International Journal For Technological Research In Engineering
Volume 4, Issue 7, March-2017 *ISSN (Online): 2347 - 4718*

IMPLEMENTATION OF 32 BIT RISC PROCESSOR ON FPGA

Payal Sakre[1], Nitesh Dodkey[2], Siddarth Singh Parihar[3]
[1]M.Tech Scholar, [2]HOD ECE, [3]Assistant Professor
Department of Electronics & Communication Engineering, Surabhi Group of Institutions, Bhopal (M.P)

Abstract: In this paper, hardware implemented a entropy encoder using Rice PSI 1,k encoding scheme. The design can calculate the value of k in real time and then selects the number of pass through bits. The adder used in this design is serial adder, this reduces the resource usage and also reduces the power consumption of the design. The target device to implement the design is Virtex 5 FPGA. Xilinx XST is used to synthesize the design and it is coded in VHDL.
Keywords: Rice algorithm, PSI1,k, Entropy Encoding, FPGA

I. INTRODUCTION

Although field programmable gate arrays (FPGA) were introduced a decade ago, they have only recently become more popular. This is not only due to the fact that programmable logic saves development cost and time over increasingly complex ASIC designs, but also because the gates count per FPGA chip has reached numbers that allow for the implementation of more complex applications. Many present day applications utilize a processor and other logic on two or more separate chips. However, with the anticipated ability to build chips with over ten million transistors, it will become possible to implement a processor within a sea of programmable logic, all on one chip. Such a design approach would allow a great degree of programmability freedom, both in hardware and in software: CAD tools could decide which parts of a source code program are actually to be executed in software and which other parts are to be implemented with hardware.

The hardware may be needed for application interfacing reasons or may simply represent a coprocessor used to improve execution time. Most computationally complex applications spend 90% of their execution time in only 10% of their code [1 thesis4]. The basic instructions executed in this 10% of the code of a given program naturally differ from application to application. These observations make the idea of a fast, yet general purpose CPU seems inconsistent. The custom compute machine (CCM), which can be customized on a per application basis, appears to be the solution to the contradiction of general purpose computing and high performance processing.

II. COUNTER CODE ALGORITHM

FPGAs have evolved significantly over recent years. From simple, regular arrangements of configurable logic blocks and routing, modern devices now boast increased complexity, in terms of both size, and the variety and capability of primitives offered. Much of this improvement has inevitably been driven by market segments where FPGAs are particularly popular, such as communications and signal processing. This is due to the ease with which such algorithms can be parallelized on FPGAs and the availability of high-level programming techniques that simplify the design process. Hence, it is not surprising to find that FPGAs have evolved to better suit such applications. The Virtex II brought with it embedded multipliers.

A large number of signal processing algorithms make use of multiplications. By embedding hard multipliers into the silicon, it becomes possible to optimize them for performance while saving the remaining resources for other uses. These later evolved into DSP Blocks: multiply-accumulation units that support the full requirements of a DSP filter tap. Recently, FPGAs have moved beyond implementation of accelerators for complex algorithms, now housing full systems.

Processors are useful when dealing with non-streaming data, in systems with multiple heterogeneous hardware tasks, and for managing complex interfacing. Vendors did previously introduce devices with embedded hard processors such as the PowerPC 405 in the Virtex II Pro, and the PowerPC 440 in the Virtex 4 FX. While these high-end FPGA devices did find an audience, they were out of the budget of many, and so, “soft" processors, implemented using logic resources, have continued to dominate. In this research, we connect these two threads. DSP Blocks are indeed highly capable primitives, yet leveraging them outside the DSP domain is extremely difficult, as they were primarily designed to suit such applications. This research investigates the feasibility of building custom soft-core processors that can allow DSP Blocks to be leveraged beyond their typical target applications, and in a manner accessible to those with minimal FPGA architecture knowledge.

The Xilinx DSP48E1 cores included in the most recent Xilinx devices are highly customizable. We aim to build a lean processor around the DSP48E1, with as little extra logic as possible that supports a full set of standard machine instructions. The prospects are even more exciting when one considers that modern FPGAs have very many of these blocks; a large Virtex-6 device contains hundreds of such DSP Slices. Hence, such processors could be used to build massively parallel many-core systems. In this research, we investigate the design of a lean single processor based on the DSP48E1 primitive.

III. INSTRUCTION SET

Table 1 shows the instruction supported along with the hardware used and latency to execute the instruction, the average latency of all the instructions is approximately 4

Table 1: Instruction Set

S.No	Instruction	Operation	Latency	Hardware Used
1	MVI	MVI RmH,#16 Bit data MVI RmL,#16 Bit data	1	Register array
2	MOV	MOV Rd, Rs	3	Register array
3	ADD	ADD Rm, Rn, Rd Rd <= Rm + Rn	3	Register array, ALU
4	SUB	SUB Rm, Rn, Rd Rd <= Rm - Rn	3	Register array, ALU
5	AND	AND Rm, Rn, Rd Rd <= Rm AND Rn	3	Register array, ALU
6	OR	OR Rm, Rn, Rd Rd <= Rm OR Rn	3	Register array, ALU
7	XOR	XOR Rm, Rn, Rd Rd <= Rm XOR Rn	3	Register array, ALU
8	XNOR	XNOR Rm, Rn, Rd Rd <= Rm XNOR Rn	3	Register array, ALU
9	NOR	NOR Rm, Rn, Rd Rd <= Rm NOR Rn	3	Register array, ALU
10	NAND	NAND Rm, Rn, Rd Rd <= Rm NAND Rn	3	Register array, ALU
11	MUL	MUL Rm, Rn, Rd Rd <= Rm MUL Rn	3	Register array, ALU
12	COMPAEQB	COMPAEQB Rm, Rn, #9_bit_Offset	3	Register array, Comparator
13	COMPAGB	COMPAGB Rm, Rn, #9_bit_Offset	3	Register array, Comparator
14	COMPAGEQB	COMPAGEQB Rm, Rn, #9_bit_Offset	3	Register array, Comparator
15	COMPALB	COMPALB Rm, Rn, #9_bit_Offset	3	Register array, Comparator
16	COMPALEQB	COMPALEQB Rm, Rn, #9_bit_Offset	3	Register array, Comparator
17	COMPANEQB	COMPANEQB Rm, Rn, #9_bit_Offset	3	Register array, Comparator
18	LSL	LSL Rd, Rs Left shift logical Rs and then	2	Register array, Shifter
19	RSL	RSL Rd, Rs Right shift logical Rs and then	2	Register array, Shifter
20	RSA	RSA Rd, Rs Right shift arithmetic Rs and	2	Register array, Shifter
21	RL	RL Rd, Rs Rotate Left Rs and then store to	2	Register array, Shifter
22	RR	RR Rd, Rs Rotate Right Rs and then store	2	Register array, Shifter
23	MULADD	MULADD Rm,Rn,Ro,Rp Ro <= (Rm * Rm) + Rp	5	Register array, ALU
24	MULSUB	MULSUB Rm,Rn,Ro,Rp Ro <= (Rm * Rm) - Rp	5	Register array, ALU
25	MULACC	MULACC Rm,Rn,Ro,Rp Ro <= (Rm * Rm) –	4	Register array, ALU

including the instruction fetch cycle. The instruction set covers instruction for data transfer, arithmetic, logical, shifting, conditional branching and BCD operations.

IV. PROPOSED 32 BIT RISC PROCESSOR

In this work hardware implementation Soft Processor is presented. The data operand size of designs available is only 16 bit. This section shows the hardware implementation of 32 bit RISC processor. Figure 1 shows the high level block diagram of processor, there are six major units namely: register set, program memory, ALU, comparator, shifter and timing and control unit.

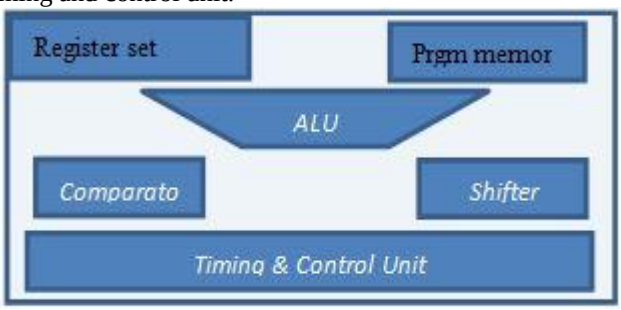

Figure 1: High Level block diagram of 32 bit RISC processor

A. Program Memory

The program memory is used to store the program or code. The program or the code memory in this processor is 512 bytes, this is implemented using block RAM (BRAM). The size of the code memory can be extended easily by changing the size of the BRAM and making minor changes in coding. CLK is used to provide clock signal, WE is used to read and write data, a is the 9 bit address input, Data_in is the 32 bit data input to write programs, SPO is the data out port to read programs from this memory.

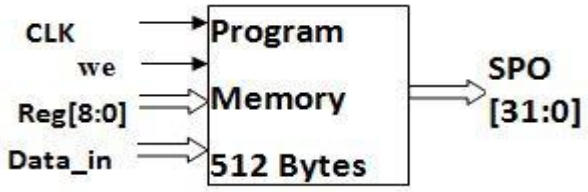

Figure 2: Program Memory

B. Register Set

In this design we have used 32 internal registers, the data word width is 32 bit and the depth is also 32. CLK is used to provide clock signal, Reg_WE is used to read and write data from registers, Reg_a is the 9 bit address input to select register from R0 to R31, only 5 bits are in use, Reg_Data_in is the 32 bit data input to write data on selected register, Reg_SPO is the data out port to read data from selected register.

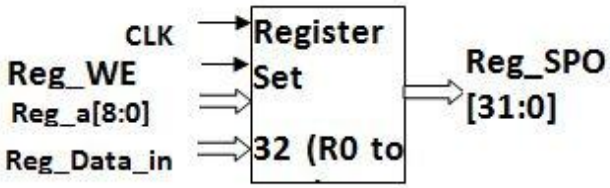

Figure 3: Register Set

C. Arithmetic and Logic Unit

The arithmetic and logic unit is the core of the processor, it performs the eight arithmetic and logic operations as listed in table 2. Addition, subtraction, multiplication operations are performed using inbuilt Xilinx DSP blocks. The block diagram of ALU is shown in figure 4, the input to the ALU unit is 32 bit input ALU_A and ALU_B, the operation to be commenced is selected via ALU_SEL, the output is obtained from ALU_OUT.

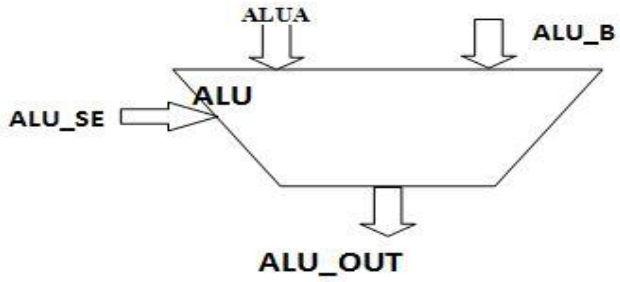

Figure 4: Arithmetic and Logic Unit

Table 2: ALU Operations

S.No	**ALU_SEL [4:0]**	**Operation**
1	00000	Addition
2	00001	Subtraction
3	00010	Logical AND
4	00011	OR
5	00100	XOR
6	00101	XNOR
7	00110	NOR
8	00111	NAND
9	01000	MUL

D. Comparator

The comparator unit is used to perform the 6 comparison operations mentioned in table 3. Whenever any condition is matched then the comparison flag COMP_FLAG is set else it is reset. Xilinx inbuilt comparator is used to perform the task.

Table 3: Comparator Operation

S.No	COMP_SEL	Operation
1	000	A = B
2	001	A > B
3	010	A > = B
4	011	A < B
5	100	A < = B
6	101	A / = B

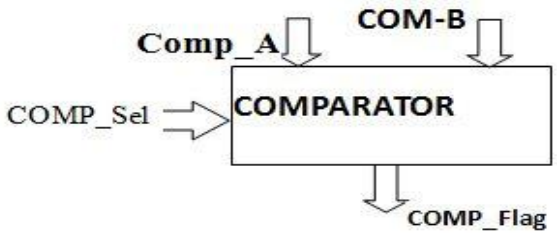

Figure 4: Comparator

E. Shifter

The shifter block performs five operations namely: left shift logical, right shift logical, right shift arithmetic, rotate left and rotate right. Flip flops are used to implement this unit. The shifter unit is used to perform the shifting operations listed in table 4.

Table 4: Shifter Operation

S.No	SHIFTER_SEL	Operation
1	000	LSL: Left Shift Logical
2	001	RSL: Right Shift Logical
3	010	RSA: Right Shift Arithmetic
4	011	RL: Rotate Left
5	100	RR: Rotate Right

F. Timing & Control Unit

The timing and control unit controls all the units discussed so far. This unit generates all the internal control signal to control all units. This block is a designed with a state machine, this state machine, which works on the instruction to be executed. The block diagram of timing and control unit is shown in figure 6.

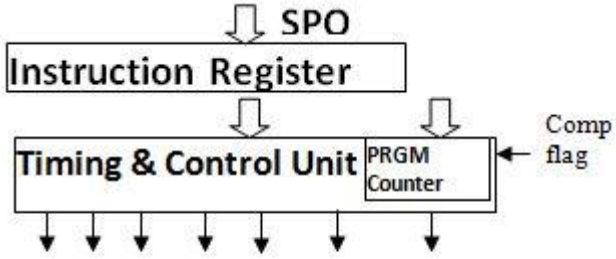

Figure 6: Timing & Control Unit

V. SYNTHESIS

This section discusses the analysis of proposed processor on the basis of resource usage and maximum operating frequency. Table 5 shows the comparison of proposed processor with the two other designs available in literature. As depicted in table 5.1 the number of instructions supported by design in [1] is only 20 and in [2] is 22, the proposed processor supports 25 instructions. So the resource used by our design is slightly greater than [1] and [2]. The maximum operating frequency of proposed processor is lower than [1] but greater than [2], hence proposed design works faster than [2] design. The prime focus of this work is to reduce the latency without changing the delay. The average latency of proposed design is 4 clock cycles including the instruction fetch cycle. The latency of [1] is six clock cycles and that of [2] is 9 clock cycles. So the time to execute one instruction by design [1] is 12.72ns, design [2] takes 22.05ns and proposed design takes only 9.496ns, figure 5.12 shows the bar graph of time to execute one instruction.

Table 5: Synthesis Summary

S.no	Parameter	Processor[1]	Processor[2]	Proposed
1	Slice Registers	238	404	398
2	Slice LUTs	190	335	557
3	BRAM	1	2	1
4	DSPs	1	1	1
5	Max Operating Freq.	470Mhz	407Mhz	421Mhz
6	Delay (ns)	2.12ns	2.45ns	2.374ns
7	Latency	6	9	4
8	Time Taken to execute	12.72ns	22.05ns	9.496ns
9	Numbers of instructions	20	22	25

VI. CONCLUSION

In this work a 32 bit RISC soft processor is developed. The proposed processor architecture supports 25 instructions which includes instruction to transfer data like MVI, MOV, arithmetic instructions like ADD, SUB and MUL, Logical instructions like AND, OR etc., Rotate and Shift instructions like RR, LSL etc, conditional branching instructions like COMPAGB, COMPALB etc. The proposed soft processor design is targeted for FPGA which uses the inbuilt DSP blocks to implement various functions. The inbuilt DSP block has adder, subtractor and multiplier blocks. The designs available in literature supports less instructions and has high latency, the prime focus of this work is to reduce latency of the instructions without changing the delay, the proposed design has almost the same delay (slightly higher) as compared to other designs available in literature. The latency of proposed design is 4 clock cycles as compared to 6 clock cycles of design [1] and 9 clock cycles of design [2], the total time to execute one instruction at maximum operating frequency is 12.72ns for design [1], 22.05 for design [2] and 9.49ns for proposed design. The percentage change to execute one instruction is 25% as compared to design [1] and 56% as compared to design [2].

REFERENCES

[1] Cheah, Hui Yan, et al. "A lean FPGA soft processor built using a DSP block." Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays. ACM, 2015

[2] Cheah, Hui Yan, Suhaib A. Fahmy, and Douglas L. Maskell. "iDEA: A DSP block based FPGA soft processor." Field-Programmable Technology (FPT), 2012 International Conference on. IEEE, 2014.

[3] Cheah, Hui Yan, et al. "The iDEA DSP block-based soft processor for FPGAs." ACM Transactions on Reconfigurable Technology and Systems (TRETS) 7.3 (2014): 19.

[4] Fort, Blair, et al. "A multithreaded soft processor for SoPC area reduction." 2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE, 2006.

[5] S. Chalamalasetti, S. Purohit, M. Margala, and W. Vanderbauwhede. MORA - an architecture and programming model for a resource ecient coarse graine recongurable processor. In NASA/ESA Conf. on Adaptive Hardware and Systems (AHS), pages 389{396, 2009.

[6] X. Chu and J. McAllister. FPGA based soft-core SIMD processing: A MIMO-OFDM xed-complexity sphere decoder case study. In Proc. Int. Conf. on Field Programmable Technology (FPT), pages 479{484, 2010.

[7] X. Chu, J. McAllister, and R. Woods. A pipeline interleaved heterogeneous SIMD soft processor array architecture for MIMO-OFDM detection. In Proc. Int. Symp. on Applied Recongurable Computing (ARC), pages 133{144, 2011.

[8] M. Milford and J. McAllister. An ultra-ne processor for FPGA DSP chip multiprocessors. In Asilomar Conf. on Signals, Systems and Computers, pages 226 {230, 2009.

[9] Xilinx Inc. Virtex-6 FPGA DSP48E1 User Guide, 2011.

[10] P. Yiannacouras, J. Stean, and J. Rose. Application-specic customization of soft processor icroarchitecture. In Proc. ACM/SIGDA Int. Symp. n Field Programmable Gate Arrays (FPGA), pages 201{210, Feb. 2006.

