FYS4220/9220

UNIVERSITY OF OSLO

State machines and large designs

Lecture #4

Reading: chapter 5, 9 - 9.1.2, 9.2 and 3.5 in Zwolinski

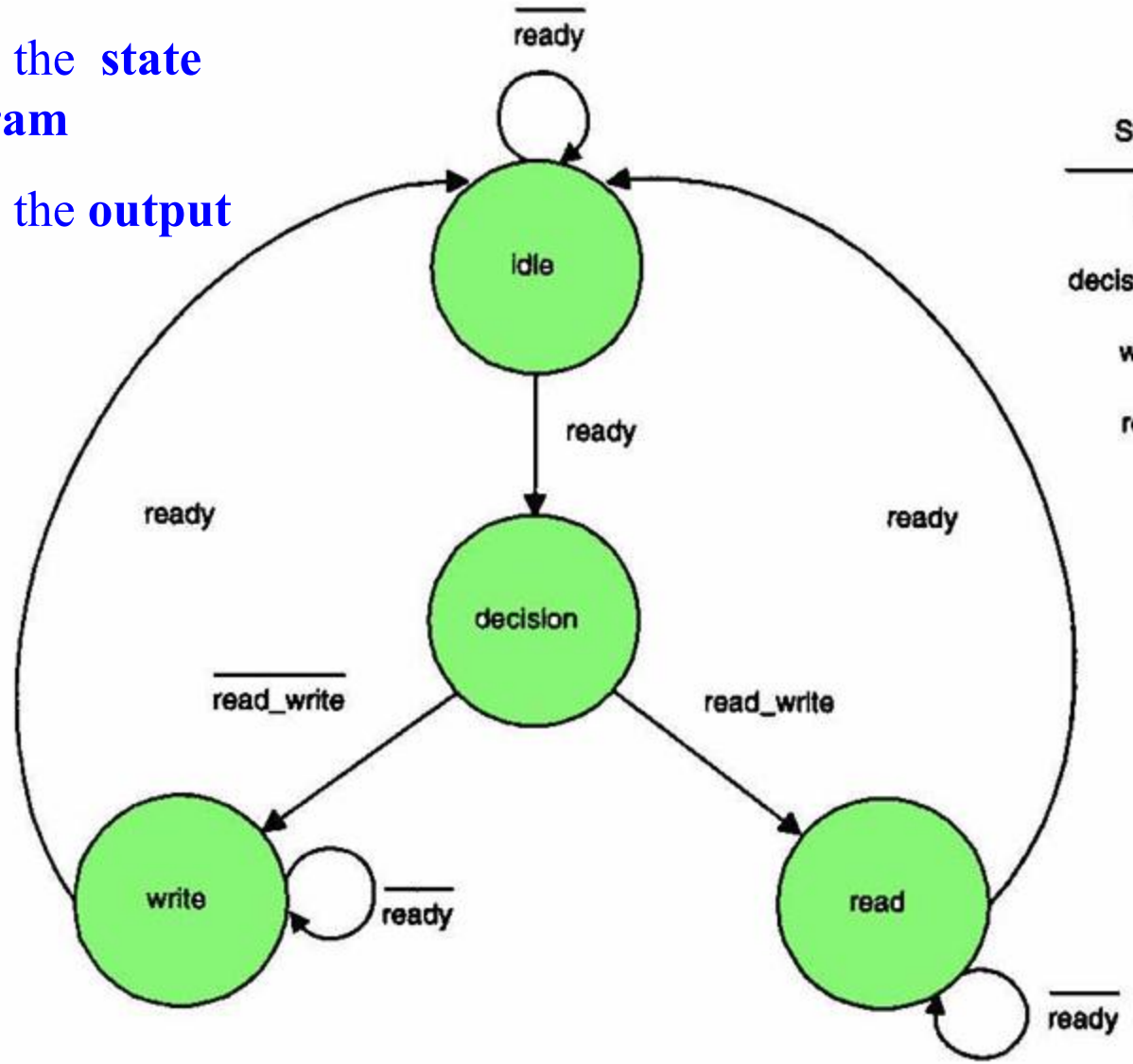
J. K. Bekkeng, 3.7.2011

UNIVERSITY
OF OSLO

FSMs- Finite State Machines

Traditional method:

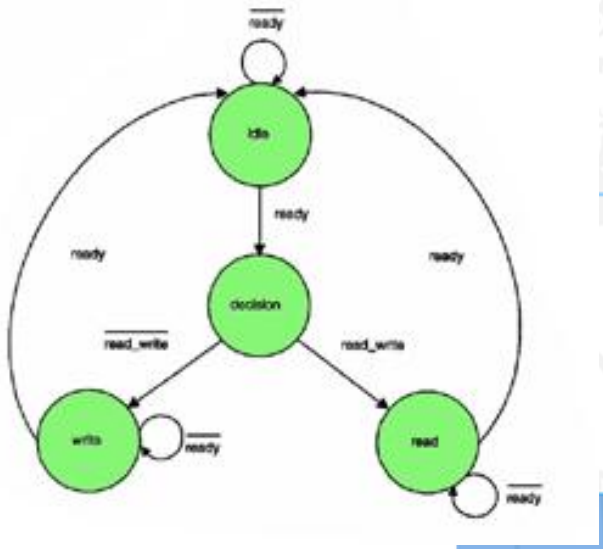
1) Make the **state diagram**
2) Make the **output table**



State	Outputs oe	we
idle	0	0
decision	0	0
write	0	1
read	1	0

Figure 5-1 Simple state machine

3) Make the **state table** from the state diagram and the output table

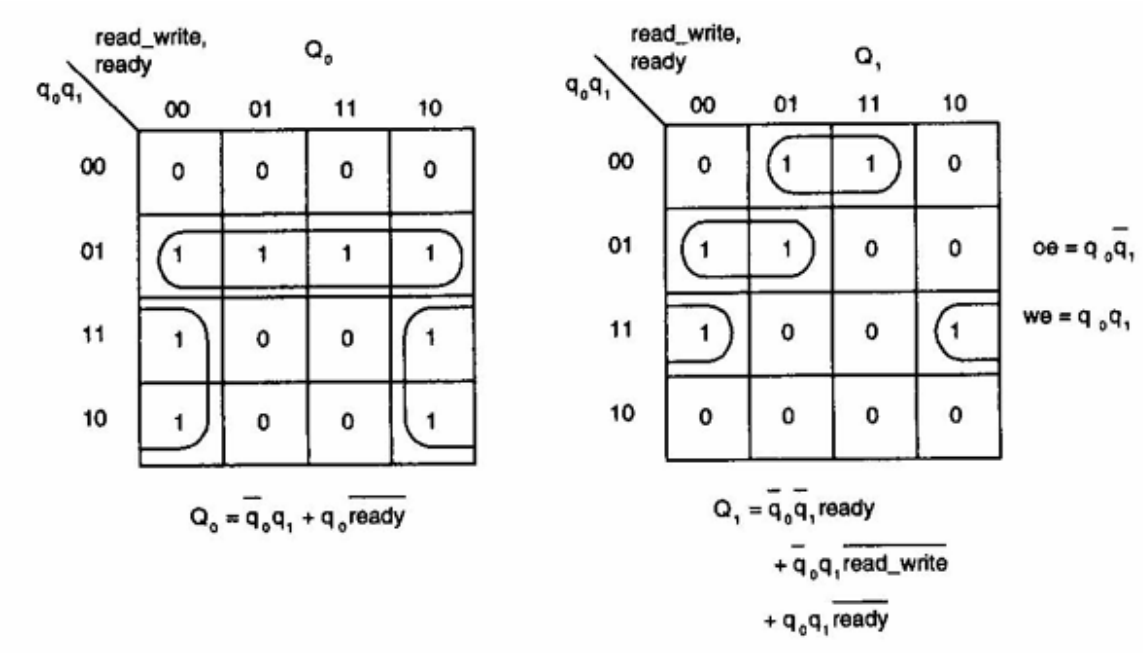


Present state and input

Next state and output

PS		read_write / ready		NS Q_0Q_1		Outputs	
State	q_0q_1	00	01	11	10	oe	we
idle	00	00	01	01	00	0	0
decision	01	11	11	10	10	0	0
write	11	11	00	00	11	0	1
read	10	10	00	00	10	1	0

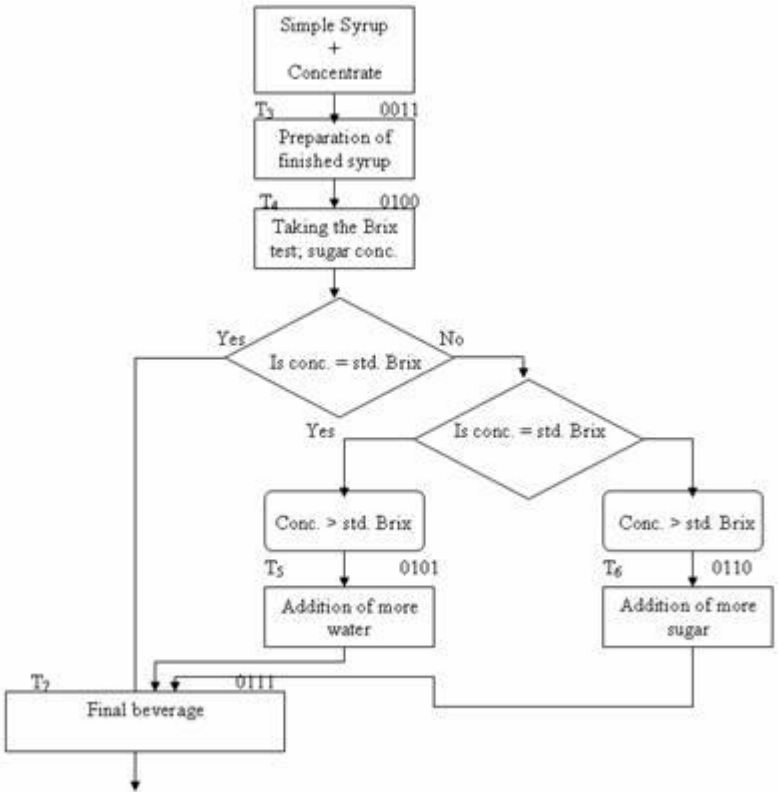
- Write the Boolean equations from the state table
- Use **Karnaugh diagrams** to simplify the Boolean equations



FSM design with VHDL

UNIVERSITY OF OSLO

Algorithmic state machine (**ASM**) chart (Flytskjema/Flowchart)



State diagram

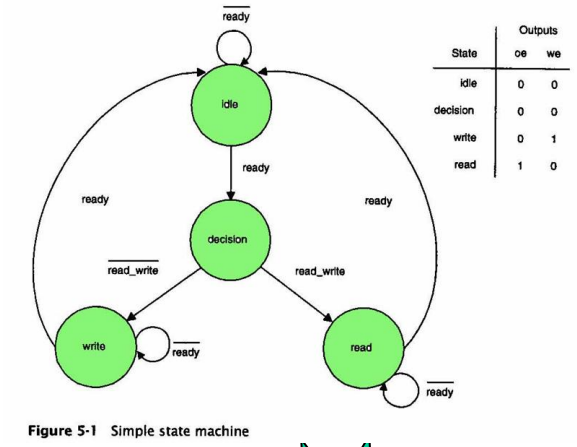


Figure 5-1 Simple state machine

or

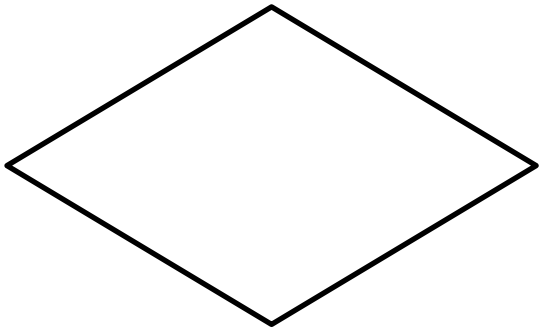
Code the FSM in VHDL

Easy to code the FSM (from the ASM) using VHDL!

Flowchart elements

UNIVERSITY OF OSLO

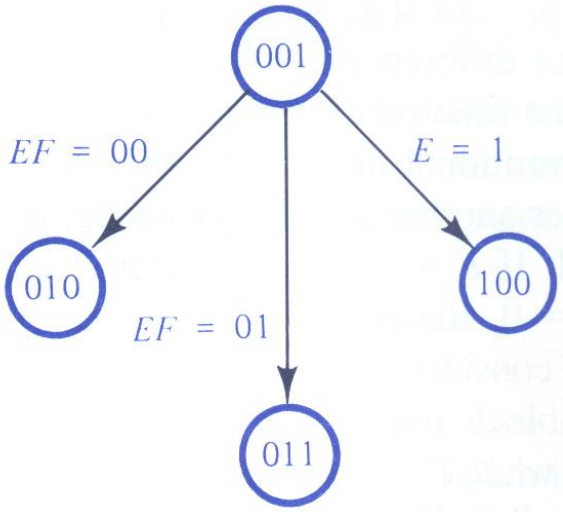
Tilstandsboks (State box)

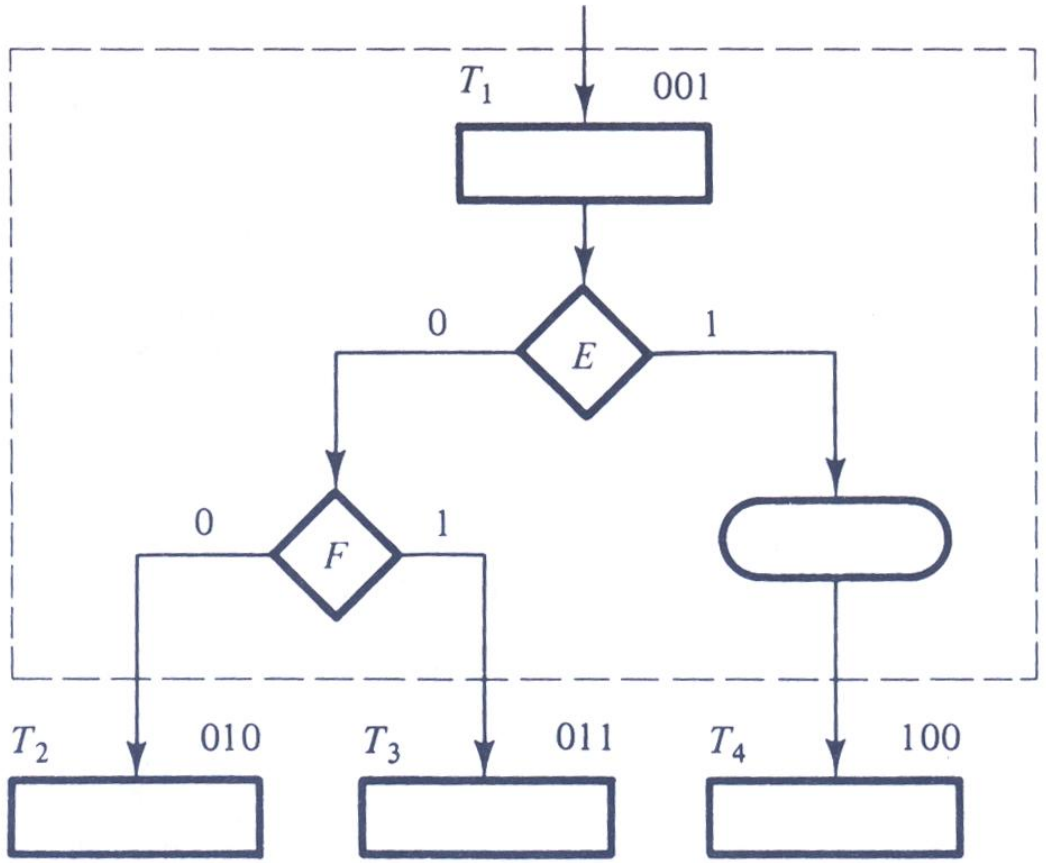


Valgboks (decision box)

Betingelsesboks (Conditional output box)

State diagram vs ASM chart



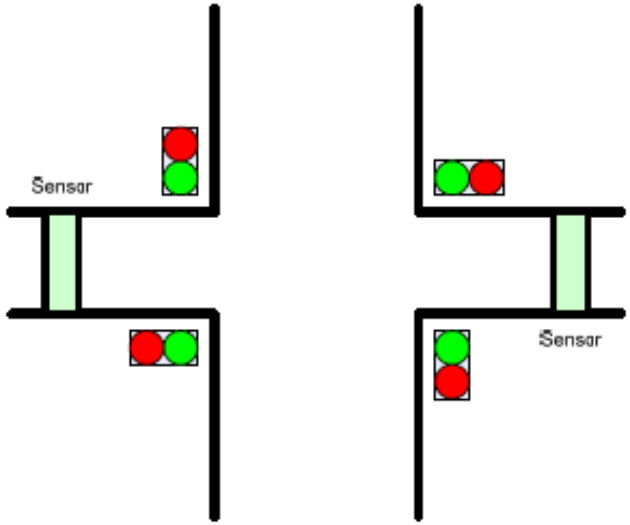


Eksempel – Styring av trafikklys

UNIVERSITY OF OSLO

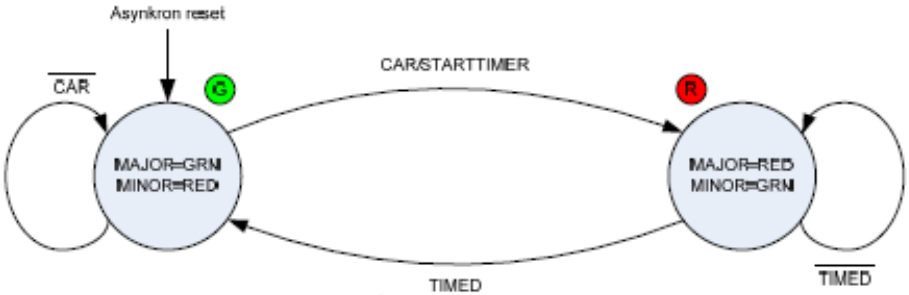
- Eksempel: Enkel trafikklysstyring mellom en hovedvei og to mindre sideveier

- Virkemåte:
  - Sensorer detekterer at det kommer en bil på en sidevei.
  - Gir sideveien grønt lys/hovedvei rødt
  - Sideveien har grønt lys i en viss tid gitt av en timer

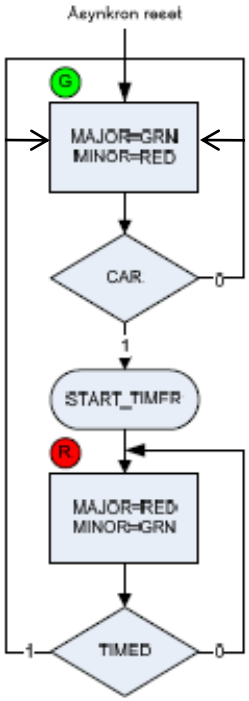


See example in chapter 5.3

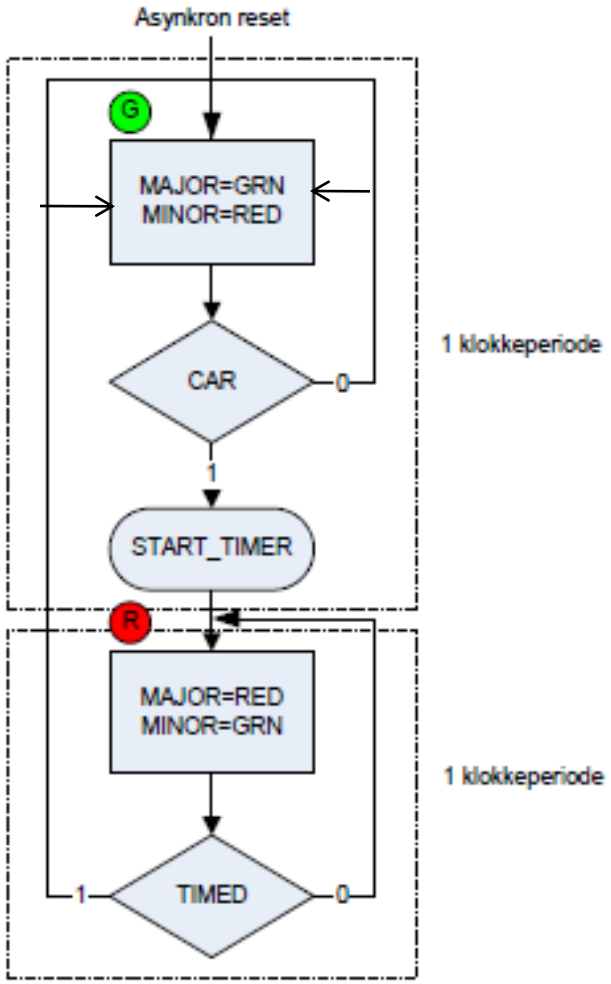
Beskrivelse av tilstandsmaskiner



Tilstandsdiagram



ASM (Algorithmic State Machine) diagram

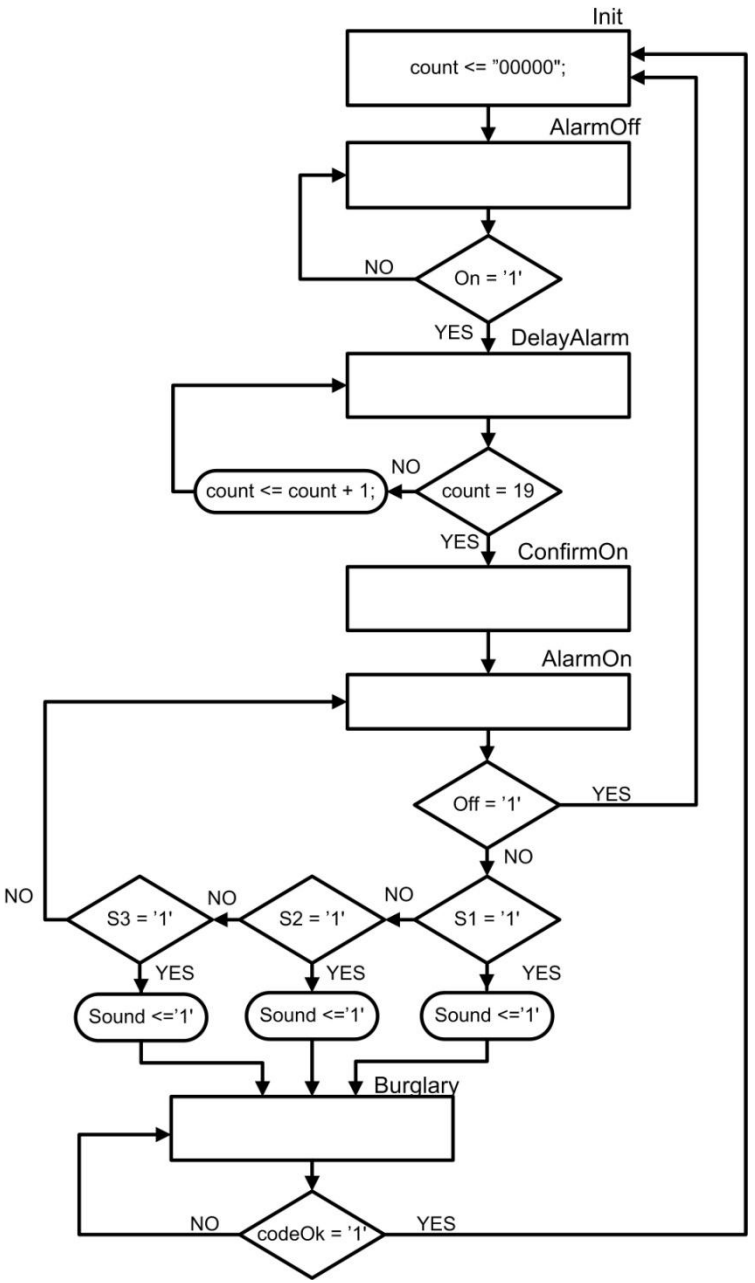


The ASM chart contains implicit timing information !

- En tilstand varer fra en tilstandsboks til neste.
- Dette tilsvarer en klokkeperiode

ASM chart guidelines

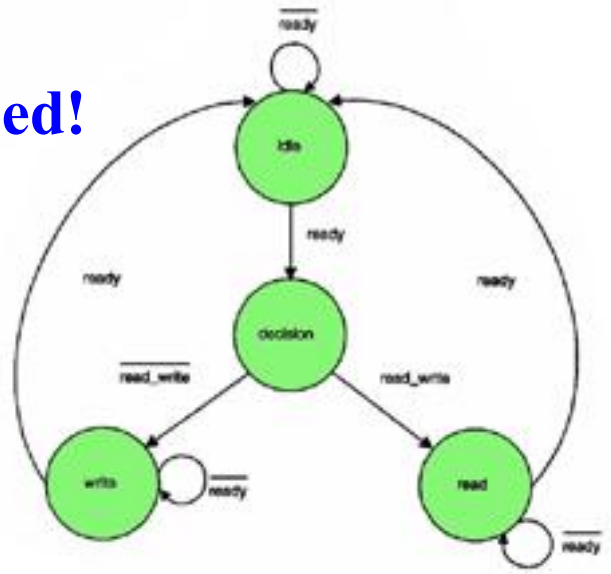
- Yes/No (1/0) labels together with decision boxes
- Arrows to show the program flow
- State names in top right/left corner of the state boxes
- Selected outputs listed in conditional output boxes and inside the state boxes
- Test condition listed in the decision boxes



State machines in VHDL

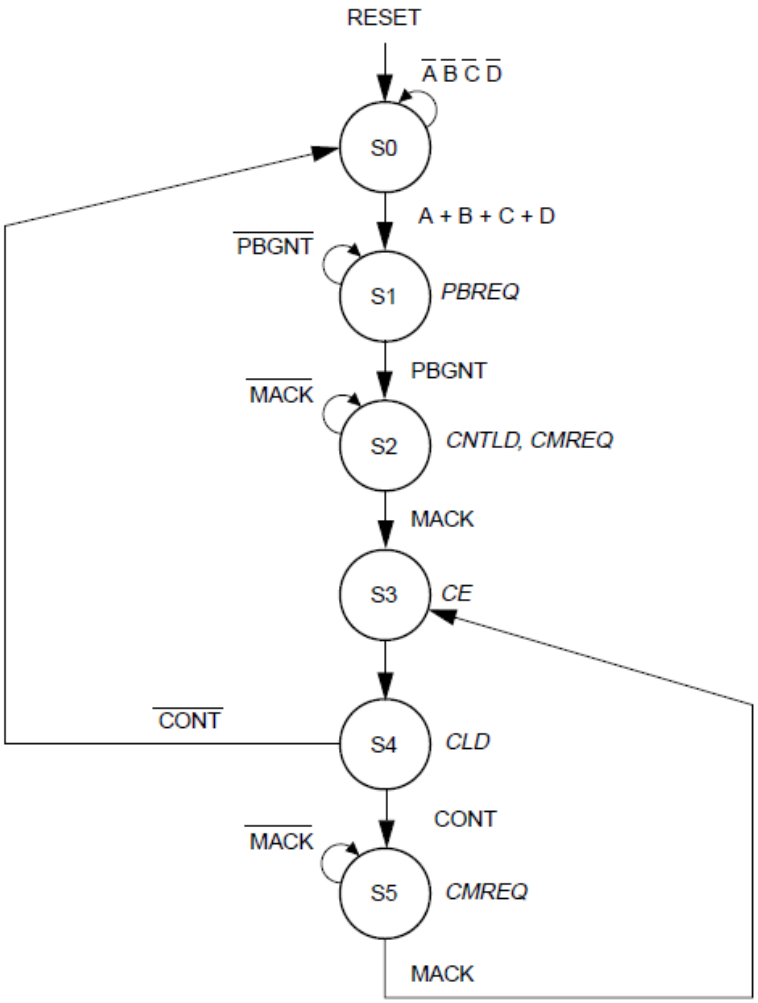
- An ASM chart (or a state diagram) can easily be translated to a VHDL description!

- In VHDL the state machine can be described in two different ways:
  - 1-process FSM ← **Recommended!**
  - 2-process FSM
    - One process describes the combinational logic, and another describes synchronization of state transitions to the clock



Example II

- The state S0 can be realized when state S4 is asserted and the input CONT is low, or it remains at state S0 if all four inputs—A, B, C, and D—are low.
- Similarly, state S1 can be achieved when state S0 is asserted and any one of the four inputs—A, B, C, or D—is high, or it remains at the same state (S1) if the input PBGNT is low



VHDL code Example II

2-process FSM

```
--library ieee;
--use ieee.std_logic_1164.all;
entity DMASM is
        port (A,B,C,D:  in bit;
              PBGNT, MACK, CONT :  in bit;
              RST, CLK :  in bit;
              PBREQ, CMREQ, CE, CNTLD, CLD :  out bit);
        end DMASM;
architecture BEHAVE of DMASM is
        type STATE is (S0, S1, S2, S3, S4, S5);
        signal CURRENT_STATE, NEXT_STATE: STATE;
begin
SEQ: process (RST, CLK)
begin
        if (RST = '0') then
                CURRENT_STATE <= S0;
        elsif (CLK' event and CLK = '1' ) then
                CURRENT_STATE <= NEXT_STATE;
        end if;
end process;
COMB: process (CURRENT_STATE, A, B, C, D, PBGNT, MACK, CONT)
begin
  PBREQ <= '0';
  CMREQ <= '0';
  CE    <= '0';
  CNTLD <= '0';
  CLD   <= '0';
case CURRENT_STATE is
    when S0 =>
        if (A = '1' or B = '1' or C = '1' or D = '1') then
        NEXT_STATE <= S1;
        else
        NEXT_STATE <= S0;
        end if;
    when S1 => PBREQ <= '1';
        if (PBGNT = '1') then
        NEXT_STATE <= S2;
        else
        NEXT_STATE <= S1;
        end if;
    when S2 => CNTLD <= '1'; CMREQ <= '1';
        if (MACK = '1') then
        NEXT_STATE <= S3;
        else
        NEXT_STATE <= S2;
        end if;
    when S3 => CE <= '1';
        NEXT_STATE <= S4;
    when S4 => CLD <= '1';
        if (CONT = '1') then
        NEXT_STATE <= S5;
        else
        NEXT_STATE <= S0;
        end if;
    when S5 => CMREQ <= '1';
        if (MACK = '1') then
        NEXT_STATE <= S3;
        else
        NEXT_STATE <= S5;
        end if;
    end case;
  end process;
end BEHAVE;
```

Synchronization to the clock

combinational logic

Example: 2 process FSM

UNIVERSITY OF OSLO

```
library ieee;
use ieee.std_logic_1164.all;
entity memory_controller is port (
    reset, read_write, ready, burst, clk    : in std_logic;
    bus_id                                  : in std_logic_vector(7 downto 0);
    oe, we                                  : out std_logic;
    addr                                    : out std_logic_vector(1 downto 0));
end memory_controller;

architecture state_machine of memory_controller is
        type StateType is (idle, decision, read1, read2, read3, read4, write);
        signal present_state, next_state : StateType;
begin
state_comb:process(reset, bus_id, present_state, burst, read_write, ready) begin
        if (reset = '1') then
                oe <= '-'; we <= '-'; addr <= "--";
                next_state <= idle;
        else
                case present_state is
                        …
                end case;
        end if;
end process state_comb;

state_clocked:process(clk) begin
        if rising_edge(clk) then
                present_state <= next_state;
        end if;
end process state_clocked;
end;
```

Combinational logic

Synchronous logic

Asynchronous reset in 2-process FSM

```
state_clocked:process(clk,reset) begin
   if reset= '1' then
      present_state <= idle;
   elsif rising_edge(clk) then
      present_state <= next_state;
   end if;
end process state_clocked;
```

1-process FSM

UNIVERSITY OF OSLO

Functionally identical to the 2-process FSM, and the same logic is produced

```
architecture state_machine of memory_controller is
   type StateType is (idle, decision, read1, read2, read3, read4, write);
   signal state : StateType;
begin
state_tr:process(reset, clk) begin          -- one process fsm
   if reset = '1' then                      -- asynchronous reset
      state <= idle;
   elsif rising_edge(clk) then              -- synchronization to clk
      case state is                         -- state transitions defined
         when idle    =>
            if (bus_id = "11110011") then
               state <= decision;
            else                            -- not req'd; for clarity
               state <= idle;
            end if;
         when decision=>
            if (read_write = '1') then
               state <= read1;
            else                            --read_write='0'
               state <= write;
            end if;
```

UNIVERSITY OF OSLO

Example: use of state machine for ADC (Analog to Digital Converter) control

ADC data sheet

VHDL code

Output from a state machine

UNIVERSITY OF OSLO

1)

```
FSM: process (reset, clk)
begin
  if (reset = '1') then
  state <= idle;

 elsif rising_edge (clk) then
   case state is
     when idle =>
       oe <= '1';
       if (input1 = '1') then
          state <= s1;
       else
          state <= idle;
       end if;
     when s1 =>
       oe <= '0';

     .......
```

Recommended!

2)

```
FSM: process (reset, clk)
begin
  if (reset = '1') then
  state <= idle;

 elsif rising_edge (clk) then
  oe <= '1'; -- Default value
  case state is
    when idle =>
       if (input1 = '1') then
          state <= s1;
       else
          state <= idle;
       end if;
    when s1 =>
      oe <= '0';

     .......
```

3)

```
FSM: process (reset, clk)
begin
  if (reset = '1') then
  state <= idle;

 elsif rising_edge (clk) then
   case state is
     when idle =>
        if (input1 = '1') then
           state <= s1;
        else
           state <= idle;
        end if;
     when s1 =>
     .......
  end process;
with state select
  oe <= '1' when idle,
        '0' when S1;
end state_machine;
```

Example: Use of default values in FSMs

```
FSM_CONF_READ:
process(clk)
begin
  if falling_edge(clk) then

    -- set default values
    cs        <= '0';
    sr_enable <= '0';
    DataReady <= '0';
    Din       <= '0';


    case state is
      when PowerUp =>
```

State encoding and enumeration type

UNIVERSITY OF OSLO

If the number of states is not 2^n there will be undefined states!

```
type StateType is (idle, decision, read1, read2, read3,
                   read4, write);
signal state : StateType;
```

Table 5-8 Sequential encoding for an enumeration type

State	q_0	q_1	q_2
idle	0	0	0
decision	0	0	1
read1	0	1	0
read2	0	1	1
read3	1	0	0
read4	1	0	1
write	1	1	0
undefined	1	1	1

Seven states

Added state; $2^3 = 8$

FSM fault tolerance

UNIVERSITY OF OSLO

- **If the state machine gets into an undefined (illegal) state it will be unpredictable** (can e.g. stay in the illegal state forever)
- In safety critical systems it is absolutely necessary to always get to known states
- Demands additional logic to get out of illegal states

```
type StateType is (idel, decision, read1, read2, read3, read4,
                   write, undefined);
```

Include a state name in the enumerated type for each undefined state

```
case present_state is
      ...
    when undefined  => next_state <= idle;
end case;
```

Specify the state transition to get out of illegal states

UNIVERSITY OF OSLO

Example with 3 undefined states

```
type states is (s0, s1, s2, s3, s4, u1, u2, u3);
signal next_state, present_state: states;
...
case present_state is
   ...
   when others => next_state <= s0;
end case;
```

One-Hot encoding of FSM

- Uses one bit for each state
- A state machine with n states needs n flip-flops to store the states
- Only one flip-flop (bit) assigned to ’1’, indicating present state
- Output logic and next state logic is very simple
- Used in FPGAs (due to many flip-flops)

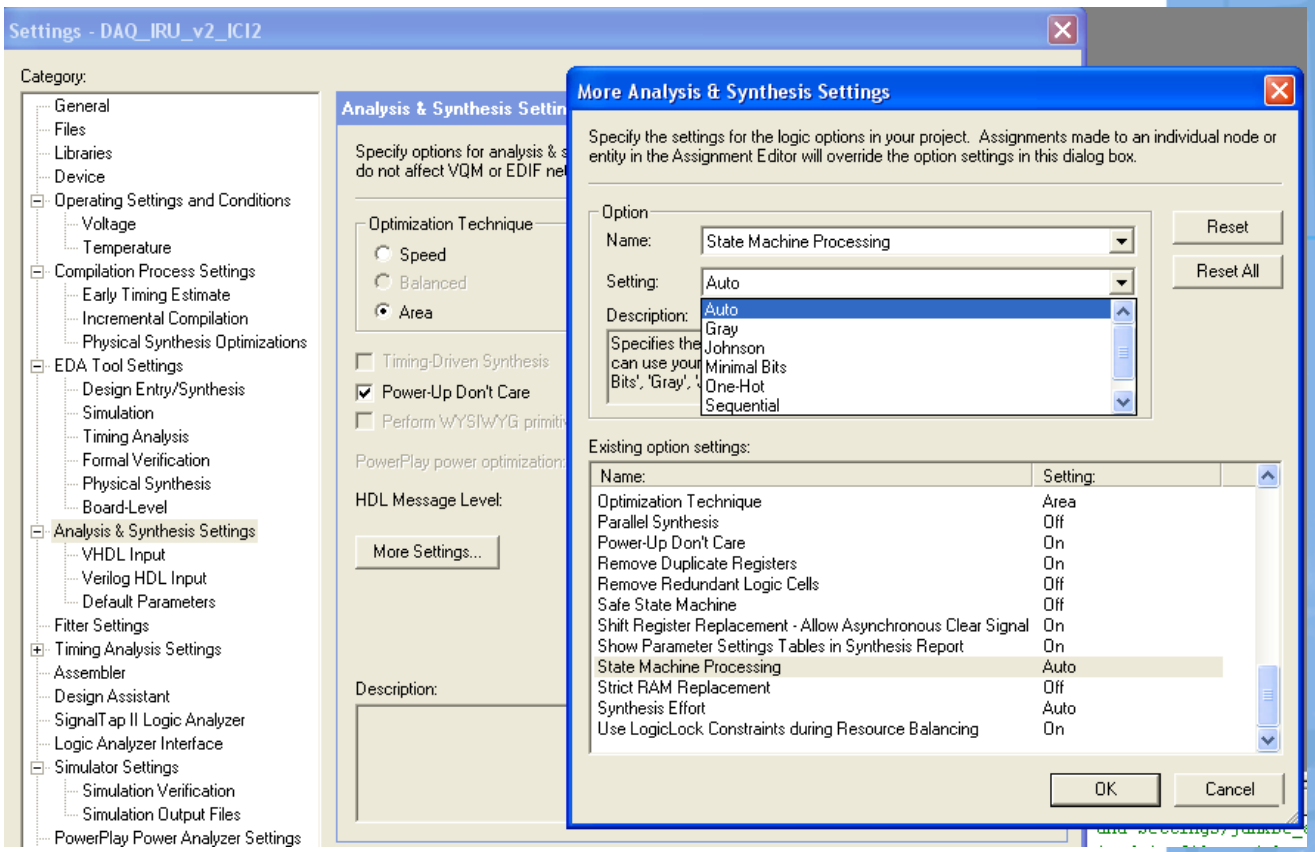
State	Sequential	One-Hot
state0	00000	000000000000000001
state1	00001	000000000000000010
state2	00010	000000000000000100
state3	00011	000000000000001000
state4	00100	000000000000010000
state5	00101	000000000000100000
state6	00110	000000000001000000
state7	00111	000000000010000000
state8	01000	000000000100000000
state9	01001	000000001000000000
state10	01010	000000010000000000
state11	01011	000000100000000000
state12	01100	000001000000000000
state13	01101	000010000000000000
state14	01110	000100000000000000
state15	01111	001000000000000000
state16	10000	010000000000000000
state17	10001	100000000000000000

5 18

State machine with 18 states

One-Hot encoding setup in Quartus II

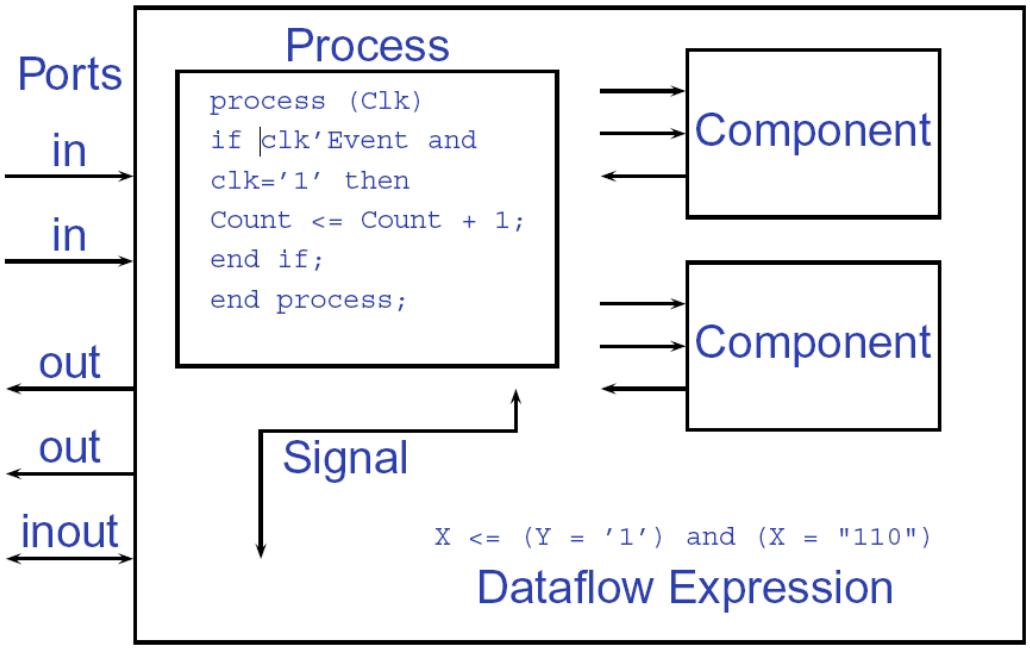
- No change in the VHDL code
- Setup in Quartus II before synthesis (**Assignments – Settings Analysis & Synthesis Settings – More Settings**)

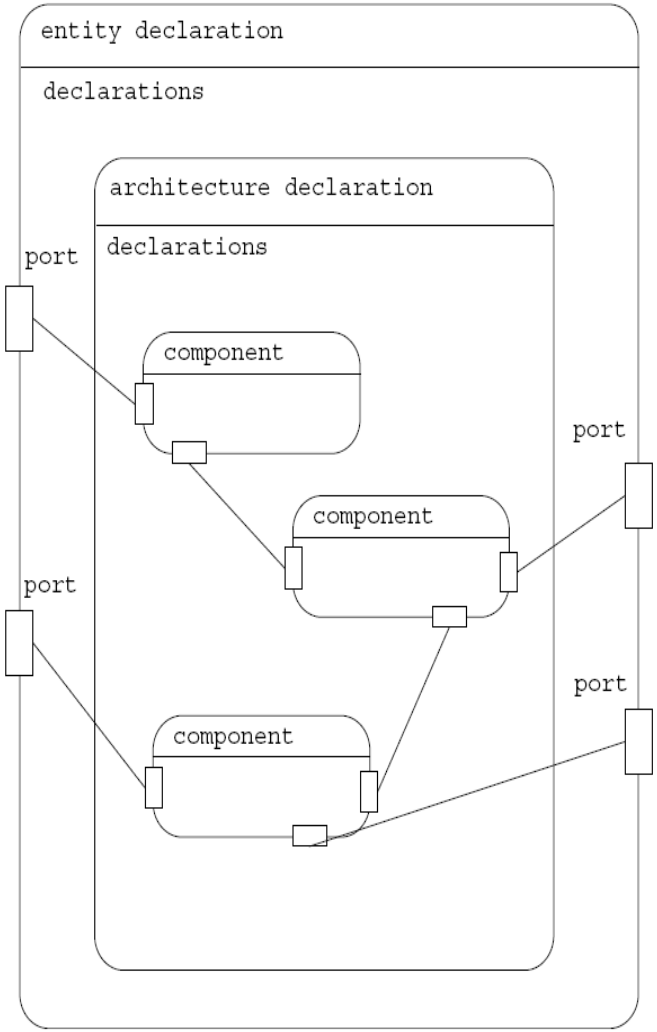


UNIVERSITY
OF OSLO

”Large designs - Hierarchy”

Block diagrams





Hvordan gå frem for å skrive VHDL kode

UNIVERSITY OF OSLO

- **Tegn et forenklet blokkskjema**
  - ytre grensesnittet, i form av innsignaler og utsignaler, tilsvarer *entity-deklarasjonen*
  - indre strukturen tilsvarer VHDL-beskrivelsens *architecture-del.*
- Det er ofte hensiktsmessig å dele opp den indre strukturen i en *dataveistruktur og en kontrollstruktur.*
- Dataveistrukturen inneholder elementer som registre, addere, multipleksere, som er sammenkoblet med databusser. Denne strukturen er velegnet til å beskrives i blokkskjema. ***Hvis strukturen er kompleks, lønner det seg å dele den opp i mindre blokker***
- Kontrollogikkens innsignaler og utsignaler kan beskrives i blokkskjema sammen med dataveistrukturen. Den indre oppbyggingen beskrives best i form av: **sannhetsverditabeller, boolske ligninger og ASM flyt-skjema/tilstandsdiagrammer**
- **Deretter kodes den strukturerte løsningen i VHDL**
- **Med utgangspunkt i en veldokumentert struktur skal det normalt være en grei jobb å lage en fungerende kode.**

Example of a block diagram

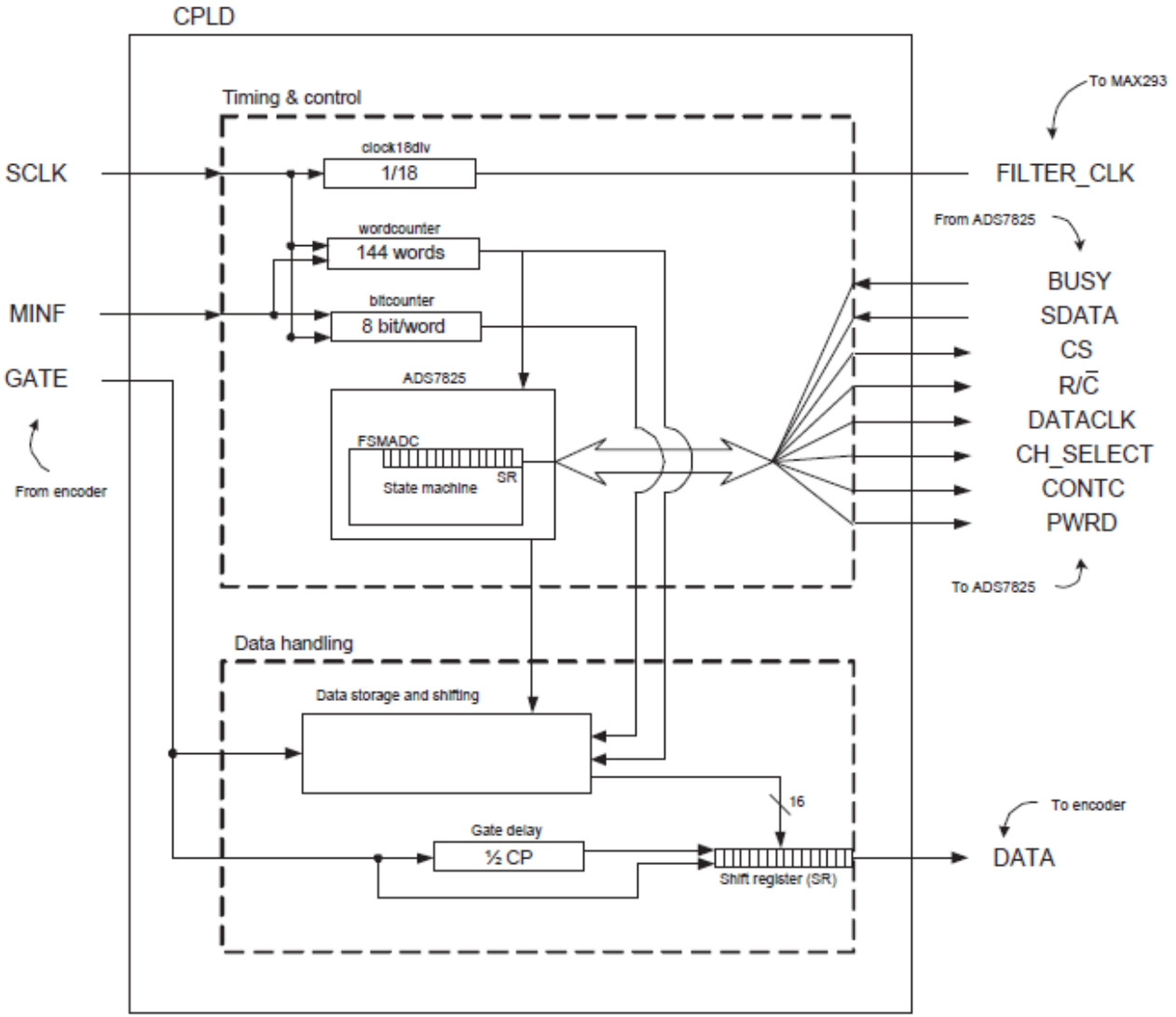


Figure 5.18: Block diagram for the digital logic

Building blocks

UNIVERSITY OF OSLO

- Library
  - library IEEE
  - use IEEE.std_logic_1164.all;

To make the library visible

- Packages
  - use work.my_package.all; -- your own package
  - **work** is the directory where the design files are located
- Components
  - An entity used in another entity
  - Needs a component declaration to make the component visible
  - Needs a component instantiation to connect the component to the top entity – using ***port map()***

Component declarations in a package

- A component declaration can be placed in a package
- To use the component in the design the package must be included in the design
- The package must have its own "**library** and **use**"
- The package is usually in a separate file, (e.g. ***my_package.vhd***), and the components are in separate files as well (e.g. dflop.vhd)

Without using a package:

```
architecture arch_test_dff of test_dff is
-- Component declaration
component dflop
port(
 d, clk  : in   std_logic
 q        :out std_logic);
end component;

-- Declaration of internal signals
signal temp : std_logic;

begin
```

Package example

UNIVERSITY OF OSLO

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
package regs_pkg is
   component rdff1 port (
       clk, reset: in std_logic;
       d:          in std_logic;
       q:          buffer std_logic);
   end component;
   component rdff
       generic (size: integer := 2);
       port (clk, reset: in std_logic;
             d:          in std_logic_vector(size-1 downto 0);
             q:          buffer std_logic_vector(size-1 downto 0));
   end component;
   component rreg1 port(
           clk, reset, load:in std_logic;
           d:               in std_logic;
           q:               buffer std_logic);
   end component;
end regs_pkg;
```

A larger design using a package

UNIVERSITY OF OSLO

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
use work.am2901_comps.all;
entity am2901 is port(
   clk, rst:    in std_logic;
   a, b:        in unsigned(3 downto 0);      -- address inputs
   d:           in unsigned(3 downto 0);      -- direct data
   i:           in std_logic_vector(8 downto 0);
                                              -- micro instruction
   c_n:         in std_logic;                 -- carry in
   oe:          in std_logic;                 -- output enable
   ram0, ram3:  inout std_logic;              -- shift lines to ram
   qs0, qs3:    inout std_logic;              -- shift lines to q
   y:           buffer unsigned(3 downto 0); -- data outputs (3-state)
   g_bar,p_bar: buffer std_logic;             -- carry generate, propagate
   ovr:         buffer std_logic;             -- overflow
   c_n4:        buffer std_logic;             -- carry out
   f_0:         buffer std_logic;             -- f = 0
   f3:          buffer std_logic);            -- f(3) w/o 3-state
end am2901;
architecture am2901 of am2901 is
   alias dest_ctl: std_logic_vector(2 downto 0) is i(8 downto 6);
   alias alu_ctl:  std_logic_vector(2 downto 0) is i(5 downto 3);
   alias src_ctl:  std_logic_vector(2 downto 0) is i(2 downto 0);
   signal ad, bd:unsigned(3 downto 0);
   signal q:unsigned(3 downto 0);
   signal r, s:unsigned(3 downto 0);
   signal f:unsigned(3 downto 0);
begin
-- instantiate and connect components
u1: ram_regs port map(clk => clk, rst => rst, a => a, b => b, f => f,
               dest_ctl => dest_ctl, ram0 => ram0, ram3 => ram3,
               ad => ad, bd => bd);
u2: q_reg port map(clk => clk, rst => rst, f => f, dest_ctl => dest_ctl,
               qs0 => qs0, qs3 => qs3, q => q);
u3: src_op port map(d => d, ad => ad, bd => bd, q => q,
               src_ctl => src_ctl, r => r, s => s);
u4: alu port map(r => r, s => s, c_n => c_n, alu_ctl => alu_ctl,
               f => f, g_bar => g_bar, p_bar => p_bar,
               c_n4 => c_n4, ovr => ovr);
u5: out_mux port map(ad => ad, f => f, dest_ctl => dest_ctl,
               oe => oe, y => y);
-- define f_0 and f3 outputs
f_0 <= '0' when f = "0000" else 'Z';
f3 <= f(3);
end am2901;
```

Component declaration

Component instantiation
(Komponenttilordninger)

Listing 6-17 AM2901 Top level

Generics

UNIVERSITY OF OSLO

- The width of a signal can be specified using a parameter
- Useful for **registers** and **counters** with different number of bits; only necessary to make one component

Default value

```
library ieee;
use ieee.std_logic_1164.all;
entity rdff is
   generic (size: integer := 2);
   port (  clk, reset:in std_logic;
      d:               in std_logic_vector(size-1 downto 0);
      q:               buffer std_logic_vector(size-1 downto 0));
end rdff;
```

```
architecture archrdff of rdff is
begin
p1:    process (reset, clk) begin
          if reset = '1' then
             q <= (others => '0');
          elsif (clk'event and clk='1') then
             q <= d;
          end if;
       end process;
end archrdff;
```

Generic counter

UNIVERSITY OF OSLO

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity ascount is
    generic (CounterSize: integer := 2);
    port(clk, areset, sreset, enable: in std_logic;
        count:buffer std_logic_vector(counterSize-1 downto 0));
end ascount;
architecture archascount of ascount is
begin
p1: process (areset, clk) begin
    if areset = '1' then
        count <= (others => '0');
    elsif (clk'event and clk='1') then
        if sreset='1' then
            count <= (others => '0');
        elsif enable = '1' then
            count <= count + 1;
        else
            count <= count;
        end if;
    end if;
  end process;
end archascount;
```

Using Generic components

- **generic map** is used to assign a value to a generic component
- If the **generic map** is not specified the default value which is specified in the component is used

Example:

Name of the component

Set the value of the *CounterSize* parameter

```
U1: ascount
generic map (3)          -- or generic map (CounterSize  => 3)
port map (clk, areset, sreset,  enable, count);
```

Generics Example

UNIVERSITY OF OSLO

- The width of a signal can be specified using a parameter
- Useful for **registers** and **counters** with different number of bits; only necessary to make one component

Entity using a generic

```
entity SR_SerIn_redge is
  generic (
    width : integer := 16);


  port
    (
    clk      : in  std_logic;
    DataIn   : in  std_logic;
    shift_en : in  std_logic;
    DataOut  : out std_logic_vector(width-1 downto 0)
    );

end SR_SerIn_redge;
```

How to use this component with a generic

```
component  SR_SerIn_redge
  generic (width : integer);

  port
    (
    clk      : in  std_logic;
    DataIn   : in  std_logic;
    shift_en : in  std_logic;
    DataOut  : out std_logic_vector(width-1 downto 0)
    );
end component;



SR_DATA: SR_SerIn_redge
  generic map (
    width => 12)

  port map (sclk, Dout, sr_enable, Pdata );
```