Tomasulo Algorithm Simulator

Shashanka Bukka Vaidyam
Computer Science and Electrical Engineering
University of Maryland Baltimore County
Arbutus, Maryland
bukka1@umbc.edu

***Abstract*—The aim of this project is to understand and implement a simplified version of Tomasulo Algorithm as a web based application. The application was created to be utilized for simulation of the algorithm for educational purposes. This project deepens the understanding of the dynamic scheduling of pipelines and gives a more complete picture of the implementation details of the CPU cores in modern computers. Also analyzing the out of order execution of instructions, register renaming and the various advantages of the Tomasulo Algorithm.**

***Index Terms*—Dynamic scheduling, out of order execution, register renaming, hazards.**

I. Introduction

One of the major improvements done for increasing the effective speed is dynamic scheduling.

Dynamic scheduling is a method in which instructions to executeare determined by the hardware, as opposed to a statically scheduled machine, where the order of execution is determined by the compiler. Dynamic scheduling is similar to a data flow machine, in which the instructions are executed based on the availability of the source operands. But a real processor would only have limited amount of resources. Thus instructions are executed based on the availability of the source operands and also on the availability of requested functional units.

Dynamically scheduled machines take the advantage of parallelism which won't be visible during compile time. This also negates the recompilation of code for effeciency since the hardware takes care of scheduling.

The Tomasulo algorithm implements register renaming in hardware, to a degree that allows instructions to be executed out of order to improve pipeline throughput and efficiency. The major creations of Tomasulos algorithm include register renaming in hardware, reservation stations for all execution units, and a common data bus (CDB) on which computed values broadcast to all reservation stations that may need them. This algorithm was invented by Robert Tomasulo, and was first utilized in the IBM 360/91. Tomasulo's algorithm is different from the scoreboarding as it uses register renaming to eliminate output and anti-dependences. Such developments allowed for improved parallel execution of instructions which would otherwise stall under the use of scoreboarding or other earlier algorithms. In this project, the hardware structure of the designed Tomasulo simulator is a simplified version.

While out-of-order execution can lead to performance improvements, it can also lead to some problems. When implementing branch prediction, a wrong prediction would result in a flushed pipeline. However, when dealing with out-of-order execution, the problem arises where an instruction which should not have been executed has already written its results to the register file. One solution, of course, is to not allow instructions past a branch to execute until the branch has been resolved. But this mitigates the effects of branch prediction. A better solution, and the one used in modern processors, is to require that instructions be completed in order, in other words, instructions are issued in-order, executed out-of-order, then committed to the register file in-order. Thus, to anything outside of it, the microprocessor looks like it is executing instructions in-order; the first instruction that comes in is the first that is completed.

In-order completion is accomplished by placing a reorder buffer right before writing to the registers. The reorder buffer is an ordered buffer which holds the results of instructions waiting to be committed. The buffer is ordered in program order, thus the top of the buffer is the first instruction that should be committed. If an instruction completes, but earlier instructions are still executing, the result is stored in the reorder buffer until all earlier instructions have been committed.

When used in conjunction with dynamic branch prediction, the reorder buffer can be used to perform speculative execution. Going back to the example with the branch, if a prediction if found to be in error, the reorder buffer can be flushed starting at the branch since no instructions past the branch have permanently changed the state of the machine. In essence, with the reorder buffer we can now do loop unrolling completely in hardware. Note that the reorder buffer also allows for precise exception handling since instructions are completed in order. Thus, if an exception occurs, a flag is set, and when the instruction goes to commit, the flag is

detected and the exception is handled.

Many of today's processors implement some form of out-of-order execution mechanism. The IBM PowerPC 604, the MIPS R10k, and the HP PA-8000 all include an out-of-order execution unit with in-order completion. Even Intel's P6 line (Pentium Pro, Pentium II, Celeron) implements out-of-order execution.

II. Background

A. Hazards

Hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards cause reduction in the performance and negate the ideal speedup gained by pipelining. This may lead to incorrect computations of the values in the registers. Three common types of hazards are structural hazards, data hazards and control flow hazards.

a) Structural Hazards: In a pipelined machine, overlapped execution of instructions needs pipelining of functional units and resource duplication for allowing possible combinations of instructions in the pipeline. If some combination of instructions can't be allowed to get executed due to a resource conflict, the machine is said to have a structural hazard.
Structural hazards usually arise when the functional units is not fully pipelined or i f some resource has not been duplicated enough. For example, a machine may have only one reigster-file write port, but in some cases the pipeline might want to perform two writes in a clock cycle.

b) Control Hazards: Control Hazards arise from the pipelining of branches and other instruction which change the PC. In many instruction pipelining architectures, the processor wont know the outcome of the branch to understand which instruction to fetch into the pipeline.

c) Data Hazards: Data hazards arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline. These occur when instructions which have data dependence modify the data in different stages of a pipeline. Ignoring such hazards can reuslt in race conditions. There are three dependencies in between instructions which might cause data hazards such as read after write, write after read, write after write.

1) Read after write: A read after write hazard occurs when an instruction refers to a result that has not yet been calculated or retrieved. These can occur because even though an instruction is executed after a prior instruction, the prior instruction is executed only partly and isn't available.

$$I1 : \mathbf{R2} \leftarrow R1 + R3$$

$$I2 : R4 \leftarrow \mathbf{R2} + R3$$

For example, in the above instruction set **I1** is computing the value of R2 and this value is being used by **I2** for calculating R4. But in a pipeline when operands are fetched for second instruction the values from the first are still being calculated or saved hence causing a dependency. There is a dependency happens with instruction **I2**, as it waits for the completion of instruction **I1**.

2) Write after read: A write after read hazard occurs when an instruction refers to a result that is currently being written. This hazard poses a problem with concurrent execution.

$$I1 : R4 \leftarrow R1 + \mathbf{R5}$$

$$I2 : \mathbf{R5} \leftarrow R1 + R2$$

For example, considering concurrent execution pipeline, if **I2** finishes before **I1**, then **I1** will utilize a wrong value of R5 as its value has been changed by the second instruction. Such hazards will produce wrong results.

3) Write after write: A write after write hazard occurs when multiple instructions are trying to write into the same registers. Such a hazard is also known as output dependency.

$$I1 : \mathbf{R2} \leftarrow R1 + R3$$

$$I2 : \mathbf{R2} \leftarrow R2 + R3$$

For example, when **I2** is in the write back stage of the pipeline it has to wait for the execution of **I1**.

B. Register Renaming

It is pipelining form which can deal with data dependencies between instructions by renaming their register operands. The architectural register names are replaced, in effect, with value names, with a new value name for each instruction destination operand. Register renaming eliminates the output (WAW) and antidependencies (WAR).
The first renaming architecture was used for floating point instructions for the Tomasulo Algorithm. The register renaming is performed not by value registers instead using reservation station numbers for naming values. Each reservation station slot has space for two source operand values. The reservation station have the connectivity with circuits for updating the source operands when functional units produce result values. Register renaming is now ultilized in almost every desktop and server processor.

C. Out Of Order Execution

Out of order execution or dynamic execution is a paraadigm which utlilizes instruction cycles that would be wasted by costly delays. In this type of execution instructions are executed in an ordered provided by availability of input data, rather than original order of instructions. By doing this processor can avoid being idle during such stalls where instructions are waiting to complete their work. This allows execution of instructions which are able to run immediately and independently. Instruction operations can be performed out of order when there are no true dependencies between them. This is done by maintaining a status bit for each value indicating whether its finished computation or not.

III. Tomasulo Algorithm

Tomasulo algorithm is a method for provided dynamic scheduling in the processors. This allows controlling the execution of multiple functional units with varying latencies in a pipelined CPU micro-architecture. It has a general

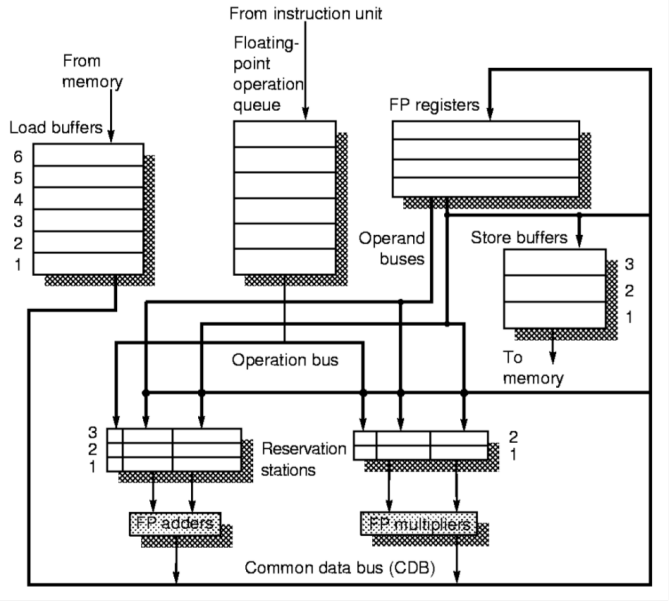

Fig. 1. Organisation of the tomasulo model

mechanism for register forwarding and hazard detection as the earlier algorithms. The major advantage of this algorithm is the out of order execution of the instructions. The key idea is to virtually execute multiple instructions in each cycle simultaneously.

A. Architecture

The major hardware structures and key features for implementing the tomasulo algorithm are the reservation stations , distributed instruction fetching and register renaming.

1) Reservation Station: It is a decentralized feature in a processor's microarchitecture that allows register renaming. This was the major changing approach of the tomasulo algorithm. These stations allow the CPU to fetch and re-use data values rapidly after its computation rather than watiing for it to be stored in a register and re-read again. It checks if operands are free of RAW hazards, hence are viable and also if the execution unit if free to eliminate the structural hazard before starting the execution. Implicitly the register renaming technique eliminates the WAR and WAW hazards. The instructions are issued in sequence to resevation stations which queue them in a buffer with their operands. If any operands are not available, the reservation stations channels into the CDB and listens for the operand to become available. Whenever the signal is recieved for availability of the operand on the bus, the reservation stations buffers it and execution of the instruction begins. Functional units also tend to have their own corresponding Reservation Station. The output of these units connect to the CDB, where the reservation stations are listening for needed operands.

2) Common data bus: The Common Data Bus (CDB) connect the reservation stations to the functional units present in the processor. This provides preservation of precedence while encouraging concurrency[2]. This provides two important results :
1. The functional units present in the processor have access to the various results of any operation with no involvement of floating - point registers. This allows multiple units which were wiaitng for a result proceed to execution. 2. The reservation station control the execution of anhy specific instruction. Also the hazard detection and control execution are distributed.

This connects all the functional units and load buffer to reservation stations, register and store buffers. This also ships results to all the hardware that could want an updated value. Eliminates RAW hazards hence not need to wait until registers are written before utilizing a value.

3) Distributed hazard detection: Each reservation decides when to dispatch instructions to its functional units. Each hardware data structure entry that needs a value from the CDB grabs the value itself by snooping. Reservation stations, store buffer entries and register have a tag saying where their data is coming from. When this tag has a match to the data producer's tag on the bus, reservation stations, store buffer entries and registers grab that data.

4) Dynamic memory disambiguation: The issue here is that we don't want loads to bypass stores to the same location. The solution her is to make the load to associatively check addresses in the store buffer, and if there is an address match then the value should be grabbed.

B. Working

The working of the tomasulo algorithm is done in three stages, issue , execute and write back.

1) Issue: Assuming that an instruction has been fetched then the fetched instruction is only issued if it doesn't pose any structural hazards. If there are no hazards then the instruction is issued or else it is stalled. The registers for the source operand are read. These values are then placed into the reservation stations if they have values. Then perform the register renaming on the functional units or load buffer. This provides the elimination of the WAR and WAW hazards.

2) Execute: During the execution phase the RAW hazard detection takes place. The missing operands are retrieved by snooping on the Common Data Bus. The instruction are dispatched to the functional units when both the operand values are retrieved. Then the operations are performed and the results are achieved. The effective address is calculated and the write back operation is started.

3) Write back: The result is broadcasted with the reservation station tag onto the common data bus (CDB) after the execution is complete. The reservation stations, registers and store buffers entries obtain the value through snooping.

IV. Simulator

A. Assumptions

In this project, the simulator simplifies the execution of instruction in the execute phase, where there is no multistaged

pipelining. In fact, in the real floating-point arithmetic unit, it is also implemented in a pipelined manner, that is, in the same floating-point arithmetic unit, there may be multiple floating-point arithmetic instructions being executed at different stages. For example, a floating-point multiplication unit has six-stage pipeline, then when the first incoming floating-point instruction enters into the subsequent pipeline section, the following instruction can enter the previous section of floating-point operation section to calculate, so as to further improve the pipeline's throughput and efficiency. The main focus of the project was to simulate the Tomasulo Algoirthm hence the absence of the multistaged pipelining for floating-point arithmetic.

In this project it has been considered that the instructions that are issued from the instruction queue must be in idle reservation stations without any direct access to the computational components. This provides an operational flow of each instruction can be uniformly processed. It is also considered that the floating-point multiplication and division are common to a single arithmetic unit.

B. Development Environment

For the simulator to be openly availble to all the students for education purposes, I have choosen a platform independent language JavaScript and have developed the simulator as a Web application. The simulator utilizes Vue.js framework for the algorithm simulation. The semantic interface utlizes beautify.JavaScript and uses the ECMAScript6 standard and uses some new features. This simulator is recommended to be ran in new browsers.

The simulator can be easily accessed by loading the index.html. The rest i.e jquery.min.js, vue.min.js, sematic.min.js in the project folder are runtime dependent libraries. The semantic.min.css is a cascading style sheet for the styling the web page. The main code is stored in the tSimulator.js. Two default input files for testing are also provided.

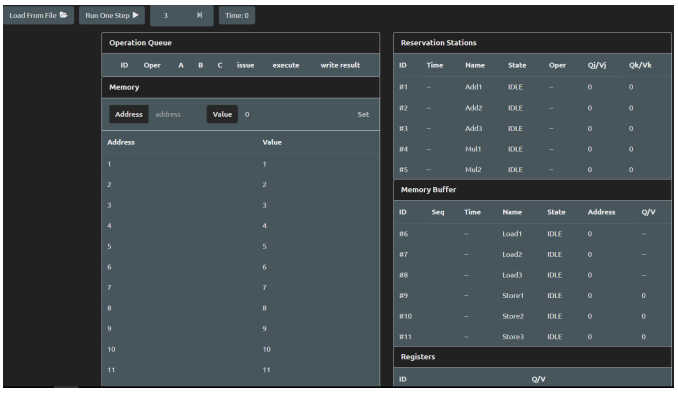

Fig. 2. Simulator interface when no file is loaded

Fig.2 depicts the basic simulator interface. Among them, the upper left Load From Filebutton can be loaded from the local file system files. For example, input1, input2. Fig.3 is the state after loading input1. The inside of the action bar Run One Step can be used for single-step debugging, and the right side can quickly run multiple steps by entering a positive integer. The rightmost is the current number of cycles (the last status, as shown in the Time: 0 represents the end of the first 0 clock cycle ready to enter the first clock cycle).

In the lower page content, the left Operation Queue is

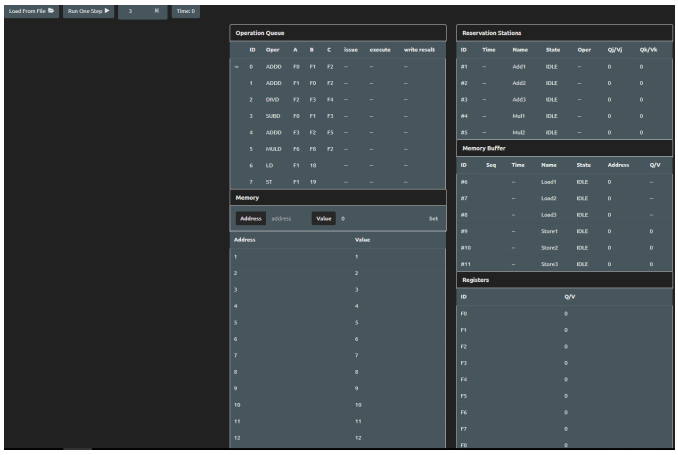

Fig. 3. Simulator interface when a MIPS input is loaded

loaded into the command queue, the arrow refers to the next instruction to be issued. ID for its number, Oper for its operation, ABC as its operand. Issue table the time of the instruction issue, execute record the start of the instruction / memory access, write result for the instruction to start the Write Back time. Memory is on the next side of the memory operation module. Below shows all memory cells that have non-zero values. You can manually specify the value of a memory location by entering the Address / Value at the top of the module. It should be noted that the updated memory cells will be placed at the bottom, rather than sorted by address size. ID indicates the number of the reservation station, State indicates the status of the instruction in the reservation station, IDLE indicates the idle wait for load instruction, ISSUED indicates that the instruction, WAITING indicates that it is in the waiting state (May be due to the computing unit is also busy may also be dependent on the operand is not calculated), RUNNING shows that its currently getting executed and WRITING states that its preparing to write back. Time indicates the amount of time that needs to be run during runtime. Q / V indicates the contents of the operand, pure numbers indicate values, and +/# numbers indicate dependent reserved station numbers. Oper record in the floating-point reservation station specific operation, the memory address of the address record to be stored in the address, Seq said the serial number. At the bottom of the Registers, the contents of the register file are recorded. Similarly, pure numbers indicate values and +/# numbers indicate dependent reserved station numbers.

C. Implementation

The main logic is in tomasulo.js. OpBuilder reads a line of content, into an Operations object. This function is mainly to help read the contents of the file. Operations class is used to store an instruction object, including its ID, operands, and various execution phases. QandV is a class used to interpret registers or operands. When Q is 0, its content is the actual value of V, and when Q is not 0, its contents depend on the Q reservation station. Class Reg is based on the QV class. ResStation is a reservation station. It has its name, status, required run time, operation and so on. Specifically divided into ALUStation floating-point reservation station and MemBuf to visit buffer storage. MemBuf is further divided into LoadBuffer and StoreBuffer. Different Reservation Stations are created to compute different works.

cleanArray, removeFromArray, addToArray are some auxiliary operations, due to some operation on the array in javascript are not directly reflected on the display section of the vue binding, so a more complex re-implementation is done with functions to facilitate the update data in the interface. loadTheFile is to achieve the function of reading instructions from the file. strMemory and ldMemory is a specific memory access operation.

runOneStep and runMultStep are single-step operation and multi-step operation of the entrance function, which specifically used issueand executes two functions (write back operation included in the execute function). In the issueInstr function there is a check to the next instruction in the instruction sequence to see if there is a corresponding free reservation station. If so, issue this instruction, load it into the reservation station, and update the contents of the corresponding output register in the register file. If there is no idle reservation station, the instruction flow is suspended. In the compute function, check the floating-point addition and subtraction reservation station, floating-point multiplication and division reservation station, Load memory buffer and Store memory buffer. First check if there is a reservation station that has finished running (in the WRITING state) and change it to the IDLE state. If there is an active (in RUNNING) reservation station, the remaining time is updated and, if processing is complete, the reservation station number is added to the notification queue and goes to WRITING state; if not, then an operation is found then the reservation station puts it in operation. The reservation station numbers in the notification queue are then fetched one after the other to update all the operands and register contents that depend on these reservation stations (the contents of the actual register are written back after the Write Back has completed but since after the execute stage has completed all location which relie on the value of this register are updated, the instructions to be entered can be bypassed to give the required data, so the register value is updated directly here). Finally, all the instructions pass the ISSUE WAITING state.

V. Results

The simulator was ran with various input files and the clock cycles were matched by doing the same procedure by hand. The simulator provided precise results for simple MIPS code patterns. The simulator can't handle loops hence all the codes were analyzed by unrolled code of the given instruction set. Consider the example of the following MIPS code in the input 2 file.

```
LD F1, 12
ST F1, 1
LD F2, 1
MULD F3, F1, F2
ADDD F3, F3, F3
```

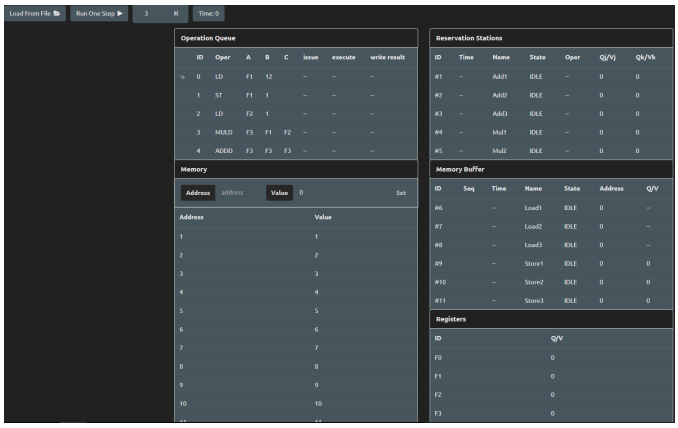

Fig. 4. Simulator status at the start of execution of given MIPS code

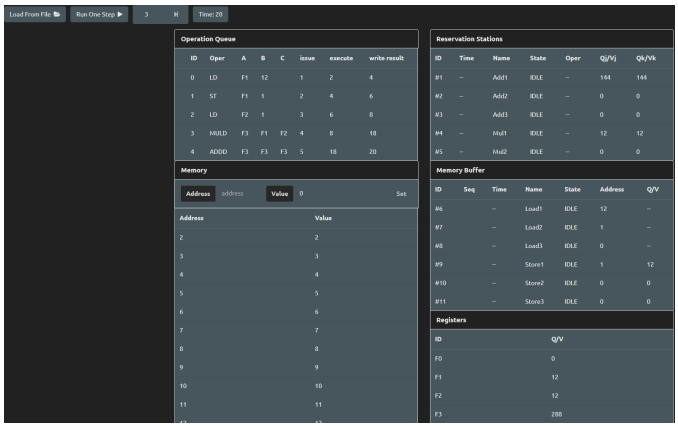

Fig. 5. Simulator status at the end of execution of given MIPS code

The default cycles for Load instructions are 3 , Store instructions are 3 , Add/Subtract are 4, Multiplication are 12 and for Division is 40 . As shown in Fig. 5, it takes 20 clock cycles for execution under these specific conditions. The status of the table can be seen for each cycle providing a better understanding of the algorithm. The status of the reservation becomes idle for all the functional units after finishing the execution.

VI. Conclusion

Tomasulo's algorithm can improve the efficiency of the pipeline for instructions that require different execution cycles to be able to perform out-of-order execution without conflict and logic, which is beneficial for the pipeline with floating-point arithmetic components. This implementation of Tomasulo's algorithm does some simplification, but the Tomasulo out-of-order execution and the core of the register renaming are all fully implemented. The scoreboarding algorithm would stall during a WAR hazard but in Tomasulo the fucntioonal unit is free to execute another instruction. And also in scoreboarding, the scoreboard keeps track of everything whereas in Tomasulo the control logic is distributed among the reservation stations.

This project deepened the understanding of the dynamic scheduling of pipelines and gave a more complete picture of the implementation details of the CPU cores in modern computers. In addition, because of using HTML and JavaScript, it can be easily deployed to web publishing. Such an easily accesible simulator can be utilized for educational purposes and also to debug any MIPS code for the status of each cycle.

Acknowledgment

I heartfully thank Dr.Ting Zhu and Mr. Yao Yao for providing me this opportunity to work on the simulation of Tomasulo Algorithm and also for guiding me by providing useful insights and support, without which the project wouldn't be finished.

References

[1] D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, The System/360 Model 91: Machine Philosophy and Instruction Handling, IBM Journal 11, 8 (1967)

[2] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. IBM J. of Research and Development, 11(1):2533, Jan. 1967.

[3] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In Computer-Aided Verification (CAV 94). Springer-Verlag, 1994.

[4] Hennessy, J. L. and Patterson, D. A. Computer Architecture: A Quantitative Approach, 3rd ed. Amsterdam, Morgan Kauffman. 2003, A-49.