Övning 2

Grindar

Transistorn är ett datorsystems grundstenar, och en transistor kan användas för att avgöra om en ledning ska vara strömförande eller inte. Det gör att man med hjälp av transistorer kan bygga grindar som ger olika utsignaler baserat på de signaler man skickar in.

AND (OCH)

En OCH-grind ger en etta som utsignal när båda insignalerna är ettor. Man brukar säga att en OCH-grind är sann när båda invärdena är sanna, och en tabell för utsignaler baserat på invärden kallas sanningstabell.

Europeisk symbol för AND

Amerikansk symbol för AND

Matematisk notation

A*B

Sanningstabell för AND (OCH):

A	**B**	**Ut**
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise AND

När man har en bitvis (bitwise) jämförelse jämför man varje bitposition i två tal med varandra och får ett resultat där varje bitposition är utsignalen för de två jämförda talen.

A	1	1	0	0	0	1	0	1
B	1	0	0	0	1	1	1	0
Ut	1	0	0	0	0	1	0	0

Logiskt AND

Istället för att jämföra varje bit kan man också jämföra två hela bitsträngar med varandra. Då tittar man på om hela bitsträngen är skild från noll, vilket ger en etta som invärde, eller om bitsträngen bara innehåller nollor, vilket då ger noll som invärde.

Så länge båda bitsträngarna är skilda från noll kommer vi ha en etta som utvärde:

A	0	0	1	1	1	0	1	0
B	1	1	1	0	0	0	0	1
Ut	0	0	0	0	0	0	0	1

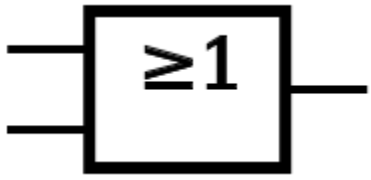
Om vi däremot har en bitsträng med bara nollor som invärde kan inte både A och B vara sanna, och därmed blir utvärdet falskt:

A	1	1	1	1	1	1	1	1
B	0	0	0	0	0	0	0	0
Ut	0	0	0	0	0	0	0	0

OR (ELLER)

En OR-grind ger en etta som utvärde så länge något av invärdena är skilt från noll.

Europeisk symbol för OR



Amerikansk symbol för OR

Matematisk notation

A+B

Sanningstabell för OR (ELLER):

A	**B**	**Ut**
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise OR

Precis som för en AND-grind kan man tillämpa OR på varje del av två bitsträngar. Då får man

en resulterande bitsträng som innehåller resultatet av OR för varje position i de två bitsträngarna som är invärden.

A	1	0	0	1	1	1	0	0
B	1	1	0	1	0	0	1	0
Ut	1	1	0	1	1	1	1	0

Logiskt OR

Även OR går att tillämpa på hela bitsträngar, där då minst en av bitsträngarna måste vara skild från noll för att utvärdet ska vara sant.

Logiskt ELLER där båda bitsträngarna är skilda från noll:

A	1	0	0	0	1	0	0	1
B	0	1	1	0	0	1	0	0
Ut	0	0	0	0	0	0	0	1

Logiskt ELLER där en av bitsträngarna är skild från noll:

A	0	0	0	0	0	0	0	0
B	0	0	1	1	1	0	0	0
Ut	0	0	0	0	0	0	0	1

A	1	0	1	0	1	0	1	0
B	0	0	0	0	0	0	0	0
Ut	0	0	0	0	0	0	0	1

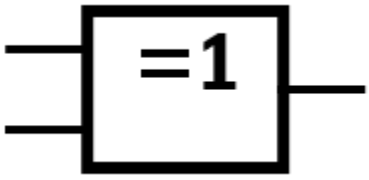
Logiskt ELLER där båda bitsträngarna är noll:

A	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0
Ut	0	0	0	0	0	0	0	0

XOR (Exclusive OR)

En XOR-grind är sann när ett och endast ett av invärdena är sant.

Europeisk symbol för XOR



Amerikansk symbol för XOR

Matematisk notation

$A \oplus B$

Sanningstabell för XOR

A	**B**	**Ut**
0	0	0
0	1	1
1	0	1
1	1	0

Ett exempel på bitvis XOR:

A	1	1	0	0	0	1	0	1
B	1	1	1	0	0	0	1	0
Ut	0	0	1	0	0	1	1	1

Det går naturligtvis också att använda logisk XOR:

Både A och B skilda från noll:

A	1	1	1	0	0	1	1	1
B	1	0	1	0	1	0	1	0
Ut	0	0	0	0	0	0	0	0

A = 0, B skilt från noll:

A	0	0	0	0	0	0	0	0
B	1	1	0	0	1	1	0	0
Ut	0	0	0	0	0	0	0	1

Maskning

När man arbetar med bitsträngar kan man ibland vara intresserad av att veta värdet i en av strängens bitar, eller att ändra värdet i bara en av bitarna. Man kan då använda en bitmask tillsammans med en av de bitvisa operationerna AND, OR och XOR beroende på vilket resultat man vill åstadkomma.

Att sätta en bit till 1

Om man har en bitsträng och vill ändra en specifik bit till 1 kan man använda OR. Om man tittar på sanningstabellen för OR ser man att så länge invärdena är skilda från noll kommer resultatet att vara 1. Det ger att vi kan skapa en bitmask med 0 på alla platser utom de vi vill ändra till 1, eftersom x OR 0 kommer ger x och vi ändrar därmed inte på övriga bitar.

Vi ändrar biten på position 4 från 0 till 1 med hjälp av en bitmask och OR:

Position	7	6	5	4	3	2	1	0
In	1	1	0	**0**	0	1	0	1
Bitmask	0	0	0	**1**	0	1	0	0
Ut	1	1	0	**1**	0	1	0	1

Att sätta en bit till 0

Om man istället vill sätta en specifik bit till 0 kan man använda AND. I sanningstabellen för AND kan vi se att det räcker med att ett av invärdena är satt till 0 för att utvärdet ska bli 0. Det kan vi utnyttja om vi vill sätta en specifik bit till 0 genom att skapa en bitmask med 0 i den eller de positioner vi vill sätta till 0, samtidigt som övriga bitar i masken får värdet 1 eftersom det inte påverkar värdet på de positionerna.

Vi ändrar biten på position 5 med hjälp av en bitmask och AND:

Position	7	6	5	4	3	2	1	0
In	1	0	**1**	0	0	1	1	0
Bitmask	1	1	**0**	1	1	1	1	1
Ut	1	0	**0**	0	0	1	1	0

Att ta reda på värdet av en bit

Om man har en bitsträng och vill veta värdet av en specifik bit kan man återigen använda AND, men den här gången genom att sätta positionen för den bit vars status vi är intresserad av till 1 medan vi sätter resten av bitarna i bitmasken till 0. Det ger att alla andra bitar tillsammans med bitmasken kommer ge 0, och resultatet av operationen In AND Bitmask kommer ha värdet av den enda position vi är intresserade av.

Vi vill veta status på biten i position 3 och använder en bitmask och AND:

Ut ger oss värdet i tredje biten (0):

Position	7	6	5	4	3	2	1	0
In	1	1	1	0	**0**	1	0	1
Bitmask	0	0	0	0	**1**	0	0	0
Ut	0	0	0	0	**0**	0	0	0

Ut ger oss värdet i tredje biten (1):

Position	7	6	5	4	3	2	1	0
In	1	1	1	0	**1**	1	0	1
Bitmask	0	0	0	0	**1**	0	0	0
Ut	0	0	0	0	**1**	0	0	0

Rent praktiskt kan man av Ut se om biten var 0 då resultatet är 0, och man ser att biten är 1 då resultatet är skilt från 0.

Att toggla en bit

Att toggla en bit är sätta biten till 1 om den var 0, och att sätta den till 0 om den var 1. Man kan tänka på en lampknapp: Om man trycker på knappen slås lampan av om den var på, medan den slås på om lampan var av.

Man kan toggla en bit genom att sätta de bitar man vill toggla till 1 i bitmasken och sedan använda XOR. I sanningstabellen för XOR kan vi se att resultatet av en XOR-operation är sant bara om ett och endast ett av invärdena är sant. Det ger att om vi har 1 i invärdet kommer XOR med 1 att ge 0, och om vi har 0 i invärdet kommer XOR med 1 att ge 1. Bitar som vi inte vill ändra låter vi i bitmasken vara 0 så ändras inte deras värde.

Här togglar vi bitarna på position 5 och 4 med hjälp av XOR:

Position	7	6	5	4	3	2	1	0
In	1	0	**1**	**0**	1	0	1	0
Bitmask	0	0	**1**	**1**	0	0	0	0
Ut	1	0	**0**	**1**	1	0	1	0

Praktisk användning av maskning

Maskning är väldigt användbart till exempel om man har ett flertal LEDar och får dessa att lysa genom att när biten på position X sätts till 1 så lyser LED X. Vill man då tända en specifik LED använder man en bitmask där alla bitar är 0 utom den bit som motsvarar lampan man vill tända tillsammans med OR. Vill man istället släcka en specifik LED sätter man alla bitar i bitmasken till 1 utom den som motsvarar LEDen. Den biten sätter man till 0, och sedan använder man AND för att släcka LEDen. Vill man toggla en specifik LED kan man använda XOR, och med hjälp av en bitmask kan man också få reda på om en LED är tänd. Ett mer användbart exempel för just att ta reda på värdet i en bit är om man har knappar som fungerar på samma sätt, så att man har en bitsträng som motsvarar varje knapp. Då kan man ta reda på om en knapp är intryckt eller inte genom att maska ut den bit som motsvarar den knapp man är intresserad av.

Man kan också använda maskning för att rita ut grafik. Mer information om det finns på http://en.wikipedia.org/wiki/Mask_(computing)#Image_masks

Binär aritmetik

Aritmetiken är den del av matematiken som behandlar de fyra räknesätten (+,-,*,/). Binär aritmetik är helt enkelt att räkna de fyra räknesätten i det binära talsystemet.

Addition

Man räknar addition i det binära talsystemet på samma sätt som i det decimala talsystemet, med skillnaden att med bara två möjliga värden på varje position kan man enkelt ställa upp de olika varianter av addition med två enbitarstal:

$0_2 + 0_2 = 0_2$
$1_2 + 0_2 = 1_2$
$0_2 + 1_2 = 1_2$
$1_2 + 1_2 = 10_2$

Om vi har två större tal kan vi använda oss av uppställning när vi räknar addition:

Minnessiffra	*+1*	*+1*	*+1*	*+1*	
Term 1		1	1	0	1
Term 2	+	0	1	1	1
Summa	**1**	**0**	**1**	**0**	**0**

Vi kan kontrollräkna genom att använda det decimala talsystemet som vi är mer bekanta med. $1101_2 = 13_{10}$ och $111_2 = 7_{10}$. Om vi då tar 13 + 7 ska vi få 20, vilket är 10100_2, och vi kan då se att vi fått rätt resultat.

Minnessiffrorna, de som blir "extra" när man måste använda en extra position till vänster för att markera värdet av additionen, kallas carry.

Subtraktion

Även vid subtraktion av två binära tal är uppställning ett bekvämt alternativ, men för enbitarstal kan vi ställa upp samma tabell som för addition:
$0 - 0 = 0$
$0 - 1 = 1$ (som då har lånat från positionen till vänster)
$1 - 0 = 1$
$1 - 1 = 0$

Om vi istället har två större tal kan vi ställa upp det på följande vis:

Minnessiffra	*10*	*10*	*10*	*10*

Term 1		1	~~1~~	0	~~1~~	~~1~~	~~1~~	0
Term 2	-	0	0	1	0	1	1	1
Differens		**1**	**0**	**1**	**0**	**1**	**1**	**1**

När vi subtraherar får vi låna från positionen till vänster, vilket ger att vi får det binära värdet 10 - 1, vilket ger svaret 1. I de positioner vi har lånat måste vi också stryka det vi har lånat, så att vi inte råkar räkna med det två gånger.

Vi kan kontrollräkna med hjälp av en översättning till det decimala talsystemet. 1101110_2 = 110_{10} och 10111_2 = 23. 110 - 23 = 87, och 87_{10} = 1010111_2 vilket är det resultat vi fått.

Tvåkomplement

Att subtrahera ett negativt tal är samma sak som att addera det motsvarande negativa talet. Exempelvis är 7 - 3 samma sak som 7 + (-3). Vi kan göra samma sak med binära tal, men där brukar man använda tvåkomplement för att ta reda på talets negativa motsvarighet. Anledningen till att vi hellre vill använda addition är för att vi då bara behöver en hårdvarukomponent för att utföra två olika typer av beräkningar, nämligen en adderare.

Ett tals tvåkomplement får man ut genom att först invertera bitsträngen och sedan addera 1. Man behöver också bestämma sig för hur långa bitsträngar man använder. I datorsammanhang brukar det bestämmas av ordlängden, i regel 32 bitar. För enkelhetens skulle kommer vi att räkna på 8 bitar här.

Exempeltal: 255 - 170 = 85
Decimalt har vi 255 - 170, vilket blir 1111 1111 - 1010 1010 i det binära talsystemet. Vi börjar med att räkna ut tvåkomplementet för 1010 1010 för att sedan utföra additionen.

Steg 1: Invertera talet

Tal	1	0	1	0	1	0	1	0
Invers	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**

Steg 2: Addera 1

Carry								*1*	
Invers		0	1	0	1	0	1	0	1
	+	0	0	0	0	0	0	0	1
Summa		**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**

Tvåkomplementet för 1010 1010 är alltså 0101 0110. Vi kan då utföra addition av de två

termerna istället för subtraktion.
Steg 3: Addera

Carry		*1*	*1*	*1*	*1*	*1*	*1*		
Term 1		1	1	1	1	1	1	1	1
Term 2	+	0	1	0	1	0	1	1	0
Summa	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**

Observera att vi använder oss av bitsträngar på 8 bitar, så den rödmarkerade 1an ska inte vara med i slutresultatet utan kastas bort.

Vår summa efter att ha adderat 1111 1111 med 0101 0110 är 0101 0101, vilket är 85_{10}. Vi kan kontrollräkna genom att använda våra ursprungliga tal: 255 - 170 = 85. Därmed stämmer vårt resultat och vi kan se att addition av ett tals tvåkomplement fungerar lika bra som att subtrahera det ursprungliga talet.

Tvåkomplement med decimaler
Exempeltal: 13,5 - 5 = 8,5 -> $13{,}5_{10} = 1101{,}1_2$ och $5_{10} = 101_2$

Steg 1: Först måste vi se till att båda talen använder samma antal bitar och får då 0101,0 och har då två tal med fem bitar.

Steg 2: Invertera det andra talet

Tal	0	1	0	1,	0
Invers	1	0	1	0	1

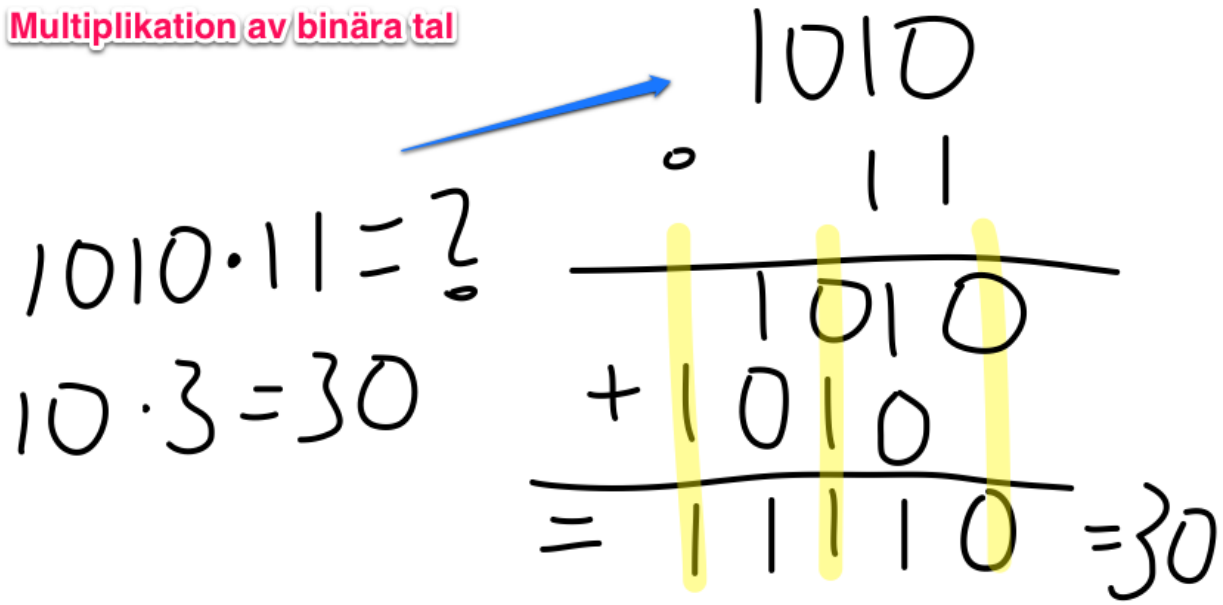
Steg 3: Addera med 1

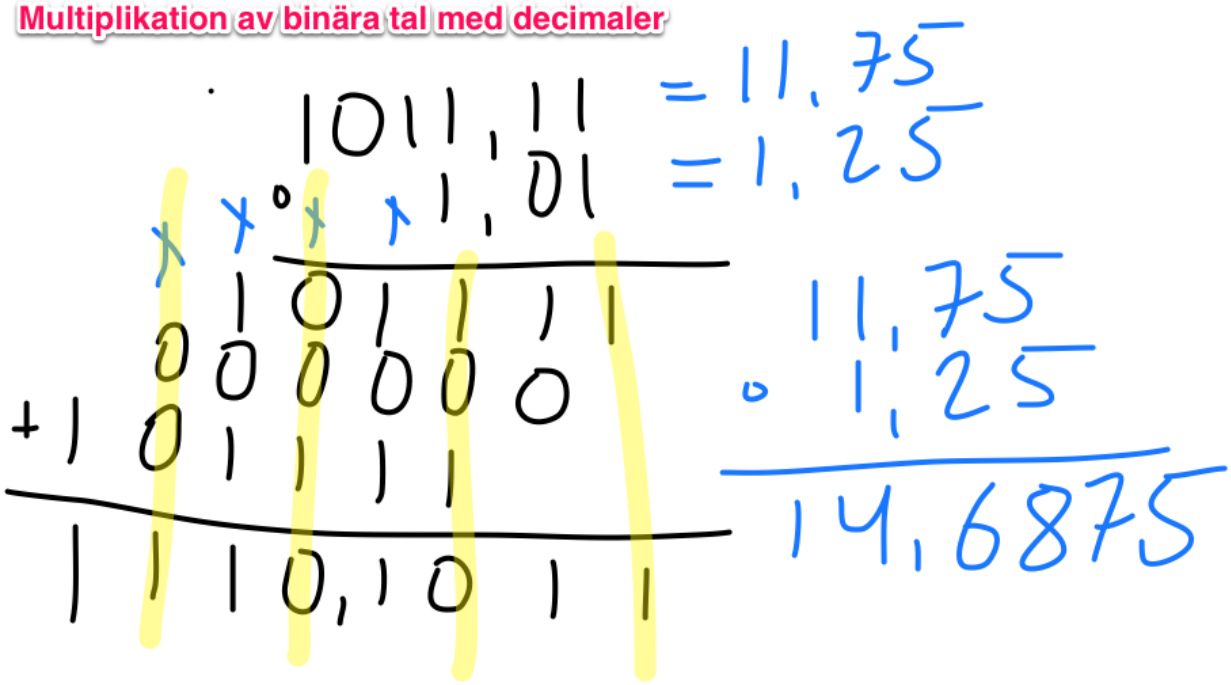
Carry					1	
Invers		1	0	1	0,	1
	+	0	0	0	0	1
Summa		**1**	**0**	**1**	**1,**	**0**

Steg 4: Addera det första talet med tvåkomplementet av det andra talet

Carry		*1*	*1*	*1*		
Term 1		1	1	0	1,	1
Term 2	+	1	0	1	1,	0
Summa	**1**	**1**	**0**	**0**	**0,**	**1**

Vi får overflow som vi ignorerar, och har då kvar $1000,1_2 = 8,5_{10}$.





Antalet decimaler i svaret är lika med summan av antalet decimaler i faktorerna.

Shift

Att skifta en bitsträng betyder att man flyttar samtliga bitar åt höger eller vänster. Om man gör en logisk skiftning betyder det att man skiftar in 0 på den position som “blir över”, och om man

istället gör en aritmetisk skiftning så skiftar man in samma värde som fanns där tidigare.

Logisk shift åt vänster:

In		1	0	1	1	1	0	1	0
Ut	←	0	1	1	1	0	1	0	0

Logisk shift åt höger:

In		1	0	1	1	1	0	1	0
Ut	→	0	1	0	1	1	1	0	1

Aritmetisk shift åt vänster:

In		1	1	0	0	1	1	1	0
Ut	←	1	0	0	1	1	1	0	0

Alltid in med en nolla från höger

Aritmetisk shift åt höger:

In		1	1	0	0	1	1	1	0
Ut	→	1	1	1	0	0	1	1	1

Alltid in med samma som tidigare stod där

Rotate

Utöver shift finns också rotate, med skillnaden att de siffror som roteras ut till höger eller vänster roteras in på motstående sida.

Rotate åt vänster:

In		1	1	1	0	1	0	1	0
Ut	←	1	1	0	1	0	1	0	1

Rotate åt höger:

In		1	1	1	0	1	0	1	0
Ut	→	0	1	1	1	0	1	0	1