Computer Architecture: Assignment 2

MIPS Processors

Pradyun Devarakonda IMT2022525
T Soma Datta IMT2022077

Non-pipelined Processor

Instruction_Manager is a class that represents an instruction in the pipeline. Each instruction goes through various stages, including "UNFETCHED," "FETCHED," "DECODED," "EXECUTED," "MEMACCESSED," and "WRITTENBACK."

The constructor initializes an instance of the Instruction_Manager class. It takes an instru parameter representing the instruction and initializes various attributes such as state, instrtype, op, regs, imm, shamt, targetaddr, result, loader, and to_Branch. The initial state is set to "UNFETCHED," and the result is initialized to -1.

The string method provides a string representation of the Instruction_Manager object. It returns a string containing the instruction and its current state.

We initialise a Program counter in the initial stage which checks every instruction basically a counter of all the lines in the machine code.

- **Fetched(self)**

This method updates the state of the instruction to "FETCHED" when the instruction is fetched from the instruction memory.

- **isFetched(self)**

This method returns True if the instruction is in the "FETCHED" state and False otherwise.

- **Decoded(self, command, opcode_map)**

This method is responsible for decoding the instruction based on its opcode. It updates the state to "DECODED" and sets various attributes such as instrtype, op, regs, imm, shamt, targetaddr, etc., based on the instruction type. For example, for an R-type instruction, it extracts fields like rs, rt, rd, and shamt.

- **isDecoded(self)**

This method returns True if the instruction is in the "DECODED" state and False otherwise.

- **Executed(self, result)**

This method updates the state to "EXECUTED" and stores the result of the execution in the result attribute.

- **isExecuted(self)**

This method returns True if the instruction is in the "EXECUTED" state and False otherwise.

- **Mem_Accessed(self)**

This method updates the state to "MEMACCESSED" when the instruction is accessing memory.

- **isMem_Accessed(self)**

This method returns True if the instruction is in the "MEMACCESSED" state and False otherwise.

- **Writtenback(self)**

This method updates the state to "WRITTENBACK" when the result is written back into the register memory.

- **isWrittenback(self)**

This method returns True if the instruction is in the "WRITTENBACK" state and False otherwise.

- **Instru_Fetch(Instru_memory, PC)**

This is a static method that returns the instruction at the specified program counter (PC) from the instruction memory (Instru_memory). It prevents the need to instantiate an object to use this method.

- **binaryToDecimal(val, bits)**

This is a method that converts a binary value to a decimal value, considering it as a signed integer. It is used for sign extension.

- **Instru_Decode(instruction, opcode_map, Reg_memory, LUIN)**

This method decodes the instruction based on its opcode and updates its attributes accordingly. It also handles specific cases for different instruction types, such as R-type, I-type, J-type, and M-type.

- **Execute(instruction, Reg_memory)**

This function performs the execution of the instruction. It retrieves values from the register memory, performs the specified operation based on the opcode, and returns the result. If the instruction is a J-type instruction, it returns -1.

- **Mem_Access(instruction, Data_memory, Reg_memory, result)**

This function handles memory access for load (lw) and store (sw) instructions. It updates the data memory (Data_memory) and returns a tuple containing the updated data memory and a loader value.

- **Writeback(instruction, Reg_memory, result, loader)**

This function handles the writeback stage, updating the register memory (Reg_memory) based on the instruction type and result. It returns the updated register memory.

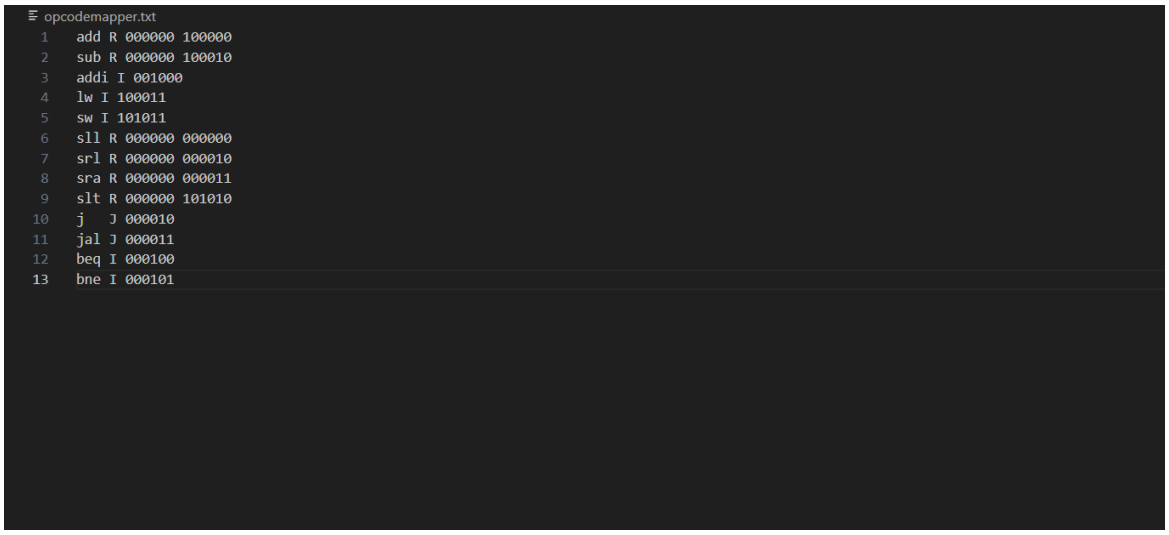
- **mapper()**

This function reads opcode and register code mappers from files, creating and returning dictionaries (opcode_map and regcode_map).

We have to Take care of the clock cycles, as it is a non-pipelined processor. So, the clock cycles is 5 times the number of instructions as 5 stages are included. IF, ID,EX,MEM,WB.

Each method in the Instruction_Manager class serves a specific purpose in the simulation of a non - pipelined processor, managing the various stages of instruction execution and memory access.

OPCODEMAPPER



REGCODEMAPPER

regcodemapper.txt

```
$0  00000 return 0
$s0 10000
$s1 10001
$s2 10010
$s3 10011
$s4 10100
$s5 10101
$s6 10110
$s7 10111
$t0 01000
$t1 01001
$t2 01010
$t3 01011
$t4 01100
$t5 01101
$t6 01110
$t7 01111
$t8 11000
$t9 11001
$ra 11111
%a0 00100
%a1 00100
%a2 00100
%a3 00100
%v0 00100
%v1 00100
$gp 11100
$sp 11101
$fp 11110
$at 00001
$k0 11010 INVALID
$k1 11011 INVALID
```

1.Output of Bubble sort code of MIPS

```
1 1 beq
13
result1
LUIN: 33
instr: 00100010000100000000000000000001
imm:   0000000000000001
I00100010000100000000000000000001
10000
3 3 addi
1
result4
LUIN: 34
R10001010101101011010100000
10101
0 0 sub
result0
LUIN: 35
instr: 00010010101000001111111111101010
imm:   1111111111101010
I00010010101000001111111111101010
10101
0 0 beq
-22
result1
LUIN: 14
instr: 00010010000011110000000000010101
imm:   0000000000010101
I00010010000011110000000000010101
10000
4 4 beq
21
result1
LUIN: 36
Register Memory:[0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 40, 1, 2, 1, 4, 4, 1, 0, 44, 0, 0, 0, 0, 44, 0, 0, 0, 0, 0, 0]
Data Memory: {0: 5, 4: 3, 8: 2, 12: 1, 16: 4, 20: 0, 24: 0, 28: 0, 32: 0, 36: 0, 40: 1, 44: 2, 48: 3, 52: 4, 56: 5, 60: 0, 64: 0, 68: 0, 72: 0, 76: 0, 80
: 0, 84: 0, 88: 0, 92: 0, 96: 0}
clock cycles: 1085
```

Pipelined Processor

All the code is similar to the non-pipelined code until the istru_decode method. Now, starting from the execute method

The instruction cycle, also known as the fetch-execute cycle, is the process by which a CPU executes a single instruction. The cycle consists of several stages, including fetching the instruction from memory, decoding the instruction, executing the instruction, and potentially writing back the result to memory or a register.

The execute stage is where the CPU actually performs the computation or operation specified by the instruction. This stage involves carrying out the necessary calculations or data manipulations

If the instruction requires accessing memory, the CPU enters the memory access stage. For instructions like load and store, the CPU may need to read from or write to memory to fetch or store data

In the write back stage, the result of the executed instruction is written back to memory or a register. This stage is necessary if the instruction modifies data that needs to be stored for future use

Overall, the instruction cycle is a fundamental concept in computer architecture and is crucial for understanding how CPUs execute instructions.

Then, the LUIN Stack is initialised. It contains the first LUIN of that branch. It is then incremented or decremented accordingly.

The LFIN,LDIN,LEIN,LMIN,LWIN inside the method execute. Pipelined registers are also initialised to ensure the storage of the correct numbers.

We have initialised sources 0,1 and 2 where source 0 is no data hazard detected where source 1 is Data hazard with rs of current instruction = destination of the previous instruction source 2 is Data hazard with rt of current instruction = destination of the previous instruction. If the instruction is of R type, Then add,sub,slt are included.

The I type instruction is similar to non pipelined code. The Write back function handles the writeback stage, updating the register memory (Reg_memory) based on the instruction type and result. It returns the updated register memory.

The Mem function handles memory access for load (lw) and store (sw) instructions. It updates the data memory (Data_memory) and returns a tuple containing the updated data memory and a loader value.

Now, running a while loop for all the instructions, We have to check each one of the fetching, decoding,ALU, memory access.

If LFIN equals the length of the instruction memory, then the instruction fetch ends. We have to do similarly for all other instructions as well.

If there is not, then the result is directly printed.

If beq or bne is present, flushing I required to ensure the jump is properly done.

This is done similarly for others, LEIN,LMIN,LWIN as well. The values are written back as well. Clock cycles are incremented by 1. The latest instructions are updated by incrementing 1. While removing a beq instruction or a jump instruction. We decrement the LUIN,LFINI,LEIN,LMIN,LWIN and here, some instructions get skipped. If the instruction is beq and no stall is present then the decrementation happens. Now finally the clock cycles get updated and the hazards are taken care(data,control and load as well as for beq). Finally the clock cycles and the output get printed.

OUTPUT

Bubblesort

```
Reg Memory:[0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 40, 1, 2, 1, 4, 4, 1, 0, 44, 0, 0, 0, 0, 44, 0, 0, 0, 0, 0, 0, 0]
Data Memory: {0: 2, 4: 1, 8: 4, 12: 3, 16: 5, 20: 0, 24: 0, 28: 0, 32: 0, 36: 0, 40: 1, 44: 2, 48: 3, 52: 4, 5
6: 5, 60: 0, 64: 0, 68: 0, 72: 0, 76: 0, 80: 0, 84: 0, 88: 0, 92: 0, 96: 0}
```

Fibonacci

```
Reg Memory:[0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 55, 89, 89, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0]
Data Memory: {0: 0, 4: 0, 8: 0, 12: 0, 16: 0, 20: 0, 24: 0, 28: 0, 32: 0, 36: 0, 40: 0, 44: 0, 48: 0, 52: 0, 5
6: 0, 60: 0, 64: 0, 68: 0, 72: 0, 76: 0, 80: 0, 84: 0, 88: 0, 92: 0, 96: 0}
```