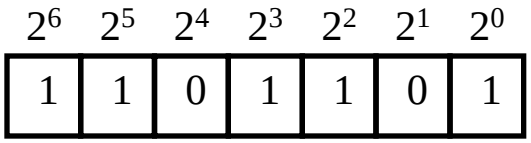
Talrepresentation

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:



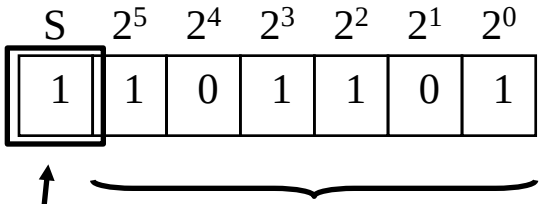
$$= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 = 109$$

Men hur representerar vi negativa tal ???

William Sandqvist william@kth.se

Sign-magnitude

Heltal:

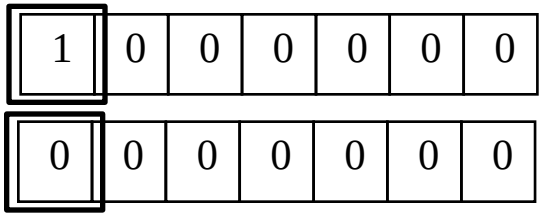


$= - (1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0) = -45$

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:

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

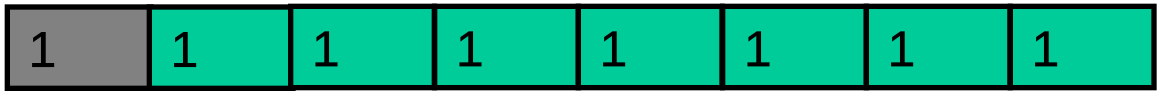
Detta är den vanligaste representationen av tal ”med tecken”.

William Sandqvist william@kth.se

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.

William Sandqvist william@kth.se

Talkonvertering positivt tal till negativt

Tvåkomplementmetoden

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

William Sandqvist william@kth.se

Talkonvertering negativt tal till positivt

Tvåkomplementmetoden

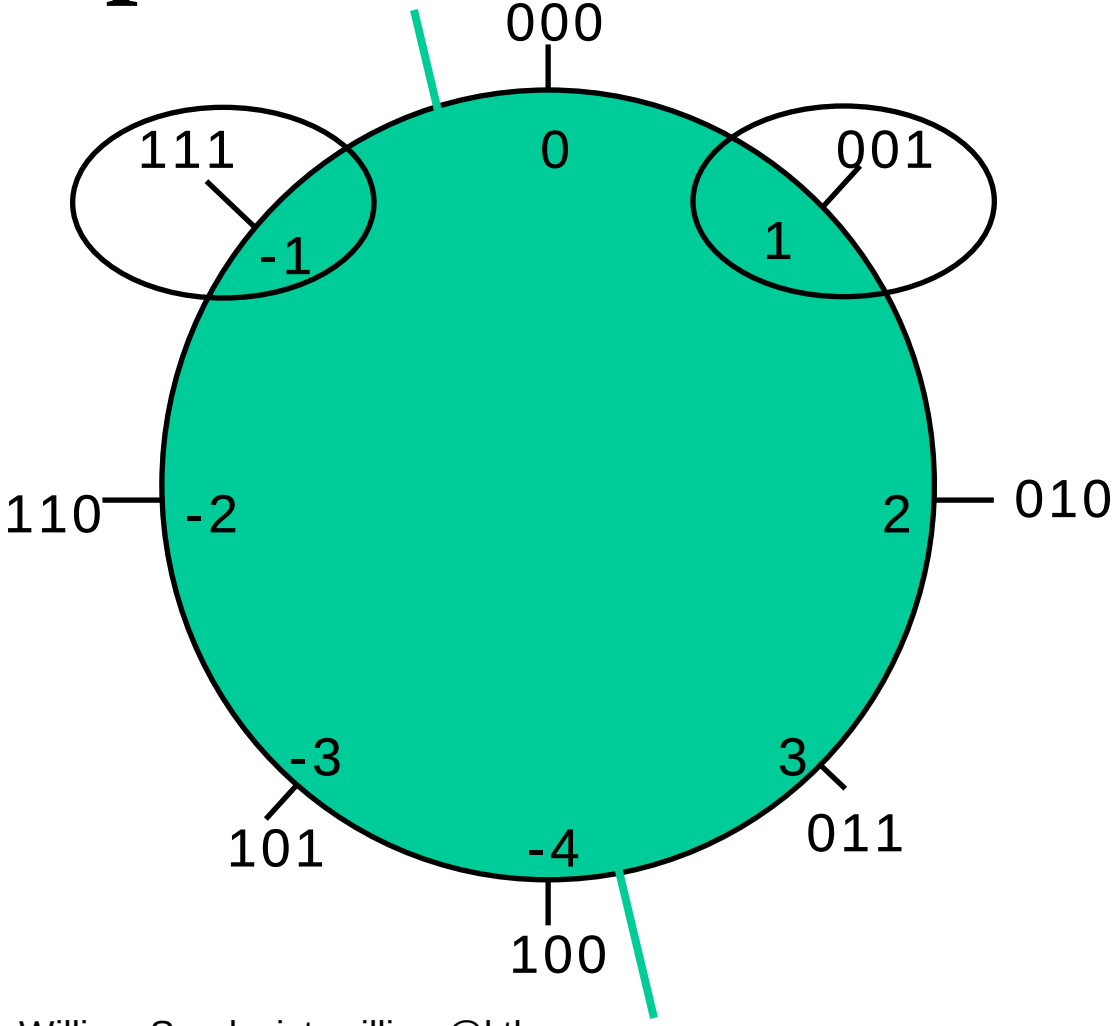
10001	**-15**
01110	**invertera**
01111	**lägg till ett**
01111	**+15**

Samma procedur i bägge riktningarna!

William Sandqvist william@kth.se

2-komplement heltal

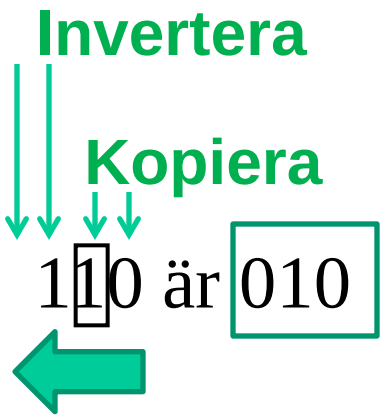
Dator-register är ”ringar”. Figuren visar ett trebitars-register. När man räknar med ”tal med tecken” är de negativa talen den vänstra halvan av ringen.



William Sandqvist william@kth.se

2-komplementet ”snabbt”

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

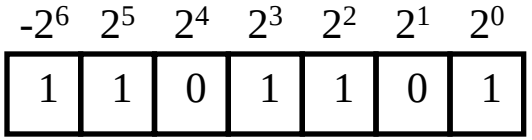


Exempel: 2-komplement från 110 är 010

Sign-extension

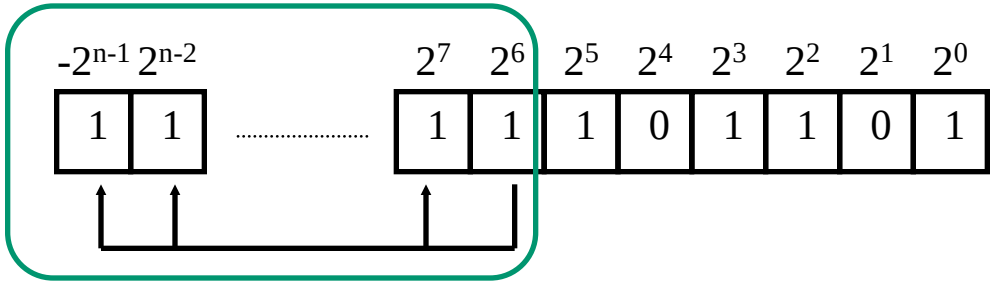
Vid beräkningar i datorer behöver man ofta öka antalet siffror (bitar) inför någon beräkning – hur gör man det med negativa tal?

Heltal:



$= -1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 = -45$

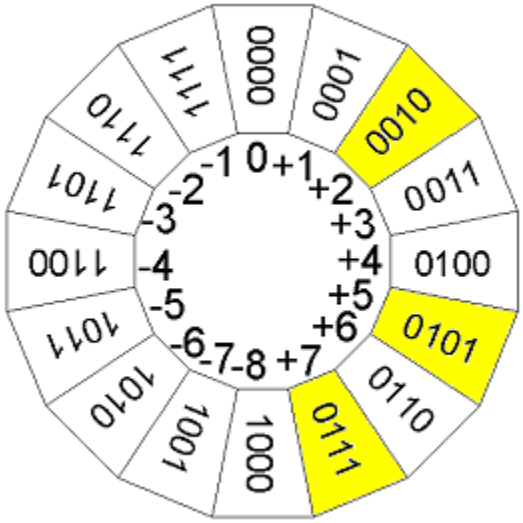
Teckenbiten har negativ vikt



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

William Sandqvist william@kth.se

Addition (BV: sida 264)

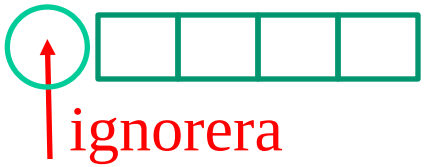


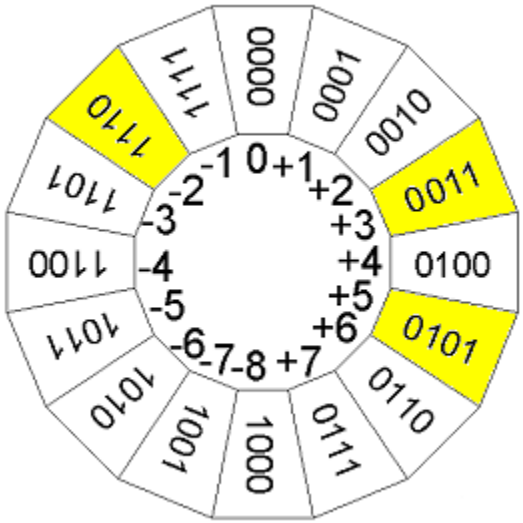

```
   (+5)
+  (+2)
-------
   (+7)
```

```
    0101
+   0010
--------
    0111
```

William Sandqvist william@kth.se

Addition (BV: sida 264)





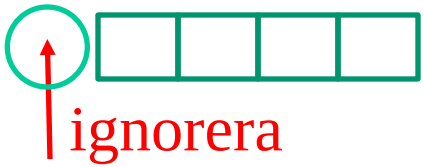

```
   (+5)
+  (-2)
-------
   (+3)
```

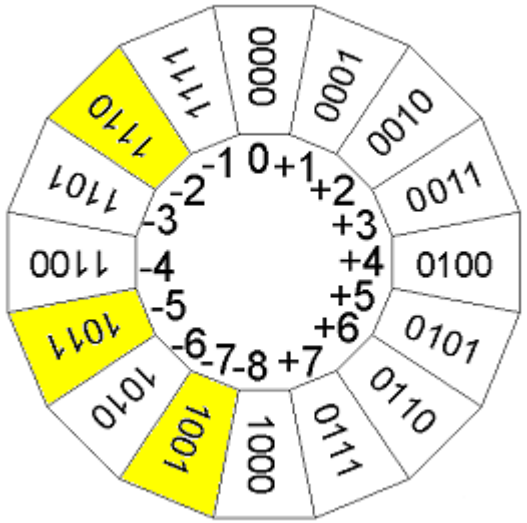
```
  1 1
    0101
+   1110
--------
  1 0011
```

Carry-biten kan ignoreras!

William Sandqvist william@kth.se

Addition (BV: sida 264)





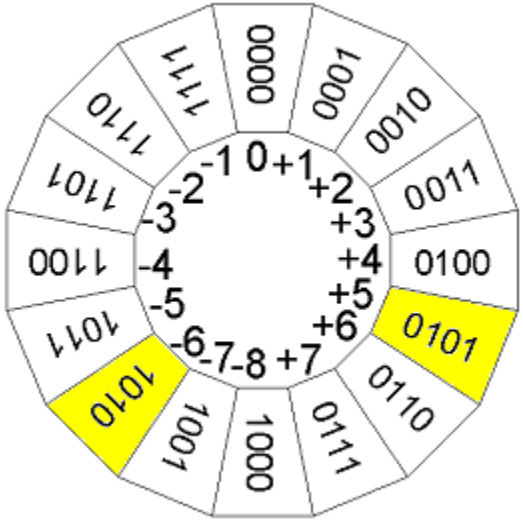

```
   (-5)
+  (-2)
-------
   (-7)
```

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

Carry-biten kan ignoreras!

William Sandqvist william@kth.se

Overflow



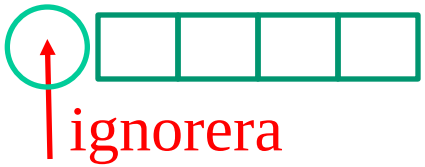
Overflow – teckenbiten stämmer *inte* överens med ingående tal...

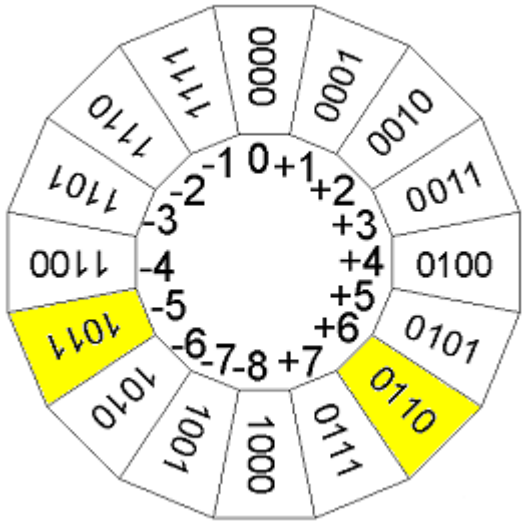
```
   (+5)
+  (+5)
-------
   (-6)
```

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

William Sandqvist william@kth.se

Overflow 2





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

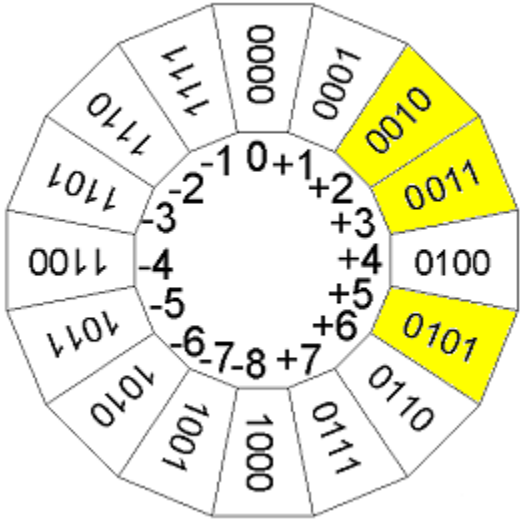
```
   (-5)
 + (-5)
 ------
  (+6)
```

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

Carry-biten kan ignoreras!

William Sandqvist william@kth.se

Subtraktion (BV: sida 265)




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

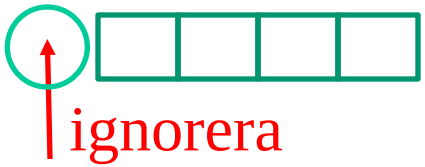
”Låna” ett

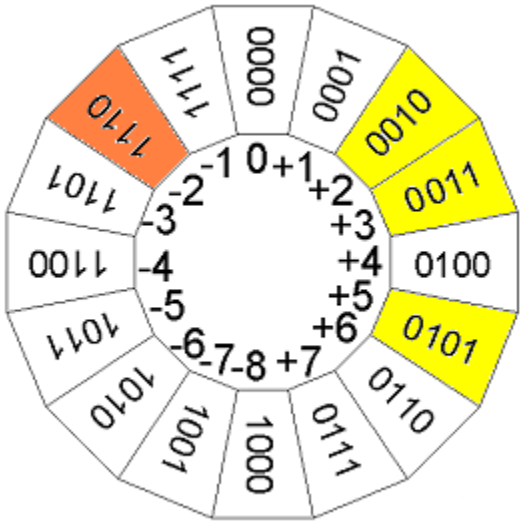
```
  10
  0101
- 0010
------
  ????
```

Hur gör man subtraktionen på ett enkelt sätt?

William Sandqvist william@kth.se

Subtraktion (BV: sida 265)





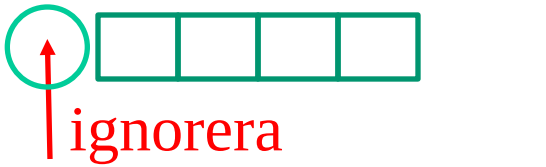

```
  (+5)        0101          1 1
- (+2)      - 0010        + 1110
-------      ------      --------
  (+3)        ????        1 0011
```

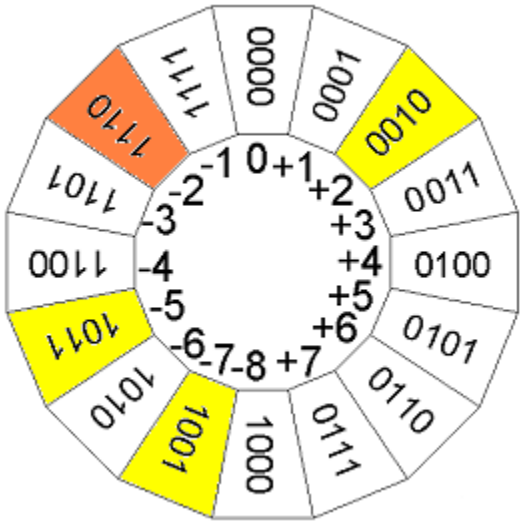
Carry-biten kan ignoreras!

Gör en addition med 2-komplementet i stället!

William Sandqvist william@kth.se

Subtraktion (BV: sida 265)



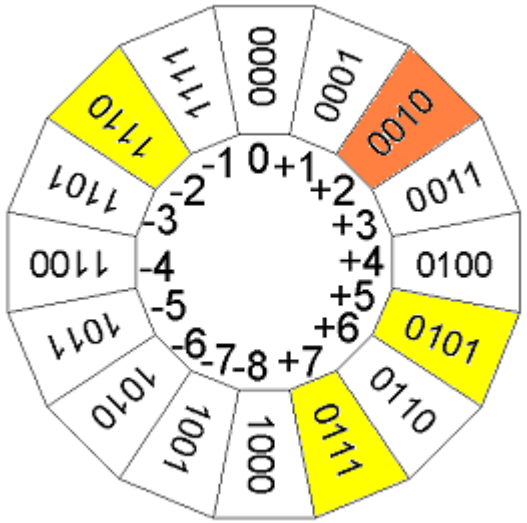



```
  (-5)        1011        111
- (+2)      - 0010      + 1110
-------     -------     -------
  (-7)        ????      1 1001
```

Carry-biten kan ignoreras!

William Sandqvist william@kth.se

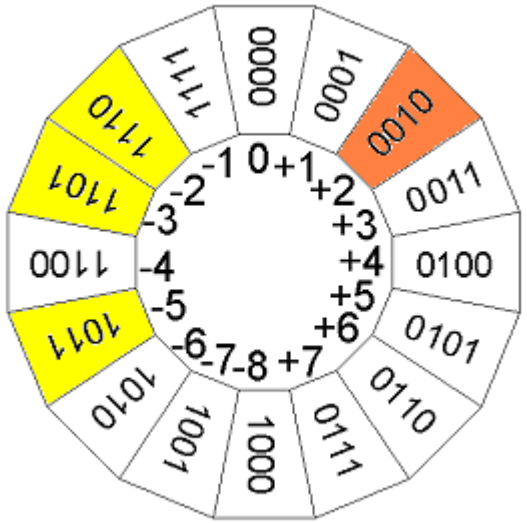
Subtraktion (BV: sida 265)




```
  (+5)      1011        0101
- (-2)    - 1110      + 0010
-------    ------      ------
  (+7)      ????        0111
```

William Sandqvist william@kth.se

Subtraktion (BV: sida 265)



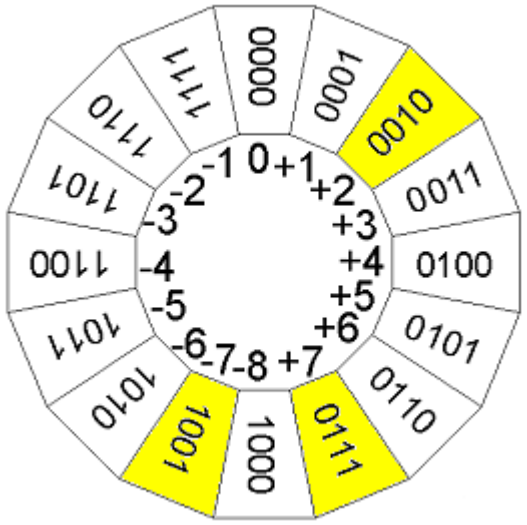
$$\begin{array}{r} (-5) \\ -\ (-2) \\ \hline (-3) \end{array} \qquad \begin{array}{r} 1011 \\ -\ 1110 \\ \hline ???? \end{array} \qquad \Rightarrow \qquad \begin{array}{r} \overset{1}{} \\ 1011 \\ +\ 0010 \\ \hline 1101 \end{array}$$

William Sandqvist william@kth.se

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 nummer 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.

Alternativt sätt att detektera overflow (BV: sida 271)



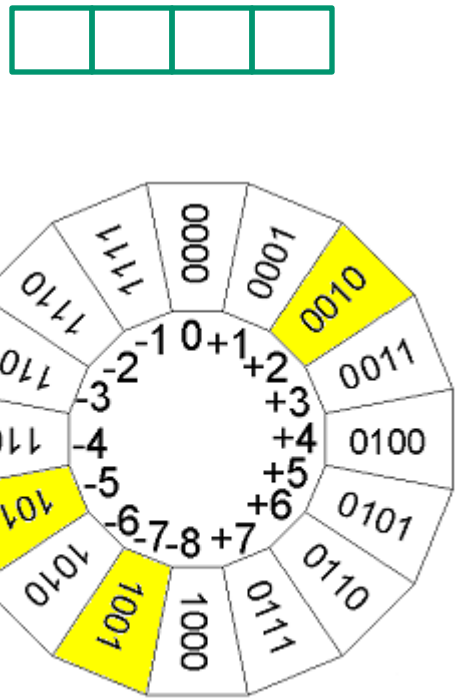
c_4=0 c_3=1

```
  (+7)          0 11
+ (+2)          0111
                + 0010
-------         ------
 (-7)           1001
```

Overflow eftersom c_4 och c_3 är olika!

William Sandqvist william@kth.se

Alternativt sätt att detektera overflow (BV: sida 271)



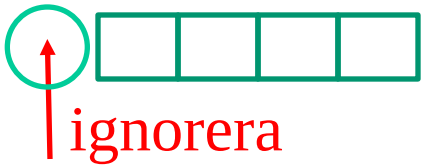
c_4=0 c_3=0

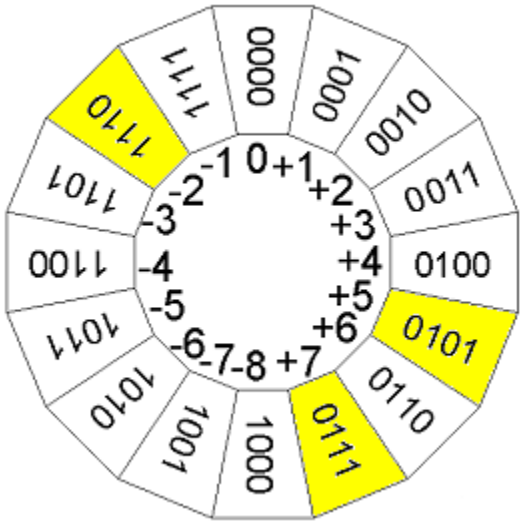
```
  (-7)          1001
+ (+2)        + 0010
-------       ------
  (-5)          1011
```

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

William Sandqvist william@kth.se

Alternativt sätt att detektera overflow (BV: sida 271)





$c_4=1$ $c_3=1$

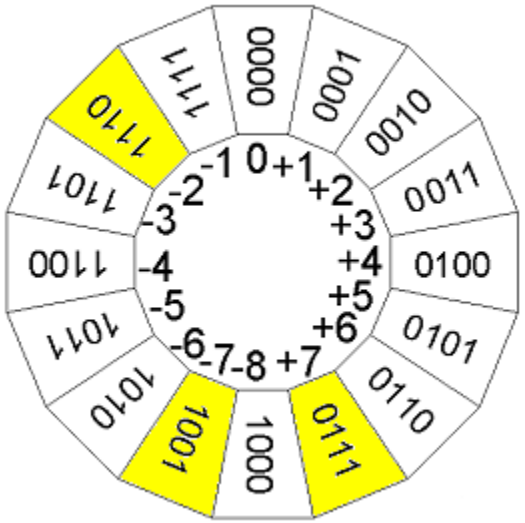
```
  (+7)        1 11        0111
+ (-2)      + 1110      + 1110
-------     --------    --------
  (+5)      1 0101      1 0101
```

Carry-biten kan ignoreras!

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

William Sandqvist william@kth.se

Alternativt sätt att detektera overflow (BV: sida 271)



c_4=1 c_3=0

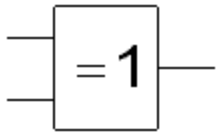
```
  (-7)        1 0        1001
+ (-2)      +          + 1110
-------     ---        ------
  (+7)      1 0111     1 0111
```

Carry-biten kan ignoreras!

Overflow eftersom c_4 och c_3 är olika!

William Sandqvist william@kth.se

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

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

William Sandqvist william@kth.se

En Snabbfråga …

Talet +5 representeras med 0101 om vi använder 4 bitar.

Vilket tal mostvarar -5 om vi använder 8 bitars två-komplement representation?

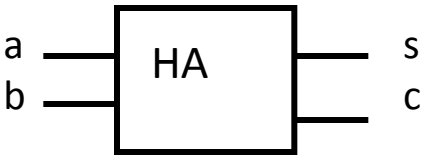
a: 1111 1011

b: 1111 1010

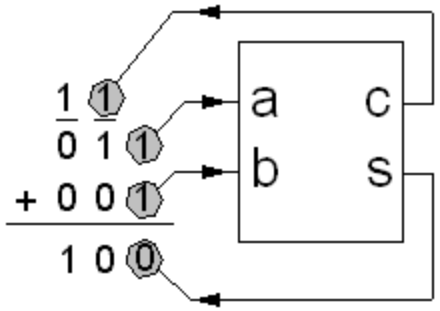
c: 1000 0101

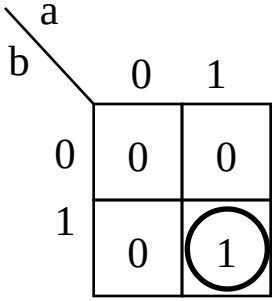
William Sandqvist william@kth.se

Halv-adderaren (Half adder)

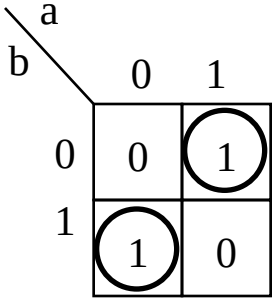


a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

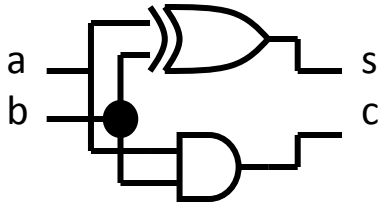




$c = a \cdot b$

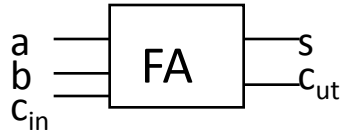


$s = a \oplus b$

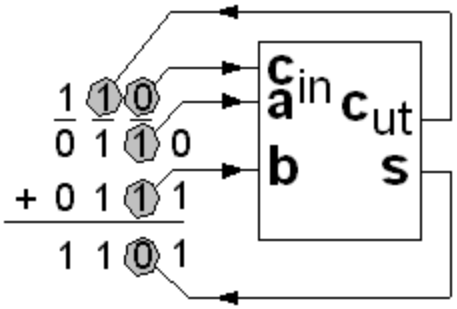


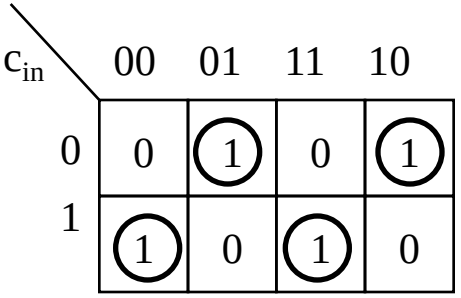
William Sandqvist william@kth.se

Hel-adderaren (Full adder)

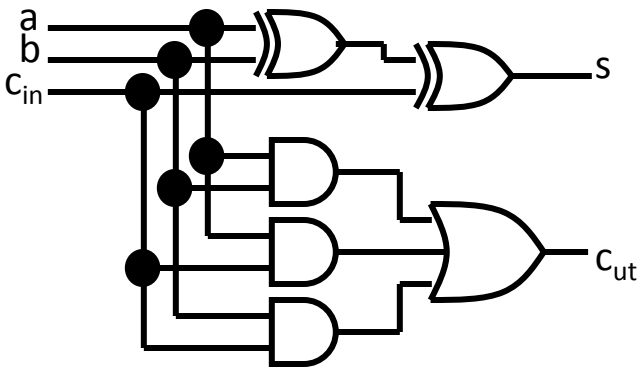


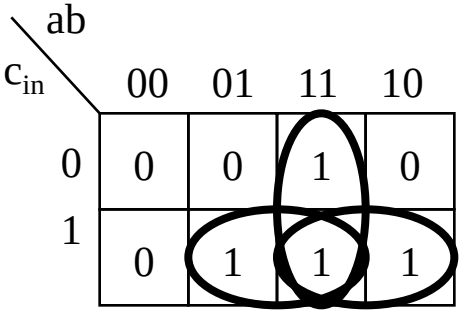
a	b	c_{in}	c_{ut}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





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

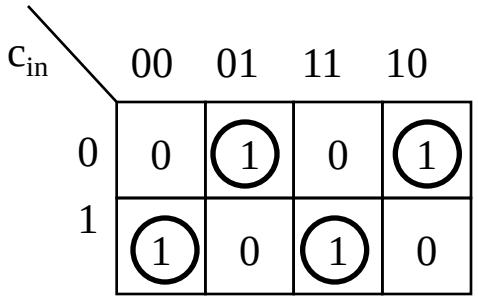




$c_{ut} = a\,b + c_{in}a + c_{in}b$

William Sandqvist william@kth.se

Summafunktionen?

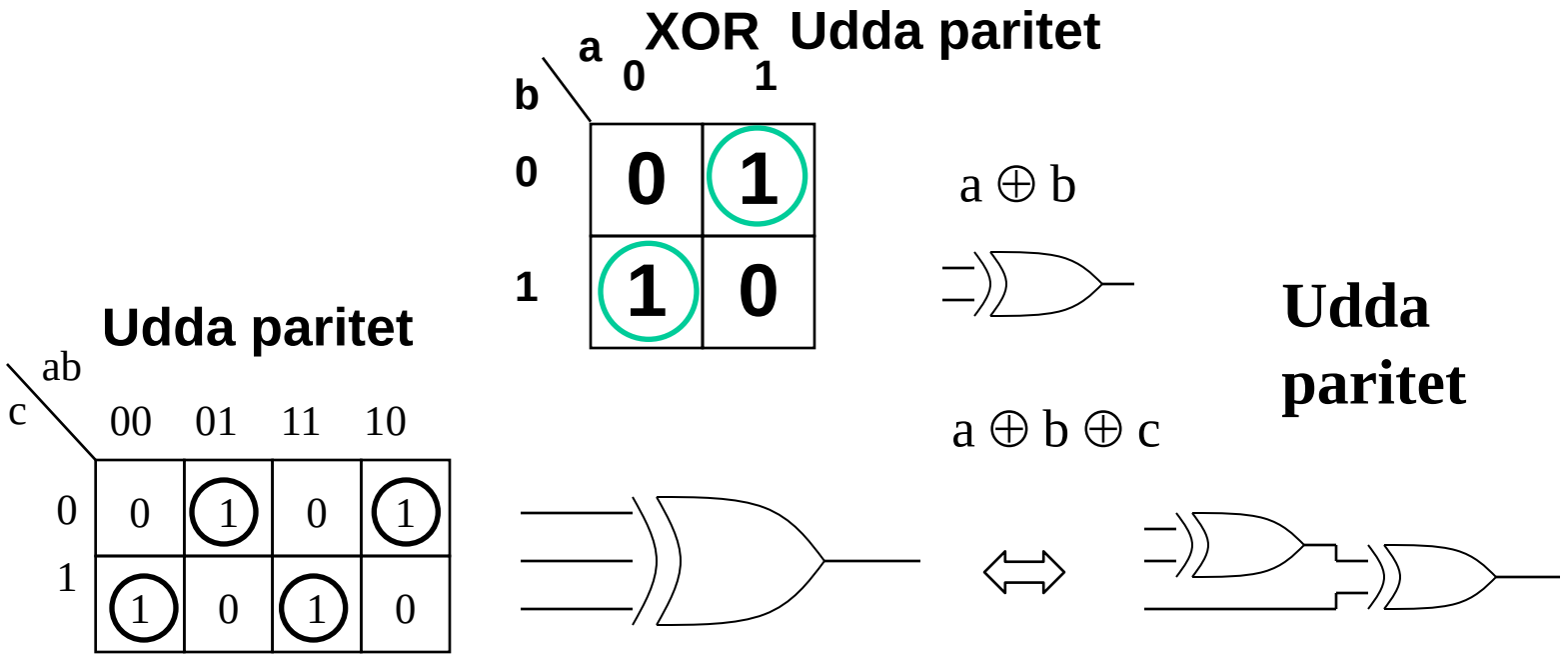


$$s = a \oplus b \oplus c_{in}$$

William Sandqvist william@kth.se

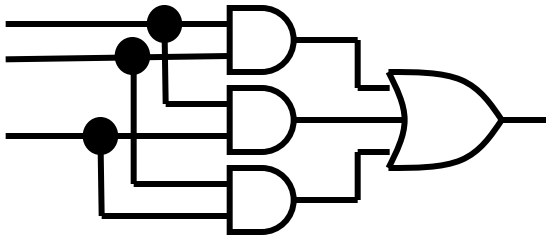
Summa = Udda paritet

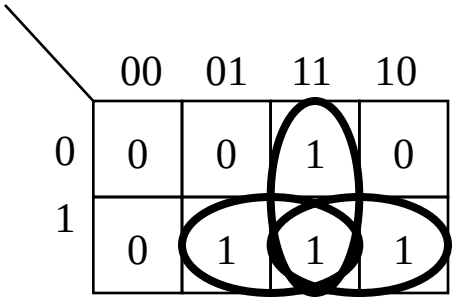
Heladderarens summafunktion är den ”udda”-paritetsfunktionen. Detta är XOR-funktionens naturliga utökning för fler variabler än två. **Udda paritet** är när antalet 1:or på ingångarna är ett udda tal.



William Sandqvist william@kth.se

Carryfunktionen?

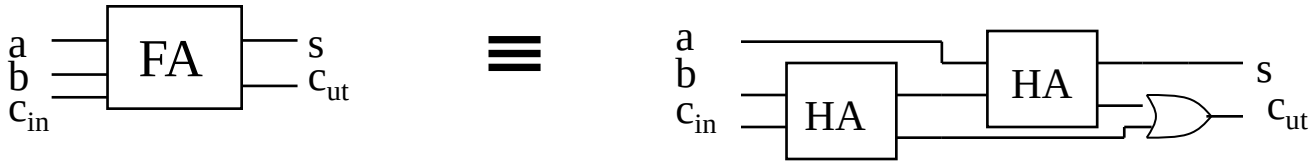




• **Majoritetsfunktionen**. Utgången antar det som är flest 1/0 på ingångarna.

Hel-adderaren med två ½ -adderare

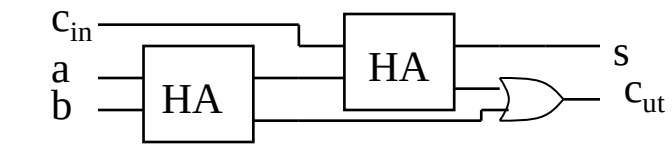
Vi kan även konstruera en hel-adderare mha två halv-adderare och en OR-grind

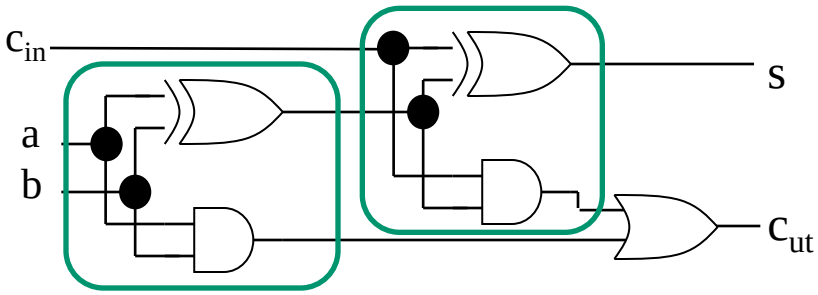


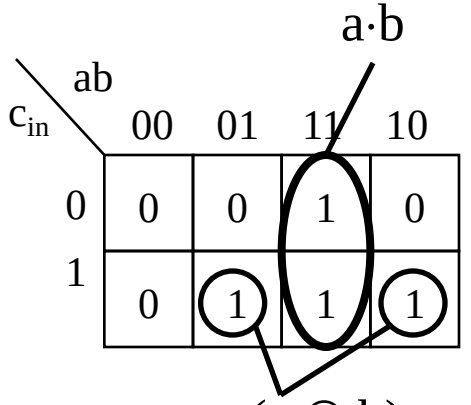
Dekomposition innebär att man ser kretsen som sammansatt av byggblock. Med hjälp av sådana kända byggblock kan man sedan bygga helt nya system **Komposition**.

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







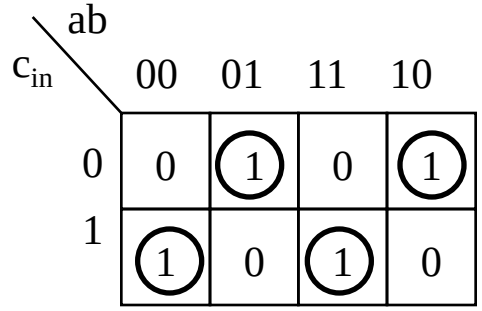
$(a \oplus b)\cdot c_{in}$

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

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

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

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



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

William Sandqvist william@kth.se

Populär tatuering?

Tatueringar är för evigt! Tyvärr är detta inte den ”bästa” adderarkretsen, inte om man vill ha snabba datorer. Spännande fortsättning om adderarkretsar följer …

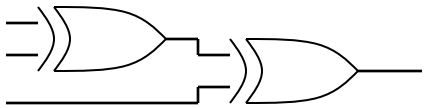
William Sandqvist william@kth.se

(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.

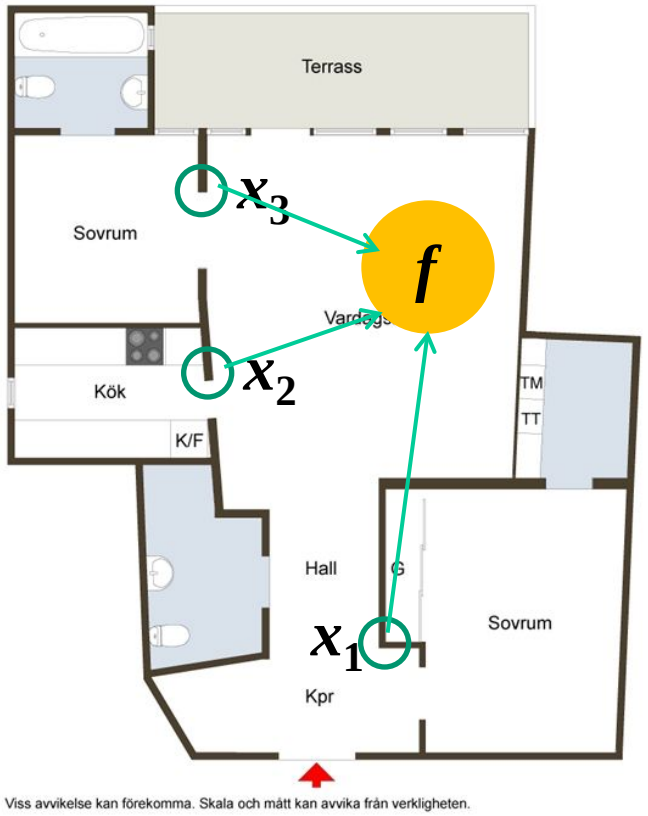


William Sandqvist william@kth.se

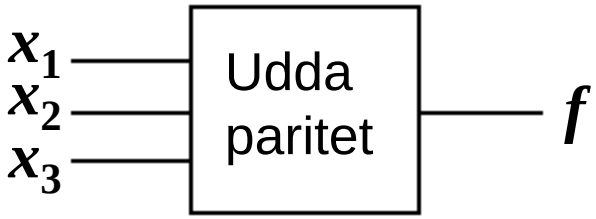
Trevägs ljuskontroll - *revisited*

Brown/Vranesic: 2.8.1

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

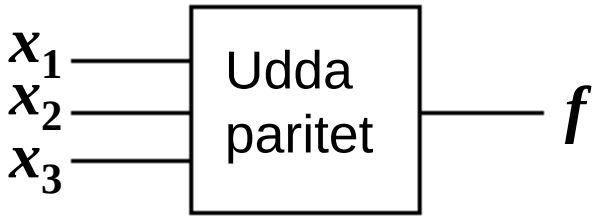


Paritetsfunktionen



William Sandqvist william@kth.se

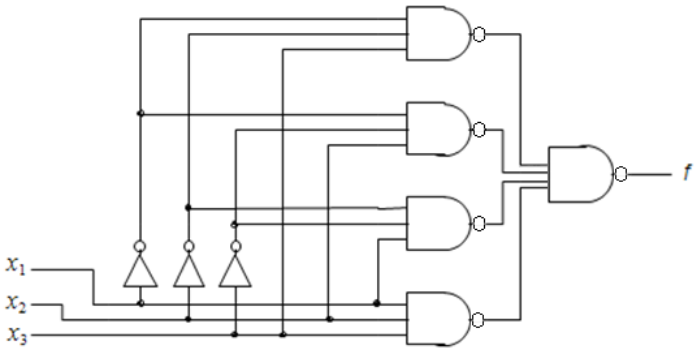
XOR eller NAND?

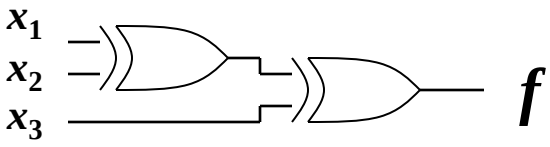


Den tidigare lösningen baserades på NAND-grindar.

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



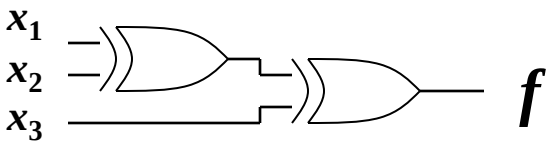


f

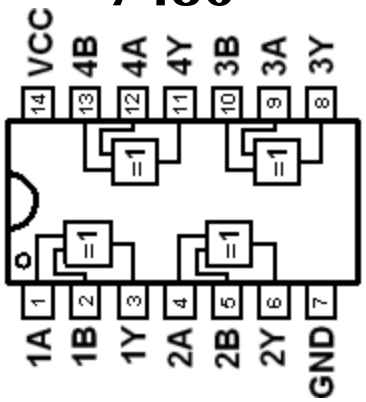
x_1 \ $x_2 x_3$	00	01	11	10
0	0	(1)	0	(1)
1	(1)	0	(1)	0

William Sandqvist william@kth.se

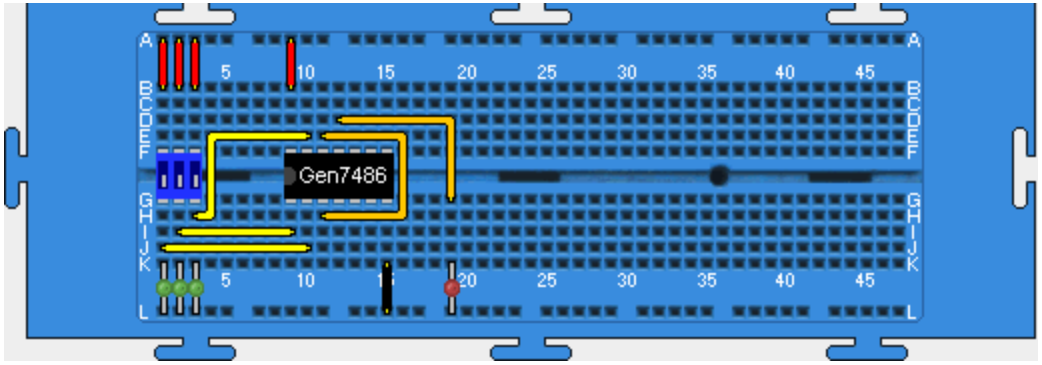
Enklare med XOR-grindar



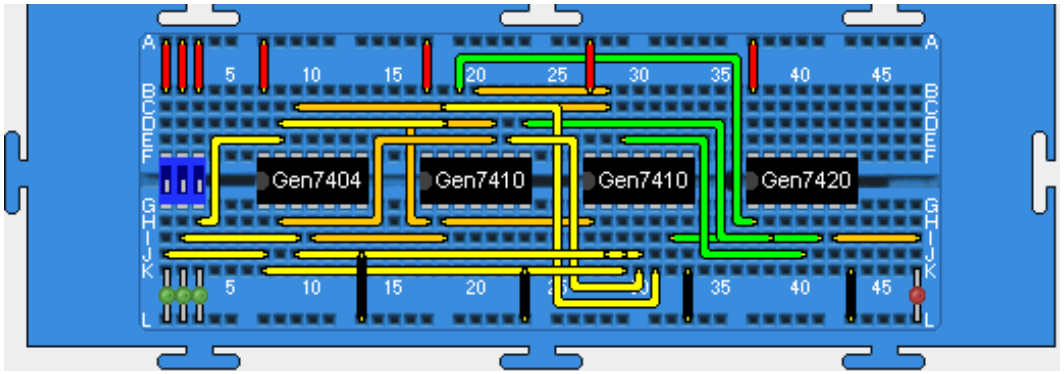
7486



Med XOR-grindar:



(Med NAND-grindar:)

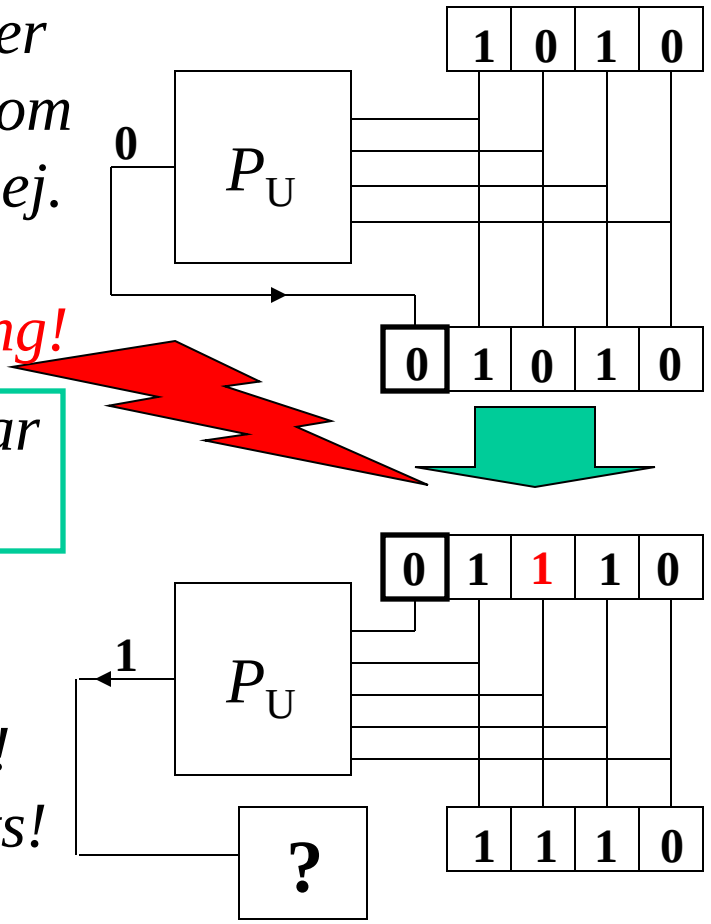


William Sandqvist william@kth.se

William Sandqvist william@kth.se

(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

William Sandqvist william@kth.se

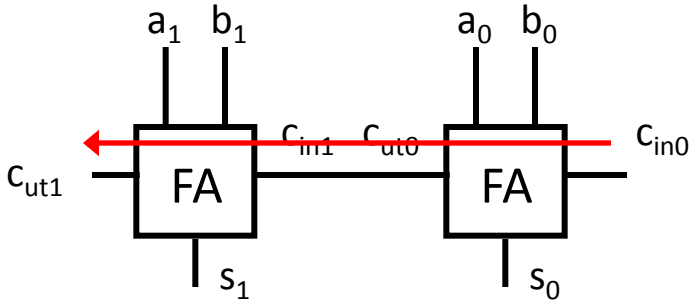
William Sandqvist william@kth.se

Mer **komposition**

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

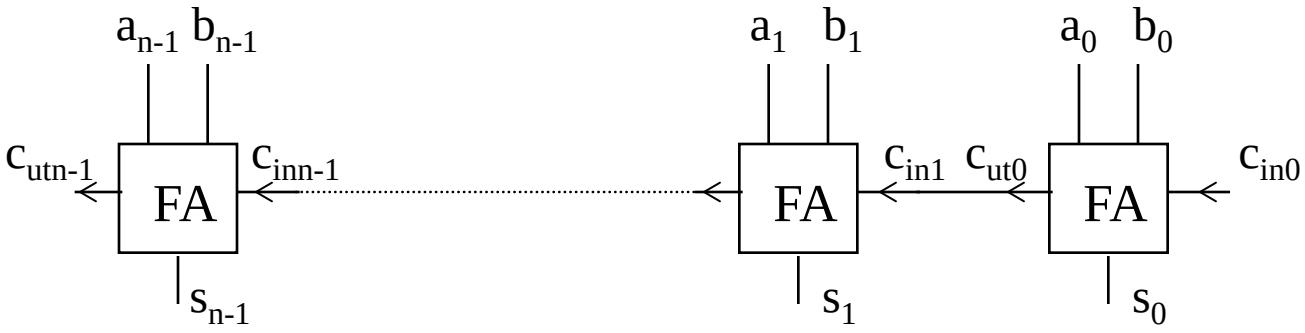
- Man behöver *n* hel-adderare för att konstruera en *n*-bit-adderare
(eller 2·*n* halvadderare)

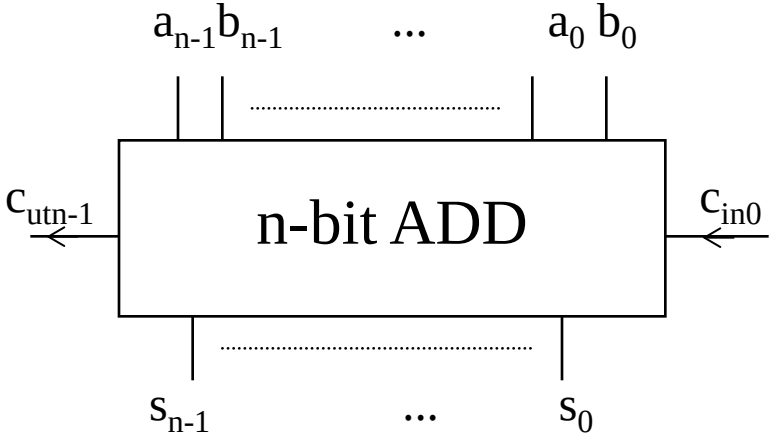
Ripple-Carry Adderare (RCA)



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

Ripple-Carry Adderare (RCA)



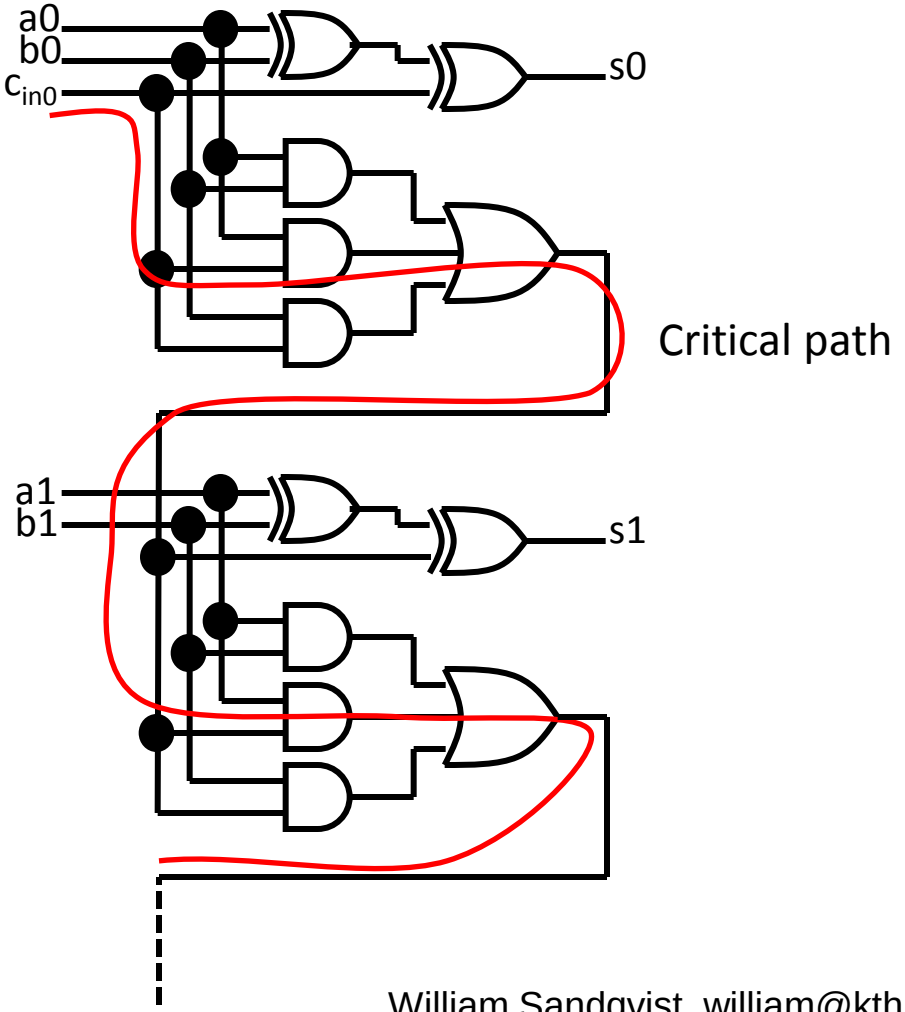


A_{FA} area för en Fulladder
t_{FA} fördröjning genom en Fulladder

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

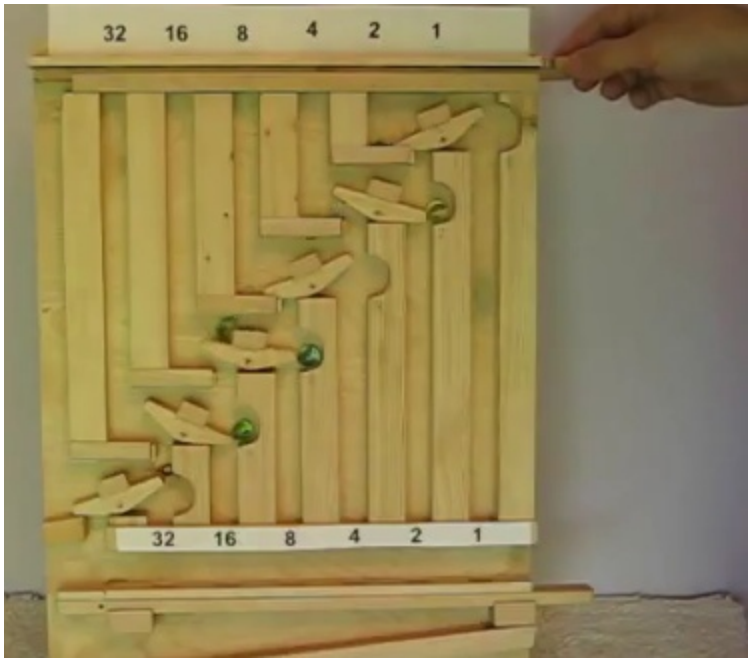
William Sandqvist william@kth.se

Ripple-Carry Adderare (RCA)



William Sandqvist william@kth.se

Ripple effect = efterdyningar



YouTube

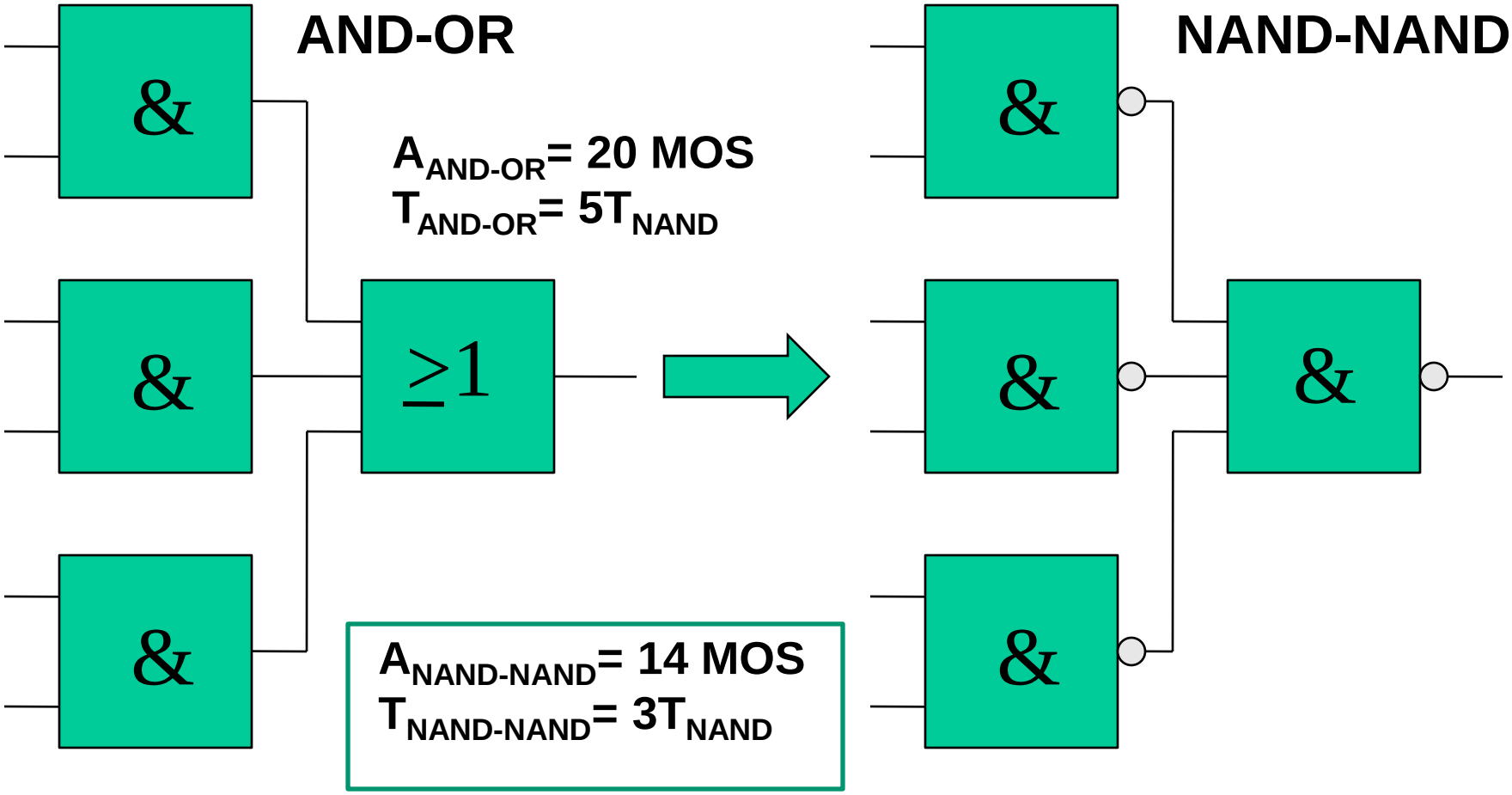
Marble adding machine

Matthiaswandel

Carry måste ”rippla” igenom hela adderaren!

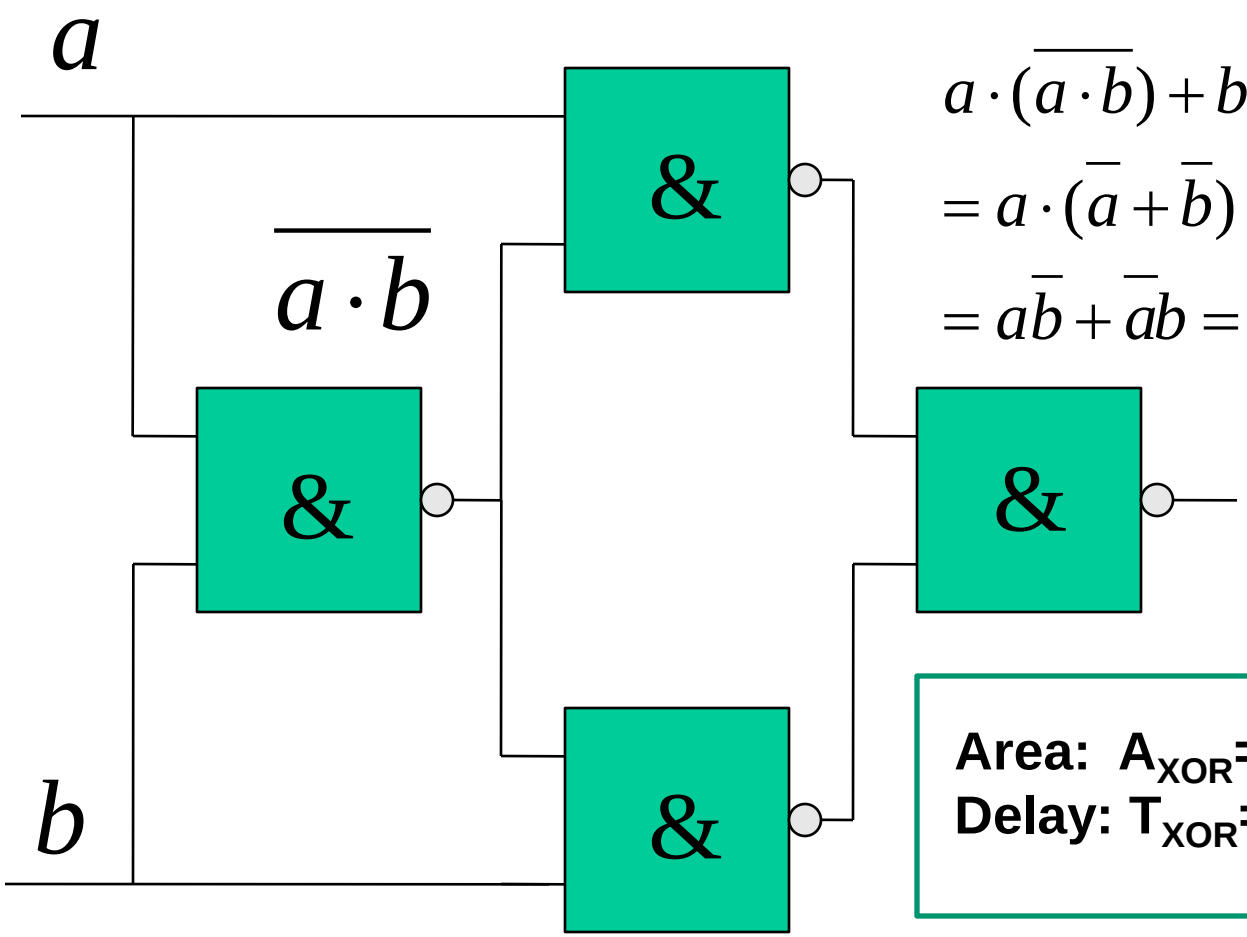
William Sandqvist william@kth.se

Heladderarens Carry-funktion



William Sandqvist william@kth.se

XOR med NAND



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

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

William Sandqvist william@kth.se

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 $g_i = 1$)

$$g_i = x_i y_i$$

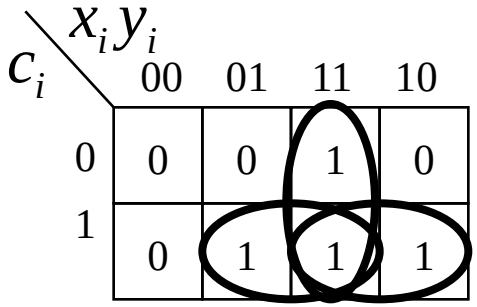
- **Propagate** p_i (carry-out $c_{i+1} = 1$ ifall $c_i = 1$ och $x_i = 1$ eller $y_i = 1$)

$$p_i = x_i + y_i$$

$$p_i = x_i \oplus y_i$$

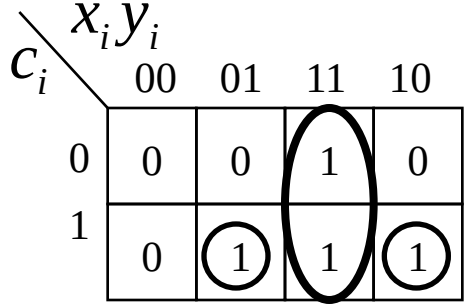
Fungerar också!

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



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

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



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

William Sandqvist william@kth.se

Carry-look-ahead funktion

- **Carry-bit** c_0

$$g_i = x_i y_i$$

$$c_1 = g_0 + p_0 c_0$$

- **Carry-bit** c_1

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

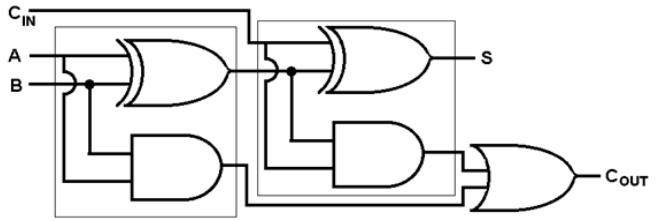
Propagate funktionen från ”föregående” bitar kan genereras parallellt

$$p_i = x_i + y_i$$

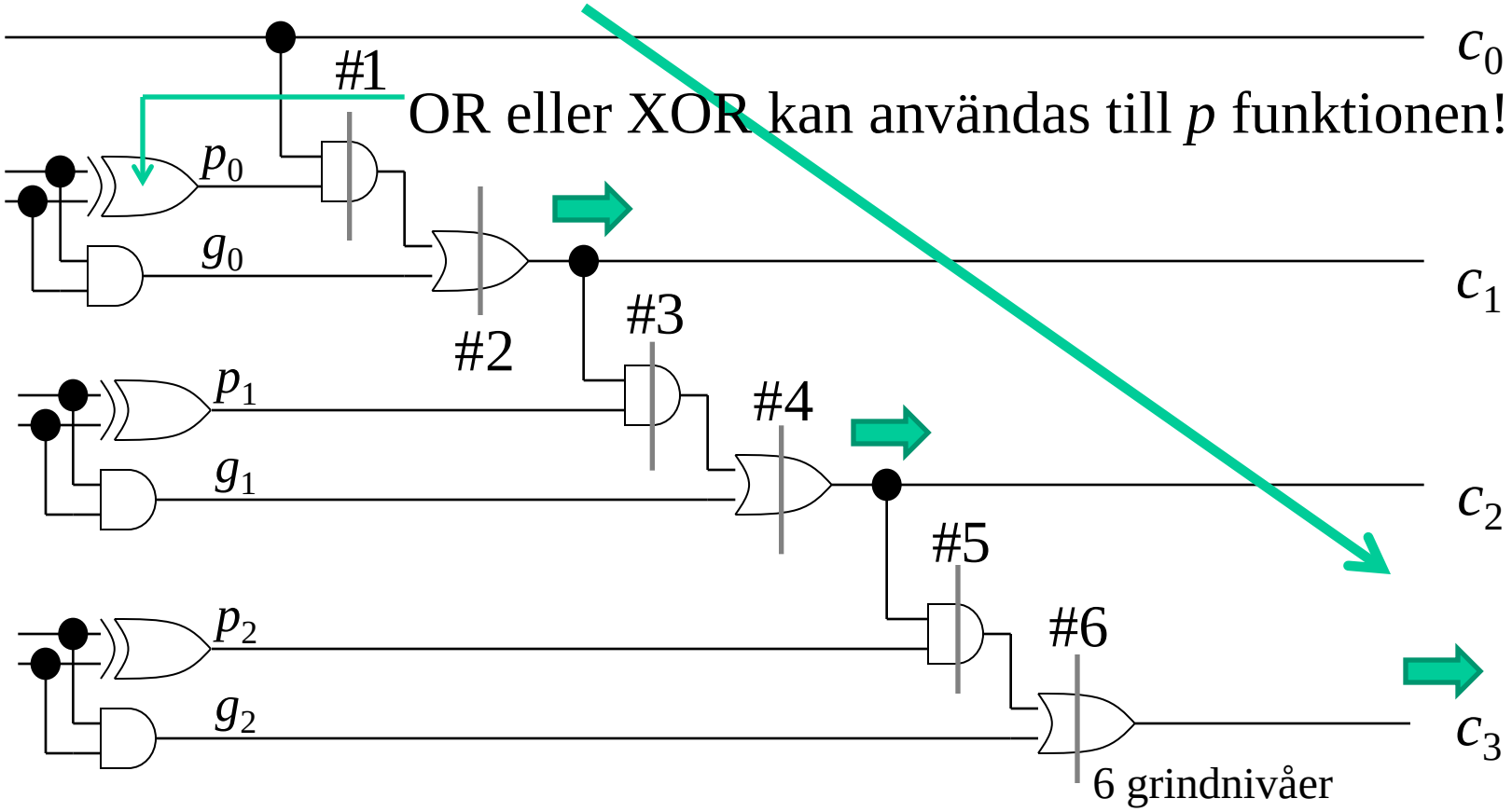
Bara två logiknivåer behövs …

William Sandqvist william@kth.se

Carry-kedjan i ripple-adderaren

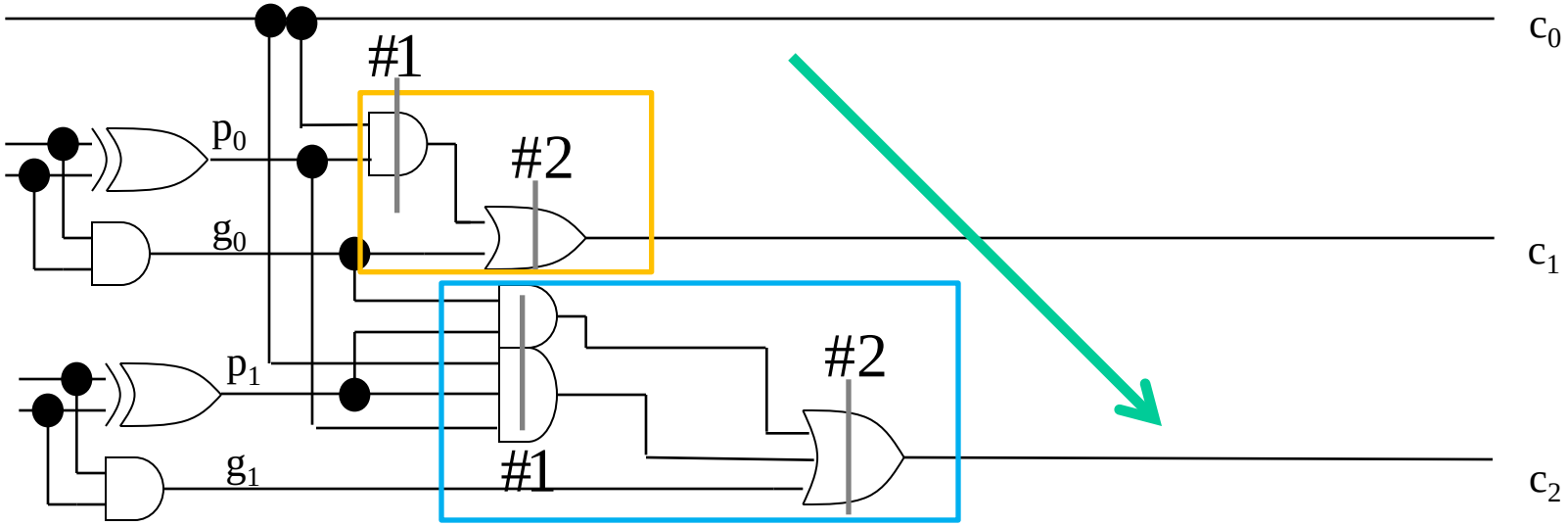


½ + ½



William Sandqvist william@kth.se

Snabbare implementering av Carry-kedjan



$$c_1 = g_0 + p_0 c_0$$

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

- Två-nivåers nät

William Sandqvist william@kth.se

(*n* = 8) med två logiknivåer

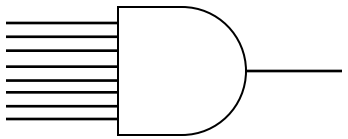
$c_1 = g_0 + p_0 c_0$

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

Snabbt, men otympligt med så här många ingångar!

Ooops!

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



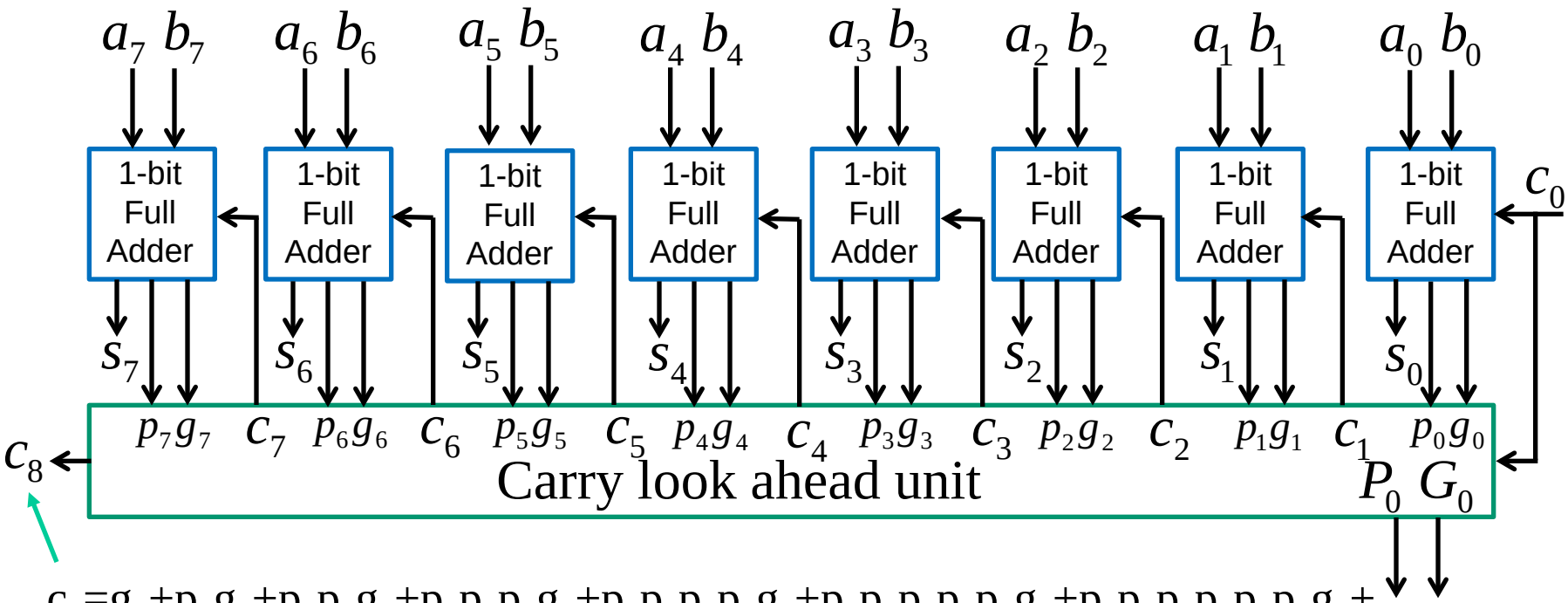
Det behövs ”specialgrindar” till detta …

William Sandqvist william@kth.se

Carry-lookahead adder (CLA)

8-bit adder

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

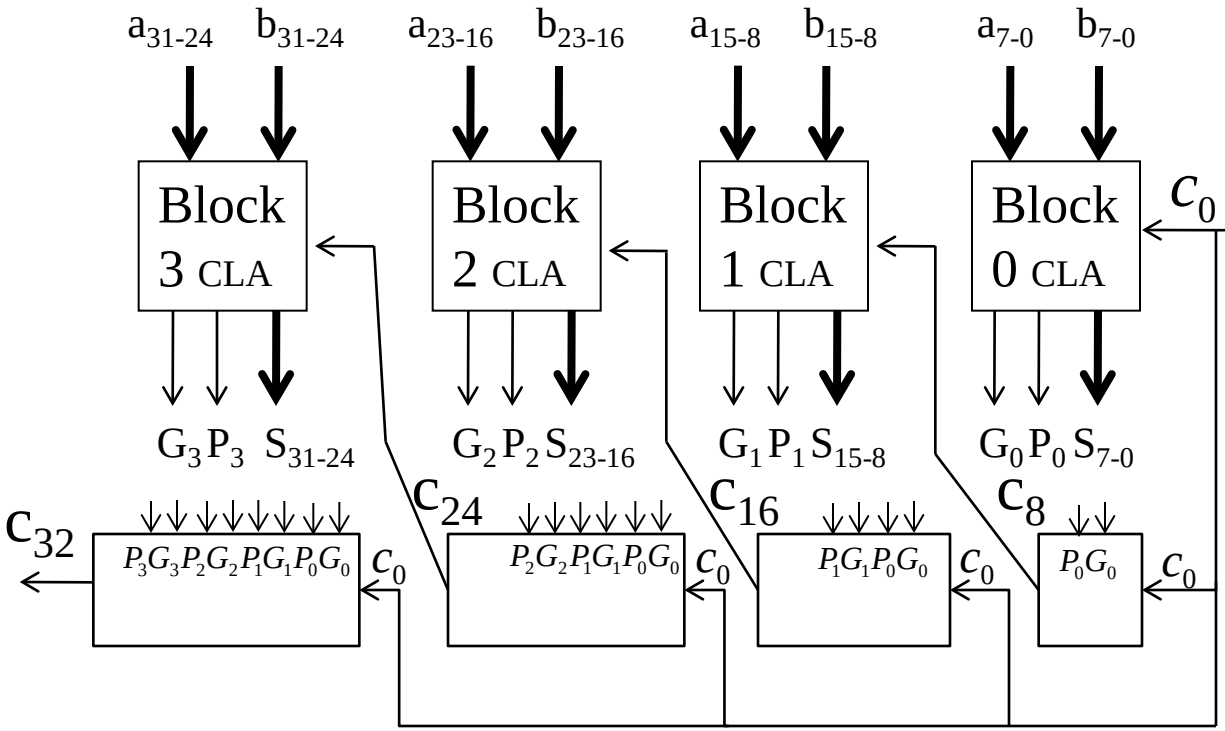


$$c_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0$$

William Sandqvist william@kth.se

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

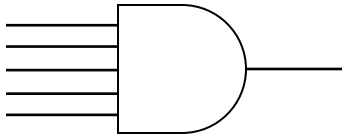
Second level
four Carry lookahead units

William Sandqvist william@kth.se

Hierarkisk expansion

Carry-bitar från en andra nivåns Carry lookahead enheter

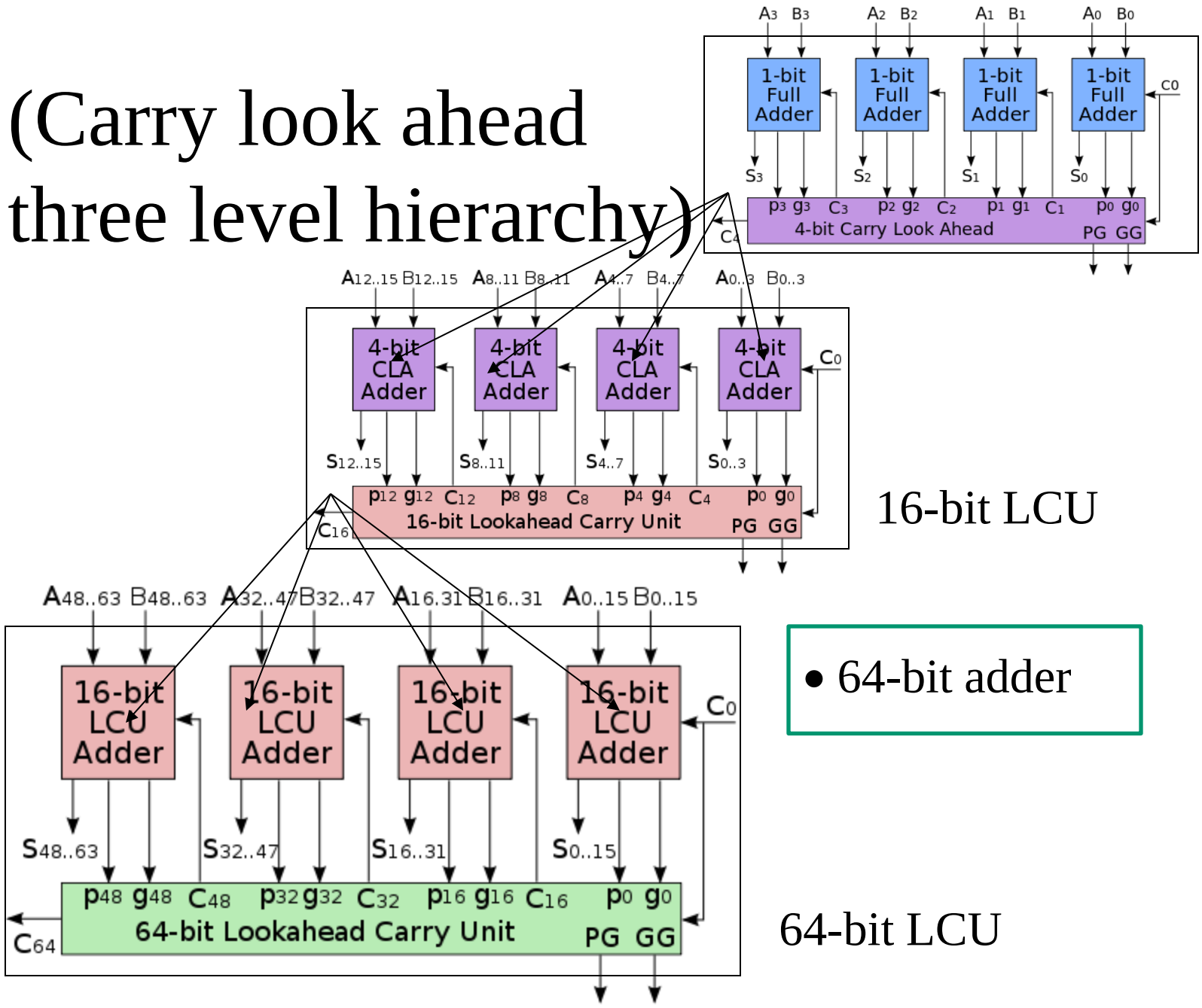
$$C_8 = G_0 + P_0 c_0$$
$$C_{16} = G_1 + P_1 G_0 + P_1 P_0 c_0$$
$$C_{24} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$C_{32} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$



etc.

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

(Carry look ahead three level hierarchy)



16-bit LCU

- 64-bit adder

64-bit LCU

William Sandqvist william@kth.se

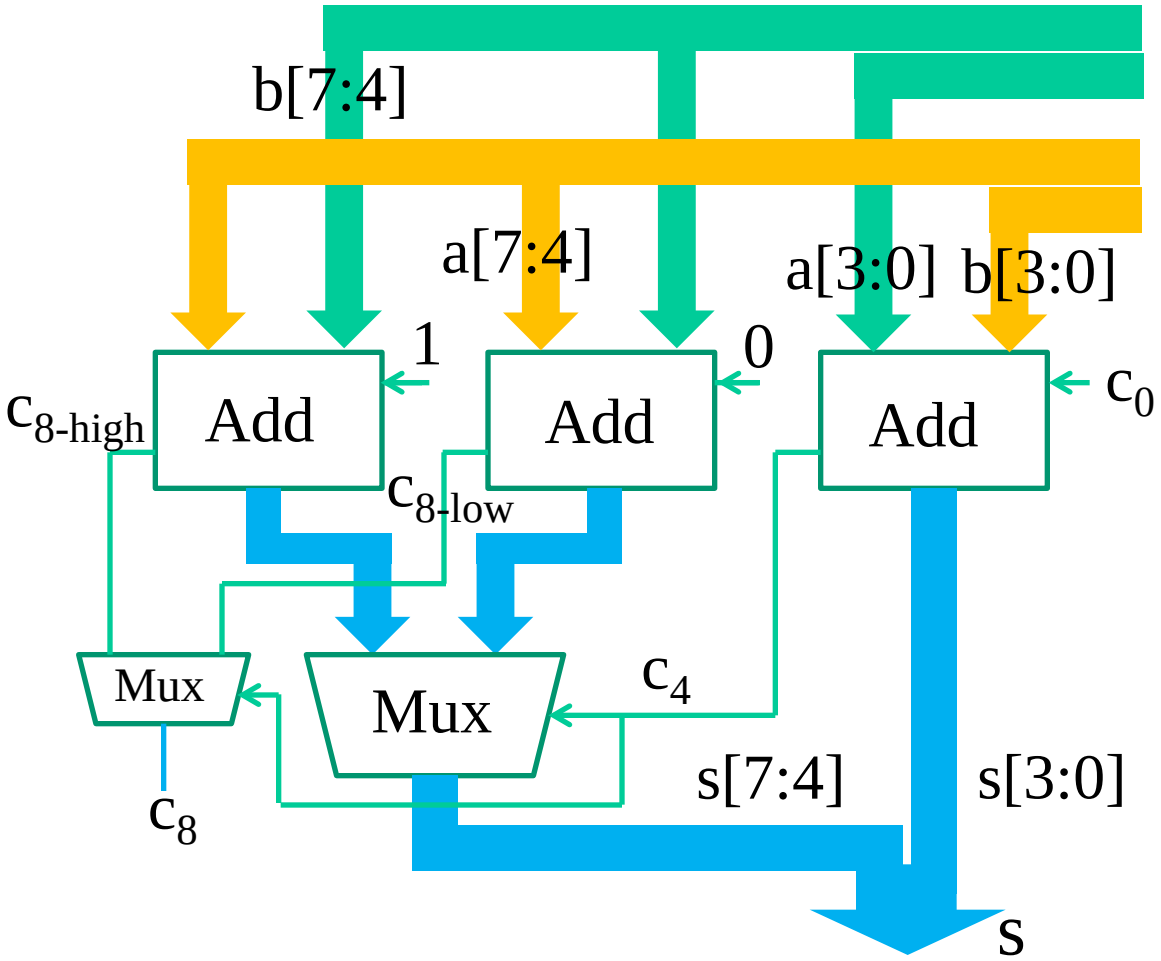
Carry-Select-Adder (CSA)

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 Carry-Select-Adder



William Sandqvist william@kth.se

Jämförelse

- Ripple-Carry Adder

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

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

- Carry-Lookahead Adder (4bits)

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

- Carry-Select Adder (8 bits)

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

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

William Sandqvist william@kth.se

Subtraktion

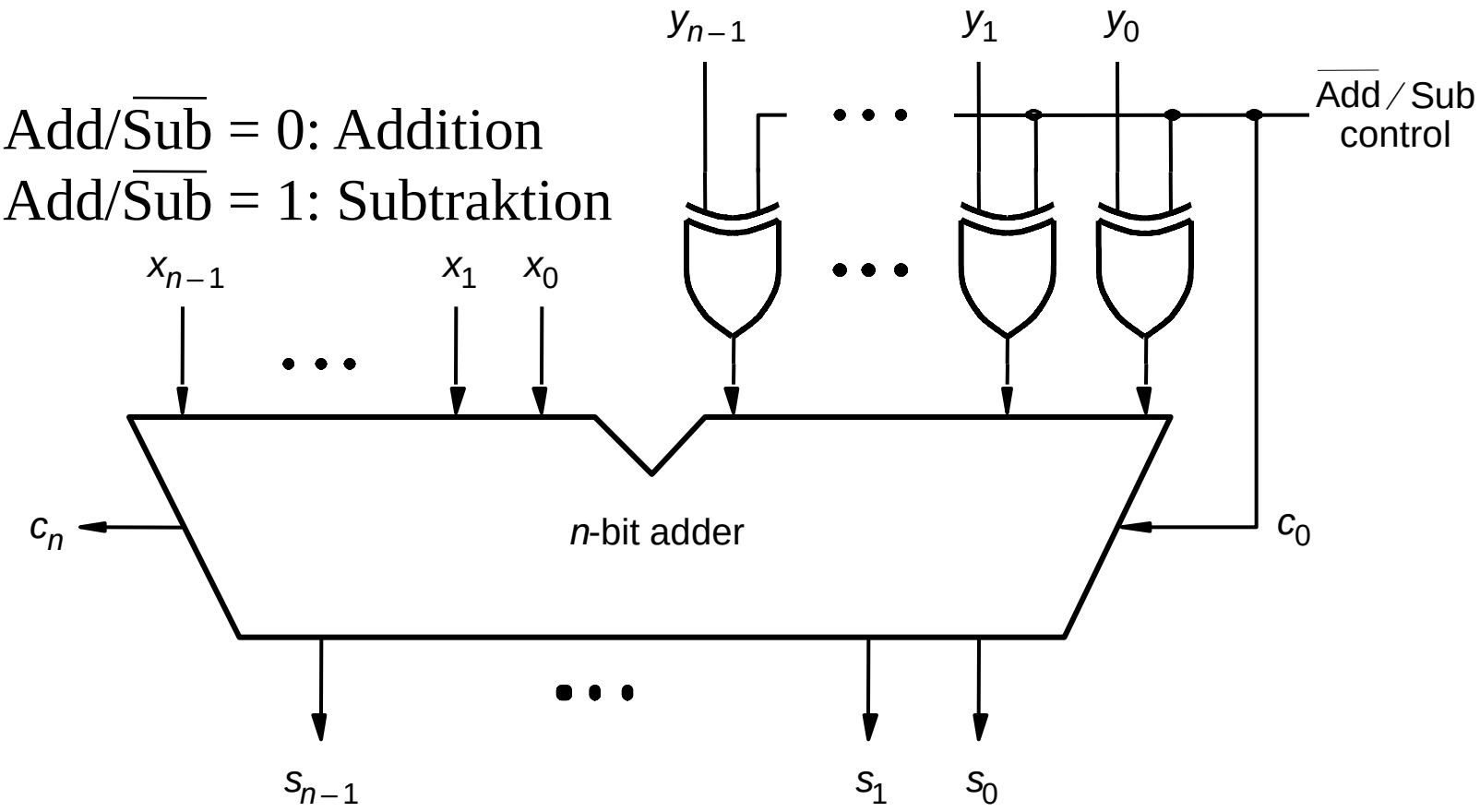
Subtraktion kan göras genom addition med två komplementet

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

Add/sub-enheten

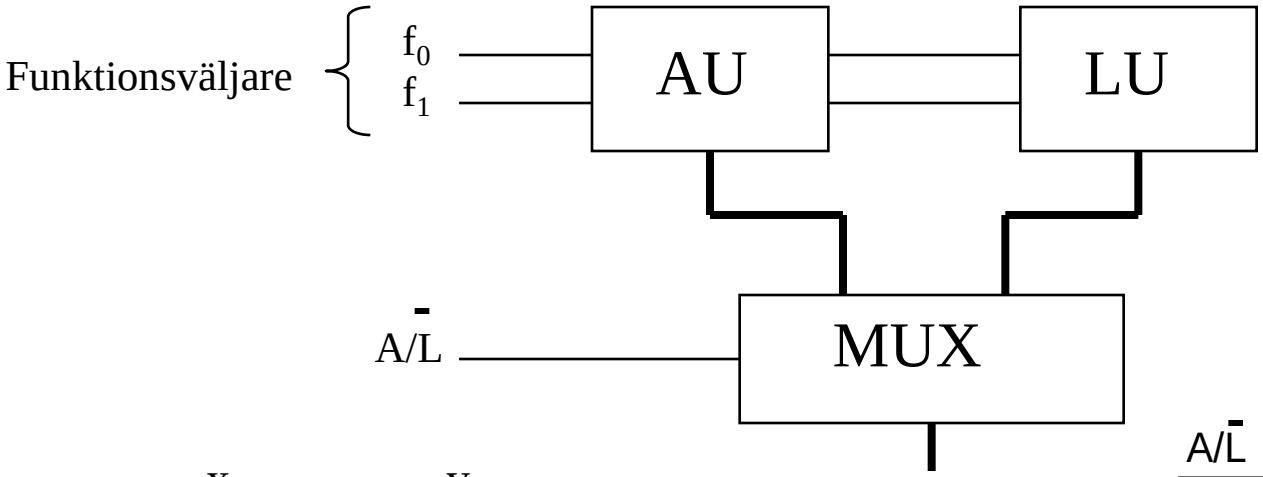
Add/$\overline{\text{Sub}}$ = 0: Addition

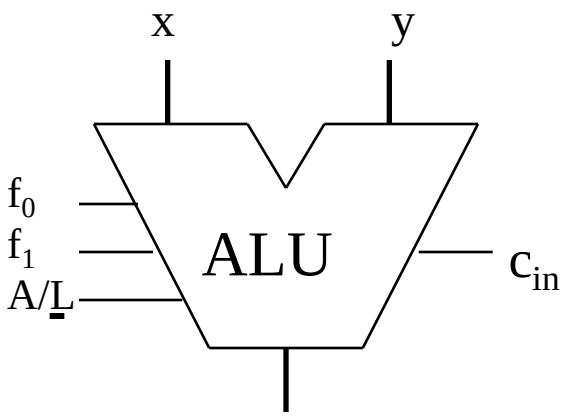
Add/$\overline{\text{Sub}}$ = 1: Subtraktion



William Sandqvist william@kth.se

Arithmetic Logic Unit (ALU)





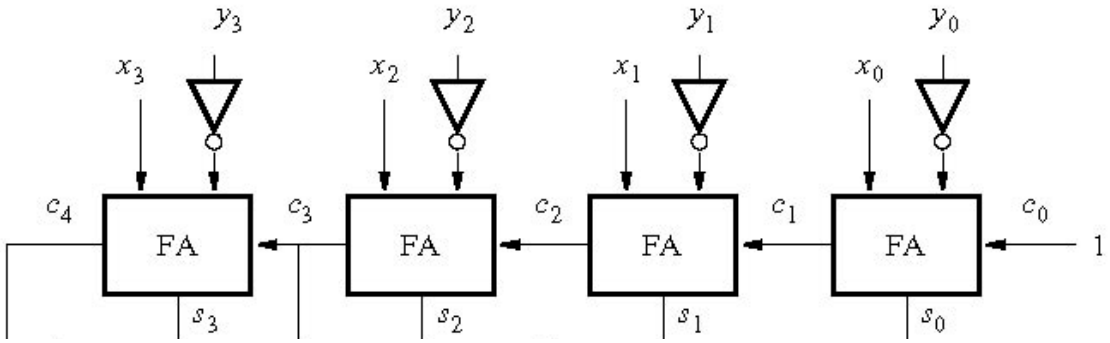
Processorer brukar alltid ha en ALU, inte bara en adderare.

$\overline{A/L}$	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

William Sandqvist william@kth.se

Komparator

Komparatorn implementeras som subtraktionskrets



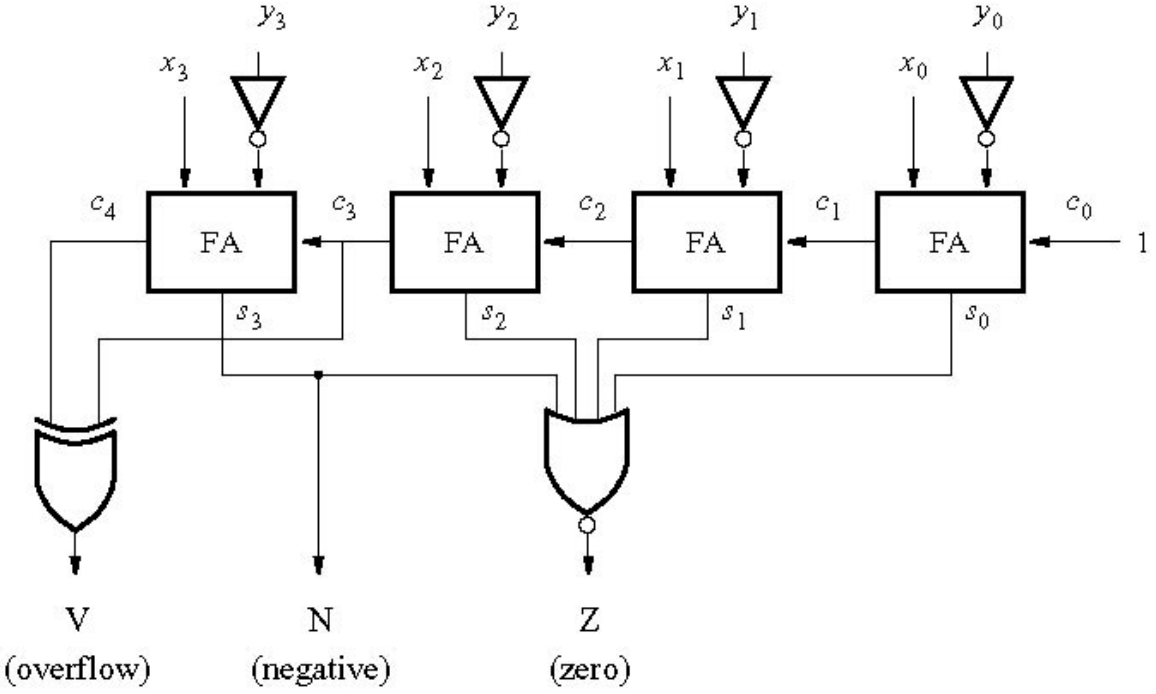
? ? ?

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

William Sandqvist william@kth.se

Komparator

Komparatorn implementeras som subtraktionskrets



William Sandqvist william@kth.se

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)**

William Sandqvist william@kth.se