CHAPTER 5

Software Design: Programming Methods and Techniques

9900 ARCHITECTURE

The 9900 system is illustrated in *Figure 5-1.* The major subsystems are the 9900 processor, the memory for program and data storage, and input and output devices for external communication and control. The processor controls the fetching of data and instructions from memory or input devices and the transferring of data from one location to another. The data and instructions are transferred 16 bits at a time in groups called words. These words are addressed or located by signals on the 15 address lines A_0 through A_{14} (called the address bus). A 15 binary bit address will select one of 32,768 memory words.

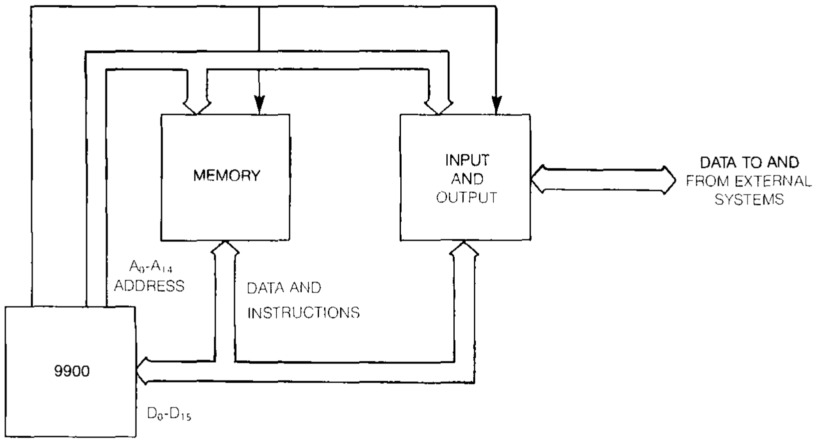


Figure 5-1. *General 9900 System Structure*

Internally, the processor generates a 16 bit address but the least significant bit, A_{15}, is not sent to the memory. Each word is further broken down into two 8 bit groups called bytes as shown in *Figure 5-2.* The first 8 bit byte of a word is located at an even address ($A_{15} = 0$). The second 8 bit byte is located at an odd address ($A_{15} = 1$). The byte selection is done internally in the processor once the full 16 bit data word is obtained from one of the 32,768 word locations in memory. Byte addressing is used only on instructions that perform byte operations; most 9900 instructions are word operations.

The processor contains certain basic elements as shown in *Figure 5-3*. The timing and control section is of primary interest to the hardware designer who must make certain that all system events occur in the correct order and at the correct time. The software designer is interested in what operations the ALU provides and the registers that determine the instruction and data addresses. These registers are the program counter, the status register, and the workspace pointer. In addition, the instruction register is of interest in understanding the basic instruction cycle of the processor. The 9900 contains other registers such as data address registers, ALU scratchpad registers, and so on. The processor also provides hardware to decode instructions, control the ALU operation, and to control the CRU input and outputs. These components all work together to provide the basic instruction fetch and execution cycle of the processor.

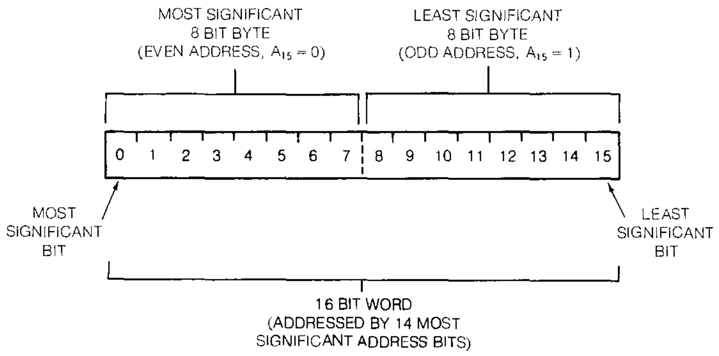


Figure 5-2. 9900 Words and Bytes

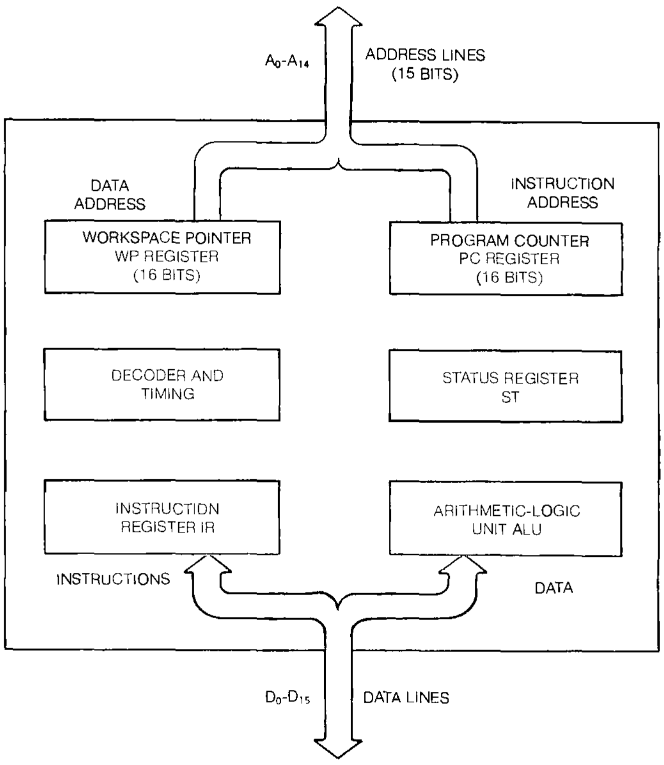


Figure 5-3. Basic 9900 Elements.

Instruction Register and Cycle

The instruction cycle that is performed over and over again by the processor consists of the following basic operations:

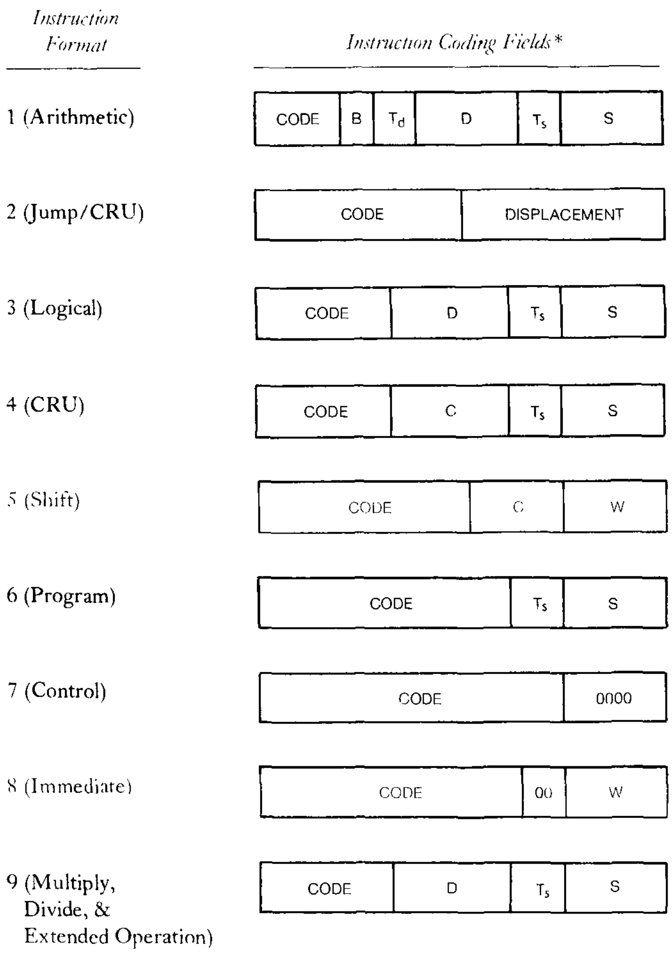
1) *Instruction Fetch* — the contents of the program counter are sent out on the address lines and a memory read is performed. The 16 bit instruction operation code word is sent from the memory along the data lines D_0 through D_{15} and is latched in the processor instruction register.

2) *Instruction Execution* — The instruction is decoded and executed. Usually, the address of the data to be operated on (source data) is generated and a memory read cycle is performed to get the data into the processor. Then a destination address is generated and a memory write cycle is performed to store the result of the operation at a desired destination memory location.

3) The contents of the program counter are changed to indicate the address of the next instruction and the processor returns to the instruction fetch operation.

This sequence is repeated continually as long as power is supplied to the processor.

The number of memory references required in the instruction operation depends on the format that is used for the instruction. Instructions can have one of 9 such formats as illustrated in *Figure 5-4.* The instruction code indicates to the processor how many memory references are required to get all the information needed by the instruction. The first memory read obtains the instruction code which determines which operation is to be performed and how the data is located. A second and possibly a third memory read may be required to obtain values or addresses for the data to be used in this operation. An immediate instruction (format 8) consists of two successive memory words: the first for the instruction code and a second word that contains the data constant to be used. Other instruction formats contain a T_s and/or a T_d field to indicate the existence of data addresses as part of the instruction. If a T_s or T_d two bit field contains a 10_2, the address of the source or destination locations or both will be contained in the one or two memory locations immediately following the instruction code word as illustrated in *Figure 5-5.* In these cases, one or two additional memory reads are required to fetch these addresses for use by the instruction to locate data in memory. Obviously, the more memory references required to get all of the instruction, the longer the execution time for that instruction. The programmer also needs to be aware of the number of words of memory required for each instruction in order to estimate program memory requirements.



*The Fields are defined as follows:
- CODE — Indicates the bits defining the operation code
- B — Byte/Word Indicator (Single bit)
- D — Workspace Register of the destination code (4 bits)
- T_d — Addressing mode of the destination operand (2 bits)
- S — Workspace Register of the source operand (4 bits)
- T_s — Addressing mode of the source operand (2 bits)
- C — Shift or Bit count (4 bits)
- W — Indicates Workspace Register to be used (4 bits)

Figure 5-4. 9900 Instruction Formats

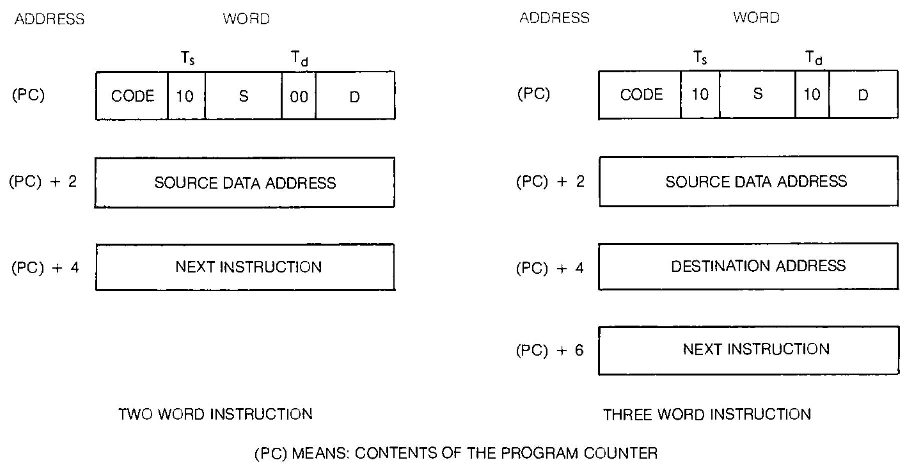


Figure 5-5. Example Memory Requirements for Format 1 Instructions.

Program Counter (PC)

The program counter, abbreviated PC, contains the address of the instruction to be executed as illustrated in *Figure 5-6.* Normally, after executing an instruction, the contents of the program counter are incremented by two to locate the next instruction word in sequence in memory. The programmer can control the contents of the program counter (and thus control where the next instruction is to be found) by using branch or jump instructions. These instructions offer the alternatives of taking the next instruction in sequence or jumping to another part of program memory for the next instruction.

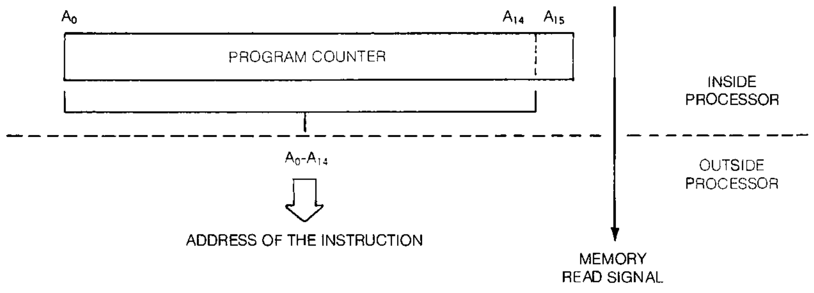
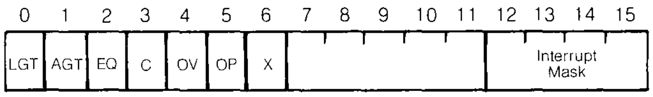


Figure 5-6. Purpose of the Program Counter

Status Register (ST)

The purpose of the status register is to store the general arithmetic and logic conditions that result from the execution of each instruction. This information lets the programmer know if the last operation caused a result equal to or greater than some reference number (often zero). It includes the information regarding the sign of the result (was it a negative or a positive number), the parity of the result (an odd or even number of one bits), and if a carry or overflow occurred (indicating that the 16 bit word length was insufficient to hold the result). The status register also contains a 4 bit code known as the interrupt mask which defines which of 16 hardware subsystem interrupt signals will be recognized and responded to by the processor. The information contained in the status register is defined in *Figure 5-7.*



Status Register Bit			
0	LGT	–	*Logical Greater Than* — set in a comparison of an unsigned number with a smaller unsigned number.
1	AGT	–	*Arthmetic Greater Than* — set when one signed number is compared with another that is less positive (nearer to −32,768).
2	EQ	–	*Equal* — set when the two words or two bytes being compared are equal.
3	C	–	*Carry* — set by carry out of most significant bit of a word or byte in a shift or arithmetic operation.
4	OV	–	*Overflow* — set when the result of an arthmetic operation is too large or too small to be correctly represented in 2's complement form. OV is set in addition if the most significant bit of the two operands are equal and the most significant bit of the sum is different from the destination operand most significant bit. OV is set in subtraction if the most significant bits of the operands are not equal and the most significant bit of the result is different from the most significant bit of the destination operand. In single operand instructions affecting OV, the OV is set if the most significant bit of the operand is changed by the instruction.
5	OP	–	*Odd Parity* — set when there is an odd number of bits set to one in the result.
6	X	–	*Extended Operation* — set when the PC and WP registers have been to set to values of the transfer vector words during the execution of an extended operation.
7-11		–	Reserved for special Model 990/10 computer applications.
12-15		–	*Interrupt Mask* — All interrupts of level equal to or less than mask value are enabled.

Figure 5-7. TMS9900 Status Register Contents

Workspace Pointer (WP)

This register addresses the first word in a group of 16 consecutive memory words called a workspace as illustrated in *Figure 5-8.* These workspace words are called workspace registers and are treated by the processor as if they were registers on the processor chip. These workspace registers can be used as accumulators for arithmetic operations or for storage of often used data. When the workspace register contains the data used by the instruction, the T_s or T_d fields in the instruction format (see *Figure 5-4*) are 00. This way of locating instruction operands is an addressing mode called workspace register addressing. The workspace register can also be used to store the address of the data to be used instead of storing the data itself. In this case the T_s or T_d fields of the instruction code or format will be 01. This type of addressing (method of data location) is known as register indirect addressing. Workspace registers 1 through 15 can also be used to store the base address to which an offset will be added to determine a data address. This type of addressing is called indexed addressing and the T_s or T_d fields for this type of addressing will be a 10.

Some of the workspace registers are reserved for specific tasks as shown in *Figure 5-8.* If a certain type of subroutine branch called a branch and link (BL) is performed, register 11 is used to save the contents of the program counter at the time of the branch. In another type of subroutine branch, the branch and link workspace (BLWP) instruction, registers 13, 14, and 15, are used to save the values of WP, PC, and ST registers, respectively, that were in the processor at the time the branch instruction occurred. These registers then allow the programmer to return to the situation or program context that existed prior to the branch. Register 12 is used to form the address of certain input and output bits that make up part of the communications register unit (CRU) subsystem.

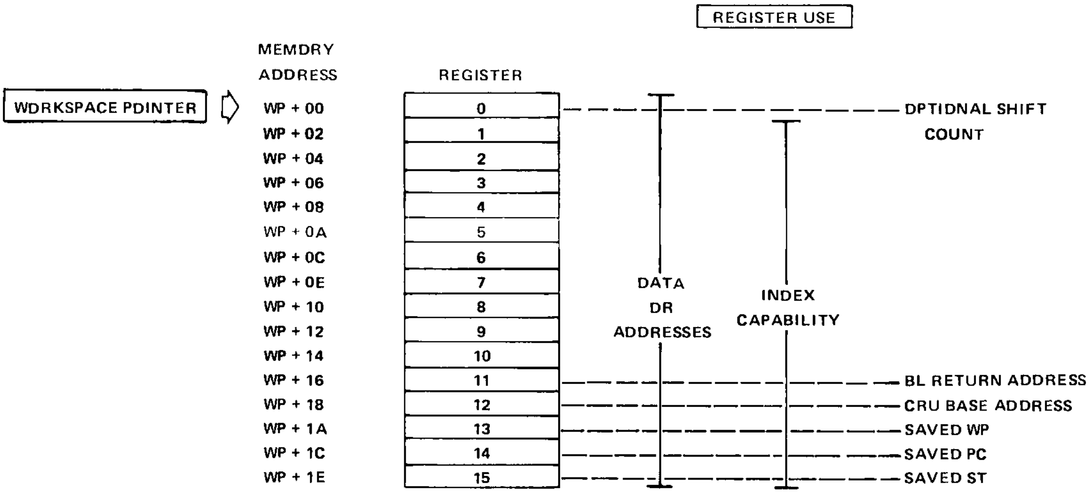


Figure 5-8. 9900 Workspace Structure

The relationships between the workspace registers and the instruction operations must be understood by the programmer to effectively utilize the 9900. Much of the addressing of data involves the use of workspace registers and branch and input/output instructions must use the dedicated registers 11 through 15. The use of the workspace in performing the basic program functions offered by the 9900 will be covered in detail throughout this chapter.

Program Environment or Context

The contents of the three processor registers (PC, WP, and ST) completely define the status of the system program at any given time. As illustrated in *Figure 5-9*, the program counter keeps track of that part of the system program currently being executed by specifying the current instruction location. The status register keeps track of the logical and arithmetic conditions that result from the execution of each instruction. The workspace pointer keeps track of the location in memory of the sixteen general purpose workspace registers currently being used by the program. The contents of the processor and workspace registers define the current program environment or *context* of the system. A change in the contents of these registers will change the environment to a new part of program memory and a new workspace area. Thus, the system will be switched to a new environment or program context by such a change. Similarly, by restoring the contents of PC, WP, and ST to original values, the program environment will be switched back to the original context and continue executing in the original program environment.

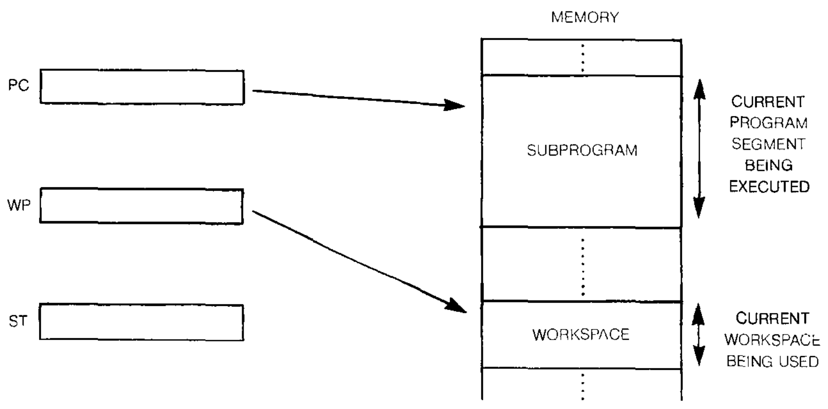

MEMORY
PC
WP
ST
SUBPROGRAM
CURRENT PROGRAM SEGMENT BEING EXECUTED
WORKSPACE
CURRENT WORKSPACE BEING USED

Figure 5-9. Program Context

MEMORY ORGANIZATION

The 9900 system memory must provide storage locations for the system program and subprograms and storage for system data. Since the physical devices used for storing instructions are often a different type of memory device from those used to store data, the program is usually stored in consecutive blocks of memory separate from the blocks of data. This is illustrated in *Figure 5-10.* Also shown in *Figure 5-10* are groups of memory locations that must be reserved for program and workspace addresses used by certain subprograms. Thus, the memory is subdivided into three types of storage of storage locations: program memory, data memory, and reserved or dedicated memory.

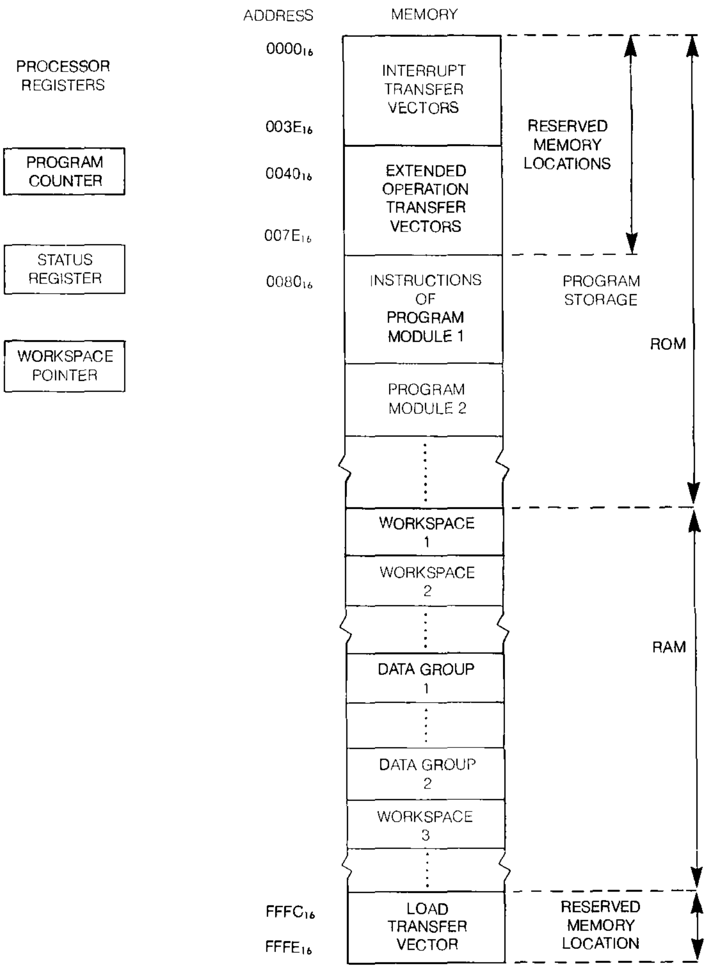


Figure 5-10. 9900 Memory Organization

RAM/ROM Partitioning

The program storage should be non-volatile so that the system program is not lost when the system power is turned off. Further, it is often desirable for the program memory to be a read-only memory or ROM. High volume read-only memory devices are mask programmable by the manufacturer. Alternatively, the program storage can be placed in a programmable read only memory (PROM). These devices may be economically programmed in smaller quantities. Programming may be performed by the user or by the distributor of the devices. Since both PROM and ROM devices provide word storage in consecutive addresses and the processor executes programs by going through instructions in sequence, the instructions that comprise a given subprogram should be placed in consecutive addresses in a block of memory words called a program module. It is not necessary that all program modules be adjacent to each other in memory, though certainly it is reasonable to do so.

System data storage, excluding input/output registers, provide storage for data being processed by the system program. These storage locations are usually located in consecutive blocks of memory. Since the data memory must provide both read and write capability, it is often called read-write memory. A more common terminology is random access memory or RAM, though this is somewhat misleading, since the program memory in ROM may also be randomly accessed.

The range of addresses that are assigned to the RAM storage locations and those that are assigned to the ROM locations are somewhat arbitrary. The reserved locations are the first locations in program memory, so that part of the ROM addresses are these reserved location areas. Often hardware considerations such as the simplification of the address decoding circuitry may decide the range of addresses that are used for each type of memory.

Reserved Memory

The program modules, workspaces, and general data storage can generally be placed anywhere in memory, as long as the following reserved locations are preserved:

1) The first 32 words of memory (addresses 0 through $3E_{16}$) are reserved for interrupt transfer vectors.
2) The next 32 words of memory (addresses 40_{16} through $7E_{16}$) are reserved for extended operation transfer vectors.
3) The last two words of memory (addresses $FFFC_{16}$ and $FFFF_{16}$) are reserved for a load or reset transfer vector.

These transfer vectors provide storage for a value to be placed in the workspace pointer and a value to be placed in the program counter in order to switch the program context from its current environment to a subprogram and new workspace. This new subprogram and workspace context is used to respond to a hardware interrupt signal, a hardware reset signal, or an instruction called an extended operation (XOP).

WORKSPACE UTILIZATION

The Workspace Concept and Uses

The advanced memory-to-memory architecture of the 9900 affords multiple register files in main memory for efficient data manipulation and flexible subroutine linkage. The usage of the workspace must follow certain constraints for optimum performance. Each workspace is a contiguous block of 16 words in main memory. All 16 general purpose registers are available to the programmer for use in any of four ways:

1) *Operand Registers* — to contain data for arithmetic and logical operations.
2) *Accumulators* — to store intermediate results of arithmetic operations.
3) *Address Registers* — to specify memory location of operands.
4) *Index Registers* — to provide an offset from a base address to define an operand location.

The workspace pointer in the processor contains the address of workspace register 0. The address of any workspace register R is:

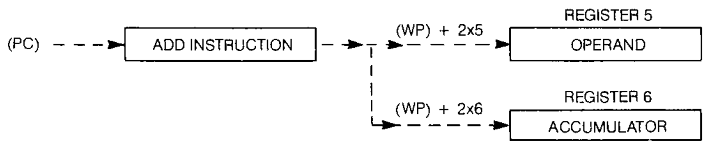
Memory Address of Register R = (WP) + 2R

where (WP) means the contents of the workspace pointer.

5 When a workspace register is specified as an operand in an instruction, (workspace register addressing mode) the workspace register contains binary data for use by the instruction. As an example, consider the addition of the data in register 5 to the data in register 6. The instruction format is:

```
        A         5,6
```

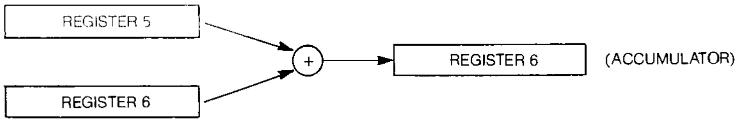
with address calculations of:



which is interpreted as follows:

1) The contents of the program counter addresses the instruction in ROM.
2) The instruction indicates workspace register addressing causing the calculation of the workspace addresses to locate the data to be used by the instruction (contained in registers 5 and 6) in RAM.

The resulting hardware operation with the data thus located is:

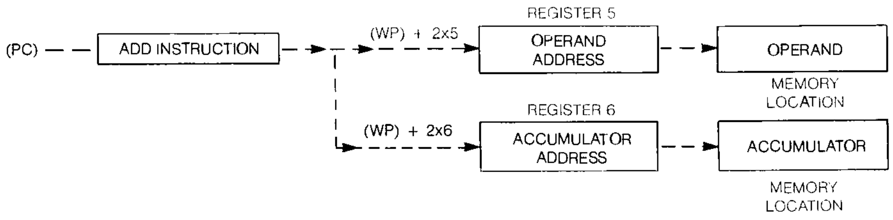


In this example, register 5 is functioning as an operand and register 6 is functioning as an accumulator. The difference between an operand and an accumulator register is that operands remain unchanged by an operation, while accumulators assume new values, the result of the operations.

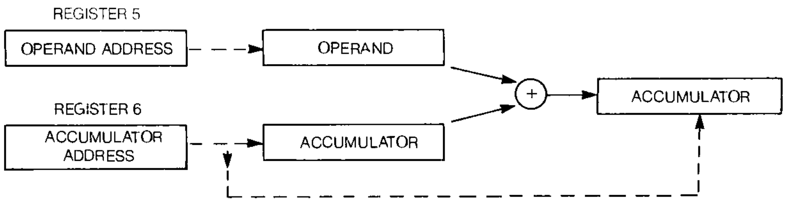
The contents of a workspace register may be the address of an operand or an accumulator in main memory. Address registers are accessed through workspace register indirect addressing, with or without autoincrementing. If autoincrementing is not used, the content of the workspace register (the address of the data) is not changed by the operation. If autoincrementing is used, the address contained in the workspace register is incremented by one for byte operations and by two for word operations. An example of an addition instruction in which both the operand and the accumulator are specified by register indirect addressing is:

```
        A              *5,*6
```

with the address computations:



with the resulting hardware operation:

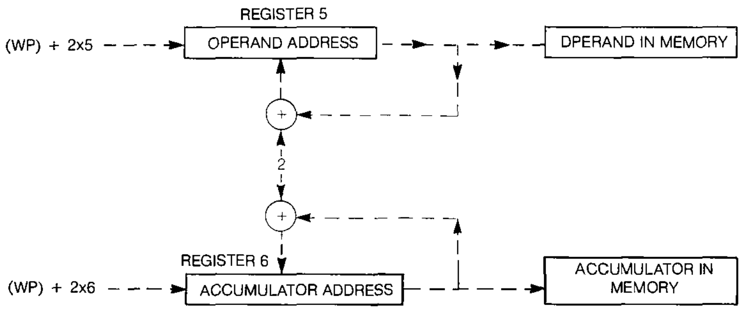


The contents of the address registers are not changed in execution since autoincrementing is not used.

Autoincrementing is often used when accessing structured data and data arrays. To add this feature to this example, the following format would be used:

```
        A          *5+, *6+
```

which would result in the same events as described for standard workspace register indirect addressing with the addition of an incrementing by two of the contents of the address register 5 and 6:

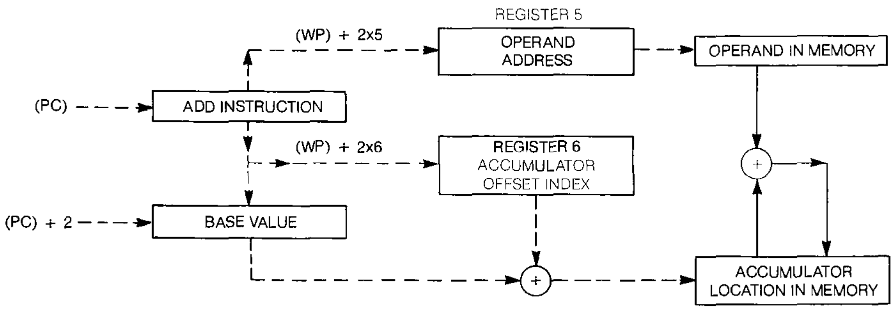


The addresses are modified (incremented by two) after the operand and accumulator addressing operations are completed.

When the workspace register is used as an index register, its contents specify an offset from a base address. The sum of this offset and the base address contained in the instruction defines the memory location of program data. Workspace registers act as index registers when the indexed addressing mode is used. The only restriction on the use of workspace registers as index registers is that register 0 cannot be used as an index register. An example of using register 5 as an indirect address register for the operand and register 6 as an index register for addressing the accumulator would be:

```
        A          *5, @BASE (6)
```

The binary address BASE is the second word of the two word add instruction, with address calculations as follows:



The operand data is added to the accumulator data and the sum is stored in the accumulator location.

Dedicated Areas of Workspaces

Any register of a workspace may be used as a general purpose register (with the exception of register 0 not being available as an index register). A few of the registers are used by 9900 hardware in certain ways, and the software designer must observe these constraints to assure the integrity of stored data and program and hardware linkages. *Figure 5-11* shows the way the workspace is viewed by the hardware.

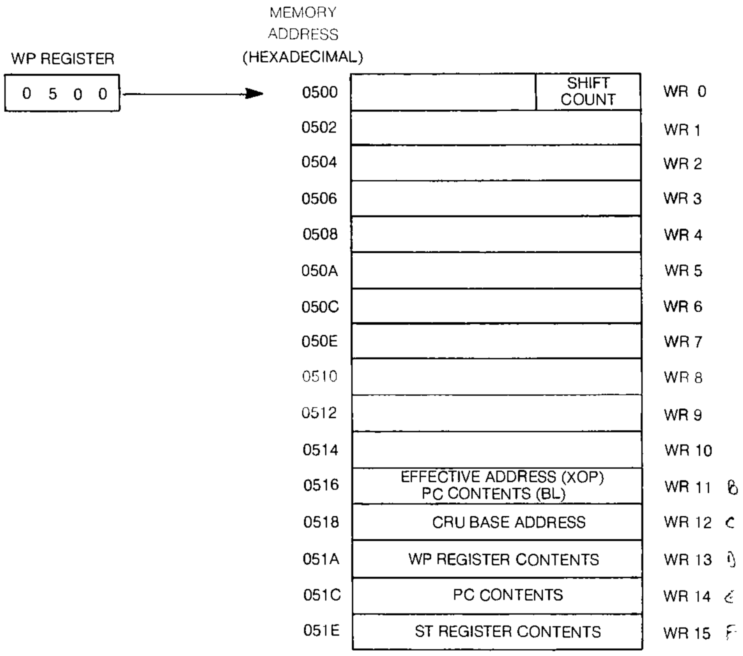


Figure 5-11. Reserved Areas of 9900 Workspaces

An examination of *Figure 5-11* reveals the following areas that may have to be reserved in a workspace:

Registers 13, 14, and 15 — Context Switches

These three workspace registers are loaded with current values of the workspace pointer, program counter, and status register with each context switch. A context switch occurs in response to an interrupt or in executing a BLWP or XOP instruction. When an RTWP return instruction is executed, the processor restores these values to the processor registers from the last three workspace registers. To insure that this return linkage is not destroyed, the programmer must insure that subprogram operations or subsequent context switches do not alter the contents of registers 13, 14, or 15.

Register 0 — Shift Instruction

Bits 12 through 15 of register 0 may specify a bit count for shift instructions. The 9900 shift instructions have the format:

```
        OPCODE      R, SCNT
```

where the OPCODE is one of the shift instruction mnemonics SLA, SRC, SRL, or SRA, R is the operand register, and SCNT specifies the number of bit position to be shifted. When SCNT is zero, bits 12 through 15 of register 0 specifies the shift count. If both SCNT and bits 12 through 15 of register 0 are zero, a 16 bit shift will occur.

Register 11 — XOP and BL Instructions

Register 11 is used to save address information in extended operation instructions (XOP) and Branch and Link subroutine jump (BL) instructions. The BL instruction provides a means of subroutine linkage without the overhead of a context switch. Previous contents of register 11 are replaced with the program counter contents when a BL occurs. Return to the calling procedure is accomplished with the RT pseudo-instruction or by an indirect branch B *11. No critical data should be stored in register 11 if a BL instruction is to be executed.

In the case of the extended operation instruction, an address is passed to register 11 during the XOP context switch. For example:

```
        XOP      VAR, OPNUM
```

OPNUM is the XOP number and locates the XOP transfer vector in main memory through the formula:

$$\text{Transfer Vector Address} = 40_{16} + 4 \times \text{OPNUM}$$

The effective address of the source operand VAR is placed into register 11 of the XOP workspace. Even if VAR is not provided, register 11 contents will be altered by executing an XOP instruction.

Register 12 — CRU Bit Addressing

The 9900 communications register unit (CRU) is a direct command-driven I/O interface. The five CRU instructions (SBO, SBZ, TB, LDCR, and STCR) all depend on the presence of a CRU hardware base address in bits 3 through 14 of workspace register 12. None of these instructions alter the content of register 12.

Workspace Location

Workspaces may be located anywhere in main memory. In practice, 66 words of memory are reserved to implement necessary hardware functions (transfer vectors). Workspaces and data may be stored in any other memory area, known as general memory. The memory locations reserved for 9900 transfer vectors for interrupts and extended operation instructions are memory addresses 0000_{16} through $007E_{16}$. The last two words of memory (addresses $FFFC_{16}$ and $FFFE_{16}$) are reserved for a load function transfer vector, so the last data or instruction word can occur at address $FFFA_{16}$.

Within general addresses 0080_{16} through $FFFA_{16}$, workspaces can be independent, or used in common by different program segments or subprograms. To reduce memory requirements of a software system, routines can share workspaces. The effect of a BL call to a subroutine is illustrated in Figure 5-12. The program counter is changed to fetch the instructions from the subroutine, but the *workspace pointer is not changed,* which results in a workspace shared by the called and the calling procedures.

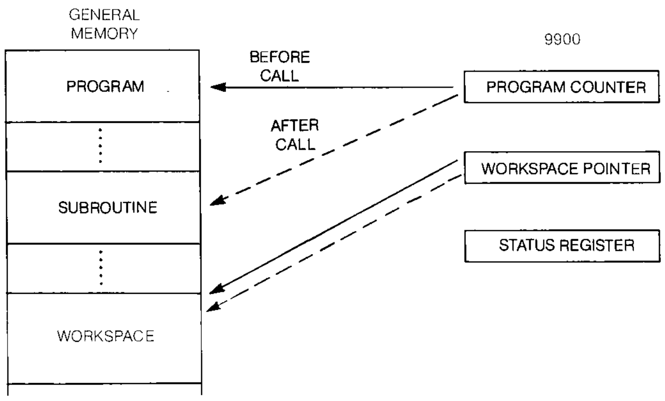


Figure 5-12. Shared Workspace Subroutine Call

When a routine requires the use of a large number of workspace registers, an independent workspace will be needed for that routine. In some cases, independent workspaces are used for routines when little common data is needed. When workspaces have no common memory words, parameter or data passing can be done by using the old program counter and workspace pointer. For example, in a context switch, which saves the old workspace pointer in the new workspace register 13, any of the old workspace registers can be accessed by referring to the contents of the new register 13. The contents of register 13 addresses the old workspace register 0. The use of register 13 as an index register allows the programmer access to any other of the old workspace registers. Thus, to access old register 0 as an operand in an add instruction, the following instruction would be used:

```
        A           *13,7
```

This instruction specifies the contents of old register 0 (addressed by the contents of new register 13) as an operand and new register 7 as an accumulator. To address old register 10, the following indexed addressing approach could be used:

```
        A           @20(13),7
```

This instruction adds 20 to the contents of new register 13 to generate the address of old workspace register 10, which is then used as an operand in the add operation. The effect of a context switch in providing an independent workspace is illustrated in *Figure 5-13.*

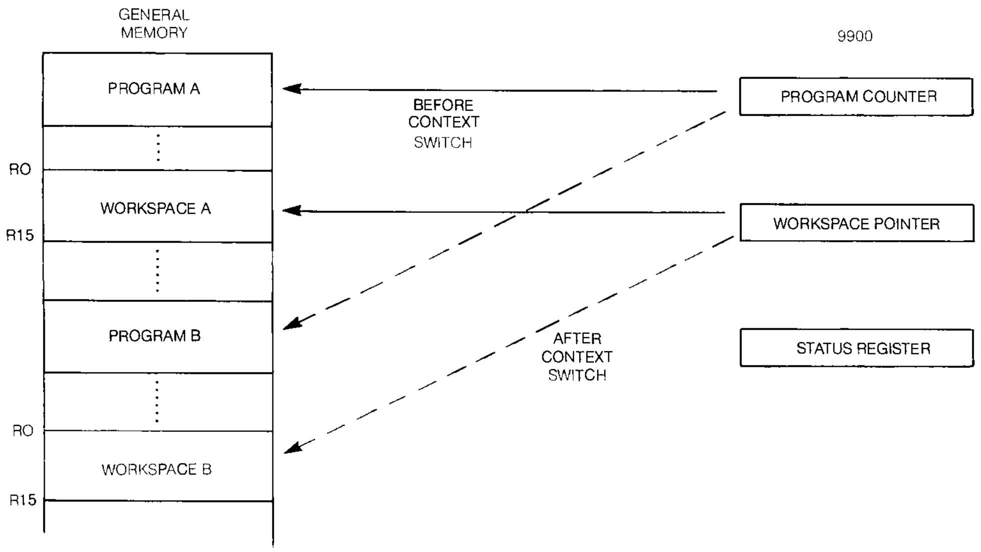


Figure 5-13. Independent Workspaces

SUBROUTINE TECHNIQUES

Software systems are implemented with a set of subprograms, usually subroutines. Subroutines offer several advantages over incorporation of all code into a large main program:

1) Repetition of code is reduced. Modular coding of repeated processes saves memory requirements of software.
2) Documentation is simplified. The clarity of complex programs is enhanced by breaking the overall task into manageable subsystems.
3) Debugging time is reduced. A complicated system can be made functional one module at a time.

These advantages point out the importance of understanding the characteristics of 9900 subroutine calls. The most important characteristics are the way the subroutine linkages back to the calling program are handled and the way parameters are passed between the calling program and the subroutine. The linkage procedures for the types of subroutine calls are discussed first.

Types of Subroutines

Three types of subroutine calls are used with the 9900. The following table summarizes the calls and returns for each type:

Call to Subroutine		*Return to Calling Procedure*	
Mnemonic	*Meaning*	*Mnemonic*	*Meaning*
BL	Branch & Link	RT or B *11	Return
BLWP	Branch & Link Workspace Pointer	RTWP	Return with Workspace Pointer
XOP	Extended Operation	RTWP	Return with Workspace Pointer

The branch and link instruction is a fast transfer to a routine that shares the workspace with the calling procedure. Execution of a BL causes the contents of the program counter to be stored in workspace register 11. The new program counter value is the single argument of the BL instruction. An example of a typical BL instruction is:

```
        PT          BL          @SUB1
```

SUB1 is the label of the first instruction of the subroutine being called. After execution of the BL instruction, program flow will continue at the symbolic address SUB1. Upon execution of the BL instruction, the update value of the program counter (address PT+4) is stored in workspace 11 (PT is the symbolic address of the BL instruction). This process of a shared workspace subroutine call is illustrated in *Figure 5-14.* Return to the calling procedure is through the RT pseudo-instruction which is equivalent to the indirect branch;

```
        B           *11
```

Since the BL instruction always reloads Workspace register 11, special steps must be taken to insure that the critical return address is not overstored. Generally, register 11 should not be used to save a variable whose value will be needed after a BL instruction occurs. Similarly, after a BL instruction has been executed (and before a RT instruction has been executed), register 11 cannot be used by any instruction that would change the contents of register 11, such as using register 11 as an accumulator or executing another BL instruction. If multiple levels of BL calls are to be used, a push-down stack must be established to save intermediate return linkage. Techniques for setting up a stack are discussed under the topics of multiple level subroutine calls and reentrancy.

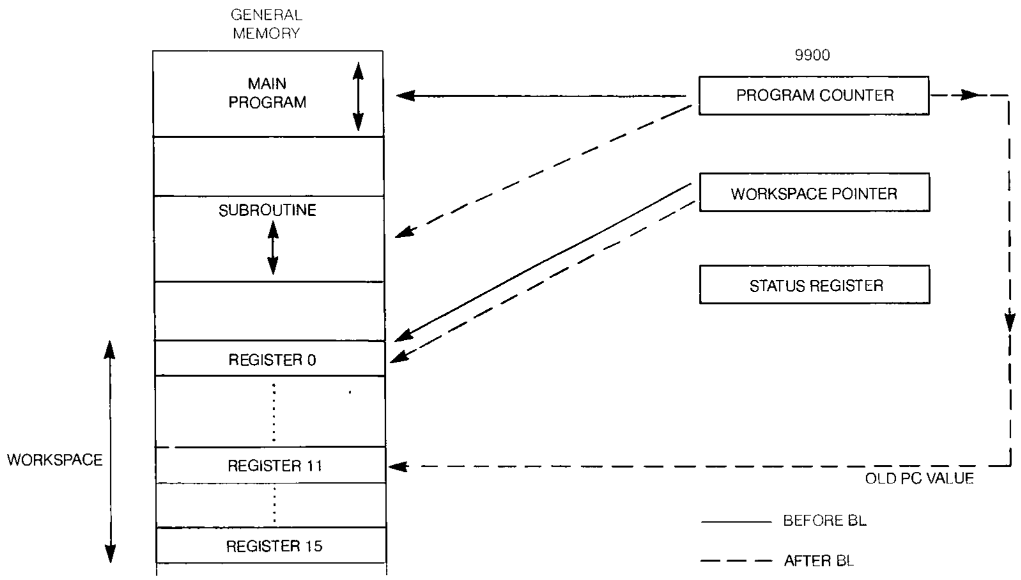


Figure 5-14. Effects of BL Instruction

The branch and load workspace pointer (BLWP) is a subroutine call that initiates a context switch. When a context switch occurs, the programming environment is changed to allow the subroutine to use a new register file (workspace). BLWP has the following effect as illustrated in *Figure 5-15:*

1) A transfer vector located by the argument of the BLWP instruction supplies a new workspace pointer value and program counter value.
2) The old values of WP, PC, and ST are saved in registers 13, 14, and 15, respectively, of the new workspace.
3) Execution proceeds in the subroutine using the new PC value.

The 9900 format for a typical BLWP using Symbolic addressing is:

```
        PCL      BLWP   @TVAL
```

where PCL is an arbitrary label and the symbolic address of the location of the BLWP instruction in general memory. TVAL is the symbolic address of the transfer vector, which in turn provides new values for the workspace pointer and the program counter. The contents of workspace register 13 through 15 of the new workspace are reserved for storage of the return linkage. Since the BLWP can store return linkage in an independent workspace, multiple subroutine levels may be implemented without a return stack as long as no two subroutines use the same workspace (transfer vector). Although the example in *Figure 5-15* uses symbolic addressing mode, other addressing modes can be used.

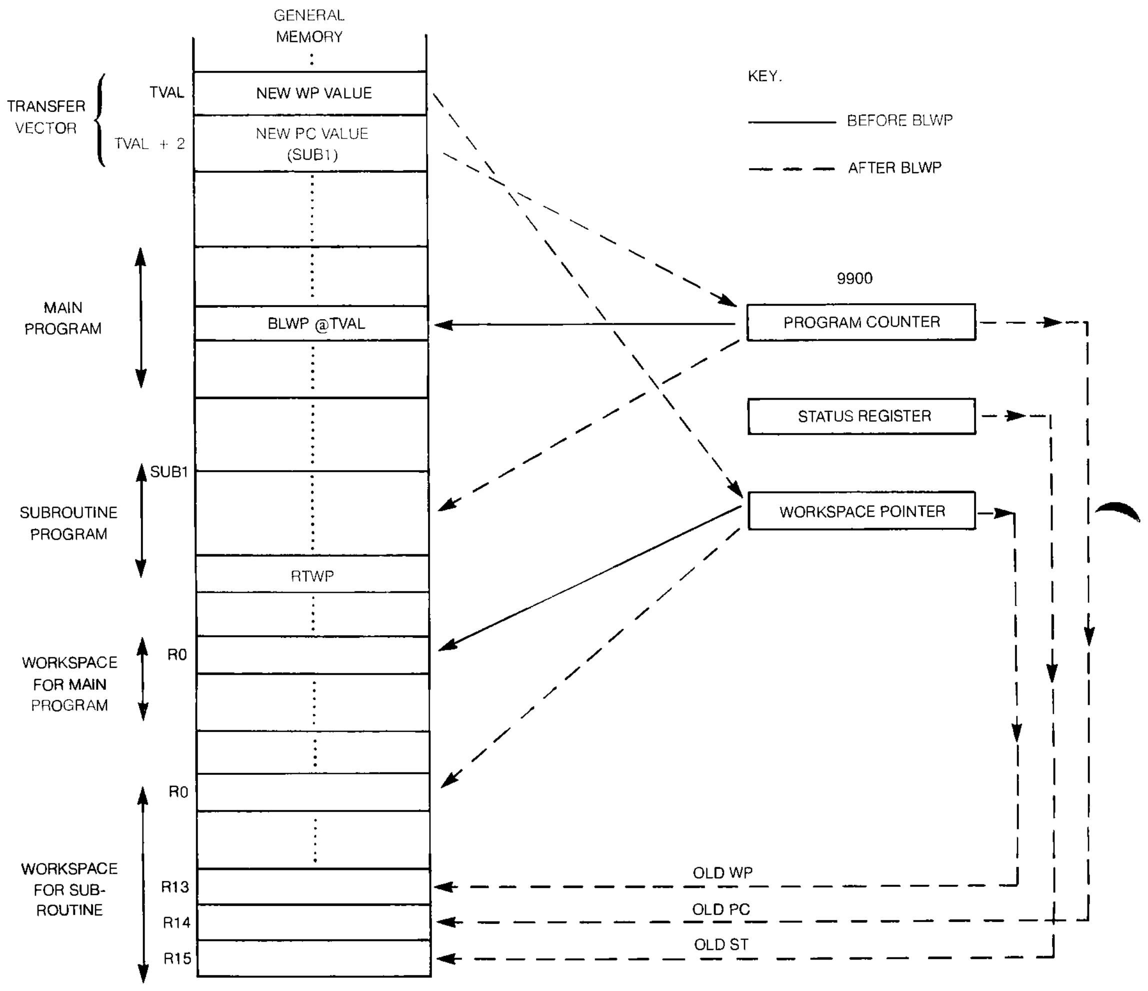


Figure 5-15. Execution of BLWP Instruction (BLWP @TVAL)

Extended operation instructions (XOP) offer a means of expanding the 9900 instruction set. The implementation of an XOP is similar to the execution of a BLWP; *the instructions differ only in the location of the transfer vector and in the parameter passing feature offered by the XOP.* The execution of an XOP is illustrated in *Figure 5-16* and consists of the following events:

1) Identify the XOP number (N) and locate a transfer vector in memory at the address $0040_{16} + 4 \times N$.
2) Use the transfer vector word one as the new workspace pointer value and the second word of the vector as the new program counter value.
3) Save the old contents of WP, PC, and ST in new workspace registers 13, 14, and 15, respectively.
4) Store the effective address of the source operand in new workspace register 11.

Thus, XOP initiates a context switch with the added benefit of direct passing of a parameter address to the new workspace (register 11). By using an assembler directive DXOP, the user can define a mnemonic string to present one of the 16 XOP transfer vectors. This mnemonic can then be used in the program as a user defined instruction, improving the clarity of the program coding. For example, to define XOP 15 as the mnemonic SAMPL, the following directive can be used:

```
        DXOP   SAMPL, 15
```

Then, instead of using the standard XOP entry in the program:

```
        XOP     @PARAM, 15
```

The programmer can insert the newly defined mnemonic:

```
        SAMPL   @PARAM
```

The XOP call is a software trap to a user-defined routine. It functions as though the routine were a single instruction added to the 9900 set of operation codes, hence the name "extended operation."

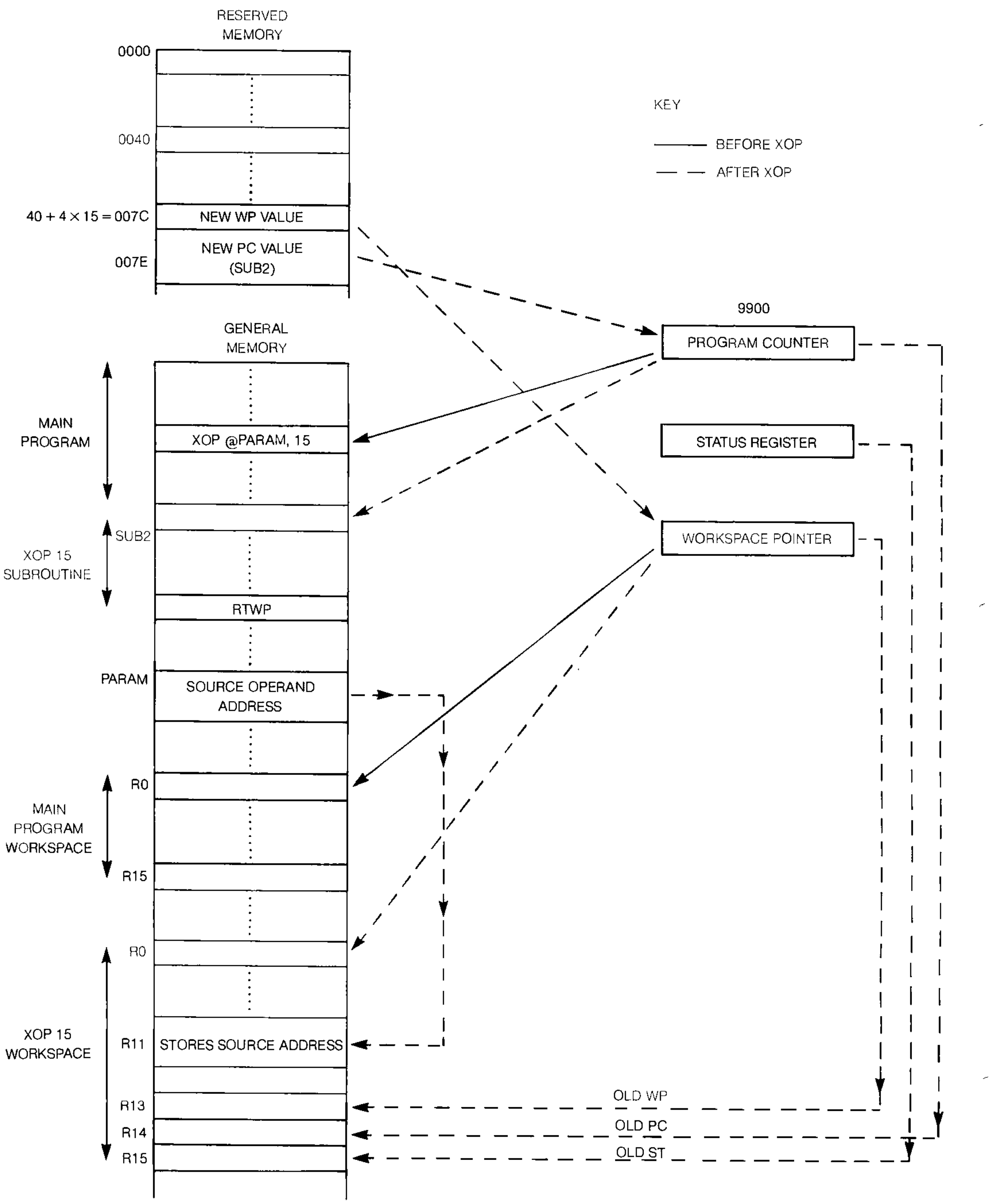


Figure 5-16. *Execution of XOP Instruction (XOP @PARAM, 15)*

▶5

Parameter Passing

Most subroutines require access to data generated by the calling procedure. Different subroutine call types mandate different parameter passing techniques. All three types of subroutine calls, BL, BLWP, and XOP, *transfer the old contents of the program counter to the called procedure.* This return linkage provides a powerful tool for parameter passing as illustrated in *Figure 5-17.* A parameter list can be assembled in a block of words following the call and accessed through the old program counter by workspace register indirect autoincrement addressing. Regardless of the number of parameters used in any given call, the program counter must be incremented past the whole list, so that the return will be to the next instruction in the calling program. A subroutine call using only one of two passed parameters is shown in the following example:

```
0200           BL     @ANG       Call Subroutine ANG
0202   FLAG1   DATA   >0         Parameter 1 is 0000₁₆
0204   FLAG2   DATA   >1         Parameter 2 is 0001₁₆

0600   ANG     MOV    *11+,3     Move Parameter 1 into R3
0602           C       3,2       Compare Parameter 1 to contents of R2
0604           JEQ    FIRSTEQ    Try next test if equal
0606           INCT   R11        Move Return PC past parameter 2
0608           B      *11        Return
060A   FIRSTEQ
```

The subroutine ANG checks the first parameter against the contents of R2. If an inequality is found, the branch to continue the routine at FIRSTEQ is not taken. The move instruction which loaded parameter 1 into the workspace increments the program counter in R11 by 2 so that register 11 now points to parameter 2. The INCT instruction is required to increment the program counter value in R11 past parameter 2 to point to the next instruction in the calling program.

When parameters are passed in this way using the program counter, good programming practice dictates that they be constants or addresses only and not variables. Variable quantities should be stored in memory external to program code. To 'nest' variable data in program code causes in-line code modification, which would produce code that would be inoperative if stored in ROM.

The example above dealt with the BL subroutine call, though the same technique can be applied to BLWP or XOP calls. These calls store the program counter in workspace register 14, so the indirect address register must be 14 instead of 11.

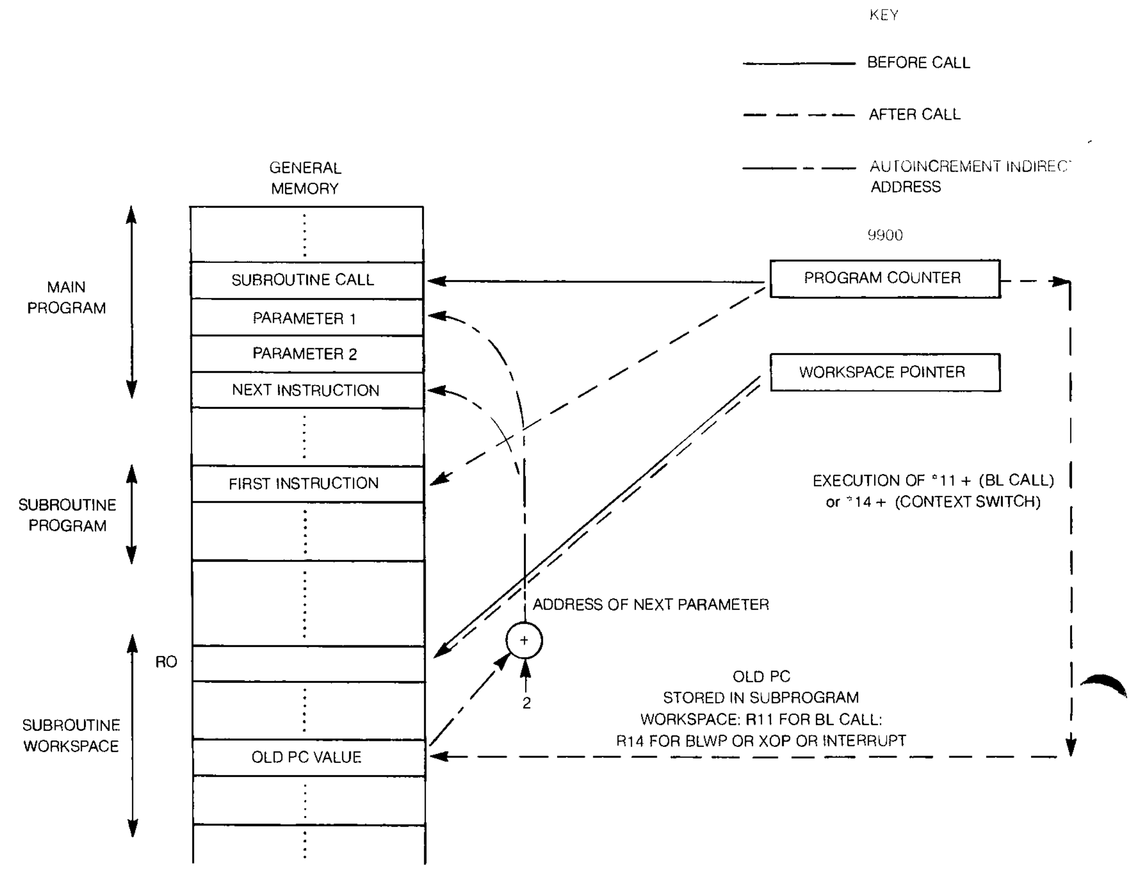


Figure 5-17. Parameter Passing Using Old Program Counter Value.

Another method of parameter passing is used when a context switch occurs. Both BLWP and XOP cause the old contents of the workspace pointer to be stored in the new workspace register 13. By using register 13 in the called procedure, access is gained to parameters in the old workspace as illustrated in *Figure 5-18.* Direct access to old register 0 is provided, but to use other parts of the old workspace, indexed addressing provides the most convenient access to old registers 1 through 15 without changing the old workspace pointer value. For example, to move the contents of old workspace register 2 to new workspace register 5, the following instruction can be used:

```
        MOV     @R2*2(13),R5
```

which causes the address of the operand to be the contents of register 13 plus 4, which is the address of old workspace register 2. Similarly, to move the contents of old workspace register 7 to old workspace register 6:

```
        MOV     @R7*2(13),@R6*2(13)
```

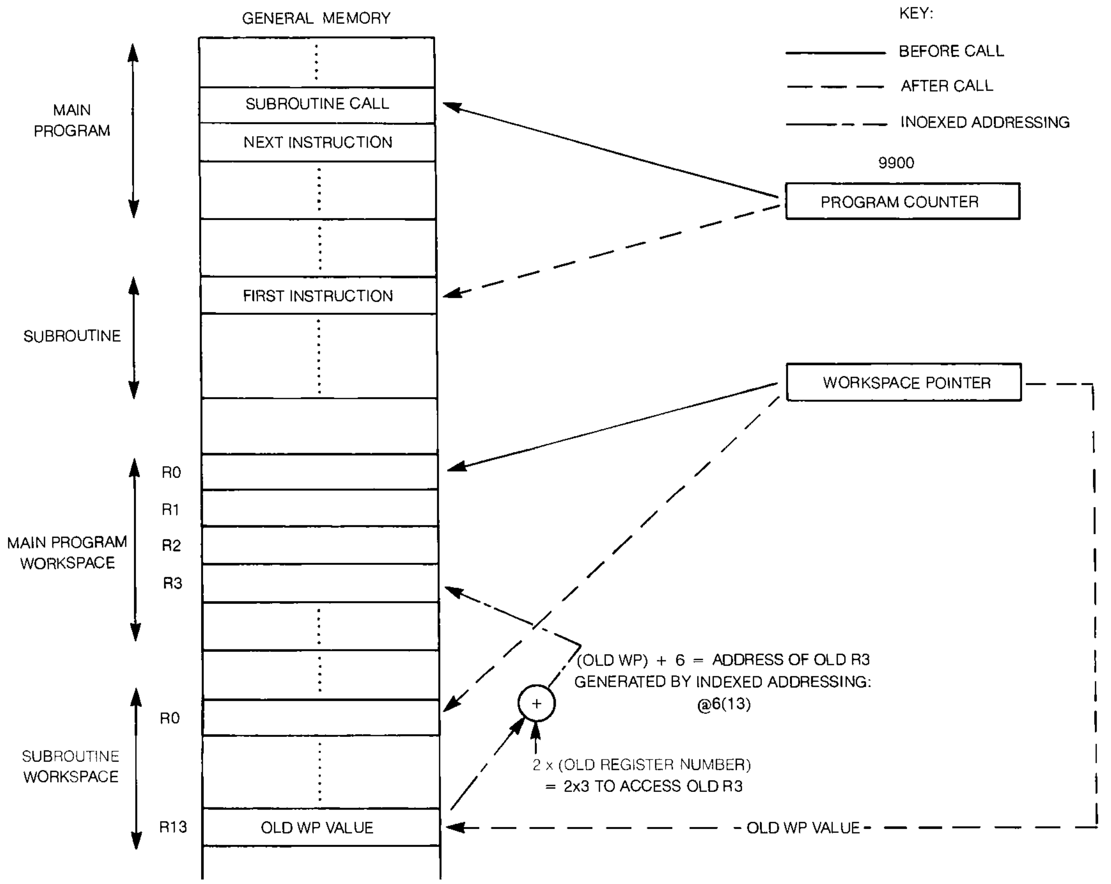


Figure 5-18. Parameter Passing through Old Workspace Pointer

A final type of parameter passing applies only to XOP context switches. The single argument of an XOP call specifies the effective address of a source operand. This form of parameter passing avoids the risk of changing the old PC and WP. The overhead of changing the WP and PC pointers is also avoided to increase execution speed. As an example, if XOP 9 has been defined as FADD by a DXOP directive, the call:

```
        FADD    @LIST
```

causes the address stored at location LIST to be placed in register 11 of the subroutine workspace. Then, workspace register indirect addressing can access the parameter. For example, if in the FADD subroutine it is desired that the parameter be incremented by two, the following instruction would be used.

```
        INCT    *11
```

The use of the parameter through its address in register 11 is straightforward and doesn't interfere with the return linkage. This type of parameter passing has already been illustrated in *Figure 5-16*.

Multiple Level Shared Workspace Subroutines

Since the BL instruction always reloads the workspace register 11, special steps must be taken to insure that the information in register 11 is not overstored. In the case of multiple levels of BL called subroutines, routines which call other routines before returning to the main program, a pushdown stack should be established to save intermediate return linkage. To create a return linkage stack for multiple levels of subroutines which share a workspace, the following procedure is employed:

1) Allocate one workspace register to the stack pointer function.
2) For each subroutine, "push" the contents of workspace register 11 before the next call, and "pop" the stack to restore the register 11 contents after each call is complete.

An example of a stack manipulation code following this procedure to push and pop return linkage is as follows, with register 5 acting as a stack pointer:

```
   :
   :    Subroutine code before next level call
   :
DECT    5          Decrement Stack Pointer          }
MOV     11,*5      Load return PC onto Stack        } Push Operation
BL      @SUBNXT       Call next level of subroutine
MOV     *5+,11        After return, restore current   }
   :                  return address and restore      } Pop Operation
   :                  stack pointer                   }
   :
```

This code allows the current subroutine to call subroutine SUBNXT without destroying the current subroutine's return linkage. The main program employs a standard BL call, and the lowest level routine would not use the stack, since its register 11 would not be replaced with a subsequent BL call. An example of this stack operation procedure with 3 nested subroutines is illustrated in *Figure 5-19.*

Shared Workspace Mapping

Software systems for small computers must efficiently utilize available memory. This section presents an organized technique for sharing workspaces between subroutines to reduce system memory requirements.

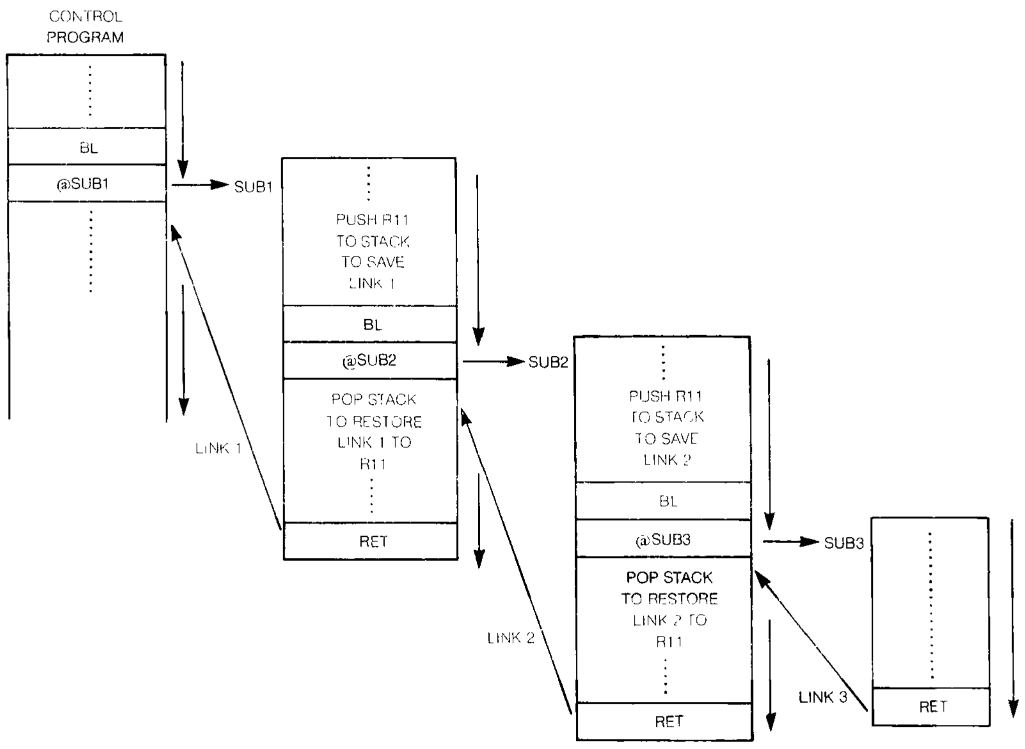

CONTROL PROGRAM
BL
@SUB1
SUB1
PUSH R11 TO STACK TO SAVE LINK 1
BL
@SUB2
POP STACK TO RESTORE LINK 1 TO R11
RET
LINK 1
SUB2
PUSH R11 TO STACK TO SAVE LINK 2
BL
@SUB3
POP STACK TO RESTORE LINK 2 TO R11
RET
LINK 2
SUB3
RET
LINK 3

Figure 5-19. Stack Operations in Nested Subroutines.

The first step in system development is to write a main program and its associated subroutines with totally independent workspaces. Avoidance of shared workspaces at the start can prevent the undesirable aspect of destruction of critical data, including return linkage.

After independent software is written, the programmer begins the process of identifying potential shared workspaces. First, the relationship between called and calling procedures is summarized graphically as shown in *Figure 5-20.* This graph represents the fact that procedure A can call either procedure B or C. Procedure B may call D or E, while E can call D or G, and so on throughout the graph. Having identified routine relationships, *Figure 5-20* can be changed to a form that reflects subroutine levels. All procedures at the same level are called from a higher level and may call routines at a lower level.

Thus, *Figure 5-20* would be changed to the form illustrated in *Figure 5-21.* This information is equivalent to the information contained in *Figure 5-20*, but it clarifies the relationship between procedures. The routines on a particular level can never call another routine at the same or at a higher level. Thus, all routines at the same level can share a common workspace since return linkage will not be overstored by a subsequent call. Therefore, for the example described in *Figures 5-20 and 5-21,* five independent workspaces will suffice for a software system of eight procedures, saving 3 workspaces or 48 words of memory. By employing this simple technique, the software designer can write efficient code with an assurance of the integrety of return linkage.

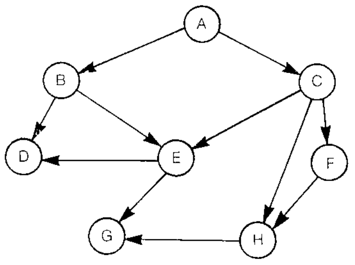


Figure 5-20. *Graphical Representation of Interrelation of Calls*

RE-ENTRANT PROGRAMMING

Re-entrant programming is a technique that allows one set of program code to be executed on multiple data sets concurrently. To be re-entrant, program code must have the following characteristics:

1) All data contained in a re-entrant routine must be common to all procedures which call it, and must be read-only to all using procedures.
2) All data unique to calling routines must be stored and used in a workspace unique to the calling procedure.
3) Re-entrant code must not alter data or instructions within its code during execution.

Re-entrant coding is a general programming technique that has many applications. Device service routines which control the operations of several similar units should be re-entrant. By passing a CRU base address with other unique data to a re-entrant service subprogram, any one of a group of calling procedures can access such a multi-purpose I/O routine, thereby saving system memory requirements. This is a case in which one routine is used for several applications at random time intervals. A re-entrant subroutine is so loosely coupled to its calling procedure that a re-entrant routine can be interrupted during execution, used on different data, and return to complete the original process without losing data integrity. Since re-entrant code is immune to problems with data resulting from interrupts, it finds application in interrupt service routines, commonly called procedures, in a multiprogramming environment such as assemblers or in real-time control applications.

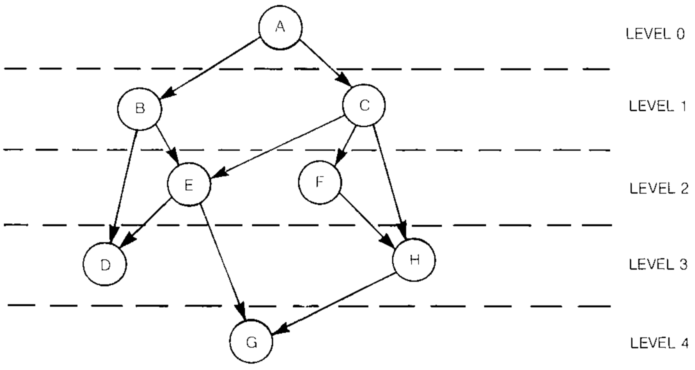


Figure 5-21. Levels of Subroutine Calls

Figure 5-22 illustrates a program flow in which subroutine A must be re-entrant. The alternative to writing subroutine A in re-entrant code is to make two copies of A, one for each time A can be executed concurrently. The re-entrant approach is more efficient in memory usage than is the multiple copy approach. In this program flow, the main program calls its first level subroutine which in turn calls subroutine A as a second level subroutine. During execution of routine A, an interrupt occurs, which in this example the interrupt handler program sequence calls several routines, including routine A. If A employs re-entrant programming, the same words of code can implement routine A for both parts of the program flow.

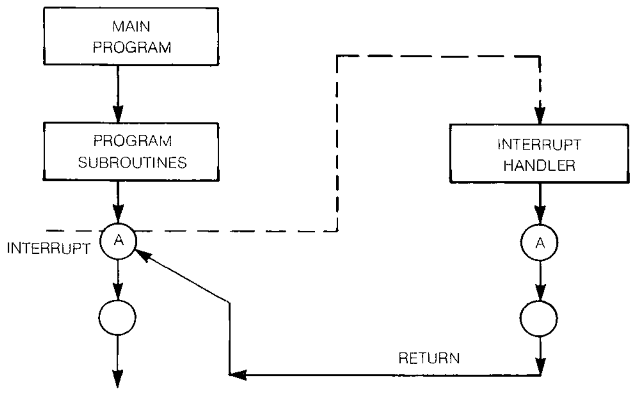


Figure 5-22. Interrupt Requiring Re-entrant Programming

As an example of re-entrant coding, consider the problem of forming a starting and an ending address for a block of data to be operated on by a subroutine. Register 1 is to hold the starting address, register 2 is to hold the ending address, and register 3 is to hold the current data address within the block.

This structure is in a subroutine that must be re-entrant, i.e., one that can be called concurrently by two different program segments as illustrated in *Figure 5-22.* Different program segments that use this subroutine will probably be dealing with blocks at different starting addresses and of different sizes. For example, the main program stream may be operating on a block of 10 words starting at address 1000_{16} while the interrupt handler may have to operate on a 32 word block starting at address 2000_{16}. Forming the starting and ending addresses as follows:

```
BLKLN  EQU  32
       LI   1,>1000      Load R1 with starting address
       LI   2,2*BLKLN    Load R2 with 2x (words in block)
       A    1,2          Form end address for block (1014₁₆)
```

would not result in re-entrant code. This sequence would be correct for the main program stream but is not correct for the interrupt handler stream. The code can be made to work for both streams by not placing the load immediates in the subroutine itself but by placing them in the program stream that calls the subroutine. Then, when either the main program or the interrupt handler gets ready to call the subroutine, the starting address and ending address can be established within the workspace for that environment. The subroutine can then concentrate on performing its manipulations, without being concerned with the address initialization process. Suppose that this addressing scheme is used in a subroutine that clears a block of memory words. *Figure 5-23* shows the re-entrant and non-re-entrant forms of this subroutine. The re-entrant form can be used in the situation of *Figure 5-22* since execution depends on the workspace being used. The subroutine can be executing with the registers 1 through 3 of the main program when an interrupt occurs. The interrupt handler uses a different workspace so when it calls the CLEAR subroutine, new starting and ending addresses are used, without affecting where the subroutine was in the main program execution. Then, when the subroutine and interrupt handler have been executed and the context is switched back to the main program, the subroutine will continue executing with the values in registers 2 and 3 of the main program environment.

This is not true of the non-re-entrant code. By moving the contents of the interrupt register 1 to the FIN location, the number of blocks to be cleared by the main program execution CLEAR subroutine could be changed, if the interrupt occurs after the MOV 1, @FIN instruction. Because the value FIN is unique for each calling program or computed in the subroutine, the code may not properly be re-entered. That is, it should not be used when an interrupting procedure may execute the same code. The re-entrant version could be used by any number of interrupting procedures without affecting execution results in either the main program or the interrupting program environments. Entrance to the routine would be performed by executing a BL @ CLRLUP.

Re-Entrant Coding

```
        MOV     1,3         Set R3 at first address of block

        A       1,2         Compute end address and place in R2

COM     C       3,2         Address past end?

        JH      PRET        If so, return
        CLR     *3+         Else, clear word and go to next address
        JMP     COM         Jump to continue clearing
PRET    B       *11         Return
```

Non-Re-entrant Coding

```
CLRLUP  MOV     1,3         Set R3 at first address of block

        MOV     1,@FIN      Set up first address in FIN

        A       2,@FIN      Compute final address and store at FIN

COM     C       3,@FIN      Address past end?

        JH      PRET        If so, return
        CLR     *3+         Else, clear word and go to next address
        JMP     COM         Jump to continue clearing
PRET    B       *11         Return
```

Figure 5-23. Subroutine Example of Re-entrant Coding.

PROGRAMMING TASKS

The programming techniques of workspace and subroutine usage, program loops, macros, and data representation must be applied to the development of programs and subprograms to perform the basic system functions of state initialization, pattern recognition, arithmetic, and input/output. Each of these system functions represents a programming task that involves programming structures peculiar to each function. This section discusses the basic requirements of the software for each function and presents some of the programming approaches that are used to meet those requirements.

INITIALIZATION

When the system is first turned on, the first few instructions encountered must initialize the state of the system to a desired predetermined starting state. The system initialization procedure is usually started by the RESET or LOAD functions. Similarly, as the system enters a subprogram or a new program sequence, the state of certain memory locations must be initialized to a desired starting state. Further, in developing the software, the transfer vectors and other program constants must be initialized by the assembly language software. The assembly language directives available for this purpose include the equate (EQU) and the data (DATA) directives. The application of these directives to the problem of initializing the reserved memory locations and program constants are covered in detail under the assembler directive discussion in Chapter 7.

Usually the first part of any program is the initialization of the system. The LWPI instruction is used to initialize the workspace pointer (WP register) to define the location of the 16 workspace registers. If the workspace of a program sequence is to be located at a starting address of 400_{16}, the following instruction will initialize the workspace pointer to that value:

```
        LWPI   >0400
```

Under the interrupts discussion the use of the LIMI (load interrupt mask immediate) instruction to establish which interrupts would be responded to was covered. For example, if the programmer wants to disable all interrupts above level 7 for a program segment, the following instruction must be used at the first of the segment:

```
        LIMI   7
```

Similarly, the load immediate (LI) instruction is used to initialize values in workspace registers. The LI can be followed by MOV instructions to further initialize other memory locations. As an example, to initialize register 3 and memory location TEST to the value $00FF_{16}$, the following instructions can be used:

```
        LI      3, >00FF
        MOV     3, @TEST
```

▶5 In some initialization sequences several registers have to be initialized to the same value, such as zero. For example, if 10 consecutive memory words starting at location 1000_{16} are to be cleared (zeroed) then a program loop is suggested. One possible implementation of this initialization task would be:

```
          LI      2,>1000     Set R2 to the starting address 1000₁₆
          LI      3,>100A     Set R3 to the address past the last data location to
                              be cleared
LOOP      CLR     *2+         Clear data, increment the address by two
          C       2,3         Is address past the 10th data location

          JNE     LOOP        If not jump to LOOP to continue clears
                              else go the next sequence of instructions
           •
           •
           •
           •
```

In this data initialization program segment, like most program segments, registers must be initialized to establish program limits, addresses, and other conditions. In this sequence register 2 was initialized to the starting data address and register 3 was initialized to indicate the first word address after the 10th data word to be cleared. Had this loop been implemented with a loop counter, the register acting as a loop counter would have been initialized to 10.

Generally, most program initialization tasks can be handled by using a combination of the techniques presented in this section. The immediate load instructions are the most commonly used operations in performing the initialization operation, followed by the use of assembler initialization directives to establish vectors and other data constant initialization.

Masking and Testing

In many cases only certain portions of a word are of interest. The program segment may be examining or modifying a single bit or a group of bits. The bits that are not involved in the operation must be masked off so that they will not affect status bits and thus affect program decisions. There are several ways of approaching this masking and single bit testing problem.

If a single I/O bit is to be examined or modified, the simplest approach is to use the CRU single bit instruction SBO, SBZ, or TB to perform the desired operation. This is possible only if the hardware has been set up to address the desired bit as a CRU bit. If the bit is not accessible through CRU addressing, one of the selective masking instructions must be used. The set to ones or zeroes instructions (SOC, SOCB, SZC, and SZCB) can be used to selectively set or clear bits. The compare ones or zeroes corresponding instructions (COC and CZC) can be used to test selected bits. Of course these instructions can be used to test or change single or multiple bits. An alternative single bit approach is to use the circulate instruction (SRC) to get the desired bit into the carry status bit for examination or changing.

To see how these non-CRU masking instructions are used, consider the task of examining the value of bit 12 of the data of workspace register 1. The mask is contained in location MASK which will contain all zeroes in all bits except for bit 12 which will contain a one. Thus, location MASK will contain 0008_{16}. Then, the instruction:

```
        CZC     @MASK, 1
```

will set the equal status bit if bit 12 of R1 contains a zero. The instruction:

```
        COC     @MASK, 1
```

will set the equal status bit if bit 12 of R1 contains a one. In these cases, the JEQ or JNE instructions can be used to test the equal status bit after the comparison.

Alternatively, the instruction:

```
        SRC     1,4
```

will cause bit 12 to be in the carry flip flop. However, this instruction will change the contents of R1 by moving all bits to the right 4 positions. The JC or JNC instructions are used to test the bit value in the carry status bit.

To selectively set bit 12 of R1, any of the following instructions could be used:

```
        ORI     1,>0008
        SOC     @MASK,1
```

To selectively clear bit 12 of R1, either or the following instructions could be used:

```
        ANDI    1,>FFF7
```

or:

```
        SZC     @MASK,1
```

If groups of bits are to be changed or examined, the above techniques can be used if all bits are to be ones or zeroes. For example, if bit 13 of register 2 is to be tested, the following instructions would jump to point P1 in the program if bit 13 of R2 is one:

```
        ANDI  2,>0004        Zero all bits but bit 13 of R2; compare to 0

        JNE   P1             If EQ=0, bit 13 was one and jump to point P1
```

A more complicated test would be to check bits 13 and 15 of R3. A jump to P2 is to be made if both of these bits are one. The following instructions would accomplish this test and program decision:

▶5

```
H5        DATA     5
          •
          •
          •
          COC      @H5,3       Compare to 5 to see if both bits are one.
          JEQ      P2          If they are, jump to point P2.
```

Thus, a combination of masks (ANDI) compares, and conditional jumps can be used to examine all features of system words and react appropriately.

If a group of bits is to be examined or modified arithmetically, a slightly different approach may be used. If for example the least four bits of R1 are to be compared to 8, one approach would be to provide a copy of the R1 contents in R2. Then the first 12 bits of R2 are zeroed with:

```
        ANDI   2,>000F
```

or with: SZC @MASK1,2 where the contents of location MASK1 are $FFF0_{16}$. Then, the contents of R2 can be compared to 8. The entire sequence would then be:

```
        MOV   1,2
        ANDI  2,>F
        CI    2,8
        JLT   P1
          •
          •   Sequence of instructions
          •     to handle case where least
          •     four bits of R1 (and R2) are
          •     greater than or equal to 8
          •
P1        •   Sequence of instructions
          •     to handle case where least
          •     four bits of R1 (and R2) are
                less than 8
```

This technique is useful in Decimal to Binary or Binary to Decimal number conversion and in implementing BCD (Binary Coded Decimal) arithmetic.

Of the techniques that can be used in masking and testing, the ANDI, ORI, SOC, and SZC instructions change the word they operate on. The Compare techniques, CZC and COC, do not affect the words being operated on but they do affect the status bits. Often, when part of a word is modified (such as portion of the word is zeroed by an ANDI instruction) the word must later be reassembled after all bit group operations have been completed. The programmer should see that such operations are performed on copies of the word so that further masking operations use the original word. As an example, if a word is to be broken down into four bit groups (to implement BCD arithmetic), at least four copies of the word are required or four accumulators must be used. If register 1 contains the master copy, and registers 2 through 5 contain the four bit groups, the following sequence of instructions would generate the desired four bit groupings in the four accumulators from the master copy in register 1:

```
MOV   @MASTER,1  Move word to be separated into R1
MOV   1,2        Move a copy of the word into
MOV   1,3        accumulators R2 through R5
MOV   1,4
MOV   1,5
ANDI  2,>000F    Mask all but least four bits in R2
ANDI  3,>00F0    Mask all but next four bits in R3
ANDI  4,>0F00    Mask all but next four bits in R4
ANDI  5,> F000   Mask all but most significant four bits in R5
```

With this program sequence, the original word can be broken into bit groups for further testing and modification. R1 still contains the original word for reference and further manipulation. However, by using ANDI mask instructions, several memory words are required to hold intermediate results. This would not have been necessary if compare (selective bit) instructions had been used. The specific application usually dictates which approach is to be used.

Arithmetic Operations

Basic arithmetic can be performed with addition and subtraction, though certain operations such as multi-word arithmetic require the use of shift instructions and conditional branch instructions such as the jump on carry or jump on greater than.

Multi-Precision Arithmetic

The 9900 arithmetic instructions perform mathematical functions on 16 bit words. For applications that require a greater numerical accuracy or a larger number (the 16 bit word can hold a magnitude number from 0 to 65,535), multiple word numbers must be used. The basic arithmetic instructions must then be used in such a way as to implement the desired mathematical functions on these multiple word numbers. This section deals with techniques for treating several words as a single binary value, that is, extended precision arithmetic.

A 16 bit two's complement word can represent a signed value in the range −32,768 to +32,767. The negate (NEG) and absolute value (ABS) instructions provide fast conversion between positive and negative 16 bit words. For sign conversion on binary values represented by multiple words, special conversion techniques are required. The process for converting a three word positive value to its negative or two's complement value is shown in *Figure 5-24.* The three word number is stored in registers 0, 1, and 2 of the workspace. The complementing procedure is to form the one's complement of the three word number using the invert (INV) instruction and then to add 1 to the result. The JNC instructions in the program check to see if a carry is to be propagated from a less significant word to a more significant word in the process of adding one to the three word number. If carries occur, the addition is handled by the increment (INC) instruction. Conversion of a number to its absolute value is accomplished by checking the sign bit (most significant bit) and executing the negate routine (COMP) on negative values.

Memory Structure

Register 0	Register 1	Register 2	
A_0	A_1	A_2	= A (48 bit number)

Procedure:

1) Form 1's complement of A, using the invert (INV) instruction.
2) Convert the 1's complement of A to the two's complement of A by adding 1 to the 1's complement of A.

2's complement of A = 1's complement of A + 1

Program:

```
COMP    INV    0        Complement contents of R0
        INV    1        Complement contents of R1
        NEG    2        Negate contents of R2
        JNC    EXIT     If no carry operation is complete, return
        INC    1        If carry, add one to contents of R1
        JNC    EXIT     If no carry operation is complete, return
        INC    0        If carry, add one to contents of R0
EXIT    RT              Return
```

***Figure 5-24.** Process to form the Negative of $A(-A)$.*

The process of adding or subtracting two multi-word numbers is to perform the operation on the least significant words, then on the next most significant words with the previous carry or borrow, and so on until the complete result is formed. Subtraction could be performed by first using the negate procedure of *Figure 5-24* on the value to be subtracted and then adding this two's complement result to the other number.

Multiple word multiplication can be handled by using the 9900 multiply (MPY) instruction to provide 32 bit partial products and then adding all partial products to achieve the final desired product. The procedure is illustrated in *Figure 5-25* for multiplication of one 32 bit number by a second 32 bit number. The multiplication of a 16 bit number by a second 16 bit number is performed by the MPY instruction. Thus, four applications of the MPY would form the required four partial products. Then, by adding these products in the correct positions, the 64 bit product is formed. The basic memory structure used by the example in *Figure 5-25* can be understood by looking at the operation of the MPY instruction. The accumulator or destination operand must be a workspace register. Then, the product is stored in two successive workspace registers, the most significant 16 bits in the destination workspace register and the least significant 16 bits in the next workspace register. The source operand which specifies the multiplier may be specified with any addressing mode, though the example of *Figure 5-25* uses register addressing for this operand. Thus, the instruction:

```
        MPY    1,8
```

multiples the contents of register 1 by the contents of register 8 and places the 32 bit product in registers 8 and 9. While the multiplier register 1 contents are unchanged, the multiplicand register 8 contents are changed to the most significant part of the product.

Thus, there must be several copies of each multiplicand to be able to form several partial products. In 32 bit by 32 bit multiplication, there are two multipliers (B_0 in register 0 and B_1 in register 1) and two multiplicands. Since each multiplicand is involved in two partial products, there must be two copies of each multiplicand. In *Figure 5-25* the copies of the A_0 multiplicand are saved in registers 4 and 6 and the copies of the A_1 multiplicand are saved in registers 2 and 8. Then, the following four MPY instructions form the four required partial products:

```
MULT MPY      1,8    Form the A1xB1 product in R8 and R9
     MPY      1,4    Form the B1xA0 product in R4 and R5
     MPY      0,2    Form the B0xA1 product in R2 and R3
     MPY      0,6    Form the B0xA0 product in R6 and R7
```

Which can be followed by the additions to form the complete 64 bit product in registers 6 through 9:

```
     A        3,5    Add two of three 16 bit groups in positions 2^16 to 2^31
     JNC      P0     If no carry, add in R8 contents
     INC      7      If carry, add one to R7 accumulator
P0   A        5,8    Finish adding 2^16 to 2^31 bits
     JNC      P1     If no carry, procede to next position adds
     INC      7      If carry, add one to R7 accumulator
P1   A        2,4    Add part of 2^32 to 2^47 bits in R2 and R4
     JNC      P2     If no carry, procede to rest of addition
     INC      6      If carry, add 1 to R6 accumulator
P2   A        4,7    Finish adding 2^32 to 2^47 bits
     JNC      FIN    If no carry, operation is complete
     INC      6      If carry, add one to R6 accumulator
FIN  RT     return
```

The process illustrated by *Figure 5-25* is for multiplication of two 32 bit magnitude numbers. Multiplication of negative numbers can be handled with the same program by converting all numbers to their absolute value, saving the sign. Then, after the magnitude multiplication is complete, the sign of the product is the exclusive OR of the multiplier and multiplicand sign bits. If desired, the product can be complemented or negated and stored in two's complement form.

Basis of Procedure:

$A \times B = (A_0 x 2^{16} + A_1) \times (B_0 x 2^{16} + B_1) =$

$A \times B = A_0 x B_0 x 2^{32} + (A_1 x B_0 + A_0 x B_1) \times 2^{16} + A_1 x B_1$

Where multiplying by 2^n implies shifting the number n positions to the left with respect to a number multiplied by 2^0 or 1.

Memory Structure:

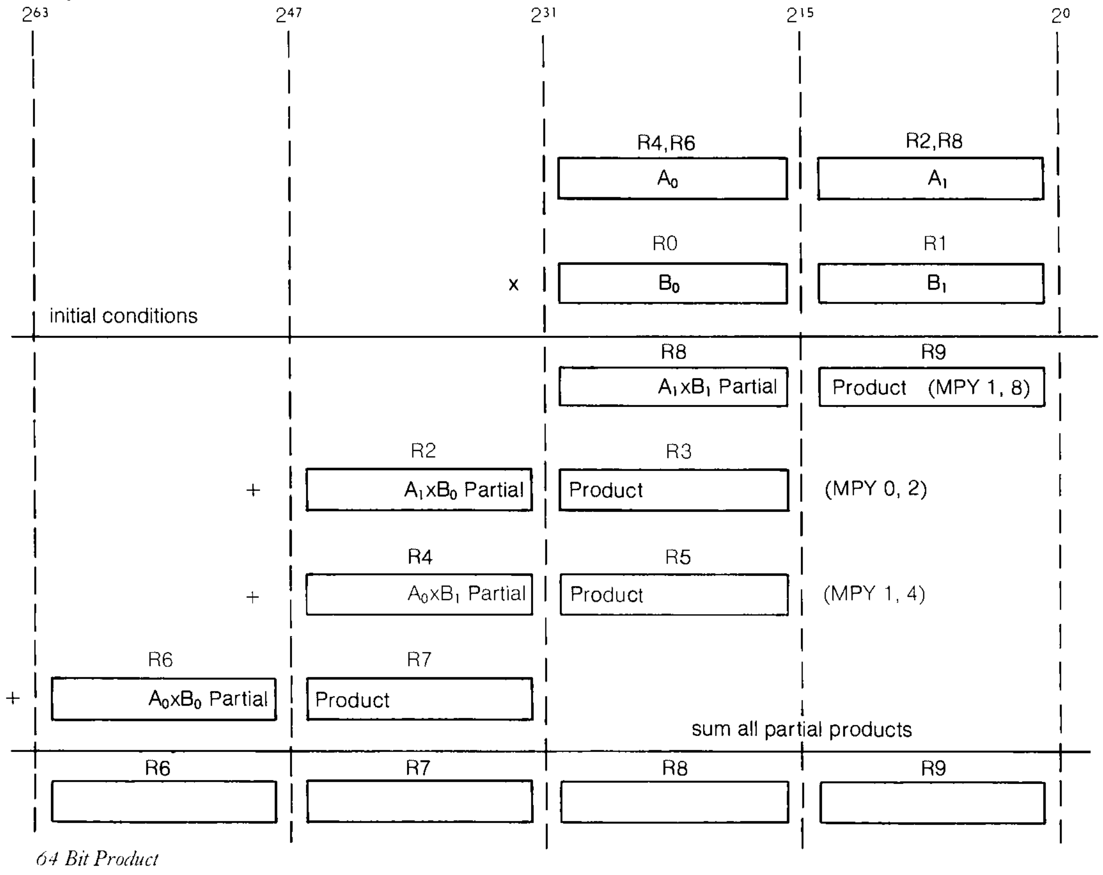


Figure 5-25. Multiple Precision Multiplication

Floating Point Arithmetic

If the system requires the ability to represent fractional numerical quantities instead of integer numbers, a method must be defined that will provide for the location of the radix point of such numbers. Just as the decimal point of 75.39 defines a quantity:

$$7x10^1 + 5x10^0 + 3x10^{-1} + 9x10^{-2}$$

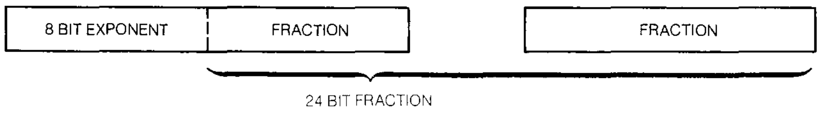
the binary point in 101.01 defines a quantity:

$$1x2^2 + 0x2^1 + 1x2^0 + 0x2^{-1} + 1x2^{-2}$$

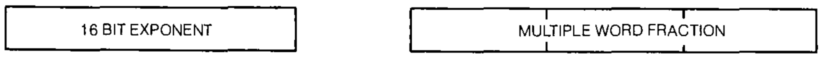
Although a group of bits can be configured in many ways to define a floating point number, most floating point representations share the following characteristics:

1) Floating point numbers are represented as a fraction and an exponent (mantissa and characteristic).
2) The fraction is by convention normalized to lie in the range $\frac{1}{2} \leq F < 1$, e.g., the binary point lies to the left of the first one bit.
3) The exponent defines the power of 2 by which the fraction is multiplied to evaluate the floating point number.

Possible floating point representatives include:



and:



When addition or subtraction of two floating point numbers is performed, the following operations must be performed:

1) Equalize exponents; increment the exponent of the smaller quantity until it is the same as the larger exponent. With each exponent increment, shift the corresponding fraction to the right with zero fill from the left.

2) Add or subtract the fractions as required. If a carry results from an addition, the sum fraction is shifted right, shifting the carry into the fraction, and the exponent is incremented. If the difference resulting from the subtraction is not normalized (zero in first bit position) the fraction must be shifted left until a one is in the leftmost position. With each left shift, the resultant exponent is decremented.

Floating point multiplication can be performed by multiplying the fractions and adding the exponents. Similarly, floating point division can be performed by dividing the fractions and subtracting the exponents. After such an operation the fraction must be normalized and the exponents must be checked for overflow or underflow. Signed numbers can be handled in the same way that the multiplication of signed integers is handled.

INPUT/OUTPUT

The most fundamental and necessary instructions of a processor are its input and output instructions and techniques. Without input and output, the system would not be able to control or communicate with the external world, and as a result would be of no use. There are two general ways of implementing input and output operations. One obvious approach is to use special input and output instructions that are interpreted by the hardware to apply to the input and output devices. The 9900 instructions that provide this capability are the CRU or communications register unit instructions (SBO, SBZ, TB, LDCR, and STCR) and the input and output hardware that respond to these instructions make up the communications register unit. Another approach to inputting and outputting information is to simply treat the input and output devices as one of the system memory locations, in which case any of the 9900 instructions can be used in accessing these locations. This approach is called memory mapped input/output, since the devices are assigned a portion of the available memory addresses, and the hardware must decode the appropriate address to activate a given device. Each of these approaches has its advantages and disadvantages, and the programmer must be aware of these trade-offs in order to provide an optimum approach to system input/output.

Memory Mapped Input/Output

The principle advantages of using memory mapped input/output are:

1) The full instruction set is available for manipulating the data in the input/output device.
2) The hardware is straightforward, since address decoding and device timing signals are required for RAM and ROM memory anyway and these can be simply extended to handle the I/O subsystem as well.
3) Transfers of information are made 8 or 16 bits at a time, offering a high bit rate transfer.

The disadvantages of memory mapped I/O are:

1) Since some of the 'memory' locations are being used by input or output devices, less memory is available for instructions and general data storage.
2) Bit transfers must be made 8 or 16 bits at a time. This is wasteful if a given device can handle a single or a few bits at a time.
3) It is a more expensive technique in terms of pinouts, board space, and layout time.
4) The hardware interface must accomodate the full width memory bus.

The most commonly used instruction in memory mapped I/O operation is the MOV instruction to effect data transfers. However, it is quite possible to set up an input output subsystem as general purpose storage or as a workspace and perform shifts, additions, multiplications, logical operations, and so on, on the data contained in the I/O subsystem.

Generally, if I/O transfers are to be made 8 or 16 bits at a time and if the system is not memory bound (memory is needed for program and system data), memory mapped I/O is often used. Certainly, if performing arithmetic, logic, or other instructions directly on input/output data is required or advantageous, memory mapped I/O must be used. If single or multiple bit transfers are all that is required, and transfer rate is not critical, then memory mapped I/O has no advantage over CRU I/O. CRU I/O hardware is normally simpler and less expensive.

CRU Input/Output

The CRU instructions provide for single bit transfers with the SBO (set bit to one), SBZ (set bit to zero), and TB (test bit) instructions. Multiple bit transfers with the bits transferred one at a time are possible using the LDCR (load communications register) and STCR (store communications register) instructions. The advantages of the CRU instruction approach to I/O are:

1) Any number of bits (up to 16) can be transferred with the appropriate CRU instruction. Thus, the designer can set up the data transfer to exactly meet the requirements of the subsystem being serviced. This is especially useful in control situations where single sense bits are to be examined and single on-off output control signals are needed.
2) No memory locations are used by the subsystem. The CRU instructions can access 4096 input and 4096 output bits (which is equivalent to 256 data words) in addition to the 65,536 memory bytes.

The disadvantages of the CRU I/O are:

1) Only data transfers are provided. Arithmetic, logical or other operations must be performed on the data after it has been moved to one of the general data storage locations in RAM.
2) The hardware must include the capability to decode and implement the CRU transfers; however, the added IC complexity is more than offset by reduced package size.
3) Single bit transfer speed may be too slow for some applications.

Input/Output Methods

There are three ways that an input/output transfer can be handled or initiated. The processor can be interrupted, causing the program to jump to a subroutine that handles the input/output task. The program can encounter an instruction to transfer data from an input location or to an output location, for the purpose of displaying results, actuating control elements, or inputting system status. The processor can be bypassed entirely and the data transferred directly to or from system memory, using direct memory access.

Interrupt Driver Input/Output

If the timing of input/output transfer is to be controlled by an external system, then the interrupt driven I/O method must be used. This approach is used in inputting data when the time of input is random. The external system inputs the data and signals the processor with an interrupt to indicate that data is in one of the 9900 system input registers. The 9900 responds by performing a context switch to a subprogram that will process the data in that register. The interrupt driven approach may also be used in outputting data when the processor needs to know when the data has been ~~taken by~~ an external system. Once the external system takes the data, it can signal the processor with an interrupt signal. The processor responds by performing a context switch to a subprogram that will then send more data to that output location.

The interrupt driven I/O procedure provides a mechanism of implementing a communications sequence known as handshaking. This communications protocol is illustrated in *Figure 5-26.* In the input mode, the data-present signal latches the 9900 system input register and serves as the interrupt signal. If desired the processor's reading of the contents of that register can be used to generate the data-taken signal. Upon receiving the data taken signal, the external system can then send more data to the 9900 system. In the output mode, the register write operation that sends data to the output register in the 9900 system can be used as a data-present signal to the external system. When the external system takes this data, it can use an interrupt signal to notify the 9900 that the register is ready to receive more data.

The interrupt driven approach has the main advantage of providing a means of setting up a handshaking communications with another system. It can handle data communications that occur at random or unpredictable times. The main disadvantage of this type of I/O is that it does involve the processor, slowing down its work on main system programs and subprograms. Further, since a context switch is involved in responding to an interrupt, such an approach may require more memory than one of the other two approaches.

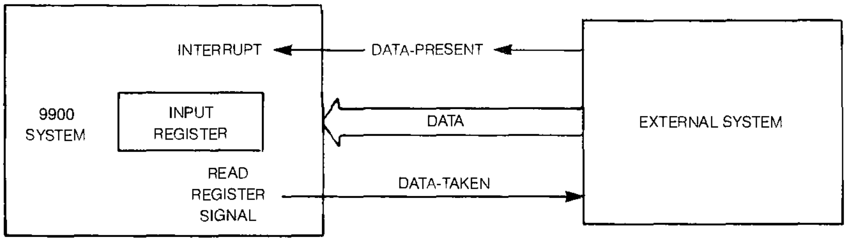


Figure 5-26a. Handshaking Input Transfer.

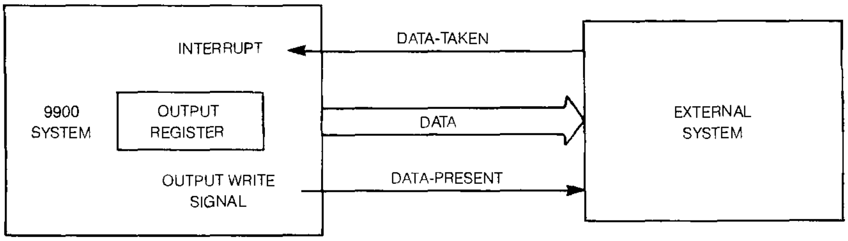


Figure 5-26b. Handshaking Output Transfer.

Programmed Input/Output

The simplest method of I/O is the programmed I/O. The times and conditions under which inputs and outputs are to occur are controlled by the system program. For example, the processor may update a display memory whenever the display is to be changed. The program determines the new information to be displayed and then outputs this information to the display storage locations. Similarly, the program may require information about the status of a subsystem in order to determine subsequent operations. Such status may be the condition of threshold detectors, the status of serial data transmission or reception, or some other system condition. When the program encounters a point at which it needs to check such status words, it simply inputs the desired word. Of course, as in all I/O methods, the input or output operations can be handled by either CRU instructions or through standard MOV or other instructions in memory mapped I/O devices.

The programmed input/output method is a high speed, low hardware cost approach to input/output, since no subroutine overhead is involved. However, it does not handle the random input situation very well, unless the program is devoted solely to waiting for the next signal input to occur. Even then, the program may not check for input occurrence and perform the desired input instruction before the external system sends new data, destroying the old data.