

VLSI Design Project Report

**MIPS AND FAULT TOLERANCE IN VHDL**

LAZARIDIS DIMITRIS

(thejimi39@hotmail.com)

ATHENS 2012

## **ABSTRACT**

Implementation microprocessor Mips in hardware, supporting almost all of it's instructions including multiply packet.

The integration made in the environment of Xilinx in version 13.1 and verified in simulation of Xilinx and the project created in VHDL language.

The whole circuit is implemented in the Xilinx Spartan 3 and Place and Route has been made.

The general purpose of this project is to implement a basic 5 stage MIPS32 cpu. Particular attention will be paid to the reduction of clock cycles for lower instruction latency as well as taking advantage of high-speed components.

The error detection is implementing in hardware in compact circuits thus has fast execution time. In this method a transient or stack error is detected giving in this method high error detection fault coverage.

## First steps

A MIPS-32 compatible Central Processing Unit (CPU) was designed, tested, and synthesized. The processor had the following attributes:

- 5 stage
- Data Forwarding to reduce stall cycles

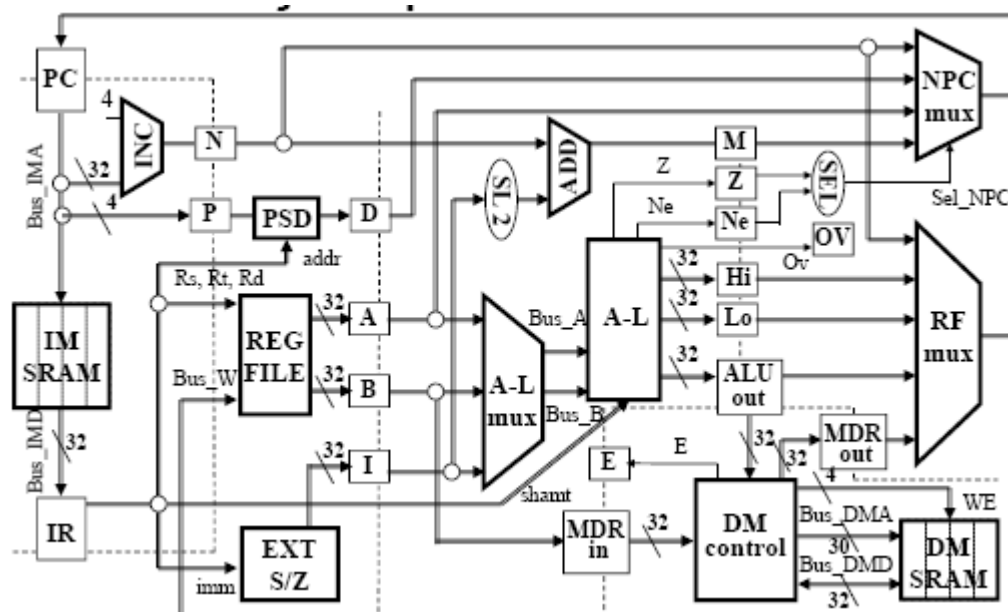
In the first step the hardware is divided into five stages IF, ID, EXE, MEM, WB. (The stages were Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back.)

These stages are the total processor which operates under control unit. Test benches verify the correction of the instructions result.

The instructions which are implemented:

LW, SW, ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, MULT, MFLO, MFHI, MTHI, MTLO, SLL, SRL, SRA, SLLV, SRLV, SRAV, BEQ, BNE, ADDI, ADDIU, ANDI, ORI, CHORI, LUI, SLT, SLTU, SLTI, SLTIU, JR, JALR.

## The five stages



The first stage is the Instruction memory, program counter and IR.

The Instruction Fetch stage is where a program counter will pull the next instruction from the correct location in program memory. In addition the program counter was updated with either the next instruction location sequentially, or the instruction location as determined by a branch.

The second stage is the Register File, the Ext (zero / s extension).

The Instruction Decode stage is where the control unit determines what values the control lines must be set to depending on the instruction. In addition, hazard detection is implemented in this stage.

The third stage is the execution the Alu with the necessary parts, Alu mux and the output registers Alu out, Hi, Lo, M.

The Execute stage is where the instruction is actually sent to the ALU and executed. If necessary, branch locations are calculated in this stage as well.

The fourth stage is the Data memory, DM control and MDR out.

The Memory Access stage is where, if necessary, system memory is accessed for data. Also, if a write to data memory is required by the instruction it is done in this stage. In order to avoid additional complications it is assumed that a single read or write is accomplished within a single CPU clock cycle.

The last stage is the RF mux and NPC mux.

Finally, the Write Back stage is where any calculated values are written back to their proper registers. The write back to the register bank occurs during the first half of the cycle in order to avoid structural and data hazards if this was not the case.

### **Instruction Fetch Stage**

The instruction fetch stage has multiple responsibilities in that it must properly update the CPU's program counter in the normal case as well as the branch instruction case. The instruction fetch stage is also responsible for reading the instruction memory and sending the current instruction to the next stage.

In IR also for the I type instructions some of the decoding is done in this stage in order to reduce the complexity in FSM

### **Sign Extender**

The sign extender is responsible for two functions. It takes the immediate value and sign extends it if the current instruction is a signed operation. It also has a shifted output for branches. The sign extender test bench checks for accuracy.

## **Decode Stage**

Describes the stage of the CPU's where the fetched instruction is decoded, and values are fetched from the register bank. It is responsible for mapping the different sections of the instruction into their proper representations (based on R or I type instructions). The Decode stage consists of the Control unit, the Sign Extender, and the Register bank, and is responsible for connecting all of these components together. It splits the instruction into its various parts and feeds them to the corresponding components. Registers Rs and Rt are fed to the register bank, the immediate section is fed to the sign extender, and the ALU opcode and function codes. The outputs of these corresponding components are then clocked and stored for the next stage.

## **Register Bank**

One of the primary pieces of data storage in the CPU is the register bank contained within the instruction decode stage. This bank of registers is directly reference from the MIPS instructions and is designed to allow rapid access to data and avoid the use of much slower data memory when possible. The register bank contained in the CPU consisted of the MIPS standard 32 registers with register 0 being defined as always zero.

The registers are defined as being read in the first half of the cycle and written in the second half. This is done to avoid structural hazards when one instruction is attempting to write to the register bank while another is reading it. Setting the register bank to this configuration also avoids a data hazard because a value that was just written can be read out in the same cycle.

## **Execute Stage**

This stage is responsible for taking the data and actually performing the specified operation on it. The execute stage consists of an ALU, ALU control and Multiplying function unit. The execute stage connects these components together so that the ALU will process the data properly, given inputs chosen by the forwarding unit, and will notify if a branch is indeed to be taken.

## Alu control

The instructions fields of mips have information and have the following structure.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

The function field is the information that analyzes the R-type commands and implements in the Alu control, which is under the control of the main control unit. This was accomplished by a large case statement dependent on the input control signals.

## ALU

The alu is responsible for performing the actual calculations specified by the instruction. It takes two 32 bit inputs and some control signals, and gives a single 32 bit output along with some information about the output – whether it is zero or negative.

## Memory Stage

This stage is responsible for taking the output of the alu and committing it to the proper memory location if the instruction is a store. The memory stage contains one component: the data\_memory object. It connects the data memory to a register bank for the write back stage to read, and also forwards on information about the current write back register. This register's number and calculated value are fed back to the forwarding unit in the execute stage to allow it to determine which value to pass to the ALU.

## Data memory

The DM circuit is only active when there is operation for read or write, otherwise freezes, via control signal, improving power reduction circuit.

## **WriteBack Stage**

The writeback stage is responsible for writing the calculated value back to the proper register. It has input control lines that tell it whether this instruction writes back or not, and whether it writes back ALU output or Data memory output. It then chooses one of these outputs and feeds it to the register bank based on these control lines.

## **Control Unit**

The Control unit takes the given Opcode, as well as the function code from the instruction, and translates it to the individual instruction control lines needed by the remaining stages. This is accomplished via a large case statement.

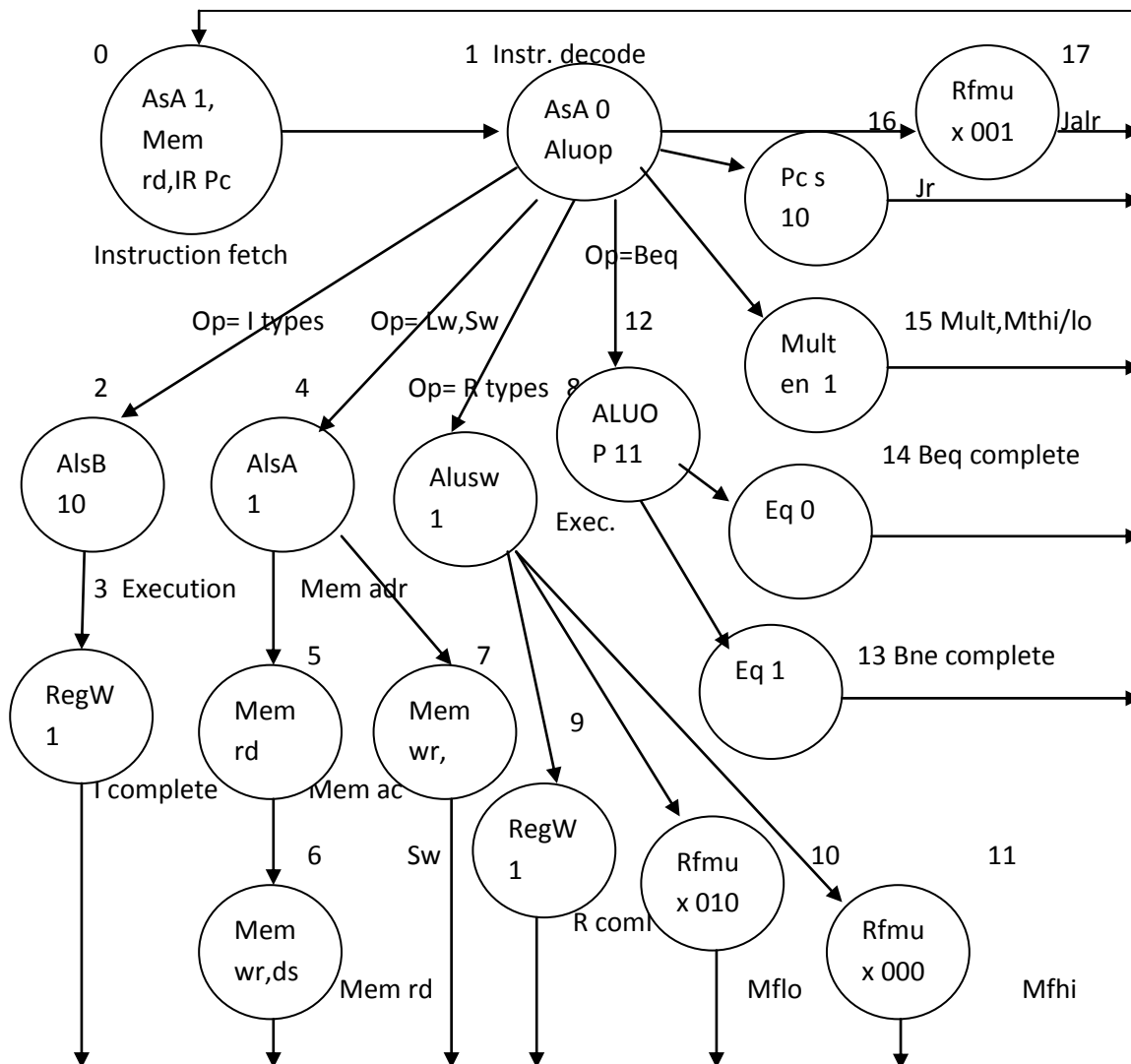
The control unit generates the control bits for the multiplexers, the Data memory and Alu control.

The inputs are the field op (6 bits), the function field, the signal clk and the signal rst.

The outputs are: RegDst, RegWrite, ALUSrcA, MemRead, MemWrite, Mult\_en, lrd, IRWrite, PCWrite, EqNq, ALUsw, ALUOp, ALUSrcB, PCSrc, ALUmux, ALUOp\_sw, RFmux.



## FSM diagram



## I type instructions

The commands I have the same next state except in the execution cycle, so by combining them can achieve the reduction stages. The differences in arithmetic operation is specified (in first stage), check carried out by ALU control.

## **R type instructions**

They have the same current and next state except arithmetic cycle execution like I type commands, the difference in function is specified by Alu control where are part decoded and executed.

## **The instructions Mult, Mflo, Mthi**

Belong to R type commands but have a different situation in their performance in this circle, the result must be recorded in the registers Hi, Lo.

A separate stage for these commands is made and different function performed by Alu control. (Record the Hi and Lo, or Hi or Lo).

The implementation of the entire transaction is not lost, even after performing the Mult many different instructions are follow, and is achieved through the control signal Mult\_en.

## **The instructions Mflo, Mfhi**

Owned in R type instructions but have a different next state to completion. No execution cycle exist it is an empty circle (bubble). Thus, it is possible to follow the course of the R type, but separated in the last cycle, reducing possible unnecessary stages.

## **To clk signal in Xilinx memories**

A numerous simulations and the best result was presented in descending pulse execution in memory (Read or Write).

The advantage of operating in the descending pulse in the memories is that there is the availability in the remaining rising pulse (at the same stage of operation) to implement any order or circuit (eg pipeline), this creates greater flexibility in hardware circuit with multiple variations, thus speed and flexibility for multiple implementations.

## **Performance**

**Instructions that are in instruction memory are in the follow order:**

AF890064	SW \$s1,100(\$s2) Store word (W)
8F890064	LW \$s1,100(\$s2) Load word
02538820	ADD \$s1 \$s2 \$s3 Addition

02538821	ADDU \$s1,\$s2,\$s3 Addition
02538822	SUB \$s1,\$s2,\$s3 Subtract
02538823	SUBU \$s1,\$s2,\$s3 Subtract
02538824	AND \$s1,\$s2,\$s3 AND
02538825	OR \$s1,\$s2,\$s3 OR
02538826	XOR \$s1,\$s2,\$s3 XOR
02538827	NOR \$s1,\$s2,\$s3 NOR
02530018	MULT \$s2,\$s3 Multiply
00008812	MFLO \$t1 Move from Lo
00008810	MFHI \$t1 Move from Hi
01200011	MTHI \$t1 Move to Hi
01200013	MTLO \$t1 Move to Lo
001288C0	SLL \$s1,\$s2, 3 Shift left logical
001288C2	SRL \$s1,\$s2, 3 Shift right logical
001288C3	SRA \$s1,\$s2, 3 Shift right arithmetic
02728804	SLLV \$s1,\$s2,\$s3 Shift left logical variable
02728806	SRLV \$s1,\$s2,\$s3 Shift right logical variable
02728807	SRAV \$s1,\$s2,\$s3 Shift right arithmetic variable
12720001	BEQ \$s1, \$s2, label Branch on equal
16720001	BNE \$s1, \$s2, label Branch on not equal
22290064	ADDI \$t1 \$s1,3 Addition immediate
26290064	ADDIU \$t1,\$s1,3 Addition immediate
32290064	ANDI \$t1,\$s1,3 AND immediate
36290064	ORI \$t1,\$s1,3 OR immediate
3A290064	XORI \$t1,\$s1,3 XOR immediate
3C090064	LUI \$t1 100 load upper immediate
0232802A	SLT \$t0, \$s0, \$s1 Set less than

0232802A	SLTU \$t0, \$s0, \$s1 Set less than unsigned
2A280064	SLTI \$t0, \$s0, 10 Set less than immediate
2D280064	SLTIU \$t0, \$s0, 10 Set less than unsi/immediate
01000008	JR \$t0 Jump register
0100F809	JALR \$t0 Jump and link register

At the end an algorithm follows.

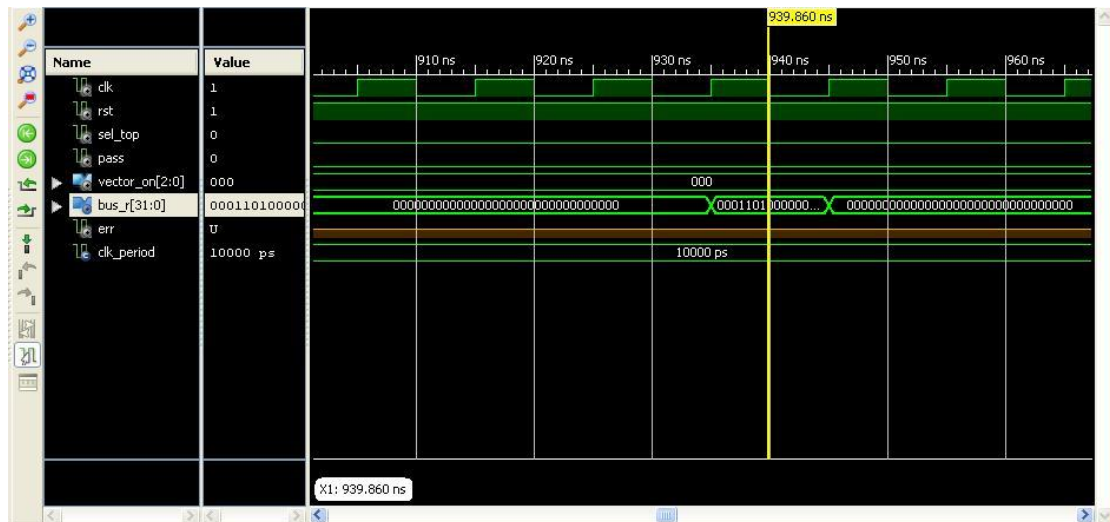
When the pc reaches the jr instruction the next instruction to be executed is the beginning of the algorithm fibonacci, and at the end of the execution of the next command execution is at x00000000. So the execution sequence of the entire program is never-ending cycle.

Created separate outside door called BUS\_W it makes it easier to monitor values and good record of execution through the simulator of Xilinx (via test benches and by executing the file main\_tst).

### The instructions results from Xilinx simulator



AF890064	SW \$s1,100(\$s2) Store word
----------	------------------------------



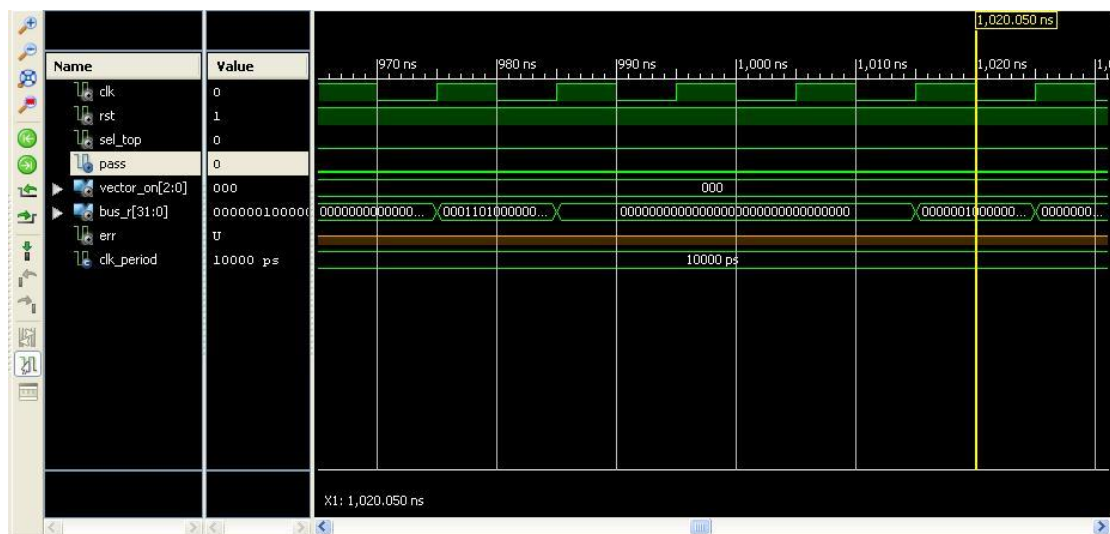
8F890064

LW \$s1,100(\$s2) Load word



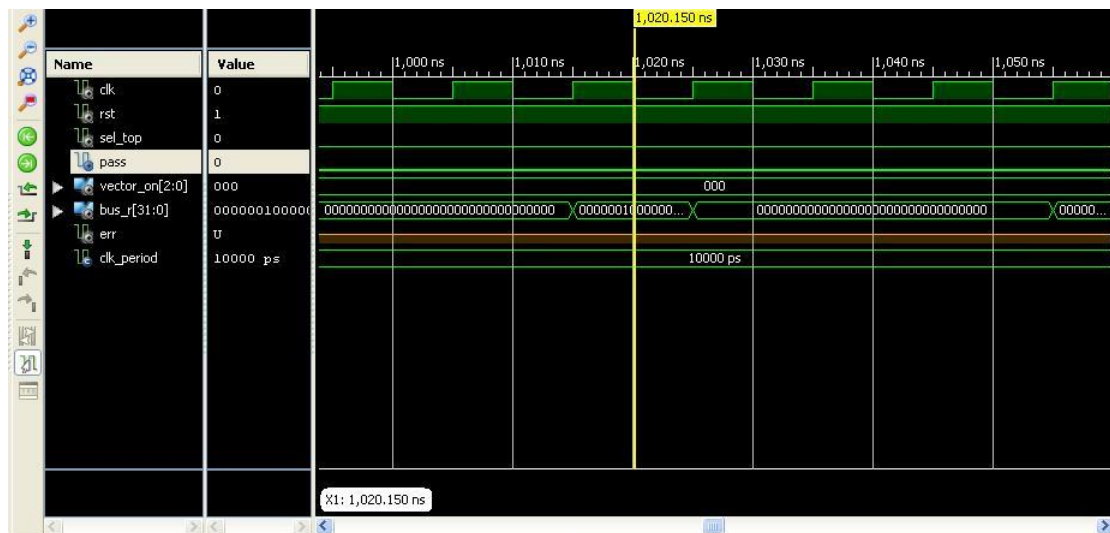
02538820

ADD \$s1 \$s2 \$s3 Addition



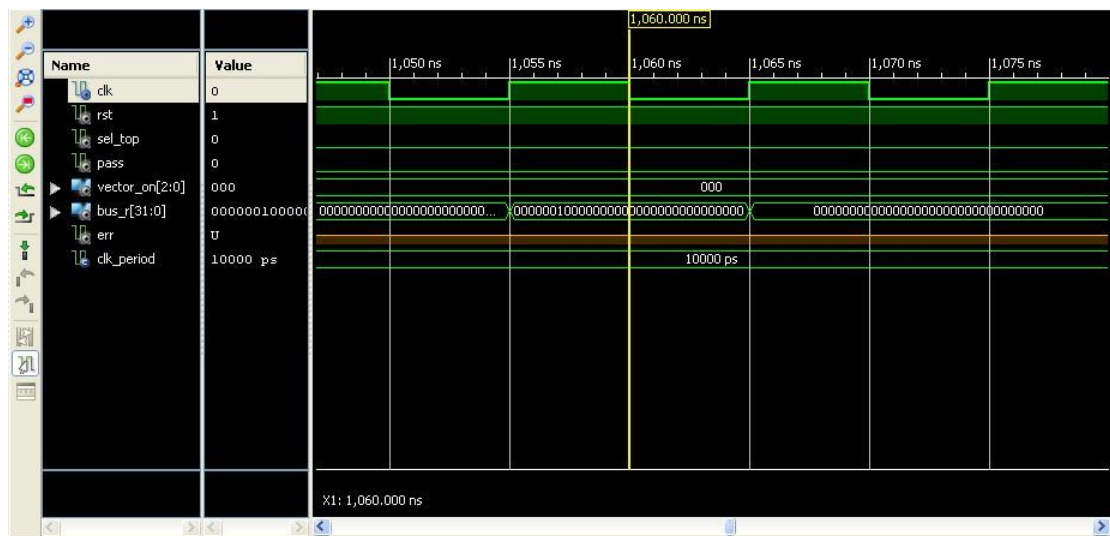
02538821

ADDU \$s1,\$s2,\$s3 Addition



02538822

SUB \$s1,\$s2,\$s3 Subtract



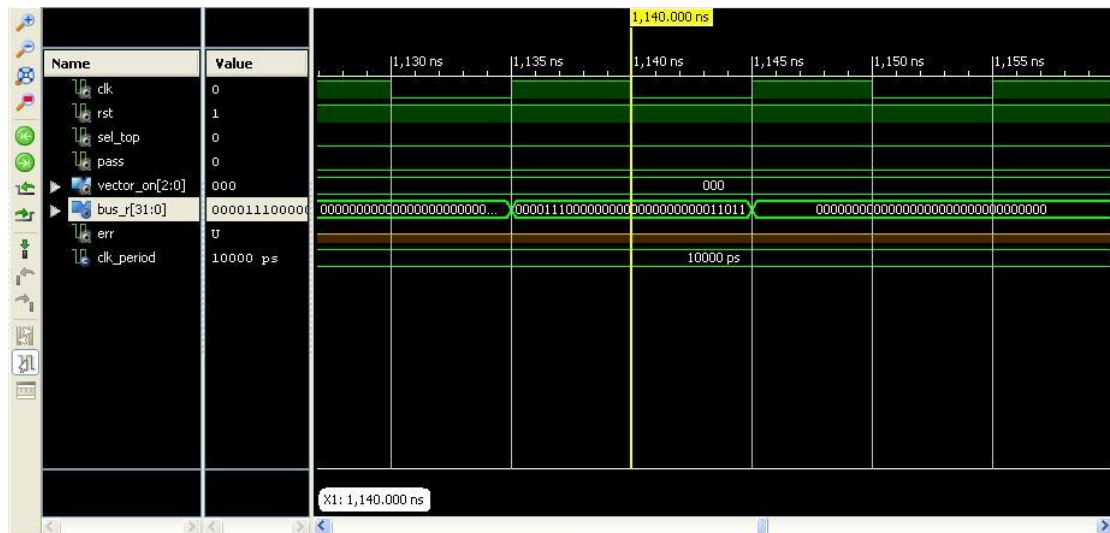
02538823

SUBU \$s1,\$s2,\$s3 Subtract



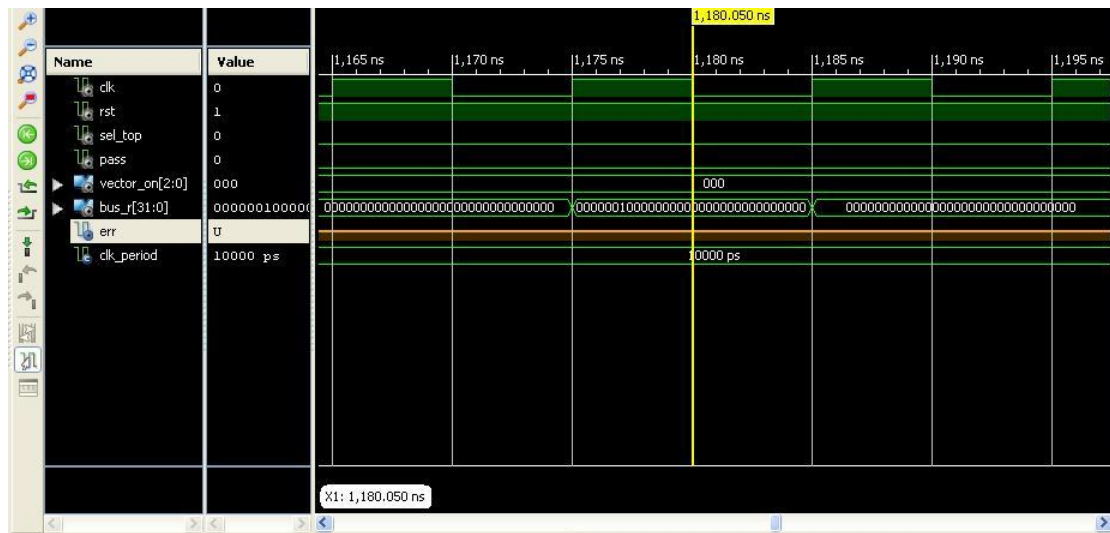
02538824

AND \$s1,\$s2,\$s3 AND



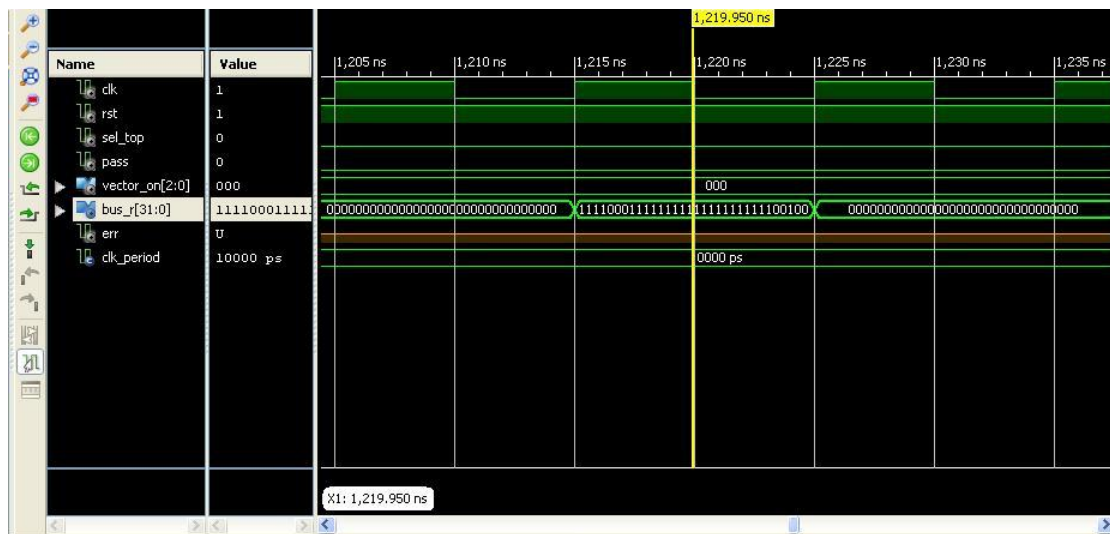
02538825

OR \$s1,\$s2,\$s3 OR



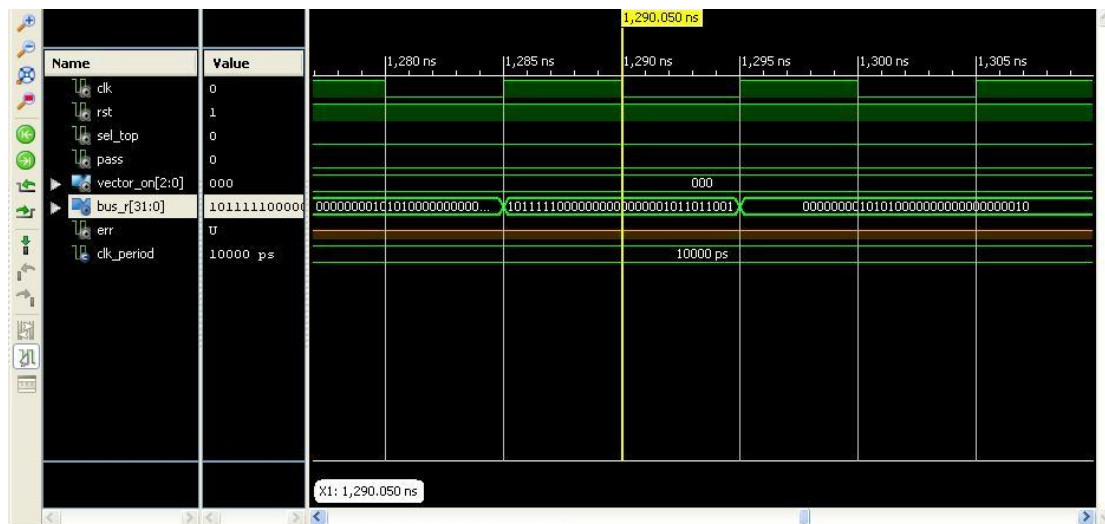
02538826

XOR \$s1,\$s2,\$s3 XOR

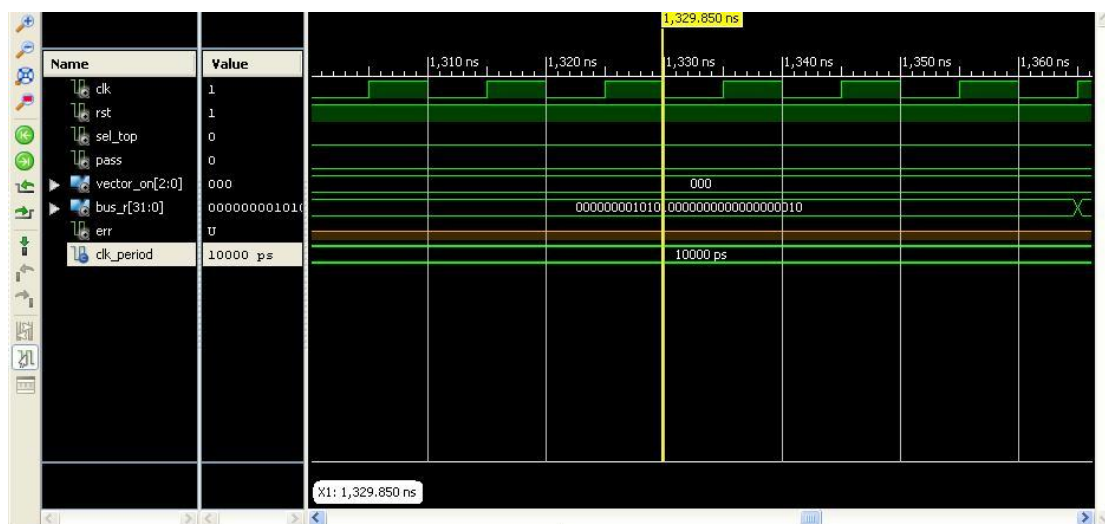


02538827

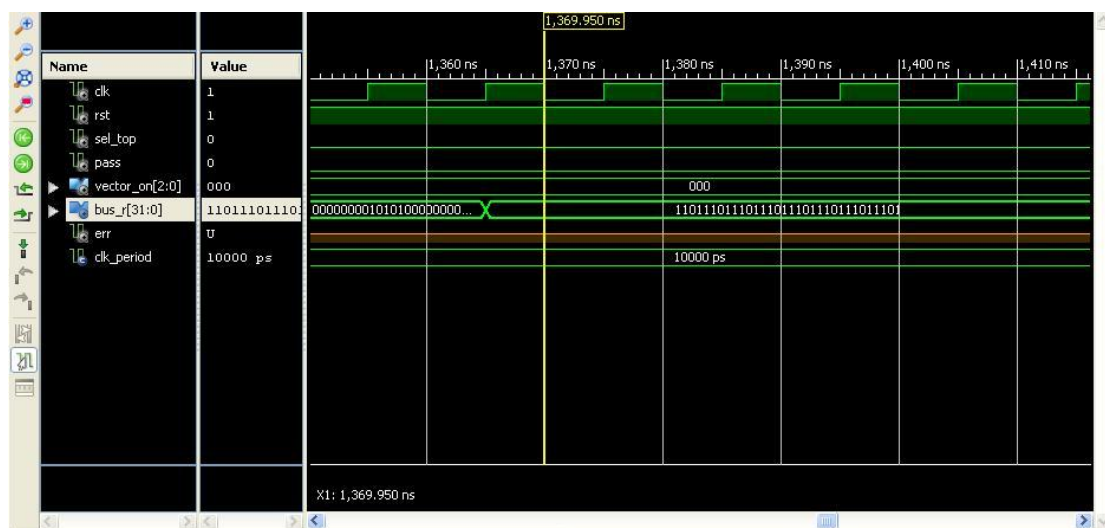
NOR \$s1,\$s2,\$s3 NOR



00008812 MFLO \$t1 Move from Lo

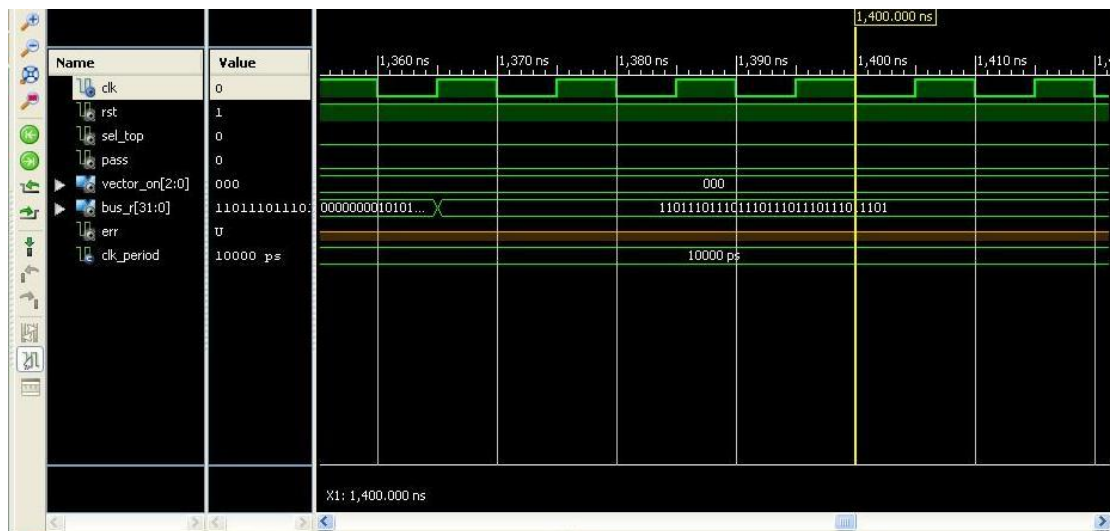


00008810 MFHI \$t1 Move from Hi



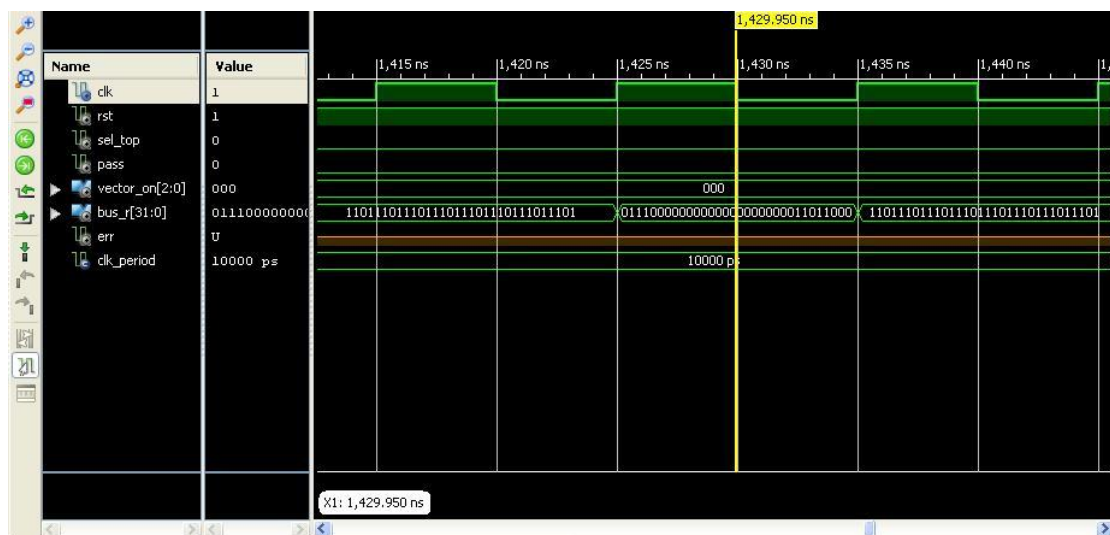
01200011 MTHI \$t1 Move to Hi





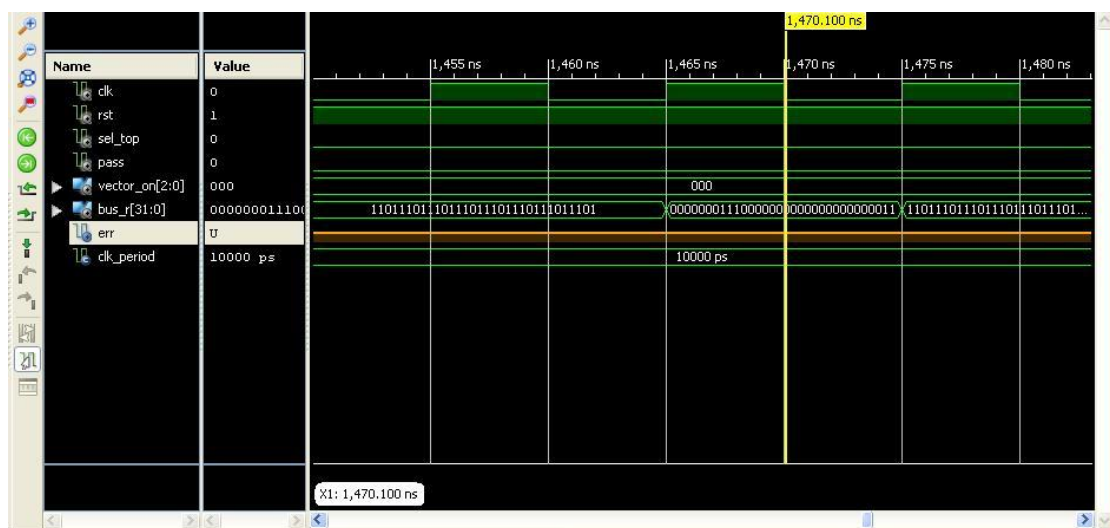
01200013

MTLO \$t1 Move to Lo



001288C0

SLL \$s1,\$s2, 3 Shift left logical



001288C2

SRL \$s1,\$s2, 3 Shift right logical

## **Error detection**

With continuous scaling in CMOS technology the number of transistors grows more and more in a single chip. Chip multiprocessors (CMPs) are an efficient way for using this very large number of transistors integrated in a chip. Several researches show that high density integration makes modern processors prone to the risk of transient or permanent fault. However, the increase of temperature and decrease of the voltage in the chip lead to a higher susceptibility to faults. As the feature size shrinks the probability of a single transistor to become faulty, it increases due to the low threshold voltages.

It is projected that the rate at which the transient errors occur will grow exponentially and will soon represent one of the most significant issues in the design of future generation high-performance microprocessors.

This work proposes a fault tolerant architecture that tolerates the high fault rates that are expected in future technologies. It is test the multiply block counting a 0f as an input data and compare the result with a signature. The error detection begins at start up before any execution. It has a high fault coverage and fast execution due to hardware implementation, which will be more popular method for errors detection in future for the time saving (there is not time penalty), reliability, low cost and high presentence to fault coverage, low power consumption.

## **Summary and Conclusion**

In conclusion, the experiment was a success. A fully realized MIPS32 compliant five stage CPU was developed, implemented, and tested successfully. A bottom up design process was followed in developing the CPU. First the smallest components, such as the program counter, were designed and tested. These functional blocks were then combined to make each of the five stages. Finally these five stages were connected together to create the final CPU. Methods that could potentially improve this project would be to test different implementations of the design in reducing I type and R type instructions, as that was a significant portion of the ALU. The store of Mult instruction in useful for compacted algorithms. The rate at which the transient errors occur will grow exponentially and error detection in fault coverage it will be an important issue. The hardware error detection method has a high fault coverage and fast execution, low cost, low consumption.

## **Further research**

Most error detect methods for fault tolerance check the mips or a circuit at start up or at once or periodically to find any errors for fault coverage, but what if an error occurs during the tests? A fault data will process as correct. To work around with this, a non stop searching method test the mips continuously, it can be implement and find any error as it appears at the beginning and further more if the fpga has enough room to relocate the damaged place in another undamaged.

To implement this error detect method, we can inject in fsm and detect the errors for fault tolerance. Knowing the next stage (instruction) through fsm, and decide to test the “multiply” block circuits, could test the multiply circuits until the next instruction it is not concern this circuits, if a multiply instruction is coming up we can stop the process and continue when it is free again, thus we can find if an error occurs in this part of cpu and cover the fault tolerance. The same process it is possible to test all critical parts of mips or central unit and find if an error exists. The advantage in this method is that the error detects circuit works continuously. This method does not require double cores, but only some additional parts (low cost) and which can work in conjunction with fsm without consume the microprocessor’s working time but it works simultaneously.

## References

Sundar Rajan. 1999. Essential VHDL. RTL synthesis Done Right

Ds099.pdf 2009. Xilinx Spartan-3 FPGA Family.Data Sheet

[www.xilinx.com](http://www.xilinx.com)

Spartan3\_hdl.pdf 2011. Xilinx. Spartan-3 Libraries Guide for HDL Designs

[www.xilinx.com](http://www.xilinx.com)

Ug222.pdf 2009. Xilinx. Spartan-3 Generation Configuration User Guide

[www.xilinx.com](http://www.xilinx.com)

xapp463.pdf 2005. Xilinx. Using Block RAM in Spartan-3 Generation FPGAs

[www.xilinx.com](http://www.xilinx.com)

L16-Multicycle-MIPS.ppt 2010. Montek Singh Multicycle MIPS. COMP541.

lec07-MIPS.pdf 2012. John Wawrzynek EECS150-Digital Design MIPS

lec20.pdf Erik Jonsson. The CPU Control Unit *The University of Texas at Dallas*

mips01.ppt Haldun Hadimioglu MipsVersion0&1 *Polytechnic institute of NYU*

MIPS\_Processor.ppt S. Reda. 2007. *8-bit MIPS Processor*.

report.doc Yu Zhang. Redesign Control FSM of a Multicycle MIPS Processor with Low Power State Encoding. *Electrical and Computer Engineering Department*.

lec06-mult.pdf 1997. Dave Patterson. *Computer Architecture and Engineering*

<http://cs.berkeley.edu/~patterson>

lecture3\_combinational\_blocks.ppt George Mason University. *FPGA and ASIC Design with VHDL*

vhdl\_math\_tricks\_mapld\_2003.pdf 2003. Jim Lewis VHDL Math Tricks of the Trade.

lecture9\_synthesis.ppt George Mason University. *VHDL Coding for Synthesis*