WANG LABORATORIES, INC.

2200 MVP COMPUTER ARCHITECTURE

Author: Bruce M. Patterson

November 30, 1979

ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851 • TEL. (617) 851-4111 • TWX 710-343-6769 • TELEX 94-7421

System Architecture

Wang 2200 computer systems employ a direction executi[illegible] high-level-language (HLL) architecture. With direct execution HL[illegible] systems the HLL is effectively the machine language of the computer. Unlike more conventional architectures where the source code is transformed into a distinct object code before processing, the direct execution system processes the source code directly.

The direct execution system provides a number of advantages over more traditional architectures, not the least of which is its conceptual simplicity. The more conventional layers of software including assemblers, linkage editors, compilers, and loaders are eliminated. The inherent conversational nature of the system facilitates programming and debugging. The debug run and execution run are identical. Error messages can easily include a listing of the actual source code. Program execution can be halted, single stepped, and restarted. Since there is no compilation phase, the system responds immediately to program entries and modifications. Programmers can understand the language semantics by observing the direct response of the system.

The 2200 provides the user with a single HLL, BASIC-2, which is used for all programming. Proficiency in system use is easily achieved since there is only one language to learn. A fundamental design criterion in the development of BASIC-2 was to provide a self-sufficient language that would be as flexible as conventional general purpose computer instruction sets. I/O and data handling language extensions provide t[illegible] user with flexibility not usually found in a high-level-language.

The 2200 is not a pure direct execution machine since the source code is preprocessed into a form more memory conservative and more efficiently interpreted. However, source and object differences are such that the preprocessor transformation is nearly completely reversible. As a result, only the condensed code is stored in the machine. The preprocessing function eliminates gross inefficiencies in memory, timing, and logic requirements.

2200 Hardware

2200 computers consist of a microprogrammed MSI processor coupled with a number of special purpose LSI I/O processors and controllers. The OS and language interpreter reside in a large control storage memory which is independent from user data memory; this microprogram directs the execution of the CPU and coordinates communication with the I/O processors. The independent I/O processors permit the overlap of the CPU and I/O processing. The CPU is relieved of the responsibility for controlling peripherals that would otherwise require frequent or dedicated CPU attention.

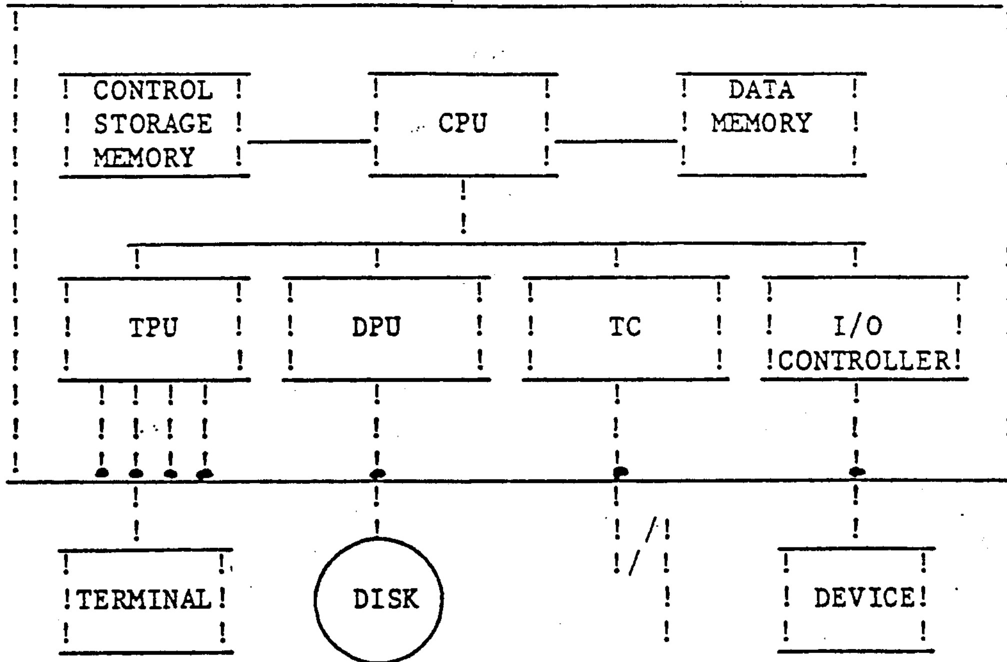


Figure 1 2200 System Block Diagram

The 2200 CPU is a pseudo 16-bit processor using a 3-bus architecture for interconnecting a bank of general purpose, status, and I/O 8-bit registers and the ALU. A microinstruction can address these registers as double, single, or half registers for performing 16, 8 or 4-bit operations. In addition, a bank of 16-bit registers that can be exchanged with the data memory address pointer provides quick access to major system pointers. The extensive microinstruction set consisting of 24-bit words provides decimal and binary arithematic, logical operations, and a wide variety of conditional branching instructions.

In a single CPU cycle, a 24-bit microinstruction can be fetched, 16-bits of data memory can be fetched, and a 16-bit operation can be performed. The wide memory path, 600 nsec. cycle time, and rich microinstruction set provides a highly effective processor for implementing direct execution languages.

User programs and system controllers are kept in data memory, of which 256K can be installed. Since the CPU's address space is limited to 64K, however, data memory is divided into 64K banks. In order to provide the microprogram with access to control tables without switching memory banks, the lower 8K of the address space always refers to bank 1. The lower 8K of banks 2, 3, and 4 is not used.

MVP Operating System

The 2200 MVP multiprogramming operating system allows several users to share a single computer effectively. To accomplish this, the operating system divides the resources of the computer -- memory, peripherals, and CPU time, -- among the users. Once each user has been allocated a share of the computer resources, the operating system acts as a monitor, allowing each user to utilize the system in turn while preventing the various users from interfering with each other's computations.

The MVP employs a fixed partition memory scheme. User memory is divided into a number of sections or "partitions", each of which can store a separate program. From the user's point of view, each partition functions independently from the other partitions in the system. Each user may LOAD and RUN BASIC software, compose a program, or perform Immediate Mode operations. As in a single-user environment, the user has complete control over his or her partition. No user on the system may halt execution in, or change the program text of, a partition controlled by another user.

Each terminal may control several partitions executing independent jobs. At any given time, however, only one of these partitions is in control of the terminal and thus capable of interacting with the operator. The partition in control of the terminal is said to be in the "foreground." Other partitions assigned to the terminal may continue to execute in the "background" until operator intervention becomes necessary.

Although partitions in general function independently of one another, there are situations in which it is useful for two or more partitions to cooperate. Cooperating partitions may share program text and/or data. The sharing of commonly used programs and data by several partitions eliminates needless duplication and produces more efficient use of available memory. The integrity and independence of a partition which contains shared programs or data are maintained by requiring the partition to explicitly declare itself to be global (sharable) before it can be accessed by other partitions. Correspondingly, a partition wishing to access shared text or data in a global partition must identify the desired global partition.

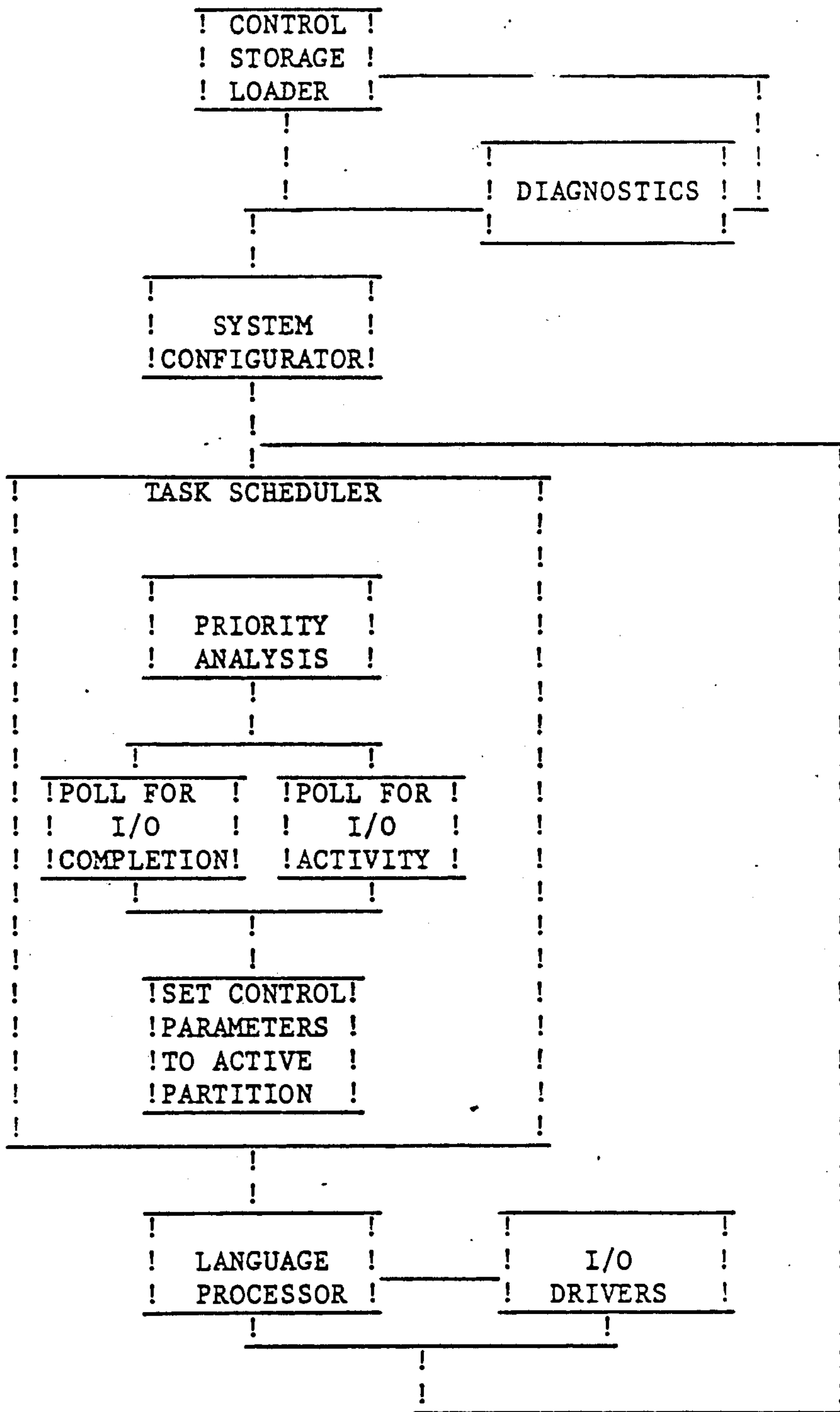


Figure 2 Block Diagram of 2200 MVP OS

To the programmer who regards the MVP system as a whole, it appears that all partitions are executing simultaneously. Because all partitions share a single CPU, however, only one partition can be executing at any given moment. The operating system creates the illusion of simultaneous execution of several programs by rapidly switching from one to the other in turn.

Partitions in the 2200 MVP are serviced by the CPU in a "round-robin" fashion, with priority given to I/O operations. Each partition in turn is given a "timeslice" 30 milliseconds (ms) in duration. The CPU has a 30 ms timer which is set at the beginning of the timeslice; at the completion of each BASIC statement (and at various points in the middle of long statements and I/O operations), the clock is checked to see whether the 30 ms timeslice has been exhausted. When a partition's timeslice has expired, the operating system saves the status of that partition so that it may be restored later when that partition's turn comes around again. The operating system then loads the status of the next partition in line and begins its 30 ms timeslice. The process of halting execution of a partition at the end of its timeslice is called a "breakpoint".

Timeslices do not always last exactly 30 ms. Unlike many operating systems, the MVP switches users (breakpoints) whenever it is convenient rather than strictly by the clock. This technique reduces the amount of status information that must be saved, giving the MVP low operating system overhead when compared with most other multiuser systems. More importantly, breakpoints may occur in the middle of BASIC I/O statements. If, for instance, the current partition attempts a disk access and the disk is hogged by another partition, this condition is quickly detected and a breakpoint occurs. I/O breakpoints differ from program breakpoints in that the partition is specifically marked as "waiting for I/O". When the partition's turn comes around again, the system takes only a few microseconds to decide whether processing may proceed or whether the partition is still waiting for the I/O device and may be bypassed. Thus, if a printer goes "busy" while it performs some mechanical function or if a partition that does not currently control the terminal attempts to write to the CRT, the system bypasses that partition almost as effectively as if it were removed entirely from the system until the I/O device becomes available.

WANG LABORATORIES, INC.

MEMORANDUM

TO: 2600 Distribution
FROM: F. Vine, B. Patterson
DATE: August 27, 1975, Revised September 12, 1975
SUBJECT: Revisions to 2600 Hardware Structure

This memo describes changes, as understood by 2600 microcoding groups, to the 2600 CPU specifications described in the document "2600 Calculator Structure" dated December 6, 1974, Revised February 14, 1975. Additional specifications are also provided. Updated pages for the specifications document are included. If any specifications are incorrect, please provide corrected specification A.S.A.P.

1. Deletion of binary add (A) instruction

   The register instruction binary add (A) has been eliminated from the micro-instruction repertoire. The binary add with carry (AC) instruction suffices since carry can be set off at the beginning of the instruction. Note, that the AI and ACI instructions have not been eliminated.

2. Addition of binary subtract with carry (SC) instruction

   The register instruction binary subtract with carry (SC) replaces the A instruction.

3. Write control memory (SR, WCM) instruction

   The SR, WCM instruction requires that the high 8-bits of the instruction being written (K register) be complemented; PH, PL remain as originally specified (true, uncomplemented).

   The data read by a SR, RCM instruction is true in K, PH, and PL.

4. Write to data memory on an extended register instruction

   In an extended register operation with write to data memory specified, the high order byte of the result is written (i.e., the result of the 2nd half of the operation).

5. Instruction timings

The cycle time is 600 nanoseconds for all instructions except for the following 3 that execute in 800 nanoseconds:

BLERX
BLRX
SR

and the following 2 instructions that execute in 1.6 microseconds.

SR, RCM
SR, WCM

and the following instruction that executes in 1.1 microsecond LPI.

6. Trap addresses (located in PROM/ROM bootstrap area)

8000 — PECM (parity error in control memory)
8001 — RESET
8002 — PEDM (parity error in data memory)
8003 — POWER ON

8004 — 800F (spares)

:

7. Load PC's immediate instruction extention

Write to data memory (W1, W2) are legal on LPI instructions; however, the data written is always zero since there are no extra bits available to specify what is to be written. In previous specifications write was illegal on LPI instructions.

8. Parity specifications

Page 14 describes instruction and data parity and parity errors.

WANG LABORATORIES, INC.

MEMORANDUM

TO: 2600 FILE

FROM: Bruce Patterson

DATE: December 6, 1974, Revised February 14, 1975, Revised Sept. 12, 1975

SUBJECT: 2600 Calculator Structure

The following memo describes the 2600 structure as of December 1, 1974. I/O specification will be described in another document. The following list summarizes the major changes to the specifications presented in the memo "2600 Calculator Structure" dated October 11, 1974:

1. Due to timing considerations, instructions that reference data memory will use the contents of PH and PL at the beginning of the instruction as the memory address. Previously, the contents of PH and PL at the end of the instructions were to be used. The LPI instruction is the only exception to the rule; if an LPI instruction specifies a read or write, the new contents of PH and PL will be used as the memory address.

2. The codes for the Mini Instructions and SHFT instruction have been slightly changed for easier decoding.

3. The instructions that Read and Write control memory (RCM, WCM) have been replaced by 2 new instructions (SR, RCM and SR, WCM).

Note, that the SHFT instruction has been modified so that either the high or low 4-bits of both the A and B BUS registers can be specified. Also, A-BUS specification of PC incrementing and decrementing is disabled during extended operations.

WANG LABORATORIES, INC.

M E M O R A N D U M

TO: 2600 File

FROM: Norman Lourie, Bob Kolk, Bruce Patterson

DATE: October 11, 1974, REVISED December 5, 1974, REVISED February 14, 1975, REVISED September 12, 1975

SUBJECT: 2600 Calculator Structure

This memo will specify in detail the register structure, instruction set and memory referencing structure for a 24-bit micro-programmed processor which is planned for the 2600. Although maintaining the A, B, C bus structure of the 800 micro-processor, it has a number of features which will significantly improve speed, code efficiency, and capacity.

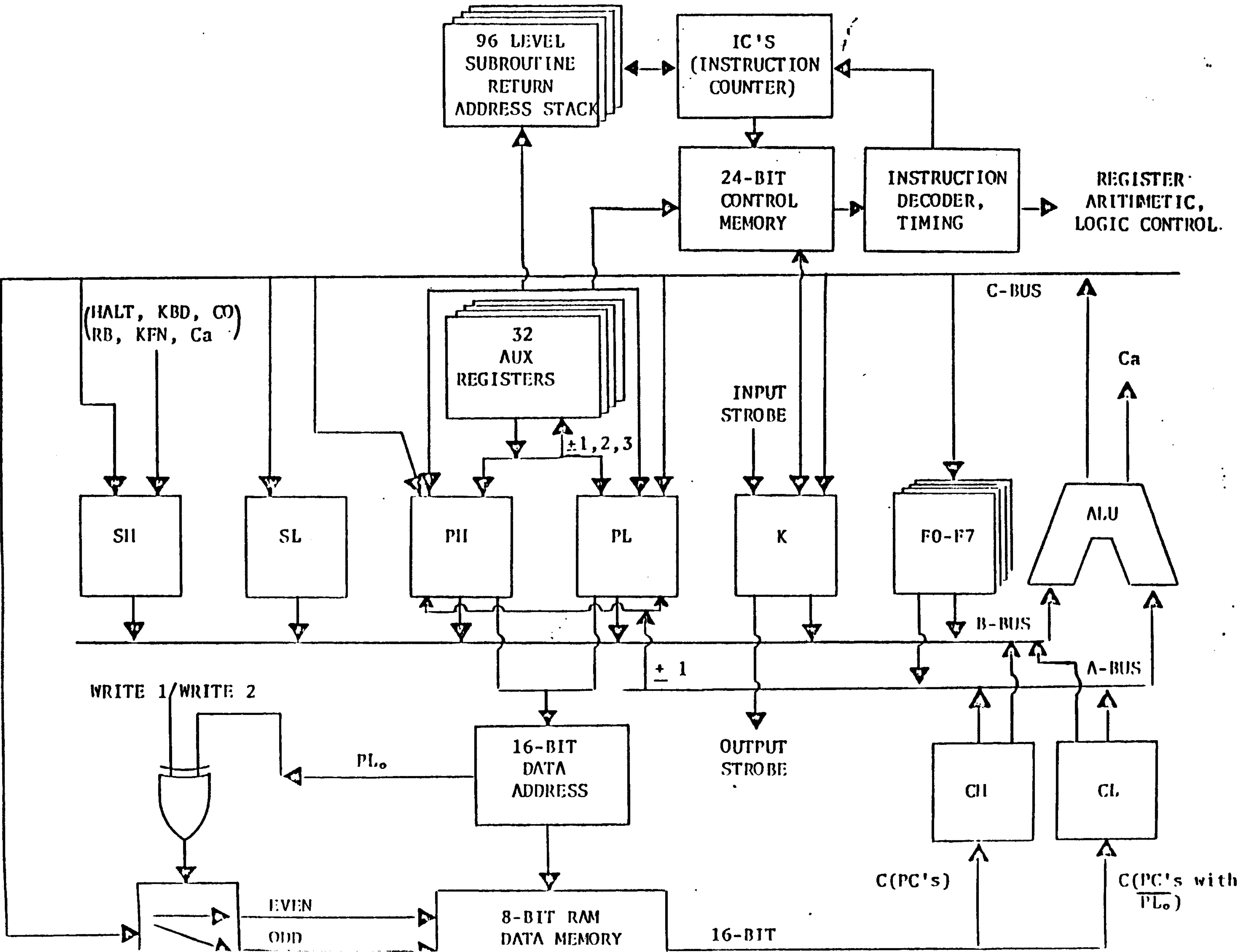
A. Register Structure

Figure 1 illustrates the tentative register structure for the processor.

The processor will contain 15 8-bit registers which can be read and/or written by micro-program instructions, an arithmetic logic unit and registers for holding the current 24-bit micro-program instruction and 16-bit address, and 32 16-bit auxiliary registers which back up the Data Memory Program Counter. In addition, an 8-bit dummy register exists, the dummy register cannot hold data; its value is always zero. Also, a 96 level subroutine stack is provided to allow efficient subroutine calling.

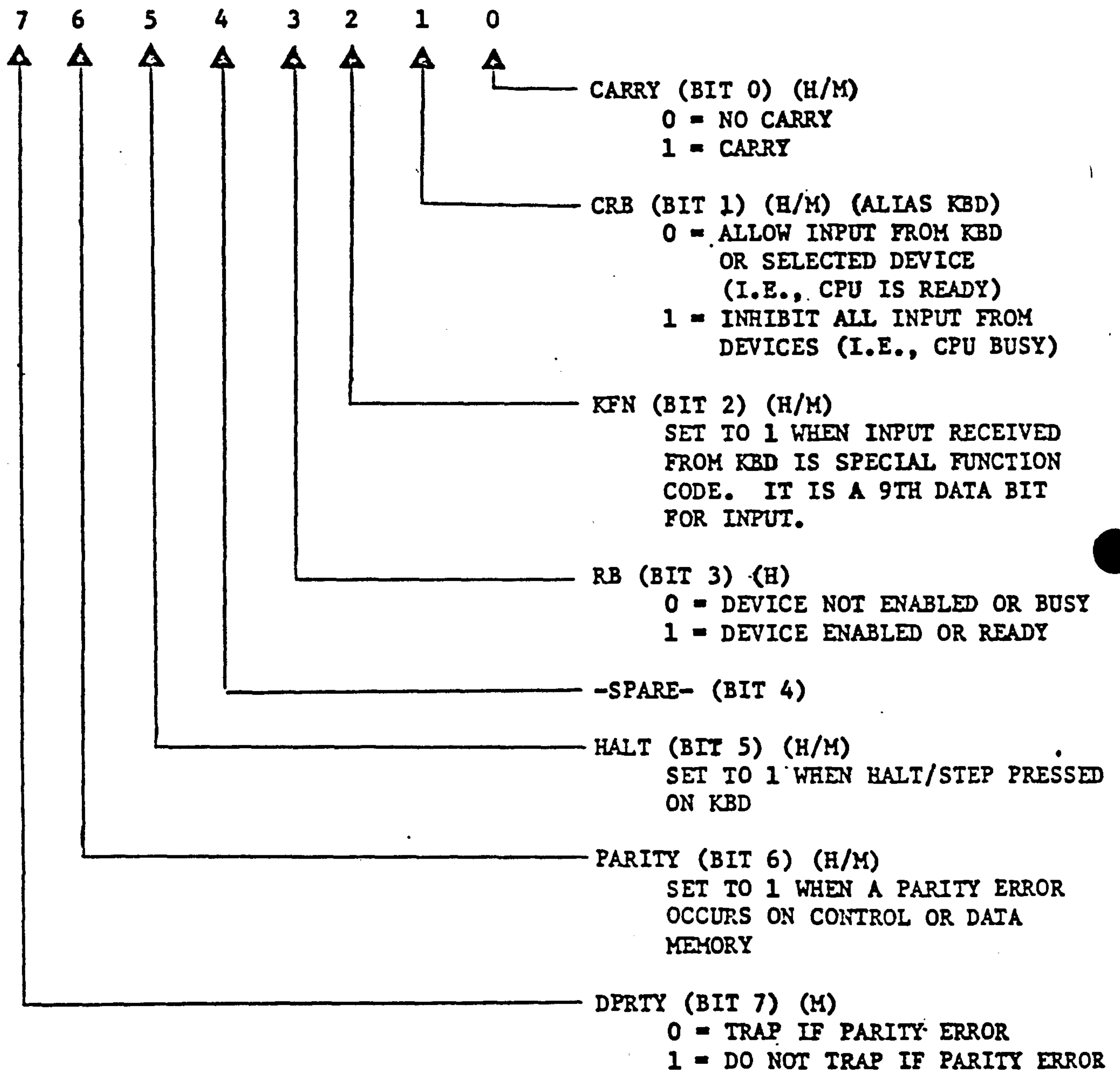
836 NORTH STREET, TEWKSBURY, MASSACHUSETTS 01876 • TEL. (617) 851-4111 • TWX 710-343-6769 • TELEX 94-7421

FIGURE 1. 2600 REGISTER JCTURE



(1) SH - Status Register High

An 8-bit register used to sense or set various arithemtic, I/O, and keyboard status conditions. It has the following assignments:



NOTE: (M) = Set by microprogram only.
(H) = Set by hardware only (D. C. level).
(M/H) = Set by microprogram or hardware.

(2) SL - Status Register Low

An 8-bit status register used by the micro-program to indicate phase of processing, mode, and other conditions. This register is set only by the micro-program.

(3) PH, PL - Data Memory Program Counters (PC's)

These 2 registers are used to hold the 16-bit current address of data words which are read from and written into Data Memory or Control Memory.

Data memory reads and writes are specified in the register instructions by use of the DD bits. For writes, 8-bit data is written from the C-bus to the Data memory location specified by the initial contents of the PC registers. For reads, 16-bits of data are read into the CH and CL registers. The details of memory addressing are described in a later section.

The PC's are also used for reading and writing the low 16-bits of a 24-bit instruction in control memory.

(4) K - Keyboard Input and I/O Registers

This 8-bit register is used to receive keyboard input and to receive and send data to and from other I/O devices. The K register is also used to read or write data to Control Memory.

(5) F0 - F7 - File Registers

These eight 8-bit registers are general purpose registers which will be used to perform arithmetic computations and related calculator processing. The file registers can be both source and object registers for any of the register transfer micro instructions.

(6) ALU - Arithmetic Logic Unit (5-bit path)

This unit is used to perform the addition, subtraction, and Boolean functions specified by the micro-program instructions.

Eight-bit data paths are input from the A and B bus and output to the C-bus. For add instructions, a carry bit is also transferred between the ALU and status register bit SH_0, if specified.

(7) AUX 0 - 1F - Auxiliary PC Save Registers

There are up to 32 16-bit registers which are used to temporarily save and restore the contents to the Data memory program counters (PH, PL). Sixteen bit transfers of PC's → AUX and AUX → PC's and a sixteen bit exchange are provided. These operations are extremely useful when Data is being moved, or when two sets of data are being operated on at the same time. When the address is transferred (or exchanged) to the Auxiliary registers, a 16-bit incrementing or decrementing of ± 1, ± 2, or ± 3 can be specified on the data received by the auxiliary register by certain mini-instructions.

The AUX registers are selected by the five Ax bits of the mini-instructions which specify the transfers and exchange.

(8) CH, CL - Data Memory Read Buffers

These two 8-bit registers are used to receive the 16-bits of data read from data memory. CH will always receive the 8-bits of data from RAM that is exactly specified by the 16-bit address in PH, PL. CL will receive the 8-bit word located at the address specified in PH, PL but with the low order bit of the address complemented. (i.e., the address in PC's ± 1).

(9) IC1, IC2, IC3, IC4 - Instruction Program Counter

The four 4-bit registers contain the current micro-program instruction address. Although these registers are not addressable by register instructions, they are reset by Branch, Subroutine Branch and Subroutine Return Instructions. A 96 level circular subroutine address save stack is available to save and restore the IC register. In addition, commands are available to transfer the PC registers (Data Memory program counter) to and from the stack.

B. Memory Addressing Structure

The processor can be considered to have two separate memories:

(1) Control Memory (24-bit RAM)

This RAM memory contains up to 64K of 24-bit words. It holds the micro-instructions. Instructions fetched and executed under control of the Instruction Program Counter, (IC1, IC2, IC3, IC4), which is indexed for sequential instruction execution and reset for branch, subroutine branch and return micro instructions.

Control Memory is available in increments of 4K words, up to 64K words. Since only 10 bits are referenced by some branch instructions, instruction memory can be thought of as paged memory with 1024 24-bit words per page for these instructions and an in-page jump is performed. Other instructions allow full 16-bit (64K) transfers.

(2) Data Memory (8/16-Bit RAM)

Data Memory is the memory which is read and written by the micro instruction. Up to 64K of 8-bit RAM (Random Access Memory) can be addressed.

The memory is addressed by the Data memory program counters (PH, PL). The program counter contains a 16-bit address which addresses a location in RAM.

Reads and writes are done by having non-zero data in the DD bits of register instructions. (00 = no read or write, 01 = read, 10 = write 1, 11 = write 2). For a read, 16-bits are read from Data memory. CH receives the 8-bits of data specified by the address in the PC's. CL receives the 8-bit word located at the address in the PC's but with the low order bit of the address (PL_0) complemented (i.e., the address in PC's $\pm$ 1).

For a write 1, 8-bits are written from the C-Bus (final result of a register instruction) to the address specified by the initial contents of the PC's. For a write 2, 8-bits are written from the C-Bus to the address in the PC's but with the low order bit of the address complemented. (i.e., the address is PC's $\pm$ 1).

Instruction	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
I. REGISTER INSTRUCTIONS	OPCODE					X		Carry		D	D	C-BUS				A-BUS				B-BUS		
OR - Or	0	0	0	0	0	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	B	[illegible]	[illegible]
XOR - Exclusive Or	0	0	0	0	1	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	[illegible]	[illegible]	[illegible]
AND - And	0	0	0	1	0	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	[illegible]	[illegible]	B
SC - Binary Subtract with Carry	0	0	0	1	1	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	B	B	B
DAC - Decimal Add with Carry	0	0	1	0	0	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	B	B	B
DSC - Decimal Subtract with Carry	0	0	1	0	1	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	B	B	B
AC - Binary Add with Carry	0	0	1	1	0	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	B	B	B
M - Binary Multiply	0	0	1	1	1	X	0	Hb	Ha	D	D	C	C	C	C	A	A	A	A	B	B	B
SHFT - Shift	0	0	0	Hb	Ha	X	0	0	1	D	D	C	C	C	C	A	A	A	A	B	B	B
II. IMMEDIATE REGISTER INSTRUCTIONS	OPCODE					IMMEDIATE (HIGH)				D	D	C-BUS				IMMEDIATE (LOW)				B-BUS		
ORI - Or Immediate	0	1	0	0	0	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B
XORI - Exclusive Or Immediate	0	1	0	0	1	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B
ANDI - And Immediate	0	1	0	1	0	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B
AI - Binary Add Immediate	0	1	0	1	1	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B
DACI - Decimal Add with Carry Immediate	0	1	1	0	0	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B
DSCI - Decimal Subtract with Carry Immed.	0	1	1	0	1	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B
ACI - Binery Add with Carry Immediate	0	1	1	1	0	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B
MI - Binary Multiply Immediate	0	1	1	1	1	0	-	Hb	-	D	D	C	C	C	C	I	I	I	I	B	B	B
III. MINI INSTRUCTIONS	OPCODE									D	D									B-BUS		
	0	0	0	1	0	0	1	1														
TAP - Transfer Aux to PC's	0	0	0	1	0	1	1	1	-	D	D	0	-	-	Ax	Ax	Ax	Ax	Ax	B	B	B
TPA - Transfer PC's to Aux	0	0	0	0	0	0	1	1	±	D	D	0	In	In	Ax	Ax	Ax	Ax	Ax	B	B	B
XPA - Exchange PC's to Aux	0	0	0	0	0	1	1	1	±	D	D	0	In	In	Ax	Ax	Ax	Ax	Ax	B	B	B
TPS - Transfer PC's to Stack	0	0	0	0	1	0	1	1	±	D	D	0	In	In	-	-	-	-	-	B	B	B
TSP - Transfer Stack to PC's	0	0	0	1	1	0	1	1	-	D	D	-	-	-	-	-	-	-	-	B	B	B
SR, RCM - Read Control Memory + SR	0	0	0	0	1	1	1	1	-	-	-	0	1	1	-	-	-	-	-	-	-	-
SR, WCM - Write Control Memory + SR	0	0	0	0	1	1	1	1	-	-	-	0	1	0	-	-	-	-	-	-	-	-
SR - Subroutine Return	0	0	0	0	1	1	1	1	-	D	D	0	0	-	-	-	-	-	-	B	B	B
CIO - Control Input/Output	0	0	1	0	1	1	1	1	-	0	0	S	T	T	T	T	T	T	T	-	-	-
	0	0	0	0	1	1	1	1														
	0	0	0	1	1	1	1	1														
LPI - Load PC's Immediate	0	0	1	1	I	I	1	I	I	D	D	I	I	I	I	I	I	I	I	I	I	I
IV. MASK BRANCH INSTRUCTIONS	OPCODE							BRANCH FIELD (LOW 10-Bits)								MASK				[illegible]		
BT - Branch if True	1	1	0	0	Hb	R	R	R	R	R	R	R	R	R	R	M	M	M	M	B	B	B
BF - Branch if False	1	1	0	1	Hb	R	R	R	R	R	R	R	R	R	R	M	M	M	M	B	B	B
BEQ - Branch if = Mask	1	1	1	0	Hb	R	R	R	R	R	R	R	R	R	R	M	M	M	M	B	B	B
BNE - Branch if ≠ Mask	1	1	1	1	Hb	R	R	R	R	R	R	R	R	R	R	M	M	M	M	B	B	B
V. REGISTER BRANCH INSTRUCTIONS	OPCODE							BRANCH FIELD (LOW 10-Bits)								A-BUS				B-BUS		
BLR - Branch if < Register	1	0	0	0	X	R	R	R	R	R	R	R	R	R	R	A	A	A	A	B	B	B
BLER - Branch if <= Register	1	0	0	1	X	R	R	R	R	R	R	R	R	R	R	A	A	A	A	B	B	B
BER - Branch if = Register	1	0	1	0	0	R	R	R	R	R	R	R	R	R	R	A	A	A	A	B	B	B
BNR - Branch if ≠ Register	1	0	1	1	0	R	R	R	R	R	R	R	R	R	R	A	A	A	A	B	B	B
VI. BRANCH INSTRUCTIONS	OPCODE					BRANCH FIELD (Low 10-Bits)										(High 6-Bits)						
SB - Subroutine Branch	1	0	1	0	1	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	-
B - Unconditional branch	1	0	1	1	1	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	-

KEY

- AAAA: A-BUS Register Address
- BBBB: B-BUS Register Address
- CCCC: C-BUS Register Address
- DD: Read/Write Specification
  - 00 = no read/write
  - 01 = read
  - 10 = write 1
  - 11 = write 2
- Hb, Ha: High/low 4-bits of register
  - 0 = low 4-bits
  - 1 = high 4-bits
- II...I: Immediate operand
- MMMM: Mask
- AxAxAxAxAx: Address of auxiliary register
- ± In In: Increment/decrement specification
  - 000 = PC's
  - 001 = PC's + 1
  - 010 = PC's + 2
  - 011 = PC's + 3
  - 100 = PC's
  - 101 = PC's - 1
  - 110 = PC's - 2
  - 111 = PC's - 3
- CaCa: Set carry (SH_0) specification
  - 00 = do not set carry
  - 10 = set carry to 0
  - 11 = set carry to 1
- X: Extended operation if X = 1
- RR...R: Branch address
- S: Set 10B flip-flops if S = 1
- TT...TT: Strobe specification
- -: Bit ignored (0 or 1 legal)

1. D D - Data Memory Read and Write Selection Bits

D D = 00 Null (No read or write)

D D = 01 Read; 16 bits read from memory into CH, CL
where C(PC's) → CH
C(PC's with $\overline{PL_0}$) → CL

D D = 10 Write 1; 8-bit write to memory
C-BUS → C(PC's)

D D = 11 Write 2; 8-bit switched write into memory
C-BUS → C(PC's with $\overline{PL_0}$)

2. A, B, and C-Bus Register Addressing

A-BUS	B-BUS	C-BUS	BINARY ADDRESS
File registers (F0 - F7)	F0-F7	F0-F7	0000 - 0111
CL with PC's = PC's - 1	PL	PL	1000
CH with PC's = PC's - 1	PH	PH	1001
CL	CL	illegal	1010
CH	CH	illegal	1011
CL with PC's = PC's + 1	SL	SL	1100
CH with PC's = PC's + 1	SH	SH	1101
Dummy with PC's = PC's + 1	K	K	1110
Dummy with PC's = PC's - 1	Dummy	Dummy	1111

1. The B-BUS and C-BUS registers are identical except that CL and CH are illegal on the C-BUS.

2. The A-BUS field can specify that the PC address bits be incremented or decremented by 1 at the completion of the instruction.

3. When the D D bits specify a read or write and the A-BUS field specifies a PC1 = PC1 ± 1, the read or write is executed before the PC's are incremented or decremented.

4. For mini commands with write selected, the B-BUS register will be written (before PC's are incremented or decremented, if applicable).

5. The "contents" of the dummy register is always zero.

3. <u>X - Extended Operation Bit</u>

Normally, a register instruction performs an 8-bit operation on the specified A-BUS and B-BUS registers and puts the result in the C-BUS register. A BLR (branch less than) or BLER (branch less than or equal) instruction compares two 8-bit registers and branches if the relation is true. In these cases, the extended operation bit is not set (i.e., X = 0).

If the extended operation bit is set (i.e., X = 1), a register instruction performs a 16-bit operation on a pair of A-BUS registers with a pair of B-BUS registers and puts the result in a pair of C-BUS registers. A BLR (branch less than) or BLER (branch less than or equal) instruction compares a pair of A-BUS registers with a pair of B-BUS registers and branches if the relation is true.

For extended operations, the register pair is treated as a single 16-bit register. The registers used are determined as follows. The low half of the pair is the register whose address is specified in the instruction. The high half of the pair is the register whose address is one more than the address specified.

EXTENDED OPERATION REGISTER PAIRS

A-BUS	B-BUS	C-BUS	BINARY ADDRESS
F1, F0	F1, F0	F1, F0	0000
F2, F1	F2, F1	F2, F1	0001
F3, F2	F3, F2	F3, F2	0010
F4, F3	F4, F3	F4, F3	0011
F5, F4	F5, F4	F5, F4	0100
F6, F5	F6, F5	F6, F5	0101
F7, F6	F7, F6	F7, F6	0110
CL, F7	PL, F7	PL, F7	0111
CH, CL	PH, PL	PH, PL	1000
CL, CH	CL, PH	illegal	1001
CH, CL	CH, CL	illegal	1010
CL, CH	SL, CH	illegal	1011
CH, CL	SH, SL	SH, SL	1100
Dummy, CH	K, SH	K, SH	1101
Dummy, Dummy	Dummy, K	Dummy, K	1110
F0, Dummy	F0, Dummy	F0, Dummy	1111

NOTE:

1. On extended operations A-BUS specification of PC incrementing or decrementing is disabled.
2. On an extended operation, if write is specified the value written is the high order result.

4. CaCa — Set Carry Field

All register instructions except M and SHFT can set the carry bit (SH_0) to 0 or 1 at the beginning of the instruction execution. The set carry options are:

CaCa = 00 — Do not set carry
CaCa = 10 — Set carry to 0
CaCa = 11 — Set carry to 1

(NOTE: 01 reserved for SHFT)

5. Ha, Hb — High/Low 4-Bit Selection

The Mask Branch, M, and MI instructions operate on either the high or low 4 bits of the A and/or B registers. The Ha bit specifies the high or low 4 bits of the A-Bus register; the Hb bit specifies the high or low 4 bits of the B-Bus register.

Ha = 0 Low 4-bits of A-Bus register
Ha = 1 High 4-bits of A-Bus register
Hb = 0 Low 4-bits of B-Bus register
Hb = 1 High 4-bits of B-Bus register

6. II...I — Immediate Operand

For Immediate Register Instructions, the actual 8 bits contained in the Immediate Operand Field (IIII) are gated directly to the A-Bus. For the LPI (load PC's immediate) instruction, the PC's are set equal to the 16-bit immediate field.

7. RR...R — Branch Addresses

The R field is the branch address specified by the micro-instruction. The 10-bit address branches are treated as in-page branching for theoretical pages of 1024 steps. (i.e., the upper 6 bits of the branch address are the same as that of the current instruction).

8. MMMM — Branch Mask

For the mask branch instructions, these 4 bits in the instruction have the following meaning:

Branch True, Branch False -- MMMM specifies what bits in the specified B-bus register are to be tested.
M = 1, test the corresponding bit; if M = 0, do not test the corresponding bit.

Branch Equal, Branch Not Equal -- MMMM is the 4-bit pattern to which the high or low 4 bits of the specified B-Bus register is to be compared.

9. AxAxAxAxAx — Auxiliary Register Field

This field specifies which of the 32 Auxiliary registers is to be used in the Auxiliary -- PC Mini-Instruction. Three mini-instructions (TPA, TAP, and XPA) transfer 16 bits between the program counter (PH, PL) and the specified Aux register (0 - 1F).

10. ± In In — Increment/Decrement Field

The ± In In field specifies whether or not the 16-bit value in the PC's is to be incremented or decremented (by 1, 2, or 3) as it is being transferred to the Auxiliary register (TPA, XPA) or subroutine return stack (TPS).

Field		Value	Source		Destination
± In In	=	000	PC's	→	Aux or stack
± In In	=	001	PC's + 1	→	Aux or stack
± In In	=	010	PC's + 2	→	Aux or stack
± In In	=	011	PC's + 3	→	Aux or stack
± In In	=	100	PC's	→	Aux or stack
± In In	=	101	PC's - 1	→	Aux or stack
± In In	=	110	PC's - 2	→	Aux or stack
± In In	=	111	PC's - 3	→	Aux or stack

E. Timing Sequence

The following timing sequence of events takes place for the 2600 micro-instructions:

Register and Mini-Instruction Timing Sequence

1. For LPI instructions, the PC registers are loaded with the specified value.

2. If DD bits specify a Read or Write, the contents of the Data Memory Program Counter (PH, PL) are transferred to the memory control logic to select the address.

3. The initial contents of the registers selected by the A-Bus (or Immediate Operand), and the B-Bus fields, and carry bit are gated to the Buses and into the ALU.

4. If set carry is specified (CaCa field of Register Instructions), the carry (SH_0) is set as specified.

5. The arithmetic or logical operation is performed in the ALU.

6. The results of the arithmetic or logical operation in the ALU is stored in the register specified by the C-Bus field.

7. If PC, stack, and Auxiliary Register transfers or exchanges are specified by the instruction, they are done. (TPA, TAP, XPA, TPS, TSP).

8. If Auxiliary register ± 1, ± 2, or ± 3 incrementing or decrementing is specified, (e.g., TP+1, XPA-3, XPA+2) ± 1, ± 2, or ± 3 is added with 16-bit of data received by the Auxiliary PC register.

9. If a Read or Write is specified, data is read into CH, CL or written from C-Bus (result of ALU operation) to memory.

10. If PC incrementing or decrementing is specified by the A-field, PC's are incremented or decremented by 1.

11. The Instruction Program Counter (IC's) is incremented by 1.

Branch Instruction Timing Sequence

1. For Conditional Branches, the test is made to branch or not branch based on the contents of the B-Bus Register.

2. If the test is valid, the branch is made by replacing the low order 10 or full 16-bits of the IC registers with the R instruction operands.

3. If the test is not valid, the IC counters are incremented by 1 to get the next instruction.

   (Note — For Subroutine Branches and Subroutine Returns, the address saved in the subroutine stack is the current instruction address + 1. The stack is circular.

4. If PC incrementing or decrementing is specified by the A-field, PC's are incremented or decremented by 1 after the branch test is performed. The incrementing will occur whether the test is true or false.

F. 2600 Trap Locations

16 control memory locations are reserved as traps (address 8000 through 800F). When a trap condition occurs, normal processing is immediately terminated and an automatic branch is made to the appropriate trap location. At the trap location is a branch instruction which transfers control to the specified microcode routine so that appropriate action can be taken. Presently, the following trap locations are defined:

8000 — PECM (parity error in control memory)
8001 — RESET
8002 — PEDM (parity error in data memory)
8003 — POR (power on — MASTER INITIALIZE)

G. Memory Parity

The 2600 uses odd parity on both control memory and data memory.

1. Control Memory Parity

The high order bit of each instruction in control memory is the parity bit; parity is odd. The parity bit of each instruction is generated by software; the SR, WCM instruction writes the 24 bits in the $\overline{K}$, PH, and PL (parity and instruction). The WCM instruction does not generate parity.

Instruction parity is checked when fetching an instruction for execution. If there is a parity error, the system will set parity error status bit (SH_6) = 1 and trap to location 8000_{16} in control memory. The address +1 of the instruction with bad parity is pushed into the subroutine return stack.

If the instruction (data) read by a SR, RCM instruction has bad parity, the parity error status bit (SH_6) is set to 1. No trap is made and the address of the instruction with bad parity is not saved in the stack.

2. Data Memory Parity

Odd parity is generated and written by the hardware at the time of a write to data memory.

On a read from data memory, parity is checked on the 16 bits read. If there is a parity error, the parity error status bits (SH_6) is set to 1. If the parity trap control status bit (SH_7) is set to 0, the system will trap to location 8002_{16} in control memory. If SH_7 = 1, the trap is inhibited. The address of the data with bad parity is not saved in the stack regardless of whether the system traps or not. Also, the PC's may not be the address of the data with bad parity (e.g., XPA, R).

3. Parity Status Bits

SH_6 — parity error. Set to 1 whenever bad parity is detected when fetching instructions or reading data.

SH_7 — parity trap control. (Data Memory)

- 0 = parity error trap for data memory enabled.
- 1 = parity error trap for data memory inhibited.

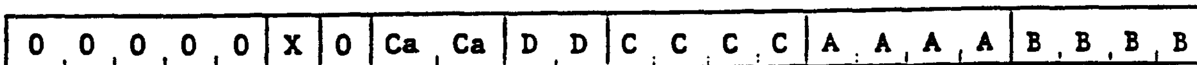
APPENDIX A

DETAILED DESCRIPTION

OF THE

INSTRUCTION SET

OR — OR



If X = 0, the OR of the registers specified by the A and B fields is formed and the result is stored in the register specified by the C field. If X = 1, the OR of the register pair specified by the A field is OR'ed with the register pair specified by the B field and the result is stored in the register pair specified by the C field.

Register use in the A, B, and C fields:

	A :	F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
	B :	F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 - F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 - F7, PL, , SL, SH, K, dummy

Carry (SH_0) options: CaCa = 00, do not change carry
CaCa = 10, set carry to 0 at beginning of instruction
CaCa = 11, set carry to 1 at beginning of instruction

Read/Write options:

D D = 00: no read or write
D D = 01: read
D D = 10: Write 1
D D = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

The A field can specify that PC's be incremented or decremented at the end of the instruction.

If A and/or B and/or C are set to indicate the dummy register, the net result will be:

A	=	Dummy	B →	C	[Memory]
B	=	Dummy	A →	C	[Memory]
C	=	Dummy	A or B →		[Memory]
A, B	=	Dummy	0 →	C	[Memory]
A, C	=	Dummy	B →		[Memory]
B, C	=	Dummy	A →		[Memory]
A, B, C	=	Dummy	0 →		[Memory]

XOR — EXCLUSIVE OR

If X = 0, the exclusive OR of the registers specified by the A and B fields is formed. The results are stored in the register specified by the C field. If X = 1, the register pair specified by the A field is exclusive OR'ed with the register pair specified in the B field and the result is stored in the register pair specified by the C field.

Register use in the A, B, and C fields:

	A :	F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
	B :	F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 - F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 - F7, PL, , SL, SH, K, dummy

Carry (SH_0) options: CaCa = 00, do not change carry
CaCa = 10, set carry to 0 at beginning of instruction
CaCa = 11, set carry to 1 at beginning of instruction

Read/Write options:

D D = 00: no read or write
D D = 01: read
D D = 10: Write 1
D D = 11: Write 2

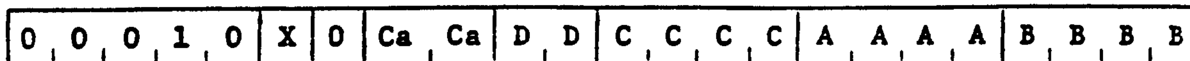
For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction. .

The A field can specify that PC's be incremented or decremented at the end of the instruction.

If A and/or B and/or C are set to indicate the dummy register, the net result will be:

A	=	Dummy	B	→	C	[Memory]
B	=	Dummy	A	→	C	[Memory]
C	=	Dummy	A ⊕ B	→		[Memory]
A, B	=	Dummy	0	→	C	[Memory]
A, C	=	Dummy	B	→		[Memory]
B, C	=	Dummy	A	→		[Memory]
A, B, C	=	Dummy	0	→		[Memory]

AND — AND



0	0	0	1	0	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	B	B	B	B

If X = 0, the AND of the registers specified by the A and B fields is formed. The result is stored in the register specified by the C field. If X = 1, the register pair specified in the A field is AND'ed with the register pair specified in the B field and the result is stored in the register pair specified in the C field.

Register use in the A, B, and C fields:

	A :	F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
	B :	F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 - F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 - F7, PL, , SL, SH, K, dummy

Carry (SH_0) options:

CaCa	=	00, do not change carry
CaCa	=	10, set carry to 0 at beginning of instruction
CaCa	=	11, set carry to 1 at beginning of instruction

Read/Write options:

D D	=	00:	no read or write
D D	=	01:	read
D D	=	10:	Write 1
D D	=	11:	Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

The A field can specify that PC's be incremented or decremented at the end of the instruction.

If A and/or B and/or C are set to indicate the dummy register, the net result will be:

A	=	Dummy	0 →	C	[Memory]
B	=	Dummy	0 →	C	[Memory]
C	=	Dummy	A . B →		[Memory]
A, B	=	Dummy	0 →	C	[Memory]
A, C	=	Dummy	0 →		[Memory]
B, C	=	Dummy	0 →		[Memory]
A, B, C	=	Dummy	0 →		[Memory]

SC — BINARY SUBTRACT WITH CARRY

0	0	0	1	1	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	B	B	B	B

If X = 0, the 8-bit register specified by the B field is complemented and added, with carry, to the 8-bit register specified by the A field. The final result is stored in the register specified by the C field, and SH_0 will receive the resultant carry. If X = 1, the register pair specified by the B field is complemented and added, with carry, to the register pair specified by the A field. The result is stored in the register pair specified by the C field, and SH_0 will receive the resultant carry.

Register use in the A, B, and C fields:

	A :	F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
	B :	F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 - F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 - F7, PL, , SL, SH, K, dummy

Carry (SH_0) options:
- CaCa = 00, do not change carry
- CaCa = 10, set carry to 0 at beginning of instruction
- CaCa = 11, set carry to 1 at beginning of instruction

Read/Write options:

- D D = 00: no read or write
- D D = 01: read
- D D = 10: Write 1
- D D = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

The A field can specify that PC's be incremented or decremented at the end of the instruction.

If SH is specified in the C-field, the results are indeterminate.

If A and/or B and/or C are set to indicate the dummy register, the net result will be:

A	= Dummy	- B - 1 + Carry	→ C	Carry	[Memory]
B	= Dummy	A - 1 + Carry	→ C	Carry	[Memory]
A, B	= Dummy	- 1 + Carry	→ C	Carry	[Memory]
C	= Dummy	A - B - 1 + Carry	→	Carry	[Memory]
A, C	= Dummy	- B - 1 + Carry	→	Carry	[Memory]
B, C	= Dummy	A - 1 + Carry	→	Carry	[Memory]
A, B, C	= Dummy	- 1 + Carry	→	Carry	[Memory]

DAC — DECIMAL ADD WITH CARRY

0	0	1	0	0	X	0	Ca	Ca	D	D	C	C	C	C	A	A	A	A	B	B	B	B

If X = 0, the 8-bit registers specified by the A and B fields are the last resultant carry (SH_0) are added together in decimal. The final sum is stored in the register specified by the C field and SH_0 will be set equal to the resultant carry. The addends must be decimal (0 - 9) or the sum will be indeterminent. If X = 1, the register pair specified by the A field and the last resultant carry are added in decimal to the register pair specified by the B field; the result is stored in the register pair specified by the C field and SH_0 receives the resultant carry.

Register use in the A, B, and C fields:

	A :	F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
	B :	F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 - F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 - F7, PL, , SL, SH, K, dummy

Carry (SH_0) options:
CaCa = 00, do not change carry
CaCa = 10, set carry to 0 at beginning of instruction
CaCa = 11, set carry to 1 at beginning of instruction

Read/Write options:

D D = 00: no read or write
D D = 01: read
D D = 10: Write 1
D D = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

The A field can specify that PC's be incremented or decremented at the end of the instruction.

If A or B field registers contain non-decimal data, or if SH is specified in the C-field, the results are indeterminate.

If A and/or B and/or C are set to indicate the dummy register, the net result will be:

A	=	Dummy	B + Carry	→	C	Carry	[Memory]
B	=	Dummy	A + Carry	→	C	Carry	[Memory]
A, B	=	Dummy	Carry	→	C	Carry	[Memory]
C	=	Dummy	A + B + Carry	→		Carry	[Memory]
A, C	=	Dummy	B + Carry	→		Carry	[Memory]
B, C	=	Dummy	A + Carry	→		Carry	[Memory]
A, B, C	=	Dummy	Carry	→		Carry	[Memory]

DSC — DECIMAL SUBTRACT WITH CARRY

If X = 0, the 8-bit register specified by the B field plus the last resultant carry (SH_0) is subtracted from the 8-bit register specified in the A field in decimal and the new carry is generated. (That is, the 9's complement of [the B register] and the carry are added to the A register and the new carry is generated). The result is stored in the C register.

Similarly, if X = 1, the register pair specified by the B field plus the last resultant carry is subtracted from the register pair specified by the A field in decimal and the new carry is generated. The result is stored in the register pair specified by the C field.

Register use in the A, B, and C fields:

	A :	F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
	B :	F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 - F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 - F7, PL, , SL, SH, K, dummy

Carry (SH_0) options:
CaCa = 00, do not change carry
CaCa = 10, set carry to 0 at beginning of instruction
CaCa = 11, set carry to 1 at beginning of instruction

Read/Write options:

D D = 00: no read or write
D D = 01: read
D D = 10: Write 1
D D = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

The A field can specify that PC's be incremented or decremented at the end of the instruction.

The A and B field registers must contain decimal data (0 - 9) or the results are indeterminate.

If SH is specified in the C field, the results are indeterminate.

A	= Dummy	- B - Carry	→	C	Carry	[Memory]
B	= Dummy	A - Carry	→	C	Carry	[Memory]
A, B	= Dummy	- Carry	→	C	Carry	[Memory]
C	= Dummy	A - B - Carry	→		Carry	[Memory]
A, C	= Dummy	- B - Carry	→		Carry	[Memory]
B, C	= Dummy	A - Carry	→		Carry	[Memory]
A, B, C	= Dummy	- Carry	→		Carry	[Memory]

AC — BINARY ADD WITH CARRY

If X = 0, the 8-bit registers specified by the A and B fields and the last resultant carry (SH_0) are added together in binary. The final sum is stored in the register specified by the C field, and SH_0 will receive the resultant carry. If X = 1, the register pair specified by the A field is and the last resultant carry are added in binary to the register pair specified by the B field; the resultant is stored in the register pair specified by the C field, and SH_0 will receive the resultant carry.

Register use in the A, B, and C fields:

	A :	F0 – F7, CL–, CH–, CL, CH, CL+, CH+, +, –
	B :	F0 – F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 – F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 – F7, PL, , SL, SH, K, dummy

Carry (SH_0) options:
CaCa = 00, do not change carry
CaCa = 10, set carry to 0 at beginning of instruction
CaCa = 11, set carry to 1 at beginning of instruction

Read/Write options:

D D = 00: no read or write
D D = 01: read
D D = 10: Write 1
D D = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

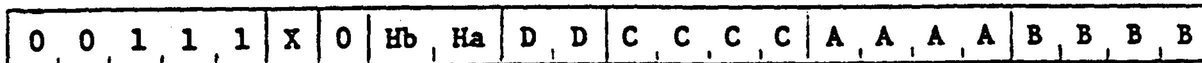
The A field can specify that PC's be incremented or decremented at the end of the instruction.

If SH is specified in the C-field, the results are indeterminate.

If A and/or B and/or C are set to indicate the dummy register, the net result will be:

A	= Dummy	0 + B + Carry	→	C	Carry	[Memory]
B	= Dummy	A + 0 + Carry	→	C	Carry	[Memory]
A, B	= Dummy	0 + 0 + Carry	→	C	Carry	[Memory]
C	= Dummy	A + B + Carry	→		Carry	[Memory]
A, C	= Dummy	0 + B + Carry	→		Carry	[Memory]
B, C	= Dummy	A + 0 + Carry	→		Carry	[Memory]
A, B, C	= Dummy	0 + 0 + Carry	→		Carry	[Memory]

M — BINARY MULTIPLY



0	0	1	1	1	X	0	Hb	Ha	D	D	C	C	C	C	A	A	A	A	B	B	B	B

If X = 0, the low (or high) 4-bits of the register specified in the A field is multiplied in binary by the low (or high) 4-bits of the register specified in the B field; the product (8-bits) is stored in the register specified by the C field. If X = 1, the above operation is performed; the above operation is then repeated but on the registers whose addresses are one greater than those specified in the A, B, and C fields.

Selection of high/low 4-bits of A, B registers:

HbHa = 00, low 4-bits of A and low 4-bits of B
HbHa = 01, high 4-bits of A and low 4-bits of B
HbHa = 10, low 4-bits of A and high 4-bits of B
HbHa = 11, high 4-bits of A and high 4-bits of B

Register use in the A, B, and C fields:

	A :	F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
	B :	F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 - F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 - F7, PL, , SL, SH, K, dummy

Read/Write options:

D D = 00: no read or write
D D = 01: read
D D = 10: Write 1
D D = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

The A field can specify that PC's be incremented or decremented at the end of the instruction.

If A and/or B and/or C are set to indicate the dummy register, the net result will be:

A	= Dummy	0 → C	[Memory]
B	= Dummy	0 → C	[Memory]
A, B	= Dummy	0 → C	[Memory]
C	= Dummy	A . B →	[Memory]
A, C	= Dummy	0 →	[Memory]
B, C	= Dummy	0 →	[Memory]
A, B, C	= Dummy	0 →	[Memory]

SHFT — SHIFT

0	0	0	Hb	Ha	X	0	0	1	D	D	C	C	C	C	A	A	A	A	B	B	B	B

If X = 0, the SHFT instruction sets the low 4-bits of the register specified by the C field equal to the high (or low) 4-bits of the register specified by the A field, and sets the high 4-bits of the C register equal to the high (or low) 4-bits of the B register. If X = 1, the above operation is performed; the above operation is then repeated on the registers whose addresses are one more than those specified in the A, B, and C fields.

Selection of high/low 4-bits of A, B registers:

Hb Ha = 00, high 4-bits of C = low 4-bits of B
low 4-bits of C = low 4-bits of A

Hb Ha = 01, high 4-bits of C = low 4-bits of B
low 4-bits of C = high 4-bits of A

Hb Ha = 10, high 4-bits of C = high 4-bits of B
low 4-bits of C = low 4-bits of A

Hb Ha = 11, high 4-bits of C = high 4-bits of B
low 4-bits of C = high 4-bits of A

Register use in the A, B, and C fields:

	A :	F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
	B :	F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
(if X = 0)	C :	F0 - F7, PL, PH, , SL, SH, K, dummy
(if X = 1)	C :	F0 - F7, PL, , SL, SH, K, dummy

Read/Write options:

D D = 00: no read or write
D D = 01: read
D D = 10: Write 1
D D = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

The A field can specify that PC's be incremented or decremented at the end of the instruction.

If A and/or B and/or C are set to indicate the dummy register, the net result will be:

A	=	Dummy	B 0	→	C	[Memory]
B	=	Dummy	0 A	→	C	[Memory]
C	=	Dummy	B A	→		[Memory]
A, B	=	Dummy	0	→	C	[Memory]
A, C	=	Dummy	B 0	→		[Memory]
B, C	=	Dummy	0 A	→		[Memory]
A, B, C	=	Dummy	0	→		[Memory]

ORI — OR IMMEDIATE

The OR of the register specified by the B field and the 8-bits in the I field are formed. The result is stored in the register specified by the C field.

Register use in B and C fields:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
C : F0 - F7, PL, PH, , SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the B field, the previous contents of the CH or CL will be used in the instruction.

If B and/or C are set to indicate the dummy register, the net result is:

B	=	Dummy	I → C	[Memory]
C	=	Dummy	B or I →	[Memory]
B, C	=	Dummy	I →	[Memory]

XORI — EXCLUSIVE OR IMMEDIATE

The exclusive OR of the register specified by the B field and the 8-bits in the I field are formed. The result is stored in the register specified by the C field.

Register use in B and C fields:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
C : F0 - F7, PL, PH, , SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the B field, the previous contents of the CH or CL will be used in the instruction.

If the B and/or C are set to indicate the dummy register, the net result is:

B	= Dummy	0 → C	[Memory]
C	= Dummy	$B \oplus I$ →	[Memory]
B, C	= Dummy	0 →	[Memory]

ANDI -- AND IMMEDIATE

0	1	0	1	0	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B	B

The AND of the register specified by the B field and the 8-bits in the I field are formed. The result is stored in the register specified by the C field.

Register use in B and C fields:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
C : F0 - F7, PL, PH, , SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the B field, the previous contents of the CH or CL will be used in the instruction.

If B and C are set to indicate the dummy register, the net result is:

B	=	Dummy	0 → C	[Memory]
C	=	Dummy	B . I →	[Memory]
B, C	=	Dummy	0 →	[Memory]

AI -- BINARY ADD IMMEDIATE

0	1	0	1	1	I	I	I	I	D	D	C	C	C	C	I	I	I	I	B	B	B	B

The 8-bit register specified by the B field and the 8-bits in the I field are added together in binary. The final sum is stored in the register specified by the C field.

Register use in B and C fields:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
C : F0 - F7, PL, PH, , SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the B field, the previous contents of the CH or CL will be used in the instruction.

If B and/or C are set to indicate the dummy register, the net result is:

B	=	Dummy	0 + I → C	[Memory]
C	=	Dummy	B + I → C	[Memory]
B, C	=	Dummy	I →	[Memory]

DACI — DECIMAL ADD IMMEDIATE WITH CARRY

The 8-bit register specified by the B field and the 8-bits in the I field and the last resultant carry (SH_0) are added together in decimal. The final sum is stored in the register specified by the C field. The resultant carry is stored in SH_0.

Register use in B and C fields:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
C : F0 - F7, PL, PH, , SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the B field, the previous contents of the CH or CL will be used in the instruction.

If SH is specified in the C field, the results are indeterminate. The addends must be decimal (0 - 9) or the results are indeterminate.

If B and/or C are set to indicate the dummy register, the net result is:

B	=	Dummy	I + Carry → C	Carry	[Memory]
C	=	Dummy	B + I + Carry →	Carry	[Memory]
B, C	=	Dummy	I + Carry →	Carry	[Memory]

DSCI -- DECIMAL SUBTRACT IMMEDIATE WITH CARRY

The 8-bit register specified by the B field plus the last resultant carry (SH_0) is subtracted in decimal from the 8-bits in the I field and the new carry is generated. (That is, the 9's complement of [the B register] and the carry are added to the immediate field and the new carry is generated). The result is stored in the register specified by the C field.

Register use in B and C fields:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
C : F0 - F7, PL, PH, , SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the B field, the previous contents of the CH or CL will be used in the instruction.

The I and B fields must contain decimal data (0 - 9) or the results are indeterminate.

If SH is specified in the C field, the results are indeterminate.

If B or C specify the dummy register, the results will be:

B	= Dummy	I - Carry → C	[Memory]
C	= Dummy	I - B - Carry →	[Memory]
B, C	= Dummy	I - Carry →	[Memory]

ACI -- BINARY ADD IMMEDIATE WITH CARRY

The 8-bit register specified by the B field and the 8-bits in the I field and the last resultant carry (SH_0) are added together in binary. The final sum is stored in the register specified by the C field. The resultant carry is stored in SH_0.

Register use in B and C fields:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
C : F0 - F7, PL, PH, , SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the B field, the previous contents of the CH or CL will be used in the instruction.

If SH is specified in the C field, the results are indeterminate.

If B and/or C are set to indicate the dummy register, the net result is:

B	= Dummy	I + Carry → C	[Memory]
C	= Dummy	B + I + Carry →	[Memory]
B, C	= Dummy	I + Carry →	[Memory]

MI -- BINARY MULTIPLY IMMEDIATE

The low (or high) 4-bits of the register specified by the B field is multiplied in binary by the 4-bit I field. The 8-bit result is stored in the register specified by the C field.

If Hb = 0, the low 4-bits of the B register are used.
If Hb = 1, the high 4-bits are used:

Register use in B and C fields:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy
C : F0 - F7, PL, PH, , SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the B field, the previous contents of the CH or CL will be used in the instruction.

If B or C specify the dummy register, the results will be:

B	= Dummy	0 → C	[Memory]
C	= Dummy	I . B →	[Memory]
B, C	= Dummy	0 →	[Memory]

TAP — TRANSFER AUX TO PC's

0	0	0	1	0	1	1	1	-	D	D	0	-	-	Ax	Ax	Ax	Ax	Ax	B	B	B	B

The contents of the auxiliary register specified by the Ax field is transferred to the PC registers.

Register use in the B field:

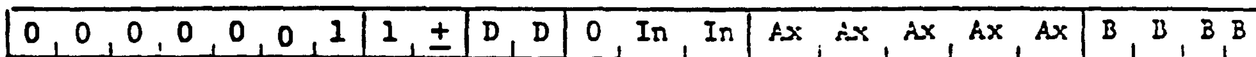
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For writes, the register specified in the B field will be written; if B specifies the dummy register, a zero will be written. The read or write address is the initial contents of the PC registers.

TPA -- TRANSFER PC's TO AUX



The 16-bit value in the PC's is optionally incremented or decremented by 1, 2, or 3 and transferred to the auxiliary register specified by the Ax field. The PC's are not modified.

± In In		PC's		
± In In	= 000	PC's	→	AUX
± In In	= 001	PC's + 1	→	AUX
± In In	= 010	PC's + 2	→	AUX
± In In	= 011	PC's + 3	→	AUX
± In In	= 100	PC's	→	AUX
± In In	= 101	PC's - 1	→	AUX
± In In	= 110	PC's - 2	→	AUX
± In In	= 111	PC's - 3	→	AUX

Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For writes, the register specified in the B field will be written; if B specifies the dummy register, a zero will be written. The read or write address is the initial contents of the PC registers.

XPA -- EXCHANGE PC's AND AUX

The 16-bit value in the PC's is optionally incremented or decremented by 1, 2, or 3 and exchanged with the 16-bit value in the auxiliary register specified by the Ax field.

± In In = 000	PC's	→	AUX
± In In = 001	PC's + 1	→	AUX
± In In = 010	PC's + 2	→	AUX
± In In = 011	PC's + 3	→	AUX
± In In = 100	PC's	→	AUX
± In In = 101	PC's - 1	→	AUX
± In In = 110	PC's - 2	→	AUX
± In In = 111	PC's - 3	→	AUX

Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For writes, the register specified in the B field will be written; if B specifies the dummy register, a zero will be written. The read or write address is the initial contents of the PC registers.

TPS -- TRANSFER PC's TO STACK

The 16-bit value in the PC's is optionally incremented or decremented by 1, 2, or 3 and transferred to the subroutine stack. The PC's are not modified.

Specifying incrementing or decrementing:

± In In	=	000	PC's →	stack
± In In	=	001	PC's + 1 →	stack
± In In	=	010	PC's + 2 →	stack
± In In	=	011	PC's + 3 →	stack
± In In	=	100	PC's →	stack
± In In	=	101	PC's - 1 →	stack
± In In	=	110	PC's - 2 →	stack
± In In	=	111	PC's - 3 →	stack

Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Read/Write options:

DD	=	00:	no read or write
DD	=	01:	read
DD	=	10:	Write 1
DD	=	11:	Write 2

For writes, the register specified in the B field will be written; if B specifies the dummy register, a zero will be written.

TSP -- TRANSFER STACK TO PC's

The last address in the subroutine stack is removed and transferred to the PC registers.

Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For writes, the register specified in the B field will be written; if B specifies the dummy register, a zero will be written. The read or write address is the initial contents of the PC registers.

LPI — LOAD PC's IMMEDIATE

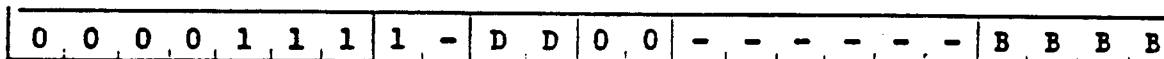
0	0	1	1	I	I	1	I	I	D	D	I	I	I	I	I	I	I	I	I	I	I	I

The PC registers are set equal to the 16-bits specified in the I field. If D = 1, data will be read from Data Memory; the read address will be the new contents of the PC's. If a write is specified, the data written will always be 0; the write address will be the new contents of the PC's.

Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: write 1 } the data written is always 0.
DD = 11: write 2 }

SR — SUBROUTINE RETURN



0	0	0	0	1	1	1	1	-	D	D	0	0	-	-	-	-	-	-	B	B	B	B

The last address stored in the 96 level subroutine stack is removed and transferred to the ROM Instruction Program Counter. The program execution will continue at that address.

Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

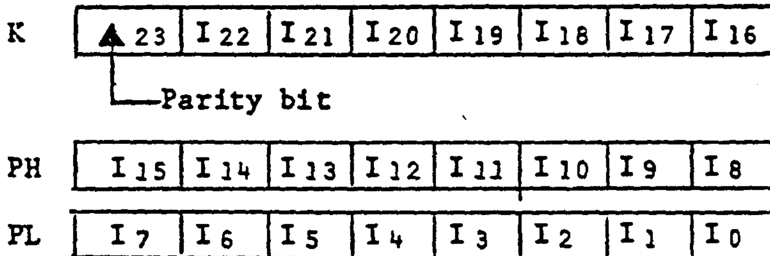
Read/Write options:

DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

For writes, the register specified in the B field will be written; if B specifies the dummy register, a zero will be written.

SR, RCM — READ CONTROL MEMORY AND SUBROUTINE RETURN

The SR, RCM instruction is used to read control memory. SR, RCM removes the last entry (16-bits) from the subroutine return stack; this value is the address of the instruction in control memory that is to be read. The specified instruction is read and stored in the registers K, PH and PL as follows:



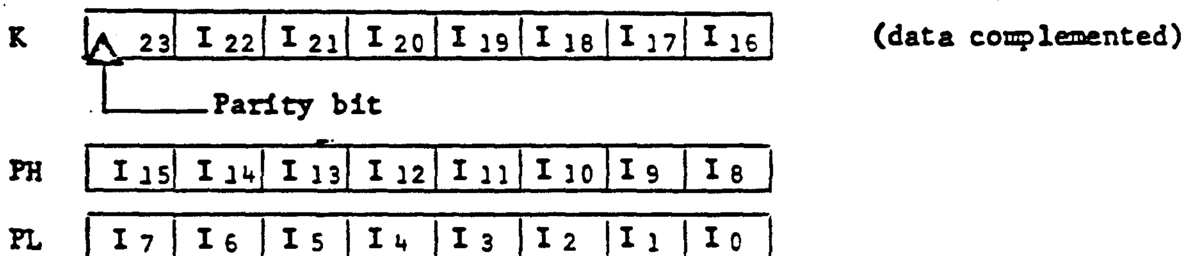
Finally, a normal SR is performed; that is, the next entry in the subroutine stack is removed, and transferred to the IC's (instruction counter). Program execution will continue at that address.

A typical instruction sequence for reading control memory is:

```
        .
        .
        .
        LPI   xxxx  set PC's to address of instruction to be read
        SB    RCM
        .
        .
        .
    RCM TPS         transfer address to stack
        SR, RCM     read control memory and return
```

SR, WCM — WRITE CONTROL MEMORY AND SUBROUTINE RETURN

The SR, WCM instruction is used to write into control memory. SR, WCM removes the last entry (16-bits) from the subroutine return stack; this value is the address of the location in control memory that is to be written to. The data in K, PH, and PL is written into control memory at the specified location; however, the data in K must be complemented. Instructions to be written are stored in K, PH, and PL as follows:



Finally, a normal SR is performed; that is, the next entry in the subroutine stack is removed, and transferred to the IC's (instruction counter). Program execution will continue at that address.

A typical instruction sequence for writing to control memory is:

```
        •
        •
        •
        MVI   x, K  }
        MVI   x, PH } K, PH, PL = instruction to be written  (K complemented)
        MVI   x, PL }
        TPA   ,0    save PH, PL in AUX
        LPI   xxxx  PC's = address to write to
        SB    WCM
        •
        •
        •
   WCM  TPS         transfer address to stack
        TAP   ,0    PH, PL = saved instruction
        XORI  OFF,K,K
        SR, WCM     write instruction and return
```

CIO — CONTROL INPUT/OUTPUT

0	0	1	0	1	1	1	1	-	0	0	S	T	T	T	T	T	T	T	-	-	-	-

If S = 1, load the IOB flip-flops with the contents of the K register. If a strobe is specified by the T field, it will be performed after the IOB flip-flops are set. The T field defines the type of strobe from the CPU to be performed. The following strobes are currently defined:

1. $\overline{ABS}$, Address Bus Strobe (TTTTTTT = 1000000)

   Each 2200 device has a unique 8-bit device associated with it. Only one device may be enabled (active) at a time. The device whose address is in the $\overline{IOB}$ address flip flops is enabled when the $\overline{ABS}$ strobe is sent. All other devices are disabled. The $\overline{IOB}$ flip flops may be set by the same instruction that issues the ABS.

2. $\overline{OBS}$, Output Bus Strobe (TTTTTTT = 0100000)

   $\overline{OBS}$ is a 5 usec data output strobe that sends the data in the K register out to the device which is currently enabled. Generally, the micro-program should check if the device is ready before the strobe is executed.

3. $\overline{CBS}$, Control Output Bus Strobe (TTTTTTT = 0010000)

   Same as $\overline{OBS}$ except strobe is on a different pin, and most devices use it for different purposes.

BT — BRANCH IF TRUE

1	1	0	0	Hb	R	R	R	R	R	R	R	R	R	R	M	M	M	M	B	B	B	B

The low (or high) 4-bits of the register specified by the B field are tested. If all of the bits specified by corresponding one bits in the M field are 1, a branch will be made to the in-page instruction memory address specified in the R field.

Since only 10 bits are specified in the R field, the branch in effect is an in-page branch with instruction memory being treated as paged memory with 1024 24-bit words per page. Therefore, when the branch is made, the low-order 10 bits of the instruction program counter are replaced by the R field.

If the mask is zero, an unconditional branch is made.

If the B field specifies the dummy register, the instruction will become a NOP, (No Branch), unless the mask is also zero, in which case an unconditional branch is made.

Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Specifying high or low 4-bits of B register:

Hb = 0 low 4-bits of B
Hb = 1 high 4-bits of B

BF — BRANCH IF FALSE

1	1	0	1	Hb	R	R	R	R	R	R	R	R	R	R	M	M	M	M	B	B	B	B

The low (or high) 4-bits of the register specified by the B field are tested. If the register bits specified by corresponding one bits in the M field are all 0, a branch will be made to the in-page instruction address specified by the R field.

Since only 10 bits are specified in the R field, the branch in effect will be executed as an in-page jump with instruction memory being treated as paged memory with 1024 24-bit instructions per page. Therefore, when the branch is made, the low-order 10 bits of the instruction program counter are replaced by the R field.

If the B field specifies the dummy register, an unconditional branch will be made.

Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Specifying high or low 4-bits of B register:

Hb = 0 low 4-bits of B
Hb = 1 high 4-bits of B

BEQ -- BRANCH IF EQUAL TO MASK

The low (or high) 4-bits of the register specified by the B field are compared to the 4-bits in the M field. If they are equal, a branch will be made to the in-page instruction address specified by the R field.

Since only 10 bits are specified in the R field, the branch in effect is an in-page branch with instruction memory being treated as paged memory with 1024 24-bit words per page. Therefore, when the branch is made, the low-order 10 bits of the instruction program counter are replaced by the R field.

If the B field specifies the dummy register, 0 is compared to the 4 bits in the M field.

Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Specifying high or low 4-bits of B register:

Hb = 0 low 4-bits of B
Hb = 1 high 4-bits of B

BNE -- BRANCH IF NOT EQUAL TO MASK

The low (or high) 4-bits of the register specified in the B field are compared to the 4-bits in the M field. If they are not equal, a branch will be made to the in-page instruction address specified by the R field.

Since only 10 bits are specified in the R field, the branch in effect is an in-page branch with instruction memory being treated as paged memory with 1024 24-bit words per page. Therefore, when the branch is made, the low 10 bits of the instruction program counter are replaced by the R field.

If the B field specifies the dummy register, 0 is compared to the 4 bits in the M field.

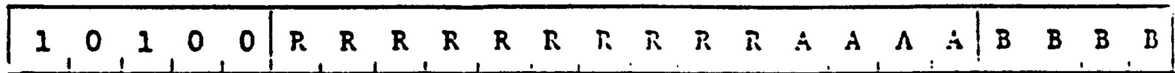
Register use in the B field:

B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Specifying high or low 4-bits of B register:

Hb = 0 low 4-bits of B
Hb = 1 high 4-bits of B

BER — BRANCH IF EQUAL TO REGISTER



The registers specified in A and B fields are compared. If they are equal, a branch will be made to the in-page instruction address specified by the R field.

Since only 10 bits are specified in the R field, the branch in effect is an in-page branch with instruction memory being treated as paged memory with 1024 24-bit words per page. Therefore, when the branch is made, the low-order 10 bits of the instruction program counter are replaced by the R field.

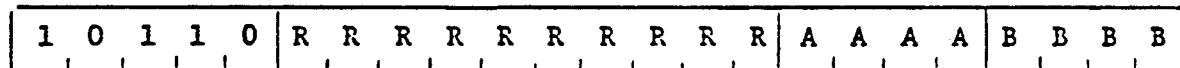
If A (or B) specify the dummy register, 0 is used in the compare.

Register use in the A, B fields:

A : F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

The A field can specify that PC's be incremented or decremented at the end of the instruction.

BNR -- BRANCH IF NOT EQUAL TO REGISTER



The registers specified in the A and B fields are compared. If they are not equal, a branch will be made to the in-page instruction address specified by the R field.

Since only 10 bits are specified in the R field, the branch in effect is an in-page branch with instruction memory being treated as paged memory with 1024 24-bit words per page. Therefore, when the branch is made, the low-order 10 bits of the instruction program counter are replaced by the R field.

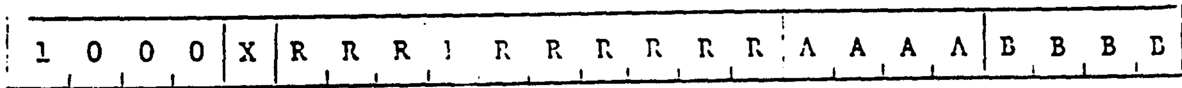
If A (or B) specify the dummy register, 0 is used in the compare.

Register use in the A, B fields:

A : F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

The A field can specify that PC's be incremented or decremented at the end of the instruction.

BLR -- BRANCH LESS THAN REGISTER



If X = 0, the registers specified in the A and B fields are compared. If X = 1, the register pairs specified in the A and B fields are compared. If A is less than B, a branch will be made to the in-page instruction address specified by the R field.

Since only 10 bits are specified in the R field, the branch in effect is an in-page branch with instruction memory being treated as paged memory with 1024 24-bit words per page. Therefore, when the branch is made, the low-order 10 bits of the instruction program counter are replaced by the R field.

If A (or B) specify the dummy register, 0 is used in the compare.

Register use in the A and B fields:

A : F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

The A field can specify that PC's be incremented or decremented at the end of the instruction.

BLER -- BRANCH LESS THAN OR EQUAL REGISTER

If X = 0, the registers specified in the A and B fields are compared. If X = 1, the register pairs specified in the A and B fields are compared. If A is less than or equal to B, a branch will be made to the in-page instruction address specified by the R field.

Since only 10 bits are specified in the R field, the branch in effect is an in-page branch with instruction memory being treated as paged memory with 1024 24-bit words per page. Therefore, when the branch is made, the low-order 10 bits of the instruction program counter are replaced by the R field.

If A (or B) specify the dummy register, 0 is used in the compare.
Register use in the A and B fields:

A : F0 - F7, CL-, CH-, CL, CH, CL+, CH+, +, -
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

The A field can specify that PC's be incremented or decremented at the end of the instruction.

SB -- SUBROUTINE BRANCH

An unconditional branch is made to the instruction memory address specified by the 16-bit address in the R field. In addition, the current contents of the Instruction Program Counter +1 are stored in the 96 level subroutine address stack. If the subroutine address stack already contains 96 addresses, the oldest address will be lost.

The rightmost 6 bits of the R field are the high order 6 bits of the branch address; the leftmost 10 bits are the low order 10 bits of the branch address.

B -- BRANCH

An unconditional branch is made to the instruction memory address specified by the 16-bit address in the R field. (i.e., the R field is transferred to the Instruction Program Counter).

The rightmost 6 bits of the R field are the high order 6 bits of the branch address; the leftmost 10 bits are the low order 10 bits of the branch address.

Index

Page

A1 . . . . OR — OR
A2 . . . . XOR — Exclusive OR
A3 . . . . AND — And
A4 . . . . SC — Binary Subtract with Carry
A5 . . . . DAC — Decimal Add with Carry
A6 . . . . DSC — Decimal Subtract with Carry
A7 . . . . AC — Binary Add with Carry
A8 . . . . M — Binary Multiply
A9 . . . . SHFT — Shift

A10 . . . . ORI — OR Immediate
A11 . . . . XORI — Exclusive OR Immediate
A12 . . . . ANDI — And Immediate
A13 . . . . AI — Binary Add Immediate
A14 . . . . DACI — Decimal Add with Carry Immediate
A15 . . . . DSCI — Decimal Subtract with Carry Immediate
A16 . . . . ACI — Binary Add with Carry Immediate
A17 . . . . MI — Binary Multiply Immediate

A18 . . . . TAP — Transfer Ax's to PC's
A19 . . . . TPA — Transfer PC's to Ax's
A20 . . . . XPA — Exchange PC's and Ax's
A21 . . . . TPS — Transfer PC's to Stack
A22 . . . . TSP — Transfer Stack to PC's
A23 . . . . LPI — Load PC's Immediate
A23 . . . . SR — Subroutine Return
A24 . . . . SR, RCM — Read Control Memory
A25 . . . . SR, WCM — Write Control Memory
A26 . . . . CIO — Control Input/Output

A27 . . . . BT — Branch if True
A27 . . . . BF — Branch if False
A28 . . . . BEQ — Branch if Equal to Mask
A28 . . . . BNE — Branch if Not Equal to Mask

A29 . . . . BER — Branch if Equal to Register
A29 . . . . BNR — Branch if Not Equal to Register
A30 . . . . BLR — Branch if Less than Register
A30 . . . . BLER — Branch if Less than or Equal Register

A31 . . . . SB — Subroutine Branch
A31 . . . . B — Unconditional Branch

WANG LABORATORIES, INC.

Virtual Assembly Language Editor

Program description

September 10 1980

revised May 11 1981

The following describes the function, operation and use of the Virtual Assembly Language Editor. This program was written by Max Blomme, 2200 Micro-code development (dept. 40).

A Purpose

To create and edit assembly language format data files for use as input to various assemblers operating on the 2200.

B Requirements

28K memory on a 2200VP or MVP (non-syntax checking) or
38K memory on a 2200VP or MVP (syntax checking)
Disk unit (a hard disk is preferred but a floppy may be used)
2226B (80x24), 2236D, 2236DE or 2236DW (last two preferred)
Printer is optional

C Features and Limitations

The virtual editor must have access to 503 sectors of catalog area on the disk for each user's work file. The edit file can be no longer than 999 lines. Caution must be exercised on long files, for some assemblers allow only up to 511 lines. For source of more than 999 (511) lines, multiple files, with unique file names, must be used. The convention to be used for file naming is that the first two characters are initials and the other six are neither all blank nor all numeric. When the file name is displayed on the CRT or entered from the keyboard, a period is included to separate the initials from the file name (this period is not actually written on the disk, so a LIST DC T will not show it).

The character set within a line includes all ASCII codes between Hex(10) and Hex(7A) that are available on a keyboard. Care should be taken not to create data which the assembler cannot handle. Backspace, space, and carriage return are used as special characters.

All or part of one or more edit files may be loaded into memory up to the restriction of 999 lines, individual lines may be edited, and groups of lines may be inserted or deleted from the file currently in the work area. The file currently in the work area may be saved, either over an old disk file, or in a newly defined disk file.

Finally, as an option, syntax checking for specific assembly languages is supported (currently Z80, 8080, 8085, 8051, 2270, 2280, 14IN, DSDD).

D Line format

Lines are edited in field format as follows:
The last data statement in the BASIC program has, among other things, the tab positions for the edit line. There are four fields and with the tab stops set at 1,10,20,51,123 they have the following meanings:

Field	Usual meaning	Length
1	tag field	8
2	opcode	9
3	operand	30
4	comment	71

The length of each field is constant and changeable only with a change of tab stops.

An asterisk at the beginning of the line, defines the whole line to be a comment whose maximum length is 123. Field orientated features, except for tab and reverse tab, are inactive for a comment line.

If the first character of a line is not an asterisk, the fields are as defined in the chart.

The line currently being edited is preceded by a right pointing triangle and the field being edited is underlined. The cursor is displayed at the character entry point. There is always at least one blank separating the fields.

E Initiating the Editor

Since the editor uses a work file on a disk, some conventions must be observed to prevent multiple users of the editor and the same work disk from using the same work file. This is usually solved by using the user's initials (two characters) and his (her) partition number.

When logging on, the users initials are asked for. Then all other users on the same work disk with the same initials are displayed and may be logged off. The users log-on tag is displayed and may be edited. From the position in the log-on file the user is assigned a unique workfile.

When the prompt is CODE; leave the field blank if no syntax checking is to be done (or if the syntax for the language being edited is not supported). As of this date the following syntaxes are supported: Z80, 8080, 8051, 8085, 2270, 2280, 14IN, DSDD. If CODE was not answered with a blank, a prompt LOGICAL EXPRESSION is shown. See the appropriate assembler memo for explanation for this.

The name of file to be edited is asked for, what disk it should be loaded from, and whether or not to load the file.

F Functions other than Edit

SF'00 MENU/EDIT Toggles between a menu listing the functions and edit

SF'01 DISK Most disk operations (SAVE, insert LOAD, and CREATE file) are done with this function.

SF'02 MOVE/COPY Moves (physically moves) or copies (replicating) the specified line(s) to an other place in the file and automatically inserts the proper amount of space.

SF'03 CAP. LOCK Allows the A/A, A/a switch to be in the lower case and have the appropriate fields still be in upper case. Four fields are represented, a 'U' means upper case only and an 'L' means upper or lower case allowed.

EDIT key LINE REQ. Allows quick access to a specific line.

SF'16 LOGOFF Logs the user off the editor and releases the work file he (she) was using.

SF'17 CHANGE NAME The name of the file currently in the work area or the name of the file to be loaded may be changed along with the disk address and the work area may be cleared and loaded with the file named earlier.

SF'18 SEARCH — Searches for the specified input from the current line and stops when the input is found. By keying the RUN key the searched-for input is replaced by the specified replacement. By keying the return key the next occurrence of the input is searched for.

SF'19 PRINTER LIST — Lists the selected lines to the selected printer.

G Editing the file

The following keys have special meaning while editing:

TAB (FN)	Non-destructively moves the cursor to the next field
TAB (shifted)	Reverse tab; non-destructively moves the cursor back one field
RETURN	Enters the line currently being edited (the line the cursor is on) into the work area and checks the syntax (if this option is enabled)
RUN	Does the same thing as RETURN but also inserts a blank line under the line currently being edited (this line is not really there until a RETURN is done on it)
space bar	Destructively moves the cursor to the next field, except in a comment field
CONTINUE	Changes the value of the next keystroke to an alternate value (space to a non-tabbing space, up arrow to a left arrow, slash to backslash) and as the second character in a blank comment line, the next keystroke is flooded into the line
BACKSPACE	Destructively moves the cursor one character space to the left (jumps across field boundaries)

The following applies to all keyboards except 2236DW

LINE ERASE	erases the whole line but does not delete it
CLEAR	Deletes the line currently being edited and moves the higher numbered lines up to fill the gap
recall	recall the line currently being edited to what it was prior to any keystroke unless the keystroke moved the cursor off the line
insert (lc)	puts the editor into insert mode or takes it out of insert mode
INSERT (uc)	allows insertion of a number of lines
delete (lc)	deletes a single character
DELETE (uc)	allows deletion of a number of lines
erase (sf'8)	erases the line starting from the cursor
ERASE (sf'24)	deletes the line currently being edited and moves the lower numbered lines down to fill the gap (similar to CLEAR)
begin (lc)	puts the cursor at the beginning of the line currently being edited
BEGIN (uc)	puts the cursor at the beginning of the file currently in the work area
end (lc)	moves the cursor to the end of the line currently being edited
END (uc)	moves the cursor to the end of the file currently in the work area
up arrow (lc)	moves the cursor up one line
UP ARROW (uc)	moves the cursor up one page
dn arrow (lc)	moves the cursor down one line
DN ARROW (uc)	moves the cursor down one page
left 5	moves the cursor left 5
left 1	moves the cursor left 1
right 1	moves the cursor right 1
right 5	moves the cursor right 5

The following applies to the 2236DW keyboard

ERASE(uc)	erases the whole line but does not delete it
erase(lc)	Deletes the line currently being edited and moves the higher numbered lines up to fill the gap
recall	recall the line currently being edited to what it was prior to any keystroke unless the keystroke moved the cursor off the line
insert (lc)	puts the editor into insert mode or takes it out of insert mode
INSERT (uc)	allows insertion of a number of lines
delete (lc)	deletes a single character
DELETE (uc)	allows deletion of a number of lines
PREV SCRN(uc)	puts the cursor at the beginning of the file currently in the work area
prev scrn(lc)	moves the cursor up one page
NEXT SCRN(uc)	moves the cursor to the end of the file currently in the work area
next scrn(lc)	moves the cursor down one page
WEST ARROW(uc)	puts the cursor at the beginning of the line currently being edited
west arrow(lc)	moves the cursor left 1
EAST ARROW(uc)	moves the cursor to the end of the line currently being edited
east arrow(lc)	moves the cursor right 1
up arrow	moves the cursor up one line
dn arrow	moves the cursor down one line

WANG LABORATORIES, INC.

Virtual Assembly Language Editor

Program description

September 10 1980

The following describes the function, operation and use of the Virtual Assembly Language Editor. This program was written by Max Blomme, 2200 Micro-code development (dept. 40).

ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851 · TEL. (617) 459-5000 · TWX 710-343-6769 · TELEX 94-7421

A Purpose

To create and edit assembly language format data files for use as input to various assemblers operating on the 2200.

B Requirements

28K memory on a 2200VP or MVP (non-syntax checking) or
38K memory on a 2200VP or MVP (syntax checking)
Disk unit (a hard disk is preferred but a floppy may be used)
2236D or 2236DE (preferred)
Printer is optional

C Features and Limitations

The virtual editor must have access to 503 sectors of catalog area on the disk for each user's work file. The edit file can be no longer than 999 lines. Caution must be exercised on long files, for some assemblers allow only up to 511 lines. For source of more than 999 (511) lines, multiple files, with unique file names, must be used. The convention to be used for file naming is that the first two characters are initials and the other six are neither all blank nor all numeric. When the file name is displayed on the CRT or entered from the keyboard, a period is included to separate the initials from the file name (this period is not actually written on the disk, so a LIST DC T will not show it).

The character set within a line includes all ASCII codes between Hex(10) and Hex(7A) that are available on a keyboard. Care should be taken not to create data which the assembler cannot handle. Backspace, space, and carriage return are used as special characters.

All or part of one or more edit files may be loaded into memory up to the restriction of 999 lines, individual lines may be edited, and groups of lines may be inserted or deleted from the file currently in the work area. The file currently in the work area may be saved, either over an old disk file, or in a newly defined disk file.

Finally, as an option, syntax checking for specific assembly languages is supported (currently Z80, 8080, 8085, 2280, DSDD).

D Line format

Lines are edited in field format as follows:
The last data statement in the BASIC program has, among other things, the tab positions for the edit line. There are four fields and with the tab stops set at 1,10,20,51,123 they have the following meanings:

Field	Usual meaning	Length
1	tag field	8
2	opcode	9
3	operand	30
4	comment	71

The length of each field is constant and changeable only with a change of tab stops.

An asterisk at the beginning of the line, defines the whole line to be a comment whose maximum length is 123. Field orientated features, except for tab and reverse tab, are inactive for a comment line.

If the first character of a line is not an asterisk, the fields are as defined in the chart.

The line currently being edited is preceded by a right pointing triangle and the field being edited is underlined. The cursor is displayed at the character entry point. There is always at least one blank separating the fields.

E Initiating the Editor

Since the editor uses a work file on a disk, some conventions must be observed to prevent multiple users of the editor and the same work disk from using the same work file. This is usually solved by using the user's initials (two characters) and his (her) partition number.

When logging on, the users initials are asked for. Then all other users on the same work disk with the same initials are displayed and may be logged off. The users log-on tag is displayed and may be edited. From the position in the log-on file the user is assigned a unique workfile.

When the prompt is CODE; leave the field blank if no syntax checking is to be done (or if the syntax for the language being edited is not supported). As of this date the following syntaxes are supported: Z80, 8080, 8085, 2280, DSDD. If CODE was not answered with a blank, a prompt LOGICAL EXPRESSION is shown. See the appropriate assembler memo for explanation for this.

The name of file to be edited is asked for, what disk it should be loaded from, and whether or not to load the file.

F Functions other than Edit

SF'00 MENU/EDIT Toggles between a menu listing the functions and edit

SF'01 DISK Most disk operations (SAVE, insert LOAD, and CREATE file) are done with this function.

SF'02 MOVE/COPY Moves (physically moves) or copies (replicating) the specified line(s) to an other place in the file and automatically inserts the proper amount of space.

SF'03 CAP. LOCK Allows the A/A, A/a switch to be in the lower case and have the appropriate fields still be in upper case. Four fields are represented, a 'U' means upper case only and an 'L' means upper or lower case allowed.

EDIT key LINE REQ. Allows quick access to a specific line.

SF'16 LOGOFF Logs the user off the editor and releases the work file he (she) was using.

SF'17 CHANGE NAME The name of the file currently in the work area or the name of the file to be loaded may be changed along with the disk address and the work area may be cleared and loaded with the file named earlier.

SF'18 SEARCH Searches for the specified input from the current line and stops when the input is found. By keying the RUN key the searched-for input is replaced by the specified replacement. By keying the return key the next occurrence of the input is searched for.

SF'19 PRINTER LIST Lists the selected lines to the selected printer.

G Editing the file

The following keys have special meaning while editing:

TAB (FN)	Non-destructively moves the cursor to the next field
TAB (shifted)	Reverse tab; non-destructively moves the cursor back one field
RETURN	Enters the line currently being edited (the line the cursor is on) into the work area and checks the syntax (if this option is enabled)
RUN	Does the same thing as RETURN but also inserts a blank line under the line currently being edited (this line is not really there until a RETURN is done on it)
CLEAR	Deletes the line currently being edited and moves the higher numbered lines up to fill the gap
space bar	Destructively moves the cursor to the next field, except in a comment field
CONTINUE	Changes the value of the next keystroke to an alternate value (space to a non-tabbing space, up arrow to a left arrow, slash to backslash) and as the second character in a blank comment line, the next keystroke is flooded into the line
BACKSPACE	Destructively moves the cursor one character space to the left (jumps across field boundaries)
LINE ERASE	erases the whole line but does not delete it
recall	recall the line currently being edited to what it was prior to any keystroke unless the keystroke moved the cursor off the line
insert (lc)	puts the editor into insert mode or takes it out of insert mode
INSERT (uc)	allows insertion of a number of lines
delete (lc)	deletes a single character
DELETE (uc)	allows deletion of a number of lines
erase (sf'8)	erases the line starting from the cursor
ERASE (sf'24)	deletes the line currently being edited and moves the lower numbered lines down to fill the gap (similar to CLEAR)
begin (lc)	puts the cursor at the beginning of the line currently being edited

BEGIN (uc)	puts the cursor at the beginning of the file currently in the work area
end (lc)	moves the cursor to the end of the line currently being edited
END (uc)	moves the cursor to the end of the file currently in the work area
up arrow (lc)	moves the cursor up one line
UP ARROW (uc)	moves the cursor up one page
dn arrow (lc)	moves the cursor down one line
DN ARROW (uc)	moves the cursor down one page
left 5	moves the cursor left 5
left 1	moves the cursor left 1
right 1	moves the cursor right 1
right 5	moves the cursor right 5

WANG LABORATORIES, INC.

TO: 2200 Microprogramming Group.
FROM: Matthew Lourie.
DATE: June 3, 1981.
SUBJECT: 2600 ASSEMBLER SPECIFICATIONS.

I. INTRODUCTION.

The 2600ASM is an assembler designed to run on the 2200VP/MVP and will convert 2600 source code into object code loadable by the 2600 bootstrap. This is not a released program and is intended for internal use only. Questions should be directed to the 2200 microprogramming group.

II. HARDWARE REQUIREMENTS:

A. 64K partition.
B. 132 column printer.
C. Some sort of Disk.

III. ASSOCIATED FILES:

A. "2600ASMS" contains the start-up program.

B. "2600ASM2" contains the assembler program.

C. "2600ASMB" contains the block allocating program.

D. "2600ASMG" contains the data file generating program.

E. "2600ASMD" contains the data file produced by "2600ASMG".

F. "SCROSS" contains the master cross reference program.

G. "READTP" contains the 9-track tape to printer program.

IV. OVERVIEW OF PROGRAMS:

A. Start-up:

1. During start-up the following device addresses will be requested:

a. Assembler output device.

This is the device to which the assembler sends listings, errors, etc. (see section on printing options).

b. Source edit file device.

This is the disk address on which the source files are located.

c. External symbol file device.

This is the disk address on which the symbol files are located.

d. Block work file device.

This is the disk address on which the block work file will be kept (see the section on blocks).

2. Then the following file names will be requested:

a. Object file name.

This is the file in which the object code will be stored. This file must be created before starting the assembly.

b. Starting module name.

This is the name of the first module to be assembled. All other modules of the assembly must be specified within the source code using the ASSEMBLE pseudo (explained later).

3. After requesting the previous items, and after asking various other questions (covered in other sections), the program chains to the main assembler program.

B. Main assembler program.

1. In order to allow large assemblies, the source code is able to be split up into logical divisions called modules. To assemble these modules the assembler must go through three or four passes:

a. Pass "W1" scans all the modules, getting the size of each block. It then passes this information to the block allocator which assigns the blocks addresses. Pass "W1" is skipped if the assembler is not in block mode.

b. Pass "W2" again scans the entire source, but this time creates a symbol table for each module and saves the table on the disk.

c. Pass "WF" scans a single module, building its symbol table in memory, and then goes on to pass "MF".

d. Pass "MF" scans a single module, producing the object code and printing the source file. Then control is passed back to pass "WF" thus initiating assembly of the next module.

2. In order to keep the user aware of the progress of an assembly, a constantly updated progess report is displayed on the CRT (i. e. current file, IC's, etc.).

C. Block allocator.

1. This program is chained to after completion of pass "W1".

2. Overview of purpose:

a. Reads in block sizes.
b. Creates spans.
c. Allocates addresses to blocks.
e. Prints a memory map out.
f. Chains back to the assembler, thus starting pass "W2".

D. Tape to printer program.

If a printer was specified as the tape output device, the tape to printer program is chained to after the assembler is done. This program will find the proper tape file and print its contents on the printer. The program can also be used as a stand alone for printing previously recorded tape file(s). To use the program in the stand alone mode do the following:

a. Run the program called "READTP".

b. Give the tape output address (default 77B) and the printer output address.

c. Give the file index numbers and the number of copies to be printed. The maximum number of file indices is ten.

d. Enter "0" in the file index field after all the desired file index numbers are entered. Thereafter the "READTP" will print out the files in the order in which their indices were entered.

V. BLOCKS.

A. Definition of a block.

A block is a segment of code that doesn't conditional reference addresses outside the segement. This means that a block can be moved around so long as the whole segment is contained on one page of memory. If a conditional reference is made outside a block, it will be flagged as an error.

B. Defining a block.

There are essentially two types of blocks, a floating block and an absolute block. A Floating block is allowed to be moved around by the assembler. This freedom enables the assembler to pack the code, allowing it to compactly fit into memory. This type of block is defined by starting it with an "ORG *". All modules must be valid blocks. An absolute block is a block which is defined to go at a certain address. They are defined by starting the code with an "ORG expression" where the expression is the block's address. The expression must not contain symbols which are defined within the current assembly. If the expression does, it will be flagged as an error. Aside from their predetermined address, absolute blocks are the same as floating blocks and may not conditionally reference other blocks.

C. Setting the LIMITS.

In order to tell the assembler where you want the code to go, you must use the LIMITS pseudo. The LIMITS pseudo is of the form:

```
LIMITS    lower address, upper address
```

The addresses should be expressions which do not use symbols defined during the current assembly. This psuedo effectively tells the assembler where to put the floating blocks. The floating blocks will be allocated addresses within the limits inclusively. Floating blocks will not be allocated addresses which conflict with absolute blocks. If a LIMITS pseudo is not specified or an invalid LIMITS pseudo is found, the assembler will give an error after pass "W1" and then abort. If the allocator cannot pack the code within the given space, a standard memory map chart will be printed out, displaying the blocks which have been allocated as well as the ones which have not. Then an error message will be given and the assembler will abort. If more than one LIMITS pseudo appears in the source, the first one will be used, and the others will be flagged as errors.

D. Allocation alogorithm.

The scrambling program has four modes:

1. Mode one tries to allocate the blocks in order. If it succeeds and listing order is specified it exits, else it switches to mode two.

2. Mode two does a fast scramble of all the blocks. Then it goes to mode three.

3. Mode three is a compression mode. It swaps the blocks around trying to fit them into smaller spaces. If after getting done the blocks fit, it exits. Otherwise it goes to mode four.

4. Mode four is incredibly slow. On a typical assembly of BASBOL, it took a half hour a pass! It is very, very unlikely that mode four will ever be needed on a large assembly.

E. Assembler modes.

a. When the block work file address is requested the user has two reply options:

1) If the user answers with "000" the assembler will function in the nonblock mode. This means that "ORG *" will be treated will be treated as do nothings, and the LIMITS psuedo is illegal.

2) If the user gives a valid address, the assembler will function in the block mode. A block work file will be opened with the name xxWK.TMP where xx are your initials. Later on, the start-up program will ask if you want the code in listing order. If you answer "Y", then the allocating program will not attempt to further compress the code so long as the code fits when it is in listing order.

F. Printing options.

The assembler offers three methods of output. These methods are as follows:

1. Write the assembler output directly to the printer. This is done by giving a valid printer address for the assembler output device.

2. Write the assembler output to the tape only. This is done by specifying a valid tape address for the assembler output device and specifying "000" for the tape printing device.

3. Write the assembler output to the tape, and after assembly completion have this tape file dumped to the printer. This is done by giving a valid tape address for the assembler output device and giving a valid printer address for the tape printing device.

VI. ASSEMBLER PSEUDOS.

A. MODULE

Defines a module title (should be first text line).

B. ORG

When in nonblock mode sets the assembler's instruction pointer to the specified address. In block mode it is used to start a block.

C. ORGD

Sets the assemblers data pointer to the specified address.

D. EQU

Assigns a symbol a value.

E. TITLE

Issue a form feed if not at top of page and print the specified comment.

F. SPACE

Skip the specified number of lines. A null operand implies skip one line.

G. EJECT

Issue a form feed if not at top of page.

H. PAGE

In nonblock mode it sets the assembler's instruction pointer to the beginning of the next page (1024 instructions) if not at the beginning of a page. In block mode this pseudo is ignored.

I. ASSEMBLE

Instructs that the specified module is part of the current assembly and is to be assembled after all previously specified modules have been assembled.

J. SYMBL

Instructs that the specified symbol file contains symbols which may be referenced in the current module.

K. CONT

Instructs that the specified source file is part of the current module and is to be assembled after all previously specified continue files have been assembled.

L. LIST and NOLIST

NOLIST causes the assembler to continue assembling, but disables listing. LIST causes the assembler to resume listing. At the start of a module, the assembler is automatically put into list mode. NOLISTs and LISTs are stacked. This way if two NOLISTs appear in a row, two LISTs are required before printing will resume. Cross references and title pages are always printed.

VII. SYMBOLS.

A. Local and multiply defined symbols.

If a symbol appears in the tag field of a line, that symbol is entered in the internal symbol table as a "local" symbol. If the symbol already appears in the symbol table the line which multiply defines the symbol is flagged with an error. The symbol is still entered into the symbol table, but as a "multiply defined" symbol.

B. External, previous, and undefined symbols.

If a symbol is referenced that is not defined within the current module, the external symbol tables defined by the SYMBL pseudos are scanned for the symbol. If the symbol is found in an external symbol table, that symbol is entered into the internal symbol table. It will be stored as an "external" symbol if it was defined within one of the modules of the current assembly. It will otherwise be store as a "previous" symbol. If the symbol is not found at all, it will be stored as an "undefined" symbol.

C. Symbol cross reference.

After each module is assembled the assembler will print out a cross reference of all symbols defined or referenced within the module. For each symbol the cross reference will display the symbol's value, the defining module's name (if not the current one), the defining line number, and the line numbers of all references to the symbol within the current module. If the symbol is an external symbol, an "X" will preceed the line number. If the symbol is a previous symbol, a "P" will preceed the line number. Otherwise a blank will preceed it.

VIII. INSTRUCTIONS.

A. Syntax.

In general an instruction consists of three fields, an opcode field, a parameter field, and an operand field. The opcode field contains the basic mnemonic (i.e. "OR", "ANDI", etc.). The parameter field contains the read/write and/or carry modifiers (i.e. ",R" etc.). And finally the operand field contains the register specifications. (For a more detailed explanation of instruction syntax see the BNF specification in the back).

B. Move Instructions.

The move instructions are implemented by using the "OR" and "ORI". Some examples are:

```
1.  MV   Fx,Fy        ==>   ORI  00,Fx,Fy

2.  MVI  n,Fx         ==>   ORI  n,,Fx

3.  MVX  FwFx,FyFz    ==>   ORX  DD,FwFx,FyFz
```

C. Omitted opcode field.

If the opcode field is omitted and the parameter field is not omitted, the source line is assembled as an "ORI" instruction.

D. Subroutine branch and return instruction.

Although it is not at first obvious, a subroutine branch followed by a subroutine return can always be replaced by simply a branch to the subroutine. The best way to see this is to think of the subroutine you want to call as the tail end of the current subroutine. For those of you who still like the idea of using a subroutine branch followed by a subroutine return, don't! Instead use the "SBR" mnemonic. It stands for "subroutine branch and return" but translates into a simple branch instruction.

ASSEMBLY LANGUAGE SYNTAX

The following pages define the syntax of the 2200 Assembly Language in Backus-Naur Form. The following meta symbols are used:

1. The "<" and ">" charactors enclose syntax classes.

2. The symbol "::=" means "is defined as".

3. The charactor "/" means "or".

4. "[" ... "]"^{x} means that the entries within may be repeated from zero to "x" times. If the "x" is omitted, assume a value of one.

5. "..." implies a sequence of elements.

6. Capital letters and symbols not in "<" ">" are actual letters in the language. Lower case letters represent English language expositions such as "space".

ASSEMBLER LINE FORMAT

```
<assembly line>  ::=  <micro line>
                 /    <pseudo line>
                 /    <data line>
                 /    [<symbol>]  <delimiter>  <delimiter>  <comment>
                 /    <null>
```

PSEUDO INSTRUCTION FORMAT

```
<pseudo line>  ::=  <pseudo>  <delimiter>  <comment>

<pseudo>       ::=  [<symbol>]  <delimiter>  ORG  <delimiter>  <address>

               /    [<symbol>]  <delimiter>  PAGE

               /    [<symbol>]  <delimiter>  ORGD  <delimiter>  <address>

               /    <symbol>  <delimiter>  EQU  <delimiter>  <word>

               /    <delimiter>  LIMITS  <address>  ,  <address>

               /    <delimiter>  MODULE  <delimiter>  <comment>

               /    <delimiter>  TITLE  <delimiter>  <comment>

               /    <delimiter>  SPACE  <delimiter>  [<expression>]

               /    <delimiter>  EJECT

               /    <delimiter>  CONT  <delimiter>  <file name>

               /    <delimiter>  SYMBL  <delimiter>  <file name>

               /    <delimiter>  ASSEMBLE  <delimiter>  <file name>
```

MICRO INSTRUCTION FORMAT

<micro line> ::= [<symbol>] <delimiter> <micro> <delimeter> <comment>

<register instruction> ::= OR/XOR/AND/DAC/DSC/AC/SC

<extended instruction> ::= ORX/XORX/ANDX/DACX/DSCX/ACX/SCX

<immediate instruction> ::= ORI/XORI/ANDI/AI/DACI/DSCI/ACI

<register multiply or shift> ::= MHH/MHL/MLH/MLL/SHFT

<extended multiply or shift> ::= MHHX/MHLX/MLHX/MLLX/SHFTX

<nibble multiply> ::= MIH/MIL

<half byte multiply> ::= MIHH/MIHL/MILH/MILL

<register branch> ::= BLR/BLER/BER/BNR

<extended branch> ::= BLRX/BLERX

<nibble branch> ::= BTH/BTL/BFH/BFL/BEQH/BEQL/BNEH/BNEL

<half byte branch> ::= BTHH/BTLH/BFHH/BFLH/BEQHH/BEQLH/BNEHH/BNELH
/ BTHL/BTLL/BFHL/BFLL/BEQHL/BEQLL/BNEHL/BNELL

<branch instruction> ::= B/SB/SBR

<aux instruction> ::= TAP/TPA/TPA+1/TPA+2/TPA+3/TPA-1/TPA-2/TPA-3
/ XPA/XPA+1/XPA+2/XPA+3/XPA-1/XPA-2/XPA-3

<transfer instruction> ::= TSP/TPS/TPS+1/TPS+2/TPS+3/TPS-1/TPS-2/TPS-3

<micro> ::= LPI [<rw>] <delimiter> <address>

/ CIO <delimiter> <byte>

/ SR [<rw>] <delimter> <b-reg>

/ SR <rw control>

/ INSTR <delimiter> <hexdigit> <hexdigit> <hexdigit>
<hexdigit> <hexdigit> <hexdigit>

/ MV [<rw>] <delimiter> <b-reg> , <c-reg>

/ MVI [<rw>] <delimiter> <byte> , <c-reg>

/ MVX [<rw and/or carry>] <delimiter> <b-ext> , <c-ext>

```
/    <register instruction>  [<rw and/or carry>]  <delimiter>
     <a-reg>  ,  <b-reg>  ,  <c-reg>

/    <extended instruction>  [<rw and/or carry>]  <delimiter>
     <a-ext>  ,  <b-ext>  ,  <c-ext>

/    <immediate instruction>  [<rw>]  <delimiter>
     <byte>  ,  <b-reg>  ,  <c-reg>

/    <rw and/or carry>  <delimter>
     <byte>  [,  <b-reg>  [,  <c-reg>]]

/    <register multiply or shift>  [<rw>]  <delimiter>
     <a-reg>  ,  <b-reg>  ,  <c-reg>

/    <extended multiply or shift>  [<rw>]  <delimiter>
     <a-ext>  ,  <b-ext>  ,  <c-ext>

/    <nibble multiply>  [<rw>]  <delimiter>
     <nibble>  ,  <b-reg>  ,  <c-reg>

/    <half byte multiply>  [<rw>]  <delimiter>
     <byte>  ,  <b-reg>  ,  <c-reg>

/    <register branch>  <delimiter>
     <a-reg>  ,  <b-reg>  ,  <address>

/    <extended branch>  <delimiter>
     <a-ext>  ,  <b-ext>  ,  <address>

/    <nibble branch>
     <nibble>  ,  <b-reg>  ,  <address>

/    <half byte branch>
     <byte>  ,  <b-reg>  ,  <address>

/    <branch instruction>  <delimiter>  <address>

/    <aux instruction>  [<rw>]  <delimiter>
     <b-reg>  ,  <aux-reg>

/    <transfer instruction>  [<rw>]  <delimiter>  <b-reg>
```

```
<a-reg>               ::=  F0/F1/F2/F3/F4/F5/F6/F7/CL-/CH-/CL/CH/CL+/CH+/+/-

<a-ext>               ::=  F1F0/F2F1/F3F2/F4F3/F5F4/F6F5/F7F6/CLF7
                      /    CHCL/CLCH/DCH/DD/F0D

<b-reg>               ::=  <c-reg>/CH/CL

<b-ext>               ::=  <c-ext>/CLPH/CHCL/SLCH

<c-reg>               ::=  F0/F1/F2/F3/F4/F5/F6/F7/PL/PH/SL/SH/K/<null>

<c-ext>               ::=  F1F0/F2F1/F3F2/F4F3/F5F4/F6F5/F7F6/PLF7
                      /    PHPL/SHSL/KSH/DK/F0D

<aux-reg>             ::=  <five bit value>


<rw>                  ::=  ,R/,W1/W2

<carry>               ::=  ,0/,1

<rw and/or carry>     ::=  <rw>  [<carry>]
                      /    <carry>  [<rw>]

<rw control>          ::=  ,RCM/,WCM
```

EXAMPLES:

```
SR,W1    F0            -- write register F0 to data memory and do a
                          subroutine return.

,W1      23,,F0        -- write a HEX(23) to data memory and copy this
                          same value to register F0.

BEQHH    75,F2,LAB1    -- branch to "LAB1" if the high nibble of register
                          F2 equals HEX(7).

MV       F0,F1         -- copy register F0 to register F1.
```

8-BIT DATA FORMAT

```
<data line>  ::=  [<symbol>]  <delimiter>  <data>  <delimiter>  <comment>

<data>       ::=  [<symbol>]  DC  <delimiter>  <value>  [<value>]^n

<value>      ::=  <hexdigit>  [<hexdigit>]^h*

             /    "  <character>  [<charactor>]^n  "

             /    (  <expression>  )
```

*note that "h" must be an odd integer.

EXAMPLES:

DC	81BC0A	--	defines a 3-byte constant; each byte represented by two hex digits.
DC	"ABCD"	--	defines a 4-byte constant whose value is the ASCII representation of the string "ABCD".
DC	(TAG+3)	--	defines a 2-byte constant whose value is the current value of 'TAG' + 3.
DC	04"STEP"	--	defines a 5-byte value.

MISCELLANEOUS

```
<nibble>          ::=  <expression>

<byte>            ::=  <expression>

<word>            ::=  <expression>

<address>         ::=  <expression>

<five bit value>  ::=  <expression>

<expression>      ::=  <term>
                  /    <expression>  +  <term>
                  /    <expression>  -  <term>

<term>            ::=  <digit> [<hexstring>]
                  /    *
                  /    <symbol>
                  /    C'<character>'
                  /    X'<hexstring>'

<symbol>          ::=  <letter>  [<letter>/<digit>]^7

<delimiter>       ::=  tab

<comment>         ::=  [<character>]^n

<hexstring>       ::=  <hexdigit>  [<hexdigit>]^n

<hexdigit>        ::=  <digit>
                  /    A/.../F

<letter>          ::=  A/.../Z

<digit>           ::=  0/.../9

<null>            ::=
```

ASSEMBLER ERROR CODES

ERRORS:

A	=	invalid A-bus specification.
B	=	invalid B-bus specification.
C	=	invalid C-bus specification.
D	=	multiple LIMITS statement.
E	=	too many operands.
I	=	illegal immediate value.
K	=	invalid CIO operand.
L	=	origin lower than address of last instruction plus one.
M	=	multiply-defined symbol.
M	=	invalid R/W field for SR.
N	=	name required.
O	=	illegal opcode field.
P	=	out of page or out of block branch.
P	=	invalid HEX codes.
Q	=	name not allowed.
R	=	illegal read/write/carry specification.
S	=	invalid HEX on 'INSTR'.
T	=	improper or too many file names.
U	=	undefined symbol referenced.
V	=	illegal value.
X	=	invalid AUX register specification.

WARNINGS:

1	=	A bus	Non-extended register
2	=	B bus	mnemonics used with
4	=	C bus	an extended instruction.

MICRO INSTRUCTION CODES

MNEMONIC	SKELETON CODE	MNEMONIC	SKELETON CODE
AC	180000	ACI	380000
ACX	1A0000	AI	2C0000
AND	080000	ANDI	280000
ANDX	0A0000	B	5C0000
BEQH	740000	BEQHH	740000
BEQHL	740000	BEQL	700000
BEQLH	700000	BEQLL	700000
BER	500000	BFH	6C0000
BFHH	6C0000	BFHL	6C0000
BFL	680000	BFLH	680000
BFLL	680000	BLER	480000
BLERX	4C0000	BLR	400000
BLRX	440000	BNEHH	7C0000
BNEH	7C0000	BNEHL	7C0000
BNELH	780000	BNEL	780000
BNELL	780000	BNR	580000
BTHH	640000	BTH	640000
BTHL	640000	BTLH	600000
BTL	600000	BTLL	600000
CIO	178000	DAC	100000
DACI	300000	DACX	120000
DSC	140000	DSCI	340000
DSCX	160000	INSTR	000000
LPI	190000	MHH	1CC000
MHHX	1EC000	MHL	1C8000
MHLX	1E8000	MIH	3C8000
MIHH	3C8000	MIHL	3C8000
MIL	3C0000	MILH	3C0000
MILL	3C0000	MLH	1C4000
MLHX	1E4000	MLL	1C0000
MLLX	1E0000	MV	200000
MVI	20000F	MVX	0200E0
NOP	200000	OR	000000
ORI	200000	ORX	020000
SB	540000	SC	0C0000
SCX	0E0000	SHFT	104000
SHFTX	124000	SR	078000
TAP	0B8000	TPA	018000
TPA+1	018200	TPA+2	018400
TPA+3	018600	TPA-1	01C200
TPA-2	01C400	TPA-3	01C600
TPS	058000	TPS+1	058200
TPS+2	058400	TPS+3	058600
TPS-1	05C200	TPS-2	05C400
TPS-3	05C600	XOR	040000
XORI	240000	XORX	060000
XPA	038000	XPA+1	038200
XPA+2	038400	XPA+3	038600
XPA-1	03C200	XPA-2	03C400
XPA-3	03C600		

INSTRUCTION SUMMARY

AC[X] -- Binary add with carry.
ACI -- Binary add with carry immediate.
AI -- Binary add immediate.
AND[X] -- Logical and.
ANDI -- Logical and immediate.
B -- Branch.

BEQ {HH, HL, H, L, LH, LL} -- Branch equal immediate.

BER -- Branch equal.

BF {HH, HL, H, L, LH, LL} -- Branch false.

BLER[X] -- Branch less than or equal.
BLR[X] -- Branch less than.

BNE {HH, HL, H, L, LH, LL} -- Branch not equal immediate.

BNR -- Branch not equal.

BT {HH, HL, H, L, LH, LL} -- Branch true.

CIO -- Control I/O.
DAC[X] -- Decimal add with carry.
DACI -- Decimal add with carry immediate.
DSC[X] -- Decimal subtract with carry.
DSCI -- Decimal subtract with carry immediate.
LPI -- Load PC's immediate.

M {HH, HL, LH, LL} [X] -- Binary multiply.

MI {HH, HL, H, L, LH, LL} -- Binary multiply immediate.

MV[X] -- Move.
MVI -- Move immediate.
NOP -- No operation.
OR[X] -- Logical or.
ORI -- Logical or immediate.
SB -- Subroutine branch.
SC[X] -- Binary subtract with carry.

SH {HH, HL, LH, LL} [X] -- Shift.

SR -- Subroutine return.
TAP -- Transfer auxiliary to PC's.

TPA [± 1, ± 2, ± 3] -- Transfer PC's to auxiliary.

TPS [± 1, ± 2, ± 3] -- Transfer PC's to stack.

TSP -- Transfer stack to PC's.
XOR[X] -- Logical exclusive or.
XORI -- Logical exclusive or immediate.

XPA [± 1, ± 2, ± 3] -- Exchange PC's to auxiliary.

SPECIAL NOTATION

OPCODE SUFFIXES:

H = high 4-bits of register.
L = low 4-bits of register.

HH = high 4-bits of A and B.
HL = high 4-bits of B, low 4-bits of A.
LH = low 4-bits of B, high 4-bits of A.
LL = low 4-bits of A and B.

X = extended operation.

PARAMETER FIELD MNEMONICS:

R = read.
W1 = write 1.
W2 = write 2.
RCM = read control memory.
WCM = write control memory.

0 = set carry to 0.
1 = set carry to 1.

WANG LABORATORIES, INC.

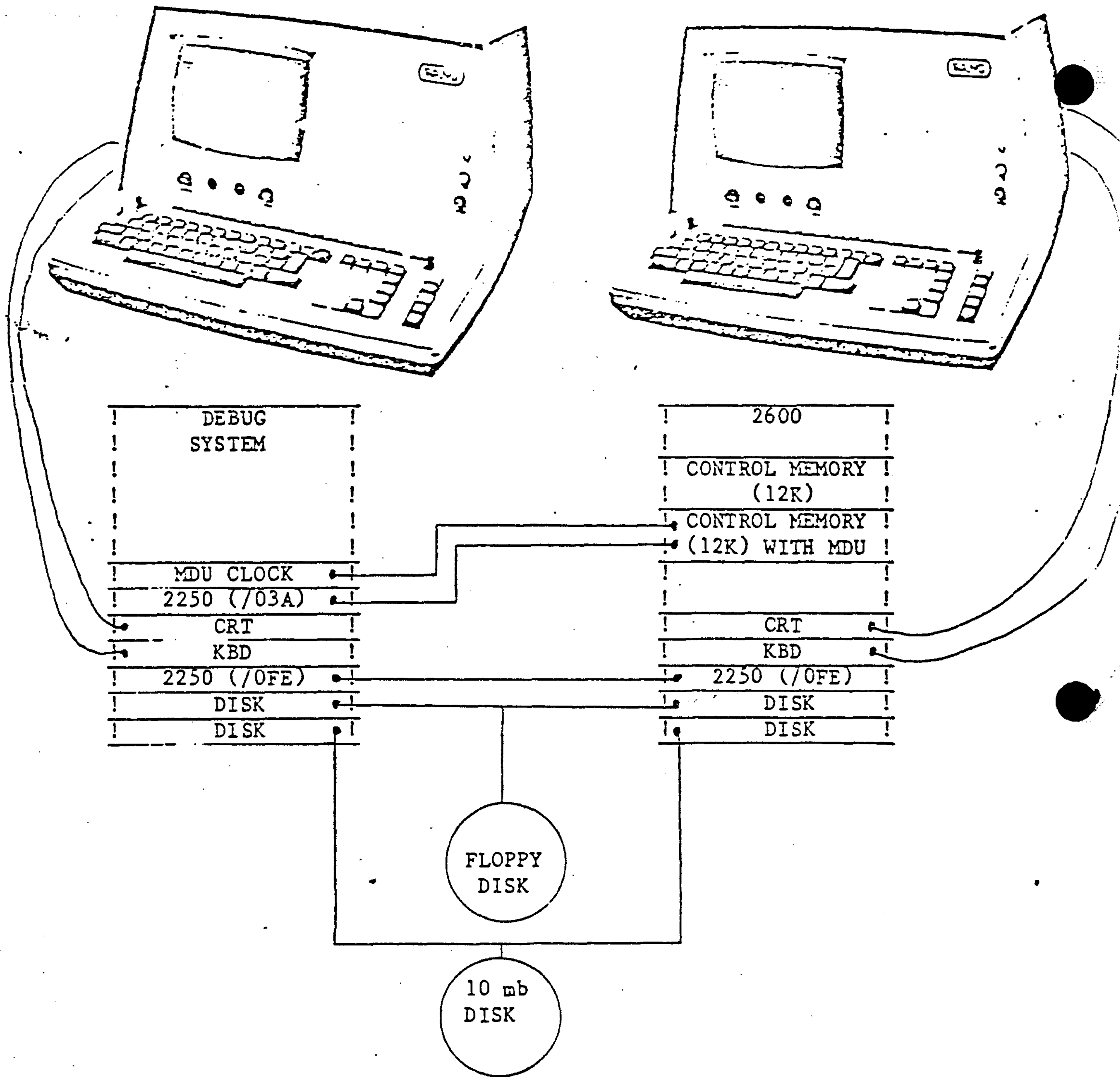
2600 MICROCODE DEVELOPMENT SYSTEM
January 11, 1977
Revised October 21, 1979

I. SYSTEM DESCRIPTION

The 2600 MDS is a combination of hardware and software that provides the user with convenient 2600 microcode debug capability. Registers, data, and instructions within the 2600 can be examined and modified. Execution can be started and stopped at specified instructions or single stepped, and register dumps can be performed at specified instructions.

The 2600 MDS hardware consists of a debug system coupled to a modified 2600. The debug system is a standard 64K 2200VP or 2200MVP with a 56K partition, a terminal with a 24x80 CRT, and two 2250's. The working 2600 is a standard 2600 with CRT, keyboard, disk, 2250 and at least 24K of control memory. One of the control memory boards is replaced by the MDU (microcode development unit) board and a jumper on the 2600 motherboard is added for disabling control memory. Typically, the 2200's are multiplexed to a floppy disk and to a 10 MB disk.

The debug 2200 may, optionally be equiped with an MDU clock, which is used for execution timing. The MDU clock is a 2228B controller loaded with a timer microprogram. The MDU clock is connected to the MDU board in the 2600.


DEBUG SYSTEM
WORKING 2600
DEBUG SYSTEM
MDU CLOCK
2250 (/03A)
CRT
KBD
2250 (/0FE)
DISK
DISK
2600
CONTROL MEMORY (12K)
CONTROL MEMORY (12K) WITH MDU
CRT
KBD
2250 (/0FE)
DISK
DISK
FLOPPY DISK
10 mb DISK

The 2600 has two states of operation. It powers up in the first, or RUN state. Executing instructions in PROM automatically puts the 2600 into RUN state. In this state the machine is executing standard microcode, as a normal 2600. After the MDU has "halted" the CPU, the machine is in the DEBUG state, and is executing custom microcode, which is communicating with the debug 2200 via 2250's. The debug microcode resides at the high end of memory (5E00 - 5FFF).

The BASIC program, "2600MDU", in the debug 2200 displays debug information and allows operator interaction. When the system returns to RUN state, all registers are first restored to their previous values via the debug microcode, then control is passed back to the standard microcode. Only microcode resident in this memory may be stepped and debugged (PROM may not be stepped).

A Content Addressable Memory (CAM) containing eight 16-bit words resides in the MDU. On power up, the outputs of this CAM are inhibited. Writing to this memory is done via the 2250 from the debug 2200 using the following sequence:

WR, CBS '00', OBS 'WX', OBS 'YZ' repeated 8 times

where:

> WX = high 8-bits of address
> YZ = low 8-bits of address

After the CAM has been loaded, the debug 2200 may enable the CAM with a CBS of 10. It may also be disabled again with a CBS of 20. If the CAM is enabled, and the system is in the RUN state, the MDU continually monitors the accesses to the control memory. Whenever a match is found between the IC's and one of the CAM locations, a "halt" is initiated.

A CBS of 30 (from the debug 2200 to the MDU) will also create a "halt" condition, if the machine is in RUN state.

Upon any "halt" condition, the MDU blocks the instruction from the RAM CM, and substitutes a SB to the debug microcode (switch selectable) in its place. This places the machine in DEBUG state, and the custom microcode communicates with the BASIC program in the debug 2200. Once the machine is in DEBUG state, subsequent "halt" conditions compare from CAM, CBS 30 from debug 2200 are ignored.

Two new instructions, no-ops to the CPU, are interpreted by the MDU as special commands:

> CIOC -- returns the machine to RUN state after 16 cycles of delay.
>
> CIOS -- generates a new HALT after 16 cycles of delay.

CIOC and CIOS are ignored by the system whenever CAM is disabled.

II. DEBUG MICROCODE FUNCTIONS

The 2600 has two states of operation. It will power up in RUN state. In this state, the machine is executing standard microcode, as a normal 2600. After the MDU has "halted" the CPU, the machine is in the DEBUG state and is executing custom debug microcode, which is communicating with the debug 2200 via 2250's. The debug microcode performs the functions listed below:

1. HALT (enter DEBUG mode) -- interrupts the 2200T and sends vital registers and current IC's.

2. STEP -- instruct the MDU to execute one 2600 microinstruction.

3. GO -- instruct the MDU to continue 2600 execution.

4. XR (examine registers) -- send registers and stack values to debug 2200.

5. RR (restore registers) -- receive register values from debug 2200.

6. XD (examine data) -- read 2600 data memory and send it to debug 2200.

7. CD (change data) -- change 2600 data memory.

8. ID (initialize data) -- initialize 2600 data memory to a specified value.

9. XI (examine instruction) -- read an instruction from control memory and send it to debug 2200.

10. CI (change instruction) -- change an instruction in 2600 control memory.

11. II (initialize instruction) -- initialize control memory to specified instruction.

III. DEBUG INTERFACE CONNECTORS

The communications channel necessary to implement the above functions consists of two standard 2250's (one in the debug 2200, one in the 2600) with a cable wired as follows:

```
OB1 -- OB8↔IB1 -- IB8
OBS       ↔IBS
RBI       ↔CPB
COB1      ↔ENDI
```

The cable is symmetric, so the reverse channel looks the same.

A command sequence consists of:

```
CBS 01
OBS XX -- (command)
CBS 00
```

followed by one or more OBS/IBS as needed.

IV. OPERATING INSTRUCTIONS

1. Load 2600 with the debug microcode (@MDU).

2. Load debug 2200 with "2600MDU".

   ```
   :CLEAR
   :SELECT DISK /xyy
   :LOAD RUN "2600MDU"
   ```

3. Pressing STEP ('15) on the debug 2200 keyboard will halt the 2600 placing it in debug state. "2600MDU" will display the registers and await a debug command.

V. DEBUG COMMANDS

When the 2600 is in RUN state, the debug 2200 displays "2600 EXECUTING...". The only command accepted on the debug 2200 is:

STEP (key '15) -- halt 2600

The 2600 enters DEBUG state and transmits the current values of the registers, etc, to the debug 2200. The debug 2200 displays the register values and waits for a debug command from the operator. The display looks as follows:

```
-XR-                                                              0.00 MS.
LAST   0000 - OR          F0,F0,F0          800000
NEXT   0001 - OR          F0,F0,F0          800000
                                                                  BREAKPTS
 K SH SL CH CL PH PL F7 F6 F5 F4 F3 F2 F1 F0
00 02 00 00 00 00 00 00 00 00 00 00 00 00 00            STACK
                                                         8269
AUX 00-07   0000 0000 0000 0000 0000 0000 0000 0000      6660
AUX 08-0F   0000 0000 0000 0000 0000 0000 0000 0000      554F
AUX 10-17   0000 0000 0000 0000 0000 0000 0000 0000      443E
AUX 18-1F   0000 0000 0000 0000 0000 0000 0000 0000      332D

0000 - 00000000 00000000 00000000 00000000        ................
```

Debug command?

The last instruction executed, next instruction to be executed, register values, the top few levels of the hardware subroutine stack, and 16 bytes of data memory are displayed. Execution time is displayed if the system is equiped with an MDU clock.

The following commands are then allowed:

1. XR (key '0) -- Examine Registers

   Displays the current contents of all the 2600 registers, the last and next instruction to be executed, and 16 bytes of data memory starting at the current value of the high 12-bit of the PC's.

2. CR (key '1) -- Change Registers

   Changes the contents of the specified registers to the specified values. Eight bit registers are specified by their mnemonic names (i.e., K, SH, SL, PH, PL, CH, CL, F0, F1, ...F7); aux registers are specified by 1 or 2 hexdigits (0 - 1F).

3. ZR (key '17) -- Zero Registers

   All 8-bit and aux registers are set to zero, except for SH which is set to 02_{16} (CRB = busy).

4. XD (key '2) -- Examine (Change) Data

Sixteen bytes of data memory from the specified starting address are displayed in both hexadecimal and ASCII. Pressing CR causes the next 16 bytes to be displayed. Data can be changed by entering EDIT mode, positioning the cursor, typing in the new data, and pressing CR.

5. DD (key '3) -- Define Data

The user can define sections of data memory to be displayed whenever XR is performed. The name, address and length of the data area are entered after pressing DD.

6. ID (key '19) -- Initialize Data

Sets each byte of memory from the specified starting address through the specified ending address to a specified value.

7. LI (key '6) -- List Instructions

The instructions from the specified starting address are displayed in mnemonic and hexadecimal form. Instructions are displayed in sections; press CR for next section.

8. EI (key '22) -- Enter Instructions

Change the contents of instruction memory starting at the specified address by the instructions specified in mnemonic format. EI displays the old instruction after the address of the next instruction to be entered. Entering a null line (CR only) skips the current instruction (instructions is not modified). EI is terminated by pressing another debug special function key. If an entered line is syntactically incorrect, it will not be entered and must be retyped.

See "2200VP Resident 2600 Assembler" for a detailed description of instruction mnemonics.

9. II (key '23) -- Initialize Instruction

Change the contents of control memory starting at the specified address through the specified ending address to a specified instruction.

10. VD (key '7) -- Verify to Disk

The contents of the specified disk file is compared against the current contents of 2600 instruction and/or data memory. Any differences are displayed.

11. LD (key '8) -- Load from Disk

The contents of the specified disk file are transferred into 2600 instruction and/or data memory.

12. SD (key '24) -- Save on Disk

The current contents of 2600 instruction and/or data memory (or portion thereof) are stored in the specified disk file.

13. TR (key '11) -- Trace On

Insert a Trace-On breakpoint at the specified location. Before the execution of the instruction at the specified location, trace mode is on. The 2600 registers will be displayed (same format as BH) after each instruction is executed until simulation terminates or a TO breakpoint is encountered. TR may be turned on immediately.

14. TO (key '27) -- Trace Off

Insert a Trace-Off breakpoint at the specified location. Before the execution of the instruction at the specified location, the trace mode will be turned off. Trace may be turned off immediately.

15. BH (key '12) -- Breakpoint Halt

Insert a Breakpoint Halt at the specified location. Execution will terminate before the execution of the instruction located at the specified address. When the termination occurs, a message will be displayed indicating the Breakpoint Halt followed by a display of the registers.

16. BC (key '28) -- Breakpoint Continue

Same as Breakpoint Halt (BH) except after the display of the registers, simulation continues at the next instruction.

17. BR (key '27) -- Breakpoint Remove

Remove all or a specified breakpoint.

18. IC (key '30) -- Set Instruction Counter

Set the IC's to a specified value. 2600 execution will continue at this address when GO, STEP, or STEP+1 is pressed.

19. GO (key '31) -- Continue Execution

Pressing '31 after 2600 execution has been halted causes execution to continue at the next instruction (i.e., current value of IC's displayed).

20. STEP (key '15) -- Step Execution

Pressing '15 after 2600 execution has been halted causes the next instruction to be executed after which execution halts.

21. STEP+1 (key '14) -- Subroutine Step

STEP+1 functions the same as STEP except that if the instruction to be executed is a subroutine branch, SB, the entire subroutine is executed before execution is halted.

22. PRINT (key '16) -- PRINT

Causes the output from the next command to be printed (/215) rather than displayed on the CRT.

23. ZC (key '9) -- Zero Clock

Zeroes the MDU clock.

24. CT (key '10) -- Clock On

Insert a Clock On breakpoint at the specified location. Before execution of the instruction at the specified location, the MDU clock is turned on. The clock may be turned on immediately.

25. CO (key '27) -- Clock Off

Same as CT except clock is turned Off instead of On.

26. CC (key '4) -- Calculate Checksums

Calculate checksums on control memory and data memory.

WANG LABORATORIES, INC.

MEMORANDUM

TO: File

FROM: Bruce Patterson

DATE: March 26, 1980

SUBJECT: 2200 Development Clock

Function: 2200 Development Clock is an event timer that can be triggered under program control or by an external hardware event.

Hardware: 2228B or 2228C with timer PROM (chip file @BPCLOCK), installed. Controller address is /0FD.

The timer PROM is a simple counter program coupled with a command decoder which interfaces with the 2200 or an external event probe. (Pin xx on cable connector).

Resolution: Approximately ± 50 usec.

Calibration: The tick count can be converted to real time by multiplying the count by the calibration factor. Determine the calibration factor by allowing the clock to execute for several hours; then, calibration factor is the actual time divided by the tick count. The calibration factor is a function of the clock board, not the CPU in which the clock is installed.

Commands: The timer program responds to the following commands.

- Zero clock -- $GIO (4400)
- Clock On -- $GIO (4401)
- Clock Off -- $GIO (4402)
- Read clock -- $GIO (4403 C620) T$ (4 byte binary count)
- Enable external probe -- $GIO (4404)
- Disable external probe -- $GIO (4405)
- Reset Clock Board -- $GIO (4508)

ONE INDUSTRIAL AVENUE LOWELL MASSACHUSETTS 01851 • TEL (617) 851-4111 • TWX 710-343-6769 • TELEX 94-7421

Utilities: BPCLOCK -- activate clock under keyboard control. Provides sample clock access subroutines.
EVENTIME -- time specified program event.

OS Activation: The 2200MVP OS (Release 1.9 or later) can be set up to keep track of CPU execution time for a given partition or for all partitions. The OS will turn clock on while the specified partition(s) is executing. An additional $INIT parameter is used to instruct the OS to access the clock.

$INIT (A$, T$, C$, P$(), D$(), P$, HEX(ab))

ab = 10 if all execution time is to be measured.
ab = 3x if execution time of partition (X+1) is to be measured.

The utility BPCLOCK can be used to obtain execution time. For example, if partition 3 is to be timed, BPCLOCK can be loaded into a different partition with a different terminal. Zero clock count, perform event in partition 3, read clock. Often, it is useful to sample the execution time of a partition by zeroing the clock count and then reading the clock after some known actual time period, in order to determine the partitions actual load on the CPU. Time spent waiting for I/O devices is not counted as execution time.

MDU: The clock can be attached to the MDU board with a standard modem cable (external clock probe). Execution time between breakpoints will be measured.