Sequential Circuits – Part 1

- Clocked sequential circuits and state machines
- From circuit description to state diagram
- From state diagram to VHDL specification
- Verifying state machines

Jon Turner
Clocked Sequential Circuits

- In *sequential* circuits, output values may depend on both current inputs and stored information
  - *clocked sequential circuits* store information in *flip flops*
  - stored information referred to as *state* of the circuit
- *Next state logic* determines new stored values
- Synchronous outputs depend only on stored values and change following clock edge
- Asynchronous outputs can change as inputs change
Fair Arbiter

- Controls access to a shared resource
  - inputs: request0, request1
  - outputs: grant0, grant1
  - grants asserted in response to inputs
  - if requests from both clients, favor least-recently-served

CLK
r0
r1
g0
g1

idle0/00
bus0/10
bus1/01
idle1/00

00
1x
1x

r0 r1
x0 01 10 0x
x1 g0 g1
00
State Tables and State Diagrams

- The fair arbiter is an example of a state machine.
- The behavior of a state machine can be conveniently specified using a state diagram.
- A state table is an equivalent representation.

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input r0 r1</th>
<th>Output g0 g1</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle0</td>
<td>0 0</td>
<td>0 0</td>
<td>idle0</td>
</tr>
<tr>
<td></td>
<td>1 x</td>
<td>0 0</td>
<td>busy0</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>0 0</td>
<td>busy1</td>
</tr>
<tr>
<td>busy0</td>
<td>1 x</td>
<td>1 0</td>
<td>busy0</td>
</tr>
<tr>
<td></td>
<td>0 x</td>
<td>1 0</td>
<td>idle1</td>
</tr>
<tr>
<td>idle1</td>
<td>0 0</td>
<td>0 0</td>
<td>idle1</td>
</tr>
<tr>
<td></td>
<td>x 1</td>
<td>0 0</td>
<td>busy1</td>
</tr>
<tr>
<td>busy1</td>
<td>x 1</td>
<td>0 1</td>
<td>busy1</td>
</tr>
<tr>
<td></td>
<td>x 0</td>
<td>0 1</td>
<td>idle0</td>
</tr>
</tbody>
</table>
entity fairArbiter is port(
  clk, reset, request0, request1: in std_logic;
  grant0, grant1: out std_logic);
end fairArbiter;
architecture a1 of fairArbiter is
  type stateType is (idle0, idle1, busy0, busy1);
  signal state: stateType;
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset = '1' then
        state <= idle0;
      else
        case state is
          when idle0 => if request0 = '1' then state <= busy0;
          elsif request0 = '1' then state <= busy0;
          end if;
          when busy0 => if request0 = '0' then state <= idle0; end if;
          ...
        end case;
      end if;
    end process;
    grant0 <= '1' when state = busy0 else '0';
    grant1 <= '1' when state = busy1 else '0';
  end a1;
Circuit Implementing VHDL Spec
State Diagrams for Asynchronous Outputs

- State machines often have outputs that depend directly on inputs
  - such outputs may change when inputs changes, not just on clock
- Alternate form of state diagram
  - output values attached to arrows
  - output has specified value when in “from state” and inputs have specified value
- New output equation
  - \( g_0 = (\text{state}=\text{idle0}) \cdot r_0 \) or \( (\text{state}=\text{idle1}) \cdot r_0 r_1' \) or \( (\text{state}=\text{busy0}) \cdot r_0 \)
- Circuits with all synchronous outputs called Moore-mode
- Circuits with \( \geq 1 \) asynchronous outputs called Mealy-mode
Exercises

1. Consider an implementation of the arbiter circuit in which the state is represented using a pair of bits. Specifically, 00 is used to represent the idle0 state, 01 is used for idle1, 10 for busy0 and 11 for busy1. Write a pair of logic equations to define the inputs of the two state flip flops, as a function of the current state signals and the two inputs. You may find it useful to start by first rewriting the state table using the assigned binary values in place of the symbolic state names.

2. Consider a 3-way fair arbiter, in which there are three request inputs and three grant outputs. We again want the arbiter to be fair, so it should give priority to the least-recently-served client whenever two or more clients make simultaneous requests. Make a list of states that can be used to implement this circuit. How many states are needed for an n-way arbiter, based on this approach?

3. We can reduce the number of states required for a 3-way fair arbiter by modifying the fairness policy a little bit. In this new version, if client 0 used the resource most recently, client 1 is given top priority, followed by client 2. If client 1 used the resource most recently, client 2 is given top priority, followed by client 0. If client 2 used the resource most recently, client 0 is given top priority, followed by client 1. Make a list of states that can be used to implement this circuit with this approach. How many states are needed for an n-way arbiter, based on this approach?
Procedure for Designing State Machines

- Make sure you understand specification
  - read description carefully and rephrase in your own words
  - if no interface timing diagram given, make one
  - determine if outputs are synchronous or asynchronous
  - ask questions
- Determine what the circuit must “remember”
  - choose state names and write down what they mean
- Construct appropriate state diagram
- Write VHDL module implementing state diagram
- Simulate circuit to verify operation
  - construct inputs to visit every state, follow each transition
From State Diagrams to VHDL

- State diagram is often used to specify the behavior of a state machine
- Easy to construct VHDL code from state diagram
  » note, no code needed for “self-loops”

```vhdl
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset = '1' then
        state <= desired initial value
      else
        case state is
        when state0 => logic for transitions from state0
        when state1 => ...
        end case;
      end if;
    end process;
    out0 <= logic specifying out0
    out1 <= logic specifying out1
    end all;
```

- initialization
- next state logic
- output logic
- outputs assigned within scope of synchronization condition are delayed until next tick
- for complex output logic, use separate combinational process
Designing a Garage Door Opener

- 4 input signals
  - open/close, obstruction, at top, at bottom
- 2 output signals
  - raise door, lower door
- Desired behavior
  - if obstruction detected when closing, reverse direction
  - if open/close signal is received while opening or closing the door, it should pause; next open/close reverses
- What must the circuit remember?
  - is door open, closed, opening, closing or paused?
  - if door is paused, did it pause on way up, or way down?
  - how many states?
Interface Timing (incomplete)
State Machine for Garage Door Opener

[Diagram showing state transitions and conditions for opening, closing, and other states]
VHDL for Garage Door Opener

entity opener is port (  clk, reset: in std_logic;  openClose: in std_logic; -- signal to open or close door  obstruction: in std_logic; -- obstruction detected  atTop: in std_logic -- door at the top (fully open)  atBot: in std_logic; -- door at the bottom (fully closed)  goUp: out std_logic; -- raise door  goDown: out std_logic); -- lower door end opener;

architecture al of opener is

type stateType is (opened, closed, opening, closing,  pauseUp, pauseDown, resetState);

signal state: stateType;

begin

process (clk) begin

if rising_edge(clk) then

if reset = '1' then

state <= resetState;

end if;

end if;

end process;

end al;
elsif state = resetState then
  if atTop = '1' then state <= opened;
  elsif atBot = '1' then state <= closed;
  else state <= pauseDown;
  end if;
else
  case state is
  when opened =>
    if openClose = '1' and obstruction = '0' then
      state <= closing;
    end if;
  when closing =>
    if openClose = '0' and obstruction = '0' and atBot = '1'
      then state <= closed;
    elsif obstruction = '1' then state <= opening;
    elsif openClose = '1' then state <= pauseDown;
    end if;
  ...
  when others =>
    end case;
  end if;
end if;
goUp  <= '1' when state = opening else '0';
goDown <= '1' when state = closing else '0';
Simulation of Garage Door Opener

- enter opened state from resetState
- typical open/close cycle
Simulation of Garage Door Opener

typical pause operations

obstruction detection
Testbench for Garage Door Opener

reset <= '1'; inputs <= "0010"; wait for pause;
reset <= '0'; wait for pause;

-- first do a "normal" close/open cycle
inputs<="1000"; wait for clk_period; inputs<="0000"; wait for clk_period;
inputs<="0001"; wait for clk_period; inputs<="0000"; wait for clk_period;
inputs<="1000"; wait for clk_period; inputs<="0000"; wait for clk_period;
inputs<="0010"; wait for clk_period; inputs<="0000"; wait for pause;

-- now check out the typical pause cases, and obstruction-detected
inputs<="1000"; wait for clk_period; inputs<="0000"; wait for clk_period;
inputs<="1000"; wait for clk_period; inputs<="0000"; wait for clk_period;
inputs<="1000"; wait for clk_period; inputs<="0000"; wait for clk_period;
inputs<="1000"; wait for clk_period; inputs<="0000"; wait for clk_period;
inputs<="0100"; wait for clk_period; inputs<="0000"; wait for clk_period;
inputs<="0010"; wait for clk_period; inputs<="0000"; wait for pause;
Exercises

1. Draw an interface timing diagram showing what happens when the garage door opener pauses on its way down, then reverses direction.
2. Make a state transition table for the garage door opener.
3. Extend the testbench for the garage door opener to exercise all of the transitions out of the reset state.
4. Suppose a synthesizer implemented the garage door controller using a one-hot encoding of the states. That is, for each state, there is a flip flop which is high when the controller is in that state and low otherwise. For each state, write a logic equation that for the input signal to that flip flop. For example, for the closed state, the equation would be
   \[ \text{closed}_\text{next} = (\text{closed} \land \neg \text{openClose}) \lor (\text{closing} \land \text{atBot} \land (\neg \text{openClose}) \land (\neg \text{obstruction})) \lor (\text{resetState} \land \text{atBot}) \]
5. Write a VHDL specification for a state machine based on the state diagram at right.