ADDRESSING MODES

- A program operates on data that reside in the computer's memory. These data can be organized in a variety of ways.
- The different ways in which the location of an operand is specified in an instruction are referred to as ***addressing modes***.

IMPLEMENTATION OF VARIABLES AND CONSTANTS

- Variables and constants are the simplest data types found in every computer program.
- In assembly language, a variable is represented by allocating a register or a memory location to hold its value.

  Ex : MOVE A, Ri
- The operand is specified by the name of the register or the address of the memory location where the operand is located. The precise definitions of these two modes are:
- ***Register mode*** — The operand is the contents of a processor register; the name (address) of the register is given in the instruction.
- ***Absolute mode*** — The operand is in a memory location; the address of this location is given explicitly in the instruction. This mode is also called as ***Direct addressing mode***
- The instruction

  Move A,R2

  uses these two modes. Processor registers are used as temporary storage locations where the data in a register are accessed using the Register mode.

- Consider the representation of constants. Address and data constants can be represented in assembly language using the Immediate mode.
- ***Immediate mode*** — The operand is given explicitly in the instruction.

  For example, the instruction

Move #200, R0

places the value 200 in register R0. Clearly, the Immediate mode(#) is only used to specify the value of a source operand.

- Constant values are used frequently in high-level language programs. For example, the statement

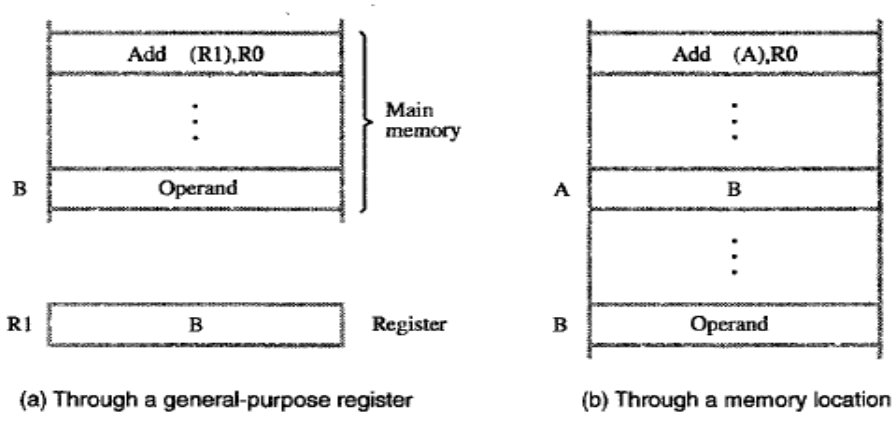
  A = B + 6

  contains the constant 6. Assuming that A and B have been declared as variables and may be accessed using the Absolute mode, this statement may be compiled as follows:

  Move B,R1
  Add #6,R1
  Move R1,A

INDIRECTION AND POINTERS

- In some addressing modes, the instruction does not give the operand or its address explicitly. Instead, it provides information from which the memory address of the operand can be determined. This address as the *effective address* (EA) of the operand.
- ***Indirect mode*** — The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.
- The indirection is denoted by placing the name of the register or the memory address given in the instruction in parentheses.



(a) Through a general-purpose register (b) Through a memory location

Figure 2.11 Indirect addressing.

- To execute the Add instruction in Figure (a), the processor uses the value B, which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location, B. The value read is the desired operand, which the processor adds to the contents of register R0.
- Indirect addressing through a memory location is also possible as shown in Figure (*b)*. In this case, the processor first reads the contents of memory location A and then, requests a second read operation using the value B as an address to obtain the operand.
- The register or memory location that contains the address of an operand is called a ***pointer.***

- Consider the program for adding a list of n numbers. Indirect addressing can be used to access successive numbers in the list, resulting in the following program.

Address	Contents		
	Move	N,R1	Initialization
	Move	#NUM1,R2	
	Clear	R0	
LOOP	Add	(R2),R0	
	Add	#4,R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,SUM	

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

- Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value *n* from memory location N into R1 and uses the immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0.
- The first time through the loop, the instruction

Add (R2),R0

fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

INDEXING AND ARRAYS

- This type of addressing mode is useful in dealing with lists and arrays.
- ***Index mode*** — The effective address of the operand is generated by adding a constant value to the contents of a register.
- The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as an *index register*.
- The Index mode is symbolically written as

  X(R*i*)

  where, X denotes the constant value contained in the instruction and R*i* is the name of the register involved.
- The effective address of the operand is given by

  EA = X + [R*i*]

  The contents of the index register are not changed in the process of generating the effective address.

- The following Figure illustrates two ways of using the Index mode. In Figure (*a)*, the index register, R1, contains the address of a memory location, and the value X defines an *offset* (or *displacement*) from this address to the location where the operand is present.

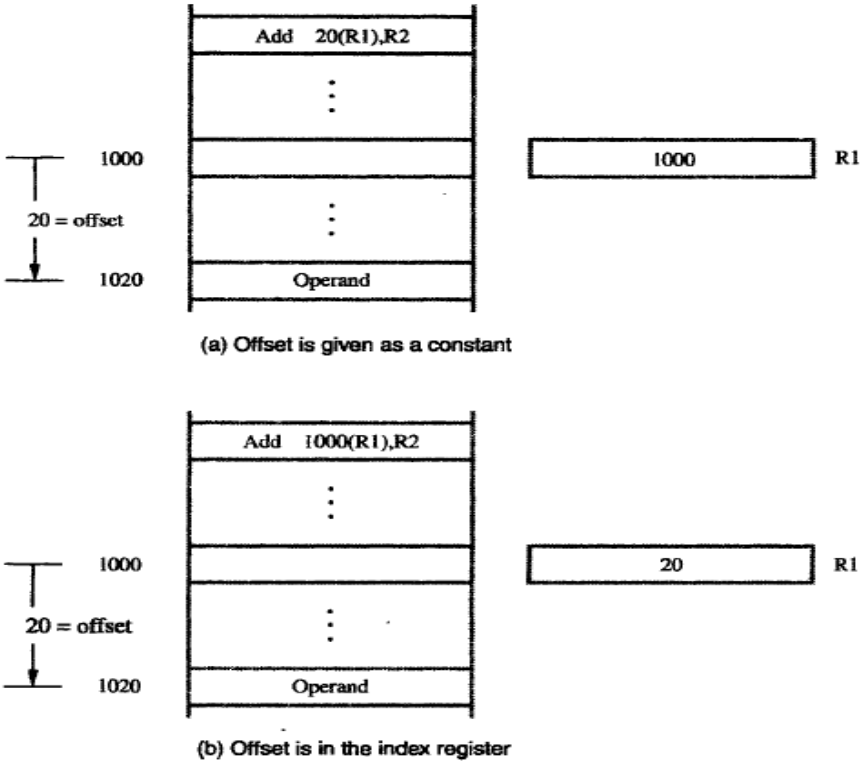


Figure 2.13 Indexed addressing.

- An alternative way is illustrated in Figure (*b*). Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand.
- In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.

RELATIVE ADDRESSING

- The Index mode is defined by using general-purpose processor registers. A useful version of this mode is obtained if the program counter, PC, is used instead of a general purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified "relative" to the program counter, the name Relative mode is associated with this type of addressing.
- ***Relative mode*** — The effective address is determined by the Index mode using the program counter in place of the general-purpose register R*i*.

- This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as

  Branch>0 LOOP

  causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

ADDITIONAL MODES

The following two modes are useful for accessing data items in successive locations in the memory.

- ***Autoincrement mode*** — The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
- We denote the Autoincrement mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Autoincrement mode is written as

  $(Ri)+$

- Implicitly, the increment amount is 1 when the mode is given in this form. But in a byte addressable memory, this mode would only be useful in accessing successive bytes of some list. To access successive words in a byte-addressable memory with a 32-bit word length, the increment must be 4.
- Computers that have the Auto increment mode automatically increment the contents of the register by a value that corresponds to the size of the

accessed operand. Thus, the increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.

- ***Auto decrement mode*** — The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.
- We denote the Auto decrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write

  $-(Ri)$
- In this mode, operands are accessed in descending address order.
- These two modes can be used together to implement an important data structure called a stack.
- An assembly language program to compute sum of all the elements in the array using Autoincrement mode is as follows:

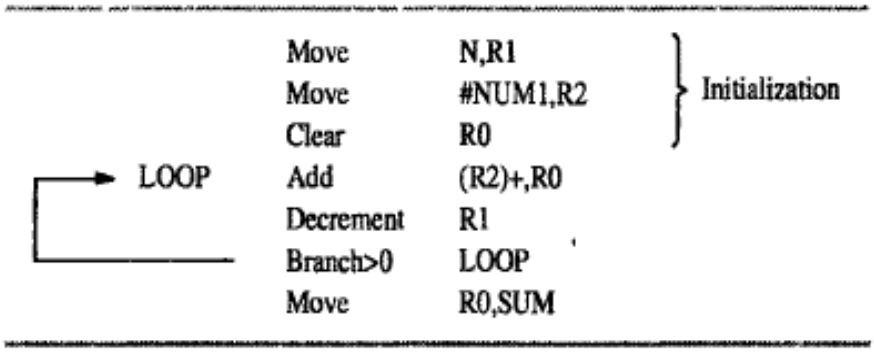


Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

Summary of Addressing Modes:

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

BASIC INPUT/OUTPUT OPERATIONS

- Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as *program-controlled I/O*.
- The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.
  - A solution to this problem is as follows: On output, the processor sends the first character and then waits for a signal from the display

that the character has been received. It then sends the second character, and so on.

- Input is sent from the keyboard in a similar way; the processor waits for a signal indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. Then the processor proceeds to read that code.

- The keyboard and the display are separate devices as shown in Figure 2.19.

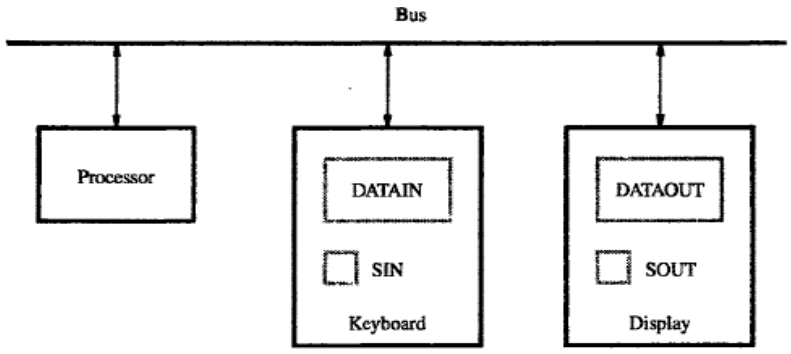


Figure 2.19 Bus connection for processor, keyboard, and display.

Read operation:

- A program monitors SIN flag, and when SIN is set to 1, the processor reads the contents of DATAIN buffer associated with the keyboard. When the character is transferred to the processor, SIN is automatically cleared to 0. If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.

- Let us assume that bit *b*3 in INSTATUS register corresponds to SIN flag. The read operation just described may now be implemented by the machine instruction sequence

```
READWAIT Testbit #3,INSTATUS
Branch=0 READWAIT
MoveByte DATAIN,R1
```

Write Operation:

- A program monitors SOUT flag, and when SOUT is set to 1, the processor transfers a character code to DATAOUT buffer associated with the display. The transfer of a character to DATAOUT clears SOUT to 0; when the display device is ready to receive a second character, SOUT is again set to 1.
- Let us assume that bit *b*3 in OUTSTATUS register corresponds to SOUT flag. The write operation just described may now be implemented by the machine instruction sequence

```
WRITEWAIT Testbit #3,OUTSTATUS
Branch=0 WRITEWAIT
MoveByte R1,DATAOUT
```

- The Testbit instruction tests the state of one bit in the destination location, where the bit position to be tested is indicated by the first operand. If the bit tested is equal to 0, then the condition of the branch instruction is true, and a branch is made to the beginning of the wait loop. When the device is ready, that is, when the bit tested becomes equal to 1, the data are read from the input buffer or written into the output buffer.

- The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a *device interface*.

Assembly language program to read a line of characters and display it:

- As the characters are read in, one by one, they are stored in a data area in the memory and then echoed back out to the display. The program finishes when carriage return, CR, is read. The address of the first byte location of the memory where the line is to be stored is LOC. Register R0 is used to point to this area, and it is initially loaded with the address

LOC by the first instruction in the program. R0 is incremented for each character read and displayed by the Autoincrement addressing mode used in the Compare instruction.

```
        Move        #LOC,R0          Initialize pointer register R0 to point to the
                                       address of the first location in memory
                                       where the characters are to be stored.
READ    TestBit     #3,INSTATUS      Wait for a character to be entered
        Branch=0    READ               in the keyboard buffer DATAIN.
        MoveByte    DATAIN,(R0)      Transfer the character from DATAIN into
                                       the memory (this clears SIN to 0).
ECHO    TestBit     #3,OUTSTATUS     Wait for the display to become ready.
        Branch=0    ECHO
        MoveByte    (R0),DATAOUT     Move the character just read to the display
                                       buffer register (this clears SOUT to 0).
        Compare     #CR,(R0)+        Check if the character just read is CR
                                       (carriage return). If it is not CR, then
        Branch≠0    READ               branch back and read another character.
                                       Also, increment the pointer to store the
                                       next character.
```

Figure 2.20 A program that reads a line of characters and displays it.

STACKS

- A computer program often needs to perform a particular subtask using the familiar subroutine structure. In order to organize the control and information linkage between the main program and the subroutine, a data structure called a stack is used.
- A *stack* is a list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. Another descriptive phrase, *last-in–first-out* (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval

begins. The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

- Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations.

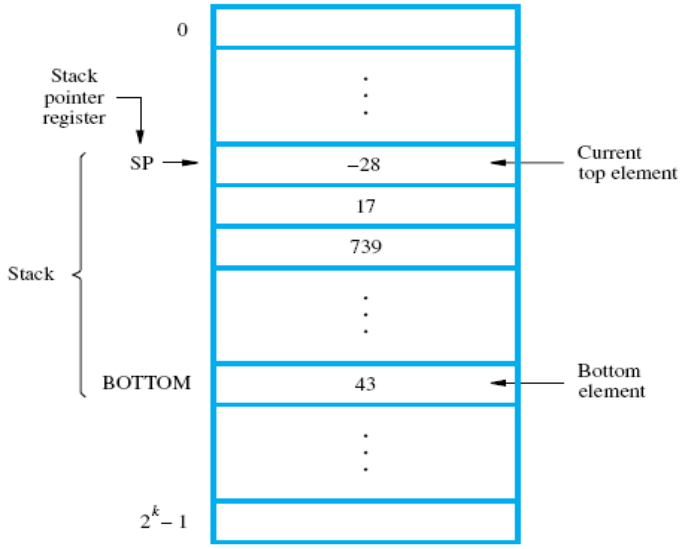


Figure 2.21 A stack of words in the memory.

- Figure 2.21 shows a stack of word data items in the memory of a computer. It contains numerical values, with 43 at the bottom and −28 at the top.
- A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the *stack pointer* (SP). It could be one of the general-purpose registers or a register dedicated to this function.
- If we assume a byte-addressable memory with a 32-bit word length, the push operation can be implemented as

```
Subtract #4,SP
Move NEWITEM,(SP)
```

where, the Subtract instruction subtracts the source operand 4 from the destination operand contained in SP and places the result in SP. These two instructions move the word from location NEWITEM onto the top of the stack, decrementing the stack pointer by 4 before the move.

- The pop operation can be implemented as

```
Move (SP),ITEM
Add #4,SP
```

These two instructions move the top value from the stack into location ITEM and then increment the stack pointer by 4 so that it points to the new top element.

- If the processor has the Autoincrement and Autodecrement addressing modes, then the push operation can be performed by the single instruction

```
Move NEWITEM,−(SP)
```

and the pop operation can be performed by

```
Move (SP)+,ITEM
```

- When a stack is used in a program, it is usually allocated a fixed amount of space in the memory. In this case, we must avoid pushing an item onto the stack when the stack has reached its maximum size. Also, we must avoid attempting to pop an item off an empty stack, which could result from a programming error. Suppose that a stack runs from location 2000 (BOTTOM) down no further than location 1500. The stack pointer is loaded initially with the address value 2004. Recall that SP is decremented by 4 before new data are stored on the stack. Hence, an initial value of 2004 means that the first item pushed onto the stack will be at location 2000. To prevent either pushing an item on a full stack or

popping an item off an empty stack, the single-instruction push pop operations can be replaced by the instruction sequences shown in Fig.

SAFEPOP	Compare Branch>0	#2000,SP EMPTYERROR	Check to see if the stack pointer contains an address value greater than 2000. If it does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action.
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

(a) Routine for a safe pop operation

SAFEPUSH	Compare Branch≤0	#1500,SP FULLERROR	Check to see if the stack pointer contains an address value equal to or less than 1500. If it does, the stack is full. Branch to the routine FULLERROR for appropriate action.
	Move	NEWITEM,−(SP)	Otherwise, push the element in memory location NEWITEM onto the stack.

QUEUES

- Another useful data structure that is similar to the stack is called a *queue*. Data are stored in and retrieved from a queue on a first-in–first-out (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.
- There are two important differences between how a stack and a queue are implemented. One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped. A single pointer is needed to point to the top of the stack at any given time. On the other hand, both ends of a queue move to higher addresses as data are added at the back and removed from the front. So two pointers are needed to keep track of the two ends of the queue.

- Another difference between a stack and a queue is that, without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a *circular buffer*. Let us assume that memory addresses from BEGINNING to END are assigned to the queue. The first entry in the queue is entered into location BEGINNING, and successive entries are appended to the queue by entering them at successively higher addresses. By the time the back of the queue reaches END, space will have been created at the beginning if some items have been removed from the queue. Hence, the back pointer is reset to the value BEGINNING and the process continues.

SUBROUTINES

- In a given program, it is often necessary to perform a particular subtask many times on different data values. Such a subtask is usually called a *subroutine*. For example, a subroutine may evaluate the *sine* function or sort a list of values into increasing or decreasing order.
- It is possible to include the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of the instructions that constitute the subroutine is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location.
- When a program branches to a subroutine we say that it is *calling* the subroutine. The instruction that performs this branch operation is named a Call instruction.
- After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to *return* to the program that called it by executing a Return instruction.

- Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling program resumes execution is the location pointed to by the updated PC while the Call instruction is being executed. Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.

- The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the *link register*. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

- The Call instruction is just a special branch instruction that performs the following operations:
  - Store the contents of the PC in the link register
  - Branch to the target address specified by the instruction

- The Return instruction is a special branch instruction that performs the operation:
  - Branch to the address contained in the link register

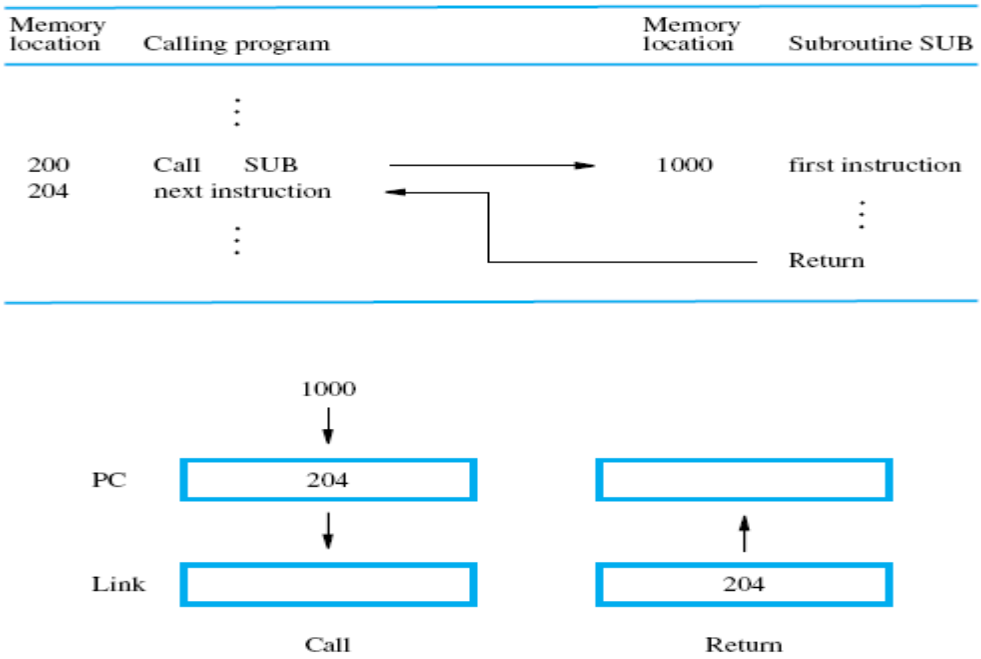


Figure 2.24 Subroutine linkage using a link register.

SUBROUTINE NESTING AND THE PROCESSOR STACK

- A common programming practice, called *subroutine nesting,* is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, destroying its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.
- Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it. The return address needed for this first return is the last one generated in the nested call sequence. That is, return addresses are generated and used in a last-in–first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto a stack.

PARAMETER PASSING

- When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, in this case, the results of the computation. This exchange of information between a calling program and a subroutine is referred to as *parameter passing*.
- Parameter passing may be accomplished in several ways. The parameters may be placed
  - in registers
  - in memory locations
  - on the processor stack
- Passing parameters through processor registers is straightforward and efficient.

Calling program

```
            Move      N,R1          R1 serves as a counter.
            Move      #NUM1,R2      R2 points to the list.
            Call      LISTADD       Call subroutine.
            Move      R0,SUM        Save result.
            .
            .
            .
```

Subroutine

```
LISTADD     Clear     R0            Initialize sum to 0.
LOOP        Add       (R2)+,R0      Add entry from list.
            Decrement R1
            Branch>0  LOOP
            Return                  Return to calling program.
```

Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.

- Figure 2.25 shows how the program for adding a list of numbers can be implemented as a subroutine, with the parameters passed through registers.
- The size of the list, *n*, contained in memory location N, and the address, NUM1, of the first number, are passed through registers R1 and R2.

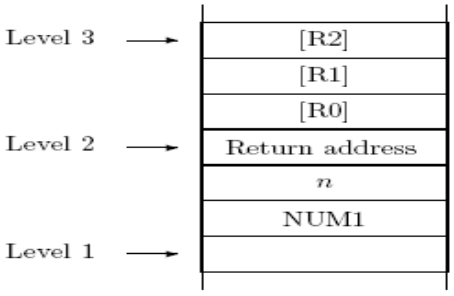
- The sum computed by the subroutine is passed back to the calling program through register R0.
- The Call instruction branches to the subroutine starting at location LISTADD. This instruction also pushes the return address onto the processor stack.
- The subroutine computes the sum and places it in R0. After the return operation is performed by the subroutine, the sum is stored in memory location SUM by the calling program.
- If many parameters are involved, there may not be enough general-purpose registers available for passing them to the subroutine. Using a stack, on the other hand, is highly flexible; a stack can handle a large number of parameters.
- Figure 2.26*a* shows the program, LISTADD, which can be called by any other program to add a list of numbers. The parameters passed to this subroutine are the address of the first number in the list and the number of entries. The subroutine performs the addition and returns the computed sum. The parameters are pushed onto the processor stack pointed to by register SP. Assume that before the subroutine is called, the top of the stack is at level 1 in Figure 2.26*b*.

```
Assume top of stack is at level 1 below.

            Move        #NUM1,−(SP)   Push parameters onto stac
            Move        N,−(SP)
            Call        LISTADD       Call subroutine
                                        (top of stack at level 2).
            Move        4(SP),SUM     Save result.
            Add         #8,SP         Restore top of stack
                                        (top of stack at level 1).
            ⋮

LISTADD     MoveMultiple R0−R2,−(SP)  Save registers
                                        (top of stack at level 3).
            Move        16(SP),R1     Initialize counter to n.
            Move        20(SP),R2     Initialize pointer to the list
            Clear       R0            Initialize sum to 0.
LOOP        Add         (R2)+,R0      Add entry from list.
            Decrement   R1
            Branch>0    LOOP
            Move        R0,20(SP)     Put result on the stack.
            MoveMultiple (SP)+,R0−R2  Restore registers.
            Return                    Return to calling program.
```

(a) Calling program and subroutine



(b) Top of stack at various times

Figure 2.26 Program of Figure 2.16 written as a subroutine; parameters passed on the stack.

- The calling program pushes the address NUM1 and the value *n* onto the stack and calls subroutine LISTADD. The Call instruction also pushes the return address onto the stack. The top of the stack is now at level 2. The subroutine uses three registers. Since these registers may contain valid data that belong to the calling program, their contents should be saved by pushing them onto the stack. We have used a single instruction, MoveMultiple, to store the contents of registers R0 through R2 on the stack. Many processors have such instructions. The top of the stack is now at level 3. The subroutine accesses the parameters *n* and

NUM1 from the stack using indexed addressing. Note that it does not change the stack pointer because valid data items are still at the top of the stack. The value *n* is loaded into R1 as the initial value of the count, and the address NUM1 is loaded into R2, which is used as a pointer to scan the list entries. At the end of the computation, register R0 contains the sum. Before the subroutine returns to the calling program, the contents of R0 are placed on the stack, replacing the parameter NUM1, which is no longer needed. Then the contents of the three registers used by the subroutine are restored from the stack. Now the top item on the stack is the return address at level 2. After the subroutine returns, the calling program stores the result in location SUM and lowers the top of the stack to its original level by incrementing the SP by 8.

Parameter Passing by Value and by Reference

- Note the nature of the two parameters, NUM1 and *n*, passed to the subroutines in Figures 2.25 and 2.26. The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called *passing by reference*. The second parameter is *passed by value,* that is, the actual number of entries, *n*, is passed to the subroutine.

THE STACK FRAME

- During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private work space for the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a *stack frame*.

- If the subroutine requires more space for local memory variables, they can also be allocated on the stack.

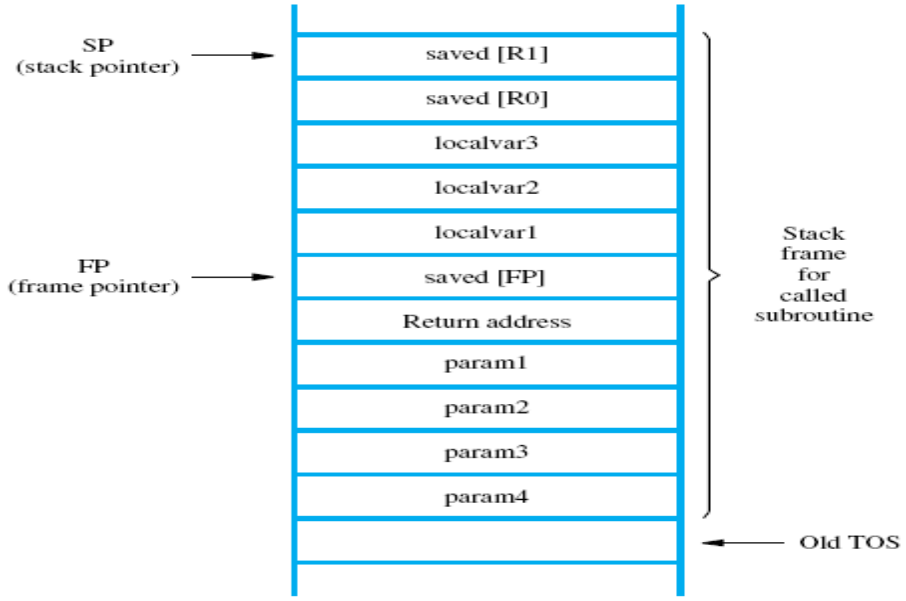


Figure 2.27 A subroutine stack frame example.

- Figure 2.27 shows an example of a commonly used layout for information in a stack frame. In addition to the stack pointer SP, it is useful to have another pointer register, called the *frame pointer* (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. These local variables are only used within the subroutine, so it is appropriate to allocate space for them in the stack frame associated with the subroutine. In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R0 and R1 need to be saved because they will also be used within the subroutine.
- With the FP register pointing to the location just above the stored return address, as shown in Figure 2.27, we can easily access the parameters and the local variables by using the Index addressing mode. The parameters can be accessed by using addresses 8(FP), 12(FP), ... The local variables can be accessed by using addresses −4(FP), −8(FP), . . .

The contents of FP remain fixed throughout the execution of the subroutine, unlike the stack pointer SP, which must always point to the current top element in the stack.

- Assume that SP points to the old top-of-stack (TOS) element in Figure 2.27. Before the subroutine is called, the calling program pushes the four parameters onto the stack. The Call instruction is then executed, resulting in the return address being pushed onto the stack. Now, SP points to this return address, and the first instruction of the subroutine is about to be executed. This is the point at which the frame pointer FP is set to contain the proper memory address. Since FP is usually a general-purpose register, it may contain information of use to the calling program. Therefore, its contents are saved by pushing them onto the stack. Since the SP now points to this position, its contents are copied into FP.
- Thus, the first two instructions executed in the subroutine are

```
Move FP,−(SP)
Move SP,FP
```

After these instructions are executed, both SP and FP point to the saved FP contents.

- Space for the three local variables is now allocated on the stack by executing the instruction

```
Subtract #12,SP
```

- Finally, the contents of processor registers R0 and R1 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in the figure.
- The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction

```
Add #12,SP
```

and pops the saved old value of FP back into FP. At this point, SP points to the return address, so the Return instruction can be executed, transferring control back to the calling program.

- The calling program is responsible for removing the parameters from the stack frame, some of which may be results passed back by the subroutine. The stack pointer now points to the old TOS.

FUNDAMENTAL CONCEPTS

The instructions of a program are loaded in sequential locations in main memory. To execute a program, processor fetches one instruction at a time and performs the operations specified. The processor keeps track of the address of the memory location containing the next instruction using a dedicated register, called the **Program Counter (PC).** After fetching an instruction, the contents of PC are updated to point to the next instruction in the sequence. A branch instruction loads a new address into the PC.

Let us assume that each instruction is of 4 bytes and is stored in one memory word. ***To execute an instruction, the processor has to perform the following three steps:***

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are interpreted as instruction to be executed. They are loaded into another special purpose register called **Instruction Register (IR)**. Symbolically, this operation is denoted as

   IR ← [PC].

2. Since, the memory is byte addressable, increment the contents of the PC by 4 to get the next word in the memory.

   PC ← PC + 4

3. Carry out the actions specified by the instruction in the IR.

- If the instruction occupies more than one word, step 1 and 2 must be repeated so as to fetch the complete instruction. These two steps are usually referred as ***fetch phase***; step 3 constitutes the ***execution phase***.
- To know how these operations are implemented, we need to know the internal structure of the processor. The various functional blocks of the CPU can be organized and interconnected in a variety of ways. The

following Fig. shows a simple organization in which the arithmetic and logic unit (ALU) and all the registers are connected using a ***single common bus***. This bus is internal to the processor.

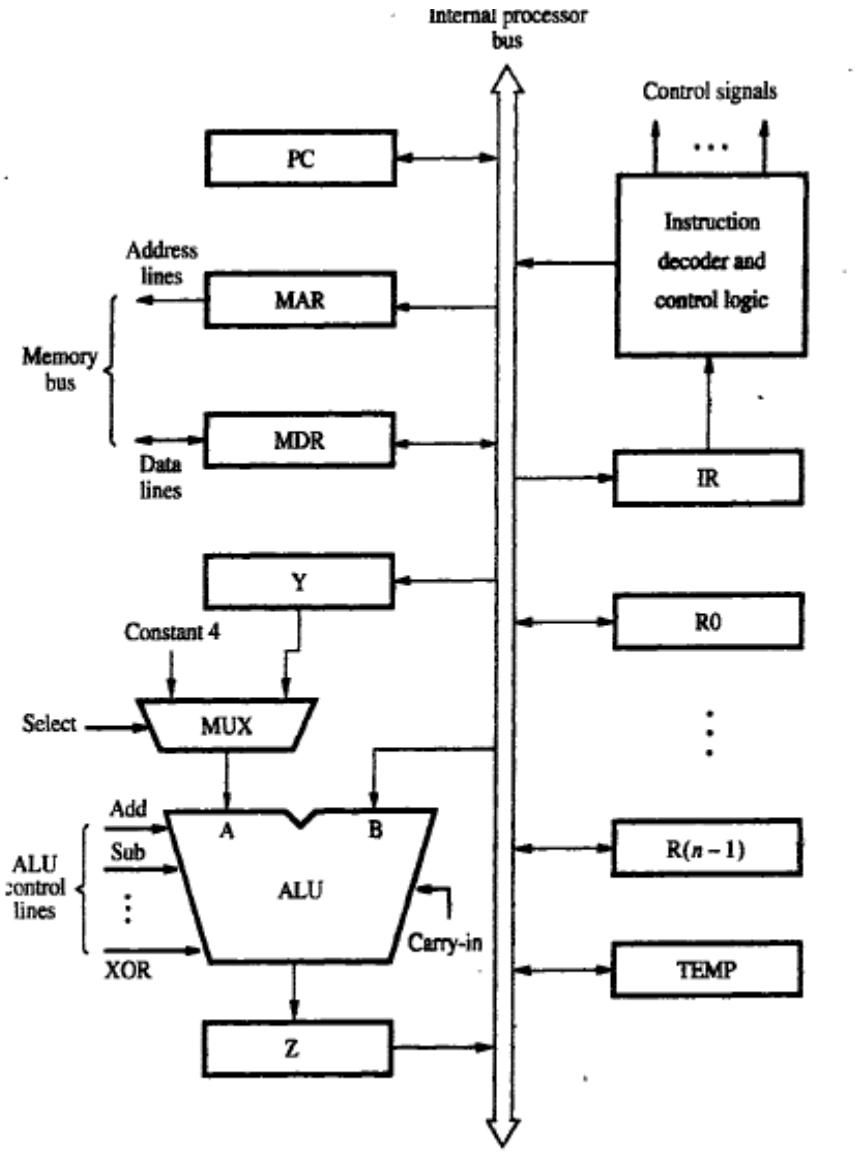


Figure 7.1 Single-bus organization of the datapath inside a processor.

- The data and address lines of the external memory bus are connected to the internal processor bus via Memory Data Register (MDR) and the Memory Address Register (MAR). Data may be loaded into MDR either from the memory bus or from internal processor bus. Data stored in

MDR may be placed on either bus. The input of the MAR is connected to the internal bus and its output is connected to the external bus.

- The control lines of the memory bus are connected to the instruction decoder and control logic block. This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for interacting with the memory bus.
- The number and function of processor register R0 through R(n-1) vary from processor to processor. Some may be general purpose; others may special purpose like stack pointer or index registers.
- Three registers Y, Z & TEMP are transparent to the programmmer, i.e, they are never referred directly by an instruction. They are used by the processor for temporary storage during execution of some instructions.
- The multiplexer MUX selects either the output of the register Y or a constant value 4.This is provided as input A to the ALU. The constant 4 is used to increment the contents of PC. The B input of the ALU is obtained directly from the processor bus. Thus, the 'select' line of MUX can be either select4 or select Y.
- The registers, the ALU and the interconnecting bus are collectively referred to as '***data path***'.

***Generally an instruction can be executed by performing one or more of the following operations in some specified sequence*:**

1. Transfer a word of data from one processor register to other or to the ALU.
2. Perform an arithmetic or logic operation and store the result in a processor register.
3. Fetch the contents of a given memory location and load them into a processor register.

4. Store a word of data from processor register into a given memory location.

REGISTER TRANSFERS :

Register gating and timing of data transfers:

- Instruction execution involves sequence of steps in which data are transferred from one register to another.
- For each register, two control signals are used to place the contents of that register on the bus or to load the data on the bus into the register.
- The input and output of register R_i are connected to the bus via switches controlled by the signals R_{in} and R_{out}. These are also called as ***gating signals.*** This is shown in the following fig.

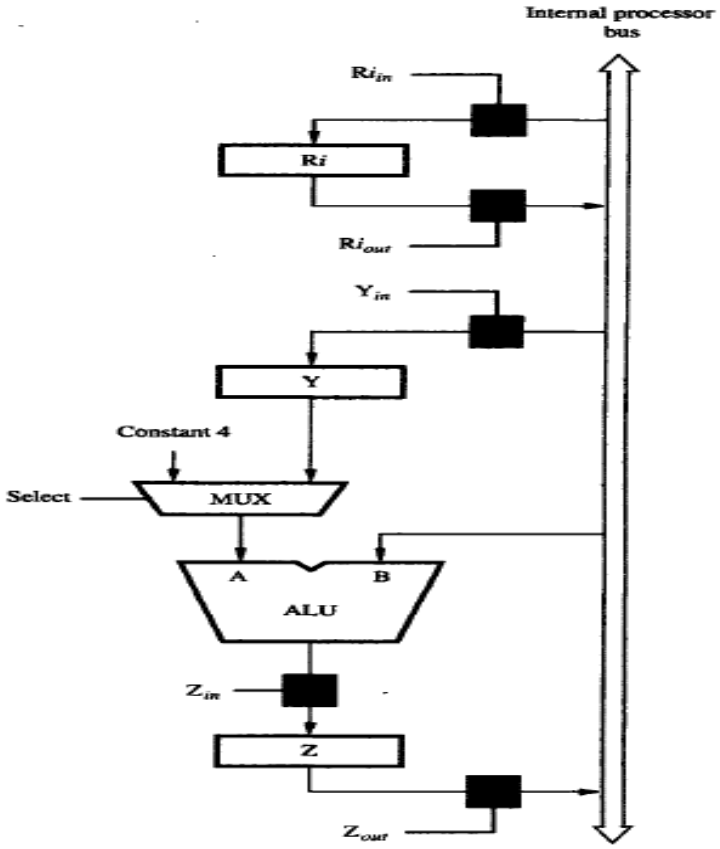


Figure 7.2 Input and output gating for the registers in Figure 7.1.

- When R_{in} is set to 1, the data on the bus are loaded into R_i and when R_{out} is set to 1, the contents of R_i are placed on the bus. When R_{out} or R_{in} is 0, data cannot be transferred between the processor bus and the register.

Example:

Consider an example of transferring data from register R_1 to R_4.The following actions are needed:

1. Enable the output of register R_1 by making R_{1out}=1.This places the contents of R_1 on the processor bus.
2. Enable the input of register R_4 by setting R_{4in}=1. This loads the data from the internal bus into register R_4.

Control step: $R1_{out}$,$R4_{in}$.

- All operations and data transfers within the processor takes place within time periods defined by the ***processor clock***.
- The control signals that govern a particular transfer are asserted at the start of the clock cycle. (In above example, $R1_{out}$ and $R4_{in}$ are set to 1).
- The registers consist of edge-triggered flip-flops. Hence, at the next active edge of the clock, the flip-flops that constitute R4 will load the data present at their inputs. At the same time, the control signals $R1_{out}$ and $R4_{in}$ will return to 0.
- When edge-triggered flip-flops are not used, two or more clock signals may be needed to guarantee proper transfer of data. This is known as ***multiphase clocking***.

Input and output gating for one register bit:

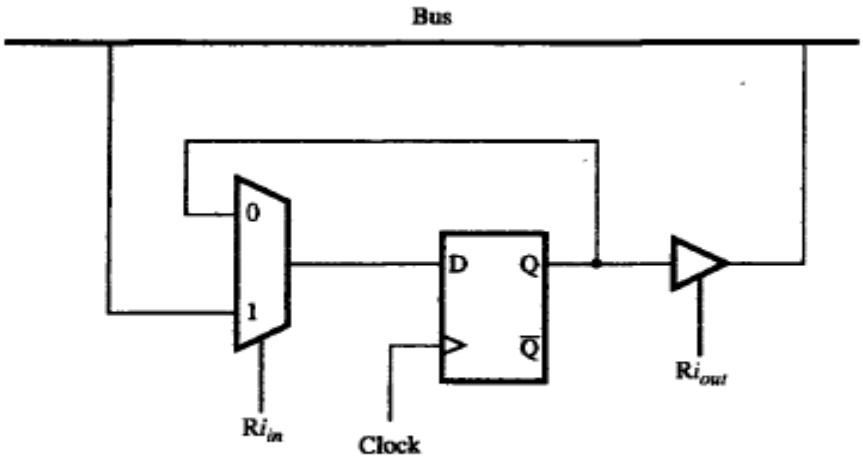


Figure 7.3 Input and output gating for one register bit.

- A two-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
- When the control input Ri_{in} =1, the multiplexer selects the data on the bus. This data will be loaded into the flip-flop at the rising edge of the clock.
- When Ri_{in}=0, the multiplexer feeds back the value currently stored in the flip-flop.
- The Q output of the flip-flop is connected to the bus via a tri-state gate.
- When Ri_{out} =0, the gate's output is in the high-impedence state. This corresponds to the open-circuit state of the switch.
- When Ri_{out}=1, the gate drives the bus to 0 or 1, depending on the value of Q.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION:

The ALU is a combinational circuit that has no internal storage. It performs arithmetic and logic operations on two operands applied to its A and B inputs. One of the operands is the output of the MUX and other operand is obtained directly from the processor bus. The result is stored temporarily in register Z.

Example:

Consider an example of adding the contents of register R1 to R2 and storing the result in R3. The sequence of operations are:

1. $R1_{out}$, Y_{in}

2. $R2_{out}$, SelectY, Add, Z_{in}

3. Z_{out}, $R3_{in}$

- The signals are activated for the duration of the clock cycle corresponding to that step, with all other signals deactivated during that time.
- In step 1, output of register R1 and inputs of register Y are enabled, causing the contents of R1 to be transferred via the bus to Y register.
- In step 2, select Y signal of MUX is selected, so the contents of Y register is transferred to input A of the ALU. Also, contents of R2 are gated onto the bus and, hence, to the input B of ALU. The add control signal of ALU is set , and hence addition is performed. The result is moved into register Z since Z_{in}=1.
- In step 3, contents of Z are transferred to destination register R3 using Z_{out} and $R3_{in}$ signals. This is an additional step, ***since only one register output can be enabled during a clock cycle in a single bus structure***.

FETCHING A WORD FROM MEMORY

- To fetch a word of information from memory, the processor has to specify the address of the memory location where this information is stored and request a Read operation.
- The processor transfers the required address to MAR, whose output is connected to the address lines of the memory bus.
- At the same time, the processor uses the control lines of the memory bus to indicate that a Read operation is needed.
- When the requested data are received from the memory, they are stored in register MDR, from where they can be transferred to other registers in the processor.

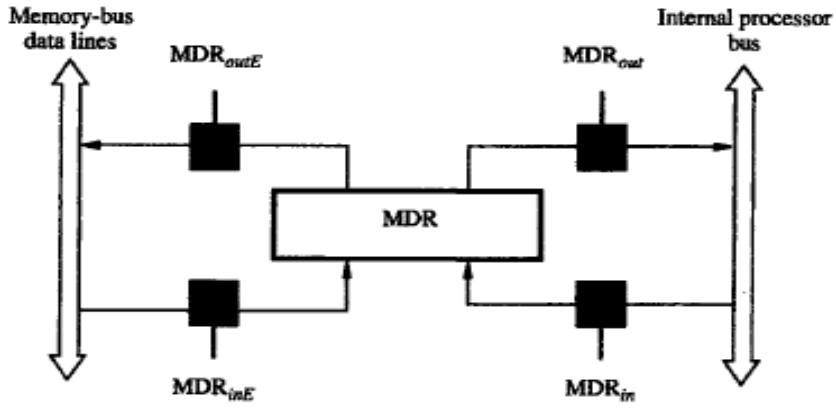


Figure 7.4 Connection and control signals for register MDR.

- Consider the above diagram. MDR has four control signals:
  - MDR_{in} and MDR_{out} control the connection to the internal bus.
  - MDR_{inE} and MDR_{outE} control the connection to the external bus.
- Circuit of input and output gating for one bit of register MDR can be easily written by modifying Fig 7.3.
  - Instead of a two-input multiplexer, a three-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.

- When the control input MDR_{inE} =1, the multiplexer selects the data on the external memory bus. This data will be loaded into the flip-flop at the rising edge of the clock.
- The Q output of the flip-flop is connected to the bus via another tri-state gate.
- When MDR_{outE} =0, the gate's output is in the high-impedence state. This corresponds to the open-circuit state of the switch.
- When MDR_{outE}=1, the gate drives the external memory bus to 0 or 1, depending on the value of Q.

- During memory Read and Write operations, the timing of internal processor operations must be coordinated with the response of the addressed device on the memory bus.
- The response time of each memory access varies.
- To accommodate the variability in response times, the processor waits until it receives an indication that the requested operation has been completed (We assume that a control signal called **Memory-Function-Completed**, **MFC** is used for this purpose).
- The addressed device sets this signal to 1 to indicate that the contents of the specified location have been read and are available on the data lines of the memory bus.

- Consider the instruction **Move (R1), R2**.

  Memory Read operation requires three steps, which can be described by the signals being activated as follows:

  1. $R1_{out}$, MAR_{in}, Read
  2. MDR_{inE}, WMFC
  3. MDR_{out}, $R2_{in}$

Timing Diagram for memory read operation:

- Assume that output of MAR is enabled all the time and its contents are always available on the address lines of the memory bus.

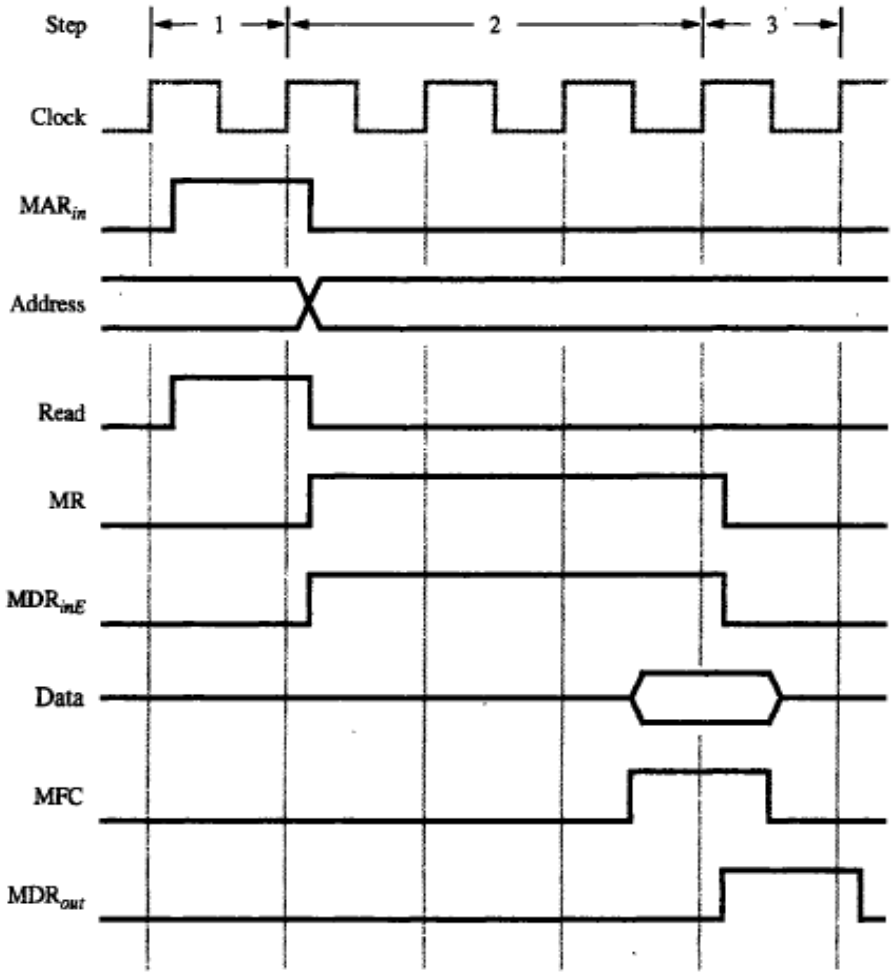


Figure 7.5 Timing of a memory Read operation.

- When a new address is loaded into MAR, it will appear on the memory bus at the beginning of the next clock cycle, as shown in fig7.5. Read control signal is activated at the same time MAR is loaded.
- Read control signal will cause the bus interface circuit to send a read command, MR, on the bus.

- MDR_{inE} is set to 1 for exactly the same period as the read command, MR. MDR_{inE} is activated while waiting for the response from the memory.
- The data received from the memory are loaded into MDR at the end of the clock cycle in which the MFC signal is received.
- In the next clock cycle, MDR_{out} is activated to transfer the data to register R2.

STORING A WORD IN MEMORY

To store a word in memory:

- The desired address is loaded into MAR.
- Then, the data to be written are loaded into MDR, and a Write command is issued.
- Execution of the instruction **Move R2, (R1)** requires the following sequence:
  1. $R1_{out}$, MAR_{in}
  2. $R2_{out}$, MDR_{in}, Write
  3. MDR_{outE}, WMFC

As in case of the read operation, the Write control signal causes the memory bus interface hardware to issue a Write command on the memory bus. The processor remains in step3 until the memory operation is completed and an MFC response is received.

********** **END OF UNIT - 2** **********