

Superscalar Execution

Prof. Loh

CS3220 - Processor Design - Spring 2005

April 19, 2005

1 What is “Superscalar”?

“Scalar execution” is a computer organization where only a single instruction is processed at a time. Note that for pipelined processors, this really translates to an organization where each pipeline stage only processes a single instruction at a time. With vector execution, each pipeline stage still processes only a single instruction, but there is more than one piece of data associated with that instruction. You can think of scalar as SISD and vector as SIMD.

With *superscalar execution*, we still fundamentally have a scalar organization from the programmer’s point of view. Each instruction *appears* to execute independently and performs its operation on only a single piece of data (i.e. not vector). In the same way that pipelining and caching are invisible to the programmer, the parallel execution of multiple instructions in a superscalar processor are also hidden. One can view superscalar as a *very* limited form of under-the-covers MIMD.

The peak execution rate of a traditional pipelined process with *no* stalls is 1 CPI. With superscalar execution, the goal is now to achieve <1 CPI, or >1 IPC (instructions per cycle).

2 Case Study: Original Pentium

The original Pentium processor (not Pentium-II or -III or P4) used two parallel integer pipelines to support a peak execution rate of two instructions per cycle. The integer pipeline consist of five stages: Prefetch (PF), two Decode stages (D1 and D2), Execute (E) and WriteBack (WB).

Stage PF is “PreFetch” which is effectively the fetch stage. D1 and D2 are for decoding the instructions, reading operands, and performing address computation for memory instructions. E is the execute stage, and WB is writeback. Stages PF and D1 can process multiple instructions simultaneously. Stages D2 through WB are duplicated into the “u-pipe” and the “v-pipe”.

2.1 PF: PreFetch Stage

Prior to decoding instructions, the bytes from the instruction cache or main memory are simply a collection of bits. Without decoding, it is unknown how many instructions are in a group of bytes, where the instructions start and end, and what kinds of instructions are present. To fetch multiple instructions per cycle, the Pentium PF stage simply grabs large chunks of instruction bytes and then lets the decode stage sort it all out.

The PF stage uses four “prefetch buffers”, each which holds 32 bytes of instructions. On each cycle, the prefetch stage fetches sequential lines from the instruction cache and writes into one of the buffers. A *branch target buffer* (BTB) determines if a branch is present, and if so the BTB also predicts the next cache line to fetch. If the BTB predicts that the branch is not-taken, then the prefetch stage continues to fetch sequential instruction cache lines. A prediction of taken uses the predicted target to fetch from the new target location. This results in a single-cycle bubble, similar to the case with the conventional scalar pipeline.

2.2 D1: Decode

The D1 stage perform preliminary decoding of the instruction. In particular, it parses prefixes and opcodes to determine the lengths of up to two separate x86 instructions. The bytes for these instructions are then separated into two bundles or groups, with one passed to the D2 stage of the u-pipe, and the other to the D2 stage of the v-pipe.

The D1 stage is not always able to deliver two instructions per cycle. If the sizes of the individual instructions are too large, then the D1 stage may only be able to handle one instruction at a time. Certain prefix bytes and complex instructions may also result in a decrease in the effective decode rate.

2.3 D2: Decode

The second decode stage reads the instruction operands. This is similar to the register file read functionality performed by the decode stage of the traditional 5-stage RISC pipelines studied earlier. Furthermore, if the instruction requires an effective address calculation (loads and stores), the address is computed in this stage. The D2 stage uses CSA addition to quickly compute the address from multiple inputs (remember that the more complex addressing modes in x86 may involve up to three register operands).

2.4 E: Execute

The execute stage performs the actual instruction execution. In the case of loads, the dual execute stages can actually perform two loads in parallel. To support this, the data cache is divided into 8 separate banks. Every four bytes of the address space are assigned to a separate bank. So long as two load instructions access different banks, the two loads can execute concurrently.

To be able to execute any two instructions in both pipes, the execute stages would have to fully duplicate all functional units. For some units like adders and bit-wise logic, the relative hardware cost of duplication is not too great. In other units such as multipliers and dividers, the cost (in terms of area, transistor count, design complexity, and/or power) for duplication is just too high.

To keep the implementation cost under control, the two execution pipes are not identical. For example, the shifter functional unit only exists in the u-pipe, and therefore all instructions must execute in the u-pipe. It thus follows that the superscalar pipeline cannot execute two instructions per cycle if both instructions are shifts. Similarly, the v-pipe is the only one that can execute `jmp`, `jcc`, `call` and `fxch` instructions. For the most part, the u-pipe can execute any instruction (except those just listed) while the v-pipe is restricted to simpler instructions (`mov`, simple ALU ops, `lea` (load effective address), `test/cmp`, `nops`). At first this may seem like a limitation of the hardware, but the average instruction mix of most applications contains a large number of these simple operations and therefore it is in general not too difficult to keep both pipelines busy.

The resource limitations introduce pairing rules which are basically a set of constraints that only allow certain instructions to execute concurrently with other instructions. The resource constraints are perhaps obvious (no way to execute two divides when you only have one divider). There are other pairing rules which will be discussed below.

2.5 WB: Writeback

This is just your traditional register file writeback stage.

3 Pairing Rules

The asymmetric execution pipelines cause the processor to have to enforce pairing rules. That is only certain instructions may execute concurrently with certain other instructions. Besides the resource contention problem, the other major reason why two instruction cannot be paired is register dependencies.

The basic dependency is a true-dependence where one instruction produces a value that the second instruction uses as an input. Obviously the second instruction cannot execute until the first has completed, which prevents their parallel execution.

A write-dependence also prevents pairing. If two instructions attempt to write the same register at the same time, then which value gets stored? For correctness, it should be the later instruction, but it is not obvious what happens to the SRAM cells in the register file when two values are written simultaneously.

During the D2 decode stage, extra logic must check for all of these (and other) pairing rules. Parallel execution of two instructions may only occur if these two instructions pass all of the pairing rules. The logic to actually check and enforce these rules adds complexity, circuitry, and power to the decode stage that otherwise would not be there if we had a simple traditional 1-wide scalar pipeline.

In the case of partial registers, the processor treats all versions of a register as the same effective register. For example, the checker for pairing rules will view the following instructions

```
A: mov al, 1
B: mov ah, 0
```

as having the same output:

```
A': mov eax, 1
B': mov eax, 0
```

which creates a write-dependence, thus prevent pairing. In theory these two instructions could be executed in parallel, but the additional hardware required to take these two results and merge them into a single version of register `eax` is fairly complex (consider also that these results may then have to be bypassed to other instructions).

4 Hardware Changes

Supporting a pipeline *width* of two instructions per stage requires increasing the complexity of the logic in many places.

4.1 Decode

The decode logic must be able to decode two instructions in parallel. The x86 decode process is inherently serial due to the fact that you cannot decode the second instruction without having completed enough decoding on the first to determine where the first instruction ends. This makes the timing of the decode logic more difficult, and is part of the reason why the decode functionality is split over two stages.

4.2 Register File

Normally, the register file only needs to have two read ports and a single write port (maybe three read ports when you consider the SIB addressing mode). If the processor must read arguments and writeback the results for two instructions per cycle, then the number of ports must be doubled. This can be a very expensive proposition as the size of the register file tends to increase quadratically with the port count. This translates into both longer access latency as well as increased power consumption.

4.3 Bypass

With two execution pipelines, there are now twice as many places to bypass to, and twice as many places to bypass from. The number of bypass paths for a conventional n -stage scalar pipeline is already $O(n^2)$. For a s -wide superscalar pipeline, the number of bypass paths increases to $O(s \cdot n^2)$. Each bypass path consists of 32-64 wires (or 80+ for x87 floating point or 128 for SSE!) and so multiplying the number of such paths by s greatly increasing the number of wires, wire area, and wire congestion. The size of each of the bypass muxes also increases since they each now have approximately twice as many inputs. The scoreboard logic must also track register values moving through more pipeline locations, and it must coordinate the data-forwarding through all of these new bypass paths.

4.4 Interrupt/Fault Handling

While the processor may execute two instructions per cycle, it must maintain the outward appearance of a sequential processor. The programmer's view of the processor is to remain unchanged from that of a traditional von Neuman paradigm. To that extent, any "externally visible" activity must occur in sequential order. For example, if the processor attempts to execute two loads at the same time, and the first (in program order) misses while the second hits in the data cache, both loads must wait for the one miss. Similarly, if the first load misses and the second load suffers a page fault, the second load must sit around and wait until the first load's value has come back from main memory. Only after that can it flush the pipeline and handle transfer control to the page fault handler. The main idea is that for all outward appearances, the processor must *look* like it is behaving in the same fashion as a purely sequential machine.