Pipelining: An Overview (Part I)

by Jon Stokes

Version 1.0 – September 20, 2004

Pipelining: An Overview (Part I)

by Jon Stokes

Understanding pipelining performance

The original Pentium 4 was a radical design for a number of reasons, but perhaps its most striking and controversial feature was its extraordinarily deep pipeline. At over 20 stages, the Pentium 4's pipeline almost twice as deep as the pipelines of the P4's competitors. Recently Prescott, the 90nm successor to the Pentium 4, took pipelining to the next level by adding another 10 stages onto the Pentium 4's already unbelievably long pipeline.

Intel's strategy of deepening the Pentium 4's pipeline, a practice that Intel calls "hyperpipelining", has paid off in terms of performance, but it is not without its drawbacks. In previous articles on the Pentium 4 and Prescott, I've referred to the drawbacks associated with deep pipelines, and I've even tried to explain these drawbacks within the context of larger technical articles on Netburst and other topics. In the present series of articles, I want to devote some serious time to explaining pipelining, its effect on microprocessor performance, and its potential downsides. I'll take you through a basic introduction to the concept of pipelining, and then I'll explain what's required to make pipelining successful and what pitfalls face deeply pipelined designs like Prescott. By the end of the article, you should have a clear grasp on exactly how pipeline depth is related to microprocessor performance on different types of code.

Note that if you read an earlier article of mine from a few years back entitled "Understanding Pipelining and Performance," you'll find the first part of this text—specifically the assembly line analogy—vaguely familiar. The present article is based in part on that earlier article, but it has been reworked from the ground up to be clearer, more precise, and more up-to-date.

The lifecycle of an instruction

The basic action of any microprocessor as it moves through the instruction stream can be broken down into a series of four simple steps, which each instruction in the code stream goes through in order to be executed:

1. **Fetch** the next instruction from the address stored in the program counter.
2. Store that instruction in the instruction register and **decode** it, and increment the address in the program counter.
3. **Execute** the instruction currently in the instruction register. If the instruction is not a branch instruction but an arithmetic instruction, send it to the proper ALU.

    a. *Read* the contents of registers the input registers.

b. *Add* the contents of the input registers.

4. **Write** the results of that instruction from the ALU back into the destination register.

In a modern processor, the four steps above get repeated over and over again until the program is finished executing. These are, in fact, the four stages in a classic RISC pipeline. (I'll define the term "pipeline" shortly; for now, just think of a pipeline as a series of stages that each instruction in the code stream must pass through when the code stream is being executed.) Here are the four stages in their abbreviated form, the form in which you'll most often see them:

1. Fetch

2. Decode

3. Execute

4. Write (or "write-back")

Each of the above stages could be said to represent one phase in the "lifecycle" of an instruction. An instruction starts out in the fetch phase, moves to the decode phase, then to the execute phase, and finally to the write phase. Each phase takes a fixed, but by no means equal, amount of time. In most of the example processors with which we'll be working in this article, all four phases take an equal amount of time; this is not usually the case in real-world processors. In any case, if a simple example processor takes exactly 1 nanosecond to complete each stage, then the that processor can finish one instruction every 4 nanoseconds.

Pipelining basics: an analogy

The present section uses a factory analogy to explain pipelining. Other folks use simpler analogies, like doing laundry, for instance, to explain this technique, but there are a few reasons why I chose a more elaborate and lengthy analogy to illustrate what is at root a relatively simple concept. First, assembly line-based factories are easy for readers to visualize and there's plenty of room for filling out the mental image in interesting ways in order to make a variety of related points. Second, and perhaps even more importantly, the many scheduling-, queuing- and resource management-related problems that factory designers face have direct analogs in computer architecture. In many cases, the problems and solutions are exactly the same, simply translated into a different domain. (Note that similar queuing-related problem/solution pairs also crop up in the service industry, which is why analogies involving supermarkets and fast food restaurants are also favorites of mine.)

Let's say that my friends and I have decided to go into the automotive manufacturing business, and that our first product is to be a sport utility vehicle (SUV). After some research, we determine that there are five stages in the SUV building process:

Stage 1: build the chassis.
Stage 2: drop the engine in the chassis.

Stage 3: put doors, a hood, and coverings on the chassis.
Stage 4: attach the wheels.
Stage 5: paint the SUV.

Each of the stages above requires the use of highly trained workers with very specialized skill sets, with the result that workers who are good at building chasses don't know much about engines, bodywork, wheels, or painting, and likewise for engine builders, painters, and the other crews. So when we make our first attempt to put together an SUV factory, we hire and train five crews of specialists, one for each stage of the SUV building process. There's one crew to build the chassis, one to drop in the engine, another for the wheels, and a painting crew. Finally, because the crews are so specialized and efficient, each stage of the SUV building process takes a crew exactly one hour to complete.

Now, since my friends and I are computer types and not industrial engineers, we had a lot to learn about making efficient use of factory resources. We based the functioning of our first factory on the following plan: place all five crews in a line on the factory floor, and have the first crew start an SUV at Stage 1. After Stage 1 is complete, the Stage 1 crew passes partially finished SUV off to the Stage 2 crew and then hits the breakroom to play some foosball, while the Stage 2 crew builds the engine and drops it in. Once the Stage 2 crew is done, the SUV moves down to Stage 3 and the Stage 3 crew takes over while the Stage 2 crew joins the Stage 1 crew in the breakroom.

The SUV moves on down the line through all five stages in this way, with only one crew working on one stage at any given time while the rest of the crews sit idle. Once the completed SUV finishes Stage 5, the crew at Stage 1 then starts on another SUV. At this rate, it takes exactly five hours to finish a single SUV, and our factory completes one SUV every five hours.

Fast-forward one year. Our SUV, the Extinction LE, is selling like... well, it's selling like an SUV, which means it's doing pretty well. In fact, our SUV is selling so well that we've attracted the attention of the military and been offered a contract to provide SUVs to the U.S. Army on an ongoing basis. The Army likes to order multiple SUVs at a time; one order might come in for 10 SUVs, and another order might come in for 500 SUVs. The more of these orders that we can fill each fiscal year, the more money we can make during that same period and the better our balance sheet looks. This, of course, means that we'll want to find a way to increase the number of SUVs that our factory can complete per hour (i.e. our factory's SUV completion rate). By completing more SUVs per hour, we can fill the Army's orders faster and make more money each year.

The most intuitive way to go about increasing our factory's SUV completion rate would be to try and decrease the production time of each SUV. If we could get the crews to work twice as fast, then our factory could produce twice as many SUVs in the same amount of time. Our crews are already working as hard as they can, though, so unless there's a technological breakthrough that increases their productivity this option is off the table for now.

Since we can't speed up our crews, we could always use the brute force approach and just throw money at the problem by building a second assembly line. If we were to hire and train five new crews to form a second assembly line, also capable of producing one

car every five hours, we could complete a grand total of two SUVs every five hours from the factory floor—double the SUV completion rate of our present factory. This doesn't seem like a very efficient use of factory resources, though, since not only would we have twice as many crews working at once but we'd also have twice as many crews in the break room at once. There has to be a better way.

Faced with a lack of options, we hired a team of consultants to figure out a clever way to increase overall factory productivity without either doubling the number of crews or increasing each individual crew's productivity. One year and thousands of billable hours later, the consultants hit upon a solution. Why let our crews spend four fifths of their work day in the break room, when they could be doing useful work during that time? With proper scheduling of the existing five crews, our factory could complete *one SUV each hour*, and thus drastically improve the both the efficiency and the output of our assembly line. The revised workflow would look as follows: The Stage 1 crew builds a chassis. Once the chassis is complete, they send it on to the Stage 2 crew. The Stage 2 crew receives the chassis and begins dropping the engine in, while the Stage 1 crew starts on a new chassis. When both Stage 1 and Stage 2 crews are finished, the Stage 2 crew's work advances to Stage 3, the Stage 1 crew's work advances to Stage 2, and the Stage 1 crew starts on a new chassis.

So as the assembly line begins to fill up with SUVs in various stages of production, more of the crews are put to work simultaneously until all of the crews are working on a different vehicle in a different stage of production. (Of course, this is how most of us nowadays in the post-Ford era expect a good, efficient assembly line to work.) If we can keep the assembly line full, and keep all five crews working at once, then we can produce one SUV every hour: a five-fold improvement in SUV completion rate over the previous completion rate of one SUV every five hours. That, in a nutshell, is pipelining.

While the total amount of time that each individual SUV spends in production has not changed from the original 5 hours, the rate at which the factory as a whole completes SUVs, and hence the rate at which the factory can fulfill the Army's orders for batches of SUVs, has increased drastically. Pipelining works its magic by making optimal use of already existing resources. We don't need to speed up each individual stage of the production process, nor do we need to drastically increase the amount of resources that we throw at the problem; all that's necessary is that we get more work out of resources that are already there.

Bringing our discussion back to microprocessors, it should be easy to see how this concept applies to the four phases of an instruction's life-cycle. Just like the owners of the factory in our analogy wanted to increase the number of SUVs that the factory could finish in a given period of time, microprocessor designers are always looking for ways to increase the number of instructions that a CPU can complete in a given period of time. When we recall that a program is an ordered sequence of instructions, then it becomes clear that increasing the number of instructions executed per unit time is one way to decrease the total amount of time that it takes to execute a program. In terms of our analogy, a program is like an order of SUVs from the military; just like increasing our factory's output of SUVs per hour enabled us to fill orders faster, increasing our processor's instruction completion rate (i.e. the number of instructions completed per unit time) enables us to run programs faster.

A non-pipelined example

Early, non-pipelined processors worked on one instruction at a time, moving each instruction through all four phases of its lifecycle during the course of one clock cycle. Thus non-pipelined processors are also called single-cycle processors, because all instructions take exactly one clock cycle to execute fully (or, to pass through all four phases of their lifecycles).

Because the processor completes instructions at a rate of one per clock cycle, we want the CPU's clock to run as fast as possible so that the processor's instruction completion rate (i.e., the number of instructions completed per nanosecond) can be as high as possible. Thus we need to calculate the minimum amount of time that it takes to complete an instruction, and make the clock cycle time equivalent to that length of time. It just so happens that on our hypothetical example CPU, the four phases of the instruction's lifecycle take a total of four nanoseconds to complete. Therefore we should set the duration of the CPU clock cycle to four nanoseconds, so that the CPU can complete the instruction's lifecycle, from fetch to write-back, in a single clock. (A CPU clock cycle is often just called a "clock", for short.)

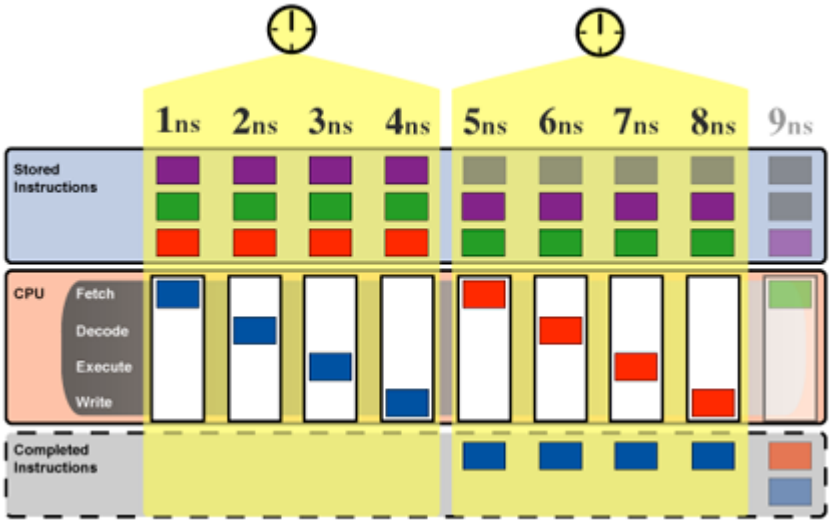

Figure PIPELINING.4: A single-cycle processor

In the diagram above, the blue instruction leaves the code storage area, enters the processor, and then advances through the phases of its lifecycle over the course of the four-nanosecond clock period, until at the end of the fourth nanosecond it completes the last phase and its lifecycle is over. The end of the fourth nanosecond is also the end of the first clock cycle, so now that the first clock cycle is finished and the blue instruction has completed its execution, the red can instruction enter the processor at the start of a new clock cycle and go through the same process. This four-nanosecond sequence of steps is repeated until, after a total of 16ns (or four clock cycles), the processor has completed all four instructions at a completion rate of 0.25 instructions/ns (= 4 instructions/16 ns).

Single-cycle processors like the one in figure PIPELINING.4 are simple to design, but they waste a lot of hardware resources. All of that white space in the diagram represents processor hardware that's sitting idle while it waits for the instruction that's currently in the processor to finish executing. By pipelining the processor above, we can put more of that hardware to work every nanosecond, thereby increasing the processor's efficiency and its performance on executing programs.

Before moving on, I should clarify a few aspects of the above diagram that some may find confusing. At the bottom of the diagram is a region labeled "Completed Instructions". Now, completed instructions don't actually go anywhere when they're finished executing; once they've done their job of telling the processor how to modify the data stream, they're simply deleted from the processor. (Note that instructions that have been deleted from the processor are still extant in the code storage area and are available for repeated use.) So the "Completed Instructions" box at the bottom of Figure PIPELINING.4 does not represent a real part of the computer, which is why I've placed a dotted line around it. This area is just a place for us to keep track of how many instructions the processor has completed in a certain amount of time, or the processor's instruction completion rate (or completion rate, for short), so that when we compare different types of processors we'll have a place where we can quickly look and get an instant feel for which processor performs better. The more instructions a processor completes in a set amount of time, the better it performs on programs, which are an ordered sequence of instructions. So think of the "Completed Instructions" box as a sort of scoreboard for tracking each processor's completion rate, and check the box in each of the following diagrams to see how long it takes for the processor to populate this box.

Following on the point above, you may be curious as to why the blue instruction that has completed in the fourth nanosecond does not appear in the "Completed Instructions" box until the fifth nanosecond. The reason is straightforward, and stems from the nature of the diagram. Because an instruction spends *one complete nanosecond*, from start to finish, in each stage of execution, the blue instruction enters the write phase at the *beginning* of the fourth nanosecond and exits the write phase at the *end* of the fourth nanosecond. This means that the fifth nanosecond is the first full nanosecond in which the blue instruction stands completed. Thus at the beginning of the fifth nanosecond (which coincides with the end of the fourth nanosecond) the processor has now completed one instruction.

A pipelined example

Pipelining a processor means breaking down its instruction execution process—what I've been calling the instruction's "lifecycle"— into a series of discrete pipeline stages which can be completed in sequence by specialized hardware. Recall the way that we broke down the SUV assembly process into five discreet steps, with one dedicated crew assigned to complete each step, and you'll get the idea.

Because an instruction's lifecycle consists of four fairly distinct phases, we can start by breaking down the single-cycle processor's instruction execution process into a sequence of four discreet pipeline stages, where each pipeline stage corresponds to a phase in the standard instruction lifecycle:

Stage 1: Fetch the instruction from code storage.
Stage 2: Decode the instruction.
Stage 3: Execute the instruction.
Stage 4: Write the results of the instruction back to the register file.

Note that the number of pipeline stages is referred to as the pipeline depth. So our four-stage pipeline has a pipeline depth of four.

For convenience's sake, let's say that each of the four pipeline stages above takes exactly one nanosecond to finish its work on an instruction, just like each crew in our assembly line analogy took 1 hour to finish its portion of the work on an SUV. So our original single-cycle processor's four-nanosecond execution process is now broken down into four discreet, sequential pipeline stages of one nanosecond each in length. Now let's step through another diagram together to see how a pipelined CPU would execute the four instructions depicted in figure PIPELINING.4.

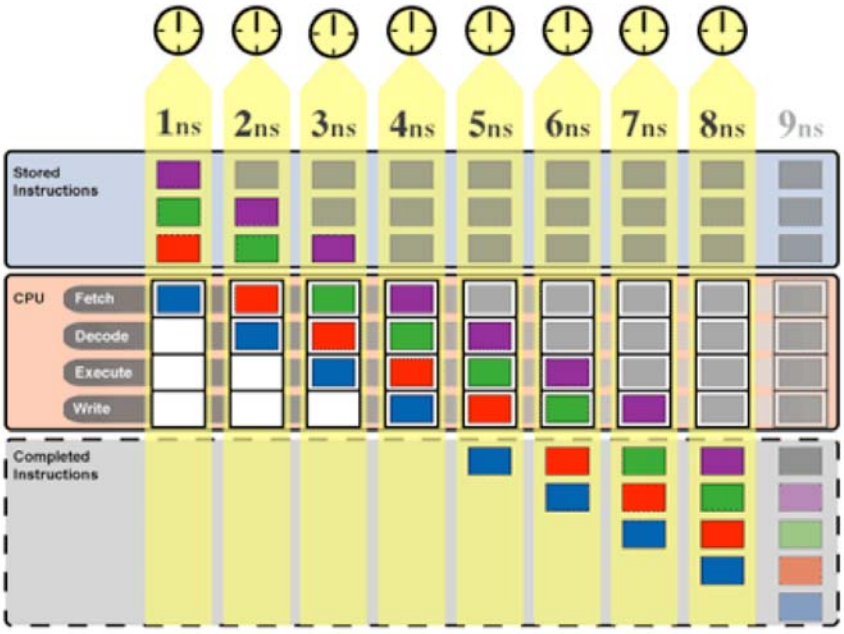

Figure PIPELINING.5: A four-stage pipeline

At the beginning of the first nanosecond, the blue instruction enters the fetch stage. After that nanosecond is complete, the second nanosecond begins and the blue instruction moves on to the decode stage while the next instruction, the red one, starts to make its way from code storage to the processor (i.e. it enters the fetch stage). At the start of the third nanosecond, the blue instruction advances to the execute stage, the red instruction advances to the decode stage, and the green instruction enters the fetch stage. At the fourth nanosecond, the blue instruction advances to the write stage, the red to the execute stage, the green to the decode stage, and the blue to the fetch stage. After the fourth nanosecond has fully elapsed and the fifth nanosecond starts, the blue instruction has passed from the pipeline and is now finished executing. Thus we can say that at the end of four nanoseconds (= four clock cycles) the pipelined processor depicted below has completed one instruction.

At start of the fifth nanosecond, the pipeline is now full and the processor can begin completing instructions at a rate of one instruction per nanosecond. This 1 instruction/ns completion rate is a four-fold improvement over the single-cycle processor's completion rate of 0.25 instructions/ns (or 4 instruction every 16 nanoseconds).

Shrinking the clock

You can see from the diagram above that the role of the CPU clock changes slightly in a pipelined processor, versus the single-cycle processor in figure PIPELINING.4. Because all of the pipeline stages must now work together simultaneously and be ready at the start of each new nanosecond to hand over the results of their work to the next pipeline stage, the clock is needed to coordinate the activity of the whole pipeline. The way that this is done is simple: shrink the clock cycle time to match the time that it takes each stage to complete its work, so that at the start of each clock cycle each pipeline stage hands off the instruction that it was working on to the next stage in the pipeline. Because each pipeline stage in our example processor takes one nanosecond to complete its work, we can set the clock cycle to be one nanosecond in duration.

This new method of clocking the processor means that a new instruction will not necessarily be completed at the close of each clock cycle, as was the case with the single-cycle processor. Instead, a new instruction will be completed at the close of only those clock cycles in which the write stage has been working on an instruction. Any clock cycle with an empty write stage will add no new instructions to the "Completed Instructions" box, and any clock cycle with an active write stage will add one new instruction to the box. Of course, this means that when the pipeline first starts to work on a program there will be a few clock cycles—three to be exact—during which no instructions are completed. But once the fourth clock cycle starts, the first instruction enters the write stage and the pipeline can then begin completing new instructions on each clock cycle, which, because each clock cycle is one nanosecond long, translates into a completion rate of one instruction per nanosecond.

Shrinking program execution time

Note that the total execution time for each individual instruction is not changed by pipelining. It still takes an instruction 4ns to make it all the way through the processor; that 4ns can be split up into 4 clock cycles of 1ns each, or it can cover one longer clock cycle, but it's still the same 4ns. Thus pipelining doesn't speed up instruction execution time, but it does speed up *program execution time* (i.e. the number of nanoseconds that it takes to execute an entire program) by increasing the number of instructions finished per unit time. Just like pipelining our hypothetical SUV assembly line allowed us to fill the Army's orders in a shorter span of time, even though each individual SUV still spent a total of five hours in the assembly line, so does pipelining allow a processor to execute programs in a shorter amount of time even though each individual instruction still spends the same amount of time traveling through the CPU. Pipelining makes more efficient use of the CPU's existing resources by putting all of its units to work simultaneously, thereby allowing it to do more total work each nanosecond.

Next week: The speed-up from pipelining

Date	Version	Changes
9/20/2004	1.0	Release