2019

ERTS Assignment 2

BUILDING SOPC DESIGNS ON A FPGA PLATFORM

JONAS KJÆR RASK

MADS KRABSEN

Description

In this exercise, you will learn how to develop embedded designs with the ZYBO platform building customized hardware and software.

Goals

- Learn about Xilinx Design Tools including Vivado, SDK and HLS.
- Learn the design flow for system on a programmable chip.
- Learn about configuring the Processing System (PS).
- Learn about the Programmable Logic (PL) and connecting to the PS.
- Learn about interface driver for USB-UART, GPIO and TIMER.
- Construct a simple command line parser to control hardware.
- Learn how to add Custom IPs to your design.
- Learn how to use High Level Synthesis to simulate C/SystemC and generate HDL code.

Exercise 3

Write a C-application that implements a command language interpreter, controlled via the USB-UART interface. The following commands must be implemented:

1) Sets the binary value from 0-15 on the red led's by reading switch input (SW0-SW3)
2) Counts binary the red led's using a timer of 1 sec

Solution

To read from the USB-UART we do the following

```
// Read UART input if any available
if (XUartPs_IsReceiveData(STDIN_BASEADDRESS))
{
   xil_printf("CMD:> ");
   value = inbyte();
   xil_printf("%c\r\n",value);
}
```

This saves the UART character pressed in the variable *value*. This is then used in a switch case for the commands.

Command 1 – SW->Leds

This command is implemented as

```
case '1':
   dip_check = XGpio_DiscreteRead(&dip, 1);
   LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR,0,dip_check);
   break;
```

The command reads the state of the GPIOs corresponding to the switches and writes these to the register connected to the LEDs.

Command 2 – Timer->Leds

For this command a timer is started that increments every second this is done using:

```
// PS Timer related definitions
XScuTimer_Config *ConfigPtr;
XScuTimer *TimerInstancePtr = &Timer;
int32_t Status;

⋮
⋮

// Init the timer
ConfigPtr = XScuTimer_LookupConfig (XPAR_PS7_SCUTIMER_0_DEVICE_ID);
Status = XScuTimer_CfgInitialize (TimerInstancePtr, ConfigPtr, ConfigPtr->BaseAddr);
if (Status != XST_SUCCESS){
   xil_printf("Timer init() failed\r\n");
   return XST_FAILURE;
}
XScuTimer_LoadTimer(TimerInstancePtr, ONE_SECOND); // Load timer with delay in multiple of ONE_SECOND
XScuTimer_EnableAutoReload(TimerInstancePtr); // Set AutoLoad mode
XScuTimer_Start(TimerInstancePtr); // Start the timer

⋮
⋮                                                              ⋮

while (run)
{
   ⋮
   ⋮

   // Check timer expired
   if(XScuTimer_IsExpired(TimerInstancePtr)) {
      // clear status bit
      XScuTimer_ClearInterruptStatus(TimerInstancePtr);
      // Increment counter
      counter = counter + 1 % 0x0F;
   }
}
```

This will create a timer that expires every one second. In the while loop we then have a statement that checks if the timer has expired. If it has the status flag is cleared and an internal counter is incremented.

When command 2 is active *counter* is written to the LEDs' register in the same way as for command 1.

Results

Visual inspection of the commands was done on the Zybo board. This confirmed that the commands perform as described above.

Exercise 4

The matrix functions have been implemented in the files matrix.h and matrix.c. Below snippets from the project files has been inserted.

Task 1

Create three global variables of the data structure ***vectorArray*** called ***pInst***, ***aInst*** and ***bTInst***.

These has been created in the main file.

Task 2

Implement a function called ***setInputMatrices*** that fills out the data structures with the input matrix values below:

$$aInst = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \qquad bTInst = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

The ***setInputMatrices*** function is implemented as follows:

```
void setInputMatrices(VectorArray A, VectorArray B)
{
   for (int i=0; i<MSIZE*MSIZE; i++)
   {
      A[i/MSIZE].comp[i%MSIZE] = i+1;
      B[i/MSIZE].comp[i%MSIZE] = (i/MSIZE)+1;
   }
}
```

Integer division and modulus is used to index into the matrices.

Task 3

Implement a function called ***displayMatrix*** that displays a 4x4 matrix via the USB-UART interface. Test it on matrix ***aInst*** or ***bTinst***.

The ***displayMatrix*** function is implemented as follows:

```
void displayMatrix(VectorArray input)
{
   for (int i=0; i<MSIZE*MSIZE; i++)
   {
      xil_printf("%3d ", input[i/MSIZE].comp[i%MSIZE]);
      if (i % MSIZE == MSIZE-1)
         xil_printf("\r\n");
   }
}
```

This makes sure that the matrix has the same size in the console as long as the elements are no longer than 3 digits.

The result of the function with input aInst is:

```
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16
```

Figure 1 - Terminal output from displayMatrix(aInst)

Task 4

Implement a function called ***multiMatrixSoft*** that computes the 4x4 matrix product of the expression: *P = A× BT*

The ***multiMatrixSoft*** function is implemented as follows:

```
void multiMatrixSoft(VectorArray A, VectorArray B, VectorArray P)
{
   for (int i=0; i<MSIZE; i++)
   {
      for (int j=0; j<MSIZE; j++)
      {
         P[i].comp[j] = 0;
         for (int k=0; k<MSIZE; k++)
         {
            P[i].comp[j] += A[i].comp[k]*B[j].comp[k];
         }
      }
   }
}
```

This function loops through all the elements of the P matrix calculating the sum of products for the corresponding rows from matrix A and B as if performing the expression: $pInst = aInst \times bTInst^T$

Task 5

Add a command called ***mult*** that multiplies ***aInst*** and ***bTinst*** matrices with the values above. Display the result P and the execution time (in clock ticks) of the function ***multiMatrixSoft*** in the USB UART window

The ***mult*** command is implemented as:

```
XScuTimer_RestartTimer(TimerInstancePtr);              //Clear Timer
tBefore = XScuTimer_GetCounterValue(TimerInstancePtr);//Read timer before
multiMatrixSoft(aInst,bTInst,pInst);                   //Do multiplication
tAfter = XScuTimer_GetCounterValue(TimerInstancePtr); //Read timer after
displayMatrix(pInst);                                  //Print matrix P
xil_printf("Execution time: %d [ticks]\r\n",tBefore-tAfter);
```

The command uses the same timer as in Exercise 3, that is started during initialization and never stopped. To make sure that the timer doesn't overflow during the timing of the operation it is reset before performing the multiplication.

The variables tBefore and tAfter is used to store the counter values before and after performing the multiplication operation. The execution time (in clock ticks) is then calculated as tBefore-tAfter

Result

Executing the ***mult*** command yields:

```
CMD:> 3
 10  20  30  40
 26  52  78 104
 42  84 126 168
 58 116 174 232
Execution time: 1657 [ticks]
```

*Figure 2 - Terminal output from executing the **mult** operation*

As it is seen the ***multiMatrixSoft*** operation takes an approximate 1657 clock cycles to complete.

Exercise 5

1. Study the implemented VHDL **matrix_ip** component in **matrix_ip_v1_0_S_AXI.vhd**
2. Use the guide "**lab3b_ EmbeddedSystem.pdf**" for how to add a customized IP to the hardware design. Unzip and use the **matrix_ip_1.0.zip** that contains the matrix IP core with user logic to perform multiplication and additions as described above.
3. Create a new function called ***multiMatrixHard*** that computes the matrix product of the expression: $P = A \times B^T$ but use the new IP core **matrix_ip** to perform part of the matrix multiplication.
4. Display result P and the execution time (in clock ticks) of the function ***multiMatrixHard*** in the USB UART window and compare it to ***multiMatrixSoft.*** The ARM processor (PS) runs at 650 Mhz and the hardware matrix multiplication (PL) is clocking at 100 Mhz. The timer is running at half of the PS clock (325 Mhz). Evaluate the computation speed of software vs. hardware implementation based on measured execution time and clock speeds.

Solution

In the matrix_ip component it is seen that the module takes two vectors of each 4 bytes. The vectors are then multiplied byte wise, such that vector1 byte1 is multiplied with vector2 byte1, byte2 with byte2 and so forth.
The result of the multiplications is then added to form the i_sum, that is the sum of products. This is then written to the register to allow the software to access the result.

The matrix_ip is inserted into the solution from Exercise 4, which allows us to implement the function ***multiMatrixHard*** this is done as:

```
void multiMatrixHard(VectorArray A, VectorArray B, VectorArray P)
{
   for (int row=0; row<MSIZE; row++)
   {
      for (int col=0; col<MSIZE; col++)
      {
         Xil_Out32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR +
            MATRIX_IP_S00_AXI_SLV_REG0_OFFSET, A[row].vect);
         Xil_Out32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR +
            MATRIX_IP_S00_AXI_SLV_REG1_OFFSET, B[col].vect);
         P[row].comp[col] = Xil_In32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR +
            MATRIX_IP_S00_AXI_SLV_REG2_OFFSET);
      }
   }
}
```

The function loops through the elements of P where the result of each element is calculated using the matrix_ip.
The axi base address was found in the file xparameters.h and the offsets where found in the file matrix_ip.h

To record the execution time of the function ***multiMatrixHard*** a similar command as ***mult*** (from Exercise 4) is created.

Results

Execution of the function yields the output:

```
CMD:> 4
Hard Matrix Multiplication
 10  20  30  40
 26  52  78 104
 42  84 126 168
 58 116 174 232
Execution time: 3845 [ticks]
```

*Figure 3 - Terminal output from **multHard** command*

At it is seen execution of the function ***multiMatrixHard*** takes 3845 clock cycles. Which is a bit more than double the execution of ***multiMatrixSoft*** with an execution time of 1657 clock cycles. This can be explained by the fact that the ARM processor is running on a clock of 650 MHz which is faster than the clock of the matrix_ip of just 100 MHz.
This means that ***multiMatrixHard*** would be approximately 3 times faster than ***multiMatrixSoft***, if the whole system was running on the same clock speed.

Exercise 7

1. Write the ADVIOS IP core and a testbench in SystemC.
2. Document and verify the result of simulation and synthesis using Vivado HLS.
3. Connect the ctrl port to the AXI4Lite interface by using pragma as described in UG902.
4. Create a Vivado project and add the ADVIOS IP core and connect it to the LEDS and SWITCHES on the ZYBO board.
5. Write a program with the Xilinx SDK that verifies the functionality of the IP core and document the results.

ADVIOS IP

Solution

The ADVIOS IP launches two threads, running the functions iosThread() and iosPulseThread().
The iasThread() sets the LEDs, either to the switch values masked by the ctrl value or the counter value for ctrl = 0. Implemented as:

```
void iosc::iosThread() {
   //Group ports into AXI4 slave slv0
   #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=ctrl

   //Initialization
   wait();

   // Process the data
   while(true)
   {
      // Wait for start
      wait();

       if (ctrl.read() != 0x0)
       {
            switchs = inSwitch.read() & ctrl.read();
       }
        else
        {
            switchs = secCounter;
        }
      outLeds.write(switchs);
   }
}
```

The iosPulseThread() increments a counter every second, by counting the clocks.

```
void iosc::iosPulseThread()
{
   while (true)
   {
      if (++counter_clk >= 50000000)
      {
         secCounter++;
         counter_clk = 0;
      }
      wait();
   }
}
```

Results

The IP has been tested using a smaller compare value in ***iosPulseThread*** to decrease simulation time. The result is

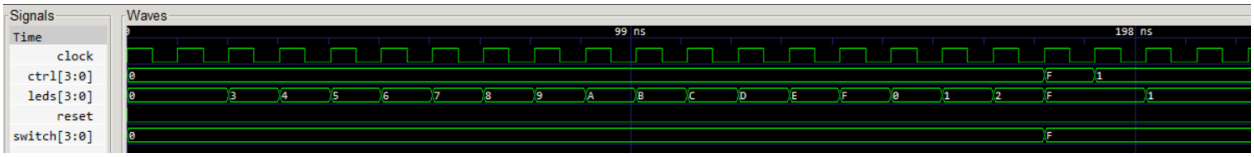

Figure 4 - Waveform of ADVIOS IP simulation

Vivado project

To verify the functionality of the ADVIOS IP a vivado project is created. In the project the ADVIOS IP is connected to the Switches and the LEDs.

A small test program has been created, where the *ctrl* value in the IP can be changed, this is done as:

```
XIosc_WriteReg(XPAR_IOSC_0_S_AXI_SLV0_BASEADDR, XIOSC_SLV0_ADDR_CTRL_DATA, 0xf);
```

Results

The verification is done by visual inspection, where it is verified that the ip performs as expected.