UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

FPGA Configuration to Simulate a Simple Accumulator Based Processor on an Educational Board

Dimitris Tsaliagos

1/15/2007

A simple accumulator-based processor has been implemented. It was destined for an FPGA-based experimental platform for educational purposes, provided by FORTH. The processor is implemented using components provided from the vendor of the targeted FPGA device, and digital logic described using the verilog hardware description language in order to use the board interface. Configuration of the FPGA device involved not only the processor design but also the design of new components that would use the board interface to work along with the processor. Such designs was the seven-segment controller which was used to control the board displays, the de-bouncer for the push-buttons, the edge detector for the processor clock and a linear feedback register. Processor main blocks are the instruction and data memories that are implemented separately, the Arithmetic Logic unit (ALU) with the accumulator of the processor and the processor bus. In order to verify the processor functionality a control circuit was implemented. The top-level design that was downloaded to the FPGA device consists of the processor block and all the appropriate components that control the board interface.

ACKNOWLEDGMENTS

First of all I would like to thank my advisor Prof. Manolis Katevenis as also Prof. George Kalokairinos for their guidance and their support. Moreover, I would like to thank Mixalis Ligerakis, co-worker at FORTH and designer of the system board, for his help and his guidance throughout this work. Finally, I would like to thank my family (Mixalis, Zoi and Marialena) and my close friend Gewrgia for their invaluable help they have offered me all this time. Thanks are also to all the co-workers at FORTH who provided a very friendly environment to work with and helped me with their knowledge.

CONTENTS

TABLE OF FIGURES

1. BACKGROUND

Academia is in need of design platforms which allow a student to explore the design space by implementing theoretical concepts learned in class. Often the theory of processor design is rigidly dictated to the students because of the lack of time and resources available. An FPGA-based educational board was developed in FORTH that would be used in a digital design course. The board layout can be seen in Figure 1. It consists of an FPGA device connected to a number of seven-segment display's ,LED[1], push-buttons and a number of IC's to interface the components mentioned before with the FPGA device. Configuration of the FPGA device can be done via the JTAG[2] or the ByteBlaster[3] connector. The appropriate circuit that allows the configuration of the FPGA are integrated on the board.

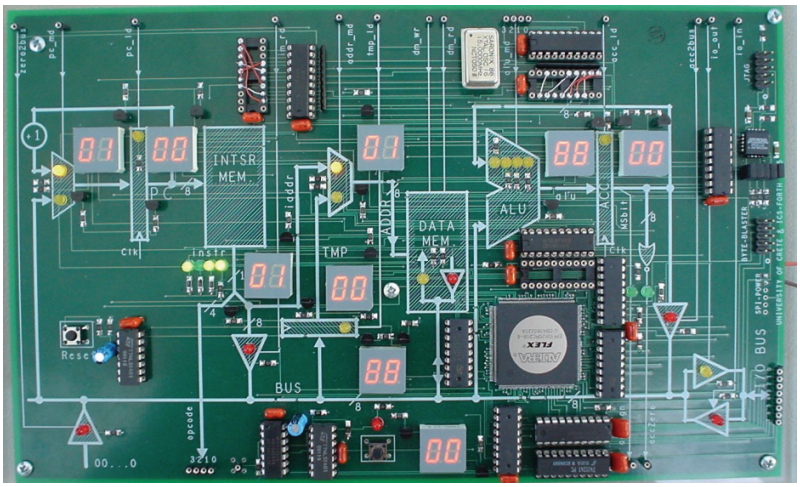

Figure 1 Board Layout

1.1. CONFIGURATION OF THE BOARD

In order to program the FPGA device a programmer tool must be used to download te compiled design to the FPGA. The programmer tool uses the JTAG or the ByteBlaster connector of the board to configure the FPGA device. In the current board, a number of jumpers must be shorted to select among the two configuration modes; further information can be found in (Kalokarinos, Ligerakis, Tsaliagos 2007).

---

[1] **Light-emitting diode** (**LED**) is a semiconductor device that emits light when voltage is applied.

[2] **Joint Test Action Group (JTAG)** is the usual name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture is used for testing sub-blocks of integrated circuits, and is useful as a mechanism for debugging embedded systems.

[3] **ByteBlaster™** is a download cable that allows you to program and configure Altera® devices. This cable drives configuration data from a standard parallel printer port on your PC to the device on the printed circuit board (PCB).

1.2. SCHEMATICS

Interfacing the FPGA device with the various components such as the seven-segment displays, the push buttons, the input signals, the reset signal and the processor clock which require some circuits in order to connect to the FPGA device. Schematics of the circuits mentioned before and of the board can be found in (Kalokarinos, Ligerakis, Tsaliagos 2007).

1.3. FIELD PROGRAMMING GATE ARRAY

An FPGA is a programmable logic device (PLD) that can be reprogrammed any number of times after it has been manufactured. Internally FPGAs contain gate arrays of pre-manufactured programmable logic elements called cells. A single cell can implement a network of several logic gates that are fed into flip-flops. The re-programmable nature of FPGAs makes them ideal for educational purposes because it allows the students to attempt as many iterations as necessary to correct and optimize their processor design. The FPGA that is used, is from Altera FLEX10K family. This FPGA device family has 20,000 typical gates and 1,152 logic elements.

2. INTRODUCTION

The system that has been build consists of three main parts and two sub-modules. These are the processor, display control unit and the input system. Instruction Read Only Memory (ROM), Data Random Access Memory (RAM) are the sub-modules which are instantiated inside the processor block. The processor is accumulator based, this means that it uses an accumulator to store arithmetic and logic results. The use of the accumulator is to calculate the results without storing the calculated data each time to the main memory ; the cost of writing data to the main memory for every instruction is much slower than writing into the accumulator. The processor implemented for the system is 8-bit and the instruction set of the processor it is constituted from 16 instructions. Further information for the instruction set of the processor will follow. During the processor design a control circuit was built in order to verify the processor functionality. Control circuit is responsible to create the appropriate signals for the execution of the instructions. In order to interface the above processor with the board, a number of components had been used. The display control unit as an example is controlling the seven-segment displays of the board to display various signals and registers of the processor and the LED's of the board. The instruction and data memories are used from the processor to store data and read instructions. The Input system is used to help the user to interact with the processor. It is constituted from de-bouncers, edge-detectors and glue logic to select the appropriate inputs among the different inputs that the board offers. Finally the top-level design of the system consists of all these modules in order to use all the board components effective. Verification of the system was done using the control circuit mentioned before implemented in an FPGA development board and connecting it to the FPGA-based educational board. An example test code that had been used for the processor verification is described later.

3. PROCESSOR DESCRIPTION

The processor designed for this system is accumulator-based thus it makes the implementation fairly simple. Examples of known accumulator-based processors are the PDP-8 and the Motorola 6809. Advantages of accumulator-based architectures are that they have smaller instruction length thus the code size of the program is smaller than Stack or General Purpose Register (GPR) architectures. Although code size decreases, the implementation efficiency and the compiler construction in such architectures is much more complex than the other ones. Of equal importance is the fact that the memory organization of the processor had been done according to the Harvard principles. In such architectures, we have two separate memories; one for the instructions and one for the data. These two memories differ in size, width and type. The block diagram in Figure 2 shows the main components that are used to implement the processor along with the two separate memories. It consists of multiplexers, registers, adders, combinational logic and memories.

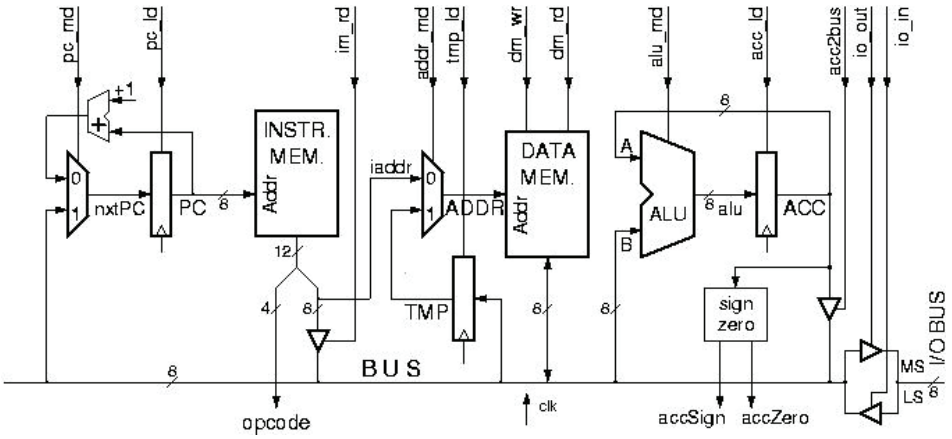

Figure 2 Processor Block Diagram

3.1. INSTRUCTION SET

The Instruction Set Architecture (ISA) of this processor is very simple, but capable of executing a large variety of simple and basic programs. It contains 16 instructions that form a simplistic ISA that has the basic characteristics of a real computer processor. A short description for the operation of each instruction and their operands is shown in Table 3-1.

INSTRUCTION	OPERATION
ADDX ADDR	ACC← ACC + DM[DM[ADDR]]
ADD ADDR	ACC ← ACC + DM[ADDR]
SUB ADDR	ACC←ACC - DM[ADDR]
AND ADDR	ACC ←ACC AND DM[ADDR]
INPUT ADDR	DM[ADDR] ← I/O BUS
NOR ADDR	ACC←NOT (ACC OR DM[ADDR])
LOAD ADDR	ACC ← DM[ADDR]
LOADX ADDR	ACC← DM[DM[ADDR]]
STORE ADDR	DM[ADDR] ← ACC
STOREX ADDR	DM[DM[ADDR] ←ADDR

JUMP ADDR	PC ←ADDR
JUMPX ADDR	PC-< DM[ADDR]
BEQ ADDR	PC< - PC +1
BNE ADDR	PC< - PC +1
BLE ADDR	PC< - PC +1
BGE ADDR	PC< - PC +1

Table 3-1 Instructions Operation

Except from the common instructions like add, sub etc, the specific processor has three instructions called indexed instructions that are a little bit different from the traditional load, store and jump instructions. Description of them will follow. As for the instruction format, it is fixed size with each instruction have 12-bit width. Only one instruction format is recognized from the processor due to the instruction set simplicity. The twelve bits are divided into two fields: the 4-bit operation code and the 8-bit operand field .

3.1.1. INSTRUCTION TYPES

The processor identifies three instruction categories. These are described below:

- ***Data Transfer***: These instructions cause data in one location (either the internal registers or external memory) to be copied to another location. In this category the ***LOAD, STORE, LOADX, STOREX and INPUT*** instructions are characterized as data transfer instructions.

- ***Data Operation :*** These include the arithmetic and bitwise operations. The ***ADD, AND, NOR, SUB, ADDX*** instructions can only operate on data.

- ***Program Control :*** These instructions change the sequence of program execution. They are often called branch instructions. These are the ***JMP, BEQ, BNE, BLT, BGT, JMPX*** instructions.

3.1.2. ADDRESSING MODES

Only small subsets of the common addressing modes are used by this processor. Briefly these are the Direct, Immediate and Indexed addressing modes. Detailed description of the modes is shown below.

- Direct: This is the same as absolute addressing. The address of the required data is part of the instruction. In this case, it will be the eight least-significant bits of the instruction. By way of example, an ADD instruction has as operand a memory location and the accumulator, such an instruction would take the data from the memory location which value is equal to the operand of the instruction.

- Immediate: The required data is part of the instruction. For this architecture, it is the eight least-significant bits of the instruction. An instruction that shows the above addressing mode is the JMP instruction which has as an operand an address,in this case the instruction uses the value of the operand to calculate the result and change the value of the program counter to PC + Immediate (the operand of the JMP instruction.

- Indexed : The location of the operand resides in the memory address pointed by the address field of the instruction. In this addressing mode, the operand is an index to the memory where the location of the data reside. A STOREX instruction for example, has as an operand an address, the data which it will operate on are located in the memory address MEMORY[storex operand].

3.2. PROCESSOR CONTROL LOGIC DESCRIPTION

The control logic for the specific processor is very simple because almost all the instructions require combinatorial logic in order to be executed except form the indexed instructions which require sequential logic to be executed. Execution time of the indexed instructions is 2 clock cycles, for all the other instructions is 1 clock cycle. Processor primary input signals are the alu_md, im_rd, pc_md, pc_ld, dm_rd, dm_wr, acc_ld, acc_md, acc2bus, ext2bus and addr_md. The pc_md and pc_ld signals are used for controlling the incrementation of the program counter. The signal that selects the appropriate input for the program counter is the pc_md. By driving pc_md active high, the contents of the bus is selected as the next program counter else if the signal is active low then the next program counter is the value of the previous one (program counter of the current clock cycle) plus one. To be able to capture the result of the previous calculation to the program counter register the pc_ld signal must be driven high in a processor positive clock edge. Signals that are used from the memory are the dm_rd which is the signal for a memory read and the dm_wr which is the signal for a memory write. It must be mentioned that these signals are active high, meaning that to be able to read an address from the data memory the dm_rd signal must be active high. The same functionality as the dm_rd signal has the dm_wr signal too. Memory address is selected by driving the addr_md signal. If addr_md is active high then the address at which will be read would be the data that exists in the tmp register, else the address is the eight least significant bit of the instruction. Data read from memory are passed to the Arithmetic Logic Unit (ALU), therefore in order to select an appropriate operation of the ALU on the data the alu_md signal selects the operation that it will be executed. Further details for the ALU modes can be found on Chapter 4.Capturing the results from the ALU to the accumulator register acc_ld signal must be driven high during a positive clock edge. As it concerns the bus control signals, these are the acc2bus, ext2bus and the im_rd which are driving the bus with an 8-bit data which is the eight least significant bits of the instructions, the accumulator data or external data that are coming from an external source and are attached to the external bus board connector. Truth tables of the control circuit follows. In Table 3-2 the control circuit of a basic set of the instruction set is shown.

OPCODE	ADDR_MD	DM_RD	ALU_MD	ACC_LD	ACC2BUS	EXT2BUS	DM_WR	IM_RD	PC_MD
ADD	0	1	000	1	0	0	0	0	0
SUB	0	1	001	1	0	0	0	0	0
AND	0	1	010	1	0	0	0	0	0
NOR	0	1	011	1	0	0	0	0	0
INPUT	0	0	Don't Care	0	0	1	1	0	0
LOAD	0	1	1-Don't Care	1	0	0	0	0	0
STORE	0	0	Don't Care	0	1	0	1	0	0
JUMP	0	0	Don't Care	0	0	0	0	1	1

Table 3-2 Truth table of basic instructions control circuit

In Table 3-3 the truth table for the branch instructions is shown. The primary signals for the execution of branch instructions are shown in the table below. Secondary signals are the signals acc2bus, dm_rd, dm_wr, acc_ld, addr_md and pc_ld which must be driven low except from the pc_ld signal which must be high.

OPCODE	ALU_MD	IM_RD	PC_MD	ACC_SIGN	ACC_ZERO
BEQ (taken)	000	Don't Care	0	Don't Care	0
BEQ	000	1	1	Don't Care	1
BNE (taken)	000	1	1	Don't Care	0
BNE	000	Don't Care	0	Don't Care	1
BLT (taken)	000	Don't Care	0	0	Don't Care
BLT	000	1	1	1	Don't Care
BGE (taken)	000	1	1	0	Don't Care
BGE	000	Don't Care	0	1	Don't Care

Table 3-3 Truth table of branch instructions control circuit

The truth table for indexed instructions are divided in two stages; the first clock cycle of their execution and the second. Such instructions may use the temporary register of the processor which hold the bus data from the previous clock cycle. Temporary register is selected from the addr_md signal, by driving it high. In Table 3-4 the truth table of the indexed instructions is shown.

OPCODE	DM_RD	DM_WR	ACC_LD	PC_LD	IM_RD	PC_MD	ALU_MD	ADDR_MD
JUMPX	1	0	0	1	0	1	Don't Care	0
$LOADX_{1st\ cc}$	1	0	0	0	0	Don't Care	Don't Care	0
$LOADX_{2nd\ cc}$	1	0	1	1	0	0	1 -Don't Care	1
$STOREX_{1st\ cc}$	1	1	0	0	0	Don't Care	Don't Care	0
$STOREX_{2nd\ cc}$	0	1	1	1	0	0	Don't Care	1
$ADDX_{1st\ cc}$	1	0	0	0	0	Don't Care	Don't Care	0
$ADDX_{2nd\ cc}$	1	0	1	1	0	0	000	1

Table 3-4 Truth table of indexed instructions control circuit

4. IMPLEMENTATION

The implementation of the system consists of a top-level module which instantiates all the system modules, such as the seven-segment display controller, the input controller and the processor module. Instruction and data memory modules also are instantiated inside processor module. Inputs and outputs of the top-level module are all the signals that are connected to the board FPGA device. Description of the processor primary input signals are described in APPENDIX A. As it concerns the non-processor signals, a description of them follows.

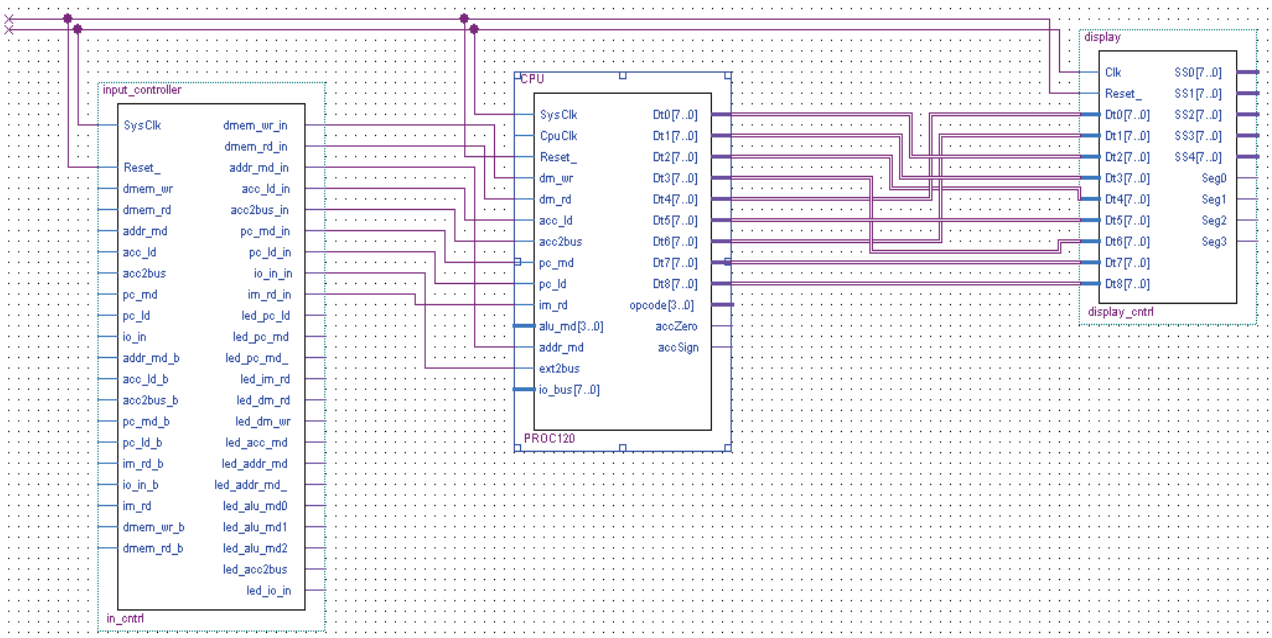

Figure 3 Block diagram of the system

In addition, the system block diagram can be seen above in Figure 3. The unconnected signals on the above figure mean that they are assigned to the FPGA pins of the board. All the other signals are internal signals that are used in the implementation and the connection of the system.

SIGNALS DESCRIPTION

Inputs
SysClk: FPGA device clock. Clock speed is 17Mhz.
Reset_: FPGA device reset signal but also a global reset for all the sub-components of the system. (Active Low)
CpuClk: An external signal generated from a hardware de-bounce circuit in order to filter the bounces of the push-button which is the source of the signal. This signal is emulating the processor clock and it is used to control the processor.

Outputs
SS0-SS3: Seven segment output signals to drive the displays
Seg0-Seg3: 7-Segment Select signal
LED's: led_signal_name

4.1. PROCESSOR

Processor module as described in CPU.v is consituted from the ALU (Arithmetic Logic Unit) , the two parameterized memories ; the Instruction memory and the Data memory and logic in order to implement the data path of the processor as described in Figure 2. The processor except from the modules mentioned before contains the program counter register, the temporary register used from some instructions, the bus control logic and also the circuit for the emulated clock provided from the boards push-button.

```
reg CpuClkD0, CpuClkD1, CpuClkD2;
//Input Sampling
always @(posedge SysClk)
        CpuClkD0 <=  CpuClk;
//Stabilizing Signal
always @(posedge SysClk)
        CpuClkD1 <=  CpuClkD0;
always @(posedge SysClk)
        CpuClkD2 <=  CpuClkD1;

//Edge Detection
wire CpuClkEdg = CpuClkD1&~CpuClkD2;
```

PROGRAM COUNTER

The program counter register is implemented as a register with synchronous reset and load enable from the code below and it can thought as the fetch unit for the processor. The input of the program counter is selected from the pc_md signal .Driving the pc_md signal active low has as a result to load the program counter with the data of the bus in order to execute some instructions such as branches and jumps, else the PC register increments by one. Also the data are captured to the PC register during the system clock positive edge, and the PC register has a synchronous reset signal which initializes the PC with zero. The pc_ld and the CpuClkEdg signal act as a load enable to the pc register. So in order to store the data to the PC register we must drive high the pc_ld signal and the CpuClkEdg signal which will be described below to be high.

PROCESSOR CLOCK

Apart from the system clock of the FPGA device, the processor clock is given as input to the fpga device from the CpuClk pin. This pin is connected to a push-button and a pin so the user can bypass the push-button

```
wire [7:0] PCmux = ~pc_md ? PC+1: bus;

 always @(posedge SysClk )  begin
        if(~Reset_) PC <= 8'h00;
        if(pc_ld&CpuClkEdg) PC <= PCmux;
        end
```

and provide the clock with an external one. In order to make this push-button work as the processors clock it must be synchronized with the system clock and make it glitch free. From this signal we generate the CpuClkEdg signal by synchronizing the asynchronous CpuClk external signal with two flip-flops and de-bouncing it at the same time from the code above. The code describes a circuit that uses two flip-flops and an AND gate with one of its input inverted. With that circuit we avoid the glitches that the push-button may have and the problem that exists with the asynchronous nature of the signal. Because of the frequency of the clock, the CpuClk signal would be asserted for thousands of clock cycles, thing that we want to avoid. A solution to this is to use an edge-detector in order to assert the output high for only one clock cycle when the input stream changes from high to low. Synchronization of signal assumes that the meta-stability resolution of the flip-flops used must be less than the clock period.

TEMPORARY REGISTER

In addition to the Program Counter, the CPU will need other registers that are not directly controlled by the user, but are necessary for data transfers within the CPU. Indexed instructions as described before require that the data of the bus of the previous cycle is stored to a register. This is done by implementing a simple register with a synchronous reset similar to the program counter register but without the CpuClkEdg signal to act as a load enable.

BUS CONTROL LOGIC

In order to control all the components that are connected to the bus , such as the lfsr module, the memories and the accumulator a simple control circuit is implemented that ensures that only one component can drive the bus each time. This is done from the code below. Also in case of two or more signals are driving it, the bus takes a pseudo-random value generated from a linear feedback shift register with period approximately 2-3 Hz. Signals that control the bus are the acc2bus, dm_rd, im_rd and the io_in.

```
always @( AccDtOut  or dmDtOut or imDtOut or im_rd or dm_rd
        or acc2bus or io_in or lftmp or io_bus )
 begin
  if (~io_in &acc2bus  &~dm_rd  &~im_rd) bus <= AccDtOut;
  else if (~io_in &~acc2bus &dm_rd  &~im_rd)  bus <= dmDtOut;
  else if (~io_in &~acc2bus &~dm_rd  & im_rd) bus <= imDtOut[7:0];
  else if  (io_in &~acc2bus &~dm_rd  &~im_rd)  bus <= io_bus;
  else if  (~io_in &~acc2bus &~dm_rd  &~im_rd) bus <= lftmp;
  else      bus <= lftmp;
end
```

LINEAR FEEDBACK SHIFT REGISTER

The logic used in order to create the processor bus uses an 8-bit linear feedback shift register in order to produce a pseudo-random value for the bus when more than one is driving the bus. Linear feedback shift registers have the property of acting like pseudo-random number generators. Typically, they use feedback from various stages of an n-bit shift register connected to the first stage by means of XNOR or XOR gates. The number of patterns they generate is $2^n - 1$. A feedback connection for a variety of n values that produces maximum-length pseudorandom sequences is described in Peterson and Weldon [3]. In the specific

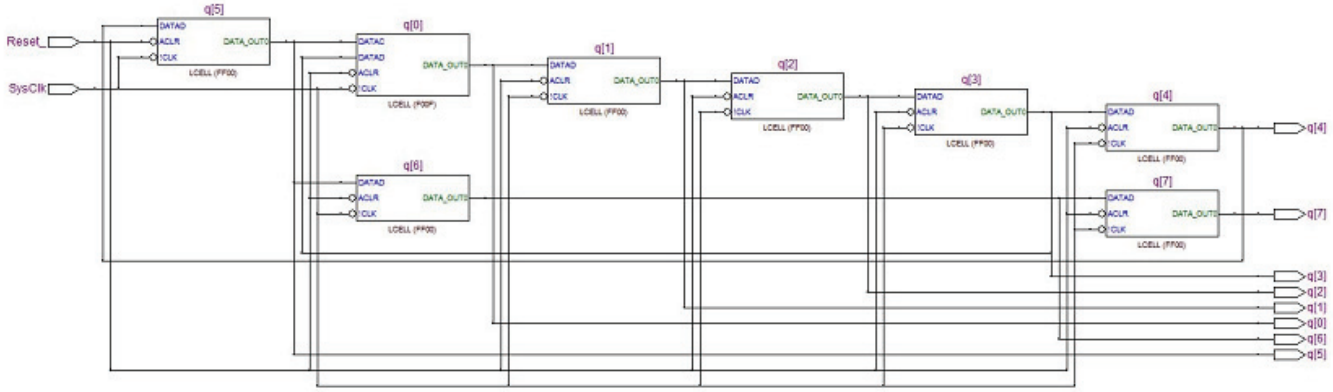

Figure 4-2 Linear Feedback Shift Register Block Diagram

implementation we use a single feedback connection with only 2 values. A block diagram of the linear feedback shift register that is implemented can be seen in Figure 4-2 Linear Feedback Shift Register Block Diagram above. It is build from a simple 8-bit shift register with the only difference that a feedback connection is used to the first flip-flop by xoring two values of the shift register. Also the linear feedback shift register is never initialized with zero because it will cause to stuck to the same value.

ARITHMETIC LOGIC UNIT

The processing and manipulation of data normally consists of arithmetic operations such as addition, subtraction and multiplication of integers and logical comparisons such as bitwise AND, OR, NOT, XOR and other Boolean operations. The CPU's instruction decoding logic determines which particular operation the ALU should perform, the source of the operands and the destination of the result. The ALU implemented has five operation modes. These are addition, subtraction in 2's Complement, AND, NOR and a special mode called passB which passes the second input of the ALU to the output. Also the ALU creates the accSign and accZero signals. The first one represents the sign of the accumulator value and the accZero when is high the accumulator value is equal to zero. A detailed table of the modes can be seen in the table /ref/. The width (in bits) of the words that the ALU handles is usually the same as that quoted for the processor. In this design and in all accumulator-based processors, any data to be processed is temporarily stored in the accumulator and the result is ending up in the accumulator before being stored in the memory unit. Always one of the two operands of the ALU is the Accumulator therefore ALU is implemented with the Accumulator in a separate module. The modes of the ALU unit is controlled from the alu_md pins which are described in the table below.

ALU_MD	Operation
0000	Accumulator + DataIn
0001	Accumulator – DataIn (2's complement)
0010	ACC AND DataIn
0011	ACC NOR DataIn
default	ACC = DataIn

For the accSign and accZero signals the ALU is responsible to handle them by checking the leftmost bit of the accumulator for the accSign signal and compare the accumulator with the zero for the accZero sign. The accumulator register is implemented inside the ALU unit because it was very simple. It is implemented as a register with load enable with synchronous reset just like all the other registers in the system. The accumulator data port is connected to the ALU output so it could store the calculated data in during a CPU positive clock edge and when the acc_ld signal is high. Also the output of the accumulator register is connected to the first input of the ALU because it is always the one operand in our architecture. As it concerns the communication of the accumulator and the bus, the acc2bus signal when is high allow the data of the accumulator to be written to the bus.

MEMORIES

In order to implement the processors memories (instruction and data), the library that Altera Quartus provide was used. This Library of Parameterized Modules (LPM) allows implementing various types of memories, and in our case RAM for the data and ROM for the instructions. The memories provided by Altera are supported to all Altera PLD devices, also these memories are fully parameterized and they can be changed using the MegaWizard Plug-In Manager. The parameters that the two implemented have, concerns the address port and the output data port width. Also for the initialization of the memories a special file MIF(Memory Initialization File) is used in order to initialize the memories with the desired contents.
Two types of memories are implemented for the processor. One is the instruction memory which is implemented as read only memory (ROM) and the data memory which is implemented as random access memory (RAM). The functions provided from altera to implement the above memories are the lpm_rom and lpm_ram_dq modules. Memories widths are 12 bit for the instruction memory and 8 bit width for the data memory. The depth of both memories is 256 words.
Interfacing the memories to the processor was simple. The only need was to connect properly the ports of the memories to the rest of the design as are shown in Figure. Read and write operations on the memories need only to assert properly the appropriate signals that correspond to the memory. So for a read operation only dm_rd signal must be high for the data memory and of course the desired address that the data resides. Instruction memory on the other hand reads a new instruction every clock cycle (SysClk) after the address is supplied. Write operation is permitted only to the data memory via the dm_wr signal. Write on the ROM cannot be done because of the fact that the memory is read only thus a write would result to a self-modifying program, executed from the processor.
The data memory module uses only one signal to write or read data, but in our case, separate signals (dm_rd and dm_wr) manipulates the operations of the data memory.
Simulation waveforms for the read/write operations of the memories are shown in APPENDIX A. Also the implementation of these two memories occupied one EAB[4] which corresponds to a memory block of 256x8 for the RAM and four combined EAB's for implementing the ROM.

4.2. DISPLAY CONTROLLER

The educational based board provided has a number of 7-segment displays connected to the FPGA device. So the values of various signals and register can be displayed in these displays that exist on the board. Typically a 7-segment display consists of 8 LEDs, without the dot. Each segment uses a separate led to illuminate, a combination of segments display a digit or a number on the display. In the board the 7-segment displays has two digits and they have their cathodes tied together in order to conserve pins. The display controller that was implemented multiplexes the 7-segment displays in order to save pins because we would need approximately 72 pins for all the displays. The display controller lit the half of the displays at any time, and by switching between them with a rate above 100Hz it seems that all the displays are lit.

[4] **EAB** Embedded Array Block is a flexible block of RAM with registers on the input and output ports, and is used to implement common gate array mega-functions. The EAB is also suitable for functions such as multipliers and error correction circuits, because it is large and flexible. Furthermore EABs can be combined together to implement more complex functions or they can be used separately. Each one provides 2,048 bits for creating various memory functions like RAM, ROM and FIFO's

The seven-segment driver takes as input the nine registers which we want to display and splits them into two 4-bit quantities. Values displayed on the seven-segment display are 8 bit wide, and they have hexadecimal format. In order to drive a 7-segment display correctly we must first decode the 4-bit data to 8-bit data that can be displayed on the seven-segment display by illuminating the LED segments that correspond to the given pattern. This is done from the DisplayDec module. Some 7-segment patterns are shown in the code below.

```
always @(In) begin
      case (In)
        5'b00000 : Out = 8'h3f;
        5'b00001 : Out = 8'h06;
        5'b00010 : Out = 8'h5b;
        5'b00011 : Out = 8'h4f;
```

In order for each of the two digits in any display has to appear bright and continuously illuminated, the two digits should be driven once every 1 to 16ms (for a refresh frequency of 1KHz to 60Hz). For example, in a 60Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for ¼ of the refresh cycle, or 4ms. The display controller assures that the correct pattern is present when the corresponding display signal is driven. The controller take the values that we want to display and for divide the input in two digits, it selects the rightmost or the leftmost bits with a frequency of 2^{17} clock cycles. Then these bits are decoded from the display decoder and the appropriate 7-segment select signal is driven in order to pass the input to the display. The 7-segment displays also are refreshed with the same frequency as the selection of the left and the right digit of the data.

4.3. INPUT CONTROLLER

The board interface provides a number of push-buttons and I/O pin connectors tied with LED's to show their values. In order to get them work with the processor implemented in the FPGA a controller is responsible for interfacing them with the processor. As it concerns the push-buttons, they have to be glitch free in order to work as control signals to the processor. Also they had to be synchronized with the system clock because they were asynchronous. This was done to similar way as the processor clock signal. A counter was used to decide the state of the push-button by maxing out the counter. Also synchronization to the system clock domain was needed by using two flip-flops. Furthermore, for each input signal to the FPGA we have two choices. These are to drive the corresponding input pin or to use the push-button. The controller simple XOR's the two signals to ensure that only one will drive the FPGA input pin. LED's for each input signal also illuminate at every change of the signal by simply asserting high the FPGA pin that they are connected.

5. DESIGN METHODOLOGY

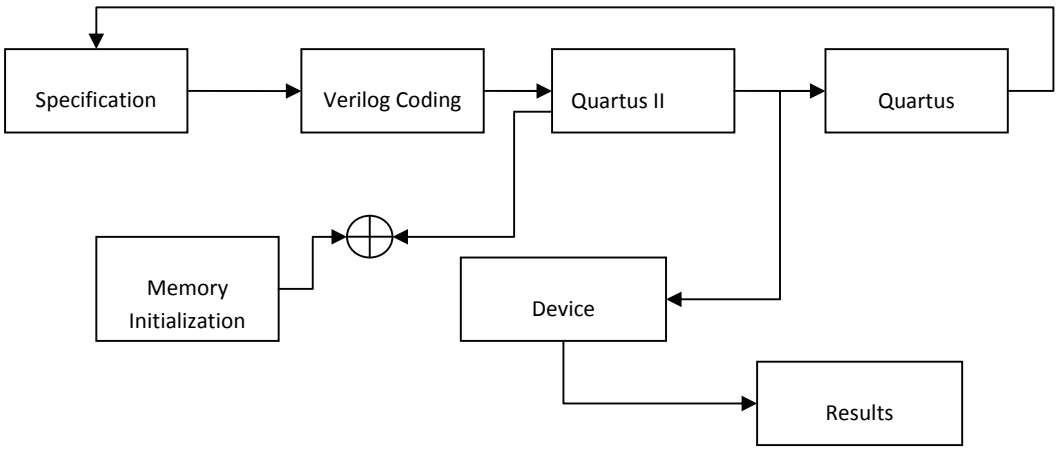

Figure 5 Design Process

Hardware design is done with CAD tools. The first step in the hardware design is to prepare the specification of the design. The architecture and the instruction set must be understood thoroughly. Design ideas are then described in Verilog HDL[5] using Quartus design software. If the design is synthesized successfully, synthesis produces a Netlist[6] file which is used for compile and verification through simulation. The hardware design process is repeated until the system is complete without any errors. Hardware implementation is performed by downloading the design in the targeted FPGA device where the board provides. The hardware implementation tests the design in real physical environment by some control applications. Different code must be written and stored to the instruction memory of the processor, before it can be executed correctly . So, before the design is downloaded to the FPGA device, the specific code must be written. The code is written to the memory initialization file of the processor instruction memory and then the system is recompiled in Quartus. Then we program the device with the new programming file that the design software created. Testing the correctness of the design was done by implementing a control circuit in a FPGA development board and connecting it to the FPGA-based educational board where the real design exists.

5.1. DESIGN SOFTWARE CONFIGURATION

During the development of the system, proper configuration of the tools used was done in order to optimize the design for space and to ensure the proper synthesis of the described design. The analysis and synthesis settings that was used, different from the default ones are described below.

[5] **Hardware Description Language** (HDL) is any language from a class of computer languages for formal description of electronic circuits. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation.

[6] **Netlist** describes the connectivity of an electronic design. Netlists usually convey connectivity information and provide nothing more than instances, nets, and maybe some attributes. A lot of time, they are designed for input to simulators.

- Auto ROM Replacement : Off
- Auto RAM Replacement : Off

These options was turned off because it was found that they changed the functionality of the design, fact that the vendor of the Quartus design software mentioned. Generally the above options try to found logic that feed into library modules that are provided from the FPGA vendor such as lpm_ram_dq and lpm_rom.Also the optimization technique used for the design was for speed because of the fact that the area of the design is quite small. Specifically, the logic elements used from the top-level design are 580 out of 1.152 and for memory bits are 5.120 out of 24.576 for the specific FPGA device. As it concerns the pins, 99/147 are used. All the pins are declared as input or output, bi-directional pins does not exist in the design. The Pin assignment can be found in (Kalokarinos, Ligerakis, Tsaliagos 2007).

APPENDIX A

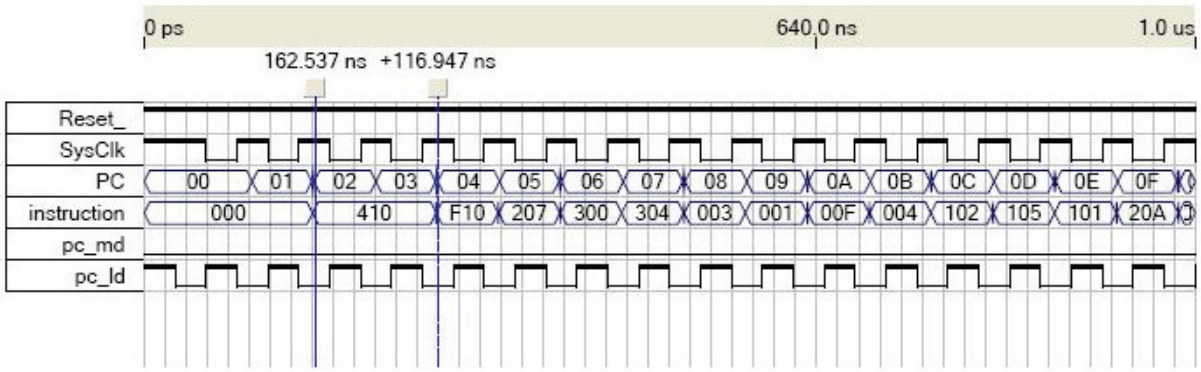

Figure A-1 Write operation (RAM)

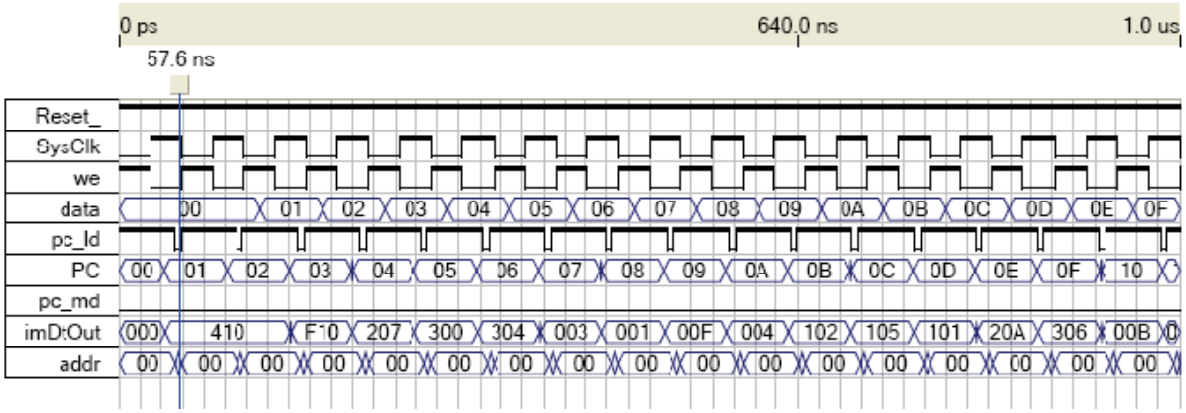

Figure A-2 Read Operation (ROM)

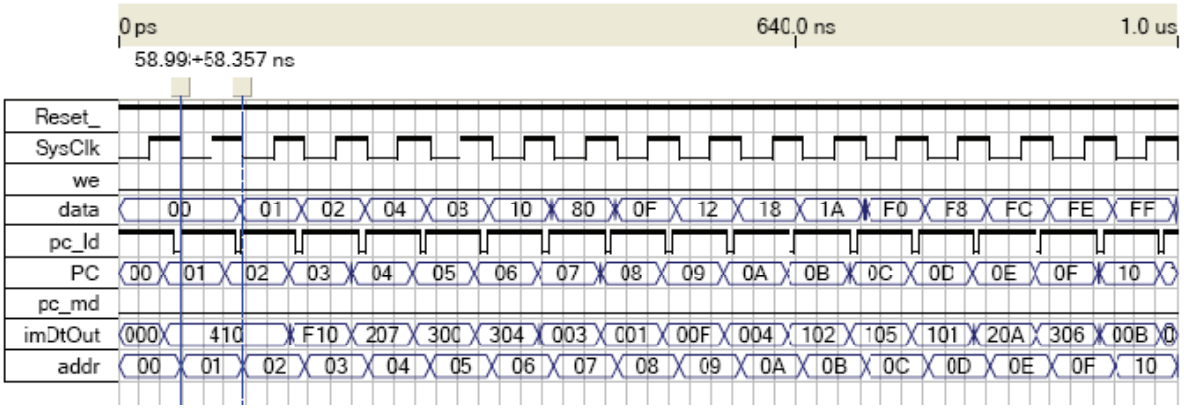

Figure A-3 Read Operation (RAM)

BIBLIOGRAPHY

Altera. *FLEX 10K Embedded Programmable Logic Device Family Data Sheet.* 2003.

Altera. *Introduction to Quartus II Getting Started Manual.* 2004.

Kalokarinos, Ligerakis, Tsaliagos. *An FPGA Educational Board Simulating a Simple Accumulator-Based Processor Datapath.* Heraklion: ICS-FORTH, 2007.

W.W. Peterson and E.J. Weldon, Jr. *Error-Correcting Codes.* Mass.: MIT Press: Cambridge, 1972.