Fundamentals of Reconfigurable Computing ECGR6090/8090

IMPLEMENTING CRC IN HARDWARE

1 Objective

The goal of this lab is to implement the CRC algorithm that you implemented in software, in hardware. With your understanding of how the algorithm works, you will create a Standalone system (not a Linux system) and use a two process VHDL state machine to produce the CRC on a string of input from the user. While this may seem complex, in actuality the solution is relatively straightforward.
The goal is to come away with:

- implementing CRC Checksum in hardware
- implementing a 2 process state machine in hardware
- understanding one method of porting software to hardware

2 Getting Started

To get started you will need to create a new folder called **lab5**. Unlike Lab 4 you will create a new base system and write a standalone application that will run on the PowerPC. You will **not** be using Linux, you will first design the hardware then in Lab 6 you will combine Linux with your hardware via a Device Driver. If necessary, look back at the *Creating an Application* tutorial to refresh your memory on how to create and implement a standalone application with your base system. **Read these directions carefully and completely before you begin**.

1. Creating the Base System

Within your **lab5** folder create a folder called **build**. Using Xilinx's EDK create a new project called **lab5.xmp** in the **build** directory with the following Base System configuration:

- PowerPC running at 300 MHz
- Keep JTAG Debugging selected
- 64 KB of Data OCM and 64 KB of Instruction OCM

- RS232_Uart - OPB **UARTLITE** (No Interrupt)
- Uncheck DDR, SPI_EEPROM, LEDs, LCD, PCI, SysAce, and IIC
- Remove PLB Block RAM
- Uncheck Memory Test and Peripheral Test

2. Creating the Hardware Core

Next you will need to create a new Hardware Core called: **my_crc_calc** and add it to the base system. If necessary review the *Creating a Peripheral* tutorial. Create a hardware core with the following configuration (version 1.00.a):

- Attach your core to the OPB
- The core should only have *User Logic S/W register support* selected (uncheck *S/W reset and MIR* and *User logic interrupt support*)
- Add 3 (32-bit) Registers with Posted Write Behavior Enabled
- The rest of the options are default options (just click Next then Finish)

3 Implementation

Following the *Two Process Finite State Machine* tutorial online you will need to modify the default *user_logic.vhd* file. Be very careful when you are making changes. It would be a good idea to make a backup of your *user_logic.vhd* file before making any changes.

1. Modifying Hardware Core for CRC Calculation

To begin, rename the slave register signals (slv_regX) to meaningful names. Signals are basically variables so make sure to rename ALL of the instances of that signal.

- Rename All **slv_reg0** to **control_reg**
- Rename All **slv_reg1** to **crc_reg**
- Rename All **slv_reg2** to **data_in_reg**

Next, add the following additional User Signals (look in user_logic.vhd and see if you can find the comment):

```
--USER signal declarations added here, as needed for user logic
signal table_index    : std_logic_vector(0 to 7);
signal table_data_out : std_logic_vector(0 to 31);
signal crc_reg_next   : std_logic_vector(0 to 31);
type CRC_CNTL_SM_TYPE is ( idle, crc_calc ) ;
signal current_state, next_state : CRC_CNTL_SM_TYPE := idle;
```

You need an 8 bit signal called *table_index* which will be used to look-up a value in the CRC Table (why is it 8 bits? Hint, what is 2^8). The CRC Table will output *table_data_out*. Since you are creating a two process state machine you need to register the signals that change within the finite state machine so you add the signal *crc_reg_next*. Finally, you need to create the signals for the state machine (*current_state, next_state*. As you can see, the state machine only consists of two states: *idle* and *crc_calc*.

2. Adding a Two Process Finite State Machine

Next create two processes which will be your "Two Process Finite State Machine." The State Process controls the transition of signals from their current value to their next value. The Logic Process performs the specific functionality of the FSM.

1. CRC_CALC_FSM_STATE_PROC
2. CRC_CALC_FSM_LOGIC_PROC

Within STATE_PROC:

- If Reset Set Current State to Idle and reset crc_reg
- Else If There is a Clock Event and Clock is 1
  - Set Current State equal to Next State
  - Set CRC Register equal to CRC Register Next signal

Within LOGIC_PROC:

- Latch Signals
- Constantly Calculate Table Index
- Case Statement on the Current State
  - Idle State
    If Control Register(31) = '1' then reset CRC Register
    Else If New Data has been written to *data_in_reg* move to the CRC Calc State
  - CRC Calculation State
    Perform calculation and then return to Idle State

3. Adding a Table Look-up Process

While there are a variety of methods for implementing a table look-up a common approach for small tables is to use an asynchronous process with a case statement. Look at the tutorial if you are unfamiliar with case statements within processes. For example:

```
case table_index is
  when x"00" => table_data_out <= x"00000000";
  when x"01" => table_data_out <= x"04c11db7";
  ...
  when x"ff" => table_data_out <= x"b1f740b4";
  when others => table_data_out <= x"ffffffff";
end case;
```

4. Reading Back CRC Result

The result of the CRC calculation is stored within crc_reg and is a software addressable resigster. If you look at the Slave Read Register Process you will see that when software reads from the core's base address + 0x04 then crc_reg is written to the bus (which in turn is written to the requester). This is fine, but you will need to modify this line to complete the CRC calculation. (**Hint:** look at your software implementation and see what you do to the crc value right before you returned it. You will need to do the same thing here right before it is written out to the bus). If you miss this step you could have implemented everything else correctly, but because you didn't do this one "small" thing you will not get the right answer so be careful and do not get frustrated, be patient and you will figure it out.

4 Hooking up Hardware Core to Base System

Now you need to add **my_crc_calc** hardware core to the base system and connect it to the OPB. You also need to generate the address. For this lab, it is ok to click "Generate Addresses" but make sure you know what addresses change. If you incorrectly connect your hardware core to the OPB or do not give it an address range that it (and the OPB) know about, when you try to read or write data to your core it will not respond.

5 Creating Standalone Application

Create a standalone (.elf) application (like you did in Labs 1 and 2). You also will need to Generate the Linker Script. The program should work as follows:

- Print Your Name and ID
- Repeatedly prompt User to enter a single character
- Write the character to your hardware core's Data In Register
- Increment a counter keeping track of the number of characters entered
- When the user enters 0 (zero) write the new line character '\n' to your Hardware Core, increment the counter and then exit the loop
- Write the counter value's to your hardware core's Data In Register
- Print the number of characters entered by the user
- Print the final CRC Result from your hardware core's CRC Register

Example of Output:

```
Name: Bond, James Bond
ID: 007
Enter Character (0 to Exit):
Entered: H
Enter Character (0 to Exit):
Entered: e
Enter Character (0 to Exit):
```

```
Entered: l
Enter Character (0 to Exit):
Entered: l
Enter Character (0 to Exit):
Entered: o
Enter Character (0 to Exit):
Entered: !
Enter Character (0 to Exit):
Entered: 0
You Entered 7 Characters
Calculated CRC: 113449826
```

Finally, create an ACE file and download it to the FPGA. Verify the functionality by looking at the Example above and reading the Reference Section at the end. Additionally, you could modify the C-code you wrote for Lab 4 to accept input from the user instead of reading from a file. This is a good exercise (but not required) if you feel you need more practice with C.

6 Grading

This lab assignment is due **Tuesday November 20th, 2007 by 5pm (EST)**. To receive credit, you must meet with either the T.A. or the instructor and demonstrate that you've completed the lab. By default, this can be done in Woodward Hall room 237 (the Unix lab). Alternative times are possible but need to be arranged in advance. Be prepared to answer in person any questions in the lab or recompile your program as part of demonstrating that you completed the lab. Do not wait until the last minute to begin the project; extensions will not be granted.

7 References

On Homer you can use **cksum** like you did in Lab 4 to verify functionality, except this time rather than running cksum on a file, you could run it on Standard Input. To do this simply type **cksum** and hit **enter**. Then type any characters you want to type. Then to calculate the CRC value press **control + d**. For example:

```
> cksum
Hello!
113449826 7
```

You will notice that the string "Hello!" actually consists of 6 characters; however, after the ! character there is a New Line character '\n' which then adds up to 7 characters. If you compare the first number (113449826) to the sample output for the application you will see they are the same.