uPD7720A/77P20 Design Manual
TABLE OF CONTENTS

LIST OF FIGURES

FIGURE NO.

LIST OF TABLES

TABLE NO.

1. SUMMARY DESCRIPTION

Fabricated in high-speed NMOS, the uPD7720A Signal Processing Interface (SPI) is a complete 16-bit microcomputer on a single chip. ROM space is provided for both program and data/coefficient storage, while the on-chip RAM may be used for temporary data, coefficients, and results. The uPD7720A is a masked-ROM device (user code and data are programmed into the instruction and data ROM areas during fabrication by NEC). The uPD77P20 is a UV-EPROM version of the uPD7720A that is functionally identical to the uPD7720A (power requirements are different). The uPD77P20 is used for prototyping and for small production runs. Computational power is provided by a 16-bit Arithmetic/Logic Unit (ALU) and a separate 16 x 16-bit fully parallel multiplier. This combination allows the implementation of a "sum of products" operation in a single 250 nsec instruction cycle. In addition, each arithmetic instruction provides for a number of data movement operations, as well as pointer modifications, to further increase throughput. Two serial I/O ports are provided for interfacing to codecs, successive-approximation A/D converters, and other serially-oriented devices while a parallel port provides both data and status information to a conventional microprocessor for more sophisticated applications. Handshaking signals, including DMA controls, allow the SPI to act as a sophisticated programmable peripheral as well as a stand alone microprocessor.

Development tools consist of an assembler, a simulator, an in-circuit emulator (EVAKIT), and the EPROM version, uPD77P20. See the corresponding manuals and data sheets for more detailed information on these tools.

APPLICATIONS	Digital Filtering	
	High Speed Data Modems	
	Fast Fourier Transforms (FFT)	
	Speech Synthesis and Analysis	
	Dual-Tone Multi-Frequency (DTMF) Transmitters/Receivers	
	Equalizers	
	Adaptive Control	
	Numerical Processing	
PERFORMANCE BENCHMARKS	Second Order Digital Filter (BiQuad)	2.25 us
	u/A Law to Linear Conversion	0.50 us
	SINE/COS of Angles	5.25 us
	FFT: 32-Point Complex	0.7 ms
	64-Point Complex	1.6 ms

1.1 Features

Fast Instruction Execution - 250 ns/8 MHz Clock

16-bit Data Word

Multi-operation Instructions for Optimizing Program Execution

Large Memory Capacities:

Program ROM	512 x 23 bits
Data/Coefficient ROM	510 x 13 bits
Data RAM	128 x 16 bits

Fast (250 ns/8 MHz) 16 x 16 Parallel Multiplier with 31-bit Result

Four-Level Subroutine Stack for Program Efficiency

Multiple I/O Capabilities
- Serial - Separate input and output (8 or 16-bit)
- Parallel - 8-bit bus
- DMA

Compatible with most Microprocessors, Including:
- uPD8080
- uPD8085
- uPD8086
- uPD780 (Z80)

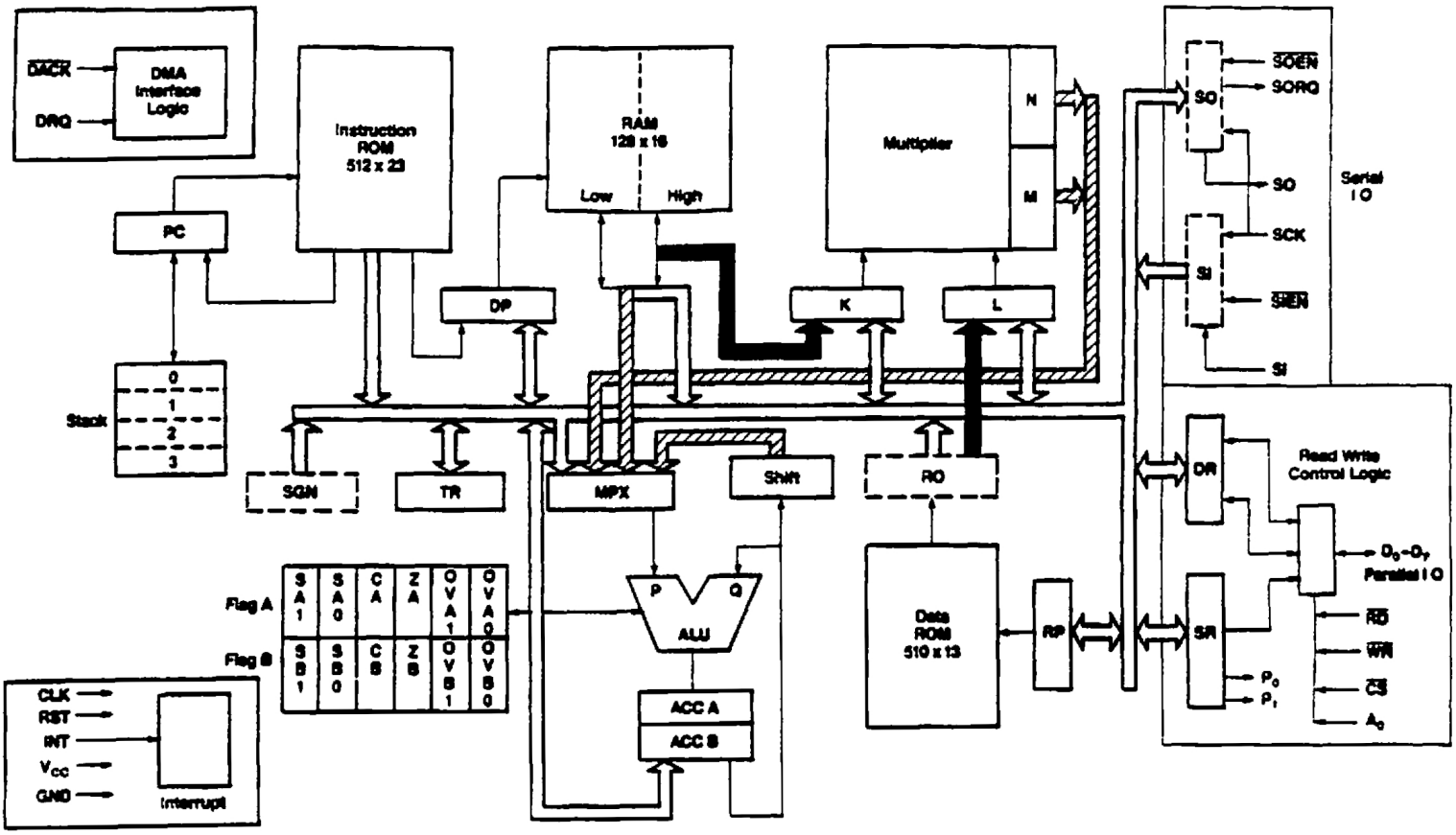
+5V Power Supply

NMOS Technology

28-pin DIP Package

Refer to the uPD7720A/P20 data sheet for AC, DC, and timing specifications.

1.2 Block Diagram



The primary bus (unshaded) makes a data path between all of the registers (including I/O), memory, and processing sections. This bus is referred to as the IDB - Internal Data Bus. The multiplier input registers K and L can be loaded not only from the IDB but alternatively (via dark buses) directly from RAM to the K register and directly from data ROM to the L register. Output from the multiplier in the M and N registers are typically added (via the shaded buses) to either accumulator A or B as part of a multi-operation instruction.

1.3 Instruction Summary

The SPI has three instruction types:

1. Load Immediate
2. OP/RT (multi-operation)
3. Jump - (conditional, unconditional, or call)

Instructions are described in detail in a later section.

1. The LDI (Load Immediate) instruction loads a 16-bit value (immediate data) to one of thirteen possible destination registers.

2. The OP/RT multi-operation instruction can perform (optionally) any or all of the following operations in a single instruction:

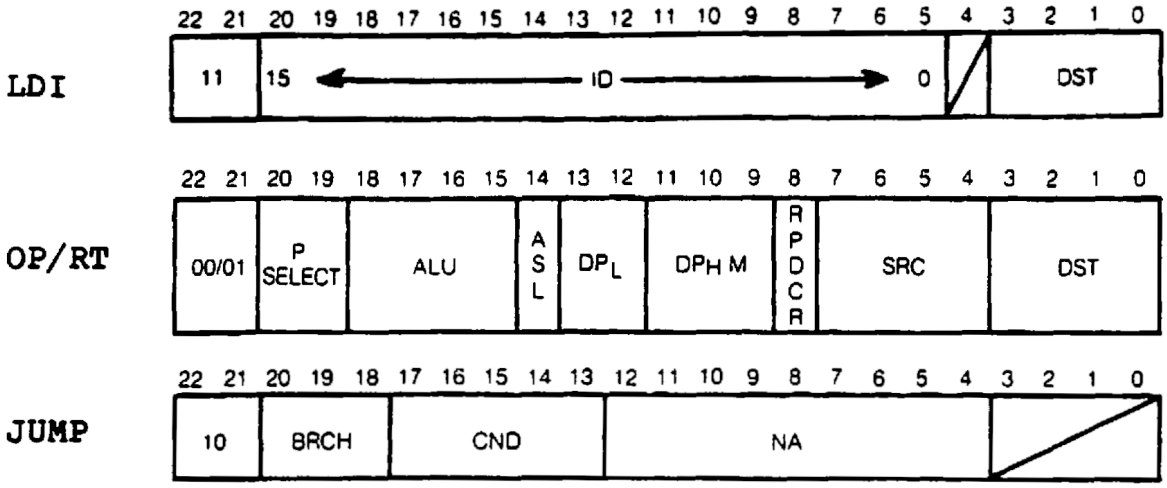
a) move data from one register to another
b) implement an ALU operation on either accumulator, using an operand from one of four inputs (including the data being moved by a))
c) modify lower four bits (column pointer) of data RAM pointer
d) modify high three bits (row pointer) of data RAM pointer
e) decrement data/coefficient ROM pointer
f) return from subroutine

While any or all of the above is happening, the contents of two 16-bit registers are multiplied, and their 31-bit product is placed in the multiplier output registers.

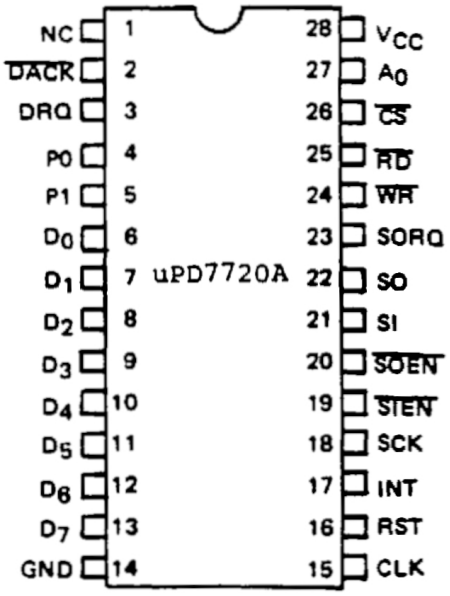
3. The Jump instruction has three variations:

a) unconditional jump
b) conditional jump (32 types of conditions)
c) subroutine call

FIGURE 1.3 INSTRUCTION FIELD SUMMARY



2. PIN CONFIGURATION



2.1 Pin Identification

PIN	NAME	I/O	FUNCTION
1	NC/Vpp	I	No Connection /Vpp For uPD77P20
2	$\overline{DACK}$	I	DMA Request Acknowledge. Indicates to the uPD7720 that the Data Bus is ready for a DMA transfer. $\overline{DACK}$=0 is equivalent to A_0=0 and $\overline{CS}$=0
3	DRQ	O	DMA Request signals that the uPD7720 is requesting a data transfer on the Data Bus.
4,5	P_0, P_1	O	P_0, P_1 are general purpose output control lines.
6-13	D_0-D_7	I/O Tristate	Port for data transfer between the Data Register or Status Register and Data Bus.
14	GND		
15	CLK	I	Single phase Master Clock input
16	RST	I	Reset initializes the uPD7720 internal logic and sets the PC to 0.
17	INT	I	Interrupt. a low to high transition on this pin will (if interrupts are enabled by the program) execute a call instruction to location 100H. (Edge sensitive)
18	SCK	I	Serial Data Input/Output Clock. A serial data bit is transferred when this pin is high.
19	$\overline{SIEN}$	I	Serial Input Enable. This line enables the shift clock to the Serial Input Register.
20	$\overline{SOEN}$	I	Serial Output Enable. This pin enables the shift clock to the Serial Output Register.
21	SI	I	Serial Data Input. This pin inputs 8 or 16 bit serial data words from an external device such as an A/D converter.
22	SO	O	Serial Data Output. This pin outputs 8 or 16 bit data words to an external device such as an D/A converter.
23	SORQ	O	Serial Data Output Request. Specifies to an external device that the Serial Data Register has been loaded and is ready for output. SORQ is reset when the entire 8 or 16 bit word has been transferred.
24	$\overline{WR}$	I	Write Control Signal writes the contents of data bus into the Data Register.
25	$\overline{RD}$	I	Read Control Signal. Enables an output to the Data Port from the Data or Status Register.
26	$\overline{CS}$	I	Chip Select. Enable data transfer with Data or Status Port with $\overline{RD}$ or $\overline{WR}$.
27	A_0	I	Select Data Register for Read/Write (low or Status Register for read (high).
28	V_{CC}		+5V POWER

3. FUNCTIONAL DESCRIPTION

3.1 Instruction ROM

The 512 x 23 bits of mask-programmable instruction ROM are addressed by a 9-bit Program Counter which can be modified by an external reset, interrupt, call/jump, or return instruction.

3.2 Program Counter

The Program Counter (PC) is a 9-bit binary counter which functions as follows:

PC is incremented by 1 at each instruction fetch.
The contents of the address field (NA field) of any of the following instructions is transferred to PC when executed:

* JMP instruction
* Conditional jump instructions (if condition is met)
* CALL instruction (and PC is pushed on to the stack)

The contents of the STACK is transferred to PC when an RT (subroutine or interrupt Return) instruction is executed.
The interrupt address (100H) is transferred to PC whenever an interrupt condition occurs, and the PC is pushed onto the stack.

3.3 Stack

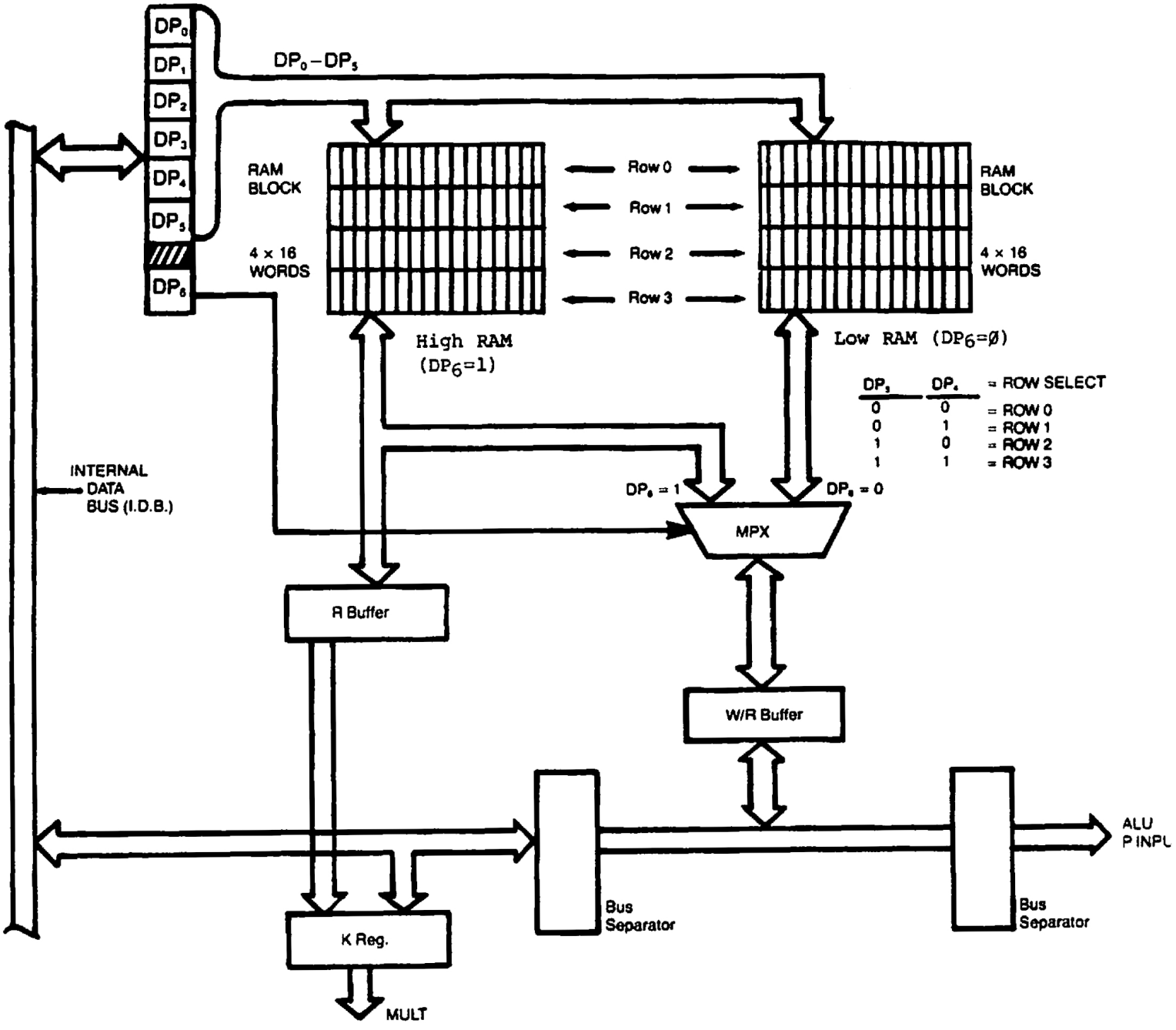
The SPI contains a 4-level program stack for efficient program usage and interrupt handling. The stack is a 4 x 9-bit LIFO register (last in first out). It stores the return addresses for subroutines and interrupts. The return address is read out of the stack and transferred to PC when the Return instruction is executed. Subroutines (or interrupts) may not be nested more than four (4) levels deep because the return addresses are pushed out the bottom of the stack and lost.

3.4 RAM

The Data RAM has 128 16-bit words and is addressed by a 7-bit Data Pointer (DP) register. The DP has addressing features that operate simultaneously with arithmetic instructions so that no added time is taken for address modification.

The Data RAM is best thought of as two blocks of memory, each with four rows of 16 words, as shown in Figure 3.1. The DP register's MS bit (DP6) selects which block will be input/output from the MPX. DP5 and DP4 select the row within a block to be accessed and DP3-DP0 (DPL) select the value (or column number) within the row. In addition to (DP6 = 1) being able to route its data through the MPX, the high order block may also send its data directly to the K register (Multiplier Input Register) via its own dedicated bus as part of an OP/RT instruction. To do this requires the use of the @KLM Destination, in which the L register is loaded through the IDB (Internal Data Bus), from the Source register specified in the instruction.

FIGURE 3.1 BLOCK DIAGRAM OF RAM AND DP CONNECTIONS



3.5 Data Pointer (DP) Register

The Data Pointer is a seven-bit register that specifies the RAM address. The three higher bits are referred to as DPH and the four lower bits as DPL.

When data is loaded from the internal bus to DP, the lower seven bits are input and the nine higher bits are ignored (IDB is a 16-bit bus). When data is output from the DP to the internal data bus, the seven lower bits contain the DP value, and the nine higher bits of the data bus are filled with zeroes.

The Data Pointer can be modified as part of an OP/RT instruction as described below.

The three higher bits of the DP can be modified by being XOR'ed with the three bits in the DPH-M field of the OP/RT instruction (M0 through M7 in assembly language mnemonics).

EXAMPLE:	DP/BIT	654	3210		DP/BIT	654	3210
	DP =	000	XXXX		DP =	001	XXXX
	M1 = XOR	001			M1 = XOR	001	
	DP =	001	XXXX		DP =	000	XXXX

The four lower bits of the DP can be modified by the DPL field of an OP/RT instruction in the following ways:

1. Increment DPL - MODULO 16 (0-15) = DPINC
2. Decrement DPL - MODULO 16 (15-0) = DPDEC
3. Clear DPL = DPCLR
4. No Change (NOP) = DPNOP

Both parts of the DP can be modified simultaneously by an OP/RT instruction, however these operations are ignored if a DP load from the Internal Data Bus (IDB) is specified as part of the same instruction (Destination = @DP).

The DP value resulting from modifications as part of an OP/RT instruction is not implemented until the end of the instruction cycle, and therefore does not affect the DP value used as part of that same instruction, but is effective for use by the next instruction.

For more detailed information on the use of the DP modifications in practical applications, see Appendix 1.

3.6 Data/Coefficient ROM (510 x 13 bits in uPD7720A, 512 x 13 bits in uPD77P20)

The Data/Coefficient ROM is organized as 510 x 13 bits and is addressed by a 9-bit ROM pointer (RP) Register. This ROM is ideal for storing any coefficients, conversion tables, and constants that may be needed for processing. All ROM locations may be used with the exceptions of addresses 0 and 1, which are used for test patterns in the uPD7720A masked-ROM version. Locations 0 and 1 may be used (512 x 13 bits total) in the uPD77P20 EPROM version. However, use of these locations is discouraged to avoid conflict if and when the code is committed to masked-ROM.

The ROM's output buffer (RO) data may be routed two places.

First, it may be routed to the Internal Data Bus (IDB), using the RO source specification in the MOV portion of an OP/RT instruction, through which it may be sent to any of the possible destinations. Secondly, it may be routed directly to the L register (multiplier input) via a special bus by use of the @KLR destination specification, in either the MOV portion of an OP/RT instruction, or as the destination of an LDI instruction. In this case, the K register (multiplier input) is loaded through the IDB from either the specified source register or the immediate data field of the LDI instruction. In either case, the 13 bits are placed in the higher order bits of the output word (16-bit wide data word) and the lower order 3 bits are always zero.

3.7 RP (ROM Pointer)

The ROM Pointer is a 9-bit register that specifies the Data ROM address. It is used in much the same manner as the DP register.

When data is input to the RP register from the internal data bus (IDB), the nine lower bits are stored and the seven higher bits are ignored. When data is output from the RP register to the internal data bus, the RP data fills the lower nine bits of the data bus, and zeroes are output to the seven higher bits.

The RP register can be decremented as part of an OP/RT instruction by the use of the RPDCR field. The RP value that results from modification as part of an OP/RT instruction is not implemented until the end of the instruction cycle, and therefore does not affect the RP value used as part of that same instruction, but is effective for use by the next instruction. The RP modification is ignored if a value is being input from the IDB as part of the same instruction (Destination = @RP).

3.8 Multiplier

This is a fully parallel multiplier that uses the secondary Booth algorithm. It multiplies two 16-bit words (taken from the K and L registers) to produce a 31-bit product. The input words and output value are all two's complement; the inputs are (sign bit) + (15-bit data) and the output is (sign bit) + (30-bit data). For more detailed information on the internal representation of numbers in the SPI (especially in the multiplier), see Appendix 3.

The 31-bit output value is output as follows: the sign bit and 15 most significant data bits are output to the M register, and the 15 least significant data bits are output to the N register, left justified (a 0 is placed in the LSB of the N register). A multiply is done every instruction cycle.

3.9 K Register

The K Register is a 16-bit register that holds either the multiplier or the multiplicand to be input to the multiplier. The K register can be filled with data that is output to the internal data bus (IDB) from some other source register (@K =

Destination), or with data that is sent via the separate high RAM bus (@KLM = Destination, in which the L register is loaded from the IDB). The @KLR destination can also be used, which loads the K register from the IDB, and loads the L register via the special bus from the data/coefficient ROM.

When data is placed into the K register, it is automatically accessed by the multiplier so that the multiplier solution is available on the next instruction cycle.

The contents of the K register can also be output to the internal data bus (K = Source).

3.10 L Register

The L Register is a 16-bit register that holds either the multiplier or the multiplicand to be input to the multiplier. The L register can be filled with data that is output to the internal data bus (IDB) from some other source register (@L = Destination), or with data that is sent via the separate ROM bus (@KLR = Destination, in which the K register is loaded from the IDB). The @KLM destination can also be used, which loads the L register from the IDB, and loads the K register via the special bus from the high RAM block.

When data is placed into the L register, it is automatically accessed by the multiplier so that the multiplier solution is available on the next instruction cycle.

The contents of the L register can also be output to the internal data bus (L = Source).

3.11 M and N Registers

The M and N registers are the output registers for the multiplier. Output from the multiplier is in 2's complement form. The sign bit and the 15 higher bits of data are placed in the M register, and the 15 lower bits of data are placed in the N register (left justified, with bit 0 set to 0). This is shown below in Figure 3.2.

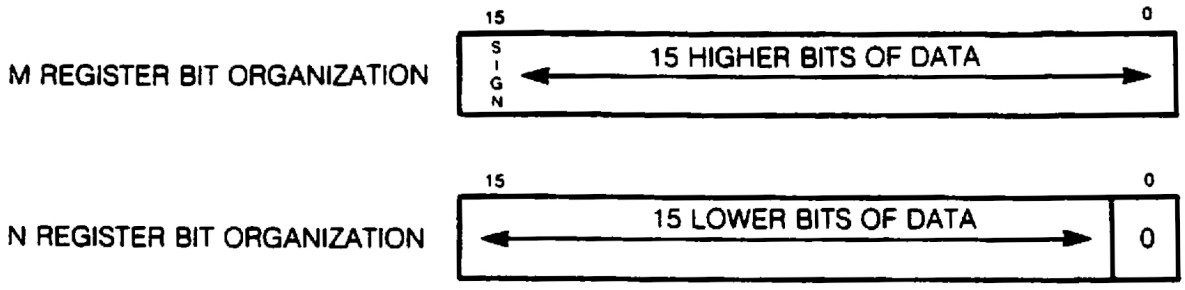


FIGURE 3.2 M AND N REGISTER BIT ORGANIZATION

If both the multiplier and multiplicand are the maximum negative value (8000H), the multiplier overflows and the output is 80000000H. This is equivalent to -1 x -1 = -1.

The outputs of the M and N registers are connected to the P input MPX of the ALU. No connection to the Internal Data Bus is

available for these registers.

For more detailed information on the internal representation of numbers in the SPI (especially in the multiplier), see Appendix 3.

3.12 ALU

The ALU is a 16-bit two's complement unit capable of executing sixteen distinct operations on either accumulator. Operations with two operands (OR/AND/XOR/SUB/SBB/ADD/ADC) may use virtually any internal register as the second operand. The operations are:

* NOP
* OR/AND/XOR
* Subtract
* Add
* Subtract with Borrow
* Add with Carry
* Decrement Accumulator
* Increment Accumulator
* Complement Accumulator
* 1-bit Right Shift
* 1-bit Left Shift
* 2-bit Left Shift
* 4-bit Left Shift
* 8-bit Exchange

The ALU outputs from these operations are stored back into the same accumulator that was operated on (ASL).

Note that in the case of a subtract that Q-P is performed (i.e. Accumulator - X). See Figure 3.3.

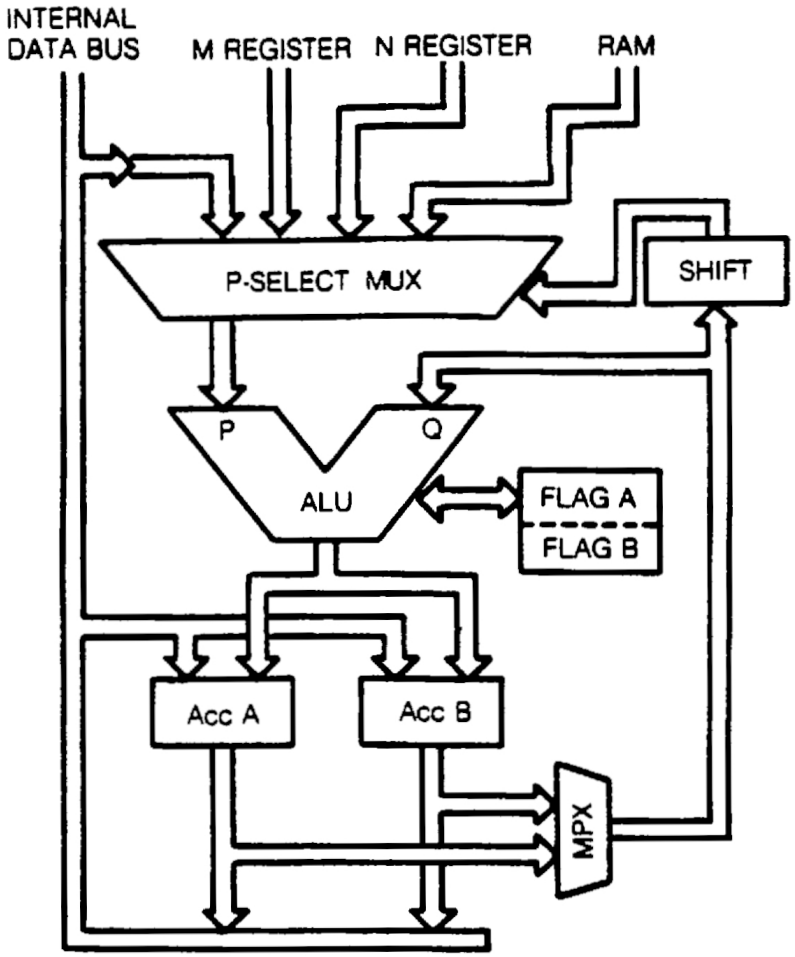


FIGURE 3.3 ALU AND CONNECTIONS BLOCK DIAGRAM

3.13 Accumulators (AccA, AccB)

The accumulators are a pair of 16-bit registers that store the results of ALU operations. Each accumulator has its own set of flags that are updated after each arithmetic operation except NOP.

Whether the ALU output goes to AccA or AccB is specified by the ASL bit of each OP/RT instruction. The contents of both AccA and AccB can be output to the internal data bus and can also be input to the Q input of ALU or to SHIFT, as needed by the particular ALU operation.

The output from the internal data bus to AccA or AccB, or conversely, from the accumulators to the internal data bus, is controlled by the SRC and DST fields of the OP/RT instruction. Data may be moved from a source to a destination during an arithmetic operation (see Figure 1.2 - Block Diagram).

If the accumulator being used for an ALU operation is specified as a destination (@A or @B) for IDB information in the same instruction, the ALU performs a NOP (regardless of the operation specified) and the IDB data is latched to that accumulator.

If the accumulator being used for an ALU operation is specified as a source (A or B) for IDB information in the same

instruction, the original contents of the accumulator at the beginning of the instruction (before the ALU operation is performed) is output to the IDB and latched to the destination register.

3.14 SHIFT

Five different shift operations may be implemented on 16-bit data in either AccA or AccB.

1) 1-bit Right Shift (Arithmetic)

The LSB is placed in the Carry bit of the Acc selected by the ASL bit of an OP/RT instruction. The sign bit is copied to the next lower bit. All other bits are shifted right one position. This operation is equivalent to a two's complement division operation.

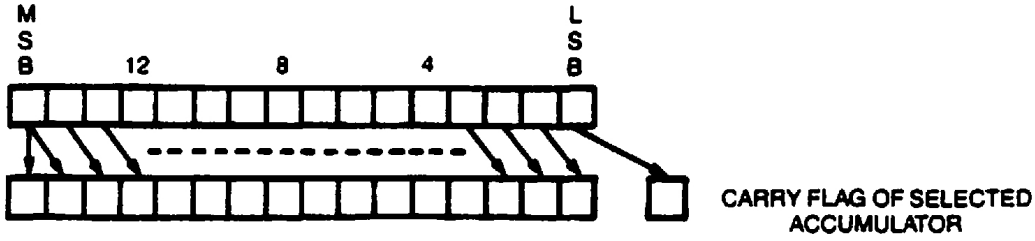


FIGURE 3.4 1-BIT RIGHT SHIFT

2) 1-bit Left Shift

The Carry flag of the Acc not selected by the ASL bit is put into the LSB, and the MSB of the selected Acc goes to its own Carry flag bit. This is useful for a 32-bit left shift (multiply by two).

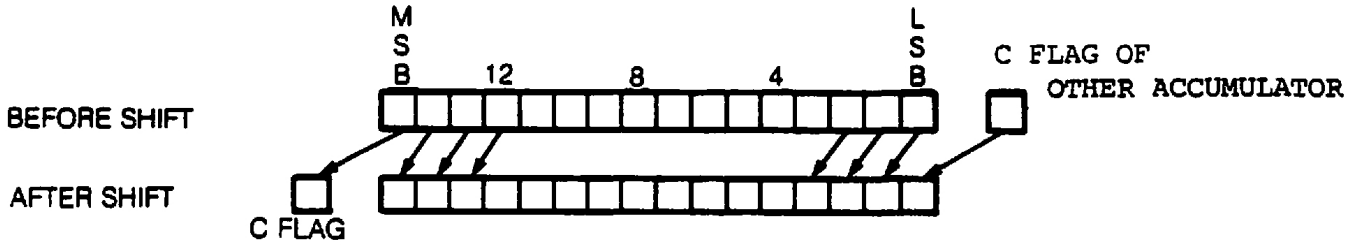


FIGURE 3.5 1-BIT LEFT SHIFT

3) 2-bit Left Shift

The two lower bits are filled with ones, and the two highest bits are discarded.

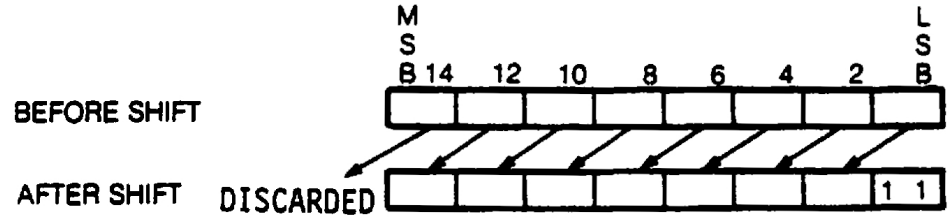


FIGURE 3.6 2-BIT LEFT SHIFT

4) 4-bit Left Shift

The four lower bits are filled with ones, and the four highest bits are discarded.

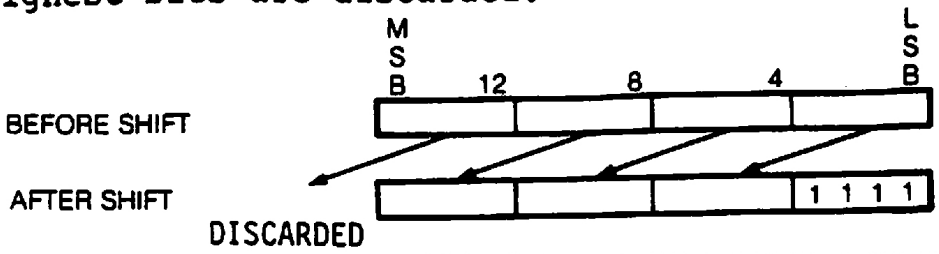


FIGURE 3.7 4-BIT LEFT SHIFT

5) 8-bit Exchange

The eight higher bits are exchanged with the eight lower bits.

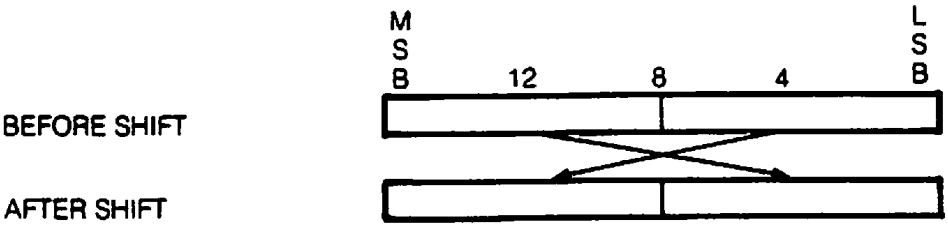


FIGURE 3.8 8-BIT EXCHANGE

3.15 Flag Registers (See ALU function and Flag operation - Table 4.2)

The uPD7720A has two flag registers: Flag A and Flag B. Flag A is a six-bit register that indicates the results and latest status of ALU operations performed on AccA. Flag B performs the same function for AccB. Note that moving data to an accumulator, when specified as the destination in either an OP/RT instruction or a LDI instruction, will not affect the flag for that accumulator. Each flag register consists of the following flag bits shown in figure 3.9.

FLAG A	SA1	SA0	CA	ZA	OVA1	OVA0
FLAG B	SB1	SB0	CB	ZB	OVB1	OVB0

FIGURE 3.9 ACCUMULATOR FLAG BITS

1) CA, CB (Carry)

These flags store the carry of operational results.

Carry from the ALU is stored after one of the following operations: SUB, ADD, SBB, ADC, DEC, or INC.

After a 1-bit Right Shift or a 1-bit Left Shift, the LSB or MSB is stored, respectively.

In operations other than those above, this flag is set to

zero. The last previous status is unchanged after a NOP.

2) ZA, ZB (Zero)

This flag is set to one if the ALU operation result data stored in the Acc is zero, and is set to zero if the contents of the Acc is nonzero. The last previous status is unchanged after a NOP.

3) SA0, SB0 (Sign)

In operations other than NOP, the MSB of the ALU contents is placed in this flag. The last previous status is unchanged after a NOP.

4) OVA0, OVB0 (Overflow)

This flag stores the logical XOR of the carry from the 15th bit (MSB (SIGN)) and the carry from the 14th bit of the Acc after one of the following operations: SUB, ADD, SBB, ADC, DEC, or INC. The last previous status is unchanged after a NOP. This flag is set to zero after any operation other than the previous ones.

5) OVA1, OVB1 (Overflow)

This flag promotes more efficient overflow processing for up to three consecutive additions.

This flag is set to one if the overflow flag (OVA0, OVB0) was set an odd number of times after the three consecutive additions. It is set to zero if the overflow flag was either never set or set twice in a row. If the overflow was set in the order 1-0-1, then:

OVA1 (OVB1) = 1, if SA1 (SB1) = SA0 (SB0), or
OVA1 (OVB1) = 0, if SA1 (SB1) is not equal to SA0 (SB0)

This applies to SUB, ADD, SBB, ADC, DEC, and INC. The previous status is unchanged after a NOP. This flag is set to zero after operations other than the above.

For a more detailed discussion of overflow processing using the OVA1 and SA1 (OVB1 and SB1) flags, see Appendix 2.

6) SA1, SB1 (Sign)

This flag, used with OVA1 (OVB1) aids in overflow processing. Direction of an overflow (positive or negative) can be judged with this flag.

If the overflow status (OVA1, OVB1) equals zero as the next ALU operation (SUB, ADD, SBB, ADC, DEC, INC) is begun, this flag is set equal to the resultant sign bit (SA0, SB0) after that operation. If the overflow status (OVA1, OVB1) equals one at the beginning of the operation, the value of this flag is unchanged. In this way, the sign of the accumulator immediately after an overflow is preserved through succeeding operations.

The previous status is unchanged after a NOP. This flag is indeterminate after operations other than the above.

Again, see Appendix 2 for a more detailed discussion of overflow processing.

3.16 Sign Register (SGN)

When OVA1 is set, the SA1 bit will hold the corrected sign of the overflow. The SGN Register will use SA1 to automatically generate saturation constants 7FFFH(+) or 8000H(-) to permit efficient limiting of a calculated value. The SGN will hold the current saturation value for up to three consecutive additions in a row.

3.17 Temporary Register (TR)

The TR register is a 16-bit temporary storage register on the internal data bus.

3.18 Status Register (SR)

The Status Register is a 16-bit register that contains the information required to handle data transfers with external devices. Of the 16 bits, only the 8 MSBs may be read by an external processor. Any attempt to write into bits that are not defined (bits 2 through 6) or under SPI control (RQM, bit 15; and DRS, bit 12) is ignored. After reset, all SR bits will be cleared to zero.

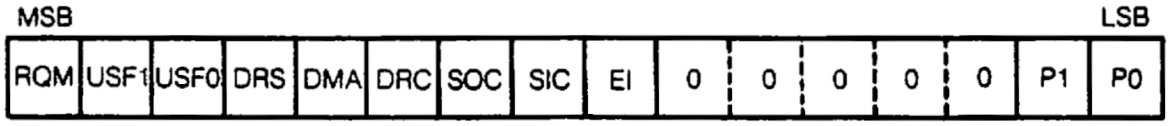


FIGURE 3.10 STATUS REGISTER BITS

1) RQM (Request for Master)

This flag bit is used for data transfers between the data register and the host processor in non-DMA mode.

A data transfer (read or write) between the internal data bus (IDB) and the Data Register (DR) sets this flag to one. An external 8- or 16-bit read or write resets this flag to zero. The status of this bit remains unchanged when DMA = 1 (DMA mode).

When using DRNF in the SRC field of an OP/RT instruction, this flag is not set, even though data is read from the data register to the internal data bus.

2) USF1, USF0 (User Flags)

These are general purpose flags that can be read by the host processor for user-defined signalling.

3) DRS (Data Register Status)

This bit indicates the status of data transfers when an external device views the data register as a 16-bit register (DRC = 0).

The data bus connecting to external devices has only 8 bits;

therefore, if the data register is operating in 16-bit mode, data must be transferred in two steps. This bit turns to one after the first transfer, then back to zero after the second transfer completes the 16-bit transfer (low byte is transferred first, then high byte).

This bit remains at zero during 8-bit transfers (DRC = 1).

4) DMA (Direct Memory Access)

This bit determines the method by which data can be transferred between an external device and the data register.

When this bit is 1, data transfers are made via DMA using DRQ and DACK/. Note that DACK/ is the equivalent of setting both CS/ and A0 to a low condition. When this bit is a zero, parallel data transfer is handled by controlling CS/ and A0 directly by the controlling device.

5) DRC (Data Register Control)

This bit controls whether the data register is used in double byte (16-bit) or single byte (8-bit) mode. When this bit is 0, the data register is set for 16-bit transfers. When this bit is 1, the data register is set for 8-bit transfers.

6) SOC (Serial Output Control)

This bit specifies the length of serial data to be output from the SO pin. When this bit is 0, all 16 bits of the data word are output. When this bit is a 1, only 8 bits of the data word are output (the low 8 bits of the SO register).

7) SIC (Serial Input Control)

This bit specifies the length of serial data to be input at the SI pin. When this bit is 0, the data input is 16 bits. When this bit is a 1, the data input is 8 bits.

8) EI (Enable Interrupt)

When this bit is 1, interrupts are accepted. When an interrupt is received, this bit is automatically reset to reject subsequent interrupts. When this bit is set to zero, interrupts are not accepted or remembered by the SPI.

9) P1, P0

These bits correspond to the output ports P1 and P0. All values input here are output as they are (e.g. setting P0 to a 1 will cause the P0 output pin to go to a high output level).

3.19 Parallel I/O Port

The 8-bit parallel I/O port may be used for transferring data or reading the status of the SPI. Data transfer is handled through a 16-bit Data Register (DR) that is software-configurable

for single or double byte transfers.

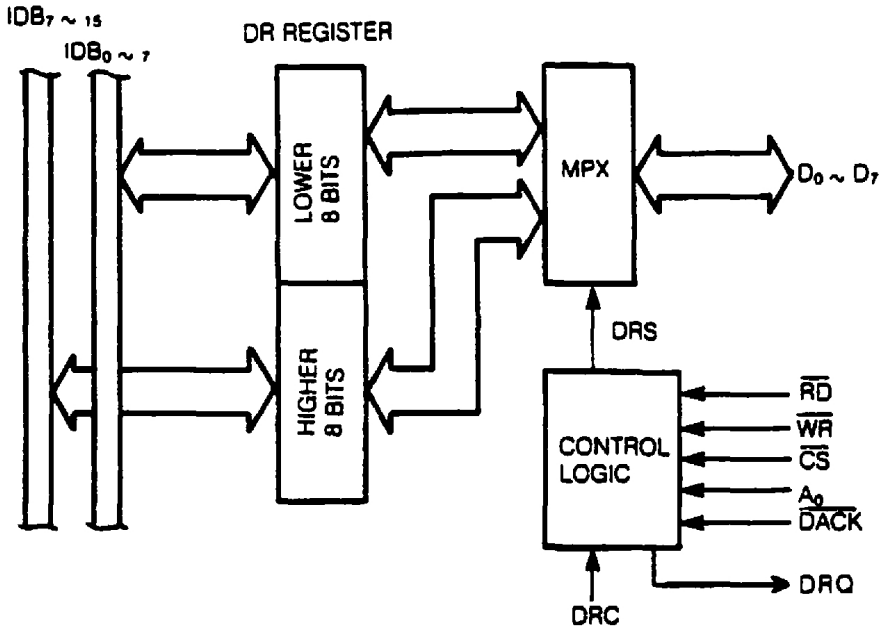


FIGURE 3.11 DATA REGISTER AND CONNECTIONS BLOCK DIAGRAM

3.20 DR (Data Register)

This 16-bit register is used to transfer data between the SPI and external devices. The data bus, D0-D7, is eight bits wide. Sixteen-bit data is transferred in two steps (low byte first, then high byte) even though a transfer is made only once internally.

If the DRC bit of the status register defines the data register as eight bits, only the lower eight bits of the data register are transferred to/from an external device.

3.21 Parallel READ/WRITE Operation

The Read/Write control logic transfers data to or from the SPI, depending on the status of the external control signals, CS/, A0, WR/, RD/, or DACK/, as shown in Table 3.1. The condition of DACK/ = 0 is equivalent to A0 and CS/ both being equal to 0.

Data is sent to or from the SPI in low byte, high byte order.

Whether the eight MSBs or eight LSBs are being transferred is denoted by the status of the DRS bit of the status register.

$\overline{CS}$	A_0	$\overline{WR}$	$\overline{RD}$	Operation
1	X	X	X	Internal operation is not affected: D_0-D_7 are kept at a high impedance state
X	X	1	1	
0	0	0	1	Data of D_0-D_7 are latched to DR register*
0	0	1	0	Contents of DR register are output to D_0-D_7*
0	1	0	1	PROHIBITED
0	1	1	0	8 higher bits of SR register are output to D_0-D_7
0	X	0	0	PROHIBITED

*Whether 8 higher bits or lower bits of DR register are assigned depends on the status of the DRS bit of the SR register.

TABLE 3.1 R/W CONTROL LOGIC

3.22 DMA Interface Logic

DMA data transfers are controlled by DRQ and DACK/, and may take place only when the DMA bit of the status register is set to 1.

1) DRQ Operation

DRQ is a DMA request for the host processor or the DMA controller to begin the DMA transfer. When data is transferred between the data register and the internal data bus, DRQ is set to 1 and output. DRNF in the SRC field of an OP/RT instruction prevents DRQ from being set even if a transfer from the data register has been made.

When DRC = 0, DRQ is reset at the second assertion of DACK/ to read or write data. In other words, if 16-bit transfers are specified, DRQ will remain active until the transfer of the second byte (of that 16-bit word being transferred) has been started.

When DRC = 1, DRQ is reset at every assertion of DACK/ to read or write data. In other words, if 8-bit transfers are specified, DRQ will remain active until the transfer of that byte of data has started.

Note: The RQM bit is not affected by data transfers in DMA mode.

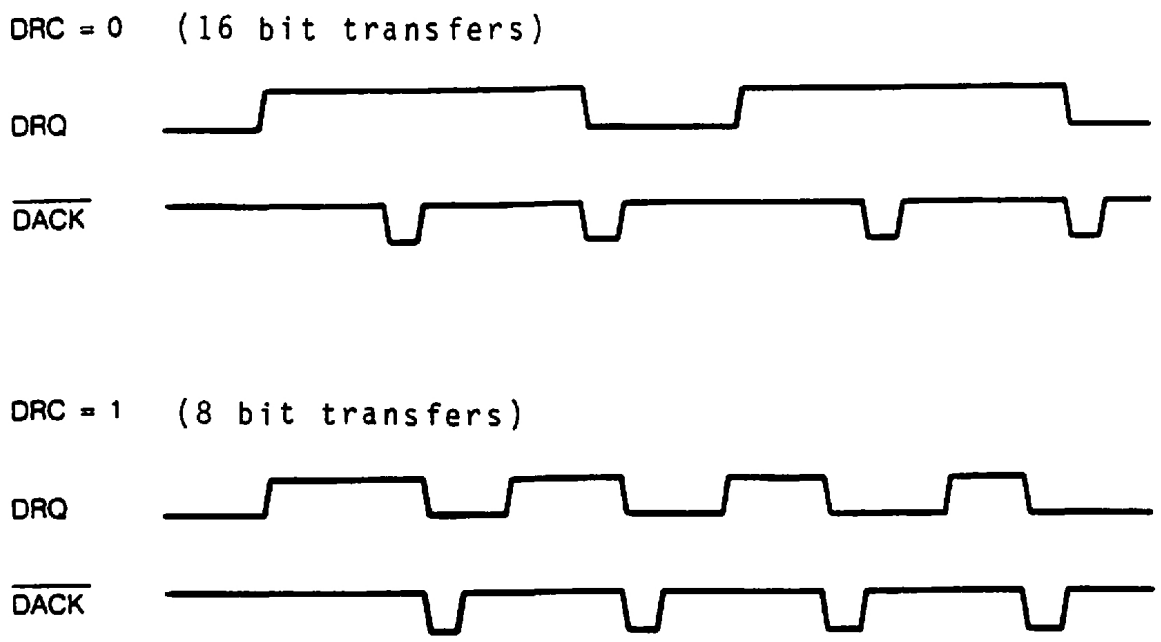


FIGURE 3.12 DRQ AND DACK/ TIMING

3.23 SI Register (Serial Input)

The SI register is the register in which serial input data is latched. It performs the following functions:

1) Serial data should be synchronized with the serial clock, such that the data may be latched at the rising edge of the serial clock.

2) Serial data is converted to parallel data by an internal shift register.

3) Shift register data is transferred to the SI register when the number of bits shifted equals the length specified by the SIC bit of the status register.

4) An internal flag (SIACK flag) is set when the data is latched to the SI register, and reset when data is read out of the SI register, by specifying it as the source of a MOV. There are conditional jump instructions that test the status of SIACK.

5) Two field specifications of the OP/RT instructions are available to read any parallel data from the SI register to the internal data bus. One specification (SIM = SRC) outputs the SI register's MSB to the MSB of the IDB (Normal order); the other specification (SIL = SRC) outputs the LSB to the MSB (bit reversed order).

6) If the SIC bit of the status register is 1 (8-bit mode) and data is shifted in MSB first, the SIM source specification should be used to load the data to the IDB in normal order. Zeroes will be placed in the high eight bits of the IDB.

7) The input of serial data to the shift register and the output of parallel data from the SI register (to the Internal Data Bus) may be performed independently. This feature permits the consecutive inputting of serial data.

8) If data is not read from the SI register before the next whole byte or word (depending on mode) has been shifted in, the previous data in the SI register will be lost, as the new data will be latched in over it into the SI register.

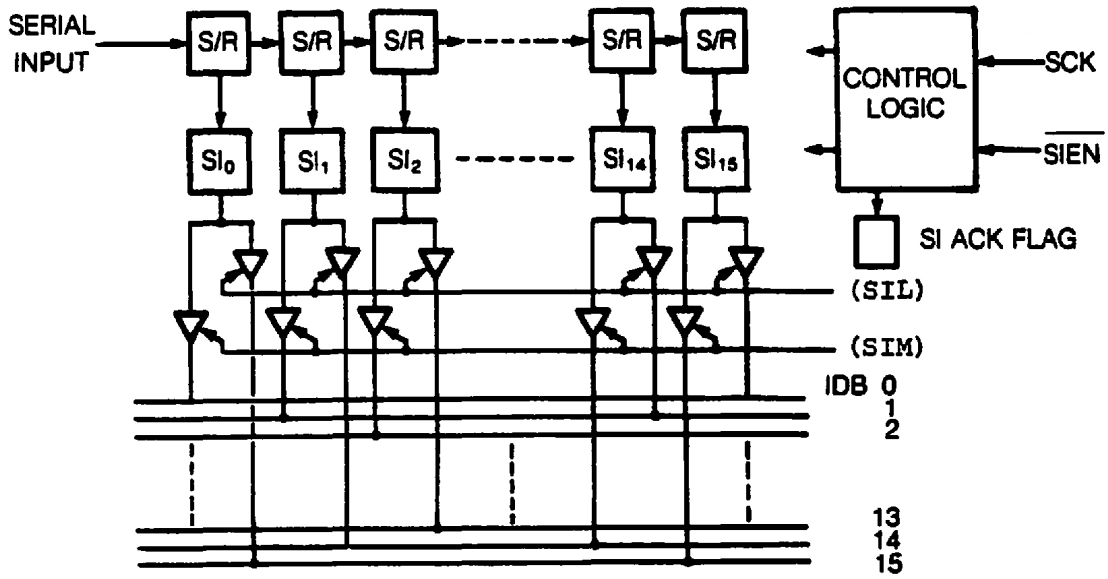


FIGURE 3.13 SERIAL INPUT BLOCK DIAGRAM

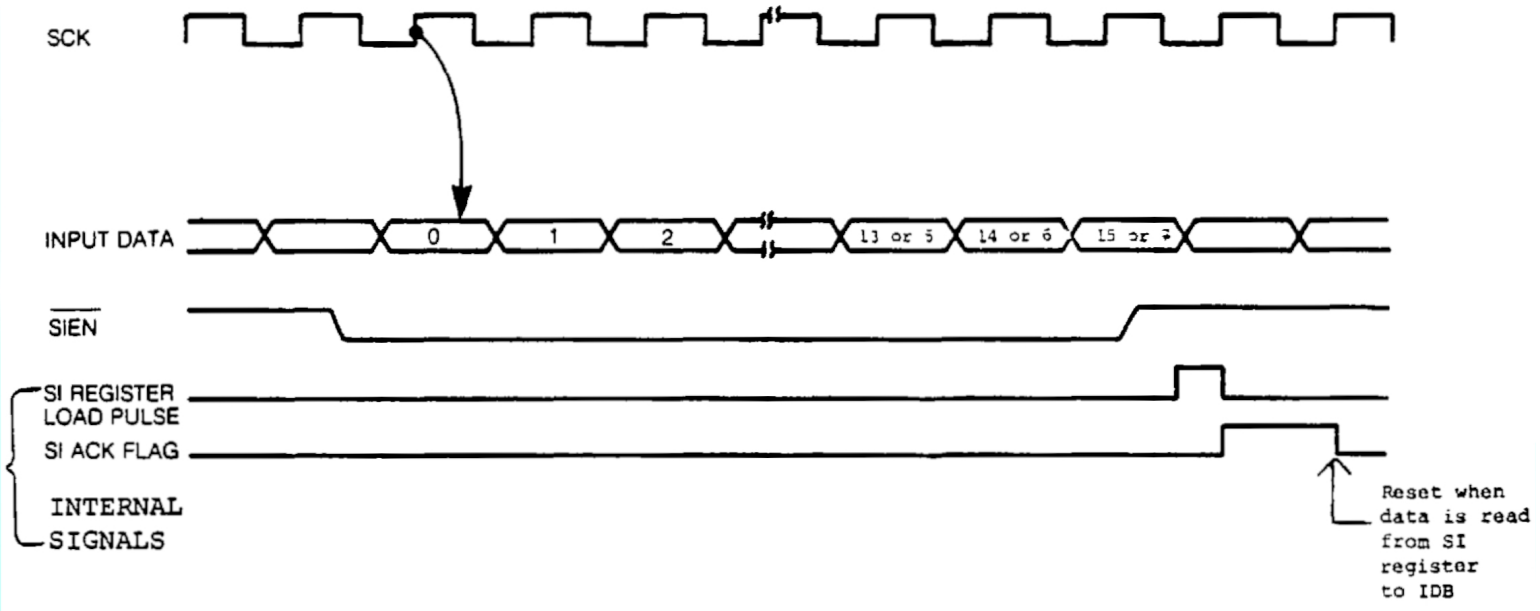


FIGURE 3.14 SERIAL INPUT TIMING

3.24 SO Register (Serial Output)

The SO register outputs serial data. It performs the following functions:

1) Data being output is loaded in the SO register (16 bits parallel) from the internal data bus.

2) An internal flag (SOACK flag) is set when data is written to the SO register.

3) Output data is transferred to the shift register from the SO register. If the shift register is busy, the data transfer is held until it is available.

4) SORQ is set to 1 when the data is transferred to the shift register informing the external device that data is available.

5) When output data is transferred from the SO register to the shift register, the SOACK flag is reset, indicating that the SO register is ready for the next word of data.

6) The serial transfer starts if SOEN/ = 0 is input after data is sent to the shift register, i.e. if SORQ = 1.

7) The SO pin is held high if SORQ = 1 and SOEN/ = 1, or if SORQ = 0 regardless of the state of SOEN/.

8) After sending the number of bits specified by the SOC bit of the status register, and if no more data is available in the SO register, SORQ is set to 0, instructing the system to wait for the next serial output data.

9) Serial data is clocked out synchronously with the falling edge of SCK, and is held until after the rising edge of SCK. It is the rising edge of SCK that should be used by the external device to accept each bit of data.

10) Like the SI register, SO register data may be sent in either normal or bit reversed order (using @SOM or @SOL, respectively, as the destination specification in the OP/RT instruction).

11) If the @SOM destination is used to write data to the SO register, the MSB of the 16-bit word written into the register is shifted out first, regardless of whether the SO register is in 8 or 16 bit mode. The result of this is that, in order to shift out 8-bit data MSB first, the 8-bit data byte must be in the high order byte of the source register before transferring it to the SO register.

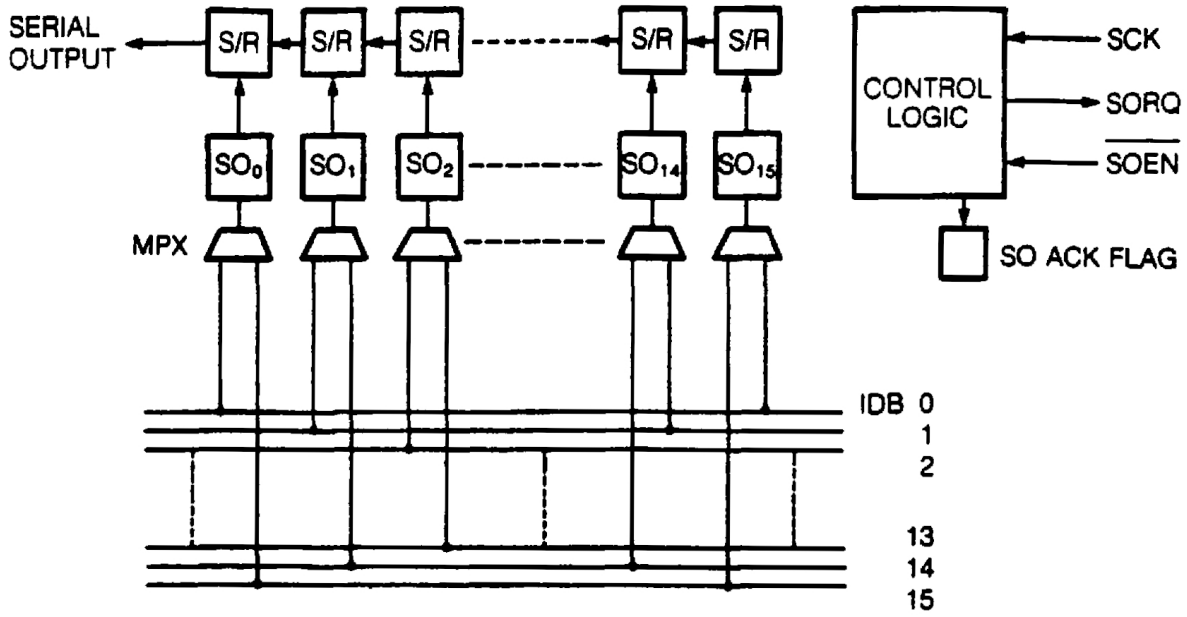


FIGURE 3.15 SERIAL OUTPUT BLOCK DIAGRAM

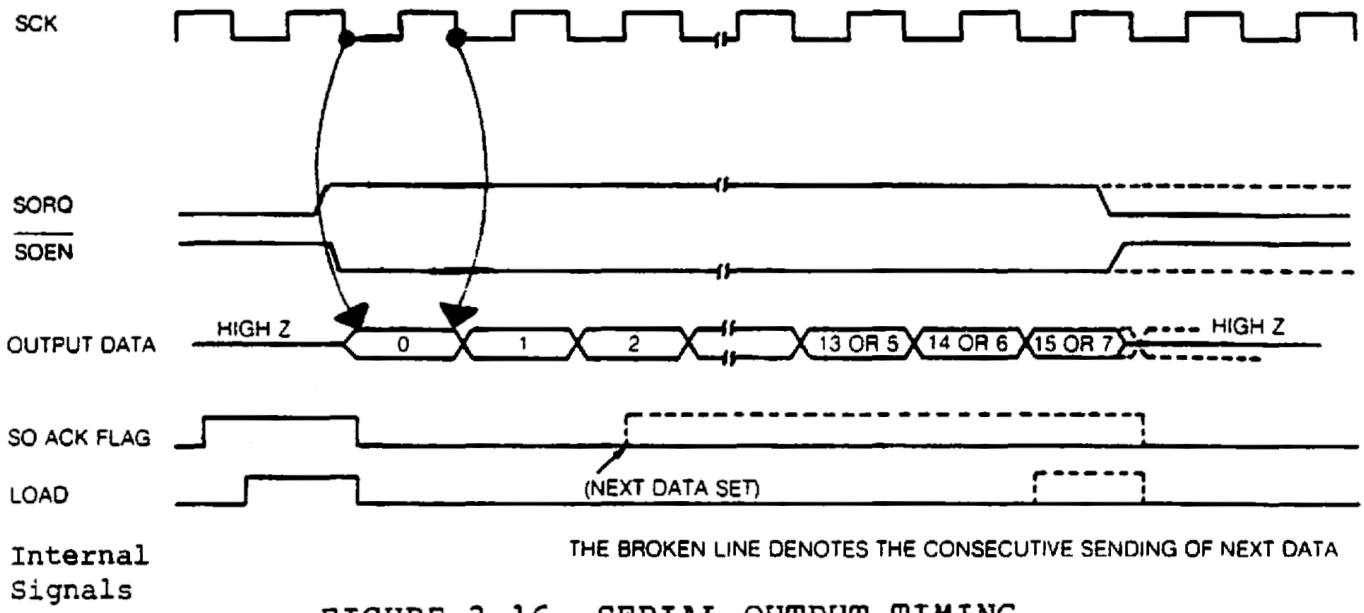


FIGURE 3.16 SERIAL OUTPUT TIMING

3.25 Interrupts

Interrupts are processed in the following manner:

1) If the EI bit of the status register is 1, when a rising edge is sensed on the INT pin, the present instruction is finished. And a NOP and JMP instruction to location 100H are implemented.

2) The return address is pushed onto the stack (during NOP).

3) All rising edges of interrupt line are ignored if the EI bit is set to zero (which will occur after an interrupt is received).

4) Interrupt will reset the EI bit to zero and must be set under program control to accept another interrupt.

5) After an interrupt is accepted, the INT pin should be reset low before the next interrupt; otherwise, only the first interrupt is accepted, even if the EI bit is high. This is a direct result of the fact that the INT pin is edge sensitive, not level sensitive.

4. INSTRUCTIONS

The SPI has three types of instructions, all one word composed of 23 bits. Each instruction type may be identified by the code in the OP field (Bits 22 & 21). All instructions execute in one instruction cycle.

4.1 OP/RT Instruction

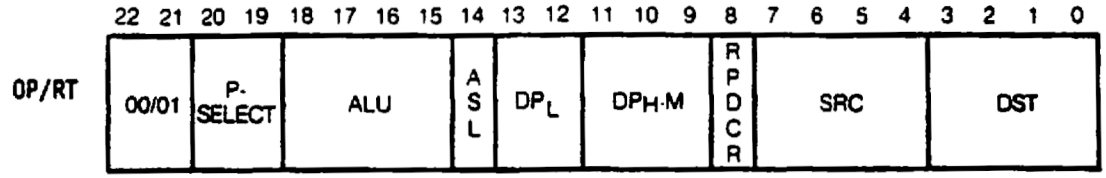


FIGURE 4.1 OP/RT INSTRUCTION FORMAT

This instruction performs the operations specified by the eight fields and the two bit OP code. It is used for arithmetic operations, data transfers, and subroutine returns.

When this is an OP instruction (OP Field = 00), the program counter holds the current address plus one as the next address. When this is an RT instruction (OP Field = 01), the program counter is set with the value at the top of the LIFO stack at the end of the instruction cycle.

FIGURE 4.2 ASSEMBLY LANGUAGE INSTRUCTION FORMAT
ARITHMETIC MOV ("OP/RT")

IN PARALLEL:

* TRANSFER DATA VIA THE INTERNAL DATA BUS

* PERFORM ALU OPERATION

* MULTIPLIER IS ALWAYS MULTIPLYING--RESULT IS ALWAYS AVAILABLE TO BE USED IN AN ALU OPERATION

* DATA RAM POINTER MODIFICATION

* DATA ROM POINTER MODIFICATION

* SUBROUTINE RETURN

EXAMPLE:

```
*    OP   MOV  @KLR,MEM  /* RAM(DP) --> K, ROM(RP) --> L */
                         /* LOAD BOTH MULTIPLIER INPUTS, */
                         /* K FROM RAM (VIA IDB), L FROM */
                         /* ROM (VIA SPECIAL BUS)        */
          ADD  ACCA,M    /* ADD PREVIOUS PRODUCT TO ACCA */
          DPINC          /* INCREMENT RAM POINTER LOW    */
          M1             /* MODIFY RAM POINTER HIGH      */
          RPDEC          /* DECREMENT ROM POINTER        */
          RET           ;/* SUBROUTINE (OR INT.) RETURN  */
```

* NOTE: 1) ALL OF THE ABOVE FUNCTIONS EXECUTE IN ONE INSTRUCTION CYCLE.

2) PRODUCT OF VALUES LOADED TO K AND L REGISTERS WILL BE AVAILABLE ON THE NEXT INSTRUCTION CYCLE.

3) ALL DP AND RP MANIPULATIONS ARE DONE AT THE END OF THE INSTRUCTION CYCLE (MODIFIED ADDRESSES WILL BE AVAILABLE ON THE NEXT INSTRUCTION CYCLE.

4) THIS IS ACTUALLY AN RT INSTRUCTION BECAUSE RET IS USED. WITHOUT RET THIS WOULD BE AN OP INSTRUCTION.

4.1.1 P-Select Field

This field selects the source for P input of the ALU. This input may come from RAM, the internal data bus, the M register, or the N register for the logical and arithmetic ALU operations. For the NOP, INC, DEC, Complement Acc, 8-bit Exchange, and Shift operations, the P input is ignored, and the accumulator alone is operated on accordingly.

	P-SELECT FIELD		
MNEMONIC	D_{20}	D_{19}	INPUT
RAM	0	0	RAM
IDB	0	1	Internal Data Bus
M	1	0	M Register
N	1	1	N Register

TABLE 4.1 P-SELECT FIELD

4.1.2 ALU Field

This field specifies the ALU function according to the table below. Note that all results from an ALU operation are left in the accumulator that was operated on (ASL bit).

							Flags Affected					
	ALU Field					Flag A	SA1	SA0	CA	ZA	OVA1	OVA0
Mnemonic	D_{18}	D_{17}	D_{16}	D_{15}	ALU Function	Flag B	SB1	SB0	CB	ZB	OVB1	OVB0
NOP	0	0	0	0	No Operation		–	–	–	–	–	–
OR	0	0	0	1	OR		X	↕	0	↕	0	0
AND	0	0	1	0	AND		X	↕	0	↕	0	0
XOR	0	0	1	1	Exclusive OR		X	↕	0	↕	0	0
SUB	0	1	0	0	Subtract		↕	↕	↕	↕	↕	↕
ADD	0	1	0	1	ADD		↕	↕	↕	↕	↕	↕
SBB	0	1	1	0	Subtract with Borrow		↕	↕	↕	↕	↕	↕
ADC	0	1	1	1	Add with Carry		↕	↕	↕	↕	↕	↕
DEC	1	0	0	0	Decrement ACC		↕	↕	↕	↕	↕	↕
INC	1	0	0	1	Increment ACC		↕	↕	↕	↕	↕	↕
CMP	1	0	1	0	Complement ACC (1's Complement)		X	↕	0	↕	0	0
SHR1	1	0	1	1	1-bit R-Shift		X	↕	↕	↕	0	0
SHL1	1	1	0	0	1-bit L-Shift		X	↕	↕	↕	0	0
SHL2	1	1	0	1	2-bit L-Shift		X	↕	0	↕	0	0
SHL4	1	1	1	0	4-bit L-Shift		X	↕	0	↕	0	0
XCHG	1	1	1	1	8-bit Exchange		X	↕	0	↕	0	0

↕ May be affected, depending on the results
– Previous status can be held
0 Reset
X Indefinite

TABLE 4.2 ALU FUNCTION AND FLAG OPERATION

1) NOP (No Operation)

No operation is made by the ALU. The Acc Flags are unchanged. All of the ALU functions that follow are equivalent to a NOP if the accumulator being operated on is used as a destination for a move as part of the same instruction (@A or @B = DST).

2) OR/AND/XOR

These operations are executed between the input selected by the P-Select field and one of the accumulators. The ASL bit chooses the Acc to be operated on.

3) SUB (Subtract)

This operation subtracts the input selected in the P-Select field from the specified accumulator and leaves the result in that accumulator. The borrow input to the lowest bit is zero (Acc - P-Select input).

4) ADD

This operation adds the input selected in the P-Select field to the specified accumulator. The carry input to the lowest bit is zero.

5) SBB (Subtract with Borrow)

This operation subtracts the input selected in the P-Select field from the specified accumulator. The Borrow (Carry) flag of the Flag register not selected by the ASL bit is input as a borrow to the lowest bit. For example, if AccA is selected, the Borrow (Carry) of Flag B (CB) is used as the borrow. This feature is useful for implementing 32-bit arithmetic using both accumulators in parallel as a 32-bit composite accumulator.

6) ADC (Add with Carry)

This operation adds the input selected in the P-Select field to the specified accumulator. The Carry flag of the Flag register not selected by the ASL is input as a carry to the lowest bit. For example, if AccA is selected, the Carry of Flag B (CB) is added. As with the SBB operation, this feature is useful for implementing 32-bit arithmetic.

7) DEC (Decrement Acc)

This operation subtracts one from the value of the accumulator specified by the ASL bit.

8) INC (Increment Acc)

This operation adds one to the value of the accumulator

specified by the ASL bit.

9) CMP (Complement Acc)

This operation takes the one's complement of the value of the accumulator specified by the ASL bit.

10) SHR1 (1-bit Shift Right)

This operation shifts the contents of the specified accumulator one bit to the right. The sign bit (MSB, bit 15) retains its value and also is entered into the MSB-1 (bit 14). The LSB is sent to the carry flag of the Acc selected. For example, if AccA is selected, the LSB of AccA before the shift is sent to the carry flag of A (CA).

11) SHL1 (1-bit Shift Left)

This operation shifts the contents of the specified accumulator one bit to the left. The Carry flag of the accumulator not selected is input to the LSB after the shift. For example, if AccA is selected, the carry flag of B (CB) is input to the LSB of A. The MSB (bit 15) before the shift is sent to its own carry flag. (In this example, AccA's MSB goes to CA).

12) SHL2 (2-bit Shift Left)

This operation shifts the contents of the specified accumulator two bits to the left. The two LSBs after the shift are both set to 1. The two MSBs before the shift are discarded.

13) SHL4 (4-bit Shift Left)

This operation shifts the contents of the specified accumulator four bits to the left. The four LSBs after the shift are all set to 1. The four MSBs before the shift are discarded.

14) XCHG (8-bit Exchange)

This operation exchanges the eight higher bits and eight lower bits of the selected accumulator.

4.1.3 ASL (Accumulator Select) Bit

This bit specifies whether AccA or AccB is used for the ALU operation. The same Acc is specified for both the input to the ALU operation and the place where the result is stored. The corresponding flag register is also selected by this bit, except in the cases already mentioned in the ALU operations section in which the other accumulator's carry flag is used.

Accumulator A is selected if this bit is 0, and B is selected if this bit is 1.

This bit is ignored if NOP is specified in the ALU field, or in the case that the Acc specified by this bit is also specified in the DST (Destination) field.

Mnemonic	ASL Field D_{14}	A_{CC} Selection
ACCA	0	A_{CC} A
ACCB	1	A_{CC} B

TABLE 4.3 ACCUMULATOR SELECT FIELD

4.1.4 DPL Field

This field specifies the modification of the four lower bits (DPL) of the data pointer. The DPL value, changed by the operation, is valid for the next instruction, and specifies the RAM address for that instruction.

The table below shows the possible operations.

Mnemonic	DP_L Field D_{13}	D_{12}	DP_3-DP_0
DPNOP	0	0	No Operation
DPINC	0	1	Increment DP_L
DPDEC	1	0	Decrement DP_L
DPCLR	1	1	Clear DP_L

TABLE 4.4 DATA POINTER LOW FIELD

This field is ignored if @DP is specified in the DST field. If DP is specified in the SRC field, the value of DP before change is output to the internal data bus. If the RAM is accessed in this instruction (either for a move or for P-Select), the value of DP before change is the effective address during this instruction.

The counter used for modification is a modulo counter (0H comes after 0FH if using DPINC). There is no carry to or borrow from the upper bits of DP (DPH).

Note that a conditional branch instruction can be used to test for the conditions that the DPL field is either 0 or 0FH.

4.1.5 DPH-M (DPH Modify) Field

This field modifies the three higher bits (DPH) of the data pointer by XORing them with the three bits of this field. The modified DPH value is valid for the next instruction, and specifies the RAM address of that instruction.

Mnemonic	DP_H-M Field			Exclusive OR
	D_{11}	D_{10}	D_9	
* M0	0	0	0	(DP_6 DP_5 DP_4) ¥ (0 0 0)
M1	0	0	1	DP_6 DP_5 DP_4 ¥ (0 0 1)
M2	0	1	0	DP_6 DP_5 DP_4 ¥ (0 1 0)
M3	0	1	1	DP_6 DP_5 DP_4 ¥ (0 1 1)
M4	1	0	0	DP_6 DP_5 DP_4 ¥ (1 0 0)
M5	1	0	1	DP_6 DP_5 DP_4 ¥ (1 0 1)
M6	1	1	0	DP_6 DP_5 DP_4 ¥ (1 1 0)
M7	1	1	1	DP_6 DP_5 DP_4 ¥ (1 1 1)

* NO CHANGE

TABLE 4.5 DATA POINTER HIGH MODIFICATION FIELD

As shown in Table 4.5, DP6, DP5, and DP4 are XORed with the value in this field, and the result is put into DP6-4. If no modification of the DPH value is desired, each bit of this field must be set to zero (M0). M0 is selected by default when the DPH-M field is not specified in the OP/RT instruction.

As is the case with the DPL field, this field is ignored if @DP is specified in the DST field. If DP is specified in the SRC field, the value of DP before change is output to the internal data bus. If the RAM is accessed in this instruction (either for a move or for P-Select), the value of DP before change is the effective address during this instruction.

See Appendix 1 for examples of how to use the DPL and DPH fields for efficient use of memory.

4.1.6 RPDCR Bit (RP Decrement)

When this bit is 1, the value in the RP register is decremented. The decremented value is valid for the next instruction. The output of the data ROM that corresponds to the decreased value can be read on the next instruction.

No change is made to the RP pointer value if this bit is set to 0.

As with the DPL and DPH fields, this bit is ignored if @RP is specified in the DST field. Also, if the ROM is accessed by the current instruction in which RP is being decremented, the address before decrementing is used to access the ROM. Finally, if RP is specified as the SRC field, the value before decrementing is output to the internal data bus.

Mnemonic	RPDCR D_8	Operation
RPNOP	0	No Operation
RPDEC	1	Decrement RP

TABLE 4.6 ROM POINTER DECREMENT FIELD

A typical usage of the ROM area would be for filter coefficients. Suppose several biquad filters are to be performed for each input sample. Then, the filter coefficients could be stored in descending ROM locations, in the order in which they will be needed to perform the filter computations. At the beginning of each sample period, the top address of the coefficient block could be loaded to @RP with a LDI (Load Immediate) instruction. Then, the filter subroutine would merely perform a RPDEC in each instruction that used a filter coefficient (using a coefficient would typically mean moving it to the multiplier). In this way, the RP would be pointing to the next coefficient that would be needed by the filter subroutine before the particular instruction that would use that coefficient. If the last access of a filter coefficient in the filter subroutine also has a RPDEC associated with it, the RP will "fall through" and point to the first coefficient for the next filter section, without needing to be set-up between calls to the filter subroutine.

4.1.7 SRC Field (Source)

This field specifies the register from which the data that is to be placed on the internal data bus comes.

Mnemonic	SRC Field				Specified Register
	D_7	D_6	D_5	D_4	
NON	0	0	0	0	NO Register
A	0	0	0	1	ACC A (Accumulator A)
B	0	0	1	0	ACC B (Accumulator B)
TR	0	0	1	1	TR Temporary Register
DP	0	1	0	0	DP Data Pointer
RP	0	1	0	1	RP ROM Pointer
RO	0	1	1	0	RO ROM Output Data
SGN	0	1	1	1	SGN Sign Register
DR	1	0	0	0	DR Data Register
DRNF	1	0	0	1	DR Data No Flag ①
SR	1	0	1	0	SR Status
SIM	1	0	1	1	SI Serial in MSB ②
SIL	1	1	0	0	SI Serial in LSB ③
K	1	1	0	1	K Register
L	1	1	1	0	L Register
MEM	1	1	1	1	RAM

① DR to IDB, RQM flag not set. In DMA mode, DRQ not set.
② First bit in goes to MSB, last bit to LSB.
③ First bit in goes to LSB, last bit to MSB (bit reversed).

TABLE 4.7 SOURCE FIELD SPECIFICATIONS

The registers that may be used for this field are shown in Table 4.7. The contents of the specified register are output to the internal data bus. In general, the specified source field should not be also specified as the destination field. Such an operation would be meaningless. If no move is desired, NON and @NON can be specified as the SRC and DST fields (the assembler will default to these specifications if these fields are not explicitly specified in the source code for any given OP/RT instruction).

As previously mentioned, registers that are specified as SRC registers that are also being changed in the same instruction (such as accumulators selected for ALU operation, or the RAM and ROM pointers that may be getting modified) are generally put onto the internal data bus with their values before change.

4.1.8 DST Field (Destination)

The value placed on the IDB (SRC) is latched to the register specified in the DST field.

Mnemonic	DST Field				Specified Register
	D_3	D_2	D_1	D_0	
@NON	0	0	0	0	NO Register
@A	0	0	0	1	ACC A (Accumulator A)
@B	0	0	1	0	ACC B (Accumulator B)
@TR	0	0	1	1	TR Temporary Register
@DP	0	1	0	0	DP Data Pointer
@RP	0	1	0	1	RP ROM Pointer
@DR	0	1	1	0	DR Data Register
@SR	0	1	1	1	SR Status Register
@SOL	1	0	0	0	SO Serial Out LSB ①
@SOM	1	0	0	1	SO Serial Out MSB ②
@K	1	0	1	0	K (Mult)
@KLR	1	0	1	1	IDB → K ROM → L ③
@KLM	1	1	0	0	Hi RAM → K IDB → L ④
@L	1	1	0	1	L (Mult)
@NON	1	1	1	0	NO Register
@MEM	1	1	1	1	RAM

① LSB is first bit out.
② MSB is first bit out.
③ Internal data bus to K and ROM to L register.
④ Contents of RAM address specified by $DP_6 = 1$ (i.e., 1, DP_5, DP_4-DP_0) is placed in K register. IDB is placed in L.

TABLE 4.8 DESTINATION FIELD SPECIFICATIONS

If the same accumulator is specified in both the DST field and the ASL bit, the ASL bit is ignored and the ALU performs a NOP. If the DP or RP register is specified in this field, any modifications specified by the DPH, DPL, and RPDCR fields are ignored.

Note that the @KLM and @KLR destinations latch data from a special bus to the K or L registers, from High RAM or ROM (respectively), as well as the value on the IDB (from wherever the SRC field specifies) to the L or K multiplier input register. This feature allows both multiplier input registers to be loaded as part of the same instruction (while the previous multiplier result may be added to an accumulator at the same time).

See the examples in Appendix 1 for illustrations of the use

of these destination specifications.

The LDI instruction also uses the DST field to specify where to load immediate data.

4.1.9 Instruction Timing

As mentioned in the preceeding sections, certain operations that can be done in an OP/RT instruction are done before or after certain other operations. For example, if an accumulator is specified in both the ASL and SRC fields, the value before the ALU operation is performed is the value that goes onto the IDB. Similar timing holds for DP or RP as sources if pointer modifications are specified. Because of these relations among operations happening concurrently in one instruction, it is strongly recommended to write OP/RT instructions according to the following convention:

```
OP    MOV   @DST,SRC  /* Move specified first, in
                         case the register specified
                         as the source (or the
                         pointer for the source) is
                         changed later in this same
                         instruction (value of
                         register before change is
                         used as source)  */
      ALU   ASL,P-SEL /* ALU operation next, since
                         P-Select may specify IDB,
                         and the source register is
                         now already specified, or
                         in case the P-Select field
                         is RAM, and the DP pointer
                         is going to be modified in
                         this instruction */
      DPL             /* Pointer modifications next,
                         in any order, come after
                         MOV and ALU operations,
                         since these modifications
                         are not effective for use
                         during this instruction,
                         but are ready in time for
                         the next instruction    */
      DPH-M
      RPDCR
      RET             /* Return at end, if desired,
                         to indicate that all
                         preceeding actions are,
                         in fact, done before the
                         return is executed */
```

Of course, the above operations may be specified in any order within the same OP/RT instruction, and the assembler will still assemble the exact same object code, and the operations will still be performed in the same order, regardless of the order written. The order suggested here is only for convenience, to minimize any confusion about the order in which the individual

operations are performed.

Note, however, that if the DST field specifies the same accumulator as ASL, the MOV will take precedence, and the ALU operation will be forced to NOP (internally in the SPI, not by the assembler). Similarly, if the DST field specifies @DP or @RP and the pointer is also specified to be modified, again the MOV will take precedence and the pointer modification will be ignored. These conflicts are described in the preceeding sections that describe the individual fields of the OP/RT instruction. Therefore, to avoid confusion, it is further recommended to avoid such conflicting specifications.

4.2 JP Instruction

The JP instruction contains three fields (other than the OP code)--branch (BRCH), condition (CND), and next address (NA), and it may take one of three forms: unconditional jump, conditional jump, or subroutine call.

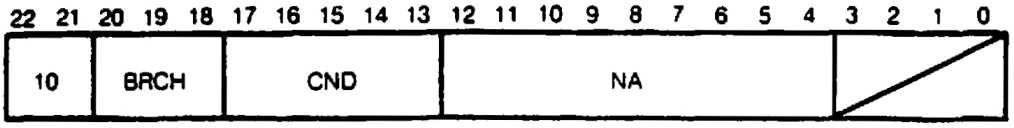


FIGURE 4.3 JP INSTRUCTION FORMAT

```
EXAMPLE:  JSA0 OVFLOW     ;     /* Jump to OVFLOW if the AccA
                                   Sign 0 bit (SA0) is "1" */
```

FIGURE 4.4 JP ASSEMBLY LANGUAGE INSTRUCTION FORMAT

4.2.1 BRCH Field (Branch)

This field specifies which of the three forms is to be executed. The instruction specifications are shown in Table 4.9.

BRCH FIELD				
MNEMONIC	D_{20}	D_{19}	D_{18}	FUNCTION
JMP	1	0	0	Unconditional Jump
CALL	1	0	1	Subroutine Call
*	0	1	0	Conditional Jump

*Mnemonic of the CND field is used.

TABLE 4.9 BRANCH FIELD

1) Unconditional Jump

The address in the NA field is transferred to PC when this instruction is executed. The CND field is ignored.

2) Conditional Jump

If the condition specified in the CND field is true, the address in the NA field is transferred to PC when this instruction is executed. If the condition is false, PC = PC + 1.

3) Subroutine Call

The address in the NA field is transferred to PC, and the return address (PC + 1) is pushed to the stack. The CND field is ignored.

4.2.2 CND Field (Condition)

This field specifies the condition that must be true for a conditional jump to be executed. All possible conditional jump instructions and their mnemonics are shown in Table 4.10.

TABLE 4.10 CONDITION FIELD

	CND FIELD					
MNEMONIC	D_{17}	D_{16}	D_{15}	D_{14}	D_{13}	CONDITION
JNCA	0	0	0	0	0	CA = 0
JCA	0	0	0	0	1	CA = 1
JNCB	0	0	0	1	0	CB = 0
JCB	0	0	0	1	1	CB = 1
JNZA	0	0	1	0	0	ZA = 0
JZA	0	0	1	0	1	ZA = 1
JNZB	0	0	1	1	0	ZB = 0
JZB	0	0	1	1	1	ZB = 1
JNOVA0	0	1	0	0	0	OVA0 = 0
JOVA0	0	1	0	0	1	OVA0 = 1
JNOVB0	0	1	0	1	0	OVB0 = 0
JOVB0	0	1	0	1	1	OVB0 = 1
JNOVA1	0	1	1	0	0	OVA1 = 0
JOVA1	0	1	1	0	1	OVA1 = 1
JNOVB1	0	1	1	1	0	OVB1 = 0
JOVB1	0	1	1	1	1	OVB1 = 1
JNSA0	1	0	0	0	0	SA0 = 0
JSA0	1	0	0	0	1	SA0 = 1
JNSB0	1	0	0	1	0	SB0 = 0
JSB0	1	0	0	1	1	SB0 = 1
JNSA1	1	0	1	0	0	SA1 = 0
JSA1	1	0	1	0	1	SA1 = 1
JNSB1	1	0	1	1	0	SB1 = 0
JSB1	1	0	1	1	1	SB1 = 1
JDPL0	1	1	0	0	0	DP_L = 0
JDPLF	1	1	0	0	1	DP_L = F (HEX)
JNSIAK	1	1	0	1	0	SI ACK = 0
JSIAK	1	1	0	1	1	SI ACK = 1
JNSOAK	1	1	1	0	0	SO ACK = 0
JSOAK	1	1	1	0	1	SO ACK = 1
JNRQM	1	1	1	1	0	RQM = 0
JRQM	1	1	1	1	1	RQM = 1

4.2.3 NA Field (Next Address)

This field specifies the address to which the program vectors (unless a conditional branch is not executed).

4.3 LDI Instruction

The LDI (Load Immediate) instruction is composed of two fields other than the OP code, and is executed in the following manner:

1) Data is output from the ID (Immediate Data) field, over the internal data bus, to the register specified in the DST field.

2) The 0 bit of the ID field is output to the 0 bit of the IDB, and bit 15 to bit 15 of the IDB.

4.3.1 ID Field (Immediate Data)

This field contains the data to be transferred to the register specified in the DST field.

4.3.2 DST Field (Destination)

This field specifies the register to which the data in the ID field is to be loaded. The registers that can be used are the same as those for the DST field of the OP/RT instruction (see Table 4.8). Note that the @KLM and @KLR specifications can be used with the LDI instruction to perform a simultaneous load of both multiplier input registers (one with immediate data, and the other from either High RAM or ROM).

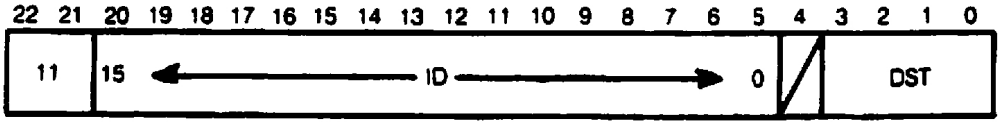


FIGURE 4.5 LDI INSTRUCTION FORMAT

```
        /*   Load Immediate Data value onto Internal Data
             Bus and store into register or memory          */

EXAMPLE: LDI  @A,1234H  ;    /*  Load 1234H to Accumulator A */
```

FIGURE 4.6 LDI ASSEMBLY LANGUAGE INSTRUCTION FORMAT

5. Timing

The uPD7720A operates by a single-phase clock applied to the CLK pin. The maximum frequency is 8 MHz, which gives a 250 ns instruction cycle. Refer to the electrical specifications in the data sheet for more detailed clock timing information.

5.1 Serial Data Timing

The serial I/O shift clock (SCK) can be input asynchronously with respect to the system clock (CLK). Refer to sections 3.23 (SI), and 3.24 (SO) for detailed information on serial I/O operation (maximum frequency is 2 MHz).

5.2 Reset Timing (RST)

Each RST input must be continued over 3 clock cycles to initialize the system. The RST input initializes the following to zero:

1) PC
2) Flags (for AccA and AccB)
3) SR Register (including EI)
4) DRQ
5) SORQ
6) Serial shift bit count

5.3 Interrupt

The interrupt input is rising edge sensed. For a rising edge to be accepted as a valid interrupt, the internal interrupt facilities must be enabled prior to that rising edge (EI status bit = 1). Interrupt must be held high for at least 8 clock cycles after the rising edge. Refer to section 3.25 for more information on the interrupt function.

5.4 I/O vs. Instructions

Since serial and parallel I/O may be performed asynchronously with respect to the system clock, it is essential to test the flag(s) associated with the I/O operation(s) to be performed, in order to assure proper operation, i.e. having data in the right place at the right time. This is done by the use of the conditional branch instructions such as JRQM, JNSIAK, JSOAK, etc. These flag test instructions should be performed, even if SCK, DACK/, CS/, RD/, WR/, etc. are synchronized with CLK. Assumptions about data being available after executing a certain number of instructions, without using the flag test instructions, are to be avoided.

6. TYPICAL SYSTEM CONFIGURATIONS

The uPD7720A is a single chip microcomputer. However, it is also designed to operate as a complex peripheral to a microprocessor.

Three configurations are shown in Figures 6.1-2-3. The first is with the SPI operating as a complex peripheral, inputting and outputting data serially as well as communicating on a microcomputer system parallel bus.

The second example shows two uPD7720As in a cascaded configuration. The SPIs could also be attached to a microcomputer bus if desired, but it is not shown in this example. This type of configuration is good for systems in which data rates exceed the capability of one SPI to process the allocated function completely within the available time.

The last configuration is a stand-alone application example.

A SIGNAL PROCESSING SYSTEM USING SPI(s) AS A COMPLEX COMPUTER PERIPHERAL

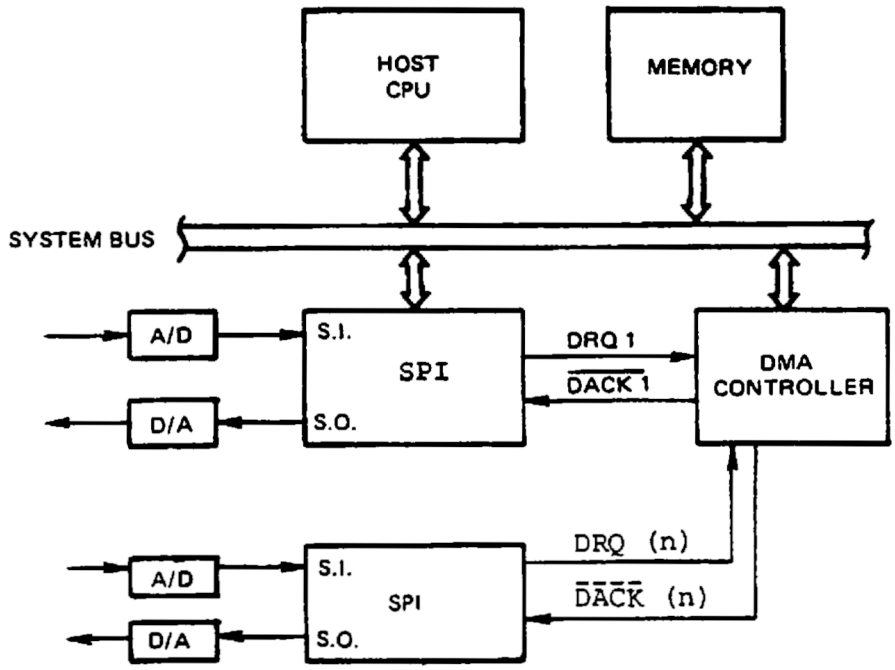


FIGURE 6.1

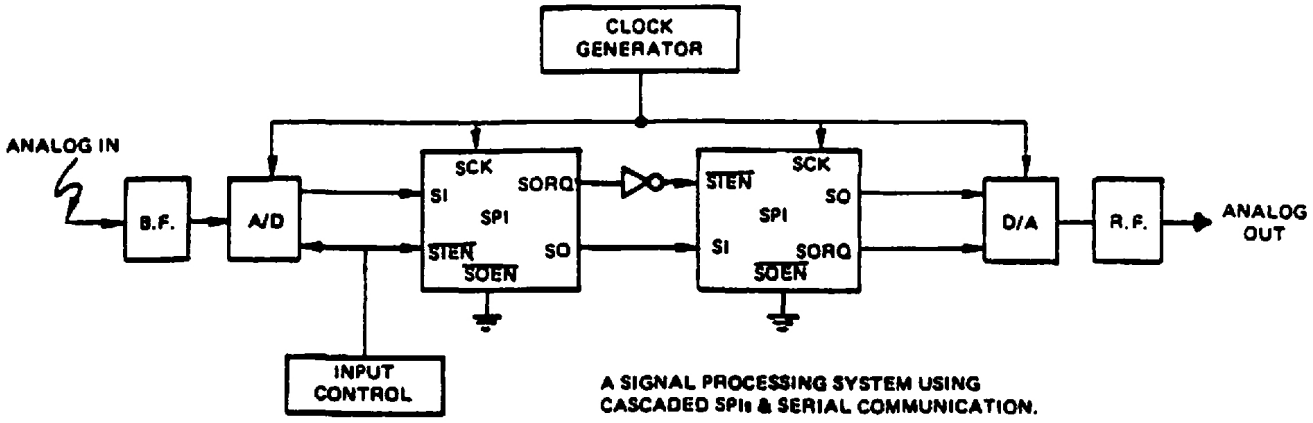


FIGURE 6.2

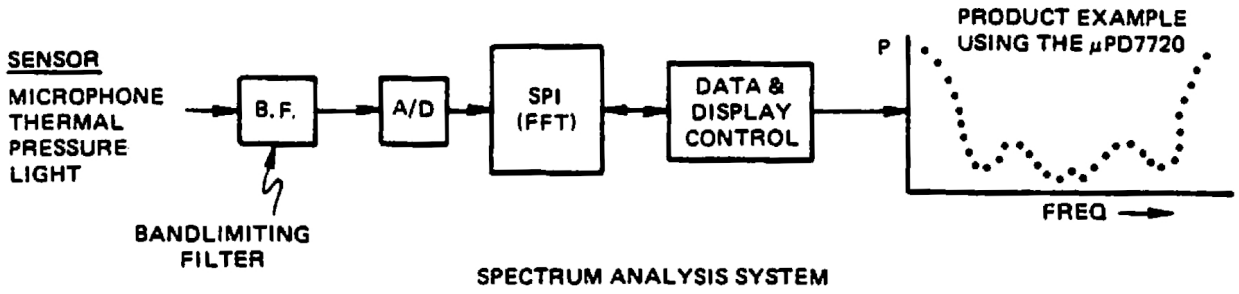


FIGURE 6.3

APPENDIX 1 ASSEMBLY LANGUAGE INSTRUCTION EXAMPLES

EXAMPLE A: Biquadratic Filter

The second order IIR digital filter, commonly known as a biquadratic, or "biquad" filter, is generally indicated by the signal flow diagram shown in Figure A-1.1. It uses two delay elements, the output of each one having both a feedback and a feed-forward coefficient. The transfer function, H(Z), is also defined in the figure. In a biquad filter, the coefficients A1 and B1 can have absolute values greater than 1. In fact, they will typically be as high as 2 for poles and zeroes located within the unit circle on the Z-plane. However, since the SPI is best thought of as having a dynamic range from -1 to almost +1, the configuration shown in figure A-1.2 will be used, in which the feedback and feed-forward products from the first delay tap will be added twice, with the coefficients stored as half of the actual desired values.

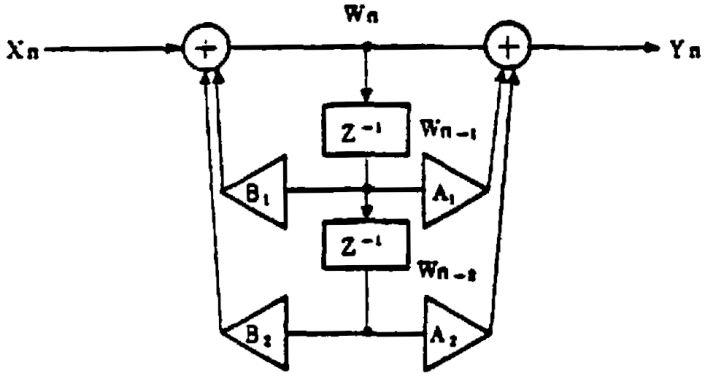
The usage of memory for the biquad filter subroutine is fairly straightforward. The two delay taps are kept in two memory locations, in the same "column" in adjacent "rows", for example locations 00 and 10H. To switch back and forth between the two delay tap locations within the subroutine, the M1 mnemonic is used to modify the upper three bits of the data RAM pointer.

The coefficients are stored in an equally simple manner. They will each be used in turn as the subroutine needs them, and so are stored in sequentially descending locations in the data ROM. As each one is used, the RPDEC mnemonic is used to modify the data ROM pointer to point to the next coefficient, ready for when it will be used.

The biquad filter subroutine, BIQFIL, occupies 11 locations in instruction ROM, executes in 2.5 usec (2.75 if an overflow occurs), takes the input sample as an input in the A accumulator, returns the filtered output sample in the A accumulator, and assumes that the DP (Data RAM Pointer) is pointing to the first delay tap, and the RP (Data ROM Pointer) is pointing to the first coefficient. The flow diagram is shown in figure A-1.3. The delay taps occupy two RAM locations, and the coefficients occupy five ROM locations. The code is as follows:

```
BIQFIL:   OP    MOV   @KLR,A     /* K=W(I), L=AN0                  */
                XOR   ACCA,IDB   /* CLEAR A FOR SUM OF PRODUCTS */
                RPDEC          ; /* POINT TO NEXT COEFFICIENT   */

          OP    MOV   @KLR,MEM   /* K=W(I-1), L=BN1                */
                ADD   ACCA,M     /* A=W(I) * AN0                   */
                RPDEC          ; /* POINT TO NEXT COEFFICIENT   */

          OP    MOV   @B,K       /* SAVE W(I-1) (NEW W(I-2))    */
                ADD   ACCA,M     /* A=W(I)*AN0 + W(I-1)*BN1     */
                M1             ; /* POINT TO SECOND DELAY TAP   */
```

```
OP    MOV  @KLR,MEM  /* K=W(I-2), L=BN2                    */
      ADD  ACCA,M    /* A=W(I)*AN0 + 2*W(I-1)*BN1         */
      RPDEC         ; /* POINT TO NEXT COEFFICIENT         */

OP    MOV  @L,RO     /* L=AN2                             */
      ADD  ACCA,M    /* A=W(I)*AN0 + 2*W(I-1)*BN1
                        + W(I-2)*BN2  (NEW W(I-1))        */
      RPDEC         ; /* POINT TO NEXT COEFFICIENT         */

JNOVA1    $+2       ; /* IF NO OVERFLOW, SKIP NEXT         */

OP    MOV  @A,SGN    ; /* IF OVERFLOW, GET SATURATION
                        VALUE                             */

OP    MOV  @TR,A     /* SAVE NEW W(I-1)                   */
      ADD  ACCA,M    /* A=NEW W(I-1) + W(I-2)*AN2         */
      M1            ; /* POINT TO FIRST DELAY TAP          */

OP    MOV  @KLR,MEM  /* K=W(I-1), L=AN1                    */
      RPDEC         ; /* POINT TO NEXT COEFFICIENT         */

OP    MOV  @MEM,TR   /* SET NEW W(I-1)                    */
      ADD  ACCA,M    /* A=NEW W(I-1) + W(I-2)*AN2
                        + W(I-1)*AN1                      */
      M1            ; /* POINT TO SECOND DELAY TAP         */

OP    MOV  @MEM,B    /* SET NEW W(I-2)                    */
      ADD  ACCA,M    /* A=NEW W(I-1) + W(I-2)*AN2
                        + 2*W(I-1)*AN1                    */
      M1             /* POINT TO FIRST DELAY TAP FOR
                        NEXT CALL TO THIS SUBROUTINE*/
      RET           ; /* RETURN FROM SUBROUTINE            */
```



$$H(Z) = \frac{1 + A_1 Z^{-1} + A_2 Z^{-2}}{1 + B_1 Z^{-1} + B_2 Z^{-2}}$$

FIGURE A-1.1 BIQUAD FILTER SIGNAL FLOW DIAGRAM AND TRANSFER FUNCTION

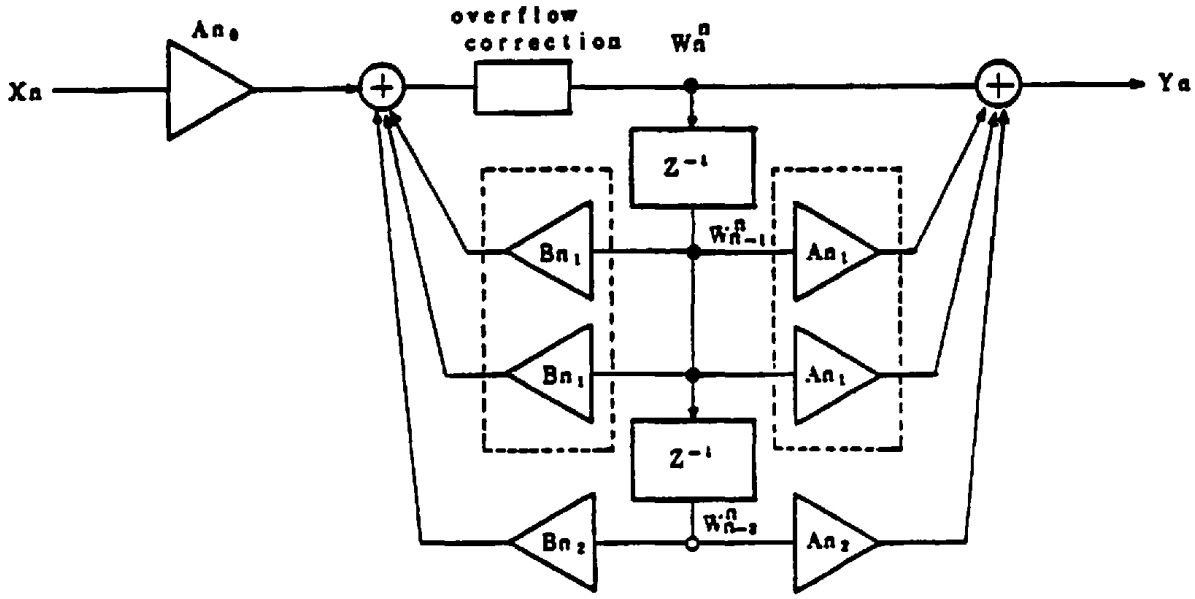
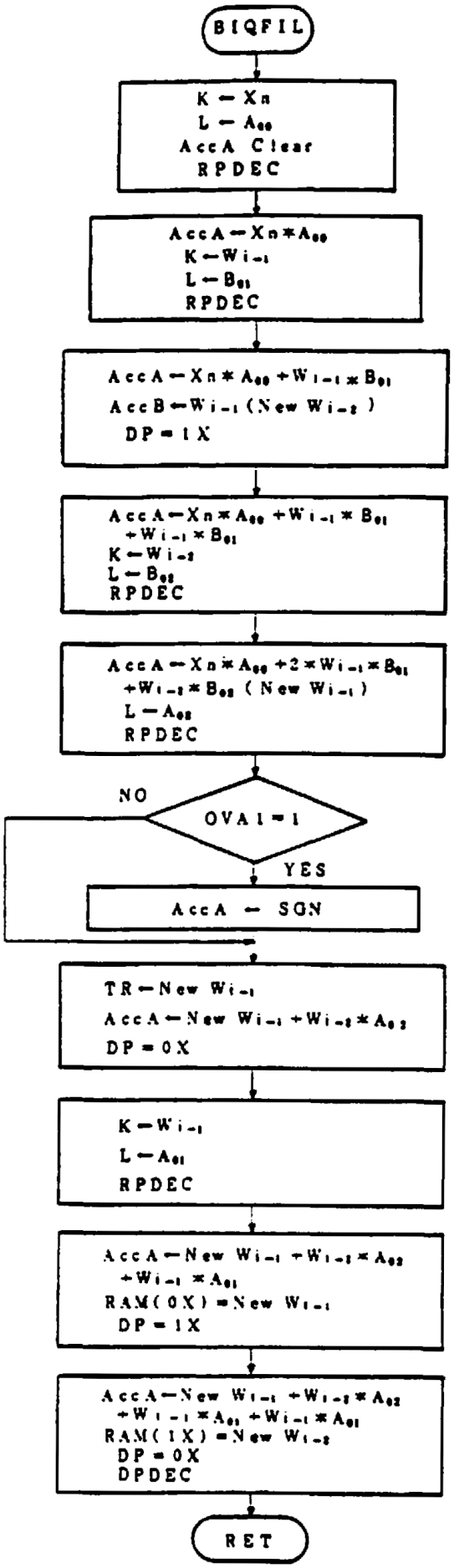


FIGURE A-1.2 MODIFIED BIQUAD FILTER SIGNAL FLOW DIAGRAM

FIGURE A-1.3 MODIFIED BIQUAD FILTER FLOW CHART



EXAMPLE B: Sixty-four Biquad Filters

This example consists of an entire program (implicitly included is the biquad filter subroutine, BIQFIL, shown in Example A) that will perform 64 biquad filter sections, each one cascading its output into the input of the next one. This will show the use of memory space and pointer manipulation for multiple filter sections, using a common filter subroutine, initialization, RAM clearing, and I/O techniques.

The data RAM memory map consists of 64 copies of the usage for the biquad filter routine shown in Example A. Each filter section uses two delay taps, one on top of the other within a column. In each column, four sections' delay taps are stacked up on top of each other (i.e. section A uses 00 and 10H, section B uses 20H and 30H, C--40H and 50H, D--60H and 70H). This is repeated for each of the 16 columns (section E would then use 01H and 11H, etc.).

The coefficient ROM again holds the filter coefficients, five for each biquad filter section. Again, for each filter section the five coefficients are stored in sequentially descending locations in the order in which the filter subroutine uses them. Section A's coefficients are in the top five ROM locations, section B's are in the next lower five locations, etc. This way, when one filter section is done, the ROM pointer will be pointing to the first coefficient for the next section.

This example assumes linear 16-bit 2's complement serial input data for each sample, as would be provided by a successive-approximation A/D converter, and outputs linear 16-bit serial data.

```
        LDI   @SR,0000H     ; /* INITIALIZE STATUS REGISTER
                                  16-BIT SERIAL I/O, ETC.      */

        LDI   @A,0000H      ; /* DATA TO FILL (CLEAR) RAM     */

        LDI   @B,007FH      ; /* STARTING ADDRESS TOP OF RAM  */

CLLOOP: OP    MOV  @DP,B    ; /* ADDRESS TO RAM POINTER       */

        OP    MOV  @MEM,A     /* DATA TO RAM (CLEAR LOCATION) */
              DEC  ACCB     ; /* DECREMENT ADDRESS            */

        JNCB  CLLOOP        ; /* IF ADDRESS NOT < 0, LOOP     */

WAITIN: JNSIAK    WAITIN    ; /* WAIT FOR SERIAL INPUT ACK.   */

        LDI   @DP,0000H     ; /* POINT TO FIRST FILTER SECTION */

        LDI   @RP,01FFH     ; /* POINT TO TOP OF COEFFICIENTS */

        OP    MOV  @A,SIM   ; /* GET SERIAL INPUT SAMPLE IN A */

BIQUAD4:                      /* THIS LOOP WILL DO THE 4 FILTER
                                 SECTIONS IN ONE COLUMN OF RAM,
```

```
                                    POINT TO THE NEXT COLUMN, AND
                                    LOOP UNTIL LAST COLUMN IS DONE*/

        CALL BIQFIL         ; /* DO FIRST FILTER SECTION IN THIS
                                    COLUMN (ADDRESSES 0X AND 1X)  */

        OP   M2             ; /* POINT TO SECOND FILTER SECTION
                                    IN COLUMN (2X AND 3X)         */

        CALL BIQFIL         ; /* DO SECOND FILTER SECTION       */

        OP   M6             ; /* POINT TO THIRD FILTER SECTION
                                    IN COLUMN (4X AND 5X)         */

        CALL BIQFIL         ; /* DO THIRD FILTER SECTION        */

        OP   M2             ; /* POINT TO FOURTH FILTER SECTION
                                    IN COLUMN (6X AND 7X)         */

        CALL BIQFIL         ; /* DO FOURTH FILTER SECTION       */

        OP   M6               /* POINT TO FIRST SECTION AGAIN   */
             DPINC          ; /* BUT POINT TO NEXT COLUMN       */

        JDPL0      OUTPUT   ; /* IF WRAPPED AROUND TO FIRST
                                    COLUMN, DONE, DO OUTPUT       */

        JMP  BIQUAD4        ; /* ELSE, LOOP FOR NEXT COLUMN     */

OUTPUT: JSOAK      OUTPUT   ; /* WAIT FOR SERIAL OUTPUT REGISTER
                                    TO BE EMPTY.  THIS STEP WOULD
                                    BE UNNECESSARY IF SERIAL INPUT
                                    AND OUTPUT ARE SYNCHRONIZED,
                                    SINCE THERE IS THE WAIT FOR
                                    SERIAL INPUT CODE AT WAITIN   */

        OP   MOV  @SOM,A    ; /* OUTPUT RESULT TO SERIAL OUTPUT
                                    REGISTER                      */

        JMP  WAITIN         ; /* JUMP BACK AND WAIT FOR NEXT
                                    INPUT SAMPLE                  */
```

EXAMPLE C: Transversal (FIR) Filter

This example shows a subroutine to execute a 16-tap transversal (FIR) filter. It not only performs the sum-of-products, but it also performs the delay function by moving the data through RAM at a rate of one location per sample. The entry to the subroutine requires that the RAM Data Pointer (DP) be pointing to the first location in the row of RAM (address 00, for instance), which will also hold the current sample if this is the first call to this subroutine, ROM Pointer (RP) be pointing to the first coefficient, located at the top of the coefficient table, Accumulator A holds either 0 or the partial sum of products so far (if not the first call to the subroutine), and Accumulator B also holds the current sample (if first call).

```
TRFIL:      OP   MOV  @KLR,MEM; /* MOVE SAMPLE AND COEFFICIENT
                                   TO MULTIPLIER INPUTS     */

            OP   ADD  ACCA,M    /* ADD PRODUCT TO SUM        */
                 MOV  @MEM,B  ; /* SAVE DELAYED SAMPLE HERE */

            OP   MOV  @B,K      /* RETRIEVE PRESENT SAMPLE
                                   FROM MULT. INPUT FOR DELAY,
                                   TO BE SAVED ON NEXT PASS */
                 RPDEC          /* POINT TO NEXT COEFFICIENT*/
                 DPINC        ; /* POINT TO NEXT DELAY TAP  */

            JDPL0     $ + 2  ; /* IF END OF ROW, SKIP NEXT */

            JMP  TRFIL       ; /* ELSE, LOOP-NEXT SAMPLE   */

            OP   M1             /* POINT TO NEXT ROW         */
                 RET          ; /* AND EXIT                  */
```

Note in the last statement that M1 is performed. This way, a 32-tap filter can be performed using two adjacent rows simply by calling TRFIL twice in a row, with no other instructions necessary between the two calls. A 64-tap filter could be done as follows: enter pointing to row 0 (address 00), CALL TRFIL; CALL TRFIL; OP M2; CALL TRFIL; CALL TRFIL; OP M2;. After the first call, DP will be pointing to row 1 for the second call. After the second call, DP will be pointing to row 0 again, so M2 is used to point to row 2 for the third call. After the third call, DP is pointing to row 3 for the fourth call, and after the fourth call, DP is pointing to row 2 again, so M2 is used to point to row 0 again for the next time through.

If the coefficients are to be stored in RAM, so that they may either be downloaded from a host, or adaptively modified by some error detection algorithm, this subroutine may be easily changed to accomodate this requirement. The coefficients should be stored in the upper half of RAM, in addresses that equal 40H plus the addresses of the delay taps that they correspond to. Then, all that needs to be done is to change the @KLR to @KLM to

do the multiplier double load, change the MOV @B,K to MOV @B,L to retrieve the delayed sample from the multiplier input, and delete the RPDEC, since the ROM Pointer is not used at all in this case.

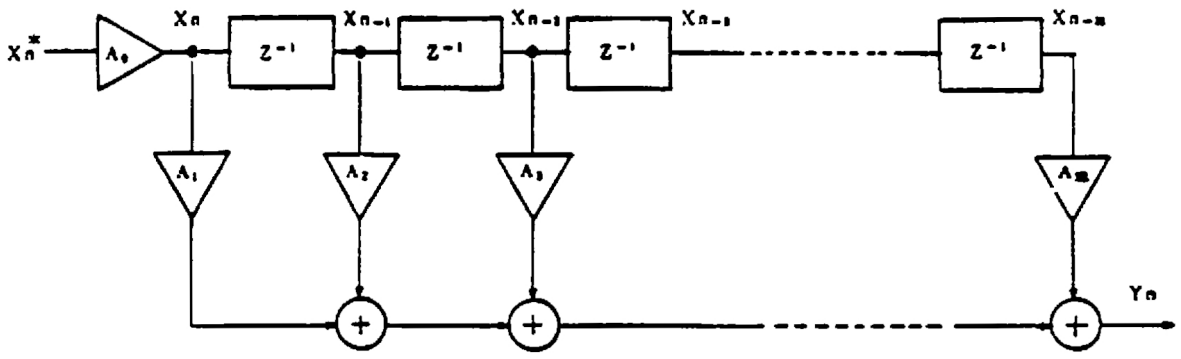
EXAMPLE D: 32-Tap Transversal Filter

This example shows a simple section of code that performs a 32-tap FIR filter using the TRFIL subroutine presented in the last example. The only requirement for this section of code is that Accumulator A holds the present input sample as 16-bit linear 2's complement data. This sample will be scaled by a preset coefficient before being input to the filter itself.

```
LDI   @RP,1FFH      ; /* POINT TO SCALAR VALUE      */

OP    MOV  @KLR,A     /* LOAD MULTIPLIER INPUTS WITH
                         SCALAR AND INPUT SAMPLE   */
      XOR  ACCA,IDB   /* CLEAR ACCUMULATOR A, TOO  */
      RPDEC         ; /* AND POINT TO FIRST
                         COEFFICIENT               */

OP    MOV  @NON,B
      XOR  ACCB,IDB; /* CLEAR ACCB                 */

OP    ADD  ACCB,M   ; /* SCALED INPUT VALUE TO B    */

OP    MOV  @MEM,B   ; /* ALSO SAVE IT IN FIRST
                         MEMORY LOCATION TO SET UP
                         CALL TO TRFIL             */

CALL TRFIL          ; /* DO FIRST 16 TAPS           */

CALL TRFIL          ; /* DO SECOND 16 TAPS          */
```

The exit condition from this section of code is that accumulator A will hold the filtered output sample. The signal flow diagram, including the input sample scaling, is shown in Figure A-1.4.

Figure A-1.4 32-TAP TRANSVERSAL FILTER SIGNAL FLOW



EXAMPLE E: Use of Parallel I/O

This example shows the use of the parallel I/O section. Included in this example are the use of the Data Register (DR), the use of the general-purpose parallel output pins (P0 & P1), and the use of the RQM flag in the Status Register. This example loads six words (16-bits each) through the DR from a host, storing them in the first six locations in RAM, calls some as-yet undefined subroutine, then outputs two words of data back to the host, from wherever the DP points to after the subroutine executes. It also signals, in both the USF0 and USF1 flags and the P0 and P1 output pins, the state it is in.

```
                LDI   @SR,0000H     ; /* INITIALIZE STATUS REGISTER
                                         DR IN 16-BIT MODE, USF0/1
                                         AND P0/1 SIGNAL STATE 0,
                                         WAITING FOR INPUTS        */

                LDI   @DP,0000H     ; /* POINT TO WORK AREA IN RAM*/

                LDI   @B,0005H      ; /* GET COUNT - 1 IN ACCB     */

                LDI   @DR,0000H     ; /* DUMMY LOAD TO SET RQM     */

INPTLP:         JRQM  INPTLP        ; /* WAIT FOR HOST TO INPUT    */

                OP    MOV  @MEM,DR    /* MOVE INPUT TO RAM         */
                      DEC  ACCB       /* DECREMENT COUNT           */
                      DPINC         ; /* POINT TO NEXT LOCATION    */

                JNZB  INPTLP        ; /* IF COUNT NOT 0, LOOP      */

                OP    MOV  @MEM,DRNF /* GET LAST INPUT VALUE, PUT
                                         IT IN RAM, BUT DON'T SET
                                         RQM, DONE GETTING INPUT   */
                      DPCLR         ; /* DP = 00 FOR SUBRTN CALL   */

                CALL  CRUNCH        ; /* CALL SOME SUBROUTINE      */

                LDI   @SR,2001H     ; /* USF0/1 & P0/1 SAY STATE 1,
                                         OUTPUTTING TO HOST        */

                OP    MOV  @DR,MEM    /* OUTPUT FIRST VALUE        */
                      DPINC         ; /* POINT TO SECOND VALUE     */

OUTPLP:         JRQM  OUTPLP        ; /* WAIT FOR HOST TO TAKE IT  */

                OP    MOV  @DR,MEM  ; /* OUTPUT SECOND VALUE       */

OUT2LP:         JRQM  OUT2LP        ; /* WAIT FOR HOST TO TAKE IT  */

                LDI   @SR,4002H     ; /* USF0/1 & P0/1 SAY STATE 2,
                                         ALL DONE                  */
```

EXAMPLE F: 32-Bit Math

This example shows some simple 32-bit math routines. It takes the six words of data input by example E, and treats them as three 32-bit words, stored low word first. The first (32-bit) word is shifted left one bit, the second word is added, the third word is subtracted, and the result is stored in the next two locations in RAM, with the DP pointing to the first one (low word of the 32-bit result). No overflow processing is done in this example. Of particular importance here is the usage of the feature of the carry flags, in which the carry flag of the unused accumulator serves as an input to the ALU in the SHL1, ADC, and SBB operations.

```
CRUNCH:     OP   MOV  @B,MEM     /* GET LOW WORD OF FIRST    */
                 AND  ACCA,IDB   /* CLEAR CA (ACCA CARRY)    */
                 DPINC         ; /* POINT TO HIGH WORD       */

            OP   MOV  @A,MEM     /* GET HIGH WORD OF FIRST   */
                 DPINC         ; /* POINT TO LOW OF SECOND   */

            OP   SHL1 ACCB    ; /* SHIFT ACCB LEFT ONE BIT,
                                   CARRY FROM A (CLEARED IN
                                   FIRST INSTR.) COMES IN   */

            OP   SHL1 ACCA    ; /* SHIFT ACCA LEFT, CARRY IN
                                   ACCB'S OLD MSB           */

            OP   ADD  ACCB,RAM  /* ADD LOW WORD OF SECOND 32-
                                   BIT VALUE TO LOW WORD OF
                                   RUNNING TOTAL            */
                 DPINC         ; /* POINT TO HIGH OF SECOND */

            OP   ADC  ACCA,RAM  /* ADD HIGH WORD OF SECOND */
                 DPINC         ; /* POINT TO LOW OF THIRD   */

            OP   SUB  ACCB,RAM  /* SUBTRACT LOW OF THIRD    */
                 DPINC         ; /* POINT TO HIGH OF THIRD  */

            OP   SBB  ACCA,RAM  /* SUBTRACT HIGH OF THIRD   */
                 DPINC         ; /* POINT TO LOW OF RESULT  */

            OP   MOV  @MEM,B     /* SAVE LOW WORD OF RESULT */
                 DPINC         ; /* POINT TO HIGH OF RESULT */

            OP   MOV  @MEM,A     /* SAVE HIGH WORD OF RESULT*/
                 DPDEC           /* POINT TO LOW OF RESULT,
                                    DESIRED EXIT CONDITION  */
                 RET           ; /* EXIT, RETURN FROM SUBRTN*/
```

APPENDIX 2 OVERFLOW PROCESSING THEORY DISCUSSION

As described in section 3.15, the OVA0, OVB0, SA0, SB0, OVA1, OVB1, SA1, and SB1 flag bits can be used to correct overflow errors after three consecutive operations, rather than having to check overflow conditions after every operation. In particular, the XXX0 bits can be used like ordinary overflow and sign bits, and the XXX1 bits are the ones that allow the three consecutive operations to be performed before overflow processing must be done.

This Appendix is intended to provide an explanation of the reasons for the unusual ways these XXX1 bits are set and used.

First, consider what happens when an arithmetic operation overflows. An overflow occurs when the operation produces a result that is either larger than 7FFFH (this discussion deals with 16-bit 2's complement arithmetic) or smaller than 8000H. Looking at a specific example, suppose 5555H is added to itself. In unsigned arithmetic, the correct answer is AAAAH. (Or, in 32 bit 2's complement arithmetic, (0000)5555 + (0000)5555 = (0000)AAAA). However, in 16 bit 2's complement, AAAAH is a negative number, and (0000)AAAAH is out of reach. Therefore, adding two positive number has yielded a negative number for a result, which is obviously incorrect. The overflow occurs when crossing the "boundary" between 7FFFH and 8000H. This can be also viewed as a boundary between different "segments" in a 32 bit universe, as shown in the following diagram:

```
-------------------------------------------------------------
(0001)7FFF
 .........      "Positive" part of next higher 16-bit segment
(0001)0000
(0000)FFFF
 .........      "Negative" part of next higher 16-bit segment
(0000)8000
-------------------------------------------------------------
(0000)7FFF
 .........      Positive part of "Base" 16-bit segment
(0000)0000
(FFFF)FFFF
 .........      Negative part of "Base" 16-bit segment
(FFFF)8000
-------------------------------------------------------------
(FFFF)7FFF
 .........      "Positive" part of next lower 16-bit segment
(FFFF)0000
(FFFE)FFFF
 .........      "Negative" part of next lower 16-bit segment
(FFFE)8000
-------------------------------------------------------------
```

With this 32-bit universe in mind, it is easier to picture what happens when 16-bit arithmetic overflows. An overflow is simply the crossing of the boundary between different segments of the longer bit-length universe, to a result whose magnitude is too large to be represented in the shorter bit-length universe,

as in (0000)5555 + (0000)5555 = (0000)AAAA. In this example, the result is in the "negative" part of the next higher segment (in the 32-bit universe), and is simply an incorrect (overflowed) negative result in the 16-bit universe. If (FFFF)AAAB (the 2's complement of (0000)5555) were then to be added, the result would be (0000)5555. Of course, in the 16-bit universe, this would again constitute an overflow condition, however the overall sum would not be an overflow condition (the order of operation could be reversed: 5555 + AAAB = 0000, 0000 + 5555 = 5555; with no overflow occurring at all). This shows the very real possibility that, in performing a sum of several numbers, there may be an intermediate result that overflowed, while the overall result does not represent an overflowed condition. Therefore, it is desireable to have a way of detecting an overall overflow after several operations, rather than having to make a decision based on an overflow condition after every operation. The XXX1 flags in the uPD7720 are intended to provide this information.

Of course, there is also the possibility of several consecutive operations remaining in an overflow condition, or even overflowing again in the same direction, with a result in a segment that is two segments away from the "base" segment. The overflow processing in the uPD7720 is designed to distinguish these situations.

Now, to explain the way the OVA(B)1 and SA(B)1 bits are set, as described in 3.15. Everything is based on what happens when an overflow occurs (OVA(B)0 is set, and a "segment boundary" is crossed). When the first overflow in a series of operations occurs, both OVA(B)0 and OVA(B)1 are set, and SA(B)0 and SA(B)1 are both set according to the sign of the (incorrect) result. If in the course of several operations, the OVA(B)0 bit is set an odd number of times, the result is obviously overflowed, since there would need to be an even number of overflows in order to cross back over any "boundaries" that were crossed in the first place, to return to the original "base" segment. Therefore, the cases of interest are those in which an even number of overflows occur.

If the OVA(B)0 bit is set twice in a row, there is no overflow, since the boundary that was first crossed has been crossed back over, into the base segment again. This is because it would be impossible to cross the next boundary in the same direction in the second operation (because it is too far away for a single 16-bit 2's complement operation to reach).

However, if OVA(B)0 is set in the order 1-0-1 (or any number of 0's in the middle, for that matter), then it is possible that either the original boundary was crossed back over, with the result returning to the base segment, or the next boundary in the same direction was also crossed, going two segments away from the base. The OVA(B)1 bit will distinguish these two cases by the behavior of SA(B)0 and SA(B)1.

Since SA(B)0 indicates the (incorrect) sign of the result of an (overflowed) operation, it will indicate the direction in which an overflow occurred. If two positive numbers were added (an overflow in the "up" direction), the sign will be negative if there was an overflow. On the first occurrence of an overflow, SA(B)1 will be set the same as SA(B)0. When the second overflow

occurs, the new SA(B)0 will be compared to SA(B)1, which saved the sign of the first overflow result. If they are the same, the overflow occurred in the same direction as the original overflow, with a result two segments away from the base, so OVA(B)1 will remain set. If SA(B)1 and SA(B)0 are not the same, the overflow occurred in the opposite direction, therefore the boundary that was originally crossed was crossed back over again, returning to the base segment, and the overall result is not overflowed, so OVA(B)1 will be cleared (even though OVA(B)0 will be set to indicate that this individual operation did overflow).

As a result, it is possible to do three consecutive operations before checking for overflows, at which time the OVA(B)1 bit should be checked to determine if the sequence of operations had on overall overflow result. Note that three operations is the maximum number that can be performed with certainty as to the validity of the flags, since it is possible to get an invalid result in four operations as follows: First operation overflows, second operation moves in the same direction, third operation overflows in the same direction. So far, the overflow bits will correctly indicate that there is an overflowed condition in this particular direction (although there is no indication, without keeping a history of the overflow bits, that the overflow is by two segment boundaries' worth). Then, the fourth operation moves in the opposite direction, overflowing. Since the overflow bits only indicate an overflow in the original direction (but not that there were two overflows' worth), this new overflow operation will leave an indication that there is no overflow, since it has crossed back over the boundary that was most recently crossed. In other words, the behavior of the flag bits at this point is identical to what happens when two consecutive overflows, one out of the base segment and one back into it, occur. There is no indication that the result is actually one segment away from the base, having gone two segments away, and come back one. For this reason, if there is no a priori knowledge about whether or not the numbers involved in the sequence of operations can produce such a doubly overflowed result, overflow checking using the OVA(B)1 bit should be performed after every three operations. This practice is illustrated in the examples in Appendix 1.

APPENDIX 3 SPI'S INTERNAL REPRESENTATION OF NUMBERS

All numbers used in multiplication are represented as fixed point 2's complement 16-bit numbers. The most significant bit is the sign bit, and the other 15 bits represent magnitude. The radix point is always just to the left of the most significant magnitude bit. Range: $-1 =< n < 1$. One LSB is therefore equal to 2 ^ -15. This is illustrated in the following table (in which K = 1 LSB = 1/(2 ^ 15) = 0.00003051757813):

HEX	BASE 10 (n)	NORMALIZED BASE 10 (nK)
7FFF	32767	0.9999694824
....	.....	.....
4000	16384	0.5
....	.....	.....
2000	8192	0.25
....	....	.....
0002	2	0.0000610352
0001	1	0.0000305176
0000	0	0.0
FFFF	-1	-0.0000305176
FFFE	-2	-0.0000610352
....	....	.....
E000	-8192	-0.25
....	....	.....
C000	-16384	-0.5
....	.....	.....
8001	-32767	-0.9999694824
8000	-32768	-1.0

This representation applies to the multiplier output registers, M and N, in a similar fashion. Since two numbers are being multiplied that each have 15 bits of magnitude, the product will have 30 bits of magnitude. The M and N registers concatenated provide 32 available bits. The MSB (MSB of M) is the sign bit, and then the next 30 bits (the 15 remaining in M and the most significant 15 bits of N) represent the magnitude (in 2's complement form). The LSB of N is filled with a 0.

This format can lead to some initial confusion for someone who wants to think of the numbers as integers, not fractions. For example 0001 X 0001 = 00000002, because (2 ^ -15) X (2 ^ -15) = 2 ^ -30. Also, 4000H X 4000H = 20000000H, because 0.5 X 0.5 = 0.25 (1/2 X 1/2 = 1/4). If multiplied as integers, of course, 4000H X 4000H would be 10000000H. These are simply the results of the 30 bit magnitude information being left-justified in a 31 bit field.

Note: multiplying 8000H X 8000H will lead to an error condition, in which the M:N output will be 80000000H. This occurs because -1 X -1 = +1, and +1 would have to be represented by 7FFFFFFFH + 1, which overflows to 80000000H, i.e. +1 is one LSB too large to be represented in this 2's complement notation.