Superscalar Pipelining Architecture and Multi-Pipeline Scheduling Policies

Tarun Bhalla†, Mohit Mittal‡

† Assistant Professor, Anand College of Engineering & Management, Kapurthala
‡ Assistant Professor, Anand College of Engineering & Management, Kapurthala

Abstract- In this paper, we present the process of pipelining using superscalar processor. A super-scalar processor is one that is capable of sustaining an instruction-execution rate of more than one instruction per clock cycle. Maintaining this execution rate is primarily a problem of scheduling processor resources (such as functional units) for high utilization. Multiple pipes are used for improving the performance of pipelining. A superscalar processor can be envisioned having multiple parallel pipelines, each of which is processing instructions simultaneously from a single instruction thread.

Index Terms- superscalar pipeline design, pipeline stalling, multipipeline scheduling i.e. in-order issue & in-order completion, in-order issue & out-order completion and out-order issue & out-order completion.

I. INTRODUCTION

In this superscalar pipelining, the processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem. There are essentially two pipelines in this design. Both pipelines having four processing stages labeled fetch, decode, execute and store respectively.[5]

The time taken by a computing system to perform a particular application is determined by three factors: the processor cycle time, the number of processor instructions required to perform the application, and the average number of processor cycles required to execute an instruction. System performance is improved by reducing one or more of these factors.[1]

In general, the cycle time is reduced via the implementation technology and memory hierarchy, the number of instructions is reduced via optimizing compilers and software design, and the average number of cycles per instruction is reduced via processor and system architecture. To illustrate, RISC processors achieve performance by optimizing all three of these factors.[1]

The simplicity of a RISC architecture permits a high-frequency implementation. Also, because the RISC instruction set allows access to primitive hardware operations, an optimizing compiler is able to effectively reduce the number of instructions performed. Finally, a RISC processor is designed to execute almost all instructions in a single cycle. Caches and software pipeline scheduling help the processor achieve an execution rate of nearly one instruction per cycle. In the future, processor performance improvements will continue to result from improving one or more of these factors. The choice among the various techniques to accomplish this is determined by cost and performance requirements of the intended processor application. For example, multi-processing, by providing more than one processor to execute instructions, can reduce by large factors the average number of cycles required for an application, but requires an application that can be decomposed into independent tasks and incurs the cost of multiple processor units.[1]

II. HISTORY

Seymour Cray’s CDC 6600 from 1965 is often mentioned as the first superscalar design. The Intel i960CA (1988) and the AMD 29000-series 29050 (1990) microprocessors were the first commercial single-chip superscalar microprocessors. RISC CPUs like these brought the superscalar concept to microcomputers because the RISC design results in a simple core, allowing straightforward instruction dispatch and the inclusion of multiple functional units (such as ALUs) on a single CPU in the constrained design rules of the time. This was the reason that RISC designs were faster than CISC designs through the 1980s and into the 1990s.[5]

Except for CPUs used in low-power applications, embedded systems, and battery-powered devices, essentially all general-purpose CPUs developed since about 1998 are superscalar. The P5Pentium was the first superscalar x86 processor; the Nx586, P6Pentium Pro and AMD K5 were among the first designs which decode x86-instructions asynchronously into dynamic microcode-like micro-op sequences prior to actual execution on a superscalar micro architecture; this opened up for dynamic scheduling of buffered partial instructions and enabled more parallelism to be extracted compared to the more rigid methods used in the simpler P5Pentium; it also simplified speculative execution and allowed higher clock frequencies compared to designs such as the advanced Cyrix 6x86.[5]

III. SUPERSCALAR PIPELINE DESIGN

The structure of the superscalar pipelines, the data dependence problem, the factor causing pipelining stalling, and multi issue instruction mechanisms for achieving parallel pipelining operations. For a superscalar machine of degree m, m instructions are issued per cycle and the instruction level parallelism should be m in order to fully utilize the pipeline.[1]
Hardware instruction-scheduling—both with single-instruction issue and multiple-instruction issue—has been the object of a number of previous investigations. This describes fundamental concepts related to hardware instruction-scheduling, and explores how these concepts have been applied in published research investigations. These investigations form the basis of the current research, either by providing ideas to explore or by indicating fruitless approaches. However, these investigations also leave open a number of questions that are addressed in the current study. Previous studies do not address the effects of superscalar techniques on general-purpose applications, focusing instead on scientific applications. In addition, they do not address the effects of super-scalar techniques in the context of the compiler optimizations and the low operations latencies that characterize a RISC processor.\[1\]

Generating and executing an instruction schedule does not intrinsically depend on the number of instructions that can be performed in a single cycle. The capability to perform more than one instruction per cycle simply makes it possible to more effectively use the available resources than if instruction issue is limited to one instruction per cycle. Whether or not this capability is beneficial depends on scheduling successes of software and hardware. Most of the published work on instruction schedulers concentrates on specific scheduling algorithms which can be implemented by software or hardware. Software-based studies usually assume minimal processor hardware in a system environment that is constrained to be deterministic and thus permit software scheduling.\[1\]

**Superscalar pipeline structure:** In an m issue superscalar processor, the instruction decoding and execution resources are increased to form essentially m pipelines operating concurrently. At some pipeline stages, the functional units may be shared by multiple pipelines. One way to view an application is that it specifies a set of operations to be performed by processing resources. The efficient execution of these operations is largely a matter of scheduling the use of processor and system resources so that overall execution time is nearly minimum. Processor software and hardware are responsible for generating a schedule that efficiently uses the available resources, and for translating this schedule into actual operations. A software scheduler can arrange the lexical order of instructions so that they are executed in some optimal (or near-optimal) order with respect to efficient use of resources. A hardware scheduler can dynamically arrange the instruction-execution sequence to make efficient use of resources. In either case, however, the schedule of operations is constrained by data dependencies between instructions and by finite processing resources.\[1\]

---

**Scheduling Constraints[1]:** Regardless of the sophistication of the scheduling policy, performance is ultimately limited by other constraints on scheduling. These constraints fall into four basic categories:

* **Procedural dependencies:** If all instructions to be scheduled were known at the beginning of execution, very high speedups would be possible.\[4\] Unfortunately, branches cause ambiguity in the set of instructions to be scheduled. Instructions following a branch have a procedural dependency on the branch instruction, and cannot be completely executed until the branch is executed.

* **Resource conflicts:** Instructions that use the same shared resources cannot be executed simultaneously.

* **True data dependencies** (or **true dependencies**): An instruction cannot be executed until all required operands are available.

* **Storage conflicts:** Storage locations (registers or memory locations) are reused so that, at different points in time, they hold...
different values for different computations. This can cause computations to interfere with one another even though the instructions are otherwise independent.[3] The term “storage conflict” is not in widespread use, but is more descriptive of the scheduling constraint than other terms in general use. There are two types of storage conflicts: anti-dependencies and output dependencies (there is little standardization on the terminology used to denote these dependencies, but the concepts are the same in any case). Anti-dependencies enforce the restriction that a value in a storage location cannot be overwritten until all prior uses of the value have been satisfied. Output dependencies enforce the restriction that the value in a storage location must be the most recent assignment to that location.

IV. PIPELINE STALLING [3]

This is the problem which may seriously lower pipelining utilization. Proper scheduling avoids pipeline utilization. The problem exists in both scalar and superscalar processors. However, it is more serious in superscalar pipelining. When a result becomes available, it is written to the reorder buffer and to any reservation station entry containing a tag for this result (this requires associative memory in the reservation stations).[2] Subsequent instructions continue to fetch the value from the reorder buffer unless the entry is superseded by a new value—until the value is retired by writing it to the register file. Retiring occurs in the order given by sequential execution, preserving the sequential state for interrupts and exceptions. The reorder buffer is implemented as a content-addressable memory. It is accessed using a register number as a key, and returns the latest value written into the register.[3]

This organization performs name mapping and operand access in a single cycle, as does the register file. During instruction decode, the reorder buffer is accessed in parallel with the register file. Then, depending on which one has the most recent value, the desired operand is selected. The operand value—if available—is copied to the reservation station. If the value is not available (because it has not been computed yet), the result tag is copied to the reservation station. This procedure is carried out for each operand required by each decoded instruction.[3]

There are two, independent approaches to increasing performance with hardware instruction scheduling. The first is to remove constraints on the instruction execution sequence by diminishing the relationship between the order in which instructions are executed and the order in which they are fetched. The second is to remove conflicts between instructions by duplicating processor resources. Either approach, not surprisingly, incurs hardware costs [3]

Instruction Scheduling Policies[3]: The simplest method for scheduling instructions is to issue them in their original program order. Instructions flow through the processor pipeline much as they do in a scalar processor: the primary difference is that the super-scalar pipeline can execute more than one instruction per cycle. Still, though the super-scalar processor can support a higher instruction execution rate than the scalar processor, the super-scalar pipeline experiences more operation dependencies and resource conflicts that stall instruction issue and limit concurrency. It illustrates the operation of the super-scalar processor when instructions are issued in-order and complete in-order. In this case, the pipeline is designed to handle a certain number of instructions (Figure 4 shows two instructions), and only this number of instructions can be in execution at once. Instruction results are written back in the same order that the corresponding instructions were fetched, making this a simple organization. Instruction issuing is stalled when there is a conflict for a functional unit (the conflicting instructions are then issued in series) or when a functional unit requires more than one cycle to generate a result. Figure 5 illustrates the operation of the super-scalar processor when instructions are issued in-order and complete out-of-order. In this case, any number of instructions is allowed to be in execution in the pipeline stages of the functional units, up to the total number of pipeline stages. Instructions can complete out-of-order because instruction issuing is not stalled when a functional unit takes more than one cycle to compute a result: a functional unit may complete an earlier instruction after
subsequent instructions have already completed. Instruction issuing is stalled when there is a conflict for a functional unit, when a required functional unit is not available, when an issued instruction depends on a result that is not yet computed, or when the result of an issued instruction might be later overwritten by an older instruction that takes longer to complete. Completing instructions out-of-order permits more concurrency between instructions and generally yields higher performance than completing instructions in-order. However, out-of-order completion requires more hardware than in-order completion:

* **Dependency logic** is more complex with out-of-order completion, because this logic checks data dependencies between decoded instructions and all instructions in all pipeline stages. The dependency logic must also insure that results are written in a correct order. With in-order completion, dependency logic checks data dependencies between decoded instructions and the few instructions in execution (for the purpose of forwarding data upon instruction completion), and results are naturally written in a correct order.

* **Out-of-order completion** creates a need for functional units to arbitrate for result buses and register-file write ports, because there are probably not enough of these to satisfy all instructions that can complete simultaneously.

Out-of-order completion also complicates restarting the processor after an interrupt or exception, because, by definition, an instruction that completes out-of-order does not modify processor or system state in a sequential order with respect to other instructions. One approach to restart relies on processor hardware to maintain a simple, well-defined restart state that is consistent with the state of a sequentially-executing processor. In this case, restarting after a point of incorrect execution requires only a branch (or similar change of control flow) to the point of the exception, after the cause of the exception has been corrected. A processor providing this form of restart state is said to support
precise interrupts or precise exceptions. Alternatively, the processor pipeline state can be made accessible by software to permit restart. Regardless of whether instructions complete in-order or out-of-order, in-order issue limits performance because there are a limited number of instructions to schedule. The flow of instructions is stalled whenever the issue criteria cannot be met. An alternative is to provide a relatively large set of instructions to be executed in an instruction window from which independent instructions are selected for issue. Instructions can be issued from the window with little regard for their original program order, so this method can issue instructions out-of-order. Figure 6 illustrates the operation of a super-scalar pipeline with out-of-order issue. The instruction window is not an additional pipeline stage, but is shown in Figure 6 as a scheduling mechanism between the decode and execute stages for clarity. The fact that an instruction is in the window implies that the processor has sufficient information about the instruction to make scheduling decisions. The instruction window can be formed by somehow looking ahead at instructions to be executed, or by fetching instructions sequentially into the window and keeping them in the window as long as they cannot be executed.[3]

### Table 1: Comparison of Scalar and Superscalar Pipeline[5]

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Scalar Processor</th>
<th>Super-Scalar Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>fetch one instruction</td>
<td>fetch multiple instructions</td>
</tr>
<tr>
<td>Decode</td>
<td>decode instruction</td>
<td>decode instructions</td>
</tr>
<tr>
<td></td>
<td>access operand from register file</td>
<td>access operands from register file and reorder buffer</td>
</tr>
<tr>
<td></td>
<td>copy operands to functional-unit</td>
<td>copy operands to functional-unit</td>
</tr>
<tr>
<td></td>
<td>input latches</td>
<td>reservation stations</td>
</tr>
<tr>
<td>Execute</td>
<td>execute instruction</td>
<td>execute instructions arbitrate for result busses</td>
</tr>
<tr>
<td>Write-back</td>
<td>write result to register file</td>
<td>write results to reorder buffer</td>
</tr>
<tr>
<td></td>
<td>forward results to functional-unit</td>
<td>forward results to functional-unit</td>
</tr>
<tr>
<td></td>
<td>input latches</td>
<td>reservation stations</td>
</tr>
<tr>
<td>Result Commit</td>
<td>n/a</td>
<td>write results to register file</td>
</tr>
</tbody>
</table>

VI. CONCLUSION

This paper shows that a general-purpose, super-scalar processor needs four major hardware features for best performance: out-of-order execution, register renaming, branch prediction, and a four-instruction decoder. The complexity of these features—although not unmanageable—may argue against the goal of achieving highest possible performance. However, multiple instructions are executed with the help of multiple pipes. The general-purpose benchmark applications used in this study do not lend themselves to software scheduling as readily as many scientific applications, but there are still many ways in which software can help simplify hardware and improve performance. The capability to perform more than one instruction per cycle simply makes it possible to more effectively use the available resources than if instruction issue is limited to one instruction per cycle.

VII. FUTUREWORK

Further study may show that more hardware should be provided to exploit the instruction independence that software is able to provide, or that less hardware is required because software is able to provide benefits that were not anticipated in this study. This study has suggested several areas where software support can improve processor performance or simplify the hardware.

### REFERENCES


### AUTHORS

**Tarun Bhalla** received his B.Tech degree in Computer Science from Punjab Technical University. He is currently working as an Assistant Professor in Anand College of Engineering and Management, Kapurthala.

**Mohit Mittal** received his B.Tech and M.Tech degree in Computer Science from Guru Nanak Dev University, in 2011. He is working as Assistant Professor in Anand college of Engineering and Management, Kapurthala.