ELEC2204 - MIPS Pipeline Simulator

Lim Kai Shan
Electrical and Electronics Engineering
Part II
University of Southampton Malaysia
ksl2g22@soton.ac.uk

I. DESIGN

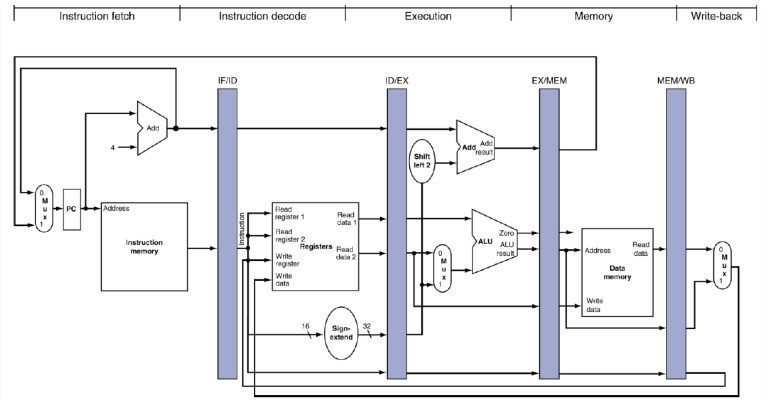

Fig 1(a). Pipeline MIPS processor

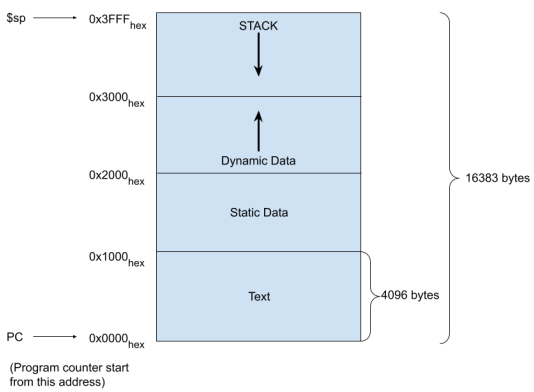

Fig 1(b). Memory Layout

name	reg#	convention
$zero	0	**constant 0**
$at	1	reserved for compiler
$v0-$v1	2–3	**results**
$a0-$a3	4–7	**arguments**
$t0-$t7	8–15	(callee-saved) temps
$s0-$s7	16–23	**caller-saved**
$t8-$t9	24–25	(callee-saved) temps
$k0-$k1	26–27	reserved for OS
$gp	28	global pointer
$sp	29	**stack pointer**
$fp	30	**frame pointer**
$ra	31	**return address**

Fig 1(c). MIPS register

Fig 1(a). shows the general MIPS processor Pipeline flow. The processor implements a five-stage pipeline architecture consisting of Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write-Back (WB) Stages. A Pipeline processor allows parallel execution of multiple instructions, with each instruction in a different stage at each cycle.

The processor in *Fig 1(a).* contains a few key components:

- Instruction Memory component stores the predefined assembly code in the Text segment of the memory shown in *Fig 1(b).*
- Registers component contains 32 general-purpose registers with specific designations as details in *Fig 1(c).*
- Data Memory unit reads and writes to the memory space shown in *Fig 1(b).*

In the Instruction Fetch (IF) stage, the Program Counter provides an address to retrieve the current instruction while simultaneously incrementing to prepare for the next instruction, with both the current PC value and fetched instruction stored in the IF/ID pipeline register. The Instruction Decode (ID) stage parses the instruction, generates control signals, and retrieves operands from registers, applying sign extension where needed, then passes all relevant data to the ID/EX pipeline register. During the Execution (EX) stage, the ALU performs the specified operation based on instruction type (R-type, I-type, or branch), with results and control signals forwarded to the EX/MEM pipeline register. The Memory Access (MEM) stage handles data memory operations for load and store instructions, while other instruction types bypass this functionality, with outcomes transferred to the MEM/WB pipeline register. Finally, in the Write-Back (WB) stage, results from either memory operations or ALU calculations are written to the destination register specified in the instruction, completing the execution cycle.

For this pipeline simulation project, I've selected a focused subset of MIPS instructions as shown in *Table 1*. This selection provides sufficient functionality to demonstrate core pipeline mechanics while maintaining manageable complexity. Each instruction type requires different execution paths through the pipeline stages, allowing for comprehensive testing of the processor's handling of R-type and I-type instructions.

Although the initial code architecture considered implementation of jump instructions, these were intentionally excluded from the final instruction set and execution stage. This decision significantly reduces the complexity of the simulator by avoiding the additional logic required for handling control flow alterations and branch prediction mechanisms operations. The absence of these instructions eliminates potential complications in pipeline control and hazard detection that could obscure the fundamental operation of the five-stage pipeline. This provides more robust initial implementation that focuses on perfecting the core pipeline functionality before introducing more complex instruction types. This approach provides a solid foundation that can be extended in future iterations once the basic pipeline mechanics have been fully verified.

II. FUNCTIONALITY

II.1 Simulator

The program is written with python using OOP method for the simplicity of writing and clean code. At the heart of the program is the Simulator class which orchestrates the entire simulation, it initializes and coordinates key components: memory, register file, pipeline registers, hazard detection unit, and the five pipeline stages (IF, ID, EX, MEM, WB). The `load_program` and `load_data` methods load instructions and data into appropriate memory segments, ensuring address validity. During each cycles, `execute_cycle` processes pipline stages in reversed order to prevent data hazard which is a common strategy in software_based MIPS simulator, update the pipeline registers. The `log_state` method captures the current state of the simulation, including pipeline registers, register file, and memory contents, utilizing the Logger class for recording. The `run` method controls the simulation loop, executing cycles until program completion or a predefined maximum cycle

count is reached, incorporating state logging and completion checks in each iteration to prevent mistakes in code that can cause the program to infinitely loop.

II.2 Memory

The `Memory` class simulates the MIPS memory architecture, dividing it into segments such as text, static, heap, and stack. It provides methods to load instructions and data, ensuring that addresses are correctly mapped within their respective segments. The class also handles memory read and write operations, facilitating instruction fetches and data accesses during simulation.

II.3 Register

The `Register` class models the 32 general-purpose MIPS registers. It maintains a mapping between register names and their numerical identifiers (to `instruction_set`), allowing for intuitive access. The class ensures that the `$zero` register remains immutable, always returning zero regardless of write attempts. It provides methods to read from and write to registers, supporting the execution of instructions that manipulate register values.

II.4 Pipeline Register

The `PipelineRegister` class provides the basic functionality for the latches between pipeline stages, such as IF/ID, ID/EX, EX/MEM, and MEM/WB. Each instance holds the necessary data and control signals required for the transition between stages. The class offers methods to read, write, and update its contents, enabling the smooth flow of instructions through the pipeline.

II.5 Hazard Unit

The `HazardUnit` class is responsible for detecting and managing hazards that can occur during instruction execution, including data and control hazards. It monitors the state of pipeline registers to indentify conflicts and determine when to use stall or forwarding. Forwarding takes Memory Access stage higher priority than Execute stage. To detect for stall, it checks if load instruction is in Execute stage and the register destination (rd) is used in Instruction Decode stage's source register if it uses as source then the method will retun true which will result in skipping the execution in Instruction Decode stage.

II.6 Pipeline stages

II.6.1 Instruction Fetch (IF)

The `IFStage` class retrieves the next instruction from memory based on the Program Counter and updates the pipeline register and PC for the next cycle.

II.6.2 Instruction Decode (ID)

The `IDStage` class decodes the fetched instruction, reads the necessary registers, and generates control signals required for execution.

II.6.3 Execute (EX)

The `EXStage` class performs arithmetic or logical operations, calculates memory addresses for load/store instructions. Forwarding is also performed in this stage.

II.6.4 Memory Access (MEM)

The `MEMStage` class accesses memory for load and store instructions, reading from or writing to the appropriate addresses.

II.6.5 Write Back (WB)

The `WBStage` class writes results back to the register file, completing the execution of the instruction.

II.7 Logger

The `Logger` class captures the state of the simulator at each cycle, including the contents of pipeline registers, the state of the register file, and the contents of memory segments. It provides methods to print the state after each cycle and to summarize the simulation after completion, aiding in debugging and analysis.

II.8 Instruction Parser

The `parse_instruction` function converts textual MIPS instructions into binary representations, extracting fields such as opcode, source and destination registers, and immediate values based on the instruction format (R-type, I-type, or J-type). This parsing facilitates the simulation of instruction execution within the pipeline.

II.9 Main

The `main.py` script is the entry point of the simulator. It initialize the simulation environment by creating the `Simulator` class instance and loading the instruction and data from the files to the class. The arguments is also being passed from main to define the maximum cycles.

II.10 Instruction_set

The instruction_set is the list of instruction set, opcode instruction format and dictionary of each register name to their number. This is the bank of all the binary of the opcodes and functions.

III. BASIC TESTING

The testing strategy is to test every module:

Memory Module: Tests verify correct storage and retrieval of words, proper handling of uninitialized memory (returning zero), enforcement of 4-byte alignment, and correct segment identification.

Register File: Tests ensure accurate read/write operations, immutability of the $zero register, and appropriate error handling for invalid register names.

Pipeline Registers: Tests confirm that data written to pipeline registers is correctly updated and that clearing the registers resets their state as expected.

Instruction Parser: Tests validate the correct parsing of R-type and I-type instructions, ensuring that operands and instruction types are accurately identified.

Simulator: Integration tests assess the execution of simple programs, verifying that arithmetic operations and memory load/store instructions produce the correct results in registers and memory.

The test suite executes 16 tests, all passed successfully, the simulator tests addi and add instructions which correctly computes and stores the result in the designated register. It also test memory operations such as sw and lw, which accurately stores and retrieves the memory. The full test result outcome can be found in the appendix.

IV. SHOWCASE TESTING

In the showcase testing shifting instruction plays a big role as shifting by 1 is essentially multiplying by 2, shifting by 2 is multiplying by 4 and so on. However, shifting only solve multiplication of even number. So for odd numbers, my solution is to shift by the highest amount that will get closest to the result then add by its own value. The following is my initial concept:

```
# For squaring 2 (2²)
addi $t1, $zero, 2      # $t1 = 2
sll $t0, $t1, 1         # $t0 = $t1 << 1 = 2 × 2 = 4

# For squaring 3
addi $t1, $zero, 3      # $t1 = 3
sll $t0, $t1, 1         # $t0 = $t1 << 1 = 3 × 2 = 6
add $t0, $t0, $t1       # $t0 = $t0 + $t1 = 6 + 3 = 9

# For squaring 4 (4²)
addi $t1, $zero, 4      # $t1 = 4
sll $t0, $t1, 2         # $t0 = $t1 << 2 = 4 × 4 = 16

# For squaring 5
addi $t1, $zero, 5          # $t1 = 5
sll $t0, $t1, 2             # $t0 = $t1 << 2 = 5 × 4 = 20
add $t0, $t0, $t1           # $t0 = $t0 + $t1 = 20 + 5 = 25

# For squaring 8 (8²)
addi $t1, $zero, 8      # $t1 = 8
sll $t0, $t1, 3         # $t0 = $t1 << 3 = 8 × 8 = 64
```

The above posses a problem which as the number gets larger the shifting gets complicated for example 0110 (6) shifted by 3 doesn't get 36 but gets 48 as there is two bits and each bits shifts is different thus the increase in instructions to achieve large squaring. In a result my program cannot handle all the instructions and the maximum it can get to is only up to 84 squared until it ran out of instruction memory. The total instruction to run the full program would take up to approximately 2000 instructions. The full outcome up to 84 squared can be found in the submitted log.txt file. It took 1012 process cycles to get to 84 squared.

V. Conclusion

In this coursework, I prioritised simplified instructions too much and ease of implementation for reliability on simple instructions and did not discover the potential issue of not able to fit all instructions in the memory. Thus, the failure to fully finish the showcase program. In the future I would like to include jump instruction type so that it can loop through some instructions without having similar intructions written over and over again.

VI. Acknowledgment

All of the codes are being modified by AI (Claude.ai), it is mainly to check the consistency and cleanliness of the code to reduce the bugs and logical errors. The final Assembly code for squaring is also mainly Claude generated excluding the starting few as the complication per calculation gets enormous.

Appendix

Instruction	Type	Format	Operand Meaning	Description
addi	I	rd, rs, imm	rd = rs + imm	Add immediate value to register rs and store result in rd.
add	R	rd, rs, rt	rd = rs + rt	Add values in rs and rt, store result in rd.
sub	R	rd, rs, rt	rd = rs - rt	Subtract rt from rs, store result in rd.
and	R	rd, rs, rt	rd = rs & rt	Bitwise AND between rs and rt, store result in rd.
or	R	rd, rs, rt	rd = rs | rt	Bitwise OR between rs and rt, store result in rd.
xor	R	rd, rs, rt	rd = rs ^ rt	Bitwise XOR between rs and rt, store result in rd.
nor	R	rd, rs, rt	rd = ~(rs | rt)	Bitwise NOR (NOT OR) between rs and rt, store result in rd.
lw	I	rt, imm(rs)	rt = MEM[rs + imm]	Load word from memory at address rs + imm into rt.
sw	I	rt, imm(rs)	MEM[rs + imm] = rt	Store word in rt into memory at address rs + imm.

Table 1. Limited Instruction Set

Basic test results:

.....Register write: $t0 = 100

Register write: $t1 = 42

Register write: $t2 = 42

===== Summary (8 cycles) =====

Final Register State:

$t0: 100

$8: 100

$t1: 42

$9: 42

$t2: 42

$10: 42

$sp: 16383

$29: 16383

Final Memory State:

TEXT SEGMENT:

TEXT[0x0]: addi $t0, $zero, 100

TEXT[0x4]: addi $t1, $zero, 42

TEXT[0x8]: sw $t1, 0($t0)

TEXT[0xc]: lw $t2, 0($t0)

TEXT[0x64]: 42

==========

.Register write: $t0 = 5

Register write: $t1 = 10

Register write: $t2 = 15

===== Summary (7 cycles) =====

Final Register State:

$t0: 5

$8: 5

$t1: 10

$9: 10

$t2: 15

$10: 15

$sp: 16383

$29: 16383

Final Memory State:

TEXT SEGMENT:

TEXT[0x0]: addi $t0, $zero, 5

TEXT[0x4]: addi $t1, $zero, 10

TEXT[0x8]: add $t2, $t0, $t1

==========

.

Ran 16 tests in 0.003s

OK