Lab 2
The IA64 Architecture And Its Implementation

This Lab is based on the following documents: *Introducing the IA-64 Architecture*, *Itanium Processor Microarchitecture*, and *The Intel IA-64 Compiler Code Generator*. All of them can be found on the web page of the course.

1 Speculation

1. What are the 3 main features of the IA64 architecture for speculation? Describe each of them and explain why they are useful to achieve high performance. Note that such features may not explicitly be named "speculation" - speculation occurs any time the processor executes instructions which may not be necessary or useful to the program.

   - ***Predication.*** *Predication is a mechanism that allows any instruction to be executed speculatively and be graduated only if an associated predicate is true. This mechanism is particularly useful to better exploit the available parallel functional unit. Moreover, one can avoid simple if-then-else structures that traditionally involve the use of expensive branches. Using predication, both the if and else blocks can be executed in parallel once the processor has computed the predicate that tells which path to execute. Since the only instructions that will graduate are the one with a predicate whose value is true, only instructions along the correct path will change the state of the processor; the other ones (i.e. the ones that needn't have been executed) will be discarded. By speculatively executing blocks in parallel, one can avoid branch instructions and avoid their high cost. This mechanism is also used in software pipelining to minimise the overhead in term of code size generally associated with this technique (see the next question for details). Finally it allows more aggressive code scheduling technique and exposes more instruction level parallelism to the compiler by allowing optimisations to work on larger blocks.*
   - ***Control speculation.*** *This mechanism allows to speculatively move a load instruction across basic block boundaries. It introduces a special* `ld.s` *instruction that perform the load but do not raise exceptions. A* `chk.s` *instruction is inserted by the compiler where the load is not speculative anymore (i.e. at its 'original' position). This instruction will branch to compiler-provided fix-up code if an exception occurred during the load. Since loads are really time consuming and often precede long calculation chains that depend on them, the earlier the load can be scheduled (by the compiler) the less instructions might wait because they depend on that load. Once more, this technique allows aggressive code scheduling techniques.*

- ***Data speculation.*** *This mechanism is somewhat similar to control specification. It allows the compiler to speculatively schedule a load instruction before store instructions that may access the same memory cell. For this purpose, the compiler uses advanced load instruction (*`ld.a`*) and the corresponding check instruction (*`chk.a`*) that will branch to fix-up code if necessary because of some store instruction modifying a loaded memory cell having been executed in between. As the two other techniques, it enables more aggressive code scheduling. Indeed a lots of programs use complex pointer-based data structures for which it is difficult (or even impossible) to get precise alias analysis, resulting in conservative decisions at compile-time. Typically, a load can hardly ever be moved before any stores at compile time without data speculation support in hardware.*

2. What is the specific hardware support needed in order to use these features and that is implemented in Itanium?

   For predication, 1-bit predicate registers are needed. Most instructions can then be associated to a predicate register to know whether they should graduate. Consequently, instruction format must include an (optional) predicate register field.

   To support control speculation, all general purpose registers have an extra bit, called NaT. If an synchronous exception occurs due to the speculative load, the NaT bit of the target register is set and the exception is discarded. When the speculative check instruction is executed, it checks whether the NaT bit of the given register is set and if it is the case, it branches to fix-up code.

   For data speculation, we need to keep track of the destination register, the address and the size of every advanced load. For that purpose, an advanced load address table (ALAT) is needed. When a `ld.a` *occurs, a new line is added into the ALAT. When a store occurs and it accesses a position that has been loaded in advance (i.e. whose address overlap an entry in the ALAT), the corresponding line in the ALAT is erased. Finally, when a* `chk.a` *occurs, the fix-up code is executed if there are no corresponding lines in the ALAT (because a store erased it).*

2 Software Pipelining and Register Model

1. Describe the specific architectural support introduced in IA64 for software pipelining. Explain why it makes software pipelining less expensive and easier.

   Dedicated instructions for loop semantics are added, such as `br.ctop`*. To support them, special registers for counting loop iterations (*`LC`*) and number of stages in the pipeline (EC) are provided. Moreover predicates are used to avoid duplicated code for the prologue and the epilogue. Finally, rotating registers (GP and predicates) are provided to support renaming among iterations: each rotation advances the pipeline by one stage.*

2. Considering the following program, write the assembly code that an Itanium compiler doing software pipelining would generate. Comment each

line. The Itanium instruction set can be found in Intel Itanium Architecture Software Developer's Manual Volume 3.

$int * ptr1, *ptr2, i;$
$ptr1 = R1;$
$ptr2 = R2;$
$for(i = 0; i < 100; ++i)$
{
$\quad *ptr2 = *ptr1;$
$\quad ++ptr1;$
$\quad ++ptr2;$
}

`mov LC = 99`	Initialise the loop counter
`mov EC = 3`	Initialise the epilogue counter
`mov pr.rot = 1 << 16`	Initialise the rotating predicate registers
`loop:`	
`(p16) ld4 R34 = [R1],4`	Stage 1: load *ptr1 *Nothing in stage 2 (p17) because of the load latency (2 cycles)*
`(p18) st4 [R2] = R36,4`	Stage 3: store to ptr2
`br.ctop loop;;`	Redo the loop

Note that we (as Intel's compiler does) have not used a rotating register for the addresses.

3. Suppose that the for loop is replaced by a while loop, for example $while(*ptr1 != NULL)$. Explain what kind of changes your code would need.

   In the case of a while loop, the LC register and the associated `br.ctop` *instruction cannot be used anymore because we cannot know at compile time the number of iteration of the loop.*

   Consequently one should use the `br.wtop` *instruction that allows to branch only if a predicate has the value true. The result of the comparison of the while statement must be put in a predicate register that will be used by the* `br.wtop` *instruction. That predicate register must be a one of the rotating predicate registers in order to be able to drain the loop when the loop condition is not true anymore.*

   Moreover, the `ld` *should be made speculative because at the time the condition becomes false, we will have already executed the load in a next iteration (since the test can only occur in late pipeline stages).*

4. Explain the register model for the programmer and the parameter passing convention. Illustrate the concept with the code which calls a C function `objA foo(int a, int b, int c)` where `objA` is a structure which occupies 32 bytes. Details on the calling conventions for Itanium can be found in the Itanium Software Conventions and Runtime Architecture Guide. List advantages and disadvantages of this scheme in terms of (1) programmer/compiler comfort, (2) microarchitectural complexity, and (3) performance. Indicate with a simple diagram how the register addressing scheme might be implemented.

Infinitely many registers are provided to the programmer through the stack engine. Beside the first 32 *registers,* 96 *registers are managed as a stack. The register stack engine (RSE) saves registers to memory when not enough free register are available. Each procedure can allocate up to 96 fresh registers through alloc instructions and deallocate them on procedure return. The interest of such a model is that it results in procedure calls being less expensive by avoiding systematic saving of registers on the (memory) stack. It simplifies the task of the compiler because of the virtually infinite number of registers. Finally, the RSE can be implemented so as to speculatively spill and fill registers in the background using unused memory ports.*