PDP-K Technical Memoranda # 2

Title:	Extension of PDP-11 Instruction Set
Author(s):	Ad van de Goor
Index Keys:	Instruction Sets Opcode Space Modes Stack Operations
Distribution Keys:	K
Revision:	None
Obsolete:	None
Date:	21 January 1970

0.0 ABSTRACT

Several methods of extending the PDP-11 instruction set are discussed. Coding comparisons are made. Subject to the trivial weighting scheme used, two solutions were excluded from further analysis because of their poor performance. The "multiply/divide" subsolution as discussed in sections 4.4 and 5.4 was the best performer.

1.0 INTRODUCTION

A more elaborate version of the PDP-11/20 is considered as a possible candidate for the PDP-K. It is felt that if the PDP-K is a member of the PDP-11 family, substantial gains could be obtained from:

1.1 Upwards Program Compatibility

For DEC this would mean a lower total software investment, and new machines could be introduced more easily as present PDP-11 software would run on PDP-K.

For customers this would mean that they could move to a larger machine without the direct need for reprogramming.

1.2 Peripheral Compatibility

Only one line of peripheral devices has to be built. The introductions of a new machine could be done more easily for this reason. Any new peripheral device would be available for the whole family.

2.0 PROBLEMS IN ADAPTING THE PDP-11 ARCHITECTURE TO A BIGGER MACHINE

Two important problems of the PDP-11 have to be solved in order to meet the PDP-K requirements.

2.1 Limited number of instructions and limited amount of opcode space left. For the PDP-K three more classes of instructions are considered:

2.1.1 EAE instructions, i.e., rotate/shift and multiply/divide for 16-bit words.

2.1.2 Double Precision Integer Arithmetic Instructions.

2.1.3 Floating Point Arithmetic Instructions.

2.2 Limited Address Space

The total amount of addressable core memory on the PDP-11/20 is 65K (1K = is 1024) bytes, or 32K 16-bit words. For a big 32-bit version of the PDP-11 this would only mean 16K 32-bit words could be addressed, which is certainly not adequate for such a machine.

3.0 PURPOSE OF MEMORANDUM

The purpose of this memorandum is to examine the suggested methods of solving the first problems: extending the basic PDP-11 instruction set. An acceptable solution, subject to several constraints, will be sought.

3.1 Program compatibility at least on the assembly language level.

3.2 Simplicity in programming by minimizing the number of instruction formats and restrictions imposed on instructions.

3.3 Opcode space left for future expansion.

3.4 Opcodes of the largest member of the family have to fit in the added instruction set, thus minimizing the number of formats, and making programming easier.

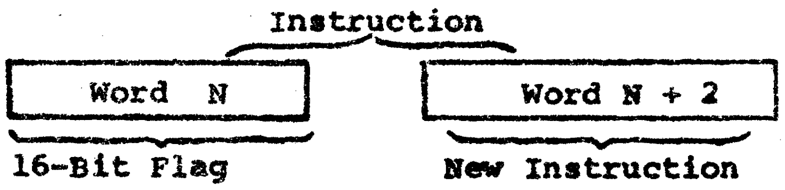
4.0 POSSIBLE SOLUTIONS

Four possible solutions to the opcode space problem are shown below. They are followed by a discussion in section 5:0.

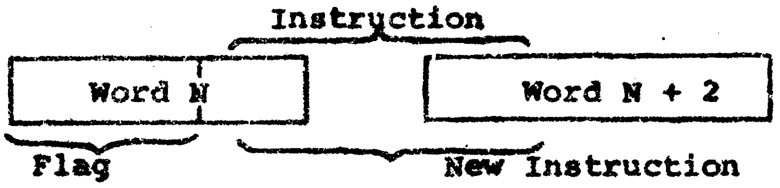
4.1 Implement new instructions as "pure stack" instructions (i.e., zero address). Each new instruction can now be specified with one combination out of 2^{16}. This allows for hundreds of new instructions. Any binary operation (like multiply, divide, etc.), would take the two operands from the top of the stack, and leave the result on the top of the stack. Register 6 would be used as the implied stack pointer.

4.2 Introduce a flag to indicate that the remainder of the word containing the flag (note: remainder can be = 0) and the next word form a new instruction. Depending on the length of the flag, two cases exist.

4.2.1 Full Word Flag



4.2.2 Partial Word Flag



The advantage of this technique is that the new instructions can have the same source-destination format as the standard (i.e., currrent PDP-11/20), instructions.

The disadvantage is that every new instruction takes two words. The partial word flag case offers the advantage of a greater number of new instructions at the expense of somewhat more complicated hardware.

4.3 Modes

A mode is a (hardware) state of the processor to allow instructions to be interpreted differently. Basically two kinds of modes have to be recognized:

4.3.1 Enter and leave modes only with dedicated commands (i.e., only switch modes when an instruction specifies to do so).

4.3.2 Enter modes for a specified number of instructions after which the mode is switched back to the standard mode automatically.

The advantage of modes is that instructions in any mode are only 1 word long. The disadvantage is that special instructions have to be given to enter, and in the case of 4.3.1, to leave the mode.

4.4 Use Reserved Multiply/Divide Space

These two opcode spaces are not used in the PDP-11/20. The to-be-added two-operand instructions can be implemented as source-destination instructions where the stack is one implied operand, and the second operand is specified with the full 6-bit destination field of the instruction. One of these 6 bits can be used as a direction bit such that operations can have either their source or destination as the implied stack. This allows for 32 new instructions to be specified.

5.0 EVALUATION OF PROPOSED SOLUTIONS

When evaluating the proposed solutions, the implementation of a 32-bit version of the PDP-11 should be included. For such a machine, double-precision floating point instructions, together with EAE instructions, operating on 32-bit registers are desirable, (assuming that these instructions can operate on registers). This means that opcode space for those instructions has to be reserved to provide for their efficient operation.

Simplicity in programming and machine organization dictate that the number of instruction formats for the three classes of new instructions, (as discussed in section 2.1), should be minimal. In order to make the extended instruction set more acceptable, it is very desirable to make the added instructions fit in currently existing formats, or add at most a single new format. Several coding comparisons are done to assist in the evaluation. The five problems below (P1-P5), are considered representative. The assumptions made in coding the problems can be deduced from the listed code in Appendixes A-D. The variables A, B, C, D and E are considered single precision floating point (32-bit numbers).

P1:	$A \leftarrow B*C$	/simple case
P2:	$A \leftarrow (B+C)*(D+E)$	/temporary variable case
P3:	$A(i) \leftarrow B(i)*C(i)$	/subscripted case
P4:	$A(i) \leftarrow B(i+3) *C(i*5)$	/mixed arithmetic case
P5:	$A(i,j) \leftarrow A(i,j)+B(i,k)*C(k,j)$	/multi-dimensional array case

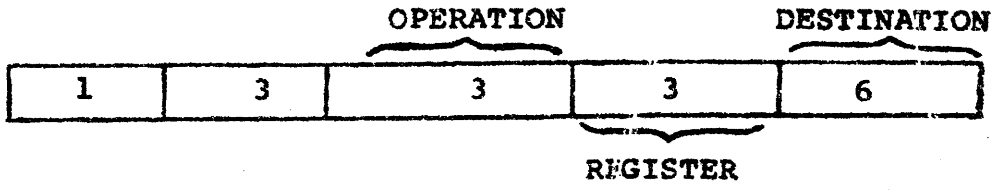
P5 is an example of the inner-loop statement of the array multiplication: $[A] \leftarrow [B] * [C]$ It is assumed that the array bounds are declared from o to u. For array B this would be: Real Array B (0 - bul, 0 - bu2). The first index of B goes to bul, the second to bu2. It will be assumed that the indexes are in registers Ri, Rj, and Rk.

Assuming that the indexes i and j are in register Ri and Rj, the value B (i,j) will be address as follows: Location of B(i,j) = location of B (i.e., starting location of matrix) + i*bul+j).

5.1 Pure Stack Operations

In order to make the pure stack operations efficient, one of the opcode spaces reserved for multiply/divide has to be used for a double move (MOVD) instructions.

MOVD:Move 2 words (32 bits) from S(ource) D(estination). This instruction is required especially in a 32-bit machine. The one binary opcode space left can be used to implement the EAE instructions.[1] The instruction format would be as follows:



This same format is used for the JSR (subroutine call) instruction. The EAE instructions are made to operate on registers only. The register involved is specified by the 3 "register" bits.

The value of the effective address of the "destination" determines the number of positions to be shifted or rotated. Because the auto-increment and auto-decrement modes do not apply to these instructions, one of the 2 mode bits can be used to specify a single or combined operation, (i.e., see PDP-10 LSH, LSHC, etc.). The remaining space can be used to implement instructions like EXCHANGE, REPEAT, etc.

Appendix A gives the coding examples for the five problems. The handling of multi-dimensional arrays is very cumbersome because the address computations have to be done on the stack. Introducing a second set of 16-bit multiply/divide instructions implemented as the above EAE instructions will solve this problem at the expense of a more complex instruction set. Subcolumn Table 1 MPD of Section 6 shows the improvement gained by this.

[1]Except for 16-bit multiply/divide

5.2 Flagged Instructions

The coding examples shown in Appendix B are the same for alternatives 4.2.1 and 4.2.2. 4.2.2 Is preferable only if the additional opcode space is needed. It is suggested that the EAE multiply/divide instruction will be implemented in the space "reserved" for them. The EAE rotate/shift instructions have to be implemented as "flagged" instructions, the format would be similar to that discussed in section 5.1., except for the flag. The double precision integer and floating point instructions would be implemented as full source-destination instructions.

5.3 Modes

Before going into detail, proposal 4.3.2 (setting the modes for a specific number of instructions (N)), will be examined. This is considered less attractive because of problems arising in a string of N[1] instructions to be executed in the new mode.

5.3.1 Branching in terms of skipping over a group of instructions in the specified string will cause problems because N is not updated automatically.

5.3.2 Programming will be very difficult because when branching into a sequence of instructions their mode, (in which those operate), will be difficult to determine.

5.3.3 It will be difficult for a compiler to set up the right "N" because it will require some kind of "look-ahead".

5.3.4 In case of interrupts/traps, the remainder of N has to be saved and restored upon exit of the interrupt/trap service routine.

[1]Where N is an arbitrary positive number.

For the reasons above, proposal 4.3.2 will be dropped, and not considered further.

The extended mode, (which contains the floating, double-precision integer instructions, etc.), is entered by the command Enter Extended Mode (EEM). The processor stays in this mode until the instruction Leave Extended Mode (LEM) is given.

In regard to 4.3.1, subroutine calls and interrrupt/traps cause problems typical for modes in saving/restoring the mode and entering the routine (subroutine or interrupt/trap service routine), in the correct mode. The interrupt/ trap case is the easiest one. The mode can be preserved in a dedicated bit in the Central Processor Status Register(PS). Entering the interrupt/trap service routine in the right mode can be done similarly by storing the mode of that routine in the PS interrupt/trap vector. The correct mode will then be entered automatically upon interrupt.

Entering a subroutine in the desired mode in a program compatible way can be done by taking the lowest bit (bit 0) of the subroutine address as the mode bit. In the current PDP-11/20, this bit has to be equal zero because the subroutine address is a word address. By defining a "0" in bit 0 of the subroutine address as the standard mode, program compatibility is preserved.

Saving/restoring the mode upon a subroutine call/ exit is much more difficult. The only hardware solution found thus far is to store the mode on the stack in a separate word. The new JSR would then store 2 words on the stack: the register to be saved and the mode. Programs making use of the knowledge that only 1 word gets stored on the stack by a JSR have to be modified.

A program compatible software solution to the mode problem is to have the called subroutine take care of the mode handling by restoring the mode (upon exit), which existed prior to the call of the subroutine. A possible way of doing this is by having the existing mode, prior to all calls for a given subroutine, fixed, such that the subroutine only has to match the mode upon exit to the existing (fixed) mode at call time.

It is suggested that the multiply and divide instructions, (operating on 16-bit integers), be implemented in the space reserved for them, and all other instructions be implemented in the extended mode.

Appendix C shows the coding examples. They suggest that an instruction to enter the extended mode for a single instruction is very useful. The column EEM1 (Enter Extended Mode for 1 Instruction), of Table 1, Section 6, shows this.

5.4 Use Mutliply/Divide Space

One of the two binary opcode spaces has to be used to implement the EAE instructions as described in section 5.1. The remaining instructions have to implemented with the stack as an implied operand as discussed in section 4.4. Coding examples are given in Appendix D. They show, like the "pure stack" case, that handling multi-dimensional arrays is cumbersome. The improvements made by adding a set of 16-bit multiply/divide instructions, as suggested in section 5.1, are shown in subcolumn MPD of Table 1, Section 6.

6.0 COMPARISON OF PROPOSED SOLUTIONS

Table 1 shows the results of the five problems for the seven[1] proposed solutions. Four quantifiers are used for each problem to measure the quality of the solutions.

6.1 The Number of Instructions

It is quite well known that the probability of making a programming error increases more than linear with the number of instructions, (apart from their complexity), thus a "good" solution should have a low number of instructions.

6.2 The Number of Words[2]

This is the number of words needed to core the algorithms given in the appendixes. This is an important criterium, especially on a small machine. For a 32-bit machine the numbers have to be divided by 2.

6.3 The Number of Memory References

The number of memory references both for a 16 and 32-bit machine are included in the tables because they are important indicators for the execution times of the algorithms. The numbers in Table 1 are derived under the following assumptions:

6.3.1 The stack is supposed to be in core-memory. (Section 6.4 discusses the results when this assumption is not made).

6.3.2 For the two operand extended instructions the arithmetic unit is supposed to behave as follows: 1) reads both operands into its internal registers; 2) it performs the required operation (e.g. FMUL, FADD); and 3) it stores the results back. In case of different assumptions the numbers in the table can be adjusted accordingly.

[1]Four main solutions, three of which have a subsolution.
[2]Words are considered to be 16 bits long.

6.4 Number of Memory References With A Hardware Stack

The idea is to implement the top M[1] words of the stack in flip-flop registers. From Table 1 it can be seen that the execution speed increases for almost all problems and solutions. Those solutions making heavy use of the stack gain most.

[1]For simplicity M is supposed to be such that in none of the problems the stack "overflows" into core.

TABLE 1 - CODING RESULTS OF PROBLEMS P1 + P5

PROBLEM NUMBERS	QUANTIFIER	PURE STACK		FLAG	MODE		MULTIPLY/DIVIDE	
			MPD			EEM[1]		MPD
1	# of Instructions	4	4	2	4	4	3	3
	# of Words	7	7	8	8	8	6	6
	# of Memory Ref	25/12.5[1]	25.12.5	18/9	18/9	18/9	20/10	20/10
	# of Memory Ref With Hardware Stack	13/6.5[1]	13/6.5	18/9	18/9	18/9	12/6	12/6
2	# of Instructions	8	8	5	7	7	6	6
	# of Words	13	13	17	14	14	11	11
	# of Memory Ref	51/25.5	51/25.5	43/21.5	40/20	40/20	41/20.5	41/20.5
	# of Memory Ref With Hardware Stack	23/11.5	23/11.5	35/17.5	32/16	32/16	21/10.5	21/10.5
3	# of Instructions	4	4	2	4	4	3	3
	# of Words	7	7	8	8	8	6	6
	# of Memory Ref	25/12.5	25/12.5	18/9	18/9	18/9	20/10	20/10
	# of Memory Ref With Hardware Stack	13/6.5	13/6.5	18/9	18/9	18/9	12/6	12/6
4	# of Instructions	10	8	6	10	8	8	7
	# of Words	15	13	14	16	14	13	12
	# of Memory Ref	39/22.5	31/15.5	24/12	26/13	24/12	31/16.5	26/13
	# of Memory Ref With Hardware Stack	21/10.5	19/9.5	24/12	26/13	24/12	19/9.5	16/8
5	# of Instructions	21	15	12	18	15	16	13
	# of Words	28	22	21	24	21	23	20
	# of Memory Ref	74/46	46/23	37/18.5	40/20	37/18.5	55/30.5	40/20
	# of Memory Ref With Hardware Stack	36/18	28/14	29/14.5	32/16	29/14.5	31/15.5	28/14

Table 2 gives a rating summary of Table 1, the rating is from 1 (lowest), to 7 (highest). When two solutions have equal rating, they both get the same number being the average rating when they would not have been equal.

The problems P1 - P3 are very similar in nature, therefore a summarized rating is given in the first part of Table 2. Similarly, for P4 - P5 in the second part of Table 2. The third part of Table 2 is a summary of the previous two tables assuming equal weights for the two previous groups of problems. Part 4 of Table 2 is merely the sum of the first two quantifiers of the third part.[1] For a small machine, the number of instructions and the number of words are the most important criteria for selecting the best solution. On a bigger machine, execution speed is becoming important. Part 5 of Table 2 is such an indicator. Its entries are the sums of the first, second, and fourth quantifiers of part 3. It is assumed that on the bigger machine the top of the stack is implemented in hardware.

[1]Again here, for simplicity reasons, equal weights are assumed.

TABLE 2 - RATING SUMMARY OF CODING PROBLEMS

PART PROBLEMS	QUANTIFIER	PURE STACK		FLAG	MODE		MULTIPLY/DIVIDE	
			MPD			EEM1		MPD
1 P1 - P3	# of Instructions	1.5	1.5	7	3.5	3.5	5.5	5.5
	# of Words	4.5	4.5	1	2.5	2.5	6.5	6.5
	# of Memory Ref	1.5/1.5	1.5/1.5	5/5	6.5/6.5	6.5/6.5	3.5/3.5	3.5/3.5
	# of Memory Ref With Hardware Stack	4.5/4.5	4.5/4.5	1/1	2.5/2.5	2.5/2.5	6.5/6.5	6.5/6.5
2 P4 - P5	# of Instructions	1	4.5	7	2	4.5	3	6
	# of Words	1	5	5	2	5	3	7
	# of Memory Ref	1/1	3/3	6.5/6.5	4.5/4.5	6.5/6.5	2/2	4.5/4.5
	# of Memory Ref With Hardware Stack	2/2	6/6	3.5/3.5	1/1	3.5/3.5	5/5	7/7
3 P1 - P5	# of Instructions	2.5	6.0	14	5.5	7.5	8.5	11.5
	# of Words	5.5	9.5	6	4.5	7.5	9.5	13.5
	# of Memory Ref	2.5/2.5	4.5/4.5	11.5/11.5	11/11	13/13	5.5/5.5	8/8
	# of Memory Ref With Hardware Stack	6.5/6.5	10.5/10.5	4.5/4.5	3.5/3.5	6/6	11.5/11.5	13.5/13.5
4	# of Instructions + Number of Words	8	15.5	20	10.0	15.0	18.0	25
5	# of Memory Ref With Hardware Stack	14.5	26	24.5	13.5	21.0	29.5	38.5

7.0 CONCLUSION

Looking at Table 2, part 4 and 5, it can be consluded that the subsolutions, (i.e., MPD for "pure stack" and "multiply/divide", and EEM1 for "mode"), are a big improvement over their "main" solutions. This, because of the improved handling of multi-dimensional arrays, the price paid for this, however, is a more complex instruction set (i.e., adding a duplicate set of 16-bit multiply/divide instructions to operate on register or enter the extended mode for a single instruction)..

The main solutions "pure stack" and "mode" have the lowest rating and can therefore be excluded from further consideration.

In order to make a definite commitment to any of the remaining five solutions, more research should be done in determining the weights of the problems and weights of the quantifiers.

From the results, this far however, the following can be said:

7.1 The "mode" subsolution has to look much better[1] in order to be a candidate because of the mode problems in subroutines. The suggested hardware solution is such that the price of storing the mode on the stack has to be paid ALWAYS. Also, in programs which do not make use of the mode, (i.e., all current PDP-11 software). For this reason the suggested software solution is a better candidate because there, the price is only paid when modes are used.

[1]When the proper weights are found.

7.2 The "flag" solution is advisable only when it is expected that the use of the "flagged" instructions (i.e. those of class 2.1.2 and 2.1.3 of section 2) is low.

7.3 The most promising solution this far is the "multiply/divide" subsolution. It consistently scored highest or second highest

APPENDIX A

PURE STACK CODING EXAMPLES

```
P1:  A ←——— B*C

     MOVD       C, - (SP)           /move C to the stack
     MOVD       B, - (SP)           /move B to the stack
     FMUL                           /floating multiply B*C
     MOVD       (SP)+,A             /store result in A

P2:  A ←——— (B+C)*(D+E)

     MOVD       B, -.(SP)
     MOVD       C, - (SP)
     FADD                           /floating add B+C
     MOVD       D, - (SP)
     MOVD       E, - (SP)
     FADD                           /floating add D+E
     FMUL                           /floating multiply (D+E)*(B+C)
     MOVD       (SP)+,A

P3:  A(i) ←——B(i)*C(i)              /assume index i is in register Ri

     MOVD       C(Ri), - (SP)       /move C(i) to the stack
     MOVD       B(Ri), - (SP)
     FMUL
     MOVD       (SP) +, A(Ri)

P4:  A(i ←——B(i+3)*C(i*5)           /Rs is a scratch register

     MOV        Ri, Rs
     ADD        #3, Rs              /index i+3 formed
     MOVD       B(Rs), - (SP)
     MOV        Ri, - (SP)
     MOV        #5, - (SP)
     IMUL                           /compute i*5 and leave 1 word result
                                     on top of stack
     MOV        (SP)+, Rs
     MOVD       C(Rs), - (SP)
     FMUL
     MOVD       (SP)+, A(Ri)        /store result
```

APPENDIX A (CONT.)

P5: $A(i,j) \leftarrow A(i,j)+B(i,k)*C(k,j)$

```
MOV        Ri, - (SP)
MOV        #bul, - (SP)
IMUL
MOV        (SP)+, Rs
ADD        Rk, Rs               /Rs contains index for array B
MOVD       B(Rs), - (SP)        /put B(i,k) on stack
MOV        Rk, - (SP)
MOV        #cul, - (SP)
IMUL
MOV        (SP)+, Rs
ADD        Ri, Rs               /Rs contains index for array C
MOVD       C(Rs), -(SP)
FMUL
MOV        Ri, - (SP)
MOV        #aul, - (SP)
IMUL
MOV        (SP)+, Rs
ADD        Rj, Rs               /Rs contains index for array C
MOVD       A(Rs), - (SP)
FADD
MOVD       (SP)+, A(Rs)         /store result
```

APPENDIX B

FLAGGED INSTRUCTIONS CODING EXAMPLES

```
P1:  A ◄──────────── B*C

     MOVD           B,A                /move B to A
     FMUL           C,A

P2:  A ◄──────────── (B+C)*(D+E)

     MOVD           B,A
     FADD           C,A                /A = B+C now
     MOVD           D,-(SP)
     FADD           C,(SP)             /top of the stack is C+D
     FMUL           (SP)+,A

P3:  A(i) ◄────────── B(i)*C(i)

     MOVD           B(Ri), A(Ri)       /move B(i) to A(i)
     FADD           C(Ri), A(Ri)

P4:  A(i) ◄────────── B(i+3)*C(i*5)    /Rs is a scratch register

     MOV            Ri, Rs
     ADD            #3, Rs             /index for B(i+3) computed
     MOVD           B(Rs), A(Ri)
     MOV            Ri, Rs
     MUL            #5, Rs             /index for C(i*5) computed
     FMUL           C(Rs), A(Ri)

P5:  A(i,j) ◄──────── A(i,j) + B(i,k) * C(k,j)

     MOV            Ri, Rs
     MUL            #bul, Rs
     ADD            Rk, Rs             /index for B(i,k) computed
     MOVD           B(Rs), - (SP)
     MOV            Rk, Rs
     MUL            #cul, Rs
     ADD            Rj, Rs             /index for C(k,j) computed
     FMUL           C(Rs), (SP)
     MOV            Ri, Rs
     MUL            #aul, Rs
     ADD            Rj, Rs             /index for A(i,j) computed
     FADD           (SP) +, A(Rs)
```

APPENDIX C

MODE CODING EXAMPLES

```
P1:  A  ◄────────────  B*C

     EEM                                  /enter extended mode
     MOVD          B,A
     FMUL          C,A
     LEM                                  /leave extended mode

P2:  A  ◄────────────  (B+C)*(D+E)

     EEM                                  /enter extended mode
     MOVD          B,A
     FADD          C,A
     MOVD          D,-(SP)
     FADD          C,(SP)
     FMUL          (SP)+,A
     LEM                                  /leave extended mode

P3:  A(i)  ◄────────  B(i)*C(i)

     EEM
     MOVD          B(Ri), A(Ri)
     FMUL          C(Ri), A(Ri)
     LEM

P4:  A(i)  ◄────────  B(i+3)*C(i*5)

     MOV           Ri,Rs
     ADD           #3, Rs
     EEM
     MOVD          B(Rs), A(Ri)
     LEM
     MOV           Ri, Rs
     MUL           #5, Rs
     EEM
     FMUL          C(Rs), A(Ri)
     LEM

P5:  A(i,j)  ◄─────  A(i,j) + B(i,k) * C(k,j)

     MOV           Ri, Rs
     MUL           #bul, Rs
     ADD           Rk, Rs                 /index for B(i,k) computed
     EEM
     MOVD          B(Rs), - (SP)
     LEM
     MOV           Rk, Rs
     MUL           #cul, Rs
     ADD           Rj, Rs                 /index for C(k,j) computed
     EEM
     FMUL          C(Rs), (SP)
```

APPENDIX C

MODE CODING EXAMPLES

P5: Cont.

```
        LEM
        MOV             Ri, Rs
        MUL             #aul, Rs
        ADD             Rj, Rs          /index for A(i,j) computed
        EEM
        FADD            (SP) +, A(Rs)
        LEM
```

[illegible]

MULTIPLY, SINGLE STACK CODING EXAMPLES

```
P1:  A ◄──────── B*C

     MOVD        B,-(SP)          /move B to the stack
     FMUL        C,(SP)           /multiply C with top of the stack
     MOVD        (SP)+,A          /move result to A

P2:  A ◄──────── (B+C)*(D+E)

     MOVD        B,-(SP)
     FADD        C,(SP)
     MOVD        D,-(SP)
     FADD        E,(SP)
     FMUL        (SP)+,(SP)
     MOVD        (SP)+,A

P3:  A(i) ◄───── B(i) * C(i)

     MOVD        B(Ri), - (SP)
     FMUL        C(Ri),(SP)
     MOVD        (SP)+,A(Ri)

P4:  A(i)◄────── B(i*3) + C(i*5)

     MOV         Ri, Rs
     ADD         #3, Rs           /index i+3 in Rs
     MOVD        B(Rs), -(SP)
     MOV         Ri, -(SP)
     IMUL        #5, (SP)
     MOV         (SP)+, Rs        /index i*5 in Rs
     FMUL        C(Rs),(SP)
     MOVD        (SP)+, A(Ri)

P5:  A(i,j) ◄─── A(i,j) + B(i,k) * C(k,j)

     MOV         Ri, - (SP)
     IMUL        #bul, (SP)
     MOV         (SP)+, Rs
     ADD         Rk, Rs           /index for B(i,k) computed
     MOVD        B(Rs), - (SP)
     MOV         Rk, - (SP)
     IMUL        #cul, (SP)
     MOV         (SP)+, Rs
     ADD         Rj, Rs           /index for C(k,j) computed
     FMUL        C(Rs), (SP)
     MOV         Ri, - (SP)
     IMUL        #aul, (SP)
     MOV         (SP)+,Rs
     ADD         Rj, Rs           /index for A(i,j) computed
     FADD        A(Rs), (SP)
     MOVD        (SP)+, A(Rs)
```