ADVANCED COMPUTER ARCHITECTURE - HOMEWORK II

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

- **EXERCISE 1: Speculation**

1.- Main features of the IA64 architecture for speculation

Predication:
Predication is a way to introduce No Operations statically and dynamically at the same time, this is, the compiler (statically) introduces instructions that, depending on the state of the machine (dynamically), will act as NOPs or will execute some operation. This is obtained by marking the execution of these instructions as dependent on some flag (bit in the processor) which can be set dynamically.

The idea behind predication is **avoiding branches**. For example, instead of branching to the if or else clause, we can execute both in parallel canceling the non-desired branch depending on the evalutaion of the condition. The key why this is not a loss of performance is because these instructions are executed in **parallel**.

This is not a really a type of speculation, we wi'll never undo any of this executions, it's only that we have defined the short-step semantics of the branches in a different way than what we are used to.

Control speculation:
Even with predication, the compiler doesn't remove all the branches. Therefore IA-64 compiler can provide out-of-order instructions to increase parallelism and branches. This could mean that an instruction in a posterior execution block could rise and exception when the previous block hadn't been fully executed yet, this is, **imprecise exceptions**.

This is avoided by making operations able to handle an special value "Not A Thing" so that in case an exception arises this can be propagated, continuing the normal execution of the program and therefore executing all the previous instructions up to the one that rised the exception. However, the IA-64, for achieving more efficiency, allows to move load instruction to previous blocks and these could also generate an exception. For solving this issue, IA-64 supports **speculative loads**, this instructions attempts to execute the load operation and, if an exception arises, it loads an NaT, what would be detected in the proper execution block by a check instruction.

Data speculation:
Again, for a better efficiency, IA-64 compiler tries to place load instructions as soon as possible maintaining the semantics of the code. This loads can even be placed before previous store instructions which we don't if they will modify the same memory address (leading to a read-before-write incoherence).

In the same way than before, this is solved by adding a pair of instructions **advanced load** - check. Unlike before, NaT is not a valid solution anymore but this goes through keeping a data structure (ALAT, explained in the next point) with the advanced loads that could lead to a conflict.

2.- Specific hardware support for speculation

Predication: Predicate Registers, 64 one-bit registers used as flags for the conditional execution.

Control speculation: Relies on NaT special value and the propagation of this in the execution units

Data speculation: Advanced Load Address Table (ALAT), to control advanced loads and previous stores collisions so that the check instruction can know if the load has to be done again.

- **EXERCISE 2: Software Pipelining and Register Model**

1.- Arquitectural support for software pipelining

LC register: To maintain the loop count.
EC register: To know when the loop has finished.
Special loop branches: To implement the loop logic depending on the stage (starting, finishing...)
RRB register: To automatically rename registers at each pipeline step and avoid collisions.
PR[16:63]: To label the pipeline stage as valid or invalid

2.- Software compiling I

```
# Code
ldr r1 = [<ptr1 initialization>]      // We get ptr1 and ptr2 addresses to registers
ldr r2 = [<ptr2 initialization>]      // Important! These have to be from the first 32

mov LC = 99                           // We then initialize LC=Iters-1 EC=Stages+1
mov EC = 3

// Assumption: load can be done in a single stage
loop:
stage 1: (p16)  ld4 r32 = [r1], 4    // r32 = *ptr1; ptr1++
stage 2: (p17)  st4 [r2] = r33, 4    // *ptr2 = r33 (= previous r32); ptr2++
                br.ctop loop;;
```

3.- Software compiling II

The previous code could be used for copying a list, the code in this subsection could be used for this purpose but it's more general, it could be used for copying a list (independently of the list size) or to deplace left from a given element of a list in itself.

Unlike the previous example, here we can't use LC register (and therefore PR flags for conditional execution) as we don't know the list size. Therefore we cannot use software pipelining and we would have to create a non-pipelined loop, what means a loss of performance:

```
      cmp p0 = r1 == Null
(p0) br endloop
      ld4 r32 = [r1], 4    // r32 = *ptr1; ptr1++
      st4 [r2] = r32, 4    // *ptr2 = r32; ptr2++
endloop:
```

4.- Register model

Physical register model:
128 General registers
32 static
96 stacked (can be renamed under software control)
+ Register Stack Engine (saves in memory and restores the register file)

Logical register model:
∞ registers

Parameter passing convention:
Typically stack structure for parameter passing criteria, as in the MIPS but, in this case, parameters to the next subrutine are not placed in the general registers but at the top of the register stack. This way when another subrutine is invoked the local registers (bottom part of the stack execution context) are hidden, and the out registers, which include the parameters for the subrutine, are renamed to start at R32 and placed on the bottom of the new stack context.

For the case of calling the example function **objA foo(int a, int b, int c)** we know that the calling method stack context would have at least 3 out registers for the three parameters which would become part of the new local stack context of **foo**. And we know that this local context would be at least 32 bytes = 4 registers because it's the size of the returned structure. If **foo** invokes some subrutine it's stack context would also have an out section with space for the parameters of the invocation.
(Figure 14 in *"Introducing the IA-64 Architecture"*)

Lets analyze now the register model from several perpectives, naming some advantages and disadvantages:

Programmer, Compiler and Hardware:
→ For the programmer the register model is invisible/transparent
→ For the compiler we have the advantage of "infinite register space" for renaming
→ For the hardware the RSE has to deal with stores and restores of the registers

Microarchitectural complexity:
→ RSE logic need to be implemented
→ Logic for registers store/restore in/from memory in parallel with the program execution
→ Syncronization of RSE and program execution
→ Compiler deals with the complexity of register renaming for parallel and OOO execution
→ Less complexity than in a superscalar OOO processor, compiler takes most of the tasks

Performance:
→ Much slower than a classical register file because might require many accesses to memory

A possible implementation for this register model of the addressing schema could be with relative addresses to a register indicating the stack size, this provides immediate register renaming following the procedure call conventions specified just by updating the stack size and would only require an adder to get the absolute address.