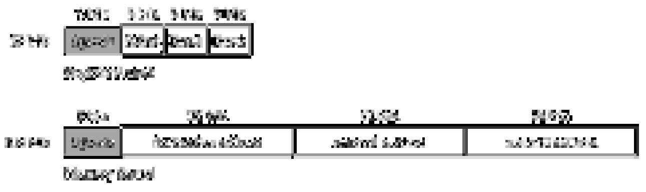
**LECTURE 3**

## 2.3.6 Processor Registers

Processors have a number of registers to hold data, instructions, and state information. We can classify the processors based on the structure of these registers and how the processor uses them. The registers can be divided into general-purpose or special-purpose registers. Special-purpose registers can be further divided into those that are accessible to the user programs and those reserved for the system use. The available technology largely determines the structure and function of the register set. The number of addresses used in instructions partly influences the number of data registers and their use. For example, in three- and two-address machines, there is no need for the internal data registers. However, having a few internal registers improves performance by cutting down the number of memory accesses required to execute a program. RISC processors typically have a large number of registers.
Some processors maintain a few special-purpose registers. For example, the Pentium uses a couple of registers to implement the processor stack. Processors also have several registers reserved for the instruction execution unit. Typically, there is an instruction register that holds the current instruction and a program counter that points to the next instruction to be executed.



[illegible]

## 2.4 Flow of Control

Program execution, by default, proceeds sequentially. The program counter (PC) register plays an important role in managing the control flow. At a simple level, the PC can be thought of as pointing to the next instruction. The processor fetches the instruction at the address pointed to by the PC. When an instruction is fetched, the PC is incremented to point to the next instruction. If we assume that each instruction takes exactly four bytes as in the MIPS processors, the PC is automatically incremented by four after each instruction fetch. This leads to the default sequential execution pattern. However, sometimes we want to alter this default execution flow. In high-level languages, we use control structures such as if-then-else and while statements to alter the execution behavior based on some run-time conditions.

### 2.4.1 Branching

Branching is implemented by means of a branch instruction. This instruction carries the address of the target instruction explicitly. Branch instructions in processors such as the Pentium are also called the jump instructions. Processors support two types of branches: unconditional and conditional. In both cases, the transfer control mechanism remains the same (see Figure 2.6).

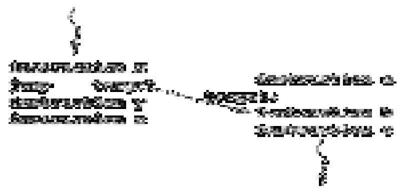


Figure 2.6 [illegible]

**Unconditional Branch**

The simplest of the branch instructions is the unconditional branch, which transfers control to the specified target. Here is an example branch instruction:

branch target

Specification of the target address can be done in one of two ways: absolute address or PC relative address. In the former, the actual address of the target instruction is given. In the PC-relative method, the target address is specified relative to the PC contents. Most processors support absolute address for unconditional branches. Others support both formats. For example, MIPS processors support absolute address-based branch by

j target

and PC-relative unconditional branch by

b target

In fact, the last instruction is an assembly language instruction. The processor only supports the j instruction. If the absolute address is used, the processor transfers control by simply loading the specified target address into the PC register. If PC-relative addressing is used, the specified target address is added to the PC contents, and the result is placed in the PC. In either case, since the PC indicates the next instruction address, the processor will fetch the instruction at the intended target address. The main advantage of using the PC-relative address is that we can move the code from one block of memory to another without changing the target addresses. This type of code is called relocatable code. Relocatable code is not possible with absolute addresses.

**Conditional Branch**

In conditional branches, the jump is taken only if a specified condition is satisfied. For example, we may want to take a branch only if two values are equal. Such conditional branches are handled in one of two basic ways:

• Set-Then-Jump: In this design, testing for the condition and branching are separated. To achieve communication between these two instructions, a condition code register is used. The Pentium follows this design, which uses a flags register to record the result of the test condition. It uses a compare (cmp) instruction to test the condition. This instruction sets the various flag bits to indicate the relationship between the two compared values. Then we can use a conditional jump instruction to jump to the target location if the specified condition bit is set.

• Test-and-Jump: Most processors combine the testing and branching into a single instruction. We use the MIPS processor to illustrate the principle involved in this strategy. The MIPS processor provides several branch instructions that test and branch. The one that we are interested in here is the branch on equal instruction shown below:

beq Rsrc1,Rsrc2,target

This conditional branch instruction tests the contents of the two registers Rsrc1 and Rsrc2 for equality and transfers control to target if equal.

Some processors maintain registers to record the condition of the arithmetic and logical operations. These are called condition code registers. These registers keep a record of the status of the last arithmetic/logical operation. For example, when we add two 32-bit integers, it is possible that the sum might require more than 32 bits. This is the overflow condition that the system should record. Normally, a bit in the condition code register is set to indicate this overflow condition. The MIPS processors, for example, do not use condition registers. Instead, it uses exceptions to flag the overflow condition. On the other hand, the Pentium uses condition registers, which are called the flags register. Some instruction sets provide branches based on comparisons to zero.

**2.4.2 Procedure Calls**

The use of procedures facilitates modular programming. Procedure calls are slightly different from the branches. Branches are one-way jumps: once the control has been transferred to the target location, computation proceeds from that location, as shown in Figure 2.6. In procedure calls, we have to return control to the calling program after executing the procedure. Control is returned to the instruction following the call instruction, as shown in Figure 2.7.

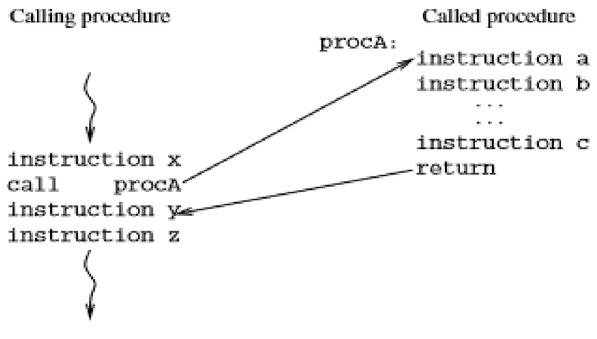


**Figure 2.7** Control flow in procedure calls.

From Figures 2.6 and 2.7, you will notice that the branches and procedure calls are similar in their initial control transfer. For procedure calls, we need to return to the

instruction following the procedure call. This return requires two pieces of information:
• End of Procedure: We have to indicate the end of the procedure so that the control can be returned. This is normally done by a special return instruction. For example, the Pentium uses ret and the MIPS uses the jr instruction to return from a procedure. We do the same in high-level languages as well. For example, in C, we use the return statement to indicate an end of procedure execution. High-level languages allow a default fall-through mechanism. That is, if we don't explicitly specify the end of a procedure, control is returned at the end of the block. In the assembly language, we must specify the end of a procedure by using the return instruction.
• Return Address: How does the processor know where to return after completing a procedure? This piece of information is normally stored when the procedure is called. Thus, when a procedure is called, it not only modifies the PC as in the branch instruction, but also stores the return address. Where does it store the return address? Two main places are used: a special register or the stack. Both MIPS and Pentium processors store the address of the instruction following the call instruction.
The Pentium uses the stack to store the return address. Thus, each procedure call involves pushing the return address onto the stack before control is transferred to the procedure code. The return instruction retrieves this value from the stack to send control back to the instruction following the procedure call.
MIPS processors allow any general-purpose register to store the return address. The return statement specifies this register. The format of the return statement is
jr $ra
where ra is the register that contains the return address.

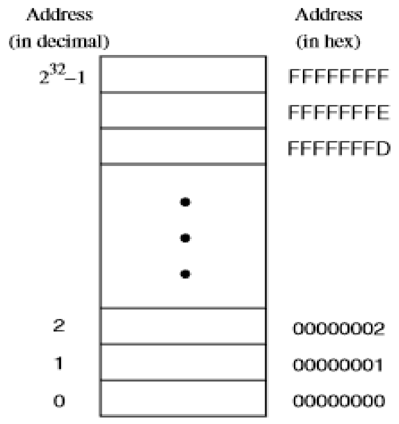


**Figure 2.8** Logical view of the system memory.

**Parameter Passing**

The general architecture dictates how parameters are passed on to the procedures. There are two basic techniques: register-based or stack-based. In the first method,

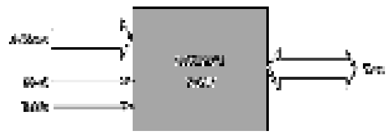
parameters are placed in the processor's internal registers and the called procedure will read the parameter values from these registers. In the stack-based method, parameters are pushed onto the stack and the called procedure would have to read them off the stack.
The advantage of the register method is that it is faster than the stack method. However, because of the limited number of registers, it imposes a limit on the number of parameters. Furthermore, recursive procedures cannot use the register-based mechanism. Because RISC processors tend to have more registers, register-based parameter passing is used in the MIPS processors. The Pentium, due to the small number of registers, tends to use the stack for parameter passing.

## 2.5 Memory

The memory of a computer system consists of tiny electronic switches, with each switch set in one of two states: open or closed. It is, however, more convenient to think of these states as 0 and 1 rather than open and closed. A single such switch can be used to represent two (i.e., binary) numbers: a zero and a one. Thus, each switch can represent a binary digit or bit, as it is known. The memory unit consists of millions of such bits. In order to make memory more manageable, bits are organized into groups of eight bits called bytes. Memory can then be viewed as consisting of an ordered sequence of bytes. Each byte in this memory can be identified by its sequence number starting with 0, as shown in Figure 2.8. This is referred to as the memory address of the byte. Such memory is called byte addressable memory. The Pentium can address up to 4 GB (232 bytes) of main memory (see Figure 2.8). This magic number comes from the fact that the address bus of the Pentium has 32 address lines. This number is referred to as the memory address space (MAS). The memory address space of a system is determined by the address bus width of the processor used in the system. Typically, 32-bit processors support 32-bit addresses.

### 2.5.1 Two Basic Memory Operations

The memory unit supports two fundamental operations: read and write. The read operation reads a previously stored data and the write operation stores a value in memory. Both of these operations require an address in memory from which to read a value or to which to write a value. In addition, the write operation requires specification of the data to be written. The block diagram of the memory unit is shown in Figure 2.9. The address and data of the memory unit are connected to the address and data buses, respectively. The read and write signals come from the control bus. Two metrics are used to characterize memory. Access time refers to the amount of time required by the memory to retrieve the data at the addressed location. The other metric is the memory cycle time, which refers to the minimum time between successive memory operations.


[illegible]

Memory transfer rates can be measured by the bandwidth metric. It specifies the number of bytes transferred per second. For example, a Pentium system with the PC133 memory can transfer 8 bytes at a frequency of 133 times per second. This gives us a bandwidth of 8 * 133 = 1064 MB/s. The read operation is non-destructive in the sense that one can read a location of the memory as many times as one wishes without destroying the contents of that location. The write operation, on the other hand, is destructive, as writing a value into a location destroys the old contents of that memory location.

Steps in a typical read cycle

1. Place the address of the location to be read on the address bus,
2. Activate the memory read control signal on the control bus,
3. Wait for the memory to retrieve the data from the addressed memory location and place it on the data bus,
4. Read the data from the data bus,
5. Drop the memory read control signal to terminate the read cycle.

A simple Pentium read cycle takes three clock cycles. During the first clock cycle, steps 1 and 2 are performed. The Pentium waits until the end of the second clock and reads the data and drops the read control signal. If the memory is slower (and therefore cannot supply data within the specified time), the memory unit indicates its inability to the processor and the processor waits longer for the memory to supply data by inserting wait cycles. Note that each wait cycle introduces a waiting period equal to one system clock period and thus slows down the system operation.

Steps in a typical write cycle

1. Place the address of the location to be written on the address bus,
2. Place the data to be written on the data bus,
3. Activate the memory write control signal on the control bus,
4. Wait for the memory to store the data at the addressed location,
5. Drop the memory write signal to terminate the write cycle.

As with the read cycle, the Pentium requires three clock cycles to perform a simple write operation. During the first clock cycle, steps 1 and 3 are done. Step 2 is performed during the second clock cycle. Pentium gives memory time until the end of the second clock and drops the memory write signal. If the memory cannot write data at the maximum processor rate, wait cycles can be introduced to extend the write cycle.

**2.5.2 Types of Memory**

The memory unit can be implemented using a variety of memory chips—different speeds, different manufacturing technologies, and different sizes. The two basic types of memory are the read-only memory and read/write memory. A basic property of memory systems is that they are random access memories in that accessing any memory location (for reading or writing) takes the same time. Contrast this with data stored on a magnetic tape. Access time on the tape depends on the location of the data. Volatility is another important property of a memory unit. A volatile memory requires power to retain its contents. A nonvolatile memory can retain its values even in the absence of power.

**Read-Only Memories**

Read-only memory (ROM) allows only read operations to be performed. As the name suggests, we cannot write into this memory. The main advantage of ROM is that it is nonvolatile. Most ROM is factory-programmed and cannot be altered. The term programming in this context refers to writing values into a ROM. This type of ROM is cheaper to manufacture in large quantities than other types of ROM. The program that controls the standard input and output functions (called BIOS), for instance, is kept in ROM. Current systems use the flash memory rather than a ROM. Other types of ROM include programmable ROM(PROM) and erasable PROM(EPROM). PROM is useful in situations where the contents of ROM are not yet fixed. For instance, when the program is still in the development stage, it is convenient for the designer to be able to program the ROM locally rather than at the time of manufacture.

In PROM, a fuse is associated with each bit cell. If the fuse is on, the bit cell supplies a 1 when read. The fuse has to be burned to read a 0 from that bit cell. When PROM is manufactured, its contents are all set to 1. To program PROM, selective fuses are burned (to introduce 0's) by sending high current. This is the writing process and is not reversible (i.e., a burned fuse cannot be restored). EPROM offers further flexibility during system prototyping. Contents of an EPROM can be erased by exposing it to ultraviolet light for a few minutes. Once erased, the EPROM can be reprogrammed.

Electrically erasable PROMs (EEPROMs) allow further flexibility. By exposing to ultraviolet light, we erase all the contents of an EPROM. EEPROMs, on the other hand, allow the user to selectively erase contents. Furthermore, erasing can be done in place; there is no need to place it in a special ultraviolet chamber. Flash memory is a special kind of EEPROM. One main difference between the EEPROM and flash memory lies in how the memory contents are erased. The EEPROM is byte-erasable whereas flash memory is block-erasable. Thus, writing in the flash memory involves erasing a block and rewriting it. Current systems use flash memory for BIOS so that changing BIOS versions is fairly straightforward (you just have to "flash" the new version). Flash memory is also becoming very popular as a removable media. The

SmartMedia, CompactFlash, and Sony's Memory Stick are all examples of various forms of removable flash media.

**Read/Write Memory**

Read/write memory is commonly referred to as random access memory (RAM), even though ROM is also random access memory. This terminology is so entrenched in the literature that we follow it here with a cautionary note that RAM actually refers to RWM. Read/write memory can be divided into static and dynamic categories. Static random access memory (SRAM) retains the data, once written, without further manipulation so long as the source of power holds its value. SRAM is typically used for implementing the processor registers and cache memories. The bulk of main memory in a typical computer system, however, consists of dynamic random access memory (DRAM). DRAM is a complex memory device that uses a tiny capacitor to store a bit. A charged capacitor represents 1 bit. Since capacitors slowly lose their charge due to leakage, they must be refreshed periodically to replace the charges representing 1 bit. A typical refresh period is about 64 ms. Reading from DRAM involves testing to see if the corresponding bit cells are charged. Unfortunately, this test destroys the charges on the bit cells. Thus, DRAM is a destructive read memory.

For proper operation, a read cycle is followed by a restore cycle. As a result, the DRAM cycle time, the actual time necessary between accesses, is typically about twice the read access time, which is the time necessary to retrieve a datum from the memory. Several types of DRAM chips are available. We briefly describe some of the most popular types next.

**FPM DRAMs**

Fast page-mode (FPM) DRAMs are an improvement over the previous generation DRAMs. FPM DRAMs exploit the fact that we access memory sequentially, most of the time.