CHAPTER FOUR

Register Transfer and Microoperations

IN THIS CHAPTER

4-1 Register Transfer Language

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. Some of the digital components introduced in Chap. 2 are registers that implement microoperations. For example, a counter with parallel load is capable of performing the micro-

microoperation

operations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers. The use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

register transfer language

The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation introduced in this chapter will be utilized in subsequent chapters to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

4-2 Register Transfer

registers

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name *MAR*. Other designations for registers are *PC* (for program counter), *IR* (for instruction register, and *R*1 (for processor register). The individual flip-flops in an n-bit register are numbered in sequence from 0 through $n - 1$, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 4-1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 4-1(a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol *L* (for low byte) and bits 8 through 15 are assigned the symbol *H* (for high byte). The name of the 16-bit register is *PC*. The symbol *PC*(0–7) or *PC*(*L*) refers to the low-order byte and *PC*(8–15) or *PC*(*H*) to the high-order byte.

register transfer

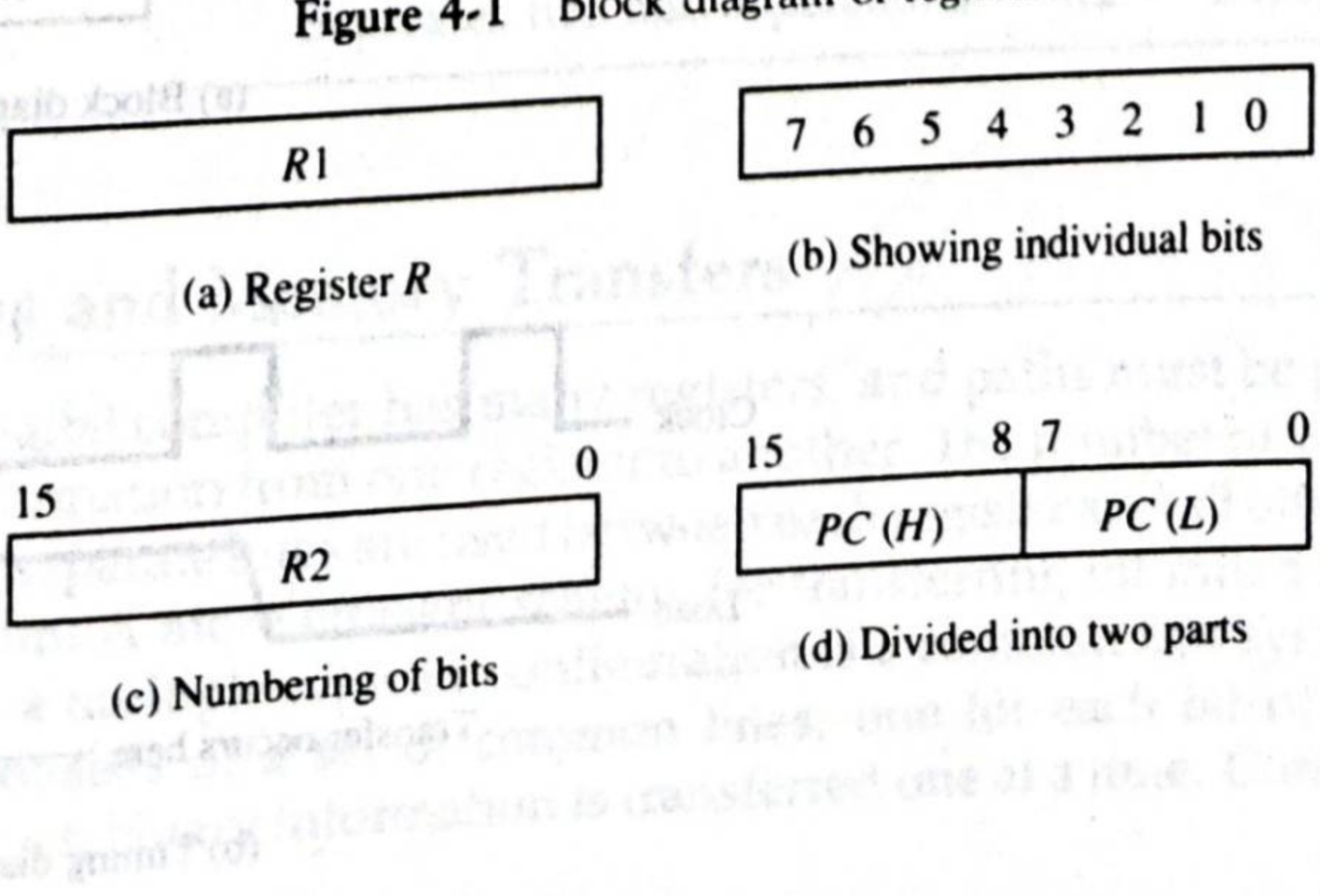
Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

denotes a transfer of the content of register *R*1 into register *R*2. It designates a replacement of the content of *R*2 by the content of *R*1. By definition, the content of the source register *R*1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Nor-

Figure 4-1 Block diagram of register.



(a) Register *R*

(b) Showing individual bits

(c) Numbering of bits

(d) Divided into two parts

mally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an *if-then* statement.

$$If\ (P = 1)\ then\ (R2 \leftarrow R1)$$

control function

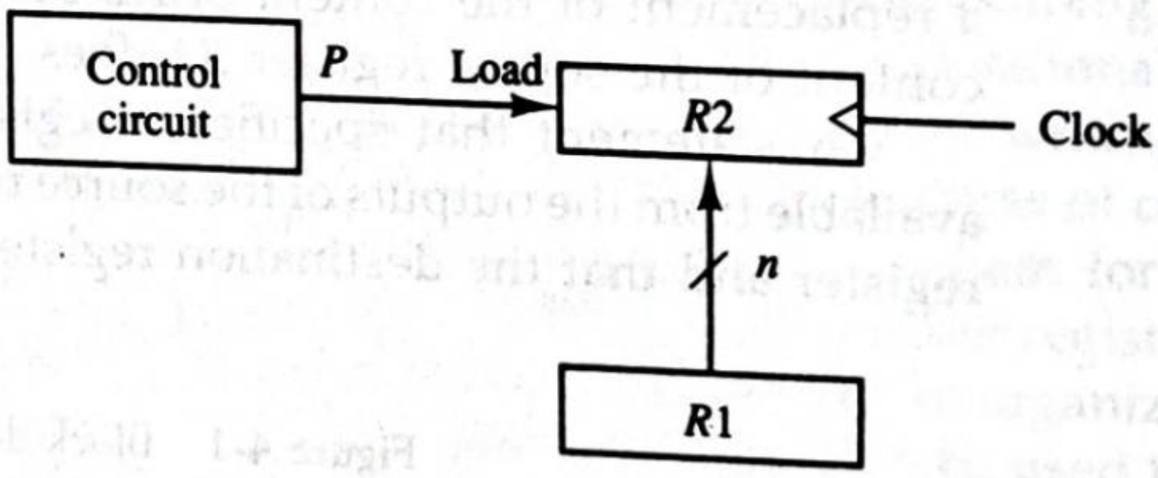
where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a *control function*. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

$$P:\quad R2 \leftarrow R1$$

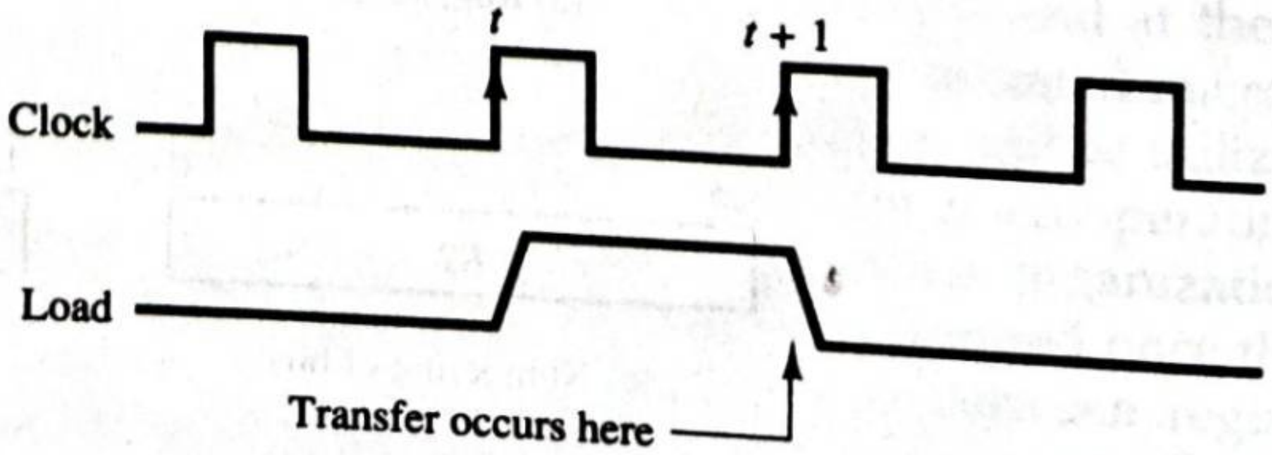
The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if $P = 1$.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 4-2 shows the block diagram that depicts the transfer from $R1$ to $R2$. The n outputs of register $R1$ are connected to the n inputs of register $R2$. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register $R2$ has a load input that is activated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As shown

Figure 4-2 Transfer from $R1$ to $R2$ when $P = 1$.



(a) Block diagram



(b) Timing diagram

in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t. The next positive transition of the clock at time $t + 1$ finds the load input active and the data inputs of $R2$ are then loaded into the register in parallel. P may go back to 0 at time $t + 1$; otherwise, the transfer will occur with every clock pulse transition while P remains active.

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as P becomes active just after time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time $t + 1$.

The basic symbols of the register transfer notation are listed in Table 4-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

$$T: \quad R2 \leftarrow R1, \quad R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that $T = 1$. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

4-3 Bus and Memory Transfers

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals

common bus

to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC-type CPU consists of LOAD and STORE instructions that have one memory and one register address, and computational-type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate $X = (A + B) * (C + D)$.

```
LOAD    R1, A        R1 ← M[A]
LOAD    R2, B        R2 ← M[B]
LOAD    R3, C        R3 ← M[C]
LOAD    R4, D        R4 ← M[D]
ADD     R1, R1, R2   R1 ← R1 + R2
ADD     R3, R3, R2   R3 ← R3 + R4
MUL     R1, R1, R3   R1 ← R1*R3
STORE   X, R1        M[X] ← R1
```

The load instructions transfer the operands from memory to CPU registers. The add and multiply operations are executed with data in the registers without accessing memory. The result of the computations is then stored in memory with a store instruction.

8-5 Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction

The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the

computer. The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Execute the instruction.

...m counter (PC) There is one register in the computer called the program counter or *PC* that keeps track of the instructions in the program stored in memory. *PC* holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction, and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

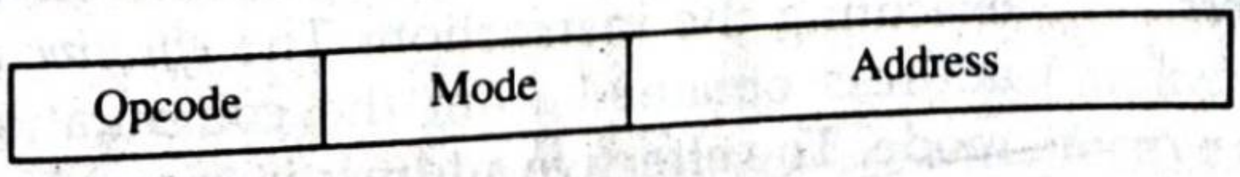
An example of an instruction format with a distinct addressing mode field is shown in Fig. 8-6. The operation code specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

...field

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions.

Figure 8-6 Instruction format with mode field.



Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode: In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Autoincrement or Autodecrement Mode: This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction. However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction. The *effective address* is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-

effective address

type instruction. It is the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in Chap. 5. They are summarized here for reference.

Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address. The indirect address mode is also explained in Sec. 5-1 in conjunction with Fig. 5-2.

A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation:

effective address = address part of instruction + content of CPU register

The CPU register used in the computation may be the program counter, an index register, or a base register. In either case we have a different addressing mode which is used for a different application.

Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction. To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The

index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position. With a base register, the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7. The two-word instruction at address 200 and 201 is a "load to *AC*" instruction with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. *PC* has the value 200 for fetching this instruction. The content of processor register *R*1 is 400 and the content of an index register *XR* is 100. *AC* receives the operand after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.