Hyperthreaded Processors

Traditionally in a processor, a single thread runs. Let us first examine why it may be desirable for multiple threads to run on a processor. When a processor runs only a single thread various functional units remain heavily underutilized. The reason is dependencies across instructions. Now almost every commercial processor is superscalar and superpipelined. In a superscalar processor, the compiler often does not find enough number of instructions to issue every cycle. This further aggravates the utilization of the functional units. If multiple threads are allowed to execute in a processor, the utilization of various functional units can be significantly increased. Further, as the threads share most of the resources among themselves, the required hardware support for running multiple threads is minimal.

Analysis of Idle Slots in a Superscalar Superpipelined Processor

There are two types of idle or waste slots in a superscalar processor: horizontal waste and vertical waste. Horizontal waste occurs when the compiler cannot find enough number of mutually independent

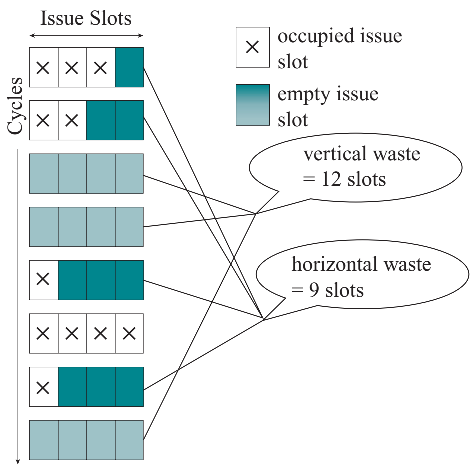

FIG. 1 A schematic representation of vertical and horizontal wastes in a superscalar processor.

instructions to issue during a cycle. Vertical waste occurs when during a cycle no instructions can be issued out, possibly because the running thread has encountered a short stall such as an L1 cache miss. Note that for long stalls such as L2 cache miss, the running thread is preempted. A schematic representation of the vertical and horizontal wastes is shown in Fig. 1.

Multithreading in a Uniprocessor

We can consider multithreading in a uniprocessor as a technique to avoid both horizontal and vertical wastes in a superscalar processor. Thus, multithreading in a uniprocessor helps to increase the utilization of various functional units considerably. This in turn shows up as an increased performance.

As we have already pointed out, threads share many of the resources among themselves. Consequently, the hardware overhead needed for supporting multiple threads to run on a uniprocessor is rather minimal. Let us now examine the extra hardware needed to support execution of multiple threads on a uniprocessor.

One set of the following is needed for each thread:

- Program counter.
- Register file (and flags).
- Per thread renaming table.

Three types of thread scheduling in a uniprocessor:

- Coarse-grained multithreading.
- Fine-grained multithreading.
- Simultaneous multithreading (SMT) or hyperthreading.

In coarse-grained multithreading, a selected thread continues to run and a thread switch occurs only when an active thread undergoes long stall (L2 cache miss, etc.). This form of multithreading only hides long latency events.

In fine-grained multithreading, few threads at any time are designated as active threads. Context switch among the active threads occurs on every clock cycle. Instructions are issued only from a single thread in a cycle. The horizontal wastes may still continue to occur in the fine-grained mutithreading, but vertical wastes are eliminated. In this scheme, both L1 and L2 cache misses can be effectively hidden, and wastes do not occur on these counts.

In simultaneous multithreading or hyperthreading, instructions are issued from multiple threads in the same cycle. Therefore, there is a high probability of finding enough number of instructions during every issue cycle. As a result, hyperthreading eliminates both horizontal and vertical wastes and would result in the best performance among the three schemes of multithreading in the uniprocessor that we discussed. As a result, hyperthreading is popularly being used among the contemporary processors.

Multicore Processors

The performance of computers kept increasing according to Moore's law—first through improvements in manufacturing technologies and subsequently by significant revolutions in processor architecture. However, as the clock rates of processors became faster and faster through the use of deeper pipelines, the switching power dissipation increased significantly. In addition, with improvements in VLSI manufacturing technologies caused the feature size to decrease rapidly. As feature size reduced, the electrons could tunnel across a reverse biased gate and this gave rise to leakage current. The leakage current flows even when a transistor is off. The leakage current soon overtook the switching power as the major producer of heat. The chips dissipated enormous power. For example, the Intel Pentium 5 processor dissipated about 100 W of power even when doing nothing (idle power). The peak power dissipation exceeded 200 W. Compare this with a tea maker that dissipates about 500 W. To make the matters worse, a tea maker dissipates the power over 10s of square inch, whereas the processors dissipate the power over the chip area which is a fraction of a square inch. This made the dissipated power density to become unacceptably high. Every processor did have a heat sink and also a cooling fan mounted on the heat sink.

But, around the year 2000 as the clock speed approached 4 GHz the chips literally melted at the peak operations. This appeared as the road block of the conventional way of increasing the processor cycle time by using deeper pipelines and also the feature size had shrunk to about 50 nm. With any further increase in clock rate or any improvements to the manufacturing technology (smaller feature size) was sure to melt the chip.

Advent of Multicore Processors

Achieving any further performance improvements from the traditional single core processor through improvements to the clock rate or manufacturing technology looked remote as the major bottleneck was unacceptably high power dissipation. This made processor architects to turn to multicore processors. In a multicore processor, each processor runs at moderate clock rate; however, the combined computing power of the multiple processor cores could beat the single core performance. In addition, any core that is idling can be switched off. As presented in Table 1, a quad core processor yields more than 3 times the performance of a single core processor performance while maintaining the peak power dissipation almost the same. The average power dissipation of a multicore processor is much smaller, as the idling cores can be switched off.

TABLE 1 Comparison of multicore performance.

	Single Core	Dual Core	Quad Core
Core area	A	~A/2	~A/4
Core power	W	~W/2	~W/4
Chip power	W + O	W + O’	W + O”
Core performance	P	0.9P	0.8P
Chip performance	P	1.8P	3.2P

Cache Organization

In a multicore processor, the L1 cache is always private to the processor. This is so, because the L1 cache access time needs to be as small as possible and a shared L1 cache would require use of multi-ported cache, thereby complicating its design. The L2 cache as shown in Fig. 1 is shared among the different cores. The multicore processors many times incorporate an L3 cache, to match the increased processing speed of multicore processors with the memory system.

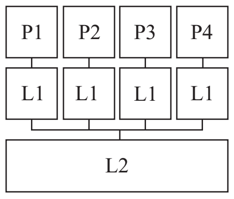

FIGURE 1 Cache organization of a multicore processor.

Multicore Performance Insights

Even when application is single threaded, it can show performance improvements when run on a multicore processor because interrupts and other system events can be handled by an idle processor, instead of preempting a running thread. However, for traditional applications the number of concurrent threads is rather limited, and these may not experience performance benefits when run on a processor with a large number of cores. However, for certain inherently parallel applications (graphic operations, servers, compilers, distributed computing), the speedup can be proportional to the number of processors. At present automatic thread creation by the compiler for an application program has already become a reality.

Parallel Computers

Contents

Parallelism in Programs and Parallel Computers

In this section, we discuss the different types of parallelism that can exist in a program and the types of parallel computers that can be developed. We also point out the types of program parallelism that can be exploited by the different parallel computers.

Parallelism in Programs

Parallelism in programs can be exploited at any of the following three levels.

Instruction-Level Parallelism: This is the parallelism existing within a single thread of program. One way to exploit instruction-level parallelism is through pipelined execution. The different instructions in a single thread are executed in a time-overlapped manner in an instruction pipeline. Instruction-level parallelism is also termed as a fine-grained parallelism, because the unit of parallel execution is the individual instructions. Exploitation of instruction-level parallelism through pipelined execution can be done transparently by hardware and no special efforts on the part of the programmer is necessary. The other ways in which instruction-level parallelism can be exploited is through use of SIMD computers (discussed later in this section) such as vector and array processors.

Thread-Level Parallelism: Different threads in a process can be executed concurrently. For example, multiprocessors can execute multiple threads of a program concurrently. Thread-level parallelism (TLP) is a medium-grained parallelism, because the unit of parallel execution is the threads (sequence of several instructions) in a process. For exploiting thread-level parallelism, the programmer has to identify the threads that can be executed in parallel through calls to some thread libraries.

TLP is inherent in almost every server application. Consider a web server. Every http request from a client (browser) leads to the server creating a thread. By running many threads at once, it becomes possible to tolerate the high amounts of I/O and memory system latency. While one thread is delayed waiting for a memory or disk access, other threads can proceed with their computation without causing a context switch.

Process-Level Parallelism: Different processes in a program can be executed parallelly in a multicomputer or multiprocessor. Process-level parallelism is also known as coarse-grained parallelism, because the unit of parallelism is the different processes in a program, with each process consisting of several threads and each thread of several instructions.

Parallel Computers

Parallel computation is an important and promising way to speedup computation. There are several ways to achieve parallel computation. In the past, a large variety of parallel computers have been proposed and many of those were commercialized. Vector computers were one of the earliest parallel computer and were used for scientific applications such as modeling and simulation. Later multiprocessors became popular, where each processor was manufactured on a processor board. The processor boards were attached to the backplane bus. Besides, several special purpose architectures such as massively parallel computers and systolic computers were proposed. Of late, the multicore computers have become popular. All these different kinds of parallel computers are distinguished by the kind of interconnection between processors (known as "processing elements" or PEs) and interconnection with the memory.

Flynn's taxonomy is a very popular way of classifying computers. Flynn's classification method is based on the number of instruction and data stream that the computer uses to process information. The four classes of computers proposed by Flynn are the following:

- **SISD** (single instruction and single data stream): This is the traditional model of sequential computation where one stream of instructions are executed one by one on a stream of data retrieved from memory.
- **SIMD** (single instruction and multiple data streams): In this case, multiple processors synchronously execute the same instruction, but on different data items retrieved from their own local memory.
- **MIMD** (multiple instruction and multiple data stream): In this model, several different instructions can be under execution at the same time. Each instruction can have its own data. Most commercial computers are of this type. This class of parallel computers includes the pipelined processors, multiprocessors, and multicomputers.
- **MISD** (multiple instruction and single data stream): This model of computation is not a commercially popular. Systolic arrays are an example of MISD computers.

Another way to classify parallel computers is into data parallel and function parallel computers. In a data parallel computer, the computer executes the same instruction on multiple data items. Examples of data parallel computers are: vector computers and systolic architectures. This classification is shown in Fig. 1.

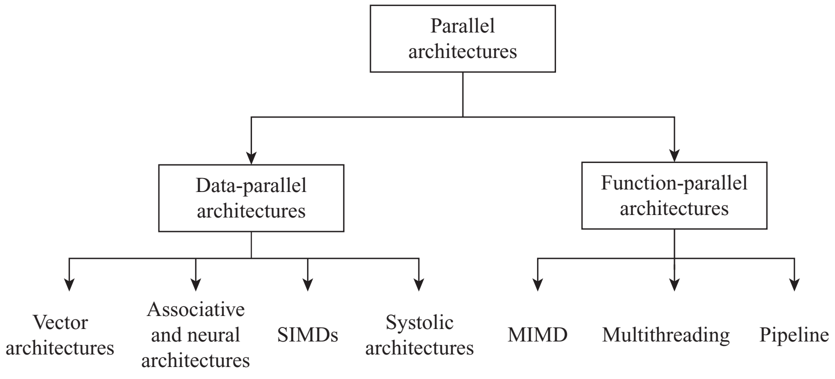

FIGURE 1 Data and function parallel computers.

In a function parallel computer, different instructions achieving different functions are executed simutaneously. Examples of function parallel computers are pipeline, multithreading, and all MIMD computers.

Instruction Pipeline

Contents

Pipelining is a technique that we unconsciously deploy in many everyday applications to improve the rate at which work can get done. Consider a college that imparts engineering education. One way the college can function is once a batch of students get admitted to the college the college does not admit any further batches of students until the last admitted students complete their 4 years of education. The other way in which the engineering college can operate and is normally followed almost by every college is after an admitted batch of students gets promoted to the second year, a fresh batch is admitted. Thus, at any point in time four batches of students are studying in the college; in different classes (years) though. This way, the (average) number of candidates that graduate per year from the college increases fourfold!

A similar technique is deployed by modern processors to improve their instruction processing speed or *throughput*. We all know from a first-level course that each instruction (in a traditional non-pipelined processor) is executed by a processor by first fetching the instruction, decoding it, executing it, and storing back the results. An implementation of these involves generating various control signals to direct various components of a CPU to perform the required operations. However, such a traditional implementation is extremely inefficient. The problem is that any component of the CPU works when directed by the control unit, and sits idle until it is asked again to perform later. Pipelining attempts to overcome this inefficiency by trying to make the different units of the processor as busy as possible. Pipelining improves performance significantly in program code execution. This is done by the pipelining technique by decreasing the idle time of the different components inside the CPU. Pipelining does not completely cancel out idle time in a CPU but a significant reduction is made. Processors with pipelining are organized inside (into stages) which can semi-independently work on separate jobs. If a processor deploys instruction pipelining, multiple instructions are overlapped in execution. A computer pipeline is divided into stages. Each stage completes a part of an instruction in parallel. The stages are connected one to the next to form a pipe—instructions enter at one end, progress through the stages, and exit at the other end.

It should be clear that pipelining does not decrease the time for execution of an individual instruction. However, it in effect increases instruction throughput. The throughput of the instruction pipeline is determined by how often an instruction exits the pipeline. Because pipeline stages are hooked together, all the stages must be ready to proceed at the same time.

The time required to move an instruction one step further in a pipeline is called a pipeline cycle.

The length of the pipeline cycle is determined by the time required for the slowest pipe stage. The pipeline designer's goal is to balance the length (i.e., the processing time) of each pipeline stage. If the stages are perfectly balanced, then the execution time per instruction on the pipelined machine is equal to t/n, where t is the execution time per instruction on non-pipelined machine and n is the number of pipe stages. When the stages are perfectly balanced, the speedup from pipelining should be equal to the number of pipe stages. Usually it is very difficult to make the different pipeline stages perfectly balanced. Besides, pipelining itself involves some overhead. The pipeline overhead arises due to the latches used between two successive pipeline stages. The latches introduce additional delay and contribute to reducing the throughput from the ideally achievable throughput.

Now let us understand the different stages of an instruction pipeline. The basic action of any microprocessor as it executes a program can be broken down into a series of four simple steps, which are repeated as it takes instructions in the code stream for execution: Fetch the next instruction from the address stored in the program counter. Store that instruction in the instruction register and decode it, and increment the address in the program counter. Execute the instruction currently in the instruction register. If the instruction is not a branch instruction but an arithmetic instruction, send it to the ALU (Arithmetic and Logic Unit). We can list the above steps as follows:

1. **Fetch:** Fetch an instruction.
2. **Decode:** Decode an instruction and fetch its operands.
3. **Ex or Execute:** Carry out any required arithmetic or logic operations.
4. **WB or Write Back:** Write the results of an instruction from the ALU back into the destination register.

In a modern processor, the four steps above get repeated over and over again until the program is finished executing. These are, in fact, the four stages in a classic RISC (Reduced Instruction Set Computer) pipeline.

An instruction under execution starts in the fetch phase, moves to the decode phase, then to the execute phase, and finally to the write phase. Each phase takes a fixed, but by no means equal amount of time. If a simple example processor takes exactly 1 ns (pipeline cycle = 1 ns) to complete each stage, then in that processor instruction would take 4 ns to complete, but every 1 ns one instruction will be completed and the next unexecuted instruction will be fetched due to pipelined execution.

As in the later modules we often refer to the MIPS (Microprocesor without Interlocked Pipeline Stages) pipeline, let us discuss the stages of the MIPS pipeline. MIPS is a classic RIC pipelined processor was developed by MIPS Technologies. The MIPS pipeline is broken into five stages with a set of flip flops serving as a latch between each stage. The different stages of the MIPS pipeline are

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

Pipeline Hazards

When programmers write assembly code, they make the assumption that each instruction is executed before execution of the subsequent instruction is begun. However, in pipelined execution this assumption can be violated. For example, consider the following code:

$$a = b + c; e = a + 5;$$

In this example code, the second instruction needs data value computed by the first instruction. However, in a pipeline instructions are executed in an overlapped manner, by the time the second instruction fetches its operands, the first instruction might not have produced its results. As a result, the second instruction will get the old value of ~ and the final computation will be incorrect. When overlapped execution of the instructions of a program in a pipeline causes a program to behave incorrectly, the situation is known as a hazard.

When a hazard such as the one discussed above arises, one simple way the hazard can be overcome is by delaying the execution till the first instruction has produced the required result. This simple technique is called *pipeline stalling*. However, stalling can slowdown a pipeline substantially, and the ideal speedup may not be realized. Fortunately, techniques such as data forwarding can significantly reduce the cases where stalling is required. We discuss various techniques being used in modern processors to avoid data hazards in the next module.

We have so far considered just one example of a situation, where a pipeline hazard can arise. However, there exist two other situations, where hazards can arise. We now summarize the different situations where hazard can arise during pipelined execution of a program.

Data Hazards: When several instructions are under partial execution, a problem arises. These instructions among themselves read and write the same data item. If an instruction that is later in the execution sequence gets to read the data than a previous instruction could write the new value of the data, then this will lead to incorrect results.

Control Hazards: In order to fetch the "next" instruction, we must know which one is required. If the address of the next instruction is obtained through a regular PC increment, then it can be easily fetched. However, if the present instruction is a conditional branch, then the next instruction to be executed may not be known until the current one is processed and the branch out come and the address of the instruction to which branching will occur are known.

Structural Hazards: A structural hazard can arise when two instructions taken up for overlapped execution in a pipeline, require the same component at the same time. Consider the example of two instructions, one of which is being fetched and the other instruction is writing some data into the memory. Both these instructions need to access the cache at the same time, the first one to fetch the instruction from the cache and the second one to write data into cache. However, the cache can process only one request at a time, a structural hazard would arise. A satisfactory solution to structural hazards would be to provide sufficient number of components that may be required to handle concurrent requests.

Inefficiencies in Instruction Pipelining and Overview of Resolution Strategies

Several difficulties make pipeline operations inefficient and prevent achieving the ideal speedup. Many of these causes of pipeline inefficiencies can be prevented by incorporating innovative design solutions discussed in the next module. However, these solutions prevent a pipeline from being as simple as our pipeline descriptions, have so far as suggested. Let us now point out the principal causes for pipeline inefficiencies:

Balancing the Stages: Not all stages take the same amount of time. Imbalance among the pipe stages reduces performance as the clock can run no faster than the time needed for the slowest pipeline stage. This means that the clock speed of a pipeline is determined by its slowest stage. Therefore, in the design of a pipeline processor, it is important to make the stage of the pipeline as balanced as possible.

Hazards: We have already seen that data, control, or structural hazards can arise during the execution of a program and these can substantially slowdown a pipeline, if pipeline stalling is used to overcome the hazards.

Interrupts: Remember from a first course on computer architecture that when an interrupt occurs, the instructions that are under execution are completed before the interrupt is processed (can you answer why?). In pipelined execution, this would require already in the pipeline to complete their executions. However, this approach has two main problems: (1) Processing of an interrupt may be unduly delayed, and this may be unacceptable to many types of interrupts. (2) Pipeline is effectively emptied and filling the pipeline again would take time, and this would degrade the pipeline performance.

Events: Certain events such as cache miss, exceptions, page faults, etc. can necessitate stalling the pipeline. For example, a cache miss stalls all the instructions on pipeline both before and after the instruction causing the miss.

Pipeline Overheads: Pipeline overheads arise from the combination of pipeline latch delays (setup time plus propagation delay) and clock skew. The pipeline registers introduce delays due to their finite setup time requirements. Clock skew is the maximum difference between the arrival of the clock signals at two registers in the pipeline. Once the clock cycle is as small as the sum of the clock skew and the latch overhead, no further pipelining is useful.

Summary

- The Instruction Set Architecture (ISA) refers to that part of the processor that is visible to the programmer or a compiler writer. On the other hand, organization of a processor deals with the internal design and structure of a processor.
- The RISC style helps in saving chip area to provide more number of registers. This helps to reduce memory traffic. In addition, the RISC style helps in implementing an instruction pipeline by making the instructions to have similar sizes and execution times.
- Parallel execution is a promising way to improve execution speed of a program. The three main types of parallelisms that can be exploited during the execution of a program are instruction level (intra-thread), thread level (inter-thread), and process-level parallelism.
- Two popular classes of parallel computers that are commercially being used are SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instructions, Multiple Data) computers.
- Pipelined and array processors can exploit instruction-level parallelism in a program; whereas multiprocessors can exploit thread-level parallelism. Process-level parallelism can be exploited by multiprocessors and multicomputers.

- Pipelining is possibly the most popular way of exploiting instruction-level parallelism in a program, because it has the advantage of being transparent to programmers and at the same time providing reasonable speedups. However, there can be several causes of pipeline inefficiencies such as hazard, which can cause the speedups to drop substantially.

REFERENCES

1. John L. Henessy and Davis A. Patterson, "Computer Architecture: A Quantitative Approach," 3rd Edition, Morgan Kaufmann, 2003.
2. John Paul Shen and Mikko Lipasti, "Modern Processor Design," Tata McGraw-Hill, 2005.
3. D. Sima, T. Fountain, P. Kacsuk, "Advanced Computer Architectures: A Design Space Approach," Addison-Wesley, 1998.
4. www.bapco.com

Exercises

1. State whether the following statements are TRUE or FALSE. Give reasons for your answer.

   a. Even when a high-level program (used for determining the MIPS rating) takes equal time to execute on a RISC machine and a CISC (Complex Instruction Set Computer) machine, the RISC machine would be determined to have a higher MIPS rating than the CISC machine.
   b. In a MIPS five-stage pipeline, every type of MIPS instruction individually takes 5 pipeline cycles to complete its execution.
   c. Data hazards in a pipelined instruction execution occurs only when two instructions exist such that one writes to a memory location and the other reads from the same location and their relative order gets changed in pipelined execution.
   d. Assume that a large C program takes identical times to execute on a RISC machine and a CISC machine. Then, it can be asserted that the RISC machine and the CISC machine have the same MIPS rating.
   e. Suppose a 10-stage pipelined processor is able to achieve a CPI of 1, then there could be no further benefit of deepening the pipeline.
   f. If an instruction *j* is data dependent on some instruction *i*, where instruction *i* precedes *j* in program order, then unless suitable hazard resolution techniques are deployed during execution the said dependency would certainly show up as a data hazard in any pipelined processor.
   g. An instruction pipeline that delays execution of an instruction until all its operands become available would be free from hazards.
   h. When a program is executed on a statically scheduled deeper instruction pipeline, a typical program execution would result in more number of data and control hazards than when executed on a shallower processor.

   - **i.** If a processor deploys five-stage (ideal) pipeline, and has a clock rate that equals that of the unpipelined version, the memory system must deliver 5 times the bandwidth of the unpipelined version.
   - **j.** The relative performance of two processors with the same instruction set architecture can be judged from the clock rates used by them.
   - **k.** In micropipelining, the pipeline stages consist of only a few logic gates each, and results in thousands of pipeline stages. Such a large number of pipeline stages should yield substantial speedup over an equivalent unpipelined processor.
2. What do you mean by the instruction set architecture of a computer?
3. What is an accumulator architecture? Using two or three sentences explain the advantages of a general purpose register architecture over an accumulator architecture.
4. Explain the important differences between RISC and CISC architectures.
5. Explain why the processor designers during 1970s and 1980s considered it a good idea to have complex set of instruction sets, and very few CPU registers.
6. Name at least two commercial RISC and two commercial CISC processors.

RISC versus CISC

Contents

In the early days of computers, programming was done directly in machine code and assembly languages were used subsequently. Much later high-level programming languages were introduced to help enhance programmer productivity. Even after high-level programming languages were introduced, it became traditional to code at least the core parts of applications in assembly or machine languages from efficiency considerations. It, however, became clear that low-level programming is very labor intensive and time consuming.

To improve programmer productivity, processor designers created more and more complex instructions that mimicked the programming constructs of high-level programming languages.

Processor designers believed that supporting complex instructions in the ISA leads to not only higher programmer productivity, but also several other advantages as well. They believed that considerable performance gain can accrue, if complex instructions are executed directly in hardware, rather being run as a series of instructions from an executable file, as the time required for program loading, instruction fetching, decoding, etc. can be considerably reduced.

Another factor that encouraged them in the quest for supporting more and more complex instructions in hardware was the woefully small primary memories that were being used. For example, a typical system during 1970s had only a few kilobytes of primary storage. In older computers, the sizes of primary memories were not only small, but also terribly slow—they were implemented using magnetic technologies (called core memories). As the available memory in computers were small, every byte of memory was precious. It was therefore advantageous to make the density of information held in computer programs as high as possible. By having dense packing of program instructions, one could decrease the access to this slow resource, as well as reduce transfers to the much slower hard disk. In addition to supporting the case for complex instruction types, this aspect also encouraged exploiting features such as highly encoded instructions, and variable-sized instructions. As a result, instruction packing issues were considered to be much more important than the issue of ease of instruction decoding.

The number of CPU registers was intentionally kept very low. CPUs had few registers for several reasons: compiler writing was at its infancy and even if more registers were to be available, the programs could hardly make use of it efficiently during execution. RAM was expensive. Having a large number of registers would have also increased the size of instructions, as the bits used as register specifiers would have increased, leading to larger instruction sizes, consuming precious RAM space.

Possibly CPU designers had all the above points in mind when they tried to make instructions to do as much work as possible. This led to a situation where one instruction that could do the work of several instructions. For example, a single instruction could load the two operands to be added from memory, add them, and then store the result back to the memory as well. Not only that, many versions of the same instructions were supported—different versions did almost the same thing with minor changes. For example, one version would read two numbers from memory, but store the result in a register. Another version would read one number from memory and the other from a register and store the result to memory.

Another important feature of the CISC (Complex Instruction Set Computer) architectures was the support of a large variety of addressing modes. A general goal of CISC was to provide every possible addressing mode for every instruction, a principle known as "orthogonality." This led to increased CPU complexity. However, hardware complexity was overlooked with the hope that each possible command could be tuned individually, making an instruction execution faster than if the programmer used many simpler commands. This lead to a situation where, the work per instruction varied greatly, and the cycles per instruction (CPI) for different instructions varied from 1 to 20 or even larger.

An example of a CISC processor is the VAX processor designed by Digital Electronics Corporation (DEC). The VAX was a minicomputer whose initial implementation was so big and complex that it required three racks of equipment for a single CPU. It supported orthogonal addressing. There were a large number of memory addressing modes, and the every addressing mode was available for every instruction. Other examples of CISC processors are IBM 360, Motorola 86000, and Intel 386.

What Are RISC Processors?

RISC (or Reduced Instruction Set Computer) architectures advocate providing a much smaller number of instruction types as compared with CISC processors. In other words, an important characteristic feature of RISC architectures is that:

RISC architectures support a small, highly optimized set of instructions, rather than many specialized sets of instructions as the CISC processors do.

This seemingly simple idea brought about a landmark revolution in computer architecture design. The RISC architecture style is now profoundly visible in almost every modern processor.

Let us now investigate some of the notable developments that lead to the emergence of the RISC architectures. As we have already discussed, until the early 1980s, the tendency among computer manufacturers was to build increasingly complex CPUs that had ever-larger sets of instructions, which are nowadays referred to as CISC processors. During early 1980s many researchers suggested to reverse this trend by building CPUs capable of executing only a very limited set of instructions, by eliminating support for the least used instructions. The idea of RISC took shape from the reports that many of the features that were included in CISC CPUs to facilitate programmers to be more productive during coding were rarely being used by the programmers. Consequently, the implementations of these instructions only ate up chip area. In addition, support for the complex instructions and varied addressing modes ate up significant chip area. In the late 1970s, researchers demonstrated that the majority of the "orthogonal" addressing modes were ignored by most programmers. This built up the case for simplifying addressing modes and supporting only a limited number of basic addressing modes.

Additionally, it was noticed that the performance gap between the processor and main memory was increasing rapidly. The clock rate at which the processors operated were almost a hundred times faster than the speed at which the memory operated. This gross disparity between the clock rates (cycle times) of the CPUs and the main memory was widening every year and was seen as a bottleneck in further improving computer system performance. This led to designers to advocate supporting a large number of CPU registers. The registers could store frequently used data and thereby reduce the total number of memory accesses. It became apparent that more registers and larger caches were the only way high operating frequencies of CPUs could be supported. The additional registers and cache memories required to reduce memory accesses would take up sizable chip areas that could be made available if the complexity of the CPU was reduced.

The first RISC processors were commercially manufactured in the mid-80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 are some of the early RISC architectures. More recent examples include ARM, DEC Alpha, SPARC, and Power Architecture.

In the following, we summarize some of the important design features that have been characteristic of most RISC processors:

One Cycle Execution Time: RISC processors typically have an average CPI of one; that is, one instruction gets completed after every processor cycle.

Load/Store Architecture: Memory access to retrieve and store data is performed using special load/store instructions. RISC architectures are therefore alternatively called load/store architectures. In these architectures, memory operands are not supported in instructions other than load/store instructions, and only register operands are supported. For this reason, RISC architectures are sometimes also referred to as register–register architectures. As memory access is a relatively very slow process, making these into separate instructions reduces the work disparity among instructions to a great deal and facilitates efficient implementation of instruction pipelines.

Pipelining: Use of instruction pipelining allows overlapped execution of multiple instructions, resulting in increased throughput. We discuss more about instruction pipelines later in this section, and also in the later modules.

Large Number of Registers: RISC designs generally incorporate a larger number of registers to reduce memory traffic. As already discussed, provision of a large number of registers becomes possible due to the elimination of many rarely used instruction types and addressing modes.

Uniform Instruction Encoding: The instructions are usually of constant length and the opcode is always in the same bit position in each instruction. This feature allows faster decoding of instructions.

General Purpose Register (GPR) Set: This feature allows any register to be used in any context, and considerably simplifies generation of efficient executable code by compilers. However, to set things in proper perspective, we must mention that in RISC processors there are almost always separate integer and floating point register files.

Simple Addressing Modes: In RISC architectures, complex and large number of addressing modes are replaced by a set of simple arithmetic instructions. It has been determined through several studies that programmers rarely use very complex addressing modes. Large number of addressing modes only lead to processor hardware complexity, and eat up chip area.

Few Data Types Supported in Hardware: RISC processors support only the very basic data types. CISC machines, for example, supported instructions that operated on byte strings. Some CISC machines supported data type to carry out operation involving polynomials and complex numbers. However, these data types were used only in very small number of programs.

Separate Instruction and Data Caches: RISC designs usually deploy the Harvard architecture (discussed in the following subsection), where the instruction stream and the data stream are conceptually separated through separate instruction and data caches. A major advantage of separate instruction and data caches is that it allows both caches to be accessed simultaneously, which can improve performance dramatically. In a pipelined implementation, it helps the pipeline to operate efficiently by eliminating structural hazard during memory access. We discuss about pipelining and the various types of hazards subsequently.

Hardwired Control: The CPU control is achieved through hardwire circuits, compared with the microprogrammed control for CISC architectures. It is very difficult to deploy hardwired control for CISC architectures because of the large number of instructions and addressing modes lead to very complex hardwire circuitry that can occupy almost the entire available silicon area. Hardwired control helps to improve the performance of the processor and makes programs run faster.

We summarize the difference between RISC and CISC architectural styles in the following table:

CISC	RISC
(Examples: VAX 8600, IBM 390)	(Examples: SPARC, PowerPC, IBM RS/6000)
Single machine instruction for most high-level program instructions	Simple instruction set
Relatively few general purpose registers (GPRs) (8 for Intel 386 and 6 for VAX)	Large number of GPRs (32 for MIPS and 500 for SPARC)
Shared cache for data and instructions.	Separate caches for data and instructions.
Large variety of addressing modes.	Limited number of addressing modes.
Instructions are of different lengths, making it difficult to decode.	Fixed length instructions, easier to decode.
Register-memory architecture	Register-register architecture (also known as load–store architecture)
Microprogrammed control	Hardwired control
CPI of instructions varies widely, for example, CPI can vary between 1 and 20	CPI is usually 1 or 2

Although RISC architecture has several advantages, it is not without any drawbacks. As a series of instructions is needed to complete tasks that could be completed by single instructions in CISC processors, the program sizes for RISC processors are much higher, and also consequently the instruction memory traffic is much higher. On the other hand, possibly the two most remarkable feature of RISC architectures is that it reduces data memory traffic and also helps in efficient implementation of an instruction pipeline.

Von Neumann versus Harvard Architectures

CISC computers are based on the von Neumann architecture. Von Neumann architecture was introduced by von Neumann when he introduced the concept of stored program computers. In a stored program computer, the instructions are also stored in memory just like data and are fetched by the processor in a fetch-execute cycle. Before the von Neumann computation, only data was stored in memory and every instruction was entered by the program through switch settings and executed. The von Neumann architecture is shown in Fig. 1(a). Observe that both the instruction and data are accessed by the processor from the memory over the same bus. In contrast, in the Harvard architecture the processor accessed data and instructions using separate buses (see Fig. 1(b)). This causes faster access to data and memory due to parallel access. In addition, this architecture assumes special significance in pipelined processors, because a von Neumann architecture suffers from structural hazard which considerably slows down the processor.

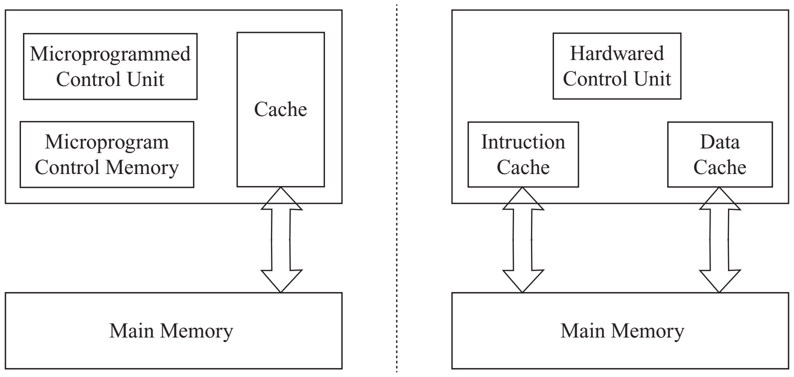

FIGURE 1 RISC versus CISC organization. (a) CISC organization (von Neumann) and (b) RISC organization (Harvard).

A Brief History of Evolution of RISC Processors

Important contributions to the RISC revolution came from both academic researchers as well as innovations and adaptations at industry. The academic research contributions came predominantly from University of California at Berkeley and MIT. The Berkeley RISC project delivered the RISC-I processor in 1982. It used only half as much transistors compared with similar CISC processors. RISC-I had only 32 instructions, compared with several hundreds for CISC processors. Still the RISC processors outperformed the contemporary CISC processors. This proved the effectiveness of the RISC architecture.

Sun Microsystems develop the SPARC processor based on the ideas of the RISC architecture, and these important ideas were used later by almost every other processor. It was Sun's use of an RISC architectures in its new machines that commercially demonstrated the potential benefits of the RISC architecture, and Sun machines quickly outpaced the competition and essentially dominated the entire workstation market.

At about the same time as Berkeley RISC project, John L. Hennessey started MIPS project at Stanford University in 1981. MIPS was the acronym of Microprocessor without Interlocked Pipeline

Stages. Hennessey observed that in the early pipelines in the RISC processors a source of inefficiency was attributed to pipeline stalls in the presence of hazards. Hennessey postulated that if the hazard conditions are prevented in the first place by smart compilers, then the pipeline need not stall at all. In the MIPS pipeline, the hardware is not stalled due to dependencies. MIPS focused in realizing an efficient instruction pipeline. Although pipelining was already in use in other designs, several features of the MIPS chip made its pipeline faster.

John Hennessey commercialized the MIPS design, by starting a company known as MIPS Computer Systems 1984. Their first design was a second-generation MIPS chip known as the R2000. As hardware manufacturing is a highly capital intensive and requires large capital investments, MIPS Computer Systems came up with a unique business plan. It entered into revenue sharing agreements with the companies that used its design, rather than selling its design outright to any company or manufacturing processor chips by itself. This set a new trend of intellectual property (IP) sharing. This aspect has now taken center stage in the development of system on chips (SoC).

After the release of the Sun SPARC, almost all other vendors quickly joined the RISC revolution, including companies with existing CISC designs. Intel released the i860 and i960 by the late 1980s. Motorola built a new design called the 88000 as an RISC successor to their famed CISC 68000 processor. AMD released their 29000 processor which went on to become a popular RISC design in the early 1990s. Today, RISC CPUs (and microcontrollers) represent the vast majority of all CPUs in use.

Processor Performance Evaluation

Contents

Over the last 50 years, computer performance has increased phenomenally. We can get a feel of the kind of improvements that have occurred to computer technology from the following analogy. If similar improvements could have occurred to commercial aircrafts, these aircrafts should be flying now at 1000 times the supersonic speed, costing just a couple of thousands of rupees and would be of the size of a single chair!

In fact, if we quantitatively analyze the computer performance improvements over the last 50 years, computer performance has been doubling approximately every 2 years. This stupendous growth in computer performance was foreseen by Gorden Moore way back in 1965. Moore's prediction was based on his thinking about expected increase over the years in the number of transistors that can be integrated into a single chip as the chip manufacturing techniques improve. In fact, in the initial years of commercial computer manufacture, computer performance improvements came largely through use of innovations in processor and memory manufacturing technologies. These innovations included VLSI circuits and the subsequent reductions in feature sizes on a regular basis. Feature size characterizes a chip manufacturing process and is the minimum size of a transistor or a wire either in the x or y dimensions. Feature size has decreased from 10 μm in 1971 to about 0.1 μm now. The smaller is the feature size, the more is the number of transistors that can be integrated into a single chip. The improvements in the manufacturing technologies have helped integrate more number of transistors into a chip and also helped them operate faster. Improved chip manufacturing techniques contributed to substantial parts of the speed improvements in the 1970s and early 1980s. However, since mid-1980s, most of the computer performance improvements have by large resulted from architectural and organizational innovations. Initial organizational innovations included techniques to exploit instruction level parallelism (ILP). Innovations to be commercialized more recently include techniques for exploitation of thread and process-level parallelism. Innovations in processor organization and architectural techniques have been supported by matching innovations in cache, memory, bus organizations, and storage techniques.

Amdahl's Law

Amdahl's law expresses a fundamental relation that exists between an improved performance of some part of a computer and the resultant improvement in the overall system performance. In our subsequent discussions, we shall often use this law to analyze the overall performance improvement that can accrue by improving some parts of a system, and shall make use of this law in making cost/benefit analysis.

Amdahl's law indicates the overall speedup that can be achieved by using a faster mode of computation. A few examples of improved modes of computation include enhanced processor speed, memory speed, I/O speed, etc. Another important way to speed up computation is by using parallel computation. Before we examine Amdahl's Law in some detail, let us first examine what is meant by speedup.

Speedup: The speedup to a program's execution can be achieved by deploying some faster mode of computation. Speedup can be defined as the time it takes to execute a program divided by the time it takes to execute the same program without deploying the faster mode of computation. Notationally, we can express speedup as:

$$S = \frac{T0}{T1}$$

where $T0$ is the time it takes to execute the program without speed enhancement, and $T1$ is the execution time with speed enhancement.

We can formally state Amdahl's law as follows:

The performance improvement that can be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

From the statement of the law, we can see that Amdahl's law is concerned with the overall speedup achievable due to an improvement that enhances the performance of only a fraction p of the overall computation by a factor of s. Amdahl's law can be interpreted to mean that the overall speedup of applying the improvement will be restricted to:

$$\text{Overall speed up} = \frac{1}{(1-p)+\frac{p}{s}}$$

For example, if an improvement to some component of the computer can speedup 30% of the computation and make it run twice as fast, then p will be 0.3 and s will be 2. Therefore, the maximum achievable speedup due to the improvement would be limited to

$$\frac{1}{(1-0.3)+\frac{0.3}{2}} = \frac{1}{0.7+0.15} = 1.18$$

In essence Amdahl's law makes a case for making the common case faster. One could speedup some part of a computer such as the disk storage system a hundred-fold or more; yet if the improvement affects only 10% of a computation, then the overall speedup could show up as only $\frac{1}{1-0.1} = 1.111$ times faster computation.

As we have already discussed, faster computation can be achieved through several techniques such as by using a faster memory system, using a faster processor, reducing the hard disk access time, etc. Faster program execution can also be achieved by deploying parallel processing techniques. In the special case of parallelization, Amdahl's law can be interpreted as follows:

If f is the fraction of a computation that is inherently sequential, and $(1 - f)$ is the fraction that can be executed in parallel, then the maximum speedup that can be achieved by using n $(n > 0)$ processors is

$$\frac{1}{f + \frac{1-f}{n}}$$

From the above expression, we can see that as n tends to infinity, the maximum speedup tends to $1/f$. Therefore, the achievable speedup due to parallel execution is restricted by the inherently sequential part of the computation.

As an example, consider that for a certain problem the fraction of a computation that is inherently sequential (f) is only 10%, and the rest 90% is amenable to be executed in parallel by a set of processors. Then, the problem can be sped up by only a maximum of a factor of 10, no matter how many processors are used.

Superscalar and Superpipeline Architectures

Instruction pipeline has been a commercial success. Starting with humble beginning, it has become the dominant architecture at present. There is hardly any processor available in the market that is fully CISC. A *k*-stage scalar pipeline can have *k* instructions concurrently executing in the processor, and can ideally achieve a factor of *k* speedup over a non-pipelined machine. And the most enticing aspect is that a pipelined processor hardly needs any extra hardware to achieve the factor of *k* speedup. Further, programming a pipelined processor is transparent to the programmer and needs no extra effort from the programmer. On the other hand, multiprocessors need significant attention from the programmer to be able to efficiently execute on a multiprocessor.

Considering that an instruction pipeline typically consists of four or five stages, the maximum achievable speedup deploying an instruction pipeline appears to be restricted to four or five. Is there a way to extend the concept of a simple instruction pipeline to achieve still higher speedup? Yes, the two available approaches are superscalar and superpipeline approaches.

Superscalar Architecture

A superscalar processor replicate an instruction pipeline *m* times. The factor *m* is called the degree of parallelism. A four-way superscalar pipeline is shown in Fig. 1. Observe that the additional hardware requirement of an *m*-way superscalar processor is *m* times a pipelined processor.

IF	ID	Ex	Mem	WB
IF	ID	Ex	Mem	WB
IF	ID	Ex	Mem	WB
IF	ID	Ex	Mem	WB

FIGURE 1 A four-way superscalar processor.

An *m*-way superscalar processor initiates execution of *m* instructions every cycle. Ideally an *m*-way *k*-stage superscalar processor would achieve a speedup of $m \times k$ over a non-pipelined processor. However, there are a few severe drags on the performance of a superscalar processor. The first is with respect to identifying *m* independent instructions every cycle to issue. It is the responsibility of the compiler to find *m* independent instructions to issue out every cycle. However, the compiler may not find *m* independent instructions every cycle to issue out.

Superpipelined Processor

A superpipelined processor means a processor deploying a much deeper pipeline than the traditional four or five stages. An *m*-way superscalar processor, splits each pipeline stage into four sub-stages. A two-way superpipelined processor is shown in Fig. 2. Superpipelined processors at present often deploy up to 20 stages (four-way superpipeline). There are some superpipelined processors whose number of stages is in excess of 30. In contrast to an *m*-way superscalar processor, which needs *m* times hardware than a simple pipelined processor, an *m*-way superscalar processor needs almost no extra hardware. However, an *m*-way superscalar and an *m*-way superscalar processor, both achieve a speedup of $m \times k$.

F1	F2	D1	D2	E1	E2	M1	M2	W1	W2

FIGURE 2 A two-way superpipelined processor.

Superscalar and Superpipelined

A processor can be both superscalar and superpipelined. A two-way superpipelined and four-way superscalar processor are shown in Fig. 3. An *m*-way superpipelined and *n*-way superscalar processor deploying a basic *k*-stage pipeline would achieve an ideal speedup of $m \times n \times k$. However, the actual speedup in reality is much less due to dependencies among instructions and the presence of branch instructions.

F1	F2	D1	D2	E1	E2	M1	M2	W1	W2
F1	F2	D1	D2	E1	E2	M1	M2	W1	W2
F1	F2	D1	D2	E1	E2	M1	M2	W1	W2
F1	F2	D1	D2	E1	E2	M1	M2	W1	W2

FIGURE 3 A two-way superpipelined and four-way superscalar processor.