Comparative Architectures 2004 – Paper 7 Question 1 (IAP)

(*a*) register windows. Like SPARC, IA-64 has support for register windows to improve the performance of the procedure call/return sequence by avoiding the need to explicitly spill registers to memory. The IA-64 implementation is more sophisticated in that register windows are not fixed size: A variable number of argument registers may be passed into a procedure (also used for the return value), and a procedure may allocate a variable number of local registers that will be preserved across further procedure calls. Some register values are global and present in all windows. On IA-64 a h/w Register Stack Engine is responsible for lazilly saving/restoring the register state to/from a special stack area.

(*b*) Registers in IA64 have a 65 bit value, used to record when the register is in a 'NaT' state. Registers enter the NaT state when the outcome of an operation is invalid, for example, division by zero, or a load that accessed an illegal address (other architectures would raise an exception in these instances). NaT bits propagate, hence any operation using NaT as source operand will generate a NaT result. This allows checking for such failures to be deferred, reducing the number of checks, which is helpful if failures are rare. The compiler must insert 'chk' instructions and branch to appropriate fix-up code (being careful that it hasn't destroyed operands needed for recovery).

(*c*) The ALAT table allows data speculation by enabling the compiler to hoist loads ahead of potentially conflicting stores (that might store to the same location). The special load instruction places the address being loaded from into the ALAT table. All stores check the ALAT table to see if the address they are storing to is present in the ALAT table, and removes the entry if present. At the appropriate point in program order, the compiler issues a chk instruction to check whether the address is still present, and hence whether the initially loaded value was correct. If not, the load must be re-issued, and potentially fixup code invoked to clean up if the errant value has been used earlier.

(*d*) Register file rotation enables s/w pipelining to be achieved without the code expansion required by loop unrolling. The register numbers referred to in instructions are effectively virtual, and converted to physical registers by adding a base offset, modulo the number of registers being used in rotated mode. At the end of every loop iteration the base is typically incremented. The loop will typically executes the number of times required plus 2x the pipeline depth (once for prologue, once for epilogue). The register number virtualization allows values to be passed between different iterations of the loop, hence enabling multiple stages of the algorithms implementation to be interleaved in the loop body. The predicate registers rotates as well as the GP registers, and in addition are used to control prolog/epilogue execution.

(*e*) Parallel compares enable fast evaluation of complex predicates in 'if' statements. They are an exception to the normal IA-64 rule that instructions in the same parallel group must not have data dependencies. A series of cmp instructions in the same parallel group are allowed to issue in parallel with the same destination predicate registers, with the results combined via a logical operation. e.g.

```
if ( (rA<0) && (rB==-15) && (rC>0) )
/* block */

as:

cmp.eq p1,p0 = r0, r0 ;; // p1 =1
cmp.ge.and p1,p0 = rA,r0
cmp.ne.and p1,p0 = rB,-15
cmp.le.and p1,p2 = rB,10
(p1) br.cond if-block
```