

MODERN ARCHITECTURES

9.1 The Emergence of RISC Architectures

The 1960's saw a rapid growth in the complexity of computers. New, sophisticated instructions were made available at the assembly language level, and programmers were writing ever more complex programs. Although assembly language instructions increased in complexity, the instructions were generally more primitive than the high level constructs that programmers used. This **semantic gap** as it is known, fueled an explosion of architectural complexity.

Unfortunately, as computer architects attempted to close the semantic gap, they sometimes made it worse. The IBM 360 architecture has the `MVC` (move character) instruction that copies a string of up to 256 bytes between two arbitrary locations. If the source and destination strings overlap, then the overlapped portion is copied one byte at a time. The runtime analysis that determines the degree of overlap adds a significant overhead to the execution time of the `MVC` instruction. Measurements show that overlaps occur only a few percent of the time, and that the average string size is only eight bytes. In general, faster execution results when the `MVC` instruction is entirely ignored. Although a greater number of instructions may be executed without the `MVC` instruction, on average, fewer cycles are needed to implement the copy operation without using `MVC` than by using it.

Architectures with complex instruction sets that have highly specialized commands (like `MVC`), are known as **complex instruction set computers (CISCs)**. Despite the `MVC` case and others like it, CISCs are not bad. In the 1960's, the speed of a computer's memory was much slower than the speed of the CPU, and the size of the memory was very small. It thus made sense to send a few very powerful instructions from the memory to the CPU, rather than to send a great number of simpler instructions.

As technology advanced, the speed and density of memory improved at a faster rate than the speed and complexity of the CPU. With this shift in performance, it became more economical to increase the speed of the CPU by making it simpler, at the expense of using more instructions to compensate for the reduced complexity of the CPU. This style of architecture is known as a reduced instruction set computer (RISC).

RISC architectures have three primary characteristics that distinguish them from CISC architectures:

- (1) A small instruction set that consists of simple, fixed length, fixed format instructions that execute in a single machine cycle;
- (2) **Pipelined** access to memory (see Section 9.4), and a large number of registers for arithmetic operations;
- (3) Use of an optimizing compiler, in which execution speed is greatly influenced by the ability of the compiler to manage resources, such as maintaining a filled pipeline during branches.

In the next few sections, we will explore the motivation for RISC architectures, and special characteristics that make RISC architectures effective.

9.2 Quantitative Analyses of Program Execution

During the 1970's, when CISC architectures enjoyed high visibility, attention turned to what computers actually spent their time doing. Up to that time, computer designers added more instructions to their machines because it was a good selling strategy to have more functionality than a competing processor. In many CISC machines (such as the IBM 360 and the Motorola 68000), the instructions are implemented in microcode (see Chapter 9 for a discussion on microcode). As a result of implementing a large instruction set in microcode, instruction decoding takes a long time and the microstore is large. Although a large, slow microstore may seem like a bad idea, it could be a good idea if overall execution time is reduced. As we will see, in general, adding complexity to the instructions *does not* improve execution time with present day technology.

Figure 9-1 summarizes the frequency of occurrence of instructions in a mix of programs written in a variety of languages. Nearly half of all instructions are assignment statements. Nearly a quarter of all instructions are `if` conditionals.

Statement	Average Percent of Time
Assignment	47
If	23
Call	15
Loop	6
Goto	3
Other	7

Figure 9-1 Frequency of occurrence of instruction types for a variety of languages. The percentages do not sum to 100 due to roundoff. (Adapted from [Tanenbaum, 1990].)

Interestingly, arithmetic and other “more powerful” operations account for only 7% of all instructions. Thus, if we want to improve the performance of a computer, our efforts would be better spent optimizing instructions that account for the greatest percentage of execution time rather than focusing on instructions that are inherently complex but rarely occur.

Related metrics are shown in Figure 9-2. From the table, the number of terms in

	Percentage of Number of Terms in Assignments	Percentage of Number of Locals in Procedures	Percentage of Number of Parameters in Procedure Calls
0	—	22	41
1	80	17	19
2	15	20	15
3	3	14	9
4	2	8	7
≥ 5	0	20	8

Figure 9-2 Percentages showing complexity of assignments and procedure calls. (Adapted from [Tanenbaum, 1990].)

an assignment statement is normally just a few. The most frequent case (80%), has just a single term on the right side of the assignment operator, as in $X \leftarrow Y$. There are only a few local variables in each procedure, and only a few arguments are normally passed to a procedure.

What we can conclude from these measurements is that the bulk of computer programs are very simple at the instruction level, even though more complex

programs could potentially be created. This means that there may be little or no payoff in increasing the complexity of the instructions.

A basic tenet of current computer architecture is to make the frequent case fast, and this often means making it simple. Since the assignment statement happens so frequently, we should concentrate on making it fast (and simple, as a consequence). One way to simplify assignments is to force all communication with memory into just two commands: `LOAD` and `STORE`. The `LOAD/STORE` model is typical of RISC architectures. We saw the `LOAD/STORE` concept in Chapter 4 with the `ld` and `st` instructions for the ARC.

By restricting memory accesses to `LOAD/STORE` instructions only, other instructions can only access data that is stored in registers. There are two consequences of this, both good and bad: (1) accesses to memory can be easily overlapped, since there are no side effects that would occur if different instruction types could access memory (this is good); and (2) there is a need for a large number of registers (this seems bad, but read on).

A simpler instruction set results in a simpler and typically smaller CPU, which frees up space on a circuit board (or a processor chip) to be used for something else, like registers. Thus, the need for more registers is balanced to a degree by the newly vacant circuit area, or **real estate** as it is sometimes called. A key problem lies in how to manage these registers, which is described in the next section.

9.3 Overlapping Register Windows

Procedure calls may be deeply nested in an ordinary program, but for a given window of time, the nesting depth fluctuates within a narrow band. Figure 9-3 illustrates this behavior. For a nesting depth window size of five, the window moves only 18 times for 100 procedure calls. Results produced by a group at UC Berkeley (Tamir and Sequin, 1983) show that a window size of eight will shift on less than 1% of the calls or returns.

The small window size for nested calls is important for improving performance. For each procedure call, a stack frame is normally constructed in which parameters, a return address, and local variables are placed. There is thus a great deal of stack manipulation that takes place for procedure calls, but the complexity of the manipulation is not all that great. That is, stack references are highly localized within a small area.

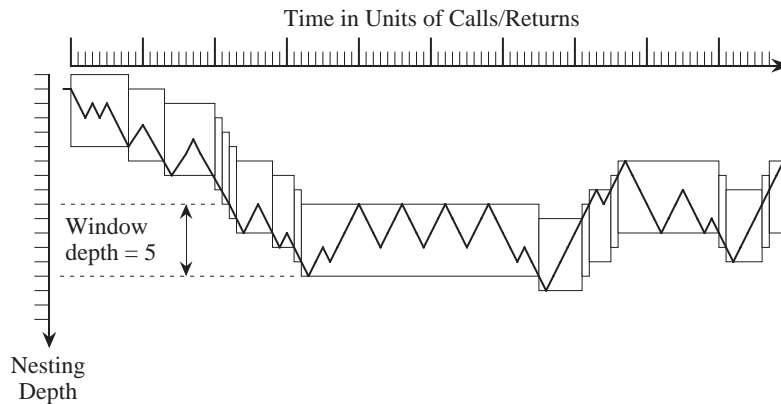


Figure 9-3 Call-return behavior as a function of nesting depth and time (Adapted from [Stallings, 19?? (2nd ed.)]).

The RISC I architecture exploits this locality by keeping the active portion of the stack in registers. Figure 9-4 shows the user's view of register usage for the RISC

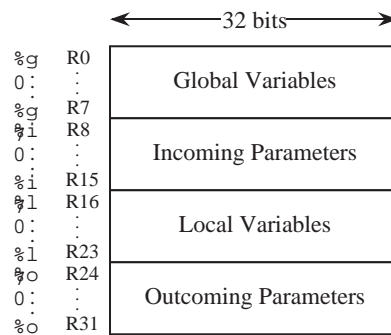


Figure 9-4 User's view of RISC I registers.

I. The user sees 32 registers in which each register is 32 bits wide. Registers R0-R7 are used for global variables. Registers R8-R15 are used for incoming parameters. Registers R16-R23 are used for local variables, and registers R24-R31 are used for outgoing parameters. The eight registers within each group are enough to satisfy the bulk of call/return activity, as evidenced by the frequency counts in Figure 9-3.

Although the user sees 32 registers, there may be several hundred registers that overlap. Figure 9-5 shows a model known as **overlapping register windows**. The global registers are detached from the others, and are continuously available as

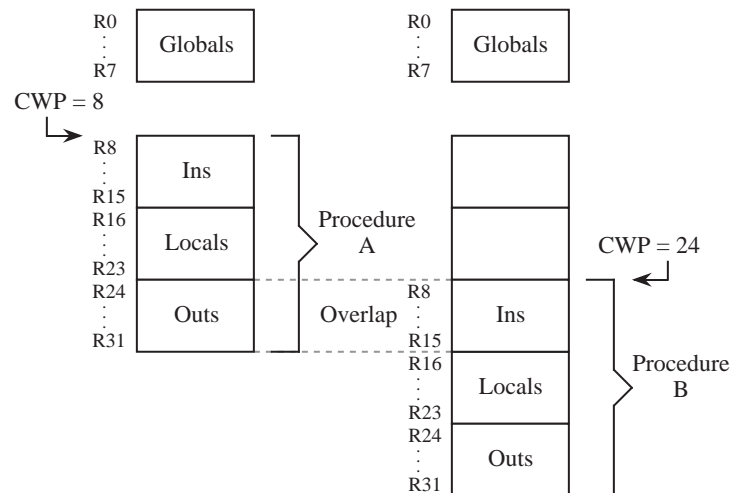


Figure 9-5 Overlapping register windows.

R0-R7. Registers R8-R31 make up the remaining 24 registers that the user sees, but this group of registers slides deeper into the **register file** (the entire set of registers) on each procedure call. Since the outgoing parameters for one procedure are the incoming parameters to another, these sets of registers can overlap. Registers R8-R31 are referred to as a **window**. A **current window pointer (CWP)** points to the current window, and increases or decreases for calls and returns, respectively.

In the statistically rare event when there are not enough registers for the level of nesting, then main memory is used. However, main memory is used for the *lowest* numbered window, so that the new current window still uses registers. The highest register location then wraps around to the lowest, forming a **circular buffer**. As returns are made, registers that were written to memory are restored to the register file. Thus, execution always takes place with registers and never directly with main memory.

9.4 Pipelining the Datapath

There are four phases of operation in the fetch-execute cycle: instruction fetch, decode, operand fetch, and execute. Each ARC instruction in our model therefore requires approximately four machine cycles to complete execution (this is not true for most commercial SPARC implementations, which have fewer phases.) We can view these four phases as taking place sequentially, as illustrated in Figure 9-6.

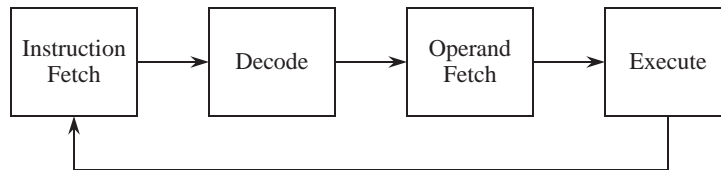


Figure 9-6 Four-stage instruction pipeline.

Each of the four units performs a different operation in the fetch-execute cycle. After the Instruction Fetch unit finishes its task, control is handed off to the Decode unit. At this point, the Instruction Fetch unit can begin fetching the *next* instruction, which overlaps with the decoding of the previous instruction. When the Instruction Fetch and Decode units complete their tasks, they hand off the remaining tasks to the next units (Operand Fetch is the next unit for Decode). The flow of control continues until all units are filled. This model of overlapped operation is referred to as **pipelining**.

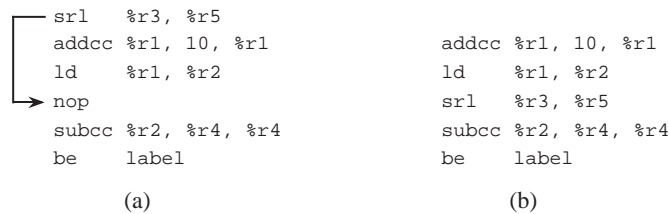
Although it takes four steps to execute an instruction in our ARC model, on average, one instruction can be executed per cycle as long as the pipeline stays filled. The pipeline does not stay filled, however, unless we are careful as to how instructions are ordered. We know from Figure 9-1 that approximately one in every four instructions is a branch. We cannot fetch the instruction that follows a branch until the branch completes execution. Thus, as soon as the pipeline fills, a branch is encountered, and then the pipeline has to be **flushed** by filling it with no-operations (NOPs). A similar situation arises with a Load or a Store instruction, which requires more than one cycle. The “wait” cycles are filled with NOPs.

Figure 9-7 illustrates the pipeline behavior during a memory reference and a branch for the ARC instruction set. The `addcc` instruction enters the pipeline on time step (cycle) 1. On cycle 2, the `ld` instruction enters the pipeline and `addcc` moves to the Decode stage. The pipeline continues filling with the `srl` and `subcc` instructions on cycles 3 and 4, respectively. On cycle 4, the `addcc` instruction is executed and leaves the pipeline. On cycle 5, the `ld` instruction reaches the Execute level, but does not finish execution because an additional cycle is needed for the memory reference. The `ld` instruction finishes execution during cycle 6.

The `ld` and `st` instructions both require five cycles, but the remaining instructions require only four. Thus, an instruction that follows an `ld` or `st` should not use the register that is being loaded or stored. A safe approach is to insert a NOP after an `ld` or an `st` as shown in Figure 9-8a. The extra cycle (or cycles, depend-

	Time							
	1	2	3	4	5	6	7	8
Instruction Fetch	addcc	ld	srl	subcc	be	nop	nop	nop
Decode		addcc	ld	srl	subcc	be	nop	nop
Operand Fetch			addcc	ld	srl	subcc	be	nop
Execute				addcc	ld	srl	subcc	be
Memory Reference						ld		

Figure 9-7 Pipeline behavior during a memory reference and a branch.

Figure 9-8 SPARC code, (a) with a `nop` inserted, and (b) with `srl` migrated to `nop` position.

ing on the architecture) for a load is known as a **delayed load**, since the data from the load is not immediately available on the next cycle. A **delayed branch** is similar, as shown for the `be` instruction in cycles 5 through 8 of Figure 9-7.

The `nop` instruction wastes a cycle as the processor waits for a Load or a Store to complete, or as the processor waits for the pipeline to be flushed. If we look at the code that surrounds a Load, Store, or Branch instruction, there is usually an instruction nearby that can replace the `nop`. In Figure 9-8a, the `srl` instruction can be moved to the position of the `nop` since its register usage does not conflict with the surrounding code. After replacing the `nop` line with the `srl` line, the code shown in Figure 9-8b is obtained. This is the code that is traced through the pipeline in Figure 9-7.

An alternative approach is to simply guess which way the branch will go, and then undo any damage if the wrong path is taken. Statistically, loops are executed more often than not, and so it is usually a good guess to assume that a branch out

of a loop will not be taken. Thus, a processor can start processing the next instruction in anticipation of the direction of the branch. If the branch goes the wrong way, then the execution phase for the next instruction, and any subsequent instructions that enter the pipeline, can be stopped so that the pipeline can be flushed. This approach works well for a number of architectures, particularly those with slow cycle speeds or deep pipelines. For RISCs, however, the overhead of determining when a branch goes the wrong way and then cleaning up any side effects caused by wrong instructions entering the pipeline is generally too great. The `nop` instruction is normally used in RISC pipelines when something useful cannot be used to replace it.

9.5 Multiple Instruction Issue Machines

[Placeholder for future section.]

9.6 VLIW Machines

[Placeholder for future section. Discussion of the Intel Merced architecture.]

EXAMPLE: PLACEHOLDER

[Placeholder for future section.]

9.7 Case Study: Inspecting Compiled Code

[Note from authors: This section is not finished.]

In this section, we analyze a C compiler produced SPARC assembly program. We start with the C program shown in Figure 9-9, in which the main routine passes two integers to a subroutine, which returns the sum of the integers to the main routine. The code produced by a Solaris Unix C compiler using the command line:

```
gcc -S file.c
```

is shown in Figure 9-10.

A line by line explanation of the assembled code is given in Figure 9-10. There are a number of new instructions and pseudo-ops introduced in this code:

```

/* Example C program to be compiled with gcc
*/
#include
<stdio.h>
main ()
{
    int a, b, c;

    a =
10; b =
4; c = add_two(a,
b);
    printf("c = %d\n",
c);

    int
add_two(a,b)
{
    int
result;
    result = a +
b; return(result);
}

```

Figure 9-9 Source code for C program to be compiled with gcc.

`.seg/.section` Unix executable programs have three segments for data, text (the instructions), and the stack. The `.seg` pseudo-op instructs the assembler to place the code that follows into one of these three segments. Some of the segments have different protections, which is why there is a data segment and also a `data1` segment. The `data1` segment contains constants, and should be protected from writing. The data segment is both readable and writable and is therefore not protected against reading or writing (but it is protected from being executed, as is data). Newer versions of Unix allow more text and data areas to be used for different read, write, and execute protections.

`.proc` *[Placeholder for unwritten text. – Au]*

`%hi` Same as `.high22`.

`%lo` Same as `.low10`.

`add` Same as `addcc` except that the condition codes are unaffected.

```

! Output produced by gcc compiler on Solaris (Sun UNIX)
! Annotations added by author

.file    add.c    ! Identifies the source program
.section ".rodata"    ! Read-only data for routine main
.align 8    ! Align read-only data for routine main on an
              ! 8-byte boundary
.LLC0
    .asciz "c = %d\n"    ! This is the read-only data
.section      "text"    ! Executable code starts here
.align 4    ! Align executable code on a 4-byte (word) boundary
.global main
.type    main,#function
.proc    04
main:    ! Beginning of executable code for routine main
    !#PROLOGUE#    0
    save %sp, -128, %sp    ! Create 128 byte stack frame. Advance
                          ! CWP (Current Window Pointer)

    !#PROLOGUE#    1
    mov 10, %o0    ! %o0 <- 10. Note that %o0 is the same as %r24.
    ! This is local variable a in main routine of C source program.
    st %o0, [%fp-20]    ! Store %o0 five words into stack frame.
    mov 4, %o0    ! %o0 <- 4. This is local variable b in main.
    st %o0, [%fp-24]    ! Store %o0 six words into stack frame.
    ld [%fp-20], %o0    ! Load %o0 and %o1 with parameters to
    ld [%fp-24], %o1    ! be passed to routine add_two.
    call add_two, 0    ! Call routine add_two
    nop    ! Pipeline flush needed after a transfer of control
    st %o0, [%fp-28]    ! Store result 67 words into stack frame.
                          ! This is local variable c in main.
    sethi %hi(.LLC0), %o1 ! This instruction and the next load
    or %o1, %lo(.LLC0), %o0 ! the 32-bit address .LLC0 into %o0
    ld [%fp-28], %o1    ! Load %o1 with parameter to pass to printf

```

Figure 9-10 gcc generated SPARC code (continued on next page).

save Advances current window pointer and increments stack pointer to create space for local variables.

mov Same as:

or %g0,register_or_immediate,destination_register. This differs from **st** because the destination is a register.

nop No-operation (the processor waits for one instruction cycle, while the branch finishes).

.ascii/.asciz Reserves space for an ASCII string.

```

        call printf, 0
        nop      ! A nop is needed here because of the pipeline flush
                  ! that follows a transfer of control.
.LL1
        ret      ! Return to calling routine (Solaris for this case)
        restore ! The complement to save. Although it follows the
                  ! return, it is still in the pipeline and gets executed.
.LLfe1
        .size    main, .LLfe1-main ! Size of
        .align 4
        .global add_two
        .type    add_two, #function
        .proc 04
add_two:
        !#PROLOGUE# 0
        save %sp, -120, %sp
        !#PROLOGUE# 1
        st %i0, [%fp+68] !Same as %o0 in calling routine (variable a)
        st %i1, [%fp+72] !Same as %o1 in calling routine (variable b)
        ld [%fp+68], %o0
        ld [%fp+72], %o1
        add %o0, %o1, %o0 ! Perform the addition
        st %o0, [%fp-20] ! Store result in stack frame
        ld [%fp-20], %i0 ! %i0 (result) is %o0 in called routine
        b .LL2
        nop
.LL2:
        ret
        restore
.LLfe2:
        .size    add_two, .LLfe2-add_two
        .ident   "GCC: (GNU) 2.5.8"

```

Figure 9-10 (cont')

set Sets a register to a value. This is a macro that expands to the `sethi, %hi,` and `%lo` constructs shown in `#PROLOGUE# 1`.

ret Return. Same as: `jmp1 %i7+8, %g0`.

restore Decrements current window pointer.

b Same as `ba`.

.file Identifies the source file.

.align Forces the code that follows onto a boundary evenly divisible by its

argument.

`.type` Associates a label with its type.

`.size` Computes the size of a segment.

`.ident` Identifies the compiler version.

Notice that the compiler does not seem to be consistent with its choice of registers for parameter passing. Prior to the call to `add_two`, the compiler uses `%o0` and `%o1` (`%r24` and `%r25`) for parameters passed to `add_two`. Then, `%r25` is used for the parameters passed to `printf`. Why did the compiler not start with `%r24` again, or choose the next available register (`%o2`)? This is the register assignment problem, which has been the object of a great deal of study. We will not go into details here¹, as this is more appropriate for a course in compiler design, but suffice it to say that any logically correct assignment of variables to registers will work, but that some assignments are better than others in terms of the number of registers used and the overall program execution time.

Why are the stack frames so large? We only need three words on the stack frame for local variables `a`, `b`, and `c` in `main`. We might also need a word to store the return address, although the compiler does not seem to generate code for that. There are no parameters passed to `main` by the operating system, and so the stack frame that `main` sees should only be four words (16 bytes) in size. Thus, the line at the beginning of routine `main`:

```
save %sp, -128, %sp
```

should only be:

```
save %sp, -16, %sp.
```

What is all of the extra space for? There are a number of runtime situations that may need stack space. For instance, if the nesting depth is greater than the num-

1. Here are a few details, for the curious: `%r0` (`%o0`) is still in use (`add_two` is expecting the address of `LLC0` to show up in `%r0`), and `%r1` is no longer needed at this point, so it can be reassigned. But then, why is `%r1` used in the `sethi` line? Would it have made sense to use `%r0` instead of introducing another register into the computation? See problem 9.2 at the end of the chapter for more on this topic.

ber of windows, then the stack must be used for overflow. (See Figure D-2 in [SPARC, 1992])

If a scalar is passed from one routine to another, then everything is fine. But if a callee refers to the address of a passed scalar (or aggregate), then the scalar (or aggregate) must be copied to the stack and be referenced from there for the lifetime of the pointer (or for the lifetime of the procedure, if the pointer lifetime is not known).

Why does the return statement `ret` cause a return to the code that is 8 bytes past the `call`, instead of 4 as we have been doing it? This is because there is a `nop` that follows `call` (the so-called “delay-slot instruction”).

Notice that routine labels that appear in the source code are prepended with an underscore in the assembly code, so that `main`, `add_two`, and `printf` in C become `_main`, `_add_two`, and `_printf` in `gcc` generated SPARC code. This means that if we want to write a C program that is linked to a `gcc` generated SPARC program, that the C calls should be made to routines that begin with underscores. For example, if `add_two` is compiled into SPARC code, and we invoke it from a C main program in another file, then the C program should make a call to `_add_two`, and *not* `add_two`, even though the routine started out as `add_two`. Further, the C program needs to declare `_add_two` as external.

If the compilation for `add_two` is continued down to an executable file, then there is no need to treat the labels differently. The `add_two` routine will still be labeled `_add_two`, but routine `main` will be compiled into code that expects to see `_add_two` and so everything will work OK. This is not the case, however, if a `gcc` program makes calls to a Fortran library.

Fortran is a commonly used language in the scientific community, and there are a number of significant Fortran libraries that are used for linear algebra (LINPACK), modeling and simulation (___), and parallel scientific applications (___). As C programmers, we sometimes find ourselves wanting to write C programs that make calls to Fortran routines. This is easy to do once we understand what is happening.

There are two significant differences that need to be addressed:

- (1) differences in routine labels;

(2) differences in subroutine linkage.

In Fortran, the source code labels are prepended with two underscores in the assembly code. A C program that makes a call to Fortran routine `add_two` would then make a call to `__add_two`, which also must be declared as external in the C source code (and declared as global in the Fortran program).

If all of the parameters that are passed to the Fortran routines are pointers, then everything will work OK. If there are any scalars passed, then there will be trouble because C uses call-by-value for scalars whereas Fortran uses call-by-reference. We need to “trick” the C compiler into using call-by-reference by making it explicit. Wherever a Fortran routine expects a scalar in its argument list, we use a pointer to the scalar in the C code. As an example, a C/Fortran version of the `add_two` code is shown below:

[Placeholder for unwritten C/Fortran figure.]

As a practical consideration, the `gcc` compiler will compile Fortran programs. It knows what to do by observing the extension of the source file, which should be `.f` for Fortran. *[The rest of this section is unfinished. – Au(s)]*

[Note to Au: Manipulation of `%sp` needs to be atomic. See page 191 of SPARC Architecture manual, under first bullet.]

9.8 Case Study: Superscalar Assembly Language Programming on the Intel Pentium II with MMX Technology

Discussion of the Intel Merced architecture.

EXAMPLE

A processor has a five stage pipeline. If a branch is taken, then four cycles are needed to flush the pipeline. The branch penalty b is thus 4. The probability P_b that a particular instruction is a branch is .25. The probability P_t that the branch is taken is .5. We would like to compute the average number of cycles needed to execute an instruction, and the **execution efficiency**.

When the pipeline is filled and there are no branches, then the average number of cycles per instruction (CPI_{No_Branch}) is 1. The average number of cycles per instruction when there are branches is then:

$$\begin{aligned} CPI_{Avg} &= (1 - P_b)(CPI_{No_Branch}) + P_b[P_t(1 + b) + (1 - P_t)(CPI_{No_Branch})] \\ &= 1 + bP_bP_t \end{aligned}$$

After making substitutions, we have:

$$\begin{aligned} CPI_{Avg} &= (1 - .25)(1) + .25[.5(1 + 4) + (1 - .5)(1)] \\ &= 1.5 \text{ cycles.} \end{aligned}$$

The execution efficiency is the ratio of the cycles per instruction when there are no branches to the cycles per instruction when there are branches. Thus we have:

$$\text{Execution efficiency} = (CPI_{No_Branch}) / (CPI_{Avg}) = 1 / 1.5 = 67\%.$$

The processor runs at 67% of its potential speed as a result of branches, but this is still much better than the five cycles per instruction that might be needed without pipelining.

There are techniques for improving the efficiency. We know that loops are usually executed more than once, so we can guess that a branch out of the loop will not be taken and be right most of the time. We can also run simulations on the non-loop branches, and get a statistical sampling of which branches are likely to be taken, and then guess the branches accordingly. As explained earlier, this approach works best when the pipeline is deep or the clock rate is slow. ■

■ SUMMARY

In the RISC approach, the most frequently occurring instructions are optimized by eliminating or reducing the complexity of other instructions and addressing modes commonly found in CISC architectures. The performance of RISC architectures is further enhanced by pipelining and increasing the number of registers available to the CPU.

■ FURTHER READING

The three characteristics of RISC architectures originated at IBM's T. J. Watson Research Center, as summarized in (Ralston and Reilly, 1993, pp. 1165 - 1167). (Hennessy and Patterson, 1990) is the seminal reference on much of the work that led to the RISC concept, although the word "RISC" does not appear in the title of their textbook. (Stallings, 1990) is a thorough reference on RISCs. (Tamir and Sequin, 1983) show that a window size of eight will shift on less than 1% of the calls or returns. (Tanenbaum, 1990) provides a readable introduction to the RISC concept.

Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, California, (1990).

Ralston, A. and E. D. Reilly, eds., *Encyclopedia of Computer Science*, 3/e, van Nostrand Reinhold, (1993).

Stallings, W., *Reduced Instruction Set Computers*, 3/e, IEEE Computer Society Press, Washington, D.C., (1991).

Tamir, Y., and C. Sequin, "Strategies for Managing the Register File in RISC," *IEEE Trans. Comp.*, (Nov. 1983).

Tanenbaum, A., *Structured Computer Organization*, 3/e, Prentice Hall, Englewood Cliffs, New Jersey, (1990).

■ PROBLEMS

9.1 Increasing the number of cycles per instruction can sometimes improve the execution efficiency of a pipeline. If the time per cycle for the pipeline described in Section 5.6.3 is 20 ns, then CPI_{Avg} is $1.5 \times 20 \text{ ns} = 30 \text{ ns}$. Compute the execution efficiency for the same pipeline in which the pipeline depth increases from 5 to 6 and the cycle time decreases from 20 ns to 10 ns.

9.2 The SPARC code below is taken from the gcc generated code in Figure 9-10. Can `%r0` be used in all three lines, instead of "wasting" `%r1` in the second line?

```

    ...
    st    %r0, [%fp-28]

```

```
sethi %hi(.LLC0), %o1  
or    %o1, %lo(.LLC0), %o1  
...
```