516-14
ADH,CC
1/16/69

GMAP Assembler for the Multi-Programmed 516

A GMAP assembler is available for assembling DDP-516 segmented programs (i.e., relocatable programs stored on disk which are called into core upon demand). The assembler output includes a symbolic listing, an octal listing, and a binary deck which may be loaded into the DDP-516 through the card reader. The deck set-up for running an assembly follows:

```
1       8         16
$       IDENT     [usual information]
$       SELECT    GEDISK/DDP-516/SEGMENT/GMAP
[symbolic deck]
$       SELECT    GEDISK/DDP-516/SEGMENT/POSTPROC
$       ENDJOB
```

The assembly requires two activities (GMAP, and the post processor). Several programs may be assembled in one run; the $ SELECT cards must be used with each deck.

The user must give a segment name to his program by calling SNAME:

```
1       8         16
        SNAME     name
```

where name is the segment name, which consists of 1-6 characters, taken from the alphabetics and period.

The 516 "unadulterated" memory-reference instructions are coded with a dot in front of the standard mnemonic. Examples:

```
1       8         16
        .JMP      X
        .LDA      Y,1
        .ANA      Z,*
        .STA      3,1*
```

The assembled format is like that of a GE memory-reference instruction. The 4-bit operation code is in bits 22-25, the tag bit position (part of the code for .LDX and .STX) is bit 21, and the flag bit position (bit 20) is always 0. Bits 18-19 are 10_2, a post-processor flag.

The segmented-program writer will normally use the 516 mnemonics without dots. These codes are defined as MACRO-instructions that compile into a variety of sequences, depending on the variable field contents. Examples of these various cases follow. Notice that STX and LDX are not available, since the index register is not available in segmented programs. Also, JST is a special case because it causes a physical address to be stored inside the program, in violation of segmented-program rules. The tag (tase register) bit is controlled by the following rule: The tag is set if and only if the address is relocatable. To help define addresses properly, the symbol .RZERO is defined as relocatable 0.

```
1      8       16
       JMP     A[,*]
```

where A is absolute or relocatable (e.g., a transfer of control inside the segment). The assembled sequence is just the machine instruction:

```
1      8       16
       .JMP    A,[*]
```

The GOTO mnemonic may be used interchangeably with JMP.

```
1      8       16
       LDA     B[,*]
```

where B is an external symbol, assembles into a subroutine call for address computation followed by the corresponding machine instruction. The computed physical address is stored in .SRPO. The direct and indirect reference cases call two different subroutines:

```
1      8       16
       .JST    .OPDAD,*  )
       VADDR   B         } direct address
       .LDA    .SRPO,*   )
```

or

```
       .JST    .OPISW,*  )
       VADDR   B         } indirect address
       .LDA    .SRPO,*   )
```

In the indirect-address case, the address (in location B of the foreign segment) is assumed to be a single-word address, either an intra-segment reference or a pointer into common storage (sector 0).

For an indirect reference in which the indirect address is a virtual-address pair, the same code is used regardless of where the indirect address is located. Thus,

```
1      8        16
       ANA      X,*VA
```

assembles into

```
1      8        16
       .JST     .OPIVA,*
       VADDR    X
       .ANA     .SRPO,*
```

regardless of whether X is absolute, relocatable, or external.

In the above examples, VADDR is a MACRO which generates a virtual-address pair of words, regardless of the address type. This MACRO is available to the applications programmer directly. The other address-generating MACRO is:

```
1      8        16
       ADDR     X
```

If X is absolute or relocatable, the result is a single word, with the tag bit 0 or 1, respectively. If X is external, the result is a virtual address pair (identical to the use of VADDR).

The MACRO EXTERN must be used to define an external symbol prior to its first use in an address:

```
1      8        16
       EXTERN   segment,(name1,offset1,...,name8,offset8)
```

The first argument is the name of the external segment. This segment name is entered on the program card. The segment name may appear in the address field of some later instruction, where it is interpreted as the first location in the external segment (offset or relative address of 0). If X is such an external segment name, then X+5 is a legal address, the sixth location in segment X. The optional argument following the segment name defines additional locations in the external segment. Thus, "name1" is the name of location "offset1" in the external segment, and so on. A maximum of 8 such auxiliary names may be defined by one EXTERN statement. These names are not global, in that they do not appear on the program card, and thus are known only inside the segment which defines them in the above manner.

The JST instruction is only to be used for intra-segment references. It may be used either in the direct-addressing mode or for 1-level indirect addressing, in which case the indirect address is single word and not in an external segment.

An ASCII macro has been defined which converts GE to ASCII. The symbolic format is:

```
1      8        16
       ASCII    (ab,cd,...)
```

where a goes into the left half of the first word, d goes into the right half of the second word. The format for ASCII literals is:

```
1      8        16
       LDA      =Hab
```

To call a subroutine either internal or external to the calling segment the following statement is used:

```
1      8        16
       CALL     (SUBR,ARG1,ARG2,...)
```

where "SUBR" is the subroutine name and "ARG1", "ARG2",... are address arguments to be passed to the called subroutine. The subroutine name and the address arguments may be defined internal to the calling segment or be defined as external to calling segment by an EXTERN statement.

The called subroutine can fetch the address arguments using:

```
1      8        16
       GETA     N
```

where N is the number of the argument to be fetched (N=1 fetches the first argument). Upon execution of a GETA statement three items are returned. .ADPTR contains a pointer to the requested address argument in the calling sequence. The requested address argument (2 word virtual address) in the calling sequence is converted to an absolute core address and left in .ADARG. The contents of the memory location pointed to by the address argument are left in the A register.

If another level of indirection is required (.ADARG points to another virtual address),

```
1      8
       ADCØNV
```

fetches the virtual address pointed to by .ADARG, converts it to an absolute core address and returns with the converted address in .ADARG. Any level of indirection can be followed by repeated ADCØNV calls. To return from a called subroutine, use

```
1      8       16
       RETURN  N
```

where N is 1 + the number of arguments in the calling sequence (if there were no arguments use RETURN 1).

Although CALL can be used to call internal subroutines it is inefficient. For internal subroutines the following set of statements can be used

```
1      8       16
       JCALL   SUBR
```

where "SUBR" must be internally defined by:

```
1      8
SUBR   JSUBR
```

This pair of statements is like using the 516 machine instruction JST but in a reentrant sense. When the subroutine has been entered the system location .JSTAD points to the memory location after the JCALL statement. It can be incremented with an IRS and used to fetch arguments just like a JST-planted address.

```
1      8
       JRETRN
```

is used to return from a JCALLed subroutine (.JSTAD must be left pointing to where control should be returned.)

Each thread in the multiprogramming system has associated with it 8 relocatable pointers (.RPO-.RP7). An absolute core address (pointer) can be deposited in one of these pointers and the system will automatically relocate it when every core shifts. Thus a segment may be accessed for data, etc. indirectly thru a relocatable pointer with no system overhead time. Loading a relocatable pointer with a pointer to a segment also has the effect of holding the segment in core. For this reason a zero address should be loaded into a relocatable pointer when the user is finished using it, to allow the segment to be pushed out of core if necessary. The indirect bit cannot be set in a relocatable pointer.

```
1      8       16
       RELPTR  .RPN
```

is used to set up one of the 8 relocatable pointers (.RP0-.RP7). The A register is assumed to contain an absolute core address to be planted in the relocatable pointer. The old contents of the pointer are destroyed, with the appropriate book-keeping done.

Sometimes it is desirable to write a non-reentrant subroutine or part of a subroutine. The following statement should precede any non-reentrant code.

```
1       8
A       GATE
```

This statement allows only one thread to pass this point. After the non-reentrant code has been executed, the gate should be reopened for any other threads by storing a zero in location A+1.

The binary card format for relocatable programs uses the following packing strategy:

column			
4k-3	top 4 bits of word 3k	top 4 bits of word 3k-1	top 4 bits of word 3k-2
4k-2	bottom 12 bits of word 3k-2		
4k-1	bottom 12 bits of word 3k-1		
4k	bottom 12 bits of word 3k		

The card format is based on the unpacked words, not columns:

Program Card

word					
1	1	1	1	1	word count (start at word 4)
2	segment length				
3	- checksum (2's complement)				
4	0 1	first program card: total number of names continuation: number of first name on this card			
5 54	1-25 names (word pairs, format as in 516-5)				

Note: First name on first program card is name (SNAME) of this segment; all other names are EXTERN names.

Data Card

word					
1	0	1	1	1	word count (start at word 4)
2	origin of this card				
3	- checksum (2's complement)				
4 6	1-48 type bits: 0 - data word 1 - index into EXTERN names				
7 54	1-48 data words				