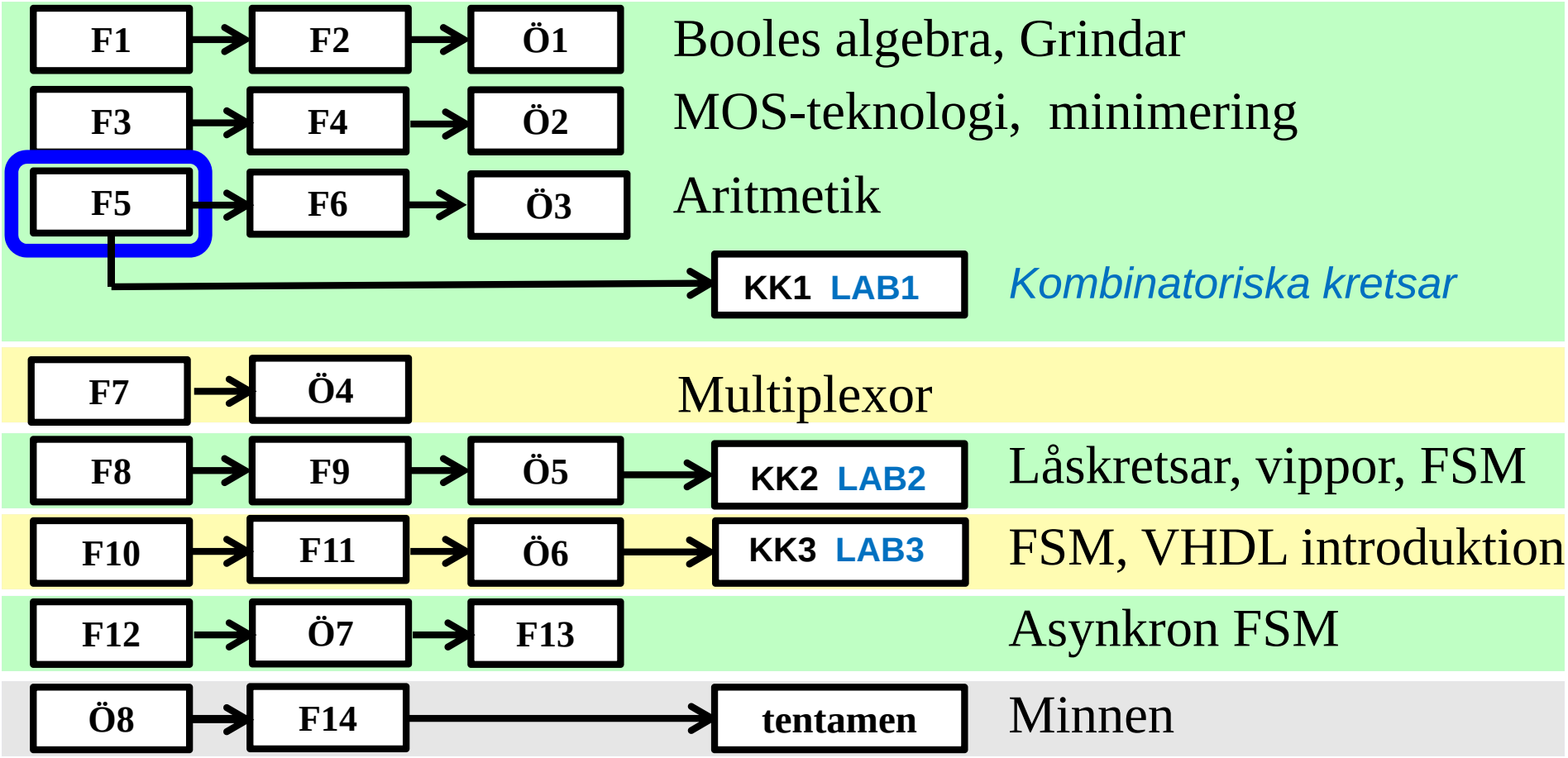
Digital Design IE1204

Föreläsningsbilder av William Sandqvist

F5 Digital aritmetik I

Carl-Mikael Zetterling
bellman@kth.se

IE1204 Digital Design



Föreläsningar och övningar bygger på varandra! Ta alltid igen det Du missat!
Läs på i förväg – delta i undervisningen – arbeta igenom materialet efteråt!

Detta har hänt i kursen …

Talsystem: Decimala, hexadecimala, oktala, binära

$$(175,5)_{10} = (AE.8)_{16} = (256.4)_{8} = (10101110.1)_{2}$$

AND OR NOT EXOR EXNOR Sanningstabell, mintermer Maxtermer PS-form SP-form deMorgans lag Bubbelgrindar Fullständig logik NAND NOR

CMOS grindar, standardkretsar

Minimering med Karnaughdiagrammet 2, 3, 4, 5, 6 variabler

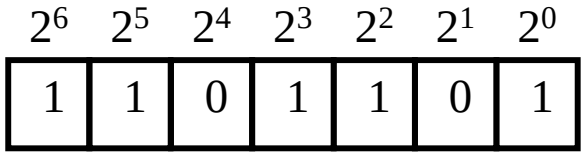
Talrepresentation

Ett tal kan representeras binärt på många sätt. De vanligaste taltyperna som skall representeras är:

- **Heltal, positiva heltal (eng. integers)**
  ett-komplementet, två-komplementet, sign-magnitude
- **Decimala tal med fix tal-område**
  Fix-tal (eng. fixed-point)
- **Decimala tal i olika talområden**
  Flyt-tal (eng. floating-point)

Heltal

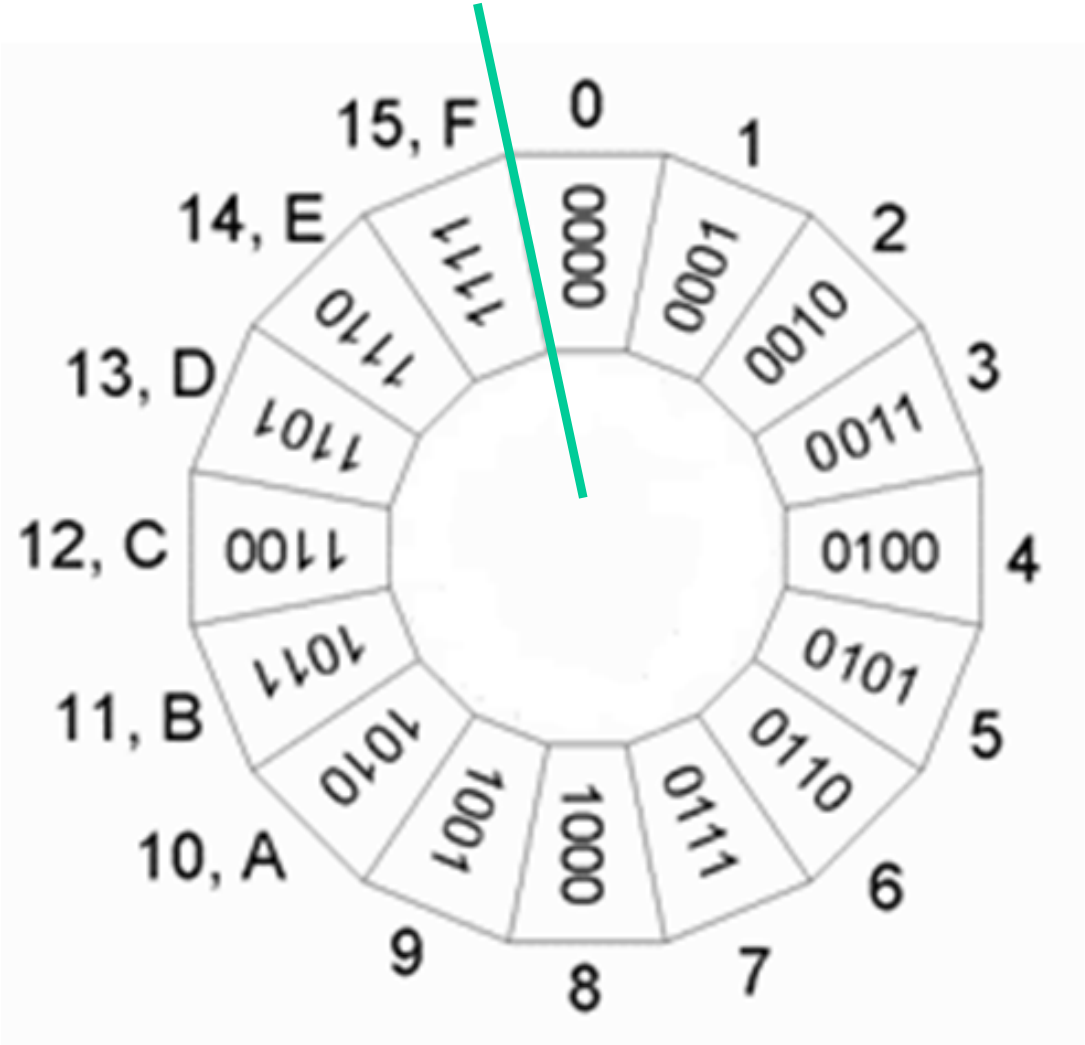
Positiva Heltal:



$$= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 = 109$$

Positiva heltal

Dator-register är ”ringar”. Figuren visar ett fyrabitars-register.
Om man adderar 1 till det högsta talet 15 hamnar man på 0.
(15+1=0)



Carry flagga

Om 15+1 ⇒ 0
Är det bra att bli varnad!
Carry biten brukar sparas för sig som en **Carry flagga** som visar att svaret av additionen blev felaktigt.

- Kanske bättre att svara med 15+1=15 ? (Saturation).

Utöka registret


```
   (9)
+  (9)
-------
   (2)
```

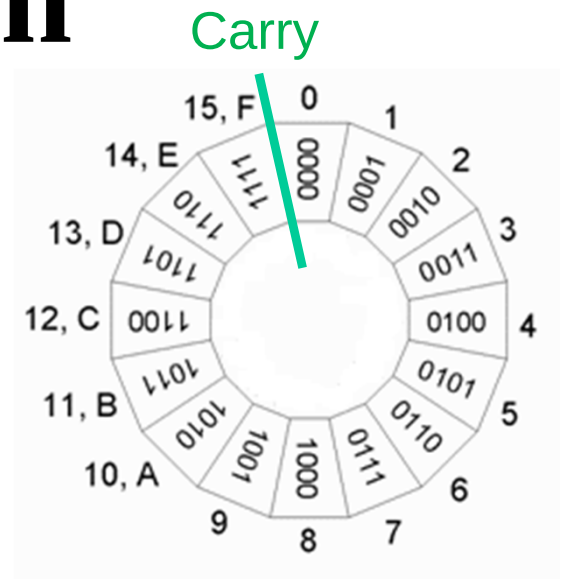
```
  1    1
   1001
+  1001
-------
   0010

 10010 (18)
```

Om resultatet av en addition blir för stort för registret så får man en Carry bit. Den kan användas till att utöka registret, och ger då rätt svar som **5 bitstal**.

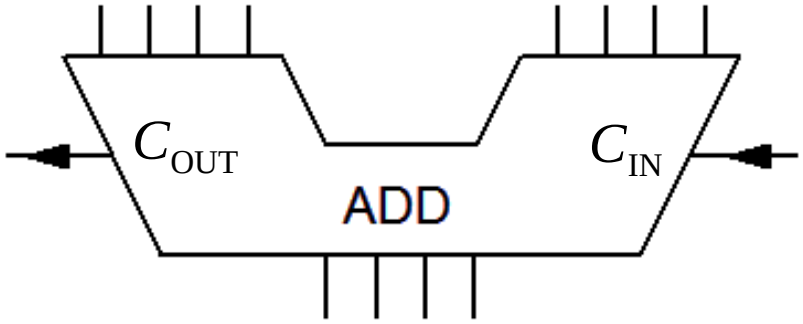
Seriell addition

Addera Hexadecimalt direkt!

$$FEC + 1DB = 11C7$$

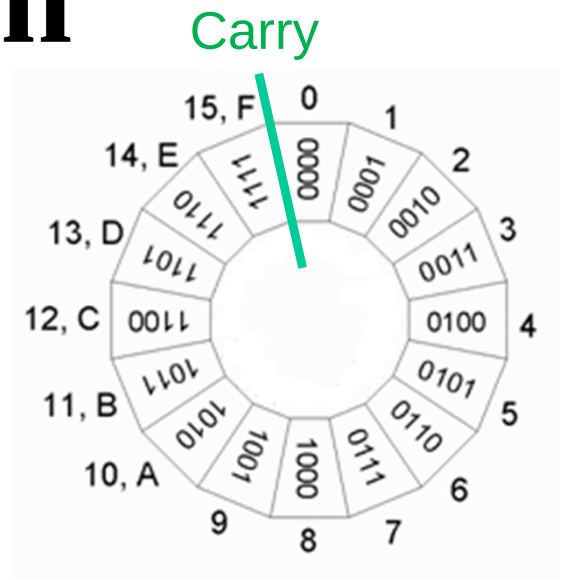


	F	E	C
+	1	D	B
=			

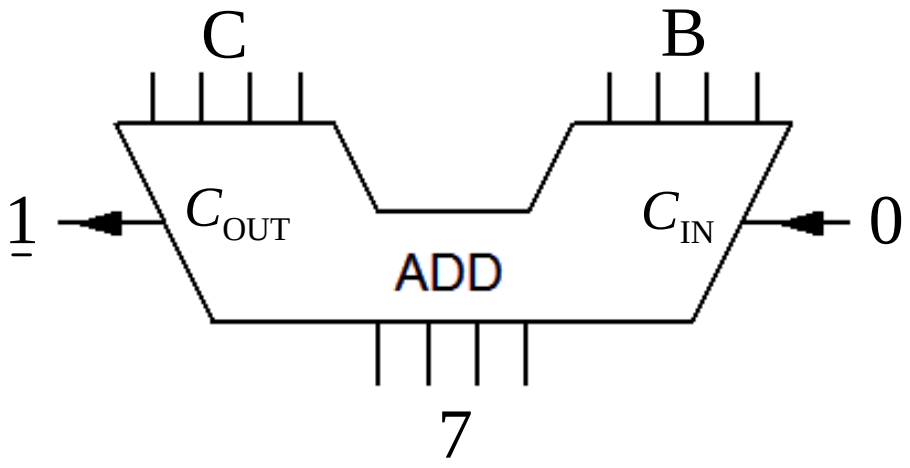


Seriell addition

$$FEC + 1DB = 11C7$$

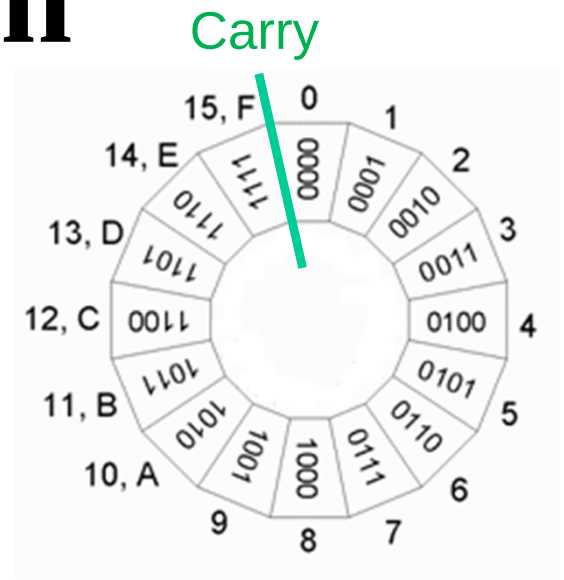


$$\begin{array}{rccc} & & \underline{1} & \\ & F & E & C \\ + & 1 & D & B \\ \hline = & & & 7 \end{array}$$

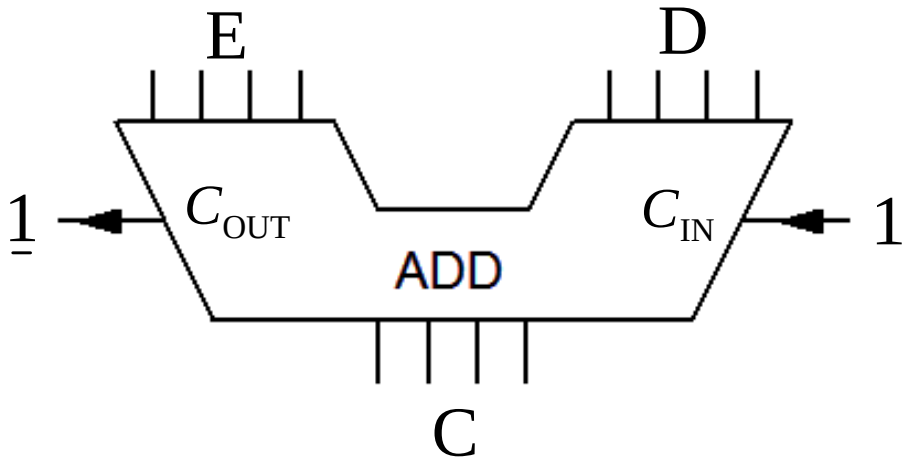


Seriell addition

$$FEC + 1DB = 11C7$$

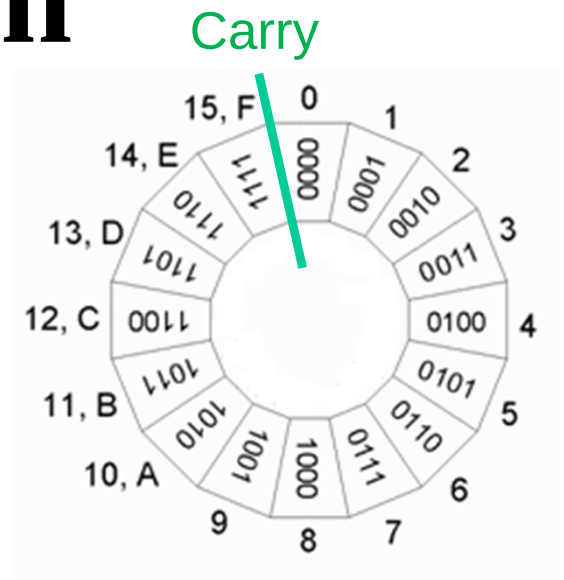


$$\begin{array}{cccc} & \underline{1} & \underline{1} & \\ & F & E & C \\ + & 1 & D & B \\ \hline = & & C & 7 \end{array}$$

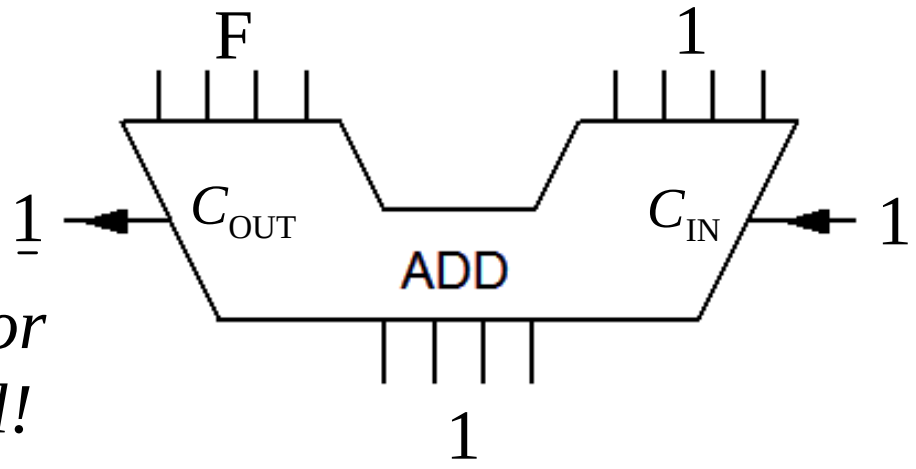


Seriell addition

$$FEC + 1DB = 11C7$$



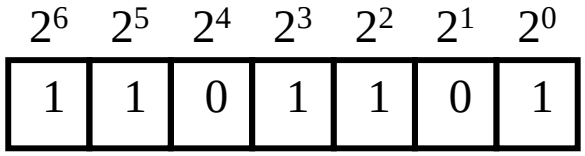
$$\begin{array}{cccc} & \underline{1} & \underline{1} & \underline{1} \\ & F & E & C \\ + & 1 & D & B \\ \hline = & 1 & 1 & C & 7 \end{array}$$



- *Även en liten processor kan räkna med stora tal!*

Heltal med tecken

Positiva Heltal:

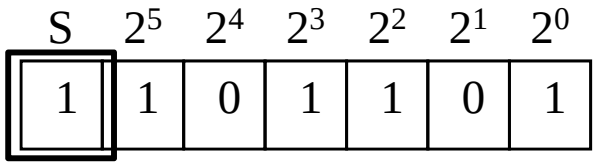


$$= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 = 109$$

Men hur representerar vi negativa tal ???

(Sign-magnitude)

Heltal:

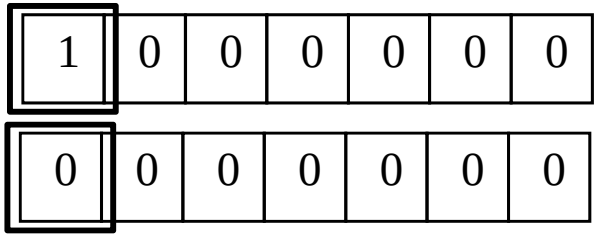


$= - (1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0) = -45$

Magnituden (beloppet) av talet

Tecken-bit

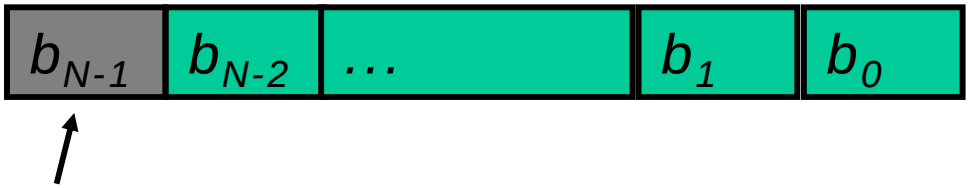
Nackdel: två nollor (+/-) 0



(1-komplement)

De negativa talen är komplementet av de positiva talen. Bit för i det positiva talet bit inverteras för att ge den negativa motsvarigheten.

$B = b_{N-1}\, b_{N-2} \ldots b_1\, b_0$ där $b_i \in \{0,1\}$



Tecken-Bit
(Sign Bit)

Nackdelar:

- **Två nollor (+/-) 0.**
- **Vid vissa additioner krävs justering av resultatet.**

2-komplement heltal

Representation med 2-komplement

$B = b_{N-1}\, b_{N-2} \ldots b_1\, b_0$ där $b_i \in \{0,1\}$

Tecken-Bit
(Sign Bit)

Decimalvärde:

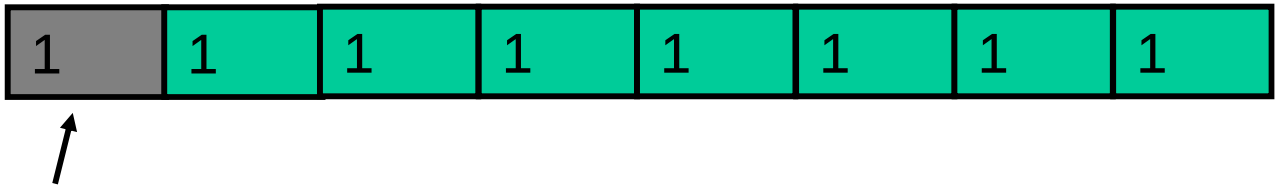
$$D(B) = -\, b_{N-1}\, 2^{N-1} + b_{N-2}\, 2^{N-2} + \ldots + b_1\, 2^1 + b_0\, 2^0$$

- **Detta är den vanligaste representationen av tal ”med tecken”.**

2-komplement heltal

Omvandlingsexempel:

$B = b_{N-1}\, b_{N-2} \ldots b_1\, b_0$ där $b_i \in \{0,1\}$



Tecken-Bit
(Sign Bit)

Decimalvärde:

$$D(B) = -\,b_{N-1}\,2^{N-1} + b_{N-2}\,2^{N-2} + \ldots + b_1\,2^1 + b_0\,2^0$$

$$= -128 + 127 = \mathbf{-1}$$

Det är alltid det största talet som motsvarar -1.

Talkonvertering positivt tal till negativt

Tvåkomplementmetoden

01111	**+15**
10000	**invertera**
10001	**lägg till ett**
10001	**-15**

Talkonvertering
negativt tal till positivt

Tvåkomplementmetoden

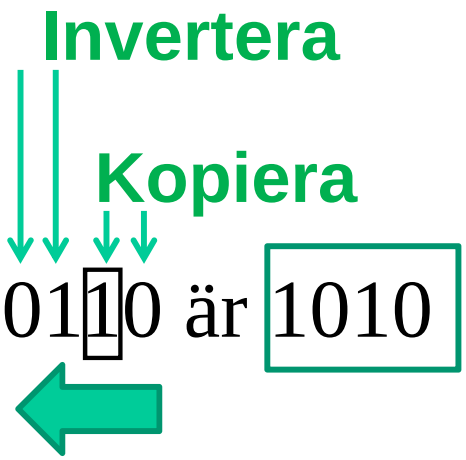
10001	**-15**
01110	**invertera**
01111	**lägg till ett**
01111	**+15**

Samma procedur i bägge riktningarna!

2-komplementet ”snabbt”

- För att lätt ta fram 2-komplementet av ett binärtal kan man använda följande förfarande:
  - Börja från högra sidan
  - Kopiera alla bitar som är 0 och fram till och med den första 1:an
  - Invertera därefter alla andra bitar

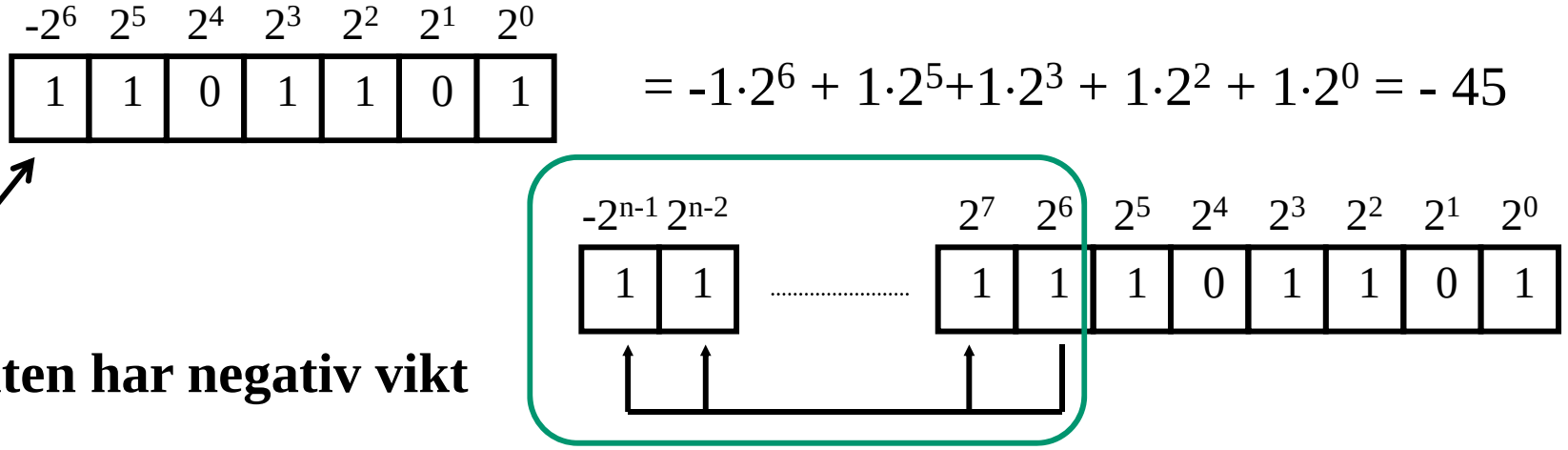
Exempel: 2-komplement från 0110 är 1010



Sign-extension

Vid beräkningar i datorer behöver man ofta öka antalet siffror (bitar) inför någon beräkning – hur gör man det med negativa tal?

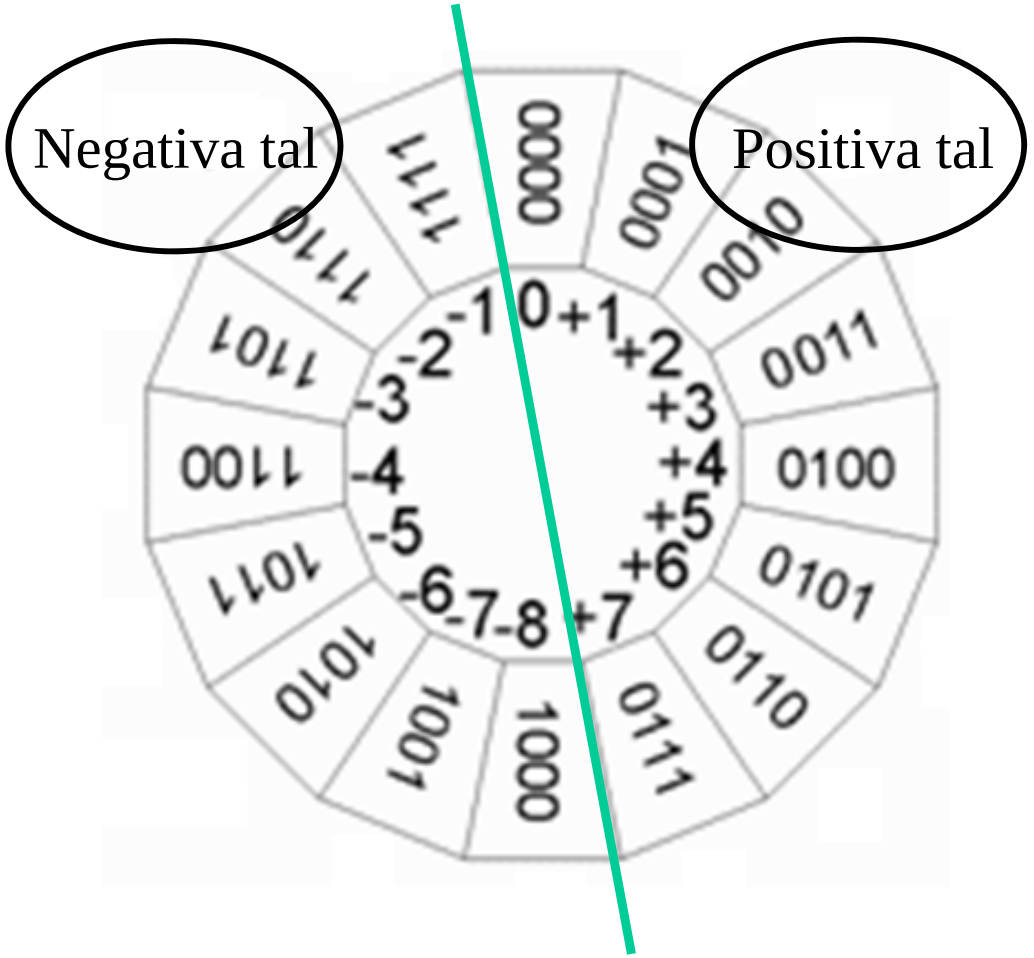
Heltal:



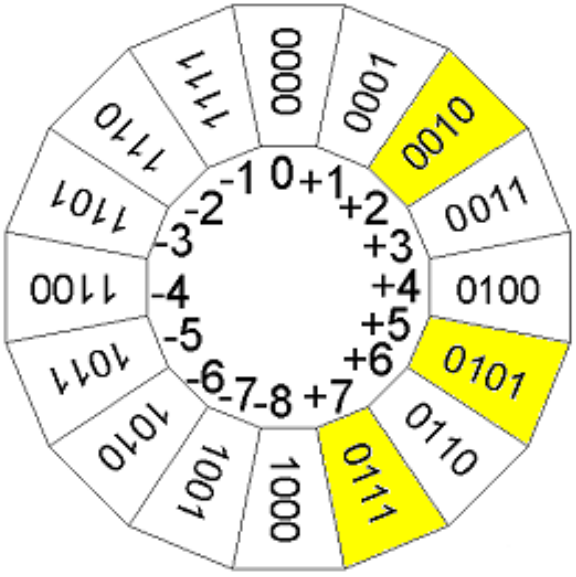
Om man vill utvidga talområdet genom att använda flera bitar kopierar man teckenbiten till de nya bitarna!

2-komplement heltal

Dator-register är ”ringar”. Figuren visar ett fyrabitars-register. När man räknar med ”tal med tecken” är de negativa talen den vänstra halvan av ringen.

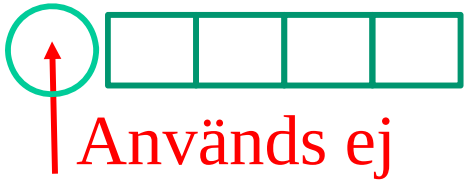


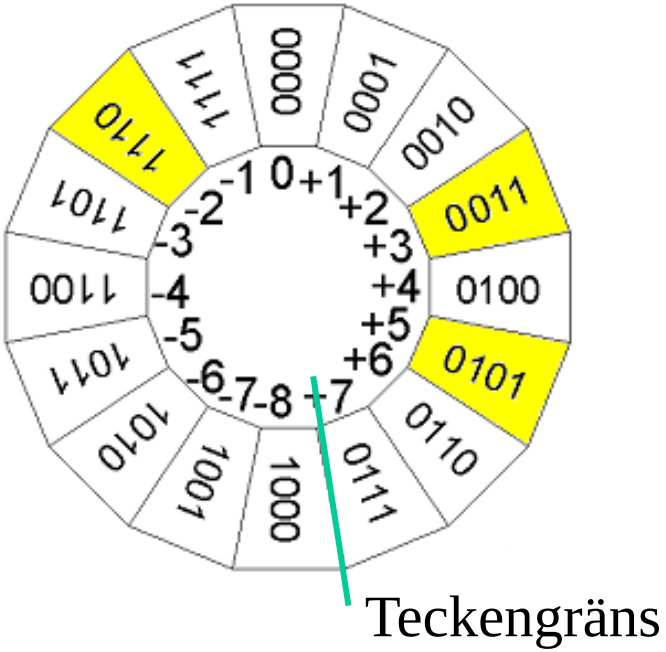
Addition (BV: sida 264)




```
   (+5)          0101
+  (+2)       +  0010
-------       -------
   (+7)          0111
```

Addition (BV: sida 264)



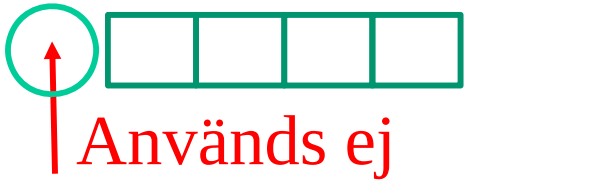


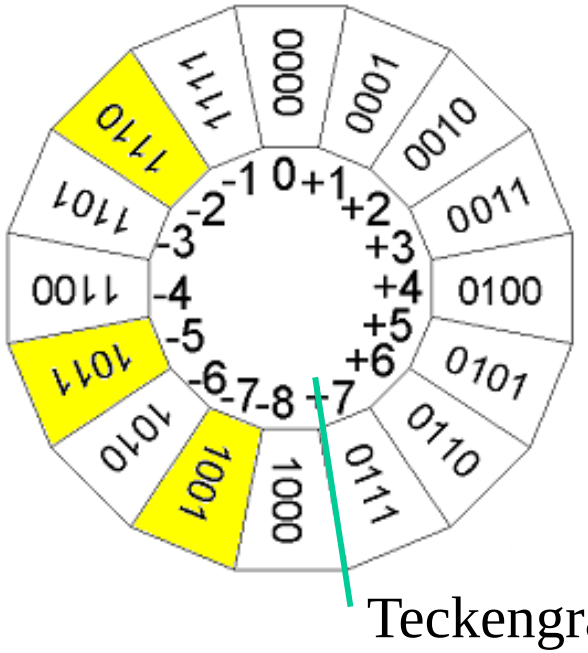

```
   (+5)
 + (-2)
-------
   (+3)
```

```
  1 1
   0101
 + 1110
--------
   0011
```

Carry-biten används ej!

Addition (BV: sida 264)





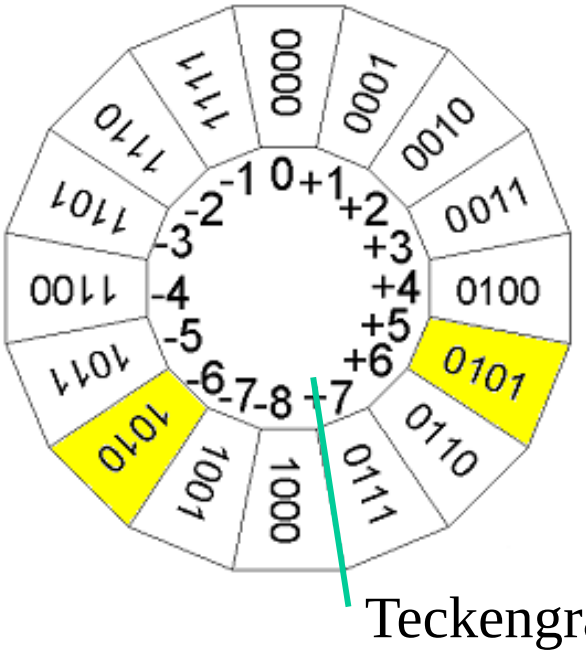

```
   (-5)
+  (-2)
-------
   (-7)
```

```
  1 11
    1011
+   1110
--------
   1001
```

Carry-biten används ej!

Overflow!

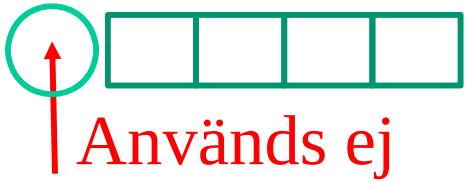
Overflow – teckenbiten stämmer *inte* överens med ingående tal...

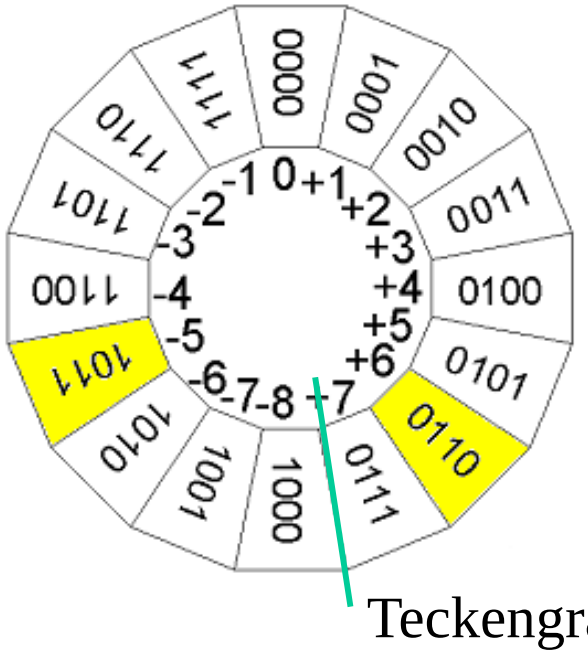



```
   (+5)
 + (+5)
-------
   (-6)
```

```
    1  1
    0101
 +  0101
-------
   1010
```

Overflow 2





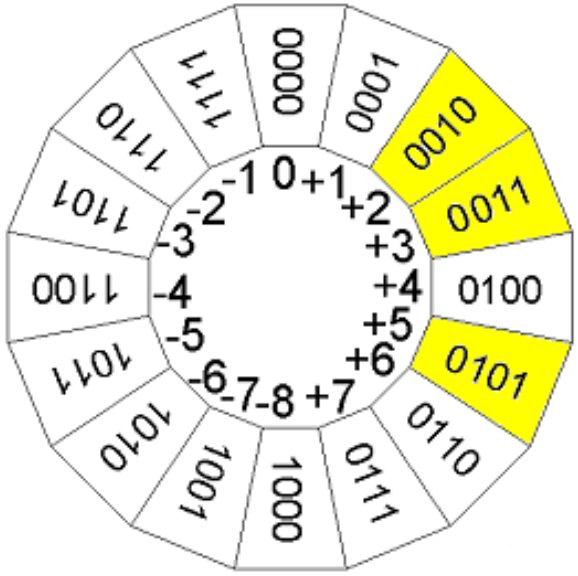
Overflow – teckenbiten stämmer *inte* överens med ingående tal...

```
   (-5)
+  (-5)
-------
  (+6)
```

```
 1  11
   1011
+  1011
--------
   0110
```

Carry-biten används ej!

Subtraktion (BV: sida 265)



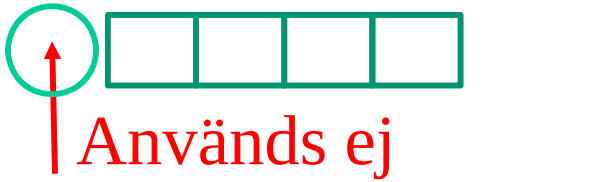

```
   (+5)
 - (+2)
--------
   (+3)
```

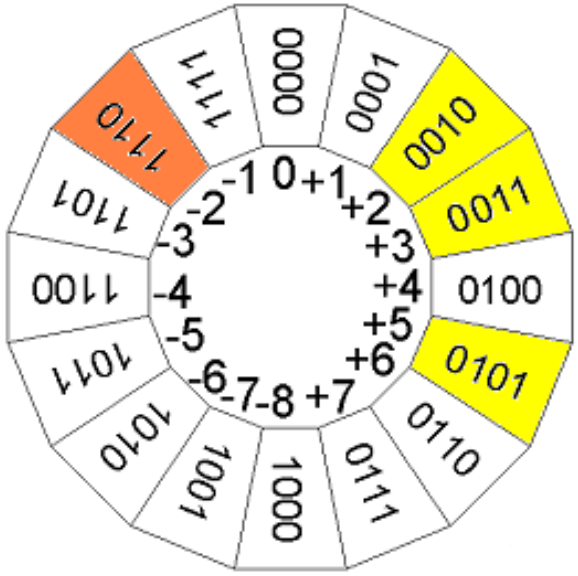
”Låna” ett

```
    10
   0101
 - 0010
--------
   ????
```

Hur gör man subtraktionen på ett enkelt sätt?

Subtraktion (BV: sida 265)



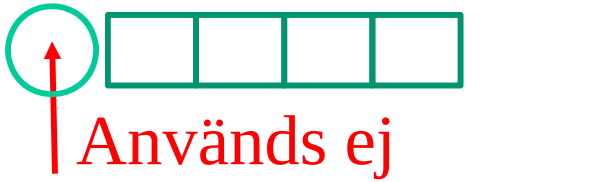


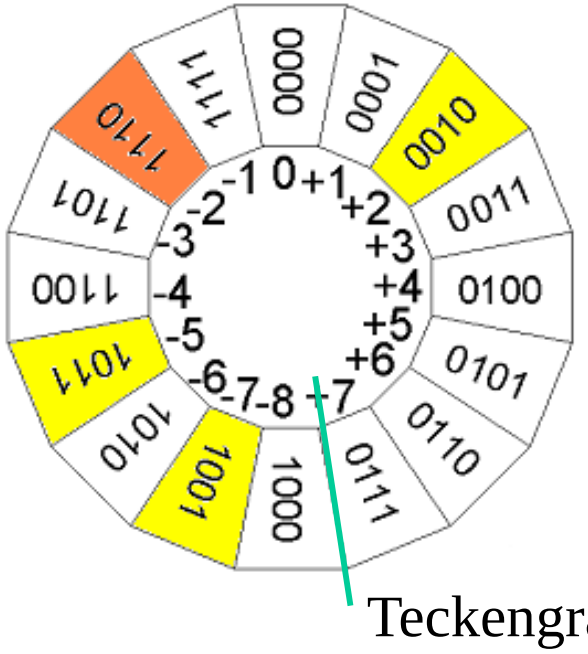

```
  (+5)      0101        1 1
- (+2)    - 0010      + 1110
-------   --------    --------
  (+3)      ????        0011
```

Carry-biten används ej!

Gör en **addition med 2-komplementet** i stället!

Subtraktion (BV: sida 265)

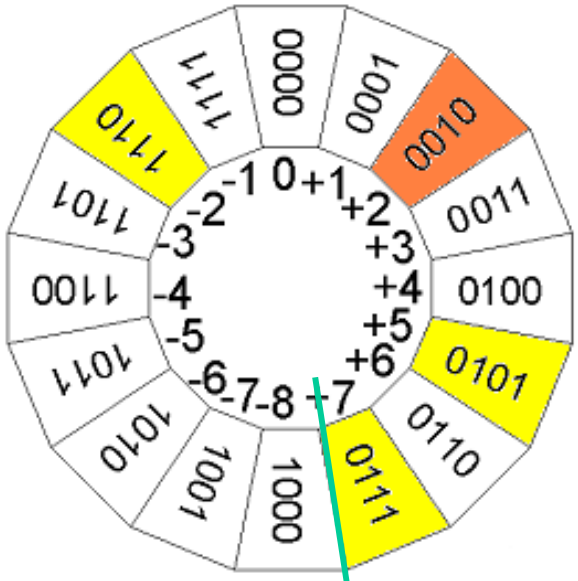





```
  (-5)        1011          111
- (+2)      - 0010      + 1110
-------     --------    --------
  (-7)        ????        1001
```

Carry-biten används ej!

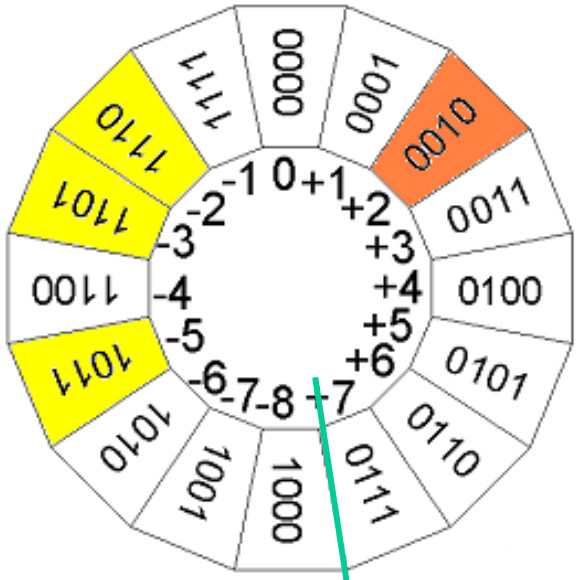
Subtraktion (BV: sida 265)



Teckengräns

```
  (+5)      0101        0101
- (-2)    - 1110      + 0010
------    ------      ------
  (+7)      ????        0111
```

Subtraktion (BV: sida 265)



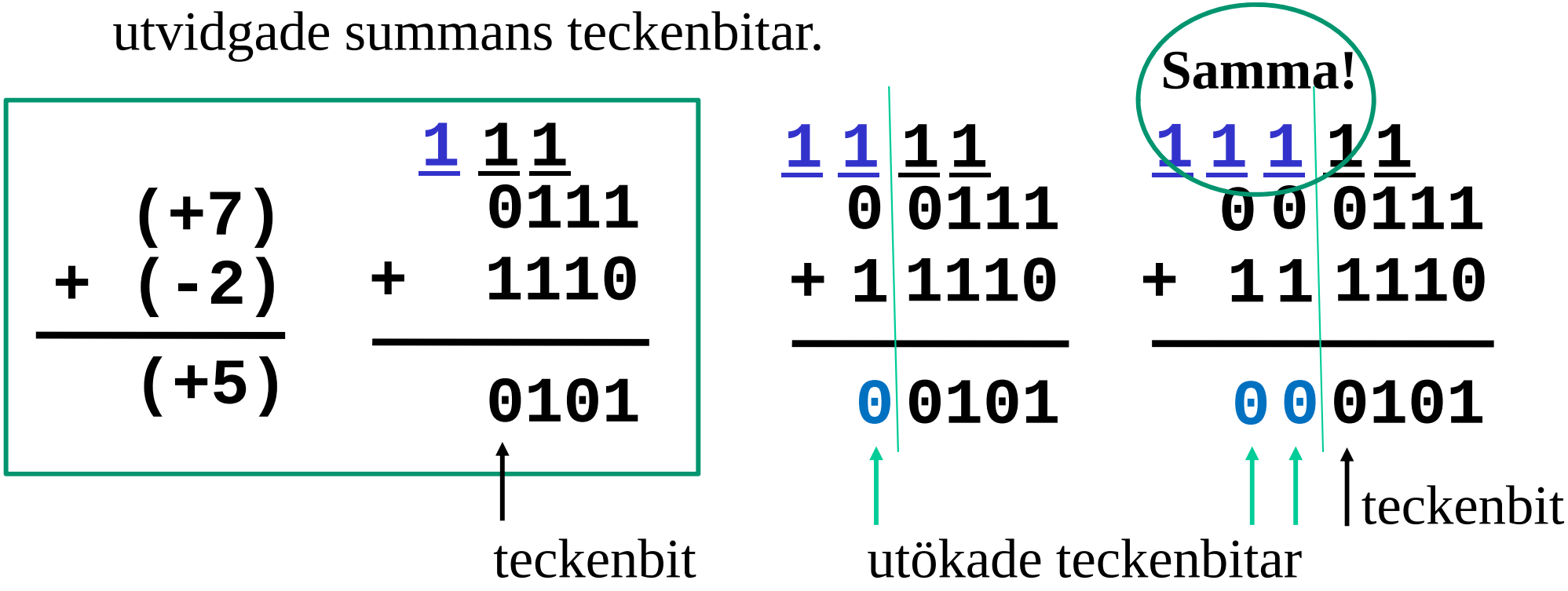

```
  (-5)        1011          1
- (-2)      - 1110        1011
-------     -------      + 0010
 (-3)         ????       ------
                          1101
```

2-komplement sammanfattning

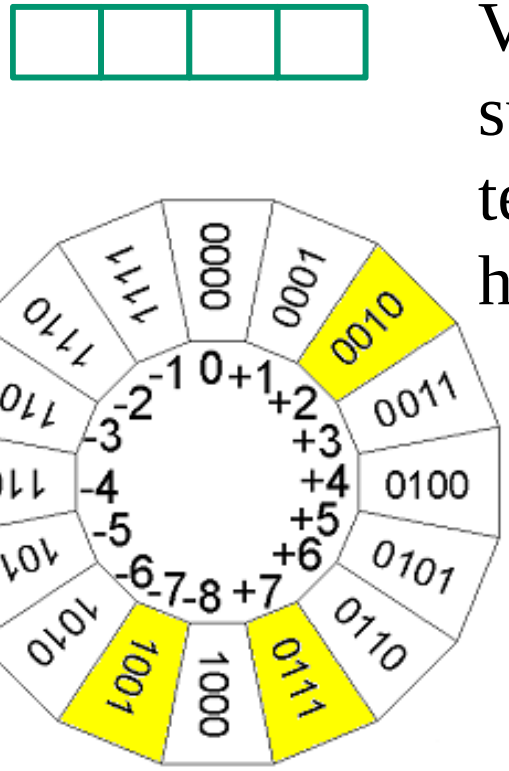
- **Område**: -2^{N-1} upp till $2^{N-1} - 1$
- **Negation**: Invertera varje bit (det boolska komplementet), addera sedan 1.
- **Expansion av bit-längd**: Lägg till ytterligare bit positioner till vänster om teckenbiten, med samma värde som teckenbiten.
- **Overflow-regeln**: Om två tal med samma tecken adderas, så har det blivit overflow om resultatet har ett motsatt tecken.
- **Subtraherings-regeln**: För att subtrahera B från A, ta två-komplementet av B och addera till A.

Hur kan Carry biten användas vid 2-komplementtal?

- Vid teckenutvidgning genererar efterföljande Carry den utvidgade summans teckenbitar.



Alternativt sätt att detektera overflow (BV: sida 271)



Vid overflow kan summan ***inte*** teckenutvidgas med hjälp av Carry!

```
   (+7)
+  (+2)
-------
   (-7)
```

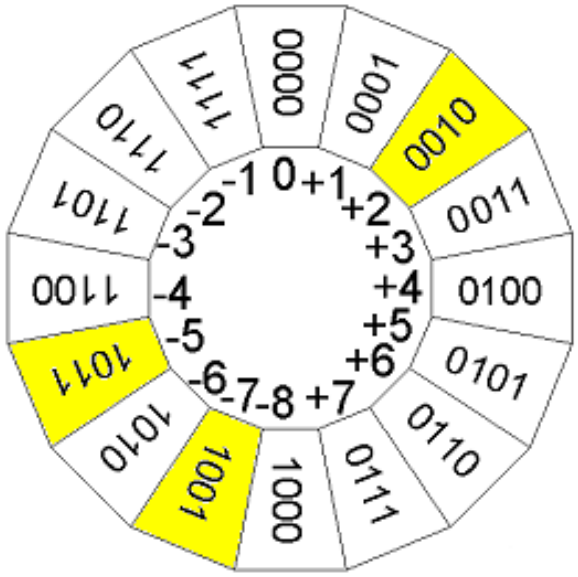
***Inte* samma!**

c_4=0 c_3=1

```
 0 11
  0111
+ 0010
------
  1001
```

Overflow eftersom c_4 och c_3 är **olika**!

Alternativt sätt att detektera overflow (BV: sida 271)

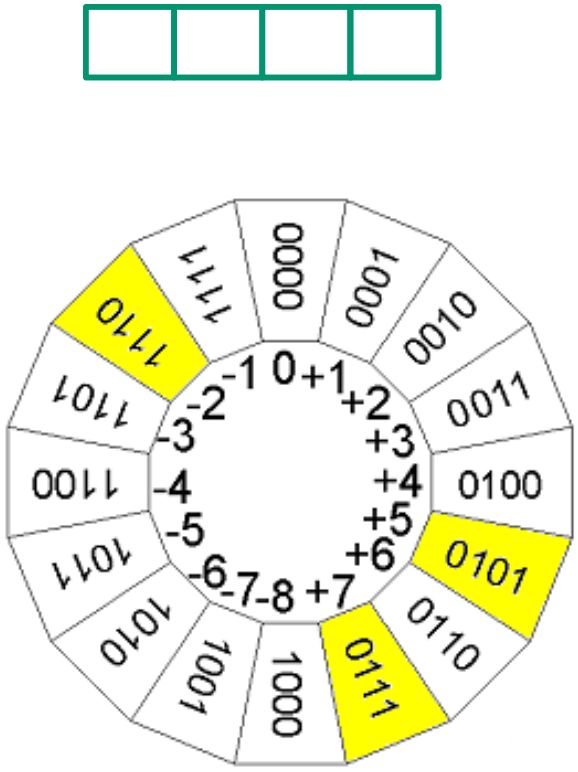


c_4=0 c_3=0

(-7)	1001
+ (+2)	+ 0010
(-5)	1011

Inte Overflow eftersom c_4 och c_3 är **lika**!

Alternativt sätt att detektera overflow (BV: sida 271)



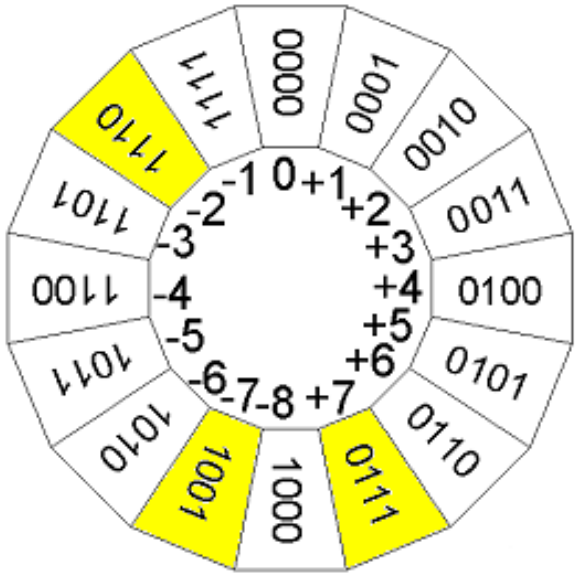

```
   (+7)          0111
 + (-2)        + 1110
--------      --------
   (+5)          0101
```

c_4=1 c_3=1

```
  1 11
```

Inte Overflow eftersom c_4 och c_3 är **lika**!

Alternativt sätt att detektera overflow (BV: sida 271)

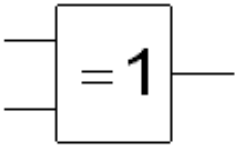


c_4=1 c_3=0

```
  (-7)          1 0        1001
+ (-2)        + 1110     + 1110
-------       -------    -------
  (+7)          0111       0111
```

Overflow eftersom c_4 och c_3 är **olika**!

Logik för att detektera overflow



XOR testar ”olikhet”

För 4-bit-tal

Overflow om c_3 och c_4 är *olika*

Annars är det inte overflow

$$\text{Overflow} = c_3\overline{c}_4 + \overline{c}_3 c_4 = c_3 \oplus c_4$$

För *n*-bit-tal

$$\text{Overflow} = c_{n-1} \oplus c_n$$

Overflow biten kan användas som **overflow flagga**

När är overflow ett problem?

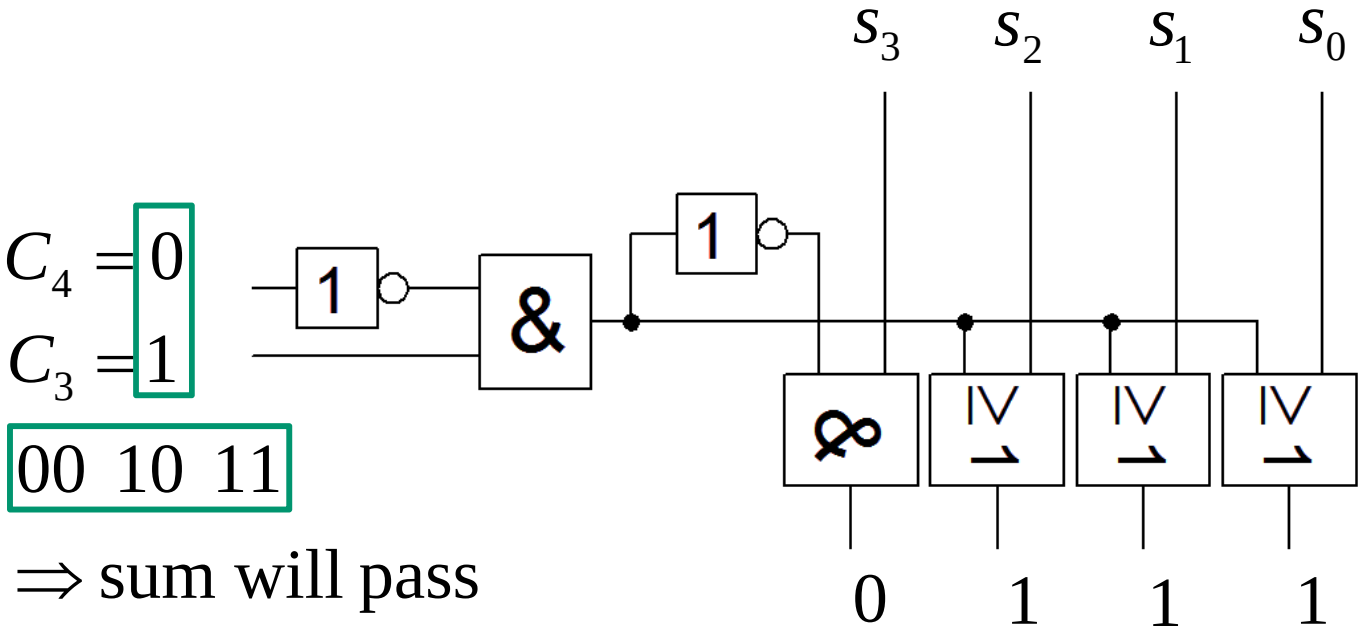
- För en 32-bitars dator är talområdet ±2147483648 stort, då kan inte overflow vara något stort problem!

- Vem ansvarar? – **Du** som programmerare ser till att använda variabeltyper som minst rymmer så stora tal som används av programmet.

Annorlunda är det med digital hårdvara som utför beräkningar på signaler i realtid – här använder man ogärna fler bitar än vad som verkligen är nödvändigt.
Då måste man se upp med overflow!

Mättning (Saturation)

• **Overflow**. En positiv summa har blivit negativ, byt till det **största möjliga positiva värdet**.



$C_4 = 0$

$C_3 = 1$

00 10 11

$\Rightarrow$ sum will pass

01

$\Rightarrow$ sum = 0111

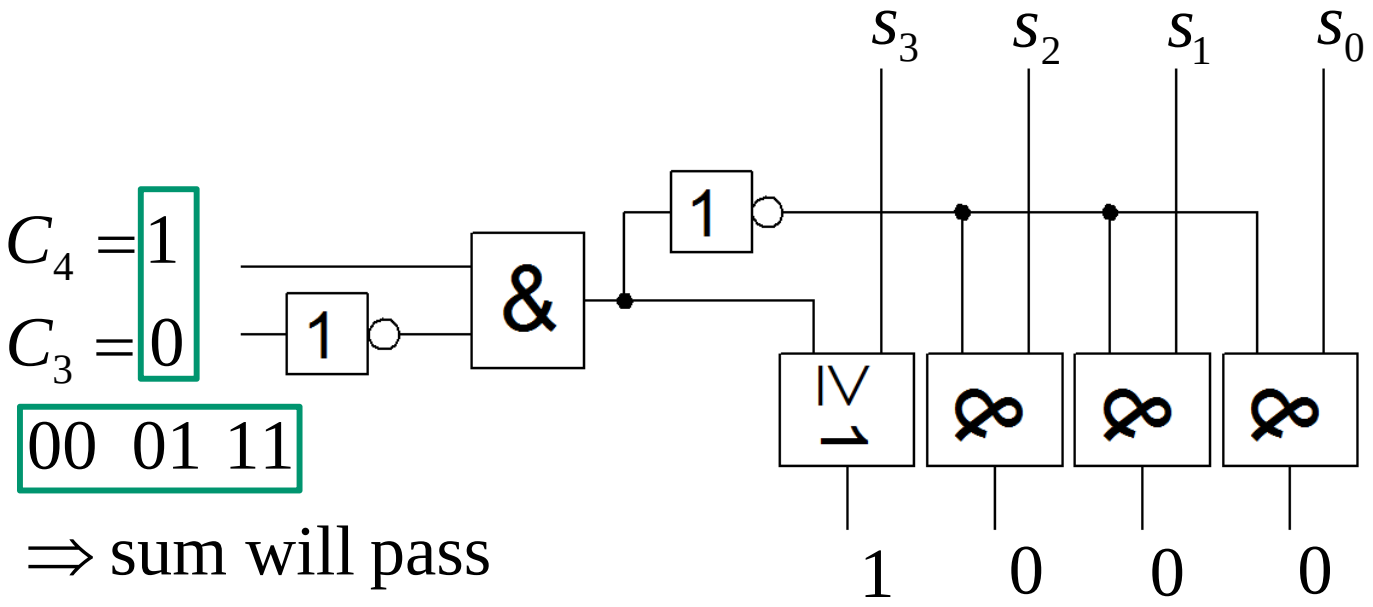
```
 0101
   0101  (+5)
+  0101  (+5)
  1010 (-6) (+10)
```

Overflow

0111 (+7)

Mättning (Saturation)

• **Underflow**. En negativ summa har blivit positiv, byt till det **största möjliga negativa värdet**.



$\Rightarrow$ sum will pass

10

$\Rightarrow$ sum = 1000

```
 1011
   1011  (-5)
+  1011  (-5)
  ------------
   0110  (+5)  (-10)
```

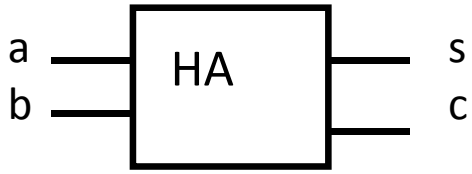
Underflow

1000 (-8)

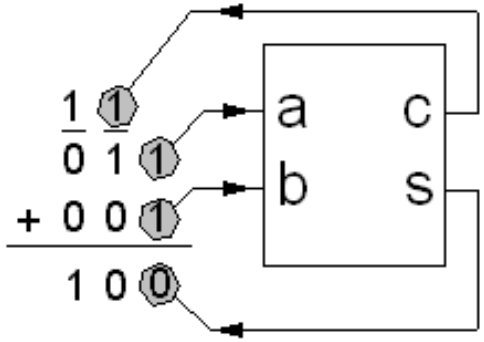
Hårdvara för aritmetik

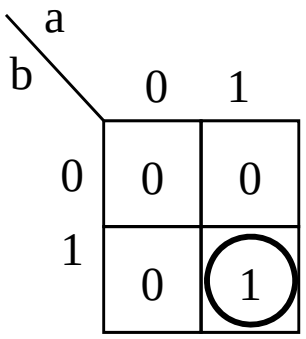
- Nu när Hastighet/Komplexitet har betydelse!

Halv-adderaren (Half adder)

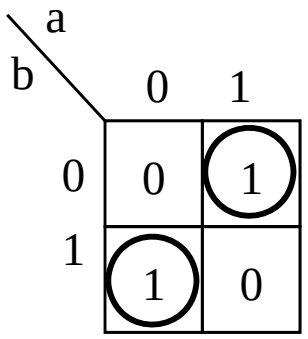


a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



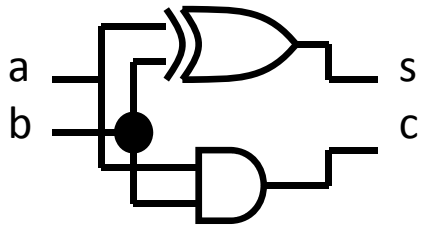


$c = a \cdot b$

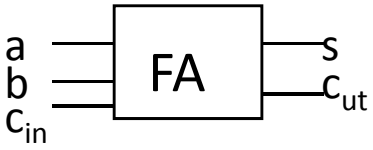


$s = a \oplus b$

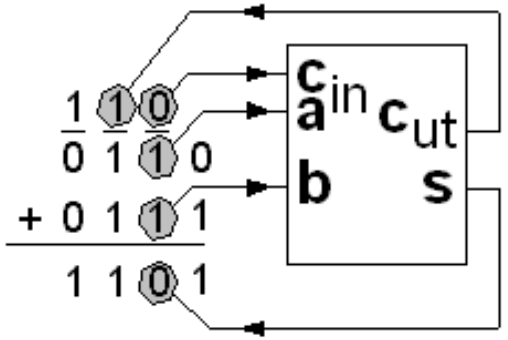
```
 0      0      0      1
   0      0      1      1
 + 0    + 1    + 0    + 1
 ---    ---    ---    ---
   0      1      1      0
```

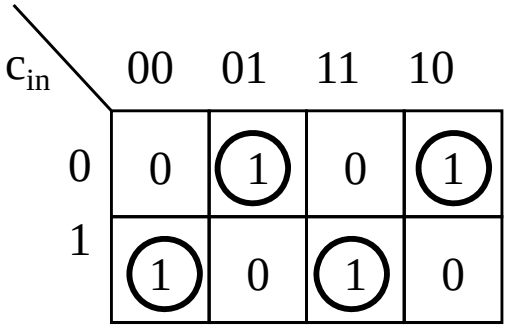


Hel-adderaren (Full adder)

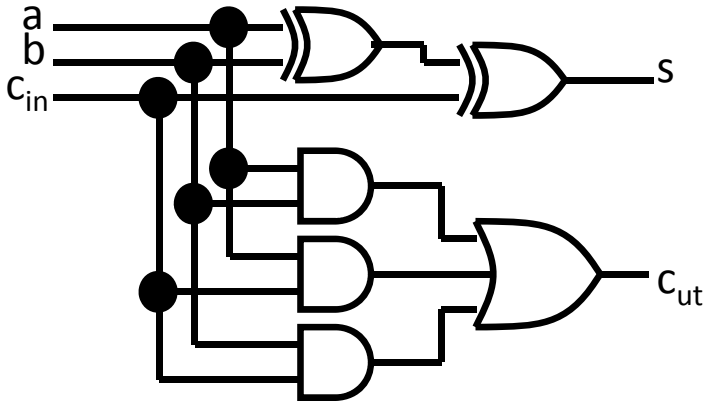


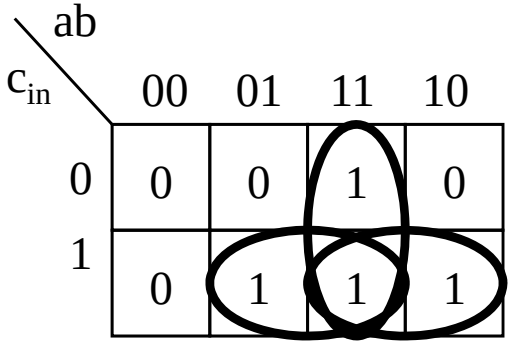
a	b	c_{in}	c_{ut}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





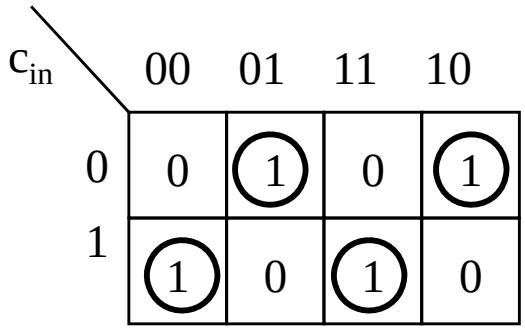
$s = a \oplus b \oplus c_{in}$





$c_{ut} = a\,b + c_{in}a + c_{in}b$

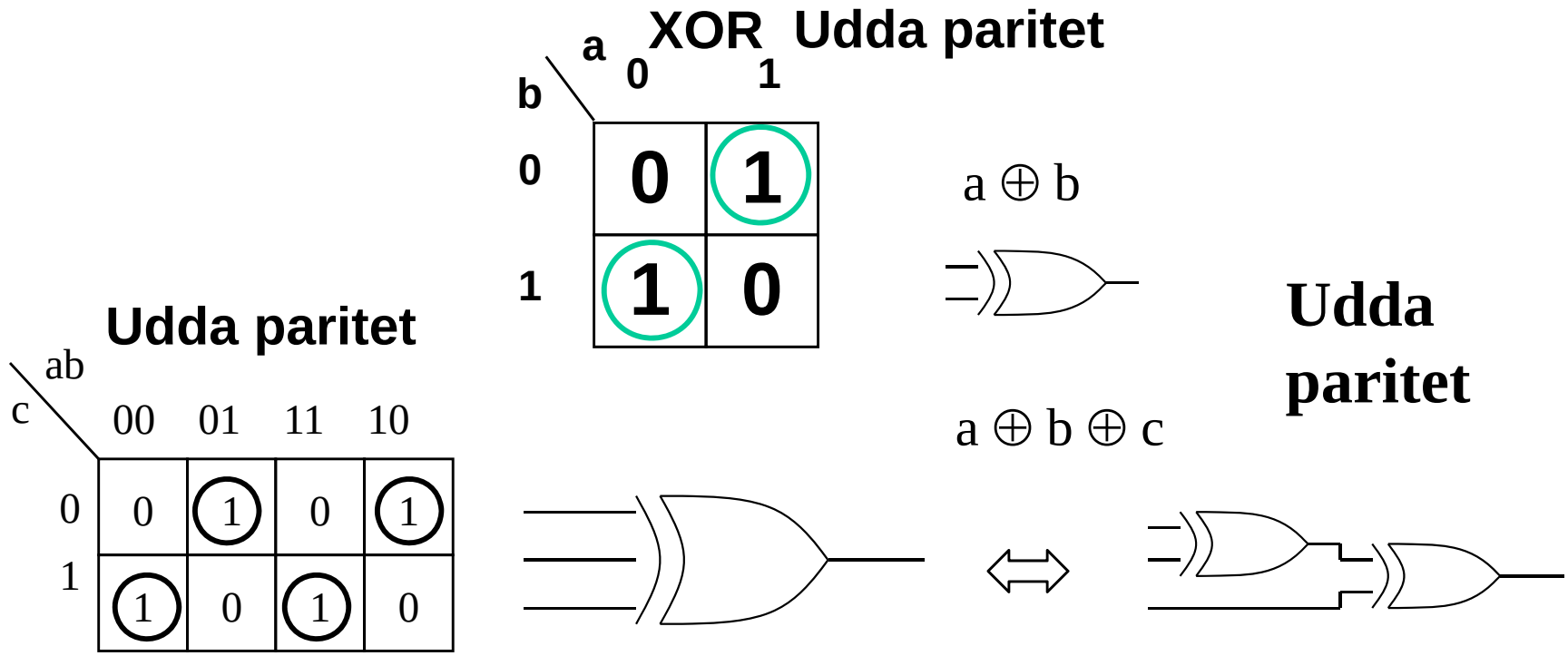
Summafunktionen?



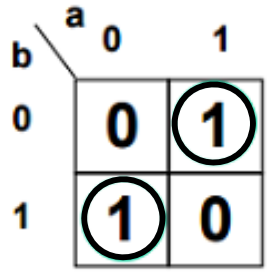
$$s = a \oplus b \oplus c_{in}$$

Summa = Udda paritet

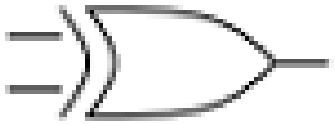
Heladderarens summafunktion är den ”udda”-paritetsfunktionen. Detta är XOR-funktionens naturliga utökning för fler variabler än två. **Udda paritet** är när antalet 1:or på ingångarna är ett udda tal.



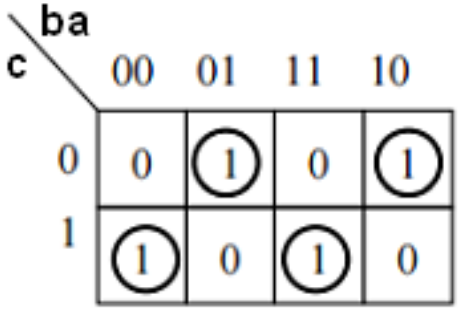
(XOR – odd parity)



$$= a \cdot \bar{b} + \bar{a} \cdot b = a \oplus b$$



XOR-2

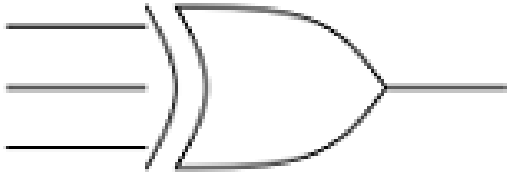


$$= \bar{c}a\bar{b} + \bar{c}\bar{a}b + cab + c\bar{a}\bar{b} =$$

$$= \bar{c}(a\bar{b} + \bar{a}b) + c(ab + \bar{a}\bar{b}) =$$

$$= \bar{c}(a \oplus b) + c(\overline{a \oplus b}) = c \oplus (a \oplus b) =$$

$$= c \oplus a \oplus b$$



XOR-3

(XOR – odd parity)

d c \ ba	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

$$\rightarrow \bar{d}\bar{c}(\bar{b}a + b\bar{a}) = \bar{d}\bar{c}(b \oplus a)$$

$$\rightarrow \bar{d}c(\bar{b}a + ba) = \bar{d}c(\overline{b \oplus a})$$

$$\rightarrow dc(\bar{b}a + b\bar{a}) = dc(b \oplus a)$$

$$\rightarrow d\bar{c}(\bar{b}a + ba) = d\bar{c}(\overline{b \oplus a})$$

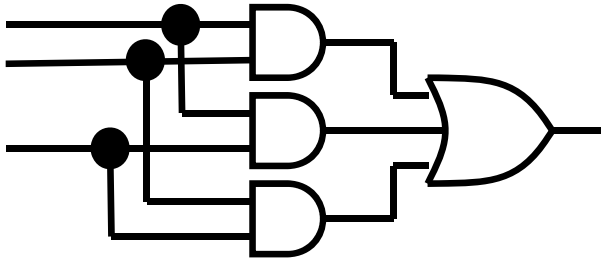
$$\bar{d}\bar{c}(b \oplus a) + dc(b \oplus a) + d\bar{c}(\overline{b \oplus a}) + \bar{d}c(\overline{b \oplus a}) =$$

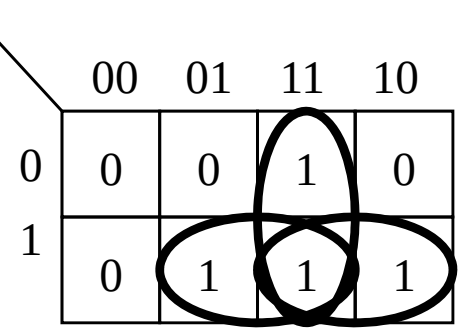
$$= (\bar{d}\bar{c} + dc)(b \oplus a) + (d\bar{c} + \bar{d}c)(\overline{b \oplus a}) = (d \oplus c)(\overline{b \oplus a}) + (\overline{d \oplus c})(b \oplus a) =$$

$$= (d \oplus c) \oplus (b \oplus a) = \boxed{d \oplus c \oplus b \oplus a}$$

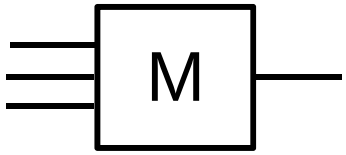
XOR-4

Carryfunktionen?



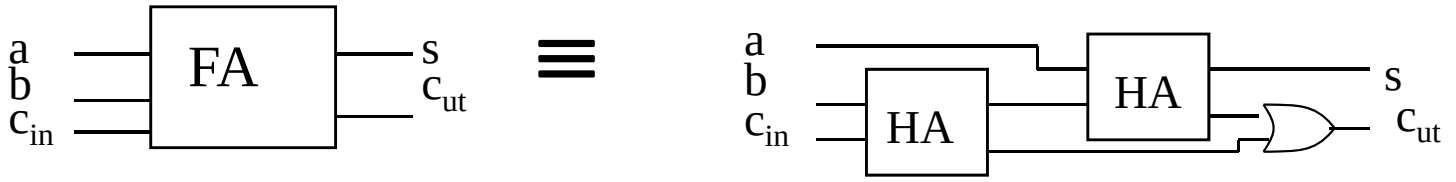


- **Majoritetsfunktionen**. Utgången antar det värde 1/0 som flest av ingångarna har.



Hel-adderaren med två ½ -adderare

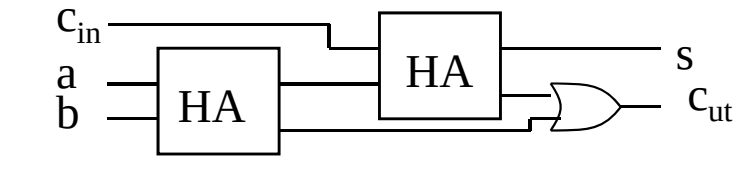
Vi kan även konstruera en hel-adderare mha två halv-adderare och en OR-grind

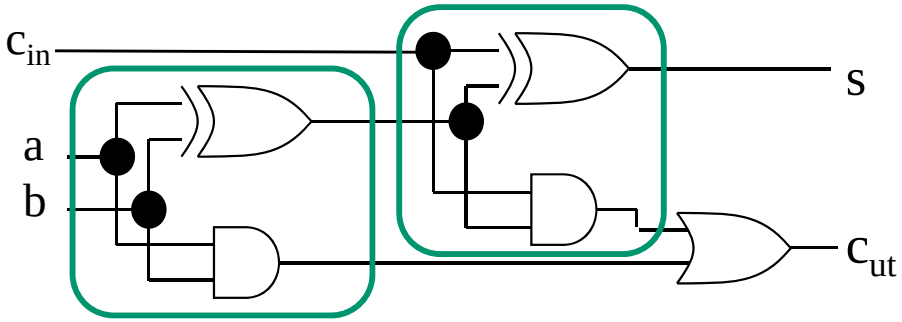


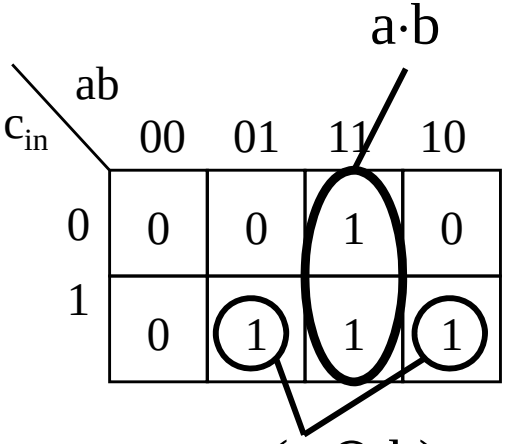
Dekomposition innebär att man ser kretsen som sammansatt av byggblock. Med hjälp av sådana kända byggblock kan man sedan bygga helt nya system **Komposition**.

Hel-adderaren ½+ ½ = 1

a	b	c_{in}	c_{ut}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1







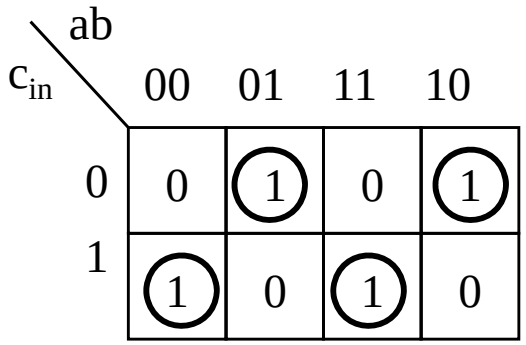
$(a \oplus b)\cdot c_{in}$

$c_{ut} = (a \oplus b)\cdot c_{in} + a\cdot b$

$s = (a \oplus b) \oplus C_{IN}$

$C_{UT} = (a \oplus b)\cdot C_{IN} + a\cdot b$

Man kan använda (a⊕b) till både s och c_{ut} !



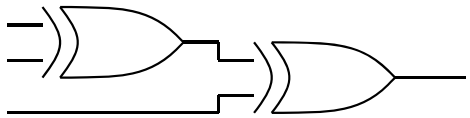
$s = a \oplus b \oplus c_{in}$

(Paritetsfunktionen – trevägs ljuskontroll)

c_{in}	00	01	11	10
0	0	(1)	0	(1)
1	(1)	0	(1)	0

$s = a \oplus b \oplus c_{in}$

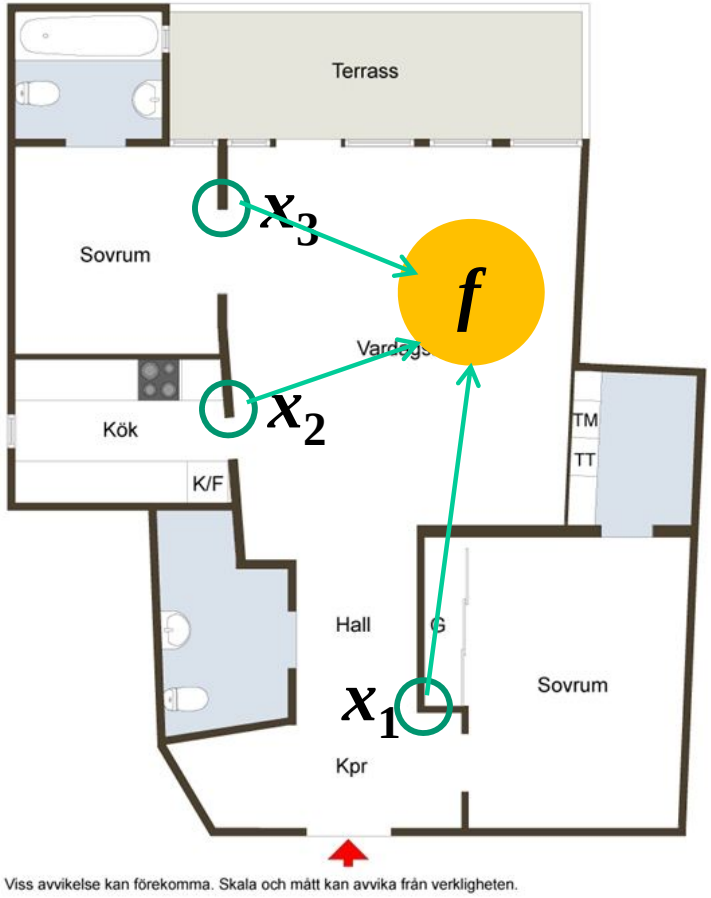
Udda paritet.



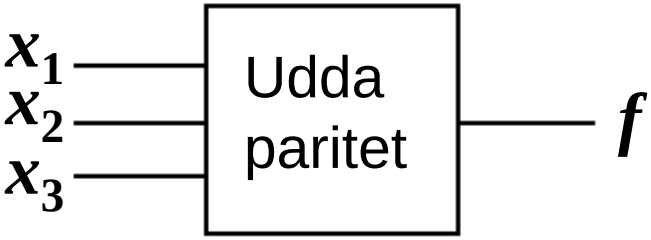
Trevägs ljuskontroll - *revisited*

Brown/Vranesic: 2.8.1

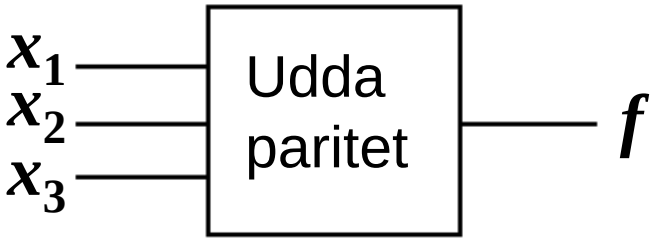
Antag att vi behöver kunna tända/släcka vardagsrummet från tre olika ställen. Lösningen är paritetsfunktionen.



Paritetsfunktionen



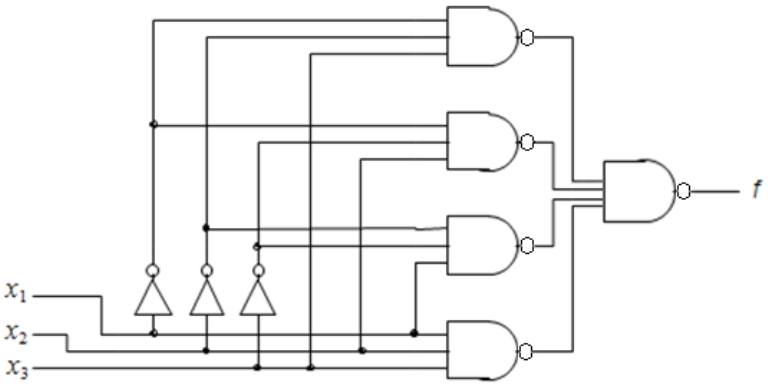
XOR eller NAND?

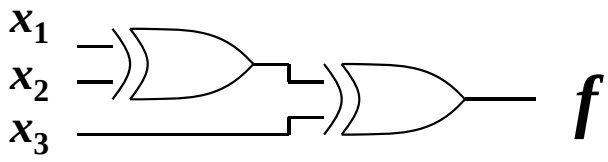


Den tidigare lösningen baserades på NAND-grindar.

XOR-grindar blir mycket effektivare än NAND grindar!

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

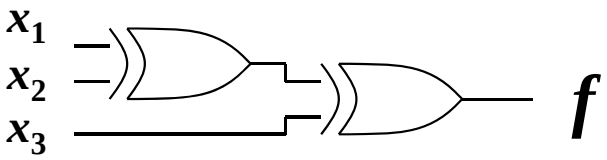




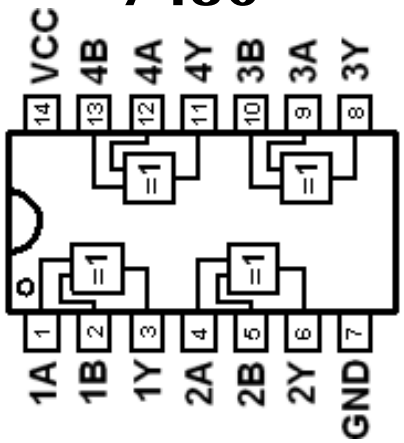
f

x_1 \ $x_2 x_3$	00	01	11	10
0	0	(1)	0	(1)
1	(1)	0	(1)	0

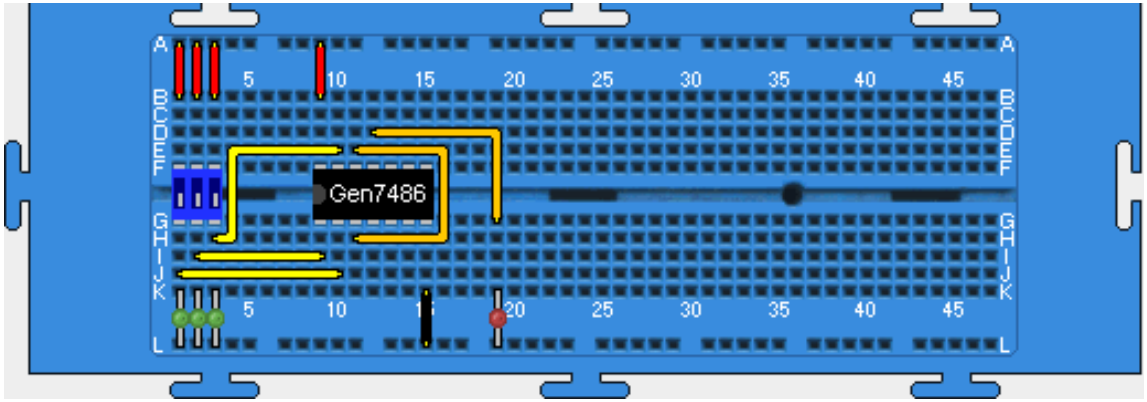
Enklare med XOR-grindar



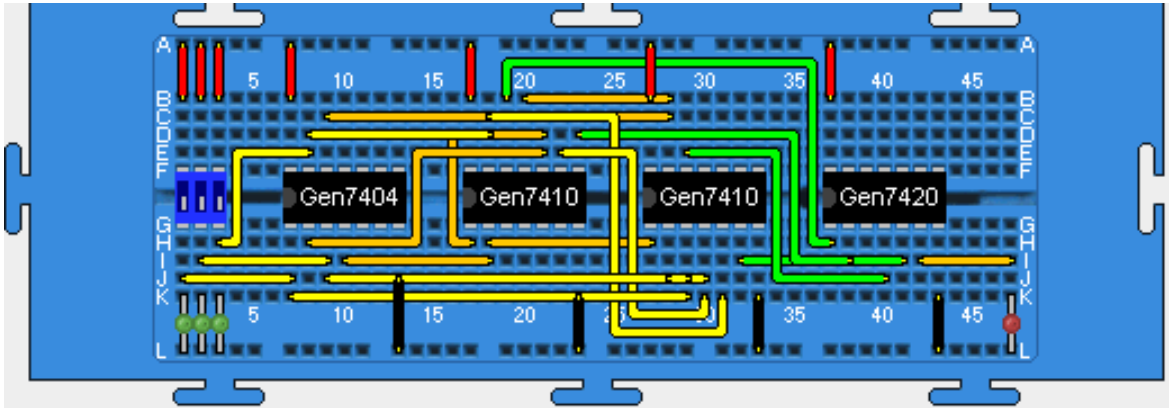
7486



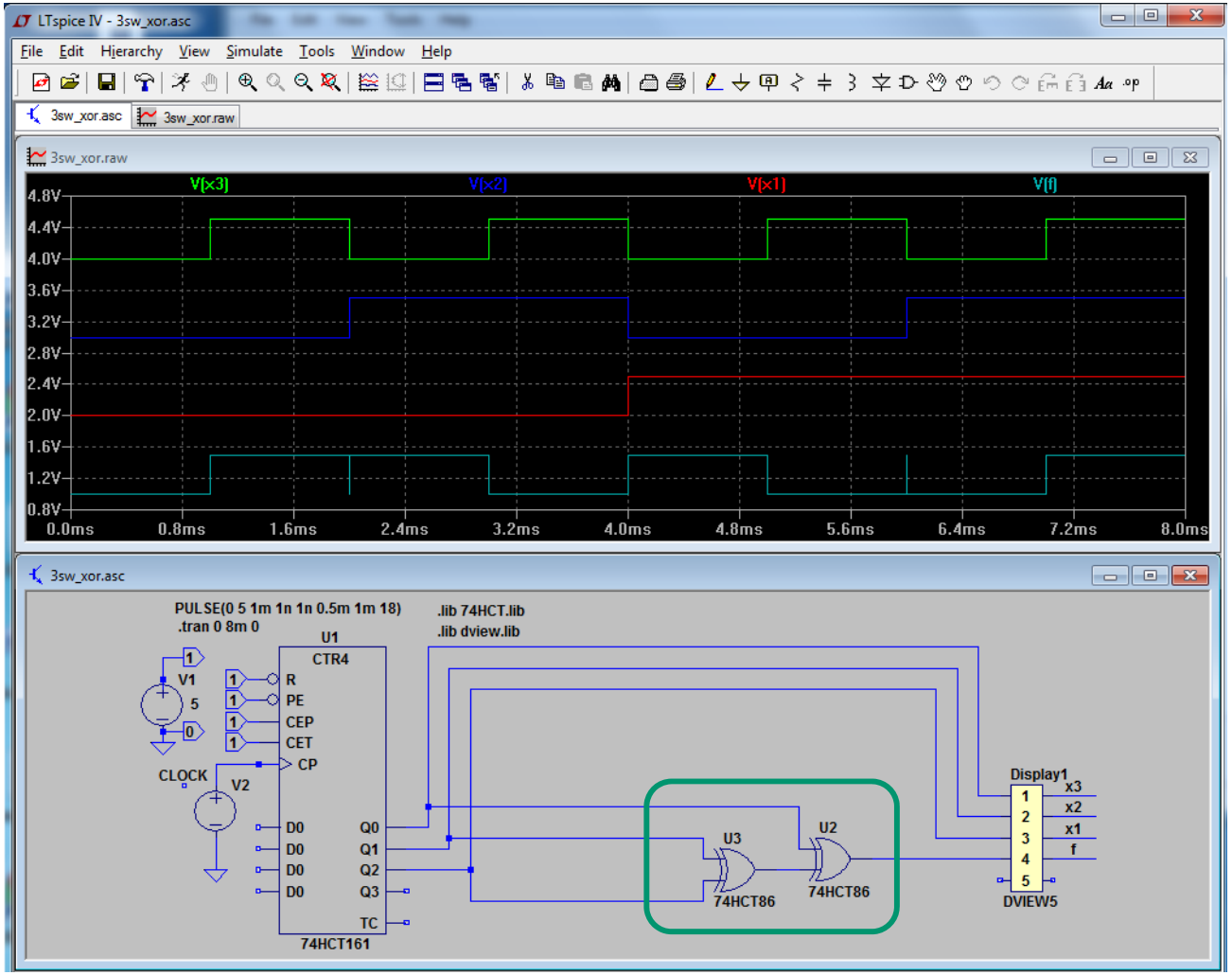
Med XOR-grindar:

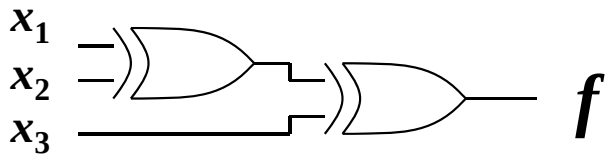


(Med NAND-grindar:)



Enklare med XOR-grindar

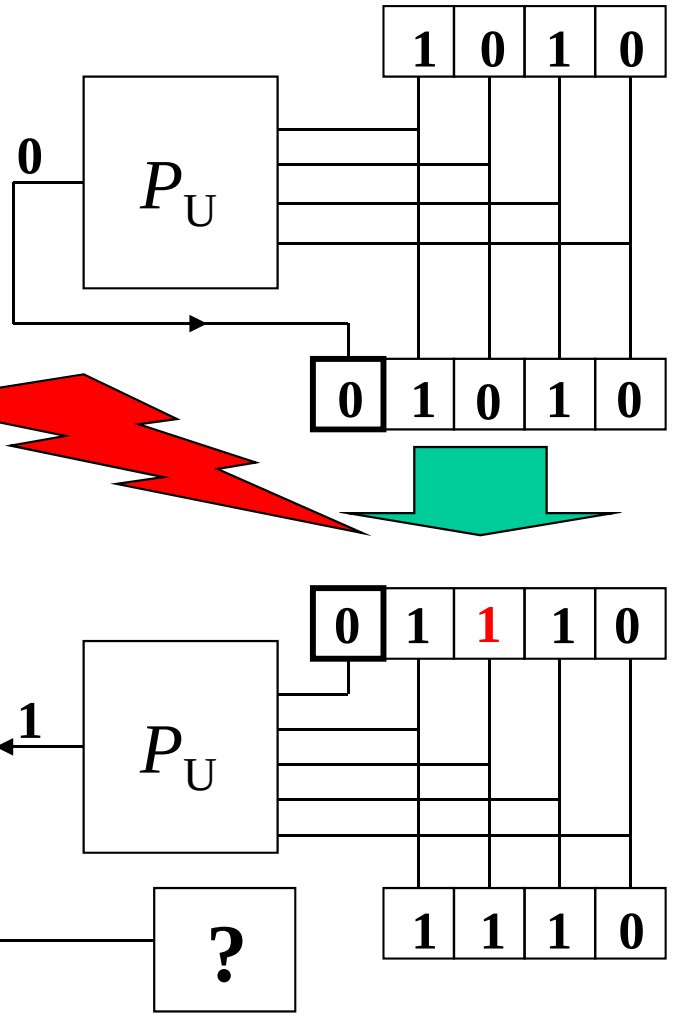




x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(Paritetscheck)

Med paritetsfunktioner kan man kontrollera om data blivit stört eller ej.



Data - orginal

Paritetsbit läggs till

Data som överförs har alltid jämn paritet!

Paritetscheck kontrollerar om udda antal ”1:or”

LARM!
En bit ändrad!
Data har störts!

Data – ev fel

Mer **komposition**

- Komposition kan även användas för att konstruera *n*-bit-adderare

F6aritmetik2.pdf

- Man behöver *n* hel-adderare för att konstruera en *n*-bit-adderare
(eller 2·*n* halvadderare)

Inte ovanligt med 128-bits adderare!

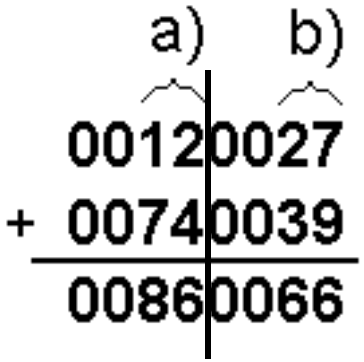
(Vem behöver en **128** bit adderare?)

2^{128} = 340282366920938463463374607431768211456 Kronor?

VLIW Very Long Instruction Word. *Kompilatorn* parar ihop lämpliga instruktioner och packar ihop dem för att exekveras *samtidigt*.

- **Matteläxa:**

a) addera 12+74 b) addera 27+39

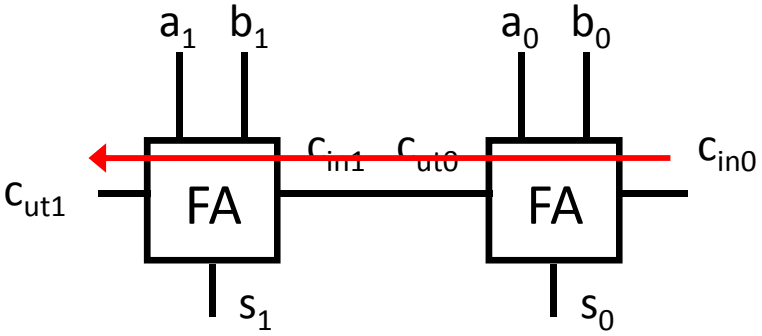


Varför *inte* göra hela matteläxan, uppgifterna a och b, på en och samma gång?

Svar: a) 86 och b) 66

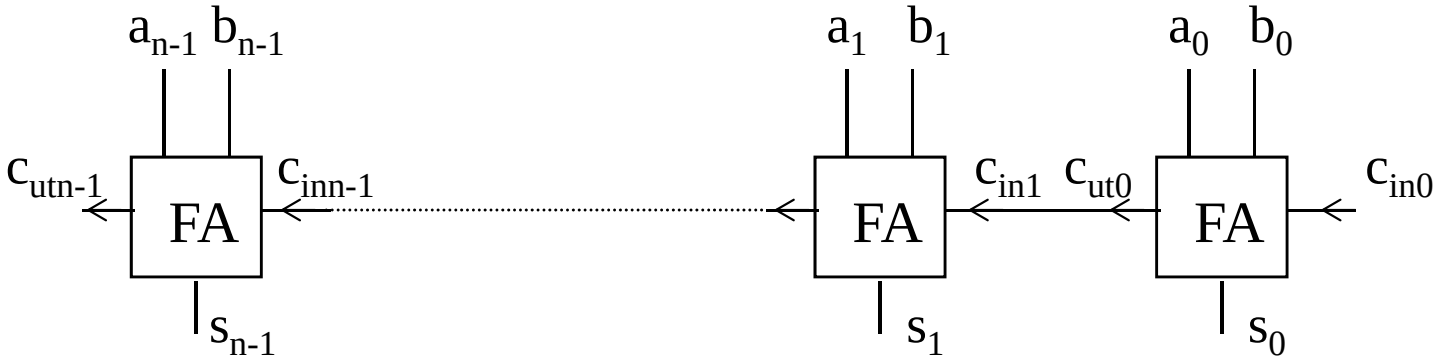
Den här tekniken används i Intel Itanium (EPIC, Explicit Parallel Instruction Code).

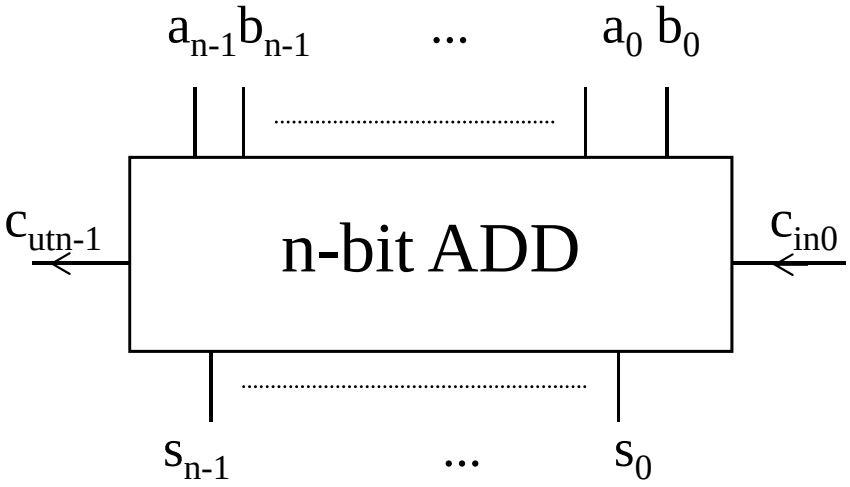
Ripple-Carry Adderare (RCA)



	c_{ut1}	c_{ut0}	c_{in0}
		b_1	b_0
+		a_1	a_0
		s_1	s_0

Ripple-Carry Adderare (RCA)

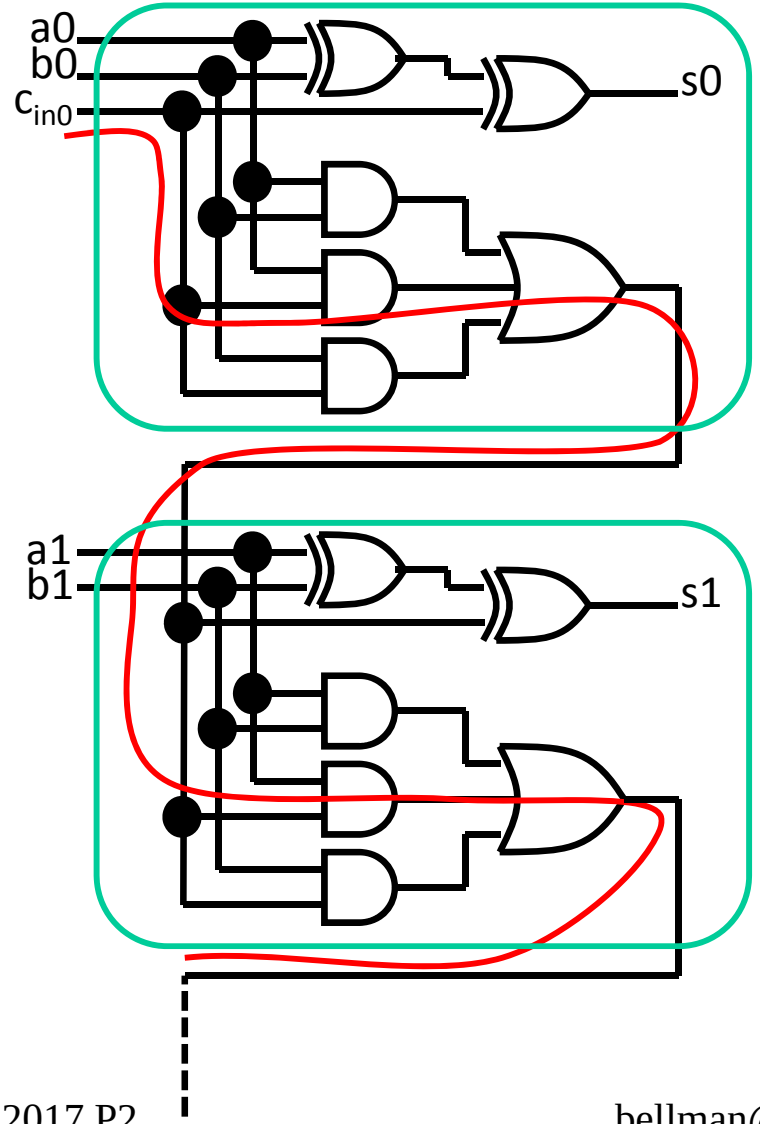




A_{FA} area för en Fulladder
t_{FA} fördröjning genom en Fulladder

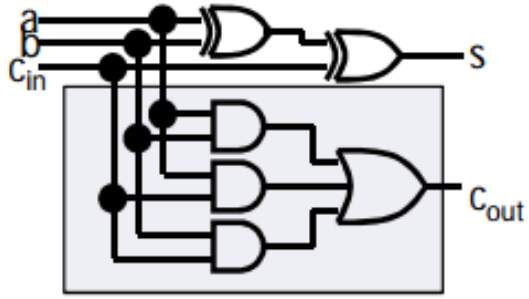
- **Tidsfördröjning från c_{in0} till c_{outn-1} blir hela $n \cdot t_{FA}$**
- **Totala arean blir $n \cdot A_{FA}$**

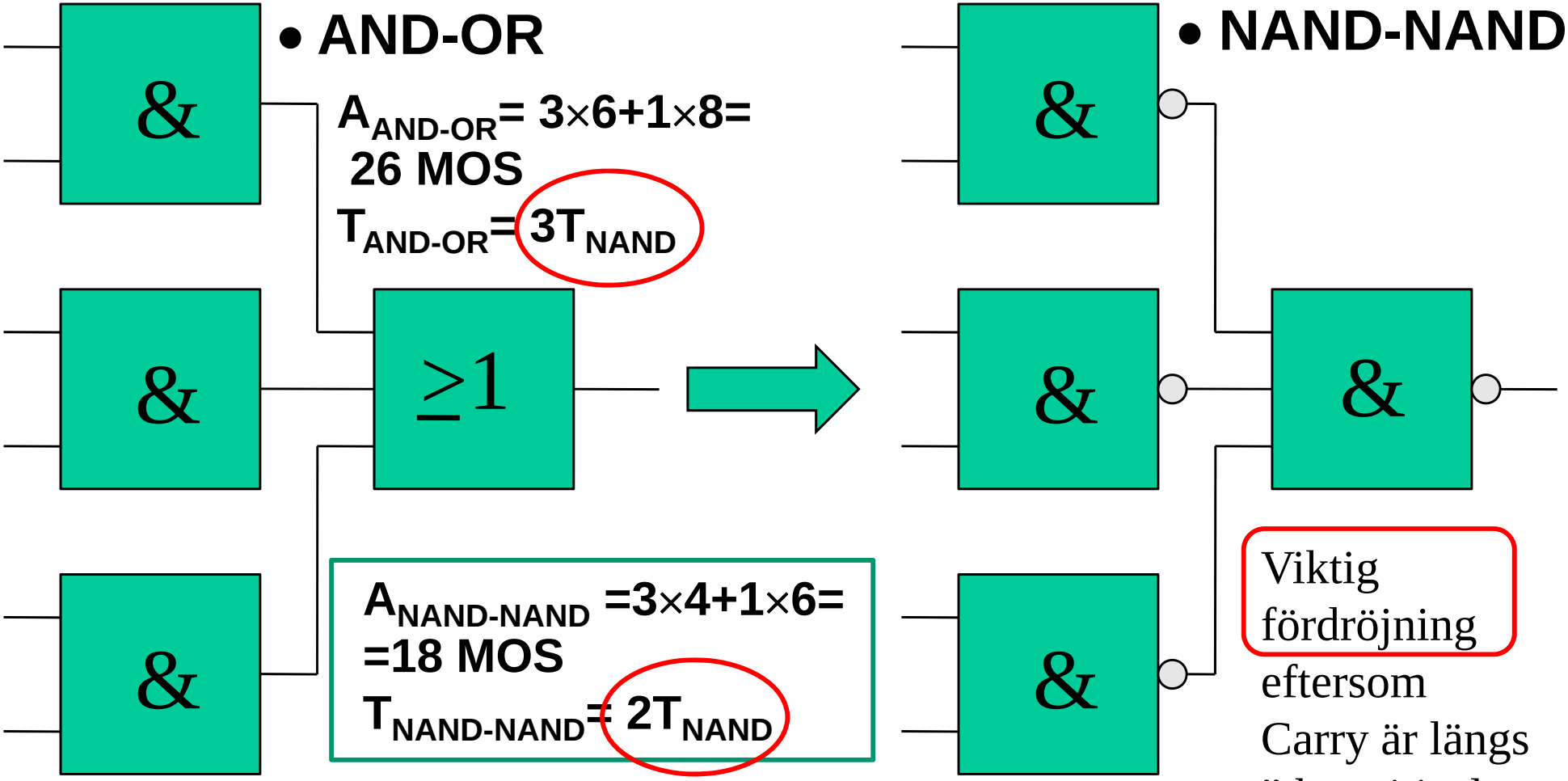
Ripple-Carry Adderare (RCA)



Critical path

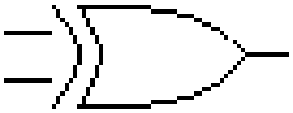
Heladderarens Carry-funktion

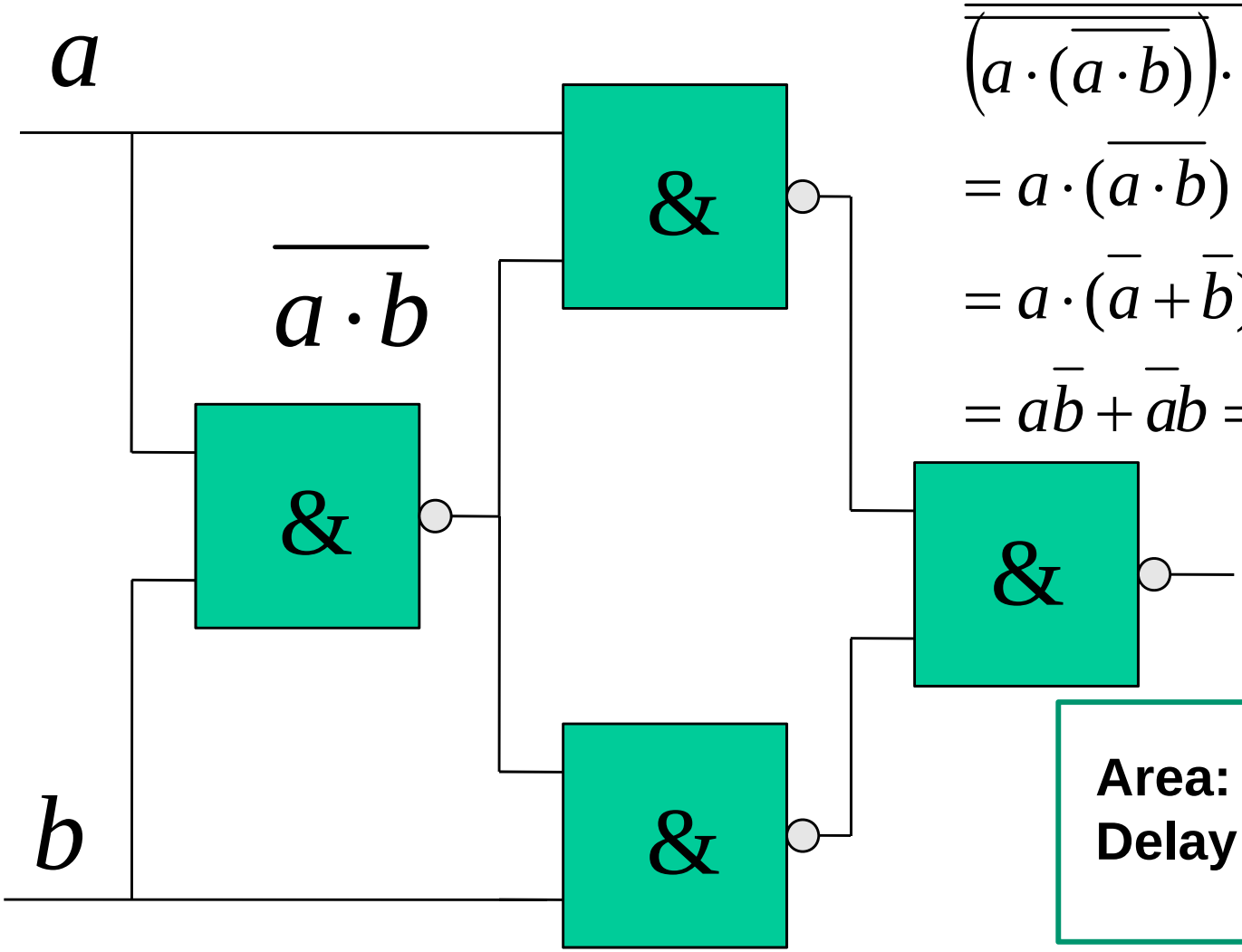




Viktig fördröjning eftersom Carry är längs ”the critical path”

XOR med NAND





$$\overline{\left(a \cdot (\overline{a \cdot b})\right) \cdot \left(b \cdot (\overline{a \cdot b})\right)} =$$

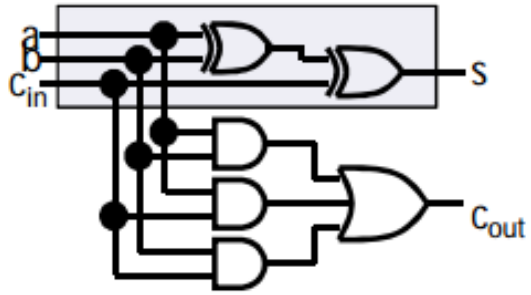
$$= a \cdot (\overline{a \cdot b}) + b \cdot (\overline{a \cdot b}) =$$

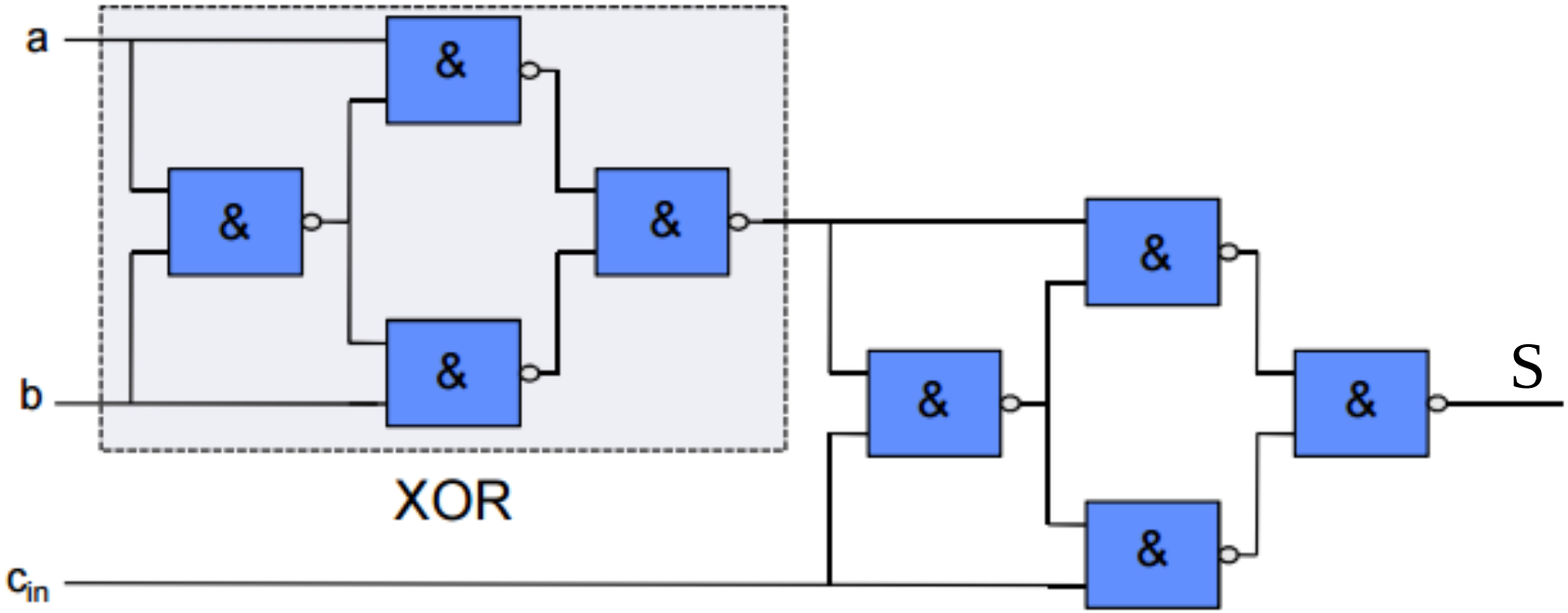
$$= a \cdot (\bar{a} + \bar{b}) + b \cdot (\bar{a} + \bar{b}) =$$

$$= a\bar{b} + \bar{a}b = a \oplus b$$

Area: A_{XOR}=16 MOS
Delay: T_{XOR}=3T_{NAND}

XOR (3) med NAND





Area: A_{XOR}=32 MOS
Delay: $T_{XOR}=6T_{NAND}$

Mindre viktig fördröjning eftersom S *inte* är längs ”the critical path”

Kan vi konstruera en snabbare adderare?

- Fördröjningen i en ripple-adderare växer proportionellt med antalet bitar
- För 32 bitar kan fördröjningen blir mycket stor (≈65 gate delays)

generate- och propagate- funktionerna

Carry-kedjan kan beskrivas med två funktioner:

- **Generate** g_i (carry-out $c_{i+1} = 1$ ifall $g_i = 1$)

$$g_i = x_i y_i$$

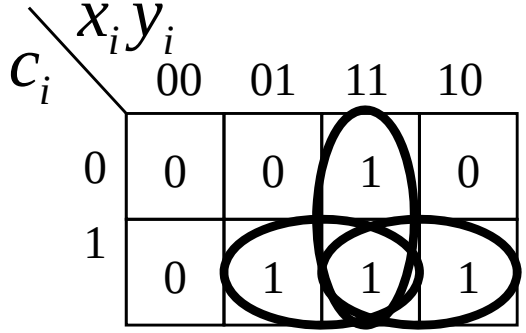
- **Propagate** p_i (carry-out $c_{i+1} = 1$ ifall $c_i = 1$ och $x_i = 1$ eller $y_i = 1$)

$$p_i = x_i + y_i$$

$$p_i = x_i \oplus y_i$$

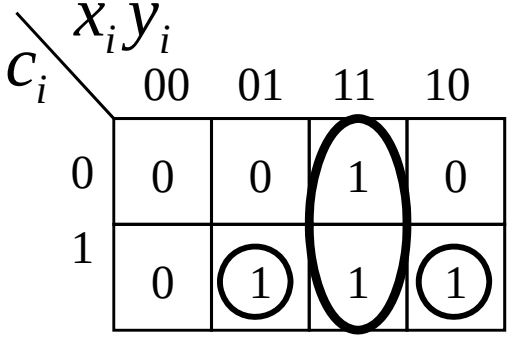
Fungerar också!

$$c_{i+1} = g_i + p_i c_i$$



$$c_{i+1} = x_i y_i + c_i x_i + c_i y_i$$

$$= x_i y_i + c_i (x_i + y_i)$$



$$c_{i+1} = x_i y_i + c_i (x_i \oplus y_i)$$

Carry-look-ahead funktion

$$g_i = x_i y_i$$
$$p_i = x_i + y_i$$

- **Carry-bit** c_0

$$c_1 = g_0 + p_0 c_0$$

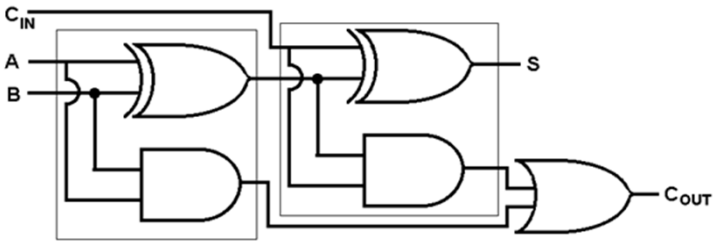
- **Carry-bit** c_1

$$c_2 = g_1 + p_1 c_1$$
$$= g_1 + p_1(g_0 + p_0 c_0)$$
$$= g_1 + p_1 g_0 + p_1 p_0 c_0$$

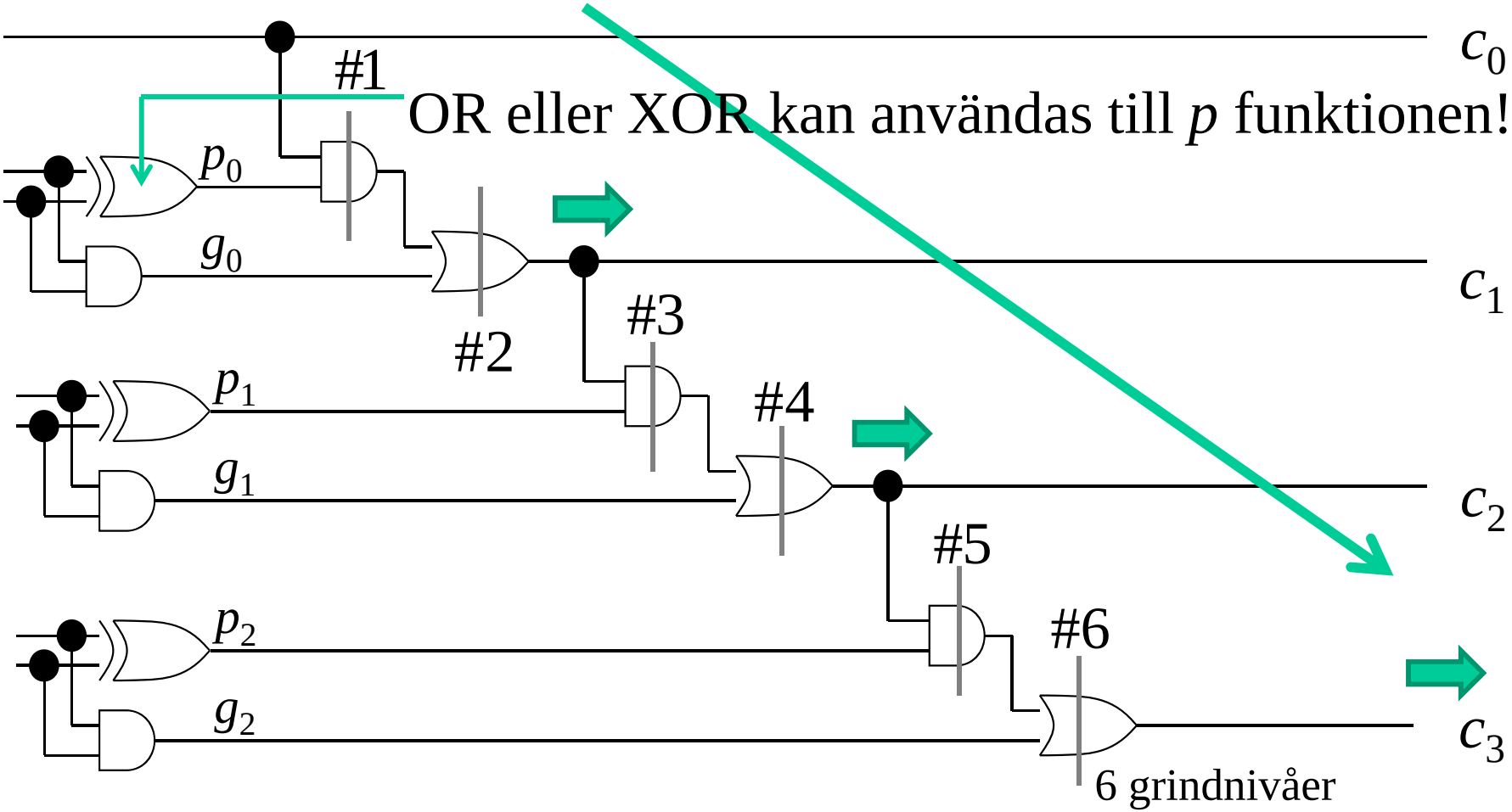
Propagate funktionen från ”föregående” bitar kan snabbas upp genom att genereras parallellt

Bara två logiknivåer behövs …

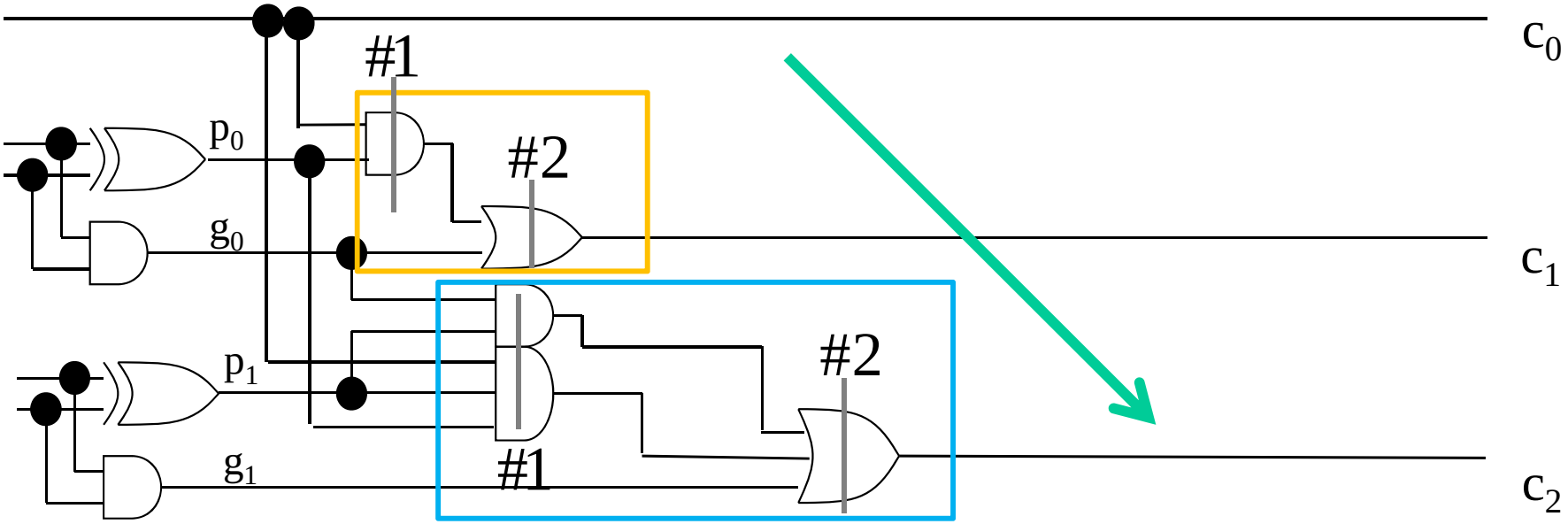
Carry-kedjan i ripple-adderaren



½ + ½



Snabbare implementering av Carry-kedjan (med g och p)



$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

- Två-nivåers nät

(n = 8) med två logiknivåer

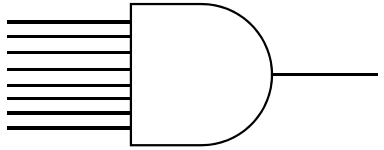
$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$

Snabbt, men otympligt med så här många ingångar!

Ooops!

$$c_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0$$

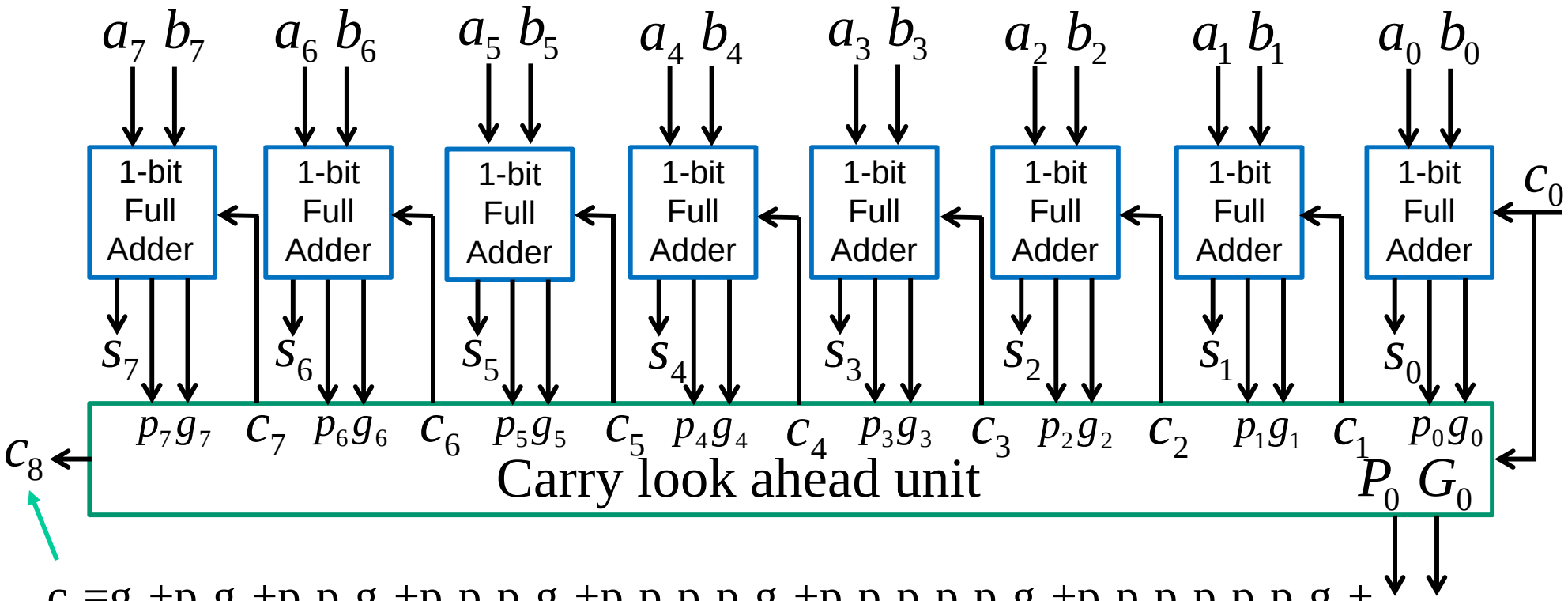


Det behövs nog ”specialgrindar” till detta …

Carry-lookahead adder (CLA)

8-bit adder

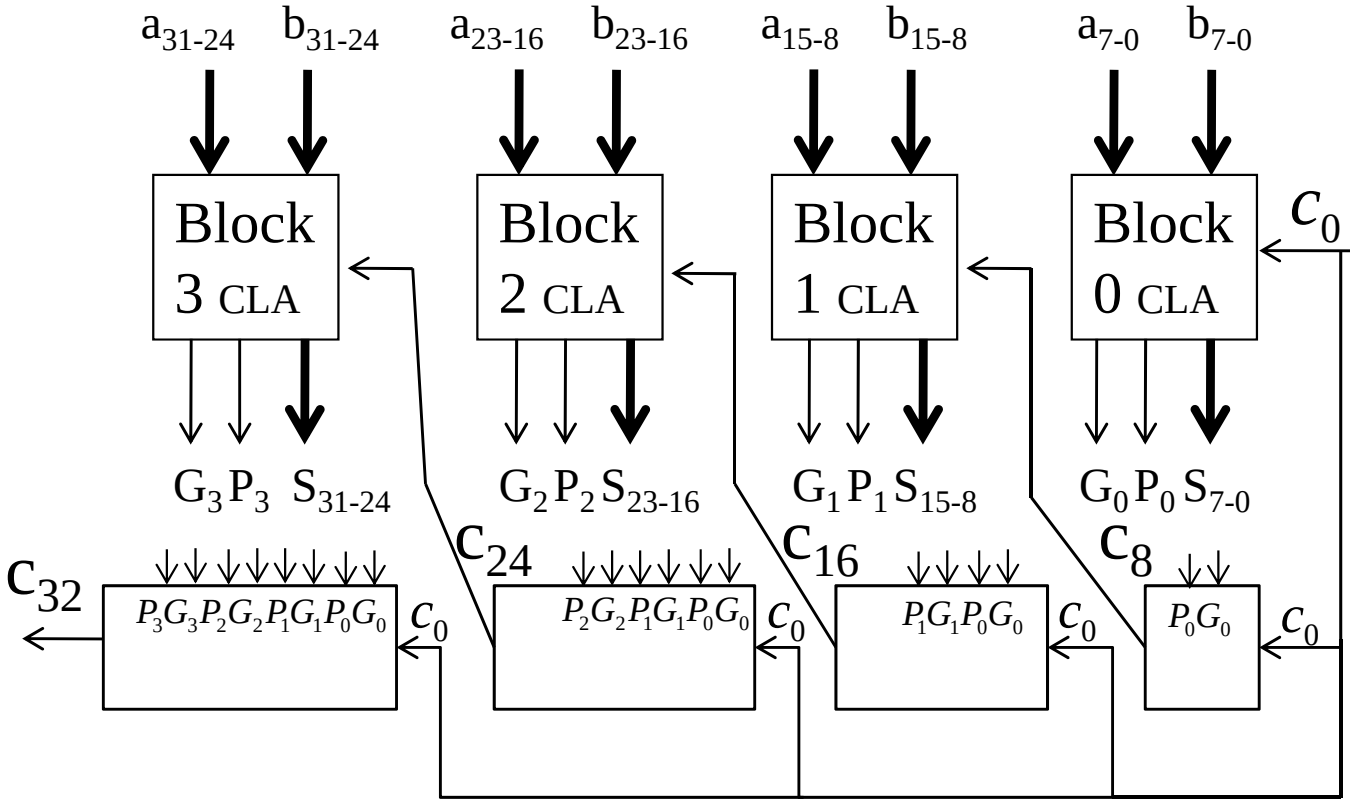
$$c_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0 = a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 + b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$



$$c_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0$$

Hierarkisk expansion (BV sid 277)

32-bit adder



32-bits adderare. Varje block består av en 8-bits CLA.

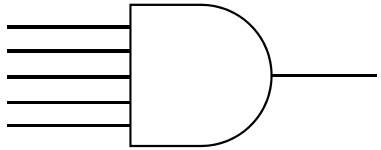
First level
four 8-bit adders with Carry lookahead

Second level
four Carry lookahead units

Hierarkisk expansion nivå 2

Carry-bitar från en andra nivåns Carry lookahead enheter

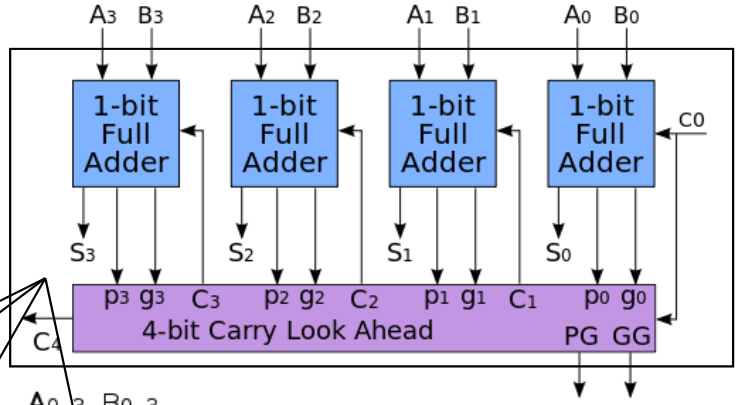
$$C_8 = G_0 + P_0 c_0$$
$$C_{16} = G_1 + P_1 G_0 + P_1 P_0 c_0$$
$$C_{24} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$C_{32} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$



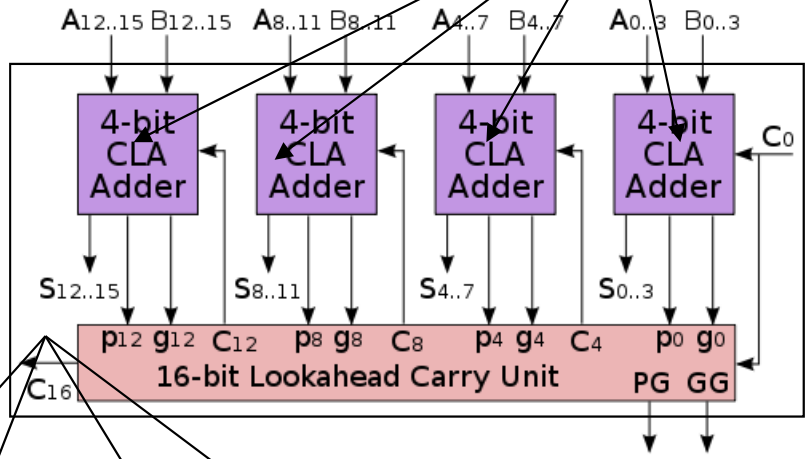
etc.

Med Carry lookahead enheter i flera nivåer kan adderare med fler bitar konstrueras. Med flera nivåer kan också grindar färre ingångar användas, men varje nivå bidrar med extra grindfördröjningar.

(Carry look ahead 3 level hierarchy)

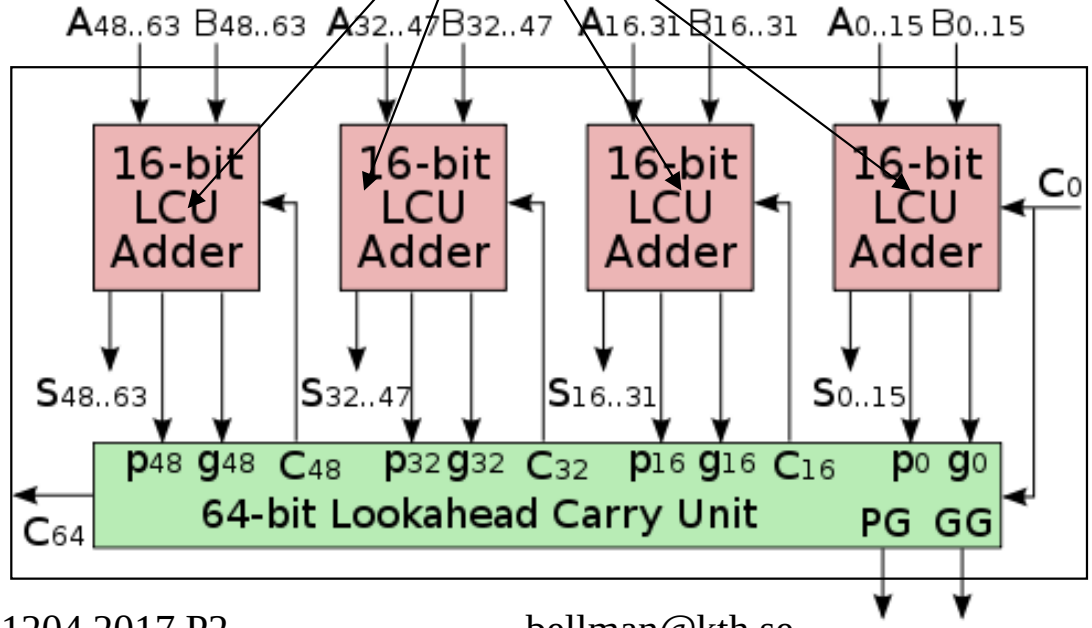


• 5-ingångars grindar med 2 nivåer



16-bit LCU

• 5-ingångars grindar med 2 nivåer



• 64-bit adder

• 5-ingångars grindar med 2 nivåer

64-bit LCU

Totalt sex grindnivåer

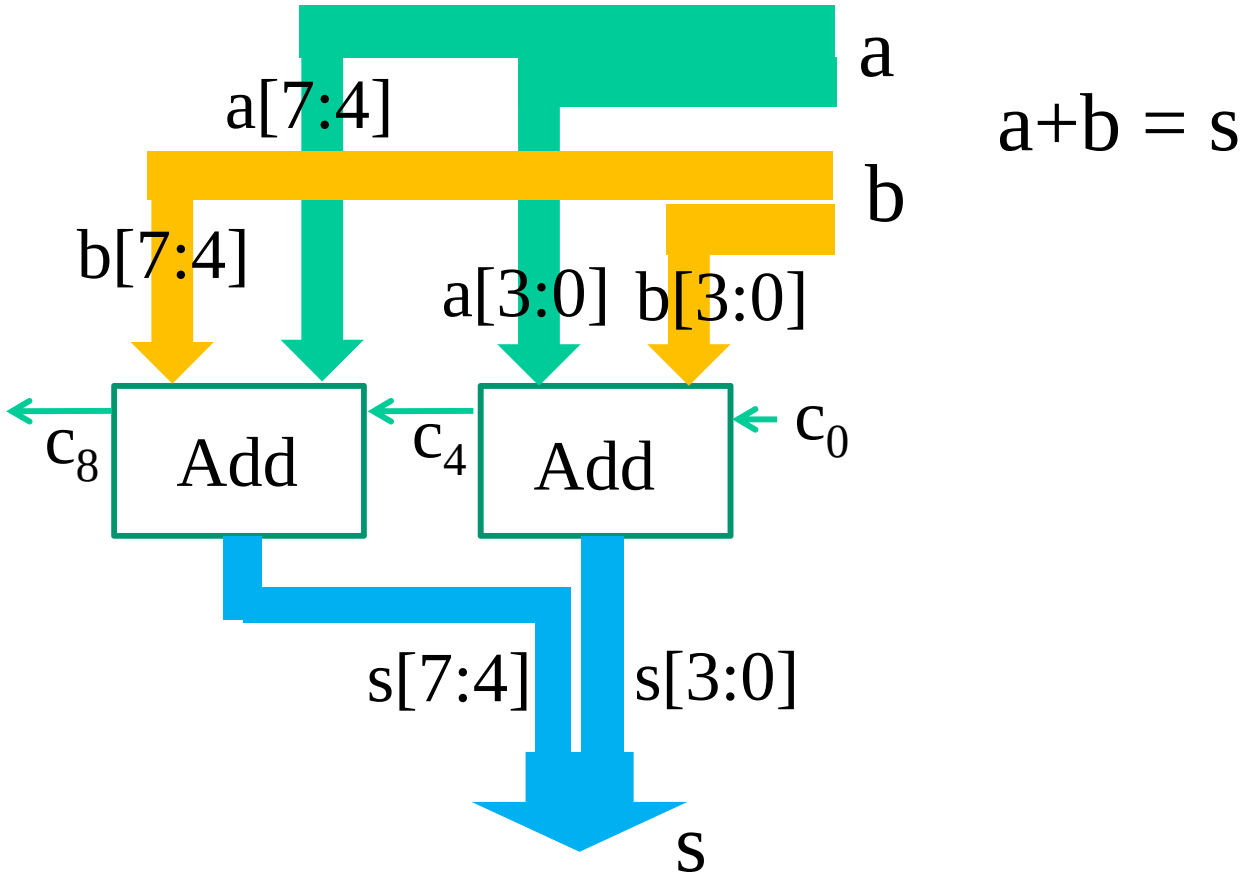
Carry-Select-Adder (CSA)

Idé

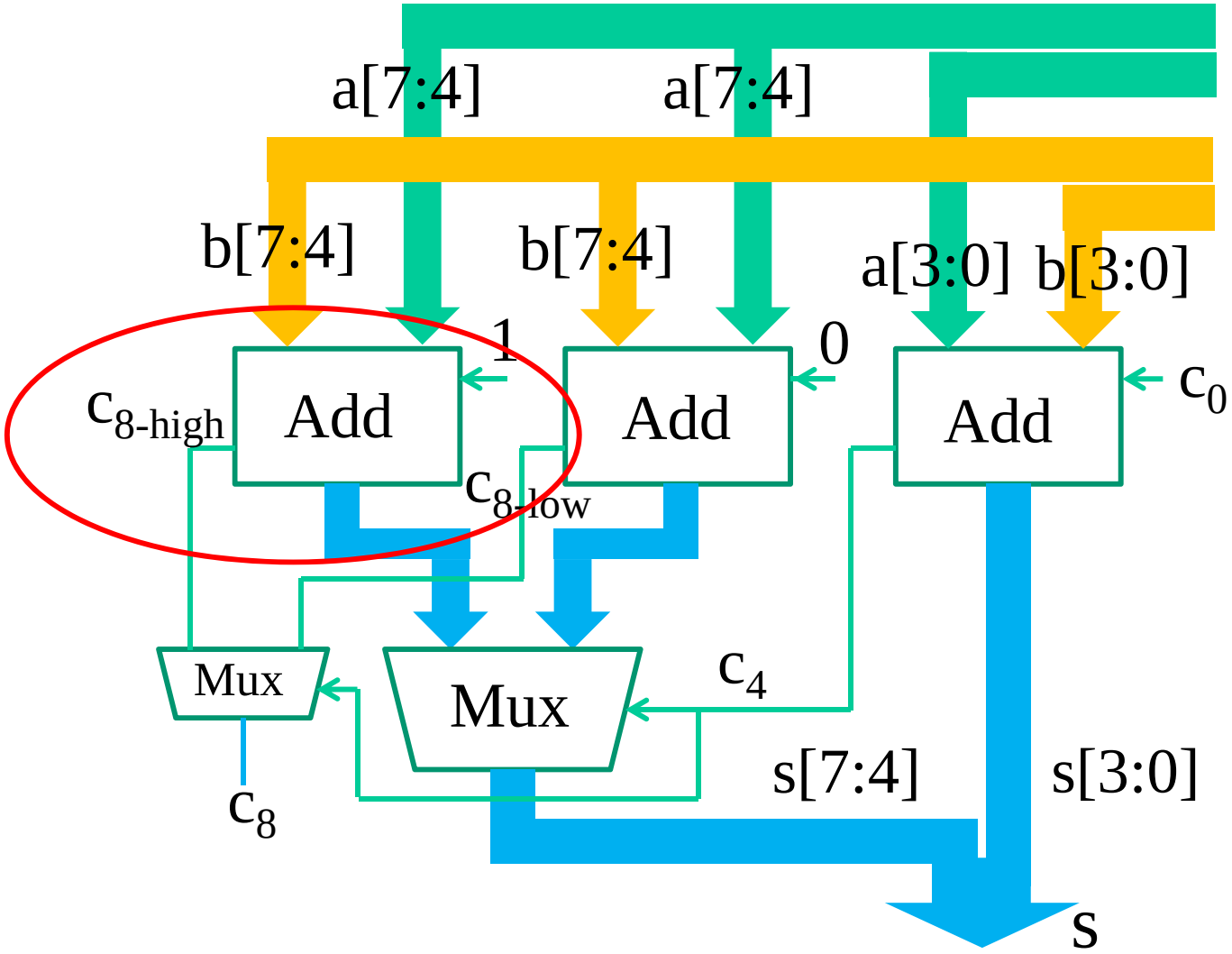
- **Man delar upp en adderare i *två steg* med samma antal bitar**
- **För att snabba upp processen så räknar man ut resultatet av det andra steget i förväg för två fall**
  - **Carry-in = 0**
  - **Carry-in = 1**

- **När beräkningen av carry-biten är klar för det första steget, så väljer man resultatet av det andra steget beroende på carry-bitens värde!**

8-bit (4+4) ripple carry Adder



4+4/4 bit Carry-Select-Adder



a+b = s

- Dubbla hastigheten
- 1/3 extra area

Jämförelser

- Ripple-Carry Adder

  $T(n) = n*2.5*T_{NAND}$, $A = n*12.5A_{NAND}$

  $T(4) = \mathbf{14}T_{NAND}$, $A(4) = \mathbf{50}\,A_{NAND}$

- Carry-Lookahead Adder (4 bits)

  $T(4) = \mathbf{\sim 8}T_{NAND}$, $A(4) = 43A_{NAND} + 4*4*A_{NAND} = \mathbf{\sim 60}A_{NAND}$

- Carry-Select Adder (**8** bits)

  $T(8) = \sim(\mathbf{8+4})T_{NAND}$, $A(8) = \sim(\mathbf{120+20})A_{NAND}$

Vilken är den bästa adderaren?

Det finns inget entydigt svar!

- **Ripple-adderaren tar *minst plats* men är *långsam***
- **Carry-lookahead-adderaren tar *mycket plats* men är *snabb***
- **Carry-select-adderaren är en *kompromiss***

Man måste göra en *trade-off* mellan area och speed

Subtraktion

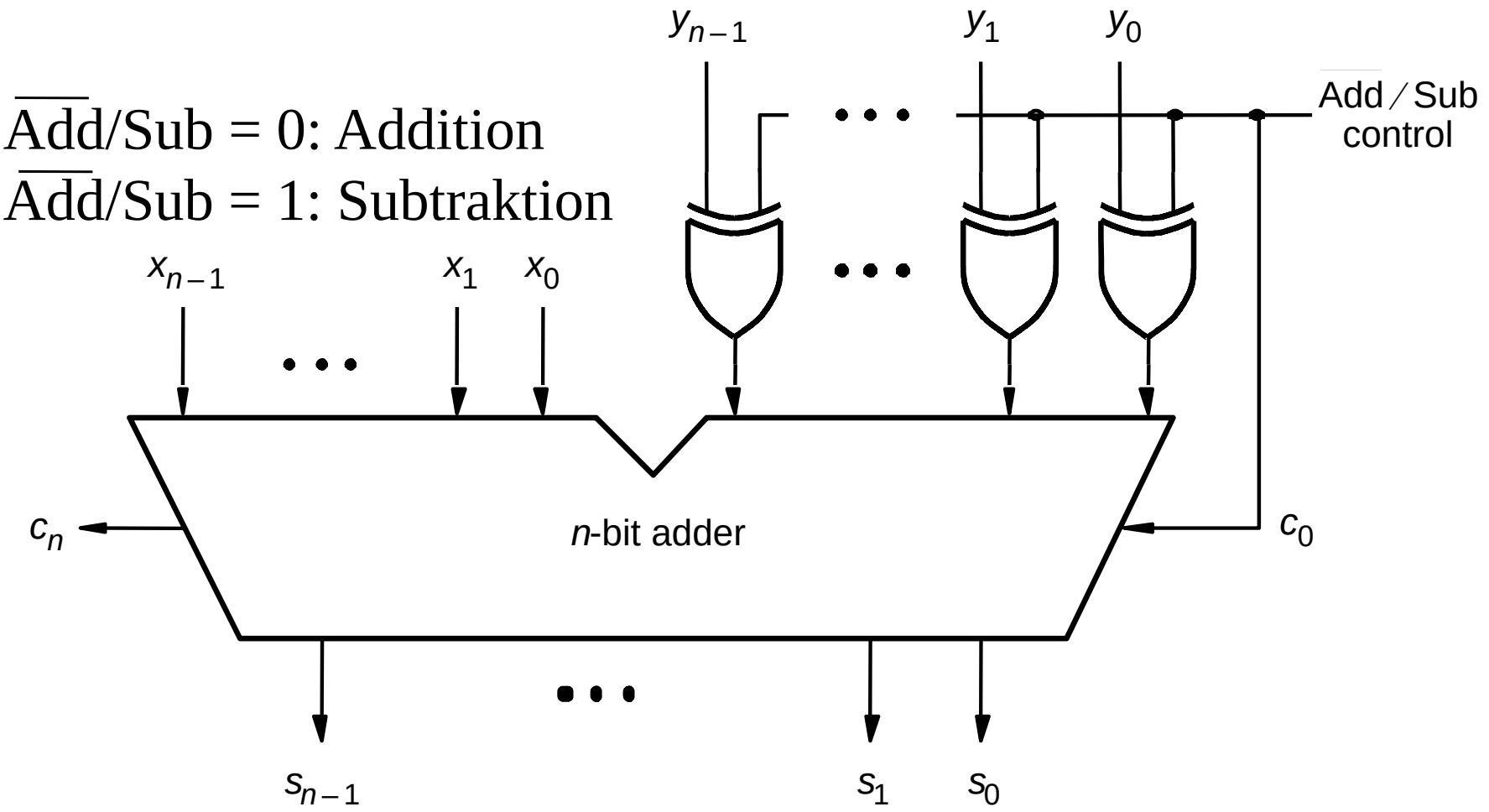
Subtraktion kan göras genom addition med två komplementet

- **Invertera alla bitar av den andra operanden**
- **Addera 1**

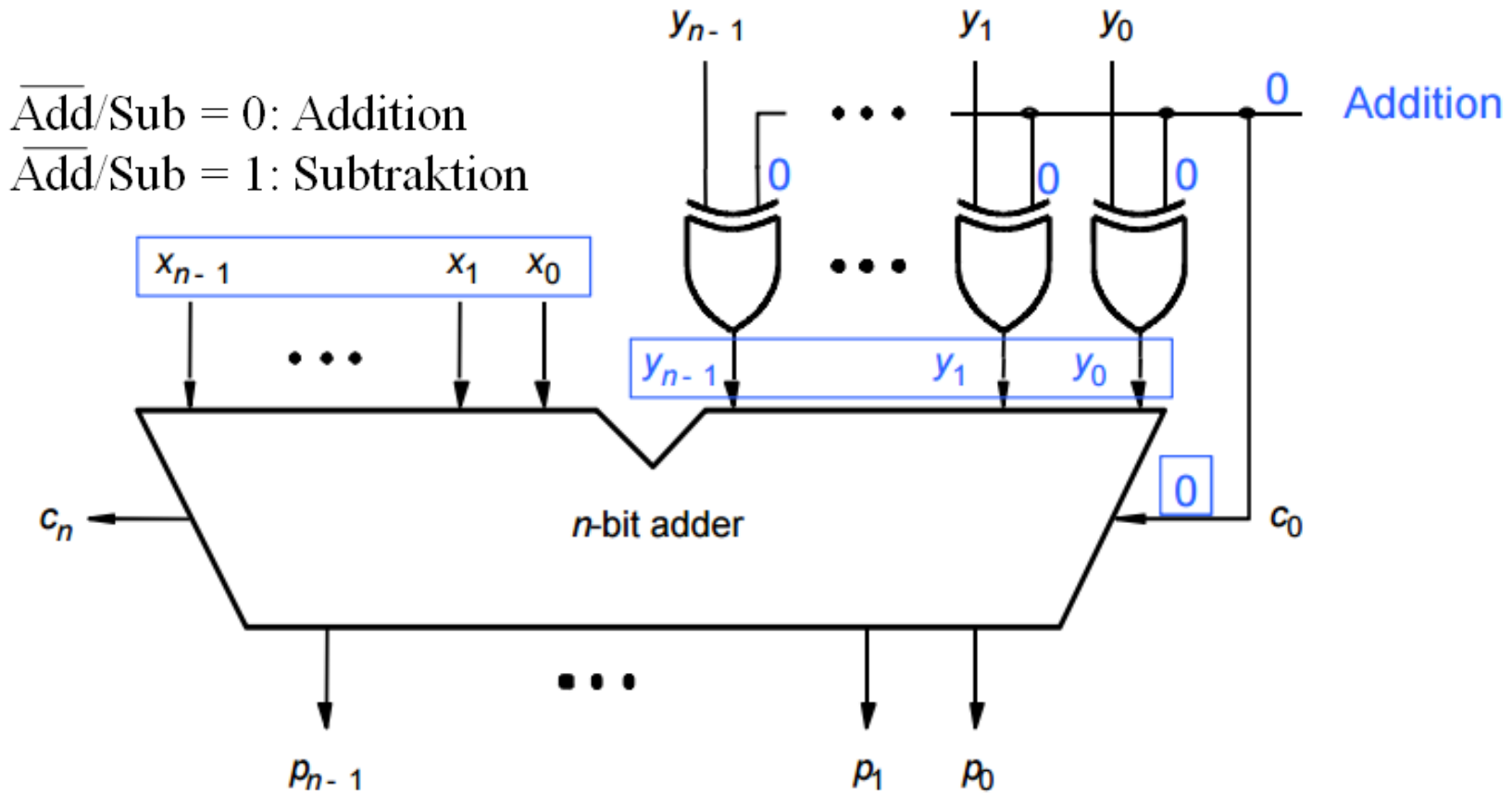
Add/sub-enheten

$\overline{\text{Add}}$/Sub = 0: Addition

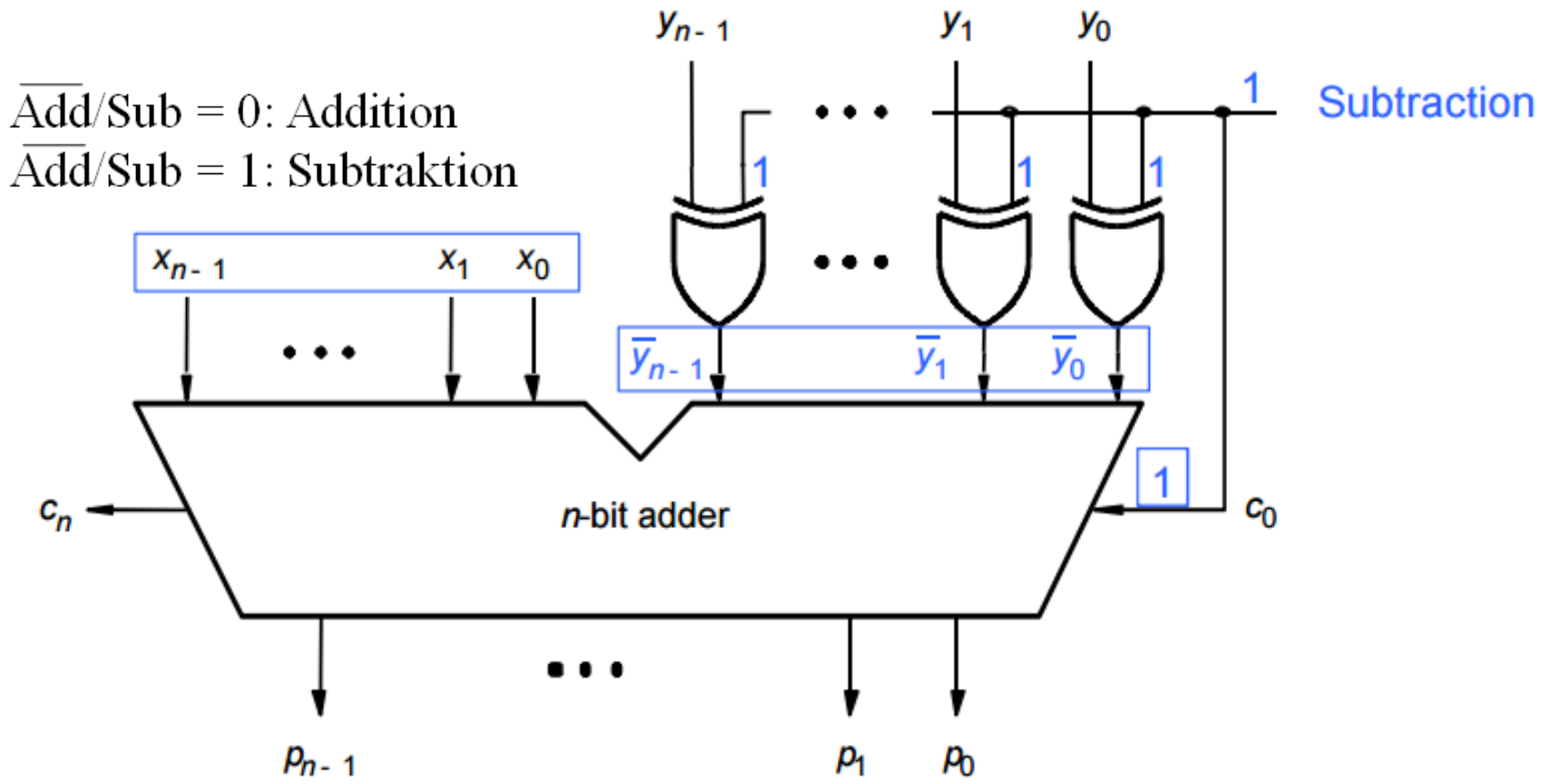
$\overline{\text{Add}}$/Sub = 1: Subtraktion



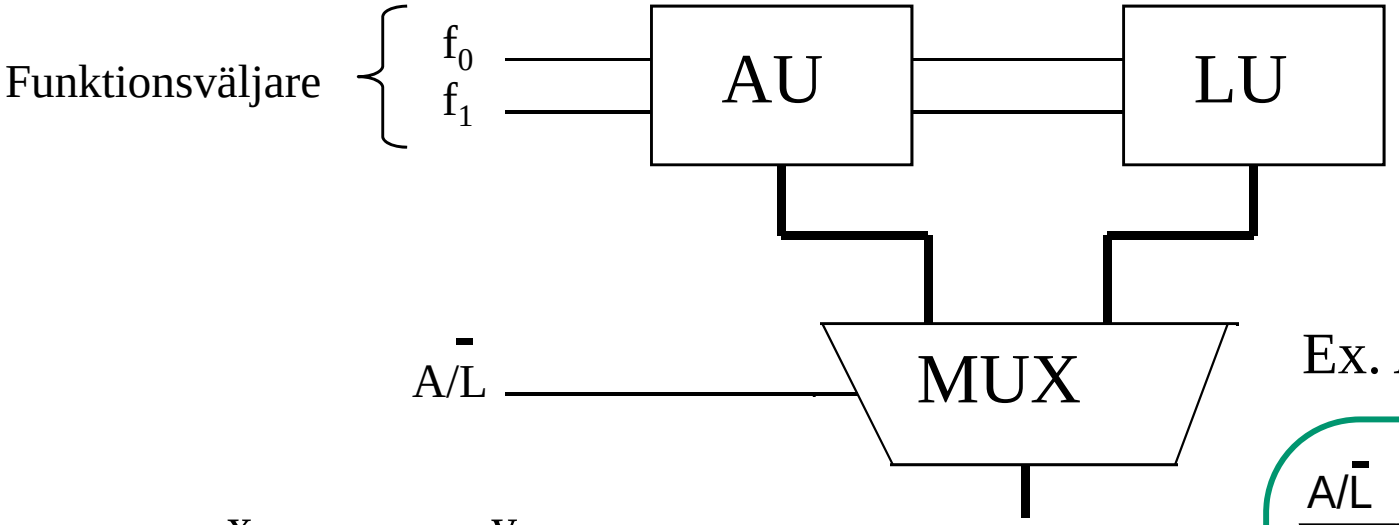
Add

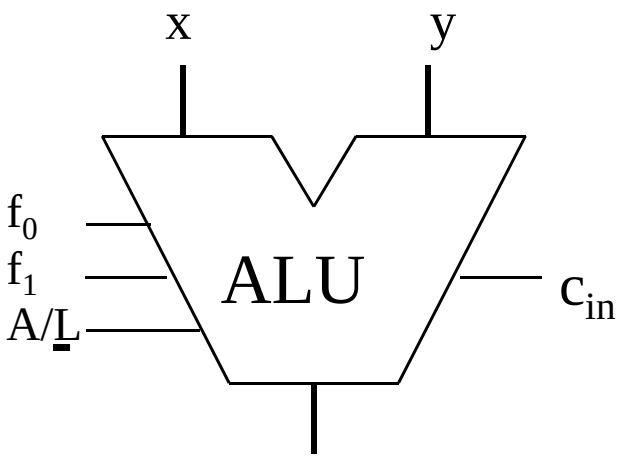

Add/Sub = 0: Addition
Add/Sub = 1: Subtraktion
y_{n-1}
y_1
y_0
0
Addition
0
0
0
x_{n-1}
x_1
x_0
y_{n-1}
y_1
y_0
0
c_n
n-bit adder
c_0
p_{n-1}
p_1
p_0

Sub


Add/Sub = 0: Addition
Add/Sub = 1: Subtraktion
y_{n-1}
y_1
y_0
1
Subtraction
1
1
1
x_{n-1}
x_1
x_0
$\overline{y}_{n-1}$
$\overline{y}_1$
$\overline{y}_0$
1
c_n
n-bit adder
c_0
p_{n-1}
p_1
p_0

Arithmetic Logic Unit (ALU)





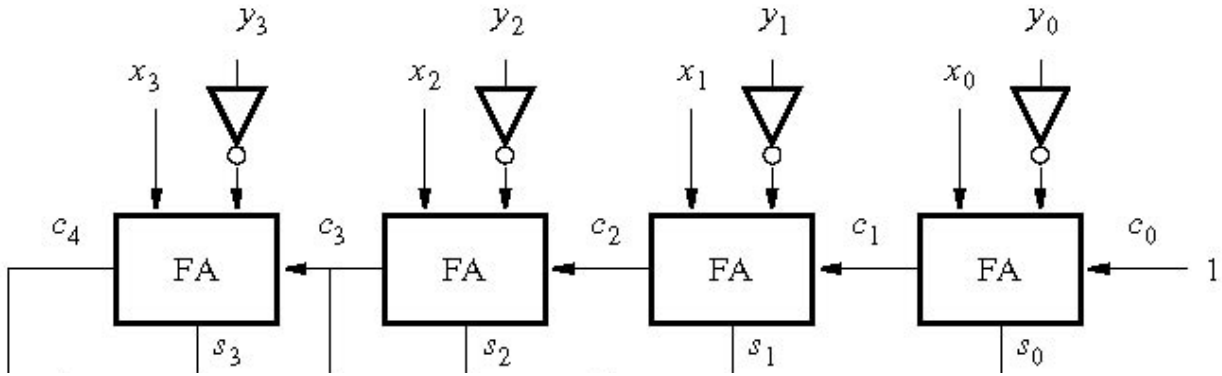
Processorer brukar alltid ha en **ALU**, inte bara en adderare.

Ex. ALU specification

$\overline{A/L}$	f1	f0	Funktion
0	0	0	x+y
0	0	1	$x+y+c_{in}$
0	1	0	x-y
0	1	1	$x-y-\overline{c_{in}}$
1	0	0	x or y
1	0	1	x and y
1	1	0	x xor y
1	1	1	inv x

Komparator och flaggor

Komparatorn, = jämförelse, implementeras som en subtraktionskrets

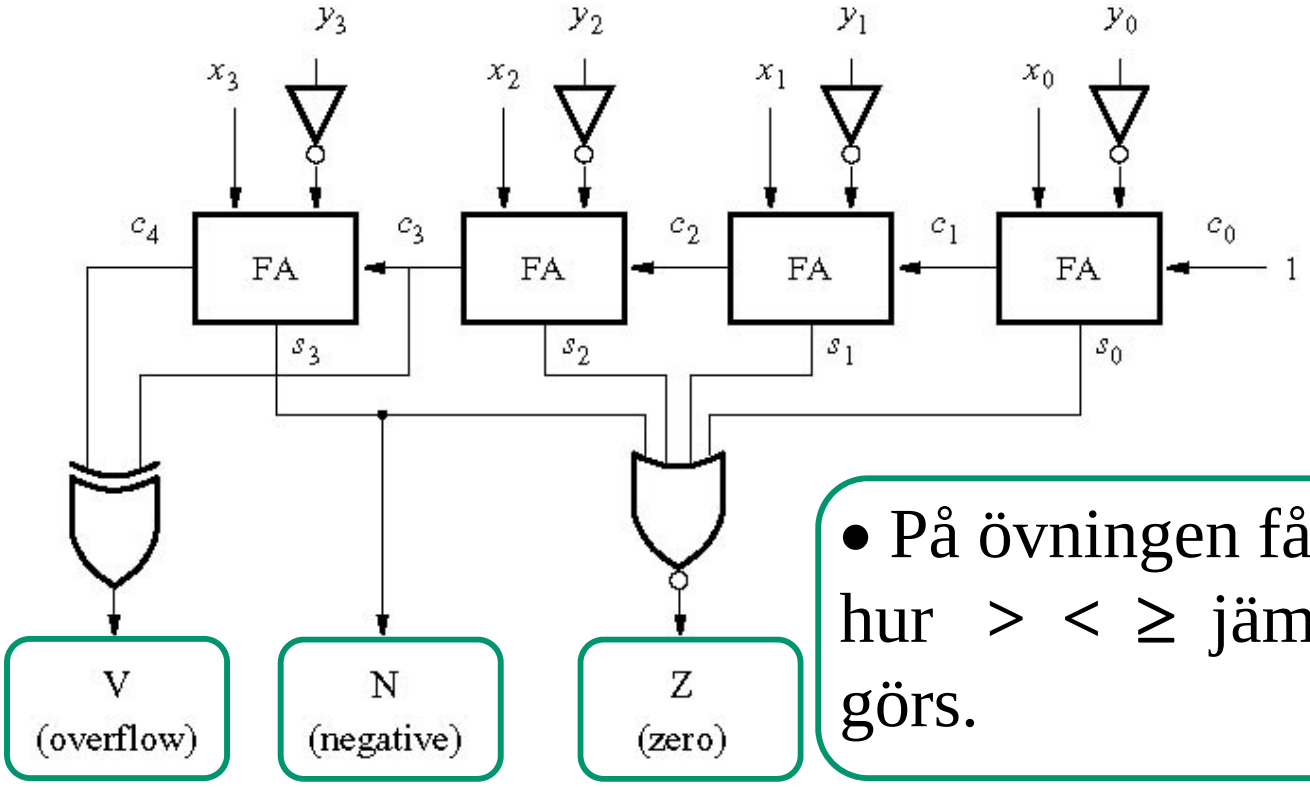


Flaggor:

?	?	?
V (overflow)	N (negative)	Z (zero)

Komparator

Komparatorn, = jämförelse, implementeras som en subtraktionskrets



Flaggor:

- På övningen får Du veta hur $>$ $<$ $\geq$ jämförelser görs.

Sammanfattning

Addition och subtraktion av heltal

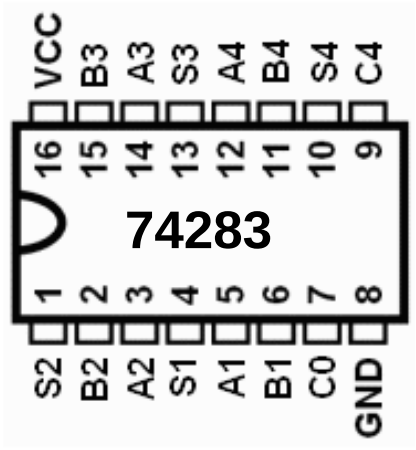
- **Två-komplementet**
- **Subtraktion av ett tal implementeras som addition med dess två-komplement**

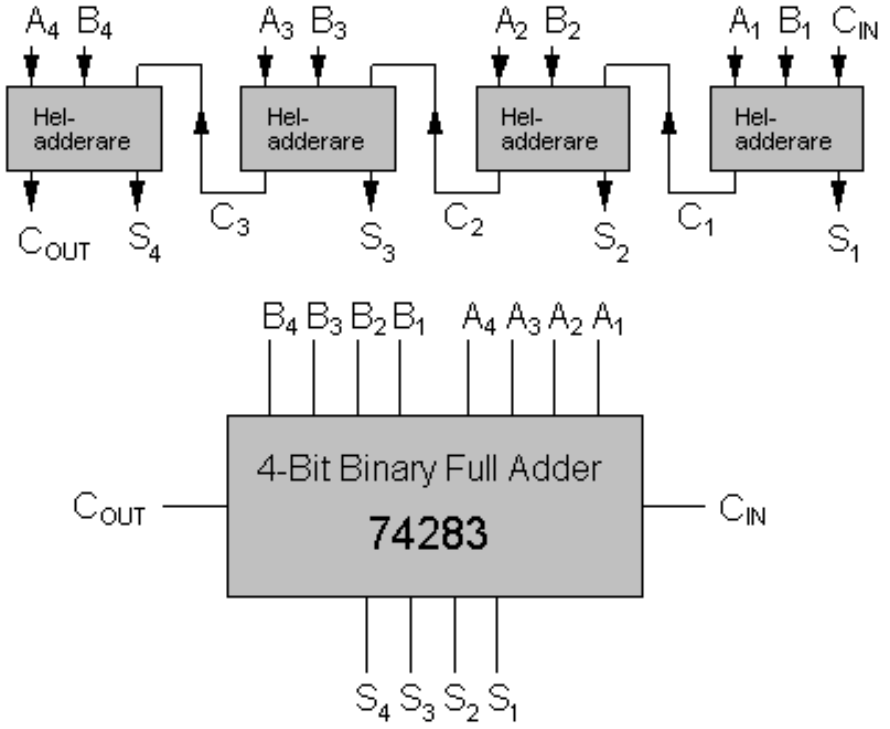
Trade-Off: Area mot Speed

Olika Adder-strukturer

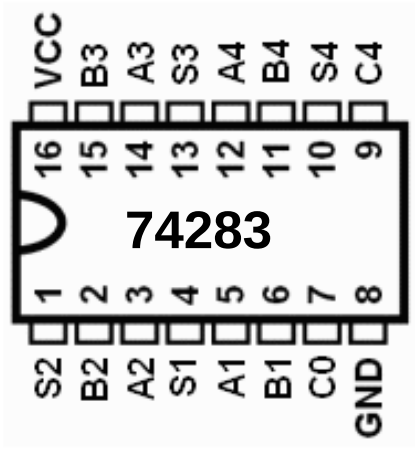
- **Ripple-Carry Adder (RCA)**
- **Carry-Lookahead Adder (CLA)**
- **Carry-Select Adder (CSA)**

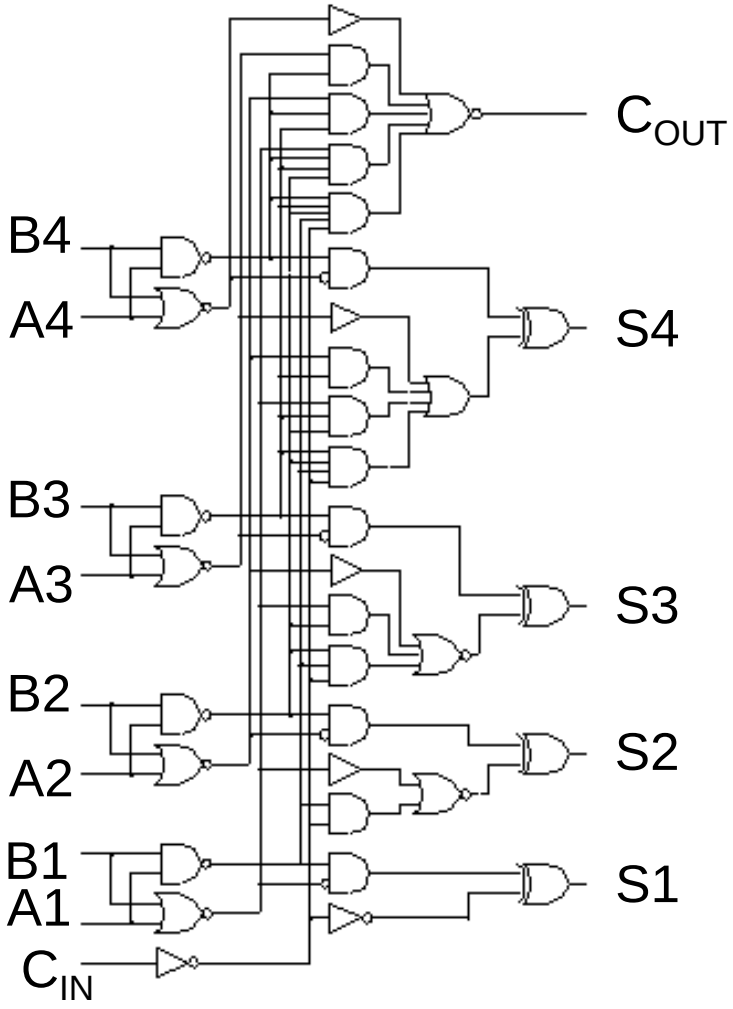
Addition vid Lab1



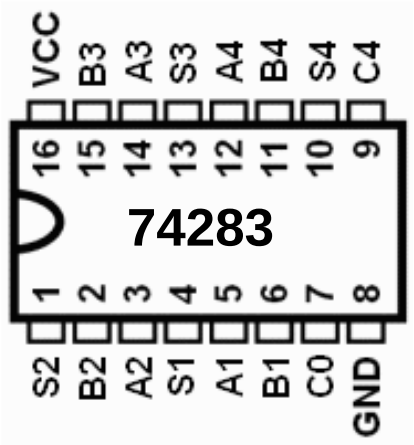


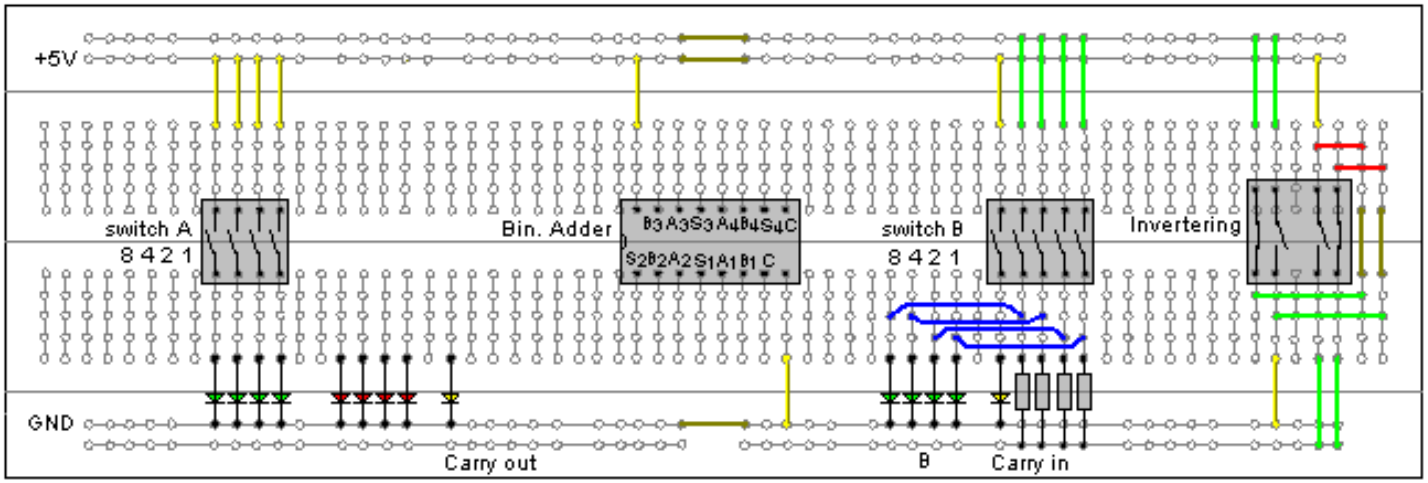
Addition vid Lab1





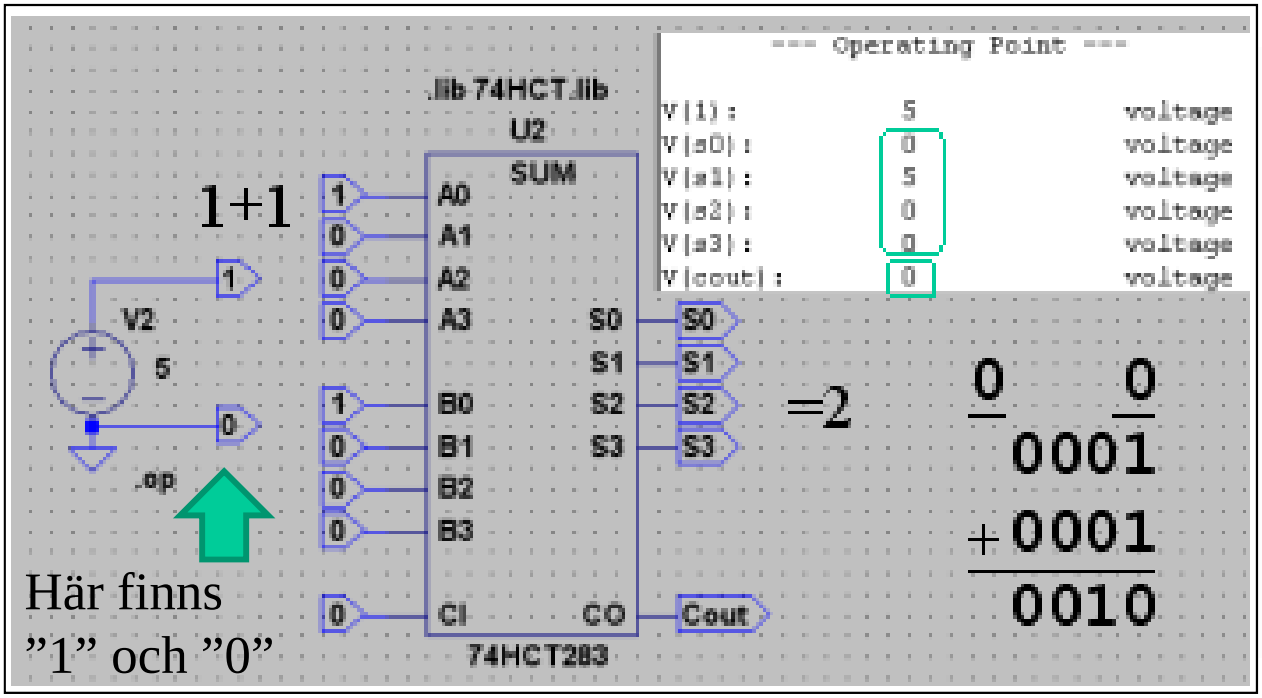
Addition vid Lab1

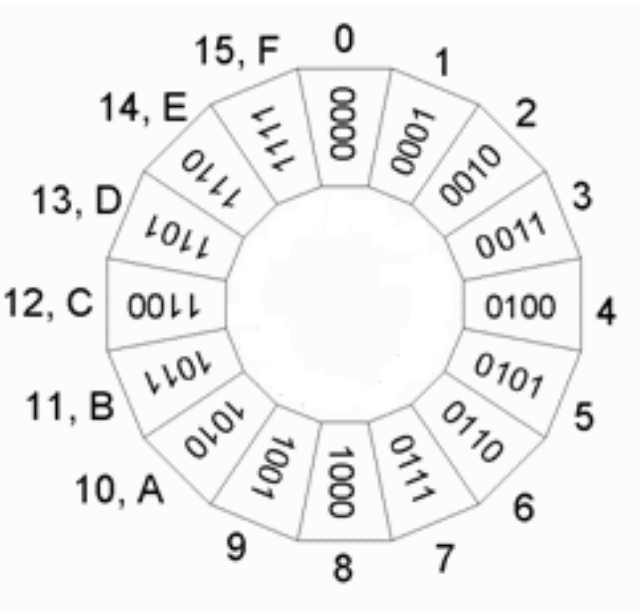




Kopplingsdäck med 4-bitsadderararkrets 74283.

Simulera addition

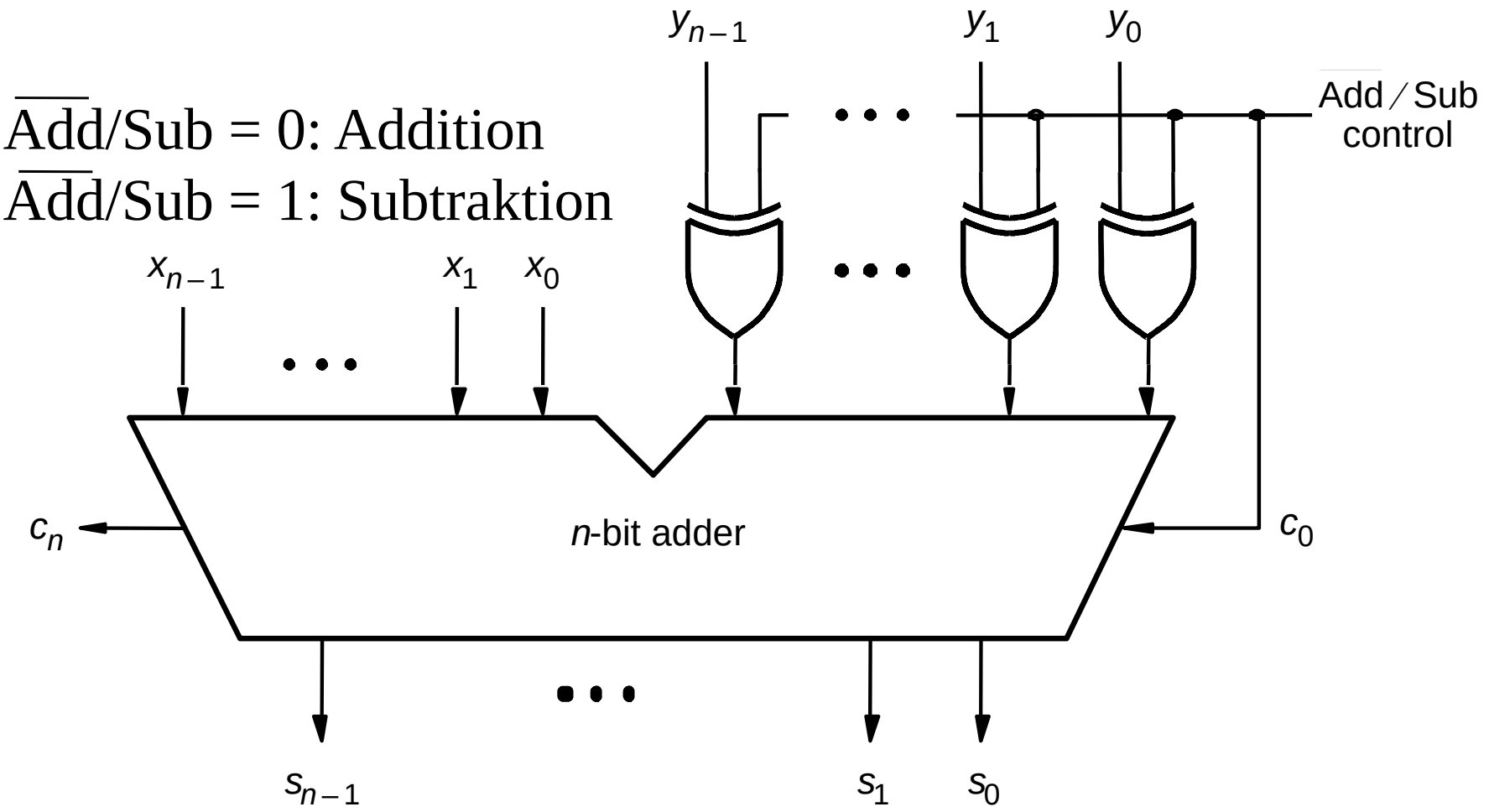




ADDn/SUB

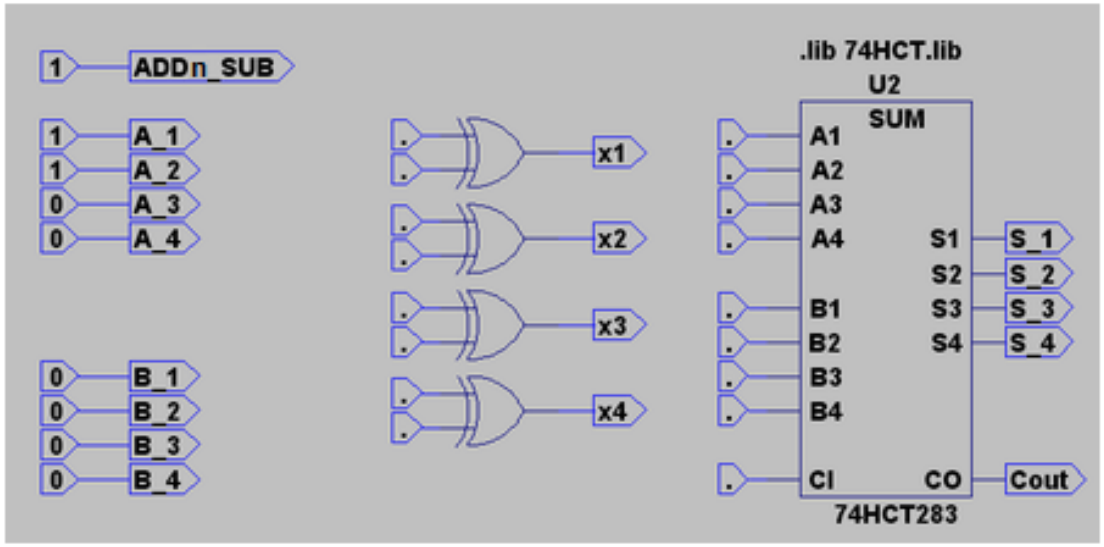
$\overline{\text{Add}}$/Sub = 0: Addition

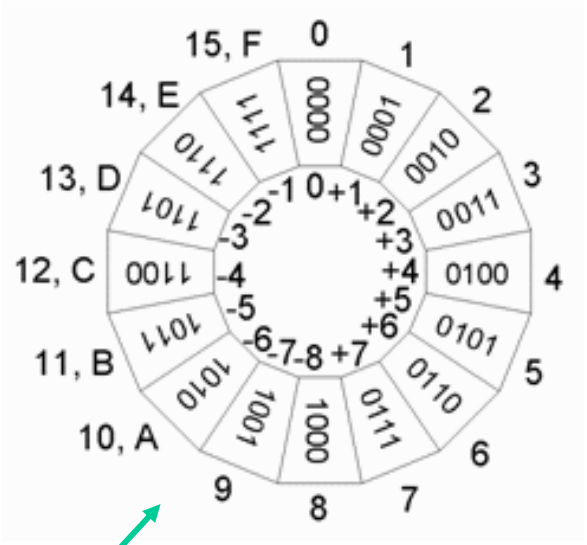
$\overline{\text{Add}}$/Sub = 1: Subtraktion



Subtraktion vid Lab1

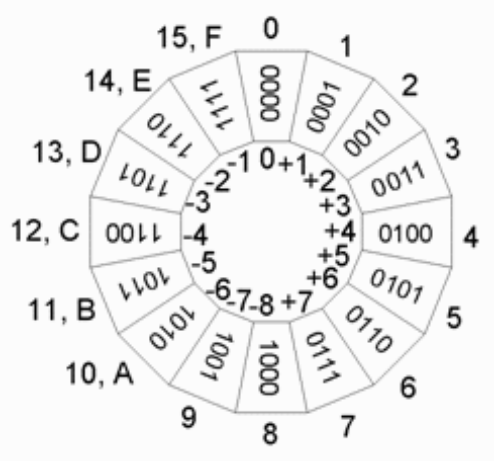
- ***Rita klart …***

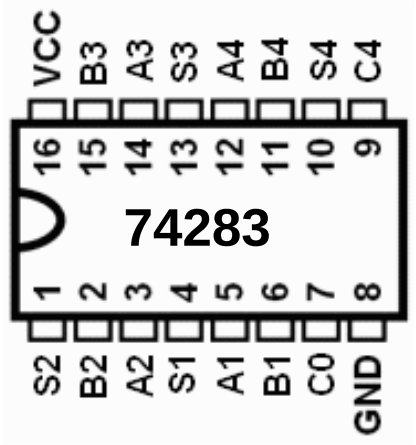




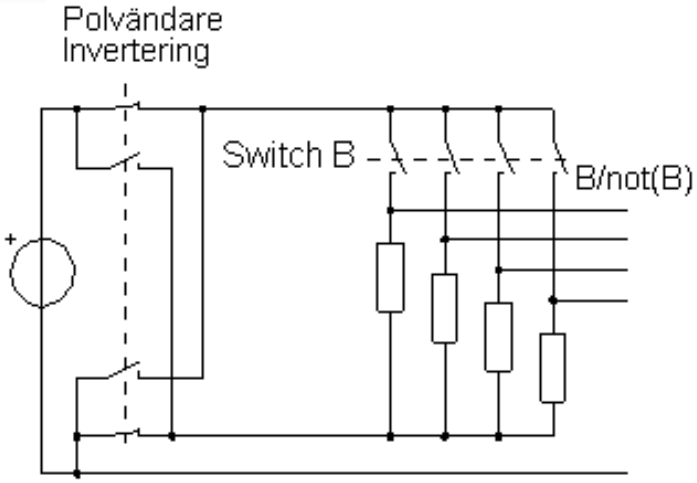
- Simulera ADD/SUB

Subtraktion vid Lab1

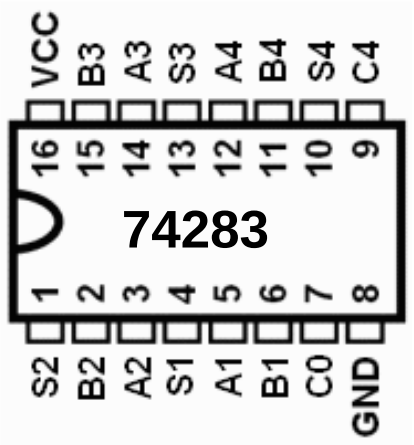




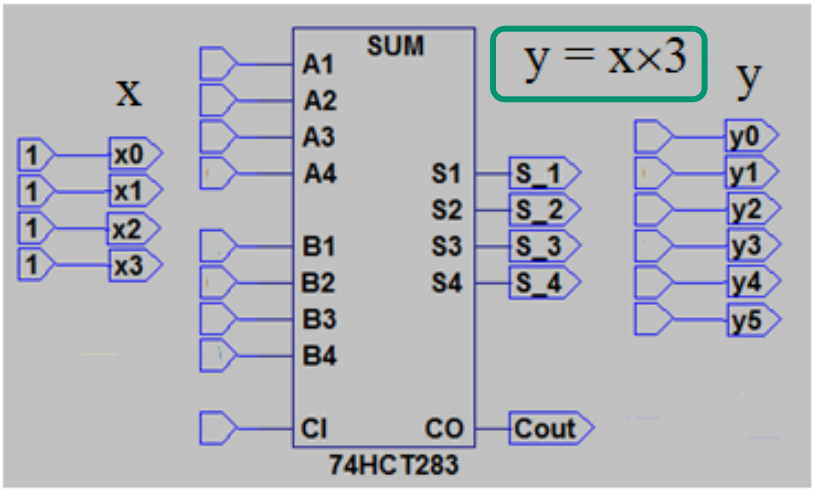
Inverteringen sker med en kontakt i stället för med XOR-grindar!



Multiplikation med konstant



?



Antag att vi behöver multiplicera ett tal x med 3. Det kan man göra som $2 \cdot x + 1 \cdot x = 3 \cdot x$.
Multiplikation med den jämna 2-potensen 2, sker genom att man "skiftar" anslutningarna för talets inbitar *ett* steg åt vänster.

- Eller som $2 \cdot (x + 0{,}5 \cdot x) = 3 \cdot x$ *Smartare!*

Just for fun

Halvadderarkrets med Dominobrickor

Just for fun

4 bits adderare med Dominobrickor