Nepal Journal of Science and Technology Vol. 15, No.1 (2014) 81-88

Design and Implementation of Synthesizable 32-bit Four Stage Pipelined RISC Processor in FPGA Using Verilog/VHDL

Bikash Poduel[1], Prasanna Kansakar[1], Sujit R.Chhetri[1] and Shashidhar Ram Joshi[2]

[1]*Department of Electronics and Communication, Thapathali Campus, Institute of Engineering, Kathmandu Nepal*

[2] *Department of Electronics and Communication, Institute of Engineering, Pulchowk Campus, Lalitpur*

e- mail: *065bex410@ioe.edu.np*

Abstract

This paper is delineating the design and implementation of high performance, synthesizable 32-bit pipelined Reduced Instruction Set Computer (RISC) Core. The design of the Harvard Architecture based 32-bit RISC Core involves design of 32-bit Data-path Unit, Control Unit, 32-bit Instruction Memory, 32-bit Data Memory, Register file with each register of size 32 bit. The processor is divided into Fetch, Decode, Execute and Write Back block in order to implement a four-stage pipeline. A 2*16 LCD is connected to the processor IO block to show the instruction execution sequence for demonstration in FPGA. The RISC Core is designed using Verilog HDL and VHDL and is tested in ISIM Simulator. The implementation of the processor is done in a Spartan 3E Starter Board using Xilinx ISE 14.7. All of the instructions incorporated with the processor have been tested successfully both in simulation and hardware implementation in FPGA.

Key words: SoC, Harvard architecture, Xilinx ISE, IP Core, Register file.

Introduction

The Reduced Instruction Set Computer (RISC) is a design philosophy used in powerful microprocessors and micro-controllers (Stallings 2011). This design philosophy involves fixed length instruction, single clock cycle execution for most of the instructions, only load and store instruction for memory access, pipelined execution, and a large register bank for fast memory operation. The most common RISC microprocessors are ARM, DEC Alpha, PA-RISC, SPARC, MIPS, and IBM's Power-PC.

This project involves designing a high performance 32-bit synthesizable RISC core in Verilog-HDL and VHDL. The implementation of the core is done in Spartan 3E FPGA. FPGA (Kilts & Steve 1978) is the short hand for Field Programmable Gate Array, which is capable of implementing combinational, sequential, and FSM based digital systems or components. Today's deep submicron fabrication technologies enable design engineers to implement an impressive number of components like microprocessor, memories, and interfaces in a single microchip called Configurable System-on-Chip (Kim & Leibson 2005). With the advent of large, fast, cheap FPGAs, it is practical and cost effective to skip the ASIC (Xilinx 2005) and ship volume embedded system in a single FPGA since FPGA implements all of the system logic including a processor core.

This project has a very promising future in the context of the synthesizable embedded system design, verification, and implementation. There is not a single company in Nepal till this date which works in designing, manufacturing, and verifying Hi-Tech electronic product as microcontrollers, simple low end phone set to high end phones like smart phones, processor cores, etc. Every year thousands of electronics Engineers gets B.E. degree from various Universities here in Nepal but there are no electronic

jobs. This Hi-Tech business has a huge upside potential technically and economically thus creating myriad of job opportunities. This paper is inception from our team to start writing and contributing IP Cores, which in our case, involves designing, and verifying processor core. This paper, we believe, shall be the reference for students who wanted to learn designing and verifying processor core, for teachers who want to teach a simple real world implementation of processor core in FPGA, and for all of those who wanted to be a contributor of IP Cores. This project shall emanates a good insight of a real deal involve in Hi-Tech work which is designing and verifying Cores as Processor, Bus, Peripherals, etc. thus bringing the horizon closer and clearer which was hazy since nobody has done it before here.

Methodology

Basic building block of RISC core

The HDL design of the RISC core involves the design of the following components, which are the major building blocks of most of the processor: memory unit, data path unit, and control unit.

Memory: The pre-dominant feature of a RISC Core is the use of registers from a register file, which is a fast memory, and immediate values for all arithmetic and logical instructions. This leads to less frequent accesses of the main memory. Since the RISC core is based on the Harvard Architecture (Iannucci 1988), there is separate data and instruction memory. Memory access is limited to Load and Store instructions only.

There are sixteen 32-bit registers in the register file. The instruction memory is implemented as a single port on-chip distributed ROM while the data memory is designed as a single port on-chip block RAM inside the FPGA. The data memory and instruction memory is 32 * 256 bits, which can be extended as per requirement. Since the core being designed is synthesizable, we can adjust the code to change the attributes of the core.

Data path unit: The data path unit comprises of ALU, multiplexers, Instruction Decoder, Branch Logic Generator, Pipeline Registers, destination register selection logic, Program Counter, Link Register, Data Register, etc. The pipeline progresses through four stages when executing an instruction. The first stage is the *Instruction Fetch* stage, where the address of the instruction to be fetched is loaded in the Program Counter (PC). The address is also saved in the pipelined register for internal purpose. The particular instruction pointed by PC is fetched from the Instruction Memory into the Instruction Register (IR). The second stage is the *Instruction Decode* stage, where the Instruction Decoder decodes the fetched instruction. Here, the decoder extracts out the source operands, destination operand, and the operation to be performed, from the available 32-bit instruction. The next, is the *Execution* stage where the operation on the source operands is performed and the result is generated in the internal pipeline register. The corresponding flags (Negative, Carry, Zero and Overflow) are set according to the result of the arithmetic and the logical operations performed. For the load and store instruction, the memory location where the data is to be loaded or fetched from is calculated. The final stage is the *Write Back* or *Memory Read/Write* stage. For the arithmetic and logical operation, the result of the ALU operation is written to the destination register and for the load/ store, instruction reading from the data memory or writing to the data memory is done.

Control unit: The pipeline is controlled by setting control values during each pipeline stage. Each control signal is active only during a single pipeline stage and hence the control lines can be divided according to the four-pipelined stages. These signals will be forwarded to the adjacent stage through the pipeline registers.

Instruction pipeline design: The process of breaking an instruction (process) into a number of independent sub-processes, which are capable of executing concurrently and executing such sub-processes simultaneously, is known as pipelining. In order to be successful commercially, any processor design must have a fast system clock and must be able to execute, on average, one or more instruction per clock cycle. In order to achieve this requirement, pipelining is essential.

The instruction pipeline is designed by pipelining the basic fetch-execute operation sequence shown in Fig 1. It shows, in particular, the four stage pipeline architecture of the RISC core, which is implemented in the proposed core. The interrupt path has been omitted to simplify the pipeline design. Table 1 and Table 2 demonstrate the performance improvement that can be achieved by pipelining.

Table 1. Execution of four-instruction takes 16 clock cycles without pipelining

Clock Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ADD	IF	ID	EX	WB												
SUB					IF	ID	EX	WB								
AND									IF	ID	EX	WB				
ADD													IF	ID	EX	WB

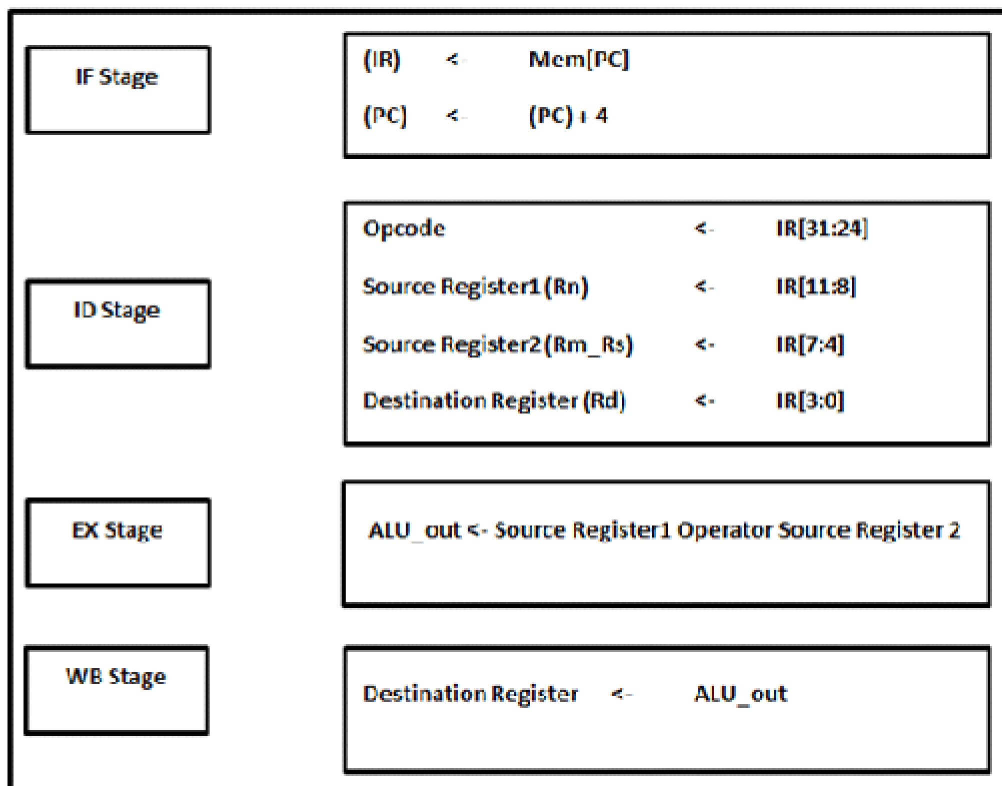

Fig.1. A four stage pipeline showing the stages and the respective operations

Pipeline hazards: Conditions that may result in the need to delay the execution of an instruction pipeline are referred to as pipeline hazard (Jiang 2006). There are three main types of pipeline hazards - structural hazard, data hazard, and control hazard.

Structural hazard refers to the situation in which two instructions must use a common resource, thereby resulting in a resource conflict. For example, a common problem of this form occurs when there is only one memory device for instruction and data and a load or store instruction is encountered. Since instruction must be fetched during each clock cycle, if an instruction in the pipeline requires an access to memory then a memory access conflict will result.

Table 2. Four-instruction execution with four-stage pipelining takes 7-clock cycles (less than half of what it takes without pipelining)

Clock Instruction	1	2	3	4	5	6	7
ADD	IF	ID	EX	WB			
SUB		IF	ID	EX	WB		
AND			IF	ID	EX	WB	
ADD				IF	ID	EX	WB

Block diagram of proposed RISC core

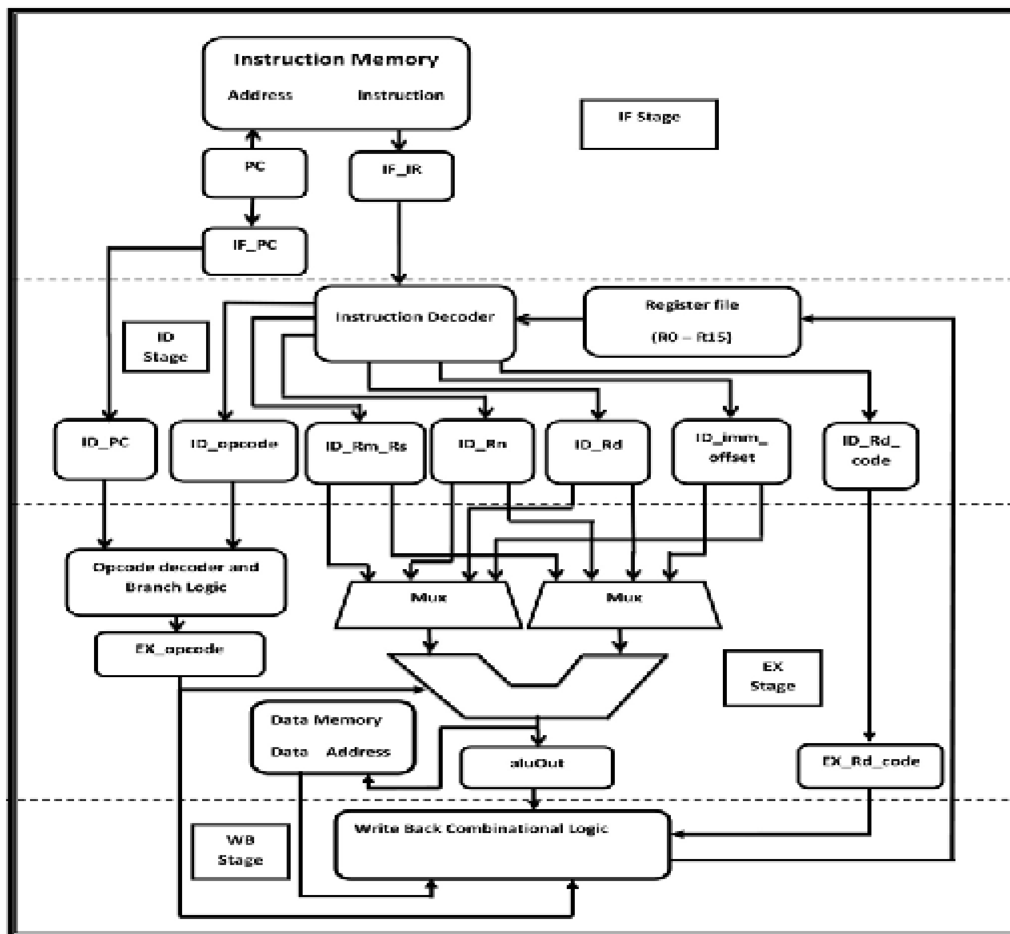

Fig.2. Internal pipelined architecture of the proposed RISC core

In data-hazard, one instruction changes the value of a register, while a following instruction uses that same register value. If the following instruction reads the register value before the preceding instruction changes the register value, the later instruction may use an incorrect value, thereby resulting in an incorrect program result.

Control hazard occurs when an unconditional or conditional branch instruction is encountered. In absence of branch instruction, all instructions typically can be fetched in strict sequential order. Then, if a fixed length instruction format is used, a sequence of instructions can be pre-fetched before the first instruction of the sequence has completed the execution. However, if the first instruction happens to be a branch instruction, then all subsequent instructions fetched in this manner will be incorrectly fetched instructions.

Pipeline hazard solution adopted: In order to remove the structural hazard, a separate memory for the data and instruction is used. Thus, there are two sets of the address and data lines in the processor. There is no any mechanism used to remove the data hazard. It is left for the assembly programmer to perform instruction scheduling in the assembly to remove the data hazard. For the control hazard, whenever a branch instruction is encountered in which the branch takes place, the following instructions currently in the pipeline are marked and subsequent instructions are fetched, starting from the branch-target address. Whenever a marked instruction is encountered during the execution stage, the result of that instruction is nullified by not storing the execution result for that instruction.

Instruction set: The RISC core has 32-bit instructions, which mostly are register based operations. Since register are fast memory RISC core mostly uses register based addressing mode. However, there is support for immediate operand and loading/storing mechanism for memory access.The instructions are divided into four categories, which are Data Processing Instructions, Load/Store Instruction, Interrupt Instruction, and Branch Instruction.

Data Processing Instruction

Table 3. List of data processing instructions and its respective address fields

Instruction	Instruction Word (IR[31:0])				Action
	Opcode (IR[31:24])	Destination (Rd)	Source2 (Rn)	Source1 (Rm_Rs)	
ADD	0000_0000	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rn + Rm_Rs
ADC	0000_0001	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rn + Rm_Rs + cFlag
SUB	0000_0100	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rn - Rm_Rs
RSB	0000_0101	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rm_Rs – Rn
AND	0000_1000	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rn&Rm_Rs
OR	0000_1001	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rn | Rm_Rs
XOR	0000_1010	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rn ^ Rm_Rs
BIC	0000_1011	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rn& (!Rm_Rs)
LSL	0001_0000	IR[119:16]	IR[15:12]	IR[11:0] (Immediate)	Rd ← Rn<< Imm#12
LSR	0001_0001	IR[119:16]	IR[15:12]	IR[11:0] (Immediate)	Rd ← Rn>> Imm#12
CMP	1000_0000	IR[11:8]	IR[7:4]	IR[3:0]	Rd ← Rn - Rm_Rs
MOV	0010_0000	IR[15:12]	-	IR[11:0] (Immediate)	Rd ← Imm#12

Load/store instruction

Table 4.List of memory access instructions and their respective address fields

Instruction	Instruction Word(IR[31:0])				Action
	Opcode (IR[31:24])	Destination (Rd)	Source2 (Rn)	Source1 (Rm_Rs)	
LDR	0011_0000	IR[11:8]	IR[7:4]	IR[3:0]	Rn ← M[Rd]
STR	0011_0001	IR[11:8]	IR[7:4]	IR[3:0]	M[Rd] ← Rn

Branch instruction

Table 5. List of branch instructions and their respective address fields

Instruction	Instruction Word(IR[31:0])				Action
	Opcode (IR[31:24])	Destination (Rd)	Source2 (Rn)	Source1 (Rm_Rs)	
B	0100_0000	-	-	IR[11:0](Immediate)	Branch_Target ← Imm#12

Interrupt instruction

Table 6. List of interrupt instructions and their respective address fields

Instruction	Instruction Word(IR[31:0])				Action
	Opcode (IR[31:24])	Destination (Rd)	Source2 (Rn)	Source1 (Rm_Rs)	
HALT	1111_1111	-	-	-	Halt the processor

Programmer model of the RISC core

There RISC core executes 32-bit instruction. The registers are all 32-bit wide. The ALU is 32-bit. The programmer model for this RISC core is shown in Table 7. There are 16 visible registers (R0-R15) each 32-bit wide and four status registers.

The Program Counter (PC) contains the address of the instruction to be fetched from the memory. The Link Register (LR), used with sub-routine calls, contains the address of the next instruction after the sub-routine call. Thus, when a return instruction is executed from within the sub-routine the Link Register points to the instruction to be executed after the return. The Stack Pointer (SP) points to the top of the Stack, which holds the data contents in a last-in-first-out manner. The condition flags are Carry, Zero, Negative, and Overflow which are all 1 bit wide. These bits are set or reset during any arithmetic and logical operations.

Table 7. Programmers model for the RISC core

General purpose registers (32 bit)		Flags (1 bit)	Special Purpose Registers (32 bit)
R0	R8		
R1	R9		
R2	R10		
R3	R11		
R4	R12	nFlag	
R5	R13	zFlag	SP
R6	R14	cFlag	LR
R7	R15	vFlag	PC

Implementation in HDL

This 32-bit RISC core is designed in both of the HDLs, Verilog and VHDL. The block diagram of the module

being implemented is shown in Fig 3. The lists of the ports are in Table 8. There are three input ports, five output ports, and one bi-directional port. The RISC core interacts with instruction memory and data memory for the instruction fetching and data load/ store respectively. The instruction memory and data memory module is shown in Fig 4.

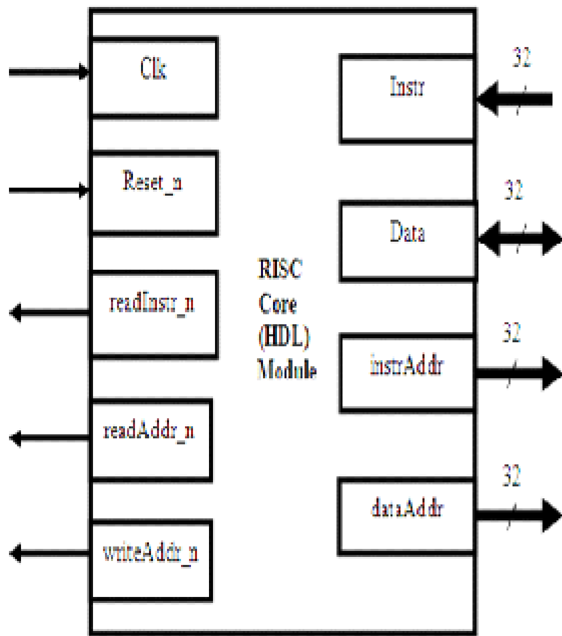

Fig.3. HDL module for proposed RISC core

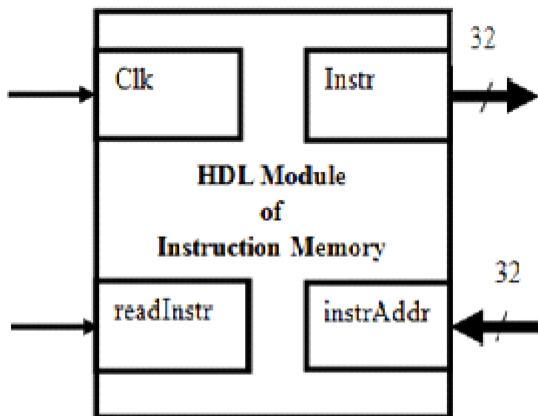

Fig.4. HDL module of instruction memory

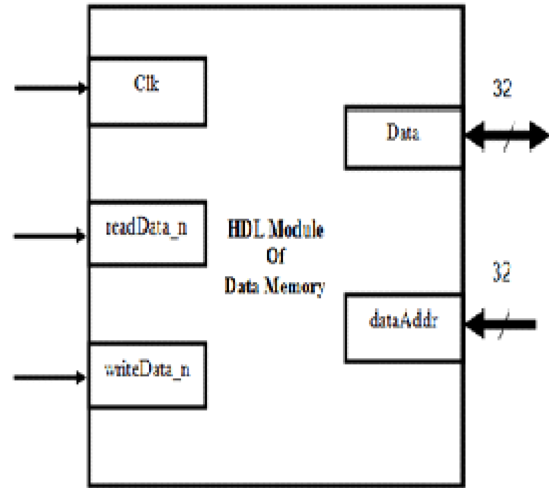

Fig.5. HDL module for data memory

Table 8. List of the port for RISC core

Port Name	Width (bits)	Direction	Sensitivity (Signal edge)
Instr	32	Input	-
Data	32	Input/output	-
instrAddr	32	Output	-
dataAddr	32	Output	-
readInstr_n	1	Output	Negative
readData_n	1	Output	Negative
writeData_n	1	Output	Negative
Clk	1	Input	Positive
Reset_n	1	Input	Negative

Results and Discussion

The simulation of the proposed RISC core was done in the Xilinx ISIM simulator where the actual core is instantiated along with the data and instruction memory in the test bench. The instruction memory was loaded with ADD, SUB, AND, and ADD instruction in the sequence, which was fetched, decoded, and executed by the processor. All of the instructions are simulated correctly and the result is shown in Table 9. In addition, the final synthesis report of the RISC core generated by Xilinx ISE is in the Table 10.

Table 9. Result of execution of the instructions

Operation	Source Register1	Source Register2	Destination register	Flag
ADD	32'h01	32'h02	32'h03	-
SUB	32'h09	32'h05	32'h04	-
ADC	32'h0A	32'h02	32'h0D	-
RSB	32'h01	32'h05	32'h04	-
AND	32'hFF	32'h0F	32'h0F	-
OR	32'hFF	32'h0F	32'hFF	-
XOR	32'hFF	32'h0F	32'hF0	-
BIC	32'hFF	32'h0F	32'h0F	-
CMP	32'h09	32'h09	-	zFlag
MOV	-	#32'h15 (Imm)	32'h154	-

Table 10. Result of operation of the instructions

Logic Utilization	Used	Available	Utilization
Number of Slices containing related logic	1472	4656	31.62%
IO Blocks	67	232	28%

The simulation and result of this processor verifies all of the instructions incorporated in the RISC core. The RISC core is useful to develop a 32-bit micro-controller by simply adding the peripherals and a bus. Since this core is synthesizable and reconfigurable one can upgrade it by increasing the memory of the processor, by moving up to 5-stage pipeline, by adding the data-forwarding technique to remove data hazard, by adding branch prediction block to remove the control hazard.

Acknowledgements

We take this opportunity to express our profound gratitude and deep regards to Professor Dr. Shashidhar Ram Joshi for his exemplary guidance, monitoring and constant encouragement throughout the course of this research project. The support, help and guidance given by him time to time shall carry us a long way in the journey of life on which we are about to embark. We would like to thank Mr. Sandesh Ghimire and Ballav Bhattarai to read our paper and to provide valuable advices.

References

Hong J. 2006. Pipeline: Hazards. cse.unl.edu/~jiang/cse430/Lecture%20Notes/Pipeline_Hazards.ppt.

Iannucci, R. A. 1988. Towards a Dataflow / Vons Neumann Hybrid Architecture.In: *Proceedings of the 15th Annual International Symposium on Computer architecture* (May 1988), IEEE Computer Society PressLos Alamitos, CA, USA, pp. 131-140.

Kilts, Steve.2007. *Advanced FPGA Design: Architecture, Implementation, Optimization*.Wiley-IEEE Press, New Delhi, India. Pp. 170-265.

Kim,James and Steve Leibson. 2005. Configurable processors: A new era in chip design.*computer* **38**:51-59.

Stallings, W. 2011. *Computer organization and architecture, designing for performance.* Dorling Kindersley India Pvt. Ltd., New Delhi, India.pp. 498-536.

Wikipedia.2011. Semiconductor intellectual property core. http://en.wikipedia.org/wiki Semiconductor_intellectual_property_core.

Xilinx.2000. FPGA vs ASIC. http://www.xilinx.com/fpga/asic.htm