# EEDG 6304
# COMPUTER ARCHITECTURE

## RESEARCH PROJECT REPORT II

## SUPERSCALAR INSTRUCTION DISPATCH USING TOMASULO ALGORITHM

**SUBMITTED BY**

**AISHWARYA BARI - 2021272669**
**SAGAR PATIL - 2021271226**
**MIHIR SHAH - 2021268117**

# INDEX

# CHAPTER I
# INTRODUCTION

## SECTION A
## MOTIVATION FOR OUT OF ORDER EXECUTION

A major limitation of simple pipelining techniques is that they use in-order instruction issue and execution: Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result. If there are multiple functional units, these units could lie idle [1].

There is a limitation while achieving lesser CPI by deepening the pipeline due to the fact of control hazards, which results in flushing of deep pipeline in case of a misprediction. Though advance techniques have developed in areas of branch prediction, the level to which parallelism can be achieved in deeper pipeline is limited to several others factors. Thus, there is a need of Instruction Level Parallelism (ILP) to reduce the CPI value.

However, while achieving ILP through Superscalar, only independent instructions can be executed in parallel. Instructions with dependencies would be stalled. To overcome this problem, the solution is to have an Out-of –Order execution which is essentially a data flow graph and hence the semantics of the program are kept intact [2].

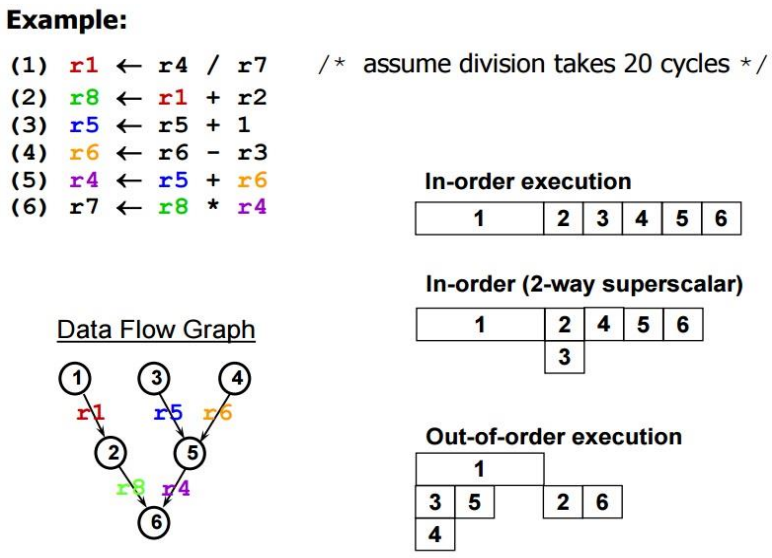



Fig.1.1: Example showing O-o-O

The example shown in Fig.1.1 on the left side gives an insight at Out-of-Order execution and compares its result wit in-order execution and superscalar as well.

The advantage of O-o-O is that we can exploit the Instruction Level Parallelism and hide latencies due to cache miss etc.

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline. However, with techniques like Re-order Buffer and Register Renaming those hazards can be eliminated.

Out-of-order execution is an approach that is used in high performance microprocessors. This approach efficiently uses instruction cycles (is a process by which a computer retrieves program instruction from its memory, determines what action the instruction requires and carries out those actions.) and reduces costly delay.

**Section B**
**Tomasulo Algorithm:**

Tomasulo's algorithm is a computer architecture hardware algorithm for dynamic scheduling of instructions that allows out-of-order execution, designed to efficiently utilize multiple execution units.

In Tomasulo's scheme, register renaming is provided by reservation stations, which buffer the operands of instructions waiting to issue. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register.

As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming. Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler. As we explore the components of Tomasulo's scheme, we will return to the topic of register renaming and see exactly how the renaming occurs and how it eliminates WAR and WAW hazards [1].

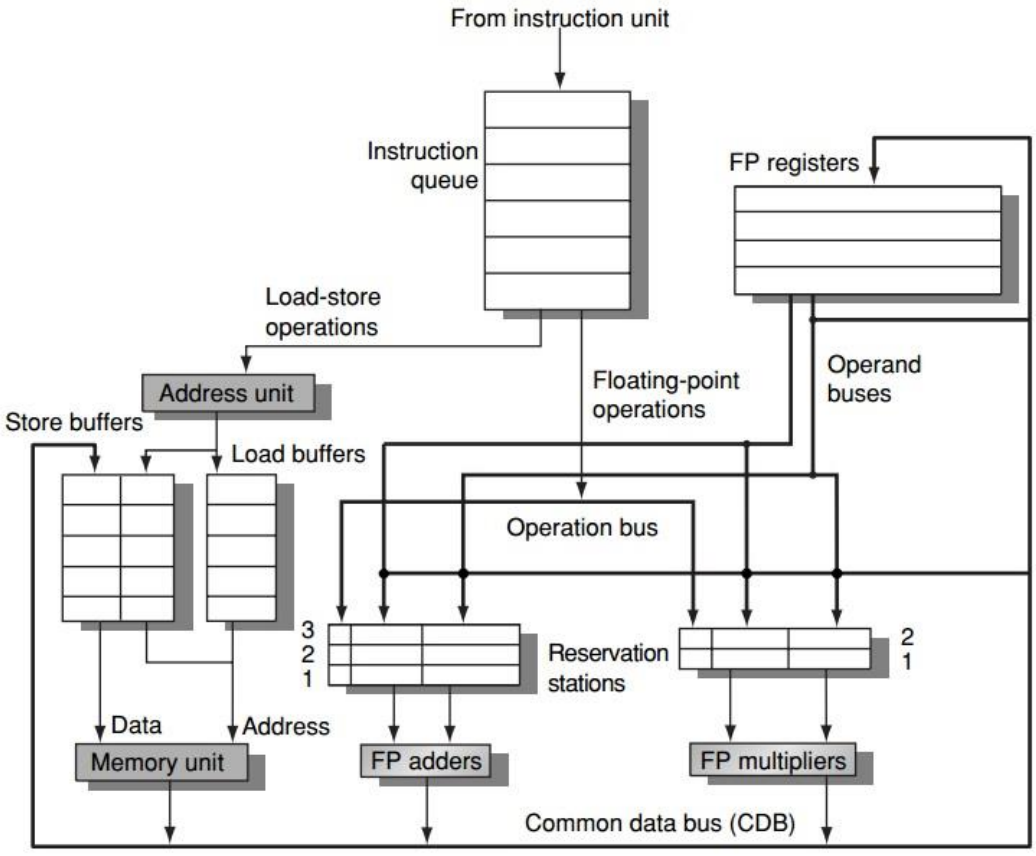



Fig.1.2 Basic Structure of MIPS Floating Point Unit using Tomasulo's Algorithm [1]

There are only three steps, although each one can now take an arbitrary number of clock cycles [1]:

1. **Issue**—Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers. If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed. If the operands are not in the registers, keep track of the functional units that will produce the operands. This step renames registers, eliminating WAR and WAW hazards. (This stage is sometimes called dispatch in a dynamically scheduled processor.)
2. **Execute**—If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into any reservation station awaiting it. When all the operands are available, the operation can be executed at the corresponding functional unit. By delaying instruction execution until the operands are available, RAW hazards are avoided.
3. **Write result**—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores are buffered in the store buffer until both the value to be stored and the store address are available, then the result is written as soon as the memory unit is free.

Tomasulo's algorithm implements register renaming through the use of what are called reservation stations as shown in Fig.1.2. Reservation stations are buffers which fetch and store instruction operands as soon as they're available. Source operands point to either the register file or to other reservation stations. Each reservation station corresponds to one instruction. Once all source operands are available, the instruction is sent for execution, provided a functional unit is also available. Once execution is complete, the result is buffered at the reservation station.

Thus, unlike in scoreboarding where the functional unit would stall during a WAR hazard, the functional unit is free to execute another instruction. The reservation station then sends the result to the register file and any other reservation station which is waiting on that result. WAW hazards are handled since only the last instruction (in program order) actually writes to the registers. The other results are buffered in other reservation stations and are eventually sent to any instructions waiting for those results. WAR hazards are handled since reservation stations can get source operands from either the register file or other reservation stations (in other words, from another instruction) [4].

**Section C**
**Research Goal: Performance with Unification of Tomasulo Algorithm and Superscalar Dispatching of Instructions**

Our goal of this Research Project is to implement and combine one or more techniques learnt in this course in order to improve the performance of CPU.

Thus, we have demonstrated in this Research Project an approach to modify the existing Tomasulo's Algorithm by combining it with Superscalar Instruction Dispatching techniques and to compare the Performance of the proposed design with existing Tomasulo's Algorithm which issues one instruction per cycle as well as an In-order Processor which consists of a Single ALU Unit and performs issuing and executing of instructions in-order.

Our goal is to redesign the micro-architecture such that there is a significant improvement in performance in terms of CPI. Thus, while programming the architecture in Behavioral Verilog Language in Xilinx ISE, we have created several modules which will be discussed in Chapter IV and its results will be discussed in Chapter V.

Thus, instead of convential Tomasulo's Algorithm, our idea behind this project is to utilize the benefits and exploit Instruction Level Parallelism (IPL) through Superscalar technique and a modified Tomasulo's Algorithm to allow execution of instructions out of order. The correct strategies of modifying traditional Tomasulo algorithm has been explained in [5]-[3], which served a great help.

Though there are advance techniques like Multithreading in single core and multi-core which enhances the performance of CPU drastically, we wanted to come up with a novel idea involving traditional techniques to solve the CPU enhancement issue.

Chapter III discusses a lot about the proposed design in terms of architecture, the assumptions that we have considered in our proposed design, functionalities of each block and finally in Chapter V we have discussed the key results that are achieved through the results obtained in Chapter IV.

Our approach to tackle this problem was to first create separate blocks of modules and define each function associated with it and also take into account several signals which would be required as input or output from neighboring or far away modules.

Finally, each of these modules are instantiated to create a Top Level Module which performs the desired functionality at micro-architectural level. This module has been rigorously tested considering all the possibilities that might occur in a real time scenario but at the same time keeping it simple due to brevity and exploring the fundamental novelty of the proposed concept. It can be later scaled to include additional functionalities.

There are lot of scopes of improving the performance of proposed design by addition of Re-order buffering or Register renaming, which is discussed in Chapter VI.

# CHAPTER II

## Literature Survey

References:

[1.] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[2.] https://iis.ee.ethz.ch/~gmichi/asocd/addinfo/Out-of-Order_execution.pdf
[3.] Aravind Ruthala, “Superscalar Instruction Dispatch”, UCSB, Department of Electrical Engineering
[4.] http://www.cs.umd.edu/class/fall2001/cmsc411/projects/dynamic/tomasulo.html
[5.] R. Ghamari and A. Rajabzadeh, "FTDIS: A Fault Tolerant Dynamic Instruction Scheduling," *Dependability (DEPEND), 2010 Third International Conference on*, Venice, 2010, pp. 32-37

# CHAPTER III
## Section A
## Proposed Design and Architecture

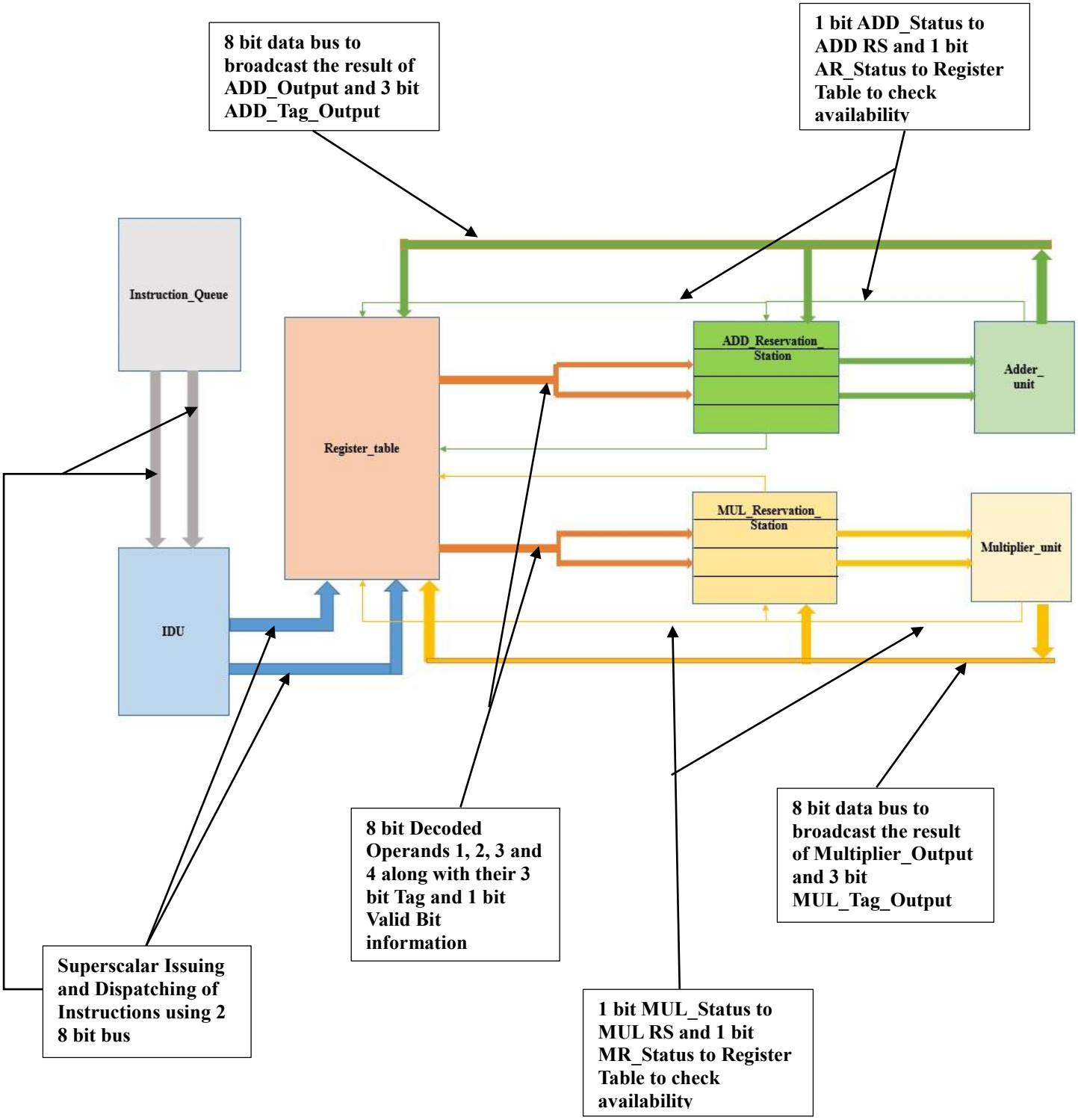



Fig.3.1. Proposed Design

The following are the Key Blocks of our Design:

1. Instruction Queue
2. Instruction Dispatch Unit
3. Register Table
4. Reservation Station – Adder and Multiplier
5. Adder Functional Unit
6. Multiplier Functional Unit

All of these blocks are programmed using Verilog Hardware Descriptive Language and tested as discussed in Chapter IV.

There are several micro-signals which are used while programming, some of them which are highly important are shown in Fig.3.1.

Since, we are implementing Superscalar Instruction Dispatching, our goal was to program such that the Fetch and Decode Stage take place in 5 half clock cycle.
**Hence, in 5 half clock cycles it is targeted to issue and decode 2 instructions.**

**As a result, Instruction Queue, Instruction Dispatch Unit and Register Table modules are operating in half clock cycles.**

## Section B

**Assumptions:**

- For our study we are assuming that the Adder Unit requires 2 clock cycles i.e. 4 half clock cycles and Multiply Unit requires 8 half clock cycles.
- Half clock period is selected to be 100 ns.
- While testing programs in proposed design, we assume the design has 1 Multiplier and 1 Adder Unit.

**Measuring Performance and Calculations:**

In order to measure the performance, the CPI is calculated for the proposed design and compared with traditional architecture and in-order processor.

We have also calculated % in Performance Improvement as well as Speedup obtained by using the proposed design model and discussed elaborately in Chapter V.

The following are the formula used:

$$\textbf{Speedup} = \frac{CPI_{in-order\ \ or\ Traditional\ Tomasulo}}{CPI_{Modified\ Tomasulo}}$$

$$\textbf{\% Performance Improvement} = \frac{|(CPI_{in-order\ \ or\ Traditional\ Tomasulo}) - (CPI_{Modified\ Tomasulo})|}{CPI_{Modified\ Tomasulo}} \text{ x } 100$$

# CHAPTER IV
# FUNCTIONAL BLOCKS

## Section A - Instruction Queue:

### (A.1) Block Diagram:

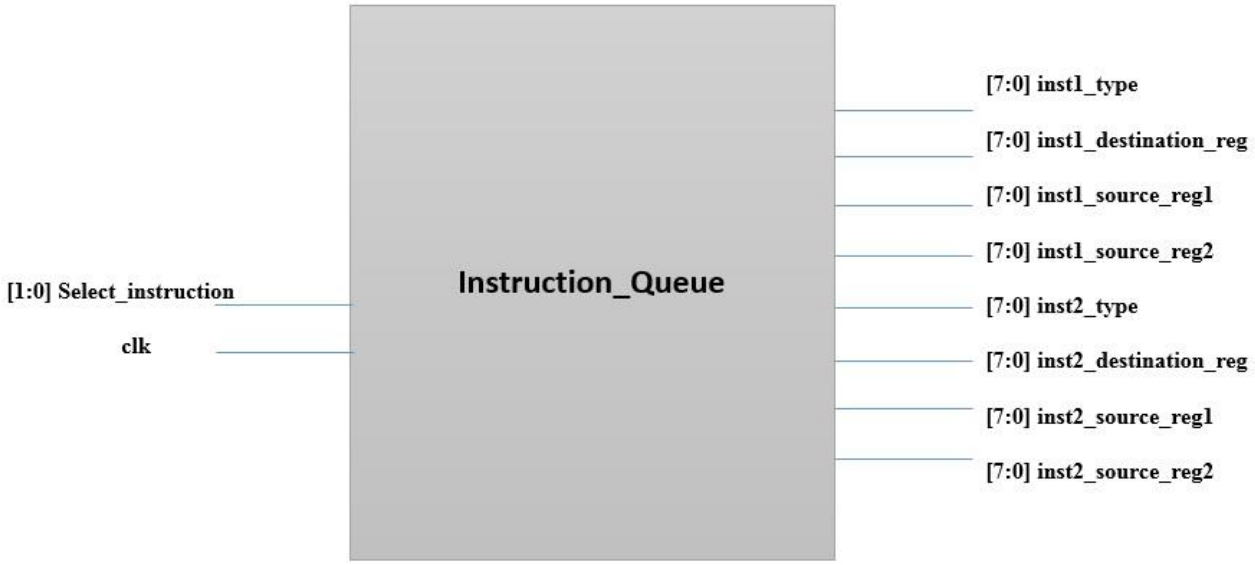



Fig.4.1: Instruction Queue Block

### (A.2) Description:

The program required to be executed is stored in this block where each instruction is stored in the form of a queue so that sequentially each and every instruction is issued. Since, we have proposed to implement superscalar, every clock cycle 2 instructions are fetched together. The inputs and outputs to the block are described in the Fig. 4.1. The block has an internal program counter to monitor the instructions. The results of this block as per the test bench mentioned in this section is shown in Fig.4.2.

### (A.3) Verilog Code:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module Instruction_Queue(
inst1_type,
inst1_destination_reg,
inst1_source_reg1,
inst1_source_reg2,
inst2_type,
inst2_destination_reg,
inst2_source_reg1,
inst2_source_reg2,
clock_count,
select_instruction,
clk);
```

```
output [7:0] inst1_type;
output [7:0] inst1_destination_reg;
output [7:0] inst1_source_reg1;
output [7:0] inst1_source_reg2;

output [7:0] inst2_type;
output [7:0] inst2_destination_reg;
output [7:0] inst2_source_reg1;
output [7:0] inst2_source_reg2;

output reg [4:0] clock_count = 5'b0;

//input [7:0] program_counter;
input clk;
input [1:0] select_instruction;

reg [7:0] inst1_type;
reg [7:0] inst1_destination_reg;
reg [7:0] inst1_source_reg1;
reg [7:0] inst1_source_reg2;

reg [7:0] inst2_type;
reg [7:0] inst2_destination_reg;
reg [7:0] inst2_source_reg1;
reg [7:0] inst2_source_reg2;
reg [31:0] ROM_Code [1023:0];          // 4 KB of Program ROM

reg [7:0] program_counter;
wire clk;
wire [1:0] select_instruction;



// Program
initial
begin
program_counter = 8'b0;
ROM_Code [0] = {`MUL, `R2, `R0, `R1};
ROM_Code [1] = {`ADD, `R5, `R3, `R4};
ROM_Code [2] = {`ADD, `R8, `R6, `R7};
ROM_Code [3] = {`ADD, `R11, `R9, `R10};
ROM_Code [4] = {`MUL, `R14, `R12, `R13};
ROM_Code [5] = {`ADD, `R17, `R15, `R16};
//inst1_type = `MUL;
//inst2_type = `ADD;
end




always @(posedge clk)
begin


                        if (program_counter == 8'b0)
                        begin
                                        inst1_type = ROM_Code[program_counter][31:24];
                                        inst1_destination_reg = ROM_Code[program_counter][23:16];
                                        inst1_source_reg1 = ROM_Code[program_counter][15:8];
                                        inst1_source_reg2 = ROM_Code[program_counter][7:0];
```

```
                                        program_counter = program_counter + 1;

                        end

                        else
                        begin
                                case (select_instruction)

                                        2'b00 : program_counter = program_counter;

                                        2'b01 :   begin
                                                        inst1_type = ROM_Code[program_counter][31:24];
                                                        inst1_destination_reg = ROM_Code[program_counter][23:16];
                                                        inst1_source_reg1 = ROM_Code[program_counter][15:8];
                                                        inst1_source_reg2 = ROM_Code[program_counter][7:0];
                                                        program_counter = program_counter + 1;
                                                end

                                        2'b10 :   begin
                                                        inst1_type = ROM_Code[program_counter][31:24];
                                                        inst1_destination_reg = ROM_Code[program_counter][23:16];
                                                        inst1_source_reg1 = ROM_Code[program_counter][15:8];
                                                        inst1_source_reg2 = ROM_Code[program_counter][7:0];
                                                       program_counter = program_counter + 1;
                                                end

                                endcase
                        end
end


always @(negedge clk)
begin
                if(clock_count >= 5'b00010)
                begin
                        if (program_counter == 8'b1)
                        begin
                                        inst2_type = ROM_Code[program_counter][31:24];
                                        inst2_destination_reg = ROM_Code[program_counter][23:16];
                                        inst2_source_reg1 = ROM_Code[program_counter][15:8];
                                        inst2_source_reg2 = ROM_Code[program_counter][7:0];

                                        program_counter = program_counter + 1;
                        end

                        else
                        begin
                                case (select_instruction)

                                        2'b00 : program_counter = program_counter;

                                        2'b10 :   begin

                                                                inst2_type = ROM_Code[program_counter][31:24];
                                                                inst2_destination_reg =
ROM_Code[program_counter][23:16];
                                                                inst2_source_reg1 = ROM_Code[program_counter][15:8];
                                                                inst2_source_reg2 = ROM_Code[program_counter][7:0];

                                                                program_counter = program_counter + 1;
                                                                end

                                endcase
                        end
```

```
                end
end

always @(clk)
begin
        clock_count = clock_count + 1;
end

endmodule
```

## (A.4) Verilog Testbench:

```
module test_IQ;

        // In
        reg [1:0] select_instruction;
        reg clk;

        // Outputs
        wire [7:0] program_counter;
        wire [7:0] inst1_type;
        wire [7:0] inst1_destination_reg;
        wire [7:0] inst1_source_reg1;
        wire [7:0] inst1_source_reg2;
        wire [7:0] inst2_type;
        wire [7:0] inst2_destination_reg;
        wire [7:0] inst2_source_reg1;
        wire [7:0] inst2_source_reg2;

        // Instantiate the Unit Under Test (UUT)
        Instruction_Queue uut (
                .inst1_type(inst1_type),
                .inst1_destination_reg(inst1_destination_reg),
                .inst1_source_reg1(inst1_source_reg1),
                .inst1_source_reg2(inst1_source_reg2),
                .inst2_type(inst2_type),
                .inst2_destination_reg(inst2_destination_reg),
                .inst2_source_reg1(inst2_source_reg1),
                .inst2_source_reg2(inst2_source_reg2),
                .select_instruction(select_instruction),
                .clk(clk)
        );

        always #100 clk = ~clk;

        initial begin
                // Initialize Inputs
                //program_counter = 0;
                //select_instruction = 0;
                clk = 0;

                // Wait 100 ns for global reset to finish
                #100  select_instruction = 2'b00;

                #200  select_instruction = 2'b10;
                #200  select_instruction = 2'b10;

                #200        select_instruction = 2'b10;
                #200  select_instruction = 2'b10;
```

```
        // Add stimulus here
    end

endmodule
```

## (A.5) Simulation Result:

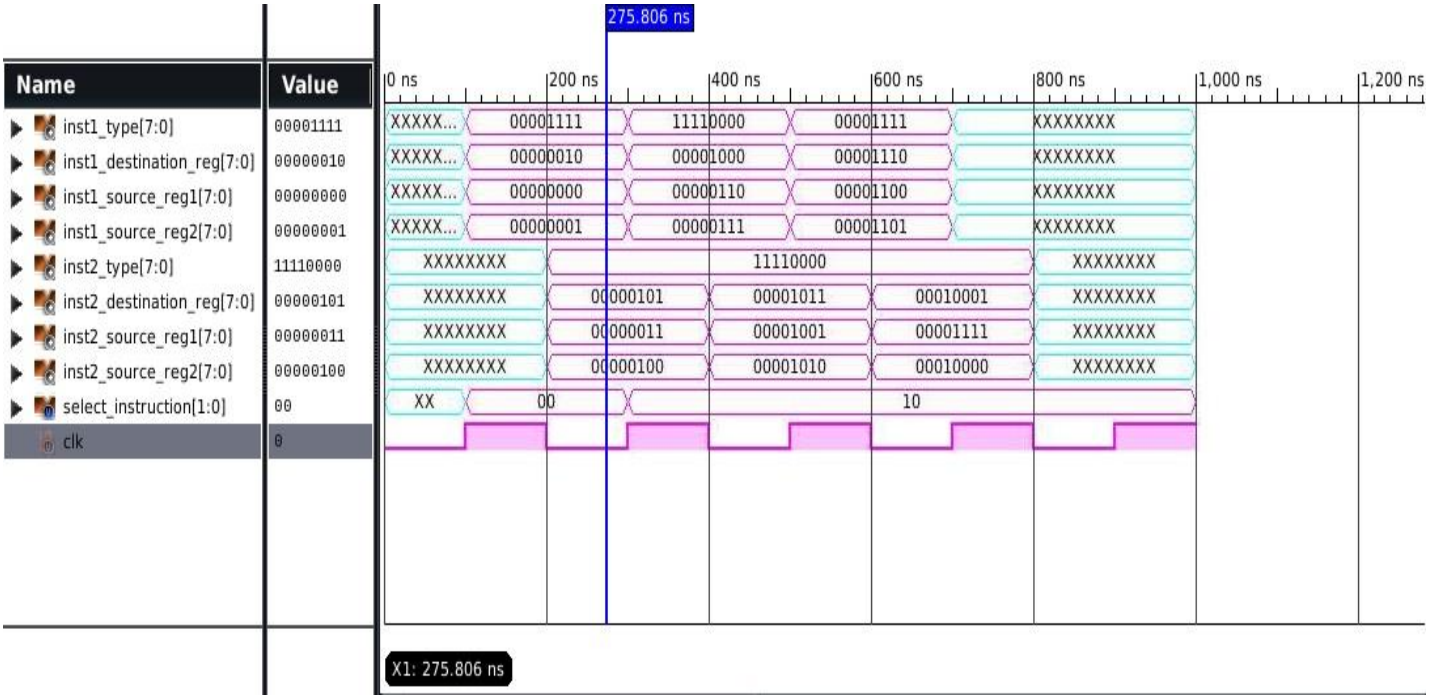



Fig.4.2: Instruction Queue Block- Waveforms

## Section B - Instruction Dispatch Unit:

### (B.1) Block Diagram:

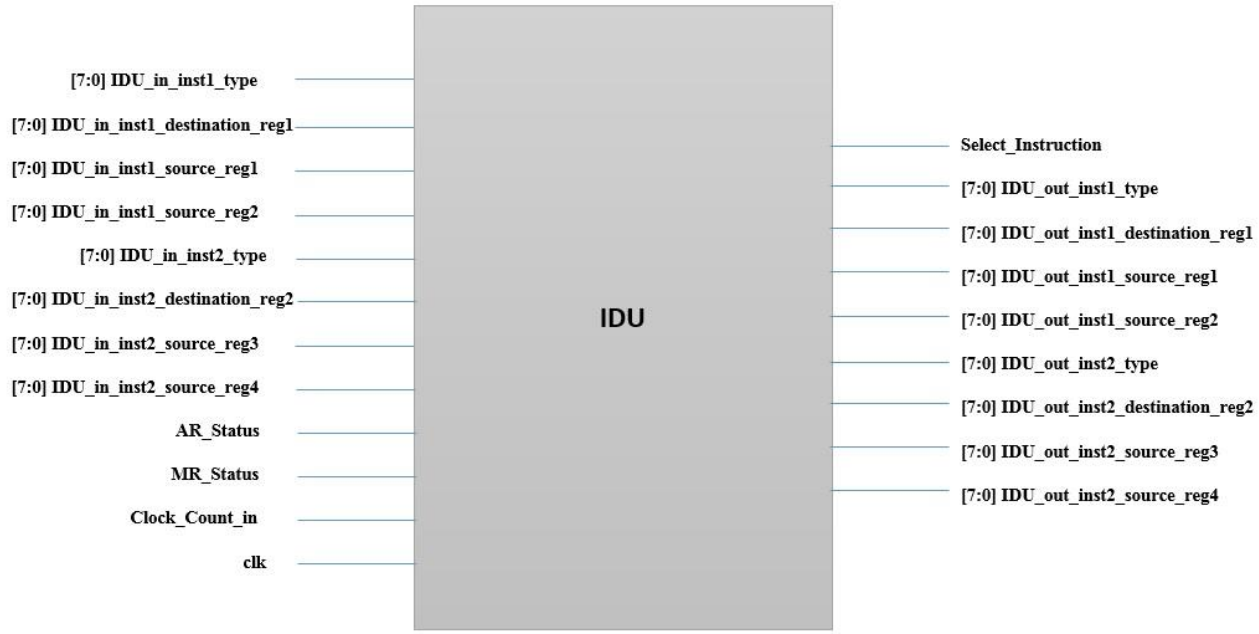



Fig.4.3: Instruction Dispatch Unit Block

**(B.2) Description:** The Instruction Dispatch Unit Dispatches two instructions every 5 clock cycles which combine fetch and decode stage. The AR_Status and MR_Status are the inputs from the Reservation Station which informs the IDU to dispatch further instructions in case any of the four slots are free. The output of IDU goes to the Register Table. The waveforms of this block are shown in Fig.4.4.

### (B.3)Verilog Code:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module IDU(
select_instruction,
IDU_out_inst1_type,
IDU_out_inst1_destination_reg1,
IDU_out_inst1_source_reg1,
IDU_out_inst1_source_reg2,
IDU_out_inst2_type,
IDU_out_inst2_destination_reg2,
IDU_out_inst2_source_reg3,
IDU_out_inst2_source_reg4,
IDU_in_inst1_type,
IDU_in_inst1_destination_reg1,
IDU_in_inst1_source_reg1,
IDU_in_inst1_source_reg2,
IDU_in_inst2_type,
IDU_in_inst2_destination_reg2,
IDU_in_inst2_source_reg3,
```

```
IDU_in_inst2_source_reg4,
AR_Status,
MR_Status,
clock_count_in,
clk);

output reg [1:0] select_instruction = 2'b10;

output reg [7:0] IDU_out_inst1_type;
output reg [7:0] IDU_out_inst1_destination_reg1;
output reg [7:0] IDU_out_inst1_source_reg1;
output reg [7:0] IDU_out_inst1_source_reg2;

output reg [7:0] IDU_out_inst2_type;
output reg [7:0] IDU_out_inst2_destination_reg2;
output reg [7:0] IDU_out_inst2_source_reg3;
output reg [7:0] IDU_out_inst2_source_reg4;


input wire [7:0] IDU_in_inst1_type;
input wire [7:0] IDU_in_inst1_destination_reg1;
input wire [7:0] IDU_in_inst1_source_reg1;
input wire [7:0] IDU_in_inst1_source_reg2;

input wire [7:0] IDU_in_inst2_type;
input wire [7:0] IDU_in_inst2_destination_reg2;
input wire [7:0] IDU_in_inst2_source_reg3;
input wire [7:0] IDU_in_inst2_source_reg4;

input wire AR_Status;
input wire MR_Status;
input wire [4:0] clock_count_in;
input wire clk;

//output integer pos_count = 0;
//output integer neg_count = 0;

always @(posedge clk)
begin

                        if(AR_Status == 1'b1 || MR_Status == 1'b1)
                                select_instruction = 2'b00;
                        else
                                begin
                                select_instruction = 2'b10;
                                if (IDU_in_inst1_type == `ADD)
                                        begin
                                                IDU_out_inst1_type = IDU_in_inst1_type;
                                                IDU_out_inst1_destination_reg1 = IDU_in_inst1_destination_reg1;
                                                IDU_out_inst1_source_reg1 = IDU_in_inst1_source_reg1;
                                                IDU_out_inst1_source_reg2 = IDU_in_inst1_source_reg2;
                                        end

                                else if (IDU_in_inst1_type == `MUL)
                                        begin
                                                IDU_out_inst2_type = IDU_in_inst1_type;
                                                IDU_out_inst2_destination_reg2 = IDU_in_inst1_destination_reg1;
                                                IDU_out_inst2_source_reg3 = IDU_in_inst1_source_reg1;
                                                IDU_out_inst2_source_reg4 = IDU_in_inst1_source_reg2;
                                        end
                                end
end

always @(negedge clk)
```

```
begin
                        if(AR_Status == 1'b1 || MR_Status == 1'b1)
                                select_instruction = 2'b00;

                        else
                        begin
                                if (IDU_in_inst2_type == `ADD)
                                        begin
                                                IDU_out_inst1_type = IDU_in_inst2_type;
                                                IDU_out_inst1_destination_reg1 = IDU_in_inst2_destination_reg2;
                                                IDU_out_inst1_source_reg1 = IDU_in_inst2_source_reg3;
                                                IDU_out_inst1_source_reg2 = IDU_in_inst2_source_reg4;
                                        end

                                else if (IDU_in_inst1_type == `MUL)
                                        begin
                                                IDU_out_inst2_type = IDU_in_inst2_type;
                                                IDU_out_inst2_destination_reg2 = IDU_in_inst2_destination_reg2;
                                                IDU_out_inst2_source_reg3 = IDU_in_inst2_source_reg3;
                                                IDU_out_inst2_source_reg4 = IDU_in_inst2_source_reg4;
                                        end
                        end


end

endmodule
```

## (B.4) Verilog Testbench:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module Test_IDU;

        // Inputs
        reg [7:0] IDU_in_inst1_type;
        reg [7:0] IDU_in_inst1_destination_reg1;
        reg [7:0] IDU_in_inst1_source_reg1;
        reg [7:0] IDU_in_inst1_source_reg2;
        reg [7:0] IDU_in_inst2_type;
        reg [7:0] IDU_in_inst2_destination_reg2;
        reg [7:0] IDU_in_inst2_source_reg3;
        reg [7:0] IDU_in_inst2_source_reg4;
        reg AR_Status;
        reg MR_Status;
  reg clk;
        // Outputs
        //wire [3:0] pos_count;
        //wire [3:0] neg_count;
        wire [1:0] select_instruction;
        wire [7:0] IDU_out_inst1_type;
        wire [7:0] IDU_out_inst1_destination_reg1;
        wire [7:0] IDU_out_inst1_source_reg1;
        wire [7:0] IDU_out_inst1_source_reg2;
        wire [7:0] IDU_out_inst2_type;
        wire [7:0] IDU_out_inst2_destination_reg2;
        wire [7:0] IDU_out_inst2_source_reg3;
        wire [7:0] IDU_out_inst2_source_reg4;

        // Instantiate the Unit Under Test (UUT)
        IDU uut (
                .select_instruction(select_instruction),
                .IDU_out_inst1_type(IDU_out_inst1_type),
```

```
		.IDU_out_inst1_destination_reg1(IDU_out_inst1_destination_reg1),
		.IDU_out_inst1_source_reg1(IDU_out_inst1_source_reg1),
		.IDU_out_inst1_source_reg2(IDU_out_inst1_source_reg2),
		.IDU_out_inst2_type(IDU_out_inst2_type),
		.IDU_out_inst2_destination_reg2(IDU_out_inst2_destination_reg2),
		.IDU_out_inst2_source_reg3(IDU_out_inst2_source_reg3),
		.IDU_out_inst2_source_reg4(IDU_out_inst2_source_reg4),
		.IDU_in_inst1_type(IDU_in_inst1_type),
		.IDU_in_inst1_destination_reg1(IDU_in_inst1_destination_reg1),
		.IDU_in_inst1_source_reg1(IDU_in_inst1_source_reg1),
		.IDU_in_inst1_source_reg2(IDU_in_inst1_source_reg2),
		.IDU_in_inst2_type(IDU_in_inst2_type),
		.IDU_in_inst2_destination_reg2(IDU_in_inst2_destination_reg2),
		.IDU_in_inst2_source_reg3(IDU_in_inst2_source_reg3),
		.IDU_in_inst2_source_reg4(IDU_in_inst2_source_reg4),
		.AR_Status(AR_Status),
		.MR_Status(MR_Status),
		.clk(clk)
	);

	initial clk = 0;

	always #100 clk = ~clk;

	initial begin
		// Initialize Inputs
		IDU_in_inst1_type = 0;
		IDU_in_inst1_destination_reg1 = 0;
		IDU_in_inst1_source_reg1 = 0;
		IDU_in_inst1_source_reg2 = 0;
		IDU_in_inst2_type = 0;
		IDU_in_inst2_destination_reg2 = 0;
		IDU_in_inst2_source_reg3 = 0;
		IDU_in_inst2_source_reg4 = 0;
		AR_Status = 1;
		MR_Status = 1;

		// Wait 100 ns for global reset to finish
#100 IDU_in_inst1_type = `MUL;
		IDU_in_inst1_destination_reg1 = `R2;
		IDU_in_inst1_source_reg1 = `R0;
		IDU_in_inst1_source_reg2 = `R1;
		AR_Status = 0;
		MR_Status = 0;

#100	IDU_in_inst2_type = `ADD;
		IDU_in_inst2_destination_reg2 = `R5;
		IDU_in_inst2_source_reg3 = `R3;
		IDU_in_inst2_source_reg4 = `R4;
		AR_Status = 0;
		MR_Status = 0;

#100 IDU_in_inst1_type = `ADD;                                          //300
		IDU_in_inst1_destination_reg1 = `R8;
		IDU_in_inst1_source_reg1 = `R6;
		IDU_in_inst1_source_reg2 = `R7;
		AR_Status = 0;
		MR_Status = 0;

#100	IDU_in_inst2_type = `ADD;                                       //400
		IDU_in_inst2_destination_reg2 = `R11;
		IDU_in_inst2_source_reg3 = `R9;
		IDU_in_inst2_source_reg4 = `R10;
		AR_Status = 0;
```

```
                    MR_Status = 0;

#100 IDU_in_inst1_type = `MUL;                                                  //500
                    IDU_in_inst1_destination_reg1 = `R14;
                    IDU_in_inst1_source_reg1 = `R12;
                    IDU_in_inst1_source_reg2 = `R13;
                    AR_Status = 0;
                    MR_Status = 0;

#100      IDU_in_inst2_type = `ADD;                                             //600
                    IDU_in_inst2_destination_reg2 = `R17;
                    IDU_in_inst2_source_reg3 = `R15;
                    IDU_in_inst2_source_reg4 = `R16;
                    AR_Status = 0;
                    MR_Status = 0;

end

endmodule
```

## (B.5) Simulation Results:

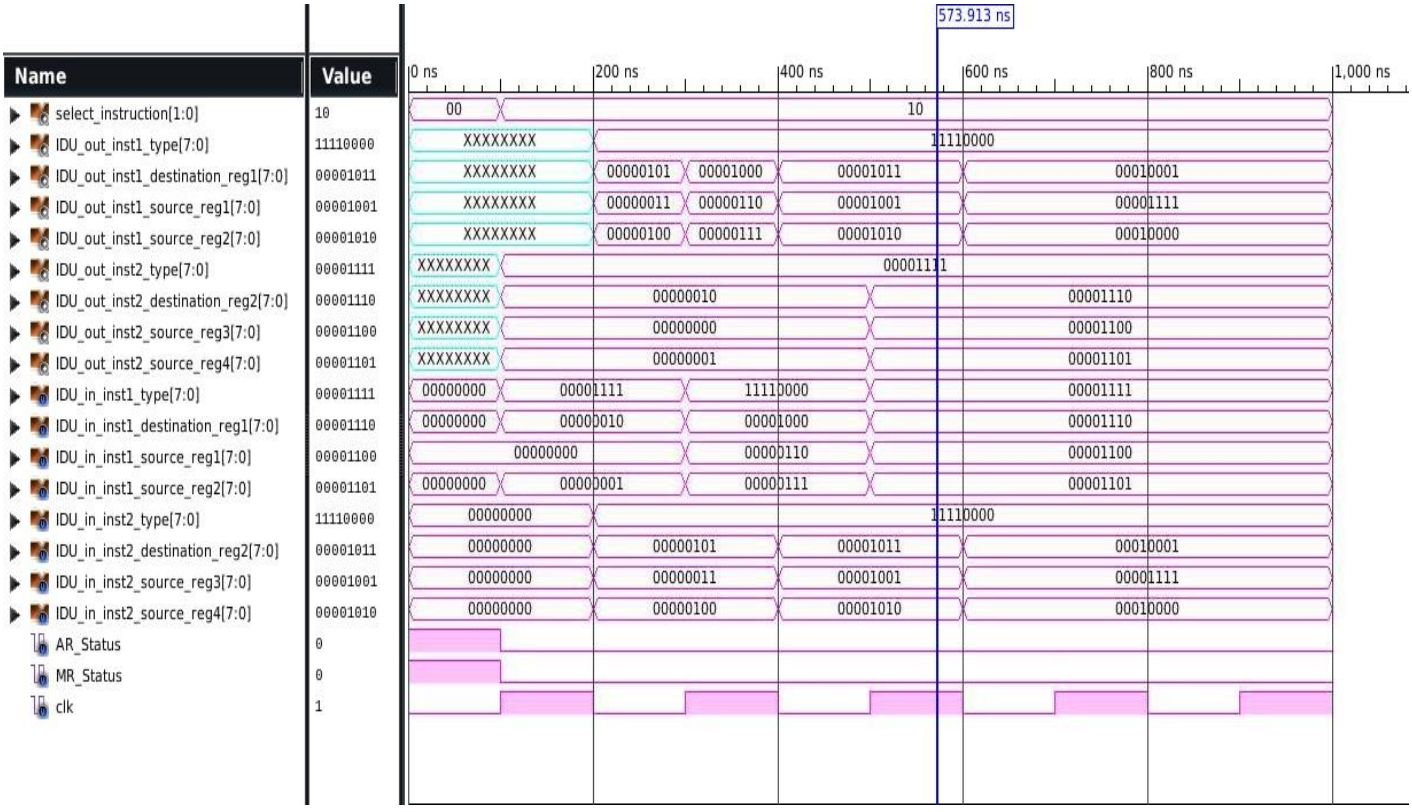



Fig.4.4 IDU-Waveforms

**Section C - Register Table:**

**(C.1) Block Diagram:**

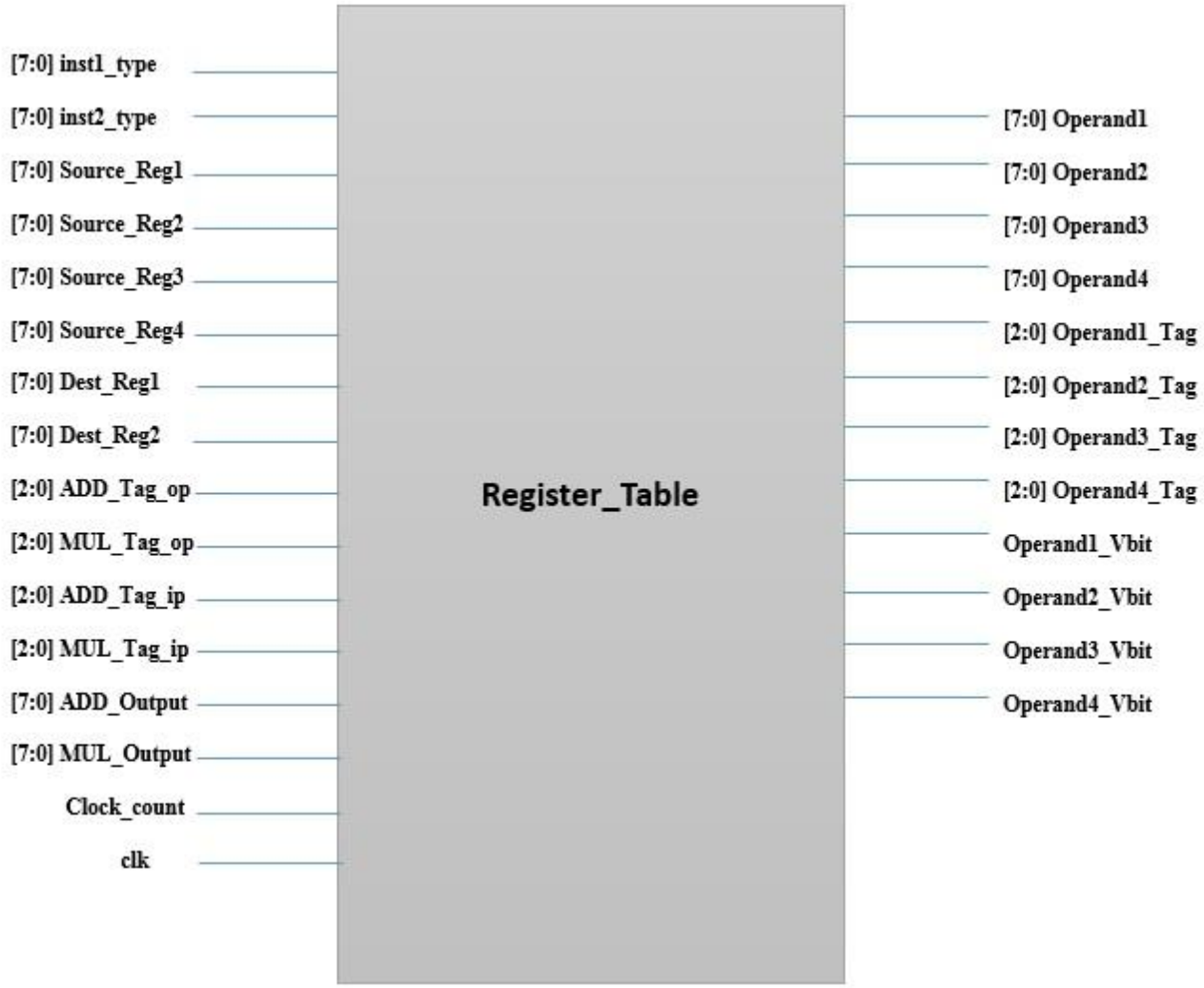



Fig.4.5 Register Table Block

**(C.2) Description:** The register table consist of updated list of Register values, tags and valid bits. The inputs to this block come from IDU which include the source register address and destination register address. It is the function of the Register Table to map the source and destination register and output the appropriate latest values of corresponding registers along with tag and valid bits as shown in Fig.4.5. The waveforms are shown in Fig.4.6 for the Testbench attached to this section.

**(C.3) Verilog Code:**

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

`define g 1'b1
`define h 3'bz

module Register_Table(
UP1,
UP2,
Operand1,
```

```
Operand2,
Operand3,
Operand4,
Operand1_Tag,
Operand2_Tag,
Operand3_Tag,
Operand4_Tag,
Operand1_Vbit,
Operand2_Vbit,
Operand3_Vbit,
Operand4_Vbit,
inst1_type,
inst2_type,
Source_Reg1,
Source_Reg2,
Source_Reg3,
Source_Reg4,
Dest_Reg1,
Dest_Reg2,
ADD_Tag_op,
MUL_Tag_op,
ADD_Tag_ip,
MUL_Tag_ip,
ADD_Output,
MUL_Output,
clock_count,
clk);

output reg [19:0] UP1;
output reg [19:0] UP2;
output reg [7:0] Operand1;
output reg [7:0] Operand2;
output reg [7:0] Operand3;
output reg [7:0] Operand4;
output reg [2:0] Operand1_Tag;
output reg [2:0] Operand2_Tag;
output reg [2:0] Operand3_Tag;
output reg [2:0] Operand4_Tag;
output reg Operand1_Vbit;
output reg Operand2_Vbit;
output reg Operand3_Vbit;
output reg Operand4_Vbit;

//output reg [7:0] var3;
//output reg [7:0] var6;

input wire [7:0] inst1_type;
input wire [7:0] inst2_type;

input wire [7:0] Source_Reg1;
input wire [7:0] Source_Reg2;
input wire [7:0] Source_Reg3;
input wire [7:0] Source_Reg4;
input wire [7:0] Dest_Reg1;
input wire [7:0] Dest_Reg2;
input wire [2:0] ADD_Tag_op;
input wire [2:0] MUL_Tag_op;
input wire [2:0] ADD_Tag_ip;
input wire [2:0] MUL_Tag_ip;
input wire [7:0] ADD_Output;
input wire [7:0] MUL_Output;
input wire [4:0] clock_count;
input wire clk;
```

```
reg [7:0] inst2_type_new;
reg [7:0] current_inst2_type;
reg [7:0] previous_inst2_type =`ADD;
reg [19:0] register_array [17:0];
integer I;

initial
begin
                register_array [0]  = {`R0, 1'b1, 3'bz, 8'b00000000};
                register_array [1]  = {`R1, 1'b1, 3'bz, 8'b00000001};
                register_array [2]  = {`R2, 1'b1, 3'bz, 8'b00000010};
                register_array [3]  = {`R3, 1'b1, 3'bz, 8'b00000011};
                register_array [4]  = {`R4, 1'b1, 3'bz, 8'b00000100};
                register_array [5]  = {`R5, 1'b1, 3'bz, 8'b00000101};
                register_array [6]  = {`R6, 1'b1, 3'bz, 8'b00000110};
                register_array [7]  = {`R7, 1'b1, 3'bz, 8'b00000111};
                register_array [8]  = {`R8, 1'b1, 3'bz, 8'b00001000};
                register_array [9]  = {`R9, 1'b1, 3'bz, 8'b00001001};
                register_array [10] = {`R10, 1'b1, 3'bz, 8'b00001010};
                register_array [11] = {`R11, 1'b1, 3'bz, 8'b00001011};
                register_array [12] = {`R12, 1'b1, 3'bz, 8'b00001100};
                register_array [13] = {`R13, 1'b1, 3'bz, 8'b00001101};
                register_array [14] = {`R14, 1'b1, 3'bz, 8'b00001110};
                register_array [15] = {`R15, 1'b1, 3'bz, 8'b00001111};
                register_array [16] = {`R16, 1'b1, 3'bz, 8'b00010000};
                register_array [17] = {`R17, 1'b1, 3'bz, 8'b00010001};
end


always @(clk)

begin
        for (i=0;i<=17;i=i+1)
                        begin
                        if (register_array[4][10:8]==ADD_Tag_op)
                                begin
                                register_array[4][7:0]=ADD_Output;
                                UP1=register_array[4][19:0];
                                end
        end
end

always @(clk)

begin
        for (i=0;i<=17;i=i+1)
                        begin
                        if (register_array[i][10:8]==MUL_Tag_op)
                                begin
                                register_array[i][7:0]=MUL_Output;
                                UP2=register_array[i][19:0];
                                end
        end
end

always @(clk)
begin


        if (inst1_type == `ADD && (clock_count == 5'b00101 || clock_count == 5'b00110 || clock_count == 5'b00111 || clock_count
== 5'b01001))
        begin

                for ( I = 0; i<=17; I = i+1)
```

```
                        begin
                                if (Source_Reg1 == register_array[i] [19:12])
                                begin
                                                Operand1 = register_array[i] [7:0];
                                                Operand1_Vbit= register_array[i] [11];
                                                Operand3 = `sanity;
                                end

                                if (register_array[i][19:12] == Dest_Reg1)
                                begin
                                                        register_array[i][11] = 1'b0;
                                                        register_array[i][10:8] = ADD_Tag_ip;
                                                        register_array[i][7:0] = 8'bz;
                                end

                                if (register_array[i][10:8] == ADD_Tag_op)
                                begin
                                                        register_array[i][11] = 1'b1;
                                                        register_array[i][7:0] = ADD_Output;
                                                        register_array[i][10:8] = 3'bz;
                                end
                        end

                for ( I = 0; i<=17; I = i+1)
                        begin
                                if (Source_Reg2 == register_array[i] [19:12])
                                begin
                                                Operand2 = register_array[i] [7:0];
                                                Operand2_Vbit= register_array[i] [11];
                                                Operand4 = `sanity;
                                end
                        end
        end


        else if (inst2_type == `MUL && (clock_count == 5'b00100 || clock_count == 5'b01000 ))
        begin

                for ( I = 0; i<=17; I = i+1)
                        begin
                                if (Source_Reg3 == register_array[i] [19:12])
                                begin
                                                Operand3 = register_array[i] [7:0];
                                                Operand3_Vbit= register_array[i] [11];
                                                Operand1 = `sanity;
                                end

                                if (Source_Reg4 == register_array[i] [19:12])
                                begin
                                                Operand4 = register_array[i] [7:0];
                                                Operand4_Vbit= register_array[i] [11];
                                                Operand2 = `sanity;
                                end
                        end
        end

end

endmodule
```

## (C.4) Verilog Testbench:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module Reg_Table_test;

	// Inputs
	reg [7:0] inst1_type;
	reg [7:0] inst2_type;
	reg [7:0] Source_Reg1;
	reg [7:0] Source_Reg2;
	reg [7:0] Source_Reg3;
	reg [7:0] Source_Reg4;
	reg [7:0] Dest_Reg1;
	reg [7:0] Dest_Reg2;
	reg [2:0] ADD_Tag_op;
	reg [2:0] MUL_Tag_op;
	reg [2:0] ADD_Tag_ip;
	reg [2:0] MUL_Tag_ip;
	reg [7:0] ADD_Output;
	reg [7:0] MUL_Output;
	reg [4:0] clock_count;
	reg clk;

	// Outputs
	//wire [19:0] UP1;
	//wire [19:0] UP2;
	wire [7:0] Operand1;
	wire [7:0] Operand2;
	wire [7:0] Operand3;
	wire [7:0] Operand4;
	wire [2:0] Operand1_Tag;
	wire [2:0] Operand2_Tag;
	wire [2:0] Operand3_Tag;
	wire [2:0] Operand4_Tag;
	wire Operand1_Vbit;
	wire Operand2_Vbit;
	wire Operand3_Vbit;
	wire Operand4_Vbit;

	// Instantiate the Unit Under Test (UUT)
	Register_Table uut (
		//.UP1(UP1),
		//.UP2(UP2),
		.Operand1(Operand1),
		.Operand2(Operand2),
		.Operand3(Operand3),
		.Operand4(Operand4),
		.Operand1_Tag(Operand1_Tag),
		.Operand2_Tag(Operand2_Tag),
		.Operand3_Tag(Operand3_Tag),
		.Operand4_Tag(Operand4_Tag),
		.Operand1_Vbit(Operand1_Vbit),
		.Operand2_Vbit(Operand2_Vbit),
		.Operand3_Vbit(Operand3_Vbit),
		.Operand4_Vbit(Operand4_Vbit),
		.inst1_type(inst1_type),
		.inst2_type(inst2_type),
		.Source_Reg1(Source_Reg1),
		.Source_Reg2(Source_Reg2),
		.Source_Reg3(Source_Reg3),
		.Source_Reg4(Source_Reg4),
		.Dest_Reg1(Dest_Reg1),
		.Dest_Reg2(Dest_Reg2),
```

```
        .ADD_Tag_op(ADD_Tag_op),
        .MUL_Tag_op(MUL_Tag_op),
        .ADD_Tag_ip(ADD_Tag_ip),
        .MUL_Tag_ip(MUL_Tag_ip),
        .ADD_Output(ADD_Output),
        .MUL_Output(MUL_Output),
        .clock_count(clock_count),
        .clk(clk)
);

initial clk = 0;

always #100 clk = ~clk;

initial begin
        // Initialize Inputs
        inst1_type = 0;
        inst2_type = 0;
        Source_Reg1 = 0;
        Source_Reg2 = 0;
        Source_Reg3 = 0;
        Source_Reg4 = 0;
        Dest_Reg1 = 0;
        Dest_Reg2 = 0;
        ADD_Tag_op = 0;
        MUL_Tag_op = 0;
        ADD_Tag_ip = 0;
        MUL_Tag_ip = 0;
        ADD_Output = 0;
        MUL_Output = 0;


        // Wait 100 ns for global reset to finish
        #300  inst2_type = `MUL;
                        Source_Reg3 = `R0;
                        Source_Reg4 = `R1;
                        Dest_Reg2 = `R2;
                        clock_count = 5'b00100;

        #100       inst1_type = `ADD;
                        Source_Reg1 = `R3;
                        Source_Reg2 = `R4;
                        Dest_Reg1 = `R5;
                        ADD_Tag_ip = `b;
                        MUL_Tag_ip = `w;
                        clock_count = 5'b00101;

        #100  inst1_type = `ADD;
                        Source_Reg1 = `R6;
                        Source_Reg2 = `R7;
                        Dest_Reg1 = `R8;
            clock_count = 5'b00110;
        #100       inst1_type = `ADD;
                        Source_Reg1 = `R9;
                        Source_Reg2 = `R10;
                        Dest_Reg1 = `R11;
                        ADD_Tag_ip = `b;
                        MUL_Tag_ip = `w;
                        clock_count = 5'b00111;

        #100  inst2_type = `MUL;
                        Source_Reg3 = `R12;
                        Source_Reg4 = `R13;
                        Dest_Reg2 = `R14;
```

```
                    clock_count = 5'b01000;

        #100    inst1_type = `ADD;
                    Source_Reg1 = `R17;
                    Source_Reg2 = `R15;
                    Dest_Reg1 = `R16;
                    ADD_Tag_ip = `b;
                    MUL_Tag_ip = `w;
                    clock_count = 5'b01001;

    end

endmodule
```

## (C.5) Simulation Result:

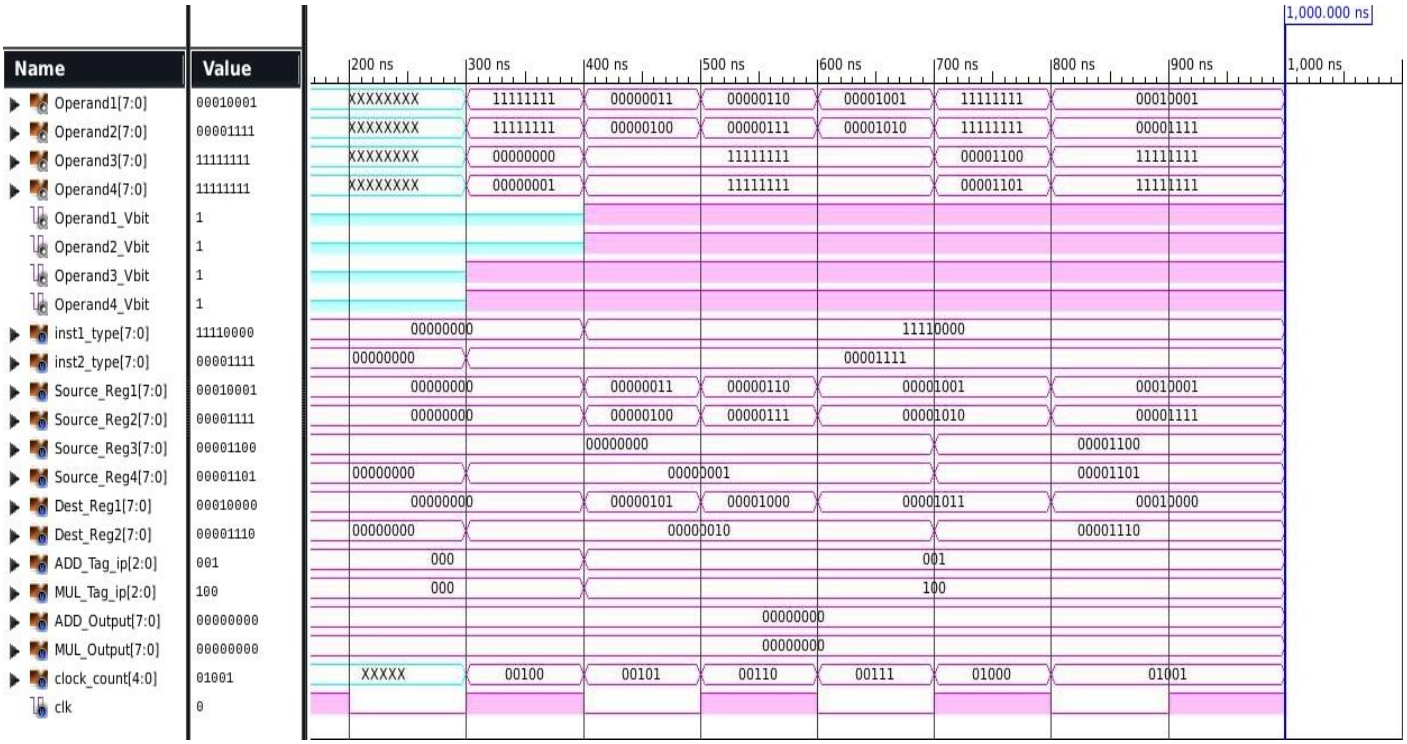



Fig.4.6 Register Table -Waveforms

## Section D - Adder Reservation Station:

### (D.1) Block Diagram:

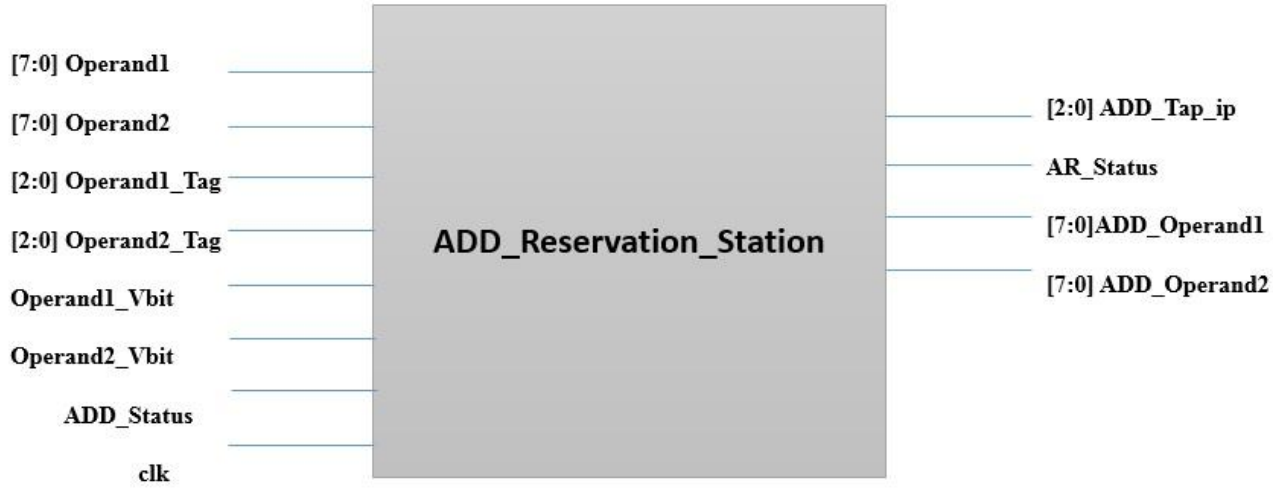



Fig.4.7: ADD Reservation Station Block

**(D.2) Description:** The ADD_Status is the input of this block, which comes from Adder Unit. It is to inform that the Adder Unit is busy or ideal waiting for an operation. We have selected the convention that when the Adder Unit is busy, ADD_Status is 1 orelse it is 0. The updated operands 1 and 2 come from Register Table along with tag and valid bits. The outputs of this block go to the Adder Unit and AR_Status is used to input IDU as described in earlier sections. The other signals are defined as described in Fig.4.7. and its waveform are shown in Fig.4.8.

### (D.3) Verilog Code:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module ADD_Reservation_Station(
ADD_Tag_ip,
AR_Status,
ADD_Operand1,
ADD_Operand2,
Operand1,
Operand2,
Operand1_Tag,
Operand2_Tag,
Operand1_Vbit,
Operand2_Vbit,
ADD_Status,
clk );

output reg [2:0] ADD_Tag_ip;
output reg AR_Status = 1'b0;
output reg[7:0] ADD_Operand1;
output reg[7:0] ADD_Operand2;

input wire [7:0] Operand1;
input wire [7:0] Operand2;
```

```
input wire [2:0] Operand1_Tag;
input wire [2:0] Operand2_Tag;
input wire Operand1_Vbit;
input wire Operand2_Vbit;
input wire ADD_Status;
input wire clk;

integer count=0;
reg [1:0] release_pointer = 2'b0;

integer I;
//integer select1=0;
reg [23:0] ADD_Res_Station [3:0];

initial
begin

        ADD_Res_Station[0][23:0] = 24'b011111111111011111111111;
        ADD_Res_Station[1][23:0] = 24'b011111111111011111111111;
        ADD_Res_Station[2][23:0] = 24'b011111111111011111111111;
        ADD_Res_Station[3][23:0] = 24'b011111111111011111111111;

end

always @(clk)
                                                        // To check which row of Res Station is free
begin
        if(ADD_Res_Station[0][23:0] == 24'b011111111111011111111111)
        begin
                AR_Status = 1'b0;
                count = 1;
        end

        else if(ADD_Res_Station[1][23:0] == 24'b011111111111011111111111)
                        begin
                                AR_Status = 1'b0;
                                count = 2;
                        end

                        else if(ADD_Res_Station[2][23:0] == 24'b011111111111011111111111)
                                        begin
                                                AR_Status = 1'b0;
                                                count = 3;
                                        end

                                        else if(ADD_Res_Station[3][23:0] == 24'b011111111111011111111111)
                                                        begin
                                                                AR_Status = 1'b0;
                                                                count = 4;
                                                        end

                                                        else AR_Status = 1'b1;

end

always @(clk)
                                                                // To fill up data in the reservation station
begin
        if((count == 1) && (Operand1 != `sanity && Operand2 != `sanity))
        begin
                ADD_Res_Station[0][23:0]={Operand1_Vbit,Operand1_Tag,Operand1,Operand2_Vbit,Operand2_Tag,Operand2};

        end
```

```
        else if((count == 2) && (Operand1 != `sanity && Operand2 != `sanity))
                        begin

        ADD_Res_Station[1][23:0]={Operand1_Vbit,Operand1_Tag,Operand1,Operand2_Vbit,Operand2_Tag,Operand2};

                        end

                        else if ((count == 3) && (Operand1 != `sanity && Operand2 != `sanity))
                                        begin

        ADD_Res_Station[2][23:0]={Operand1_Vbit,h,Operand1,Operand2_Vbit,Operand2_Tag,Operand2};

                                        end

                                        else if ((count == 4) && (Operand1 != `sanity && Operand2 != `sanity))
                                        begin

        ADD_Res_Station[3][23:0]={Operand1_Vbit,Operand1_Tag,Operand1,Operand2_Vbit,Operand2_Tag,Operand2};

                                        end
end




always @(clk)
                                                                                        // To issue operands to adder functional unit
begin
        if(ADD_Status==1'b0)
        begin
                if (ADD_Res_Station[0][23] == 1'b1 && ADD_Res_Station[0][11] == 1'b1 && release_pointer == 2'b0)
                begin
                        ADD_Operand1=ADD_Res_Station[0][19:12];
                        ADD_Operand2=ADD_Res_Station[0][7:0];
                        ADD_Tag_ip=`a;
                        release_pointer = release_pointer + 2'b01;

                end

                else if (ADD_Res_Station[1][23] == 1'b1 && ADD_Res_Station[1][11] == 1'b1 && release_pointer == 2'b01)
                begin
                        ADD_Operand1=ADD_Res_Station[1][19:12];
                        ADD_Operand2=ADD_Res_Station[1][7:0];
                        ADD_Tag_ip=`b;
                        release_pointer = release_pointer + 2'b01;

                end

                else if (ADD_Res_Station[2][23] == 1'b1 && ADD_Res_Station[2][11] == 1'b1 && release_pointer == 2'b10)
                begin
                        ADD_Operand1=ADD_Res_Station[2][19:12];
                        ADD_Operand2=ADD_Res_Station[2][7:0];
                        ADD_Tag_ip=`c;
                        release_pointer = release_pointer + 2'b01;

                end

                else if (ADD_Res_Station[3][23] == 1'b1 && ADD_Res_Station[3][11] == 1'b1 && release_pointer == 2'b11)
                begin
                        ADD_Operand1=ADD_Res_Station[3][19:12];
```

```
                        ADD_Operand2=ADD_Res_Station[3][7:0];
                        ADD_Tag_ip=`d;
                        release_pointer = 2'b0;

                end

                else
                begin
                        ADD_Operand1=`sanity;
                        ADD_Operand2=`sanity;
                end
        end
        else if (ADD_Status==1'b1)
                        begin
                                ADD_Operand1=`sanity;
                                ADD_Operand2=`sanity;
                        end
end

endmodule
```

## (D.4) Verilog Testbench:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"


module ADD_Res_test;

        // Inputs
        reg [7:0] Operand1;
        reg [7:0] Operand2;
        reg [2:0] Operand1_Tag;
        reg [2:0] Operand2_Tag;
        reg Operand1_Vbit;
        reg Operand2_Vbit;
        reg ADD_Status;
        reg clk;

        // Outputs
        //wire [1:0] release_pointer;
        wire [2:0] ADD_Tag_ip;
        wire AR_Status;
        wire [7:0] ADD_Operand1;
        wire [7:0] ADD_Operand2;
        wire [2:0] select;

        // Instantiate the Unit Under Test (UUT)
        ADD_Reservation_Station uut (
                //.release_pointer(release_pointer),
                /*.dummy_out(dummy_out),
                .dummy_out1(dummy_out1),
                .dummy_bit1(dummy_bit1),
                .dummy_out2(dummy_out2),
                .dummy_bit2(dummy_bit2),
                .dummy_out3(dummy_out3),
                .dummy_bit3(dummy_bit3),
                .dummy_out4(dummy_out4),
                .dummy_bit4(dummy_bit4),*/
                .ADD_Tag_ip(ADD_Tag_ip),
```

```
        .AR_Status(AR_Status),
        .ADD_Operand1(ADD_Operand1),
        .ADD_Operand2(ADD_Operand2),
        .Operand1(Operand1),
        .Operand2(Operand2),
        .Operand1_Tag(Operand1_Tag),
        .Operand2_Tag(Operand2_Tag),
        .Operand1_Vbit(Operand1_Vbit),
        .Operand2_Vbit(Operand2_Vbit),
        .ADD_Status(ADD_Status),
        .clk(clk)
);

initial clk = 0;

always #100 clk = ~clk;

initial
begin

#400  ADD_Status= 1'b1;                                 //400
    Operand1 = `sanity;
                Operand2 = `sanity;
                Operand1_Tag = 3'bz;
                Operand2_Tag = 3'bz;
                Operand1_Vbit = 1'b0;
                Operand2_Vbit = 1'b0;

#100  ADD_Status= 1'b0;                                 //500
    Operand1 = `R3;
                Operand2 = `R4;
                Operand1_Tag = 3'bz;
                Operand2_Tag = 3'bz;
                Operand1_Vbit = 1'b1;
                Operand2_Vbit = 1'b1;

#100  ADD_Status= 1'b0;                         //600
    Operand1 = `R6;
                Operand2 = `R7;
                Operand1_Tag = 3'bz;
                Operand2_Tag = 3'bz;
                Operand1_Vbit = 1'b1;
                Operand2_Vbit = 1'b1;

#100  ADD_Status= 1'b0;                                 //700
    Operand1 = `R9;
                Operand2 = `R10;
                Operand1_Tag = 3'bz;
                Operand2_Tag = 3'bz;
                Operand1_Vbit = 1'b1;
                Operand2_Vbit = 1'b1;

#100  ADD_Status= 1'b0;                                 //800
    Operand1 = `sanity;
                Operand2 = `sanity;
                Operand1_Tag = 3'bz;
                Operand2_Tag = 3'bz;
                Operand1_Vbit = 1'b0;
                Operand2_Vbit = 1'b0;

#100  ADD_Status= 1'b0;                                 //900
    Operand1 = `R15;
                Operand2 = `R16;
                Operand1_Tag = 3'bz;
                Operand2_Tag = 3'bz;
                Operand1_Vbit = 1'b1;
                Operand2_Vbit = 1'b1;
```

```
#400  ADD_Status= 1'b0;                    //1300

#100  ADD_Status= 1'b1;                    //1400

#100  ADD_Status= 1'b0;                    //900

#100  ADD_Status= 1'b1;                    //900

#100  ADD_Status= 1'b0;                    //900

#100  ADD_Status= 1'b1;                    //900
end

endmodule
```

## (D.5) Simulation Results:

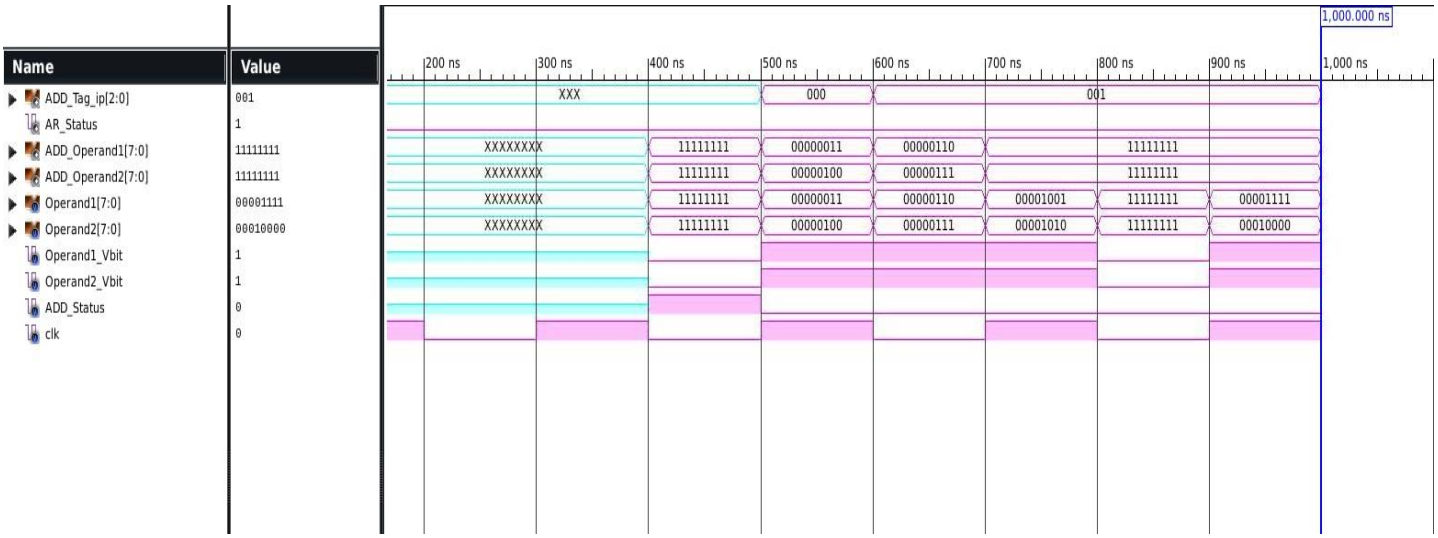



Fig.4.8 Adder Reservation Station - Waveforms

## Section E - Adder Unit

### (E.1) Block Diagram:


Fig.4.9: Adder Unit Block

**(E.2) Description:** The Adder Unit performs the operation and the output is broadcasted to the Register Table as well as the Register Table along with Tag output which is basically the tag associated with the reservation slot. The Adder Unit Block is shown in Fig.4.9 and its waveforms for Testbench in this section are shown in Fig.4.10.

### (E.3) Verilog Code:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module Adder_Unit(
count,
ADD_Status,
 ADD_Output,
 ADD_Tag_op,
 ADD_Tag_ip,
 ADD_Operand1,
 ADD_Operand2,
 clk
 );

output reg ADD_Status;
output reg [7:0] ADD_Output;
output reg [2:0] ADD_Tag_op;

output reg [2:0] count=3'b0;
reg [2:0] I;

input wire [2:0] ADD_Tag_ip;
input wire [7:0] ADD_Operand1;
input wire [7:0] ADD_Operand2;
input wire clk;
```

```
reg [7:0] Source_array1 [3:0];
reg [7:0] Source_array2 [3:0];
reg [7:0] ADD_Buffer;

always @(clk)
        begin
                        case (count)

                        3'b000 :        begin
                                        ADD_Status = 1'b1;
                                        count = count+1;
                                        ADD_Output=8'bz;
                                        ADD_Tag_op=3'bz;
                                        end

                        3'b001 :        begin
                                        ADD_Status = 1'b0;
                                        ADD_Output = ADD_Operand1 + ADD_Operand2;
                                        ADD_Output = ADD_Buffer;
                                        ADD_Tag_op = ADD_Tag_ip;
                                        count = count+1;

                                        end

                        3'b010 :        begin

                                        ADD_Status = 1'b1;
                                        ADD_Output=8'bz;
                                        ADD_Tag_op=3'bz;
                                        count = count+1;
                                        end

                        3'b011 :        begin
                                         ADD_Buffer = ADD_Operand1 + ADD_Operand2;
                                        count = 3'b0;
                                        ADD_Status = 1'b1;
                                        ADD_Output=8'bz;
                                            ADD_Tag_op=3'bz;
                                        end

                        endcase
        end


endmodule
```

## (E.4) Verilog Testbench:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"
module Adder_test;

        // Inputs
        reg [2:0] ADD_Tag_ip;
        reg [7:0] ADD_Operand1;
        reg [7:0] ADD_Operand2;
        reg clk;

        // Outputs
        wire [2:0] count;
        wire ADD_Status;
        wire [7:0] ADD_Output;
        wire [2:0] ADD_Tag_op;
```

```
	// Instantiate the Unit Under Test (UUT)
	Adder_Unit uut (
		.count(count),
		.ADD_Status(ADD_Status),
		.ADD_Output(ADD_Output),
		.ADD_Tag_op(ADD_Tag_op),
		.ADD_Tag_ip(ADD_Tag_ip),
		.ADD_Operand1(ADD_Operand1),
		.ADD_Operand2(ADD_Operand2),
		.clk(clk)
	);

	initial clk = 0;

	always #100 clk = ~clk;

	initial begin
		// Initialize Inputs
		#600	ADD_Tag_ip = `a;
				ADD_Operand1 = `R3;
				ADD_Operand2= `R4;

		#500	ADD_Tag_ip = `b;
				ADD_Operand1 = `R6;
				ADD_Operand2= `R7;


		#800	ADD_Tag_ip = `c;
				ADD_Operand1 = `R9;
				ADD_Operand2= `R10;


		#800	ADD_Tag_ip = `c;
				ADD_Operand1 = `sanity;
				ADD_Operand2= `sanity;

		#100	 ADD_Tag_ip = `d;
				ADD_Operand1 = `R15;
				ADD_Operand2= `R16;

		end
endmodule
```

**(E.5) Simulation Result:**

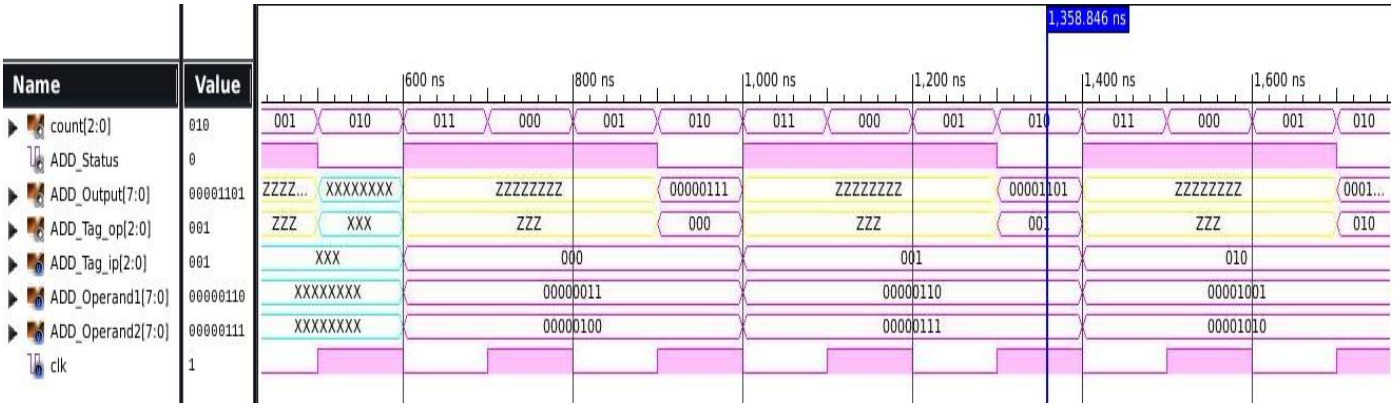



Fig.4.10 (a): Adder Unit Waveforms

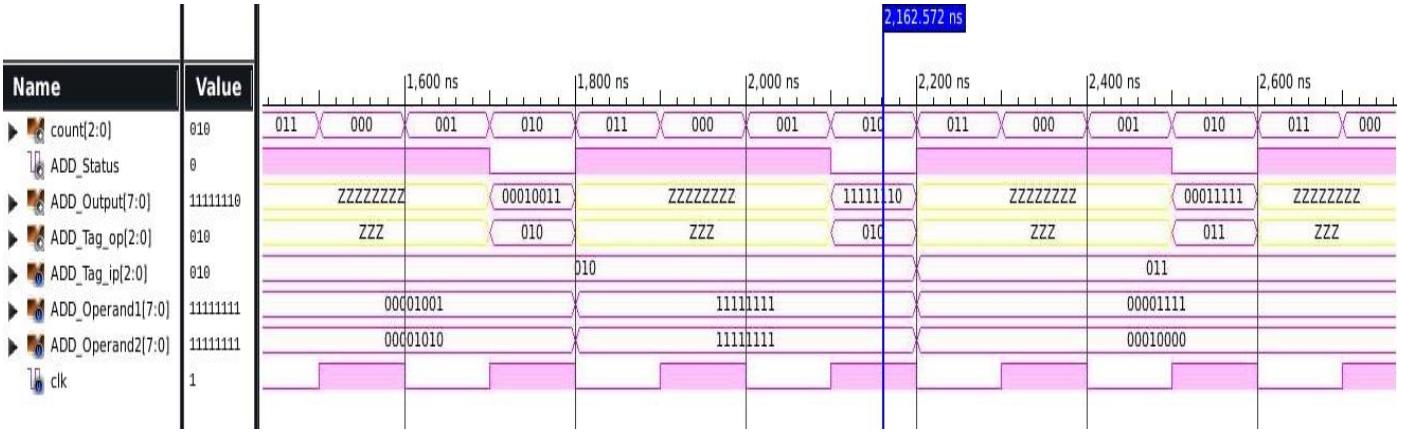



Fig.4.10 (b): Adder Unit Waveforms

## Section F - Multiplier Reservation Station:

### (F.1) Block Diagram:


Fig.4.11: Multiplier Reservation Station Block

**(F.2) Description:** The MUL_Status is the input of this block, which comes from Multiplier Unit. It is to inform that the Multiplier Unit is busy or ideal waiting for an operation. We have selected the convention that when the Multiplier Unit is busy, MUL_Status is 1 orelse it is 0. The updated operands 3 and 4 come from Register Table along with tag and valid bits. The outputs of this block go to the Multiplier Unit and MR_Status is used to input IDU as described in earlier sections. The other signals are defined as described in Fig.4.11. and its waveform are shown in Fig.4.12.

### (F.3) Verilog Code:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module MUL_Reservation_Station(
MUL_Tag_ip,
MR_Status,
MUL_Operand3,
MUL_Operand4,
Operand3,
Operand4,
Operand3_Tag,
Operand4_Tag,
Operand3_Vbit,
Operand4_Vbit,
MUL_Status,
clk );

output reg [2:0] MUL_Tag_ip;
output reg MR_Status = 1'b0;
output reg[7:0] MUL_Operand3;
```

```
output reg[7:0] MUL_Operand4;

input wire [7:0] Operand3;
input wire [7:0] Operand4;
input wire [2:0] Operand3_Tag;
input wire [2:0] Operand4_Tag;
input wire Operand3_Vbit;
input wire Operand4_Vbit;
input wire MUL_Status;
input wire clk;

integer count=0;
reg [1:0] release_pointer = 2'b0;

integer I;
reg [23:0] MUL_Res_Station [3:0];

initial
begin
        MUL_Res_Station[0][23:0] = 24'b011111111111011111111111;
        MUL_Res_Station[1][23:0] = 24'b011111111111011111111111;
        MUL_Res_Station[2][23:0] = 24'b011111111111011111111111;
        MUL_Res_Station[3][23:0] = 24'b011111111111011111111111;

end

always @(clk)
                                                                        // To check which row of Res Station is free
begin
        if(MUL_Res_Station[0][23:0] == 24'b011111111111011111111111)
        begin
                MR_Status = 1'b0;
                count = 1;
        end

        else if(MUL_Res_Station[1][23:0] == 24'b011111111111011111111111)
                        begin
                                MR_Status = 1'b0;
                                count = 2;
                        end

                        else if(MUL_Res_Station[2][23:0] == 24'b011111111111011111111111)
                                        begin
                                                MR_Status = 1'b0;
                                                count = 3;
                                        end

                                        else if(MUL_Res_Station[3][23:0] == 24'b011111111111011111111111)
                                                        begin
                                                                MR_Status = 1'b0;
                                                                count = 4;
                                                        end

                                                        else MR_Status = 1'b1;

end

always @(clk)
                                                                                // To fill up data in the reservation station
begin
        if((count == 1) && (Operand3 != `sanity && Operand4 != `sanity))
        begin
                MUL_Res_Station[0][23:0]={Operand3_Vbit,Operand3_Tag,Operand3,Operand4_Vbit,Operand4_Tag,Operand4};

        end

        else if((count == 2) && (Operand3 != `sanity && Operand4 != `sanity))
                        begin
```

```
		MUL_Res_Station[1][23:0]={Operand3_Vbit,Operand3_Tag,Operand3,Operand4_Vbit,Operand4_Tag,Operand4};

				end

				else if ((count == 3) && (Operand3 != `sanity && Operand4 != `sanity))
					begin

		MUL_Res_Station[2][23:0]={Operand3_Vbit,Operand3_Tag,Operand3,Operand4_Vbit,Operand4_Tag,Operand4};

					end

					else if ((count == 4) && (Operand3 != `sanity && Operand4 != `sanity))
					begin

		MUL_Res_Station[3][23:0]={Operand3_Vbit,Operand3_Tag,Operand3,Operand4_Vbit,Operand4_Tag,Operand4};

					end
end

always @(clk)
                                                                                    // To issue operands to MULer functional unit
begin
	if(MUL_Status==1'b0)
	begin
		if (MUL_Res_Station[0][23] == 1'b1 && MUL_Res_Station[0][11] == 1'b1 && release_pointer == 2'b0)
		begin
			MUL_Operand3=MUL_Res_Station[0][19:12];
			MUL_Operand4=MUL_Res_Station[0][7:0];
			//MUL_Operand3 = 8'b10101010;
			//MUL_Operand4 = 8'b01010101;
			MUL_Tag_ip=`w;
			release_pointer = release_pointer + 2'b01;

		end

		else if (MUL_Res_Station[1][23] == 1'b1 && MUL_Res_Station[1][11] == 1'b1 && release_pointer == 2'b01)
		begin
			MUL_Operand3=MUL_Res_Station[1][19:12];
			MUL_Operand4=MUL_Res_Station[1][7:0];
			MUL_Tag_ip=`x;
			release_pointer = release_pointer + 2'b01;

		end

		else if (MUL_Res_Station[2][23] == 1'b1 && MUL_Res_Station[2][11] == 1'b1 && release_pointer == 2'b10)
		begin
			MUL_Operand3=MUL_Res_Station[2][19:12];
			MUL_Operand4=MUL_Res_Station[2][7:0];
			MUL_Tag_ip=`y;
			release_pointer = release_pointer + 2'b01;

		end

		else if (MUL_Res_Station[3][23] == 1'b1 && MUL_Res_Station[3][11] == 1'b1 && release_pointer == 2'b11)
		begin
			MUL_Operand3=MUL_Res_Station[3][19:12];
			MUL_Operand4=MUL_Res_Station[3][7:0];
			MUL_Tag_ip=`z;
			release_pointer = 2'b0;

		end

		else
		begin
			MUL_Operand3=`sanity;
			MUL_Operand4=`sanity;
		end
```

```
		end
		else if (MUL_Status==1'b1)
				begin
					MUL_Operand3=`sanity;
					MUL_Operand4=`sanity;
				end
end

endmodule
```

## (F.4) Verilog Testbench:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module MUL_Res_test;

		// Inputs
		reg [7:0] Operand3;
		reg [7:0] Operand4;
		reg [2:0] Operand3_Tag;
		reg [2:0] Operand4_Tag;
		reg Operand3_Vbit;
		reg Operand4_Vbit;
		reg MUL_Status;
		reg clk;

		// Outputs
		wire [2:0] MUL_Tag_ip;
		wire MR_Status;
		wire [7:0] MUL_Operand3;
		wire [7:0] MUL_Operand4;

		// Instantiate the Unit Under Test (UUT)
		MUL_Reservation_Station uut (
			.MUL_Tag_ip(MUL_Tag_ip),
			.MR_Status(MR_Status),
			.MUL_Operand3(MUL_Operand3),
			.MUL_Operand4(MUL_Operand4),
			.Operand3(Operand3),
			.Operand4(Operand4),
			.Operand3_Tag(Operand3_Tag),
			.Operand4_Tag(Operand4_Tag),
			.Operand3_Vbit(Operand3_Vbit),
			.Operand4_Vbit(Operand4_Vbit),
			.MUL_Status(MUL_Status),
			.clk(clk)
		);
initial clk=0;

always #100 clk=~clk;

		initial begin

			// Wait 100 ns for global reset to finish
			#100  MUL_Status= 1'b0;                                  //400
		    Operand3 = `R0;
				Operand4 = `R1;
				Operand3_Tag = 3'bz;
				Operand4_Tag = 3'bz;
				Operand3_Vbit = 1'b1;
```

```
                    Operand4_Vbit = 1'b1;

            #200  MUL_Status= 1'b0;                                    //500
        Operand3 = `R12;
                    Operand4 = `R13;
                    Operand3_Tag = 3'bz;
                    Operand4_Tag = 3'bz;
                    Operand3_Vbit = 1'b1;
                    Operand4_Vbit = 1'b1;

                // Add stimulus here

        end

endmodule
```

## (F.5) Simulation Result:

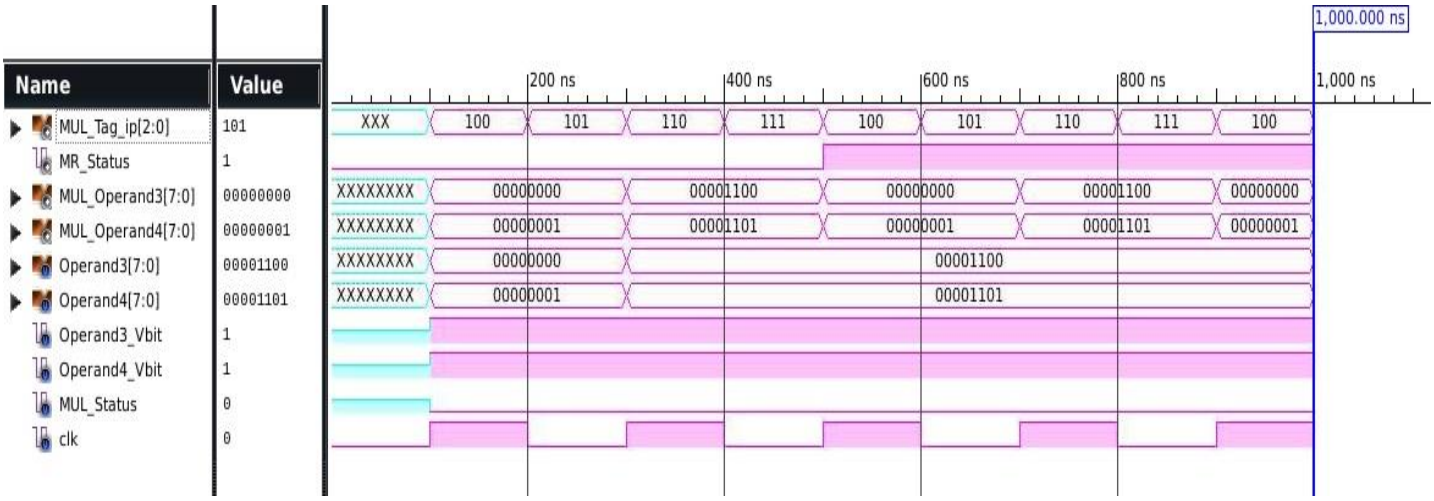



Fig.4.12: Multiplier Reservation Station Waveforms

**Section G - Multiplier Unit:**
**(G.1) Block Diagram:**

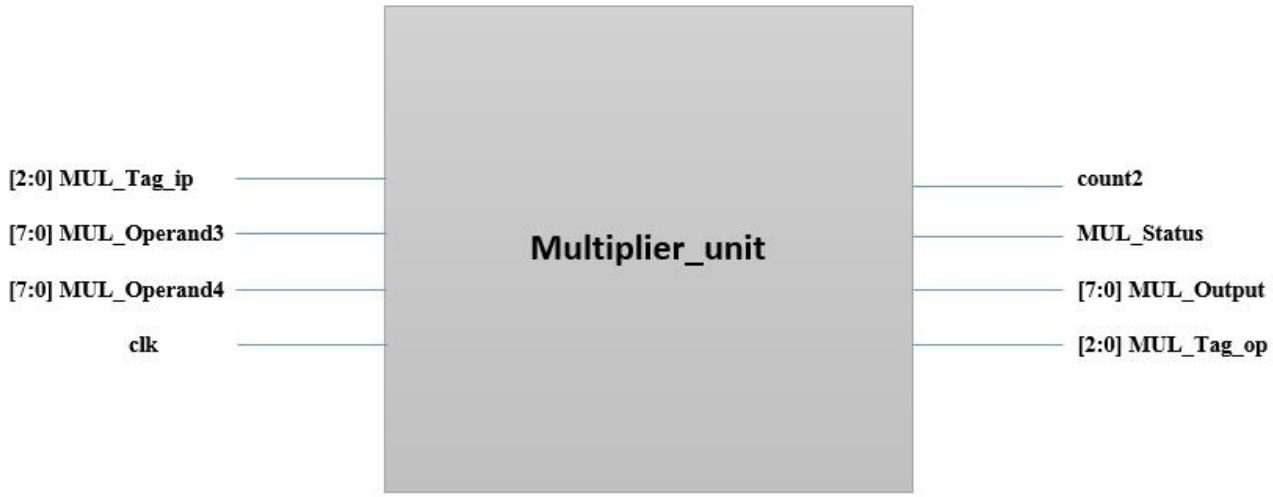



Fig.4.13: Multiplier Block

**(G.2) Description:** The Multiplier Unit performs the operation and the output is broadcasted to the Register Table as well as the Register Table along with Tag output which is basically the tag associated with the reservation slot. The Multiplier Unit Block is shown in Fig.4.13 and its waveforms for Testbench in this section are shown in Fig.4.14.

**(G.3) Verilog Code:**

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module Multiplier_Unit(
count2,
MUL_Status,
 MUL_Output,
 MUL_Tag_op,
 MUL_Tag_ip,
 MUL_Operand3,
 MUL_Operand4,
 clk
 );


output reg MUL_Status;
output reg [7:0] MUL_Output;
output reg [2:0] MUL_Tag_op;

output reg [2:0] count2=3'b0;
reg [2:0] I;
```

```
input wire [2:0] MUL_Tag_ip;
input wire [7:0] MUL_Operand3;
input wire [7:0] MUL_Operand4;
input wire clk;

//integer count21;
//integer count22;
reg [7:0] Source_array1 [3:0];
reg [7:0] Source_array2 [3:0];
reg [7:0] MUL_Buffer;

always @(clk)

                        case (count2)

                        3'b000 :        begin
                                        MUL_Status = 1'b1;
                                        count2 = count2+1;
                                        MUL_Output=8'bz;
                                        MUL_Tag_op=3'bz;
                                        end

                        3'b001 :        begin
                                        MUL_Output = MUL_Operand3 * MUL_Operand4;
                                        MUL_Output=MUL_Buffer;
                                        MUL_Status = 1'b1;
                                        count2 = count2+1;
                                        MUL_Output=8'bz;
                                        MUL_Tag_op=3'bz;

                                        end

                        3'b010 :        begin

                                        MUL_Status = 1'b1;
                                        count2 = count2+1;
                                        MUL_Output=8'bz;
                                        MUL_Tag_op=3'bz;
                                        end
                        3'b011 :        begin

                                        MUL_Status = 1'b1;
                                        count2 = count2+1;
                                        MUL_Output=8'bz;
                                        MUL_Tag_op=3'bz;
                                        end
                        3'b100 :        begin

                                        MUL_Status = 1'b1;
                                        count2 = count2+1;
                                        MUL_Output=8'bz;
                                        MUL_Tag_op=3'bz;

                                        end
                3'b101 :        begin

                                        MUL_Status = 1'b0;
                                        count2 = count2+1;
                                        MUL_Status = 1'b0;
                                        MUL_Output = MUL_Operand3 * MUL_Operand4;
                                        MUL_Output=MUL_Buffer;
```

```
                                        MUL_Tag_op=MUL_Tag_ip;
        end

                3'b110 :        begin
                                        MUL_Status = 1'b1;
                                        count2 = count2+1;
                                        MUL_Output=8'bz;
                                        MUL_Tag_op=3'bz;
                                        end

        3'b111 :        begin
                                        MUL_Buffer = MUL_Operand3 * MUL_Operand4;
                                        MUL_Status = 1'b1;
                                        MUL_Output=8'bz;
                                        MUL_Tag_op=3'bz;
                                        count2 = 3'b0;
                                        end
                        endcase

endmodule
```

## (G.4) Verilog Testbench:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module Multiplier_test;

        // Inputs
        reg [2:0] MUL_Tag_ip;
        reg [7:0] MUL_Operand3;
        reg [7:0] MUL_Operand4;
        reg clk;

        // Outputs

        wire [2:0] count;
        wire MUL_Status;
        wire [7:0] MUL_Output;
        wire [2:0] MUL_Tag_op;

        // Instantiate the Unit Under Test (UUT)
        Multiplier_Unit uut (
                .count(count),
                .MUL_Status(MUL_Status),
                .MUL_Output(MUL_Output),
                .MUL_Tag_op(MUL_Tag_op),
```

```
		.MUL_Tag_ip(MUL_Tag_ip),
		.MUL_Operand3(MUL_Operand3),
		.MUL_Operand4(MUL_Operand4),
		.clk(clk)
	);

	initial clk = 0;

	always #100 clk = ~clk;
	initial begin
		// Initialize Inputs

		#600
				MUL_Tag_ip = `x;
				MUL_Operand3 = `R0;
				MUL_Operand4= `R1;

		#800		MUL_Tag_ip = `y;
				MUL_Operand3 = `R12;
				MUL_Operand4= `R13;
		#1200 $finish;
	end

endmodule
```

## (G.5) Simulation Results

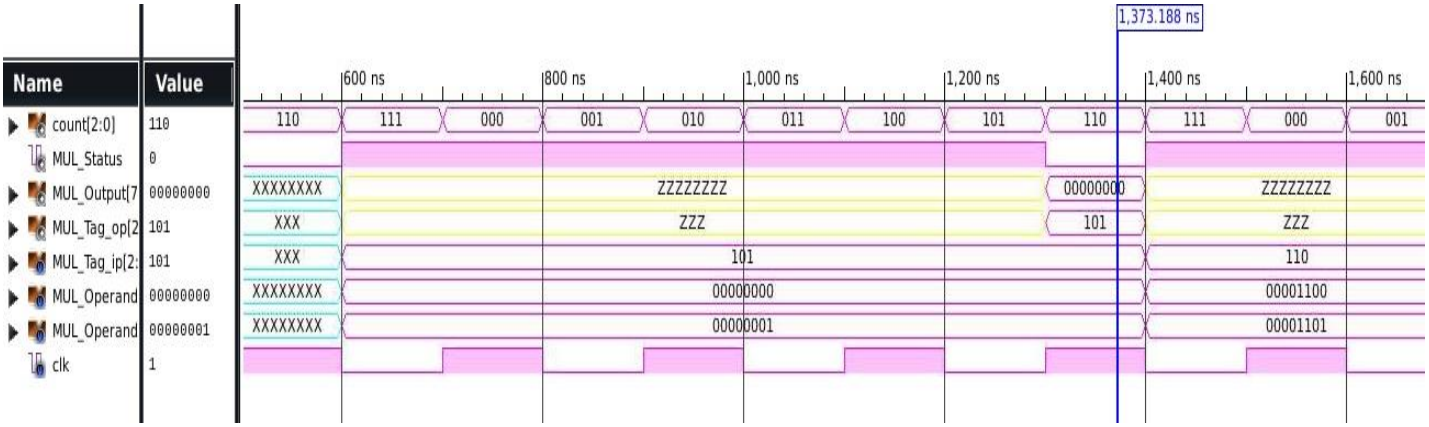



Fig.4.14 (a) Multiplier Unit Waveforms

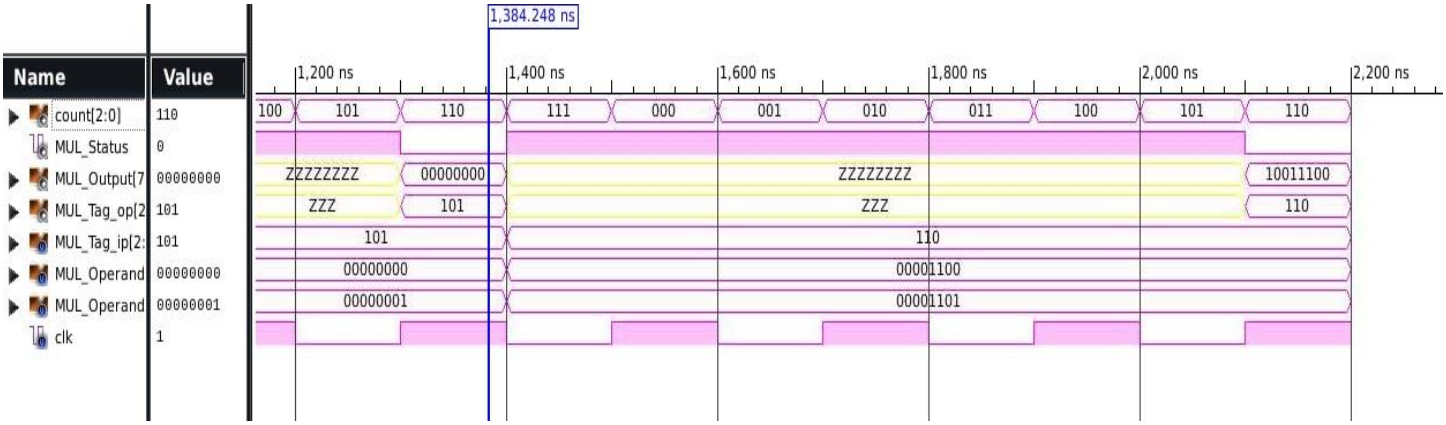



Fig.4.14 (a) Multiplier Unit Waveforms

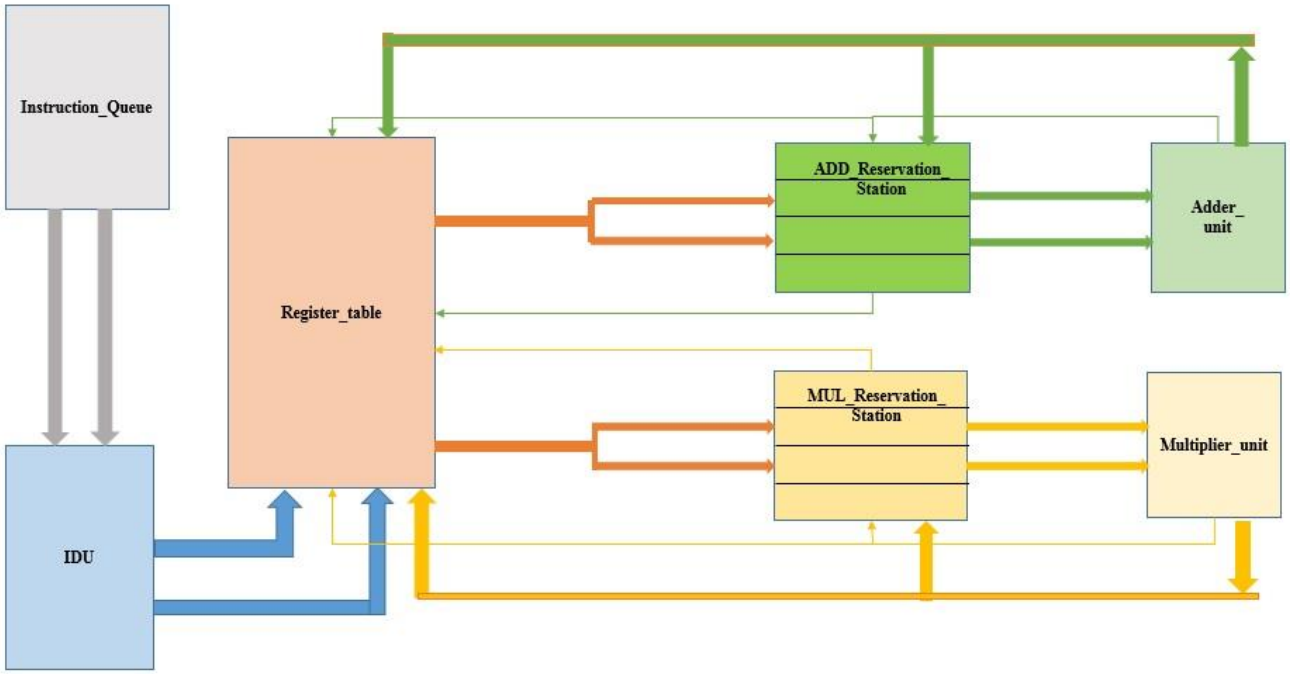
## Section H – Definition File

```
// Functional Unit Definition

`define MUL 8'b00001111
`define ADD 8'b11110000
`define LW  8'b00110011
`define SW  8'b11001100

//Register File Definition

`define R0  8'b00000000
`define R1  8'b00000001
`define R2  8'b00000010
`define R3  8'b00000011
`define R4  8'b00000100
`define R5  8'b00000101
`define R6  8'b00000110
`define R7  8'b00000111
`define R8  8'b00001000
`define R9  8'b00001001
`define R10 8'b00001010
`define R11 8'b00001011
`define R12 8'b00001100
`define R13 8'b00001101
`define R14 8'b00001110
`define R15 8'b00001111
`define R16 8'b00010000
`define R17 8'b00010001

//Adder Reservation Table Tag

`define a 3'b000
`define b 3'b001
`define c 3'b010
`define d 3'b011

//Multiplier Reservation Table Tag

`define w 3'b100
`define x 3'b101
`define y 3'b110
`define z 3'b111

//Sanity Bits

`define sanity 8'b11111111
```

# CHAPTER V
# ASSEMBLY

## Section A - Top Module

### (A.1) Block Diagram:




**(A.2) Description:** The modules described in Chapter 4 are instantiated and each module performs its function as described earlier. Certain internal wires are programmed to be input as well as output. In Chapter III, the proposed design is explained using the same Fig as shown above.
Thus, the input to the whole system is simply a clock. And the outputs are the output of each functional unit. The results are extensively discussed in Fig. 5.1 (a)-(b).

### (A.3) Verilog Code:

```
`include "/home/eng/s/spp150130/spp150130/Tomasulo_Implementation/Definition_File.v"

module Top_Module(
Op1,
Op2,
Op3,
Op4,
clock_count,
count,
count2,
ADD_Status,
ADD_Output,
ADD_Tag_op,
```

```
MUL_Status,
MUL_Output,
MUL_Tag_op,
clk);

//output [7:0] ADD_Operand 1;
//output [7:0] ADD_Operand 2;
//output [19:0] UP1;
//output [19:0] UP2;
output [7:0] Op1;
output [7:0] Op2;
output [7:0] Op3;
output [7:0] Op4;
output [2:0] count;
output [2:0] count2;
inout [4:0] clock_count;
output ADD_Status;
output MUL_Status;
output [7:0] ADD_Output;
output [2:0] ADD_Tag_op;
output [7:0] MUL_Output;
output [2:0] MUL_Tag_op;

input wire clk;

// Instruction Queue

wire [7:0] inst1_type;                                                                  // Output
wire [7:0] inst1_destination_reg;
wire [7:0] inst1_source_reg1;
wire [7:0] inst1_source_reg2;
wire [7:0] inst2_type;
wire [7:0] inst2_destination_reg;
wire [7:0] inst2_source_reg1;
wire [7:0] inst2_source_reg2;
//wire [4:0] clock_count;
wire [1:0] select_instruction;

//IDU
wire [7:0] IDU_out_inst1_type;
wire [7:0] IDU_out_inst1_destination_reg1;
wire [7:0] IDU_out_inst1_source_reg1;
wire [7:0] IDU_out_inst1_source_reg2;
wire [7:0] IDU_out_inst2_type;
wire [7:0] IDU_out_inst2_destination_reg2;
wire [7:0] IDU_out_inst2_source_reg3;
wire [7:0] IDU_out_inst2_source_reg4;

//Register table
wire [2:0] ADD_Tag_ip;
wire [2:0] MUL_Tag_ip;
wire [7:0] Operand1;
wire [7:0] Operand2;
wire [2:0] Operand1_Tag;
wire [2:0] Operand2_Tag;
wire Operand1_Vbit;
wire Operand2_Vbit;
wire AR_Status;
wire MR_Status;

wire [7:0] Operand3;
wire [7:0] Operand4;
wire [2:0] Operand3_Tag;
wire [2:0] Operand4_Tag;
```

```
wire Operand3_Vbit;
wire Operand4_Vbit;

//Adder_unit
wire [7:0] ADD_Operand1;
wire [7:0] ADD_Operand2;

wire [7:0] MUL_Operand3;
wire [7:0] MUL_Operand4;


// Port Mapping


Instruction_Queue T1 (
inst1_type,
inst1_destination_reg,
inst1_source_reg1,
inst1_source_reg2,
inst2_type,
inst2_destination_reg,
inst2_source_reg1,
inst2_source_reg2,
clock_count,
select_instruction,
clk);

IDU T2 (
select_instruction,
IDU_out_inst1_type,
IDU_out_inst1_destination_reg1,
IDU_out_inst1_source_reg1,
IDU_out_inst1_source_reg2,
IDU_out_inst2_type,
IDU_out_inst2_destination_reg2,
IDU_out_inst2_source_reg3,
IDU_out_inst2_source_reg4,
inst1_type,
inst1_destination_reg,
inst1_source_reg1,
inst1_source_reg2,
inst2_type,
inst2_destination_reg,
inst2_source_reg1,
inst2_source_reg2,
AR_Status,
MR_Status,
clock_count,
clk
);


Register_Table T3(
Operand1,
Operand2,
Operand3,
Operand4,
Operand1_Tag,
Operand2_Tag,
Operand3_Tag,
Operand4_Tag,
Operand1_Vbit,
Operand2_Vbit,
Operand3_Vbit,
```

```
Operand4_Vbit,
IDU_out_inst1_type,
IDU_out_inst2_type,
IDU_out_inst1_source_reg1,
IDU_out_inst1_source_reg2,
IDU_out_inst2_source_reg3,
IDU_out_inst2_source_reg4,
IDU_out_inst1_destination_reg1,
IDU_out_inst2_destination_reg2,
ADD_Tag_op,
MUL_Tag_op,
ADD_Tag_ip,
MUL_Tag_ip,
ADD_Output,
MUL_Output,
clock_count,
clk
);

ADD_Reservation_Station T4(
ADD_Tag_ip,
AR_Status,
ADD_Operand1,
ADD_Operand2,
Operand1,
Operand2,
Operand1_Tag,
Operand2_Tag,
Operand1_Vbit,
Operand2_Vbit,
ADD_Status,
clk );

Adder_Unit T5(
count,
ADD_Status,
 ADD_Output,
 ADD_Tag_op,
 ADD_Tag_ip,
 ADD_Operand1,
 ADD_Operand2,
 clk
);

MUL_Reservation_Station T6(
MUL_Tag_ip,
MR_Status,
MUL_Operand3,
MUL_Operand4,
Operand3,
Operand4,
Operand3_Tag,
Operand4_Tag,
Operand3_Vbit,
Operand4_Vbit,
MUL_Status,
clk );

Multiplier_Unit T7(
count2,
MUL_Status,
 MUL_Output,
 MUL_Tag_op,
 MUL_Tag_ip,
```

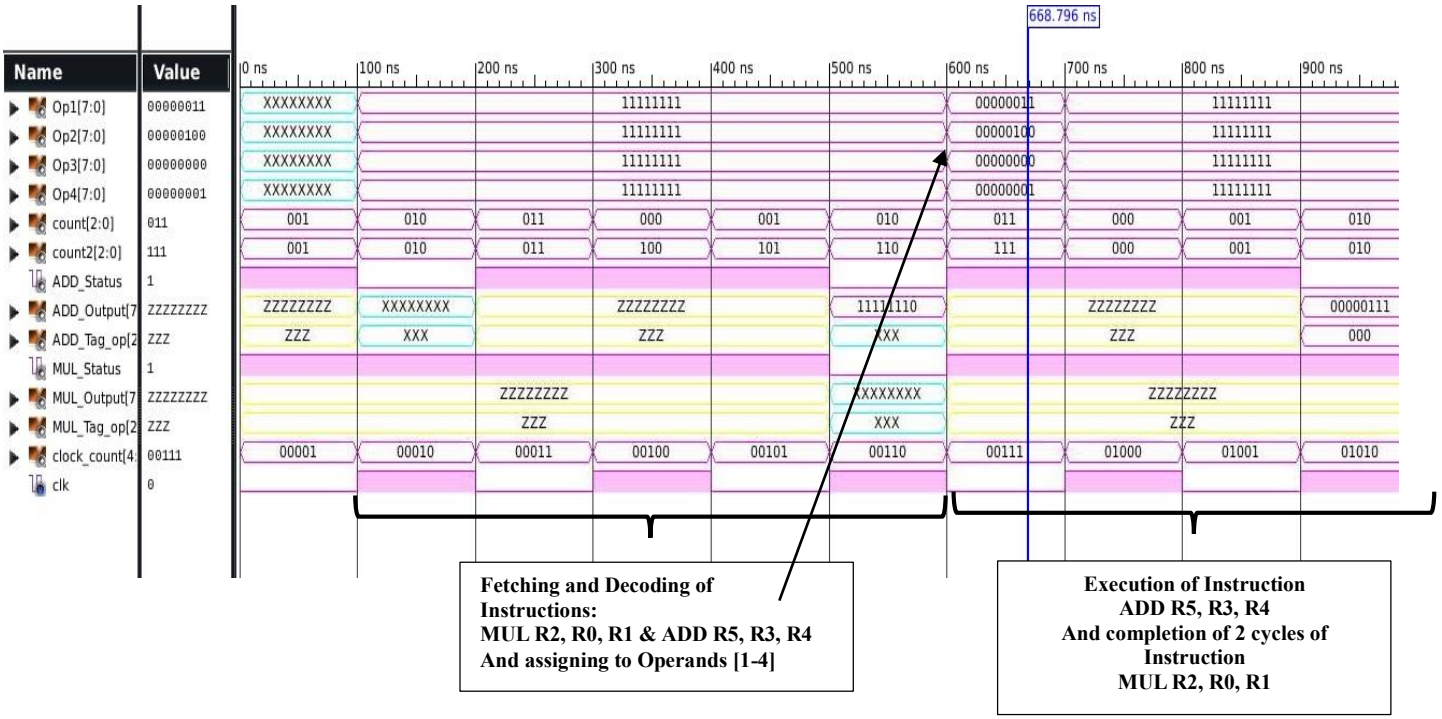
```
 MUL_Operand3,
 MUL_Operand4,
 clk
);

assign Op1 = ADD_Operand1;
assign Op2 = ADD_Operand2;
assign Op3 = MUL_Operand3;
assign Op4 = MUL_Operand4;

endmodule
```

## (A.4) Verilog Testbench:

```
module Top_test;

                                                // Inputs
                                                reg clk;

                                                // Outputs

                                                wire [7:0] Op1;
                                                wire [7:0] Op2;
                                                wire [7:0] Op3;
                                                wire [7:0] Op4;
                                                wire [2:0] count;
                                                wire [2:0] count2;
                                                wire ADD_Status;
                                                wire [7:0] ADD_Output;
                                                wire [2:0] ADD_Tag_op;
                                                wire MUL_Status;
                                                wire [7:0] MUL_Output;
                                                wire [2:0] MUL_Tag_op;

                                                // Bidirs
                                                wire [4:0] clock_count;

                                                // Instantiate the Unit Under Test (UUT)
                                                Top_Module uut (
                                                        .Op1(Op1),
                                                        .Op2(Op2),
                                                        .Op3(Op3),
                                                        .Op4(Op4),
                                                        .clock_count(clock_count),
                                                        .count(count),
                                                        .count2(count2),
                                                        .ADD_Status(ADD_Status),
                                                        .ADD_Output(ADD_Output),
                                                        .ADD_Tag_op(ADD_Tag_op),
                                                        .MUL_Status(MUL_Status),
                                                        .MUL_Output(MUL_Output),
                                                        .MUL_Tag_op(MUL_Tag_op),
                                                        .clk(clk)
                                                );

initial clk=0;

always #100 clk=~clk;

                                                initial begin
                                                        // Initialize Inputs

                                                end
```

endmodule

## (A.5) Simulation Results:



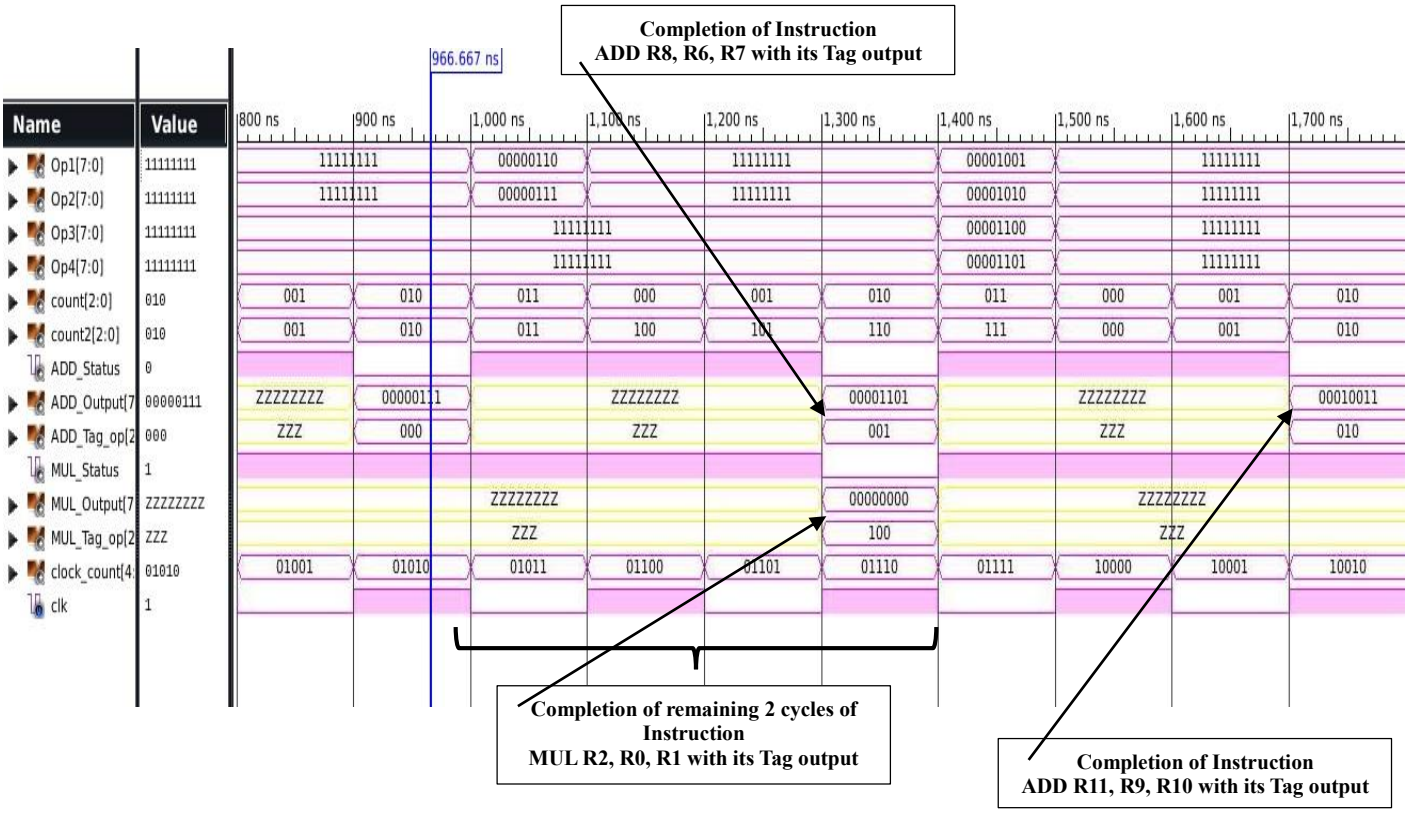



Fig. 5.1 (a) Top Module - Waveforms

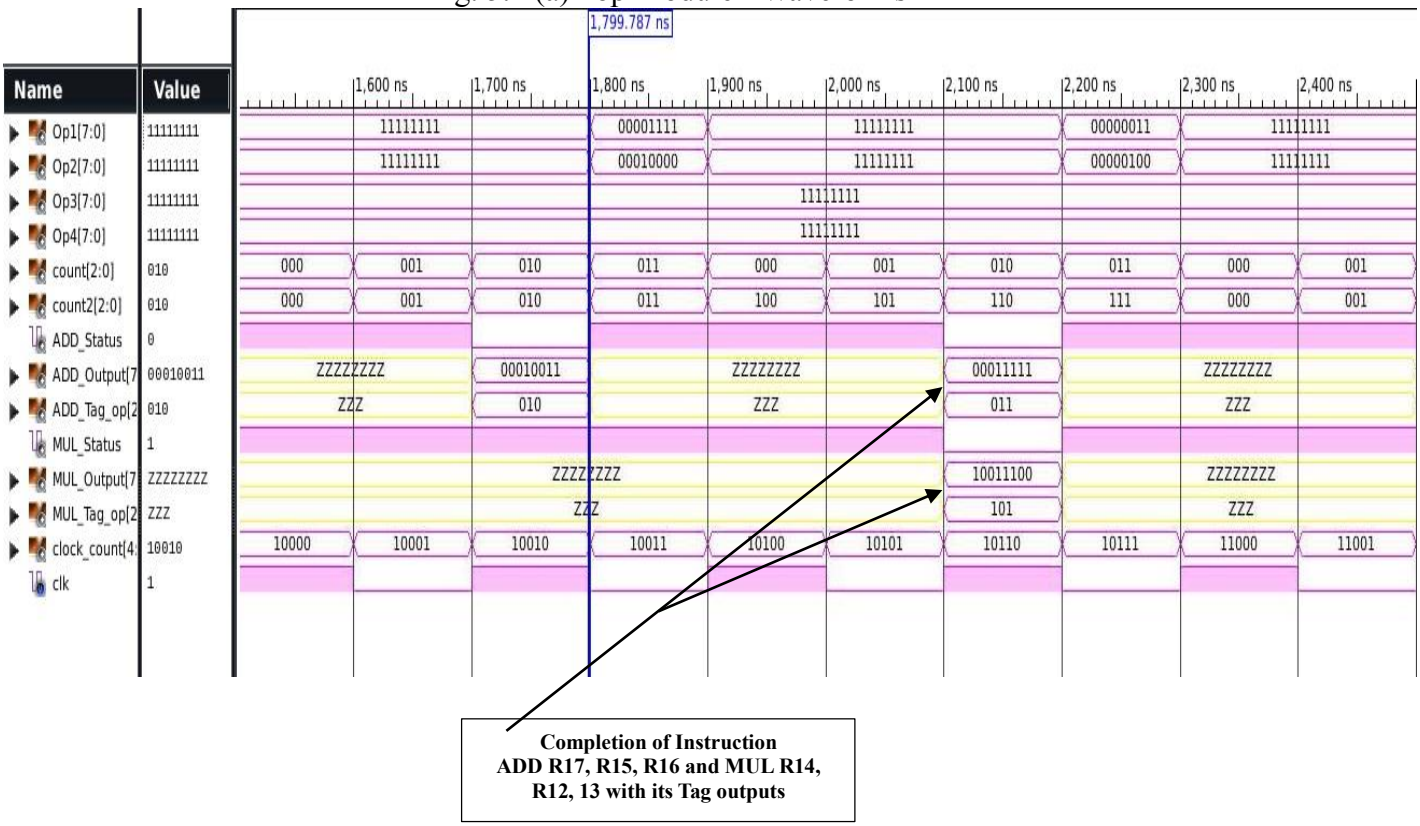



Fig. 5.1 (b) Top Module - Waveforms

## Section B – Analysis of Proposed Design in terms of Performance

In order to prove the efficacy of our proposed design, we have compared two programs with and without data hazards. In Section B.1 and Section B.2 of this Chapter, we have showed the results of two test cases and compare its performance with Traditional Tomasulo Algorithm and In-order Processor having a single ALU Unit with the results obtained in Section A.

**Assumptions:**

- For the sake of brevity, we have not included in our program any test cases involving Load/Store operations. However, a similar Verilog code can be written to translate the functionality of its operations as seen in Chapter 4 in order to get the true number of half cycles required by our proposed design to compare and analyze such a program.
- We will assume 1 Adder and 1 Multiplier Unit in our proposed design while considering the two test cases.
- While considering the case of In-order processor with Single ALU Unit, the Decode Unit is considered to be having 3 half cycles instead of 2 half cycles.
- We are considering a 4-stage pipeline : F-D-E-W Stages for Section B.1 and B.2

**Section B.1:**

Consider the following Program without Data Hazards:

MUL R2, R0, R1
ADD R5, R3, R4
ADD R8, R6, R7
ADD R11, R9, R10
MUL R14, R12, R13
ADD R17, R15, R16

**Assumption:** The execution of next ADD/MUL Operation can begin executing while the previous one is writing the results back since no data dependencies are involved

### 1. Execution of Instructions using Proposed Design (Tomasulo Algorithm + Superscalar Instruction Dispatch)

| Instructions | Number of half cycles | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| MUL R2, R0, R1 | F | F | D | D | D | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W | | | | | | | | |
| ADD R5,R3,R4 | F | F | D | D | D | E1 | E1 | E2 | E2 | W | W | | | | | | | | | | | | |
| ADD R8,R6,R7 | | | F | F | D | D | D | - | - | E1 | E1 | E2 | E2 | W | W | | | | | | | | |
| ADD R11,R9,R10 | | | F | F | D | D | D | - | - | - | - | - | - | E1 | E1 | E2 | E2 | W | W | | | | |
| MUL R14,R12,R13 | | | | | F | F | D | D | D | - | - | - | - | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W |
| ADD R17,R15,R16 | | | | | F | F | D | D | D | - | - | - | - | - | - | - | - | E1 | E1 | E2 | E2 | W | W |

Table 5.1

**Explanation:**

As per the results obtained in Section A of this Chapter, the fetch (F) and decoding (D) of instructions in our proposed design require 5 half clock cycles, the execution (E) of add and multiply instruction require 4 and 8 half cycles respectively and Write (W) takes 2 half cycles.
Hence, Table 5.1 shows the execution of instructions of a program and the **results are analogous as obtained in Section A**.

Each block in no of clock cycles (x-axis) represents a half cycle, hence approximately 12 clock cycles are required to execute the whole program.

**Cycles per Instructions (CPI): 12/6 = 2.0**

## 2. Execution Table with Traditional Tomasulo Algorithm

| Instructions | Number of half cycles | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| MUL R2, R0, R1 | F | F | D | D | D | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W | | | | | | | | | | | | |
| ADD R5,R3,R4 | | | F | F | - | D | D | D | E1 | E1 | E2 | E2 | W | W | | | | | | | | | | | | | |
| ADD R8,R6,R7 | | | | | | F | F | - | D | D | D | - | E1 | E1 | E2 | E2 | W | W | | | | | | | | | |
| ADD R11,R9,R10 | | | | | | | | | F | F | - | - | D | D | D | - | E1 | E1 | E2 | E2 | W | W | | | | | |
| MUL R14,R12,R13 | | | | | | | | | | | | | F | F | D | D | D | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W |
| ADD R17,R15,R16 | | | | | | | | | | | | | | | F | F | - | D | D | D | E1 | E1 | E2 | E2 | W | W | |

Table 5.2

**Explanation:**
Since, the traditional Tomasulo's architecture requires separate DIV/MUL and ADD/SUB Units, in order to exploit out of order execution of instructions not involving dependencies, in Table 5.2 the execution of instruction 2 is completed before instruction 1.
All the instructions are issued in-order as observed from the Table5.2, whereas the execution happens out of order.

Each block in no of clock cycles (x-axis) represents a half cycle, hence approximately 14 clock cycles are required to execute the whole program.
**Cycles per Instructions (CPI): 14/6 = 2.34**

$$\text{Speedup} = \frac{CPI_{Traditional\ Tomasulo}}{CPI_{Modified\ Tomasulo}} = \frac{2.34}{2.0} = 1.17$$

$$\%\ \text{Performance Improvement} = \frac{0.34}{2.0} \text{ x } 100 = 17.0\%$$

### 3. Execution Table of an In-order Processor having a Single ALU Unit

| Instruction | No of half cycles | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | | |
| MUL R2, R0, R1 | F | F | D | D | D | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W | | | | | | | | | | | |
| ADD R5,R3,R4 | | | F | F | - | D | D | D | - | - | - | - | - | E1 | E1 | E2 | E2 | W | W | | | | | | | |
| ADD R8,R6,R7 | | | | | | F | F | - | - | - | - | - | - | D | D | D | - | E1 | E1 | E2 | E2 | W | W | | | |
| | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| ADD R11,R9,R10 | F | F | - | - | D | D | D | - | E1 | E1 | E2 | E2 | W | W | | | | | | | | | | | | |
| MUL R14,R12,R13 | | | | | F | F | - | - | D | D | D | - | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W | | | | |
| ADD R17,R15,R16 | | | | | | | | | F | F | - | - | D | D | D | - | - | - | - | - | E1 | E1 | E2 | E2 | W | W |

Table5.3

**Explanation:**
Since, there is a single ALU Unit while considering this case with the capability of issuing instructions and executing both in-order, the Table 5.3 shows the number of half cycles required by such a processor design.
Each block in no of clock cycles (x-axis) represents a half cycle, hence approximately 20 clock cycles are required to execute the whole program.

**Cycles per Instructions (CPI): 20/6 = 3.34**

$$\text{Speedup} = \frac{CPI_{in-order}}{CPI_{Modified\ Tomasulo}} = \frac{3.34}{2.0} = 1.67$$

$$\%\ \text{Performance Improvement} = \frac{1.34}{2.0} \text{ x } 100 = 67.0\%$$

## Section B.2

**Consider the following Program with Data Hazards:**

ADD R2, R0, R1
MUL R4, R2, R3
ADD R6, R4, R5
ADD R9, R7, R8

**Assumption:** The execution of next ADD/MUL Operation cannot begin executing while the previous one is writing the results back since data dependencies are involved

### 1. Execution of Instructions using Proposed Design (Tomasulo Algorithm + Superscalar Instruction Dispatch)

| Instructions | Number of half cycles | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| ADD R2,R0,R1 | F | F | D | D | D | E1 | E1 | E2 | E2 | W | W | | | | | | | | | | | | | | | | |
| MUL R4, R2, R3 | F | F | D | D | D | - | - | - | - | | | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W | | | | | | |
| ADD R6,R4,R5 | | | F | F | D | D | D | - | - | - | - | - | - | - | - | - | - | - | - | - | - | E1 | E1 | E2 | E2 | W | W |
| ADD R9,R7,R8 | | | F | F | D | D | D | - | - | - | - | E1 | E1 | E2 | E2 | W | W | | | | | | | | | | |

Table 5.4

**Explanation:**

In order to target the data hazards of RAW type, the methodology used in traditional Tomasulo's algorithm is used which is to assign a tag to each of the reservation station operands requiring the results of previous instructions. This is shown in Section D and F (Adder and Multiplier Reservation Stations) of Chapter 4 while the register table update functionality is added in Section E and G of Adder and Multiplier Modules.

Therefore, approximately 14 clock cycles are required to execute the whole program with which **CPI achieved is: 14/4 = 3.5**

## 2. Execution Table with Traditional Tomasulo Algorithm

| Instruction | Number of half cycles | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| ADD R2,R0,R1 | F | F | D | D | D | E1 | E1 | E2 | E2 | W | W | | | | | | | | | | | | | | | | | | | |
| MUL R4, R2, R3 | | | F | F | - | D | D | D | - | - | - | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W | | | | | | | | | |
| ADD R6,R4,R5 | | | | | | F | F | - | - | - | - | D | D | D | - | - | - | - | - | - | - | E1 | E1 | E2 | E2 | W | W | | | |
| ADD R9,R7,R8 | | | | | | | | | | | | F | F | - | - | - | - | - | - | - | - | D | D | D | E1 | E1 | E2 | E2 | W | W |

Table 5.5

$$CPI_{Traditional\ Tomasulo} = \frac{30}{4} = 7.5$$

$$\text{Speedup} = \frac{CPI_{Traditional\ Tomasulo}}{CPI_{Modified\ Tomasulo}} = \frac{7.5}{3.5} = 2.14$$

$$\%\ \text{Performance Improvement} = \frac{4.0}{3.5} \text{ x } 100 = 114.28\%$$

## 3. Execution Table of an In-order Processor having a Single ALU Unit

| Instructions | Number of half cycles | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 34 |
| ADD R11,R9, R10 | F | F | D | D | D | E1 | E1 | E2 | E2 | W | W | | | | | | | | | | | | | | | | | | | | | | |
| MUL R2, R0, R1 | | | F | F | - | D | D | D | - | - | - | E1 | E1 | E2 | E2 | E3 | E3 | E4 | E4 | W | W | | | | | | | | | | | | |
| ADD R11,R9, R10 | | | | | | F | F | - | - | - | - | D | D | D | - | - | - | - | - | - | - | E1 | E1 | E2 | E2 | W | W | | | | | | |
| MUL R14,R12, R13 | | | | | | | | | | | | F | F | - | - | - | - | - | - | - | - | D | D | D | - | - | - | E1 | E1 | E2 | E2 | W | W |

Table 5.6

Table 5.4, 5.5 and 5.6 show the execution of the program involving data hazard using the proposed design, traditional Tomasulo algorithm and in-order processor.

$$CPI_{In-order} = \frac{34}{4} = 8.5$$

$$\text{Speedup} = \frac{CPI_{in-order}}{CPI_{Modified\ Tomasulo}} = \frac{8.5}{3.5} = 2.42$$

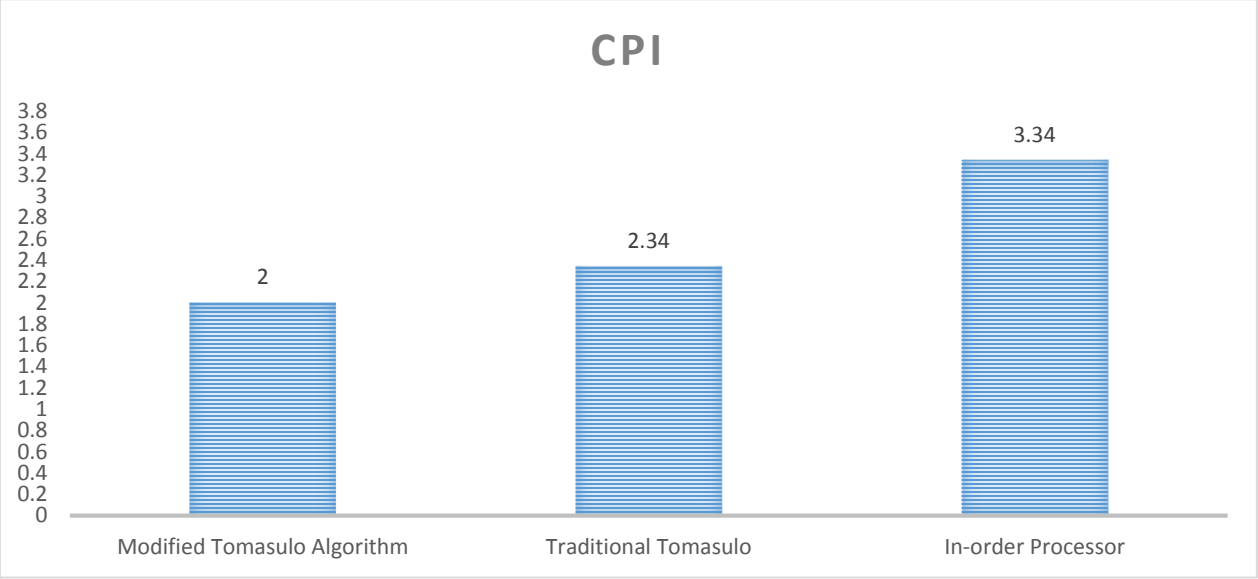
$$\%\ \text{Performance Improvement} = \frac{5.0}{3.5} \text{ x } 100 = 142.85\%$$

# CHAPTER VI

## Section A

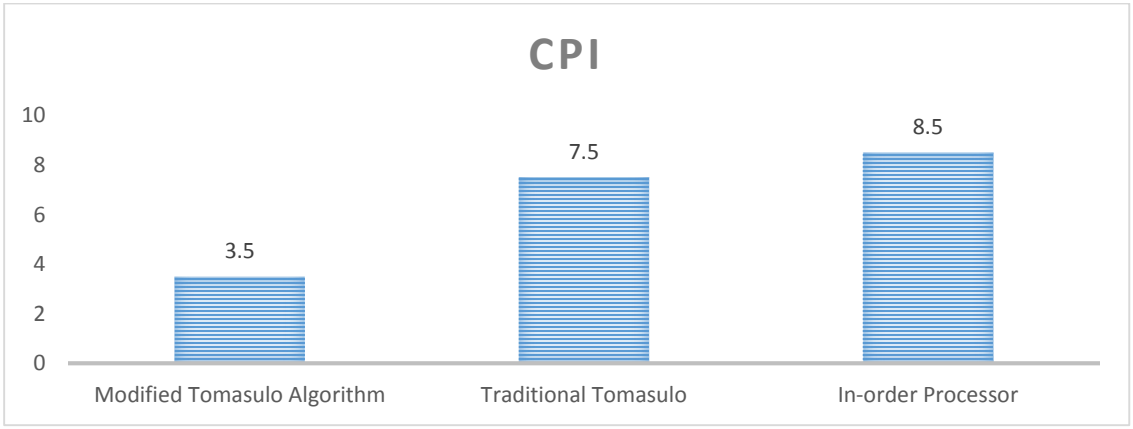
### Key Results of Chapter V Section B1 (Program without Data Hazard):

- Thus, there is a reduction of 0.34 in CPI of the Modified Design compared to Traditional Tomasulo Algorithm and a 1.34 reduction compared to In-order Processor.
- However, a Speedup of 1.17 and 1.67 in each case is achieved respectively which is significant.




### Key Results of Chapter V Section B2 (Program with Data Hazard):

- Thus, there is a significant reduction of 4.0 in CPI of the Modified Design compared to Traditional Tomasulo Algorithm and a 5.0 reduction compared to In-order Processor.
- However, in terms of Speedup a 2.14 and 2.42 is achieved respectively which is significant improvement.



**Conclusions:**

- ✓ **We were able to simulate our design in Verilog completely and testing of the Top module was done which drew the results as expected.**
- ✓ **The efficacy of the proposed design is certainly distinct when it comes to data hazards and hence the proposed design performs significantly well compared to the other two designs.**
- ✓ **Our desired target of improving the CPU Performance was achieved and there was a significant speedup observed while comparing the remaining two designs.**
- ✓ **The desired target of fetching and decoding of two instructions per 5 half clock cycles was achieved, which led to saving a lot of cycles, which is our key contribution.**

CHAPTER VII

## SECTION A
## SCOPE OF FURTHER IMPROVEMENTS

- **Structural Pipelining can be done further in our existing design in order to decrease the value of CPI and hence boost the performance.**
- **The concept of Re-order Buffer can be added to our design in order to deal with false dependencies such as WAW, WAR and that would make our system more unique and robust.**
- **Comparison with Multithreaded Processors such as Fine Grain, Coarse Grain and Simultaneous Multithreading Processors should be done in order to know the state of art of the proposed system.**
- **Further analysis of the proposed design can be done by synthesizing the code and estimating the Area/Power/Timing requirements.**