Project Early Implementation Report
Pipelining with Hazard Detection
CS2323 Computer Architecture

Anant Maheshwary (CS24BTECH11006)

October, 2025

Introduction

The topic chosen for the final project is "**Extend the in-house simulator to support pipelining and handle various hazards**," built for assembling a **RISCV-64I** assembly program. The following report discusses the scope of the project, the implementation steps and the verification plan for the project.

Design overview

C++ Components

1. **The Pipeline Registers and Control Signals:**
   The four pipeline registers are defined as `structs` in the `pipeline_registers.h` file (as all of their members need to be `public`).

```cpp
    struct ID_EX_Reg {
        bool is_valid = false;
        ControlSignals control;   // All control signals
        uint64_t pc = 0;           // the original pc
        uint64_t pc_inc = 0;          // the incremented pc

        // Data read from register file
        uint64_t rs1_data = 0;
        uint64_t rs2_data = 0;

        // Immediate value
        int32_t immediate = 0;

        // Register indices
        uint8_t rs1_index = 0;
        uint8_t rs2_index = 0;
        uint8_t rd_index = 0;
    };

```

Listing 1: Example Pipeline Register Definition

The control signals are declared as a struct, `ControlSignals`. The control signals generated in the ID stage are encapsulated in this struct and then propagated through the later registers, each of which has a `ControlSignals` member.

```
 struct ControlSignals {
     // EX Stage Controls
     alu::AluOp alu_op = alu::AluOp::kNone;
     bool alu_src_b = false;     // ALU src2: true=Imm, false=rs2
     instruction_type::AluSrcA alu_src_a;  // a variable of AluSrcA enum to handle lui, auipc

     // MEM Stage Controls
     bool mem_read = false;
     bool mem_write = false;
     // enum variables to distinguish between different instructions
     instruction_type::MemReadOp mem_read_op = instruction_type::MemReadOp::MEM_READ_NONE;
     instruction_type::MemWriteOp mem_write_op = instruction_type::MemWriteOp::MEM_WRITE_NONE;

     bool branch = false;
     // enum variables to distinguish between different branch instructions
     instruction_type::BranchOp branch_op = instruction_type::BranchOp::B_NONE;

     // WB Stage Controls
     bool reg_write = false;
     bool mem_to_reg = false;
     // enum variables to distinguish between different write back sources
     instruction_type::WriteBackSrc wb_src = instruction_type::WB_NONE;

     // Additional Contrls
     bool is_csr = false;
     bool is_syscall = false;
     bool is_nop = true;
 };

```

Listing 2: ControlSignals Struct Definition

2. **RV5S Virtual Machine:** The `RV5SVM` class is extended from `VmBase` - a composition of the base components: `Memory`, `ALU`, `Registers`, etc.
   The `RV5SVM` class provides the implementation for functions like `Step()`, `Run()`, `DebugRun()`, etc.

3. **RV5S Control Unit:** A key design choice was having the `RV5SVM` control unit not be inherited from the `VmBase` control unit (unlike the original implementation for

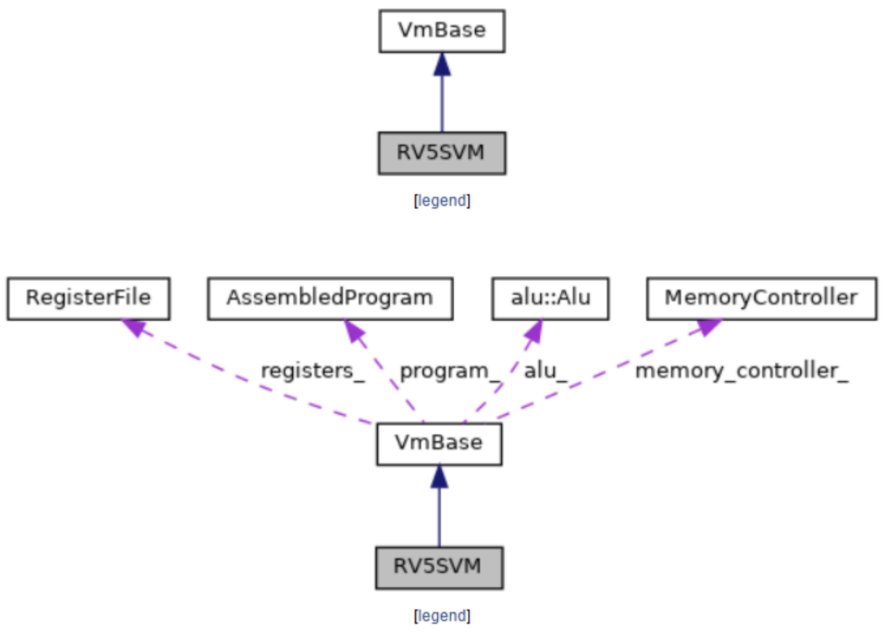


Figure 1: Class dependency diagram generated using Doxygen.

the `RVSSVM` control unit).

- The original control unit (`RVSSControlUnit`) for the single-stage VM was **stateful**. It inherited from a base class, stored control signals internally (set via `SetControlSignals` during Decode), and provided getter methods for later stages (Execute, Memory, WriteBack) to retrieve these stored signals within the same clock cycle.
- In contrast, the new `RV5SControlUnit` for the 5-stage pipeline is **stateless**. It does not inherit from the old base class. Its primary function, `getControlSignals`, takes the instruction, immediately calculates all required control signals, and returns them packaged in a `ControlSignals` struct. This struct is then explicitly passed down through the pipeline registers (`IF_ID_Reg`, `ID_EX_Reg`, etc.) from one stage to the next.

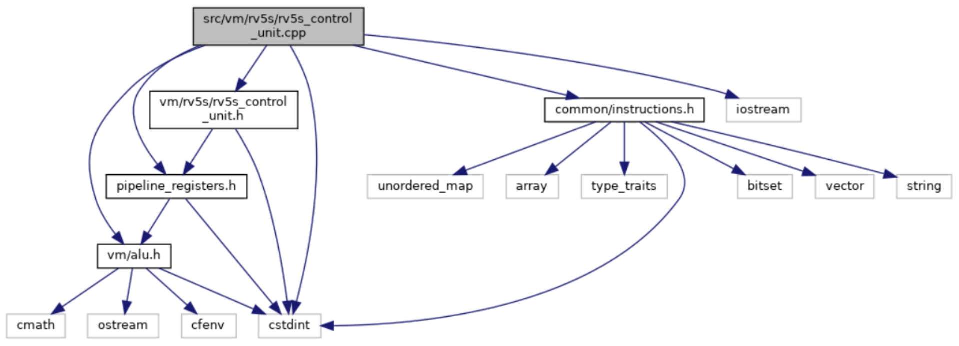


Figure 2: Class dependency diagram generated using Doxygen.

Datapath and Control Flow for Pipelining

The program tries to emulate the actual hardware as much as possible.

- **Reverse-Order Execution:** The `Step()` function simulates a single clock cycle. Crucially, it calls the pipeline stage functions in reverse order (`WriteBack_Stage()` first, `Fetch_Stage()` last). This models the fact that in a real pipeline, all stages operate concurrently on different instructions, and the results from a previous cycle's stage are available at the start of the current cycle's next stage.

- **State Transfer:** The `RV5SVM` class has two members corresponding to each pipeline register: one for storing the state at the beginning of a cycle and the other to store the state at the completion of the current clock cycle (e.g., `id_ex_reg_` and `next_id_ex_reg_`).
  At the end of each `Step()`, the main pipeline registers (`if_id_reg_`, etc.) are updated with the contents of the temporary `‘next_‘` registers, advancing each instruction one stage down the pipeline.

- **Stage Independence:** As a design choice, the later pipeline registers may have some extra controls in their `ControlSignals` struct that may not be required at that point in the pipeline (e.g., the `WB` stage does not require `AluOp`). This simplifies the software implementation so that the control signals can be copied easily from one pipeline register to the other.

  However, the **VM** is coded in a way such that each stage fetches and stores information from its adjacent pipeline registers only and uses only those controls that it would access in actual hardware.
  For instance, the `EX` stage does not use the opcode to distinguish the type of load instruction (`lb`, `lh`, `ld`, etc.), store instruction (`sb`, `sh`, `sd`, etc.), or branch instruction (`blt`, `bge`, etc.). All decoding steps occur only in the Decode stage, and the instruction types are propagated as control signals to the subsequent registers.

  Figure 3: code photos of op enums.

- **Bubble Injection:** The `CreateBubble<T>()` function is implemented for handling invalid states: It handles the start of pipelining where first instructions of the program are propagating through the pipeline and is then used towards the end of the program to drain the remaining instructions from the pipeline.

Implementation Status

The current project version has full support for the **Pipeline Mode-1 (without hazard detection)** along with **breakpoint handling** and **debugging**. It can execute programs without data and control hazards correctly, along with the ability to look at the `VM State` at any point in the program.
It also provides functionality for **NOP** and **Bubble Injection** to be used in later modes.
The project files can be found at `https://github.com/DWBH21/riscv-simulator-pipelined`.

RV5S_VM Functions

The current program has support for the following execution features:

- **`Step()`:** In the single-stage implementation, a `Step()` represented stepping through an entire instruction. In the new `RV5SVM`, the `Step()` function represents **one clock tick** and causes the pipeline to execute one clock cycle (all five stages execute once).
- **`DebugRun()`:** Executes the loaded file, considering breakpoints and with a delay between steps (`run_step_delay`).
- **`Run()`:** Executes the loaded file, without considering breakpoints and with no delay between steps.

The functions mentioned above enable debugging of the program: stepping through the clock cycles one by one and looking at the `vm_state`.

Dump State Functions

Functions that enable analysis of the `vm_state` (to be used for verification):

- **DumpPipeLineRegisters():** Dumps the state of all **5-stage pipeline registers** into a `JSON` file.
- **DumpState():** 5-stage specific implementation. In addition to the general VM state, it dumps additional members: `number of cycles`, `CPI`, `IPC`, and `pipeline_drain_counter` to a `JSON` file. This will also hold information about execution statistics in later pipelining modes.
- **DumpMemory():** Dumps the memory contents for each specified address into a `JSON` file (borrowed from the single-stage implementation).

Config Options

The `config` and `main.cpp` files were changed to allow switching from one pipeline mode to another.

This can be done by running the command:

```
modify_config Execution processor_type multi_stage
```

The `main.cpp` file has the code:

```cpp
std::unique_ptr<VmBase> vm_ptr; // Pointer to the vm_base class
if (vm_config::config.getVmType() == vm_config::VmTypes::
    SINGLE_STAGE) {
  std::cout << "Initializing Single-Stage vm_ptr->.." << std::endl;
        vm_ptr = std::make_unique<RVSSVM>();
} else {
        std::cout << "Initializing 5-Stage Pipeline vm_ptr->.." <<
    std::endl;
        vm_ptr = std::make_unique<RV5SVM>();
}
```

Listing 3: Runtime VM Selection in main.cpp

This allows the type of VM (`RVSSVM` or `RV5SVM`) to be selected at **runtime** using **dynamic polymorphism**.

Testing

The 5-stage pipeline has been tested extensively for programs that don't contain **control or data hazards**. These **unit tests** include programs containing single instructions as well as combinations of instructions (across all types of instructions).

Verification was done using a script file which compared the final output stored in the 5-stage `vm_state` files: `memory_dump.json`, `registers_dump.json`, and `vm_state_dump.json` with the corresponding output of the single-stage processor (mode 0) as a reference.

However, the intermediate pipeline states (the pipeline registers) were checked manually only for a handful of programs, using the **Ripes simulator** as reference. Methods for automated testing of intermediate states are being explored.

Execution of an Example Program

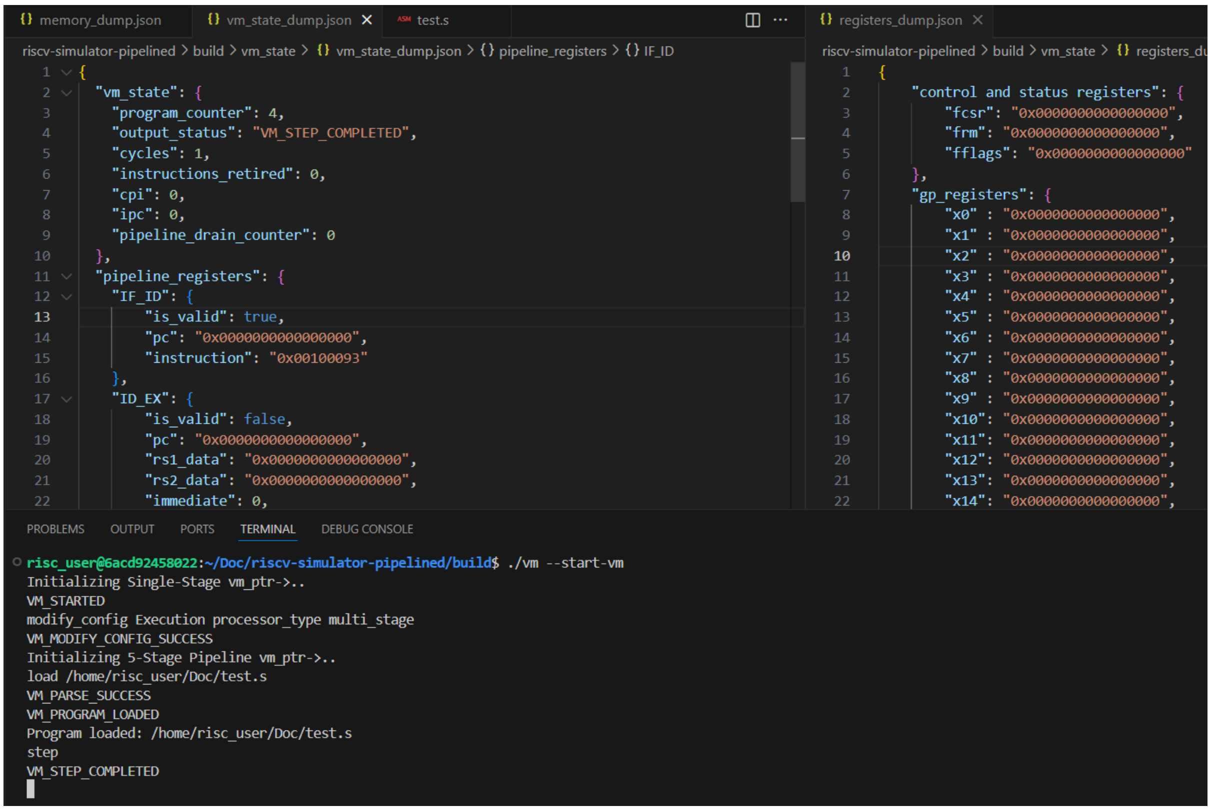
The following are screenshots of the `registers.json` and `vm_state.json` for a simple program whose execution can be tracked easily.

```
    .data
    .dword 2

    .text
        addi x1, x0, 1
        addi x2, x0, 2
        addi x3, x0, 3
        addi x4, x0, 4
        addi x5, x0, 5
        addi x6, x0, 6
        addi x7, x0, 7

```

Listing 4: test.s

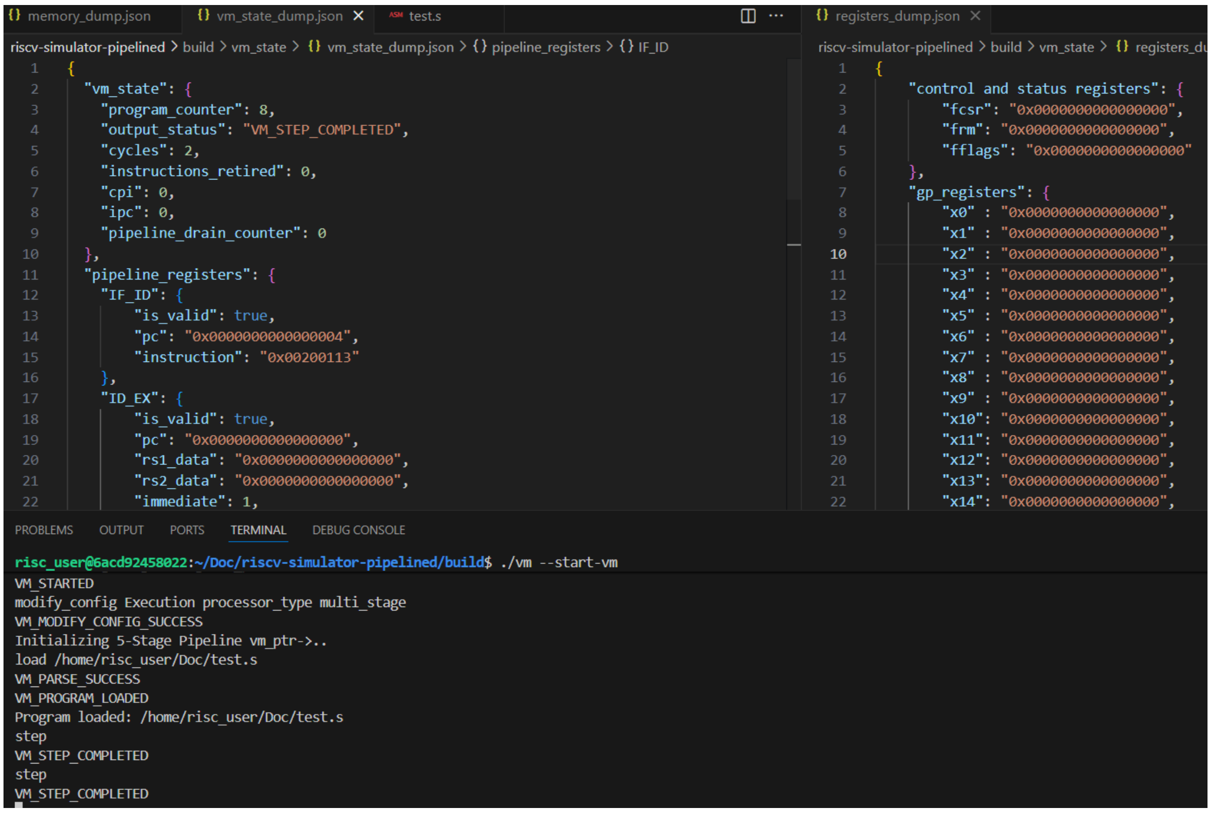

memory_dump.json
vm_state_dump.json
test.s
registers_dump.json
riscv-simulator-pipelined > build > vm_state > vm_state_dump.json > pipeline_registers > IF_ID
riscv-simulator-pipelined > build > vm_state > registers_du
PROBLEMS
OUTPUT
PORTS
TERMINAL
DEBUG CONSOLE

```
{
  "vm_state": {
    "program_counter": 4,
    "output_status": "VM_STEP_COMPLETED",
    "cycles": 1,
    "instructions_retired": 0,
    "cpi": 0,
    "ipc": 0,
    "pipeline_drain_counter": 0
  },
  "pipeline_registers": {
    "IF_ID": {
        "is_valid": true,
        "pc": "0x0000000000000000",
        "instruction": "0x00100093"
    },
    "ID_EX": {
        "is_valid": false,
        "pc": "0x0000000000000000",
        "rs1_data": "0x0000000000000000",
        "rs2_data": "0x0000000000000000",
        "immediate": 0,
```

```
{
    "control and status registers": {
        "fcsr": "0x0000000000000000",
        "frm": "0x0000000000000000",
        "fflags": "0x0000000000000000"
    },
    "gp_registers": {
        "x0" : "0x0000000000000000",
        "x1" : "0x0000000000000000",
        "x2" : "0x0000000000000000",
        "x3" : "0x0000000000000000",
        "x4" : "0x0000000000000000",
        "x5" : "0x0000000000000000",
        "x6" : "0x0000000000000000",
        "x7" : "0x0000000000000000",
        "x8" : "0x0000000000000000",
        "x9" : "0x0000000000000000",
        "x10": "0x0000000000000000",
        "x11": "0x0000000000000000",
        "x12": "0x0000000000000000",
        "x13": "0x0000000000000000",
        "x14": "0x0000000000000000",
```

```
risc_user@6acd92458022:~/Doc/riscv-simulator-pipelined/build$ ./vm --start-vm
Initializing Single-Stage vm_ptr->..
VM_STARTED
modify_config Execution processor_type multi_stage
VM_MODIFY_CONFIG_SUCCESS
Initializing 5-Stage Pipeline vm_ptr->..
load /home/risc_user/Doc/test.s
VM_PARSE_SUCCESS
VM_PROGRAM_LOADED
Program loaded: /home/risc_user/Doc/test.s
step
VM_STEP_COMPLETED
```

Figure 4: `vm_state` files for Cycle 1

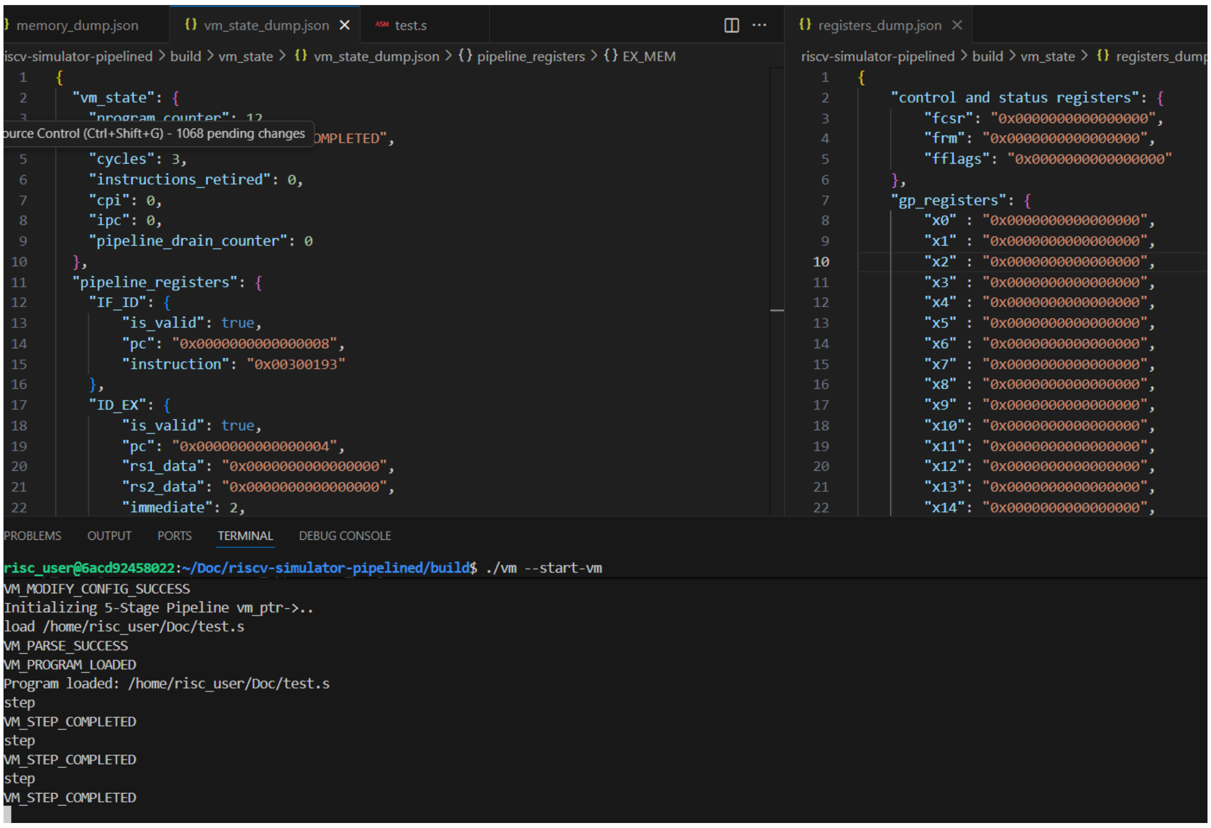

memory_dump.json
vm_state_dump.json
test.s
registers_dump.json
riscv-simulator-pipelined > build > vm_state > vm_state_dump.json > pipeline_registers > IF_ID
riscv-simulator-pipelined > build > vm_state > registers_du
PROBLEMS
OUTPUT
PORTS
TERMINAL
DEBUG CONSOLE

```
{
  "vm_state": {
    "program_counter": 8,
    "output_status": "VM_STEP_COMPLETED",
    "cycles": 2,
    "instructions_retired": 0,
    "cpi": 0,
    "ipc": 0,
    "pipeline_drain_counter": 0
  },
  "pipeline_registers": {
    "IF_ID": {
        "is_valid": true,
        "pc": "0x0000000000000004",
        "instruction": "0x00200113"
    },
    "ID_EX": {
        "is_valid": true,
        "pc": "0x0000000000000000",
        "rs1_data": "0x0000000000000000",
        "rs2_data": "0x0000000000000000",
        "immediate": 1,
```

```
{
    "control and status registers": {
        "fcsr": "0x0000000000000000",
        "frm": "0x0000000000000000",
        "fflags": "0x0000000000000000"
    },
    "gp_registers": {
        "x0" : "0x0000000000000000",
        "x1" : "0x0000000000000000",
        "x2" : "0x0000000000000000",
        "x3" : "0x0000000000000000",
        "x4" : "0x0000000000000000",
        "x5" : "0x0000000000000000",
        "x6" : "0x0000000000000000",
        "x7" : "0x0000000000000000",
        "x8" : "0x0000000000000000",
        "x9" : "0x0000000000000000",
        "x10": "0x0000000000000000",
        "x11": "0x0000000000000000",
        "x12": "0x0000000000000000",
        "x13": "0x0000000000000000",
        "x14": "0x0000000000000000",
```

```
risc_user@6acd92458022:~/Doc/riscv-simulator-pipelined/build$ ./vm --start-vm
VM_STARTED
modify_config Execution processor_type multi_stage
VM_MODIFY_CONFIG_SUCCESS
Initializing 5-Stage Pipeline vm_ptr->..
load /home/risc_user/Doc/test.s
VM_PARSE_SUCCESS
VM_PROGRAM_LOADED
Program loaded: /home/risc_user/Doc/test.s
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
```

Figure 5: `vm_state` files for Cycle 2



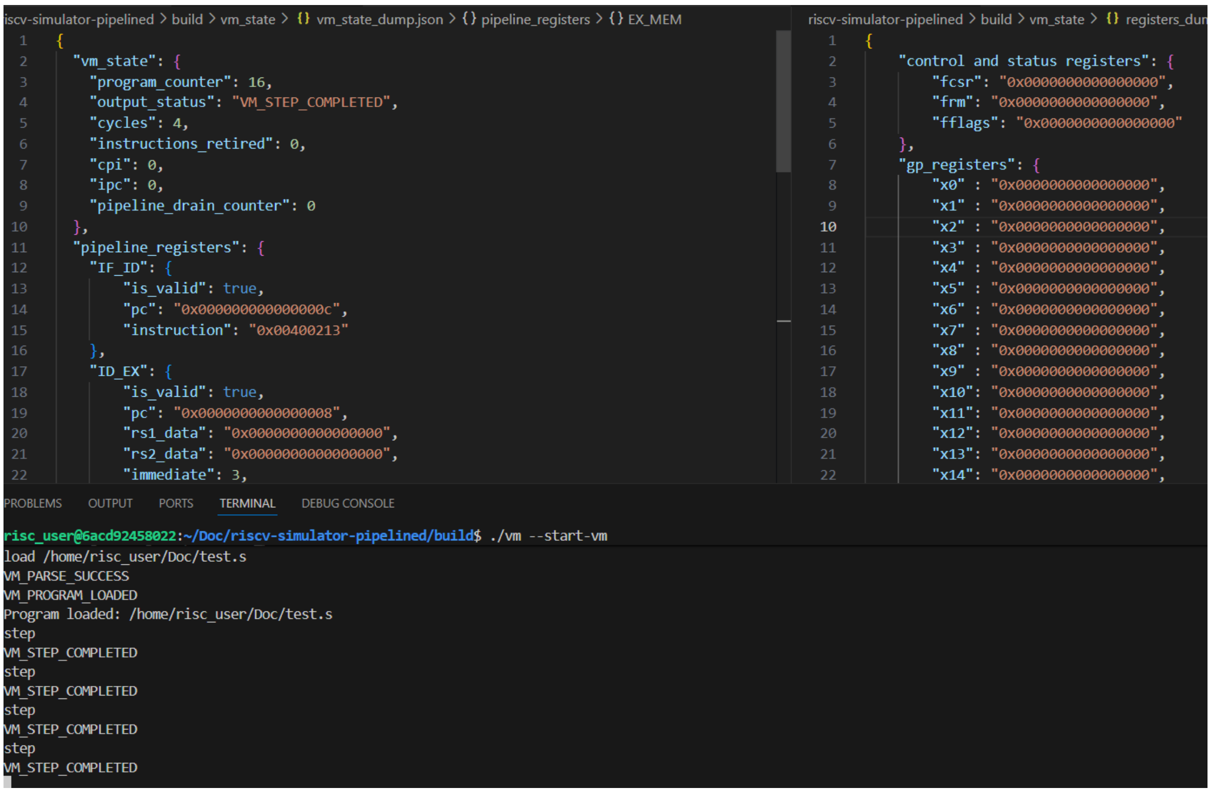
} memory_dump.json
{} vm_state_dump.json ×
test.s
{} registers_dump.json ×
iscv-simulator-pipelined > build > vm_state > {} vm_state_dump.json > {} pipeline_registers > {} EX_MEM
riscv-simulator-pipelined > build > vm_state > {} registers_dump
ource Control (Ctrl+Shift+G) - 1068 pending changes
PROBLEMS
OUTPUT
PORTS
TERMINAL
DEBUG CONSOLE

```
{
  "vm_state": {
    "program_counter": 12,
                                 OMPLETED",
    "cycles": 3,
    "instructions_retired": 0,
    "cpi": 0,
    "ipc": 0,
    "pipeline_drain_counter": 0
  },
  "pipeline_registers": {
    "IF_ID": {
      "is_valid": true,
      "pc": "0x0000000000000008",
      "instruction": "0x00300193"
    },
    "ID_EX": {
      "is_valid": true,
      "pc": "0x0000000000000004",
      "rs1_data": "0x0000000000000000",
      "rs2_data": "0x0000000000000000",
      "immediate": 2,
```

```
{
    "control and status registers": {
      "fcsr": "0x0000000000000000",
      "frm": "0x0000000000000000",
      "fflags": "0x0000000000000000"
    },
    "gp_registers": {
      "x0" : "0x0000000000000000",
      "x1" : "0x0000000000000000",
      "x2" : "0x0000000000000000",
      "x3" : "0x0000000000000000",
      "x4" : "0x0000000000000000",
      "x5" : "0x0000000000000000",
      "x6" : "0x0000000000000000",
      "x7" : "0x0000000000000000",
      "x8" : "0x0000000000000000",
      "x9" : "0x0000000000000000",
      "x10": "0x0000000000000000",
      "x11": "0x0000000000000000",
      "x12": "0x0000000000000000",
      "x13": "0x0000000000000000",
      "x14": "0x0000000000000000",
```

```
risc_user@6acd92458022:~/Doc/riscv-simulator-pipelined/build$ ./vm --start-vm
M_MODIFY_CONFIG_SUCCESS
Initializing 5-Stage Pipeline vm_ptr->..
load /home/risc_user/Doc/test.s
M_PARSE_SUCCESS
M_PROGRAM_LOADED
Program loaded: /home/risc_user/Doc/test.s
step
M_STEP_COMPLETED
step
M_STEP_COMPLETED
step
M_STEP_COMPLETED
```

Figure 6: `vm_state` files for Cycle 3



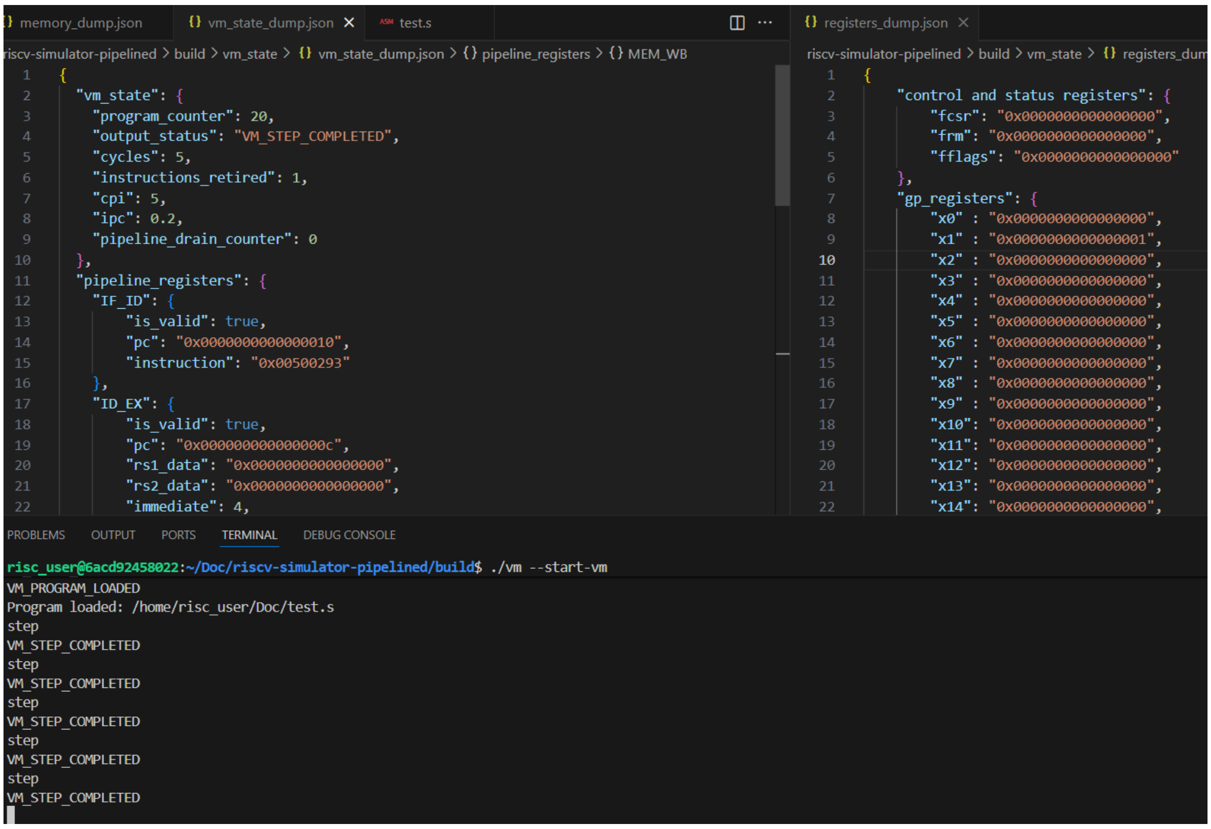
iscv-simulator-pipelined > build > vm_state > {} vm_state_dump.json > {} pipeline_registers > {} EX_MEM
riscv-simulator-pipelined > build > vm_state > {} registers_dun
PROBLEMS
OUTPUT
PORTS
TERMINAL
DEBUG CONSOLE

```
{
  "vm_state": {
    "program_counter": 16,
    "output_status": "VM_STEP_COMPLETED",
    "cycles": 4,
    "instructions_retired": 0,
    "cpi": 0,
    "ipc": 0,
    "pipeline_drain_counter": 0
  },
  "pipeline_registers": {
    "IF_ID": {
      "is_valid": true,
      "pc": "0x000000000000000c",
      "instruction": "0x00400213"
    },
    "ID_EX": {
      "is_valid": true,
      "pc": "0x0000000000000008",
      "rs1_data": "0x0000000000000000",
      "rs2_data": "0x0000000000000000",
      "immediate": 3,
```

```
{
    "control and status registers": {
      "fcsr": "0x0000000000000000",
      "frm": "0x0000000000000000",
      "fflags": "0x0000000000000000"
    },
    "gp_registers": {
      "x0" : "0x0000000000000000",
      "x1" : "0x0000000000000000",
      "x2" : "0x0000000000000000",
      "x3" : "0x0000000000000000",
      "x4" : "0x0000000000000000",
      "x5" : "0x0000000000000000",
      "x6" : "0x0000000000000000",
      "x7" : "0x0000000000000000",
      "x8" : "0x0000000000000000",
      "x9" : "0x0000000000000000",
      "x10": "0x0000000000000000",
      "x11": "0x0000000000000000",
      "x12": "0x0000000000000000",
      "x13": "0x0000000000000000",
      "x14": "0x0000000000000000",
```

```
risc_user@6acd92458022:~/Doc/riscv-simulator-pipelined/build$ ./vm --start-vm
load /home/risc_user/Doc/test.s
M_PARSE_SUCCESS
M_PROGRAM_LOADED
Program loaded: /home/risc_user/Doc/test.s
step
M_STEP_COMPLETED
step
M_STEP_COMPLETED
step
M_STEP_COMPLETED
step
M_STEP_COMPLETED
```

Figure 7: `vm_state` files for Cycle 4

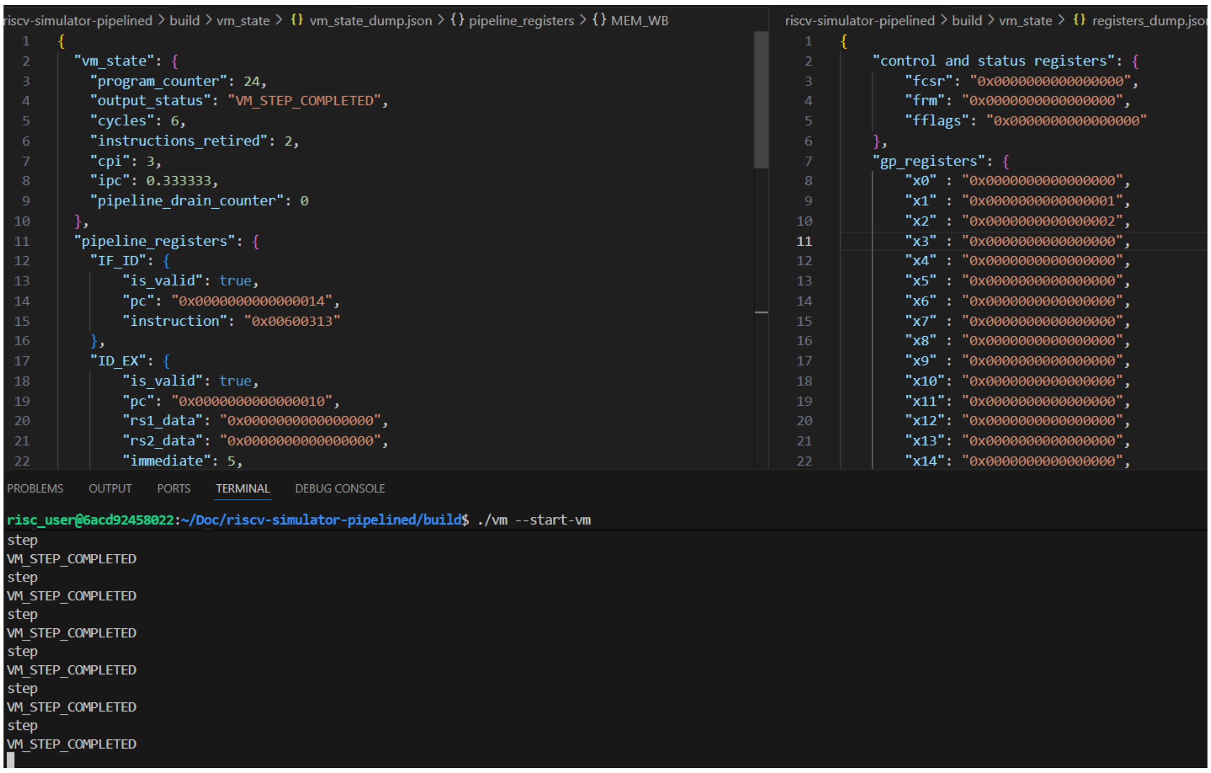

memory_dump.json
vm_state_dump.json
test.s
registers_dump.json
riscv-simulator-pipelined > build > vm_state > vm_state_dump.json > pipeline_registers > MEM_WB
riscv-simulator-pipelined > build > vm_state > registers_dum
PROBLEMS
OUTPUT
PORTS
TERMINAL
DEBUG CONSOLE

```json
{
  "vm_state": {
    "program_counter": 20,
    "output_status": "VM_STEP_COMPLETED",
    "cycles": 5,
    "instructions_retired": 1,
    "cpi": 5,
    "ipc": 0.2,
    "pipeline_drain_counter": 0
  },
  "pipeline_registers": {
    "IF_ID": {
        "is_valid": true,
        "pc": "0x0000000000000010",
        "instruction": "0x00500293"
    },
    "ID_EX": {
        "is_valid": true,
        "pc": "0x000000000000000c",
        "rs1_data": "0x0000000000000000",
        "rs2_data": "0x0000000000000000",
        "immediate": 4,
```

```json
{
    "control and status registers": {
        "fcsr": "0x0000000000000000",
        "frm": "0x0000000000000000",
        "fflags": "0x0000000000000000"
    },
    "gp_registers": {
        "x0" : "0x0000000000000000",
        "x1" : "0x0000000000000001",
        "x2" : "0x0000000000000000",
        "x3" : "0x0000000000000000",
        "x4" : "0x0000000000000000",
        "x5" : "0x0000000000000000",
        "x6" : "0x0000000000000000",
        "x7" : "0x0000000000000000",
        "x8" : "0x0000000000000000",
        "x9" : "0x0000000000000000",
        "x10": "0x0000000000000000",
        "x11": "0x0000000000000000",
        "x12": "0x0000000000000000",
        "x13": "0x0000000000000000",
        "x14": "0x0000000000000000",
```

```
risc_user@6acd92458022:~/Doc/riscv-simulator-pipelined/build$ ./vm --start-vm
VM_PROGRAM_LOADED
Program loaded: /home/risc_user/Doc/test.s
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
```

Figure 8: `vm_state` files for Cycle 5


riscv-simulator-pipelined > build > vm_state > vm_state_dump.json > pipeline_registers > MEM_WB
riscv-simulator-pipelined > build > vm_state > registers_dump.jso
PROBLEMS
OUTPUT
PORTS
TERMINAL
DEBUG CONSOLE

```json
{
  "vm_state": {
    "program_counter": 24,
    "output_status": "VM_STEP_COMPLETED",
    "cycles": 6,
    "instructions_retired": 2,
    "cpi": 3,
    "ipc": 0.333333,
    "pipeline_drain_counter": 0
  },
  "pipeline_registers": {
    "IF_ID": {
        "is_valid": true,
        "pc": "0x0000000000000014",
        "instruction": "0x00600313"
    },
    "ID_EX": {
        "is_valid": true,
        "pc": "0x0000000000000010",
        "rs1_data": "0x0000000000000000",
        "rs2_data": "0x0000000000000000",
        "immediate": 5,
```

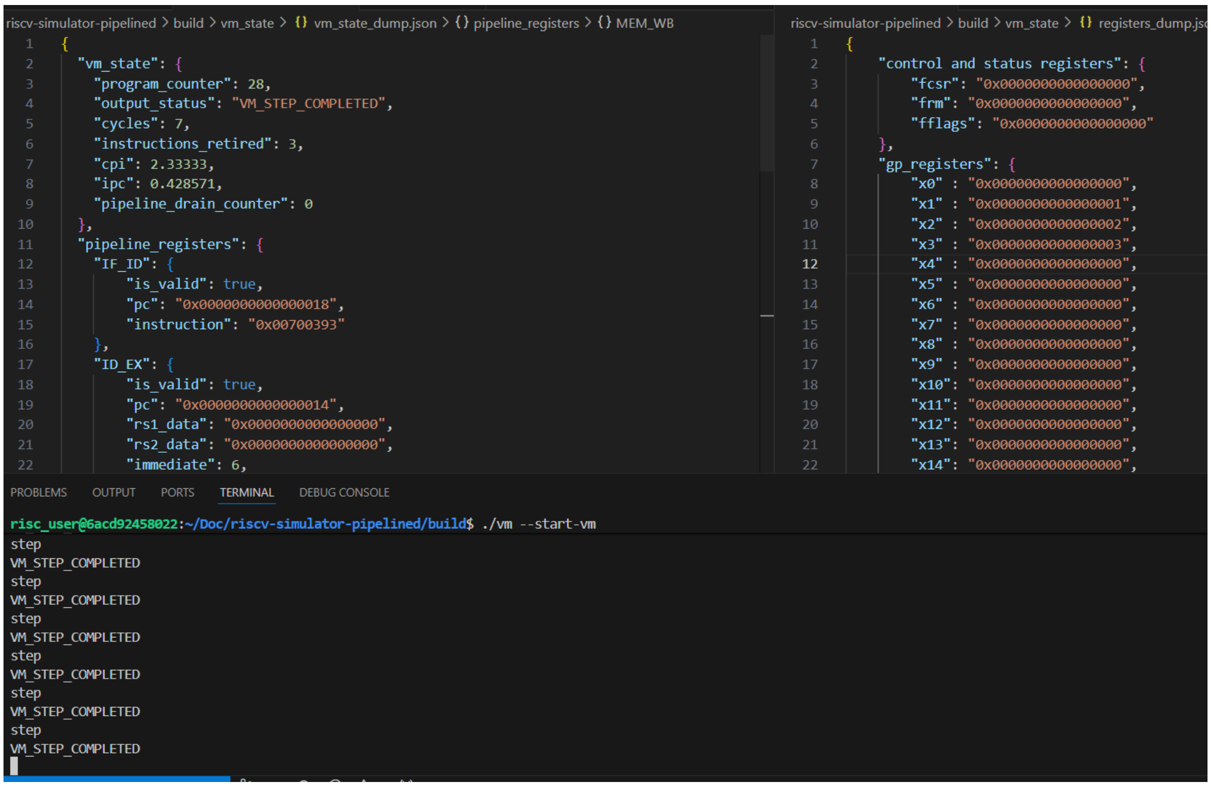
```json
{
    "control and status registers": {
        "fcsr": "0x0000000000000000",
        "frm": "0x0000000000000000",
        "fflags": "0x0000000000000000"
    },
    "gp_registers": {
        "x0" : "0x0000000000000000",
        "x1" : "0x0000000000000001",
        "x2" : "0x0000000000000002",
        "x3" : "0x0000000000000000",
        "x4" : "0x0000000000000000",
        "x5" : "0x0000000000000000",
        "x6" : "0x0000000000000000",
        "x7" : "0x0000000000000000",
        "x8" : "0x0000000000000000",
        "x9" : "0x0000000000000000",
        "x10": "0x0000000000000000",
        "x11": "0x0000000000000000",
        "x12": "0x0000000000000000",
        "x13": "0x0000000000000000",
        "x14": "0x0000000000000000",
```

```
risc_user@6acd92458022:~/Doc/riscv-simulator-pipelined/build$ ./vm --start-vm
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
```

Figure 9: `vm_state` files for Cycle 6

Challenges Faced

1. **Execution of M and F Extension Instructions:** The single-stage implementation treats floating-point, multiplication and division instructions as normal


riscv-simulator-pipelined > build > vm_state > {} vm_state_dump.json > {} pipeline_registers > {} MEM_WB
riscv-simulator-pipelined > build > vm_state > {} registers_dump.js
PROBLEMS
OUTPUT
PORTS
TERMINAL
DEBUG CONSOLE

```json
{
  "vm_state": {
    "program_counter": 28,
    "output_status": "VM_STEP_COMPLETED",
    "cycles": 7,
    "instructions_retired": 3,
    "cpi": 2.33333,
    "ipc": 0.428571,
    "pipeline_drain_counter": 0
  },
  "pipeline_registers": {
    "IF_ID": {
        "is_valid": true,
        "pc": "0x0000000000000018",
        "instruction": "0x00700393"
    },
    "ID_EX": {
        "is_valid": true,
        "pc": "0x0000000000000014",
        "rs1_data": "0x0000000000000000",
        "rs2_data": "0x0000000000000000",
        "immediate": 6,
```

```json
{
    "control and status registers": {
        "fcsr": "0x0000000000000000",
        "frm": "0x0000000000000000",
        "fflags": "0x0000000000000000"
    },
    "gp_registers": {
        "x0" : "0x0000000000000000",
        "x1" : "0x0000000000000001",
        "x2" : "0x0000000000000002",
        "x3" : "0x0000000000000003",
        "x4" : "0x0000000000000000",
        "x5" : "0x0000000000000000",
        "x6" : "0x0000000000000000",
        "x7" : "0x0000000000000000",
        "x8" : "0x0000000000000000",
        "x9" : "0x0000000000000000",
        "x10": "0x0000000000000000",
        "x11": "0x0000000000000000",
        "x12": "0x0000000000000000",
        "x13": "0x0000000000000000",
        "x14": "0x0000000000000000",
```

```
risc_user@6acd92458022:~/Doc/riscv-simulator-pipelined/build$ ./vm --start-vm
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
step
VM_STEP_COMPLETED
```

Figure 10: `vm_state` files for Cycle 7

arithmetic instructions executed by the `ALU`. However, in real hardware, they have their own pipelined implementations so as be fast enough to fit within a clock cycle.

- **Resolution:**
  - The current implementation deviates from actual hardware and assumes **M-Instructions** are executed in a **single cycle** in the `EX` stage, just like the other arithmetic instructions.
  - For **floating-point instructions**, however, many changes would have to be made to the pipelined datapath (`fp` register file, extra read ports in the Decode stage, `MUXes` in the EX stage, and write-back ports to the `fp` registers) and control signals to distinguish these instructions.

    The current implementation disables the assembling of FP instructions altogether in a program by modifying the `config` file.
    Support for pipelining of floating-point instructions might be explored later.

2. **System Calls and CSR instructions** are decoded in the program but not executed (basically treated as `NOPs`).

3. **Explored Multithreading:** The current program is inefficient and deviates from actual hardware in the sense that the 5 stages, though simulating pipeline behavior, are not actually being executed in parallel.
   In theory, these stages, being independent of each other, could be multi-threaded with a **Producer-Consumer pattern**, with the preceding stage acting as the

producer, the current stage as the consumer, and the pipeline registers acting as synchronized buffers between these threads.

- **Resolution:** Although multithreading enables parallel execution, the execution time for each stage is **dynamic**, which does not map well to a **fixed-frequency clock** in a CPU. The **buffer size** would have to be limited to a single instruction at a time, defeating the purpose of multithreading.
  A more complicated design pattern involving the clock as a separate thread and having all stage threads synchronized to this common clock will also not work, as there is no way to determine the **maximum stage delay** in software.

4. **Undo/Redo:** Undo/Redo functionality in breakpoint handling is very complex for the 5-stage VM and is not supported.