REPORT – Assignment 8

Design Details:

Pipelined Latches

The pipeline was implements using 4 classes:
- IF_ID_latch: Carries the information required by ID stage, updated by the IF stage
- ID_EX_latch: Carries the information required by EXE stage, updated by the ID stage
- EX_MEM_latch: Carries the information required by MEM stage, updated by the EXE stage
- MEM_WB_latch: Carries the information required by WB stage, updated by the MEM stage

Information here means: control signals and data needed by the stage.

A pipeline class was implemented which had 2 object of each of the above 4 classes which carried the information available to a stage and updated by the previous staeg in each cycle.

Pipeline Stages

Parallel processing of each stage is implementd using the 2 objects of each class: 1 object is updated by a stage which is called the temp object and the other object's information is used by the next stage. Before the beginning of the next cycle the temp objects are assigned to the main objects which will be used by the stages for getting the control signals and .

Each stage is implemented as a method of the pipeline class:

1. IF stage - Fetches the instruction in pc and updates the temp_IF_ID_latch
2. ID stage -
    - 2.1 Updates the temp_ID_EX_latch
    - 2.2 Checks whether we have to stall or not:
        - 2.2.1 If we have to read from a register that was updated (written or loaded) in the previous cycle then we need to stall for 2 cycles and also make sure that registers are written by the WB stage first and then read by the ID stage.
        - 2.2.1 If we have to read from a register that was updated (written or loaded) in the previous to previous cycle then we need to stall for 1 cycles and also make sure that registers are written by the WB stage first and then read by the ID stage.
        - 2.2.3 If this is a conditional branch instruction then we need to inform the IF stage to not fetch the next instruction as it may be the wrong instruction (if the conditional branch is taken) therefore ID stage sends a signal to the IF stage to stall for 1 cycle.
    - 2.3 If the branch is an unconditional branch then we pc is updated.
3. EXE stage -
    - 3.1 Updates the temp_EXE_MEM_latch
    - 3.2 Performs all the ALU operations including condition checks for conditional branch instuctions.
    - 3.3 If the condition for the conditional branch instruction evaluates to true then the branch is taken and the pc is updated.
4. MEM stage -
    - 4.1 Updates the temp_MEM_WB_latch
    - 4.2 Performs the memory access for load instuction and store instruction.
5. WB stage -
    - 5.1 If any register is to be updated then it is done by this stage.

Preventing Hazards

1. When a register was updated in the previous clock cycle then ID stage stalls for 2 cycles.

2. When a register was updated in the previous to previous clock cycle then ID stage stalls for 1 cycle.
3. If a conditional branch is encountered in the ID stage then the IF stage stalls for 1 cycle.
4. The EXE stage processed before the IF stage as if a conditional branch is taken then pc can be updated in the EXE stage and the IF stage can fetch that instruction in the same cycle preventing the need of another cycle stall.
5. ID stage processes before the IF stage as if there is an unconditional branch then pc can be updated in the ID stage and the IF stage can fetch the correct instruction in the same cycle preventing the need of another cycle stall.TE
6. WB stage processes before the ID stage if the register to be written is also read in the same cycle then we can prevent hazardous data by writing in the first half cycle and reading in the second half cycle.

Stalls

In the ID stage if we find that a register to be read was updated in the previous cycle or the previous to previous cycle or this instruction is a conditional branh instruction then we update the stall control signals accordingly.
By “a stage stalls” we mean that that stage passes null to the latch of the next stage so that no hazardous computation is performed and preserves its latch’s signals for the required number of cycles, this is done by not updating the latch of this stage at the end of this clock cycle also in forming the previous stages to not process further. After the stalling for the required number of cycles the stage can process according to the signals in its latch without any hazards.

Testing

TestCases: ($a1 <= 1 in all the testcases)

Register updated in previous cycle (2 cycle stall + read after write from registers)

```
add $a0 $a1 $a2
sub $t0 $a0 $a1
```

****
Register updated in previous to previous cycle (1 cycle stall + read after write from registers)

```
sub $sp $sp $a1
lw $a0 0 $sp
add $a2 $zero $zero
sw $a0 0 $sp
```

****
Conditional branch instructions (1 cycle stall)

```
add $a0 $zero $zero
beq $a0 $zero exit
add $a0 $a0 $a1
exit:
```

****

```
        reg[reg_stoi("$a1")] = 1;
        reg[reg_stoi("$at")] = 1;
        reg[reg_stoi("$v1")] = 10;
        reg[reg_stoi("$a3")] = 10;
```

Finding sum of first $a3 numbers:

```
add $a3 $a3 $a1
add $a3 $a3 $a1
add $a3 $a3 $a1
add $a3 $a3 $a1
jal findsum
j exit
findsum:
sub $sp $sp $a1
sub $sp $sp $a1
sw $a0 0 $sp
sw $a3 1 $sp
add $a0 $zero $zero
for:
add $a0 $a0 $a3
sub $a3 $a3 $a1
bne $a3 $zero for
add $v0 $a0 $zero
lw $a3 1 $sp
lw $a0 0 $sp
add $sp $sp $a1
add $sp $sp $a1
jr $ra
exit:
```

Result for assignment:
Toal Cycles: 119
No of instruction executed: 59
Average instruction per cycle: **0.495798**

****

```
//Storing first $v1 fibonacci numbers in the momory from (4095..4095 - $v0 + 1)
$a1 <= 1, $a1 <= 1

label1:
sub $sp $sp $at
sw $a0 0 $sp
sub $sp $sp $at
sw $a1 0 $sp
add $v0 $v0 $at
label2:
add $a2 $a1 $a0
sub $sp $sp $at
sw $a2 0 $sp
lw $a0 1 $sp
lw $a1 0 $sp
add $v0 $v0 $at
beq $v0 $v1 label3
j label2
label3:
```

Result for assignment:
Toal Cycles: 129
No of instruction executed: 76
Average instruction per cycle: **0.589147**