



Academie voor Technology, Innovation &  
Society Delft  
Academie voor ICT & Media

# Inleiding Digitale Techniek

Week 4 – Binaire optellers, tellen, vermenigvuldigen, delen  
Jesse op den Brouw  
INLDIG/2018-2019

**DE HAAGSE**  
HOGESCHOOL

# Optellen

- Optellen is één van meest gebruikte rekenkundige operatie in digitale systemen.
- Elke *general purpose* microprocessor heeft een optelcircuit aan boord.
- Daarnaast is het eenvoudig een optelschakeling om te zetten in een aftrekschakeling.
- Een speciaal geval van optellen (*add*) is verhogen met één (*increment*). Veel processoren hebben ook een telschakeling (*counter*).

# Optellen

- Optellen in het binaire systeem is identiek aan optellen in het decimale systeem.
- Ook alle andere rekenregels zijn identiek.
- Vermenigvuldigers kunnen worden gemaakt met behulp van optelschakelingen.
- Eerst wordt er uitgegaan van niet-negatieve\* gehele getallen.

\* niet-negatief = *unsigned*

# Optellen

- Het optellen van twee decimale cijfers levert een decimaal getal op van maximaal twee decimale cijfers:

$$\begin{array}{r} 0 \\ \hline 0 \end{array} + \begin{array}{r} 4 \\ \hline 9 \end{array} + \begin{array}{r} 5 \\ \hline 10 \end{array} + \begin{array}{r} 9 \\ \hline 18 \end{array} +$$

- Als het antwoord groter wordt dan 9, moet een *overloop (carry)* naar de volgende kolom worden doorgegeven.
- Het is hierdoor mogelijk twee getallen van willekeurige lengte op te tellen.

# Optellen

- Het optellen van twee decimale getallen gebeurt kolomsgewijs.
- Als het resultaat van een kolomoptelling groter is dan 9, moet een carry naar de volgende kolom worden doorgegeven.

1	1	0	1	0	+ carry
0	9	6	4	9	
0	9	7	3	9	
1	9	3	8	8	

# Binaire opteller

- Om inzicht te krijgen in het optellen van twee binaire cijfers moeten de vier mogelijkheden bekeken worden.

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ \underline{0} + & \underline{1} + & \underline{0} + & \underline{1} + \\ 0 & 1 & 1 & 10 \end{array}$$

- In de eerste drie gevallen past de uitkomst (de som) ook in één binair cijfer.
- Bij de optelling 1+1 moet het resultaat met twee binaire cijfers worden weergegeven (er is een overloop).

# Optellen

- Optellen in het binaire systeem is identiek aan optellen in het decimale systeem. In totaal moeten er per kolom drie bits worden opgeteld.
- In het voorbeeld worden twee 8-bit getallen opgeteld.

								carry	
getal A →	0	1	1	0	1	1	0	0	
getal B →	0	1	0	1	1	0	1	0	
	<hr/>								+
	1	1	0	0	0	1	1	0	

# Optellen

- Het is mogelijk om een optelschakeling te ontwerpen voor twee binaire getallen.
- Hiervoor wordt een overgang gemaakt van numerieke 0-en en 1-en naar logische 0-en en 1-en:

numeriek 0  $\leftrightarrow$  logische 0

numeriek 1  $\leftrightarrow$  logische 1

- Eerst wordt gekeken naar een optelschakeling voor twee binaire cijfers.



# Half adder

- Voor deze opteller kan een waarheidstabel worden opgesteld.
- De variabelen a en b zijn de aangeboden bits.
- De variabele  $c_{out}$  is het carry-bit en s is het sombit.
- De functies zijn eenvoudig:

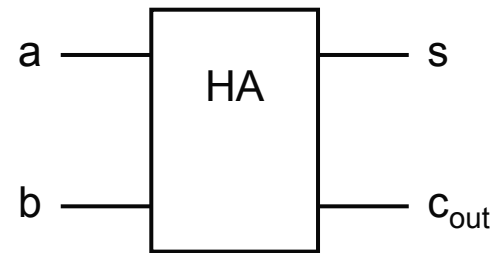
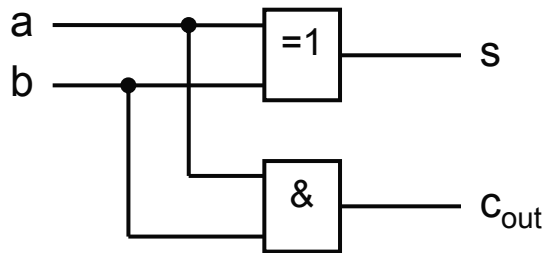
$$c_{out} = a \cdot b$$

$$s = a \oplus b$$

a	b	$c_{out}$	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

# Half adder

- Dit wordt in de digitale techniek een *half adder* genoemd.
- Het schema:



# Full adder

- Een full adder is in staat om drie bits op te tellen.
- De variabelen a en b zijn de bits van de getallen.
- De variabele  $c_{in}$  is de *inkomende* carry.
- $c_{out}$  is de *uitgaande* carry.

$c_{in}$	a	b	$c_{out}$	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

# Full adder

- De functie voor s is eenvoudig te vinden:

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

- Dit kan worden omgewerkt naar:

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

De exor heeft de associatieve eigenschap.

$c_{in}$	a	b	$c_{out}$	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

# Full adder

- De functie voor  $c_{out}$  is als volgt:

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

- Dit kan worden omgewerkt naar:

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

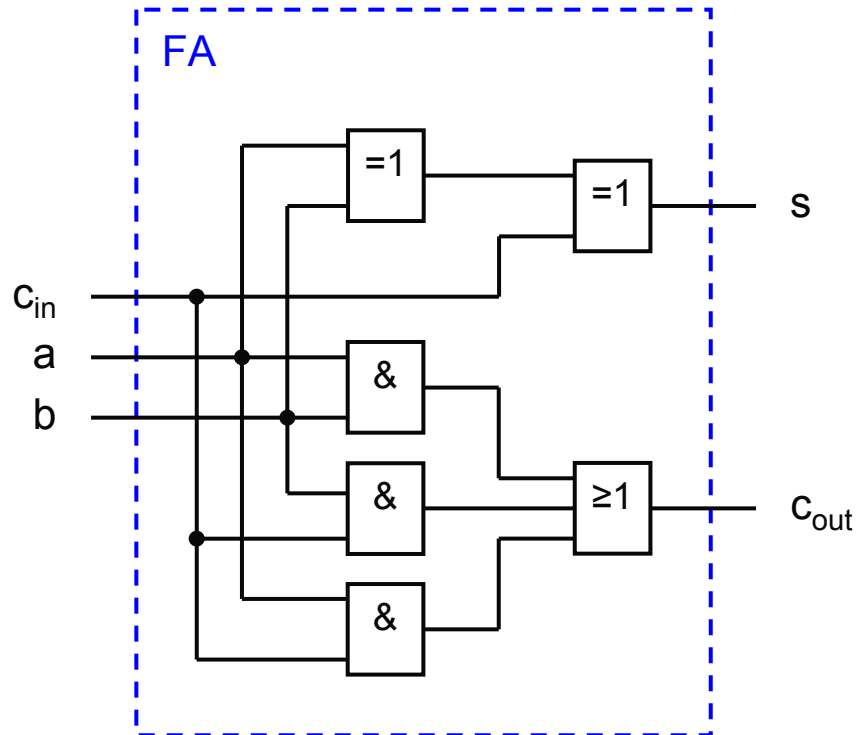
- Maar ook naar:

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

$c_{in}$	$a$	$b$	$c_{out}$	$s$
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

# Full adder

- Het schema kan als volgt worden opgebouwd.

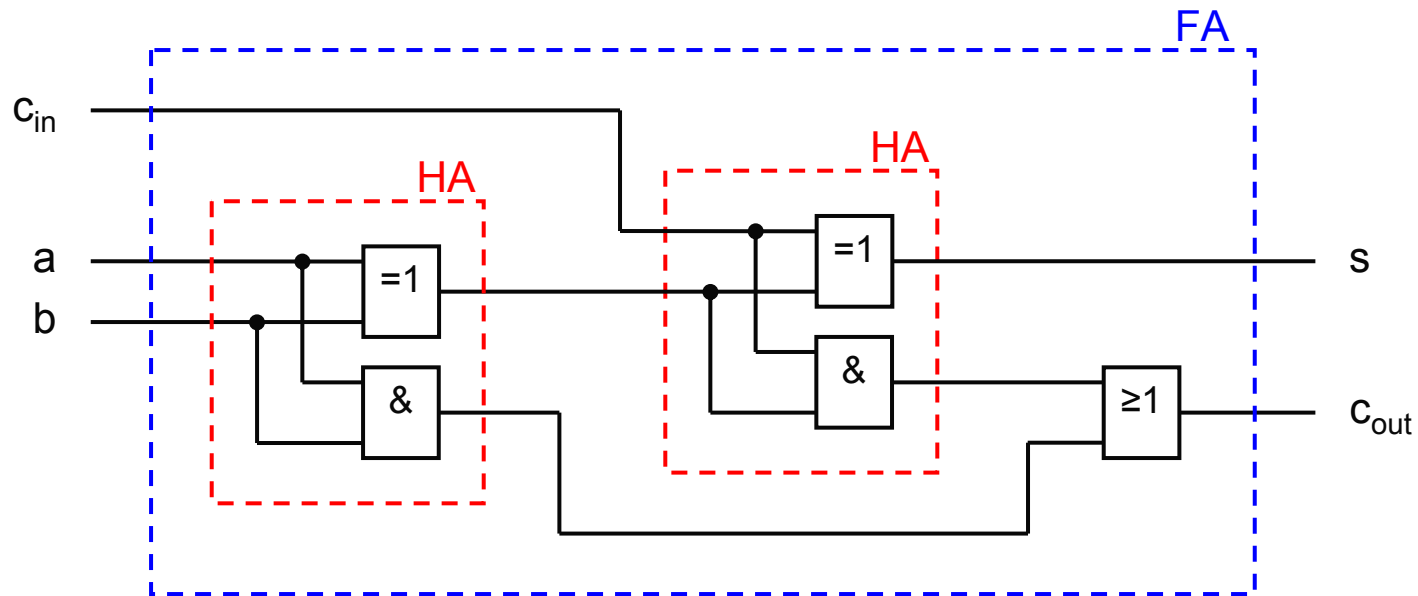


$$s = c_{in} \oplus (a \oplus b)$$

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

# Full adder

- Als alternatief kan de full adder ook als volgt worden opgebouwd.



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

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

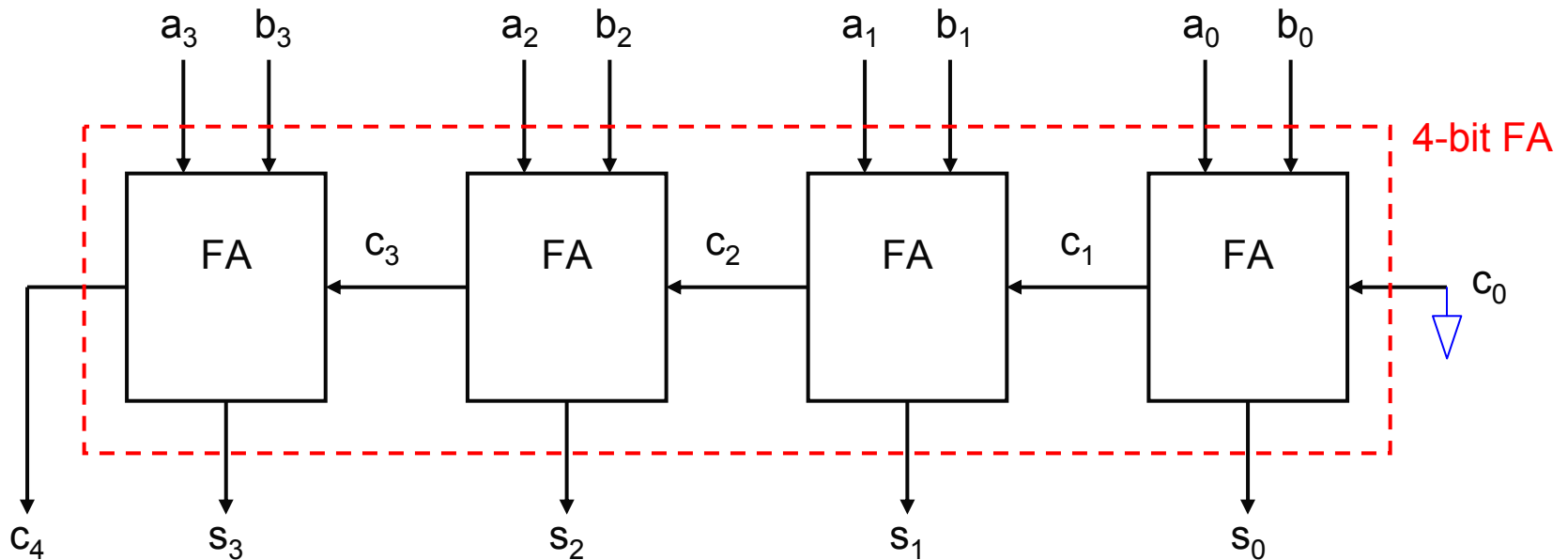
# 4-bit Full Adder

- Een 4-bit opteller kan worden opgebouwd uit een cascade-schakeling van 1-bit full adders.
- De getallen A en B worden opgesplitst in hun afzonderlijke bits. De bits krijgen een index:  $a_3 a_2 a_1 a_0$  en  $b_3 b_2 b_1 b_0$
- De indexnummers komen overeen met de posities van de afzonderlijke binaire cijfers en geven ook de exponent van het gewicht aan ( $a_3 \rightarrow 2^3, \dots$ ).
- De naamgeving van  $c_{in}$  en  $c_{out}$  verandert: de inkomende c-bit krijgt hetzelfde nummer als de a- en b-bits, de uitgaande c-bit krijgt één hoger:  $a_1 b_1 \rightarrow c_{in} = c_1, c_{out} = c_2$



# 4-bit Full Adder

- Een 4-bit opteller kan worden opgebouwd uit een *cascadeschakeling* van 1-bit full adders.



$c_4$  kan als 5<sup>e</sup> sombit gebruikt worden

## 4-bit Full Adder

- Het voordeel van deze realisatie is dat er maar één logische schakeling hoeft worden te ontworpen en het systeem is eenvoudig uitbreidbaar.
- Het nadeel van deze realisatie is dat het lang duurt om de juiste waarde voor de uitgaande  $c_4$ -bit te krijgen. Na het instellen van de getallen A en B en carrybit  $c_0$ , kost het enige tijd voordat  $c_4$  beschikbaar is.
- Dit wordt een *ripple carry adder* genoemd.
- Deze vertraging heeft geleid tot een scala aan andere implementaties: carry look-ahead, carry-select, carry-skip, carry-completion. Deze implementaties zijn allemaal bedoeld om het berekenen van de carry's te versnellen.

# Opgaven

- Tel de volgende binaire getallen op:

$$1011001 + 0111011 \quad 01010 + 01010 \quad 01111 + 00001$$

- Toon aan dat:  $c_{out} = \overline{c_{in}} \cdot (a \cdot b) + c_{in} \cdot (a + b) = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$
- Toon aan dat:  $c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in} = a \cdot b + c_{in} \cdot (a \oplus b)$
- Als de functies van  $s$  en  $c_{out}$  vanuit de 0-en zouden worden gemaakt, wordt de functie dan kleiner (minder poorten)?


# Tellen

- De bewerking *tellen* komt in veel applicaties voor. Meestal betreft het toepassingen waarbij wordt bijgehouden hoeveel keer een bepaalde gebeurtenis optreedt.
- Tellen wordt meestal gedaan in het binaire stelsel, maar het is heel goed mogelijk in het decimale talstelsel te tellen. In dit geval worden de decimale cijfers in de BCD-code voorgesteld.
- Tellers hebben de eigenschap een getelde hoeveelheid te *onthouden*. Dat betekent dat tellers *geheugen* bezitten.

# Tellen

- Tellers worden vrijwel altijd *modulair* opgebouwd (bv. in processoren: 8 bits, 16 bits). In de BCD-code is dat van nature vier bits.
- Een cyclus van een *3-decaden* teller loopt van  $000_{\text{BCD}}$  tot  $999_{\text{BCD}}$ , waarna de teller weer in  $000_{\text{BCD}}$  start. Zo'n teller heet cyclisch.

000 → 001 → 002 → 003 → ... → 099 → 100 → 101 →  
102 → ... → 999 → 000 → 001 → 002 → ...

 cyclus start opnieuw

# Telcyclus 4-bit teller

- Hieronder een voorbeeld van een 4-bit binaire teller.

0000 → 0001 → 0010 → 0011 → 0100 → 0101 → 0110 → 0111 →  
1000 → 1001 → 1010 → 1011 → 1100 → 1101 → 1110 → 1111 →  
0000

↑  
└───┘ cyclus start opnieuw

- Duidelijk is dat bij de huidige telstand steeds 1 wordt opgeteld om de nieuwe telstand te krijgen. Dit kan dus met een opteller waarvan één getal de vaste waarde 0001 krijgt.

# Verhogen met één

- Als voorbeeld verhogen we onderstaand getal met 1.
- Verwisselen van  $c_0$  met  $b_0$  levert iets moois op:

$$\begin{array}{rcccc} & 0 & 1 & 1 & 0 & & c_0 \\ \hline & 1 & 0 & 1 & 1 & & \\ & 0 & 0 & 0 & 1 & & \\ \hline & 1 & 1 & 0 & 0 & & + \end{array} \quad \longrightarrow \quad \begin{array}{rcccc} & 0 & 1 & 1 & 1 & & c_0 \\ \hline & 1 & 0 & 1 & 1 & & \\ & 0 & 0 & 0 & 0 & & \\ \hline & 1 & 1 & 0 & 0 & & + \end{array}$$

- B is 0!

# Waarheidstabel

- De waarheidstabel voor de full adder kan aanzienlijk vereenvoudigd worden.
- Alle regels met  $b = 1$  kunnen worden geschrapt.
- Alleen de regels met  $b = 0$  blijven over.
- Aangezien  $b$  altijd 0 is kan deze kolom worden geschrapt.

$c_{in}$	a	b	$c_{out}$	s
0	0	0	0	0
<del>0</del>	<del>0</del>	<del>1</del>	<del>0</del>	<del>1</del>
0	1	0	0	1
<del>0</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>0</del>
1	0	0	0	1
<del>1</del>	<del>0</del>	<del>1</del>	<del>1</del>	<del>0</del>
1	1	0	1	0
<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>



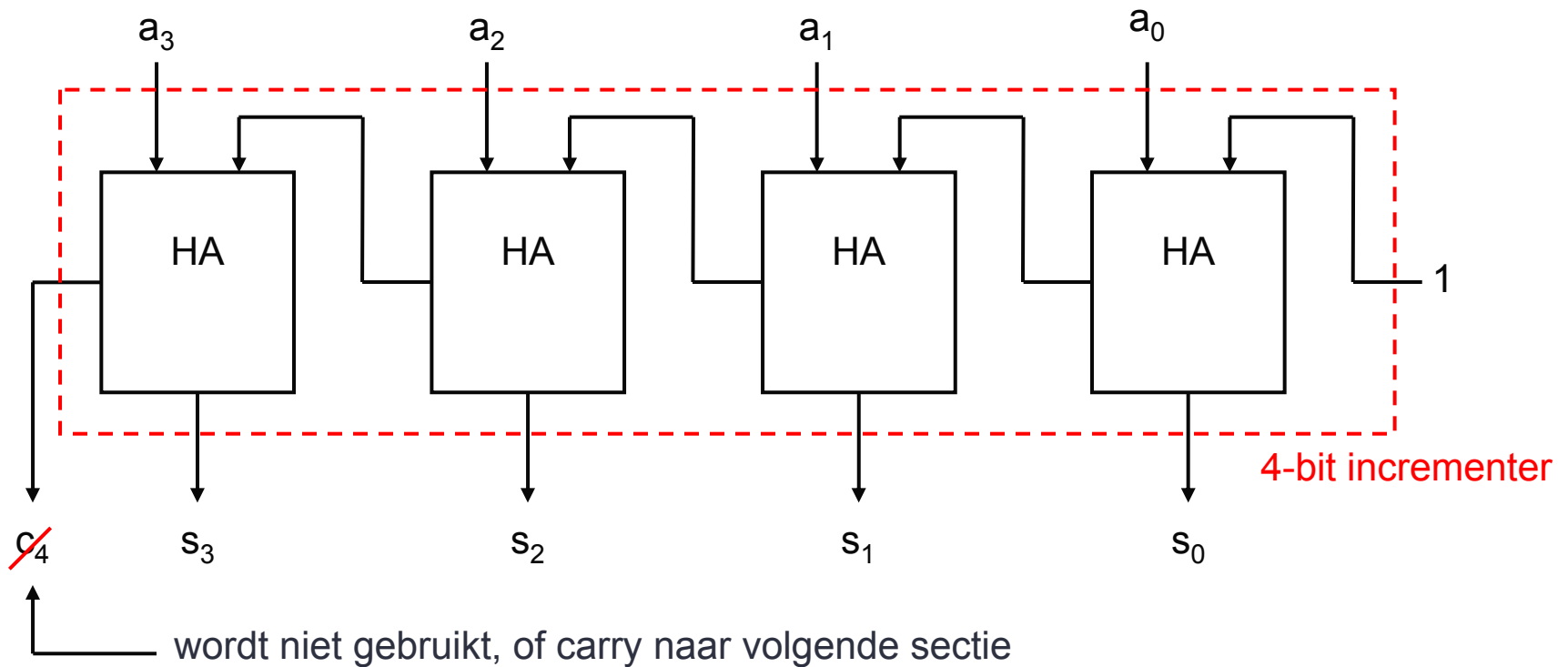
# Half adder

- De waarheidstabel wordt vereenvoudigd.
- Dit is een half adder.
- Een telschakeling kan dus gemaakt worden door een cascadeschakeling van half adders.

$c_{in}$	a	$c_{out}$	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

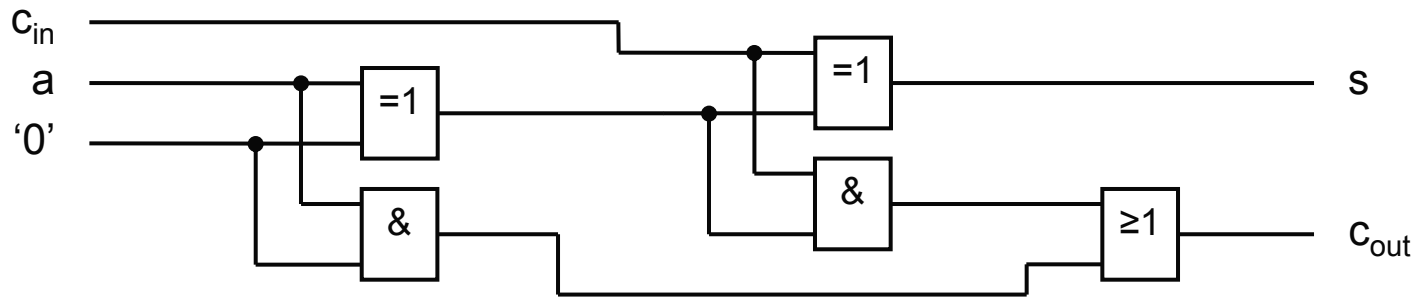
# 4-bit incrementer

- Hieronder het resultaat. Merk op dat de carry-ingangen nu verdwenen zijn en de carry-uitgangen zijn verbonden met de b-ingangen.



# Opgave

- Hieronder is de full adder die eerder is besproken nog eens afgebeeld, maar nu is de b-ingang aan een logische 0 gekoppeld. Vereenvoudig het schema (“minimaliseer b weg”). Doe hetzelfde voor b is logisch 1.



# Subtractor

- Op eenzelfde wijze als het ontwerpen van een optelschakeling, kan ook een aftrekschakeling gemaakt worden.
- In de praktijk wordt echter een optelschakeling gebruikt en wordt de wiskundige gelijkheid gebruikt:

$$A - B = A + (-B)$$

- Dit vereist echter wel het gebruik van *negatieve getallen*. Negatieve binaire getallen worden later behandeld.

# Vermenigvuldigen

- Het vermenigvuldigen van twee getallen is erg gemakkelijk in het binaire systeem. Er zijn maar drie tafels nodig: voor 0, 1 en 10.
- Als voorbeeld een vermenigvuldiging in het decimale systeem.

$$\begin{array}{r} 391 \\ 283 \\ \hline 1173 \quad x \\ 31280 \quad \leftarrow \text{één plek opschuiven, want 8 is een tiental} \\ 78200 \quad \leftarrow \text{twee plekken opschuiven, want 2 is een honderdtal} \\ \hline 110653 \quad + \quad \leftarrow \text{lastig, meerder rijen optellen} \end{array}$$

deelvermenigvuldigingen leveren deelantwoorden die groter zijn dan 9.

# Vermenigvuldigen

- In het binaire systeem werkt het net zo:

$$\begin{array}{r} 1010 \\ 1110 \\ \hline 0000 \end{array} \times$$

Vermenigvuldigen is eenvoudig:

10100

Vermenigvuldigen met 0 levert 0!

101000

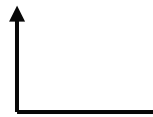
Vermenigvuldigen met 1 levert getal!

1010000

0-en schuiven voor tweetal, viertal, ...

$$\begin{array}{r} 1010000 \\ \hline 10001100 \end{array} +$$

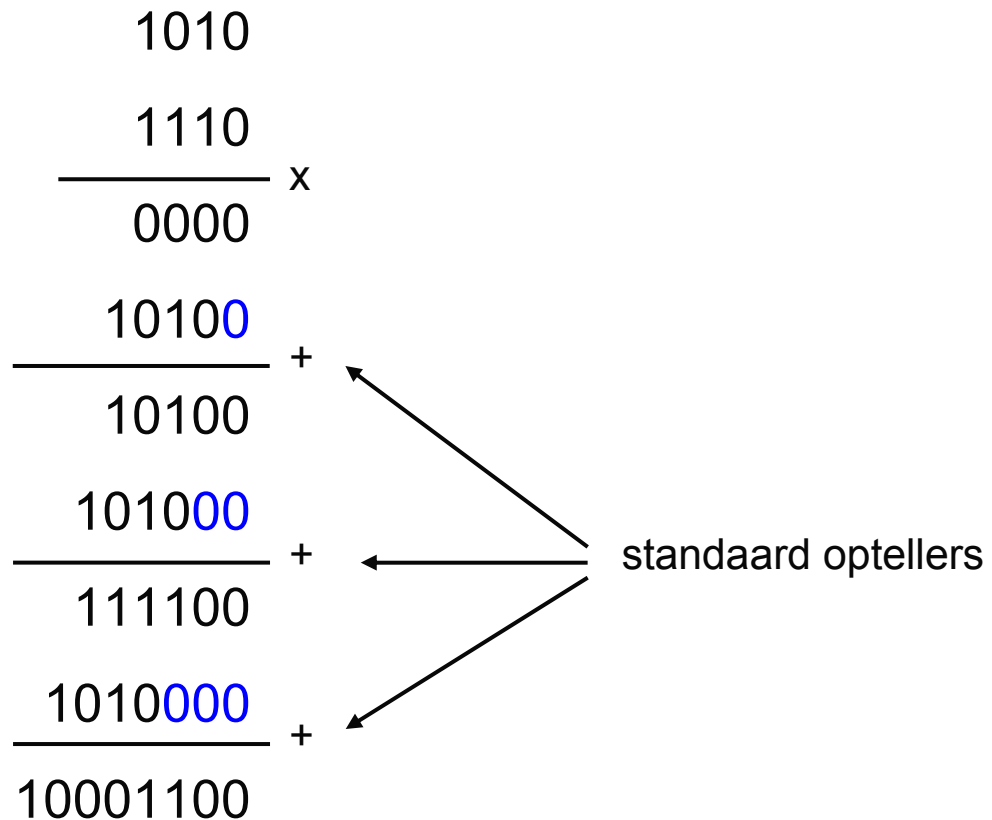
Nadeel: multi-input opteller nodig



Maximaal 4+4 = 8 cijfers

# Vermenigvuldigen

- De multi-input opteller kan vermeden worden door tussentijds op te tellen:



# Vermenigvuldigen

- Er zijn maar twee tafels nodig: de tafel van 0 en van 1. Deze kunnen gecombineerd worden.

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

- Dit is precies de tabel van een AND!
- Een vermenigvuldiger is te bouwen uit ANDs en optellers.



# Vermenigvuldigen

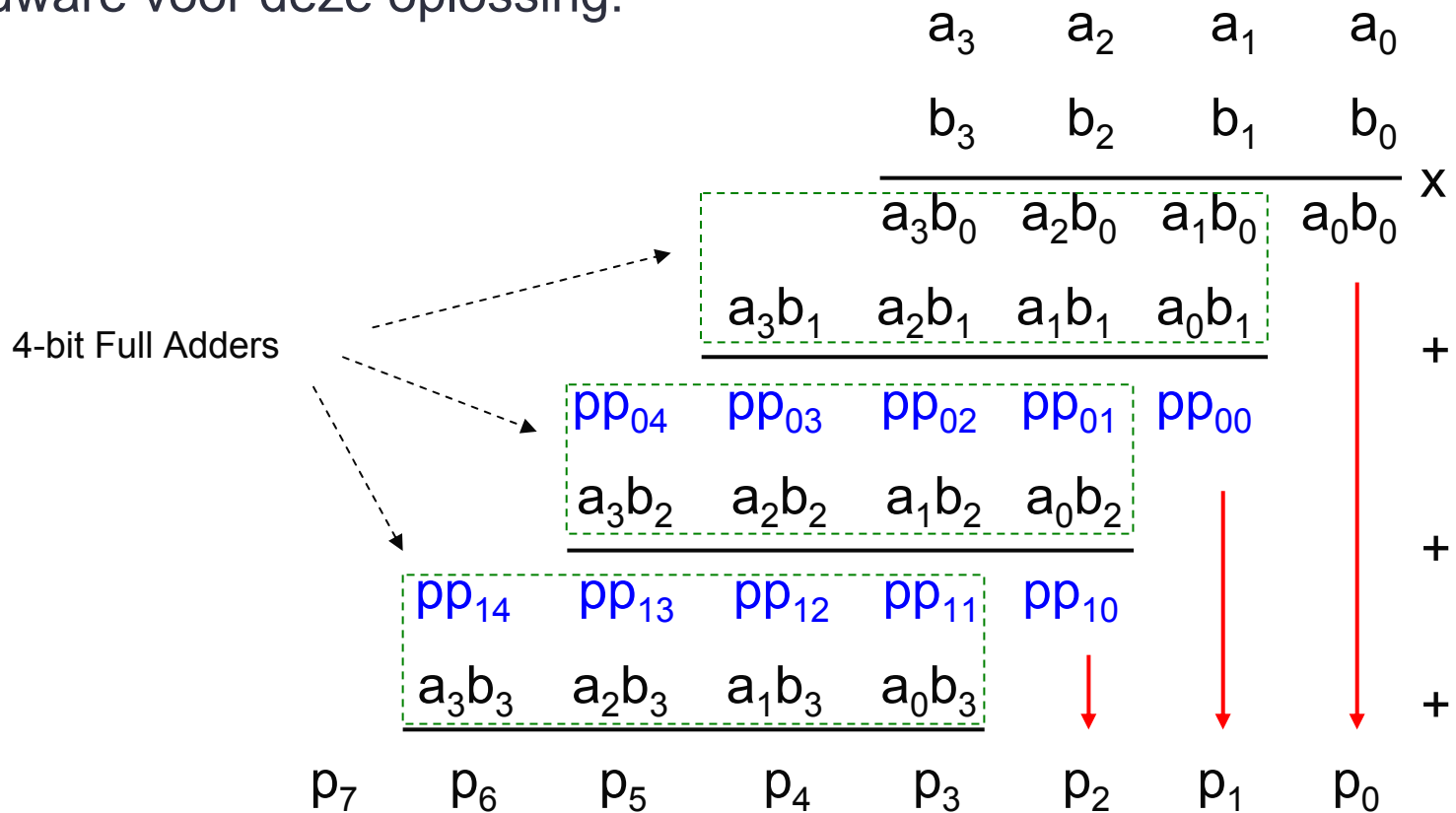
- Hardware ontwikkelen gaat ook eenvoudig:

$$\begin{array}{r}
 \begin{array}{cccc}
 & & & a_3 & a_2 & a_1 & a_0 \\
 & & & b_3 & b_2 & b_1 & b_0 \\
 \hline
 & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & 0 \\
 & & & & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & 0 & 0 \\
 & & & & & & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & 0 & 0 & 0 \\
 \hline
 & & & & & & & & & & & & & + \\
 p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
 \end{array}
 \end{array}$$

$p = \text{product term}$

# Vermenigvuldigen

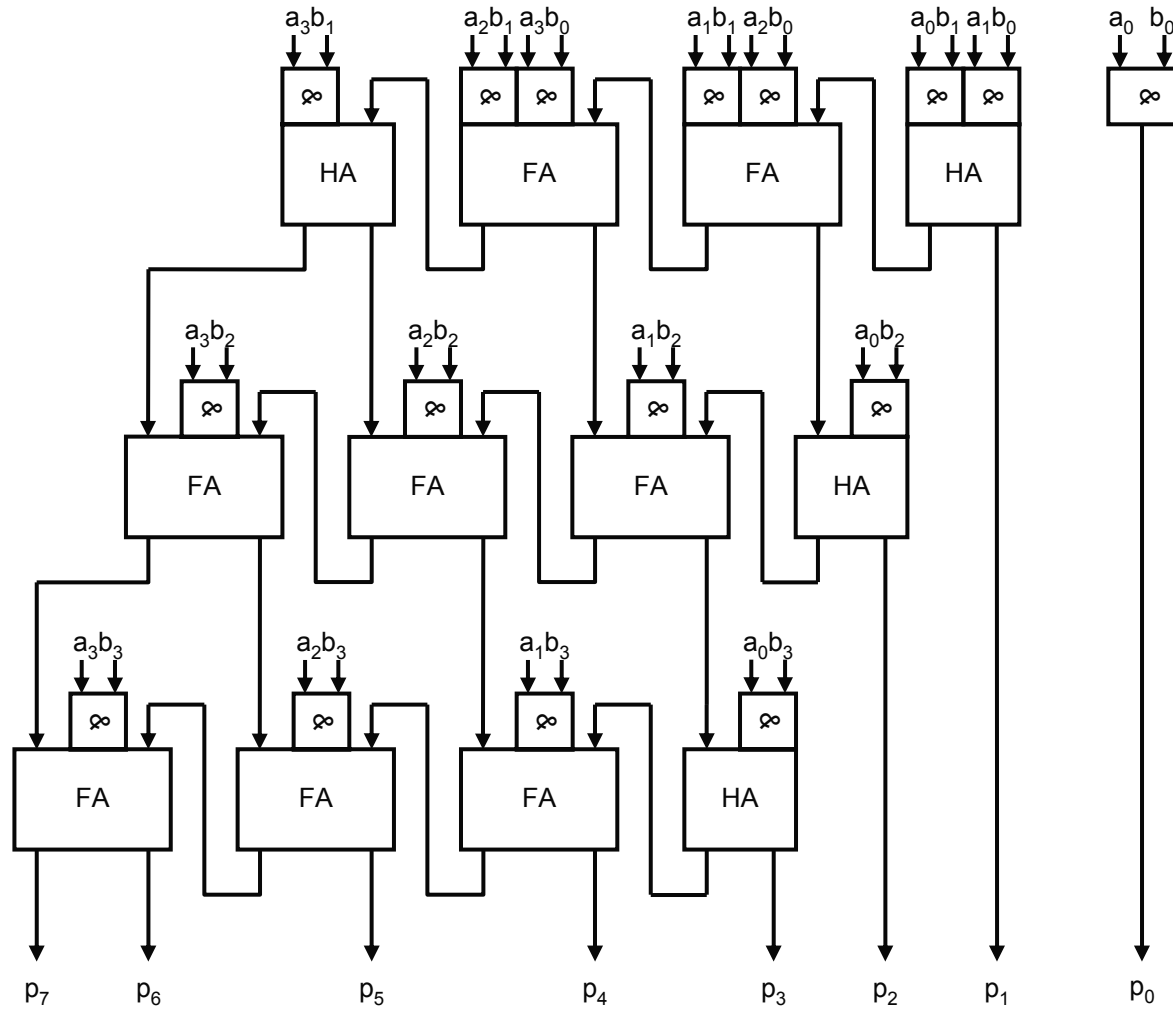
- Hardware voor deze oplossing:



$pp$  = partial product term  
 $p$  = product term

# Vermenigvuldigen

- Hardware:



# Vermenigvuldigen

- Het langste pad van  $a_1b_0$  of  $a_0b_1$  naar  $p_7$  is 8 optelsecties.
- Het kan slimmer met een carry-save structuur.
- Dit wordt niet besproken.

# Vermenigvuldigen

- Natuurlijk kan een vermenigvuldiger ook volgens de bekende oplossingsstructuur van digitale systemen worden ontworpen.
- Stel een waarheidstabel op met 2x vier ingangen en acht uitgangen:

$a_3 a_2 a_1 a_0$	$b_3 b_2 b_1 b_0$	$p_7$	$p_6$	$p_5$	$p_4$	$p_3$	$p_2$	$p_1$	$p_0$
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
	....								
1	1	1	1	0	1	0	0	1	0
1	1	1	1	1	1	1	1	1	1

# Vermenigvuldigen

- Dit levert echter zeer veel logica op. Een groot gedeelte van het IC-oppervlakte wordt dan gebruikt voor de multiplier. Let hier op tijdens het gebruik van de "\*"-operator in VHDL.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity vmul8x8i is
    port (
        x: in  unsigned (7 downto 0);
        y: in  unsigned (7 downto 0);
        p: out unsigned (15 downto 0);
    );
end vmul8x8i;

architecture vmul8x8i_arch of vmul8x8i is
begin
    p <= x*y;
end vmul8x8i;
```

# Vermenigvuldigen met een constante

- Een vermenigvuldiging met een constante kan eenvoudig worden omgezet naar een serie optellingen.
- Bijvoorbeeld: vermenigvuldigen met  $13_{10} \times 11_{10} = 1101_2 \times 1011_2$
- Het getal  $11_{10}$  is te schrijven als  $8 + 2 + 1 = 2^3 + 2^1 + 2^0$
- Dus de vermenigvuldiging is  $13 \cdot 8 + 13 \cdot 2 + 13 \cdot 1 = 13 \cdot 2^3 + 13 \cdot 2^1 + 13 \cdot 2^0$

# Vermenigvuldigen met een constante

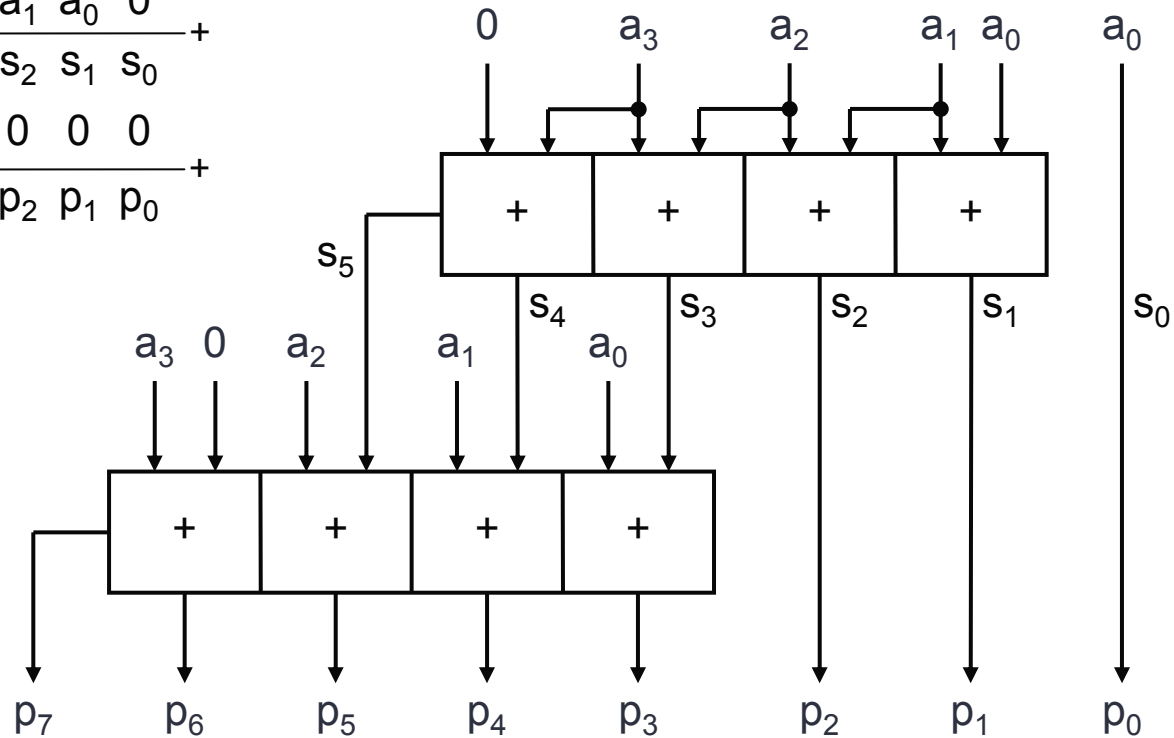
- Nu is vermenigvuldigen met 8 ( $2^3$ ) niets anders dat drie plaatsen naar links schuiven en aanvullen met nullen.
- Vermenigvuldigen met 2 ( $2^1$ ) is één plaats naar links schuiven en aanvullen met nullen.
- $1101 \times 1011 = 1101\underline{000} + 1101\underline{0} + 1101$
- Vermenigvuldigen van een 4-bit getal  $a_3a_2a_1a_0$  met  $1011_2$ :

$$a_3a_2a_1a_0 \times 1011_2 = a_3a_2a_1a_0000 + a_3a_2a_1a_00 + a_3a_2a_1a_0$$



# Vermenigvuldigen met een constante

$$\begin{array}{r}
 0 \ a_3 \ a_2 \ a_1 \ a_0 \\
 a_3 \ a_2 \ a_1 \ a_0 \ 0 \\
 \hline
 0 \ s_5 \ s_4 \ s_3 \ s_2 \ s_1 \ s_0 \\
 a_3 \ a_2 \ a_1 \ a_0 \ 0 \ 0 \ 0 \\
 \hline
 p_7 \ p_6 \ p_5 \ p_3 \ p_3 \ p_2 \ p_1 \ p_0
 \end{array}$$



# Opgaven

- Ontwerp volgens de bekende oplossingsstructuur van digitale systemen een 2x2-bit vermenigvuldiger.
- Ontwerp een schakeling die test of twee niet-negatieve 4-bit getallen gelijk zijn.
- Ontwerp een 4x4 bit carry save multiplier (tip: uiteraard heeft iemand dat allang gedaan).
- Hoeveel optellers zijn er nodig voor een 5x3-bit vermenigvuldiger? Hoe breed zijn de optellers?

# Delen

- Delen gaat op vergelijkbare wijze als in het decimale systeem:

$$101101101 : 1010 = \underline{100100,1}$$

$$\underline{1010} -$$

$$1011$$

$$\underline{1010} -$$

$$101,0$$

$$\underline{101,0} -$$

$$\underline{0}$$

$$365 : 10 = 36,5$$

$$\underline{30}$$

$$65$$

$$\underline{60}$$

$$5,0$$

$$\underline{5,0}$$

$$\underline{0}$$

- Het algoritme is gebaseerd op “aftrekken als het mogelijk is” en het “bijtrekken” van de volgende cijfers. Combinatorische delers leveren veel hardware op.

# Referenties

- De volgende boeken zijn gebruikt:

Digitale techniek, van probleemstelling tot realisatie deel 1; A.P. Thijssen; 5<sup>e</sup> druk.

Digital Design, Principles and Practices; J.F. Wakery; 3<sup>th</sup> Ed.

Fundamentals of Digital Logic with VHDL Design, S. Brown, 3<sup>th</sup> Ed.

Dictaat hoofdstuk 5.

# Carry lookahead

- Een 4-bit full adder ontworpen als ripple carry adder heeft als nadeel dat het veel tijd kost voordat  $c_4$  beschikbaar is, ongeveer 8 poortvertragingen.
- Maar  $c_4$  kan natuurlijk ook geschreven worden als functie van de ingangen  $a_3$  t/m  $a_0$ ,  $b_3$  t/m  $b_0$  en  $c_0$ .
- Dit levert echter heel veel hardware op.
- Slimmer is om uit te gaan van de functie voor de carry.

# Carry lookahead

- De carry voor de eerste 1-bit full adder kan geschreven worden als:

$$c_1 = a_0 \cdot b_0 + (a_0 + b_0) \cdot c_0$$

- Er worden nu twee hulpfuncties geïntroduceerd:

$$G_0 = a_0 \cdot b_0$$

$$P_0 = a_0 + b_0$$

- G staat voor *carry generate*, want het genereert een carry  $c_1$  onafhankelijk van de  $c_0$ .
- P staat voor *carry propagate*, want het geeft een eventuele  $c_0$  door aan  $c_1$ .

# Carry lookahead

- De functie voor  $c_1$  is nu te schrijven als

$$c_1 = G_0 + P_0 \cdot c_0$$

- Maar dan is voor  $c_2$  te schrijven

$$c_2 = G_1 + P_1 \cdot c_1$$

- In deze functie kan de functie voor  $c_1$  ingevuld worden

$$c_2 = G_1 + P_1 \cdot c_1 = G_1 + P_1 \cdot (G_0 + P_0 \cdot c_0) = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0$$

# Carry lookahead

- En dan kan de functie voor  $c_3$  ook uitgewerkt worden:

$$\begin{aligned}c_3 &= G_2 + P_2 \cdot c_2 \\ &= G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0) \\ &= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0\end{aligned}$$

- En natuurlijk uiteindelijk de functie voor  $c_4$ :

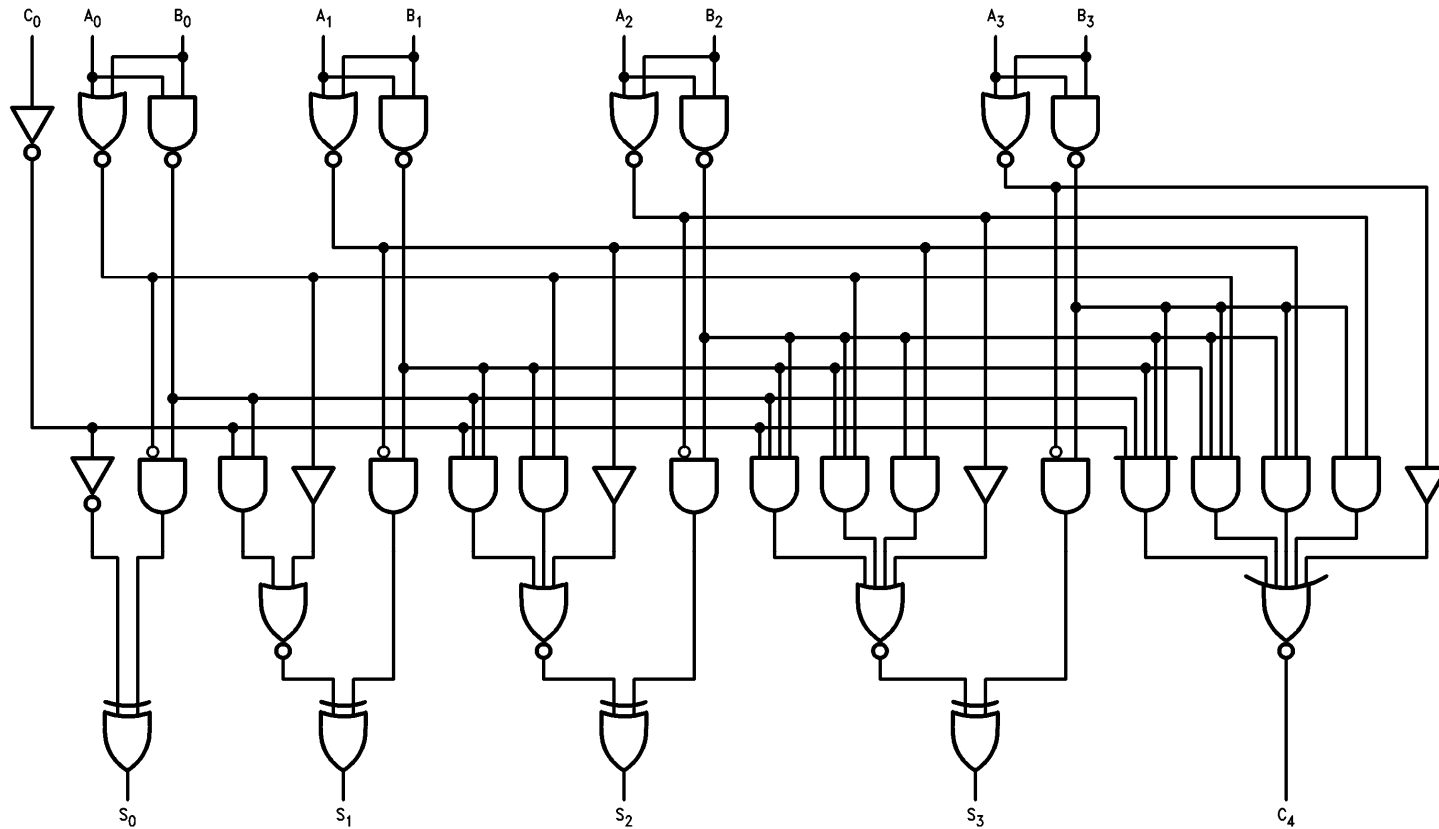
$$\begin{aligned}c_4 &= G_3 + P_3 \cdot c_3 \\ &= G_3 + P_3 \cdot (G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0) \\ &= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0\end{aligned}$$



# Carry lookahead

- De functie voor  $c_4$  is nu te maken met AND2, AND3, AND4, AND5 en een OR4.
- Samen met de P- en G-hulpfuncties levert dit een schakeling die maximaal drie poortvertragingen heeft.
- Deze realisatie van carry-propagatie heet *carry lookahead*.
- Op de volgende slide staat een schema voor een 4-bit Full Adder. Merk op dat de inversen van P en G gegenereerd worden, dat levert snellere logica op.

# Carry lookahead



Uitvoering van de SN74283 4-bit full adder. Merk op dat de inversen van P en G gegenereerd worden.



Academie voor Technology, Innovation &  
Society Delft  
Academie voor ICT & Media

De Haagse Hogeschool, Delft  
+31-15-2606311  
J.E.J.opdenBrouw@hhs.nl  
www.dehaagsehogeschool.nl

**DE HAAGSE**  
HOGESCHOOL