4

Instruction Set Processing

Thus far we have been dealing with the blocks from which computers are built. Chapter 2 described some of the decisions involved with choosing a method for representing the information within the computer. Chapter 3 is a discussion of the issues involved in doing some of the arithmetic operations required of a machine. In both cases, tradeoffs must be made to assure that the system resources are utilized in an efficacious manner. Representation ranges of number systems must be effectively weighed against the cost of those representations, and the targeted applications of the machine. Similarly, the methods used for doing the arithmetic must be balanced in such a way that the speed and complexity match the intended uses of the system.

In this chapter, we will look at how the arithmetic building blocks can be combined with other functional units, such as registers and memories, to create computing systems. Here we seek to address some of the basic questions conceming data manipulation methods. What are some of the issues involved in choosing an instruction set? What basic operations should be included? How do we specify the operations to be performed, and identify the operands to be used in that operation? What are the steps required to accomplish the specified work? What are the costs associated with the specification and execution of these instructions?

Let us first look at some of the basic tools used to describe machine structure and data manipulation methods. The tools are very simple: diagrams to identify structure and a register transfer language to specify data movement within that structure. Then we will identify some of the methods utilized by different machines to accomplish their work. Often what is considered "good" depends on several factors, and good design practices using one set of constraints will not be considered good design practices using a different set of constraints. Like the other ideas explored in the previous chapters, engineering choices are made after a careful examination of the alternative methods of doing the work. The key is to

choose appropriate metrics or measurement methods and to apply the metrics uniformly to the various alternatives.

The first area of interest concerns the data manipulation instructions and related topics: single address machines, two address machines, operand specification methods, and so on. Then we will look at program flow instructions: jumps, branches, subroutine calls, and the like. In a related area we will look at the machine reactions when exception conditions occur: interrupts and traps. This will necessitate some discussion of I/O programming methods as well. Finally, we will identify some of the issues in the ongoing RISC/CISC dehate, and explore reasons that the two methods are alternately considered good and bad.

4.1. Basic Building Blocks for Instruction Specification

As the computers space expands, the distinction between the responsibilities of the individual pans becomes more and more blurred. So, we will begin by looking at some of the concepts utilized in the early machines, and then as the operations and methods become more complex, we can recognize the parentage of the ideas, and see possible applications and design methods.

The building blocks used by the earliest machines comprised a very small set: registers, ALUs, memory. and data paths. In this discussion we will assume that the ALU model is as shown at the beginning of Chapter 3: two different inputs and an output. The ALU is assumed to be as wide as the machine; the word width is a decision based on what needs to be represented. We will assume that the ALU is capable of all of the arithmetic and logic operations which are required by the instructions.

Figure 4.1 shows the basic building blocks we will use in consideration of machine operation. The ALU we have already mentioned; it is used for data manipulation. Missing from the diagram are some very necessary lines, and in that sense the representation is incomplete. The missing lines include the control lines, which specify the action of the ALU (add, subtract, AND, etc.). and the data lines that do not form part of the designated inputs and output. These additional data lines often connect directly to a status register and include such things as the carry (in and out), the sign bit, overflow bit, and the like. Thus, for operations needing this additional interaction, we will assume that the connections do indeed exist and that the bits are transferred appropriately.

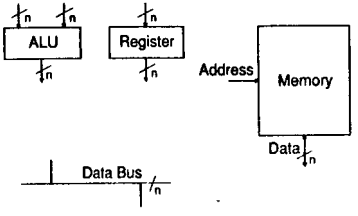


Figure 4.1. Basic Building Blocks for Instruction Set Processing: ALU, Register. Memory, and Communication Paths.

The data to be manipulated by an ALU is first stored in a memory, and such an element is shown in the figure. For our purposes we will say that the memory is as wide as the data path, but we will see later how this may be altered as part of the machine design. Our model for the memory element is simply that there are M memory locations, and these are arranged in such a way that they can be accessed by an address supplied on $\lceil \log_2(M) \rceil$ address lines. The data path allows reading and writing of data in these locations. As with the ALU, some lines are missing from the memory block as shown in Figure 4.1. These lines are the control lines used to cause the actual write or read of data from the memory devices. These lines are system-specific, and we will assume that the designer is aware of the required lines and handles them in an appropriate fashion.

Much of our current practice of memory system design and utilization is the result of the ideas explored by Von Neumann and his colleagues in the late 1940s [BuGo46]. Some of the earliest memory systems were organized such that the instructions could be held in one memory, and the data in another, and that these two memories were disjoint in function and fabrication. However, Von Neumann observed that the memories organized in that manner were not always effectively used; some tasks would leave the data memory practically empty while crowding the program memory, or vice versa. So he reasoned that since both instructions and data were basically information, both could be stored in the same memory space. Organizing the memory system in this manner brought a number of benefits, since programs could be treated as data. Instructions could be selectively altered to allow different functions or addresses as required, or data values could act as instructions if the conditions permitted. By organizing the memory in this fashion, only one memory element was needed, with its associated addressing and data retrieval capabilities. The two types of information, data and instructions, were combined into the same memory. The principal drawback to the arrangement was that interaction with the memory element was now needed for both types of information, and hence the path between computational functions and storage functions became a primary impediment to the effective processing speed. This has become known as the Von Neumann bottleneck, and we will present some of the suggestions made to minimize its effect. However, we will still treat memory as a linear array of storage locations, accessed by an appropriate address.

Another element shown in the figure is the data path. The width of the data path is assumed to be the same as the machine, but, as with most generalizations, exceptions can be found. We will use that width as a natural value, and later we will discuss ways to use widths other than the basic machine width for transferring information. These interconnections can be point-to-point wires from one element to another (containing the appropriate number of individual wires), or they can be buses, which are capable of transferring information between several distinct elements. Direct connections allow for high speed, but have low versatility. With tri-state logic readily available, a number of alternative busing arrangements can be made. We will discuss various types of buses in Chapter 6.

The final element shown in Figure 4.1 is the register. For our purposes, this is an element that is as wide as needed to match the buses, memories, and ALUs, used for storing information. This is another basic device that needs additional control lines not included in the figure. A register will require a clock line identifying when data is stable on the input line, and the register should load that data into its collection of storage elements. Other control lines may also be needed, such as output control lines for tri-state devices, or shift/load control lines for multifunction registers. Again, we will assume that the designer of the system is

aware of the capabilities of the registers being used, and that appropriate control lines are included in the machine.

Registers are used for a variety of applications, and generally receive names that denote their function. Figure 4.2 shows a block diagram that will serve as a vehicle for describing how the various registers and other elements function together to accomplish work. By work we mean the information transfers required to do some task. The registers shown in the figure form a fairly minimal set:

- ***Memory Address Register (MAR).*** This collection of storage elements bas the responsibility of identifying the memory location of the information to be transferred. The transfer could be either into or out of memory.
- ***Memory Buffer Register (MBR).*** The memory buffer register is used to store information moved into and out of memory. With destructive readout devices, such as core memory, it is a requirement; reading the value of a memory location destroys the contents of that location, and to preserve what was there it must be written back. The value to be restored is obtained from the MBR as the value is being used by the other parts of the circuit. With most semiconductor memories, the storage of the data going into and out of the memory is not required, and this register is optional, and used only in systems where there is a specific requirement to maintain the data after it is read.
- ***Program Counter (PC).*** The program counter is used to identify the location of the instruction to execute next. For machines that store one instruction per memory location, this register will increment by 1 during the execution of an instruction, which is why it has become known as a program "counter." Other organizations will have differing requirements for updating the PC value. For now, we will assume that the program counter will increment as needed to specify the next value needed from the instruction stream.
- ***Instruction Register (IR).*** The instruction register is used to store the instruction currently being executed. This allows the control portion of the machine to assert the control lines of the registers, memories, and arithmetic elements in an appropriate manner to cause the action needed. The design of the control section will be the subject of the next chapter. The IR may only be as wide as

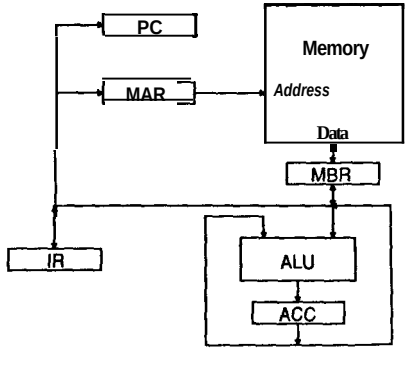


Figure 4.2. Block Diagram of Simple Machine.

needed to store the operation code of the instruction, or it may be wide enough to keep a temporary copy of all of the information associated with an instruction.

- *Accumulator* (ACC). The accumulator is shown here as the receptacle of the action of all of the data manipulation instructions. In the next section we will discuss the implications of the use of an accumulator for doing arithmetic.

This collection of resources (registers, ALUs, memory, and data paths) provides a sketchy view of the system, but it is sufficient to represent transactions that occur within the machine. We also need some method of describing those transactions. To do useful work we will need to specify the work to be done, and this work will be directed by the control section. The user of a computer system has a view of what the capabilities of the computer system are, and this view results directly from the instructions that the machine can execute. This view of the machine, or the appearance of the machine as seen by the assembly language programmer, is sometimes called the instruction set architecture of the system. By the use of the instructions included in this set, the user specifies the action which should occur on the data.

This work of an instruction is accomplished by a fetch-decode-execute mechanism: the instruction is fetched from memory and placed in a register specifically designated for that purpose (the IR), the required decoding is performed, and then the data transfers required by that instruction are executed. At the completion of this action, the machine starts over again, requesting another instruction, decoding it, and performing the needed action. The process continues until the machine has completed all of the designated instructions.

The action of an instruction can be described by identifying the data transfers needed to do the requested work. The specification of this work is done by a register transfer language (RTL); the transfers occur along the permissible data paths in the machine from one major component to another. Only transfers that can actually occur, given an accurate block diagram of the system, are permissible components of the specification for an instruction. For example, transfers from the PC or MAR to the MBR of the system shown in Figure 4.2 would not be possible, since the data paths between those elements do not permit data transfer in that direction. Thus, RTL descriptions specify the order of register transfers and arithmetic action required to carry out the work of an instruction. This information can then be utilized by the designer of the control system to identify the order of activation of control lines to actually cause the desired transfers. This points out one of the basic divisions of the computer design process: the data path (with its appropriate arithmetic capabilities) is specified. and then in a quite separate process the control section for the data path is designed. The design of the data path section is done in such a way that data manipulation goals are met. The design of the control section is then carried out so that the timing requirements of the system are met.

A register transfer language can become as simple or as complex as needed to specify the transfers required in the system. Since we will be using an RTL to describe the action of systems in this chapter and in the remainder of the book, we will describe the few primitives which will follow. The basic operation is the transfer of the contents of one register to another:

$$PC \rightarrow MAR$$

specifies that the contents of the program counter are transferred to the memory address register. If the data paths of the system are rich enough to allow multiple operations in the same time period, these can be represented by specifically linking the transfers together:

$$\begin{matrix} PC + 1 \rightarrow PC \\ MBR \rightarrow IR \end{matrix}$$

identifies that in the same time period the value of the program counter is incremented and the contents of the MBR are transferred to the IR. Normally, all of the information is involved in the transfer. However, if a subset of the information is to be transferred, then the specific bits are identified by the use of pointed brackets:

$$IR\langle 3{:}0\rangle \rightarrow ALU$$

specifies that bits 3 to 0 of the instruction register are directed to the ALU. Similarly, locations of memory or a set of registers are specified with square brackets:

$$REG[2] \rightarrow MEM[MAR]$$

indicates that the contents of register 2 in a general register set (REG) is transferred to the location in memory identified by the memory address register. Finally, for operations that are conditional in nature, we include an "if' facility patterned after the C language if construct:

$$\begin{matrix} \text{if (carry} == 1) & PC - 24 \rightarrow PC \\ \text{else} & PC + 1 \rightarrow PC \end{matrix}$$

identifies that if the carry is equal to 1, the program counter is adjusted by a factor of –24; otherwise the program counter is incremented.

Using the above constructs, a wide variety of instructions can be specified. For example, consider the following add instruction:

fetch: These register transfers get the instruction.

PC → MAR	Instruction location to MAR.
M[MAR] → MBR	Put instruction in MBR.
MBR → IR	And then put it in the IR.
PC + 1len → PC	Bump the program counter to next instruction.

decode The decode process identifies the instruction.

execute: and the execute portion performs the needed work.

IR<adr> → MAR	Address of operand to MAR.
M[MAR] → MBR	This is value to add to ACC.
ACC + MBR → ACC	Do the *actual* work of *instruction.*

At this point we will pause to consider briefly some of the timing considerations. All of the operations identified by the RTL require some finite time to accomplish. Exactly how much time is required depends on the technology of implementation and the electrical characteristics of the system. A simple register

transfer in a tri-state bus system requires time for the source register to be enabled, time for the data to become stable on the bus, and a setup time and a hold time for the data at the destination register. These times become very important to the designer of the control system, as all of the appropriate timing requirements must be met. In this chapter, we will assign times for the operations specified by the **RTL** for some of the instructions. These times, when added together. identify the total time required for the execution of the instruction. The times required for operations specified in **RTL** statements will be identified by a number in parentheses with the statement, and that number represents the execution time in nanoseconds.

By identifying the times required for the actions specified by the **RTL** statements, time can be used as a metric for the comparisons that need to be made in system evaluations. The overall instruction rate is then the inverse of the average instruction time. It is possible to increase the instruction rate (decrease the instruction time) by increasing the complexity of the system. For example, concurrent register transfers can be possible if multiple data paths exist within the system. Note, however, that the increased complexity may also result in longer machine cycle times, and this must be considered in the process of creating a system. As before, the tradeoffs involving complexity and speed must be made by the system architect using reasonable engineering judgements based on metrics that demonstrate the effective use of system resources.

Another piece of information used in the RTL descriptions included here is a statement number, which allows identificationof the steps of an instruction. This identification is often needed in the description of the process.

With the ability to represent the machines at the. register level, the data paths connecting the registers and the transfers of data between the major components, let us examine some of the methods used to organize machines and perform useful work.

4.2. Single Address Machines

The first machines constructed made very judicious use of registers since registers required a nontrivial amount of system resources. One of the registers was designated as the one that would be utilized in arithmetic and logic operations; others were also involved as needed. The register involved in these operations was most often called the accumulator, as we have indicated in Figure **4.2.** This same technique has been used in many different machines, and provides insight when compared to techniques more prevalent in newer architectures.

On machines that operate in this manner, operations requiring only one operand, such as complement, increment, clear, and the like, find the operand in the accumulator. And the result remains in the accumulator. Functions requiring two operands also use the value in the accumulator as one of the operands. The other operand is identified by a single address in the instruction; hence the name single address machine. To demonstrate how these machines might perform each kind of instruction. let us use the block diagram shown in Figure **4.2** and identify the transfers needed for a negate instruction and a subtract instruction. We are assuming that the machine in question uses the two's complement number system, so forming the negative of a given value can be accomplished by complementing and incrementing. The following **RTL** description implements the negate instruction:

```
        fetch:
1           PC → MAR        Instruction location to MAR.
2       M[MAR] → MBR        Put instruction in MBR.
3          MBR → IR         And then put it in the IR.
4       PC + Ilen → PC      Bump the program counter to next instruction.
        decode
        execute:
5          ACC → ACC        Complement value in ACC.
6       ACC + 1 → ACC       And then increment it.
```

All instructions start as does this one, with the fetch cycle. The address from the program counter, which identifies the location of the next instruction to execute, is placed in the memory address register (step 1). The value pointed to by this address is fetched from memory (step 2), and placed in the instruction register (step 3). The machine then readjusts the program counter to point to the next instruction (step 4). To correctly do this, the machine must be aware of the length of the instruction. That is, it is possible that machines have instructions of different length, and when the program counter is adjusted to identify the next instruction, the amount of that adjustment (Ilen) is information which is associated with the instruction. For example, the 68020 has instructions ranging in length from 2 to 14 bytes.

The actual work of the instruction is accomplished by steps 5 and 6 above: the value in the accumulator register is fed to the arithmetic/logic unit, where it is first complemented and that result is then incremented. In general, the exact steps utilized to do the work of an instruction depend on the capabilities of the ALU in the system. (Alternatively, the capabilities of the ALU can be based on the requirements of the instruction set.) Usually two iterations through the unit will not be needed. However, this is a good example of some of the possible methods that can be used to accomplish work: the system resources are used as required to complete the tasks of an instruction. These transfers are coordinated by the control unit in agreement with the technology demands of the system.

The subtract instruction requires two operands. One question is the order of operands: which should be the subtracted value? We will assume that the instruction SUB X means, subtract the value stored in the location X from the value currently in the accumulator and store the result in the accumulator. We will further assume that the address X is adequately contained in the instruction itself, so no additional information beyond the instruction will be required. With those assumptions, a set of data transfers that will perform the work of the subtract instruction follows:

```
        fetch:
1           PC → MAR        Instruction location to MAR.
2       M[MAR] → MBR        Put instruction in MBR.
3          MBR → IR         And then put it in the IR.
4       PC + Ilen → PC      Bump the program counter to next instruction.
        decode
        execute:
5           X → MAR         Put address X in the MAR.
6       M[MAR] → MBR        Value at memory address X to MBR.
7      ACC - MBR → ACC      Subtract it from value currently in ACC.
```

The fetch cycle of this instruction is identical to the other fetch cycles: get the instruction and bring it into the instruction register, then bump the program

counter. The real work begins in step 5, where the address of the operand is transferred to the **MAR.** The intended operand of the instruction, the value stored at location X, is then transferred (step 6) to the memory buffer register. Since the address is contained in the instruction, the value of X needed for step 5 can come from either the instruction register or the **MBR.** Finally, the value is subtracted (step 7) from the value currently in the accumulator, and the result left there. This mechanism for doing the subtraction assumes a more capable ALU than did the negate instruction above. If the **ALU** needed to form the negative of the value in the **MBR** by a complement and increment fashion, then additional operand storage facilities would be required.

A number of variations of this method have been made, while the machines have remained basically single address machines. The IAS, Von **Neumann's** machine built in **1946-7,** utilized a word length of 40 bits. The word length was capable of storing more information than required for a single instruction and a single address, so two 20-bit instructions were placed in a single 40-bit word. Each instruction was composed of an **8-bit** op code and a 12-bit address; up to 256 operations could be specified, and, if needed, the single address could identify one of 4,096 data locations. But although each word of storage was capable of handling two addresses, the instruction format was limited to a single address per instruction. The restriction of 12 bits for an address in the single address machine of IAS was not restrictive since the total addressable memory was only 4,096 words. However, this limit is generally not acceptable, so different mechanisms have been implemented to extend the permissible range of the operands.

One of the mechanisms utilized for storing addresses needed to identify the location of operands is to place them directly after the operation code (op code) that identifies the work to be done. This method has several advantages that make it an attractive alternative. If there is no need for an address (such as the negate instruction above), then no **room** is taken up in the instruction itself for a value (address), which will not be used. If multiple length addresses are permitted, that is, addresses of 1, 2, or more bytes depending on addressing mechanism, then only the requisite number of bytes after the op code are utilized to identify the address. And after the fetch portion of the instruction the program counter identifies the location of the address itself. An **RTL** implementation of this type of subtract instruction is shown in Figure 4.3. Notice the change that results if the assumption is made that the operand address is located in the instruction **stream** directly following the bits specifying the instruction.

The **RTL** included in Figure 4.3 indicates that the program counter is used twice, once for the address of the instruction to be executed, and once for the address of the operand. In the first instance, it was incremented by the **length** of the instruction; in the second, it was incremented by the length of the address. We **are** making the assumption here that the decoding of the **instruction/address** identified the appropriate lengths and treated the program counter appropriately. By separating the op code fetch from the address fetch in this manner, the number of bits needed to specify the operation is allowed to expand to meet the appropriate requirements.

Example 4.1: RTL and timing calculations for ADD: How much time is required to execute an **ADD** instruction for a machine organized as demonstrated above?

The time required for execution of the instruction will include the time necessary to obtain the instruction from memory, decode it, and **exe-**

fetch:

1	PC → MAR		Instruction location to MAR.
2	M[MAR] → MBR		Put instruction in MBR.
3	MBR → IR		And then put it in the IR.
4	PC + Ilen → PC		Bump the program counter to next value the PC will then point to memory location holding the address of the operand.
	decode		
	execute:		
5	PC → MAR		PC is needed again.
6	M[MAR] → MBR		Address at this location to MBR.
7	MBR → MAR		This is address of operand.
8	M[MAR] → MBR		And this is operand.
9	PC + Alen → PC		Now bump PC by length of address.
10	ACC − MBR → ACC		Subtract it from value currently in ACC.

Figure 43. RTL Implementation of a Subtract Instruction for a Single Address Machine.

cute the necessary steps. To determine the time required for instruction execution, we must first develop an appropriate RTL implementation of the operations. One such implementation is shown in Figure 4.4.

Each of the items involved in Figure 4.4 will take time to accomplish, and the time for the operation will be implementation dependent. We will assume for the purposes of this example that the accesses to memory cost 300 nsec, the access to a register cost 50 nsec, and that the add itself can be done in 100 nsec, not including the register time. The amount of time for each of the operations identified above is given in the RTL itself. Note thal we have assumed that the bumping of the PC can be done in the time it takes to load the register. Also note that step 10 accounts for both the add time and the register delay time. With these figures, we can see that the total time is 1.1 μsec. The instruction fetch itself requires 450 nsec, which is almost half of the total time. If we look only at the time metric, we can draw some conclusions concerning the efficient use of time to accomplish

fetch:

1	PC → MAR	(50)		Start by loading MAR.
2	M[MAR] → MBR	(300)		And get instruction
3	MBR → IR	(50)		into the IR.
4	PC + Ilen → PC	(50)		Bump the program counter. PC now points to address of operand.
	decode			
	execute:			
5	PC → MAR	(50)		PC is needed again.
6	M[MAR] → MBR	(300)		This is really address of operand.
7	MBR → MAR	(50)		So put in MAR.
8	M[MAR] → MBR	(50)		And get operand to MBR.
9	PC + Alen → PC	(50)		Now bump PC by length of address.
10	ACC + MBR → ACC	(150)	(1100)	ADD value in MBR to value in ACC.

Figure 4.4. RTL Implementation and Timing Considerations an ADD Instruction.

the work of the system. If we now ask how many bits are required. and what are the costs involved in storing and moving data, a different type of conclusion may be available. However, this demonstrates that the fetch of an instruction from memory is definitely not free. It also demonstrates one of the mechanisms that can be used to obtain information about the execution time for instructions.

One of the single address machines built in the mid-1960s that enjoyed wide popularity was the **PDP** 8, made by Digital Equipment Corporation. This was a 12-bit machine, a block diagram of which is shown in Figure 4.5.

The instruction format called for a 3-bit op code, which left 9 bits of the 12-bit instruction for the address. With 3 bits for specifying the action of the instruction, the possible operations were limited to 8, and these 8 were chosen with care. One of them was an **ADD** instruction, which added to the accumulator the value identified by the single address included in the instruction. The 9 bits of address specification in the instruction limited the number of addressable operands, so different operand specification mechanisms, such as indirect addressing, were used to increase the number of accessible values. We will discuss alternative addressing methods in Section 4.4. The instructions that required an address for operand identification, such as **DCA** (deposit value currently located in the **ACC** to the memory location identified and clear accumulator). **TAD** (two's complement add), and **ISZ** (increment and skip if zero), used the 9 address bits to specify the location of the operand. Instructions that did not require an address, such as CLA (clear accumulator), **INA** (increment accumulator), and CLE (complement accumulator), expanded on one of the eight available op codes to specify the action to take place.

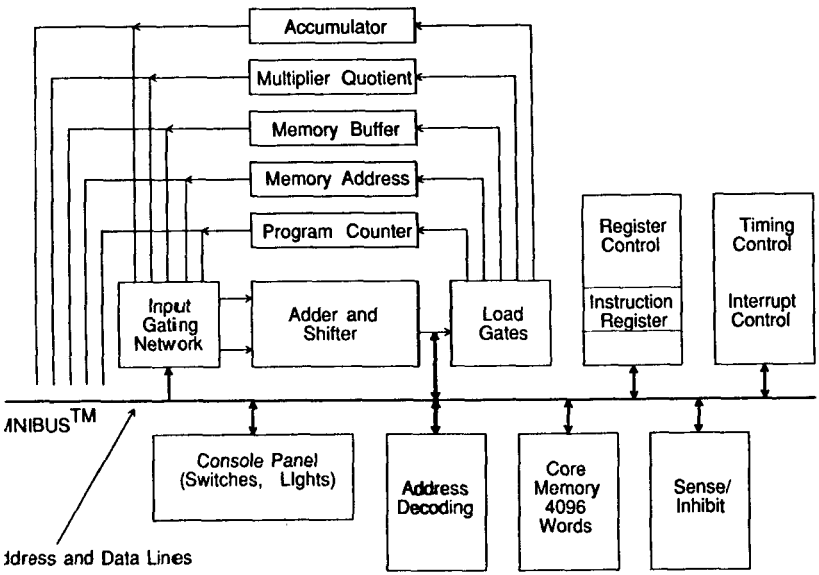


Figure 4.5. Block diagram of the PDP 8 Computer.

The format of the PDP 8 instruction set is given in Figure 4.6. At the time of the creation of the system, memory was a very expensive system resource, and hence the word length was limited to 12 bits. As the relative costs of memory and other system components change and diminish, uses of those system resources will also be appropriately change. The designers of the PDP 8 system, with a limited number of bits to work with, chose the operations of the system with care. A 3-bit op code limited the number of instruction patterns to 8. Six of the possible instructions required an address, and this address was determined by the 9 LSBs of the instruction. These six instructions were logical AND, add, increment-and-skip-if-zero, deposit-and-clear-accumulator, subroutine jump, and unconditional jump. Another of the eight patterns identified an UO instruction, and the remaining 9 bits specified one of 64 I/O devices, and one of eight operations. The operations were defined by the design of the UO device itself. The final pattern identified instructions that needed no address, and hence could all share a common instruction code in the op code bits. This allowed a number of operations to be specified, such as clear the accumulator, or increment the accumulator, and so on. One of the most challenging tasks facing a computer architect is to identify the instructions to be incorporated into a new machine, and then encode the specification of those instructions in a format acceptable for the new system. We will examine some more examples of instruction formats later in this chapter, each of which demonstrates a different view of the optimal utilization of system resources.

All of the above examples have a common operational mode: the instruction stream provides a single address, and this address is utilized to identify the location of an operand. For data operations, that operand is used in conjunction with whatever is needed in an assumed location (the accumulator), and the result is left in a predefined place, usually the accumulator. With this type of a machine all of the operations needed by a system can be performed, but the result may not be as efficient as desired. With the fetch utilizing a large fraction of the instruction time, one approach would be to try to utilize more effectively the information fetched from memory. One method proposed for this is to make the system more

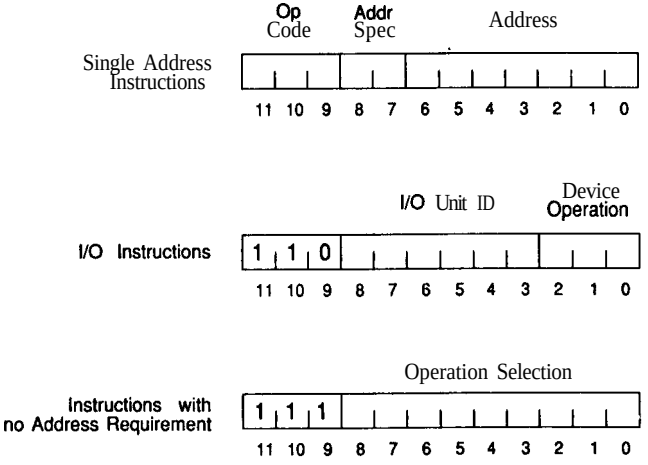


Figure 4.6. Instruction Formats for the PDP 8 Computer.

efficient by using more than one address in a single instruction to specify a greater variety of operations and operands.

4.3. Multiple Address Instructions

Multiple address instruction formats carry with them both benefits and added specification requirements. With a single instruction more operations art identified, so fewer instructions are required to implement a string of arithmetic. At the same time, the instructions must identify all of the work to do, since no assumptions will be made concerning the location of the data. Thus, multiple address instructions will identify both source and destination of the information. The myriad possibilities are exemplified by the following formats:

```
ADD2   A,B
ADD3   A,B,C
```

Although the system architect can choose any reasonable specification mechanism. the assumption we will make concerning the syntax of these instructions is that the final address specified is the destination of the function. With this assumption, the **ADD2** instruction adds the value in the location identified with the **A** address to the value in the location identified by the B address, and the result is returned to the location specified by the B address. Thus, this instruction changes the value identified by the B address. The **ADD3** instruction obtains the operand identified by address A. adds to it the value stored at the location specified by address B, and places the result at the location identified by address C. In a machine that utilizes this type of capability. the op codes must differentiate between the various types of operations.

That is. a separate code must be available for each instruction; **ADD2** and **ADD3 will** be specified by different patterns. This results in a larger operation code field, since many different codes must be representable. And it also results in different length instructions. since some instructions will require three addresses, while others will require only two. Consider the following example, in which we compare two and three address add instructions.

Example 4.2: Two ond three address instructions: Compare the operation of the **ADD2** and **ADD3** instructions, using the times identified in Example 4.1. Assume that the operation codes require the same number of bits to represent as the addresses. (Is this a valid assumption?) What is the execution time required for each of the instructions?

In order to address these questions, we need to identify some of the details of the system. That is, before the RTL of implementation can be determined, we need to understand what mechanisms are being utilized. Let us assume that the first value obtained from memory at the location identified by the PC is the appropriate op code, and that the next values are the respective addresses. This is somewhat simplistic, as we shall see a little later. But it will help to identify some of the underlying issues. The next problem to be dealt with needs a little more detailed consideration. This consideration is the mechanism for the addition: how is it to be carried out. In order to visualize the transfers necessary and the order of events, we need to know the available registers and their interconnection. The basic elements required for this example are given in Figure 4.7. The figure has

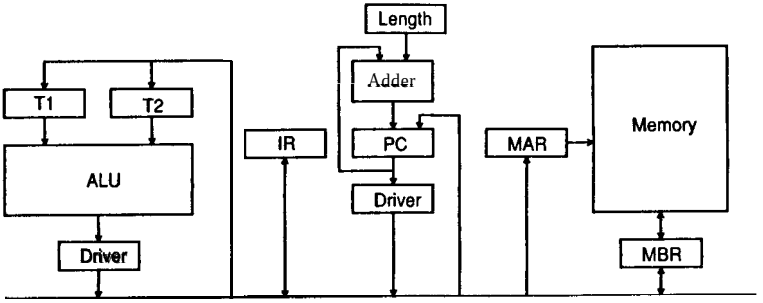


Figure 4.7. Block Diagram of System for Example 4.2.

two registers, TI and T2, which are not pan of the instruction set architecture. That is, the system as defined by the instruction set does not include these storage elements. However, they are very useful when doing instructions that require holding information to be utilized by the system. Armed with this knowledge about the underlying **structure,** let us examine RTL representations of the instructions.

The **RTL** statements describing one implementation of the two address ADD instruction is found in Figure 4.8. The figure also contains timing information with the RTL statements, indicating the time required to complete the task.

The RTL for the three address case is included in Figure 4.9. Note the similarity with the two address version in the initial stages of the instruction

	fetch:		
I	PC → MAR	(50)	Address of instruction to MAR.
2	M[MAR] → MBR	(300)	Instruction to MBR.
3	PC + Ilen → PC	(50)	Bump the PC to point at address.
4	MBR → IR	(50)	Instruction finally to IR.
	decode		
	execute:		
5	PC → MAR	(50)	Go get address of operand.
6	PC + Alen → PC	(50)	Bump PC to point at next address.
7	M[MAR] → MBR	(300)	This is address of first operand.
8	MBR → MAR	(50)	So put in MAR.
9	M[MAR] → MBR	(300)	And get the value there. first to MBR.
10	MBR → TI	(50)	And then to TI.
11	PC → MAR	(50)	This is to get address of second operand.
12	PC + Alen → PC	(50)	Bump PC to next instruction.
13	M[MAR] → MBR	(300)	Address of second operand to MBR.
14	MBR → MAR	(50)	And then to MAR.
15	M[MAR] → MBR	(300)	The second operand goes to MBR.
16	MBR → T2	(50)	And then to T2.
17	TI + T2 → MBR	(150)	Do the add. results to MBR.
18	MBR → M[MAR]	(300)	Put results where operand two used to be.
		(2500)	Total time: 2.5 μsec

Figure 4.8. RTL Implementation of a Two Address ADD Instruction.

efficient by using more than one address in a single instruction to specify a greater variety of operations and operands.

4.3. Multiple Address Instructions

Multiple address instruction formats carry with them both benefits and added specification requirements. With a single instruction more operations are identified. so fewer instructions are required to implement a string of arithmetic. At the same time, the instructions must identify all of the work to do, since no assumptions will be made conceming the location of the data. Thus, multiple address instructions will identify both source and destination of the information. The myriad possibilities are exemplified by the following formats:

```
ADD2   A,B
ADD3   A,B,C
```

Although the system architect can choose any reasonable specification mechanism, the assumption we will make concerning the syntax of these instructions is that the final address specified is the destination of the function. With this assumption, the ADD2 instruction adds the value in the location identified with the A address to the value in the location identified by the B address, and the result is returned to the location specified by the B address. Thus, this instruction changes the value identified by the B address. The ADD3 instruction obtains the operand identified by address A, adds to it the value stored at the location specified by address B. and places the result at the location identified by address C. In a machine that utilizes this type of capability, the op codes must differentiate between the various types of operations.

That is. a separate code must be available for each instruction; ADD2 and ADD3 will be specified by different patterns. This results in a larger operation code field, since many different codes must be representable. And it also results in different length instructions, since some instructions will require three addresses, while others will require only two. Considei the following example, in which we compare two and three address add instructions.

> Example 4.2: ***Two and three address instructions:*** Compare the operation of the ADD2 and ADD3 instructions, using the times identified in Example 4.1. Assume that the operation codes require the same number of bits to represent as the addresses. (Is this a valid assumption?) What is the execution time required for each of the instructions?
>
> In order to address these questions. we need to identify some of the details of the system. That is, before the RTL of implementation can be determined, we need to understand what mechanisms are being utilized. Let us assume that the first value obtained from memory at the location identified by the PC is the appropriate op code, and that the next values are the respective addresses. This is somewhat simplistic, as we shall see a little later. But it will help to identify some of the underlying issues. The next problem to be dealt with needs a little more detailed consideration. This consideration is the mechanism for the addition: how is it to be carried out, In order to visualize the transfers necessary and the order of events, we need to know the available registers and their interconnection. The basic elements required for this example are given in Figure 4.7. The figure has

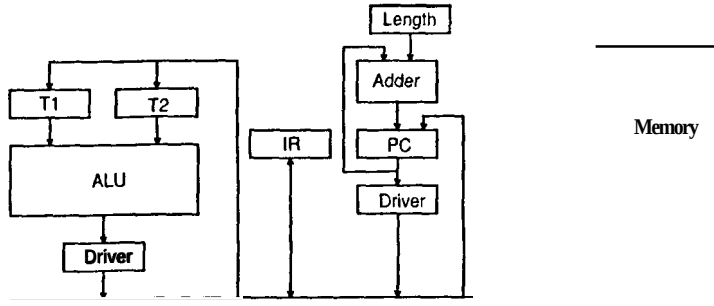


Figure 4.7. Block Diagram of System for Example 4.2

two registers, T1 and T2, which are not part of the instruction set architecture. That is, the system as defined by the instruction set does not include these storage elements. However, they are very useful when doing instructions that require holding information to be utilized by the system. Armed with this knowledge about the underlying structure, let us examine RTL representations of the instructions.

The RTL statements describing one implementation of the two address ADD instruction is found in Figure 4.8. The figure also contains timing information with the RTL statements, indicating the time required to complete the task.

The RTL for the three address case is included in Figure 4.9. Notr the similarity with the two address version in the initial stapes of the instruction

	fetch:		
1	PC → MAR	(50)	Address of instruction to MAR.
2	M[MAR] → MBR	(300)	Instruction lo MBR.
3	PC + Ilen → PC	(50)	Bump the PC to point at address.
4	MBR → IR	(50)	Instruction finally lo IR.
	decode		
	execute:		
5	PC → MAR	(50)	Go get address of operand.
6	PC + Alen → PC	(50)	Bump PC to point at next address.
7	M[MAR] → MBR	(300)	This is address of first operand.
8	MBR → MAR	(50)	So put in MAR.
9	M[MAR] → MBR	(300)	And get the value there, first lo MBR,
10	MBR → TI	(50)	And then to TI.
11	PC → MAR	(50)	This is to get address of second operand.
12	PC + Alen → PC	(50)	Bump PC to next instruction.
13	M[MAR] → MBR	(300)	Address of second operrnd to MBR.
14	MBR → MAR	(50)	And then to MAR.
15	M[MAR] → MBR	(300)	The second operand goes lo MBR.
16	MBR → T2	(50)	And then to T2.
17	T1 + T2 → MBR	(150)	Do the add, results to MBR.
18	MBR → M[MAR]	(300)	Put results where operand two used lo be.
		(2500)	Total time: 2.5 μsec

Figure 4.8. RTL Implementation of a Two Address ADD Instruction

	fetch:		
1	PC → MAR	(50)	Address of instruction to MAR.
2	M[MAR] → MBR	(300)	Instruction to MBR.
3	PC + Ilen → PC	(50)	Bump the PC to point at address.
4	MBR → IR	(50)	Now, instruction to IR.
	decode		
	execute:		
5	PC → MAR	(50)	This is to get first address.
6	PC + Alen → PC	(50)	And bump PC by address length.
7	M[MAR] → MBR	(300)	Now the address to the MBR.
8	MBR → MAR	(50)	And then to the MAR.
9	M[MAR] → MBR	(300)	This is the first operand.
10	MBR → TI	(50)	So put it in TI.
11	PC → MAR	(50)	Now, go get the second address.
12	PC + Alen → PC	(50)	Bump the PC appropriately.
13	M[MAR] → MBR	(300)	This is the address itself.
14	MBR → MAR	(50)	So, put it in the MAR.
15	M[MAR] → MBR	(300)	Now, get the second operand.
16	MBR → T2	(50)	And put it in T2.
17	PC → MAR	(50)	Gotta go get the final address.
18	PC + Alen → PC	(50)	Bump PC to point to next instruction.
19	M[MAR] → MBR	(300)	Get the address of the result.
20	MBR → MAR	(50)	And put in the MAR.
21	T1 + T2 → MBR	(150)	This is actual work of the instruction.
22	MBR → M[MAR]	(300)	Put in location specified by third address.
		(2950)	Total time: 2.95 μsec

Figure 4.9. RTL Implementation of a Three Address ADD Instruction.

implementation. Then, when fetch has been completed and the actual work of the instruction begins, the statements in the RTL reflect the different action of the two instructions.

Since the addresses of the operands are stored in the instruction stream, obtaining and storing information requires two memory references for each value: one to obtain the appropriate address, and another to utilize that address for a fetch or store. Each of these interactions requires time to complete, resulting in seemingly long instruction times, 2.5 μsec for the ADD2 instruction and 2.95 μsec for the ADD3 instruction. As would be expected, the ADD3 instruction takes longer than the ADD2 instruction, since one more address is involved in the operand specification. This requires modifying the PC to point at the address, and an additional memory access to fetch to get the appropriate address. The resource utilization of these instructions can be viewed in number of ways. If one simply looks at the time required for the instruction, then the ADD2 instruction is more attractive than the ADD3 instruction. However, if one looks at the time required to implement a set of operations, such as

$$X = Y \cdot Z + W \cdot V$$

then the differences become more apparent:

With ADD2	*With* ADD3
MOVE Y,X	**AND3 Y,Z,T**
AND2 Z,X	**AND3 W,V,Y**
MOVE W,Y	**ADD3 T,Y,X**
AND2 V,Y	
ADD2 Y,X	

The stream of instructions that utilize the **ADD2** method require **15** memory locations to store and 12.5 μsec to execute; the **ADD3** method requires 12 memory locations, and executes in 8.55 μsec. In contrast to the above methods, a single address implementation of the equation would require **14** memory locations to store, and be executed in 8.95 psec, making similar assumptions about the address storage and execution mechanisms. To more appropriately evaluate the merits of one, two, and three address instruction mechanisms, a more complete set of example instructions and system usage is required.

It is possible to generate examples in which each of the mechanisms discussed thus far — single address machines, two address machines, and three address machines — has a better time characteristic than the other two. Among other things. this indicates that the metric we have chosen for comparison, combined with the underlying assumptions, is not a sufficient test. To make a more realistic comparison, further analysis and additional criteria are required. Nevertheless, the above example illustrates a viable method: when a choice between different alternatives is to be made, a metric is chosen that demonstrates the use of the appropriate system resources, and the associated costs are determined. Caution must be exercised to ascertain that the costs not included in the metrics will not undermine the effectiveness of the comparison.

One observation that could be made concerning the system is that a great deal of the execution time for the **ADD2** and **ADD3** instructions, as shown above, is consumed in fetching addresses of operands and the operands themselves. **A** similar comment can be made concerning the number of bits required to store the addresses: if the range of addresses can be limited in some fashion, the number of bits required for addresses (and hence the entire instruction) can be greatly reduced. For both of these reasons — the time required for operand access and the number of bits needed for address specification — register sets have been included in machines.

The use of a register set reduces the time required for instruction performance. One demonstration of this is to rework Example 4.2, this time assuming that the add instructions deal with values in registers, rather than values that reside anywhere within the memory space of the machine. The block diagram for this example is given in Figure 4.10. Note the similarities and differences with Figure 4.7. The main difference is the inclusion of a set of registers, shown here to contain 8 different storage locations. Thus, to represent the operand location requires only 3 bits, and this field can be incorporated into the instruction format. The net result is a reduction in the number of memory references required by each instruction to get information.

Example 4.3: ADD2 and ADD3 instructions with registers: Again compare the operation of the **ADD2** and **ADD3** instructions, but this time assume that the operands reside in registers, and that the register specification is con-

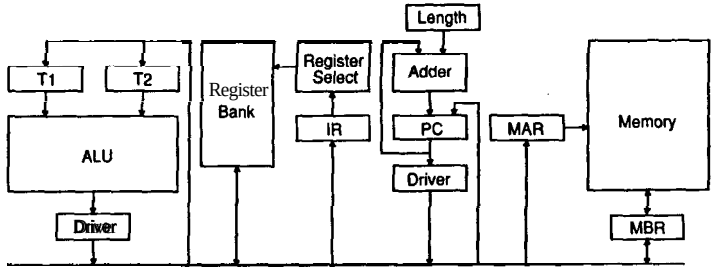


Figure 4.10. Block Diagram of System for Example 4.3.

tained within the instruction itself. That is, an additional memory cycle to obtain addresses is not required, since the identification of the appropriate register is accomplished by using a multiplexer (register select MUX) to select the appropriate bits from the instruction register, as shown in Figure 4.10.

The RTL required for this example follows the RTL for the previous example, with the obvious differences:

ADD2 R_A, R_B		ADD3 R_A, R_B, R_C	
PC → MAR	(50)	PC → MAR	(50)
M[MAR] → MBR	(300)	M[MAR] → MBR	(300)
PC + llen → PC	(50)	PC + llen → PC	(50)
MBR → IR	(50)	MBR → IR	(50)
R_A → TI	(50)	R_A → T1	(50)
R_B → n	(50)	R_B → T2	(50)
TI + T2 → R_B	(150)	T1 + T2 → R_C	(150)
	(700)		(700)

Note that the operands are in the registers, and the resulting instruction times reflect the reduced requirements for operand access. Both instructions now require 700 nsec, but we must recognize that the storage requirements are different for both instructions. That is, the ADD2 instruction must be wide enough to include two addresses, while the ADD3 instruction must be even wider, sufficient for three addresses. If all instructions are to be the same width, it must be the wider of the two formats. That is, if instructions are to be a common width (to match a memory constraint. for example), then the word width must match the widest instruction. For a system utilizing this technique, an instruction that requires fewer than three addresses will waste some of the capabilities of the storage mechanisms. The point is that tradeoffs must be applied to each situation to determine their relative merits, and the choice of the metric will directly impact the comparisons. The metrics may include the number of bits (or bytes) required to store a program segment, the time required to execute, the complexity of the algorithms required to implement the instructions. or any of a number of other appropriate metrics.

The above example demonstrates that the use of registers greatly reduces the time requirements for instructions. As mentioned above, the main reasons for this are the reduced time requirements for interacting with the operands and reduced memory requirements for storing the instruction itself. The reduced time requirements for operand access result from the fact that register access is faster than main store access. The reduced memory requirements are a function of operand identification, since identification of an appropriate register requires a few bits, while identification of a main store address requires a great many more bits. We have used the example of the IAS, which used 12 bits to identify a location in memory; more recent systems. such as the 68030 microprocessor, require as many as 32 bits to specify a location in memory.

A number of existing machines utilize multiple address formats, and we can benefit from an examination of the instruction set architecture of those systems. However, before we consider those machines, we will need to examine a "feature" that we have ignored to this point. The very mechanism that saves time by reducing the memory requirements also reduces to a very small number the allowable locations for operands to reside. However, in general. we would like to be able to access any operand, and operands should be able to reside anywhere in main store. Thus, some mechanisms must exist that will allow operand access to arbitrary locations. Let us examine some of the mechanisms used for operand access.

4.4. Operand Addressing Mechanisms

When an instruction requires an operand for execution, the location of the operand can be assumed. as in the CLA (clear accumulator) instruction, or the operand location can be identified in the instruction itself. In this section we will examine different mechanisms for the specification of the location of the operand. First we will look at direct and indirect addressing, and some variations of indirection that have proved useful in different machines. Then we will look at some of the indexed and register relative modes. Combinations of these mechanisms will provide the versatility needed to identify locations in main store for all types of machine instructions. A visual representation of the addressing mechanisms is included in Figure 4.15 (page 154), and it may be useful to refer to that figure throughout the section.

In our discussion of addressing modes, we are concerned with the manner of specification of the effective address of the operand. That is, how is the location of the operand identified. Thus, we are concerned with the generation mechanism or formula for the effective address (EA).

The term "direct addressing" refers to the situation where the effective address of the operand is supplied directly by the instruction. Thus, for direct addressing

$$EA = A$$

That is, for the ADD2 X,Y instruction. with direct addressing,

$$EA_{OPERAND\ 1} = X$$

$$EA_{OPERAND\ 2} = Y$$

The actual address is contained within the instruction. This is the situation that was assumed for the instructions considered in Example 4.3. As we have

mentioned, various costs associated with this method diminish its effectiveness, so other approachs to operand identification are used. One useful mechanism is to use the information contained in an instruction to identify not the operand, but rather the address of the operand.

The term indirect addressing is applied when the instruction identifies not the operand, but rather the location of the operand. That is, for an ADD2 **X, Y** instruction with indirect addressing,

$$EA_{OPERAND\ 1} = M[X]$$

$$EA_{OPERAND\ 2} = M[Y]$$

The information in the instruction tells the machine where to find the address of the appropriate operand. Different manufacturers have different mechanisms for specifying that a value identified by the instruction is not an operand, but rather the address of an operand. The mechanism we will use is to include an asterisk (*) before the operand specifier. Thus,

```
ADD2 X. Y
```

specifies an instruction that adds a value stored at location X to a value stored at location Y. However,

```
ADD2 *X. *Y
```

specifies an instruction that adds two values: the address of the first value is found at location X, and the address of the second value (as well as the result) is found at location Y. These mechanisms can be combined in instructions, so that

```
ADD2 *X. Y
```

adds the value found in main store at the address found in location X to the value at location Y, and the result is placed in location Y.

The usefulness of indirect addressing is best demonstrated by example. Then a variety of uses becomes apparent, such as accessing arrays in a regular fashion or accessing information in a data dependent fashion.

> ***Example 4.4: Indirect addressing:*** Using indirect addressing and two address instructions, demonstrate a method for adding the elements of three single dimensional arrays together. These arrays are located in main store. and their starting locations are also found in main store at the locations named ARRAY,. $ARRAY_2$, and ARRAY,. The result is to be placed in an array in main store, the starting location of which is in a location named $ARRAY_4$.
>
> We have not yet considered the branching instructions needed for this problem, so those functions will be identified but not specified. Also, we will make the assumption that the information is stored one value per location (instead of double precision or other considerations), so that incrementing an address by one automatically points to the next value. This addition could be performed in a number of ways, but one way is demonstrated by the following instructions.

Set up problem first, then enter this loop:

```
over:   MOVE *ARRAY1, TEMP      Get value from first array.
        ADD2 *ARRAY2, TEMP      And add value from second array.
        ADD2 *ARRAY3, TEMP      And third array.
        MOVE TEMP, 'ARRAY,      Now move answer to right spot.
                                  With the arithmetic over, adjust
                                  the addresses appropriately.
        INC ARRAY1              These increment instructions
        INC ARRAY2                bump each address to point
        INC ARRAY,                to the next value.
        INC ARRAY4
        if nor done. go to over End of loop.
```

Note that the **MOVE** and ADD2 instructions access the information in the arrays indirectlv. Thus, the location identified in the instruction is not the location of the operand, but rather the location where the address of the operand is found. Then the increment instructions, which access their appropriate locations directly, cause the addresses to point to the next elements of the appropriate array. The way in which the above section of code was written modifies the locations $ARRAY_{1-4}$, which is in general not a good idea. A better solution would have been to place these addresses in temporary locations and operate on them in those locations. Another comment that can be made concerns the use of the temporary location. The location would not be needed if the **MOVE** instruction placed the value in the location identified by ARRAY„ and the subsequent ADD2 instructions used that location to sum the value. Thus, the number of memory locations needed for the execution of the program would be reduced. However, indirect references require one more memory reference than direct references, so the required time to complete the code would be increased. Thus, the "best" solution will be determined by which metric is the critical one for the application.

Including both direct and indirect addressing mechanisms in an instruction set allows a wide variety of operand access capabilities. These concepts are directly applicable to systems with register sets, where the identification bits in the address refer to a specific register. Direct addressing in this fashion is sometimes referred to as register direct addressing. An indirect reference occurs when the value contained in the register is an address identifying the location in main store of an operand. This would then be register indirect addressing, and operates in the same fashion as the indirect addressing mentioned above. The benefits af this mechanism have already been identified: the number of bits required to specify the address are reduced, and the time required for register access is much less than that required for main store access.

Example 4.5: Cost of direct and indirect addressing: Determine the times for the ADD2 instruction using direct and indirect addressing. Compare the system of Figure 4.7, which doesn't have a register set, with the system of Figure 4.10, which includes a general register set.

The times required for these instructions can be obtained only if we know the set of register transfers required to accomplish the work of the additions. So, the first step is to obtain the RTL of instruction implementation. First we will look at the system without registers, then observe the

effect when a register set is available. The direct addressing implementation of the ADD2 instruction is shown in Figure 4.11. The transfers required to perform the work consume a total of 2.5 psec. Of that time, 0.450 psec is required for fetching the instruction, the other 2.05 psec is used in execution. Another view of the time requirements comes from examining the time used by the memory interaction. There are six memory transfers, one for the instruction and five for addresses and operands; these total 1.8 psec. We would expect the indirect addressing example to take even more time, and this is confirmed by examining the RTL of the indirect addressing version contained in Figure 4.12. The indirect addressing system is longer, but only by 0.7 psec. The instruction fetch again took 0.45 μsec, while the eight memory transfers consumed 2.4 psec, or 75% of the total instruction time. This gives an indication of one of the reasons that computer architects have attempted to reduce the memory interaction as much as possible. The times involved in the register implementations of the ADD2 instruction indicate how well that can be accomplished.

The work required for register-oriented ADD2 instructions, both for direct and indirect addressing, is demonstrated by the RTL implementations in Figure 4.13.

An examination of the implementations of Figure 4.13 indicates that indeed time is saved when the operands (and/or addresses) are contained in the registers. When the operands are located directly in the registers, then the ADD2 instruction requires only 0.7 psec, 28% of the time required for the memory implementation. The principal contributor is the fact that this implementation requires only one memory transfer, compared to six transfers for the ADD2 X, Y instruction.

ADD2 X. Y *(Direct Addressing)*

fetch:				
PC	→	MAR	(50)	First, address of instruction to MAR.
PC + Ilen	→	PC	(50)	Now bump PC.
M [MAR]	→	MBR	(300)	Retrieve instruction.
MBR	→	IR	(50)	And move to IR.
decode				
execute:				
PC	→	MAR	(50)	This to get address of X.
PC + Alen	→	PC	(50)	Bump PC by length of address.
M [MAR]	→	MBR	(300)	MBR now contains address of X.
MBR	→	MAR	(50)	So put in MAR.
M [MAR]	→	MBR	(300)	And retrieve X.
MBR	→	TI	(50)	Move operand to T1.
PC	→	MAR	(50)	Do same thing for Y.
PC + Alen	→	PC	(50)	
M [MAR]	→	MBR	(300)	
MBR	→	MAR	(50)	
M [MAR]	→	MBR	(300)	
MBR	→	T2	(50)	Move Y to T2.
T1 + T2	→	MBR	(150)	Do the ADD.
MBR	→	M [MAR]	(300)	And store back where Y was.
			(2500)	

Figure 4.11. RTL Implementation of a Two Address ADD Instruction with Direct Addressing.

fetch:			
PC	→ MAR	(50)	As before, address of instruction to **MAR.**
PC + Ilen	→ PC	(50)	Bump PC.
M[MAR]	→ MBR	(300)	Retrieve instruction.
MBR	→ IR	(50)	And move to **IR.**
decode			
execute:			
PC	→ MAR	(50)	This to get address of address X.
PC + Alen	→ PC	(50)	Bump PC by length of address.
M[MAR]	→ MBR	(300)	**MBR** now contains address of address X.
MBR	→ MAR	(50)	So, put in **MAR.**
M[MAR]	→ MBR	(300)	And retrieve address of **X.**
MBR	→ MAR	(50)	Put address of X in **MAR.**
M[MAR]	→ MBR	(300)	And retrieve **X.**
MBR	→ T1	(50)	Move X to **T1.**
PC	→ MAR	(50)	Do same thing for Y.
PC + Alen	→ PC	(50)	
M[MAR]	→ MBR	(300)	
MBR	→ MAR	(50)	
M[MAR]	→ MBR	(300)	
MBR	→ MAR	(50)	
M[MAR]	→ MBR	(300)	
MBR	→ T2	(50)	Move Y to T2.
T1 + T2	→ MBR	(150)	Do the **ADD.**
MBR	→ M[MAR]	(300)	And store back where Y was.
		(3200)	

Figure 4.12. RTL Implementation of a Two Address **ADD** Instruction with Indirect Addressing.

A similar savings is obtained with the register indirect method, also shown in Figure 4.13. The speedup of the register indirect implementation is not as dramatic as the register direct method, but 1.7 μsec is 53% of the time required by the system when no registers are present. Again the difference reflects the extent to which memory is utilized: with registers the instruction required only four memory transfers, while the system without registers required eight memory transfers. The following table summarizes this information:

Addressing Technique	*Memory References*	*Fetch Time*	*Execute Time*	*Total Time*
Direct	6	450	2,050	2,500
Indirect	8	450	2,750	3,200
Register Direct	1	450	250	700
Register Indirect	4	450	1.250	1.700

Note from the table that the instruction fetch time of all of these instructions is identical. For the ADD2 ***X, *Y** instruction, this is only 17% of the instruction time, while for the ADD2 R_X, R_Y instruction, this is 64% of the instruction time. This will form a portion of an interesting observation later in the chapter.

ADD2 R_X, R_Y (*Register Direct Addressing*)

ferch:		
PC ← MAR	*(50)*	Once again. address of instruction to MAR.
PC + Ilen → PC	*(50)*	Bump PC.
M[MAR] ← MBR	(300)	Retrieve instruction.
MBR → IR	*(50)*	And move to IR.
decode		
execute:		
R_X → TI	*(50)*	Get first operand to T1.
R_Y ← T2	*(50)*	Get second operand to T2.
T1 + T2 ← R_Y	*(150)*	And result back to R_Y.
	(700)	

ADD2 *R_X, *R_Y (*Register Indirect Addressing*)

fetch:		
PC ← MAR	*(50)*	Address of instruction to MAR.
PC + Ilen → PC	*(50)*	Bump PC.
M[MAR] ← MBR	(300)	Retrieve instruction.
MBR ← IR	*(50)*	And move to IR.
decode		
execute:		
R_X → MAR	(50)	R_X holds address of first operand.
M[MAR] → MBR	(300)	Retrieve operand.
MBR → TI	(50)	And put in TI.
R_Y → MAR	(50)	R_Y holds address of second operand.
M[MAR] → MBR	(300)	Retrieve operand.
MBR → T2	(50)	And put in T2.
TI + T2 → MBR	(150)	Do the work.
MBR → M[MAR]	(300)	And store results.
	(1700)	

Figure 4.13. RTL Implementation of a Two Address ADD Instruction with Register Direct and Register Indirect Addressing.

The inclusion of registers in the system reduces the time required to perform most functions, as shown in the above example. The register indirect method is a very useful mechanism for identifying the location of operands in main store. In a previous example, we considered the use of indirect instructions to access every element in an array. This type of mechanism is used often enough to justify including a specific addressing mode which handles the incrementing of the address automatically. This is called an autoincrement capability, and is included in many instruction sets. We will indicate that an address is to be incremented after it is used by including a plus sign (+) after the indirect specification. That is, an ADD instruction that uses the indirect autoincrement mechanism for its first operand and register direct access for the second operand would be specified as

ADD2 *R_X+, R_Y

The increment amount used in the instruction is generally tied to the size of the operand. That is, an instruction set may have three different integer add instructions: one for byte, one for word (two bytes), and one for double word (four bytes) operands. The process of autoincrement for these instructions would increase the address by one, two, or four, respectively, for the different situations.

The autoincrement mechanism is very useful for dealing with data in data structures within the computer. For example, one data structure utilized extensively in some types of processing is the stack. Stacks can be created in main store by allocating space for the structure and associating a "stack pointer" address with it. Conceptually, information is placed on a stack and then removed as needed. That is, it is a last-in, first-out mechanism for storing information. We will identify some of the uses of this type of information storage later in this chapter.

Stacks can be constructed by means of a number of methods, but perhaps the most prevalent mechanism is to allow the stack to grow downwards in memory. A POP operation for such a stack is included in Figure 4.14. The address provided by $R_{STACK\ POINTER}$ indicates where the current top of the stack (TOS) is located. This address can identify either the next available location for storing information, or it can specify the location containing the information on top of the stack. For a stack that grows downward in memory, the common mechanism is to utilize an address that points to the value currently on the top of the stack. Notice that the action of extracting information from the stack can be achieved with the register indirect autoincrement addressing mode:

$$\text{MOVE } *R_{STACK\ POINTER}+, R_X$$

The above instruction moves information from the stack to R_X. Since $R_{STACK\ POINTER}$ points to the value currently on the top of the stack, the read action transfers that value from the memory at that location. Then the system automatically increments the stack pointer by the appropriate amount, and at the end of the instruction the stack pointer identifies the next element to be on the top of the stack.

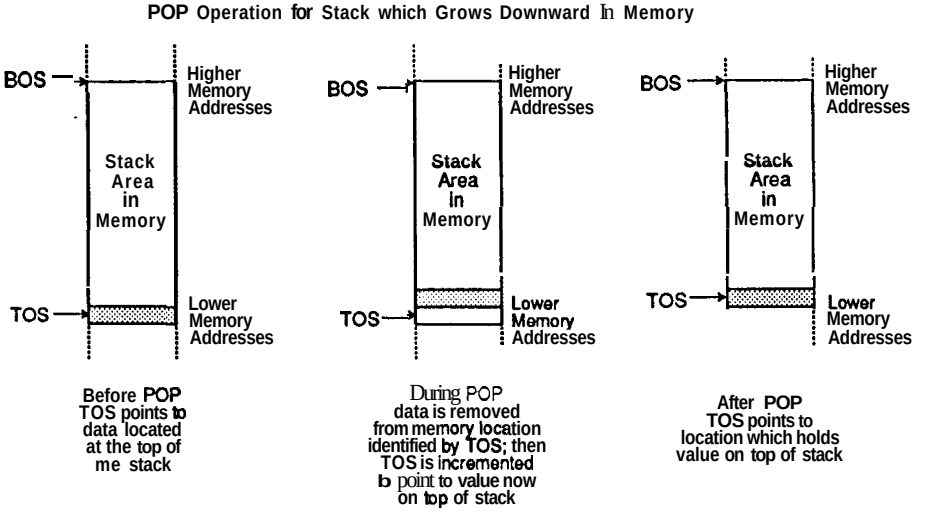


Figure 4.14. Stack Mechanisms in Main Store.

To complete the data transfers needed for stack implementation, we must consider the action required to place information on the stack. Obviously, we will need the ability to decrement the address in $R_{STACK\ POINTER}$. To do this, many instruction sets also include an autodecrement facility. This works in exactly the same fashion as the autoincrement mechanism, except that generally the decrement is done before the address is used, rather than after. In this text we will assume that all autoincrement operations are postincrement operations, and that all autodecrement operations are performed in a predecrement manner. This permits the following pair of operations to be used for stack manipulation:

MOVE R_X, *$R_{STACK\ POINTER}$-	Push value in R_X onto stack.
MOVE *$R_{STACK\ POINTER}$+, R_X	Pop value from stack to R_X.

The first instruction does a push: the address in $R_{STACK\ POINTER}$ is first decremented and then used as an address by the system, and the information in R_X is written to that address in main store. The second instruction is used to pop information from the stack: the address in R_{STACK} m , is used to access the information, and then incremented to point to the next element on the stack. Both the push action and the pop action leave the address pointing at the value on the top of the stack, as expected.

Thus far we have identified direct and indirect addressing, with and without registers, and the idea of autoincrementing (autodecrementing) a value being used as an address. Before we look at some real machines to see in what way these mechanisms are specified and used, three other addressing schemes need to be mentioned.

One mechanism that can be used to access information which is known when a program is created is instruction stream addressing. This mechanism uses the PC to identify data and addresses in the same manner that instructions are identified. As an instruction executes, the information is retrieved from the instruction stream, the location of which is identified by the PC. This method is sometimes called the immediate mode, since data and addresses are "immediately" available for use. In this way, constants (or predetermined addresses) can be included in the instruction stream.

Another method has several names, but we will call it register relative addressing. The basic idea is that a location in main store is specified by identifying an offset from a value in a register. Thus, the effective address of the location will be obtained by adding the two values:

Effective address = Address in register + Offset amount

A common use of this type of addressing is to identify locations in a program relative to the current position of the program counter. This is often called PC relative addressing, and is used extensively for identifying the destinations of branches or jumps in programs. The offset amount is generally included in the instruction itself.

Another use of register relative addressing is to locate information based on the mode of execution of the program combined with the instruction. Some systems have registers that are given specific operating system responsibilities, and references to information are automatically made relative to these registers. An example of this is the 80X86 series of processors made by Intel, which contain four segment registers. The addresses in these registers identify the location in

main store of data, program, stack, and extra segments. Thus, any access to memory is automatically made relative to the appropriate segment register.

Finally, another mechanism for addressing information is indexing. Here, the address of the desired information is the sum of at least two values. One of these values is considered the base value, and can be supplied by the instruction stream or be stored in a register. The second value is usually in a general purpose register. The sum of these two values provides the effective address of the desired location. Thus the base value is "indexed by the value in the register. One example of the use of this mechanism is to provide the base address of an array in the instruction stream, and then to identify the desired element of the array by a value in a register. Indexed references provide an effective way to reference structured data.

It is helpful to visualize the relationship of the various components making up the various addressing modes. A visual summary of the addressing mechanisms described above is shown in Figure 4.15. Additional addressing mechanisms can be constructed by combining the different basic mechanisms to extend the total number of possibilities. We will use many of these addressing mechanisms in examples throughout the text, and a summary of the nomenclature used in the assembly language level examples is included in Table 4.1.

These basic methods are combined in a variety of ways to accomplish the task of identifying in main store a desired location. One of the remaining tasks, which we have not yet discussed, is representing these different choices in a manner that they will be acted upon in a reasonable fashion by the CPU. We know from Chapter 2 that with N bits we can represent 2^N different entities. The problem is to use N bits to specify the operation code (op code) or instruction to perform, the register(s) needed for operand identification, if any, and the appropriate addressing mode. Let us consider two examples of how this has been accomplished by system architects in real machines. First, we will look at some of the mechanisms used by DEC, then the NS32032 processor.

Example 4.6: Encoding of addressing modes: The PDP *11* series of computers utilizes a number of different addressing schemes to identify locations. How are the single and double operand instructions encoded?

The PDP 11 has been one of the most popular 16-bit computers ever built. One of the features of the PDP 11 instruction encoding scheme is that all of the op codes for the instructions fit into 16-bit words, with whatever additional information (addresses, for example) needed occupying additional 16-bit words. To accomplish this, different formats are utilized for those 16-bit instructions. The two of interest to us are those used for single address and double address functions, the formats of which are shown in Figure 4.16. The PDP 11 utilizes eight general purpose registers, which are numbered from 0 to 7. Register 7 is also the program counter, and there are some special modifications to the addressing mechanisms when this register is specified. To identify one of the eight registers requires 3 bits, and locations for these bits are reserved in both representations. An additional 3 bits are used to specify how the register is to be used. Note that using both register and mode bits for the two address format leaves only 4 bits for identification of the instruction. These 4 bits are coded as shown in Table 4.2.

As can be seen from the table, certain patterns in the 4 most significant bits (MSB) expand to consume the bits used for the source

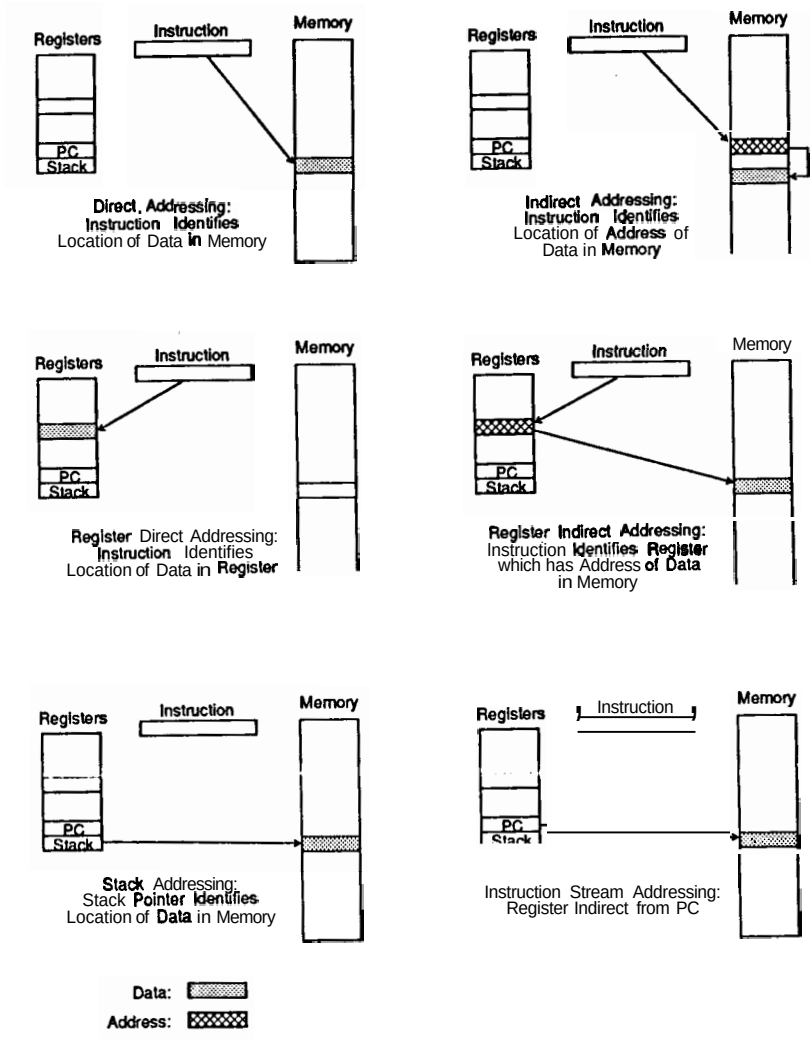


Figure 4.15. Addressing Mechanisms for Accessing Information in Main Store

register (R_S) mode and register identification bits. This allows a few patterns in the MSBs to be utilized to represent many different single address and program control instructions. The bits used to specify the addressing mechanism are coded as shown in Table 4.3.

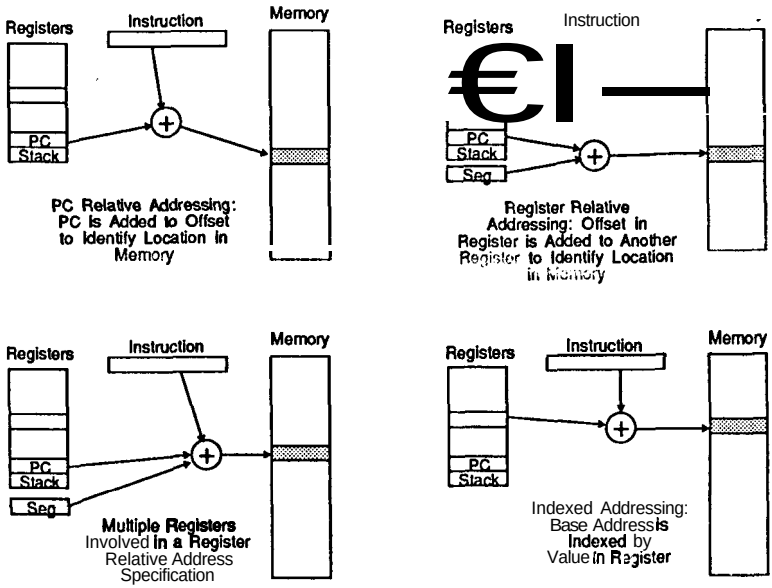


Figure 4.15. *(cont)* Addressing Mechanisms for Accessing Information in Main Store.

The addressing mechanisms detailed in Table 4.3 indicate one approach to operand addressing, an approach that can be extended or modified to meet the needs of a system. The instruction set architecture, as well as the structure of the machine, will reflect the intended use of the

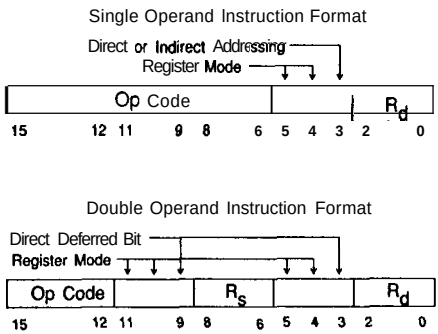


Figure 4.16. PDP 11 Instruction Formats for Single and Double Operand Instructions.

Table 4.1. Addressing Modes and their Nomenclatures.

Addressing Mode	*Represented By*	*Comment*
Direct	@<address>	Address is part of instruction.
Register Direct	Rn	Operand in found in register.
Indirect	*(<address>)	Address is part of instruction; operand is located in memory at that address.
Register Indirect	*Rn	Address found in register; operand in memory at that address.
Instruction Stream	#<value>	Value is stored in instruction stream.
Register Indirect Autoincrement	*Rn+	Register used as address; value in register incremented at end of instruction.
Slack Addressing	Push Pop	Stack pointer indentifies location in main store for transfers; value in stack pointer adjusted as necessary.
PC Relative	$<offset>	Offset identifies target address relative to current location identified by program counter.
Memory-Based Index	(<address> i Rm)	Operand is located in memory at address which is sum of <address>and Rm.
Register-Based Index	(Rn i Rm)	Operand is located in memory at address which is sum of Rn and Rm.

system and the relative importance of system resources: the number of registers, the amount of memory, and the times required for arithmetic, register, and memory interaction. The register direct addressing referred to in Table 4.3 is as we expect: the operand is located in the specified register. And the register indirect uses the specified register as an address pointing to the desired location. The register indirect autoincrement is as described above, the value in the register being used as an address and incremented as part of the instruction. Mode 3 is a multiple use of indirection, with the register being incremented after use; that is, the value in the specified register is used as an address and then incremented. But the address extracted from the register points not to the operand, but rather to the address of the operand. DEC refers to this additional level of indirection as "deferred" addressing. The same thing happens on the autodecrement and two level indirect with autodecrement. The decrementing of the register value is done first, and then the address used, in the first case as the address of the operand, and in the second case as the address of the address of the operand.

Table 4.2. Encoding of Instructions for the PDP 11 Architecture.

Op Code	Function Performed
0 0 0 0	Single address and special function instructions
0 0 0 1	Move instruction
0 0 1 0	Compare instruction
0 0 1 1	Bit test instruction
0 1 0 0	Bit clear instruction
0 1 0 1	Bit set instruction
0 1 1 0	**ADD2** instruction
0 1 1 1	Single address instructions
1 0 0 0	Single address and special function instructions
1 0 0 1	Move instruction (byte)
1 0 1 0	Compare instruction (byte)
1 0 1 1	Bit test instruction (byte)
1 1 0 0	Bit clear instruction (byte)
1 1 0 1	Bit set instruction (byte)
1 1 1 0	Subtract instruction
1 1 1 1	Special purpose instructions

Table 43. Encoding of Addressing Information in the PDP 11 Architecture.

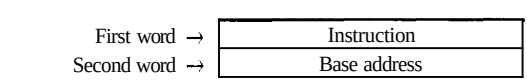
Addressing modesfor PDP 11 operands

Addr bits		Addressing mode
0	0 0	Register direct
0	0 1	Register indirect
0	1 0	Register indirect — autoincrement
0	1 1	Two level indirect. autoincrement register
1	0 0	Register indirect — autodecrement
1	0 1	Two level indirect, autodecrement register
1	1 0	Indexed
1	1 1	Indexed indirect

Addressing modes when PC is target register

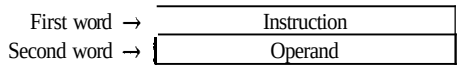
Addr bits		Addressing mode
0	1 0	Immediate mode
0	1 1	PC absolute mode
1	1 0	PC relative
1	1 1	**PC** relative, indirect

The index mode uses the specified register as an index, and a value from the instruction stream as the base. This information is coded as follows:

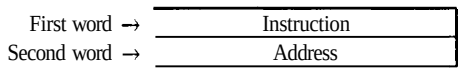


The address of the operand is the sum of the base address and the amount in the register. For the indexed indirect mode, the address resulting from the sum points not to the operand, but to the address of the operand.

Finally, the PC specific addressing modes all require a second value in the instruction stream. The first value is the instruction that identifies the appropriate **PC** addressing mode. The immediate mode is used to supply an operand directly from the instruction stream:



The **PC** absolute mode is used to specify an address directly in the instruction stream:

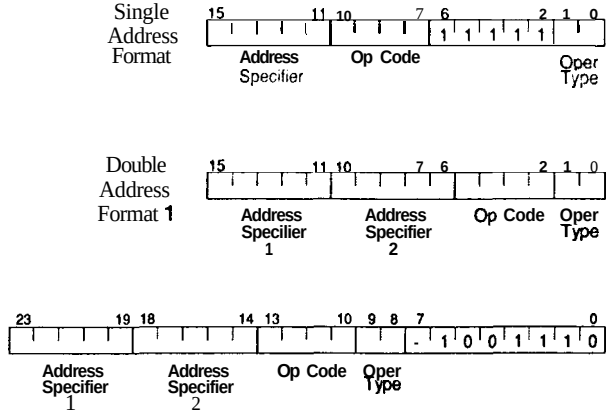


The **PC** relative mode is also coded as above, but the address is relative to the PC (actual address is sum of PC and supplied address). The **PC** relative, indirect mode uses the same mechanism to identify the address of an operand, rather than the operand itself.

When **DEC** expanded on the ideas and concepts of the **PDP** 11 to create the VAX11 architecture, the capabilities of the address mechanism were also expanded. However, the same basic elements are utilized: direct and indirect addressing, indexing, and relative addresses. The number of registers was expanded to 16, and the bits identifying the different addressing modes expanded to 4, so specifying an address required 8 bits. The number of instructions has also been expanded, so that the list includes not only one and two address instructions, but three address instructions as well.

Example 4.7: Expanding op codes: The advances in semiconductor technology have allowed microprocessors to become more and more powerful. One of the 32-bit microprocessors is the **NS32032,** by National Semiconductor. Although the instruction set does not include three address instructions, it does have some interesting capabilities in the addressing mechanisms. How are the one and two address instructions encoded?

Some of the addressing formats for the **NS32032** are shown in Figure 4.17. The processor has several different addressing modes, but the ones that concern us are one and two address formats. An interesting feature of the instruction set is that many of the instructions which are usually associated with a single address are two address instructions for the 32032. The single address instructions are used for functions like JUMP and JSR. The target location is identified by the use of the five address specifier bits. These allow 32 different addressing combinations, some of which use the eight general purpose registers in the processor. Included in the mechanisms are register direct, register relative, register indirect, two level indirect, immediate, absolute, stack, and indexed references. When a displacement or other constant is needed (constant for immediate values. addresses for memory locations), this value directly follows the instruction bits in the instruction stream. The size of an immediate value is determined by the instruction (byte, word, longword). However, an address displacement is composed of 1, 2, or 4 bytes. as shown in Figure 4.17. This allows storing in the instruction stream only the bits needed to identify the target address.



Double Address Format 2

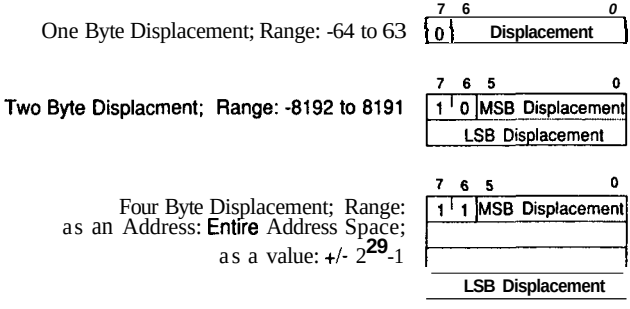


Figure 4.17. NS32032 Formats for Single and Double Operand Instructions.

As mentioned above, the two address format is used not only for adds and subtracts, but also for instructions traditionally considered single address instructions. Thus, the negate instruction extracts a value from one location, forms the negative value by subtracting it from zero, and then places the result, not back in the original location, but rather in a location identified by the destination address. Figure 4.17 includes two of the two address formats; the more often used instructions make use of the shorter format. These include add, subtract, compare, move, and others. The longer formats are used by instructions that do not occur as often, such as divide, test bit, shift, and absolute value. In the two address formats, all of the addressing modes are possible, allowing location of operands in both registers and memory.

The 32032 instruction set is a good example of the concept of the expanding op code. That is, the bits required to specify interaction expand to provide the necessary information. The shortest instructions occupy a single byte; more complex instructions can consume 3 bytes in instruction specification, then more bytes for index, address, and constant specifications.

Both the PDP 11 and the NS32032 provide examples of one and two address instruction sets, as well as providing real examples of a variety of addressing modes. The specification of a target address, whether for operand identification or for program control, can utilize a combination of the basic modes, as we have seen. The decisions involved in selecting the modes to include in an instruction set reflect the design philosophy of the system architects. The basis for those decisions is formed by the intended application, the resources available (time, power, chip area, *etc.*), and the targeted system goals. Before we examine some of those issues let us look at another approach: machines that use instructions with no address specification.

4.5. Zero Address Machines: the Use of Stacks

A stack is a last-in, first-out storage mechanism, where information is stacked up much like pieces of paper. The stack mechanisms described in the previous section are built in main store with appropriate instructions. However, it is also possible to do arithmetic with stacked values: an arithmetic operation is specified, and any needed operands are extracted from the stack. The result of the arithmetic operation is then placed on top of the stack. Because the operands are assumed to be located on the stack, no addresses are needed in the instruction to identify operand location. Hence, this type of system is called a zero address machine.

In addition to the arithmetic or logic instructions that actually cause work to occur, additional instructions are needed to push information onto the stack, and then to pop it off the stack when the arithmetic is finished. The operation of a stack system to do work is demonstrated by a simple example.

Example 4.8: Arithmetic with a stack: Consider the expression

$$F = A + (B \times C + D \times (E / F))$$

Give a set of stack-oriented instructions that will calculate the expression.

This could be done in several ways; we will mention two. These are listed below, assuming that the machine can perform push, pop, add, divide, and multiply operations.

```
PUSH A          PUSH E
PUSH B          PUSH F
PUSH C          DIV
MULT            PUSH D
PUSH D          MULT
PUSH E          PUSH B
PUSH F          PUSH C
DIV             MULT
MULT            ADD
ADD             PUSH A
ADD             ADD
POP F           POP F
```

Notice that the only instructions that require addresses are the push and pop instructions. All of the other instructions merely indicate the action to take place. In fact, some stack systems use push and pop instructions that do not require addresses, but rather use the value located on the top of the stack as

the address of the target location of the instruction. The ADD instruction. for example, pops two values off of the stack and places on the top of the stack the sum of the two values. Another thing that needs to be pointed out is the depth of the stack. The major difference between the two solutions above is that the stack depth of the first solution (the maximum number of items on the stack) is 5, whereas the stack depth of the second solution is 3. The depth of the stack will have a direct impact on the speed of execution, depending on the implementation of the hardware.

Another thing that needs to be pointed out is that the instructions here will be best implemented if they are of variable length. Note that the push and pop instructions will need to be long enough to include the appropriate address bits, but the arithmetic instructions can be very short, since only action specification is required.

The use of a stack for implementing a variety of functions is very attractive in certain circumstances. The most obvious drawback is that the time required can be great because of moving data to and from the stack, especially since the stacks we have mentioned to this point have been constructed in main store. One solution to this is to construct a special hardware module that places the top elements of the stack in hardware registers. A block diagram of such a module is shown in Figure 4.18. The figure shows four hardware registers forming the top of the stack. Information to be placed on top of the stack (by instructions) comes from the memory. and information popped off of the stack flows to the memory through the memory interface. This module has the responsibility for maintaining the stack pointer and the transfer of information from the appropriate hardware register to/from memory. Organization of a hardware stack control system is an attempt to minimize the interaction with memory, since stack depths of up to four (for the system shown in Figure 4.18) needn't require interaction with memory (except as called for by the instructions being executed). The ALU is shown receiving input from the top two registers. This arrangement allows the ALU to perform needed arithmetic and place the result back on top of the stack, all in a single clock period. The stack control circuitry is then responsible for handling the flow of information within the stack, and between the hardware registers and the memory.

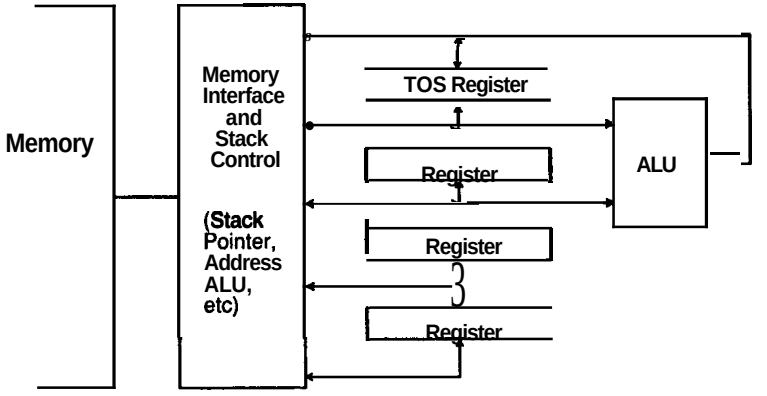


Figure 4.18. Block Diagram for a Hardware-Oriented Stack System.

The use of a stack system within a machine organization allows for some very useful capabilities. Stacks can be effectively utilized for some arithmetic capabilities and also for specific algorithms, such as optimization algorithms in compilers and other software systems. Another example of effective stack usage is parameter passing between routines, since operands needed for a subroutine can be placed on the stack, and then the subroutine is called. The code of the subroutine knows that the operands are located on the stack, so it performs the needed operations and places the results on the stack before returning control to the calling program.

However, there are some drawbacks to the use of the stack, such as saving results for further use. For example, consider the following expression:

$$A = B \times (C + D + E) - F \times (C + D)$$

The C + D portion of this statement can be used twice, and in a register machine this would be straightforward to accomplish. The method of doing this on a stack machine is not so obvious, since only items on the top of the stack can be used for calculations. However, this type of operation is very prevalent in most calculations that a machine will perform. Another operation used extensively in computations involves structured data (arrays, queues, et~.). A calculation like

$$ARRAY\_1[I] = ARRAY\_2[J] + ARRAY\_3[J + I]$$

which involves references into several arrays and address on array subscripts, is handled very naturally on a register machine with the various addressing modes already discussed. These manipulations are not easily accomplished on a pure stack machine. Because of the need to access information in situations such as this, most practical stack machines include capabilities not available in a pure stack machine. These include additional registers for addressing, such as index registers, as well as operand referencing with respect to the top of the stack. The ability to reference information held in the stack (but not at the top of the stack) adds capabilities that can be effectively utilized by a computer system. These various capabilities can allow stack machines to be used for many types of computations.

4.6. Program Control Instructions

The instructions dealt with thus far are instructions required to do work in machines, where work is defined as arithmetic or logic operations. These concepts will also apply to operations that are often not considered as part of the "computing" realm, such as editing or control processing. However, this is not a broad enough definition to cover all of the types of operations that of a machine. Calculation of values covers only one type of operation that computers must provide. In addition to computing, a machine must be able to make decisions, transfer information, and control devices. The instructions oriented toward input/output operations (I/O) will be dealt with in a later section; we now turn our attention to instructions used to control the program flow.

The area of program flow instructions can be divided into two general groups: instructions that change the flow of the program without side effects, and instructions that modify the program counter and also cause additional operations to occur. Examples of the first type of instruction are conditional and uncondi-

tional branches, while the second type of instruction is exemplified by a subroutine call.

The simplest instructions to deal with are those that change the program flow without any side effects. As we have indicated by the RTL representations of instruction execution, the assumed address for the next instruction to execute identifies the location immediately following the current instruction. That is, normal program behavior calls for the program counter to be incremented from one instruction to the next. When the next instruction to execute is not the next one in the memory, then the program counter must be modified accordingly. The program counter must be changed to identify to the appropriate instruction to be fetched. We will follow the terminology used by many manufacturers that a program counter change that uses direct addressing mechanisms is called a jump, and a program counter change that identifies its target address as an offset from the current location (PC relative) is a branch.

The **jump/branch** instruction is very straightforward: the target address is identified, and the program counter is changed accordingly. The target address can be specified by combinations of the various addressing modes that we have already identified. The system operation changes somewhat when the branch is made conditional. In this situation, the contents of the PC at the completion of the branch instruction is dependent upon some system status condition or on some comparison identified by the instruction. The conditions may include the status bits contained within the status register of the machine; some arithmetic possibilities were identified in Sections 3.2 and 3.7. Other conditions found in status registers reflect the status, not of the arithmetic operations, but rather of the entire system. These include such information as **interrupt** information, errors and traps that have occurred, semaphores used in synchronizing system resources, and any other information that details the state of the system.

In the definition of the system architecture, the designers of the system must determine the information to be included in the status register, as well as the possible conditions that will be **testable** with the instructions defined in the instruction set. Two different examples of the approaches that can be taken are available in the **VAX** architecture (from Digital Equipment Corporation), and the **MIPS** architecture (from ***MIPS*** Computer Systems). Both systems include a 32-bit system status word; however, the information contained within the status word is different for both systems. The **VAX** status word contains bits that reflect arithmetic conditions, while the status register of the **MIPS** system does not contain results of arithmetic operations. The **VAX** system, which is an architecture based on a complex instruction set philosophy, has over 35 instructions to test various combinations of bits in the status register. The **MIPS** system, on the other hand, has eight conditional branch instructions, two of which compare two general purpose registers (equal, not equal), and the rest of which check conditions of a single register (equal to zero, not equal to zero, positive, etc.). The **MIPS** system is an example of the reduced instruction set approach to machine design, which we will discuss in Section 4.8.

Regardless of the type of instruction set architecture chosen for a particular system, if the proper conditions are satisfied, the PC contents are modified to allow the program to continue at an address identified by the instruction. If the conditions are not satisfied for modifying the program flow, then the program counter is incremented in the normal fashion and execution of the program continues with the next instruction in the normal order of execution. These program counter modification mechanisms are demonstrated by the following example.

Example 4.9: Jump and branch instructions in a PDP 11 type architecture: In the PDP 11 architecture, jumps can use any of the appropriate addressing modes to identify the target address. Assuming that the target address is included in the instruction stream, give the RTL for a jump instruction. Also, give the RTL for an instruction that branches if the carry is set. The branch instruction on the PDP 11 encodes the target address as an offset from the PC, and 8 bits are included in the instruction to specify the offset. Since PDP 11 instructions must be on even word boundaries, the offset is multiplied by two before it is added to the PC.

The jump instruction required by the example must retrieve the target address from the instruction stream and move it to the PC. This can be accomplished as follows:

```
fetch:                  Go get the instruction
      PC → MAR          Address of instruction to MAR.
  PC + 2 → PC           Bump PC to point to address.
M[MAR] → MBR            Retrieve instruction.
   MBR → IR             And move to IR.
decode                  Control systemfigures out what to do
execute:                And begins the proper action:
      PC → MAR          Go get the target address.
M[MAR] → MBR            And put in MBR.
   MBR → PC             This is actual modification of PC.
```

As seen by the RTL, this is a very simple instruction, and because of its simplicity it can be done relatively fast. Nevertheless, time is required for each of the steps. For the instruction mechanism shown in the example, two memory fetches are required, one for the instruction and one for the address. For that reason, the branch instruction is often a desirable alternative, since the target address is identified with respect to the PC, and the offset is included in the instruction. Consider the RTL for the instruction that branches if the carry is set:

```
fetch:
          PC → MAR          Address of branch instruction to MAR.
      M[MAR] → MBR          Retrieve instruction.
         MBR → IR           And move to IR.
decode
execute:
    if (carry == 1) {       Check the condition: if satisfied, then . . .
PC + (2 x IR<7:0>) → PC     Next instruction is at target address.
          } else {          Also, IR<7:0> is sign extended to 16 bits.
        PC + 2 → PC         Change PC address if condition not true.
              }
```

To perform the work of the instruction it is necessary to be able to selective execute the appropriate transfers. That is, the control section sets the PC to PC + 2 or to PC + offset depending on the appropriate condition, which in this case is the contents of the carry bit. The manner in which the arithmetic is done will be system-dependent; however, the logic required to increment the program counter will be available to be utilized as needed. See Figure 4.10 for an example of a system where a separate adder is used to add the appropriate length to the PC. Including a multiplexer to select

either the instruction length of the offset, based on the selected condition, would permit the necessary decision to be made.

As the above example indicates, the instructions that modify the program counter do so in a manner that reflects the capabilities of the machine and the instruction set. The target address is identified, by whatever combinations of addressing modes are available, and the specified address is placed in the program counter. In the case of conditional execution, the necessary condition is tested, and then the appropriate action is taken. We have indicated a simple choice, where the program counter goes to the next instruction or to a different target address. However, more complicated mechanisms can be set up in a system. For example, one minicomputer used a three-way branch for its arithmetic tests: three target addresses followed an arithmetic conditional branch instruction. A different target address was used for the greater than, equal to, and less than arithmetic conditions.

The above example also contains an anomaly when compared to other RTL descriptions of instructions included in this chapter. The fetch portion of the conditional branch did not include incrernenting the program counter to point to the next instruction. Historically, the modification of the program counter to identify the location of the next instruction has been done in the fetch portion of the instruction. For example, many 16-bit computers configure all instructions to occupy one 16-bit word. Then, in the fetch portion of an instruction the program counter is incremented by two bytes. If the instruction needs an immediate value or an address in its execution, the PC then points to this value, and the execution portion of the instruction will obtain this value and increment the PC accordingly. Thus, by the end of the instruction execution, the PC does indeed point to the next instruction to execute.

The conditional mechanisms provided in instruction sets reflect the intended use of the systems. For example, some instruction sets will contain a dedicated CASE instruction (see the NS32000 system) that facilitates decisions requiring a multiway branch capability. Other systems will use combinations of instructions to perform this function. Another example is the use of a special LOOP instruction, such as used in the I80X86 system, to simplify implementation of loops. This instruction decrements a register and branches to a target address unless the result of the decrement is zero.

In the definition of a computer system, the system architect must decide the mechanism for PC relative references, and then maintain consistency in the application of the methods to all instructions. One of the most natural mechanisms is to identify the offset from the address of the instruction itself for all PC relative references, both references for additional values obtained from the instruction stream and references for other locations with an address that is specified with respect to the PC. One mechanism used to implement this technique (address specification from instruction address) is to delay updating tht PC until the end of the instruction. Otherwise, some other method must be utilized to adjust the references made during instruction execution to account for the continued incrementing of the PC. A different approach is to make all PC-relative references made with respect to the contents of the PC as it adjusts itself during the execution of the instruction. The two approaches result in different hardware requirements, with a corresponding difference in programming techniques. Whatever mechanism is selected, the resources of the system (address adders, registers, data paths, etc.) must be used in a reasonable fashion to obtain the desired results.

While the PC modification instructions are relatively simple, the extension of the ideas to linkage instructions brings additional complications. The basic

requirement is that the program flow is changed in such a way that control transfers to another routine, a subroutine, in such a way that program flow can return to the point of departure having accomplished some useful function. The machine then executes the code that follows the subroutine call. The method of accomplishing the subroutine linkage can be very simple or quite complicated. To transfer control in a reasonable fashion, we must create a mechanism that will cause the program counter to change so that instructions are fetched from the subroutine. At the same time, the linkage mechanism must provide a way to retum to the calling routine. There are a number of methods which are used to provide this facility; we will describe three.

One subroutine calling sequence used by a few machines in the mid-1960s is to have the subroutine itself remember from which address it was called. The PDP 8, a 12-bit machine, used the first location of the subroutine to store the return address. The action would then proceed somewhat like:

fetch:	
PC → MAR	Start instruction.
PC + 1 → PC	Bump PC to point at next instruction.
M[MAR] → MBR	Get instruction from memory.
MBR → IR	And transfer to IR.
decode	
execute:	
IR → MAR	This assumes address contained in instruction.
PC → M[MAR]	Put return address in first location of subroutine.
IR → PC	Now put same address in PC.
PC + 1 → PC	And bump it to point to next location, which is actually the first instruction of the subroutine.

At the completion of the transfers outlined above, program execution proceeds in the new routine. The address required to retum to the original (calling) code has been stored in memory with the subroutine. The return from this subroutine mechanism is accomplished with an indirect jump. That is, the target of the jump is the address stored at the beginning of the subroutine. The last instruction of the subroutine identifies the first location of the subroutine, fetches the address stored there, and jumps to that address. The RTL for this action would be:

fetch:	
PC → MAR	Get the instruction.
PC + 1 → PC	Bump PC; this value not actually used.
M[MAR] → MBR	Get instruction (which is a jump) from memory.
MBR → IR	And transfer to IR.
decode	
execute:	
IR → MAR	This assumes address contained in instruction. Address identifies first location of subroutine.
M[MAR] → MAR	Get return address from first location of subroutine.
M[MAR] → PC	Which is actually address to return to; put in PC and program continues at instruction after subroutine call.

This type of subroutine linkage does indeed work, but it has some inherent problems. One problem is the implementation of reentrant code, or subroutines with recursion. Since the return address is stored in a specific location in memory, only one calling routine can utilize the subroutine at any one time.

Thus, a system with more than one user (such as a time-sharing system) would be unable to share code between users. Likewise, a subroutine could not call itself, since in the process of doing so the return address to the original call would be destroyed. Another problem encountered with this mechanism is that the technique does not work in systems that store the programs in read only memory (ROM). And since microprocessors make extensive use of ROM for storing programs this method is particularly unattractive for those systems.

A second type of subroutine linkage involves storing the return address in a general purpose register. Instead of copying the updated program counter (which identifies the instruction after the subroutine call) to a location in memory, it is saved in a general purpose register. This solves the memory and reentrant problems, but not the recursion problem. It does provide a more rapid linkage mechanism. That is, since references to memory are not required to store and retrieve the address, the time required for both the call and the return will be proportionately less. This is the mechanism used by the Texas Instruments 9900 architecture, where the branch and link instruction (BL) places the return address into general register 11. A similar mechanism is used by some instructions in the IBM *370* system.

Perhaps the most extensively utilized method for subroutine linkage is the use of a subroutine stack. Systems that use this method will have a register designated as the stack pointer, which will control a stack in the memory of the machine. A subroutine call will push the return address onto this stack. The return reverses the process, popping the address from the stack to the program counter. This method is very attractive from several aspects, since it provides a solution to the problems identified earlier. The stack is built in memory, which need not be shared with the program, so the program can be in ROM while the stack is in RAM. When multiple users are executing programs on a single computer, then each user will have a private stack space and can share a single copy of the code. Since each call to a routine will push a new return address onto the stack, recursive routines can be utilized as well. For these reasons, the stack method for establishing subroutine linkage is used by many systems.

Example 4.10: Subroutine linkage: The *68020* instruction set architecture has eight data registers ($D_0 - D_7$) and eight 'address registers ($A_0 - A_7$), to which users have access, one of which (A_7) is considered the stack pointer. In addition, there is a program counter. Give an appropriate RTL for the JSR (jump subroutine) instruction, assuming an address register indirect, relative addressing mode to identify the location of the subroutine. Also give an RTL for the RTS (return from subroutine) instruction.

This subroutine linkage instruction mechanism is very straightforward. Note that the following RTL is not necessarily accurate if the *68020* is considered in detail, since that processor has a pipelined implementation that increases the complexity of the system. However, as far as the user is concerned, the action of the JSR instruction can be considered as shown in the RTL included in Figure 4.19.

Notice that since the *68000* series processors utilize a stack that grows down in memory, the adjusting of the stack pointer to put information on the stack is to decrement it; the proper value is then placed at this address. At the end of the operation the SP is pointing at the value at the top of the stack. The address register on the *68020* is 32 bits long, so the SP must be decremented by 4 since the addresses of the system are byte addresses. To pop information off of the stack, the value is first removed, then the stack

RTL for JSR (Jump to Subroutine) Instruction

fetch:	
PC → MAR	Start fetch of instruction.
PC + 2 → PC	All 68000 instructions start with 2 bytes. So point to next value to be fetched.
M[MAR] → MBR	Get the JSR instruction from memory.
MBR → IR	And transfer to IR.
decode	***This must identify addressing mode, etc.***
execute:	
PC → MAR	Since addressing mode needs value from instruction stream.
M[MAR] → Temp	Get the value and store temporarily.
PC + 2 → PC	Bump PC by 2 to identify return address.
SP − 4 → SP	Get stack pointer ready for new addition to stack.
SP → MAR	Set up MAR to identify memory location for return address.
PC → M[MAR]	And push PC, which has return address, onto stack.
Temp + A_N → PC	Now put address of subroutine in PC. Note that A_N is specified by instruction.

RTL for RTS (Return from Subroutine) Instruction

fetch:	
PC → MAR	Start fetch of the return instruction.
PC + 2 → PC	All 68000 instructions start with 2 bytes.
M[MAR] → MBR	Get the RTS instruction from memory.
MBR → IR	Transfer to IR.
decode	
execute:	
SP → MAR	Identify memory location where return address is stored.
M[MAR] → PC	This will be a 4-byte transfer.
SP + 4 → SP	Increment stack pointer to point at next value on stack and the action is completed.

Figure 4.19. RTL Implementations of Subroutine Linkage for a 68020 System.

pointer adjusted accordingly. One such implementation is also included in Figure 4.19. The call and the return, as demonstrated by the RTL of this example, implement a very simple but effective mechanism for linking calling routines with subroutines.

The mechanisms above for providing subroutine linkage control the flow of the instructions from one routine to another. But what has been ignored in the above discussion is the treatment of parameters being passed to and from a subroutine. Several techniques are used in different circumstances, each of which has its relative merits. Rather than discuss the implementation techniques and how they may be used by different language systems, let us discuss some of the instructions included in different machines to help with the problem.

The most obvious methods require no special instructions: leave the operand in a known place, like a register, and call the subroutine. When the subroutine has completed its work, leave the result in a known location and return to the calling routine. The complexities arise when the called routine wants to use general resources, such as registers, but leave those resources unchanged when control is returned to the calling routine. The subroutine can then copy the registers that it will use, do the work, then restore the registers and return. For this reason some

instruction sets include special features to simplify this process. One example is the SAVE instruction of the NS32000 system, which pushes onto the stack copies of the selected registers. These can then be restored at the proper time with a RESTORE instruction that works in the reverse manner, popping values from the stack and placing them in the registers.

Another facility is provided by the LINK facility in the 68000 instruction set. Often when a routine is accessed, it is desirable to provide for it an area in memory for local variables. One way to accomplish this is to use some of the system stack for this purpose. The LINK instruction allocates space on the stack for the routine to use as needed. This stack space can be used not only for local variables, but also for parameter passing between the two routines. The values are accessed by indexing into the stack (with the indexing mechanism provided by the addressing modes) from the current stack pointer.

One of the more complicated mechanisms for linking routines is demonstrated by the CALLS instruction in the VAX architecture. This instruction uses the stack to pass arguments to a routine, with the assumption that the stack has already been modified to contain those arguments. The instruction then needs to know the number of arguments and the address of the target routine. The action of the CALLS instruction begins by pushing a number of arguments onto the stack. The location of the routine being accessed is identified; this could involve combinations of the addressing mechanisms already mentioned. The first 16 bits of the routine being called form an entry mask, which identifies the registers to be saved before the routine can be entered. The stack pointer (SP) is aligned to a 32-bit boundary, and those registers are pushed onto the stack, as well as the program counter (for return address), the frame pointer, and the argument pointer. Then two 32-bit values containing status and mask information are also placed on the stack. Finally, the new frame pointer and argument pointer are set up, and the program counter is set to the location after the entry mask, and control passed to that point. This mechanism performs the work of transferring control to the new routine and providing, via the stack, a parameter passing mechanism.

4.7. I/O, Interrupts, and Traps

The instructions investigated thus far have included mechanisms for doing work (arithmetic and logic instructions), mechanisms for passing information (moves, etc.), and mechanisms for controlling the work (program control instructions). One of the areas not mentioned is the transfer of information to and from external devices. This is generally called input/output processing (I/O), but involves more than transfer of data. Additional requirements include such things as testing of conditions and initiating action in an external device. Some of the I/O programming is in response to an external event signaling the processor that a device needs to be serviced. This signaling process is called an interrupt, and the processor responds to the interrupt in a predetermined fashion. Finally, traps serve much the same purpose as interrupts, but result from conditions detected internal to the processor.

I/O processing has evolved from the very simple capabilities of the first machines to sophisticated mechanisms used in some machines available today. In its simplest form, I/O transfers data to or from an addressed device under the direct control of the processor. This is called programmed I/O. In single address machines, the data is moved from/to the accumulator. For example, the PDP 8 instruction set has input and output instructions that identify one of 64 devices.

Having selected the appropriate device, the system can transfer information in the ACC to the device, information from the device to the ACC, or test a condition in the device. The conditional instruction is similar to the conditional branches already discussed: the order of instruction execution is modified if the proper conditions are met.

This concept of having specific instructions for input and output has continued, and some *microprocessors* include an additional control signal to indicate that the address appearing on the address lines is to identify an I/O device address, rather than a memory address. However, another technique, called memory mapped I/O, is perhaps more widely utilized. With this method, I/O devices are assigned specific locations in the address space of the processor, and any access to that address actually results in an I/O transfer of some kind. The memory mapped I/O scheme has the advantage that no special I/O instructions are required in the instruction set, and this reduces the complexity of the instruction decode mechanism. In addition, devices attached to the processor need not decode special signals to differentiate between memory and VO requests. However, the fact that I/O instructions are included in an instruction set does not prevent the use of memory mapped I/O techniques in a system. The user of the system can decide which technique would be most appropriate for the goals of that particular implementation.

We will identify specific I/O functions and methods in Chapter 6. But as far as the instruction set architecture is concerned, the important point is that the system be capable of transferring information to and from devices attached to the processor. This can be data or status information, and can be used in calculations and decisions in the same manner as other data/status information within the system. Consider the following simple example of a transfer method.

Example 4.11: Memory mapped I/O: A 16-bit computer system that uses the memory mapped I/O scheme has been configured so that the addresses $FFE0_{16}$ and $FFE1_{16}$, are assigned to a simple I/O device. The status of this device is obtained by reading $FFE0_{16}$. The least significant bit is set whenever the device is ready to accept a value, which can be written to the address $FFE1_{16}$. The second bit is set whenever the device has data that the processor can read. This data is obtained by reading address $FFE1_{16}$. Create an appropriate code segment to move data from the array DATA_OUT to the device, at the same time accepting data from the device and putting it in the array DATA–IN. Assume that the operation will finish when the last value of DATA–OUT has been transferred.

The actual code for this example would depend on the instructions in the system, but the point here is to use the memory mapped capabilities to do the desired work. The code must check to see what data transfers are possible and perform them:

```
        MOVE #length, R0      Put the number of transfers in R0
        MOVE #FFE0, R1        Move the status address to a register.
        MOVE #FFE1, R2        And the data address.
        MOVE #<datain>, R3    Fill R3 with address of input area.
        MOVE #<dataout>, R4   Fill R4 with address of output area.
loop:   MOVE *R1,R5           The MOVE sets status bits according to value
        JZERO lwp               transferred; back to loop if it's zero.
        AND #1,R5             If it's not zero, is the LSB set?
```

```
        JZERO output      If LSB not 1, must be ready for output.
        MOVE *R2, *R3+    This is input function.
        BRANCH loop       Now go back and check again.
output: MOVE *R4+, *R2    This outputs one value (and inc's address).
        DECREMENT R0      Now are we done? Decrement R0, and if the
        JNZERO loop         result is not zero, go back and try again.
        instruction       Otherwise, we this is the next instruction.
```

This code will cause the machine to poll the I/O device until either the device has information for the system, or the I/O device can accept data from the system. The polling is done by reading the status register of the I/O device, which is available at address $FFE0_{16}$. When the status register indicates that transfers can occur, they are performed by writing/reading the appropriate memory location.

The above example identifies one method by which information can be transferred between a processor and a peripheral device. However, the polling mechanism demonstrated by the code is extremely inefficient in many circumstances. For example, if the processor issued a command to a tape drive to seek a particular file on a tape, a very long time will pass between issuing the request and having the device respond with the desired results. With the polling technique, the capabilities of the system are not available for anything else during the seek time. Therefore, it is more efficient to have the I/O device send a signal to the system when the action of a command (in this case the seek action) has been completed. This signal is an interrupt, and it signals the processor to interrupt its current action and do something. What the system should do when it responds to an interrupt is defined in a routine called an interrupt service routine. The behavior of the system when responding to an interrupt is identical in many respects with the action of calling a subroutine. Thus, the instructions dealing with interrupts mimic the instructions involved in subroutine linkage.

One of the most basic requirements of the system with respect to the interrupt facility is the ability to enable or disable interrupts. This action is provided in some systems by including two specific instructions: one to turn on the interrupt facility, and one to turn it off. This function can also be handled by a bit (often called the interrupt enable bit) in the status/control register of the system. This bit is set and cleared by the logic instructions of the system.

If the interrupt capability of the system is enabled, the interrupt facility is checked at the end of each instruction. If an interrupt is pending, the appropriate action is taken. If the conditions are such that the interrupt should be recognized, then the system responds by causing execution of the interrupt sequence. If the interrupt conditions are not met, then the request is ignored. The "conditions" range from the simple to the complex. In some extremely simple systems, all interrupting devices activate the same control line, so there is no way of differentiating (in hardware) what caused the interrupt. Hence, any interrupt will cause the interrupt sequence to be initiated. Another method is to group interrupting devices into levels, and to prevent interrupts below a specified level from being recognized. A system with this ability may prevent interrupts from devices of a lower priority from being handled until the action required by a higher priority device has been completed. Another method is to individually control the interrupt ability of I/O devices. Thus, before an interrupt can be recognized, both the system level interrupt and the device level interrupt facilities must be enabled.

The interrupt acknowledge sequence is identified by the system architect when the design is specified. One intermpt sequence is to do an automatic subroutine call with a predefined target address. The subroutine, which is the interrupt handling routine, is then responsible for disabling further interrupts, saving whatever information is needed, and then doing the work. When the interrupt service routine has completed the action needed by the interrupting device, then the system reenables the interrupt, and returns to the program where the execution was active when the interrupt occurred.

The desired behavior is for the interrupt action to be invisible, except for the time required by the interrupt service routine. That is, the state of the machine should be the same directly after the interrupt has been serviced as it was before the interrupt occurred. The state of the machine refers to the contents of all of the appropriate registers, both general purpose and special registers. For this reason, the information saved by the interrupt action must include all registers altered by the interrupt service routine. These will be restored in the process of returning to the interrupted program. For example, when the 6800 microprocessor recognizes an interrupt, the accumulators (there are two), the program counter, and the status register are all pushed to the stack. The RET instruction restores all of these values and continues execution. Other systems respond in a similar fashion, storing a sufficient amount of state information to return to normal programming at the end of the interrupt service routine.

If one address is specified for all interrupts in the system, then the interrupt service routine does not have a sufficient amount of information to identify which of the possible interrupting devices actually is requesting attention. Therefore, the routine must poll all appropriate devices to ascertain which one needs service. A more time-efficient mechanism is to ask the devices to identify themselves so that a specific routine can be accessed. This is called a vectored interrupt, and is implemented in a variety of ways. One is to have an interrupting device supply the address at which the interrupt service routine will be located. This is the method used in UNIBUS devices: a special interrupt acknowledge bus cycle requests the address from the applicable device, and obtains from that address the address of the interrupt service routine (one level of indirection) and a new status word.

A similar mechanism for vectored interrupts is used by a variety of microprocessors. The technique calls for a number of interrupt levels, and to each level is assigned a device or number of devices. Within the memory of the processor is created a table containing the address of the interrupt service routine for each level. An interrupt is requested by asserting the lines to indicate the appropriate level, and the processor automatically extracts the corresponding address from the table. With this technique, if several devices have the same priority level, then the interrupt service routine must use an additional information mechanism, such as polling, to find the device that actually requested the service.

The interrupt mechanism is very useful for handling any event that needs service. The above discussion deals with external interrupts, such as a disk or tape drive interacting with the processor. But the same kinds of information are needed for exception conditions that occur during the execution of a program. For example, if an overflow or similar arithmetic problem occurred during the execution of a program, the system must deal with it in a reasonable fashion. One way is to ignore it, which requires no additional facilities. Another way is to allow the system hardware to cause an interrupt in the same manner as a disk drive. This is called a trap or exception, and is used for a variety of functions, as demonstrated by the following example.

Example 4.12: Interrupt mechanisms: The instruction set architecture for the 68020 is given in Figure 4.20. What is the interrupt mechanism for this device? What traps are established in the system that also use this mechanism?

The figure indicates that the system can function at two levels, the user level and the system level. The user is prevented from accessing some of the registers of the device, whereas the system does have access to all the user registers and those that have a direct impact on the system operation. Of particular interest for this example is the treatment of the status register. The user has access to the condition codes (extend, negative, zero, overflow, carry), but not the other portions of the status register. The user cannot influence the 3 bits comprising the interrupt mask. The system establishes the interrupt level by setting the 3 bits to the desired level. At the end of each instruction the three interrupt lines of the processor are checked to see if they form a number that indicates a high enough priority level to request an interrupt. If no interrupt is pending, or if the pending interrupt is not of sufficient priority to request attention, then the next instruction is fetched and processing continues. However, if an interrupt of sufficiently high priority is pending, then several things happen to begin the appropriate processing. The status register is copied so that it can be restored when the interrupt processing is complete. The system state is changed to supervisor mode. The interrupt mask level is set to the level that caused the interrupt, so that interrupt requests of the same (or lower) priority are ignored until the current one is completed. The processor requests a vector number from the interrupting device; the vector number is obtained on the data bus. This will be used to obtain the address of the interrupt service routine from the exception vector table shown in Table 4.4. After obtaining the interrupt number, the current processor context is saved. This is done by pushing onto the active supervisor stack an exception stack frame, whose format is shown in Figure 4.21. The figure shows the information in a 16-bit configuration, but the processor, which is capable of 32-bit transfers, will use the wider data transfers as much as possible to speed up the process. The status register used is the copy of the status register made at the initiation of interrupt processing. The program counter is the 32-bit address of the next instruction to execute. The vector offset is the offset value (interrupt number x 4) that will be used to identify the address of the interrupt service routine. And the additional state information contains other system registers and information. This will vary from 0 to 42 words, depending on the interrupt that initiated the action. (For further details, refer to the device specification.) Under certain circumstances, an additional exception stack frame will be created on the interrupt stack. Finally, the address of the interrupt service routine will be obtained by adding the offset value to the contents of the vector base register. At that location is the address of the intermpt service routine. Processing continues at that location.

As can be seen from Table 4.4, addresses are maintained in the exception vector table for both user-defined interrupts and system traps. If the floating point unit detects an underflow, for example, the interrupt sequence for intermpt number 51 is initiated. Regardless of the source of the interrupt, when the service routine has completed the necessay processing, control is returned to the controlling program by the RTE (Return from exception) instruction, which pops the appropriate information off of the stack and restores it to the appropriate registers. The format bits shown in

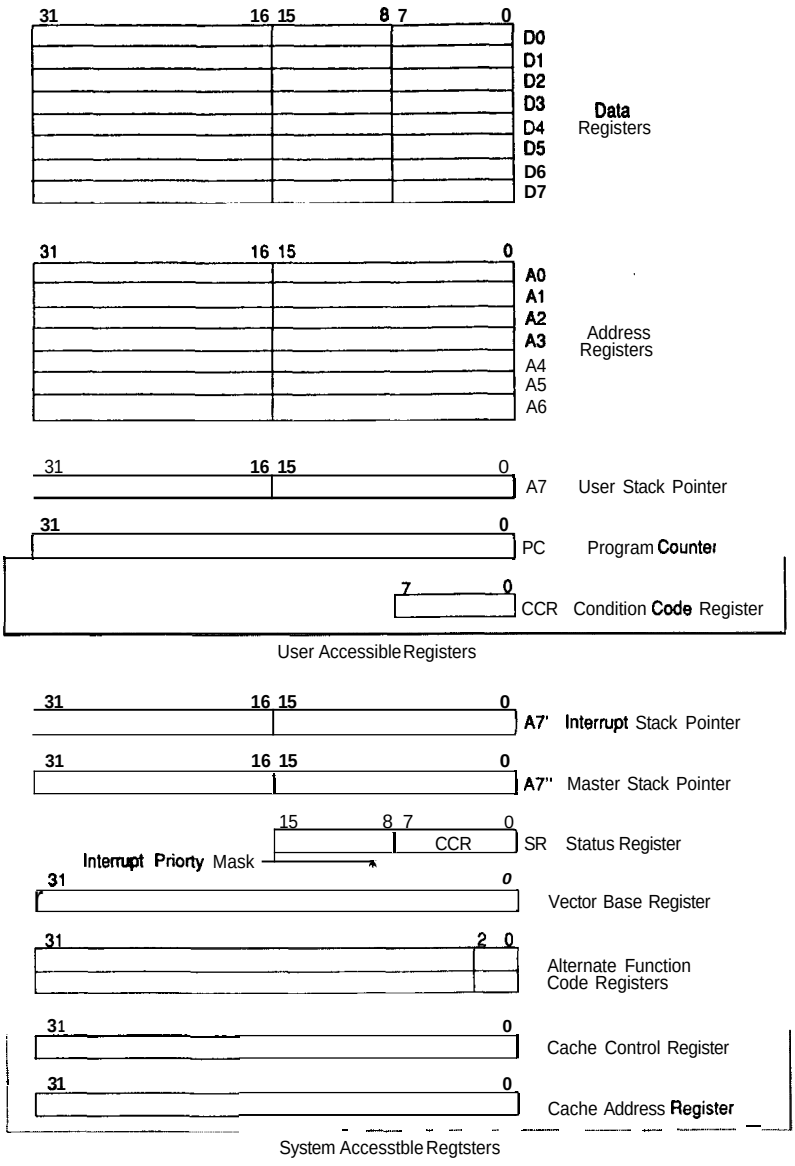


Figure 4.20. Instruction Set Architecture for the 68020.

Table 4.4. Exception Vector Assignments for the 68020.

Vecror Number	Vecror Offset Hex	Spare	Assignment
0	000	SP	Reset: initial interrupt slack pointer
1	004	SP	Reset: initial program counter
2	008	SD	Bus error
3	00C	SD	Address error
4	010	SD	Illegal instruction
5	014	SD	Zero divide
6	018	SD	CHK, CHK2 instruction
7	01C	SD	ccTRAPcc, TRAPcc, TRAPV instructions
8	020	SD	Privilege violation
9	024	SD	Trace
10	028	SD	Line 1010 emulator
11	02C	SD	Line 1111 emulator
12	030	SD	Unassigned, reserved
13	034	SD	Coprocessor protocol violation
14	038	SD	Format error
15	03C	SD	Uninitialized interrupt
16-23	040-05C		Unassigned, reserved
24	060	SD	Spurious interrupt
25	064	SD	Level 1 interrupt auto vector
26	068	SD	Level 2 interrupt auto vector
27	06C	SD	Level 3 interrupt autc vector
28	070	SD	Level 4 interrupt auto vector
29	074	SD	Level 5 interrupt auto vector
30	078	SD	Level 6 interrupt auto vector
31	07C	SD	Level 7 interrupt auto vector
32-47	080-OBC		TRAP #0-15 instruction vectors
48	0C0	SD	FPCP Branch or set on unordered condition
49	0C4	SD	FPCP Inexact result
50	0C8	SD	FPCP Divide by zero
51	OCC	SD	FPCP Underflow
52	OW	SD	FPCP Operand error
53	OD4	SD	FPCP Overflow
54	OD8	SD	FPCP Signaling NAN
55	ODC	SD	Unassigned, reserved
56	0E0	SD	PMMU Configuration
57	0E4	SD	PMMU Illegal operation
58	0E8	SD	PMMU Access level violation
59-63	OEC-OFC	SD	Unassigned, reserved
64-255	100-3FC	SD	User-defined vectors

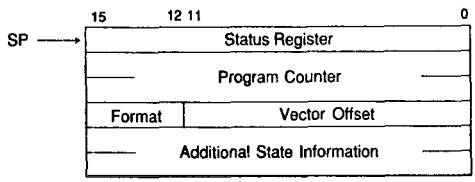


Figure 4.21. Exception Stack Frame of the 68020.

Figure 4.21 will indicate to the system the number of words in the additional state information, and the system can then restore them appropriately.

An important time in determining the efficiency of an interrupt system is the amount of time required to recognize an interrupt and return, without performing any work. The actual times will depend on the processor speed, the memory speed, and the state of the cache memory. However, a "normal" intermpt will require four 16-bit words, or at least two 32-bit transfers onto the stack, to save the state, then a memory transfer to obtain the address of the interrupt service routine, and a memory transfer to obtain the first instruction to execute. Thus, a minimum of four memory transactions are required to initiate an interrupt service routine. A minimum of two transfers (from the stack) is required to restore the system to functioning order, plus one to obtain the next instruction to execute. Additional times would be required for the other, nonmemory activities as well.

In this section we have seen that a processor requires the ability to communicate with external devices. This can be accomplished by using dedicated I/O instructions or by using the memory mapped I/O technique. In either case, the system hardware has the ability to transfer information to and from the external device. This can be data, or it can contain status and control information. However, for transfers involving large amounts of data, programmed I/O techniques are not always applicable, and some form of automatic transfers are used. These techniques, such as direct memory access (DMA), will be discussed in Chapter 6. When an I/O device has completed its assigned task, it often has the ability to signal the CPU that it needs attention, and this interrupt facility allows the processor to be doing other tasks while the I/O device is busy.

When interrupts are recognized by the processor, the CPU will save a sufficient amount of information to be able to return to what it was doing, and then transfer control to an interrupt service routine. This routine identifies the device that requested the service, and performs the necessary processing. The identification process can be taken care of by polling, by a vector mechanism, or by appropriate combinations of these techniques. When the interrupt processing is complete, the CPU can return to the original processing in much the same way that a subroutine is performed.

4.8. RISC vs. CISC: Instruction Set Strategies

To this point in this chapter we have identified different types of instructions and addressing mechanisms. One of the questions that must be addressed by a system architect concerns the number and type of instructions to be included in a specific computer system. One strategy is to include a large number of instruction types and addressing modes. A system of this type is called a complex instruction set computer (CISC). An alternative method is to reduce the complexity of the instruction set, and hence the reduce the logic required for the implementation of the system, including only the instructions needed for the desired application. A system of this type is called a reduced instruction set computer (RISC). In this section, we will examine some of the issues involved in the decision process, and some of the techniques that have evolved with the RISC machines.

The earliest machines were very simple in their architecture and implementation, both because experience with computing systems was nonexistent and

because the technology of implementation mandated a simple machine. Thus, the language of the machine was correspondingly simple. However, users of the computers wanted to solve relatively complex problems, and these users described their problems in a language that treated variables and arithmetic at a higher level than the language of the machine. This resulted in what has become known as the semantic gap, which is the gap between the language of the machine and the language of the user. The languages of users (FORTRAN, Pascal, **LISP, C,** etc.) became more complex to represent increasingly more complex problems. In response to this trend, computers themselves became more complex, changing with the available technology and user demands for speed and versatility. The attempt was to reduce the semantic gap by creating more complex computing systems. This would enable users of computers to more effectively utilize the computational capabilities of the system.

Effective utilization of a computing system is accomplished by creating a suitable bridge for the semantic gap. The most common bridge is a compiler, which accepts as input a problem written in the language of a user, and creates as output a corresponding solution in the language of the machine. Complex instruction set computers seek to reduce the difficulty of the task of the compiler by making the instructions of the machine more closely conform to the instructions of the higher level language. Some systems [RiSm71, Ditz81] have been created in which a high level language is the native language of the processor, but this is not a general practice.

Observations of the behavior of programs executing on real machines provided some interesting insight into the operation of computers. These observations indicated that most of the time the computer was utilizing a small subset of all available instructions. Carrying this observation to the next logical step, system architects concluded that the system speed could be enhanced by including only the often used instructions, and by making them as fast as possible. This simplification of the instruction set and the implementation hardware results in a unit that can run faster. However, the more complex functions of a programming language must be accomplished with subroutines or with longer instruction sequences than corresponding **CISC** instruction sequences. The result is that a program may require more instructions to complete on a **RISC** machine than on a **CISC** machine, but the **RISC** instructions will, in general, have a higher execution rate.

The **RISC** approach, then, is to create a system that is simpler in architecture and faster in implementation than a **CISC** machine. With the simplicity comes the promise of speed, and with many implementations this promise is realized. However, care must be taken when comparing machines based on a rate of instructions per second, since the work accomplished by a **RISC** instruction will, in general, not be as great as the work accomplished by a **CISC** instruction.

The basic issue, which is treated differently by the **RISC** and **CISC** approaches, is one of resource utilization. How can the system resources be used most effectively? Different answers to this question are possible, based on the relative costs associated with the resources by the system architect.

The tenets of **RISC** architectures strive to maximize the speed and minimize the complexity of the implementation. Simplicity is the basis of both the architectural definition and the implementations. Some of the basic policies which are the result of this type of an architecture are a minimal number of instructions and addressing modes, fixed instruction formats, hardwired instruction decoding, single cycle execution of most instructions, and the use of a load/store type of organization.

Minimal number of instructions and addressing modes. By including only the instructions that are executed often, the system need not include seldom used features. The result is a smaller, faster system, that is capable of doing more instructions in a given amount of time than a CISC machine. The CISC machine, on the other hand, will specify more work in a single instruction. Thus, while the CISC instruction will take longer to complete, fewer such instructions are required to do the work of a high level task.

Fixed instruction formats. By restricting the format of the instructions, the tasks of the control system are simplified. In the ***fetch-decode-execute*** mechanism of stored program computers, the decode function must identify the work to be done. By causing all of the instructions to use the same format, then the decisions required of a decoder are minimized. For more complicated instructions, such as those of a CISC system, the decoder must first ascertain the length of the instruction, extract the necessary information from the instruction stream, and then finally specify the tasks needed to do the work. With f[illegible] a[illegible] choice, and a restricted location for the specification information, the speed of the system is enhanced.

Hardwired instruction decoding This characteristic accompanies the fixed instruction format idea, and can be useful for two different reasons. The first is that hardwared instruction decoding (using random logic to implement the decoding function) can, in general, be done more rapidly than the altemative mechanisms. such as microcoding. We will discuss different alternatives for the control system in the next chapter. Hardwired logic has traditionally been faster than memory based techniques, such as microcode. The early machines used this technique simply because the memory technology was not sufficiently fast to be attractive. However, the development time was longer because of the difficulty of generating correct logic for all conditions. When small, high speed memories became a reasonable altemative, then microcoded systems became attractive because of their regularity and versatility. The speed ratio of data memory and microcode memory has been steadily decreasing in recent years, so the use of microcode for speed is not as beneficial now as it was previously, although the use of microcode for versatility is still attractive. Thus, to enhance the speed of the control function, hardwired logic for instruction decode is a reasonable alternative. The increased use of computers as tools to aid in the design process has made this altemative viable, since the correctness of the design can be tested before the design is committed to hardware or silicon.

Single cycle execution of instructions. If a computer system can be so organized that one instruction is executed in each cycle, then by some standards maximum utilization of all system resources can be approached. Again, the technology plays a part in the decision process, limiting and shaping the tjpcs of things that can be done in a cycle time. As VLSI technology evolves, functions that once took many cycle times, such as floating point arithmetic, can now be done in a very short time. Thus, organizing the system to take advantage of this can be very beneficial. However, this limits some of the action of a system, since certain types of operations cannot be accomplished in a single cycle. For example, incrementing a value located in memory cannot be done in a single cycle. since the value must be obtained from memory, then updated, and then rewritten to memory. Hence, the instructions included in the system are all restricted to what can be accomplished in a single cycle. Some RISC systems deviate from this to allow certain instructions to take two (or more) cycles, which permits reuse of certain system resources, or allows for delays through logic that require more time than allowed in a single cycle.

Load/store memory organization. With a load/store memory system, the only instructions that deal with memory are those that load information into registers from memory or that store information from registers to memory. All arithmetic/logic instructions work with values in registers. By placing the operands of arithmeticbogic instructions in registers, the above stated objective of an instruction per cycle can be met. With this organization, the operands are readily available, and can be extracted as needed from the registers. No time is lost waiting for operands to be obtained from the data memory. However, separate instructions are required to move the information to the registers to be used. The RISC technique relies on the observation that in general infoimation will be used several times before results are written to memory. The CISC technique, which does not restrict the location of the operands for the instructions, allows either the register intensive technique or memory-to-memory operations to be used.

In addition to the features listed above, the RISC architectures rely on effective utilization of additional architectural techniques such as pipelining, multiple data paths, and large register sets. These techniques are not strictly associated with RISC machines, but combining the techniques with the reduced instruction set ideas often results in a higher speed system. At this point we should hasten to add that not all RISC systems adhere to all of the tenets listed above, and that most available RISC systems violate at least one of them.

The basic concepts and ideas of pipelining are discussed in Chapter 8, so we will not elaborate on the RISC use of pipelines here. But one of the reasons that pipelining functions well for RISC machines is that the restricted operand placement for arithmetic/logic instructions minimizes pipeline delays for operand fetches. An operand required for execution of an instruction must be obtained by a pipeline before the operation can continue. If these operands are always restricted to fast registers, such as in the RISC case, then the delays associated with operand access are minimized. If the operand is in general purpose memory, such as in a CISC machine, then a relatively long time is required to obtain the information, which reduces apparent system speed.

The use of multiple data paths allows a greater amount of parallelism and concurrency to be used in the implementation of systems. This is evident in two areas, as seen by the block diagram for the Motorola 88000 RISC system, shown in Figure 4.22. The two areas identified in the figure are the multiple buses contained within the 88100 processor chip and the distinct instruction and data paths to memory.

The use of multiple buses internal to a processor allows transfer of multiple operands in any given cycle. In particular, two source operands can be provided to a functional unit, and a destination operand from a functional unit provided to the register file within a single cycle. This requires buffer registers within the functional units to hold the values while the buses are released to be used elsewhere. And the multiplicity of functional units increases the opportunities for parallel activity within the processor itself.

Providing different paths for both the instruction and data transfers allows those two functions to proceed simultaneously. This is necessary if the goal of one instruction per cycle is to be achieved. But by using this technique a new instruction can be made available in each cycle, regardless of the data transfers needed by the system. For arithmetic instructions, the data path to memory would not be needed. But for instructions that transfer information to and from memory, both ports would be used very efficiently.

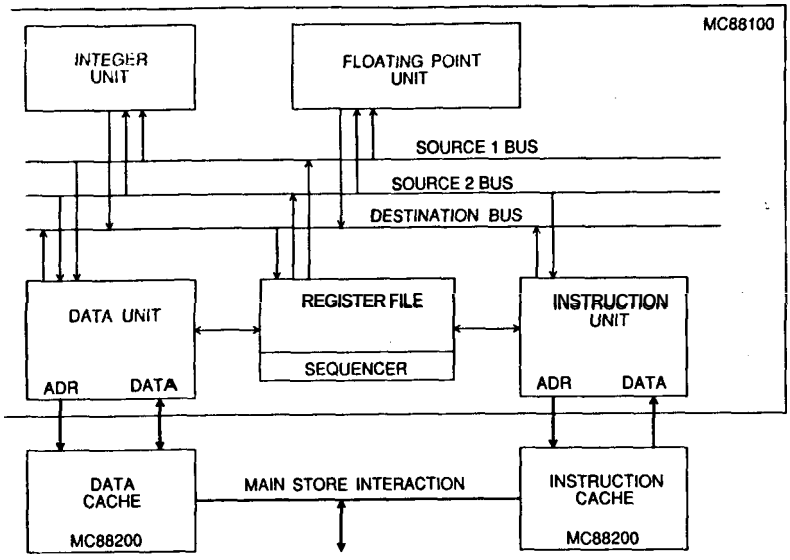


Figure 4.22. Motorola 88000 **RISC** system.

Like pipelining, the concept of multiple paths for information transfer is not limited to use in **RISC** systems. However, a system that follows the **RISC** concepts will be able to optimize the use of multiple information paths for enhanced system speed. The same is hue of large register sets. This technique can be used in systems of any type. However, one of the techniques that has been linked with **RISC** systems, and that utilizes a large register set, is the use of register windows for parameter passing in subroutines.

By measuring the frequency of instruction execution, it has been observed that the process of calling and returning from subroutines consumes a large amount of processor time. In an effort to minimize this, the idea of using register windows on a large register set has been proposed. The basic idea is that many registers **are** included in the system, but that only a limited number of them **are** accessible by the system at any one time. This limited number of registers is identified as the "window" into the set of all registers. To change the window, a pointer that identifies the active registers is modified to specify a new set of active registers. When the windows overlap between routines, then parameters can be passed by placing them in the registers that are accessible by both routines. In this way, the memory **transactions** required for pushing parameters onto a stack, and then popping them off, are minimized. As long as the number of **parameters** is less than the register overlap, no memory transactions are required for passing of the parameters.

This technique was utilized by the architects of the **RISC** I system at the University of California at Berkeley, and their use of registers is shown in Figure 4.23. The instruction set uses 5 bits to specify registers to be used in an instruction. Thus, 32 different locations can be accessed. The first ten registers **(R0–R9)** are global registers, and **are** accessed by any routine, regardless of the number of subroutine calls. The remaining 22 registers are broken into three groups: the

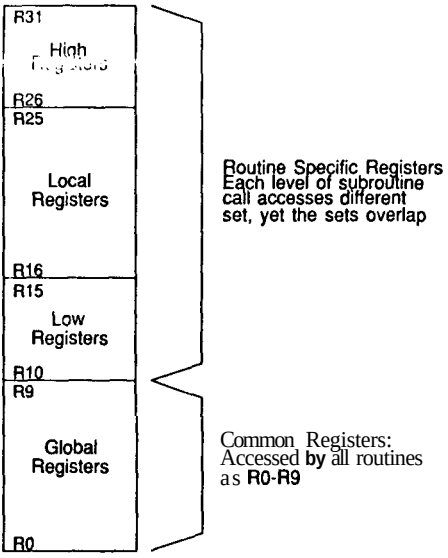


Figure 4.23. Register Use in RISC I.

high registers, the local registers, and the low registers. The high and low groups each contain six registers, while the local group contains 10 registers. Together these three groups form a routine specific set of registers. Thus, when a routine is accessing register storage, it will identify a value in either the global registers or the routine specific registers. It is then the responsibility of the system/user to use the registers in a coherent manner.

As mentioned above, one of the primary reasons for using register windows is for parameter passing in subroutines. When returning from or calling a subroutine, a pointer that identifies the location of the routine specific registers within the set of all registers is modified to point to the next set of routine specific registers. This modification is an increment/decrement by 16, which causes six of the registers to be shared between routines. With 22 registers in the routine specific set, this causes an overlap of six registers between the two routines. Any data to be exchanged between the two is merely left in an agreed-upon register by one routine, and the other routine knows where to obtain the information when it is needed.

This process of information exchange is graphically depicted in Figure 4.24. Only a portion of the overall register set is shown. If a program is executing Routine A, then the 32 registers to which it has access are the global register set (R0–R9) and the routine specific set which begins at location 90 in the register set. The routine specific registers for Routine A are referred to as R10 to R31 by instructions within the routine, but the system actually utilizes registers R90 to R111. Now assume that Routine A is going to call Routine B, and that it needs to pass two values. Routine A places the values in R10 and R11 (which are physically R90 and R91) and calls Routine B. The subroutine call identifies the address of Routine B; execution of the instruction changes the program counter, creates the appropriate return linkage, and decrements by 16 the pointer identifying the

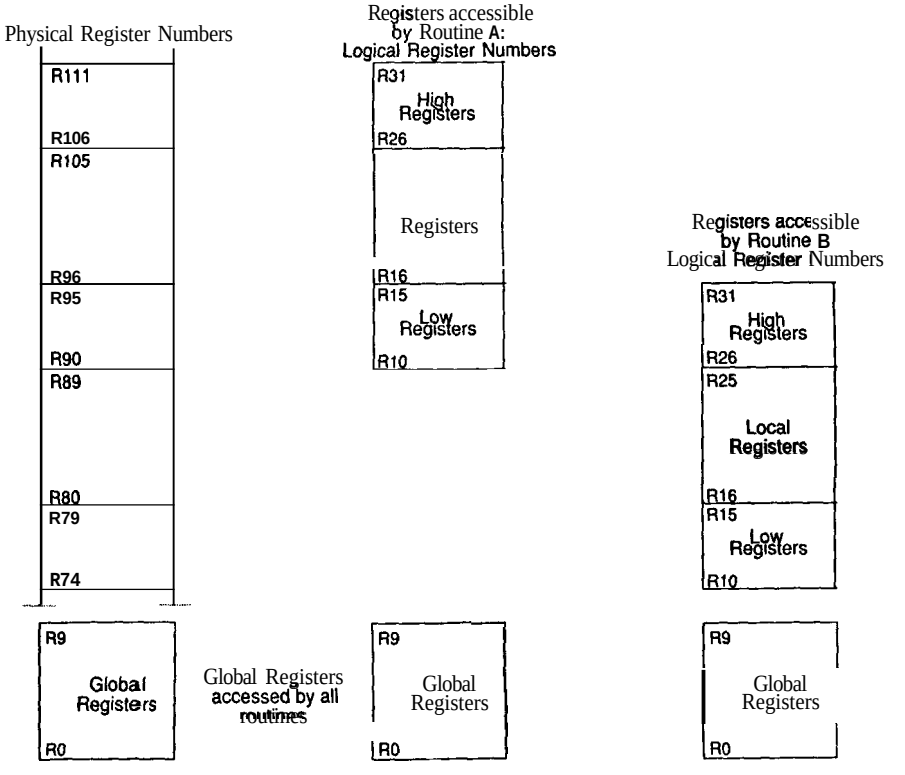


Figure 4.24. Parameter Passing with Register Windows.

routine specific set of registers. When Routine **B** needs the information passed to it. it will access **R26** and **R27** (which are physically **R90** and **R91**). Parameters passed back to a calling routine will utilize the same technique, with Routine B leaving results in, say, R31 (physically **R95)** and returning control to Routine **A.** And Routine A obtains the value by accessing **R15** (which is physically R95). Note that no special stack operations were involved to pass parameters; the parameter passing was accomplished by merely organizing the processing in such a way that, when the subroutine was called, the information to be passed was found in the overlapped register area.

The use of register windows allows parameters to be passed without memory intensive stack operations. **A** second benefit is that a subroutine need not save state before beginning actual work. In a "normal" machine, if a suhroutine is going to modify eight of the general purpose registers, it will first save the contents of those registers (probably on the stack). Then, before returning to the calling routine, the registers can be restored to their previous value. These operations are not needed if register windows are used, since different physical registers are used for each routine specific set of registers. However, care must be taken to be sure that overlapped registers are used in a reasonable fashion.

The above technique will minimize the memory interactions needed for parameter passing and subroutine use of registers, but the technique incurs some different costs. One of the costs is the number of registers needed to store the information. Obviously, it would be ideal to have an infinite number of registers, but that is not a reasonable solution. The number of registers included is based upon the expected depth of subroutine calls. Studies of actual programs have shown that, for most applications, the nesting level of subroutines is on the order of eight. Including 144 registers would allow the above technique to have a subroutine call depth of eight before additional transfers would be needed. Obviously, if the nesting level exceeds eight, then a great many memory transfers would be required to save either the entire set of registers or some designated portion of it. As with other techniques, the idea of including register windows in a system is not solely a RISC concept, but rather a mechanism that can be utilized wherever it will result in an improved system.

The use of memory is another of the interesting aspects of RISC architectures that needs to be considered in a system. Memory technology has made rapid advances in both speed and size of available memory systems. In a time when memory systems were quite small by today's standards, the size of a program was a critical measure of the effectiveness of the system. However, as memories have become faster and larger, the need for having small programs has been reduced. In general, programs on a RISC machine will occupy a somewhat larger section of memory than similar programs on CISC machines, since more instructions are required to do the work. However, since memories are becoming increasingly larger, this is often not considered a drawback. Also, since the architecture attempts to minimize delays due to memory interaction (separate data/instruction paths, and register only arithmetic, for example), overall effect is to create a system that can do work faster.

The term RISC refers to an approach rather than to a specific system or set of requirements. For example, one of the tenets listed above is that a RISC system will use hardwired control, yet some computer systems advertise themselves to be RISC computers that utilize microcoded control systems. Real computer systems will range from units that adhere strictly to the RISC approach and simplify all aspects of the system, to units that follow the CISC approach and include highly complex capabilities. The "best" system will be the one that makes the most judicious use of system resources to solve the problem for which it is intended. And whether a RISC approach or a CISC approach is a better choice cannot be determined without applying appropriate metrics, and perhaps trying the systems in a real application.

4.9. Summary

We have discussed a number of mechanisms for doing work in computers, where work is defined as directing a CPU to perform a specific task. The work that a computer is capable of doing is defined by the set of instructions controlling the operation of the machine. The set of instructions of the system also identifies the apparent architecture of the system or the instruction set architecture. Implementations of the architecture may or may not contain all of the registers, functional units, and data paths alluded to in the instruction set architecture.

The structural aspect of the system — the functional units, data paths, and storage elements included in the machine — will determine the mechanisms

needed to implement the instruction set architecture. When the structure of the system is known, then the internal transfers required to carry out the work of the instructions can be represented in a register transfer language.

Instructions that control the arithmetic and logic operations of a system can have a varying number of addresses, from zero address stack machines to three address systems capable of identifying both sources and destination of an operation. The choice of the instructions to be included in the system is made by the system architect after careful consideration of the application area of the machine and the utilization of available system resources to accomplish the required system objectives.

The use of registers for operand storage reduces the number of bits required to identify the location of information as well as the time required to obtain the information. Registers can also be used to effectively identify the location of values in a memory system.

Operands for instructions can be located in general purpose memory or in registers. The instruction set may contain multiple addressing modes to identify the location of the information. These include combinations of direct and indirect addressing, indexing, stack operations, and instruction stream accesses.

Program control instructions allow changing the flow of control in a program executing on the system. This change of flow can be unconditional or based on some status of the system. Also, routines can be called from within a program, and a retum linkage established.

Interaction with devices external to the system is accomplished with I/O instructions, or I/O techniques like memory mapped I/O. These devices have the ability to signal the computer system, or "interrupt" the program flow, when interaction is needed. In addition, internal conditions, such as arithmetic overflows, can cause interrupts within the system.

The RISC approach to computer architecture is to simplify actions to a minimal set, and use high speed hardware and optimizing software techniques to create a system that will execute programs at a high speed.

The functional units of a computer system, the interconnection system, and the instruction set that controls the action of the system must be created with all of the above ideas in mind. The architecture that is most effective in a given application will make the most efficient use of system resources, where resources can be time, power, memory, or any of a number of other measurable quantities.

4.10. Problems

4.1 A general purpose computer system must have the ability to perform certain basic functions in order to do useful work. Three of the basic functions are store, load, and add.

a. Name four other basic functions that the computer must do.

b. Name four additional instructions that would be nice to have.

4.2 Consider a machine with the following characteristics:

It is a two address machine.

Subroutine linkage is through a stack mechanism, in main memory.

There are eight general purpose registers, plus other special purpose registers.

The machine is capable of absolute, indirect, base plus displacement. and general indexed addressing modes.

a. Give a block diagram showing the major components of the system and their interconnection. Include arrows indicating flow direction of the data.

b. Using the block diagram, give the RTL necessary for

ADD R1, R2
Add the contents of register 1 to register 2

MOV *R1, *R2+
Move the contents of memory stored at thr location identified by R1 to the location identified by R2; then increment R2

CALL #A0F4
Go to the subroutine located at address A04F; this address is stored in the location following the CALL instruction in the instruction stream.

RETURN
Return to the calling routine from a subroutine.

4.3 One of the methods of evaluation for a machine is to determine its behavior for a program or program segment. Two of the basic computer methods discussed in this chapter are single address machines with a general purpose accumulator, and two address machines with a general register set. Create block diagrams for a single address machine and a machine with a general purpose register set. Then create assembly level code to implement the following statements:

```
for ( i = 0 ; i < 100 ; i++ )
    A[i] = B[i] * C[i];
```

Use the code generated and contrast the two methods. In particular, identify the number of instructions executed, the number of memory references required, and the number of arithmetic operations. Which of the figures of merit is the most crucial? Why?

4.4 Use the technique of Problem 4.3 to compare a CISC machine approach to a RISC machine approach. That is, create block diagram representations for a CISC architecture and a RISC architecture. Then create code to implement the loop of Problem 4.3. Use the number of memory references, the number of register references, and the number of arithmetic operations to contrast the two methods.

4.5 You have been given the task of developing a single address computer to be utilized in general purpose applications. This machine is to be a 16-bit single address computer capable of direct and indirect addressing. Operands obtained via the direct addressing mode are identified by their position with respect to the PC, so the access method could be called PC relative. This permits programs to be located anywhere in the memory. The machine will have more than eight but less than 16 instructions requiring memory access.

a. Give a block diagram of a computer that will fit these requirements. Show all major registers, and all data paths, including the direction(s) of data flow on the data paths.

b. Propose a method for encoding the instruction information for the system. That is, what should the instruction format be in the 16 bits stored in the computer's memory.

c. Give the register transfer language steps required for the following instructions:

ADD (indirect addressing)

CLEAR

JUMP TO SUBROUTINE (direct addressing)

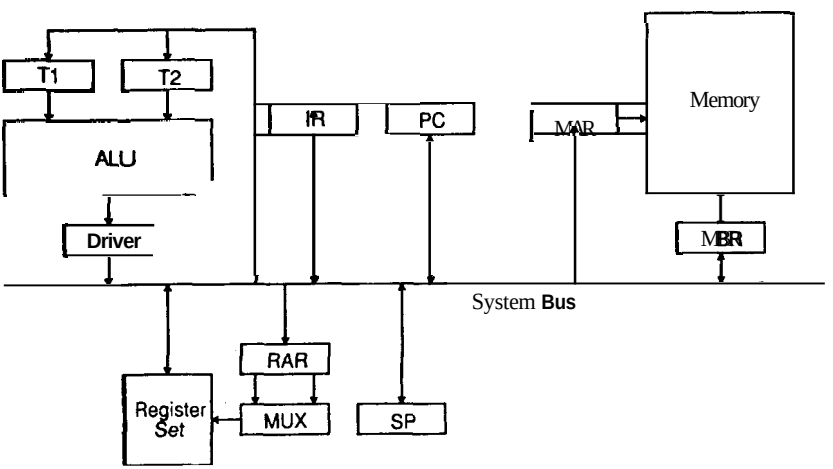
RETURN FROM SUBROUTINE

4.6 Computer Designers, Inc., has been contracted to design a special purpose computer with the following requirements (not a complete list): The machine will operate with a two address, register-oriented instruction set, with 16 general purpose registers. These registers are denoted R0–R15. The subroutine linkage is accomplished with a stack, R15 being the stack pointer. The program counter is R14. Operands (results) are obtained (deposited) either directly from (to) the registers or indirectly through the registers from (to) fast semiconductor memory. The memory space is 65,536 bytes. The indirect references can leave the pointer-register unchanged, increment it, or decrement it. The instruction set is composed of over 16 instructions, including **ADD, SUBTRACT, INVERT, AND, OR, EX-OR, NEGATE, JUMP, JUMP-SUBROUTINE, RETURN,** and **INCREMENT.**

a. Give a block diagram of the data path of the machine.

b. Give sufficient formats to accomplish the instructions (that is, however many formats are necessary: 1, 2, or more...).

c. Give an RTL description of **ADD** (with direct addressing of the operands), **INVERT** (use indirect, autoincrement mode to identify operand), **JUMP-SUBROUTINE** (use indirect addressing to identify location of subroutine), **and RETURN.**



4.7 Consider the above block diagram of a 16-bit single bus system. The program counter (PC), stack pointer (SP), and instruction register (IR) are 16-bit

registers capable of receiving information from and sourcing information to the general bus; the temporary registers (T1 & T2), the register address register (RAR), and the memory address register (MAR) are only capable of receiving information. The ALU can increment, decrement, invert, and add. The stack pointer identifies the next available location; stack grows to lower addresses in memory. The register memory contains 16 registers, and the main memory has 65,536 locations. The MUX on the RAR is to select either the source or destination register identification bits out of the 16-bit word loaded into the RAR. The machine has a two address instruction set with the following address modes: register, register indirect, register indirect autoincrement, immediate/absolute (absolute address is stored in next word of instruction), and program counter relative (used for jumps only; 8-bit displacement is stored in instruction word). Give the register transfer language statements for the following instructions: (operand order is Source, Destination)

a. ADD R1, *R2

b. SUBTRACT *R5+, #2A48

c. CLEAR R9

d. JUMP $-9

e. GOSUB #9BA4

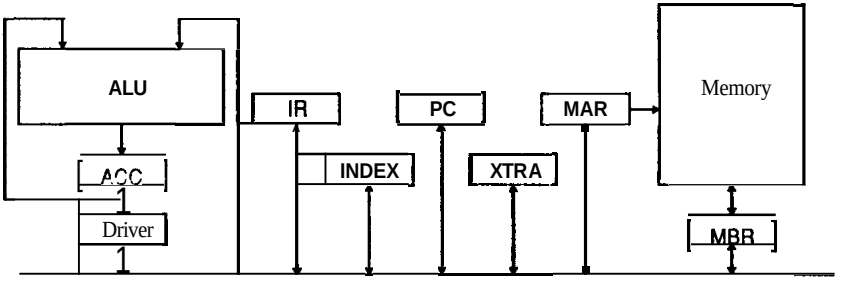
4.8 For the block diagram of Figure 4.10, give the RTL representation for

a. JUMP INDIRECT <52>
52 is stored in location following jump in instruction stream. Use contents of memory location 52 as target of jump.

b. ADD R1, R3
Add the contents of register I to the location in memory identified by register 3

c. JUMP TO SUBROUTINE 145
Transfer control to a subroutine located at address 145. This address is stored in the memory location following the instruction in the instruction stream.

d. INCREMENT R7



4.9 Consider the above block diagram of a computer. ACC, XTRA, IR, INDEX, PC, MAR, and MBR are registers that can be filled from the system bus or gated to the system bus. The ALU can do add, subtract, increment, decrement, and all of the logic operations. Give a register transfer representation of the complete instruction cycle for

a. indirect addition

b. indexed AND

c. unconditional jump

4.10 Consider a microcomputer that is a single address machine, with a general purpose accumulator (ACC) and a number of special registers. These registers include a 16-bit program counter (PC), a 16-bit stack pointer (SP), and a 16-bit index register (X). The address space is 16 bits, and the system is an 8-bit system. The system has four interrupt lines (0, 1, 2, 3, with line 0 assigned highest priority), which devices can assert to cause a vectored interrupt. The numbers associated with the vector lines refer to address allocations starting from the last location of memory (FFFF). A software system has been created that contains intermpt handlers for a floppy disk (routine starts at EF36), a cassette tape recorder interface (routine starts at F340), a line printer (routine starts at D45A), and a terminal (routine start. at C344).

a. Give the allocation for the upper part of memory (give memory map for' vector locations).

b. Give a register transfer language representation of the action that occurs when the cassette recorder asserts its intermpt line (which one is it?)

4.11 For the block diagram of Figure 4.2, give the register transfers required to implement an ADD instruction, and a NEGATE instruction. Assume that the number system involved is the two's complement number system. The ALU can do the following: feed MBB through, add ACC and MBR, increment ACC, and NAND ACC and MBR. Also, ACC can be cleared.

4.12 Consider the block diagram given for a Problem 4.7. The ALU is capable of addition (T1 + T2), subtraction (T1 − T2), increment (TI + 1), decrement (T1 − 1), and logical operations (T1 op T2). All registers are registers only, not counters too. Data paths, addresses, data are all 16 bits. Give the RTL for the following instructions:

a. ADD *R1, *(*R2)
Add the contents of memory whose address is in R1 to the contents of another memory location, and store the results back in this second memory location. The second memory location is identified by an address which itself is located in memory, and the address of the address is found in R2.

b. JMS addr
Transfer program control to the subroutine located at "addr," which is an address stored at the location following the instruction in the instruction stream.

c. **MOV** address I R3, R5
Move the contents of a memory location to register 5. The address of the memory location is found by indexing "address" by the contents of register 3. That is. "address." which is found in the location following the instruction, is added to the value in register 3, and the result is used as an address at which to find the operand.

4.13 One of the mechanisms discussed for parameter passing was the concept of register windows. Contrast a "standard system and a system with register windows by doing the following:

a. Prepare a block diagram of a "standard" system with 32 registers. Include as many time saving mechanisms as possible to help the execution time of instructions.

b. Specify instructions needed in this architecture to do a subroutine call.

Include not only the JMS itself, but also whatever other instructions are needed to pass parameters.

c. Prepare RTL implementations for these instructions to identify execution times. Then create a table of execution times for subroutine calls, since calling routines with a different number of parameters will result in different effective times for the subroutine calls.

d-f. Repeat steps a-c for an architecture that uses the concept of register windows. Use the same number of registers for the architecture as for the nonregister window system. Assume that the system has a sufficient number of registers to allow subroutine nesting to a depth of eight.

g. Suggest a mechanism to be utilized when the subroutine nesting level exceeds eight. How much time will be required to handle that situation?

4.14 Obtain instruction set specifications and instruction set architecture descriptions for the VAX architectcure, the MIPS architecture, and the 32000 architecture. Compare the contents of the status registers for the three systems, and the conditional branch instructions available. Defend one of the approaches as better than the other two, using system resource requirements and reasonable metrics to explain your position.

4.11. References and Readings

[AgDa78] Aguero, U., and S. Dasgupta, "A Plausibility-Driven Approach to Computer Architecture Design." *Communications of the ACM.* Vol. 30, No. 11, November 1987. pp. 922–932.

[AlWo75] Alexander. W. G., and D. B. Wortman, "Static and Dynamic Characteristic of XPL Programs," *IEEE Computer.* Vol. 8, No. 11, November 1975, pp. 41-46.

[AmBl64] Amdahl, G. M.. G. A. Blaauw, and F. P. Brooks. Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development.* Vol. 8, No. 2, April 1964, pp. 87–101.

[Baer84] Baer, J. L., "Computer Architecture," *Computer.* Vol. 17, No. 10, October 1984, pp. 77-87.

[Baer80] Baer. J. L.. *Computer Systems Architecture.* Rockville, MD: Computer Science Press. 1980.

[Barb81] Barbacci, M. R., "Instruction Set Processor Specification (ISPS): The Notation and its Application," *IEEE Transactions on Computers.* Vol. C-30, No. 1, January 1981. pp. 24–40.

[BaSi82] Barbacci, M. R.. and D. P. Siewiorek, *The Design and Analysis of Instruction Set Processors.* New York: McGraw-Hill Book Company, 1982.

[BaNo80] Barbacci. M. R., and J. D. Northcutt. "Application of ISPS, An Architecture Description Language." *Journal of Digital Systems.* Vol. 4, No. 3, Fall 1980, pp. 221–239.

[Bart85] Bartee, T. C., *Digital Computer Fundamentals, 6th edition.* New York: McGraw Hill Book Company. 1985.

[Basa85] Basart, E.. "RISC Design Streamlines High Power CPUs," *Computer Design.* Vol. 24, No. 7, July 1, 1985, pp. 119–122.

[BeNe71] Bell. C. G. and A. Newell, *Computer Structures: Readings and Examples.* New York: McGraw-Hill Book Company, 1971.

[Blak77] Blake. R. P.. "Exploring a Stack Architecture." *Computer.* Vol. 10. No. 5, May 1977. pp. 30–41.

[Boot84] Booth, T. L., *Introduction to Computer Engineering: Hardware and Software Design.* New York: John Wiley & Sons, Inc., 1984.

[Bulm77] Bulman. D. M.. "Stack Computers: An Introduction." *Computer.* Vol. 10, No. 5, May 1977, pp. 18–29.

[BuGo46] Burks, A. W.. H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Institute for Advanced Studies, 1946, reprinted in [Swar76].

[Case85] Case, B., "Building Blocks Yield Fast 32-Bit RISC Machines," *Computer Design.* Vol. 24, No. 7, July 1, 1985, pp. 111–117.

[CoHi85] Colwell. R. P.. C. Y. Hitchcock, E. D. Jensen, et al., "Computers. Complexity. and Controversy," *Computer.* Vol. 18, No. 9, September 1985, pp. 8–20.

[Dasg89] Dasgupta, S., *Computer Architecture: A Modern Synthesis.* New York: John Wiley & Sons, Inc., 1989.

[Dasg84] Dasgupta, S., *The Design and Description of Computer Architectures.* New York: John Wiley & Sons, Inc., 1984.

[DECC77] Digital Equipment Corporation. *PDP 8/e Small Computer Handbook.* Maynard. MA: Digital Equipment Corporation, 1970.

[DECC77] Digital Equipment Corporation, *VAX11-780 Architecture Handbook,* Vol 1. Maynard. MA: Digital Equipment Corporation, 1977.

[Ditz81] Ditzel. D. R., "Reflections on the High-Level Language Symbol Computer System." *Computer.* Vol. 14, No.7, July 1981, pp. 55–66.

[FlWa86] Fleming, P. J., and J. J. Wallace. "How Not to Lie with Statistics: The Correct way to Summarize Benchmark Results," *Communications of the ACM.* Vol. 29. No. 3, March 1986. pp. 218–221.

[FoDy82] Foderaro. J.. K. van Dyke, and D. A. Patterson. "Running RISCs," *VLSI Design.* Vol. 3, No. 5, 1982, pp. 27–32.

[Folb85] Foster. C. C., and T. Iberall, *Computer Architecture.* 3rd. Edition. New York: Van Nostrand Reinhold Co.. 1985.

[FuBu77] Fuller, S. H.. and W. E. Burr, "Measurement and Evaluation of Alternative Computer Architectures," *Computer.* Vol. 10, No. 10, October 1977, pp. 24–35.

[HaVr78] Hamacher, V. C.. Z. G. Vranesic, and S. G. Zaky. *Computer Organization.* New York: McGraw-Hill Book Company. 1984.

[HaDe68] Hauck, E. A., and B. A. Dent. "Burroughs' B6500/B6700 Stack Mechanisms." *Proceedings Spring Joint Computer Conference.* 1968. pp. 245–251.

[HeJo82] Hennessy, J. L.. N. Jouppi. S. Przybylski, et al., "MIPS: A Microprocessor Architecture," *Proceedings of the 15th Annual Workshop on Microprogramming.* Los Angeles: IEEE Computer Society Press, 1982, pp. 17–22.

[Hirs84] Hirsch, A., "Tagged Architecture Supports Symbolic Processing," *Computer Design.* Vol. 23, No. 6, June 1, 1984, pp. 75–80.

[HiSp85] Hitchcock. C. Y., and H. M. B. Sprunt, "Analyzing Multiple Register Sets," *Proceedings of the 12th Annual International Symposium on Computer Architecture.* Silver Spring. MD: IEEE Computer Society Press. 1985, pp. 55–63.

[HuLa85] Hugnet, M., and T. Lang, "A Reduced Register File for RISC Architectures." *SIGARCH Computer Architecture News.* Vol. 13, No. 4, September 1985. pp. 22–31.

[Kain89] Kain, R. Y., *Computer Architecture, Software and Hardware.* Englewood Cliffs, NJ: Prentice Hall. 1989.

[Kane87] Kane. Gerry, *MIPS R2000 RISC Architecture.* Englewood Cliffs, NJ: Prentice Hall, 1987.

[Kate85] Katevenis, M. G. H., *Reduced Instruction Set Computer Architectures for VLSI.* Cambridge, MA: MIT Press, 1985.

[Kee78a] Keedy. J. L., "On the Use of Stacks in the Evaluation of Expressions." *SIGARCH Computer Archirecrure News.* Vol. 6, No. 6, February 1978, pp. 22-28.

[Kee78b] Keedy, J. L.. "On the the Evaluation of Expressions Using Accumulators, Stacks, and Store-Store Instructions," *SIGARCH Computer Architecture News,* Vol. 7, No. 4, December 1978, pp. 24–27.

[Keed79] Keedy, J. L., "More on the Use of Stacks in the Evaluation of Expressions." *SIGARCH Computer Architecture News.* Vol. 7, No. 8, June 1979, pp. 18–22.

[Lang82] Langdon, G. G.. Jr.. *Computer Design* San Jose, CA: Computeach Press, Inc, 1982.

[LoKi61] Lonergan, W.. and P. King. "Design of the B 5000 system," *Datamation.* Vol. 7, No. 5, May 1961. pp. 28–32.

[Lund77] Lunde, A., "Empirical Evaluation of Some Features of lnstruction Set Processor Architectures." *Communications of the ACM.* Vol. 20, No. 3, March 1977, pp. 143–152.

[Mano82] Mano, M. M., *Computer System Architecture.* Englewood Cliffs. NJ: Prentice Hall, 1982.

[Myer77] Myers, G. J., "The Case Against Stack-Oriented lnstruction Sets," *SIGARCH Computer Architecture News.* Vol. 6, No. 3, August 1977. pp. 7–10.

[Myer78] Myers, G. J., "The Evaluation of Expressions in a Storage-Storage Architecture," *SIGARCH Computer Architecture News.* Vol. 6, No. 9, June 1978, pp. 20–23.

[Myer82] Myers, G. J., *Advances in Computer Architecture.* New York: John Wiley & Sons. Inc., 1982.

[Patt85] Patterson. D. A.. "Reduced Instruction Set Computers," *Communications of the ACM.* Vol. 28, No. 1, January 1985, pp. 8–21.

[PaSe82] Patterson, D. A., and C. Sequin, "A VLSI RISC," *Computer.* Vol. 15, No. 9, September, 1982, pp. 8–21.

[PaSe81] Patterson, D. A., and C. Sequin, "RISC I: A Reduced Instruction VLSI Set Computer," *Proceedings of the 8th Annual International Symposium on Computer Architecture.* New York: IEEE Computer Society Press, 1981, pp. 443–458.

[PaDi80] Patterson, D. A.. and D. Dietzel, "The Case for the Reduced Instruction Set Computer," *SIGARCH Computer Architecture News (SIGARCH).* Vol. 8, No. 6, 1980, pp. 25–33.

[PaPi82] Patterson, D. A.. and R. Piepho. "RISC Assessment: A High Level Language Experiment." *Proceedings of the 9th Annual Symposium on Computer Architecture.* New York: IEEE Computer Society Press, 1982. pp. 3–8.

[PeSh77] Peuto. B. L., and L. J. Shustek, "An Instruction Timing Model of CPU Performance." *Proceedings of the 4th Annual Symposium on Computer Architecture.* New York: ACM/IEEE, March 1977, pp. 165–178.

[Radi82] Radin, G., "The 801 Minicomputer." *Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems.* New York: ACM, 1982, pp. 39–47.

[RiSm71] Rice, R., and W. R. Smith. "SYMBOL-A Major Departure from Classic Software Dominated von Neumann Computing Systems," AFIPS Conference Proceedings, 1971 SJCC, Vol. 38. Montvale, NJ: AFIPS Press, 1971, pp. 575–587.

[SaCh81] Saur, C. H., and K. M. Chandy, *Computer Systems Performance Modeling.* Englewood Cliffs, NJ: Prentice Hall, 1981.

[Schn85] Schneider. G. M., *The Principles of Computer Organization.* New York: John Wiley & Sons, Inc., 1985.

[ShKa84] Sherbume, R. W., Jr, M. G. H. Katevenis, D. A. Patterson, and C. H. Sequin, "A 32-Bit NMOS Processor with a Large Register File," *IEEE Journal of Solid State Circuits.* Vol. SC-19, No. 5, October 1984, pp. 682–689.

[Shus78] Shustek, L. J.. "Analysis and Performance of Computer Instruction Sets," Ph.D. Dissertation, Stanford. CA: Computer Systems Laboratory, Stanford University, 1978.

[SiBe82] Siewiorek, D. P., C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples.* New York: McGraw-Hill Book Company. 1982.

[Ston80] Stone. H. S. (Ed.), *Introduction to Computer Architecture.* Chicago. IL: Science Research Associates. 1980.

[Swar76] Swartzlander, E. E., Jr. (Ed.). *Computer Design Development: Principal Papers.* Rochelle Park, NJ: Hayden Book Company, 1976.

[Tibe84] Tiberghien, J. (Ed.), *New Computer Architectures.* San Diego, CA: Academic Press. 1984.

[Wall85] Wallich, P., "Toward Simpler, Faster Computers," *IEEE Spectrum.* August 1985. pp. 38–45.

[Wilk83] Wilkes, M. V., "Size, Power, and Speed." *Proceedings of the 10th Annual International Symposium on Computer Architecture.* Silver Spring, MD: IEEE Computer Society Press. 1983, pp. 2–4.