Computer Architecture Simulator Documentation

Shahd Abdelkhalek 55-3480 T18
Nabila Shreif 55-2803 T26
Youssef El Shamy 55-12061 T17
Jana Saad 55-4038 T26
Diana Rehan 55-12832 T5
Team 6
Package 3

May 17, 2024

1 Introduction

This document provides documentation for the computer architecture simulator project. The simulator is designed to emulate the pipeline stages of a simple computer architecture, including instruction fetching, decoding, execution, memory access, and write back.

2 Methodology

The simulator is implemented in C. It consists of several main components:

2.1 Global Variables Initialization

This section initializes global variables such as the program counter (`pc`), register file, memory, and other control variables.

Listing 1: Global Variables Initialization

```
int pc= 0; //our global pc
int registerFile[32];
int memindexInstr=0;
int memindexData=1024;
int totalclkcycles;
int clockCycles=1;
int branchflag=0;
int decodeFlag=0; //fakedecode
int executeFlag=-1; //fakeexecute
int j=6;
int memafterbranch=0;
```

2.2 Fetch Stage

In this stage, the simulator fetches instructions from memory based on the current value of the program counter (`pc`).

Listing 2: Fetch Stage

```
Function fetch():
    f.instruction = memory[pc]
    pc = pc + 1
```

2.3 Decode Stage

The decode function is responsible for decoding the fetched instruction, extracting various fields such as opcode, register operands (r1, r2, r3), shift amount (shamt), immediate value (imm), and address from the instruction.
This is done by masking the instruction with a bitmask that isolates the needed bits and then shifting them to the rightmost position.

The decode function ensures that all necessary components of the instruction are properly decoded, and the needed register operands are brought from the register file.

Listing 3: Decode Stage

```
Function decode():
    unsigned opcode = f.instruction &
    0b11110000000000000000000000000000
    dec.opcode = opcode >> 28
    dec.r1 = (f.instruction &
    0b00001111100000000000000000000000) >> 23
    dec.valueOfR1 = registerFile[dec.r1]
    dec.r2 = registerFile[(f.instruction &
    0b00000000011111000000000000000000) >> 18]
    dec.r3 = registerFile[(f.instruction &
    0b00000000000000111110000000000000) >> 13]
    dec.shamt = f.instruction &
    0b00000000000000000001111111111111
    int temp = f.instruction & 0b00000000000000011111111111111111
    int temp1 = temp >> 17
    if temp1 == 1 then
        dec.imm = temp | 0b11111111111111000000000000000000
    else
        dec.imm = temp
    dec.address = f.instruction &
    0b00001111111111111111111111111111
```

2.4 Execution Stage

The execute function is responsible for performing the execution stage of the pipeline, where instructions are processed based on their opcode and operands.

Firstly, it propagates the opcode and register values from the decode stage to the execution stage.

Then,it evaluates the opcode using a switch-case structure:

- For arithmetic operations such as addition, subtraction, and multiplication (opcode 0, 1, and 2 respectively), it performs the corresponding arithmetic op-

eration on the operands (r2 and r3).

- For the Move Imm operation (opcode 3), it simply loads the immediate value stored in the instruction.

- For branching instructions (opcode 4), it checks if the value of a register matches another register. If the condition is met, it sets a flag to indicate a branch and updates the program counter accordingly.

- For bitwise operations such as AND and XOR (opcode 5 and 6), it performs the respective operation on the operands.

- For the JUMP operation (opcode 7), it modifies the program counter to the specified address, effectively jumping to a new instruction.

- For shifting operations (opcode 8 and 9), it performs left shift and arithmetic right shift operations, preserving the sign bit for the latter.

- For MOVR and MOVM operations (opcode 10 and 11), it adds an immediate value to a register value to use them later for mameory accessing.

If there is no valid instruction to execute (both opcode and r1 are -1), it sets the opcode, r1, and result values to -1 as default.

Listing 4: Execution Stage

```
Function execute():
    // propagates opcode and register values from decode stage
    exec.opcode = dec.opcode
    exec.r1 = dec.r1

       switch (dec.opcode):
    case 0:// ADD operation: Adds dec.r2 and dec.r3, stores
    result in exec.result

    case 1:// SUBTRACT operation: Subtracts dec.r3 from dec.r2,
    stores result in exec.result

    case 2:// MULTIPLY operation: Multiplies dec.r2 and dec.r3,
    stores result in exec.result

    case 3:// MOV IMM operation: put dec.imm into exec.result

    case 4:/ BRANCH IF EQUAL operation: Compares dec.r1 with
    dec.r2, adjusts pc if equal, sets branchflag
```

```
    case 5:// AND operation: Bitwise AND between dec.r2 and
    dec.r3, stores result in exec.result

    case 6:// XOR operation: Bitwise XOR between dec.r2 and
    dec.imm, stores result in exec.result

    case 7:// JUMP operation: Constructs new pc from msb_pc and
    dec.address, sets branchflag

    case 8:// SHIFT LEFT operation: Shifts dec.r2 left by
    dec.shamt bits, stores result in exec.result

    case 9:// ARITHMETIC SHIFT RIGHT operation: Arithmetic shift
    right on dec.r2, preserves sign bit, by dec.shamt bits

    case 10, 11:// MOVR and MOVM operations: Adds dec.imm to
    dec.r2, stores result in exec.result
```

2.5 Memory Access Stage

The memoryAccess function handles memory operations based on the exec.opcode. after propagating the values from the previous stage it checks the opcode:

Opcode 10 (MOVR): Reads data from the memory location specified by exec.result for writing it back in a register in the following WB stage.

Opcode 11 (MOVM): Writes the value from registerFile to the memory location specified by exec.result.

This function facilitates memory read and write operations based on the executed instruction, ensuring proper interaction with the simulated memory subsystem.

Listing 5: Memory Access Stage

```
Function memoryAccess():
    mem.opcode = exec.opcode
    mem.memloc = exec.result
    mem.r1 = exec.r1

    if exec.opcode is not equal to -1 and exec.result is not
    equal to -1 and exec.r1 is not equal to -1 then
        if exec.opcode equals 10 then
            Print "reading from memory location " + exec.result
            mem.valuefrommemory = memory[exec.result]
            Print mem.valuefrommemory
```

```
        else if exec.opcode equals 11 then
            memory[exec.result] = registerFile[exec.r1]
            Print "value stored at memory location " +
    exec.result + " is now set to " + exec.r1
        else
            Print "entered memory but no memory involvement"
```

2.6 Write Back Stage

For opcodes other than 11 (MOVM), 10 (MOVR), 4 (BRANCH IF EQUAL), and 7 (JUMP), it updates the register value (registerFile[mem.r1]) with the calculated register's value

If the opcode is 10 (MOVR), it updates the register value (registerFile[mem.r1]) with mem.valuefrommemory, which represents the value read from memory during the memory access stage.

This function ensures that results computed in earlier stages of the pipeline are properly written back to the register file, maintaining the correct state of the simulated processor.

Listing 6: Write Back Stage

```
Function writeBack():
    if mem.opcode is not -1 and mem.memloc is not -1 and mem.r1
    is not -1 then
        if mem.opcode is not 11 and mem.opcode is not 10 and
    mem.opcode is not 4 and mem.opcode is not 7 then
            if mem.r1 is not equal to 0 then
                registerFile[mem.r1] = mem.memloc

        else if mem.opcode equals 10 then
            registerFile[mem.r1] = mem.valuefrommemory

    else
        Print "no write back"
```

2.7 Pipeline Execution

The pipelining function simulates the pipelined execution of instructions in a computer architecture. It iterates until all instructions are processed.

Initialization and Loop Control:

The loop continues until the program counter (pc) reaches the last instruction (memindexInstr) and this last instruction actually passes through the pipeline stages.

Loop Termination: Ends the loop when j becomes zero, indicating all instructions have been processed.

Listing 7: Pipeline Execution

```
Function pipelining():

    while pc < memindexInstr or j > 0:
        if pc == memindexInstr:
            j--

        if mem.r1 != -1 and clockCycles % 2 == 1:
            writeBack()

        if exec.opcode != -1 and clockCycles % 2 == 0 and
    (memafterbranch == 0 or memafterbranch == 1):
            memoryAccess()

        decode and execute stages based on decodeFlag and
    executeFlag which gives the desired result (each of them
    should take 2 cycles).

        Fetch instructions if any.

        Handle branch instructions to clear pipeline stages.

        Increment clock cycles.
```

2.8 Input Reading and Saving in Memory

2.8.1 Save In Memory

Function void saveInMemory(char instruction[]): This function reads an assembly-like instruction string, parses it, and converts it into a 32-bit machine instruction format suitable for simulation. It identifies the operation type (ADD, SUB,

MUL, MOVI, JEQ, AND, XORI, JMP, LSL, LSR, MOVR, MOVM) from the instruction string, extracts operands (registers or immediate values), and constructs the corresponding machine instruction. Finally, it stores the generated instruction in a memory array.

2.8.2 Read From File

Function void readFromFile(): This function reads lines of assembly-like instructions from a file named "test.txt". Each line represents an instruction to be processed by the saveInMemory function.

These functions together facilitate the loading and processing of assembly-like instructions from a file into our memory.

2.9 Initialization and Main Function

2.9.1 Initialization

Function void init()

Initializes various data structures (f, dec, mem, exec) and register file (registerFile) to empty states.
Calls readFromFile() to read assembly-like instructions from "test.txt" file and store them in memory.

Calls pipelining() to simulate the pipeline execution of instructions stored in memory.
Calculates Total Clock Cycles:

Calculates totalclkcycles based on the number of instructions read (memindexInstr), assuming a fixed cycle count for each instruction and pipeline stages.

2.9.2 Main Function

Function int main()

Calls init() to set up the processor environment, load instructions, and start simulation.

Prints the total number of clock cycles (clockCycles) elapsed during simulation.

Prints totalclkcycles, which represents the total expected clock cycles for executing all instructions.

3 Challenges and Solutions

3.1 Control Flow Handling and Memory Access Timing

One challenge was handling jump instructions correctly within the pipelined architecture. The issue was because the simulator entered stage functions based on signals set in the previous cycle. For jump instructions, unnecessary memory access and write-back operations were performed.

To address this, a flag called `mem_after_branch` was introduced. When a jump instruction is detected, this flag is set to 2 before entering the memory stage. If the flag is set to 2, the simulator skips the memory access and write-back stages for the next two cycles. Additionally, instructions in the fetch and decode stages are flushed to prevent incorrect execution.

3.2 Method Invocation Order

Another challenge was the sequential order of method invocation within the simulator, which caused issues such as overwritten values. This occurred because the methods were called in an order that did not preserve the correct data flow through the pipeline stages.

We resolved this issue by adjusting the order of method calls, we inverted the order of calling the methods to ensure that data dependencies are respected, and values are not overwritten before they are used.

```
the order of calling the methods en the pipeline:

        write_back(r3, registers[r3]) // Write back earlier
        memory_access(opcode, r1, r2, imm)
        execute(opcode, r1, r2, r3, shamt, imm, address)
        decode(instruction)
        instruction = fetch()
```

3.3 Shifting Operations

for right shifts, the sign bit is shifted to preserve the sign of the number.

Listing 8: Shifting Operations

```
shifting right in case 9 of the decode function switch:
exec.result = dec.r2;  // Initialize exec.result with dec.r2
    value
signbit = dec.r2 & 0b10000000000000000000000000000000;  //
    Extract sign bit from dec.r2
```

```
// Perform arithmetic shift right on exec.result preserving the
    sign bit
for (int i = 0; i < dec.shamt; i++) {
    exec.result >>= 1;  // Shift right by 1
    exec.result |= signbit;  // Maintain the sign bit
}
```

4 Output

The output of the simulator includes the final state of the register file, memory contents, and the number of clock cycles to execute all instructions. It is important to note that the number of execution cycles depends on the entry of instructions into the pipeline rather than the actual count of instructions.

5 Conclusion

The computer architecture simulator provides simulating key components such as instruction fetching, decoding, execution, memory access, and write-back. One notable aspect is that the number of clock cycles for the execution of all instructions depends on the entering of the pipeline rather than the actual number of instructions. This is due to the pipelined architecture, which allows for overlapping execution and improves overall throughput.