Логичко пројектовање

Предавање 9/10

Аритметичка логичка кола

Увод

- Ова лекција даје преглед пројектовања логичких кола за извршавање основних аритметичких операција:
  - сабирање
  - одузимање
  - множење
  - дељење
- Такође је презентовано како се аритметичка кола моделују у VHDL–у.
  - сабирачи
  - одузимачи

Сабирачи

- Сабирање бинарних бројева се изводи на сличан начин као и сабирање децималних бројева.
- Сабирање почиње на позицији најмање тежине броја (п = 0). Сабирање даје збир за ову позицију. У случају да збир за ову позицију не може да се представи једним симболом, тада се симбол вишег реда преноси на следећу позицију (п = 1). Сабирање у следећој вишој позицији мора укључити број који је пренет (carry bit) из сабирања на нижој позицији. Овај процес се наставља до свих симбола у броју.
- Коначна сума на највишој позицији такође може произвести пренос, што треба узети у обзир у посебном систему.
- Пројектовање бинарног сабирача укључује креирање комбинационог логичког кола за сабирање бројева на једној позицији. Пошто комбинационо логичко коло може само произвести скаларни излаз, потребна су кола за креирање суме и преноса за сваку позицију.
- Величина бинарног сабирача је унапред одређена и фиксирана пре имплементације логике (тј. n-битни сабирач).
- Оба улаза у сабирач морају бити фиксне величине, без обзира на њихову вредност. Мањи бројеви једноставно садрже на вишим позивцијама нуле.
- За n-битни сабирач, највећа сума која се може произвести ће захтевати n+1 бит. Да би ово илустровао,размотра се 4-битни сабирач. Највећи бројеви на којима ће сабирач радити су $1111_2 + 1111_2$. (или $15_{10} + 15_{10}$). Резултат овог додавања је 11110_2 (или 30_{10}).
- Приметите да се највећи произведени збир уклапа у 5 битова, или n+1. Када се пројектује коло за сабирање, за збир се увек користе n-битова са одвојеним битом за пренос. У нашем 4-битном примеру, збир би био изражен као „1110" са преносом.
- Бит преноса се може користити код сабирања више речи, које се користе као део броја када се декодирају за приказ, или једноставно се одбацује као у случају када се користе бројеви у двоструком комплементу.

Полу-сабирачи

- Приликом креирања сабирача пожељно је пројектовати инкременталне подсистеме који могу бити поново коришћени.
- Ово смањује време потребно за пројектовање и минимизије сложеност решавања проблема.
- Најосновнија компонента у сабирачу се зове полусабирач. Ово коло израчунава збир и то извршава на два улазна аргумента.
- Разлог због којег се зове полу-сабирач уместо пуног сабирача је зато што не прихвата пренос током израчунавања, тако да не пружа све што је потребно функционалност позиционог сабирача.
- Пример приказује архитектуру полу-сабирача. За полу-сабирач су потребна 2 комбинациона логичка кола, за креирање збира (XOR коло), и за пренос (AND коло).
- Ова два кола су паралелна једна са другим, тако да је кашњење кроз полусабирач последица само једног нивоа логике.

Архитектура полу-сабирача

Example: Design of a Half Adder

Recall in binary addition, the output consists of a sum and a carry bit.

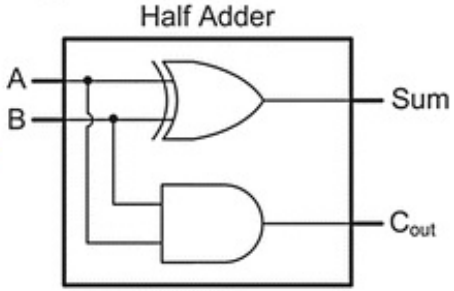
$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \leftarrow \text{Sum} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 1 \\ \hline 10 \end{array} \leftarrow \text{Carry}$$

We can build a simple circuit callled a "Half Adder" to compute these outputs.

A	B	C_{out}	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$\text{Sum} = A \oplus B$

$C_{out} = A \cdot B$



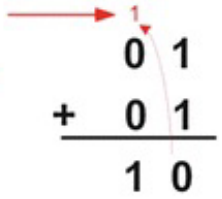
Потпуни сабирач

- Потпуни сабирач је коло које креира збир и пренос, али разматра три улаза у прорачунима (A, B и C_{in}). Пример приказује пројектовање пуног сабирача.

Example: Design of a Full Adder

In order to create multi-bit adders, a circuit is needed that also includes a "Carry In" bit.

The sum of position 1 needs to include the "Carry Out" from the sum of position 0. The sum of position 1 must include this carry, which is reffered to as the "Carry In" bit.



This circuit is called a "Full Adder".

C_{in}	A	B	C_{out}	Sum
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

Sum

B \ C_{in} A	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$Sum = A \oplus B \oplus C_{in}$$

C_{out}

B \ C_{in} A	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$C_{out} = A \cdot C_{in} + A \cdot B + B \cdot C_{in}$$
$$= A \cdot B + (A+B) \cdot C_{in}$$

Потпуни сабирач – логички изрази

- Као што је раније поменуто, пожељно је поново користити компоненте које су већ пројектоване за креирање сложенијих система.
- Један такав приступ поновној употреби пројекта је креирање пуног сабирача помоћу два полусабирача.
- Ово је једноставно за излаз за сабирање пошто је логика једноставна где имамо два каскадна XOR кола ($Sum = A \oplus B \oplus C_{in}$).
- Логика за пренос није тако једноставна. Примећује се да израз за C_{out} изведен у примеру садржи терм (A+B). Ако би се овај терм могао извести са XOR колом, то би пуном сабирачу омогућило да искористи предности постојећег кола у систему.
- Следећа слика приказује извођење еквиваленције која дозвољава да се терм (A+B) замени са ($A \oplus B$) у C_{out} логичком изразу.

Потпуни сабирач – логичка еквиваленција

A Useful Logic Equivalency that can be Exploited in Arithmetic Circuits

The logic expression for the carry out of a full adder was given as: $C_{out} = A \cdot B + (A + B) \cdot C_{in}$. It turns out that the exact same output is produced by the expression $A \cdot B + (A \oplus B) \cdot C_{in}$. Let's examine how this is possible by breaking down the expressions into their individual parts and solving at each step.

FA Inputs			Desired Output	$C_{out} = A \cdot B + (A + B) \cdot C_{in}$			$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$		
C_{in}	A	B	C_{out}	$A \cdot B$	$(A+B) \cdot C_{in}$	$A \cdot B + (A + B) \cdot C_{in}$	$A \cdot B$	$(A \oplus B) \cdot C_{in}$	$A \cdot B + (A \oplus B) \cdot C_{in}$
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	1	1	1	1	0	1	1	0	1
1	0	0	0	0	0	0	0	0	0
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1	0	1

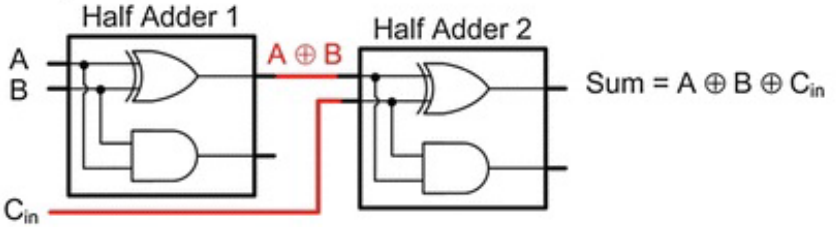
$C_{out} = A \cdot B + (A + B) \cdot C_{in} = A \cdot B + (A \oplus B) \cdot C_{in}$

Equivalent !

Архитектура потпуног сабирача

Example – Design of a Full Adder Out of Two Half Adders

It is often desirable to create a full adder out of two half adders in order to re-use existing design components. The "Sum" of the full adder can be created by using two cascaded XOR gates provided by the half adders.



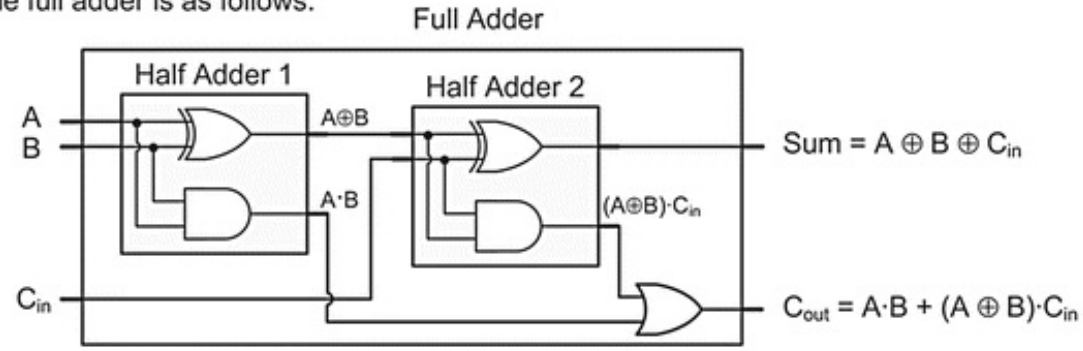
The expression for the "Carry Out" of the full adder is:

$$C_{out} = A \cdot B + (A + B) \cdot C_{in}$$

or

$$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$$

Notice that the carry out of Half Adder 1 produces the A·B term in this expression. Also notice that the carry out of Half Adder 2 produces the $(A \oplus B) \cdot C_{in}$ term. The only remaining logic needed to create the carry out of the full adder is an OR gate. The final logic diagram for the full adder is as follows:



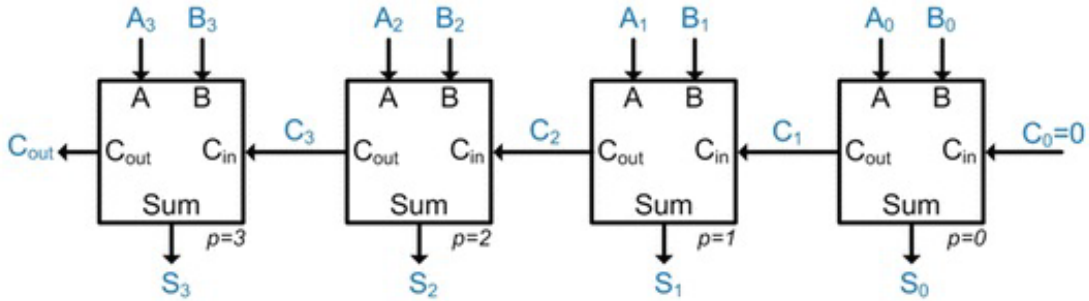
Ripple Carry Adder (RCA) сабирач

- Потпуни сабирач се сада може користити код креирања вишебитних сабирача.
- Најједноставнија архитектура која користи потпуни сабирач назива се сабирач са таласастим преносом.
- Код овог приступа, потпуни сабирачи се користе за креирање збира и преноса на следећу вишу позицију.
- Пренос сваког пуног сабирача се користи као пренос за следећи виши ниво. Пошто сваки следећи потпуни сабирач треба да сачека пренос произведен у претходној фази, за пренос се каже да се таласа кроз коло, по коме је овај приступ добио име.
- Пример показује како се пројектује 4-битни RCA сабирач користећи ланац потпуних сабирача. Примећује се да је пренос 0 за сабирач на позицији 0.
- Улаз 0 нема утицаја на резултат збира, али омогућава коришћење пуног сабирача на 0. позицији.

Пример 4-битног RCA сабирача

Example: Design of a 4-Bit Ripple Carry Adder (RCA)

Full adders can be cascaded together to form a multi-bit adder. The symbols are typically drawn in the following fashion to mirror a positional number system.



The sum of position 1 cannot complete until it receives the carry in (C_1) from the sum in position 0. The position 2 sum cannot complete until it receives the carry in (C_2) from the sum in position 1, etc. In this way, the carry "ripples" through the circuit from right to left. This configuration is known as a Ripple Carry Adder (RCA).

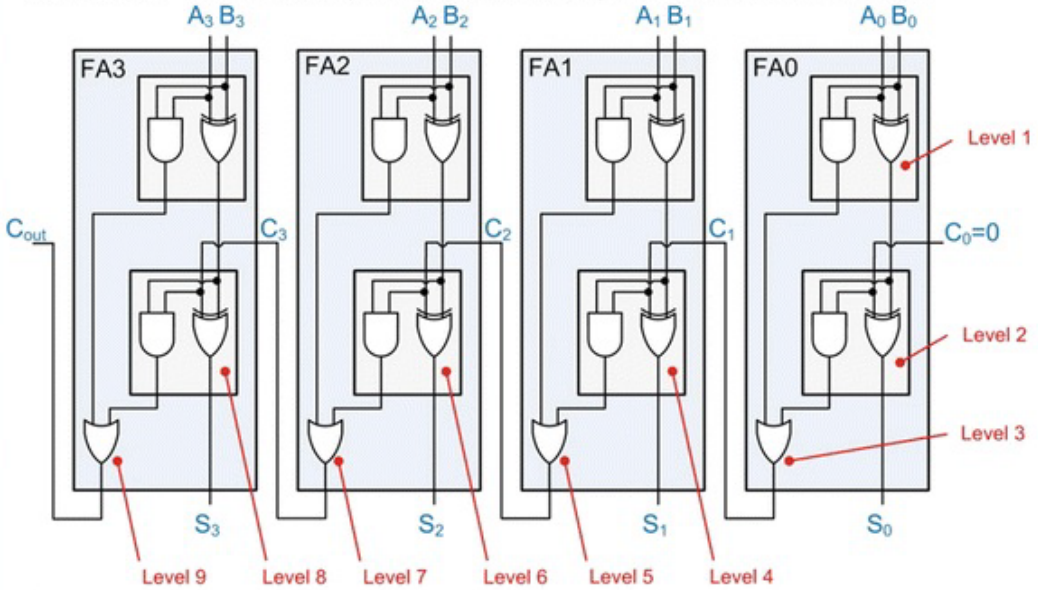
Кашњење код RCA сабирача

- Док RCA сабирач пружа једноставну архитектуру засновану на пројектовању са поновним коришћењем компоненти, његово кашњење може постати значајно при скалирању на веће величине улаза (нпр. n = 32 или n = 64).
- Једноставна анализа времена може се извршити тако да ако се време да потпуни сабирач изврши свој позициони збир означи са t_{FA}, онда се време за n-битни сабирач таласа за извршење израчунавања може израчунати као $t_{RCA} = n \cdot t_{FA}$.
- Ако детаљније испитамо RCA сабирач, можемо разложити кашњење у нивоима логике неопходних за израчунавање.
- Пример показује анализу времена 4-битног RCA сабирача. Ова анализа одређује број логике нивоа у сабирачу. Стварна кашњења кола се онда могу укључити да би се пронашло коначно кашњење. Улази у сабирач су A, B и C_{in} и увек се претпоставља да су ажурирани у исто време.
- Први потпуни сабирач захтева два нивоа логике да би се креирао збир и три нивоа да би се произвео његов пренос. Обзиром да се за коло увек узима његово најгоре време кашњења, кажемо да први потпуни сабирач узима три нивоа логике. Када се пренос (C_1) прослеђује до следећег пуног сабирача
- (FA1), мора се пропагирати кроз два додатна нивоа логике да би се креирало C_2 Примећује се да први полусабирач у FA1 зависи само од A_1 и B_1, тако да је у стању да одмах изврши ово израчунавање. Овај полусабирач сматра се логиком првог нивоа. Што је још важније, то значи да када стигне (C_1), потребна су само 2 додатна нивоа логике, а не 3.
- Нивои логике за RCA сабирач могу се изразити као $3 + 2\cdot(n - 1)$. Ако сваки ниво логике има кашњење t_{gate}, тада је тачнији израз за RCA кашњење $\tau_{RCA} = (3 + 2\cdot(n - 1))\cdot t_{gate}$.

Кашњење код 4-битног RCA сабирача

Example: Timing Analysis of a Ripple Carry Adder

The carry ripples through the adder chain. The first full adder (FA0) takes three levels of logic to complete. When C_1 reaches the FA1, its first half adder has already completed its computation. This means only two more levels of logic are needed to compute C_2.



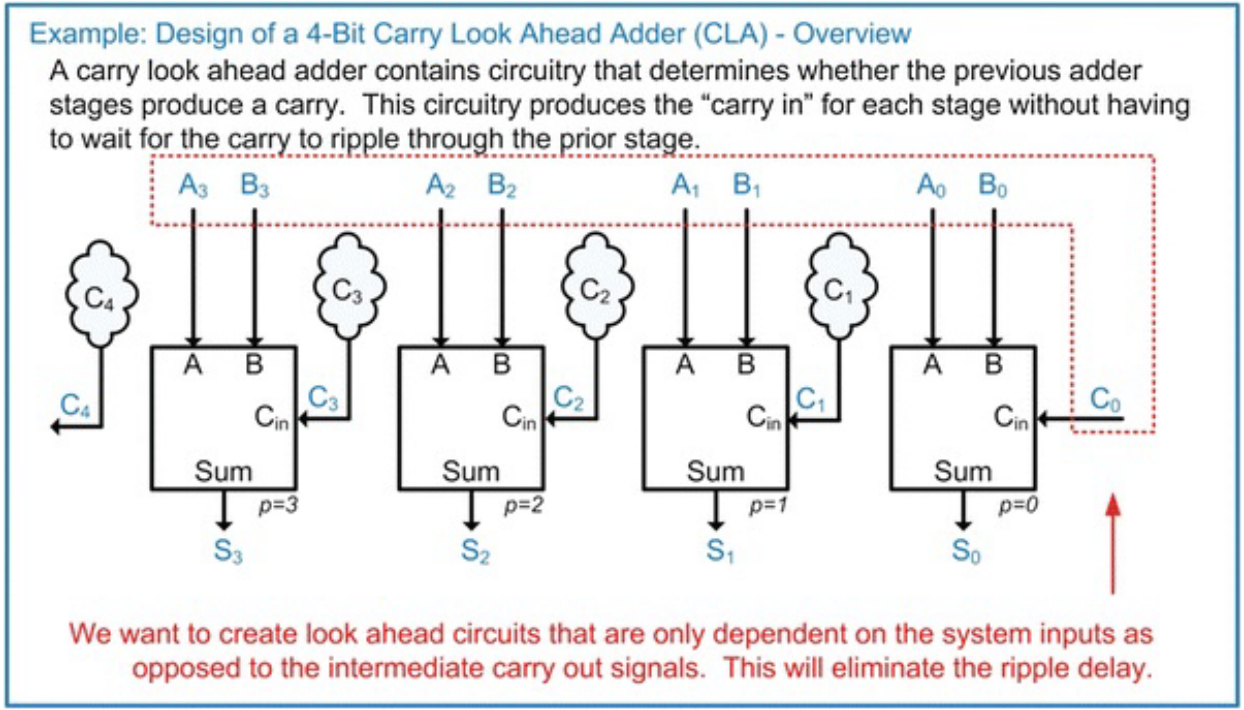
The number of logic levels for this adder is given as:

$$\text{Levels of Logic} = 3 + 2 \cdot (n-1)$$

Carry Look Ahead (CLA) сабирач

- Да би се решило потенцијално кашњење RCA сабирача направљен је сабирач са праћењем преноса унапред (CLA). Код овог приступа, додатна кола су укључена који производе међу пренос одмах, уместо да га чекају да их креира претходна пуна фаза сабирача.
- Ово омогућава да сабирач који треба да се заврши у фиксном временском периоду уместо оног који се скалира са број битова у сабирачу. Пример приказује преглед приступа за пројектовање CLA сабирача.

Пример 4-битног CLA сабирача



Carry Look Ahead (CLA) сабирач

- Да би CLA архитектура била ефикасна, кола за гледање унапред морају бити зависна само од системских улаза A, B и C_{in} (тј. C0).
- Секундарна карактеристика за CLA је да треба да се искористи што је више могуће поновне употребе пројектовања.
- Да би се испитали аспекти поновног коришћења пројектовања вишебитног сабирача, користе се концепти генерисања преноса (g) и пропагације (p).
- За потпуни сабирач се каже да генерише пренос ако његови улази A и B резултирају са C_{out} = 1 када је C_{in} = 0.
- За потпуни сабирач се каже да пропагира пренос ако његови улази A и B резултирају са C_{out} = 1 када је C_{in} = 1.
- Ови једноставни искази се могу користити за извођење логичких израза за сваки степен сабирача који може искористити предности постојећих логичких термова из претходне фазе.
- Пример показује извођење ових термова и како се алгебарске замене могу искористити за креирање кола унапред за сваки пуни сабирач који зависи само од системских улаза. У овим дериватима, променљива *i* се користи за представљање позиције пошто се *p* користи за термин пропагације.

Пример 4-битног CLA сабирача

Example: Design of a 4-Bit Carry Look Ahead Adder (CLA) – Algebraic Formation

The look ahead circuitry considers whether the prior adder stages create a carry by considering two conditions: 1) whether a stage will **generate** (*g*) a carry; and 2) whether the stage will **propagate** (*p*) a carry. Let's look at the truth table for a full adder.

C_{in}	A	B	C_{out}	
0	0	0	0	For the input codes where C_{in}=0, the full adder "generates" a new carry when A=1 and B=1. This behavior can be described with the expression: $g = A \cdot B$
0	0	1	0	
0	1	0	0	
0	1	1	1	
1	0	0	0	For the input codes where C_{in}=1, the full adder "propagates" the incoming carry when either A=1 or B=1. This behavior can be described with the expression: $p = A+B$
1	0	1	1	
1	1	0	1	
1	1	1	1	

The entire expression for the carry out can be written as:

$$C_{out} = g + p \cdot C_{in}$$
$$C_{out} = A \cdot B + (A+B) \cdot C_{in}$$

Let's see how this can be used to our advantage in a multiple bit adder. Recall that for any arbitrary adder position, the generate, propagate, and carry out terms are:

$$g_i = A_i \cdot B_i$$
$$p_i = A_i + B_i$$
$$C_{i+1} = g_i + p_i \cdot C_i$$

Note: We'll use the subscript "i" to denote position since we're using "p" for *propagate*.

We can now write expressions for the subsequent carry terms as:

$$C_1 = g_0 + p_0 \cdot C_0$$

The C_1 expression only depends on the inputs A, B, and C_0.

$$C_2 = g_1 + p_1 \cdot C_1$$
$$C_2 = g_1 + p_1 \cdot (g_0 + p_0 \cdot C_0)$$
$$C_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot C_0$$

For C_2, we can plug in the expression for C_1 to create an expression that only depends on A, B, and C_0...

$$C_3 = g_2 + p_2 \cdot C_2$$
$$C_3 = g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_0 \cdot p_1 \cdot C_0)$$
$$C_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot C_0$$

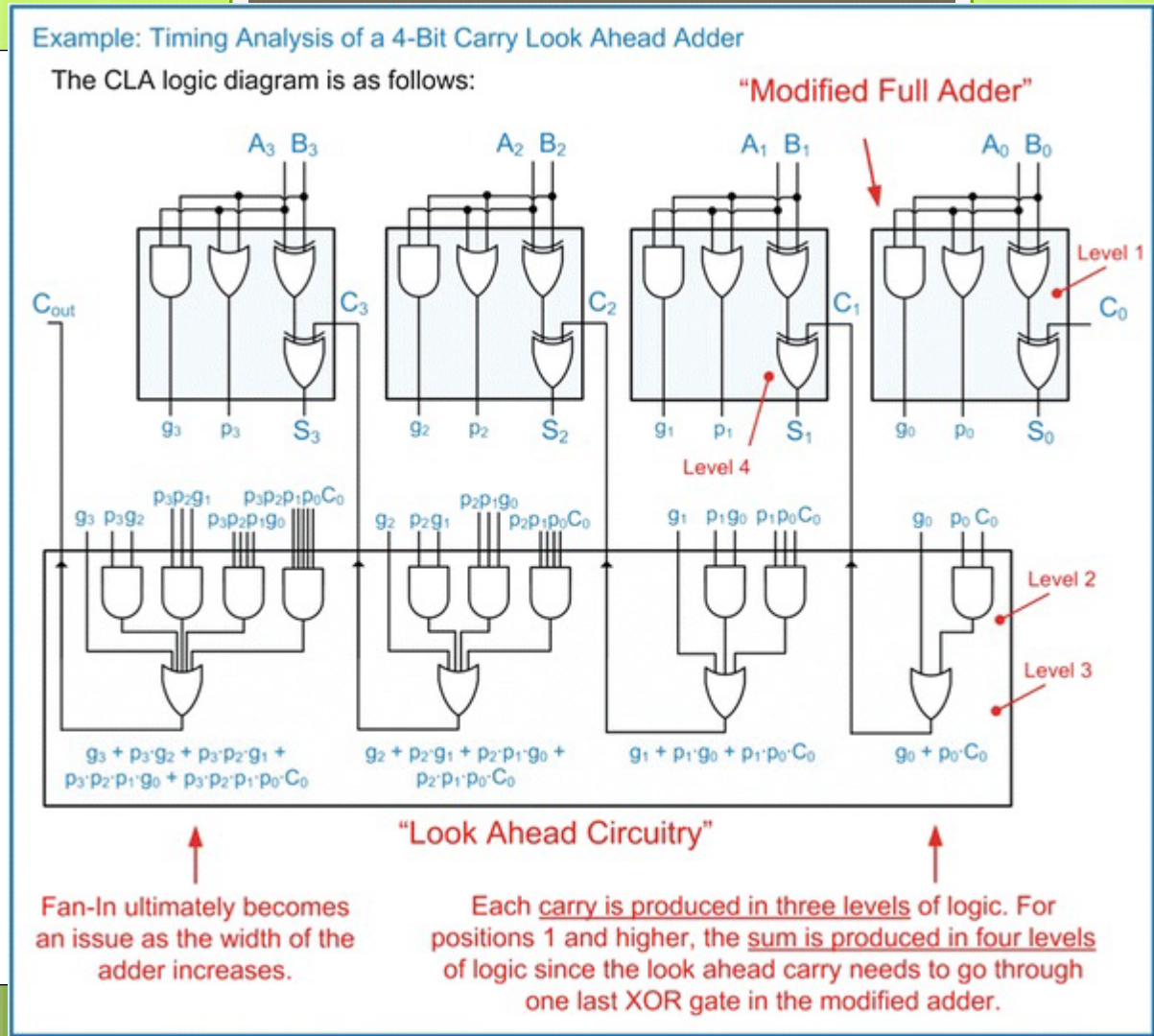
and again for C_3…

$$C_4 = g_3 + p_3 \cdot C_3$$
$$C_4 = g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot C_0)$$
$$C_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot C_0$$

and again for C_4…

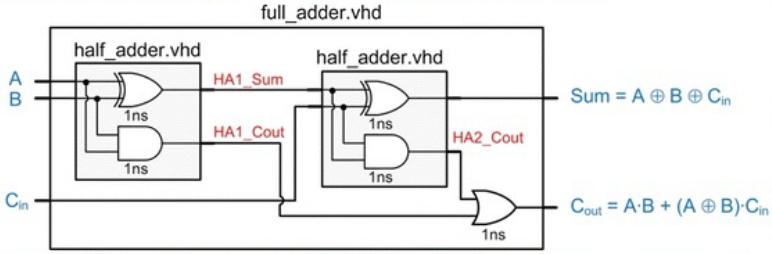
All of these expressions only depend on the inputs A, B, and C_0. Also notice that each expression is in a 2-level sum of products form.

Анализа кашњења 4-битног CLA саб.



Потпуни сабирач у VHDL-у

Example: Structural Model of a Full Adder in VHDL Using Two Half Adders

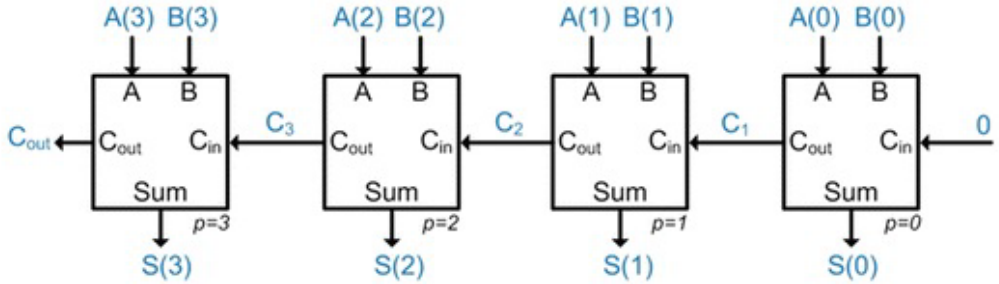



```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
  port (A, B      : in  std_logic;
        Sum, Cout : out std_logic);
end entity;

architecture half_adder_arch of half_adder is

 begin

   Sum  <= A xor B after 1 ns;
   Cout <= A and B after 1 ns;

end architecture;
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
  port (A, B, Cin  : in  std_logic;
        Sum, Cout  : out std_logic);
end entity;

architecture full_adder_arch of full_adder is

   component half_adder
     port (A, B      : in  std_logic;
           Sum, Cout : out std_logic);
   end component;

   signal HA1_Sum, HA1_Cout, HA2_Cout : std_logic;

 begin

   HA1 : half_adder port map (A, B, HA1_Sum, HA1_Cout);
   HA2 : half_adder port map (HA1_Sum, Cin, Sum, HA2_Cout);

   Cout <= HA1_Cout or HA2_Cout after 1 ns;

end architecture;
```

RCA сабирач у VHDL-у

Example: Structural Model of a 4-Bit Ripple Carry Adder in VHDL




```
library IEEE;
use IEEE.std_logic_1164.all;

entity rca_4bit is
  port (A, B  : in  std_logic_vector(3 downto 0);
        Sum   : out std_logic_vector(3 downto 0);
        Cout  : out std_logic);
end entity;

architecture rca_4bit_arch of rca_4bit is

   component full_adder
     port (A, B, Cin  : in  std_logic;
           Sum, Cout  : out std_logic);
   end component;

   signal C1, C2, C3 : std_logic;

 begin

   A0 : full_adder port map (A(0), B(0), '0', Sum(0), C1);
   A1 : full_adder port map (A(1), B(1), C1,  Sum(1), C2);
   A2 : full_adder port map (A(2), B(2), C2,  Sum(2), C3);
   A3 : full_adder port map (A(3), B(3), C3,  Sum(3), Cout);

end architecture;
```

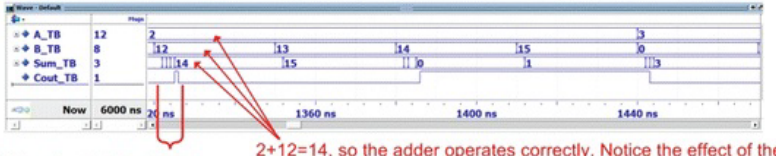
RCA сабирач у VHDL-у TB

Example: VHDL Test Bench for a 4-Bit Ripple Carry Adder Using Nested For Loops

Nested for loops can be used in order to generate an exhaustive set of test vectors to stimulate the adder.

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

entity rca_4bit_TB is
end entity;

architecture rca_4bit_TB_arch of rca_4bit_TB is

   component rca_4bit
     port (A, B   : in  std_logic_vector(3 downto 0);
           Sum    : out std_logic_vector(3 downto 0);
           Cout   : out std_logic);
   end component;

   signal A_TB, B_TB, Sum_TB  : std_logic_vector(3 downto 0);
   signal Cout_TB             : std_logic;


 begin

   DUT : rca_4bit port map (A_TB, B_TB, Sum_TB, Cout_TB);

   STIM : process
     begin

       for i in 0 to 15 loop
          for j in 0 to 15 loop
            A_TB <= std_logic_vector(to_unsigned(i,4));
            B_TB <= std_logic_vector(to_unsigned(j,4));
            wait for 30 ns;
          end loop;
       end loop;

   end process;

end architecture;
```

The simulation waveform for the ripple carry adder is as follows. The numbers are shown in unsigned decimal format for readability.



Glitches due to ripple delay.

2+12=14, so the adder operates correctly. Notice the effect of the ripple through the circuit. In addition to the correct output being delayed, there are glitches on both the Sum and C_{out} ports.

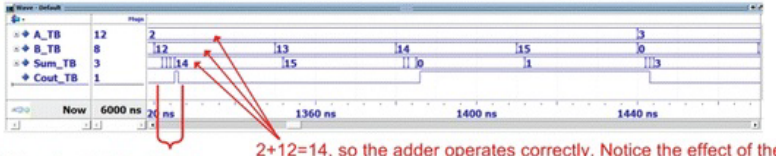
RCA сабирач у VHDL-у TB

Example: VHDL Test Bench for a 4-Bit Ripple Carry Adder Using Nested For Loops

Nested for loops can be used in order to generate an exhaustive set of test vectors to stimulate the adder.

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

entity rca_4bit_TB is
end entity;

architecture rca_4bit_TB_arch of rca_4bit_TB is

   component rca_4bit
     port (A, B   : in  std_logic_vector(3 downto 0);
           Sum    : out std_logic_vector(3 downto 0);
           Cout   : out std_logic);
   end component;

   signal A_TB, B_TB, Sum_TB  : std_logic_vector(3 downto 0);
   signal Cout_TB             : std_logic;


 begin

   DUT : rca_4bit port map (A_TB, B_TB, Sum_TB, Cout_TB);

   STIM : process
     begin

       for i in 0 to 15 loop
          for j in 0 to 15 loop
            A_TB <= std_logic_vector(to_unsigned(i,4));
            B_TB <= std_logic_vector(to_unsigned(j,4));
            wait for 30 ns;
          end loop;
       end loop;

   end process;

end architecture;
```

The simulation waveform for the ripple carry adder is as follows. The numbers are shown in unsigned decimal format for readability.



Glitches due to ripple delay.

2+12=14, so the adder operates correctly. Notice the effect of the ripple through the circuit. In addition to the correct output being delayed, there are glitches on both the Sum and C_{out} ports.

CLA сабирач у VHDL-у

Example: Structural Model of a 4-Bit Carry Look Ahead Adder in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mod_full_adder is
  port (A, B, Cin  : in std_logic;
        Sum, p, g  : out std_logic);
end entity;

architecture mod_full_adder_arch of mod_full_adder is

   constant tgate : time := 1 ns;

 begin

   Sum   <= (A xor B xor Cin) after 2*tgate;
   p     <= (A or B)          after 1*tgate;
   g     <= (A and B)         after 1*tgate;

end architecture;
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity cla_4bit is
  port (A, B  : in std_logic_vector(3 downto 0);
        Sum   : out std_logic_vector(3 downto 0);
        Cout  : out std_logic);
end entity;

architecture cla_4bit_arch of cla_4bit is

   constant tgate : time := 1 ns;

   component mod_full_adder
     port (A, B, Cin  : in  std_logic;
           Sum, p, g  : out std_logic);
   end component;

   signal C0, C1, C2, C3 : std_logic;
   signal p, g           : std_logic_vector(3 downto 0);

 begin

   C0   <= '0';
   C1   <= g(0) or (p(0) and C0) after 2*tgate;
   C2   <= g(1) or (p(1) and C1) after 2*tgate;
   C3   <= g(2) or (p(2) and C2) after 2*tgate;
   Cout <= g(3) or (p(3) and C3) after 2*tgate;

   A0 : mod_full_adder port map (A(0), B(0), C0, Sum(0), p(0), g(0));
   A1 : mod_full_adder port map (A(1), B(1), C1, Sum(1), p(1), g(1));
   A2 : mod_full_adder port map (A(2), B(2), C2, Sum(2), p(2), g(2));
   A3 : mod_full_adder port map (A(3), B(3), C3, Sum(3), p(3), g(3));

end architecture;
```

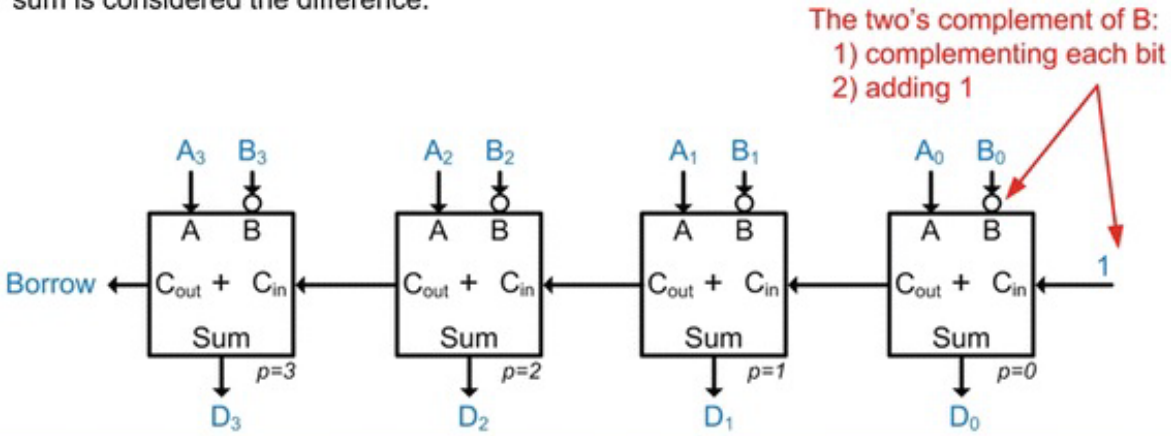
Пример 4-битног одузимача

Example: Design of a 4-Bit Subtractor Using Full Adders

A subtractor can be made from an adder by taking advantage of two's complement representation. When we wish to perform a subtraction we simply take the two's complement of the subtrahend (e.g., complement all bits and add 1) and then add the two numbers.

	A	← Minuend			A
-	B	← Subtrahend	=	+	(-B)
Difference				Difference	

Adders can be converted into subtractors by inverting the input B and adding 1. Since the adder is already setup to accommodate a carry in on position 0 (e.g., C_0), we can simply set C_0=1 to accomplish the "add 1" step. All carries are now considered borrows and the sum is considered the difference.



XOR инвертор

Example: Creating a Programmable Inverter Using an XOR Gate

An XOR gate can be used as a programmable inverter. Notice that when input A=0, the output F is equal to B. Also notice that when input A=1, the output is the inversion of B. This means we can selectively pass or invert the input B using A as the control signal.

	A	B	F
When A=0, F=B. This is simply a buffer.	0	0	0
	0	1	1
When A=1, F=B'. This is an inverter.	1	0	1
	1	1	0

Пример 4-битног сабирача/одузимача

Example: Design of a 4-Bit Programmable Adder/Subtractor

The control signal "ADDn/SUB" is used to select whether the circuit performs addition (ADDn/SUB=0) or subtraction (ADDn/SUB=1). When in subtraction mode, the XOR gates invert the subtrahend B and add 1 to the first adder stage. These steps take the two's complement of B and allow an add operation to conduct subtraction.