Two Process FSM Tutorial

November 4, 2007

1 Objective

The goal of this tutorial is to explain how to create a Two Process Finite State Machine (FSM). While there are a variety of methods to creating finite state machines, this method provides synthesizable VHDL for use with FPGAs. The tutorial will explain the steps of modifying an existing IP Hardware Core template (look at *Creating a Hardware Core* tutorial).

2 Getting Started

This tutorial assumes you have already built a base system and have gone through and created a Hardware Core through **Create or Import Peripheral**. Make sure your core has at least 3 Software Accessible Registers (32-bits each) as we will use 3 for our example. If you have more that is ok, if you have less, go and create a new hardware core. Navigate to the core's VHDL folder which contains *user_logic.vhd*. Make a backup copy of the file.

```
cp user_logic.vhd user_logic.vhd_bkup
```

This tutorial will create a hardware core that will perform a look-up from a table, increment the value by 5 and store the result in a software addressable register. This requires a state machine because we will either be in some idle state, waiting for the processor to write a look-up value (the index for the table) or processing the data (adding 5 to the output of the table).

3 Modifying Signals Name

Using your favorite Text Editor (cough - Emacs!) open *user_logic.vhd*. Xilinx's Hardware Core template provides basic read and write functionality to your three software addressable registers. Xilinx's names these registers *slv_reg0*, *slv_reg1*, *slv_reg2*. These names don't provide a lot of meaning so we will rename them to be more meaningful.

- Rename **slv_reg0** to **control_reg**
- Rename **slv_reg1** to **data_in_reg**
- Rename **slv_reg2** to **calc_reg**

It is important to rename **all** instances of the signals. The signals are like variables in C. Feel free to rename these registers to whatever makes the most sense to you or fits within your own coding style. Understand that these were renamed for this specific example.

4 Adding Additional Signals

It is possible to add additional signals to your User Logic. These signals; however, will not be software accessible. You can think of them as local variables to your hardware core. The first three software accessible registers are more like global variables. The local signals will allow us to register the transition between signals in order to synchronize them to the clock.

We will add these signals within the *user_logic.vhd* under the Architecture IMP declaration (hint, look around line 124). You will see the following comment:

```
--USER signal declarations added here, as needed for user logic
```

Add your own signals below this line. The signals we need for this example are:

```
signal table_index    : std_logic_vector(0 to 3);
signal table_data_out : std_logic_vector(0 to 31);
signal calc_reg_next   : std_logic_vector(0 to 31);
type CALC_CNTL_SM_TYPE is ( idle, calc ) ;
signal current_state, next_state : CALC_CNTL_SM_TYPE := idle;
```

Table Index will be a number between decimal 0 and 7 that we calculate based on the user's Data In value. It will be used by a separate process that will calculate the Table Data Out value. We will then add 5 to this Table Data Out value. Calc Reg Next is the registered value of the calculation. In order to synchronize the transition between calc_reg and the clock we need to explicitly state the value for calc_reg (the current value) and calc_reg_next (the value of calc_reg during the next clock cycle).

Finally we need to create a State Machine type. This is like an enumeration in C. The CALC_CNTRL_SM_TYPE (Calculation Control State Machine Type) can either be in the **idle** or **calc** state. That means we have two states in our state machine. This is; however, only the type of the state machine. We need to create two signals that will actually hold the current state and next state of the state machine. They will both be of type CALC_CNTL_SM_TYPE and be initialized to idle.

5 Adding FSM Functionality

If you look over the *user_logic.vhd* file you will see that there are currently two processes created:

1. SLAVE_REG_WRITE_PROC
2. SLAVE_REG_READ_PROC

We want to add two processes of our own under these two processes. Look for the line (around line 221):

```
end process SLAVE_REG_READ_PROC;
```

After this line you will need to create the first Process of our Two Process Finite State Machine. The first process is going to control the state transitions and the current signal value to their next value. This will happen every clock cycle. To begin we need to create the process (Look at **Figure 1**).

The first line says we will create a process called *CALC_FSM_STATE_PROC*. It will be sensitive to the Clock and Reset signals from the Bus into our IP hardware core, the next state signal and the calculation register's next signal. Within the process you see the If statement on the Reset signal. This means if there

```
CALC_FSM_STATE_PROC : process ( Bus2IP_Clk, Bus2IP_Reset,
                                next_state, calc_reg_next ) is
begin
  if Bus2IP_Reset = '1' then
    -- Reset Calculation Register (Hardware Reset)
    calc_reg      <= (others => '0');
    current_state <= idle;
  elsif Bus2IP_Clk'event and Bus2IP_Clk = '1' then
    -- Update Signals
    current_state <= next_state;
    calc_reg      <= calc_reg_next;
  else
     -- Latch Signals (keep them the same value)
    current_state <= current_state;
    calc_reg      <= calc_reg;
  end if;
end process CALC_FSM_STATE_PROC;
```

Figure 1: Calculation Finite State Machine State Process

is a Reset issued from the Bus to our core it should set the value of the calculation register to all 0's (zeros) and set the current state back to the idle state.

The Else If clause is what is doing the synchronization (registering the signals). If there is a Bus Clock Event (meaning the Clock has either changed from low to high or high to low) and the value is 1 then that means we should update the signals to their "next" value. Otherwise we should keep them the same.
Next we need to write the Logic part of the Finite State Machine. This is the implementation portion of the FSM. We are telling the hardware what to do. We will create the second process of our Two Process Finite State Machine (look at **Figure 2**).

The Logic Process is in charge of doing the "work" of the FSM. Here we begin by assigning the next value of the signal to its current value. This is done with a Flip Flop. We do this for all signals that will change in the State Process. We also calculate the table_index by taking the data_in_registers 24 to 31 bits. We do not need to register the table_index because we constantly are calculating it and want it to calculate the next value immediately after data_in_reg changes.

The Case Statement looks at the current state and depending on which state it is, performs some logic. We will begin by discussing what each state is doing.

First the Idle state:

First we check the control register. This register allows the user reset the Calc Register by writing a '1' to control_register 31st bit. Otherwise, if that bit isn't set we will check to see if slv_reg_write_select has a value of "010" which means that data has been written to the Data In Register. If data has been written then we want to jump to the Calc State and perform the calculation. Otherwise we want to remain in the Idle State waiting for data to be written.

Second the Calc State:

Now all we have to do is perform the calculation. Since we are constantly calculating the Table Look-up value we know that the Table Data Out register contains the value we want to add 5 to. We simply add 5 and store it into the calc_reg_next register. This will update calc_reg on the next clock cycle. Then since calc_reg

```
CALC_FSM_LOGIC_PROC : process( current_state,
                  control_reg, data_in_reg ) is
begin
  -- Flip-Flop Signals
  next_state    <= current_state;
  calc_reg_next <= calc_reg;
  -- Constantly Calculate the Table Index
  table_index <= data_in_reg(28 to 31);
  -- State Machine
  case current_state is
   -- Idle State
    when idle =>
      if control_reg(31) = '1' then
        -- Reset calc (Software Reset)
        calc_reg_next <= (others => '0');
        next_state   <= idle;
      elsif slv_reg_write_select = ``010'' then
        -- Else If Data is written to Data In Register
        next_state <= calc;
      else
        -- Else Stay in Idle State
        next_state <= idle;
      end if;
    -- Calc State
    when calc =>
      calc_reg_next <= table_data_out + 5;
      next_state    <= idle;
    -- Other Case (Something Crazy Happened)
    when others =>
      calc_reg_next <= x''DEADDEAD'';
      next_state   <= idle;
  end case;
end process CALC_FSM_LOGIC_PROC;
```

Figure 2: Calculation Finite State Machine Logic Process

```
TABLE_LOOKUP_PROC : process( table_index ) is
begin
  case table_index is
    when x''0'' => table_data_out   <= x''00000000'';
    when x''1'' => table_data_out   <= x''00000001'';
    when x''2'' => table_data_out   <= x''00000002'';
    when x''3'' => table_data_out   <= x''00000003'';
    when x''4'' => table_data_out   <= x''00000004'';
    when x''5'' => table_data_out   <= x''00000005'';
    when x''6'' => table_data_out   <= x''00000006'';
    when x''7'' => table_data_out   <= x''00000007'';
    when others => table_data_out <= x''ffffffff'';
  end case;
end process TABLE_LOOKUP_PROC;
```

Figure 3: Table Look-up Process

is software accessible it will be able to be read from the PowerPC.

Others:

We need an others state just in case something crazy happens (we should never transition to this state, but if we ever read back DEADDEAD we should be suspicious of our hardware core's code.

6 Adding Look-up Table

Now before we can put it all together we need to add a process that will contain the Look-up Table. We do this by simply creating another process called TABLE_LOOKUP_PROC which will only be sensitive to table_index. Then depending on the table_index we will output a specific value to table_data_out which will be used by our FSM (Look at **Figure 3**).

7 Putting It All Together

Now you could connect your hardware core to your base system, generate addresses, and write an application to drive it. Your application could write data to the Data In Register and then Read back from the Calc Register and you should see a result you expect.