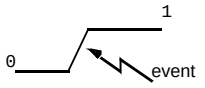
F5: Sekventiell logik i VHDL

- **Innehåll:**
  - **Klocksignal**
  - **latchar**
  - **vippor**
  - **reset och preset**

  - **Tillståndsmaskiner**
    - **räknare**
    - **Mealy maskiner**
    - **Moore maskiner**

Exempel: Positivt flank-triggad D-vippa

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL
ENTITY dff IS
  PORT (d, clk : IN std_logic;
        q : OUT std_logic);
END dff;


ARCHITECTURE behavior OF dff IS
BEGIN
  PROCESS(clk)                -- sensitive ONLY to the clk
  BEGIN
    IF (clk'EVENT AND clk = '1') THEN
      q <= d;
    ELSE
      q <= q;
    END IF;
  END PROCESS;
END behavior;
```



Exempel på andra vippor och latchar

- **Positivt flank triggad D vippa**

```
ARCHITECTURE dataflow OF pos_dff IS
BEGIN
  q <= d WHEN (clk'EVENT AND clk = '1') ELSE q;
END dataflow;
```

- **Negativt flank triggad D vippa**

```
ARCHITECTURE dataflow OF neg_dff IS
BEGIN
  q <= d WHEN (clk'EVENT AND clk = '0') ELSE q;
END dataflow;
```

- **D latch**

```
ARCHITECTURE dataflow OF dlatch IS
BEGIN
  q <= d WHEN (clk = '1') ELSE q;
END dataflow;
```

- **Positivt flank triggad T vippa**

```
ARCHITECTURE dataflow OF dlatch IS
BEGIN
  q <= not q WHEN (clk'EVENT AND clk = '1') ELSE q;
END dataflow;
```


Wait-Until

- ***Wait-Until*satsen är ett alternativ till sensitivitetslistan i en process**
- **Använd antigen *Wait-Until* eller sensitivitetslista, aldrig båda**

```
PROCESS
BEGIN
  WAIT UNTIL (clk = '1');   -- stigande flank
  q <= d;
END PROCESS;cd
```

-

Rising_edge och *Falling_edge*

- **Finktioner som är definierade i std_logic_1164 package**
- **Ersätter** `(clk'EVENT AND clk = '1')` **eller** `(clk'EVENT AND clk = '0')`
- **Funktionerna är endast sanna vid 0-till-1 eller 1-till-0 transitioner**
- **klocksignalen *clk* måste vara av std_logic**
- **Exempel:**

```
IF rising_edge(clk) THEN
```


Reset och *Preset* för vippor

- **Behövs för att initiera innehållet i vippan vid systemstart**
- **Synkron och asynkron**
- **Exempel: Asynkron reset**

```
PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN q <= '0';
  ELSIF (clk'EVENT AND clk = '1') THEN q <= d;
  END IF;
END PROCESS;
```

- **Exempel: Asynkron preset**

```
PROCESS (clk, preset)
BEGIN
  IF preset = '1' THEN q <= '1';
  ELSIF (clk'EVENT AND clk = '1') THEN q <= d;
  END IF;
END PROCESS;
```


Reset och *Preset* för vippor

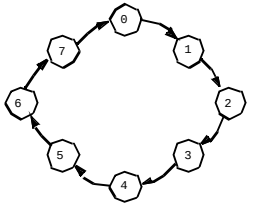
- **Exempel: synkron reset**

```
IF (clk'EVENT AND clk = '1') THEN
  IF reset = '1' THEN q <= '0';
  ELSE q <= d;
  END IF;
END IF;
```

- **Exempel: dataflödes variant på asynkron reset**

```
ARCHITECTURE dataflow OF dff_reset IS
BEGIN
  q <= WHEN reset = '1' ELSE   -- Async. reset
  d <= WHEN (clk'EVENT AND clk = '1')
  ELSE q;
END dataflow;
```


Enkel tillståndsmaskin - 8-räknare




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.numeric_std.ALL;

ENTITY count8 IS
  PORT(clk: IN std_logic;
       cnt: BUFFER unsigned (7 DOWNTO 0));
END count8;

ARCHITECTURE simple OF count8 IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF rising_edge(clk) THEN cnt <= cnt +1;
  END IF
END PROCESS;
END simple;
```

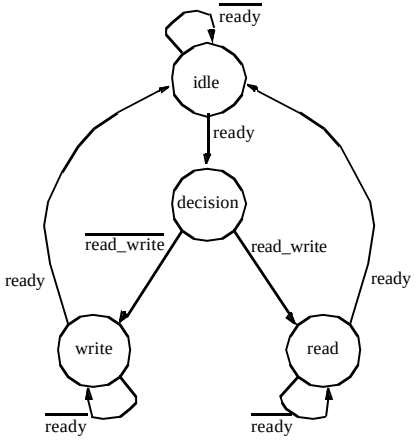
Introduktion till tillståndsmaskiner i VHDL

- Vad är tillståndsmaskiner
  - sekventiella logiska kretsar som endast kan ha ett fixt antal möjliga tillstånd
  - Dess utgångar och nästa tillstånd beror på ingångarna och nuvarande tillstånd
- Metoder för att beskriva en tillståndsmaskin
  - tillståndsdiagram
  - tillståndstabeller
  - Hårdvarubeskrivande språk (t.ex VHDL)

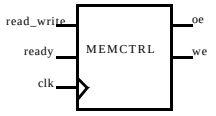
Traditionell konstruktionsmetodik

- Med en given beskrivning av problemet utförs följande steg för att konstruera en tillståndsmaskin:
  - Rita ett tillståndsdiagram för det givna problemet
  - Ta fram en tillståndstabell från tillståndsdiagrammet
  - Eliminera ekvivalenta tillstånd
  - Tilldela tillstånden koder (*state assignment*) och skapa en *state transition table*
  - Från *state transition table* skapa *next-state* logik samt *output* logik

Exempel: Minneskontroller i VHDL



	outputs	
state	**oe**	**we**
idle	0	0
decision	0	0
write	0	1
read	1	0



State transition table för minneskontroller

nuvarande tillstånd		read_write, ready				utgångar	
Tillstånd	**q0 q1**	**0 0**	**0 1**	**1 1**	**1 0**	**oe**	**we**
idle	0 0	0 0	0 1	0 1	0 0	0	0
decision	0 1	1 1	1 1	1 0	1 0	0	0
write	1 1	1 1	0 0	0 0	1 1	0	1
read	1 0	1 0	0 0	0 0	1 0	1	0
		nästa tillstånd		**Q0 Q1**			

$Q0 = q0^* \overline{q1} + q1^* \overline{ready}$

$Q1 = \overline{q0}^* \overline{q1}^* ready + \overline{q0}^* q1^* \overline{read\_write} + q0^* q1^* \overline{ready}$

$we = q0^* q1$

$oe = q0^* \overline{q1}$

Tillståndsmaskin i VHDL

- **Ett givet tillståndsdiagram kan översättas till en högnivåbeskrivning i VHDL utan att man behöver att skpapa en *state transition table.***
- **I VHDL kan varje tillstånd översättas till en *CASE-WHEN* konstruktion**
- **Övergångarna mellan olika tillstånd kan specificeras i *IF-THEN-ELSE* satser**

VHDL kod för minneskontroller

```
ENTITY example IS PORT
  read_write, ready, clk: IN bit,
  oe, we: OUT bit;
END example;

ARCHITECTURE state_machine OF example IS
  TYPE StateType IS (idle, decision, read, write);
  SIGNAL present_state, next_state: StateType;
BEGIN
  state_comb: PROCESS (present_state, read_write, ready)
  BEGIN
    CASE present_state IS
      WHEN idle => oe <= '0'; we <= '0';
        IF ready = '1' THEN
          next_state <= decision;
        ELSE
          next_state <= idle;
        END IF;

          :
          :
```


forts. VHDL kod för minneskontroller

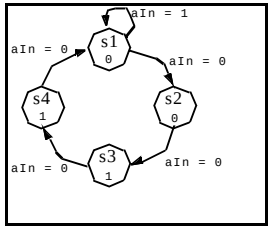
```
      WHEN decision => oe <= '0'; we <= '0';
        IF (read_write = '1') THEN
          next_state <= read;
        ELSE
          next_state <= write;
        END IF;

      WHEN read => oe <= '1'; we <= '0';
        IF (ready = '1') THEN
          next_state <= idle;
        ELSE
          next_state <= read;
        END IF;

      WHEN write => oe <= '0'; we <= '1';
        IF (ready = '1') THEN
          next_state = idle;
        ELSE
          next_state <= write;
      END IF;
    END CASE;
  END PROCESS state_comb;
```


forts. VHDL kod för minneskontroller

```
  state_clocked: PROCESS(clk)
  BEGIN
    IF (clk'EVENT and clk = '1') THEN
      present_state <= next_state;
    END IF;
  END PROCESS state_clocked;
END ARCHITECTURE state_machine;
```

Moore maskin i VHDL

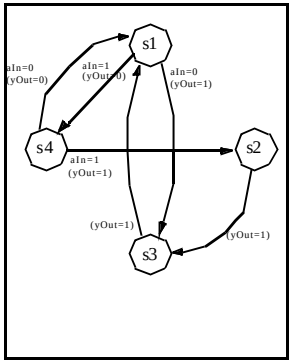
```
LIBRARY ieee; use ieee.std_logic_1164.all;
ENTITY fsm1 IS
  PORT (aIn, clk: IN std_logic; yOut: OUT std_logic);
END fsm1;

ARCHITECTURE moore OF fsm1 IS
  TYPE state IS (s1, s2, s3, s4);
  SIGNAL present_state, next_state: state;
  BEGIN
    PROCESS (aIn, present_state) BEGIN
      CASE present_state IS
        WHEN s1 => yOut <= '0';
          IF (aIn = '1') THEN next_state <= s1
          ELSE next_state <= s2;
        WHEN s2 => yOut <= '0'; next_state <= s3;
        WHEN s3 => yOut <= '1'; next_state <= s4;
        WHEN s4 => yOut <= '1'; next_state <= s1;
      END CASE;
    END PROCESS;
    PROCESS BEGIN
      WAIT UNTIL clk = '1';
      present_state <= next_state;
    END PROCESS;
END moore;
```



- **Notera att utgångens värde bestäms enbart av det tillstånd man befinner sig i.**

Mealy maskin i VHDL

```
ARCHITECTURE mealy OF fsm2 IS
  TYPE state IS (s1, s2, s3, s4);
  SIGNAL present_state, next_state: state;
  BEGIN
  PROCESS (aIn, present_state) BEGIN
    CASE present_state IS
      WHEN s1 => IF (aIn = '1')
        THEN yOut <= '0'; next_state <= s4;
        ELSE yOut <= '1'; next_state <= s3;
      END IF;
      WHEN s2 => yOut <= '1'; next_state <= s3;
      WHEN s3 => yOut <= '1'; next_state <= s1;
      WHEN s4 => IF (aIn = '1')
        THEN yOut <= '1'; next_state <= s2;
        ELSE yOut <= '0'; next_state <= s1;
      END IF;
    END CASE;
  END PROCESS;
  PROCESS BEGIN
    WAIT UNTIL clk = '1';
    present_state <= next_state;
  END PROCESS;
END mealy;
```



- **I tillstånden *s1* och *s4* beror utgången *yOut* på både tillståndet och ingången *aIn***

Testbänkar i VHDL

- **Innehåll:**
  - **Tabulär teknik**
  - **File I/O teknik**
  - **Programmerad teknik**

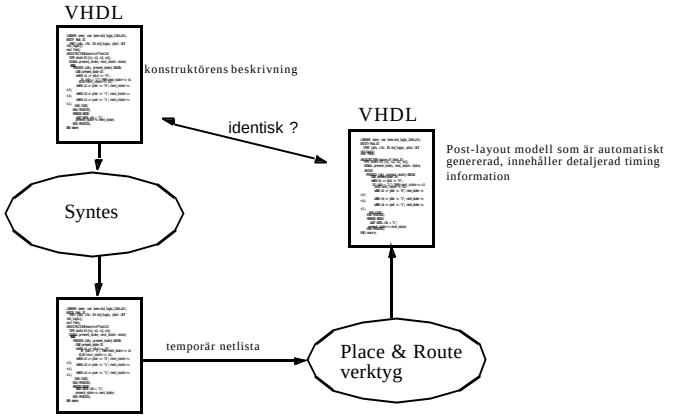
Introduktion

- **Verifiering av konstruktionens funktion är nödvändig.**
  - **Simulera FÖRE syntes för att ta de grova felen tidigt för att undvika onödigt syntesarbete**
  - **Simulera EFTER syntes för att verifiera timing och syntesproceduren**
  - **Realisera konstruktionen i programmerbar komponent eller i ASIC och mät för att verifiera.**
- **VHDL simulering kan vara interaktiv eller automatiserad i en testbänk**
  - **skapa vågformer för att driva simulatorn och utvärdera resultatet i form av grafiska vågformer**
  - **En testbänk är VHDL kod som läggs till för att driva insignaler till konstruktionen (Device Under Test -DUT).**
  - **Responsen från simuleringen kan loggas eller analyseras under det att simuleringen görs.**

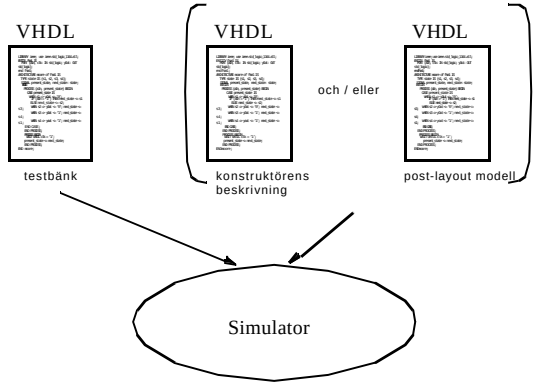
Varför testbänkar ?

- **In- och utsignaler dokumenteras bra - man kan gå tillbaks och se vad man har simulerat**
- **Ett metodiskt arbetssätt**
- **Ger en mer automatiserad test**
- **Samma funktionella test kan upprepas**
- **Samma testbänk kan användas för att verifiera orginal VHDL koden och resultatet från syntesen**

VHDL processering



EN testbänk för både pre- och post-layout implementeringar



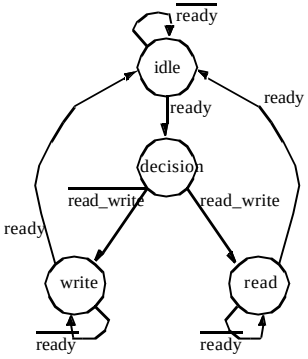
Skapa testbänkar

- **Olika tekniker för att skriva en testbänk**
  - **Tabulär teknik**
    - **en tabell med testvektorer som läggs in i testbänken (d.v.s i VHDL-loden)**
  - **File I/O teknik**
    - **en tabell med testvektorer läggs i en separat datafil som läses av testbänken under simulering**
  - **Programmerad teknik**
    - **en algoritm, skriven i VHDL, används för att skapa insignaler och för att beräkna förväntad respons. Detta används sedan för att automatiskt under simuleringen jämföra förväntad respons med det simulerade resultatet.**

Exempel: minneskontroller som ska verifieras 1(2)

- **entity- och package-deklarationer**

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


package memctrl_package is
component memctrl
  port (
    read_write, ready, clk : in  bit;
    oe, we : out bit);
end component;
end memctrl_package;

library ieee;
use ieee.std_logic_1164.all;
entity memctrl is
  port (
    read_write, ready, clk : in  bit;
    oe, we : out bit);
end memctrl;
```

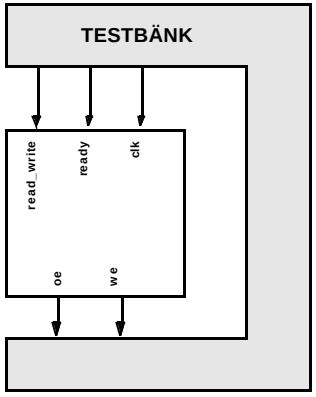


Exempel: minneskontroller som ska verifieras 2(2)

```
architecture
  rtl of memctrl is
  TYPE statetype is (idle, decision, read_state,
                     write_state);
  signal present_state , next_state : statetype;

  begin
    state_comb: process (present_state , read_write, ready)
    begin  -- process
      case present_state is
        when idle => oe <= '0'; we <= '0';
                     if ready = '1' then
                       next_state <= decision;
                     else
                       next_state <= idle;
                     end if;
        when decision => oe <= '0'; we <= '0';
                         if read_write = '1' then
                           next_state <= read_state;
                         else
                           next_state <= write_state;
                         end if;
        when read_state => oe <= '1'; we <= '0';
                           if ready = '1' then
                             next_state <= idle;
                           else
                             next_state <= read_state;
                           end if;
        when write_state => oe <= '0'; we <= '1';
                            if ready = '1' then
                              next_state <= idle;
                            else
                              next_state <= write_state;
                            end if;
      end case;
    end process;
```

```
state_clocked: process(clk)
begin
  if (clk'event and clk = '1') then
     present_state <= next_state;
  end if;
end process;
end rtl;
```


Översikt: Exempel, testbänk för minneskontroller



Testbänk för minneskontroller, tabulär teknik 1(2)

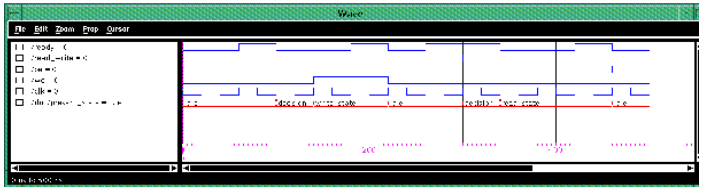
```
library ieee;
use ieee.std_logic_1164.all; use std.textio.all; use work.memctrl_package.all;

entity memctrl_tb1 is
end memctrl_tb1;

architecture behave of memctrl_tb1 is
  signal ready, read_write, oe, we : bit;
  signal clk : bit := '1';
  type test_vector is record
                        ready      : bit;
                        read_write : bit;
                      end record;

  type test_vector_array is array (natural range <>) of test_vector;

  constant test_vectors : test_vector_array := (
  (ready=>'0', read_write=>'0'),
  (ready=>'1', read_write=>'0'),
  (ready=>'0', read_write=>'0'),
  (ready=>'0', read_write=>'0'),
  (ready=>'1', read_write=>'0'),
  (ready=>'0', read_write=>'0'),
  (ready=>'1', read_write=>'0'),
  (ready=>'1', read_write=>'1'),
  (ready=>'0', read_write=>'0'),
  (ready=>'0', read_write=>'0'),
  (ready=>'1', read_write=>'0'),
  (ready=>'0', read_write=>'0')
  );
```

Testbänk för minneskontroller, tabulär teknik 2(2)

```
  constant period : time := 40 ns;
  constant setup_time : time := 5 ns;

begin  -- behave
dut : memctrl port map (
  ready      => ready,
  read_write => read_write,
  oe         => oe,
  we         => we,
  clk        => clk);

  test: process(clk)

    variable cycle : integer := 0;
    variable vector : test_vector;
    begin
      if (clk'event and clk = '1') then
        vector := test_vectors(cycle);
        ready <= vector.ready;
        read_write <= vector.read_write;
        cycle := cycle +1;
      end if;
    end process;

    synch: process
    begin
        clk <= not clk;
        wait for period/2;
    end process;
end behave;
```


Simuleringsresultat från tabulär testbänk

- **Grafiska vågformer från simulatorn**



File I/O teknik

- **Användbart då man behöver ett stort antal testvektorer för att verifiera sin konstruktion**
  - **vektorerna listas i en fil och de läses ut under simulering**
  - **responsen från kretsen loggas i en fil för att kontrolleras efter simuleringen**
- **Tillåter att använda samma testbänk för många olika testkörningar**
  - **detta åstadkoms genom att ändra innehållet i data på filen**
- **Testbänken anropar file I/O procedurer**
  - **läsa in en rad från fil**
  - **skriva en rad till fil**
  - **standard procedurer finns för tecken, strängar, bit, boolean, heltal och flyttal**

File I/O procedurer

- **Datatyper i TEXTIO package**

```
TYPE LINE IS ACCESS STRING;  -- LINE är en pekare till STRING värde

TYPE TEXT IS FILE OF STRING; -- Fil med ASCII records
```

- **Procedurer i TEXTIO package**

```
-- Följande procedurer finns deklarerade för datatypen T, där T kan vara
-- BIT, BIT_VECTOR, BOOLEAN, CHARACTER, INTEGER, REAL, TIME, STRING


PROCEDURE READLINE(FILE F: TEXT; L: OUT LINE); -- Läser in en textrad
PROCEDURE WRITELINE(FILE F:OUT TEXT; L:INOUT LINE); --skriver en textrad
PROCEDURE READ(L : INOUT LINE; VALUE : OUT T);
PROCEDURE WRITE(L : INOUT LINE; VALUE : IN T; <formatterare>);
```

File I/O testvektor fil 1, Testbänk 1(2)

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.memctrl_package.all;

entity memctrl_tb is
end memctrl_tb;

architecture behave of memctrl_tb is
  signal ready, read_write, oe, we : bit;
  signal clk : bit := '1';
  file vector_file : text open read_mode is "control_input.vec";
  file output_file : text open write_mode is "result.dump";

  constant period : time := 40 ns;
  constant setup_time : time := 5 ns;

  begin  -- behave
  dut : memctrl port map (          -- component instance of DUT
    ready      => ready,
    read_write => read_write,
    oe         => oe,
    we         => we,
    clk        => clk);

  test: process(clk)
    variable invecs, outvecs : line;
    variable vready, vread_write : bit;
    variable outvec : bit_vector(1 downto 0);
      :
```


File I/O testvektor fil, Testbänk 2(2)

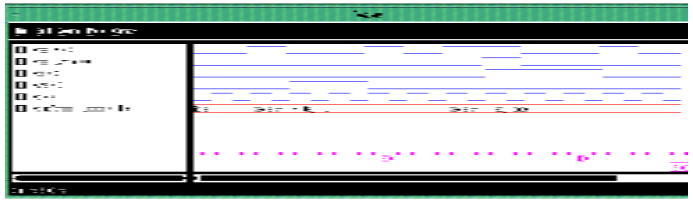
```
      :

      :

    begin
        if (not endfile(vector_file) and clk = '1') then
          readline(vector_file, invecs);
          read(invecs, vready);
          read(invecs, vread_write);

          ready <= vready after period - setup_time;
          read_write <= vread_write after period - setup_time;

          outvec := oe & we;
          write(outvecs, outvec);
          writeline(output_file, outvecs);
        end if;
    end process;

    synch: process
    begin
        clk <= not clk;
        wait for period/2;
    end process;
end behave;
```

exempel på innehållet i vektorfilen:

```
0 0
1 0
0 0
0 0
1 0
0 0
1 0
1 1
0 0
0 0
1 0
0 0
```


File I/O testvektor fil, simuleringsresultet

- **Grafiska vågformer från simulatorn**



- **Textformat i loggfilen som skrivs av testbänken**

```
00
00
00
00
01
01
00
00
00
10
10
10
00
```

result.dump

Programmerad teknik för testbänk

- **Bygg in en algoritm i testbänken som bestämmer utgångarnas värde på DUT**
- **Användbart då man har ett stort antal testvektorer och de kan på ett enkelt sätt beräknas i en algoritm (t.ex aritmetiska enheter som adderare och multiplikatorer)**
- **Också användbart då man ska testa komplicerade tillståndsmaskiner där testbänken ger stimuli till DUT beroende på dess respons.**
- **Exempel: verifiera en modul som delar upp ett positivt heltal `X` i `fac_out` och `exp_out` enligt `X = fac_out * 2`$^{\texttt{exp\_out}}$. Operationen tar ett antal cykler att utföra**