Recap – 4:1 Multiplexer

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux4to1 IS
  PORT ( w0, w1, w2, w3 : IN STD_LOGIC;
        s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        f : OUT STD_LOGIC );
END mux4to1;

ARCHITECTURE dataflow OF mux4to1 IS
  BEGIN
    WITH s SELECT
      f <= w0 WHEN "00",
           w1 WHEN "01",
           w2 WHEN "10",
           w3 WHEN OTHERS;
  END dataflow;
```

Recap – N-bit Register with Reset

```vhdl
ENTITY regn IS
  GENERIC ( N : INTEGER := 16 );
  PORT ( D                     : IN STD_LOGIC_VECTOR( N-1 DOWNTO 0 ) ;
          Resetn, Clock : IN STD_LOGIC ;
          Q : OUT STD_LOGIC_VECTOR( N-1 DOWNTO 0 ) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
  BEGIN
    PROCESS ( Resetn, Clock )
      BEGIN
        IF Resetn = '0' THEN
          Q <= (OTHERS => '0');
        ELSIF Clock'EVENT AND Clock = '1' THEN
          Q <= D;
        END IF;
      END PROCESS ;
  END Behavior ;
```

Recap – 4-bit Up-Counter with Reset

```vhdl
ARCHITECTURE Behavior OF upcount IS
  SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0 );
  BEGIN
    PROCESS ( Clock, Resetn )
      BEGIN
        IF Resetn = '0' THEN
          Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
          IF Enable = '1' THEN
            Count <= Count + 1 ;
          END IF ;
        END IF ;
      END PROCESS ;
    Q <= Count ;
  END Behavior ;
```

Design Exercise

- Design a simple 32-bit CPU
  - Requirements
    - Three instruction types: load/store, register ALU, branch-if-equal
    - 8 32-bit registers
    - ALU operations: ADD, SUB, OR, XOR, AND, CMP
    - Memory operations: load word, store word
  - Components
    - Instruction memory / decode
    - Register file
    - ALU
    - Data memory
    - Other control

Outline

- Recap
  - Finite State Machines
    - Moore Machines
    - Mealy Machines
  - FSMs in VHDL
  - State Encoding
  - Example Systems
    - Serial Adder
    - Arbiter Circuit
Structure of a Typical Digital System

Execution Unit (Datapath)
- Provides all necessary resources and interconnects among them to perform specified task
- Examples of resources
  - Adders, multipliers, registers, memories, etc.

Control Unit (Control)
- Controls data movements in operational circuit by switching multiplexers and enabling or disabling resources
- Follows some ‘program’ or schedule
- Often implemented as Finite State Machine or collection of Finite State Machines

Finite State Machines (FSMs)
- Any circuit with memory is a Finite State Machine
  - Even computers can be viewed as huge FSMs
- Design of FSMs involves
  - Defining states
  - Defining transitions between states
  - Optimization / minimization
- Above approach is practical for small FSMs only

Moore FSM
- Output is a function of present state only

Mealy FSM
- Output is a function of a present state and inputs
**Moore vs. Mealy FSM**

- Moore and Mealy FSMs can be functionally equivalent
  - Equivalent Mealy FSM can be derived from Moore FSM and vice versa
- Mealy FSM has richer description and usually requires smaller number of states
  - Smaller circuit area

**Moore vs. Mealy FSM (cont.)**

- Mealy FSM computes outputs as soon as inputs change
  - Mealy FSM responds one clock cycle sooner than equivalent Moore FSM
- Moore FSM has no combinational path between inputs and outputs
  - Moore FSM is more likely to have a shorter critical path

**Moore FSM Example**

- Moore FSM that recognizes sequence “10”

  Meaning of states:
  - S0: No elements of the sequence observed
  - S1: “1” observed
  - S2: “10” observed

**Mealy FSM Example**

- Mealy FSM that recognizes sequence “10”

  Meaning of states:
  - S0: No elements of the sequence observed
  - S1: “1” observed
Mealy FSM Example (cont.)

FSMs in VHDL

- Finite State Machines can be easily described with processes
- Synthesis tools understand FSM description if certain rules are followed
  - State transitions should be described in a process sensitive to clock and asynchronous reset signals only
  - Outputs described as concurrent statements outside the 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;
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 <= S0;
        ELSE
          Mealy_state <= S1;
        END IF;
    END CASE;
  END IF;
END PROCESS;
```

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';
```
State Encoding Problem

- State encoding can have a big influence on optimality of the FSM implementation
  - No methods other than checking all possible encodings are known to produce optimal circuit
  - Feasible for small circuits only
- Using enumerated types for states in VHDL leaves encoding problem for synthesis tool

Types of State Encodings

- Binary (Sequential) – States encoded as consecutive binary numbers
  - Small number of used flip-flops
  - Potentially complex transition functions leading to slow implementations
- One-Hot – only one bit is active
  - Number of used flip-flops as big as number of states
  - Simple and fast transition functions
  - Preferable coding technique in FPGAs

Types of State Encodings (cont.)

<table>
<thead>
<tr>
<th>State</th>
<th>Binary Code</th>
<th>One-Hot Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>000</td>
<td>100000000</td>
</tr>
<tr>
<td>S1</td>
<td>001</td>
<td>010000000</td>
</tr>
<tr>
<td>S2</td>
<td>010</td>
<td>001000000</td>
</tr>
<tr>
<td>S3</td>
<td>011</td>
<td>000100000</td>
</tr>
<tr>
<td>S4</td>
<td>100</td>
<td>000010000</td>
</tr>
<tr>
<td>S5</td>
<td>101</td>
<td>000001000</td>
</tr>
<tr>
<td>S6</td>
<td>110</td>
<td>000000100</td>
</tr>
<tr>
<td>S7</td>
<td>111</td>
<td>000000010</td>
</tr>
</tbody>
</table>

Manual State Assignment

(ENTITY declaration not shown)

ARCHITECTURE Behavior OF simple IS
  TYPE State_type IS (A, B, C);
  ATTRIBUTE ENUM_ENCODING : STRING := "00 01 11";
  SIGNAL y_present, y_next : State_type;
BEGIN
  ... cont

ARCHITECTURE Behavior OF simple IS
  SUBTYPE ABC_STATE is STD_LOGIC_VECTOR(1 DOWNTO 0);
  CONSTANT A : ABC_STATE := "00";
  CONSTANT B : ABC_STATE := "01";
  CONSTANT C : ABC_STATE := "11";
  SIGNAL y_present, y_next : ABC_STATE;
BEGIN
  BEGIN
    PROCESS ( w, y_present )
      CASE y_present IS
        WHEN A =>
          IF w = '0' THEN y_next <= A;
          ELSE y_next <= B;
          END IF;
      WHEN B =>
          IF w = '0' THEN y_next <= B;
          ELSE y_next <= C;
          END IF;
      WHEN C =>
          IF w = '0' THEN y_next <= C;
          ELSE y_next <= A;
          END IF;
      WHEN OTHERS =>
          y_next <= "00";
      END CASE;
  ... cont

Serial Adder – Block Diagram

A: Shift register
B: Adder FSM
C: Clock

Sum = A + B
Serial Adder FSM

\[ (ab/s) \]
\[
\begin{array}{c|cc}
ab & G & H \\
00 & 1 & 1 \\
01 & 1 & 0 \\
10 & 0 & 0 \\
11 & 0 & 1 \\
\end{array}
\]

G: carry-in = 0
H: carry-in = 1

Serial Adder FSM – State Table

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 10 11</td>
<td>00 1 1 0</td>
</tr>
<tr>
<td>G</td>
<td>G G G H</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>H</td>
<td>G H H H</td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>

Serial Adder – Entity Declaration

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY serial IS
  GENERIC ( length : INTEGER := 8 ) ;
  PORT ( Clock : IN STD_LOGIC ;
         Reset : IN STD_LOGIC ;
         A, B : IN STD_LOGIC_VECTOR(length-1 DOWNTO 0) ;
         Sum : BUFFER STD_LOGIC_VECTOR(length-1 DOWNTO 0));
END serial ;

ARCHITECTURE Behavior OF serial IS
  SIGNAL QA, QB, Null_in : STD_LOGIC_VECTOR(length-1 DOWNTO 0) ;
  SIGNAL s, Low, High, Run : STD_LOGIC ;
  SIGNAL Count : INTEGER RANGE 0 TO length ;
  TYPE State_type IS (G, H) ;
  SIGNAL y : State_type ;
  BEGIN
    Low <= '0' ; High <= '1' ;
    ShiftA: shiftrne GENERIC MAP (N => length)
     PORT MAP ( A, Reset, High, Low, Clock, QA ) ;
    ShiftB: shiftrne GENERIC MAP (N => length)
     PORT MAP ( B, Reset, High, Low, Clock, QB ) ;
    AdderFSM: PROCESS ( Reset, Clock )
     BEGIN
       y <= G ;
       ELIF Clock'EVENT AND Clock = '1' THEN
       CASE y IS
         WHEN G =>
           WHEN QA(0) = '1' AND QB(0) = '1' THEN y <= H ;
           ELSE y <= G ;
           END IF ;
         WHEN H =>
           IF QA(0) = '0' AND QB(0) = '0' THEN y <= G ;
           ELSE y <= H ;
           END IF ;
         ELSE y <= G ;
         END CASE ;
       END IF ;
     END PROCESS AdderFSM ;
Serial Adder – Architecture (5)

WITH y SELECT
s <= QA(0) XOR QB(0) WHEN Q,
NOT ( QA(0) XOR QB(0) ) WHEN H;
Null_in <= (OTHERS => '0');
ShiftSum: shiftrne GENERIC MAP ( N => length )
PORT MAP ( Null_in, Reset, Run, s, Clock, Sum );

Serial Adder – Architecture (5)

Stop: PROCESS
BEGIN
WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
IF Reset = '1' THEN
Count <= length ;
ELSIF Run = '1' THEN
Count <= Count -1 ;
END IF ;
END PROCESS ;
Run <= '0' WHEN Count = 0 ELSE '1' ;  -- stops counter and ShiftSum
END Behavior ;

Serial Adder - Mealy FSM Circuit

Arbiter Circuit

Arbiter Moore State Diagram

Grant Signals – VHDL Code

PROCESS( y )
BEGIN
  g(1) <= '0';
g(2) <= '0';
g(3) <= '0';
IF y = gnt1 THEN g(1) <= '1';
ELSIF y = gnt2 THEN g(2) <= '1';
ELSIF y = gnt3 THEN g(3) <= '1';
END IF ;
END PROCESS ;
END Behavior ;
### Arbiter Simulation Results

<table>
<thead>
<tr>
<th>Name</th>
<th>250 ns</th>
<th>500 ns</th>
<th>750 ns</th>
<th>1.0 µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rasatin</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>Clock</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>r1</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>r2</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>8</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>qf</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>qo</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>qf</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>qo</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
<tr>
<td>y</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
<td>![Waveform]</td>
</tr>
</tbody>
</table>