

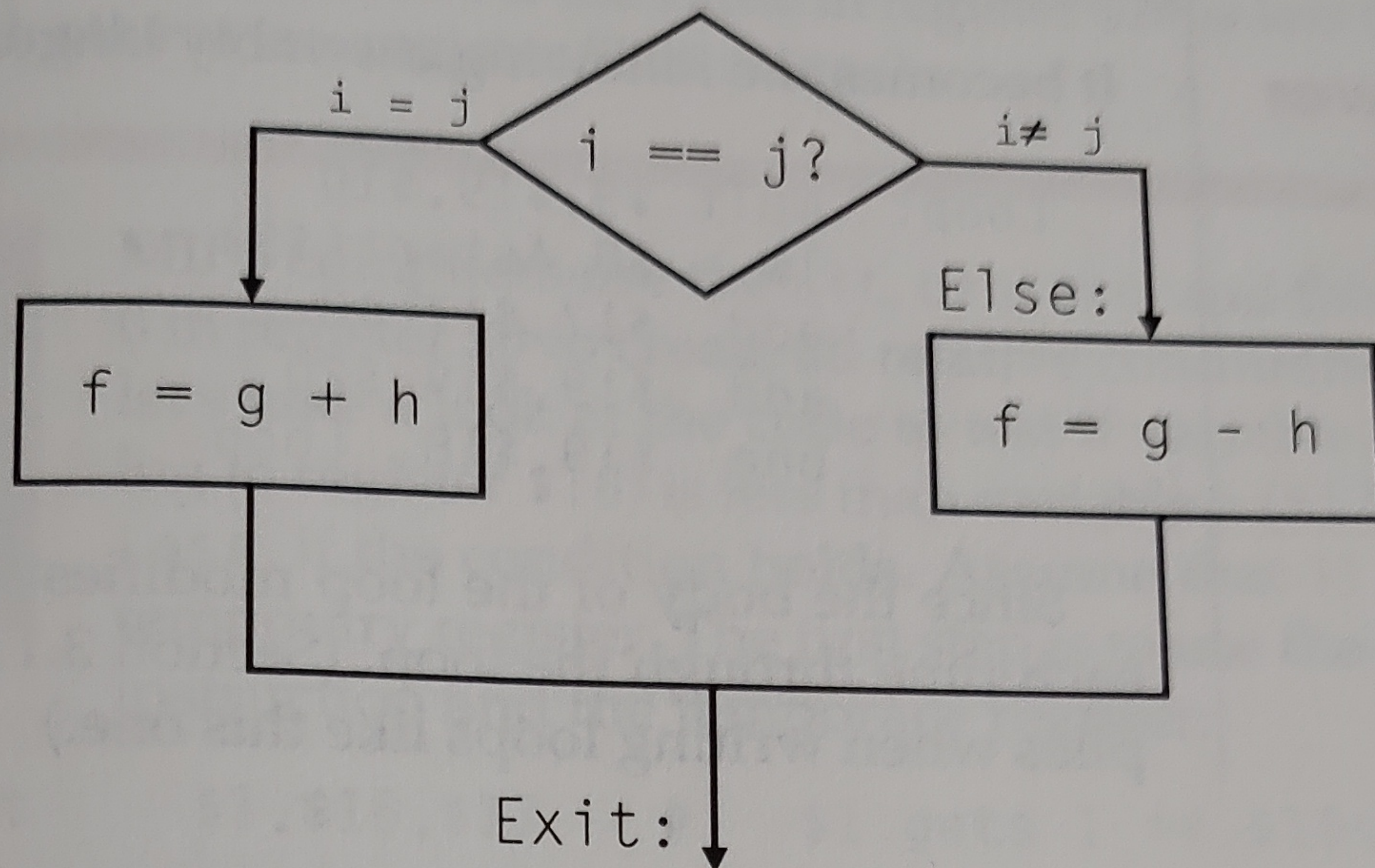


FIGURE 3.9 Illustration of the options in the if statement above. The left box corresponds to the “then” part of the *if* statement and the right box corresponds to the “else” part.

Example

Here is a loop in C:

```

Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
  
```

Assume *A* is an array of 100 elements and that the compiler associates the variables *g*, *h*, *i*, and *j* to the registers \$17, \$18, \$19, and \$20, respectively. Let's assume that the array starts at *Astart*. Remember that byte addressing means we will need to multiply the index *i* by 4 before we can use it in the load instruction. Let's assume that 4 has already been placed in register \$10 and that the MIPS instruction `mult` (further described in Chapter 4), is available for multiplies. What is the MIPS assembly code corresponding to this C segment?

Elaboration: Readers who might have heard about delayed branches, covered in Chapter 6, should not worry: The MIPS assembler makes them invisible to the assembly language programmer.

3.6

Supporting Procedures in Computer Hardware

A procedure or subroutine is one way that programmers structure programs, both to make them easier to understand and to allow code to be reused. An instruction set must provide a way to jump to a procedure and then return from the procedure to the instruction just after the calling point. Programmers must also have conventions governing how to pass parameters and how to support the nesting of procedure calls.

First, let's look at the primitive operations that support procedures. MIPS provides an instruction that jumps to an address and simultaneously saves the address of the following instruction in register \$31. The *jump-and-link* instruction (`jal`) is simply written

```
jal ProcedureAddress
```

The *link* portion of the name means that a link is formed to the calling site to allow the procedure to return to the proper address. This link, stored in register \$31, is called the *return address*.

We already have an instruction to do the return jump:

```
jr $31
```

The Jump register instruction, useful in the *switch* statement, jumps to the address stored in register \$31—which is just what we want.

Implicit in the instruction set is the need to have a register to hold the address of the current instruction being executed. The `jal` instruction just increments this register to point to the next instruction before saving it in register \$31, and `jr $31` simply copies \$31 into that register. For historical reasons, this register is almost always called the *program counter*, abbreviated *PC*, although a more sensible name would have been *Instruction Address register*. MIPS follows tradition and calls it the *PC*.

Suppose a procedure wanted to call another procedure. Then the programmer would need to save the old value of register \$31, since the new `jal` would clobber the old return address. This is an example of where we need to spill registers to memory, as mentioned in the Hardware Software Interface on page 102.

Since procedures can call procedures that call procedures, and so on, the ideal data structure for spilling registers is a *stack*—a last-in-first-out queue. A

stack needs a pointer to the top of the stack to show where the next procedure should place the registers to be spilled or where old register values can be found. The top of the stack is adjusted by the number of registers that are saved or restored. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: Placing data onto the stack is called a *push*, and removing data from the stack is called a *pop*.

Example

Assume procedure A has invoked procedure B, that procedure B is about to call procedure C, and that C calls no more procedures. Figure 3.12 on page 122 shows the steps we must perform. Before calling C, procedure B must save its return address on the stack. The stack pointer, register \$29, is adjusted to point to the new top of stack. Procedure C is then called, and the *jal* instruction changes register \$31 to contain C's return address. After procedure C returns to procedure B, the old return address is restored from the stack into register \$31. The stack pointer is then changed back to the old top-of-stack location. What is the essential MIPS code to implement this calling protocol?

Answer

Below is the basic MIPS assembly code segment. Assume that register \$29 contains the pointer to the top of the stack and one register, say \$24, already has the value to adjust the top of stack. Recall that the MIPS assembly convention is to list labels on the left, indent the instruction to the right of the labels, and follow these with an optional comment that starts with the sharp symbol (#).

```

A:...
...
    jal    B                # call B & save return address in $31
...
B:...
...
    add    $29,$29,$24      # now ready to call C
                                # adjust stack to make room
                                # for next item
    sw     $31, 0($29)      # save the return address
    jal    C                # call C & save return address in
                                # $31; return from C to next instr
    lw     $31, 0($29)      # restore B's return address...
    sub    $29,$29,$24      # adjust stack to pop
                                # B's return address

```

of these factors without significantly affecting other critical factors in the memory hierarchy.

7.3

Virtual Memory

... a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.

Kilburn et al., *One-level storage system*, 1962

In the previous section, we saw how caches served as a method for providing fast access to recently used portions of a program's code and data. Similarly, the main memory can act as a "cache" for the secondary storage, usually implemented with magnetic disks. This technique is called *virtual memory*. There are two major motivations for virtual memory: to allow efficient sharing of memory among multiple programs and to remove the programming burdens of a small, limited amount of main memory.

Consider a collection of programs running at once on a machine. The total memory required by all the programs may be much larger than the amount of physical memory available on the machine, but only a fraction of this memory is actively being used at any point in time. Main memory need contain only the active portions of the programs, just as a cache contains only the active portion of one program. This allows us to efficiently share the processor as well as the main memory.

A second motivation is to allow user programs to exceed the size of primary memory. Formerly, if a program became too large for physical memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These *overlays* were loaded or unloaded under user program control during execution, with the programmer ensuring that the program never tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. As one can well imagine, this responsibility was a substantial burden on programmers. *Virtual memory*, which was invented to relieve programmers of this difficulty, automatically manages the two levels of the memory hierarchy represented by main memory and secondary storage.

Of course, we cannot know which programs will share the physical memory with other programs when we compile them. In fact, the programs sharing the physical memory can even change dynamically while the programs are running. Because of this, we would like to compile each program into its own *address space*, that is, a separate range of memory locations accessible only to this program. Because multiple user programs share a single physical memo-

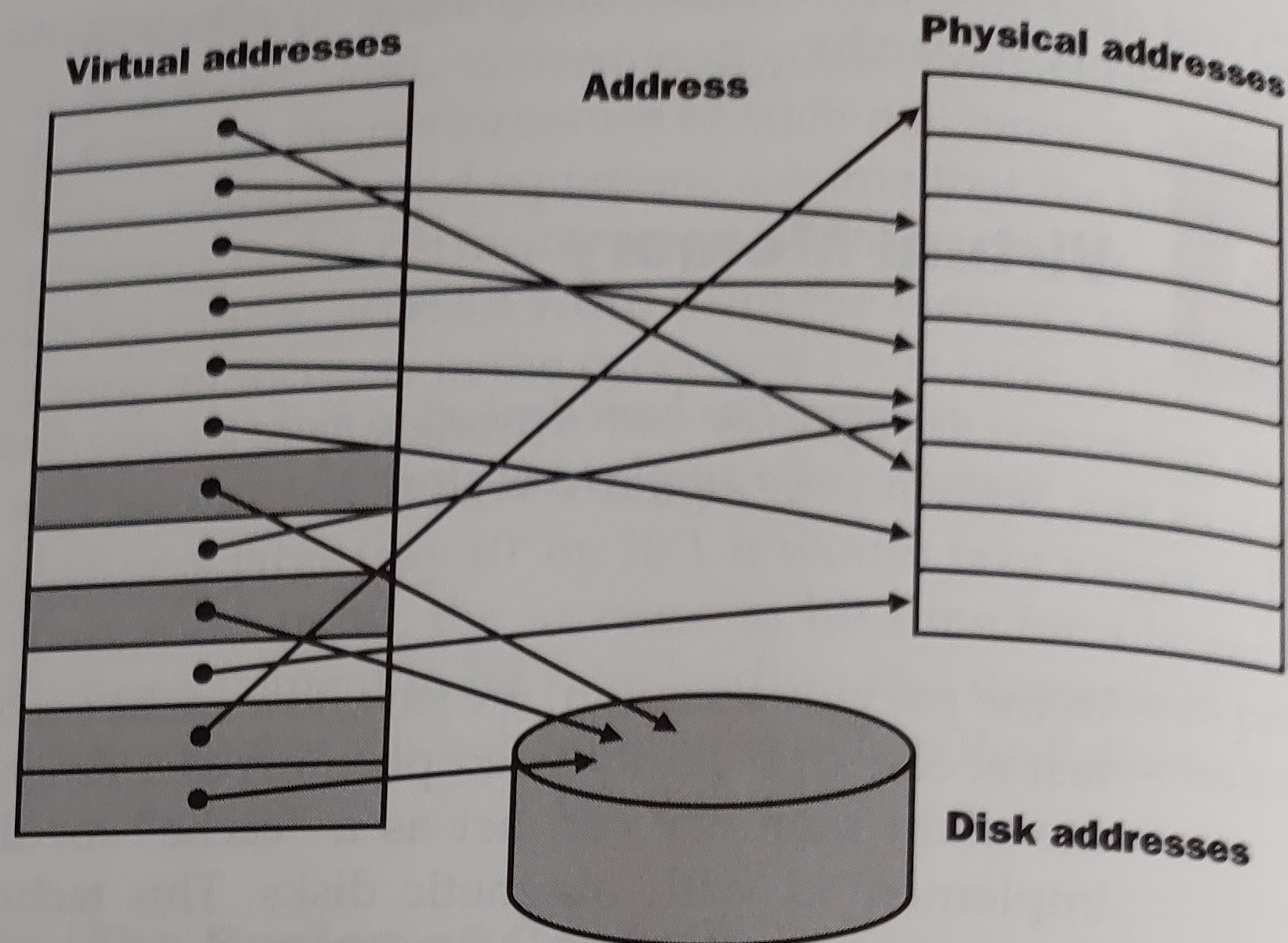


FIGURE 7.14 In virtual memory, pages are mapped from one set of addresses, called **virtual addresses**, to another set called **physical addresses**. The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is really mapped to a physical page. Of course, it is also possible for a virtual page to be absent from physical memory and not be mapped to a physical address as soon, residing instead on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.

ry, which the operating system must also share, we must be able to protect the programs from one another. Both the translation of each program's address space and the protection of the address space from other programs are provided by virtual memory.

Although the concepts at work in virtual memory and in caches are the same, their differing historical roots have led to the use of different terminology. A virtual memory block is called a *page*, and a virtual memory miss is called a *page fault*. With virtual memory, the CPU produces a *virtual address*, which is translated by a combination of hardware and software to a *physical address*, which in turn can be used to access main memory. Figure 7.14 shows the virtual addressed memory with pages mapped to physical memory. This process is called *memory mapping* or *address translation*. Today, the two memory hierarchy levels controlled by virtual memory are DRAMs and magnetic disks. If we return to our library analogy, we can think of a virtual address as the title of a book and a physical address as the location of that book in the library (which might be given by the Library of Congress number).