This driver enables easy implementation of the Am9511 as a coprocessor in Intel 8085-based systems.

An Efficient Software Driver for Am9511 Arithmetic Processor Implementation

Borivoje Furht, University of Miami

Peter Lee, IBM, Corp.

An arithmetic processor unit interfaced to a host microprocessor system as a coprocessor significantly improves system performance, performing both a variety of fixed and floating point arithmetic operations and trigonometric and mathematical operations. An efficient software driver has been developed that enables easy implementation of the arithmetic processor unit (an Am9511 or Intel 8231A) in a concurrent operation with a host microprocessor (e.g., Intel 8085).

Basic hardware configuration

The Am9511 arithmetic processing unit can be interfaced to an Intel 8085/8086 host microprocessor just as any programmable I/O unit can. All transfers between the microprocessor and the arithmetic processing unit are performed using an 8-bit directional data bus. Transfers to and from the APU are handled by using a conventional programmed I/O technique. To provide I/O mapped I/O an 8205 decoder is used to get the chip select to the APU with IO/$\overline{M}$ signal equal 1. The signals $\overline{RD}$ and $\overline{WR}$ are tied directly to the APU. Since the upper address lines (A_8-A_{15}) are non-multiplexed, they are used for selection of the APU. This technique relies on the fact that the I/O port number is copied onto A_8-A_{15} as well as A_0-A_7 during an input or output instruction. The basic hardware interface scheme is shown in Figure 1.

The signal $\overline{PAUSE}$ from the APU is tied to the READY line of the 8085 to avoid data loss. This line is high in its initial state and is pulled low by the APU under the following conditions:

- A previously initiated operation of the APU is in progress and a command entry is attempted.
- A previously initiated operation of the APU is in progress and stack access has been attempted.
- The APU is not busy and data removal or data entry has been requested.

Am9511 operation

The Am9511 APU operation is based on commands supplied by a host microprocessor. Each command represents a single 8-bit datum, shown in Figure 2.

- Bits 0-4 select the operation of the APU.
- Bits 5-6 select the data format for the operation. If bit 5 = 1, then a fixed-point datum is specified. If bit 5 = 0, floating-point format is specified. Bit 6 selects the precision of the data. If bit 6 = 1 single precision (16 bits) is selected, while bit 6 = 0 selects double precision (32 bits).
- Bit 7 indicates whether a service request is to be issued after the command is executed. If bit 7 = 1, a service request will be generated; if bit 7 = 0, a service request will not be generated.

0272-1732/84/0600-0007$01.00 © 1984 IEEE

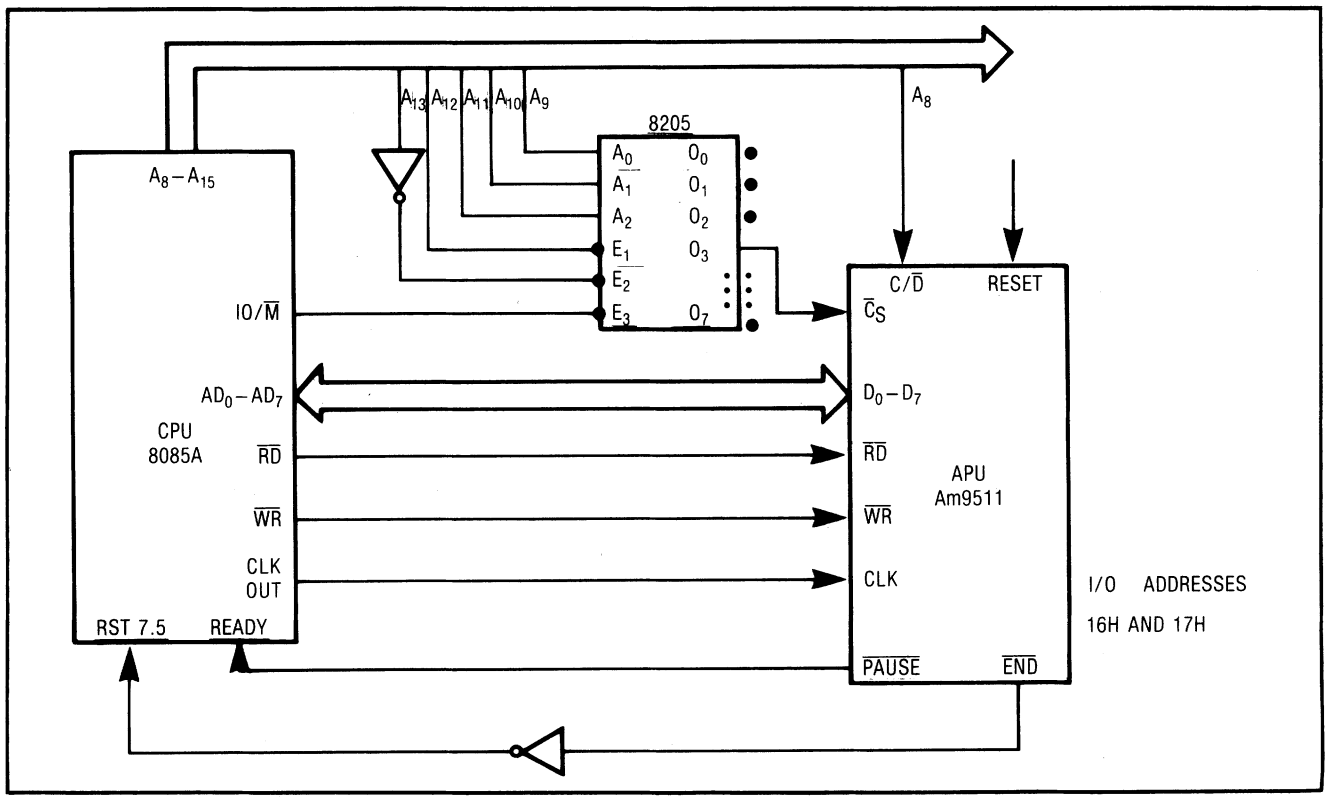


Figure 1. Basic hardware interface.

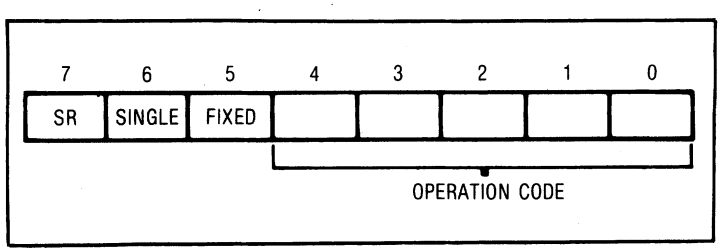


Figure 2. Command format.

A summary of the APU commands is given in Table 1. (In Table 1, TOS means top of stack and NOS means next of stack. In the hex-code column, SVREQ is a 0.

The Am9511 APU is a stack-oriented processor, so the host processor has access to an eight-level, 16-bit-wide data stack. Single-precision fixed-point operands are 16 bits wide, so eight such values may be stored in the stack (Figure 3a). When the data are double-precision fixed-point or floating-point operands (32 bits), the four values may be stored (Figure 3b). Data are written into the stack byte after byte, in the order shown in Figure 3 (B1, B2, B3, . . .). Data are removed from the stack in reverse order (B8, B7, B6, . . .).

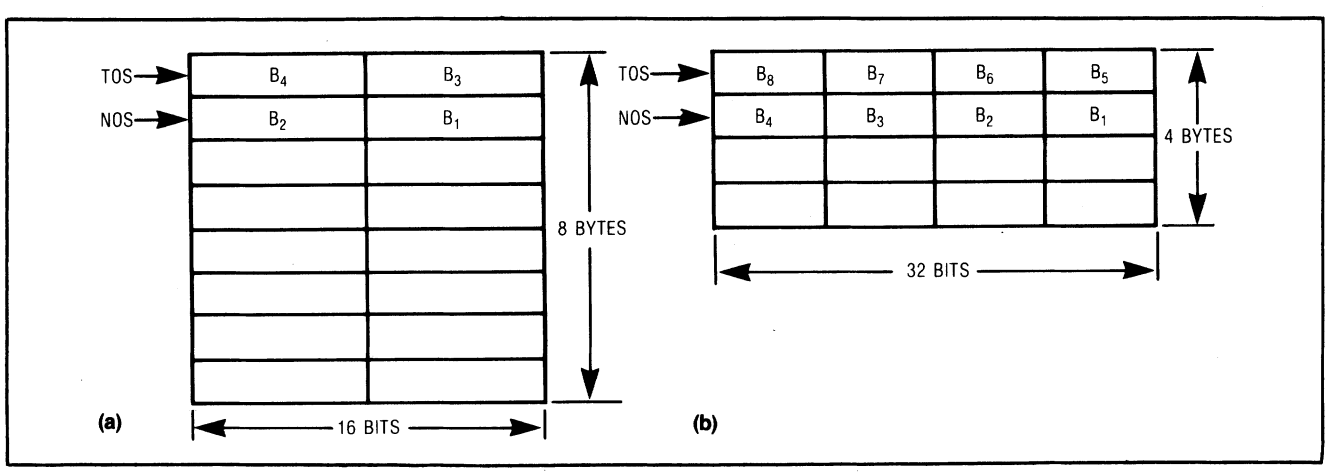


Figure 3. Structure of the APU's stack: (a) single-precision fixed-point operands; (b) double-precision fixed-point or floating point operands.

Execution times

Timing for execution of the Am9511 commands is shown in Table 2. Speeds are given in terms of clock cycles and can be converted to microseconds by multiplying by the clock period used. Maximum clock frequency for the Am9511 is 2 MHz and for the Am9511-1 is 3 MHz. One version of the Intel 8231A is faster (4 MHz).

Interface strategies

Two different interface techniques have been applied: pseudopolling and interrupt-driven interfaces.

Pseudopolling interface technique. In this mode of operation the host processor sends one command after another to the APU using a standard output operation. Since a previously initiated operation will usually be in progress when a new command entry is attempted, the APU will pull the $\overline{\text{PAUSE}}$ line low, causing the host processor to skip into a waiting state. The $\overline{\text{PAUSE}}$ line will remain low until completion of the current command execution. It will then go high, permitting entry of the new command.

The software driver for the pseudopolling mode of operation (SDPPM) is simple and consists of a set of subroutines that send commands to the APU, have access to the APU stack, and remove/enter data from/to the

Table 1.
Command summary.

COMMAND MNEMONIC	COMMAND DESCRIPTION	HEX COMMAND CODE
	FIXED POINT 16 BIT	
SADD	Add TOS to NOS. Result to NOS. Pop Stack.	6C
SSUB	Subtract TOS from NOS. Result to NOS. Pop Stack.	6D
SMUL	Multiply NOS by TOS. Lower half of result to NOS. Pop Stack.	6E
SMUU	Multiply NOS ty TOS. Upper half of result to NOS. Pop Stack.	7C
SDIV	Divide NOS by TOS. Result to NOS. Pop Stack.	6F
	FIXED POINT 32 BIT	
DADD	Add TOS to NOS. Result to NOS. Pop Stack.	2C
DSUB	Subtract TOS from NOS. Result to NOS. Pop Stack.	2D
DMUL	Multiply NOS by TOS. Lower half of result to NOS. Pop Stack.	2E
DMUU	Multiply NOS by TOS. Upper half of result to NOS. Pop Stack.	3C
DDIV	Divide NOS by TOS. Result to NOS. Pop Stack.	2F
	FLOATING POINT 32 BIT	
FADD	Add TOS to NOS. Result to NOS. Pop Stack.	10
FSUB	Subtract TOS from NOS. Result to NOS. Pop Stack.	11
FMUL	Multiply NOS by TOS. Result to NOS. Pop Stack.	12
FDIV	Divide NOS by TOS. Result to NOS. Pop Stack.	13
	DERIVED FLOATING POINT FUNCTIONS	
SQRT	Square Root of TOS. Result in TOS.	01
SIN	Sine of TOS. Result in TOS.	02
COS	Cosine of TOS. Result in TOS.	03
TAN	Tangent of TOS. Result in TOS.	04
ASIN	Inverse Sine of TOS. Result in TOS.	05
ACOS	Inverse Cosine of TOS. Result in TOS.	06
ATAN	Inverse Tangent of TOS. Result in TOS.	07
LOG	Common Logarithm (base 10) of TOS. Result in TOS.	08
LN	Natural Logarithm (base e) of TOS. Result in TOS.	09
EXP	Exponential (e^X) of TOS. Result in TOS.	0A
PWR	NOS raised to the power in TOS. Result in NOS. Pop Stack.	0B
	DATA MANIPULATION DEMANDS	
NOP	No Operation	00
FIXS	Convert TOS from floating point to 16-bit fixed-point format.	1F
FIXD	Convert TOS from floating point to 32-bit fixed-point format.	1E
FLTS	Convert TOS from 16-bit fixed-point to floating-point format.	1D
FLTD	Convert TOS from 32-bit fixed-point to floating-point format.	1C
CHSS	Change sign of 16-bit fixed-point operand on TOS.	74
CHSD	Change sign of 32-bit fixed-point operand on TOS.	34
CHSF	Change sign of floating-point operand on TOS.	15
PTOS	Push 16-bit fixed-point operand on TOS to NOS (Copy).	77
PTOD	Push 32-bit fixed-point operand on TOS to NOS (Copy)	37
PTOF	Push floating-point operand on TOS to NOS (Copy).	17
POPS	Pop 16-bit fixed-point operand from TOS. NOS becomes TOS.	78
POPD	Pop 32-bit fixed-point operand from TOS. NOS becomes TOS.	08
POPF	Pop floating-point operand from TOS. NOS becomes TOS.	38
XCHS	Exchange 16-bit fixed-point operands TOS and NOS.	39
XCHD	Exchange 32-bit fixed-point operands TOS and NOS.	39
XCHF	Exchange floating-point operands TOS and NOS.	19
PUPI	Push floating-point constant "π" onto TOS. Previous TOS becomes NOS.	1A

Table 2.
Command execution times.

COMMAND MNEMONIC	CLOCK CYCLES	COMMAND MNEMONIC	CLOCK CYCLES
SADD	17	SQRT	800
SSUB	30	SIN	4464
SMUL	84-94	COS	4118
SMUU	80-98		
SDIV	84-94	TAN	5754
DADD	21	ASIN	7668
DSUB	38	ACOS	7734
DMUL	194-210	ATAN	6006
DMUU	182-218	LOG	4474-7132
DDIV	208		
FIXS	92-216	LN	4298-6956
FIXD	100-346	EXP	3794-4876
FLTS	98-186	PWR	8290-12032
FLTD	98-378		
FADD	54-368	NOP	4
FSUB	70-370	CHSS	23
FMUL	146-168	CHSD	27
FDIV	154-184	CHSF	18
PTOS	16	POPF	12
PTOD	20	XCHS	18
PTOF	20	XCHD	26
POPS	10	XCHF	26
POPD	12	PUPI	16

APU stack. The software driver SDPPM, written in assembly language for the 8085, requires 256 bytes of memory.

A typical command routine for floating point multiplication is shown below:

```
FMUL:  MVI A, 12H   ; Code for multiplication
       OUT 17H      ; Send command to the APU
       RET
```

A typical entry routine ENT16 for entering a 16-bit value into the APU stack has the following form:

```
ENT16: MOV A, M
       OUT 16H      ; Enter low byte
       INX H        ; Increment pointer
       MOV A, M
       OUT 16       ; Enter high byte
       RET
```

Before calling a routine for data entry/removal, the user should load the register pair H&L with the address of data.

The user program is a simple sequence of CALL instructions to the corresponding APU routines.

Example 1. The following example illustrates the implementation of the SDPPM by a user program. Suppose the following computation should be performed as

$$y = \sqrt{x \cdot \sin x}$$

where x is a 16-bit integer.

Here is an efficient way to perform the desired function:

- Load the 16-bit integer value of x in the APU.
- Execute APU instruction FLTS to convert to floating point.
- Execute APU instruction PTOF to duplicate the result of the floating-point conversion.
- Execute APU instruction SIN to replace the value of x at the top of the stack with its sine.
- Execute APU instruction FMUL to compute $x \cdot \sin x$.
- Execute APU instruction SQRT to compute square root of $x \cdot \sin x$.
- Finally, remove the result from the APU stack.

The user program that performs the computation of y is the following sequence of CALL instructions:

```
LXI   H, XDATA ;Pointer to x
CALL  ENT16    ;Load 16-bit data into the APU
CALL  FLTS     ;Convert to floating point
CALL  PTOF     ;Duplicate x
CALL  SIN      ;Sin(x)
CALL  FMUL     ;x·sin(x)
CALL  SQRT     ;Square root of x·sin(x)
LXI   H, YDATA ;Pointer to y
CALL  REM32    ;Remove 32-bit result from the
                APU
```

The execution time analysis of the given example is shown in Figure 4.

As can be seen from Figure 4, the microprocessor is in a waiting state during the operation of the APU, so a concurrent operation of the μP and the APU is not realized.

Interrupt interface technique. In this mode of operation, the signal $\overline{\text{END}}$, generated by the APU, is used to activate an interrupt service routine in the μP.

The host microprocessor, after preparing a set of commands and the corresponding data for the APU, is able to continue the execution of its own program.

In order to enable a concurrent operation of the μP and the APU, the following strategy, consisting of two steps, has been implemented. First, the user commands to the APU, given in a form of macroinstructions, are not executed directly but are temporarily stored in a command buffer in RAM. Also, the corresponding data are not entered directly onto the APU stack, but its addresses-pointers to the data are temporarily stored into a pointer buffer in RAM. The command buffer is 20 bytes long, so the user can write an operation consisting of a maximum of 20 APU commands. The data buffer is 10 bytes long, so five 16-bit pointers to the data can be stored in it.

In the second step, at the end of a macro sequence, the user activates an interrupt service routine by using a CALL instruction. The interrupt service routine coordinates the execution of the commands previously stored in the command buffer. Upon completion of each command from the command buffer, the APU will issue an "end of execution signal" that will again activate the interrupt service routine. Then the ISR will send the next command from the command buffer to the APU.

Software driver for the interrupt mode. In defining software requirements for an APU software driver for the interrupt mode of operation (SDIM), the main objectives were

- to make the use of the Am9511 easy, efficient, and flexible from the user's point of view, and
- to provide a concurrent operation of the host μP and the APU.

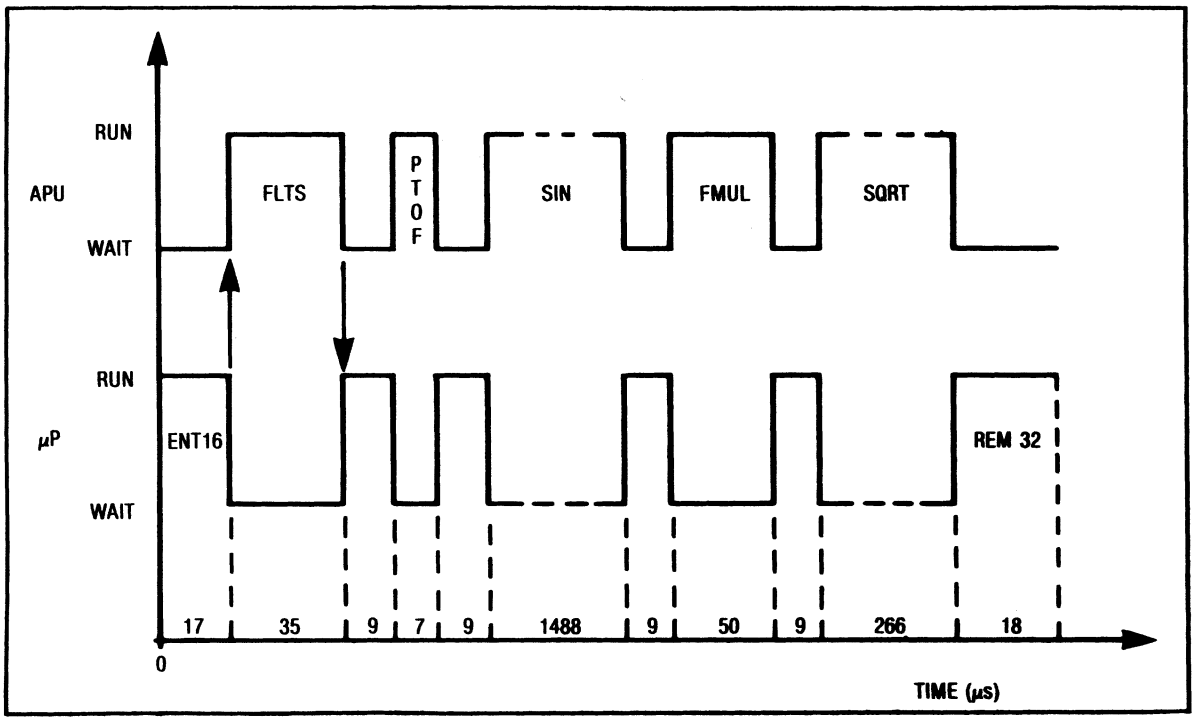


Figure 4. Execution time analysis: pseudopolling technique.

The designed software-driver SDIM consists of a set of macro definitions that perform all necessary data/command preparation and an interrupt service routine that coordinates commands execution.

Data entry/removal macros. There are two data-entry and two data-removal macroinstructions that initialize 16-and 32-bit data transfer. These macroinstructions store the pointers to the data into the pointer buffer and an internal code for the corresponding command (entry or removal) into the command buffer.

For example, macroinstruction ENT16 DATAX stores the pointer to the data, with a symbolic address DATAX, into the pointer buffer, and the command 40H, that corresponds to 16-bit data-entry into the command buffer. The macro definition for ENT16 has the following format:

```
ENT16 MACRO DATA
      LXI   D, DATA  ;Pointer to the data . . .
      MOV   M, E
      INX   H
      MOV   M, D
      INX   H        ; . . . store into the PB
      MVI   A, 40H   ;Code store into . . .
      STAX  B
      INX   B        ; . . . the CB
      ENDM
```

The internal codes for the entry/removal instructions are as follows:

Instruction	Hex Code
ENT16	40
ENT32	41
REM16	50
REM32	51

Register pairs HL and BC are used as the pointers to the pointer buffer and the command buffer, respectively. In order to initialize any sequence of the APU operations, the user has to set these pointers to the top of the pointer buffer and the command buffer by using the macroinstruction INIT at the beginning of an APU sequence. Also, the macro INIT clears the command buffer and the pointer buffer. The macro definition INIT has the following format:

```
INIT   MACRO PBUF,CBUF
       LXI   H,PBUF    ;Clear the pointer buffer
       MVI   D,10
       XRA   A
LOOP1: MOV   M,A
       INX   H
       INR   D
       JNZ   LOOP1
       LXI   B,CBUF    ;Clear the command
                        buffer
       MVI   D,20
LOOP2  STAX  B
       INX   B
       INR   D
       JNZ   LOOP2
       LXI   H,PBUF    ;Pointer to the top of
                        the pointer buffer
       LXI   B,CBUF    ;Pointer to the top of
                        the command buffer
       ENDM
```

Command entry macros. There are 42 command entry macro definitions that store the corresponding hex code (see Table 1) into the next available location of the command buffer and then increment the pointer to the command buffer. For example, macro definition FMUL for floating-point multiplications has the following format:

```
FMUL MACRO
     MVI    A,92H ; Code for multiplication
     STAX   B     ; Store code into the
                    command buffer
     INX    B     ; Increment pointer to
                    the command buffer
     ENDM
```

All the other APU commands have the same structure of macro definition.

Interrupt service routine. The interrupt service routine coordinates the execution of the commands previously stored in the command buffer. The ISR also accomplishes data entry onto the APU stack and data-removal from the APU stack, using the pointers to the data previously stored in the pointer buffer. The current pointer to the data is always stored at the top of the pointer buffer, and the current command code is always stored at the top of the command buffer. After completion of a data entry, the current pointer will be removed and the next pointer from the pointer buffer will be shifted to the top of the pointer buffer. Similarly, after completion of a command entry to the APU, the command code from the command buffer will be shifted to the top of the command buffer. This avoids the necessity of saving pointers of the pointer buffer and the command buffer.

The flowchart of the ISR is shown in Figure 5, and the corresponding program is given in the appendix. When the last command from an APU sequence has been completed, the ISR will set an "end of the APU sequence" flag in order to inform the main program of the completion of the whole APU operation. This flag can be at any location in memory. In the designated SDIM the flag is at the top location of the pointer buffer.

Example 2. In order to illustrate the implementation of the SDIM, consider the same expression $y = \sqrt{x \cdot \sin x}$ given in Example 1. The user program is a sequence of macro references given below. At the beginning the user should reserve memory locations for the pointer buffer and command buffer.

```
        ; Memory reservation
PBUF:   DS     10    ; Pointer buffer
CBIF:   DS     20    ; Command buffer
X:      DS     2     ; 16-bit data
Y:      DS     4     ; 32-bit result
        ; The APU sequence y=square root(x·sinx)
START:  INIT         ; Initialization
        ENTI6  X     ; Enter 16-bit data
        FLTS         ; Convert to floating-
                       point
        PTOF         ; Duplicate X
        SIN          ; Sin x
        FMUL         : x·sin x
        SQRT         ; Square root of x·sin
        REM32        ; Remove 32-bit data
        CALL   ISR   ; Call interrupt service
                       routine
                     ; to activate the APU
                       operation
```

After execution of the whole sequence of macroinstructions, the APU is not yet activated; however, the pointer buffer contains the pointers to the data x and y, and the command buffer contains the codes of the used APU commands, as shown in Figure 6.

In order to compare the described interrupt-interface technique and the pseudopolling technique, the execution time analysis for Example 2 is shown in Figure 7.

It is obvious from Figure 7 that the microprocessor is never in a waiting state, so the microprocessor and the APU operate concurrently. After preparing a set of APU commands and the corresponding data in the command buffer and the pointer buffer, the host microprocessor is able to continue the execution of its own program.

In the case when microprocessor program needs a result from a previously activated APU sequence, it can test the "end of the APU sequence" flag set by the ISR at the end of an APU sequence.

APU implementation in a multimicroprocessor system

In recent years the cost of microprocessor components has been reduced, so the concept of applying multiple processors to achieve better system performance, previously avoided, has now become an attractive and feasible design strategy. Since the cost of an APU is still high, however, including more than one APU in an MMP system would be very expensive.

In this section we propose an MMP configuration with multiple processors that share a single APU. The APU software driver described can be used in this MMP system.

A multiple master/dual bus multimicroprocessor architecture has been applied and described by Adams and Rolander.[4] Each processor-master has its own private memory and I/O. Access to the common bus occurs only when a master requires access to the common memory or common I/O. A parallel bus priority technique has been used which enables up to eight masters to run concurrently. The control logic for the common bus access was designed using an Intel 8219 bus-controller.

An Am9511 APU is a common I/O unit shared by all processors. The corresponding APU software drivers are the same for all processors and stored in their local memories. When a master requires an APU operation, it executes the corresponding APU driver from its local memory and activates the APU.

After completing a single operation, the APU sends an interrupt signal to the corresponding master.

The architecture of an MMP system with common APU is shown in Figure 8. Note that there needs to be additional logic to allow each processor select to either the local or common bus. Showing the details of the selection would make the figure complicated, so they are omitted.

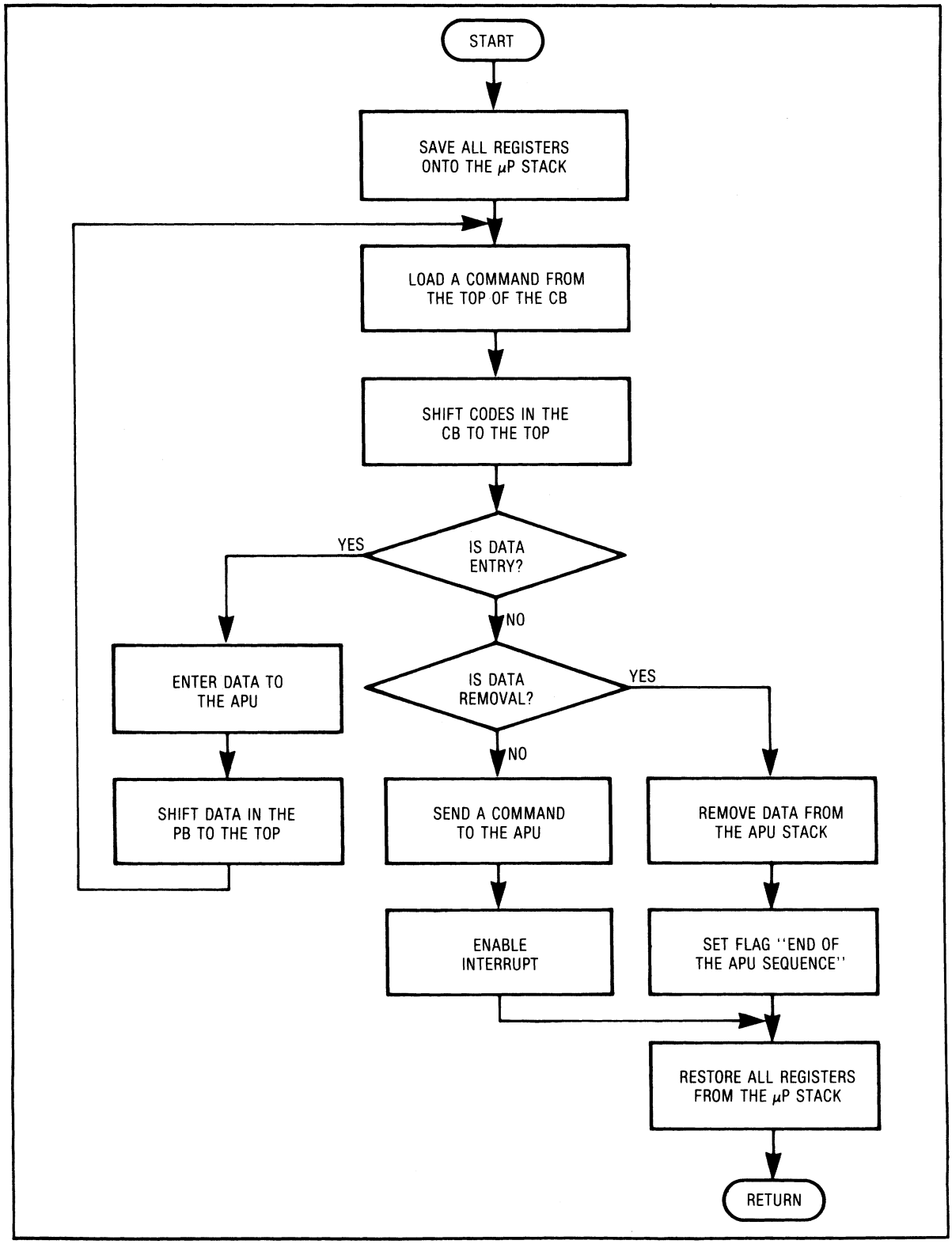


Figure 5. The interrupt service routine.

Two problems have to be solved in the MMP configuration. The first problem is mutual exclusion. It is necessary to assure that a master will have uninterrupted access to the APU. Therefore, a critical section of code has been introduced which, once begun, must complete its execution without interruption. This critical sequence has been controlled by using the hardware/software solution described below. The second problem is the interrupt signal, sent by the APU after completion of its operation, that is interfaced to all the masters (Figure 8). In order to cause the ac-

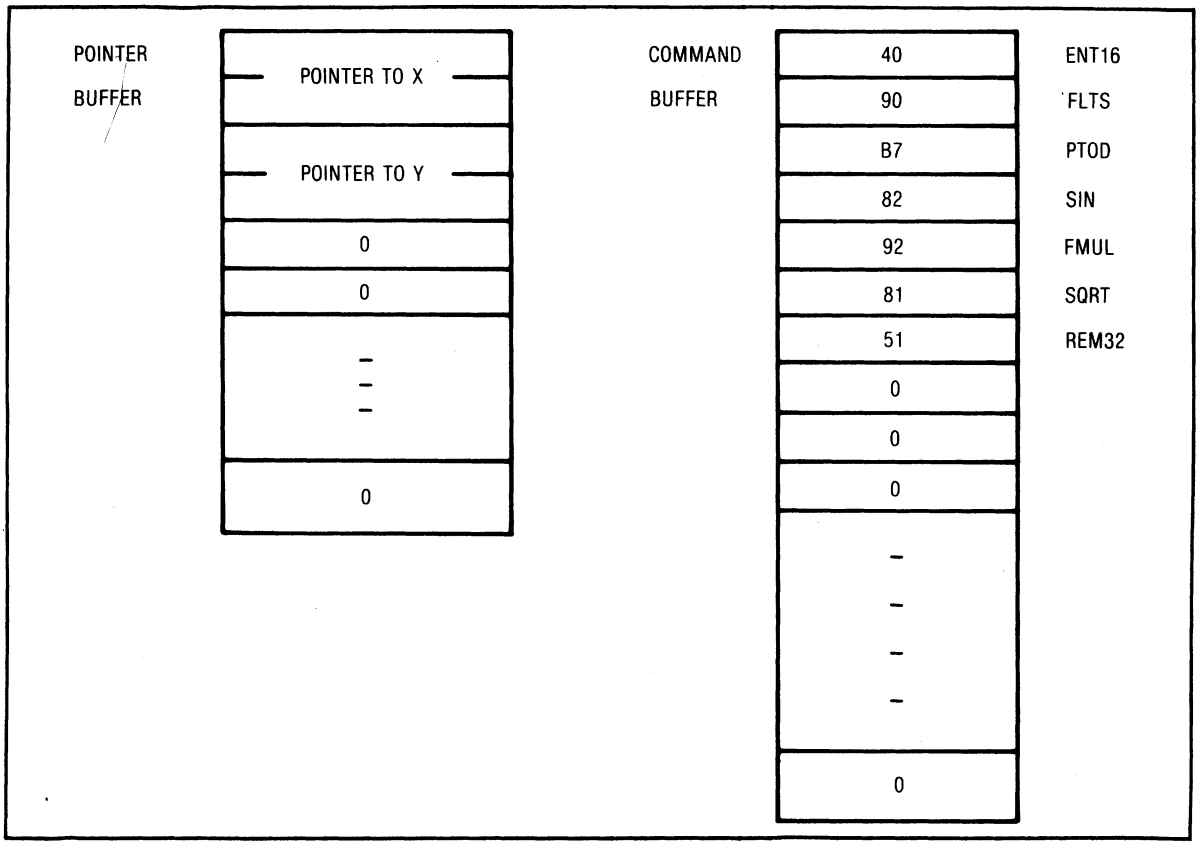


Figure 6. The contents of the pointer buffer and the command buffer in Example 2.

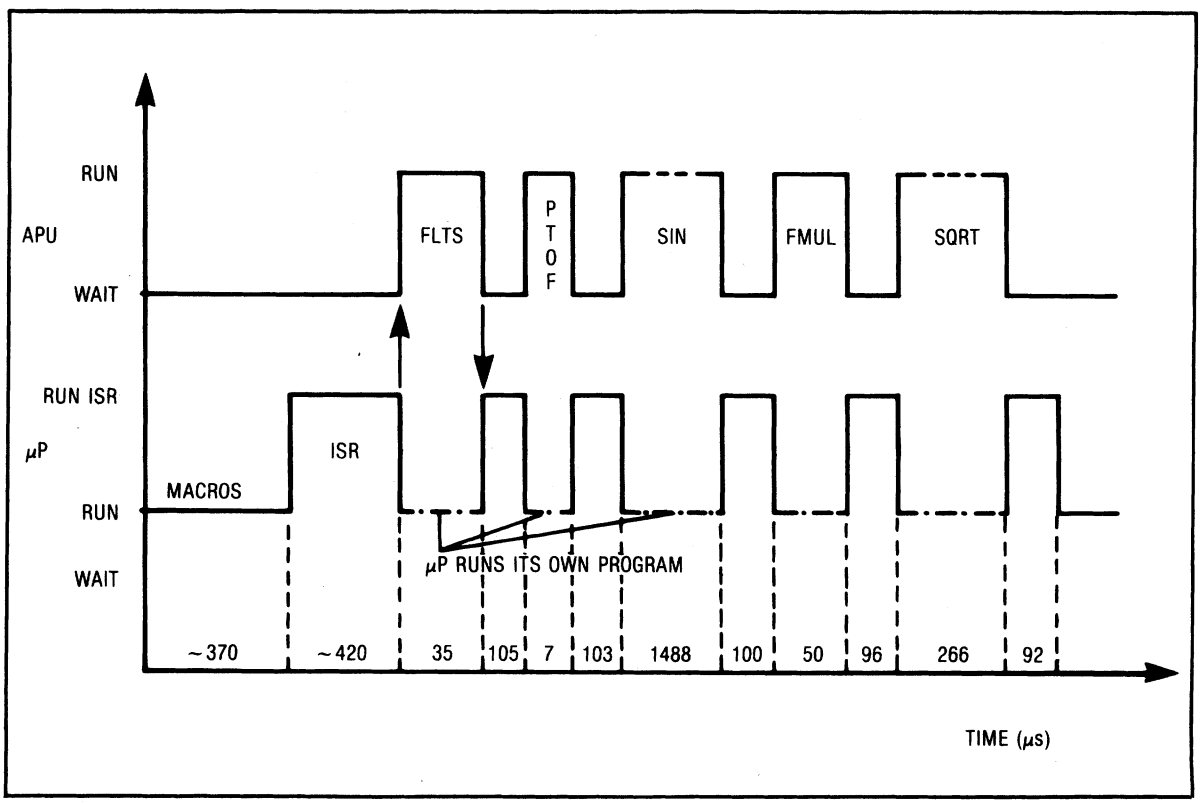


Figure 7. Execution time analysis: interrupt technique.

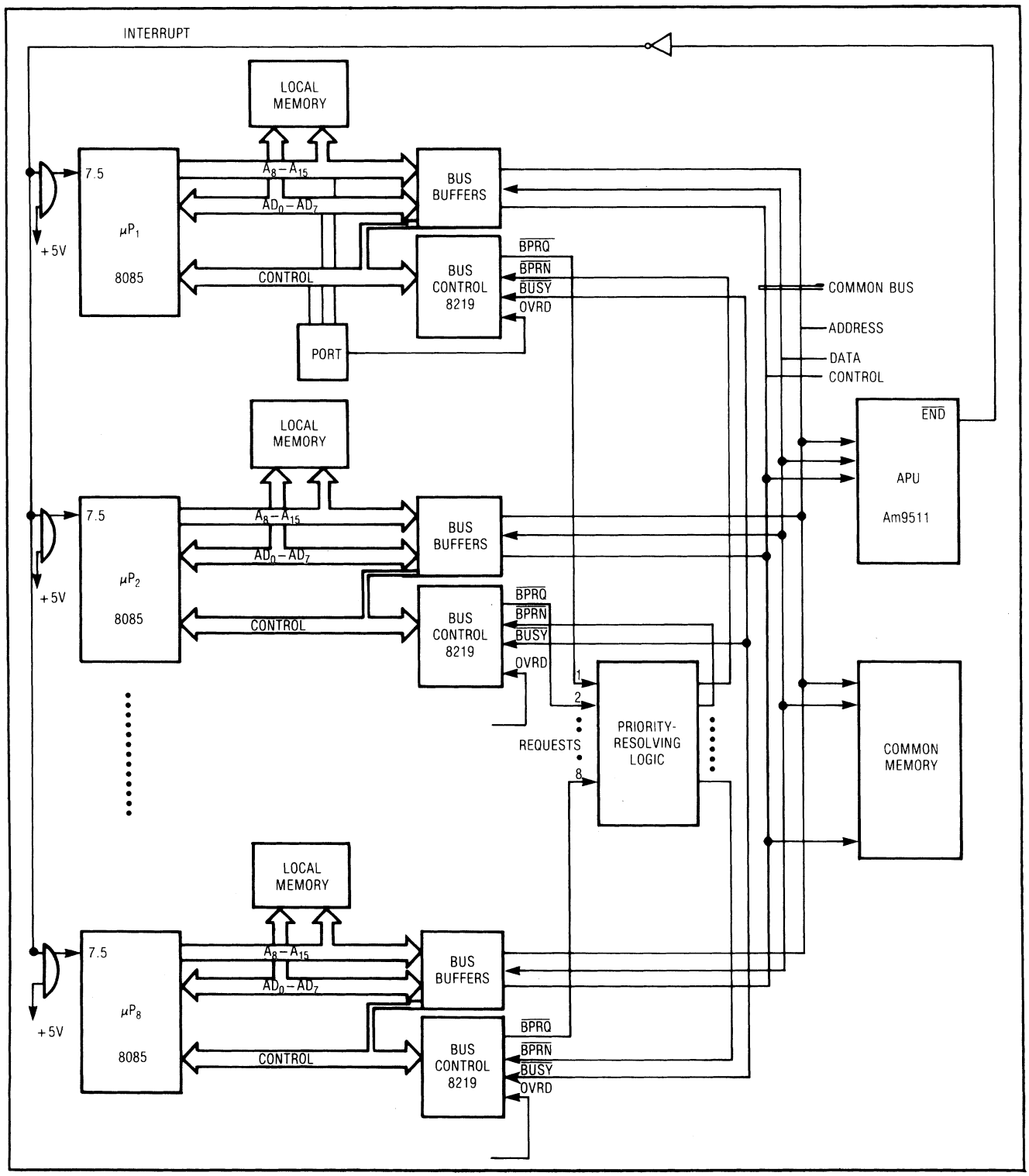


Figure 8. Multimicroprocessor system with a common APU.

tivation of the interrupt routine in the master which had activated the APU, a software interrupt-mask technique should be used.

The software driver SDIM, described in the previous section, can be easily extended in order to be used in the proposed MMP system with a common APU. First, a subroutine TEST (see Figure 9) is introduced into the driver that controls the critical section of code and that solves the mutual exclusion problem. The routine TEST should be called by the user program after the preparation of the commands has been completed and before the execution of the ISR.

Mutual exclusion is controlled by an "APU busy flag" in the common RAM memory set by the processor which

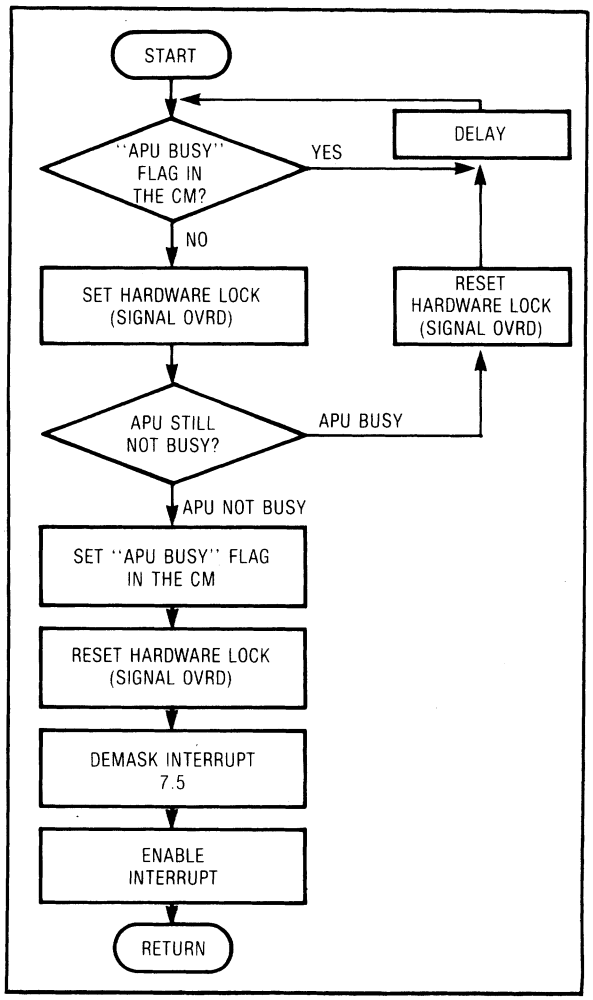


Figure 9. The routine TEST that controls the critical section of code.

uses the APU, and using a hardware signal "Override" generated by the microprocessor through an output port. The override signal temporarily locks the access of the other processors to the common bus.

Finally, the APU interrupt routine ICR should also be modified in order to be applied in the MMP system. The "APU busy" flag should be set at the beginning of the ISR, and the same bit should be reset at the end of an APU sequence. At the same time, the interrupt mask for 7.5 should be set in order to disable the activation of the ISR in the processor.

Example 3. A simple example illustrates the implementation of the described software driver in an MMP system in which two microprocessors share an APU.

Suppose that processor 1 (P1) should perform the following operation:

$$x = a*b$$

while processor 2 (P2) performs the following operation:

$$y = c*d$$

The user programs, stored in the local memories of P1 and P2, are as follows:

PROCESSOR	**1**	**PROCESSOR**	**2**
INIT		INIT	
ENT16	A	ENT16	C
FLTS		FLTS	
PLTD		PLTD	
ENT16	B	ENT16	D
FLTS		FLTS	
FMUL		FADD	
REM32	X	REM32	Y
CALL	TEST	CALL	TEST
CALL	ISR	CALL	ISR

Each microprocessor can be in one of the three possible states:

- running state (RUN), executing its main program;
- ISR-state, executing its interrupt APU service routine;
- waiting state, when the APU is busy executing an APU sequence of another processor.

The APU itself can be in either RUN or WAIT state. The timing diagram, shown in Figure 10, illustrates the operation of two processors that share an APU.

- Suppose that at time t_1 processor 1 starts the execution of an APU sequence, preparing the commands and the data.
- At time t_2, P_1 has completed the execution of its macro sequence, storing all necessary commands and data into its command and pointer buffer, respectively. It executes the routine TEST and, since the APU is not busy, it starts the execution of the APU commands. Because of the use of this mutual exclusion technique, the access to the APU is inhibited for the other processors (in this example, for processor 2).
- At time t_3 processor 2 also starts the execution of its APU sequence, preparing the corresponding commands and data.
- At time t_4, P2 starts the execution of its TEST routine, testing the "APU busy" flag in the common memory. Since the APU is still busy executing the sequence of P1, P2 starts the execution of a waiting loop, shown in Figure 9.
- At time t_5, the APU operation of P1 has been completed, so the ISR of P1 will clear the "APU busy" flag in the CM. P2, which was in a waiting loop testing this flag, will obtain an access to the APU and will start the execution of its APU sequence.

This analysis can be extended to the case when several processors share an APU.

The APU software driver in a high-level language environment

The APU software driver SDPPM can be easily modified and used in a high-level language environment.

Intel's 8085 Floating-Point Arithmetic Library contains basic floating-point subroutines and functions referred to as procedures.[6] The FPAL can be used by assembly language or PL/M programs.

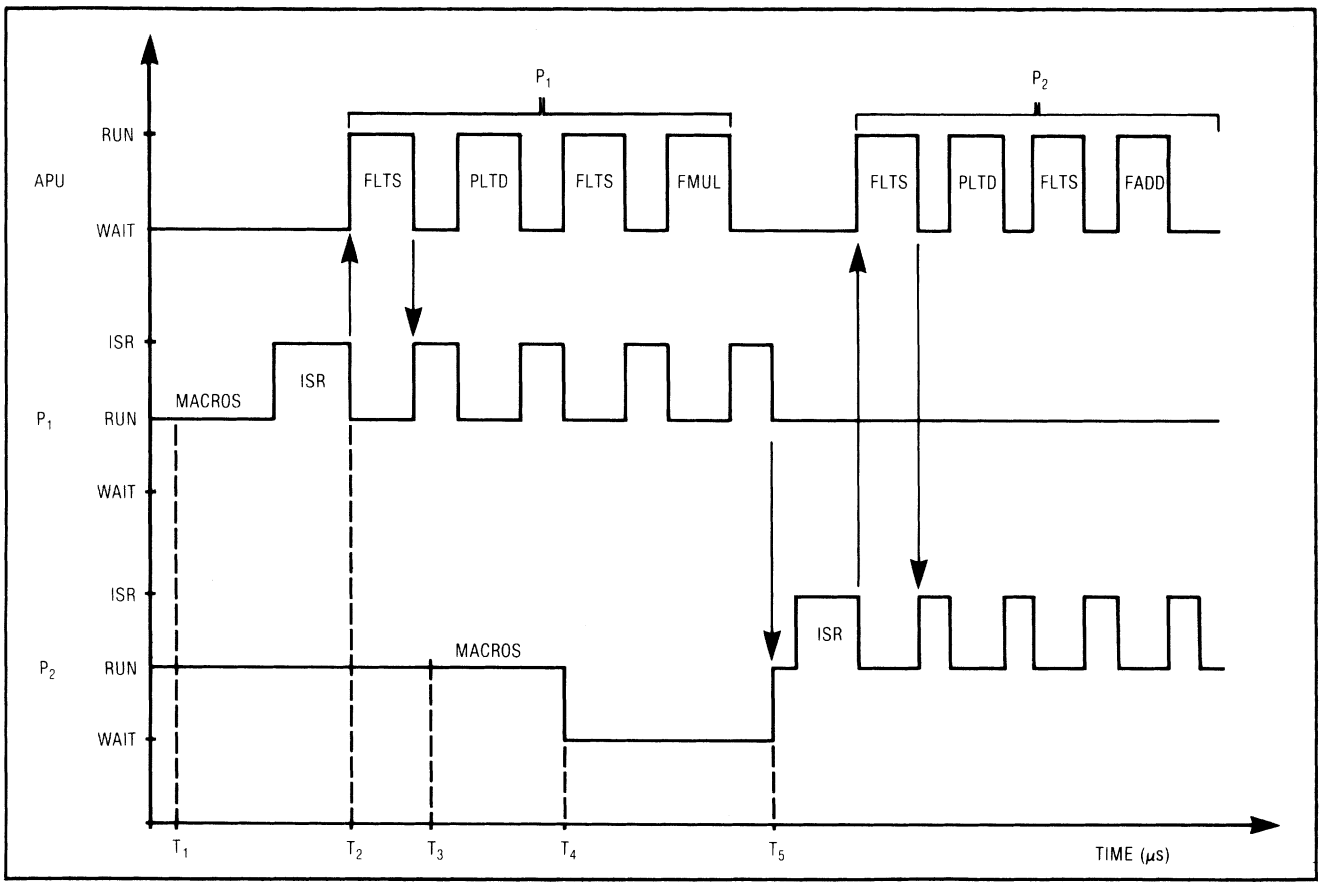


Figure 10. Timing diagram—MMP system.

We have implemented the same strategy in designing an APU software driver to a high-level language, or SDHLL. A set of the procedures has been designed that use the APU to perform the arithmetic operations, instead of doing this by software routines. From the user's point of view, the APU is invisible and the use of the SDHLL routines in a PL/M program is the same as the use of the FPAL routines.

In general, the following steps should be observed to use the SDHLL:

- An area of memory must be reserved for the floating-point record.
- The names of the SDHLL procedures the user plans to use must be declared to be "external" using the EXTERNAL attribute in PL/M-80.
- The SDHLL procedure references must be embedded in the user's source code where appropriate.
- The SDHLL procedure used by user's program should be linked to the user's object file.

Example 4. The following PL/M example computes the function

$$F = A*B+C$$

where A, B, and C represent addresses of the floating-point numbers. F is the address where the result is to be stored, and FPR is the address of the floating-point register.

We must first declare the SDHLL procedures used to be external and reserve the FPR memory as an array. The PL/M program is given below:

```
/* EXAMPLE 4 - F = A * B + C */
/* ------------------------------------- */
/* DEFINE EXTERNAL PROCEDURES */
FSET: PROCEDURE (FA,0P1,0P2) EXTERNAL;
      DECLARE (FA,0P1,0P2) ADDRESS;
END FSET;
FADD: PROCEDURE (FA,0A) EXTERNAL;
      DECLARE (FA,0A) ADDRESS;
END FADD;
FMUL: PROCEDURE (FA,0A) EXTERNAL;
      DECLARE (FA,0A) ADDRESS;
END FMUL;
FLOAD: PROCEDURE (FA,0A) EXTERNAL;
       DECLARE (FA,0A) ADDRESS;
END FLOAD;
FSTOR: PROCEDURE (FA,0A) EXTERNAL;
       DECLARE (FA,0A) ADDRESS;
END FSTOR;
/* DECLARE BYTE ARRAYS */
DECLARE FPR(18) BYTE,
            A(4)    BYTE,
            B(4)    BYTE,
            C(4)    BYTE,
            F(4)    BYTE;
```

Table 3.
Execution times: SDHLL vs. FPAL.

PROCEDURE	AVERAGE EXECUTION TIME (MICROSECONDS) FPAL	SDHLL
FADD	700	30
FSUB	800	35
FMUL	1500	50
FDIV	3700	57

```
/* COMPUTATION OF F = A*B + C*/
/* FSET SHOULD BE CALLED FIRST */
        CALL FSET(.FPR,0,0);      /*Initialization of FPR*/
        CALL FLOAD (.FPR,.A);     /*Load data A*/
        CALL FMUL (.FPR,.B);      /*Multiply A and B*/
        CALL FADD (.FPR,.C);      /*Compute A*B* + C*/
        CALL STOR (.FPR,.F);      /*Store result in F*/
```

The SDPPM driver can be modified in order to be consistent to the existing FPAL library package. A detailed description of the SDHLL is given by Furht and Milutinovic.[7]

The existing PL/M programs, which use the FPAL routines, can implement the SDHLL routines without any change. In general, the only change occurs during the linking process. When the user has completed program development, he should link the SDHLL, instead of the FPAL, to his object modules.

It is obvious that execution speeds of SDHLL procedures are much faster than those of the corresponding FPAL procedures. Table 3 gives a brief comparison of the execution speeds of some commonly used procedures. The microprocessor clock frequency is supposed to be 6MHz and the APU clock frequency is 3MHz.

These software drivers enable easy and efficient APU implementation both in single microprocessor systems and in a multimicroprocessor system with a common APU. Two different interface techniques have been used: pseudopolling and interrupt techniques. The corresponding APU drivers have been developed—the SDPPM and the SDIM, respectively. The SDPPM can be modified for use in a high-level language environment. Using the APU driver, the user can focus attention on the problem by simply calling a set of macroinstructions or "CALL procedures" to activate the APU functions.

The APU software driver could be included in a compiler for a language such as PL/M, Pascal, or Ada. ■

Appendix: The Interrupt service routine of the SDIM

```
; ***********************************************
; INTERRUPT SERVICE ROUTINE FOR SDIM
  -RST 7.5
; ***********************************************
; First activation: CALL instruction from the main
;                   program at the end of an APU
;                   sequence.
; Next activation: Interrupt signal END whenever
;                  a single APU operation has been
;                  completed.
; Function:        Send a new command/data to
;                  the APU using the top of
;                  command/pointer buffers.
; APU addresses: 16H - Enter/read byte into/from
;                  the APU.
;                17H - Enter command/ read
;                  status

ISR:    PUSH    PSW         ; Save registers
        PUSH    B
        PUSH    D
        PUSH    H
ISRO:   LDA     CBUF        ; Load next
                              command from
                              the CB
        MOV     B,A         ; Save command
                              in B
        LXI     H,CBUF      ; Pointer to the
                              top of the CB
        LXI     D,CBUF+1    ; Pointer to next
                              address
ISR1:   LDAX    D           ; Shift codes in
                              the CB . . .
        MOV     M,A
        ORA     A
        JZ      ISR2
        INX     H
        INX     D
        JMP     ISR1        ; . . . to the top
                              until 0 code is
                              found
ISR2:   MOV     A,B         ; Restore current
                              command code
        ANI     0F0H        ; Mask most
                              significant 4
                              bits
        CPI     40H
        JZ      ISR3        ; Test if data
                              entry command
        CPI     50H
        JZ      ISR6        ; Test if data
                              removal
                              command
; Command entry to the APU
        MOV     A,B         ; Restore
                              command code
        OUT     17H         ; Send to the
                              APU
        EI                  ; Enable
                              interrupt
        JMP     ISR9        ; Jump to the
                              end
; Data entry to the APU
ISR3:   LHLD    PBUF        ; Address of the
                              first byte
        MOV     A,M         ; Send 1. byte
        OUT     16H
        INX     H           ; Address of the
                              second byte
        MOV     A,M         ; Send 2. byte
```

```
        OUT     16H
        MOV     A,B         ; Restore
                              command code
        CPI     41H
        JNZ     ISR5        ; If 16-bit data
        INX     H           ; Address of the
                              third byte
        MOV     A,M         ; Send 3. byte
        OUT     16H
        INX     H           ; Address of the
                              third byte
        MOV     A,M         ; Send 3. byte
        OUT     16H
ISR5:   LXI     B,CBUF      ; Shift pointers
                              in the PB . . .
        LXI     D,CBUF+2    ; . . . to the top
                              until 0 pointer
                              is found
ISR5A:  LDAX    D
        STAX    B
        MOV     L,A         ; LSB of the
                              address being
                              moved
        INX     D
        INX     B
        LDAX    D
        STAX    B
        INX     D
        INX     B
        ORA     L           ; MSB address is
                              currently in A
        JNZ     ISR5A
        JMP     ISR0        ; Jump to "Load
                              next
                              command"
```

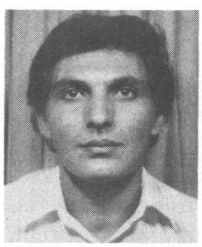
Data removal from the APU stack in reverse order

```
ISR6:   LHLD    PBUF        ; Address of the
                              result in HL
        MOV     A,B         ; Restore
                              command word
        CPI     51H
        JZ      ISR7        ; If 32-bit result
        INX     H           ; End of the
                              16-bit data
                              buffer
        IN      16H
        MOV     M,A         ; Store 1. byte
        DCX     H
        IN      16H
        MOV     M,A         ; Store 2. byte
        JMP     ISR8
ISR7:   INX     H
        INX     H
        INX     H           ; End of 32-bit
                              data buffer
        IN      16H
        MOV     M,A         ; Store 1. byte
        DCX     H
        IN      16H
        MOV     M,A         ; Store 2. byte
        DCX     H
        IN      16H
        MOV     M,A         ; Store 3. byte
        DCX     H
        IN      16H
        MOV     M,A         ; Store 4. byte
ISR8:   MVI     A,I         : Set flag "End
                              of APU
                              operation"
        STA     PBUF
```

```
ISR9:   POP     H           ; Restore
                              registers
        POP     D
        POP     B
        POP     PSW
        RET
```

References

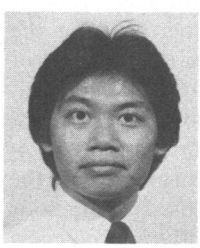
1. "Am9511 Arithmetic Processor Unit," data sheet, Advanced Micro Devices, Inc., Sunnyvale, CA, Mar. 1978.
2. B. K. Gupta, "Arithmetic Processor Chips Enhance Microprocessor Performance," *Computer Design,* Vol. 19, No. 7, July 1980, pp. 85-94.
3. E. B. Croson, F. H. Carlin, and J. A. Howard, "Integrated Arithmetic Processing Unit Enhances Processor Execution Times," *Computer Design,* Vol. 20, No. 4, Apr. 1981, pp. 163-171.
4. G. Adams and T. Rolander, "Design Motivations for Multiple Processor Microcomputer Systems," *Computer Design,* Vol. 17, No. 3, Mar. 1978.
5. R. O. Parker and J. H. Kroeger, "Algorithm Details for the Am9511 Arithmetic Processing Unit," Pub. No. AM-PUB072, Advanced Micro Devices, Inc., Sunnyvale, CA, 1978.
6. *8085/8085 Floating-Point Arithmetic Library User's Manual,* Pub. No. 9800452B, Intel Corp., Santa Clara, CA, 1977.
7. B. Furht and V. Milutinovic, "The Am9511 Arithmetic Processor Implementation in a High-Level Language Environment," in progress.



Borivoje Furht is an associate professor in the Electrical and Computer Engineering Department of the University of Miami, Coral Gables. From 1970 to 1981, he was with the Computer System Division of the Institute "Boris Kidric" —Vinca, Yugoslavia, where he was a project leader in designing specialized microprocessor-based systems and their application in telecommunication and control. Since 1982, he has been an assistant professor of electrical and computer engineering at the University of Miami. He has done research in computer architecture, multi-microprocessor systems, supercomputers, and software engineering. He is currently designing a super microcomputer used in image and signal processing.

Furht is the author of more than 40 technical papers and articles, about 30 technical reports, and two books. He is a member of IEEE and its Computer Society and a member of the Editorial Board of the *International Journal of Mini and Microcomputers.* He received the BSc, MSc, and PhD degrees in electrical and computer engineering from the University of Belgrade, Yugoslavia, in 1970, 1973, and 1978, respectively.

Furht's address is Department of Electrical and Computer Engineering, University of Miami, P.O. Box 248294, Coral Gables, Florida 33124.



Peter Lee is a member of the technical staff at IBM in Boca Raton, Florida, involved in design and development of advanced computers. Earlier he was on the technical staff at Gould, Inc, SEL. He received his BSEE in 1980 and MSEE in 1982 from the University of Miami.

Lee's address is IBM, Computer Systems Division, 2000 51st Street, Boca Raton, Florida 33431.