16-Bit Microprocessor

Ajay Bindingnavale (ajay.morph@gmail.com)
Chaitanya Reddy (chinna.bommu@gmail.com)
Revanth Rajalbandi (rajrevanth61@gmail.com)
Harsha Gadiraju (harshagadiraju123@gmail.com)

***Abstract* — Processor is an integral part of the computer where processing is done in the binary level and a human readable form of output is displayed on the screen. In the recent decades many computer architectures exhibiting various design methodologies and computational models have been developed. One of the widely accepted architecture is the von-Neumann architecture (Princeton). In this paper we describe the implementation of a 16 bit microprocessor, which is based on MIPS architecture and has a von-Neumann model of memory. The target application of the processor is to implement any kind of instruction that could be implemented by the integral part of the general purpose computer. Processor has been designed and synthesized for a 0.6µm CMOS technology. Fibonacci Series program is used for verifying the correctness and functionality of the processor.**

Keywords—Datapath; Controller; opcode; operand; UART; processor; Functional Simulation; Gate level Simulation;.

I. INTRODUCTION

MIPS stands for Microprocessor without Interlocked Pipeline Stages. Processors based upon the MIPS instruction set have been in production since 1988. Over time several enhancements of the instruction set were made. The different revisions which have been introduced are MIPS I, MIPS II, MIPS III, MIPS IV and MIPS V. This evolution has promoted the arrival of strategies and techniques such as pipelining, cache management, multithreading, parallelism which are a few to mention.

In an un-pipelined processor such as the one which is implemented in this paper, it fetches the first instruction from memory, increments the program counter, performs the operation it is called for, writes back either the memory or the registers and then fetches the next instruction from the memory, and so forth. These kinds of processors can be implemented with both the instruction memory and data memory separately or can be in common. While fetching the instruction, the ALU part of the processor is idle and the fetch of the next instruction happens only after the completion of the previous instruction.

The 16-bit processor implemented has 16-bit address bus, 16-bit data bus and eight 16-bit data registers along with a program counter. This 16-bit processor designed supports wide range of instructions accounting to a number of 22, which have been exhaustively tested for the correct functionality using ModelSim and NC_Verilog. In order to view the result of the computation, UART is used for the serial transmission of output to the LCD Screen.

Figure 1 presents the block diagram for the proposed system. Instruction execution generally flows from left to right. The program counter (PC) specifies the address of the instruction. The instruction is loaded 2 bytes at a time over one cycles from an off-chip memory into the 16-bit instruction register (IR). The Op field is sent to the controller, which sequences the datapath through the correct operations to execute the instruction. The controller contains a finite state machine (FSM) that generates multiplexer select signals and register enables to sequence the datapath. A state transition diagram for the FSM is shown in Figure 2. The fetch stage shown in the figure comprises of 3 fetch stages which will are used to fetch the instruction from the memory. The FSM then is dispatched based on Opcode field to execute the particular instruction.

II. MICROPROCESSOR DESIGN ARCHITECTURE

The Microprocessor architecture implemented is based on MIPS architecture which was developed in 1980's by John Hennessey in Stanford University. The design implemented is a mix of synthesized Verilog and custom circuits. The two main custom circuits that would be designed are Custom ALU and Custom Memory. The custom memory would be an on chip 8X16 SRAM. This would be used for data registers, program status register, program counter and UART buffer registers. The required cells are present in the library Lib6710_2.

The devices which are interfaced are memory and LCD screen. The mode of communication which is used for interacting with these interfaces are parallel data interface and UART interface respectively. RS-232 UART protocol is used with a baud rate of 9600 bits/sec for communication between the processor and the LCD.

Figure 3: UART Implementation

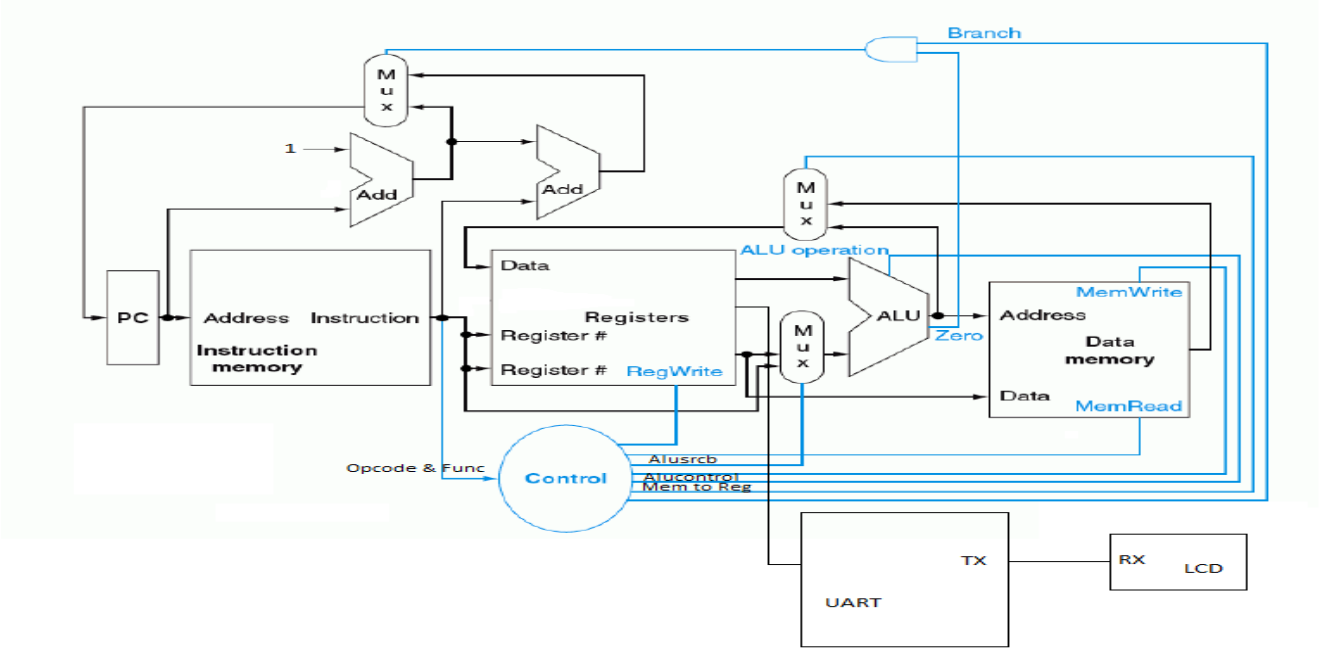
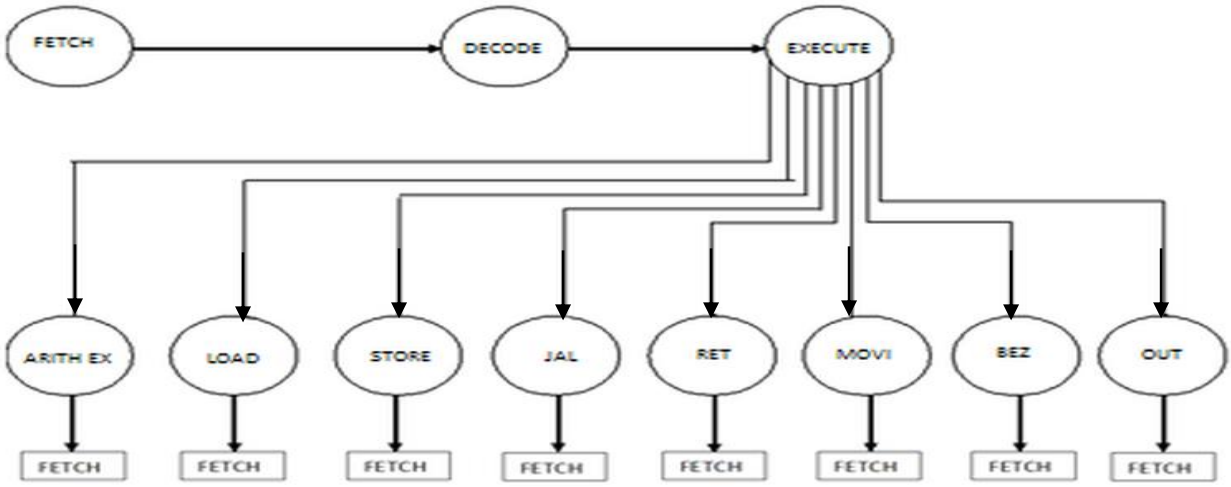


Figure 1: Block Diagram of the 16-bit microprocessor showing the interaction between datapath and the controller. Where the blue lines represent the control signals of the controller and black lines represent the control signals of the datapath.



F

Figure 2: State diagram depicting the various state of the operation of the processor

III. INSTRUCTION SET ARCHITECTURE

Instructions implemented in this microprocessor are 16 bit wide.

- **Arithmetic Instructions**

ADD, SUB, AND, OR, XOR, CMP.

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[001]	Function	Regsrc1	Regsrc2	Regdest

MOV

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[001]	Function	-----	Regsrc2	Regdest

CLEAR

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[001]	Function	Regsrc1	----	----

INR, DCR

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[001]	Function	Regsrc1	----	----

Table 1: Function Fields of Arithmetic Instructions

Name of the Instruction	Function Field
ADD	0000
SUB	0001
AND	0010
OR	0011
XOR	0100
CMP	0101
MOV	1000
CLR	1001
INR	0110
DCR	0111

- **Logical Instructions**

LOGICAL AND, LOGICAL OR, LOGICAL SHIFT LEFT, LOGICAL SHIFT RIGHT.

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[011]	Function	Regsrc1	Regsrc2	Regdest

LOGICAL NOT.

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[011]	Function	Regsrc1	----	Regdest

Table 2: Functional Fields of Logical Instructions

Name of the Instruction	Function Field
AND	1011
OR	1100
SHIFT LEFT	1101
SHIFT RIGHT	1110
NOT	1111

- **Data Transfer Instructions**

LOAD.

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[001]	----	Regsrc1	----	Regdest

STORE.

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[001]	----	Regsrc1	Regsrc2	----

- **Branch Instruction**

Jump & Link (JAL)

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[100]	----	Regsrc1	Regsrc2	----

BEZ

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[111]	----	Regsrc1	Regsrc2	----

RET

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[101]	----	Regsrc1	----	----

- **Immediate Instruction**

MOVI

15 – 13	12 – 3	2 – 0
Opcode[110]	Immediate Value	Regdest

- **Output Instruction**

OUT (Explicitly for UART).

15 – 13	12 – 9	8 – 6	5 – 3	2 – 0
Opcode[000]	----	Regsrc1	----	----

Instruction set with operation

Table 3: 16-bit Processor Instruction Set

Assembly Instruction	Operation
ADD $rs, $rt, $rd	$rd = $rs +$rt
SUB $rs, $rt, $rd	$rd = $rs - $rt
AND $rs, $rt, $rd	$rd = $rs & $rt
OR $rs, $rt, $rd	$rd = $rs | $rt
XOR $rs, $rt, $rd	$rd = $rs ^ $rt
CMP $rs, $rt, $rd	($rs == $rt)? 1'b0: ($rs > $rt)? 2'b10 : 1'b1
MOV $rs, $rd	$rd = $rs
CLEAR $rs	$rs == 0
INR $rs	$rs = $rs + 1
DCR $rs	$rs = $rs - 1
LOGIAND $rs, $rt, $rd	$rd = $rs && $rt
LOGIOR $rs, $rt, $rd	$rd = $rs || $rt
LOGINOT $rs, $rd	$rd = ~$rs
SHIFTLEFT $rs, $rt, $rd	$rd = $rs << $rt
SHIFTRIGHT $rs, $rt, $rd	$rd = $rs >> $rt
LOAD $rs, $rd	$rd = M[$rs]
STORE $rs, $rt	M[$rs] = $rt
MOVI $rd, #imm	$rd = imm
OUT $rs	Value contained in $rs is transmitted.
BEZ $rs, $rt	If($rs == 0) then PC = $rt
JAL $rs, $rt	PC = $rt $rs = old pc
RET $rs ($rs has previously saved PC Value)	PC = $rs

Explanation of the instruction set architecture

Instruction is of 16 bits

- Out of 16 bits [15-13] 3 bits are used for denoting Opcode.
- 4 bits [12-9] are used as function field in order to differentiate between arithmetic and logical instructions, in other kind of instructions this field is left blank.
- 3 bits from [8-6] and also [5-3] are used as source registers .The second register field is left blank where the instruction requires only one operand to work with.
- 3 bits from [2-0] are used for destination register, even this field is left blank when instruction has nothing to write in any of the registers.

IV. Interesting Circuit Parts

The special feature of our Microprocessor is that it can execute all the instructions executed by a general purpose computer and we are also implementing UART in order to communicate with the external world.

Datapath

Datapath Instantiates ALU and the register files, ALU contains the giant mux and the combinational logic that produces the results of the instructions described in Table 3. Apart from this datapath has the registers for PC, aluresult, datain, srcA and srcB.

Datapath consists of the multiplexer that selects whether PC or ALU sends address to the external memory, and a multiplexer which decides whether ALU or external memory writes to the register file and a multiplexer to decide whether PC or srcA register is the source 1 of the instruction. One more multiplexer which selects whether the value 1 or srcB or immediate value is selected. One more mux which selects whether write to register is happening on the first register or the third register. Datapath is shown in figure 1.

Controller

Controller maintains a state machine which decides the next state based on the current state and the instruction. Depending on the current state respective control signals are enabled which are used to control the datapath

UART

UART is acronym of universal asynchronous receiver and transmitter. A UART receives serial data and stores it as parallel data (usually one byte), and takes parallel data and transmits it as serial data. UARTs are commonly used in conjunction with communication standards such as EIA, RS-232, RS-422 or RS-485

Interfacing to the chip is done by UART. This method is used for the interfacing because it requires few I/O pins and also it is easy to implement. In order to maintain the synchronization between the transmitted and received signals we have to maintain a transmission and reception rate which is called as the baud rate, for our UART implementation we are going for a baud rate of 9600 bits/s.

UART baud rate generator is implemented using a 11-bit counter which counts from 0 to 2^{11} . when this state is reached a pulse called baud tick is generated which determines the state of the UART and decides the next bit to be transmitted which might be a start bit, stop bit or data bit. The UART was implemented for only transmission purposes. UART implementation is shown in the figure 3.

V. Chip Testing

The microprocessor instructions have been tested individually with functional simulations and with structural simulations, later these instructions are tested together by taking the Fibonacci Series as an example

Functional Simulations

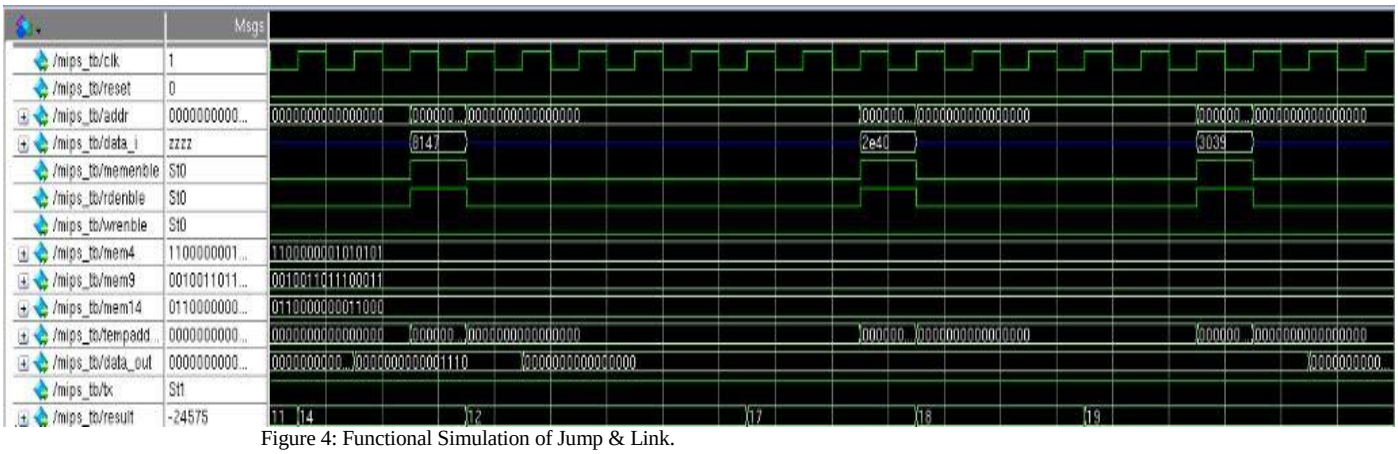


Figure 4: Functional Simulation of Jump & Link.

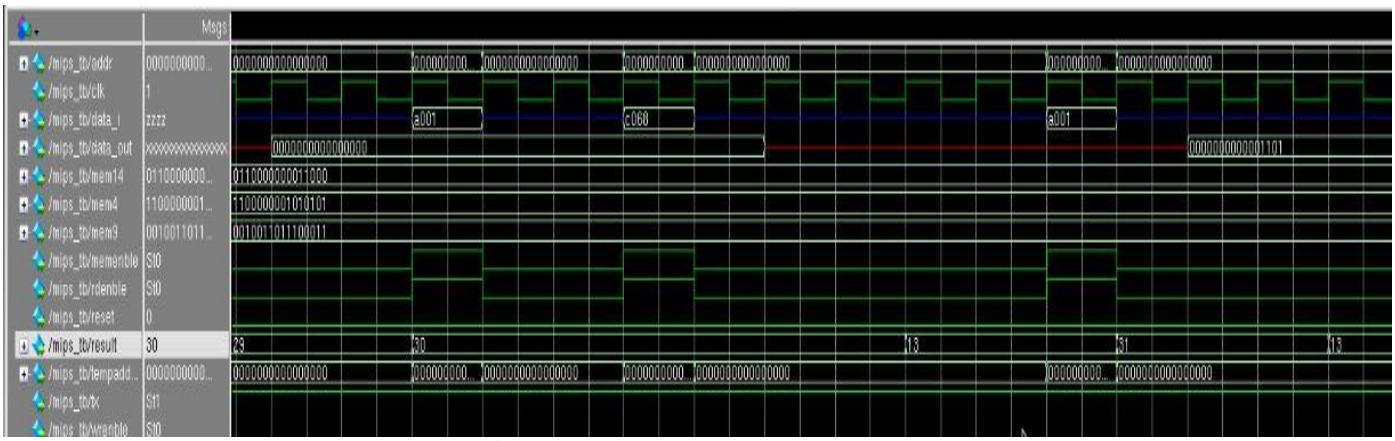


Figure 5: Functional Simulation of RET instruction

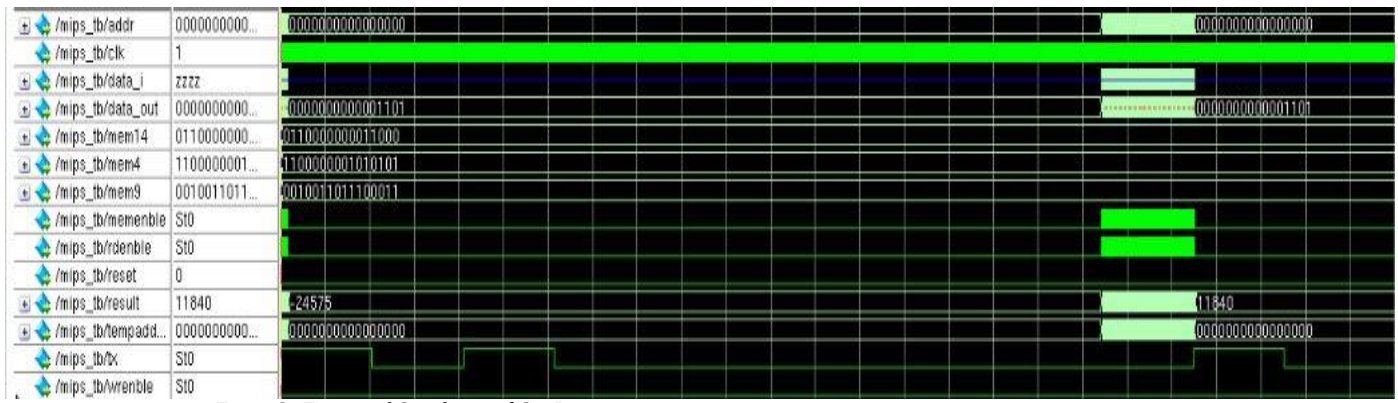


Figure 6: Functional Simulation of Out Instruction

- **Jump & Link (JAL)**

The Verification for the working of the JAL is shown with the functional Simulation in figure 4. By observing the fig 4 the value in the data_i (8147) is the instruction word of JAL(opcode are stored in external asynchronous SRAM). By executing the JAL instruction the PC value is being changed from 14 to 17 which can be viewed in result pin of the figure 4.

- **RET**

The Verification for the working of the RET is shown with the functional Simulation in figure 5. By observing fig 4 the value in the data_i (a001) is the instruction word of RET instruction. By executing the RET instruction the PC value is being changed from 30 to 13 which was the previous PC value saved when Jump & Link Instruction was executed.

- **OUT**

The Verification for the working of the OUT instruction which is used explicitly for the UART is shown with the functional Simulation in figure 6. The 8 bit data to be sent out is 10000000. By observing the TX signal initially stop bit was there. At the time of transmission first start bit is sent

(0) which is followed by 10000000 and then the stop bit is sent (1).

Fibonacci Series Simulation Results

The Fibonacci Series Assembly level program is as below, the opcodes are calculated from the above section. This code is run for 10 cycles and the Fibonacci series obtained is 0 1 1 2 3 5 8 13 21 34. From the Simulation waveform shown in the figure 7 also shows that the output obtained is 34, so this verifies that the instructions are correctly executed.

INSTRUCTION	OPCODE
MOVI R1, 0	C001
MOVI R2, 1	C00A
MOVI R3, 2	C043
MOVI R5, 12	C05D
MOVI R6, 6	C035
ADD R4, R1, R2	2054
MOV R1, R2	3011
MOV R2, R4	3022
DCR R3	2EC0
BEZ R3, R5	E0E8
RET R6	A180
OUT R4	0100

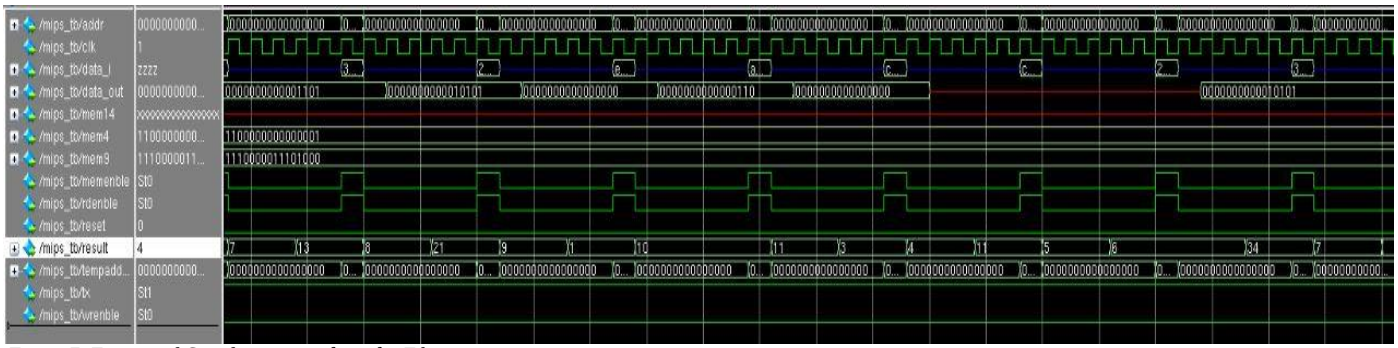


Figure 7: Functional Simulation waveform for Fibonacci series program.

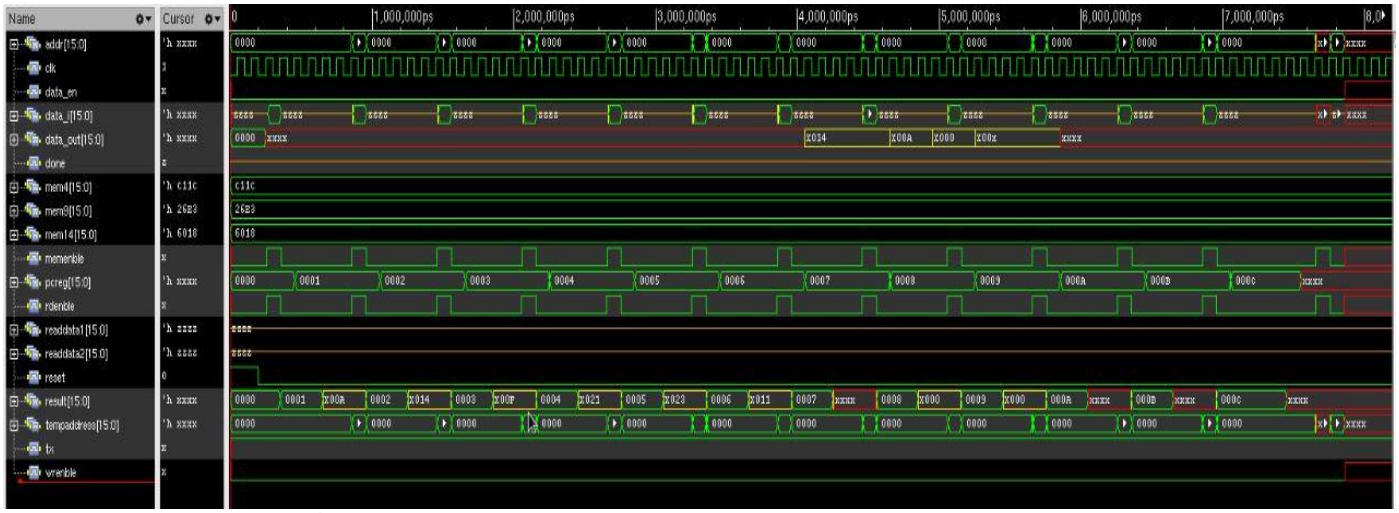


Figure 8: Gate level simulation waveform.

Gate level Simulation Results

Gate level Simulation results are shown in the figure 8. If we observe the 16 bit result signal (which is output of datapath). 15th bit is always going into undetermined state. For exmaple when we are doing MOVI 0X000A to some register R0, R0 is getting the value 0XX00A so when we are performing any arithmetic or logical operations using this R0 as one of the operand the result comes out properly other than undetermined bits. When we are performing RET, JAL using this register as the operand (which holds the memory address of the next instruction word to be fetched) as one of the bits are don't cares the value which we will get from the memory is XXXXX, This creates the problems for JAL in structural simulations.

VI. Conclusion

The main purpose of our project is to design a 16 bit Microprocessor which will be capable of supporting quintessential instruction set. We have accomplished this task by designing controller and datapath and also by testing the functionality of each instruction and intermediate values generated by those instruction exhaustively.

VII. References

[1] Verilog HDL: A guide to digital design and synthesis – Samir palnitkar.

[2] John L. Hennessy and David A. Patterson, Computer Organization and Design, 2nd edition, p.357

[3] http://www.asic-world.com/

[4] http://www.wikipedia.org/

Acknowledgement

Team would take this opportunity to thank Prof. Erik Brunvand for his constant help and guidance throughout the project. We would also like to thank Paymon Saebi for providing valuable information from time to time to make things work for the library characterization.