Android-based Simulator to Support Tomasulo Algorithm Teaching and Learning

Dimitris Kehagias
Department of Informatics
T.E.I. of Athens
Greece

V. Douskas-Bertlviser
Department of Informatics
T.E.I. of Athens
Greece

ABSTRACT
Tomasulo's algorithm is a dynamic instruction scheduling algorithm that allows out-of-order execution, to minimize "Read-After-Write" (RAW) hazards and by register renaming to reduce "Write-After-Read" (WAR) and "Write-After-Write" (WAW) hazards. This paper describes an Android based simulator that shows how dynamic scheduling is obtained using Tomasulo's Algorithm. The simulator is configurable, while the simulation can be operated in a step by step mode and with animation in order to help students comprehend the concepts of dynamic scheduling anytime, anywhere.

General Terms
Computer Simulation, Algorithms, Hardware, Applied Computing.

Keywords
Tomasulo's algorithm, Simulator, Computer architecture, Interactive animation.

1. INTRODUCTION
Pipelining is extensively used in modern processors in order to achieve instruction level parallelism and improve performance. In a conventional pipelined processor there are 5-pipe stages, namely Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB) [9]. In the IF stage the program counter is used to get the instruction from instruction memory and put into the Instruction Register. In the ID stage, the instruction sent from the IR is decoded. The instructions are executed in the EX stage. The load/store instructions access memory during the MEM stage and in the last stage (WB) the results come from data memory or the ALU are written into the register file.

However, it is not always possible to run the pipeline at full capacity because of control, structural or data hazards. Data hazards - RAW, WAR or WAW - exist when reads and writes of data occur in a different order in the pipeline than in the program code.

Rearranging the execution sequence of instructions that belong to the same code segment can reduce data hazards and improve the performance. Dynamic scheduling algorithms such as Tomasulo and Scoreboard are examples of implementing the algorithms in hardware.

Tomasulo's algorithm was developed by R. Tomasulo at IBM in 1967 [3] and first used in the IBM System/360 Model 91 floating point unit. There are many variations on this algorithm in modern processors, although the key concepts of tracking instruction dependences to allow execution as soon as operands are available and renaming registers to avoid WAR and WAW hazards are common characteristics [1, 11].

1.1 Motivation
For students in an undergraduate advanced computer architecture course, implementing dynamic scheduling using Tomasulos' algorithm is often confusing as it is not that distinct. That's why Tomasulo simulation tools are used to support learning [13, 14].

Our intention to build a Tomasulo simulator was motivated by the fact that many students, in the undergraduate advanced computer architecture course offered by the Informatics department of the Technological Educational Institute (T.E.I.) of Athens [12], exhibit difficulties fully understand what Tomasulo's algorithm is doing clock cycle per clock cycle and how it is used to minimize data hazards.

1.2 Objectives
The main objective of the work presented in this paper was to create a suitable tool to support "dynamic scheduling" teaching and learning in the context of an "Advanced Computer Architecture" course and especially in the "Computer Architecture" and "Advanced Computer Architecture" courses, offered by the Informatics department of the Technological Educational Institute (T.E.I.) of Athens.

This simulator is an indispensable tool to clarify some important issues in these courses:

- How dynamic scheduling allows independent instructions behind a stall to proceed
- How an instruction can begin execution as soon as its operands are available
- How dynamic scheduling allows instructions to execute and complete out of order
- Register renaming

And it seeks to help students:

- Better understand hardware scheduling techniques for exploiting instruction-level parallelism
- To see how Tomasulo's algorithm implements dynamic scheduling
- Better understand the concept of reservation station
- To see visual representation of each stage of the algorithm (issue, execute and write back)
- To see how instructions are completed out of order
- To see how the algorithm eliminates WAW/WAR hazards
- To see how register renaming is provided by reservation stations

The rest of the paper is organized as follows. Section II provides a brief overview on the most relevant educational simulators. Section III explains the Tomasulo's algorithm. Section IV presents an overview of simulator implementation, its functioning and its features. Section V concludes the paper.

2. RELATED WORK

Various standalone tools exist to explain how dynamic scheduling is obtained using the Tomasulo's Algorithm. The most relevant ones are presented in the following lines:

In [5] a HASE simulation model, which closely follows the design of the IBM system 360/91 floating-point unit, has been built in order to demonstrate dynamically the Tomasulo's algorithm.

The simulator in [6] simulates Tomasulo's algorithm for a floating-point MIPS-like instruction pipeline, demonstrating out-of-order execution.

[7] and [4] present two web-based tools that have been developed for students to understand the concepts of the Tomasulo's algorithm used for dynamic scheduling.

However, none of them includes all the features that our proposal offers. These features include operation in a step by step mode, animation, written explanations in every animation step, configurable execution core, variable issue rate, variable latency per instruction class. Also, allows the user (i) to see memory contents during simulation, (ii) to show or hide animations, (iii) to move to the cycle in which some visible action occurs and get help during simulation. In addition, the android version of our simulator makes it a useful and unique tool, considering how android applications becoming very popular [8].

3. TOMASULO'S ALGORITHM

Figure 1 [1] shows the basic structure of a Tomasulo based processor. The major components of the processor are as follows [10]:

Reservation stations: these units receive an instruction from the instruction unit, wait for source operand data to be ready before starting the execution of the instruction and broadcast the result of the instruction on the Common Data Bus (CDB) when the result is ready.

Functional units: these are the circuits that perform the execution steps for an instruction. Example functional units are FP adders, FP multipliers, integer ALUs, shifters, and so on.

Register File: Contains the data produced by the functional units.

The CDB: connects the output of the functional units to all components expecting those results.

Load and store buffers: hold data and addresses for memory access.

Each instruction in Tomasulo's algorithm has 3 main stages. These are issue, execute and write back. In the issue stage, the next instruction from the top of the instruction queue is sent to an appropriate free reservation station with its operant values if they are available in the register file. If the operands are not in the register file, the instruction keeps track of the functional unit that is going to produce it. In effect, this stage renames registers. When all operands are available for an instruction, it will proceed the execute stage; otherwise, it waits for the operands to be available. That means the execution of instructions may be out of order. Once an instruction has finished executing, it enters the write back stage, where it will write its result to the CDB. Any instruction as well as registers waiting for this specific result will collect it from the CDB.

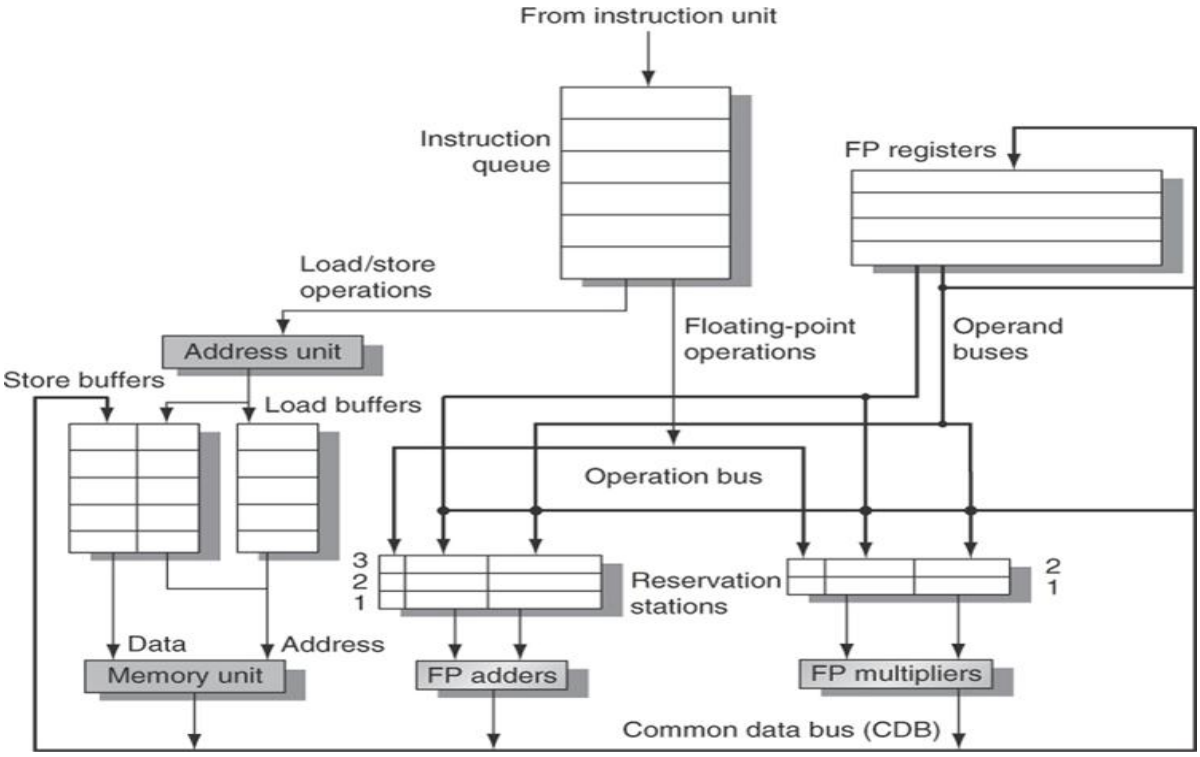

Figure 1[1]: The basic structure of a MIPS floating-point unit using Tomasulo's algorithm.

4. TOMASULO SIMULATOR

4.1 Functional Description

The code segment to be simulated can be changed by adding or deleting instructions of the segment. For each instruction, the destination and the source registers must be specified. The supported instructions are ADDD, SUBD, MULD, DIVD, LD and SD. Each type of instruction can have its own latency, ranging from 1 to 50. The ADDD and SUBD instructions are executed in the integer execution units, while MULD and DIVD are executed in the multi-cycle execution units. There are also load and store buffers to hold data and addresses for memory accesses. The number of units is also configurable, and can be set to 1, 2 or 3 units of each class. The simulation can be operated in all at once mode or in a step by step mode.

The instructions to be processed reside in an instruction queue, in the order entered by the user, waiting to be executed in first-in, first-out order. There are four stages an instruction goes through in order to complete its execution. These stages are issue, dispatch, execute and broadcast:

Issue: During the issue stage the next -in program order- instruction is taken from the instruction queue and placed into a reservation station of correct kind. In the case of load/store instructions, they are placed in a load/store queue. No instruction is issued if all the reservation stations or the load/store queues are occupied. An issued instruction to a reservation station is followed with its operands values if available or with associated tags indicating the reservation station that will produce the operands. In the issue stage an instruction monitors the CDB to see if it broadcasts the values it is waiting for, by comparing the tags it is waiting on with the tags of the instruction producing the result.

Dispatch: An instruction can be dispatched to a functional unit to start execution, when its source operands are ready and the corresponding functional unit is free. When an instruction is dispatched, its reservation station is freed.

Execute: Dispatched instructions get executed after a certain amount of time determined by the specific functional unit's delay, defined during initialization process.

Broadcast: Once a functional unit has finished executing an instruction outputs its result with the associated tag to the CDB for broadcasting. When a load instruction comes back from memory, the value that has been read is also broadcasted on the CDB. During broadcast: (a) the waiting instructions in reservation stations and in the store buffers get these results only if their operand entries match the tag of the instruction producing the result, (b) the appropriate register will be updated in the register file, and (c) the register allocation table entry that matches the broadcasted tag will be cleared. During this stage if there is more than one functional unit asking for the CDB in the same cycle, priority is given to the one which has completed an instruction with the highest execution latency. If in the same cycle the completed instructions have the same execution latency, they are broadcasted arbitrary.

4.2 Overview of Simulator Implementation

The simulator is implemented using Java in the Eclipse (Kepler) development environment. Genymotion [2] Android emulation has been used during the development of the application for testing. *As a starting point for our work, we have made the following assumptions*:

- Each instruction completes execution after successively passed the stages of issue, dispatch, execute and broadcast. In a single cycle, under normal circumstances, an instruction may be passed through stages issue and dispatch or dispatch and execute but not by the stages issue and execute.
- In each cycle only one instruction can pass by the broadcast stage. If multiple instructions are ready for broadcast, then priority is given to the one with the highest latency.
- A load instruction must wait before entering the execute stage, if an older store instruction with the same data memory address is also ready to enter the execute stage.
- A store instruction must wait before entering the execute stage, if an older load or store instruction with the same data memory address is also ready to enter the execute stage.

Classes used in simulator:

Each screen of the application is accompanied by an appropriate class that extends the Android Activity class, which supports the development of interfaces and activities. For each of these classes, the screen layout is defined by a corresponding xml file that includes all the necessary elements for describing the appearance of the interface in the user's mobile device. Also, for every activity there are several auxiliary classes that support user interaction with the simulation. Figure 2 shows the interconnection of classes of the application.

Verification:

During the development process we have made exhaustive tests to verify the correctness, functional behavior, and appearance of the application.

Initially the application was installed on many different mobile devices of different screen sizes, to improve and adapt the appearance of the various components in a way that there are no deviations from one device to another. Thus, the consistency in the appearance of the application on different devices was achieved with appropriate sharing of the space required by each component of the interface in conjunction with the animated parts during simulation.

The execution of the application was tested in order to properly implement the simulation of the algorithm, not led to a collapse, to handle exceptions that may arise during execution of the code and finally to be backward-compatible to mobile applications.

4.3 The User Interface

As shown in Figure 3, the application includes multiple interconnected screens. To ensure consistency in terms of graphical layout across the application, landscape orientation was chosen. The individual screens are: (a) Language selection screen: Upon starting the simulator, a user has the option to select between Greek or English language. (b) Help screen: it displays instructions for how to use the simulator, including description of various components of the simulation screen, and implementation assumptions that have been made during the designing phase. (c) Main screen: on this screen a user can choose from among several options, including entering code to be processed, starting simulation, configuring hardware, going back to starting screen, and reading the help text. (d) Code entering screen: This screen enables the user to enter instructions to be processed and initial values into registers and memory locations. A drop down list has been provided to select the required instruction. Each instruction is followed by three fields to choose the registers or memory

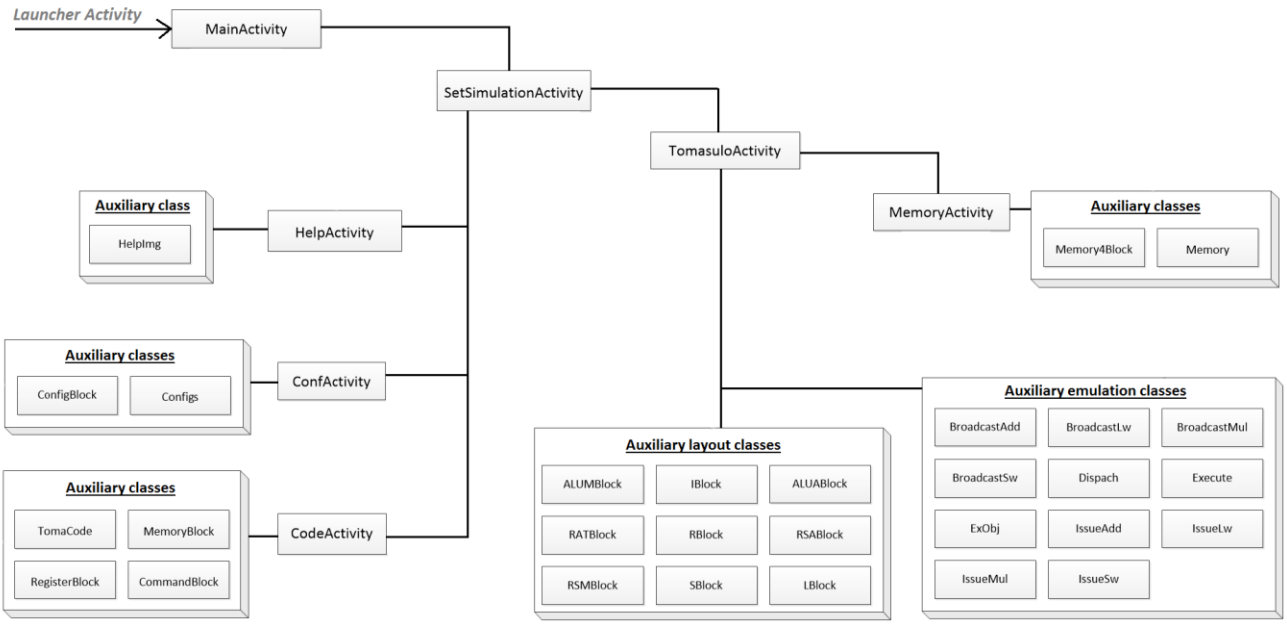

Figure 2: Interconnection of classes

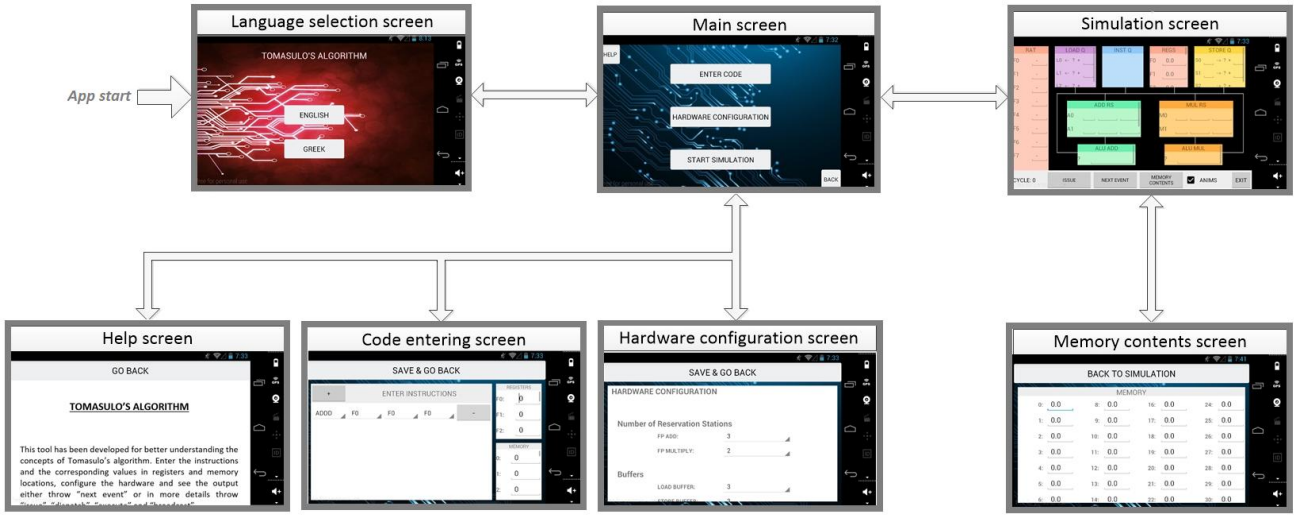

Figure 3: Arrangement of various screens

location relevant to each instruction selected. The initial values given in registers and memory locations are checked for validity. (e) Hardware configuration screen: On this screen the user defines the simulated execution environment, including the size of load/store buffers, the number of reservation stations, the number of execution cycles (latencies) taken by the functional units, and the number of functional units. (f) Memory contents screen: On this screen a user can view the contents of memory locations as they are formed during the execution of the algorithm. (g) Simulation screen (Figure 4): This screen is where simulation takes place. Its description follows in the next section.

4.3.1 The simulation screen

The simulation screen (Figure 4) has a very rich and friendly visual interface. It illustrates the movement of instructions to the reservation stations and the movement of results from the functional units. It consists with the following components:

RAT: Register Alias Table is a structure for performing register renaming. It maintains the mappings between reservation stations and destination registers of instructions.

LOAD Q / STORE Q: Load and store buffers for LD and SD instructions. They hold data and addresses for memory access.

INST Q: The "INST Q" component is a queue that contains the instructions in the order entered by the user. The instructions are issued into the reservation stations in first-in, first-out order.

REGS: The "REGS" component implements the Floating-point (F) and integer (R) register file. The registers contain values entered by the user during the configuration process, or broadcasted since instructions complete their execution. These values that are already in registers, meaning the values that are present and ready for execution, are entered to reservation stations.

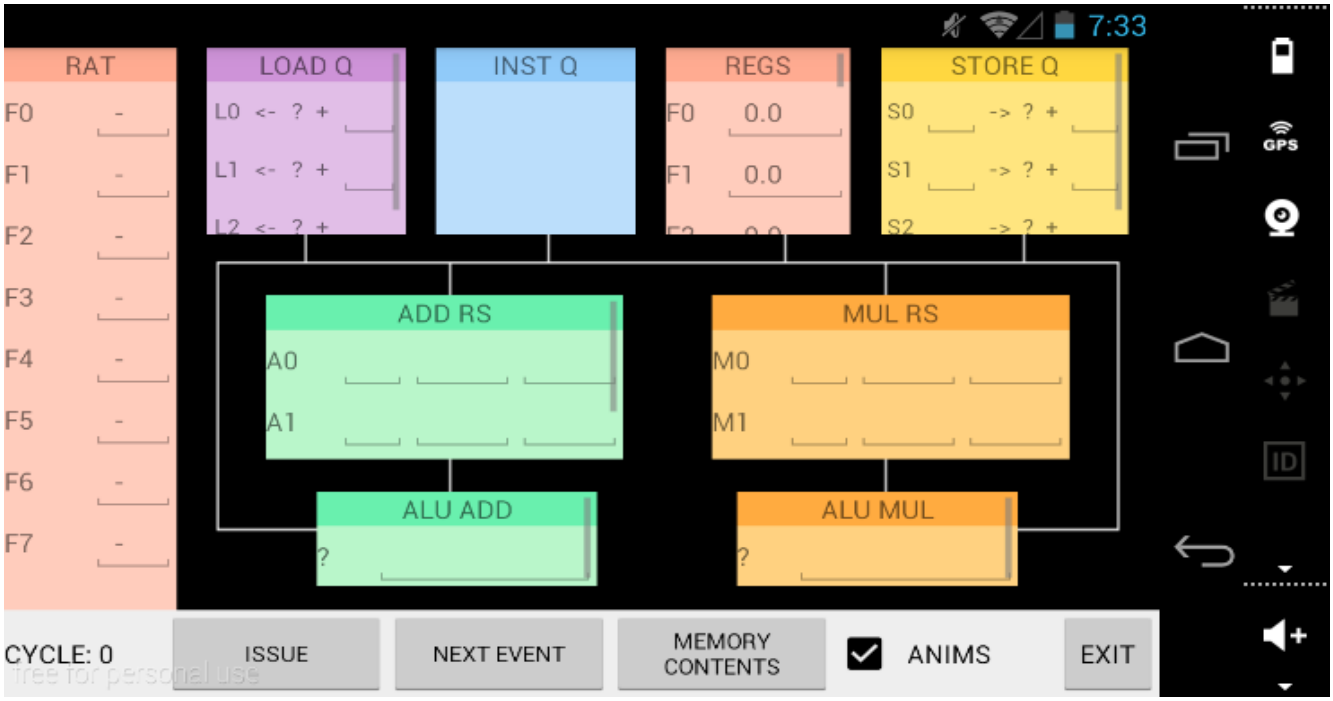

Figure 4: Simulation screen

ADD RS / MUL RS: There are two types of reservation stations "ADD RS" and "MUL RS". One is for ADDD and SUBD instructions, while the second is for MULTD and DIVD instructions. Each reservation station is made up of three fields. The first field in a row holds the opcode for the pending instruction in the form of an arithmetic symbol (+,-,*,/, for ADDD, SUBD, MULTD and DIVD instructions respectively) and the other two fields hold either operand values, or names of reservation stations or load/store buffers that will provide them.

ALU ADD / ALU MUL: Functional Units (FUs) to accomplish the execution step of instructions. The "ALU ADD" FUs are floating point adders which execute ADDD and SUBD instructions while the "ALU MUL" is floating point multipliers which execute MULTD and DIVD instructions. The FUs receive instruction and operand packets from the RSs and send operand result packets to the common data bus. The number of clock cycles required to execute an instruction is a parameter read from the hardware configuration activity at the start of a simulation.

All the above mentioned components are interconnected with a common data bus (CDB), which is used to broadcast result from the adder, multiplier and the load buffer to the reservation stations, the register file and the store buffers.

The simulation screen provides the user with several choices, including:

ISSUE: During the issue process the next -in program order- instruction is taken from the instruction queue and putted into a free reservation station of correct kind (ADD RS or MUL RS).

DISPATCH: The process of sending an instruction to execution from a reservation station to a functional unit (ADD RS to ALU ADD or MUL RS to ALU MUL).

EXECUTE: Is the phase during which a functional unit (ALU ADD or ALU MUL) operates on ready operands of an instruction.

BROADCAST: When an instruction finishes execution broadcasts its results on a common data bus and from there into registers and reservation stations.

NEXT EVENT: Allows the user to move to the cycle in which some visible action occurs.

MEMORY CONTENTS: Memory contents can be seen during simulation.

ANIMS: Show or hide animations.

5. CONCLUSION

A tool to aid students and teachers in an undergraduate advanced computer architecture course was presented. This tool, an Android based simulator, shows how dynamic scheduling is obtained using Tomasulo's Algorithm. Each stage of the simulation is represented with animation and with reference to flying information messages in order to give a clear and detail picture of the whole process. Different configurations of the simulator can be created, each with a different performance/resource ratio. Initial use of the simulator has shown learning effectiveness. The students were helped to better recognize the process of register renaming. In near future the simulator will be evaluated in the classroom through student surveys.

6. REFERENCES

[1] Hennessy J. L. and Patterson D. A., "Computer Architecture: A Quantitative Approach". Morgan Kaufmann, 5th Edition, 2012.

[2] Genymotion Android Emulator. Available at: https://www.genymotion.com/account/login. Accessed on Oct. 2016.

[3] Tomasulo R.M., "An efficient algorithm for exploiting multiple arithmetic units". IBM Journal of Research and Development, 11(1):25–33, 1967.

[4] "Tomasulo's Algorithm for Dynamic Scheduling". Available at: http://dark.eit.lth.se/darklab/tomasulo/script/tomasulo.htm. Accessed on Feb. 2017.

[5] "Tomasulo's Algorithm. University of Edinburgh". Available at: http://www.icsa.inf.ed.ac.uk/research/groups/hase/models/tomasulo/index.html. Accessed on Feb. 2017.

[6] Typanski N., "Tomasulo algorithm simulator (prototype)". Available at: http://nathantypanski.github.io/tomasulo-simulator/ Accessed on Feb. 2017.

[7] University of Massachusetts at Amherst. "Dynamic Scheduling Using Tomasulo's Algorithm". Available at: http://www.ecs.umass.edu/ece/koren/architecture/. Accessed on Feb. 2017.

[8] Butler M., "Android: Changing the Mobile Landscape". IEEE Pervasive Computing, vol. 10, no. 1, pp. 4 – 7, January-March 2011.

[9] Patterson D. A. and Hennessy J. L., "Computer Organization and Design - The Hardware/Software Interface". 5th ed., Morgan Kaufmann, 2014.

[10] "CSE P548 - Tomasulo", washington.edu. Washington University. 2006. Accessed on Feb. 2017.

[11] Hwang K. and Jotwani N., "Advanced Computer Architecture-Parallelism, Scalability, Programmability". 3rd ed., McGraw Hill, 2016.

[12] "Advanced Computer Architecture". Available at: http://www.cs.teiath.gr/?page_id=6450.

[13] Hatfield B. and Rieker M., "Incorporating simulation and implementation into teaching computer organization and architecture". 35th ASEE/IEEE Frontiers in Education Conf, Indianapolis, USA, pp: FIG-18, 2005.

[14] Carpinelli J. D., and Jaramillo F., "Simulation tools for digital design and computer organization and architecture". Paper presented at the 31st ASEE/ IEEE Frontiers in Education Conference, Reno, NV, 2001.