ADVANCED COMPUTER ARCHITECTURE - HOMEWORK I

Student: Francisco Javier Blázquez Martínez
E-mail: francisco.blazquezmartinez@epfl.ch
Sciper: 331229

- **EXERCISE 1: Pipeline and Register Renaming**

1.- In the third stage the R10000 writes the fetched and decoded instructions in the queues and, at the same time, prepares the execution of instructions in the queue which have now their operands available. It starts the execution by reading these operands, what is done in the second half of the cycle. These implies a faster execution of the instructions (we save one stage for reading the register file), but also could imply setup and hold violations when operating at high frequencies.

2.- When operating with registers we always face the same issue, the writing operation is destructive, makes us loose the previously stored value with no option to get it back. And of course we don't want to wait for writing in a register. This is even more important in our case with out-of-order execution and several pipelines executing instructions at the same time. Having another bit in the register representation after renaming is because we have internally twice the number of visible registers to solve this issue (logical registers doesn't mean a single physical reguster allows also affirm that execution doesn't mean graduation of the instruction).

3.- Specify the role of the following components:

queues:
Address, integer and FP queues for keeping every type of active instruction.

active list:
To track the assignements of registers, needed for knowing with registers are required in case a instruction doesn't graduate and to properly free the register that are not required anymore.

map tables:
Maps logical to physical registers.

4.- Role of the following components in Picture 5:

Rdy: Bits that indicate whether the different operands were ready or not at decode stage.
OpA: Physical register (after renaming) of first operand
OpB: Physical register (after renaming) of second operand
OpC: Physical register (after renaming) of third operand
Dest: Physical register (after renaming) of result
Old Dest: Phisical register that would be overwritten by the result (same logical register than Dest).
Log Dest: Logical register of result
Tag: Position in the active list
D: Done bit

5.- Memory bits required for:
We consider ordered data structures so that there is no need to keep any key for organizing the data.

FP queue = 32*(#Tag + #Rdy + #Operands + #Destination) = 32x(5 + 3 + 6x3 + 6)
Active list = 32*(#OldDest + #Log Dest + #Done) = 32x(6 + 5 + 1)
FP busy bit table = 64
FP register map table = 32x6
Free register list = 32x6

6.- Number of read ports required for the components:
FP busy bit table = 4, because we decode 4 instructions at the same time
FP map table R= 5, because we could have the case of the five pipelines at stage 3 at the same time
FP map table W= 2, because in stage 5 and 7 we might need to graduate two instructions

7.- How each kind of dependency is handled:

RAR = Not a problem
WAR = Not a problem
WAW = Not a problem
RAW = The instruction doesn't get into the execution stages unless all the operands are available, this is controlled with the Rdy bits and the structures for keeping them updated (busy bit table).

8.- See *Simulation.ods* attached

- **EXERCISE 2: Exception Handling**

1.- This would be trivial if it wasn't because the R10000 executes instructions out of their sequential order and it has several segmentated pipelines.

When an exception arises, we have to get the processor inner state as it would be immediately before executing the instruction that caused the exception. Four the R10000 it means that we have to cancell the posterior instructions already fetched and undo the changes that these posterior executions made if they were executed out of order, before the instruction that caused the exception.

The execution of posterior instructions can be undone thanks to register renaming, we can abort any instruction not graduated (executed and also all the previous ones) thanks to having twice the number of visible registers. The abortion the posterior fetched instructions is possible thanks to the active list.

2.- With the help of the file *Simulation.ods* we see that instruction 6 is in stage 4 (when it multiplies) at the cicle 9. If an exception arises, we have to "undo" all the posterior instructions but still execution the previous ones, in our case, we would mark as executed (D tag) all instructions later than 6 and we would have to restore the register map table of those instructions after 6 which had finished their execution already. We can do this only changing their entry with key Log Dest in the register map table by Old Dest.

- **EXERCISE 3: Branch Prediction**

1.- “The prediction uses a 2-bit algorithm based on a 512-entry branch history table.” This is, we keep a history table with key the bits 11:3 (up to 512 different) of the latest branch instructions address in memory and value the memory address of the instruction we jump to. It’s dynamic because the this history table is continuously updated and the decission is taken at run time.

2.- Before taking the branch the processor saves its state in a branch stack. This branch stack can keeps up to 32 processor states identified by a 4-bit branch mask, if the stack gets full, the processor continues decoding until it encounters the next branch instruction and if this scenario occurs, the processor has to wait until the resolution of one of the previous branches.
In the case that a prediction was incorrect, the processor immediately aborts all instructions fetched along the mispredicted path and restores its state from the branch stack. This is possible because the 4-bit branch mask of the branch the instruction belongs is part of every decoded instruction. After this it jumps to the instruction the branch was really refering to.

3.- “When the program execution takes a jump or branch, the processor discards any instructions already fetched beyond the delay slot. It loads the jump’s target address into the program counter and fetches new instructions from the cache after a one-cycle delay. This introduces one “branch bubble” cycle, during which the RlOOOO decodes no instructions.”

The jumps and branches are resolved in decode stage. This means, at the point when they are resolved, next instruction has already been fetched, and obviously it shouldn’t be executed if we are finally doing a jump and breaking the sequential execution. This is why a NOP was added after the branch instruction in the example.

However, we can see that the instruction #00000009 could be fetched after the branch without altering the behavour of the program since it ($I3) is not involved in the branch decission.