ICIT 2015 The 7th International Conference on Information Technology
doi:10.15849/icit.2015.0087

Multicore RISC Processor Implementation by VHDL for Educational Purposes

Safaa S. Omran and Ali J. Ibada
Department of computer engineering techniques
College of Electrical and Electronic Engineering Techniques
Baghdad, Iraq
omran_safaa@ymail.com , ali.alshukri@yahoo.com

***Abstract*— With trends computer manufacturers to build computers that have Multicore processors, it becomes necessary to study the hardware architecture of this processor and the way of manage data between Cores. All the previous researches were designing single cycle processors or pipeline processors by FPGA (Field Programmable Gate Array). This is a first research work on parallel processing to design and implement a Multicore processor by FPGA. In this work Multicore processor has two Cores and each Core consists of 5-stage pipeline MIPS (Microprocessor without Interlocked Pipeline Stages) RISC (Reduced Instruction Set Computer) processor. Separated data cache and instruction cache were added to each Core. MESI (Modified, Exclusive, Shared and Invalid) protocol is used to manage cache coherence and memory coherence which support Write-back policy where replacement algorithm is not needed. Many programs are tested on this design and the correct results were obtained. The VHDL (Very high speed integrated circuit Hardware Description Language) of the complete Multicore processor is implemented by using (Xilinx ISE Design Suite 13.4) Software and configured on FPGA Spartan-3AN starter kit and results from the kit were obtained.**

***Keywords*— Multicore; MIPS; RISC; MESI protocol; VHDL; FPGA.**

I. I.INTRODUCTION

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a memory hierarchy, which takes advantage of locality and trade-offs in the cost performance of memory technologies. The principle of locality says that most programs do not access all code or data uniformly. Locality occurs in time (temporal locality) and in space (spatial locality) [1]. The different levels form what is commonly termed the memory hierarchy is a tiered description of how the different levels compare to and interact with each other. The different levels of the memory hierarchy are managed by different parts of the system [2].On modern architectures a main memory access may take hundreds of cycles, so there is a real danger that a processor may spend much of its time just waiting for the memory to respond for requests. To alleviate this problem one or more caches are logically situated between the processor and the memory [3].

To get continuing performance gains of Multicore processor, it is requisite to use parallel software. Most parallel software relies on the shared-memory programming model in which all processors access the same physical address space, this cause cache coherency problem. To address the cache coherency problem, there are many protocols to deal with this [4]. In this paper MESI protocol is used.

Many previous researches have designed single Core (single cycle or pipeline processor) that can execution some instruction of MIPS processor [5-10]. In this work all instructions are designed with extra (*hlt*) instruction that could be used to stop program execution.

VHDL is a VHSIC Hardware Description Language. VHSIC is an abbreviation for Very High Speed Integrated Circuit. It describes the behavior of an electronic circuit or system, such as ASICs (Application Specific Integrated Circuit) and FPGAs as well as conventional digital circuits. A fundamental motivation to use VHDL is that VHDL is a standard, technology/vendor independent language, and is therefore portable and reusable [11]. VHDL has Feature to allow the synthesis of a circuit or system in a programmable device. This paper studies the designing and prototyping of a complete design of Multicore MPIS RISC processor in VHDL. FPGA is a digital integrated circuit that contains configurable (programmable) blocks of logic along with configurable interconnects between these blocks. Design engineers can program such devices to perform a tremendous variety of tasks [12].

II. CACHE MEMORY PRINCIPLES AND DESIGN ELEMENTS

The cache contains a copy of portions of main memory [13]. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor [14]. Each Core in the processor has its own cache and the cache lies on the same chip of the processor as shown in Figure 1. The cache has the following design choices:

doi:10.15849/icit.2015.0087

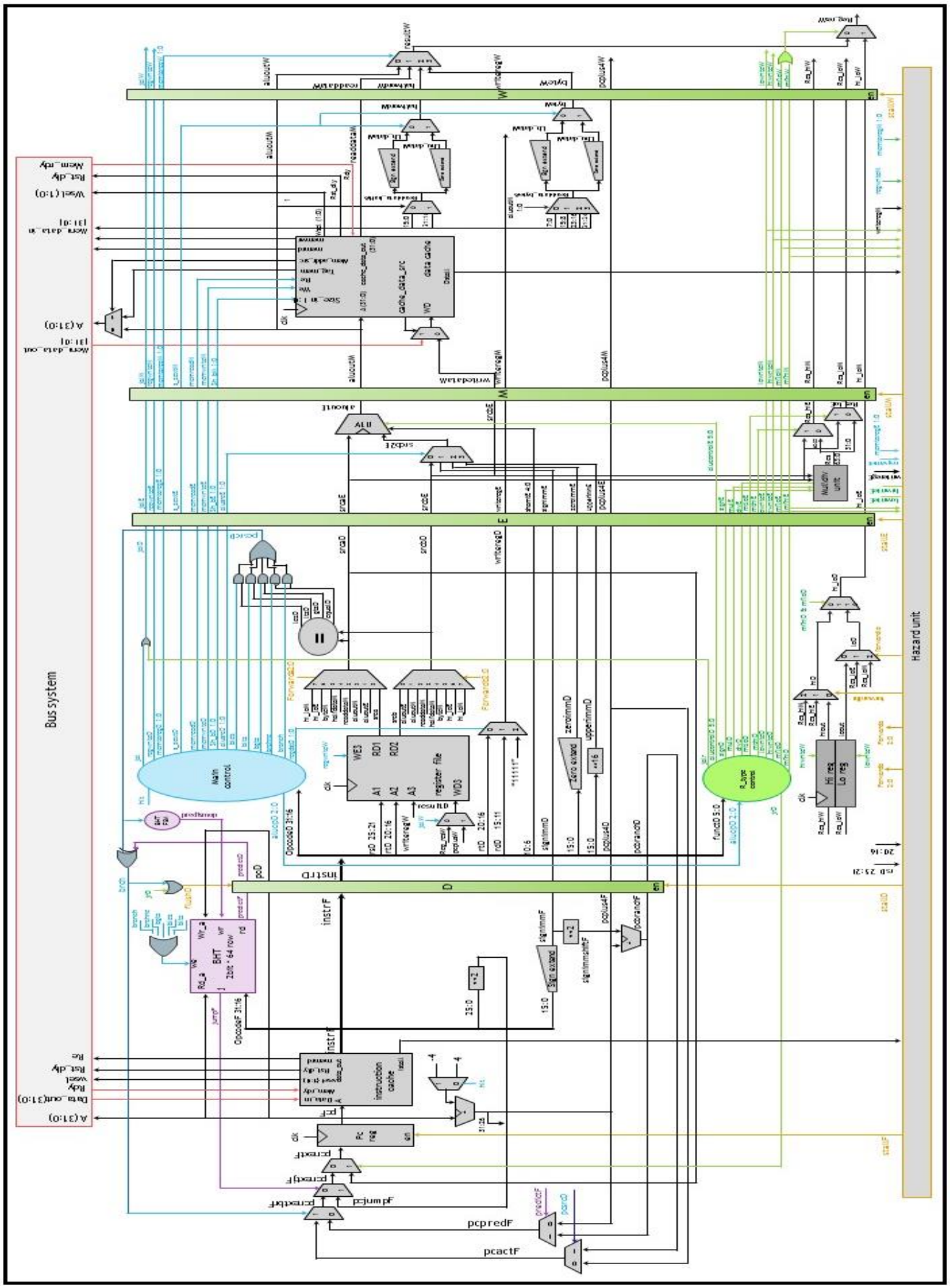

Figure 1 Complete design of one Core with cache memories

doi:10.15849/icit.2015.0087

A. Number of Caches

The cache has been spliced into instruction cache and data cache to avoid structural hazards. These two caches both exist in the CPU (Central Processing Unit), typically as two level one (L1) caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

B. Cache Addresses

The cache memory directly receives physical addresses instead of virtual addresses from the MIPS processor, for this reason there is no need to include a Memory Management Unit (MMU) in this design.

C. Mapping Function

Direct mapping is used in this design to map each block of main memory into only one possible cache line. Direct mapping is picked because there are fewer cache lines than main memory blocks.

D. Write policy

Write policy is needed for data cache only, because the processor will not update the program instruction. Write back policy is used in this design to minimize memory writes, where a copy of the data is written to data cache by the processor and not to main memory. When new data is written to cache, a MESI state is change to M (Modified) bits associated with the line is set. Then, when a block is replaced, it is written back to main memory and MESI state get the new line state.

E. Cache Size

For this design, each Core has cache size of 64 bytes, organized as 4 lines, each line has 4 words, and each word is 4 bytes in length. Therefore, main memory address is organized as shown in Figure 2.

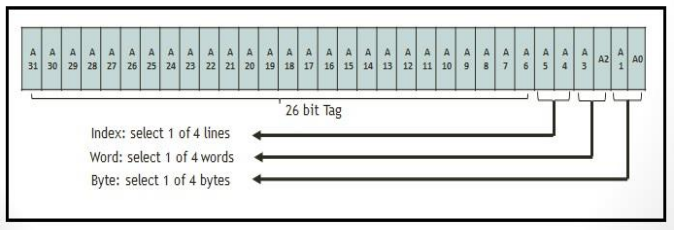

Figure 2 Address bit field format

III. CACHE COHERENCY MECHANISMS

To implement cache coherency protocols in the Multicore system and managed the consistency of memory, cache coherency mechanism is used. In this work Snooping Based Coherency is used as a cache coherency mechanism. It allows each cache to monitor the address lines so that to gain access to main memory which they have cached. Any activity on cache line will trigger message, which will be broadcasted to all the caches to update the cache line with the activity.

IV. CACHE COHERENCE PROTOCOL

In Multicore systems, coherence must occur inside each core and among cores through bus system. For this design MESI protocol is chosen. It is one of the mostly used cache coherency protocol. Any cache line can be in one of 4 states (2 bits):

1) Modified (00): cache line has been modified from the value in the main memory.
2) Exclusive (01): cache line is the same as main memory and is the only cached copy.
3) Shared (10): Same as main memory but copies exist in other caches.
4) Invalid (11): Line data is not valid (as in simple cache). So it should not be used.

A state transition diagram in Figure 3 shows what happens to a cache line in a processor as a result of memory accesses made by that processor (read hit/miss, write hit/miss). Memory accesses made by other processors that result in bus transactions observed by this snoopy cache (Mem RD, Mem WR, Invalidate) as shown in Figure 4.

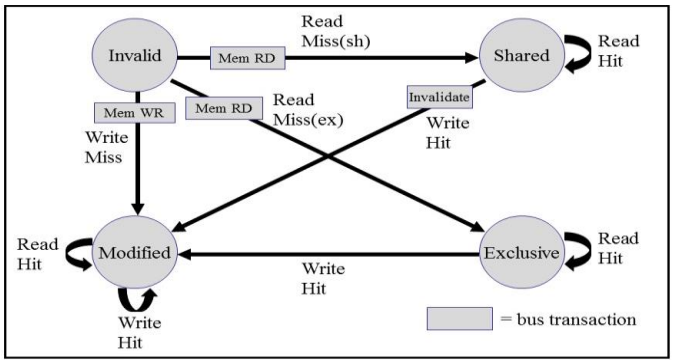

Figure 3 MESI – locally initiated accesses

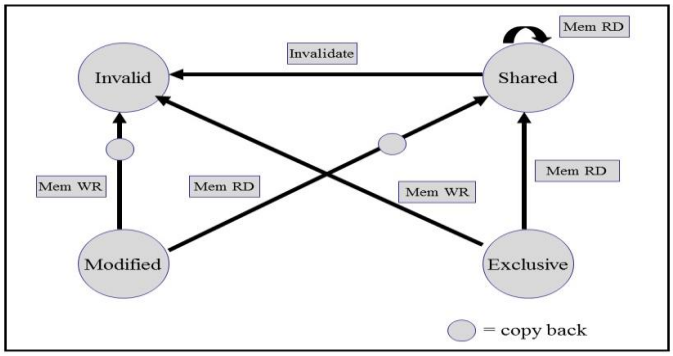

Figure 4 MESI – remotely initiated accesses

V. CACHE CONTROLLER

Cache controller is used to regulate cache memory. When Core wants to access memory location for read or write, it is first send address to its cache controller which decides this address is exists in tag cache or not. If it is, then no memory access is needed, the data is provided to Core directly from its cache; if not, then the cache controller fetches several words from main memory consecutively to fill the corresponding line

doi:10.15849/icit.2015.0087

in the cache. Data cache controller use two bits for MES protocol while instruction cache controller use one bit (valid o not valid only), because instruction cache does not make an change to instruction program. Also data cache controller ha extra control signals to manage write-back that requested fron bus system when a Core need to access a data that modified i another Core's cache, however instruction cache controller doe not have this signals because there is no write-back in instruction cache. The cache controller consists of:

1) Finite State Machine (FSM): the FSM of data cache differs from that of instruction cache because data is accessed for read or write while instructions are executed without modification. FSM of data cache is shown in Figure 5.

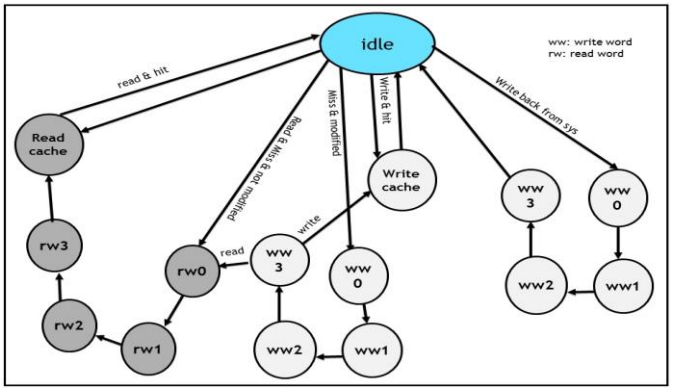

Figure 5 Data cache FSM

Write back from system has the priority to execute if read or write happen at the same time with system write back. Table 1 explains FSM work, and table 2 describes the function of FSM.

TABLE I. DATA CACHE FSM TRUTH TABLE

state	inputs					outputs									
	Hit	read	write	wb_in	MESI	stall	cachewr	cacherd	memrd	memwr	Cache_data_src	Mem_addr_src	Rst_dly	wsel	wb_done_out
idle (st0)	x	0	0	0	xx	1	0	0	0	0	x	x	1	00	1
Write cache (st1)	x	0	1	0	/= 00	1	1	0	0	0	0	x	1	00	1
Read cache (st2)	1	1	0	0	xx	1	0	1	0	0	x	x	1	00	1
WW 0 (st3)	0	1 or 1		0	00	0	0	1	0	1	x	1	0	00	1
WW 1 (st4)	0	1 or 1		0	00	0	0	1	0	1	x	1	0	01	1
WW 2 (st5)	0	1 or 1		0	00	0	0	1	0	1	x	1	0	10	1
WW 3 (st6)	0	1 or 1		0	00	0	0	1	0	1	x	1	0	11	1
RW 0 (st7)	0	1	0	0	/= 00	0	1	0	1	0	1	0	0	00	1
RW 1 (st8)	0	1	0	0	/= 00	0	1	0	1	0	1	0	0	01	1
RW 2 (st9)	0	1	0	0	/= 00	0	1	0	1	0	1	0	0	10	1
RW 3 (st10)	0	1	0	0	/= 00	0	1	0	1	0	1	0	0	11	1
WW 0 (st11)	x	x	x	1	xx	0	0	1	0	1	x	1	0	00	0
WW 1 (st12)	x	x	x	1	xx	0	0	1	0	1	x	1	0	01	0
WW 2 (st13)	x	x	x	1	xx	0	0	1	0	1	x	1	0	10	0
WW 3 (st14)	x	x	x	1	xx	0	0	1	0	1	x	1	0	11	1

TABLE II. FUNCTION OF FSM SIGNALS

Signal name	Signal value	Signal effect
stall	0	Main memory is accessed and the whole pipeline is stalled.
	1	Cache memory is accessed and the pipelined registers are captured on the next falling edge.
cachewr	0	None
	1	When cache hit occurs, data supplied by the processor is written into cache memory.
cacherd	0	None
	1	When cache hit occurs, data is supplied to the processor from cache memory.
memrd	0	
	1	When cache miss occurs, data is supplied to the cache memory from main memory.
memwr	0	
	1	When cache miss occurs and dirty bit is set, data block which is supplied by cache memory is written into main memory.
Cache_data_src	0	The value fed to the cache_data_in input of cache memory comes from the processor.
	1	The value fed to the cache_data_in input of cache memory comes from main memory.
Mem_addr_src	0	The address fed to the amem input of main memory comes from the processor.
	1	The address fed to the amem input of main memory equals to (tag & I & 0).
Rst_dly	0	The address fed to the amem input of main memory equals to (tag & I & 0).
	1	There is no main memory activity.
wsel	00	The first (least significant) word of memory block is selected.
	01	The second word of memory block is selected.
	10	The third word of memory block is selected.
	11	The fourth (most significant) word of memory block is selected.
wb_done_out	0	Cache controller is responding to write back request from bus system and Core is stall.
	1	Write back is done by cache controller and Core work properly.

FSM of instruction cache is part of FSM data cache that does not contain the states performing write actions and write back system. Figure 6 shows instruction cache FSM, and table 3 explains its work.

doi:10.15849/icit.2015.0087

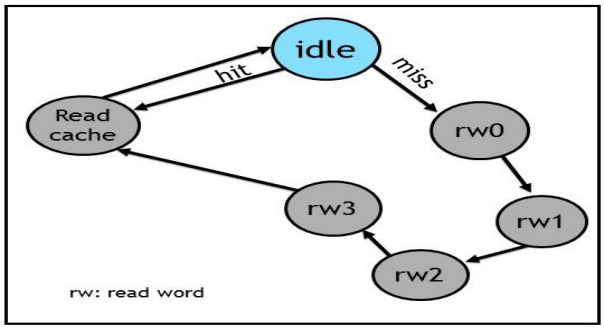

Figure 6 Instruction cache FSM

TABLE III. INSTRUCTION CACHE FSM TRUTH TABLE

state	inputs		outputs					
	Hit	Mem_rdy	stall	cachewr	cacherd	memrd	Rst_dly	wsel
idle (st0)	x	x	1	0	0	0	1	00
Read cache (st1)	1	1	1	0	1	0	1	00
Rw 0 (st2)	0	1	0	1	0	1	0	00
Rw 1 (st3)	0	0	0	1	0	1	0	01
Rw 2 (st4)	0	0	0	1	0	1	0	10
Rw3 (st5)	0	0	0	1	0	1	0	11

2) Tag cache: data tag cache contains 26 tag bits, 2 bits for MESI protocol for each data cache line. Tag bits are used for holding the 26 most significant bits of the address being accessed. MESI bits are reset when the machine restarts. Instruction tag cache is similar to data tag but does not have 2 bits for MESI protocol, instead it has 1 bit to indicate the line valid or not (valid bit).

VI. COMPLETE CACHE DESIGN AND MEMORY SYSTEM

For each Core, data cache controller is combined with its data cache as shown in Figure 7, while Figure 8 shows instruction cache controller that is combined with instruction cache.

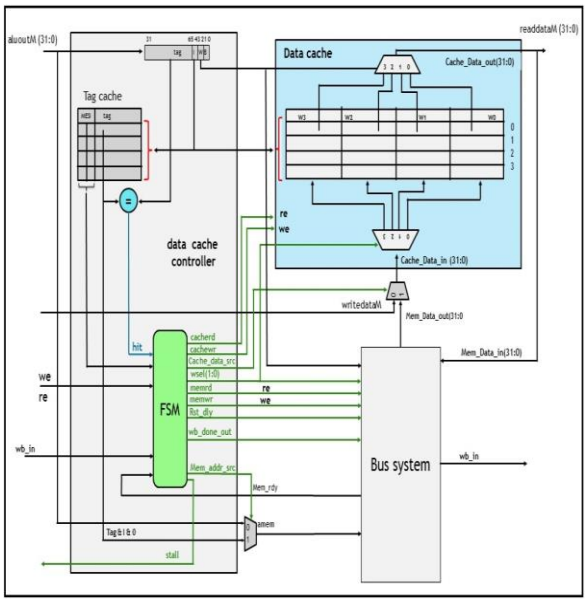

Figure 7 Complete design of data cache

Both caches access the main memory that consists of 1 kilobyte, arranged as 2 segments each one has 512 bytes; one segment for data and the other for instruction, each segment has 32 blocks, each block consists of 4 words and each word contains 4 bytes. Figure 9 shows main memory.

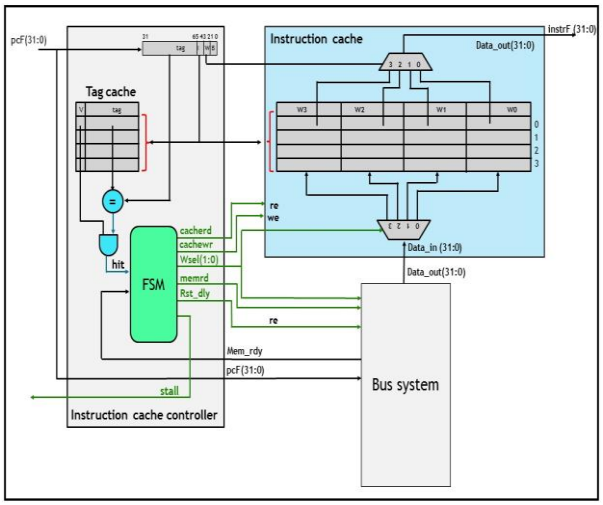

Figure 8 Complete design of instruction cache

doi:10.15849/icit.2015.0087

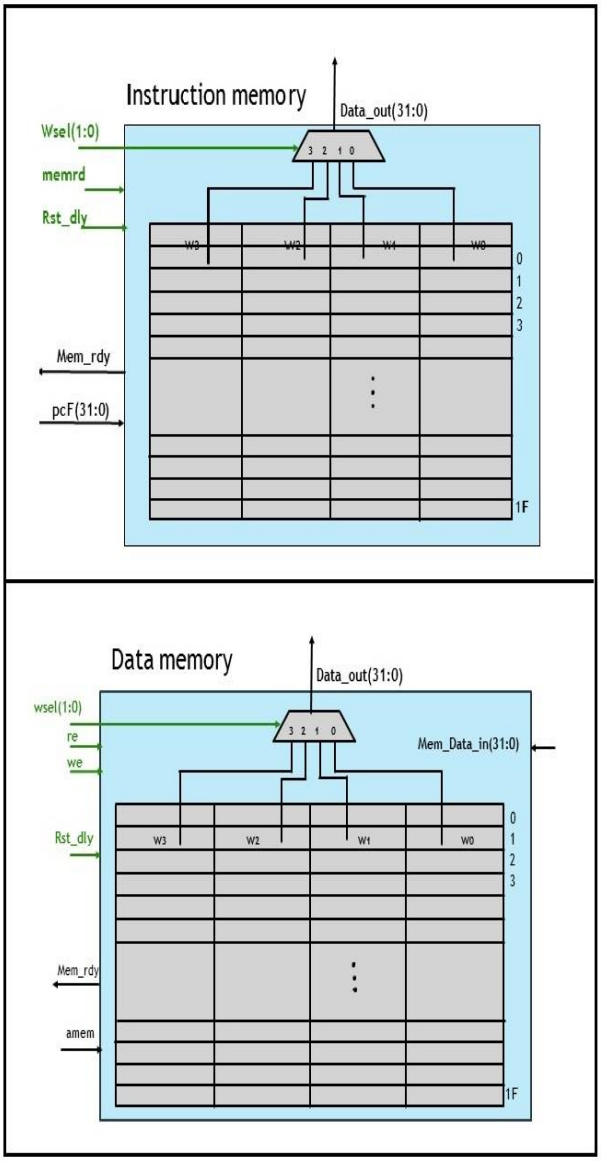

Figure 9 Main memory

VII. VHDL TOP- LEVEL IMPLEMENTATION

Top level of Multicore processor connects two Cores to data and instruction memories through bus system as shown in Figure 10. Later a test bench is written and used to execute a program.

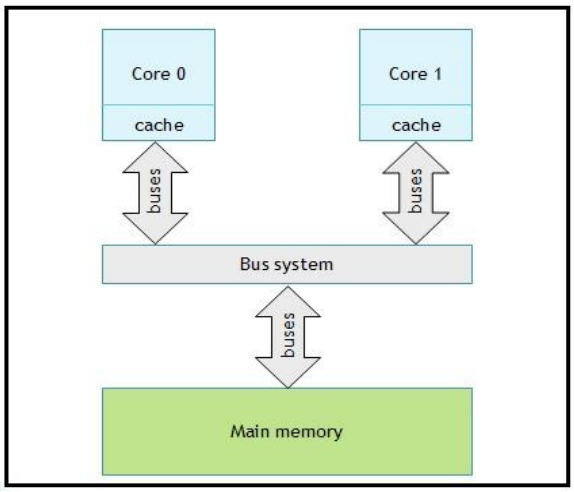

Figure 10 Multicore processor system

VIII. RESULTS

The test program shown in Figure 11 is stored in main memory. This program can be executed as a parallel code to get profit of Multicore system. This program used to find the summation of numbers $(1 - 10)_h$ plus factorial of number 7. The results should be $(00001438)_h$ stored in memory location $(40)_h$ and $(00000000)_h$ stored in memory location $(44)_h$.

	Assembly	address	discretion	machine
	addi $t0,$0,10	0	$t0 = 10h	20080010
	addi $t6,$0,0	4	$t6 = 0h	200E0000
loop:	add $t6,$t6,$t0	8	$t6 = $t6 + $t0	01C87020
	subi $t0,$t0,1	c	$t0 = $t0 - 1	2108FFFF
	bne $t0,$0,loop	10	if ($t0 != $zero) goto loop	1408FFFD
	sw $t6,40($0)	14	mem[$zero + 64] = $t6	AC0E0040
	addi $t3,$0,7	18	$t3 = 7	200B0007
	addi $a0,$0,1	1c	$a0 = 1	20040001
loop2:	mult $t3,$a0	20	$hi , low = ($t3 * $a0)	01640018
	mflo $a0	24	$a0 = $lo	00002012
	mfhi $a1	28	$a1 = $hi	00002810
	subi $t3,$t3,1	2c	$t3 = $t3 -1	216BFFFF
	bne $t3,$0,loop2	30	if ($t3 != $zero) goto loop2	140BFFFB
	lw $v0,40($0)	34	$v0 = mem[$zero + 64]	8C020040
	add $a0,$a0,$v0	38	$a0 = $a0 + $v0	00822020
	sw $a0,40($0)	3c	mem[$zero + 64] = $a0	AC040040
	sw $a1,44($0)	40	mem[$zero + 68] = $a1	AC050044
	hlt	44	stop program execution	F00000000

Figure 11 Top level test program

This program has been executed as a parallel code in Multicore processor. By using VHDL testbench, the right results have been gotten as shown in Figure 12 which indicates the correctness of the design. When *memwrite* signal is 1, the results are stored in data memory.

The program shown in Figure 11 is executed in single core system and Multicore system to make a comparison in terms of

doi:10.15849/icit.2015.0087

performance and speedup between single Core processor and Multicore processor as shown in table 4. The CPI (Clock Per Instruction) metric is calculated by using equation:

$$Program\ Execution\ time = Instruction\ count \times CPI \times Clock\ period \quad \ldots\ldots\ldots (1)$$

TABLE IV. PERFORMANCE COMPARISON BETWEEN SINGLE CORE AND MULTICORE PROCESSORS

Processor	Instruction count	Program execution time	No of clock cycles	Clock period	CPI	Speedup
Single Core	92	1255	125.5	10 ns	1.36	1
Multicore	92	865	86.5	10 ns	0.94	1.45

This design is configured on Xilinx Spartan-3AN starter kit FPGA. To show all results, VGA (Video Graphic Array) screen is interfaced with FPGA via a standard high-density HD-DB15 female connector VGA display port and driving the VGA monitor in 640 by 480 mode. Figure 13 shows results of test program on VGA screen. The left column is an assembly test program machine code with its locations in instruction memory that would the processor fetches it to be execution. Drawing in the center is illustration that the Processor is connected to data and instruction memories via buses. The Right column represents the data memory that would the processor uses it to store or load data, results of test program are shown in data column with its locations.

IX. CONCLUSIONS

VHDL design of Multicore RISC processor has been implemented for whole instructions which consist of 49 instructions. Also hlt instruction was added to stop program execution. Each Core was Pipelined to five stages. MESI protocol was used to deal with data coherence which represents the main problem of Multicore system. On chip cache system was added for each Core. Cache system used direct mapping function, write back policy. The cache system consists of two separated caches; one for data and one for instruction. After all system design was completed, various programs simulated and results were obtained. It is meaning that design work properly. The Xilinx ISE Design Suite 13.4 program is used for design synthesis while the Xilinx ISim simulator program is used to simulate this design which is then configured on a Xilinx Spartan-3AN FPGA starter kit and results from kit were obtained.

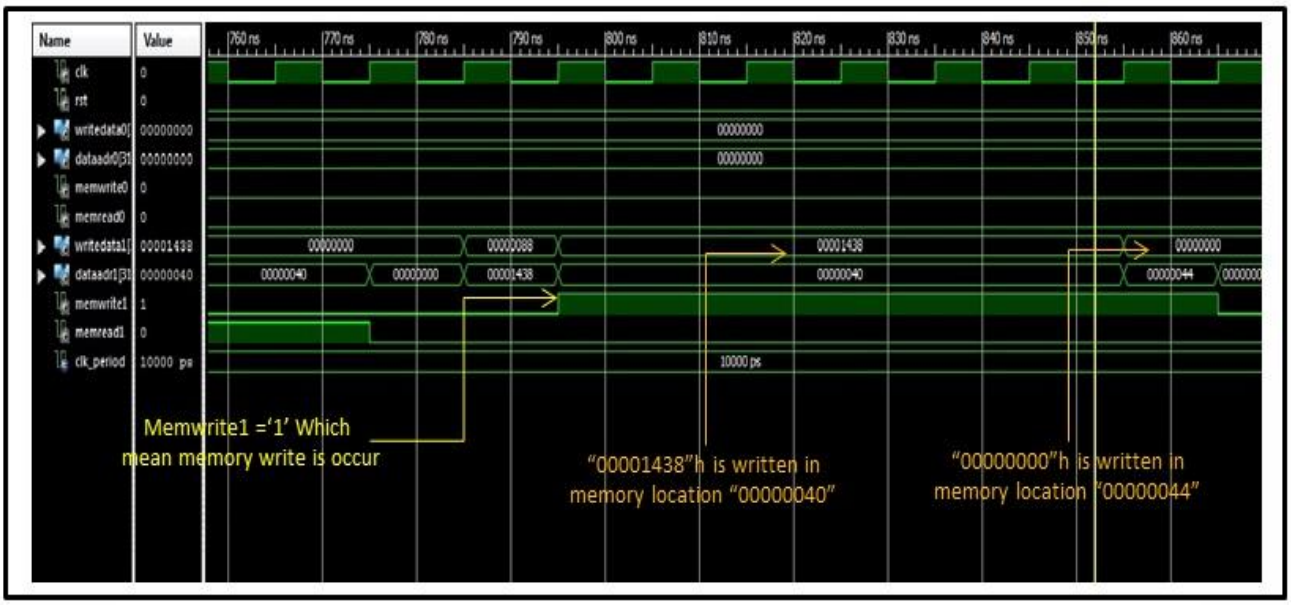

Figure 12 Simulation waveform of test program

doi:10.15849/icit.2015.0087

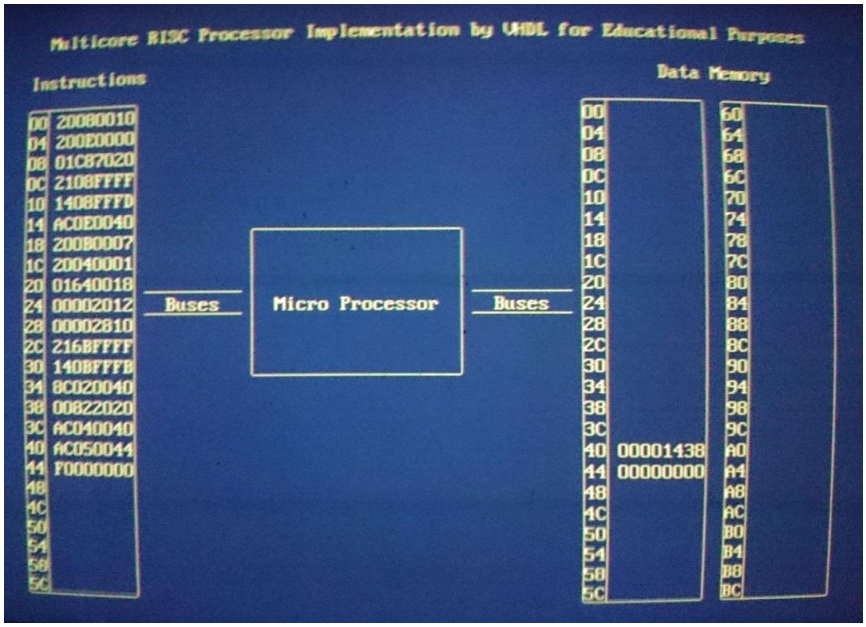

Figure 13 Results of test program on VGA screen

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, "*Computer Architecture: A Quantitative Approach"*, 5th ed., San Francisco, USA: Morgan Kaufmann, 2012.

[2] D. Page, "A Practical Introduction to Computer Architecture", London, UK: Springer- Verlag, 2009.

[3] M. Herlihy and N. Shavit, "The Art of Multiprocessor Programming", Burlington, USA, Morgan Kaufmann, 2008.

[4] S. Dey, and M. S. Nair, "Design and Implementation of a Simple Cache Simulator in Java to Investigate MESI and MOESI Coherency Protocols", International Journal of Computer Applications, Vol. 87 – No.11, 0975 – 8887, 2014.

[5] M. B. I. Reaz, Sh. Islam and M. S. Sulaiman, "A Single Clock Cycle MIPS RISC Processor Design using VHDL", IEEE International Conference on Semiconductor Electronics (ICSE2002), Penang, Malaysia, PP. 126 – 129, DEC. 2002.

[6] S. P. Katke and G. P. Jain, "Design and Implementation of 5 Stages Pipelined Architecture in 32 Bit RISC Processor", International Journal of Emerging Technology and Advanced Engineering, vol. 2, no. 4, PP. 340-346, Apr. 2012.

[7] V. Robio, "A FPGA Implementation of A MIPS RISC Processor for Computer Architecture Education", MSc. thesis, New Mexico State University, Las Cruses, New Mexico, America, 2004.

[8] B. valli, A. U. Kumar and B. V. Bhaskar, "FPGA Implementation and Functional Verification of a Pipelined MIPS Processor", International Journal Of Computational Engineering Research, Vol. 2, No. 5, PP. 1559-1561, Sep. 2012.

[9] I. Anthony, "VHDL Implementation of Pipelined DLX Microprocessor", MSc. Thesis, University Teknologi Malaysia (UTM), Malaysia, 2008.

[10] H. Mahmood and S. omran, "Pipelined MIPS Processor with Cache Controller using VHDL Implementation for Educational Purposes", International Conference on Electrical Communication, Computer, Power, and Control Engineering ICECCPCE1, Mosul, Iraq, 2013.

[11] Pedroni V., "circuit design with VHDL", MIT Press, London, England, 2004.

[12] C. Maxfeild, The Design Warrior's Guide to FPGAs: Devices, Tools and Flows, Burlington, USA: Elsevier, 2004.

[13] M. Abd-El-Barr and H. El-Rewini, Fundamentals of Computer Organization and Architecture, New Jersey, USA: John Wiley& Sons, 2005.

[14] W. Stallings, Computer Organization and Architecture: Designing for Performance, 8th ed., New Jersey, USA: Pearson Education, 2010.