SEGA OF AMERICA, INC.
Consumer Products Division

The SATURN SCU DSP Assembler User's Manual Addendum

Doc. # ST-240-A-SP1-052295

READER CORRECTION/COMMENT SHEET

Keep us updated!

If you should come across any incorrect or outdated information while reading through the attached document, or come up with any questions or comments, please let us know so that we can make the required changes in subsequent revisions. Simply fill out all information below and return this form to the Developer Technical Support Manager at the address below. Please make more copies of this form if more space is needed. Thank you.

General Information:

Your Name ______________________ **Phone** __________

Document number ST-240-A-SP1-052295 **Date** __________

Document name The SATURN SCU DSP Assembler User's Manual Addendum

Corrections:

Chpt.	pg. #	Correction

Questions/comments: ______________________

Where to send your corrections:

Fax: (415) 802-1717
Attn: Evelyn Merritt,
Developer Technical Support

Mail: SEGA OF AMERICA
Attn: Evelyn Merritt,
Developer Technical Support
150 Shoreline Dr.
Redwood City, CA 94065

The SATURN SCU DSP Assembler User's Manual Addendum

by Dennis Caswell
5/1/95

Introduction

The DSP assembler (dspasm.exe) assembles programs written for the DSP that is a part of the SATURN System Control Unit (SCU). It produces object files in S-record format, in Hitachi assembler source format, and in C source format.

Invoking the Assembler

The syntax for invoking the assembler is:

```
dspasm [<options>] <source file> [<object file>]
```

If the `<object file>` is omitted, the assembler will produce an object file with the same name as the source file and the extension ".s".

Command Line Options

`/a`	Output the DSP program as an assembler source file having the extension ".d" in addition to the S-record object file, which is always produced. The file is in Hitachi format, and it consists of a series of `.DATA.L` statements.
`/c`	Output the DSP program as a C source file having the extension ".d" in addition to the S-record object file, which is always produced. The file contains a comma-separated list of 32-bit hexadecimal numbers which can be used in an array initializer.
`/l`	Output an assembled listing having the extension ".lst" in addition to the S-record object file, which is always produced.
`/m`	Enables midbox (Model M) compatibility mode.

These options may be invoked with either a slash or a minus sign, and they may be used in combination with one another.

Source File Format

Each line of a DSP assembler source file has the following format.

```
[<label>]  [<opcode> [<operand list>]] ... [<comment>]
```

Either spaces or tabs may be used as separators. Labels need not begin in column 1 (however, see "Labels," below), and opcodes must not begin in column 1. Comments begin with a semicolon and continue to the end of the line.

A backslash may be used as a continuation character at the end of a source line, allowing a long logical line to be broken into several physical lines. In order to continue a line which contains a comment, the continuation character must immediately precede the semicolon.

The maximum line length is 255 characters. The assembler does not distinguish between upper and lower case.

Labels

Labels consist of up to 32 letters, numbers, and/or underscores. The first character of a label may not be a number. A label on an instruction need not begin in column 1, but, if it doesn't, it must be appended by a colon. If the label does begin in column 1, the colon is optional.

Reserved Words

The following symbols are reserved and should not be used as labels: `ALH, ALL, ALU, M0, M1, M2, M3, MC0, MC1, MC2, MC3, MUL`. Using these symbols as labels will not generate an error message in all cases, but it may cause your code to be assembled in ways you probably didn't expect.

Opcodes

The assembler recognizes the following opcodes.

Operation commands: `MOV, ADD, SUB, AD2, AND, OR, XOR, SR, RR, SL, RL, RL8, CLR, NOP`

Load Immediate command: `MVI`

DMA commands: `DMA, DMAH`

Jump command: `JMP`

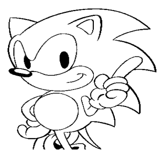
Looping commands: `BTM, LPS`

End commands: `END, ENDI`

Normally, at most one opcode appears on each source line, however, up to six operation commands may appear on a single source line, subject to the DSP's hardware limitations (see the programming examples at the end of this manual).

Assembler Directives

`ELSE`	Concludes an `IF` clause and begins an `ELSE` clause. If the expression in the preceding `IF` directive was zero, then the statements between the `ELSE` directive and the next `ENDIF` directive will be assembled; otherwise, they will be ignored.
`ENDIF`	Concludes an `IF` directive.
`ENDS`	Causes the assembler to ignore everything from this directive to the end of the file.
`EQU`	Equates a label with a constant, e.g. `foo equ 1`.



`IF <expr>|<label>`	If the expression or label is non-zero, then the statements between the `IF` directive and the next `ENDIF` or `ELSE` directive will be assembled; otherwise, they will be ignored. `IF` directives may be nested up to 16 levels deep.
`IFDEF <label>`	If the given label has been defined, then the statements between the `IFDEF` directive and the next `ENDIF` or `ELSE` directive will be assembled; otherwise, they will be ignored.
`ORG`	Set the assembler's target address, e.g. `org 0`. The directive has no effect on the contents of the C and assembler object files apart from altering the comments that show the target addresses.
`=`	Equivalent to `EQU`.

Constants, Expressions, and Operator Precedence

Numerical literals are decimal by default. Hexadecimal literals must be preceded by a dollar sign. Binary literals must be preceded by a percent sign.

Constants and labels may be combined in expressions using the following operators, which are listed in the order of their precedence.

`+, -, ~`	Unary plus, unary minus, and bitwise negation.
`*, /, %`	Multiplication, division, and modulus.
`+, -`	Addition and subtraction.
`<<, >>`	Left and right shifting.
`&`	Bitwise and.
`|, ^`	Bitwise or and exclusive or.

Notes

The DSP's program memory is limited to 256 words, but the assembler will assemble programs having up to 2048 words, and the DSP simulator will load and run them. This allows you to develop and debug your algorithms first and worry about program size later. If your program exceeds 256 words in length, a warning will be displayed. Use caution, however, since jumping to destinations beyond the 256-word boundary will not work, because the destination field in a `JMP` command is only eight bits wide.

The assembler stops as soon as it encounters an error, so it never generates more than one error message.

Sample Programs

1. Copying a block of memory from RAM0 to RAM1.

```
block_size =     12                ;  Number of words to be copied.
ram0_index =     0                 ;  Index to start of source data in R0.
ram1_index =     0                 ;  Index to destination buffer in R1.

           mov   ram0_index,ct0    ;  Load index register for R0.
           mov   ram1_index,ct1    ;  Load index register for R1.
           mov   block_size-1,lop  ;  Initialize loop counter.
           lps                     ;  Repeat next instruction.
           mov   mc0,mc1           ;  Move a word from R0 to R1, auto-
           endi                    ;    incrementing the index registers.
```

2a. Calculating (2x3) + (4x5) without parallelism.

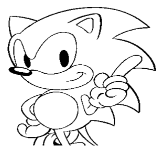
```
ram0_index =     0                 ;  Index to location of 2 and 4 in R0.
ram1_index =     0                 ;  Index to location of 3 and 5 in R1.
ram2_index =     0                 ;  Index to location of result in R2.

           mov   ram0_index,ct0    ;  Load index register for R0.
           mov   ram1_index,ct1    ;  Load index register for R1.
           mvi   #2,mc0            ;  Store 2 in the first word of R0.
           mvi   #3,mc1            ;  Store 3 in the first word of R1.
           mvi   #4,mc0            ;  Store 4 in the second word of R0.
           mvi   #5,mc1            ;  Store 5 in the second word of R1.
           mov   ram0_index,ct0    ;  Reset R0 index.
           mov   ram1_index,ct1    ;  Reset R1 index.
           mov   ram2_index,ct2    ;  Load index to destination buffer in R2.
           mov   mc0,x             ;  Load the 2 into the RX register.
           mov   mc1,y             ;  Load the 3 into the RY register.
           mov   mul,p             ;  Move the product into the P register.
           mov   mc0,x             ;  Load the 4 into the RX register.
           mov   mc1,y             ;  Load the 5 into the RY register.
           clr   a                 ;  Clear the accumulator.
           ad2   mov alu,a         ;  Add the P register to the accumulator.
           mov   mul,p             ;  Move the second product into P.
           ad2   mov all,mc2       ;  Add the two products and store the
           endi                    ;    result in R2.
```

2b. Calculating (2x3) + (4x5) with parallelism.

```
ram0_index =     0                 ;  Index to location of 2 and 4 in R0.
ram1_index =     0                 ;  Index to location of 3 and 5 in R1.
ram2_index =     0                 ;  Index to location of result in R2.

                                                          mov  ram0_index,ct0
                                                          mov  ram1_index,ct1
          mvi  #2,mc0
          mvi  #3,mc1
          mvi  #4,mc0
          mvi  #5,mc1
```



```
                                                                  mov  ram0_index,ct0
                                                                  mov  ram1_index,ct1
               mov  mc0,X             mov  mc1,Y                  mov  ram2_index,ct2
               mov  mc0,X  mov  mul,p mov  mc1,Y  clr  a
          ad2              mov  mul,p             mov  alu,a
          ad2                                                     mov  all,mc2
          endi
```

3. Multiplication of a 4x3 matrix by a 4-element vector.

```
;  Perform a typical 3D point-transformation calculation:
;
;      | m00 m01 m02 m03 |   | x0 |     | x1 |
;      | m10 m11 m12 m13 | * | y0 |  =  | y1 |
;      | m20 m21 m22 m23 |   | z0 |     | z1 |
;                            |  1 |
;
;  We assume that the matrix is already present in RAM0.  The vector will be
;  loaded into RAM1 using DMA.  The resultant vector will be stored in RAM2.
;
;  The address of the input vector in external memory is divided by four,
;  because addresses in the DMA read register (RA0) get multiplied by four
;  before being sent out onto the system bus.

;
vector_adr =   $10000 >> 2
in_mat_r0  =   0
in_vec_r1  =   0
out_vec_r2 =   0
;
;  Transfer the x0, y0, and z0 from external memory to RAM1.
;
          mvi  #vector_adr,ra0
                                                                  mov  in_vec_r1,ct1
          dma  d0,mc1,#3
;
;  Wait for the transfer to complete.
;
dma_wait:  jmp  t0,dma_wait
;
;  Initialize the data RAM index registers.
;
                                                                  mov  in_mat_r0,ct0
                                                                  mov  in_vec_r1,ct1
;
```

```
;  Start computing sums of products and storing the results.
;
             mov mc1,x           mov mc0,y           mov out_vec_r2,ct2
             mov mc1,x mov mul,p mov mc0,y clr a
         ad2 mov mc1,x mov mul,p mov mc0,y mov alu,a mov in_vec_r1,ct1
         ad2           mov mul,p mov mc0,y mov alu,a mov #1,rx
         ad2 mov mc1,x mov mul,p mov mc0,y mov alu,a
         ad2 mov mc1,x mov mul,p mov mc0,y clr a     mov all,mc2
         ad2 mov mc1,x mov mul,p mov mc0,y mov alu,a mov in_vec_r1,ct1
         ad2           mov mul,p mov mc0,y mov alu,a mov #1,rx
         ad2 mov mc1,x mov mul,p mov mc0,y mov alu,a
         ad2 mov mc1,x mov mul,p mov mc0,y clr a     mov all,mc2
         ad2 mov mc1,x mov mul,p mov mc0,y mov alu,a
         ad2           mov mul,p mov mc0,y mov alu,a mov #1,rx
         ad2           mov mul,p           mov alu,a
         ad2                                         mov all,mc2
         endi
```