TSEA83 : Datorkonstruktion
Fö8

VHDL 1/3

LINKÖPINGS UNIVERSITET

Fö8 : Agenda

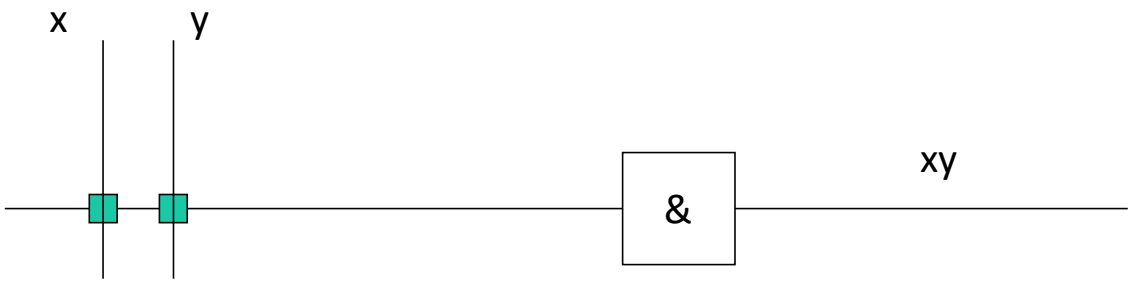
- Programmerbara kretsar
  -CPLD
  -FPGA
- VHDL
  -Kombinatorik
  `with-select-when`
  `when-else`
  -Sekvensnät
  `process`
  `case`
  `if-then-else`
- Extra redovisningstillfälle
- Gruppbildning, klar!
  -Projektanmälan
  -Kravspec

Programmerbara kretsar

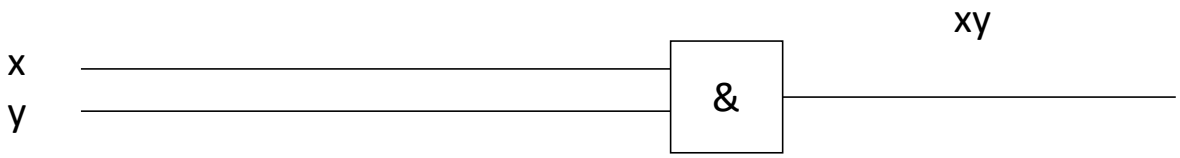
Programmerbara kretsar

- PLD = programmable logic device
- CPLD = complex PLD,
  i princip flera PLD-er på ett chip
  ex: 108 vippor + 540 produkttermer
- FPGA = field programmable gate array,
  komplexa kretsar upp till flera miljoner grindar.
  Ex: 20000 vippor + komb.logik
  + 32 2 kB RAM + 32 DSP (mult+ack)

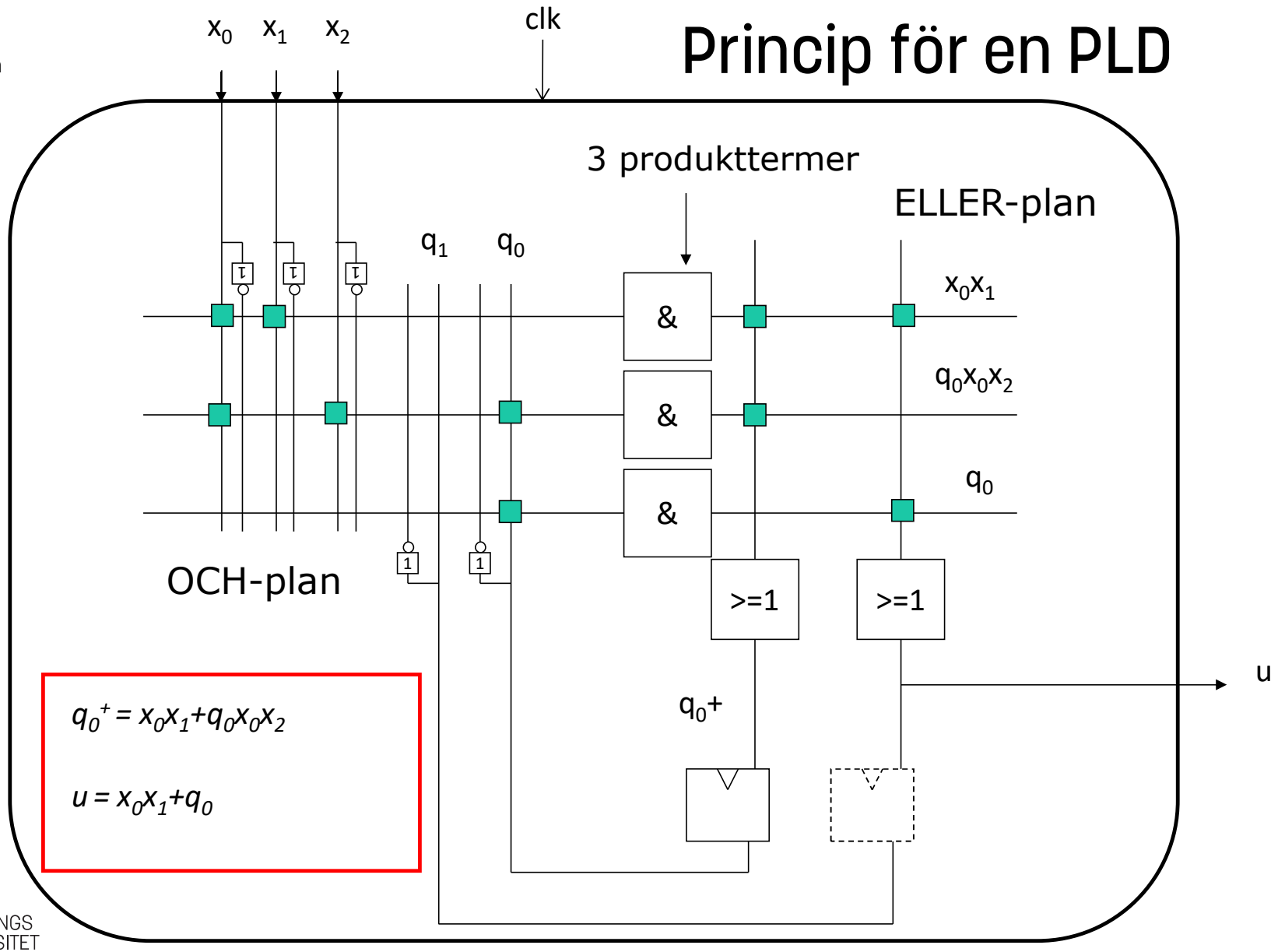
En notation



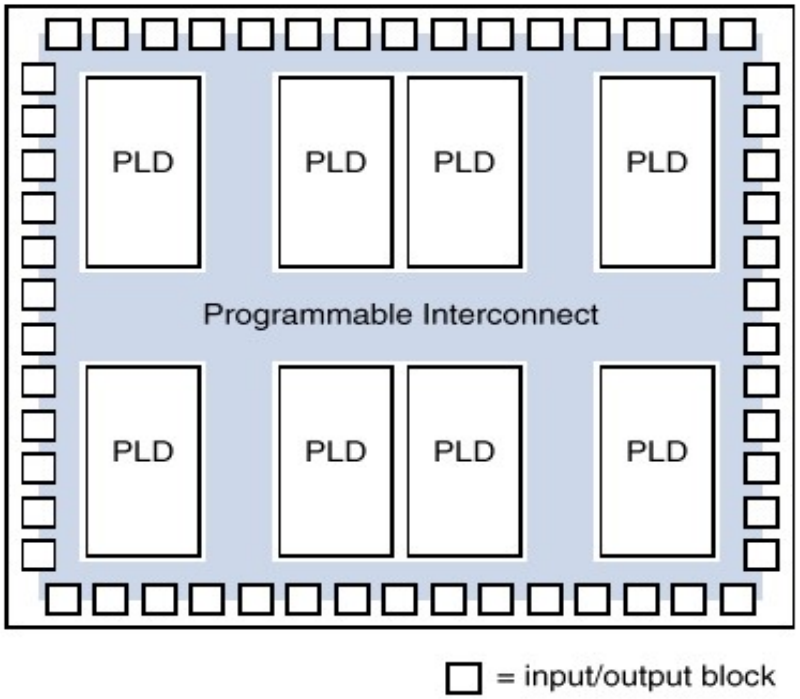
betyder



Princip för en PLD

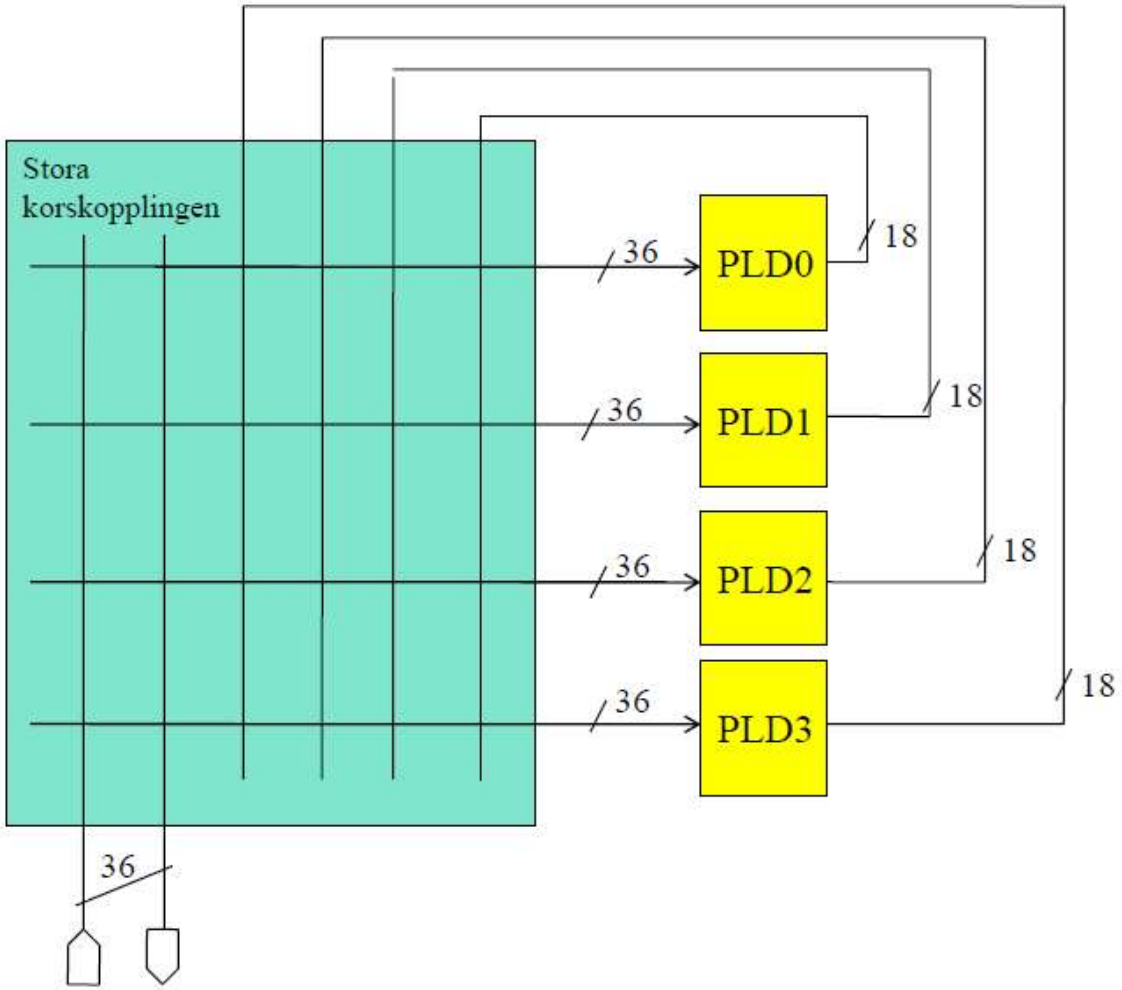


En allmän CPLD

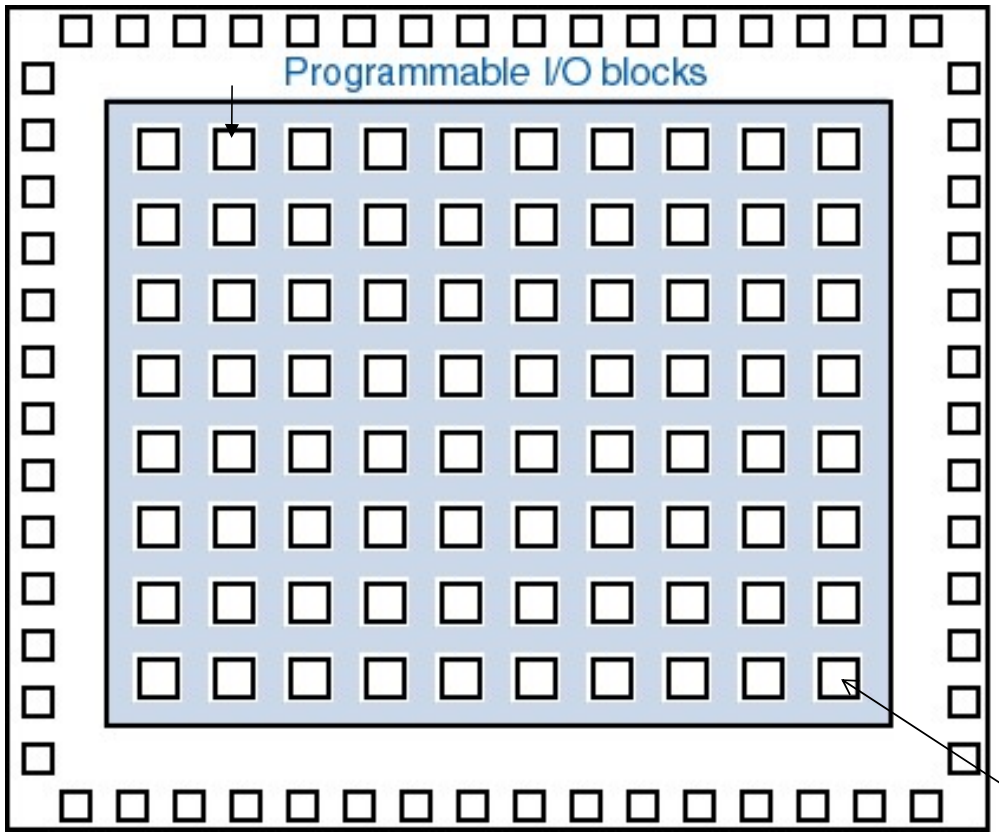


En samling PLDer på ett chip med programmerbara hopkopplingar

CPLD Xilinx 9572 - blockschema



General FPGA chip architecture



= Programmable interconnect

= Programmable logic block

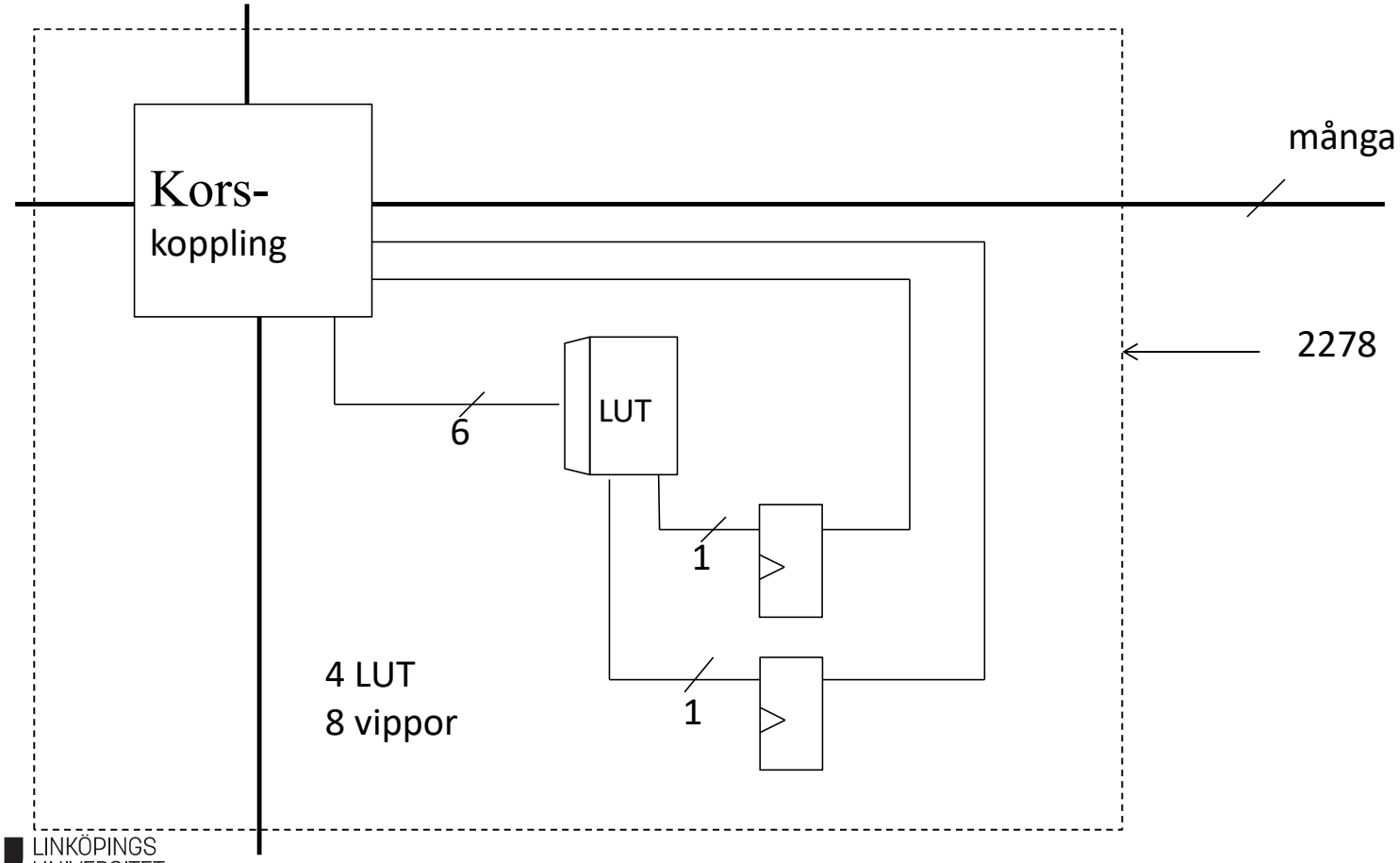
= I/O pad

Dessutom 32 x 2kB RAM
32 x DSP (mult + add)
Vissa FPGA-er innehåller CPU-er

CLB = configurable logic block

Vad innehåller en CLB?

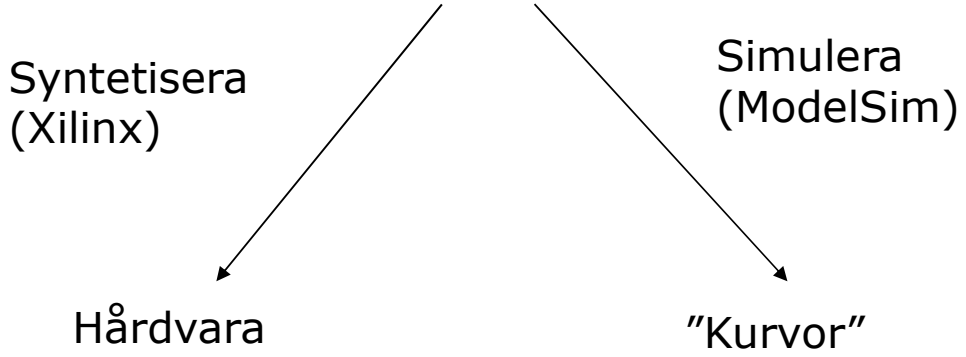
Logik görs med LUT (look up table)
”fyll i sanningstabellen i ett minne”



VHDL

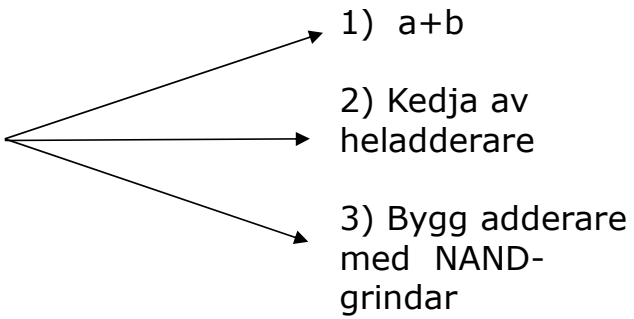
VHDL

- VHDL=VHSIC Hardware Description Language
  - VHSIC = Very High Speed Integrated Circuit
- Ett programspråk för att:



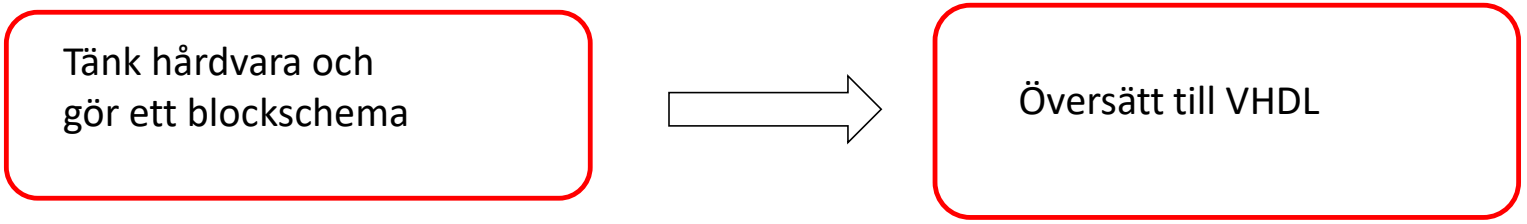
Varför VHDL?

- Hantera komplexitet
  - VHDL-koden kan simuleras
  - Beskrivning på flera olika abstraktionsnivåer
- Ökad produktivitet
  - snabbare än schemaritning
  - återanvändbar kod
- Modernt programmeringsspråk
  - Rikt, kraftfullt
  - Parallellt, ADA-liknande, starkt typat, overloading
  - Ej objektorienterat



VHDL nackdelar?

- Svårt att lära sig?
  - **Delmängd för syntes : 1-2 dagar!**
  - Avancerade simuleringar : 1-2 månader
- Nytt sätt att tänka
  - Lätt att hamna i mjukvarutänkande!
  - FPGA-n, CPLD-n är **inte en processor för VHDL**
  - VHDL är **inte sekvensiellt utan parallellt**
  - Tilldelning, variabler betyder inte samma sak som i andra prog.språk
  - Gör så här:



Hur ser ett VHDL-program ut?

```
entity namn1 is
  -- beskrivning av in- och utgångar
end entity namn1;
```

Gränssnitt mot omvärlden

```
architecture namn2 of namn1 is
  -- beskrivning av interna signaler
begin
  -- beskrivning av funktion
end architecture namn2;
```

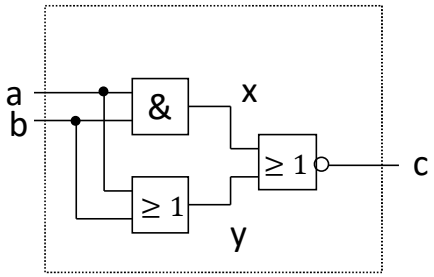
Funktion

VHDL är inte case sensitive, små eller stora bokstäver spelar ingen roll, ej heller mellanslag.

VHDL

Kombinatorik

VHDL för kombinatoriska nät




```
entity cnet is
  port(a,b: in std_logic;
       c: out std_logic);
end entity cnet;

architecture firsttry of cnet is
  signal x,y:std_logic;
begin
  c <= x nor y;
  x <= a and b;
  y <= a or b;
end architecture firsttry;
```

Parallellt ”exekverande” satser. När ex `a` ändras så blir `x<=a and b` och `y<= a or b`, vilket gör att `c<= x nor y`.
Ordningen spelar ingen roll.

Vad betyder ett VHDL-program?

Syntetisering (Xilinx)

- **x <= a and b;**
  betyder att en OCH-grind kopplas in
  mellan trådarna a,b och x

  Endast en tilldelning på x tillåten.

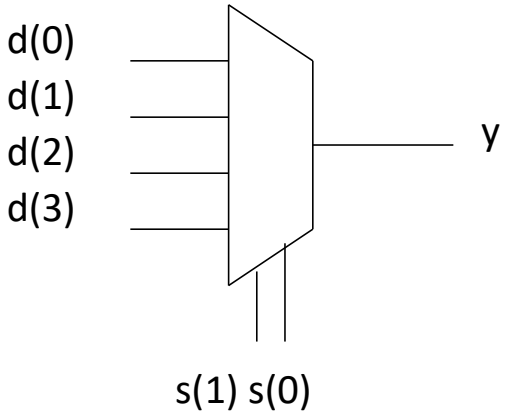
Simulering (ModelSim)

- **x <= a and b;**
  är en parallellt exekverande sats
  som körs om a,b ändras

Än så länge är ordningen mellan satserna oviktig
”Programmera” aldrig i VHDL!
Tänk hårdvara => översätt till VHDL

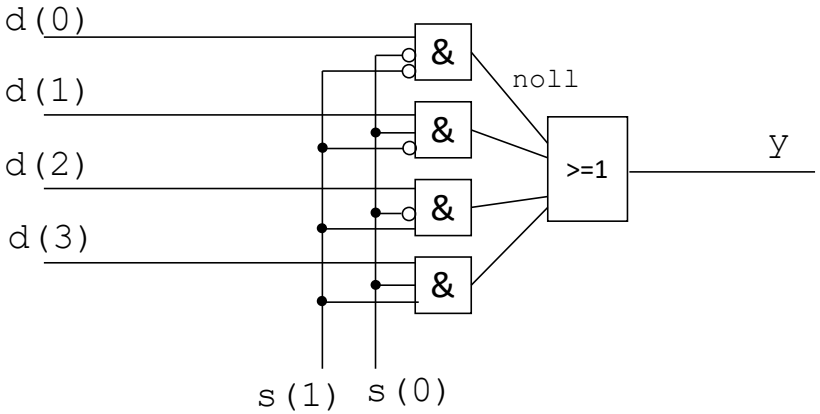
En multiplexer

```
entity mux is
  port( d: in std_logic_vector(0 to 3);
        s: in std_logic_vector (1 downto 0);
        y: out std_logic);
end entity mux;
```



Multiplexern, forts

```
architecture booleq of mux is
signal noll,ett,tva,tre: std_logic;
begin
  noll <= not s(1) and not s(0) and d(0);
  ett <= not s(1) and s(0) and d(1);
  tva <=  s(1) and not s(0) and d(2);
  tre <= s(1) and s(0) and d(3);
  y <= noll or ett or tva or tre;
end architecture booleq;
```



Multiplexern, forts

VHDL har en programsats som precis motsvarar en mux:

```
architecture behavior1 of mux is
begin
  with s select
    y <= d(0) when ”00”,
         d(1) when ”01”,
         d(2) when ”10”,
         d(3) when others;
end architecture behavior1;
```

Lägg märke till:
- det finns **enn** <= i programsatsen.
- enn rad är sann

With-select-when

- Är en parallell sats, concurrent statement
- Endast utanför `process`

```
with (styrsignal) select
 (utsignal) <= (uttryck 1) when (signalvärde 1),
               (uttryck 2) when (signalvärde 2),
               ...
               (uttryck n-1) when (signalvärde n-1),
               (uttryck n) when others;
```

OBS: samtliga värden på `styrsignal` måste täckas!

Multiplexern, forts

```
architecture behavior2 of mux is
begin
    y <= d(0) when s = ”00” else
         d(1) when s = ”01” else
         d(2) when s = ”10” else
         d(3);
end architecture behavior2;
```

When-else

- Är en parallell sats, concurrent statement
- Endast utanför `process`

```
(signal) <= (Signal1) when (Villkor1) else
        (S2) when (V2) else
        ...
        (SN-1) when (VN-1) else
        (SN);
```

Lägg märke till:

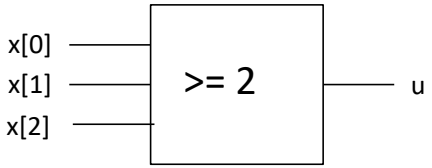
- det finns enn <= i satsen.
- noll eller flera villkor är sanna (första sanna villkoret ger tilldelning)

$$S = S_1V_1 + S_2\overline{V}_1V_2 + S_2\overline{V}_1\overline{V}_2V_3 \ldots$$

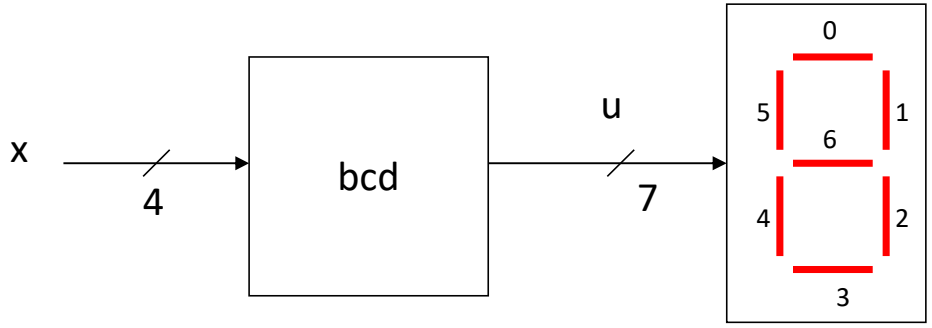
Kommentar

Både **`with-select-when`** och **`when-else`**
kan uttrycka vilken Boolesk funktion (K-nät) som helst!

```
signal x: std_logic_vector(2 downto 0);
signal u: std_logic;

-- man kan skriva så här
with x select
    u   <= ‘1’ when “011”,
           ‘1’ when “101”,
           ‘1’ when “110”,
           ‘1’ when “111”,
           ‘0’ when others;

-- eller så här
u <= ’1’ when x=3 else
     ’1’ when x>4 else
     ’0’;

-- eller …
```



Exempel: BCD -> 7-segment



x	u
0000	0111111
0001	0000110
...	...

Exempel: BCD -> 7-segment

```
entity bcd is
    port ( x : in std_logic_vector(3 downto 0);
           u : out std_logic_vector(6 downto 0));
end bcd;

architecture sanningstabell of bcd is
begin
  with x select
  u <= "0111111" when "0000",
    "0000110" when "0001",
    "1011011" when "0010",
    "1001111" when "0011",
    "1100110" when "0100",
    "1101101" when "0101",
    "1111100" when "0110",
    "0000111" when "0111",
    "1111111" when "1000",
    "1100111" when "1001",
    "1111001" when others;

end sanningstabell;
```

```
u <= "0111111" when x="0000“ else
     "0000110" when x="0001“ else
     "1011011" when x="0010“ else
     ...

     kanske inte lika bra då
     x=... måste upprepas.
```

Vad har vi så långt?

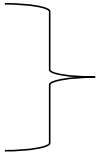
- **`entity`** beskriver gränssnittet
- **`architecture`** beskriver innehållet
- Mellan **`begin`** och **`end`** har vi parallella satser.
  - ”vanlig” signaltilldelning **`c <= a and b;`**
  - **`with-select-when`** är en mux.
  - **`when-else`** är en generaliserad mux.
  - Ovanstående används för kombinatorik utanför process-satsen

VHDL

Sekvensnät

Vad kommer nu?

- VHDL för sekvensnät,
- process-satsen
  - case-when
  - if-then-else



Endast inuti process-sats!

Sekvensnät – en D-vippa

```
entity de is
  port(d,clk: in STD_LOGIC;
           q: out STD_LOGIC);
end de;

architecture d_vippa of de is
begin
  process(clk)
  begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end d_vippa;
```

Processen exekveras när `clk` ändras sensitivity list

`q` uppdateras på positiv `clk`-flank

I en `process(clk)`-sats gäller (med rising_edge(clk))
=> alla VL får en vippa på sig!

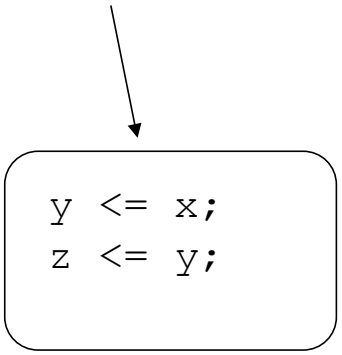
Vad blir detta?

```
process(clk)
begin
  if rising_edge(clk) then
    y <= x;
    z <= y;
  end if;
end process;
```

Låt y=0, z=0.
Sätt `x=1` och klocka en gång. Då blir väl `z=y=1`?

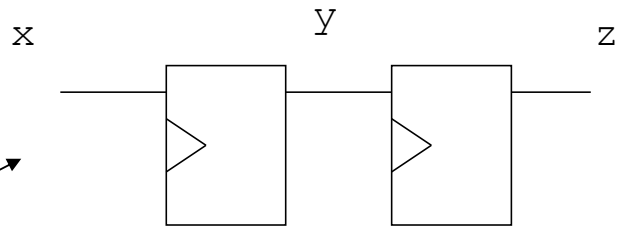
Så här blir det!

Inuti klockad process



synt

Alla VL får
en vippa på sig



sim

Alla VL får
ett + på sig

$y^+ = x;$
$z^+ = y;$

} Nästa värde

$y = y^+;$
$z = z^+;$

} Tilldelning

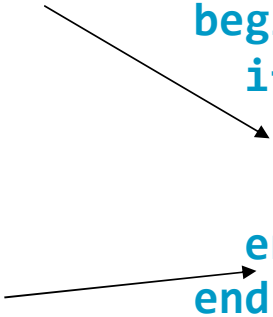
Simulering

Ibland får man höra att: Koden inom processen
exekveras ”sekvensiellt”!

Det är bara halvt sant!

Så här är det:
(eller så här gör ModelSim):

1) Evaluera nästa värde sekvensiellt
   $y^+ = x$
   $z^+ = y$
2) Uppdatera parallellt
   $y = y^+$
   $z = z^+$

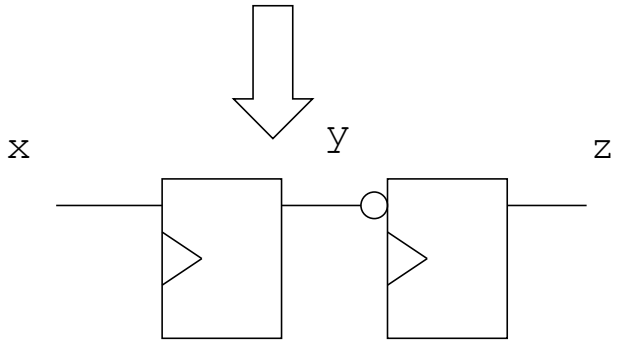


```
process(clk)
begin
  if rising_edge(clk) then
    y <= x;
    z <= y;
  end if;
end process;
```

Till slut

```
process(clk)
begin
  if rising_edge(clk) then
      y <= x;
      z <= y;
      z <= not y;
  end if;
end process;
```

Är faktiskt OK!
Men skriv inte så!




```
-- 1
process(clk)
begin
  if rising_edge(clk) then
      y <= x;
  end if;
end process;

-- 2
process(clk)
begin
  if rising_edge(clk) then
      z <= y;
  end if;
end process;

-- 3
process(clk)
begin
  if rising_edge(clk) then
      z <= not y;
  end if;
end process;
```

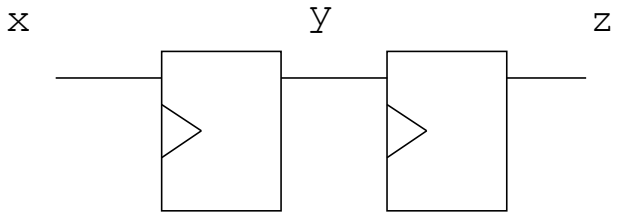
2 och 3 ihop går inte!

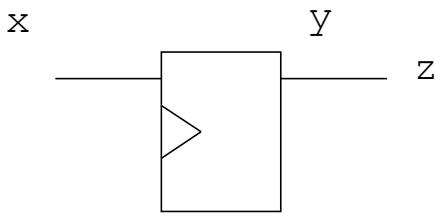
Ett exempel

```
process(clk)
begin
  if rising_edge(clk) then
      y <= x;
      if (y='1') then
        z <= '1';
      else
        z <= '0';
      end if;
  end if;
end process;
```

```
process(clk)
begin
  if rising_edge(clk) then
    y <= x;
  end if;
end process;

z <= '1' when (y='1')
     else '0';
```

Ger båda samma funktion för z?

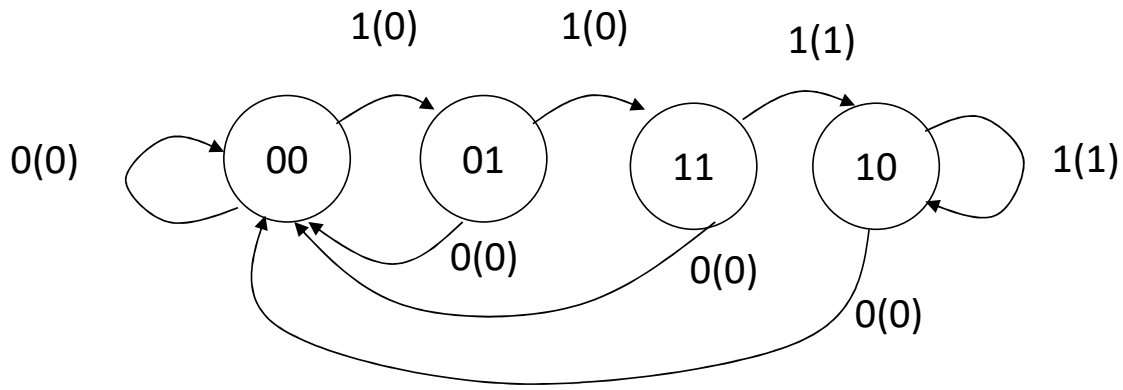




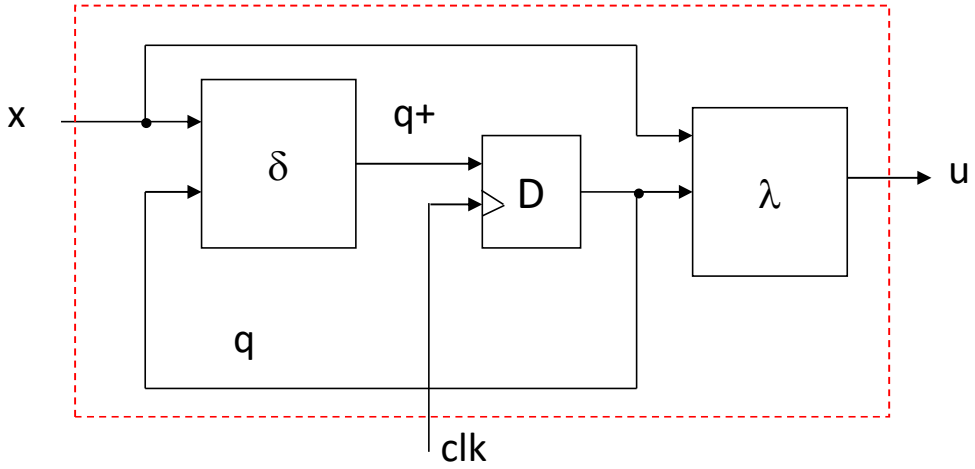
Nej, z till vänster kommer en klockpuls senare än z till höger.

Ett annat exempel

Bygg ett sekvensnät, som ger utsignalen 1 när insignalen varit 1 i minst tre klockcykler i rad.



Sekvensnät - Mealy




```
entity sn is
  port(x,clk: in std_logic;
       u: out std_logic);
end sn;

architecture booleq2 of sn is
  signal q: std_logic_vector(1 downto 0);
begin
  process(clk)
  begin
    -- delta: nästa-tillstånd
  end process;
    -- lambda: utsignal
end booleq2;
```

Case-when

- Endast inuti **process**
- Måste beskriva vad som händer för alla värden på styrsignal
- motsvarar **with-select-when**, men är kraftfullare

```
case (styrsignal) is
  when (värde 1)   => (sats 1);
  when (värde 2)   => (sats 2);
...
  when (värde n-1) => (sats n-1);
  when others      => (sats n);
end case;
```

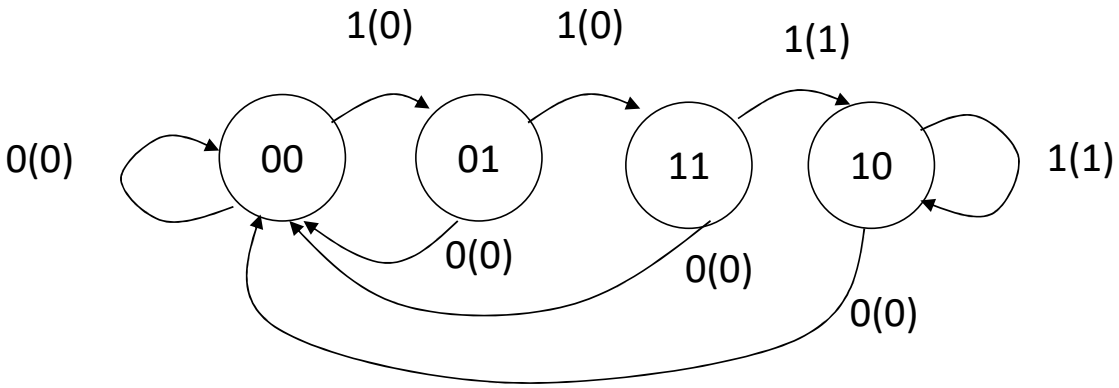
If-then-else

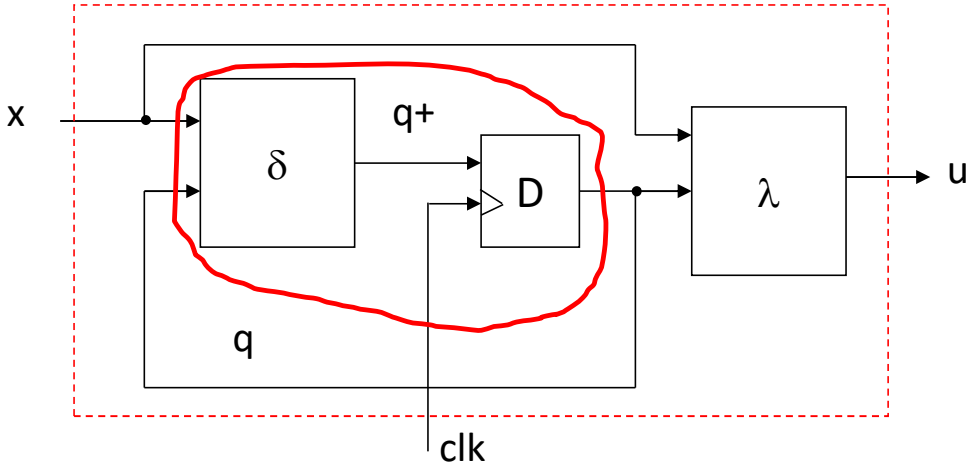
```
if (uttryck 1) then
  (sats 1)
elsif (uttryck 2) then
  (sats 2)
elsif (uttryck n-1) then
  (sats n-1)
else
  (sats n)
end if;
```

- Endast inuti **`process`**
- motsvarar **`when-else`**,

men är kraftfullare

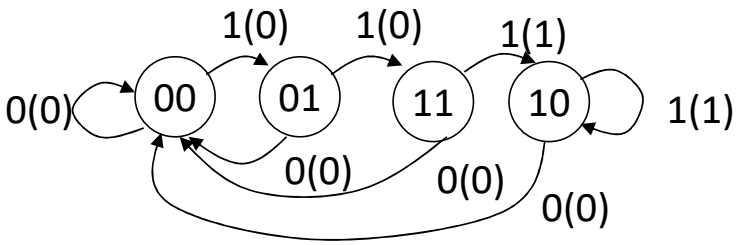
Vi slår ihop δ-nätet och tillståndsvipporna



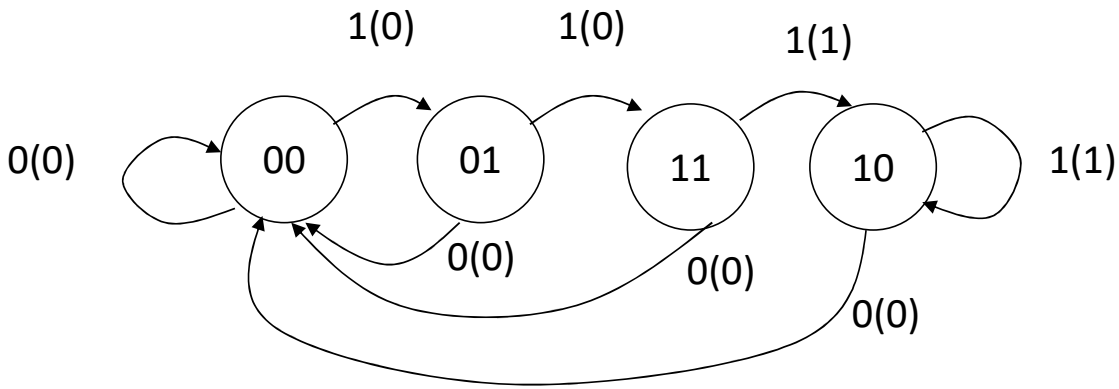


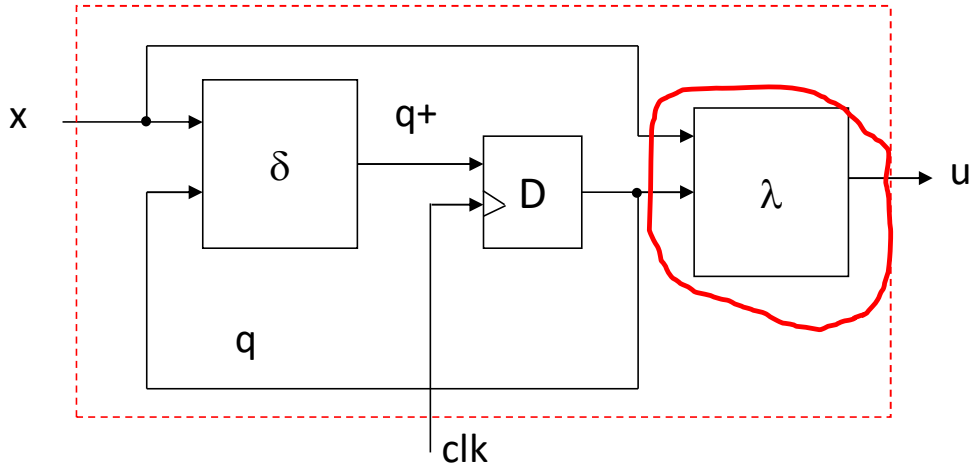
δ-nätet och tillståndsvipporna

```
-- delta: nästa-tillstånd
process(clk)
begin
  if rising_edge(clk)then
    case q is
      when ”00” => if x=’1’ then q <= ”01”;
                   else q <= ”00”;
                   end if;
      when ”01” => if x=’1’ then q <= ”11”;
                   else q <= ”00”;
                   end if;
      when ”11” => if x=’1’ then q <= ”10”;
                   else q <= ”00”;
                   end if;
      when ”10” => if x=’1’ then q <= ”10”;
                   else q <= ”00”;
                   end if;
      when others => q <= ”00”;
    end case;
  end if;
end process;
```



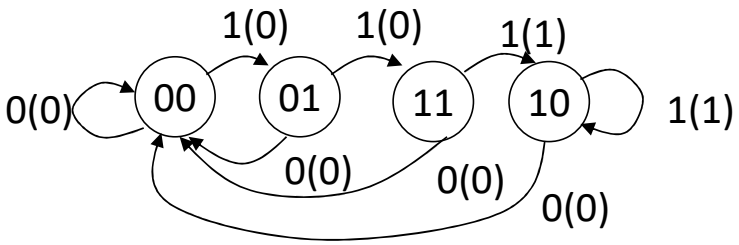
λ-ekvationerna ska vara kombinatorik





λ-ekvationen

K-nät utanför processen




```
u <= '1' when q="11" and x='1' else
     '1' when q="10" and x='1' else
     '0';
```

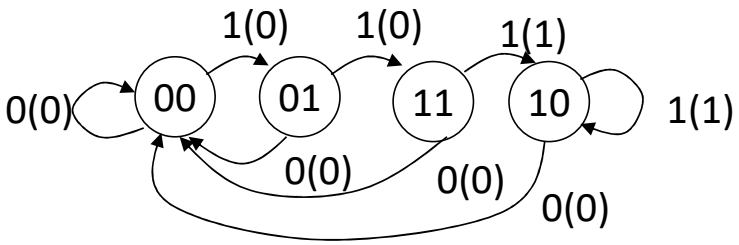
eller optimerat (fixas av syntesverktyget):

```
u <= x and q(1);
```

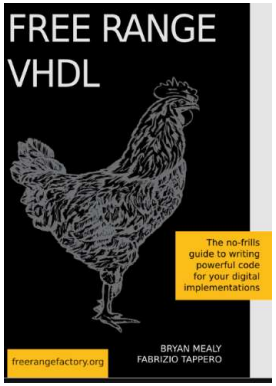
Kommentar: `and` funkar alltså på typerna `std_logic` och `boolean`

Komplett kod

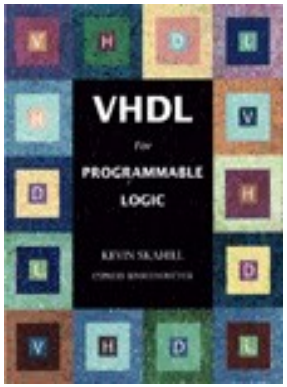
```
entity sn is
  port(x,clk: in std_logic;
       u: out std_logic);
end sn;

architecture booleq2 of sn is
  signal q: std_logic_vector(1 downto 0);
begin

 -- delta
 process(clk)
 begin
  if rising_edge(clk)
   case q is
     when ”00” => if x=’1’ then q <= ”01”;
                  end if;
     when ”01” => if x=’1’ then q <= ”11”;
                  else q <= ”00”;
                  end if;
     when ”11” => if x=’1’ then q <= ”10”;
                  else q <= ”00”;
                  end if;
     when ”10” => if x=’0’ then q <= ”00”;
                  end if;
     when others => q <= ”00”;
   end case;
  end if;
 end process;

 -- lambda

 u <= x and q(1);
end booleq2;
```



Mer om VHDL



Mealy, Tappero: *Free Range VHDL,* freerangefactory.org



Kevin Skahill: *VHDL for programmable logic,*
Addison-Wesley

Extra redovisningstillfälle

Extra redovisningstillfälle

Imorgon 9/2 kl 15-17

För dig som har labb 1 eller 2 kvar.
Ingen anmälan, bara att ”dyka upp” i LabbKö Extra.

Gruppbildning

Gruppbildning

Gruppbildningen är klar!
-Ni har nu: en egen kanal i Teams och ett eget repo i Gitlab
-Kontrollera gruppbildningsdokumentet
-Ta kontakt med varandra
-Fyll i projektanmälan (se kurshemsidan under Projekt)
-Skriv en kravspec
-Lämna in projektanmälan **så snart det går**, i Gitlab.
-Lämna in kravspec ver 0.1 **senast må 14/2 kl 16**, i Gitlab.

Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET