# F7: Konstruktion av sekventiell logik

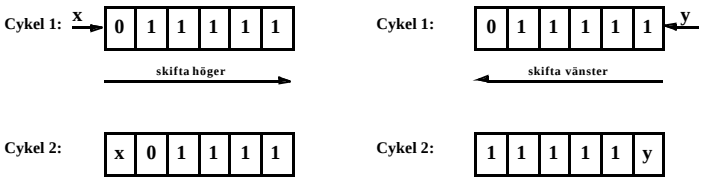
- Innehåll
  - Skiftregister
  - Bit-seriell överföring
  - Skiftregister räknare
  - Ringräknare
  - Johnson räknare
  - LFSR räknare
  - Iterativa jämfört med sekventiella kretsar





# Skiftregister

- Ett skiftregister är ett fler-bitars register som flyttar data i "sidled" vänster/höger
- Förflyttningen (skiftningen) sker 1 bit per klockcykel
- Exempel:



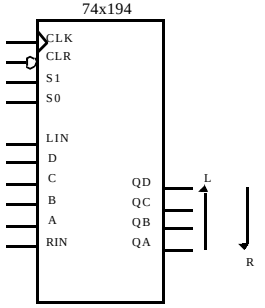

- Används bl.a till att göra multiplikation/division med 2





# MSI skiftregister

- 4-bitars universiellt PIPO register




| | input | | next state | | | |
|---|---|---|---|---|---|---|
| Function | S1 | S0 | QA+ | QB+ | QC+ | QD+ |
| Hold | 0 | 0 | QA | QB | QC | QD |
| Shift right | 0 | 1 | RIN | QA | QB | QC |
| Shift left | 1 | 0 | QB | QC | QD | LIN |
| Load | 1 | 1 | A | B | C | D |



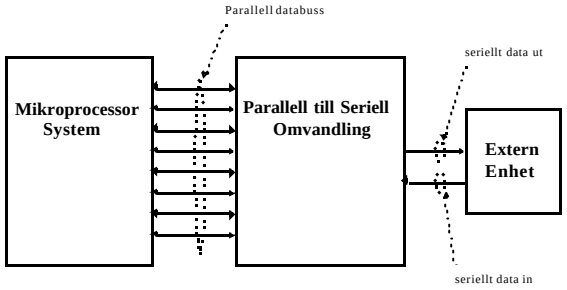


# Skiftregister i VHDL, 74x164

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

entity shreg7164 is
    port (
    clk, clr, sera, serb          : in std_logic;
    qa, qb, qc, qd, qe, qf, qg, qh : out std_logic);

end shreg7164;

architecture rtl of shreg7164 is

signal q_vector : std_logic_vector(7 downto 0);

begin  -- rtl
  process (clk, clr)
  begin  -- process
    if clr = '0' then                   -- asynchronous reset (active low)
      q_vector <= (others=>'0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      q_vector <= shl(q_vector, "1");
      q_vector(0) <= sera and serb;
    end if;
  end process;
  qa <= q_vector(0);  qb <= q_vector(1);  qc <= q_vector(2);  qd <= q_vector(3);
  qe <= q_vector(4); qf <= q_vector(5);  qg <= q_vector(6); qh <= q_vector(7);
end rtl;
```

## Ett seriellt system

- Kommunikation mellan enheter i t.ex mikroprocessor baserat system kan ske parallellt eller seriellt

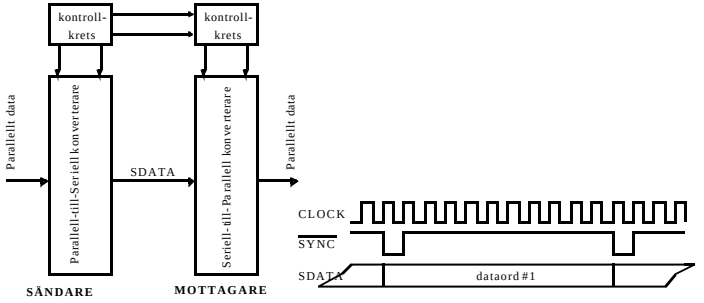







## Synkron överföring

- Synkronisering mellan sändare och mottagare sker med speciella signaler
  - Klocksignal: bestämmer datatakten
  - Synkroniseringssignal: anger början (eller slut) av ett dataord (ett paket med bitar)

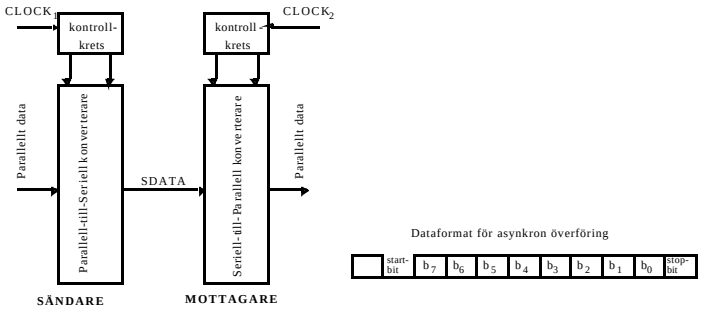







## Asynkron överföring

- Sändare och mottagare arbetar med olika klocksignaler (de är inte synkroniserade med varandra)
- Synkronisering sker genom kodning av datasignalen




Dataformat för asynkron överföring

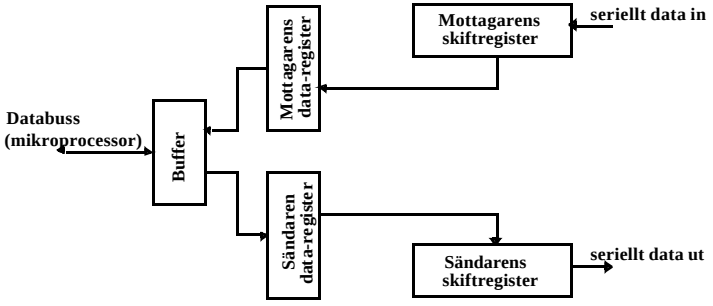




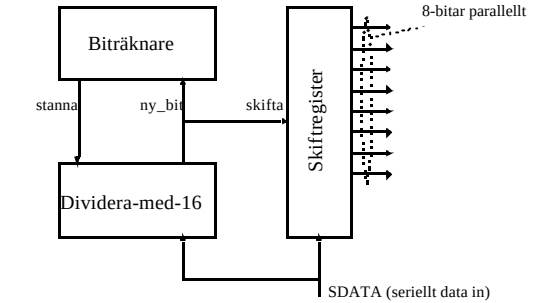


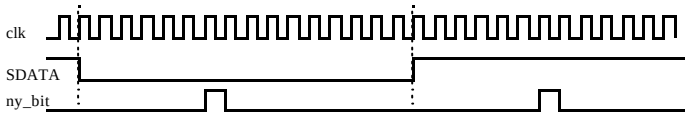

## Universal Asynchronous Receiver Transmitter (UART)

- En UART är en generell interface-krets för att sända och ta emot seriellt data
- Exempel: UART 8250 från Intel
- Blockdiagram:



## UART





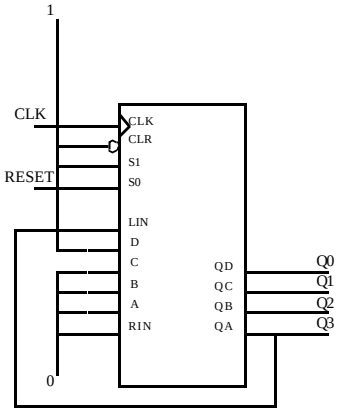



## Tillämpning på skiftregister: Ringräknare

- **Skiftregister-räknare**
  - **cykliska tillståndsgrafer**
  - **inte en "normal" räknesekvens**
- **Ringräknare**
  - **n tillstånd med n-bitars skiftregister**
  - **enklaste formen av räknare**
  - **vid uppstart laddas 0001 → $Q_{ABCD}$**

Räknesekvens

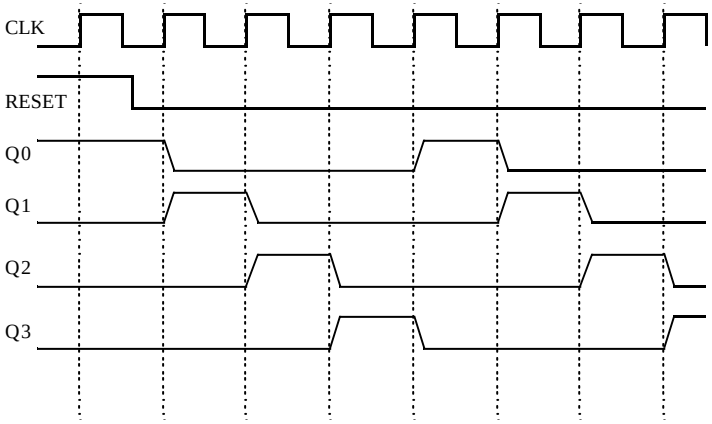
| $Q_A$ | $Q_B$ | $Q_C$ | $Q_D$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |








## Timing för ringräknaren

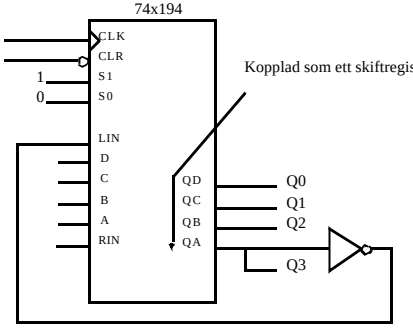






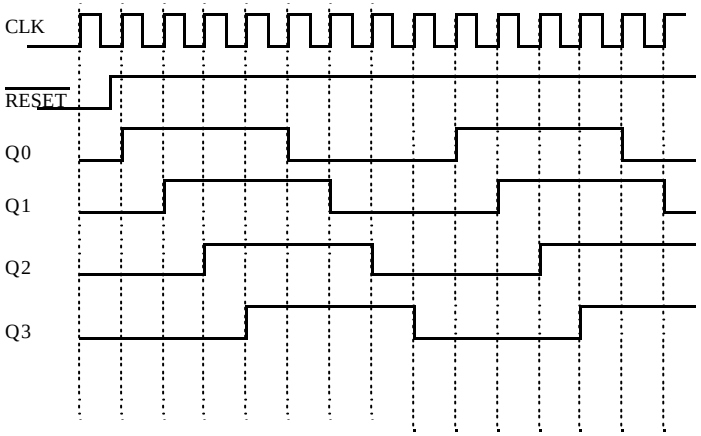

## Johnson-räknare, 4-bitars (8 tillstånd)

- **Kallas också för Möbius-räknare**
  - **seriellt in = ICKE(seriellt ut)**
  - **2·n tillstånd med n D-vippor**




Kopplad som ett skiftregister som skiftar åt vänster.

## Timing för Johnson-räknaren







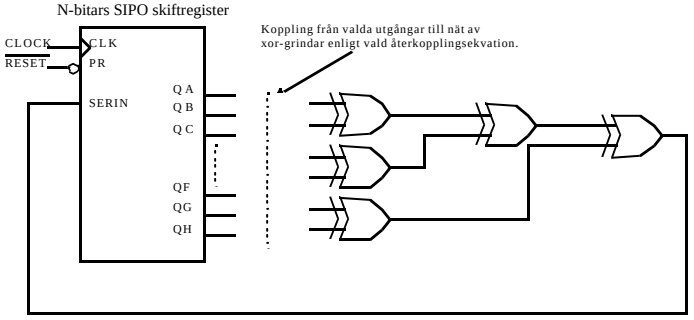

## *Linear Feedback Shift Register* (LFSR) räknare

- **LFSR med n stycken vippor har $2^{n-1}$ tillstånd (nästan maximalt av $2^n$ möjliga)**
  - **Alltså mer ekonomisk än en Johnson-räknare**
- **Seriell ingång = modulo-2 summa av vissa utgångar**
- **För alla n finns det en återkopplingsekvation som ger $2^{n-1}$ tillstånd**
- **Tillstånd med tillståndskoden 000 ... 0 får ej förekomma (eftersom samtliga summor av 000 ... 0 är lika med 0)**
- **Används inom:**
  - **"pseudo-random" nummergenerering**
  - **Feldetekterande kretsar**
  - **Generering och kontroll av feldetekterande koder som t.ex CRC (Cyclic Redundancy Code)**





## Generell LFSR struktur








## LFSR återkopplingsekvationer

| n | Återkopplingsekvation |
|---|---|
| 2 | x2 = x1 xor x0 |
| 3 | x3 = x1 xor x0 |
| 4 | x4 = x1 xor x0 |
| 5 | x5 = x2 xor x0 |
| 6 | x6 = x1 xor x0 |
| 7 | x7 = x3 xor x0 |
| 8 | x8 = x4 xor x3 xor x2 xor x0 |
| 12 | x12 = x6 xor x4 xor x1 xor x0 |
| 16 | x16 = x5 xor x4 xor x3 xor x0 |
| 20 | x20 = x3 xor x0 |
| 24 | x24 = x7 xor x2 xor x1 xor x0 |
| 28 | x28 = x3 xor x0 |
| 32 | x32 = x22 xor x2 xor x1 xor x0 |

# LFSR i VHDL

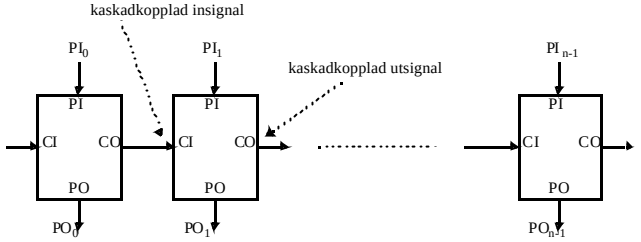
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity LFSR is
  port (
    CLK, RESET : in  std_logic;
    Q          : buffer std_logic_vector(7 downto 0));

end LFSR;

architecture rtl of LFSR is

begin  -- rtl
  synch : process (clk, reset)
  begin  -- process synch
    if reset = '0' then                 -- asynchronous reset (active low)
      Q <= ('0','0','0','0','0','0','0','1');
    elsif clk'event and clk = '1' then  -- rising clock edge
      for i in 7 downto 1 loop
        Q(i) <= Q(i-1);
      end loop;  -- i
      Q(0) <= Q(4) xor Q(3) xor Q(2) xor Q(0);
    end if;
  end process synch;
end rtl;
```





# Iterativa kretsar

- **En iterativ krets är en kombinatorisk krets som:**
  - **består av n stycken identiska moduler**
  - **har både primära ingångar och utgångar samt in- och utgångar för kaskadkoppling**
  - **har signaler som angränsar till modulens yttre gränser**
    - **insignalen längst till vänster kopplas till ett konstant värde**
    - **utsignalen längst till höger kan i många fall innehålla användbar information**
- **Generell struktur för en iterativ kombinatorisk krets**

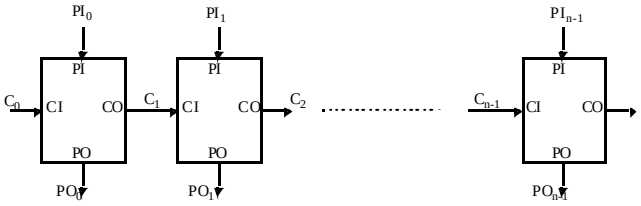







# Iterativa kretsar

- **Iterativa kretsar är lämpliga för att lösa enkla problem som kan beskrivas med algoritmer som:**
  - **1. Sätt $C_0$ till sitt initialvärde och sätt $i$ till 0**
  - **2. Använd $C_i$ och $PI_i$ för att beräkna värdena $PO_i$ och $C_{i+1}$**
  - **3. $i \leftarrow i+1$**
  - **Om $i < n$ gå till steg 2**





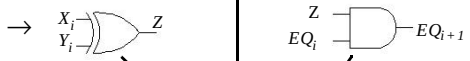



# Exempel: Iterativ komparator

- **Två stycken n-bits värden, $X$ och $Y$, kan jämföras bitvis genom att använda en en-bitars komparator för varje bit. Följande algoritm håller reda på om samtliga bit-par har varit lika:**
  - **1. Sätt $EQ_0 = 1$ och sätt $i = 0$**
  - **2. Om $EQ_i$ är 1 samt $X_i$ och $Y_i$ är lika, sätt $EQ_{i+1} = 1$ annars sätt $EQ_{i+1} = 0$**
  - **3. $i \leftarrow i+1$**
  - **4. Om $i < n$ gå till steg 2**
- **Kretslösning för en en-bitars komparator**

Om $EQ_i = 1$

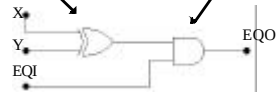
| $X_i$ | $Y_i$ | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |




Om $EQ_i = 0$ blir $EQ_{i+1} = 0$

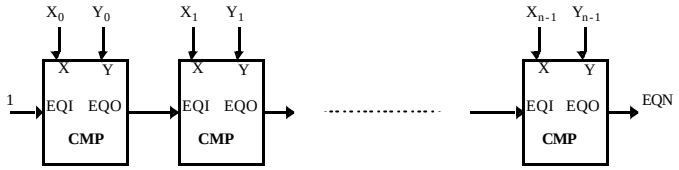


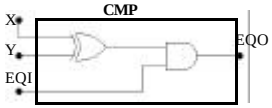

En-bitars komparator (CMP):



## Exempel: Iterativ komparator, n-bitars

- **Komplett krets:**



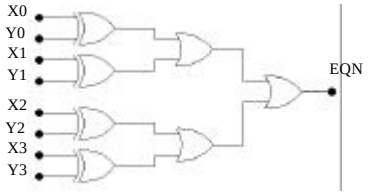

- **EQN indikerar om X och Y är lika**



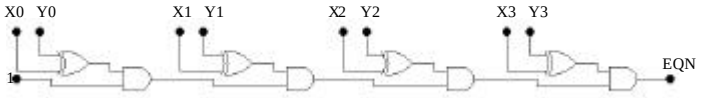


## Iterativ lösning jämfört med en parallell lösning

- **4-bitars parallell komparator**




- **4-bitars iterativ komparator**







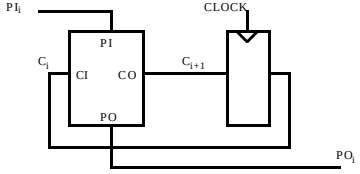

## Iterativa kretsar → sekventiella kretsar

- **Funktionen för en iterativ krets som består av n stycken moduler kan utföras av en sekventiell krets bestående av en modul som arbetar under n stycken klockcykler.**


**n × moduler A under en klockcykel**


**1 × modul A under n klockcykler**

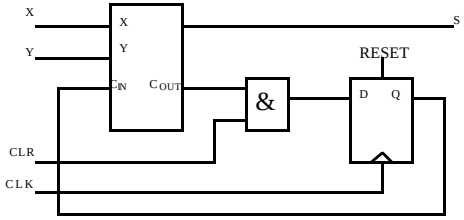
- **Vilken lösning man väljer är en avvägning mellan tid och kretsstorlek**
- **Generell struktur för sekventiell version av en iterativ krets**



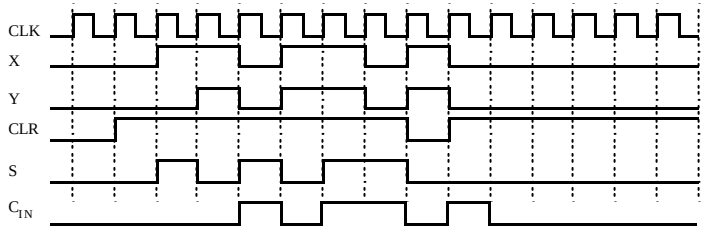





## Exempel: Bit-seriell adderare




| Signal | Funktion |
|---|---|
| X | operand 1 |
| Y | operand 2 |
| S | Summa (X+Y) |
| CLR | nollställ carry |
| CLK | klocka |
| RESET | Asynkron reset av vippan |



# Bit-seriell adderare i VHDL

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

entity bsfa is
    port (
    x, y, reset, clr, clk : in  std_logic;
    s                     : out std_logic);
end bsfa;

architecture rtl of bsfa is
signal cin : std_logic;

  begin  -- rtl
  synch : process (clk, reset, clr, cin, x, y)
  begin  -- process bsfa
    if reset = '0' then
      cin <= '0';
    elsif clk'event and clk = '1' then  -- rising clock edge
      cin <= ((x and y) or (y and cin) or (cin and x)) and clr;
    end if;
    s <= x xor y xor cin;
  end process synch;
+ rtl;
```