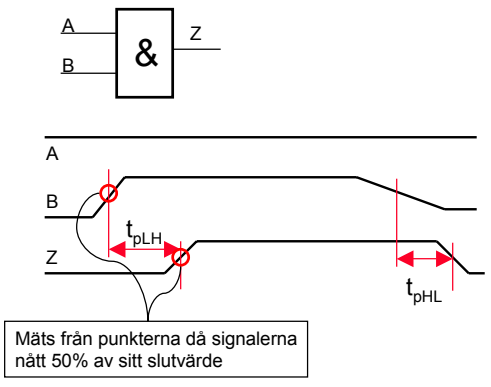
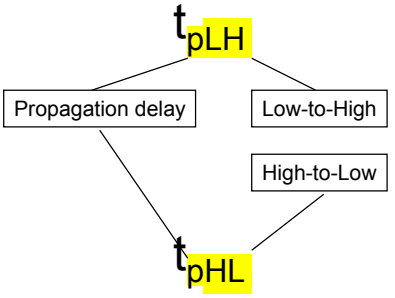
Konstruktion av kombinatorisk logik

- Innehåll
  - Fördröjningar i kombinatorisk logik
  - Hasard
  - Kombinatoriska byggblock
    - Multiplexer / Demultiplexer
    - Kodare / Avkodare
    - Aritmetiska funktioner

Grindfördröjning

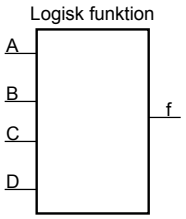
- Grindfördröjning är den tid det tar för en förändring på ingången ger en förändring på utgången.

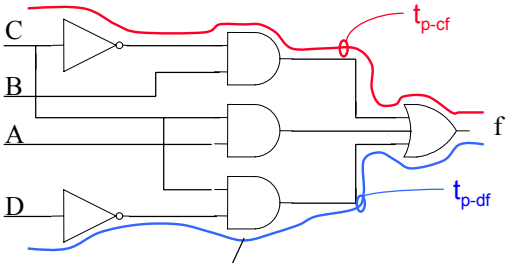




Fördröjningar i kombinatorisk logik

- Bestäm fördröjningen i ett kombinatoriskt nät





Fördröjningar för Samtliga signalvägar:
- t_{pHL-af} , t_{pLH-af}
- t_{pHL-bf} , t_{pLH-bf}
- t_{pHL-cf} , t_{pLH-cf}
- t_{pHL-df} , t_{pLH-df}

Ta fram den längsta fördröjningen i grindnätet. (t_{p-cf} eller t_{p-df}):

T_{p-cf} = 0.21 + 0.35 + 0.47 ns
= 1.03 ns

Grindfördröjningar från datablad

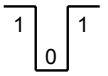
Grind	Fördröjning
Inv	tpLH = 0.21 tpHL = 0.19
and2	tpLH = 0.32 tpHL = 0.35
or3	tpLH = 0.47 tpHL = 0.43

Hasard i kombinatorisk logik

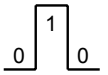
- Hasard (*glitchar*):
  - Oönskade omslag på utgången av ett komb. nät
    - Uppstår då olika signalvägar har olika fördröjningar
    - Kan vara ”farligt” om de oönskade omslagen startar andra händelser
      - Man kanske måste garantera att de aldrig uppkommer
- Vanliga lösningar
  - Vänta tills signalerna är stabila innan de används
  - Konstruera ”hasard-fria” kretsar

Olika typer av hasard

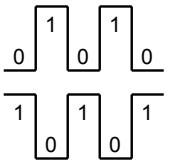
- Statisk-1 hasard
  - En förändring på ingången gör att utgången går från 1 till 0 till 1.



- Statisk-0 hasard
  - En förändring på ingången gör att utgången går från 0 till 1 till 0.

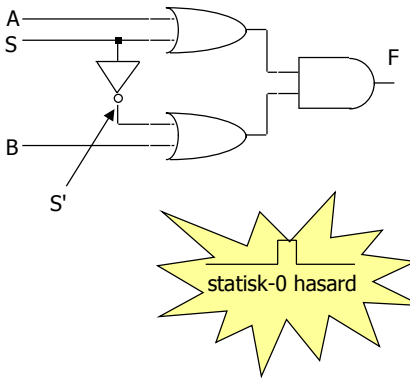


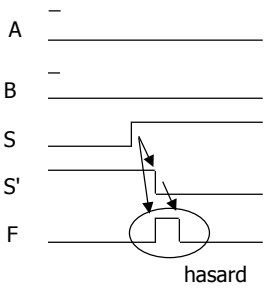
- Dynamisk hasard
  - En förändring på ingången gör att utgången går från 0 till 1 till 0 till 1 eller från 1 till 0 till 1 till 0.



Statisk hasard

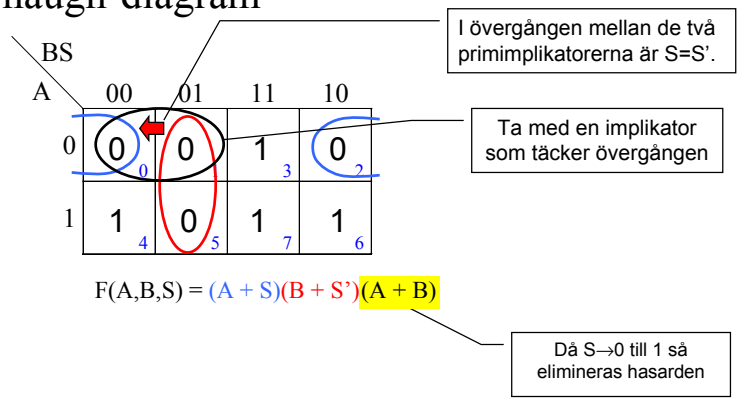
- Beror på att en variabel och dess komplement tillfälligt har samma värde
- Exempel:



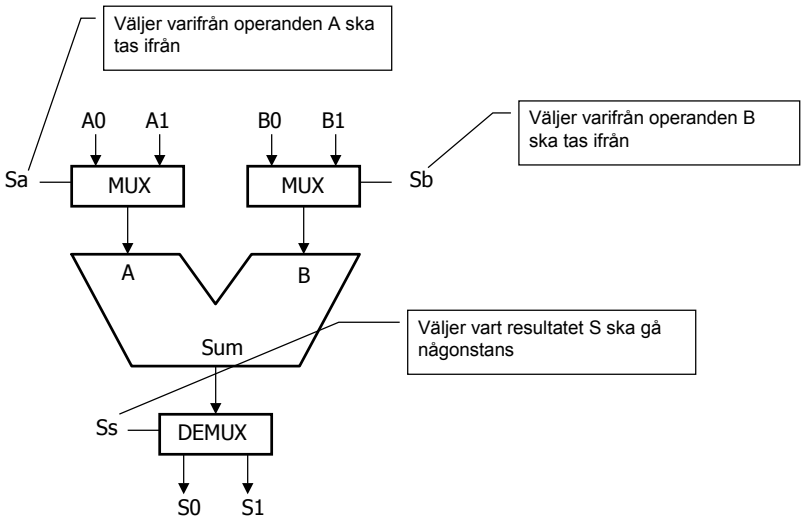


Eliminering av statisk hasard

- F = (A + S)(B + S')
- Karnaugh-diagram



Multiplexer/demultiplexer

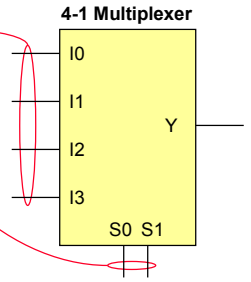


4-till-1 multiplexer i VHDL

```
architecture rtl of mux4_1 is
begin  -- rtl
  process (I0, I1, I2, I3, S0, S1)
  variable sel: bit_vector(1 downto 0);
  begin  -- process comb
    sel := S1 & S0;
    case sel is
      when "00" => Y <= I0;
      when "01" => Y <= I1;
      when "10" => Y <= I2;
      when others => Y <= I3;
    end case;
  end process;
end rtl;
```

Alla ingångar i sensitivitetslistan

Slå ihop de enskilda bitarna till en *bitvektor* så att det blir enkelt att hantera i *case*-satsen



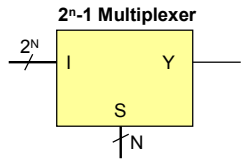
Generell 2^N-till-1 multiplexer

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity muxN_1 is
  generic (
    N : integer := 3); -- log2(antal ingångar)
  port (
    I: in  std_logic_vector((2**N - 1) downto 0);
    S: in  std_logic_vector(N-1 downto 0);
    Y : out std_logic);
end muxN_1;

architecture rtl of muxN_1 is

begin  -- rtl
  process (I, S)
    variable v_S : integer;

  begin
    v_S := conv_integer(S);
    Y <= I(2**v_S);
  end process;
end rtl;
```

Medger parameteriserbara komponenter

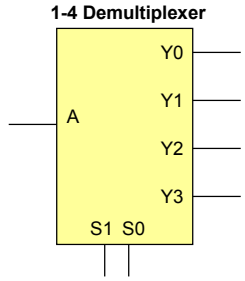
Konverterar signalen S, som är en bitvektor till integer (v_S)

Använd v_S för att beräkna vilken av ingångarna som ska tilldelas utgången Y



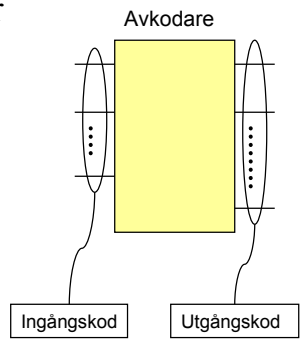
Demultiplexer

```
architecture rtl of demux1_4 is

begin

  process (A, S0, S1)

  begin
    case S1&S0 is
      when "00" => Y0 <= A;   Y1 <= '0'; Y2 <= '0'; Y3 <= '0';
      when "01" => Y0 <= '0'; Y1 <= A;   Y2 <= '0'; Y3 <= '0';
      when "10" => Y0 <= '0'; Y1 <= '0'; Y2 <= A;   Y3 <= '0';
      when others =>
                   Y0 <= '0'; Y1 <= '0'; Y2 <= '0'; Y3 <= A;
    end case;
  end process;
end rtl;
```

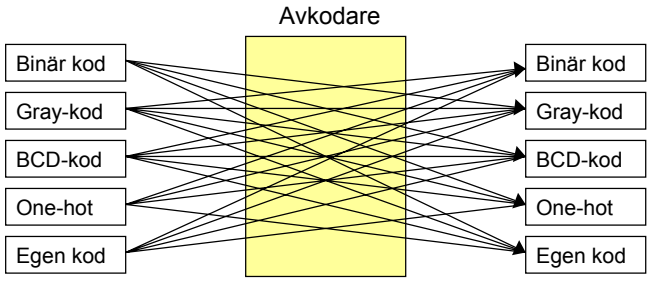


Avkodare – generellt

- Avkodare (eng. decoder)
  - Kombinatorisk krets med en eller flera ingångar samt flera utgångar
  - Antalen ingångar är färre än antalet utgångar
  - Konverterar en ingångskod till en utgångskod
  - En-till-en mappning
    - För varje ingångskod finns endast en utgångskod

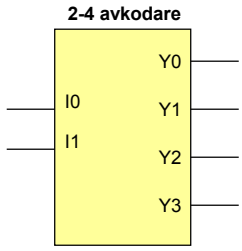


Avkodare – generellt



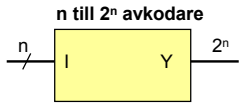
2-4 binär avkodare i VHDL

```
architecture rtl of encoder_2_4 is

begin  -- rtl

  process (I0, I1)

  variable I10 : bit_vector(1 downto 0);
  begin  -- process
    I10 := I1 & I0;
    case I10 is
      when "00" => Y3 <= '0'; Y2 <= '0'; Y1 <= '0'; Y0 <= '1';
      when "01" => Y3 <= '0'; Y2 <= '0'; Y1 <= '1'; Y0 <= '0';
      when "10" => Y3 <= '0'; Y2 <= '1'; Y1 <= '0'; Y0 <= '0';
      when "11" => Y3 <= '1'; Y2 <= '0'; Y1 <= '0'; Y0 <= '0';
      when others =>
                   Y3 <= '1'; Y2 <= '0'; Y1 <= '0'; Y0 <= '0';
    end case;

  end process;
end rtl;
```



N-till 2^N binär avkodare

```
entity encoder_n_m is
generic (
  n : integer := 3;  -- antal ingångar
  m : integer := 8); -- antal utgångar (=2**n)
  port (
    I: in  std_logic_vector(n-1 downto 0);
    Y: out std_logic_vector(m-1 downto 0)
  );
end encoder_n_m;


architecture rtl of encoder_n_m is
begin
  process(I)
    variable v_I: integer; -- integer värden för ingång
  begin
    v_I := conv_integer(I);
    Y <= conv_std_logic_vector((2**v_I),m);
 end process;
end architecture;
```

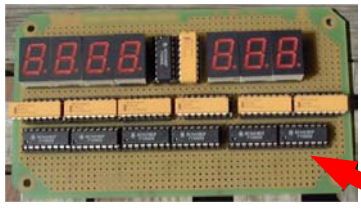


Konverterar signalen *I* till en *integer*

Konverterar det avkodade värdet från *integer*
Till *bitvektor* som är *m* bitar

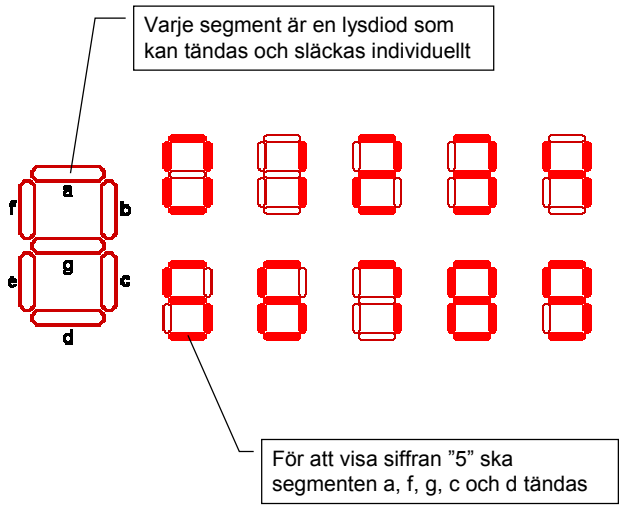
Exempel på användning av avkodare

- 7-segments display
  - LED (Light Emitting Diod)



  - LCD (Liquid Crystal Display)

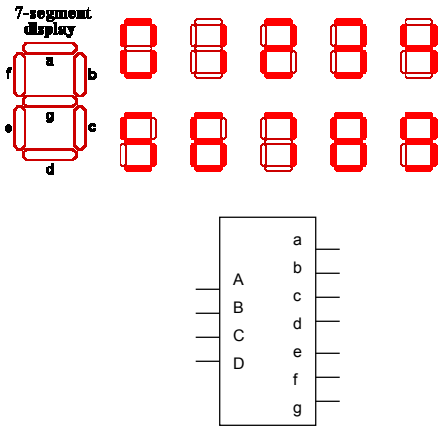
7-segment display



Avkodare för 7-segm. display

- Ingångskod: BCD (tal mellan 0 och 9)
- Utgångskod: 7-segmentskod

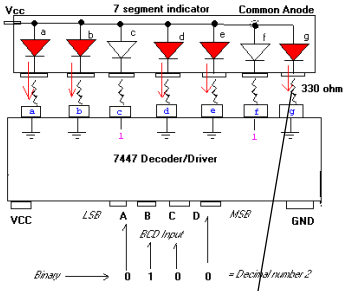
Tal	D C B A	a b c d e f g
0	0 0 0 0	1 1 1 1 1 1 0
1	0 0 0 1	0 1 1 0 0 0 0
2	0 0 1 0	1 1 0 1 1 0 1
3	0 0 1 1	1 1 1 1 0 0 1
4	0 1 0 0	0 1 1 0 0 1 1
5	0 1 0 1	1 0 1 1 0 1 1
6	0 1 1 0	0 0 1 1 1 1 1
7	0 1 1 1	1 1 1 0 0 0 0
8	1 0 0 0	1 1 1 1 1 1 1
9	1 0 0 1	1 1 1 0 0 1 1

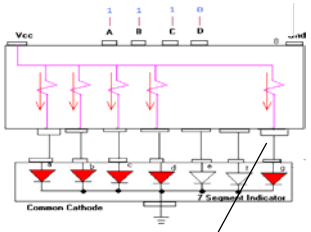


7-segment display

Common anod

Common cathode





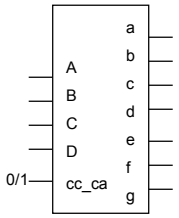
Utgångarna aktivt låga

Utgångarna aktivt höga

7-segment avkodare i VHDL

- Generell avkodare
  - Klarar av både ’Common cathode’ och ’Common anode’ displayer

```
entity ss_decoder is
  generic (
    cc_ca : bit := '1');
  port (
    Ai, Bi, Ci, Di: in  bit;
    a, b, c, d, e, f, g: out bit
  );
  end ss_decoder;
```



Väljer om avkodaren ska används till ’common cathode’ eller ’common anode’ display

Forts. 7-segment avkodare i VHDL

```
architecture rtl of ss_decoder is
  begin  -- rtl
      process (Ai, Bi, Ci, Di)
      variable BCD_string : bit_vector(3 downto 0);
      variable control_string : bit_vector(6 downto 0);
      begin  -- process
        BCD_string := Di & Ci & Bi & Ai;
        case BCD_string is
          when "0000" => control_string := "1111110";
          when "0001" => control_string := "0110000";
          when "0010" => control_string := "1101101";
          when "0011" => control_string := "1111001";
          when "0100" => control_string := "0110011";
          when "0101" => control_string := "1011011";
          when "0110" => control_string := "0011111";
          when "0111" => control_string := "1110000";
          when "1000" => control_string := "1111111";
          when "1001" => control_string := "1110011";
          when others =>
                         control_string := "1110011";
end case;
```

Skapa en bitsträng för ingångskoden

Forts. 7-segment avkodare i VHDL

```
        -- Common Cathode
        if cc_ca = '1' then
          a <= control_string(6);
          b <= control_string(5);
          c <= control_string(4);
          d <= control_string(3);
          e <= control_string(2);
          f <= control_string(1);
          g <= control_string(0);
        -- Common Anode
        else
          a <= not control_string(6);
          b <= not control_string(5);
          c <= not control_string(4);
          d <= not control_string(3);
          e <= not control_string(2);
          f <= not control_string(1);
          g <= not control_string(0);
        end if;
      end process;
  end rtl;
```

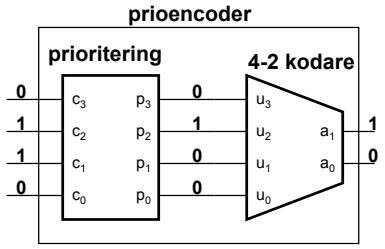
Signalerna är aktivt höga
För common cathode

Signalerna är aktivt låga
För common anode och
måste inverteras

Prioritetsavkodare

- Ingångarna har inbördes prioritet
- När mer än en ingång är aktiv så genereras en kod för den ingång med högst prioritet

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity prioencoder is
  port (
    C3, C2, C1, C0: in  std_logic;
    A1, A0 : out std_logic);
end prioencoder;
```

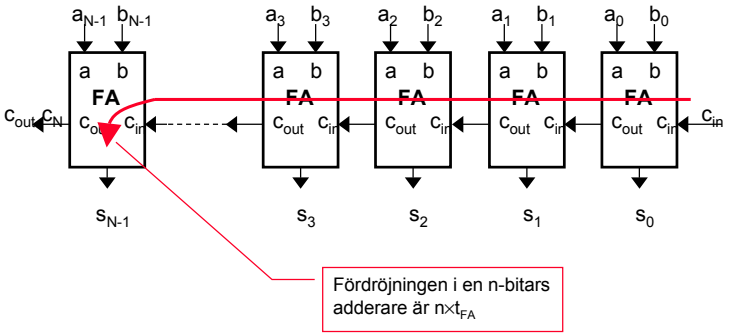


Forts. prioritetsavkodare

```
architecture rtl of prioencoder is
begin  -- rtl
  process (C3, C2, C1, C0)
  variable out_vector : std_logic_vector(1 downto 0);

  begin
    if C3= '1' then
      out_vector := "11";
    elsif C2 = '1' then
      out_vector := "10";
    elsif C1 = '1' then
      out_vector := "01";
    elsif C0= '1' then
      out_vector := "01";
   end if;
   A1 <= out_vector(1);
   A0 <= out_vector(0);
 end process;
end rtl;
```

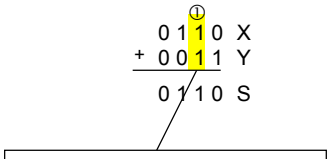
Adderare

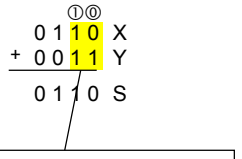
- Carry-Ripple Adderare
  - Enkel
  - Långsam



Carry-Lookahead-adderare

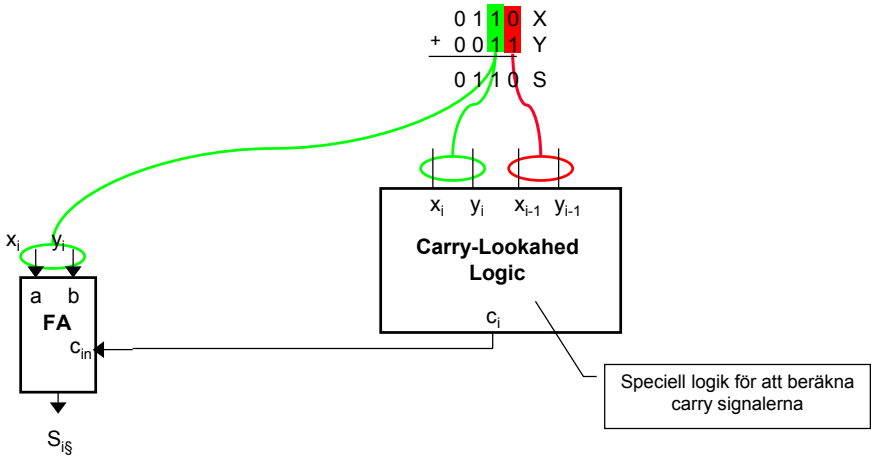
- Snabbare adderare
  - Snabbar upp beräkningen av carry-bitarna
  - Idé: Att samtidigt titta på flera positioner av det tal ska ska adderas
  - Exempel:





Carry-Lookahead logik

- Logik för att beräkna carry signaler



Carry-Lookahead logic

- Carry-Lookahead logiken är baserad på två funktioner
  - Carry generate:

    Addition i position *i* sägs generera en carry om den producerar en carry=1 oberoende av x_{i-1}, y_{i-1}, … x_0, y_0, c_0

    $$g_i = x_i \cdot y_i$$

  - Carry propagate:

    Addition i position *i* sägs propagera en carry om den producerar en carry=1 (c_{i+1})då x_{i-1}, y_{i-1}, … x_0, y_0, c_0 orsakar en carry-in (C_i=1)

    $$p_i = x_i + y_i$$

  - Ett additionssteg genererar en carry (c_{i+1}):

    Om ingångarna (x_i, y_i) antingen genererar en carry (g_i=1) eller att steget propagerar carry-in (p_i=1)

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

Exempel på *propagate* och *generate* funktionerna

```
  0 1 0 1 1 0   X
+ 0 0 1 1 0 1   Y
```

$0\,0\,0\,1\,0\,0 \quad g_i = x_i \cdot y_i$

$0\,1\,1\,1\,1\,1 \quad p_i = x_i + y_i$

$0\,1\,1\,1\,0\,0 \quad c_{i+1} = g_i + p_i \cdot c_i$

```
  0 1 0 1 1 0   X
  0 0 1 1 0 1   Y
+ 0 1 1 1 0 0   C
  1 0 0 0 1 1   S
```

Addition utan carry-propagering

Utveckla uttryck för CLA-logik

- Utveckla uttrycken rekursivt
  - Ta fram uttryck för c_1, c_2, c_3 och c_4

$$c_1 = g_0 + p_0 \cdot c_0$$

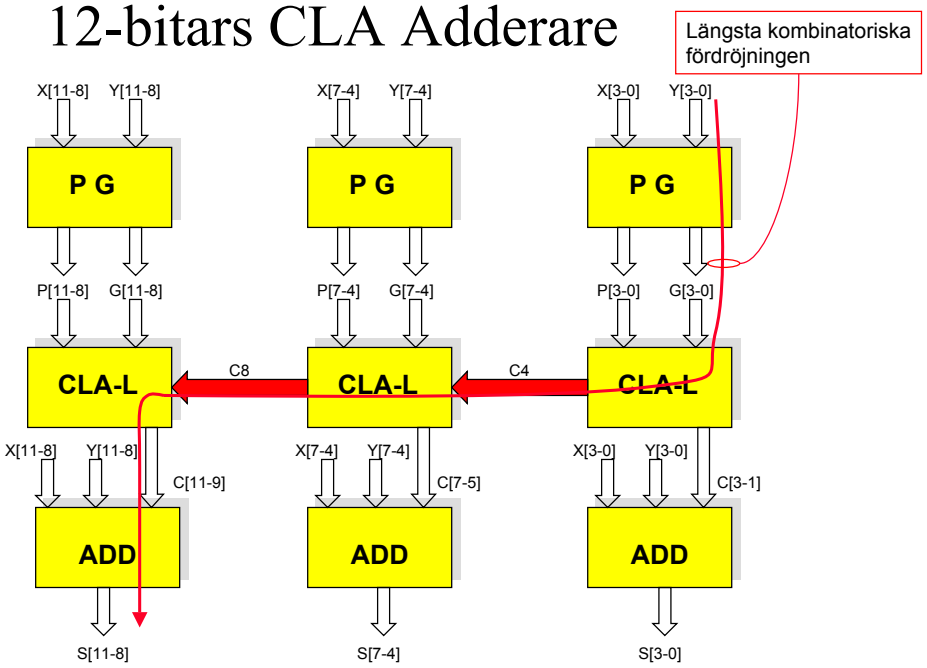
$$\begin{aligned} c_2 &= g_1 + p_1 \cdot c_1 \\ &= g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \\ &= g_1 + p_1 \cdot g_0 + p_1 p_0 \cdot c_0 \end{aligned}$$

c_1, c_2, c_3 och c_4 är funktioner av g- och p

$$\begin{aligned} c_3 &= g_2 + p_2 \cdot c_2 \\ &= g_2 + p_2(g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) \\ &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \end{aligned}$$

$$\begin{aligned} c_4 &= g_3 + p_3 \cdot c_3 \\ &= g_3 + p_3(g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\ &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0) \end{aligned}$$

12-bitars CLA Adderare



Adderare i VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity add is
  generic (
    N : integer := 32);

  port (
    X, Y : in  std_logic_vector(N-1 downto 0);
    S    : out std_logic_vector(N-1 downto 0);
    Cout : out std_logic);
end add;

architecture rtl of add is

begin  -- rtl
  add_sub : process (X, Y)
  variable sum : std_logic_vector(N downto 0);

  begin  -- process add_sub
    sum := ('0' & X) + ('0' & Y);
    S <= sum(N-1 downto 0);
    Cout <= sum(N);
  end process add;

end rtl;
```

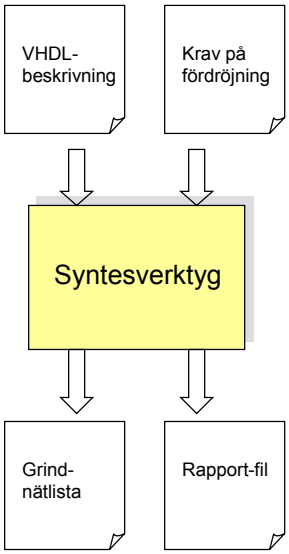
Ska vara med för att **+** ska fungera för vektorer av *std_logic*

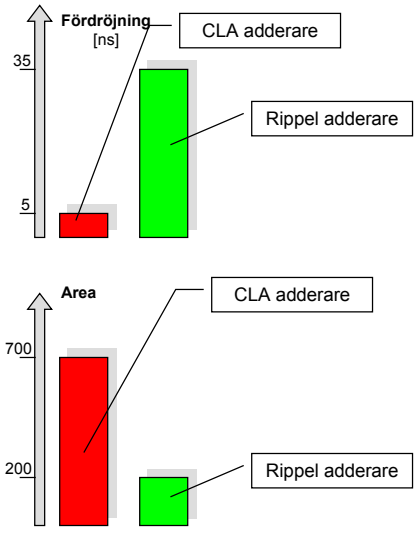
'0'&X är med för att + ska returnera en summa på N+1 bitar

Summan ut (S) är de N minst signifikanta bitarna

Carry-out är den mest signifikanta biten

Exempel på syntes av adderare





Multiplikation i VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multiplikator is
  generic (
    N : integer := 32);

  port (
    X, Y : in  std_logic_vector(N-1 downto 0);
    P    : out std_logic_vector(2*N-1 downto 0);
end multiplikator;

architecture rtl of multiplikator is

begin  -- rtl
  process (X, Y)
  begin  -- process
    P <= X*Y;
  end process;
end rtl;
```

Ska vara med för att * ska fungera för vektorer av *std_logic*

Antal bitar i produkten är summan av antalet bitar i multiplikator och multiplikand

SLUT på Föreläsning 4

- Innehåll
  - Fördröjningar i kombinatorisk logik
  - Hasard
  - Kombinatoriska byggblock
    - Multiplexer / Demultiplexer
    - Kodare / Avkodare
    - Aritmetiska funktioner