CA Project Report

Abdelrahman Elsamalouty(55-0842) T-21

Abdelrahman Elaby(55-0990) T-21

Habiba Hesham(55-0966) T-21

Hazem Mansour(55-1036) T-21

Zeina Mohamed(55-1095) T-21

Ehab Medhat(55-4571) T-21

Team Name: ASHEZ
Team Number: 9
Package Number: 1
Package Name: Spicy Von Neumann Fillet with extra shifts

1 Introduction

In this Project, we conducted a simulation to the Von Neumann Architecture that includes pipelining. The simulation code mimics the behavior of a MIPS (Microprocessor without Interlocked Pipeline Stages) processor, allowing us to observe the execution of MIPS assembly instructions and the behavior of its pipeline stages.

2 Methodology

This methodology section describes in detail the steps and functions that are used to mimic a MIPS processor. A simple pipeline design with phases for memory access, write-back, fetching, decoding, and executing instructions is implemented by the code. The intention is to emulate the memory interactions and instruction cycle of a streamlined MIPS processor.

2.1 Instructions Implemented

1. **Add (ADD)**:
   - **Mnemonic**: ADD
   - **Type**: R (Register)
   - **Format**: ADD R1 R2 R3
   - **Operation**: $R1 = R2 + R3$
   - **Description**: The ADD instruction adds the values in registers R2 and R3 and stores the result in register R1.
2. **Subtract (SUB)**:
   - **Mnemonic**: SUB
   - **Type**: R (Register)
   - **Format**: SUB R1 R2 R3
   - **Operation**: $R1 = R2 - R3$
   - **Description**: The SUB instruction subtracts the values in registers R3 from R2 and stores the result in register R1.
3. **Multiply Immediate (MULI)**:
   - **Mnemonic**: MULI
   - **Type**: I (Immediate)
   - **Format**: MULI R1 R2 IMM
   - **Operation**: $R1 = R2 \times \text{IMM}$

- **Description**: The MULI instruction multiplies the value in register R2 by the immediate value (IMM) and stores the result in register R1.

4. **Add Immediate (ADDI)**:
   - **Mnemonic**: ADDI
   - **Type**: I (Immediate)
   - **Format**: ADDI R1 R2 IMM
   - **Operation**: $R1 = R2 + \text{IMM}$
   - **Description**: The ADDI instruction adds the immediate value (IMM) to the value in register R2 and stores the result in register R1.

5. **Branch if Not Equal (BNE)**:
   - **Mnemonic**: BNE
   - **Type**: I (Immediate)
   - **Format**: BNE R1 R2 IMM
   - **Operation**: If $R1 \neq R2$, then PC = PC + 1 + IMM
   - **Description**: The BNE instruction checks if the values in registers R1 and R2 are not equal. If true, it updates the program counter (PC) to jump to the specified address.

6. **And Immediate (ANDI)**:
   - **Mnemonic**: ANDI
   - **Type**: I (Immediate)
   - **Format**: ANDI R1 R2 IMM
   - **Operation**: $R1 = R2\&\text{IMM}$
   - **Description**: The ANDI instruction performs a bitwise AND operation between the value in register R2 and the immediate value (IMM), storing the result in register R1.

7. **Or Immediate (ORI)**:
   - **Mnemonic**: ORI
   - **Type**: I (Immediate)
   - **Format**: ORI R1 R2 IMM
   - **Operation**: $R1 = R2|\text{IMM}$
   - **Description**: The ORI instruction performs a bitwise OR operation between the value in register R2 and the immediate value (IMM), storing the result in register R1.

8. **Jump (J)**:

- **Mnemonic**: J
- **Type**: J (Jump)
- **Format**: J ADDRESS
- **Operation**: $PC = PC[31:28]||\text{ADDRESS}$
- **Description**: The J instruction unconditionally jumps to the specified address. It updates the program counter (PC) to the first 4 bits of the PC concatenated with the target address.

9. **Shift Left Logical (SLL)**:
   - **Mnemonic**: SLL
   - **Type**: I (Immediate)
   - **Format**: SLL R1 R2 SHAMT
   - **Operation**: $R1 = R2 \ll \text{SHAMT}$
   - **Description**: The SLL instruction shifts the value in register R2 left by the specified shift amount (SHAMT) and stores the result in register R1.
10. **Shift Right Logical (SRL)**:
    - **Mnemonic**: SRL
    - **Type**: I (Immediate)
    - **Format**: SRL R1 R2 SHAMT
    - **Operation**: $R1 = R2 \gg \text{SHAMT}$
    - **Description**: The SRL instruction shifts the value in register R2 right by the specified shift amount (SHAMT) and stores the result in register R1.
11. **Store Word (SW)**:
    - **Mnemonic**: SW
    - **Type**: I (Immediate)
    - **Format**: SW R1 R2 IMM
    - **Operation**: $\text{MEM}[R2 + \text{IMM}] = R1$
    - **Description**: The SW instruction stores the value in register R1 into memory at the address specified by the sum of the value in register R2 and the immediate value (IMM).
12. **Load Word (LW)**:
    - **Mnemonic**: LW
    - **Type**: I (Immediate)
    - **Format**: LW R1 R2 IMM

- **Operation**: $R1 = \text{MEM}[R2 + \text{IMM}]$
- **Description**: The LW instruction loads a word from memory at the address specified by the sum of the value in register R2 and the immediate value (IMM), storing it in register R1.

2.2 Data Structures and Global Variables

- **Memory and Registers:**
  - `int *memoryfile`: Pointer to the memory array holding the instructions.
  - `int registerFile[32]`: Array representing 32 general-purpose registers.
  - `int NumberofInstructions`: Tracks the number of instructions loaded into memory.
  - `int pc`: Program counter, indicating the address of the next instruction.
  - `int zeroFlag, overflowFlag, carryFlag`: Flags used in arithmetic operations.
- **Pipeline Registers:**
  - `int* IFID, *IDEX, *EXMEM, *MEMWB`: Arrays representing pipeline registers between different stages.
- **Cycle and Control Signals:**
  - `int cycle`: Counter to keep track of the number of cycles.
  - `int fetchInt, fetchSave, decodeInt, executeInt, memoryInt, writeBackInt`: Control signals to manage instruction flow between stages.

2.3 Helper Functions

- `printBits(size_t const size, void const * const ptr)`:
  - Prints the binary representation of a value.
- `swapIntegers(int *a, int *b)`:
  - Swaps the values of two integers.
- `printarray(int *arr)`:
  - Prints the elements of an array.

2.4 Core Functions

- `loadInstToMemory(char *fileName)`:
  - Loads instructions from a file into memory.
  - Converts assembly instructions to binary using `LineToBinary(char *line)`.
- `fetch()`:
  - Fetches the next instruction from memory using the program counter (`pc`).
  - Updates the `IFID` pipeline register with the program counter and the fetched instruction.
- `decode()`:
  - Decodes the instruction in the `IFID` register.
  - Extracts opcode, source, and destination register identifiers.
  - Updates the `IDEX` pipeline register with decoded values.
- `execute()`:
  - Executes the decoded instruction.
  - Performs arithmetic or logical operations based on the opcode.
  - Uses the `ALU(int operandA, int operandB, int operation)` function to perform calculations.
  - Updates the `EXMEM` pipeline register with the results and control signals.
- `memory()`:
  - Accesses memory for load/store instructions.
  - Updates the `MEMWB` pipeline register with the data to be written back to registers.
- `writeback()`:
  - Writes the results from the `MEMWB` register back to the appropriate register in `registerFile`.

2.5 ALU Operations

The ALU function performs various operations based on the provided opcode:

- Addition, subtraction, and multiplication with overflow and carry checks.
- Logical AND, OR, left shift, and right shift operations.

2.6 Main Execution Loop

The `main()` function orchestrates the simulation:

1. **Initialization:**
   - Loads instructions into memory.
   - Allocates memory for pipeline registers and temporary arrays.
2. **Pipeline Simulation:**
   - Iterates through cycles, managing the pipeline stages.
   - Updates control signals and calls appropriate functions for each stage.
   - Prints the state of the pipeline and registers for each cycle.
   - Beginning with clock cycle 1, you retrieve an instruction every two clock cycles.
   - An instruction spends two clock cycles in the Decode (ID) stage.
   - An instruction spends two clock cycles in the Execute (EX) stage.
   - An instruction spends one clock cycle in the Memory (MEM) stage.
   - An instruction spends one clock cycle in the Write Back (WB) stage.
   - The Memory (MEM) and Instruction Fetch (IF) phases cannot operate simultaneously. At any one time, only one of them is in use.
   - Branch or Jump instructions cause the pipeline to be flushed, meaning all the instructions that were fetched or decoded will be flushed.

Figure 1: Example of a pipeline with no branch or jump

Package 1 Pipeline					
	Instruction Fetch (IF)	**Instruction Decode (ID)**	**Execute (EX)**	**Memory (MEM)**	**Write Back (WB)**
Cycle 1	Instruction 1				
Cycle 2		Instruction 1			
Cycle 3	Instruction 2	Instruction 1			
Cycle 4		Instruction 2	Instruction 1		
Cycle 5	Instruction 3	Instruction 2	Instruction 1		
Cycle 6		Instruction 3	Instruction 2	Instruction 1	
Cycle 7	Instruction 4	Instruction 3	Instruction 2		Instruction 1
Cycle 8		Instruction 4	Instruction 3	Instruction 2	
Cycle 9	Instruction 5	Instruction 4	Instruction 3		Instruction 2
Cycle 10		Instruction 5	Instruction 4	Instruction 3	
Cycle 11	Instruction 6	Instruction 5	Instruction 4		Instruction 3
Cycle 12		Instruction 6	Instruction 5	Instruction 4	
Cycle 13	Instruction 7	Instruction 6	Instruction 5		Instruction 4
Cycle 14		Instruction 7	Instruction 6	Instruction 5	
Cycle 15		Instruction 7	Instruction 6		Instruction 5
Cycle 16			Instruction 7	Instruction 6	
Cycle 17			Instruction 7		Instruction 6
Cycle 18				Instruction 7	
Cycle 19					Instruction 7

Figure 2: Example of a pipeline with a branch in instruction 3 to instruction 7

	Instruction Fetch (IF)	**Instruction Decode (ID)**	**Execute (EX)**	**Memory (MEM)**	**Write Back (WB)**
Cycle 1	Instruction 1				
Cycle 2		Instruction 1			
Cycle 3	Instruction 2	Instruction 1			
Cycle 4		Instruction 2	Instruction 1		
Cycle 5	Instruction 3	Instruction 2	Instruction 1		
Cycle 6		Instruction 3	Instruction 2	Instruction 1	
Cycle 7	Instruction 4	Instruction 3	Instruction 2		Instruction 1
Cycle 8		Instruction 4	Instruction 3	Instruction 2	
Cycle 9	Instruction 5	Instruction 4	Instruction 3		Instruction 2
Cycle 10				Instruction 3	
Cycle 11	Instruction 7				Instruction 3
Cycle 12		Instruction 7			
Cycle 13	Instruction 8	Instruction 7			
Cycle 14		Instruction 8	Instruction 7		
Cycle 15		Instruction 8	Instruction 7		
Cycle 16			Instruction 8	Instruction 7	
Cycle 17			Instruction 8		Instruction 7
Cycle 18				Instruction 8	
Cycle 19					Instruction 8

3. **Termination:**
   - Frees allocated memory.
   - Prints the final state of registers and memory.

2.7 Pseudo Algorithm

Main Loop:

1. Initialize the MIPS pipeline stages and program counter (PC).
2. Iterate through pipeline stages in each cycle until all instructions are processed.
3. Execute each pipeline stage in the following order: Write Back, Memory Access, Execute, Instruction Decode, Instruction Fetch.
4. Update control signals and data flow between pipeline stages.
5. Increment the cycle counter to track the progress of execution.
6. Terminate the loop when all instructions are processed.

3 Results

3.1 Segmentation Faults

We faced segmentation faults due to illegal memory access, such as dereferencing a null or invalid pointer, or accessing out-of-bounds array indices.

3.2 Solution

- **Pointer Initialization**: Ensure all pointers are properly initialized before use. For instance, the pointer arrays like IFID, IDEX, EXMEM, and MEMWB should be allocated enough memory.
- **Bounds Checking**: Implement bounds checking to ensure array indices are within valid range. This is especially crucial when accessing memoryfile and registerFile.
- **Memory Allocation**: Before accessing dynamically allocated memory, check if the memory allocation was successful.

3.3 Uninitialized Variables

We had some problems with uninitialized variables that led to unpredictable behavior and incorrect results. For instance, when fetchInt, decodeInt, executeInt, or memoryInt were used without being properly initialized, it caused the program to malfunction.

3.4 Solution

- **Initialization**: Ensure all variables are properly initialized before use.
- **Default Values**: Set default values for all variables and structures at the beginning of the program.

3.5 Memory Leaks

We allocated memory dynamically using malloc without freeing it appropriately. If memory is not freed, it led to memory leaks, especially in long-running programs.

3.6 Solution

- **Memory Management**: Ensure that all dynamically allocated memory is freed at the end of the program. This is already partly done, but make sure it's comprehensive

3.7 Undefined Behavior in ALU Function

The ALU function exhibited undefined behavior during operations like multiplication or shifts if inputs were not handled properly.

3.8 Solution

- **Sanitize Inputs**: Ensure that inputs to the ALU function from the text file are sanitized and within expected ranges. Implement checks for edge cases, such as large integer values that might cause overflow, and handle negative number inputs from the arithmetic operations.

3.9 Instruction Fetch and Decode Synchronization

The synchronization between fetch, decode, execute, memory, and writeback stages was not correct, leading to improper instruction execution order.

3.10 Solution

- **Pipeline Control**: Ensure that the pipeline stages are correctly synchronized. Implement control logic to manage the pipeline flow, handling hazards and stalls appropriately.
- **Cycle Management**: Verify that the cycle count and stage transitions are accurately implemented.

4 Conclusion

To sum up, the MIPS pipeline simulation offered insightful knowledge on how a MIPS processor functions and how its pipeline stages behave. The simulation showed where the code needed to be improved, especially in memory management and address manipulation, even if it ran into some problems. By resolving these problems, the simulation can be improved to offer a more realistic depiction of an actual MIPS processor, supporting additional computer architecture research and development.