MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Machine Structures Group Memo 4

Memorandum MAC-M-163
June 11, 1964

A PROCESSOR DESIGN: MAP-1

by

Earl C. Van Horn

I INTRODUCTION

This note proposes a logical structure and an order code for a typical central processing unit of a multi-user computing system. The processor is considered as a unit that is logically distinct from the i/o equipment and the memory system. While this note discusses neither i/o activities nor memory structure, it does rely on certain characteristics of the memory-processor interface which have been outlined by the author and J. B. Dennis*.

II CRITERIA

The following three criteria have influenced the MAP-1 design.

Syllable-structured order code. For a given instruction set, the number of instruction bits per instruction can be reduced by designating the more frequently used instructions with a fewer number of bits. In fact, the maximum possible reduction is obtained in an order code in which every instruction, i, is encoded using n_i bits, where n_i satisfies the following relation.

$$-\log_2 \Pr(i) \leq n_i < -\log_2 \Pr(i) + 1$$

Here Pr(i) is the probability that the instruction i will be issued by the stored program.

In MAP-1, an attempt has been made to increase the efficiency of instruction encoding by making instructions and immediate data of variable length, while still maintaining reasonable simplicity in the decoding hardware. Instructions and immediate data are made of eight bit syllables. The shortest instruction is 1 syllable long, and the longest instruction consists of over 10 syllables.

Programmed addressing. It is anticipated that the vast majority of MAO-1 programs will be pure procedures. Moreover, MAP-1 programs will deal with a

*Van Horn, E.C., "An Approach to Dynamic Storage Allocation", Project MAC Memorandum MAC-M-115, Nov. 1, 1963.

*Dennis, J.B., "A Machine Structure for Dynamic Storage Allocation", Project MAC Memorandum MAC-M-137, Jan. 31, 1964.

segmented memory structure. Very little experience has been accumulated in the coding of pure procedures in a segmented memory structure, and there is reason to believe that conventional addressing schemes will only be partially applicable to this new environment. For example, it is difficult to decide whether indirect addressing should precede or follow indexing in these circumstances.

In MAP-1, the programmer issues instructions to build up the memory address in the memory address register, and then issues a "fetch from memory address register" instruction. This facility allows the programmers using MAP-1 to develop their own techniques for address computation with a minimum of bias from the hardware.

It is likely that after a few years of experience with MAP-1's programmed addressing feature, designers will be able to recognize and make automatic, in a suceeding generation of processors, a few of the more frequently used addressing sequences.

While programmed addressing tends to increase the number of instructions required to express a given algorithm, the use of a syllable-structured order code reduces the average number of bits per instruction to such an extent that the algorithm encoding efficiency of a MAP-1 program might compare favorably with that of say a 7090 program.

Variable-size, general-purpose registers. MAP-1 is intended for use in a multi-processor, multi-memory system with interlaced addressing. In such a shared-memory system, each processor can expect to spend some fraction of its time waiting in queues for memory service.

By designing the order code of each processor so as to reduce the number of memory references per algorithm executed, one reduces not only the number of opportunities for delay per algorithm but also the expected delay per opportunity for delay. Both of these reductions serve to increase the rate at which the system executes algorithms. Experience has shown that it is both possible and convenient to code algorithms in such a way that certain intermediate results are referenced much more frequently than others. It seems reasonable therefore to provide in the processor a certain amount of storage where temporary results can be deposited and read.

The "state word" of a processor is the data that the processor stores within itself from one instruction to the next. The above arguments persuade

us that MAP-1 should have a large state word. However, there are two reasons why the MAP-1 state word should not be arbitrarily large. First, whenever the processor suspends a program sequence to take up a new program sequence in a different sphere of protection, the old state word must be completely exchanged with the new state word. This exchange operation takes an amount of time that is proportional to the state word size. If idle state words are stored in the main memory, the exchange also consumes the memory-processor communication resource in proportion to the size of the state words. The second reason for desiring a small state word is that if a processor has a large state word, then the storage facilities within the processor that are necessary to hold this state word are not allocatable at electronic speeds. Each program sequence has available to it in its processor some "state word storage", which is useful to it, but to no other sequence. If the state word storage is too large, then some sequences will not be able to use all of it. This idle storage cannot be used by other sequences, and so constitutes a wasted system resource.

To summarize, small state words tax the memory-processor communication resource, while large state words make sequence switching cumbersome and contradict the basic rationale for an allocatable computing facility. Current technology indicates that a state word size of around 1000 bits is an appropriate compromise among the above factors.

We next consider the question of how the state word storage should be organized into registers so that the programmer can deal with the state word data in a convenient way. Some parts of the state word storage are so often used to hold specialized data that they can be organized into fixed length registers which have designated functions and are referenced in special ways. The usage of the other parts of the state word storage varies from problem to problem, and hence the organization of these parts ought to be as flexible as possible.

In MAP-1, a portion of the state word storage is arranged into specially-referenced registers. The remainder of the state word storage is organized into a "byte pool", consisting of a string of 64 8-bit bytes within which are designated 16 registers. Six of these registers are so frequently used for certain purposes, such as subroutine linkage, that they are assigned fixed lengths and fixed positions in the byte pool. These 6 registers consume 16 bytes from the pool. The remaining 10 registers are designated by the stored program to have arbitrary lengths and arbitrary positions within the 64 bytes of the pool.

Any register may be used as an accumulator, a temporary storage register, an index register, a counter, a program flag register, or a memory address register. The 6 fixed registers also have additional special properties, in that they are automatically referred to by certain instructions.

It is possible that, after a few years of experience with the variavle-size, general-purpose registers of MAP-1, designers will be able to arrive at a configuration of fixed-size, general-purpose registers which will satisfy the majority of programmers.

III. Detailed Description

This section describes in a diagramatic way the structure of the MAP-1 processor. The following topics are covered in the following order.

1. State word storage organization
2. A few typical instructions
3. Natural data formats
4. Immediate operands
5. Syllable types
6. Concise instruction tabulation
7. Description of selected instructions
8. A suggested assembly language
9. A programming example - matrix multiplication
10. A comparison with a 7095 program to perform an equivalent matrix multiplication.

The MAP-1 disign is incomplete in that consideration has not been given to i/o operations, inter-sphere communication, segment lock-out, and other topics which are closely associated with an over-all system design. Additional memoranda will be published to cover these topics as various properties of the MACS-1 (Multiple Access Computing System) become more definite.

The remainder of this memo is intended to be an introduction to the MAP-1 design, as well as a reference manual for those familiar with

the design. Its primary purpose is to introduce the philosophy and general flavor of the MAP-1 disign to MAC participants, and to stimulate discussion on the topic of processor design. The author bopes that interested persons will discuss MAP-1 with him, both to clarigy its details and to submit comments, criticisms, and suggestions.

The construction of this order code has been influenced by the author's personal opinions and habits. No doubt the design contains some shortcomings. These shortcomings can best be located by individuals whose opinions and habits are different from those of the auhtor.

1. State Word Storage Organization

Data that are exchanged during a sequence switch

Register	Bits	Description
R0	24	data relative (i.e., "memory address register")
R1	24	current routine's temporary storage base pointer, relative in C(T1)
R2	24	calling instruction's temporary storage base pointer, relative in C(T1)
R3	24	calling instruction's T0
R4	24	PR of the next instruction after the calling instruction
R5	8	calling instruction's ST
	8, 8, ..., 8	48 bytes, 8 bits each, which complete the 64 byte pool from which R6-R15 are formed, as indicated by D6-D15
D6	12	gives starting byte and byte count of R6
D7	12	gives starting byte and byte count of R7
...		
D15	12	gives starting byte and byte count of R15
PR	24	program relative
T0	24	attached program segment
T1	24	attached temporary storage segment
T2	24	attached general segment
T3	24	attached general segment

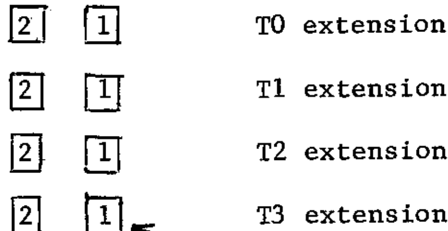
SN [16] sphere number

ST [8] status byte

Bits:

0-2: condition code
3: enable integer overflow break
4: enable real underflow and significance break
5: enable jump break
6: real arithmetic significance mode
7: inhibit block look aside update

Data that are not exchanged during a sequence switch



type of reference allowed:

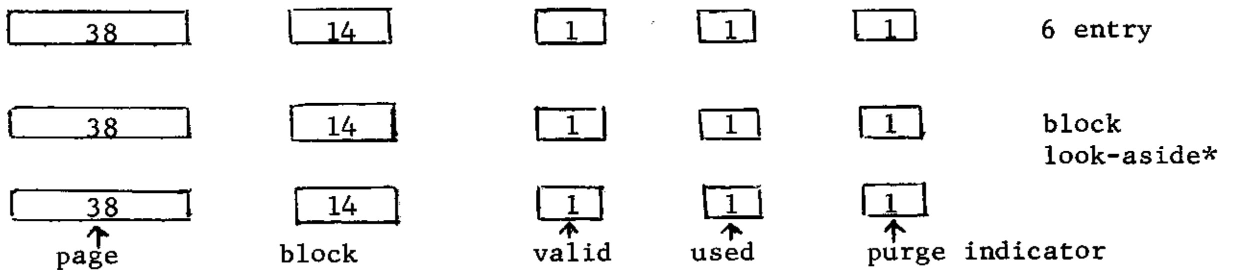
inhibit reference because of lockout

00: read only for instructions and immediate data only

01: read only

10: not used

11: all references permitted



* Lee, F.F., "Look-Aside Memory Implementation," Project MAC Memorandum MAC-M-99, August 19, 1963.

2. A Few Typical Instructions

The "read from memory" instruction, "mrtn" is one syllable in length, and has the following structure

10yyxxx
IN

A memory name is formed by the juxtaposition of C(Tyy) and C(RO). The data by this name is fetched from memory and placed in Rxxxx. The number of bytes fetched is equal to the length of Rxxxx. The first byte is placed in the left most cell, and bytes of consecutively higher names are placed to its right. The condition is unchanged by the mrtn instruction.

The "logical load register" instruction, "llod", is 2 or more syllables in length, and has the following format

000i1111	xxxxyyyy
IN	RD

If i=0, Rxxxx is cleared, and C(Ryyyy) are placed right - justified into Rxxxx. If Ryyyy is longer than Rxxxx, the excess C(Ryyyy) are ignored. Ryyyy is unchanged. Here C(Ryyyy) is called a "register operand," and yyyy is called a "register number."

If i=1, the operation is the same except that the operand to be loaded into Rxxxx is formed from the syllables following the second syllable of the instruction according the information contained in yyyy. The operand so formed is called an "immediate operand," and yyyy is called an "immediate control group".

The second syllable of the llod instruction is called an "RD" syllable, because it gives "register and data".

The llod instruction sets the condition code as follows.

code	condition
0	<0, no non-zero bits ignored
1	$=0$, no non-zero bits ignored
2	>0, no non-zero bits ignored

code	condition
3	———
4	<0, non-zero bits ignored
5	=0, non-zero bits ignored
6	>0, non-zero bits ignored
7	———

The "quick add to register" instruction, "qadd", is two syllables long, with the following format.

```
00010110        xxxxyyyy
   IN              RQ
```

yyyy is added to C(Rxxxx) at its low order (right hand) end. The result is placed in Rxxxx. Here yyyy is called a "quick operand." The second syllable of this instruction is called an "RQ" syllable, because it gives "register and quick data."

The condition code is set by the qadd instruction as follows

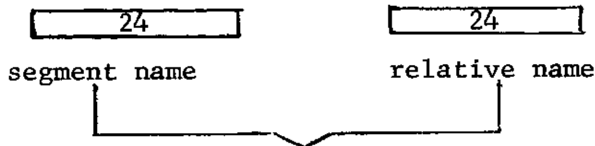
code	condition
0	<0, no ov
1	=0, no ov
2	>0, no ov
3	———
4	<0, ov
5	=0, ov
6	>0, ov
7	———

3. Natural Data Formats

The left most bit of a word is numbered bit 0, and successive bits to its right are given successively higher numbers. The bytes of a word are likewise numbered starting with zero at the leftmost byte, with successive bytes to the right being given successively higher numbers.

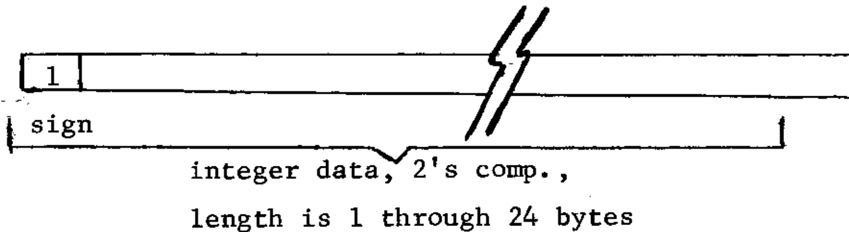
A word is to a register as a byte is to a cell as a bit is to a flip-flop. That is, words, bytes, and bits occupy registers, cells, and flip-flops respectively.

The unit of data that is named in the memory is the eight-bit cell. The programmer indicates a byte in the memory by specifying a segment name and a relative name. These two quantities are juxtaposed to form a "full memory name."



full memory name to indicate one eight-bit cell

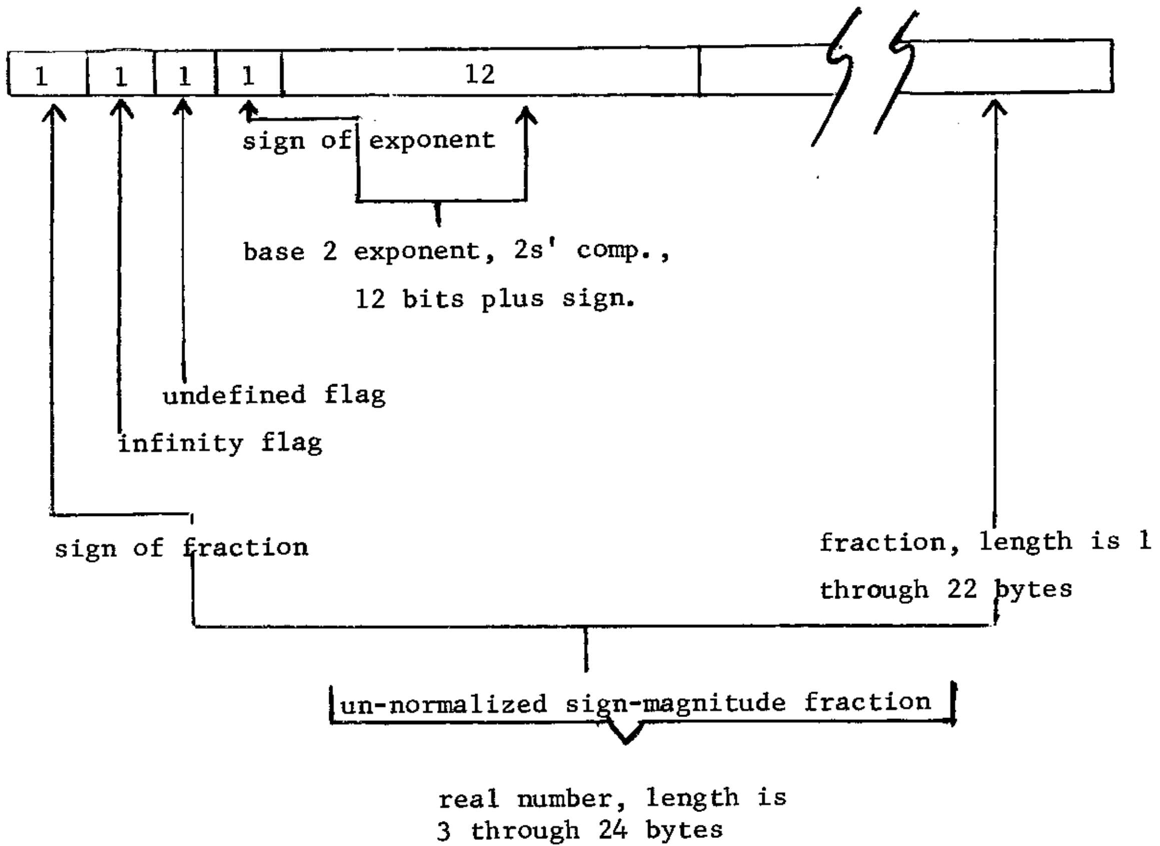
Integers consist of from one to 24 bytes, in two's-complement notation, with the leftmost bit of the leftmost byte being the sign bit.



integer data, 2's comp.,
length is 1 through 24 bytes

Real (floating point) numbers consist of from 3 to 24 bytes. The fraction is expressed in sign-magnitude notation, with minus zero being converted to plus zero before and after all real arithmetic operations. The result of a real arithmetic operation is not necessarily in normalized form. The exponent is a two's-complement number which gives the power of two which multiplies the fraction.

The leftmost (lower named) two bytes of real numbers give the exponent, infinity flag, undefined flag, and sign of the fraction. These two bytes have the same format regardless of the length of the real number.

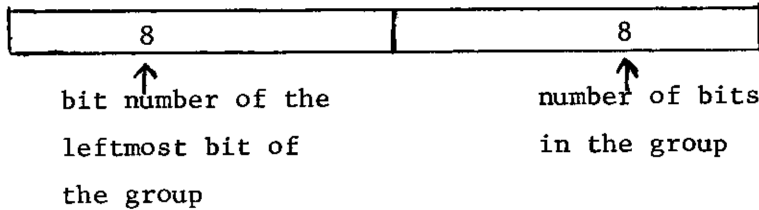


Significance is controlled in MAP-1 by entering a "significance mode", in which all real arithmetic is performed on pairs of numbers in adjacent registers, the first register being specified by the instruction, and the second register having a number one higher than the first. The real number in the first register represents the algebraicly lowest value that the quantity can have, while the real number in the second register represents the algebraicly highest value that the quantity can have. In the significance mode, each real arithmetic operation is automatically done twice, to develop the appropriate pair of numbers for the result.

In both modes of real arithmetic, right adjustment of the fraction is always followed by rounding. When left adjusting is performed, zeros fill the low order part of the fraction in the non-significance mode, while in the significance mode either zeros or ones fill the low order part of the fraction, whichever will result in the greatest difference between the

two final results. Whenever a floating point result is to be placed in a register which cannot hold the entire fraction of the result, the result is left-adjusted by an amount that is just sufficient to allow the result to occupy the register. If the result becomes normalized before it can be left-adjusted a sufficient number of places, then the result is placed in the register in normalized form and the fraction is rounded.

The "pack or unpack" instruction "poup", requires the programmer to prepare a "group control word." This word specifies a contiguous group of bits having arbitrary length and position within a register. The group control word is 2 bytes long and has the following format.



4. Immediate Data

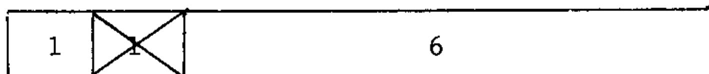
Two types of immediate operand specifications are used in MAP-1: the specification hereinafter called "immediate", and that called "quick". An example of a quick operand specification in given in the previous description of the qadd instruction. A quick operand is always given in one of the instruction syllables and is usually a number from 0 through 15.

An immediate operand is specified in a string of syllables following a syllable containing an "immediate control group." The control group specifies the number of syllables in this string. The operand is formed by placing the string right-justified in an imaginary 64 cell register, shifting the string left by a number of cells that is specified in the contol group, making the bits to the left of the string either ones or zeros according to the control group, and finally making the bits to the right of the string either ones or zeros according to the control group. One value of the control group bits calls for an additional syllable of control information to appear before the string of data syllables. The following table specifies the manner in which immediate data is coded.

Immediate Control Group (icg)

Code	no. of syllables in data string which follows immediately	no. cells by which data string is to be shifted left	Bits to left	Bits to right	Picture of right-most 5 bytes of operand				
0000	1	0	0	-	←0	0	0	0	—
0001	1	1	0	0	←0	0	0	—	0
0010	1	2	0	0	←0	0	—	0	0
0011	1	3	0	0	←0	—	0	0	0
0100	2	0	0	-	←0	0	0	—	—
0101	2	1	0	0	←0	0	—	—	0
0110	2	2	0	0	←0	—	—	0	0
0111	3	0	0	-	←0	0	—	—	—
1000	1	0	1	-	←1	1	1	1	—
1001	1	1	1	1	←1	1	1	—	1
1010	1	2	1	1	←1	1	—	1	1
1011	1	3	1	1	←1	—	1	1	1
1100	2	0	1	-	←1	1	1	—	—
1101	2	1	1	1	←1	1	—	—	1
1110	2	2	1	1	←1	—	—	1	1

The code 1111 in an icg indicates that the next syllable is not part of the data string, but gives additional control information. This additional syllable has the following format.



no. of syllables in data string to follow, which are not to be shifted left in the formation of the operand

1 means put ones to left of data string
0 means put zeros to left of data string

In the following operations the data string is placed left justified in the imaginary 64 cell register and shifted right.

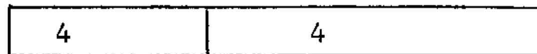
wder
hlod
radd
rsub
rmul
rdvd
wnds

5. Syllable Types

IN (instruction)

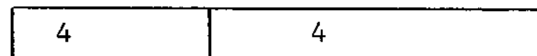
opcode

RR (register - register)



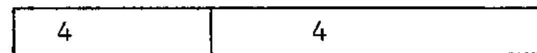
R# R#

RD (register - data)



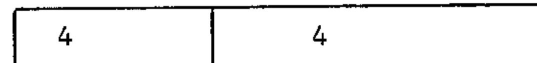
R# R# or icg if a preceding i bit is one

RQ (register - quick data)



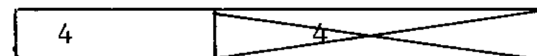
R# quick data

RS (register - shift data)



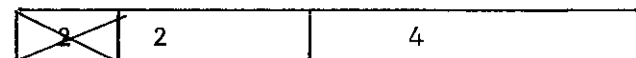
R# R# containing shift data or quick shift data if a proceeding i bit is one

RN (register - not used)



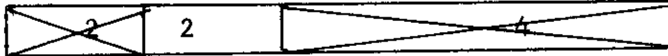
R#

TR (attachment register - register)



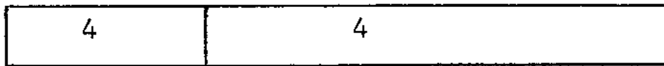
T# R#

TN (attachment register - not used)

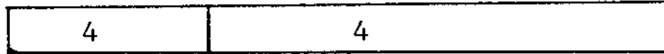


QK (quick data)

QR (quickdata - register)



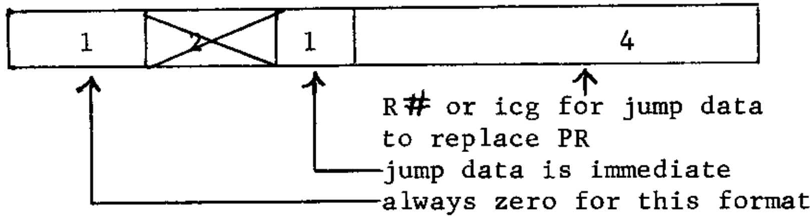
QD (quickdata - data)



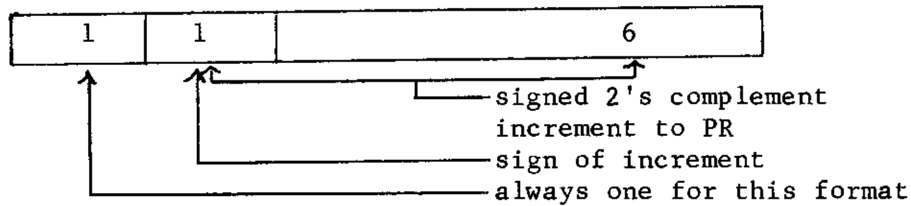
JM (jump)

There are two formats, depending on the leftmost bit

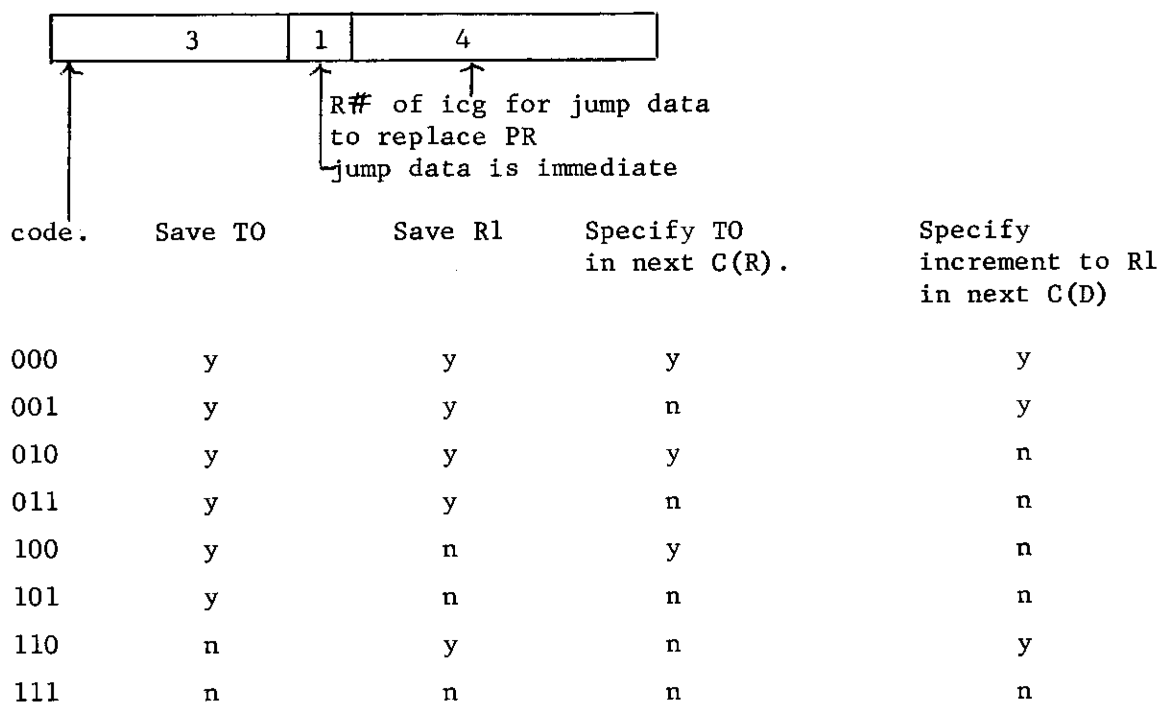
1) leftmost bit is zero



2) leftmost bit is one



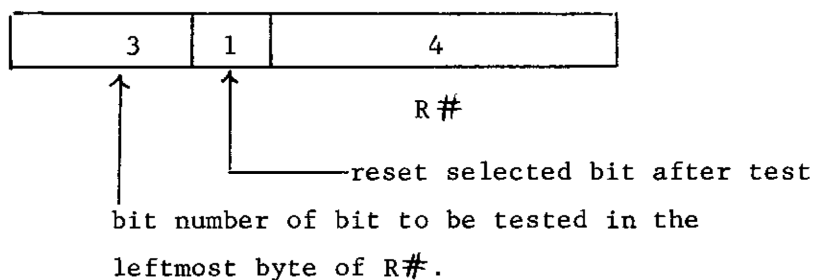
S1 (first special syllable - used in jcal)



code.	Save TO	Save R1	Specify TO in next C(R).	Specify increment to R1 in next C(D)
000	y	y	y	y
001	y	y	n	y
010	y	y	y	n
011	y	y	n	n
100	y	n	y	n
101	y	n	n	n
110	n	y	n	y
111	n	n	n	n

An RD syllable follows for all codes except 011, 101, and 111.

S2 (second special syllable - used in jbit)



S3 (third special syllable - used in jqad)

3	1	4

R# or icg of comparison data

comparison data is immediate

type of comparison: jump if C(R): comparison data

000	never jump
001	$>$
010	$=$
011	$\geq$
100	$<$
101	$\neq$
110	$\leq$
111	always jump

S4 (fourth special syllable - used in poup)

1	1	1	1	4

R# or icg of group control word

increment group control word

pack

unpack

clear C(R)

6. Concise Instruction Tabulation

Syllables in parentheses are optional. The asterisk indicates that the instruction is described in detail in the next section entitled, "A description of selected instructions." The double asterisk indicates that the instruction is described in detail in the section entitled, "A few typical instructions." For additional details on the behaviour of the instructions, refer to the section entitled, "syllable types."

Instructions With One INSyllable

code in IN syl.	nmem. of first syl.	syl. to follow	description
11yyxxxx	mntn	____	write to memory using R0 and attachment register
10yyxxxx	mrtn	____**	read from memory using R0 and attachment register
011ixxxx	nlod	____	logical load R0
010ixxxx	nadd	____	logical add to R0
0011xxxx	nqsb	____	quick subtract from R0
0010xxxx	nqad	____	quick add to R0
000i1111	llod	RD**	logical load
i1110	ladd	RD	logical add
i1101	lsub	RD	logical subtract
i1100	wder	QD	write into one designator
i1011	jxlo	RD,JM	add one to C(R), jump if less than C(D)
i1010	jxhe	RD,JM	add one to C(R), jump if higher than or equal to C(D)
i1001	jseg	RD	inter-segment jump, to segment in C(R), and relative in C(D).
i1000	jcal	S1, (AD) *	subroutine call
10111	qlod	RQ	quick load register
10110	qadd	RQ**	quick add to register
10101	qaub	RQ	quick subtract from register
10100	rder	QR	read from one designator
10011	jbit	S2, JM	jump on bit with option reset
10010	jqad	RQ, S3, JM	quick add and compare
10001	jcon	QK, JM*	jump on masked condition code
10000	jump	JM	jump
00111	nwtr	RN, TR	write to memory using C(right R) and attachment register
00110	nwrr	RN, RR	write to memory using segment in C(mid.R) and relative in C(right R).
00000101	mrtr	RN, TR	read from memory using C(right R) and attachment register

code in IN syl.	mnem. of first syl.	syl. to follow	description
00000100	mrrr	RN, RR	read from memory using segment in C(mid. R) and relative in C(right R)
00011	qdvd	RQ, RN	quick divide
00010	qmul	RQ	quick multiply
00001	supd	——	suppress block loadaside updating
00000000	genl	?	prefix to more instructions

Instructions with two IN syllables.

The first IN syllable is always 00000000.

code in IN syl.	mnem. of second syl.	syl. to follow	description
11yyssss	wath	______	write to attachment register
10yyxxxx	rath	______	read from attachment register
011i0000	lorl	RS	logical shift register left
0001	lorr	RS	logical shift register right
0010	locl	RS	logical shift combined left
0011	locr	RS	logical shift combined right
0100	rorl	RS	rotate register left
0101	rorr	RS	rotate register right
0110	rocl	RS	rotate combined left
0111	rorc	RS	rotate combined right
1000	isrl	RS	integer shift register left
1001	isrr	RS	integer shift register right
1010	___	___	not used
1011	rjlz	RS, RN *	real adjust left with zeros
1100	rjlw	RS, RN	real adjust left with ones
1101	rajr	RS, RN	read adjust right
1110	hlod	RD *	high order load
011i1111	ilod	RD *	integer load
010i0000	iadd	RD	integer add
0001	isub	RD	integer subtract
0010	imul	RD *	integer multiply
0011	idvd	RD, RN *	integer divide
0100	radd	RD	real add
0101	rsub	RD	real subtract
0110	rmul	RD	real multiply
0111	rdvd	RD	real divide
1000	exor	RD	exclusive or
1001	inor	RD	inclusive or

code in IN syl.	mnem. of second syl.	syl to follow	description
010i1001	amdr	RD	and
1011	tums	RD	test under mask
1100	poup	S4, RR *	pack or unpack with optional increment
1101	motr	RD, TR	or to memory and load previous
1110	matr	RD, TR	and to memory and load previous
010i1111	mitr	RD, TR *	increment memory and load previous
001ixxxx	wnds	QD	write n designators
0001xxxx	rnds	QR *	read n designators
00001111	exch	RR	exchange registers
1110	exec	RN	execute from register
1101	wsta	RN	write status
1100	rsta	RN	read status
1011	qwst	QK	quick write status
1010	qost	QK	quick or to status
1001	qast	QK	quick and to status
1000	comp	RN	complement register
0111	ineg	RN	integer negate register
0110	jret	________	subroutine return, retore T0 and R1
0101	jrit	________	subroutine return, restore R1,not T0
0100	jrat	________	subroutine return, restore T0,not R1
0011	mptr	RN, TR	read parameter from attached segment
0010	mprr	RN, TR *	read parameter from general segment
0001	mxtr	TR	execute from attached segment
00000000	rest	?	prefix to more instructions

7. Description of Selected Instructions

This section describes in detail the following instructions in the following order

jcal
jcon
rjlz
hlod
ilod
imul
idvd
poup
mitt
rnds
mprr

jcal - subroutine call

000i1000	xxxjyyyy	(zzzzwwww)
IN	S1	(AD)

Refer to the description of the type S1 syllable. i is the immediate bit for yyyy. If i = 1, the yyyy immediate data string comes between the second and third syllables. j is the immediate bit for wwww, if the third syllable is given. Otherwise j is not used. PR and ST are saved in R4 and R5. The PR that is saved is the relative name of the first byte of the instruction following the entire jcal instruction. yyyy gives the R# or icg of data which is loaded into PR. The code xxx specifies additional actions to be taken. TO can be saced in R3. R1 can be saved in R2. zzzz gives the R# of the data to replace TO. wwww gives the R# or icg of the data to be added to R1. All loads are logical loads. The condition code is not changed.

jcon - jump on condition

00010001	$x_0x_1x_2x_3x_4x_5x_6x_7$	yyyyyyyy
IN	QK	JM

Refer to the description of the type Jm syllable. If the condition code has value n, and x_n is one, then PR is altered according to the JM syllable. Otherwise, the next sequential instruction is executed. The condition code is not changed.

rjlz - real adjust left with zeros

00000000	011i1100	xxxxyyyy	zzzz - - - -
IN	IN	RS	RN

Refer to the descriptio n of the type RS syllable. i is the quick-immediate bit for yyyy. yyyy gives the R# or the quick data of a shift count. The shift count is taken modulo 256. C(Rxxxx) is assumed to be a real number. The fraction of C(Rxxxx) is shifted left until either the shift count is satisfied or the number is normalized. Zeros fill the low order part of the fraction. Rzzzz is loaded with the number of positions that were actually shifted. The exponent of C(Rxxxx) is reduced by this amount. This instruction sets the condition code as follows.

code	condition
0	_____
1	count satisfied, number not normalized
2	count not satisfied, number normalized
3	count satisfied, number normalized
4	_____
5	_____
6	_____
7	_____

hlod - high order load register

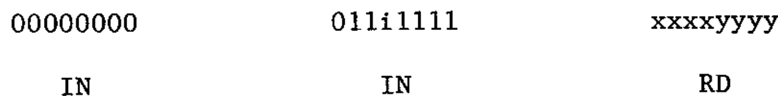
00000000	011i1110	xxxxyyyy
IN	IN	RD

i is the immediate bit for yyyy. yyyy is the R or icg of data to be loaded into Rxxxx. For this instruction the immediate data string is left - justified and right sifted in its imaginary 64 cell register. Rxxxx is cleared, and the data is placed left - justified in Rxxxx. If the data is longer than Rxxxx, the excess is ignored. This instruction is identical to llod, except that it places the data left - justified in Rxxxx instead of right justified, hlod sets the condition code as follows.

code	condition
0	< 0, no non-zero bits ignored
1	$= 0$, no non-zero bits ignored
2	> 0, no non-zero bits ignored
3	———
4	< 0, non-zero bits ignored
5	$= 0$, non-zero bits ignored
6	> 0, non-zero bits ignored
7	———

To load a real number into a smaller register with rounding, first clear the target register using qlod, then use radd.

ilod - integer load



00000000	011i1111	xxxxyyyy
IN	IN	RD

i is the immediate bit for yyyy. yyyy gives the R# or icg of data to be loaded into Rxxxx. If the data is shorter than Rxxxx, it is lengthened by extending the leftmost bit to the left. If the data is longer than Rxxxx, the data is shortened by an integer left shoft. Ryyyy is not changed. The condition code is set as follows.

code	condition
0	< 0, no ov
1	$= 0$, no ov
2	> 0, no ov

3	———
4	<0, ov
5	=0, ov
6	>0, ov
7	———

imul - integer multiply

00000000	010i0010	xxxxyyyy
IN	IN	RD

i is the immediate bit for yyyy. yyyy is the R# or icg of the multiplier. C(Rxxxx) is multiplied by the multiplier, and the result is placed in Rxxxx. Ryyyy is unchanged. The condition code is set as for the ilod instruction.

idvd - integer divide

00000000	010i0011	xxxxyyyy	xxxx - - - -
IN	IN	RD	RN

i is the immediate bit for yyyy. yyyy gives the R#or icg of the divisor. C(Rxxxx) is divided by the divisor. The quotient is placed in Rxxxx, and the remainder is placed in Rzzzz. Ryyyy is unchanged. The condition code is set as for the ilod instruction.

00000000	010i1100	abcdxxxx	yyyyzzzz
IN	IN	S4	RR

Refer to the description of the type S4 syllable. i is the immediate bit for xxxx. xxxx is the R# or icg of a group control word. Ryyyy is the receiving register. Rzzzz is the sending register. The bits abcd give a microprogram which is executed from left to right. In these bits, a zero means do nothing, a one means perform the following actic

a: clear Ryyyy

b: unpack from Rzzzz into Ryyyy

c: pack from Rzzzz into Ryyyy

d" increment the group control word

The group control word specifies a group of contiguous bits of arbitrary length at an arbitrary position within a register. The group control word consists of 2 bytes, and is assumed to be right-justified in its register or immediate data string. Byte 0 specifies the bit number within a register of that bit which is the leftmost bit of the group. Byte 1 gives the number of bits in the group. If byte 1 is absent, it is assumed to be equal to 1. If both bytes are absent, then the pack, unpack, and increment microsteps are no-operation steps.

In the unpack microstep, the group is specified within Rzzzz. The group is fetched and replaces the group of the same size at the right hand end of Ryyyy. Only the righthand end of Ryyyy is altered by the unpack microstep.

In the pack microstep, the group control word specifies a group within Ryyyy. This group is replaced by the group of the same size at the right hand end of Rzzzz. The group in Ryyyy is the only group that is altered by the pack microstep.

The increment microstep adds byte 1 of the group control word to byte 0 of the group control word, changing only byte 0 of the group control word.

After the four microsteps are complete, the Boolean variable (limit) is computed.

$$\text{(limit)} \leftarrow [\text{(byte 0 of group control word)} + \text{(byte 1 of group control word)}] > \text{(register length)}$$

The register whose length is included in the above equation is Ryyyy, except that when b = 1 and c = 0, the register is Rzzzz. (limit) is used to set the condition code.

Notice that specifying both a pack and an unpack microstep in the same instruction is not equivalent to specifying neither a pack n or unpack microstep. For example, poup may be used to exchange the left and right halves of a register.

The poup instruction sets the condition code as follows.

code	(limit)	condition unpacked group = 0	packed group = 0
0	F	F	F
1	F	F	T
2	F	T	F
3	F	T	T
4	T	F	F
5	T	F	T
6	T	T	F
7	T	T	T

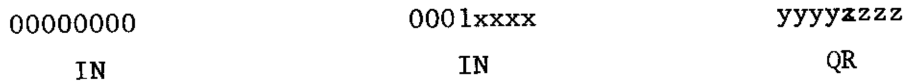
mitr - increment memory and load previous

00000000	010i1111	xxxxyyyy	- - zzwwww
IN	IN	RD	TR

i is the immediate bit for yyyy. yyyy gives the R# or icg of an addend to be added to a quantity in memory. The memory quantity consists of the same number of bytes as Rxxxx, starting with the byte named by the justaposition of C(Tzz) and C(Rwwww) and bytes of consecutively higher names. The memory quantity is placed in Rxxxx. The addend is then added to the memory quantity with the result replacing the memory quantity. If the addend is shorter than the memory quantity, the sign bit of the addend is extended to the left. If the addend is longer than the memory quantity, the addend is shortened by an integer left shift. The addition is performed in such a way that no intervening reference to the memory quantity can be made by another processor between the time that previous value is read out and the time that the new value is stored. mitr sets the condition code as follows.

code	condition
0	Rxxxx 0, no ov in add
1	Rxxxx = 0, no ov in add
2	Rxxxx 0, no ov in add
3	______
4	Rxxxx 0, ov in add
5	Rxxxx = 0, ov in add
6	Rxxxx 0, ov in add
7	______

rnds - read n designators



xxxx designators are read into Rzzzz starting with the designator for register yyyy. The designators are maximally packed and left-justified in Rzzzz. The remainder of Rzzzz is cleared. If Rzzzz is not long enough to hold all of the designators, the right hand end of the designator data string is ignored. The condition code is unchanged.

mprr - read practice from general segment

Ryyyy is considered to contain a segment name. A memory name is formed by the justaposition of C(Ryyyy) and C(Rzzzz). The rightmost four bits of the byte so named are taken as an immediate control group. An immediate operand is formed according to this control group from the bytes which follow it in the memory. This operand is loaded into Rxxxx in the manner of the llod instruction. The number of bytes that were read from the memory is logically added to Rzzzz. The condition code remains unchanged. Note that the leftmost 4 bits of the named bytes are ignored, but the named byte is included in the count that is added to Ryyyy.

8. A Suggested Assembly Language.

This section describes the rudiments of an assembly language for writing MAP-1 programs. This language is specified in order to allow individuals evaluating MAP-1 to communicate more easily with each other. No attempt is made to completely specify the language.

Blanks are ignored.

The characters / and <carriage return> indicate the end of an syllable specification, and are identical from the assembler's point of view. It is siggested that <carriage return> be used to signify the end of an instruction, while / be used to delineate syllables within an instruction.

The character , means "inclusive or".

Symbols must have a leading alphabetic character, followed by any number of alphabetic or numeric characters. The IN syllable codes are permanently defined as symbols, and the symbol i is premanently defined to be octal 20.

The character : indicates that the value of the precceding symbol is to be defined as the relative name of the next syllable to be assembled.

The character - at the end of an expression means that the value of the expression is to be shifted left four places.

In expressions, the characters + and - and * stand for addition, subtraction, and multiplication respectively. When the character * stands in a non-operator position, it is a symbol whose value is the relative name of the syllable in whose specification it appears.

When a syllable specification consists entirely of an expression enclosed within the characters (and) the value of the expression is coded into immediate syllables in the most economical way, and the appropriate immediate control group is placed in the rightmost four bits of the preceeding syllable.

Expressions are delimited on the left by

<carriage return>

/

,

:

(

Expression aredelimited on the right by

carriage return

/

,

-,

)

The following program, which continually adds one to register 10, illustrates most of the features of the assembly language.

```
start:   qlod/12-, 0
         qadd/12-, 1
         jump/i/ (start +2)
```

9. A Programming Example - Matrix Multiplication

The program given below multiplies an n x m matrix a by an m x p matrix b to yield an n x p matrix c. n, m, and p are each less than 256. The elements of a, b, and c are real numbers that are 6 bytes long. a, b, and c exist in the segment whose name is in T2. This segment is assumed to be less than 2.P.16 bytes in length. The relative names of the origins of a, b, and c within seg (C(T2)) are contained in three 2-byte quantities that also exist in seg (C(T2)). The relative names of these quantities are aloc, bloc, and cloc, respectively. In this example, n, m, p, aloc, bloc, and cloc are assembly parameters. The program is coded to minimize the number of bits in the program code rather than the speed of the program's execution.

The matrices are stored forward by rows, so that, if the subscripts start from zero, relative name (a(i,j)) = C(aloc) + (i*m +j) *6.

The matrix product is evaluated using the formula

$$c(i,j) = \sum_{k=0}^{m-1} a(i, k) * b(k, j)$$

The general registers 6 through 15 are used as indicated below. It is assumed that the designators for these registers have already been properly loaded, and that the processor is in an appropriate state to trap on any exceptional

conditions that might occur in the course of the real arithmetic

register (octal)	use	number of bytes (decimal)
6	i	1
7	j	1
10	k	1
11	relative name of current element in a	2
12	relative name of current element in b	2
13	relative name of current element in c	2
14	accumulator	6
15	multiplicand	6
16	multiplier	6
17	not used	21

```
mampy:      nlod, i/(aloc)
            mrtn, 2-, 11
            nlod, i/(bloc)
            mrtn, 2-, 12
            nlod, i/(cloc)
            mrtn, 2-, 13
            qlod/6-, 0

colop:      qlod/7-, 0

rolop:      qlod/10-, 0
            qlod/14-, 0

iplop:      nlod, 11
            mrtn, 2-, 15
            nlod, 12
            mrtn, 2-, 16
            genl/rmul/15-, 16
            genl/radd/14-, 15
            qadd/11-, 6
            ladd, i/12-/(p * 6)
            jxlo, i/10-/(m)/iplop-*-1
            nlod, 13
            mwtn, 2-, 14
            qadd/13-, 6
            jxhe, i/7-/(p)/endro-*-1
            lsub, i/11-/(m*6)
            lsub, i/12-/(m*p*6-6)
            jump/rolop-*-1

endro:      jxhe, i/6-/(n)/end mp-*-1
            lsub, i/12-/(m*p*6 + p*6-6)
            jump/colop-*-1

endmp:
```

The various parts of the program consume the following amounts of storage.

	bytes	bits
inner loop	20	160
middle loop (additional)	22	176
outer loop (additional)	12	96
initialization (additional	14	112
total	68	544

10. A Comparison With a 7094 Program to Perform an Equivalent Matrix Multiplication.

The nomenclature and specifications of this example are essentially the same as those of the proceeding one. A, B, and C are assumed to reside simultaneously in the 7094 memory, which consists of 2.P.15 words. ALØC, BLØC, and CLØC are the names of memory registers that contain the addresses of the matrices A, B, and C, respectively. N, M, P, ALØC, BLØC, and CLØC are assembly parameters. A, B, and C are stored forward by rows, as in the preceding example.

The index registers of the 7094 are appropriated as follows.

register	use
1	N-i
2	P-j
3	M-k
4	not used
5	complement of address of current element in A
6	complement of address of current element in B
7	complement of address of current element in C

```
MAMPY        LAC     ALØC, 5
             LAC     BLØC, 6
             LAC     CLØC, 7
             AXT     N, 1
CØLØP        AXT     P. 2
RØLØP        AXT     M, 3
             STZ     0, 7
IPLØP        LDQ     0, 5
             FMP     0, 6
             FAD     0, 7
             STØ     0, 7
             TXI     *+1, 5, -1
             TXI     *+1, 6, -P
             TIX     IPLØP, 3, 1
             TXI     *+1, 7, -1
             TNX     ENDRØ, 2, 1
             TXI     *+1, 5, M
             TXI     RØLØP, 6, M*P-1
ENDRØ        TNX     ENDMP, 1, 1
             TXI     CØLØP, 6, M*P+P-1
ENDMP
```

The storage requirements of these two matrix multiplication algorithms are compared below.

	7094 bits	MAP-1 bits
inner loop	252	160
middle loop (additional)	216	176
outer loop (additional)	108	96
initialization (additional)	144	112
total	720	544

The 7094 example was prepared with the assitance of A.L. Scherr. The above program, which closely follows the MAP-1 coding, was the most efficient of several programs which Mr. Scherr prepared.