2.0 INSTRUCTION SET

2.1 9900 CPU Overview

2.1.1 Introduction

The 9900 CPU is not the only 16-bit microprocessor, but it ranks as one of the most powerful ones. The architecture of the 9900 is unlike that of most other microprocessors (8 or 16 bits). It has an architecture close to that of a minicomputer. In fact, the 9900 instruction set is identical to that of the Texas Instruments 990 minicomputer. This section provides an overview of the 9900 CPU from a programming viewpoint. Combined with the individual instruction descriptions in section 2.2 you have all the tools to begin writing code.

As already mentioned, the 9900 CPU is a 16-bit computer. Its architecture is vastly different from the simpler 8-bit computers. One difference is that the working registers are contained in memory. The only registers within the processor itself are: the program counter, status register, and a pointer to the working registers in memory. The overall processor architecture is shown in Figure 2.1. The program counter (PC) contains the address of the current instruction. The workspace pointer (WP) is a 16-bit register which holds the address of the first working register in memory. The sixteen general registers R0-R16, called workspace registers, are contained in the sixteen sequential memory locations addressed by the WP.

For easy reference, the entire 9900 instruction set is described in detail in section 2.2 and summarized at the end of that section.

Computations in the 9900 CPU are performed between the registers, between the registers and memory, or between two memory locations. The memory of the 9900 is addressed by byte or word. The processor always references a word because the least significant address bit is not available as an external pin on the processor. Internally, however, you can address either words (two consecutive bytes, starting with an even byte), or bytes. All instructions are stored as one, two, or three consecutive words. The addressing modes available in the 9900 CPU are:

(1) immediate - The operand is contained in the word following the instruction. For example,

```
LI   R1,>1234        ; load R1 with 1234 (hex)
```

will load register R1 with the value 1234 hexadecimal.

will load R2 with the memory location addressed by the contents of R1 plus 10.

(6) relative - Relative addressing is used to obtain the destination address for most of the 9900's jump instructions. To obtain the final destination address, the second byte of the instruction is multiplied by two and added to the address of the next sequential instruction. The addition is performed using two's complement arithmetic. This allows the programmer to transfer control to an address within the range of -254 to +256 of the present instruction. Since all instructions are stored as words (two bytes), you can transfer control to a word within the range of -127 to +128 of the present instruction. An example of relative addressing is:

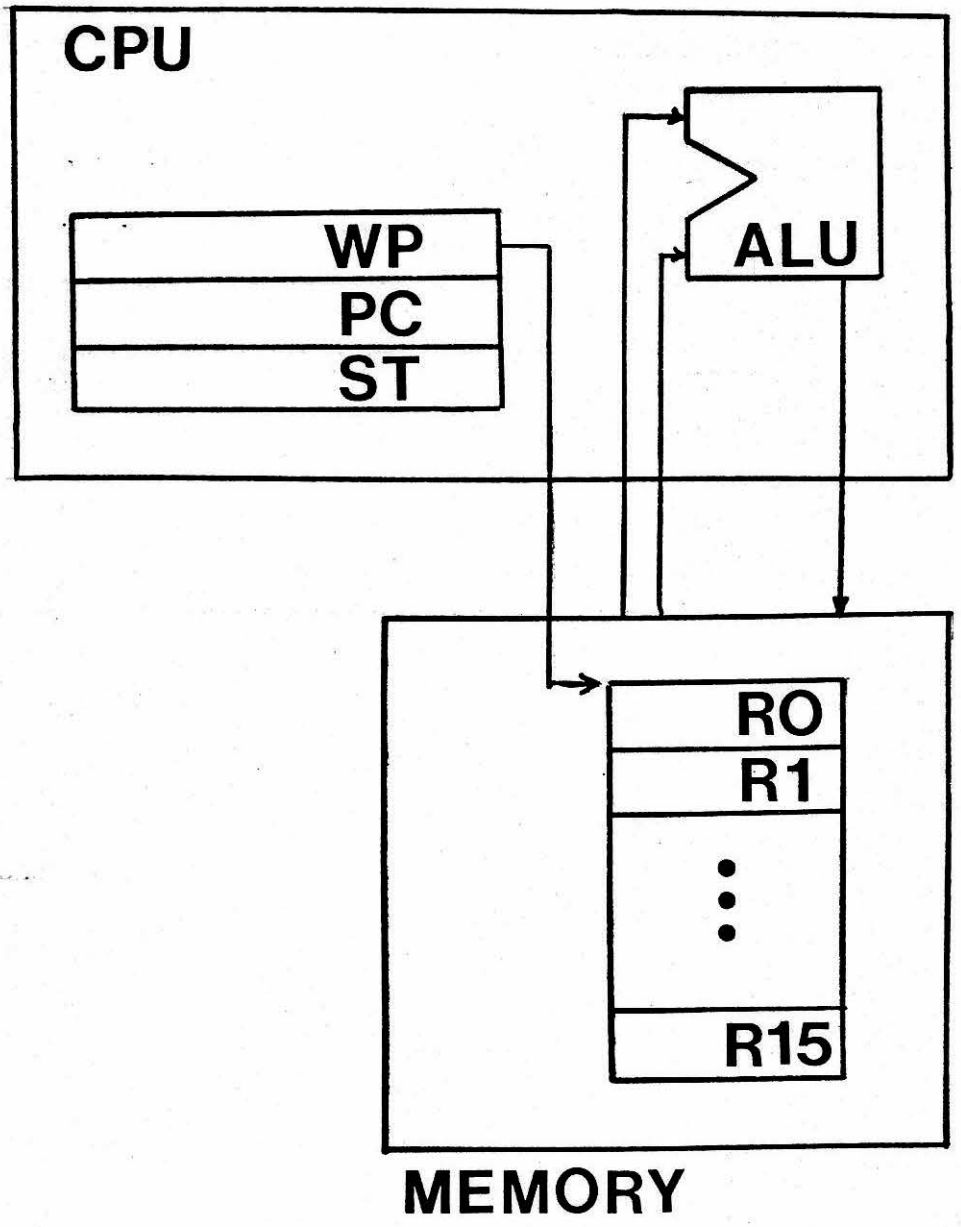
```
JMP  +10
```

This instruction will transfer control to the address of the next sequential instruction plus 20 (10*2). If the jump were at >1200, this would transfer control to address >1216.

All of the op-codes are one word long. If immediate, indirect, or indexed addressing is used, the constant is stored in the word(s) following the op-code. The constant for the source operand is stored in the first word following the op-code and the constant for the destination operand is stored in the next available word. This means that 9900 instructions are one to three words long, or two to six bytes. The following six byter will transfer the contents of variable VAR1 to VAR2:

```
MOV  @VAR1,@VAR2     ; VAR2=VAR1
```

Figure 2.1 Processor Architecture



2.1.2 Subroutine Linkage

Unlike many machines, the 9900 does not use a stack to hold subroutine return addresses. Instead, the processor saves the return address in workspace register R11. For example, the following instruction will save the address of BACK in R11 and will transfer control to ROUT:

```
      BL    @ROUT           ; call ROUT
BACK  .
      .
      .
      .
```

To return from the subroutine, all you need to do is jump to the contents of R11 (B *R11).

If one subroutine must call upon another, it must save the contents of R11 prior to that call, since the new return address will be placed in R11 - thus destroying the old return address. There are several different ways to approach this problem. The first, and simplest, method is to save the return address in one of the general registers. For example, if ROUT is called as indicated above and must then call ROUT2, the sequence below can be used:

```
      MOV   R11,R1          ; save return address
      BL    @ROUT2          ; call next subroutine
      .
      .
      .
      B     *R1             ; exit
```

If you have only two or three levels of subroutine, this may be the most efficient approach. However, in larger systems there are usually too many levels of subroutines to store all the return addresses in the registers. In that case, the return address can be saved in RAM. One way to do that is:

```
      MOV   R11,@TEMP       ; save return
```

To exit the subroutine, the following two instructions are used:

```
      MOV   @TEMP,R11       ; get return
      B     *R11            ; exit
```

The major disadvantage of this technique is that four words of instruction memory are required for the exit sequence, not to mention the word used to hold the return address. If the program is always to be run in RAM (never put in PROM/ROM storage), an alternate entry/exit sequence is:

```
      MOV  R11,@EX+2        ; save return in exit branch
      .
      .
      .
EX    B    @0               ; exit
```

This time we saved the return address in the second word of the branch instruction, thus eliminating the move. The disadvantage here is that the program modifies itself. This means that the program can never be placed in ROM. Most microprocessor programs are eventually stored in ROM so this sequence couldn't be used. However, if you are writing a quick and dirty routine, to be run only from RAM, this approach works well.

There is yet another way to save the return address. We can put it on a stack. What stack, you say? Because of the flexible modes of addressing, creation of a software stack is a very simple task. During the initial start of the program, we load one of the general registers, let's say R15, with the address of the first location of the stack. Then, an entry can be placed on the stack with the following move:

```
      MOV  R11,*R15+        ; stack R11
```

The stack pointer is incremented after the store, so the stack builds up instead of down as in other micros. To retrieve an entry from the stack, the following instructions are used:

```
      DECT R15              ; R15=R15-2
      MOV  *R15,R11         ; get the top entry
```

The stack could also be used to save some of the other general registers that would be used by the subroutine.

If a subroutine requires a number of registers, another method of call is the Branch and Link Workspace Pointer (BLWP). This instruction is also a subroutine call, but before performing the call it resets the workspace pointer. This means that the subroutine has a whole new set of registers to work with - without having to store the old ones! This instruction is very valuable, but should be used with discretion because it requires more memory. More memory for the call and sixteen words more memory for the new set of registers.

2.1.3 Passing Parameters

There are many different methods for passing data to subroutines - in the registers, following the subroutine

call, or addresses following the subroutine call. Since the return address of the routine is already in one of the general registers (R11), passing parameters or their addresses following the call is especially useful with the 9900. For example, consider the floating point subroutines called FMUL and FADD which are the multiply and add floating point routines, respectively. Each one requires three parameters, the address of which could be placed after the subroutine call. If this approach is used with the 9900, the following sequence is used to calculate X1=X2*X3+X4:

```
BL    @FMUL           ; TMP=X2*X3
DATA  X2
DATA  X3
DATA  TMP
BL    @FADD           ; X1=TMP+X4
DATA  TMP
DATA  X4
DATA  X1
```

Before we can manipulate the parameters, it may be necessary to place them in the registers. This is easily accomplished by the following:

```
MOV   *R11+,R1        ; R1=address of param 1
MOV   *R11+,R2        ; R2=address of param 2
MOV   *R11+,R3        ; R3=address of param 3
```

Notice how the indirect with auto increment addressing mode avoids the need for intermediate increments.

2.1.4 Returning Results

Many subroutines must return results to the calling program. The easiest way is to return the result in one of the general registers. This works fine if the subroutine is called via a BL instruction. On the other hand, if a BLWP (or XOP - which will be discussed later) is used, the calling routine uses a different set of registers than the subroutine. Therefore, if we place the results in the registers, they will be lost when control is returned to the calling program since the workspace pointer will be reset. Since the 9900's registers are located in memory, there is a simple way around this problem. Let's assume that we want to return a value in R0 and R1 - in the old workspace. When the BLWP is executed, the old workspace pointer is saved in R13. Using this fact, we can create a sequence to store values in the previous workspace:

```
MOV   R0,*R13         ; old R0=new R0
MOV   R1,@2(R13)      ; old R1=new R1
```

As you see, the old register R1 is the same as memory location R13+2*1. That location may be addressed by @2(R13). R0 is a special case since @0(R13) is the same as *R13.

2.1.5 Byte Operations

Although the 9900 is a 16-bit processor, it can still handle byte operations. There are a few aspects of the byte operations that are initially confusing. First, whenever, a register is addressed in the byte mode, the left byte of the register is used (not the right byte). Second, whenever the processor references memory it reads a full word. The proper byte of that word is selected within the processor. This means that it is not necessary for the processor to supply the external memory addressing circuitry with the least significant address bit - so it does not. If you examine the hardware carefully you will note that there are only fifteen address bits. The missing bit is the least significant address bit. It is unnecessary because the processor performs the byte selection.

Recognizing the special byte addressing operation, you will quickly discover that the 9900 can cope with byte operands nearly as well as it can with full word operands. To add the contents of byte B1 to B2 we can use:

```
AB    @B1,@B2          ; B2=B2+B1
```

2.1.6 Extended Operations

The 9900 offers a unique instruction, Extended Operation (XOP). The XOP execution is similar to the BLWP, but the target address is determined by the XOP transfer vectors. There are sixteen possible XOPs. During the XOP call, the source operand is placed in R11 of the new workspace. For example, the following:

```
XOP   @X,15
```

will perform an extended operation 15 and will place the address of variable X in the new R11. The workspace pointer and address for extended operation 15 is in memory locations 7C-7F. For other extended operations, the extended operation transfer vector is stored in location 40+4*I through 43+4*I.

The monitor uses three extended operations. Refer to the monitor description details of the monitor XOP'S.

2.1.7 Multiply/Divide

One of the truly unique operations offered in the 9900 is the hardware multiply and divide. Notice, however, that they require unsigned operands. This is different than the other instructions, which use two's complement operands. We can easily form a signed two's complement multiply. If X1 and X2 are two arbitrary numbers, then X1*X2's sign is the exclusive-or of the signs of X1 and X2. Using this fact we can devise the routine to perform signed multiply. The sequence below will calculate X3=X1*X2.

Assume: X1 is @>200, X2 is @>202, X3 is @>204

```
     MOV @>200,R1        ; R1=X1
     MOV @>202,R3        ; R3=X2
     MOV R1,R2           ; R2(SIGN)=SIGN OF X1*X2
     XOR R3,R2           ;
     ABS R1              ; GET RID OF SIGNS
     ABS R3              ;
     MOV R2,R2           ; TEST SIGN OF ANSWER
     MPY R3,R1           ; (R1,R2)=ANSWER
     JGT OK              ; CORRECT THE SIGN
     NEG R2              ;
OK   MOV R2,@>204        ; SAVE ANSWER
```

The multiply operation produces a 32-bit result (in R1, R2 for the example above), but does not affect any of the condition bits (thats why the test can be performed before the multiply). After the multiply, the result can be converted back to two's complement. Since you will often use the result for some further add/subtract operation, only the lower word of the product was converted. If you need to convert both words, its a bit more difficult. The following sequence will <u>not</u> work:

```
     NEG  R2
     NEG  R3
```

Why not? if R2=1 and R3=1, then the two's complement of (R2,R3) is >FEFF. However, the two's complement of 1 is FF. So you see that the above sequence would yield >FFFF instead of the required >FEFF. The solution is to take the one's complement of R2 except in the case where R3=0. The required code is:

```
     INV  R2             ; R2=one's comp. of R2
     NEG  R3             ; R3=R3
     JNE  ZRO            ; if R3=0, adjust R2
     INC  R2             ; R2=two's comp. of R2
ZRO  .
     .
     .
```

A similar approach can be used to construct a signed divide. The sign of X1/X2 is again the exclusive-or of X1,X2. If X1 and X2 are both 16-bit two's complement variables, then the routine below will calculate X2=X1/X2.

Assume: X1 is @>200, X2 is @>202

```
     MOV @>200,R2        ; R2=X1
     MOV @>202,R3        ; R3=X2
     MOV R2,R4           ; R4(SIGN)=SIGN OF X1/X2
     XOR R3,R4           ;
     ABS R2              ; GET RID OF SIGNS
     ABS R3              ;
     CLR R1              ; CLEAR UPPER BITS OF NUMERATOR
     DIV R3,R1           ; R1=(R1,R2)/R3
     MOV R4,R4           ; CORRECT SIGN
     JGT OK              ;
     NEG R1
OK   MOV R1,@>202        ; SAVE ANSWER
```

As you may have observed in that sequence, the divide operation divides a 32-bit operand by a 16-bit operand. Since we used only a 16-bit operand, the operand is placed in the lower register of the pair of registers and the upper register of the pair is cleared. If we want to use the full divide capability, the routine must be recoded as:

Assume: X1 is @>200 to >203 and X2 is @>204 to >207

```
     MOV @>200,R1        ; (R1,R2)=X1
     MOV @>202,R2        ;
     MOV @>204,R3        ; R3=X2
     MOV R1,R4           ; R4(SIGN)=SIGN OF X1/X2
     XOR R3,R4           ;
     ABS R3              ; GET RID OF SIGN OF X2
     ABS R1              ; GET RID OF SIGN OF X1
     JGT OK1             ; IF X1<0, INVERT LOWER HALF
     NEG R2              ;
     JEQ OK1             ; IF R2 NOT ZERO, ADJUST R1
     DEC R1              ;
OK1  DIV R3,R1           ; R1=X1/X2
     MOV R4,R4           ; CORRECT THE SIGN
     JGT OK2             ;
     NEG R1              ;
OK2  MOV R1,@>204        ; X2=X1/X2
```

The multiply is restricted to integer operands, but that does

not mean you cannot use it to perform fractional operations. The approach for fractional multiplication is called scaling. Lets take a sample case. If the decimal point of X1 is at the extreme right and the decimal point of X2 is at the extreme left, then the decimal point of X1*X2 is between the two registers. Using this approach, we can multiply ABC by .75:

```
CON  DATA >C000          ; constant of .75 (decimal at left)
     .
     .
     .
     MOV @ABC,R1         ; get operand
     MPY @CON,R1         ; R1=integer part, R2=fraction part
```

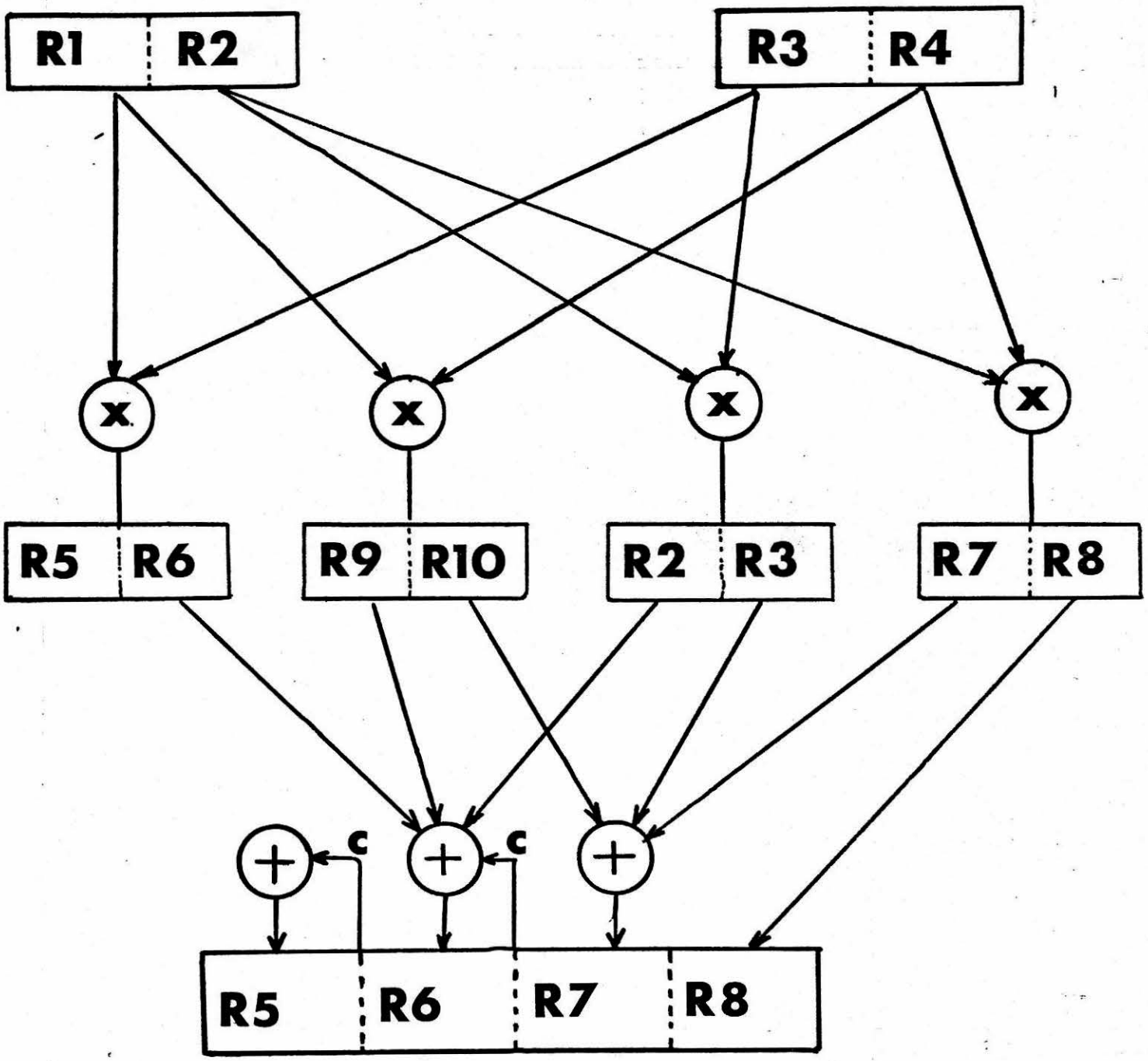
In the beginning of this discussion, We indicated that it was unusual that the multiply was unsigned. Yet, we can turn this into an asset. Consider the problem of creating a double precision multiply (32-bits times 32-bits). If we consider unsigned numbers only (signs can be handled as in the previous examples), then a 32-bit multiply (which produces a 64-bit result) can be formed using four single precision multiplies. Figure 2.2 illustrates the concept. We use what is commonly called "cross multiply" techniques. Before presenting the double precision multiply, lets look at the double precision add which is an integral part of the multiply routine. To calculate (R1,R2)=(R1,R2)+(R3,R4) we can use the following (all values are assumed to be unsigned):

```
     A    R4,R2          ; add lower half
     JNC  L1             ; if Cy, correct upper
     INC  R1
L1   A    R3,R1          ; add upper half
```

Now, using this same concept for the subproduct additions, we can create the 32-bit multiply routine:

```
        MOV R1,R5           ; (R5,R6)=R1*R3
        MPY R3,R5           ;
        MOV R2,R7           ; (R7,R8)=R2*R4
        MPY R4,R7           ;
        MOV R1,R9           ; (R9,R10)=R1*R4
        MPY R4,R9           ;
        MPY R2,R3           ; (R3,R4)=R2*R3
        CLR R0              ; R0=CARRY ACCUMULATOR
        A R3,R7             ;
        JNC OK1             ;
        INC R0              ;
OK1     A R10,R7            ;
        JNC OK2             ;
        INC R0              ;
OK2     CLR R1              ; R1=CARRY ACCUMULATOR
        A R2,R6             ;
        JNC OK3             ;
        INC R1              ;
OK3     A R9,R6             ;
        JNC OK4             ;
        INC R1              ;
OK4     A R0,R6             ; ADD FIRST CARRY
        JNC OK5             ;
        INC R1              ;
OK5     A R1,R5             ; ADD SECOND CARRY
```

Figure 2.2 32-Bit Multiply Technique



2.1.8 ARITHMETIC

The advanced instruction set of the 9900 CPU, opens up a new microprocessor application area - signal processing. Because of the mathematics involved, most signal processing tasks cannot be done with the off-the-shelf microprocessor. The 9900 certainly cannot handle all of the signal processing applications, but it can tackle a few of them.

Many signal processing algorithms use the SIN, COS, or other trigonometric functions. An algorithm to compute trig functions - ideally suited to the 9900, is the CORDIC (Coordinate Rotation Digital Computer) algorithm. Although you may not recognize it, it is the same algorithm used in many hand calculators. We will see later why the 9900 is ideally suited for the CORDIC procedure.

The CORDIC algorithm relies on a few very simple mathematical facts. First, any given angle (we will restrict the angle to 0-90 degree) can be represented as a sum/difference of a set of base angles. Mathematically this can be expressed:

A=SUM(d(i)*a(i)), where d(i)=+/-1 a(i)=base angle

This identity is certainly not true for any random delection of base angles, but you can intuitively sees that 90 degrees, 45 degrees, 22.5 degree, ... is one possible base set. The second cornerstone of this algorithm is a pair of trigonometric identities:

SIN(a+b)=(SIN(a)+TAN(b)COS(a))COS(b)
COS(a+b)=(COS(a)-TAN(b)SIN(a))COS(b)

Now, if we have a given angle represented as a sum/difference of a set of base angles, which are as yet unspecified, then we can devise a simple process for calulating the SIN and COS of that angle (called A):

```
X(i)=A
Y(i)=1
X(i)=X(i-1)+TAN(d(i)a(i))*Y(i-1)
Y(i)=Y(i-1)-TAN(d(i)a(i))*X(i-1)
```

After executing the above procedure, we don't really have the SIN and COS. Instead, we have X(n) = R(n)SIN(A) and Y(n) = R(n)COS(A), where the constant R(n) is 1/(COS(d(i)a(i))* ... *COS(d(i)a(i)). So far, we have nothing to cheer about because our algorithm involves many more multiplies, than a simple Taylor series. But, the plot thickens. If we define the base angles as:

```
a(i)=ArcTan(.5**(i-1))
```

then

```
TAN(a(i))=(.5**(i-1))
```

This means that all of the multiply operations can be reduced to a right shift. We must, of course, prove that all angles can be represented as a sum of our base angles or the whole algorithm collapses. I will not do so here, but it can be done rather easily. Now, if we use the base angles defined above, the algorithm may be restated as:

```
V(i)=-A
X(i)=0
Y(i)=1/R(i)=.60725
X(i)=X(i-1)-SIGN(V(i-1))*Y(i-1)/2**(i-1)
Y(i)=Y(i-1)+SIGN(V(i-1))*X(i-1)/2**(i-1)
V(i)=V(i-1)-SIGN(V(i-1))*ATAN(1/(2**(i-1))
```

If we store the ArcTan values in a table, then this algorithm requires only shift, add, and subtract. The shift operation requires a variable shift constant. This is why the algorithm fits nicely in the 9900. If the shift count is stored in R0, the variable shift can be performed by a single 9900 instruction:

```
SRA  R1,R0          ; shift R1 right by (R0)
```

Since the SIN and COS are fractional values, we must scale the input to our routine. To keep matters simple, we scale the angle so that R1=angle*256. This provides 8-bits of integer and 8-bits of fraction. We scale the X,Y values so that X=SIN*32768, and Y=COS*32768. This provides 16-bits of signed fraction. The entire algorithm is shown in Figure 2.3. The input angle is in R1, and the outputs are in R2 and R3. This subroutine calculates <u>both</u> the SIN and COS. The TAN can be calculated by one additional divide. As you see, this algorithm is a very fast and efficient way to obtain the trigonometric values.

Figure 2.3 Cordic Routine

```
      CLR R2              ; X=0
      LI R3,19898         ; Y=.6072526*(2**15)
      CLR R4              ; X0=0
      MOV R3,R5           ; Y0=Y
      CLR R0              ; SHIFT=0
      CLR R6              ; COUNT=0
      NEG R1              ; V1=-V
LOOP  MOV R1,R1           ; TEST SIGN OF ANGLE
      JLT LESS            ; JUMP IF MINUS
      S R5,R2             ; C=C-Y/2**I
      A R4,R3             ; Y=Y+X/2**I
      S @TAB(R6),R1       ; V=V-ATAN(1/2**I)
      JMP CONT            ;
LESS  A R5,R2             ; X=X+Y/2**I
      S R4,R3             ; Y=Y-X/2**I
      A @TAB(R6),R1       ; V=V+ATAN(1/2**I)
CONT  INC R0              ; UPDATE SHIFT COUNT
      INCT R6             ; UPDATE ANGLE INDEX
      MOV R2,R4           ; R4=X/2**I
      SRA R4,R0           ;
      MOV R3,R5           ; R5=Y/2**I
      SRA R5,R0
      CI R0,12            ; END?
      JNE LOOP            ;
      B *R11              ; RETURN TO CALLER


TAB   DATA 11520          ; ATAN(1/1)*256
      DATA 6800           ; ATAN(1/2)*256
      DATA 3593           ; ATAN(1/4)*256
      DATA 1824           ; ATAN(1/8)*256
      DATA 916            ; ATAN(1/16)*256
      DATA 458            ; ATAN(1/32)*256
      DATA 229            ; ATAN(1/64)*256
      DATA 115            ; ATAN(1/128)*256
      DATA 57             ; ATAN(1/256)*256
      DATA 29             ; ATAN(1/512)*256
      DATA 14             ; ATAN(1/1024)*256
      DATA 7              ; ATAN(1/2048)*256
```

2.2 Instructions and Addressing

2.2.1 Workspace Register Addressing

The contents of the indicated workspace register is the operand. (e.g. R3, R7)

2.2.2 Workspace Register Indirect Addressing

The contents of the indicated workspace register contains the memory address of the operand. (e.g. *R3,*R6)

2.2.3 Indexed Addressing

The contents of the indicated workspace register (R0 is not allowed as an index register) are added to the address enclosed in the second command word. (e.g. @2(R1),@6(R4))

2.2.4 Direct Addressing

The word following the instruction contains the memory address of the operand. (e.g. @6, @123)

2.2.5 Workspace Register Indirect with Auto Increment Addressing

The contents of the indicated workspace register contain the memory address of the operand. The workspace register is automatically incremented after the access (plus 2 for word operations and plus 1 for byte operations). (e.g. *R1+,*R9+,*14+)

2.2.6 Immediate Addressing

The word following the instruction contains the operand. (e.g. 26)

2.2.7 Relative Addressing

The 8-bit displacement of the instruction is added to the updated program counter in jump instructions or to the base address in single-bit CRU instructions.

2.2.8 Status Register

The CPU status register holds the condition bits as follows:

0	1	2	3	4	5	6-11	12-15
LGT	AGT	EQ	C	OV	OP	N/A	Interrupt

LGT - Logical Greater Than
AGT - Arithmetic Greater Than
EQ - Equal
C - Carry
OV - Overflow
P - Odd Parity

2.2.9 Instruction Description

The following shorthand notation is used to describe the 9900 CPU instruction set.

S - General address for the source operand. Any addressing mode is acceptable.

D - General address for the destination operand. Any addressing mode is acceptable.

IOP - Immediate operand

W - Workspace register

DISP - Relative displacement

WP - Workspace pointer

PC - Program counter

ST - Status Register

() - Contents of address or register

---> - Replaces

INSTRUCTION: ADD

INST FORMAT: A S,D

HEX. OPCODE: A000

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: The source operand is added to the destination operand. The sum replaces the destination operand.

INST RESULT: (S)+(D)--->(D)

APPL. NOTES: Use to add 16 bit numbers from:

Memory to Memory	A @SCALE,@TABLE
Register to Register	A R10,R9
Memory to Register	A @PRIME,R6
Register to Memory	A R14,@SUM

INSTRUCTION: ADD BYTES

INST FORMAT: AB S,D

HEX. OPCODE: B000

STAT CHANGE: LGT,AGT,EQ,C,OV,OP

DESCRIPTION: Add two 8-bit bytes. The 8-bit source operand is added to the 8-bit destination operand. If either address is a workspace register, then the left-most eight bits of that workspace register will be used.

INST RESULT: (S)+(D)--->(D)

APPL. NOTES: Used to add signed 8-bit numbers from:

Memory to Memory	AB @X,@Y
Register to Memory	AB R1,@Y
Memory to Register	AB @X,R1
Register to Register	AB R1,R2

INSTRUCTION: ABSOLUTE VALUE

INST FORMAT: ABS S

HEX. OPCODE: 0740

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: Compute the absolute value of the source operand and replace the source operand with that result.

INST RESULT: Absolute value of (S)--->(S)

APPL. NOTES: Used to compute the absolute value of a 16-bit number.

```
ABS @LISTA
ABS @LISTB
```

	BEFORE	AFTER
LISTA	FFF4	000C
LISTB	000C	000C

INSTRUCTION: ADD IMMEDIATE

INST FORMAT: AI W,IOP

HEX. OPCODE: 0220

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: Add the immediate value to the specified workspace register.

INST RESULT: (W)+IOP--->(W)

APPL. NOTES: Add a constant to a workspace register.

```
AI R4,100     Add 100 to register R4
AI R11,10     Add ten to register R11
```

INSTRUCTION: AND IMMEDIATE

INST FORMAT: ANDI W,IOP

HEX. OPCODE: 0240

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Perform a bit-by-bit logical AND operation between the workspace register and the immediate operand. Place the result in the workspace register.

INST RESULT: (W) AND IOP--->(W)

APPL. NOTES: Use to isolate certain bits of a workspace register.

```
ANDI 6,>F00E

Before: (R6)=>9877      1001 1000 0111 0111
Immed. Operand=>F00E    1111 0000 0000 1110
After: (R6)=>9006       1001 0000 0000 0110
```

INSTRUCTION: UNCONDITIONAL BRANCH

INST FORMAT: B S

HEX. OPCODE: 0440

STAT CHANGE: None

DESCRIPTION: Replace PC with the source address. Effectively, transfers control to the source address.

INST RESULT: S--->(PC)

APPL. NOTES: This is the most flexible jump and can be used to transfer control to any location in memory. If the jump is out of range (+127, -128 words) for a relative jump instruction, use B.

B @107 will cause PC to be set to 107

INSTRUCTION: BRANCH AND LINK TO SUBROUTINE

INST FORMAT: BL S

HEX. OPCODE: 0680

STAT CHANGE: None

DESCRIPTION: Place source address in PC and place the address of the instruction following the BL instruction in workspace register R11.

INST RESULT: (PC)--->(R11)
S--->(PC)

APPL. NOTES: Use to transfer control to a subroutine. Return from the subroutine is accomplished with a branch indirect through register 11.

```
BL @SUB  ---------------> SUB .
.                           .
.                           .
.        <--------------- B *R11
```

INSTRUCTION: BRANCH AND LOAD WORKSPACE POINTER

INST FORMAT: BLWP S

HEX. OPCODE: 0400

STAT CHANGE: None

DESCRIPTION: Place source operand into WP and the word following it into the PC. Place previous contents of WP into R13 of the new workspace, PC(address immediately following BLWP) into the new R14 and ST into the new R15.

INST RESULT:
(S)--->(WP)
(S+2)--->(PC)
(original WP)--->(R13)
(original PC)--->(R14)
(original ST)--->(R15)

APPL. NOTES: Use to call a subroutine and change the workspace environment. The subroutine must return via RTWP command.

```
BLWP R4      Place (R4) in WP, (R5) in PC
BLWP @SBR    WP=(SBR), PC=(SBR+2)
```

The calling routine's registers can be accessed using indexed addressing since R13 is the old workspace pointer. For example, *R13 is the calling routine R0, @8(R13) is the calling R4, etc.

INSTRUCTION: COMPARE

INST FORMAT: C S,D

HEX. OPCODE: 8000

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Compare the contents of the source operand with the contents of the destination operand and set/reset designated status register bits.

INST RESULT: Status register bits set/reset after comparison.

APPL. NOTES: Use to compare 16-bit numbers from:

```
Memory to Memory       C @TOP,@LAST
Register to Register   C R1,R6
Memory to Register     C @BOT,R5
Register to Memory     C R7,@11
```

INSTRUCTION: COMPARE BYTES

INST FORMAT: CB S,D

HEX. OPCODE: 9000

STAT CHANGE: LGT,AGT,EQ,OP

DESCRIPTION: Compare the contents of the source operand byte with the contents of the destination operand byte and set/reset the designated status register bits.

INST RESULT: Status Register bits set/reset after comparison.

APPL. NOTES: Use to compare 8-bit numbers. If a workspace register is used for S or D, the left-most 8-bits will be used.

CB R1,R2 Compare R1(byte) to R2(byte)

INSTRUCTION: COMPARE IMMEDIATE

INST FORMAT: CI W,IOP

HEX. OPCODE: 0280

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Compare the contents of the specified register with the immediate operand and set/reset the designated status register bits.

INST RESULT: status register bits set/reset after comparison

APPL. NOTES: Compare the contents of workspace register with some known value and set status register bits accordingly.

```
CI R2,>73     Compare register R2 to >73
CI R3,0       Compare register R3 to zero.
              (A more efficient way is:
               MOV R3,R3)
```

INSTRUCTION: CLEAR

INST FORMAT: CLR S

HEX. OPCODE: 04C0

STAT CHANGE: None

DESCRIPTION: Replace source operand with a full 16-bit word of zeroes.

INST RESULT: 0--->(S)

APPL. NOTES: Use to zero workspace registers or memory locations.

```
CLR R5          Clear register R5
CLR @SUM        Clear location SUM


                LI R1,X  Clear (X) to (X+10)
LOOP            CLR *R1+
                CI R1,X+12
                JL LOOP
```

INSTRUCTION: COMPARE ONES CORRESPONDING

INST FORMAT: COC S,W

HEX. OPCODE: 2000

STAT CHANGE: EQ

DESCRIPTION: When all ones in the source operand have a corresponding one in the destination workspace register, set the equal bit in the status register.

INST RESULT: EQ status bit is set/reset.

APPL. NOTES: Use to check if a bit or bits in a destination workspace register are set to one. Bits correspond to the one bits in the source operand. If corresponding bits in destination are also set, the equal bit in Status Register is also set.

```
Assume      TEST=C102=1100 0001 0000 0010
              R8=E306=1110 0011 0000 0110

Then        COC @TEST,R8
```

Every one bit in TEST has a corresponding one bit in register R8. Therefore the equal status bit is set.

```
MASK  DATA  8000
      COC @MASK,R1     Is sign in R1 set?
      JEQ  ADD         If so, jump to ADD
```

INSTRUCTION: COMPARE ZEROES CORRESPONDING

INST FORMAT: CZC S,W

HEX. OPCODE: 2400

STAT CHANGE: EQ

DESCRIPTION: When the bits in the destination workspace register corresponding to the one bits in the source operand are all equal to a logic zero, set equal status bit.

INST RESULT: Set/reset status register equal bit.

APPL. NOTES: Use to test single/multiple bits within a workspace register.

```
Assume      TEST=C102=1100 0001 0000 0010
              R8=2201=0010 0010 0000 0001

Then        CZC @TEST,R8
```

Every logic one bit in TEST corresponds to a logic zero in register R8. Therefore, the equal status bit is set.

INSTRUCTION: DECREMENT BY ONE

INST FORMAT: DEC S

HEX. OPCODE: 0600

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: Subtract one from the 16-bit source operand.

INST RESULT: (S)-1--->(S)

APPL. NOTES: Used for indexing or controlling loops.

```
DEC @TEC      TEC=TEC-1
JNE LOOP      Jump if TEC not zero
```

INSTRUCTION: DECREMENT BY TWO

INST FORMAT: DECT S

HEX. OPCODE: 0640

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: Subtract two from the 16-bit source operand.

INST RESULT: (S)-2--->(S)

APPL. NOTES: Useful for counting and indexing full word arrays.

```
DECT @COUNT   Subtract two from COUNT
DECT R10      Subtract two from register 10
```

INSTRUCTION: DIVIDE

INST FORMAT: DIV S,W

HEX. OPCODE: 3C00

STAT CHANGE: OV

DESCRIPTION: Divide the destination operand (a 32-bit unsigned integer) by the source operand (a 16-bit unsigned integer) using integer arithmetic and place the quotient in the destination operand and the remainder in the second word of the destination operand. If the quotient exceeds 16-bits, the overflow is set.

INST RESULT: (W,W+1)/(S)---->(W) quotient;
(W+1) remainder

APPL. NOTES: Use divide for integer division (unsigned).

```
DIV R3,R4     Divide R4,R5 by (R3)
DIV @SUM,2    Divide R2,R3 by (SUM)
```

INSTRUCTION: IDLE COMPUTER

INST FORMAT: IDLE

HEX. OPCODE: 0340

STAT CHANGE: None

DESCRIPTION: Place the computer in an IDLE state.

INST RESULT: Computer is IDLE.

APPL. NOTES: Used to halt the processor and wait for an interrupt.

INSTRUCTION: INCREMENT BY ONE

INST FORMAT: INC S

HEX. OPCODE: 0580

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: Add one to the 16-bit source operand.

INST RESULT: (S)+1--->(S)

APPL. NOTES: Useful for controlling byte addressing of an index.

```
INC R6        R6=R6+1
INC @T(R1)    increment table location
              selected by R1
```

INSTRUCTION: INCREMENT BY TWO

INST FORMAT: INCT S

HEX. OPCODE: 05C0

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: Add two to the 16-bit source operand.

INST RESULT: (S)+2--->(S)

APPL. NOTES: Useful for controlling word addressing of an index.

INSTRUCTION: INVERT

INST FORMAT: INV S

HEX. OPCODE: 0540

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: The 16-bit source operand is replaced with its one's complement.

INST RESULT: One's complement of (S)--->(S)

APPL. NOTES: Use this operation to "flip" the bits in some memory location or register.

```
INV R2        Invert location (SUM)
INV *R3       Invert location in register R3
```

INSTRUCTION: JUMP EQUAL

INST FORMAT: JEQ DISP

HEX. OPCODE: 1300

STAT CHANGE: None

DESCRIPTION: When the equal .status bit is set, the signed displacement is added to the PC.

INST RESULT: (PC)+(displacement)--->PC (EQ=1)
(PC)+2--->PC (EQ=0)

APPL. NOTES: Used to transfer if equal

```
C @X,@Y
JEQ YES      go to YES if (X)=(Y)
```

INSTRUCTION: JUMP IF GREATER THAN

INST FORMAT: JGT DISP

HEX. OPCODE: 1500

STAT CHANGE: None

DESCRIPTION: When the arithmetic greater than status bit is set, add the signed displacement to the PC.

INST RESULT: (PC)+Displacement--->(PC) (AGT=1)
(PC+2--->(PC) (AGT=0)

APPL. NOTES: Used following a 16-bit arithmetic operation:

```
C @ONE,@TWO
JGT @OUI      go to OUI if (ONE)is
              arithmetically greater than
(TWO)
```

The arithmetic greater than is the result of a signed compare, so >FFFF (-1) is <u>not</u> arithmetic greater than >7FFF, but it is logical greater than.

INSTRUCTION: JUMP ON HIGH

INST FORMAT: JH DISP

HEX. OPCODE: 1B00

STAT CHANGE: None

DESCRIPTION: When the logical greater than status bit is set and the equal status bit is clear then the signed displacement is added to the PC.

INST RESULT: (PC)+Displacement--->(PC) (LGT=1 and EQ=0)
(PC+2--->(PC) (LGT=0 or EQ=1)

APPL. NOTES: Used when comparing logical or unsigned values.

```
C @BIG,@GOOD
JH @BAD        go to BAD if (BIG) is
               logically greater than
               (GOOD) - (unsigned)
```

Since the logical greater than is an unsigned compare, this instruction is most often used for address comparisons. But beware, nothing is higher than >FFFF.

INSTRUCTION: JUMP ON HIGH OR EQUAL

INST FORMAT: JHE DISP

HEX. OPCODE: 1400

STAT CHANGE: None

DESCRIPTION: When the equal status bit or the logical greater than status bit is set, the signed displacement is added to the PC.

INST RESULT: (PC)+Displacement--->(PC) (LGT=1 or EQ=1)
(PC)+2--->(PC) (LGT=0 and EQ=0)

APPL. NOTES: Use to branch or ttansfer control when either logical greater than or equal status bits=1.

```
JHE $+4     If SR bits 0 or 2=1, skip one
            word.
JHE SUB     If SR bits 0 or 2=1, jump to
            SUB
```

INSTRUCTION: JUMP ON LOW

INST FORMAT: JL DISP

HEX. OPCODE: 1A00

STAT CHANGE: None

DESCRIPTION: When the logical greater than and equal status bits are both reset, then the signed displacement is added to the PC.

INST RESULT: (PC)+Displacement--->(PC) (LGT=EQ=0)
(PC)+2--->(PC) (LGT=1 or EQ=1)

APPL. NOTES: Use to transfer control when a logical or unsigned less than condition is detected.

```
C @ONE,@TWO
JL @GO          go to GO if (ONE) logically
                less than (TWO) (unsigned)
```

INSTRUCTION: JUMP ON LOW OR EQUAL

INST FORMAT: JLE DISP

HEX. OPCODE: 1200

STAT CHANGE: None

DESCRIPTION: When the equal status bit is set or the logical greater than is reset, then the signed displacement is added to the PC.

INST RESULT: (PC)+Displacement--->(PC) (LGT=0 or EQ=1)
(PC)+2--->(PC) (LGT=1 and EQ=0)

APPL. NOTES: Use to test status register bits and transfer control if LGT=0 or EQ=1.

```
JLE ADDNO     If SR bits 0=0 or 2=1,
              go to ADDNO
```

INSTRUCTION: JUMP ON LESS THAN

INST FORMAT: JLT DISP

HEX. OPCODE: 1100

STAT CHANGE: None

DESCRIPTION: If the arithmetic greater than and equal status bits are reset then add the signed displacement to the PC.

INST RESULT: (PC)+Displacement--->(PC) (LGT=EQ=0)
(PC)+2--->(LGT=1 or EQ=1)

APPL. NOTES: Used when comparing arithmetic values.

```
C @A,@B
JLT LESS      go to LESS if (A) is
              arithmetically less than (B)
```

INSTRUCTION: UNCONDITIONAL JUMP

INST FORMAT: JMP DISP

HEX. OPCODE: 1000

STAT CHANGE: None

DESCRIPTION: Add the signed displacement to the PC and place the sum into the PC.

INST RESULT: (PC)+Displacement--->PC

APPL. NOTES: Use to transfer control unconditionally.

```
JMP LOOP     Begin execution at loop
JMP $        Remain at this location
JMP $+4      Jump over next address
```

The destination address must be within the range+127 to -128 words. If not, use the branch (B) instruction.

INSTRUCTION: JUMP ON NO CARRY

INST FORMAT: JNC DISP

HEX. OPCODE: 1700

STAT CHANGE: None

DESCRIPTION: If the carry status bit is clear, add the signed displacement to the PC.

INST RESULT: (PC)+Displacement--->(PC) (C=0)
(PC)+2--->(PC) (C=1)

APPL. NOTES: Use to branch when carry cleared.

```
        JNC YES      If carry clear, go to YES
```

Can be used to check for 16-bit carry for multi-precision arithmetic. The following will calculate (R1,R2)+(R3,R4).

```
            A R4,R2
            JNC GO
            INC R1
GO          A R4,R1
```

INSTRUCTION: JUMP ON NOT EQUAL

INST FORMAT: JNE DISP

HEX. OPCODE: 1600

STAT CHANGE: None

DESCRIPTION: If the equal status bit is reset, add the signed displacement to the PC.

INST RESULT: (PC)+Displacement--->(PC) (EQ=0)
(PC)+2--->(PC) (EQ=1)

APPL. NOTES: Used to branch when not equal.

```
A R1,R2
JNE X          go to X if R1+R2 not zero
MOV R1,R1
JNE NO         go to NO if R1 not zero
```

INSTRUCTION: JUMP ON NO OVERFLOW

INST FORMAT: JNO DISP

HEX. OPCODE: 1900

STAT CHANGE: None

DESCRIPTION: When the overflow status bit is reset, add the signed displacement to the PC.

INST RESULT: (PC)+Displacement--->(PC) (OV=0)
(PC)+2--->(PC) (OV=1)

APPL. NOTES: Used to test arithmetic overflow.

```
A R1,R2
JNO GOOD      go to GOOD IF R1+R2 does
              not overflow
```

An overflow occurs during an add if the sign of the two operands are the same but the sign of the sum is not the same.

INSTRUCTION: JUMP ON CARRY

INST FORMAT: JOC DISP

HEX. OPCODE: 1800

STAT CHANGE: None

DESCRIPTION: When the carry status bit is set, add the signed displacement to the PC.

INST RESULT: (PC)+Displacement--->(PC) (C=1)
(PC)+2--->(PC) (C=0)

APPL. NOTES: Use to branch or transfer control if carry is set.

```
JOC START     If Carry, Go to Start
JOC $-2       If Carry, Go to Previous
              Instruction
```

INSTRUCTION: JUMP ON ODD PARITY

INST FORMAT: JOP DISP

HEX. OPCODE: 1000

STAT CHANGE: None

DESCRIPTION: When the odd parity status bit is set, add the signed displacement to the PC.

INST RESULT: (PC)+Displacement--->(PC) (OP=1)
(PC)+2--->(PC) (OP=0)

APPL. NOTES: Used to test parity of 8-bit values.

```
MOVB @CH,R1
JOP ODD       go to ODD if CH is odd parity
```

Note that the OP flag is only changed by byte instructions (e.g. MOVB,CB)

INSTRUCTION: LOAD COMMUNICATIONS REGISTER UNIT (OUTPUT)

INST FORMAT: LDCR S,C

HEX. OPCODE: 3000

STAT CHANGE: LGT,AGT,EQ,OP (IF C<9)

DESCRIPTION: Transfer the number of bits specified (C) from the source operand to consecutive CRU lines. The contents of R12 determines the least significant CRU line.

INST RESULT: (S)--->CRU for C bits

APPL. NOTES: Use this to output a bit pattern to CRU lines for versatile I/O. If the number of bits specified is less than nine, then S is a byte address. If the number of bits is nine or more, S is a word address. The least significant memory bit goes to the least significant CRU bit. If the bit count (C) is zero, then 16 bits are output. Prior to an LDCR instruction, resigter R12 (CRU Base Address) must be loaded witheth appropriate address. For the T99SS CPU module, R12=0 will address bit 0.

```
LDCR 2,0      16 bits to CRU from R2
LDCR @NM,8    8 bits to CRU from NM
```

INSTRUCTION: LOAD IMMEDIATE

INST FORMAT: LI W,IOP

HEX. OPCODE: 0200

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Place the immediate operand in the specified register.

INST RESULT: IOP--->(W)

APPL. NOTES: Use to initialize register for counters or addresses.

```
LI R5,TABLE   Load R5 with address of TABLE
LI R1,10      Set R1 to 10
LI R2,>100    Set R2 to 100 (Hex)
```

INSTRUCTION: LOAD INTERRUPT MASK IMMEDIATE

INST FORMAT: LIMI IOP

HEX. OPCODE: 0300

STAT CHANGE: Interrupt Mask

DESCRIPTION: Place the four least significant bits of IOP into the interrupt mask (bits 15-12 of the Status Register).

INST RESULT: IOP (15-12)--->ST (15-12)

APPL. NOTES: Used to enable or disable interrupts.

```
LIMI 0      disable all interrupts
LIMI >F     enable all interrupts
```

INSTRUCTION: LOAD WORKSPACE POINTER IMMEDIATE

INST FORMAT: LWPI IOP

HEX. OPCODE: 02E0

STAT CHANGE: None

DESCRIPTION: Replace contents of workspace pointer register with the beginning address of 16 contiguous words. This changes the current workspace pointer and environment.

INST RESULT: IOP--->(WP)

APPL. NOTES: Use to initialize the WP register to alter workspace environment.

```
LWPI >100     Place >100 in workspace pinter
LWPI WSP      Location WSP = Register 0
```

INSTRUCTION: MOVE WORDS

INST FORMAT: MOV S,D

HEX. OPCODE: C000

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Replace destination operand with a copy of the source operand.

INST RESULT: (S)--->(D)

Memory to Memory	MOV @TABLE,@TEMP
Register to Register	MOV R5,R9
Register to Memory (STORE)	MOV R3,@ANSWER
Memory to register (LOAD)	MOV @TABL,R8

INSTRUCTION: MOVE BYTES

INST FORMAT: MOVB S,D

HEX. OPCODE: D000

STAT CHANGE: LGT,AGT,EQ, OP

DESCRIPTION: Move the source byte operand to the destination byte operand. Whenever S or D is a workspace register, then the leftmost 8-bits are used.

INST RESULT: (S)--->(D)

APPL. NOTES: Transfer bytes of data.

Load register	MOVB @X,R1
Store register	MOVB R3,@13(R10)
Move Memory to Memory	MOVB @X,@Y
Move Register to Register	MOVB R3,R4

INSTRUCTION: MULTIPLY

INST FORMAT: MPY S,W

HEX. OPCODE: 3800

STAT CHANGE: None

DESCRIPTION: Mutiply the destination operand, an unsigned 16-bit integer by the source operand, an unsigned 16-bit integer. Place the product into the 32 bit (two word) destination field right justified.

INST RESULT: (W)*(S)--->(W,W+1)

APPL. NOTES: Use multiply (MPY) to multiply two 16-bit unsigned integers. The destination operand must be a workspace Register, therefore the result will be in the workspace register specified and the next one. If workspace register 15 is specified then the next memory location following the workspace area is the second half of the product.

```
MPY *1,4      Mult. reg R4 by reg R1 (ind)
MPY @NUM,4    Mult. reg R4 by (NUM)
```

INSTRUCTION: NEGATE

INST FORMAT: NEG S

HEX. OPCODE: 0500

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: Replace source operand with two's complement value of the source operand.

INST RESULT: 0-(S)--->(S)

APPL. NOTES: Use NEG to replace the operand with its additive inverse.

```
NEG R7
```

The contents of workspace register R7 is replaced with its two's complement value.

INSTRUCTION: OR IMMEDIATE

INST FORMAT: ORI W,IOP

HEX. OPCODE: 0260

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Perform a logical OR operation between the specified workspace register and the immediate operand. Place the result in the workspace register.

INST RESULT: (W) OR IOP--->(W)

APPL. NOTES: Use to perform logical OR between workspace register and some known immediate value.

Example: ORI R10,>202D

```
Before:  R10=>1AD5     0001 1010 1101 0101
Imed.  Operand=        0010 0000 0010 1101
After:   R10=>3AFD     0011 1010 1111 1101

ORI R5,>8000           Set R5 sign bit
ORI R10,>F             Set four LSB of R10
```

INSTRUCTION: RETURN WITH WORKSPACE POINTER

INST FORMAT: RTWP

HEX. OPCODE: 0380

STAT CHANGE: All status bits set by R15, including interrupt mask.

DESCRIPTION: Replace contents of WP with contents of current R13, PC with contents of R14, ST with current value of R15.

INST RESULT: (R13)--->(WP)
(R14)--->(PC)
(R15)--->(ST)

APPL. NOTES: Use to return from a BLWP, XOP or a hardware interrupt.

INSTRUCTION: SUBTRACT WORDS

INST FORMAT: S S,D

HEX. OPCODE: 6000

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: Subtract the source operand from the destination operand and place the result in the destination operand.

INST RESULT: (D)-(S)--->(D)

APPL. NOTES: Use to subtract signed 16-bit integers from:

```
Memory to Memory        S @OLDVAL,@NEWVAL
Register to Register    S R8,R7
Register to Memory      S R10,@DIET
Memory to Register      S @CONS,R14
```

INSTRUCTION: SUBTRACT BYTES

INST FORMAT: SB S,D

HEX. OPCODE: 7000

STAT CHANGE: LGT,AGT,EQ,C,OV,OP

DESCRIPTION: Subtract the source operand byte from the destination operand byte and place the difference in the destination operand byte.

INST RESULT: (D)-(S)--->(D)

APPL. NOTES: Use to subtract signed integer bytes.

```
SB @3,@>503   Result in address >503
SB R1,R2      Result in upper byte of R2
```

INSTRUCTION: SET BIT ONE

INST FORMAT: SBO DISP

INSTRUCTION: SET BIT ZERO

INST FORMAT: SBZ DISP

HEX. OPCODE: 1E00

STAT CHANGE: None

DESCRIPTION: Set output CRU bit to a logical zero. The CRU bit address is determined by adding contents of bits 3-14 of R12 to the signed displacement.

INST RESULT: 0--->(CRU bit specified by bits 3-14 of R12 + displacement)

APPL. NOTES: Use To get the particular CRU line to a logical zero.

```
LI 12,>280    CRU base address=>140 (R12/2)
SBZ >28       Clears CRU bit >168 (140+28)
SBZ -2        Clears CRU bit >13E (140-2)
```

INSTRUCTION: SET TO ONES

INST FORMAT: SETO S

HEX. OPCODE: 0700

STAT CHANGE: None

DESCRIPTION: Replace the source operand with a 16-bit word of one's.

INST RESULT: FFFF--->(S)

APPL. NOTES: Use to initialize a table with -1 values instead of zeroes if your application requires such. Use to initialize register with -1.

```
SETO 5      Set register 5 to >FFFF
SETO @SUM   Set SUM to -1
```

INSTRUCTION: SHIFT LEFT ARITHMETIC

INST FORMAT: SLA W,C

HEX. OPCODE: 0AOO

STAT CHANGE: LGT,AGT,EQ,C,OV

DESCRIPTION: The contents of the workspace register are shifted left the specified number of bits (C) with zeroes filling the vacated bit positions. The last bit shifted out is placed in the carry out bit. If C=0; the right four bits of register R0 are used as the shift count.

INST RESULT: (W) is shifted left the specified shift count (C).

APPL. NOTES: Use to shift the contents of a workspace register left by some shift count.

```
SLA R4,8      Shift reg R4 left 8 places
SLA R4, 2     Effectively mult. reg R4 by 4
SLA R4,0      Shift reg R4 by contents of R0
```

Note that SLA R4,0 will shift R4 by the contents of the lower four bits of R0. If R0=17, the shift count is one because 17=10001 (binary).

INSTRUCTION: SET ONES CORRESPONDING (LOGICAL OR)

INST FORMAT: SOC S,D

HEX. OPCODE: E000

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Set to logic one all of the bits in the destination operand that correspond to any logic one value in the source operand. This result is placed in the destination. This is effectively a logical OR operation.

INST RESULT: (S) OR (D)--->(D)

APPL. NOTES: Use to perform a logical OR operation. This is similar to ORI except it may be done between two general addresses.

Before: (PATRN1)=>E06B=1110 0000 0110 1101
(PATRN2)=>4482=0100 0100 1000 0010

SOC @PATRN1,PATRN2

After: (PATRN1)=>E06B
(PATRN2)=>E4EF=1110 0100 1110 1111

INSTRUCTION: SET ONES CORRESPONDING BYTE (LOGICAL OR)

INST FORMAT: SOCB S,D

HEX. OPCODE: FOOO

STAT CHANGE: LGT,AGT,EQ,C

DESCRIPTION: Set to a logical one the bits in the destination operand byte that correspond to a logic one in the source operand byte. This is effectively an 8-bit logical OR operation.

INST RESULT: (S) OR (D)--->(D)

APPL. NOTES: Use to perform an 8-bit OR.

SOCB R1,@X (X)=(X) OR R1

INSTRUCTION: SHIFT RIGHT ARITHMETIC

INST FORMAT: SRA W,C

HEX. OPCODE: 0800

STAT CHANGE: LGT,AGT,EQ,C

DESCRIPTION: Shift the contents of the specified workspace register right by the number of places specified by C. The sign bit is extended to fill the vacated bits. If C=0 then the right four bits of workspace register R0 are used for the shift count. The last bit shifted out is placed in the carry bit of the status register.

INST RESULT: (W) shifted right C places--->(W)

APPL. NOTES: Use to shift to the right a signed integer.

SRA R14,5

Shift right the contents of R14 by 5 places. This is a divide by 32.

INSTRUCTION: SHIFT RIGHT CIRCULAR

INST FORMAT: SRC W,C

HEX. OPCODE: 0B00

STAT CHANGE: LGT,AGT,EQ,C

DESCRIPTION: Shift the specified workspace register right by the specified number of places (C), with the bits being shifted out of bit 15 placed in bit 0. If C=0, the right four bits of register R0 are used as the shift count.

INST RESULT: (W) shifted right circ. C places--->(W)

APPL. NOTES: Shift right circular some specified workspace register.

SRC R9,5

INSTRUCTION: SHIFT RIGHT LOGICAL

INST FORMAT: SRL W,C

HEX. OPCODE: 0900

STAT CHANGE: LGT,AGT,EQ,C

DESCRIPTION: Shift the specified work register to the right the specified shift count filling the vacated bits with zeroes. The last bit shifted out is placed in the carry out bit. If C=0, the right four bits of register R0 are used as the shift count.

INST RESULT: (W) shifted right C places--->(W)

APPL. NOTES: Use to shift a workspace register right logical.

```
SRL R10,5    Shift reg R10 right 5 places
SRL R9,1     Effectively divide reg 9 by 2
```

INSTRUCTION: STORE COMMUNICATION REGISTER UNIT (INPUT)

INST FORMAT: STCR S,C

HEX. OPCODE: 3400

STAT CHANGE: LGT,AGT,EQ,OP(<9 bits)

DESCRIPTION: Transfer number of bits specified (C) from the CRU lines addressed by R12 to the source operand. If the number of bits does not fill entire mamory word, then zeroes are added on the left. If C<9 , then S is a byte address.

INST RESULT: CRU lines--->(S) for C bits

APPL. NOTES: Use to store contents of CRU lines in some memory location. The least significant CRU line is transferred to the least significant memory bit.

If C<9 byte addressing
If C>9 word addressing

INSTRUCTION: STORE STATUS REGISTER

INST FORMAT: STST W

HEX. OPCODE: 02C0

STAT CHANGE: None

DESCRIPTION: Transfer the status register to workspace register W.

INST RESULT: Status Register--->(W)

APPL. NOTES: Used to transfer the status register to workspace so it can be manipulated.

```
STST R5      R5=status
```

INSTRUCTION: STORE WORKSPACE POINTER

INST FORMAT: STWP W

HEX. OPCODE: 02A0

STAT CHANGE: None

DESCRIPTION: Transfer the workspace pointer to workspace register W.

INST RESULT: WP--->(W)

APPL. NOTES: Used to determine the address of the register file.

```
STWP R6        R6 = address of R0
```

After execution of the above instruciton, the following two instructions are the same.

```
INC            R0
INC            *R6
```

INSTRUCTION: SWAP BYTES

INST FORMAT: SWPB S

HEX. OPCODE: 06C0

STAT CHANGE: None

DESCRIPTION: Swap the upper byte of the source operand with the lower byte of the source operand.

INST RESULT: Swap (S) upper and (S) lower.

APPL. NOTES: Used for character manipulation.

```
MOVB @C1,R1   R1=character one
SWPB R1       reverse bytes
MOVB @C2,R1   R1=character two,one
```

INSTRUCTION: SET ZEROES CORRESPONDING

INST FORMAT: SZC S,D

HEX. OPCODE: 4000

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Set to a logic zero the bits in the destination operand that correspond to bit positions equal to logic one in the source operand. The source is not changed. Effectively this is a logical AND with the source being inverted prior to the AND.

INST RESULT: NOT (S) AND D--->D

APPL. NOTES: Use to turn off flag bits or AND the contents of one's complement source and destination.

```
Before:  (PAT1)=>3030=0011 0000 0011 0000
         (PAT2)=>5511=0101 0101 0001 0001

SZC @PAT1,@PAT2

After:   (PAT1)=>3030
         (PAT2)=>4501=0100 0101 0000 0001
```

INSTRUCTION: SET ZEROES CORRESPONDING (BYTE)

INST FORMAT: SZCB S,D

HEX. OPCODE: 5000

STAT CHANGE: LGT,AGT,EQ,OP

DESCRIPTION: Set to a logical zero the bits in the destination operand byte that correspond to bit positions equal to a logical one in the source operand byte.

INST RESULT: NOT (S) AND (D)--->(D)

APPL. NOTES: Useful for character or flag manipulation.

```
SZCB @X,@Y    Y=not X and Y
```

INSTRUCTION: TEST BIT

INST FORMAT: TB DISP

HEX. OPCODE: 1F00

STAT CHANGE: EQ

DESCRIPTION: Read the specified CRU input bit whose address is computed by adding the signed displacement to bits 3-14 of R12. Set the equal status register bit to the value read.

INST RESULT: CRU line read--->EQ

APPL. NOTES: Use to read a particular CRU line and depending on the result, make appropriate decisions.

```
CLR R12      set CRU base
TB 14        wait for bit 14 to be set
JNE $-2
```

INSTRUCTION: EXECUTE

INST FORMAT: X S

HEX. OPCODE: 0480

STAT CHANGE: None (remote instruction may, however)

DESCRIPTION: The instruction at the source operand is executed.

INST RESULT: Used to execute an instruction out of line, typically in a table.

```
X @TAB(R1)    execute the instruction in
              table TAB pointed to by R1
```

INSTRUCTION: EXTENDED OPERATION

INST FORMAT: XOP S,N

HEX. OPCODE: 2C00

STAT CHANGE: None

DESCRIPTION: Place extended operation into execution. The (N) field indicates which XOP trap location to utilize.

INST RESULT: S--->(R11) of XOP workspace
(0040+4n)--->(WP)
(0042+4n)--->(PC)
(WP)--->(R13) of XOP workspace
(PC)--->(R14) of XOP workspace
(ST)--->(R15) of XOP workspace

APPL. NOTES: Use to implement software routines which are used frequently. For example: floating point arithmetic, signed multiply, extended precision integer arithmetic. The monitor uses XOP 0 as a breakpoint call. That is, a breakpoint replaces the users instruction by an XOP 0. XOP 1 and XOP 2 are used for input and output. The following will print the letter "A".

```
LETTER      BYTE 'A'
            XOP @LETTER,2
```

INSTRUCTION: EXCLUSIVE OR

INST FORMAT: XOR S,W

HEX. OPCODE: 2800

STAT CHANGE: LGT,AGT,EQ

DESCRIPTION: Perform a bit by bit exclusive OR of the 16-bit source operand with the 16-bit destination workspace register.

INST RESULT: (S) XOR (W)--->(W)

APPL. NOTES: Use to perform an exclusive OR between a workspace register and a source operand.

Assume: (R0)=>21BD = 0010 0001 1011 1101
(TC)=>E436 = 1110 0100 0011 0110

Then: XOR @TC,0

(R0)=>C58B = 1100 0101 1000 1011

INSTRUCTION: EXTERNAL CONTROL

INST FORMAT: CKOF (Clock Off)
CKON (Clock On)
LREX (Load Ram/Execute)
RSET (Reset)

HEX. OPCODE: 03C0
03A0
03E0
0360

DESCRIPTION: These instructions can be decoded by external hardware. The 9900 CPU does not perform any function when they are executed. The T99SS CPU module does not decode these instructions, so they should be avoided.

2.3 Instruction Summary

It is frequently necessary to obtain the hex equivalent or time required for a specific instruction. The 9900's addressing often becomes confusing when trying to do that. To assist the user, the instruction tables are provided. The first gives the hexadecimal op-code and basic execution time; the second defines the additional digits in the opcode for addresssing; and the last one specifies operand address time. For example, if the hex equivalent of MOV *R1,@6(R2) is needed, the following steps are used:

(1) op-code=Cxxx (from Table)
(2) xxx=89s (from Addressing Table)
(3) Thus, instruction=C89s=C891 (s=R1)

The time for the instruction is "14AA" cycles. The two letters after the time are the formula for source address and destination address modification. The last table in this section provides this time. For our example the first operand is *R1 and requires 4 cycles of added time (WR indirect). The second is @6(R2) so it requires 8 cycles more (indexed). Thus the total time is 14+4+8=26 cycles. If two times are shown (e.g. 8/10) then the first is for a jump not taken and the second for a jump that is taken.

Mnemonic	Op-code	Time	Description
A	Axxx	14AA	add Rs to Rd
AB	Bxxx	14BB	add Rs (byte) to Rd (byte)
AI	022s	14--	add constant to Rs
ANDI	024s	14--	AND Rs with Rd
C	8xxx	14AA	compare Rs with Rd
CB	9xxx	14BB	compare Rs (byte) to Rd (byte)
CI	028s	14--	compare constant with Rs
CKOF	03C0	12--	clock-off
CKON	03A0	12--	clock-on
COC	2aaa	14A-	compare (Rd AND Rs) with Rs
CZC	2bbb	14A-	compare (Rd AND Rs) with zero
DIV	3ccc	see note 1	Rd=(Rd,Rd+1)/Rs, Rd+1=rem.
IDLE	0340	12--	idle
JEQ	13yy	8/10	jump if equal
JGT	15yy	8/10	jump if greater than
JH	1Byy	8/10	jump if high
JHE	14yy	8/10	jump if high or equal
JL	1Ayy	8/10	jump if low
JLE	12yy	8/10	jump if low or equal
JLT	11yy	8/10	jump if less than
JMP	10yy	8/10	jump unconditional
JNC	17yy	8/10	jump if no carry
JNE	16yy	8/10	jump if not equal
JNO	19yy	8/10	jump if no overflow
JOC	18yy	8/10	jump if carry set
JOP	1Cyy	8/10	jump if odd parity
LDCR	3aaa	see note 2	d-bits of Rs to CRU
LI	020s	12--	load Rs immediate
LIMI	0300	16--	load interrupt mask immediate
LREX	03E0	12--	load Rom and execute
LWPI	02E0	10--	load WP immediate
MOV	Cxxx	14AA	move Rs to Rd
MOVB	Dxxx	14BB	move Rs (byte) to Rd (byte)
MPY	3ddd	52A-	(Rd,Rd+1)=Rd times Rs
ORI	026s	14--	OR or constant with Rs
RSET	0360	12--	reset
RTWP	0380	14--	return with workspace
S	6xxx	14AA	subtract Rs from Rd
SB	7xxx	14BB	subtract Rs (byte) from Rd (byte)
SBO	1Dyy	12--	set CRU bit yy
SBZ	1Eyy	12--	clear CRU bit yy
SLA	0Ans	see note 3	shift Rs left (alg.) by n
SOC	Exxx	14AA	OR Rs with Rd
SOCB	Fxxx	14BB	OR Rs (byte) to Rd (byte)
SRA	08ns	see note 3	Shift Rs right (alg.) by n
SRC	0Bns	see note 3	Shift Rs right (circ.) by n
SRL	09ns	see note 3	shift Rs right (log.) by n
STCR	3bbb	see note 4	d-bits of CRU to Rs
STST	02Cs	8--	Rs = status register

STWP	02As	8--	Rs = workspace pointer
SZC	4xxx	14AA	RD = Rd AND NOT Rs
SZCB	5xxx	14BB	Rd (byte) = Rd (byte) AND NOT Rs
TB	1Fyy	12--	test CRU bit yy
XOP	2ccc	36A-	extended operation
XOR	2ddd	14A-	ex-OR Rs with Rd

	Rs	*Rs	*Rs+	@Rs		
ABS	074s	075s	077s	076s	12A-(MSB=0) 14A-(MSB=1)	absolute value of Rs
B	----	045s	047s	046s	8A-	branch
BL	----	069s	06Bs	06As	12A-	branch and link R11
BLWP	----	041s	043s	042s	26A-	branch and link WP
CLR	04Cs	04Ds	04Fs	04Es	10A-	clear Rs
DEC	060s	061s	063s	062s	10A-	decrement Rs by one
DECT	064s	065s	067s	066s	10A-	decrement Rs by two
INC	058s	059s	05Bs	05As	10A-	increment Rs by one
INCT	05Cs	05Ds	05Fs	05Es	10A-	increment Rs by two
INV	054s	055s	057s	056s	10A-	invert Rs (ones comp.)
NEG	050s	051s	053s	052s	12A-	negate Rs (twos comp.)
SETO	070s	071s	073s	072s	10A-	set Rs to ones
SWPB	06Cs	06Ds	06Fs	06Es	10A-	swap bytes of Rs
X	048s	049s	04Bs	04As	see note 5	execute inst. at Rs

Note 1: 16 cycles if OV is set. 92 to 124 if OV is not set. Actual time depends upond the partial quotient after each clock cycle during execution.

Note 2: 20+2*number of bits transferred

Note 3: If C not zero, 12+2*number of bits shifted. If C=0 then 20+2*number of bits shifted.

Note 4: Time determined by number of bits as:

1 to 7	42
8	44
9 to 15	58
16	60

Note 5: 8+time for instruction executed

ADDRESSING

	R0	R1	R2	R3	R4	R5	R6	R7		
Rs,Rd	00s	04s	08s	0Cs	10s	14s	18s	1Cs	Rs,Rd	
*Rs,Rd	01s	05s	09s	0Ds	11s	15s	19s	1Ds	*Rs,Rd	
*Rs+,Rd	03s	07s	0Bs	0Fs	13s	17s	1Bs	1Fs	*Rs+,Rd	aaaa
@Rs,Rd	02s	06s	0As	0Es	12s	16s	1As	1Es	@Rs,Rd	
Rs,*Rd	40s	44s	48s	4Cs	50s	54s	58s	5Cs	Rs,Rd	
*Rs,*Rd	41s	45s	49s	4Ds	51s	55s	59s	5Ds	*Rds,Rd	bbbb
*Rs+,Rd	43s	47s	4Bs	4Fs	53s	57s	5Bs	5Fs	*Rs+,Rd	
@Rs,*Rd	42s	46s	4As	4Es	52s	56s	5As	5Es	@Rs,Rd	
Rs,*Rd+	C0s	C4s	C8s	CCs	D0s	D4s	D8s	DCs	Rs,Rd	
*Rs,*Rd+	C1s	C5s	C9s	CDs	D1s	D5s	D9s	DDs	*Rs,Rd	cccc
*Rs+,*Rd+	C3s	C7s	CBs	CFs	D3s	D7s	DBs	DFs	*Rs+,Rd	
@Rs,*Rd+	C2s	C6s	CAs	CEs	D2s	D6s	DAs	DEs	@Rs,Rd	
Rs,@Rd	80s	84s	88s	8Cs	90s	94s	98s	9Cs	Rs,Rd	
*Rs,@Rd	81s	85s	89s	8Ds	91s	95s	99s	9Ds	*Rs,Rd	dddd
*Rs+,@Rd	83s	87s	8Bs	8Fs	93s	97s	9Bs	9Fs	*Rs+,Rd	
@Rs,@Rd	82s	86s	8As	8Es	92s	96s	9As	9Es	@Rs,Rd	

	R8	R9	R10	R11	R1 2	R13	R14	R15		
Rs,Rd	20s	24s	28s	2Cs	30s	34s	38s	3Cs	Rs,Rd	
*Rs,Rd	21s	25s	29s	2Ds	31s	35s	39s	3Ds	*Rs,Rd	aaaa
*Rs+,Rd	23s	27s	2Bs	2Fs	33a	37s	3Bs	3Fs	*Rs+,Rd	
@Rs,Rd	22s	26s	2As	2Es	32s	36s	3As	3Es	@Rs,Rd	
Rs,*Rd	60s	64s	68s	6Cs	70S	74s	78s	7Cs	Rs,Rd	
*Rs,*Rd	61s	65s	69s	6Ds	71s	75s	79s	7Ds	*Rs,Rd	bbbbb
*Rs+,*Rd	63s	67s	6Bs	6Fs	73s	77s	7Bs	7Fs	*Rs+,Rd	
@Rs,*Rd	62s	66s	6As	6Es	72s	76s	7As	7Es	@Rs,Rd	
Rs,*Rd+	E0s	E4s	E8s	ECs	F0s	F4s	F8s	FCs	Rs,Rd	
*Rs,*Rd+	E1s	E5s	E9s	EDs	F1s	F5s	F9s	FDs	*Rs,Rd	cccc
*Rs+,*Rd+	E3s	E7s	EBs	EFs	F3s	F7s	FBs	FFs	*Rs+,Rd	
@Rs,*Rd+	E2s	E6s	EAs	EEs	F2s	F6s	FAs	FEs	@Rs,Rd	
Rs,@Rd	A0s	A4s	A8s	ACs	B0s	B4s	B8s	BCs	Rs,Rd	
*Rs,@Rd	A1s	A5s	A9s	ADs	B1s	B5s	B9s	BDs	*Rs,Rd	dddd
*Rs+,@Rd	A3s	A7s	ABs	AFs	B3s	B7s	BBs	BFs	*Rs+,Rd	
@Rs,@Rd	A2s	A6s	AAs	AEs	B2s	B6s	BAs	BEs	@Rs,Rd	

ADDRESS MODIFICATION TIME

Mode	Time(A)	Time(B)
Register	0	0
Register Indirect	4	4
Register Indirect with increment	8	6
indexed	8	8