RISC-V Verilog Implementation Report

June 28, 2025

1 Introduction

This report presents the design and implementation of a Verilog-based microarchitecture for the RV32I Instruction Set Architecture (ISA) with the RV32M multiply extension. The project focuses on a 5-stage pipelined processor, incorporating mechanisms to handle pipeline hazards and support various instruction types. A Python-based assembler converts assembly code to hexadecimal, and a testbench facilitates simulation. The report details the design specifications, project flow, hardware modules, hazard management, and references used.

2 Design Specifications

The RISC-V microarchitecture is designed with the following features:

- **Multi-cycle Design**: Implements a 5-stage pipeline consisting of Instruction Fetch (IF), Operand Fetch (OF), Execute (EX), Memory Access (MA), and Register Write (RW) stages.
- **Memory Access**: Uses separate instruction (`i_mem`) and data (`d_mem`) memories, both byte-addressable and accessed in big-endian format.
- **Instruction Support**: Supports all RV32I base integer instructions (except `ecall` and `ebreak`) and includes the RV32M multiply extension. The design allows for future integration of RV32A, RV32F, and RV32C extensions.
- **Pipeline Hazards**:
  - **Control Hazards**: Managed using pipeline flushing to handle branch mispredictions.
  - **Data Hazards**: Addressed through forwarding mechanisms, including RW to OF, RW to EX, RW to MA, and MA to EX paths.
  - **Load-Use Hazard**: Handled by pipeline stalling to resolve dependencies involving load instructions.
- **Operation Modes**: Supports both single-clock and double-clock (non-overlapping) execution modes.

3 Project Design Flow

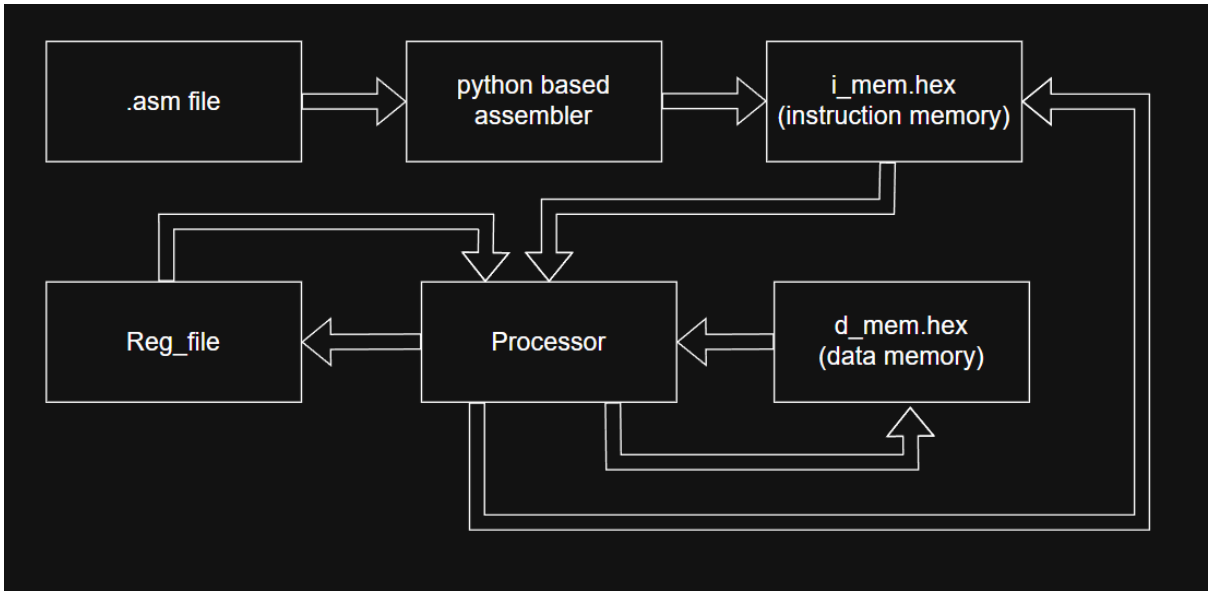

Figure 1:

The design flow for the RISC-V implementation is as follows:

1. **Assembly Code**: Instructions are written in a `.asm` file.
2. **Assembler Conversion**: A Python-based assembler converts the assembly instructions into hexadecimal format, outputting to a `.hex` file in byte-addressable big-endian format. A separate `.hex` file initializes the data memory.
3. **Memory Initialization**: The processor reads instructions from `i_mem` and data from `d_mem`, both implemented as 8-bit wide arrays with 1024 addresses, scalable as needed.
4. **Simulation**: The Verilog testbench instantiates initial data memory and register values, executes the instructions, and updates `d_mem.hex` and `reg.hex` files with final memory and register values post-simulation.

4 Python-Based Assembler

The assembler comprises two main Python scripts:

- **`I_binaries.py`**: Contains the logic to convert RV32I (excluding `ecall` and `ebreak`) and RV32M instructions to hexadecimal.
- **`RiscV.py`**: Reads the `.asm` file and writes the hexadecimal output to a specified `.hex` file.

To support additional ISA extensions, `I_binaries.py` must be modified. The assembler is executed by navigating to the code directory and running:

```
python RiscV.py
```

Remember to modify the input and output file path in RiscV.py

5 Hardware Modules

The microarchitecture follows a 5-stage pipelined design:

1. **Instruction Fetch (IF)**: Retrieves instructions from `i_mem`.
2. **Operand Fetch (OF)**: Decodes instructions and fetches operands from the register file.
3. **Execute (EX)**: Performs arithmetic and logical operations.
4. **Memory Access (MA)**: Executes memory read/write operations.
5. **Register Write (RW)**: Writes results back to the register file.
6. **Stall**: Contains the logic for stalling the pipeline in case of load - use hazard.

The supported instructions are listed in a separate file within the project repository.

6 Pipeline Hazard Management

The design addresses pipeline hazards to ensure correct execution.

6.1 Control Hazards

Control hazards arise when a branch instruction (`b_t`) is taken, leading to incorrect instruction fetches of next two instruction as the branch is resolved in 3rd clock cycle(EX stage). For ex, consider the following code snippet.

```
main:   i1
        i2
        bt
        i3
        i4
        ..
        ..
branch: b1
        b2
        ..
        ..
```

The pipeline behavior before and after flushing is shown below:

Before Flushing:

Stage	1	2	3	4	5	6	7	8	9	10
IF	i1	i2	b_t	i3	i4	b1				
ID		i1	i2	b_t	i3	i4	b1			
EX			i1	i2	b_t	i3	i4	b1		
MA				i1	i2	b_t	i3	i4	b1	
RW					i1	i2	b_t	i3	i4	b1

After Flushing:

Stage	1	2	3	4	5	6	7	8	9	10
IF	i1	i2	b_t	i3	i4	b1				
ID		i1	i2	b_t	i3	nop	b1			
EX			i1	i2	b_t	nop	nop	b1		
MA				i1	i2	b_t	nop	nop	b1	
RW					i1	i2	b_t	nop	nop	b1

These are managed by pipeline flushing:

- A D flip-flop generates a flush signal with `ex_isbranchtaken` as input and `flush` as output, activated in the next clock cycle after a branch is detected in the EX stage.
- In the ID stage, the flush signal converts the instruction (`i4`) to a nop.
- In the EX stage, the flush signal converts the instruction (`i3`) to a nop and discards operands op1 and op2.

6.2 Data Hazards

Data hazards are resolved using forwarding mechanisms:
*Note : The instructions used are dummy instruction and not part of actual ISA

- **RW to OF (3-stage)**: The destination of the load instruction(i1) is one of the source register of an alu instruction (i4). The load result of i1 is ready at the end of 4th clock pulse(end of MA and start of RW) and the value is needed by i4 at the start of 5th clock pulse(start of EX and end of OF). So the operands of ALU can choose between the value read from reg file or value forwarded from load result.

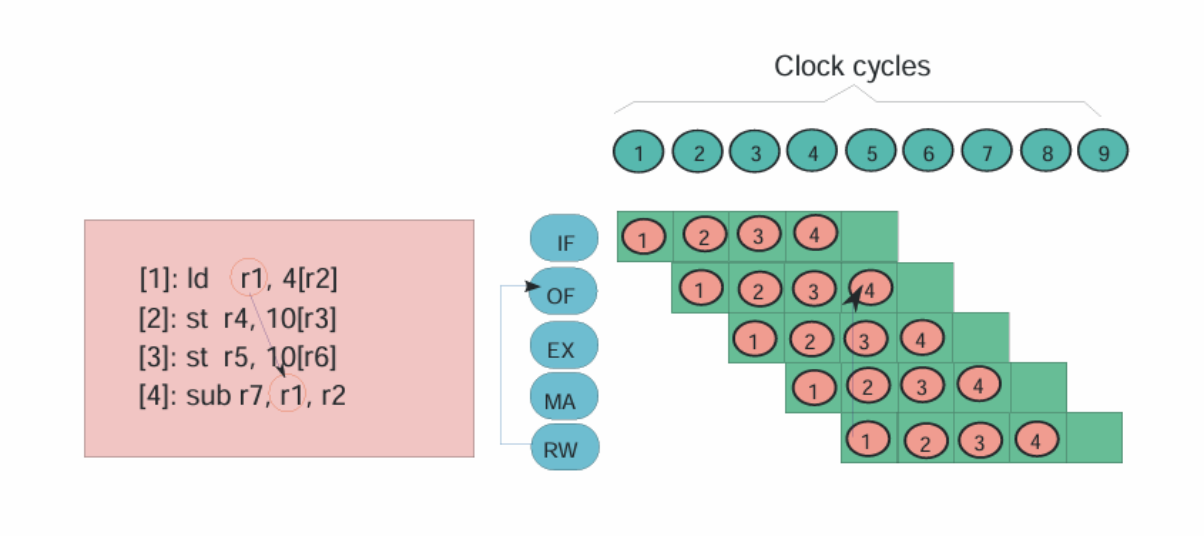

Figure 2:

- **RW to MA (1-stage)**: The destination reg of a load instruction (i1) is the source register of a store instruction. For such case , in the execution of store instruction the mdr can choose value from reg file or previos load result.

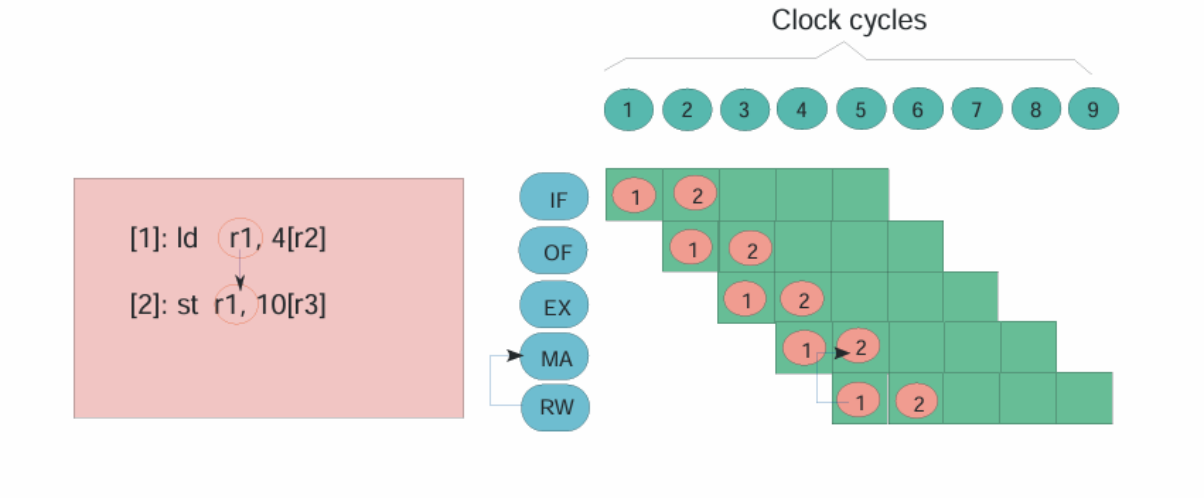

Figure 3:

- **MA to EX (1-stage)**: The destination of one alu instruction is used immediately as a source value by consecutive alu instruction.

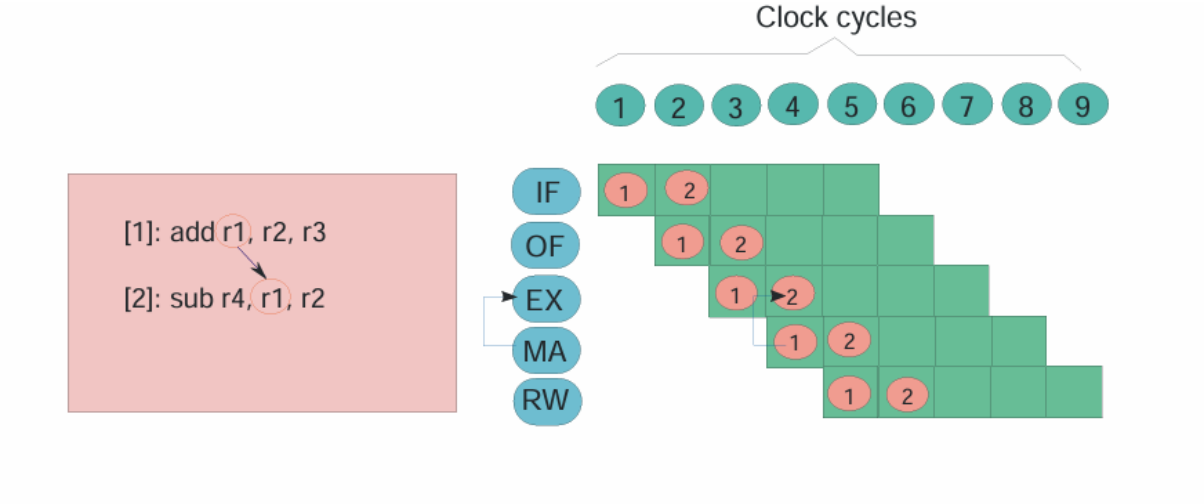

Figure 4:

- **RW to EX (2-stage)**: The logic is same as MA to EX stage forwarding only difference being the value forwarded in this case is the load result instead of alu result.

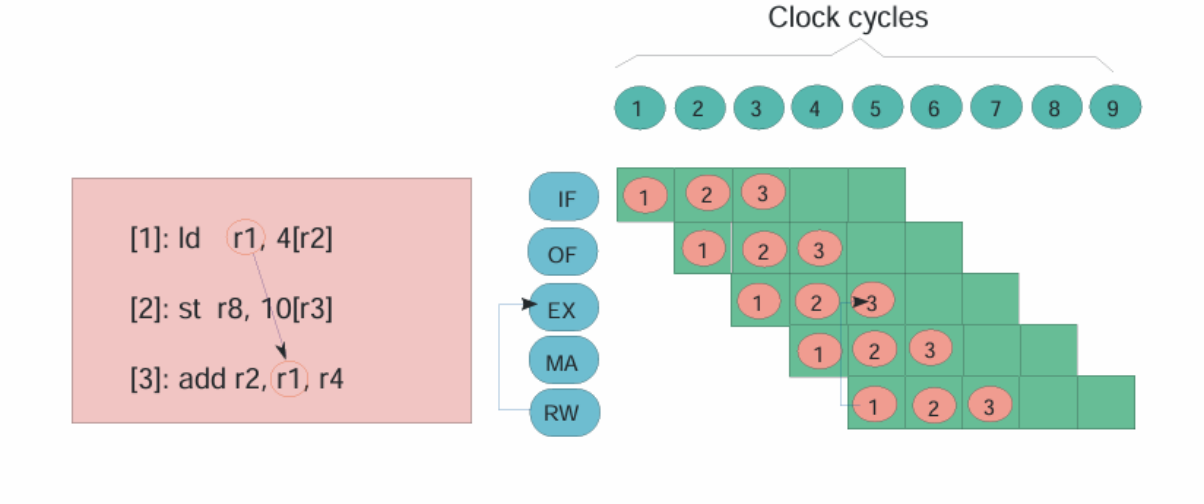

Figure 5:

6.3 Load-Use Hazard

Load-use hazards occur when an instruction depends on a load result before it is available, The load result is available at the end of the MA stage (cycle 4), but the subsequent instruction requires it in the EX stage (cycle 3). Forwarding cannot resolve this because we cant forward values back in time. So a pipeline stall inserts a bubble, delaying the dependent instruction by one cycle.

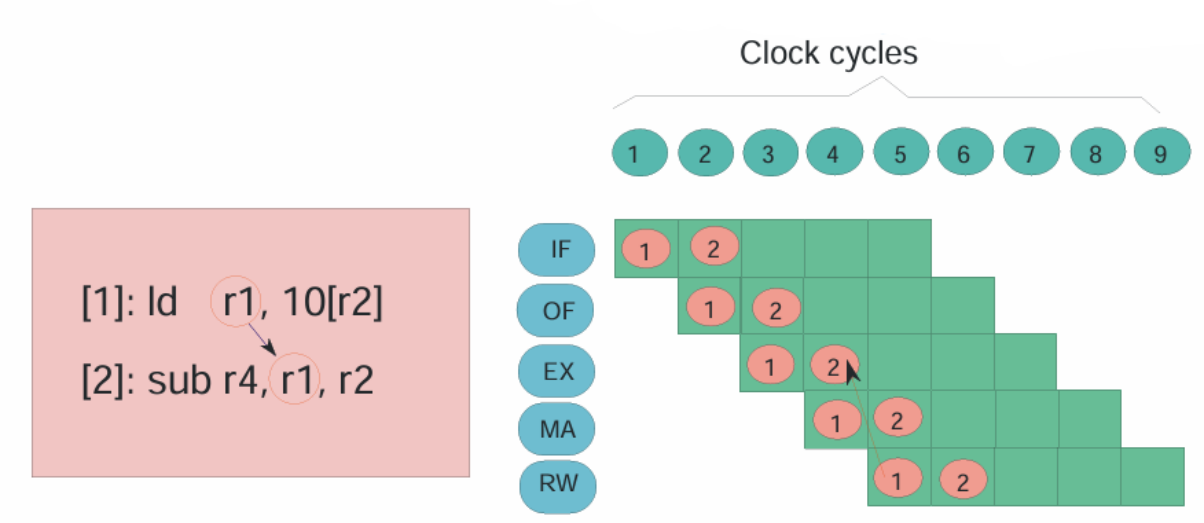

Figure 6:

The pipeline befor and after stalling :
Before Stalling:

Stage	1	2	3	4	5	6	7	8	9	10
IF	i1	i2	i3	i4						
ID		i1	i2	i3	i4					
EX			i1	i2	i3	i4				
MA				i1	i2	i3	i4			
RW					i1	i2	i3	i4		

After Stalling:

Stage	1	2	3	4	5	6	7	8	9	10
IF	i1	i2	i3	#i3	i4					
ID		i1	i2	#i2	i3	i4				
EX			i1	#i1	i2	i3	i4			
MA				i1	#i1	i2	i3	i4		
RW					i1	#i1	i2	i3	i4	

#i3 = # means a stall in the pipeline where i3 is representing the state at which instruction the pipeline is stalled

7 References

The implementation draws from the following resources:

- Chapters 6 and 7 of *Digital Design and Computer Architecture: RISC-V Edition* by Sarah L. Harris and David Money Harris.
- Lectures on basic computer architecture by Smruti R. Sarangi.