0,010111$. Observera att den första ettan kommer ur formeln för flyttal.

När båda talen nu har samma exponent kan vi addera deras respektive mantissor.

~~1~~	~~1~~	~~1~~	~~1~~	~~1~~			
	0,	0	1	0	1	1	1
+	1,	1	0	1	1	0	0
1	0,	0	0	0	0	1	1

Vi får resultatet $10,000011_2$, vilket är mantissan i det resulterande talet. Mantissan behöver dock normeras eftersom alla tal enligt formeln ska ha följande form: 1,mantissan. Vårt resultat just nu har formen 10,mantissan. Vi måste därför skifta ned resultatet ett steg, vilket ger $10,000011 >> 1,0000011$. Eftersom vi skiftade ned mantissan måste vi öka exponenten. Exponenten var 11, ökas med 1 och blir 12. Enligt formeln ska exponentbitarna vid uträkning subtraheras med 15, alltså måste vi nu öka 12 med 15, vilket ger 27. Vi går från decimal notation till binärt: $27_{10} = 11011_2$

Både flyttal A och B var positiva, och vårt nya flyttal kommer därmed vara positivt. Därmed har vi samtliga fält klara och kan sätta ihop dem till ett helt flyttal: 0 11011 0000011000.

Subtraktion

Även vid subtraktion måste man se till att exponenterna för de båda talen är likadana. Om de inte är det måste man återigen skifta talen så att exponenterna blir lika stora. I övrigt är skillnaden mot addition att man istället subtraherar mantissorna för de båda talen.

Vi tar 0 11010 0111000000 - 0 11001 1011000000 som exempel. Först beräknar vi exponenten för de båda termerna. Första termen har 11010_2, vilket ger 26_{10}. Enligt formeln ska vi ta minus 15, vilket ger 26-15=11. Andra termen har 11001_2, vilket blir 25_{10}. 25-15=10, och vi kan se att våra två exponenter skiljer sig åt. Den andra termen är mindre, vilket ger att vi måste skifta om den för att få dem på samma exponent.

Skillnaden är ett, så vi skiftar mantissan för andra termen ett steg: 1,1011 >> 0,11011. Exponenten för båda talen är nu 11, och vi har båda mantissorna beräknade. Nästa steg blir att subtrahera de två mantissorna.

	1,	0	1	1	1	0
-	0,	1	1	0	1	1
	0,	1	0	0	1	1

Resultatet är $0,10011_2$, men enligt formeln ska mantissan vara på formen 1,mantissan. Därför måste vi normera mantissan och skiftar ett steg. Eftersom vi skiftar upp mantissan måste vi minska exponenten. Tidigare beräknade vi exponenten till 11, men eftersom vi skiftade upp mantissan ett steg minskar vi exponenten med ett: 11-1=10. Omvandlat till binärt blir $10_{10}=01010_2$.

Båda termerna var positiva, och vårt resultat är också positivt. Därmed blir teckenbiten för det nya talet noll. Om vi sätter ihop samtliga delar får vi flyttalet 0 11001 0011000000.

Multiplikation

Till skillnad från addition och subtraktion behöver man inte se till att exponenterna har samma värde vid multiplikation. Anledningen till det är att man multiplicerar exponenterna och mantissorna var för sig, enligt formeln $(2^x \times a) \times (2^y \times b) = 2^{x+y} \times (a \times b)$. Vi tar 0 10100 1001000000 * 0 10110 0101000000 som exempel.

För första termen har vi exponentvärdet 10100_2, vilket blir 20_{10}. 20-15=5, och vi har då räknat ut exponenten för första termen.

För den andra termen har vi exponentvärdet 10110_2, vilket blir 22_{10}. 22-15=7, vilket ger exponenten för den andra termen. Om vi ser till formeln ovan har vi räknat ut x och y, och kan därmed sätta in värdena direkt i formeln eftersom a och b är mantissorna för de båda talen.

$(2^5 \times 1,1001) \times (2^7 \times 1,0101) = 2^{5+7} \times (1,1001 \times 1,0101)$. 5+7=12, vilket ger exponenten. Mantissan får vi räkna ut med hjälp av ”vanlig” binär multiplikation.

					1,	1	0	0	1
				*	1,	0	1	0	1
~~1~~	~~1~~	~~1~~	~~1~~	~~1~~					
					1	1	0	0	1
				0	0	0	0	0	
			1	1	0	0	1		
		0	0	0	0	0			
+	1	1	0	0	1				
1	0,	0	0	0	0	1	1	0	1

Resultatet av multiplikationen är 10,00001101, men vi ser att vi måste normera mantissan eftersom den inte följer den specificerade formen. Vi skiftar ned mantissan ett steg och måste därför öka exponenten med ett, vilket ger oss exponenten 13. Vi måste lägga på vårt *bias* och får därför $13+15=28_{10}=11100_2$.

För att beräkna teckenbiten adderar vi båda teckenbitarna och tar sedan modulus på resultatet: 0+0=0 → 0 % 2=0. Med den formeln kan man alltid räkna ut den nya teckenbiten, och med den kan vi se att om ett tal är negativt blir resultatet negativt (1+0=1 → 1 % 2=1) och att om man har två negativa tal blir resultatet positivt (1+1=2 → 2 % 2=0).

Eftersom vi nu har räknat ut samtliga delar kan vi sätta ihop dem till ett flyttal och får resultatet 0 11100 0000011010.

6.3 Övningsuppgifter

Uppgift 6.3.1
Omvandla följande tal från 16-bitars flyttal enligt IEEE 754 till det decimala talsystemet

a. 0 10011 1010100000
b. 0 11000 0011110100
c. 1 10010 1101000000
d. 1 01101 1000000000
e. 0 01111 1010000000
f. 0 11111 0000000000
g. 1 00000 0000000000
h. 0 01111 0000000000
i. 0 01010 1000000000

Uppgift 6.3.2
Omvandla följande tal från det decimala talsystemet till 16-bitars flyttal enligt IEEE 754

a. 16
b. 5,25
c. 2057
d. -324
e. -76,5
f. -0,375
g. - Oändligheten
h. $\frac{11}{64} = \frac{1}{8} + \frac{1}{32} + \frac{1}{64} = 0,171875$
i. -1707

Uppgift 6.3.3
Vad blir resultatet av följande tal? Svara med ett 16-bitars flyttal enligt IEEE 754.

a. 0 10001 1110000000 + 0 10001 0100000000
b. 0 11011 0011010100 + 0 11011 0100111010
c. 0 10001 0001000000 + 0 10000 0111000000
d. 0 10101 1010100000 + 0 10001 1000000000
e. 0 00111 0110000000 + 0 01000 0011000000
f. 0 00100 1010110011 + 0 00100 0101111011

Uppgift 6.3.4
Beräkna resultatet av följande tal. Svara med ett 16-bitars flyttal enligt IEEE 754.

a. 0 10001 1110000000 - 0 10001 0100000000
b. 0 11011 1010110100 - 0 11011 0101011010
c. 0 10001 0111000000 - 0 10000 0111000000
d. 0 10101 1010100000 - 0 10001 1000000000
e. 0 01000 0011000000 - 0 00111 0110000000
f. 0 00100 1010110011 - 0 00100 0101011011

Uppgift 6.3.5
Beräkna resultatet av följande tal. Svara med ett 16-bitars flyttal enligt IEEE 754.

a. 0 10001 1110000000 * 0 10001 0100000000
b. 0 01011 1011000000 * 0 01011 0101000000
c. 0 10001 0111000000 * 0 10000 0111000000
d. 1 10101 1010100000 * 0 10001 1100000000
e. 0 01000 0011000000 * 0 00111 0110000000
f. 1 10100 1010000000 * 1 10110 0101000000

7. Maskininstruktioner

För att programmera ett datorsystem behövs ett programmeringsspråk. Det språk som ligger närmast processorn är Assembly, där i stort sett varje instruktion motsvarar en binär maskininstruktion. Nackdelen med Assembly är att det finns en dialekt av språket för varje processorakitektur, eftersom varje arkitektur har olika instruktionsuppsättningar.

I kursen Datorsystem används Nios II-assembler eftersom kursen använder Altera DE2-labbkort i undervisningen. Mer information om Nios II-processorn som finns på DE2-korten och dess instruktionsuppsättning finns i Alteras referenshandbok[1].

I Nios II-assembler finns tre format för instruktionerna: I, R och J. I står för immediate och används för alla instruktioner med två register-operander och en heltalsoperand. R står för register och används för alla instruktioner där samtliga tre operander är register. Det sista formatet är J, där J står för jump och används för de flesta hoppinstruktioner. De tre formaten har varsin bitindelning eftersom de har olika krav på den information som ska finnas i instruktionen. En instruktion av typen R behöver kunna adressera tre register medan en instruktion av typen I behöver kunna adressera två register samtidigt som det finns plats nog för ett immediate-värde.

7.1 Översätta assemblerinstruktioner till maskininstruktioner

För att förstå precis hur en instruktion ser ut när en processor hanterar den kan man översätta assemblerinstruktioner till binära maskininstruktioner. För att kunna göra det behöver man ha tillgång till referenshandboken eftersom den innehåller information om de instruktionsformat som används och hur varje instruktion mappar mot 32-bitarsinstruktioner.

För att se hur man går tillväga tar vi instruktionen `add r8, r9, r10` som exempel. Vi börjar med att slå upp add i handboken, där vi kan se hur instruktionen fungerar, hur syntaxen ser ut och ett exempel på hur man kan använda instruktionen. Längst ned finns också en tabell som visar hur add ska översättas till 32 bitar maskininstruktion. Med hjälp av tabellen, som också ses i tabell 7.1, kan vi se bitindelningen av instruktionen och därmed också se på vilka platser alla delar ska placeras.

31 30 29 28 27	**26 25 24 23 22**	**21 20 19 18 17**	**16 15 14 13 12 11**	**10 9 8 7 6**	**5 4 3 2 1 0**
A	B	C	OPX = 0x31	0	OP = 0x3A

Tabell 7.1: Bitindelningen av instruktionen add

Utifrån instruktionen, `add r8, r9, r10`, kan vi hämta de tre operanderna som också är de tre okända fälten A, B och C. Vi måste börja med att titta på hur syntaxen för instruktionen ser ut eftersom det inte är säkert att registrena A, B och C kommer i den ordningen i assemblerinstruktionen. Syntaxen finns i handboken och är `add rC, rA, rB`. Utifrån `add r8, r9, r10` kan vi se att A = r9, B = r10 och C = r8. Det som återstår är att sätta in registren i maskinstruktionen, och det görs genom att man sätter in registrets nummer binärt på rätt plats. Exempelvis får vi då r8 = 01000_2, r9 = 01001_2 och r10 = 01010_2.

Eftersom instruktionens OP- och eventuell OPX-kod ges av tabellen längst ned i informationen om instruktionen omvandlar vi bara de delarna från hexadecimalt till binärt och kan se hela instruktionen i tabell 7.2

[1] `http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf`

31 30 29 28 27					**26 25 24 23 22**					**21 20 19 18 17**					**16 15 14 13 12 11**						**10 9 8 7 6**					**5 4 3 2 1 0**					
A					B					C					OPX						0					OP					
0	1	0	0	0	0	1	0	1	0	0	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	1	1	1	0	1	0

Tabell 7.2: Binär representation av operationen `add r8, r9, r10`

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A	B	IMM16	OP = 0x04

Tabell 7.3: Bitindelningen av instruktionen addi

Instruktionen add är av typen R, vilket ger instruktionsindelningen ovan. Om vi däremot tar instruktionen addi som exempel kan vi se i tabell 7.3 att indelningen är en annan eftersom ett annat instruktionsformat används.

Utifrån bitindelningen av R-instruktioner kan vi se att det heltalsvärde som används som operand som mest kan bestå av 16 bitar. Det ger att man som högst kan skicka in värden mellan 0 och 65535 om man använder enbart positiva heltal (unsigned) eller mellan –32768 och 32767 om man använder en teckenbit (signed).

7.2 Pseudoinstruktioner

Det finns ett flertal instruktioner i Nios II-assembler som inte har egna OP-koder utan istället översätts till andra instruktioner. Till exempel finns flera instruktioner för att jämföra villkor, bland andra branch if equals(beq), branch if greater than(bgt) och branch if less than(blt). Om man ska jämföra två tal och se om det ena är större än det andra kan man antingen säga ”Om A är större än B” eller ”Om B är mindre än A”. Resultatet blir likadant oavsett vilken metod man använder, och på precis samma sätt finns flera instruktioner för som man använder när man programmerar Assembly än det egentligen finns maskininstruktioner. Om vi tittar i tabellen Assembler Pseudo-Instructions i handboken kan vi se samtliga pseudoinstruktioner och vilken instruktion de översätts till. Till exempel kan man se att instruktionen `subi rB, rA, IMMED` implementeras som `addi rB, rA, (-IMMED)`. Subtraktion är alltså addition med negerad andra term, precis som vid subtraktion av binära tal.

För att se hur en pseudoinstruktion översätts till maskininstruktioner kan vi ta `movi r8, 0x4F` som exempel. I handboken slår vi upp instruktionen movi under Instruction Set Reference för att se hur den implementeras. Där står att `movi rB, IMMED` implementeras som `addi rB, r0, IMMED`. Det betyder att vi får hoppa till informationen om addi och utgå från bitindelningen där när vi assemblerar instruktionen. Bitindelningen för addi finns i tabell 7.3 och med hjälp av den kan vi assemblera `movi r8, 0x4F`. OP-koden blir 0x04 = 0b000100 eftersom det är OP-koden för addi, A = r0 = 00000_2, B = r8 = 01000_2 och IMM16 = 0x4F = 0000 0000 0100 1111_2 eftersom vi måste padda så att det blir 16 bitar. I tabell 7.4 kan vi se den binära representationen av `movi r8, 0x4F`.

Movia

Instruktionen movia är speciell eftersom den inte översätts till en annan assemblerinstruktion utan två. I handboken kan man se att `movia rB, label` implementeras som `orhi rB, r0, %hiadj(label)` och `addi rB, rB, %lo(label)`. Anledningen till att det behövs två instruktioner för movia är att movia flyttar in en 32-bitars adress i ett register, men det finns inga enskilda instruktioner som kan ta emot ett IMM32 eftersom hela instruktionen består av 32 bitar så det görs i två steg.

%hiadj och %lo är makron för att hämta ut olika delar av ett 32-bitars register. I handboken finns en beskrivning av de fyra makron som används av olika instruktioner. Följande står om %lo(immed32): ”Extract bits [15..0] of immed32”. Det betyder att makrot %lo hämtar ut de 16 minst signifikanta bitarna ur ett 32-bitars heltal. Om %hiadj(immed32) står det ”Extract bits [31..16] and adds bit 15 of immed32”, vilket innebär att makrot hämtar de 16 mest signifikanta bitarna samt adderar med värdet av den femtonde biten.

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A	B	IMM16	OP = 0x04
0 0 0 0 0	0 1 0 0 0	0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1	0 0 0 1 0 0

Tabell 7.4: Binär representation av instruktionen movi

Om man vill översätta movia till binära maskinstruktioner måste man börja med att sätta in operanderna från movia i de två instruktionerna som movia översätts till och dessutom använda makrona så att rätt IMM16 sätts som operand för varje instruktion. Efter det kan man utgå från handbokens bitindelning för varje instruktion och sätta in bitarna på rätt plats.

7.3 Övningsuppgifter

Uppgift 7.3.1
I Nios II-assembler finns tre instruktionsformat. Vilka är formaten och hur ser bitindelningen ut?

Uppgift 7.3.2
Översätt instruktionerna nedan till maskinkod. Utgå från följande värden när du gör uppgifterna: write_char = 0xB31A, set_leds = 0x2B34CE7. Alla register översätts som det nummer det har, till exempel r8=0x8, r12=0xC och så vidare.

a. `addi r8, r9, 0x35`
b. `muli r8, r9, 0x77`
c. `stw r8, 4(r9)`
d. `and r6, r7, r8`
e. `bne r6, r7, write_char`
f. `call set_leds`
g. `mov r8, r9`
h. `movia r20, 0xDECA8E00`

Uppgift 7.3.3
Implementera följande i Nios II-assembler:

a. Spara 0x5 i ett register.
b. Spara 0x10 i ett register, 0x20 i ett annat register. Addera de båda registren och spara värdet i ett tredje register.
c. Hoppa till labeln *light_leds*.
d. Hoppa till labeln *light_leds* om register r20 är noll.
e. Kopiera värdet i register r9 till register r10.
f. Maska ut värdet av den sjunde biten i r8 till register r12.
g. Kontrollera om värdet i den nionde biten i r8 är noll, och hoppa till *green_leds* om den är det.
h. Om register r5 är 5, hoppa till labeln *write_char*, kopiera annars värdet i r5 till r9.
i. Hämta ordet som sparas på minnesadressen som pekas ut av r10 till register r14.
j. Skriv värdet från register r15 till den minnesadress som pekas ut av r8.
k. Hämta den byte som sparas på minnesadressen som pekas ut av r10 + 4 till r17. Minnesadressen går till en IO-enhet.
l. Hämta ordet som sparas på minnesadressen som pekas ut av r8 till r10, spara sedan värdet i r10 till den minnesplats som pekas ut av r9. Upprepa tills programmet stängs av.

Uppgift 7.3.4
Implementera följande program med hjälp av JNiosEmu

a. Tänd LED 2-4 för de gröna LEDarna.
b. Läs in status för slider/dip-switcharna och tänd motsvarande grön LED för de switchar som är på. Låt programmet loopa tills det stängs av.

8. Minne

8.1 Övningsuppgifter

Uppgift 8.1.1
Ett givet cacheminne har plats för 64 bytes totalt, där varje rad innehåller 16 bytes.

a. Dela in en minnesadress på 32 bitar i följande delar: Tag, Rad och Byte.

b. Dela in en minnesadress på 16 bitar i följande delar: Tag, Rad och Byte

Uppgift 8.1.2
Ett cacheminne har plats för totalt 128 bytes och varje rad är 16 bytes lång. Minnet är **direktmappat** och en adress är 16 bitar lång. Förutsatt att cacheminnet är tomt från början, bestäm för varje instruktion nedan om det blir en cacheträff eller en cachemiss. Du kan också anta att instruktionerna sker sekventiellt och att data som läggs till i en uppgift finns kvar till nästa.

a. `movi r8, 0xBC40`

b. `ldw r10, 0(r8)`

c. `ldw r11, 4(r8)`

d. `stw r10, 32(r8)`

e. `ldw r10, 40(r8)`

Uppgift 8.1.3
Ett cacheminne har plats för totalt 128 bytes och varje rad är 16 bytes lång. Minnet är **2-vägsassociativt** och en adress är 16 bitar lång. Förutsatt att cacheminnet är tomt från början, bestäm för varje instruktion nedan om det blir en cacheträff eller en cachemiss. Du kan också anta att instruktionerna sker sekventiellt och att data som läggs till i en uppgift finns kvar till nästa.

a. `movi r8, 0x7A50`

b. `ldw r10, 0(r8)`

c. `ldw r11, 16(r8)`

d. `stw r10, 168(r8)`

e. `ldw r10, 44(r8)`

Uppgift 8.1.4
Ett cacheminne har plats för totalt 64 bytes och varje rad är 8 bytes lång. Minnet är **associativt** och en adress är 16 bitar lång. Förutsatt att cacheminnet är tomt från början, bestäm för varje instruktion nedan om det blir en cacheträff eller en cachemiss. Du kan också anta att instruktionerna sker sekventiellt och att data som läggs till i en uppgift finns kvar till nästa.

a. `movi r8, 0x3348`

b. `ldw r10, 0(r8)`

c. `ldw r11, 10(r8)`

d. `stw r10, 16(r8)`

e. `ldw r10, 4(r8)`

Uppgift 8.1.5
Ett cacheminne har plats för totalt 384 bytes och varje rad är 32 bytes lång. Minnet är **3-vägsassociativt** och en adress är 32 bitar lång. Förutsatt att cacheminnet är tomt från början, bestäm för varje instruktion nedan om det blir en cacheträff eller en cachemiss. Du kan också anta att instruktionerna sker sekventiellt och att data som läggs till i en uppgift finns kvar till nästa.

a. `movi r8, 0x12127400`

b. `ldw r10, 0(r8)`

c. `ldw r11, 37(r8)`
d. `stw r10, 148(r8)`
e. `ldw r10, 154(r8)`
f. `ldw r10, 60(r8)`
g. `ldw r11, 28(r8)`
h. `stw r10, -108(r8)`
i. `ldw r10, 4(r8)`

Uppgift 8.1.6
Ett cacheminne har plats för totalt 256 bytes och varje rad är 8 bytes lång. Minnet är **direktmappat** och en adress är 32 bitar lång. Förutsatt att cacheminnet är tomt från början, bestäm för varje instruktion nedan om det blir en cacheträff eller en cachemiss. Du kan också anta att instruktionerna sker sekventiellt och att data som läggs till i en uppgift finns kvar till nästa.

a. `movi r8, 0xBEDA12C4`
b. `ldw r10, 0(r8)`
c. `ldw r11, 8(r8)`
d. `stw r10, 16(r8)`
e. `ldw r10, 32(r8)`
f. `ldw r10, 40(r8)`
g. `ldw r11, 24(r8)`
h. `stw r10, 4(r8)`
i. `ldw r10, 64(r8)`

9. Facit

Lösning till 2.1.1

a. En bit är den minsta enheten för att representera ett tal eller en logisk enhet.

b. 8 bitar

Lösning till 2.1.2

a. $2^{10} = 1024$ bytes

b. 1024 kB = $2^{20} = 1024^2 =$ 1 048 576 bytes

c. 1024 MB = $2^{30} = 1024^3 =$ 1 073 741 824 bytes

d. 1024 GB = $2^{40} = 1024^4 =$ 1 099 511 627 776 bytes

e. 3*1024 = 3072 bytes

f. 4*1024 kB = $4 * 1024^2 = 4$ 194 304 bytes

g. 2,5*1024 MB = $2,5 * 1024^3$ = 2 684 354 560 bytes

h. 5*1024 kB = $5 * 1024^2 = 5$ 242 880 bytes

i. 0,5*1024 MB = $0,5 * 1024^3$ = 536 870 912 bytes

Lösning till 2.1.3

•Moderkort

•Arbetsminne (primärminne, också kallat RAM-minne)

•Hårddisk (sekundärminne)

•Processor

•Nätaggregat

•Chassi

•Grafikkort*

* Om man inte är ute efter en speldator kan man ofta köpa ett moderkort med integrerat grafikkort. Moderkortet har i regel också nätverks- och ljudkort.

Lösning till 2.1.4
1947

Lösning till 3.5.1

a. 3_{10}

b. 5_{10}

c. 15_{10}

d. 16_{10}

e. 117_{10}

f. 11_{10}

g. 29_{10}

h. 154_{10}

i. 243_{10}

j. 278_{10}

k. 512_{10}

l. 616_{10}

Lösning till 3.5.2

a. 10_2

b. 1101_2

c. 11 0100_2

d. 1010 1100_2

e. 1 1010_2

f. 10 0011_2

g. 11 0001_2

h. 100_2

i. 100 1101_2

j. 1000 1111_2

k. 1100 1000_2

l. 10 0010 1100_2

Lösning till 3.5.3

a. A_{16}
b. $9F_{16}$
c. 8_{16}
d. $7A3_{16}$
e. E_{16}
f. $5C_{16}$
g. 77_{16}
h. 999_{16}
i. $B0_{16}$
j. $9F_{16}$
k. CA_{16}
l. 256_{16}

Lösning till 3.5.4

a. 11_2
b. $0001\ 0011_2$
c. $0001\ 0010\ 0111_2$
d. $0101\ 1011\ 0011\ 1010_2$
e. $0010\ 1010_2$
f. $0110\ 0000_2$
g. $0001\ 0010\ 0000_2$
h. $0011\ 0011\ 0000_2$
i. $0101\ 0111\ 0010_2$
j. $0001\ 0010\ 1010\ 1011_2$
k. $1111\ 0011\ 1111\ 0100_2$
l. $0111\ 1010\ 0110\ 1001_2$

Lösning till 3.5.5

a. 43_{10}
b. 12_{10}
c. 187_{10}
d. 794_{10}
e. 39_{10}
f. 61_{10}
g. 828_{10}
h. 5166_{10}
i. 48879_{10}
j. 64222_{10}
k. 57005_{10}
l. 47806_{10}

Lösning till 3.5.6

a. $4D_{16}$
b. FF_{16}
c. $A0C_{16}$
d. D_{16}
e. 539_{16}
f. 808_{16}
g. CAB_{16}
h. $2DE_{16}$
i. $FACE_{16}$
j. $CAFE_{16}$
k. $BEAD_{16}$
l. $DECAF_{16}$

Lösning till 3.5.7

a. 5_8
b. 31_8
c. 574_8
d. 7_8
e. 4_8
f. 70_8
g. 15_8
h. 406_8
i. 55_8
j. 640_8
k. 777_8
l. 3341_8

Lösning till 3.5.8

a. 20_5
b. 32_5
c. 1000_5
d. 2322_5
e. 100_5
f. 202_5
g. 302_5
h. 2011_5
i. 313_5
j. 340_5
k. 1010_5
l. 4030_5

Lösning till 3.5.9

a. $1{,}5_{10}$
b. $2{,}25_{10}$
c. $1{,}75_{10}$
d. $0{,}6875_{10}$
e. $10{,}125_{10}$
f. $10{,}6875_{10}$
g. $1{,}9375_{10}$
h. $0{,}00390625_{10}$
i. $1{,}125_{10}$
j. $8{,}015625_{10}$
k. $1{,}140625_{10}$
l. $0{,}001953125_{10}$

Lösning till 4.1.1

a. OCH (AND), A·B
b. ICKE (NOT), $\overline{A}$
c. ELLER (OR), A+B
d. EXKLUSIVE ELLER (XOR), A$\bigoplus$B

Lösning till 4.1.2

a.

Indata		Utdata
A	B	A OCH B
0	0	0
0	1	0
1	0	0
1	1	1

b.

Indata		Utdata
A	B	A ELLER B
0	0	0
0	1	1
1	0	1
1	1	1

c.

Indata		Utdata
A	B	A OCH B
0	0	0
0	1	1
1	0	1
1	1	0

d.

Indata	Utdata
A	ICKE A
0	1
1	0

Lösning till 4.1.3

a. 10
b. 101
c. 000
d. 1010 0000
e. 1000 0000
f. 1010 0000
g. 0111 1001
h. 0010 0010
i. 1010 0100

Lösning till 4.1.4

a. 11
b. 111
c. 111
d. 1110 1011
e. 1001 1011
f. 1111 1010
g. 1111 1111
h. 1011 1011
i. 1110 1111

Lösning till 4.1.5

a. 01
b. 010
c. 111
d. 0100 1011
e. 0001 1011
f. 0101 1010
g. 1000 0110
h. 1001 1001
i. 0100 1011

Lösning till 4.1.6

a. 01
b. 000
c. 111
d. 0101 0110
e. 0110 0100
f. 0000 1111

Lösning till 4.1.7
Man skapar en bitmask med alla bitar nollställda utom den femte biten, sedan utförs den logiska operationen OCH på den ursprungliga bitsträngen och bitmasken. Om resultatet är större än noll var den femte biten en etta, annars var den en nolla.

Exempel där femte biten är 1:

	1	1	0	1	0	1	0	1
$\cdot$	0	0	0	1	0	0	0	0
=	0	0	0	1	0	0	0	0

Exempel där femte biten är noll:

	1	1	1	0	1	0	0	1
$\cdot$	0	0	0	1	0	0	0	0
=	0	0	0	0	0	0	0	0

Lösning till 4.1.8
Man skapar en bitmask med alla bitar nollställda utom den femte biten som man istället sätter till ett. Sedan utförs den logiska operationen ELLER på den ursprungliga bitsträngen och bitmasken. Alla övriga bitar kommer behålla sitt ursprungliga värde medan den femte biten kommer ändras till 1.

Exempel:

	1	1	1	0	1	0	0	1
+	0	0	0	1	0	0	0	0
=	1	1	1	1	1	0	0	1

Lösning till 4.1.9
Man skapar en bitmask med alla bitar ettställda utom den femte biten, som man sätter till noll. Sedan utförs den logiska operationen OCH på den ursprungliga bitsträngen och bitmasken. Alla övriga bitar kommer behålla sitt ursprungliga värde medan den femte biten kommer ändras till 1.

Exempel:

	1	1	1	1	1	0	0	1
$\cdot$	1	1	1	0	1	1	1	1
=	1	1	1	0	1	0	0	1

Lösning till 4.1.10
Man skapar en bitmask med alla bitar nollställda utom den femte biten, som man sätter till ett. Sedan utförs den logiska operationen XOR på den ursprungliga bitsträngen och bitmasken. Alla övriga bitar kommer behålla sitt ursprungliga värde medan den femte biten kommer ändras från sitt nuvarande läge till det motsatta.

Exempel:

	1	1	1	1	1	0	0	1
$\cdot$	0	0	0	1	0	0	0	0
=	1	1	1	0	1	0	0	1

Lösning till 5.4.1

a. 10_2
b. 1_2
c. 1_2
d. 100_2
e. 1111_2
f. 11000_2
g. 10100_2
h. 10000_2
i. $1\ 0101\ 0100_2$
j. $1\ 1011\ 1110_2$
k. $1101\ 0110_2$
l. $1\ 1111\ 1000_2$

Lösning till 5.4.2

a. 0_2
b. 1_2
c. 1_2
d. 10_2
e. 101_2
f. 1_2
g. 100_2
h. 1101_2
i. $11\ 1110_2$
j. $100\ 0000_2$
k. 10110_2
l. $101\ 1100_2$

Lösning till 5.4.3

a. 1_2
b. 10_2
c. 0_2
d. 1001_2
e. 110010_2
f. $1001\ 1100_2$
g. $1010\ 0101_2$
h. 1010_2
i. $110\ 1101\ 0010\ 0011_2$
j. $101\ 1001\ 1100\ 1011_2$
k. $1\ 1001\ 1000_2$
l. $1001\ 1010\ 0100\ 1100_2$

Lösning till 5.4.4

a. $11{,}1_2$
b. 11_2
c. $100{,}01_2$
d. $10111{,}01_2$
e. $1{,}1_2$
f. 11_2
g. $10{,}11_2$
h. $1000{,}1_2$
i. $10001{,}1_2$
j. $110{,}11_2$
k. $10101{,}01_2$
l. $10\ 0011{,}0111_2$

Lösning till 6.3.1

a. 26,5
b. 634
c. -14,5
d. -0,375
e. 1,625
f. ∞ (oändligheten)
g. -0
h. 1
i. $\frac{1}{32} + \frac{1}{64} = \frac{3}{64} = 0,046875$

Lösning till 6.3.2

a. 0 10011 0000000000
b. 0 10001 0101000000
c. 0 11010 0000000100
d. 1 10111 0100010000
e. 1 10101 0011001000
f. 1 01101 1000000000
g. 1 11111 0000000000
h. 0 01100 0110000000
i. 1 11001 1010101011

Lösning till 6.3.3

a. 0 10010 1001000000
b. 0 11100 0100000111
c. 0 10001 1100100000
d. 0 10101 1100000000
e. 0 01000 1110000000
f. 0 00101 1000010111

Lösning till 6.3.4

a. 0 10000 0100000000
b. 0 11001 0101101000
c. 0 10000 0111000000
d. 0 10101 1001000000
e. 0 00111 0000000000
f. 0 00010 0101100000

Lösning till 6.3.5

a. 0 10100 0010110000
b. 0 01000 0001101110
c. 0 10011 0000100010
d. 1 11000 0111001100
e. 0 00000 1010001000
f. 0 11100 0001000100

Lösning till 7.3.1

I-formatet:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6	5 4 3 2 1 0
A | B | IMM16	OP

R-formatet:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6	5 4 3 2 1 0
A | B | C | OPX	OP

J-formatet:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6	5 4 3 2 1 0
IMM26	OP

Lösning till 7.3.2

a.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																OP = 0x04					
0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	0	0	0	1	0	0

b.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																OP = 0x24					
0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	1	1	0	0	1	0	0

c.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																OP = 0x15					
0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1

d.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					C					OPX = 0x0E						0					OP = 0x3A					
0	0	1	1	1	0	1	0	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	1	0	1	0

e.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																OP = 0x1E					
0	0	1	1	0	0	0	1	1	1	1	0	1	1	0	0	1	1	0	0	0	1	1	0	1	0	0	1	1	1	1	0

f.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM26																										OP = 0x0					
1	0	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	0

g.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					C					OPX = 0x31						0					OP = 0x3A					
0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	1	1	1	0	1	0

h.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																OP = 0x34					
0	0	0	0	0	1	0	1	0	0	1	1	0	1	1	1	1	0	1	1	0	0	1	0	1	1	1	1	0	1	0	0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																OP = 0x04					
1	0	1	0	0	1	0	1	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Lösning till 7.3.3

a.
```
movi    r8, 0x5
```

b.
```
movi    r8, 0x10
movi    r9, 0x20
add     r10, r8, r9
```

c.
```
br  light_leds
```

d.
```
beq     r20, r0, light_leds
```

e.
```
mov     r10, r9
```

f.
```
# Create a bitmask
movi    r10, 0b1000000
# Mask out seventh bit
and r12, r8, r10
```

g.
```
movi    r10, 0b100000000
and     r12, r8, r10
beq     r12, r0, light_leds
```

h.
```
movi    r8, 5
beq     r5, r8, write_char
mov     r9, r5
```

i.
```
ldw     r14, 0(r10)
```

j.
```
stw     r15, 0(r8)
```

k.
```
ldbio   r17, 4(r10)
```

l.
```
top_of_loop:
    ldw     r10, 0(r8)
    stw     r10, 0(r9)
    br      top_of_loop
```

Lösning till 7.3.4

a.
```
movia   r8, 0x10000010
movi    r9, 0b11100
# Using stbio instead of stb since we're interacting with an I/O-unit.
stbio   r9, 0(r8)
```

b.
```
    movia   r8, 0x10000010      # Green LEDs
    movia   r9, 0x10000040      # Slider swit
top_of_loop:
    ldbio   r10, 0(r9)
    stbio   r10, 0(r8)
    br      top_of_loop
```

Lösning till 8.1.1

a.

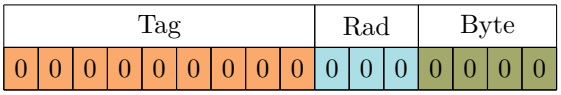
Tag																										Rad		Byte			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

b.

Tag										Rad		Byte			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Lösning till 8.1.2

a. Påverkar inte cacheminnet, men bitindelningen av adressen är följande:



b. Eftersom de minst signifikanta bitarna tillsamans bestämmer rad och byte börjar vi att kolla de delarna: $40_{16} = 0 \mid 100 \mid 0000_2$. Det ger att raden 100, vilken är tom eftersom cacheminnet är tomt. Det ger en **cachemiss** och vi lägger in hela blocket 0xBC40 - 0xBC4F på rad 100.

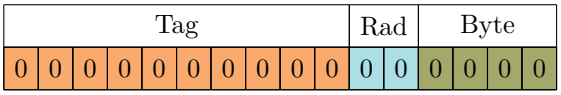
c. Nästa minnesadress som används är 0xBC44, vilket ger $44_{16} = 0 \mid 100 \mid 0100_2$. Raden är återigen 100, och vi måste kolla om taggarna stämmer överens. Båda har taggen 0xBC, och vi har därmed en **cacheträff**.

d. Härnäst används adressen 0xBC60, vilket ger $60_{16} = 0 \mid 110 \mid 0000_2$. Ur det har vi att raden är 110, som är tom. Vi får en **cachemiss** och lägger in 0xBC60 - 0xBC6F på raden.

e. Sista instruktionen hämtar från 0xBC68. $68_{16} = 0 \mid 110 \mid 1000_2$, vilket ger raden 110. Där finns redan data, och med hjälp av taggen ser vi att det är rätt data på raden och får en **cacheträff**.

Lösning till 8.1.3

a. Påverkar inte cacheminnet, men bitindelningen av adressen är följande:



b. Från 0x7A50 får vi $50_{16} = 01 \mid 01 \mid 0000_2$. Det ger raden 01, vilken är tom eftersom cacheminnet är tomt. Det ger en **cachemiss** och vi lägger in hela blocket 0x7A50 - 0x7A5F på rad 01.

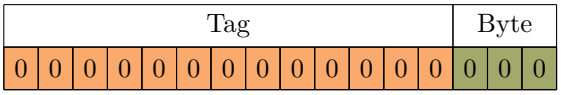
c. Nästa minnesadress som används är 0x7A60, vilket ger $60_{16} = 01 \mid 10 \mid 0000_2$. Raden är 10 och den är tom, vilket ger en **cachemiss**. På raden läggs 0x7A60 - 0x7A6F in.

d. Härnäst används adressen 0x7AF8, vilket ger $F8_{16} = 11 \mid 11 \mid 1000_2$. Ur det har vi att raden är 11, som är tom. Vi får en **cachemiss** och lägger in 0x7AF0 - 0x7AFF på raden.

e. Sista instruktionen hämtar från 0x7A7C. $7C_{16} = 01 \mid 11 \mid 1100_2$, vilket ger raden 11. Där finns redan data, och med hjälp av taggen ser vi att det är fel data på raden och vi får en **cachetmiss**. På andra mängden i rad 11 lägger vi in 0x7A70 - 0x7A7F.

Lösning till 8.1.4

a. Påverkar inte cacheminnet, men bitindelningen av adressen är följande:



b. Cacheminnet är tomt, vilket ger en **cachemiss**. På första raden lägger vi in 0x3348 - 0x334F.

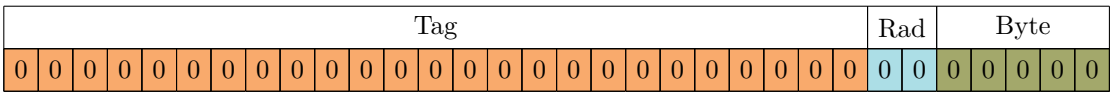
c. Nästa minnesadress som används är 0x3352. Vi ser med hjälp av taggen att det data vi söker inte finns i cacheminnet och får en **cachemiss**. 0x3350 - 0x3357 läggs in på andra raden i minnet.

d. Härnäst används adressen 0x3358, och vi ser återigen utifrån taggen att sökt data inte finns i cachen. Det blir en cachemiss och 0x3358 till 0x335F läggs in på tredje raden.

e. Sista instruktionen hämtar från 0x334C. Med hjälp av taggen kan vi se att första raden innehåller sökt data och får därmed en **cacheträff**.

Lösning till 8.1.5

a. Påverkar inte cacheminnet, men bitindelningen av adressen är följande:



b. Första anropet går till 0x12127400 och ur det går vi $00_{16} = 0 \mid 00 \mid 0\ 0000_2$. Eftersom cacheminnet är tomt får vi en **miss** och lägger in 0x12127400 - 0x1212741F på rad 00.

c. Härnäst används adressen 0x12127425. Ur $25_{16} = 0 \mid 01 \mid 0\ 0101_2$ får vi raden 01. Raden är tom, vilket ger en **cachemiss**, och 0x12127420 - 0x1212743F läggs in på raden.

d. Nästa instruktion skriver till 0x12127494. Ur $94_{16} = 1 \mid 00 \mid 1\ 0100_2$ får vi raden 00. På rad 00 finns redan data i första mängden, men vi kan se att taggen inte stämmer överens och vi får en **cachemiss**. I andra mängden på rad 00 läggs 0x12127480 - 0x1212749F in.

e. Adressen är 0x1212749A och vi får $9A_{16} = 1 \mid 00 \mid 1\ 1010_2$, vilket ger rad 00. Utifrån taggen kan vi se att vårt sökta data finns i andra mängden på rad 00 och vi får en **cacheträff**.

f. Nästa minnesadress som används är 0x1212743C. $3C_{16} = 0 \mid 01 \mid 1\ 1100_2$, vilket ger raden 01. Där finns redan data och vi jämför taggarna för att se om det är rätt data, vilket det är. Vi har därmed en **cacheträff**.

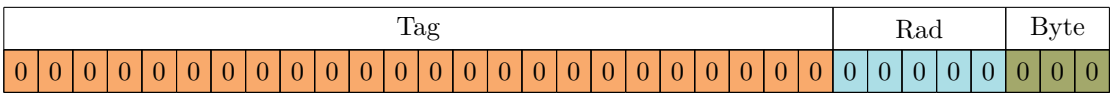
g. Härnäst används adressen 0x1212741C, vilket ger $1C_{16} = 0 \mid 00 \mid 1\ 1100_2$. På rad 00 finns data i både mängd 1 och 2, och vi får jämföra taggarna, vilket ger en **träff** i första mängden.

h. Nästa instruktion skriver till 0x12127394. $94_{16} = 1 \mid 00 \mid 1\ 0100_2$. Återigen får vi rad 00 och måste jämföra taggarna. Taggarna för det data som ligger på rad 00 är 0x121274 och 0x1212748, medan den sökta taggen är 0x1212738, och vi får därmed en **cachemiss**. I tredje mängden på rad 00 lägger vi in 0x12127380 till 0x1212739F.

i. Sista instruktionen hämtar från 0x12127404, vilket ger $04_{16} = 0 \mid 00 \mid 0\ 0100_2$. Det ger rad 00, och utfrån taggarna ser vi att data redan finns i första mängden på raden och vi får en **cacheträff**.

Lösning till 8.1.6

a. Påverkar inte cacheminnet, men bitindelningen av adressen är följande:



b. Cacheminnet är tomt → **cachemiss**. 0xBEDA12C4 → $C4_{16} = \mid 1100\ 0 \mid 100_2$ → 0xBEDA12C0 - 0xBEDA12C7 på rad 11000.

c. 0xBEDA12CC → $CC_{16} = \mid 1100\ 1 \mid 100_2$ → Tom rad = **cachemiss** → 0xBEDA12C8 - 0xBEDA12CF på rad 11001.

d. 0xBEDA12D4 → $D4_{16} = \mid 1101\ 0 \mid 100_2$ → Tom rad = **cachemiss** → 0xBEDA12D0 - 0xBEDA12D7 på rad 11010.

e. 0xBEDA12E4 → $E4_{16} = \mid 1110\ 0 \mid 100_2$ → Tom rad = **cachemiss** → 0xBEDA12E0 - 0xBEDA12E7 på rad 11100.

f. 0xBEDA12EC → $EC_{16} = \mid 1110\ 1 \mid 100_2$ → Tom rad = **cachemiss** → 0xBEDA12E8 - 0xBEDA12EF på rad 11101.

g. 0xBEDA12DC → $DC_{16} = \mid 1101\ 1 \mid 100_2$ → Tom rad = **cachemiss** → 0xBEDA12D8 - 0xBEDA12DF på rad 11011.

h. 0xBEDA12C8 → $C8_{16} = \mid 1100\ 1 \mid 000_2$ → Tag på rad 11001 stämmer = **cacheträff**.

i. 0xBEDA1304 → $04_{16} = \mid 0000\ 0 \mid 100_2$ → Tom rad = **cachemiss** → 0xBEDA1300 - 0xBEDA1307 på rad 00000.