Design of Controllers

Finite State Machines and Algorithmic State Machine (ASM) Charts
Required reading

- P. Chu, *RTL Hardware Design using VHDL*

  **Chapter 10, Finite State Machine: Principle & Practice**

  **Chapter 11, Register Transfer Methodology: Principle**
Recommended reading

• P. Chu, *RTL Hardware Design using VHDL*

  Chapter 12, *Register Transfer Methodology: Practice*
Slides based partially on

- S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*
  - Chapter 8, *Synchronous Sequential Circuits*
    - Sections 8.1-8.5
  - Chapter 8.10, *Algorithmic State Machine (ASM) Charts*
  - Chapter 10.2 Design Examples
Datapath
VS.
Controller
Structure of a Typical Digital System

Datapath (Execution Unit)

Data Inputs

Control & Status Inputs

Control Signals

Status Signals

Data Outputs

Control & Status Outputs

Controller (Control Unit)
Datapath (Execution Unit)

- Manipulates and processes data
- Performs arithmetic and logic operations, shifting/rotating, and other data-processing tasks
- Is composed of registers, multiplexers, adders, decoders, comparators, ALUs, gates, etc.
- Provides all necessary resources and interconnects among them to perform specified task
- Interprets control signals from the Controller and generates status signals for the Controller
Controller (Control Unit)

- Controls data movements in the Datapath by switching multiplexers and enabling or disabling resources
  - Example: enable signals for registers
  - Example: select signals for muxes
- Provides signals to activate various processing tasks in the Datapath
- Determines the sequence of operations performed by the Datapath
- Follows Some ‘Program’ or Schedule
Programmable vs. Non-Programmable Controller

• Controller can be programmable or non-programmable

• Programmable
  • Has a program counter which points to the next instruction
  • Instructions are held in a RAM or ROM
  • Microprocessor is an example of a programmable controller

• Non-Programmable
  • Once designed, implements the same functionality
  • Another term is a “hardwired state machine,” or “hardwired FSM,” or “hardwired instructions”
  • In this course we will be focusing on non-programmable controllers.
Finite State Machines

• Digital Systems and especially their Controllers can be described as Finite State Machines (FSMs)

• Finite State Machines can be represented using
  • State Diagrams and State Tables - suitable for simple digital systems with a relatively few inputs and outputs
  • Algorithmic State Machine (ASM) Charts - suitable for complex digital systems with a large number of inputs and outputs

• All these descriptions can be easily translated to the corresponding synthesizable VHDL code
Hardware Design with RTL VHDL

Pseudocode

Datapath

Block diagram

VHDL code

Controller

Block diagram

State diagram or ASM chart

VHDL code

Interface
Steps of the Design Process

1. Text description
2. Interface
3. Pseudocode
4. Block diagram of the Datapath
5. Interface with the division into Datapath and Controller
6. ASM chart of the Controller
7. RTL VHDL code of the Datapath, Controller, and Top-Level Unit
8. Testbench of the Datapath, Controller, and Top-Level Unit
9. Functional simulation and debugging
10. Synthesis and post-synthesis simulation
11. Implementation and timing simulation
12. Experimental testing
Steps of the Design Process
Introduced in Class Today

1. Text description
2. Interface
3. Pseudocode
4. Block diagram of the Datapath
5. Interface with the division into Datapath and Controller
6. **ASM chart of the Controller**
7. **RTL VHDL code of** the Datapath, **Controller**, and Top-Level Unit
8. **Testbench of** the Datapath, **Controller**, and Top-Level Unit
9. Functional simulation and debugging
10. Synthesis and post-synthesis simulation
11. Implementation and timing simulation
12. Experimental testing
Finite State Machines Refresher
Finite State Machines (FSMs)

• An FSM is used to model a system that transits among a finite number of internal states. The transitions depend on the current state and external input.

• The main application of an FSM is to act as the controller of a medium to large digital system

• Design of FSMs involves
  • Defining states
  • Defining next state and output functions
  • Optimization / minimization

• Manual optimization/minimization is practical for small FSMs only
Moore FSM

- Output Is a Function of a Present State Only
Mealy FSM

- Output is a function of a present state and inputs

```
Inputs → Next State function → Present State register
          ↓ clock, reset
          ← Present State
          → Next State
              ↓ Outputs
Output function
```

Present State

Inputs → Next State function
State Diagrams
Moore Machine

state 1 / output 1

transition condition 1

state 2 / output 2

transition condition 2
Mealy Machine

state 1

transition condition 1 / output 1

state 2

transition condition 2 / output 2
Moore FSM - Example 1

- Moore FSM that Recognizes Sequence “10”

Meaning of states:

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

```

```

unset

reset
Mealy FSM - Example 1

• Mealy FSM that Recognizes Sequence “10”

Meaning of states:

S0: No elements of the sequence observed
S1: “1” observed

Diagram:

- States: S0, S1
- Edges:
  - S0 to S0 with input 0/0
  - S0 to S1 with input 1/0
  - S1 to S1 with input 1/0
  - S1 to S0 with input 0/1
  - S0 to S0 with label “reset”
Moore & Mealy FSMs – Example 1

Moore

input

output

state

S0 S0 S1 S2 S0 S0

S0 S0 S1 S0 S0 S0

Mealy

input

output

state

S0 S0 S1 S0 S0 S0

S0 S0 S1 S0 S0 S0
Moore vs. Mealy FSM (1)

• 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 (2)

- 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 less likely to affect the critical path of the entire circuit
Which Way to Go?

Mealy FSM

- Fewer states
- Lower Area
- Responds one clock cycle earlier

Moore FSM

Safer.
Less likely to affect the critical path.
Problem 1

Assuming state diagram given on the next slide, supplement timing waveforms given in the answer sheet with the correct values of signals State and c, in the interval from 0 to 575 ns.
Finite State Machines in VHDL
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
  • Output function described using rules for combinational logic, i.e. as concurrent statements or a process with all inputs in the sensitivity list
Moore FSM

process(clock, reset)

Inputs

Next State function

Next State

Present State Register

clock
reset

concurrent statements

Output function

Present State

Outputs
Mealy FSM

process(clock, reset)

concurrent statements
Moore FSM - Example 1

• Moore FSM that Recognizes Sequence “10”

![Moore FSM Diagram]

- States: S0 / 0, S1 / 0, S2 / 1
- Transitions:
  - S0 / 0 to S1 / 0 on input 1
  - S1 / 0 to S2 / 1 on input 0
  - S2 / 1 to S0 / 0 on input 1
  - Reset state connection

reset
Moore FSM in VHDL (1)

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

U_Moore: PROCESS (clock, reset)
BEGIN
    IF(reset = '1') THEN
        Moore_state <= S0;
    ELSIF rising_edge(clock) THEN
        CASE Moore_state IS
            WHEN S0 =>
                IF input = '1' THEN
                    Moore_state <= S1;
                ELSE
                    Moore_state <= S0;
                END IF;
            END CASE;
    END IF;
END;
Moore FSM in VHDL (2)

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 1

- Mealy FSM that Recognizes Sequence “10”
TYPE state IS (S0, S1);
SIGNAL Mealy_state: state;

U_Mealy: PROCESS(clock, reset)
BEGIN
  IF(reset = '1') THEN
    Mealy_state <= S0;
  ELSIF rising_edge(clock) THEN
    CASE Mealy_state IS
    WHEN S0 =>
      IF input = '1' THEN
        Mealy_state <= S1;
      ELSE
        Mealy_state <= S0;
      END IF;
    END CASE;
  END IF;
END PROCESS;
Mealy FSM in VHDL (2)

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';
Algorithmic State Machine (ASM) Charts
Algorithmic State Machine

Algorithmic State Machine – representation of a Finite State Machine suitable for FSMs with a larger number of inputs and outputs compared to FSMs expressed using state diagrams and state tables.
Elements used in ASM charts (1)

(a) State box

(b) Decision box

(c) Conditional output box

- State name
- Output signals or actions (Moore type)
- Condition expression
- 0 (False)
- 1 (True)
- Conditional outputs or actions (Mealy type)
State Box

- **State box** – represents a state.
- Equivalent to a node in a state diagram or a row in a state table.
- Contains register transfer actions or output signals
- **Moore-type outputs are listed inside of the box.**
- It is customary to write only the name of the signal that has to be asserted in the given state, e.g., z instead of z<=1.
- Also, it might be useful to write an action to be taken, e.g., count <= count + 1, and only later translate it to asserting a control signal that causes a given action to take place (e.g., enable signal of a counter).
Decision Box

- **Decision box** – indicates that a given condition is to be tested and the exit path is to be chosen accordingly. The condition expression may include one or more inputs to the FSM.
Conditional Output Box

- Conditional output box
- Denotes output signals that are of the Mealy type.
- The condition that determines whether such outputs are generated is specified in the decision box.

Conditional outputs or actions (Mealy type)
ASMs representing simple FSMs

- Algorithmic state machines can model both Mealy and Moore Finite State Machines
- They can also model machines that are of the mixed type
Moore FSM – Example 2: State diagram
Moore FSM – Example 2: State table

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output $z$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$w = 0$</td>
<td>$w = 1$</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>
ASM Chart for Moore FSM – Example 2
Example 2: VHDL code (1)

USE ieee.std_logic_1164.all;

ENTITY simple IS
  PORT ( clock   : IN STD_LOGIC;
         resetn  : IN STD_LOGIC;
         w       : IN STD_LOGIC;
         z       : OUT STD_LOGIC );
END simple;

ARCHITECTURE Behavior OF simple IS
  TYPE State_type IS (A, B, C);
  SIGNAL y : State_type;
BEGIN
  PROCESS ( resetn, clock )
  BEGIN
    IF resetn = '0' THEN
      y <= A;
    ELSIF rising_edge(clock) THEN
      Example 2: VHDL code (1)
例 2：VHDL 代码 (2)

```vhdl
CASE y IS
  WHEN A =>
    IF w = '1' THEN
      y <= B;
    ELSE
      y <= A;
    END IF;
  WHEN B =>
    IF w = '1' THEN
      y <= C;
    ELSE
      y <= A;
    END IF;
  WHEN C =>
    IF w = '0' THEN
      y <= A;
    ELSE
      y <= C;
    END IF;
END CASE;
```
Example 2: VHDL code (3)

END IF;
END PROCESS;

z <= '1' WHEN y = C ELSE '0';

END Behavior;
Mealy FSM – Example 3: State diagram

\[w_0 = z_0 = \frac{1}{w_1} = z_1 = \frac{1}{B}\]

\[w = 0/z = 0\]

\[w = 1/z = 0\]

\[w = 0/z = 0\]

\[w = 1/z = 1\]
ASM Chart for Mealy FSM – Example 3
Example 3: VHDL code (1)

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Mealy IS
  PORT ( clock : IN STD_LOGIC;
         resetn : IN STD_LOGIC;
         w : IN STD_LOGIC;
         z : OUT STD_LOGIC );
END Mealy;

ARCHITECTURE Behavior OF Mealy IS
  TYPE State_type IS (A, B);
  SIGNAL y : State_type;
BEGIN
  PROCESS ( resetn, clock )
  BEGIN
    IF resetn = '0' THEN
      y <= A;
    ELSIF rising_edge(clock) THEN
      Example 3: VHDL code (1)
Example 3: VHDL code (2)

CASE y IS
  WHEN A =>
    IF w = '1' THEN
      y <= B ;
    ELSE
      y <= A ;
    END IF ;
  WHEN B =>
    IF w = '0' THEN
      y <= A ;
    ELSE
      y <= B ;
    END IF ;
END CASE ;
Example 3: VHDL code (3)

END IF ;
END PROCESS ;

z <= '1' WHEN (y = B) AND (w='1') ELSE '0' ;

END Behavior ;
Control Unit Example: Arbiter (1)

Arbiter

reset

g1

g2

g3

clock

r1

r2

r3
Control Unit Example: Arbiter (2)
Control Unit Example: Arbiter (3)
ASM Chart for Control Unit - Example 4
Example 4: VHDL code (1)

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY arbiter IS
  PORT ( Clock, Resetn : IN STD_LOGIC ;
         r : IN STD_LOGIC_VECTOR(1 TO 3) ;
         g : OUT STD_LOGIC_VECTOR(1 TO 3) ) ;
END arbiter ;

ARCHITECTURE Behavior OF arbiter IS
  TYPE State_type IS (Idle, gnt1, gnt2, gnt3) ;
  SIGNAL y : State_type ;
BEGIN
PROCESS ( Resetn, Clock )
BEGIN
    IF Resetn = '0' THEN y <= Idle ;
    ELSIF rising_edge(Clock) THEN
        CASE y IS
            WHEN Idle =>
                IF r(1) = '1' THEN y <= gnt1 ;
                ELSIF r(2) = '1' THEN y <= gnt2 ;
                ELSIF r(3) = '1' THEN y <= gnt3 ;
                ELSE y <= Idle ;
                END IF ;
            WHEN gnt1 =>
                IF r(1) = '0' THEN y <= Idle ;
                ELSE y <= gnt1 ;
                END IF ;
            WHEN gnt2 =>
                IF r(2) = '0' THEN y <= Idle ;
                ELSE y <= gnt2 ;
                END IF ;
        END CASE ;
    END IF ;
END PROCESS ;
Example 4: VHDL code (3)

WHEN gnt3 =>
    IF r(3) = '0' THEN y <= Idle ;
    ELSE y <= gnt3 ;
    END IF ;

    END CASE ;
    END IF ;
END PROCESS ;

    g(1) <= '1' WHEN y = gnt1 ELSE '0' ;
    g(2) <= '1' WHEN y = gnt2 ELSE '0' ;
    g(3) <= '1' WHEN y = gnt3 ELSE '0' ;
END Behavior ;
Problem 2

Assuming ASM chart given on the next slide, supplement timing waveforms given in the answer sheet with the correct values of signals State, $g_1$, $g_2$, $g_3$, in the interval from 0 to 575 ns.
ASM Chart
ASM Summary by Prof. Chu

• ASM (algorithmic state machine) chart
  – Flowchart-like diagram
  – Provides the same info as a state diagram
  – More descriptive, better for complex description
  – ASM block
    • One state box
    • One or more optional decision boxes:
      with T (1) or F (0) exit path
    • One or more conditional output boxes:
      for Mealy output
Figure 10.4  ASM block.
ASM Chart Rules

• Difference between a regular flowchart and an ASM chart:
  – Transition governed by clock
  – Transition occurs between ASM blocks

• Basic rules:
  – For a given input combination, there is one unique exit path from the current ASM block
  – Any closed loop in an ASM chart must include a state box

Based on RTL Hardware Design by P. Chu
Incorrect ASM Charts

Based on RTL Hardware Design by P. Chu
Generalized FSM

Based on RTL Hardware Design by P. Chu
Class Exercise 1

STATISTICS
Pseudocode

no_1 = no_2 = no_3 = sum = 0
for i=0 to k-1 do
    sum = sum + din
    if din > no_1 then
        no_3 = no_2
        no_2 = no_1
        no_1 = din
    elseif (din > no_2) then
        no_3 = no_2
        no_2 = din
    elseif (din > no_3) then
        no_3 = din
    end if
end for
avr = sum / k
Circuit Interface

 clk  -->  done
 reset  -->
 din  -->  n  -->  dout
 go  -->  2  -->  dout_mode

Statistics
## Interface Table

<table>
<thead>
<tr>
<th>Port</th>
<th>Width</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>System clock.</td>
</tr>
<tr>
<td>reset</td>
<td>1</td>
<td>System reset.</td>
</tr>
<tr>
<td>din</td>
<td>n</td>
<td>Input Data.</td>
</tr>
<tr>
<td>go</td>
<td>1</td>
<td>Control signal indicating that the first input is ready.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Active for one clock cycle.</td>
</tr>
<tr>
<td>done</td>
<td>1</td>
<td>Signal set to high after the output is ready.</td>
</tr>
<tr>
<td>dout</td>
<td>n</td>
<td>Output dependent on the dout_mode input.</td>
</tr>
<tr>
<td>dout_mode</td>
<td>2</td>
<td>Control signal determining value available at the output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00: avr, 01: no_1, 10: no_2, 11: no_3.</td>
</tr>
</tbody>
</table>
STATISTICS:
Block Diagram & Interface
Block diagram of the Datapath
Interface with the division into the Datapath and the Controller

Datapath

Controller

- din
- dout_mode
- clk
- reset
- go
- dout
- n
- 2
- en1
- en2
- en3
- esum
- enc
- s2
- s3
- done
Pseudocode

Split M into two halves A and B, \( w \) bits each

\[
\begin{align*}
A &= A + S[0] \\
B &= B + S[1] \\
\text{for } j = 1 \text{ to } r \text{ do } \\
\{ \\
A' &= ((A \oplus B) \ll B) + S[2j] \\
B' &= ((B \oplus A') \ll A') + S[2j+1] \\
A &= A' \\
B &= B' \\
\} \\
C &= A \parallel B
\end{align*}
\]
Notation

A, B, A’, B’ = w-bit variables
S[2j], S[2j+1] = a pair of round keys, each round key is a w-bit variable
⊕ = an XOR of two w-bit words
+ = unsigned addition mod 2^w
A <<< B = rotation of the variable A by a number of positions given by the current value of the variable B
A || B = concatenation of A and B

The algorithms has two parameters:
• r = number of rounds (e.g., 3)
• w = word size (always a power of 2, e.g., w = 2^4 = 16)
Circuit Interface

RC5

clk
reset
M
write_M
Si
write_Si
i

2w
2w
C
Done
w
m
# Circuit Interface

<table>
<thead>
<tr>
<th>Port</th>
<th>Width</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>System clock.</td>
</tr>
<tr>
<td>resct</td>
<td>1</td>
<td>System reset – clears internal registers.</td>
</tr>
<tr>
<td>M</td>
<td>$2w$</td>
<td>Message block.</td>
</tr>
<tr>
<td>write_M</td>
<td>1</td>
<td>Synchronous write control signal for the message block $M$. After the block $M$ is written to the RC5 unit, the encryption of $M$ starts automatically.</td>
</tr>
<tr>
<td>Si</td>
<td>$w$</td>
<td>Round key $S[i]$ loaded to one of the two internal memories. The first memory stores values of $S[i=2j]$, i.e., only round keys with even indices. The second memory stores values of $S[i=2j+1]$, i.e. only round keys with odd indices.</td>
</tr>
<tr>
<td>write_Si</td>
<td>1</td>
<td>Synchronous write control signal for the round key $S[i]$.</td>
</tr>
<tr>
<td>i</td>
<td>$m$</td>
<td>Index of the round key $S[i]$ loaded using input Si.</td>
</tr>
<tr>
<td>C</td>
<td>$2w$</td>
<td>Ciphertext block = Encrypted block $M$.</td>
</tr>
<tr>
<td>DONE</td>
<td>1</td>
<td>Asserted when ciphertext is ready, and available at the output.</td>
</tr>
</tbody>
</table>

Note:

$m$ is a size of index $i$. It is a minimum integer, such that $2^m \geq 2r+2$. 
Protocol (1)

An external circuit first loads all round keys $S[0], S[1], S[2], \ldots, S[2r], [2r+1]$ to the two internal memories of the RC5 unit.

The first memory stores values of $S[i=2j]$, i.e., only round keys with even indices. The second memory stores values of $S[i=2j+1]$, i.e. only round keys with odd indices.

Loading round keys is performed using inputs: $Si, i, write_Si, clk$.

Then, the external circuits, loads a message block $M$ to the RC5 unit, using inputs: $M, write_M, clk$.

After the message block $M$ is loaded to the RC5 unit, the encryption starts automatically.
Protocol (2)

When the encryption is completed, signal Done becomes active, and the output C changes to the new value of the ciphertext.

The output C keeps the last value of the ciphertext at the output, until the next encryption is completed. Before the first encryption is completed, this output should be equal to zero.
Assumptions

- one round of the main for loop of the pseudocode executes in one clock cycle
- you can access only one position of each internal memory of round keys per clock cycle

As a result, the entire encryption of a single message block M should last r+1 clock cycles.
Alternative Coding Styles
by Dr. Chu
(to be used with caution)
Traditional Coding Style

process(clock, reset)

Present State

Next State

inputs

Next State

Present State Register

Mealy Output function

Moore Output function

Mealy Outputs

Moore Outputs

concurrent statements
Alternative Coding Style 1

Process(Present State, Inputs)

Inputs

Next State function

Next State

Present State Register

Process(clock, reset)

Present State

Process(Present State, Inputs)

Mealy Output function

Mealy Outputs

Moore Output function

Moore Outputs
library ieee;
use ieee.std_logic_1164.all;

entity mem_ctrl is
port (  
    clk, reset: in std_logic;
    mem, rw, burst: in std_logic;
    oe, we, we_me: out std_logic);
end mem_ctrl ;

architecture mult_seg_arch of mem_ctrl is

  type mc_state_type is 
    (idle, read1, read2, read3, read4, write);

  signal state_reg, state_next: mc_state_type;

begin
  -- state register
  process(clk, reset)
  begin
    if (reset='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end process;

  process(clk, reset)
  begin
    if (reset='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end process;

  process(clk, reset)
  begin
    if (reset='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end process;

end mult_seg_arch;
-- next-state logic
process(state_reg, mem, rw, burst)
begin
  case state_reg is
    when idle =>
      if mem='1' then
        if rw='1' then
          state_next <= read1;
        else
          state_next <= write;
        end if;
      end if;
    else
      state_next <= idle;
    end if;
    when write =>
      state_next <= idle;
  end case;
end process;
when read1 =>
    if (burst='1') then
        state_next <= read2;
    else
        state_next <= idle;
    end if;
when read2 =>
    state_next <= read3;
when read3 =>
    state_next <= read4;
when read4 =>
    state_next <= idle;
end case;
end process;
-- moore output logic
process (state_reg)
begin
  we <= '0';  -- default value
  oe <= '0';  -- default value
  case state_reg is
    when idle =>
    when write =>
      we <= '1';
    when read1 =>
      oe <= '1';
    when read2 =>
      oe <= '1';
    when read3 =>
      oe <= '1';
    when read4 =>
      oe <= '1';
  end case;
end process;
— mealy output logic

process (state_reg, mem, rw)
begin
  we_me <= '0'; — default value
  case state_reg is
    when idle =>
      if (mem='1') and (rw='0') then
        we_me <= '1';
      end if;
    when write =>
    when read1 =>
    when read2 =>
    when read3 =>
    when read4 =>
      end case;
  end process;
end mult_seg_arch;
Alternative Coding Style 2

Process(Present State, Inputs)

Process(clk, reset)
process(state_reg, mem, rw, burst)
begin
    oe <= '0';  --- default values
    we <= '0';
    we_me <= '0';
    case state_reg is
    when idle =>
        if mem='1' then
            if rw='1' then
                state_next <= read1;
            else
                state_next <= write;
                we_me <= '1';
            end if;
        else
            state_next <= idle;
        end if;
    when write =>
        state_next <= idle;
        we <= '1';
    end case;
end
when read1 =>
    if (burst='1') then
        state_next <= read2;
    else
        state_next <= idle;
    end if;
    oe <= '1';
when read2 =>
    state_next <= read3;
    oe <= '1';
when read3 =>
    state_next <= read4;
    oe <= '1';
when read4 =>
    state_next <= idle;
    oe <= '1';
end case;
end process;