Final Report – by Abdulsami & Khalid 05/15/2013

Register Renaming with a Reorder Buffer

Project Objective

In the labs, so far, we have implemented a SMIPS processor based on scoreboarding concept to handle data hazards. This concept can also enables the processor to handle out-of-order execution and avoid structural hazards to maximize the utilization of the functional units in the processor; however, the scorebording concept does not handle WAR hazards efficiently, where it introduces unnecessary stalls; also, it minimize potential utilization of the functional units by preventing instruction issues in the case of a WAW hazard. One solution to this problem is using Tomasulo's algorithm [1].

The whole idea behind Tomasulo's algorithm, shown in figure 1, is by having multiple Reservation Stations "RS" for each functional unit. The RS will serve as a mechanism to check for Structural Hazards and to do Register Renaming at the same time. This new addition to the design will let the processor work without worrying about WAR's and WAW's in the case of out-of-order execution. Also, the introduction of a Re-Order Buffer "ROB" along with Tomasulo's architecture, will handle roll-backs in the case of a misprediction. ROB will allow the processor to have an in-order issue → out-of-order execution → in-order commit which will keep track of the processor's state if the branch predictor mispredicted. In-line with this implementation, a Memory Operation Queue must be implemented as well. The Memory Operation Queue will act as a RS for the Memory operations to be performed. This can range from a simple queue to handle memory operations or can even be more complicated to act as a Memory Reorder Buffer, which will expedite the memory operations by searching the queue for possible store operations that could serve subsequent load operations based on the memory address. All the modules Tomasulo's architecture will be connected through a Common-Data Bus "CDB".

However, the implementation of Tomasulo's algorithm in modern system is infeasible. One reason for that would be the expense of implementing a common bus that has to be routed to all the modules in the system. Another reason would be the nature of distributed reservation stations will greatly limit the performance. This can occur if a ready instruction in the instruction queue of a certain type that has an available RS is being blocked by another instruction of a different type whose RS is full.

In this project, we will implement register renaming by splitting the logical registers (specified by the ISA) from the physical RegFile. We will have to maintain a mapping table along with a free-list of physical registers that are available for use. A centralized ROB will be used to handle the issue and commit stages of the pipeline. Although this implementation would be intended to boost the performance by adding the functionality of out-of-order execution, this project will implement these modules in a fully pipelined in-order processor to enable register renaming. We will the FPGA in this project to show the performance difference in comparison to the processor implemented in the lab and to have an idea about how much extra hardware we need to enable register renaming.

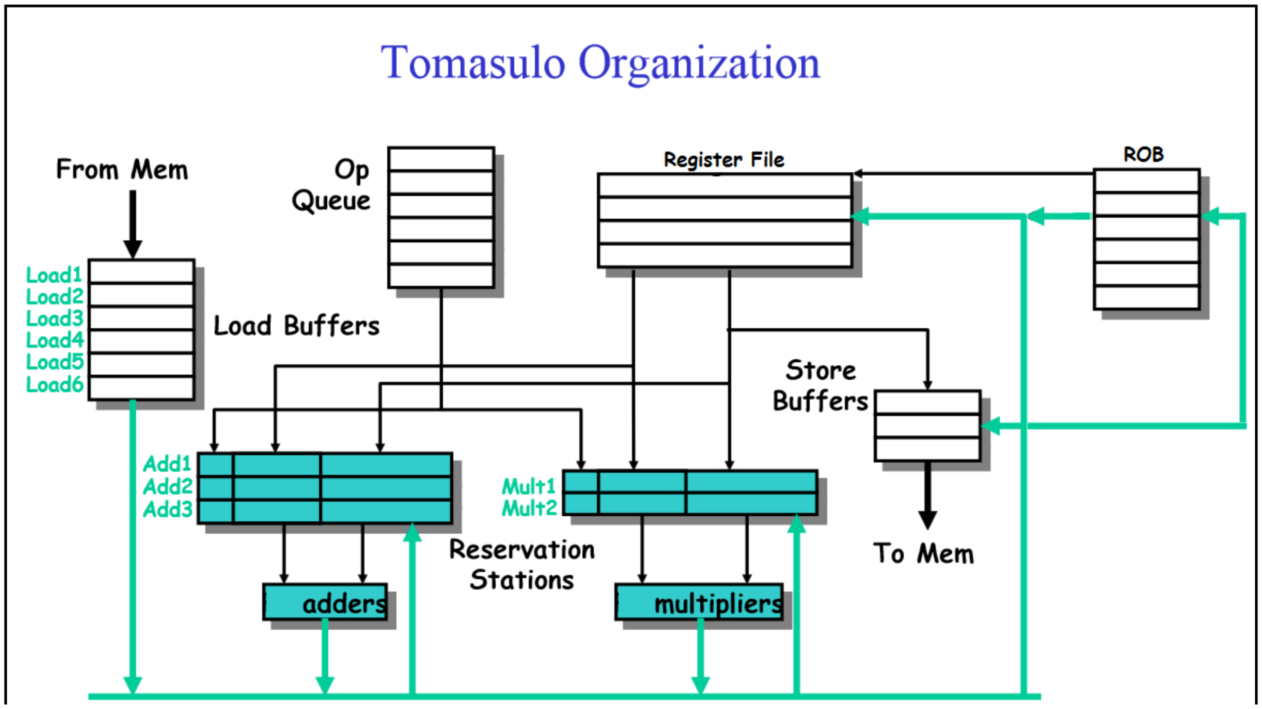

Figure 1: Tomasulo's Algorithm Architecture and Organization (reproduced from [2])

Background

Handling data and control hazards is an important aspect of processor design. While the compiler tried to reduce data dependencies between consecutive instructions, it faces a bunch of limitations. First of all the compiler is limited by the number of general purpose registers that are visible to the programmer which is a feature enforced by the underlying Instruction Set Architecture (ISA). Secondly, the complier cannot figure out dependencies that can be a result of data changing in runtime. To overcome the compiler limitation in scheduling these instructions, a dynamic scheduling mechanism should be implemented. Imagine we have the following piece of code that a compiler generated and has to run on a processor that uses scoreboard to handle data hazards:

```
LD    R1, 0(R5)
ADD   R1, R2, R3
```

Using scoarboard, the ADD instruction will stall the processor until the LD instruction writes its result back to register R1. WAW and WAR are called "false hazards" because if the compiler was smart enough to figure that out it would have used a different destination register and the pipeline would not have to stall until the LD instruction finishes.

In our implementation, we will use a register file (Physical Register File PRF) that is addressed by tags instead of the logical resister names enforced by the ISA. A new destination register will be assigned for any fetched instruction. Applying this method on the code above we get:

```
LD    PR32, 0(PR13)
ADD   PR54, PR22, PR33
```

PR32 and PR54 are free physical registers that got assigned to the destinations of the LD and ADD instructions respectively. This renaming methodology will solve both WAW and WAR hazards without the need of copying the values of the source registers to the reservation stations as in the case of Tomasulo's algorithm.

High-Level Design

The standard in-order processor design implemented in the lab will be used as a backbone for our design. For the standard processor, we have defined the following 5-stage pipeline implementation:

- **Fetch:** place a request for the I-Mem to read the instruction at current PC; set PC to predict PC.
- **Issue:** get the instruction from I-Mem and pass it to the Decode stage.
- **Decode:** decode the instruction, insert destination register to the scoreboard and read the operands from RegFile.
- **Execute:** executes the decoded instruction and place request for D-Mem when necessary.
- **Write Back:** get the data from D-Mem in case of a load instruction and writes results into the RegFile. Also removes the destination register from the scoreboard.

To implement register renaming with a ROB over this processor, we will now have the following stages for our pipeline:

- **Fetch:** place a request for the I-Mem to read the instruction at current PC; set PC to predicted PC.
- **Decode:** get the instruction from I-Mem; decodes the instruction.
- **Rename:** consults the mapping table to replace the logical registers with the physical register tag for source operands; assigns a destination tag from the free-list; inserts the Decoded Renamed Instruction into the Re-Order Buffer.
- **Issue:** chooses an instruction from the list of ready instructions in ROB to pass it to the execution unit.
- **Execute:** executes the decoded renamed instruction and place request for D-Mem in case of Load instructions.
- **Write Back:** get the data from D-Mem in case of a load instruction and writes results into the physical RegFile; update ready flags in the ROB.
- **Commit:** set the new Head for the ROB; update the free-list; if the instruction at head is a Store, place a request on D-Mem.

The description of each stage will be discussed thoroughly in the Microarchitectural Description section. The layout of the system is shown in figure 2.

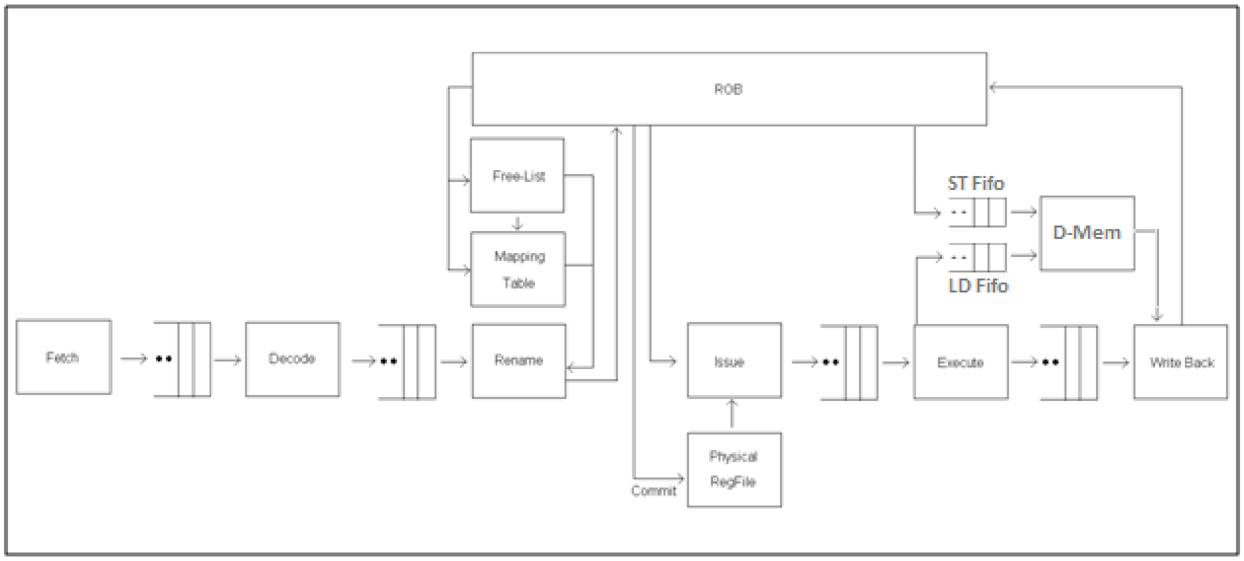

Figure 2: General Overview of the Proposed Architecture

Test-Plan

For testing our implementation, we will reuse the assembly codes and benchmarks given in the lab to insure correct functionality in simulation and when running on the FPGA. These benchmarks are as follows:

- Median
- Multiply
- Qsort
- Towers
- vvadd

Microarchitectural Description

Modules:

- **Predictor**
  - **Description**: The predictor contains logic that enables it to predict the next PC depending on the current PC; this logic will be adaptive in the sense that it monitors the execution and updates its logic with the feedback from execution whether it had mispredicted or predicted correctly.
  - **Interface**: The predictor module has two functionality that can be invoked by other modules
    - Predict: By giving it the current PC, the predictor returns the next predicted PC.
    - Train: By supplying the PC that the predictor used to generate a wrong predicted PC, the predictor trains its internal logic with the correct next PC and or it reinsure itself by the positive feedback in the case of correct prediction.
- **Fetch**
  - **Description**: This module would be responsible for fetching the instruction pointed at by the PC register and updating the PC register by the predicted PC address generated by the predictor module. In the case of a negative feedback from other stages where it says that the Fetch has mispredicted in one instruction, the Fetch will load the corrected PC and start functioning normally starting from the subsequent cycle changing the Epoch.
  - **Interface**: Fetch module will perform the following functionality that interacts with other modules
    - Push to Decode: The Fetch module will place a memory request and push needed information on the queue feeding the decode stage.

    Fetch module will have the following functionality that enables other modules to interact with it
    - Push Feedback: Any module, execution in specific, can push a feedback to the fetch module to inform it in the case that we have a mispredict so it can change the course of execution, specifically it would change the PC and the Epoch of the instructions to be fetched by next cycle.
- **Decode**
  - **Description**: The Decode module will be used in the Decode stage. This module will get the data from the Fetch stage, presented by the Fetch module, along with the instruction from the memory and decode this instruction. Finally, the Decode will push the decoded instruction to the Issue stage.

- **Interface**: Deocde will perform the following functionality that will affect other modules
  - Push to Rename: The Decode module will push the decoded instructions to queue feeding the Rename stage.

- **Free List**
  - **Description**: The Free List is a list that maintains free physical registers that can be used for mapping any logical register. It is a fast and efficient solution to find which physical registers can be used to be mapped to which logical registers.
  - **Interface**: The Free List has the following functionality that can be invoked by any other modules
    - Add Register: Add a given physical register to the free list.
    - Get Register: Returns a physical register that is free from the list and removes it making it not free anymore.
- **Mapping Table**
  - **Description**: The mapping Table is a table that maintains the correspondence between logical registers, used by the ISA, and physical registers, used by the pipeline.
  - **Interface**: The Mapping Table has the following functionality that can be used by other modules
    - Get Mapping For Logical Register: For a given logical register, return the currently mapped tag for the corresponding physical register.
    - Map a Logical Register to a Physical Register: Change the mapping for a given logical register to a given physical register by storing the corresponding tag. The Mapping Table would return the previous mapping to be used in other restoring cases as will be described later on.
- **Rename**
  - **Description**: The Rename module would represent the whole rename stage. The functionality performed by the Rename module would be to get one decoded instruction from the input queue. Then, Rename will consult the Free List to map the destination register to a new physical register. Finally, the Rename module would ask the Mapping Table to change the mapping of the destination register to the tag of the physical register returned by the Free List; Rename would also get the previously mapped physical register to be included in the output of the Rename module. Rename will also change the sources to the mapped tags returned by the Mapping Table. The Rename module would insert the Decoded Renamed Instruction (DRI) into the Reorder Buffer (ROB) to be executed. Also, the Rename would check for the Epoch of the Decoded Instruction and discard any decoded instructions that their Epoch do not match.
  - **Interface**: The Rename module will have the following functionality that will interact with other modules
    - Insert DRI into ROB: Insert the full constructed DRI into the ROB to be executed.

- **Reorder Buffer (ROB)**
  - **Description**: The reorder buffer is an essential element in the case of register renaming to handle precise exceptions and mispredicts. The ROB will hold every instruction in the Issue stage and beyond. It has two pointers, one to point to the head of the reorder buffer and the other to point to the tail. ROB will hold the decoded renamed instruction (DRI) as processed by the Rename stage. Each entry in the ROB will have Ready flags for each source register. Also, we have a Status field that indicates the state of the DRI whether it is:
    - Waiting: Waiting for its Operands to be Ready.
    - Ready: All operands are ready to be read and can be loaded into the functional unit.
    - Execute: The DRI is being executed.
    - Done: The DRI result has been written back to the physical register file or, in case of Stores, the request has been placed in the store buffer.

    In the case of mispredict, the ROB can perform a Replay to reconstruct the Mapping Table and the Free List to the way they were before a specific point in execution. The idea behind this is that for every DRI entry in the ROB, we will have the Logical Register and its current mapping along with its previous mapping. Therefore, we would sweep the ROB from the Tail up until the instruction at which the mispredict of the next PC was discovered. The Replay would basically reverse the mapping performed of every instruction processed between the Tail and the aforementioned instruction by replaying and reversing the mapping to the previously mapped physical register that is stored in the DRI. We also would place the currently mapped physical register of every DRI processed into the Free List to reconstruct the Free List. Also on mispredicts, the ROB will fill in the Redirect Fifo and feed it the new Epoch status.
  - **Interface**: The ROB will have the following functionality that can be used by other modules
    - Insert DRI: Insert a given DRI into the ROB checking for available space and modifying the Tail pointers.
    - Replay: This would stall the whole pipeline from the Issue stage and beyond given that the instruction at which the mispredict of the next PC was discovered is on the Head of the ROB.
    - Update: This would update the flag of the waiting instructions to Ready if the source they been waiting for has been committed.

The ROB has the following functionality that will interact with other modules:

  - Pop DRI: This will remove the head of the ROB signaling the successful committing of the DRI if its status is Commit, advance the Head pointer to the next DRI.

- **Issue**
  - **Description**: The Issue module will represent the Issue stage. This module is part of a group of modules that are controlled by the ROB. Issue would actually see the ready DRIs in the ROB and would Issue any ready DRIs into the Execute stage by queuing it on the queue feeding the Execute stage. In our specific implementation, the Issue would just look at the Head of the ROB and the subsequent entry to the Head entry in the case that the ROB head was in the WriteBack. This would guarantee for us an in-order execution as it is needed. Also, the Issue would read the operand needed for the taken DRI from the physical RegFile.
  - **Interface**: The Issue module has the following functionality that will interact with other modules
    - Push DRI to Execute: The Issue stage would consult a selection policy as described above to select next ready DRI to be pushed. It will queue the Issued DRI with the operands on the queue feeding the execution stage. The selection policy can later be enhanced to support out-of-order execution.
- **Execute**:
  - **Description**: This module represents the whole Execute stage. Basically, the Execute module would get the Issued DRIs with the operand values and check execute the instruction and push the Executed Instruction to the write back and the memory interface, in case of loads and stores.
  - **Interface**: The Execute module will have the following functionality that will interact with other modules
    - Push Executed Instruction to Write Back: Push any Executed Instruction ready from the functional units to the queue feeding the Write Back stage.
    - Place D-Mem Request: in case of Loads, the execute stage will enqeueu the Load Fifo feeding into D-Mem.

- **Write Back**:
    - **Description**: Write Back module will constitute the Write Back Stage. The Write Back module will process executed instructions and get any data from the memory interface if needed. Then, the write back will write the value back to the RegFile and change the status of the DRI in the ROB accordingly. The Write Back module will also update the Ready flags in the ROB to make it possible for the Issue module to read available DRIs in the ROB by the subsequent cycle.
    - **Interface**: The Write Back module will have the following functionality that will interact with other modules
        - Write Back into RegFile: Write the value taken from either the Executed Instruction or the Memory Interface into the RegFile.
        - Update ROB Status: Updates the Ready flags for corresponding source entries for every DRI and updates, thus, the status of the DRIs.
- **Memory Management Unit**:
    - **Description**: The memory management unit will be responsible for resolving memory disambiguation issues that can result because of mispredicts or out-of-order executed Loads and Stores. Also, because D-Mem has only one port that can accept requests, this unit should organize the requests issued to D-Mem in case two requests were placed in the same cycle. This is possible because the execute stage and commit stage can both fire at the same time were a request for Load and Store can be issued simultaneously. Therefore, these requests will not be issued to D-Mem directly. The requests has to go through a Fifo managed by this unit and can track which request to service before the other using timestamps. Each request will get a timestamp before entering the Fifo and therefore it can track these timestamps and make sure the loads and stores are being handled in the order they was enqeued into the Fifo. If a Load and Store had the same timestamp, Store will have the priority to be executed first.
    - **Interface**: The Memory Management Unit module will have the following functionality that will interact with other modules
        - Enqeue Load: This module would insert an incoming Load request tagged with an appropriate timestamp generated within the module.
        - Enqeue Store: This module would insert an incoming Store request tagged with an appropriate timestamp generated within the module.

Implementation Evaluation and Design Exploration

The Register Renaming processor was implemented and showed success in both simulation and on the FPGA. In the implementation phase, the most difficult task was rule scheduling. Since our ROB is a major component that reacts with Rename, Issue, WriteBack and Fetch, every ROB entry had to be declared as a 4-port Ehr. Although different rules will never access the same entry, bluespec cannot trust you in that and we had to accurately assign these ports in a specific order so that we do not get conflicting rules. Since we used the SMIPS processor from the labs as a backbone, we only had to write around 500 lines of code to incorporate register renaming where 200 lines of those were only for implementing the ROB module. All design parameters can be altered from the ProcTypes file such as ROB size, Intermediate Fifos size and physical RegFile size.

The first working design had to be modified to ensure correctness. These modifications included the check of mispredicts to be in the commit stage instead of execute stage. Our assumption was that during Replay mode we can deqeue the Fifos between issue&execute and execute&writeback. This assumption did not take into account the case where Replay can actually finish before these Fifos get empty, which is only valid if only our Replay method is used which is a cycle-by-cycle replay. Advanced Replay methods can take 3 cycles or less to clear the whole ROB but requires extra space to reconstruct the mapping table that fast. The second major modification was the way of handling memory instructions which required the addition of the Memory Management Unit (MMU) module. The current MMU, as explained in the Microarchitectural Description section, is simplified to a level that can greatly degrade performance. If a store exists in ROB, all subsequent loads have to wait until the preceding stores have been committed. This is not taken into account the case where the addresses are different which indicates that no conflict can occur if those two instructions went out-of-order.

For design exploration, the most important feature to explore was to actually enable the processor to handle out-of-order execution. Although this goal is not part of this project, the design we implemented is flexible enough to incorporate this feature with minimal modifications. The only major modification needed is to implement a selection policy that can choose any instruction out of those having their ready flag set and push that instruction to execution. Another important modification to go with that is to add more functional units in the execute stage such as integer units, multi-cycle multipliers, and multi-cycle dividers. This would also require the addition of a table holding the status of these functional units (busy, or ready). If we would like to see major performance difference between this processor, including out-of-order, and the in-order 5-stage processor, we have to add a Memory Reorder Buffer (MOB) in the MMU to efficiently handle memory operations.

Simulation and FPGA Results

The explorations we have done included different implementations of the in-order register renaming processor that shows the effect of different ROB size and RegFile size on performance (in terms of IPC), hardware utilization, maximum operating frequency, and critical path. All intermediate Fifos were set to size '2' and were Conflict Free Fifos. The results will be summarized below:

Basic 5-Stage Processor

Benchmark (IPC)	median	0.66	**Frequency**	Pre-Routing	115.929 MHz
	multiply	0.72		Post-Routing	51.738 MHz
	qsort	0.68	**Average MIPS**	84.86	
	towers	0.77			
	vvadd	0.83			
FPGA Operating Frequency	**50.00 MHz**	**FPGA Result Status**	**Working**	**Critical Path**	Levels of Logic = 39
Device Utilization	Slice Registers	14%	Slice LUTs		22%

Critical Path Description:

The report shows that the critical path starts from "scemi_dma_saved_length_2_1" to "scemi_csr_wr_xfer_count_31". These two signals run inside a module called *mkTLPArbiter*.

ROB Size = 2 RegFile Size = 32

Benchmark (IPC)	median	0.36	**Frequency**	Pre-Routing	90.628 MHz
	multiply	0.45		Post-Routing	50.679 MHz
	qsort	0.39	**Average MIPS**	37.24	
	towers	0.40			
	vvadd	0.45			
FPGA Operating Frequency	50.00 MHz	**FPGA Result Status**	Working	**Critical Path**	Levels of Logic = 16
Device Utilization	Slice Registers	18%	Slice LUTs		29%

Critical Path Description:
The report shows that the critical path starts from "scemi_dut_dut_dutIfc_m_dut/m/rob_robTable_0_rl_118_1" to "scemi_dut_dut_dutIfc_m_dut/m/i2e_tempData_rl_65". This critical path, unlike the following ones, occurs from the commit stage setting the "Ready" flag of an instruction up to pushing that instruction onto the Issue2Execute Fifo.

ROB Size = 4 RegFile Size = 64

Benchmark (IPC)	median	0.40	**Frequency**	Pre-Routing	64.889 MHz
	multiply	0.55		Post-Routing	29.612 MHz
	qsort	0.44	**Average MIPS**	31.62	
	towers	0.46			
	vvadd	0.58			
FPGA Operating Frequency	12.5 MHz	**FPGA Result Status**	Working	**Critical Path**	Levels of Logic = 21
Device Utilization	Slice Registers	21%	Slice LUTs		35%

Critical Path Description:
The report shows that the critical path starts from "scemi_dut_dut_dutIfc_m_dut/m/rob_robTable_3_rl_116" to "scemi_dut_dut_dutIfc_m_dut/m/freelist_list_deqEn_rl". This critical path, and the ones that follows, occurs in generating the WILL_FIRE signal of the rename rule (which uses method Insert in ROB) up to deqeueing a free physical register tag from the free list module.

ROB Size = 8 RegFile Size = 64

Benchmark (IPC)	median	0.37	**Frequency**	Pre-Routing	57.816 MHz
	multiply	0.53		Post-Routing	21.769 MHz
	qsort	0.40	**Average MIPS**	26.85	
	towers	0.44			
	vvadd	0.58			
FPGA Operating Frequency	12.5 MHz	**FPGA Result Status**	Working	**Critical Path**	Levels of Logic = 22
Device Utilization	Slice Registers	22%	Slice LUTs		41%

Critical Path Description:
The report shows that the critical path starts from "scemi_dut_dut_dutIfc_m_dut/m/rob_robTable_3_rl_116" to "scemi_dut_dut_dutIfc_m_dut/m/freelist_list_deqEn_rl". This critical path occurs in generating the WILL_FIRE signal of the rename rule (which uses method Insert in ROB) up to deqeueing a free physical register tag from the free list module.

ROB Size = 16 RegFile Size = 64

Benchmark (IPC)	median	0.34	**Frequency**	Pre-Routing	52.924 MHz
	multiply	0.50		Post-Routing	20.078 MHz
	qsort	0.37	**Average MIPS**	23.20	
	towers	0.40			
	vvadd	0.58			
FPGA Operating Frequency	12.5 MHz	**FPGA Result Status**	Not Stable	**Critical Path**	Levels of Logic = 25
Device Utilization	Slice Registers	25%	Slice LUTs		52%

Critical Path Description:
The report shows that the critical path starts from "scemi_dut_dut_dutIfc_m_dut/m/rob_robTable_3_rl_116" to "scemi_dut_dut_dutIfc_m_dut/m/freelist_list_deqEn_rl". This critical path occurs in generating the WILL_FIRE signal of the rename rule (which uses method Insert in ROB) up to deqeueing a free physical register tag from the free list module.

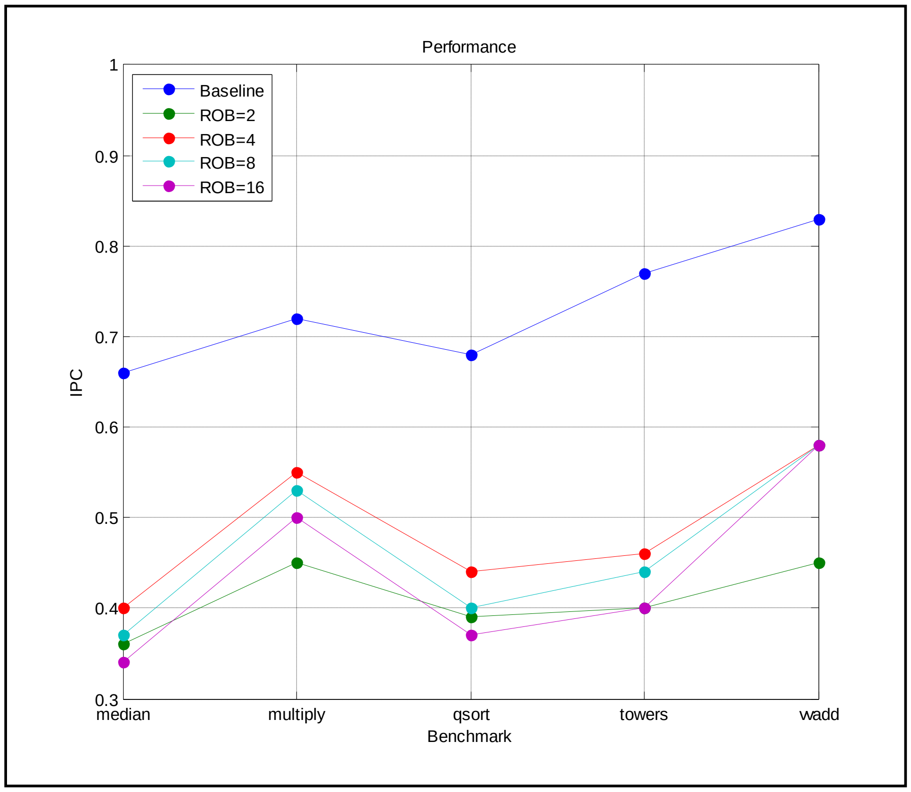

Figure 3: Performance for Each Benchmark across Different Implementations in IPC

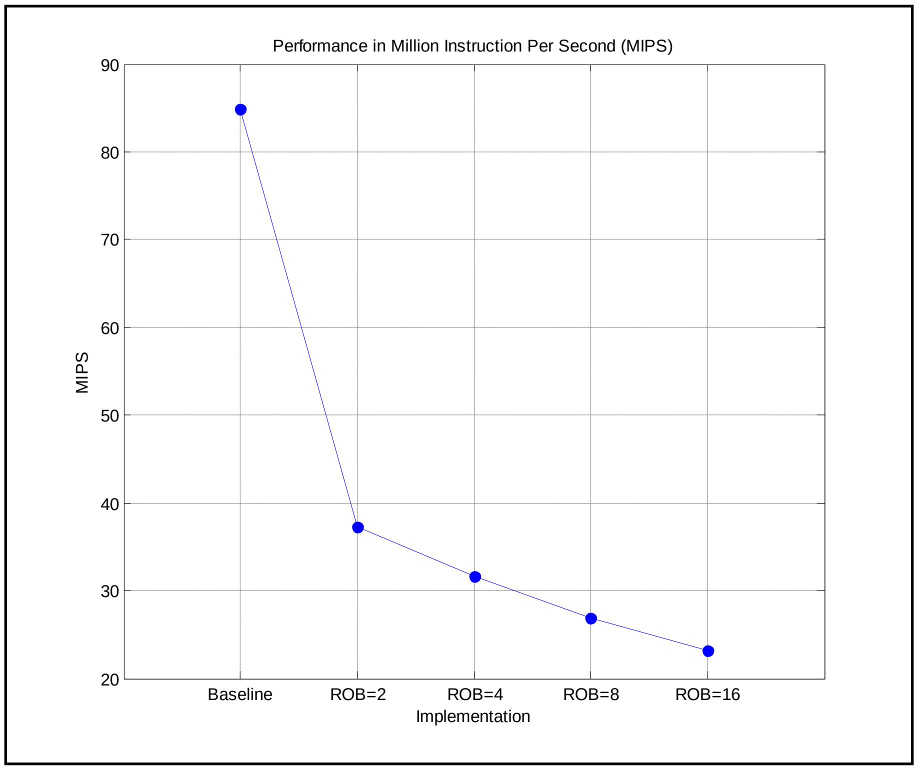

Figure 4: Performance for Each Benchmark across Different Implementations in MIPS

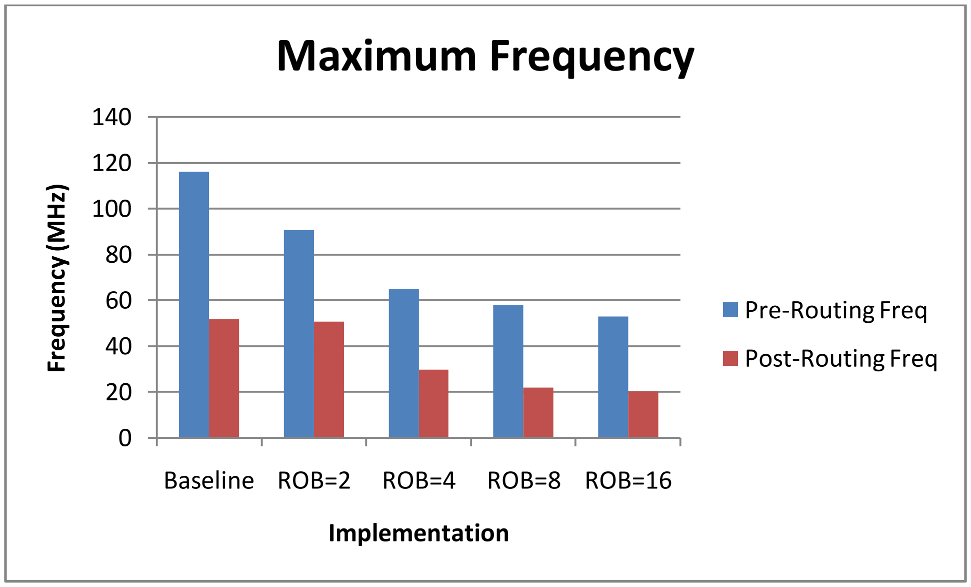

Figure 5: Maximum Pre- and Post-Routing Frequencies

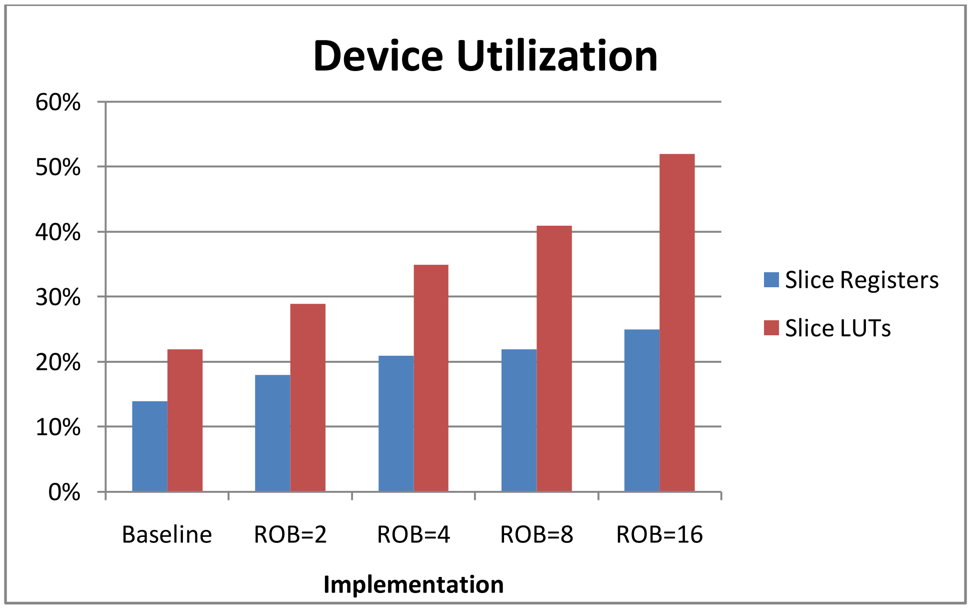

Figure 6: Device Utilization Chart

References

[1] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", Sept. 1965.

[2] Brooks, David., "Tomasulo's Algorithm", CS246: Advanced Computer Architecture – Harvard School of Engineering and Applied Sciences Course, Feb. 2013.

[3] Brooks, David., "MOB, P6 and R10K", CS246: Advanced Computer Architecture – Harvard School of Engineering and Applied Sciences Course, Feb. 2013.