A Tour of the Pentium(R) Pro Processor Microarchitecture

Introduction

Achieving twice the performance of a Pentium(R) processor while being manufactured on the same semiconductor process was one of the Pentium Pro processor's primary goals. Using the same process as a volume production processor practically assured that the Pentium Pro processor would be manufacturable, but it meant that Intel had to focus on an improved microarchitecture for ALL of the performance gains. This guided tour describes how multiple architectural techniques - some proven in mainframe computers, some proposed in academia and some we innovated ourselves - were carefully interwoven, modified, enhanced, tuned and implemented to produce the Pentium Pro microprocessor. This unique combination of architectural features, which Intel describes as Dynamic Execution, enabled the first Pentium Pro processor silicon to exceed the original performance goal.

Building from an already high platform

The Pentium processor set an impressive performance standard with its pipelined, superscalar microarchitecture. The Pentium processor's pipelined implementation uses five stages to extract high throughput from the silicon - the Pentium Pro processor moves to a decoupled, 12-stage, superpipelined implementation, trading less work per pipestage for more stages. The Pentium Pro processor reduced its pipestage time by 33 percent, compared with a Pentium processor, which means that from a semiconductor manufacturing process (i.e., transistor speed) perspective a 133MHz Pentium Pro processor and a 100MHz Pentium processor are equivalent.

The Pentium processor's superscalar microarchitecture, with its ability to execute two instructions per clock, would be difficult to exceed without a new approach. The new approach used by the Pentium Pro processor removes the constraint of linear instruction sequencing between the traditional "fetch" and "execute" phases, and opens up a wide instruction window using an instruction pool. This approach allows the "execute" phase of the Pentium Pro processor to have much more visibility into the program's instruction stream so that better scheduling may take place. It requires the instruction "fetch/decode" phase of the Pentium Pro processor to be much more intelligent in terms of predicting program flow. Optimized scheduling requires the fundamental "execute" phase to be replaced by decoupled "dispatch/execute" and "retire" phases. This allows instructions to be started in any order but always be completed in the original program order. The Pentium Pro processor is implemented as three independent engines coupled with an instruction pool as shown in Figure 1 below.

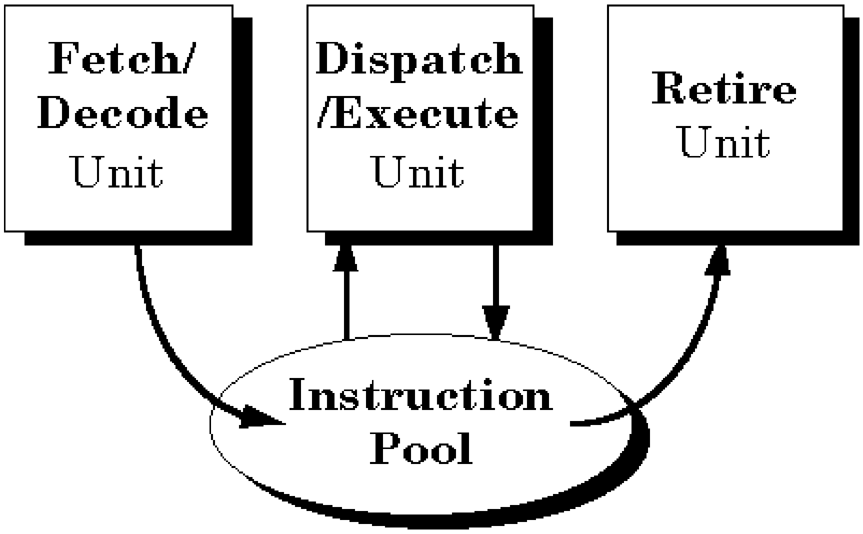

Figure 1. The P6 is implemented as three independent engines that communicate using an instruction pool.

What is the fundamental problem to solve?

Before starting our tour on how the Pentium Pro processor achieves its high performance it is important to note why this three- independent-engine approach was taken. A fundamental fact of today's microprocessor implementations must be appreciated: most CPU cores are not fully utilized. Consider the code fragment in Figure 2 below:

```
r1 <=  mem [r0]          /* Instruction 1 */
r2 <=  r1 + r2           /* Instruction 2 */
r5 <=  r5 + 1            /* Instruction 3 */
r6 <=  r6 - r3           /* Instruction 4 */
```

Figure 2. A typical code fragment.

The first instruction in this example is a load of r1 that, at run time, causes a cache miss. A traditional CPU core must wait for its bus interface unit to read this data from main memory and return it before moving on to instruction 2. This CPU stalls while waiting for this data and is thus being under-utilized.

While CPU speeds have increased 10-fold over the past 10 years, the speed of main memory devices has only increased by 60 percent. This increasing memory latency, relative to the CPU core speed, is a fundamental problem that the Pentium Pro processor set out to solve. One approach would be to place the burdon of this problem onto the chipset but a high-performance CPU that needs very high speed, specialized, support components is not a good solution for a volume production system.

A brute-force approach to this problem is, of course, increasing the size of the L2 cache to reduce the miss ratio. While effective, this is another expensive solution, especially considering the speed requirements of today's L2 cache SRAM components. Instead, the Pentium Pro processor is designed from an overall system implementation perspective which will allow higher performance systems to be designed with cheaper memory subsystem designs.

Pentium Pro processor takes an innovative approach

To avoid this memory latency problem the Pentium Pro processor "looks-ahead" into its instruction pool at subsequent instructions and will do useful work rather than be stalled. In the example in Figure 2, instruction 2 is not executable since it depends upon the result of instruction 1; however both instructions 3 and 4 are executable. The Pentium Pro processor speculatively executes instructions 3 and 4. We cannot commit the results of this speculative execution to permanent machine state (i.e., the programmer-visible registers) since we must maintain the original program order, so the results are instead stored back in the instruction pool awaiting in-order retirement. The core executes instructions depending upon their readiness to execute and not on their original program order (it is a true dataflow engine). This approach has the side effect that instructions are typically executed out-of-order.

The cache miss on instruction 1 will take many internal clocks, so the Pentium Pro processor core continues to look ahead for other instructions that could be speculatively executed and is typically looking 20 to 30 instructions in front of the program counter. Within this 20- to 30- instruction window there will be, on average, five branches that the fetch/decode unit must correctly predict if the dispatch/execute unit is to do useful work. The sparse register set of an Intel Architecture (IA) processor will create many false dependencies on registers so the dispatch/execute unit will rename the IA registers to enable additional forward progress. The retire unit owns the physical IA register set and results are only committed to permanent machine state when it removes completed instructions from the pool in original program order.

Dynamic Execution technology can be summarized as optimally adjusting instruction execution by predicting program flow, analysing the program's dataflow graph to choose the best order to execute the instructions, then having the ability to speculatively execute instructions in the preferred order. The Pentium Pro processor dynamically adjusts its work, as defined by the incoming instruction stream, to minimize overall execution time.

Overview of the stops on the tour

We have previewed how the Pentium Pro processor takes an innovative approach to overcome a key system constraint. Now let's take a closer look inside the Pentium Pro processor to understand how it implements Dynamic Execution. Figure 3 below extends the basic block diagram to include the cache and memory interfaces - these will also be stops on our tour. We shall travel down the Pentium Pro processor pipeline to understand the role of each unit:

- The FETCH/DECODE unit: An in-order unit that takes as input the user program instruction stream from the instruction cache, and decodes them into a series of micro-operations (uops) that represent the dataflow of that instruction stream. The program pre-fetch is itself speculative.

- The DISPATCH/EXECUTE unit: An out-of-order unit that accepts the dataflow stream, schedules execution of the uops subject to data dependencies and resource availability and temporarily stores the results of these speculative executions.

- The RETIRE unit: An in-order unit that knows how and when to commit ("retire") the temporary, speculative results to permanent architectural state.
- The BUS INTERFACE unit: A partially ordered unit responsible for connecting the three internal units to the real world. The bus interface unit communicates directly with the L2 cache supporting up to four concurrent cache accesses. The bus interface unit also controls a transaction bus, with MESI snooping protocol, to system memory.

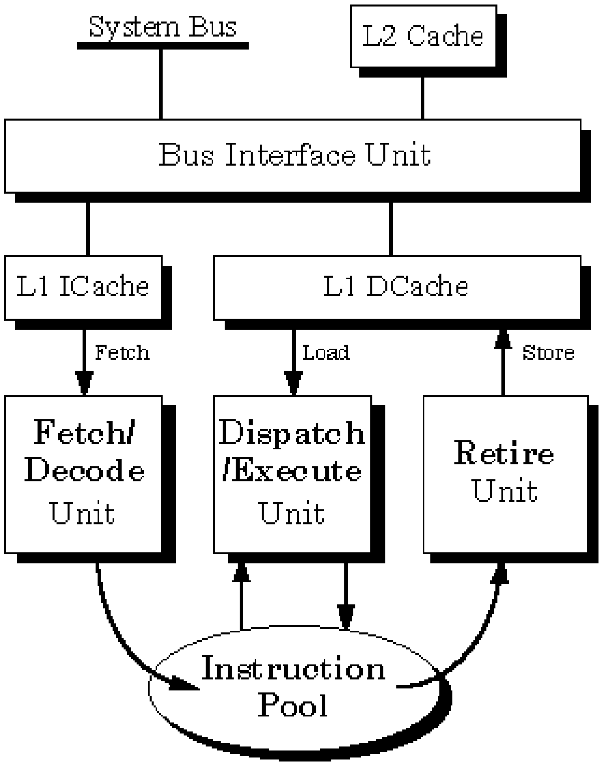

Figure 3. The three core engines interface with the memory subsystem using 8K/8K unified caches.

Tour stop #1: The FETCH/DECODE unit.

Figure 4 shows a more detailed view of the fetch/decode unit:

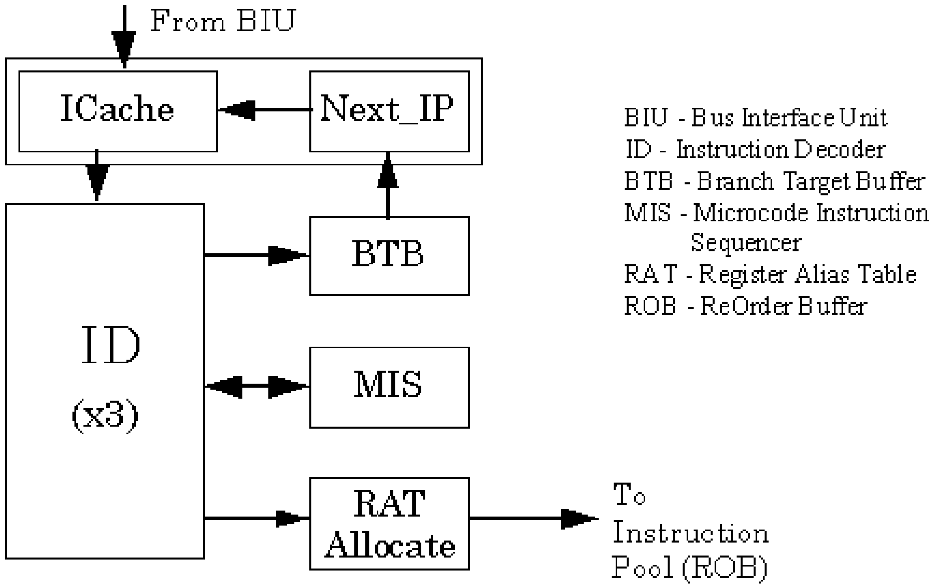

Figure 4: Looking inside the Fetch/Decode Unit

Let's start the tour at the Instruction Cache (ICache), a nearby place for instructions to reside so that they can be looked up quickly when the CPU needs them. The Next_IP unit provides the ICache index, based on inputs from the Branch Target Buffer (BTB), trap/interrupt status, and branch-misprediction indications from the integer execution section. The 512 entry BTB uses an extension of Yeh's algorithm to provide greater than 90 percent prediction accuracy. For now, let's assume that nothing exceptional is happening, and that the BTB is correct in its predictions. (The Pentium Pro processor integrates features that allow for the rapid recovery from a mis-prediction, but more of that later.)

The ICache fetches the cache line corresponding to the index from the Next_IP, and the next line, and presents 16 aligned bytes to the decoder. Two lines are read because the IA instruction stream is byte-aligned, and code often branches to the middle or end of a cache line. This part of the pipeline takes three clocks, including the time to rotate the prefetched bytes so that they are justified for the instruction decoders (ID). The beginning and end of the IA instructions are marked.

Three parallel decoders accept this stream of marked bytes, and proceed to find and decode the IA instructions contained therein. The decoder converts the IA instructions into triadic uops (two logical sources, one logical destination per uop). Most IA instructions are converted directly into single uops, some instructions are decoded into one-to-four uops and the complex instructions require microcode (the box labeled MIS in Figure 4, this microcode is just a set of preprogrammed sequences of normal uops). Some instructions, called prefix bytes, modify the following instruction giving the decoder a lot of work to do. The uops are enqueued, and sent to the Register Alias Table (RAT) unit, where the logical IA-based register references are converted into Pentium Pro processor physical register references, and to the Allocator stage, which adds status information to the uops and enters them into the instruction pool. The instruction pool is implemented as an array of Content Addressable Memory called the ReOrder Buffer (ROB).

We have now reached the end of the in-order pipe.

Tour stop #2: The DISPATCH/EXECUTE unit

The dispatch unit selects uops from the instruction pool depending upon their status. If the status indicates that a uop has all of its operands then the dispatch unit checks to see if the execution resource needed by that uop is also available. If both are true, it removes that uop and sends it to the resource where it is executed. The results of the uop are later returned to the pool. There are five ports on the Reservation Station and the multiple resources are accessed as shown in Figure 5 below:

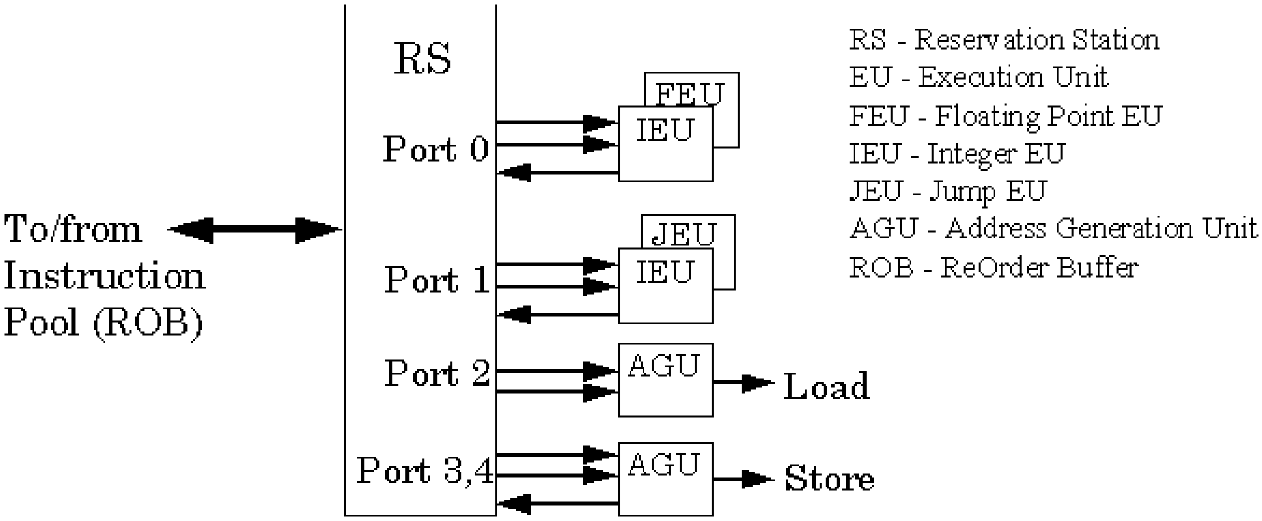

Figure 5: Looking inside the Dispatch/Execute Unit

The Pentium Pro processor can schedule at a peak rate of 5 uops per clock, one to each resource port, but a sustained rate of 3 uops per clock is typical. The activity of this scheduling process is the quintessential out-of-order process; uops are dispatched to the execution resources strictly according to dataflow constraints and resource availability, without regard to the original ordering of the program.

Note that the actual algorithm employed by this execution-scheduling process is vitally important to performance. If only one uop per resource becomes data-ready per clock cycle, then there is no choice. But if several are available, which should it choose? It could choose randomly, or first-come-first-served. Ideally it would choose whichever uop would shorten the overall dataflow graph of the program being run. Since there is no way to really know that at run-time, it approximates by using a pseudo FIFO scheduling algorithm favoring back-to-back uops.

Note that many of the uops are branches, because many IA instructions are branches. The Branch Target Buffer will correctly predict most of these branches but it can’t correctly predict them all. Consider a BTB that’s correctly predicting the backward branch at the bottom of a loop: eventually that loop is going to terminate, and when it does, that branch will be mispredicted. Branch uops are tagged (in the in-order pipeline) with their fallthrough address and the destination that was predicted for them. When the branch executes, what the branch actually did is compared against what the prediction hardware said it would do. If those coincide, then the branch eventually retires, and most of the speculatively executed work behind it in the instruction pool is good.

But if they do not coincide (a branch was predicted as taken but fell through, or was predicted as not taken and it actually did take the branch) then the Jump Execution Unit (JEU) changes the status of all of the uops behind the branch to remove them from the instruction pool. In that case the proper branch destination is provided to the BTB which restarts the whole pipeline from the new target address.

Tour stop #3: The RETIRE unit

Figure 6 shows a more detailed view of the retire unit:

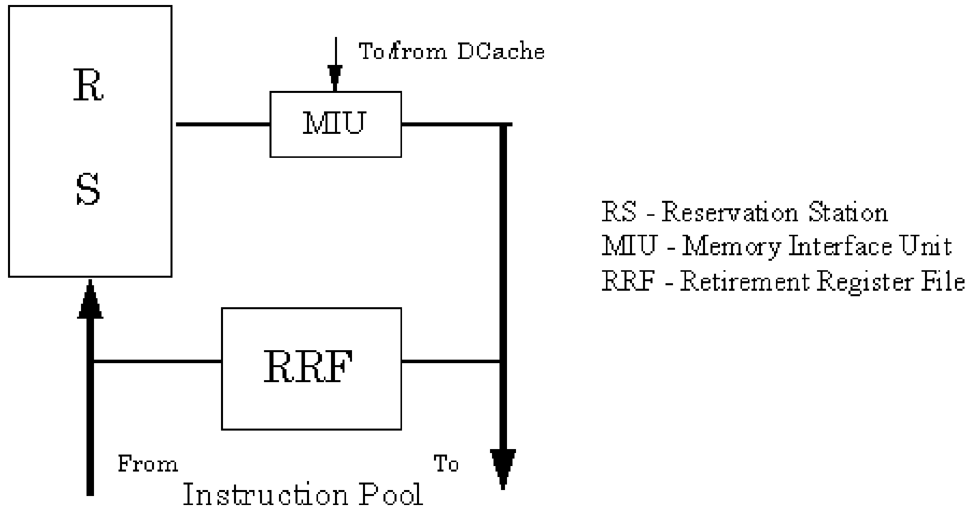

Figure 6: Looking inside the Retire Unit

The retire unit is also checking the status of uops in the instruction pool - it is looking for uops that have executed and can be removed from the pool. Once removed, the uops' original architectural target is written as per the original IA instruction. The retirement unit must not only notice which uops are complete, it must also re-impose the original program order on them. It must also do this in the face of interrupts, traps, faults, breakpoints and mis- predictions.

There are two clock cycles devoted to the retirement process. The retirement unit must first read the instruction pool to find the potential candidates for retirement and determine which of these candidates are next in the original program order. Then it writes the results of this cycle's retirements to both the Instruction Pool and the RRF. The retirement unit is capable of retiring 3 uops per clock.

Tour stop #4: BUS INTERFACE unit

Figure 7 shows a more detailed view of the bus interface unit:

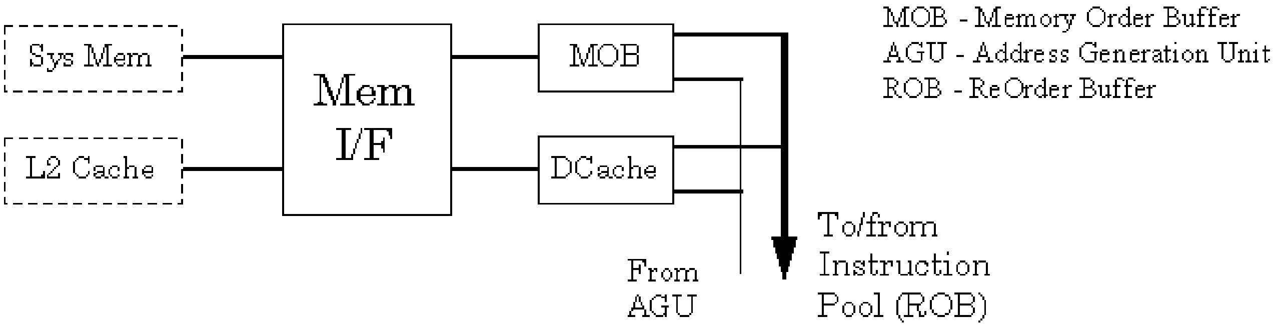

Figure 7: Looking inside the Bus Interface Unit

There are two types of memory access: loads and stores. Loads only need to specify the memory address to be accessed, the width of the data being retrieved, and the destination register. Loads are encoded into a single uop. Stores need to provide a memory address, a data width, and the data to be written. Stores therefore require two uops, one to generate the address, one to generate the data. These uops are scheduled independently to maximize their concurrency, but must re-combine in the store buffer for the store to complete.

Stores are never performed speculatively, there being no transparent way to undo them. Stores are also never re- ordered among themselves. The Store Buffer dispatches a store only when the store has both its address and its data, and there are no older stores awaiting dispatch.

What impact will a speculative core have on the real world? Early in the Pentium Pro processor project, we studied the importance of memory access reordering. The basic conclusions were as follows:

- Stores must be constrained from passing other stores, for only a small impact on performance.
- Stores can be constrained from passing loads, for an inconsequential performance loss.
- Constraining loads from passing other loads or from passing stores creates a significant impact on performance.

So what we need is a memory subsystem architecture that allows loads to pass stores. And we need to make it possible for loads to pass loads. The Memory Order Buffer (MOB) accomplishes this task by acting like a reservation station and Re-Order Buffer, in that it holds suspended loads and stores, redispatching them when the blocking condition (dependency or resource) disappears.

Tour Summary

It is the unique combination of improved branch prediction (to offer the core many instructions), data flow analysis (choosing the best instructions to operate upon), and speculative execution (executing instructions in the preferred order) that enables the Pentium Pro processor to deliver twice the performance of a Pentium processor on the same semiconductor manufacturing process. This unique combination is called Dynamic Execution and it is similar in impact as "Superscalar" was to previous generation Intel Architecture processors.

And while our architects have been honing the Pentium Pro processor microarchitecture, our silicon technologists have been working on the next Intel process - this 0.35 micron process will enable future Pentium Pro Processor CPU core speeds in excess of 200MHz.