PlasMa

aka Phil's lights and switches Machine

Mini-Mainframe Simulator Project

Instruction Set

and

I/O Functions Manual - PleX

Phil Tipping

www.philizound.co.uk

Table of Contents

1. Introduction

This describes the PleX instruction set, which is one of several supported by the PlasMa machine. Syntax details are included for use with the Plasm offline assembler; more details are on the philizound.co.uk website.

This instruction set architecture (ISA) emulates a more complex computer than the Toy ones, and uses many ideas from the NICE and QNICE ISAs; see Acknowledgements.

Additional features and peripherals use ideas taken from various real-life machines.

The architecture uses 15 x 16-bit work registers, each accessible with up to 4 addressing modes, giving a rich set of features with fewer restrictions on register usage (as with the format-2 instructions in Toy-B). The stack pointer (SP) is held in one of the work registers so stack operations can share normal register-based instructions.

Memory addresses are 16 bits, as opposed to 8 bits in Toy, so more memory is potentially available, only limited by the MCU and hardware implementation.

The first few words of memory are non-volatile to simulate a small read-only ‘fixed store’; this can be programmed with a bootstrap routine for loading larger programs from the peripherals.

The PleX I/O system provides access to a full set of ‘system’ peripherals, including an operator’s console and emulated peripherals such as a paper tape reader and punch, magnetic tape decks and exchangeable disk drives.

In this manual, hex numbers are prefixed with '$'.

2. Acknowledgements

The PleX instruction set is adapted from NICE and QNICE by kind permission of Bernd Ulmann (www.vaxman.de). Thanks to Bernd and Mirko (www.sy2002.de) for their encouragement and support. Details of QNICE and its FPGA implementation are at github.com/sy2002/QNICE-FPGA.

The Toy name and instruction set is used by kind permission of Robert Sedgewick and Kevin Wayne at Princeton university, and is described in their book 'Computer Science'; more info at:
introcs.cs.princeton.edu/java/home and introcs.cs.princeton.edu/java/60machine
Coursera course site: www.coursera.org/learn/cs-algorithms-theory-machines

3. Documentation and Programs

The following items are available from philizound.co.uk

Documentation

- Machine Manual
- Instruction Set Manual - Toy-A
- Instruction Set and I/O Functions Manual - Toy-B
- Instruction Set and I/O Functions Manual - PleX
- Simulator Manual
- Assembler Manual

Programs

- PlasMaSim Simulator
- Plasm Assembler

4. Architecture summary

16-bit instructions.

2048 (implementation-dependent) x 16-bit words of main memory.

16 x 16-bits words of non-volatile 'fixed store' memory (fst) occupying memory addresses 0 to 15.

15 x 16-bit registers, r0, r1, r2… r9, ra, rb… re.

Stack pointer (SP) uses register 14 (re).

32-bit accumulator.

Flags for: overflow, negative, zero, carry, extended.

I/O devices.

5. Memory

Memory comprises 16-bit words. Addresses are also 16 bits, although the current address range on the machine is 0 to 2047 ($7FF)[1]. Addresses outside this range wrap round.

5.1 Fixed store (FST)

This emulates a hard-wired unit which would normally be wired or pre-configured offline with a set of instructions such as a bootstrap routine; the contents are ‘fixed’ as they are retained over a power-off.

The fixed store occupies the first 16 words of memory, addresses 0 to 15 ($F).

When PlasMa is running, the contents are treated as read-only; any write operations are discarded.

When PlasMa is stopped, the fixed store can be loaded, or 'wired', using the load switches in conjunction with the safety switch. This is the only way of writing values to it[2].

6. System and I/O overview

The 'sys' instruction handles system functions (extra-codes) and I/O via a 16-bit 'function word’ which defines the peripheral/device, device command, and other parameters depending on the device. The function word can be held in memory or one of the work registers.

6.1 Built-in peripherals

- Operator's console (oper) with screen and keypad
- Paper tape reader and punch for serial data
- Two mag tape decks for serial data
- Two exchangeable disks for random access data

The oper screen uses the built-in display. There are no automatic conversions into hex and decimal as with the Toy TTY emulation, so cursor positioning and ASCII conversions must be handled within your program.

All built-in storage peripherals use individual plug-in sd-cards; see the Machine Manual for details.

Other 'devices' include switch states, timers/delays etc. Other extra-codes include maths, conversions and accumulator functions not available in the PleX ISA.

1 The upper value is dependent on the MCU implementation.

2 Until you've decided what to store in the FST, you can make it easier to develop programs starting at ram address $10 by loading a single ‘jmprs $10’ instruction into address 0. This will be retained over power-offs.

6.2 External peripherals

- Printer functions using the parallel interface socket or other 5v-level devices
- MIDI functions using the MIDI input and output sockets
- Qwerty keyboard functions using the PS/2 socket for PS/2 or USB keyboards

7. Instructions overview

The PleX instruction set architecture (ISA) is adapted from the 32-bit NICE and 16-bit QNICE ISAs (see Acknowledgements). There is no compatibility as some features have been removed and others added or enhanced to meet PlasMa's requirements[3].

In this document, opcode descriptions use the programming symbol '=' instead of ' ← ' used in Toy, and the term 'ram' is used to distinguish the volatile part of memory from the non-volatile 'fst' part. The term 'mem' means both, i.e. the whole address range.

7.1 Operand Addressing Modes

The 'general format' instructions have 2 operands; a destination (dst) and source (src). The general format jump/call instructions have 1 operand.

All of these operands use 4 bits to access work registers similar to Toy, but the addressing mode (such as direct or indirect) is defined by 2 additional bits, giving up to 4 possible formats for each operand:

00 rx/n direct; the contents of the register or the value of a literal/constant.
01 @rx/n indirect; the memory contents addressed by the contents of the register/literal value.
10 @rx+ indirect, post-increment; as indirect but increment the register contents *after* access.
11 @-rx indirect, pre-decrement; as indirect but decrement the register contents *before* access.

If the operand register field is 15 ($F)[4], the operand refers to a literal; see Literals/Constants.

Operand pre/post processing is done in the order of source first, then destination.

The 'short' and 'misc' format instructions trade some or all of these address modes for extra functionality.

7.2 Literals/Constants

Instructions can access literal/constant values in two ways.

7.2.1 Long literals

These can be up to 16 bits and are held in the memory location(s) immediately following the instruction word.

Long literals are referenced when an operand register field value is 15 ($F). Some addressing modes are meaningless with literals/constants so the following rules apply:

- Literals used as source operands can only be used in direct or indirect addressing modes.
- Literals used as destination operands can only be used in indirect mode.

If an instruction uses 2 long literal operands, the source value is held in the word immediately following the instruction, and the destination value in the word after that.

3 For example, there are no interrupts or register banks (although the latter may be added later), and the work register set does not include the flags or PC. Additions include instructions for short literals, a 32-bit accumulator, and a single 'sys' function for I/O and extra-codes. The source and destination operand order has also been reversed to conform to the existing Toy convention.

4 There is no register 15 in PleX.

The Plasm assembler will plant literals in the correct locations automatically, e.g.

- 'mov r5 1000' is assembled as a 'mov' instruction with a direct literal source operand, followed by a word containing the literal value of 1000.
- 'mov @30 @1000' is assembled as a 'mov' instruction with an indirect literal operand for both source and destination, followed by a word containing 1000, followed by another word containing 30[5].

7.2.2 Short literals

The 'general format' instructions use a separate memory location to hold literal values. To avoid wasting memory for small literals, some of these instructions have a 'short' variant which trades some functionality for the ability to hold the literal within the instruction word itself.

The following restrictions apply:

- Short literals can only be used as source operands.
- Short literals have no address mode so they are always accessed in direct mode.
- Short literals for non-jumps/calls are limited to 4 bits, giving a range of 0 to 15.
  Shift and rotate instructions treat them as signed quantities, giving a range of -8 to +7.
- Short literals for jumps/calls are limited to 7 bits and are treated as signed relative offsets, giving a range of -64 to +63.

If a short instruction also contains a destination operand, this can be a register or long literal as normal, but with the following restrictions:

- Destination registers can only be used in direct or indirect address modes.
- Destination literals can only be used in indirect mode.

The Plasm assembler identifies long instructions which may be candidates for short variants.

7.3 Jumps and Calls

Jumps and calls handle PC and the stack pointer (SP) automatically. They can all be unconditional or conditional subject to the state of the Flags; the combinations are listed in the instruction table.

Target address operands for 'general format' jump/calls can be registers or literals and can be absolute or relative. Literal values are held in the memory location(s) immediately following the instruction word. The following rules apply:

- Register operands can use all 4 address modes.
- Literal operands can only use direct or indirect modes.

7.3.1 Short jumps/calls

A set of short jump and call variants allow small target address values to be embedded within the instruction with the following restrictions:

- Short addresses are relative addresses only.
- Short addresses are limited to a range of -64 to +63.

The Plasm assembler identifies long instructions which may be candidates for short variants.

7.4 Stacks

7.4.1 System stack

The system stack uses main memory and decrements the stack pointer (SP) before storing the item. SP always points to the last item stored on the stack[6].

5 This is a store-to-store operation without using registers.
6 The location of the last item pushed is called 'top of stack' even though the stack grows downwards. in memory.

The stack pointer (SP) is held in reg 14 (re) so no special instructions are required to access it; the indirect pre/post address modes are sufficient to handle push and pop operations[7].

To push items onto the stack, use a destination operand of '@-re' (SP indirect, pre-decrement). This decrements SP before storing the item.
To pop items from the stack, use a source operand of '@re+' (SP indirect, post-increment). This retrieves the item before incrementing SP.

The Plasm assembler provides push and pop mnemonics for these operations.

'Call' instructions use the system stack to store return addresses; they adjust SP automatically.

The PleX ISA does not contain a dedicated ‘return’ instruction, but this can be achieved with a ‘jmp @re+’ instruction which pops the return address from the stack and jumps to it; this can be conditional or unconditional. The Plasm assembler provides a set of ‘ret’ mnemonics for these.

7.4.2 User stacks

You can create your own stacks by nominating other work registers as stack pointers, and push and pop items using @-rx and @rx+ operands as required.

8. Operational notes

Processing is stopped on illegal instructions.

Memory access instructions do not incur a time penalty (unlike the Toy emulations) as the emphasis with PleX is more on programming logic than simulating the hardware speed difference between registers and main store.

8.1 Register 14 (RE/SP)

This is used implicitly by 'call' instructions as the stack pointer (SP); see Stacks. It can still be used as a general-purpose register if those instructions are avoided.

8.2 'Register' 15 (IR2)

There is no work register 15 (RF) so the row of lights used as RF in the Toy emulations is used to display the contents of memory at PC+1. This will show the long literal value for instructions which use them[8]; see Long literals.

The row is named IR2 as it complements the main instruction register IR; IR shows the memory contents at address PC, and IR2 shows the memory contents at address PC+1.

IR2 can be used with the load and break-point switches the same way as IR.

8.3 Flags

There are 5 flags as described below. The flags column in the instruction and system/IO function tables list all flags affected by that operation. If a flag is listed, it will be set according to the condition shown and unset if the condition is not met. Flags not listed are unchanged.

Flags are only set/reset by the function itself; pre-decrement and post-increment address mode operations do not affect them.

Conditional instructions allow an additional condition 'g' (greater than 0) which is true if both Z and N flags are unset. The inverse condition 'ng' is true if either Z or N is set.

---

7 The stack can be placed anywhere in main store (outside the FST) by setting SP to the required 'bottom of stack' + 1. To place the stack at the top of main store, set SP to zero; the first ‘push’ operation causes SP to wrap-round to the highest memory location before storing the item.

8 If the instruction refers to 2 long literals, you can use the Inc PC button to inspect the 2nd value, but don't forget to use Dec PC before continuing.

Conditional jump instructions only perform pre-decrement and post-increment operations if the jump is to be taken.

8.3.1 Overflow (V)

This is set if an arithmetic operation gives an incorrect result if the operands are treated as signed numbers (the Carry (C) flag is set if the operands are treated as unsigned numbers).

It can also be set/reset by system functions.

8.3.2 Negative (N)

This flag is set when the ms-bit of the operation result is 1. This is the sign bit for signed numbers, so means the result (if treated as a signed number) is negative.

It can also be set/reset by compare instructions or system functions.

8.3.3 Zero (Z)

This flag is set when the operation result is zero.

It can also be set/reset by compare instructions or system functions.

8.3.4 Carry (C)

This is set if an arithmetic operation gives an incorrect result if the operands are treated as unsigned numbers (the Overflow (V) flag is set if the operands are treated as signed numbers).

It can also be set/reset with set/clear-carry, rotates, left shifts, compare or system functions.

8.3.5 eXtended (X)

This acts as an extension to the ls end of the destination, and is set/reset with set/clear-x or right shift instructions.

8.4 Conditional Jumps/Calls

All conditional jump/call instructions test *flag* states as opposed to *register* states (unlike the Toy instructions). Use the instruction and system tables to confirm when and where flags are updated.

8.5 Shift instructions

The shift direction depends on the sign of the shift amount; postive values shift left, negative values shift right[9]. The C and X flags are read or written depending on the direction.

A shift right copies the C flag into the ms bit of the destination, and copies the ls bit into the X flag.
A shift left copies the X flag into the ls bit of the destination, and copies the ms bit into the C flag.

For arithmetic right shifts, set C to the value of the sign bit before shifting.
For logical right shifts, clear C before shifting[10].

8.6 Rotate instructions

The rotate direction depends on the sign of the rotate amount; postive values rotate left, negative values rotate right.

As the destination is rotated, the ls/ms bits are copied to the C flag as well as being fed back to the ms/ls end. After the operation, the C flag will be a copy of the ls-bit (for rotate left) or the ms-bit (for rotate right).

9 This is the same convention as a VAX machine.
10 The 'set/clear c/x' instructions can be used; the 'load/save flags' instructions give finer control over multiple flags.

8.7 Shifts/Rotates with the Plasm assembler

The Plasm assembler uses generic mnemonics 'sh' and 'rot' for shifts and rotates, and as described above, the sign of the operand determines the direction, e.g. both of these examples plant code to shift register 7 to the left by 12 places.

```
sh r7 -12
mov r3 -12, sh r7 r3
```

However, if the value is known at assembly time (i.e. the amount is a literal), Plasm accepts alternative mnemonics (shl, shr, rotl and rotr) to clarify the direction and avoid using -ve amounts, e.g. both of these examples shift r3 to the right by 6 places.

```
sh r3 -6
shr r3 6
```

With short variants, the src value (i.e. the shift/rotate amount) is always a literal, so all short mnemonics use this alternative format (shls, shrs, rotls, rotrs). The 4-bit short literal is just large enough to allow shift/rotates of 1 to 8 in either direction.

```
shrs r3 8      ;shift r3 right by 8 places
rotls @r6 8    ;rotate the memory location @r6 left by 8 places
```

8.8 Accumulator

This 32-bit register can be accessed by the 'misc format' instructions and 'sys' extra-codes. The functions duplicate those available for the 16-bit work registers but include more arithmetic functions such as multiply and divide.

Accumulator instructions requiring a 32-bit non-literal operand use the concatenated value from the two specified 16-bit work registers.

Accumulator instructions requiring a 32-bit literal operand are available in 2 variants:

- The long variant uses the 2 words following the instruction as the literal (ms first).
- The short variant uses the word following the instruction as the ls 16 bits of the literal, with the ms part set to zero.

The Plasm assembler identifies long instructions which may be candidates for short variants.

8.9 Sounds/MIDI

PlasMa can generate two type of monophonic[11] sound: audio tones or MIDI note-on/note-off messages. PleX supports both of these, but only one type can be active at any one time.

The Play Note and End Note functions allow the programmer to drive the audio or MIDI system in two different ways:

1. The user program controls the timing.
   Start the note using Play Note with an 'infinite' time interval and stop it some time later using the End Note function. Any concept of tempo or gate-time is handled by the user program. This is similar to conventional sequencer-like programs but can cause a 'stuck note' on the external device if you fail to terminate it.

2. The system controls the timing.
   Start the note using Play Note with the required musical time interval, and let the system handle the note termination. The timing is defined by the tempo and gate-time which the user program sets with the Set Note Attribs function. End Note can still be used to abort a note prematurely if required.

The Play Note function initiates the note using a Note Definition value along with global values which have previously been defined with the Note Attribute value in the Set Note Attribs function.

Audio notes are sent to the built-in speaker if it is in Notes mode (see the Machine Manual), otherwise you will just hear the normal clicking sound from jump instructions. Sounds can only be heard on the real machine; the PlasMaSim simulator is unable to play any sounds but the note on/off timing is still simulated on the SND light.

MIDI notes are sent to the MIDI out socket so the resultant sound (or action) depends on the connected device. When a note is initiated, a MIDI note-on message is sent. If a musical time interval was used, a MIDI note-off message is sent automatically when the interval has elapsed. If an 'infinite' interval was used, a MIDI note-off message is sent when End Note is called, or when the 'infinite' time elapses; see footnote 12.

8.9.1 Note Attribute

A note attribute is specified in the Set Note Attribs function and qualifies subsequent notes initiated by the Play Note function. Attributes comprise 4 nibbles in the format:

```
cgtt
```

For audio notes, **c** is reserved.
For MIDI notes, **c** is the 4-bit channel number for MIDI notes; $0 = channel 1, $F = channel 16.

g is the 4-bit gate-time ratio which is used to divide the note interval between sound and silence to provide varying amounts of staccato through to legato:-

$1 = 6% on, 94% off (staccato)	$9 = 60% on, 40% off
$2 = 13% on, 87% off	$A = 66% on, 34% off
$3 = 20% on, 80% off	$B = 73% on, 27% off
$4 = 26% on, 74% off	$C = 80% on, 20% off
$5 = 33% on, 67% off	$D = 86% on, 14% off
$6 = 40% on, 60% off	$E = 93% on, 7% off
$7 = 46% on, 54% off	$F = 100% on, 0% off (legato)
$8 = 50% on, 50% off	

The function does nothing if the value is outside this range.

tt is an 8-bit value representing the tempo minus 30, so values 0 to 255 correspond to tempos 30 to 285 beats per minute, e.g. tempo 60 is $1E, tempo 120 is $5A.

11 You cannot initiate a note if a note is already playing.

8.9.2 Note Definition

A note definition is specified in the Play Note function, and comprises 16 bits in the format:

```
tnnn nnnn vvvv iiii
```

t is the type of note: 0 = audio, 1 = MIDI.

n is a 7-bit note number using standard MIDI definitions (this applies even if it's an audio note), e.g. 0=C-1, 47($2F)=B2, 60($3C)=C4 (middle C), 127=G9.

MIDI notes can use the full range of **n**, but audio notes are unable to play sounds below B2 so are capped at this value.

i is a 4-bit interval/length where values $0 to $E are interpreted as standard musical note times, and $F is interpreted as 'infinite'[12]:-

$0 = 1/32 (demisemiquaver)	$8 = 1/2 (minim)
$1 = 1/32 dotted	$9 = 1/2 dotted
$2 = 1/16 (semiquaver)	$A = 1 (semibreve)
$3 = 1/16 dotted	$B = 1 dotted
$4 = 1/8 (quaver)	$C = 2 (breve)
$5 = 1/8 dotted	$D = 2 dotted
$6 = 1/4 (crotchet)	$E = reserved
$7 = 1/4 dotted	$F = non-musical 'infinite'

For the musical intervals, the actual times used for the audible and muted parts of the note are dependent on the tempo and gate ratio values set by the last call to Set Note Attribs. The note is not initiated if the calculated note length is shorter than the machine's clock resolution (currently 10ms).

The 'infinite' interval gives the programmer full control over the on/off timing. Use the End Note function to terminate the note.

The musical intervals are all multiples of 1/64 notes, so the Note Time To Interval system function can be used for convenience to derive the interval code from the number of 64th notes required.

For both audio and MIDI notes, if **v** is zero, the note is interpreted as a **rest**, in which case there is no sound for the whole interval (the gate ratio is ignored). The Get Note Status function will respond with BUSY during this time.

The End Note function can be used at any time to terminate an audio or MIDI note (or rest) prematurely.

For MIDI notes, note-on and note-off messages are handled automatically by the Play Note and End Note functions.

For audio notes, if **v** is non-zero, notes will play at the volume set by the Speaker Volume pot (see Machine manual); the actual value of **v** is irrelevant.

For MIDI notes, if **v** is non-zero (1 to 15), the value is scaled up and used as the standard MIDI velocity. The latter has a range of 1 to 127 so the algorithm used is:

velocity = ((**v** + 1) x 8) - 1

This gives a velocity range of 15 to 127.

12 'Infinite' means 10 mins 55 secs with the current firmware.

9. Instruction table - PleX

Terminology:

- n[bbb...] or nn[bbb...] = 16-bit instruction code
  where n's are hex nibbles, and b's are the subsequent binary bits in groups of 4.
- (s) = signed, (u) = unsigned
- V,N,Z,C,X = flags
- ru.rv means concatenate ru with rv to form a 32-bit value; ru is the ms part.

Plasm syntax terminology:

- src = source operand, dst = destination operand
- m = operand address mode
- ar = address relative, e.g. 5 -42
- n = literal number, e.g. 7 -3 $e

9.1 General format

These are of the form: function dst src
where dst and src can use any of the 4 operand address modes independently.

bits desc
4 function ($1 to $D)
4 dst register, dddd
2 dst address mode, mm
4 src register, ssss
2 src address mode, mm

Code	Description	Flags	Plasm syntax
1[dddd mmss ssmm]	**move** dst = src dddd = destn, 0 to 14 = reg, 15 = long literal ssss = source, 0 to 14 = reg, 15 = long literal mm = operand address mode, 0 to 3	NZ	mov dst src
2[dddd mmss ssmm]	**add** dst = dst + src V=1 if signed error, C=1 if unsigned error	VNZC	add dst src
3[dddd mmss ssmm]	**subtract** dst = dst - src V=1 if signed error, C=1 if unsigned error	VNZC	sub dst src
4[dddd mmss ssmm]	**shift** dst = dst << src(s) if src +ve, shift **left** fill ls-bit from X, shift ms-bit into C if src -ve, shift **right** fill ms-bit from C, shift ls-bit into X	 NZC NZX	sh dst src(s) if src=lit n shl dst n shr dst n
5[dddd mmss ssmm]	**rotate** dst = dst <rot< src(s) if src +ve, shift **left** fill ls-bit from ms-bit, copy ls-bit into C if src -ve, shift **right** fill ms-bit from ls-bit, copy ms-bit into C	 NZC NZC	rot dst src(s) if src=lit n rotl dst n rotr dst n

Code	Description	Flags	Plasm syntax
6[dddd mmss ssmm]	**compare** flags = dst - src V=1 if dst(s) < src(s) N=1 if dst(u) < src(u) Z=1 if dst=src	VNZ	cmp dst src
7[dddd mmss ssmm]	**and** dst = dst & src	NZ	and dst src
8[dddd mmss ssmm]	**or** dst = dst | src	NZ	or dst src
9[dddd mmss ssmm]	**xor** dst = dst ^ src	NZ	xor dst src
A[dddd mmss ssmm]	**not** dst = src ^ 0xFFFF	NZ	not dst src
B[dddd mmss ssmm]	**add with carry** dst = dst + src + C V=1 if signed error, C=1 if unsigned error	VNZC	adc dst src
C[dddd mmss ssmm]	**subtract with borrow** dst = dst - src - C V=1 if signed error, C=1 if unsigned error	VNZC	sbb dst src
D[dddd mmss ssmm]	**swap bytes** e.g. if src = $1234, dst = $3412	NZ	swb dst src

9.2 Jump/Call general format

These are of the form: function condition type address
where condition tests the flag state(s), and address is either absolute or relative to PC.
All 4 address modes can be used.

bits desc
4 $F
1 invert condition, i
3 condition, ccc
2 type (jump/call, absolute/relative)
4 src register, ssss
2 src address mode, mm

Code	Description	Flags	Plasm syntax
F[iccc 00ss ssmm]	**jump** c = condition code 000 always jump 001 jump if X set 010 jump if C set 011 jump if Z set 100 jump if N set 101 jump if V set 110 jump if both Z and N unset 111 reserved i = invert condition logic: 0=normal, 1=invert ssss = source, 0-14 = reg, 15 = long literal mm = operand address mode, 0 to 3		jmp src jmp.x src jmp.c src jmp.z src jmp.n src jmp.v src jmp.g src jmp.nx src jmp.nc src etc
F[iccc 0011 0010]	**return** This is the same as: jmp @re+ (jump SP indirect, post_increment) c = condition code 000 always return 001 return if X set 010 return if C set 011 return if Z set 100 return if N set 101 return if V set 110 return if both Z and N unset 111 reserved i = invert condition logic: 0=normal, 1=invert		ret src ret.x src ret.c src ret.z src ret.n src ret.v src ret.g src ret.nx src ret.nc src etc
F[iccc 01ss ssmm]	**call** push ret addr, jump to src i, c, s, m: as above Top of stack = address of next instruction		call src call.f src call.nf src etc
F[iccc 10ss ssmm]	**jump relative** jump to PC + src(s) i, c, s, m: as above		jmpr src jmpr.f src jmpr.nf src etc
F[iccc 11ss ssmm]	**call relative** push ret addr, jump to PC + src(s) i, c, s, m: as above Top of stack = address of next instruction		callr src callr.f src callr.nf src etc

9.3 Short literal format

These are of the form: function dst n
where dst is restricted to 2 address modes (direct or indirect), and n is a literal value 0..15 (0..$F).

bits desc
4 $0
3 function ($1 to $7)
4 dst reg, dddd
1 dst address mode (direct or indirect), m
4 src literal (0..15), nnnn

Code	Description	Flags	Plasm syntax
0200-03FF 0[001d dddm nnnn]	**move short literal** dst = n dddd = dst, 0 to 14 = reg, 15 = long literal m = direct(0)/indirect(1) nnnn = src short literal, 0 to 15	NZ	movs dst n movs @dst n
0400-05FF 0[010d dddm nnnn]	**add short literal** dst = dst + n V=1 if signed error, C=1 if unsigned error	VNZC	adds dst n adds @dst n
0600-07FF 0[011d dddm nnnn]	**subtract short literal** dst = dst - n V=1 if signed error, C=1 if unsigned error	VNZC	subs dst n subs @dst n
0800-09FF 0[100d dddm nnnn]	**shift short literal** dst = dst << n(s) if src +ve, shift **left** (1 to 7 places) fill ls-bit from X, shift ms-bit into C if src = 0, shift **left** (8 places) fill ls-bit from X, shift ms-bit into C if src -ve, shift **right** (1 to 8 places) fill ms-bit from C, shift ls-bit into X	NZC NZX	shls dst 1-8 shls @dst 1-8 shrs dst 1-8 shrs @dst 1-8
0A00-0BFF 0[101d dddm nnnn]	**rotate short literal** dst = dst <rot< n(s) if src +ve, rotate **left** (1 to 7 places) fill ls-bit from ms-bit, copy ls-bit into C if src = 0, rotate **left** (8 places) fill ls-bit from ms-bit, copy ls-bit into C if src -ve, shift **right** (1 to 8 places) fill ms-bit from ls-bit, copy ms-bit into C	NZC NZC	rotls dst 1-8 rotls @dst 1-8 rotrs dst 1-8 rotrs @dst 1-8
0C00-0DFF 0[110d dddm nnnn]	**compare short literal** flags = dst - n V=1 if dst(s) < n(s) N=1 if dst(u) < n(u) Z=1 if dst=n	VNZ	cmps dst n cmps @dst n
0E00-0FFF 0[111d dddm nnnn]	**and short literal** dst = dst & n	NZ	ands dst n ands @dst n

9.4 Jump/Call short format

These are of the form: function condition type address
where condition tests the flag state(s), and address is a literal value relative to PC.

bits desc
4 $E
1 invert condition, i
3 condition, ccc
1 type (jump/call)
7 relative address (-64 to +63), aaaaaaa

Code	Description	Flags	Plasm syntax
E[iccc 0aaa aaaa]	**jump relative short literal** c = condition code, 0 to 7 000 always jump 001 jump if X set 010 jump if C set 011 jump if Z set 100 jump if N set 101 jump if V set 110 jump if >0 (both Z and N unset) 111 reserved i = invert condition logic: 0=normal, 1=invert a = relative address(s), -64 to +63		jmprs ar jmprs.x ar jmprs.c ar jmprs.z ar jmprs.n ar jmprs.v ar jmprs.g ar jmprs.nx ar jmprs.nc ar etc
E[iccc 1aaa aaaa]	**call relative short literal** c, i, a: as above Top of stack = address of next instruction		callrs ar callrs.f ar callrs.nf ar etc

9.5 Misc1 format

These have one operand and are of the form: function src/dst
where the operand is a src or dst depending on the function,
and is restricted to 2 address modes (direct or indirect).

bits desc
8 $00
3 function (0-1 = misc0, 2-7 = misc1)
4 src/dst register, dddd
1 src/dst address mode (direct or indirect), m

Code	Description	Flags	Plasm syntax
0000-003F	misc0 functions, see Misc0 format		
0040-005F 00[010d dddm]	**store flags** dst = flags dst ls-byte = VNZCX000 dddd = destn, 0 to 14 = reg, 15 = long literal m = direct(0)/indirect(1)		stf dst
0060-007F 00[011s sssm]	**load flags** flags = src src ls-byte = VNZCX--- ssss = source, 0 to 14 = reg, 15 = long literal m = direct(0)/indirect(1)	VNZCX	ldf src

Code	Description	Flags	Plasm syntax
0080-009F 00[100s sssm]	**system** 'function word' = src ssss = source, 0 to 14 = reg, 15 = long literal m = direct(0)/indirect(1) See System I/O function table - PleX	various	sys src
00A0-00BF 00[101d dddm]	**swap nibbles** dst = swapped dst e.g. dst before $1234, after = $4321 dddd = destn, 0 to 14 = reg, 15 = long literal m = direct(0)/indirect(1)	NZ	swn dst
00C0-01FF	reserved		

9.6 Misc0 format

These functions have zero operands within the instruction word, although some accumulator functions expect source literals in the word(s) following the instruction.
More accumulator and arithmetic functions are available using sys extra-codes; see System I/O function table - PleX.

bits desc
8 $00
2 $0
6 function

Code	Description	Flags	Plasm syntax
0000	**halt**		hlt
0001	**nop**		nop
0002	**clear carry flag**	C	clc
0003	**set carry flag**	C	stc
0004	**clear x flag**	X	clx
0005	**set x flag**	X	stx
0006	**push acc** ls 16 bits are pushed first, then ms 16 bits		pusha
0007	**pop acc** ms 16 bits are popped first, then ls 16 bits	NZ	popa
0008	**increment acc** acc = acc + 1 V=1 if signed error, C=1 if unsigned error	VNZC	inca
0009	**decrement acc** acc = acc - 1 V=1 if signed error, C=1 if unsigned error	VNZC	deca
000A	**negate acc** acc = -acc(s) (2's comp) V=1 if signed error	VNZ	nega
000B	**complement acc** acc = acc ^ $FFFFFFFF (1's comp)	NZ	cpla

Code	Description	Flags	Plasm syntax
000C	**swap words acc** e.g. before = $12345678, after = $56781234	NZ	swwa
000D	**swap bytes acc** e.g. before = $12345678, after = $34127856	NZ	swba
000E	**swap nibbles acc** e.g. before = $12345678, after = $87654321	NZ	swna
000F	**test acc** set flags depending on acc contents[13]	NZ	tsta
0010	**clear acc** acc = 0	NZ	clra
0011 nnnn	**load acc n short** acc = $0000nnnn	NZ	ldas n
0012 nnnn	**load acc n short signed** if nnnn +ve, acc = $0000nnnn if nnnn -ve, acc = $FFFFnnnn	NZ	ldass n
0013 nnnn(ms) nnnn(ls)	**load acc n** acc = nnnnnnnn	NZ	lda n
0014 nnnn	**add acc short n** acc = acc + nnnn V=1 if signed error, C=1 if unsigned error	VNZC	addas n
0015 nnnn(ms) nnnn(ls)	**add acc n** acc = acc + nnnnnnnn V=1 if signed error, C=1 if unsigned error	VNZC	adda n
0016 nnnn(ls)	**subtract acc short n** acc = acc - nnnn V=1 if signed error, C=1 if unsigned error	VNZC	subas n
0017 nnnn(ms) nnnn(ls)	**subtract acc n** acc = acc - nnnnnnnn V=1 if signed error, C=1 if unsigned error	VNZC	suba n
0018 nnnn	**shift acc n** acc = acc << nnnn(s) if nnnn +ve, shift **left** fill ls-bit from X, shift ms-bit into C if nnnn -ve, shift **right** fill ms-bit from C, shift ls-bit into X	 NZC NZX	sha n shla n(+ve) shra n(+ve)
0019 nnnn	**rotate acc n** acc = acc <rot< nnnn(s) if nnnn +ve, shift **left** fill ls-bit from ms-bit, copy ls-bit into C if nnnn -ve, shift **right** fill ms-bit from ls-bit, copy ms-bit into C	NZC	rota n rotla n(+ve) rotra n(+ve)

13 Setting flags for 16-bit registers can be done with various instructions using the same register for source and destn, e.g. mov r1 r1, or r5 r5, and r8 r8 etc.

Code	Description	Flags	Plasm syntax
001A nnnn	**compare acc short n** flags = acc - nnnn V = 1 if acc(s) < nnnn(s) N = 1 if acc(u) < nnnn(u) Z = 1 if acc = nnnn	VNZ	cmpas n
001B nnnn(ms) nnnn(ls)	**compare acc n** flags = acc - nnnnnnnn V = 1 if acc(s) < nnnnnnnn(s) N = 1 if acc(u) < nnnnnnnn(u) Z = 1 if acc = nnnnnnnn	VNZ	cmpa n
001C nnnn	**and acc short n** acc = acc & nnnn	NZ	andas n
001D nnnn(ms) nnnn(ls)	**and acc n** acc = acc & nnnnnnnn	NZ	anda n
001E nnnn	**or acc short n** acc = acc | nnnn	NZ	oras n
001F nnnn(ms) nnnn(ls)	**or acc n** acc = acc | nnnnnnnn	NZ	ora n
0020 nnnn	**xor acc short n** acc = acc ^ nnnn	NZ	xoras n
0021 nnnn(ms) nnnn(ls)	**xor acc n** acc = acc ^ nnnnnnnn	NZ	xora n
0022 nnnn	**add with carry acc short n** acc = acc + nnnn + C V=1 if signed error, C=1 if unsigned error	VNZC	adcas n
0023 nnnn(ms) nnnn(ls)	**add with carry acc n** acc = acc + nnnnnnnn + C V=1 if signed error, C=1 if unsigned error	VNZC	adca n
0024 nnnn	**subtract with borrow acc short n** acc = acc - nnnn - C V=1 if signed error, C=1 if unsigned error	VNZC	sbbas n
0025 nnnn(ms) nnnn(ls)	**subtract with borrow acc n** acc = acc - nnnnnnnn - C V=1 if signed error, C=1 if unsigned error	VNZC	sbba n
0026 nnnn	**unsigned multiply acc short n** acc = acc * nnnn(u) V=1 if error	VNZ	umulas n
0027 nnnn(ms) nnnn(ls)	**unsigned multiply acc n** acc = acc * nnnnnnnn(u) V=1 if error	VNZ	umula n
0028 nnnn	**signed multiply acc short n** acc = acc * nnnn(s) V=1 if error	VNZ	smulas n

Code	Description	Flags	Plasm syntax
0029 nnnn(ms) nnnn(ls)	**signed multiply acc n** acc = acc * nnnnnnnn(s) V=1 if error	VNZ	smula n
002A nnnn	**unsigned divide acc short n** acc = acc / nnnn(u) V=1 if error, C=1 if rd = 0 (div zero)	VNZC	udivas n
002B nnnn(ms) nnnn(ls)	**unsigned divide acc n** acc = acc / nnnnnnnn(u) V=1 if error, C=1 if rd = 0 (div zero)	VNZC	udiva n
002C nnnn	**signed divide acc short n** acc = acc / nnnn(s) V=1 if error, C=1 if rd = 0 (div zero)	VNZC	sdivas n
002D nnnn(ms) nnnn(ls)	**signed divide acc n** acc = acc / nnnnnnnn(s) V=1 if error, C=1 if rd = 0 (div zero)	VNZC	sdiva n
002E-003F	reserved		

10. System I/O function table - PleX

The 4 function word nibbles are **dcuv**, where **d** is the device ident, **c** is the device command, and **u** and **v** depend on the device.

d	c	u	v	flags	description
0	0	0	dest reg		**Platform Info** rv = platform ident and version number. Format: 2-bit platform ident, 14-bit version number. Platform ident: 0=Plasm machine 1=PlasMaSim simulator 2=Reserved 3=Reserved The format of the version number is platform-specific. For Plasm and PlasMaSim, the nibbles are binary-coded decimals to give mm.nn (major.minor), so the max version is 39.99
0	1	dest reg	dest reg		**Read Timer** ru = minutes. rv = seconds in 1/100ths of a second. Return the elapsed time since the timer was started. e.g. a time of 1 hour, 2 mins and 3.45 seconds will give: ru = 62, rv = 345. The maximum value for ru is 1439 (23 hours, 59 mins). The maximum value for rv is 5999 (59.99 secs). The operation is atomic so the timer does not have to be stopped first. The panel timer display is unaffected. Functions are available to convert either value into binary-coded decimal (BCD) to facilitate converting to ASCII.
0	2	0	0		**Stop Timer** Stop the timer used by the panel display.
0	3	0	0		**Reset/Start Timer** Reset the panel timer to zero and start it running. If it is already running, reset to zero and continue running. The panel display updates every second when the timer is running.

d	c	u	v	flags	description
0	4	0	delay no.	Z	**Poll Delay** Flag Z = delay status for delay number v. The function is ignored if the delay number is outside the range 0-3. Z = 1 if the specified delay has not elapsed. Z = 0 if the specified delay has elapsed, in which case the delay is re-primed automatically. Re-priming only occurs at the time of the poll, so for accurate delays, call the poll function as frequently as possible. Alternatively, construct your own delays using the Read Timer function. Use the Set Long Delay and Set Short Delay functions to initialise and prime the delays. The delay timers are independent of the system timer used for the panel display.
0	5	0	src rv		**Set Short Delay** Time delay = rv Set a short time delay from 0.01 secs to 59.99 secs as specified in rv. 2-bit delay no, 14-bit seconds in 1/100ths of a second = rv This is similar to the Set Long Delay function, but only needs a single register to define the delay. See Set Long Delay for more details.

d	c	u	v	flags	description
0	6	src ru	src rv		**Set Long Delay** Time delay = ru, rv Set a long time delay from 0.01 seconds to 23hrs 59mins 59.99 secs as specified in ru and rv: Hours & minutes in minutes = ru 2-bit delay no, 14-bit seconds in 1/100ths of a second = rv This function is used in conjunction with Poll Delay to provide up to 4 simple delays (delay nos. 0..3). e.g. to set delay 0 to be 1 hour, 2 mins and 3.45 seconds, set ru to 62 and rv to 345. To set delay 1 to the same value, ‘or’ rv with $4000. To set delay 2 to the same value, ‘or’ rv with $8000. To set delay 3 to the same value, ‘or’ rv with $C000. The function is ignored if ru exceeds 1439 (23 hrs, 59 mins), or if rv exceeds 5999 (59.99 secs). The delay timer is (re)primed whenever Set Long Delay is called, or when a poll indicates the timer has elapsed. For accurate delays, call the poll function as frequently as possible or use the Read Timer function directly. The Set Short Delay function can also be used. It is limited to delays of less than 1 minute, but only needs one register instead of 2.
0	7	src ru	dest	NZ	**Read Random** Generate a 16-bit random value in the range 0 to R inclusive, where R is the value in ru. If **v**=0..14, set the flags and write the status into rv If **v**=15, just set the flags Flags: N=1 if value<0 Z=1 if value=0 e.g. if R=7, the random number will be between 0 and 7 inclusive. R=0 has the same effect as R=$FFFF and gives a random number 0..$FFFF inclusive.

d	c	u	v	flags	description
0	8	0=lod-ms 1=lod-ls 2=brk-ms 3=brk-ls 4=mem	dest	NZ	**Read Switches** Read the value of the latching switches defined by **u**. If **v**=0..14, set the flags and write the status into rv If **v**=15, just set the flags Flags: N=1 if value<0 Z=1 if value=0 If **u**=4, the value of all 16 switches is returned regardless of the memory size in the current emulation.
1	0	0	0	Z	**Get Keypad Status** Flag Z = status If a button on the built-in keypad has been pressed since the previous Keypad Read call, set Z=0, otherwise set Z=1.
1	1	0	dst rv	Z	**Keypad Read** If a button on the built-in keypad has been pressed since the previous Keypad Read call, write the key code to rv and set Z=0, otherwise set Z=1. In all cases, the function returns immediately. Key codes are $00 to $0F depending on which of the 16 hex buttons are pressed.
2	0	0	0	Z	**Get Oper Status** Flag Z = status If the oper status is RDY (i.e. ready to accept commands), set Z=0, otherwise set Z=1.
2	1	dst ru	dst rv		**Oper Size** ru = max chars, rv = max lines. The resisters are loaded in the order ru-rv, so if the same register is used for both, the value will be max lines. Can be called at any time; returns immediately.

d	c	u	v	flags	description
2	2	src	src	Z	**Oper Set Coord** If the oper status is RDY, set the oper screen coordinates as shown below and set Z=0, otherwise set Z=1. If **u=**0..14, set coordX = ru If **u**=15, set coordX = 0 If **v=**0..14, set coordY = rv If **v**=15, set coordY = 0 In all cases, the function returns immediately. CoordX is the char number from the left, starting at 0. CoordY is the line number from the top, starting at 0. Values outside the width and height of the screen are ignored.
2	3	dst	dst	ZCV	**Oper Get Coord** If the oper status is RDY, return the current oper screen coordinates as shown below and set Z=0, otherwise set Z=1. If **u=**0..14, set ru=coordX and set flag C If **u**=15, just set flag C If **v=**0..14, set rv=coordY and set flag V If **v**=15, just set flag V Flags: C=1 if coordX=0 V=1 if coordY=0 The nibbles u & v are processed in that order, so if the same register (0..14) is used for both, the value returned will be coordY. Flag C will still represent coordX. In all cases, the function returns immediately. CoordX is the char number from the left, starting at 0. CoordY is the line number from the top, starting at 0.

d	c	u	v	flags	description
2	4	0	src rv	Z	**Oper Write Char** Display = rv If the oper status is RDY, start the process of writing the ls-byte in rv as an ASCII character to the oper screen at the current X/Y coordinate, set Z=0 and set the status to BUSY, otherwise set Z=1. In all cases, the function returns immediately. The status remains BUSY and changes to RDY when the process is completed; the current coordinates will then be: X=X+1 (with wrap-round), Y unchanged. See also Oper Write Literal.
2	4	1	0	Z	**Oper Write Newline** If the oper status is RDY, start the process of writing a newline, set Z=0 and set the status to BUSY, otherwise set Z=1. In all cases, the function returns immediately. The status remains BUSY and changes to RDY when the process is completed; the current coordinates will then be: X=0, Y=Y+1 (with wrap-round).
2	4	2	tab	Z	**Oper Write Tab** If the oper status is RDY, start the process of writing spaces up to the next tab position (using **v** as the tab spacing), set Z=0 and set the status to BUSY, otherwise set Z=1. In all cases, the function returns immediately. The status remains BUSY and changes to RDY when the process is completed; the current coordinates will then be: X=X+n (with wrap-round), Y unchanged. See also Oper Tab X Coord and Oper BackTab X Coord which move the X coordinate *without* writing spaces.
2	4	3	src rv line number	Z	**Oper Write Horiz Line 1** This is the same as Oper Clear Line except that a single horizontal line is drawn at the specified line number, and the coordinates are set to X=0, Y=line+1 (with wrap-round). If the line number is out of range, the function does nothing.

d	c	u	v	flags	description
2	4	4	src rv line number	Z	**Oper Write Horiz Line 2** This is the same as Oper Clear Line except that a double horizontal line is drawn at the specified line number, and the coordinates are set to X=0, Y=line+1 (with wrap-round). If the line number is out of range, the function does nothing.
2	4	5	src rv column number	Z	**Oper Write Vert Line 1** This is the same as Oper Write Horiz Line 1 except that a single vertical line is drawn at the specified column number, and the coordinates are set to X=col+1 (with wrap-round), Y=0. If the column number is out of range, the function does nothing.
2	4	6	src rv column number	Z	**Oper Write Vert Line 2** This is the same as Oper Write Horiz Line 1 except that a double vertical line is drawn at the specified column number, and the coordinates are set to X=col+1 (with wrap-round), Y=0. If the column number is out of range, the function does nothing.
2	4	7	0	Z	**Oper Inc X Coord** If the oper status is RDY, increment the current oper screen X coordinate (with wrap-round) and set Z=0, otherwise set Z=1. In all cases, the function returns immediately.
2	4	7	1	Z	**Oper Dec X Coord** This is the same as Oper Inc X Coord except that the X coordinate is decremented.
2	4	7	2	Z	**Oper Inc Y Coord** This is the same as Oper Inc X Coord except that the Y coordinate is incremented.
2	4	7	3	Z	**Oper Dec Y Coord** This is the same as Oper Inc X Coord except that the Y coordinate is decremented.

d	c	u	v	flags	description
2	4	8	tab	Z	**Oper Tab X Coord** If the oper status is RDY, increment the current oper screen X coordinate to the next tab position with wrap-round (using **v** as the tab spacing) and set Z=0, otherwise set Z=1. In all cases, the function returns immediately. This function only changes the X coordinate, as opposed to Oper Write Tab which writes spaces up to the tab position.
2	4	9	tab	Z	**Oper BackTab X Coord** This is the same as Oper Tab X Coord except that the X coordinate is decremented to the previous tab position (using **v** as the tab spacing).
2	5	ms char	ls char	Z	**Oper Write Literal** Display = char This is the same as Oper Write Char except that the byte is the 8-bit value in the **u** and **v** nibbles of the instruction instead of the contents of rv[14].
2	6	srv ru top left	src rv bottom right	Z	**Oper Draw Box 1** If the oper status is RDY, start the process of drawing a box and change the status to BUSY, set Z=0 and set the status to BUSY, otherwise set Z=1. In all cases, the function returns immediately. The status remains BUSY and changes to RDY when the process is completed. The box is defined by: ms-byte ru = top: 0 to (height-1) ls-byte ru = left: 0 to (width-1) ms-byte rv = bottom: 0 to (height-1) ls-byte rv = right: 0 to (width-1) When the process completes, X=left+1, Y=top+1 (with wrap-round). If any coordinates are out of range, or if right <= left, or bottom <= top, the function does nothing.
2	7	srv ru top left	src rv bottom right	Z	**Oper Draw Box 2** Same as Oper Draw Box 1 except the vertical lines are doubled.

14 This can be used to display short messages without tying up a register.

d	c	u	v	flags	description
2	8	srv ru top left	src rv bottom right	Z	**Oper Draw Box 3** Same as Oper Draw Box 1 except the horizontal lines are doubled.
2	9	srv ru top left	src rv bottom right	Z	**Oper Draw Box 4** Same as Oper Draw Box 1 except the vertical and horizontal lines are doubled.
2	A	0	0	Z	**Oper Clear** If the oper status is RDY, start the process of clearing the screen, set Z=0 and set the status to BUSY, otherwise set Z=1. In all cases, the function returns immediately. The status remains BUSY and changes to RDY when the process is completed; the coordinates will then be: X=0, Y=0.
2	A	1	src rv line number	Z	**Oper Clear Line** This is the same as Oper Clear except that a single line is cleared, and the coordinates are set to X=0, Y=line. If the line number is out of range, the function does nothing.
2	A	2	0	Z	**Oper Clear To EOL** This is the same as Oper Clear except that the current line is cleared from the current value of X to the end of line, and X is set to 0.
2	A	3	src rv line number	Z	**Oper Clear Upper** This is the same as Oper Clear except that lines from the top to the specified line number inclusive are cleared, and the coordinates are set to X=0, Y=0. If the line number is out of range, the function does nothing.
2	A	4	src rv line number	Z	**Oper Clear Lower** This is the same as Oper Clear except that lines from the specified line number to the bottom are cleared, and the coordinates are set to X=0, Y=line. If the line number is out of range, the function does nothing.

d	c	u	v	flags	description
2	B	srv ru field range	src rv line number	Z	**Oper Clear Field** This is similar to Oper Clear Line except that only part of the line is cleared. The field to be cleared is defined in ru: ms-byte = start column: 0 to (width-1) ls-byte = end column: 0 to (width-1) When the process completes, X=start column, Y=line. If the start, end, or line number are out of range, or if start is less than end, the function does nothing.
3	0	0	dest	VNZC	**PTR Get Status** Read the status for the paper tape reader If **v**=0..14, set the flags and write the status into rv If **v**=15, just set the flags Flags: Z=1 if BUSY V=1 if EOT* C=1 if RDY N=1 if DATA_RDY Status values: 0=OFF 1=RDY 2=BUSY 3=unused 4=DATA_RDY 5=ERR_DATA (data value unexpected) 6=ERR_READ (media) 7=ERR_PARITY (only used by the 7-bit+parity tape reader) * EOT is only returned in the flag; it does not appear in rv as it causes the tape to be unloaded so rv returns OFF instead.

d	c	u	v	flags	description
3	1	0	0		**PTR Req Data** If the tape reader status is RDY, request the next data value from the paper tape reader and set the status to BUSY, otherwise do nothing. In all cases, the function returns immediately. The status remains BUSY until some indeterminate time later, after which, if it is DATA_RDY, the data value has been read successfully from the paper tape and is now held in the device's read buffer ready to be fetched via the next PTR Get function. The buffer only holds a single data value so it will be overwritten each time a request is actioned. The data value depends on the current tape reader mode: either 7-bit plus parity, or 8-bit no parity. In 7-bit mode, the data from the tape is treated as a 7-bit ASCII character in the range $00 to $7F. The reader checks the parity bit and sets the status to ERR_PARITY if the check fails. In 8-bit mode, the data returned is an 8-bit value in the range $00 to $FF. No parity checking is performed.
3	2	0	dest rv	NZ	**PTR Get Data** rv = data N=1 if rv<0, Z=1 if rv=0 If the reader status is DATA_RDY, transfer the data value from the device's read buffer to rv, setting flags as shown, and set the status to RDY, otherwise do nothing. The data corresponds to the most recent PTR Req Data call. In all cases, the function returns immediately.

d	c	u	v	flags	description
4	0	0	dest	VZC	**PTP Get Status** Read the status for the paper tape punch If **v**=0..14, set the flags and write the status into rv If **v**=15, just set the flags Flags: Z=1 if BUSY V=1 if EOT* C=1 if RDY Status values: 0=OFF 1=RDY 2=BUSY 3=unused 4=ERR_WRITE (media) * EOT is only returned in the flag; it does not appear in rv as it causes the tape to be unloaded so rv returns OFF instead.
4	1	0	dest rv		**PTP Write Data** data = rv If the punch status is RDY, write the ls-byte in rv to the punch device's write buffer ready for punching to paper-tape and set the status to BUSY, otherwise do nothing. In all cases, the function returns immediately. The status remains BUSY until some indeterminate time later, after which, if it is RDY, the data value has been punched successfully onto the paper tape. The data value depends on the tape punch mode: either 7-bit plus parity, or 8-bit no parity. In 7-bit mode, the data is treated as a 7-bit ASCII character in the range $00 to $7F. The punch generates and writes a parity bit automatically. In 8-bit mode, the data is treated as an 8-bit value in the range $00 to $FF. No parity is written.

d	c	u	v	flags	description
5	0	mt deck	dest	VNZC	**MT Get Status** Read the status for mt deck number **u** (0..1) If **v**=0..14, set the flags and write the status into rv If **v**=15, just set the flags Flags: Z=1 if BUSY V=1 if WARN_EOT* C=1 if RDY N=1 if DATA_RDY Status values: 0=OFF 1=RDY 2=BUSY 3=WARN_EOT (end of tape imminent*) 4=TMARK (tape mark encountered) 5=DATA_RDY 6=ERR_DATA (data invalid; read after write) 7=ERR_READ (media error or checksum fail) 8=ERR_WPR (writing to tape with no 'write permit ring') 9=ERR_WRITE (media) 10=ERR_EOT (actual end of tape*) * WARN_EOT is an early warning for end-of-tape and is returned instead of RDY if there are less than 8 blocks remaining on the tape. Reads & writes are still allowed after this point so logical volumes can be 'closed' with additional blocks and/or tape-marks. ERR_EOT is the actual end-of-tape and means no blocks remain.
5	1	mt deck	dest	Z	**MT Get Info** If the deck status is not OFF, return the current block size and WPR state for the deck, otherwise do nothing. If **v**=0..14, set the flag and write the block size into rv If **v**=15, just set the flag Flag: Z=1 if WPR fitted (i.e. tape can be overwritten) The block size is currently a fixed value of 256 bytes but may change in future. A block will occupy (block size / 2) words of memory.

d	c	u	v	flags	description
5	2	mt deck	0		**MT Req Block** If the specified deck status is RDY, TMARK or WARN_EOT*, request the next block of data from it and set the status to BUSY, otherwise do nothing. In all cases, the function returns immediately. The status remains BUSY until some indeterminate time later, after which, if it is DATA_RDY, the block has been read successfully from the tape and is now held in the device's read buffer ready to be fetched via the next MT Get Block function. The buffer only holds a single block so it will be overwritten each time a request is actioned. *The status is WARN_EOT instead of RDY if near the end of tape; see notes in MT Get Status.
5	3	mt deck	addr rv		**MT Get Block** rv = ram address for block from specified mt deck. If the deck's status is DATA_RDY, transfer the data block from the deck's read buffer to ram starting at the address in rv and set the status to RDY or WARN_EOT*, otherwise do nothing. The data corresponds to the most recent MT Req Block call. Memory locations within the Fixed store (FST) region will not be overwritten. Memory addresses will wrap round to zero if top of store is exceeded. In all cases, the function returns immediately. *The status is WARN_EOT instead of RDY if near the end of tape; see notes in MT Get Status.
5	4	mt deck	0		**MT Rewind** Rewind tape on specified mt deck, leaving it loaded. If the deck's status is OFF or BUSY, do nothing, otherwise start rewinding the tape and set the status to BUSY. The status remains BUSY until some indeterminate time later, after which, if it is RDY, the tape has been successfully rewound. The tape remains loaded; it can only be unloaded by the MT Unload switch.

d	c	u	v	flags	description
5	5	mt deck	src rv		**MT Skip Blocks Forwards** rv = number of blocks to skip on specified mt deck. If the deck's status is OFF or BUSY, do nothing, otherwise start skipping forwards by rv blocks and change the status to BUSY. If the number of blocks is zero, the function does nothing. The status remains BUSY until some indeterminate time later, after which, if it is RDY or WARN_EOT*, the tape has been successfully positioned. If a tape mark is encountered before all blocks are skipped, the tape is positioned after that tape mark and the status changes to TMARK. *The status is WARN_EOT instead of RDY if near the end of tape; see notes in MT Get Status.
5	6	mt deck	src rv		**MT Skip Blocks Backwards** rv = number of blocks to skip on specified mt deck This is similar to MT Skip Blocks Forwards except that the tape skips backwards by rv blocks. If a tape mark is encountered before all blocks are skipped, the tape is positioned before that tape mark and the status changes to TMARK. If start of tape is reached before all blocks are skipped, the tape is positioned at the start and the status changes to TMARK.
5	7	mt deck	src rv		**MT Skip Tape Marks Forwards** rv = number of tape marks to skip on specified mt deck. If the number of tape marks is zero, the function does nothing. This is similar to MT Skip Blocks Forwards except that the tape skips forwards by rv tape marks. If end of tape (a pair of adjacent tape marks) is reached before all tape marks are skipped, the tape is positioned after the first of these tape marks and the status changes to TMARK.
5	8	mt deck	src rv		**MT Skip Tape Marks Backwards** rv = number of tape marks to skip on specified mt deck. This is similar to MT Skip Tape Marks Forwards except that the tape skips backwards by rv tape marks. If start of tape is reached prematurely, tape is positioned at the start and the status changes to TMARK.

d	c	u	v	flags	description
5	9	mt deck	addr rv		**MT Write Block** rv = ram address for block on specified mt deck. If the deck's status is RDY, TMARK or WARN_EOT*, write the data block from mem starting at the address in rv to the device's write buffer ready for writing to tape and change status to BUSY, otherwise do nothing. In all cases, the function returns immediately. The status remains BUSY until some indeterminate time later, after which, if it is RDY or WARN_EOT*, the data block has been written successfully to the mag tape. All data following a write is deemed invalid; attempts to read invalid data will return a status of ERR_DATA. If the tape is not fitted with a WPR, the status is set to ERR_WPR. Memory addresses will wrap round to zero if top of store is exceeded. *The status is WARN_EOT instead of RDY if near the end of tape; see notes in MT Get Status.
5	A	mt deck	tmarks		**MT Write Tape Marks** v = number of tape marks to be written on specified mt deck. If the deck's status is RDY, TMARK or WARN_EOT*, start the process of writing **v** tape marks and change the status to BUSY. The status remains BUSY until some indeterminate time later, after which, if it is RDY or WARN_EOT*, the tape mark(s) have been successfully written. All data following a write is deemed invalid; attempts to read invalid data will return a status of ERR_DATA. If the number of tape marks is not in the range 1..2, do nothing. In all cases, the function returns immediately. *The status is WARN_EOT instead of RDY if near the end of tape; see notes in MT Get Status.

d	c	u	v	flags	description
6	0	drive	dest	VNZC	**EDS Get Status** Read the status for eds drive **u** (0..1) If **v**=0..14, set the flags and write the status into rv If **v**=15, just set the flags Flags: Z=1 if BUSY V=1 if EOD C=1 if RDY N=1 if DATA_RDY Status values: 0=OFF 1=RDY 2=BUSY 3=EOD (end of disc, sector number too big) 4=DATA_RDY 5=ERR_READ (media) 6=ERR_WRITE (media)
6	1	drive	dest rv		**EDS Get Sector Info** rv = sector size for specified eds drive. acc = total number of sectors on specified eds drive. If the drive status is not OFF, read the current sector size (in bytes) into rv, and the total number of sectors into acc, otherwise do nothing. The sector size is currently a fixed value but may change in future. A sector will occupy (sector size / 2) words of memory. The total includes both used and unused sectors. It is capped to realistic values even if high capacity sd-cards are used.
6	2	drive	0		**EDS Set Sector Number** Sector number = acc If the drive status is RDY, set the current sector number to the value held in acc, otherwise do nothing. In all cases, the function returns immediately. The current sector number is used by subsequent calls to EDS Req Sector, and range from 0 to the max available for the loaded disc. The value is only checked when a read or write is requested. The current sector is set to zero when a disc is loaded.

d	c	u	v	flags	description
6	3	drive	0	NZ	**EDS Get Sector Number** acc = current sector number N=1 if acc<0, Z=1 if acc=0 If the drive status is RDY, set acc to the current sector number and set flags as shown, otherwise do nothing. In all cases, the function returns immediately.
6	4	drive	0		**EDS Req Sector** If the drive status is RDY, request the contents of the current sector from it and set the status to BUSY, otherwise do nothing. In all cases, the function returns immediately. The status remains BUSY until some indeterminate time later, after which, if it is DATA_RDY, the sector has been read successfully from the disc and is now held in the device's read buffer ready to be fetched via the next EDS Get Sector function. Other status values are possible, such as ERR_EOD if the current sector number is out of range. The buffer only holds a single sector so it will be overwritten each time a request is actioned.
6	5	drive	addr rv		**EDS Get Sector** rv = ram address for sector from specified eds drive. If the drive's status is DATA_RDY, transfer the data sector from the drive's read buffer to ram starting at the address in rv, increment the current sector number and set the status to RDY, otherwise do nothing. The data corresponds to the most recent EDS Req Sector call. Memory locations within the Fixed store (FST) region will not be overwritten. Memory addresses will wrap round to zero if top of store is exceeded. In all cases, the function returns immediately.

d	c	u	v	flags	description
6	6	drive	addr rv		**EDS Write Sector** rv = ram address for sector on specified eds drive. If the drive's status is RDY, write the data sector from mem starting at the address in rv to the device's write buffer ready for writing to disc using the current sector number and change status to BUSY, otherwise do nothing. In all cases, the function returns immediately. The status remains BUSY until some indeterminate time later, after which, if it is RDY, the data block has been written successfully to the disc. Other status values are possible, such as ERR_EOD if the current sector number is out of range. Memory addresses will wrap round to zero if top of store is exceeded.
7	0	0	0	Z	**LP Get Status** Flag Z = status If the printer is ready to accept commands, set Z=0 (RDY), otherwise set Z=1 (BUSY)[15] This device is not simulated in PlasMaSim.
7	1	0	src rv	Z	**LP Write Data** Printer data = rv If the printer status is RDY, start the process of writing the ls-byte in rv as an ASCII character to the printer, set Z=0 and set the status to BUSY, otherwise set Z=1. In all cases, the function returns immediately. The status remains BUSY and changes to RDY when the process has completed successfully. See footnote in LP Get Status for clarification of BUSY.
7	2	ms char	ls char	Z	**LP Write Literal** Printer data = char This is the same as LP Write Data except that the byte is the 8-bit value in the **u** and **v** nibbles of the instruction instead of the contents of rv.

15 The h/w monitors a single data line from the printer interface to keep things simple, the trade-off being that a program cannot tell *why* the printer is busy (it could be processing a transfer, waiting for paper, offline, or not connected). This is a reasonable trade-off, but the strategy breaks down if the printer is plugged in but powered *off*, in which case it still appears to be ready. This needs a small hardware mod to fix.

d	c	u	v	flags	description
8	0	0	src rv		**Set Note Attribs** Attribs = rv Set the Note Attribute for subsequent notes played via the Play Note function. Attribs can be changed at any time but only take effect at the next Play Note.
8	1	0	src rv	Z	**Play Note** Audio/MIDI note = rv If the note handler status is RDY, initiate a note according to the Note Definition value in rv using attributes defined by the last call to Set Note Attribs, set Z=0 and set the status to BUSY, otherwise set Z=1. In all cases, the function returns immediately. The status remains BUSY until the total note interval has finished, after which it changes to RDY.
8	1	1	0		**End Note** If a note is currently active, request it to be terminated as soon as possible, otherwise do nothing. If a note was active, the note status will still remain BUSY until the note has actually terminated, after which it changes to RDY. In all cases, the function returns immediately.
8	2	0	0	Z	**Get Note Status** Flag Z = status If there is no audio or MIDI note currently active (meaning Play Note is ready to accept requests), set Z=0 (RDY), otherwise set Z=1 (BUSY)
8	3	src ru	dst rv		**Note Time To Interval** rv (interval code) = ru (64^{th} notes) Convert the note time measured in units of 64^{th} notes (2 to 192) to the interval code ($0 to $D) used in the Note Definition for Play Note, e.g. if ru = 4, rv is set to 2 (4/64 = 1/16 semiquaver) if ru = 24, rv is set to 7 (24/64 = 3/8 = 1/4 dotted) If ru is outside the range 2 to 192, it is constrained to those values. If ru does not map exactly to a recognised interval code, it is rounded up to the next valid code.

d	c	u	v	flags	description
9	0	src ru	dst rv	C	**Convert Mins to BCDHrsMins** rv BCD hrs mins = ru mins. C=1 if the binary number exceeds 5999 (BCD $9959). Convert the binary number of minutes in ru to a binary-coded decimal (BCD) value of hours & mins in rv. e.g. if ru = 62, rv = $0102 (62 mins = 1 hour 2 mins).
9	1	src ru	dst rv	C	**Convert BCDHrsMins to Mins** rv mins = ru BCD hrs mins. C=1 if any BCD nibbles exceed 9, or if the ls two nibbles exceed $59. Convert the BCD hours & mins value in ru to minutes in rv. e.g. if ru = $0215 (2 hours 15 mins), rv = 135.
9	2	src ru	dst rv	C	**Convert Reg to BCD** rv BCD = ru binary. C=1 if the binary number exceeds 9999. Convert the binary number in ru to a binary-coded decimal (BCD) value in rv. e.g. if ru = 345, rv = $0345.
9	3	src ru	dst rv	NZC	**Convert BCD to Reg** ru binary = rv BCD. C=1 if any BCD nibbles exceed 9 or if the BCD number exceeds $9999. Convert the BCD value in ru to a binary number in rv. e.g. if ru = $0345, rv = 345.
9	4	src/dst ru	src rv	VNZ	**Unsigned Multiply Reg** ru = ru(u) * rv(u) V=1 if error
9	5	src/dst ru	src rv	VNZ	**Signed Multiply Reg** ru = ru(s) * rv(s) V=1 if error
9	6	src/dst ru	src/dst rv	VNZC	**Unsigned Divide Reg** ru = ru(u) / rv(u), rv = remainder V=1 if error, C=1 if rd = 0 (div zero)
9	7	src/dst ru	src/dst rv	VNZC	**Signed Divide Reg** ru = ru(s) / rv(s), rv = remainder V=1 if error, C=1 if rd = 0 (div zero)
A	0	0	src rv	NZ	**Load Acc from Reg UnSigned** acc(u) = rv(u)

d	c	u	v	flags	description
A	0	1	dst rv	NZC	**Store Acc to Reg Unsigned** rv(u) = acc(u) C=1 if error
A	0	2	src rv	NZ	**Load Acc from Reg Signed** acc(s) = rv(s)
A	0	3	dst rv	VNZ	**Store Acc to Reg Signed** rv(s) = acc(s) V=1 if error
A	0	4	src rv	NZ	**Load Acc Indirect** acc = mem[rv].mem[rv+1]
A	0	5	src rv		**Store Acc Indirect** mem[rv].mem[rv+1] = acc
A	0	6	src rv	NZC NZX	**Shift Acc** acc = acc << rv(s) if rv +ve, shift **left** fill ls-bit from X, shift ms-bit into C if rv -ve, shift **right** fill ms-bit from C, shift ls-bit into X
A	0	7	src rv	NZC	**Rotate Acc** acc = acc <rot< rv(s) if rv +ve, shift **left** fill ls-bit from ms-bit, copy ls-bit into C if rv -ve, shift **right** fill ms-bit from ls-bit, copy ms-bit into C
A	1	src ru	src rv	NZX	**Load Acc** acc = ru.rv
A	2	dst ru	dst rv		**Store Acc** ru.rv = acc
A	3	src ru	src rv	VNZC	**Add Acc** acc = acc + ru.rv V=1 if signed error, C=1 if unsigned error
A	4	src ru	src rv	VNZC	**Sub Acc** acc = acc - ru.rv V=1 if signed error, C=1 if unsigned error

d	c	u	v	flags	description
A	5	src ru	src rv	VNZ	**Compare Acc** flags = acc - ru.rv V = 1 if acc(s) < ru.rv(s) N = 1 if acc(u) < ru.rv(u) Z = 1 if acc = ru.rv
A	6	src ru	src rv	VNZC	**Add With Carry Acc** acc = acc + ru.rv + C V=1 if signed error, C=1 if unsigned error
A	7	src ru	src rv	VNZC	**Sub With Borrow Acc** acc = acc - ru.rv - C V=1 if signed error, C=1 if unsigned error
A	8	src ru	src rv	NZ	**Not Acc** acc = ru.rv ^ $FFFFFFFF
A	9	src ru	src rv	NZ	**And Acc** acc = acc & ru.rv
A	A	src ru	src rv	NZ	**Or Acc** acc = acc | ru.rv
A	B	src ru	src rv	NZ	**Xor Acc** acc = acc ^ ru.rv
A	C	src ru	src rv	VNZ	**Unsigned Multiply Acc** acc = acc(u) * ru.rv(u) V=1 if error
A	D	src ru	src rv	VNZ	**Signed Multiply Acc** acc = acc(s) * ru.rv(s) V=1 if error
A	E	src ru	src/dst rv	VNZC	**Unsigned Divide Acc** acc = acc(u) / ru.rv(u), ru.rv = remainder V=1 if error, C=1 if rd=0 (div zero)
A	F	src ru	src/dst rv	VNZC	**Signed Divide Acc** acc = acc(s) / ru.rv(s), ru.rv = remainder V=1 if error, C=1 if rd=0 (div zero)
B	0	dst ru	dst rv	C	**Convert Acc to BCD** ru.rv BCD = acc binary C=1 if the binary number exceeds 99999999. Convert the binary number in acc to a binary-coded decimal (BCD) value in ru and rv e.g. if acc=345678, ru=$0034 and rv = $5678.

d	c	u	v	flags	description
B	1	src ru	src rv	NZC	**Convert BCD to Acc** acc binary = ru.rv BCD. N=1 if acc<0, Z=1 if acc=0. Convert the BCD value in ru.rv to a binary number in acc. e.g. if ru=$0345 and rv = $67, acc = 34567. C=1 if any BCD nibbles exceed 9 or if the BCD number exceeds $99999999.
C	m	src ru	dst strz rv=&strz	NZ	**Convert N to ASCIIZ** Convert the source value to an ASCII string and write it to consecutive memory locations starting at the address in r**v**. The string is terminated with a zero. If **u** = 0..14, source value is ru (16 bits). If **u** = 15, source value is acc (32 bits). Flags N=1 if value<0, Z=1 if value=0. The 4 bits in **c** define the conversion mode:- bit 3 (ms): 0=decimal, 1=hex bit 2: if decimal, 0=unsigned, 1=signed if hex, 0=uppercase A-F, 1=lowercase a-f bit 1: 0=left-align, 1=right-align bit 0: padding if right-aligned, 0=spaces, 1=zeros Strings use the same format as the Plasm assembler; characters are stored in the ms & ls bytes of each 16-bit memory location. The zero terminator is stored in the ls-byte if there is room, or in the following word, e.g. if string = ‘1’, 1 word is written: $3100 if string = ‘12’, 2 words are written: $3132 $0000 if string = ‘123’, 2 words are written: $3132 $3300 Signed mode uses one extra byte for the sign; either a minus sign if negative or a space character if positive. The actual number of words written depends on the source value, but the maximums are:- 16-bit decimal unsigned: 3 16-bit decimal signed: 4 16-bit hex: 3 32-bit decimal unsigned: 6 32-bit decimal signed: 6 32-bit hex: 4

d	c	u	v	flags	description
D	0	0	0	ZC	**Get Keyboard Status** Flags = status Set C=1 if PS/2 keyboard not connected or is incompatible[16], otherwise set C=0. If a key has been pressed since the previous Keyboard Read call, set Z=0, otherwise set Z=1. This device is not simulated in PlasMaSim.
D	1	0	dst rv	NZC	**Keyboard Read** Set C=1 if PS/2 keyboard not connected or is incompatible, otherwise set C=0. If a key has been pressed since the previous Keyboard Read call, write the ASCII key value to rv and set Z=0, otherwise set Z=1. Set N=1 if key is a normal printable character (see below), otherwise set N=0. In all cases, the function returns immediately. Key values are $00 to $FF: $00 = key already handled, e.g. shift, caps lock $20..$7E = printable ASCII characters (flag N=1) $80..$FF = control keys

16 Keyboard must support code-set 3. Layout assumed to be basic US; numlock not yet supported.