# F6: Konstruktion av kombinatorisk logik

- **Innehåll:**
  - **Dokumentationsstandard för digitala system**
  - **Timing och fördröjningar i digitala kretsar**
  - **Konstruktion av digitala kretsar**
    - **avkodare (*decoder*)**
    - **kodare (*encoder*)**
    - **multiplexer (*multiplexer*)**





# Dokumentationsstandarder

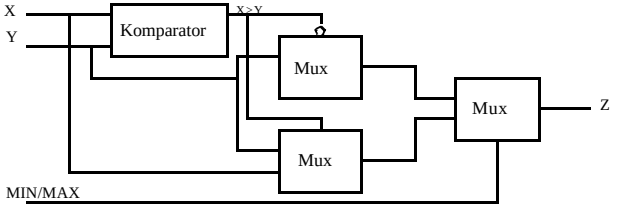
- **Dokumentationen för ett digitalt system ska ge tillräcklig information för att man ska kunna:**
  - **bygga det**
  - **testa det**
  - **använda det**
  - **underhålla det**
- **Generellt sett så innehåller dokumentationen**
  - **Ett blockdiagram som innehåller ingångar, utgångar, de stora byggblocken (modulerna) i systemet och hur de är sammankopplade**
  - **En schemaritning som visar samtliga komponenter, deras typ samt hur de är sammankopplade**
  - **Ett timingdiagram som visar de logiska signalerna som funktion av tiden**
  - **En logisk beskrivning av olika delar i konstruktionen (logiska ekvationer, tillståndsdiagram m.m)**
  - **En beskrivning hur man använder konstruktionen**





# Blockdiagram

- **Ett blockdiagram ska visa:**
  - **alla in- och utgångar**
  - **de ingående byggblocken**
  - **hur blocken logiskt är sammankopplade**
- **Interna detaljer i blocken ska inte visas**
- **Signaler som hör samman kan kombineras till en logisk signal som ritas som en tjockare linje, detta kallas för buss-signal**
- **Exempel på blockdiagram:**



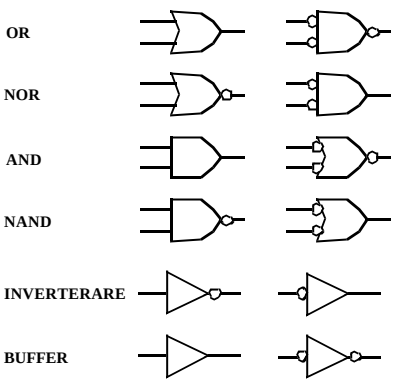





# Schemaritning

- **Symboler för grindar**
- **Signalnamn och aktiva nivåer**
- **bubbla - till - bubbla**
- **Layout**

## Symboler för grindar




- **Övning: visa att de ekvivalenta grindarna har samma funktion.**





## Övning

- **OR :** A + B = ( (A + B)' )' = (A'· B')'
- **NOR:** (A + B)' = A'· B'
- **AND:** A · B = ( (A · B)' )' = (²A'+ B')'
- **NAND:** (A · B)' = A' + B'





## Signalnamn och aktiva nivåer

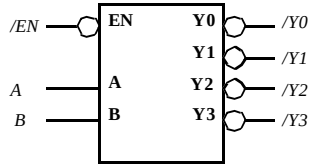
- **In- och utsignaler ska märkas med variabelnamn (X, Y, A, ...), eller med namn som reflekterar händelser eller villkor (ENABLE, REQUEST, /READY, ERROR, PAUSE ...).**
- **Aktiva nivåer: aktiv hög eller aktiv låg.**
- **Signalen blir *sann* då den antar sin aktiva nivå och *falsk* då den inte har sin aktiva nivå.**
- **En signal som är aktiv låg har ofta prefixet / som en del av variabelnamnet**
- **Exempel:**
  - **ERROR är aktiv hög och den har innebörden att då signalen är hög indikerar den att ett fel har uppstått.**
  - **/READY är aktiv låg och indikerar att data finns tillgängligt då signalen är låg.**





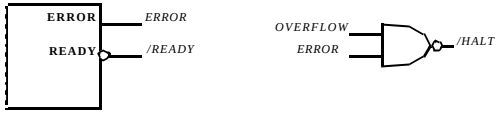
## Aktiva nivåer för pinnar

- **I logiska grindar och logiska strukturer används inverterings "bubblan" för att indikera aktiv nivå för signalen.**
- **Exempel: 2-4 Dekoder**
  - **/EN är aktivt låga**
  - **A och B är aktivt höga**
  - **/Y0, /Y1, /Y2, /Y3 är aktivt låga.**



## Logisk konstruktion med "bubbla-till-bubbla"

- Regler
  - Den aktiva nivån för en utgångssignal från en logisk komponent ska stämma överens med den aktiva nivån för komponentens utgångspinne.



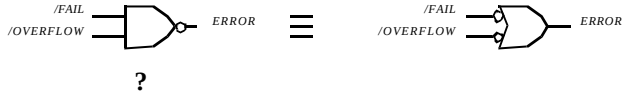

  - Den aktiva nivån för en ingånssignal till en logisk komponent ska stämma överens med den aktiva nivån för komponentens ingångspinne.






## "Bubbla-till-bubbla"

- Syfte: Underlätta att förstå den logiska kretsens funktion



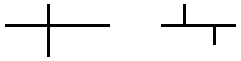

?





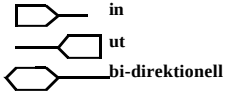
## Schemaritning

- Ingångar till vänster och utgångar till höger
- Signalflöden från vänster till höger



- Signalvägar ska vara anslutna
- Brutna signalvägar ska markeras med signalkälla eller destination


- Linjer som korsar varandra (ej sammankopplade) och kopplingar (av T-typ)
- Bussar ska namnges: DATA[7:0] , DATA
- Signaler som är tagna ur bussar ska vara namngivna: DATA[5]
- Riktning på in- och utsignaler anges med portsymbol
- Flera sidor i ett schema organiseras som:
  - Hierarkiskt
  - Platt struktur

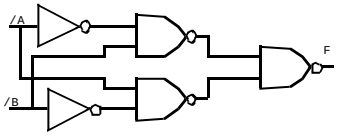




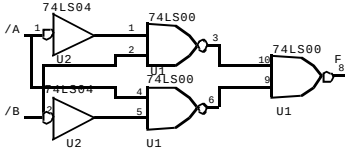



## Schematiska och logiska diagram

- Logik

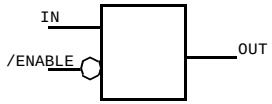


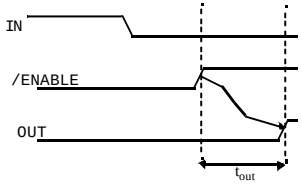

komponenter



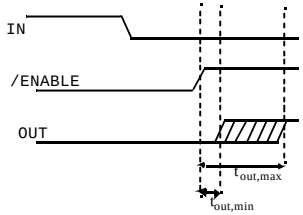

- "Bubbla-till-bubbla" logik
- Typ av IC och logikfamilj
- Pin nummer
- Referens till vilken av kretsarna som grinden ligger i (Unit-number U*)

## Timing diagram



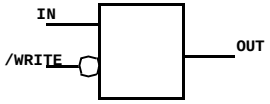

- **Fördröjningar beror på:**
  - **den interna strukturen i kretsen**
  - **typ av logikfamilj**
  - **matningsspänning**
  - **temperatur**

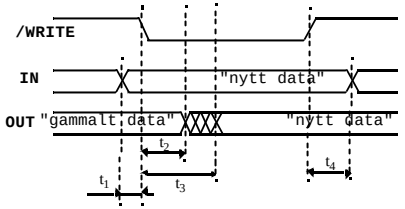







## Timing diagram för datasignaler i buss




- **$t_1$ : setup tid**
- **$t_2$ : min. fördröjning för utsignalen**
- **$t_3$: max. fördröjning för utsignalen**
- **$t_4$: hålltid för insignalen**





## Fördröjningar

- **Den tid det tar för en en förändring på ingången på en grind till det att utgången förändras kallas för grindfördröjning (gate propagation delay), $t_p$.**
- **$t_p$ för en signal beror på signalvägen inne i den logiska kretsen**
- **grindfördröjningen för en signal som går från låg till hög ($t_{pLH}$) och för en signal som går från hög till låg ($t_{pHL}$) är inte lika**
- **$t_p$ specificeras av tillverkarens datablad för IC:n**
- **Exempel:**

| | Typisk | | Max. | |
|---|---|---|---|---|
| | $t_{pHL}$ | $t_{pLH}$ | $t_{pHL}$ | $t_{pLH}$ |
| 74LS00 | 9 | 10 | 15 | 15 |
| 74HCT00 | 11 | 11 | 35 | 35 |
| 74ACT00 | 5.5 | 4.0 | 9.5 | 8.0 |

- **Den totala $t_p$ för en krets fås genom att addera alla grindfördröjningar längs signalvägen**



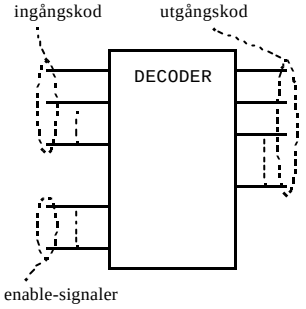


## Kombinatoriska strukturer

- **Avkodare generellt (eng. *decoder*)**
- **Binär avkodare**
- **Avkodare till sjusegment display**
- **Kodare generellt (eng. *encoder*)**
- **Prioritetskodare**
- **Tillämpning av kodare**

# Avkodare

- **Krets med flera ingångar och flera utgångar**
- **Antalet ingångar (n) är mindre än antalet utgångar (m)**
- **Konverterar en ingångkod till en utgångskod**
- **1 - till - 1 mappning**
  - **d.v.s för varje ingångskod finns det endast en utgångskod**
- **Exempel på ingångskoder**
  - **binär kod**
  - **Gray kod**
  - **BCD kod**
  - **vilken-som-helst kod**








# VHDL beskrivning av 2-till-4 avkodare

```
entity encoder_2_4 is

    port (
    I0, I1, EN     : in  bit;
    Y0, Y1, Y2, Y3 : out bit);

end encoder_2_4;

architecture rtl of encoder_2_4 is

begin  -- rtl

  p1 : process (I0, I1, EN)

  variable I10 : bit_vector(1 downto 0);    -- concatenation variable

  begin  -- process p1
  I10 := I1 & I0;
  if (EN = '1') then

  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 => null;
  end case;

  else
    Y3 <= '0'; Y2 <= '0'; Y1 <= '0'; Y0 <= '0';
  end if;
end process;
end rtl;
```





# n-till-m avkodare i VHDL

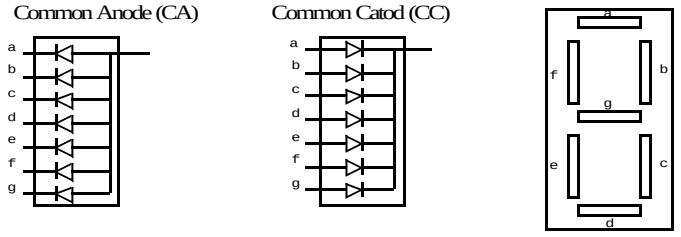
```
-- General binary n-to-m encoder
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.all;

entity encoder_n_m is
generic (
  n : integer := 3;                    -- number of inputs
  m : integer := 8);                   -- number of outputs
  port (
    I: in  std_logic_vector(n-1 downto 0);
    EN : in std_logic;
    Y: out std_logic_vector(m-1 downto 0)
  );
end  encoder_n_m;

architecture rtl of encoder_n_m is
begin
  process(I, EN)
    variable v_I, v_Y : integer;            -- integer values of inputs
  begin
    v_I := conv_integer(I);
    if EN = '1' then
      Y <= conv_std_logic_vector((2**v_I),m);
    else
      Y <= conv_std_logic_vector(0,m);
    end if;
  end process;
end architecture;
```





# Sju-segment Display

- **Visar decimala siffror och en del alfabetiska tecken**
- **LED (Light Emitting Diode) eller LCD (Liquid Crystal Display)**
- **LED-typ:**



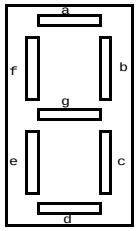

- **CA: kräver ingångar som är aktivt låga (en drivare med aktivt låga utgångar)**
- **CC: kräver utgångar som är aktivt höga (en drivare med aktivt höga utgångar)**

## Sju-segment avkodare/drivare

- **Ingångskod: BCD**
  **Utgångskod: sju-segment-kod**

- **Sanningstabell för en aktiv-hög avkodare**

| Ingångar | | | | Utgångar | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **D** | **C** | **B** | **A** | **a** | **b** | **c** | **d** | **e** | **f** | **g** |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

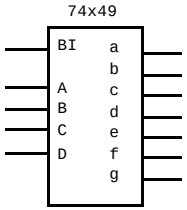




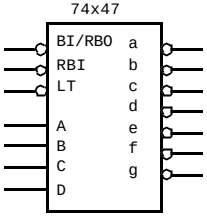



## MSI sju-segment avkodare

Common Cathod drivare




Common Anode drivare

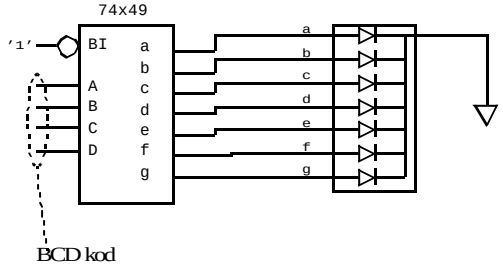



BI: Blanking Input
BI/RBO: BI: indertrycker tillfälliga förändringar då insignalen ändras
LT: Tänder alla segment





## 74x79 drivare








## Sju-segment avkodare i VHDL 1(2)

```
entity ss_decoder is

  generic (
    cc_ca : bit := '1');            -- Select Common Cathode / Common Anode
  port (
    Ai, Bi, Ci, Di          : in  bit;  -- Inputs in BDC-format
    a, b, c, d, e, f, g : out bit  -- control signals to seven-segment display
  );
  end ss_decoder;

  architecture rtl of ss_decoder is
  begin  -- rtl
      process (Ai, Bi, Ci, Di)
      variable BCD_string : bit_vector(3 downto 0); -- concatenation variable
      variable control_string : bit_vector(6 downto 0); -- concatenation variable
      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 => null;
        end case;
```

## Sju-segment avkodare i VHDL 2(2)

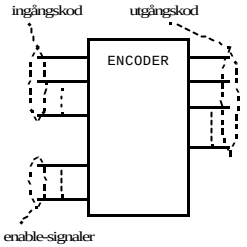
```
        -- Common Cathode
        if cc_ca = '1' then
          a <= control_string(6);
          b <= control_string(5);
          c <= control_string(4);
          d <= control_string(3);
          e <= control_string(2);
          f <= control_string(1);
          g <= control_string(0);
        -- 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;
```





## Kodare (Encoder)

- **Krets med många ingångar och många utgångar**
- **Utför den motsatta funktionen av en avkodare**
- **Antal utgångar (m) är färre än antalet ingångar (n)**
- **Konverterar ett ingående kodord till ett utgående kodord**



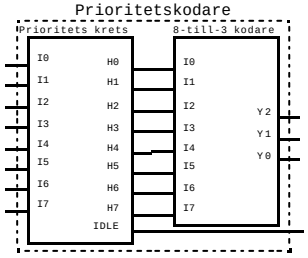





## Prioritetskodare (Priority encoder)

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

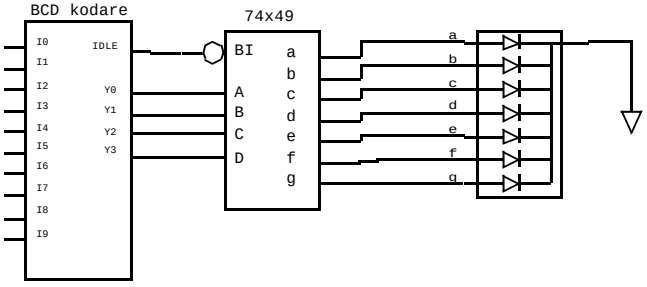
H7 = I7
H6=I6·H7´
H5=I5·I6´·H7´
H4=I4·I5´·I6´·H7´
H3=I3·I4´·I5´·I6´·H7´
H2=I2·I3´·I4´·I5´·I6´·H7´
H1=I1·I2´·I3´·I4´·I5´·I6´·H7´
H0=I0·I1´·I2´·I3´·I4´·I5´·I6´·H7´
IDLE=I0´·I1´·I2´·I3´·I4´·I5´·I6´·H7´








## Tillämpning av kodare för BCD



# Prioritetskodare i VHDL

```
-- The 74x148 Priority Encoder
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity prioencoder is
  port (
    NI0, NI1, NI2, NI3, NI4, NI5, NI6, NI7, NEI     : in  std_logic;
    NA0, NA1, NA2, NGS, NEO : out std_logic);
end prioencoder;

architecture rtl of prioencoder is
begin  -- rtl

  p1 : process (NI0, NI1, NI2, NI3, NI4, NI5, NI6, NI7, NEI)

  variable out_vector : std_logic_vector(4 downto 0);    -- concatenation variable

  begin  -- process p1
    if NEI = '1' then
      out_vector := "11111";
    elsif NI7 = '0' then
      out_vector := "00001";
    elsif NI6 = '0' then
      out_vector := "00101";
    elsif NI5 = '0' then
      out_vector := "01001";
    elsif NI4 = '0' then
      out_vector := "01101";
    elsif NI3 = '0' then
      out_vector := "10001";
    elsif NI2 = '0' then
      out_vector := "10101";
    elsif NI1 = '0' then
      out_vector := "11001";
    elsif NI0 = '0' then
      out_vector := "11101";
    else
     out_vector := "11110";
    end if;
```

```
    NEO <= out_vector(0); -- Enable output for cascading
    NGS <= out_vector(1); -- Device enabled and one or more
                          -- inputs active
    NA0 <= out_vector(2); -- encoded output bits
    NA1 <= out_vector(3);
    NA1 <= out_vector(4);
  end process;

end rtl;
```

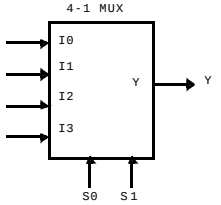




# Multiplexers

- **Multiplexing: överföra ett stort antal signaler över en litet antal kanaler eller ledningar**
- **Digital multiplexer (MUX): väljer en av många ingångssignaler och dirigerar den till en enda utgång**
- **kontrollsignaler väljer en specifik ingång**
- **n stycket kontrollsignaler kan välja en av $2^n$ insignaler**
- **Exempel: 4-till-1 multiplexer:**

| S1 | S0 | Y |
|---|---|---|
| 0 | 0 | I0 |
| 0 | 1 | I1 |
| 1 | 0 | I2 |
| 1 | 1 | I3 |








# 4-till-1 Multiplexer i VHDL

```
entity mux4_1 is
  port (
    I0, I1, I2, I3, S0, S1 : in  bit;
    Y : out bit);
end mux4_1;

architecture rtl of mux4_1 is

begin  -- rtl
  comb : 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 "11" => Y <= I3;
      when others => null;
    end case;
  end process comb;
end rtl;
```



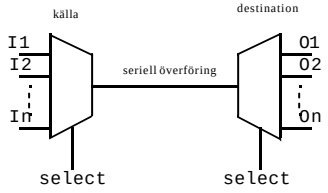


# N-till-1 Multiplexer i VHDL

```
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(number of inputs)
  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
  comb : process (I, S)
    variable v_S : integer;

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

# Demultiplexer

- **En demultiplexer (DMUX) har den motsatta funktionen till en multiplexer**
- **En digital demultiplexer tar emot indata från en enda ingång och dirigerar den till en-av-många utgångar enligt kontrollsignalens värde**
- **MUX/DMUX används i seriell dataöverföring**








# VHDL kod för en 1-till-4 demultiplexer

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

entity demux1_4 is

  port (
    I0, I1, I2, I3: out  std_logic;
    A : in std_logic;
    S : in std_logic_vector(1 downto 0));
end demux1_4;

architecture rtl of demux1_4 is

begin  -- rtl

  comb : process (A, S)

  begin  -- process comb
    case S is
      when "00" => I0 <= A;   I1 <= '0'; I2 <= '0'; I3 <= '0';
      when "01" => I0 <= '0'; I1 <= A;   I2 <= '0'; I3 <= '0';
      when "10" => I0 <= '0'; I1 <= '0'; I2 <= A;   I3 <= '0';
      when "11" => I0 <= '0'; I1 <= '0'; I2 <= '0'; I3 <= A;
      when others => null;
    end case;
  end process comb;
end rtl;
```



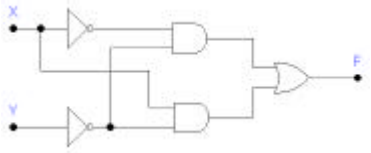


# Exklusiv ELLER (exclusive OR)

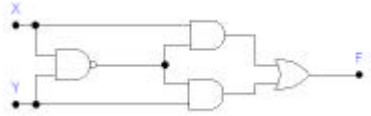
- **XOR :** $X \oplus Y = X' \cdot Y + X \cdot Y'$
- **XNOR :** $(X \oplus Y)' = X \cdot Y + X' \cdot Y'$

| X | Y | XOR | XNOR |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

XOR




XNOR








# Symboler för XNOR och XOR

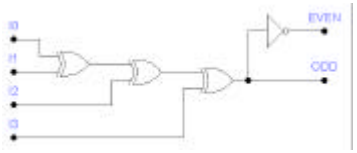
- **Ekvivalenta symboler för XOR grindar**

- **Ekvivalenta symboler för XNOR grindar**

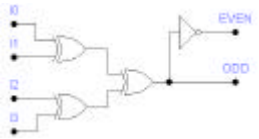
- **SSI XOR grind**
  - **74x86, 4 st. XOR grindar i en kapsel**

## Tillämpning av XOR grind: Paritetskontroll

- **Krets för udda paritet: Utgången blir 1 om ett udda antal ingångar är 1**
- **Krets för jämn paritet: Utgången blir 1 om ett jämt antal ingångar är 1**
- **Exempel: 4 bitars paritetskrets**




**Kaskadkopplad struktur**




**Trädstruktur**

**Exempel på indata:** `1101` **ger** `EVEN=0` **och** `ODD=1`





## VHDL kod för N-bitars paritetskrets

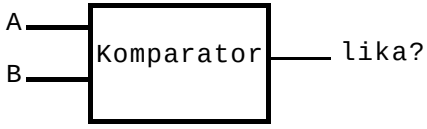
```
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.ALL;

entity parity_N is
     generic (
     N : integer := 4);                  -- number of inputs

  port (
     I          : in  std_logic_vector(N-1 downto 0);
     odd, even  : out std_logic);
end  parity_N;

architecture rtl of parity_N is
begin  -- rtl
  parity : process (I)
  variable odd_v : std_logic;
  variable j : integer;
  begin  -- process parity
    odd_v := '0';
    for j in I'range loop
      odd_v := odd_v xor I(j);
    end loop;  -- j

    odd <= odd_v;
    even <= not odd_v;

  end process parity;
end rtl;
```



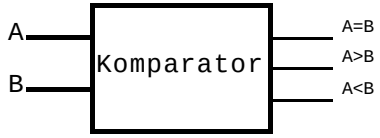


## Komparator

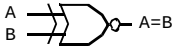
- **Jämför två binära ord och indikerar om de är lika**




- **Avancerad komparator**




- **1-bits komparator: XNOR grind**

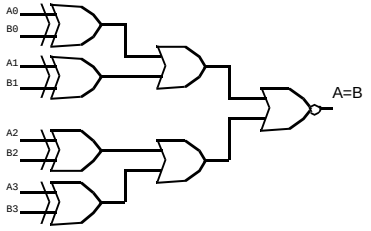







## Fler-bitars parallell komparator

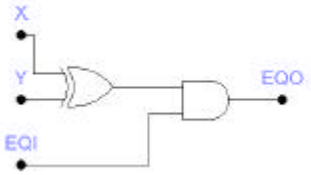
- **Parallell bitvis jämförelse**
- **Exempel: 4-bitars komparator**



# Iterativ Komparator

- **Jämför en bit av de två ingångarna**
- **Iterativ komparator är ett antal kaskadkopplade 1-bits komparatorer**
- **1-bits komparator, Funktionstabell:**

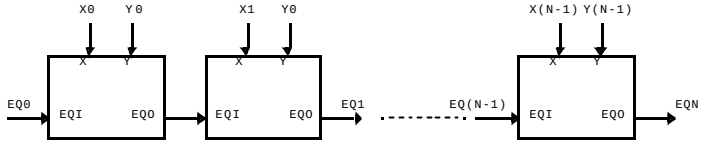
| EQI | X | Y | EQO |
|---|---|---|---|
| 0 | x | x | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |








# Fler-bitars iterativ komparator

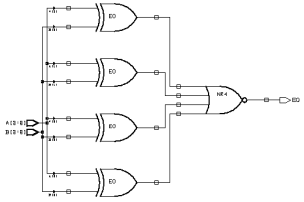







# N-bits komparator i VHDL

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

entity comp_N is

    generic (N : integer := 4);                  -- number of input bits
  port (
    A, B : in  std_logic_vector(N-1 downto 0);
    EQ   : out std_logic);
end comp_N;

architecture rtl of comp_N is
begin  -- rtl
  compare : process (A, B)
  begin  -- process compare
    if (A = B) then
      EQ <= '1';
    else
      EQ <= '0';
    end if;
  end process compare;

end rtl;
```



| | | |
|---|---|---|
| design: comp_N | designer: Bengt Oelmann | date: 2/9/99 |
| technology: HTC35000 | company: Mitth gskolan, ITE | sheet: 1 of 1 |





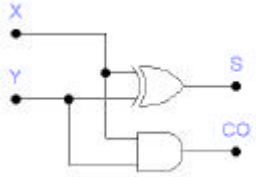
# Adderare och subtraherare

- **Halv adderare**
- **Full adderare**
- **Rippel adderare**
- **Full subtraherare**
- **Rippel subtraherare**
- **Adderare/Subtraherare krets**
- ***Carry-Look-Ahead* adderare**

# Halv adderare (*half adder*)

- **Sanningstabell:**

| X | Y | S | CO |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |




- **S = $X \oplus Y = X' \cdot Y + X \cdot Y'$**
- **CO = X·Y**

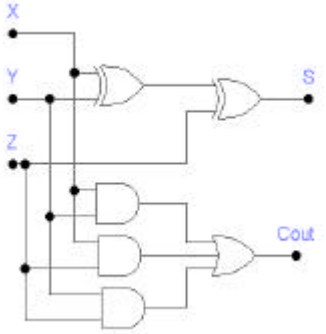




# Full adderare

- **Sanningstabell:**

| X | Y | Cin | S | Cout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |




- **S = X'·Y'·Cin + X·Y'·Cin + X·Y·Cin' + X·Y·Cin**
  **S = X xor Y xor Cin**
- **Cout = X·Y + X·Cin + Y·Cin**





# Full-adderare i VHDL

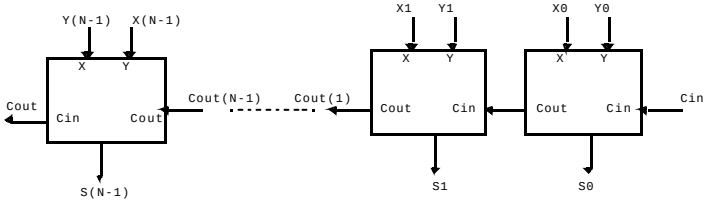
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity fa is
  port (
    x, y, cin : in  std_logic;
    s, cout   : out std_logic);
end fa;

architecture rtl of fa is

begin  -- rtl
  s <= x xor y xor cin;
  cout <= (x and y) or (y and cin) or (cin and x);
end rtl;
```





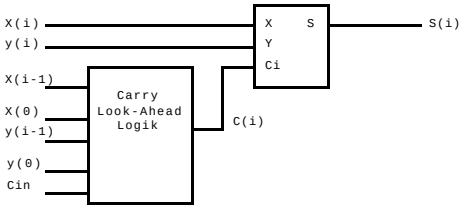
# Rippel adderare

- **Kaskadkoppling av *n* stycken full-adderare för att få en *n*-bits adderare**



## *Carry Look-Ahead* adderare

- Beräkning av summan i en addition är enkel och tar i sig kort tid
- Problemet med rippel adderare är att carry måste passera alla bitar
- Med speciell logik, som kallas *Carry Look-Ahead* logik, kan carry $c_i$ beräknas från $x_{i-1},y_{i-1} \ldots x_0,y_0$








## *Carry Look-Ahead* Logik

- *Cary Look-Ahead* logiken är baserad på två funktioner:
  - ***Carry Generate*: Ett additionssteg *i* sägs generera en carry om det producerar en carry = 1 oberoende av ingångarna $x_{i-1},y_{i-1} \ldots x_0,y_0, c_0$.**
  - ***Carry Propagate:* Ett additionssteg i sägs propagera en carry om den producerar en carry = 1 då ingångarna $x_{i-p},y_{i-1} \ldots x_0,y_0, c_0$, orsakar en carry-in = 1 ($C_i$ = 1)**
- Logiska uttryck baserat på det ovanstående:
  - ***Generate*: $g_i = x_i \cdot y_i$**
  - ***Propagate*: $p_i = x_i + y_i$**
- Ett steg genererar en carry ($c_{i+1}$) om båda operatorerna ($xi,yi$) är 1, och det propagerar en carry om åtminstode en operator är 1. $C_{i+1}$ kan skrivas som:

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





## forts. *Carry Look-Ahead* Logik

- För att undvika att rippla carry från steg till steg expanderas uttrycken rekursivt:

$c_1 = g_0 + p_0 \cdot c_0$

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

$c_3 = g_2 + p_2 \cdot c_2$
$= g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 p_0 \cdot c_0)$
$= g_2 + p_2 \cdot g_1 + p_2 p_1 \cdot g_0 + p_2 \cdot p_1 p_0 \cdot c_0$

$c_4 = g_3 + p_3 \cdot c_3$
$= g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 p_1 \cdot g_0 + p_2 p_1 p_0 \cdot c_0)$
$= g_3 + p_3 \cdot g_2 + p_3 p_2 \cdot g_1 + p_3 \cdot p_2 p_1 \cdot g_0 + p_3 p_2 p_1 p_0 \cdot c_0$





## 12-bitars Carry Look-Ahead Adderare i VHDL 1(3)

```
entity PG is
  port (
    X, Y : in std_logic_vector(3 downto 0);
    P : out std_logic_vector;
    G : out std_logic_vector  );
end PG;

architecture rtl of PG is
begin  -- rtl
  PG : process (X, Y)
  begin  -- process PG
    P <= X or Y;
    G <= X and Y;
  end process PG;
end rtl;
```

```
entity add is
  port (X, Y, C : in  std_logic_vector(3 downto 0);
        S       : out std_logic_vector(3 downto 0));
end add;

architecture rtl of add is
begin  -- rtl
  sum : process (X, Y, C)
  begin  -- process sum
    S <= X xor Y xor C;
  end process sum;
end rtl;
```

```
entity CLL is
  port (
    P, G : in  std_logic_vector(3 downto 0);
    C0   : in  std_logic;
    C    : out std_logic_vector(4 downto 1));
end CLL;

architecture rtl of CLL is
begin  -- rtl
  CLL : process (P, G, C0)
  begin  -- process CLL
    C(1) <= G(0) or (P(0) and C0);
    C(2) <= G(1) or (P(1) and G(0)) or (P(1) and P(0) and C0);
    C(3) <= G(2) or (P(2) and G(1)) or (P(2) and P(1) and G(0)) or (P(2) and P(1) and P(0) and C0);
    C(4) <= G(3) or (P(3) and G(2)) or (P(3) and P(2) and G(1)) or (P(3) and P(2) and P(1) and G(0))
                 or (P(3) and P(2) and P(1) and P(0) and C0);
  end process CLL;
end rtl;
```

# 12-bitars Carry Look-Ahead Adderare i VHDL 2(3)

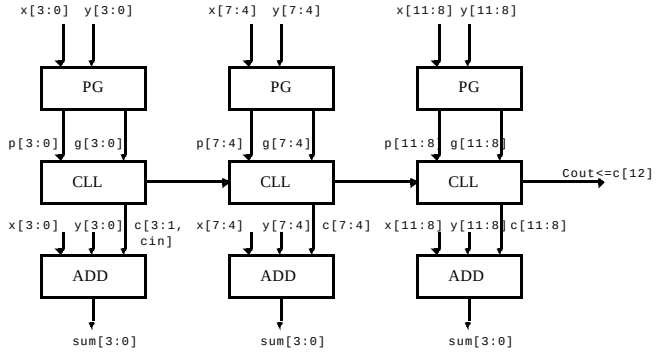
```
use work.CLA_package.all;

entity CLA12 is
    port (
    A, B : in  std_logic_vector(11 downto 0);
    CIN  : in  std_logic;
    SUM  : out std_logic_vector(11 downto 0);
    COUT : out std_logic);
end CLA12;

architecture rtl of CLA12 is
  signal C : std_logic_vector(12 downto 1);
  signal P, G : std_logic_vector(11 downto 0);

begin  -- rtl
  PG0 : PG port map (
    X => A(3 downto 0),
    Y => B(3 downto 0),
    P => P(3 downto 0),
    G => G(3 downto 0));

  PG1 : PG port map (
    X => A(7 downto 4),
    Y => B(7 downto 4),
    P => P(7 downto 4),
    G => G(7 downto 4));

  PG2 : PG port map (
    X => A(11 downto 8),
    Y => B(11 downto 8),
    P => P(11 downto 8),
    G => G(11 downto 8));

  CLL0 : CLL port map (
    P  => P(3 downto 0),
    G  => G(3 downto 0),
    C0 => CIN,
    C  => C(4 downto 1));

  CLL1 : CLL port map (
    P  => P(7 downto 4),
    G  => G(7 downto 4),
    C0 => C(4),
    C  => C(8 downto 5));

  CLL2 : CLL port map (
    P  => P(11 downto 8),
    G  => G(11 downto 8),
    C0 => C(8), C  => C(12 downto 9));

  SUM0 : ADD port map (
    X => A(3 downto 0),
    Y => B(3 downto 0),
    C(3 downto 1) => C(3 downto 1),
    C(0) => CIN,
    S => SUM(3 downto 0));

  SUM1 : ADD port map (
    X => A(7 downto 4),
    Y => B(7 downto 4),
    C => C(7 downto 4),
    S => SUM(7 downto 4));

  SUM2 : ADD port map (
    X => A(11 downto 8),
    Y => B(11 downto 8),
    C => C(11 downto 8),
    S => SUM(11 downto 8));

  COUT <= C(12);
end rtl;
```





# 12-bitars Carry Look-Ahead Adderare i VHDL 3(3)

- **Struktur:**








# Adderare i VHDL

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

entity add_sub 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_sub;

architecture rtl of add_sub 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_sub;

end rtl;
```

Exempel på syntesresultat:

| max. delay | Typ | Area |
|---|---|---|
| 35 ns | CRA | 200 a.e |
| 5 ns | CLA | 700 a.e |

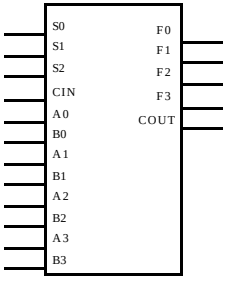




# Aritmetisk och Logisk enhet (ALU)

- **En ALU är en enhet som består av kombinatorisk logik och kan utföra ett antal aritmetiska och logiska operationer på två n-bitars operander**
- **Vilken funktion som ska användas bestäms av en antal kontrollsignaler**
- **Exempel på ALU funktioner (74x382)**

| S2 | S1 | S0 | FUNKTION |
|---|---|---|---|
| 0 | 0 | 0 | F = 0000 |
| 0 | 0 | 1 | F = B - A - 1 + Cin |
| 0 | 1 | 0 | F = A - B - 1 + Cin |
| 0 | 1 | 1 | F = A + B + Cin |
| 1 | 0 | 0 | F = A xor B |
| 1 | 0 | 1 | F = A and B |
| 1 | 1 | 0 | F = A or B |
| 1 | 1 | 1 | F = 1111 |



# Exempel : 74x382:s ALU funktioner i VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity ALU is
    generic (
    N : integer := 4);

  port (
    A, B : in  std_logic_vector(N-1 downto 0);
    S    : in  std_logic_vector(2 downto 0);
    CIN  : in  std_logic;
    F    : out std_logic_vector(N-1 downto 0);
    COUT : out std_logic);
end ALU;

architecture rtl of ALU is
begin  -- rtl
  sel_func : process (A, B, S, CIN)
  variable F_var: std_logic_vector(N downto 0);
  variable zero_vector : std_logic_vector(N-1 downto 0) := (others=>'0');

  begin  -- process sel_func
    case S is
      when "000" => F_var := (others => '0');
      when "001" => F_var := ('0' & B) - ('0' & A) - (zero_vector & '1') + (zero_vector & CIN);
      when "010" => F_var := ('0' & A) - ('0' & B) - (zero_vector & '1') + (zero_vector & CIN);
      when "011" => F_var := ('0' & A) + ('0' & B) + (zero_vector & CIN);
      when "100" => F_var := ('0' & A) xor ('0' & B);
      when "101" => F_var := ('0' & A) and ('0' & B);
      when "110" => F_var := ('0' & A) or ('0' & B);
      when "111" => F_var := (others => '1');
      when others => null;
    end case;
    COUT <= F_var(N);
    F <= F_var(N-1 downto 0);
  end process sel_func;
end rtl;
```