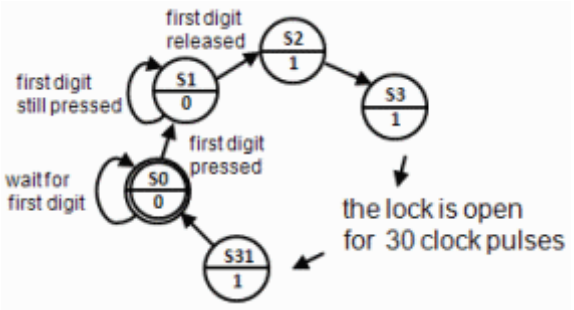
VHDL för ett kodlås

Beskrivning av kodlåsmallen

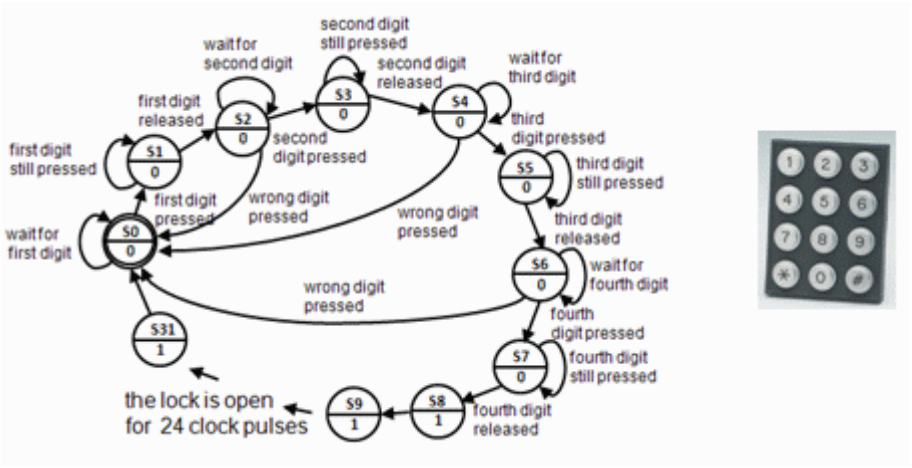


Kodlåsmallen gäller för ett förenklat lås som öppnar när man trycker på tangenten för "1" och sedan släpper tangenten.

Så gott som all digital design sker numera med hjälp av högnivåspråk som VHDL/VERILOG. Vår grundkurs i digital-teknik ger inte utrymme att lära ut VHDL-språket, däremot kommer Du att kunna omforma "kodlåsmallen" till användbar VHDL-kod inför laborationen.

Tycker Du att VHDL-språket verkar intressant, så har skolan sedan flera digitaltekniska fortsättningskurser.

Vid laborationen utökar Du denna kod till ett fyrsiffrigt kombinationslås!

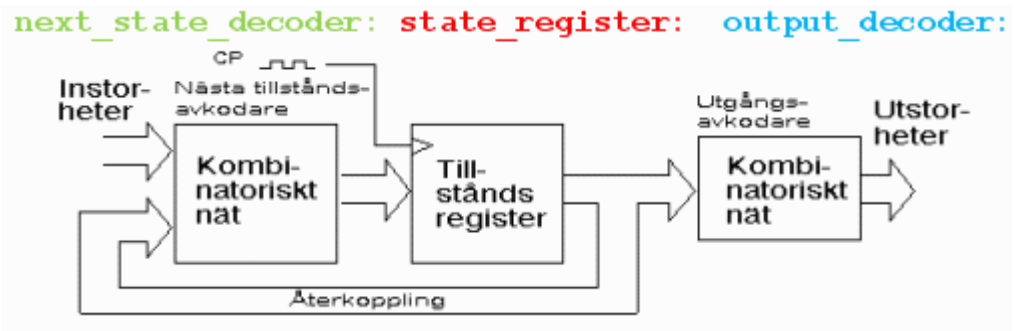


lockmall.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity codelock is
   port( clk:      in  std_logic;
         K:      in  std_logic_vector(1 to 3);
         R:      in  std_logic_vector(1 to 4);
         q:      out std_logic_vector(4 downto 0);
         UNLOCK: out std_logic );
end codelock;

architecture behavior of codelock is
subtype state_type is integer range 0 to 31;
signal state, nextstate: state_type;

begin
nextstate_decoder: -- next state decoding part
process(state, K, R)
 begin
   case state is
      when 0 => if (K = "001" and R ="0001")     then nextstate <= 1;
                else nextstate <= 0;
                end if;
      when 1 => if (K = "001" and R = "0001")    then nextstate <= 1;
                elsif (K = "000" and R = "0000") then nextstate <= 2;
                else nextstate <= 0;
                end if;
      when 2 to 30 => nextstate <= state + 1;
      when 31 => nextstate <= 0;
   end case;
end process;

debug_output:  -- display the state
q <= conv_std_logic_vector(state,5);

output_decoder: -- output decoder part
process(state)
begin
  case state is
     when 0 to 1  => UNLOCK <= '0';
     when 2 to 31 => UNLOCK <= '1';
  end case;
end process;

state_register: -- the state register part (the flipflops)
process(clk)
begin
  if rising_edge(clk) then
     state <= nextstate;
  end if;
end process;
end behavior;
```

Mooreautomat

Kodlåset är utformat som en Moore-atomat.

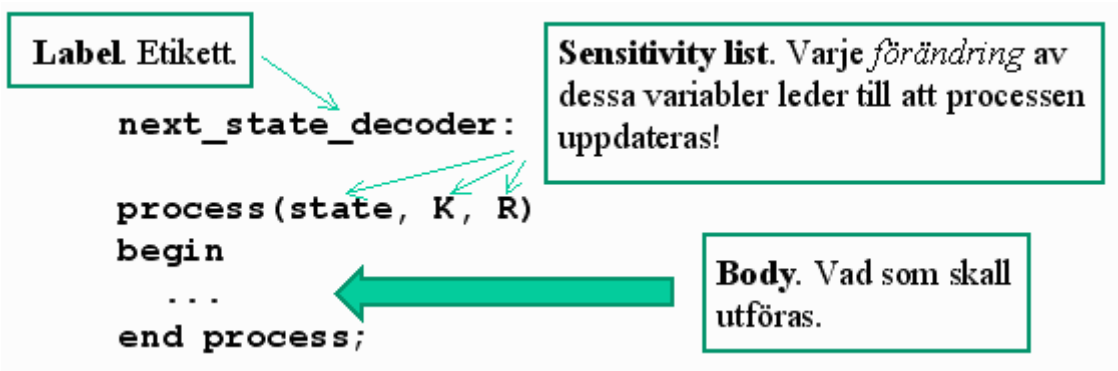


De olika blocken identifieras i koden med etiketter, "labels".

```
next_state_decoder:
output_decoder:
state_register:
```

VHDL processer

Med "processer" kan man beskriva vad ett block ska utföra utan att behöva gå in på detaljer om hur detta skall gå till.

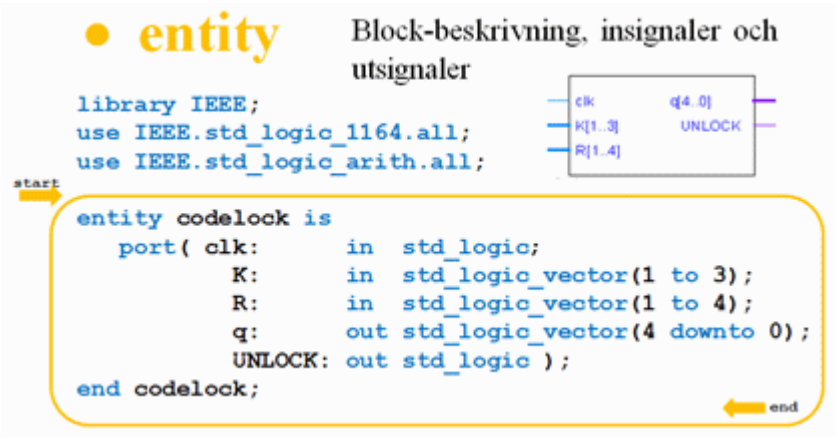


VHDL-koden är skriven som ett antal sådana processer.

Programmets delar

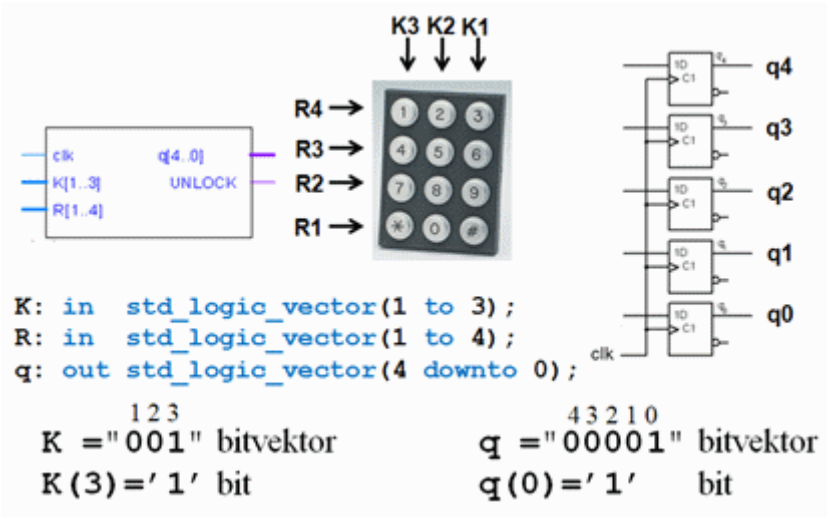

```
entity
architecture
     next_state_decoder:
     output_decoder:
     state_registers:
```

entity



Programmets `entity` är en beskrivning av kodlåset som en "black box" med insignaler och utsignaler.

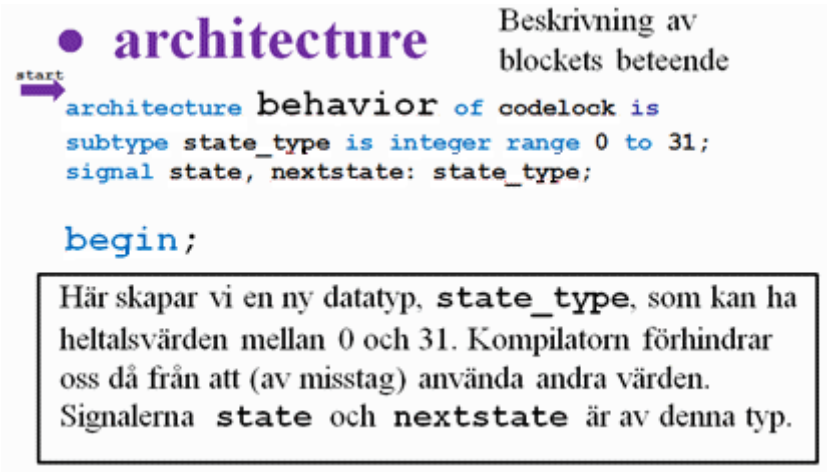
Bitar och Bitvektorer



Genom att välja datatyper som stämmer överens med problemet, blir det mindre risk för misstag. Man kan tex. anpassa indexeringen av variabler så att den överensstämmer med hur detta anges i datablad, på så sätt minimerar man risken att man "skriver av" fel.

Architecture - egendefinierade datatyper

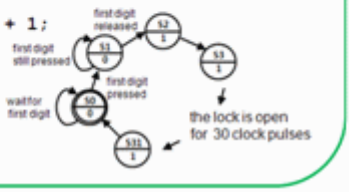
Avsnittet `architecture` innehåller beskrivningen av blockets beteende.



Vi skapar här en egendefinierad datatyp `state_type` som passar för att beskriva Moore-automatens tillstånd. Fördelen med detta är att kompilatorns felutskrifter kan hjälpa oss, om vi råkar använda värden utanför variablernas definitionsområden.

Architecture - tillståndsavkodaren

```
nextstate_decoder: -- next state decoding part
process(state, K, R)
 begin
   case state is
     when 0 => if (K = "001" and R ="0001")      then nextstate <= 1;
               else nextstate <= 0;
               end if;
     when 1 => if (K = "001" and R ="0001")      then nextstate <= 1;
               elsif (K = "000" and R = "0000")  then nextstate <= 2;
               else nextstate <= 0;
               end if;
     when 2 to 30 => nextstate <= state + 1;
     when 31      => nextstate <= 0;
  end case;
end process;
```



Nästa tillstånds avkodaren är programmets centrala del. Genom att använda `case`-satsen kan man skriva koden så att den helt följer tillståndsdiagrammet.

Felsökningshjälpmedel - state som visas med fem lysdioder

```
debug_output:  -- display the state
q <= conv_std_logic_vector(state,5);
```

Funktionen **`conv_std_logic_vector()`** omvandlar **`state`** (ett heltal mellan 0…31) till en 5-bitars bitvektor **`q`**, **`q(4)`** … **`q(0)`**.

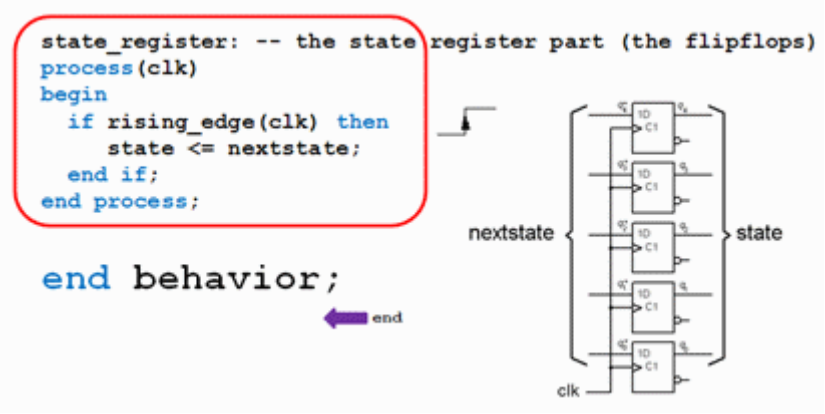
För att kunna felsöka hårdvaran, labutrustningen, vill vi kunna följa vilket tillstånd automaten befinner sig i. Funktionen `conv_std_logic_vector()` omvandlar `state` (ett heltal mellan 0...31) till en 5-bitars bitvektor `q`, `q(4)` ... `q(0)`. För att kunna använda denna konverteringsfunktion behöver man ta med biblioteket `IEEE.std_logic_arith.all`.

Utgångsavkodaren

```
output_decoder: -- output decoder part
process(state)
begin
  case state is
     when 0 to 1  => UNLOCK <= '0';
     when 2 to 31 => UNLOCK <= '1';
  end case;
end process;
```

Utgångsavkodaren är skriven "rakt på sak" med ett `case`-uttryck.

Tillståndsregistret



Genom att vi använder funktionen `rising_edge(clk)` "förstår" kompilatorn att vi vill utnyttja vipporna som finns i MAX-kretsen för att bygga ett register.

William Sandqvist `william@kth.se`