Indian Institute of Technology Kanpur

Undergraduate Project - III

EE491A

Implementation of an 8-bit Microprocessor and Signal Monitoring System

Author:
Pranav Kumar (13490)

Supervisor:
Dr. S. Qureshi

November 8, 2016

Contents

1 Abstract **3**

2 Introduction **3**

3 Relevance **4**

4 Part I - Microprocessor Basics **4**
4.1 Instruction Set Architecture 4
4.2 Datapath 5
4.3 Memory 7
4.4 Bits of the Control Word 8

5 Part 1 - Microprocessor Implementation and Results **9**
5.1 Multiplexer results 9
5.2 Decoder results 10
5.3 Register results 10
5.4 Register File results 10
5.5 Memory results 11
5.6 Microprocessor results 11

6 Part II - Signal Monitoring System Basics - XADC Functionality **12**
6.1 XADC Operating Modes 13
6.2 Input Signal Types 13
6.3 Status Registers 14
6.4 Control Registers 15
6.5 Instantiating the XADC 15

7 Part II - Signal Monitoring System Realization **17**
7.1 Implementation 17
7.2 Results 17

8 Future Work **20**

9 References **20**

10 Appendix A - Code for part I **21**
10.1 Multiplexer (mux.v) 21
10.2 Register (register.v) 21
10.3 Decoder (decoder.v) 22
10.4 Register File (RegFile.v 22
10.5 Multiplexer for datapath (mux8.v) 23
10.6 Function Unit (FunctionUnit.v) 23
10.7 Datapath (datapath.v) 25
10.8 Conditional Multiplexer (muxCond.v) 25
10.9 Program Counter (progCounter.v) 26
10.10Instruction Register (instructionReg.v) 26
10.11Memory (memory.v) 27
10.12Main (main.v) 27

11 Appendix B - Code for Part II **28**

1 Abstract

Since its "birth" in 1971, embedded microprocessor has been widely used as a tool for technological innovations and cost reduction. For a design to be competitive, its processor has to fit the following characteristics: relatively inexpensive, flexible, adaptable, fast, and reconfigurable. A solution to this is the use of Field-programmable gate arrays (FPGA) as design tool. Part I of this project describes the realization of an 8-bit FPGA based simple microprocessor. The system was implemented on the Xilinx Spartan 3e Starter Kit using ISE 10.1 and Verilog. Part II of this project uses the Basys3 board to interface the Analog-to-Digital Converter to process a real analog signal. The digitized signal is further processed to check if it is a deterministic signal or random noise. If it is perceived as a deterministic signal, an alarm is raised. This is done on Xilinx Vivado Design Suite.

Keywords: Microprocessor, Field Programmable Gate Array (FPGA), Analog to Digital Converter, Xilinx Spartan 3E FPGA, Basys3 Artix-7 FPGA, Signal Processing

2 Introduction

Embedded microprocessors, today, are used in a variety of electronic products such as mobiles, computers and everyday household appliances like washing machines and microwaves. They consist of thousands of electronic components and perform various ALU and memory operations through a collection of machine instructions.

The Central Processing Unit is the core of the computer. It determines the speed and the capabilities of the computer. In Part I of this project, I have implemented an FPGA-based 8-bit microprocessor which does basic 8085-level ALU and memory operations. It is designed in Verilog and mainly consists of 8 8-bit registers in a Register File, 256-word memory with 16-bit words, a Control Unit and a Function Unit implementing ALU operations.

In the second part, a custom microprocessor is realized. An Analog-to-Digital Converter is of prime importance whenever real-world signals come into play. The XADC (Analog-to-Digital Converter) is interfaced to convert an incoming real-world analog signal to digital form. This data is validated and further checked to determine if it is noise or contains a signal.

3 Relevance

The 8085 is regarded as the father of all CPU designs and all later microprocessors were an enhancement of it. Thus, implementing a simple 8-bit microprocessor would give me a good idea about basic computer architecture. Part II of this project has great applications in the monitoring of illicit activities in forests. Forests in Gujarat and Maharashtra are particularly troubled by the increase of deforestation and poaching of endangered wildlife. For that, an ASIC based on my FPGA-prototype of an Audio (Analog Signal) Monitoring System will help in surveillance. The chip will detect any out-of-the-ordinary sounds in the surroundings and communicate a warning to the base station. Unlike CCTVs, this IC will be powered by a battery and will consume minimal power because of its simple functionality.

4 Part I - Microprocessor Basics

Microprocessors are the devices in a computer that make things happen. They are capable of performing basic arithmetic operations, moving data from place to place and making decisions. An 8-bit microprocessor can read/write 1 byte at a time and can directly access 256 bytes. In this part, I implement a general purpose microprocessor, which are typically the kind of CPUs found in desktop computer systems. The CPU design is reduced instruction set computing (RISC) and the clock rate is 50 MHz (on-board oscillator).

4.1 Instruction Set Architecture

The processor is designed to input 16-bit set of instructions. This instruction is simply made equivalent to a control word by omitting branch control in the design. The control word determines the operations required to be done as well as the memory addresses of the operands. The control word format is given as follows.

15 14 13 12	11	10	9	8 7 6	5 4 3	2 1 0
FS	MD	RW	MB	DA	AA	BA

Here AA, BA and DA are the A, B and Destination addresses i.e. one register out of R0 to R7. The control word further contains MB (Mux B select), FS

(Function select), MD (Mux D select) and RW (Register Write). The control word is held by the Instruction Register module and fed into the datapath.

4.2 Datapath

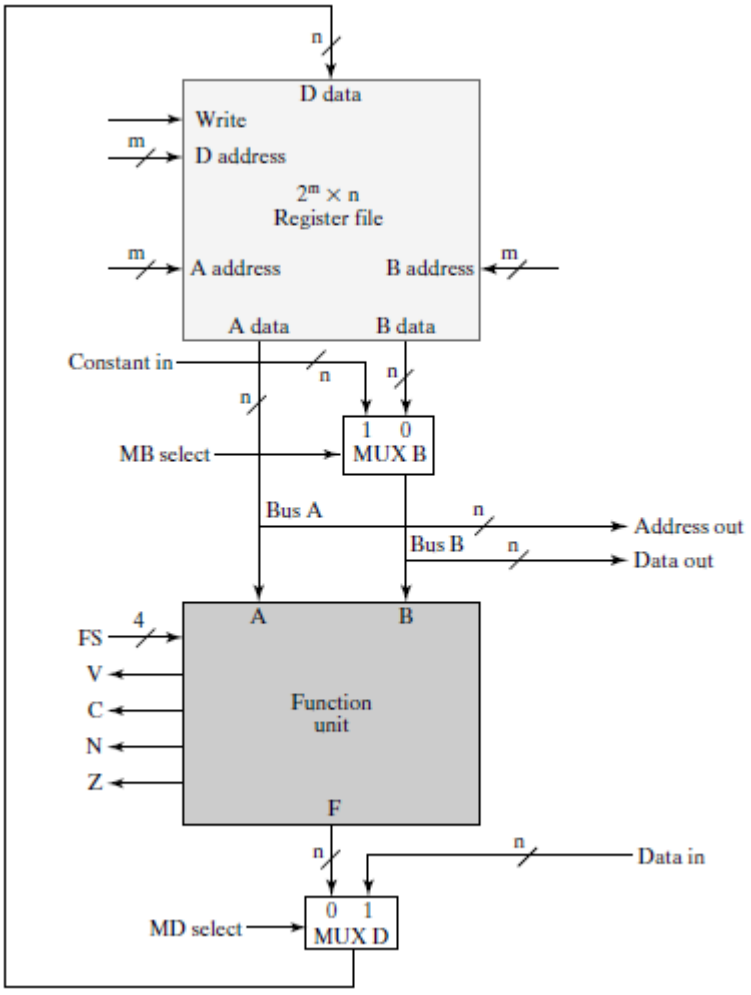

Figure 1: The Datapath

Datapath is a set of functional units that carry out data processing operations. As shown in figure 1, it is comprised of a register file, a function unit

and two multiplexers. The register file is an array of 8 registers with common data input lines (D Data) and two sets of output data lines (A and B data). A simpler 4-register Register File with n-bit data storage is shown in figure 2. The operands for and the result of ALU operations is stored in the

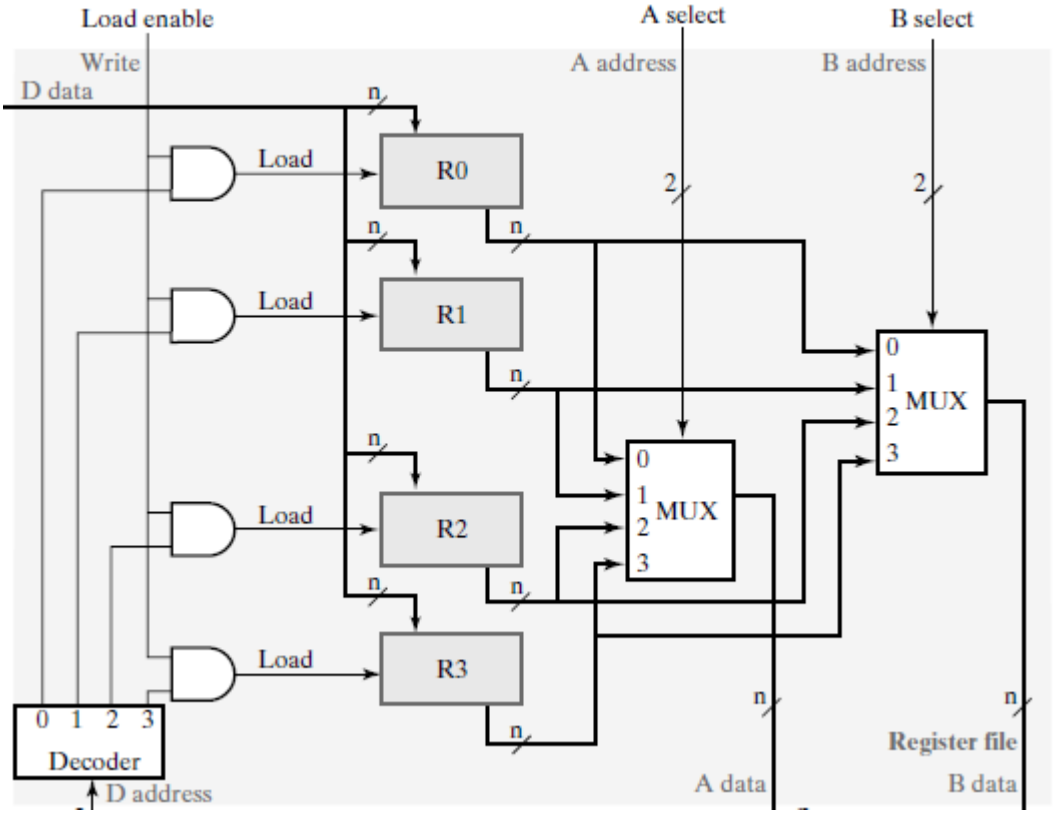

Figure 2: A simple 4*n register file

registers in the register file and could be re-used in a subsequent operation or saved into memory. The Function Unit is the Arithmetic and Logic Unit (ALU) which takes 2 inputs, A and B, and returns the result of the operation specified by the Function Select (FS) bits. The following table enlists all the operations of the Function Unit.

Function Select	Operation	Function
0000	F = A	Transfer A
0001	F = A + 1	Increment A
0010	F = A + B	Addition
0011	F = A + B + 1	Add with carry input of 1
0100	F = A + $\tilde{B}$	A plus 1s complement of B
0101	F = A + $\tilde{B}$ + 1	Subtraction
0110	F = A - 1	Decrement A
0111	F = A	Transfer A
1000	F = A AND B	AND
1001	F = A OR B	OR
1010	F = A XOR B	XOR
1011	F = $\tilde{A}$	NOT
1100	F = B	Transfer B
1101	F = B >>1	Right Shift by 1
111X	F = B <<1	Left shift by 1

Table 1: Function Select bits

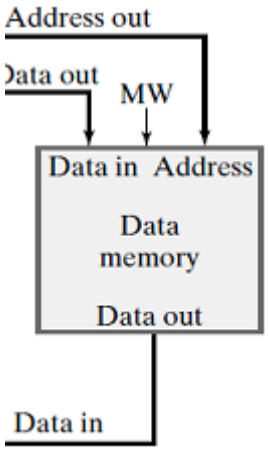

Figure 3: Memory

4.3 Memory

The memory has 256 words and each word is comprised of 16 bits. An 8-bit address specifies the memory location which is being written to or read from.

4.4 Bits of the Control Word

The function of various bits in the Control Word are as follows.

AA, BA, DA	Register select
000	Select R0
001	Select R1
010	Select R2
011	Select R3
100	Select R4
101	Select R5
110	Select R7
111	Select R7

Table 2: AA, BA and DA bits

MD	Operation
0	Bus D is fed with the ALU output
1	Bus D is fed with an external data input

Table 3: Mux D select bit

MB	Operation
0	BDATA from the register file is fed into the ALU
1	An external data input is fed into the ALU

Table 4: Mux B select bit

RW	Operation
0	No writing to memory
1	Writing to memory

Table 5: RW bit

The operations of the Function Select (FS) bits have already been tabulated in Table 1.

5 Part 1 - Microprocessor Implementation and Results

The implementation was done in a modular fashion. The main components of the microprocessor are the Control Unit, datapath and the memory. The control unit simply comprises of the instruction register. We do not have to decode the instruction into a control word as the processor has been designed such that the instructions are control words themselves. The datapath contains many sub-blocks including a Function Unit (ALU), a Register File, multiplexers, decoder and logic gates. The coding was done, starting from the lowest level sub-blocks and gradually moving up in hierarchy. The working was verified after each step. The results are as follows and the codes are given in appendix A. The testbenches for the codes are given at

`home.iitk.ac.in/~pranavk/ugp3/tbs`

5.1 Multiplexer results

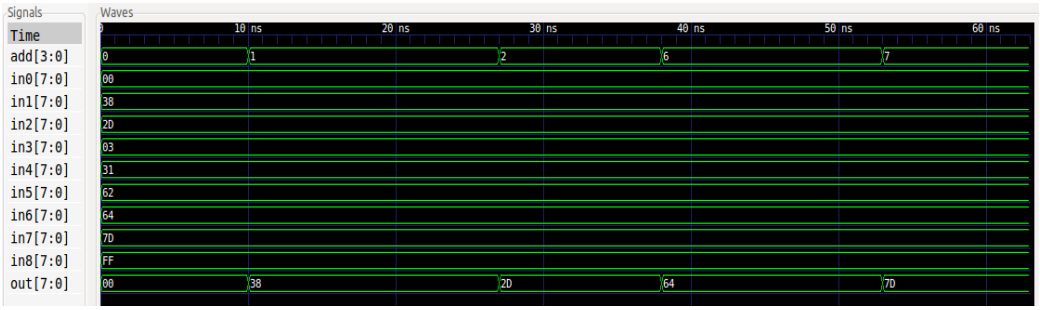

Figure 4: Multiplexer Results

In this, the registers in0 through in8 are having constant 8 bit values and depending upon the 3 bit address add, one of them is copied onto the output, out. For example, at t = 10 ns, in1 is copied, at 27 ns, in2 is copied, at 38 ns, in6 is copied and at 53 ns, in7 is copied. The output out verifies the correct working of the 8:1 multiplexer. The code is given in Section 10.1. The 2:1 multiplexer with one select bit (used in the datapath) and that with an enable input have similar working and the codes are given in Section 10.5 and 10.8.

5.2 Decoder results

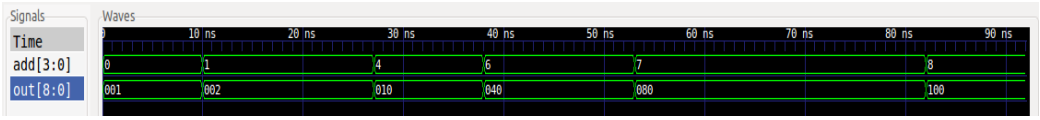

Figure 5: Decoder Results

The decoder outputs 1 at the corresponding bit position of out according to the decimal value of address add. For instance, at 53 ns, add is equal to decimal value 7, and hence out[7] = 1 and rest all bits would be 0. Thus, out = 010000000 = 080h. If add = 8, then out = 100000000 = 100h. The code is given in Section 10.3.

5.3 Register results

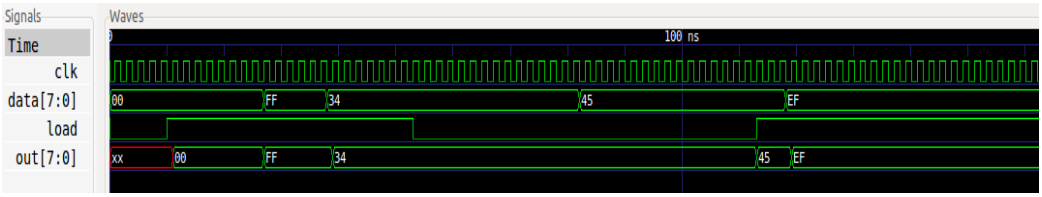

Figure 6: Register Results

Whenever the load is low, the register is in HOLD state, that is, it holds the last acquired value of data. When the load is high, the register samples the 8 bit data onto the output out. The working is exemplified in the graph attached. The code is given in section 10.2. The instruction register is a special purpose register with similar working and its code is given in Section 10.10.

5.4 Register File results

Register File takes as input 3-bit A, B, D register addresses, 8-bit DDATA and various multiplexer select bits and depending upon the inputs, either stores DDATA in one of the eight registers or any one or two registers sample their value onto outputs ADATA and BDATA. The code is given in Section 10.4.

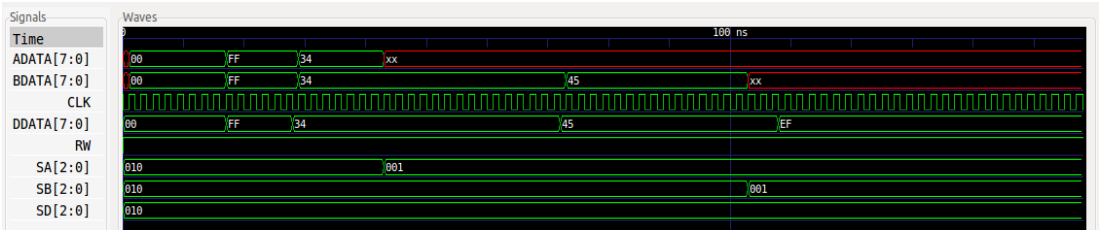

Figure 7: Register File Results

5.5 Memory results

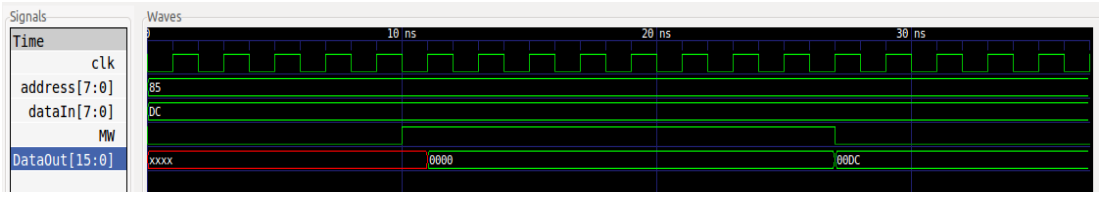

Figure 8: Memory Results

The Memory Write (MW) bit in the memory module dictates whether read or write should occur. At 10 ns, when the MW bit goes high, the dataIn (00DCh) is written to memory address 85h and the dataOut is 0 for whatever time a write is happening. At 27 ns, when MW switches to 0, a read operation starts to occur, and the data at memory location address[7:0] is sampled onto the output, dataOut. The code is given in section 10.11.

5.6 Microprocessor results

To illustrate the correct working of the Datapath including the ALU, the ABUS and BBUS are set to 85h and DCh respectively. Depending on the Function Select (FS) bits, FSOUT is calculated. For instance when FS = 0, FSOUT = A; when FS = 1, FSOUT = A+1; when FS = 4, FSOUT = A+$\tilde{B}$; when FS = 5, FSOUT = A-B; when FS = 8, FSOUT = A&B and when FS = 13, FSOUT = B>>1. The correct value of FSOUT exemplifies the correct working of the ALU and datapath. Some extra multiplexer select and other bits are not shown to reduce cluttering. The full codes for ALU and datapath are given in sections 10.6 and 10.7. The datapath, memory

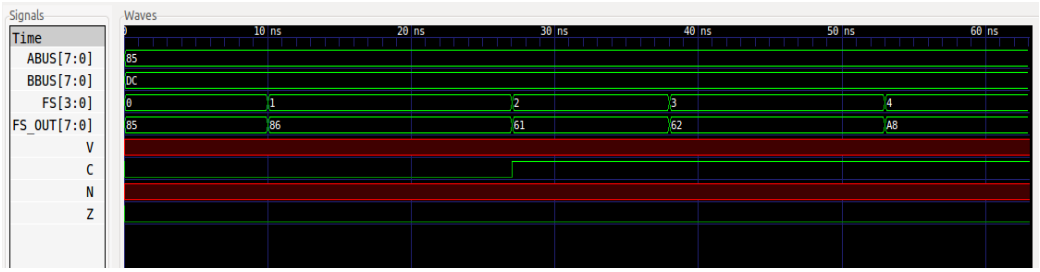

Figure 9: Microprocessor Results - I

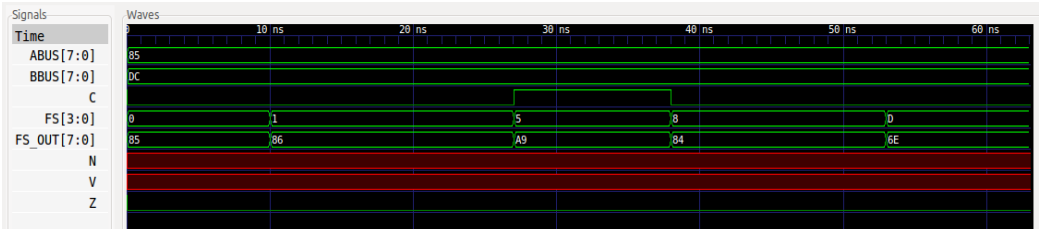

Figure 10: Microprocessor Results - II

and instruction register is integrated into a microprocessor module (main.v) in Section 10.12.

6 Part II - Signal Monitoring System Basics - XADC Functionality

This part of the project makes use of an XADC (Analog-to-Digital Converter) on the Basys3 FPGA Board to firstly digitize an incoming analog signal. Further, it is checked if the signal is deterministic or random noise. If it is deterministic, an alarm is raised for an indefinite time until it is reset manually. I start off by explaining the XADC functionality of the Xilinx 7 series FPGA boards (including the Basys3).

The XADC includes a dual channel 12-bit, 1 Mega sample per second (MSPS) ADC and on-chip sensors. Figure 11 shows a block diagram of the ADC. The dual ADCs support a range of operating modes and various analog input signal types. The ADCs can access up to 17 external analog input channels. However, to keep the system simplistic, I will be working with only 2 external inputs. One is the analog signal itself and the other is

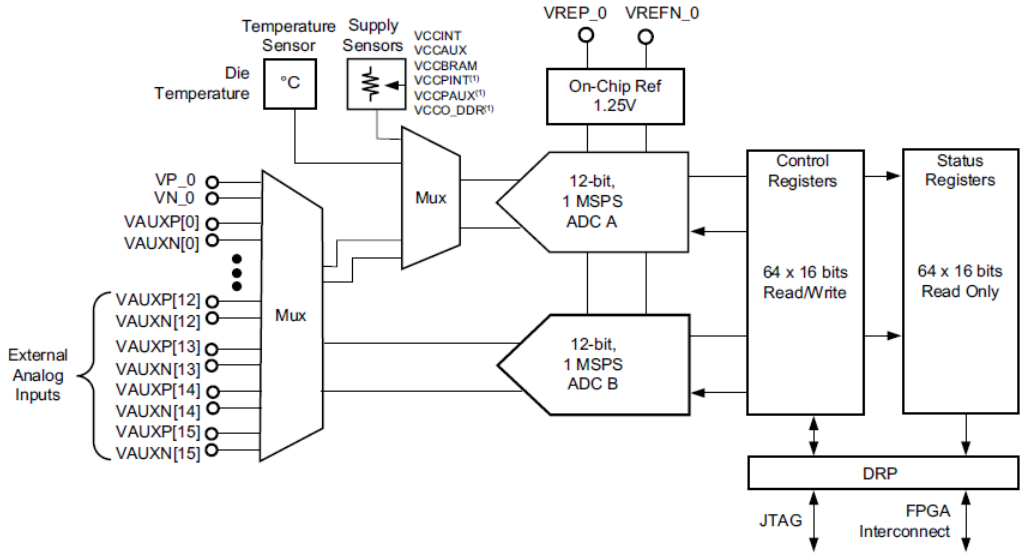

Figure 11: XADC Block Diagram

an external reset signal, which can be sent to reset the alarm.

6.1 XADC Operating Modes

The XADC includes several operating modes like default, Single Channel Mode and Simultaneous Sampling Mode. I have used the Single Channel Mode in which one must select the channel for analog-to-digital conversion by writing to bit locations CH4 to CH0 in control register 40h. Various configurations for single channel mode, such as analog input mode (BU), must also be set.

6.2 Input Signal Types

The external analog input channels can be configured to operate in unipolar or bipolar mode.

1. Unipolar Mode - The nominal analog input range to the ADC is 0V to 1V in this mode. The ADC produces a zero code (000h) when 0V is present on the ADC input and a full scale code of all 1s (FFFh) when 1V is present on the input.

2. Bipolar Mode - The nominal analog input range is -0.5V to 0.5V in this mode. The ADC produces a two's complement output code, that is, 000h for 0V, 7FFh for 0.5V and 800h for -0.5V. The transfer function is shown in figure 12.

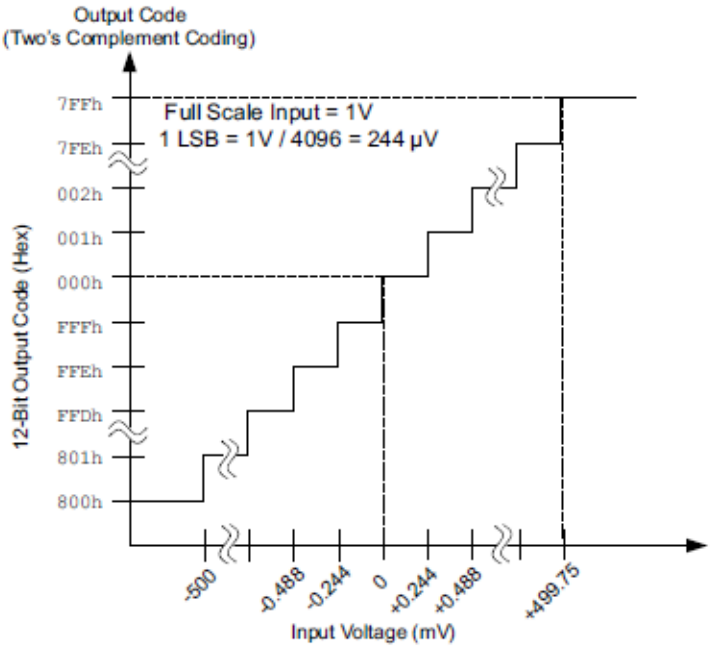

Figure 12: Bipolar mode transfer function

As I am dealing with real-world signals, I'll be operating in the bipolar mode.

6.3 Status Registers

The status registers contain the results of an analog-to-digital conversion of the on-chip sensors and external analog channels. All sensors and external analog-input channels have a unique channel address to which the measurement is stored. There are many status registers, however, I tabulate only those which I will be requiring.

Name	Address	Description
Temperature	00h	Measurement of the on-chip temperature sensor
VCCINT	01h	Power Supply Sensor measurement
VCCAUX	02h	Power Supply Sensor measurement
VCCBRAM	06h	Power Supply Sensor measurement
VP/VN	03h	Measurement of the dedicated analog input
VAUXP/N[15:0]	10h to 1Fh	Digitized external analog inputs

Table 6: Status Registers (Read-Only)

6.4 Control Registers

The XADC has 32 control registers that are located at addresses 40h to 5Fh. These registers are used to configure the XADC operation and the ones used are tabulated as follows.

Name	Address	Software Attribute	Description
Configuration Register 0	40h	INIT40	To configure XADC
Configuration Register 1	41h	INIT41	To configure XADC
Configuration Register 2	42h	INIT42	To configure XADC
Test Registers 0-4	43h-47h	INIT43-INIT47	Default value = 0000h

Table 7: XADC Control Registers

6.5 Instantiating the XADC

To allow access to the status registers, the XADC must be instantiated. Figure 13 shows the different ports of the XADC. The functionality of the used ports is tabulated in table 8. Instantiation of the XADC refers to the initialization of the control registers (Table 7). For sake of simplicity,

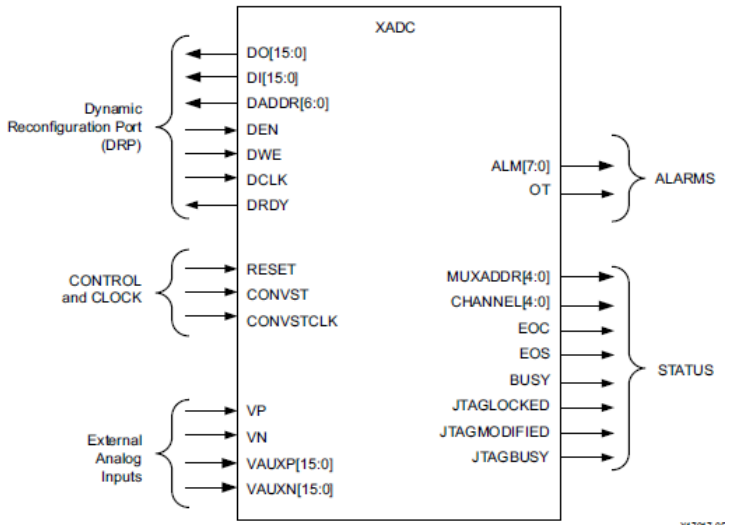

Figure 13: XADC Ports

Port	I/O	Description
DI[15:0]	Inputs	Input data bus
DO[15:0]	Outputs	Output data bus
DADDR[6:0]	Input	Address bus
DEN	Input	Enable signal
DWE	Input	Write enable signal
DCLK	Input	Clock input for the DRP = 50MHz
DRDY	Output	Data ready signal for the DRP
RESET	Input	Asynchronous ADC Reset signal
CONVST	Input	Convert Start Input
CONVSTCLK	Input	Convert Start Clock Input
VP, VN	Input	Dedicated Analog Input Pair
VAUXP/N[15:0]	Inputs	Sixteen external analog input pairs
EOC	Output	End of conversion signal
EOS	Output	End of sequence signal
Busy	Output	ADC Busy signal

Table 8: XADC Port Descriptions

7 Part II - Signal Monitoring System Realization

7.1 Implementation

The three configuration registers in the control register block configure the operating mode of the XADC. Their bit definitions are illustrated in figure 14. For sake of simplicity, the individual bit descriptions of the configuration registers have been omitted and can be found in the 7-series FPGAs XADC manual. The individual bits are set in each of the configuration registers and

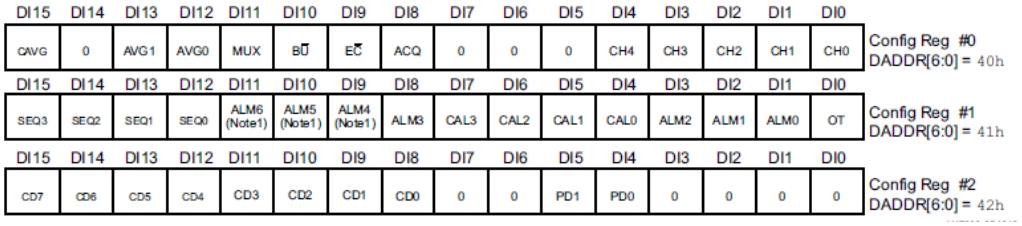

Figure 14: Configuration Registers

the XADC instantiated. The external input is averaged over 16 samples so as to reduce the error. On averaging over N samples, the standard deviation of the error reduces to one-$\sqrt{N}$th of the original. Once error is reduced, the averaged digitized value is checked with a threshold (0.1V). If it is above it, I claim it to be a deterministic signal and raise an indefinite alarm. If it was random noise, the averaged value would fall below the threshold, no alarm would be raised and the process would continue. The second analog signal is the external Reset signal. If it goes above a certain value (0.5V), the alarm is reset. The code was built upon an example in the Xilinx 7-Series XADC Manual and is given in Appendix B. The design file, through which input is simulated, as well as the obtained results follow in the next section.

7.2 Results

Inputs 1 and 2 (Tables 9 and 10) are the analog stimulus files being fed into the system and Figures 15 and 16 demonstrate the corresponding results. The ALARM bit is set whenever the average of the last 16 samples of MEASUREDAUX0 goes above 0.1V and is reset whenever the average of the last

16 samples of MEASUREDAUX1 goes above 0.5V. The averaging is evident in Result 2 as the measured value doesn't change every 10000 ns as in the analog stimulus file, rather it changes every 16 samples to a value equal to the average of the previous 16 samples.

TIME	VAUXP[0]	VAUXN[0]	VAUXP[1]	VAUXN[1]
00000	0.000	0.0	0.1	0.0
50000	0.2	0.0	0.0	0.0
90000	0.5	0.0	0.0	0.0
121000	0.0	0.0	0.6	0.0

Table 9: Input 1

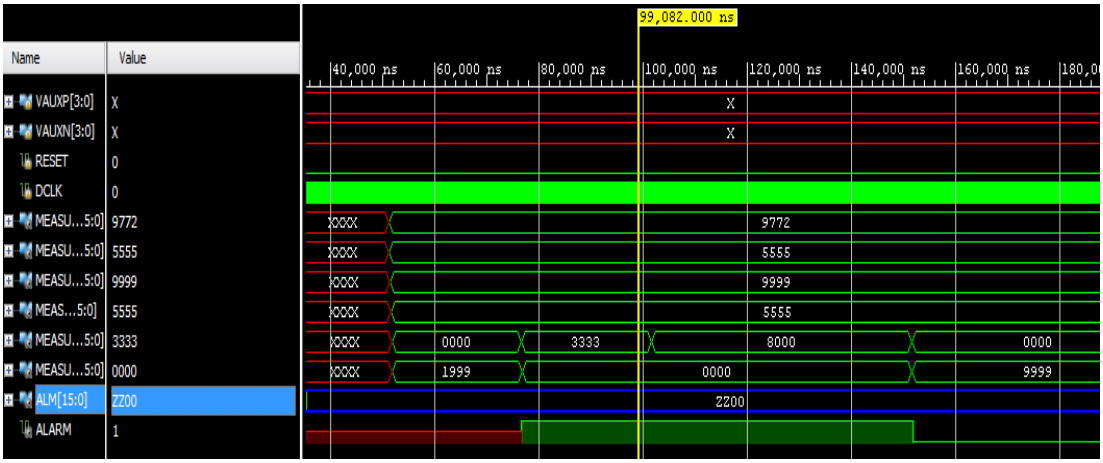

Figure 15: Results 1

TIME	VAUXP[0]	VAUXN[0]	VAUXP[1]	VAUXN[1]
00000	0.000	0.0	0.1	0.0
10000	0.1	0	0.15	0
20000	0.15	0	0.1	0
30000	0.13	0	0.65	0
40000	0.08	0	0.15	0
50000	0.2	0.0	0.0	0.0
60000	0.3	0	0.15	0
65000	0.35	0	0.15	0
70000	0.1	0	0.15	0
80000	0	0	0.15	0
85000	0.05	0	0.65	0
90000	0.5	0.0	0.0	0.0
100000	0.0	0	0.15	0
110000	0.5	0	0.15	0
121000	0.0	0.0	0.6	0.0

Table 10: Input 2

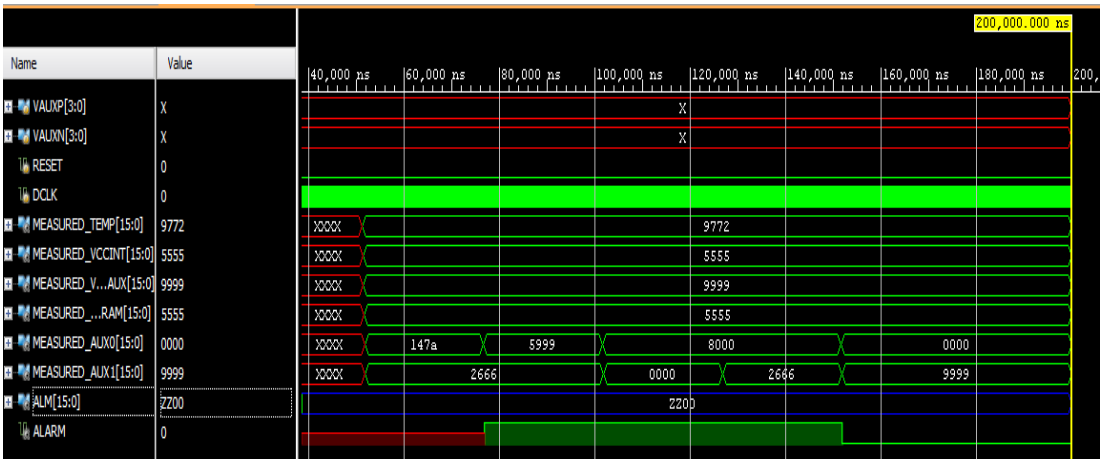

Figure 16: Results 2

8 Future Work

The part II of the project can be extended and built as a System-on-chip for forest surveillance purposes. For forest monitoring, devices have to be power-efficient and portable. The proposed ASIC would be battery-powered. It would be light-weight, portable and would consume minimal power due to its simplistic design and hence would last long. It would efficiently determine any out-of-the-ordinary sounds (analog signals) in the forest and any anomalous situation would be immediately reported to the base station via an alarm. Once the situation has been checked on and resolved, the alarm may be stopped by an external electromagnetic reset signal to the ASIC.

9 References

[1] Xilinx. 7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide. http://www.xilinx.com. UG480 (v1.9) September 27, 2016

[2] Y.S.Chauhan. Memory Design and Register Transfer, Micro-operations.[PDF Document] Digital Electronics(EE370). IIT Kanpur.2015.

[3] E. Ayeh, Y. Morita, K. Agbedanu, O. B. Adamo, and P. Guturu. FPGA Implementation of an 8-bit Simple Processor. In Proceedings of the IEEE Region 5 Technical, Professional and Student Conference (TPS), pp. 1-5, 2008.

[4] Xilinx. Spartan-3E FPGA Starter Kit Board User Guide. UG230 (v1.2). http://www.xilinx.com. January 20, 2011

[5] Digilent. Basys 3^{TM} FPGA Board Reference Manual. Revised April 8, 2016.

[6] M. Morris Mano, Charles R. Kime. Logic and Computer Design Fundamentals. Pearson. 2004. ISBN-13: 9780131405394

10 Appendix A - Code for part I

10.1 Multiplexer (mux.v)

```
always @ (add or in0 or in1 or in2 or in3 or in4 or in5 or
in6 or in7 or in8) begin
if (add == 4'b0000)
out = in0 ;
else if (add == 4'b0001)
out = in1;
else if (add == 4'b0010)
out = in2;
else if (add == 4'b0011)
out = in3;
else if (add == 4'd4)
out = in4;
else if (add == 4'd5)
out = in5;
else if (add == 4'd6)
out = in6;
else if (add == 4'd7)
out = in7;
else
out = in8;
end
```

10.2 Register (register.v)

```
module register(
    input [7:0] data,
    input load,
    output reg [7:0] out,
    input clk
    );
always @ (posedge clk)
if (load)
out = data ;
endmodule
```

10.3 Decoder (decoder.v)

```
module decoder(add, out);
input [3:0] add ;
output wire [8:0] out;
assign out[0] = (add==4'd0);
assign out[1] = (add==4'd1);
assign out[2] = (add==4'd2);
assign out[3] = (add==4'd3);
assign out[4] = (add==4'd4);
assign out[5] = (add==4'd5);
assign out[6] = (add==4'd6);
assign out[7] = (add==4'd7);
assign out[8] = (add==4'd8);
endmodule
```

10.4 Register File (RegFile.v

```
module RegFile(input [2:0] SA, input [2:0] SB, input [2:0] SD, input TD,
input TA, input TB, input RW, input [7:0] DDATA, input CLK, output [7:0] ADATA,
    output [7:0] BDATA);
wire [8:0] DEC_OUT;
wire [7:0] R0_OUT;
wire [7:0] R1_OUT;
wire [7:0] R2_OUT;
wire [7:0] R3_OUT;
wire [7:0] R4_OUT;
wire [7:0] R5_OUT;
wire [7:0] R6_OUT;
wire [7:0] R7_OUT;
wire [7:0] R8_OUT;
wire [3:0] AA, BA, DA;
assign AA[3] = TA;
assign BA[3] = TB;
assign DA[3] = TD;
assign AA[2:0] = SA;
assign BA[2:0] = SB;
assign DA[2:0] = SD;
```

```
decoder DEC(DA,DEC_OUT);
register R0(DDATA, DEC_OUT[0]&RW, R0_OUT, CLK);
register R1(DDATA, DEC_OUT[1]&RW, R1_OUT, CLK);
register R2(DDATA, DEC_OUT[2]&RW, R2_OUT, CLK);
register R3(DDATA, DEC_OUT[3]&RW, R3_OUT, CLK);
register R4(DDATA, DEC_OUT[4]&RW, R4_OUT, CLK);
register R5(DDATA, DEC_OUT[5]&RW, R5_OUT, CLK);
register R6(DDATA, DEC_OUT[6]&RW, R6_OUT, CLK);
register R7(DDATA, DEC_OUT[7]&RW, R7_OUT, CLK);

//Temporary Register for storage, invisible to the user
register R8(DDATA, DEC_OUT[8]&RW, R8_OUT, CLK);
mux MUX_A(AA, R0_OUT, R1_OUT, R2_OUT, R3_OUT, R4_OUT, R5_OUT, R6_OUT, R7_OUT, R8_O
mux MUX_B(BA, R0_OUT, R1_OUT, R2_OUT, R3_OUT, R4_OUT, R5_OUT, R6_OUT, R7_OUT, R8_O
endmodule
```

10.5 Multiplexer for datapath (mux8.v)

```
module mux_8(
   input sel, input [7:0] in0, input [7:0] in1, output reg  [7:0] out);
always @ (sel or in0 or in1) begin
  if (sel==0)
   out = in0 ;
  else
   out = in1 ;
end
endmodule
```

10.6 Function Unit (FunctionUnit.v)

```
module FunctionUnit(
    input [7:0] ABUS, input [7:0] BBUS, input [3:0] FS, output reg V,
    output  C, output reg N, output reg Z, output [7:0] FS_OUT);
reg [8:0] result = 0;
assign FS_OUT[7:0] = result[7:0];
assign C = result[8];
always @ (ABUS or BBUS or FS) begin
if (FS == 2'b00)
```

```
result = ABUS ;
else if (FS == 1)
result = ABUS + 1;
else if (FS == 2)
result = ABUS + BBUS ;
else if (FS == 3)
result = ABUS + BBUS + 1 ;
else if (FS == 4)
result = ABUS + ~(BBUS) ;
else if (FS == 5)
result = ABUS + ~(BBUS) + 1 ;
else if (FS == 6)
result = ABUS - 1 ;
else if (FS == 7)
result = ABUS ;
else if (FS == 8)
result = ABUS & BBUS ;
else if (FS == 9)
result = ABUS | BBUS ;
else if (FS == 10)
result = ABUS ^ BBUS ;
else if (FS == 11)
result = ~(ABUS);
else if (FS == 12)
result = BBUS ;
else if (FS == 13)
result = BBUS >> 1 ;
else
result = BBUS << 1;
end
always @(result) begin
if (FS_OUT==0)
Z = 1;// zero flag
else
Z = 0;
end
endmodule
```

10.7 Datapath (datapath.v)

```
module datapath(DA,AA, BA, TD, TA, TB, RW, MB, MD, FS, const_in, data_in, clk, V,
input TD, TA, TB, MB, MD, RW, clk;
   input [2:0] DA, AA, BA;
input [3:0] FS;
   input [7:0] const_in, data_in;
output V, C, N, Z;
   output [7:0] A_data, B_data;
  wire [7:0] D_in, A_data, B_data, muxB_out, FU_out, data_write;

  RegFile reg_file(AA, BA, DA, TA, TB, TD, RW, data_write, clk, A_data, B_data);
  FunctionUnit FU(A_data, muxB_out, FS, V, C, N, Z, FU_out);
  mux_8 MuxB(MB, B_data, const_in, muxB_out);
  mux_8 MuxD(MD, FU_out, data_in, data_write);
endmodule
```

10.8 Conditional Multiplexer (muxCond.v)

```
module mux_cond(sel, in2, in3, in4, in5, in6, in7, out);
 input in2, in3, in4, in5, in6, in7;
 input [7:0] sel;
 output reg out;
 always @(sel or in2 or in3 or in4 or in5 or in6 or in7)
 begin
case(sel)
8'd0: out <= 0;
8'd1: out <= 1;
8'd2: out <= in2;
8'd3: out <= in3;
8'd4: out <= in4;
8'd5: out <= in5;
8'd6: out <= in6;
8'd7: out <= in7;
endcase
end
endmodule
```

10.9 Program Counter (progCounter.v)

```
module program_counter(add_out, AD, PI, PL, clk);
  input PI, PL, clk;
  input[5:0] AD;
  output[7:0] add_out;
  reg [7:0] offset;
  reg [7:0] add_out = 0;
  always @(AD)
    begin
   if (AD[5] == 1)
        begin
        offset <= 8'b11000000 + AD[5:0];
     end
   else
     begin
       offset <= 8'b00000000 + AD[5:0];
    end
    end
  always @(posedge clk)
    begin
      if ((PI == 1'b1) & (PL == 1'b0))
        begin
          add_out <= add_out + 1;
        end
      else if ((PL == 1'b1) & (PI == 1'b0))
        begin
           add_out <= (add_out + offset);
        end
    end
endmodule
```

10.10 Instruction Register (instructionReg.v)

```
module instruction_reg(ins, load, clk, out
    );
 input [15:0] ins;
 input load, clk;
```

```
 output reg [15:0] out;

 always @(posedge clk) begin
 if (load)
out = ins;
end
endmodule
```

10.11 Memory (memory.v)

```
module memory(MW, address, DataIn, DataOut);
input MW;
input [7:0] address;
input [7:0] DataIn;
output reg[15:0] DataOut;
reg [15:0]mem[0:255];
always begin
if(MW==1) begin
mem[address] = 16'b0 + DataIn[7:0];
DataOut <= 0;
end
else
DataOut <= mem[address];
end

endmodule
```

10.12 Main (main.v)

```
//Declaration and Initialization of variables
 assign AD[5:3] = DR;
 assign AD[2:0] = SB;
 assign opt_add = 8'b0 + inst[15:9];
 assign DR = inst[8:6]; // Register Address assignment
 assign SB = inst[2:0];
 assign SA = inst[5:3];

 mux_8 muxM(MM, B_data, PC_out, add_out);  // memory mux
```

```
 mux_8 mux0(M0, NA, opt_add, BA); // operation mux
 mux_cond MuxC(MC, V, C, N, Z, ~C, ~Z, cond); // condition mux
 instruction_reg IR(data_in, IL, clk, inst); // Instruction Register
 program_counter PC(PC_out, AD, PI, PL, clk); // Program Counter
 datapath DP(DR,SA, SB, TD, TA, TB, RW, MB, MD, FS, const_in, data_in[7:0], clk,
 V, C, N, Z, data_out, B_data);
endmodule
```

11 Appendix B - Code for Part II

```
//start of module
//declaration and initialization of variables

parameter init_read = 8'h00,
                    read_waitdrdy   = 8'h01,
                    write_waitdrdy  = 8'h03,
                    read_reg00      = 8'h04,
                    reg00_waitdrdy  = 8'h05,
                    read_reg01      = 8'h06,
                    reg01_waitdrdy  = 8'h07,
                    read_reg02      = 8'h08,
                    reg02_waitdrdy  = 8'h09,
                    read_reg06      = 8'h0a,
                    reg06_waitdrdy  = 8'h0b,
                    read_reg10      = 8'h0c,
                    reg10_waitdrdy  = 8'h0d,
                    read_reg11      = 8'h0e,
                    reg11_waitdrdy  = 8'h0f,
                    read_reg12      = 8'h10,
                    reg12_waitdrdy  = 8'h11,
                    read_reg13      = 8'h12,
                    reg13_waitdrdy  = 8'h13;

BUFG i_bufg (.I(DCLK), .O(dclk_bufg));
   always @(posedge dclk_bufg) begin
   if (MEASURED_AUX0 > 16'h2222) begin
                ALARM <= 1;
```

```
          end
          else if (MEASURED_AUX1 > 16'h8000) begin
            ALARM <= 0;
          end


if (RESET) begin
   state   <= init_read;
   den_reg <= 2'h0;
   dwe_reg <= 2'h0;
   di_drp  <= 16'h0000;
   ALARM <= 0;
end

else
   case (state)
   init_read : begin
      daddr <= 7'h40;
      den_reg <= 2'h2; // performing read
      if (busy == 0 ) state <= read_waitdrdy;
      end
   read_waitdrdy :
      if (eos ==1)   begin
         di_drp <= do_drp  ; //add & 16'h03_FF for Clearing AVG bits for Cor
         daddr <= 7'h40;
         den_reg <= 2'h2;
         dwe_reg <= 2'h2; // performing write
         state <= write_waitdrdy;
      end
      else begin
         den_reg <= { 1'b0, den_reg[1] } ;
         dwe_reg <= { 1'b0, dwe_reg[1] } ;
         state <= state;
      end
   write_waitdrdy :
      if (drdy ==1) begin
         state <= read_reg00;
         end
```

```
   else begin
      den_reg <= { 1'b0, den_reg[1] } ;
      dwe_reg <= { 1'b0, dwe_reg[1] } ;
      state <= state;
   end
read_reg00 : begin
   daddr   <= 7'h00;
   den_reg <= 2'h2; // performing read
   if (eos == 1) state   <=reg00_waitdrdy;
   end
reg00_waitdrdy :
   if (drdy ==1)   begin
      MEASURED_TEMP <= do_drp;
      state <=read_reg01;
      end
   else begin
      den_reg <= { 1'b0, den_reg[1] } ;
      dwe_reg <= { 1'b0, dwe_reg[1] } ;
      state <= state;
   end
read_reg01 : begin
   daddr   <= 7'h01;
   den_reg <= 2'h2; // performing read
   state   <=reg01_waitdrdy;
   end
   reg01_waitdrdy :
  if (drdy ==1)   begin
      MEASURED_VCCINT = do_drp;
      state <=read_reg02;
      end
   else begin
      den_reg <= { 1'b0, den_reg[1] } ;
      dwe_reg <= { 1'b0, dwe_reg[1] } ;
      state <= state;
   end
read_reg02 : begin
   daddr   <= 7'h02;
   den_reg <= 2'h2; // performing read
```

```
   state   <=reg02_waitdrdy;
   end
reg02_waitdrdy :
   if (drdy ==1)   begin
      MEASURED_VCCAUX <= do_drp;
      state <=read_reg06;
      end
   else begin
      den_reg <= { 1'b0, den_reg[1] } ;
      dwe_reg <= { 1'b0, dwe_reg[1] } ;
      state <= state;
   end
read_reg06 : begin
   daddr   <= 7'h06;
   den_reg <= 2'h2; // performing read
   state   <=reg06_waitdrdy;
   end
reg06_waitdrdy :
   if (drdy ==1)   begin
      MEASURED_VCCBRAM <= do_drp;
      state <= read_reg10;
   end
   else begin
      den_reg <= { 1'b0, den_reg[1] } ;
      dwe_reg <= { 1'b0, dwe_reg[1] } ;
      state <= state;
   end
read_reg10 : begin
      daddr   <= 7'h10;
      den_reg <= 2'h2; // performing read
      state   <= reg10_waitdrdy;
   end
reg10_waitdrdy :
   if (drdy ==1)   begin
      MEASURED_AUX0 <= do_drp;
      state <= read_reg11;
   end
   else begin
```

```
      den_reg <= { 1'b0, den_reg[1] } ;
      dwe_reg <= { 1'b0, dwe_reg[1] } ;
      state <= state;
   end
read_reg11 : begin
   daddr   <= 7'h11;
   den_reg <= 2'h2; // performing read
   state   <= reg11_waitdrdy;
   end
reg11_waitdrdy :
   if (drdy ==1)   begin
      MEASURED_AUX1 <= do_drp;
      state <= read_reg12;
      end
   else begin
      den_reg <= { 1'b0, den_reg[1] } ;
      dwe_reg <= { 1'b0, dwe_reg[1] } ;
      state <= state;
   end
read_reg12 : begin
   daddr   <= 7'h12;
   den_reg <= 2'h2; // performing read
   state   <= reg12_waitdrdy;
   end
reg12_waitdrdy :
   if (drdy ==1)   begin
      MEASURED_AUX2 <= do_drp;
      state <= read_reg13;
      end
   else begin
      den_reg <= { 1'b0, den_reg[1] } ;
      dwe_reg <= { 1'b0, dwe_reg[1] } ;
      state <= state;
   end
read_reg13 : begin
   daddr   <= 7'h13;
   den_reg <= 2'h2; // performing read
   state   <= reg13_waitdrdy;
```

```
            end
         reg13_waitdrdy :
            if (drdy ==1)   begin
               MEASURED_AUX3 <= do_drp;
               state <=read_reg00;
               daddr   <= 7'h00;
            end
            else begin
               den_reg <= { 1'b0, den_reg[1] } ;
               dwe_reg <= { 1'b0, dwe_reg[1] } ;
               state <= state;
            end
         default : begin
            daddr <= 7'h40;
            den_reg <= 2'h2; // performing read
            state <= init_read;
            end
         endcase
         end

//instantiation of the XADC
//End of module
```