TSEA83 : Datorkonstruktion
Fö9

VHDL 2/3

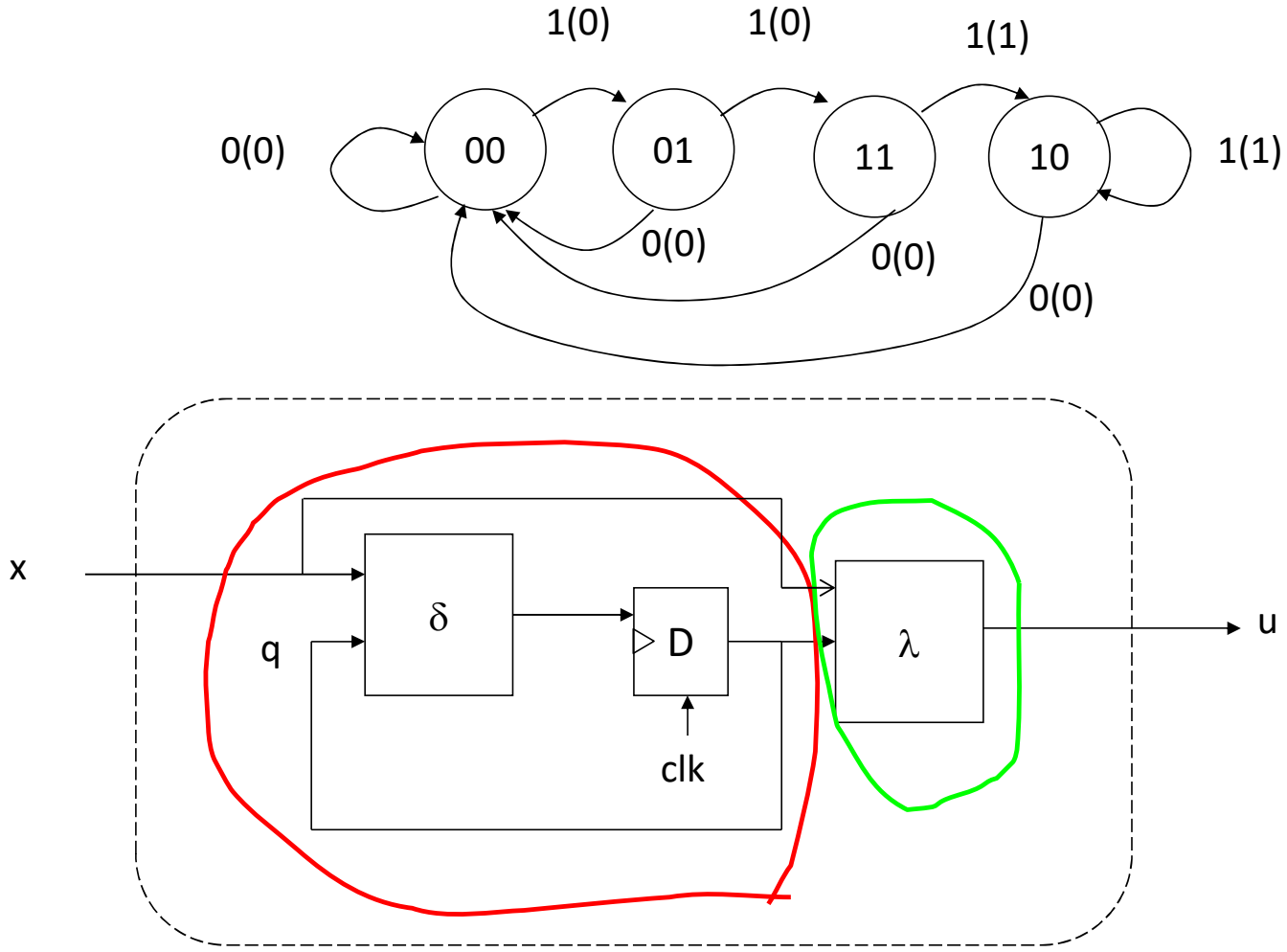
LINKÖPINGS UNIVERSITET

Fö9 : Agenda

- Kort repetition, sekvensnät
- Moder – portsatsen
- Datatyper
- Ett exempel, stegmotorstyrning
- Hierakisk konstruktion, instansiering
- Kombinatorisk process
- Record, loop
- Lab3 : UART

Repetition, sekvensnät

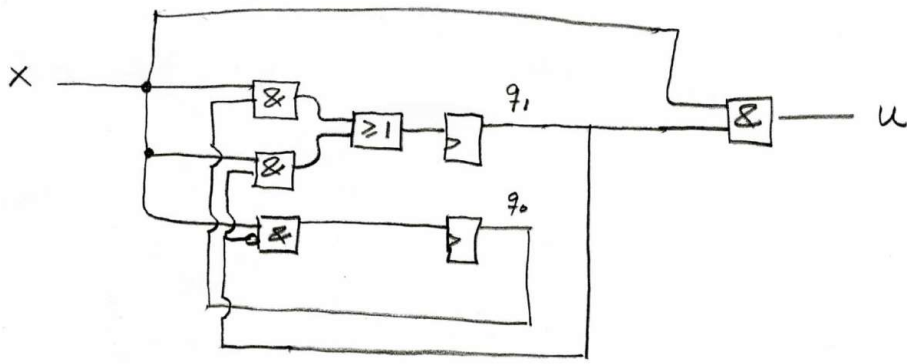
Sekvensnätexemplet



1) Gör en `process(clk)` av detta

2) Gör kombinatorik av det här!

Sekvensnätexemplet




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

architecture booleq2 of snet 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;
    -- överhoppade rader
    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 architecture;
```

VHDL beskriver hårdvara!

VHDL-kod är bygginstruktioner för hur hårdvaran ska konstrueras.

1. En VHDL-modul består av två delar
   - `entity`, som beskriver gränssnittet
   - `architecture`, som beskriver innehållet
2. För att göra **kombinatorik** används
   1. Booleska satser `z <= x and y;`
   2. `with-select-when`-satser
   3. `when-else`-satser

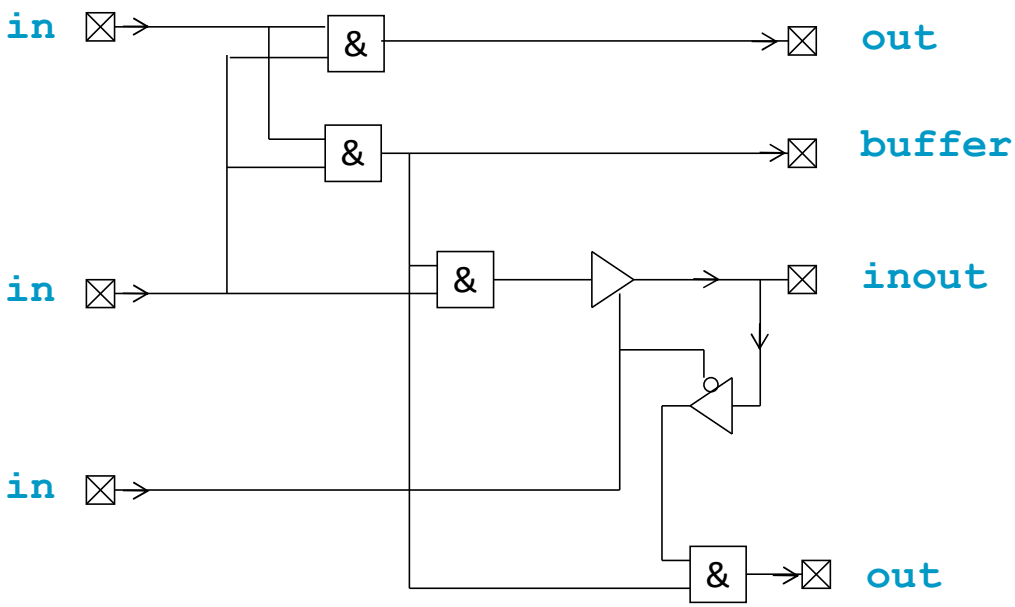
   } Samtidiga satser
3. För att göra sekvensnät används (en eller flera) `process(clk)`-satser
   1. enn `if rising_edge(clk) … end if;`
   2. booleska satser `z <= x and y;`
   3. `case-when-satser`
   4. `if-then-else-satser`

   } VL får vippa på sig
Alla klockas samtidigt

Moder : portsatsen

Moder : portsatsen

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



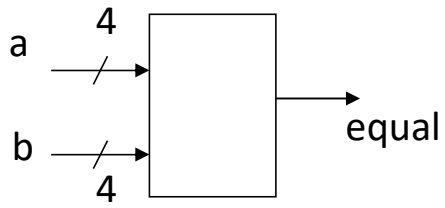
Datatyper

VHDL är ett **mycket starkt** typat språk! *Inbyggda* typer är bl a:

Typ	Möjliga värden	operatorer
integer	-2147483648 – 2147483647	**ABS** ** * / **MOD REM** + - (tecken) + - = /= < <= > >=
bit **boolean**	’0’,’1’ ’false’,’true’	**NOT** = /= < <= > >= **AND NAND OR NOR XOR XNOR**
bit_vector	Obegränsad vektor av bit	**NOT** & **SLL SRL SLA SRA ROL ROR** = /= < <= > >= **AND NAND OR NOR XOR XNOR**

Exempel : 4-bits komparator

```
entity comp4 is
  port(a,b: in BIT_VECTOR(3 downto 0);
       equal: out BIT);
end entity;


architecture func of comp4 is
begin
  equal <= '1' when a=b else '0';
end architecture;
```



`std_logic` istället för `bit`

Och `std_logic_vector` istället för `bit_vector`.

```
signal reset: std_logic;
```

Reset kan nu anta följande värden:

”vanliga” 0:an och 1:an	`'0'`: Forcing 0
	`'1'`: Forcing 1
simulering	`'U'`: Uninitialized
	`'X'`: Forcing Unknown
tristate →	`'Z'`: High impedance
ej i denna kurs	`'W'`: Weak Unknown
	`'L'`: Weak 0
	`'H'`: Weak 1
t ex för att speca sanningstabell	`'-'`: Don’t care

`std_logic` forts.

För att få tillgång till std_logic skriver man följande i början av filen

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

```
(IEEE = International Electrical & Electronics Engineers)
```

Nu får du datatyperna **`std_logic`** och **`std_logic_vector`**
och kan använda dem pss **`bit`** och **`bit_vector`**.

Man *kan* (men bör definitivt **inte**) lägga till raden

```
use IEEE.STD_LOGIC_UNSIGNED.ALL
```

Då kan man göra aritmetik med **`std_logic_vector`**

Användning av STD_LOGIC_UNSIGNED
REKOMMENDERAS INTE!

~~std_logic_unsigned~~

Nu kan du göra följande:

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

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

VHDL är ett språk som har *operator overloading*.

VARNING!

Undvik användning av:

STD_LOGIC_UNSIGNED
STD_LOGIC_ARITH
STD_LOGIC_SIGNED

Dessa bibliotek är **inte IEEE standard-bibliotek**,
och orsakar dessutom lätt problem vid typkonvertingar.

Lösningen är att istället använda:

NUMERIC_STD

NUMERIC_STD

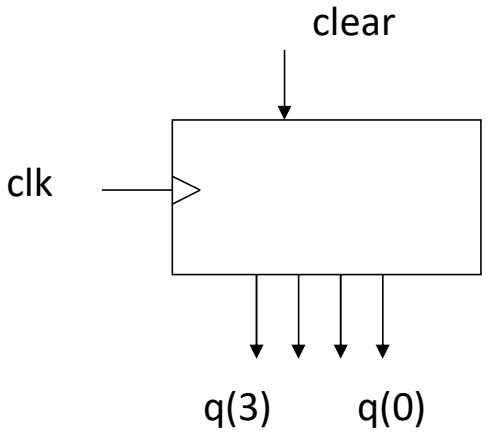
NUMERIC_STD ger datatyperna **signed** och **unsigned** samt aritmetik (och diverse funktioner) på dessa, och vi kan skriva:

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

Dvs **unsigned** är ett tal [0, 15] och kan även hanteras som en boolesk vektor.

Googla på ”why numeric_std is preferred” för mer info.

4-bits dekadräknare med synkron clear

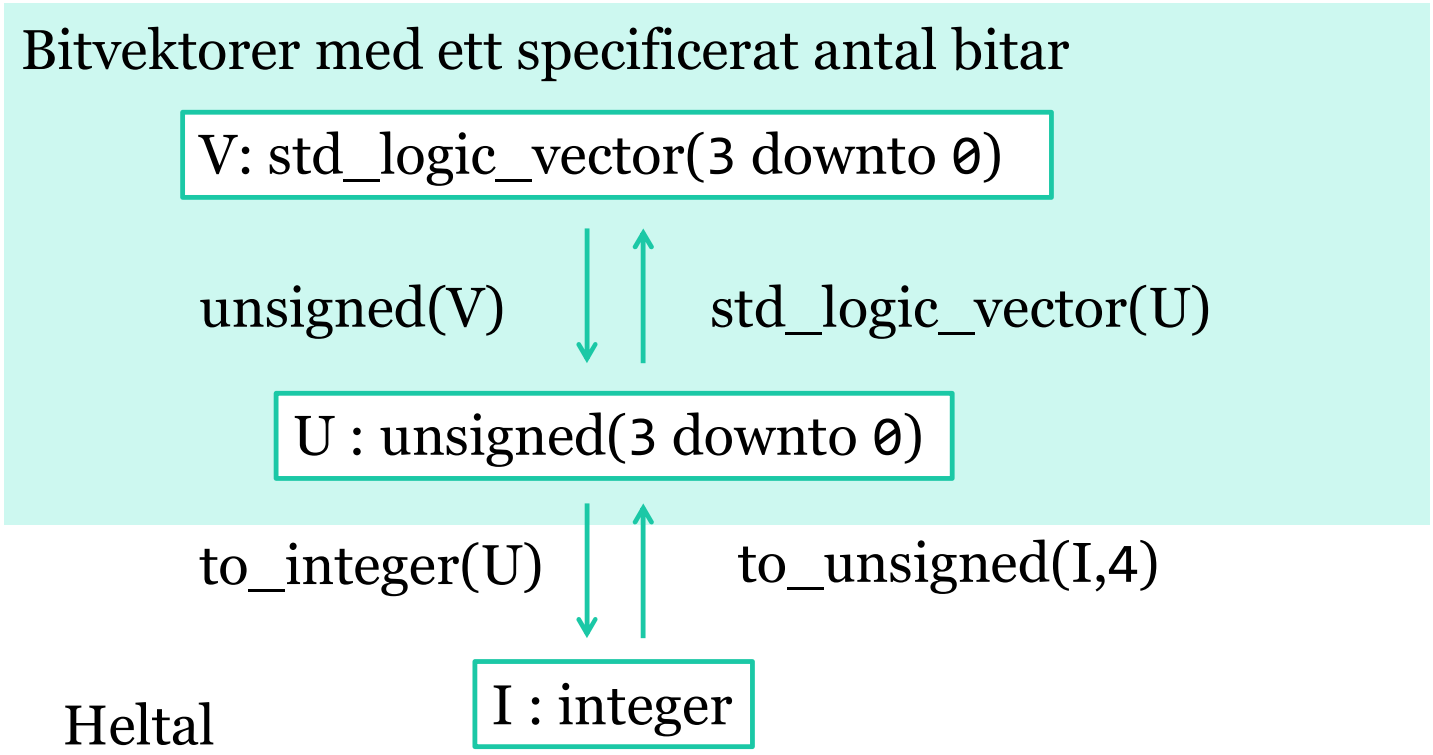



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

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

architecture func of counter is
begin
  process(clk)
  begin
    if rising_edge(clk) then
        if clear='0' then
            q <= "0000";
        elsif q=9 then
            q <= "0000";
        else
            q <= q + 1;
        end if;
    end if;
  end process;
end architecture;
```

Vi rekommenderar ang. typer

- Använd endast
  std_logic och
  std_logic_vector (använd bara som vektor)
  unsigned (använd som vektor + aritmetik)
- Vill ni räkna inkludera alltså
  NUMERIC_STD
- Skippa **integer**. Går ej att indexera på bit-nivå.
- Skippa **bit**. Tristate och don’t care saknas.

Typkonvertering



Uppräknade datatyper

Vi kan skapa egna datatyper. Det kan vara händigt vid konstruktion av sekvensnät med symboliska namn på tillstånden:

```
type state is   (odd,even);
...
signal q: state;
...
-- ett delta-nät
case q is
  when odd => if x='1' then q <= even;
                       else q <= odd;
              end if;
  when even => if x='1' then q <= odd;
                        else q <= even;
               end if;
end case;
```

Några praktiska småsaker

Konstanter

```
signal bus: unsigned(3 downto 0);
constant max: unsigned(3 downto 0) := ”1111”;
...
if bus = max then ...
```

Alias

```
signal address: unsigned(31 downto 0);
alias top_ad: unsigned (3 downto 0)
                        is address(31 downto 28);
```

Concatenation

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

En adderare med konkatenering

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


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


architecture func of adder is
begin

-- zero extension
   s <= ('0'& a) + ('0'& b);

-- s(4) är Carry-biten
-- s(3 downto 0) är summan
end architecture;
```

Asynkron reset / Synkron reset?

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

Känsligt för störningar på rst.
T ex vid hazard, dvs korta spikar innan signalvärdet stabiliserats mellan klockflanker.

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

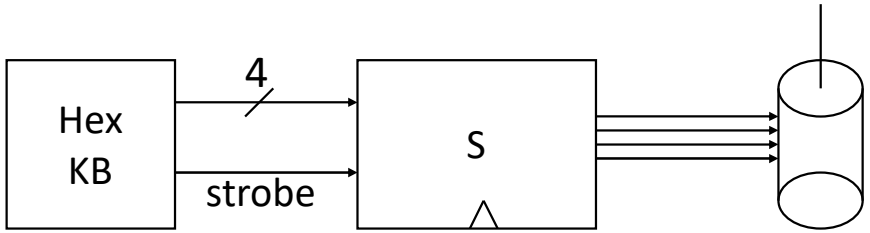
Synkron reset är säkrare.

Stegmotorstyrning

Ett lite större exempel

Exempel : Stegmotorstyrning

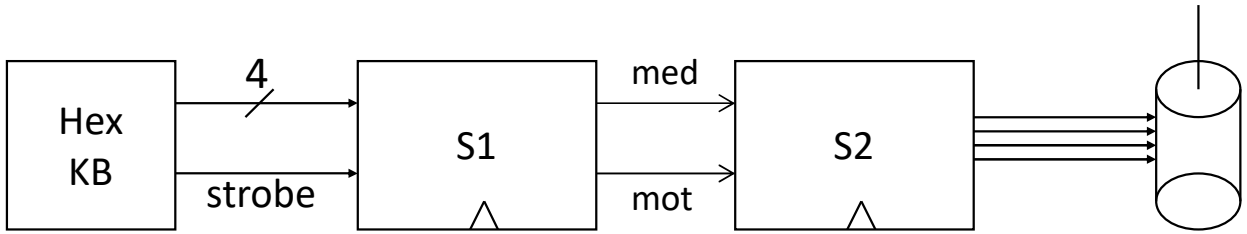
(ur Lennart Bengtsson: *Digitala system*)



- För varje ändring enligt :
  0001->1001->1000->1010->0010->0110->0100->0101->
  rör sig motorn ett steg medurs.

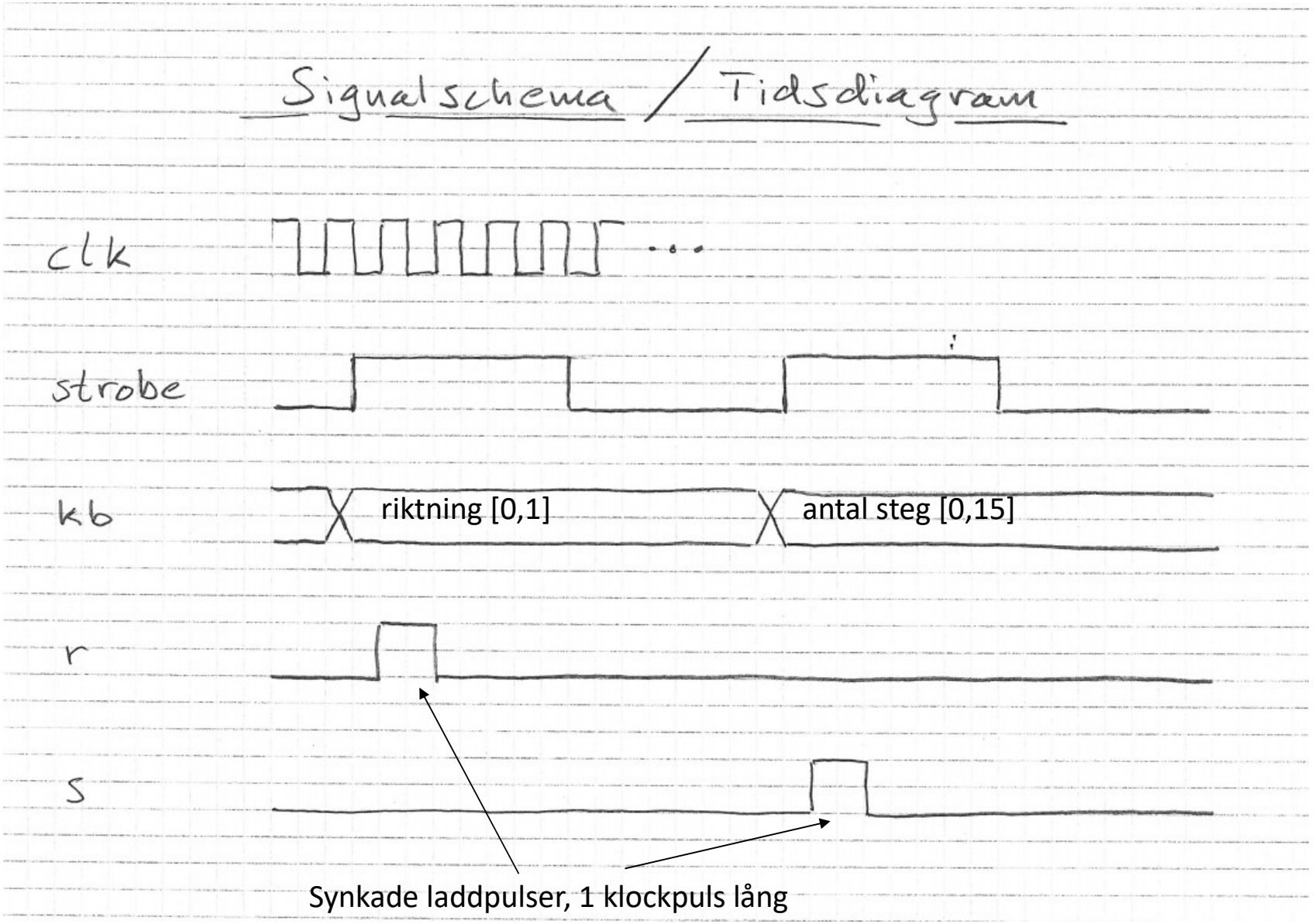
- Första knapptryckningen anger riktning
  (0=medurs,1=moturs),
  andra tryckningen anger antal steg (0-15).

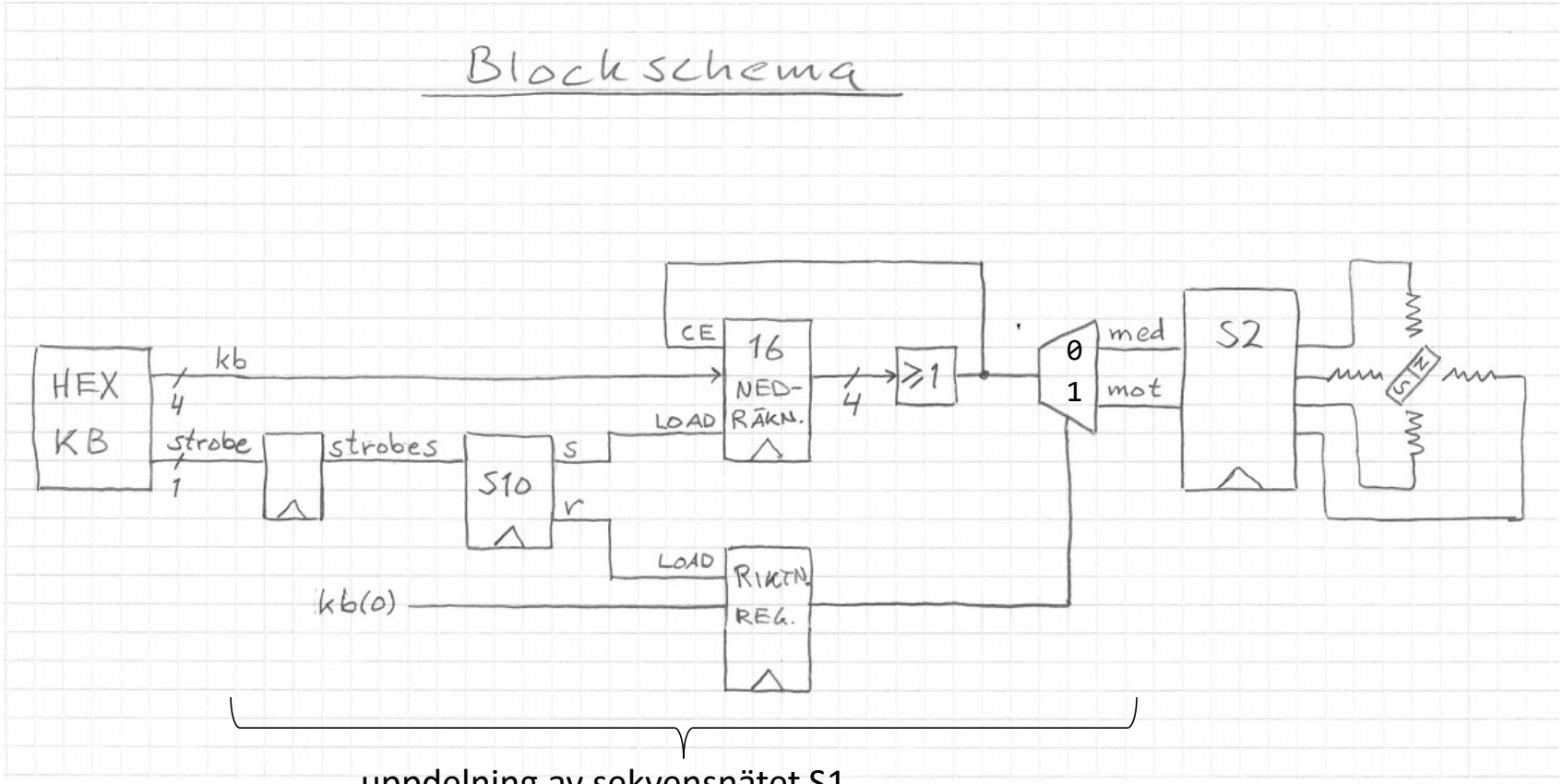
Dela upp nätet



Lyssna på tangentbordet

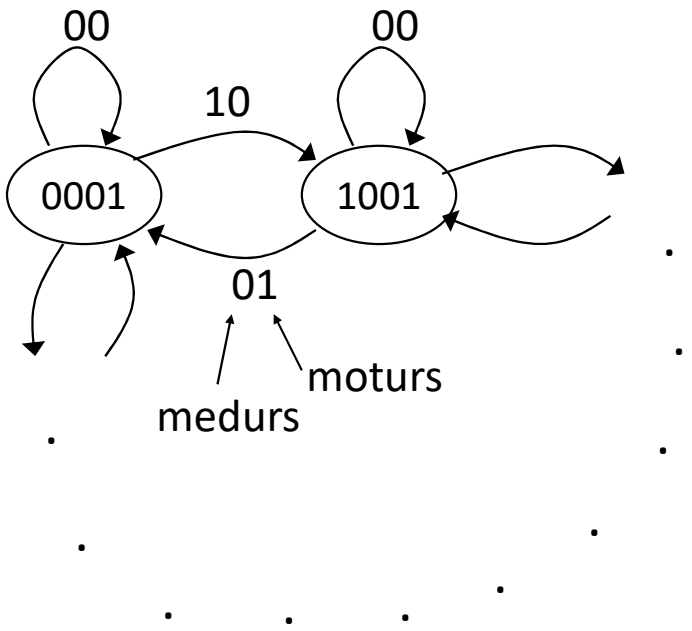
Styr stegmotorn


Signalschema / Tidsdiagram
clk
strobe
kb
riktning [0,1]
antal steg [0,15]
r
s
Synkade laddpulser, 1 klockpuls lång

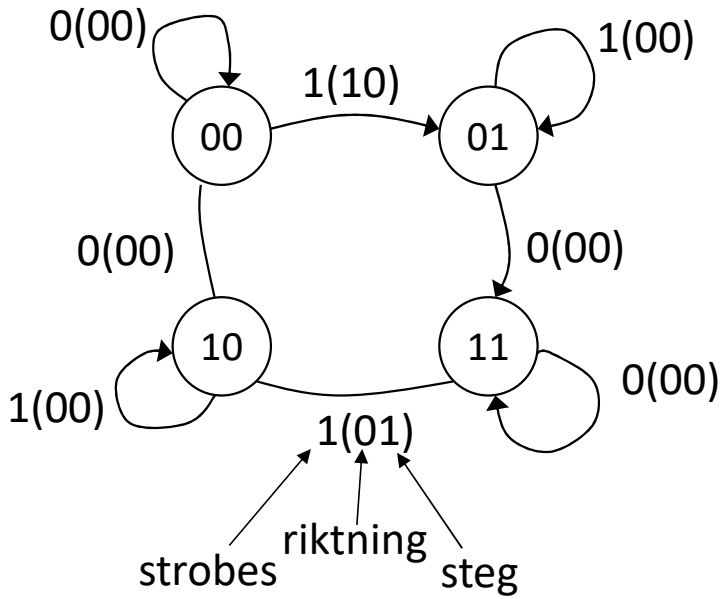


uppdelning av sekvensnätet S1

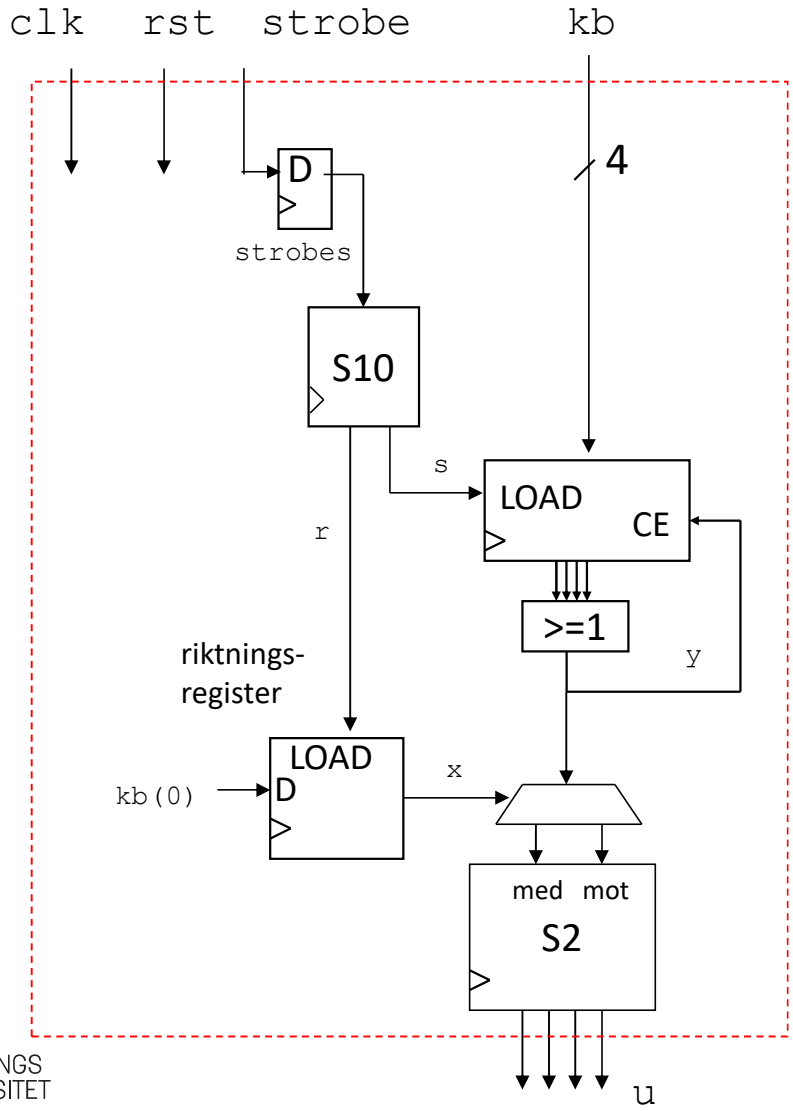
Nu kan vi konstruera
S2 direkt. Utgångarna
ska vara hasardfria,
så vi gör en Mooremaskin
med U=Q.



Strobeseparatorn S10

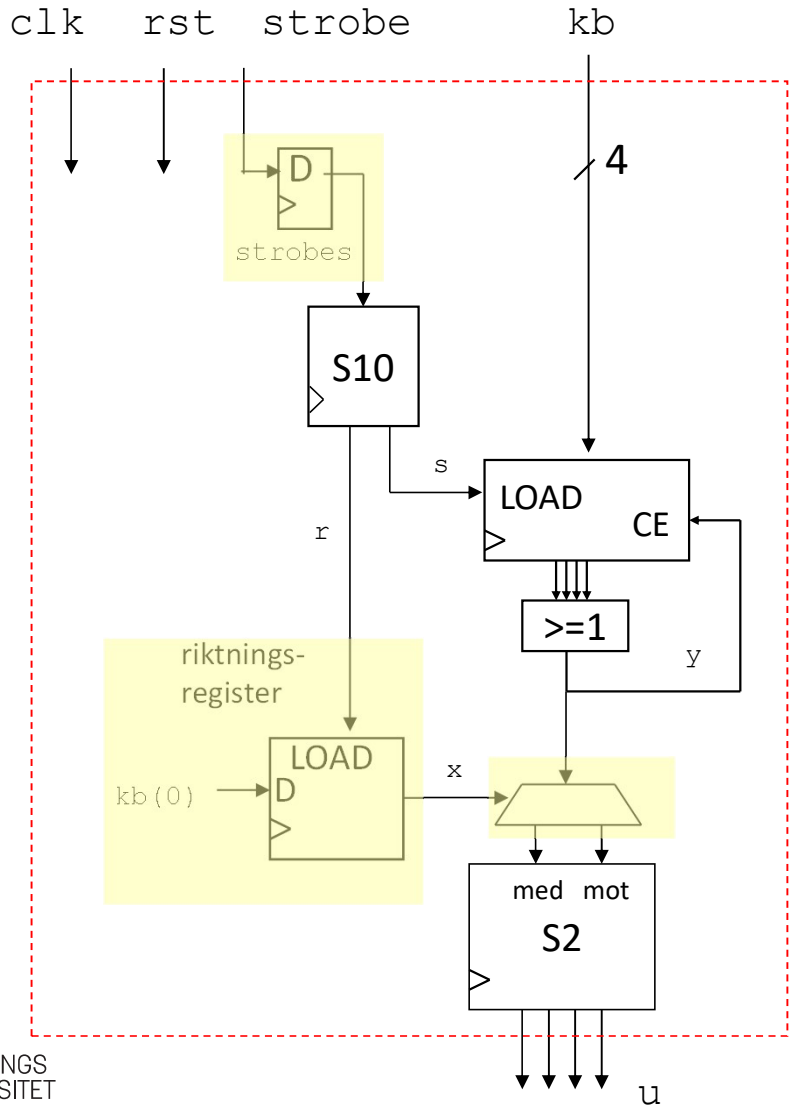


Blockschema => VHDL




```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity stegmotor is
  port(kb: in unsigned(3 downto 0);
       clk,strobe,rst: in std_logic;
       u: out unsigned(3 downto 0));
end entity;


architecture func of stegmotor is
  signal s,r,x,y,strobes: std_logic;
  signal s10: unsigned (1 downto 0);
  signal q: unsigned(3 downto 0);
  signal: med,mot: std_logic;
begin

-- hela vår konstruktion

end architecure;
```

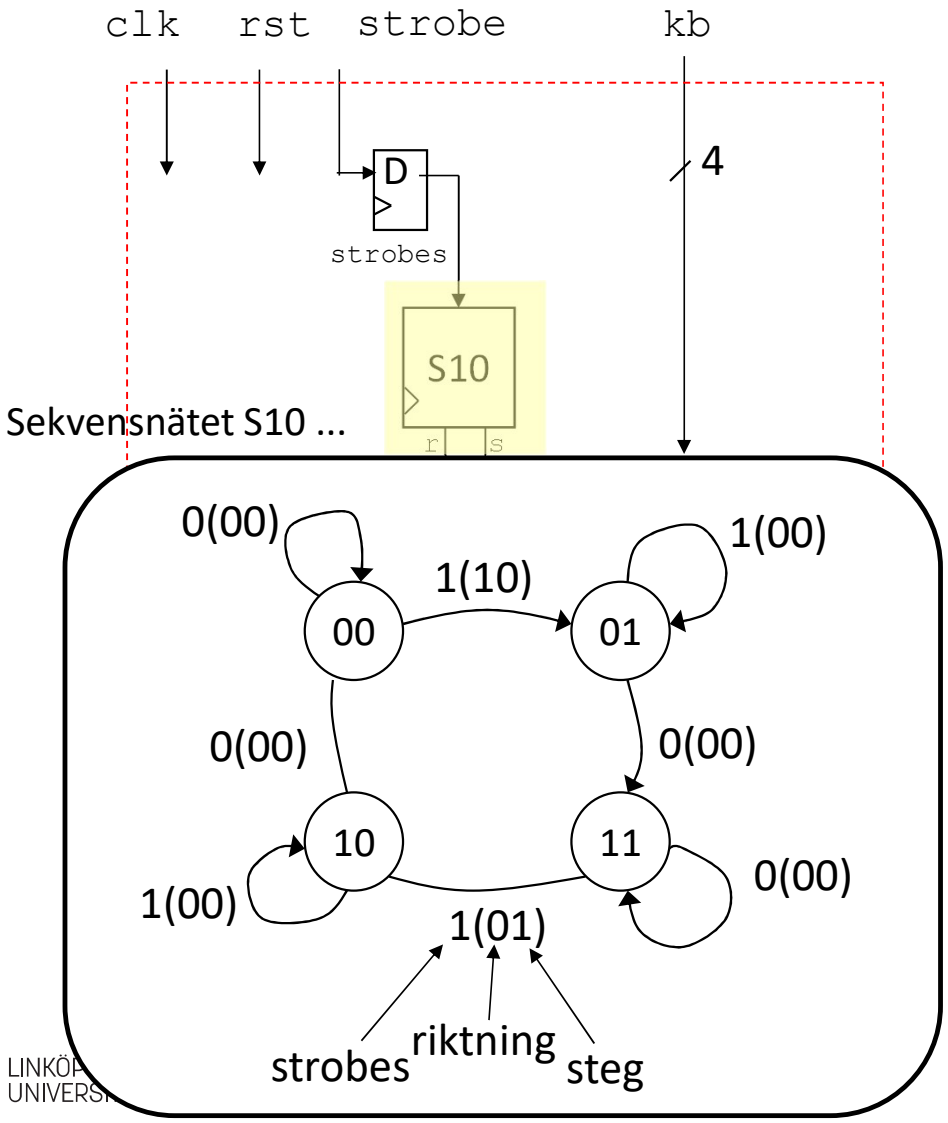
Blockschema => VHDL



Vi börjar med dom små blocken...

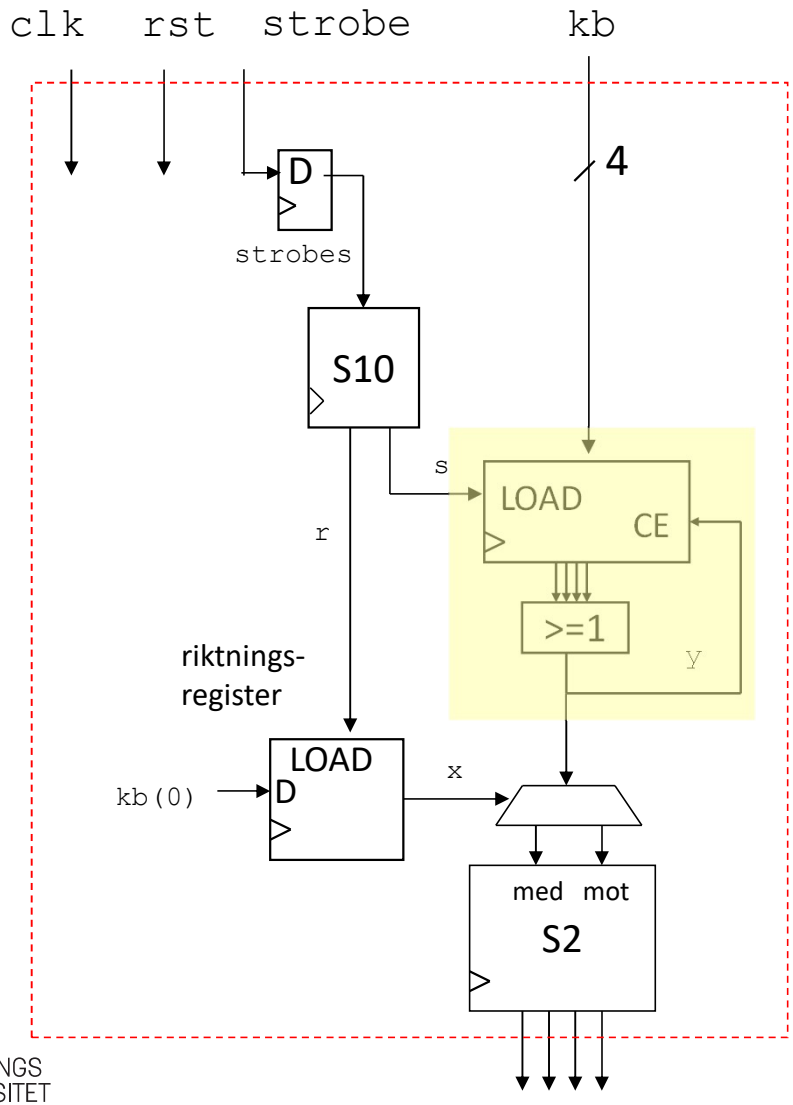
```
-- synkvippa
  sync: process(clk)
  begin
    if rising_edge(clk) then
      strobes <= strobe;
    end if;
  end process sync;


  -- riktningsregister
  riktning: process(clk)
  begin
    if rising_edge(clk) then
      if rst='1' then
        x <= '0';
      elsif r = '1' then
        x <= kb(0);
      end if;
    end if;
  end process riktning;

-- motorstyrning
  med <= not x and y;
  mot <= x and y;
```

Blockschema => VHDL




```
-- strobeseparator S10
  kbfix: process(clk)
  begin
    if rising_edge(clk) then
    case s10 is
      when "00"  => if strobes='1' then
                      s10 <= "01";
                    end if;
      when "01"  => if strobes='0' then
                      s10 <= "11";
                    end if;
      when "11"  => if strobes='1' then
                      s10 <= "10";
                    end if;
      when "10"  => if strobes='0' then
                      s10 <= "00";
                    end if;
      when others => null;
    end case;
    end if;

    if rst='1' then
      s10 <= "00";
    end if;
  end process kbfix;


-- ut från S10
r <= '1' when (s10=0 and strobes='1') else '0';
s <= '1' when (s10=3 and strobes='1') else '0';
```

Blockschema => VHDL

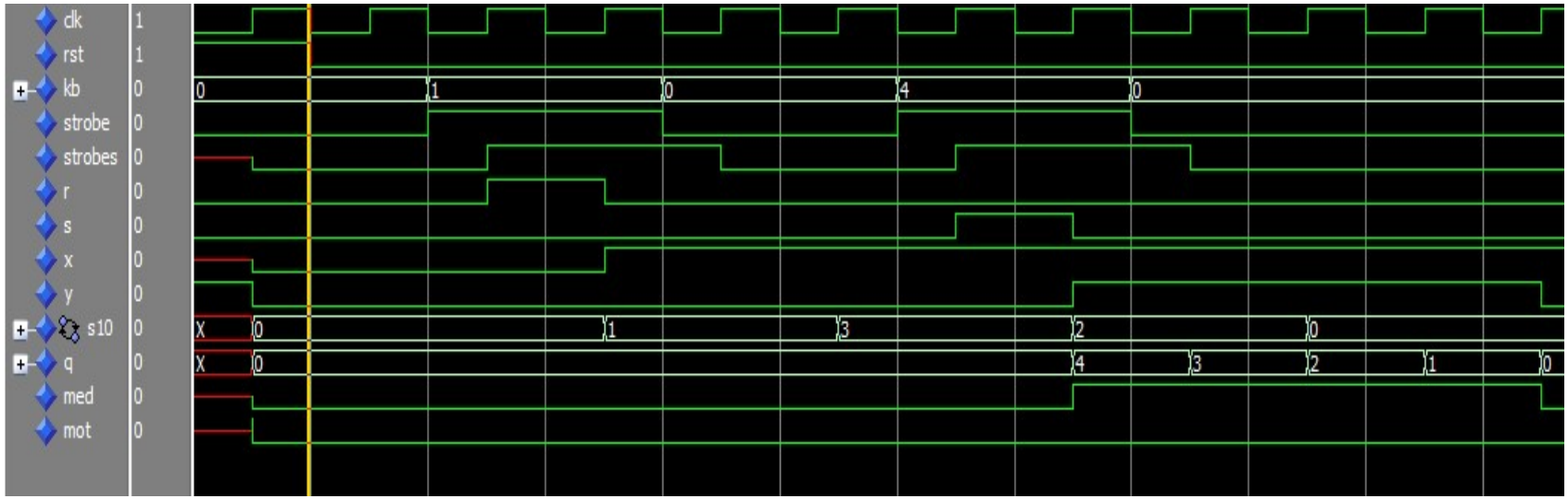


Räkneverket ...

```
-- räknaren
  ctr16: process(clk)
    begin
      if rising_edge(clk) then
        if rst='1' then
          q <= "0000";
        elsif s='1' then
          q <= kb;
        elsif q>0 then
          q <= q-1;
        end if;
      end if;
    end process ctr16;

  -- ut från räknaren
  y <= '0' when q=0 else '1';

  -- S2 hemuppgift
end architecture;
```

Modelsim

reset



http://www.isy.liu.se/edu/kurs/TSEA22/lektion/Modelsim.pdf
http://www.isy.liu.se/edu/kurs/TSEA56/Manualer/SE_tutor.pdf

VHDL-kod

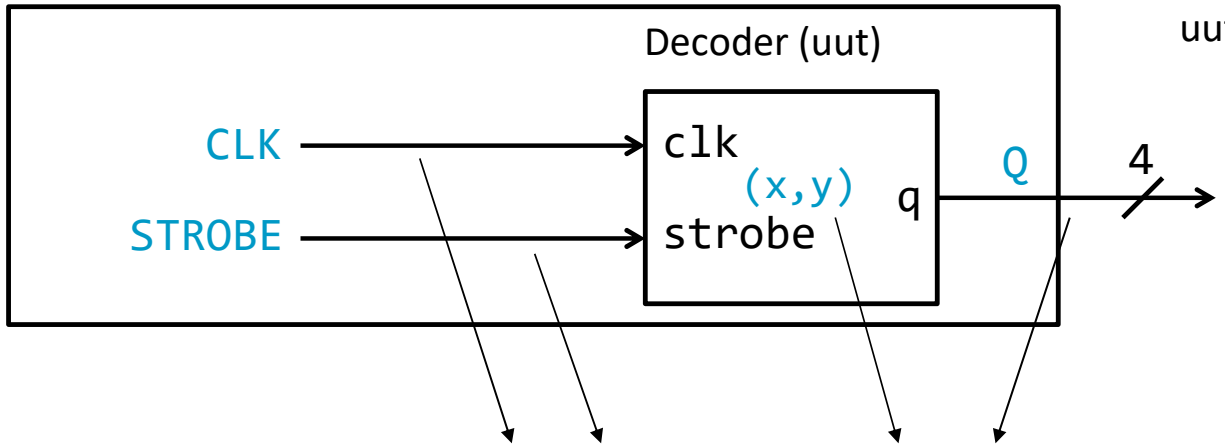
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Decoder is
  Port (clk : in  STD_LOGIC;
        rst : in STD_LOGIC;
        strobe: in  STD_LOGIC;
        kb : in UNSIGNED (3 downto 0);
        q : out  UNSIGNED (3 downto 0));
end Decoder;

architecture func of Decoder is

signal x, y :STD_LOGIC;

begin

  process(clk) begin
    if rising_edge(clk) then
      code
      code
    end if;
  end process;

  code
  code

end architecture;
```

+ testbänk

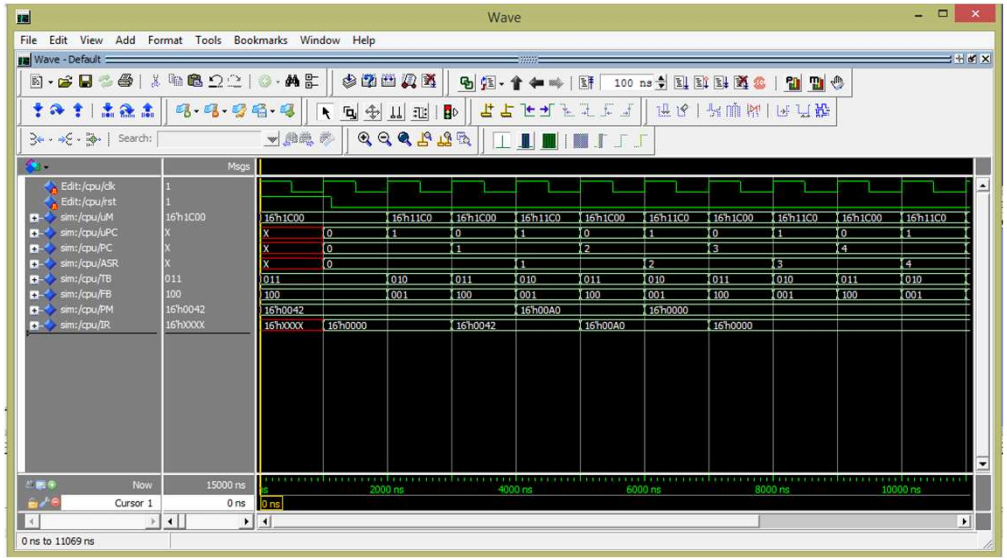
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Decoder_tb is
end Decoder_tb;

architecture func of Decoder_tb is

  component Decoder
    Port (clk : in STD_LOGIC;
          rst : in STD_LOGIC;
          strobe : in STD_LOGIC;
          kb : in UNSIGNED (3 downto 0);
          q : out UNSIGNED (3 downto 0));
  end component;

  -- Testsignaler
  signal CLK : STD_LOGIC;
  signal RST : STD_LOGIC;
  signal STROBE : STD_LOGIC;
  signal KB : UNSIGNED(3 downto 0);
  signal Q : UNSIGNED(3 downto 0);

begin
  uut: Decoder PORT MAP(
    clk => CLK, rst => RST, strobe => STROBE, kb => KB, q => Q);

  -- Klocksignal 10MHz
  CLK <= not CLK after 50 ns;
  STROBE <= '0', '1' after 1 us, '0' after 2 us;
  ...
end;
```

VHDL-kod + testbänk

Testbänk



uut = Unit Under Test



Slutsatser

- Vi ritade blockschema först!
- Vi har samma struktur på koden som på blockschemat!
  - Alltså: små processer som precis motsvarar ett block.
  - Vi har bra koll på mängden hårdvara

För CPLD

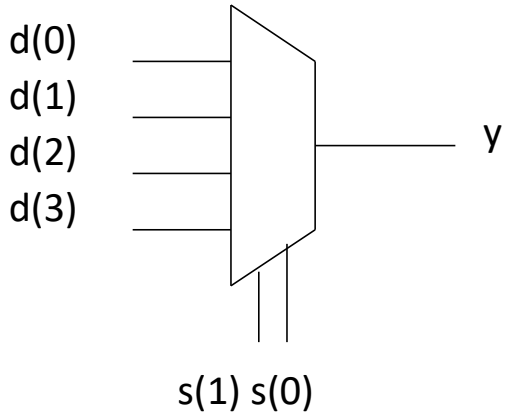
Macrocells Used	Pterms Used	Registers Used	Pins Used	Function Block Inputs Used
10/108 (10%)	40/540 (8%)	8/108 (8%)	9/69 (14%)	19/216 (9%)

Hierakisk konstruktion

Hierarkisk konstruktion

Exempel: vi bygger en 8/1-mux mha 2 st 4/1-muxar.

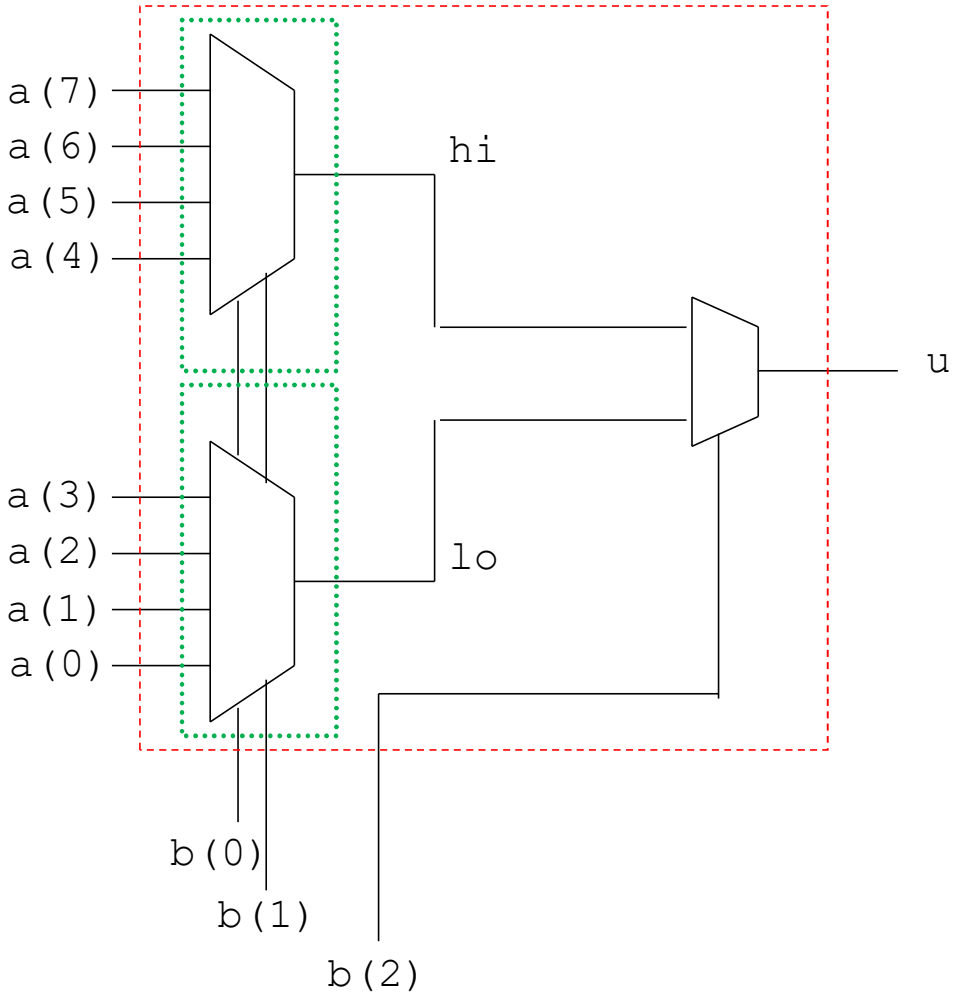
Vi använder 4-1-muxen som komponent:



- Kod finns i fö VHDL1
- Flera olika architectures

En 8-1 mux

```
architecture func of mux4 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;
```



En 8-1 mux

```
entity mux8 is
  port( a: in unsigned(7 downto 0);
        b: in unsigned (2 downto 0);
        u: out std_logic);
end entity;



architecture func of mux8 is
  -- deklarera komponenten mux4
  -- deklarera lokala signaler
begin
  -- instantiera en mux4
  -- instantiera en mux4 till
  -- och några grindar och koppla ihop mux4:orna
end architecture;
```

En 8-1 mux

```
architecture func of mux8 is
  entity mux8 is port( ...
        ...);
  end entity;
  -- deklarera 4-1 mux-komponenten
  component mux4 is
    port( d: in unsigned (3 downto 0);
          s: in unsigned (1 downto 0);
          y: out unsigned);
  end component;
  -- och några lokala signaler
  signal lo,hi: std_logic;
begin
  -- instantiera två mux4:or
  m1: mux4 port map(a(3 downto 0), b(1 downto 0),lo);
  m2: mux4 port map (a(7 downto 4), b(1 downto 0),hi);

  -- och koppla ihop dem
  u <= lo when b(2)='0' else hi;
end architecture;
```

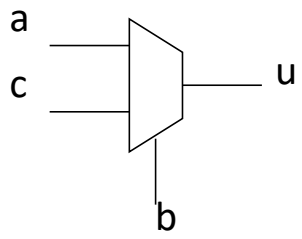
Vi rekommenderar

- Föregående exempel lönar sig knappast. Det är ju enklare att skriva om 4-muxen till en 8-mux.
- Hierarkisk konstruktion blir dock lämplig för stora byggen, tex processorer.
- `component`-satsen används även i testbänkar.
- om man vill simulera ett bygge bestående av flera CPLD-er är `component`-satsen mycket bra att ha!

Lite speciella saker

Kombinatoriska processer
Risker: Multipel tilldelning, latchar
Record
Loop

Kombinatoriska processer använd bara om du måste



OBS! Alla insignaler!!!
Ej klockan!

```
process (a,b,c)
begin
    if b = ´1´ then
        u <= c;
    else
        u <= a;
    end if;
end process;
```

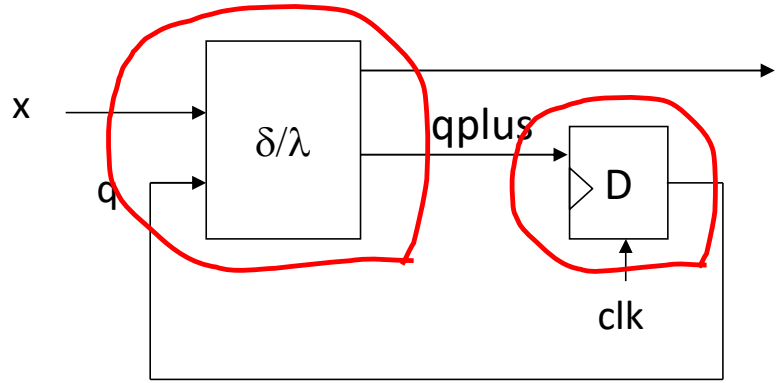
```
u <= c when b='1' else a;
```

Den Booleska funktionen måste vara specad för alla insignalkombinationer!

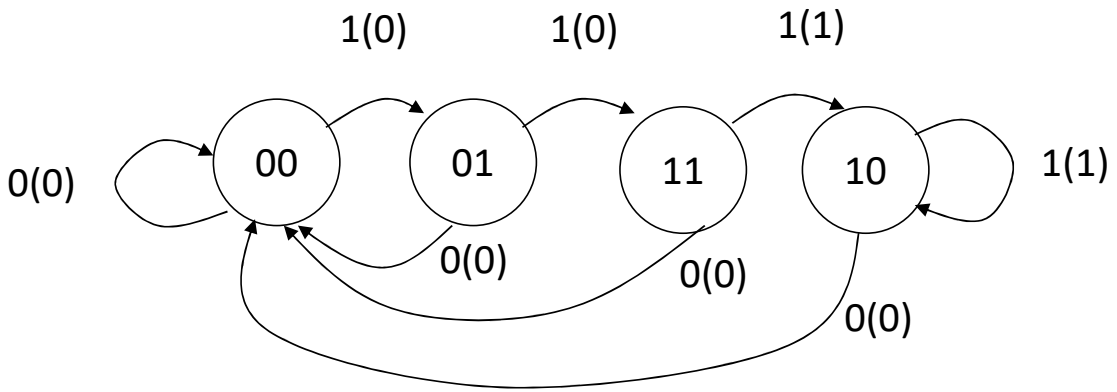
$u = f(a,b,c)$

Sekvensnät 2

1) Gör en kombinatorisk `process(x,q)` av detta



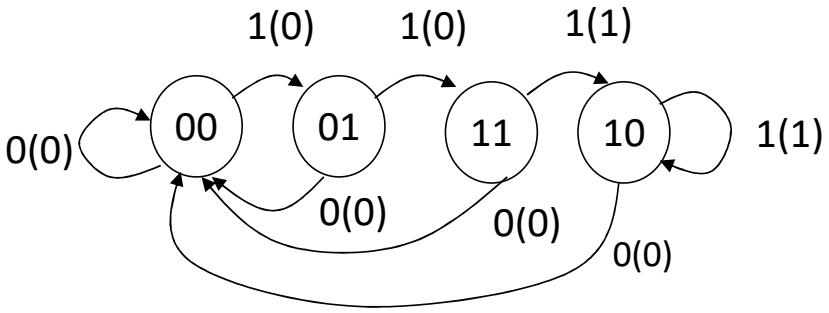
2) Gör en klockad `process(clk)` av detta



Sekvensnät 2

Samma skal som förra gången

```
-- tillståndsvipporna
process(clk)
begin
 if rising_edge(clk)
   q <= qplus;
 end if;
end process;
```




```
-- delta och lambda
process(x,q)
begin
  u <= '0';   -- default-värden
  qplus <= "00";
  if x='1' then
    if q="00" then
      qplus <= "01";
    elsif q="01" then
      qplus <= "11";
    elsif q="11" then
      qplus <= "10";
      u <= '1';
    elsif q="10" then
      qplus <= "10";
      u <= '1';
    end if;
  end if;
end process;
```

Då vi förutsätter att u <= '0' och qplus <= "00" så behöver endast avvikelser från det anges i if-satsen i den kombinatoriska processen.

Nackdel:
Här har man också blandat tilldelning av nästa tillstånd och tilldelning av utsignal i samma process, vilket kan bli svårtläst/svårtolkat.

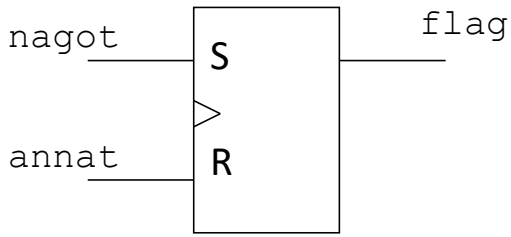
Ett varningsord 1 : Multipel tilldelning (”flagg-tänkande

fel: mjukvarutänk

```
One: process(clk)
...
if nagot then
  flag <= ’1’;
end if;
end process one;


Two: process(clk)
...
if annat then
  flag <= ’0’;
end if;
end process two;
```

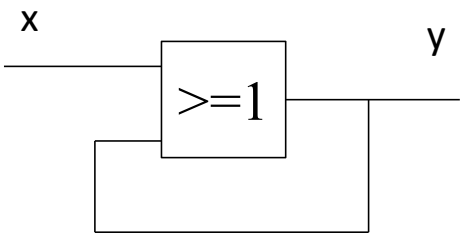
rätt: hårdvarutänk

```
Three: process(clk)
...
if nagot then
  flag <= ’1’;
elsif annat then
  flag <= ’0’;
end if;
end process three;
```



Latch

- Oönskat (oklockat, asynkront) minneselement
  pga kombinatorisk loop
- Kan uppstå pga:
  - Ofullständigt specad kombinatorik
  - Ihopkoppling av Mealynät



x	y
0	0
1	1
-	1

Ett varningsord 2 : Oönskade latchar

Vid **select**-sats och **case**-sats kräver VHDL att alla fall täcks!

Det är inte nödvändigt vid **if**-sats och **when**-sats!
Ibland är detta bra och ibland är det förskräckligt dåligt.
För de fall som inte täcks bibehålls föregående utsignal.

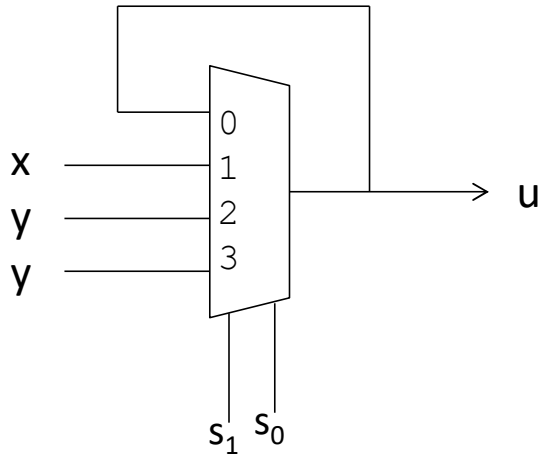
	Sekvensnät (inuti klockad process)	Kombinatorik?

Ofullst.	`if count='1' then`
	`  q <= q+1;`
	`end if;`
		`u <= y when s(1) = '1' else`
		`  x when s(0) = '1';`

Fullst.	`if count='1' then`
	`  q <= q+1;`
	`else`
	`  q <= q;`
	`end if;`
		`u <= y when s(1) = '1' else`
		`  x when s(0) = '1' else`
		`  '0' when others;`

Ett varningsord 2 : Oönskade latchar

Latch = asynkront minneselement



s	u
1	x
0	x
2	y
0	y

record

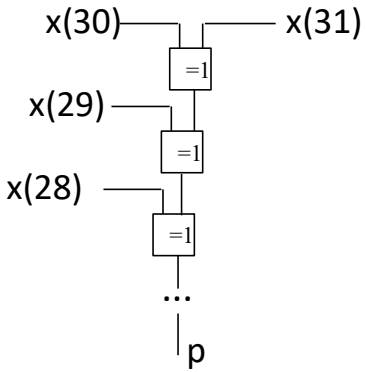
```
type controlword is record
     alu: unsigned(3 downto 0);
     tobus: unsigned(2 downto 0);
     halt: std_logic;
end record;
type styrminne is array(0 to 31) of controlword;

signal styr1, styr2: controlword;
signal mm: styrminne;
--
styr1.halt <= '0';
styr1.alu <= "1011";
styr1.tobus <= styr2.tobus;
--
mm(3) <= ("1011","111",'0');
```

Lite överkurs - loop

Vi har en buss x, med 32 ledningar. Vi vill bilda paritet mellan alla ledningarna. Loopen beskriver på ett kompakt sätt det kombinatoriska nätet!

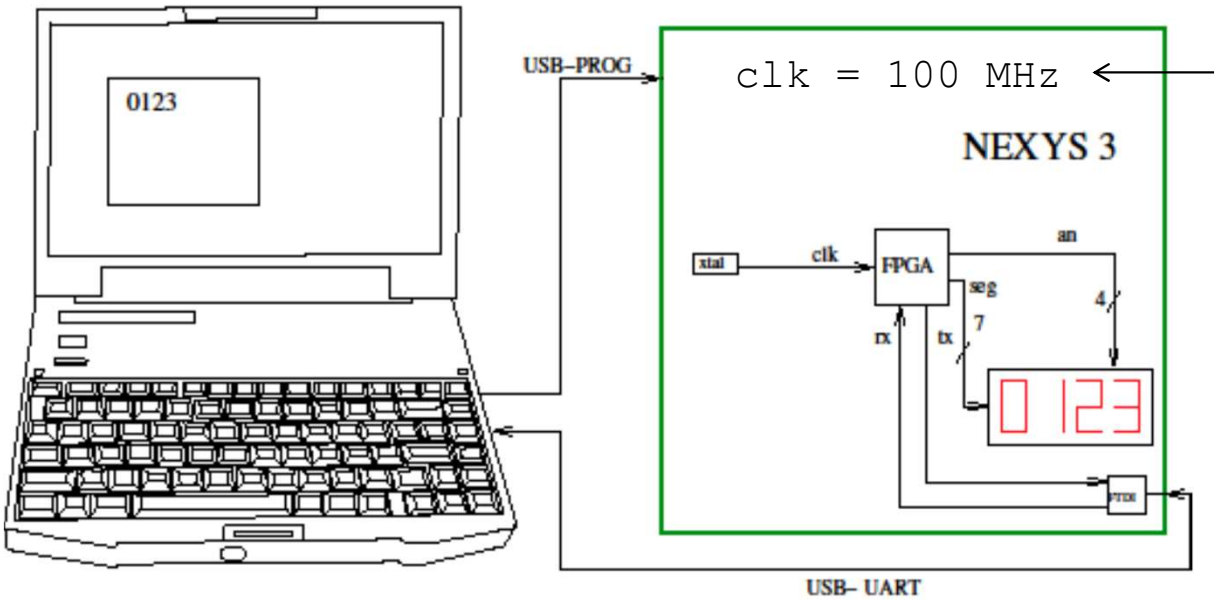
```
entity parity is
    port ( x : in  UNSIGNED (31 downto 0);
           pout : out  STD_LOGIC);
end entity;

architecture func of parity is
begin
  -- kombinatoriskt nät
  process(x)
    variable p: std_logic := '0';
  begin
    for i in 31 downto 0 loop
        p := p xor x(i);
    end loop;
    if p='1' then
      pout <= '1';
    else
      pout <= '0';
    end if;
  end process;

end architecture;
```

p är en variabel
⇒Ingen HW
⇒ vanlig tilldelning

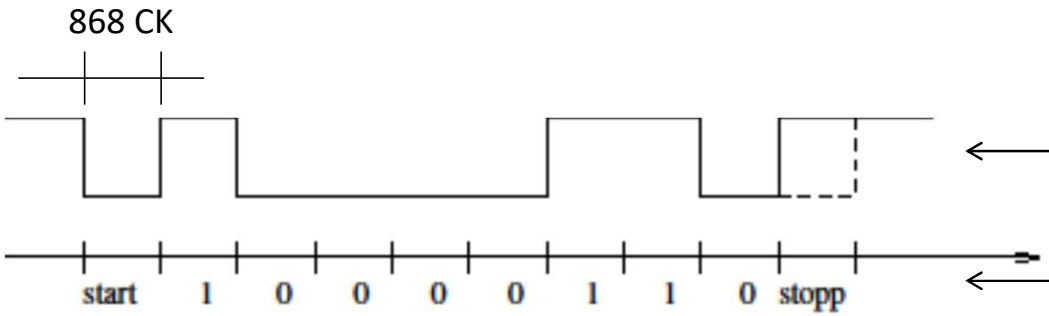


Lab3 : UART

Lab3 : UART



Enda tillåtna klockan, dvs dela inte clk och använd som ny klocksignal (s k grindad klocka).

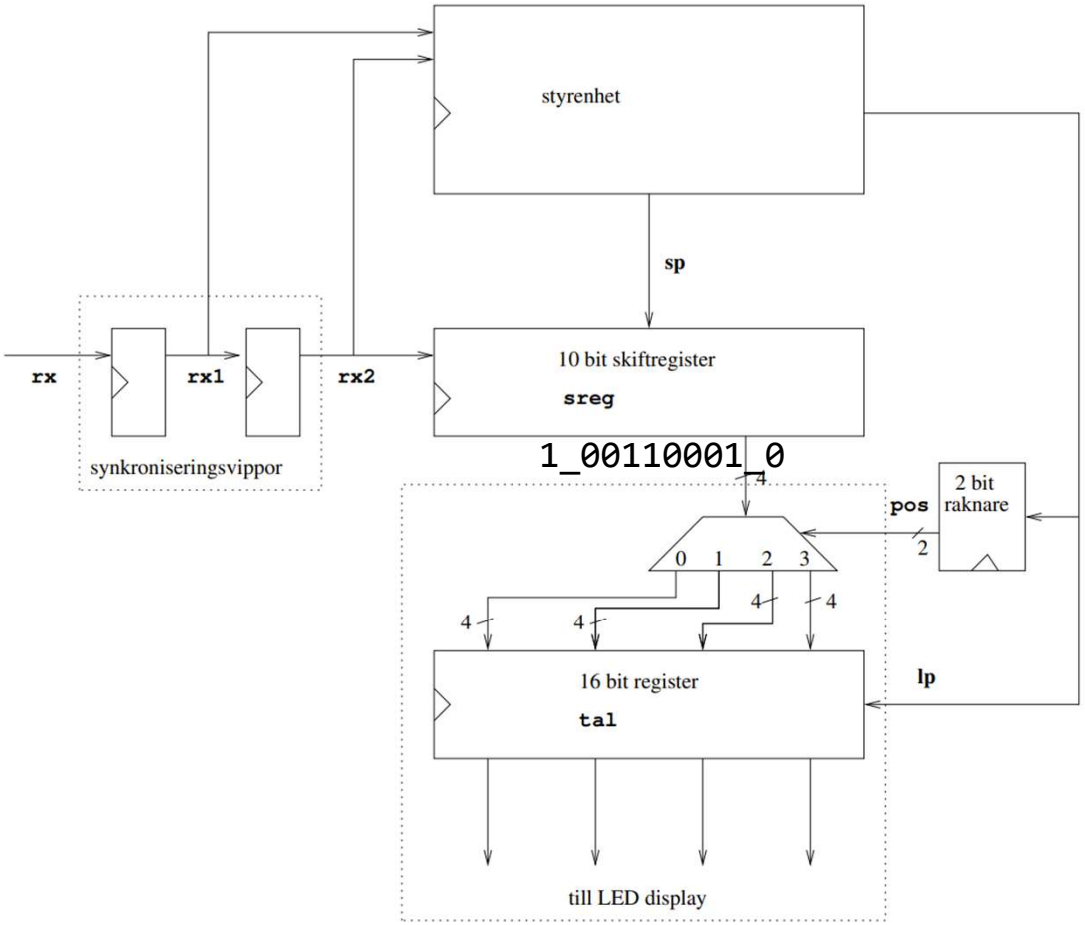


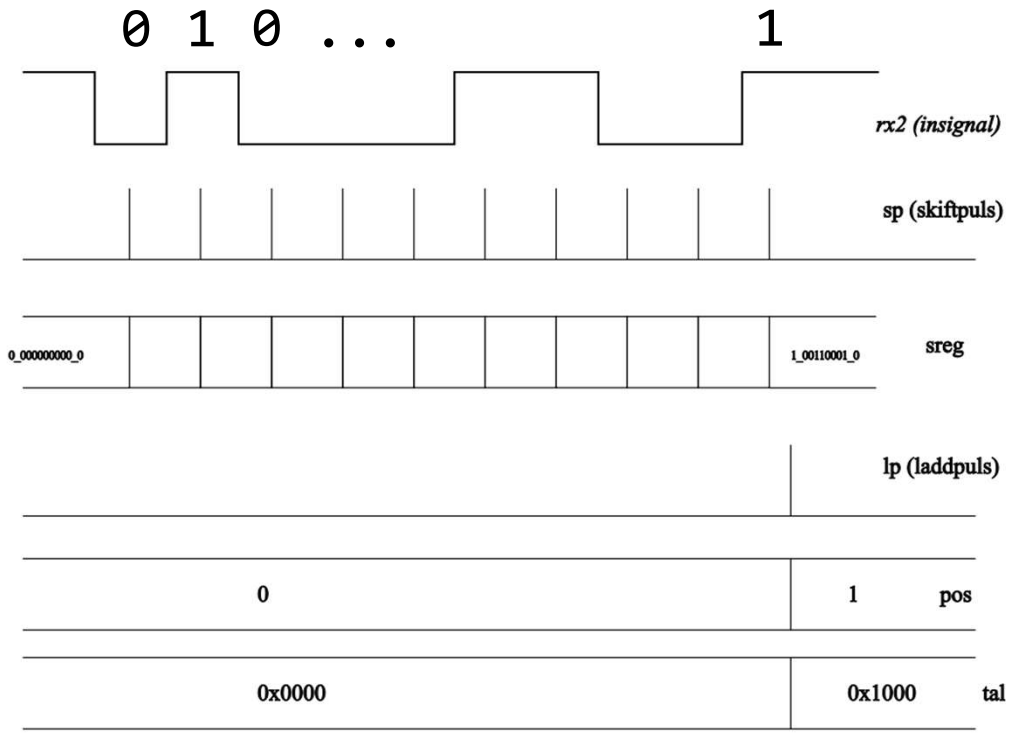
Baud rate 115200 bit/s,
=> 100E6 / 115200 = 868

$01100001_2 = 31_{16} = \text{'1'}_{\text{ASCII}}$

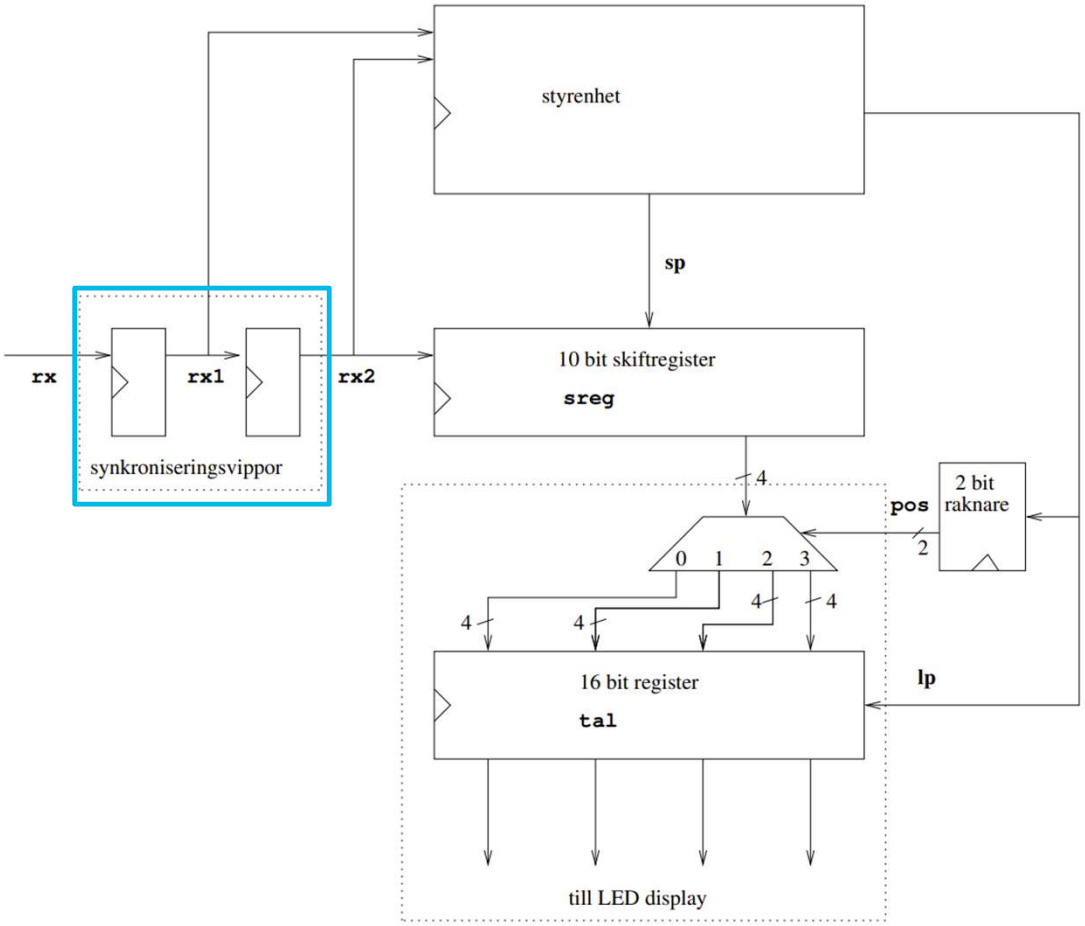
Datorkonstruktion

Lab3 : UART





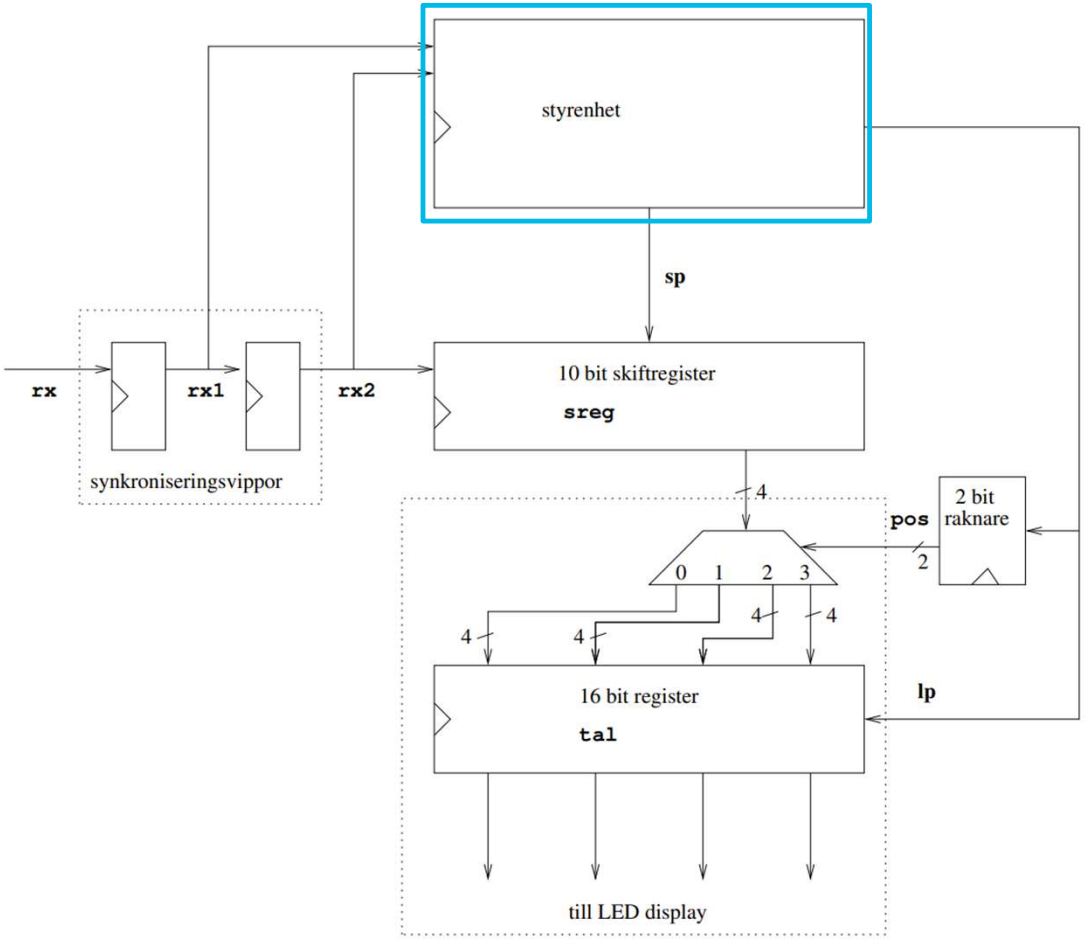
Lab3 : UART



Synkroniseringsvippor

Synkronisera insignalen `rx`, samt använda `rx1` och `rx2` för att hitta startbit (negativ flank).
Dvs, när `rx2=1` och `rx1=0` så har vi hittat flanken.

Lab3 : UART



Styrenhet

Skapa skiftpuls `sp` och laddpuls `lp`.

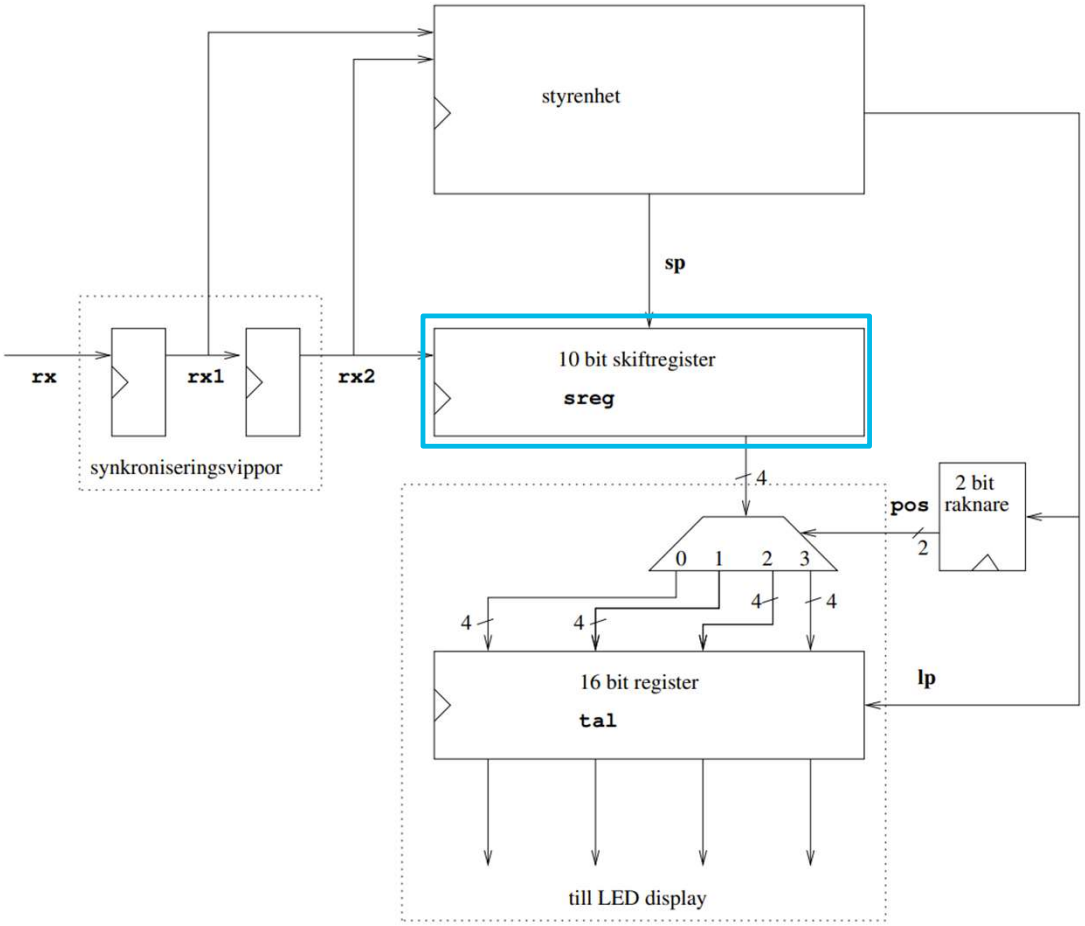
Förslag:

Skapa en stor räknare som börja räkna klockpulser när startpulsens negativa flank hittats (`rx1=0`, `rx2=1`) Använd räknaren för att generera pulserna `sp` och `lp`, vid lämpliga räknarvärden.

Räknaren behöver något som håller ordning på när man ska börja räkna klockpulser, och när man ska sluta och vänta på startpulsen. Det borde räcka med en D-vippa.

Pulserna `sp` och `lp` görs lämpligen som kombinatoriska signaler. Observera att de ska vara en klockpuls långa.

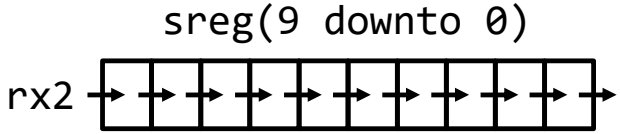
Lab3 : UART



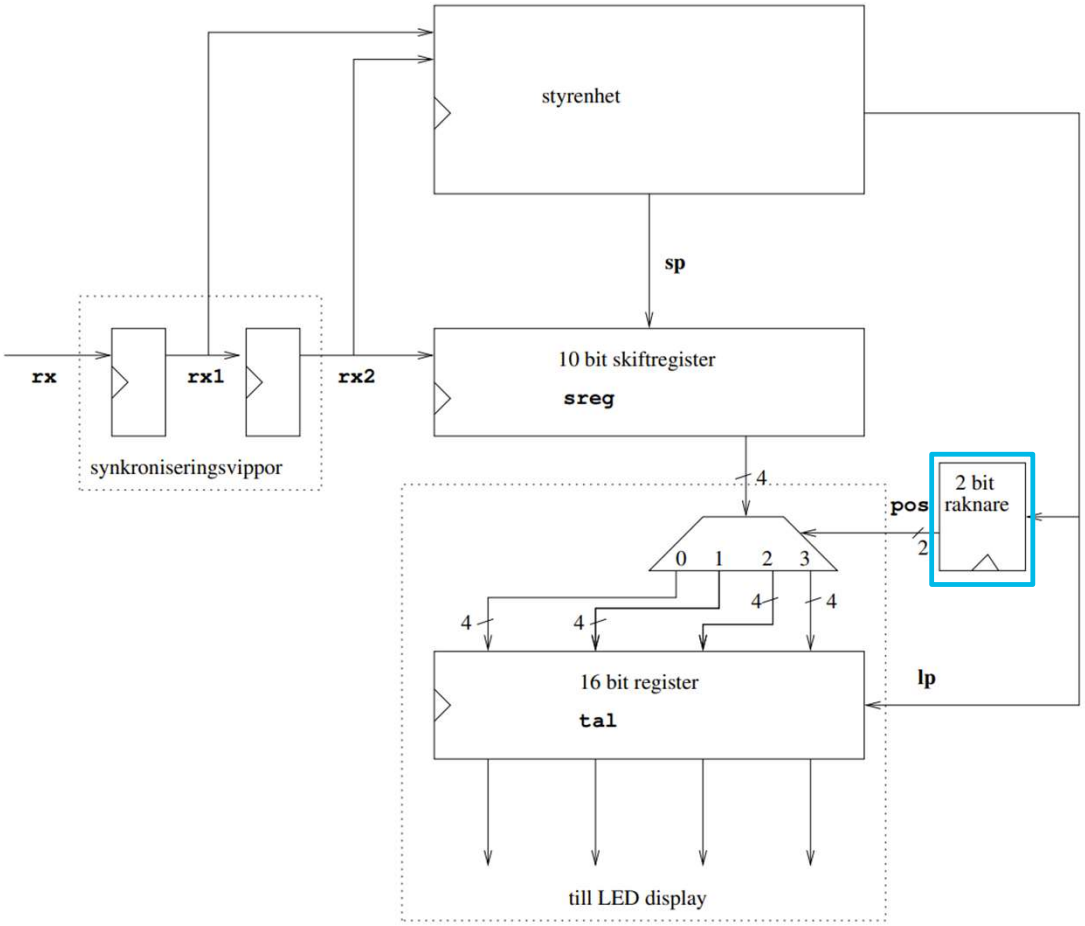
10 bit skiftregister

Skifta in startbit, 8 databitar och stoppbit.

Var gång skiftpulsen `sp` kommer, skifta in värdet av `rx2` till `sreg`.



Lab3 : UART

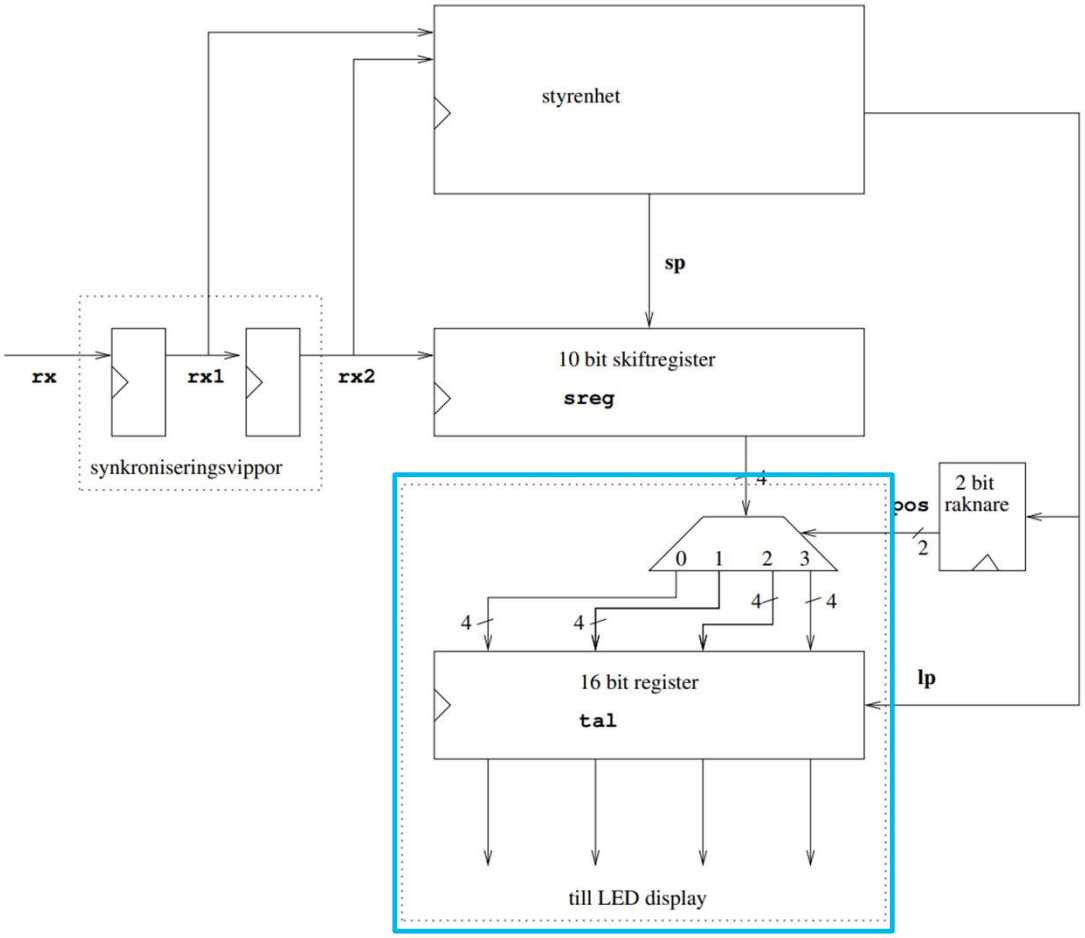


2 bit räknare

Håll ordning på var inkommen siffra ska placeras i 16 bit register.

Var gång laddpulsen `lp` kommer, räkna upp räknarens position `pos` ett steg.
`0->1->2->3->0->...`

Lab3 : UART



16 bit register

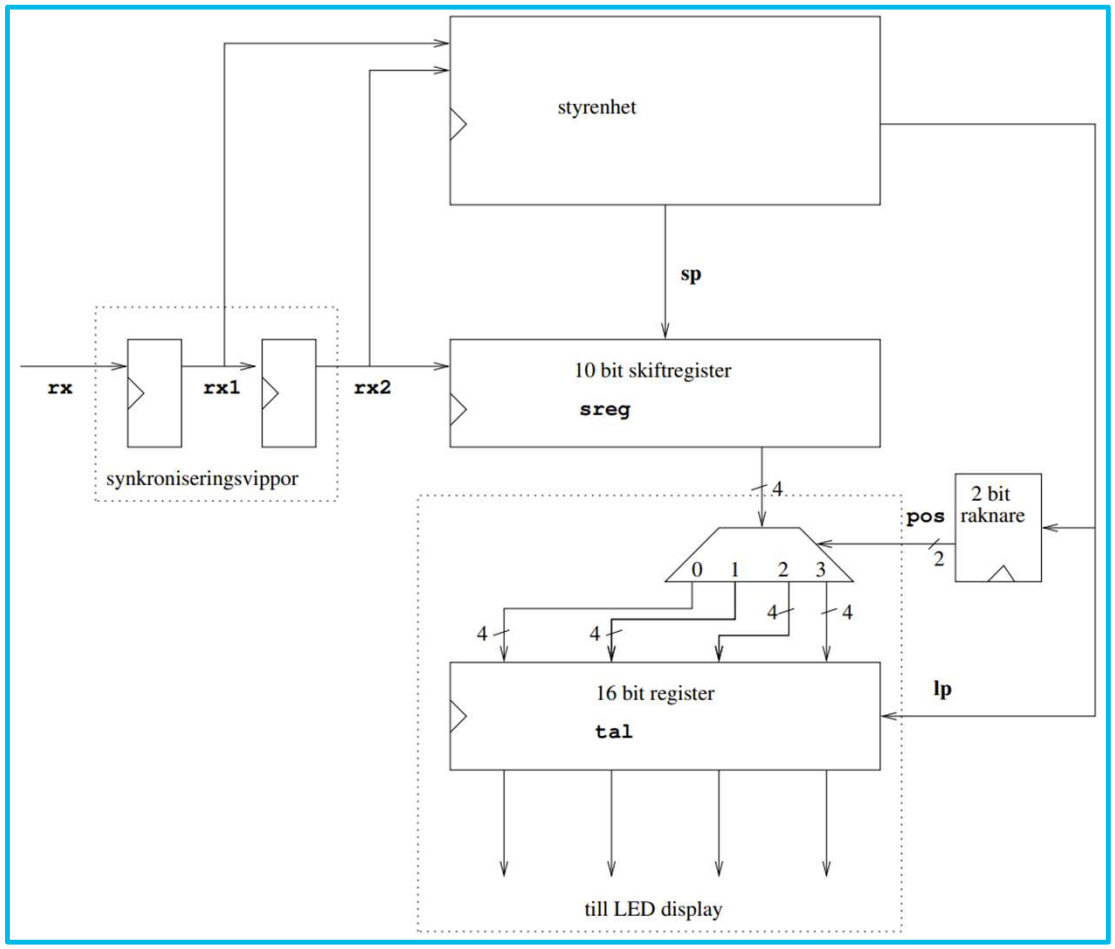
Placera inkommen siffra på rätt plats i registret, utan att påverka övriga siffror.

Var gång laddpulsen `lp` kommer, ta de fyra minst signifikanta bitarna av datavärdet i skiftregistret, och placera dessa på rätt plats (dvs beroende på pos) i 16 bit registret `tal`.

Dvs,

```
om pos=0, tal(15 downto 12) <= sreg(...)
om pos=1, tal(11 downto 8) <= sreg(...)
...
```

Lab3 : UART



lab.vhd

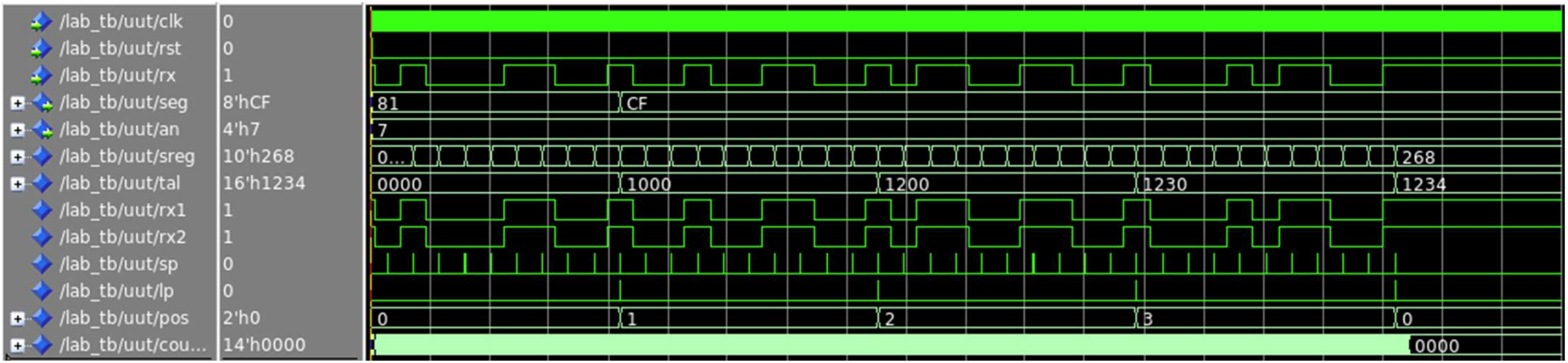
Hela labben är en hierakirsk konstruktion, där `lab.vhd` är toppmodulen och `leddriver.vhd` är en undermodul.

Undermodulen `leddriver.vhd` är redan färdig och de båda modulerna knyts samman i en `Makefile` (se labskelettet).

Datorkonstruktion

Lab3 : UART

Simulera i modelsim

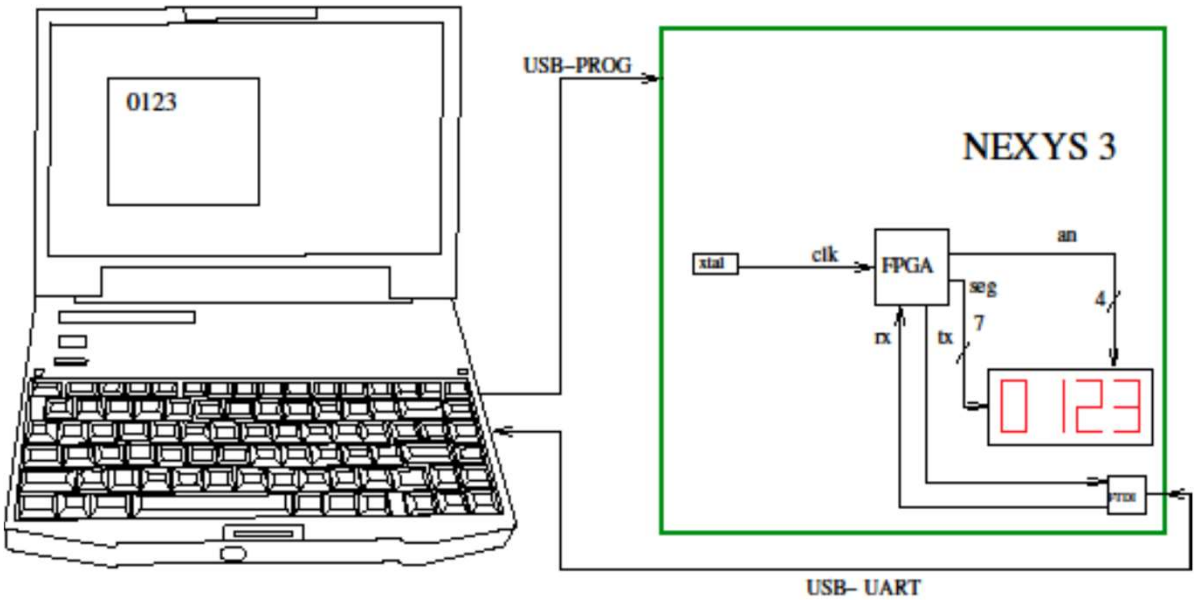


Testbänken `lab_tb.vhd` är färdig och ger insignal på rx som motsvarar indata för siffrorna 1, 2, 3 och 4 (i den ordningen).

Starta simulering i Modelsim med
`make lab.sim`
lägg till önskade signaler
i vågformsfönstret och kör sedan simuleringen i 400 us med kommandot
`run 400 us`
i Modelsims kommandofönster.

Lab3 : UART

I labsalen Muxen



OBSERVERA!

Var försiktig med utrustningen. Den är känslig för statisk elektricitet.

Logga bara ut från datorerna, stäng aldrig av dom. Datorerna används även i andra kurser på distans.

Programmera konstruktionen med
`make lab.prog`

Starta programmet `gtkterm` (och konfigurera enligt lab-PM).
Därefter ska man kunna trycka på datorns siffror och dessa ska synas i sjusegmentsdisplayen på Nexys3-kortet.

Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET