2920 ASSEMBLY LANGUAGE MANUAL

Manual Order Number 9800987-01

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

i	iSBC	Multimodule
ICE	Library Manager	PROMPT
iCS	MCS	Promware
Insite	Megachassis	RMX
Intel	Micromap	UPI
Intelevision	Multibus	μScope
Intellec		

and the combination of ICE, iCS, iSBC, MCS, or RMX and a numerical suffix.

A101/0879/10K FL

The assembler translates 2920 mnemonics (such as ADD or L02 or IN3) into machine code. At the same time, it produces an object file and a listing file showing each source line and the generated code, plus any detected errors. Its object output may be used by the PROM programmers and the 2920 Simulator.

The minimum hardware configuration to run the AS2920 Assembler is as follows:

- INTELLEC or INTELLEC-II with 32K random access memory (RAM)
- Teletypewriter, CRT, or equivalent for console input and output
- Single diskette drive unit

If a line printer is available, it can be used for large-volume or hard-copy output.

The AS2920 Assembler uses the ISIS-II keyboard and disk input/ouput functions. You may wish to refer to other documents containing valuable information about this supervisor, the Intellec system and the other software used on it. These include:

Intellec Operator's Manual	9800129
INTELLEC/DOS Diskette Operating System Operator's Manual	9800206
ISIS-II System User's Guide	9800306

Reader's Guide

The tutorial and reference formats necessarily require repetitive information. In reading this manual for the first time, you might first skim Chapters 1 and 4. They will familiarize you with the general setup of the language and the simpler applications of general interest. When you begin to have questions about particular instructions, you can look them up in Chapter 3.

Chapter 2 gives the detailed rules of this assembly language. Chapter 5 discusses the controls you may use either when you invoke the Assembler or on control lines embedded in your source file. The Appendices discuss other specialized issues around the language or the part.

CONTENTS

CONTENTS (Continued)

CHAPTER 1
INTRODUCTION

Capabilities, Features, and Functional Elements

The 2920 can perform a variety of analog signal processing functions by using digital techniques. Such 2920 programs may realize, by simulation, a collection of major analog modules and their interconnections. A program is executed repeatedly and continuously, thereby imitating the behavior of the analog system on a sampled basis.

Examples of such functions include simple to complex filters, oscillators, limiters, rectifiers, modulators, non-linear functions, correlations, and logical operations, among many others. Several such functions can be achieved using a single 2920 chip, and if additional capability is required for extremely complex designs, additional 2920s can be cascaded together.

The major elements of the chip include:

1. an arithmetic processor
2. a scratch-pad random-access memory (RAM)
3. a digital-to-analog and analog/digital conversion unit (DAC)
4. a user-programmable and erasable PROM

The chip is realized in an n-channel MOS technology, and is supplied in an 28-lead dual-inline package.

The analog/digital conversion system provides four analog inputs and eight analog outputs, with built-in sample-and-hold. Inputs and outputs may be used for logic levels if desired.

Basic 2920 Operation

The 2920 implements the analog functions by simulating them in real time. Digital processing guarantees stability, accuracy, and reproducibility of results. Amplitude stability is determined by that of an external voltage reference, and frequency stability is determined by that of the clock used to operate the device.

Clocks

Clock sources for the 2920 may be derived from an externally generated pulse train, or by connecting an external crystal to the device. The crystal or clock frequency is four times the instruction rate. An instruction rate clock is provided by the 2920 as a TTL output. Other TTL inputs and outputs are available to you for greater flexibility in applications using the 2920.

Minimum Hardware Requirements

In simple applications you may need only to provide power supplies (including the reference voltage), a crystal, and a sample capacitor. Some applications may require simple analog anti-aliasing filters at the inputs and/or outputs.

Overview of Functional Elements

Figure 1-1 shows a block diagram of the 2920. In the figure, the 2920 has been divided into three major sections: the PROM, the arithmetic unit with memory, and an analog section.

The PROM section of the 2920 includes an instruction clock generator and program sequence counter. Signals from the clock generator and PROM control the other two sections.

The arithmetic section includes a 40 word by 25-bit random access memory (RAM) with two ports, and an arithmetic and logic unit (ALU). One of the two inputs to the ALU is passed through a scaler or barrel shifter. The arithmetic section executes commands from PROM, thereby performing digital simulation of analog functions in real time.

The analog section performs analog to digital (A/D) and digital to analog (D/A) conversions upon commands from the PROM section. The analog section includes:

- an input multiplexer (4 inputs),
- an input sample-and-hold circuit,
- a digital to analog converter (DAC),
- a comparator, and
- an output multiplexer with 8 output sample-and-hold circuits.

The analog section also has a special register called the DAR (for digital/analog register), which acts as a link between the arithmetic and analog sections.

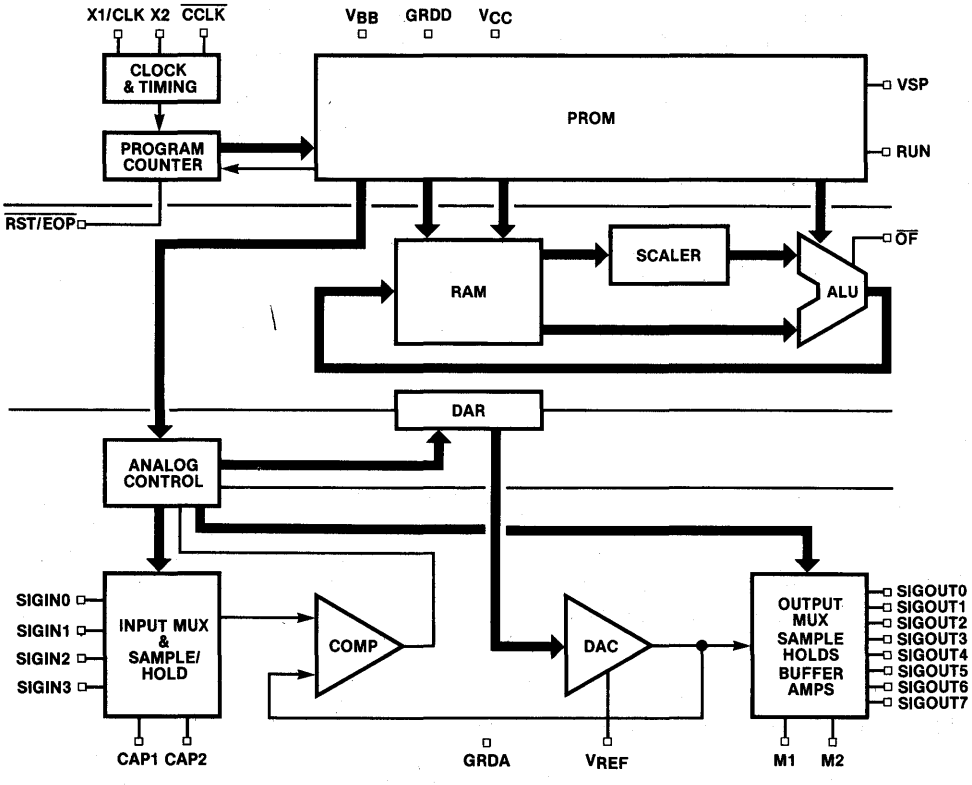


Figure 1-1. Block Diagram of 2920 Signal Processor

Figure 1-2 shows the labels used for pinouts while running a program in the 2920. Several pins perform different functions during programming than during normal operation.

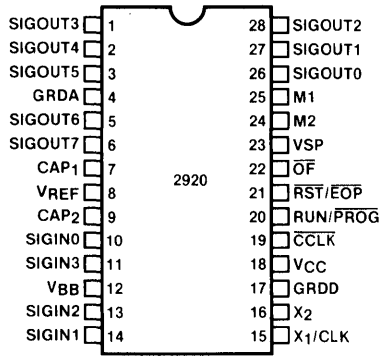


Figure 1-2. Run Mode Pin Configuration

The numeric conventions used in the 2920 are covered in more detail in Appendices I and J.

Basic 2920 Performance Parameters and Limits

The limits to 2920 capabilities are established by the size of the on-chip PROM and RAM, the speed and capability of the processor, and the resolution of the A/D and D/A converters.

A program for the 2920 consists of a series of basic 2920 instructions which are executed sequentially at a fixed rate. The program allows no internal jumps, and is therefore of fixed length and execution time.

A sample interval is the time between samples of the same input channel. Normally, one pass through a program establishes a sample interval, i.e. input/output operations usually take place once per program pass. Similarly, the functions implemented by the sequence of instructions usually occur once per sample interval. However, the signal on a given input channel may be sampled more than once during a single pass through the program. In this case, sample interval is determined by multiplying the instruction-clock-period by the number of instructions between input samples.

The number of functions which can be realized with a single 2920 is established by the amount of PROM provided and the number of RAM words on the chip. For example, a typical digital filter requires at least one RAM word per pole or two per complex conjugate pole pair. Thus the RAM limits the number of poles to less than 40, or less than 20 complex conjugate pairs. The number of PROM words needed to realize a complex conjugate pole pair is variable, but has a typical value of approximately 10. Therefore, PROM capacity also limits the number of conjugate pole pairs to less than 20.

What is a Program? An Assembler? Assembly Language?

A program is a sequence of instructions intended for execution by a particular computer. In the beginning, computers speak only binary, that is, they obey instructions coded as sequences of ones and zeros like 111 000000 000100 1101 00001. Humans tend to find such "sentences" difficult to deal with and error-prone. We usually prefer symbols more familiar, or at least easier to remember (more mnemonic).

An assembler is a program usually supplied by a manufacturer to create the needed strings of ones and zeros from "words" you write in the special (assembly) language. Such a program bridges the gap by translating the more human-like languages into the computer's binary tongue. For example

```
LDA ZETA,GAMMA,L01,IN1
```

is an instruction you would code in the 2920 Assembly Language to produce the binary string given above. Each separate abbreviation in the instruction is called a "mnemonic" because it is easier to remember than its binary meaning. This instruction could be read as saying

> "take whatever number is in the location named GAMMA, shift it left one bit position, and then load the location named ZETA with the resulting number. While you're at it, sample input-channel-1."

If the location named GAMMA contains the number 1/4, the location named ZETA will be filled with the number 1/2. Shifting a binary number left one binary position effectively multiplies by 2, just as shifting a decimal number left one decimal position effectively multiplies by ten (decimal 1234 shifted one position becomes 12340). This will all be covered in more detail later.

Thus most programs are written in a language different from the codes used directly by the computer. Such a language requires translation into those codes. The translator for an assembly language is called an assembler.

A program that needs translation before the computer can execute it is called a source program. The assembler performs various functions on the source program to create an "object program," the thing that the computer actually executes. The assembler also creates several reports enabling you, as programmer, to evaluate, modify, or use this object program.

Overview of 2920 Assembly Language Keywords

The assembly language for the 2920 is simple. Table 1-1 lists the symbols used in its instructions. Naturally, a detailed discussion of the assembly language is the major content of this whole manual. The next few paragraphs are merely a quick, condensed overview, which may facilitate your assimilation of the details given in later sections.

Table 1-1. 2920 Keywords

Section 1: Arithmetic Operation Codes

ABA ABS ADD AND LDA LIM SUB XOR

Section 2: Analog Control Codes

CNDS	CVTS	EOP	IN0	NOP	OUT0
CND7	CVT7		IN1		OUT1
CND6	CVT6		IN2		OUT2
CND5	CVT5		IN3		OUT3
CND4	CVT4				OUT4
CND3	CVT3				OUT5
CND2	CVT2				OUT6
CND1	CVT1				OUT7
CND0	CVT0				

Section 3: Constant Source Codes and the DAR

	KM1	KM2	KM3	KM4	KM5	KM6	KM7	KM8
KP0	KP1	KP2	KP3	KP4	KP5	KP6	KP7	
DAR								

Section 4: Scaler Control Codes

R00 R01 R02 R03 R04 R05 R06 R07 R08 R09 R10 R11 R12 R13
R0 R1 R2 R3 R4 R5 R6 R7 R8 R9

L01 L02
L1 L2

Section 5: Assembler Commands and Modifiers

DEBUG	END	OBJECT( )	PAGING	SYMBOLS
NODEBUG	EQU	NOOBJECT	NOPAGING	NOSYMBOLS
PAGELENGTH( )		PAGEWIDTH( )	PRINT	PRINT( )
			NOPRINT	
EJECT	LIST	TITLE(' ... ')		
	NOLIST			

As shown in Section 1 of Table 1-1, there are eight basic arithmetic and logic commands. Section 2 shows the analog codes. "IN" is used for input, "OUT" for output. "CVT" stands for converting a sampled analog input into a digital number, one bit at a time. "CND" means conditional use of an arithmetic instruction, depending on testing a specified bit. The 16 built-in constants shown in Section 3 run from −8/8 (KM8) to +7/8 (KP7).

There are 16 separate shift codes shown in Section 4, which permit scaling (multiplying) by powers of 2. The codes run from R13, meaning 2^{-13}, up to L02, meaning 2^{2}. The R and L stand for shifts to the right and left, respectively.

You would use the additional keywords named in Section 5 of the Table to direct certain assembler operations, or to qualify certain of your instructions. Every symbol in Table 1-1 is explained in this manual.

The portion of a 2920 program shown below illustrates where certain types of symbols typically appear in the fields of an instruction:

```
.
.
SUB OSC,KP4,R12,IN3          ; OSC exemplifies a
ADD DAR,OSC,R00,IN3          ; user symbol for some
ADD OSC,KP4,L01,CNDS         ; variable
.
.
.
```

A Closer Look at the 2920 Language

The Assembly Language must be able to generate all the possible legal binary strings, but still be easier to use than binary itself. To this end, instructions are created out of a limited number of predefined symbols, strung together in a predefined structure (instruction). User-defined symbols can be used to name locations containing variables (data, in RAM) or instructions (in ROM).

Each part of an instruction is called a field, and only certain symbols are permitted in specific fields. Fields are separated by something called a delimiter, such as a sequence of one or more blanks, a comma, a semicolon, or a colon.

A field can be (for example) an indicated operation like ADD or an operand like XX or YY in the instruction

```
ADD  XX,YY
```

which would add the contents of the location named YY into the contents of the location named XX.

Opcodes, Labels, and Operands

A 2920 instruction can have up to seven fields. They are called, from left to right: label, opcode, destination operand, source operand, shiftcode, iocode, and comment fields.

ADD is an example of what is allowed in the opcode field. The field containing XX is called the destination field; that containing YY is called the source field. There is an optional label field, giving a name to the location of the instruction itself. If used, a label always appears as the first field on the left, immediately followed by a colon.

Thus, if you wanted to code an add instruction to put the sum of VARIABLE11 and VARIABLE22 into VARIABLE22, and name that instruction SUM, it would look like this:

```
SUM:    ADD VARIABLE22,VARIABLE11
```

This instruction has four fields: the label, opcode, destination, and source fields. Note that since the 2920 instruction set contains no branch instructions, the label SUM is only useful in debugging, when it can be accessed by the Simulator.

Shift Codes

The two remaining fields, before an optional comment, are called the shiftcode and the iocode. If you omit coding them explicitly, the assembler supplies a default automatically, as explained below. The shiftcode affects the source operand only, shifting it left or right so many bits (or none) before the opcode is performed. As mentioned above, this "scaling" has the effect of multiplying the source operand (only) by 2 to the power of the shiftcode. Thus, to add half of VARIABLE11 into VARIABLE22, you could write

```
SUM:    ADD VARIABLE22,VARIABLE11,R01
```

Iocodes

The optional iocode can direct the 2920 to perform one of four functions:

1. sample an input channel
2. put a sample out to an output channel
3. convert from the input sample into the DAR
4. make the execution of an arithmetic opcode execution conditional on some specified bit of the DAR.

These iocode operations occur simultaneously with the arithmetic operations, such as ADD, because they use a different part of the 2920. Arithmetic instructions use the ALU, arithmetic logic unit. Iocode operations use the analog control decoder (see Figure 1.1).

Here are some examples of the four iocode possibilities:

1. To sample input-channel-3 at the same time as doing the ADD above, you simply code its name into the iocode field, as follows:

   ```
   SUM:    ADD VARIABLE22,VARIABLE11,R01,IN3
   ```

   If no arithmetic operation is currently needed or planned to go on simultaneously, you could simply code

   ```
   IN3
   ```

   and the assembler would automatically fill in the other fields with

   ```
   LDA 0,0,R00
   ```

   which is the ALU no-operation code (which also clears the carry flag), thus creating the instruction

   ```
   LDA 0,0,R00,IN3
   ```

   which does nothing but sample input-channel-3 (and clear the carry).

2. To output to channel 5, you could append OUT5 to an arithmetic instruction, e.g.

   ```
   SUM:    ADD VARIABLE22,VARIABLE11,R01,OUT5
   ```

   or code it alone

   ```
   OUT5
   ```

   with the result in either case that the contents of the DAR are converted to an output voltage applied to channel 5. (As in 1 above, the latter form will also clear the carry due to the use of the default LDA instruction.)

3. Analog to digital conversions of input samples are performed one bit at a time into the DAR using the 9 CVT codes:

```
LDA     DAR,KP0     ; initialize DAR to zero.
CVTS                ; convert sign bit
CVT7                ; convert bit 7 (most-significant-bit)
CVT6                ; convert bit 6
CVT5
CVT4
CVT3
CVT2
CVT1
CVT0                ; convert bit 0 (least-significant-bit)
```

In actual practice, each cycle of conversion must allow time for the DAC to settle, which is achieved by inserting NOP (no-operation) iocodes after all but the last CVT (or by putting the CVT iocodes only on every other sequential instruction).

```
LDA     DAR,KP0     ; initialize DAR to zero.
CVTS
NOP
CVT7
NOP
CVT6
NOP
CVT5
NOP
CVT4
NOP
CVT3
NOP
CVT2
NOP
CVT1
NOP
CVT0
```

This sequence could also be used as the iocodes on any 18 arithmetic instructions in order, providing the DAR was not altered by any of them. (Since this sequence changes the DAR, you should also not be relying on it to retain any value from prior work.) The result will be a digital value in the DAR represent ing the amplitude of the input signal as a fraction of the reference voltage.

4. The conditional iocode, CND, uses the result of testing a specified bit of the DAR to allow, cancel or change the operation of its associated arithmetic instruction (LDA, ADD, or SUB). The bit is specified by the fourth character of the iocode, as it was in the CVT iocodes above (0 is least significant bit):

```
SUM:    ADD VARIABLE22,VARIABLE11,R01,CNDS
```

In this example, if the sign bit of the DAR is 0, the add will not occur; if the sign bit of the DAR is 1, the add will occur. This is similarly true if the opcode were LDA:

```
LDA VARIABLE22,VARIABLE11,R01,CNDS
```

This will fill variable22 with half the contents of variable11 only if the sign bit of the DAR is 1. If it is 0, the operation does not take place (though it does take time).

The case of SUB is slightly more complicated and will be dealt with in Chapter 3.

Basic Program Flow

Figure 1-3a shows the flow diagram for a typical 2920 application. The program has no beginning or end, but repeatedly performs the same set of calculations at a constant rate.

In Figure 1-3a, the input/calculate/output sequence is shown in series. However, there may be some parallelism in actual programs because the input/output operations may execute in parallel with some arithmetic. As an example of an extreme case of parallelism, consider the 2920 programmed to perform two independent operations, each of which takes less than half of the 2920 capacity. The flow chart can be as shown in Figure 1-3b. In actuality, the two loops execute simultaneously and in synchronism, with the I/O functions of one loop being executed while the second is performing computation.

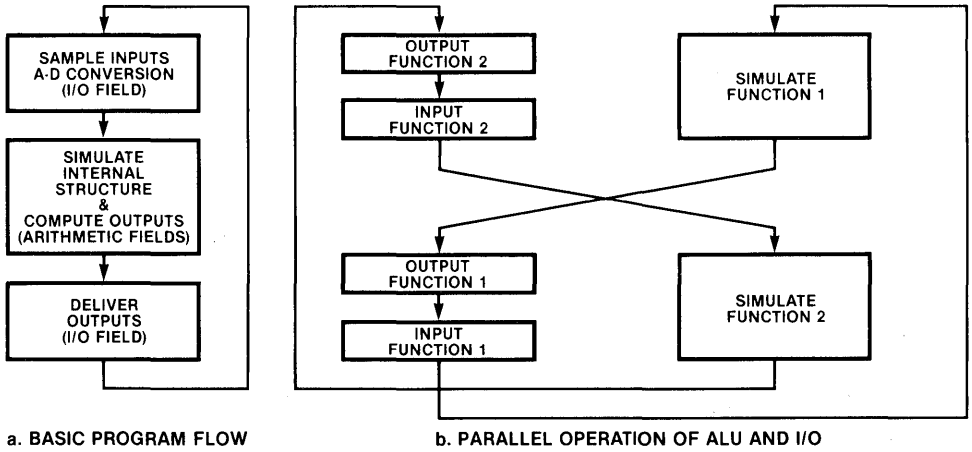


Figure 1-3. 2920 Program Flow Diagrams

Assembler Functions and Controls

The assembler translates 2920 mnemonics (such as ADD or L02 or IN3) into machine code. At the same time, it produces an object file and a listing file showing each source line and the generated code, plus any detected errors. Its object output may be used by the PROM programmers and the 2920 Simulator.

You can control the operations of the assembler with respect to most of its functions. This control may be exercised in the command invoking the assembler, or in control lines embedded in the source program.

The individual functions of this assembler are:

1. Symbol Table Management: keeping track of all symbols and their values and automatically assigning RAM locations to variable names as they are encountered.
2. Location Counter Management: keeping track of locations available for instructions and assigning locations for each instruction assembled.
3. Instruction Assembly: translating mnemonic opcodes and operands into their machine language equivalents.

4. Control and Directive Processing: noting and executing all controls, e.g. assembly listing and object output control, and directives such as symbol definition. This includes controls given as part of the invocation command.
5. Assembler Output Generation: creating the assembly listing, object code file, and error diagnostics.

The complete set of controls is given in Chapter 5. You may specify whether or not you want an output listing printed, the length and width of listing pages, the desired title on it, whether you want an object file generated, and whether or not you want symbol table and/or debugging information (for use by the Simulator) generated with it.

If you do not specify any options, then certain default assumptions are used, as follows:

> that you DO want a listing, sent to a file with the same basic name as your source file (called, say, PROG.SRC) but with "LST" as the qualifying "extent", e.g. PROG.LST;
>
> that the listing should be divided into numbered pages of 66 lines, each up to 120 characters long;
>
> that you do want an object program created, and sent to a file named like the source but with a qualifying extent of ".HEX", e.g. PROG.HEX ;
>
> and that the symbol table should be put out to the listing but not to the object file.

These defaults correspond in order to the following option choices:

```
LIST
PRINT(filename.LST)
PAGING
PAGELENGTH(66)
PAGEWIDTH(120)
OBJECT(filename.HEX)
SYMBOLS
NODEBUG
```

A Closer Look at the Functional Elements

PROM Section

The PROM Section contains 4608 bits of user programmable and erasable read-only memory. In normal operation of the 2920, i.e., in RUN mode, it is arranged as 192 words of 24 bits each. Each word corresponds to one 2920 instruction. (During programming, each 24-bit word is treated as six 4-bit nibbles; i.e., in PROGRAM mode the PROM appears as 1152 words of four bits each. Each six nibbles appear on a separate line of the hexadecimal object-file-listing as shown in Appendix E.)

PROM RUN Mode

During RUN mode the PROM section acts as the system controller. Each 24-bit control word contains bit patterns that determine the operations to be performed by the analog and arithmetic sections.

Ignoring labels and comments, the control word in RUN mode can be viewed as five fields, of which one controls the analog section and the remaining four control the arithmetic section. The four arithmetic section control fields include the two 6-bit fields which identify RAM operands, plus a 4-bit scaler control field and a 3-bit ALU control field (operation code or opcode).

In RUN mode, PROM word addresses are numbered from 0 to 191. In normal operation all locations are accessed in sequence and no program jumps are allowed. The PROM returns to location 0 upon completion of execution of the command in word 191, or when an EOP instruction is encountered in the analog control instruction field. The EOP feature allows the program to be terminated at the end of a user's program shorter than 192 words. Placement of the EOP is explained below.

The PROM may be thought of as a crystal- or clock-controlled cycle generator as it determines the sampling frequency of the analog signals. If an input is sampled once per program pass, the sampling frequency is 1/NT where N is the number of words (instructions) in the program and T is the time required to execute one instruction.

The PROM fetch/execute cycle is pipelined four deep, meaning that the next four instructions are being fetched while the previously fetched instructions are being executed. Although otherwise invisible to the user, this technique makes it necessary to require that the EOP instruction be inserted in a word with an address divisible by four, e.g. 0, 4,....,188. The EOP does not take effect until the three following instructions are executed because those three are already fetched.

Arithmetic Unit and Memory

A block diagram of this subsystem is shown in Figure 1-4. This subsystem consists of three major elements: a RAM storage array, a scaler, and an arithmetic and logic unit (ALU).

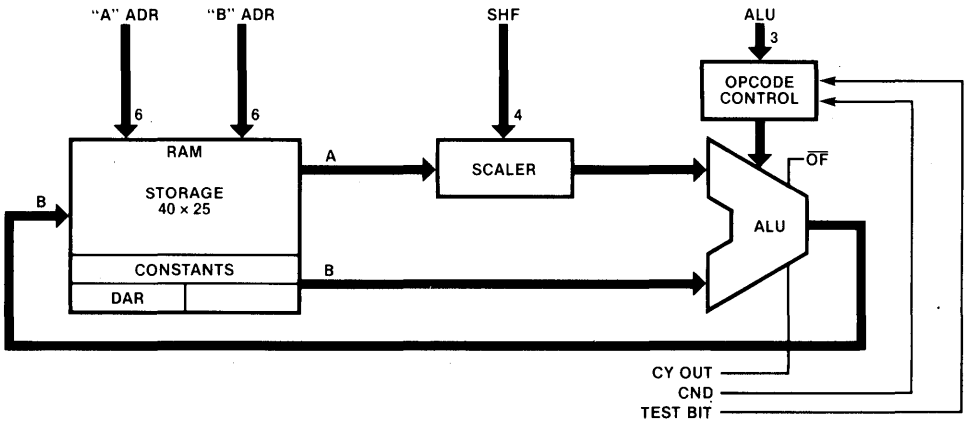


Figure 1-4. ALU Block Diagram

Data within this structure are processed using 25-bit two's complement arithmetic, although at certain locations larger or smaller words may be found. (Two's complement representation of data is explained in Appendix I.) It is most convenient to consider an imaginary binary point just to the right of the highest order (i.e. sign) bit of each word. Thus the normal range of any variable x is considered to be

$$-1.0 \leq x < 1.0$$

and the smallest resolvable change, delta, in any variable is given by

$$\text{delta} = 2^{-24} = 5.96 * 10^{-8}$$

Each of the elements making up this portion of the 2920 receives command or address information from the PROM. The storage array receives two 6-bit address fields, the scaler receives a 4-bit control field, and the ALU receives a 3-bit control field.

The Storage Array and the Constant Array

The storage array consists of a random access memory, with two ports, organized as 40 words of 25 bits each. Each port is independently controlled from the PROM by a 6-bit control field.

These 6 bits enable operand addresses from 0 up to 63, since 2 to the 6th allows for 64 possible addresses. However, only 40 of these refer to actual RAM locations. The remaining possibilities are used to refer to an array of constants, and to an input/output register (the DAR) which serves to link the arithmetic and analog/digital conversion sections of the 2920.

The two ports are called "A" and "B." The A port is read-only. Data read from it are passed, through the scaler, to one input of the ALU, as the source operand. The B port passes data to the second ALU input, and receives the ALU results, as the destination operand.

The constant array consists of 16 "pseudo-locations" in the RAM address field. These constants should be accessed only from the A port, i.e., only as a source operand. A warning is issued if they are referenced as destination operands, and they remain unaffected. The least significant four bits of the "address" are directly translated to the high-order four bits of the data field, with the remaining data bits equivalent to zeroes. Consequently, each unscaled constant is some number of eighths, from −8/8, −7/8,..., up to +6/8, +7/8. A much wider range of constants is actually available, because the selected constant passes through the scaler, and can thus be modified as explained below. Figure 1-5 shows the "address" mapping for constants.

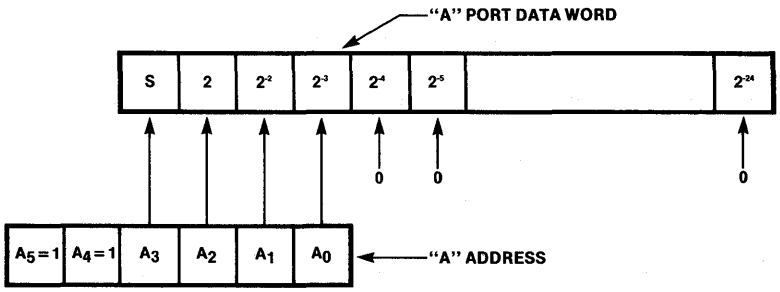


Figure 1-5. Address Mapping for Constants

Table 1-2. Constant Codes

CONSTANT MNEMONIC	UNSCALED VALUE	CONSTANT MNEMONIC	UNSCALED VALUE
KPO	0	KM1	− .125
KP1	+.125	KM2	− .250
KP2	+.250	KM3	− .375
KP3	+.375	KM4	− .500
KP4	+.500	KM5	− .625
KP5	+.625	KM6	− .750
KP6	+.750	KM7	− .875
KP7	+.875	KM8	−1.0

The DAR can be used as a source or a destination operand. It is a digital to analog register and an analog to digital register. It is nine bits wide, occupying the nine most significant bit positions of a word whose other bits are set to ones in order to correct for A/D conversion offset.

The DAR output is also tied directly to the digital to analog converter (DAC) inputs. The DAR is used as a successive-approximation register for analog to digital conversion, under control of the analog function instruction fields (CVT iocodes) explained above and in later chapters. Each bit position of the DAR can also be tested by the ALU for conditional arithmetic operations using the CND iocodes.

Scaler

The scaler is an arithmetic barrel shifter located between the A port of the RAM and the ALU. Values read from the A port can be shifted left or right. The shifts can be a maximum of two positions to the left and a maximum of thirteen positions right. Left shifts fill with zeroes at the right; right shifts fill with the sign bit at the left.

Table 1-3. Scaler Codes and Operations

SCALER CODE	BIT VALUES	EQUIVALENT MULTIPLIER	OPERATION	SCALER CODE	BIT VALUES	EQUIVALENT MULTIPLIER
L02	1110	$2^2 = 4.0$	"A"x2^2	R06	0101	$2^{-6} = 0.015625$
L01	1101	$2^1 = 2.0$	"A"x2^1	R07	0110	$2^{-7} = 0.0078125$
R00	1111	$2^0 = 1.0$	"A"x2^0	R08	0111	$2^{-8} = 0.00390625$
R01	0000	$2^{-1} = 0.5$	"A"x2^{-1}	R09	1000	$2^{-9} = 0.001953125$
R02	0001	$2^{-2} = 0.25$	.	R10	1001	$2^{-10} = 0.0009765625$
R03	0010	$2^{-3} = 0.125$	.	R11	1010	$2^{-11} = 0.00048828125$
R04	0011	$2^{-4} = 0.0625$	"A"x2^{-4}	R12	1011	$2^{-12} = 0.000244140625$
R05	0100	$2^{-5} = 0.03125$	.	R13	1100	$2^{-13} = 0.0001220703125$

As explained above, these arithmetic shifts are equivalent to multiplication of the A port value by a power of two, where the number of positions shifted is the power.

The scaler is controlled by a 4-bit wide control field from the PROM, as shown in Table 1-3. Note that left shifts may produce numbers which are too large to fit within a 25-bit field. The handling of such large numbers is described in the ALU section below.

The ALU

The Arithmetic Logic Unit calculates a 25-bit result from its A and B operands (source and destination) based on an operation code from the PROM. The 25-bit result is written back into the B (destination) memory location at the end of the instruction cycle.

The ALU uses extended precision to allow calculation of the correct result even when receiving left-shifted operands from the scaler. If the computed result YY exceeds the bounds

$$-1.0 \leq YY < 1.0$$

an overflow condition is indicated. When overflow limiting is enabled, this condition causes the result to be replaced with the legal value closest to the desired result, i.e. with -1 if the computed value was negative, and with $+1.0$ if the result was positive.

In binary these extreme values appear as

1000	000	000	000	000	000	000	000 and
0111	111	111	111	111	111	111	111 (=1.0*, or $1-2^{-24}$)

respectively. This overflow situation characteristic is useful for realizing certain nonlinear functions such as limiters, and is beneficial to the stability of filters. The OF pin tells you that an overflow is occurring on the current operation (cycle). This output is active low and open-drain. In the case where overflow is not enabled, each binary number is extended to 28-bit precision by extending the sign bit to the left. The calculation is done and the low 25 bits are written back to the destination.

The operations performed by the ALU are summarized in Table 1-4. Although most of them are self-explanatory, you may find the following details useful at this point.

Absolute value (ABS) and absolute add (ABA) convert the "A" operand (source) to its absolute value before performing any calculations. Load A (LDA) and ABS are treated as arithmetic operations by the ALU, meaning that the source is added to zero and then replaces the "B" operand (destination). This causes the correct handling of those overflows caused by left shift operations.

The operation LIM sets the result to positive or negative full scale, based on the "A" port sign bit, behaving much like a forced overflow. However, the overflow flag will not indicate overflow for a LIM unless the given source operand and shift-code would produce an overflow in an LDA operation.

The constant source codes allow you to select constants for arithmetic operations. The procedure is described in the section on the storage array of the ALU and memory. Table 1-2 above lists the mnemonics and corresponding unscaled value of each constant. Each value is passed through the scaler, and so may be multiplied by a value 2^k, where k runs from $+2$ to -13. The scaler codes and equivalent multiplier values are shown in Table 1-3.

Conditional Arithmetic Operations

In addition to the basic operations described in Table 1-4, some ALU functions may execute conditionally. Certain codes in the analog/digital control field (iocode) cause the execution of the arithmetic operation to be conditional on a selected bit, usually of the DAR. The conditional instructions are tabulated in Table 1-4b.

As discussed above under IOCODES for ADD and LDA, the conditional field code selects a bit of the DAR, using its value to determine how the instruction is to be executed. For conditional subtract, the bit actually used is the carry from the previous result. In this case the selected bit of the DAR is set equal to the carry from the current instruction. This is discussed further in Chapter 4 and Appendices I and J.

Conditional additions are used to multiply one variable by a second, as discussed in Chapter 4. The multiplier is loaded into the DAR, and the multiplicand is added conditionally to the partial product.

Conditional subtraction is used to divide one positive variable by another, using a non-restoring division algorithm. The divisor is conditionally subtracted from the dividend, and quotient bits are assembled in the DAR.

Conditional operations may also be useful for performing logic, also shown in Chapter 4. Table 1-5 summarizes the properties of the arithmetic section.

Table 1-4. Memory—ALU Instruction Opcodes

a. Non-Conditional Arithmetic

ALU	MNEM	OPERATION*	DESCRIPTION/COMMENTS
2 1 0			
0 0 0	XOR	$B+(A\cdot 2^k)\rightarrow B$	
0 0 1	AND	$B\wedge(A\cdot 2^k)\rightarrow B$	
0 1 0	LIM	$+1^{**}\rightarrow B$ if $A\geq 0$	Sign of A saturates output
		$-1\rightarrow B$ if $A<0$	
0 1 1	ABS	$0+\lvert A\cdot 2^k\rvert\rightarrow B$	Absolute Value
1 0 0	ABA	$B+\lvert A\cdot 2^k\rvert\rightarrow B$	Absolute Value and Add
1 0 1	SUB	$B- A\cdot 2^k\rightarrow B$	
1 1 0	ADD	$B+ A\cdot 2^k\rightarrow B$	
1 1 1	LDA	$0+ A\cdot 2^k\rightarrow B$	

*Note—k is the value selected by the shift code, $-13\leq k\leq +2$
**Note—the largest positive value ($1-2^{-24}$) is stored.

b. Conditional Arithmetic Operations

ALU Functions made Conditional by selected codes in the Analog Control Field.

ALU FUNCTION	BIT TESTED	IF TESTED BIT = 0	IF TESTED BIT = 1
ADD (110)	DAR (n)	NO-OP($B+0\rightarrow B$)	ADD ($B+A\cdot 2^k\rightarrow B$)
LDA (111)	DAR (n)	NO-OP($B+0\rightarrow B$)	LDA ($0+A\cdot 2^k\rightarrow B$)
SUB (101)	PREV cy	ADD($B+A\cdot 2^k\rightarrow B$)	SUB ($B-A\cdot 2^k\rightarrow B$)
		cy → DAR(n)	cy → DAR(n)

Note—DAR(n) represents a bit of the DAR, as selected by the conditional operand in the analog control field. For ADD and LDA, the selected bit is tested. For SUB, the selected bit is altered by being set to the carry output of the highest order position of the ALU; and the conditional operation is based on a test of the carry resulting from the previous ALU operation.

Table 1-5. Memory—ALU Section Summary

ALU result bit width	25 bits
Number System	2's complement
Operand A	Read Only Memory Port A, Scaled by shifter, 28 bits wide Read Port B, Unscaled, 25 bits expanded to 28 bit equivalent
ALU instruction field width	3 bits
Scaler instruction field width	4 bits
"A" and "B" port address field width	6 bits each
Ancillary Instructions	Conditional arithmetic, op codes are part of analog control field
Available Storage Locations	"A" port, Adr0-39, Read Only, 25 bits. "B" port, Adr0-39, Read-Write 25 bits.
Digital-Analog-Register	"A" port, Adr40, Read Only, 9 MSBs, 16 LSB's are extended sign. "B" port, same as "A" port but read-write.
Constant Register	"A" port only, low 4 bits of adr. field placed in 4 MSB's of 25-bit width. Low 21 LSB's fill as 0's.
Scaler Range	2^2 (left 2) to 2^{-13} (right, 13).

The Analog Section

Figure 1-6 shows a detailed block diagram of the 2920's analog section, which provides four analog input channels and eight analog output channels. It includes circuitry for analog-to-digital conversion by successive approximation, and the sample- and- holds for both input values and output values.

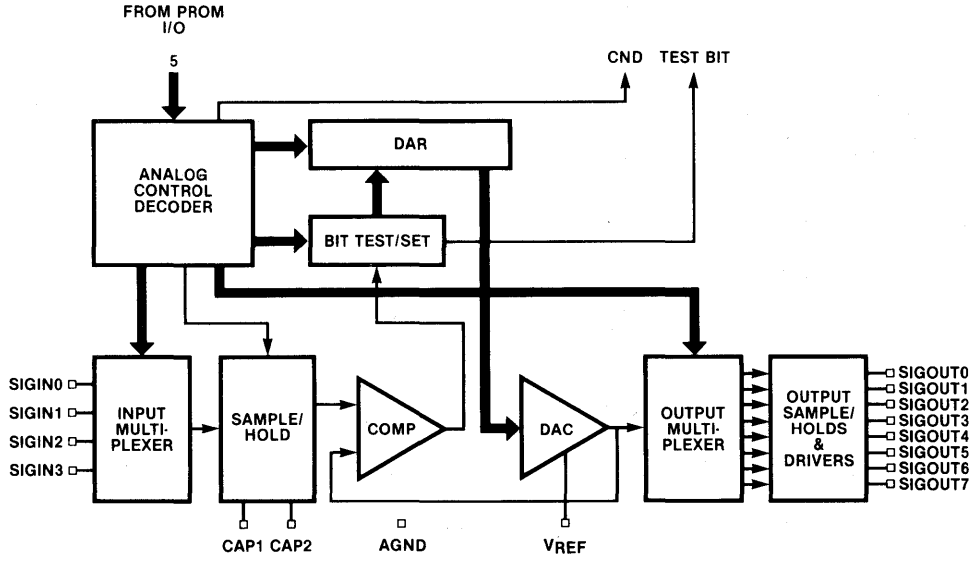


Figure 1-6. Analog Section Block Diagram

All operations of this section are controlled by a five-bit control field, the iocode. It can be conceptually divided into two subfields: a two-bit function selector and a three-bit modifier field. Table 1-6 summarizes the analog section operations. Giving the most-significant-bits first, the function select bits are designated ADF1 and ADF0; the modifier bits are ADK2, ADK1, and ADK0.

The basic analog functions are as follows:

Execution of one or more "IN" instructions provides a sample of one of the input leads. You may have to execute several such instructions in sequence due to the time constants of the sample capacitor charging circuit.

The sample is converted to its digital equivalent by a series of "CVT" instructions in descending order. The digital equivalent is produced in the DAR, which may then be read by the arithmetic section.

Calculation results in the DAR may be delivered to an output pin via an "OUT" instruction.

Input and output sample rates are determined by the frequency of execution of input/conversion sequences and output instructions, respectively.

The input channel multiplexer consists of four analog switches which directly connect a common external sampling capacitor to the input terminals. The size of this capacitor affects both the time constant of the sampling circuit and also the offset voltage, due to the charge coupled through the sampling switches. The value of sample capacitor is usually that value which will result in an offset voltage in the order of 1/2 least significant bit.

Note that the sample capacitor is shared among all inputs. Its selection must be based on the most stringent combination of input parameters.

The analog to digital conversion system uses successive approximation via a binary search routine under program control. Using the CVTS and CVT(K) iocodes in descending sequence puts a 9-bit digital representation of an input sample into the DAR.

The DAR is a two's-complement binary register, nine bits long. When DAR values are delivered to the DAC, they are converted to sign magnitude format via a one's complement operation (see Appendix I). This leads to a potential 1 least significant bit offset during A/D conversion, and a half least significant bit offset during D/A conversion. To compensate for the A/D offset, values read from the DAR have all bits (save the high-order nine) set to ones.

Each CVT cycle sets the selected bit of the DAR to a value derived from the comparator, and also sets the next lower bit to a logic 1. Each cycle must allow the DAC to settle, so at least one NOP iocode is needed between each pair of successive CVT instructions.

Some applications, such as sampling of logic inputs, may not require a full conversion sequence. You may use a partial conversion sequence, with the understanding that the partially converted value undergoes the transformation meant to correct a full 9-bit conversion.

Each of the 2920's eight analog output channels includes an individual sample-and-hold circuit demultiplexed from a common, buffered DAC output.

There are several factors which affect the nature of the output waveform. Writing to the DAR, i.e. using the DAR as the destination, automatically activates anti-crosstalk circuitry in the buffer amplifier. An "OUT" operation should not appear in an instruction which writes to the DAR. For the most error-free output, the first "OUT" instruction should appear only after the time needed for the amplifier to settle; that is, it should be delayed by several instructions after the one which writes into the DAR. Acquisition time of the output sample-and-hold is typically longer than the instruction cycle, so that a sequence of several "OUT" instructions will usually be necessary.

Table 1-6. Analog Instruction Opcodes

a. Basic Codes

CODE	MNEM	FUNCTION
ADF		
1 0		
0 0	IN (k), ADK= 0-3	Acquire input k
0 0	NOP , ADK=4	No operation
0 0	EOP , ADK= 5	Return PROM to Location 0
0 0	CVTS , ADK= 6	Convert Sign Position (MSB)
0 0	CNDS , ADK= 7	Conditional Arith. of Sign Bit (MSB)
1 0	CVT (k), ADK=0-7	A to D convert bit k*
0 1	OUT (k), ADK=0-7	Output Channel k
1 1	CND (k), ADK= 0-7	Cond. Artih., Test DAR bit k*

b. Code Assignment and Mnemonics

ADK	ADF 1,0=			
2 1 0	00	01	10	11
0 0 0	IN0	OUT0	CVT0	CND0
0 0 1	IN1	OUT1	CVT1	CND1
0 1 0	IN2	OUT2	CVT2	CND2
0 1 1	IN3	OUT3	CVT3	CND3
1 0 0	NOP	OUT4	CVT4	CND4
1 0 1	EOP	OUT5	CVT5	CND5
1 1 0	CVTS	OUT6	CVT6	CND6
1 1 1	CNDS	OUT7	CVT7	CND7

*Note—The DAR bits are designated S, 7, 6, ... 0, where S is the sign bit, 7 the next most significant bit, etc. Conversion of bit k consists of setting bit k to a value determined by the comparator, and bit k-1 equal to a logic 1.

Chapter 2
2920 ASSEMBLY LANGUAGE ELEMENTS

Introduction

You write 2920 programs in a symbolic language, using selected mnemonics to represent the desired contents of the various control fields. Table 1-1 showed the various mnemonic representations for commands, which will be further explained in this chapter.

In addition to command mnemonics, you may assign symbolic names to locations in the RAM, so long as the reserved words are not used. Table 2-1 and Appendix B list the words reserved by the 2920 Assembler.

Symbolic names for RAM variables must start with a question mark (?), an at-sign (@), an underline (—), or an alphabetic character (A-Z). Characters after the first may be these or numerals. Names may have up to 31 alphanumeric characters. No spaces, punctuation, or any other characters may be used within a name. After 31 contiguous legal characters in a name, additional characters are flagged as errors.

Except for comments and Assembler controls, explained below, each statement corresponds to one 2920 PROM word. Statements may be labeled, but the 2920 allows no jumps except the EOP. Thus you normally label statements solely as a prospective aid during the simulation, debugging, testing, and optimization phases of program development.

Chapter 1 provided examples of typical contents for each field that may appear in a 2920 statement:

- an optional label,
- an ALU operation,
- the destination address,
- the source address,
- an optional scaler control code (shiftcode),
- an optional analog control command (iocode),

in that order, left to right. The current chapter will provide the detailed rules for constructing legal names and statements.

All PROM word statements and commands end in a carriage-return (CR) and linefeed (LF) pair, which delineates statements. The ISIS-II Editor automatically appends the LF when you hit CR. Comments, which must begin with a semicolon, may be coded prior to this CR. They are useful in explaining the intended results of each part of a program, and may appear alone on a line.

Here is an example of an instruction using all seven fields:

```
Label    Opcode   Dst,Src,Shift,Iocode    Comment
LOAD:    LDA      YYY,XXX,R02,NOP         ; Move XXX/4 to YYY. (CR)
```

This statement means fetch the contents of the RAM location symbolically named XXX, shift the value right two bit positions, and store the result in the RAM location symbolically named YYY. NOP iocode means no I/O operation is done.

If XXX and YYY have not appeared earlier, then the Assembler reserves a RAM location for each, and the names "XXX" and "YYY" are entered into the symbol table with the value of their respective addresses.

Characters

The alphabet and numerals are legal in assembly language source statements.

AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXx
YyZz0123456789

Although the Assembler converts lower-case to upper case for internal processing, the listing of your input shows the character as you typed it originally. Thus to the Assembler Xyy is identical to XYY.

In addition, the following special characters are recognized as legal in certain contexts:

CHARACTER	COMMON NAME	USAGE
,	comma	separates operands, shiftcode, and iocode
:	colon	must immediately follow last character of label
@	commercial at-sign	valid character of name
?	question mark	valid character of name
space	blank	separates label, opcode, and first operand field
_	underscore	valid character of name
;	semicolon	must be first character of comment
CR	carriage return	statement terminator
LF	line feed	must appear only following a CR —otherwise an error
HT	horizontal tab	separator, same as space
$	dollar sign	for control options (e.g., LIST, TITLE, or EJECT) when appearing within the source lines of a program file, $ must be first character on the line
		in user-symbols, a non-first character used to space words in a long name, the $ ignored by the assembler, for example, LASTDARSAVED

All null and RUBOUT characters are ignored upon input. All other characters, or characters in an inappropriate place, are flagged as errors. However, in a comment field, any ASCII character may be used.

Delimiters

Certain characters are used to define the end of a statement or a field or a component of a field. These can be called "separating characters," "terminators," or "delimiters". The six characters—space, comma, CR, HT, semicolon, and colon—do this, as described above.

Symbols

There are two kinds of symbols: reserved symbols and user symbols.

Reserved symbols, as mentioned in Chapter 1, cover the predefined opcodes, directives, registers, special field bit patterns, and special data memory locations. You may not redefine any of these. They are repeated here from Chapter 1, and appear in Appendix B for easy reference.

Table 2-1. Reserved Symbols

Section 1: Arithmetic Operation Codes

ABA ABS ADD AND LDA LIM SUB XOR

Section 2: Analog Control Codes

CNDS	CVTS	EOP	IN0	NOP	OUT0
CND7	CVT7		IN1		OUT1
CND6	CVT6		IN2		OUT2
CND5	CVT5		IN3		OUT3
CND4	CVT4				OUT4
CND3	CVT3				OUT5
CND2	CVT2				OUT6
CND1	CVT1				OUT7
CND0	CVT0				

Section 3: Constant Source Codes and the DAR

	KM1	KM2	KM3	KM4	KM5	KM6	KM7	KM8
KP0	KP1	KP2	KP3	KP4	KP5	KP6	KP7	
DAR								

Section 4: Scaler Control Codes

R00 R01 R02 R03 R04 R05 R06 R07 R08 R09 R10 R11 R12 R13
R0 R1 R2 R3 R4 R5 R6 R7 R8 R9

L01 L02
L1 L2

Section 5: Assembler Commands and Modifiers

DEBUG	END	OBJECT( )	PAGING	SYMBOLS
NODEBUG	EQU	NOOBJECT	NOPAGING	NOSYMBOLS
PAGELENGTH( )		PAGEWIDTH( )	PRINT	PRINT( )
			NOPRINT	
EJECT	LIST	TITLE(' ... ')		
	NOLIST			

User-created symbols refer to instruction locations or data locations by name. You must refer to RAM variables symbolically. You may not specify absolute locations. Instead, locations are automatically allocated whenever variable names are first encountered. No declaration of variables is needed or allowed. These symbols are defined by the Assembler, i.e. given an address, the first time they appear, as discussed in the following three cases:

1. in the label field of a statement (see below)
2. in the right-hand side of an EQU statement (see Chapter 3) or as a source or destination
3. in the left-hand side of an EQU statement.

- In case (1) the value of the symbol is the address of the instruction, which is the value of the location counter when the instruction is assembled. These symbols can only be accessed with the 2920 Simulator.

- In case (2), if the symbol is not already defined, the Assembler automatically allocates a RAM location and enters that address as the value of the symbol in the symbol table. One previously unused location is reserved for each symbol so defined.

  The assembler keeps a RAM location counter incremented each time a symbol is created in this manner. This counter starts with 0 and is allowed to increment indefinitely. A warning is issued, however, each time a user-symbol with address- value greater than 39 is used, although the assignment is made and code is generated anyway. Note that assignment of user symbols continues in sequence to 63, and then wraps around to 0,1,2, etc. Warnings will continue to be issued.

- In case (3), the symbol is given the value of an already created symbol, namely the one on the right of the EQU. If the symbol on the right-hand side had not yet been defined, it is defined first per case (2). This usage establishes an equivalence of variables and can be used to conserve RAM space by reusing "scratch" variables.

Reserved symbols cannot be used as user-created symbols. Attempts to do so will cause a "multiply-defined symbol" error message. Attempts to recreate a user-created symbol will also cause this error message, i.e. using the same symbol more than once on the left side of an EQU.

Statements

A statement is composed of one or more fields, identified by their order of appearance and by specific terminating characters. However, certain fields may not appear alone, i.e. a shiftcode, a source, or a destination. The statement is free-form, allowing any number of blanks and/or horizontal tabs to separate fields. A comma must be used to separate the operands. No continuation lines are allowed: the entire statement must appear before the CR.

Certain default constructions are supplied when specific fields are omitted or appear alone:

1. If the shift code is omitted, then the Assembler inserts a shift code "R00," i.e. no shift.
2. If the iocode is omitted, then the Assembler inserts a NOP, i.e. no iocode operation.
3. If only the iocode appears, then the Assembler inserts LDA 0,0,R00; in effect, no ALU operation (0 means loc. 0). (This also clears the carry flag.)

Thus you need not explicitly code a no-shift and no-I/O-operation. Similarly, if you need ONLY the iocode, you need not explicitly code a no-arithmetic-operation.

If an instruction is incorrectly formed for any reason, then no code is produced and no location is reserved. An appropriate error message is placed in the list file.

Label Field

A label names a ROM location intended to contain an instruction. It is a user-created symbol, and must be unique within the first 31 characters. (Characters after 31 are flagged as errors.) It is assigned the value of the ROM location counter, and this address is entered into the symbol table as the value of the symbol. There is no way to use these labels within the assembler, but they can be useful in testing and debugging via the 2920 Simulator. A label may not have the same name as a variable.

A label is optional. If used, it must be immediately followed by a colon and at least one blank or HT. Once used, it can not be used again as a label or a variable in the same program without causing an error.

Opcode Field

This field must contain the mnemonic ALU operation code for a machine instruction. It specifies the 2920 instruction to be generated by the Assembler and the ALU action to be performed on the operands which appear in the operand field. A blank or HT must be used to separate the opcode from the operands or a label.

Operand Fields

The destination and source operands appear in these fields, in that order, separated by a comma. They must be symbolic names. They are followed by the (optional) shiftcode and the iocode, also separated by commas when present.

The destination operand is intended to be written to, and should therefore correspond to the DAR or to a RAM address not greater than 40. The source operand may refer to any of the 64 RAM locations, including the DAR or the constant registers. An error message is issued if you use a source address above 63.

Comment Field

This optional field may contain text descriptive of the statement or the program. Comments are ignored by the Assembler but echoed to the assembly listing as part of the source statement.

The comment field must begin with a semicolon, and is terminated by a carriage return. Any ASCII character may appear in the comment field except a CR or LF. The comment may appear alone on a source line.

CHAPTER 3
INSTRUCTION SET

How to Use This Chapter

This chapter is a dictionary of 2920 instructions. The instruction descriptions, including opcodes and iocodes, are listed alphabetically for quick reference. However, the Assembler controls are not listed here, but appear instead in Chapter 5.

This reference format necessarily requires repetitive information. If you are reading this manual for the first time, you might skim this chapter or skip it at first, reading instead Chapters 1 and 4. They will familiarize you with the general setup of the language and the simpler applications of general interest. When you begin to have questions about particular instructions, look them up in this chapter.

Timing Information

The instruction descriptions in this manual do not explicitly state execution timings. The clock frequency used in your system will determine the operating speed of your processor. The maximum sample rate for a full length program is found by dividing the maximum clock rate by 768, which represents 192 instructions at four clock cycles per instructions.

To realize higher sample rates, either shorter programs must be used, or multiple copies of the appropriate program segments must be contained in the PROM.

The external clock or crystal frequency and the length of the program establishes the system sample rate as explained in Chapter 1.

ABA Absolute Value and Add

The absolute value of the source, after any shifting, is added to the destination. The CND iocode affects overflow limiting logic if used with this instruction.

Examples:

```
GAINER:     ABA     DEST_4,SOURCE_1
```

This will take the absolute value of the contents of SOURCE_1 and add that to the value in DEST_4, placing the result in DEST_4.

```
DOUBLER:    ABA     DEST_3,SOURCE_1,L01
```

This will double the value from SOURCE_1 by a left shift one position, then take the absolute value of that result and add it into DEST_3.

```
            ABA     DEST_55,SOURCE_1,L02,CNDS
```

After shifting the value from SOURCE_1 left two positions, effectively multiplying it by four, this command will add the absolute value of that result into DEST_55.

When a CND iocode is used on this instruction, the limiting effect of overflow detection is turned off. It will be turned on again when an XOR instruction with any CND iocode is encountered, or when an EOP is encountered.

The normal standard carry and overflow apply to ABA, as explained in Appendix J.

ABS Absolute Value

This instruction takes the absolute value of the source operand, after any shifting, and stores it in the destination. If the source was positive, the destination becomes identical to the source. If it was negative, the destination is the "negative" of the source, that is, of same magnitude and opposite sign. The Assembler issues a warning if this instruction is used with a "CND" iocode but there is no effect on execution.

Examples:

```
AMPLITUDE:   ABS     DEST_7,SOURCE_1
```

This instruction places the absolute value of SOURCE_1 into DEST_7. If SOURCE_1 were 0.0000 0001, DEST_7 would become 0.0000 0001. If SOURCE_1 were 1.1111 1111, DEST_7 would become 0.0000 0001.

```
HALFAMP:     ABS     DEST_7,SOURCE_1,R01
```

This command shifts the value from SOURCE_1 to the right 1 position, effectively halving that value, and then places the absolute value of this result into DEST_7.

```
NONO:        ABS     DEST_7,SOURCE_1,CND4
```

A warning will be issued due to the use of the CND iocode on this command, but the execution is unaffected.

ABS never has a carry; it clears the carry flag to zero. A left-shift could cause overflow.

ADD Addition

After any shifting of the source operand, this instruction forms the sum of the source and the destination operands. The result is stored in the destination. The instruction will have no effect on the destination if the iocode CND(K) is specified and the corresponding bit of the DAR, i.e. DAR(k), is zero.

Examples:

```
SUM_1:       ADD     DEST_5,SOURCE_1
```

The sum of the contents of SOURCE_1 and DEST_5 will be placed in DEST_5.

```
SUM_2:       ADD     DEST_5,SOURCE_1,R01
```

The shiftcode R01 will shift the value from SOURCE_1 right one position, effectively halving it. This result will be added into the current contents of DEST_5.

```
SUMMA:       ADD     DEST_ZETA,SOURCE_XI,L02,CND4
```

The value from SOURCE_XI will be shifted left 2 positions, effectively multiplying it by four. This result will be added to the value currently in DEST_ZETA, with the final sum stored in DEST_ZETA. These operations will be performed only if bit four of the DAR is 1. If that bit is zero, no operation will take place.

The normal standard carry and overflow apply, as explained in Appendix J.

AND Logical Conjunction

After any shifting of the source operand, this instruction performs the logical AND of that shifted value with the value from the destination, and stores the result in the destination. The Assembler issues a warning if this instruction is used with a CND iocode, but there is no effect on execution.

Examples:

```
ANDER:      AND     DEST_1,SOURCE_2
```

The value from SOURCE_2 will be ANDed against the value from DEST_1, with the result of this logical operation placed into DEST_1.

```
            AND     DEST_1,SOURCE_2,R03
```

After shifting the value from SOURCE_2 to the right three positions, this instruction will AND the result against the value from DEST_1, storing the result of this logical operation back into DEST_1. The effect depends on the sign bit of SOURCE_2, since right shifting fills from the left with whatever the sign bit was, 1 if negative, 0 if positive.

```
ENDER:      AND     DEST_1,SOURCE_2,L2,CND7
```

A warning will be issued due to the use of a CND iocode on this instruction. The value from SOURCE_2 would be shifted left two positions, filling the two vacated bit positions with zeroes, and then that result would be ANDed against the value from DEST_1. The AND result is stored back into DEST_1.

The normal standard carry and overflow apply, as explained in Appendix J.

CNDS,CND7,CND6,CND5,CND4,CND3,CND2,CND1,CND0 Iocodes for Conditional Operations

Each of these iocodes refers to a single bit, either a bit of the DAR or the carry bit. CNDS means conditional on the sign bit; the others refer to specific bit positions in the DAR. CND0 refers to the least significant bit.

If the tested bit is a 1, the operation is performed as written. If the tested bit is a 0, the operation is either not performed at all or is altered. Only three ALU opcodes are affected: ADD, LDA, and SUB.

Add Conditional

```
ADD     DEST_ONE,SOURCE_ONE,CNDS
```

If the sign bit of the DAR is 1, this instruction will add the contents of SOURCE_ONE to the contents of DEST_ONE and store the result in DEST_ONE.

If the sign bit of the DAR is 0, then the sum of DEST_ONE with zero is placed into DEST_ONE, i.e. no change except that the carry flag is cleared.

Load Conditional

```
LDA    DEST_ONE,SOURCE_ONE,R2,CND5
```

If bit five of the DAR is 1, this instruction will get the value of SOURCE_ONE, shift it right two positions to create 1/4 the value, and put it into DEST_ONE, writing over whatever value was formerly there.

If bit five is 0, the effect is the same as with the conditional add.

Subtract Conditional

```
SUB    DEST_ONE,SOURCE_ONE,CND0
```

Conditional subtract is a special operation, requiring information about the previous carry situation.

> If the carry resulting from the previous ALU operation is a 1, then the subtraction indicated is performed, i.e., the value from SOURCE_ONE is subtracted from the value in DEST_ONE, and the result is written into DEST_ONE.
>
> If the carry resulting from the prior ALU operation is 0, then the operands are added instead of being subtracted, i.e., the value from SOURCE_ONE is added to the value from DEST_ONE, and the sum is written into DEST_ONE.

The above instruction will set the first bit of the DAR, DAR(0), to the carry output of the highest order position of the ALU. Then, depending on the carry resulting from the previous ALU operation, it will perform either an addition or a subtraction.

A detailed discussion of subtraction appears in Chapters 2 and 4.

IN ALL INSTRUCTIONS, THE ABSENCE OF A SHIFTCODE CAUSES THE USE OF THE DEFAULT R00; i.e., no shift. If a shiftcode is coded, it is performed prior to the indicated operation. This means the source operand is shifted before it is added to or loaded into or subtracted from the destination operand.

The normal standard carry and overflow apply, as explained in Appendix J. LDA, however, never has a carry.

CVTS,CVT7,CVT6,CVT5,CVT4,CVT3,CVT2,CVT1,CVT0 A/D Conversion Iocodes

In order to convert to a digital value from an input sample value in the sample-and-hold for input, each of these iocodes will set the named bit of the DAR (e.g. bit 7 for CVT7) to 1 or 0 based on that input value. Each CVT also sets the next lower bit (e.g., bit 6 for CVT7) to 1 as part of the conversion process. The process uses the comparator and the reference voltage (VREF) to decide the sign and the fraction of VREF which represents the input sample. As mentioned briefly in Chapter 1, it is necessary to allow the DAC to settle between each cycle of conversion. This is achieved by inserting NOP iocodes after all but the last CVT, or placing CVT iocodes only on every other ALU instruction.

```
CVTS                                    or, say,        ADD     DET,SIC,CVTS
NOP                                                     ADD     DET,NXT,NOP
CVT7                                                    SUB     DET,LST,R01,CVT7
NOP                                                     ADD     DET,SIC,R04,NOP
CVT6                                                    ADD     DET,NXT,R07,CVT6
NOP                                                     ADD     DET,NXT,R09,NOP
CVT5                                                    LDA     SIC,NXT,CVT5
NOP                                                     LDA     NXT,LST,NOP
CVT4                                                    ADD     LST,LST,L02,CVT4
NOP                                                     SUB     LST,LST,R05,NOP
CVT3                                                    SUB     LST,LST,R07,CVT3
NOP                                                     SUB     LST,LST,L01,NOP
CVT2                                                    ADD     NXT,LST,CVT2
NOP                                                     SUB     SIC,NXT,NOP
CVT1                                                    LDA     LST,SIC,CVT1
NOP                                                     ADD     LST,LST,L01,NOP
CVT0                                                    SUB     LST,NXT,CVT0
LDA     SIC,DAR                                         LDA     SIC,DAR
```

Either column causes the conversion of an input sample into a digital value in the DAR. If the left column is coded, the Assembler supplies the default arithmetic-no-operation coding of

```
LDA 0,0,R00
```

The right column takes advantage of the parallel processing capability of the 2920 to compute some arithmetic function of the values in locations named DET,LST,NXT, and SIC while the conversion process is going on. It then stores the converted value from the DAR into SIC for further processing. The DAR could then be used to output the computed value in LST or NXT.

END Terminating Assembly

This command is properly termed a directive to the Assembler rather than an instruction, since it does not cause code to be generated. When the Assembler sees the first END in a source program/file, it terminates its scan of the source program and proceeds to finish all Assembler functions and outputs. There should be only one END per program and it should be the last source line of the program. It must have no name, label, operands, or comment.

```
END
```

EOP End of Program (Iocode)

EOP signals the end-of-program condition, causing a transfer back to the instruction in location zero. This iocode must be on/in a location whose address is a multiple of four, or a warning is issued.[1] The 2920 instruction words are pipelined in groups of four. If any of the three locations following the EOP do not contain assembled code, they are padded with NOP instructions, and a warning is issued.

The EOP does not terminate assembly. Only the END or an end-of-file condition does this. The Assembler will continue to process statements after the EOP, but only the next three will be executed by the 2920.

```
EOP     ; begin fetching locations 0-3 while executing this and the next three
          instructions.
```

or

```
LDA     SRC,DAR,R01,EOP
```

Overflow limiting is turned on by the execution of an EOP and thus is enabled during the last four instructions of the program.

[1]If a program with a misplaced EOP is executed, the results are unpredictable.

EQU Equate—Creating a Synonym for a Single Location (Address)

This command is properly termed a directive to the Assembler rather than an instruction, since it does not cause code to be generated.

The general form of the EQU statement is

```
name_1    EQU    name_2
```

The symbol "name_1" is created and assigned the symbol table value (address) of "name_2." It may appear on the left-hand side of an EQU only once.

If "name_2" has not been defined prior to this command, i.e. this is its first appearance also, then "name_2" is defined first. After "name_2" has a value, the EQU creates "name_1" as a synonym for that value in the symbol table.

If "name_1" had been defined earlier, an error message would be issued for attempting to use that symbol for more than one location.

EQU can be used to economize RAM space usage: "scratch" variables can be reused, although care must be taken to ensure that such variables are not changed to serve one purpose while they are relied upon for another purpose.

IN0, IN1, IN2, IN3 Input Iocodes

You use these iocodes to obtain an input sample from one of the four input channels. It is generally necessary to use a sequence of several INs in order to obtain a reliable sample. The number of INs is a function of the capacitor.

As explained in Chapter 1, the sample capacitor is shared among all inputs and is chosen as a compromise between rapid sampling and offset voltage. Suppose the capacitor selected has been determined to adequate accuracy. The assembly language instructions might appear as either of the examples below:

```
IN3                    or, say,          LDA    NXT,LST,R01,IN3
IN3                                      ADD    NXT,DET,IN3
IN3                                      ADD    NXT,SIC,R02,IN3
IN3                                      SUB    NXT,LST,R03,IN3
```

As with all the iocodes, INs may appear alone or appended to instructions.

LDA Load Source to Destination

This instruction writes into the specified destination the value of the source operand after any shifting. If a CND iocode was specified on the LDA instruction, then the LDA will be executed only if the DAR bit specified by the CND iocode is 1. If the specified DAR bit is 0, the LDA executes as an ADD of the source operand with zero, effectively a NOP (no-operation) except that carry is cleared.

```
LDA    DEST_TWO,SOURCE_TWO,R01
```

This instruction writes into DEST_TWO half the value from SOURCE_TWO because that value is right-shifted one bit position before the LDA gets it.

```
LDA    DEST_TWO,SOURCE_TWO,R3,CND4
```

This instruction will operate as a NOP if bit four of the DAR is 0. If that bit is a one, then DEST_TWO will be filled with one eighth the value from SOURCE_TWO, due to the right shift three positions specified by the shiftcode R03.

LDA always clears the carry. A left-shift could cause overflow.

LIM Load Destination with Source Limit

This instruction loads one of two extreme values into the destination, based on the sign of the source operand. If the source is positive or zero, the destination gets a plus 1 (0.11111111111111111111111). If the source is negative, the destination gets a minus 1 (1.00000000000000000000000).

The Assembler issues a warning if this instruction is used with a shiftcode or with a CND iocode, but there is no effect on execution.

```
LIM    DEST_ONE,SOURCE_ONE
```

The contents of DEST_ONE will be −1.0 or +1.0 depending on whether SOURCE_ONE is negative or not, respectively (zero being non-negative).

```
LIM    DEST_ONE,SOURCE_ONE,CNDS
```

A warning will be issued due to the use of the iocode CNDS. Other iocodes would be allowed. The LIM is unaffected by the CNDS.

```
LIM    DEST_ONE,SOURCE_ONE,R12,IN2
```

A warning will be issued because a shiftcode has no effect on a LIM. Input-line-2 will be sampled. DEST_ONE will be written with −1.0 if SOURCE_ONE is negative, +1.0 if zero or positive.

The normal standard carry and overflow apply, as explained in Appendix J. LIM sets the carry to 0, and can have an overflow only via a left shift.

NOP No-Operation, Instruction or Iocode

As an instruction, NOP means

```
LDA    0,0,R00,NOP ; no effect but to clear the carry
```

As an iocode, NOP means no-operation for the analog section of the 2920 chip.

OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7 Output Iocodes

These iocodes cause the value in the DAR to be converted to analog and output to the specified channel.

As explained briefly in Chapter 1, the acquisition time of the output sample-and-hold is important in determining how many successive OUT iocodes should be used. The technique is to divide the acquisition time by the time it takes to execute each instruction, i.e., by one-fourth the external clock rate.

An OUT iocode should be delayed after the DAR is written, because the amplifier activated by a write to the DAR takes some time to settle. This usually represents

several successive instructions. Writing to the DAR (using DAR as the destination) automatically activates anti-crosstalk circuitry in the buffer amplifier, and then activates the amplifier. An OUT iocode should not be coded onto such a write, but delayed until the amplifier settles.

```
LDA    DAR,DEST_TWO
SUB    DEST_TWO,NXT,R01,NOP
SUB    DEST_TWO,NXT,R04,NOP
ADD    DEST_TWO,SIC,R01,NOP
SUB    DEST_TWO,SIC,R04,NOP
NOP
LDA    LST,DEST_TWO,OUT1
OUT1
OUT1
```

Here the first instruction shown writes the value from DEST_TWO into the DAR. The next five iocodes are NOP. In the last two source lines, coding only an iocode causes the Assembler to supply a NOP for the ALU, namely LDA 0,0,R00. This is also true of the 6th line.

SUB Subtraction

This instruction subtracts the value in the source operand (after any shifting), from the value in the destination operand. Subtraction is done by adding the one's complement of the source and forcing a carry input at the lowest-order bit. See Appendix I.

If a conditional iocode is specified, then the previous carry is tested. If that carry was a 1, the SUB instruction operates as a subtraction. If it was a 0, the instruction operates as an addition. The carry produced by the SUB operation is stored in the DAR bit specified by the conditional iocode.

```
SUB    DEST_TWO,SOURCE_ONE
```

Here the value from SOURCE_ONE is subtracted from the value in DEST_TWO, writing the result into DEST_TWO. No test of a prior carry was done.

```
SUB    DEST_TWO,SOURCE_ONE,CNDS
```

Here the DAR sign bit will get the carry from this operation. The prior carry will determine whether this instruction is executed as a subtraction or an addition.

The normal standard carry and overflow apply, as explained in Appendix J.

XOR Exclusive OR Instruction

This instruction forms the exclusive OR of the source (after any shifting) with the destination, and stores the result in the destination. A CND iocode will affect overflow limiting logic if used with this instruction. Exclusive OR gives a 1 in each bit position where only one of the two values has a 1, and gives a 0 in those bit positions where both have ones or both have zeroes.

```
XOR    DEST_ONE,SOURCE_ONE
```

This will form the exclusive OR of the values in these two operands, and the result will be written into DEST_ONE.

```
XOR    DEST_ONE,SOURCE_ONE,CNDS    ; overflow affected
```

The exclusive OR will be formed as before, but the overflow limiting on overflow detection will be turned on.

XOR is implemented as an ADD with no carries. See Appendix J for further discussion of carry and overflow for XOR.

CHAPTER 4
PROGRAMMING TECHNIQUES—SOME SOLVED PROBLEMS

Elementary Arithmetic

Overflow Considerations and Scaling

Whenever doing arithmetic with the 2920, you should consider the impact of scaling the variables. If variables are improperly scaled, either quantization noise will be added to the signal or overflow saturation can result. These effects are similar to those encountered in analog systems, where use of poorly chosen signal levels can lead to poor signal-to-noise ratios or amplifier overload distortions.

During certain 2920 operations, such as multiplying by a constant, intermediate values may be larger than the final result. If these intermediate values are large enough to produce overflow saturation, undesirable non-linearities may result.

As a rule, you should estimate signal levels throughout your system, and scale so as to maintain the largest levels without exciting overflow. Some calculation sequences are less prone to overflows than others.

Two Methods

One way to achieve this is to order all instructions so that only the last step involves values large enough to produce overflow. In some cases, there may be more than one large term being summed, so that this method is not always possible. A second method consists of computing a submultiple of the desired value, which is then increased at the end of the sequence, possibly by loading or adding a shifted version to itself. The most likely submultiples are 1/2, 1/3, 1/4, or 1/5, because the multiplications necessary to restore the proper value are easily done in one microinstruction.

Addition and Subtraction

The basic arithmetic instructions of the 2920 allow you to add, subtract, or replace one variable with another in a single instruction. For example,

```
ADD    YYY,XXX
```

adds the value stored in the RAM location labeled XXX to the value in the RAM location labeled YYY, storing the result in YYY. No scaler code is specified, thus invoking the default of R00, a right-shift of zero.

Similarly, the value to be added or subtracted can be scaled by a power of two in a single instruction, e.g.,

```
SUB    YYY,XXX,R02
```

causes one fourth of the value in XXX to be subtracted from the variable YYY. The equivalent FORTRAN language statements for the two operations above would be

```
YYY = YYY + XXX
YYY = YYY - (0.25 * XXX)
```

respectively. In general, the 2920 instruction set makes it easy to implement the equivalent of the FORTRAN statement

$$YYY = YYY + (C * XXX)$$

where C is an arbitrary constant. The next section describes some general rules for achieving this result.

Multiplication and Division

Multiplication by a Constant

The number of 2920 steps required to perform the above operation depends on the value of C. Any value C can be expressed as an expression consisting of sums and differences of powers of 2, using positive and negative powers. Once a constant is expressed this way, the equivalent to YYY = YYY + C * XXX can be easily converted to 2920 code.

Consider a value of C = 1.875. This value could be expressed in several different ways, e.g.,

$$1.875 = 1.0 + 0.5 + 0.25 + 0.125 = 2^0 + 2^{-1} + 2^{-2} + 2^{-3}$$
$$1.875 = 2.0 - 0.125 = 2^1 - 2^{-3}$$

The first expression could be easily derived from the binary representation of 1.875, i.e., 1.111. However, the second expression uses fewer terms, which will result in the use of fewer 2920 PROM words.

Using the second form, a FORTRAN-like expression for YYY becomes

$$YYY = YYY + 2^1*XXX - 2^{-3}*XXX$$

which could be written as two sequential FORTRAN-like statements,

$$YYY = YYY + 2^1 * XXX$$
$$YYY = YYY - 2^{-3} * XXX$$

These statements are directly convertible to 2920 code:

```
ADD     YYY,XXX,L01
SUB     YYY,XXX,R03
```

The sequence of operations can sometimes be found by inspecting a binary representation of the constant C. For example, consider

C = 1.88184 (= 1.111 0000 111 in binary)

C might be represented by

$$C = 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-8} + 2^{-9} + 2^{-10}$$

which would take seven steps, or more simply

$$C = 2^1 - 2^{-3} + 2^{-7} - 2^{-10}$$

which takes only four steps in 2920 coding, as follows:

```
ADD     YYY,XXX,L01
SUB     YYY,XXX,R03
ADD     YYY,XXX,R07
SUB     YYY,XXX,R10
```

An Algorithm for Multiplication by a Constant

Multiplication by a constant usually requires fewer 2920 instructions than multiplication by a variable, which will be covered in a later section. A technique is shown below for deriving the expression that represents the constant you want.

1. Let C be the value desired for the constant, and let V represent the result of estimating C with a series of sums and differences of powers of 2. Initially V=0.
2. Define an error ERR=C−V, representing the difference between the desired value and the current estimate.
3. Choose T, a power of 2, which is closest to ERR and minimizes the absolute difference, i.e., with least $|T-ERR|$.
4. Let V=V+T, and compute a new ERR as in step 2. If it is small enough, you're done, and C is now expressed as powers of two, in V. If not, repeat steps 3 and 4 until it is.

For example, suppose you need a C=−0.65, within a tolerance of ± 0.01. The steps of the algorithm are as follows:

Initially V0=0 and ERR0=−0.65

Step 1: T1 = -2^{-1} or −0.5; V1 = −0.5 and ERR1 = −0.150

Step 2: T2 = -2^{-3} or −0.125 (the closest power of 2 to −.15);

so V2 = −0.625 and ERR2 = −0.025 (i.e., □−0.65−(−0.625))

Step 3: T3 = -2^{-5} or −0.03125; V3 = −0.65625; ERR3 = +0.00625

At Step 3, the error value has fallen within the specified bounds. V may be expressed as $V = -2^{-1} - 2^{-3} - 2^{-5}$. Therefore YYY=YYY—C*XXX may be approximated by the following 2920 code:

```
SUB   YYY,XXX,R01
SUB   YYY,XXX,R03
SUB   YYY,XXX,R05
```

If the form YYY = C * XXX is desired (instead of YYY=YYY+C*XXX), then YYY can be initialized to zero by either of these two instructions:

```
LDA   YYY,KPO
SUB   YYY,YYY
```

If there is an ADD later in a sequence beginning with a SUB, the order of operations can sometimes be rearranged to place the ADD first, where it can be replaced by an LDA. This can avoid the need for the initialization to zero.

Multiplication of the Form Y = C * Y

To achieve this form, the sequence

```
W = C * Y
Y = W
```

could be used, or the coefficient C may be factored into a sequence of terms T of the form 2^k or $1+2^k$ or $1-2^k$. The factoring can follow an algorithm similar to the one above, such that the value V is updated by B=V*T, with V set initially to 1.0. The Ts are chosen to minimize the error at each step. Each factor (term) corresponds to one 2920 instruction.

Example:

Generate YYY = C * YYY for C = −1.

Note that $-1 = 1-2^1$. Then the 2920 operation

```
SUB    YYY,YYY,L01
```

performs the desired operation.

Generate YYY = C * YYY for C = 0.145 within 0.001

Initially V0 = 1.0

Step 1: $T1 = 2^{-3} = 0.125$; V1 = 0.125; ERR1 = 0.020

Step 2: $T2 = 1 + 2^{-3} = 1.125$; V2 = 0.140625; ERR2 = 0.004375

Step 3: $T3 = 1 + 2^{-5} = 1.03125$; V3 = 0.145020; ERR3=0.00002

Therefore, Y = C * Y for C = 0.145 within 0.001 can be generated by the 2920 sequence

```
LDA    YYY,YYY,R03
ADD    YYY,YYY,R03
ADD    YYY,YYY,R05
```

These algorithms may be implemented by computer, allowing you to painlessly examine several approaches. Hybrid algorithms can produce the closest approximation in the fewest instructions.

Multiplication by a Variable

Multiplication of one variable by another can be done using the conditional ADD instruction. Equivalents to the FORTRAN statements

$Y = W * X$

$Y = Y + W * X$

$Y = Y + W * X * 2^n$

may be derived, where Y, W, and X are variables and n (if used) is a fixed constant integer. Multiplication is easiest if one of the variables, say W, is limited to nine bits of precision.

Consider Y = W * X, where W is the multiplier, X the multiplicand, and Y the product. Several steps are required. The intermediate values of Y are called partial products.

You load W into the DAR and conditionally ADD X, suitably shifted, to the partial product Y, the conditional add tests bits in the DAR. The following example may help to clarify how this is done.

Consider multiplying the binary values W = 0.1011 and X = 0.1101. The sequence is as follows:

```
0.00000
0.01101          multiplicand times 1st multiplier bit
0.01101          1st partial product = 0.1 * 0.1101
0.00000          multiplicand times 2nd multiplier bit
0.01101          2nd partial product = 0.10 * 0.1101
0.0001101        multiplicand times 3rd multiplier bit
0.1000001        3rd partial product = 0.101 * 0.1101
0.00001101       multiplicand times 4th multiplier bit
0.10001111       final product = 0.1011 * 0.1101
```

If the multiplier's sign is negative, an additional step must be included, assigning the weight −1 to the multiplier's sign by adding the negative of the multiplicand. Thus the 2920 code to achieve Y=W*X for a 9-bit multiply is

```
SUB   YYY,YYY,R00
LDA   DAR,WWW,R00          ; multiplier to DAR
ADD   YYY,XXX,R01,CND7     ; multiply by 1st bit
ADD   YYY,XXX,R02,CND6     ; multiply by 2nd bit, etc.
ADD   YYY,XXX,R03,CND5
ADD   YYY,XXX,R04,CND4
ADD   YYY,XXX,R05,CND3
ADD   YYY,XXX,R06,CND2
ADD   YYY,XXX,R07,CND1
ADD   YYY,XXX,R08,CND0
SUB   XXX,XXX,L01          ; complement XXX
ADD   YYY,XXX,R00,CNDS     ; test multiplier sign
```

If more bits of multiplier precision are required, the high order bits of the multiplier may be masked off, and the remaining bits shifted left and loaded to the DAR. The masking operation is necessary to prevent overflow saturation.

The last two steps above can be eliminated if the multiplier is known to be positive. The first step must be eliminated if the operation is of the form

```
Y = Y + W * X
```

Division by a Variable

Division of a variable by a constant can be done by using the inverse of the constant as a multiplier. However, to divide a variable by another variable you must use the conditional subtract. If you use negative variables, you can compute the sign using XOR, and do the division with the absolute magnitudes.

The sequence conditionally subtracts the divisor from the dividend, assembling the quotient in the DAR. You should scale the source operand in the first instruction, which is an unconditional subtraction, to produce a negative result.

Consider dividing 0.100 by 0.111

```
                 0.1000000
                -0.111          ; initial subtract
CY=0             1.1010000      ; first carry, partial remainder
                + .0111         ; 1st conditional subtract (adds)
CY=1             0.00010000
                - .00111
CY=0             1.11011
                + .000111
CY=0             1.111101
                + .0000111
CY=1             0.0000001
                - .00000111
CY=0             1.11111011
```

The quotient so far = 0.10010

The full sequence for a four quadrant divide (Y=X/W) is shown below. This division only works if the quotient is less than 1, i.e., if $X < W$. It is accurate to seven binary places.

```
LDA   TMP,W,R13
XOR   TMP,X,R13
ABS   X,X,R00
ABS   W,W,R00
SUB   X,W,R00
SUB   X,W,R01,CND7
SUB   X,W,R02,CND6
SUB   X,W,R03,CND5
SUB   X,W,R04,CND4
SUB   X,W,R05,CND3
SUB   X,W,R06,CND2
SUB   X,W,R07,CND1
SUB   X,W,R08,CND0
XOR   TMP,DAR,R13
```

Note that the first two and last operations are used to save and restore the sign of the result. The quotient is available in the DAR.

If greater precision is needed, you can save the contents of the DAR before restoring the sign, clear the DAR, and continue the conditional subtractions after restoring the carry value. (The carry should always equal the complement of the sign of the partial remainder.) Restoration of carry can be done by adding and then subtracting the divisor, appropriately shifted, from the partial remainder.

Designing Filters With the 2920

Many analog signal processing applications involve filtering of the signals. This filtering function can be simulated on a sampled basis using digital calculations. Most analog filters can be characterized by the locations of their poles and zeroes. They can be realized as a cascade of sections, each of which realizes a subset of the poles and zeroes. Similarly, sampled filters can be characterized by their pole and zero locations. Simple transformations exist for translating between a continuous filter and its sampled counterpart. The behavior of the sampled counterpart will be similar to the original continuous filter except for frequencies approaching or exceeding half the sample rate. Thus if your signal is band-limited to a frequency f_{max}, then you need to sample at $2*f_{max}$.

To design a filter using the 2920, you first determine the sample rate and the locations of the poles and zeroes of the filter. Given this list, you design one filter section to realize each real pole, and one for each complex conjugate pole pair. Most zeroes will be realized by adding them to one of the pole sections.

The gain of each section is determined, and inputs are scaled according to the needs of the design. The two types of filter sections are described below. It is assumed that the poles and zeroes location for the equivalent continuous analog filter have been determined.

Much of the design of such sections consists of picking values for B and G (see below) which best meet design goals yet which are easily realized in 2920 code. The two following examples illustrate the procedures involved.

Simulating Single Real Poles

Figure 4-1 shows a circuit which realizes a single real pole. A buffer amplifier is included to eliminate effects of loading. Proper choice of resistor, capacitor, and buffer amplifier gain determine the pole frequency and the stage gain characteristic.

Figure 4-2 shows a block diagram of an equivalent sampled realization. The block labeled Z^1 represents a unit delay, i.e., a delay equivalent to one sample interval or one 2920 program pass. The blocks labeled X represent multiplications, in each case by a constant. The block labeled Σ is an adder.

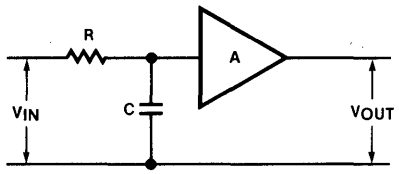


Figure 4-1. Continuous Realization of a Single Real Pole

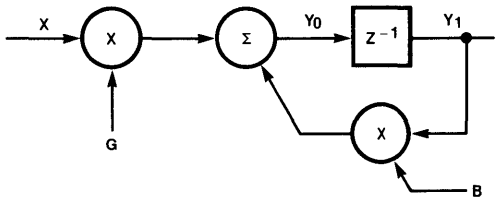


Figure 4-2. Sampled Realization of a Single Real Pole

The FORTRAN statements to implement Figure 4-2 would be as follows:

```
Y1 = Y0
Y0 = B * Y1 + G * X
```

These would be converted to 2920 statements as shown above in the section on arithmetic.

For example, if you have determined the B = 0.9922 (=0.11111110 in binary) and G = 0.0078125 (=0.0000010 in binary), the 2920 instructions could be generated as follows:

```
LDA   Y1,Y0,R00
LDA   Y0,Y1,R00   ; Y0 = 1.0 * Y1
SUB   Y0,Y1,R07   ; Y0 = B * Y1
ADD   Y0, X,R07   ; Y0 = B * Y1 + G * X
```

The comments indicate how the new value of Y0 is being generated. However, in this special case, the first two instructions are superfluous, and could in most cases be omitted.

Design Example 1

For a sample interval of 76.8 microseconds, realize a single-pole filter with a time constant (R*C) of 1.50 milliseconds ±1%, and DC gain of 1.00 ±1%.

The limits on B can be found from evaluating $B = e^{-T/RC}$ for the range 1.485 to 1.515, i.e., within the 1% tolerance (0.015) specified for RC: $1.485 \leqslant RC \leqslant 1.515$. Then −T/RC, using milliseconds for both, becomes −76.8/1485 to −76.8/1515, and B is thus in the range 0.94960 to 0.95057. Expressed in binary, $0.1111001100011000 \leqslant B \leqslant 0.1111001101011000$. The central value is B=0.95009 or 0.1111001100111001 in binary.

Any value in the specified range may be chosen and still meet the design criteria. If you choose a value of B=0.1111001101, rounding up, this meets the criteria and can be realized in five steps: $B = 2^0 - 2^{-4} + 2^{-6} - 2^{-8} + 2^{-10}$. This can be seen as follows:

$.1111 = 1.0 - 0.0001$, or $2^0 - 2^{-4}$

```
(   1.0000 )
( −0.0001 )
(= 0.1111 )
```

and

$0.0000\ 0011 = 0.0000\ 0100 - 0.0000\ 0001$, or $2^{-6} - 2^{-8}$

```
(   0.0000 0100 )
( −0.0000 0001 )
(= 0.0000 0011 )
```

In decimal this value is 0.950195. The effective time constant for this value of B can be derived as follows:

$\ln B = -T/RC$

$RC = -T/\ln B = -76.8/\ln(0.950195) = -76.8/-0.05108805 =$
1503.2869 microseconds, or 1.5033 milliseconds.

From the DC gain equation above,

$DC\ Gain = G/(1-B)$

note that $G = (1-B)$ ±1%. Given the value for B above, the range of acceptable values for G is 0.0504 to 0.04943, with a target value of 0.049805. Expressed in binary,

$0.0000\ 1100\ 1010 < G < 0.0000\ 1100\ 1110$, with a target of 0.0000 1100 1100.

This target value can be realized as easily as any of the others, in four steps:

$G = 2^{-4} - 2^{-6} + 2^{-8} - 2^{-10}$

With the two constants evaluated, the 2920 code is readily generated. Prior to evaluating the final 2920 code, you should consider overflow possibilities. If the input values are suitably limited, oveflow can be made impossible. In other cases, a proper sequence of instructions can at least limit overflow to the last instruction, so that saturation occurs only if the final value is too large. In the code generated below, terms have been ordered to prevent overflow from occurring on any but the last line.

The following sequence realizes the single pole section above. Comments show the contribution of instruction sequences.

```
LDA   Y1,Y0,R00    ; Y1 = Y0
LDA   Y0,X,R04
SUB   Y0,X,R06
ADD   Y0,X,R08
SUB   Y0,X,R10     ; Y0 = G * X
ADD   Y0,Y1,R04
SUB   Y0,Y1,R08
ADD   Y0,Y1,R10
ADD   Y0,Y1,R00    ; Y0 = G * X + B * Y1
```

Further Optimization for Single Poles

Some single pole stages could eliminate the first two LDA operations above by computing B * Y0 in place. If B can be expressed as a product of terms of the form $1+2^k$ or $1-2^k$, then B * Y0 can be computed using only the variable Y0. As an example, using the same problem statement and range for B from the example above, an acceptable value for B may be expressed as follows:

$$B = (1-2^{-4}) * (1+2^{-6}) * (1-2^{-9}) = 0.05029$$

The value for G computed above is still adequate, $G=2^{-4} - 2^{-6} + 2^{-8} - 2^{-10}$. The 2920 code for this problem now becomes:

```
SUB   Y0,Y0,R04
ADD   Y0,Y0,R06
SUB   Y0,Y0,R09    ; Y0 is now replaced with B*Y0
ADD   Y0,X,R04
SUB   Y0,X,R06
ADD   Y0,X,R08
SUB   Y0,X,R10     ; Y0 now equals G*X + B*Y0
```

One RAM location and three PROM words have been saved.

Simulating Complex Conjugate Pole Pairs

Figure 4-3 shows an RLC circuit which realizes a complex conjugate pole pair, while Figure 4-5 shows a sampled realization of the type used with the 2920. Again the blocks labeled X are multipliers, those labeled Z^{-1} are unit (one sample interval) delays, and the block labeled Σ is an adder. Coefficients B1 and B2 control the frequency parameters and G adjusts the overall gain.

Figure 4-4 shows the frequency response of this type of stage. The choice of parameter values determines both the frequency at which the gain peaks, and the height of sharpness of that peak.

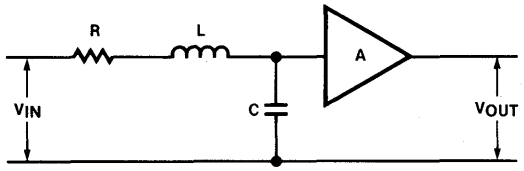


Figure 4-3. Continuous Realization of Complex Conjugate Pole Pair

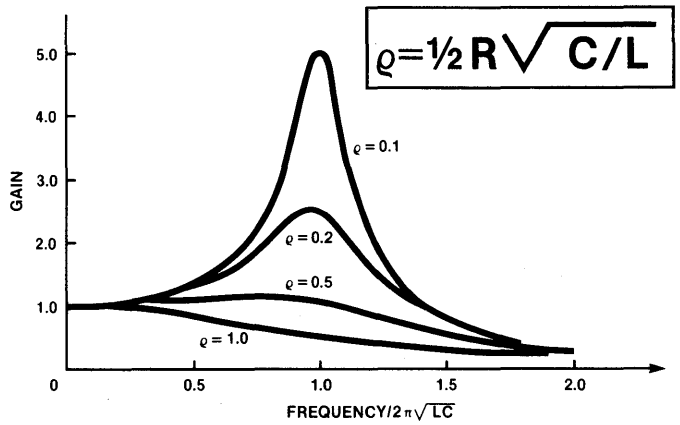


Figure 4-4. Gain of Complex Conjugate Pole Pair Section

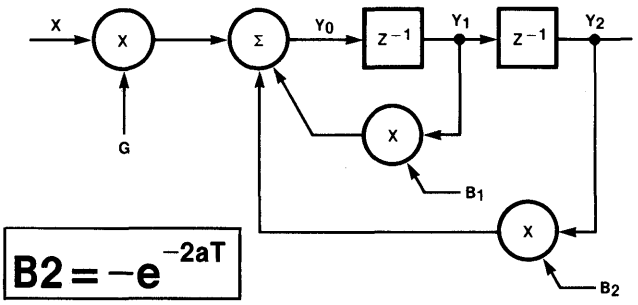


Figure 4-5. Sampled Realization of Complex Conjugate Pole Pair

The FORTRAN equations for a complex conjugate pole pair section are:

```
Y2 = Y1
Y1 = Y0
Y0 = B1*Y1 + B2*Y2 - G*X
```

Once the coefficients of the third equation are found, the equations can be converted to 2920 code using the procedures described above. Thus the major portion of the design task still consists of finding values for the coefficients which meet the design requirements, yet take the minimum number of 2920 steps to realize.

Design Example 2

For a sample interval of 76.8 microseconds, realize a resonance at 1000 Hz ±0.5% with a Q in the range $75 \leq Q \leq 100$. The peak gain should be 1.0 ±10%.

A complex conjugate pair of s plane poles at s=−a+jb and −a−jb has an impulse response which rings at a frequency f=b/2pi, and a value for Q given by Q=b/2a.

Thus bT = 0.48255 ±0.0024 and, at bT = 0.48255, aT falls in the range 0.002412 (Q=100) $\leq aT \leq$ 0.003217 (Q=75). Using $B2 = -e^{-2at}$, we can express the negative of B2 inbinary as follows:

$$0.1111\ 1100\ 1011 \leq -B2 \leq 0.1111\ 1110\ 1100$$

A value which falls in this range and can be expressed in only three powers of two is

$$-B2 = 0.1111\ 11101 = 2^{0} - 2^{-7} + 2^{-9} = 0.99414$$

Once B2 is established, B1 may be found using the relationships $e^{-aT} = -B2$, and $B1 = 2*\cos(bT)*e^{-aT}$. In binary, 1.1100 0100 11 $\leq$ B1 $\leq$1.1100 001110. A suitable value for B1 is given by $B1 = 1.1100\ 01 = 2^{-1} - 2^{-2} + 2^{-6} = 1.7656$.

To test the values of B1 and B2 chosen, the resonant frequency and Q may be calculated: $f_r = 1001.8$, Q = 82.

Maximum gain $= G_m = 1/((1-e^{-2at})* \quad 1-\cos^2(bT))$.

Substituting in the equations for maximum gain gives $f_m = 1001.8$, and maximum gain G_m as

$$G_m = G/0.002724$$

To meet the problem gain constraints, a value of G given by

$$G = 2^{-8} - 2^{-10} = 0.00293$$

is adequate.

The corresponding 2920 code can be written from the evaluations of the coefficients:

```
LDA   Y2,Y1,R00    ; Y2 = Y1
LDA   Y1,Y0,R00    ; Y1 = Y0
LDA   Y1,Y0,L01
SUB   Y0,Y1,R02
ADD   Y0,Y1,R06    ; Y0 = B1*Y1
SUB   Y0,Y2,R00
ADD   Y0,Y2,R07
SUB   Y0,Y2,R09    ; Y0 = B1*Y1 + B2*Y2
ADD   Y0,X,R03
SUB   Y0,X,R10     ; Y0 = B1*Y1 + B2*Y2 + G*X
```

The comments show how the values are built up from the sequences of 2920 instructions.

Overflow Considerations

If the inputs are scaled so that overflows in the calculation of Y0 are possible, a reordering of the terms may be necessary. At the third step above, a value of Y1 > 0.5 would produce overflow. Reordering the steps to add the 2*Y1 term last might reduce overflow probability. An alternate step would be to reduce the gain at the filter input and boost the filter output to compensate.

A variation on this method (gain reduction and boosting), is to generate a fraction of Y0 then boost the value of Y0 when shifting it into Y1. (The boost occurs on the next program pass). The fraction will usually be 1/2 or 1/4, and is accomplished by modifying the shift codes of all terms contributing to the Y0 calculation.

Simulation of Rectifiers

The absolute magnitude function, $Y = |X|$, can be realized with a single 2920 instruction (ABS). This function behaves as an idealized full-wave rectifier. The add absolute function (ABA) is useful for combining full-wave rectification with input to a filter.

Half-wave rectifiers can be realized using the equation $y = (x+|x|)/2$. The corresponding 2920 code for this operation is:

```
LDA    Y,X,R01    ; Y = X/2
ABA    Y,X,R01    ; Y = X/2 + ABS(X)/2
```

Other rectification characteristics may be simulated using piece-wise linear approximations, multiplication, or division.

Simulation of Limiters

Limiters may be realized in three ways using the 2920: via the LIM function, via overflow, or by calculations using absolute magnitudes (ABS,ABA).

The LIM function produces an ideal threshold logic element. Even the smallest signal forces a full positive or negative output.

In some systems, signals below some level should not be allowed to excite limiting. These systems require a transfer characteristic similar to that shown in Figure 4-6, where signals with amplitude below the threshold "a" do not produce full scale output. This type of limiter characteristic can be realized using overflow saturation or with the use of absolute magnitude functions.

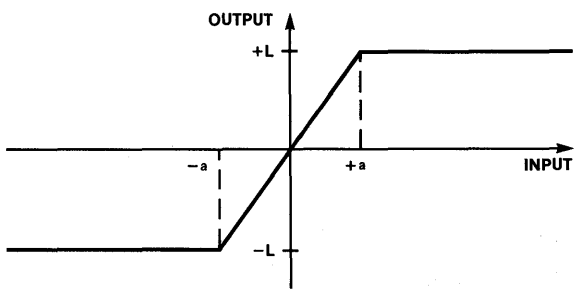


Figure 4-6. Limiter Transfer Characteristic

To use overflow saturation to implement such a limiter, the value X is loaded to Y with a left shift code, after which Y may be loaded or added to itself with additional left shifts. Consider this sequence:

```
LDA   Y,X,L02    ; Y = 4*X
ADD   Y,Y,L02    ; Y = 5*4*X = 20*X
```

The effect is to generate a value of Y which is 20 times X. If X exceeds a value of 0.05, Y will be held to +1.0, or if X ≤ −0.05, Y will take the value −1. Thus the characteristic realized is that of Figure 4-6 with a=0.05, and L=1.0.

Another realization can be based on the equation

$$y = |x+a| - |x-a|$$

which realizes the same shape curve as that of Figure 4-6, with a value of L=2a. This form generally takes more steps than the overflow saturation method, but allows greater freedom in setting parameters. The 2920 code might appear as follows, where A represents the limiter threshold and T is a location used only for intermediate calculations:

```
LDA   T,X,R00
ADD   T,A,R00    ; X+A in T
ABS   Y,T,R00    ; Y = ABS(X+A)
SUB   T,A,L01    ; X-A in T
ABS   T,T,R00    ; ABS (X-A) in T
SUB   Y,T,R00    ; Y = ABS(X+A) - ABS(X-A)
```

Other Signal Processing and Logic Functions

Many other signal processing functions can be performed by the 2920. Relaxation and gain controlled oscillators, and adaptive filters, are discussed in a 2920 Application Note.

Modulators can be realized using multiplication of a variable representing the carrier by a variable representing the modulating waveform. Automatic Gain Control (AGC) can be realized by dividing the signal by a level derived from signal magnitude.

Correlation functions involve delays, products, and filtering. The delay achievable is limited by the number of RAM words provided, but two or more samples may be packed in a word to increase the achievable delay. The AND operation is used as a mask to aid in unpacking such words.

Logical operations can be performed using the logical functions AND and XOR, by conditional arithmetic, or by using threshold logic, i.e., summation combined with the LIM function. In some cases, several logical variables can be stored in one RAM word.

CHAPTER 5
CONTROLS

Introduction

Since this Assembler uses the ISIS keyboard and file capabilities, ISIS must be loaded before invoking the Assembler. The full procedure for this is given in the ISIS manual named in the Preface. Once ISIS is present, you can enter the Editor to key-in the source text of your Assembler program. After developing and editing your program into a form ready to test, you can invoke the Assembler as described below.

The 2920 Assembler may be resident on the ISIS system diskette or on a non- system diskette. You load the assembler by entering a command that names the assembler and specifies the source file. You may also name the list and object files, but you don't have to. Control options may also be specified as part of the command.

After the assembler goes into execution, all assembler operations specified are performed without further intervention. If the invocation line has an error, the error is reported and you must retype the commands. You may use upper or lower case indiscriminately. The assembler converts all to upper case except for echoing back what you wrote.

Examples:

```
-AS2920 PROG.SRC
```

(After an ISIS prompt, shown here as a dash, you type the command to get the assembler to assemble your source program, which is in a file here called PROG.SRC. An assembly listing and object code file will be output to PROG.LST and PROG.HEX respectively. In addition, a symbol table listing will be supplied, and the symbol table debugging output to the object file is suppressed. These defaults are automatic when you do not specify any controls. It is exactly as if you had typed (on one line only)

```
-AS2920 PROG.SRC PRINT (PROG.LST) LIST OBJECT (PROG.HEX) SYMBOLS
        NODEBUG PAGING PAGEWIDTH(120) PAGELENGTH(66)
```

All but the last two options have opposites beginning with NO, like NODEBUG, whose opposite (however) is DEBUG. So you can say NOPRINT, NOLIST, NOOBJECT, NOSYMBOLS, or NOPAGING.

All such control options (except PRINT or NOPRINT) may be specified on the invocation line or on control lines (described below). If any control option is specified on the invocation line and also on a control line, the invocation specification takes priority and remains in effect. When a control is specified in different ways on the invocation line itself, the rightmost specification is used. (PRINT or NOPRINT can only appear on the invocation line.)

After running the one assembler pass and completing assembly listing and object output, the assembler outputs a sign-off message and summary:

```
ASSEMBLY COMPLETE
ERRORS     = XXXX
WARNINGS   = XXXX
RAMSIZE    = XXXX
ROMSIZE    = XXXX
```

Semantic Description

PRINT You get a list-file named like your source file, but with an extension of .LST, e.g. PROG.LST	NOPRINT The list-file is suppressed.
PRINT(filename.ext) You get a listing, put out to the file you name, using names that fit the ISIS rules, e.g. :F1:MYNEW.LST or :F0:TRYTWO.FIX or :PR: or :TO:	
LIST You get a listing of the code generated for each source line, sent to the list-file.	NOLIST The list file will contain only error messages and a summary of the assembly. (Unless use of the NOPRINT option has suppressed the list-file completely.)
OBJECT(filename.ext) You get executable code put to the file you specify.	NOOBJECT No object code is created.
DEBUG If OBJECT is specified, the symbol table is output to the object file.	NODEBUG Symbol table is not output to the object file. This is the default.
SYMBOLS Symbol table is output to the list file.	NOSYMBOLS Symbol table not listed.
PAGING Assembler will break the listing into pages with header lines on each.	NOPAGING No page breaks and no header lines after page 1.
EJECT Spaces are skipped to the next top-of-form if paging is specified.	
TITLE ('...') The character string specified (within the required parentheses and single-quotes) is printed on the second line of a page header. Strings of more than 64 characters are truncated to the first 64.	

PAGEWIDTH(number)

The number you supply specifies the maximum line width in characters, for listing output. It must be between 72 and 132; the default is 120. If you give a number outside these limits, the nearest limit will be used. If a listing line exceeds the pagewidth specified and is less than 133, it will be "wrapped around," with continuation beginning in column 20. Characters beyond column 132 will be truncated and lost.

PAGELENGTH(number)

This specifies the total number of lines per listing page. You have to count three blank lines at both top and bottom, and any header lines. The minimum pagelength is 15. The default is 66. Specified pagelengths are maintained by issuing a form feed to reach the top of the next page.

As implied by the discussion of the invocation line and defaults there, the following are used if no specification of an option is made:

```
PRINT(filename.LST)
LIST
OBJECT(filename.HEX)
NODEBUG
SYMBOLS
PAGING
PAGEWIDTH(120)
PAGELENGTH(66)
```

Control Records

A control record is a line in the source file which specifies any number of control options. Those contradicting a specification on the invocation line will cause an error message to be issued, and the new specification to be ignored, except for LIST, NOLIST, EJECT, or TITLE. A control line must begin with a dollar sign ($) and may have several options in it, separated by blanks. Commas are flagged as errors. If a control line has an error in it, the erroneous control setting and those to its right will be ignored. A control record containing PRINT or NOPRINT will be flagged as an error and ignored. If an option is not specified on the invocation line or any control line,its default is used.

Control records specifying LIST, NOLIST, EJECT, or TITLE may appear anywhere in the source file. The other control options allowed on control lines must appear before any source lines.

Example:

```
$EJECT TITLE ('FIRST TRY FILTER') LIST
```

APPENDIX A
EXAMPLE OF LISTING FORMAT

The following example shows the format of the listing output from an assembly, including error flags. The resulting hexadecimal object code is also shown. (These program-sections were designed to exercise all the execution-conflict error messages. They do not comprise a meaningful program.)

```
      1             ISIS-II 2920 ASSEMBLER V1.0

ASSEMBLER INVOKED BY: :F0:AS2920 EXECON.SRC


  LINE  LOC OBJECT SOURCE STATEMENT

      1             ; TEST ALL KNOWN 2920 ECECUTION CONFLICTS.
      2
      3             ; OUT FOLLOWS DAR AS DST.
      4
      5    0 4044EF LDA DAR,    RAM0
E     6    1 8000EF OUT0
      7    2 4000EF NOP
      8
      9             ; OUT FOLLOWS CND SUB.
     10
     11    3 8300FB SUB RAM1,   RAM2,   CND0
E    12    4 9000EF OUT1
     13    5 4000EF NOP
     14
     15             ; CYT FOLLOWS A CYT.
     16
     17    6 0100EF CYT0
E    18    7 1100EF CYT1
     19    8 4000EF NOP
     20
     21             ; CYT FOLLOWS DAR AS DST.
     22
     23    9 4044EF LDA DAR,    RAM0
E    24   10 0100EF CYT0
     25   11 4000EF NOP
     26
     27             ; CYT FOLLOWS CND SUB.
     28
     29   12 8300FB SUB RAM1,   RAM2,   CND0
E    30   13 1100EF CYT1
     31   14 4000EF NOP
     32
     33             ; CYT FOLLOWS A CYT.
     34
     35   15 0100EF CYT0
E    36   16 1100EF CYT1
     37   17 4000EF NOP
     38
     39             ; CYT FOLLOWS IN.
     40
     41   18 0000EF IN0
E    42   19 1100EF CYT1
     43   20 4000EF NOP
     44
     45             ; CND USED WITH AND, LIM, OR ABS.
     46
E    47   21 F708E3 AND RAM2,   RAM3,   CND7
     48   22 4000EF NOP
E    49   23 E700F5 LIM RAM3,   RAM2,   CND6
     50   24 4000EF NOP
E    51   25 DD00E7 ABS RAM2,   RAM8,   CND5
     52   26 4000EF NOP
```

```
  LINE  LOC OBJECT SOURCE STATEMENT

     1              ; TEST ALL KNOWN 2920 ECECUTION CONFLICTS.
     2
     3              ; OUT FOLLOWS DAR AS DST.
     4
     5    0 4044EF LDA DAR,    RAM0
E    6    1 8000EF OUT0
     7    2 4000EF NOP
     8
     9              ; OUT FOLLOWS CND SUB.
    10
    11    3 8300FB SUB RAM1,   RAM2,   CND0
E   12    4 9000EF OUT1
    13    5 4000EF NOP
    14
    15              ; CYT FOLLOWS A CYT.
    16
    17    6 0100EF CYT0
E   18    7 1100EF CYT1
    19    8 4000EF NOP
    20
    21              ; CYT FOLLOWS DAR AS DST.
    22
    23    9 4044EF LDA DAR,    RAM0
E   24   10 0100EF CYT0
    25   11 4000EF NOP
    26
    27              ; CYT FOLLOWS CND SUB.
    28
    29   12 8300FB SUB RAM1,   RAM2,   CND0
E   30   13 1100EF CYT1
    31   14 4000EF NOP
    32
    33              ; CYT FOLLOWS A CYT.
    34
    35   15 0100EF CYT0
E   36   16 1100EF CYT1
    37   17 4000EF NOP
    38
    39              ; CYT FOLLOWS IN.
    40
    41   18 0000EF IN0
E   42   19 1100EF CYT1
    43   20 4000EF NOP
    44
    45              ; CND USED WITH AND, LIM, OR ABS.
    46
E   47   21 F708E3 AND RAM2,   RAM3,   CND7
    48   22 4000EF NOP
E   49   23 E700F5 LIM RAM3,   RAM2,   CND6
    50   24 4000EF NOP
E   51   25 DD00E7 ABS RAM2,   RAM8,   CND5
    52   26 4000EF NOP
    53
    54              ; LIM WITH SHIFT CODE (OTHER THAN R00)
    55
E   56   27 460014 LIM RAM3,   RAM2,   R01
    57   28 4000EF NOP
    58   29 4600F5 LIM RAM3,   RAM2,   R00
    59   30 4000EF NOP
    60
    61              ; CND SUB AND DAR AS DST.
    62
E   63   31 814CEB SUB DAR,    RAM1,   CND0
    64   32 4000EF NOP
    65
    66              ; CYT AND DAR AS DST.
    67
E   68   33 0144EF LDA DAR,    RAM0,   CYT0
    69
B   70   34 5000EF EOP
    71
P   72   35 4000EF NOP
P   73   36 4000EF NOP
P   74   37 4000EF NOP
    75              END
```

```
ISIS-II 2920 ASSEMBLER V1.0


  LINE  LOC OBJECT SOURCE STATEMENT


SYMBOL:                          VALUE:

RAM0                                   0
RAM1                                   1
RAM2                                   2
RAM3                                   3
RAM8                                   4


ASSEMBLY COMPLETE
ERRORS    =     0
WARNINGS  =    17
RAMSIZE   =     5
ROMSIZE   =    38
:1600000F4F0F4F4FEFFF8F0F0F0FEFFF4F0F0F0FEFFF8F3F0F0FFFBD4
:1300180F9F0F0F0FEFFF4F0F0F0FEFFF0F1F0F0FEFFF1F1F0F0FEFFCC
:1300300F4F0F0F0FEFFF4F0F4F4FEFFF0F1F0F0FEFFF4F0F0F0FEFFAF
:1300480F8F3F0F0FFFBF1F1F0F0FEFFF4F0F0F0FEFFF0F1F0F0FEFF9D
:1300600F1F1F0F0FEFFF4F0F0F0FEFFF0F0F0F0FEFFF1F1F0F0FEFF8C
:1300780F4F0F0F0FEFFFFF7F0F8FEF3F4F0F0F0FEFFFEF7F0F0FFF556
:1300900F4F0F0F0FEFFFDFDF0F0FEF7F4F0F0F0FEFFF4F6F0F0F1F458
:1300A80F4F0F0F0FEFFF4F6F0F0FFF5F4F0F0F0FEFFF8F1F4FCFEFB2E
:1300C00F4F0F0F0FEFFF0F1F4F4FEFFF5F0F0F0FEFFF4F0F0F0FEFF1E
:0C00D80F4F0F0F0FEFFF4F0F0F0EFF9A
:00000001FF
```

APPENDIX B
KEYWORDS, INSTRUCTIONS, IOCODES, AND DIRECTIVES

Section 1: Arithmetic Operation Codes

ABA ABS ADD AND LDA LIM SUB XOR

Section 2: Analog Control Codes and Digital/Analog Register

CNDS	CVTS	EOP	IN0	NOP	OUT0
CND7	CVT7		IN1		OUT1
CND6	CVT6		IN2		OUT2
CND5	CVT5		IN3		OUT3
CND4	CVT4				OUT4
CND3	CVT3				OUT5
CND2	CVT2				OUT6
CND1	CVT1				OUT7
CND0	CVT0				

Section 3: Constant Source Codes

	KM1	KM2	KM3	KM4	KM5	KM6	KM7	KM8
KPO	KP1	KP2	KP3	KP4	KP5	KP6	KP7	
DAR								

Section 4: Scaler Control Codes

R00	R01	R02	R03	R04	R05	R06	R07	R08	R09	R10	R11	R12	R13
R0	R1	R2	R3	R4	R5	R6	R7	R8	R9				

L01	L02
L1	L2

Section 5: Assembler Commands and Modifiers

DEBUG	END	OBJECT( )	PAGING	SYMBOLS
NODEBUG	EQU	NOOBJECT	NOPAGING	NOSYMBOLS

PAGELENGTH( )	PAGEWIDTH( )	PRINT	PRINT( )
		NOPRINT	

EJECT	LIST	TITLE('...')
	NOLIST	

APPENDIX C
HEXADECIMAL OBJECT FILE FORMAT

All user programs loaded via the SM2920 module must conform to Intel's standard for hexademical object files, partly because the language translators generate only hexadecimal code.The hexadecimal object code file generated by the AS2920 assembler contains the contents of program memory which would result from loading the assembled source program. The code is formatted in hexadecimal bytes of data. The file contains the ASCII representation of the hexadecimal bytes of data. The object code itself is preceded by a symbol table. These two parts may be loaded or saved together or separately.

The symbol table is a series of records, terminated by a dollar sign. Each record contains three fields separated by one or more ASCII spaces:

- a number field (not used by SM2920)
- a label field containing the ASCII representation of a source program symbol, and
- an address field containing the hexadecimal address assigned to the symbol by the language translator.

The symbol table is terminated by a record whose first nonblank character is a dollar sign.

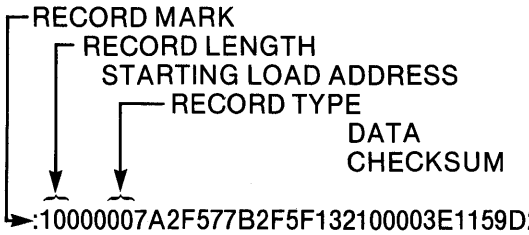
The object code generated by the language translator follows the symbol table. The symbol table has records. Each of these records or physical lines is six logical fields of varying length in characters or frames:

```
FIELD 0:  RECORD MARK (FRAME 0 IS ALWAYS ':')
FIELD 1:  RECORD LENGTH (FRAMES 1 AND 2)
FIELD 2:  LOAD ADDRESS FIELD (FRAMES 3,4,5 AND 6)
FIELD 3:  RECORD TYPE (FRAMES 7 AND 8)
FIELD 4:  DATA (FRAMES 9 TO 9+2*[RECORD LENGTH]-1)
FIELD 5:  CHECKSUM (FRAMES 'DATA FIELD' + AND 'DATA FIELD' +2)
```

For an example of the object file format see the sample program in Appendix A.

Object Code Output Format

The format of the object code is a series of records, each containing its record length, type, memory load address, checksum, and data. The figure following shows a typical output file in hexdecimal format.




```
:10000007A2F577B2F5F132100003E1159D21282
:1000000000E3E1F579174F7817477D176F7C176775
:0D002000F13DC20C00B77C1F577D1F5FC96A
:00000001
```

(Because record length equals 0 and record type equals 01, this record specifies end-of-file.)

APPENDIX D
POWERS OF TWO TABLE

POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

APPENDIX E
TABLE OF ASCII CHARACTER CODES

ASCII CODES

The 2920 assembler uses the seven bit ASCII code, with the high-order eighth bit (parity bit) always reset.

GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)
NUL	00
SOH	01
STX	02
ETX	03
EOT	04
ENQ	05
ACK	06
BEL	07
BS	08
HT	09
LF	0A
VT	0B
FF	0C
CR	0D
SO	0E
SI	0F
DLE	10
DC1 (X-ON)	11
DC2 (TAPE)	12
DC3 (X-OFF)	13
DC4 (~~TAPE~~)	14
NAK	15
SYN	16
ETB	17
CAN	18
EM	19
SUB	1A
ESC	1B
FS	1C
GS	1D
RS	1E
US	1F
SP	20
!	21
"	22
#	23
$	24
%	25
&	26
'	27
(	28
)	29
*	2A

GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)
+	2B
,	2C
–	2D
.	2E
/	2F
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
@	40
A	41
B	42
C	43
D	44
E	45
F	46
G	47
H	48
I	49
J	4A
K	4B
L	4C
M	4D
N	4E
O	4F
P	50
Q	51
R	52
S	53
T	54
U	55

GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)
V	56
W	57
X	58
Y	59
Z	5A
[	5B
\	5C
]	5D
Λ (↑)	5E
– (←)	5F
`	60
a	61
b	62
c	63
d	64
e	65
f	66
g	67
h	68
i	69
j	6A
k	6B
l	6C
m	6D
n	6E
o	6F
p	70
q	71
r	72
s	73
t	74
u	75
v	76
w	77
x	78
y	79
z	7A
{	7B
|	7C
} (ALT MODE)	7D
~	7E
DEL (RUB OUT)	7F

APPENDIX F
BIT PATTERNS OF THE 2920 ASSEMBLY LANGUAGE MNEMONICS

Instruction Field Bit Assignments

The instruction word for the 2920 is 24 bits long and is divided into six four-bit nibbles:

Nibble	MSB			LSB
0	ADF0	ADK2	ADK1	ADK0
1	A2	B1	A1	ADF1
2	A4	B3	A3	B2
3	A0	B5	A5	B4
4	S2	S1	S0	B0
5	L2	L1	L0	S3

where	L0 -	L2	is the opcode,
	A0 -	A5	is the source address,
	B0 -	B5	is the destination address,
	S0 -	S3	is the shiftcode,
	ADF1 -	ADF2	
	ADK1 -	ADK3	is the iocode.

Opcode Field

L2 L1 L0	Mnemonic
000	XOR
001	AND
010	LIM
011	ABS
100	ABA
101	SUB
110	ADD
111	LDA

Source and Destination Fields

The destination operand is a six-bit address pointing into RAM. Each address bit is located as follows:

ADDR BIT	0 (LSB)	B0
	1	B1
	2	B2
	3	B3
	4	B4
	5 (MSB)	B5

The source operand is also a six-bit address pointing into RAM. Each address bit is located as follows:

ADDR BIT	0 (LSB)	A0
	1	A1
	2	A2
	3	A3
	4	A4
	5 (MSB)	A5

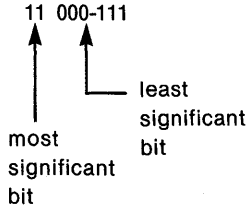
Shift Code Field

S0 S1 S2 S3	Mnemonic
1100	R13
1011	R12
1010	R11
1001	R10
1000	R09
0111	R08
0110	R07
0101	R06
0100	R05
0011	R04
0010	R03
0001	R02
0000	R01
1101	L01
1110	L02
1111	R00

Input/Output Code Field

The iocode is located in bits 23-19 and is encoded in the following manner:

ADF0 ADF1/ADK0 ADK1 ADK2	Mnemonic
00 000	IN0
00 001	IN1
00 010	IN2
00 011	IN3
00 100	NOP
00 101	EOP
00 110	CVTS
00 111	CNDS
10 000-111	CVT0—CVT 7
01 000-111	OUT0—OUT 7
11 000-111	CND0—CND 7



APPENDIX G
ERROR HANDLING AND REPORTING

Command Language and Run-Time Errors

COMMAND SYNTAX ERROR	This message means one of the following conditions has been directed: illegal syntax, missing or illegal delimiter use, or a missing required parameter.
PREMATURE EOF	An end-of-file was encountered before an END directive. This is an unrecoverable error condition, causing the Assembler to terminate abnormally.

Syntax Errors, Control Record Errors, and Semantic Warnings

These errors are indicated by single letter codes which appear on the same line of the listing as the source line in which they were found. When multiple errors are detected in a single source line, only the first error is reported. A summary of syntax errors is output to the console and list devices.

C	An illegal control option or placement.
I	Illegal ASCII character; possibly a missing linefeed after a carriage-return.
M	Multiple definition of symbols, which must be unique in the first 31 characters.
O	Illegal operation code or operand.
S	Invalid syntax, usually due to invalid or missing opcode.
	Semantic Warning Flags:
B	Boundary error. EOP does not fall on legal address, i.e., a multiple of four.
E	Execution conflict: e.g., a CVT on or immediately after using the DAR as a destination, or right after an input; or a conditional subtract with the DAR as a destination; or an attempt to use a conditional iocode on an AND, LIM, or ABS, or an attempt to use LIM with a shiftcode other than R0.
P	Pad insertion, putting NOPs after the EOP.
V	Illegal value specified for an operand with a limited range.

APPENDIX H
SYNTAX OF 2920 ASSEMBLY LANGUAGE

The following is a formal definition of the language accepted by the 2920 Assembler. Upper-case character strings in this description represent terminal symbols used exactly as shown. Lower-case strings represent metalinguistic variables which are either defined here or self-evident. Punctuation shown after the string ::=, e.g. commas, must be used as and where shown. Blanks shown between variables or terminals, e.g. in the EQU directive, are needed as separators.

program	::= stmt_list end_stmt
stmt_list	::= stmt / stmt_list stmt
stmt	::= basic_stmt nl
nl	::= CR / comment CR
basic_stmt	::= optional_label machine_inst / assemb_directive / empty
optional_label	::= name: / empty
machine_inst	::= opcode dest,src optional_shiftcode optional_iocode / iocode
optional_shiftcode	::= ,shiftcode / empty
optional_iocode	::= ,iocode / empty
assemb_directive	::= name EQU name
opcode	::= XOR / AND / LIM / ABS / ABA / SUB / ADD / LDA
shiftcode	::= R00 / R01 / R02 / R03 .../ R13 / L01 / L02 / R1 / R2 / R3 .../ R9 / L1 / L2
iocode	::= IN0 / IN1 / IN2 / NOP / EOP / CVTS / CVT7 / CVT6 / CVT5 .../ .../ CVT0 / CNDS / CND7 / CND6 .../ CND0
dest	::= name
src	::= name
name	::= letter / special_char / name letter / name special_char /name decimal_digit
special_char	::= @ / ? / _
comment	::= ; / ; ascii_characters
letter	::= A / B / C / D / E / F / G / H / I / J / K / L / M / N / O / P / Q / R / S / T / U / V / W / X / Y / Z
decimal_digit	::= 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9
end_stmt	::= END

Notes: The slash means you may choose among the items separated. "Empty" means the entry can be omitted entirely. Terminal symbols shown in upper case are also accepted from the keyboard in lower-case.

APPENDIX I
TWO'S COMPLEMENT DATA HANDLING IN THE 2920

Data in the 2920 are stored using a two's complement binary form. Using this form, the highest order bit indicates the sign of the value, with this bit being zero (0) for positive and zero values, and one (1) for negative values. If the intended value is positive, the remaining bits correspond to that value, independent of the sign bit. If the intended value is negative, then the remaining bits correspond to the number (one minus that value).

A convention used with the 2920 places an imaginary binary point just to the right of the highest order (25th) bit, as shown below:

1.011010 111001 111110 000100

Each bit to the right of the binary point has a positive fractional weight associated with it, the first having the value $2^{-1} = \frac{1}{2}$, the second $2^{-2} = \frac{1}{4}$, and so on. If x is the number represented by the bits to the right of the binary point, then $0 \leqslant x < 1.0$. If s represents the sign bit (0 for non-negative values, 1 for negative), then the full 25-bit number represents the value $-s + x$.

Two's complement arithmetic is used because it allows relatively simple hardware realizations of arithmetic functions. Addition in two's complement follows normal binary addition rules, and can be realized using standard adder building blocks. If two numbers of like sign bit are added and the sign bit of the result differs from that of the original operands, the result is too large in magnitude to be contained within the allotted number of bits. In this case an "overflow" is said to have occurred.

Subtraction in two's complement arithmetic may be done by adding the two's complement of the subtrahend. The two's complement of a number is formed by first taking the one's complement and then adding a 1 in the lowest order position. The one's complement is formed by complementing all bits in place, i.e., replacing all original zeroes with ones, and all original ones with zeroes. (Note that the number -1.0 has no valid two's complement in the 25-bit number system used.) In practice, subtraction is accomplished by adding the one's complement, and forcing a carry input into the lowest order adder stage—which is equivalent to adding a 1 in the lowest order position.

Using two's complement arithmetic therefore simplifies addition and subtraction as compared with sign/magnitude representation in that no sign bit testing of either operand is necessary to set up for addition or subtraction. Only one set of adders is needed because the conversion from one's complement to two's complement can be achieved within the adder.

Multiplication and division by powers of two corresponds to shifts left or right respectively. When shifting left, the low order bit is filled with zeroes and when shifting right, the high order bit is filled with the sign bit. To extend precision to the left, the sign bit is extended into each added position before any shift operations are done. The sign bit behaves as if it extends to the left on to infinity. Overflow corresponds to the case where the recorded sign bit does not correspond to the sign bit at infinity.

In the 2920, arithmetic is performed with a left extension to a total of 28 bits, adequate to perform any 2920 operation without possibility of overflow. Thus the highest order bit corresponds to the sign bit at infinity. If the storable portion of the result (low 25-bits) does not correspond to the correct result, an overflow is

indicated, and if overflow limiting is enabled, the 25-bit value stored is the positive extremum (if the correct sign bit was 0) or the negative extremum (if the correct sign bit was 1).

positive extremum = 0.1111 1111 1111 1111 1111 1111 = approx. +1.0

negative extremum = 1.0000 0000 0000 0000 0000 0000 = −1.0

In two's complement arithmetic, multiplication can be performed in a manner similar to that used for positive binary numbers. However, because the sign bit has a negative rather than a positive weight, some additional corrections are needed.

Multiplication in the 2920 may be achieved using the conditional add, with the multiplier being loaded into the DAR, and the multiplicand being conditionally added to the (partial) product. Because adds in the 2920 provide for sign extension during shifting, a positive multiplier can produce a correct product without any further correction, shown in the examples below.

```
  1111.11 (−¼)
x 0000.11 (+ ¾)
...1111.111
. ..1111.1111
...11111.1101 = −1 + 13/16 = −3/16
```

Note that in each case, the sign bit was extended to the left in the partial products. The example shown above is drawn in a manner different from that used in grade school arithmetic classes. The somewhat different display results from noting that each bit of the multiplier to the right of the binary point has a weight equal to some negative power of 2, i.e., is equivalent to a right shift of one or more positions.

```
  0.1100 (3/4)
x 0.1101 (13/16)
 0000.01100
 0000.001100
 0000.0000000
 0000.00001101
00000.10011101
```

Thus if the multiplication is done starting at the binary point of the multiplier, and running through the multiplier from left to right, the binary point can be maintained (and aligned) for each partial product. Each bit of the multiplier corresponds to a possible addition of the multiplicand, shifted to the right by one or more positions, to the product. If the multiplier bit is a 1, the addition takes place, otherwise it does not take place.

If the sign bit of the multiplier is non-zero, because this bit has a negative weight the multiplicand should be subtracted from the product. In the 2920 this function is achieved by complementing the multiplicand, and conditionally adding the resulting complement to the product based on the sign bit of the multiplier.

Division tends to be complex in two's complement arithmetic, and so may be simplified by extracting the signs of the operands, performing the division using only the magnitudes of the dividend and divisor. The quotient is converted to the proper sign based on the extracted signs.

Restoring binary division is performed using a series of test subtractions of the divisor from the dividend, with the original value restored if the result becomes negative. The sequence of test subtractions proceeds from left to right, with each successful subtraction (one leaving a positive difference, thus not requiring restora-

tion) reducing the magnitude of the dividend. The locations of the successful subtractions are noted by ones, those unsuccessful by zeroes, in the DAR.

Restoration of the dividend corresponds to adding the divisor back to the dividend. Because this operation is followed by a test subtraction with the divisor shifted one position further right, the restoration/test subtraction sequence can be replaced by a single addition of the divisor after it is shifted to the right. (As a right shift is equivalent to a multiply by 1/2, the first sequence is $+d-d/2 = d/2$; the second operation is $+d/2$.)

In the 2920, the conditional subtract operation is used to perform this non- restoring divide algorithm. For any arithmetic operation, the high order carry from the extended arithmetic is saved for possible testing by the conditional subtract instruction. This carry has the same value as the sign of the result generating it, i.e., 1 for negative, 0 for non-negative numbers. The conditional subtract performs the addition or subtraction required by the previous result (i.e., carry), and then stores the new result of the operation in a designated location (bit) of the DAR as selected by the condition code used.

APPENDIX J
DISCUSSION OF CARRY AND OVERFLOW CONDITIONS

The detailed ramifications of carry and overflow are discussed in this appendix. The tables show the three major forms and the possible cases in each.

Overflow occurs when ALU operations produce numbers outside the legal range of $-1.0 \leqslant X < +1.0$.

Normal standard carry logic applies to the ALU instructions ADD, SUB, ABA, LIM, AND. For the instructions XOR, ABS, and LDA, the carry logic includes additional considerations.

Normal Carry and Overflow

The 2920 standard representation of data is a signed 25-bit binary fraction. Positive data can be considered simply 24-bit fractions with a sign bit, e.g., 0.100000000000000000000000 means +1/2. Negative data have a sign bit of 1 with the remaining 24 bits representing the two's complement of the value, i.e., one minus that value. An example:

1.010000000000000000000000 means −3/4.

However, the capability to shift left two positions makes it necessary to allow for a 26th and 27th bit for the sign. A 28th bit is necessary to preserve the sign in the case of carry information if two numbers are left-shifted and then added.

Therefore, the 2920 logic carries 28 bits, four bits to the left of the imaginary binary point and 24 bits to its right:

ssss.bbbb bbbb bbbb bbbb bbbb bbbb

If the source operand is to be negated during an instruction, then before the indicated operation is carried out, the one's complement of the source is formed by complementing all its bits and setting the carry-in bit to one. This happens in three circumstances: in taking the absolute value of a negative number, in an unconditional subtraction, or in a conditional subtraction when the prior carry was 0.)

Standard carry, then, is propagated to the left, beginning at the least-significant (right-most) bit and continuing into the sign bits if necessary. Carry into the sign bits may mean an overflow condition, since in overflow the four sign bits become unequal. The leftmost bit of the source operand always retains the original sign even if shifting occurs.

Normal practice is to keep the numbers scaled between −1.0 and +1.0, such that the arithmetic operations do not create values outside this range. If out-of-range values do result, this is an overflow situation.

Conditional iocodes do not affect carry and overflow for standard-carry instructions (except conditional subtract). The calculation acts as if a straight add were being done: if a carry into the sign bit occurs, then the carry flag is set. (Subtraction is performed as an add after taking the two's complement of the source operand and setting carry-in to 1.)

LIM produces a +1.0 or a −1.0 using the sign of the source only, and sets the carry to 0. Overflow for LIM depends on whether a left-shift occurred. When overflow limiting is enabled and an overflow condition occurs on an ADD, ABS, or ABA instruction, the result is limited, i.e., becomes −1.0 or +1.0 (This never applies to AND.)

Nonstandard Carry and Overflow

There are three instructions in this group: ABS, LDA, and XOR.

For ABS and LDA, carry is set to 0. The implicit value of the destination is initially forced to zero. Overflow limiting is on. There can be overflow only if the source was left shifted, possibly making the four sign bits unequal.

A conditional iocode on an LDA can turn the instruction into a no-operation if the bit tested in the DAR is 0. Conditional iocodes on an ABS instruction get a warning (only) from the Assembler.

The case of XOR is complicated by the potential of left shifts. The table shows these extra cases. Use of a conditional iocode on an XOR causes overflow limiting to be turned on. (So does an EOP. ABA with a conditional iocode turns it off.) Limiting will occur if overflow limiting is enabled.

Table of Carry and Overflow Cases

OPER.	SIGN OF DESTINATION CONTENTS	SIGN OF SOURCE CONTENTS	CARRY BIT	
	ADD			
ADD	positive,	positive	0	
ADD	negative,	negative	1	
ADD	positive,	negative	***	
ADD	negative,	positive	***	
	SUB			
SUB	positive,	positive	***	
SUB	negative,	negative	***	
SUB	positive,	negative	***	
SUB	negative,	positive	1	
	ABA			
ABA	positive,	positive	0	
ABA	negative,	negative	1	
ABA	positive,	negative	0	
ABA	negative,	positive	***	
	XOR			**OVERFLOW BIT**
XOR	positive,	positive	0	0 unless 1 left-shifted into sign bits
XOR	negative,	negative	1	1
XOR	positive,	negative	0	0 unless shift
XOR	negative,	positive	0	0 with no shift
			1	0 if shift only ones
			1	1 all other cases

(*** means "if the result is positive, carry is set to 1; otherwise, 0")

INDEX

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.