Educational Simulator for Analysing Pipelined LEGv8 (subset of ARMv8)Architecture

Jia Tian Chia, Smitha K G
School of Computer Science and Engineering
Nanyang Technological University
50 Nanyang Avenue, Singapore 639798
jchia033@e.ntu.edu.sg, smitha@ntu.edu.sg

***Abstract*—This paper presents the design and implementation of a pipelined Lessen Extrinsic Garrulity (LEGv8) architecture simulator, subset of ARMv8 architecture. The simulator is developed as a web application that can simulate the assembly of instructions in the LEGv8 assembly language, the execution of the instructions, and the visualisation of data path during execution. The simulator supports both single cycle and pipelined execution, with the option to select the control and data hazard handling methods to use. Users will be able to analyse the changes in the registers and memory, branching behaviour, hazard detection and elimination, as well as visualise data flow when stepping through instructions. This gives users the freedom to comprehend computer architecture more easily at their own pace by making use of the user-friendly and interactive educational simulator to enhance their understanding beyond what can be taught in the classroom.**

Keywords—Educational Simulator, Lessen Extrinsic Garrulity (LEGv8) Architecture, Advanced RISC Machines (ARM v8) architecture, Pipelined Architecture, Data Path Visualisation, Computer Architecture

I. INTRODUCTION

The LEGv8 architecture is a constrained subset of the Advanced RISC Machines (ARM) v8 architecture, which is used for teaching purposes, such as in the Computer Architecture module offered at various universities. The architecture employs 32-bit instructions, a 64-bit address bus, 64-bit data, and 32 registers with 64 bits each [1]. According to their functionality, the LEGv8 instruction set can be split into three categories: data-processing instructions, load and store instructions, and branch instructions [1].

Understanding the inner workings of computer architecture and computer hardware, such as the CPU and memory, can be challenging due to the inability to physically observe data flow and changes. This poses a barrier for students from diverse educational backgrounds who may find these concepts intimidating and challenging to visualise. Additionally, when learning assembly language, students often lack direct feedback on their code's correctness or any misconceptions they may have when simply learning assembly instruction execution, pipelining, and data flow using the traditional pencil-and-paper method. While static graphical diagrams and examples are often used during lessons, simulators are more beneficial as they allow students to insert their own code, manipulate memory values, and then explore and visualise the code execution simulation. Unfortunately, contrary to the abundance of simulators for other ISAs, such as the MARS, which is a well-liked interactive development environment for the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture [2], there are no comparable simulators available specifically for the LEGv8 (subset of ARMv8) architecture, making it necessary to develop a tool to support students' learning in education settings.

The main objective of the work presented in this paper was to create a web-based educational simulator that can be used to enhance students' understanding of computer architecture by providing them with a user-friendly and interactive medium to learn and analyse the execution of the basic set of instructions that the LEGv8 architecture supports. The scope of the simulator includes:

- Assembly of LEGv8 instructions with support for syntax highlighting and code linting in the code editor.
- Single cycle and pipelined execution with registers and memory values that can be initialised before execution.
- Data and control hazard detection and configurable hazard handling methods for hazard elimination.
- Data path visualisation during step-by-step execution.
- Error logging and execution statistics results.
- User-friendly and intuitive web-based visual interface.

II. RELATED WORKS

Although there is a lack of comprehensive and functional simulators created for the LEGv8 architecture, there are a variety of educational simulators available for other architectures. The following section will discuss and compare the features and limitations of some of these existing simulators.

A. MARS

MARS (MIPS Assembler and Runtime Simulator) is an educational tool designed for MIPS assembly language programming [2]. It serves as an integrated editor, assembler, simulator, and debugger for the MIPS processor. Its major strength lies in its interactive debugging capabilities, allowing users to modify registers and memory, set breakpoints, and step through execution as well as a user-friendly graphical interface. However, a limitation of MARS is its support for single cycle execution only, which can be overcome by using a plugin for pipelined execution with features like Data Forwarding and branch prediction [3].

B. DrMIPS

DrMIPS is a free and open-source graphical simulator of the MIPS processor designed for teaching and learning computer architecture [4]. It visually represents data path and supports the step-by-step execution of assembly programs. The simulator offers flexibility with different unicycle and pipeline data paths, allowing the configuration of jump or branch instructions, hazard detection, and data forwarding. It also allows the creation of CPUs with custom instruction sets

and provides relevant statistics such as clock period, CPI, and CPU cycles. However, its accessibility is limited, as it is only available as a desktop or Android application.

C. WebMIPS

The WebMIPS simulator is a web-based MIPS simulation environment that allows users to upload and assemble MIPS code, simulate a partially or fully five-stage pipeline, and view register and memory values, input/output data from pipeline elements and both the data path and control path on the diagram [5]. It only focuses on the fundamental set of instructions covered in an introductory computer architecture course. However, WebMIPS has limitations such as the absence of support for single cycle execution and an unintuitive and difficult-to-navigate web interface.

D. RIPES

RIPES is a comprehensive visual computer architecture simulator and assembly code editor designed for the RISC-V Instruction Set Architecture [6]. It stands out for its seamless integration of a built-in assembler, compiler support, and cache simulator centred around its visual microarchitecture simulator, enabling users to develop and test RISC-V programs and also provides a clear understanding of the inner workings of the RISC (Reduced Instruction Set Computer).

E. WebRISC-V

WebRISC-V is a web-based educational tool designed for exploring the pipelined execution of assembly programs based on the RV32IM and RV64IM specifications [7]. It focuses on enabling users to analyse and comprehend the impact of pipeline stalls on program execution as well as investigate the internal state of the pipeline components in the RISC-V architecture during step-by-step execution.

F. Comparison and Evaluation

TABLE I. EVALUTAION OF EXISTING WORKS

	MARS	DrMIPS	Web MIPS	RIPES	Web RISC-V
Platform	Desktop	Desktop, Android	Web	Desktop	Web
Single Cycle Execution	Yes	Yes	No	Yes	No
Pipeline Execution	Yes (Plugin)	Yes	Yes	Yes	Yes
Hazard Detection	Yes (Plugin)	Yes	Yes	Yes	Yes
Visual Data Path	Yes	Yes	Yes	Yes	Yes
Code Editor	Yes	Yes	Yes	Yes	Yes
Error Message Console	Yes	Yes	No	No	Yes
Effective and Attractive GUI	Yes	Yes	No	No	No

From Table 1, it can be seen that the simulators that offer the most comprehensive features for analysing execution are DrMIPS and MARS for understanding the MIPS architecture and RIPES for the RISC architecture. MARS has a user-friendly GUI with extensive use of tooltips and popups, while DrMIPS has a clear and organized interface. On the other hand, only WebMIPS and WebRSIC-V are fully web-based and easily accessible. They effectively make use of pop-ups to provide a lot of information about the data flow at every component in the architecture diagram but it also makes the interface relatively complex and difficult to navigate. It can also be observed that simulators supporting both single and pipelined execution are limited. By combining the strengths and functionalities of existing simulators, a user-friendly LEGv8 simulator for ARMv8 architecture, which integrates the most essential features in design and development from literature.

III. PROPOSED FEATURES AND IMPLEMENTATION

A web application is selected as the platform for the simulator to ensure maximum accessibility for users. It eliminates the need for downloading software and can be accessed through any browser on a computer with an internet connection, regardless of the operating system. The simulator is built using React.js, a front-end JavaScript UI library, along with third-party React components to enhance functionality and minimize development time.

A. Instruction Assembly

An assembler performs the role of converting assembly language to machine code that is stored in the text memory of the processor and can then be directly executed by the processor. During the assembly process, each line of instruction string is first parsed to break the line down into its constituent tokens, such as the instruction opcode, registers, and immediate values, and then analysed. This allows syntax errors to be identified when the source instruction does not match the grammar of the LEGv8 instructions. Additionally, name checking is also carried out to ensure that there is no duplicate use of label names and the labels referenced by branching instructions have been declared in the code. These syntax and semantic errors that are identified are passed to the code editor to be displayed with relevant error messages for easier debugging.

All LEGv8 instructions are thirty-two bits long, but instructions of different types are made up of a different combination of fields, which include the opcode, destination and source registers, and address of the destination, each field being of different lengths [8]. After each line of instruction is parsed into its constituent tokens, if no errors are identified, the value of each field is derived by converting the token to its binary value, which is then appended together to form the full 32-bit instruction, which is then allocated a memory address and stored in the text memory.

B. Single-Cycle Execution

During single cycle execution, only one instruction is being executed at any one time hence, the critical path with the longest delay determines the clock period. LEGv8 instructions can be classified into five main types according to their format: Register, Immediate, Data Transfer, Unconditional Branch and Conditional Branch [8].

The simulator in this paper only focuses on a small set of the most fundamental instructions from the extensive LEGv8 architecture, consisting of over fifty instructions. This selection effectively demonstrates the functionality of instructions from each instruction type, providing a basic understanding of computer architecture operations. The core set of instructions of each type that are supported by the simulator are shown in Table 2 below.

TABLE II. SUPPORTED INSTRUCTION SET

Instruction Type	Instruction Names & Mnemonics
Register (R) Type	Add (ADD), Add & Set flags (ADDS), Subtract (SUB), Subtract & Set flags (SUBS), And (AND), Or (ORR), Exclusive Or (EOR)
Immediate (I) Type	Add Immediate (ADDI), Add Immediate & Set flags (ADDIS), Subtract Immediate (SUBI), Subtract Immediate & Set flags (SUBIS), And Immediate (ANDI), Or Immediate (ORRI), Exclusive Or Immediate (EORI), Logical Shift Left (LSL), Logical Shift Right (LSR)
Data Transfer (D) Type	Load Register Unscaled offset (LDUR), Store Register Unscaled offset (STUR)
Branch (B) Type	Branch (B), Branch with Link (BL), Branch to Register (BR)
Conditional Branch (CB) Type	Compare & Branch if Not Zero (CBNZ), Compare & Branch if Zero (CBZ)

Instructions can be executed line by line by stepping forward and backward or entirely at once using the Run function by clicking the buttons in the top nav bar, as seen in Fig. 1. The next instruction to be executed will be the next instruction in the table unless the latest executed instruction is a B type instruction or the branch condition is met for a CB type instruction, whereby the next instruction will instead depend on the PC-relative address or label specified in the instruction. According to [8], during a procedure call in the LEGv8 processor, registers X0 to X7 are allocated for parameters and return values, while LR (X30) holds the return address. Each procedure call creates a stack frame, with the Frame Pointer (FP) pointing to the frame's start. To return from a procedure, the BR instruction unconditionally branches to the address stored in LR, and the temporary registers are popped from the stack and reloaded into registers. The values of stack pointer (SP), FP, and LR are restored to their values before the procedure call.

Values in the registers and data memory can be initialised before execution and changes in the registers, data memory, stack and the four flags: Negative, Zero, Overflow and Carry, that results from the execution of each instruction will be reflected and highlighted in the interface as shown in Fig. 1.

C. Pipelined Execution

Pipelined execution allows multiple sub-tasks to be carried out at the same time using independent resources. This increases the amount of useful work the processor can complete in a given length of time and decreases the cycle time of the processor, which often increases the throughput of instructions. A single cycle instruction execution can be split into five pipeline stages according to their functionality [8]:

- *Instruction Fetch (IF):* Fetch the current instruction from the instruction memory at the address stored in the PC.
- *Instruction Decode (ID):* Read values stored in the source registers and sign-extend immediate values.
- *Execute (EX):* Perform arithmetic and logical operations such as addition, subtraction and shifting using the ALU.
- *Memory Access (MEM):* Perform read or write on the memory.
- *Write Back (WB):* Write the results from the EX or MEM stage into the destination registers.

Pipelining allows the processor to be much more performant compared to single cycle execution as up to five instructions can be executed at one time with a 5-stage pipeline as seen from the five labelled instructions in Fig. 1.

D. Hazard Detection and Elimination

One of the complications that arise due to pipelining is the occurrence of pipeline hazards, which are events that arise due to dependency between concurrently executing instructions, disrupting pipeline flow, stalling the pipeline, and leading to a drop in efficiency. The two types of hazards that can be detected and eliminated by the simulator are data hazards and control hazards. To eliminate the pipeline hazards, their presence has to first be detected when the instruction is assembled, and then an appropriate number of NOP instructions, which are software stalls, need to be added.

Data hazards arise due to either the source or destination register being unavailable when it is needed, which results in a stall to wait for the needed value to become available. Therefore, to detect data hazards, the code needs to be checked for data dependencies. Table 3 shows the three ways that are supported by the simulator to handle true dependence, which will require a different number of stalls in between the two instructions with dependencies.

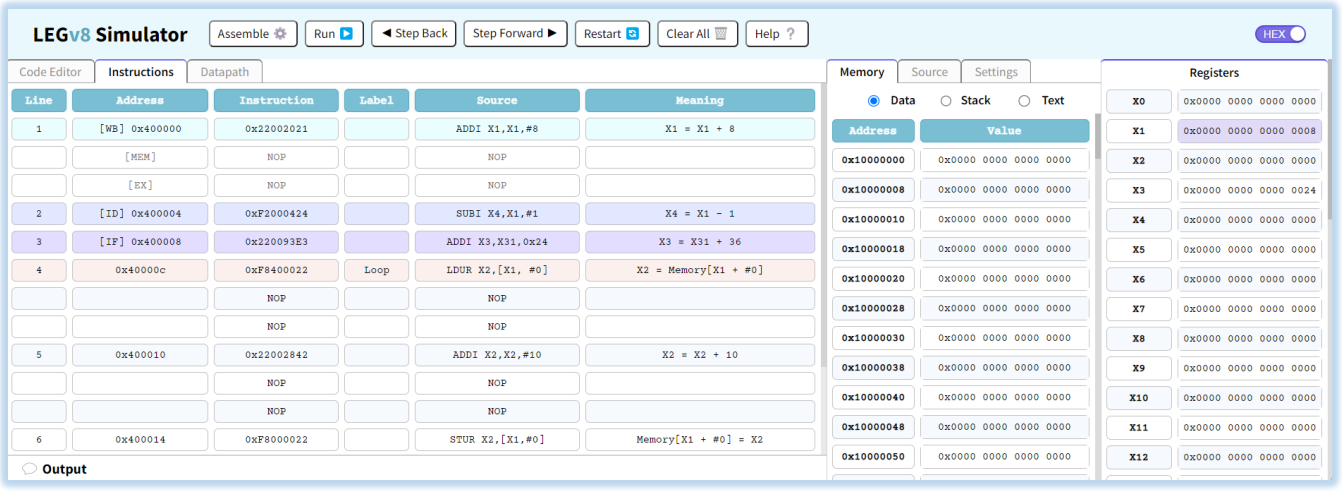

Fig. 1. Example of instruction execution step through for pipelined execution(settings to handle pipeline hazards is shown in Fig. 2)

TABLE III. DATA HAZARD ELIMINATION METHODS [8]

	No. of stalls
No data forwarding	3
Write back and decode simultaneously	2
Full data forwarding	0-1

Branching instructions cause control hazards because the pipeline has to wait for the branching outcome to be evaluated when executing B and CB type instructions before the subsequent execution sequence of instructions can be determined. The number of stalls required depends on the pipeline stage in which the Program Counter (PC) is updated upon the pipeline stage at which branch outcome is evaluated. The number of stalls needed can be reduced by evaluating and updating the PC value at an earlier stage, as seen in Table 4.

TABLE IV. CONTROL HAZARD ELIMINATION METHODS [8]

	No. of stalls
Jump or Branch instruction evaluated at MEM	3
Jump or Branch instruction evaluated at EXE Stage)	2
Jump or Branch instruction evaluated at ID Stage	1

Users can select and save the desired elimination method to simulate for both the hazards in the settings, as seen in Fig. 2 below, which will reassemble the code and updates the number of stalls that are inserted, and the corresponding data path diagram is displayed. This setting is used in Fig. 1

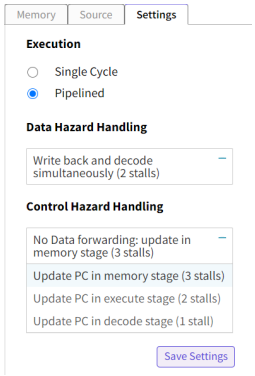

Fig. 2. Settings pane to select pipeline hazard handling options

E. *Data Path Visualisation*

The datapath diagram represents the flow of data that occurs during the execution of each instruction by displaying the inputs and outputs of each of the components of the processor in each clock cycle. The diagram helps users visualise each step of instruction execution so that it is easier to understand each type of LEGv8 instruction. The main elements of the datapath diagram are the PC and Instruction Memory used to fetch the current instruction in the Fetch stage, the Registers file to handle read and write to registers in the Decode and Write Back stages, the ALU to perform arithmetic and logical operations in the Execute stage, and Data Memory to handle memory access in the Memory stage [8]. To represent the data that is inputted to and outputted by each of the components, labels are used to show the value of the data, and they are positioned below the name of the architecture component as seen in the single cycle data path represented in Fig. 3.

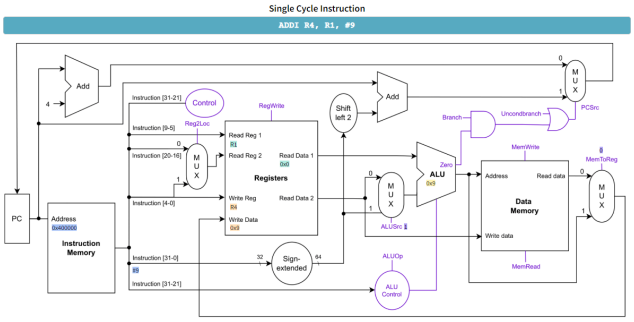

Fig. 3. Example of a data path visualisation for single cylce execution

Additionally, careful consideration has been given to making the data path visualisation more user-friendly and intuitive. For example, the control unit and control signals are coloured purple in the data path diagrams to make a separation between the control flow and data flow in the diagram. A deliberate decision is made to remove the lines connecting the control unit to each of the control signals so as to reduce the complexity of the data path diagram so that it will be easier to focus on the data flow with a less cluttered diagram, and the user can also zoom and pan in the diagram to focus on specific areas.

During single cycle execution as seen in Fig. 3, the diagram does not represent a snapshot of the execution at any instant in time as the data flow happens sequentially but instead shows all the data that has flowed through each of the components throughout the clock cycle to carry out the intended operation of the instruction. The output of each component is directly passed as the input to the next component that it is connected to, following the direction of the connecting arrow. Depending on the type of the instruction, not all the components of the diagram will be used and only the used components will be labelled.

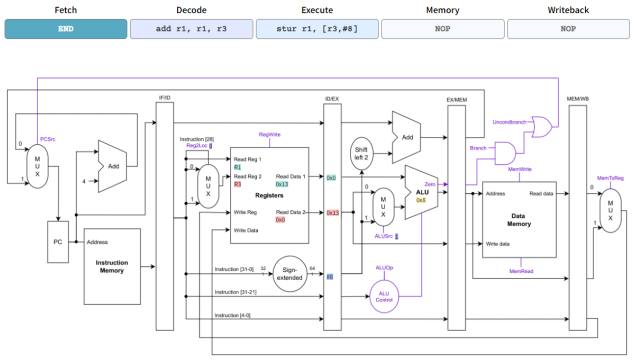

Fig. 4. Example of a data path visualisation for pipelined execution

The components of the datapath diagram for pipelined execution, shown in Fig. 4, are largely similar to those for single cycle execution, but with the addition of the pipeline buffers in between each of the pipeline stages, which are represented by the long rectangles labelled IF/ID, ID/EX, EX/MEM and MEM/WB. These buffers temporarily hold the outputs from the current clock cycle and then pass them as input for the next stage of execution in the next clock cycle.

Based on the options for data and control hazard handling for pipelined execution selected in the settings pane shown in Fig. 2, the datapath diagram will be updated. Firstly, to accommodate PC update in the desired stage

depending on the selected control hazard handling option, the logic gates that evaluate branching outcome, as seen in the MEM stage in Fig. 4, need to be repositioned. Additionally, when updating the PC in the ID stage, an additional zero checking component is required since the ALU is no longer available for this purpose. Secondly, additional components need to be added to enable data forwarding. For instance, data forwarding in the register file requires components to verify that the read and write registers are the same and the RegWrite control signal is enabled as illustrated in Fig. 8. However, to support data path visualisation with full data forwarding, the data path diagram would become too complex and hence, full data forwarding data path alone is not implemented in the simulator. Therefore, given that there are two implemented options for data hazard handling and three options for control hazard handling for data path visualisation, there are a total of six possible combinations of options and hence there are six different datapath diagrams for pipelined execution that are supported by the simulator.

F. Output

The output pane serves two purposes: to log execution error messages and show execution results. When errors are caught, the error messages and line numbers will be logged in the Output Pane. Secondly, when the last instruction in the instruction table has successfully executed without any errors, execution has completed, and an execution summary result will be generated. During pipelined execution, the total number of instructions executed, the number of stalls encountered, and the number of iterations, if there are loops present, will be recorded. The number of instructions executed and the number of stalls encountered are then used to calculate the steady state cycles per instruction (CPI).

IV. Evaluation

The evaluation process involves functional testing, specifically black box testing, and user acceptance testing. By conducting these evaluations, any necessary improvements can be made to enhance the simulator's functionality and user experience, ensuring its effectiveness and success.

A. Functional Testing

Functional testing assesses the simulator's performance, identifies any potential bugs or issues, and verifies that it operates as expected. Black box testing is a software testing method that only looks at the functionality of the product without considering its code structure or internal workings. The following section will cover an illustrative test case to demonstrate and verify the assembly, execution and data path visualisation functionalities of the simulator.

TABLE V. Illustrative Test Case (ref Table 2 for instructions in testcode)

Test Code	(1) ADDI R2, R1, 0xA (2) B loop (3) LSL R1, R1, #4 (4) loop: STUR R2, [R3, #8] (5) SUBI R1, R2, #4 (6) LDUR R3, [R1,#2]
Test Conditions	1. Pipelined Execution 2. Data Hazard Handling: Write back and decode simultaneously (2 stalls) 3. Control Hazard Handling: No data forwarding, update in MEM stage (3 stalls)

Table 5 shows the test code and conditions for testing the complete user flow when stepping through instructions using pipelined execution. After successfully assembling the instructions without any errors, NOP instructions will be inserted between the source instructions based on the options selected to handle the pipeline hazards.

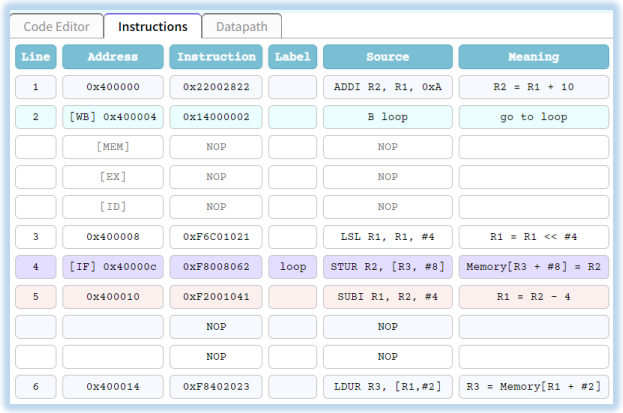

Fig. 5. Stepping through assembled instructions after stalls are inserted

As seen from Fig. 5, three stalls are inserted between lines (2) and (3) as line (2) is an unconditional branch (B) which is a branching instruction that incurs a control hazard. Two more stalls are also inserted between lines (5) and (6) as there is a read-after-write dependency on R3 between the two instructions. By stepping through the instructions, the expected execution flow is as follows:

1. R2 updates to 0xA (R1 (0_{10}) + 0xA = 0xA)
2. PC updates to the address of Line 4 with the label "loop"
3. Value of R2 (0xA) is stored at base-relative memory address of 8_{10} since value of R3 (0_{10}) + 8_{10} = 8_{10}
4. R1 updates to 0x6 (R2 (0xA) – 4_{10} = 0x6)
5. Value of R3 updates to the value stored at base-relative memory address of 8 (0xA) since value of R1 (6_{10}) + 2_{10} = 8

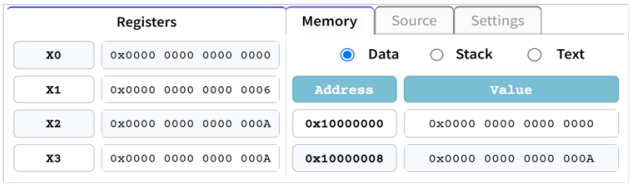

Fig. 6. Final state of registers and data memory after execution

Fig. 6. correctly reflects the expected final state of updated registers and memory in the simulator after all the instructions have finished execution. It can also be observed that line (3) is skipped as line (2) causes an unconditional branch to the label, loop, which starts at line (4). Lastly, upon completion of execution, the execution statistics are displayed as shown in Fig. 7.

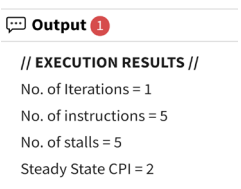

Fig. 7. Execution results statistics

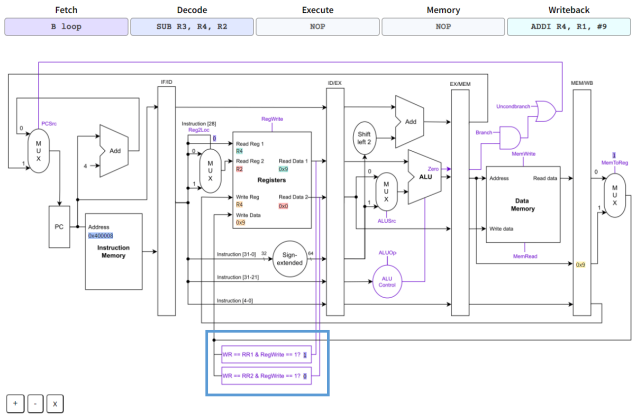

Fig. 8. Data path visualisation during instruction step through(writeback and decode simultaneously)

In addition, the data path visualisation should use the correct data path diagram, reflecting the data flow instruction step through based on the pipeline hazard handling method as per the settings in Fig. 2. The data path diagram in Fig. 8 highlights the additional hardware added to the architecture to allow for write back and decode simultaneously, as well as the branching logic gate being in the MEM stage according to the test conditions indicated in Table 5 (PC updated in MEM stage).

Overall, functional testing is conducted using test cases that cover both single cycle and pipelined execution for all combinations of pipeline hazard handling methods. Faulty code is also tested to ensure that syntax and semantic errors are caught during assembly and that execution errors are properly caught and logged without crashing the simulator.

B. User Acceptance Testing

A group of twenty students who are currently learning or have learnt the LEGv8 architecture through the computer architecture course taught at Nanyang Technological University were given an online survey to complete after they had time to learn and use the simulator. The aim was to evaluate the usability of the simulator and the users' overall satisfaction and experience with the simulator.

TABLE VI. RESULTS OF SURVEY

Question	Average Score
The LEGv8 architecture difficult to understand when taking the Advanced Computer Architecture module	4.4
The simulator is easy to learn and understand with the instructions provided.	4.1
The simulator is user-friendly.	4.4
The simulator is a useful tool to further enhance students' understanding of the LEGv8 architecture.	4.6

The survey results in Table 6 show that all of the respondents agreed that they found the LEGv8 architecture difficult to understand, which highlights the limitations of traditional course materials in providing students with sufficient understanding of complex computer architecture concepts. In general, the participants found the simulator easy to learn and user-friendly and some commented that the layout of the user interface is intuitive to use especially with the use of visual aids. Overall, all of the respondents felt that the simulator was able to improve their understanding of the LEGv8 architecture, especially for visualising pipelined execution and data path which are especially difficult to understand when taught using the traditional paper and pen method. This shows that the main objective of the simulator, which was to enhance students' understanding of computer architecture, was able successfully met. Lastly, some comments from the users could be used to make future improvements such as to add more explanation regarding the number of stall insertions.

V. CONCLUSION

The objective of this project was to develop an educational simulator for the LEGv8 architecture, which currently lacks a dedicated simulator. The simulator allows users to assemble LEGv8 assembly code, execute instructions for single cycle and pipelined architectures that can be configured, and visualize the data path. It provides a user-friendly and interactive tool for students to analyse their code, understand register and memory changes during execution, and visualize data flow in the processor. The aim is to make learning computer architecture more accessible and less challenging for students. The simulator is available online at http://jiatian2300.github.io/LEGv8-Simulator and is currently used as a teaching tool in the Advanced Computer Architecture module at Nanyang Technological University to help students better understand the hidden details of a pipelined processor and how the performance of the programs can be enhanced by appropriate optimisations. Future work can include implementing data path diagrams for architectures with full data forwarding, displaying and including explanations for identified pipeline hazards, adding more labels and details to labels in the data path diagrams, and supporting dynamic branch prediction.

ACKNOWLEDGMENT

The authors express their gratitude to the School of Computer Science and Engineering at Nanyang Technological University for the support provided.

REFERENCES

[1] ARM, "ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile," 2013. [Online]. Available: https://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf.

[2] K. Vollmar and P. Sanderson, "MARS: an education-oriented MIPS assembly language simulator," in ACM SIGCSE Bulletin, vol. 38, no. 1, pp. 239-243, Mar. 2006, doi: 10.1145/1124706.1121415.

[3] D. X. Lim and K. G. Smitha, "Pipelined MIPS Simulation: A plug-in to MARS simulator for supporting pipeline simulation and branch prediction," 2019 IEEE International Conference on Engineering, Technology and Education (TALE), Yogyakarta, Indonesia, 2019, pp. 1-7, doi: 10.1109/TALE48000.2019.9225934.

[4] B. Nova, J. C. Ferreira and A. Araújo, "Tool to support computer architecture teaching and learning," 2013 1st International Conference of the Portuguese Society for Engineering Education (CISPEE), Porto, Portugal, 2013, pp. 1-8, doi: 10.1109/CISPEE.2013.6701965.

[5] I. Branović, R. Giorgi, and E. Martinelli, "WebMIPS: a new web-based MIPS simulation environment for computer architecture education," in Proceedings of the 2004 workshop on Computer architecture education, WCAE '04, June 2004.

[6] M. B. Petersen, "Ripes: A Visual Computer Architecture Simulator," 2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE), Raleigh, NC, USA, 2021, pp. 1-8, doi: 10.1109/WCAE53984.2021.9707149.

[7] R. Giorgi and G. Mariotti, "WebRISC-V: a Web-Based Education-Oriented RISC-V Pipeline Simulation Environment," in Proceedings of the 2019 ACM International Conference on Interactive Surfaces and Spaces, 2019, pp. 1-6, doi: 10.1145/3338698.3338894.

[8] D. A. Patterson and J. L. Hennessy, Computer Organization and Design ARM Edition: The Hardware Software Interface. Cambridge, MA: Elsevier, Morgan Kaufmann Publishers, 2016.