20

RISC Architectures

Manolis Katevenis

This chapter is about processor architecture—it is not about parallel or distributed computer architecture. However, its contents are very relevant to the topic of this book. Three key hardware ingredients are needed to make high-performance parallel and distributed computer systems: (1) high-speed computation, (2) high-speed communication, and (3) high-speed peripheral (I/O) devices. This chapter talks about the first of these three hardware components, and specifically about how to make high-speed processors (at a reasonable cost, too), which are a necessary building block for high-performance parallel and distributed computers. Having reached this chapter, the reader will probably already realize that the overall performance of a parallel or distributed system will get worse if the system has f times more processors, but each processor has f times lower performance ($f > 1$). This is so because every time we introduce more parallelism into a computation, more communication and synchronization overhead is also introduced. Thus, the way to high-performance parallel computers is *first* to choose the highest performance processors at the desired cost level, and then to interconnect as many of them as are needed to achieve the desired overall system performance. This chapter is about the "first step."

Reduced instruction set computer (RISC) architectures are the basis of modern high-performance processors. They have a simplified instruction set, with register-to-register arithmetic instructions; memory can only be accessed by *load* and *store* instructions, usually with a single, simple addressing mode; the instructions have a fixed size and few, simple formats. RISC processors use pipelining, and their instruction set allows tight control of their pipeline by the software. Optimizing compilers perform instruction scheduling so as to exploit the parallelism offered by the pipelined data path. RISC processors achieve high performance at low cost: the smaller and simpler circuits allow higher clock rates; the data path can use pipelining, and the compiler can exploit it. In addition, design time, design cost, and silicon area are reduced.

20.1 What is RISC?

RISC stands for *reduced instruction set computer,* a pun on the word "risk." Scientifically, RISC is a *style* or a *family* of processor architectures that share a certain number of characteristics, which are described and explained in this chapter. RISC is *not* one, single computer architecture. RISC is *not a strictly defined* set of architectures, either; there are many architectures "near the boundaries" of RISC that some people will say they are RISC and other people will say are not RISC.

The word RISC, or better *RISC concept,* has also come to be used in another way. It may refer, depending on the context, to the set of basic design principles that led to the development of these architectures. As technology changes, the particular design choices of RISC architectures may cease to be as good as they were when originally made (in the 1980s). However, the design principles that led to these choices should (hopefully) stay valid for a longer time and guide computer designers in the future on how to make good choices given a certain implementation technology. Also, these principles sometimes have a wider range of applications than just uniprocessor design—or than just hardware design.

The RISC ideas were developed mostly in the early 1980s and became popular in the second half of that decade. RISC architectures came, in part, as a reaction to the direction that computer architecture (except supercomputer architecture) had taken in the 1970s. Today, RISC is considered to be the basis for designing high-performance processors, and almost at any price level. In this introductory section, we will first discuss the basic design principles that led to RISC, and then we will list the most important architectural characteristics of RISC.

20.1.1 Design principles that led to RISC

Use quantitative engineering analysis rather than myths and marketing fads. It sounds quite obvious that the design of a technical device such as a processor should be based on engineering analysis rather than myths. However, it was not so with the state of the art in computer architecture in the 1970s. There was a widespread impression that if a task is executed *in hardware*, (i.e., as one computer instruction), then that task and the whole computer is faster than if this task is executed *in software* (i.e., by multiple instructions). Based on this groundless impression, the marketing strategy for several computers was based on how many tasks the computer could execute as single instructions, thus leading to *complex instruction set compilers* (*CISC*). RISC came as a reaction to this unfounded CISC trend.

Understand trade-offs across levels of abstraction. Computer processors are complex devices; hence, they can be mastered only by using engineering abstractions. Levels of these abstractions are, from bottom to top: integrated circuit (IC) polygon level, transistor level, logic gate level, register transfer level, hardware block diagram level, instruction set level, operating system (OS) and standard library level, programming language level, and application program level. Unfortunately, changing a design choice in one level often has repercussions at all other levels, and the repercussions are usually very difficult to evaluate accurately—even more so if the engineers doing this do not understand (or ignore!) some of these levels of abstraction. In the past, it was thought that, for the "purity" or the longevity of an architecture, this latter ought to be designed independently of the implementation hardware; it was even sometimes considered a bit pejorative for a computer architect to deal with such "low-level details" as anything below hardware block diagrams! Yet, parasitic capacitances and transistor currents are where the clock speed game is eventually won or lost....

Make the common case fast. Overall performance is a weighted mean of performance on individual tasks, where the weight factors are the frequencies with which these individual tasks appear in the overall workload. Thus, speeding up the frequent operations pays back much more than doing so for the infrequent ones. Furthermore, if the acceleration of one operation has even slightly negative side effects on the speed of other operations, then such an acceleration can easily turn out to reduce overall performance if its target operation is one that rarely occurs.

Smaller is faster. In electronic circuits, the shorter a transmission line is, the faster electromagnetic waves will travel through it. When the dominant delay mechanism is not propagation delay through a transmission line, it is the charging or discharging of a parasitic capacitance by a given electric current. Reducing the parasitic capacitance will speed up the circuit. Shorter wires have less parasitic capacitance; also, wires with fewer transistors (fewer gates, simpler logic) connected on them have lower parasitic capacitance. As we see, in all cases, smaller circuits (i.e., simpler circuits and circuits with shorter wires) are faster. It is not surprising that, in processor design, fewer operations, simpler operations, and a more regular set of operations lead to smaller, and thus faster, circuits. The crucial point, of course, is where the golden mean lies; the rest of the chapter discusses this point.

Hardware is not necessarily faster than software. In some cases, a hardware implementation of a certain task is much faster than its software implementation. This is particularly the case when circuits that are specialized for this task are designed and added to the processor; for example, floating-point operations are much faster in hardware than when performed through repeated use of the integer arithmetic unit(s). In other cases, encoding a task as a single instruction is not (or cannot be) accompanied by any such circuit optimization. In many such cases, this single instruction takes as many clock cycles to execute as the multiple, simpler instructions that can execute the same task would; this will be further discussed in Section 20.3. In these cases, there is no "magic" in hardware that makes it any better than software. Providing a reduced instruction set in the processor and synthesizing the complex tasks in software is just as good as providing a complex instruction set—actually, it is better, because of the previous point: *smaller is faster* for those operations that remain in the reduced instruction set.

20.1.2 Architectural characteristics of RISC

As was said earlier, RISC is a style or a family of processor architectures that share some characteristics. Below, we list these characteristics, in order of importance, according to this author's view. Remember that RISC is not a strictly defined set of architectures, so what follows is not a "defining set" of characteristics.

Register orientation. Smaller is faster, as we said earlier, and smaller memories are faster than larger ones. This is the basis of memory hierarchies: register file, cache memory, main memory, disk, tape. The processor registers are the fastest level of this hierarchy, for three reasons: (1) short access time because of smaller size (shorter than level-1 (on-chip) cache memory, although the difference is often relatively small); (2) multiport access, i.e., multiple registers can be accessed in parallel (cache memories rarely have this capability, since it would greatly increase their cost and slightly lengthen their access time); and (3) low addressing overhead, because register specifiers consume only a few bits in the instruction, and addressing is simple and fast—no effective address calculation is involved, and no address translation exists. For these reasons, good exploitation of the registers is of capital importance in speeding up execution. Unfortunately, unlike the cache memory, the register file is not a transparent level of the memory hierarchy. (But this is

also one reason why registers are faster than the cache!) Thus, it takes effort for the compiler to properly exploit the register file. The development of RISC was linked to the development of optimizing compilers that could allocate frequently used program variables in registers. Thus, in RISC architectures, the operands are more frequently in registers, or conversely, once the operands are more frequently in registers, RISC architectures make more sense than CISC. Also, RISCs tend to have more registers than CISCs.

Load/store architecture. Perhaps the most important concrete characteristic of RISC architectures is the fact that memory is only accessed via load (read) and store (write) instructions, while all other instructions operate exclusively on register-resident operands. To put it differently, transferring information between the memory hierarchy and the processor is separated from operating on this information. There are several reasons for this separation of work, and they will become apparent in the rest of this chapter; in brief, they are:

1. Operands are frequently in registers, so the demand for operations on memory operands is reduced.
2. Pipeline organization is easy.
3. Interruptibility and restartability is available for instructions in the pipeline.
4. Instruction size is fixed.
5. There is increased flexibility for instruction scheduling.

Fixed instruction size. The instruction fetch hardware of the processor is greatly simplified when instruction placement in memory follows a simple alignment pattern. Also, control transfer instructions (branch, jump) are sped up when their target instructions do not cross word boundaries. For these reasons, RISC architectures usually have a fixed instruction size—32 bits. In the early 1980s, RISC architectures were heavily criticized because this fixed instruction size requires less compact code. However, it became apparent with time that this drawback is less important than the aforementioned advantages. We will come back to these issues in Section 20.4.

Fixed position of source operands in instruction format. The latency of decoding and executing an instruction is reduced when the descriptors for the source operands (two source registers, usually) are in fixed positions within the instruction, independent of the instruction's opcode. When this property holds, as in RISC architectures, the source operands can be read *in parallel* with decoding of the opcode. Reducing the latency of instruction execution helps in reducing the branch execution time.

Single result per instruction. The pipeline is simplified if each instruction writes one result into one memory word or register and does not modify the user-visible machine state in any other complicated or intricate way. In particular, instruction interruptibility and restartability is made easier in this way. Most RISC architectures follow this; where condition codes exist, setting them is the most notable exception.

Regularity. As we saw, smaller is faster—in particular, simpler logic circuits with fewer gates are faster. The primary method to reduce the number of transistors connected to wires (other than the undesirable method of reducing the processor's capacity as, for example, the number of general-purpose registers) is to make the circuit more regular. Regularity means more similarity among the various operations so that they can all be handled by the same subsystems, and less special cases, thus reducing the number of circuits, buses, and multiplexers needed. RISC architectures strive for regularity in many respects: instructions perform a similar amount of work each, consisting of similar operations (read sources, perform one arithmetic operation, and so forth) on similar operands (two source

registers, one destination register) in a fixed set of pipeline stages. There is usually one single memory addressing mode, and that resembles the rest of the operations.

Provide primitives—not solutions. The instruction set of an architecture is designed once and stays fixed, often for more than a decade. The software that runs on it can be changed at any time. Thus, hardware (the instruction set) should only provide general-purpose building blocks. Special features that match special programming language constructs should be left for the compiler or the libraries; if placed in the hardware, they are either too specialized and thus useless most of the time, or too general and often too slow (see Ref. [1], pp. 121, 124). Hardware instructions should each do one, simple thing well and fast, and should combine flexibly and fast with each other; combining them in intelligent ways should be left to the software.

Make performance visible and optimizable. The old, CISC view was that the hardware should provide an embellished, abstract interface (instruction set) that is appropriate for easy programming in assembly language by a human. The new, RISC view is that the hardware should provide a raw interface with visible performance characteristics, appropriate for use by an optimizing compiler. Accordingly, RISCs make pipeline characteristics visible at the instruction set level. Examples include delayed loads (Section 20.3.2) and delayed branches (Section 20.4.2).

The rest of this chapter is organized as follows. Section 20.2 explains pipelining, which is the basic speed-up technique for processor hardware. Section 20.3 discusses how a pipelined processor can be kept busy with useful work for as much of the time as possible and why RISC is better than CISC in this respect. Section 20.4 examines the issues related to instruction size and format, while Section 20.5 discusses other disadvantages of CISC: control and interrupt complexity, design time and cost, and clock frequency. A brief historical note and a general perspective follow in Section 20.6.

20.2 Pipelining and Bypassing

Pipelining is the principal method to achieve high performance in the operation of a processor. Up until the 1980s, substantial pipelining was used only in supercomputers and the high-end mainframes. Supercomputers often had a simple instruction set; for example, the machines designed by Seymour Cray have used register-oriented, load/store instruction sets. The lower-end computers did not use any appreciable amount of pipelining; these included some of the most CISC architectures, such as the DEC VAX [3]. Perhaps pipelining was considered too costly to be applied to low-end systems, or rather it was found too complicated and costly for the complex instruction sets that had evolved in the meanwhile. RISC emerged when it was realized that pipelining can perfectly well be used in low-cost processors (microprocessors) provided that the instruction set is simplified; furthermore, this approach led to significantly higher performance. For this reason, understanding pipelining is a prerequisite for understanding RISC. This section briefly explains pipelining; for a longer presentation, see Chapter 6 of Ref. [2], and for deeper treatments see Chapter 6 of Ref. [1] and Ref. [4].

20.2.1 The basic five-stage RISC pipeline

The basic pipeline employed by the integer part (i.e., non-floating-point) of RISC processors is the five-stage pipeline illustrated in Figs. 20.1 and 20.2. Some classic RISC processors follow this pipeline [5], while others use small variations of it [6, 7]. Figure 20.1

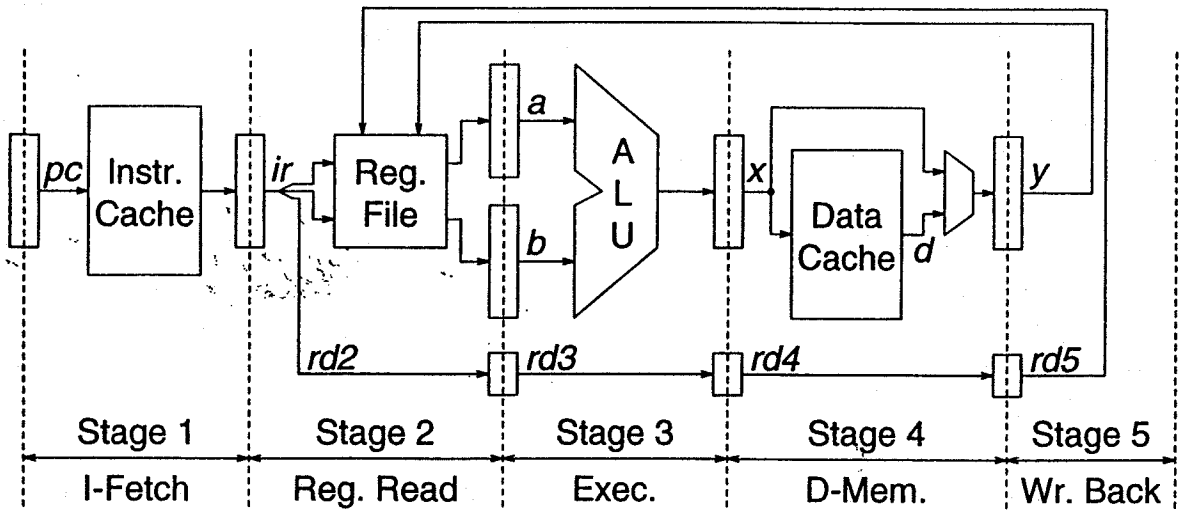

Figure 20.1 Simplified data path of a five-stage RISC pipeline

shows a simplified processor data path organized according to such a five-stage pipeline. Things missing from this figure are the control section, control transfer instruction paths, store data path, immediate constant paths, and bypass paths.

The vertical rectangles along the dashed lines in Fig. 20.1 are edge-triggered registers; they separate the stages of the pipeline. An instruction is fetched and executed by one pass of information flowing from left to right in this figure. It takes five clock cycles for this to complete; in each clock cycle, information advances from the input register(s) to the output register(s) of one pipeline stage. In the first stage, the program counter (*pc*) is used as address for reading the next instruction from the *I*-cache. In the second stage, the instruction register (*ir*) specifies the register numbers of the three operands of the instruction (as well as the opcode, which is not shown). Two of the operands are sources, so they are read immediately from the register file; the third register number, *rd2, is* the destination for the result *y* of the instruction, so it is held for later use. In the third stage, the two source operands, *a* and *b*, feed the arithmetic/logic unit (ALU), which computes their desired function. Two cases must be distinguished: if this is a data memory instruction (*load* or *store*), then the ALU computes $a + b$, which is the effective address for the memory access; otherwise, for the arithmetic or logic instructions, the ALU computes the desired final result. In the fourth stage, load or store instructions use the result of the ALU, *x*, as address for a data cache access; if this is a load instruction, the data read, *d,* constitute the result of the instruction. Non-memory instructions do nothing in this stage—they just pass *x*, the ready result, to the output; this may seem like a waste of time, but we will see that it is not, when we discuss bypassing. Finally, in the fifth stage, the final result *y* of the instruction, which is either *x* or *d, is* written back into the destination register, *rd5,* of the register file.

The data path of Fig. 20.1 is designed so that, in each clock cycle, the entire state of execution of an instruction is contained completely and exclusively within the circuits of *one, single* stage. This is the reason why the destination register number, *rd, is* copied (without modification) from *rd2* to *rd3 to rd4* to *rd5*, thus flowing from left to right in parallel with the data of the instruction. Given this property of the data path, at any time when an instruction is under processing inside it, four of the five stages of the circuit are free. Thus, if other *independent* instructions are available, they can also be processed, in parallel, within the other four stages. Obviously, the instructions that are executing in parallel will always be in different stages of completion each. This parallel processing in the style of an assembly line is called *pipelining.*

As a concrete example, consider five consecutive instructions, *I*1 through *I*5, from a program under execution. Figure 20.2 shows how their execution can proceed through the pipeline of Fig. 20.1, if the latter instructions are *independent* from the former. Assume that the data path was idle (the pipeline was empty); then, instruction *I*1 is fetched from (cache) memory. In the next cycle, say at time *t*2, *I*1 is in *ir*, and its two source registers are being read from the register file. Simultaneously, since the instruction cache is free from *I*1, and since *I*2 was assumed not to depend on *I*1 (i.e., in this case, *I*1 was not a branch or jump), *I*2 is fetched from the instruction cache.

In the next clock cycle, at time *t*3, the source operands of *I*1 are in *a* and *b*, and *I*1's operation is being performed in the ALU. Simultaneously, since the register file is free from *I*1, and since *I*2 was assumed not to depend on *I*1 (i.e., in this case, *I*2 does not read the result of *I*1), *I*2 can use the second stage; that is, its source registers are being read from the register file. At the same time, the instruction cache is free from *I*1 and *I*2, and since *I*3 was assumed not to depend on *I*1 or *I*2 (i.e. neither *I*1 nor *I*2 were branches), *I*3 is fetched from the instruction cache.

Similarly, in the next clock cycle, at time *t*4, *I*1 may be accessing the data cache if it is a load or store instruction, or else it may be doing nothing. Simultaneously, *I*2's operation is being performed in the ALU. At the same time, the source registers of *I*3 are being read from the register file, since we assumed that *I*3 does not read the results of *I*1 or *I*2. Also, at *t*4, *I*4 is fetched from the instruction cache, since none of its three previous instructions was a branch or a jump. In the fifth cycle (rightmost column in Fig. 20.2), the pipeline is full: the final result of *I*1 is being written back into the register file, while *I*2 is in the fourth stage, *I*3 is in execution, *I*4 is reading its source registers, and *I*5 is being fetched. Notice that three accesses to the register file need to be going on in parallel: one write (of an older instruction's result), and two reads (of a younger instruction's sources); thus, the register file needs to be *three-ported.*

From that cycle on, the pipeline may continue being full and operating in a similar manner, if every instruction is *independent of its four previous* instructions. Five consecutive instructions will always be in the pipeline, each at various stages of completion. Given our assumption of mutual independence, none of the instructions in the pipeline needs any of the results of a previous instruction that is also in the pipeline (at most, four such instructions exist), and thus execution can proceed unhindered at the rate of one instruction every clock cycle, rather than one instruction every four to five clock cycles if pipelining were

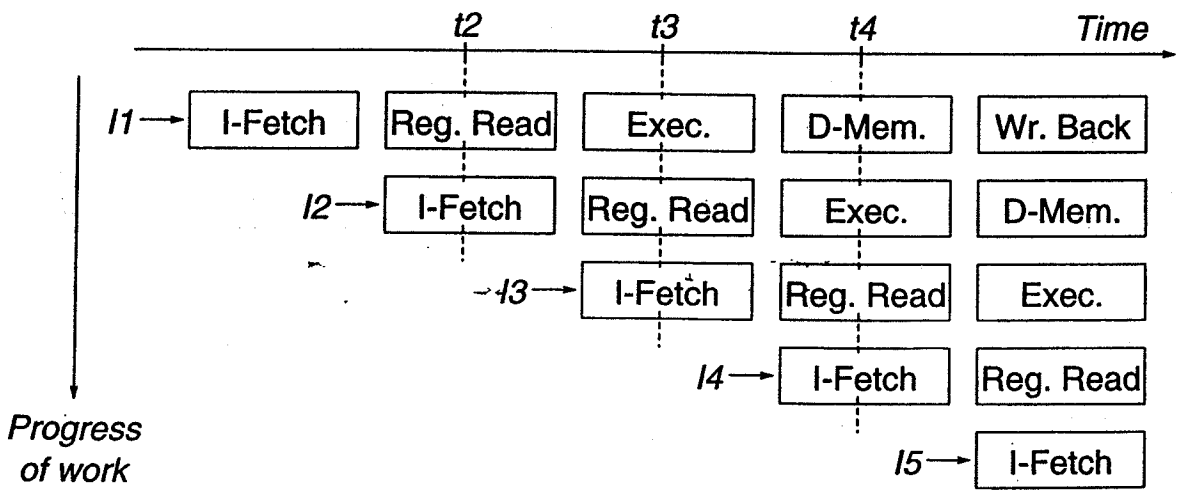

Figure 20.2 The basic five-stage RISC pipeline

not used. Of course, life is not as simple as the above ideal picture! Lots and lots of instruction dependences exist in real programs, so a real data path must be more complicated than Fig. 20.1, as explained next.

20.2.2 Dependences and bypassing

Consider the following simple example of dependence between two consecutive instructions in a program: *I1*: *add* $R1 + R2 \rightarrow R3$; *I2*: *sub* $R3 - R4 \rightarrow R5$; these compute the sum of two numbers minus a third number. Instruction *I2* depends on *I1*, since it needs to read the result of *I1* as a necessary source operand for its computation. Referring to Fig. 20.2, it would not be possible to execute *I2* with the timing shown there, because that would produce the erroneous effect of *I2* subtracting *R4* from the *old* value of *R3*, which was read at *t2*, rather than from the new value, which will only be written into the register file in the fifth cycle. For correct execution, under the circumstances of Figs. 20.1 and 20.2, *I2* would have to wait until the fifth cycle for its fetch stage (i.e., as *I5* in Fig. 20.2), or at best until the fourth cycle (as *I4*) if we assume that the register file has the property of supplying the new value when the same register is simultaneously written and read. In the simple data path of Fig. 20.1, all dependences lead to underutilization of the pipeline and loss of performance.

However, there is often something better that can be done. Upon closer examination, one sees that *I2* needs the new value of *R3* (the result of the addition) when it starts performing its operation (subtraction), that is at the beginning of the fourth clock cycle (the cycle of *t4* in Fig. 20.2). In the meanwhile, this desired value (the result of the addition) has *already been produced* by *I1*, during the third cycle, in the ALU. The only thing that has not yet been done is to write this result in its proper place, into *R3* in the register file. If *I2* uses the value that it reads at *t3* from the register file, it will fail; however, if it *bypasses* stages 4 and 5 of *I1* as well as its own stage 2, it can get the correct value directly from the source—from the output of the ALU.

Figure 20.3 shows the additions to the data path of Fig. 20.1 that are necessary for this bypassing to work. Only the second and third stages are shown, since they are the only ones that need modification. Two multiplexers are added after the two outputs of the register file, allowing one or both of its output values (the "old" values) to be discarded and replaced by the correct, new value that has just been computed by the ALU. The right-hand side of Fig. 20.3 shows the timing of this bypassing, by reference to Fig. 20.2. In the third

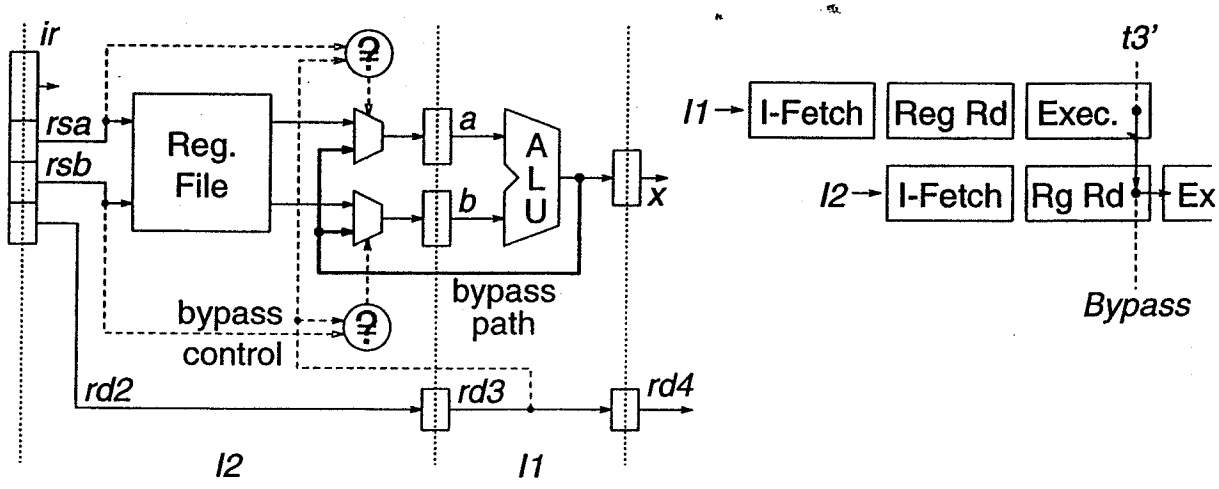

Figure 20.3 Bypassing from the third to the second stage

clock cycle, *I*1 is in the third pipeline stage, computing the new value of *R*3, while *I*2 *is* in the second stage, reading the old value of *R*3 from the top port of the register file (*rsa* = 3). What needs to be done is to switch the top multiplexer to its bypass input so that the old value is discarded and the new value is used instead. The control circuit necessary to make this decision is relatively simple, as shown in Fig. 20.3. The destination register of *I*1, *nd*3, is compared to the two source registers of *I*2, *rsa* and *rsb* (notice here the important distinction between *rd*2 (*I*2's destination) and *rd*3 (*I*1's destination)). Whenever equality is detected, this signals a dependence, and bypassing must be enabled. One additional detail to be checked is whether the instruction that is currently in the third (execution) stage has its write-destination-register flag asserted, since not all instructions write a result into a register. Time-wise, bypassing is decided during the second stage of the "receiving" instruction (*t*3 in Fig. 20.2), and it actually occurs at the very end of that cycle (*t*3′ in Fig. 20.3). Notice that this bypassing technique allows a series of consecutive dependent instructions to be executed at the maximum possible speed: the result of an operation in the ALU is fed back to it right away and is used as an input to the next operation, in the very next cycle. We will return to this point in Section 20.3.1. The clock cycle is determined by the delay of the ALU plus one multiplexer and one register delay.

Now consider a more complicated case. If *I*1 is a *load* instruction, then its result is produced in the fourth pipeline stage, rather than in the third as for the *add* instruction above. This means that when *I*2 enters its execution stage, at the beginning of the fourth clock cycle in Fig. 20.2, the result of *I*1 is *not yet* available, so it cannot be bypassed! In other words, either *I*2 must not use the result of *I*1, or *I*2 will have to wait, with a resulting performance penalty. Figure 20.4 shows the former case, where *I*2 does not read the register into which *I*1 will write. Now, assume that the next instruction, *I*3, does need the result of *I*1, as in the example of Fig. 20.4: *I*1: load *M*[...] → *R*3; *I*2: ...; *I*3: sub *R*3 – *R*4 → *R*5. Can *I*3 be executed without any pipeline delay? As we see, when *I*3 enters its execution stage, in the fifth clock cycle, the desired new value of *R*3 has been produced by *I*1, since it has been returned from the data cache to the processor in the fourth clock cycle; hence, *I*3 can take and use it with bypassing. Figure 20.4 shows (in bold line) the new bypass path that must be added to the data path of Fig. 20.3 for this new bypassing to work. As shown

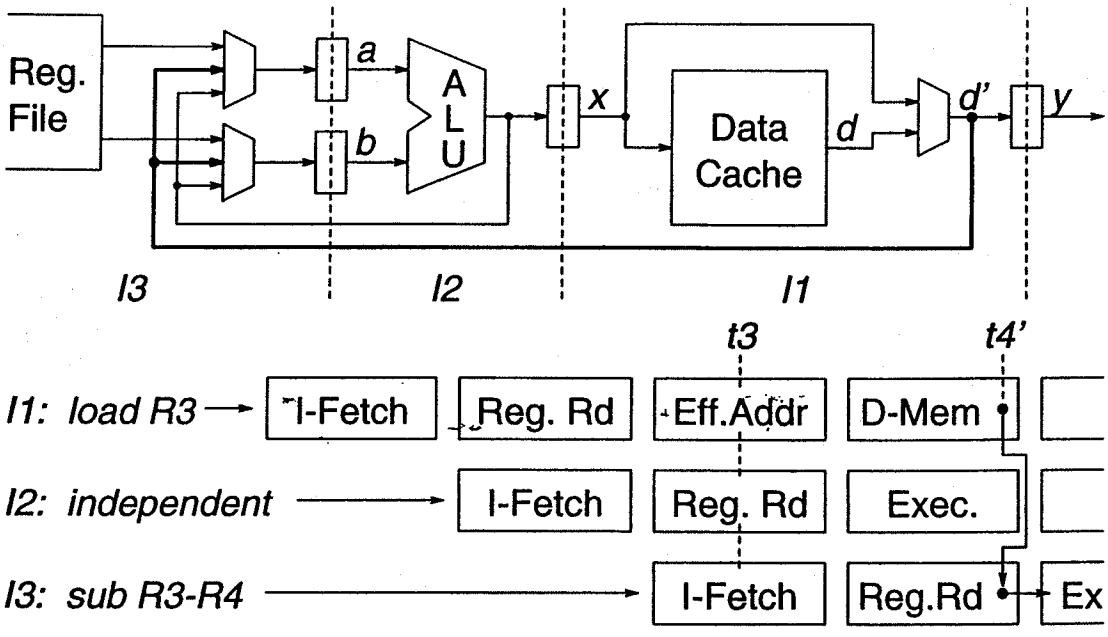

Figure 20.4 Bypassing from the fourth to the second stage

in the timing diagram, in the lower part of the figure, this new path is used at time $t4'$, right after the completion of the memory access of $I1$, and right before $I3$ enters the execution stage. As before, the bypass decision is made in the fourth cycle, when $I3$ is in the second stage; the control circuit that makes the decision compares the source register numbers of $I3$, *rsa* and *rsb* (Fig. 20.3), against the destination register number of $I1$, *rd4* (Fig. 20.1). Notice that bypassing works correctly whether $I1$ is a load or an arithmetic instruction: the value bypassed is d', which will equal d if $I1$ is a load, or x if $I1$ is an arithmetic/logic instruction.

What happens if $I1$ is a *load* instruction, and $I2$ is not independent of it as assumed in Fig. 20.4? One of the original research RISC machines, the MIPS processor [8], would let $I2$ execute right away and use the *old* value of its source registers. It was left to the compiler to place a no-operation instruction after the load, if no other useful and independent instruction could be found, when the new value of the source registers was needed; this was called *delayed load.* The modern commercial RISC processors delay the execution of $I2$ for one cycle when they detect a dependence from its previous load instruction. After this one cycle delay, bypassing can be used to let $I2$ proceed, as in Fig. 20.4. All other instructions beyond $I2$ are also delayed by one cycle, so there is a performance penalty in this case (similar to the execution of a useless no-operation instruction in the research MIPS machine). In Section 20.3, we will discuss how the optimizing compiler should rearrange the object code of a RISC program in order to avoid these performance losses and how this relates to CISC architectures.

20.3 Dependences and Parallelism in CISC and in RISC

When there is no low-level parallelism in a program, the operation dependences are the limiting factor for performance. When low-level parallelism exists, the limiting factor for performance is the number and type of hardware resources that are available for performing the operations. In this section, we will see how CISC architectures try to approach these limits and how RISC architectures do it. The conclusion will be that the RISC approach is more flexible (it applies to more cases) and less costly. This section will also show why implementing a task "in hardware" (as a single instruction) is not necessarily faster than implementing it "in software" (using multiple, simpler instructions).

20.3.1 Dependence and resource limited performance

We will use a typical CISC instruction as a first, long example: a memory-to-memory, three-operand add; i.e., two words must be read from memory, added together, and the result written to a third location in memory. We will compare how this task is performed in a CISC machine and in a RISC machine. We assume a frequently used memory addressing mode (which is also the usual or the only RISC addressing mode): the address of each operand is the sum of the contents of a register plus a constant that is contained in the instruction. We assume *similar data path resources* for the two machines: a single-ported data cache, which is independent of the instruction cache, one general-purpose arithmetic/logic unit, and a register file. We count the number of cycles for execution, assuming that one cycle can contain a register file access, or an ALU operation, or a data cache access.

A first observation is that because our task requires three memory accesses, and because we have a single-ported data cache, it is impossible for the task to be completed in less

than three cycles. This number—the number of operations divided by the number of functional units that can perform them—is one of the two fundamental performance limits that were mentioned above. The second observation is that the memory write access cannot be performed before the addition is completed, the addition cannot start before both memory read accesses have been completed, and the read accesses cannot start before their address calculations are done. Given that an address calculation takes two cycles (register read, add), and the two memory reads cannot be performed concurrently, our task cannot be completed in less than six cycles (two for the first address calculation, two for the memory reads, one for the data addition, and one for the memory write). This new lower limit also takes into consideration the second fundamental performance limit that was mentioned above: the data dependences, i.e., the fact that certain operations cannot be started before certain others have completed.

Now let us see how CISC and RISC architectures try to achieve high performance; that is, how they try to approach the above lower limit on the number of execution cycles. Figure 20.5 shows a possible implementation of our task as a single instruction in a CISC machine with the above hardware resources, and a typical implementation as a small piece of code in a RISC machine with the pipeline of Section 20.2. The idea behind complex instructions like this one in CISC architectures was to have a large enough task in the instruction that considerable parallelism would exist in it, and hence this parallelism could be exploited. Accordingly, Fig. 20.5 shows an implementation of this "add_memory" (*addm*) instruction with up to three operations going on in parallel. Not all CISC machines would do it in this way. There are several reasons [such as design complexity, structured microprogramming (e.g. microsubroutines), and so on] that have led several CISC machines to suboptimal implementations of instructions. However, the possibility shown in the figure is one of the claimed advantages of CISC, so it is of interest to examine how it relates to RISC.

The bottom part of Fig. 20.5 shows how this task would be implemented in RISC. Two *load* instructions are used to bring the desired addends from memory into temporary registers. The effective addresses of the *loads* are sums of a register and a constant, each; this is

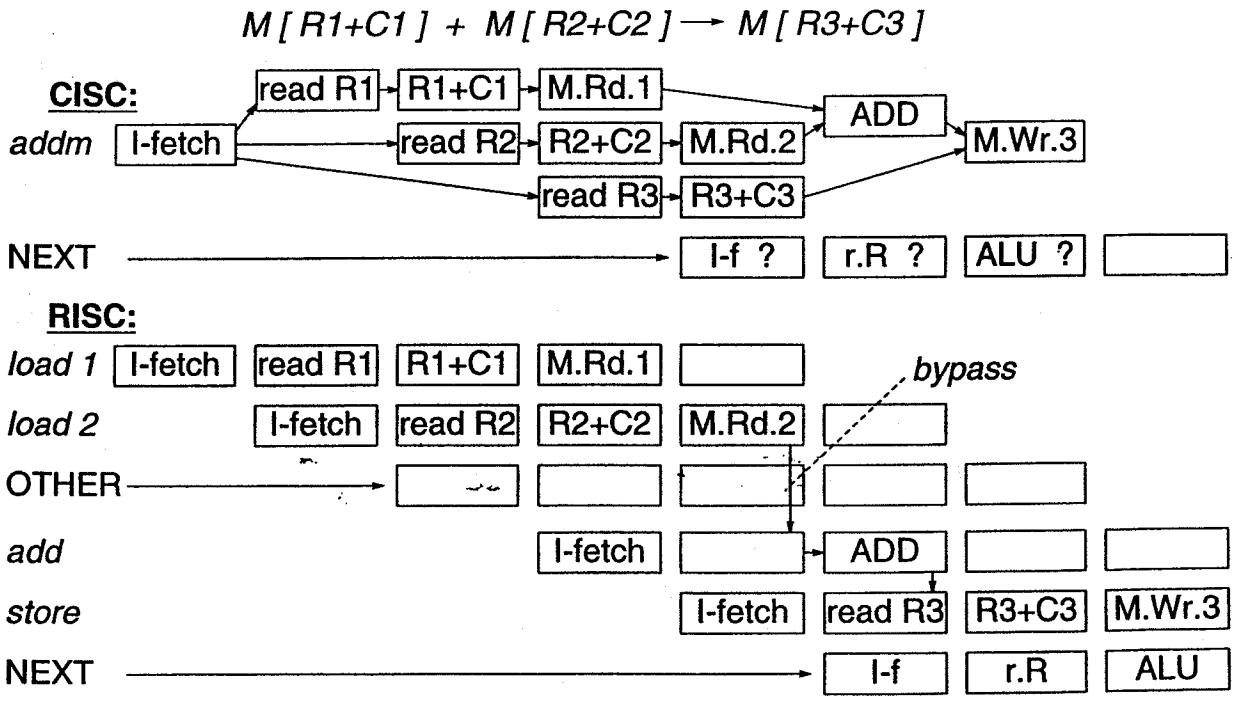

Figure 20.5 Memory-to-memory addition in CISC and RISC

slightly different from what was said (for simplicity) in Section 20.2 (sum of two registers), but the respective timing and pipeline organization remain the same. An *add* instruction follows after the two *loads,* adding the two above temporary registers and placing the result into a third one. This is a case like the one of Fig. 20.4, so we will either lose a cycle or insert another independent instruction between the second *load* and the *add.* Figure 20.5 shows the latter situation, labeling this instruction as *OTHER;* Section 20.3.2 discusses what this instruction can be. A *store* instruction follows after the *add,* writing the contents of the third temporary register into the desired memory address. As seen in Fig. 20.5, the load data are bypassed to the *add* instruction, and the result of the latter is bypassed to the *store* instruction. With these bypassings, the timing of the RISC implementation becomes very much similar to the (optimistic) CISC implementation shown in the same figure. The only difference is that the memory write access is performed one cycle later than in the CISC machine, due to the *store* instruction following after the *add* instruction in RISC.

To determine the actual performance, it is necessary to examine both cases in their overall execution context. This can be done by examining when the "next" instruction can start execution. Figure 20.5 shows such an instruction, labeled *NEXT.* For the RISC case, its timing is clear. For the CISC case, Fig. 20.5 shows an *aggressive* design (maximum possible pipelining for the given hardware resources), which many actual CISC machines do not follow. Many CISC machines use no pipelining, or use a modest amount of it, due to several design complexities that arise from the combination of complex instructions and pipelining (see Section 20.5.1). In the case of such an aggressive CISC design, the *NEXT* instruction is executed one cycle earlier in CISC than in RISC. However, RISC has executed one additional instruction in the meanwhile—the *OTHER* instruction. If this other instruction is a useful one, which it often is, then CISC will also have to execute that, and both machines will achieve exactly the same performance on this example.

No premature conclusions should be drawn based on this single example. In reality, many things can be different from the above simplified picture, as discussed further in Sections 20.4 and 20.5: the cycle time of the two machines may not be the same, the CISC machine may have less pipelining, the CISC implementation of the instruction may not be as good as in Fig. 20.5, there may be an additional "instruction decode" cycle in the CISC machine, the relative cycle counts may be different for other tasks, and so on. Still, several interesting facts can be discussed based on the above example, and we do so in the rest of this section.

First of all, as the above example illustrated, when performance is limited by dependences, there is no advantage to be gained by going to complex instructions. Even though the typical RISC (integer) instruction takes five cycles to complete (Section 20.2.1), *bypassing* (Section 20.2.2) allows dependent computations to be performed at the peak rate and with the minimum latency allowed by the ALU and data cache hardware delays. As seen in Fig. 20.3, an ALU computation using the result of a previous similar computation is performed in RISC with a delay relative to the previous computation as short as the ALU hardware delay plus one multiplexor and one register delay (the ALU delay being the dominant one among the three). Nothing better than that is possible—in CISC or in RISC. Notice that we do not discuss here issues of code size and instruction fetching; see Section 20.4 for that. Also, we assume well balanced delays among the ALU, the register file, and the cache memories; the slowest of them determines the clock cycle time, so we want all of them to be close to each other. This latter point affects RISC and CISC processors in a similar way (except that CISC may have a disadvantage if its (complex) control circuitry is slower than the above three data path components).

20.3.2 Instruction scheduling: RISC + compiler vs. CISC

To gain performance, we try to find useful operations to be performed in parallel with and while waiting for the results of other operations whose delay, due to dependences, cannot be reduced. In CISC, this is primarily done by defining compound instructions that contain a number of operations, some of which can be performed in parallel with the others. One example was seen in Fig. 20.5: while the first operand is being read from *M*[*R*1 + *C*1] the address of the second operand is being computed, and so on. Consider another example: *auto-increment* (or autodecrement) addressing mode. In this example, while an operand is being read from memory, the register that was used for computing its address is incremented (or decremented) by the size of the operand. This is sometimes useful, e.g., when a program is sequentially stepping through an array of operands, and hence auto-incrementing prepares the register for the access to the next element in the array. (Statistics show that the number of cases where this applies is not very impressive: about 3 percent of the VAX memory operands in three benchmark programs used auto-increment/decrement mode (see Ref. [1] p. 170); yet, the point to be made here is independent of this low usage.)

In RISC, the execution of useful operations in parallel with and while waiting for the results of other operations is done first by exploiting the *pipelined* execution of instructions (as in Fig. 20.5), and second by the *compiler,* which properly *schedules* the sequence of instructions in a program. Thus, while in CISC the operations to be executed in parallel have to have some relation with each other so that it is realistic for them to be encoded as a single instruction, in RISC these operations can be *unrelated to* each other, since they are described by separate instructions each. This gives RISC a considerable advantage relative to CISC: RISC is *more flexible* than CISC in finding and exploiting parallelism. To put it in different words, CISC determines and defines the exploitable instruction-level parallelism at machine-design time—roughly, once per decade, and once globally—while RISC does so at compile time—once per program, and specifically for that program.

Instruction scheduling [9] is one of the basic techniques in modern compiling (see Ref. [1], p. 114). It appeared at about the same time as RISC architectures, and it enabled RISC to achieve better performance than CISC at lower cost. It applies to all operations whose execution takes more than one clock cycle. Typically, these are loads, branches, and floating-point operations. We will illustrate the technique using loads as an example. By carefully studying Fig. 20.4, the reader will see that, in a processor where a data cache access takes *N* clock cycles (usually $N = 1$), the *N* instructions after any *load* instruction should be independent of the data being loaded (and perform useful work) to avoid performance loss. These *N* (usually one) instructions are often called *load delay slots,* because they are instruction slots that the compiler should try to fill with operations independent of the loaded data for the load delay to be "hidden." Modern optimizing compilers do this by instruction rearrangement (code motion): some instructions are moved later or earlier than their previous position in a block of code. To preserve program correctness, this motion (usually) can be done only when the instructions being swapped are mutually independent and reside in the same basic block (a block of code always executed in its entirety, i.e., containing no branches or labels).

As a first example of instruction scheduling, and specifically of filling load delay slots with independent instructions, consider the above case of auto-increment addressing mode. Assume that we are in a loop that was originally compiled as: *load Rtmp* ← *M*[*Rp* + *offs*]; *compare Rtmp to Rkey; increment Rp; branch.* The *compare* and the *increment* instructions are independent of each other, and thus their positions can be interchanged. This interchange leads to better performance, since the incrementation is independent of the

load—a fact that was not true for the comparison. In the new code sequence, the first two instructions, *load* and *increment,* are the equivalent of the CISC auto-increment addressing mode. Since the incrementation is done by a separate instruction, the address register can be changed by any amount—not just the size of the operand being loaded as is customary in CISC auto-increment addressing mode. For example, when stepping through an array of structures, this must be the size of the structure. As seen, RISC is more flexible than CISC here.

As a second example of filling load delay slots, consider a case that can be exploited in RISC, owing to its flexibility, but not in CISC. Assume that we wish to add three integers from memory, *A, B,* and *C* (with addressing mode, as in Fig. 20.5), and we wish the sum to be placed into register *Rd.* In CISC, this will be done using two addition instructions: *add Rd* ← *A* + *B; add Rd* ← *Rd* + *C.* The first of these instructions includes two reads from data memory; the processor stays idle during the second of them, waiting for the data to arrive from the cache (the timing is as in Fig. 20.5, but no *R*3 + *C*3 needs to be computed now). The second instruction includes one read from data memory; the processor also stays idle during that period.

In RISC, the code sequence starts out as: *load Rd* ← *A; load Rt* ← *B; add Rd* ← *Rd* + *Rt; load Rt* ← *C; add Rd* ← *Rd* + *Rt.* The first *add is* in the delay slot of the second *load,* but it depends on that load, so it is desirable to move it farther down in the instruction block. At first sight, it appears not to be interchangeable with the third *load,* because that *load* alters *Rt*—one of the source registers of the *add.* However, *Rt* is just a temporary register, used first to hold *B* and then to hold *C.* There is no need to use the same temporary register for the two purposes; if we use *Rt*1 for *B* and *Rt*2 for C, the first *add* and the third *load* become mutually independent, hence interchangeable. This technique is called *anti-dependence elimination* in instruction scheduling terminology. Now, the RISC code sequence becomes: *load Rd* ← *A; load R*1 ← *B; load Rt*2 ← *C; add Rd* ← *Rd* + *Rt*1*; add Rd* ← *Rd* + *Rt*2. In this new code sequence, the third *load* fills the delay slot of the second *load,* and the first *add* does the same for the third *load;* the pipeline never stalls, and the processor never stays idle! The reader may draw, as an exercise, a diagram similar to Fig. 20.5 for this case. If we label as "1" the cycle when the first instruction of each code sequence is fetched, then CISC will fetch its second instruction in cycle 5, and it will fetch the *NEXT* instruction after this code block in cycle 9 (instructions cannot be overlapped so much that their ALU cycles coincide, since we assumed a single ALU, and no CISC overlaps instructions so much as to interchange their ALU cycles, because of the difficulties mentioned in Section 20.5.1). On the contrary, RISC will fetch each of the five instructions of its (new) code sequence in cycles 1, 2, 3, 4, and 5, respectively, and will fetch its *NEXT* instruction after the block in cycle 6: RISC is faster than CISC by 3 clock cycles (or 60 percent) in this example.

Instruction scheduling at compile time is called *static* because it is done once for all executions of a specific code segment. It is also possible to perform instruction scheduling *dynamically,* i.e., at run time (see Ref. [1] Section 6.7, and Ref. [10]); obviously, this cannot be done by the compiler—it must be done by the hardware (by the processor control unit). Dynamic scheduling is also called *out-of-order execution* of instructions; supercomputers of the past have supported this. Dynamic scheduling is obviously much more expensive than static scheduling, since the implementation of the former needs considerable hardware, while the latter is done purely in software and not at run time; among others, dynamic scheduling usually costs one extra pipeline stage, which increases the branch cost (see Section 20.4.3). Dynamic scheduling is useful only when static scheduling cannot fill all the delay slots because it is not known at compile time whether some instructions are

mutually independent so as to be able to interchange their positions. In these cases, dynamic scheduling will interchange their execution at run time, only when they actually turn out to be independent, thus saving clock cycles in those cases. The most interesting of such cases of potential dependences that cannot be resolved at compile time are (1) instructions not in the same basic block (instruction scheduling across conditional branches), and (2) memory loads and stores whose effective addresses cannot be proven to differ from each other (compile-time "address disambiguation" fails). However, modern compiling has progressed considerably: the former case frequently can be treated by trace scheduling, loop unrolling, or software pipelining (see Ref. [1], Section 6.8), while sophisticated address disambiguation techniques [11] have reduced the frequency of occurrence of the latter case. It follows that dynamic scheduling, with its high cost, is rarely needed and thus rarely provided in modern processors. The general advice is: Do not postpone for run time what you can do at compile time!

20.3.3 Parallel hardware: superscalar RISC vs. CISC

In all of the preceding examples, we assumed that the CISC processor has one ALU, a three-port register file, and a single-port data cache, just like the typical RISC processor (Fig. 20.1). Was that a fair comparison? What if the CISC designer is willing to pay for more functional units in the data path so that the multiple operations in the complex instructions can all be performed in parallel and faster than the typical RISC processor of Section 20.2? In fact, the way we compared RISC and CISC up to now in this section *is* fair: the designer of any processor—CISC or RISC—can decide to include in its data path more functional units than what was assumed in Fig. 20.1. Usually, these are additional register file ports, additional ALUs, and more complicated instruction decoding and control logic. (Multiported data caches are usually too expensive to justify having them.) These additional functional units increase the cost of the processor (e.g., its silicon area), but also increase its performance, since more operations can now be performed in parallel. (Beyond a certain number of unit, performance gains become minuscule relative to the cost increase, so this is *not* a source of infinite performance improvement.)

CISC processors exploit their multiple functional units by performing in parallel the multiple operations in the currently executing complex instruction. RISC processors, on the other hand, exploit their multiple functional units by fetching, and subsequently executing, multiple *instructions* in each clock cycle of their pipeline. These are called *superscalar or* VLIW (very long instruction word) processors; a close relative of theirs are the *superpipelined* processors; see Refs. [7, 10, 12–16]. What was said for a single-ALU processor also applies to the case of multiple functional units: the RISC method of independently controlling each functional unit via a separate instruction in each clock cycle, and of scheduling the placement of these instructions during compilation, is more flexible and yields better results than the CISC method.

20.4 Instruction Alignment, Size, and Format

One of the differentiating characteristics of RISC relative to the earlier CISC architectures is the (almost always) fixed 32-bit instruction size, and the few, simplified instruction formats. In this section we explain the reasons behind these choices. Since they are related to the execution of control transfer instructions, we also treat the topic of delayed branches in Section 20.4.2.

20.4.1 Code size, I-cache Size, and fetch throughput

The size of usual RISC programs (binary code size) is considerably larger (usually 20 to 70 percent) than the size of the same programs in the popular CISC architectures of the 1970s and 1980s. Examples are measurements described in Ref. [1] (p. 79) of 35 to 70 percent, and Ref. [17] (p. 80) of 20 to 50 percent. The reason is twofold: (1) there are more instructions for a given task in RISC than in CISC (50 to 180 percent more instructions executed (dynamic measurements) by RISC relative to the VAX, according to Ref. [1], p. 123), and (2) more code space is lost in RISC due to "fragmentation," because RISC code is quantized in multiples of 32 bits, while CISC instruction size is often quantized in multiples of 8 bits. The former—more instructions in RISC—is a result of the smaller and simpler RISC instruction set, the advantages of which are explained throughout this chapter. The reason for the latter—the simple instruction format and fixed instruction size of RISC—is the saving of the alignment and decoding time, as detailed in Section 20.4.3. As for the disadvantages of the resulting increased code size in RISC, three of these exist or have been claimed at various times, and we proceed to discuss them here.

First, the increased code size of RISC means that more disk space is needed for the files containing the binary code of executable programs. A similar case holds for the main memory of the computer. The evolution of the RISC versus CISC controversy in the eight years that have passed since RISC appeared commercially shows that the customers prefer RISC for its higher performance and do not mind paying a little bit more for the increased disk and memory space. Factors in favor of RISC are: (1) the price per byte of disks and memory is steadily decreasing; (2) binary code occupies only a small fraction of disks and memory—data usually occupy the majority; and (3) dynamic linking of library routines has further reduced the above fraction.

Second, the increased code size of RISC means that a correspondingly larger instruction cache is needed to hold an equivalent working set, thus achieving a similar hit ratio. Given that it is desirable to have the (first level) instruction cache on the same chip as the processor, the chip area limitation poses a performance problem: for a given cache area (and, hence, capacity) RISC will suffer a lower hit ratio. On the other hand, the fixed instruction size, the simple instruction format, and the reduced instruction set of RISC result in large savings of processor chip area; no alignment is needed, instruction decoding is simplified (see Section 20.4.3), and no microcode memory is needed (see Section 20.5.2). Although it is difficult to precisely quantify these savings, in general they are more important than the increase needed in instruction cache size in order to maintain a similar hit ratio.

Third, it has been claimed that processor performance is limited to a large extent by the throughput of the path that feeds instructions to the processor, and hence, since RISC has a larger code size and needs a correspondingly larger instruction fetch throughput, the performance of RISC will be worse because of this effect. Today, when the instruction cache is on the same chip with the processor, it is certainly *not* true that the instruction fetch throughput is a bottleneck. But, even at the time when this claim was made—when there was no cache or the cache was not on the processor chip—this claim did not reflect reality. To see why, we must differentiate between *average* and *peak* instruction fetch throughput. CISC indeed needs less *average* fetch throughput than RISC, as we saw above. However, for an efficient (CISC) computer design, the *peak* throughput of the instruction fetch path must be significantly higher than the above average to avoid loss of performance on long but fast-executing instructions, such as to set a register to a 32-bit constant, where the constant is contained in the instruction. If the actual circuit paths can supply this higher instantaneous throughput when it is needed, there is no reason why they could not do so on a longer-term basis; the average throughput can be made equal to its peak instantaneous

value at no extra cost. RISC does just that: its peak instantaneous requirements are for one instruction (32 bits) per cycle, and its average is the same in the lack of pipeline dependences. It is very hard to imagine a well performing CISC with a peak fetch throughput of less than 32 bits per cycle, because (1) most circuit path widths are quantized to powers of 2, and (2) there are several frequent instructions that need to execute in one cycle and that do not fit in 16 bits. The claim about reduced fetch throughput yielding higher performance looks more realistic if the processor contains an *instruction fetch buffer*—a FIFO queue that absorbs the fetch throughput fluctuations. However, such an instruction buffer must be invalidated after every successful branch instruction—roughly every 10 to 20 clock cycles—and thus its effect is greatly reduced. Finally, instructions whose size is not an integer multiple of the word width will present the problem of alignment, which is particularly manifest when the instruction at a branch target is split between multiple memory words, as discussed in Section 20.4.3.

20.4.2 Branches under pipelining: delayed branches

We divert briefly from our discussion of instruction size to see how branch instructions are executed in a pipelined processor, and especially to discuss delayed branches, which are a form of instruction scheduling (see Section 20.3.2). Figure 20.6 (top) shows the first two stages of the pipelined data path of Figs. 20.1 and 20.3, with the additional paths needed to fetch sequential instructions and to execute conditional branches. PC-relative addressing mode is assumed for the branches, as is customary in all modern processors: the branch target is the sum of the address of the branch instruction itself plus a (sign-extended) constant *offset,* which is contained in the instruction. Figure 20.6 shows a "fast" version of branch, which can be completely executed in the second stage of the pipeline. To achieve that, the branch condition can only be the result of testing a source register for sign and/or equality to zero; no register-to-register comparisons are provided, since they would require

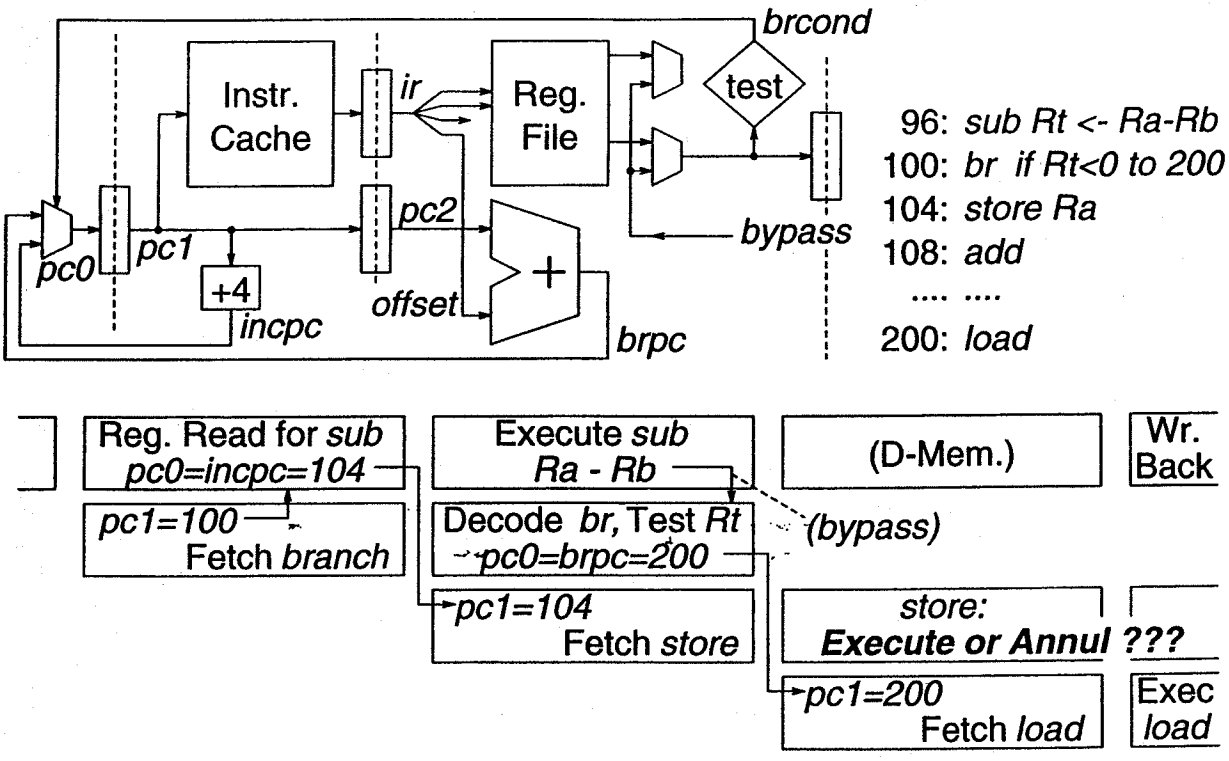

Figure 20.6 Conditional branch data path and timing

a full ALU carry-chain propagation delay. The register that is tested may contain the result of a previous full register-to-register comparison, in the form of a subtraction, as shown in the sample program fragment in Fig. 20.6. Using general-purpose registers rather than special-purpose *condition code* bits to store the result of comparisons increases the flexibility of instruction scheduling (see Section 20.3.2) and simplifies the hardware; see also Ref. [8] and Ref. [1], p. 106.

In Fig. 20.6, *pc*1 is the program counter (PC) of the instruction that is currently in the first stage of the pipeline: the address from which the instruction is being fetched; *pc*2 is the PC of the instruction currently in the second stage; *pc*0 is the address from which an instruction will be fetched in the next clock cycle. This latter instruction, from address *pc*0, can be determined in two different ways: either as the next sequential instruction after the one currently being fetched (*incpc*), or as the target of a branch instruction currently being decoded and executed in the second pipe stage (*brpc*). Due to pipelining, when a branch instruction is executed, it can only affect the instruction fetched *after its next* instruction. In the example program fragment shown in Fig. 20.6, the *store* instruction is fetched from address 104 while the *branch* is being decoded and executed—the *branch* can only affect whether the instruction fetched after the *store* will be the *add* from address 108 (branch failure) or the *load* from address 200 (branch success). As seen, branch instructions have a *delayed* effect that is quite similar to the delayed effect of load instructions seen in Section 20.2.2.

What should be done with the "extra" instruction that is fetched while the branch is being executed (*store* in Fig. 20.6) when the branch is successful? Should the processor always *annul* (squash, abort) it, like traditional processors (including CISC) did? But then, successful branch instructions will cost two clock cycles—one more than normal. Alternatively, the processor can always execute this "extra" instruction; this is called *delayed branch,* and the place occupied by the "extra" instruction is called *delay slot.* The advantage is obviously that no clock cycle is lost; the "problem" is that the instruction in the delay slot is always executed, regardless of the branch outcome, as if it were *before* the branch, yet the branch instruction cannot depend on it, since this instruction is actually executed *after* the branch. This strange behavior was considered undesirable in the old CISC machines, since the instruction set of those processors had to be "clean" to be appropriate for hand coding in assembly language. However, with the advent of optimizing compilers, the above behavior becomes merely another case of instruction scheduling, similar to the case of load instructions that was presented in Section 20.3.2. Hence, many RISC architectures do have delayed branches, and they similarly have delayed jumps, calls, and returns (i.e., unconditional control transfers).

There are many issues related to branch optimization, and much literature on them. For example, see Ref. [1], pp. 272–277 and 307–314, and Refs. [18] and [19]. Due to lack of space, we only mention some of the issues, and only briefly. In processors where branches take more cycles to be resolved, there are correspondingly more delay slots. Delay slots can be filled either (preferably) with instructions that are moved there *from before* the branch, provided there is independence that allows them to move, or else with instructions that are moved there from one of the two branch targets—preferably from the most likely *predicted target,* provided that these are *harmless* when the branch goes in the other direction, or that the hardware *annuls* them when that happens, or else with *noop* instructions (instructions that do nothing). In case the target address can be computed before the branch condition becomes resolved, the instructions fetched (and conditionally executed) in the meanwhile can be from that target, if the branch is predicted to usually succeed. The first (or only) delay slot after branches can be filled with useful instructions in many cases;

sor are the fastest level of the memory hierarchy. Hence, it is essential to use them as much as possible and to use them efficiently. For this reason, RISC architectures rely to a large extent on register-resident operands; memory is accessed only through separate instructions (*load* and *store*). Instruction fetch and decode latency is minimized when instructions have a fixed size, when they are aligned on word boundaries in memory, and when the source operand field positions are fixed in them; RISC architectures do all of these. Reduced instruction set computers were the result of deeply understanding both hardware and software, and of applying quantitative engineering analysis to the design of both, and to the design of their interactions and their interface—the instruction set.

After the "plain" RISC processors of the 1980s, superscalar RISCs (Section 20.3.3) were the next step in the evolution, in the early 1990s. Currently, in the mid-1990s, it looks like the hottest topic is *latency tolerance,* i.e., finding and performing useful work in parallel with waiting for long-running operations to complete. Interesting techniques in this direction include multiple-context processors [26], access/execute decoupling [27], and software pipelining [28].

20.6.3 Reading list

For those readers who know very little about computer organization and processor design but want to learn more, we suggest starting with Ref. [2]; for those who know basic computer organization but want to learn about pipelining, we propose Chapter 6 of that same book. For a more advanced treatment of computer architecture in general, readers are referred to Ref. [1]; Chapter 6 of that book treats pipelining in more depth than does Ref. [2]. RISC architectures, as seen in the mid-1980s, are discussed in Refs. [17, 29, 30]. Interesting articles on contemporary RISC processors are Refs. [7, 10, 13–16, 25].

20.7 References

1. Hennessy, J. L., and D. A. Patterson. 1990. *Computer Architecture: A Quantitative Approach.* San Mateo, Calif.: Morgan Kaufmann.
2. Patterson, D. A., and J. L. Hennessy. 1994. *Computer Organization and Design: The Hardware/Software Interface.* San Mateo, Calif.: Morgan Kaufmann.
3. *DEC VAX 11 Architecture Handbook.* 1979. Maynard, Mass.: Digital Equipment Corp.
4. Kogge, P. M. 1981. *The Architecture of Pipelined Computers.* New York: McGraw-Hill.
5. Kane, G. 1987. *MIPS R2000 RISC Architecture.* Englewood Cliffs, N. J.:Prentice Hall.
6. Garner, R. A. et al. 1988. Scalable processor architecture (SPARC). *Proceedings, IEEE COMPCON,* San Francisco, 278-283.
7. Smith, J., and S. Weiss. 1994. PowerPC 601 and Alpha 21064: a tale of two RISCs. *IEEE Computer,* Vol. 27, No. 6, 46–58.
8. Hennessy, J., N. Jouppi, F. Baskett, T. Gross, J. Gill. 1982. Hardware/software Trade-offs for increased performance. *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-I), ACM SIGARCH-10.2 SIGPLAN-17.4, 2–11.
9. Hennessy, J. L., and T. R. Gross. 1983. Postpass code optimization of pipeline constraints. *ACM Trans. on Programming Languages and Systems,* Vol. 5, No. 3, 422–448.
10. Popescu, V. , et al. 1991. The Metaflow architecture. *IEEE Micro,* Vol. 11, No. 3, 10.
11. Padua, D., and M. Wolfe. 1986. Advanced compiler optimizations for supercomputers. *Communications of the ACM,* Vol. 29, No. 12, 1184–1201.
12. Jouppi, N. P., and D. W. Wall. 1989. Available instruction-level parallelism for superscalar and superpipelined machines. *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-III*), IEEE/ACM, Boston, 272–282.
13. 1990. The IBM RISC System/6000 processor (collection of papers). *IBM Journal of Research and Development,* Vol. 34, No. 1.
14. Sites, R. 1993. Alpha AXP Architecture. *Communications of the ACM,* Vol. 36, No. 2, 33–44.

15. Diefendorff, K, R. Oehler, and R. Hochsprung. 1994. Evolution of the PowerPC Architecture. *IEEE Micro,* Vol. 14, No. 2, 34–49.
16. Hsu, P. Yan-Tek. 1994. Designing the TFP Microprocessor. *IEEE Micro, Vol.* 14, No. 2, 23–33.
17. Katevenis, M. G. H. 1985. *Reduced Instruction Set Computer Architectures for VLSI* (ACM doctoral dissertation award 1984). Cambridge, Mass.: MIT Press.
18. McFarling, S., and J. Hennessy. 1986. Reducing the cost of branches. *Proceedings of the 13th Int. Symp. on Computer Architecture,* Tokyo, Japan, ACM SIGARCH Vol. 14, No. 2, . 396–403.
19. Lilja, D. 1988. Reducing the branch penalty in pipelined processors. *IEEE Computer,* Vol. 21, No. 7. 47–55.
20. Johnson, W. 1984. A VLSI superminicomputer CPU. *Proceedings of the IEEE Int. Solid-State Circuits Conf.,* San Francisco, 174–175.
21. Beck, J., et al. 1984. A 32b microprocessor with on-chip virtual memory management. *Proceedings of the IEEE Int. Solid State Circuits Conf.,* San Francisco, 178–179.
22. Radin, G. 1982. The 801 minicomputer. *Proceedings, Symp. on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-I), ACM SIGARCH-10.2 SIGPLAN-17.4, 39–47.
23. Patterson, D. A., and C. H. Sequin. 1982. A VLSI RISC. *IEEE Computer Magazine,* Vol. 15, No. 9, 8–21.
24. Hennessy, J., N. Jouppi, S. Przybylski, C. Rowen, and T. Gross. 1983. Design of a high performance VLSI processor. *Proceedings, 3rd Caltech Conference on VLSI,* Pasadena, Calif., R.Bryant, ed. Comp. Sci. Press, 33–54.
25. Lee, R. B. 1989. Precision architecture. *IEEE Computer,* Vol. 22, No. 1, 78–91.
26. Agarwal, A., B-H. Lim, D. Kranz, and J. Kubiatowicz. 1990. APRIL: A Processor architecture for multiprocessing. *Proceedings of the 17th Int. Symp. on Computer Architecture,* Seattle, Wash. ACM SIGARCH Vol. 18, No. 2, 104–114.
27. Smith, J. 1984. Decoupled access/execute computer architectures. *ACM Trans. on Computer Systems,* Vol. 2, No.4, 289–308.
28. Tirumalai, P., M. Lee, and M. Schlansker. 1990. Parallelization of loops with exits on pipelined architectures. *Proceedings of the IEEE Supercomputing Conference,* 1990, pp. 200-212.
29. Patterson, D. A. 1985. Reduced instruction set computers. *Communications of the ACM, Vol.* 28, No. 1, 8–21.
30. Hennessy, J. L. 1984. VLSI processor architecture. *IEEE Transactions on Computers, Vol.* 33, No. 12, 1221–1246.