Lecture 6 notes

We can increase ILP by overcoming control dependencies and fetching instructions beyond branches, before knowing the actual outcome of the branch. This technique is broadly called "speculation" and we have already discussed extensively one form of it, branch prediction.

Speculation differs from dynamic scheduling. Speculation, issues, fetches and executes instructions (possibly without committing their results), as if branch prediction is always correct. Dynamic scheduling fetches and executes instructions that are ready to trigger, in a non-speculative manner. Dynamic scheduling implements essentially a data-flow execution model, instructions are triggered as soon as their operands are ready.

Speculation can complement and enhance processors with dynamic scheduling. More specifically, dynamic branch prediction can help fetching instructions from multiple basic blocks in the processor, by looking beyond one or several branches at a time. These instructions can then be executed with dynamic scheduling to exploit parallel execution units and OOO execution. Note that speculation needs an "undo" mechanism, in case the prediction is wrong.

How would we implement speculation in Tomasulo's algorithm? The key idea is to separate the execution stage of an instruction from the commit of results. We save the results of the instruction in some storage on chip, and we allow an instruction to commit results only when we make sure that the instruction is no longer speculative. To accomplish this, we are using a buffer to hold the results of instructions, called the **reorder buffer (ROB).** Besides holding the results of speculative instructions, the ROB also serves the purpose of passing data between instructions and resolving dependencies.

In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file. With speculation, the register file is not updated until the instruction commits**,** i.e. until we know definitively that the instruction should execute. Thus, the ROB supplies operands in the interval between completion of instruction execution and instruction commit. The ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in the original Tomasulo's algorithm. ROB also extends the set of registers in the architecture and instructions can use ROB entries to rename their registers.

The presence of a reorder buffer may give rise to RAW and WAW hazards between loads and stores (i.e. dependencies due to reordering of writes to memory locations). In principle, we need to resolve the addresses of target memory locations to check whether there are hazards or not in these situations.

WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending .RAW hazards through memory are maintained by two restrictions: not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and maintaining the program order for the computation of an effective address of a load with respect to all earlier stores. These restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data

Tomasulo's algorithm ignores precise interrupts. Recall that with precise interrupt handling, if an interrupt is triggered by an instruction we want the machine to operate as if it executed all instructions before the one that caused the interrupt and no instruction after the one that caused the interrupt. If the interrupt is processed and resolved, we need to restart the program at the interrupting instruction. Similarly, with an external interrupt, we need to restart the program at a specific interrupted instruction, with the machine in a consistent state.

Fortunately, interrupt processing resembles speculation. In both cases, we need to roll back the program to a consistent state of execution. The trick is to ensure that the instructions commit in order (i.e. they update processor state in order). We use a ROB and we do not let an instruction commit, unless the instruction is at the head of the ROB and the hardware is sure that the instruction is non-speculative and the instruction has not triggered an exception. If the instruction triggers an exception we record it in the ROB.

Can we get CPI < 1?

This means that we need to be able to issue more than one instructions per cycle and commit the results of more than one instructions per cycle (over the execution of the program). A processor issuing more than one instructions per cycle is called a multi-issue processor. These come in 3 flavors**:**

1.statically-scheduled superscalar processors,

2.dynamically-scheduled superscalar processors, and

3.VLIW (very long instruction word) processors

The 2 types of superscalar processors issue varying numbers of instructions per clock. If they are statically scheduled they issue the instructions in order and execute them in order. If they are dynamically scheduled, they issue the instructions in order, execute them out of order and commit them in order. VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium). This means that the compiler is responsible for finding parallelism in the instruction and creating these instruction packets to utilize

VLIW processors

In VLIW processors, each "long instruction" has explicit coding for multiple operations (called a "packet" or a "molecule" in various implementations). Obviously, long instruction words have room for many operations (primitive instructions, e.g. loads, stores, adds, etc.). There is a trade-off between instruction space and simplicity of instruction decoding, when deciding on the size of the very long word. By definition, all the operations the compiler puts in the long instruction word are independent, meaning **they are guaranteed that they can execute in parallel.** To accomplish this, we need really advanced compiler techniques. In particular, we need compiler techniques that can fetch and schedule instructions across several branches.

What are the potential problems with VLIW?

First, code size may increase, because the compiler can not always use all the instruction slots in the very long word. If slots are left empty, instruction space is wasted. We also need very aggressive and hard compiler optimizations to generate enough instructions in a straight-line code fragment. In particular, we need aggressive loop unrolling. Unused instruction slots also mean unused functional units, aka wasted resources. Instructions operate in a lock-step and there is no hazard detection in HW (the software bears all the responsibility here). A stall in any functional unit pipeline causes the entire processor to stall, since all the functional units must be kept synchronized. The compiler may predict easily the stalls in functional units, but not the stalls in caches, to schedule further down the program easily. VLIW creates also binary incompatibility issues: Different generations of a VLIW processor may need different versions of the binary code (contrary to most other processor families…).

What are other mechanisms to increase instruction fetch bandwidth (more instruction to fetch per cycle)?

•Integrated branch prediction: branch predictor is part of instruction fetch unit and is constantly predicting branches

•Instruction prefetching: Instruction fetch units prefetch to deliver multiple instructions per clock, integrating it with branch prediction

To achieve prefetching, we typically buffer instructions in an instruction queue. Prefetching typically requires fetching multiple cache blocks, and needs to be effective (i.e. accurate, timely, and non-intrusive) to effectively improve instruction fetch bandwidth.

An alternative to using a ROB is to use more physical registers, combined with renaming. Instruction issue now maps names of architectural registers to physical register numbers in extended register set . On issue, the processor allocates a new unused register for the destination (which avoids WAW and WAR hazards) Speculation recovery in this case is easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits. Most current Out-of-Order processors today use extended registers with renaming.

Lastly, we discuss some more aggressive speculation techniques in class, most notably value prediction and memory address dependence prediction.