Recap – Moore FSM Example

• Moore FSM that recognizes sequence “10”

Moore_state: state

U_Moore: PROCESS (clock, reset)
BEGIN
  IF(reset = '1') THEN
    Moore_state <= S0;
  ELSIF (clock = '1' AND clock'event) THEN
    CASE Moore_state IS
      WHEN S0 =>
        IF input = '1' THEN
          Moore_state <= S1;
        ELSE
          Moore_state <= S0;
        END IF;
      WHEN S1 =>
        IF input = '0' THEN
          Moore_state <= S2;
        ELSE
          Moore_state <= S1;
        END IF;
      WHEN S2 =>
        IF input = '0' THEN
          Moore_state <= S0;
        ELSE
          Moore_state <= S1;
        END IF;
    END CASE;
  END IF;
END PROCESS;

Output <= '1' WHEN Moore_state = S2 ELSE '0';

Recap – Mealy FSM Example

• Mealy FSM that recognizes sequence “10”

Mealy_state: state

U_Mealy: PROCESS (clock, reset)
BEGIN
  IF(reset = '1') THEN
    Mealy_state <= S0;
  ELSIF (clock = '1' AND clock'event) THEN
    CASE Mealy_state IS
      WHEN S0 =>
        IF input = '1' THEN
          Mealy_state <= S1;
        ELSE
          Mealy_state <= S0;
        END IF;
      WHEN S1 =>
        IF input = '0' THEN
          Mealy_state <= S2;
        ELSE
          Mealy_state <= S1;
        END IF;
    END CASE;
  END IF;
END PROCESS;

Moore FSM Example – VHDL

TYPE state IS (S0, S1, S2);
SIGNAL Moore_state: state;

U_Moore: PROCESS (clock, reset)
BEGIN
  IF(reset = '1') THEN
    Moore_state <= S0;
  ELSIF (clock = '1' AND clock'event) THEN
    CASE Moore_state IS
      WHEN S0 =>
        IF input = '1' THEN
          Moore_state <= S1;
        ELSE
          Moore_state <= S0;
        END IF;
      WHEN S1 =>
        IF input = '0' THEN
          Moore_state <= S2;
        ELSE
          Moore_state <= S1;
        END IF;
      WHEN S2 =>
        IF input = '0' THEN
          Moore_state <= S0;
        ELSE
          Moore_state <= S1;
        END IF;
    END CASE;
  END IF;
END PROCESS;

Moore FSM Example – VHDL (cont.)

When S1 =>
  IF input = '0' THEN
    Moore_state <= S2;
  ELSE
    Moore_state <= S1;
  END IF;
END CASE;
END PROCESS;

Output <= '1' WHEN Moore_state = S2 ELSE '0';

Mealy FSM Example – VHDL

TYPE state IS (S0, S1);
SIGNAL Mealy_state: state;

U_Mealy: PROCESS (clock, reset)
BEGIN
  IF(reset = '1') THEN
    Mealy_state <= S0;
  ELSIF (clock = '1' AND clock'event) THEN
    CASE Mealy_state IS
      WHEN S0 =>
        IF input = '1' THEN
          Mealy_state <= S1;
        ELSE
          Mealy_state <= S0;
        END IF;
      WHEN S1 =>
        IF input = '0' THEN
          Mealy_state <= S2;
        ELSE
          Mealy_state <= S1;
        END IF;
    END CASE;
  END IF;
END PROCESS;

Output <= '1' WHEN Mealy_state = S2 ELSE '0';
**Mealy FSM Example – VHDL (cont.)**

WHEN S1 =>
  IF input = '0' THEN
    Mealy_state <= S0;
  ELSE
    Mealy_state <= S1;
  END IF;
END CASE;
END IF;
END PROCESS;

Output <= '1' WHEN (Mealy_state = S1 AND input = '0') ELSE '0';

**Finite State Machine Design**

A more “fair” bus arbiter
- 5 resources contending for the bus
  - Inputs r1 -> r5, Outputs g1 -> g5
- Tuesday’s arbiter
  - Resource r(i) has precedence over r(j>i) when bus is idle
  - Once granted access, resources can hold on to the bus as long as they want to
- Group 1 – same precedence, but now resource r(i) can only have bus for i cycles at a time
- Group 2 – if multiple requests for bus, tie goes to least recently used resource
- Group 3 – each resource can also “interrupt” the bus if necessary and gain instant access

**Outline**

- Recap
- Memories
  - Modeling RAM
  - Modeling ROM
- Writing Synthesizable Code
- Additional VHDL Features
  - Functions
  - Procedures
  - Attributes
  - Variables
  - Constants

**Generic RAM**

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ram IS
  GENERIC (bits: INTEGER:=8;         -- # of bits per word
            words: INTEGER := 16); -- # of words in the memory
  PORT (wr_ena, clk: IN STD_LOGIC;
        addr: IN INTEGER RANGE 0 to words-1;
        data_in:  IN STD_LOGIC_VECTOR(bits -1 downto 0);
        data_out:  OUT STD_LOGIC_VECTOR(bits – 1 downto 0)
          );
END ram;

ARCHITECTURE behavioral OF ram IS
  TYPE vector_array IS ARRAY (0 TO words-1) OF
    STD_LOGIC_VECTOR(bits – 1 DOWNTO 0);
  SIGNAL memory: vector array;
BEGIN
  PROCESS(clk)
  BEGIN
    IF(wr_ena='1') THEN
      IF (clk'EVENT AND clk='1') THEN
        memory(addr) <= data_in;
      END_IF;
    END IF;
  END PROCESS;
  data_out <= memory(addr);
END ram;

**Generic RAM (cont.)**

ARCHITECTURE behavioral OF ram IS
  TYPE vector_array IS ARRAY (0 TO words-1) OF
    STD_LOGIC_VECTOR(bits – 1 DOWNTO 0);
  SIGNAL memory: vector array;
BEGIN
  PROCESS(clk)
  BEGIN
    IF(clk'EVENT AND clk='1') THEN
      IF (clk'EVENT AND clk='1') THEN
        memory(addr) <= data_in;
      END_IF;
    END IF;
  END PROCESS;
  data_out <= memory(addr);
END ram;

**Generic ROM**

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY rom IS
  GENERIC (bits: INTEGER:=8;         -- # of bits per word
            words: INTEGER := 8); -- # of words in the memory
  PORT (addr: IN INTEGER RANGE 0 to words-1;
        data:  OUT STD_LOGIC_VECTOR(bits – 1 downto 0)
          );
END rom;
### Constants

**Syntax:**

```
CONSTANT name : type := value;
```

**Examples:**

```
CONSTANT high : STD_LOGIC := '1';
CONSTANT datamemory : memory := ((X"00", X"02"));
```

### Constants – Features

- Constants can be declared in a PACKAGE, ENTITY, or ARCHITECTURE
- When declared in a PACKAGE, the constant is truly global, for the package can be used in several entities
- When declared in an ARCHITECTURE, the constant is local, i.e., it is visible only within this architecture
- When declared in an ENTITY, the constant can be used in all architectures associated with this entity

### Generic ROM (cont.)

```
ARCHITECTURE behavioral OF rom IS
    TYPE vector_array IS ARRAY (0 TO words-1) OF
    STD_LOGIC_VECTOR(bits – 1 DOWNTO 0);
    CONSTANT memory: vector_array :=
        (("0000_0000", "0000_0010", "0000_0100", "0000_1000", "0001_0000", "0010_0000", "0100_0000", "1000_0000");
BEGIN
    data <= memory(addr);
END rom;
```

### Distributed RAM

- CLB LUT configurable as Distributed RAM
  - A LUT equals 16x1 RAM
  - Implements Single and Dual-Ports
  - Cascade LUTs to increase RAM size
- Synchronous write
- Synchronous/Asynchronous read
  - Accompanying flip-flops used for synchronous read

### RAM 16x1

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

entity RAM_16X1_DISTRIBUTED is
    port(
        CLK : in STD_LOGIC;
        WE : in STD_LOGIC;
        ADDR : in STD_LOGIC_VECTOR(3 downto 0);
        DATA_IN : in STD_LOGIC;
        DATA_OUT : out STD_LOGIC
    );
end entity;
```

### RAM 16x1 (cont.)

```
architecture RAM_16X1_DISTRIBUTED_STRUCTURAL of RAM_16X1_DISTRIBUTED is
    -- part used by the synthesis tool, Synplify Pro, only; ignored during simulation
    attribute INIT : string;
    begin
        INIT of RAM16X1_S_1: label is "0000";
    end;

    component ram16x1s
        generic(
            INIT : BIT_VECTOR(15 downto 0) := X"0000"
        );
        port(
            O :  out std_ulogic;
            A0 : in std_ulogic;
            A1 : in std_ulogic;
            D : in std_ulogic;
            WCLK : in std_ulogic;
            WE : in std_ulogic;
        );
    end component;
```

### RAM 16x1 (cont.)

```
architecture RAM_16X1_DISTRIBUTED_STRUCTURAL of RAM_16X1_DISTRIBUTED is
    -- part used by the synthesis tool, Synplify Pro, only; ignored during simulation
    attribute INIT : string;
    begin
        INIT of RAM16X1_S_1: label is "0000";
    end;

    component ram16x1s
        generic(
            INIT : BIT_VECTOR(15 downto 0) := X"0000"
        );
        port(
            O :  out std_ulogic;
            A0 : in std_ulogic;
            A1 : in std_ulogic;
            D : in std_ulogic;
            WCLK : in std_ulogic;
            WE : in std_ulogic;
        );
    end component;
```
### RAM 16x1 (cont.)

```vhdl
begin
  RAM_16X1_S_1: ram16x1s generic map (INIT => 'X'0000')
  port map
    (O => DATA_OUT,
    A0 => ADDR(0),
    A1 => ADDR(1),
    A2 => ADDR(2),
    A3 => ADDR(3),
    D  => DATA_IN,
    WCLK => CLK,
    WE => WE
  );
end RAM_16X1_DISTRIBUTED_STRUCTURAL;
```

### Writing Synthesizable Code

- For combinational logic, use only concurrent statements
  - Concurrent signal assignment (\(\Leftarrow\))
  - Conditional concurrent signal assignment (when-else)
  - Selected concurrent signal assignment (with-select-when)
  - Generate scheme for equations (for-generate)

### Writing Synthesizable Code (cont.)

- For circuits composed of
  - Simple logic operations (logic gates)
  - Simple arithmetic operations (addition, subtraction, multiplication)
  - Shifts/rotations by a constant
- Use concurrent signal assignment (\(\Leftarrow\))

### Writing Synthesizable Code (cont.)

- For circuits composed of
  - Multiplexers
  - Decoders, encoders
  - Tri-state buffers
- Use
  - Conditional concurrent signal assignment (when-else)
  - Selected concurrent signal assignment (with-select-when)

### Left-Side v. Right-Side of Assignment

- \(\Leftarrow\) = when-else
- \(\Leftarrow\) = with-select

**Left side**
- Internal signals (defined in a given architecture)
- Ports of the mode
  - out
  - in
  - buffer
**Right side**
- Expressions including:
  - Internal signals (defined in a given architecture)
  - Ports of the mode
    - in
    - inout
    - buffer

### Arithmetic Operations

- Synthesizable arithmetic operations:
  - Addition, +
  - Subtraction, -
  - Comparisons, >, >=, <, <=
  - Multiplication, *
  - Division by a power of 2, \(\mathit{2}^n\) (equivalent to right shift)
  - Shifts by a constant, SHL, SHR
Arithmetic Operations (cont.)

- The result of synthesis of an arithmetic operation is a
  - Combinational circuit
  - Without pipelining

- The exact internal architecture used (and thus delay and area of the circuit) may depend on the timing constraints specified during synthesis (e.g., the requested maximum clock frequency)

Operations on Numbers

- For operations on unsigned numbers
  - USE ieee.std_logic_unsigned.all
  - Signals (inputs/outputs) of the type STD_LOGIC_VECTOR
  - Or, USE ieee.std_logic_arith.all
  - Signals (inputs/outputs) of the type UNSIGNED

- For operations on signed numbers
  - USE ieee.std_logic_signed.all
  - signals (inputs/outputs) of the type STD_LOGIC_VECTOR
  - Or, USE ieee.std_logic_arith.all
  - Signals (inputs/outputs) of the type SIGNED

  Signed / Unsigned types behave exactly like
  STD_LOGIC_VECTOR
  Also determine whether the number should be treated as a signed or unsigned number

Representing Integers

- Operations on signals (variables) of the integer types (INTEGER, NATURAL) and their subtypes are synthesizable in the range:
  - \([-2^{31}-1 ... 2^{31}-1]\) for INTEGERs and their subtypes
  - \([0 ... 2^{31}-1]\) for NATURALs and their subtypes

- Operations on the integer types are less flexible and more difficult to control than operations STD_LOGIC_VECTOR and are recommended to be avoided by beginners

Addition of Signed Numbers

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY adder16 IS
PORT ( Cin : IN STD_LOGIC;
       X, Y : IN SIGNED(15 DOWNTO 0);
       S : OUT SIGNED(15 DOWNTO 0);
       Cout, Overflow : OUT STD_LOGIC);
END adder16;

ARCHITECTURE Behavior OF adder16 IS
SIGNAL Sum : SIGNED(16 DOWNTO 0);
BEGIN
Sum <= ('0' & X) + Y + Cin;
S <= Sum(15 DOWNTO 0);
Cout <= Sum(16);
Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15);
END Behavior;

Addition of Signed Numbers (cont.)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY adder16 IS
PORT ( X, Y : IN INTEGER RANGE -32768 TO 32767;
       S : OUT INTEGER RANGE -32768 TO 32767);
END adder16;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
S <= X + Y;
END Behavior;

Addition of Signed Numbers (cont.)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY adder16 IS
PORT ( Cin : IN STD_LOGIC;
       X, Y : IN SIGNED(15 DOWNTO 0);
       S : OUT SIGNED(15 DOWNTO 0);
       Cout, Overflow : OUT STD_LOGIC);
END adder16;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
Sum <= ('0' & X) + Y + Cin;
S <= Sum(15 DOWNTO 0);
Cout <= Sum(16);
Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15);
END Behavior;

Addition of Signed Numbers (cont.)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY adder16 IS
PORT ( X, Y : IN INTEGER RANGE -32768 TO 32767;
       S : OUT INTEGER RANGE -32768 TO 32767);
END adder16;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
S <= X + Y;
END Behavior;
Combinational Logic using Processes

Rules that need to be followed:
1. All inputs to the combinational circuit should be included in the sensitivity list
2. No other signals should be included in the sensitivity list
3. None of the statements within the process should be sensitive to rising or falling edges
4. All possible cases need to be covered in the internal IF and CASE statements in order to avoid implied latches

Covering the IF Statement

Using ELSE

IF A = B THEN
  AeqB <= '1';
ELSE
  AeqB <= '0';

Using default values

AeqB <= '0';
IF A = B THEN
  AeqB <= '1';

Covering the CASE Statement

Using WHEN OTHERS

CASE y IS
  WHEN S1 => Z <= "10";
  WHEN S2 => Z <= "01";
  WHEN OTHERS => Z <= "00";
END CASE;

Using default values

Z <= "00";
CASE y IS
  WHEN S1 => Z <= "10";
  WHEN S2 => Z <= "01";
END CASE;

Covering the IF Statement

Using ELSE

IF A = B THEN
  AeqB <= '1';
ELSE
  AeqB <= '0';

Using default values

AeqB <= '0';
IF A = B THEN
  AeqB <= '1';

Variable – Example

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Numbits IS
  PORT ( X : IN STD_LOGIC_VECTOR(1 TO 3);
         Count : OUT INTEGER RANGE 0 TO 3);
END Numbits;

ARCHITECTURE Behavior OF Numbits IS
BEGIN
  PROCESS(X) -- count the number of bits in X equal to 1
  VARIABLE Tmp: INTEGER;
  BEGIN
    Tmp := 0;
    FOR i IN 1 TO 3 LOOP
      IF X(i) = '1' THEN
        Tmp := Tmp + 1;
      END IF;
    END LOOP;
    Count <= Tmp;
  END PROCESS;
END Behavior;
Variables – Features

• Can only be declared within processes and subprograms (functions & procedures)
• Initial value can be explicitly specified in the declaration
• When assigned take an assigned value immediately
• Variable assignments represent the desired behavior, not the structure of the circuit
• Should be avoided, or at least used with caution in a synthesizable code

Variables vs. Signals

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY test_delay IS
PORT(
  clk : IN STD_LOGIC;
  in1, in2 : IN STD_LOGIC;
  var1_out, var2_out : OUT STD_LOGIC;
  sig1_out : BUFFER STD_LOGIC;
  sig2_out : OUT STD_LOGIC
);
END test_delay;

ARCHITECTURE behavioral OF test_delay IS
BEGIN
  PROCESS(clk) IS
    VARIABLE var1, var2: STD_LOGIC;
    BEGIN
      if (rising_edge(clk)) THEN
        var1 := in1 AND in2;
        var2 := var1;
        sig1_out <= in1 AND in2;
        sig2_out <= sig1_out;
      END IF;
      var1_out <= var1;
      var2_out <= var2;
    END PROCESS;
  END behavioral;
END;

Variables vs. Signals (cont.)

ARCHITECTURE behavioral OF test_delay IS
BEGIN
  PROCESS(clk) IS
    VARIABLE var1, var2: STD_LOGIC;
    BEGIN
      if (rising_edge(clk)) THEN
        var1 := in1 AND in2;
        var2 := var1;
        sig1_out <= in1 AND in2;
        sig2_out <= sig1_out;
      END IF;
      var1_out <= var1;
      var2_out <= var2;
    END PROCESS;
  END behavioral;
END;

Simulation Result

Assert Statements

• Assert is a non-synthesizable statement whose purpose is to write out messages on the screen when problems are found during simulation
• Depending on the severity of the problem, the simulator is instructed to continue simulation or halt
• Syntax:
  • ASSERT condition [REPORT "message"] [SEVERITY severity_level];
  • The message is written when the condition is FALSE
  • Severity_level can be: Note, Warning, Error (default), or Failure

Array Attributes

A'left(N) left bound of index range of dimension N of A
A'right(N) right bound of index range of dimension N of A
A'low(N) lower bound of index range of dimension N of A
A'high(N) upper bound of index range of dimension N of A
A'range(N) index range of dimension N of A
A'reverse_range(N) index range of dimension N of A
A'length(N) length of index range of dimension N of A
A'ascending(N) true if index range of dimension N of A is an ascending range, false otherwise
Subprograms

- Include **functions** and **procedures**
- Commonly used pieces of code
- Can be placed in a library, and then reused and shared among various projects
- Use only sequential statements, the same as processes
- Example uses:
  - Abstract operations that are repeatedly performed
  - Type conversions

Functions – Basic Features

- Always return a single value as a result
- Are called using formal and actual parameters the same way as components
- Never modify parameters passed to them
- Parameters can only be constants (including generics) and signals (including ports);
- Variables are not allowed; the default is a CONSTANT
- When passing parameters, no range specification should be included (for example no RANGE for INTEGERS, or TO/DOWNTO for STD_LOGIC_VECTOR)
- Are always used in some expression, and not called on their own

Function Syntax and Example

```markdown
FUNCTION function_name (<parameter_list>) RETURN data_type IS
[declarations]
BEGIN
(sequential statements)
END function_name;
```

**Example:**

```markdown
FUNCTION f1 (a, b: INTEGER; SIGNAL c: STD_LOGIC_VECTOR) RETURN BOOLEAN IS
BEGIN
(sequential statements)
END f1;
```

Procedures – Basic Features

- Do not return a value
- Are called using formal and actual parameters the same way as components
- May modify parameters passed to them
- Each parameter must have a mode: IN, OUT, INOUT
- Parameters can be constants (including generics), signals (including ports), and variables
- The default for inputs (mode in) is a constant, the default for outputs (modes out and inout) is a variable
- When passing parameters, range specification should be included (for example RANGE for INTEGERS, and TO/DOWNTO for STD_LOGIC_VECTOR)
- Procedure calls are statements on their own

Procedure Syntax and Example

```markdown
PROCEDURE procedure_name (<parameter_list>) IS
[declarations]
BEGIN
(sequential statements)
END procedure_name;
```

**Example:**

```markdown
PROCEDURE p1 (a, b: in INTEGER; SIGNAL c: out STD_LOGIC_VECTOR) [declarations]
BEGIN
(sequential statements)
END p1;
```