SIMULATION OF PIPELINED MIPS FLOATING-POINT UNITS USING NODE-RED

Bryan McClain[1], Jinyu Fang[1], Prathamesh Kale[1] and John J. Lee[2]

[1]Department of Computer & Information Science, IUPUI, Indiana, USA
[2]Department of Electrical and Computer Engineering, IUPUI, Indiana, USA

ABSTRACT

The pipelined processor architecture is the best way to increase instruction-level parallelism, and thus, understanding its operation is one of the keys in computer architecture learning. To help with the leaning process, we have devised a series of pipeline simulation methodologies. This article presents one of them – a simulation methodology of hazard detection and forwarding in MIPS32 pipelined floating-point units. Our implementation approach is using the Node-RED programming environment, an event-driven drag-and-drop system for designing data flows with business logic. In addition, to implement sophisticated operations not supported by Node-RED, we also employ WebAssembly code and the Rust language. We simulate not only the standard 5-stage pipelined MIPS ISA but also pipelined (addition, subtraction, and multiplication) and unpipelined (division) floating-point operations. Our study focuses mainly on hazard detection, which is required to ensure program correctness, as well as forwarding used to improve system performance. Lastly, we design a dashboard interface to visually represent the pipeline stages and CPU status during execution. Using the dashboard interface, MIPS32 machine code can be loaded into our simulator from hexadecimal text files. We verified that our simulator is handling hazard detection and forwarding correctly. A screenshot of the dashboard interface is included that shows all the stages of floating-point pipelines.

KEYWORDS

Instruction Set Architecture, Pipelining, Hazard Detection, Forwarding, Floating-Point Operations, Microprocessor Simulation, MIPS32.

1. INTRODUCTION

The MIPS32 ISA is a 32-bit reduced instruction set computer (RISC) CPU architecture [1]. Hailed for its instruction simplicity, performance, and low power consumption, MIPS is one of standard choices for use in embedded systems. To improve instruction throughput and overall performance, MIPS CPUs employ the use of pipelining, allowing multiple instructions to be executed concurrently. As part of ensuring program correctness, the pipeline needs to properly stall certain stages to avoid data, structural, and control hazards. The pipeline may also employ the use of forwarding, which reads the results from later stages and bypasses the register file, thus reducing the number of stalls and improving performance.

Although the base MIPS32 ISA only includes support for integer arithmetic and multiplication, processor implementations may also contain an additional unit for IEEE 754 floating-point arithmetic [2]. Referred to as "coprocessor 1" in the official documentation [1], this unit enhances the MIPS CPU with a variety of floating-point mathematical operations. In a pipelined architecture, these operations often take multiple cycles to compute and thus significantly complicate the CPU architecture. Some operations (such as addition and multiplication) can be

pipelined, whereas other operations (like division) cannot be easily pipelined. Such factors need to be considered when handling hazard detection and forwarding in the pipeline.

In this article, we simulate a pipelined MIPS CPU with the floating-point coprocessor unit, focusing specifically on hazard detection and forwarding. Our simulator has been implemented using the Node-RED [3] programming environment, which is built on top of the Node.js framework [4] and written in the JavaScript programming language [5]. Node-RED provides a single-threaded event-driven environment for executing a sequence of operations, referred to as a "flow", which is programmed with JavaScript code using the Node-RED drag-and-drop interface. Our simulator decodes and runs real MIPS machine code, simulating a subset of the MIPS instructions and floating-point instructions. We also employ the use of a custom WebAssembly [6] library for handling some of the low-level operations that are difficult (if not impossible) to implement using JavaScript code. Our simulator also includes a visual dashboard to show the execution status of the simulation, which can be a great learning tool for computer architecture students.

Specifically, our contributions include the following:

- Visualization of the status of a 5-stage MIPS pipelined CPU
- Supports for multi-cycle pipelined IEEE-754 floating-point function units
- Supports for pipeline forwarding and hazard detection
- Shows disassembled assembly outputs
- Node-RED based and thus it can be easily modifiable with new ideas as the learning curve of Node-RED is very low, thereby making it excellent for educational purposes

The article is laid out as follows: in Section 2, we discuss related work to this study. Section 3 presents the low-level details of the MIPS floating-point coprocessor related to this study. Sections 4 and 5 discuss our specific implementation details about the pipeline stages and Web Assembly library. Then, Section 6 and Section 7 discuss the hazard detection unit and forwarding unit, respectively. After that, Section 8 explains the visual dashboard interface. Finally, Section 9 concludes this article.

2. RELATED WORK

Microprocessor simulation has been a very hot topic for several decades. Although there have been numerous microprocessor simulators, we focus this section on educational simulators with visualization features as this study is one of them. The MIPS assembler and runtime simulator (MARS) [7] has been mainly utilized for computer architecture education as Patterson and Hennessy [8] describe RISC machine using the MIPS architecture. The greatest advantage of MARS is that it provides GUI functionality. It helps students see through how a microprocessor executes MIPS instructions with visible register values, memory values, and program status. However, MARS has the following drawbacks. First, it does not simulate pipelined architecture, hazard detection, or forwarding. Second, MARS does not allow new design, which may limit the students' knowledge of computer architecture.

Arif [9] developed a highly visual ARM simulator that has many features for visualizing instructions, memory, and branches, but it lacks a pipeline visualization. Kabir et al [10,11] developed and published a visual simulator of MIPS32 pipelined processor that can visualize assembly source and detailed signal values of various processor internal components. However, the simulator lacks support for multiplication and division of integer as well as floating-point numbers, and also its Java-based implementation has a high learning curve. Recently Anderson et

al [12] presented a Node-RED-based MIPS-32 processor simulator which has features of 5-stage pipeline visualization, cache configuration and its statistics visualization, operand forwarding for the resolution of data dependency, and branch prediction mechanisms. However, the simulator does not have multi-cycle floating-point functional units and their forwarding, nor hazard detection mechanisms.

3. FLOATING-POINT COPROCESSOR

The floating-point coprocessor extends the standard MIPS instruction set to include operations for floating-point mathematics. In the sections below, we will discuss some of the low-level details for working with the floating-point coprocessor.

3.1. Registers

The coprocessor contains 32 registers, labeled f0 to f31, which are used for storing floating-point values. These registers are separate from the general-purpose registers (r0 to r31). Each register is 32 bits wide and represents an IEEE 754 single-precision floating-point number [2]. 64-bit pairs of registers (f0-f1, f2-f3, etc.) can also be combined to produce an IEEE 754 double-precision floating-point number [2], but our project only simulates single-precision operations.

Floating-point registers can be loaded directly from memory using the *lwc1* opcode (load word coprocessor 1) and stored back to memory using the *swc1* opcode (store word coprocessor 1). Values can also be moved between the general-purpose registers and the floating-point registers by using the *mtc1* (move to coprocessor 1) and *mfc1* (move from coprocessor 1) opcodes. In this way, a MIPS program can use immediate instructions to load a floating-point value into a general-purpose register, and then copy the value into a floating-point register. Finally, floating-point registers can be moved between each other using the *mov.s* (move single-precision) opcode.

3.2. Operations

Floating-point opcodes are prefixed based on the precision, with *.s* for single-precision and *.d* for double precision. However, as previously stated, this study only focuses on single-precision operations. Due to the complexity of floating-point numbers, operations may take multiple execution cycles to compute the result. The list of supported floating-point operations is summarized in Table 1.

Table 1. Summary of floating-point operations supported in the simulation

Opcode	Execution Cycles	Pipelined	Description
add.s	4	Yes	Add two single-precision floats
sub.s	4	Yes	Subtract two single-precision floats
mul.s	7	Yes	Multiply two single-precision floats
div.s	24	No	Divide two single-precision floats
abs.s	1	Yes	Absolute-value of a single-precision float
neg.s	1	Yes	Negate the sign (+ or -) of a single-precision float

3.3. Branches

The floating-point coprocessor has eight condition code (cc) flags used for branching. To perform a branch, a cc specific flag must first be set by a floating-point comparison instruction. Then, a branch can be performed by calling either *bc1t* (branch if coprocessor 1 true) or *bc1f* (branch if

coprocessor 1 false), which branches based on the value of the specified condition code register. These comparison and branch instructions take a cc index as an integer (0 to 7) which specifies which specific flag to set or check.

Our simulator supports three floating-point comparison instructions: *c.eq.s* (compare equal single-precision), *c.lt.s* (compare less-than single-precision), and *c.le.s* (compare less-than or equal-to single-precision). By combining multiple cc flags and branches, these instructions can be used to perform any combination of floating-point comparisons.

4. PIPELINE IMPLEMENTATION

In our simulation using Node-RED, each pipeline stage is represented by a different function node. To simulate pipeline buffers between function nodes, we use the *flow* object in Node-RED to store global objects, with a different object between each pipeline stage. The JavaScript value *null* is used to represent a *No-op* inside a buffer. The flow object also stores the register values, condition code flags, program counter, global cycle counter, and memory contents. In this way, the current state of the CPU can always be determined by reading the global objects, which is how the dashboard has been implemented (we discuss the dashboard in Section 8). The Node-RED code uses a unified register file where registers r0 to r31 have the index 0 to 31, and registers f0 to f31 have the index 32 to 63. This allows all registers to be uniquely identified when handling hazard detection and forwarding.

During execution, all function nodes receive a rising edge followed by a falling edge clock message. In general, execution occurs during the rising edge of the clock cycle, and results are written to the next buffer during the falling edge. After reading a buffer in the rising edge, each stage sets the previous buffer to *null* so the previous stage can write to it.

To simplify hazard detection and prevention, each stage has a global flag that can be used to stall the stage for the current clock cycle. If the flag is set, the stage does not perform any of its normal operations. Each stage clears its respective global flag during the falling edge of the clock cycle. With this mechanism, the hazard detection unit (discussed in the next section) can independently stall any pipeline stage to prevent data and structural hazards.

The block diagram view of the complete pipeline datapath can be seen in Figure 1, which lists the named stages and corresponding buffers. A screenshot of the working Node-RED flow is shown in Figure 2 which is our implementation of Figure 1. The main datapath (boxed in green around the center of Figure 2) contains the 5 stages of a standard MIPS32 pipeline (fetch, decode, execute, memory, and write-back). The addition unit (boxed in blue) and multiplication unit (boxed in purple) have 4 and 7 stages, respectively to simulate these instructions taking multiple cycles. Finally, the division unit (boxed in red) simulates a non-pipelined instruction that takes 24 cycles to execute. Many of the pipeline stages output additional information to debug nodes for logging purposes (the small green boxes). On the left is the program clock, which first calls the hazard detection unit and the forwarding unit, and then sends a rising-edge and a falling-edge to each pipeline stage. The top flow handles CPU initialization and loads the machine code file into Node-RED. Note that the dashboard code has been omitted from this diagram but is also part of this Node-RED flow. We now discuss specific details about each pipeline stage.

4.1. Fetch

The fetch stage does nothing on the rising clock edge. On the falling edge, it fetches an instruction from memory and increments the program counter and global cycle counter. In our

simulation, all memory accesses must be word-aligned, so the fetch always sets the lower two bits of the program counter to 0 before reading from memory. When passing the instruction into the next buffer, it also stores the start cycle to track the order in which instructions are issued. By waiting until the falling edge to fetch the next instruction, the simulation only wastes one cycle (instead of two) for a branch or jump.

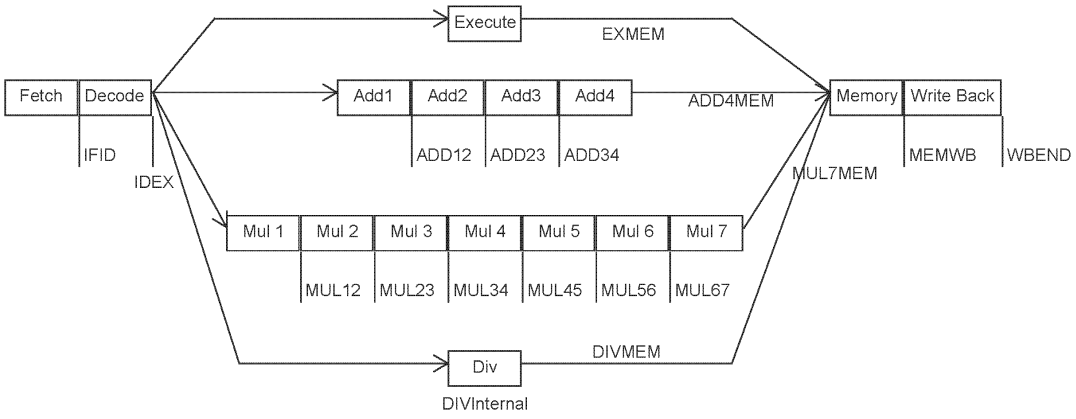

Figure 1. List of pipeline stages with the corresponding buffer names.

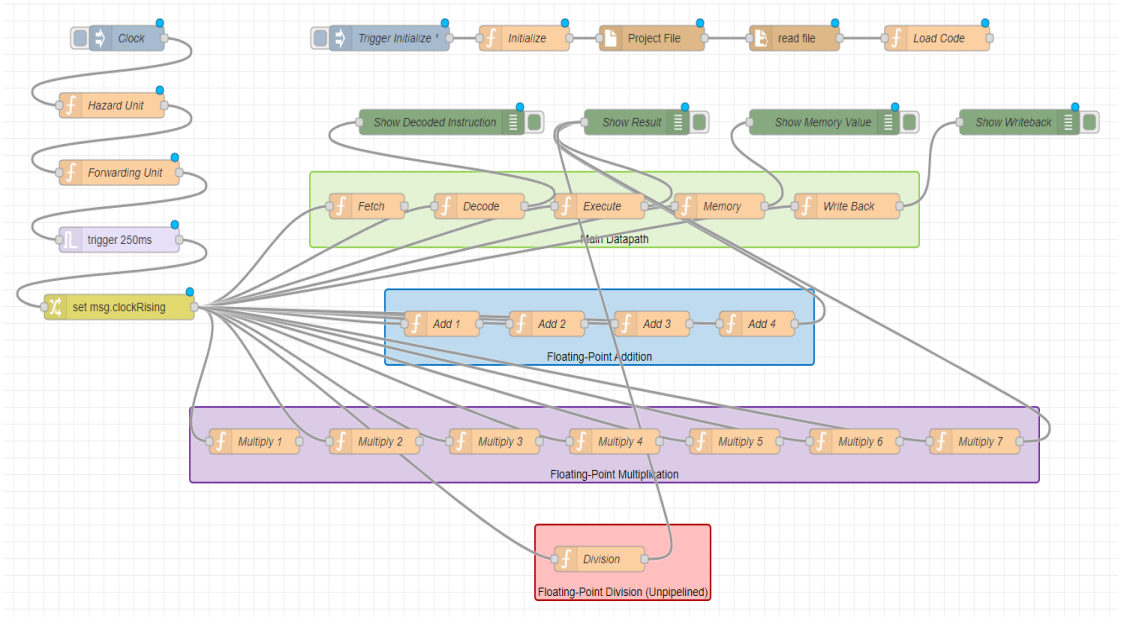

Figure 2. High-level Node-RED flow view of the simulated pipelined processor.

4.2. Decode

Instruction decoding happens during the rising clock edge, and then registers are read during the falling clock edge. In our simulation, instructions are stored in memory as 32-bit big-endian integers. The Node-RED code parses the individual bits from the instruction to determine the opcode and operands needed for execution, making this one of the most complicated nodes in our simulator. However, the benefit is that our simulator parses real MIPS32 machine code and can show disassembled instructions.

When parsing the instruction bit-fields, the decode stage sets the *rs* and *rt* (source registers) and the *rd* (destination register) inside the buffer. Registers are given a unified index from 0 to 63 to uniquely distinguish between the general-purpose registers and the floating-point registers. The decode stage also sets additional fields, such as the shift amount, immediate field, jump address, and condition code flag. If any register or field is unused by the instruction, it is set to 0. The instruction opcode, once decoded, is stored as a string and used by the execute stage to decide what computation to perform. This stage also uses string formatting to generate the assembly-code instruction, which is used by the dashboard to display the complete instruction being executed by each stage.

For dealing with branching, the decode stage has a special *flush* flag that causes it to discard the decoded instruction. If the execute stage encounters a branch or jump instruction, it sets this flag during the rising clock edge. The decode stage will then discard its instruction during the falling clock edge. This prevents the need to stall the pipeline an extra cycle.

4.3. Execute

Calculation happens during the rising clock edge, and the computed result is written to the next buffer during the falling edge. The execute stage uses the opcode string to determine which operation to perform. Instructions are only executed by this stage if the *IDEX* buffer does not have a floating-point addition, subtraction, multiplication, or division instruction.

As a special case in this stage, branch and jump instructions are resolved, and the program counter is updated during the rising edge. This allows the fetch stage to read the next instruction during the falling edge. If a branch is taken, this stage also sets the flag to flush an already fetched instruction in the decode stage.

4.4. Memory Access

Memory is read or written during the rising clock edge, and the result is passed to the next buffer during the falling clock edge. As previously mentioned, all memory accesses in our simulation must be word-aligned. Thus, this stage always sets the lower two bits of the memory address to 0 before reading from or writing to memory.

Since different units take different numbers of cycles to complete, it is possible that two instructions need to access memory during the same cycle. This is so called a structural hazard, as our simulator only has one memory unit. To resolve this conflict, the memory unit looks at the start cycle time of each instruction and always picks the instruction with the earliest start time. The hazard detection unit is responsible for stalling the other pipeline stages when this conflict occurs.

4.5. Write-Back

Registers are written back during the rising clock edge so the decode stage can then read them during the falling edge. During the falling edge, this stage writes the results to a *pseudo-buffer* after the write-back stage which is needed by the dashboard.

4.6. Addition

Although we simulate the pipelined behavior of the floating-point adder, we do not attempt to simulate the intermediate pipelined computations for floating-point addition due to the limitations

of Node-RED. Rather, each addition stage simply reads the previous buffer during the rising edge, and then passes the data to the next buffer on the falling edge. The last stage then computes the actual result using the WebAssembly library (discussed later in Section 5). Instructions are only read into this stage if the *IDEX* buffer contains a floating-point addition instruction.

4.7. Multiplication

Just like addition, we do not attempt to simulate the intermediate pipelined computations for floating-point multiplication. Rather, each multiplication stage simply reads the previous buffer during the rising edge, and then passes the data to the next buffer on the falling edge. The last stage then computes the actual result using the WebAssembly library (discussed later in Section 5). Instructions are only read into this stage if the *IDEX* buffer contains a floating-point multiplication instruction.

4.8. Division

The division stage is a special case because it is not pipelined. Rather, it has an internal buffer along with a counter to track the number of remaining cycles. The counter is set when the instruction is first loaded on the rising edge, and then decremented on the falling edge of each clock cycle. When the counter reaches 0 on the falling edge, it computes the division result using the WebAssembly library and passes it to the next buffer. Instructions are only read into this stage if the *IDEX* buffer contains a floating-point division instruction.

5. WEB ASSEMBLY LIBRARY

As previously mentioned, Node-RED is based on the JavaScript programming language [5]. According to the JavaScript standard, the numeric type is always stored and represented as an IEEE 754 double-precision floating-point number. Therefore, when simulating our MIPS CPU, certain low-level operations are difficult if not impossible to implement in pure JavaScript code. For example, the instructions *lwc1*, *swc1*, *mtc1*, and *mfc1* reinterpret a 32-bit integer value as an IEEE 754 single-precision floating-point value. Another common operation is performing a sign-extension on a 16-bit value, which is used for immediate operations and branching. There is no JavaScript equivalent of such operations, requiring complicated algorithms to achieve the same result.

To address these issues, we have utilized WebAssembly [6] to write an additional library to handle these low-level operations. WebAssembly allows native code to be compiled and called from JavaScript, thus simplifying many low-level bitwise operations. Our library has been written using the Rust programming language [13] and compiled to WebAssembly using the wasm-pack [14] framework. Rust is chosen for its inclusion of many primitive low-level operations, strong support for WebAssembly, and guaranteed word-size (16-bit, 32-bit) and integer signedness. Once installed in Node-RED, the WebAssembly library provides low-level operations that can be called from all function nodes of a flow.

The WebAssembly library includes functions for the following operations:

- 32-bit addition and subtraction, including the result of the carry flag
- 32-bit bitwise operations, including *and*, *or*, *not*, *nor*, *exclusive or*, *shift logical left*, *shift logical right*, and *shift arithmetic right*
- Bit-casting between a 32-bit integer and an IEEE 754 single-precision floating-point number

- 16-bit signed integer to 32-bit signed integer sign extension
- Decoding the bit fields of 32-bit MIPS instruction (machine code)
- 32-bit integer string formatting in hexadecimal, signed decimal, and unsigned decimal
- IEEE 754 single-precision floating-point string formatting in hexadecimal and decimal

6. HAZARD DETECTION

The hazard detection unit is responsible for detecting any data and structural hazards and stalling the various pipeline stages to prevent such hazards. When executing instructions, the hazard detection unit must handle three types of hazards: read after write (RAW), write after write (WAW), and structural hazards. Examples of these three hazards can be seen in Figure 3. Although control hazards can occur due to branch instructions, this problem is handled by the *flush* flag (as previously discussed) and does not require the hazard detection unit. Write after read (WAR) hazards cannot occur in our simulation because the current register values are read and stored in the *IDEX* buffer, meaning later instructions will not invalidate the register values of earlier instructions.

To make discussion of the hazard detection algorithm with multi-cycle floating-point units concise, we define the following terms:

- **Execution Stages** - Stages in the pipeline responsible for the actual computation. This includes *Execute*, *Add 1* to *Add 4*, *Mul 1* to *Mul 7*, and *Div*.
  - **Beginning Execution Stages** - The first stage in the execution stages. This includes *Execute*, *Add 1*, *Mul 1*, and *Div*.
  - **Ending Execution Stages** - The last stage in the execution stages. This includes *Execute*, *Add 4*, *Mul 7*, and *Div*.
- **Intermediate Execution Buffers** - Buffers between the execution stages where the result cannot be forwarded. This includes *ADD12* to *ADD67*, *MUL12* to *MUL67*, and *DIVInternal*.
- **Incoming Memory Buffers** - Buffers between the execution stages and the Memory stage. This includes *EXMEM*, *ADD4MEM*, *MUL7MEM*, and *DIVMEM*.

We now discuss how the hazard detection unit handles all three types of hazards in the pipeline.

6.1. Read after Write (RAW) Hazards

RAW hazards occur when an instruction is dependent on the result of a prior instruction in the pipeline. If the result is inside an intermediate execution buffer then it cannot be forwarded, meaning the *Fetch*, *Decode*, and corresponding beginning execution stage must be stalled. We only need to stall the execution stage for the corresponding instruction type: *Add 1* for addition, *Mul 1* for multiplication, *Div* for division, and *Execute* for everything else.

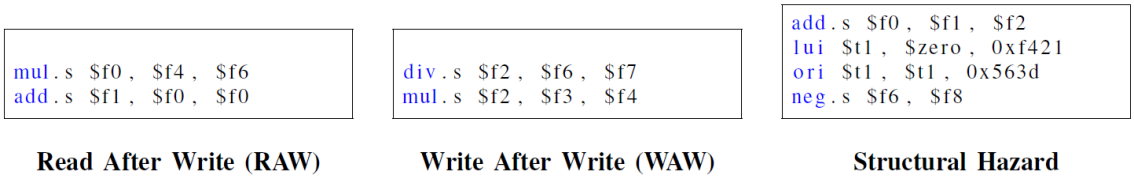

Figure 3. Examples of the three types of hazards that can occur in the pipeline

With the RAW hazard, the *add* instruction must stall until f0 can be forwarded from the *MUL7MEM* buffer. With the WAW hazard, both instructions will need to write to register f2. Although the *mul* instruction finishes before *div* (7 cycles vs 24), it must stall or else writes are out-of-order. With the structural hazard, both the *add* and *neg* instructions finish on the same cycle. The memory unit will write the addition result first because it has the earlier start cycle, so the *Execute* stage must be stalled.

To implement this algorithm, the hazard detection unit first checks to see if the *rs* or *rt* operands are updated by any later pipeline stages. If so, it next sees if the results can be forwarded from a later pipeline buffer. Forwarding can only occur from the incoming memory buffers or the *MEMWB* buffer. As a special case, values read from memory can only be forwarded from the *MEMWB* buffer (this is called a read-and-use hazard). If any of these values cannot be forwarded, then the hazard detection unit must stall the pipeline. Any writes to r0 are ignored when determining RAW hazards, as this register is always equal to 0 in MIPS.

6.2. Write After Write (WAW) Hazards

WAW hazards occur when multiple instructions attempt to write to the same register out-of-order. This problem can occur in our simulation because some instructions take longer to finish computation than others. Although the memory unit always picks the instruction with the earliest start time, this is not enough to ensure results are written in the correct order.

To prevent WAW hazards, the hazard detection unit must first check all of the incoming memory buffers. For each buffer, it tests if there is an instruction in one of the earlier execution stages that writes to the same register and has a lower start cycle. If one is found, then both the *Memory* stage and corresponding ending execution stage must be stalled. With the addition and multiplication units, the stall is propagated back along the execution stages in order, stopping if there is a *No-op* (*null*) in the pipeline. This allows pipelined computations to continue if they are not blocked by stalling the ending execution stage.

6.3. Structural Hazards

There are two types of structural hazards that can occur within the pipeline. The first structural hazard can occur with the division unit, which is not pipelined. If another division instruction attempts to execute while the division stage is busy, then both the *Fetch* and *Decode* stages must be stalled until the division unit is available.

The second structural hazard that can occur is when two or more instructions need to use the memory unit on the same cycle. During the *Memory* stage, the memory unit looks at the start cycle time of each instruction and always picks the instruction with the earliest start time. The other unpicked ending execution stages need to be stalled so the memory unit can handle their results on future cycles. Just like with WAW hazards, the stall is propagated back along the addition and multiplication execution stages in order, stopping only if there is a *No-op* (*null*) in the pipeline. This allows pipelined computations to continue if they are not blocked when the ending execution stage is stalled.

7. FORWARDING UNIT

By relying on the hazard detection unit, the pipeline greatly simplifies forwarding by implementing a naïve algorithm. The forwarding unit looks at the instruction currently in the *IDEX* buffer and sees if the *rs* or *rt* register matches the *rd* register of a later pipeline stage. It

only looks at the incoming memory buffers and the *MEMWB* buffer, as these are the only buffers that can be forwarded. If multiple *rd* registers match, then the forwarding unit selects the one with the latest start cycle (which will be the most recent result). As a special case, values read from memory can only be forwarded from the *MEMWB* buffer.

With this approach, it may be possible that a value is forwarded too early (that is, a more recent value is still being computed by the pipeline). However, since the hazard detection unit stalls stages to avoid RAW hazards, we are guaranteed that the *IDEX* buffer not be read until all source operands are available. Therefore, the correct value will get forwarded on a later cycle.

8. DASHBOARD INTERFACE

Our Node-RED simulator includes a visual dashboard interface for interacting with the CPU simulation. A screenshot of the dashboard is shown in Figure 4. The dashboard interface makes it easy to run machine-code files on our MIPS CPU simulator and see that the pipeline is working correctly.

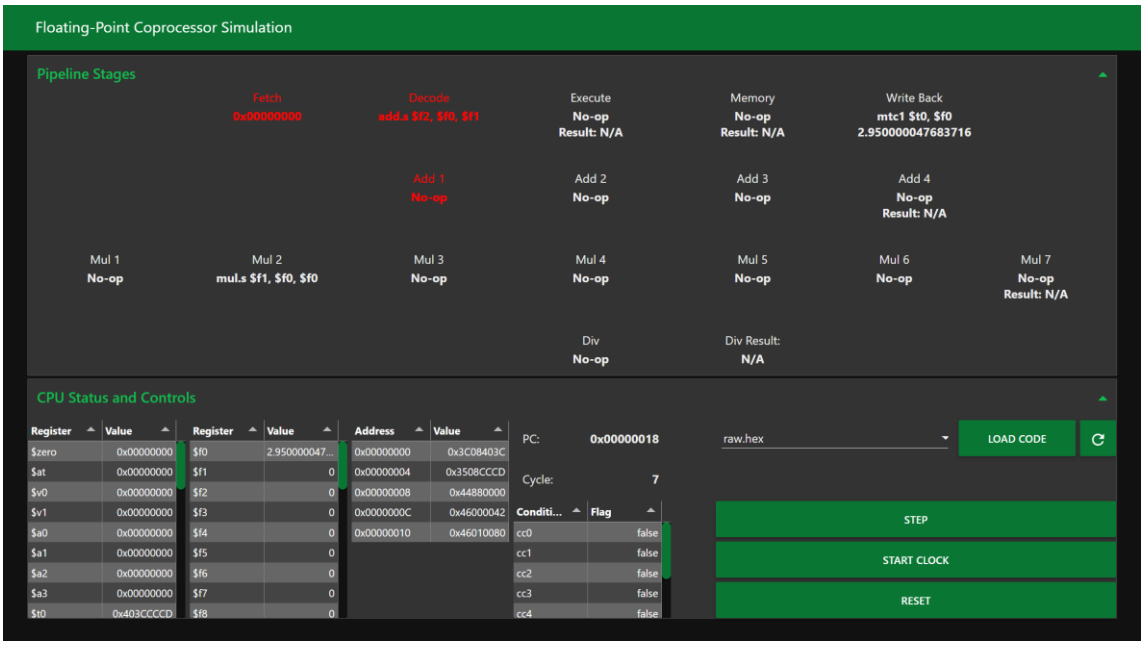

Figure 4. Screenshot of the Node-RED dashboard interface executing MIPS machine code.

The top part of the dashboard has labels that show the instructions being executed at every stage in the pipeline. A stage not being used on the current cycle (*null* entry in the flow object) is labeled with a *No-op* instruction. The ending execution stages (*Execute*, *Add 4*, *Mul 7*, and *Div*), the *Memory* stage, and *Write-Back* stage also show the computed result, if available, that will be passed to the next pipeline stage. The *Div* stage, which is not pipelined, shows the number of cycles remaining to finish the operation. Stages highlighted in red are being stalled during the current cycle due to a hazard detected by the hazard detection unit.

The bottom part of the dashboard shows additional details about the CPU status. This includes a table for the general-purpose registers, a table for the floating-point registers, and a table for the memory contents. The unused memory locations are not listed in the memory contents table. The dashboard also shows the program counter, current cycle number, and list of condition code flags.

The dashboard includes several control buttons for interacting with the simulator. For loading code into the dashboard, there is a dropdown that lists the provided sample machine-code files. Machine-code files are ASCII text documents placed inside the *machine-code* folder of the project directory with the *.hex* file extension. Each line of the *.hex* file should be an 8-digit big-endian hexadecimal instruction encoded using ASCII characters. Many MIPS assemblers can output machine code in this text format, including the MARS IDE (MIPS assembler and runtime simulator) [7] which was used to write the sample code for this study. Clicking "Load Code" button will parse the selected machine-code file line-by-line and store the 32-bit values into memory, starting at address 0. To the right is a refresh button (↻) that re-reads the contents of the folder and updates the dropdown list. This allows additional code files to be loaded into our simulator while Node-RED is running.

Below the dropdown and load buttons, the "Step" button is used to advance the simulation by a single clock cycle. There is also a "Start Clock" button, which begins automatic program execution when clicked. Clicking this button disables the "Load Code", "Step", and "Reset" buttons, which have their background changed to gray to indicate they are disabled. The text is also updated to say "Stop Clock". Clicking this button again will stop automatic program execution and re-enable the disabled buttons (also changing their backgrounds back to the original color). Finally, the "Reset" button is used to reset the CPU to the default status. This button clears all registers to 0, resets the program and cycle counters, sets all condition code flags to *false*, and re-loads the currently selected machine-code file back into memory.

9. CONCLUSION

In this article, we have presented our Node-RED-based simulation of the MIPS pipelined multi-cycle floating-point coprocessor instructions. By decoding MIPS machine code, our simulator can accurately execute real MIPS programs. Our implementation correctly handles hazard detection and forwarding in the pipeline to ensure program correctness. The visual dashboard makes it easy to load and run code on the simulator for testing purposes, as well as visually monitor instruction processing and inspect the contents of registers and memory.

The biggest limitation we see with our simulator at this point is the small number of supported floating-point instructions. Future work on this study could simulate additional floating-point instructions, such as square root, floor and ceiling operators, and integer to floating-point conversions. A more complex change would be supporting IEEE 754 double-precision operations, which would complicate the forwarding unit with two-register operands. Additionally, the simulator could be extended to support various branch prediction strategies to further improve performance.

We see this study as a helpful tool for future research when simulating pipelined MIPS multi-cycle floating-point functional units. Much of the JavaScript code has been written with flexibility in-mind, allowing the simulator to be easily extended to support new operations with minimal modifications. Even without such enhancements, our simulator is a powerful learning tool for executing complex sequences of pipelined and unpipelined instructions.

REFERENCES

[1] "MIPS32 architecture", https://www.mips.com/products/architectures/mips32-2/, Accessed in May 2022.
[2] "IEEE standard for floating-point arithmetic", (2008) IEEE Std 754-2008, pp1–70.
[3] "Node-RED: Low-code programming for event-driven applications", https://nodered.org/, Accessed in May 2022.

[4] "NodeJS." https://nodejs.org/, Accessed in May 2022.
[5] E. Ecma, (1999) "262: Ecmascript language specification", *ECMA (European Association for Standardizing Information and Communication Systems)*.
[6] A. Rossberg, "WebAssembly Core Specification", https://www.w3.org/TR/wasm-core-1/, Accessed in May 2022.
[7] K. Vollmar and P. Sanderson, (2006) "MARS: an education-oriented MIPS assembly language simulator," in *ACM's Special Interest Group on Computer Science Education (SIGCSE)*, vol. 6, pp239–243.
[8] D. A. Patterson and J. L. Hennessy, 2020, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*, ISBN-13: 978-0128201091, Morgan Kaufmann.
[9] S. Arif, (2015) VisUAL, a highly visual ARM simulator, https://salmanarif.bitbucket.io/visual/index.html, Accessed in September 2022.
[10] M. T. Kabir, M. T. Bari, and A. L. Haque, (2011) "ViSiMIPS: Visual simulator of MIPS32 pipelined processor," *6th International Conference on Computer Science & Education (ICCSE)*, pp788-793, doi: 10.1109/ICCSE.2011.6028756.
[11] M. T. Kabir, (2021) ViSiMIPS, https://github.com/tkr22777/visimips, Accessed in September 2022.
[12] E. Anderson, S. M. Abrar Jahin, N. Talukder, Y. Chu, and J. J. Lee, (2022) "A Node-RED-based MIPS-32 processor simulator," in *Advances in Data Computing, Communication and Security, Springer*, pp695-705.
[13] N. D. Matsakis & F. S. Klock II, (2014) "The Rust language", *ACM SIGAda Ada Letters*, vol. 34, pp103–104.
[14] rust-wasm Group, (2022) "wasm-pack", https://github.com/rustwasm/wasm-pack, Accessed in May 2022.

AUTHORS

Bryan McClain received his dual B.S. degree in computer science and applied mathematics from Indiana University–Purdue University Indianapolis (IUPUI) in December 2021. He is currently pursuing his M.S. degree in computer science at IUPUI and will graduate in December 2022. During his schooling, Bryan interned as a developer at several Indianapolis software companies, including Time Compression Strategies and Mimir. He is currently a part-time software engineer at OneCause and will begin his full-time employment in January 2023 after receiving his M.S. degree. Bryan's research interests include low-level hardware design, cryptography, compilers, and graph database theory.

Jinyu Fang received her B.S degree in Electrical Engineering from East China Jiaotong University in 2015. She worked as an assistant electrical engineer in a railway electric engineering company in China for four years. Jinyu is currently pursuing her M.S degree in computer and information science in IUPUI, meanwhile, she works as a research assistant studying reinforcement learning in stochastic games. Her research interest is machine learning and artificial intelligence.

Prathamesh Kale is currently pursuing MS degree in computer science at IUPUI.

John J. Lee received his MS and PhD degrees in Electrical and Computer Engineering from the Georgia Institute of Technology in 2003 and in 2004, respectively. He joined the faculty of the Department of Electrical and Computer Engineering, IUPUI in 2005 where he is currently an associate professor. His research interests include Internet of Things (IoT), blockchain technology, novel hardware-oriented, FPGA-assisted, or GPU-based parallel acceleration of algorithms and applications, and high-performance computing.