Programmerbara kretsar och VHDL 1

Föreläsning 9
Digitalteknik, TSEA22
Oscar Gustafsson, Mattias Krysander
Institutionen för systemteknik

LINKÖPINGS UNIVERSITET

Dagens föreläsning

- Programmerbara kretsar
- Kombinationskretsar i VHDL

  with-select-when, when-else
- Sekvenskretsar i VHDL

  process, if-then-else
- In-/ut-signaler, datatyper, mm
- Räknare i VHDL
- Exempel

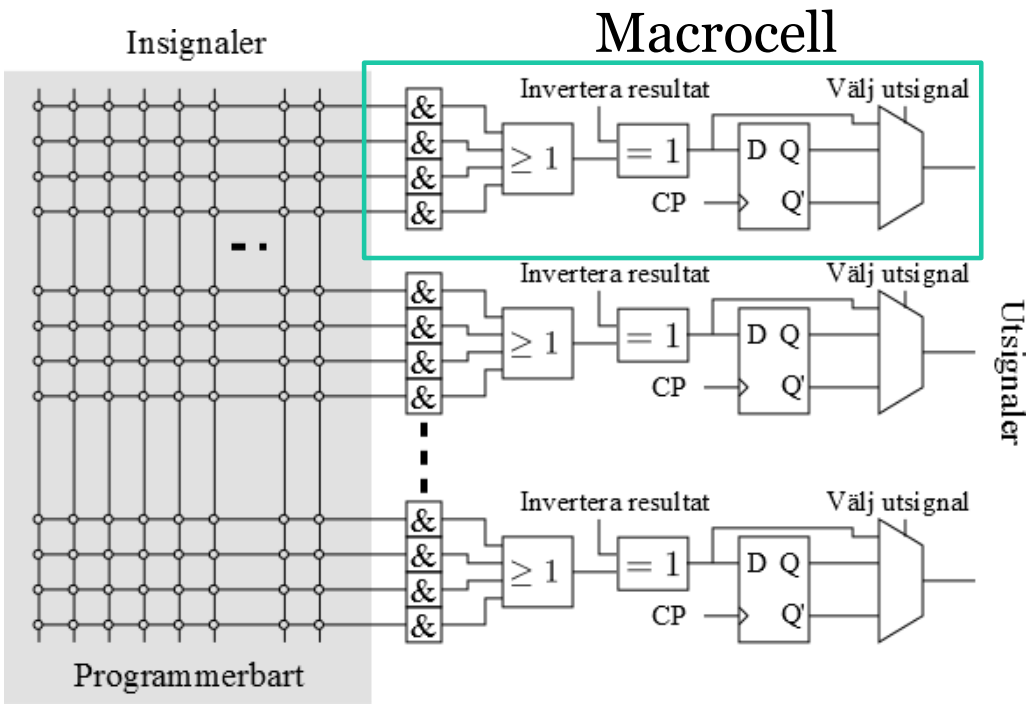
  Blockschema -> VHDL, kodstandard, simulering, programmering, verifiering
- Lab 3

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
  - 5 000 000 vippor
  - 2 000 000 Look-up-tables (LUT)
  - 500 Mb RAM
  - Digital Signal Processing (DSP)
  - Processor

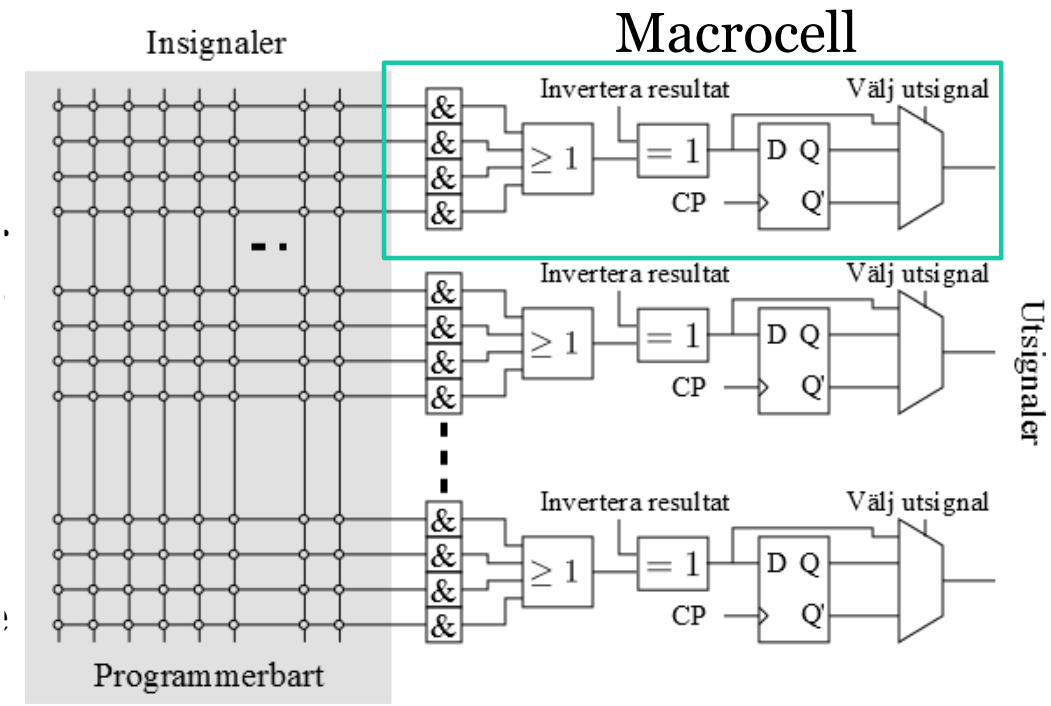
CPLD - konstruktion

- Grundblocket i en CPLD består oftast av ett AND-OR-nät
- AND-grindarnas ingångar är programmerbara
- Kallas Programmable Logic Array (PLA)

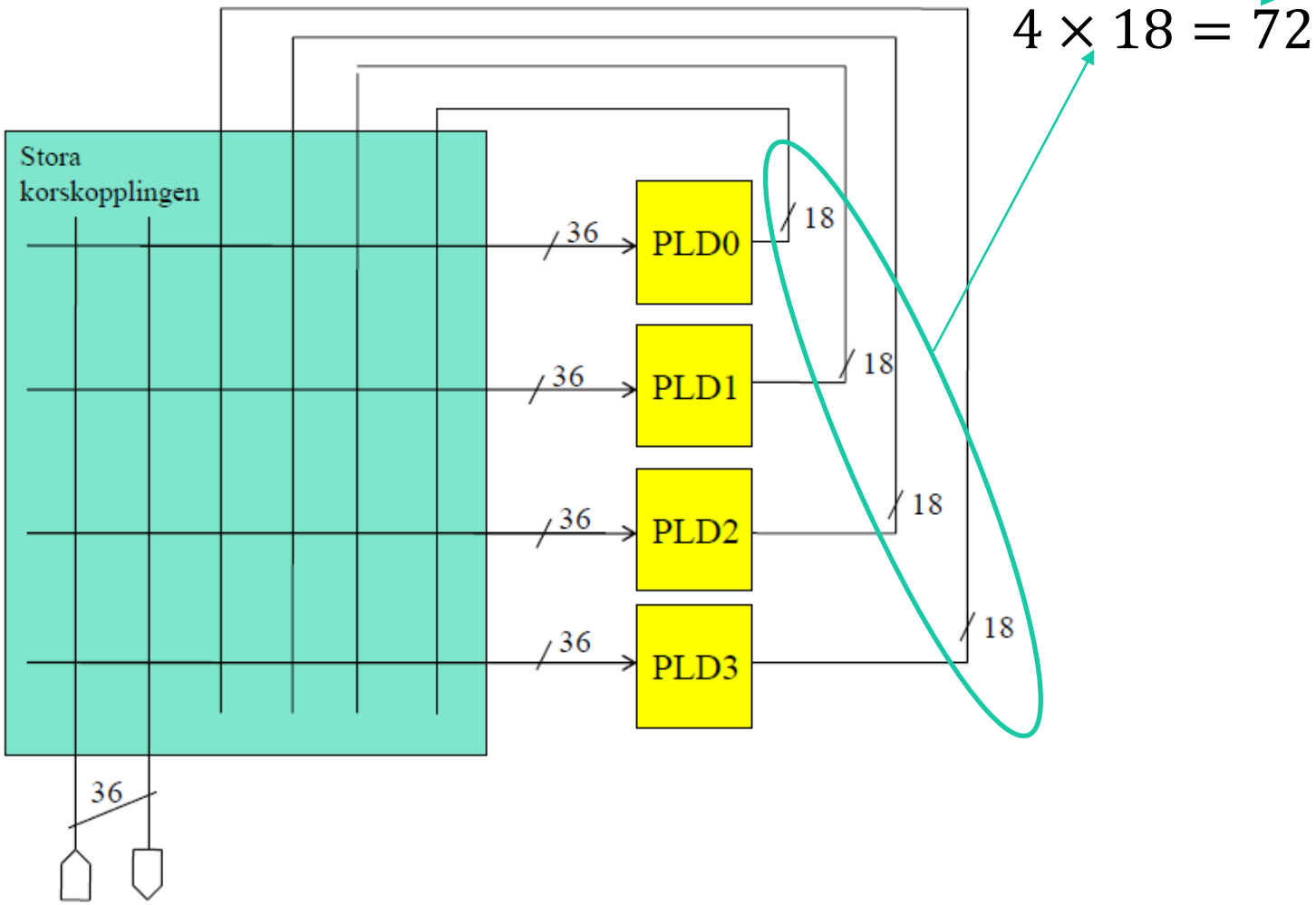


CPLD - konstruktion

- Alltså kan en CPLD skapa funktioner på summa-produkt-form
- Samt invertera dem om det behövs
- Och sätta en D-vippa på slutet vid behov



CPLD Xilinx 9572 - blockschema



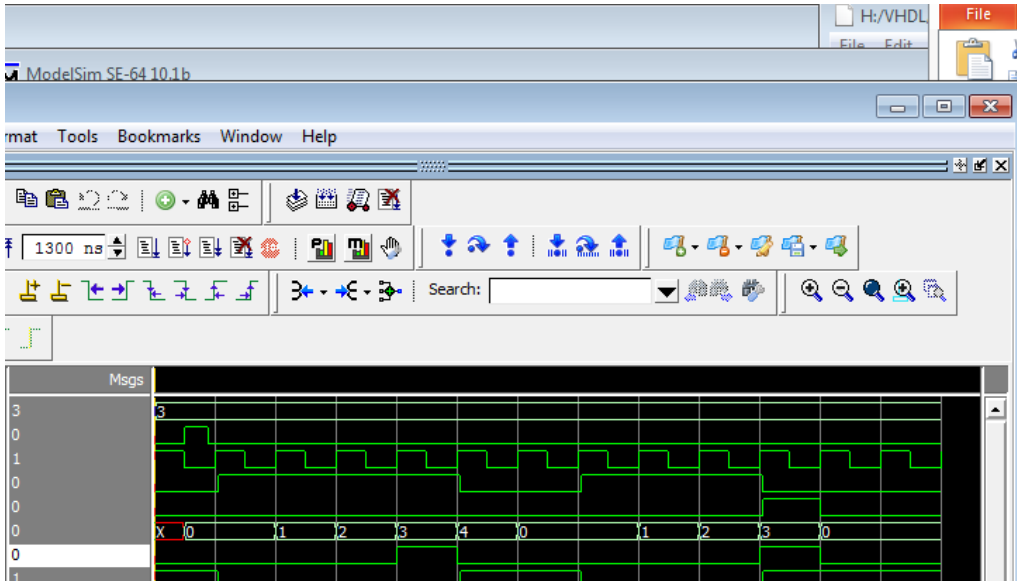
VHDL

Ett programspråk för att:

Syntetisera
(Xilinx)

Simulera
(ModelSim)

Hårdvara



VHDL

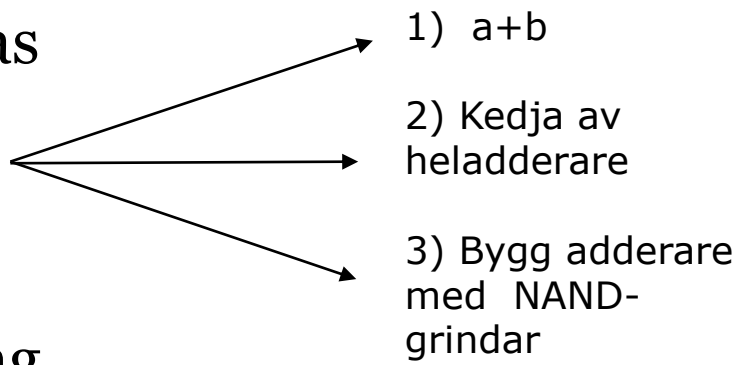
- VHDL = VHSIC HDL
- VHSIC = Very High Speed Integrated Circuits
- HDL = Hardware Description Language
- Hardware Description Language = hårdvarubeskrivande språk
- Hårdvarubeskrivande språk = språk som beskriver hårdvara
- Beskriver hårdvara
- Hårdvara (inte mjukvara)

VHDL

- VHDL är ett av två dominerande HDL
- Det andra är Verilog
- Verilog används mer i USA
- VHDL används mer i Europa
- VHDL uvecklades av USAs försvarsdepartement
- Lånar mycket syntax från ADA

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, 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 parallellt, inte sekvensiellt
  - Tilldelning, variabler betyder inte samma sak som i andra programmeringsspråk (för det är inte ett prog.språk)
  - Gör så här:

Tänk hårdvara och gör ett blockschema → Beskriv hårdvaran med VHDL

Hur ser ett VHDL-program ut?

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

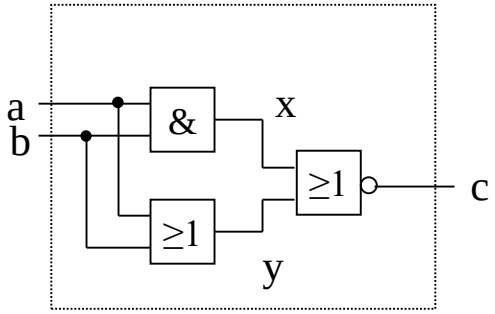
Gränssnitt mot omvärlden

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

Innehåll

VHDL är inte skiftlägeskänsligt (case sensitive), små eller stora bokstäver spelar ingen roll, ej heller mellanslag (förutom i namn och nyckelord då..)

VHDL för kombinatoriska kretsar




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

architecture firsttry of knet is
  signal x, y : std_logic;
begin
  c <= not (x or y);
  x <= a and b;
  y <= a or b;
end architecture firsttry;
```

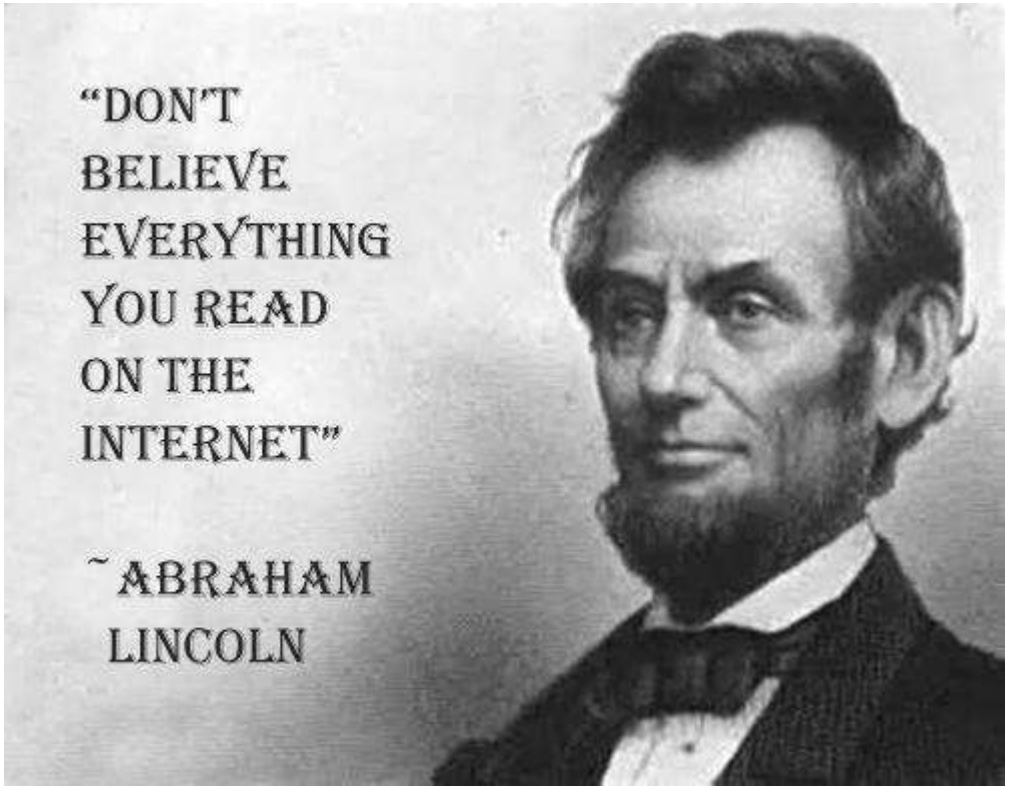
Parallellt exekverande satser.
Om `a` ändras så körs `x <= a and b` och `y <= a or b`,
vilket gör att `c <= x nor y` körs.
Ordningen spelar ingen roll.

Datatyper

- VHDL innehåller nästan inga datatyper
- Det är mycket hårt typat, inga underförstådda typkonverteringar här inte…
- I princip är nästan alla typer som används definierade ovanpå VHDL, men det har så klart skapats standarder som används
- Följdaktligen kan ni i princip glömma det som står här, förutom komma ihåg att det är hårt typat när ni råkar ut för det

Datatyper

- Samt att det innebär att om ni söker efter information på nätet så kan ni få information om gamla datatyper och liknande
- Så viktigt att veta vilka som är relevanta



Datatyper

- **`std_logic`** är datatypen som används för att representera bitar
- Kan anta följande värden: `01uxz-wlh`
  - `'U'`: Uninitialized
    - I simulering innan annat värde antagits
  - `'X'`: Forcing Unknown
    - I simulering när flera utgångar försöker driva signalen

Datatyper

- `'0'`: Forcing 0
- `'1'`: Forcing 1
- `'Z'`: High impedance
  - Tri-state
- `'-'`: Don't care
  - Precis lika smidigt som när ni gör Karnuaghdiagram
- `'W'`: Weak Unknown, `'L'`: Weak 0, `'H'`: Weak 1
  - Inget vi behöver bry oss om i denna kursen

Datatyper

- **`std_logic_vector`** används för vektorer/arrayer av **`std_logic`**
- Storleken anges efter, t ex
  - **`std_logic_vector(0 to 2)`**
  - **`std_logic_vector(5 downto 2)`**
- För **`std_logic`** och **`std_logic_vector`** måste paketet **`std_logic_1164`** importeras från biblioteket **`ieee`**
  - **`library ieee;`**
  - **`use ieee.standard_logic_1164.all;`**

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 eller b ändras

Än så länge är ordningen mellan satserna oviktig
”Programmera” aldrig i VHDL!
Tänk hårdvara => beskriv hårdvaran med VHDL

Två grundläggande typer av satser

- Sekventiella satser: **`process`**
- Parallella satser: allt annat
- Alla sekventiella satser är parallella gentemot varandra och alla parallella satser

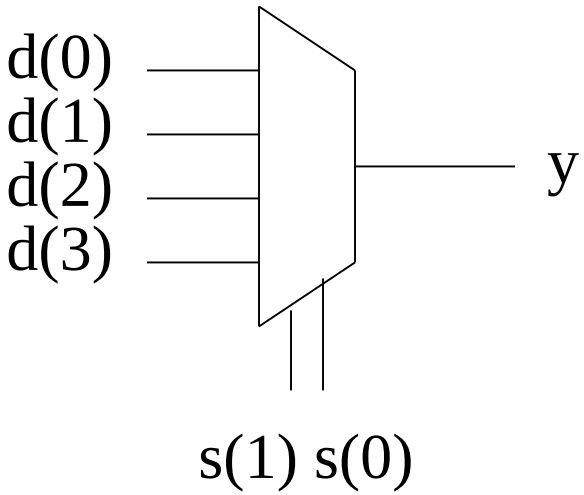
VHDL för kombinatoriska kretsar

Kombinationskretsar implementeras med

- signaltilldelning med Booleska uttryck
  - `c <= a and b;`
- **`with-select-when`** är en mux (använt för minne i lab 2).
- **`when-else`** är en generaliserad mux.

En multiplexer

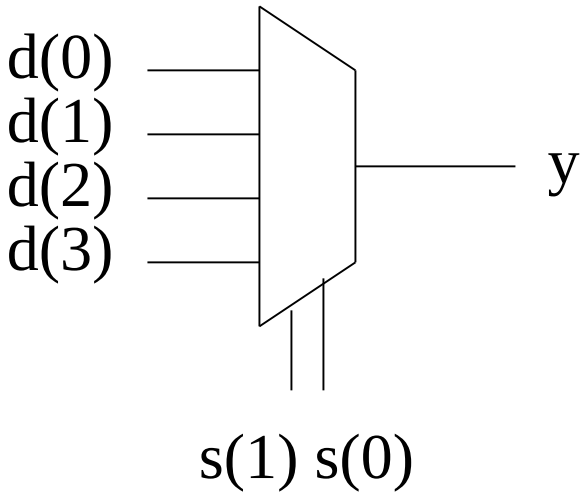
```
entity mux is
  port(  d: in std_logic_vector(0 to 3);
         s: in std_logic_vector (1 downto 0);
         y: out std_logic);
end entity mux;
```



Multiplexern, forts

VHDL har en sats 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 satsen.
- enn rad är sann

with-select-when

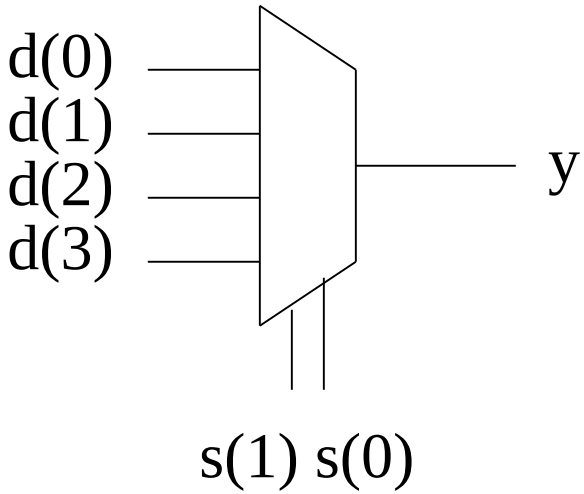
- Är en parallell sats
- Kan ej användas i en sekventiell sats (**process**)

```
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, (ej i **process)**

```
signal <= uttryck 1 when villkor 1 else
       uttryck 2 when villkor 2 else
       ...
       uttryck n-1 when villkor n-1 else
       uttryck n;
```

- Lägg märke till:
  - Det finns enn tilldelning (<=) i satsen
  - Noll eller flera villkor är sanna $s = u_1v_1 + u_2v_1´v_2 + \ldots + u_nv_1´\ldots v_{n-2}´v_{n-1}´$
  - Första sanna villkoret bestämmer
- Både **with-select-when** och **when-else** kan uttrycka vilken Boolesk funktion som helst!

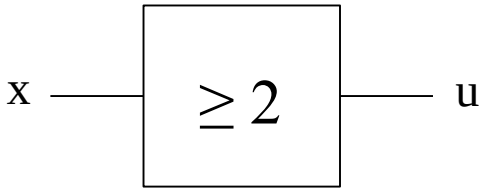
Exempel

u = 1 omm x = (x(2), x(1), x(0)) innehåller 2 eller 3 ettor.

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

-- kan skrivas så här
with x select
    u   <= '1' when "011", -- 3
           '1' when "101", -- 5
           '1' when "110", -- 6
           '1' when "111", -- 7
           '0' when others;

-- eller så här (korrekt operatorprecedens)
u <= x(0) and x(1) or x(0) and x(2) or x(1)
and x(2);
```



Vad har vi så långt?

- **entity** beskriver gränssnittet
- **architecture** beskriver innehållet
- Mellan **begin** och **end** har vi parallella satser.
  - Tilldelning med Booleska uttryck **c <= a and b;**
  - **with-select-when** är en mux
  - **when-else** är en generaliserad mux
  - Allt sker parallellt

VHDL för sekvenskretsar

- process-satsen
  - if-then-else Endast inuti process-sats!

(Fler nästa föreläsning)

process

process-satsen exekveras sekventiellt. Här exempel på 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 signaler i *känslighetslistan* ändras, i detta fallet `clk`

`q` uppdateras på positiv `clk`-flank

Det gamla värdet på `q` ligger kvar om ett nytt ej specas.

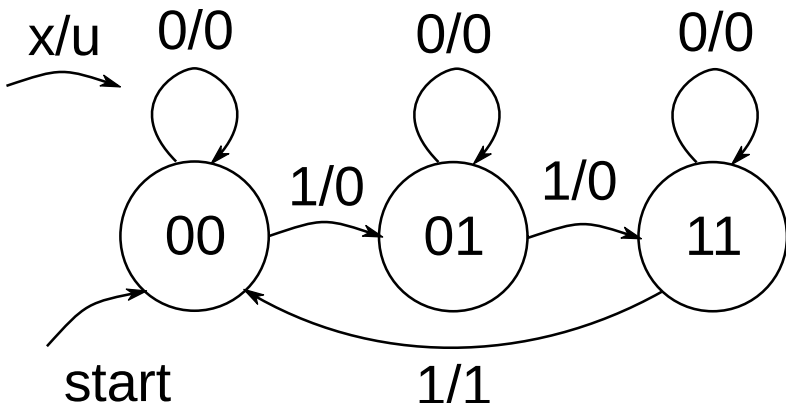
if-then-else

```
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**
- Observera stavning på **elsif**

Exempel

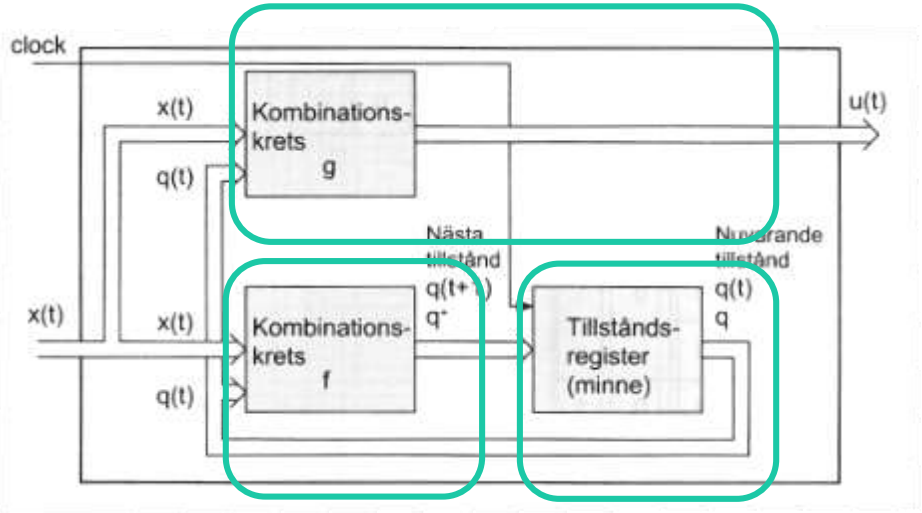
Bygg en sekvenskrets som ger utsignalen 1 i samma klockintervall som var tredje 1:a.

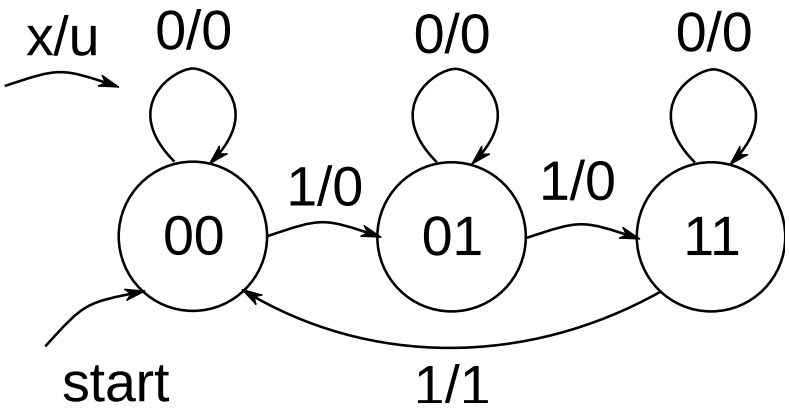


VHDL-beskrivning som liknar grafen.

Sekvenskretsar

Utsignal
Gör kombinatorik av detta





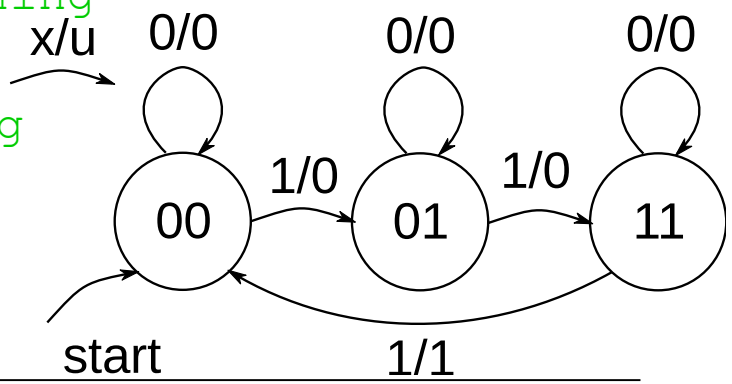
Tillståndsberäkning
Gör kombinatorik av detta.

Tillståndsuppdatering
Gör en `process(clk)` av detta.

Sekvenskrets

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

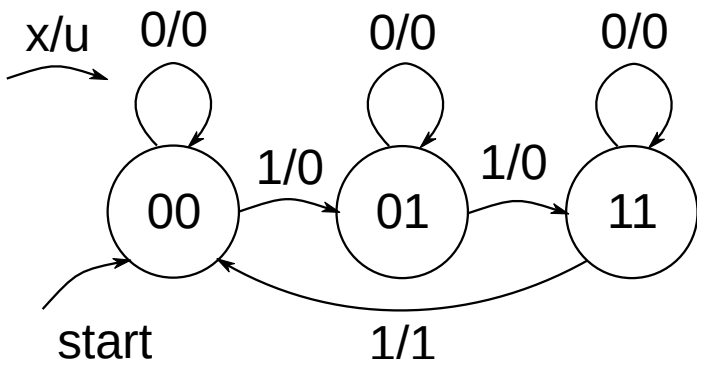
architecture graf of skrets is
  signal q, qplus: std_logic_vector(1 downto 0);
begin
  -- q+ = f(q,x): tillståndsberäkning
  -- u = g(q,x):  utsignal
  -- q = q+:  tillståndsuppdatering
end graf;
```



Eller i annan ordning

Tillståndsuppdatering – sekventiell sats

```
-- q = q+: tillståndsuppdatering
process(clk)
begin
 if rising_edge(clk)then
  q <= qplus
 end if;
end process;
```

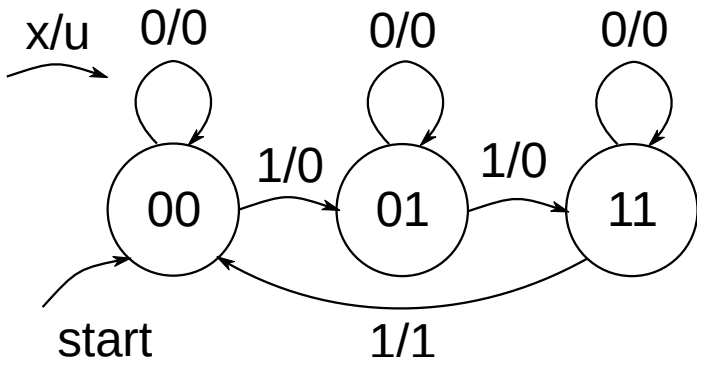


Tillståndsberäkning – parallell sats

```
-- q+ = f(q,x): tillståndsberäkning
qplus <= "00" when q = "00" and x = '0' else
         "01" when q = "00" and x = '1' else
         "01" when q = "01" and x = '0' else
         "11" when q = "01" and x = '1' else
         "11" when q = "11" and x = '0' else
         "00";
```

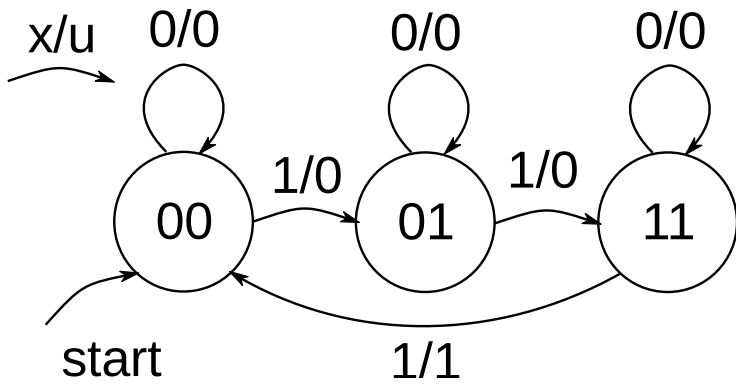
Alternativt

```
-- q+ = f(q,x): tillståndsberäkning
qx <= q & x;   -- Behöver definieras
with qx select
qplus <= "00" when "000",
         "01" when "001",
         "01" when "010",
         "11" when "011",
         "11" when "110",
         "00" when others;
```



Utsignal – parallell sats

- Parallell sats




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

-- alternativt
u <= '1' when q(1) = '1' and
    q(0) = '1' and x = '1' else '0';
-- Förenklat
u <= q(1) and q(0) and x;
```

Komplett kod

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

architecture graf of skrets is
  signal q, qplus: std_logic_vector(1 downto 0);
  signal qx      : std_logic_vector(2 downto 0);
begin
  -- q+ = q: tillståndsuppdatering
  process(clk)
  begin
    if rising_edge(clk) then
      q <= qplus;
    end if;
  end process;
  -- q+ = f(q,x): tillståndsberäkning
  qx <= q & x; -- konkatenering
  with qx select
  qplus <= "00" when "000",
           "01" when "001",
           "01" when "010",
           "11" when "011",
           "11" when "110",
           "00" when others;
  -- u = g(q,x) : utsignal
  u <= q(1) and q(0) and x;
end graf;
```

VHDL beskriver hårdvara!

1. En VHDL-modul består av två delar
   a) `entity`, som beskriver gränssnittet
   b) `architecture`, som beskriver innehållet

2. För att göra kombinatorik används
   a) Booleska satser: `z <= x and y;`
   b) `with-select-when`-satser
   c) `when-else`-satser

   } Parallella satser

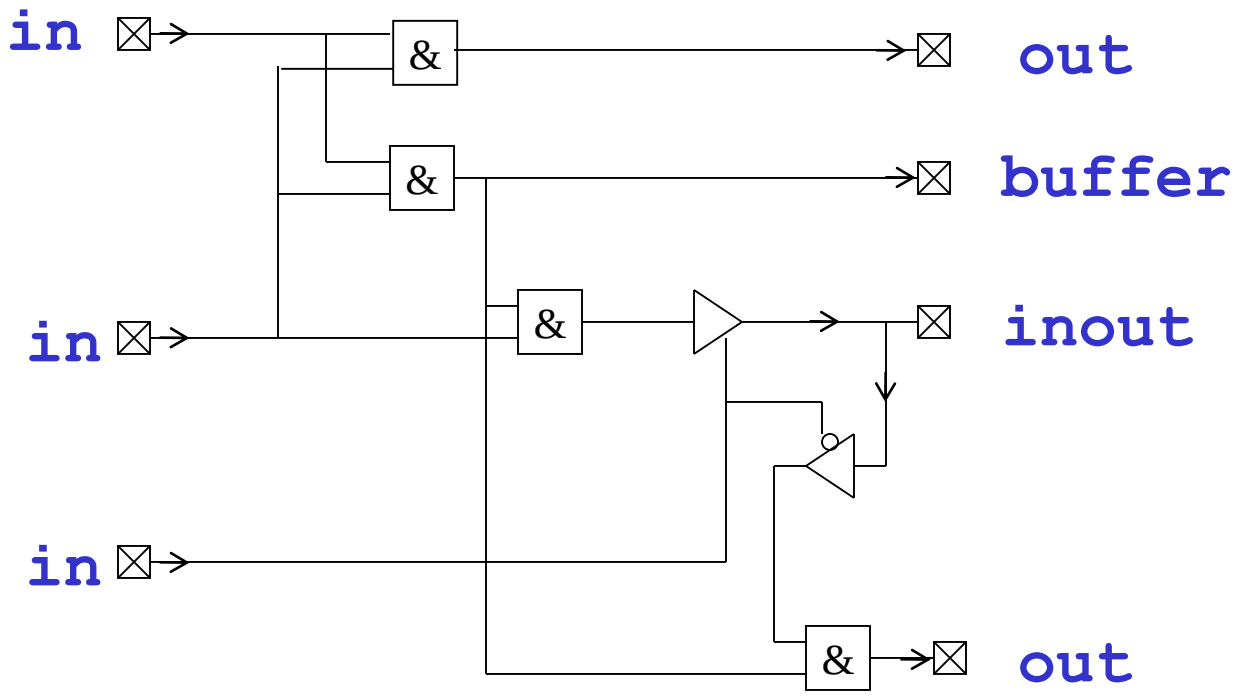
3. För att göra sekvensnät används (en eller flera) `process(clk)`-satser
   a) enn `if rising_edge(clk) … end if;`
   b) Booleska satser: `z <= x and y;`
   c) `if-then-else`-satser

   } VL får värdet när processen avslutas

4. Parallella och sekventiella satser exekveras parallellt gentemot varandra

In/utsignaler

OBS, om en utsignal också används inuti nätet, så ska den deklareras som **`buffer`**.



In/utsignaler

- Oftast bättre att undika buffer och ha en extra intern signal
- ```
  entity example is
  port (a, b : in std_logic;
        c, d : out std_logic);
  end entity example;

  architecture basic of example is
  signal cint : std_logic;
  begin
       cint <= a and b;
       d <= cint xor a;
       c <= cint;
  end architecture basic;
  ```
  ```

Numeric_std

```
use ieee.numerc_std.all -- lägger till paket

signal q: unsigned(3 downto 0);  -- 4 bit ctr
…
q <= q + 1;
if q = 10 ...
    q <= ”0011”;
    q(0) <= ’1’;
```

Notera hur värdet av heltal, bitvektor samt bit anges

Dvs vi kan hantera `q` både som en boolesk vektor och som ett tal på intervallet [0,15];

`unsigned` och `signed` är “`std_logic_vector` som går att räkna med”

4-bits dekadräknare med synkron clear

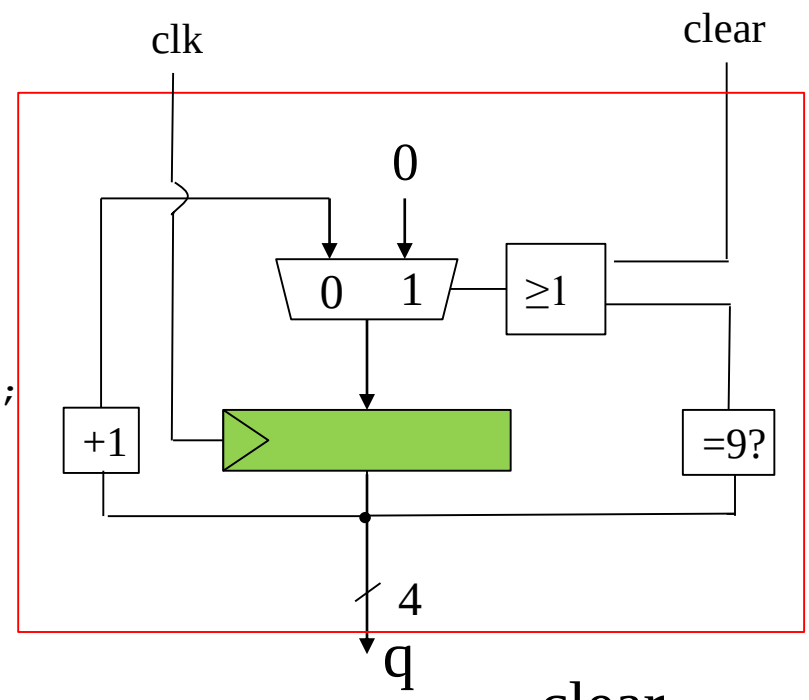

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

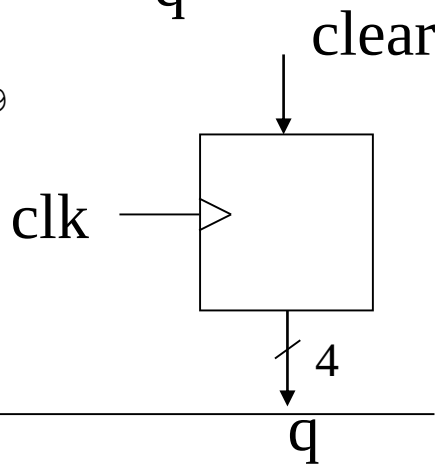
entity counter is
port(clk, clear: in std_logic;
     q: out std_logic_vector(3 downto 0));
end counter;

architecture simple of counter is
    signal q_int, q_p: unsigned (3 downto 0));
begin
  process(clk)
  begin
    if rising_edge(clk) then
        q_int <= q_p;
    end if;
  end process;

  q_p <= to_unsigned(0, 4) when clear = '1' or q_int = 9
         else q_int+1; -- to_unsigned(0, 4) = "0000"

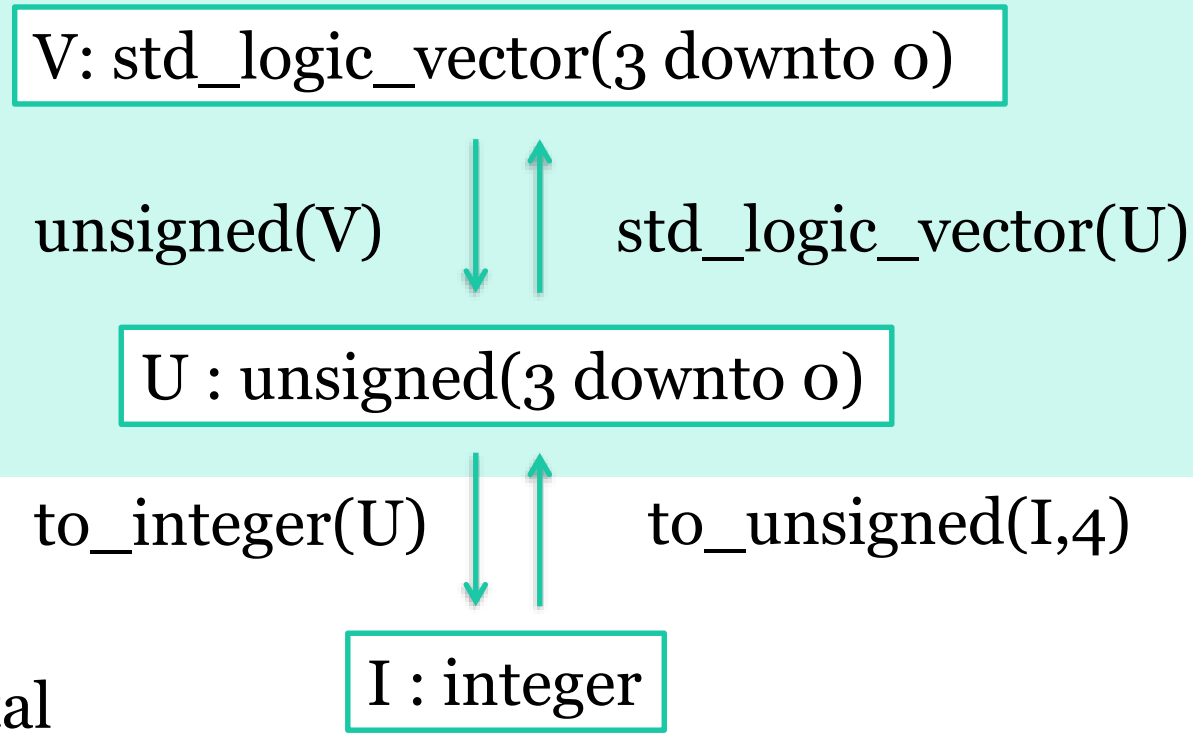
  q <= std_logic_vector(q_int); -- Hårt typat som sagt
end simple;
```





Typkonvertering

Bitvektorer med ett specificerat antal bitar



Konkatenering

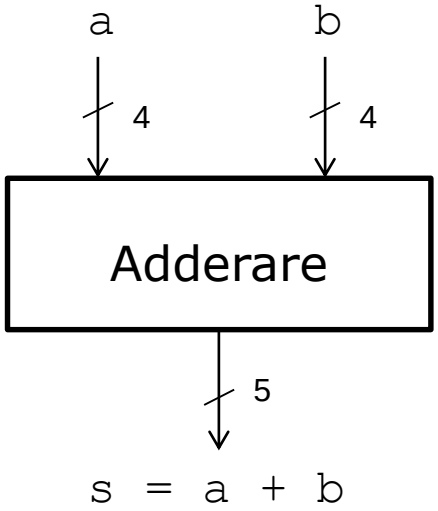
```
signal bus: std_logic_vector(1 downto 0);
signal a, b: std_logic;
bus <= a & b;
```

4-bitsadderare

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use NUMERIC_STD.ALL;

entity adder is
port(a, b: in UNSIGNED(3 downto 0);
     s: out UNSIGNED(4 downto 0));
end adder;

architecture simple of adder is
begin
-- zero extension
  s <= ('0'& a) + ('0'& b);
-- alternativt
  s <= resize(a, 5) + resize(b, 5);
end simple;
```



Asynkron reset

```
process(clk, reset)
begin
 if reset = '1' then
  q <= '0';
 elsif rising_edge(clk) then
  q <= x;
 end if;
end process;
```

Synkron reset

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

Känslighetslistan (ändringar på signaler exekverar processen):

- `clk` ska alltid vara med
- Ev. asynkrona insignaler till vippa/räknare/register
- Andra signaler ska ej vara med.

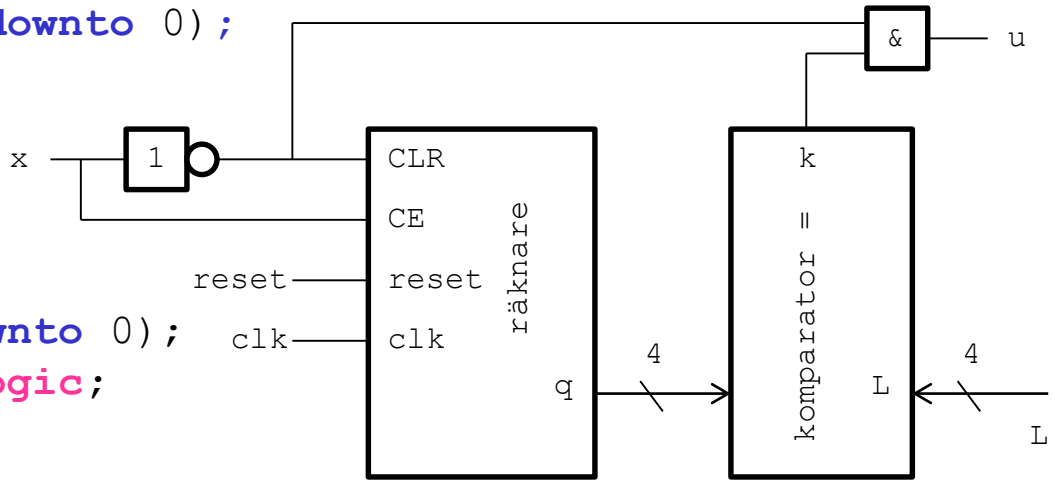
Logiskt blockschema => VHDL


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

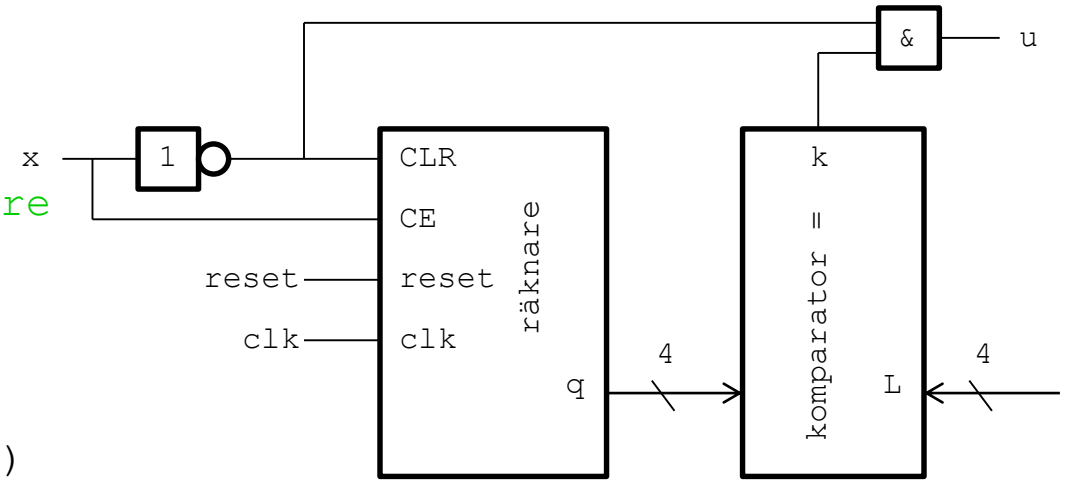
entity pulsdetektor is
  port( clk, x : in  std_logic;
        reset : in  std_logic;
        L : in unsigned(3 downto 0);
        u : out std_logic);
end pulsdetektor;

architecture Behavioral of
pulsdetektor is
  signal q : unsigned(3 downto 0);
  signal CLR, CE, k: std_logic;
begin
  -- hela vår konstruktion

end Behavioral;
```



Räknaren



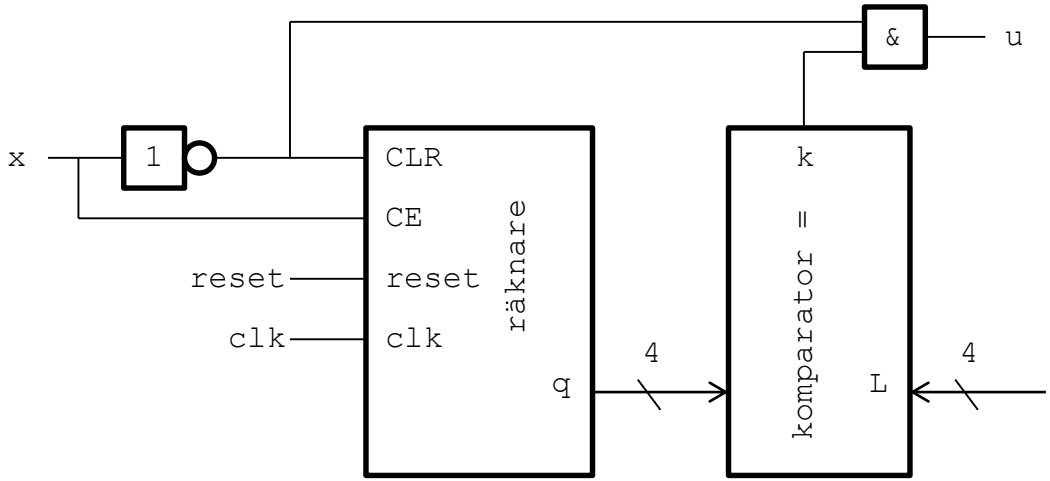

```
-- insignaler till räknare
CE <= x;
CLR <= not x;

-- räknare
ctr16: process(clk,reset)
begin
    if reset = '1' then        -- asynkron reset
       q <= "0000";
    elsif rising_edge(clk) then
       q <= qplus;
end if;
 end process ctr16;
-- synkron clear, count enable
qplus <= "0000" when CLR = '1' else q + 1 when CE = '1' else q;
-- fyra bitar, så 15 + 1 = 0
```

Komparator och utsignal

```
-- komparator
k <= '1' when L = q else '0';

-- utsignal
u <= CLR and k;
```



Blockschema ->VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pulsdetektor is
  ...
end pulsdetektor;

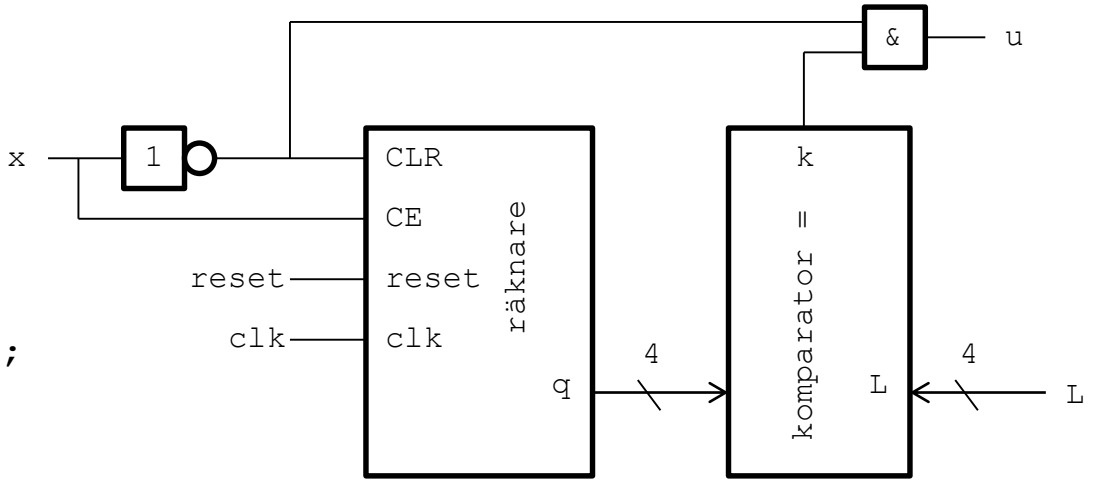
architecture Behavioral of pulsdetektor
is
  ...
begin
  -- insignaler till räknare
  CE <= x;
  CLR <= not x;

  -- räknare
  ctr16: process(clk,reset)
    ...
  end process ctr16;
  qplus <= ...

  -- komparator
  k <= '1' when L = q else '0';

  -- utsignal
  u <= CLR and k;
end Behavioral;
```

- Varje block har motsvarande kod.
- Överensstämmande signalnamn i blockschema och kod.



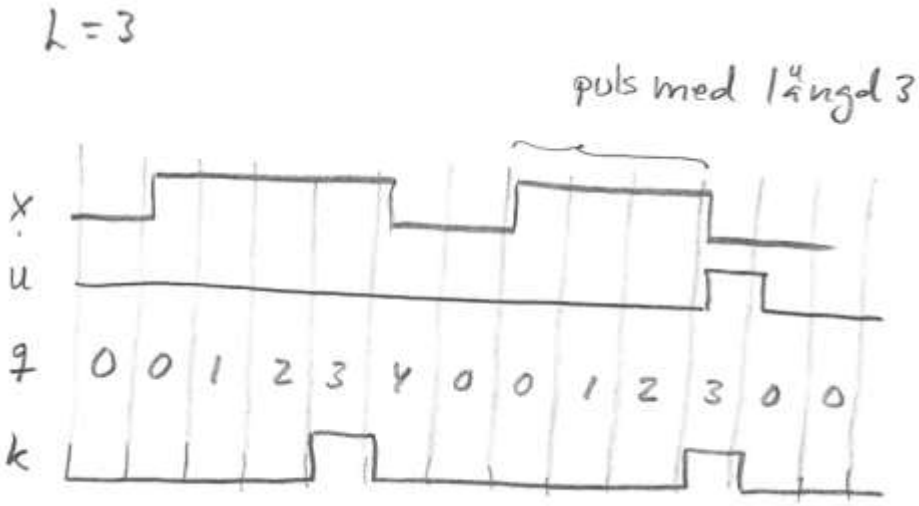
Simulering i ModelSim

Utdrag ur fil för att definiera insignaler och köra en simulering.

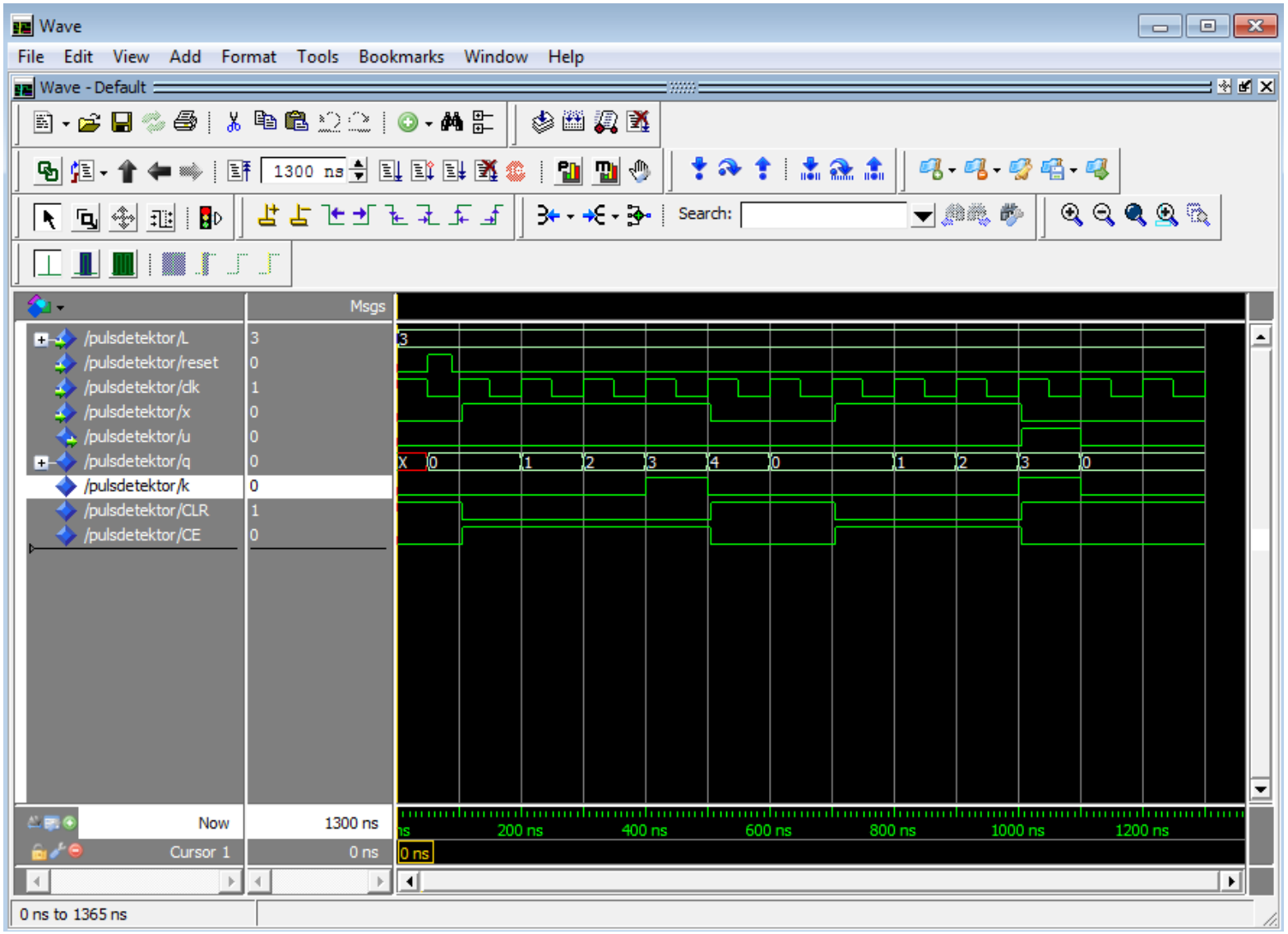
```
# sim.do
...
# clk Periodtid 100 ns
force -freeze sim:/pulsdetektor/clk 1 0, 0 50 -r 100
# reset
force -freeze sim:/pulsdetektor/reset 0 0, 1 50, 0 90
# x
force -freeze sim:/pulsdetektor/x 0 0, 1 105, 0 505, 1 705, 0 1005
# L
force -freeze sim:/pulsdetektor/L 0011 0
run 1300
```

Starta simulering i transcriptfönstret

```
VSIM > do sim.do
```



ModelSim


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
1300 ns
Search:
Msgs
/pulsdetektor/L
/pulsdetektor/reset
/pulsdetektor/clk
/pulsdetektor/x
/pulsdetektor/u
/pulsdetektor/q
/pulsdetektor/k
/pulsdetektor/CLR
/pulsdetektor/CE
3
0
1
0
0
0
0
1
0
Now
1300 ns
Cursor 1
0 ns
200 ns
400 ns
600 ns
800 ns
1000 ns
1200 ns
0 ns to 1365 ns

Slutsatser

- Rita logiska blockscheman först
  - Konstruktionsprocessen är identisk med tidigare
- Samma struktur på koden som på blockschemat
  - Alltså: små process-satser som precis motsvarar ett block
  - Leder till bra koll på mängden hårdvara

Macrocells Used	Pterms Used	Registers Used	Pins Used	Function Block Inputs Used
5/36 (14%)	16/180 (9%)	4/36 (12%)	8/34 (24%)	9/72 (13%)

Använda produkttermer

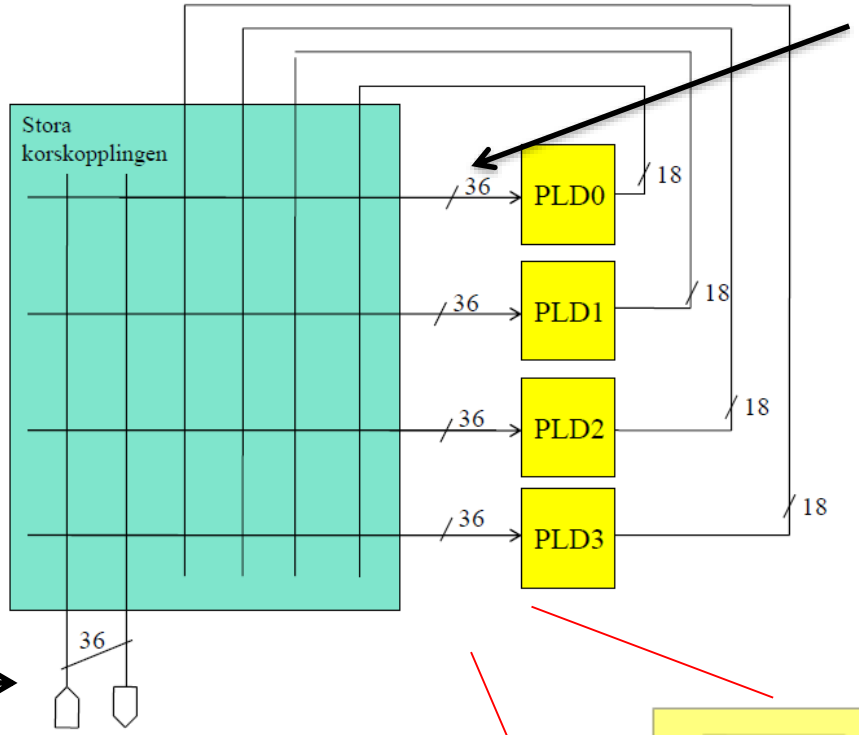
L3-L0, x, u, clk, reset

Beräkning av 4 tillstånd + 1 utsignal

4 D-vippor i 4-bitsräknaren

L3-L0, q3-q0, x

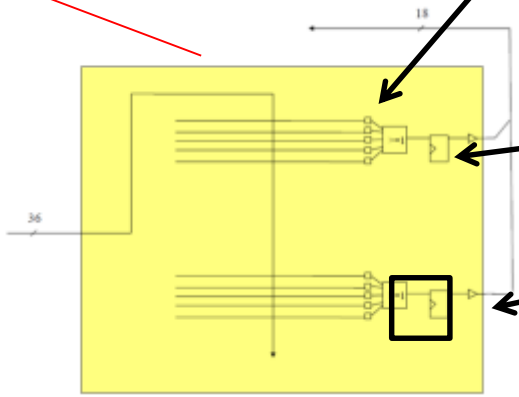
Får det plats?



functional block inputs!

produkttermer!
5 AND/vippa

pinnar!



vippor!

makro-cell!

Vad kan ta slut?

- Pinnar
- Vippor
- Produkttermer

Inför Lab 3

- Obligatorisk lektion i Xilinx ISE (i morgon förmiddag)
- Fyra uppgifter: enkel prova-på-uppgift, kodlåset i VHDL, två olika räknarrelaterade uppgifter
  - Byt möjligen plats på 3.3 c) (kodlås med PROM) och 3.3 a) (kodlås med ekvationer)
- Förberedelseuppgifter:
  - Logiska blockscheman till uppgifterna
  - Förbered så mycket kod som går
- ModelSim finns installerat på datorerna i Freja (och i Grinden).

Examination

- Vid examination ska logiskt blockschema, VHDL-kod samt korrekt fungerande krets uppvisas.
- Logiskt blockschema
  - Block ska bestå av väldefinierad hårdvara:
    - räknare, register, enpulsare, synkroniseringsvippor, sekvenskretsar (med tillståndsdiagram)
    - MUX, DMUX, adderare, komparatorer, kombinatoriska kretsar (minne, Booleska funktioner)
- VHDL-kod
  - Varje block ska motsvara ett kodavsnitt, t ex en process-sats och/eller parallella tilldelningar
  - Överensstämmande signalnamn i blockschema och kod.

Rekommendation

- Använd endast
  `std_logic` och **`std_logic_vector`**
  - `library ieee;`
    `use ieee.std_logic_1164.all;`
- Vill ni räkna inkludera `NUMERIC_STD` –biblioteket
  - `library ieee; -- behövs en gång`
    `use ieee.numeric_std.all;`
- Skriv logik och vippor separat
- Undvik **`integer`**. Går ej att indexera på bit-nivå.

Digitalteknik

Oscar Gustafsson, Mattias Krysander

www.liu.se

LINKÖPINGS UNIVERSITET