Finite State Machines

- A finite state machine is a sequential logic circuit which moves between a finite set of states, dependent upon the values of the inputs and the previous state. The state transitions are synchronized on a clock.

There are many ways to describe a finite state machine in VHDL. The most convenient is with a process statement.
The state of the machine can be stored in a variable or signal, and the possible states conveniently represented with an enumeration type.

```vhdl
architecture Explicit of FSM is
begin
    process
        type StateType is (Idle, Start, Stop, Clear);
        variable State: StateType;
    begin
        wait until RISING_EDGE(Clock);
        if Reset = '1' then
            State := Idle; F <= '0'; G <= '1';
        else
            case State is
                when Idle  => State := Start; G <= '0';
                when Start => State := Stop;
                when Stop  => State := Clear; F <= '1';
                when Clear => State := Idle;  F <= '0';
                                           G <= '1';
            end case;
        end if;
    end if;
end process;
end;
```
State Names

Using enumeration type allows you to give symbolic names to the states, but say nothing about the hardware implementation.

You should choose meaningful names, as this makes the VHDL code easy to understand. The names will also be visible during simulation, which makes debugging easier.

In practice, it is important that finite state machines are initialized by means of an explicit reset signal.

Otherwise, there is no reliable way to get the VHDL and gate level representations of the FSM into the same known state, and thus no way to verify their equivalence.
architecture Explicit of FSM is
begin
  process
    type StateType is (Idle, Start, Stop, Clear);
    variable State: StateType;
    begin
      wait until RISING_EDGE(Clock);
      if Reset = '1' then
        State := Idle; F <= '0'; G <= '1';
      else
        case State is
          when Idle => State := Start; G <= '0';
          when Start => State := Stop;
          when Stop  => State := Clear; F <= '1';
          when Clear => State := Idle; F <= '0';
            G <= '1';
        end case;
      end if;
    end process;
end;
The following description of an FSM consists of a process synchronized on a clock edge, and assigning the variable state (the state vector) and signal F and G (the outputs).

Thus, four flip flops will be synthesized, two for the state vector, and one each of the two outputs.
architecture Explicit of FSM is
begin
process

type StateType is (Idle, Start, Stop, Clear);
variable State: StateType;
begin
wait until RISING_EDGE(Clock);
if Reset = '1' then
  State := Idle; F <= '0'; G <= '1';
else
  case State is
    when Idle  => State := Start; G <= '0';
    when Start => State := Stop;
    when Stop  => State := Clear;
    when Clear => State := Idle;
  end case;
end if;
end process;
end;
architecture SeparateDecoding of FSM is
type StateType is (Idle, Start, Stop, Clear);
signal State: StateType;
begin
  Change_state: process
  begin
    wait until RISING_EDGE(Clock);
    if State = Clear or Reset = '1' then
      State <= Idle;
    elsif State = Idle then State <= Start;
    elsif State = Start then State <= Stop;
    else State <= Clear;
  end if;
end process;

Output: process (State)
begin
  F <= '0'; G <= '0';
  if State = Clear then F <= '1';
  elsif State = Idle then G <= '1';
  end if;
end process;
end;
architecture RegistersPlusLogic of FSM is
  type StateType is (Idle, Start, Stop, Clear);
signal State: StateType;
begin
  Registers: process begin
    wait until RISING_EDGE(Clock);
    if Reset = '0' then
      State <= Idle;
    else
      State <= NextState;
    end if;
  end process;
  C_logic: process (State) begin
    if State = Clear then NextState <= Idle;
      F <= '1';
    elsif State = Idle then NextState <= Start;
      G <= '1';
    elsif State = Start then NextState <= Stop;
    else
      NextState <= Clear;
    end if;
  end process;
end;
Concatenation

• The concatenation operator “&” is used to join together two arrays end to end to make one longer array

```
Signal A,B: STD_LOGIC_VECTOR(7 downto 0);
Signal F: STD_LOGIC_VECTOR(15 downto 0);
F<= A & B;
```
• Concatenation can also be used to concatenate arrays with single bits, or even bits with bits.

Signal A,B,C: STD_LOGIC;

Signal F: STD_LOGIC_VECTOR(3 downto 0);

F(3 downto 1) <= (A & B) & C;

F(3) = A
F(2) = B
F(1) = C
F(0) = unchanged
• Shift and rotate operations can be performed in one line by combining concatenation with slice names.

```vhdl
Signal Reg: STD_LOGIC_VECTOR(7 downto 0);
Reg <= Reg(6 downto 0) & '0';  -- Shift left one digit
Reg <= Reg(6 downto 0) & Reg(7);  -- Rotate left one digit
```
Shift and Rotate

- In VHDL '93 these operations can be performed also using the shift and rotate operators:
  - Shift left / right logical : sll srl
  - Shift left / right arithmatic : sla sra
  - Rotate left / right : rol ror
Signal Reg: STD_LOGIC_VECTOR(7 downto 0);

Reg <= Reg(6 downto 0) & ‘0’;

Reg <= Reg sll 1;  \hspace{1cm} \text{Shift left one digit}

Reg <= Reg(6 downto 0) & Reg(7);

Reg <= Reg rol 1;  \hspace{1cm} \text{Rotate left one digit}
Variable A: STD_LOGIC_VECTOR (3 downto 0);

\[ A := \text{"0110"}; \]

- What is the value of this expression?

\[ \text{A(0) \& '1' \& A(2 downto 1)} \]

\[ \text{"0111"} \]
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ADDER is
  port(A, B: in
       STD_LOGIC_VECTOR(7 downto 0);
  SUM: out
       STD_LOGIC_VECTOR(7 downto 0));
end;

architecture A1 of ADDER is
begin
  SUM <= A + B;
end;

• This is illegal because the operator “+” is not defined to work on type Std_logic_vector.
• The `Std_logic_vector` type is declared in the package `Std_logic_1164`.

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

• The package also declares a set of operators which work on the type.

• Defining operators to work on new types is called operator overloading.
• There are also some operators that are implicitly defined for all array types.
• These are the relational operators

\[
\begin{align*}
> \\
\geq \\
< \\
\leq \\
= \\
\neq
\end{align*}
\]
• “+” is not defined in std_logic_1164, but it can be defined for std_logic_vector and put in a package.

• All VHDL synthesis tools provide a package that overloads certain arithmetic operators on std_logic_vector or on related types.
• Package Numeric_Std includes new array types signed and unsigned to represent numbers and overloaded operators.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
entity ADDER is
  port(A, B: in UNSIGNED(7 downto 0);
      SUM: out UNSIGNED(7 downto 0));
end;
architecture A1 of ADDER is
begin
  SUM <= A + B;
end;
```
signal A, B, C: STD_LOGIC;

F <= (A \text{nand} B) \text{xor} (\text{not} C);

\begin{center}
\text{signal A, B: STD\_LOGIC\_VECTOR;}
\end{center}

F <= A + B ;
Synthesis of Arithmetic

- The synthesis of the arithmetic operators depend on the synthesis tool.
- Some tools will map the operators to discrete gates. Others will make use of macro-cells optimized for the target technology.
- A plus operator (+) can be implemented in hardware using a ripple carry or a carry look-ahead scheme. The implementation will be smaller or faster, depending on which one you choose.
\[ F \leq A + B ; \]

- Discrete Gates
  - Non-optimal connection of cells
- ADDER8
  - Optimized structure of cells
F <= A + B;

C1: ADDER8 port map (A,B,B);

Technology independent VHDL code

Vendor Specific VHDL code
• If your synthesis tool cannot map an arithmetic operator to an optimized macro-cell, you will need to instantiate the macro-cell directly in your VHDL code.

• This would make your code vendor specific.
• However, sometimes you are forced to compromise technology independence in favour of efficiency.
F <= A + B;
+ Constraints

Ripple Carry

Carry Look-ahead
C1: SMALL-ADDER … (Ripple Carry Adder)
C1: FAST_ADDER8 … (Carry Look-Ahead Adder)
Resource Sharing

- Resource sharing allows a single hardware to be shared by more than one VHDL operator
- Some tools share resources automatically.

```vhdl
process (A, B, C, D, K)
begin
  if K then
    Z <= A + B; -- 8 Bit
  else
    Z <= C + D;
  end if;
end process;
```
process (A, B, C, D, K) begin
  if K then
    Z <= A + B; -- 8 Bit
  else
    Z <= C + D;
  end if;
end process;
• If your tool does not do resource sharing, you must rewrite your code to achieve same results.

```vhdl
process (A, B, C, D, K)
variable V1, V2 : ...
begin
  if K then
    V1 := A;
    V2 := B;
  else
    V1 := C;
    V2 := D;
  end if;
  Z <= V1 + V2;
end process;
```
State Machines (Again!)
• A State Machine should not have registers at the output stage.
• In order to eliminate the output registers and synthesize a pure state machine, we must re-write the VHDL description.
• In order to synthesize the correct hardware, we must split the description into at least two processes.
Exercise

• What is the minimum number of processes necessary to describe this architecture for synthesis?

- 2 processes (AC) (BDE)
- 4 processes (AC) (B) (D) (E)
- 3 processes (A) (CE) (BD)
- 4 processes (A) (BC) (D) (E)
- 2 processes (ABCE) (D)
- 3 processes (A) (BCE) (D)
entity FSM1 is
port (Clock : in std_logic;
      SlowRAN: in std_logic;
      Read, Write: out std_logic);
end entity;

architecture RTL of FSM1 is
begin
  SEQ_abd_COMB: process
    type StateType is (ST_Read, ST_Write, ST_Delay);
    variable State:StateType := ST_Read;
    begin
      wait until rising_edge*Clock);
      case State is
        when ST_Read => Read <= '1';
               Write <= '0';
               State := ST_Write;
        when ST_Write => Read <= '0';
               Write <= '1';
               if (SlowRam = '1') then
                 State := ST_Delay;
               else
                 State := ST_Read
               end if;
        when ST_Delay => Read <= '0';
               Write <= '0';
               State := ST_Read;
        end case;
      end process SEQ_AND_COMB;
    end architecture RTL;
entity FSM2 is
  port (Clock : in std_logic;
        SlowRAN: in std_logic;
        Read, Write: out std_logic);
end entity;

architecture RTL of FSM2 is
  type StateType is (ST_Read, ST_Write, ST_Delay);
  signal CurrentState, NextState: StateType;
begin
  SEQ: process (Clock)
  begin
    if rising_edge(Clock) then
      if (Reset = '1') then
        CurrentState <= ST_Read;
      else
        CurrentState <= NextState;
      end if;
    end process SEQ;
  end process SEQ;

  COMB: process (CurrentState)
  begin
    case CurrentState is
    when ST_Read =>
      Read <= '1'; Write <= '0';
      NextState := ST_Write;
    when ST_Write =>
      Read <= '0'; Write <= '1';
      if (SlowRam = '1') then
        State := ST_Delay;
      else
        NextState := ST_Read
      end if;
    when ST_Delay =>
      Read <= '0'; Write <= '0';
      NextState := ST_Read;
    end case;
  end process COMB;
end architecture RTL;
SEQ: process (Clock)
begin
    if rising_edge(Clock) then
        if (Reset = '1') then
            CurrentState <= ST_Read;
        else
            CurrentState <= NextState;
        end if;
    end if;
end process SEQ;
COMB: process (CurrentState)
begin
  case CurrentState is
    when ST_Read =>
      Read <= '1';
      Write <= '0';
      NextState := ST_Write;
    when ST_Write =>
      Read <= '0';
      Write <= '1';
      if (SlowRam = '1') then
        State := ST_Delay;
      else
        NextState := ST_Read
      end if;
    when ST_Delay =>
      Read <= '0';
      Write <= '0';
      NextState := ST_Read;
  end case;
end case;
end process COMB;