Assignment-8 COL216

The pipelined version of the processor was implemented with the following stages:

1. Instruction Fetch (If)
2. Instruction Decode (Id)
3. Instruction Execution (Ex)
4. Memory Access (Mem)
5. Register File Write Back (Wb)

A do-while construct is used for the implementation and within the loop, there are sections for each stage in the order opposite to the one mentioned above. Along with sections, intermediate registers between the stages are modeled by using vectors using the names of the stages to the left and right e.g. IfId, IdEx, ExMem, and MemWb.

Firstly MIPS format instructions are read from the .asm file and processed and then stored in a vector "input". From the input vector instructions at the index corresponding to the program counter (pc) is fetched during the Fetch Stage. The detailed description of the stages is as follows:

1. **Instruction Fetch:** In this stage instructions corresponding to the index pc are fetched. Here each instruction is a vector of strings. The vector "input" is a vector containing these vector<string> instructions. If the pc has exceeded the size of the vector, then a vector with an empty string is inserted to the IfId vector. Thereby denoting the NOP operation for this stage. The pc is incremented if its value is less than n. Before inserting the new instruction into the IfId register, the IfId vector is first cleared.

2. **Instruction Decode:** In this stage, firstly the condition of data hazard is checked for the individual instructions as it might be the case that for the

execution of this stage, certain register values are needed and that would be written by any further instructions.

If there exists any data hazard, the pipeline is stalled, i.e. The rest of the code in the loop is exited, and therefore the pipeline is stalled. The values which are checked are the ones that are from the register ExMem and MemWb.

If there isn't any hazard, the pipeline proceeds as it is and in this stage, the values are fetched from the register file "file_array". The fetched values are then inserted into the intermediate register IdEx. For the instructions which involve potential branching, a stall (NOP instruction) is introduced by inserting a vector with an empty string in the IfId register and exiting the code for the Fetch stage by invoking "continue". This is done because the decisions related to branching are derived in the Execute stage of the pipeline.

3. **Instruction Execute:** In this stage, the arithmetic operations are carried out for the instructions. They involve arithmetic operations for arithmetic instructions. Offset calculations for load word and save word, and comparison, checks for the potential branching instructions. For the arithmetic instructions, the computed result is then stored along with the address at which it needs to be written is then inserted into the register ExMem. For instructions that involve branching the value of the pc is changed in this stage if the required condition is satisfied otherwise, the pipeline proceeds as it is.

4. **Memory Access:** In this stage, all the transactions involving the data memory are handled. Only LW and SW instructions read and write the values into the data memory. For all the other instructions, the values are simply passed on to the register MemWb. For LW the result of the memory read operation is passed along with the address at which it needs to be written in the memory.

5. **Register File Write Back:** In this stage, the values are written back to the register. For the arithmetic instructions, it would be the result of the computation in the execution stage. For LW, it would be the result that was read from the data memory in the preceding stage. For the JAL, the value of the register RA would also be written in this stage.

For the registers MemWb, ExMem, and IdEx, a value of -1, indicates that there isn't any instruction in progress; in other words, it is a NOP instruction. For the register IfId, a vector with an empty string would denote NOP instruction.

The exit condition for the loop is when the pc becomes equal to n+4 where n is the number of instructions in the input. During intermediate NOP instructions, the value of the PC isn't changed.

Test Cases:

The implementation has been tested rigorously. Some of the test cases are as follows:

```
ADD $t4,$t0,$t1
J 4
SUB $t2,$t3,$t1
SLL $t7,$t1,2
SRL $t5,$t1,1
SUB $t6,$t2,$t5
ADD $t3,$t4,$t1
```

In this test case, the arithmetic instructions have been tested along with J. There aren't any conflicts, therefore the number of cycles required for 5 instructions is 12.

```
SW $t1,1024($t2)
ADD $t4,$t1,$t0
SUB $t5,$t4,$t2
SLL $t6,$t5,2
SRL $t7,$t6,2
LW $t3,1024($t2)
SLL $t6,$t3,2
SRL $t7,$t3,2
```

There are data hazards, and for each data hazard, there is a stall of 2 cycles for consecutive instructions and 1 cycle for alternate instructions. The number of cycles required for 8 instructions is 20.

```
ADD $t4,$t0,$t1
BGTZ $t4,3
BLEZ $t4,4
BNE $t4,$t1,6
SUB $t5,$t1,$t2
SLL $t6,$t1,2
SRL $t7,$t1,2
```

In this test-case other branching instructions have been tested such as BGTZ, BLEZ, BNE etc.

```
ADD $t4,$t0,$t1
ADD $t6,$t2,$t0
JAL 4
J 13
ADD $t4,$t0,$t4
ADD $sp,$t6,$sp
SW $ra,0($sp)
JAL 12
```

```
LW $ra,0($sp)
SUB $sp,$sp,$t6
JR $ra
ADD $t4,$t0,$t4
JR $ra
BEQ $t1,$t7,15
SUB $t4,$t0,$t4
```

Here, the total number of instructions is 14 and the clock cycles required are 28. Thus, on increasing the number of instructions is increasing the average number of instructions per cycle as well

The test cases are MIPS instructions and are present along with the source code. In order to execute the script the terminal command would be
./a.out tc2.asm, if tc2 needs to be used.

Manav Modi	2018CS10353
Param Khakhar	2018CS10362