California State University, Northridge

Single Issue Instruction Dispatcher Based on Tomasulo's Algorithm

A graduate project submitted in partial fulfillment of the requirements

For the degree of Master of Science

In Electrical Engineering

By

Heema S Shah

December 2014

The graduate project of Heema Shah is approved by:

_____	_____
Dr. Ali Amini, Ph.D.	Date
_____	_____
Dr. Ramin Roosta, Ph. D.	Date
_____	_____
Dr. Shahnam Mirzaei, Ph.D., Chair	Date

California State University, Northridge

Acknowledgment

I would take this opportunity to express my sincere thanks to all the committee members without whom the project would never have been a success. They have guided me and supported me throughout this period and I would always be indebted to them.

I am grateful to Dr. Mirzaei for his continuous guidance on this journey towards my project completion. At the same time, Dr. Amini offered unwavering support and Dr. Roosta played an important part towards fulfilling this task.

Also, I would also like to thank my family and friends whose constant support always kept me motivated and gave the strength needed for such a challenging task.

Table of Contents

List of Tables

List of Figures

Abstract

Single Issue Instruction Dispatcher Based on Tomasulo's Algorithm

By

Heema Shah

Master of Science in Electrical Engineering

In order to achieve maximum throughput and efficiency, today's processors employ various techniques and algorithms. Some of these would involve implementing multi-level cache, multiprocessor systems or exploring parallelism within instructions. Tomasulo's algorithm by Robert Tomasulo has proved to be a major technique of exploiting the possibility of parallel and out-of-order execution of instructions. This project involves HDL implementation of Tomasulo's algorithm on Xilinx ISE 14.2 using Verilog HDL. The algorithm consists of a scheduler which is the heart of the system responsible for eliminating structural and data hazards to allow efficient instruction execution.

CHAPTER 1 – INTRODUCTION

1.1Instruction Level Parallelism

Instructional level parallelism (ILP) is derived from the concept of pipelining that deals with the overlapping of instructions in order to improve the performance of the CPU and to compute the instructions in parallel.

ILP can be exploited or used by two separate approaches: (1)a dynamic approach that involves the use of hardware by which parallelism is discovered and exploited or (2) a static approach that is dependent on software to discover parallelism at compile time.[1] The former one is more focused on saving energy and hence, is used for the personal mobile device market while the latter dominates the desktop and server domains.

Since many years, several techniques have been developed that were originally used for one approach, but ended up being exploited for a design depending on the other approach. ILP has a few more limitations on its hand due to which a transition had tobe made to multicore systems.

The CPI (cycles per instruction) is determined by the following equation:

Pipeline CPI = Ideal pipeline CPI + Structured stalls + Data hazard stalls + Control stalls[1]

The maximum performance that can be attained by the system is given by the Ideal Pipeline CPI. In order to increase the Instructions Per Clock (IPC) we can decrease the terms on the right side so that the overall CPI decreases.

A basic block defined as a straight-line code sequence that does not have any braches except at the entry and the exit to get in and get out respectively has a very small amount of parallelism available to it. Due to which, ILP has to be exploited within multiple basic blocks.

The most common technique to increase the ILP is to use the iterations of the loop for our parallelism techniques. This is called loop-level parallelism.[1] We will use the example of a 500-element array that adds itself and is totally parallel:

```
for (j=0; j <=500; j=j + 1)
    x[j] = x[j] + y[j];
```

Even though there is an almost negligible chance of overlapping within this loop iteration, each iteration of the loop can overlap with any other as well. A number of techniques have been developed in order to transform loop-level parallelism to instruction-level parallelism. These work by opening up the loop statically, i.e., by using the compiler or dynamically, i.e., by using the hardware.

1.2Types of Dependences and Hazards

It is important to determine how the instructions are interdependent in order to evaluate how much parallelism actually exists in the program and how we can use that parallelism. In order to use instruction-level parallelism we have to find out which instructions can be run at the same time.

There are mainly three different types of dependencies:

(1) Data Dependences
(2) Name Dependences
(3) Control Dependences

1.2.1 Data Dependences

Any instruction *a* can be said to be data dependent on instruction *b* if any one of the following is true:

- The result produced by instruction *b* may be used by instruction *a.*
- If instruction *a,b,c* are inter data dependent, i.e. instruction *a* is dependent on *c* and instruction *c* is dependent on *b.*

1.2.2 Name Dependences

Name dependence will occur in any one of the two cases: if they use the same register or the same memory location which is referenced as a name. However, no data flow is related with that name. Two types of name dependences occur between instruction *a* and instruction *b*:

- Antidependence between instructions *a* and *b* takes place when the memory or register read by instruction *b* is written by instruction *a*.
- The other form of name dependence, output dependence, occurs when two instructions *a* and *b* write to the same memory location or register.

1.2.3 Control Dependences

The third and the last type of dependences are the control dependences. These type of dependences determine in what order an instruction has to be executed with respect to a branch in the program. This in turn makes sure that the instruction executes in proper order. One can understand control dependences with the use of an if-then code:

```
if C1 {
        A1;
}
```

```
if C2 {
        A2;
}
```

Here, one can say that A1 is dependent on C1 and A2 is dependent on C2 but A2 is not dependent on C1.

There are certain limitations to control dependences:

(1) Any control dependent instruction cannot be taken out from the **if** condition to make it no-control dependent.
(2) Any non-control dependent instruction cannot be brought in an **if** condition to make it control dependent.

1.2.4 Data Hazards

If a name or data dependence exists between two instructions and that dependence is such that the execution overlap changes the operand access order that involves that dependence, then a hazard will come into play. Due to this dependence, we are required to preserve the program order. The program order is nothing but the order in which the instructions would have executed if they were executed sequentially.

There are basically three types of data hazards. Let us consider two different instructions *a* and *b*:
RAW (read after write) – Before a writes to a source, b starts reading it, thus b will get the previous value and not the correct one. This is one of the most common hazards and usually occurs due to data dependences. We can solve this hazard by simply preserving the program order.

WAW (write after write) – Before *a* has written an operand, *b* tries to write it. This leads to an out-of-order write by the instructions that leaves the last value as the value given by *a* rather than *b*. This hazard takes place due to output dependences.

WAR (write after read) – Before *a* reads a destination *b* tries to write it leading to *a* getting the incorrect new value instead of the correct previous value. WAR hazards are not common and occur only if there are a few instructions that write the results early and other instructions read it late.

All these hazards can be removed by proper scheduling in the design.

1.3 Scheduling

Scheduling deals with the re-arrangement of the order in which the instructions will be executed so that one can achieve the maximum performance and throughput of the design. In order to perform a good scheduling, the designer needs to be aware of the processor design, latencies and dependences.

1.3.1 Types of Scheduling

There are two types of scheduling methods:

Static Scheduling – This type of scheduling takes place due to the compiler and hence, it is software based scheduling. This type of scheduling is more focused towards in-order execution of instructions i.e. if one instruction gets stalled in the queue then the next instruction will not be executed. This type of scheduling is highly inefficient.

Dynamic Scheduling – This type of scheduling takes place due to the hardware present. The focus of dynamic scheduling is more towards out-of-order execution of instructions. Hence, one reduces the chances of any instruction getting stalled due to a previous instruction getting stalled. This is also known as the Non-Von Neumann method.[1]

1.4 Scoreboarding and Tomasulo's algorithm

There are mainly two techniques that are used in order to implement dynamic scheduling in a design:

(1) Scoreboarding
(2) Tomasulo's algorithm

1.4.1 Scoreboarding

In this method a scoreboard is maintained and the data dependencies of each instruction are logged in. As soon as the scoreboard comes to the conclusion that there are no further conflicts with the preceding instructions then the next instruction is released. However, if any instruction gets stalled, the scoreboard will keep an eye out towards the flow of the executing instructions until and unless all the dependencies have been cleared off and only then will it release the stalled instruction.

1.4.2 Stages in Scoreboarding

There are four stages in scoreboarding:

(1) **Issue** – The system remembers and checks which registers will be read and written in the next stages. The instruction is stalled until and unless all other instructions trying to write to the same register have finished executing. The instruction is also stalled the

necessary functional units are temporarily busy. The WAW types of dependences are resolved at this stage.

(2) **Read operands –** Once the instruction is issued and proper hardware allocation is done, it will wait till the operands that it requires are available. This stage takes care of RAW type of dependences.

(3) **Execution –** After all the operands are available, the functional unit starts running. Once the result is available, it is sent to the scoreboard and updated.

(4) **Write result –**Here, the result is supposed to be written in the destination register. But, if there are any pending read instructions then the current write instruction is delayed until the previous instruction is completed. Thus, the data dependencies and WAR are handled at this stage.

1.4.3 Status Tables in Scoreboarding

In order to control the flow of the execution of the instructions, the scoreboard keeps updating three status tables:

(1) **Instruction Status –**This table retains the information about the stage in which a particular is.

(2) **Functional Unit Status –** Retains and shows the status of each functional unit. Every single functional unit maintains 9 different fields in a table:

- Busy – tells us if the unit is being used or not
- F_i – Destination register
- Op – operation to be performed in that unit
- F_j, F_k – Source register numbers
- R_j, R_k – Flags indicating whether F_j,F_k are ready or not
- Q_j,Q_k – Funtional units that produce source registers[1]

(3) **Register Status –** Tells us about each register and which functional unit is going to write results into it.
This method stalls the instruction at the issue stage when it does not have a ready functional unit. Due to this, the instructions that could have been executed will have to wait until this structural hazard is dealt with. Here comes the importance of the Tomasulo's algorithm which deals with the structural hazard and also resoles WAW and WAR dependencies with the help of register renaming.

1.4.4 Tomasulo's Algorithm

A hardware based algorithm developed by Robert Tomasulo in 1967 for IBM, it allows several instructions to be executed non-sequentially or out-of-order which would have been otherwise stalled due to certain dependencies. [1]

The Tomasulo approach involves a technique known as register renaming wherein control and buffers are spread among functional units (FUs) and are called reservation stations (RS), these RSs are used by the instructions in place of registers.
The Tomasulo approach can perform several optimizations which the compiler isn't even capable of.
The Tomasulo algorithm will be further discussed in the next chapter.

1.4.5 Comparison between Scoreboarding and Tomasulo's algorithm

Tomasulo's	Scoreboarding
Control and buffers (reservation stations) are spread out among the functional units	The control and buffers are centralized.
The registers in the instructions are replaced with pointers given in the reservation station buffers. (Register renaming)	Actual registers are used. (No use of register renaming)
The Common Data Bus (CDB) issues the result to all the functional units.	No use of CDB is done. Results are directly sent out to the registers.
Load and Store instructions treated as different functional units.	All functional units are combined into one.
Stages: Issue, Execution and Result	Stages: Issue, Read Operands, Execution, Write Result.

Table 1.1 Comparison between Scoreboarding and Tomasulo's Algorithm[1]

CHAPTER 2 – TOMASULO: THE CONCEPT

As we understand, there are three hazards to be taken care of while scheduling instructions dynamically, namely RAW, WAR and WAW. These three hazards can be handled using the Tomasulo algorithm. Tomasulo issues the instructions IN-ORDER but the execution order maybe different. There are two important principles in the Tomasulo design:

(1) Make sure there are no structural hazards while issuing instructions
(2) If there is a data hazard, make sure we stall and wait for the data to come in

Since Tomasulo involves out-of-order execution, WAR and WAW hazards need to be resolved at the time of decoding. Since these hazards involve interdependence of the register names, we technically rename the registers, which automatically take care of these hazards. There are different techniques for register renaming such as:

a. Use of reservation stations (Original Tomasulo)
b. Re-order buffer (ROB) acting as additional registers
c. Use a physical register file and a Register Alias Table (RAT)

Now, let's discuss the first method in detail, as that is what I have implemented.

2.1. Use of reservation stations (Original Tomasulo)

In this technique, WAR and WAW hazards are eliminated by renaming destination registers and also those registers which are future operands or destinations. Thus, the instruction uses the correct value of the operand.

Let us consider an example of this technique.

```
ADD   R0, R1, R2
MUL   R6, R0, R8
SUB   R1, R6, R3
MUL   R8, R10, R14
DIV   R6, R10, R8
```

There are three WAR hazards: firstly, between ADD and SUB on R1, secondly, between SUB and DIV on R6, and thirdly between MUL and MUL on R8. Also, there is a possible output dependence, which is WAW hazard on R6 between MUL and DIV. Along with these, there are three possible data dependences, which is RAW hazard: firstly, on R0 between ADD and MUL instruction as ADD would take more than one cycle to execute and write back, while MUL would be already in the pipeline. Secondly, on R6 between MUL and SUB, and thirdly, on R8 between MUL and DIV.

Thus, there are four name dependences which constitute WAR and WAW hazards and they can be eliminated by register renaming. Let us assume three temporary registers, P, Q and R. Using these aliases, we can re-write the sequence as:

```
ADD   R0, R, R2
MUL   P, R0, R8
SUB   R, P, R3
MUL   Q, R10, R14
DIV   R6, R10, Q
```

Hence, any further usage of R8 should be replaced by Q. This method of renaming can be achieved statically by a compiler. Again, branches create a more complication in figuring out the subsequent uses of renamed registers. However, Tomasulo is also capable of resolving that.

Basically, register renaming is achieved by employing reservation stations. Their job is to buffer the operands of the pending instructions as soon as they are available. This means, as soon as a result is available, any instructions waiting for that register result are provided with the result immediately. This provision removes the extra step of reading the updated register file to get the recent value of the operand. Also, the instructions waiting to be executed denote the reservation station which will provide the necessary operands for executing it. Moreover, when there are consecutive writes to a register, only the last instruction is permitted to write the result to the register. This process of renaming registers takes place during *Issue* stage, where further operands are assigned the name of the reservation stations, which are going to provide the results for those registers.

The entire Tomasulo approach is based on utilizing hardware support by creating reservation stations for register renaming. Reservation stations can be more than actual registers in the processor.

In Tomasulo, hazard detection and pipeline control is distributed all over the processor. Each reservation station has certain fields that are updated as per the execution of instructions and these fields determine whether a particular instruction can start execution or not. Another important characteristic: internal forwarding exists from functional units to reservation stations, where these reservation stations buffer the operands and start instruction execution without having to read the operands from the register file. This by-passing is achieved by using a common data bus, which broadcasts the results to all waiting units and avoids the need for communication between various units for operands.

The basic Tomasulo processor, along with the load/store and memory unit looks as shown in figure. [1]

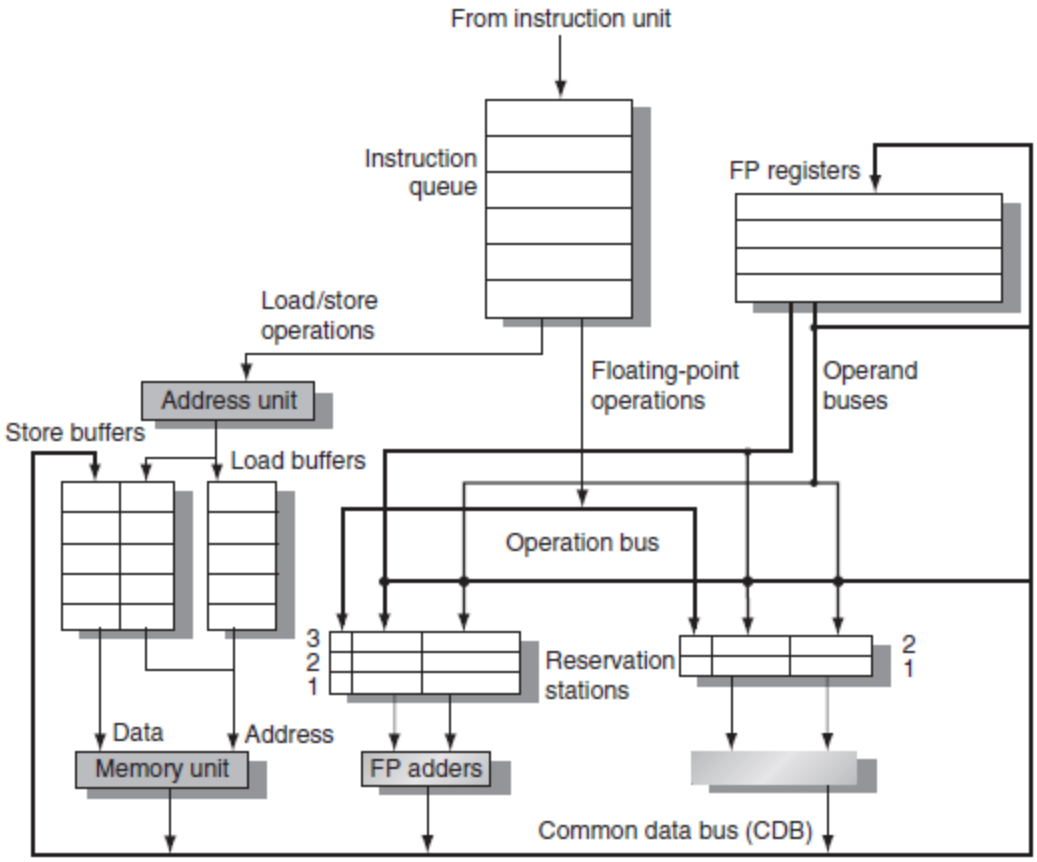

Fig2.1 MIPS floating point unit using Tomasulo's algorithm [1]

The instruction queue maintains a certain number of instructions in the queue. Instructions are issued in-order and each reservation station consists of an instruction waiting for a functional unit to get free or for the operands to be evaluated. The function of load and store buffers is to keep track of data or addresses going to or from memory and serve as reservation stations. The output of memory and functional units is broadcasted by the Common Data Bus (CDB) to all units except load station.
An instruction goes through the following steps:

2.1.1 Issue – This basically involves fetching the instruction from the queue in FIFO order to detect hazards. According to the instruction, if a corresponding reservation station is available, the instruction is issued to it along with the operand values, if available. If operands are unavailable, tag names of reservation stations producing them are assigned to each register value. All name dependences including WAW and WAR hazards are taken care of in this stage.

2.1.2 Execute– In this step, we wait if the operands are not available and keep tolling the Common Data Bus for the value of the operand. On availability, operand is immediately forwarded to the reservation station waiting for it. Once all data

values are read, operation is performed. In this way, RAW hazards are avoided by stalling for operands and by forwarding on accessibility.

There is one possibility that multiple instructions can be ready for execution in the same cycle. Hence, it is necessary to make choices from the ready instructions and decide which one can occupy the functional unit first.

Depending on the CPU architecture, load-store units need to compute the effective address first. Once the address is calculated, the instruction can be executed. Effective address is the key which allows the tracking of load-store execution and prevent memory hazards.

If an exception occurs, all prior branch instructions are completed and only then the control is allowed to jump to the exception routine. Branch prediction processors need to make sure that the prediction was actually true by executing all the instructions and making sure that the instruction might have actually raised an exception.

2.1.3 ***Write back*** – Once the result is ready, it is passed to the CDB. From CDB, it is forwarded to the waiting reservations and register file including the store units. When the data and address for a store become available, the store is executed by the memory unit when it is free.

Tomasulo is a distributed control algorithm - which means that the data items responsible for hazard detection are associated with the reservation stations, register file as well as the load/store buffers. These data items are tags which, in the proposed implementation are 8-bit items that give rise to an extended set of alias registers. These tags are reservation station (RS) tags that come into picture when an instruction is issued and once the result is available, these tags are discarded after forwarding it to other units.

Every reservation station has particular fields for functioning:

Op – Refers to the function to be done on operands.

Qj, Qk – RS tags which will provide the corresponding source data. A zero indicates an operand is available.

Vj, Vk– Source operands data.

A – Holds the location for load/store.

Busy– Denotes RS is busy.

2.2 Re-order buffer (ROB) acting as additional registers

The ROB also serves as an extended set of registers and they hold the result from the execution time till the instruction commits. The difference between a RS and ROB is -

the register file is not updated as soon as the result is available, but, only after the instruction commits. Until then, the source values to pending instructions are supplied by the ROB.

2.3 Use a physical register file and a Register Alias Table (RAT)[2]

A register alias table maps the architectural registers (RAT index) to the physical registers (RAT values). From this table, the mapping of two source operands and the destination register is read. A free list provides the physical register names[2]. The processor gets rid of the physical alias register once it sure that no other instruction needs its value.

CHAPTER 3 –TOMASULO: THE DESIGN

3.1 The Proposed Design Approach

To design a CPU, first figure out the Instruction set you want to support. The proposed design supports ADD, MUL, SUB and to some extent DIV instructions. I wished to implement the following type of CPU:

- 40-bit long instruction
- A register file of 256 registers (0 h – FF h). However, I have used only first 16 registers for now.
- The instruction format looks like below:

4 bit	8 bit	8 bit	8 bit	12 bit
Opcode	Destination	Source 1	Source 2	Address

Table 3.1 Instruction Format

The 4 bit opcode maps as below:

Opcode	Operation
0	ADD
A	SUB
F	MUL
B	DIV

Table 3.2 Opcode Maps

Hence, instructions and their equivalent Hex format look like this:

ADD R0, R1, R2	00001010001
MUL R0, R3, R1	F0003010002
SUB R3, R4, R2	A0303030005

Let me outline the different units used for this design:

(1) Instruction Queue
(2) Issue unit
(3) Register File
(4) Multiplexer unit
(5) Reservation station 1 with ADD Functional unit
(6) Reservation station 2 with SUB Functional unit
(7) Reservation station 3 with MUL Functional unit
(8) Reservation station 1 with DIV Functional unit

All these units have been combined in a top-level module for a complete functioning design. I will now explain the block diagram, algorithm flow and give a detailed view of each unit.

The flow chart for Tomasulo goes like this:

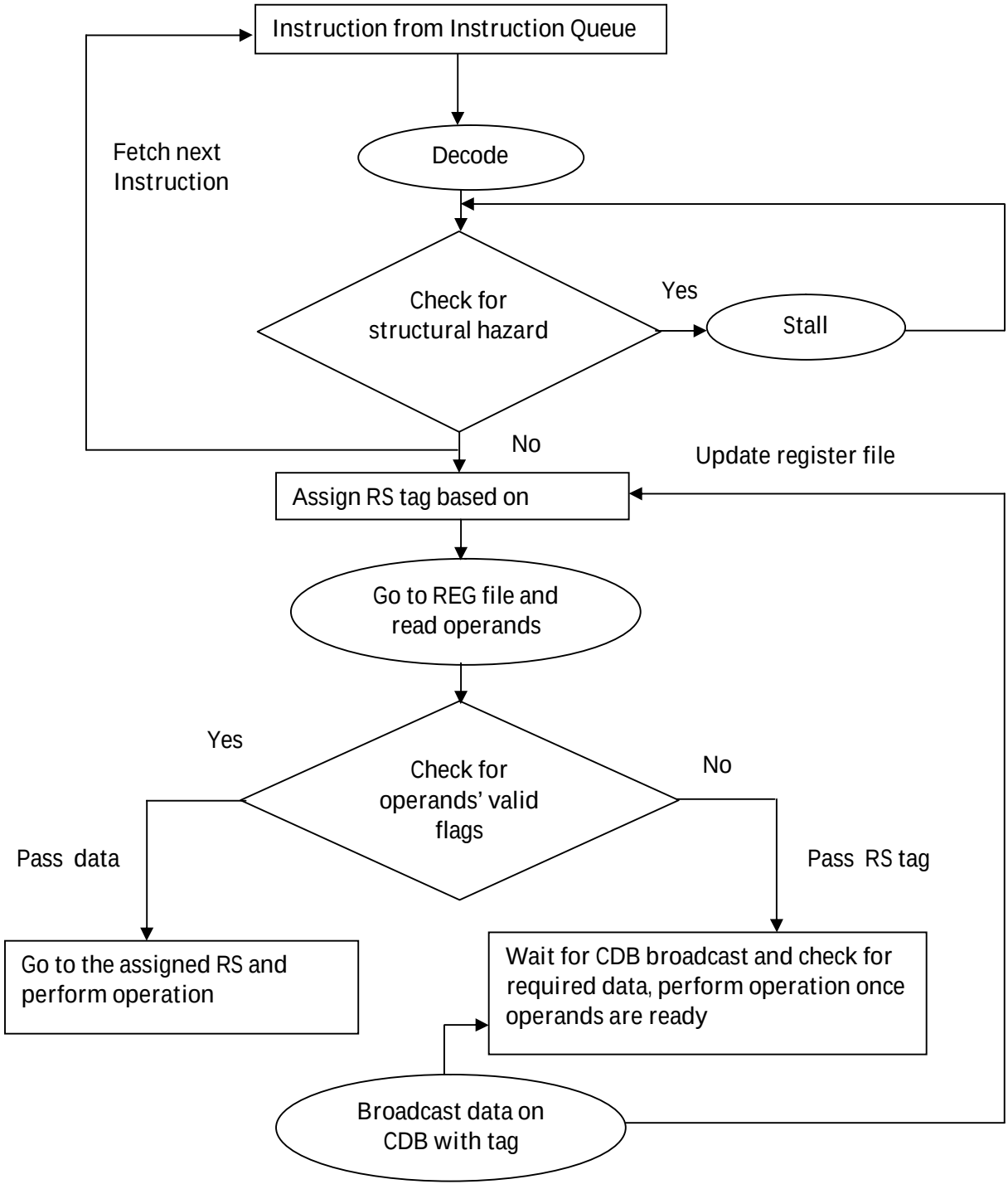

Fig.3.1 Algorithm flow chart

Now, consider the individual blocks of the design.

3.1.1 Instruction Queue Unit

The instruction queue maintains a window of 4 instructions in the queue. Also, the main function is to accept the first instruction, decode it and pass the decoded signals to the issue stage. Furthermore, there is one more stall signal from the issue stage which indicates whether the instruction pointer should be incremented to the next instruction. If the stall signal is high, then the current decoded instruction is held in the register and the output signals of this stage do not change. The decoding happens at the positive edge of the clock and the entire design is synchronous.

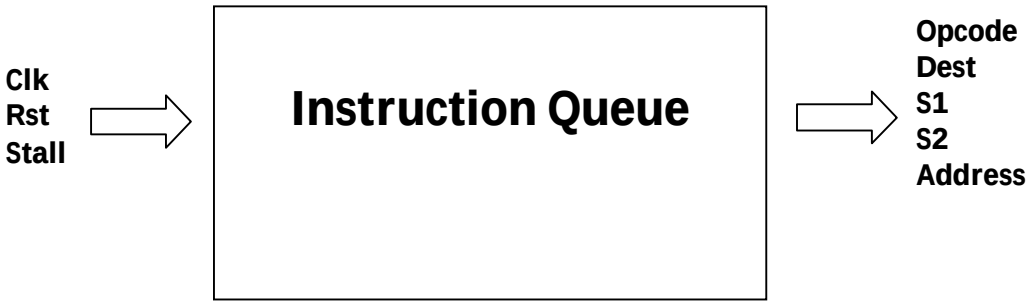

Fig.3.2 Instruction Queue

3.1.2 Issue Unit (Dispatch Stage)

From the instruction queue the decoded signals, which are: opcode, source1, source2, destination and address are passed to the issue unit in the pipeline.

Each reservation station consisting of the functional unit has its own flag that indicates whether it's busy or idle. These flags are used in the issue stage to determine which functional unit is idle to accept a new instruction.

Based on the opcode, the corresponding RS flag is checked. If the unit is free, a certain set of signals are sent to the register file which is the next stage in the pipeline. Also, when the unit is idle, the tag of the corresponding RS is assigned to this instruction. This tag is helpful in determining which RS is going to produce the required register result.

If the opcode is invalid, all the signals sent to the register file are kept on hold and the stall signal is activated.

If the RS unit is not free the stall signal is made active high and this signal goes to the instruction queue. If the stall signal is low the counter in the IQ is incremented and the next instruction from the queue is decoded.

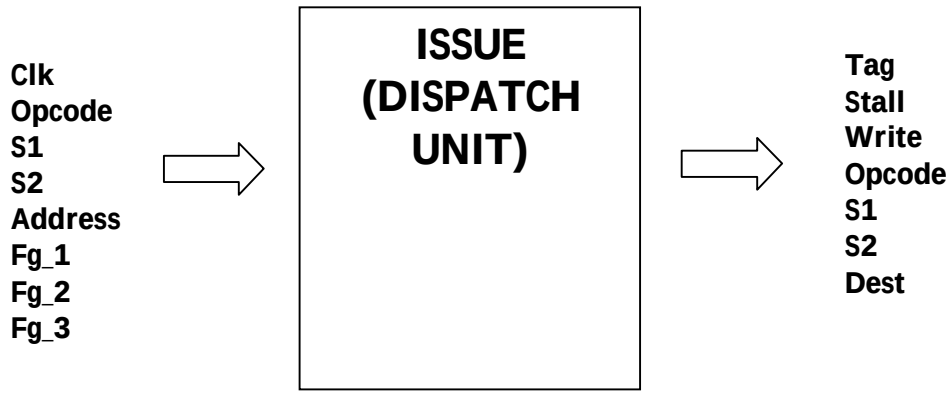

Fig.3.3 Issue Unit (Dispatch Unit)

3.1.3 Register File

The basic register file is shown in the figure below:

R	0	1	2	3	4	5
Data	(R0)	(R1)	(R2)	(R3)	(R4)	(R5)
Valid	1	1	1	1	1	1
Address	0	0	0	0	0	0

Table 3.3 Register File

In all, there are 256 registers available as per the 8-bit field assigned for the register operand field. However, for simplicity, I will be using only the first 16 registers from R0 to R15.

The basic functioning of a register file requires dynamic register renaming. This is where Tomasulo's algorithm uses reservation station as extra registers to rename the operands. As per the destination register in the instruction, the corresponding register's address in the above table is swapped with the RS tag that is supposed to produce the value for that register. At the same time, the valid flag of the corresponding register is made zero to indicate to the future instructions that the register value is being computed by some other functional unit. This takes care of the RAW hazard and the instructions waiting for the register value are stalled until the result is broadcasted on the Common Data Bus (CDB).

The register file supports forwarding. In other words, there is a common data bus in the design which broadcasts the result to all the units in the design along with the register file. In return, the register addresses contain RS tag if they are destination registers for instructions in execution. Thus, if the CDB tag matches the address of the register in the register file, the CDB data is put into the register. As a result, the register file automatically updates with the help of the RS tag.

Since this is a pipeline, at this stage, the instruction fields are registered and passed on to the next stage to the multiplexer.

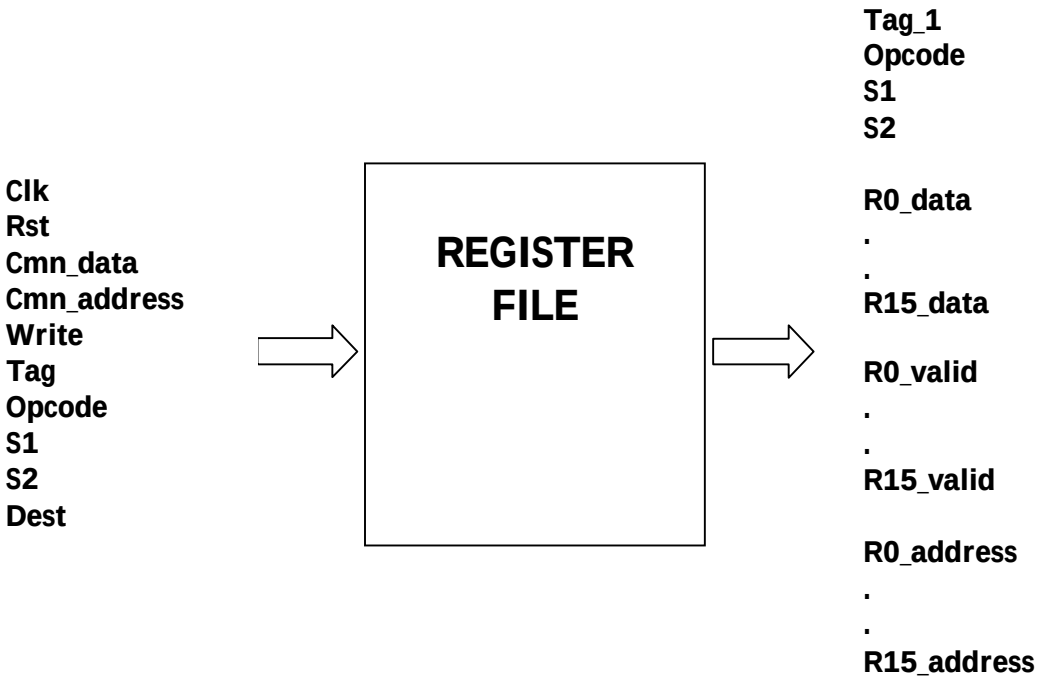

Fig. 3.4 Register File

3.1.4 Multiplexer

As we have seen, the source operands, opcode, code and destination are carried throughout the pipeline. The job of this multiplexer is to look at the source operands and read the correct values to pass it to the appropriate reservation station.
This involves a couple of steps as WAW, WAR and RAW hazards are resolved at this stage. Here, the multiplexer checks for the valid flags of the corresponding source operands. If they are valid, the operand values along with the valid signal are sent to the assigned reservation station. Since the RS tag is still with us, we can send the appropriate data to the correct RS based on it.

However, if the valid flag of one or more operands is low, the valid signals along with the corresponding RS tags are sent to the reservation stations. As I said before, these RS tags indicate the functional unit responsible to produce the result of the registers.

Also, in case the tag is invalid, the entire packet of signals is held and left unchanged and the instruction pointer jumps to the next instruction.

So, basically the multiplexer selects the correct data values and sends them to the required reservation station. The following figure gives a top level view of the multiplexer:

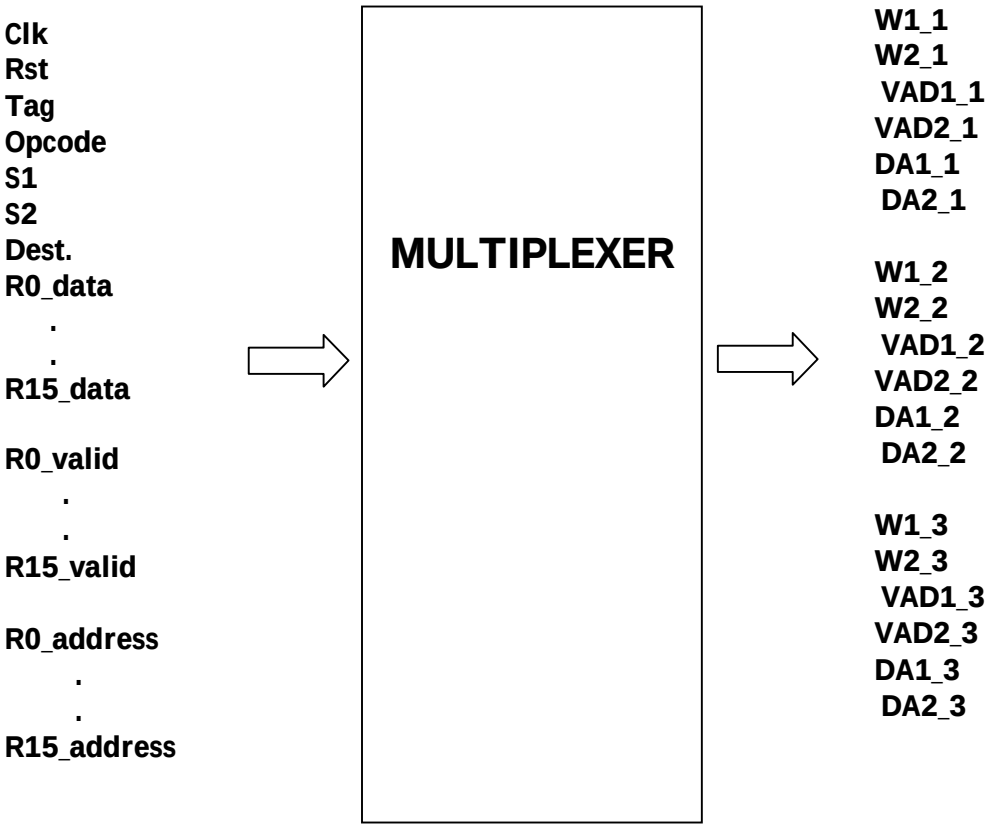

Fig.3.5 Multiplexer Unit

Now, let's take a look at the structure of a reservation station.

3.1.5 Adder/ Multiplier/ Subtractor Reservation Station

Each reservation station has the following structure to support the Tomasulo algorithm:

RS	STATUS	Opcode	Dest.	Src1 (value)	Vld1 (valid flag)	Add1 (RS tag)	Src2	Vid2	Add2
ADD	0								
MUL	0								
SUB	0								
DIV	0								

Table 3.4 Reservation Station Structure

As explained before, the Src1 and Src2 reflect the actual values of the registers known as Vj and Vk. The valid flags are indicated as Vj_valid and Vk_valid and they indicate the status of the operands, whether they are available or not. The address flags contain the RS tag if the operands are not available readily and we need to wait for the data to come in from the CDB.

If both operands are readily available, the result is computed according to the opcode sent to the CDB. If the data is not available, we wait for CDB to broadcast the required result. This means, when the CDB broadcasts the result, it also gives the RS tag along with it and when any RS waiting for that register result immediately gets the data by checking RS tag. Thus, the result is forwarded to the functional unit as well as the register file, so we save the extra cycle needed to get the result from the register file.

The STATUS field indicates whether the RS is busy or idle. These are the flags checked in the ISSUE stage to know which RS is empty. For all reservation stations, the same format is used. The only thing which varies is the operation to be performed depending on the opcode.

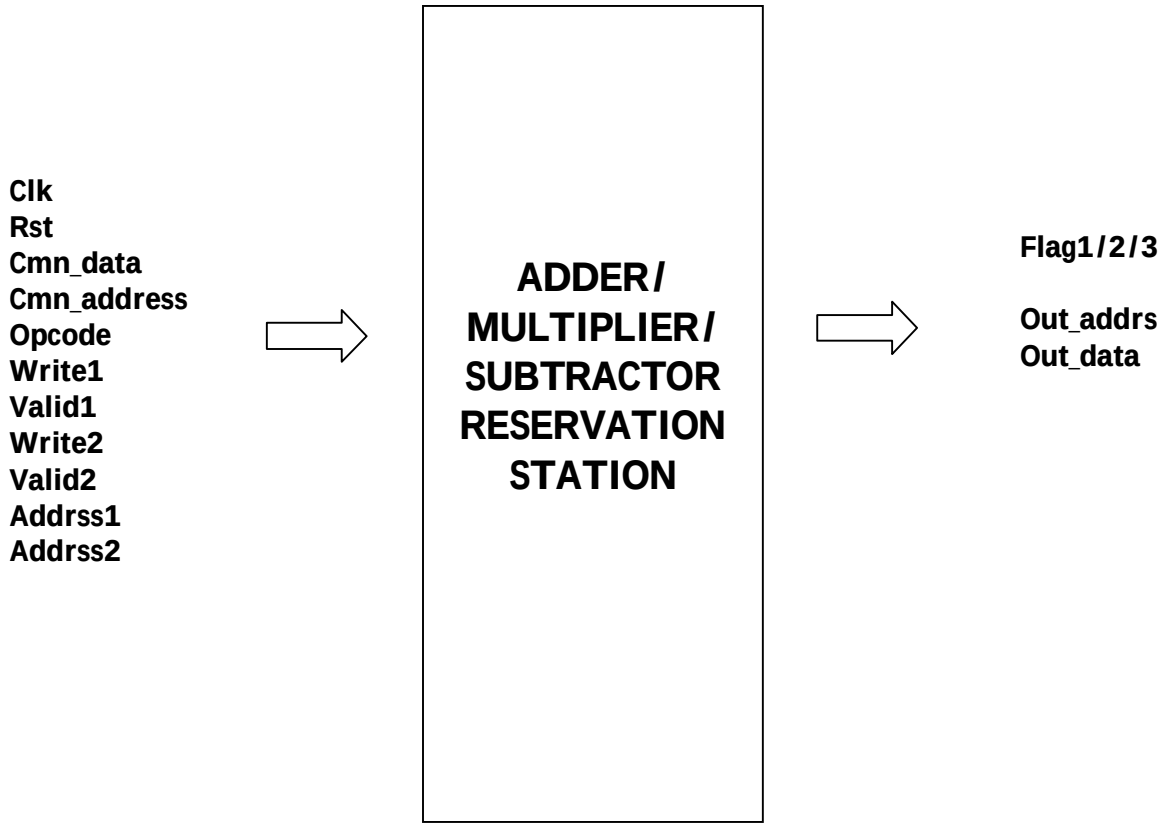

Fig.3.6 Adder/Multiplier/ Subtractor/ Reservation Station

CHAPTER 4 - TESTING AND SYNTHESIS

4.1 Testing And Schematics

In order to test this design, all we need is clock and reset. The instructions are pre-fed in a memory array and the Tomasulo core should schedule instructions and increase throughout. Remember, the main reason Tomasulo was implemented was to save cycles and explore instruction level parallelism.
The testbench top view would look like this:

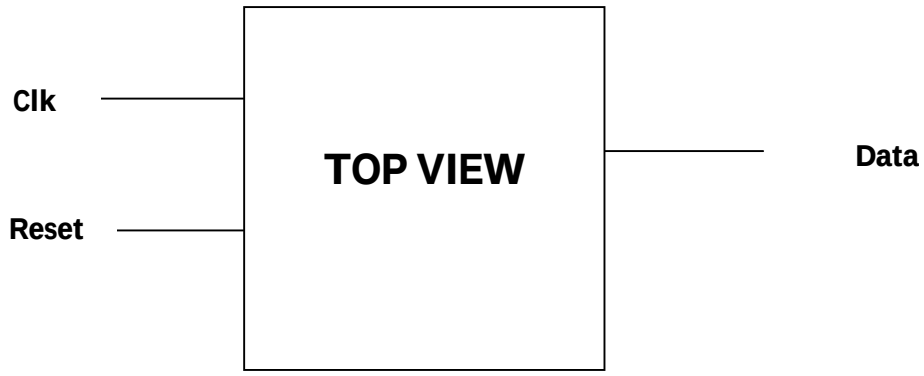

Fig.4.1 Testbench Top View

At each clock, a new instruction goes in; the previous instruction goes to the issue stage and so on as the 5-stage pipeline continues. Also, it depends whether there are stalls or not depending on how the structural and data hazards are resolved.
The instructions used in the Instruction Queue can be listed as below:

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

The schematic after synthesis looks as below:

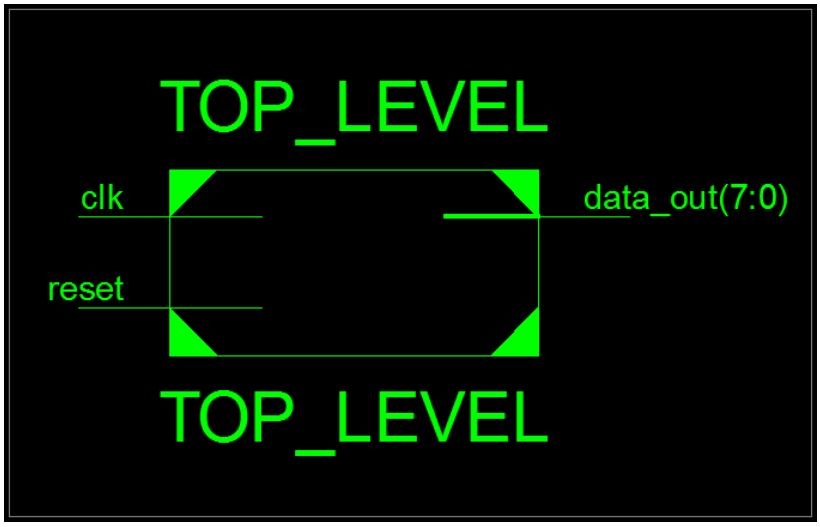

Fig 4.2 Schematic of Top-Level

The detailed Tomasulo design generated on Xilinx ISE 14.2 is shown in the following figure:

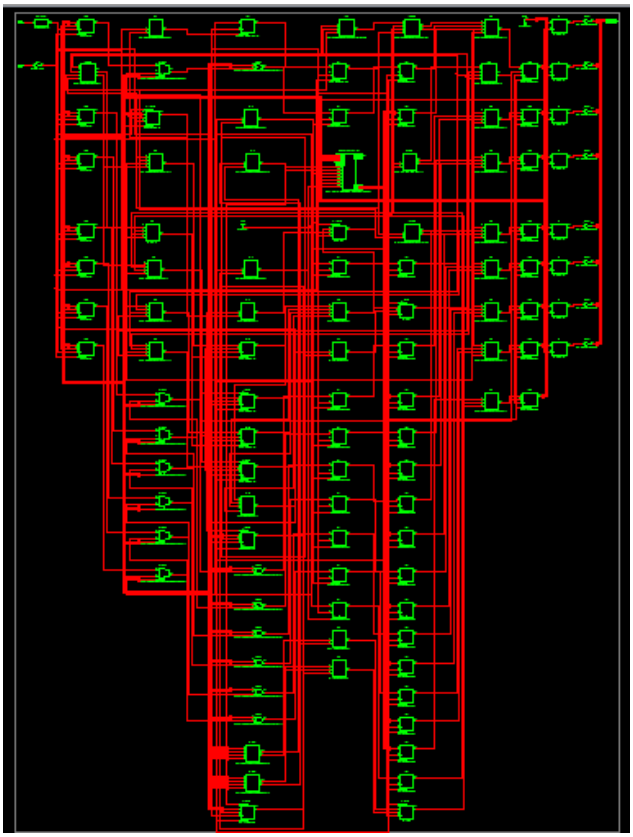

Fig 4.3 Schematic of Tomasulo Design

A detailed schematic view of the multiplier is shown below:

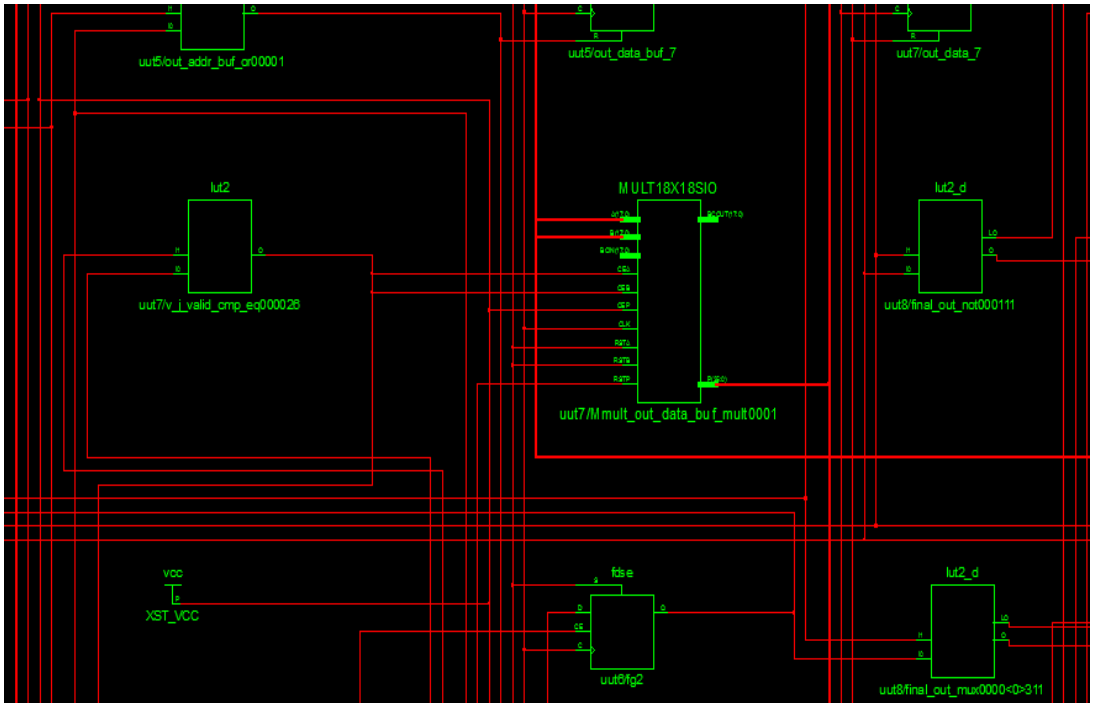

Fig 4.4 Schematic of Multiplier

As per the resources available on Spartan3E, the design summary report generated is as below:

Project File:	Grad_Proj.xise	**Parser Errors:**	No Errors
Module Name:	TOP_LEVEL	**Implementation State:**	Placed and Routed
Target Device:	xc3s250e-5vq100	**• Errors:**	No Errors
Product Version:	ISE 14.2	**• Warnings:**	0 Warnings
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	61	4,896	1%	
Number of 4 input LUTs	32	4,896	1%	
Number of occupied Slices	49	2,448	2%	
Number of Slices containing only related logic	49	49	100%	
Number of Slices containing unrelated logic	0	49	0%	
Total Number of 4 input LUTs	32	4,896	1%	
Number of bonded IOBs	10	66	15%	
Number of BUFGMUXs	1	24	4%	
Number of MULT18X18SIOs	1	12	8%	
Average Fanout of Non-Clock Nets	2.88			

Fig 4.5 Design Summary Report

4.2 Simulation Waveforms

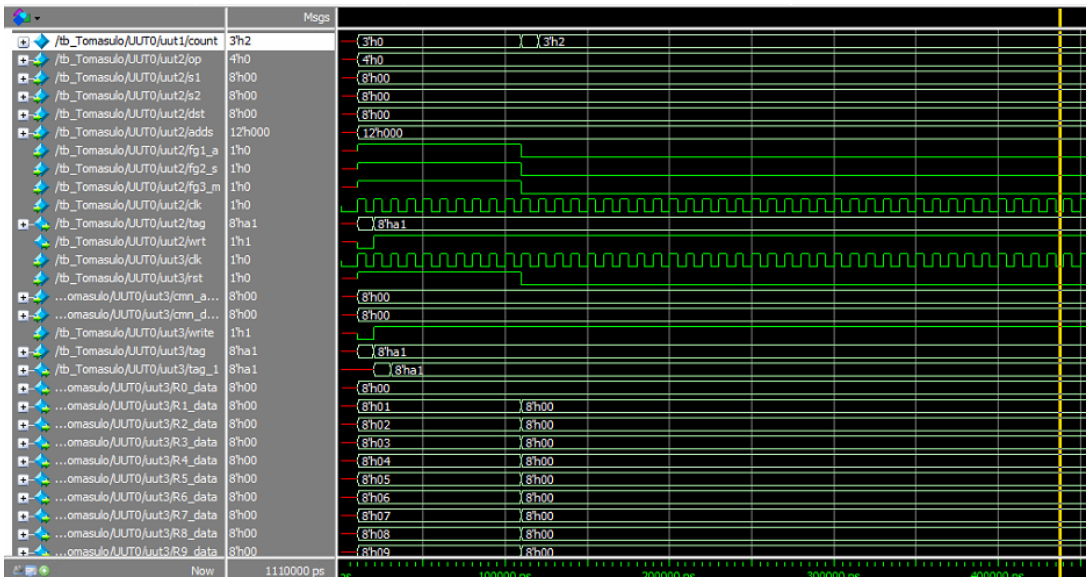

Fig. 4.6 Simulation Waveform 1

As shown in the figure above, a tag 8'hA1 is assigned to the instruction ADD and this is the Reservation station ID corresponding to adder functional unit.

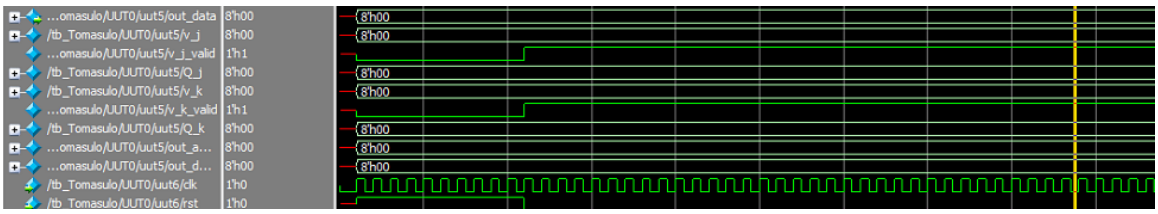

Fig. 4.7 Simulation Waveform 2

The valid flags and the reservation station fields Vj, Vk are shown in figure above and these flags change as the instruction executes.

4.3 Synthesis Report

Once the design has been synthesized a complete Synthesis Report is generated by Xilinx.

--> Reading design: TOP_LEVEL.prj

TABLE OF CONTENTS

3) Design Hierarchy Analysis

4) HDL Analysis

5) HDL Synthesis

5.1) HDL Synthesis Report

6) Advanced HDL Synthesis

6.1) Advanced HDL Synthesis Report

7) Low Level Synthesis

8) Partition Report

9) Final Report

9.1) Device utilization summary

9.2) Partition Resource Summary

9.3) TIMING REPORT

```
=========================================================================
*                      Synthesis Options Summary                        *
=========================================================================
---- Source Parameters
Input File Name                    : "TOP_LEVEL.prj"
Input Format                       : mixed
Ignore Synthesis Constraint File   : NO

---- Target Parameters
Output File Name                   : "TOP_LEVEL"
Output Format                      : NGC
Target Device                      : xc3s250e-5-vq100

---- Source Options
Top Module Name                    : TOP_LEVEL
Automatic FSM Extraction           : YES
FSM Encoding Algorithm             : Auto
Safe Implementation                : No
FSM Style                          : LUT
RAM Extraction                     : Yes
RAM Style                          : Auto
ROM Extraction                     : Yes
Mux Style                          : Auto
```

Decoder Extraction	: YES
Priority Encoder Extraction	: Yes
Shift Register Extraction	: YES
Logical Shifter Extraction	: YES
XOR Collapsing	: YES
ROM Style	: Auto
Mux Extraction	: Yes
Resource Sharing	: YES
Asynchronous To Synchronous	: NO
Multiplier Style	: Auto
Automatic Register Balancing	: No

---- Target Options

Add IO Buffers	: YES
Global Maximum Fanout	: 100000
Add Generic Clock Buffer(BUFG)	: 24
Register Duplication	: YES
Slice Packing	: YES
Optimize Instantiated Primitives	: NO
Use Clock Enable	: Yes
Use Synchronous Set	: Yes
Use Synchronous Reset	: Yes
Pack IO Registers into IOBs	: Auto
Equivalent register Removal	: YES

---- General Options

Optimization Goal	: Speed
Optimization Effort	: 1
Keep Hierarchy	: No
Netlist Hierarchy	: As_Optimized
RTL Output	: Yes
Global Optimization	: AllClockNets
Read Cores	: YES
Write Timing Constraints	: NO
Cross Clock Analysis	: NO
Hierarchy Separator	: /
Bus Delimiter	: <>
Case Specifier	: Maintain
Slice Utilization Ratio	: 100
BRAM Utilization Ratio	: 100
Verilog 2001	: YES
Auto BRAM Packing	: NO
Slice Utilization Ratio Delta	: 5

CHAPTER 5 – CONCLUSION AND FUTURE EXPANSION

5.1 Conclusion

Building this entire design was a challenging task as dynamic scheduling can be done in many different ways as pointed out earlier in chapter-2. As a result, I had to come up a method of dynamic tag assignment based on the proposed design. As we saw, this dynamic assignment takes place in the register file stage where the destination register is replaced with the reservation tag. Although, I had been introduced to Tomasulo's algorithm in ECE 622, I was nowhere near the implementation details. This project gave me an insight on the actual hardware support required to design a scheduler.

For simplicity reasons, this 40-bit processor does not support Load/Store and Branch instructions. The main goal of this project was to develop coding skills in Xilinx ISE 14.2 using Verilog 2001. Hence, I can conclude that this task has given me a better view of HDL coding and design.

5.2 Future Expansion

As I pointed out earlier, this processor can indeed incorporate a Load/Store Unit and a separate data memory. Also, additional mechanism for BRANCH and DIVISION instructions can also be added. Due to high level of complexity I was limited to design a single instruction issue processor. But, this design can be replicated to design a Super Scalar Instruction Processor. All these future enhancements can be well implemented in this design with considerable efforts.

References

1. *Computer-Architecture-A-Quantitative-Approach-5th-Edition* by John L. Hennessy and David A. Patterson

2. *Towards A Viable Out-Of-Order Soft Core: Copy-Free, Checkpointed Register Renaming* by Kaveh Aasaraai and Andreas Moshovos, Electrical and Computer Engineering, University of Toronto

3. *Optimality of Tomasulo's Algorithm* by Tian Sang and Lin Liao, Department of Computer Science and Engineering, University of Washington.

4. *Sixth Lecture: Chapter 3: CISC Processors Tomasulo Scheduling and IBM System 360/91*

5. *web.cs.dal.ca/~mheywooh/CSCI3121/Pipe/07-DynSchedule-pt1.pdf*

6. *Register Renaming: people.ee.duke.edu/~sorin/ece252/lectures/4.2-tomasulo.pdf*

Appendix

Testbench:

```
`timescale 1 ns / 10 ps
module tb_Tomasulo;

	// Inputs
	reg clk;
	reg reset;

	// Outputs
	wire [7:0] data_out;

	// Instantiate the Unit Under Test (UUT)
	TOP_LEVEL UUT0 (
		.clk(clk),
		.reset(reset),
		.data_out(data_out)
	);
	initial
		// Initialize Inputs
		begin
		clk = 1'b0;
		#5
		forever #5 clk = ~clk;
		end
	initial
		begin
	  #10 reset = 1'b1;
```

```
        // Wait 100 ns for global reset to finish
        #100 reset = 0;
        // Add stimulus here
        #1000 $finish;
        End
endmodule
```

Source Code:

The multiple source files are shown below:

Top Level:

```
`timescale 1 ns / 10 ps
module TOP_LEVEL(
        input clk,
        input reset,
        output reg [7:0] data_out

    );

        wire flag1, flag2, flag3;
        wire stl, WR;
        wire [3:0] op_s1, op_s2, op_s3, op_s4;
        wire [7:0] dst1, s1_1, s2_1, dst2, s1_2, s2_2, s1_3, s2_3, univ_tag, tag_pass;
        wire [11:0] address;
        wire [7:0] cdb_add, cdb_data;
        wire [7:0] R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15;
        wire V0, V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14, V15;
        wire [11:0] adrs0, adrs1, adrs2, adrs3, adrs4, adrs5, adrs6, adrs7, adrs8, adrs9,
                        adrs10, adrs11, adrs12, adrs13, adrs14, adrs15;
```

```
        wire WA1, WA2, VADA1, VADA2,WS1, WS2, VADS1, VADS2, WM1, WM2, VADM1,
VADM2;

        wire [7:0]  DAA1, DAA2, DAS1, DAS2, DAM1, DAM2;

        wire [7:0] tag1, tag2,tag3, out1, out2, out3;


                Inst_Q uut1(.clk(clk), .rst(reset), .stall(stl), .opcode(op_s1),

                                                .dest(dst1), .src1(s1_1), .src2(s2_1), .addrs(address));


                Issue uut2(.op(op_s1), .s1(s1_1), .s2(s2_1), .dst(dst1), .adds(address),

                                        .fg1_a(flag1), .fg2_s(flag2), .fg3_m(flag3), .clk(clk),

                                        .tag(univ_tag), .stall(stl), .opcod(op_s2), .src1(s1_2),
.src2(s2_2), .dstn(dst2), .wrt(WR));


                Reg_File uut3(.clk(clk), .rst(reset), .cmn_addrs(cdb_add), .cmn_data(cdb_data),
.write(WR),

                                        .dest(dst2), .tag(univ_tag), .S1(s1_2),
.S2(s2_2),.opcode(op_s2), .src1(s1_3),

                                        .src2(s2_3), .opcd(op_s3), .tag_1(tag_pass),

                                        .R0_data(R0), .R1_data(R1), .R2_data(R2),
.R3_data(R3), .R4_data(R4), .R5_data(R5),

                                        .R6_data(R6), .R7_data(R7), .R8_data(R8),
.R9_data(R9), .R10_data(R10), .R11_data(R11),

                                        .R12_data(R12), .R13_data(R13),
.R14_data(R14), .R15_data(R15),


                                        .R0_valid(V0),
.R1_valid(V1),.R2_valid(V2), .R3_valid(V3), .R4_valid(V4),

                                        .R5_valid(V5), .R6_valid(V6),
        .R7_valid(V7), .R8_valid(V8), .R9_valid(V9),

                                        .R10_valid(V10), .R11_valid(V11),
.R12_valid(V12), .R13_valid(V13),

                                        .R14_valid(V14), .R15_valid(V15),
```

```
                                                    .R0_addrs(adrs0), .R1_addrs(adrs1),
.R2_addrs(adrs2), .R3_addrs(adrs3), .R4_addrs(adrs4),
                                                    .R5_addrs(adrs5), .R6_addrs(adrs6),
.R7_addrs(adrs7), .R8_addrs(adrs8), .R9_addrs(adrs9),
                                                    .R10_addrs(adrs10), .R11_addrs(adrs11),
.R12_addrs(adrs12), .R13_addrs(adrs13),
                                                    .R14_addrs(adrs14), .R15_addrs(adrs15));


                MUX uut4(.clk(clk), .rst(reset), .S1(s1_3), .S2(s2_3), .opcode(op_s3),
                                                    .valid_R0(V0), .valid_R1(V1), .valid_R2(V2), .valid_R3(V3),
.valid_R4(V4), .valid_R5(V5),
                                                    .valid_R6(V6), .valid_R7(V7), .valid_R8(V8), .valid_R9(V9),
.valid_R10(V10), .valid_R11(V11),
                                                    .valid_R12(V12), .valid_R13(V13), .valid_R14(V14),
.valid_R15(V15),


                                                    .data_R0(R0), .data_R1(R1), .data_R2(R2), .data_R3(R3),
.data_R4(R4), .data_R5(R5),
                                                    .data_R6(R6), .data_R7(R7), .data_R8(R8), .data_R9(R9),
.data_R10(R10), .data_R11(R11),
                                                    .data_R12(R12), .data_R13(R13), .data_R14(R14),
.data_R15(R15),


                                                    .addrs_R0(adrs0), .addrs_R1(adrs1), .addrs_R2(adrs2),
.addrs_R3(adrs3), .addrs_R4(adrs4),
                                                    .addrs_R5(adrs5), .addrs_R6(adrs6), .addrs_R7(ards7),
.addrs_R8(adrs8), .addrs_R9(adrs9),
                                                    .addrs_R10(adrs10), .addrs_R11(adrs11), .addrs_R12(adrs12),
.addrs_R13(adrs13),
                                                    .addrs_R14(adrs14), .addrs_R15(adrs15),


                                                    .opcod(op_s4), .W1_1(WA1), .W2_1(WA2),
.VAD1_1(VADA1), .VAD2_1(VADA2), .DA1_1(DAA1),
                                                    .DA2_1(DAA2),
```

```
                                        .W1_2(WS1), .W2_2(WS2), .VAD1_2(VADS1),
.VAD2_2(VADS2), .DA1_2(DAS1), .DA2_2(DAS2),


                                        .W1_3(WM1), .W2_3(WM2), .VAD1_3(VADM1),
.VAD2_3(VADM2), .DA1_3(DAM1), .DA2_3(DAM2));


            RS1_ADD uut5(.clk(clk), .rst(reset), .opcode(op_s4), .cmn_addr(cdb_add),
.cmn_data(cdb_data),

                                        .write1(WA1), .valid1(VADA1), .write2(WA2),
.valid2(VADA2), .data_addr1(DAA1),

                                        .data_addr2(DAA2), .fg1(flag1), .out_addr(tag1),
.out_data(out1));


            RS2_SUB uut6(.clk(clk), .rst(reset), .opcode(op_s4), .cmn_addr(cdb_add),
.cmn_data(cdb_data),

                                        .write1(WS1), .valid1(VADS1), .write2(WS2),
.valid2(VADS2), .data_addr1(DAS1),

                                        .data_addr2(DAS2), .fg2(flag2), .out_addr(tag2),
.out_data(out2));


            RS3_MUL uut7(.clk(clk), .rst(reset), .opcode(op_s4), .cmn_addr(cdb_data),
.cmn_data(cdb_data),

                                        .write1(WM1), .valid1(VADM1), .write2(WM2),
.valid2(VADM2), .data_addr1(DAM1),

                                        .data_addr2(DAM2), .fg3(flag3), .out_addr(tag3),
.out_data(out3));


            SORTING uut8( .clk(clk), .rst(reset), .tag1(tag1), .tag2(tag2), .tag3(tag3), .out1(out1),

                                        .out2(out2), .out3(out3), .flag1(flag1), .flag2(flag2),
.flag3(flag3), .tag_out(cdb_add),

                                        .final_out(cdb_data));


            always @(posedge clk)

                  data_out <= cdb_data;

endmodule
```

Instruction Queue:

```
`timescale 1ns / 10 ps
module Inst_Q(clk,rst,stall,opcode,dest,src1,src2,addrs);
    input clk;
    input rst;
    input stall;
    output [3:0]opcode;
    output [7:0]dest;
    output [7:0]src1;
    output [7:0]src2;
    output [11:0]addrs;

    reg [3:0] opcode;
    reg [7:0] dest;
    reg [7:0] src1;
    reg [7:0] src2;
    reg [11:0] addrs;

        reg [39:0] queue [0:4];
        reg [2:0] count;

        always @(posedge clk)
        begin
                if(rst == 1)
                begin

                        //reset state
```

```
            opcode      <= 4'b0000;
            src1        <= 8'b0000_0000;
            src2        <= 8'b0000_0000;
            dest        <= 8'b0000_0000;
            addrs       <= 12'b0000_0000_0000;
            count       <= 0;


            //initialize array with instructions
            queue [0] <= 40'b0000_0000_0000_0000_0001_0000_0100_0000_0000_0000;
            queue [1] <= 40'b1111_0000_0000_0000_0011_0000_0001_0000_0000_0000;
            queue [2] <= 40'b1010_0000_0011_0000_0100_0000_0010_0000_0000_0000;
            queue [3] <= 40'b0000_0000_0010_0000_0000_0000_0001_0000_0000_0000;
            queue [4] <= 40'b1010_0000_0110_0000_1000_0000_0000_0000_0000_0000;

end
else if (stall == 1'b1)
                        //hold values for stall
                        begin
                        opcode      <= opcode;
                        src1        <= src1;
                        src2        <= src2;
                        dest        <= dest;
                        addrs       <= addrs;
                        count       <= count;
                        end
else if(count > 4)
                        //when the counter is four, reset
                        begin
```

```
                opcode    <= 4'b0000;
                src1      <= 8'b0000_0000;
                src2      <= 8'b0000_0000;
                dest      <= 8'b0000_0000;
                addrs     <= 12'b0000_0000_0000;
                count     <= 0;

                end

        else
                begin
                        count   <=      count + 1;
                        opcode  <=      queue [count][39:36];
                        src1    <=      queue [count][27:20];
                        src2    <=      queue [count][19:12];
                        dest    <=      queue [count][35:28];
                        addrs   <=      queue [count][11:0];

                end

        end
endmodule
```

Issue:

```
`timescale 1ns / 10ps
module Issue(
   input [3:0] op,
   input [7:0] s1,
   input [7:0] s2,
   input [7:0] dst,
```

```
    input [11:0] adds,
    input fg1_a,                                    //flags indicating status of all RSs
    input fg2_s,
    input fg3_m,
    input clk,
    input rst,
    output reg [7:0] tag,                   //RS # to which instruction is issued
            output reg stall,                       //input to InstQ to control Instruction pointer
            output reg [3:0] opcod,
            output reg [7:0] src1,
    output reg [7:0] src2,
            output reg [7:0] dstn,
            output reg wrt
            );

always @ (posedge clk)
        begin
                if (rst) begin
                        tag <= 0;
                        stall <= 0;
                        opcod <= 0;
                        src1 <= 0;
                        src2 <= 0;
                        dstn <= 0;
                        wrt <= 0;
                end
                else begin
                case (op)
```

```
0:      begin                                   //if operation is ADD
            if(fg1_a == 1'b1) //check Adder RS flag if free
                begin

                tag <= 8'hA1;       //tag corresponding to Adder1

                case(dst)           //check dest register to assign RS tag in next
stage

                    0 : wrt <= 1;
                    1 : wrt <= 1;
                    2 : wrt <= 1;
                    3 : wrt <= 1;
                    4 : wrt <= 1;
                    5 : wrt <= 1;
                    6 : wrt <= 1;
                    7 : wrt <= 1;
                    8 : wrt <= 1;
                    9 : wrt <= 1;
                    'hA : wrt <= 1;
                    'hB : wrt <= 1;
                    'hC : wrt <= 1;
                    'hD : wrt <= 1;
                    'hE : wrt <= 1;
                    'hF : wrt <= 1;
            default : wrt <= 1;

                endcase
                //send couple of signals to RS as well as GPR
                stall <= 1'b0;
```

```
                    opcod <= op;
                    src1 <= s1;
                    src2 <= s2;
                    dstn <= dst;
                    end

                else                                    //if RS not free

                    begin
                        //hold all signals
                        tag <= tag;
                        wrt <= wrt;
                        stall <= 1'b1;
                        opcod <= opcod;
                        src1 <= src1;
                        src2 <= src2;
                        dstn <= dstn;
                    end

            end                                         //end of operation ADD

        //          future expansion
        'hA:  begin                                     //if operation is SUB
                if(fg2_s == 1'b1) //check Subr RS flag if free
                    begin

                    tag <= 8'hB1;       //tag corresponding to Subtractor1

                    case(dst)           //check dest register to assign RS tag in next
stage
```

```
                0 : wrt <= 1;
                1 : wrt <= 1;
                2 : wrt <= 1;
                3 : wrt <= 1;
                4 : wrt <= 1;
                5 : wrt <= 1;
                6 : wrt <= 1;
                7 : wrt <= 1;
                8 : wrt <= 1;
                9 : wrt <= 1;
                'hA : wrt <= 1;
                'hB : wrt <= 1;
                'hC : wrt <= 1;
                'hD : wrt <= 1;
                'hE : wrt <= 1;
                'hF : wrt <= 1;
default : wrt <= 1;

        endcase
        //send couple of signals to RS as well as GPR
        stall <= 1'b0;
        opcod <= op;
        src1 <= s1;
        src2 <= s2;
        dstn <= dst;
        end

else                                    //if RS not free
```

```
                begin
                        //hold all signals
                        tag <= tag;
                        wrt <= wrt;
                        stall <= 1'b1;
                        opcod <= opcod;
                        src1 <= src1;
                        src2 <= src2;
                        dstn <= dstn;
                end

        end                                     //end of operation SUB

'hF: begin                                      //if operation is ADD
                if(fg3_m == 1'b1)//check Adder RS flag if free
                        begin

                        tag <= 8'hC1;       //tag corresponding to Adder1

                        case(dst)           //check dest register to assign RS tag in next
stage

                                0 : wrt <= 1;
                                1 : wrt <= 1;
                                2 : wrt <= 1;
                                3 : wrt <= 1;
                                4 : wrt <= 1;
                                5 : wrt <= 1;
                                6 : wrt <= 1;
```

```
                7 : wrt <= 1;
                8 : wrt <= 1;
                9 : wrt <= 1;
                'hA : wrt <= 1;
                'hB : wrt <= 1;
                'hC : wrt <= 1;
                'hD : wrt <= 1;
                'hE : wrt <= 1;
                'hF : wrt <= 1;
default : wrt <= 1;

        endcase
        //send couple of signals to RS as well as GPR
        stall <= 1'b0;
        opcod <= op;
        src1 <= s1;
        src2 <= s2;
        dstn <= dst;
        end

else                                    //if RS not free

        begin
                //hold all signals
                tag <= tag;
                wrt <= wrt;
                stall <= 1'b1;
                opcod <= opcod;
                src1 <= src1;
```

```
                src2 <= src2;
                dstn <= dstn;
            end

        end                                     //end of operation MUL

        default: begin

                stall  <= 1'b1;                 //if no RS free,
stall
                wrt    <= 1'b0;
                tag        <= 0;
                opcod <= opcod;
                src1 <= src1;
                src2 <= src2;
                dstn <= dstn;
            end

        endcase

    end
    end
endmodule
```

Register File:

```
`timescale 1ns / 10ps
module Reg_File(
   input clk,
   input rst,
   input [7:0]cmn_addrs,
   input [7:0]cmn_data,
```

```
input write,
input [7:0] dest,
input [7:0] tag,                    //RS tag to be assigned to GPR address
        input [7:0] S1,                     //to be used by MUX stage
        input [7:0] S2,
        input [3:0] opcode,                 //to be used in RS stage

        //entire packet for instruction
        output reg [7:0] tag_1,             //      pass to MUX
        output reg [7:0] src1,              //to be used by MUX stage
        output reg [7:0] src2,
        output reg [3:0] opcd,              //to be used in RS stage

output reg [7:0] R0_data,           //16 GPR's values
output reg [7:0] R1_data,
output reg [7:0] R2_data,
output reg [7:0] R3_data,
output reg [7:0] R4_data,
output reg [7:0] R5_data,
output reg [7:0] R6_data,
output reg [7:0] R7_data,
output reg [7:0] R8_data,
output reg [7:0] R9_data,
output reg [7:0] R10_data,
output reg [7:0] R11_data,
output reg [7:0] R12_data,
output reg [7:0] R13_data,
output reg [7:0] R14_data,
output reg [7:0] R15_data,
```

```
        output reg [7:0] R0_valid, //corresponding valid tags
output reg [7:0] R1_valid,
output reg [7:0] R2_valid,
output reg [7:0] R3_valid,
output reg [7:0] R4_valid,
output reg [7:0] R5_valid,
output reg [7:0] R6_valid,
output reg [7:0] R7_valid,
output reg [7:0] R8_valid,
output reg [7:0] R9_valid,
output reg [7:0] R10_valid,
output reg [7:0] R11_valid,
output reg [7:0] R12_valid,
output reg [7:0] R13_valid,
output reg [7:0] R14_valid,
output reg [7:0] R15_valid,

        output reg [7:0] R0_addrs, //GPR addresses
output reg [7:0] R1_addrs,
output reg [7:0] R2_addrs,
output reg [7:0] R3_addrs,
output reg [7:0] R4_addrs,
output reg [7:0] R5_addrs,
output reg [7:0] R6_addrs,
output reg [7:0] R7_addrs,
output reg [7:0] R8_addrs,
output reg [7:0] R9_addrs,
output reg [7:0] R10_addrs,
```

```
    output reg [7:0] R11_addrs,
    output reg [7:0] R12_addrs,
    output reg [7:0] R13_addrs,
    output reg [7:0] R14_addrs,
    output reg [7:0] R15_addrs
    );

        always@(posedge clk)
            begin
                if(rst == 1'b1 )                    //reset state
      begin
                    R0_data  <= 8'b0000_0000;
                    R0_valid <= 1;                  //valid is high
                    R0_addrs  <= 0;
                end
                else if(write == 1 && dest == 8'b0000_0000)     //if R0 is dest get the RS
tag
                    begin
                        R0_data  <= 0;
                        R0_addrs  <= tag;
                        R0_valid <= 0;
                    end
                else if(cmn_addrs == R0_addrs)                  //if common data bus TAG
matches GPR tag
                        begin
                            R0_addrs <= 0;
                            R0_data  <= cmn_data;           //write the
common data
                            R0_valid <= 1;
                        end
```

```
                                        else
                    //don't change anything
                                                begin
                                                        R0_data  <= R0_data;
                                                        R0_valid <= R0_valid;
                                                        R0_addrs  <= R0_addrs;
                                                end
                            end



        always@(posedge clk)
                    begin
                            if(rst == 1'b1 )                    //reset state
        begin
                                    R1_data  <= 8'b0000_0001;
                                    R1_valid <= 1;                      //valid is high
                                    R1_addrs  <= 0;
                            end
                            else if(write == 1 && dest == 8'b0000_0001)          //if R0 is dest get the RS
tag
                                    begin
                                            R1_data  <= 0;
                                            R1_addrs  <= tag;
                                            R1_valid <= 0;
                                    end
                                    else if(cmn_addrs == R1_addrs)              //if common data bus TAG
matches GPR tag
                                            begin
                                                    R1_addrs <= 0;
```

```
                                        R1_data  <= cmn_data;                //write the
common data
                                        R1_valid <= 1;
                                    end
                                    else
                    //don't change anything
                                        begin
                                            R1_data  <= R1_data;
                                            R1_valid <= R1_valid;
                                            R1_addrs  <= R1_addrs;
                                        end
                            end


        always@(posedge clk)
                    begin
                            if(rst == 1'b1 )                    //reset state
        begin
                                R2_data  <= 8'b0000_0010;
                                R2_valid <= 1;                      //valid is high
                                R2_addrs  <= 0;
                            end
                            else if(write == 1 && dest == 8'b0000_0010)          //if R0 is dest get the RS
tag
                                begin
                                    R2_data  <= 0;
                                    R2_addrs  <= tag;
                                    R2_valid <= 0;
                                end
                                else if(cmn_addrs == R2_addrs)              //if common data bus TAG
matches GPR tag
```

```
                                        begin
                                                R2_addrs <= 0;
                                                R2_data  <= cmn_data;                   //write the
common data
                                                R2_valid <= 1;
                                        end
                                        else
                        //don't change anything
                                                begin
                                                        R2_data  <= R2_data;
                                                        R2_valid <= R2_valid;
                                                        R2_addrs  <= R2_addrs;
                                                end
                                end


        always@(posedge clk)
                        begin
                                if(rst == 1'b1 )                        //reset state
        begin
                                        R3_data  <= 8'b0000_0011;
                                        R3_valid <= 1;                          //valid is high
                                        R3_addrs  <= 0;
                                end
                                else if(write == 1 && dest == 8'b0000_0011)             //if R0 is dest get the RS
tag
                                        begin
                                                R3_data  <= 0;
                                                R3_addrs  <= tag;
                                                R3_valid <= 0;
                                        end
```

```
                else if(cmn_addrs == R3_addrs)          //if common data bus TAG
matches GPR tag
                        begin
                                R3_addrs <= 0;
                                R3_data  <= cmn_data;           //write the
common data
                                R3_valid <= 1;
                        end
                        else
                //don't change anything
                                begin
                                        R3_data  <= R3_data;
                                        R3_valid <= R3_valid;
                                        R3_addrs  <= R3_addrs;
                                end
                end


        always@(posedge clk)
                begin
                        if(rst == 1'b1 )                //reset state
      begin
                                R4_data  <= 8'b0000_0100;
                                R4_valid <= 1;                  //valid is high
                                R4_addrs  <= 0;
                        end
                        else if(write == 1 && dest == 8'b0000_0100)     //if R0 is dest get the RS
tag
                                begin
                                        R4_data  <= 0;
                                        R4_addrs  <= tag;
```

```
                R4_valid <= 0;
            end
            else if(cmn_addrs == R4_addrs)          //if common data bus TAG matches GPR tag
                begin
                    R4_addrs <= 0;
                    R4_data  <= cmn_data;           //write the common data
                    R4_valid <= 1;
                end
                else
        //don't change anything
                    begin
                        R4_data  <= R4_data;
                        R4_valid <= R4_valid;
                        R4_addrs  <= R4_addrs;
                    end
        end


    always@(posedge clk)
        begin
            if(rst == 1'b1 )                    //reset state
  begin
                R5_data  <= 8'b0000_0101;
                R5_valid <= 1;                  //valid is high
                R5_addrs  <= 0;
            end
            else if(write == 1 && dest == 8'b0000_0101)     //if R0 is dest get the RS tag
                begin
```

```
                R5_data  <= 0;
                R5_addrs  <= tag;
                R5_valid <= 0;
            end
            else if(cmn_addrs == R5_addrs)          //if common data bus TAG
matches GPR tag
                begin
                    R5_addrs <= 0;
                    R5_data  <= cmn_data;           //write the
common data
                    R5_valid <= 1;
                end
                else
        //don't change anything
                    begin
                        R5_data  <= R5_data;
                        R5_valid <= R5_valid;
                        R5_addrs  <= R5_addrs;
                    end
            end


    always@(posedge clk)
            begin
                if(rst == 1'b1 )                    //reset state
  begin
                    R6_data  <= 8'b0000_0110;
                    R6_valid <= 1;                  //valid is high
                    R6_addrs  <= 0;
                end
```

```
                else if(write == 1 && dest == 8'b0000_0110)          //if R0 is dest get the RS
tag
                        begin
                                R6_data  <= 0;
                                R6_addrs  <= tag;
                                R6_valid <= 0;
                        end
                        else if(cmn_addrs == R6_addrs)               //if common data bus TAG
matches GPR tag
                                begin
                                        R6_addrs <= 0;
                                        R6_data  <= cmn_data;                //write the
common data
                                        R6_valid <= 1;
                                end
                                else
                //don't change anything
                                        begin
                                                R6_data  <= R6_data;
                                                R6_valid <= R6_valid;
                                                R6_addrs  <= R6_addrs;
                                        end
                        end


        always@(posedge clk)
                begin
                        if(rst == 1'b1 )                        //reset state
    begin
                                R7_data  <= 8'b0000_0111;
                                R7_valid <= 1;                          //valid is high
```

```
                R7_addrs  <= 0;
            end
            else if(write == 1 && dest == 8'b0000_0111)          //if R0 is dest get the RS
tag
                begin
                    R7_data  <= 0;
                    R7_addrs  <= tag;
                    R7_valid <= 0;
                end
                else if(cmn_addrs == R7_addrs)                   //if common data bus TAG
matches GPR tag
                    begin
                        R7_addrs <= 0;
                        R7_data  <= cmn_data;                    //write the
common data
                        R7_valid <= 1;
                    end
                    else
            //don't change anything
                        begin
                            R7_data  <= R7_data;
                            R7_valid <= R7_valid;
                            R7_addrs  <= R7_addrs;
                        end
            end


    always@(posedge clk)
            begin
                if(rst == 1'b1 )                    //reset state
  begin
```

```
                R8_data  <= 8'b0000_1000;
                R8_valid <= 1;                  //valid is high
                R8_addrs  <= 0;
            end
            else if(write == 1 && dest == 8'b0000_1000)          //if R0 is dest get the RS
tag
                begin
                    R8_data  <= 0;
                    R8_addrs  <= tag;
                    R8_valid <= 0;
                end
                else if(cmn_addrs == R8_addrs)                   //if common data bus TAG
matches GPR tag
                    begin
                        R8_addrs <= 0;
                        R8_data  <= cmn_data;                    //write the
common data
                        R8_valid <= 1;
                    end
                    else
        //don't change anything
                        begin
                            R8_data  <= R8_data;
                            R8_valid <= R8_valid;
                            R8_addrs  <= R8_addrs;
                        end
            end


    always@(posedge clk)
        begin
```

```
                if(rst == 1'b1 )                    //reset state
        begin
                        R9_data  <= 8'b0000_1001;
                        R9_valid <= 1;                  //valid is high
                        R9_addrs  <= 0;
                end
                else if(write == 1 && dest == 8'b0000_1001)     //if R0 is dest get the RS
tag
                        begin
                                R9_data  <= 0;
                                R9_addrs  <= tag;
                                R9_valid <= 0;
                        end
                        else if(cmn_addrs == R9_addrs)          //if common data bus TAG
matches GPR tag
                                begin
                                        R9_addrs <= 0;
                                        R9_data  <= cmn_data;           //write the
common data
                                        R9_valid <= 1;
                                end
                                else
                        //don't change anything
                                        begin
                                                R9_data  <= R9_data;
                                                R9_valid <= R9_valid;
                                                R9_addrs  <= R9_addrs;
                                        end
                end
```

```
always@(posedge clk)
        begin
            if(rst == 1'b1 )                    //reset state
    begin
                R10_data  <= 8'b0000_1010;
                R10_valid <= 1;                 //valid is high
                R10_addrs  <= 0;
            end
            else if(write == 1 && dest == 8'b0000_1010)     //if R0 is dest get the RS
tag
                begin
                    R10_data  <= 0;
                    R10_addrs  <= tag;
                    R10_valid <= 0;
                end
                else if(cmn_addrs == R10_addrs)     //if common data bus TAG
matches GPR tag
                    begin
                        R10_addrs <= 0;
                        R10_data  <= cmn_data;      //write the
common data
                        R10_valid <= 1;
                    end
                    else
        //don't change anything
                        begin
                            R10_data  <= R10_data;
                            R10_valid <= R10_valid;
                            R10_addrs  <= R10_addrs;
                        end
            end
```

```
always@(posedge clk)
        begin
            if(rst == 1'b1 )                    //reset state
    begin
                R11_data  <= 8'b0000_1011;
                R11_valid <= 1;                 //valid is high
                R11_addrs  <= 0;
            end
            else if(write == 1 && dest == 8'b0000_1011)     //if R0 is dest get the RS
tag
                begin
                    R11_data  <= 0;
                    R11_addrs  <= tag;
                    R11_valid <= 0;
                end
                else if(cmn_addrs == R11_addrs)     //if common data bus TAG
matches GPR tag
                    begin
                        R11_addrs <= 0;
                        R11_data  <= cmn_data;      //write the
common data
                        R11_valid <= 1;
                    end
                    else
        //don't change anything
                        begin
                            R11_data  <= R11_data;
                            R11_valid <= R11_valid;
                            R11_addrs  <= R11_addrs;
```

```
                                        end
                end


        always@(posedge clk)
                begin
                        if(rst == 1'b1 )                        //reset state
    begin
                                R12_data  <= 8'b0000_1100;
                                R12_valid <= 1;                 //valid is high
                                R12_addrs  <= 0;
                        end
                        else if(write == 1 && dest == 8'b0000_1100)          //if R0 is dest get the RS
tag
                                begin
                                        R12_data  <= 0;
                                        R12_addrs <= tag;
                                        R12_valid <= 0;
                                end
                                else if(cmn_addrs == R12_addrs)              //if common data bus TAG
matches GPR tag
                                        begin
                                                R12_addrs <= 0;
                                                R12_data  <= cmn_data;               //write the
common data
                                                R12_valid <= 1;
                                        end
                                        else
                //don't change anything
                                                begin
                                                        R12_data  <= R12_data;
```

```
                                        R12_valid <= R12_valid;
                                        R12_addrs  <= R12_addrs;
                                    end
                end


        always@(posedge clk)
                begin
                        if(rst == 1'b1 )                    //reset state
        begin
                                R13_data  <= 8'b0000_1101;
                                R13_valid <= 1;                     //valid is high
                                R13_addrs  <= 0;
                        end
                        else if(write == 1 && dest == 8'b0000_1101)         //if R0 is dest get the RS
tag
                                begin
                                        R13_data  <= 0;
                                        R13_addrs  <= tag;
                                        R13_valid <= 0;
                                end
                                else if(cmn_addrs == R13_addrs)             //if common data bus TAG
matches GPR tag
                                        begin
                                                R13_addrs <= 0;
                                                R13_data  <= cmn_data;              //write the
common data
                                                R13_valid <= 1;
                                        end
                                        else
                //don't change anything
```

```
                                        begin
                                                R13_data  <= R13_data;
                                                R13_valid <= R13_valid;
                                                R13_addrs  <= R13_addrs;
                                        end
                        end


        always@(posedge clk)
                begin
                        if(rst == 1'b1 )                        //reset state
        begin
                                R14_data  <= 8'b0000_1110;
                                R14_valid <= 1;                         //valid is high
                                R14_addrs  <= 0;
                        end
                        else if(write == 1 && dest == 8'b0000_1110)             //if R0 is dest get the RS
tag
                                begin
                                        R14_data  <= 0;
                                        R14_addrs  <= tag;
                                        R14_valid <= 0;
                                end
                                else if(cmn_addrs == R14_addrs)                 //if common data bus TAG
matches GPR tag
                                        begin
                                                R14_addrs <= 0;
                                                R14_data  <= cmn_data;                  //write the
common data
                                                R14_valid <= 1;
                                        end
```

```
                                        else
                //don't change anything
                                                begin
                                                        R14_data  <= R14_data;
                                                        R14_valid <= R14_valid;
                                                        R14_addrs  <= R14_addrs;
                                                end
                        end


        always@(posedge clk)
                begin
                        if(rst == 1'b1 )                        //reset state
    begin
                                R15_data  <= 8'b0000_1111;
                                R15_valid <= 1;                         //valid is high
                                R15_addrs  <= 0;
                        end
                        else if(write == 1 && dest == 8'b0000_1111)          //if R0 is dest get the RS
tag
                                begin
                                        R15_data  <= 0;
                                        R15_addrs  <= tag;
                                        R15_valid <= 0;
                                end
                                else if(cmn_addrs == R15_addrs)              //if common data bus TAG
matches GPR tag
                                        begin
                                                R15_addrs <= 0;
                                                R15_data  <= cmn_data;               //write the
common data
```

```
                                        R15_valid <= 1;
                                end
                                else
                //don't change anything
                                        begin
                                                R15_data  <= R15_data;
                                                R15_valid <= R15_valid;
                                                R15_addrs  <= R15_addrs;
                                        end
                        end

                always @ (posedge clk)                                  //to be passed to MUX
stage
                        begin

                                src1    <= S1;
                                src2    <=      S2;
                                opcd    <=      opcode;
                                tag_1 <= tag;



                        end

endmodule
```

Multiplexer:

```
`timescale 1ns / 10ps
module MUX(
   input clk,
        input rst,
```

```
        input S1,
        input S2,
        input [3:0] opcode,                    //to be used in RS stage
    input [7:0] tag_2,                         //coming from REG file to decide to which RS signals packet
should be sent
        input valid_R0,
    input valid_R1,
    input valid_R2,
    input valid_R3,
    input valid_R4,
    input valid_R5,
    input valid_R6,
    input valid_R7,
    input valid_R8,
    input valid_R9,
    input valid_R10,
    input valid_R11,
    input valid_R12,
    input valid_R13,
    input valid_R14,
    input valid_R15,

        input [7:0] data_R0,
        input [7:0] data_R1,
        input [7:0] data_R2,
        input [7:0] data_R3,
        input [7:0] data_R4,
        input [7:0] data_R5,
        input [7:0] data_R6,
        input [7:0] data_R7,
```

```
input [7:0] data_R8,
input [7:0] data_R9,
input [7:0] data_R10,
input [7:0] data_R11,
input [7:0] data_R12,
input [7:0] data_R13,
input [7:0] data_R14,
input [7:0] data_R15,

input [7:0] addrs_R0,
input [7:0] addrs_R1,
input [7:0] addrs_R2,
input [7:0] addrs_R3,
input [7:0] addrs_R4,
input [7:0] addrs_R5,
input [7:0] addrs_R6,
input [7:0] addrs_R7,
input [7:0] addrs_R8,
input [7:0] addrs_R9,
input [7:0] addrs_R10,
input [7:0] addrs_R11,
input [7:0] addrs_R12,
input [7:0] addrs_R13,
input [7:0] addrs_R14,
input [7:0] addrs_R15,

output reg [3:0] opcod,

// signals ccorrespoding to RS1
```

```
        output reg W1_1,
        output reg      W2_1,
        output reg      VAD1_1,
        output reg      VAD2_1,
        output reg      [7:0] DA1_1,
        output reg [7:0] DA2_1,

        // signals ccorrespoding to RS2
        output reg      W1_2,
        output reg W2_2,
        output reg VAD1_2,
        output reg      VAD2_2,
    output reg [7:0] DA1_2,
        output reg      [7:0] DA2_2,

        // signals ccorrespoding to RS3
        output reg      W1_3,
        output reg W2_3,
    output reg VAD1_3,
        output reg      VAD2_3,
        output reg      [7:0] DA1_3,
        output reg [7:0] DA2_3

    );

        reg wr1;                                    //signal indicating S1 is not ready, assign RS
tag
        reg wr2;                                    //signal indicating S2 is not ready, assign RS
tag
        reg valid_addrs1;           //imitates valid flag for S1 register
```

```
reg valid_addrs2;                   //imitates valid flag for S2 register
reg [7:0] write_data1;              //data/ RS tag passed to appropriate RS for S1
reg [7:0] write_data2;                      //data/ RS tag passed to appropriate RS for S2

always @ (posedge clk)
        begin
                case(S1)

                0:      begin
                        if(valid_R0 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R0;         //read data and
pass the valid value
                                end
                        else
                                begin
                                        wr1 <= 1'b1;
        //Reg value not valid
                                        valid_addrs1 <= 1'b0;
                                        write_data1 <= addrs_R0; //pass the tag to RS
operands
                                end
                        end

                1:      begin
                        if(valid_R1 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr1 <= 1'b0;
```

```
                        valid_addrs1 <= 1'b1;
                        write_data1 <= data_R1;          //read data and
pass the valid value
                    end
                else
                    begin
                        wr1 <= 1'b1;
        //Reg value not valid
                        valid_addrs1 <= 1'b0;
                        write_data1 <= addrs_R1; //pass the tag to RS
operands
                    end
                end

            2:  begin
                if(valid_R2 == 1'b1)          //if Reg value is valid, don't assign
any tag to RS operands
                    begin
                        wr1 <= 1'b0;
                        valid_addrs1 <= 1'b1;
                        write_data1 <= data_R2;          //read data and
pass the valid value
                    end
                else
                    begin
                        wr1 <= 1'b1;
        //Reg value not valid
                        valid_addrs1 <= 1'b0;
                        write_data1 <= addrs_R2; //pass the tag to RS
operands
                    end
                end
```

```
                3:      begin
                        if(valid_R3 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R3;         //read data and
pass the valid value
                                end
                        else
                                begin
                                        wr1 <= 1'b1;
        //Reg value not valid
                                        valid_addrs1 <= 1'b0;
                                        write_data1 <= addrs_R3; //pass the tag to RS
operands
                                end
                        end

                4:      begin
                        if(valid_R4 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R4;         //read data and
pass the valid value
                                end
                        else
                                begin
                                        wr1 <= 1'b1;
        //Reg value not valid
                                        valid_addrs1 <= 1'b0;
```

```
                write_data1 <= addrs_R4; //pass the tag to RS
operands
            end
        end

    5:  begin
        if(valid_R5 == 1'b1)        //if Reg value is valid, don't assign
any tag to RS operands
            begin
                wr1 <= 1'b0;
                valid_addrs1 <= 1'b1;
                write_data1 <= data_R5;        //read data and
pass the valid value
            end
        else
            begin
                wr1 <= 1'b1;
    //Reg value not valid
                valid_addrs1 <= 1'b0;
                write_data1 <= addrs_R5; //pass the tag to RS
operands
            end
        end

    6:  begin
        if(valid_R6 == 1'b1)        //if Reg value is valid, don't assign
any tag to RS operands
            begin
                wr1 <= 1'b0;
                valid_addrs1 <= 1'b1;
                write_data1 <= data_R6;        //read data and
pass the valid value
            end
```

```
                else
                        begin
                                wr1 <= 1'b1;
        //Reg value not valid
                                valid_addrs1 <= 1'b0;
                                write_data1 <= addrs_R6; //pass the tag to RS
operands
                        end
                end

                7:      begin
                if(valid_R7 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                        begin
                                wr1 <= 1'b0;
                                valid_addrs1 <= 1'b1;
                                write_data1 <= data_R7;         //read data and
pass the valid value
                        end
                else
                        begin
                                wr1 <= 1'b1;
        //Reg value not valid
                                valid_addrs1 <= 1'b0;
                                write_data1 <= addrs_R7; //pass the tag to RS
operands
                        end
                end

                8:      begin
                if(valid_R8 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                        begin
```

```
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R8;          //read data and
pass the valid value
                                    end
                                else
                                    begin
                                        wr1 <= 1'b1;
        //Reg value not valid
                                        valid_addrs1 <= 1'b0;
                                        write_data1 <= addrs_R8; //pass the tag to RS
operands
                                    end
                                end

                            9:  begin
                                if(valid_R9 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                                    begin
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R9;          //read data and
pass the valid value
                                    end
                                else
                                    begin
                                        wr1 <= 1'b1;
        //Reg value not valid
                                        valid_addrs1 <= 1'b0;
                                        write_data1 <= addrs_R9; //pass the tag to RS
operands
                                    end
                                end
```

```
                4'hA:   begin
                        if(valid_R10 == 1'b1)           //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R10;        //read data and
pass the valid value
                                end
                        else
                                begin
                                        wr1 <= 1'b1;
        //Reg value not valid
                                        valid_addrs1 <= 1'b0;
                                        write_data1 <= addrs_R10;       //pass the tag to
RS operands
                                end
                        end

                4'hB:   begin
                        if(valid_R11 == 1'b1)           //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R11;        //read data and
pass the valid value
                                end
                        else
                                begin
                                        wr1 <= 1'b1;
        //Reg value not valid
```

```
                                        valid_addrs1 <= 1'b0;
                                        write_data1 <= addrs_R11;          //pass the tag to
RS operands
                                end
                        end

                4'hC:   begin
                        if(valid_R12 == 1'b1)           //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R12;           //read data and
pass the valid value
                                end
                        else
                                begin
                                        wr1 <= 1'b1;
        //Reg value not valid
                                        valid_addrs1 <= 1'b0;
                                        write_data1 <= addrs_R12;          //pass the tag to
RS operands
                                end
                        end

                4'hD:   begin
                        if(valid_R13 == 1'b1)           //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr1 <= 1'b0;
                                        valid_addrs1 <= 1'b1;
                                        write_data1 <= data_R13; //read data and pass the
valid value
```

```
                end
            else
                begin
                    wr1 <= 1'b1;
        //Reg value not valid
                    valid_addrs1 <= 1'b0;
                    write_data1 <= addrs_R13;          //pass the tag to
RS operands
                end
            end

        4'hE:   begin
            if(valid_R14 == 1'b1)              //if Reg value is valid, don't assign
any tag to RS operands
                begin
                    wr1 <= 1'b0;
                    valid_addrs1 <= 1'b1;
                    write_data1 <= data_R14;           //read data and
pass the valid value
                end
            else
                begin
                    wr1 <= 1'b1;
        //Reg value not valid
                    valid_addrs1 <= 1'b0;
                    write_data1 <= addrs_R14;          //pass the tag to
RS operands
                end
            end

        4'hF:   begin
            if(valid_R15 == 1'b1)              //if Reg value is valid, don't assign
any tag to RS operands
```

```
                                        begin
                                                wr1 <= 1'b0;
                                                valid_addrs1 <= 1'b1;
                                                write_data1 <= data_R15;        //read data and
pass the valid value
                                        end
                                else
                                        begin
                                                wr1 <= 1'b1;
        //Reg value not valid
                                                valid_addrs1 <= 1'b0;
                                                write_data1 <= addrs_R15;       //pass the tag to
RS operands
                                        end
                                end


                                endcase
                        end



        always @ (posedge clk)
                begin
                        case(S2)

                        0:      begin
                                if(valid_R0 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                                        begin
                                                wr2 <= 1'b0;
                                                valid_addrs2 <= 1'b1;
                                                write_data2 <= data_R0;         //read data and
pass the valid value
```

```
                end
            else
                begin
                    wr2 <= 1'b1;
        //Reg value not valid
                    valid_addrs2 <= 1'b0;
                    write_data2 <= addrs_R0; //pass the tag to RS
operands
                end
            end

        1:  begin
            if(valid_R1 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                begin
                    wr2 <= 1'b0;
                    valid_addrs2 <= 1'b1;
                    write_data2 <= data_R1;         //read data and
pass the valid value
                end
            else
                begin
                    wr2 <= 1'b1;
        //Reg value not valid
                    valid_addrs2 <= 1'b0;
                    write_data2 <= addrs_R1; //pass the tag to RS
operands
                end
            end

        2:  begin
            if(valid_R2 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
```

```
                begin
                    wr2 <= 1'b0;
                    valid_addrs2 <= 1'b1;
                    write_data2 <= data_R2;          //read data and
pass the valid value
                end
            else
                begin
                    wr2 <= 1'b1;
        //Reg value not valid
                    valid_addrs2 <= 1'b0;
                    write_data2 <= addrs_R2; //pass the tag to RS
operands
                end
            end

        3:  begin
            if(valid_R3 == 1'b1)             //if Reg value is valid, don't assign
any tag to RS operands
                begin
                    wr2 <= 1'b0;
                    valid_addrs2 <= 1'b1;
                    write_data2 <= data_R3;          //read data and
pass the valid value
                end
            else
                begin
                    wr2 <= 1'b1;
        //Reg value not valid
                    valid_addrs2 <= 1'b0;
                    write_data2 <= addrs_R3; //pass the tag to RS
operands
                end
```

```
                end

            4:  begin
                if(valid_R4 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                    begin
                        wr2 <= 1'b0;
                        valid_addrs2 <= 1'b1;
                        write_data2 <= data_R4;         //read data and
pass the valid value
                    end
                else
                    begin
                        wr2 <= 1'b1;
        //Reg value not valid
                        valid_addrs2 <= 1'b0;
                        write_data2 <= addrs_R4; //pass the tag to RS
operands
                    end
                end

            5:  begin
                if(valid_R5 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                    begin
                        wr2 <= 1'b0;
                        valid_addrs2 <= 1'b1;
                        write_data2 <= data_R5;         //read data and
pass the valid value
                    end
                else
                    begin
```

```
                                        wr2 <= 1'b1;
        //Reg value not valid
                                        valid_addrs2 <= 1'b0;
                                        write_data2 <= addrs_R5; //pass the tag to RS
operands
                                end
                        end

                6:      begin
                        if(valid_R6 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr2 <= 1'b0;
                                        valid_addrs2 <= 1'b1;
                                        write_data2 <= data_R6;         //read data and
pass the valid value
                                end
                        else
                                begin
                                        wr2 <= 1'b1;
        //Reg value not valid
                                        valid_addrs2 <= 1'b0;
                                        write_data2 <= addrs_R6; //pass the tag to RS
operands
                                end
                        end

                7:      begin
                        if(valid_R7 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                                begin
                                        wr2 <= 1'b0;
                                        valid_addrs2 <= 1'b1;
```

```
                write_data2 <= data_R7;          //read data and
pass the valid value
            end
        else
            begin
                wr2 <= 1'b1;
    //Reg value not valid
                valid_addrs2 <= 1'b0;
                write_data2 <= addrs_R7; //pass the tag to RS
operands
            end
        end

    8:  begin
        if(valid_R8 == 1'b1)          //if Reg value is valid, don't assign
any tag to RS operands
            begin
                wr2 <= 1'b0;
                valid_addrs2 <= 1'b1;
                write_data2 <= data_R8;          //read data and
pass the valid value
            end
        else
            begin
                wr2 <= 1'b1;
    //Reg value not valid
                valid_addrs2 <= 1'b0;
                write_data2 <= addrs_R8; //pass the tag to RS
operands
            end
        end

    9:  begin
```

```
                        if(valid_R9 == 1'b1)            //if Reg value is valid, don't assign
any tag to RS operands
                            begin
                                wr2 <= 1'b0;
                                valid_addrs2 <= 1'b1;
                                write_data2 <= data_R9;         //read data and
pass the valid value
                            end
                        else
                            begin
                                wr2 <= 1'b1;
        //Reg value not valid
                                valid_addrs2 <= 1'b0;
                                write_data2 <= addrs_R9; //pass the tag to RS
operands
                            end
                    end

                4'hA:   begin
                        if(valid_R10 == 1'b1)           //if Reg value is valid, don't assign
any tag to RS operands
                            begin
                                wr2 <= 1'b0;
                                valid_addrs2 <= 1'b1;
                                write_data2 <= data_R10;        //read data and
pass the valid value
                            end
                        else
                            begin
                                wr2 <= 1'b1;
        //Reg value not valid
                                valid_addrs2 <= 1'b0;
```

```
                                        write_data2 <= addrs_R10;           //pass the tag to
RS operands
                                    end
                                end

                        4'hB:   begin
                                if(valid_R11 == 1'b1)               //if Reg value is valid, don't assign
any tag to RS operands
                                    begin
                                        wr2 <= 1'b0;
                                        valid_addrs2 <= 1'b1;
                                        write_data2 <= data_R11;            //read data and
pass the valid value
                                    end
                                else
                                    begin
                                        wr2 <= 1'b1;
        //Reg value not valid
                                        valid_addrs2 <= 1'b0;
                                        write_data2 <= addrs_R11;           //pass the tag to
RS operands
                                    end
                                end

                        4'hC:   begin
                                if(valid_R12 == 1'b1)               //if Reg value is valid, don't assign
any tag to RS operands
                                    begin
                                        wr2 <= 1'b0;
                                        valid_addrs2 <= 1'b1;
                                        write_data2 <= data_R12;            //read data and
pass the valid value
                                    end
```

```
                else
                        begin
                                wr2 <= 1'b1;
        //Reg value not valid
                                valid_addrs2 <= 1'b0;
                                write_data2 <= addrs_R12;       //pass the tag to
RS operands
                        end
                end

                4'hD:   begin
                if(valid_R13 == 1'b1)           //if Reg value is valid, don't assign
any tag to RS operands
                        begin
                                wr2 <= 1'b0;
                                valid_addrs2 <= 1'b1;
                                write_data2 <= data_R13; //read data and pass the
valid value
                        end
                else
                        begin
                                wr2 <= 1'b1;
        //Reg value not valid
                                valid_addrs2 <= 1'b0;
                                write_data2 <= addrs_R13;       //pass the tag to
RS operands
                        end
                end

                4'hE:   begin
                if(valid_R14 == 1'b1)           //if Reg value is valid, don't assign
any tag to RS operands
                        begin
```

```
                                        wr2 <= 1'b0;
                                        valid_addrs2 <= 1'b1;
                                        write_data2 <= data_R14;        //read data and
pass the valid value
                                    end
                                else
                                    begin
                                        wr2 <= 1'b1;
        //Reg value not valid
                                        valid_addrs2 <= 1'b0;
                                        write_data2 <= addrs_R14;       //pass the tag to
RS operands
                                    end
                                end

                        4'hF:   begin
                                if(valid_R15 == 1'b1)           //if Reg value is valid, don't assign
any tag to RS operands
                                    begin
                                        wr2 <= 1'b0;
                                        valid_addrs2 <= 1'b1;
                                        write_data2 <= data_R15;        //read data and
pass the valid value
                                    end
                                else
                                    begin
                                        wr2 <= 1'b1;
        //Reg value not valid
                                        valid_addrs2 <= 1'b0;
                                        write_data2 <= addrs_R15;       //pass the tag to
RS operands
                                    end
                                end
```

```
                endcase
        end

always @ (posedge clk)                                  //pass to RS stage
        begin
                opcod <= opcode;
        end


always @ (clk)
        begin
                case(tag_2)

                8'hA1: begin
                                W1_1    = wr1;
                                W2_1    = wr2;
                                VAD1_1 = valid_addrs1;
                                VAD2_1 = valid_addrs2;
                                DA1_1 = write_data1;
                                DA2_1 = write_data2;
                        end


                //all six signals to be asserted

                8'hB1: begin
                                W1_2    = wr1;
                                W2_2    = wr2;
```

```
                VAD1_2 = valid_addrs1;
                VAD2_2 = valid_addrs2;
                DA1_2 = write_data1;
                DA2_2 = write_data2;
            end

        8'hC1: begin
                W1_3    = wr1;
                W2_3    = wr2;
                VAD1_3 = valid_addrs1;
                VAD2_3 = valid_addrs2;
                DA1_3 = write_data1;
                DA2_3 = write_data2;
            end

    default: begin                                  //hold all signals
                W1_1    = W1_1;
                W2_1    = W2_1;
                VAD1_1 = VAD1_1;
                VAD2_1 = VAD2_1;
                DA1_1 = DA1_1;
                DA2_1 = DA2_1;
                W1_2    = W1_2;
                W2_2    = W2_2;
                VAD1_2 = VAD1_2;
                VAD2_2 = VAD2_2;
                DA1_2 = DA1_2;
                DA2_2 = DA2_2;
```

```
                                        W1_3    = W1_3;
                                        W2_3    = W2_3;
                                        VAD1_3 = VAD1_3;
                                        VAD2_3 = VAD2_3;
                                        DA1_3 = DA1_3;
                                        DA2_3 = DA2_3;
                                    end
                                endcase
                            end

endmodule
```

Reservation Station 1:

```
`timescale 1ns / 10ps
module RS1_ADD(
   input clk,
   input rst,
   input [3:0] opcode,
   input [7:0] cmn_addr,
   input [7:0] cmn_data,
   input write1,
   input valid1,
   input write2,
   input valid2,
         input [7:0] data_addr1,
   input [7:0] data_addr2,
         output reg fg1,                    //shows the RS status empty/in use
   output reg [7:0] out_addr,
```

```
output reg [7:0] out_data
);

        reg   [7:0]  v_j;                  //value of S1
reg     v_j_valid;                         //valid bit corresponding to S1
        reg   [7:0]  Q_j;                  //RS tag indicating RS producing the data
reg   [7:0]  v_k;
        reg    v_k_valid;
        reg   [7:0]  Q_k;

        reg   [7:0]  out_addr_buf;
        reg   [7:0]  out_data_buf;


            always@ (posedge clk)
                begin
                    if(rst == 1)
                        begin
                        out_addr_buf <= 0;
                        out_data_buf <= 0;
                        end
                    else if(v_j_valid == 1 && v_k_valid == 1 && opcode == 1'h0)
                        begin
                        out_addr_buf <= 8'hA1;                    //tag of RS
                        out_data_buf <= v_j + v_k;
                        end
                    /*else if(v_j_valid == 1 && v_k_valid == 1 && opcode == 4'h0100)
                        begin
                        out_addr_buf <= 8'hA2;
```

```
                out_data_buf <= v_j - v_k;
                end
            */else
                begin
                out_addr_buf <= 0;
                out_data_buf <= 0;
                end
        end

    always@ (posedge clk)
        begin
            if(rst == 1)
                begin
                v_j_valid   <= 0;
                v_j         <= 0;
                Q_j         <= 0;
                fg1                         <= 1'b1;
    //RS is now empty
                end
            else if(write1 == 1)
                begin
                v_j_valid   <= 0;
                v_j         <= 0;
                Q_j         <= data_addr1;
                fg1                         <= 1'b0;                    //RS is
now in use
                end
            else if(valid1 == 1)
                begin
                v_j_valid   <= 1;
```

```
                v_j     <= data_addr1;
                Q_j     <= 0;
                fg1                     <= 1'b0;
                end
            else if(cmn_addr == Q_j)            //internal forwarding for EX -> EX
                begin
                v_j_valid   <= 1;
                v_j     <= cmn_data;
                Q_j     <= 0;
                fg1                     <= 1'b0;
                end
            else if(v_j_valid == 1 && v_k_valid == 1)
                begin
                v_j_valid   <= 0;
                fg1             <= 1'b1;            //result is written
to CDB and RS is freed

                end
            else
                begin
                v_j_valid   <= v_j_valid;
                v_j     <= v_j;
                Q_j     <= Q_j;
                fg1                     <= fg1;
                end
        end


    always@ (posedge clk)
        begin
```

```
if(rst == 1)
        begin
        v_k_valid   <= 0;
        v_k         <= 0;
        Q_k         <= 0;
        end
else if(write2 == 1)
        begin
        v_k_valid   <= 0;
        v_k         <= 0;
        Q_k         <= data_addr2;
        end
else if(valid2 == 1)
        begin
        v_k_valid   <= 1;
        v_k         <= data_addr2;
        Q_k         <= 0;
        end
else if(cmn_addr == Q_k)
        begin
        v_k_valid   <= 1;
        v_k         <= cmn_data;
        Q_k         <= 0;
        end
else if(v_j_valid == 1 && v_k_valid == 1)
        begin
        v_k_valid   <= 0;
        end
else
```

```
                                        begin
                                        v_k_valid   <= v_k_valid;
                                        v_k         <= v_k;
                                        Q_k         <= Q_k;
                                        end
                end



        always@ (posedge clk)
                begin
                        if(rst == 1)
                                begin
                                out_addr <= 0;
                                out_data <= 0;
                                end
                        else
                                begin
                                out_addr <= out_addr_buf;
                                out_data <= out_data_buf;
                        //      fg1             <= 1'b1;            //result is written to CDB
and RS is freed
                                end
                end



endmodule
```

Reservation Station 2:

```
`timescale 1ns / 10ps
module RS2_SUB(
    input clk,
    input rst,
    input [3:0] opcode,
    input [7:0] cmn_addr,
    input [7:0] cmn_data,
    input write1,
    input valid1,
    input write2,
    input valid2,
            input [7:0] data_addr1,
    input [7:0] data_addr2,
            output reg fg2,                         //shows the RS status empty/in use
    output reg [7:0] out_addr,
    output reg [7:0] out_data
    );

            reg    [7:0]   v_j;                     //value of S1
    reg     v_j_valid;                              //valid bit corresponding to S1
            reg    [7:0]   Q_j;                     //RS tag indicating RS producing the data
    reg    [7:0]   v_k;
            reg     v_k_valid;
            reg    [7:0]   Q_k;

            reg    [7:0]   out_addr_buf;
            reg    [7:0]   out_data_buf;

                    always@ (posedge clk)
```

```
        begin
                if(rst == 1)
                        begin
                        out_addr_buf <= 0;
                        out_data_buf <= 0;
                        end
                else if(v_j_valid == 1 && v_k_valid == 1 && opcode == 1'h0)
                        begin
                        out_addr_buf <= 8'hB1;                  //tag of RS
                        out_data_buf <= v_j - v_k;
                        end
                /*else if(v_j_valid == 1 && v_k_valid == 1 && opcode == 4'h0100)
                        begin
                        out_addr_buf <= 8'hA2;
                        out_data_buf <= v_j - v_k;
                        end
                */else
                        begin
                        out_addr_buf <= 0;
                        out_data_buf <= 0;
                        end
        end


always@ (posedge clk)
        begin
                if(rst == 1)
                        begin
                        v_j_valid   <= 0;
                        v_j         <= 0;
```

```
                Q_j     <= 0;
                fg2                     <= 1'b1;
        //RS is now empty
                end
        else if(write1 == 1)
                begin
                v_j_valid   <= 0;
                v_j     <= 0;
                Q_j     <= data_addr1;
                fg2                     <= 1'b0;                //RS is
now in use
                end
        else if(valid1 == 1)
                begin
                v_j_valid   <= 1;
                v_j     <= data_addr1;
                Q_j     <= 0;
                fg2                     <= 1'b0;
                end
        else if(cmn_addr == Q_j)        //internal forwarding for EX -> EX
                begin
                v_j_valid   <= 1;
                v_j     <= cmn_data;
                Q_j     <= 0;
                fg2                     <= 1'b0;
                end
        else if(v_j_valid == 1 && v_k_valid == 1)
                begin
                v_j_valid   <= 0;
                fg2             <= 1'b1;                //result is written
to CDB and RS is freed
```

```
                end
        else
                begin
                v_j_valid   <= v_j_valid;
                v_j         <= v_j;
                Q_j         <= Q_j;
                fg2                             <= fg2;
                end
    end


always@ (posedge clk)
        begin
                if(rst == 1)
                        begin
                        v_k_valid   <= 0;
                        v_k         <= 0;
                        Q_k         <= 0;
                        end
                else if(write2 == 1)
                        begin
                        v_k_valid   <= 0;
                        v_k         <= 0;
                        Q_k         <= data_addr2;
                        end
                else if(valid2 == 1)
                        begin
                        v_k_valid   <= 1;
```

```
                v_k         <= data_addr2;
                Q_k         <= 0;
                end
            else if(cmn_addr == Q_k)
                begin
                v_k_valid   <= 1;
                v_k         <= cmn_data;
                Q_k         <= 0;
                end
            else if(v_j_valid == 1 && v_k_valid == 1)
                begin
                v_k_valid   <= 0;
                end
            else
                begin
                v_k_valid   <= v_k_valid;
                v_k         <= v_k;
                Q_k         <= Q_k;
                end
        end


always@ (posedge clk)
        begin
            if(rst == 1)
                begin
                out_addr <= 0;
                out_data <= 0;
                end
```

```
                else
                        begin
                        out_addr <= out_addr_buf;
                        out_data <= out_data_buf;
                        //fg2           <= 1'b1;            //result is written to CDB
and RS is freed
                        end
                end

endmodule
```

Reservation Station 3:

```
`timescale 1ns / 10ps
module RS3_MUL(
   input clk,
   input rst,
   input [3:0] opcode,
   input [7:0] cmn_addr,
   input [7:0] cmn_data,
   input write1,
   input valid1,
   input write2,
   input valid2,
        input [7:0] data_addr1,
   input [7:0] data_addr2,
        output reg fg3,                 //shows the RS status empty/in use
   output reg [7:0] out_addr,
   output reg [7:0] out_data
   );
```

```
        reg    [7:0]   v_j;                    //value of S1
reg     v_j_valid;                             //valid bit corresponding to S1
        reg    [7:0]   Q_j;                    //RS tag indicating RS producing the data
reg    [7:0]   v_k;
        reg     v_k_valid;
        reg    [7:0]   Q_k;


        reg    [7:0]   out_addr_buf;
        reg    [7:0]   out_data_buf;


                always@ (posedge clk)
                        begin
                                if(rst == 1)
                                        begin
                                        out_addr_buf <= 0;
                                        out_data_buf <= 0;
                                        end
                                else if(v_j_valid == 1 && v_k_valid == 1 && opcode == 1'h0)
                                        begin
                                        out_addr_buf <= 8'hC1;                  //tag of RS
                                        out_data_buf <= v_j * v_k;
                                        end
                                /*else if(v_j_valid == 1 && v_k_valid == 1 && opcode == 4'h0100)
                                        begin
                                        out_addr_buf <= 8'hA2;
                                        out_data_buf <= v_j - v_k;
                                        end
                                */else
```

```
                begin
                out_addr_buf <= 0;
                out_data_buf <= 0;
                end
        end

    always@ (posedge clk)
        begin
            if(rst == 1)
                begin
                v_j_valid   <= 0;
                v_j         <= 0;
                Q_j         <= 0;
                fg3                 <= 1'b1;
    //RS is now empty
                end
            else if(write1 == 1)
                begin
                v_j_valid   <= 0;
                v_j         <= 0;
                Q_j         <= data_addr1;
                fg3                 <= 1'b0;            //RS is
now in use
                end
            else if(valid1 == 1)
                begin
                v_j_valid   <= 1;
                v_j         <= data_addr1;
                Q_j         <= 0;
                fg3                 <= 1'b0;
```

```
                end
            else if(cmn_addr == Q_j)            //internal forwarding for EX -> EX
                begin
                v_j_valid   <= 1;
                v_j         <= cmn_data;
                Q_j         <= 0;
                fg3                         <= 1'b0;
                end
            else if(v_j_valid == 1 && v_k_valid == 1)
                begin
                v_j_valid   <= 0;
                fg3                 <= 1'b1;                    //result is written
to CDB and RS is freed

                end
            else
                begin
                v_j_valid   <= v_j_valid;
                v_j         <= v_j;
                Q_j         <= Q_j;
                fg3                         <= fg3;
                end
        end


    always@ (posedge clk)
        begin
            if(rst == 1)
                begin
                v_k_valid   <= 0;
```

```
            v_k         <= 0;
            Q_k         <= 0;
            end
        else if(write2 == 1)
            begin
            v_k_valid   <= 0;
            v_k         <= 0;
            Q_k         <= data_addr2;
            end
        else if(valid2 == 1)
            begin
            v_k_valid   <= 1;
            v_k         <= data_addr2;
            Q_k         <= 0;
            end
        else if(cmn_addr == Q_k)
            begin
            v_k_valid   <= 1;
            v_k         <= cmn_data;
            Q_k         <= 0;
            end
        else if(v_j_valid == 1 && v_k_valid == 1)
            begin
            v_k_valid   <= 0;
            end
        else
            begin
            v_k_valid   <= v_k_valid;
            v_k         <= v_k;
```

```
                                        Q_k         <= Q_k;
                                        end
                end


        always@ (posedge clk)
                begin
                        if(rst == 1)
                                begin
                                out_addr <= 0;
                                out_data <= 0;
                                end
                        else
                                begin
                                out_addr <= out_addr_buf;
                                out_data <= out_data_buf;
                                //fg3                   <= 1'b1;                        //result is written to CDB
and RS is freed
                                end
                end

endmodule
```

CDB Control:

```
`timescale 1ns / 10ps
module SORTING(
   input clk,
   input rst,
   input [7:0] tag1,
```

```
  input [7:0] tag2,
  input [7:0] tag3,
  input [7:0] out1,
  input [7:0] out2,
  input [7:0] out3,
        input [7:0] flag1,
        input [7:0] flag2,
        input [7:0] flag3,
  output reg [7:0] tag_out,
  output reg [7:0] final_out
  );

 always @ (posedge clk)
            begin
                if(rst == 1'b1)
                    begin
                        tag_out <= 0;
                        final_out <= 0;
                    end
                else if((tag3 == 8'hC1 && flag3 == 1'b1)&&(tag1 == 8'hA1 && flag1 == 1'b1)&&(tag2 == 8'hB1 && flag2 == 1'b1))
                        begin
                            tag_out <= tag3;
                            final_out <= out3;
                        end
                    else if((tag1 == 8'hA1 && flag1 == 1'b1)&&(tag2 == 8'hB1 && flag2 == 1'b1))
                            begin
                                tag_out <= tag1;
                                final_out <= out1;
```

```
                                                                    end
                                                            else if ((tag3 == 8'hC1 && flag3 ==
1'b1)&&(tag2 == 8'hB1 && flag2 == 1'b1))
                                                                    begin
                                                                    tag_out <= tag3;
                                                                    final_out <= out3;
                                                                    end
                                                                    else if ((tag3 == 8'hC1
&& flag3 == 1'b1)&&(tag1 == 8'hA1 && flag1 == 1'b1))
        begin
        tag_out <= tag3;
        final_out <= out3;
        end
                                                                                    else if
(tag3 == 8'hC1 && flag3 == 1'b1)
                begin
                        tag_out <= tag3;
                        final_out <= out3;
                end
        else if (tag2 == 8'hB1 && flag2 == 1'b1)
                        begin
                        tag_out <= tag2;
                        final_out <= out2;
                        end
```

```
                else if(tag1 == 8'hA1 && flag1 == 1'b1)
                                        begin
                                        tag_out <= tag1;
                                        final_out <= out1;
                                        end
                else
                        begin
                        tag_out <= tag_out;
                        final_out <= final_out;
                        end
                end
endmodule
```

Testbench:

```
module tb_Tomasulo;

        // Inputs
        reg clk;
        reg reset;

        // Outputs
        wire [7:0] data_out;

        // Instantiate the Unit Under Test (UUT)
```

```
TOP_LEVEL UUT0 (
        .clk(clk),
        .reset(reset),
        .data_out(data_out)
);



initial
        // Initialize Inputs
        begin
        clk = 1'b0;
        #5
        forever #5 clk = ~clk;
        end



initial

        begin

  #10 reset = 1'b1;



        // Wait 100 ns for global reset to finish
        #100 reset = 0;
```

```
        // Add stimulus here

        #1000 $finish;

        end

endmodule
```