Understanding CPU Pipelining Through Simulation Programming

Michael D. Filsinger
University of Cincinnati

Abstract

Understanding the operation of modern Central Processing Units (CPUs) is essential for all Computer Engineering Technology students, but the black box nature of the CPU prevents the easy demonstration of many of the features of a modern CPU. In particular, pipelining has a tremendous effect on the real-world performance of a CPU. Typically, benchmarks are used to compare different processors, though the validity of such results is considered highly questionable. Alternatively, a variety of software packages exist for simulating the operation of a CPU. However, I have found in my Computer Architecture class that making the students write their own simple simulator programs results in a better understanding of some of the design issues involved in CPU performance. In this way, students can study the effects of the pipeline without needing to worry about any of the other details.

Introduction

All students in the Electrical and Computer Engineering Technology programs at the University of Cincinnati take a Junior/Senior level course in Computer Architecture. One of the most important concepts from this class is the effect of pipelining on Central Processing Unit (CPU) performance. This is one of the most fundamental issues facing CPU designers today. Unfortunately, it is difficult to demonstrate the effects through practical experimentation, as one cannot simply compare the performance of a non-pipelined CPU versus a pipelined CPU, since the performance will likely have been strongly affected by other optimizations in the (presumably newer) pipelined CPU.

This paper will examine one project I use to demonstrate this concept. In this project, students write a simulation of a series of “instructions” moving through a CPU data path. This simulation takes three forms: a single-cycle implementation, where each instruction takes exactly one clock cycle to execute (but this clock cycle must be long enough to handle all of the requirements of the longest instruction type); a multi-cycle implementation, where each instruction can take multiple clock cycles, depending on how many subtasks are involved in its execution; and a pipelined implementation, where multiple instructions can simultaneously be in different partial stages of execution. By writing simulations of this process, students can focus on the effects of pipelining on the performance of their simulated system. For the sake of simplicity, we do not actually consider what the instructions are, nor do we take branch or data hazards into consideration.

In later sections, I outline the basic problem of CPU pipelining, the actual programming project assigned to the students, the benefits of this approach, and student feedback. I also present an
expanded project for more advanced students using object-oriented techniques to more
accurately simulate actual instructions moving through the CPU pipeline.

Problem Description

The CPU is the single most constrained resource in a computer system. The execution of
instructions is an inherently sequential process, with the results of one instruction typically being
used in the calculation of the next. As a result, a CPU can only execute a single instruction at
one time.

Improving the performance of a CPU is a complex task. The most obvious approach is to make
each instruction use less time in execution. This represents an increase in raw “clock speed”,
typically represented in units such as gigahertz (GHz). However, technological limitations
prevent the unlimited increase in clock speed. The execution of an instruction involves a
sequence of transistor responses, and only so much can be done to decrease the time required for
each response. Note that some instructions are more complex than others, and the length of a
clock cycle must be long enough to accommodate the most complex instruction. As a result, all
instructions are limited to the speed of the slowest instruction.

One possible partial solution is to optimize the implementation of complex instructions. In other
words, we attempt to reduce the number of sequential transistor responses required for these
instructions. As an example, a brute force approach to performing integer addition using a
method called a “ripple adder” requires approximately C*N sequential transistor interactions
(where C is a constant and N is the number of bits contained in an integer on that system). Using
a “sum of products” approach to performing the addition, the number of transistors is increased
dramatically, but the “depth” of the transistor interactions can be reduced to a small constant.

Such optimizations can help the problem of increasing the clock speed, but we are still faced
with an absolute limit (based on the existing technologies and production techniques). A further
improvement can be created by breaking each instruction into a series of sub-steps. Each sub-
step would be executed in a single clock cycle. An instruction would now require multiple clock
cycles to complete, but each clock cycle involves less work, so the clock cycle can be shorter. In
this way, simple instructions can complete in only a few (relatively short) clock cycles, while
complex instructions can take many clock cycles to complete.

The following is a 5-cycle implementation based on the MIPS architecture as presented by
Patterson and Hennessy\(^2\).

a. Instruction Fetch
b. Instruction Decode and Register Fetch
c. Arithmetic Computation
d. Memory Read or Write
e. Register Write
A much greater improvement can be achieved by noting that each sub-step from the multi-cycle implementation involves a different set of circuitry from within the CPU. When a particular sub-step is being performed, the rest of the CPU is idle. While one instruction uses one part of the CPU, other instructions could theoretically use the otherwise idle parts of the CPU. In such a way, a degree of parallelism can occur, resulting in a dramatic improvement in instruction throughput.

Such an implementation is called a pipelined implementation. The CPU is broken up into semi-independent, sequential stages. An instruction moves from stage 1 to stage 2 to stage 3 and so on, until it has passed through all stages of the CPU. When the instruction moves from stage 1 to stage 2, the next instruction can now enter the now-empty stage 1. To the extent that one instruction does not depend on the results of other “nearby” instructions, a speed increase of roughly a factor of the number of stages in the pipeline can be achieved. Unfortunately, instructions do tend to interact with each other, and various types of “hazards” (most notably: data hazards, where one instruction’s computation depends on the result of the previous instruction’s computation; and branch hazards, where, due to a branch in the execution path, it is not immediately known which instruction will be the “next” one to be executed) exist. The existence of these hazards makes actual real-world pipeline performance far less than the ideal, and the problem only becomes worse with long pipelines.
As can be seen, a Pipelined implementation combines the short clock cycle length of the multi-cycle implementation with the low number of clock cycles found with the single-cycle implementation. At least in the absence of branch and data hazards, the pipelined implementation represents the best of both worlds.

**Performance Measurement**

The goal of the laboratory described in this paper is to demonstrate the theoretical performance benefits of using a pipelined CPU implementation over using a non-pipelined implementation. In principle, one could simply run the same set of instructions through two different processors and compare the results. In practice, the situation is far more complex.

First, in order to be able to execute exactly the same instructions, the two processors being compared must belong to the same family of processors, such as the Intel x86 family or the Motorola 68xx family of processors. However, in order to compare a pipelined processor to a non-pipelined processor, one must look at different generations of processors, such as an 8086 processor versus an 80386 processor. While accounting for the differences in clock speeds is easy, later generations of processors generally have a variety of other performance enhancements as well, which make isolating the benefits of the pipeline difficult.

Furthermore, the actual performance increase derived from using a pipelined implementation does not match the theoretical increase. Branch and data hazards significantly erode the performance gains. A gain will always be seen from using a pipelined implementation, but that gain may fall far short of the roughly N-times (where N is the number of stages in the pipeline) increase predicted by theory. In fact, increasing the size of the pipeline might have negative effects on performance. As an example, in real-world applications, an Intel Pentium III processor will run approximately 30% faster than a Pentium IV run at the same clock speed. This is due to the much deeper pipeline on the Pentium IV. The actual performance increase of the Pentium IV comes from the fact that having many pipeline stages allows each stage to do less work, allowing a massive increase in clock frequency.

Despite these problems, the use of benchmarks has been a common way of measuring relative CPU performance. Unfortunately, software benchmarks do not yield valid general comparisons. All a benchmark will reveal is that one CPU performed better than another CPU in this benchmark. A different benchmark might yield a completely different result. In fact, this has been a problem when comparing Intel CPUs to AMD CPUs (nominally compatible CPU families). One benchmark might say that the Intel CPU is faster than the AMD, while another benchmark says that the AMD CPU is faster.

In any case, even if we accept benchmark results as valid, it remains difficult to isolate what part of the performance increase comes from the pipeline and what part comes from other enhancements. The only viable way to measure such benefits is to simulate the operation of a CPU in a tightly-controlled software environment. In such a way, one can compare two different CPU implementations where the only difference between the CPUs is that one is pipelined and the other is not.
A variety of software packages can be used to simulate a CPU. Some packages allow the student to actually fully specify the CPU, but such an approach is far more complex than necessary for the stated objective. Other packages allow a student to simulate a pre-designed processor. Unfortunately, such packages do not necessarily give as much of a hands-on feel as desired in a technology program. Furthermore, I prefer that the simulations in the lab resemble the MIPS architecture used by the course textbook.

**Project Description**

As an alternative to using an existing simulator package, students write simple simulations of their own. In this way, the students can easily examine the theoretical benefit of using a pipelined environment without the distraction of other aspects of the CPU. While the results represent the theoretical improvement as opposed to the real-world improvement, they are free of the dependence on the actual sequence of simulated instructions found in actual benchmark results.

The students construct a simple environment, where a linear sequence of “instructions” is passed through a simulated CPU. The actual instructions are not identified, and it is assumed that each instruction is independent of the other instructions – in other words, no hazards exist. Three simulations are actually written: a single-cycle CPU, a multi-cycle non-pipelined CPU, and a pipelined CPU.

---

**Pseudocode of Single-Cycle Program**

**Data:**

- cycles // Keeps track of total clock cycles used
- i // Instruction counter

**Procedures:**

main()
{
    cycles = 0
    for (i = 1 to the number of instructions)
    {
        cycles = cycles + 1
        output current state
    }
    output results (cycles * cycle length)
}
Pseudocode of Multi-Cycle Program

Data:

instructions[] // Initialized to the number of cycles required by each instruction.
cycles        // Keeps track of total clock cycles used
i            // Instruction counter

Procedures:

main()
{
    cycles = 0
    for (i = 1 to the number of instructions)
    {
        for (j = 1 to instructions[i])
        {
            cycles = cycles + 1
            output current state
        }
    }
    output results (cycles * cycle length)
}

Pseudocode of Pipelined Program

Data:

cycles        // Keeps track of total clock cycles used
i            // Instruction counter
pipeline[]    // Keeps track of which instruction is in which stage of the pipeline

Procedures:

main()
{
    cycles = 0
    i = 0
    while (an unfinished instruction exists)
    {
        cycles = cycles + 1
        // advance all instructions in pipeline
        for (j = pipeline length down to 1)
As can be seen, the simulations are extremely easy to write in virtually any programming language. The effect of the pipeline can be modeled without any need to model the actual machine language. Of course, the issues of data and branch hazards have not been addressed, but to do so would require designing a machine language and simulating actual “programs”. This aspect has been sacrificed in favor of simplicity. However, in the next section, a possible extension of the program is outlined which could be written by more advanced students.

A More Advanced Simulator

The main reason for keeping the simulation so trivial is that many of the students taking our Computer Architecture course have only taken a single ten-week course in C programming. While this course covers most of the basics, the students have not yet truly absorbed even the most basic concepts. More advanced students are capable of tackling a much richer problem.

Object-oriented design lends itself naturally to simulation problems. In this case, the CPU pipeline, the individual instructions, and possibly the registers are clearly objects. The simulation is about the interactions between these objects.

For the most part, modeling the computational aspects of an assembly language is trivial, since there are single-statement equivalents in all high level languages. The trick is to model the cycle by cycle interactions between these instructions.

First, build an inheritance hierarchy based on the class “Instruction”. Each instruction in the simulated assembly language is represented as a subclass of Instruction. The following list of instructions is sufficient to demonstrate the problems of data and branch hazards:

```
add  regD, reg1, reg2
sub  regD, reg1, reg2
mov  regD, const
jeq  reg1, reg2, trueDest, falseDest
jlt  reg1, reg2, trueDest, falseDest
```

Of course, additional instructions could be added to the list, but their behavior would be similar.

To account for data hazards, include a flag for each simulated register indicating whether that register has a pending uncalculated value. When an instruction enters the “Instruction Decode
and Register Fetch” stage, it checks this flag for each of its source registers. If the register is unavailable, the instruction does not advance to the next CPU stage. When the registers become available, the instruction will load their values, mark the destination register as unavailable, and advance to the next CPU stage. When the instruction enters the final stage (analogous to the various “Completion” stages of the MIPS implementation), it saves the result in the destination register and marks that register as available.

Branch hazards are even simpler. While advanced CPUs attempt to perform branch prediction, and in fact, one could attempt to model such an implementation, it is sufficient to simply force execution to “stall” until it is determined which branch will be taken. To do this, include a flag in the CPU object to indicate a stall waiting for branch resolution. When a branch instruction (jeq or jlt) is decoded, set this flag to TRUE and proceed with execution of the branch instruction. When the branch instruction completes, set the instruction counter appropriately and set this flag to FALSE, allowing other instructions to proceed.

Benefits

Essentially, two major benefits are derived from writing simulations as opposed to traditional benchmarking or pre-defined simulations. First, it demonstrates the theoretical effects of pipelining, without worrying about implementation details or the effects of other performance enhancements. Furthermore, unlike benchmark results, the results are consistent and independent of the actual sequence of “instructions” being simulated. By removing these distractions, the underlying concepts become more accessible to the student.

The second benefit is that this approach provides students with valuable programming experience, while not presenting them with challenges beyond their current level of ability. One of the major problems I encounter as a professor of Computer Engineering Technology is that many of the students have only had one course in C programming, and these students generally do not understand even the most basic programming concepts yet. This single, quarter-long course struggles to cover all of the basic ideas of programming, and students have no time to absorb and practice what they have learned. This assignment gives these students valuable experience, programming with functions, loops, and arrays - the most fundamental of programming skills.

Student Feedback

Only anecdotal feedback exists, in the form of informal personal comments and comments made in lab reports and teaching evaluations. In general, student comments fall into two categories. First, students seem to gain a better understanding of the benefits of pipelining. I have yet to find a pre-made simulation package that I find suitable for my purposes, so I have no way of comparing a student-written simulation to one simply used off the shelf. However, students definitely seem to understand the concepts much better than when using techniques such as benchmarking. Since the student must model the pipeline’s behavior instead of simply measuring obscure values, there is a much stronger hands-on connection between the experimental values and the theoretical concept.
The second class of comments is from students who complain that “This is not a programming class, so why am I required to write programs?” To some extent, I see this attitude as validation of my belief that the students need more programming practice, not less. In fact, many of these students later come back to me and tell me how much more confidence they have concerning programming after the assignment. Much of their initial resistance stems from their lack of confidence and practice. As they discover that the problem is relatively simple to solve, that resistance fades.

Conclusion

Many further applications of student-written simulation programs exist. In my Operating Systems class, I use this technique for demonstrating page replacement (memory allocation) algorithms, process scheduling, and other topics. Essentially, any application which involves interactions between system components or timesharing of a limited resource is an ideal candidate for student-written simulation. One can model only the desired interactions without dealing with the complexity of the overall system in which these interactions take place.

When studying Computer Science or Computer Engineering, many concepts exist which are very simple conceptually, but difficult to demonstrate practically. The writing of simulation programs to demonstrate these concepts is the best way I have yet found for making these ideas accessible to the students.

Bibliography


Biography

MICHAEL D. FILSINGER is an Assistant Professor of Electrical and Computer Engineering Technology at the University of Cincinnati. He received a BA in Mathematics and MS degrees in Mathematics and Computer Science from the University of Cincinnati in 1990, 1992, and 1994, respectively. In addition to teaching, he has served as a computer system administrator. He is a member of IEEE, ASEE, and the Phi Beta Kappa honor society.