IHSAN DOGRAMACI BILKENT UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

CS 224 – COMPUTER ORGANIZATION

DESIGN REPORT

LAB 05

SECTION 03

EMRE KARATAŞ

22001641

30.11.2022

QUESTION B:

RTL Design for "sracc":
IM[PC]
RF[rd] <- RF[rd]+ RF[RF[rs] >> RF[rt]]
PC <- PC + 4

Compute-use hazard:
-Data hazard type
-Execute, Memory or WriteBack stages are affected.
Load-use hazard:
-Data hazard type
-Fetch, Decode, Execute, WriteBack stages are affected.
Load-store hazard:
-Data hazard type
-Fetch, Decode, Execute WriteBack stages are affected.
Branch hazard:
-Control hazard type
-Fetch and Decode stages are affected.

QUESTION C:

Compute-Use Hazard:
The reason of this hazard is that if there is data dependency in either the next instruction or in two instructions later. If we are writing a result that we have computed to a register then using that register in next instructions afterwards, this hazard occurs. For this case, If the "rd" value of our current instruction is the same as the "rs" or "rt" values of the next instruction or the instruction after that, it is said that there is data dependency. This hazard is solved by data forwarding whenever there is data dependency in the process. Data that is either in Memory stage or WriteBack stage is forwarded to Execute stage to avoid this hazard.

- If the data dependency is in the next instruction, data that is in the Memory stage is forwarded to the Execute stage.
- In other case, if the data dependency is 2 instructions later, data that is in the WriteBack stage is forwarded into the Execute stage.
- There is another case as well, if data dependency is 3 instructions later, there won't be a hazard because the computed result will already be written in the register file.

Load-Use Hazard:
It is caused if there is data dependency in either the next instruction or in two instructions later. If we are writing data with the load word instruction to a register then using that register in next instructions, this hazard occurs. Data forwarding on a load-use hazard is not possible therefore, stalling is required to solve this hazard.

- If the "rs" value in the Decode stage equals the "rt" value in the Execute stage or the "rt" value in the Decode stage equals the "rt" value in the Execute stage and the MemtoRegE signal is 1, we have to stall the processor.

To stall the processor, program count is stopped. We stall the Fetch stage, Decode stage and we have to flush the execute stage. By flushing the Execute stage,and stalling Fetch and Decode stages, the instruction flushed will simply be repeated in the next clock cycle, but this time with correct data. Data is also forwarded if necessary.

Load -Store Hazard:

Same process applies with load-use hazard. That is, this hazard can be solved by using stalling.

Branch Hazard:

This hazard occurs because branch is not determined until the 4th stage of the pipeline. In this implementation the check operation is done which shows whether we branch or not in the Decode stage. There is operator which checks if "rs" and "rt" of the register file are equal and sends the result to the Fetch stage to update the pc if they are equal. Furthermore, if there is data dependency, we have to forward the data in the Memory stage to the Decode stage in order to avoid another hazard. If the branch is taken, the processor flushes the instructions that shouldn't be taken to avoid any hazard. This mechanism saves clock cycles most of the time and makes the processor more efficient.

QUESTION D:

Data Forwarding Logic:

ForwardAE: (inputs are: rsE, WriteRegM, RegWriteM, WriteRegW, RegWriteW)

```
if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
        then ForwardAE = 10
else
        if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
              then ForwardAE = 01
        else ForwardAE = 00
```

ForwardBE: (inputs are: rtE, WriteRegM, RegWriteM, WriteRegW, RegWriteW)

```
if ((rtE != 0) AND (rtE == WriteRegM) AND RegWriteM)
      then ForwardBE = 10
else
       if ((rtE != 0) AND (rtE == WriteRegW) AND RegWriteW)
            then ForwardBE = 01
       else ForwardBE = 00
```

Logic Operation for forwarding in Branch Hazard:

ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM

Stalling Logic:

(Inputs are: rsD, rtD, rtE, MemtoRegE, BranchD, RegWriteE, WriteRegE, MemtoRegM, WriteRegM. Outputs are: StallF, StallD, FlushE)
StallF = (((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE) OR (BranchD AND RegWriteE

AND (WriteRegE == rsD OR WriteRegE == rtD) OR BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD))

StallD = ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE OR (BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD) OR BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD))

Logic Operation for Flushing:

FlushE = ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE OR (BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD) OR BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD))

QUESTION E:

sracc is an R-type instruction which uses an additional read port as well as an adder to compute the result. First of all, it will cause a **compute-use hazard** as the result of the sracc instruction will be saved to the rd register in the write-back stage and the next instruction won't be able to use its value right afterwards if it uses the rd register of the sracc instruction as a rf or rs register value.
Since the sracc instruction uses an adder to compute the result after the ALU computation after the execution stage. This adder overlaps with the **memory stage** which implies that the adder computation is not completed before the memory stage hence **we cannot forward the data from the execute stage to the next instruction** as it will use incorrect data values in its calculations causing another **data hazard**.
Solution:
The solution is to use **forwarding** to use the result of the adder from the memory and write-back stage and use it in the second and third instructions. For the instruction right after the sracc instruction, we need to stall the pipeline for one clock cycle. Then we can use forwarding to send the data value to the execute stage of the instruction. Furthermore, we will need to use an additional signal in the hazard unit which will be inputted to the rs/rd mux as well as the adder mux. This will resolve the issues that we are facing in due to **compute-use hazard**.

SRACC HAZARDS:

```
addi $s0, $zero, 0x0003
addi $s1, $zero, 0x00AA
addi $s2, $zero, 0x0005
sracc $s2, $s1, $s0
sw $s2, 0 ($s1)
```

COMPUTE-USE HAZARDS:

```
add $s1, $zero, $zero
addi $s0, $zero, 0x00AA
addi $s1, $s0, 0x0003
and $s2, $s0, $s1
```

LOAD-USE HAZARDS:

```
//Assuming word stored at $t0 = 0xFFAA
addi $s0, $zero, 0x0010
lw $s0, 0 ($t0)
add $s2, $zero, $s0
```

LOAD-STORE HAZARDS:

```
addi $t0, $zero, 0x0010
addi $t1, $zero, 0x00CC
sw $t0, 0($t1)
lw $s0, 0 ($t1)
add $s0, $zero, $s0
```

BRANCH HAZARDS:

```
addi $s0, $zero, 0x0000
addi $s1, $zero, 0x0002
beq $s0, $zero, 0x0002
addi $s1, $s1, 0x0010 //Not computed
add $s1, $0, $s1
```

NO HAZARDS:

```
addi $t0, $zero, 0x002
addi $t1, $zero, 0x00F0
addi $t2, $zero, 0x0000
addi $t3, $zero, 0x0112
addi $t4, $t0, 0x0002
sw $t1, 0 ($t0)
sracc $t3, $t4, $t0
beq $t2, $zero, 0x0004
addi $t2, $t2, 0x0010
slt $t5, $t1, $t3
lw $t6, 0 ($t0)
```