This PDF contains the contents of a folder from the Wang 2200 development group, labeled

2600 DEVELOPMENT TOOLS

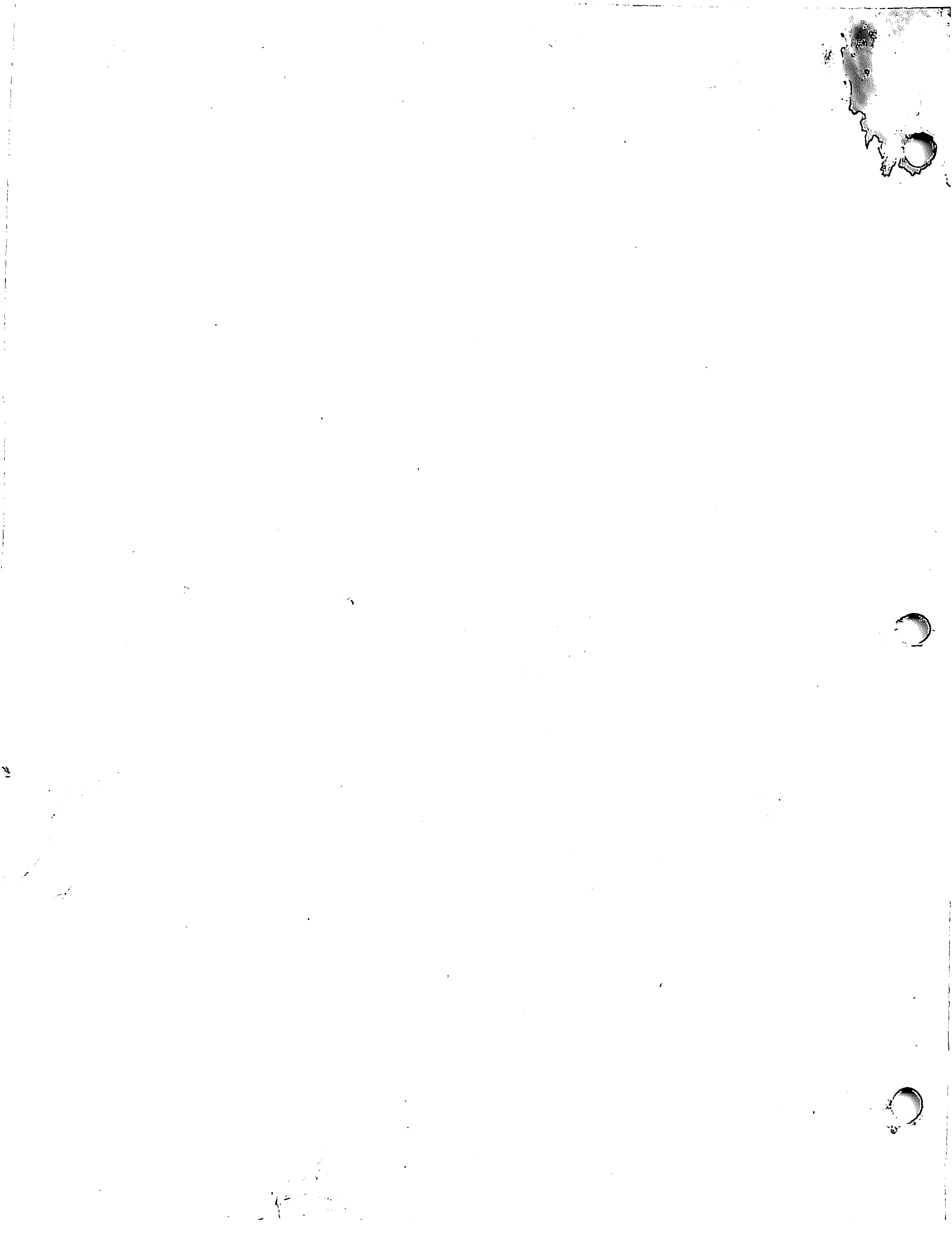
It is an assortment of specifications and some handwritten notes.

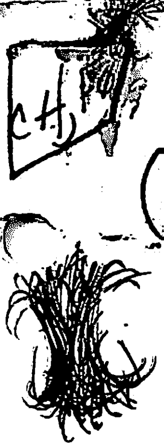
BASBOL

MICROPROGRAMMING

Revised March 24, 1981

Bruce Patterson





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 execution high-level-language (HLL) architecture. With direct execution HLL 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 the 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.

O.S. + language Interpreter = control storage memory.

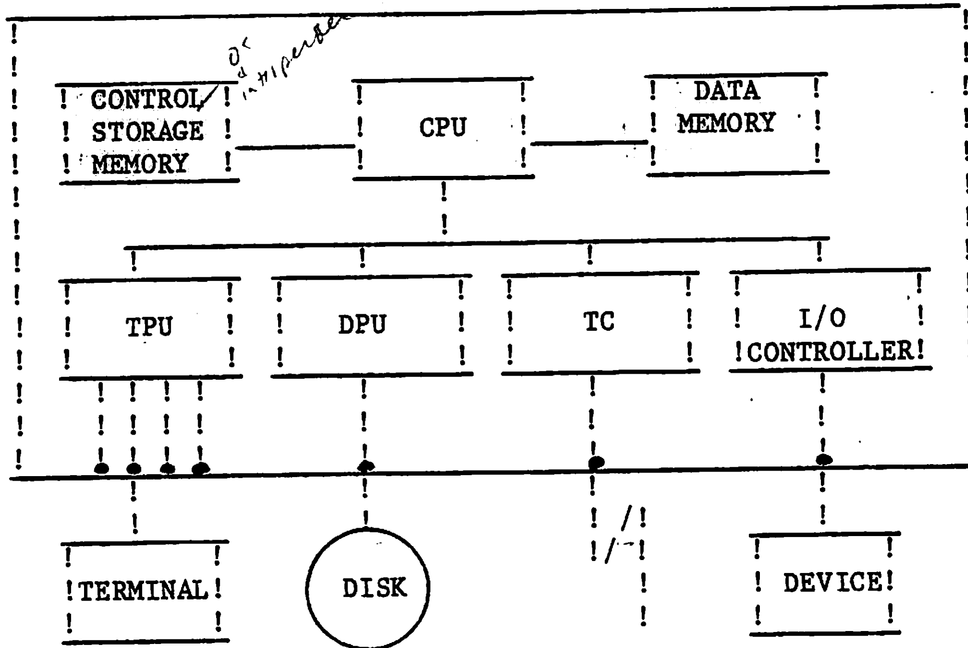


Figure 1 2200 System Block Diagram

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.

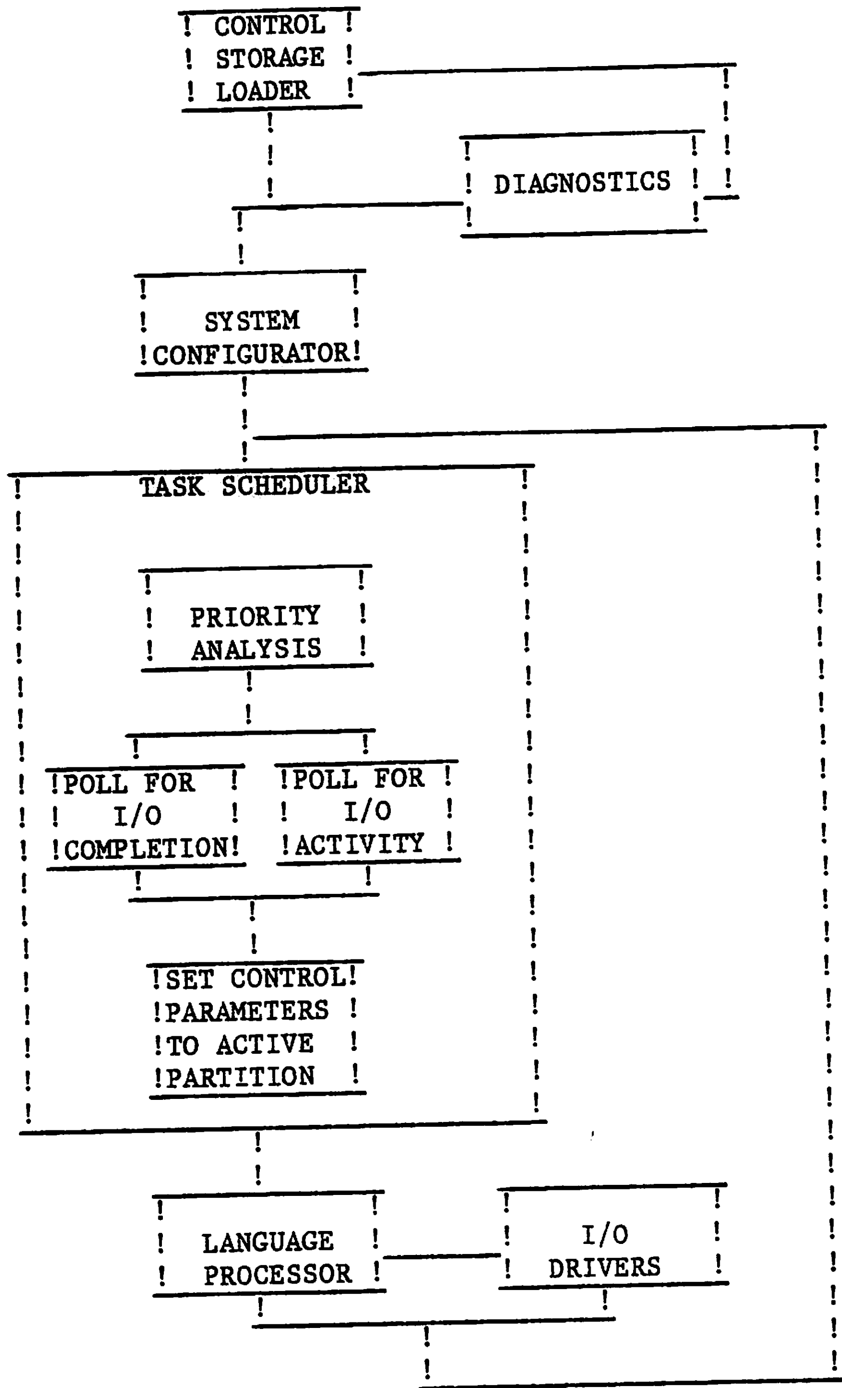


Figure 2 Block Diagram of 2200 MVP OS

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.

4. DOCUMENTATION

Careful documentation is a requirement of all BASBOL system software. It is essential for system maintenance and will encourage complete, well thought out design with well defined interfaces. Most of the module documentation should be included within the source listings; however, it is acceptable to provide supplementary narratives and diagrams in memo form. Documentation should be structured, providing first a quite general view of the project followed by progressively more detailed levels of description. The following approach should be used when documenting a logical software module.

a. Module documentation

1. Introduction

   Should present an overview of the function and scope of the naive reader. This is the starting point for understanding what the module does and how it performs that function.

2. Completion Report

   Documentation should include a section describing what portion of the project has currently been implemented. For BASIC-3, specify all implementation exceptions and extensions to BASIC-2. For COBOL, include all exceptions and extensions to ANSI and VS COBOL.

3. Processing

   This section should describe how the software performs the specified function. The explanation should describe what happens in each phase (entry, resolution, and execution) of processing general flow diagrams should be included to clarify the process flow where necessary.

4. Data Structures

   Descriptions and diagrams explaining the function, relationships, and data structures used should be provided.

5. Register and Data Memory Use

   Register and data memory use common throughout the module should be described. Include equates for AUX registers and data memory locations.

c. Routine Section Comments

A routine generally should be decomposed into logical sections. Each section should be separated and started with a description of the function of that section.

For example, the following routine shifts a string to the right or left.

```
*%= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

*     Describe whether to shift left or right.

      LPI,R       START               get start address
      BLRX        CHCL,FIFO,LEFT      branch if start  dest.

*%= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

*     LEFT: shift string to left.  Now we start at left end of
            string.

  LEFT      XPA + 1,R,        ,0
```

It is often convenient to include checkpoint information at major section headers. Current register contents are particularly useful for program checkout. For example, the following logic is part of the MOVE statement:

```
*%===SETUP======
* Setup parameters for move
* routine

     MVX F3F2, PHPL

*%===PERFORM MOVE====
* At this point the following
* registers are setup:
*
*      AUX 3 = address of description A
*      AUX 4 = address of description B
*      AUX 5 = offset to A
*      AUX 6 = offset to B

        SB    MOVE COB    MOVE DATA
```

d. Source Line Comments

Nearly every source line should be commented as to its logical function. Avoid describing physically what the instruction does, for example,

```
     MV    CH,FO       load FO with CH
```

e. Blank Lines

Blank lines should be used freely to improve code readability. Separate logical units of code with blank lines.

f. Tags

Tags within a particular routine should all begin with the same prefix, to avoid conflicts with other programmers. Tags internally referenced should have a numeric suffix. Tags externally referencable (i.e., entry points) should have an alphabetic suffix.

5. NOTATION CONVENTIONS:

a. byte = logical 8-bit group

b. bit notation:

1. in a byte,

X	X	X	X	X	X	X	X
80	40	20	10	08	04	02	01
				8	4	2	1

2. to reference more than 1 bit in a byte, the HEX value of the combined bits can be used. For example,

   bits C0 = bit 80 and bit 40
   bits 05 = bit 04 and bit 01
   bits A2 = bit 80, bit bit 20 and bit 02

3. in multibyte values (e.g., AUX's), the HEX value of the bits is used. For example,

   bit 0001 = low order bit
   bits 000F = low 4 bits
   bits FF00 - high byte of value

6. REGISTER USE

Auxiliary registers 10-1F are reserved for system pointers and flags. They are referenced by name only (e.g., VSPTR) to improve code readability. AUX registers 00-0F are for general use. By convention, subroutines should use the lowest numbered registers possible. Hopefully, this will conserve the register use and minimize conflicts.

The following list specifies which AUX's are available during the various phases of processing.

Entry phase: AUX 0-7

Resolution phase:
   COBOL
      IDENTIFICATION DIVISION: 0-7
      ENVIRONMENT DIVISION: 0-7
      DATA DIVISION: 0-F
      PROCEDURE DIVISION:
         statements: 0-7
         global logic: 8-F

BASIC-2

statements: 0-7

global logic: 8-F

Execution phase: 0-F

The lower level routines should use lower numbered AUX's. For example, in COBOL the arithmetic and move primitives use AUX registers 0-6.

Lowest numbered file registers should be used whenever possible. Typically, entry and return parameters are passed through low numbered file registers. When 16 bit values are stored in a file registers pair, the register with the higher address contains the high byte of the value.

The low 2-bits of register SL specify the processing phase. The following equates should be used whenever phase checking is done:

```
EPHASE EQU FF-03        entry phase (00)
RPHASE EQU 01           resolution phase (01)
XPHASE EQU 02           execution phase (02)
```

For example:

```
BFL   XPHASE,SL,TAG1             branch if not exection
BTL   RPHASE,SL,TAG2             branch if resolution
BFL   RPHASE+XPHASE,SL, TAG3     branch if entry
```

7. BASBOL Base Code

The MVP BASIC-2 Release 2.1 microprogram provides the base code for BASBOL development. The base code resides at 0000-5FFF and will be changed infrequently, perhaps once a month, to incorporate developed code. Each microprogrammer will be assigned his own 2280 platter which will contain the base code external symbol table files, the base object code, and space for microcode development. The base source code should be modified at the prescribed update times. During development, the base code can be patched or routines can be duplicated in the development space and then modified.

The base object code is contained in the files @BAS.

8. CONTROL MEMORY MAP

0000-5DFF BASE CODE
5E00-5FFF MDU
6000-7FFF NEW CODE

Current development units are limited to 32K; however, 64K units are under development.

9. DEVELOPMENT SYSTEMS

Four MDU systems are available for debugging microcode. A floppy disk (/310 or /320) and a sort to the band printer (/216) is available on each system.

Three additional 256K MVP systems are used for monitoring the MDU systems, editing, and assembling.

- System #1 is used for editing, assembling, and 2280 backup to tape. Assembling is done via terminals in the lab. The following perpherals are available:

2270A	(/310)	floppy disk
2260BC-2	(/320)	10 mb disk
2280-3	(/D30-/D-75)	Pair of 80 mb disks
2263	(/215)	chain printer
2273	(/216)	band printer
2209A	(/07B)	tape drive

- System #2 monitors MDU #1 and MDU #2 and provides partitions for editing. The following perpherals are available:

2270A	(/310)	floppy disk, shared with MDU #1
2270A	(/320)	floppy disk, shared with MDU #2
2280-3	(/D30-/D75)	pair of 80 mb disks
2273	(/216)	band printer

- System #3 monitors MDU #3 and MDU #4 and provides partitions for editing. The following perpherals are available:

2270A	(/310)	floppy, shared with MDU #3
2270A	(/320)	floppy, shares with MDU #4
2280-3	(/D30-/D75)	pair of 8 mb disks
2273	(/216)	band printer

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.

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

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.

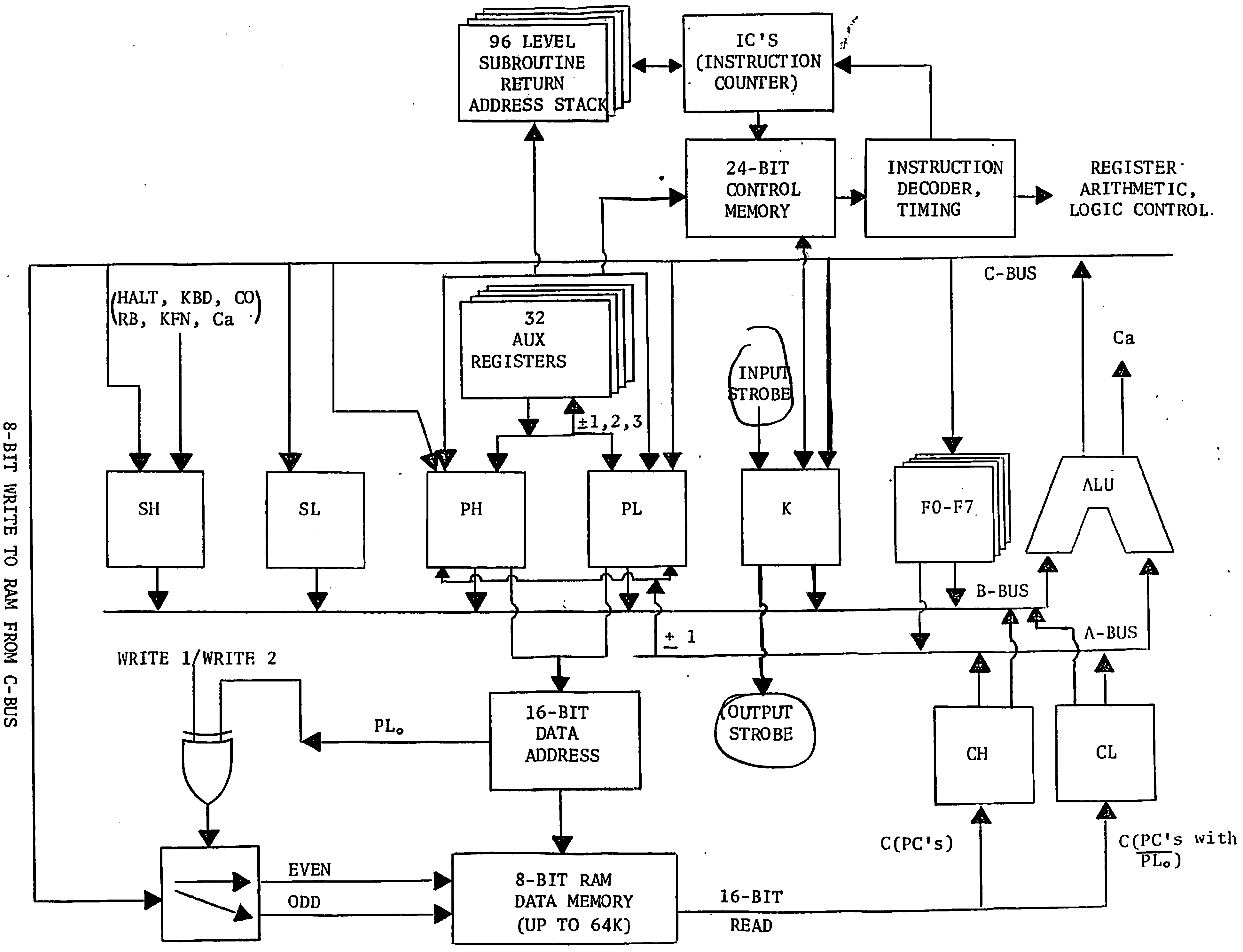
A. Register Structure

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 [illegible]RUCTURE



13

(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:

DP	PE	HALT		RB	KFN	CRB	Ca
7	6	5	4	3	2	1	0

CARRY (BIT 0) (H/M)
0 = NO CARRY
1 = CARRY

CRB (BIT 1) (H/M) (ALIAS KBD)
0 = ALLOW INPUT FROM KBD OR SELECTED DEVICE (I.E., CPU IS READY)
1 = INHIBIT ALL INPUT FROM DEVICES (I.E., CPU BUSY)

KFN (BIT 2) (H/M)
SET TO 1 WHEN INPUT RECEIVED FROM KBD IS SPECIAL FUNCTION CODE. IT IS A 9TH DATA BIT FOR INPUT.

RB (BIT 3) (H)
0 = DEVICE NOT ENABLED OR BUSY
1 = DEVICE ENABLED OR READY

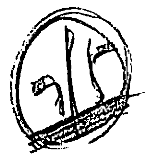
-SPARE- (BIT 4)

HALT (BIT 5) (H/M)
SET TO 1 WHEN HALT/STEP PRESSED ON KBD

PARITY (BIT 6) (H/M)
SET TO 1 WHEN A PARITY ERROR OCCURS ON CONTROL OR DATA MEMORY

DPRTY (BIT 7) (M)
0 = TRAP IF PARITY ERROR
1 = DO NOT TRAP IF PARITY ERROR

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).

D. 2600 INSTRUCTION SET

		22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
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	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	B	B	B	B
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
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
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
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	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	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	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	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	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	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	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	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	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	B
ACI	Binary 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	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	B
III. MINI INSTRUCTIONS		OPCODE									D	D									B-BUS			
TAP	Transfer Aux to PC's	0	0	0	1	0	0	1	1															
TPA	Transfer PC's to Aux	0	0	0	1	0	1	1	1	-	D	D	0	-	-	Ax	Ax	Ax	Ax	Ax	B	B	B	B
XPA	Exchange 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	B
TPS	Transfer PC's to Stack	0	0	0	0	0	1	1	1	±	D	D	0	In	In	Ax	Ax	Ax	Ax	Ax	B	B	B	B
TSP	Transfer Stack to PC's	0	0	0	0	1	0	1	1	±	D	D	0	In	In	-	-	-	-	-	B	B	B	B
SR, RCM	— Read Control Memory + SR	0	0	0	1	1	0	1	1	-	D	D	-	-	-	-	-	-	-	-	B	B	B	B
SR, WCM	— Write Control Memory + SR	0	0	0	0	1	1	1	1	-	-	-	0	1	1	-	-	-	-	-	-	-	-	-
SR	Subroutine Return	0	0	0	0	1	1	1	1	-	-	-	0	1	0	-	-	-	-	-	-	-	-	-
CIO	Control Input/Output	0	0	0	0	1	1	1	1	-	D	D	0	0	-	-	-	-	-	-	B	B	B	B
		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	T	T	1	I	I	D	D	I	I	I	I	I	I	I	I	I	I	I	I
IV. MASK BRANCH INSTRUCTIONS		OPCODE						BRANCH FIELD (LOW 10-Bits)									MASK				B-BUS			
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
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
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	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	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	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	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	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	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
TTTTTT: 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) → $\overline{\text{CH}}$
C(PC's with 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 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. X - Extended Operation Bit

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.

M

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).

± 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.

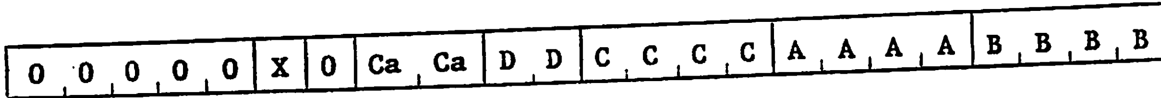
APPENDIX A

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

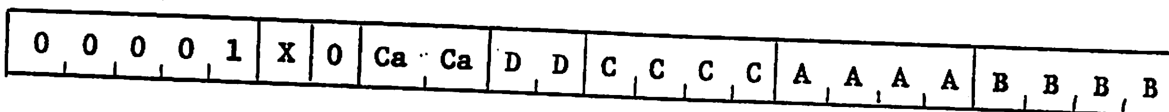
For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

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

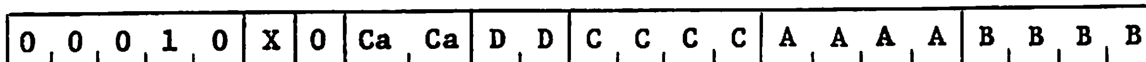
For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

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

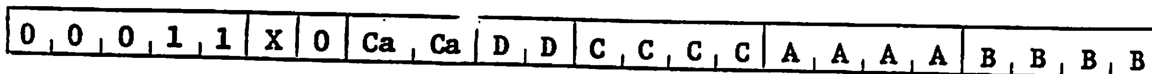
For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

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



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.

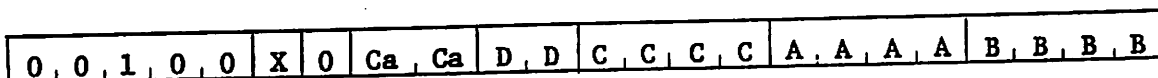
The A field can specify that PC's be incremented or decremented at the end of the instruction.

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:

Dummy field(s)		Result			
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



If X = 0, the 8-bit registers specified by the A and B fields are the last resultant carry (SH0) are added together in decimal. The final sum is stored in the register specified by the C field and SH0 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

0	0	1	0	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 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.

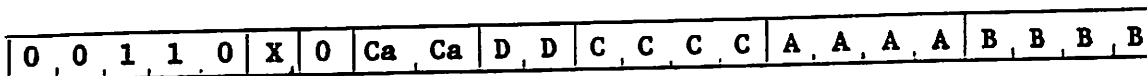
The A field can specify that PC's be incremented or decremented at the end of the instruction.

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



0	0	1	1	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 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

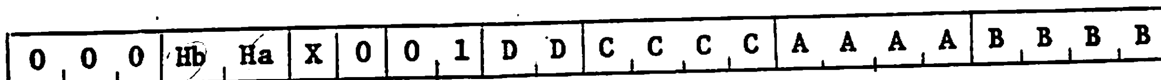
For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

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

05 10

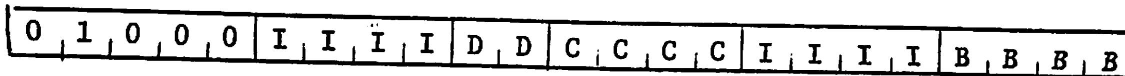
For instructions that modify the PC registers, the read and write address will be the initial contents of the PC registers. If CH or CL is specified in the A or B fields, the previous contents of CH or CL will be used in the instruction.

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:

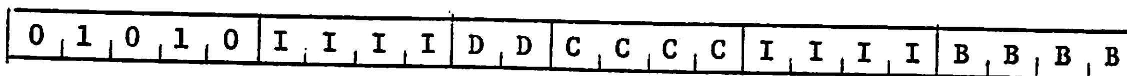
DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

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

ANDI -- AND IMMEDIATE



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:

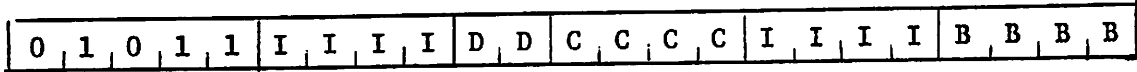
DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

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:

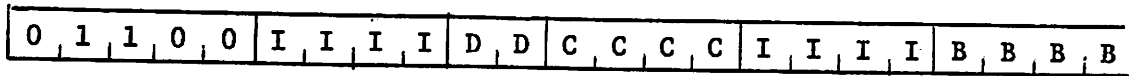
DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

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

0	1	1	0	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 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.

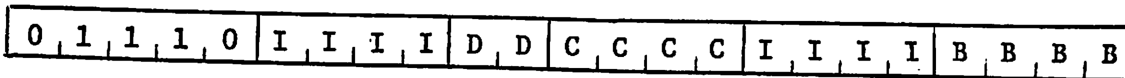
The I and B fields must contain decimal data (0 - 9) or the results are indeterminate.

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



0	1	1	1	0	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 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:

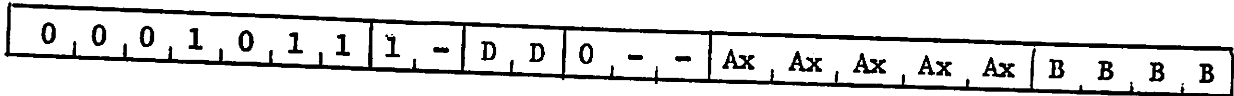
DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

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



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

Register use in the B field:

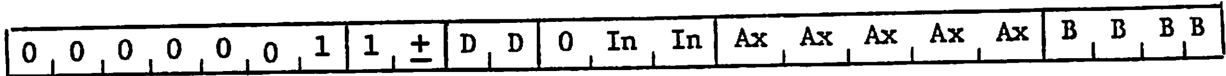
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

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	=	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:

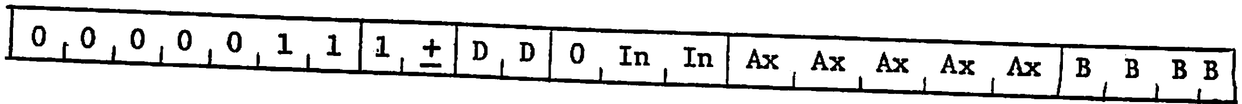
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

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:

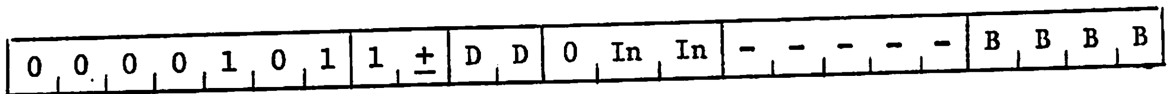
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

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				
± 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:

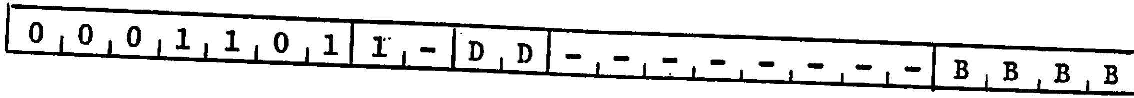
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

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:

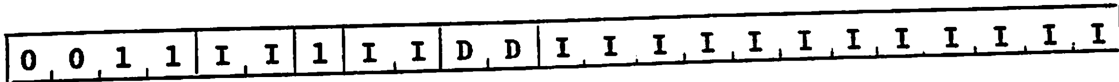
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

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

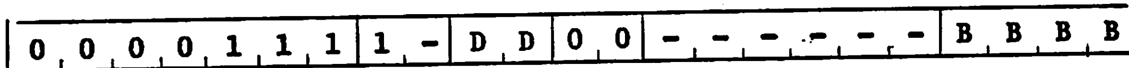


The PC registers are set equal to the 16-bits specified in the I field. If D = 1, data will be read from Data Memory; the read address will be the new contents of the PC's. If a write is specified, the data written will always be 0; the write address will be the new contents of the PC's.

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



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:

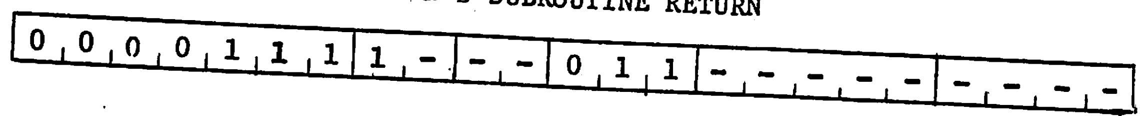
B : F0 - F7, PL, PH, CL, CH, SL, SH, K, dummy

Read/Write options:

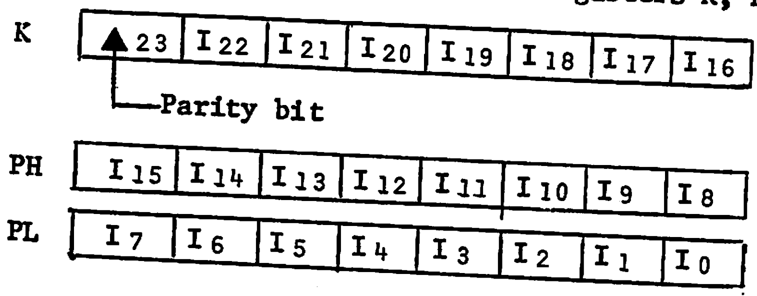
DD = 00: no read or write
DD = 01: read
DD = 10: Write 1
DD = 11: Write 2

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:

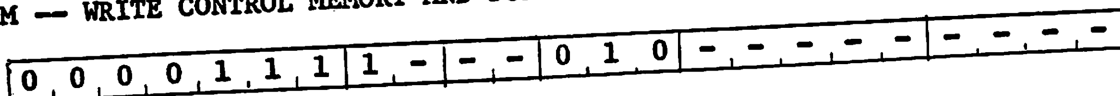


Finally, a normal SR is performed; that is, the next entry in the subroutine stack is removed, and transferred to the IC's (instruction counter). Program execution will continue at that address.

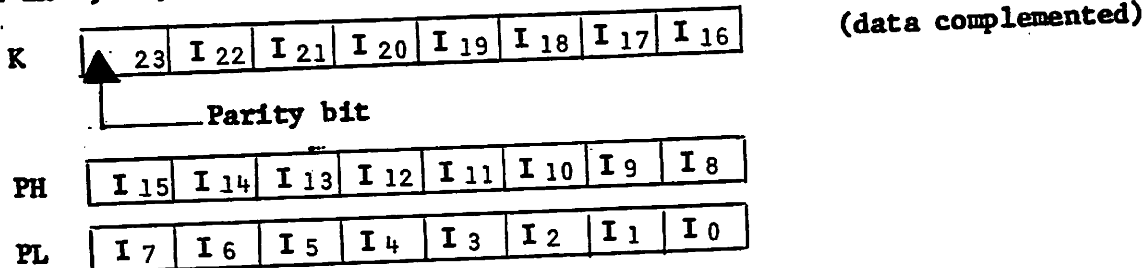
A typical instruction sequence for reading control memory is:

```
        .
        .
        .
        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
```

C IO — 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

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

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.

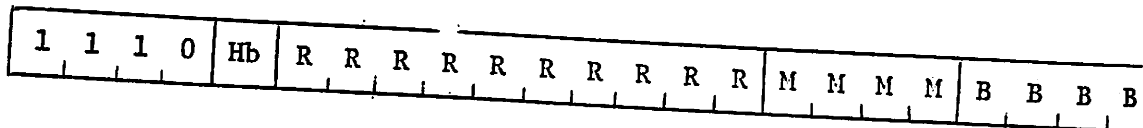
Register use in the B field:

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.

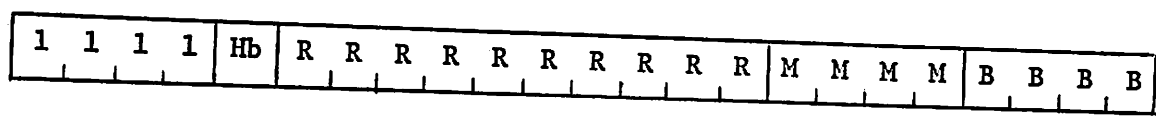
Register use in the B field:

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.

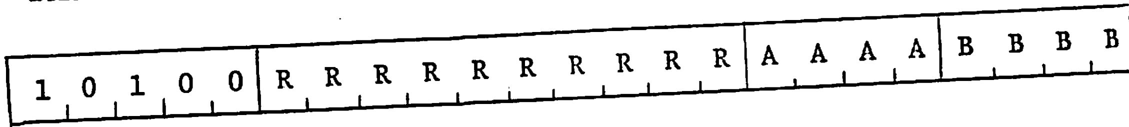
Register use in the B field:

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.

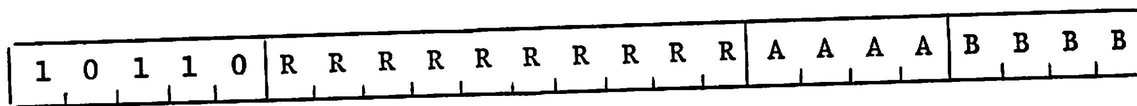
If A (or B) specify the dummy register, 0 is used in the compare.

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.

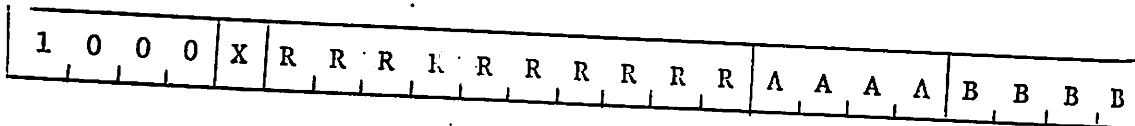
If A (or B) specify the dummy register, 0 is used in the compare.

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.

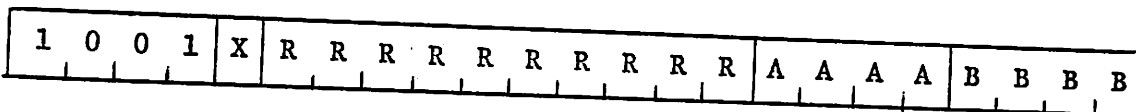
If A (or B) specify the dummy register, 0 is used in the compare.

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

PSEUDO INSTRUCTION FORMATS (ASSEM26)

```
<pseudo> ::= <name>   <delimiter>  ORG    <delimiter> <expression> /
             <symbol> <delimiter>  EQU    <delimiter> <expression> /
                      <delimiter>  MODULE <delimiter> <comment> /
                      <delimiter>  TITLE<delimiter> <comment> /
                      <delimiter>  SPACE<delimiter> <expression> /
                      <delimiter>  EJECT  /
             <name>   <delimiter>  PAGE   /
                      <delimiter>  CONT   <delimiter> <file name> /
                      <delimiter>  SYMBL  <delimiter> <file name>
```

MICRO INSTRUCTION FORMATS (2600AI)

<micro>::= <register instruction> <rw> <carry> <delimiter> <a-reg> [,<b-reg> [,<c-reg>]] /

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

<immediate instruction> <rw> <delimiter> <expression[1]> [,<b-reg> [,<c-reg>]] /

<immediate multiply> <rw> <delimiter> <expression[2]> [,<b-reg> [,<c-reg>]] /

<mask branch> <delimiter> <expression[2]>, <b-reg>,<expression[4]>/

<register branch> <delimiter> <a-reg>, <b-reg>, <expression[4]>/

<branch instruction> <delimiter> <expression[4]>/

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

<misc. mini> <rw> <delimiter> <b-reg> /

LPI <rw> <delimiter> <expression[4]> /

CIO <delimiter> <expression[1]> /

SR <rw control> /

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

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

MVI <rw> <delimiter> <expression[1]>, <c-reg>

MVX <rw> <delimiter> <b-reg>, <c-reg>

1 $0 \le$ expression value $\le FF_{16}$
2 $0 \le$ expression value $\le F_{16}$
3 $0 \le$ expression value $\le 3FF_{16}$
4 $0 \le$ expression value $\le FFFF_{16}$

<name> ::= <symbol> / <null>

<symbol> ::= <letter>[<letter> / <digit>] $_0^7$

<delimiter> ::= [<space>] $_1^n$

<comment> ::= [<character>] $_0^n$

<a-reg> ::= F0/F1/F2/F3/F4/F5/F6/F7/CL-/CH-/CL/CH/CL+/CH+/+/- (non-extended)
F1F0/F2F1/F3F2/F4F3/F5F4/F6F5/F7F6/CLF7/CHCL/CLCH/DCH/DD/FOD (extended)

<b-reg> ::= <c-reg> /CH/CL (non-extended)
<c-reg> /CLPH/CHCL/SLCH (extended)

<c-reg> ::= F0/F1/F2/F3/F4/F5/F6/F7/PL/PH/SL/SH/K/ <null> (non-extended)
F1F0/F2F1/F3F2/F4F3/F5F4/F6F5/F7F6/PLF7/PHPL/SHSL/KSH/DR/FOD (extended)

<aux-reg>::= <expression[2]>

<expression[1]>::= <term> / <expression> + <term> / <expression> - <term>

<term> ::= <hexstring> / * / <symbol> / C'character'

<hexstring> ::= <digit> [<hexdigit>] $_0^n$

[1] For 2600 EI, <expression> ::= <hexstring> / * / * + hexdigit / * - hexdigit

[2] $0 \leq$ expression value $\leq 1F_{16}$

<hexdigit> ::= <digit> /A/B/C/D/_/F

<letter> ::= A/B/C/.../Z/@/$/#

<digit> ::= 0/1/.../9

<null> ::=

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

<carry> ::= ,0/,1/<null>

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

<register instruction> ::= OR/XOR/AND/A/DAC/DSC/AC/ORX/XORX/ANDX/AX/DACX/DSCX/ACX/NOP/<null>

<multiply or shift> ::= MHH/MHL/MLH/MLL/SHFT/MHHX/MHLX/MLHX/MLLX/SHFTX

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

<immediate multiply> ::= MIH/MIL

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

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

<branch instruction> ::= SB/B

<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

<misc. mini> ::= SR/TSP/TPS/TPS+1/TPS+2/TPS+3/TPS-1/TPS-2/TPS-3

8-BIT DATA

<micro> ::= DC <delimiter> [<value>]$_1^n$

<value> ::= [<hexdigit>]$_1^h$ /

"[<character>]$_1^n$" /

(<expression>)

where: h must be an even integer

Examples:

DC	81BC0A --	defines a 3-byte constant; each byte represented by 2 hex digits.
DC	"ABCD" --	defines a 4-byte constant whose value is the ASCII codes of the characters A, B, C and D.
DC	(TAG+3)--	defines a 2-byte constant whose value is the current value of 'TAG' + 3.

DC 04"STEP"B0($STEP) -- defines an 8-byte value.

ASSEMBLER ERROR CODES

A	=	invalid A-bus specification
B	=	invalid B-bus specification
C	=	invalid C-bus specification
E	=	too many operands
I	=	illegal immediate value
K	=	invalid CIO operand (> FF)
L	=	origin lower than address of last instruction + 1
M	=	multipli-defined symbol
M	=	invalid R/W field for SR
N	=	name required
O	=	illegal opcode field
P	=	out of page branch
P	=	invalid HEX codes
Q	=	name not allowed
R	=	illegal read/write/carry specification
S	=	invalid HEX on 'INSTR'
T	=	improper name (SYMBL) or too many names
U	=	undefined symbol referenced
V	=	illegal value
X	=	invalid AUX register specification
Z	=	CONT not last line in an EDIT file
+	=	feature not supported

Warnings

1	=	A bus } non-extended register mnemonics used with extended instruction
2	=	B bus }
4	=	C bus }
8	=	'A' or 'AX' instruction. These instructions no longer exist, and are assembled as 'SC' and 'SCX'

MEMORANDUM

TO: Bruce Patterson

FROM: Matthew Lourie

DATE: Aug 19, 1980

SUBJECT: CHANGES TO THE 2600 ASSEMBLER

I Overview of Changes

A File names

1 "2600ASMS" is the start-up program (used to be called "ASSEM26S").

2 "2600ASM2" is the assembler program (used to be called "ASSEM26").

3 "2600ASMB" is the block allocating program.

4 "2600ASMG" is the data generating program (used to be called "ASSEM26G").

5 "2600ASMD" is the data file produced by "2600ASMG" (used to be called "ASSEM26D").

B Programs

1 Changes to Start-up ("2600ASMS")

a Entry display slightly reorganized.

b Asks for a work file address:

1) If an answer of 000 is given the assembler will function essentially the same as the old one (i.e. no pass "M1" is done, etc.).

2) If a valid address is given, a block work file name will be made as such: XXWK.TMP where XX equals your initials. The assembler will assume that the source is to be assembled the new way (i.e. as blocks). The question "Do you want the code in order?" will be asked (default equals "Y"). If the answer is "Y" then allocating program will first attempt to deal the blocks in listing order before scrambling them.

d Although it is not apparent to the user, initialization is now done in start-up rather than in the assembler overlay.

2 Changes to the assembler ("2600ASM2")

a Format of displaying has changed.
b Passes have changed:
1) Pass "W1" gets block sizes and allocates blocks.
2) Pass "W2" is the same as "PREPASS".
3) Pass "WF" is the same as "PASS ONE"
4) Pass "MF" is the same as "PASS TWO"

3 Description of the block allocator ("2600ASMB")

a Chained to after completion of pass "M1".

b Overview of function:

1) Reads in block sizes.
2) Creates spans.
3) Allocates addresses to blocks.
4) Saves the addresses out.
5) Prints chart of block allocations.
6) Chains back to the assembler, thus starting pass "M2".

4 . Changes to the data generator:

a Renumbered.

b Restructured.

c Data file name changed.

II Blocks

A Description of blocks

A block is a segment of code that can be moved around so long as the whole segment is contained on one page of memory. This means that there can be no conditional references to addresses outside the block. If a conditional reference is made outside a block, it will be flagged as a "P" error. A block should not fall through since there is no way to tell where it will fall to (most likely to the scratch disk routine!). This error unfortunately can not be detected by the assembler (especially with computed branches).

B Defining of Blocks

There are essentially two types of blocks: floating blocks and absolute blocks. Floating blocks can be orged anywhere by the assembler. This allows 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 are considered to be floating blocks by definition. Absolute blocks are blocks which are defined to go at a certain address. They are defined by starting the code with an "ORG address" where the address is a hexadecimal number specifying where you want it to go. The address may not be anything but a simple hex number (i.e. 0120). 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
```

This statement 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, the assembler will give an error after pass "M1" and then abort. If the limits are not big enough to hold the code, the program will still attempt to allocate the blocks. After failing the standard memory map chart will be printed out, displaying the blocks which have been allocated as well as the ones which were 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 "D" errors.

D Notes on using the new assembler

1 Advantages

a Source files need not be rearranged to make the code fit. (This should save a lot of time and paper).

b A person with a disassembler would have a tough time understanding the organization or lack thereof of the code!

2 Disadvantages

With the old assembler, after doing an entire assembly, one could simply reassemble one module and insert it back into the old code. With the new assembler it is not as easy to do this. If one chooses or is forced to scramble the code, it is impossible to reassemble and insert a module since a module's object code would be all over the place. If one chooses to assemble in listing order then with a little trouble it is possible to reassemble a single module. One could put the proper limits statement into the module and reassemble, taking it out afterwards. This same method can be used to patch code into existing code.

3 Allocation method

The scrambling program has four modes:

a Mode one tries to allocate the blocks in order. If it succeeds it exits, else it switches to mode two.

b Mode two does a fast scramble of all the blocks. If there is less than zero free space it exits. Otherwise it goes to mode three.

c Mode three swaps the blocks around trying to compress them into smaller spaces. If after getting done the blocks fit, it exits. Otherwise it goes to mode four.

e 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 invoked even with zero free space. Even if it were invoked, it should only require a couple of passes but then who knows?

E NOLIST/ LIST feature

NOLIST and LIST pseudos have been added to the assembler (as an after thought of course). NOLIST causes the assembler to continue assembling (I hope!), but disables listing. LIST causes the assembler to resume listing. At the start of a module, the assembler is automatically put into list mode. NOLIST's and LIST's are stacked. This way if two NOLIST's appear with no LIST between them, two LIST's are required before listing will resume and vice versa (read that carefully). Cross references and title pages are always printed.

F Multiply Defined Symbols

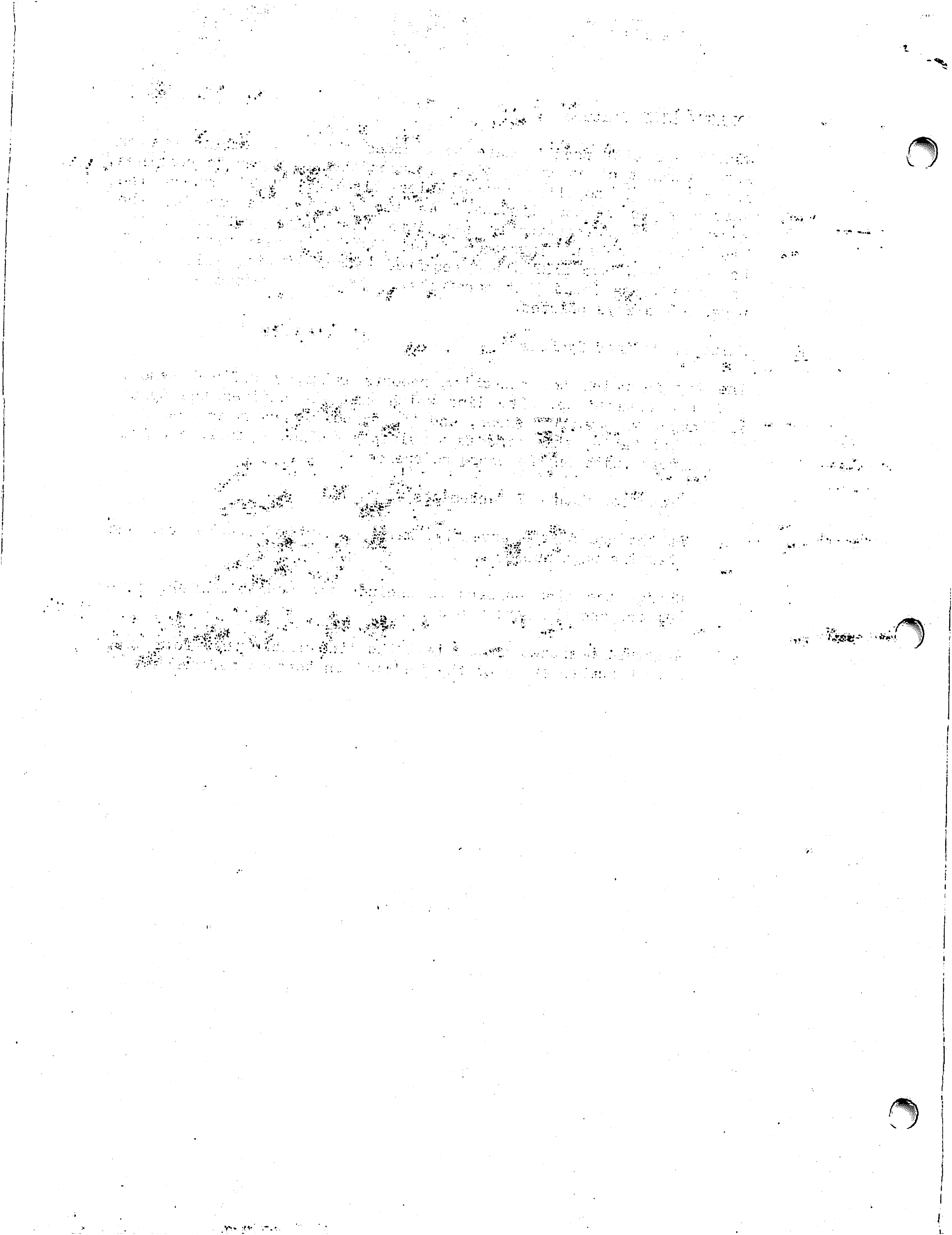
The way in which the assembler reports multiply defined symbols has been cleaned up. The line which multiply defines the symbol is flagged with an "M" error, and the symbol is again entered into the symbol table, this time as a multiply defined symbol, and thus will appear twice in the cross reference.

G Suggestions for Future Enhancements

1 At the end of the assembly, make a chart of module name and starting page number.

2 Change the line numbers to include the module number, file number, and the line number.

3 Absolute branches should tell the line number (new form i.e. module number etc.) of the instruction that it references.



MV = ORI 8?, Reg-1, Reg-2
MVI = ORI expression, dummy register, register
MVX = OR dummy reg pair, Reg-1, Reg-2

SR	-	subroutine return	B-Reg	Address from stack → IPC; Aux_{0-1F} → [illegible] PC
TAP	-	transfer auxiliary to PC's	B-Reg, Aux Reg_{0-1F}	PC ±1,2,3 → Aux_{0-1}
TPA [+1 / +2 / +3]	-	transfer PC's to auxiliary	B-Reg, Aux Reg 0-1F	
TPS [+1 / +2 / +3]	-	transfer PC's to stack	B-Reg	PC ±1,2,3 → Stack
TSP	-	transfer stack to PC's	B-Reg	TOStack → PC
XOR[X]	-	exclusive or	A-Reg, B-Reg, C-Reg	A XOR B → C
XORI	-	exclusive or immediate	lit8, B-Reg, C-Reg	lit8 XOR B-Reg → C-Reg
XPA [+1 / +2 / +3]	-	exchange PC's and auxiliary	B-Reg, Aux Reg_{0-1F}	PC ±1,2,3 ⇄ Aux_{0-1F}

where: 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

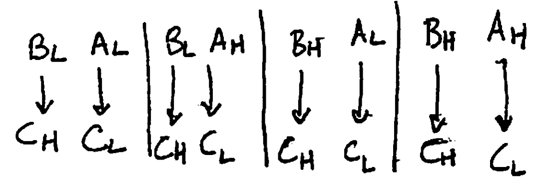
X = extended operation

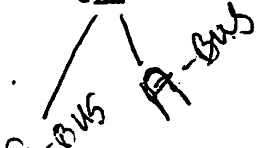
	Non-Extended	Extended
A-Reg	FO→F7 CL-, CH- CL, CH CL+, CH+ +, -	F1FO, F2F1, F3F2, F4F3, F5F4, F6F5, F7F6 CLF7, CHCL, CLCH, DCH, DD, FOD
B-Reg	C-Reg, CH, CL	C-Reg, CLPH, CHCL, SLCH
C-Reg	Fo→F7 PL, PH, SL, SH, K, Null	F1FO, F2F1, F3F2, F4F3, F5F4, F6F5, F7F6 PLF7, PHPL, SHSL, KSH, DK, FOD

2600 ASSEMBLY LANGUAGE MNEMONICS

INSTRUCTIONS

Mnemonic		Description	Operands (handwritten)	Operation (handwritten)
null	-	ORI		
AC[X]	-	binary add with carry	Cin, A-Reg, B-Reg, C-Reg	A+B+Cin → C+ Cout
ACI	-	binary add with carry immediate	lit8, B-Reg, C-Reg	lit8+B+Cin → C, Cout
AI	-	binary add immediate	lit-8, B-Reg, C-Reg	lit8 + B → C
AND[X]	-	and	A-Reg, B-Reg, C-Reg	A AND B → C
ANDI	-	and immediate	literal8, B-Reg, C-Reg	literal8 AND B-Reg → C-Reg
B	-	branch	lit16	
BEQ {H, L}	-	branch if equal to mask	lit4, B-Reg, lit16	
BER	-	branch if equal to register	A-Reg, B-Reg, lit16	
BF {H, L}	-	branch if false	→ lit4 B-Reg lit16	
BLER[X]	-	branch if less than or equal to register	A-Reg, B-Reg, lit16	if A≤B, branch
BLR[X]	-	branch if less than register	A-Reg, B-Reg, lit16	
BNE {H, L}	-	branch if not equal to mask	lit4, B-Reg, lit16	
BNR	-	branch if not equal to register	A-Reg, B-Reg, lit16	
BT {H, L}	-	branch if true		
CIO	-	control I/O		
DAC[X]	-	decimal add with carry	Carryin, A-Reg, B-Reg, C-Reg	A+B+Cin → C + Cout
DACI	-	decimal add with carry immediate	lit8, B-Reg, C-Reg	B+lit8+Cin → C+Cout
DSC[X]	-	decimal subtract with carry	Carryin, A-Reg, B-Reg, C-Reg	A−B−Cin → C, Cout
DSCI	-	decimal subtract with carry immediate	lit8, B-Reg, C-Reg (Lit8 + 9's comp (B) + Cin)	lit8 − B − Cin → C, Cout
LPI	-	load PC's immediate	lit16	
M {HH, HL, LH, LL} [X]	-	binary multiply	A-Reg₄, B-Reg₄, C-Reg₈	A * B → C
MI {H, L}	-	binary multiply immediate	lit4, B-Reg, C-Reg	lit4 * B → C
MV[X]	-	move value of register to register	B-Reg, C-Reg	
MVI	-	move value to register	lit8, C-Reg	
NOP	-	ORI		
OR[X]	-	or	Carryin, A-Reg, B-Reg, C-Reg	A OR B → C
ORI	-	or immediate	literal8, B-Reg, C-Reg.	literal8 OR B-Reg → C Reg
SB	-	subroutine branch	lit16	
SC[X]	-	binary subtract with carry	Cin, A-Reg. B-Reg. C-Reg	A+B̄+Cin → C, Cout
SH {HH, HL, LH, LL} [X]	-	shift	A-Reg, B-Reg, C-Reg	

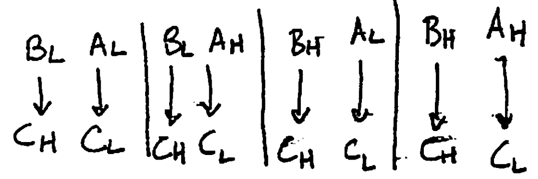




2600 ASSEMBLY LANGUAGE MNEMONICS

INSTRUCTIONS

Mnemonic		Description	Operands (handwritten)	Operation (handwritten)
null	-	ORI		
AC[X]	-	binary add with carry	Cin, A-Reg, B-Reg, C-Reg	A+B+Cin → C+ Cout
ACI	-	binary add with carry immediate	lit8, B-Reg, C-Reg	lit8+B+Cin → C, Cout
AI	-	binary add immediate	lit8, B-Reg, C-Reg	lit8 + B → C
AND[X]	-	and	A-Reg, B-Reg, C-Reg	A AND B → C
ANDI	-	and immediate	literal8, B-Reg, C-Reg	literal8 AND B-Reg → C-Reg
B	-	branch	lit16	
BEQ {H, L}	-	branch if equal to mask	lit4, B-Reg, lit16	
BER	-	branch if equal to register	A-Reg, B-Reg, lit16	
BF {H, L}	-	branch if false	→ lit4, B-Reg, lit16	
BLER[X]	-	branch if less than or equal to register	A-Reg, B-Reg, lit16	if A≤B, Branch
BLR[X]	-	branch if less than register	A-Reg, B-Reg, lit16	
BNE {H, L}	-	branch if not equal to mask	lit4, B-Reg, lit16	
BNR	-	branch if not equal to register	A-Reg, B-Reg, lit16	
BT {H, L}	-	branch if true		
CIO	-	control I/O		
DAC[X]	-	decimal add with carry	Carryin, A-Reg, B-Reg, C-Reg	A+B+Cin → C + Cout
DACI	-	decimal add with carry immediate	lit8, B-Reg, C-Reg	B+lit8+Cin → C+Cout
DSC[X]	-	decimal subtract with carry	Carryin, A-Reg, B-Reg, C-Reg	A-B-Cin → C, Cout
DSCI	-	decimal subtract with carry immediate	lit8, B-Reg, C-Reg (Lit8 + 9's comp (B) + Cin)	lit8 - B - Cin → C, Cout
LPI	-	load PC's immediate	Lit16	
M {HH, HL, LH, LL} [X]	-	binary multiply	A-Reg (4), B-Reg (4), C-Reg (8)	A * B → C
MI {H, L}	-	binary multiply immediate	Lit4, B-Reg, C-Reg	lit4 * B → C
MV[X]	-	move value of register to register	B-Reg, C-Reg	
MVI	-	move value to register	lit8, C-Reg	
NOP	-	ORI		
OR[X]	-	or	Carryin, A-Reg, B-Reg, C-Reg	A OR B → C
ORI	-	or immediate	literal8, B-Reg, C-Reg	literal8 OR B-Reg → C Reg
SB	-	subroutine branch	lit16	
SC[X]	-	binary subtract with carry	Cin, A-Reg, B-Reg, C-Reg	A+B̄+Cin → C, Cout
SH {HH, HL, LH, LL} [X]	-	shift	A-Reg, B-Reg, C-Reg	



B-Bus A-Bus

MV = ORI 8?, Reg-1, Reg-2
MVI = ORI expression, dummy register, register
MVX = OR dummy reg pair, Reg-1, Reg-2

SR	-	subroutine return	B-Reg	Address from stack → IPC
TAP	-	transfer auxiliary to PC's	B-Reg, Aux Reg 0-1F	Aux_{0-1F} → PC
TPA [+1 +2 +3]	-	transfer PC's to auxiliary	B-Reg, Aux Reg 0-1F	PC ±1,2,3 → Aux_{0-1}
TPS [+1 +2 +3]	-	transfer PC's to stack	B-Reg	PC ±1,2,3 → Stack
TSP	-	transfer stack to PC's	B-Reg	TOStack → PC
XOR[X]	-	exclusive or	A-Reg, B-Reg, C-Reg	A XOR B → C
XORI	-	exclusive or immediate	lit8, B-Reg, C-Reg	lit8 XOR B-Reg → C-Reg
XPA [+1 +2 +3]	-	exchange PC's and auxiliary	B-Reg, Aux Reg 0-1F	PC ±1,2,3 ⇄ Aux_{0-1F}

where: 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

	Non-Extended	Extended
A-Reg	F0→F7 CL-, CH- CL, CH CL+, CH+ +, -	F1F0, F2F1, F3F2, F4F3, F5F4, F6F5, F7F6 CLF7, CHCL, CLCH, DCH, DD, F0D
B-Reg	C-Reg, CH, CL	C-Reg, CLPH, CHCL, SLCH
C-Reg	F0→F7 PL, PH, SL, SH, K, Null	F1F0, F2F1, F3F2, F4F3, F5F4, F6F5, F7F6 PLF7, PHPL, SHSL, KSH, DK, F0D

WANG LABORATORIES, INC.

ASSEMBLY LANGUAGE EDITOR

Program Description

Revised September 9, 1975

The following describes the function, operation and use of the 2200 Assembly Language EDITOR program. This program was written by Dave Angel, Research and Development Department in Tewksbury.

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

EDITOR — 9/9/75 VERSION

A. Purpose

To create and edit assembly language format data files for use as input to various assemblers operating on the 2200 and the 360.

B. Requirements

32K 2200B or 2200C with Options 2 and 5
or
Equivalent WCS series
Disk Unit (floppy or cartridge) and/or cassette unit
Optional Printer

C. Features and Limitations

The editor operates on disk files of no more than 320 lines. These lines are stored in a compressed format, 4 lines to a sector; usually in a file of 86 sectors. For programs of more than 320 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 LISTDCR will not show it). Periods, blanks, commas, and dashes are special characters and cannot be used in the file name except exactly as defined.

The character set within a line includes all codes between HEX(10) and HEX(7F) that are available on a keyboard. Care should be taken not to create data which cannot be handled by the assembler. Index and reverse index are ignored, and backspace, space, and carriage return are used as special characters.

All or part of one or more disk data files (or all of one cassette data file) may be loaded into memory up to the restriction of 320 lines, individual lines may be edited, and groups of lines may be inserted or deleted from the file in memory. A range of lines may be initialized to a particular image. Finally, the file in memory may be saved, either over an old disk file, in a newly defined disk area, or on a cassette.

D. Line Edit Format

Lines are edited in a field format as follows. The last statement in the BASIC program is a data statement with four numbers. These define the maximum lengths of the four fields in each line. Generally, the values used are 8, 9, 23, 59. These have the following meanings:

Field	Usual Meaning	Maximum Length
1	Tag Field	8
2	Opcode	9
3	Operand Field	23
4	Comment Field	59*

*The length available in the comment field is (59 - 23 - 9 - 8) plus whatever is not used in the first 3 fields. Thus in a typical line with no tag, an opcode of 4 characters, and an operand of 10 characters, the comment field can be up to 45.

The display of a line, for list or edit, consists of one or two lines, as follows.

If the first character of a line is an asterisk, the line is a comment line, whose maximum length is automatically 58 characters. All field oriented features are inactive for the line. At the left of the display is a line number and a colon. Then the line (beginning with the asterisk) is displayed.

If the first character of a line is not an asterisk, the fields are as defined in the chart. The line number and colon are at the left, followed by the first 3 fields in tab form. The fields always start in column 6, 15, and 25 (with this field definition convention), in order to line up the columns. There is always at least one blank separating the fields. If there is a comment field, it is displayed on the following line, indented under the tag field.

If these field definitions are not suitable, the data statement may be changed subject to the following restrictions. There must be four positive integers in the data statement. The first 3 numbers must not total more than 55. The fourth number should be 59.

E. RUN Sequence

When the program is first loaded into memory, as well as whenever a different person wishes to EDIT, the RUN sequence should be followed.

Type RUN (EXEC). The CRT will then display the special function key options. At any later time, this display can be recalled by S. F. '16, or a condensed version by S. F. '0.

'0 - S. F. KEY OPTIONS DISPLAY	'16 - EXPANDED S. F. DISPLAY
'1 - EDIT MODE	'17 - EDIT LAST LINE
'2 - LIST	'18 - LIST TO PRINTER
'3 - INSERT MODE	'19 - COPY LINES
'4 - DELETE MODE	'20 - LOAD '!' FROM DISK
'5 - DISK/CASS.	'21 - CHANGE EDIT FILE NAME
'6 - SEARCH EDIT	'22 - INITIALIZE LINES

The program will then go to the change EDIT FILE NAME routine (see S. F. '21 description) and ask for today's date, operation initials, and the EDIT FILE NAME. Finally it asks if it should load the EDIT FILE from disk.

F. Special Function Keys

Once the program has been RUN, the first seven special function keys ('0 to '6, '16 to '22) are used to get into, and to change, modes of operation. A function key may be pressed at any time except as listed below, and the current operation will be aborted in favor of the new. There is no danger of stacking subroutines too high, as the stack is always kept under careful control. The function keys are described below.

Each time a function key is pressed ('1, '2, '3, '4, '5, '6, '17, '18, '19, '21, '22) one or more questions will be asked. There are only four types of questions, and they are answered as follows:

If a number is needed

Only positive numbers are used; a negative number will be rejected. A decimal is truncated before interpreting. In most cases only a range (i.e., 1 to 320, or 1 to 319, ...) is legal. Carriage return will get a default value.

If a file name is needed

Up to 6 character file name is taken. There must be some non-numeric character in the name. Thus J341, 285L6 are legal, but 489 is not. Carriage return will get a default name (displayed). The initials of the operator may be overridden by typing II.NNNNNN where II are the overriding initials, and NNNNNN is the file name. The initials may not be overridden for a SAVE operation; this protects the other person's data.

If a keyword is needed

L is interpreted the same as LOAD, likewise for S = SAVE, F = FILE, T = TAPE, D = DISK, Y = YES, N = NO.

If a 'character string' is needed

This character string (used by the SEARCH EDIT) must be typed exactly as you want it. Blanks and punctuation (if any) are important, and trailing blanks are not ignored. Type the string exactly as it appears, and press EXEC.

If the questions can be anticipated correctly, more than one answer may be typed, separated by blank or comma.

Example:

```
LIST FROM LINE # 10, 20
```

will list from 10 to 20 and be equivalent to

```
LIST FROM LINE # 10
       TO LINE # 20
```

If default values for the questions are wanted, a period can be used to indicate this.

Example:

```
LIST FROM LINE # 10,.
```

will list from 10 to the end of text and be equivalent to

```
LIST FROM LINE # 10
       TO LINE # (CR)
```

S. F. '0 and '16 — display options

These keys each display the S. F. key options, then return the machine to Console Input (CI) mode. At all other times, once a S. F. key has been pressed, the program stays in KEYIN mode. S. F. '0 gives a short list of options, in order to save CRT space, and S. F. '15 gives a long list (see Chart on Page 4). Undefined S. F. keys default to S. F. '0.

S. F. '1 — EDIT

This is the mode used most of the time. A question appears: "EDIT LINE #" and the response should be a number in the range from 1 to 320. A zero will default to line 1. If ",L" is typed after the line number to be edited, the previous 12 lines will be displayed (listed) before editing the specified line. Once the number has been accepted, that line is displayed (if the line is null, only the line number appears) with the cursor at the end. There are nine S. F. keys for cursor positioning, inserting, deleting, etc., and semicolon, LINE ERASE, asterisk, BACKSPACE, SPACE and the text atom keys are specially defined. When the line is correct or complete, a carriage return will store it in the array, and recall the next line. If it is desired to edit out of sequence, either press CONTINUE or S. F. '1 to return to the question "EDIT LINE #". A line is not changed in memory until CR/LF, EXP(, or LOG(is pressed. To leave EDIT mode and get into a different mode press the appropriate S. F. key at any time.

EDIT mode S. F. keys

The following S. F. keys have meaning only within EDIT mode, and while editing a line. Most of them are very similar, or at least analogous to the built in EDIT functions for the same keys. For most of '7 to '14, shift does not affect operation; but '15 and '31 are different, and '12 and '28 are different.

S. F. '7 (and '23) —	Reverse tab (circular)
S. F. '8 (and '24) —	Erase remainder of line
S. F. '9 (and '25) —	Delete one character from current field (This does not affect following fields).
S. F. '10 (and '26) —	Insert "↑" or " " at current cursor position. This moves the rest of the field, and the last column of the field is lost.
S. F. '11 (and '27) —	Move right 5 columns
S. F. '12 —	Move right 1 column
S. F. '13 (and '29) —	Move left 1 column
S. F. '14 (and '30) —	Move left 5 columns
S. F. '15 —	Move cursor to end of line
S. F. '28 —	Change character to lowercase
S. F. '31 —	Move cursor to begin of line

EDIT mode special characters

Backspace — move cursor left one space, operates just like S. F. '13.
Semicolon — move cursor to beginning of next field. If already in comment field, this key is a normal semicolon.
Line erase — erase all of current line, move cursor to position 1. This is the only key which can delete the * from column 1 of a comment line.
Asterisk — (in column 1 only) — this makes the line a comment.
Space — clears remainder of field and tabs to beginning of the next field. If already in comment field this is an ordinary blank character.

EDIT mode text atom characters

In order to expand the capabilities of the keyboard, certain text atom keys have been defined. These fall into three groups.

(1) to replace those characters used as special characters.

PRINT — put a blank in current field without tabbing. In the comment field this is equivalent to a space. It should be used with caution anywhere else.

ARC — becomes a semicolon with no tab function.

(2) to provide ASCII characters not normally provided on a keyboard.

SIN(— left bracket '['
COS(— right bracket ']'
TAN(— left arrow '←'
#PI — back slash '\'

(3) to provide additional features while editing.

EXP(— after editing a line, the line may be entered by pressing EXEC, EXP(or LOG(. The difference lies in what line is displayed next. When EXP(is pressed, the current line is saved, the next four listed, and the fifth displayed for editing. This is useful when paging through the program to find a particular line.

LOG(— after editing a line if you wish to go back to the previous line, press 'LOG(' instead of 'EXEC'. The present line will be saved and the previous line displayed for editing. This is useful for correcting mistakes, as well as for charts and tables where columns must be aligned.

RUN — this key interrupts the editing in order to change the line number; effectively moving the line. A question is asked 'NEW LINE #'. Type in the line number where this line is to go, and the new number will be displayed with the line. Finish editing the line and press EXEC. The line will be stored at the new line # instead of the old.

S. F. '2 — LIST

A question appears: "LIST FROM LINE #" and the response should be in the range 1 to 320. Then the question "TO LINE #" appears, and a number greater than or equal to the first should be entered. For convenience both numbers may be entered at once, separated by a comma. The defaults are 1 and the highest defined line respectively.

This listing appears in EDIT format. Null lines are represented by the line ** NULL ** XX where XX is the number of null lines. This is to save CRT space. After the listing is complete, the program goes automatically into EDIT mode, asking the question "EDIT LINE #".

If you wish to interrupt the listing there are two choices — please do not press RESET.

(1) press a S. F. key — program will immediately jump to that routine, ignoring remainder of listing.
(2) press a regular character (e.g., EXEC). This will halt the listing, which will resume when the key is pressed again, or abort is a S. F. key is pressed.

S. F. '3 — INSERT

This is used to insert lines between existing lines. Operationally it inserts null lines at the specified point, then moves the remaining lines down to accommodate them. The questions are:

INSERT AFTER LINE #
HOW MANY LINES?

The first answer is in the range 0 to 319, and the sum of two must not be greater than 320. Default for number of lines is 1. Once the insert operation has started, it should be allowed to complete (do not press RESET).

If text may overflow (because the last line is moved beyond 320) the question "OKAY TO OVERFLOW?" will appear. The answer Y (or YES) will cause the INSERT to be performed, and possibly lines will be lost. Any other answer will interrupt the INSERTING process.

S. F. '4 — DELETE

This is the inverse of INSERT. It removes lines from the specified range, then moves the remaining lines up to fill the void. Null lines are added at the end as needed. The questions are:

DELETE STARTING LINE #
ENDING LINE #

The first answer is in the range 1 to 320, the second must not be less than the first, nor greater than 320. The default for the second is the value entered for the first. Note, that by using 1,320, all the data is quickly cleared from memory. Once the delete operation has started, it should be allowed to complete (do not press RESET).

S. F. '5 — DISK and CASSETTE OPERATIONS

This begins by asking the question:

CASSETTE (C), DISK (D), OR DELETE NULLS (N),

The response to this will branch the editor to one of the three types of logic. The default is Disk. A fourth response is also valid; LOAD will cause a disk load from the current EDIT FILE name.

CASSETTE:

There are 2 cassette operations. The question will be asked,

CASSETTE: SAVE (S) OR LOAD (L) ?

After typing L or S, the system will wait with the message

PRESS EXEC WHEN CASSETTE IS MOUNTED

When EXEC is pressed, the editor will begin saving or loading.

SAVE — The current EDIT FILE name and the entire contents of memory is saved on the cassette. It is the operators responsibility to (1) rewind the cassette before and after saving, (2) be sure that the proper cassette is mounted, (3) protect his data from other users by proper cassette storage.

LOAD — all of memory is cleared, and the cassette is read in. The date and file name from the cassette is displayed for the operator (but not saved anywhere) and the data is loaded. There is no provision for combining data from more than one cassette. This should be done with disk load and save operations.

DISK:

There are 3 disk operations, identified by the first letter of the keyword. The question will appear:

DISK: LOAD (L), SAVE (S), RESERVE NEW DISK FILE (F)

The response to this will branch the editor to one of the three routines below.

(L) LOAD — loads in data from disk saved previously by the editor. Four questions must be answered before loading commences, as follows:

LOAD WHAT DISK FILE -- this is the name of the EDIT FILE when it was saved, and the name of the actual disk file to be loaded. If it is the same name as that of the current EDIT FILE, press CR/LF to get the default.

LOAD STARTING AT WHAT (MEMORY) LINE NUMBER — first line to be loaded over. (default = 1)

ENDING AT LINE NUMBER — last line to be loaded over. Note, that all lines in the range will be initialized, whether or not the disk file is large enough to fill them. (default = 320).

DISPLACEMENT (DSKIP) IN DISK FILE -- this is basically a DSKIP to be performed before reading. It allows a portion other than the beginning of a disk file to be loaded. (default = 0).

(S) SAVE — this command saves the entire current edit file (without null lines) on a previously defined disk file.

The program will ask what the old disk file name is with the question

SAVE OVER WHAT OLD DISK FILE ?

Note, that this disk file's information is to be lost, and the disk space re-used. For this reason the disk file must be your own (identified by initials). It will generally be either an earlier version of the EDIT FILE, or a newly created (see (F) RESERVE NEW DISK FILE, below) disk file. In either of these cases, the OLD DISK FILE name is the same as the EDIT FILE name, and CR/LF will get the correct default. If the new name does not match the old, then the OLD FILE name must be entered.

The file is renamed if necessary with the name of the current EDIT FILE. This is the name to be used in the assembler or in the future editing.

(F) RESERVE NEW DISK FILE -- is used to allocate new disk space. 86 sectors are reserved for each file, no matter how many lines have been entered. This allows total compatibility for renaming and combining files. This should not be used if current files exist that are useable; SAVE allows renaming while SAVING.

The program will ask what name to use for the new disk file with the question.

CREATE WHAT NEW DISK FILE ?

The default name for the reserved file is the name of the current EDIT FILE. Disk errors such as ERR 79 (File Already Catalogued) are not fatal to the program or the data. Simply press S. F. '5, and try again with a name that hasn't been used before.

S. F. '6 -- SEARCH EDIT

This allows a handy way to find (and possibly change) all lines with some character or character string. The program asks:

SEARCH EDIT, WHAT CHARACTER STRING ?

The response is to type 1 to 24 characters followed by a carriage return. Type the string exactly as it appears in the line. Do not include leading or trailing blanks unless they are meaningful. All the characters should be within one field for comparison purposes. Thus the editor can search for a particular tag (and find both the tag and any operands using it), but cannot look for both tag and opcode or for both operand and comment. In operation, the editor searches for the first matching line. When it is found, it is displayed for editing. There is a + before the line number to remind the operator he is in SEARCH EDIT mode. The line may be changed, and EXEC will save it and find the next matching line. The RUN key for renumbering should not be used while in SEARCH EDIT mode, as it will return the EDITOR to standard EDIT mode. When all the lines have been displayed, the editor goes into standard EDIT mode (with EDIT LINE # message). If there are no matches, the editor goes directly to EDIT mode. Note, that the search can take seconds, depending on the amount of text and the particular characters being searched for.

S. F. '17 — EDIT LAST LINF

This displays the last 12 lines and enters EDIT mode with the following one. This is a convenient way to add to a file.

S. F. '18 — LIST TO PRINTER

This operates like LIST, except the listing is to device 215 instead of 005. Also, null lines are represented by a single blank line, instead of the symbol ** NULL ** XX.

S. F. '19 — COPY

This statement is used to copy a range of lines within the EDIT FILE. There are two fields, of the same length; a FROM field and a TO field. The location and direction of the copy is specified by the first line of each field. Four questions are asked:

```
COPY FROM -- FIRST LINE #
     FROM -- LAST LINE #
       TO -- FIRST LINE #
       TO -- LAST LINE # (XXX)
```

The first question must be answered in the range 1 to 320. The second defaults to the first (e.g., for copying 1 line), and if entered must be at least as large. The third answer should be from 1 to 320. The fourth question is redundant information. The editor calculates from the first 3 values what this should be, and displays this (default) value in parentheses. This provides some protection in case of mis-types. Usually the operator should press EXEC to start the COPY. However, one may type a replacement value, which in effect will override the second answer. The number of lines copied is

$$B - A + 1 = D - C + 1$$

S. F. '20 — LOAD '!' FROM DISK

Assuming a program is on the fixed disk platter called '!', this key will load it in, clearing the EDITOR in the process. This can be used to go quickly from the EDITOR to other programs, such as an assembler.

S. F. '21 — CHANGE EDIT FILE NAME

This routine (automatically called during a RUN sequence, as well as whenever S. F. '21 is pressed) allows the operator to change the revision date and the EDIT FILE name, as well as allowing the old text to be cleared and new to be loaded. The program asks:

REVISION DATE (   ) ?

with the default date in parentheses. If the default is correct, press EXEC, otherwise type a new date (up to 9 characters). If several revisions are made to one file in a single day, it is useful to append a letter to the date. Thus typical dates are: 11/20/75 12/14/74A 8/12/75C. Blanks and commas may not be included in the date. After the date has been entered, the system may ask

OKAY TO CLEAR OLD TEXT ?

(this message is skipped if there is no text currently in memory). The response Y or YES will cause the text to be cleared. The default answer is N (for NO). Next the EDITOR will ask for the

NEW EDIT FILE NAME

Once again, the default is displayed. If the default is not correct, it can be changed by typing name, or initials.name, in the format described on Page 5. Neither the name nor the initials may have any dashes in them.

Next comes the question

DISK LOAD THE FILE ?

The default is NO. If Y is typed, the EDITOR will clear the text in memory, then load the entire disk file with the same name as the current EDIT FILE name. This load is the fastest kind (and is equivalent to answering LOAD to the first question on S. F. '5).

S. F. '22 — INITIALIZE LINES

This statement is used to set a range of lines equal to a certain value, e.g., a null line. The program asks:

INITIALIZE STARTING AT LINE #

Type the first line number of the range. The present value of that line will be displayed and the program will go into INIT mode. This mode is identical to EDIT mode except that

(1) There is an asterisk before the line number to remind the user than he is initializing.
(2) When CR/LF is pressed, control returns to the next question:

INITIALIZE ENDING LINE #

If the number entered is less than or equal to the first, then only the one line is changed. Otherwise, all lines in the specified range are initialized to the value of the starting line. Note: the RUN key for renumbering must not be used while initializing. It will turn the operation into the standard EDIT.

G. After a file has been saved

Once a file has been edited and saved, the data may be cleared for another file in one of the following ways:

(1) If the parameters of the new disk load are 1,320, then all previous data will be cleared while loading the new. A cassette load always clears the previous data.
(2) If an initialize is done with a range 1 to 320, all lines are set to a specific value (i.e., to clear, type S. F. '20, 1 EXEC, LINE ERASE EXEC, 320 EXEC.
(3) In a similar way, delete 1,320 will clear out all lines very quickly.
(4) Immediate mode INIT (20) L$()
(5) Change Edit File Mode Name (S. F. '21) — The editor will ask if the old text should be cleared. Type Y EXEC.

The last method is preferable, since it's quick and foolproof. Also, it allows the EDIT FILE name to be changed at the same time.

H. Miscellaneous Comments

If the system ever locks out, and a printer list preceeded it, suspect that the printer is still selected. Key RESET, S. F. '0, to reselect the CRT.

If disk errors occur, the program and data is generally still safe. Check error number to find out what was wrong (N1$ is EDIT FILE name; N2$ is the name of the DISK FILE to be loaded, and the old file name to be scratched and saved over). Then press RESET and retry the sequence. A LIST DCR may help find the problem.

Anytime a system command may be useful, rather than pressing RESET, type S. F. '0, which returns the 2200 to CI mode. At this time, disk catalogues may be examined, variables printed, or immediate mode calculations made.

Incidentally, if RESET is ever pressed during an operation, S. F. '0 should cure all pointer problems, but we recomment doing a list (S. F. '2) to check, especially if an insert, delete, or load was in operation.

If disk data is coming from or going to a disk other than the R (removable = right) disk, press S. F. '0 to get CI mode, then type an appropriate SELECT DISK command.

SELECT DISK B10 — Removable or right
SELECT DISK 310 — Fixed or left
SELECT DISK 350 — 3rd platter (model 2243)

B10 is reselected each time the RUN sequence is followed.

Similarly, if more than one cassette is to be used, or other than the standard 10A, an immediate mode

SELECT TAPE 10B
SELECT TAPE 10C

should be typed to reselect. 10A is reselected each time the RUN sequence is followed.

I. DISK FILE FORMAT

This section is needed only by those writing assemblers for this editor, not for users.

The disk file is saved as a catalogued disk file, with up to 86 sectors. On each sector are four 62 byte line images, and after the last used sector is an end of file sector. The first sector is not considered part of the data, and contains the file name and revision date in the first line image. This information is useful for a subtitle on assemblies. The 62 byte compressed line images are defined as follows:

(1) If blank, it is a null line and should be ignored for assembling.

(2) If the first column is an asterisk, then it is a comment line, and in print form.

(3) If the first column is not an asterisk, then it is compressed as follows.

There are three HEX(A0) 'tab' characters separating the four fields. Each field may have embedded blanks, but trailing blanks are not included in the compressed format. Using POS and STR functions the fields may be separated into four variables, an array of length four, or any other format the assembler desires. The first field may be null, indicated by a leading HEX(A0). The second or third field may be null, indicated by two or three consecutive HEX(A0) characters. The HEX(A0) may be eliminated by AND(,7F) after separation. A simple program could be (assuming B$ = 62 byte compressed line).

```
FOR I = 1 TO 3
L = POS(B$ = A0,
A$(I) = STR(B$, 1, L)
AND(A$(I), 7F)
B$ = STR(B$, L + 1)
NEXT I
A$(4) = B$
```

A faster way, if the $UNPACK instruction is available, is the following. (Assuming D$ is a two byte alpha variable).

```
INIT (20) A$()
D$ = HEX(01A0)
$UNPACK = (D = D$) B$ TO A$()
```

J. CASSETTE FILE FORMAT

This section is needed by those writing assemblers for this editor, not for users.

The cassette file is saved as a data file of up to 62 blocks on the cassette. There is a data header block with the name "EDIT".

Next is a block with four alpha variables in it. Only the first is used, and this contains the FILE name and revision date. Next is the data, four 62 byte line images per block, in the format described in section I., DISK FILE FORMAT. Finally, there is a trailer record.

Generally, only one file will be on a cassette, but if the cassette was deliberately not rewound by the operator, it may safely contain 3 files, and possibly more.

K. NOTES ON ASSEMBLERS

Note, that since the commment is restricted to 58 characters it is useful to allow some sort of 'tab' character to allow part of a comment to line up with comment fields of regular lines. This is entirely a function of the assembler, but we are planning in the future assemblers to use the backarrow '←' as a tab.

Also note that since multiple files are needed for an assembly, an automatic chaining technique would be useful. One successful technique is to end each file with the line

```
CONT <filename>
```

where CONT is an assembler PSEUDO op and <filename> is the (eight character) file name of the next file to be assembled.

L. 2200 ERROR CODES

If any of the following 2200 errors occur, the description may help to find the problem.

61 -- Disk hardware error -- Try pressing RESET, then repeat the sequence.

62 -- File Full -- If the edit file on disk was originally catalogued by other than the EDITOR, perhaps not enough space was allocated. Use LIMITS or LISTDCR to check; at least 86 sectors should be reserved.

65 -- Disk hardware malfunction -- See error 61.

66 -- Format key engaged -- Turn format key to LOCK.

67 -- Disk format error -- If this is a new platter, it must be formatted at SCRATCHED before the EDITOR can use it. If it's an old platter, see error 61.

68 -- LRC error -- This is usually a dirty or loose connector to the disk.

72 -- Cyclic Read error -- See error 61.

78 -- File not scratched -- This error should never occur.

79 -- File already catalogued -- This error may occur when saving the EDIT FILE over some other disk file. It means that there is already a disk file with the same name on the catalogue. There are two solutions: Change the EDIT FILE name before saving (e.g., if the duplication was accidental, or if two versions are to be maintained), or save the current EDIT FILE over the disk file with the same name (i.e., update).

80 -- File not in catalog -- Check if the proper platter is mounted. If attempting to save a new file, space must be reserved on the disk prior to saving. Otherwise, check for spelling errors.

WANG LABORATORIES, INC.

2200VP RESIDENT 2600 ASSEMBLER

January 31, 1978

INTRODUCTION

ASSEM26 is an assembler designed to run on the 2200VP (BASIC-2) and assemble 2600 source code, both control memory and data memory. This is not a released program and is intended for internal use only. Questions should be directed to the 2200 Microprogramming group.

HARDWARE REQUIREMENTS:

1. 2200VP with 64K RAM
2. 132 column printer (address 215)
3. Dual disk drive (normally address 320, B20)

SYSTEM INPUT

1. Source Text

Source lines packed 4/record are read from a disk file on the removable platter. Each line is up to 62 characters in length and must be in one of the following formats:

1. * comment
2. name $A0_{16}$ opcode $A0_{16}$ operand $A0_{16}$ comment

Blank lines are ignored.

The first record of the source file contains the source file name and the REV. date. (The first 25 characters of the first variable (alpha) in the record). The name and date are printed at the top of each page of the assembler listing.

Two generalized editing programs written by Dave Angel produce a source file in the above format, "EDITOR" which runs on a 2200C or 2200T, and "EDIT26" which runs on a 2200VP.

2. External Symbol Files (#2)

External symbol files may be referenced by use of the SYMBL pseudo in the assembly.

3. System Tables (#1)

4. External Reference Check (#5)

This file should contain a list of all the modules that may possibly reference this module. When reassembling only part of a system, the external reference feature allows all linking errors to be noticed and corrected.

SYSTEM OUTPUT

1. CRT display while running hows progress of assembly and counts errors.

2. Listing and cross reference on 132 column printer.

3. Object file (#4).

4. Symbol table files (#3) (internal and external).

All are optional except CRT output.

To customize the program all disk selects are done and documented at the end of text. By convention we use 320 for all files except the EDIT files, which can be selected to B20 or B10 by the operator.

One line can be changed to alter any or all of these.

OPERATING INSTRUCTIONS

A. RUN

B. Press EDIT, then S. F. key '0 (as prompted)

   1. Hard Copy -- this will select whether the listing will be printed or not.

   2. Continue -- this determines whether the assembler will accept the CONT pseudo.

   3. Title Page -- this determines whether title page(s) will be included in each assembly module.

   4. Chk extern -- this determines whether an external cross reference check will be performed after the module is completed, to identify any symbols which have changed and will cause problems in other modules.

   5. Edit disk -- this identifies which source disk will be used for the edit files.

   6. Assemlist -- this allows a single file to contain all the names and symbol files to be used for this assembly. The file is created "AS.MLIST" by the EDITOR and consists of simply name fields with the file names spelled out. This option can save a lot of typing if several modules need to be assembled often.

C. Press EXEC when all options are acceptable.

D. Type initials.

E. Type object file name -- this must begin with your initials and end with an "@". It should not have a period in between.

F. For each module.

1. Type one or more edit file names.

2. Type external symbol file name (ends in $) or type just $ to not save external symbols.

These names will default to the operators initials, or initials can be entered explicitly, followed by a period.

G. Press CR an extra time.

H. Enter date and time.

2600 ASSEMBLER LANGUAGE SYNTAX DEFINITION

December 3, 1974, Revised January 27, 1975

Revised January 31, 1978

The following pages define the syntax of the 2600 Assembler Language in Backus Normal Form where the following meta symbols are used:

< >	encloses syntax classes
::=	means "is defined as"
/	or
$[\ ]_a^b$	encloses entries that may be repeated from 'a' to 'b' times. (if 'a' is omitted, default = 0, if 'b' is omitted, default = 1)
...	implies a sequence of elements

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

Three forms of the 2600 Assembler Language are defined:

1. 360 Assembler

   <360 assembly line>::= <name> <delimiter> <micro>
   <delimiter><comment> / *<comment>/ <pseudo><delimiter><comment>

   Card format: columns 1 - 71 <360 assembly line>
   columns 72 - 80 <sequence number>
   <name> must start in column 1.

2. 2600 AI Assembler (ASSEM26)

   <2600 assembly line>::= <name> <delimiter> <micro>
   <delimiter> <comment> / * <comment> / <pseudo> <delimiter> <comment>

3. 2600 EI

   <2600 EI line>::= <instruction> <delimiter>
   <comment>

2600 ASSEMBLER SEMANTICS

A. Pseudo Instructions

MODULE - define a module title (should be first text line).

ORG - origin instructions at the specified address.

EQU - define the symbol equal the specified value.

TITLE - issue a form feed if not at top of page and print the specified comment.

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

EJECT - issue a form feed if not at top of page.

PAGE - origin instructions to the beginning of the next page (1024 instructions) if not at the beginning of a page.

1024 Inst.

SYMBL - defines an external symbol table that can be referenced during the assembly.

CONT - assembly source code continues in the specified file.

B. Move Instructions

The move instructions are implemented by using the OR and ORI instructions as follows:

MV = ORI 00, register 1, register 2

MVI = ORI expression, dummy register, register

MVX = OR dummy register pair, register 1, register 2

2600 ASSEMBLY LANGUAGE MNEMONICS

INSTRUCTIONS

Mnemonic	Description
null	ORI
AC[X]	binary add with carry
ACI	binary add with carry immediate
AI	binary add immediate
AND[X]	and
ANDI	and immediate
B	branch
BEQ {H / L}	branch if equal to mask
BER	branch if equal to register
BF {H / L}	branch if false
BLER[X]	branch if less than or equal to register
BLR[X]	branch if less than register
BNE {H / L}	branch if not equal to mask
BNR	branch if not equal to register
BT {H / L}	branch if 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 {H / L}	binary multiply immediate
MV[X]	move value of register to register
MVI	move value to register
NOP	ORI
OR[X]	or
ORI	or immediate
SB	subroutine branch
SC[X]	binary subtract with carry
SH {HH / HL / LH / LL} [X]	shift

2600 ASSEMBLER SEMANTICS

A. Pseudo Instructions

MODULE	-	define a module title (should be first text line).
ORG	-	origin instructions at the specified address.
EQU	-	define the symbol equal the specified value.
TITLE	-	issue a form feed if not at top of page and print the specified comment.
SPACE	-	skip the specified number of lines. A null operand implies skip one line.
EJECT	-	issue a form feed if not at top of page.
PAGE	-	origin instructions to the beginning of the next page (1024 instructions) if not at the beginning of a page.
SYMBL	-	defines an external symbol table that can be referenced during the assembly.
CONT	-	assembly source code continues in the specified file.

B. Move Instructions

The move instructions are implemented by using the OR and ORI instructions as follows:

MV	=	ORI 00, register 1, register 2
MVI	=	ORI expression, dummy register, register
MVX	=	OR dummy register pair, register 1, register 2

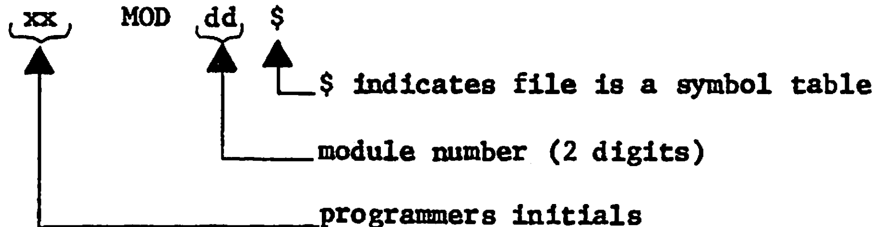
SYMBL PSEUDO

SYMBL is used to define external symbol tables that can be referenced during an assembly. The operand field contains a file name of the symbol table file that can be referenced. Up to 40 SYMBL pseudos are legal in any assembly. The external symbol tables are disk files stored on the fixed platter.

When the assembler encounters a SYMBL pseudo, the specified symbol table name is entered in the External Symbol Table Name Table. Whenever a symbol is referenced, the internal symbol table (the symbol table generated by this assembly) is scanned for that symbol. If the symbol is not found, the external symbol tables defined by the SYMBL pseudos are scanned for the symbol in the order in which they were defined. If the symbol is found in an external symbol table, that symbol and its value are entered into the internal symbol table and marked as being externally defined. If the symbol is not found, it is entered into the internal symbol table and marked as being undefined.

External Symbol Table Names

Generally, external symbol table names will have the following format:



When an externally defined symbol is entered into the internal symbol table, the module name is entered into the definition field. The cross-reference then prints the definition for the external symbol as the module name in which it was found.

Creation of External Symbol Tables

During assembly setup the user is asked:

INITIALS. FILE NAME (CR IF NO MORE)?

If a name ending in $ is specified an external symbol table will be created. Only those symbols defined in the current assembly will be saved. If '-N' follows the name, a new file will be created; if '-N' is omitted, the file is assumed to be old (already created). If old, the existing file will be overwritten.

MICRO INSTRUCTION CODES

MNEMONIC	SKELETON CODE
*A	0C0000
AC	180000
ACI	380000
ACX	1A0000
AI	2C0000
AND	080000
ANDI	280000
ANDX	0A0000
*AX	0E0000
B	5C0000
BEQH	740000
BEQL	700000
BER	500000
BFH	6C0000
BFL	680000
BLER	480000
BLERX	4C0000
BLR	400000
BLRX	440000
BNEH	7C0000
BNEL	780000
BNR	580000
BTH	640000
BTL	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
XPA	038000
XPA+1	038200
XPA+2	038400
XPA+3	038600
XPA-1	03C200
XPA-2	03C400
XPA-3	03C600
MHLX	1E8000
MIH	3C8000
MIL	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

*These mnemonics no longer valid

2600 ASSEMBLY LANGUAGE MNEMONICS

INSTRUCTIONS

Mnemonic		Meaning
null	-	ORI
AC[X]	-	binary add with carry
ACI	-	binary add with carry immediate
AI	-	binary add immediate
AND[X]	-	and
ANDI	-	and immediate
B	-	branch
BEQ {H, L}	-	branch if equal to mask
BER	-	branch if equal to register
BF {H, L}	-	branch if false
BLER[X]	-	branch if less than or equal to register
BLR[X]	-	branch if less than register
BNE {H, L}	-	branch if not equal to mask
BNR	-	branch if not equal to register
BT {H, L}	-	branch if 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 {H, L}	-	binary multiply immediate
MV[X]	-	move value of register to register
MVI	-	move value to register
NOP	-	ORI
OR[X]	-	or
ORI	-	or immediate
SB	-	subroutine branch
SC[X]	-	binary subtract with carry
SH {HH, HL, LH, ...} [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] - exclusive or

XORI - exclusive or immediate

XPA [+1 / +2 / +3] - exchange PC's and auxiliary

where: 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

Other instruction field parameters:

```
instruction [{ ,    } [,0] ]
             { ,R   } [,1]
             { ,W1  }
             { ,W2  }
             { ,RCM }
             { ,WCM }
```

where: R = read
W1 = write 1
W2 = write 2
RCM = read control memory (SR only)
WCM = write control memory (SR only)

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

CC84B/25A

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.

√ what is 2250's?

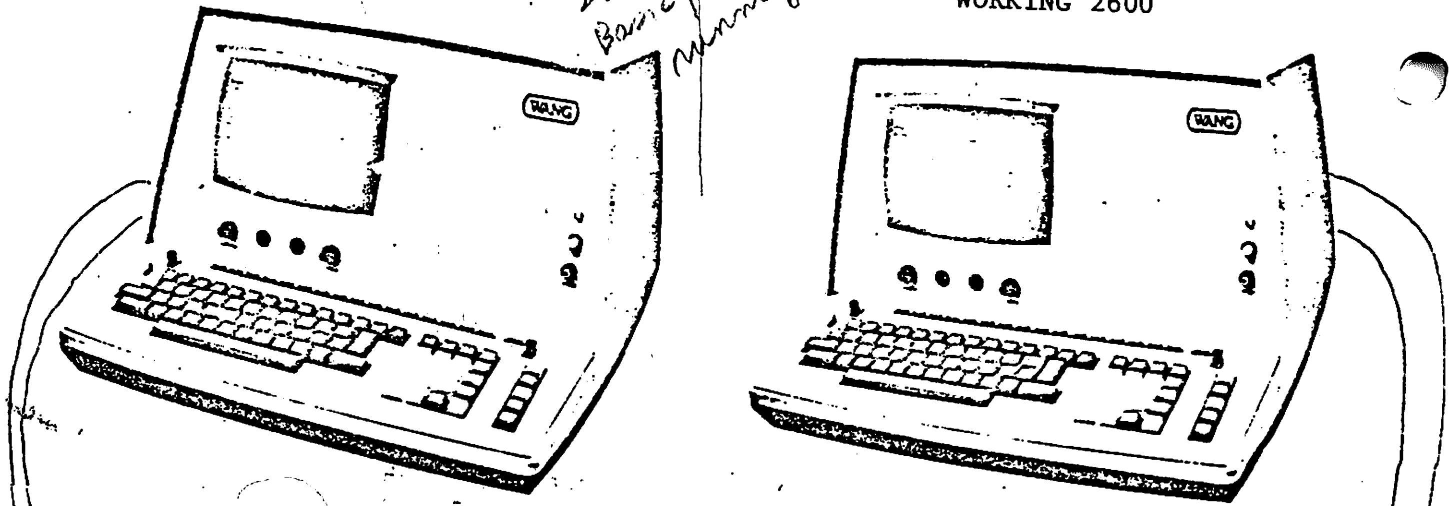
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851 • TEL. (617) 851-4111 • TWX 710-343-6769 • TELEX 94-7421

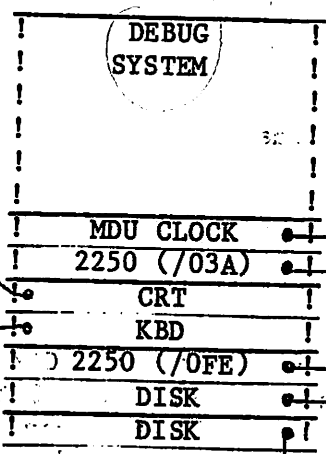
Standard

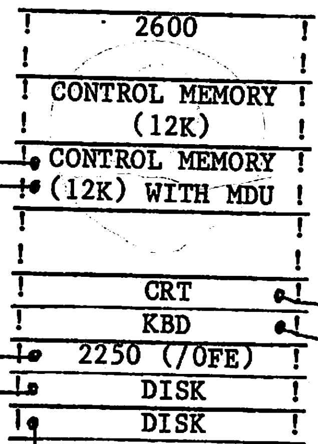
DEBUG SYSTEM

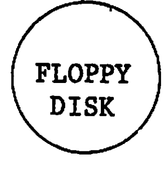
Standard 2600

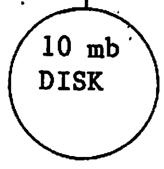
WORKING 2600











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-
LAST   0000 - OR        F0,F0,F0     800000                  0.00 MS.
NEXT   0001 - OR        F0,F0,F0     800000

 K SH SL CH CL PH PL F7 F6 F5 F4 F3 F2 F1 F0                 BREAKPTS
00 02 00 00 00 00 00 00 00 00 00 00 00 00 00
                                                    STACK
AUX 00-07   0000 0000 0000 0000 0000 0000 0000 0000  8269
AUX 08-0F   0000 0000 0000 0000 0000 0000 0000 0000  6660
AUX 10-17   0000 0000 0000 0000 0000 0000 0000 0000  554F
AUX 18-1F   0000 0000 0000 0000 0000 0000 0000 0000  443E
                                                     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)

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.

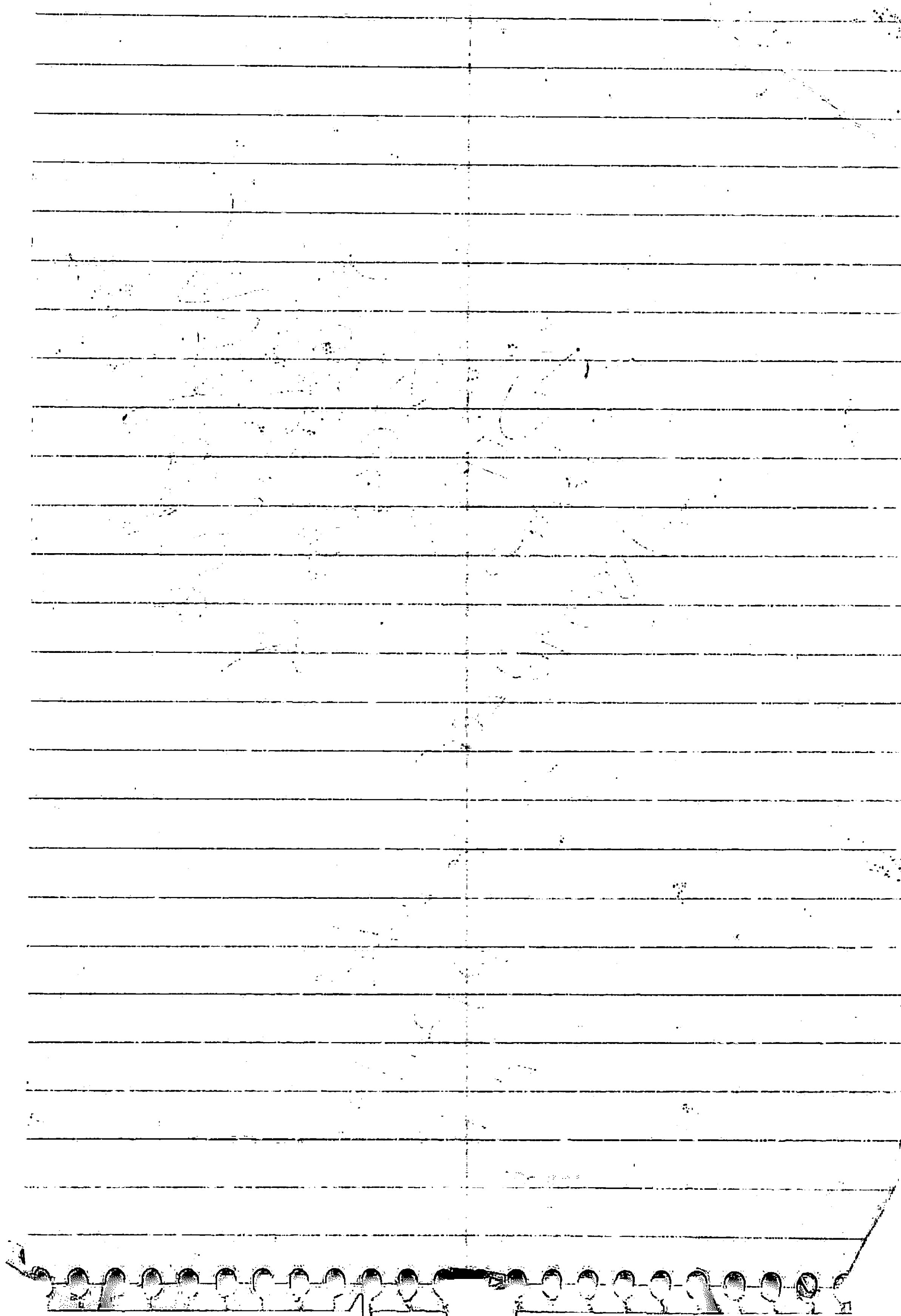
$INIT (A$, T$, C$, P$(), D$(), P$, HEX(ab))

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.

This book
needs whole
section of
EDITOR
AND
ASSEMBLER
to be whole
12/21/84
MCF



To: Jerry Sevigny
From: Bruce Patterson
Date: December 15, 1986

Subject: 2200 PRINTING TO VS PRINTERS

This document outlines the procedures used to print the 2200 BASIC-2 assembly listings on the VS high-speed band printer. This method has some advantages:

(1) The VS band printer is fast (1100 lpm).
(2) The VS band printer stacks paper reasonably well.
(3) Print files are created on the PC that can be accessed via PCEDIT.
(4) The 2209A is not used (it's broken)

However, this method is not without problems:

(1) VS server errors occurred, requiring operator attention.

Assembly

The print output from the assembly is captured in a PC text file. To do this:

(1) Run MCS 2200 Terminal Emulator on the PC in 2200 lab.

Use the 9600 baud, for spooling to PC configuration.

(2) Setup a text file for capturing print output. Do this by:

Press PRINT key.
Enter the text file name when prompted. For example:
File for output: /BASIC2/BASIC2.TXT
Return to terminal emulation by pressing RETURN

select capture terminal printer output

(3) Run the 2200 Block Assembler.

Assembler output 004

Note, the Assembler was modified to supress CRT output while printing to /004. This avoids a bug in MCS code -- MCS cannot handle alternating CRT and print output. Unfortunately, the display of the number of errors on each module is also suppressed.

code in order? Y/N
N is probably faster & it doesn't matter

(4) Close the print file by pressing SHIFT+PRINT. after Assembly is complete

source file is 22MVP - which holds a list of files to assemble
object file is MVP@

after this is done, at MDU, initialize instructions & data memory to 7DFF (7E04 - MDU code begins), load MVP@ (name of output object code), set instruction F to *805FFF & hit SF key to calculate checksums, & save out to disk again. * must end in FFF - and PD loads at 6400 so 5FFF is highest location available to use.

set 000F = 5FFF
initial inst & data to 7DFF
0BFF
7DFF
load @ 31A

Chop Assembly Print File

The assembly step produced a single PC text file containing all the print output from the assembly. This could be printed directly (assuming the VS can handle such a large print file). However, it is convenient to chop the text file up into separate text files for each module. To do this:

(1) Run PC BASIC-2 (select from PC Main Menu).

(2) :LOAD "CHOPLIST.BS2"
:RUN

Enter name of the file to be chopped. For example:
Assembly text filename: /BASIC2/BASIC2.TXT

Enter name of the 1st text file to be created. For example:
1st text filename: /BASIC2/START

The 1st text file receives the beginning of the assembly listing. Then, the listing of each module encountered is put into a separate text file. Each module begins with a comment of the form:

*=module=nnnnnnnn, where nn...n is the module name.

CHOPLIST closes the last text file and creates a new text file with the name nnnnnnnn in the same directory as the 1st text file whenever one of these special comments is found.

CHOPLIST also produces a text file, *ERRORS.txt, containing the number of errors in each module.

Printing on the VS

Printing is done by transferring the PC text files to the VS for printing using DATA EXCHANGE. To do this:

(1) Set the VS up for DATA EXCHANGE.

Select 928 Workstation from the PC Main Menu.
Logon to the VS.
Press Funtion key 1 and run DEXACCES.
Press RETURN (this might not be necessary).
Suspend terminal emulation by pressing SHIFT+CONTROL+CANCEL and then S and EXEC.

(2) Send the text files to the VS.

Enter DOS Command Processor.
Run VSPRINT.BAT (in /bin) as follows:
C:VSPRINT fully-qualified-text-filename vs-print-filename
(e.g., C:VSPRINT C:/BASIC2/BPMVP00 BPMVP00)

The batch file is setup to print to printer 159 (the high-speed band printer). Other printers can be specified by modifying the batch file. The user may also want to change the identifying initials in the VS print file name "HOTLIPS:#BMPPRT.Z2".

A batch file has been set up to send all the BASIC-2 listing files to the VS. To run this:

C:CD /BASIC2
C:VPRINTB2

Master Cross Reference

The print output from the Cross Reference is captured in a PC text file. To do this:

(1) Run MCS 2200 Terminal Emulator on the PC in 2200 lab.

Use the 9600 baud, for spooling to PC configuration.

(2) Setup a text file for capturing print output. Do this by:

Press PRINT key.
Enter the text file name when prompted. For example:
File for output: /BASIC2/XREF.TXT
Return to terminal emulation by pressing RETURN

(3) Run Cross Reference.

Output device address 004

(4) Close the print file by pressing SHIFT+PRINT.

(5) Print on the VS using VSPRINT.BAT.

L12 pin 8 = High when halted
Low when running

Debug System – a stock VP or MVP with 2 2250 boards + a standard O.S. loaded.	Working System – Unit to be tested, has MDU board + one 2250

Start ~~Working~~ Debug system with a standard release of MVP O.S.

Select DISK to address of the MDU BASIC-2 programs

~~LOAD RUN "2600MDU". It'll say it's executing~~

Power on ~~Debug~~ Working system, ~~with~~

Press RESET, Type @MDU and hit the SF key that ~~corres~~ corresponds to address of @MDU code (in ~~DEBUG~~ working system)

after ~~DEBUG~~ @MDU display stops on working system console, go back to ~~Working~~ Debug system console

On ~~Working~~ Debug system LOAD RUN "2600MDU" + press SF'15 (halt) twice
RESET CLEAR V, RUN
SF'15 (Halt) should now work if
~~CR (SF'01) IA to address of~~
you hit it a few times.

SF'08 (Load) specify disk address within ~~Working~~ Debug system of @MVP. Then enter @MVP

After Load is complete,

SF'01 (Change Reg) reg 1A to disk address on the ~~Debug~~ working system (This is the address of @GENPART)

SF'30 (set 1C's) to 0003

SF'31 (GO)

Now the ~~Debug~~ working system should load @GENPART. Run that if desired, and two systems are ready for test.

At this point SF'15 (Halt) on ~~working~~ Debug system should work, and once halted, every other command should work.

2250 addresses in Debug system:

From terminal 1 of ~~working~~ Debug system

#5 - 03B - to MDU board in work sys.
#6 - 0FB } to working systems
#7 - 0FA } 2250

From a second terminal (only if you have ~~a~~ an MVP ~~working~~ Debug system, and presumably more than one ~~debug~~ working systems)

#5 - 03D - to MDU board in work sys.
#6 - 0FF } to ~~Debug~~ working
#7 0FE } systems 2250

In either case #8 = 10FD for the 2228B clock board.

2250 address in Working system (apparently) set to 10FE

To write to CAM + enable breakpoints

$O10 #5(4400 A000 4410)STR(H$(),16)

To exec

$O10 #6(4401 40F0 4400

4260 4270 4240 4250

4220 4230 4210, G$)

where G$ is initialized from R$ + R9$
and last byte is 00 for step, 01 for go

KEYIN is done from #7, with non END1
bytes thrown away, until an END1
is received (line 140)

then a 5A command is sent
to #6

Disable CAM

$O10 #5 (4400)

Get 8 bit rep from MDU

$O10 #6 (4401 40F1 4400)
followed by $O10#7 (C360) —
for each section of registers,
auxes, + stack

Display registers

$O10 #8 (4503 ... C620)

Examine data

O10 #6 (4401 4005 4400 4210 4220 4230,4240)
O10 #7 (C360) A1$

7E00 0111 1100 0000 0000

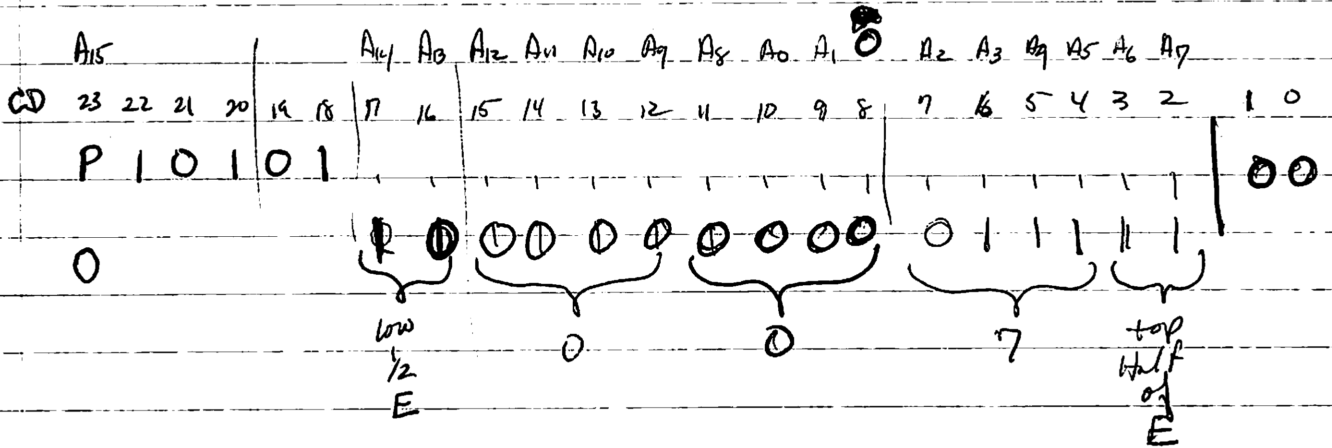
A7 A0

1 2 3 4 5 6 | 7 8 1 2 3 4 5 6 7 8

SW1 F F F F F F | N N S2 N F N N N N N F

#1 1111 1100 0000 0001

#2+ 1111 1000 0100 0000



56007C

0101 0110 0000 0000 0111 1100

RedShaw memo

Test description	2.6 VLSI	3.4 VLSI	% gain	386 1.0	386 RRR	% gain
	!	!	!	!	!	!
TEST1 (Resolve w brackets @MRTIAN - 10 times)	3:16	3:17	0	:58		+70
TEST2 (Resolve w/o brackets @MRTIAN - 10 times)	3:20	3:20	0	:60		+70
Risk retrieval No contention - cache clear 10 different risks in succession	:24					
22 sec						
	@2.4 ache 1.0 386	@2.2		@4.6		-92