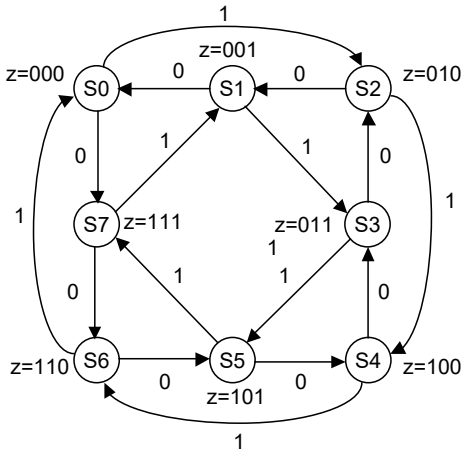
EDA322 Digital konstruktion

Övningar

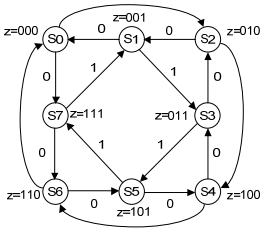
Övning 15 VHDL och testbänk

Övning 15

Vi ritar tillståndsgraf



Övning 15

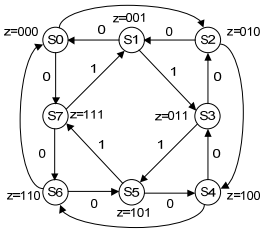


Och översätter till tillståndstabell

Present state	Next state w=0	Next state w=1	Count
S0	S7	S2	0
S1	S0	S3	1
S2	S1	S4	2
S3	S2	S5	3
S4	S3	S6	4
S5	S4	S7	5
S6	S5	S0	6
S7	S6	S1	7

Tidigare har vi realiserat systemet med D-vippor.
Låt oss nu skriva VHDL-kod

Övning 15



Vi kan göra det som
- strukturell kod
- beteendemässig kod
- tillståndsmaskin

Låt oss börja med strukturell kod.

Vi har från tidigare en realisering med tre D-vippor.

Present state	Next state w=0	Next state w=1	Count
S0	S7	S2	0
S1	S0	S3	1
S2	S1	S4	2
S3	S2	S5	3
S4	S3	S6	4
S5	S4	S7	5
S6	S5	S0	6
S7	S6	S1	7

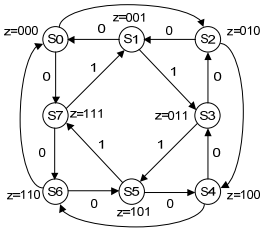
$$D_2 = \overline{w}\cdot Q_2\cdot Q_1 + Q_2\cdot\overline{Q_1}\cdot Q_0 + w\cdot Q_2\cdot\overline{Q_1} + w\cdot\overline{Q_2}\cdot Q_1 + \overline{w}\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{Q_0}$$

$$D_1 = w\cdot\overline{Q_1} + \overline{Q_1}\cdot\overline{Q_0} + \overline{w}\cdot Q_1\cdot Q_0$$

$$D_0 = w\cdot Q_0 + \overline{w}\cdot\overline{Q_0} = \overline{w\oplus Q_0}$$

Övning 15

Vi har entiteten

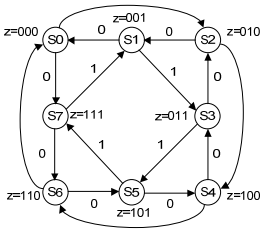


Present state	Next state w=0	Next state w=1	Count
S0	S7	S2	0
S1	S0	S3	1
S2	S1	S4	2
S3	S2	S5	3
S4	S3	S6	4
S5	S4	S7	5
S6	S5	S0	6
S7	S6	S1	7

```
ENTITY ex_15_VHDL_structurell IS
   PORT(Clock:IN STD_LOGIC;
        Resetn:IN STD_LOGIC;
        w:IN STD_LOGIC;
        Q2:OUT STD_LOGIC;
        Q1:OUT STD_LOGIC;
        Q0:OUT STD_LOGIC);
END ex_15_VHDL_structurell;
```

Övning 15

Vi arkitekturen så behöver vi ett antalsignaler för att koppla ihop D-vipporna



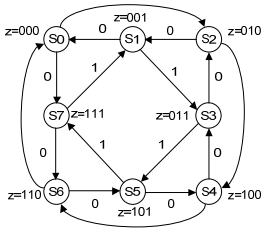
Present state	Next state w=0	Next state w=1	Count
S0	S7	S2	0
S1	S0	S3	1
S2	S1	S4	2
S3	S2	S5	3
S4	S3	S6	4
S5	S4	S7	5
S6	S5	S0	6
S7	S6	S1	7

```
SIGNAL D2_signal:STD_LOGIC;
   SIGNAL D1_signal:STD_LOGIC;
   SIGNAL D0_signal:STD_LOGIC;
   SIGNAL Q2_signal:STD_LOGIC;
   SIGNAL Q1_signal:STD_LOGIC;
   SIGNAL Q0_signal:STD_LOGIC;
   SIGNAL Q_vector_signal:STD_LOGIC_VECTOR(2 DOWNTO 0);
```

`Q_vector_signal` är till för att kunna se den samlade utsignalen från de tre vipporna

Övning 15

Vi behöver D-vippor som komponenter.

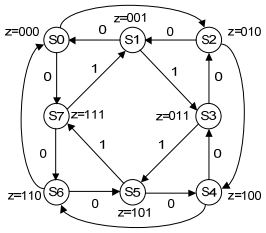


Present state	Next state w=0	Next state w=1	Count
S0	S7	S2	0
S1	S0	S3	1
S2	S1	S4	2
S3	S2	S5	3
S4	S3	S6	4
S5	S4	S7	5
S6	S5	S0	6
S7	S6	S1	7

```
ENTITY D_flipflop IS
   PORT(Clock:IN STD_LOGIC;
        Resetn:IN STD_LOGIC;
        D:IN STD_LOGIC;
        Q:OUT STD_LOGIC);
END D_flipflop;
ARCHITECTURE arch_D_flipflop OF D_flipflop IS
BEGIN
   D_proc:
   PROCESS(Resetn,Clock)
   BEGIN
      IF (Resetn='0') THEN
         Q <= '0';
      ELSIF rising_edge(Clock) THEN
         IF (D='1') THEN
            Q <= '1';
         ELSE
            Q <= '0';
         END IF;
      END IF;
   END PROCESS D_proc;
END arch_D_flipflop;
```

Övning 15

Vi deklarerar komponenten D_flipflop.



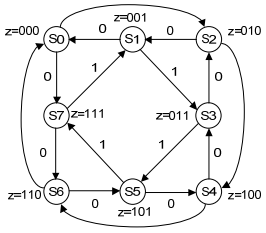
Present state	Next state w=0	Next state w=1	Count
S0	S7	S2	0
S1	S0	S3	1
S2	S1	S4	2
S3	S2	S5	3
S4	S3	S6	4
S5	S4	S7	5
S6	S5	S0	6
S7	S6	S1	7

```
COMPONENT D_flipflop IS
      PORT(Clock:IN STD_LOGIC;
               Resetn:IN STD_LOGIC;
               D:IN STD_LOGIC;
               Q:OUT STD_LOGIC);
   END COMPONENT D_flipflop;
```

Övning 15

Vi instantierar tre D-vippor

```
D_flipflop_comp_2:
   D_flipflop
   PORT MAP(Clock => Clock,
            Resetn =>Resetn,
            D => D2_signal,
            Q => Q2_signal);

   D_flipflop_comp_1:
   D_flipflop
   PORT MAP(Clock => Clock,
            Resetn =>Resetn,
            D => D1_signal,
            Q => Q1_signal);

   D_flipflop_comp_0:
   D_flipflop
   PORT MAP(Clock => Clock,
            Resetn =>Resetn,
            D => D0_signal,
            Q => Q0_signal);
```



Present state	Next state w=0	Next state w=1	Count
S0	S7	S2	0
S1	S0	S3	1
S2	S1	S4	2
S3	S2	S5	3
S4	S3	S6	4
S5	S4	S7	5
S6	S5	S0	6
S7	S6	S1	7

Övning 15

$$D_2 = \overline{w}\cdot Q_2\cdot Q_1 + Q_2\cdot\overline{Q_1}\cdot Q_0 + w\cdot Q_2\cdot\overline{Q_1} + w\cdot\overline{Q_2}\cdot Q_1 + \overline{w}\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{Q_0}$$

$$D_1 = w\cdot\overline{Q_1} + \overline{Q_1}\cdot\overline{Q_0} + \overline{w}\cdot Q_1\cdot Q_0$$

$$D_0 = w\cdot Q_0 + \overline{w}\cdot\overline{Q_0} = \overline{w\oplus Q_0}$$

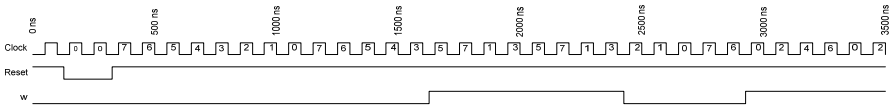
Vi skapar våra styrsignaler

```
D2_signal <=(NOT(w) AND Q2_signal AND Q1_signal) OR
             (Q2_signal AND NOT(Q1_signal) AND0_signal) OR
             (w AND Q2_signal AND NOT(Q1_signal)) OR
             (w AND NOT(Q2_signal) AND Q1_signal) OR
             (NOT(w) AND NOT(Q2_signal) AND
                          NOT(Q1_signal) AND NOT(Q0_signal));
D1_signal <= (w AND NOT(Q1_signal)) OR
              (NOT(Q1_signal) AND NOT(Q0_signal)) OR
              (NOT(w) AND Q1_signal AND Q0_signal);
   D0_signal <= w XNOR Q0_signal;

Q_vector_signal <= Q2_signal & Q1_signal & Q0_signal;

Q2 <= Q2_signal;
Q1 <= Q1_signal;
Q0 <= Q0_signal;
```

Och vår design är klar

Övning 15

Vi simulerar med en do-fil och behöver då först skapa ett testmönster



Övning 15

Och så en do-fil

```
restart -f -nowave
view signals wave
add wave Clock Resetn w
add wave count_signal count
add wave -radix binary count
add wave -unsigned count
force Clock 0 0, 1 50ns -repeat
100ns
force Resetn 1
force w 0
run 125ns
force Resetn 0
run 200ns
force Resetn 1
run 1300ns
force w 1
run 800ns
force w 0
run 500ns
force w 1
run 500ns
```

Formatkon-verteringar

Övning 15

Vi övergår till beteendemässig kod

Anslutningarna ändras inte så vi har samma entitet som tidigare

```
ENTITY ex_15_VHDL_behavioral IS
   PORT(Clock:IN STD_LOGIC;
        Resetn:IN STD_LOGIC;
        w:IN STD_LOGIC;
        count:OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END ex_15_VHDL_behavioral;
```

Övning 15

Så arkitekturen

```
ARCHITECTURE arch_ex_15_VHDL_behavioral OF
 ex_15_VHDL_behavioral IS
   SIGNAL count_signal:INTEGER RANGE 0 TO 7;
BEGIN
   count_proc:
   PROCESS(Resetn,Clock)
   BEGIN
      IF (Resetn='0') THEN
         count_signal<=0;
      ELSIF rising_edge(Clock) THEN
         IF (w='1') THEN
            IF (count_signal = 6) THEN
               count_signal <= 0;
            ELSIF (count_signal = 7) THEN
               count_signal <= 1;
            ELSE
              count_signal <= count_signal + 2;
            END IF;
         ELSE
            IF (count_signal = 0) THEN
               count_signal <= 7;
            ELSE
               count_signal <= count_signal -1;
            END IF;
         END IF;
      END IF;
   END PROCESS count_proc;
      count <= STD_LOGIC_VECTOR(TO_UNSIGNED(count_signal,3));
END arch_ ex_15_VHDL_behavioral;
```

Vi använder `INTEGER RANGE` som räknarvariabel

Övning 15

Vi kan använda samma do-fil

```
restart -f -nowave
view signals wave
add wave Clock Resetn w
add wave count_signal count
add wave -radix binary count
add wave -unsigned count
force Clock 0 0, 1 50ns -repeat 100ns
force Resetn 1
force w 0
run 125ns
force Resetn 0
run 200ns
force Resetn 1
run 1300ns
force w 1
run 800ns
force w 0
run 500ns
force w 1
run 500ns
```

Övning 15

Vi övergår till som tillståndsmaskinkod

Anslutningarna ändras inte så vi har samma entitet som tidigare

```
ENTITY ex_15_VHDL_FSM IS
   PORT(Clock:IN STD_LOGIC;
        Resetn:IN STD_LOGIC;
        w:IN STD_LOGIC;
        count:OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END ex_15_VHDL_FSM;
```

Övning 15

I arkitekturen har vi normalt tre processer

- En process som styr övergången mellan tillstånden
- En process där vi bestämmer nästa tillstånd
- En process där vi tilldelar utsignalerna värden

Övning 15

Vi börjar med processen som styr övergången mellan tillstånden.
Denna process ser alltid likadan ut.

```
state_transition_proc:
   PROCESS(Resetn,Clock)
   BEGIN
      IF (Resetn='0') THEN
         state_signal<=S0;
      ELSIF rising_edge(Clock) THEN
         state_signal<=next_state_signal;
      END IF;
   END PROCESS state_transition_proc;
```

Övning 15

I processen som bestämmer nästa tillstånd följer vi tillståndstabellen.

```
flow_proc:
   PROCESS(state_signal,w)
   BEGIN
      CASE state_signal IS
          WHEN S0 =>
            IF w = '1' THEN
               next_state_signal <= S2;
            ELSE
               next_state_signal <= S7;
            END IF;
         WHEN S1 =>
            IF w = '1' THEN
               next_state_signal <= S3;
            ELSE
               next_state_signal <= S0;
            END IF;
         WHEN S2 =>
         ...
```

Present state	Next state w=0	Next state w=1	Count
S0	S7	S2	0
S1	S0	S3	1
S2	S1	S4	2
S3	S2	S5	3
S4	S3	S6	4
S5	S4	S7	5
S6	S5	S0	6
S7	S6	S1	7

Här visar vi bara några av tillstånden.

Övning 15

Oftast deklarerar vi tillståndsvariablerna som en uppräknande typ som vi sedan skapar signaler från.

```
TYPE state_type IS (S0,S1,S2,S3,S4,S5,S6,S7);
   SIGNAL state_signal:state_type;
   SIGNAL next_state_signal:state_type;
```

I det här fallet har vi dock ett unikt räknevärde som utsignal i varje tillstånd och vi kan då ge tillståndsvariabeln precis samma värde som utsignalen skall ha i detta tillstånd.

Vi deklarerar tillståndsvariablerna som konstanter.

Övning 15

```
CONSTANT S0:STD_LOGIC_VECTOR(2 DOWNTO 0):="000";
CONSTANT S1:STD_LOGIC_VECTOR(2 DOWNTO 0):="001";
CONSTANT S2:STD_LOGIC_VECTOR(2 DOWNTO 0):="010";
CONSTANT S3:STD_LOGIC_VECTOR(2 DOWNTO 0):="011";
CONSTANT S4:STD_LOGIC_VECTOR(2 DOWNTO 0):="100";
CONSTANT S5:STD_LOGIC_VECTOR(2 DOWNTO 0):="101";
CONSTANT S6:STD_LOGIC_VECTOR(2 DOWNTO 0):="110";
CONSTANT S7:STD_LOGIC_VECTOR(2 DOWNTO 0):="111";
SIGNAL state_signal:STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL next_state_signal:STD_LOGIC_VECTOR(2 DOWNTO 0);
```

Övning 15

Genom att göra på detta sätt behöver vi inga speciella utsignaler utan kan helt enkelt lägga tillståndssignalen `state_signal` på utgångarna.

```
count<=state_signal;
```

Övning 15

Låt oss nu övergå till att skapa testbänkar till vår konstruktion

Det finns tre typer av testbänkar

- Typ 1 – vi anger bara instimuli som i en do-fil och får själva utvärdera resultatet
- Typ 2 – vi inför en testsignal som signalerar när något har gått fel men vi får själva utreda vad som gick fel och när det hände
- Typ 3 – vi inför meddelanden som anger när något gått fel och vi kan lägga in meddelanden som förklarar felen

Övning 15

I alla tre testbänkstyperna skapar vi en överliggande konstruktion där vi använder vår design som en komponent.

Vi genererar också instimuli som skall testa vår design.

Testbänken använder sig bara av vår designs in- och utgångar, dvs entiteten till vår testbänk skall fungera all de designer vi gjort.

Låt oss använda namnen från vår beteendemässiga design.

Övning 15

Alla testbänkarna har samma entitet som är tom. Vi har inga in- eller utportar

```
ENTITY ex_15_VHDL_behavioral_tb1 IS

END ex_15_VHDL_behavioral_tb1;
```

Övning 15

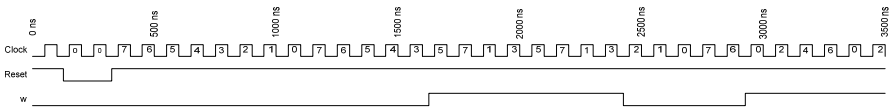
I arkitekturen instantierar vi vår design som en komponent.

```
COMPONENT ex_15_VHDL_behavioral IS
      PORT(Clock:IN STD_LOGIC;
           Resetn:IN STD_LOGIC;
           w:IN STD_LOGIC;
           count:OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
   END COMPONENT ex_15_VHDL_behavioral;

   SIGNAL Resetn_tb_signal:STD_LOGIC;
   SIGNAL Clock_tb_signal:STD_LOGIC:='0';
   SIGNAL w_tb_signal:STD_LOGIC;
   SIGNAL count_tb_signal:STD_LOGIC_VECTOR(2 DOWNTO 0);
   BEGIN
   counter_integer_comp:
   COMPONENT ex_15_VHDL_behavioral
      PORT MAP(Clock=>Clock_tb_signal,
               Resetn=>Resetn_tb_signal,
               w=>w_tb_signal,
               count=>count_tb_signal);
```

Anslutningar till komponenten

Övning 15

Vi genererar nu instimuli till vår design enligt vår tidigare graf.




```
Resetn_tb_signal<='1',
                '0' AFTER 125 ns,
                '1' AFTER 325 ns;
  w_tb_signal<='0',
                '1' AFTER 1625 ns,
                '0' AFTER 2425 ns,
                '1' AFTER 2925 ns;
```

Övning 15

Vi genererar också en klocka.

```
clock_process:
   PROCESS
   BEGIN
      WAIT FOR 50 ns;
      Clock_tb_signal<=NOT(Clock_tb_signal);
   END PROCESS clock_process;
```

En symmetrisk klocka med periodtiden 100 ns.
Den börjar låg då vi initierat signalen till noll (0).

```
SIGNAL Clock_tb_signal:STD_LOGIC:='0';
```

Notera att det här är tillåtet att använda initialvärden då signalen bara används vid simulering och aldrig blir hårdvara.

Övning 15

Vad vi nu gjort är en testbänk av typ 1.

- Vi skapar en testbänk
- Vi instantierar vår design som en komponent
- Vi skapar instimuli
- Vi skapar en klocka

Övning 15

Även här behöver vi en do-fil men den skall bara ange vilka signaler vi vill se och hur lång simuleringstid som skall köras, inte ge några signaler.

```
restart -f -nowave
view signals wave
add wave Resetn_tb_signal Clock_tb_signal
add wave w_tb_signal count_tb_signal
run 3500ns
```

Övning 15

Om vi övergår till testbänk typ 2 så innehåller även den ovanstående delar men vi kompletterar med en test av resultatet.

Vi inför en signal som skall vara ett (1) då testen är klar om allt har gått som det skall. Är något fel så har den vid den tid då felet kom gått ner till noll (0).

```
SIGNAL test_OK_signal:STD_LOGIC;
```

Övning 15

Vi inför en testprocess där vi efter varje signal-förändring som vi vill testa gör halt (`WAIT`) och gör testen.

Vi lägger våra tester vid tider då vi inte har någon föränd-ring av något instimuli

```
test_process:
  PROCESS
  BEGIN
     test_OK_signal<='1';
     WAIT FOR 210 ns; -- 210 ns
     IF (Resetn_tb_signal/='0' OR
         count_tb_signal/="000") THEN
            test_OK_signal<='0';
     END IF;
     WAIT FOR 200 ns; -- 410 ns
     IF (Resetn_tb_signal/='1' OR
         count_tb_signal/="111") THEN
            test_OK_signal<='0';
     END IF;
     WAIT FOR 100 ns; -- 510 ns
     IF count_tb_signal/="110" THEN
            test_OK_signal<='0';
     END IF;
     WAIT FOR 1000 ns; -- 1510 ns
     IF count_tb_signal/="100" THEN
            test_OK_signal<='0';
     END IF;
     ...
```

Här visar vi bara några av alla tester.

Lägg märke till test-signalen `test_-OK_signal` från början ges de-faultvärdet ett (1) och att den kommer att ligga kvar på noll (0) efter ett fel även om efterföljande tester är OK

Övning 15

do-filen blir den samma som för testbänk typ 1 med den skillnaden att vi lägger till att vi vill se signalen `test_OK_signal` i vågformfönstret.

Tack vare testsignalen kan vi nu i vågformfönstret se när något blev fel men vi får själva analysera vågformerna för att ta rätt på vad som hänt.

Övning 15

I testbänk typ 3 har ersätter vi den tidigare testprocessen med en ny sådan.

Här gör vi motsvarande test som i testbänk typ 2 men vi markerar inte via en signal att nå´t är fel utan vi kan i stället få utskrivet tiden då felet skedde och en beskrivande text som kan förklara felet.

Övning 15

Kommentar

```
test_proc:
   PROCESS
   BEGIN
      WAIT FOR 210 ns; -- 210 ns
      ASSERT (count_tb_signal="000")
      REPORT "count_tb_signal/=000"
      SEVERITY ERROR;
      ASSERT (Resetn_tb_signal='0')
      REPORT "Resetn_tb_signal/=0"
      SEVERITY ERROR;
      ...
      WAIT FOR 100 ns; -- 510 ns
      ASSERT (count_tb_signal="110")
      REPORT "count_tb_signal/=110"
      SEVERITY ERROR;
      ...
```

Återigen bara några av testerna.

Två tester vid tiden 210 ns

En test vid tiden 510 ns

ASSERT reagerar om villkoret inom parentes INTE är uppfyllt

REPORT används för att skriva ut meddelande

SEVERITY anger felets svårighetsgrad

Övning 15

Vi har alltså fyra olika svårighetsgrader som vi kan ange för felet.
I de tre första fallen kommer simuleringen att fortsätta efter meddelandet

- NOTE är egentligen inget fel utan bara en kommentar
- WARNING är som namnet anger inget fel utan en varning
- ERROR anger ett fel men simuleringen kan fortsätta
- FALIURE är ett fel som gör fortsatt simulering meningslös så simuleringen stannar vid denna tidpunkt