DECODING THE 8080 INSTRUCTIONS

8080 HAS 1-, 2- OR 3-WORD INSTRUCTIONS. IT IS CLEAR HOW LONG THE INSTRUCTION IS BY JUST EXAMINING THE FIRST BYTE.

FIRST DIVIDE THE OPCODE INTO 8 BITS AS FOLLOWS:

XX YYY ZZZ

LET XX BE THE PAGE NUMBERS: 00, 01, 10 AND 11. YYY AND ZZZ ARE 2 OCTAL DIGITS RANGING FROM 000 TO 111.

TO UNDERSTAND THE OPCODES, DEFINE RRR AS THE REGISTER ADDRESS AS FOLLOWS:

RRR=	000	B
	001	C
	010	D
	011	E
	100	H
	101	L
	110	-NOT USED (FLAG REGISTER EXCLUDED)
	111	A (THE ACCUMULATOR)

DEFINE DD AS THE ADDRESSES OF THE REGISTER PAIRS, SET #1 AS FOLLOW:

DD=	00	PAIR BC
	01	PAIR DE
	10	PAIR HL
	11	SP (THE STACK POINTER)

DEFINE QQ AS THE ADDRESSES OF THE REGISTER PAIRS, SET #2 AS FOLLOWS:

QQ=	00	PAIR BC
	01	PAIR DE
	10	PAIR HL
	11	PAIR AF (REFERRED TO AS PSW)

DEFINE THE CONDITIONS CCC USED IN THE JUMP, CALL AND RETURN INSTRUCTIONS AS FOLLOWS:

CCC=	000	NZ
	001	Z
	010	NC
	011	C
	100	PO
	101	PE
	110	P
	111	M

WITH THE ABOVE ADDRESSES AND CONDITIONS DEFINED, THE 8080 INSTRUCTIONS CAN BE DECODED AS SHOWN BELOW.

PAGE XX=00

-IN THIS PAGE, THE INSTRUCTIONS ARE CLASSIFIED USING THE OCTAL DIGIT ZZZ.

	YYY	ZZZ	OPCODE	INSTRUCTION NAME
(A)	000	000	00 000 000	NOP
(B)	DD0	001	00 DD0 001	LXI REGISTER PAIR (BC,DE,HL,SP)
	DD1	001	00 DD1 001	DAD REGISTER PAIR (BC,DE,HL,SP)
(C)	000	010	00 000 010	STAX B
	010	010	00 010 010	STAX D
	100	010	00 100 010	SHLD
	110	010	00 110 010	STA
	001	010	00 001 010	LDAX B
	011	010	00 011 010	LDAX D
	101	010	00 101 010	LHLD
	111	010	00 111 010	LDA
(D)	DD0	011	00 DD0 011	INX REGISTER PAIR(BC,DE,HL,SP)
	DD1	011	00 DD1 011	DCX REGISTER PAIR(BC,DE,HL,SP)
(E)	RRR	100	00 RRR 100	INC REGISTER(B,C,D,E,H,L,A)
(F)	RRR	101	00 RRR 101	DCR REGISTER(B,C,D,E,H,L,A)
(G)	RRR	110	00 RRR 110	MVI REGISTER(B,C,D,E,H,L,A)
(H)	000	111	00 000 111	RLC
	001	111	00 001 111	RRC
	010	111	00 010 111	RAL
	011	111	00 011 111	RAR
	100	111	00 100 111	DAD
	101	111	00 101 111	CMA
	110	111	00 110 111	STC
	111	111	00 111 111	CMC

PAGE XX=01

THIS PAGE IS USED UP BY THE MOV R1,R2 INSTRUCTION AND THE HALT INSTRUCTION.

	YYY	ZZZ	OPCODE	INSTRUCTION NAME
(A)	RRR	RRR'	01 RRR RRR'	MOV R,R'
(B)	110	110	01 110 110	HALT (SINCE REGISTER ADDRESS 110 IS NOT USED)

PAGE XX=10

THIS PAGE IS ISED UP BY 8 ARITHMATIC-LOGIC INSTRUCTIONS OPERATING ON REGISTER ADDRESS RRR. NOTE THE REGISTER ADDRESS 110 IS NOT USED.

YYY	ZZZ	OPCODE	INSTRUCTION NAME
000	RRR	10 000 RRR	ADD R
001	RRR	10 001 RRR	ADC R
010	RRR	10 010 RRR	SUB R
011	RRR	10 011 RRR	SBB R
100	RRR	10 100 RRR	ANA R
101	RRR	10 101 RRR	XRA R
110	RRR	10 110 RRR	ORA R
111	RRR	10 111 RRR	CMP R

PAGE XX=11

IN THIS PAGE, THE INSTRUCTIONS ARE CLASSIFIED USING THE OCTAL DIGIT ZZZ.

	YYY	ZZZ	OPCODE	INSTRUCTION NAME
(A)	CCC	000	11 CCC 000	CONDITIONAL RETURN INSTRUCTIONS (RNZ,RZ,RNC,RC,RPO,RPE,RP,RM)
(B)	QQO	001	11 QQO 001	POP REGISTER PAIR(BC,DE,HL,PSW)
	001	001	11 001 001	RET
	011	001	11 011 001	NOT USED
	101	001	11 101 001	PCHL
	111	001	11 111 001	SPHL
(C)	CCC	010	11 CCC 010	CONDITIONAL JUMP INSTRUCTIONS (JNZ,JZ,JNC,JC,JPO,JPE,JP,JM)
(D)	000	011	11 000 011	JMP
	001	011	11 001 011	NOT USED
	010	011	11 010 011	OUT
	011	011	11 011 011	IN
	100	011	11 100 011	XTHL
	101	011	11 101 011	XCHG
	110	011	11 110 011	DI
	111	011	11 111 011	EI
(E)	CCC	100	11 CCC 100	CONDITIONAL CALL INSTRUCTIONS (CNZ,CZ,CNC,CC,CPO,CPE,CP,CM)
(F)	QQO	101	11 QQO 101	PUSH REGISTER PAIR(BC,DE,HL,PSW)
	001	101	11 001 101	CALL
	011	101	11 011 101	NOT USED
	101	101	11 101 101	NOT USED
	111	101	11 111 101	NOT USED
(G)	PPP	111	11 PPP 111	RST P (THE RESTART INTRUCTION)

BUILDING A COMPUTER

To design a computer, considerations must be given to the following:

(1) WORDLENGTH: As stated in the Bible, John 1:1 "In the beginning was the WORD, ..." Therefore, to create a computer, the size of the WORD must be defined. It has an impact on arithmatic accuracy as well as the design of the memory bank.

(2) ADDRESS LENGTH: The number of bits to be used for addresses. It has an impact on the maxiumum amount of memory that the CPU can handle. The INTEL 8088 has 20-bit addresses and that's why the IBM-PC cannot handle more than 1 megabytes of memory. (MS-DOS reduced that even further to 640Kbytes, a software design limitation. In fact, thats why the new COMPAC 80386 machine cannot do much even with a great 32-bit CPU).

(3) The INSTRUCTION SET. The designer must choose a set of machine instructions which will make writing software an easy task. The number of instructions to be supported by the machine is limited mostly by hardware cost but sometimes by the wordlength.

(4) The INTERNAL ARCHITECTURE. The design decision to be made for the internal architecture is the number of DATA, ADDRESS and INDEX registers to be used, the speed of the clock and the decoding scheme for the isntructions. This has a strong impact on system programming, e.g., writing of operating system software, compilers and general purpose Assembler Language programming.

DESIGNING A LOUSY COMPUTER FROM SCRATCH

The following computer is an example given in Tannenbaum's book. it is not nearly powerful enough for general computational purposes, but it will explain how one can be built from scratch. It is called the "STACK MACHINE" in the book. See Appendix "MP" for details.

(1) WORDLENGTH: The wordlength of this stack machine is 16 bits. It will be used primarily for 16-bit 2's complement arithmatics. Note: for this machine, the data will be fetched 16 bits at a time.

(2) ADDRESS LENGTH: 13-bit addresses will be used in this machine. 3 bits are used for the OPCODE (Operation Code) of the instruction set, so 13 bits are left for addressing the main memory.

(3) The INSTRUCTION SET: Normally, a 3-bit OPCODE will give you only 8 unique instructions because 2**3=8. But since this machine operates on a stack, the arithmatic instructions ADD, SUB, MUL and DIV do not have to specify a memory address because the operands of these instructions are all located on the stack already. The address field (the 13 bits normally reserved for the memory address) of these arithmatic instructions can therefore be used for OPCODEs as well. The instruction OPCODEs are defined as follows:

PUSH <memory>	=	000 mmmmmmmmmmmmm
POP <memory>	=	001 mmmmmmmmmmmmm
JUMP <memory>	=	010 mmmmmmmmmmmmm
JNEG <memory>	=	011 mmmmmmmmmmmmm
JZER <memory>	=	100 mmmmmmmmmmmmm
JPOS <memory>	=	101 mmmmmmmmmmmmm
CALL <memory>	=	110 mmmmmmmmmmmmm
ADD	=	111 0000000000000
SUB	=	111 0000000000001
MUL	=	111 0000000000010
DIV	=	111 0000000000011
RETURN	=	111 0000000000100

Note: The instructions ADD, SUB, MUL, DIV and RETURN all have 111 in the 3-bit OPCODE field. Further decoding is done using the remaining 13 bits. In fact, there are plenty of room for expanding the instruction set. In the above instructions, mmmmmmmmmmmmm represents a 13-bit memory address. For the PUSH and the POP instructions, <memory> is the memory address to fetch and store the data, respectively. For the unconditional jump instruction "JUMP" and the conditional Jump instructions "JNEG", "JZER" and "JPOS", <memory> is the memory location where the next program instruction resides if the condition is true.

(4) The INTERNAL ARCHITECTURE:

(a) The stack machine has 4 internal registers A, B, C and D, but the user cannot access them directly. They are used by the machine internally for temporary results. (This is similar to the HP calculator where the y, z and t registers cannot be accessed directly.)

(b) The Program Counter "PC" is a 13-bit register which "COUNTS" the instructions addresses, i.e., the address of the program instructions are counted sequentially, one line at a time, unless a jump, call or return instruction is executed to change the content of the PC. Since the "PC" holds nothing but addresses, it has a length of 13 bits.

(c) The Stack Pointer "SP". "SP" is a 13-bit register which holds the address of the TOP of stack. Lets say the stack bottom is located at 6000 and there are 5 items on it, then the stack would look like

address 6000	<16-bit number>
address 6001	<16-bit number>
address 6002	<16-bit number>
address 6003	<16-bit number>
address 6004	<16-bit number>

At the moment, SP has the value of 6004 (in base 10) or in binary 1011101110100. If a PUSH 5143 instruction is performed, then the stack pointer SP will be incremented such that SP is replaced by SP+1=6005 and the number presently stored in memory location m(5143) is transferred to memory location m(6005). The stack will now look like

address 6000	<16-bit number>
address 6001	<16-bit number>
address 6002	<16-bit number>
address 6003	<16-bit number>
address 6004	<16-bit number>
address 6005	<16-bit number fetched from m(5143)>

If a POP 4597 instruction is now executed, the number on the stack top, i.e., m(6005) will be copied to m(4597), the address 4597 was specified in the instruction. Then the stack pointer SP will be decremented from 6005 to 6004, making the stack top now at 6004 again. Hence, the two instructions PUSH 5143 and POP 4597 effectively transferred the number from m(5143) to m(4597). In this machine, there is no way to go directly from one memory to another without going through the stack. For most microprocessors, the data transfer between two memory locations has to go through an internal register.

If an ADD instruction is executed, the two numbers at the stack-top will be popped and added, i.e., m(6004)+m(6003). After the addition, the value of SP is 6002 because two items were popped. But the result will be pushed back on stack in location m(6003). The language to be used to program this STACK MACHINE follows a "postfix" notation, i.e., the operator comes behind the operands. This notation was made famous by the success of the HP calculators (it is also called Reversed Polish Notation). The new graphic language "POSTSRIPT" is also written completely in postfix notation. So is the language FORTH. The DOC likes it very much.

(d) The Memory Address Register MAR. Since the CPU has hardly any memory to work with, it must go to the main memory constantly to get either data or program instructions. To get data, the CPU must tell the memory controller the address of the data, e.g., PUSH 5143 will tel the memory it wants the data code from location 5143. To get the program instruction, the CPU tells the main memory it wants m(PC), the memory pointed to by the program counter. In either case, the address must be put onto the address bus which connects the CPU and the main memory. On the CPU side, the address bus is hooked up to the Memory Address Register MAR. Hence, whenever the CPU wants to get something from or send something to the main memory, it must put the address in the MAR.

(e) The Memory Buffer Register MBR (sometimes called MDR for Memory Data Register). After request for READ or WRITE has been sent to the main memory, the data to be received from the main memory or the data to be transferred to the main memory must be stored temporarily in the MBR. Since the data are 16 bits long, MBR is also 16 bits.

(f) The Instruction Register IR. When the program instruction comes in from the main memory, it will be stored in the instruction register IR for decoding. After the CPU decodes the instruction, it will open or close the electronic gates which control the internal registers to perform the functions. The sequence of gate control functions is called a microprogram and it is stored in a ROM inside the CPU.

(g) The constant ROM's. There are some constants which are used frequently inside a CPU, e.g., -1, 0, 1, etc. The number 15 is also used a lot in this STACK MACHINE because all the multiply and divide instruction will need to shift and add/subtract 16 times, hence, the loop count is always initialized to 15 then count to 0 for these instructions. In some other processors, like the 8087 numeric coprocessor, constants like pi=3.14159, e=2.71828, 10., 100., 2., ..., etc., are stored in ROMs.

The following examples are assembler language programs for the STACK machine described above. They illustrate clearly how an assembler convert word instructions into machine codes as a one-to-one process. But the part which interest us the most, at this stage, is how the microprogram for this STACK MACHINE can be developed.

TWO-PASS ASSEMBLERS

Most Assemblers are two-pass Assemblers, i.e., they need to read the Assembler Language program twice before the final binary machine codes are generated. During the first pass, the symbol table is created. In this table, all the labels in the first column of the Assembler Language program will be given a unique memory address. This can be done simply by counting the number of instruction words from the beginning of the program using the ILC (Instruction Location Counter). In Example Number One, the ORG 00100 statement is called a pseudo-op because it does not generate an executable instruction, but it is used to help the Assembler setup the code. In this case, "ORG 00100" sets the ILC to 00100 octal. The symbol table for this example is therefore: T1=00100, T2=00101, T3=00102, SUM=00103, BEGIN=00104 and END=00112.

During the second pass, the instructions which have a label name can have the label replaced by a binary address. For example, the POP SUM instruction has an OPCODE of $\underline{001}$ (see the previous pages) and the 13-bit binary address for SUM is 0 000 001 $\overline{000}$ 011 =00103 octal. In example number 2, the JUMP LOOP instruction has an OPCODE of $\underline{010}$ and the memory location to jump to is LOOP=00274 octal or 0 000 010 $\overline{111}$ 100 binary. Therefore, after the second pass, all symbolic names used in the programs are completely replaced; everything will be in machine codes.

EXAMPLE NUMBER ONE:

(ASSUME OPERATING SYSTEM ADDRESS STARTS AT OPSYS=00000)

```
                                                   ORG 00100
00100   000 0 000 000 000 100           T1:        4
00101   000 0 000 000 000 101           T2:        5
00102   000 0 000 000 000 010           T3:        2
00103   000 0 000 000 000 000           SUM:       DW

00104   000 0 000 001 000 000           BEGIN:     PUSH T1
00105   000 0 000 001 000 001                      PUSH T2
00106   111 0 000 000 000 000                      ADD
00107   000 0 000 001 000 010                      PUSH T3
00110   111 0 000 000 000 000                      ADD
00111   001 0 000 001 000 011                      POP SUM
00112   010 0 000 000 000 000?          END:       JUMP OPSYS
```

EXAMPLE NUMBER TWO:

```
                                                   ORG 00264
00264   000 0 000 000 000 000           FACTOR:    DW
00265   000 0 000 000 000 101           NUMBER:    5
00266   000 0 000 000 000 000           TERM:      DW
00267   000 0 000 000 000 001           ONE:       1

00270   000 0 000 010 110 111                      PUSH ONE
00271   001 0 000 010 110 100                      POP FACTOR
00272   000 0 000 010 110 101                      PUSH NUMBER
00273   001 0 000 010 110 110                      POP TERM
00274   000 0 000 010 110 100           LOOP:      PUSH FACTOR
00275   000 0 000 010 110 110                      PUSH TERM
00276   111 0 000 000 000 010                      MUL
00277   001 0 000 010 110 100                      POP FACTOR
00300   000 0 000 010 110 110                      PUSH TERM
00301   000 0 000 010 110 111                      PUSH ONE
00302   111 0 000 000 000 001                      SUB
00303   001 0 000 010 110 110                      POP TERM
00304   000 0 000 010 110 110                      PUSH TERM
00305   100 0 000 011 000 111                      JZER END
00306   010 0 000 010 111 100                      JUMP LOOP
00307   010 0 000 000 000 000?          END:       JUMP OPSYS
```

EXAMPLE NUMBER THREE:

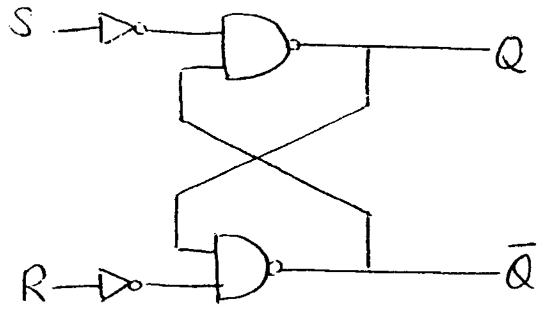
```
                                                 ORG 00300
00300   000 0 000 000 000 000           SUM:     DW
00301   000 0 000 000 000 000           ARG1:    DW
00302   000 0 000 000 000 000           TERM:    DW
00303   000 0 000 000 000 000           FACTOR:  DW
00304   000 0 000 000 000 001           ONE:     1
00305   000 0 000 000 000 110           NUM1:    6
00306   000 0 000 000 000 011           NUM2:    3
00307   000 0 000 011 000 001           F:       PUSH ARG1
00310   001 0 000 011 000 010                    POP TERM
00311   000 0 000 011 000 100                    PUSH ONE
00312   001 0 000 011 000 011                    POP FACTOR
00313   000 0 000 011 000 011           LOOP:    PUSH FACTOR
00314   000 0 000 011 000 010                    PUSH TERM
00315   111 0 000 000 000 010                    MUL
00316   001 0 000 011 000 011                    POP FACTOR
00317   000 0 000 011 000 010                    PUSH TERM
00320   000 0 000 011 000 100                    PUSH ONE
00321   111 0 000 000 000 001                    SUB
00322   001 0 000 011 000 010                    POP TERM
00323   000 0 000 011 000 010                    PUSH TERM
00324   100 0 000 011 010 110                    JZER R
00325   010 0 000 011 001 011                    JUMP LOOP
00326   111 0 000 000 000 100           R:       RETURN

00327   000 0 000 011 000 101           MAIN:    PUSH NUM1
00330   001 0 000 011 000 001                    POP ARG1
00331   110 0 000 011 000 111                    CALL F
00332   000 0 000 011 000 011                    PUSH FACTOR
00333   001 0 000 011 000 000                    POP SUM
00334   000 0 000 011 000 110                    PUSH NUM2
00335   001 0 000 011 000 001                    POP ARG1
00336   110 0 000 011 000 111                    CALL F
00337   000 0 000 011 000 011                    PUSH FACTOR
00340   000 0 000 011 000 000                    PUSH SUM
00341   111 0 000 000 000 000                    ADD
00342   001 0 000 011 000 000                    POP SUM
00343   010 0 000 000 000 000?          END:     JUMP OPSYS
```

CLOCKED-RS-FLIP-FLOP

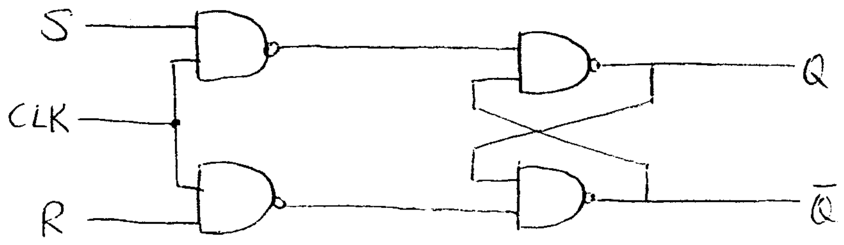
RESET - SET

Consider the RS-flip-flop as shown in the figure, its truth table has the form:



S	R	Q0	Q1	
0	0	0	0	no change
0	0	1	1	
0	1	0	0	reset to 0
0	1	1	0	
1	0	0	1	set to 1
1	0	1	1	
1	1	0	?	race condition,
1	1	1	?	result uncertain

Since Q1=Q0 if [S=1; R=0], we can introduce a clock signal so that when the clock CLK=0, the inputs to S and R are 0. Hence, whatever Q is, it will remain the same during the time that CLK=0. The diagram for a clocked RS flip-flop is



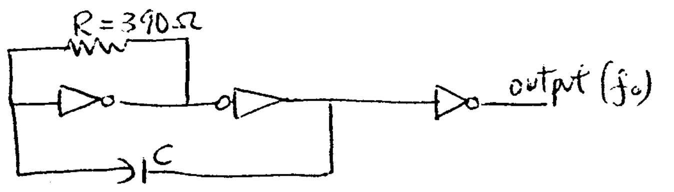
When CLK=1, the function [illegible] clocked RS flip-flop becomes the sam as that of the regular RS-flip-flop.

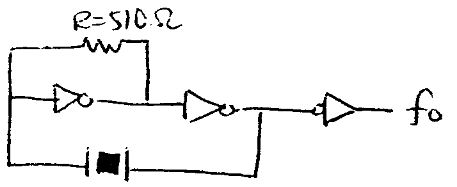
The function of the clock is used here to synchronize all the components within the computer so that the faster components will wait for the slower ones. If all the components are fast, then it is wise to increase the clock rate to improve the performance of the computer as a whole.

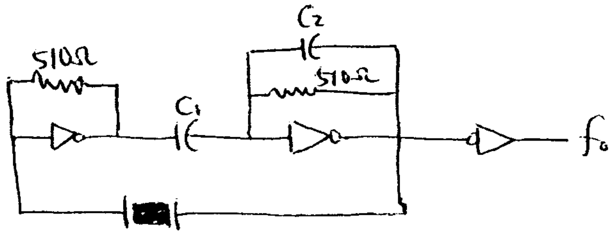
CLOCK CIRCUITS

In physics, we know of the RLC circuit. It oscillates with a sinusoidal frequency which is its resonance frequency. But since the components of resistors, capacitors and inductors are all sensitive to the environment, the frequency of the circuit cannot be controlled accurately.

The QUARTZ crystal is the central element of the modern electrical clock circuit. It provides electromechanical coupling to stabilize the frequency with incredible accuracy. As you may recall, the mechanical mass-spring-dashpot system also has a resonance frequency. The coupling of both these effects can increase the "quality" of the resonance peak as much as you like. Some circuits are shown below. The inverters serve to rectify the sinusoidal waves to rectangular waves.





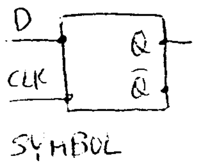


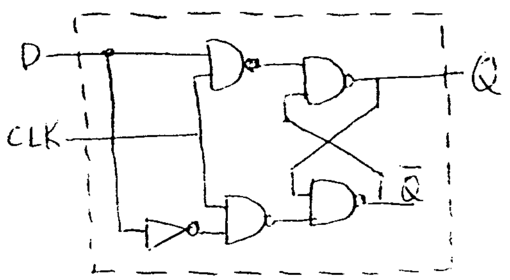
Some better clock circuits have more than one quartz crystals. In fact, many watches have "double quartz" accuracy, about 2 seconds gain/loss per year, fantastic!

In today's watches, nearly all of them have "Quartz Movement" inside even though they may look like "analog watches". The old watches goes"tic-toc" for every second, but the new ones goes "tic...tic...tic", with 1 tick per second. Thats because all quartz crystal clock circuits come in 2**12 Hz. By putting together 12 JK or T flip-flops in series, the clockrate of 2**12 Hz can be reduced to 1 Hz or 1 tick per second. Sad but true, most $30 watches have the same quartz movement as the $5000 ones. (Can't say the same about the $3.99 watch I bought for my daughter in San Francisco.)

D-FLIP-FLOP

Since the [illegible] flop has a "don't do" condition for [R=1; S=1] and the [R=0; S=0] condition is handled nicely by the clock, the remaining two conditions have R and S being opposite. So the D-flip-flop can be introduced by setting R and S opposite using an inverter in the following manner,



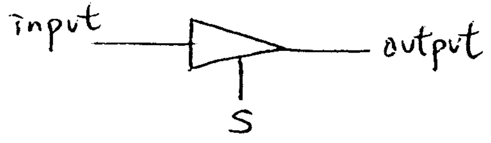


D	Q(t)	Q(t+T)
0	0	0
0	1	0
1	0	1
1	1	1

In the above truth table, Q(t) is the value "stored" in the D-flip-flop at time t. One clock cycle (T) later the value of Q(t+T) is set according to the input value D.

TRI-STATE GATES

The TRI-STATES gate had an important impact on the development of the computer. As its name suggests, the output line has 3 states: 0, 1 or disconnected.

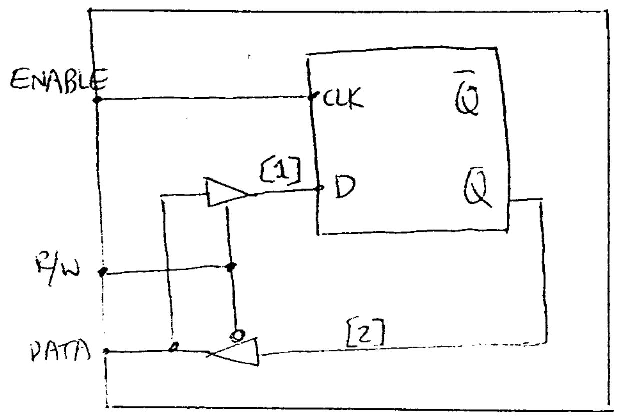


S	input	output
1	0	0
1	1	1
0	0 1	disconnected

When the selector line S=1, the gate behaves like an ordinary wire, i.e., output=input. If S=0, the gate goes into a "high impedance" state (approach infinite in resistance) so the input line is effectively disconnected from the output.

A READ/WRITE MEMORY CELL

As indicated before, RAM (Random Access Memory) should be called READ/WRITE memory because ROMs can also be randomly accessed. The following diagram shows how a D-flip-flop can be made into a R/W memory cell with 2 tri-state gates.



The R/W line controls whether the "data line" is input or output. (For READ, R/W=0, DATA is output. For WRITE, R/W=1, DATA is input).

If R/W=1, line [2] is disconnected from DATA, but line [1] is connected to DATA. So the data will enter the D-flip-flop as an input. This is a WRITE operation.

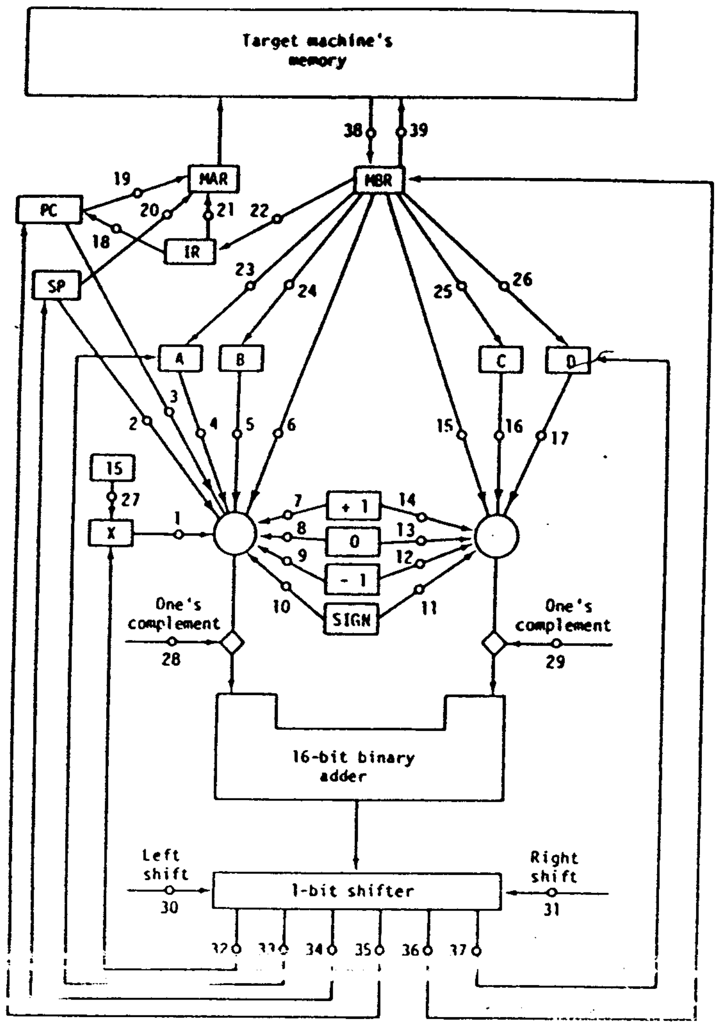
If R/W=0, line [2] is connected from Q to DATA bit line [1] is disconnected. Hence the value of the stored number in Q is output to the outside world through the DATA pin. This is a READ operation.

The ENABLE line is connected to the CLK signal of the D-flip-flop, so when ENABLE is 0, the element stored in Q cannot be changed by a WRITE operation. But the way the element is presently set up, it can be read at any time regardless of the state of the ENABLE line. Most of the commercial memory chips are disconnected completely from the DATA line when ENABLE=0, neither READ nor WRITE operation can be performed.

The ENABLE pin is needed on all the memory chips because the larger memory banks are organized into PAGES. Since only 1 page is accessed at any one time, the ENABLE pin can be used to turn all the other PAGES off. This way, they can all share the same Address and Data bus. The following is an example of how 64K-bit of memory can be constructed using 16K-bit memory chips. (The earlier Apple and CP/M machines use 16K-bit chips, it usually fills up an entire board.)

The Internal Controls of the STACK MACHINE

The figure to the right has been extracted from page MP3 in the Appendix "MP". It shows 39 different sets of TRI-STATE gates which control the internal operation of the STACK MACHINE. The arrow with a tiny circle, e.g., —o→, represents a one-directional bus for data or addresses, i.e., READ —o→ WRITE. The circle indicates a valve which may connect or disconnect the bus. The width of the buses are 13-bit wide for addresses, e.g., bus# 18, 19, 20, etc, and 16-bit wide for data, e.g., bus# 4, 5, 16 and 17, etc.



The instructions such as PUSH, POP or ADD can be accomplished by setting off appropriate gate sequences, perfectly timed. The controller which generates these gate sequences, is also a computer, but it has only 2 instructions: a test instruction and an execute instruction.

An execute instruction is very simple, it has a "1" in bit position 0 to distinguish it from the test instruction. Hence, bit #0 serves as the "OPCODE" of the two instructions, either a "0" or a "1" (a very short OPCODE). The following executed instruction,

[illegible]

will open gates 18, 19 in a cycle. Gate 19 puts the value of the PC into MAR, then from the memory comes the next instruction in the MBR. The next step is to move the instruction from MBR to IR [illegible]

THE TEST INSTRUCTION

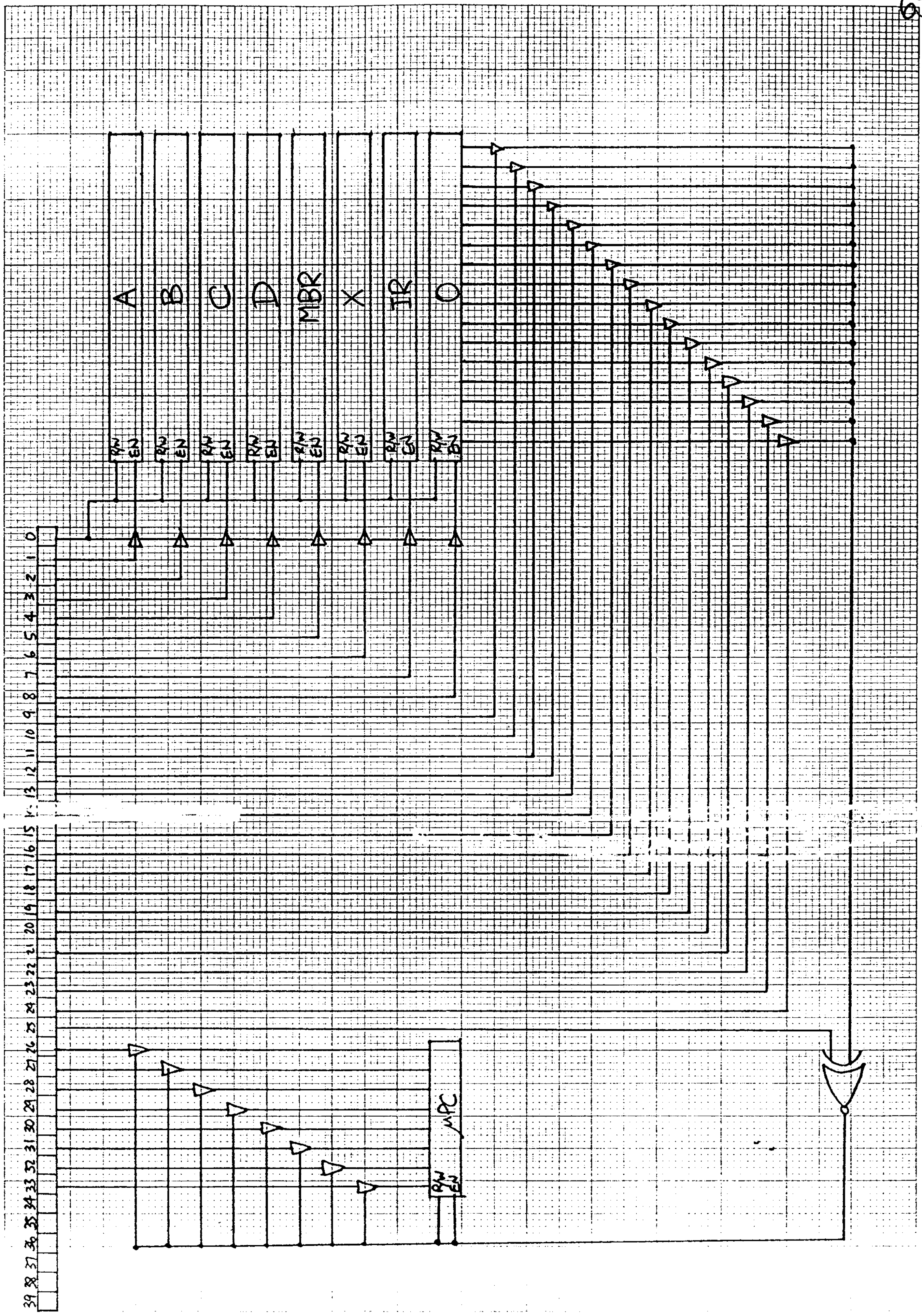
The TEST instruction has an opcode of "0" at bit position 0. The instruction format is

```
                 3333333333222222222211111111111
   BIT NO.       9876543210987654321098765432109876543210
                 ----------------------------------------
                 NNNNNNJJJJJJJcSSSSSSSSSSSSSSSS0IXMDCBA0
```

in which
- 0 = The 1-bit opcode field,
- A = Register A is selected if bit "1" is "ON",
- B = Register B is selected if bit "2" is "ON",
- C = Register C is selected if bit "3" is "ON",
- D = Register D is selected if bit "4" is "ON",
- M = The Memory Buffer Register MBR is selected if bit "5" is "ON",
- X = The counter register X is selected if bit "6" is "ON",
- I = The Instruction Register IR is selected if bit "7" is "ON",
- 0 = The "ZERO" register is selected if bit "8" is "ON",
- SSSSSSSSSSSSSSSS = A 16-bit field selecting which bit of data to test,
- c = The value of either 0 or 1 to be compared against,
- JJJJJJJJ = An 8-bit micro-address to jump to if the test is TRUE,

and
- NNNNNN = These 6 bits are currently unused, it can be used expansion.

A diagram showing how this TEST instrution can be implemented using basic electronic gates is shown in the following page. If bit-0 is "0", i.e., a TEST instruction, then the 8 registers: A,B,C,D,MBR,X,IR and 0 will receive a "0" or "READ" signal on the "R/W" line. Only one of the registers will be read, they are selected by bits 1 through 8. If only bit 4 is ON, register D will be the only one with the ENABLE on, thus only register D will be read from. There are also 16 bits in each register, we must select only 1 bit to test against "c", i.e., bit number 25. The selection of which bit to test is indicated by the SSSSSSSSSSSSSSSS field. For example, if SSSSSSSSSSSSSSSS = 0100000000000000, then bit 14 of the reg [illegible] be test. Note the bits are numbered from 0 to 15 with the rightmost bit being bit 0. As shown in the diagram for the TEST instruction, bit 9 is connected to a tri-state gate at test bit position 0 while bit 24 is connected to test bit position 15; mathematically, the offset is 9 because bit "J+9" is connected to test bit position "J". Only one of the 16 bits will be connected to the IOR gate located on the lower-left-hand corner of the graph. The other input for the IOR gate is "c" or bit 25 of the TEST instruction word, if "c" is equal to the test bit, then the output of the IOR gate will be 1, enabling the micro-PC to be written to. The "TEST IS TRUE" line will also connect bits 26 through 33 to the micro-PC and set the R/W line to "WRITE". Therefore, the micro-address that is stored in the TEST instruction will be copied into the micro-PC and the next micro-instruction to be executed should be fetched from this new location rather than the one immediately following the present instruction. If the test is false, then the micro-PC will retain its present value, i.e., 1 larger than the present instruction location.


A
B
C
D
MBR
X
IR
Q
R/W
EN
μPC
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39

THE COMPLETE MICROPROGRAM FOR THE STACK MACHINE

```
ADDRESS       STATEMENT

  0   MAINLOOP: MAR= PC; MBR= MEMORY (MAR); /* FETCH NEXT TARGET INSTR */
  1             IR= MBR; PC= PC + 1; /* MOVE INSTR TO IR AND ADVANCE PC */
  2             IF BIT(15,IR)= 1  THEN GO TO OP4567; /* DETERMINE TYPE */
  3             IF BIT(14,IR)= 1  THEN GO TO OP23;
  4             IF BIT(13,IR)= 1  THEN GO TO POP;

  5   PUSH:     MAR=IR; SP= SP + 1; MBR= MEMORY (MAR);
  6             MAR =SP; MEMORY (MAR) = MBR;
  7             GO TO MAINLOOP;

  8   POP:      MAR = SP;  SP = SP + (-1); MBR = MEMORY(MAR);
  9             MAR=IR; MEMORY(MAR) = MBR;  /* MAR=IR IS 13 BITS WIDE */
 10             GO TO MAINLOOP;

 11   OP23:     IF BIT(13,IR) = 1  THEN GO TO JNEG;

 12   JUMP:     PC = IR;       /* THIS DATA PATH IS 13 BITS WID  */
 13             GO TO MAINLOOP;

 14   JNEG:     MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
 15             IF BIT(15,MBR) = 1  THEN GO TO JUMP;
 16             GO TO MAINLOOP;

 17   OP4567:   IF BIT(14,IR) = 1  THEN GO TO OP67;
 18             IF BIT(13,IR) = 1  THEN GO TO JPOS;

 19   JZER:     X = 15;  MAR = SP;  SP = SP + (-1);   MBR = MEMORY(MAR);
 20   JLOOP:    IF BIT(15,MBR) = 1  THEN GO TO MAINLOOP;
 21             MBR = LEFT_SHIFT(MBR + 0);
 22             X = X + (-1);
 23             IF BIT(15,X) = 0  THEN GO TO JLOOP;
 24             GO TO JUMP;

 25   JPOS:     MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
 26             IF BIT(15,MBR) = 0  THEN GO TO JUMP;
 27             GO TO MAINLOOP;

 28   OP67:     IF BIT(13,IR) = 1  THEN GO TO OP7;

 29   CALL:     SP = SP + 1;
 30             MAR = SP;  MBR = PC + 0;  MEMORY(MAR) = MBR;
 31             GO TO JUMP;

 32   OP7:      IF BIT(2,IR) = 1  THEN GO TO RETURN;
 33             IF BIT(1,IR) = 1  THEN GO TO MULDIV;

 34   ADDSUB:   MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
 35             IF BIT(0,IR) = 0  THEN GO TO SUM;
 36             MBR = COM(MBR) + 1;    /* FOR SUBTRACTION ONLY
 37   SUM:      MAR = SP;  A = MBR;  MBR = MEMORY(MAR);
 38             MBR = A + MBR;  MEMORY(MAR) = MBR;
 39             GO TO MAINLOOP;

 40   MULDIV:   IF BIT(0,IR) = 1  THEN GO TO DIV;

 41   MUL:      MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
 42             C = MBR;  MAR = SP;  MBR = MEMORY(MAR);
 43             X = 15;  A = 0 + 0;  D = 0 + 0;
 44             IF BIT(15,MBR) = 0  THEN GO TO MULLOOP;
 45             MBR = COM(MBR) + 1;
 46             B = MBR;  MBR = 1 + COM(C);
 47             C = MBR;  MBR = 0 + B;
 48   MULLOOP:  IF BIT(0,MBR) = 0  THEN GO TO NOADD;
 49             A = A + C;
 50   NOADD:    MBR = RIGHT_SHIFT(MBR + 0);
 51             D = RIGHT_SHIFT(0 + D);
 52             IF BIT(0,A) = 0  THEN GO TO NOCARRY;
 53             D = SIGN + D;
 54   NOCARRY:  A = RIGHT_SHIFT(A + 0);
 55             IF BIT(15,C) = 0  THEN GO TO MULEND;
 56             IF BIT(14,A) = 0  THEN GO TO MULEND;
 57             A = A + SIGN;
 58   MULEND:   X = X + (-1);
 59             IF BIT(15,X) = 0  THEN GO TO MULLOOP;
 60             MBR = 0 + D;  MEMORY(MAR) = MBR;
 61             GO TO MAINLOOP;

 62   DIV:      MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
 63             C = MBR;  B = MBR;  MAR = SP;  A = 0 + 0;  MBR = MEMORY(MAR);
 64             D = MBR;  MBR = COM (MBR) + 1;  X = 15;
 65             IF BIT(15,MBR) = 1  THEN GO TO DVDPOS;
 66             D = MBR;  MBR = COM(B) + 1;
 67             B = MBR;
 68   DVDPOS:   MBR = 1 + COM(C);
 69             IF BIT(15,C) = 0  THEN GO TO DIVLOOP;
 70             C = MBR;  MBR = COM(MBR) + 1;
 71   DIVLOOP:  A = LEFT_SHIFT (A + 0);
 72             IF BIT(15,D) = 0  THEN GO TO NOCARRY2;
 73             A = A + 1;
 74   NOCARRY2: D = LEFT_SHIFT(0 + D);
 75             A = A + MBR;
 76             IF BIT(15,A) = 1  THEN GO TO DIVNEG;
 77             D = D + 1;
 78   DIVNEG:   IF BIT(15,A) = 0  THEN GO TO DIVPOS;
 79             A = A + C;
 80   DIVPOS:   X = X + (-1)
 81             IF BIT(15,X) = 0  THEN GO TO DIVLOOP;
 82             IF BIT(15,B) = 0  THEN GO TO DIVEND;
 83             D = 1 + COM(D);
 84   DIVEND:   MBR = 0 + D;  MEMORY(MAR) = MBR;
 85             GO TO MAINLOOP;

 86   RETURN:   MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
 87             IR = MBR;
 88             GO TO JUMP;
```

SYMBOL TABLE FOR MICRO-PROGRAM

NAME	DECIMAL ADDRESS	BINARY ADDRESS
PUSH	5	00000101
POP	8	00001000
OP23	11	00001011
JMUP	12	00001100
JNEG	14	00001110
OP4567	17	00010001
JZER	19	00010011
JLOOP	20	00010100
JPOS	25	00011001
OP67	28	00011100
CALL	29	00011101
OP7	32	00100000
ADDSUB	34	00100010
SUM	37	00100101
MULDIV	40	00101000
MUL	41	00101001
MULLOOP	48	00110000
NOADD	50	00110010
NOCARRY	54	00110110
MULEND	58	00111010
DIV	62	00111110
DVDPOS	68	01000100
DIVLOOP	71	01000111
NOCARRY2	74	01001010
DIVNEG	78	01001110
DIVPOS	80	01010000
DIVEND	84	01010100
RETURN	86	01010110

40-BIT MICROCODE IN HEXADECIMAL NOTATION

0	4000080001				
1	0800404009	31	0030000300	61	0000000300
2	0047000080	32	015A000880	62	4400101005
3	002E800080	33	00A2000480	63	4203102101
4	0022400080	34	4400101005	64	101C004041
5	4400204005	35	0094000280	65	0113000020
6	8000100001	36	1010004041	66	1014004021
7	0000000300	37	4000900001	67	0001000001
8	0400101005	38	9000008011	68	1020010081
9	8000200001	39	0000000300	69	011D000008
10	0000000300	40	00FA000280	70	1012004041
11	003A400080	41	4400101005	71	0240002011
12	0000040001	42	4002100001	72	0129000010
13	0000000300	43	2208002101	73	0200004011
14	4400101005	44	00C1000020	74	2040020101
15	0033000020	45	1010004041	75	0200008011
16	0000000300	46	1021010081	76	013B000002
17	0072800080	47	1002002021	77	2000020081
18	0066400080	48	00C8000220	78	0141000002
19	4408101005	49	0200010011	79	0200010011
20	0003000020	50	1080002041	80	0100001003
21	1040002041	51	2080020101	81	011D000040
22	0100001003	52	00D8000202	82	0151000004
23	0051000040	53	2000020401	83	2020020081
24	0030000300	54	0280002011	84	9000020101
25	4400101005	55	00E9000008	85	0000000300
26	0031000020	56	00E8800002	86	4400101005
27	0000000300	57	0200000811	87	0000400001
28	0082400080	58	0100001003	88	0030000300
29	0400004005	59	00C1000040		
30	9000102009	60	9000020101		

THE PROGRAM COUNTER AND THE MICRO-PROGRAM COUNTER

The Program Counter is used to fetch the user's program from the main memory while the Micro-Program Counter is used to fetch the microprogram from the CPU's internal Read-Only-Memory. These two counters are operated completely independently; usually they have their own clock for synchronization. The internal clock usually runs a lot faster, yielding several sub-clock-cycles for every external clock cycle.

The name "counter" is used because the instructions are fetched sequentially, e.g., location 5, location 6, location 7,..., etc., so most of time the PC is changed by counting. The only time that the PC is changed differently is when a JUMP, CALL or RETURN instruction is executed. The JUMP instruction is like a GO TO statement in FORTRAN or BASIC, it causes a branch in the program to a different location, so the PC must be set to a new value specified by the JUMP instruction. For example, in the FORTRAN statement "GO TO 200", the label "200" is an identifier for one particular instruction in the memory, so when the JUMP is executed, the address of the instruction labeled "200" should be stored into the PC, breaking the count.

The micro-PC is also counted sequentially, in the microprogram example shown in Appendix MP, the count always begins from 0 for every program instruction fetched. The count is again sequential unless broken by a successful TEST instruction. For example, the first two instructions of the microprogram are

```
0       MAINLOOP:          MAR=PC; MBR=MEMORY(MAR);
1                          IR=MBR; PC=PC+1;
```

they are executable micro-instructions, so they will not affect the micro-PC.

STEP #1 - The mPC will start at 0, so the micro-instruction

```
0100 0000 0000 0000 0000 1000 0000 0000 0000 0001 = 4000080001 hex
```

[illegible] executed. As you can see, the micro-instruction is coded with a "1" in bit positions 38, 19 and 0. The "1" in bit position 0 is to indicate that this is an executable micro-instruction, i.e., it is the opcode. Bit position 19 is "1" because it will open tri-state gate number 19 to allow the content of PC to move to MAR; this fulfills the "MAR=PC" part of the micro-instruction. Tri-state gate number 38 is opened one sub-cycle later, by this time the value of PC is already in MAR so the opening of gate 38 will allow the content of memory location pointed to by MAR to move to the memory buffer register MBR. This fulfills the "MBR=MEMORY(MAR)" part.

STEP #2 - While the last micro-instruction was being executed, the mPC is incremented to 1, so the next micro-instruction to be fetched is

```
0000 1000 0000 0000 0100 0000 0100 0000 0000 1001 = 0800404009 hex
```

Again bit0=1 is the opcode for an executable micro-instruction. bit22=1 allows the program instruction just fetched by the last miro-instruction to move from the MBR to the instruction register IR. In the IR, the program instruction will be decoded later by micro-instructions 2, 3, 4, 11, 17, 18, 28, 32, 33, 35 or 40. Depending on what the opcode of the program instruction is, the decoding process may take 3 to 6 micro-instructions (see page MP11). Note, bit22=1 fulfills the "IR=MBR" part of the micro-instruction. The other part is "PC=PC+1", incrementing the PC to get ready for the next program fetch from the main memory. To implement "PC=PC+1" requires two subcycles, the first subcycle to perform the addition and the second subcycle to store the sum back into PC. Therefore, in subcycle 1, the gates 3 and 14 are open to let the value of the PC to move to the left side of the adder and the constant "+1" to move to the right of the adder. After the adder has performed its task, the sum goes through the 1-bit shifter untouched in this case (shifts are used in multiply and divides only) and the opening of gate 35 in subcycle 2 allows the sum to be stored in the PC.

The next 3 micro-instructions are TEST instructions, i.e.,

```
2              IF BIT(15,IR)=1 THEN GO TO OP4567
3              IF BIT(14,IR)=1 THEN GO TO OP23
4              IF BIT(13,IR)=1 THEN GO TO POP
```

the labels OP4567, OP23 and POP represent micro-instruction location 17, 11 and 8, respectively. Labels are used to make programming simpler, once the symbol table has been created, they are no longer needed. In the top half of page 67 is the symbol table of this micro-program; the decimal as well as binary address of each label is given. With these addresses from the symbol table, the above micro-instructions can be coded as

```
000000 00010001 1 1000000000000000 01000000 0 = 0047000080 hex

000000 00001011 1 0100000000000000 01000000 0 = 002E800080 hex

000000 00001000 1 0010000000000000 01000000 0 = 0022400080 hex
```

By separating the bits as shown above, you can clear identify the addresses 00010001, 00001011 and 00001000 for OP4567, OP23 and POP, respectively. The "c" bit is 1 for all 3 micro-instructions because the test is done to see if selected bit in the IR is 1. The selected bits are 15, 14, and 13 for micro-instructions 2, 3, and 4, respectively. If all 3 TESTs fail, then the OPCODE of the program instruction would be 000, meaning a PUSH instruction. The micro-program for the PUSH instruction starts immediately after the third TEST instruction, i.e., locations 5 and 6. In location 7, the micro-instruction "GO TO MAINLOOP" starts the process again to perform a different program instruction, because the value of the PC is now pointing to a different memory location.

Lets assume that the program instruction is 000 0110011001101, a PUSH instruction in memory location 01441 octal. Clearly, the opcode is 000 and the address of the data is in memory address 06315 octal. The micro-program of the stack machine would go through the following steps:

mPC = 0 : The 13-bit address 01441 octal or "0 001 100 100 001" is moved from the PC to MAR. In the second subcycle, the instruction stored in Memory(0001100100001) is READ and WRITTEN into MBR. For this example, the instruction is 000 0110011001101.

mPC = 1 : The value of the MBR, i.e., 000 0110011001101, is moved into the IR. The PC is incremented from 0001100100001 to 0001100100010.

mPC = 2 : Test bit 15 of IR. In this case BIT(15,IR) is 0, so the result is false. The jump address OP4567 will not be used in this case, the new value of mPC is 3.

mPC = 3 : Test bit 14 of IR. In this case BIT(14,IR) is 0, so the result is false. The jump address OP23 will not be used in this case, the new value of mPC is 4.

mPC = 4 : Test bit 13 of IR. In this case BIT(13,IR) is 0, so the result is false. The jump address POP will not be used in this case, the new value of mPC is 5.

mPC = 5 : "MAR=IR", the content of IR is moved to MAR. Since the BUS between IR and MAR is 13 bits, only the least significant 13 bits will be copied. After this subcycle, MAR will contain 0110011001101, the address of the data.

"SP=SP+1", since this is a PUSH instruction, the stack pointer SP must be incremented before the data is copied onto the stack.

"MBR=Memory(MAR)", the data in Memory(0110011001101) is READ and WRITTEN into the buffer MBR. After this the mPC has value of 6.

mPC = 6 : "MAR=SP", the address of the TOP of stack is copied into the Memory Address Register MAR.
"Memory(MAR)=MBR", the data in the MBR, or the data fetched previously from Memory(0110011001101) is copied onto the TOP of stack as specified by MAR. The new value of mPC is 7.

mPC = 7 : "GO TO MAINLOOP", this unconditional JUMP instruction can be implemented by "IF BIT(15,0)=0 THEN GO TO MAINLOOP". The fact that the "ZERO" register has a zero in each bit position guarantees a successful TEST so the result will definitely be "GO TO MAINLOOP". The micro-address of MAINLOOP is 00000000, so after this TEST instruction, the value of mPC is 0.

mPC = 0 : THE NEXT PROGRAM INSTRUCTION WILL BE FETCHED ACCORDING TO [illegible] VALUE OF PC.

The entire microprogram for the STACK MACHINE was converted into binary and listed in page 67. The ROM that is required to store this microprogram is 89 words of 40 bits, about 3.5K bits or less than 1/2 KBytes. For a much more complicated processor, the microprogram is actually very large, each micro-instruction may be several hundred bits wide. Thus, the word "micro" is for most instances misleading. To create the microprogram for the powerful processors such as the Mototola 68020 or the INTEL 80386, the programming project requires many months of testing. Clearly, many algorithms developed for previous chips can be used again, so a "FAMILY" of chips may not be that hard to produce once the first one was made successfully.

A GENERAL RULE : Micro-programmers earn more than system-programmers (Assembler Language Level) and system-programmers earn more than Application Programmers (PASCAL, FORTRAN, COBOL, BASIC, C,...etc). Hence, the lower level you go, the fewer capable programmers.

Micro Programming Work Sheet

39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

NATIVE CODES VS EMULATED CODES

The OPCODE for most of the instructions in the STACK MACHINE are 3 bits long, but it still takes from 3 to 6 micro-instructions to decode. For a processor like the 8080 (see earlier pages), the opcode can range from 2 bits for a MOV instruction to 16 bits for some of the less frequently used instructions. The number of micro-instruction required to decode a long instruction may be quite large. In general, a significant fraction of the instruction's execution time is spent on decoding, therefore, the less decoding required, the more efficiently the processor can operate.

In the STACK MACHINE, there is a MUL and a DIV instruction, they are very long instructions, i.e., they require approximately 310 micro-instructions to complete (19 micro-instructions looped 16 times plus decoding). But out of the 310 mirco-instructions, only about 10 of them were for decoding, therefore, most of the time was used for calculation. For the processors which do not have a MUL or DIV instruction, the multiply or divide operation must be done using many SHIFT and ADD/SUBTRACT instructions, i.e., using a SOFTWARE subroutine to emulate the missing instructions. Although the same task can be accomplished, a lot more time is required to decode each ADD, SUBTRACT or SHIFT instructions. In another word, most of the computational time is used to decode the instructions rather than calculating.

If there is an algorithm, e.g., multiplying, that is used frequently, it would be much more efficient to code it in the microprogram level. Depending on the purpose of the processor, different instruction may be included as a NATIVE instruction while others have to be emulated using a combination of several basic native instructions. The earlier processors like the INTEL 8080, the Z80 (TRS-80) and the 6502 (APPLE II, ATARI) do not have a MUL or DIV instruction because of cost concerns. So when these older machines are used for word processing (not much arithmatics), they are fine. All the arithmatics, integers or floating point, must be software emulated. Hence, it is very difficult to do large scale calculation using hardware that were not designed for that purpose.

Recent technological advances allow more complicated micro-programs to be included in microprocessors, so the INTEL 8088, 80186 and 80286 all have 16-bit integer multiply and divide instructions. The more advanced INTEL 80386 and Motorola 68020 also have 32-bit multiply and divide instructions. But still, none of them have floating point add/subtract/multiply/divide/compare instructions. In fact, most general purpose processors to be introduced in the future will probably not have floating point instructions. Most manufacturers have gone with the idea of a MATH-COPROCESSOR, i.e., a separate processor dedicated to do floating point or high precision integer arithmatics only. The more famous processor pairs are

GENERAL PURPOSE PROCESSOR	MATHEMATICS COPROCESSOR	
INTEL 8088	INTEL 8087	(IBM-PC, IBM-PC/XT)
INTEL 80286	INTEL 80287	(IBM-PC/AT)
INTEL 80386	INTEL 80387	(new 386 machines)
MOTOROLA 68020	MOTOROLA 68881	(McIntosh II, Sun, Appollo)

The Math-coprocessors, or "FLOATING POINT HARDWARE" are introduced only recently. The Motorola 68881 came out in 1985, the INTEL 8087 in 1984. Many earlier versions of compilers or application programs (e.g., spread sheets) did not even support the floating point hardware. In fact, the so-called powerful machines like the AT&T 3B2 did not have a math coprocessor until 1986. For engineers and scientists, a machine is not worth the time and effort until there is "FLOATING POINT HARDWARE".

Shown in the APPENDIX 8087 is the list of floating point instructions available. It has instructions like FMUL, FDIV, FADD, FSUB, FMOVE, ..., etc. In addition to the arithmatic instructions, there are also a few trigonometric, inverse trigonometric, square root and exponential instructions. For the 8087, there is no FSIN or FCOS instructions, so a software subroutine must be written to use the tangent instruction FPTAN. In fact, FPTAN is good only for angles from 0 to pi/4, so even the tangent function would require a software routine. Shown in the next 3 pages are assembler language subroutines for tangent, sine and cosine. Pretty Ugly?

The Motorola 68881 is far superior to the INTEL 8087 or 80287. It has FSIN, FCOS, FTAN, ..., even a FSINCOS instruction which will give you both sine and cosine in a single call. It will also run at clock speed of 12.5, 16.7, 20 and 25 megahertz. The 80287 runs at 10 Mhz, the new 80386 family can run at 16 MHz. (Hughes aircraft missile processors run at 35 MHz). It is save to say that Motorola is now ahead of the game in 1987.

How does the 8088 and 8087 know when the execute and when to wait? The coprocessor concept is implemented using the ESC (escape) instruction of the 8088. When the ESC instruction is executed in the 8088, the 8087 coprocessor takes over and decodes the instruction. When the 8087 is executing, say a FMUL instruction, the 8088 is allowed to perform other instructions. The floating point instructions usually takes about 5 to 15 times longer than the general purpose integer instructions, so FORTRAN statements like

```
A(2*M-1)=A(2*M)*B(3*M-6)+3.19
```

would have the 8087 doing the floating point multiply and at the same time use the 8088 to calculate the address for A(2*M-1). Most compilers are not that good at these little things, so for maximum efficiency you shoulf write the assembler language program yourself.

Why not a Bessel's function instruction in the 68881? Because they are not use frequent enough by the people who buy these chips. But there are "HARDWARE FFT" boxes for doing Fast Fourier Transforms on digital data, no, not just sine and cosine, Fourier Integrals! There are also array processor which do vector multiplications (many multiplications concurrently like the CRAY) and more. They are not as famous as the 8087's or the 68881's because they are a lot more expensive and their market is limited to crazy scientists and rich defense contractors. Furthermore, most of them are not in the form of an integrated chip. Oh, by the way, floating point hardware is usually more expennsive than the general purpose processors.

Trigonometric Functions

The tangent function provides the base for calculating all the common trigonometric functions. FPTAN calculates the tangent for arguments between 0 and pi/4. Computation of a trigonometric function involves three broad steps. First, prologue code is used to bring the argument within range of the FPTAN instruction. Second, the FPTAN instruction is applied. Third, epilogue code is used to correct the result of FPTAN. The trigonometric identities used are described in the code below.

```
;TANGENT   (ST)                              370 MICROSECONDS
;THETA IN ST IS ASSUMED TO BE A VALID NUMBER
;THERE MUST BE AT LEAST 2 FREE STACK LOCATIONS
;THIS ROUTINE ASSUMES THAT THE FOLLOWING MEMORY
;LOCATIONS HAVE BEEN DEFINED:
;STATUS_WORD   2 BYTES
;SIGN_STORE    1 BYTE
;MINUS2        2 BYTES INITIALIZED TO -2
;
TANGENT    PROC       NEAR
           PUSH       AX
           PUSH       BX
;FIRST CHECK FOR A NEGATIVE ARGUMENT
```

```
;  NOTE TAN(-X)=-TAN(X)
           MOV        SIGN_STORE,0        ;ASSUME POSITIVE
           FTST
           FSTSW      STATUS_WORD
           FWAIT
           MOV        AH,BYTE PTR STATUS_WORD+1
           SAHF
           JNC        NON_NEGATIVE
           MOV        SIGN_STORE,-1       ;ITS NEGATIVE
           FABS                           ;NOW POSITIVE
NON_NEGATIVE:
;NOW GET ST BETWEEN 0 AND PI/4
           FILD       MINUS2              ;LOAD -2
           FLDPI                          ;LOAD PI
           FSCALE                         ;GOT PI/4
           FSTP       ST(1)               ;DUMP -2
           FXCH
; NOW X IS IN ST AND PI/4 IN ST(1)
RANGE:
           FPREM
           FSTSW      STATUS_WORD
           FWAIT
           MOV        AH,BYTE PTR STATUS_WORD+1
           SAHF
           JP         RANGE               ;THIS TESTS BIT C2
;AT THIS POINT AH HAS THE STATUS BITS
;NOW LETS SEE IF THE REMAINDER WAS EXACTLY ZERO
           FTST
           FSTSW      STATUS_WORD
           FWAIT
;IT WAS ZERO IF C3=1 AND C0=0
;IF ZERO, SET BX=-1, ELSE BX=0
           MOV        BX,0
           AND        BYTE PTR STATUS_WORD+1,01000001B
           CMP        BYTE PTR STATUS_WORD+1,01000001B
           JNE        NOT_ZERO
           MOV        BX,-1
NOT_ZERO:
;THERE ARE FOUR POSSIBILITIES GIVEN ST NOW HAS X MOD PI/4

;OCTANT     C3   C1    CALCULATE             IF ZERO
;0,4        0    0     FPTAN(ST)                0
;1,5        0    1     1/FPTAN(PI/4 - ST)       1
;2,6        1    0     -1/FPTAN(ST)          INFINITY
;3,7        1    1     -FPTAN(PI/4 - ST)       -1
;
;FIRST CHECK BIT C1 AND TAKE FPTAN
           TEST       AH,10B              ;IS C1 ON
           JZ         C1ISOFF             ;JUMP IF OFF
           CMP        BX,0                ;ST EXACTLY ZERO?
           JNE        STOANDC1            ;JUMP IF YES
```

```
        FSUBP     ST(1),ST          ;NOW PI/4-ST
        FPTAN
        JMP       TANDONE
STOANDC1:
        FSTP      ST                ;POP ST
        FSTP      ST                ; AND PI/4
        FLD1                        ;LOAD RATIO 1 TO 1
        FLD1
        JMP       TANDONE
C1ISOFF:
        FSTP      ST(1)             ;GET RID OF PI/4
        CMP       BX,0              ;ST EXACTLY ZERO?
        JNE       STOANDNOC1        ;JUMP IF YES
        FPTAN
        JMP       TANDONE
STOANDNOC1:
        FSTP      ST                ;DUMP ST
        FLDZ                        ;LOAD RATIO 0 TO 1
        FLD1
TANDONE:
;PUT C1 XOR C3 IN BX
        MOV       BX,0              ;ASSUME C3 OFF
;IF C3 IS ON THEN CHANGE SIGNS
        TEST      AH,01000000B
        JZ        NOC3              ;JUMP IF OFF
        FCHS
        MOV       BX,1              ;NOTE C3 ON
NOC3:
;IS C1 ON ?
        TEST      AH,10B
        JZ        NOC1              ;JUMP IF OFF
        XOR       BX,1
        JMP       RECIP
NOC1:   XOR       BX,0
RECIP:
;IF BX=1 THEN WE WANT RECIPROCAL OF RATIO
        CMP       BX,1
        JNE       NORECIP
        FXCH
NORECIP: FDIVP    ST(1),ST          ;THAT'S IT
;DID WE ORIGINALLY CHANGE SIGN?
        CMP       SIGN_STORE,0
        JE        LEAVE_POS
        FCHS
LEAVE_POS:
        POP       BX
        POP       AX
        RET
TANGENT ENDP
```

Sine and cosine functions are also calculated using FPTAN. Since a cosine is just a sine rotated 90 degrees, we build the cosine routine to make use of the code for sines.

```
;SINE     {ST}                                       513 MICROSECONDS
;THETA IN ST IS ASSUMED TO BE A VALID NUMBER
;THERE MUST BE AT LEAST 3 FREE STACK LOCATIONS
;THIS ROUTINE ASSUMES THAT THE FOLLOWING MEMORY
;LOCATIONS HAVE BEEN DEFINED:
;STATUS_WORD   2 BYTES
;SIGN_STORE    1 BYTE
;MINUS2        2 BYTES INITIALIZED TO -2
;REALLY_COS    1 BYTE
SINE       PROC      NEAR
           PUSH      AX
           PUSH      BX
;FIRST CHECK FOR A NEGATIVE ARGUMENT
; NOTE SIN(-X)=-SIN(X)
           MOV       SIGN_STORE,0      ;ASSUME POSITIVE
           FTST
           FSTSW     STATUS_WORD
           FWAIT
           MOV       AH,BYTE PTR STATUS_WORD+1
           SAHF
           JNC       NON_NEGATIVE
           MOV       SIGN_STORE,-1     ;ITS NEGATIVE
           FABS                        ;NOW POSITIVE

NON_NEGATIVE:
           MOV       REALLY_COS,0      ;SINE, NOT COSINE
COS_ENTRY:
;NOW GET ST BETWEEN 0 AND PI/4
           FILD      MINUS2            ;LOAD -2
           FLDPI                       ;LOAD PI
           FSCALE                      ;GOT PI/4
           FSTP      ST(1)             ;DUMP -2
           FXCH

; NOW X IS IN ST AND PI/4 IN ST(1)
RANGE:
           FPREM
           FSTSW     STATUS_WORD
           FWAIT
           MOV       AH,BYTE PTR STATUS_WORD+1
           SAHF
           JP        RANGE             ;THIS TESTS BIT C2

;AT THIS POINT AH HAS THE STATUS BITS

;IF WE ARE REALLY DOING COSINE, WE NEED TO ADD TWO TO THE
 OCTANT
           CMP       REALLY_COS,0
           JE        ITS_SINE
;ADD INTO C3 AND CARRY INTO C0
           XOR       AH,01000000B
           TEST      AH,01000000B
           JNZ       NOCARRY
           XOR       AH,1B
```

```
NOCARRY:
ITS_SINE:
;NOW LETS SEE IF THE REMAINDER WAS EXACTLY ZERO
          FTST
          FSTSW       STATUS_WORD
          FWAIT

;IT WAS ZERO IF C3=1 AND C0=0
;IF ZERO, SET BX=-1, ELSE BX=0
          MOV         BX,0
          AND         BYTE PTR STATUS_WORD+1,01000001B
          CMP         BYTE PTR STATUS_WORD+1,01000001B
          JNE         NOT_ZERO
          MOV         BX,-1

NOT_ZERO:
;THERE ARE FOUR POSSIBILITIES GIVEN ST NOW HAS X MOD PI/4
;OCTANT     C3  C1      CALCULATE          IF ZERO
;0          0   0       SIN(ST)              0
;1          0   1       COS(PI/4 - ST)       SQRT(2)/2
;2          1   0       COS(ST)              1
;3          1   1       SIN(PI/4 - ST)     SQRT(2)/2
;
; OCTANTS 4-7 ARE JUST LIKE 0-3 ONLY NEGATIVE
;NOTE: IF TAN(THETA)=X/Y, THEN
;           SIN(THETA)=X/SQRT(X*X+Y*Y)
;           COS(THETA)=Y/SQRT(X*X+Y*Y)
;
;FIRST CHECK BIT C1 AND TAKE FPTAN
          TEST        AH,10B              ;IS C1 ON
          JZ          C1ISOFF             ;JUMP IF OFF
          CMP         BX,0                ;ST EXACTLY ZERO?
          JNE         STOANDC1            ;JUMP IF YES
          FSUBP       ST(1),ST            ;NOW PI/4-ST
          FPTAN
          JMP         SINDONE
 STOANDC1:
          FSTP        ST                  ;POP ST
          FSTP        ST                  ; AND PI/4
          FLD1                            ;LOAD RATIO 1 TO 1
          FLD1
          JMP         SINDONE

 C1ISOFF:
          FSTP        ST(1)               ;GET RID OF PI/4
          CMP         BX,0                ;ST EXACTLY ZERO?
          JNE         STOANDNOC1          ;JUMP IF YES
          FPTAN
          JMP         SINDONE
 STOANDNOC1:
          FSTP        ST                  ;DUMP ST
          FLDZ                            ;LOAD RATIO 0 TO 1
          FLD1
```

```
SINDONE:
;IS C1 XOR C3 TRUE?
          MOV         BX,0                ;ASSUME C3 OFF

;IF C3 IS ON
          TEST        AH,01000000B
          JZ          NOC3                ;JUMP IF OFF
          MOV         BX,1                ;NOTE C3 ON
NOC3:
;IS C1 ON ?
          TEST        AH,10B
          JZ          NOC1                ;JUMP IF OFF
          XOR         BX,1
          JMP         DOSINE
NOC1:     XOR         BX,0
DOSINE:
;IF BX=1 THEN WE WANT WANT COSINE FUNCTION
          CMP         BX,1
          JNE         SINFUNC
          FXCH
SINFUNC:
;ST(1)=X, ST(0)=Y
;SIN(THETA)=X/SQRT(X*X+Y*Y)
          FMUL        ST(0),ST(0)         ;ST(0)=Y*Y
          FLD         ST(1)               ;ST(0)=X
          FMUL        ST(0),ST(0)         ;ST(0)=X*X
          FADDP       ST(1),ST(0)         ;ST(0)=X*X+Y*Y
          FSQRT
          FDIVP       ST(1),ST(0)
;
;IS BIT C0 ON?
          TEST        AH,1B
          JZ          C0OFF
          NOT         SIGN_STORE
C0OFF:
;DO WE NEED TO CHANGE SIGN?

          CMP         SIGN_STORE,0
          JE          LEAVE_POS
          FCHS
LEAVE_POS:
          POP         BX
          POP         AX
          RET
SINE      ENDP

;COSINE   (ST)                                        510 MICROSECONDS
;THETA IN ST IS ASSUMED TO BE A VALID NUMBER
;THERE MUST BE AT LEAST 3 FREE STACK LOCATIONS
;THIS ROUTINE USES THE SINE ROUTINE
COSINE    PROC        NEAR
          PUSH        AX
          PUSH        BX
```

```
        FABS
        MOV       SIGN_STORE,0      ;ITS POSITIVE NOW
        MOV       REALLY_COS,-1
        JMP       COS_ENTRY
COSINE  ENDP
```

For further explanation of trigonometric calculations and for programs which perform sophisticated error checking, see

Getting Started With the Numeric Data Processor, by Bill Rash, Intel Corporation, Application Note AP-113.

Inverse Trigonometric Functions

The 8087 instruction FPATAN performs the core calculations for the inverse trigonometric functions: Arctan, Arcsin, Arccos, Arccot, Arccsc, and Arcsec. Just as FPTAN produces a result in the form Y/X, so FPATAN accepts an argument in the form Y/X. The inverse trigonometric functions require somewhat less programming, because the argument range is less restricted for FPATAN than for FPTAN. (The direct trigonometric functions are periodic, where the inverse trigonometric functions aren't.) For FPATAN, we need only assure that the arguments obey the relation $0 < Y < X < \text{infinity}$. Thus to compute Arctan(Z) we need to check seven cases: Z equal 0, Z positive or negative and ABS(Z) less than, equal to, or greater than 1. We bring Z into the proper range by using the identities:

$$\text{Arctan}(Z) = -\text{Arctan}(-Z)$$
$$\text{Arctan}(Z) = \text{pi}/2 - \text{Arctan}(1/Z)$$

```
;ARCTAN   {ST}                                  351 MICROSECONDS
;ST IS ASSUMED TO BE A NORMAL NUMBER
;THERE MUST BE AT LEAST 3 FREE STACK LOCATIONS
;THIS ROUTINE ASSUMES THAT THE FOLLOWING MEMORY
;LOCATIONS HAVE BEEN DEFINED:
;STATUS_WORD   2 BYTES
;SIGN_STORE    1 BYTE
ARCTAN     PROC      NEAR
           PUSH      AX
;THE FIRST PROBLEM IS TO CHECK FOR A ZERO OR
;      NEGATIVE ARGUMENT
           MOV       SIGN_STORE,0    ;ASSUME NON-NEGATIVE
           FTST
           FSTSW     STATUS_WORD
           FWAIT
           MOV       AH,BYTE PTR STATUS_WORD+1
           SAHF
           JA        POSITIVE
           JZ        ZERO            ;ASSUME ITS ZERO
           JMP       NEGATIVE
ZERO:
;ARCTAN(0)=0
           FSTP      ST(0)
           FLDZ
           JMP       DONE
NEGATIVE:      ;DEAL WITH A NEGATIVE ARGUMENT USING IDENTITY
;ARCTAN(-X)=-ARCTAN(X)
           FCHS
           MOV       SIGN_STORE,-1
POSITIVE:                            ; HOW DOES 1 COMPARE TO
                                      X
           FLD1
           FCOM
           FSTSW     STATUS_WORD
           FWAIT
           MOV       AH,BYTE PTR STATUS_WORD+1
           SAHF
           JA        Z_LT_1
           JC        Z_GT_1
;EXACTLY 1 RETURN ARCTAN(1)=PI/4
           FCHS                      ;ST NOW=-1
           FADD      ST(0),ST(0)     ;ST=-2
           FLDPI
           FSCALE                    ;ST NOW PI/4
           FSTP      ST(1)
           JMP       RESTORE_SIGN
Z_GT_1:
;USE IDENTITY ATAN(X)=PI/2 - ATAN(1/X)
           FXCH                      ;ST=Z,ST(1)=1
           FPATAN
           FLD1                      ;NOW ADJUST BY PI/2
           FCHS
           FLDPI
           FSCALE
           FSTP      ST(1)
           FSUBRP    ST(1),ST
           JMP       RESTORE_SIGN
Z_LT_1:
           FPATAN                    ;ST=1,ST(1)=Z
RESTORE_SIGN:
           TEST      SIGN_STORE,0FFH
           JZ        DONE
           FCHS
DONE:
           POP       AX
           RET
ARCTAN     ENDP
```