International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-3, Issue-5, May 2017
ISSN: 2395-3470
www.ijseas.com

# Implementation of a pipelined MIPS CPU with single cycle

**S.G.Nafreen Sultana [1], K.Sudhakar[2] K.PrasadBabu[3] S.Ahmed Basha[4]**
[1] 15G31D0610 M.Tech DSCE, Sjcet, Yerrakota, Kurnool, Andhra Pradesh India
[2] HOD & Associate Professor, Dept of ECE, Sjcet, Yerrakota, Kurnool, Andhra Pradesh India
[3]Assistant Professor, Dept of ECE, Sjcet, Yerrakota, Kurnool, Andhra Pradesh India
[4]Associate Professor, Dept of ECE, Sjcet, Yerrakota, Kurnool, Andhra Pradesh India

## Abstract
In this paper we are going to design the 5 stage pipeline processor with single cycle. The design starts with an ALU, register file and SRAM driver. To allow for the five stages (fetch, decode, execute, memory access, write back) several control blocks are to be implemented to enable communication between the ALU, register file and SRAM driver. To ensure single cycle instructions, the controls used only combinational logic. Each instruction needs to be decoded according to MIPS formatting and provide a subset of the inputs for the control blocks which then decide the datapath. Most of the control logic utilized XOR logic for equality, basic logical ANDs and ORs, and several MUXs to change the datapath depending on the instruction. For the pipeline CPU we start with working single cycle MIPS CPU implementation. To begin we develop the pipelined datapath by separating the modules and control logic into the 5 MIPS pipeline phases: fetch, decode, execute, memory access, write back.

Keywords: ***CPU, MIPS, DataPath, SRAM, Pipelining.***

## 1. INTRODUCTION
One of the most effective ways to speed up a digital design is to use pipelining. The processor can be divided into subparts, where each part may execute in one clock cycle. This implies that it is possible to increase the clock frequency compared to a non-pipelined design. It will also be easier to optimise each stage than trying to optimise the whole design. While the instruction throughput increases, instruction latency is added. The processor is implemented using Harvard Architecture consisting of separate Instruction and Data Memories. The main motivation behind pipelining the processor is to increase the throughput.

The various stages are as follows:
1. Instruction Fetch, instructions are fetched from the instruction memory.
2. Instruction Decode, instructions are decoded and control signals are generated.
3. Execute, arithmetic and logic instructions are executed.
4. Memory access, memory is accessed on load and store instructions.
5. Write back, the result is written back to the appropriate register.

Pipeline hazards :

In some cases the next instruction cannot execute in the following clock cycle. These events are called hazards. In this design there are three types of hazards.

1. Structural hazards:

Though the MIPS instruction set was designed to be pipelined, it does not solve the structural limitation of the design. If only one memory is used it will be impossible to solve a store or load instruction without stalling the pipeline. This is because a new instruction is fetched from the memory every clock cycle, and it is not possible to access the memory twice during a clock cycle.

2. Control hazards:

Control hazards arise from the need to make a decision based on the results of one instruction while others are executing. This applies to the branch instruction. If it is not possible to solve the branch in the second stage, we will need to stall the pipeline. One solution to this problem is branch prediction, where one actually guess, based on statistics, if a branch is to be taken or not. In the MIPS architecture delayed decision

was used. A delayed branch always executes the next sequential instruction following the branch instruction. This is normally solved by the assembler, which will rearrange the code and insert an instruction that is not affected by the branch. The assembler made for this project does not support code reordering, it has to be done manually.

3. Data Hazards:

If an instruction depends on the result of a previous instruction still in the pipeline, we will have a data hazard. These dependencies are too common to expect the compilers to be able avoid this problem. A solution is to get the result from the pipeline before it reaches the write back stage. This solution is called forwarding or bypassing.

Dealing with the hazards

1. Using two memories solves the structural hazard. One for instructions and one for data. Normally only one memory is used in a system. In that case separate instruction and data caches can be used to solve the structural hazard. In this project only one memory was available and because no caches were implemented, the processor is stalled for each load and store instruction.

2. Using delayed decision solves the control hazards.

3. Forwarding solves the data hazards. Still it will not be possible to combine a load instruction and an instruction that reads its result. This is due to the pipeline design and a hazard detection unit will stall the pipeline one cycle.

## 2. PROBLEM DEFINITION

The previous MIPS implementation in block diagram is shown below figure

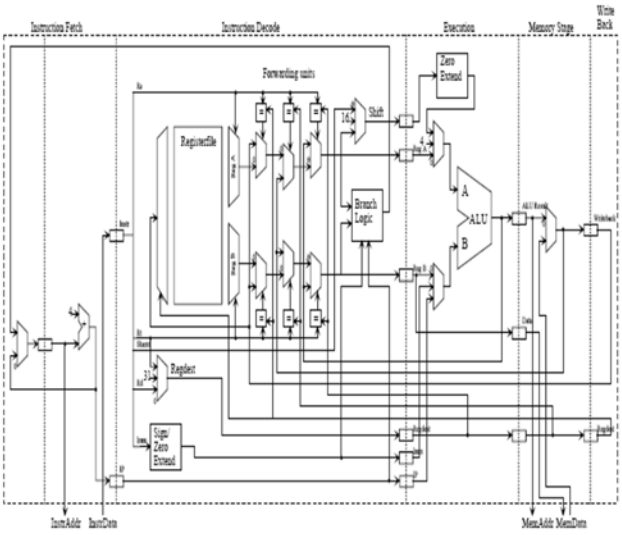

**Figure 1 Pipelined processor**

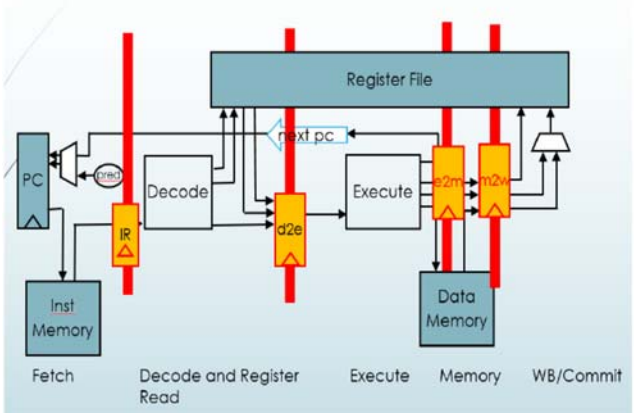


**Figure 2 Five stage pipelined processor**

### Single-Cycle Implementation

A single cycle Implementation means that all the operations take equal amount of time. There are many instructions in a CPU. Each instruction might take different amount of time. But in a Single Cycle, all the operations take equal amount of time which is completed in on clock cycle. So the question arises about how the clock cycle is determined? There might be 'n' number of instructions. But the clock cycle is determined by the time taken by the slowest instruction. Usually the slowest instruction is load word. Other instructions might be executed before the clock tick. In that case the instruction will just

International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-3, Issue-5, May 2017
ISSN: 2395-3470
www.ijseas.com

have to wait for the next clock, to start executing.

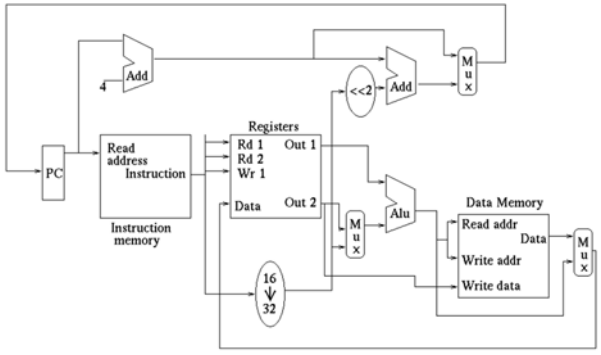


**Figure 3 Single-Cycle CPU**

The above figure depicts the data path for single-cycle implementation of CPU. It uses I-Type branch instruction. The read address for the Instruction Memory is generated from program counter. The output of Instruction Memory is divided and a part of it is sent to registers module, to determine the data which needed to be accessed and the other part is sent to sign extend which extends the 16-bit data to 32-bit data. The output of the registers module, which is the data selected based on output from Instruction Memory, is sent to ALU, where all the arithmetic operations are performed based on the op-code. The result from the ALU is sent to Data Memory, which writes the result to the memory.

Coming to PC, it depends on the comparison of the operands. If operands are eq, then PC is incremented by 4 and also the value of offset*4 is added to it. Otherwise just the PC is incremented by 4 and the offset is not taken

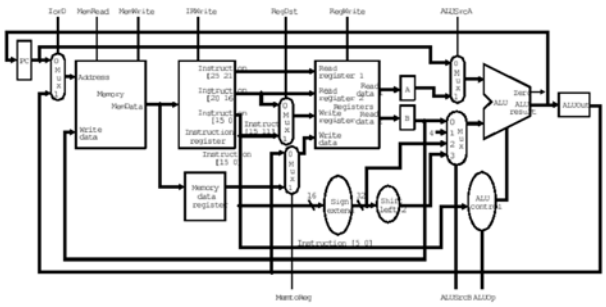

**Figure 4 Multi-Cycle CPU**

The above figure depicts the multi cycle data path. After looking at the datapath of multi-cycle CPU, we can notice few differences with the single-cycle, the most prominent one is the addition of extra registers A, B etc. There is only one ALU, which is also used for incrementing PC by 4 followed by ALUout register, as compared to single, which has an adder as well. PC+4 is executed in the first clock cycle. Then the ALU is used again for another operation. But we may need PC+4 at a later stage, hence to keep ALU from loosing the information, we make use of these registers. The various stages followed here are:

Instruction Fetch: PC is incremented by 4 and loaded to PC. And Instruction Register is loaded with instruction at PC.

Instruction Decode: The values from register module are loaded into A and B and also ALUout is loaded with the target address.

Execute: ALU operation takes place and it is loaded into ALUout. Again there are 2 types of instructions in this. The regular ALU operations and Branch equal to (Beq) operations. During arithmetic operations, the operation is done and the result is loaded to ALUout. In case of Beq, the data at A and B are subtracted. If the result is 0, the the value which is at ALUout is loaded into PC. In this case, process is done and we return again to Instruction Fetch.

Memory: Here again there can be 3 steps load, store and arithmetic. If load is going on, the data at the address of ALUout, is loaded into Memory Data Register. If store operation is going on, the dta at register B is loaded into memory at ALUout address. If arithmetic operation is going on, the value in ALUout is written into register module.In the store and arithmetic cases, we return ti 1st step Instruction Fetch.

WriteBack: Here load instruction takes place. The value in memory data register is written into register. The process is completed here and we return to 1st step Instruction Fetch again

International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-3, Issue-5, May 2017
ISSN: 2395-3470
www.ijseas.com

## 4.1 RESULTS

Below figures represents the individual RTL view of each module and the simulation of module.

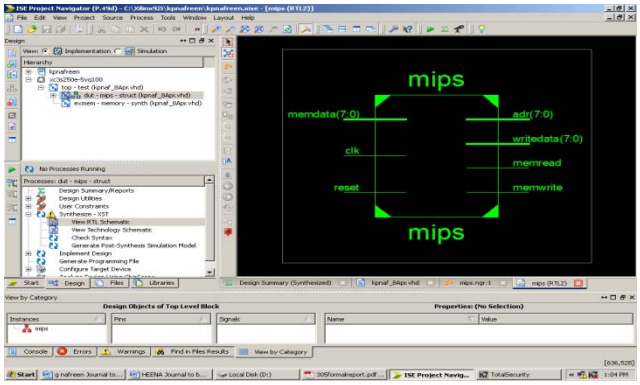


**Figure 5 RTL view of MIPS**

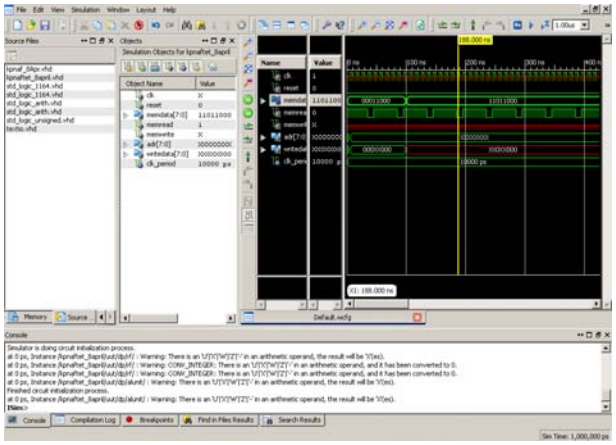

**Figure 6 Simulation wave form of MIPS.**

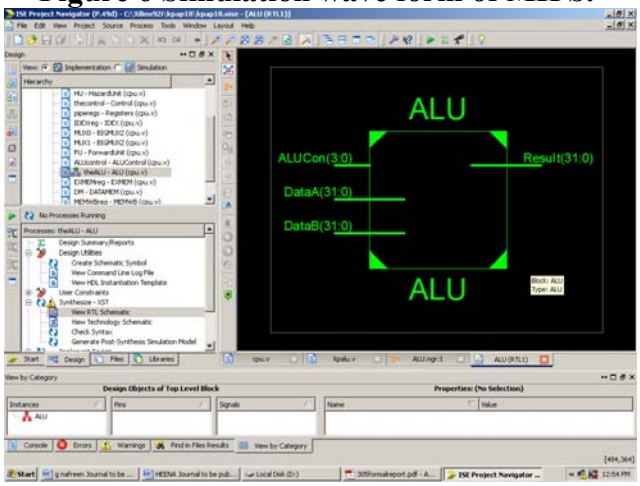


**Figure 7 RTL view of ALU**

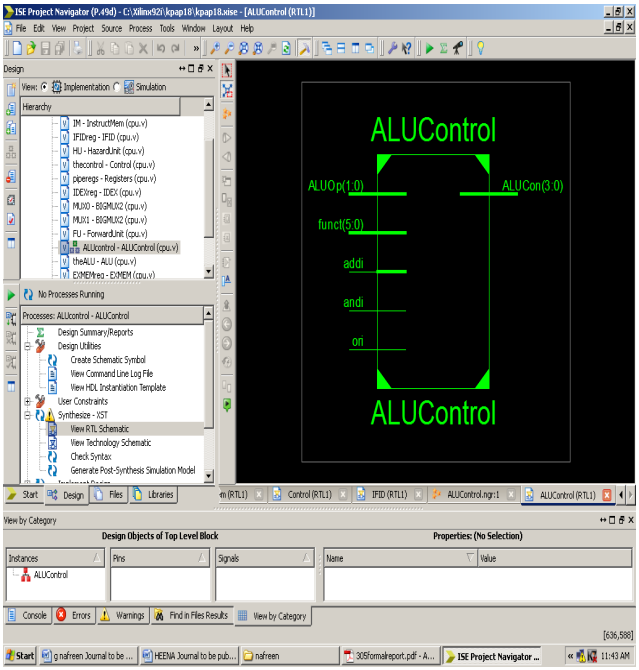


**Figure 8 RTL view of ALUControl**

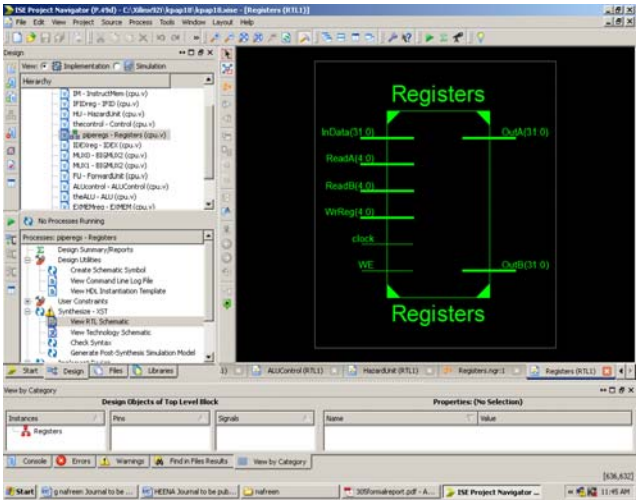


**Figure 9 RTL view of Registers**

International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-3, Issue-5, May 2017
ISSN: 2395-3470
www.ijseas.com

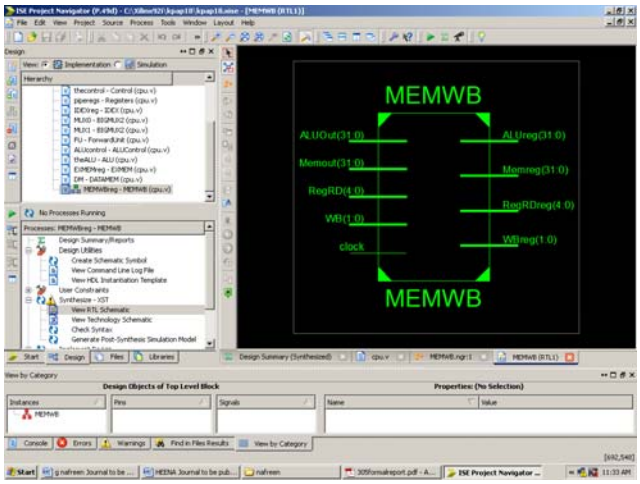


Figure 10 RTL view of Memory Write Back

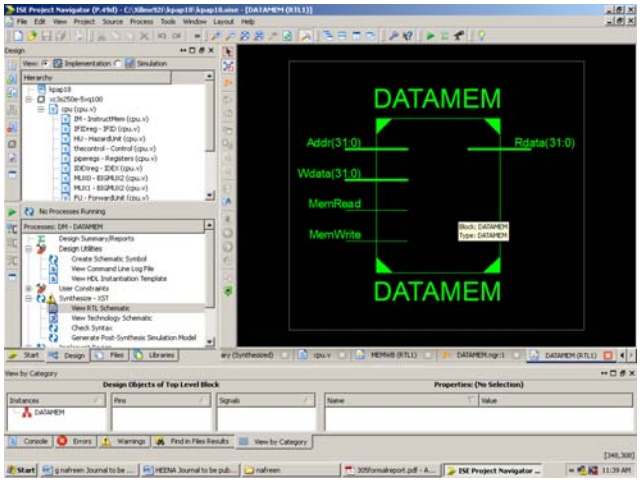


Figure 11 RTL view of Datamemory

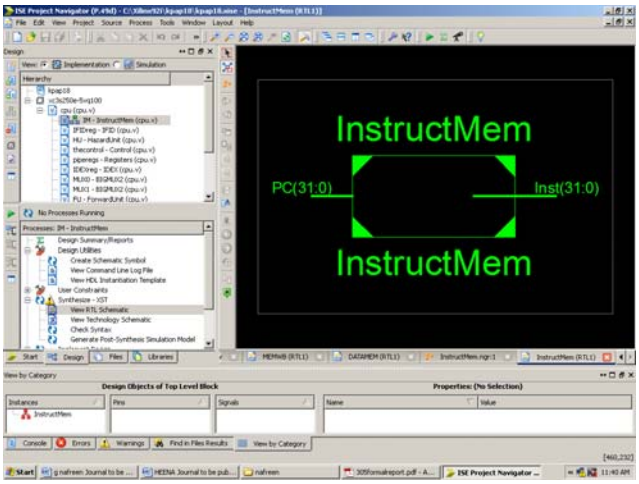


Figure 12 RTL view of Instruct memory

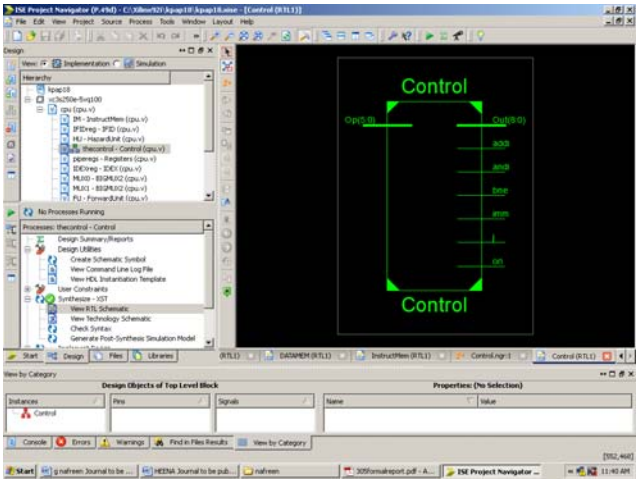


Figure 13 RTL view of Control

*International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-3, Issue-5, May 2017*
*ISSN: 2395-3470*
*www.ijseas.com*

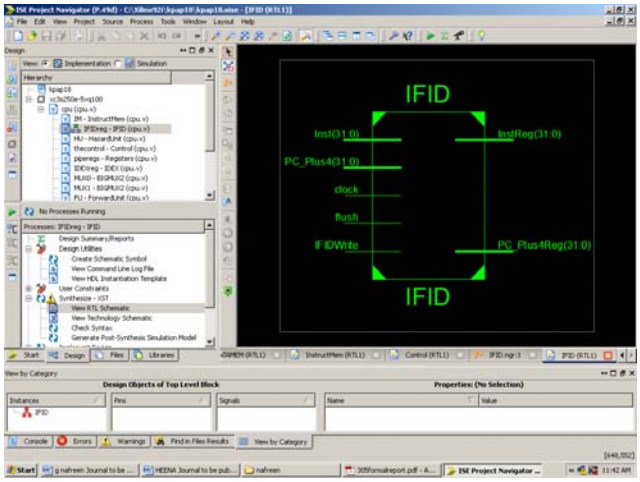


**Figure 14 RTL view of IFID**

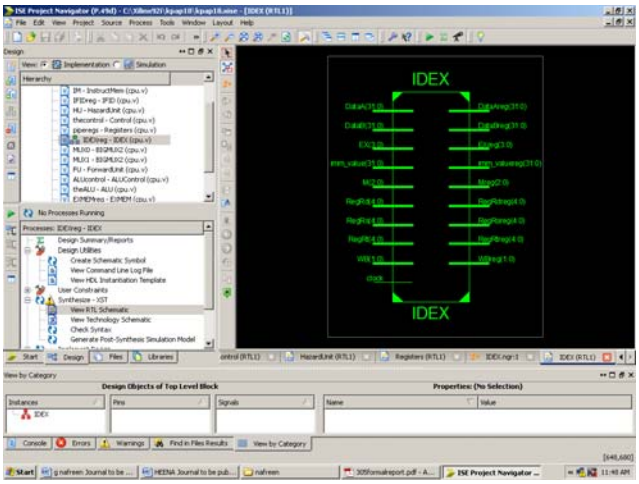


**Figure 16 RTL view of Instruction Decode & Execute unit**

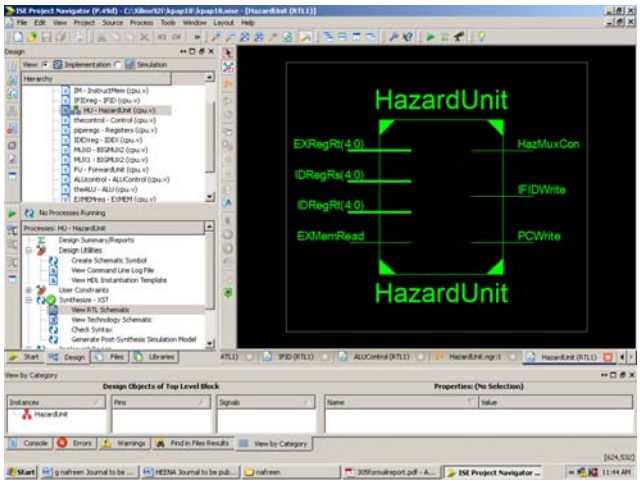


**Figure 15 RTL view of HazardUnit**

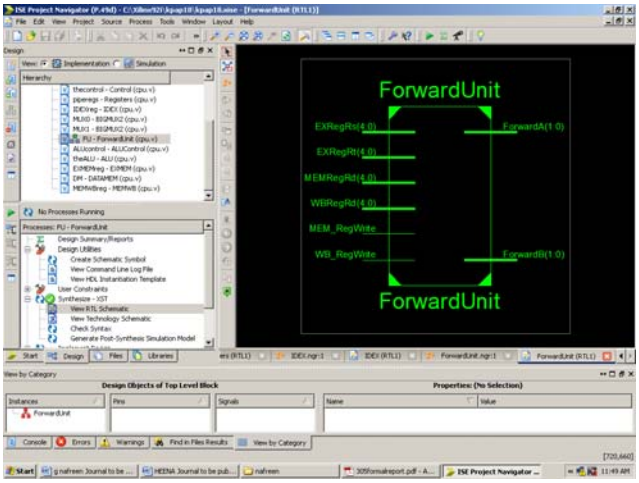


**Figure 17 RTL view of Forward Unit**

International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-3, Issue-5, May 2017
ISSN: 2395-3470
www.ijseas.com

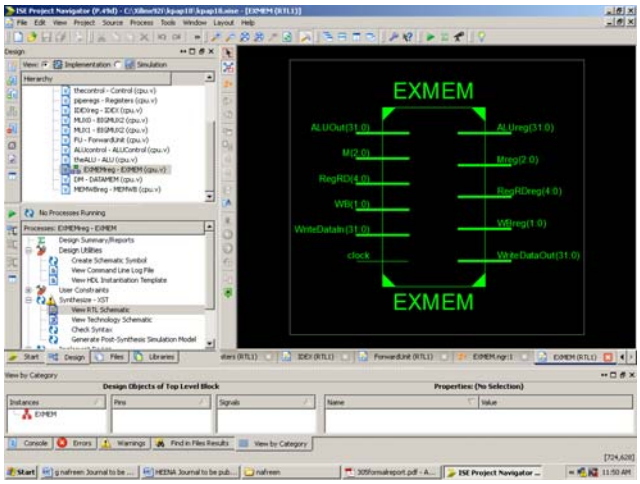


**Figure 18 RTL view of Exmem**

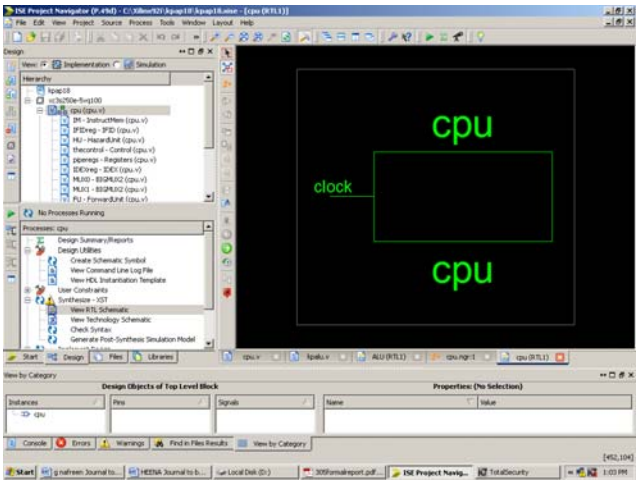


**Figure 20 RTL view of CPU**

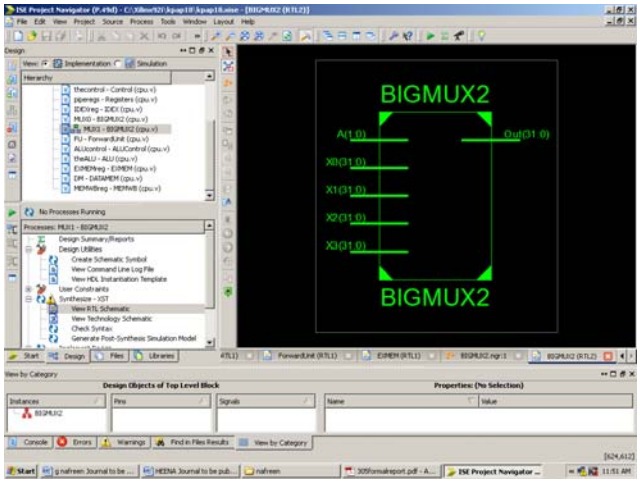


**Figure 19 RTL view of Multiplexer**

## 5. CONCLUSIONS

In this paper, the pipelines MIPS are implemented. The future enhancements of this work are detection of various hazards like data hazards, control hazards etc. Along with forwarding unit and hazard detection unit, detection of data hazard can be added. CACHE memory can be implemented in place of memory.

**Acknowledgments**

I would like to thank my Guide, HOD sir K.Sudhakar, Project Co-ordinator T.Chakrapani, & other Staff members of ECE department SJCET for helping me directly or indirectly in completion of this project. A special note thanks to K. Prasadbabu and S Ahmed Basha Sir's, who involved in project completion.

**References**

1. Five staged pipelined processor with self clocking mechanism by Anish Gupta*, *Vinayak Kini* 978-1-4673-7910-6/15/$31.00 c2015 IEEE
2. David A. Patterson, John L. Hennessy: Computer Organization and Design – The

Hardware/Software Interface. Fourth Edition (2006). Morgan Kaufmann Publisher, Inc.
3. http://nptel.iitm.ac.in/video.php?subjectId=106102062 Computer Architecture Principals video tutorials, Retrieved: August, 2013
4. http://www.iitg.ernet.in/asahu/cs222/ Lectures on Computer Organization and Architecture, Retrieved: August, 2013
5. Prof.Grishman,http://cs.nyu.edu/courses/fall08/V22.0436-001/lecture18.html retrieved: September, 2013
6. MIPSarchitecture,http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/MIPS.html retrieved: September, 2013.
7. Multicycle processor ppt of EE422 class of CSUN sent to me by Dr. Roosta . L. Crist´ofoli, A. Henglez, J. Benfica, L. Bolzani, F. Vargas, A. Atienza, and F. Silva, "On the comparison of synchronous versus asynchronous circuits under the scope of conducted power-supply noise," in *Electromagnetic Compatibility (APEMC), 2010 Asia-Pacific Symposium on. IEEE,* 2010, pp. 1047–1050.