Pipelining Implementation in a RISC Environment

Kevin Martin, CIS655, Summer 2020

***Abstract*—Pipelining is the method by which multiple resource sharing processes can run concurrently without interference. This is more a design philosophy than a definitive list of steps, and its application is not localized to one specific aspect of the computer. Additionally, there are multiple ways to implement pipelining using either software, hardware, or a combination of both. The advantages create far more efficient throughput of the given system through thoughtful design decisions.**

***Index Terms*—Pipeline, throughput, processor, scheduling**

I. Introduction

The pipelining process is fundamental to modern computer architecture and allows for a great degree of usefulness from each hardware component. Indeed, one of the limiting factors of a computer's performance is its hardware, so getting the most out of each part is critical. As technology improves in speed and storage, this burden may become lessened in certain applications, such as consumer-grade desktops, but in situations where processing power is limited and hardware is a luxury, the paradigm remains. As such, we will examine how intelligent code design, as further explained by M. Lam, combined with very specific hardware applications can drastically increase workload efficiency [1]. To measure this, we will look at multiple metrics, but in general, the concept of throughput will be used. Additionally, while pipelining can be applied to various areas, we will primarily focus on the CPU. The concept was developed to solve instruction decoding and information fetching for the processor and evolved to accommodate not only different instruction set architectures, but also every changing hardware components.

II. Background

A. CPU Scheduling

A processor can only run one program at a time and thus must rely on "context switching" to allow for seamless multitasking. More specifically, a processor can only execute a single instruction at any given time, so the ability to balance which program gets its instructions to the CPU is of great importance. Arguably, it is the most important function an operating system must perform. As the medium between the user and the hardware, the operating system is crucial to a usable computing environment. Allowing multiple programs to run in parallel via context switching is the heart of modern computing. It is important to note that even in a multi-CPU environment, context switching is still incredibly important and very much present.

It is important to understand this concept first before looking at pipelining, because the two concepts are deeply interwoven. As the CPU executes an instruction, it relies on the other surrounding components to have the information ready. For example, if a register needs to be accessed, the CPU must get the most current value of that register in order to perform an accurate operation. If a register is delayed in receiving relevant information, the entire operation will be incorrect. This is a foundational concept for CPU scheduling, and where we can first see why the need for sophisticated problem solving exists.

B. Scheduling Algorithms

The "choice" of which program gets time on the CPU ultimately boils down to the result of the operating system's choice of CPU scheduling algorithm. There are many, many variations, and we will just briefly introduce them here in order to understand their implications in the context of pipelining. Consider the basic first in first out (FIFO) approach, wherein whichever program first requests time on the CPU is awarded that time. On the one hand, it is simple to implement both from an algorithmic point of view and from the required hardware. On the other, it may not allow programs that require lots of CPU time to get enough to finish.

Next, shortest job first. This has the benefit of allowing little jobs to get out of the way first, thus potentially speeding up the overall time to completion. The difficulty in implementation is twofold. First, the time for each job must be computed. Often this time is not perfectly known and must rely on some amount of estimation. This can raise the potential for errors and inaccurate scheduling decisions. Second, those times must be stored and compared, which requires additional overhead.

Third, there is priority scheduling. Of the three mentioned so far, this is the "smartest" choice in that the jobs considered most important will be given priority. Again, this requires additional overhead, but also additional foresight in determining the relevant importance of each process. Choosing the incorrect priority will result in sub-optimal outcomes regardless of how well the scheduler follows protocol.

Paper submitted August 4, 2020.

III. Pipelining Concepts

A. Basic Comparison

To understand the benefits of pipelining, let us examine a very basic comparison between a series of processes that need to run on the CPU. Arora et. al suggest the following toy example to show how pipelining can be implemented in concept [2]. We have three processes, P1, P2, and P3, all of which are scheduled to run simultaneously. P1 was created first, followed by P2, and then P3. Additionally, P1 also has the highest priority (followed by P2 and P3, respectively). The operating system is using the third scheduling algorithm discussed, priority based. Without pipelining, P1 would run to completion, followed by P2, and then P3. Each program has three steps: fetch, decode, and execute. If each stage takes one CPU cycle the entire time to completion would be 9 cycles. See Figure 1 below:

	CPU Cycles								
	1	2	3	4	5	6	7	8	9
P1	P1	P1	P1						
P2				P2	P2	P2			
P3							P3	P3	P3

Figure 1: No pipelining

If, on the other hand, the CPU was allowed to employ pipelining (in this case via extra hardware), P2 could be fetched and decoded while P1 was still in execution. Similarly, P3 could be fetched while P1 was finishing and P2 had already been decoded. This would effectively bring the amount of CPU cycles down from 9 to 5, as shown in Figure 2 below:

	CPU Cycles								
	1	2	3	4	5	6	7	8	9
P1	P1	P1	P1						
P2		P2	P2	P2					
P3			P3	P3	P3				

Figure 2: With pipelining

B. RISC-Based Architecture

Now let us look at a real-world implementation, the Reduced Instruction Set Computer (RISC) architecture. This type of computer favors simpler instructions with less available commands to perform. The pros and cons are debatable, and outside the scope of this discussion, but suffice to say there are certainly valid reasons to use this approach. Mainly, in situations where hardware is at a premium, such as in embedded systems. As we have discussed previously, making the most of limited hardware is one of the main advantages of pipelining. S. Jain points out how "aggressive optimizations" need to be implemented for MIPS, a common RISC architecture [4]. Therefore, the RISC approach is great candidate for actual usage. As S. Kunkel et. al discuss, the approach is appropriate for systems ranging from cutting edge supercomputers down to "lower performance computer systems" because speedup is speedup, and any system can benefit from more efficiently designed architecture [3].

In the RISC architecture, there are five major stages for processing each instruction: fetch, decode, execute, memory, and write-back. We will quickly look at each to see why pipelining may be employed at some stages but not others.

The first stage, fetch, grabs the instruction from the dedicated instruction register. At this point, the computer does not know what is in the instruction, only that it has one ready. Additionally, the program counter (PC) is incremented to account for this new instruction.

Next, is the decode stage. Here not only is the instruction read, but also the relevant registers. So, if data from a specific register is needed in a calculation or comparison, its value becomes known at this time. Additionally, possible branch target addresses are computed at this time. The branch targets are coded as locations in memory, and they are now available once decoded.

Third, the actual execution of the instruction. Generally, the execution is carried out by the ALU for both arithmetic operations as well as completing the full memory address calculation (in the case of a branch).

Fourth, the memory access. If the request is for a load, the memory does a ready using the effective address. If store, then the data is written to them memory. Because the memory stage is so fast, the read is done on the rising part of the clock cycle, and the write on the falling. While technically fourth in line, this stage comprises both the read and write functions. This fact comes into play during our discussion on pipelining.

Finally, the write-back stage. This is where the register field is updated either from the ALU output or from the previously loaded value. Also, this is where the next instruction is prepped for delivery, and the cycle can begin again.

C. Pipelining in a RISC Environment

In our initial example in section A, we used a toy example with only three stages. Now that we understand that there are actually five main stages to the full execution of an instruction, we can see where pipelining can save us time. We can layer in the five stages on top of each other, dramatically reducing the overall time to completion. Assume the same three processes, P1, P2, and P3, only this time with the full five stages of RISC architecture. First see the complete picture without pipelining in Figure 3:

	CPU Cycles														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P1	F	D	X	M	W										
P2						F	D	X	M	W					
P3											F	D	X	M	W

Figure 3: RISC without pipelining

The result is now 15 cycles to complete the three processes. Now, with pipelining implemented, that amount of time is reduced to just seven cycles. Observe the improvement in Figure 4:

	CPU Cycles														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P1	F	D	X	M	W										
P2		F	D	X	M	W									
P3			F	D	X	M	W								

Figure 4: RISC with pipelining

However, we should note that, while impressive, this "perfect" pipelining situation and speedup exists only in theory. It would be quite incredible if three processes could be done in under half the time with this relatively simple setup. Issues begin to arise when multiple resources are competed for concurrently. We will next examine some of the potential problems with pipeline implementation, as well as the remedies to work around them.

IV. Pipelining Hazards

A. Three Potential Issues

As mentioned, it is not enough to just choose to exercise each step of the process as soon as physically possible. More intelligent decisions need to be made in order to facilitate accurate processing of the instruction. The potential complications for implementation are generally referred to as "hazards" and fall into three major categories: structural, data, and control. We will look at both the causes and potential solutions to each type. Note that the solutions can be in the form of software, hardware, or a combination thereof. When considering the hardware solutions, those come at a much greater expense than software. Physical hardware needs to be added to the device, which increases costs and requires additional space. In theory this can seem like the obvious choice, but in practice the cost/benefit must be carefully weighed before making any such choice.

B. Structural Hazards

Structural hazards are the result of resource conflicts. In this context, resources refer to items like the registers or memory. In short, when two processes require the same resource, a potential issue exists. This can occur in a situation where there is a single ALU, and two instructions happen to be in the phase where they will need to use it on the next clock cycle.

Unfortunately, this type of hazard is not easily solved with software/algorithm changes, and often the case is to add more hardware. In the case described above, an additional ALU may be the only practical solution.

C. Data Hazards

Data hazards occur during the read and write phases of program execution. Here, because of the increased speedup from pipelining, it is possible that the required data is not in the proper place when it is needed. To illustrate, consider an example with just two instructions: an add of the contents in register R1 and register R2 with the result being stored in register R3. The second instruction is another add, where the newly calculated contents of register R3 are to be added to whatever is stored in register R4, and the result stored in R5. Here, R3 is the potential hazard. The second instruction may actually call R3 *before* the result of the first instruction has been written into the register. That does not mean that the addition has not been performed, but rather that R3 does not have the result stored in it.

The solution to this could come from either software or hardware. Of course, there are tradeoffs to either approach, so the proper solution will come down to the specific application. To solve this with software, the introduction of a "stall" would be used. A stall in this context is basically an empty cycle for a specific instruction to allow another instruction to catch up. This is necessary because it depends on when the information is available for use from a register. In the case of our previous two add instruction situation, the second add could be delayed for two cycles to ensure that when it calls on the freshly calculated value of R3, it is up to date. More specifically, the value of R3 is available from the first add instruction halfway through the fifth cycle (the write-back cycle). R3 is needed at the beginning of the second cycle for the second add instruction. Thus, two stall cycles are needed to allow the value from the first add operation to be written into R3 before the second add instruction calls it.

The other way to solve this situation would be with additional hardware. In this case, we would need a forwarding unit which would allow the result of the first add operation to be available immediately after it was calculated and before it was written into the register. With a forwarding unit, no stalls need to be introduced in this specific example. Because stalls waste CPU cycles and thus time, the forwarding unit would be preferable from a pure efficiency standpoint. There could be cost and/or design limitations that prevent this addition. Furthermore, there are cases where, depending on where the data hazard is occurring, a forwarding unit still cannot eliminate the need for a stall. In such circumstances, a combination of the two approaches would provide the optimal solution.

D. Control Hazard

A control hazard arises when a branch or jump instruction is introduced. Up until now, we have examined code that executes in a purely sequential fashion. This makes it easier to see potential hazards and also visualize appropriate solutions to minimize them. In cases where one or more instructions may be skipped, the potential for a hazard exists. There are several ways to combat this situation with varying levels of sophistication.

One method is where program "assumes" that the branch will not be taken and thus continue to execute in a sequential order. This determination has been made over many years of observation and the resulting statistical likelihood. H. Young describes how, when this happens, the program will begin to setup the next sequential steps and will undo them if the

branch is taken [5]. During this undo time, stall cycles can be added (similar to the software solution of the data hazard) to allow the instruction sets to catch up and be ready to branch correctly.

A more sophisticated approach could be to keep track of the program itself and what it did previously. An example would be a loop where a set of instructions is set to execute *n* number of times. In this case, if the program assumes the loop will not branch, it will be right *n*-1/*n* percent of the time. This tracking can be achieved with only a single additional bit. So, while the overhead is higher than without prediction, the overhead is relatively small.

A third approach involves yet more overhead, and that is a two-bit prediction system. In this case the prediction is only changed upon two successive mispredictions. The result is an overall faster and more accurate program, but once again this comes at the expense of extra hardware.

V. Conclusion

Speed and efficiency are the two of the main drivers in computer architecture, and smart improvements in these areas result in novel gains for performance. Pipelining is the one of the best ways to optimize CPU scheduling to allow for the execution of multiple programs simultaneously. By taking advantage of how an instruction is decoded and executed, less time is wasted in preparing for the next one. While this technique is still necessary and heavily utilized on even the most modern processor architecture, it is essential in systems where cost and size of hardware must be minimized. While an elegant solution to a complex problem, pipelining is not without issues. Hazards can arise when multiple instructions are mistimed and the potential for inaccurate calculations exist. Fortunately, there are multiple ways of combating these to achieve safe and successful pipelining. Software choices, additional hardware, or a combination of both can mitigate these risks. While additional hardware usually yields the quickest and most optimized execution, it is not always viable due to external constraints. Thus, the choice of which technique to use is not always obvious and requires careful evaluation of the situation.

References

[1] M. Lam. 1988. Software pipelining: an effective scheduling technique for VLIW machines. SIGPLAN Not. 23, 7 (July 1988), 318–328. DOI: https://doi.org/10.1145/960116.54022

[2] H. Arora, D. Goel, and P Jain, “An Improved CPU Scheduling Algorithm” in *IJAIS*. vol. 6, No. 6, 2013, URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.470.4781&rep=rep1&type=pdf

[3] S. R. Kunkel and J. E. Smith. 1986. Optimal pipelining in supercomputers. SIGARCH Comput. Archit. News 14, 2 (May 1986), 404–411. DOI: https://doi.org/10.1145/17356.17403

[4] Suneel Jain. 1991. Circular scheduling: a new technique to perform software pipelining. SIGPLAN Not. 26, 6 (June 1991), 219–228. DOI: https://doi.org/10.1145/113446.113464

[5] Honesty Cheng Young. 1985. Evaluation of a decoupled computer architecture and the design of a vector extension (pipelined processor; delayed branch, code scheduling, software pipelining, queue register). Ph.D. Dissertation. The University of Wisconsin - Madison. Order Number: AAI8512336. URL (abridged): https://search.proquest.com/docview/303383720