Pipelining: An Overview (Part II)

by Jon Stokes

Version 1.0 – September 27, 2004

Pipelining: An Overview (Part I)

by Jon Stokes

he speedup from pipelining

In general, the speedup in completion rate versus a single-cycle implementation that's gained from pipelining is ideally equal to the number of pipeline stages. A four-stage pipeline yields a four-fold speedup in the completion rate versus single-cycle, a five-stage pipeline yields a five-fold speedup, a twelve-stage pipeline yields a twelve-fold speedup, and so on. This speedup is possible because the more pipeline stages there are in a processor, the more instructions the processor can work on simultaneously and the more instructions it can complete in a given period of time. So the more finely you can slice those four phases of the instruction's lifecycle, the more of the hardware that's used to implement those phases you can put to work at any given moment.

To return to our assembly line analogy, let's say that each crew is made up of six workers, and that each of the hour-long tasks that each crew performs can be readily subdivided into two shorter, 30-minute tasks. So we can double our factory's throughput by splitting each crew into two smaller, more specialized crews of three workers each, and then having each smaller crew perform one of the shorter tasks on one SUV per 30 minutes.

- Stage 1: build the chassis.
  - Crew 1a: Fit the parts of the chassis together and spot-weld the joins.
  - Crew 1b: Fully weld all the parts of the chassis.
- Stage 2: drop the engine in the chassis.
  - Crew 2a: Place the engine in the chassis and mount it in place.
  - Crew 2b: Connect the engine to the moving parts of the car.
- Stage 3: put doors, a hood, and coverings on the chassis.
  - Crew 3a: Put the doors and hood on the chassis.
  - Crew 3b: Put the other coverings on the chassis.
- Stage 4: attach the wheels.
  - Crew 4a: Attach the two front wheels.
  - Crew 4b: Attach the two rear wheels.
- Stage 5: paint the SUV.
  - Crew 5a: Paint the sides of the SUV.
  - Crew 5b: Paint the top of the SUV.

After the modifications described above, the ten smaller crews in our factory would now have a collective total of ten SUVs in progress during the course of any given 30 minute period. Furthermore, our factory could now complete a new SUV every 30 minutes, a ten-fold improvement over our first factory's completion rate of one SUV every five

hours. So by pipelining our assembly line even more deeply, we've put even more of its workers to work simultaneously, thereby increasing the number of SUVs that can be worked on simultaneously and increasing the number of SUVs that can be completed within a given period of time.

Deepening the pipeline of our four-stage processor works on similar principles and has a similar effect on completion rates. Just like the five stages in our SUV assembly line could be broken down further into a longer sequence of more specialized stages, we can take the execution process that each instruction goes through and break it down into a series of much more than just four discreet stages. By breaking the processor's four-stage pipeline down into a longer series of shorter, more specialized stages, we can put even more of the processor's specialized hardware to work simultaneously on more instructions and thereby increase the number of instructions that the pipeline completes each nanosecond.

We first moved from a single-cycle processor to a pipelined processor by taking the four-nanosecond time period that the instruction spent traveling through the processor and slicing it into four discreet pipeline stages of one nanosecond each in length. These four discrete pipeline stages corresponded to the four phases of an instruction's lifecycle. A processor's pipeline stages aren't always going to correspond exactly to the four phases of a processor's lifecycle, though. Some processors have a five-stage pipeline, some have a six-stage pipeline, and many have pipelines deeper than ten or twenty stages. In such cases, the CPU designer must slice up the instruction's lifecycle into the desired number of stages in such a way that all the stages are equal in length.

Now let's take that four nanosecond execution process and slice it into eight discreet stages. Because all eight pipeline stages must be of exactly the same duration for pipelining to work, the eight pipeline stages must each be 4 ns ⁄ 8 = 0.5ns in length. Since we're presently working with an idealized example, let's pretend that splitting up the processor's four-phase lifecycle into eight equally long (0.5ns) pipeline stages is a trivial matter, and that the results look like what you find in figures PIPELINING.6.1 and PIPELINING.6.2. (In reality, this task is not trivial and involves a number of tradeoffs. As a concession to that reality, I've chosen to use the eight stages of a real-world pipeline, the MIPS pipeline, in the diagrams below, instead of just splitting each of the four traditional stages in two.)

Because pipelining requires that each pipeline stage take exactly one clock cycle to complete, then our clock cycle can now be shortened to 0.5ns in order to fit the lengths of the eight pipeline stages. Take a look at figures PIPELINING.6.1 and PIPELINING.6.2 below to see the impact that this increased number of pipeline stages has on the number of instructions completed per unit time.

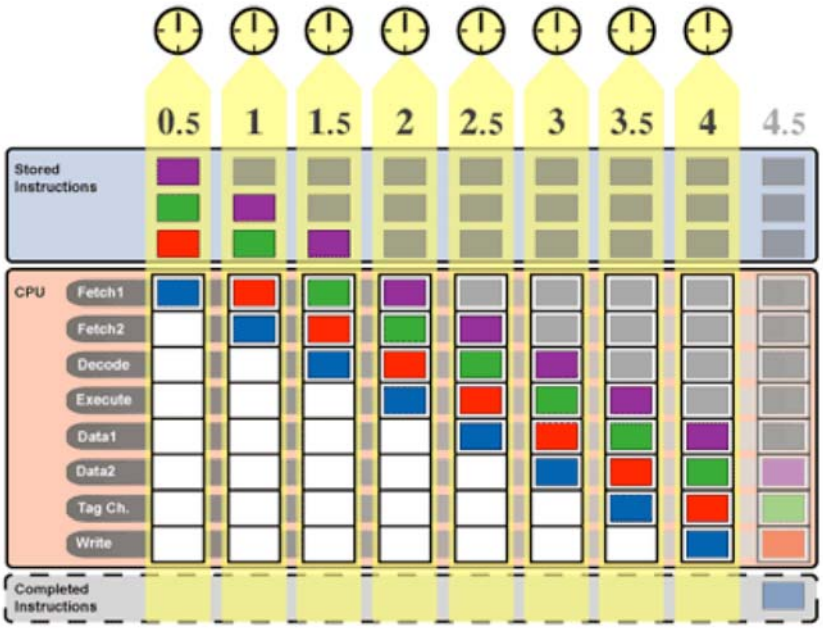


Figure PIPELINING.6.1: An eight-stage pipeline

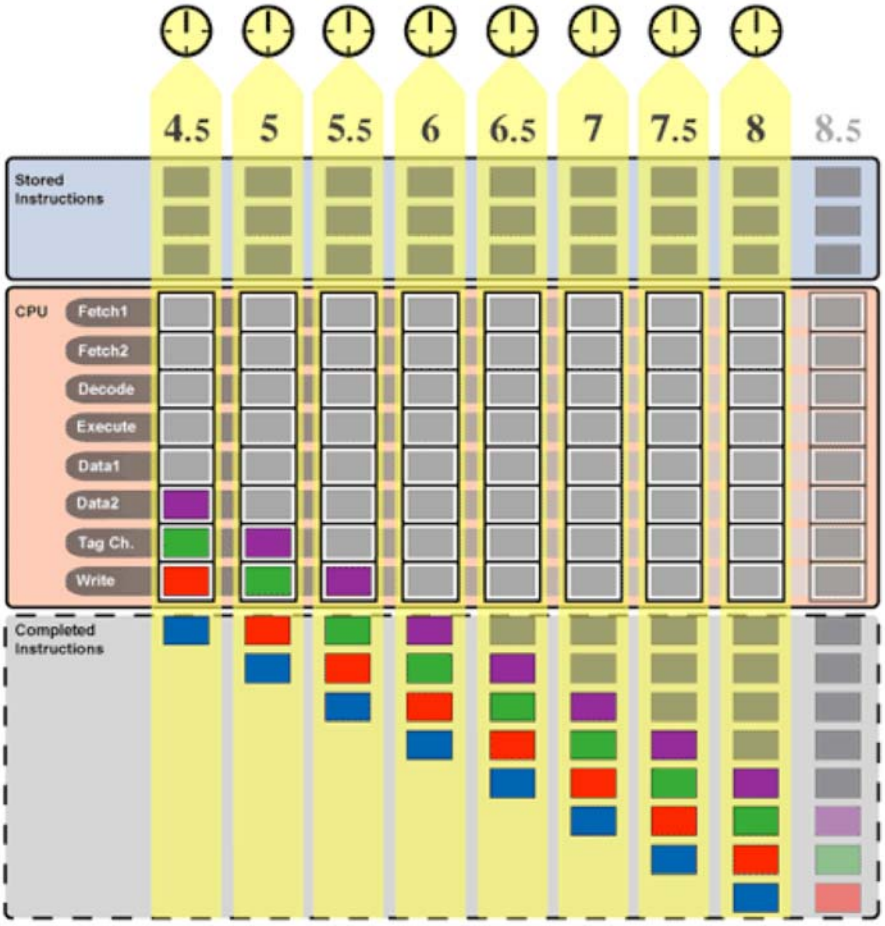


Figure PIPELINING.6.2: An eight-stage pipeline

Our single-cycle processor could complete one instruction every four nanoseconds, for a completion rate of 0.25 instructions/ns, and our four-stage pipelined processor could complete one instruction every nanosecond for a completion rate of 1 instructions/ns. The eight-stage processor depicted above improves on both of these by completing one instruction every 0.5ns, for a completion rate of 2 instructions/ns. Note that because each instruction still takes 4ns to execute, the first four nanoseconds of the eight-stage processor are still dedicated to filling up the pipeline. But once the pipeline is full, the processor can begin completing instructions twice as fast as the four-stage processor and eight times as fast as the single-stage processor.

This eight-fold increase in completion rate versus a single-cycle design means that our eight-stage processor can execute programs much faster than either a single-cycle or a four-stage processor. But does the eight-fold increase in completion rate translate into an eight-fold decrease in program execution time? Not exactly.

Completion rate and program execution time

If the program that the single-cycle processor in figure PIPELINING.4 is running consisted of only the four instructions depicted, then that program would have a program execution time of 16 ns, or 4 ns/instruction x 4 instructions. If the program consisted of, say, seven instructions, it would have a program execution time of 4 ns/instruction x 7 instructions = 28 ns. In general, a program's execution time is equal to the processor's instruction completion rate (number of instructions completed per nanosecond) multiplied by the total number of instructions in the program.

program execution time = instruction completion rate x number of instructions in program.

In the case of a non-pipelined, single-cycle processor, the instruction completion rate (X ns per 1 instruction) is simply the inverse of the instruction execution time (1 instruction per X ns). With pipelined processors, this is not the case.

If you look at the "Completed Instructions" box of the four-stage processor in figure PIPELINING.5, you'll see that a total of five instructions have been completed at the start of the ninth nanosecond. In contrast, the non-pipelined processor sports two completed instructions at the start of the ninth nanosecond. Five completed instructions in the span of eight nanoseconds is obviously not a fourfold improvement over two completed instructions in the same time period, so what gives?

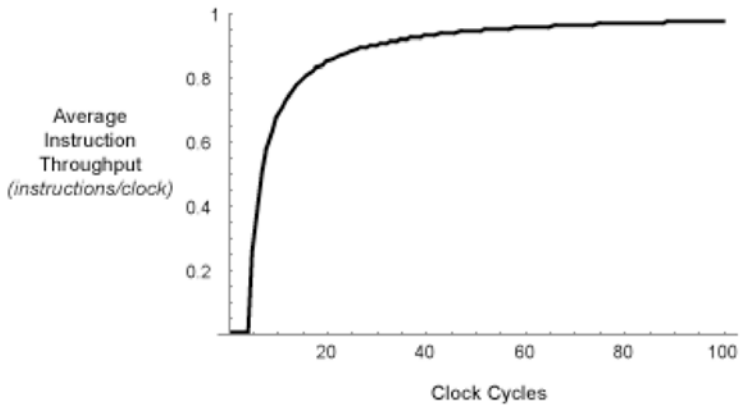
You have to remember that it took the pipelined processor four nanoseconds initially to fill up with instructions; the pipelined processor did not complete its first instruction until the end of the fourth nanosecond. Therefore it completed fewer instructions over first 8 ns of that program's execution than it would have had the pipeline been full for the entire 8 ns.

When the processor is executing programs that consist of thousands of instructions, then as the number of nanoseconds stretches into the thousands the impact on program execution time of those four initial nanoseconds, during which only one instruction was completed, begins to vanish and the pipelined processor's advantage

begins to approach the fourfold mark. For example, after 1000 nanoseconds, the non-pipelined processor will have completed 250 instructions (1000ns ⁄ 0.25 instructions/ns = 250 instructions), while the pipelined processor will have completed 996 instructions ((1000ns – 4ns) ⁄ 1 instructions/ns)—a 3.984-fold improvement.

What I've just described using the concrete example above is the difference between a pipeline's maximum theoretical completion rate and its real-world average completion rate. In the previous example, the four-stage processor's maximum theoretical completion rate, i.e. its completion rate on cycles when its entire pipeline is full, is 1 instructions/ns. However, the processor's average completion rate during its first eight nanoseconds is 5 instructions/8 ns = 0.625 instructions/ns. The processor's average completion rate improves as it passes more clock cycles with its pipeline full, until at 1000 nanoseconds its average completion rate is 996 instructions/1000 ns = 0.996 instructions/ns.

At this point, it might help to look at a graph of our four-stage pipeline's average completion rate as the number of nanoseconds increases:



Graph Pipelining.1: Average completion rate of a four-stage pipeline

You can see how the processor's average completion rate stays at zero until the 4 ns mark, after which point the pipeline is full and the processor can begin completing a new instruction on each nanosecond, causing the average completion rate for the entire program to curve upward and eventually to approach the maximum completion rate of 1 instructions/ns.

So in conclusion, a pipelined processor can only approach its ideal completion rate if it can go for long stretches with its pipeline full on every clock cycle.

Instruction throughput and pipeline stalls

In spite of what I may have led you to believe in the preceding pages, pipelining isn't totally "free". Pipelining adds some complexity to the microprocessor's control logic, because all of these stages have to be kept in sync. Even more important for our present discussion, though, is the fact that pipelining adds some complexity to microprocessor design and to the ways in which we assess the processor's performance.

Up until now, we've talked about microprocessor performance only in terms of instruction completion rate, or the number of instructions that the processor's pipeline can complete each nanosecond. A more common performance metric in the real world is a pipeline's instruction throughput, or the number of instructions that the processor completes *each clock cycle*. You might be thinking that a pipeline's instruction throughput should always be 1 instructions/clock, because I stated above that a pipelined processor completes a new instruction at the end of each clock cycle *in which the write stage has been active*. But notice how the emphasized part of that definition qualifies it a bit; we've already seen that the write stage is inactive during clock cycles in which the pipeline is being filled, so on those clock cycles the processor's instruction throughput is 0 instructions/clock. In contrast, when the instruction's pipeline is full and the write stage is active, the pipelined processor has an instruction throughput of 1 instructions/clock.

So just like there was a difference between a processor's maximum theoretical completion rate and its average completion rate, there's also a difference a processor's maximum theoretical instruction throughput and its average instruction throughput.

1. Instruction throughput: the number of instructions that the processor finishes executing on each clock cycle. You'll also see instruction throughput referred to as instructions per clock (IPC).
2. Maximum theoretical instruction throughput: the theoretical maximum number of instructions that the processor can finish executing on each clock cycle. For the simple kinds of pipelined and non-pipelined processors described so far, this number is always one instruction per cycle (1 instructions/cycle or 1 IPC).
3. Average instruction throughput: the average number of instructions per clock (IPC) that the processor has actually completed over a certain number of cycles.

A processor's instruction throughput is closely tied to its instruction completion rate—the more instructions that the processor completes each clock cycle, the more instructions it also completes each nanosecond. We'll talk more about the relationship between these two metrics in a moment, but for now just remember that a higher instruction throughput translates into a higher instruction completion rate, and hence better performance.

Pipeline stalls

In the real world, a processor's pipeline can be found in more conditions than just the two described so far—i.e. a full pipeline or a pipeline that's being filled. Sometimes, instructions get hung up in one pipeline stage for multiple cycles.. There are a number of reasons why this might happen, but when it happens, the pipeline is said to stall. When the pipeline stalls, or gets hung in a certain stage, all of the instructions in the stages below the one where the stall happened continue advancing normally, while the stalled instruction just sits in its stage and backs up all the instructions behind it. In the figure below, the orange instruction is stalled for two extra cycles in the fetch stage. Because the instruction is stalled, a new gap opens ahead of it in the pipeline for each cycle that it stalls. Once the instruction starts advancing through the pipeline again, the gaps in the pipeline that were created by the stall, gaps that are commonly called

"pipeline bubbles", travel down the pipeline ahead of the formerly stalled instruction until they eventually leave the pipeline.

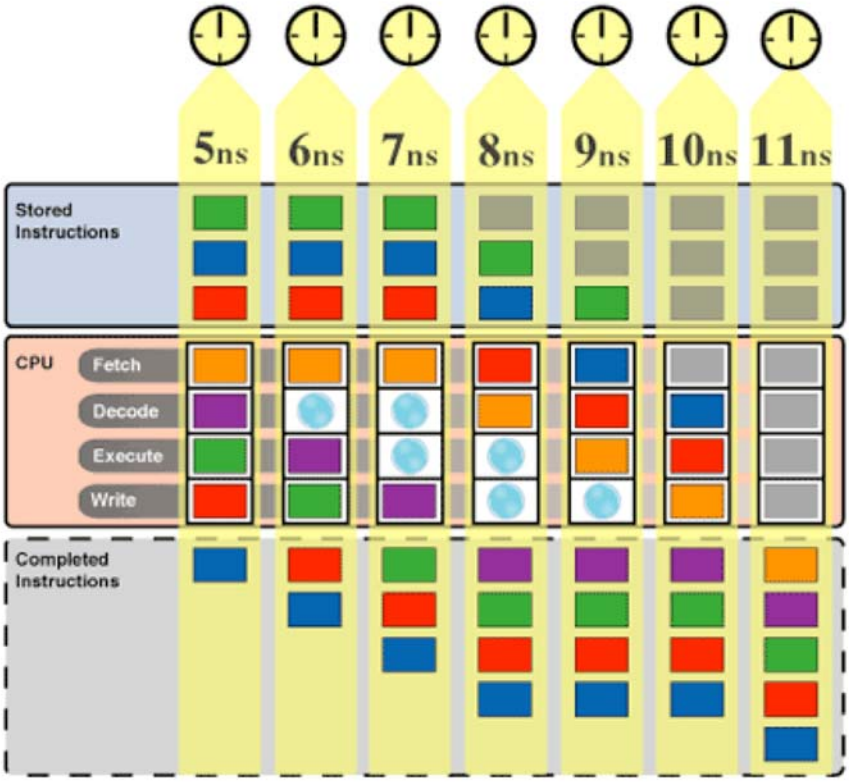


Figure PIPELINING.7: Pipeline stalls in a four-stage pipeline

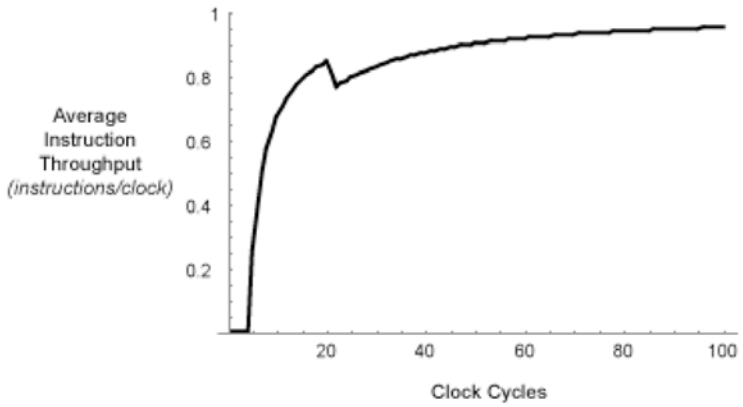
Pipeline stalls, or bubbles, reduce a pipeline's average instruction throughput, because they prevent the pipeline from attaining the maximum throughput of one finished instruction per cycle. In figure PIPELINING.4 above, the orange instruction has stalled in the fetch stage for two extra cycles, creating two bubbles that will propagate through the pipeline. (Again, the bubble is simply a way of signifying that the pipeline stage in which the bubble sits is doing no work during that cycle.) Once the instructions below the bubble have completed, the processor will complete no new instructions until the bubbles move out of the pipeline. So at the ends of clock cycles 9 and 10, no new instructions are added to the "Completed Instructions" region; normally, two new instructions would be added to the region at the ends of these two cycles. Because of the bubbles, though, the processor is two instructions "behind schedule" when it hits the 11th clock cycle and begins racking up completed instructions again.

The more of these bubbles that crop up in the pipeline, further away the processor's actual instruction throughput is from its maximum instruction throughput. In the example above, the processor should ideally have completed 7 instructions by the time it finishes the 10th clock, for an average instruction throughput of 0.7 instructions per clock. (Remember, the maximum instruction throughput possible under ideal conditions is 1 instruction per clock, but many more cycles with no bubbles would be needed to approach that maximum.) But because of the pipeline stall, the processor only completes 5 instructions in 10 clocks, for an average instruction throughput of 0.5

instructions per clock. 0.5 instructions per clock is half the theoretical maximum instruction throughput, but of course the processor spent a fwe clocks filling the pipeline so it couldn't have achieved that after 10 clocks even under ideal conditions. More important is the fact that 0.5 instructions per clock is only 71% of the throughput that it could have achieved were there no stall (i.e. 0.7 instructions per clock).

Because pipeline stalls decrease the processor's average instruction throughput, they increase the amount of time that it takes to execute the currently running program. If the program in the example above consisted of only the seven instructions pictured, then the pipeline stall would have resulted in a 29% program execution time increase.

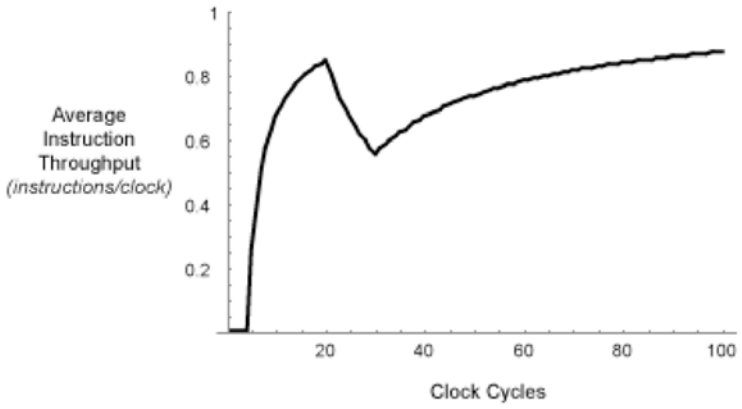
Let's take look at a graph that shows what that two-cycle stall does to the average instruction throughput:



Graph PIPELINING.2: Average instruction throughput of a four-stage pipeline with a two-cycle stall

The processor's average instruction throughput stops rising and begins to plummet when the first bubble hits the write stage, and it doesn't recover until the bubbles have left the pipeline.

To get an even better picture of the impact that stalls can have on a pipeline's average instruction throughput, let's now look at the impact that a stall of ten cycles (starting in the fetch stage of the 18th cycle) would have over the course of 100 cycles in the four-stage pipeline described so far.



Graph PIPELINING.3: Average instruction throughput of a four-stage pipeline with a two-cycle stall

After the first bubble of the stall hits the write stage in the 20th clock the average instruction throughput stops increasing and begins to decrease. For each clock in which there's a bubble in the write stage, the pipeline's instruction throughput is 0 instructions/clock so its average instruction throughput for the whole period continues to decline. After the last bubble has worked its way out of the write stage, then the pipeline begins completing new instructions again at a rate of 1 instructions/cycle and its average instruction throughput begins to climb. And when the processor's instruction throughput begins to climb, so does its completion rate and its performance on programs.

Quite a large percentage of the architectural features of the processors that I've covered over the years have been dedicated to preventing pipeline stalls. Branch prediction in particular comes to mind, because it's an essential tool for keeping the processor from stalling for a large number of cycles in the fetch phase.

Instruction latency and pipeline stalls

Before closing out our discussion of pipeline stalls, I should introduce another term that you'll be seeing periodically throughout the rest of this article: instruction latency. An instruction's latency is the number of clock cycles it takes for the instruction to pass through the pipeline. For a single-cycle processor, all instructions have a latency of one clock cycle. In contrast, for the simple four-stage pipeline described so far, all instructions have a latency of four cycles. To get a visual image of this, take one more look at the blue instruction in figure PIPELINING.4 above; this instruction takes four clock cycles to advance, at a rate of one clock cycle per stage, through each of the four stages of the pipeline. Likewise, instructions have a latency of eight cycles on an eight-stage pipeline, twelve cycles on a twelve-stage pipeline, and so on.

In real-world processors, instruction latency is not necessarily a fixed number that's equal to the number of pipeline stages. Because instructions can get hung up in one or more pipeline stages for multiple cycles, each extra cycle that they spend waiting in a pipeline stage adds one more cycle to their latency. So the instruction latencies given above, i.e. four cycles for a four-stage pipeline, eight cycles for an eight-state pipeline, etc., represent *minimum* instruction latencies. Actual instruction latencies in pipelines of

any length can be longer than the depth of the pipeline, depending on whether or not the instruction stalls in one or more stages.

Limits to pipelining

As you can probably guess, there are some practical limits to how deeply you can pipeline an assembly line or a processor before the actual speedup in completion rate that you gain from pipelining starts to become significantly less than the ideal speedup that you might expect. In the real world, the different phases of an instruction's lifecycle don't easily break down into an arbitrarily high number of shorter stages of perfectly equal duration. Some stages are inherently more complex and take longer than others. But because each pipeline stage must take exactly one clock cycle to complete, then the clock pulse that coordinates all the stages can be no faster than the pipeline's slowest stage. In other words, the amount of time it takes for the slowest stage in the pipeline to complete will determine the length of the CPU's clock cycle and thus length of every pipeline stage. This means that the pipeline's slowest stage will spend the entire clock cycle working, while the faster stages will spend part of the clock cycle idle. Not only does this waste resources, but it increases each instruction's overall execution time by dragging out some phases of the lifecycle to take up more time than they would were the processor not pipelined—all of the other stages must wait a little extra time each cycle while the slowest stage plays catch-up.

So as you slice the pipeline more finely in order to add stages and increase throughput, the individual stages get less and less uniform in length and complexity, with the result that the processor's overall instruction execution time gets longer. Because of this feature of pipelining, one of the most difficult and important challenges which the CPU designer faces is that of balancing the pipeline so that no one stage has to do more work to do than any other. The designer must distribute the work of processing an instruction evenly to each stage, so that no one stage takes up too much time and thus slows down the entire pipeline.

You can see the evidence of this difficult balancing act most clearly in the Pentium 4's drive stages, which are stages whose sole purpose is to drive signals across the chip. If Intel hadn't given these two signal propagation periods their own separate stages, then all of the Pentium 4's pipeline stages would've had to have been lengthened because of signal propagation delays in just two portions of the pipeline.

Clock period and completion rate

If the pipelined processor's clock cycle time, or clock period, is longer than its ideal length (i.e. unpipelined instruction execution time / pipeline depth), and it always is, then the processor's completion rate will suffer. If the instruction throughput stays fixed, say at 1 instruction/clock, then as the clock period increases the completion rate decreases. Because new instructions can be completed only at the end of each clock cycle, a longer clock cycle translates into fewer instructions completed per nanosecond, which in turn translates into longer program execution times.

To get a better feel for the relationship between completion rate, instruction throughput, and clock cycle time, let's take our eight-stage pipeline from figure

PIPELINING.6 and increase its clock cycle time to 1ns instead of 0.5ns. Its first nine nanoseconds of execution would then look as follows:

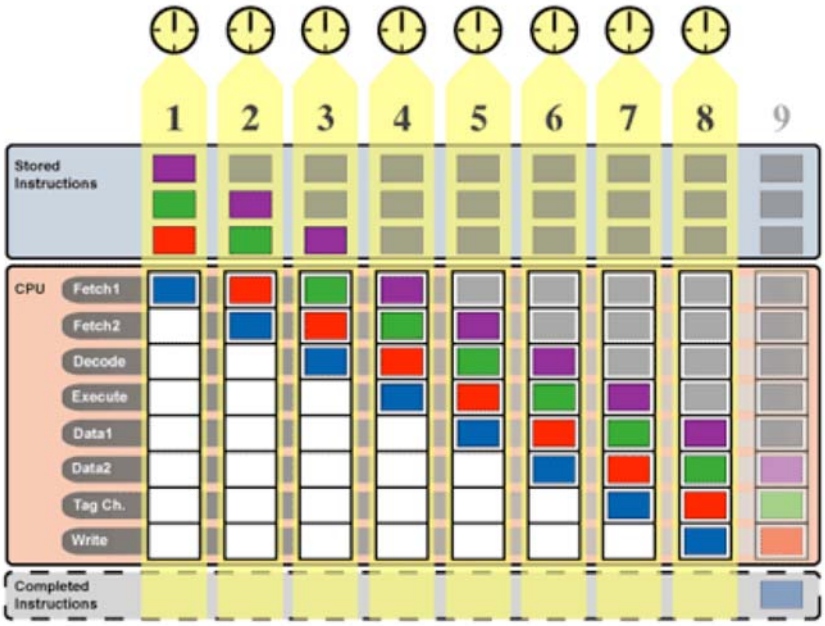
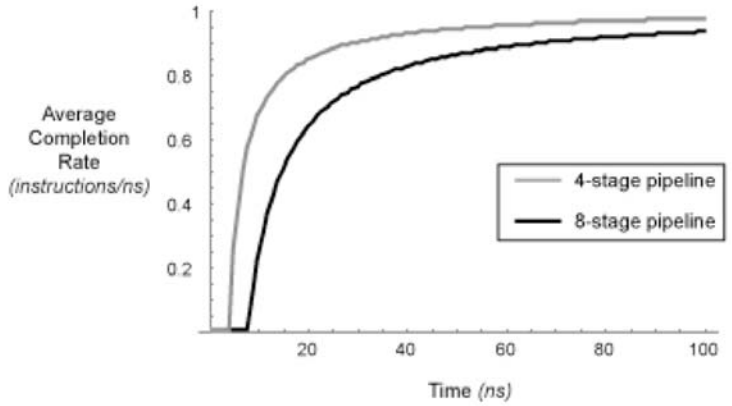


Figure PIPELINING.8: An eight-stage pipeline with a 1ns clock period

As you can see, the instruction execution time has now increased from an original time of four nanoseconds to a new time of eight nanoseconds, which means that the eight-stage pipeline does not complete its first instruction until the end of the eighth nanosecond. Once the pipeline is full, the processor pictured above begins completing instructions at a rate of one instruction per nanosecond. This completion rate is half the completion rate of the ideal eight-stage pipeline with the 0.5ns clock cycle time. It's also the exact same completion rate as the 1 instruction/ns completion rate of the ideal four-stage pipeline. In short, the longer clock cycle time of our new eight-stage pipeline has robbed the deeper pipeline of its completion rate advantage. Furthermore, the eight-stage pipeline now takes twice as long to fill. Take a look at what this doubled execution time does to the eight-stage pipeline's average completion rate curve versus the same curve for a four-stage pipeline.



Graph PIPELINING.4: Average instruction completion rate for 4- and 8-stage pipelines with a 1ns clock period

It takes longer for the slower eight-stage pipeline to fill up, which means that its average completion rate—and hence its performance—ramps up more slowly when the pipeline is first filled with instructions. There are many situations in which a processor that's executing a program must flush its pipeline entirely and then begin refilling it from a different point in the code stream. In such instances, that slower-ramping completion rate curve causes a performance hit.

Superscalar computing and pipelining

Superscalar computing allows a microprocessor to increase the number of instructions per clock that it completes beyond 1 instruction/clock. Recall that 1 instruction/clock was the maximum theoretical instruction throughput for a pipelined processor as described above. Because a superscalar machine can have multiple instructions in multiple write stages on each clock cycle, the superscalar machine can complete multiple instructions per cycle. If we adapt our previous pipeline diagrams to depict two-way superscalar execution, they look as follows:

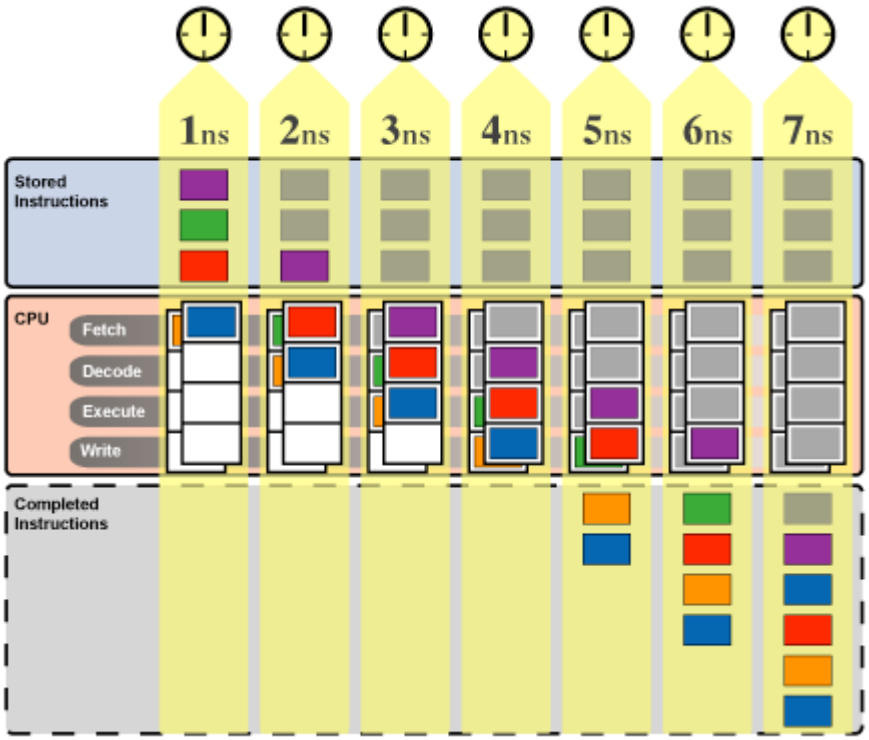


Figure PIPELINING.11: Superscalar execution and pipelining combined

In the figure above, two instructions are added to the "Completed Instructions" box on each cycle once the pipeline is full. The more ALU pipelines that a processor has operating in parallel, the more instructions that it can add to that box on each cycle. Thus superscalar computing allows us to increase a processor's IPC by adding more hardware. There are some practical limits to how many instructions can be executed in parallel, so the processor doesn't always reach the ideal completion rate of two instructions per clock. Sometimes, the processor can't find two instructions to execute in parallel on a particular cycle, which means that it must insert a pipeline bubble into one of the pipelines on that cycle, bringing the completion rate down.

SMT and pipelining

Keeping up the processor's average completion rate on a superscalar, pipelined machine involves finding ways to schedule instructions to execute in parallel on each cycle. But because the code stream is designed to be sequential, there are some inherent limits to how much parallelism that the processor can extract from it.

One of the ways that the latest processors for Intel, IBM, and AMD solve this problem is by asking the programmer and/or compiler to make the code stream as explicitly parallel as possible. This means identifying portions of an application that can be split into discrete and independent tasks, and assigning those tasks to separate threads of execution. This process of multithreading an application in effect turns a single sequential code stream into a bundle of two or more related code streams that can execute in parallel.

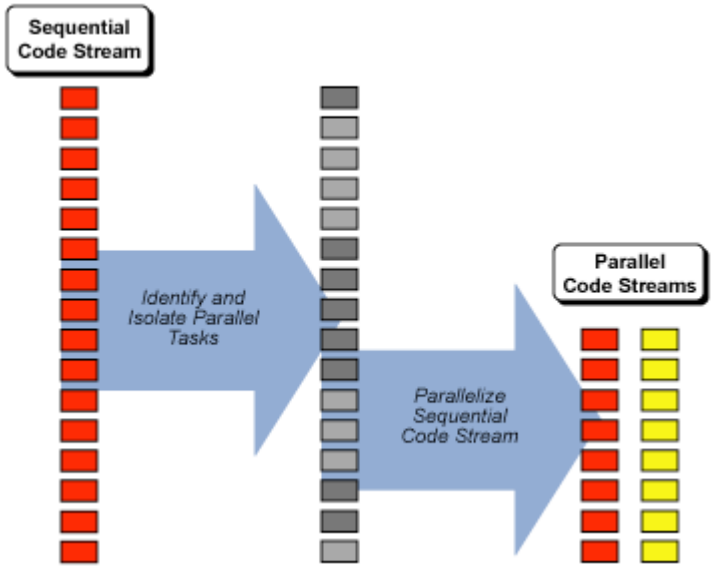


Figure PIPELINING.12: Multithreaded application design

By designing the code stream from the ground up as a bundle of smaller, simultaneously executing code streams, some of the burden of extracting instruction-level parallelism is moved from the processor to the programmer/compiler. Note that compilers are notoriously poor at such parallelization, so it usually falls to the programmer to design the application so that it can be broken into multiple threads.

Not all applications lend themselves to multithreaded implementation. In such cases, SMT offers few advantages. Well talk about why this is the case in a moment, though. For now, let's look at the other way that SMT can improve application performance.

SMT not only improves application performance by increasing a multithreaded application's average completion rate, but it also can prevent the completion rate from dropping to zero as a result of cache misses and memory latencies. When the processor is executing two threads simultaneously, and one of the threads stalls in the fetch stage (i.e., there's a cache miss so the thread must fetch instructions or data from main memory), the processor can continue normally executing the non-stalled thread. In a non-SMT processor, pipeline bubbles would crop up beneath the stalled instruction and propagate into the execution core, killing the application's average completion rate. The SMT processor, on the other hand, can schedule instructions from the non-stalled thread for execution in those available pipeline slots.

If the two threads in the scenario above are from the same application, then SMT prevents the multithreaded application's completion rate from dropping to zero for the duration of the stall by continuing to execute code from a non-stalled thread. This helps to keep the application's average completion rate up and to reduce the time it take to execute the application. On the other hand, if the two threads are from two separate single-threaded applications, then the completion rate for the application with the stalled thread will drop to zero, while the completion rate for application with the non-stalled thread may either stay the same or improve. In cases where the non-stalled

application's completion rate improves, it's because the stalled thread is no longer tying up processor resources that the non-stalled thread needs access to.

In the end, it's likely that two single-threaded applications will execute slightly slower on an SMT processor, depending on the types of applications and other conditions, because they may contend with each other for shared resources (e.g., cache, execution units, queue entries, etc.). Most SMT designs use a variety of techniques to minimize such contention and its effects, but it's always a factor.

Conclusions

In the end, the performance gains brought about by pipelining depend on two things:

1. Pipeline stalls must be avoided. As we've seen earlier, pipeline stalls cause the processor's completion rate and performance to drop.
2. Pipeline fills must be avoided *at all costs*. Filling up the processor's pipeline takes a serious toll on both completion rate and performance. This is especially true when a pipeline is very long but has a clock rate that's comparable to that of a processor with a shorter pipeline.

The deep buffers that make up the Pentium 4's instruction window (described in detail here), are aimed at eliminating pipeline stalls. Again, the Pentium 4 pays a relatively high price in terms of instruction tracking and buffering overhead in order to keep stalls from sapping performance.

As I've outlined in previous articles, the Pentium 4 and Prescott spend a ton of transistor resources on branch prediction in order to eliminate unnecessary pipeline fills. Pipeline fills are extremely deadly for performance on a hyperpipelined machine, especially when the clock rate isn't high enough to compensate, so Prescott especially pays a fairly high cost in transistors and power consumption for its elaborate branch prediction logic.

Furthermore, Prescott will have to scale to a significantly higher clock rate than that of the Pentium 4 if it's to perform better on branchy, poorly predictable code. At its current, relatively low clock rate, Prescott is going to have a very hard time with anything but the most predictable, non-branchy code types (e.g. media and gaming applications). As Intel tries to squeeze another year or two out of Prescott, expect to see it paired with an extremely large amount of on-die cache. This cache will help with branchy integer code, like that found in server applications.

It should also be clear at this point why Prescott is in so much trouble that Intel has turned the entire company around and embarked on a significantly different course involving shallower pipelines and multicore designs. As I demonstrated above, frequency scaling is critical if a deeply pipelined machine is to perform well, and frequency scaling is exactly what all microprocessor companies are facing power-related problems with at the 90nm transition. In fact, frequency scaling problems could be said to be the Achilles heel of a hyperpipelined design, and at the problems that are preventing such scaling are like the proverbial poison arrow right in Prescott's heel.

Date	Version	Changes
9/27/2004	1.0	Release