CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

1 Introduction

Modern computer systems built from the most sophisticated microprocessors and extensive memory hierarchies achieve their high performance through a combination of dramatic improvements in technology and advances in computer architecture. Advances in technology have resulted in exponential growth rates in raw speed (i.e., clock frequency) and in the amount of logic (number of transistors) that can be put on a chip. Computer architects have exploited these factors in order to further enhance performance using architectural techniques, which are the main subject of this book.

Microprocessors are over 30 years old: the Intel 4004 was introduced in 1971. The functionality of the 4004 compared to that of the mainframes of that period (for example, the IBM System/370) was minuscule. Today, just over thirty years later, workstations powered by engines such as (in alphabetical order and without specific processor numbers) the AMD Athlon, IBM PowerPC, Intel Pentium, and Sun UltraSPARC can rival or surpass in both performance and functionality the few remaining mainframes and at a much lower cost. Servers and supercomputers are more often than not made up of collections of microprocessor systems.

It would be wrong to assume, though, that the three tenets that computer architects have followed, namely *pipelining, parallelism,* and the *principle of locality,* were discovered with the birth of microprocessors. They were all at the basis of the design of previous (super)computers. The advances in technology made their implementations more practical and spurred further refinements. The microarchitectural techniques that are presented in this book rely on these three tenets to translate the architectural specification – the *instruction set architecture* – into a partition of *static* blocks whose *dynamic* interaction leads to their intended behavior.

In the next few pages, we give an extremely brief overview of the advances in technology that have led to the development of current microprocessors. Then, the remainder of this chapter is devoted to defining performance and means to measure it.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

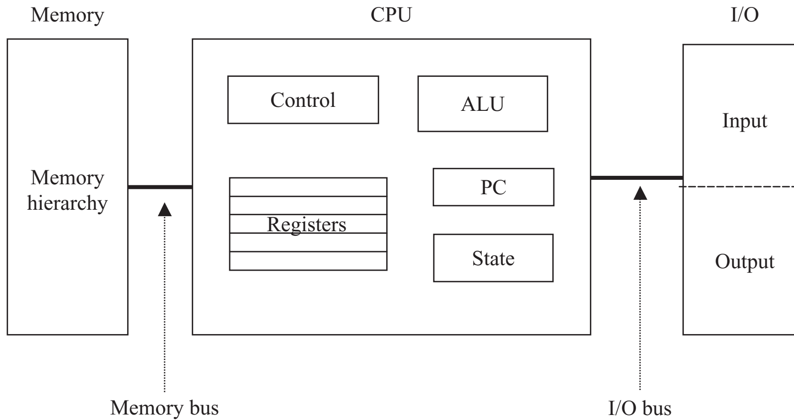

Figure 1.1. The von Neumann machine model.

1.1 A Quick View of Technological Advances

1.1.1 The von Neumann Machine Model

Modern microprocessors have sophisticated engineering features. Nonetheless, they still follow essentially the conventional von Neumann machine model shown in Figure 1.1. The von Neumann model of a stored program computer consists of four blocks:

- A *central processing unit* (CPU) containing an arithmetic–logical unit (ALU) that performs arithmetic and logical operations, registers whose main function is to be used for high-speed storage of operands, a control unit that interprets instructions and causes them to be executed, and a program counter (PC) that indicates the address of the next instruction to be executed (for some authors the ALU and the control unit constitute separate blocks).
- A *memory* that stores instructions, data, and intermediate and final results. Memory is now implemented as a hierarchy.
- An *input* that transmits data and instructions from the outside world to the memory.
- An *output* that transmits final results and messages to the outside world.

Under the von Neumann model, the *instruction execution cycle* proceeds as follows:

1. The next instruction (as pointed to by the PC) is fetched from memory.
2. The control unit decodes the instruction.
3. The instruction is executed. It can be an ALU-based instruction, a load from a memory location to a register, a store from a register to the memory, or a testing condition for a potential branch.
4. The PC is updated (incremented in all cases but a successful branch).
5. Go back to step 1.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

Of course, in the more than sixty years of existence of stored program computers, both the contents of the blocks and the basic instruction execution cycle sequence have been optimized thoroughly. In this book, we shall look primarily at what can be found on a microprocessor chip. At the outset, a microprocessor chip contained the CPU. Over the years, the CPU has been enhanced so that it could be pipelined. Several functional units have replaced the single ALU. Lower levels of the memory hierarchy (caches) have been integrated on the chip. Recently, several microprocessors and their low-level caches coexist on a single chip, forming *chip multiprocessors* (CMPs). Along with these (micro)architectural advances, the strict sequential execution cycle has been extended so that we can have several instructions proceed concurrently in each of the basic steps. In Chapters 2 through 6, we examine the evolution of single processor chips; Chapters 7 and 8 are devoted to multiprocessors.

1.1.2 Technological Advances

The two domains in which technological advances have had the most impact on the performance of microprocessor systems have been the increases in clock frequency and the shrinking of the transistor features leading to higher logic density. In a simplistic way, we can state that increasing clock frequency leads to faster execution, and more logic yields implementation of more functionality on the chip.

Figure 1.2 shows the evolution of the clock frequencies of the Intel microprocessors from the inception of the 4004 in 1971 to the chip multiprocessors of 2003 and beyond. From 1971 to 2002, we see that processor raw speeds increased at an exponential rate. The frequency of the 4004, a speck on the figure that cannot be seen because of the scaling effect, was 1.08 MHz, a factor of 3,000 less than the 3.4 GHz of the Pentium 4. This corresponds roughly to a doubling of the frequency every 2.5 years. To be fair, though, although the Pentium 4 was the fastest processor in 2003, there were many mainframes in 1971 that were faster than the 4004. Some supercomputers in the late 1960s, such as the IBM System 360/91 and Control Data 6600/7600 – two machines that will be mentioned again in this book – had frequencies of over 100 MHz. However, if they were two orders of magnitude faster than the 4004, they were about six orders of magnitude more expensive. After 2003, the frequencies stabilize in the 3 GHz range. We shall see very shortly the reason for such a plateau.

The number of transistors that can be put on a chip has risen at the same pace as the frequency, but without any leveling off. In 1965, Gordon Moore, one of the founders of Intel, predicted that "the number of transistors on a given piece of silicon would double every couple of years." Although one can quibble over the "couple of years" by a few months or so, this prediction has remained essentially true and is now known as *Moore's law*. Figure 1.3 shows the exponential progression of transistors in the Intel microprocessors (notice the log scale). There is a growth factor of over 2,000 between the 2,300 transistors of the Intel 4004 and the 1.7 billion transistors of the 2006 Intel dual-core Itanium (the largest number of transistors in the

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

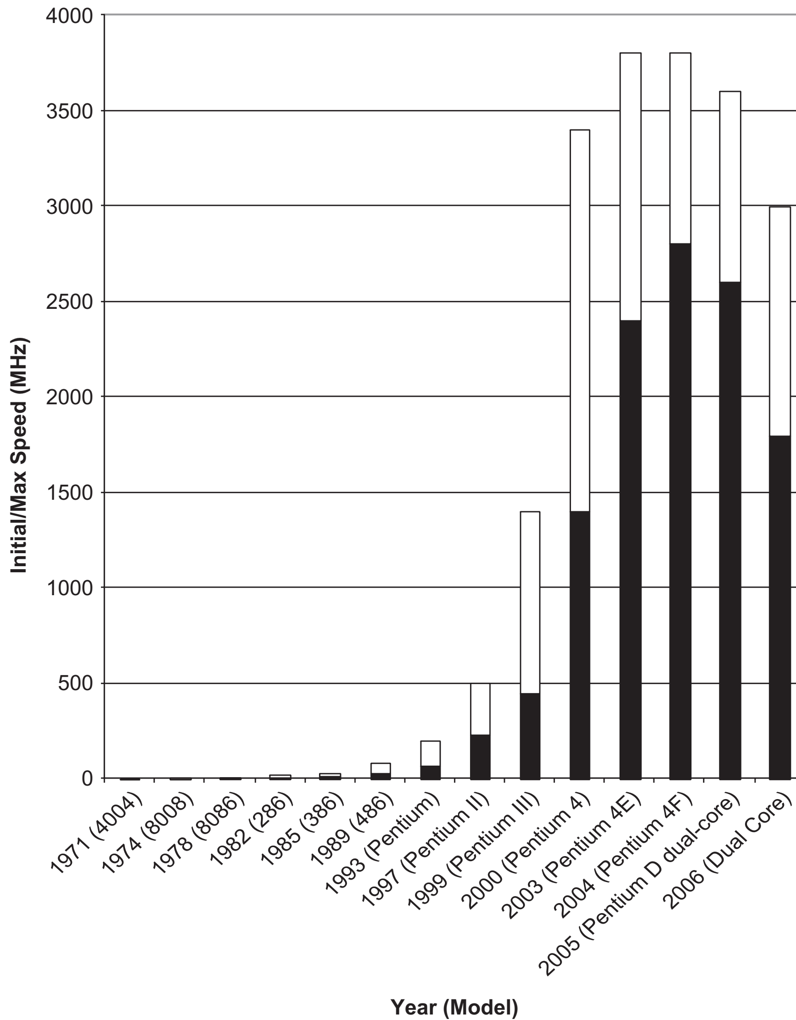

Figure 1.2. Evolution of Intel microprocessors speeds (black bars: frequency at introduction; white bars: peak frequency).

microprocessors of Figure 1.2 is a little over half a billion for a dual-core Extreme). It is widely believed that Moore's law can be sustained until 2025.

Besides allowing implementers to use more logic (i.e., provide more functionality) or more storage (i.e., mitigate the imbalance between processor speed and memory latency), consequences of Moore's law include reduced costs and better reliability.

However, the exponential growths in speed and logic density do not come without challenges. First, of course, the feature size of the manufacturing process, which is closely related to transistor sizes, cannot be reduced indefinitely, and the number of transistors that can be put on a chip is physically bounded. Second, and of primary importance now, is the amount of power that is dissipated. Figure 1.4 illustrates this point vividly (again, notice the log scale). Third, at multigigahertz frequencies, the on-chip distance (i.e., sum of wire lengths) between producer and consumer of

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

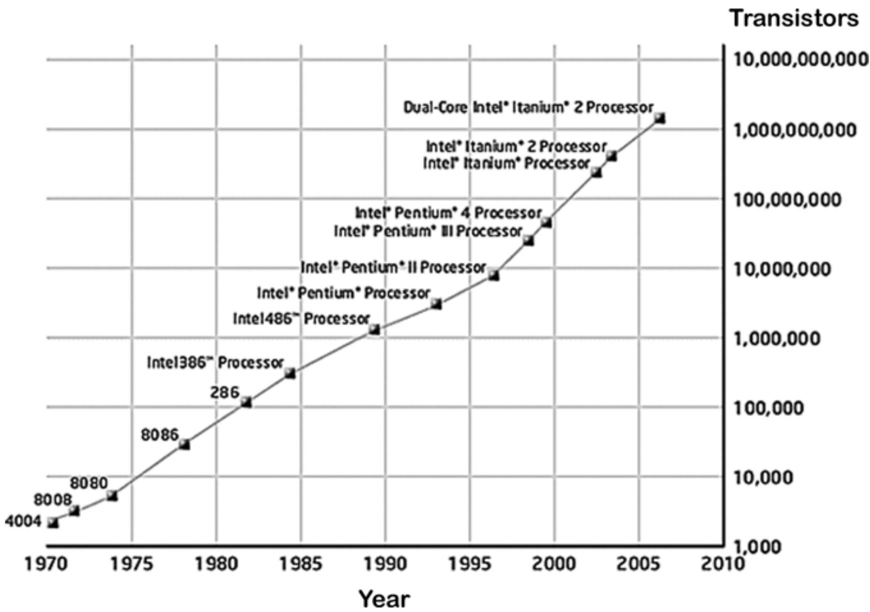

Figure 1.3. Illustration of Moore's law for the Intel microprocessors.

information becomes a factor and influences microarchitectural design decisions. We will elaborate more on these design problems in Chapter 9.

Now we can come back to the source of the leveling of speeds in Figure 1.2 after 2002. The power requirements as shown in Figure 1.4 and the growth in the number of transistors as shown by Moore's law limit the speeds that can be attained, because the switching (or dynamic) power dissipation is related to the frequency, and the

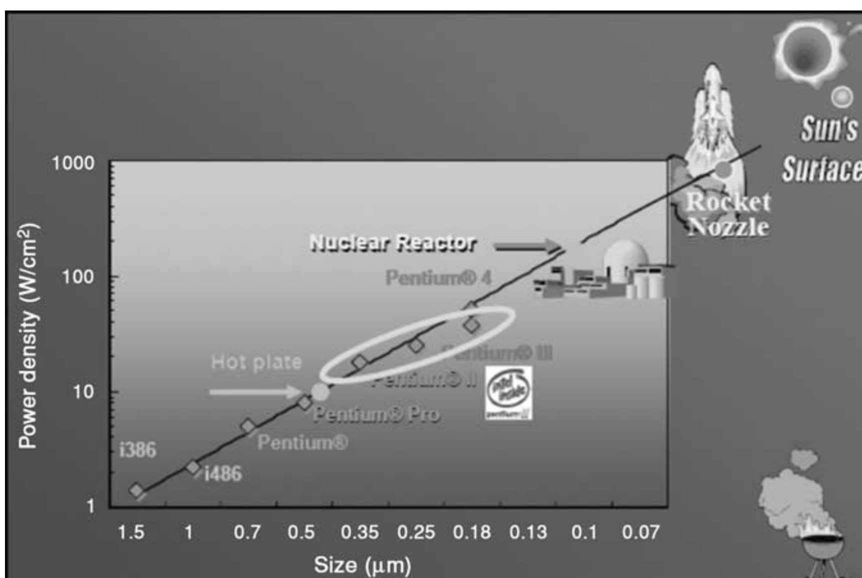

Figure 1.4. Power dissipation in selected Intel microprocessors (from Fred Pollack, Micro32 keynote).

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

static power (leakage) dissipation is related to the number of transistors on the chip. Therefore, Intel and other manufacturers have capped the frequencies at their 2003 level. An exception is the IBM Power6, introduced in 2007, which is clocked at slightly over 4.7 GHz. However, even with this latest development, the exponential speed increase has stopped. The persistence of Moore's law and the resulting increase in the amount of logic that can be put on a chip have led to the emergence of CMPs.

1.2 Performance Metrics

Raw speed and the number of transistors on a chip give a good feel for the progress that has been made. However, in order to assess the performance of computer systems, we need more precise metrics related to the execution of programs.

From a user's viewpoint, what is important is how fast her programs execute. Often, though, this time of execution depends on many factors that have little to do with the processor on which the programs execute. For example, it may depend on the operating system (which version of Windows or Linux), the compiler (how well optimized it is), and network conditions (if the user's workstation is sharing I/O devices with other workstations on a local area network). Moreover, the programs that user A is interested in can be completely different from those of interest to programmer B. Thus, to perform meaningful architectural comparisons between processors and their memory hierarchies or to assess the benefits of some components of the design, we need:

- Metrics that reflect the underlying architecture in a program-independent fashion.
- A suite of programs, or *benchmarks*, that are representative of the intended load of the processor.

Neither of these goals is easy to attain in a scientific manner.

1.2.1 Instructions Per Cycle (*IPC*)

Metrics to assess the microarchitecture of a processor and of its memory hierarchy, (i.e., caches and main memory), should be defined independently of the I/O subsystem. The execution time of a program whose code and data reside in the memory hierarchy will be denoted EX_{CPU} to indicate the context in which it is valid.

The execution time of a program depends on the number of instructions executed and the time to execute each instruction. In a first approximation, if we assume that all instructions take the same time to execute, we have

$$EX_{CPU} = Number\ of\ instructions \times Time\ to\ execute\ one\ instruction \qquad (1)$$

Now, *Time to execute one instruction* can be expressed as a function of the cycle time (the reciprocal of the clock frequency) and the number of cycles to execute an

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

instruction, or *CPI* (for "cycles per instruction"). Equation (1) can then be rewritten as:

$$EX_{CPU} = Number\ of\ instructions \times CPI \times cycle\ time \quad (2)$$

CPI is a program-independent, clock-frequency-independent metric. Recently, computer architects have preferred to quote figures related to its reciprocal, *IPC* (for "instructions per cycle.") The main reason for this cosmetic change is that it is psychologically more appealing to strive for increases in *IPC* than for decreases in *CPI*. Thus, we can rewrite Equation (2) as

$$EX_{CPU} = (Number\ of\ instructions \times cycle\ time)/IPC \quad (2')$$

and define *IPC* as

$$IPC = (Number\ of\ instructions \times cycle\ time)/EX_{CPU} \quad (3)$$

or

$$IPC = Number\ of\ instructions/(clock\ frequency \times EX_{CPU}) \quad (3')$$

or, if we express EX_{CPU} as *Total number of cycles* × *cycle time*,

$$IPC = Number\ of\ instructions/Total\ number\ of\ cycles \quad (3'')$$

Another reason to use *IPC* as a metric is that it represents *throughput*, (i.e., the amount of work per unit of time). This is in contrast to EX_{CPU}, which represents *latency*, which is an amount of time.

In an ideal single-pipeline situation, when there are no hazards and all memory operations hit in a first-level cache, one instruction completes its execution at every cycle. Thus, both *CPI* and *IPC* are equal to one. We can forget about the overhead of filling the pipeline, for this operation takes only a few cycles, a negligible amount compared to the billions – or trillions – of cycles needed to execute a program. However, when there are hazards – for example, branches, load dependencies, or cache misses (we will review these hazards and the basics of the memory hierarchy in Chapter 2) – then *CPI* will increase (it is easier to reason about *CPI* than about *IPC*, although for the reasons mentioned above we shall often refer to *IPC*). More specifically, assuming that the only disruptions to the ideal pipeline are those that we just listed, *CPI* can be rewritten as

$$CPI = 1 + CPI_{load\ latency} + CPI_{branches} + CPI_{cache} \quad (4)$$

where CPI_x refers to the extra number of cycles contributed by cause x. This yields the formula for *IPC*:

$$IPC = \frac{1}{1 + \sum_x CPI_x} \quad (4')$$

EXAMPLE 1: Assume that 15% of instructions are loads and that 20% of the instructions following a load depend on its results and are stalled for 1 cycle. All instructions and all loads hit in their respective first-level caches. Assume

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

further that 20% of instructions are branches, with 60% of them being taken and 40% being not taken. The penalty is 2 cycles if the branch is not taken, and it is 3 cycles if the branch is taken. Then, 1 cycle is lost for 20% of the loads, 2 cycles are lost when a conditional branch is not taken, and 3 cycles are lost for taken branches. This can be written as

$$CPI_{load\ latency} = 0.15 \times 0.2 \times 1 = 0.03$$

and

$$CPI_{branches} = 0.2 \times 0.6 \times 3 + 0.2 \times 0.4 \times 2 = 0.52$$

Thus

$$CPI = 1.55 \quad \text{and} \quad IPC = 0.65.$$

A very simple optimization implementation for branches is to assume that they are not taken. There will be no penalty if indeed the branch is not taken, and there will still be a 3 cycle penalty if it is taken. In this case, we have

$$CPI_{branches} = 0.2 \times 0.6 \times 3 = 0.36, \quad \text{yielding } CPI = 1.39 \quad \text{and} \quad IPC = 0.72$$

Of course, it would have been more profitable to try optimizing the branch-taken case, for it occurs more frequently and has a higher penalty. However, its optimization is more difficult to implement. We shall return to the subject of branch prediction in detail in Chapter 4.

Let us assume now that the hit ratio of loads in the first level cache is 95%. On a miss, the penalty is 20 cycles. Then

$$CPI_{cache} = (1 - 0.95) \times 20 = 1$$

yielding (in the case of the branch-not-taken optimization) $CPI = 2.39$ and $IPC = 0.42$.

We could ask whether it is fair to charge both the potential load dependency and the cache miss penalty for the 5% of loads that result in a cache miss. Even in a very simple implementation, we could make sure that on a cache miss the result is forwarded to the execution unit and to the cache simultaneously. Therefore, a better formulation for the load dependency penalty would be $CPI_{load\ latency} = 0.15 \times 0.2 \times 1 \times 0.95 = 0.029$. The overall *CPI* and *IPC* are not changed significantly (in these types of performance approximation, results that differ by less than 1% should be taken with a large grain of salt). Although this optimization does not bring any tangible improvement, such small benefits can add up quickly when the engines become more complex, as will be seen in forthcoming chapters, and should not be scorned when there is no cost in implementing them. This example also indicates that the memory hierarchy contributions to *CPI* could be dominating. We shall treat techniques to reduce the penalty due to the memory hierarchy in Chapter 6.

There exist other metrics related to execution time and, more loosely, to *IPC*. One of them – *MIPS,* or millions of instructions per second – was quite popular in

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

the 1970s and the 1980s, but has now fallen into desuetude because it is not very representative of the architecture of contemporary microprocessors. *MIPS* is defined as

$$MIPS = \frac{Number\ of\ instructions}{EX_{CPU} \times 10^6} \tag{5}$$

The main problem with this metric is that all instructions are considered to be equal and neither classes of instructions nor explicit decreases in *IPC* (hazards) are taken into account. This can lead to discrepancies such as a system having a higher *MIPS* rating (*MIPS* is a rate inversely proportional to execution time, so the higher, the better) than another while having a larger execution time because, for example, it executes fewer instructions but some of these instructions have much higher latency (see the exercises at the end of the chapter).

MFLOPS, the number of millions of floating-point operations per second, has a definition similar to that of *MIPS*, except that only floating-point operations are counted in *Number of instructions.* It can be argued that it is a representative measure for systems whose main goal is to execute scientific workloads where most of the computation time is spent in tight loops with many floating-point operations. However, it does suffer from some of the same shortcomings as *MIPS*, mainly compiler-dependent and algorithm-dependent optimizations. Reaching *teraflops*, that is, 1,000 megaflops, was for a long time the goal of supercomputer manufacturers; it was achieved in 1996 by a massively parallel computer (i.e., a system composed of thousands of microprocessors). The next Holy Grail is to achieve *pentaflops* (a million megaflops): it is not the subject of this book.

1.2.2 Performance, Speedup, and Efficiency

Equation (2) is *the* rule that governs *performance*, which can be defined as the reciprocal of the execution time. A smaller execution time implies better performance. The three factors that affect performance are therefore:

- The number of instructions executed. The smaller the number of instructions executed, the smaller the execution time, and therefore the better the performance. Note that reducing the number of instructions executed for a given program and a given instruction set architecture (ISA) is a *compiler*-dependent function.
- *CPI* or *IPC*. The smaller the *CPI* (the larger the *IPC*), the better the performance. It is the goal of computer architects to improve *IPC*, (i.e., this is a question of *microarchitecture* design and implementation).
- The clock cycle time or frequency. The smaller the cycle time (the higher the frequency), the better the performance. Improving the cycle time is *technology*-dependent.

Improvement in any of these three factors will improve performance. However, in practice these factors are not completely independent. For example, minimizing the

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-76992-1 - Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors
Jean-Loup Baer
Excerpt
More information

number of instructions executed can be counterproductive. A case in point is that a multiplication can be sometimes replaced by a small sequence of add-and-shift operations. Though more instructions are executed, the time to execute the add–shift sequence may be smaller than that of the multiply. The apparent contradiction is that in Equation (2) the factor *CPI* was assumed to be the same for all instructions. In fact, in the same way as we introduced CPI_x for showing specific penalties due to cause x, we should make sure that *CPI* reflects the mix of instructions and their execution times. If we have c classes of instructions with frequencies $f_1, f_2, \ldots, f_c$ and with cycles per instruction $CPI_1, CPI_2, \ldots, CPI_c$, then Equation (4) should be rewritten as

$$CPI = \sum_{i=1}^{c} f_i \times CPI_i + \sum_x CPI_x \quad (6)$$

or

$$IPC = \frac{1}{\sum_{i=1}^{c} f_i \times CPI_i + \sum_x CPI_x} \quad (6')$$

Similarly, *IPC* and clock cycle time are related. With smaller cycle times, fewer operations can be performed in a single stage of the pipeline, and the latter becomes deeper. Hazards now can inflict larger penalties when these are expressed in terms of number of cycles. Thus, *IPC* might decrease with a smaller cycle time. However, recall from Equation (2′) that what is important is decreasing the ratio *cycle time/IPC*.

IPC is a useful metric when assessing enhancements to a given microarchitecture or memory latency hiding technique. We will often make use of it in subsequent chapters. However, using *IPC* as a basis for comparison between systems that have different ISAs is not as satisfactory, because the choice of ISA affects and is affected by all three components of performance listed above.

When comparing the performance of two systems, what we are interested in is their relative performance, not their absolute ones. Thus, we will say that system A has better (worse) performance than system B if the execution time of A – on some set of programs (cf. Section 1.2.3) – is less (more) than the execution time of B:

$$\frac{Performance_A}{Performance_B} = \frac{EX_{CPU-B}}{EX_{CPU-A}} \quad (7)$$

A particularly interesting ratio of performance is *Speedup*, which is the ratio of performance between an enhanced system and its original implementation:

$$Speedup = \frac{Enhanced\ performance}{Original\ performance} = \frac{EX_{CPU-original}}{EX_{CPU-enhanced}} \quad (8)$$

Historically, the speedup was defined for parallel processors. Let T_i denote the time to execute on i processors ($i = 1, 2, \ldots, n$); then the speedup for n processors is

$$Speedup_n = \frac{T_1}{T_n}$$

© in this web service Cambridge University Press www.cambridge.org