Aarhus Universitet

ERTS - Group 2

Rapport

Assignment 2

Gruppemedlemmer:	**Studienumre:**
Daniel Tøttrup	201509520
Mathias Lønborg Friis	201505665
Stinus Lykke Skovgaard	201401682

6. oktober 2019

Kapitel 1

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.**

1.1 Command 1

The essence of the first command is to set a binary value from 0-15 by using the four switches, which will be represented as a binary value by the four LED's on the board.

The code snippet below shows how we read input from the terminal, and execute the a command based on that input.

```
1  //read input from the terminal in byte
2  int userInput = inbyte();
3  xil_printf("Received: %d\r\n", userInput);
4
5  switch(userInput)
6  {
7  case (int)'1':
8  xil_printf("Received command 1\r\n");
9  execute_command_1();
10 break;
11 case (int)'2':
12 xil_printf("Received command 2\r\n");
13 execute_command_2();
14 break;
15 case (int)'3':
16 xil_printf("Received command 3\r\n");
17 execute_command_3();
18 break;
19 case (int)'4':
```

```
20 xil_printf("Received command 4\r\n");
21 execute_command_4();
22 break;
23 default:
24 xil_printf("Received unknown command.");
25 break;
26 }
```

Below is a code snippet of the implemented first command.
When the function **execute_command_1()** is called the switches on the board is read with the function **XGpio_DiscreteRead(&dip, 1)**. Where &dip is the InstancePtr that points to the XGpio instance that is being worked on, and the '1' is the channel.

The **writeValueToLEDs(int val)** takes the value returned from the **XGpio_DiscreteRead()** and writes the value to the LED registers.

```
1  void execute_command_1()
2  {
3  xil_printf("Executing command 1\r\n");
4  while(1)
5  {
6  dip_check = XGpio_DiscreteRead(&dip, 1);
7
8  writeValueToLEDs(dip_check);
9
10 }
11 }
12
13 void writeValueToLEDs(int val)
14 {
15 LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, val);
16 }
```

Command 1 work as expected.

1.2 Command 2

Command 2 should, as states in the problem description, count the binary red LED's using the timer of 1 sec.

First the timer needs to be initialized. Below is a code snippet of how the timer is implemented. Comments in the code explains the implementation.

```
#define ONE_SECOND 325000000 // half of the CPU clock speed

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

int main (void)
{
//initialize timer
ConfigPtr = XScuTimer_LookupConfig (XPAR_PS7_SCUTIMER_0_DEVICE_ID);
s32 Status = XScuTimer_CfgInitialize (TimerInstancePtr, ConfigPtr,
    ConfigPtr->BaseAddr);
if (Status != XST_SUCCESS){
xil_printf("Timer init() failed\r\n");
return XST_FAILURE;
}

// Load timer with delay in multiple of ONE_SECOND
XScuTimer_LoadTimer(TimerInstancePtr, ONE_SECOND);
// Set AutoLoad mode
XScuTimer_EnableAutoReload(TimerInstancePtr);
}
```

The function **execute_command_2()** sets op a counter and writes the value of counter to the LED's with the function **writeValueToLEDs(counter)** which was explained above. Then it starts the timer, and waits for the timer to expire, which will take one second. When the timer has expired the timer is cleared and the counter is incremented, before the loop starts again. Below is a code snippet of command 2.

```
void execute_command_2()
{
xil_printf("Executing command 2\r\n");
int counter = 0;
while(1)
{
xil_printf("Counter: %d\r\n", counter);
writeValueToLEDs(counter);

// Start the timer
XScuTimer_Start(TimerInstancePtr);

// Wait until timer expires
while(!XScuTimer_IsExpired(TimerInstancePtr));

// clear status bit
XScuTimer_ClearInterruptStatus(TimerInstancePtr);

counter++;
```

```
20
21 // Timer auto-enables
22
23 }
```

The function **execute_command_2()** works as expected.

Kapitel 2

EXERCISE 4

This exercise will deal with the steps to implement a matrix multiplier.

At first three global variables of the data structure **vectorArray** called **pInst**, **aInst** and **bTInst** were created.

Next a function called **setInputMatrices()** was implemented. This function will initialize two matrices. Below is a code snippet of the implementation. The comments in the code explains how the each element in the matrices are initialized.

```
1  void setInputMatrices()
2  {
3  //Set matrix a
4  // A, row 0
5  aInst[0].comp[0] = 1;
6  aInst[0].comp[1] = 2;
7  aInst[0].comp[2] = 3;
8  aInst[0].comp[3] = 4;
9
10 // A, row 1
11 aInst[1].comp[0] = 5;
12 aInst[1].comp[1] = 6;
13 aInst[1].comp[2] = 7;
14 aInst[1].comp[3] = 8;
15
16 // A, row 2
17 aInst[2].comp[0] = 9;
18 aInst[2].comp[1] = 10;
19 aInst[2].comp[2] = 11;
20 aInst[2].comp[3] = 12;
21
22 // A, row 3
23 aInst[3].comp[0] = 13;
24 aInst[3].comp[1] = 14;
25 aInst[3].comp[2] = 15;
26 aInst[3].comp[3] = 16;
27
28 //Set matrix bT
29 // A, row 0
30 bTInst[0].comp[0] = 1;
31 bTInst[0].comp[1] = 2;
```

```
32  bTInst[0].comp[2] = 3;
33  bTInst[0].comp[3] = 4;
34
35  //Set matrix bT
36  // A, row 1
37  bTInst[1].comp[0] = 1;
38  bTInst[1].comp[1] = 2;
39  bTInst[1].comp[2] = 3;
40  bTInst[1].comp[3] = 4;
41
42  //Set matrix bT
43  // A, row 2
44  bTInst[2].comp[0] = 1;
45  bTInst[2].comp[1] = 2;
46  bTInst[2].comp[2] = 3;
47  bTInst[2].comp[3] = 4;
48
49  //Set matrix bT
50  // A, row 3
51  bTInst[3].comp[0] = 1;
52  bTInst[3].comp[1] = 2;
53  bTInst[3].comp[2] = 3;
54  bTInst[3].comp[3] = 4;
55  }
```

Next a function called **displayMatrix(vectorArray matrix)** was implemented. As the name of the function strongly implies, this function will display a given matrix in the terminal. To implement **displayMatrix(vectorArray matrix)** a function called **displayMatrixRow(vectorArray matrix, int row)** is implemented in addition. That's because we have to display the matrix row for row. Below is a code snippet of the implemented function.

```
1   void displayMatrix(vectorArray matrix)
2   {
3   xil_printf("[\r\n");
4   displayMatrixRow(matrix,0);
5   displayMatrixRow(matrix,1);
6   displayMatrixRow(matrix,2);
7   displayMatrixRow(matrix,3);
8   xil_printf("]\r\n");
9   }
10
11  void displayMatrixRow(vectorArray matrix, int row)
12  {
13  xil_printf("%d %d %d %d \r\n", matrix[row].comp[0],
        matrix[row].comp[1], matrix[row].comp[2], matrix[row].comp[3]);
14  }
```

Next a function called **multiMatrixSoft(vectorArray in1, vectorArray in2, vectorArray out)** was implemented, which is responsible for computing 4x4 matrix product of the expression

$$P = AxB^T$$

This function takes three input because its not possible to return a matrix. So the result matrix is given as the third input parameter, that way we can set the new value of the result matrix in the function. To iterate through all combinations of the result matrix P to calculate the sum of products of each element in P, three nested for-loops were used. Below is a code snippet of the **multiMatrixSoft(vectorArray in1, vectorArray in2, vectorArray out)**

```
1  void multiMatrixSoft(vectorArray in1, vectorArray in2, vectorArray
       out)
2  {
3    for (int row = 0 ; row < MSIZE; row++)
4    {
5      for (int col = 0 ; col < MSIZE; col++)
6      {
7        xil_printf("[%d,%d]\r\n", row, col);
8        for (int i = 0 ; i < MSIZE ; i++)
9        {
10         out[row].comp[col] += in1[row].comp[i] * in2[row].comp[i];
11       }
12     }
13   }
14 };
```

Kapitel 3

EXERCISE 5

The goal og this exercise is to integrate a IP core into the project. This IP core is capable of multiplying two matrices and output the result. First we have to add the core to the project. This is done by following the lab3b_ EmbeddedSystem.pdf guide.

When the IP core has been added a method for it also needs to be implemented. This can be seen in the code below.

```
1  void multiMatrixHard(vectorArray in1, vectorArray in2, vectorArray
       out)
2  {
3    for (int row = 0 ; row < MSIZE; row++)
4    {
5      for (int col = 0 ; col < MSIZE; col++)
6      {
7        xil_printf("[%d,%d]\r\n", row, col);
8
9        Xil_Out32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR +
             MATRIX_IP_S00_AXI_SLV_REG0_OFFSET, in1[row].vect);
10       Xil_Out32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR +
             MATRIX_IP_S00_AXI_SLV_REG1_OFFSET, in2[col].vect);
11       out[row].comp[col] = Xil_In32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR
             + MATRIX_IP_S00_AXI_SLV_REG2_OFFSET);
12
13     }
14   }
15
16 }
```

It can be seen that the code iterates the rows and columns of the matrices. This means that each element of the matrices are being multiplied and added. At line 11 it can be seen that the output of this is put into an output array, which is the result of the multiplication.

Lastly we will look into the execution time between the hardware and the software implementation. We do this by using the XScuTimer from before. It is initiated just before the multiMatrixSoft() method and when the program has finished the matrix calculations a new timestamp is extracted from the timer. These two numbers are subtracted from each other and this gives us the execution time for the method. The

same approach has been made for the multiMatrixHard() method.

```
1  xil_printf("Executing command 3\r\n");
2  xil_printf("Matrix multiplication\r\n");
3  setInputMatrices();
4
5  xil_printf("A = \r\n");
6  displayMatrix(aInst);
7
8  xil_printf("bT = \r\n");
9  displayMatrix(bTInst);
10
11 // Multiply matrices in SW.
12 xil_printf("Calculating P in SW\r\n");
13
14 XScuTimer_Start(TimerInstancePtr); //Start timer
15 multiMatrixSoft(aInst, bTInst,pInst);
16
17 int time_SW_post = XScuTimer_GetCounterValue(TimerInstancePtr); //
       Get elapsed time in clock cycles
18 int time_elapsed_SW = ONE_SECOND - time_SW_post; // Calculate
       elapsed time
19
20 xil_printf("P = \r\n");
21 displayMatrix(pInst);
22
23
24
25 // Multiply matrices in HW.
26 int time_HW_post;
27 setInputMatrices(); // Make sure that matrices are "reset".
28 xil_printf("Calculating P in HW\r\n");
29
30 XScuTimer_RestartTimer(TimerInstancePtr);  // Start the timer
31 multiMatrixHard(aInst, bTInst, pInst);
32 time_HW_post = XScuTimer_GetCounterValue(TimerInstancePtr); // Get
       elapsed time in clock cycles
33 int time_elapsed_HW = ONE_SECOND - time_HW_post; // Calculate
       elapsed time
34
35 xil_printf("P = \r\n");
36 displayMatrix(pInst);
37
38
39 //Display elapsed time of matrix multiplications
40 xil_printf("\r\n");
41 xil_printf("Multiplication times (clock cycles):\r\n");
42 xil_printf("SW: %d\r\n",time_elapsed_SW);
43 xil_printf("HW: %d\r\n",time_elapsed_HW);
```

The two variables **time_elapsed_SW** and **time_elapsed_HW** hold the amount of clock cycles used for the two different calculation methods. This can be recalculated into seconds by using the below equation:

$$TimeElapsedInSecs = \frac{ClockCyclesUsed}{FreqOfProcessor}$$

The result of this gives us

- Hardware = 9.7224ms
- Software = 9.7219ms

The hardware solution should in theory be faster, but because of the frequency of the CPU is faster than the FPGA the software solution is still the fastest.

Kapitel 4

EXERCISE 7