SOLUTION NOTES

Comparative Architectures 2002 Paper 8 Question 10 (IAP)

(a) Even statically scheduled processors have complex issue rules that must be resolved before an instruction is assigned to a pipeline and issued down it. This requirement for complex logic often leads to a requirement for multiple stages to perform the instruction decode, "slotting" and register read.

As well as the requirement for more stages, some designers have deliberately increased the pipeline length in order to reduce the work done by each pipeline stage so that the clock speed can be increased. This will increase the instruction issue rate, increasing the work rate provided instructions don't stall.

(b) Long pipelines are very vulnerable to stalling.

Due to the number of stages before execution, it's likely that the mis-predicted branch penalty on such processors is high. Accurate dynamic branch prediction techniques must be employed, or at the very least, branch hints inserted by the compiler.

One such long pipelines register "values" spend a long time out of the register file. Forwarding paths will be necessary from each of these pipeline stages back to the ALU multiplexors. Since this is a super-scalar machine, we will also need forwarding paths between the various pipelines if we are to avoid long stalls waiting for values to be returned to the register file.

Load data hazards are likely to be a significant problem. Even with forwarding paths in place, there will be a window of several instructions after each load that data-dependent instructions can not issue in (even if the load hits in the L1). The length of the window is exacerbated on super-scalar machines. One possible scheme for reducing the window is to enable the value read from the cache to be used speculatively before tag check has been performed.

A further problem is that if the pipeline stages are too small the overhead of the inter-stage latches may constitute a significant fraction of the cycle time, reducing efficiency.

(c) The register scoreboard tracks whereabouts in the CPU the value associated with an architectural register can currently be found.

For instance, it can indicate that the value currently resides in the register file, and can hence be read directly from there by an instruction requiring it as a src operand in this cycle.

The scoreboard entry associated with a register that is the destination operand for an instruction will be updated as the instruction progresses through the pipeline. The scoreboard will indicate to following instructions whether they can currently access the result, and if so, from where (and hence which forwarding path to use).

For example, a Load will mark the destination register as unavailable for all following instructions until the value has been read from the cache and is available for forwarding. Instructions trying to access an unavailable register will stall until access is available.

As well as RaW hazards, the scoreboard may also be used to prevent WaW hazards these could occur when a long instruction (e.g. multiply) is followed a quick instruction writing to the same register. As well as the src operands, the location of the destination can be checked too. The following instruction can be held back, or a decision taken to nullify the late write to the register file. Since WaW hazards are fairly rare in code the stall approach may be taken at little cost.

(*d*) Arithmetic exceptions can occur very late on in the pipeline, potentially after following instructions have already finished executing and written to the register file.

For architectures requiring precise arithmetic exceptions some mechanism is required to ensure the state of the machine can be "unwound" back to the faulting instruction. One technique would be to avoid issuing following instructions until the preceding one is past the last point when it could fault. This is likely to be highly inefficient.

Many more recent architectures allow imprecise exceptions. For example arithmetic exceptions can be raised any number of instructions after the faulting one. It is down to the compiler to insert Trap Barriers to force the exception to be raised before entering code which will cause permanent side effects if the fault occurred.

IA64 adopts a different approach, storing poison bits in each register to indicate whether the result was valid or whether an exception occurred. Instructions that operate on poisoned inputs generate poisoned outputs, hence propagating the bit. The presence of an invalid value can be checked using a "branch if poison" instruction.