

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

I. INTRODUCTION.

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

II. HARDWARE REQUIREMENTS:

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

III. ASSOCIATED FILES:

- A. "2600ASMS" contains the start-up program.
- B. "2600ASM2" contains the assembler program.
- C. "2600ASMB" contains the block allocating program.
- D. "2600ASMG" contains the data file generating program.
- E. "2600ASMD" contains the data file produced by "2600ASMG".
- F. "SCROSS" contains the master cross reference program.
- G. "READTP" contains the 9-track tape to printer program.

IV. OVERVIEW OF PROGRAMS:

A. Start-up:

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

- a. Assembler output device.

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

- b. Source edit file device.

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

- c. External symbol file device.

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

- d. Block work file device.

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

2. Then the following file names will be requested:

- a. Object file name.

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

- b. Starting module name.

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

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

B. Main assembler program.

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

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

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

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

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

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

C. Block allocator.

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

2. Overview of purpose:

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

D. Tape to printer program.

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

- a. Run the program called "READTP".
- b. Give the tape output address (default 77B) and the printer output address.
- c. Give the file index numbers and the number of copies to be printed. The maximum number of file indices is ten.
- d. Enter "0" in the file index field after all the desired file index numbers are entered. Thereafter the "READTP" will print out the files in the order in which their indices were entered.

V. BLOCKS.

A. Definition of a block.

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

B. Defining a block.

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

C. Setting the LIMITS.

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

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

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

D. Allocation algorithm.

The scrambling program has four modes:

1. Mode one tries to allocate the blocks in order. If it succeeds and listing order is specified it exits, else it switches to mode two.
2. Mode two does a fast scramble of all the blocks. Then it goes to mode three.
3. Mode three is a compression mode. It swaps the blocks around trying to fit them into smaller spaces. If after getting done the blocks fit, it exits. Otherwise it goes to mode four.
4. Mode four is incredibly slow. On a typical assembly of BASBOL, it took a half hour a pass! It is very, very unlikely that mode four will ever be needed on a large assembly.

E. Assembler modes.

- a. When the block work file address is requested the user has two reply options:
 - 1) If the user answers with "000" the assembler will function in the nonblock mode. This means that "ORG *" will be treated as do nothings, and the LIMITS psuedo is illegal.
 - 2) If the user gives a valid address, the assembler will function in the block mode. A block work file will be opened with the name xxWK.TMP where xx are your initials. Later on, the start-up program will ask if you want the code in listing order. If you answer "Y", then the allocating program will not attempt to further compress the code so long as the code fits when it is in listing order.

F. Printing options.

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

1. Write the assembler output directly to the printer. This is done by giving a valid printer address for the assembler output device.
2. Write the assembler output to the tape only. This is done by specifying a valid tape address for the assembler output device and specifying "000" for the tape printing device.
3. Write the assembler output to the tape, and after assembly completion have this tape file dumped to the printer. This is done by giving a valid tape address for the assembler output device and giving a valid printer address for the tape printing device.

VI. ASSEMBLER PSEUDOS.

A. MODULE

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

B. ORG

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

C. ORGD

Sets the assemblers data pointer to the specified address.

D. EQU

Assigns a symbol a value.

E. TITLE

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

F. SPACE

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

G. EJECT

Issue a form feed if not at top of page.

H. PAGE

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

I. ASSEMBLE

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

J. SYMBL

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

K. CONT

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

L. LIST and NOLIST

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

VII. SYMBOLS.

A. Local and multiply defined symbols.

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

B. External, previous, and undefined symbols.

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

C. Symbol cross reference.

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

VIII. INSTRUCTIONS.

A. Syntax.

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

B. Move Instructions.

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

1. MV Fx,Fy ==> ORI 00,Fx,Fy
2. MVI n,Fx ==> ORI n,,Fx
3. MVX FwFx,FyFz ==> ORX DD,FwFx,FyFz

C. Omitted opcode field.

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

D. Subroutine branch and return instruction.

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

ASSEMBLY LANGUAGE SYNTAX

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

1. The "<" and ">" characters enclose syntax classes.
2. The symbol "::=" means "is defined as".
3. The character "/" means "or".
4. "[" ... "]"^x means that the entries within may be repeated from zero to "x" times. If the "x" is omitted, assume a value of one.
5. "... " implies a sequence of elements.
6. Capital letters and symbols not in "<" ">" are actual letters in the language. Lower case letters represent English language expositions such as "space".

ASSEMBLER LINE FORMAT

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

PSEUDO INSTRUCTION FORMAT

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

```
<pseudo> ::= [<symbol>] <delimiter> ORG <delimiter> <address>
/ [<symbol>] <delimiter> PAGE
/ [<symbol>] <delimiter> ORGD <delimiter> <address>
/ <symbol> <delimiter> EQU <delimiter> <word>
/ <delimiter> LIMITS <address> , <address>
/ <delimiter> MODULE <delimiter> <comment>
/ <delimiter> TITLE <delimiter> <comment>
/ <delimiter> SPACE <delimiter> [<expression>]
/ <delimiter> EJECT
/ <delimiter> CONT <delimiter> <file name>
/ <delimiter> SYMBL <delimiter> <file name>
/ <delimiter> ASSEMBLE <delimiter> <file name>
```



```

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

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

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

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

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

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

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

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

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

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

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

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

/ <branch instruction> <delimiter> <address>

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

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

```

<a-reg> ::= F0/F1/F2/F3/F4/F5/F6/F7/CL-/CH-/CL/CH/CL+/CH+/+/-
 <a-ext> ::= F1F0/F2F1/F3F2/F4F3/F5F4/F6F5/F7F6/CLF7
 / CHCL/CLCH/DCH/DD/FOD
 <b-reg> ::= <c-reg>/CH/CL
 <b-ext> ::= <c-ext>/CLPH/CHCL/SLCH
 <c-reg> ::= F0/F1/F2/F3/F4/F5/F6/F7/PL/PH/SL/SH/K/<null>
 <c-ext> ::= F1F0/F2F1/F3F2/F4F3/F5F4/F6F5/F7F6/PLF7
 / PHPL/SHSL/KSH/DK/FOD
 <aux-reg> ::= <five bit value>
 <rw> ::= ,R/,W1/W2
 <carry> ::= ,0/,1
 <rw and/or carry> ::= <rw> [<carry>]
 / <carry> [<rw>]
 <rw control> ::= ,RCM/,WCM

EXAMPLES:

SR,W1 F0 -- write register F0 to data memory and do a
 subroutine return.
 ,W1 23,,F0 -- write a HEX(23) to data memory and copy this
 same value to register F0.
 BEQHH 75,F2,LAB1 -- branch to "LAB1" if the high nibble of register
 F2 equals HEX(7).
 MV F0,F1 -- copy register F0 to register F1.

8-BIT DATA FORMAT

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

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

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

/ " <character> [<character>]ⁿ "

/ (<expression>)

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

EXAMPLES:

```
DC      81BC0A      -- defines a 3-byte constant; each byte represented
                    by two hex digits.

DC      "ABCD"      -- defines a 4-byte constant whose value is the
                    ASCII representation of the string "ABCD".

DC      (TAG+3)     -- defines a 2-byte constant whose value is the
                    current value of 'TAG' + 3.

DC      04"STEP"    -- defines a 5-byte value.
```

MISCELLANEOUS

<nibble> ::= <expression>
<byte> ::= <expression>
<word> ::= <expression>
<address> ::= <expression>
<five bit value> ::= <expression>
<expression> ::= <term>
/ <expression> + <term>
/ <expression> - <term>
<term> ::= <digit> [<hexstring>]
/ *
/ <symbol>
/ C'<character>'
/ X'<hexstring>'
<symbol> ::= <letter> [<letter>/<digit>]⁷
<delimiter> ::= tab
<comment> ::= [<character>]ⁿ
<hexstring> ::= <hexdigit> [<hexdigit>]ⁿ
<hexdigit> ::= <digit>
/ A/.../F
<letter> ::= A/.../Z
<digit> ::= 0/.../9
<null> ::=

ASSEMBLER ERROR CODES

ERRORS:

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

WARNINGS:

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

MICRO INSTRUCTION CODES

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

INSTRUCTION SUMMARY

AC[X]	--	Binary add with carry.
ACI	--	Binary add with carry immediate.
AI	--	Binary add immediate.
AND[X]	--	Logical and.
ANDI	--	Logical and immediate.
B	--	Branch.
BEQ	--	Branch equal immediate.
BER	--	Branch equal.
BF	--	Branch false.
BLER[X]	--	Branch less than or equal.
BLR[X]	--	Branch less than.
BNE	--	Branch not equal immediate.
BNR	--	Branch not equal.
BT	--	Branch true.
CIO	--	Control I/O.
DAC[X]	--	Decimal add with carry.
DACI	--	Decimal add with carry immediate.
DSC[X]	--	Decimal subtract with carry.
DSCI	--	Decimal subtract with carry immediate.
LPI	--	Load PC's immediate.
M	--	Binary multiply.

MI	$\left. \begin{array}{c} \text{HH} \\ \text{HL} \\ \text{H} \\ \text{L} \\ \text{LH} \\ \text{LL} \end{array} \right\}$	-- Binary multiply immediate.
MV[X]		-- Move.
MVI		-- Move immediate.
NOP		-- No operation.
OR[X]		-- Logical or.
ORI		-- Logical or immediate.
SB		-- Subroutine branch.
SC[X]		-- Binary subtract with carry.
SH	$\left\{ \begin{array}{c} \text{HH} \\ \text{HL} \\ \text{LH} \\ \text{LL} \end{array} \right\} [X]$	-- Shift.
SR		-- Subroutine return.
TAP		-- Transfer auxiliary to PC's.
TPA	$\left[\begin{array}{c} +1 \\ +2 \\ \overline{+3} \end{array} \right]$	-- Transfer PC's to auxiliary.
TPS	$\left[\begin{array}{c} +1 \\ \overline{+2} \\ \overline{+3} \end{array} \right]$	-- Transfer PC's to stack.
TSP		-- Transfer stack to PC's.
XOR[X]		-- Logical exclusive or.
XORI		-- Logical exclusive or immediate.
XPA	$\left[\begin{array}{c} +1 \\ \overline{+2} \\ \overline{+3} \end{array} \right]$	-- Exchange PC's to auxiliary.

SPECIAL NOTATION

OPCODE SUFFIXES:

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

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

X = extended operation.

PARAMETER FIELD MNEMONICS:

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

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