PlasMa

aka Phil’s lights and switches Machine

Mini-Mainframe Simulator Project Instruction Set and I/O Functions Manual - Toy-B

Phil Tipping

www.philizound.co.uk

Table of Contents

1. Introduction

This describes the Toy-B instruction set, which is one of several supported by the PlasMa machine. Syntax details are included for the Plasm assembler, available from the philizound.co.uk website.

This instruction set is another variant of the ‘Toy’ computer used as a teaching aid at Princeton university, USA. It is similar to Toy-A but is based on the CS 126 Lectures A1/A2 for the TOY Machine by Randy Wang.

The main differences from Toy-A are:

- New indexed addressing option for all 'format 2' instructions. Note this restricts the 'd' register number to 0-7 as opposed to 0-15.
- Opcode 'branch zero' (jz) is withdrawn.
- The above changes make room for 3 new opcodes:
  - Multiply (mul)
  - Jump and count (djnz)[1]
  - System call (sys) for I/O, which means memory address $FF can be used as a normal memory location. The Princeton system only mentioned two I/O functions[2] so I've added a few from the PleX design for added interest to the programmer; see System and I/O overview and I/O function table - Toy-B.
- Register 0 is no longer fixed at zero, and is a normal read-write register.

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

2. Acknowledgements

The Toy 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 details 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

Thanks also to Adrian Rawson for suggestions and support.

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

1 I’ve renamed this to 'decrement and jump if not zero' to clarify when the decrement actually occurs.
2 I only found examples for reading numbers from the keypad and writing numbers to the TTY device.

4. Architecture

16-bit instructions.

256 x 16-bit words of volatile main memory.

16 x 16-bit registers, r0, r1, r2… r9, ra, rb… re, rf (all read/write).

5. System and I/O overview

Unlike Toy-A, this variant has a dedicated ‘system call’ opcode which sends and receives a 16-bit value to and from a specified device.

The Princeton documentation only shows two I/O functions. These are for accessing a TTY-like device for numeric I/O:

- TTY-Write outputs the 16-bit value in hex and decimal to the display screen.
- TTY-Read causes the program to halt while hex digits are entered from the keypad.

It’s not clear if other devices or functions were available, but I’ve included I/O functions and ‘extra-codes’ from the PleX design for added interest to the programmer[3], hopefully without compromising backwards compatibility.

5.1 Built-in peripherals

- Teletype (TTY) screen and keypad
- Paper tape reader and punch for serial data
- Two rings of lights for display purposes

The TTY screen uses the built-in display. System functions are available to convert and displays 16-bit values in hex and decimal, and to display ASCII characters. A secondary input function is available to read the keypad without stopping the program.

The paper tape reader and punch uses individual plug-in sd-cards; see the Machine Manual for details.

Other 'devices' include switch states, timers/delays etc. Other extra-codes include maths and conversion functions not available in the Toy-B ISA.

5.2 External peripherals

- Printer functions using the parallel interface socket or other 5v-level devices

6. Instructions overview

Instructions are encoded in a similar way to Toy-A, where the left-hand (ms) nibble is the opcode, and the others are encoded in (mostly) two ways, called format 1 and 2.

6.1 Format 1

The 3 nibbles after the opcode refer to register numbers. The first is the destination reg **d** and the next two are source regs **s** and **t**.

Example 1: $15A8 decodes as opcode 1, dest reg 5, source regs A and 8.
This translates to **add** where R[d] ← R[s] + R[t], so the result of adding the contents of reg A to the contents of reg 8 is written to reg 5.

6.2 Format 2

The nibble after the opcode is a register number **d**, and the next two nibbles are combined into an 8-bit memory address or literal value.

3 For example, you can write text to the TTY screen so the customary 'Hello World' program is now feasible!

The major change from Toy-A is that the opcode uses an extra bit (stolen from the **d** nibble) to provide more functions. If this bit is set, the function uses an 'indexed address' variation where the last 2 address nibbles now mean the address formed by adding registers **s** and **t**.

The sacrifice is that only 3 bits are left for register number **d**, restricting its range to 0 to 7, as opposed to 0 to 15. The other nibbles are unaffected so they can still refer to all 16 registers.[4]

The 'code' column in Instruction table - Toy-B (opcode order) expands the first 2 nibbles into binary for clarity.

Example 1: $5078 decodes as opcode 5, 5th bit unset, memory address $78.
This translates to **jump** where pc ← addr, so the program jumps to address $78.

Example 2: $5878 decodes as opcode 5, 5th bit set, source regs 7 and 8.
This translates to **jump indexed** where pc ← R[s]+R[t], so the program jumps to the address formed by adding the contents of reg 7 to the contents of reg 8.

The above examples do not use register number **d** so there are no restrictions, but the following ones do, so reg d can only refer to registers 0 to 7.

Example 3: $92B8 decodes as opcode 9, 5th bit unset (first bit from '2' nibble **0**010), dest reg 2 (last 3 bits from '2' nibble 0**010**), memory address $B8.
This translates to **load address** where R[d] ← addr, so value $B8 is written to reg 2.

Example 4: $9EB8 decodes as opcode 9, 5th bit set (first bit from 'E' nibble **1**110), dest reg 6 (last 3 bits from 'D' nibble 1**110**), source regs B and 8.
This translates to **load address indexed** where R[d] ← mem[R[s]+R[t]], so the value from memory at the specified address is written to reg 6. The specified address is the result of adding the contents of reg B to the contents of reg 8.

More details and other encodings are shown in the Instruction table - Toy-B (opcode order).

7. Operational notes

PlasMa's microcode 2 should be backwards compatible with existing programs for this Toy variant, although there is not as much published documentation as Toy-A. If your program does not run, it may be due to my incorrect assumptions or enhancements, in which case please let me know. The following notes and tables state PlasMa's interpretation.

Processing is stopped on illegal instructions. The illegal light will turn on.

All memory access instructions incur a timing overhead to demonstrate the benefits of work registers. The system timer I/O instructions can be used for timing specific sections of code; details of the system timer are in the Machine Manual.

In shift instructions, the number of places to be shifted is modulo 16 and always treated as a positive value.

The 'right shift' instruction maintains the sign of the result as it is shifted, but the 'left shift' instruction does not.

7.1 Sounds

PlasMa can generate two type of notes (tones): audio or MIDI. Toy-B supports audio notes only.

Audio notes are sent to the built-in speaker if it is in Notes mode, otherwise you will just hear the normal clicking sound from jump instructions (see the Machine Manual). Sounds can only be heard on the real machine; they are muted on the PlasMaSim simulator although the note-on period is always shown on the SND light.

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

4 It is unclear if this Princeton variant only had an 8-register architecture, but either way, PlasMa remains a 16-register machine, subject to the restrictions in format 2. This should still be backwards compatible.

7.1.1 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
```

c is only applicable to MIDI notes so is reserved in Toy-B.

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.

7.1.2 Note Definition

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

```
tnnn nnnn vvvv iiii
```

t is the type of note: 0 = audio, 1 = MIDI. It is ignored in Toy-B as audio notes are assumed.

n is a 7-bit note number interpreted as a standard MIDI note, e.g. 0=C-1, 60($3C)=C4 (middle C), 127=G9.

For audio notes, firmware restrictions restrict the lowest note to B2, 47($2F). Notes lower than this are treated as B2.

i is a 4-bit interval/length interpreted as standard musical note times:-

$0 = 1/32 (demisemiquaver)	$7 = 1/4 dotted
$1 = 1/32 dotted	$8 = 1/2 (minim)
$2 = 1/16 (semiquaver)	$9 = 1/2 dotted
$3 = 1/16 dotted	$A = 1 (semibreve)
$4 = 1/8 (quaver)	$B = 1 dotted
$5 = 1/8 dotted	$C = 2 (breve)
$6 = 1/4 (crotchet)	$D = 2 dotted

The actual times used for the audible and muted parts of the note are derived from this interval code[5] along with the tempo and gate ratio values set by the last call to Set Note Attribs. The note is not initiated if **i** is out of range or if the calculated note length is shorter than the machine's clock resolution (currently 10ms).

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 still respond with BUSY during this time.

5 The PleX emulation includes a function to convert the number of 64th notes to this interval code.

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.

8. Instruction table - Toy-B (opcode order)

The instruction set is from Princeton's Toy design with permission; see Acknowledgements.

Bits and nibbles marked '-' are ignored.
Plasm assembler terminology:

- rd, rs, rt = registers, e.g. r5 rb
- aa = address absolute, e.g. 200 $34

Code	Description	Plasm syntax
`0---`	**`halt`**`: stop processing` The halt light will turn on.	`hlt`
`1dst`	**`add`**`: R[d] ← R[s] + R[t]` Add the contents of reg s to the contents of reg t and store the result in reg d.	`add rd rs rt`
`2dst`	**`subtract`**`: R[d] ← R[s] - R[t]` Subtract the contents of reg t from the contents of reg s and store the result in reg d.	`sub rd rs rt`
`3dst`	**`multiply`**`: R[d] ← R[s] * R[t]` Multiply the contents of reg s with the contents of reg t and store the result in reg d.	`mul rd rs rt`
`4dst`	**`system call`**`: d, s and t define the system function; see I/O function table - Toy-B`	`sys fcn`

`50aa`
`0101 0--- a a`
	`jump``: pc ← addr`
	Jump to the specified memory address.
		`jmp aa`

`58st`
`0101 1--- s t`
	`jump indexed``: pc ← R[s]+R[t]`
	Jump to the address formed by adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`jmpi rs rt`

`6daa`
`0110 0ddd a a`
	`jump if greater``:`
	`if (R[d] > 0) pc ← addr`
	Jump to the specified address if the contents of reg d are positive, i.e. if the signed value is +1 or greater.
	The reg number d is limited to 0..7.
	There is no explicit 'jz' instruction in Toy-B, but two 'jp's and an 'add' can be used to emulate it, e.g.
	`jp reg fail  ;fail, number is +1 or greater`
	`add 1 to reg ;if number was zero, it's now 1`
	`jp reg ok    ;ok, number was zero`
	`jmp fail     ;fail, number was negative`
		`jp rd aa`

`6Dst`
`0110 1ddd s t`
	`jump if greater, indexed``:`
	`if (R[d] > 0) pc ← R[s]+R[t]`
	Jump to the specified address if the contents of reg d are positive, i.e. if the signed value is +1 or greater.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`jpi rd rs rt`

Code	**Description**	**Plasm syntax**

`7daa`
`0111 0ddd a a`
	`jump & count (dec and jump if not zero`[6]`)``:`
	`R[d] = R[d] - 1; if (R[d] <> 0)  pc ← addr`
	Decrement the contents of reg d by one. If the result is not zero, jump to the specified address.
	The reg number d is limited to 0..7.
		`djnz rd aa`

`7Dst`
`0111 1ddd s t`
	`jump & count (dec & jump if not zero), indexed``:`
	`R[d] = R[d] - 1; if (R[d] <> 0) pc ← R[s]+R[t]`
	Decrement the contents of reg d by one. If the result is not zero, jump to the specified address.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`djnzi rd rs rt`

`8daa`
`1000 0ddd a a`
	`jump and link``: R[d] ← pc + 1; pc ← addr`
	Save the address of the next instruction in reg d, then jump to the specified address. The 'jump indexed' instruction can be used to return to that 'next' instruction.
	The reg number d is limited to 0..7.
		`jlk rd aa`

`8Daa`
`1000 1ddd s t`
	`jump and link, indexed``:`
	`R[d] ← pc + 1; pc ← R[s]+R[t]`
	Save the address of the next instruction in reg d, then jump to the specified address. The 'jump indexed' instruction can be used to return to that 'next' instruction.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`jlki rd rs rt`

`9daa`
`1001 0ddd a a`
	`load``: R[d] ← mem[addr]`
	Load a 16-bit value from memory at the specified address into reg d.
	The reg number d is limited to 0..7.
		`ld rd aa`

`9Dst`
`1001 1ddd s t`
	`load indexed``: R[d] ← mem[R[s]+R[t]]`
	Load a 16-bit value from memory at the specified address into reg d.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`ldi rd rs rt`

`Adaa`
`1010 0ddd a a`
	`store``: mem[addr] ← R[d]`
	Store the contents of reg d to memory at the specified address.
	The reg number d is limited to 0..7.
		`st aa rd`

6 PlasMa uses this alternative name to clarify how the instruction works, e.g. the decrement occurs *before* the jump.

Code	Description	Plasm syntax

`ADst`
`1010 1ddd s t`
	`store indexed``: mem[R[s]+R[t]] ← R[d]`
	Store the contents of reg d to memory at the specified address.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`sti rs rt rd`

`Bdaa`
`1011 0ddd a a`
	`load address``: R[d] ← addr`
	Load the specified 8-bit address into reg d. The ms 8 bits of reg d are cleared to zero.
	The instruction can be used to load other values which are not addresses, but they are still restricted to 8-bit quantities; you cannot load a full 16 bits into a register this way. To load such values, use the "load' or 'load indexed' instructions.
	The reg number d is limited to 0..7.
		`lda rd aa`

`BDst`
`1011 1ddd s t`
	`load address, indexed``: R[d] ← R[s]+R[t]`
	Load an 8-bit address into reg d.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
	The instruction can be used to load other values which are not addresses, but they are still restricted to 8-bit quantities; you cannot load a full 16 bits into a register this way. To load such values, use the "load' or 'load indexed' instructions.
		`ldai rd rs rt`

`Cdst`	**`xor`**`: R[d] ← R[s] ^ R[t]` 'Exclusive or' the contents of reg s `with` the contents of reg t and store the result in reg d.	`xor rd rs rt`

`Ddst`
 	
`Ddss`
 	
`D000`
	`and``: R[d] ← R[s] & R[t]`
	'And' the contents of reg s `with` the contents of reg t and store the result in reg d.
	If **s** and **t** are the same, this behaves like a 'copy reg' instruction as something 'anded' with itself is unchanged, so Plasm offers a 'cp' mnemonic for convenience.
	If **d**, **s** and **t** are all the same, this behaves like a 'no-operation', so Plasm offers a 'nop' mnemonic for convenience.
		`and rd rs rt`
		
		`cp rd rs`
		
		`nop`

`Edaa`
`1110 0ddd a a`
	`shift right``: R[d] ← R[d] >> addr`
	Shift the contents of reg d to the `right` by the specified number of bits. The sign bit is propagated.
	The reg number d is limited to 0..7.
		`shr rd aa`

`EDst`
`1110 1ddd s t`
	`shift right, indexed``: R[d] ← R[d] >> R[s]+R[t]`
	Shift the contents of reg d to the `right` by the specified number of bits.
	The reg number d is limited to 0..7, and the number of bits is the result of adding the contents of reg s to the contents of reg t. The result is truncated to $FF. The sign bit is propagated.
		`shri rd rs rt`

Code	Description	Plasm syntax

`Fdaa`
`1111 0ddd a a`
	`shift left``: R[d] ← R[d] << addr`
	Shift the contents of reg d to the `left` by the specified number of bits.
	The reg number d is limited to 0..7.
		`shl rd aa`

`FDst`
`1111 1ddd s t`
	`shift left, indexed``: R[d] ← R[d] << R[s]+R[t]`
	Shift the contents of reg d to the `left` by the specified number of bits.
	The reg number d is limited to 0..7, and the number of bits is the result of adding the contents of reg s to the contents of reg t. The result is truncated to $FF.
		`shli rd rs rt`

9. Instruction table - Toy-B (functional order)

The instruction set is from Princeton's Toy design with permission; see Acknowledgements.

Bits and nibbles marked '-' are ignored.
Plasm assembler terminology:

- rd, rs, rt = registers, e.g. r5 rb
- aa = address absolute, e.g. 200 $34

9.1 Arithmetic

Code	Description	Plasm syntax
`1dst`	**add**: `R[d] ← R[s] + R[t]` Add the contents of reg s to the contents of reg t and store the result in reg d.	`add rd rs rt`
`2dst`	**subtract**: `R[d] ← R[s] - R[t]` Subtract the contents of reg t from the contents of reg s and store the result in reg d.	`sub rd rs rt`
`3dst`	**multiply**: `R[d] ← R[s] * R[t]` Multiply the contents of reg s with the contents of reg t and store the result in reg d.	`mul rd rs rt`

9.2 Input/Output

Code	Description	Plasm syntax
`4dst`	**system call**: `d, s and t define the system function; see I/O function table - Toy-B`	`sys fcn`

9.3 Jump

Code	Description	Plasm syntax

`50aa`
`0101 0--- a a`
	jump: `pc ← addr`
	Jump to the specified memory address.
		`jmp aa`

`58st`
`0101 1--- s t`
	jump indexed: `pc ← R[s]+R[t]`
	Jump to the address formed by adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`jmpi rs rt`

`6daa`
`0110 0ddd a a`
	jump if greater:
	`if (R[d] > 0) pc ← addr`
	Jump to the specified address if the contents of reg d are positive, i.e. if the signed value is +1 or greater.
	The reg number d is limited to 0..7.
	There is no explicit 'jz' instruction in Toy-B, but two 'jp's and an 'add' can be used to emulate it, e.g.
	`jp reg fail  ;fail, number is +1 or greater`
	`add 1 to reg ;if number was zero, it's now 1`
	`jp reg ok    ;ok, number was zero`
	`jmp fail     ;fail, number was negative`
		`jp rd aa`

Code	Description	Plasm syntax

`6Dst`
`0110 1ddd s t`
	`jump if greater, indexed``:`
	`if (R[d] > 0) pc ← R[s]+R[t]`
	Jump to the specified address if the contents of reg d are positive, i.e. if the signed value is +1 or greater.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`jpi rd rs rt`

`7daa`
`0111 0ddd a a`
	`jump & count (dec and jump if not zero`[7]**`)`**`:`
	`R[d] = R[d] - 1; if (R[d] <> 0)  pc ← addr`
	Decrement the contents of reg d by one. If the result is not zero, jump to the specified address.
	The reg number d is limited to 0..7.
		`djnz rd aa`

`7Dst`
`0111 1ddd s t`
	`jump & count (dec & jump if not zero), indexed``:`
	`R[d] = R[d] - 1; if (R[d] <> 0) pc ← R[s]+R[t]`
	Decrement the contents of reg d by one. If the result is not zero, jump to the specified address.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`djnzi rd rs rt`

`8daa`
`1000 0ddd a a`
	`jump and link``: R[d] ← pc + 1; pc ← addr`
	Save the address of the next instruction in reg d, then jump to the specified address. The 'jump indexed' instruction can be used to return to that 'next' instruction.
	The reg number d is limited to 0..7.
		`jlk rd aa`

`8Daa`
`1000 1ddd s t`
	`jump and link, indexed``:`
	`R[d] ← pc + 1; pc ← R[s]+R[t]`
	Save the address of the next instruction in reg d, then jump to the specified address. The 'jump indexed' instruction can be used to return to that 'next' instruction.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`jlki rd rs rt`

9.4 Load/Store

Code	Description	Plasm syntax

`9daa`
`1001 0ddd a a`
	`load``: R[d] ← mem[addr]`
	Load a 16-bit value from memory at the specified address into reg d.
	The reg number d is limited to 0..7.
		`ld rd aa`

7 PlasMa uses this alternative name to clarify how the instruction works, e.g. the decrement occurs *before* the jump.

Code	Description	Plasm syntax

`9Dst`
`1001 1ddd s t`
	`load indexed``: R[d] ← mem[R[s]+R[t]]`
	Load a 16-bit value from memory at the specified address into reg d.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`ldi rd rs rt`

`Adaa`
`1010 0ddd a a`
	`store``: mem[addr] ← R[d]`
	Store the contents of reg d to memory at the specified address.
	The reg number d is limited to 0..7.
		`st aa rd`

`ADst`
`1010 1ddd s t`
	`store indexed``: mem[R[s]+R[t]] ← R[d]`
	Store the contents of reg d to memory at the specified address.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
		`sti rs rt rd`

`Bdaa`
`1011 0ddd a a`
	`load address``: R[d] ← addr`
	Load the specified 8-bit address into reg d. The ms 8 bits of reg d are cleared to zero.
	The instruction can be used to load other values which are not addresses, but they are still restricted to 8-bit quantities; you cannot load a full 16 bits into a register this way. To load such values, use the "load' or 'load indexed' instructions.
	The reg number d is limited to 0..7.
		`lda rd aa`

`BDst`
`1011 1ddd s t`
	`load address, indexed``: R[d] ← R[s]+R[t]`
	Load an 8-bit address into reg d.
	The reg number d is limited to 0..7, and the address is the result of adding the contents of reg s to the contents of reg t. The address is truncated to $FF.
	The instruction can be used to load other values which are not addresses, but they are still restricted to 8-bit quantities; you cannot load a full 16 bits into a register this way. To load such values, use the "load' or 'load indexed' instructions.
		`ldai rd rs rt`

9.5 Logical

Code	Description	Plasm syntax

`Ddst`
 	
`Ddss`
 	
`D000`
	`and``: R[d] ← R[s] & R[t]`
	'And' the contents of reg s `with` the contents of reg t and store the result in reg d.
	If **s** and **t** are the same, this behaves like a 'copy reg' instruction as something 'anded' with itself is unchanged, so Plasm offers a 'cp' mnemonic for convenience.
	If **d**, **s** and **t** are all the same, this behaves like a 'no-operation', so Plasm offers a 'nop' mnemonic for convenience.
		`and rd rs rt`
		
		`cp rd rs`
		
		`nop`

Code	Description	Plasm syntax
`Cdst`	**`xor`**`: R[d] ← R[s] ^ R[t]` 'Exclusive or' the contents of reg s `with` the contents of reg t and store the result in reg d.	`xor rd rs rt`

9.6 Misc

Code	Description	Plasm syntax
`0---`	**`halt`**`: stop processing` The halt light will turn on.	`hlt`

9.7 Shift

Code	Description	Plasm syntax

`Edaa`
`1110 0ddd a a`
	`shift right``: R[d] ← R[d] >> addr`
	Shift the contents of reg d to the `right` by the specified number of bits. The sign bit is propagated.
	The reg number d is limited to 0..7.
		`shr rd aa`

`EDst`
`1110 1ddd s t`
	`shift right, indexed``: R[d] ← R[d] >> R[s]+R[t]`
	Shift the contents of reg d to the `right` by the specified number of bits.
	The reg number d is limited to 0..7, and the number of bits is the result of adding the contents of reg s to the contents of reg t. The result is truncated to $FF. The sign bit is propagated.
		`shri rd rs rt`

`Fdaa`
`1111 0ddd a a`
	`shift left``: R[d] ← R[d] << addr`
	Shift the contents of reg d to the `left` by the specified number of bits.
	The reg number d is limited to 0..7.
		`shl rd aa`

`FDst`
`1111 1ddd s t`
	`shift left, indexed``: R[d] ← R[d] << R[s]+R[t]`
	Shift the contents of reg d to the `left` by the specified number of bits.
	The reg number d is limited to 0..7, and the number of bits is the result of adding the contents of reg s to the contents of reg t. The result is truncated to $FF.
		`shli rd rs rt`

10. I/O function table - Toy-B

The first two functions are from Princeton's Toy design with permission; see Acknowledgements; all others are PlasMa designs.

The 4 nibbles in an I/O system call instruction are of the form: **4dst**
where **t** is the function[8], and **s** and **d** are function qualifiers.

d	**s**	**t (hex)**	**description**
dest reg	0	1	**TTY Read** R[d] ← TTY keypad Halt the program and read hex digits from the hex keypad.
srce reg	0	2	**TTY Write** TTY display ← R[d] Write the 16-bit value to a new line on the TTY display in hex and decimal format. If the display is empty, lines start at the top, otherwise they continue on the next line, wrapping round to the top when the last line is reached. The line following the written line is cleared to show where the next write will occur. The function does not return until the display has been updated; this may impact your program's performance[9].
srce reg	0	3	**TTY Write Char** TTY display ← R[d] Write/append the least significant byte in R[d] as an ASCII character to the current line of the TTY screen. When the last character in the line is written, the following line is cleared to show where the next write will occur. Lines wrap round to the top when the bottom line is full. The value $00[10] forces a new line and clears the following line as above. The function does not return until the display has been updated; this may impact your program's performance. See also TTY Write Char Literal.
0	1	3	**TTY Clear** Clear the display screen. The function does not return until the display has been updated; this may impact your program's performance.

8 The function **t** is in the ls nibble for compatibility with the original Toy tty functions; the table columns are shown in reverse order for ease of manual assembly, as this is the order they appear in the instruction.
9 The PleX oper functions are asynchronous so your program can do other things while the display is updating.
10 This value is deliberately chosen instead of the typical value $0A to simplify Toy programming; if you already have a register dedicated for the constant 0, only one I/O instruction is needed.

d	s	t (hex)	description
dest reg	2	3	**KPad Read** R[d] ← keypad key number. The key number is $00 to $0F depending on which of the 16 hex buttons are pressed at the time of the call. If no key is pressed, the key number is $FFFF. This function does not stop the machine, unlike TTY Read.
0	3	3	**Stop System Timer** Stop the timer used by the panel display.
0	4	3	**Reset/Start System Timer** Reset the panel timer to zero and start it running. If it is already running, reset to zero and continue running. The panel display updates every second when the timer is running.
srce & dest reg R[d]=delay number	5	3	**Poll Timer Delay** R[d] ← delay status Store the required delay number (0-3) in the specified register *before* calling this function. The function is ignored if the delay number is outside this range. R[d] ← 1 if the specified delay has not elapsed. R[d] ← 0 if the specified delay has elapsed, in which case the delay is re-primed automatically. Re-priming only occurs at the time of the poll, so for accurate delays, call the poll function as frequently as possible. Alternatively, construct your own delays using the Read System Timer function. Use the Set Timer Long Delay and Set Timer Short Delay functions to initialise and prime the delays. The delay timers are independent of the system timer used for the panel display.
srce & dest reg R[d]=limit	6	3	**Read Random** R[d] ← 16-bit random value. The random number is in the range 0 to N inclusive, where N is the value in the specified register *before* calling this function. e.g. if N=7, the random number will be between 0 and 7 inclusive. N=0 has the same effect as N=$FFFF and gives a random number 0..$FFFF inclusive.

d	s	t (hex)	description
dest reg	7	3	**Read Platform Info** R[d] ← Platform ident and version number. Format: 2-bit platform ident, 14-bit version number. Platform ident: 0=Plasm machine 1=PlasMaSim simulator 2=Scamp simulator[11] 3=TBA 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
srce reg	8	3	**Play Note** Audio note ← R[d] Play an audio note according to the value in R[d] using attributes defined by the last call to Set Note Attribs. See Note Definition for details. The note is not played if the previous note is still playing, i.e. its interval has not elapsed at the time of the call. The function returns immediately. Call the Get Note Status function to check if the previous interval has elapsed, and is therefore able to accept another Play Note function. The speaker must be in Notes mode to hear any notes, otherwise you will just hear clicks from jump instructions; see the Machine Manual. Sounds can only be heard on the real machine; they are muted on the PlasMaSim simulator.
srce & dest reg	9	3	**Convert Mins to BCDHrsMins** R[d] BCD hrs mins ← R[d] mins Convert the binary number of minutes in R[d] to a binary-coded decimal (BCD) value of hours & mins. e.g. if R[d]=62, R[d] ← $0102 (62 mins = 1 hour 2 mins). If the binary number exceeds 5999 (BCD $9959), the function does nothing. You can use it to generate BCD from the Read System Timer *long* time value. Use the Convert N to BCD function to generate BCD from the *short* time value.

11 Scamp is a Toy instruction set simulator by Adrian Rawson at ahrprojects.co.uk.

d	**s**	**t (hex)**	**description**
srce & dest reg	A	3	**Convert BCDHrsMins to Mins** R[d] mins ← R[d] BCD hrs mins Convert the BCD hours & mins value in R[d] to minutes. e.g. if R[d]=$0215 (2 hours 15 mins), R[d] ← 135. If any BCD nibbles exceed 9, or if the ls two nibbles exceed $59, the function does nothing. The function is general-purpose so any BCD value up to $9959 can be converted. You can use it to generate a *long* time value for the Set Timer Long Delay function, but out-of-range values will be ignored by those functions. Use the Convert BCD to N function to generate a *short* time value.
srce & dest reg	B	3	**Convert N to BCD** R[d] BCD ← R[d] binary Convert the binary number in R[d] to a binary-coded decimal (BCD) value. e.g. if R[d]=345, R[d] ← $0345. If the binary number exceeds 9999 (BCD $9999), the function does nothing. You can use it to generate BCD from the Read System Timer *short* time value. Use the Convert Mins to BCDHrsMins function to generate BCD from the *long* time value.
srce & dest reg	C	3	**Convert BCD to N** R[d] binary ← R[d] BCD Convert the BCD value in R[d] to a binary number. e.g. if R[d]=$0345, R[d] ← 345. If any BCD nibbles exceed 9, the function does nothing. The function is general-purpose so any BCD value up to $9999 can be converted. You can use it to generate a *short* time value for the Set Timer Short Delay and Set Timer Long Delay functions, but out-of-range values will be ignored by those functions. Use the Convert BCDHrsMins to Mins function to generate a *long* time value.
srce reg	D	3	**Set Note Attribs** Attribs ← R[d] Define attributes 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. See Note Attribute for details.

d	s	t (hex)	description
dest reg	E	3	**Get Note Status** R[d] ← status If the interval defined by the previous Play Note has elapsed, R[d] is set to 0, otherwise it is set to 1.
dest reg	F	3	**Get TTY Size** R[d] ← TTY size The ms byte is the width in chars; the ls byte is the height in lines.
dest reg d	dest reg s	4	**Read System Timer** R[d] ← Hours and minutes in minutes. R[s] ← Seconds in 1/100ths of a second. Return the elapsed time since the system timer was started. e.g. a time of 1 hour, 2 mins and 3.45 seconds will give: R[d] ← 62, R[s] ← 345. The maximum value for R[d] is 1439 (23 hours, 59 mins). The maximum value for R[s] 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 displaying.

d	**s**	**t (hex)**	**description**
srce reg d where d & s reg numbers are **different**	srce reg s	5	**Set Timer Long Delay** Time delay ← R[d], R[s] Set a long time delay from 0.01 seconds to 23hrs 59mins 59.99 secs as specified in R[d] and R[s]: Hours & minutes in minutes ←R[d] 2-bit delay no, 14-bit seconds in 1/100ths of a second ← R[s] This function is used in conjunction with Poll Timer 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 R[d] to 62 and R[s] to 345. To set delay 1 to the same value, 'or' R[s] with $4000. To set delay 2 to the same value, 'or' R[s] with $8000. To set delay 3 to the same value, 'or' R[s] with $C000. The function is ignored if R[d] exceeds 1439 (23 hrs, 59 mins), or if R[s] exceeds 5999 (59.99 secs). The delay timer is (re)primed whenever Set Timer 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 System Timer function directly. The Set Timer 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.
srce reg d where d & s reg numbers are **equal**	srce reg s	5	**Set Timer Short Delay** Time delay ← R[s] The **s** and **d** register numbers must be the same, e.g. $4**77**5. Set a short time delay from 0.01 secs to 59.99 secs as specified in R[s]. 2-bit delay no, 14-bit seconds in 1/100ths of a second ← R[s] This is similar to the Set Timer Long Delay function, but only needs a single register to define the delay. See Set Timer Long Delay for more details.
dest reg	0 = lod-ms 1 = lod-ls 2 = brk-ms 3 = brk-ls 4 = mem	6	**Read Switches** R[d] ← switch bank s Read the on/off states of the latching switches defined by the value in **s**. If **s**=4, the value of all 16 switches is returned regardless of the memory size in the current emulation.

d	**s**	**t (hex)**	**description**
srce reg	MT0 0 = lt inner 1 = lt outer 2 = rt inner 3 = rt outer 4 = status MT1 5 = lt inner 6 = lt outer 7 = rt inner 8 = rt outer 9 = status	7	**Write MT Lights** Left/right inner/outer rings ← R[d] The mag tape decks are not available in Toy-B, so the 'spool' lights are free to use how you like. Each spool has an inner and outer ring of 15 lights. They are numbered clockwise from 0 to 14, starting with 0 at the 12 o'clock position. The status lights refer to the 4 lights above each pair of spools. These are numbered from right to left as 0 to 3. If the value in **s** indicates a ring, the corresponding bits in R[d] are written to it. Bit 0 (the ls bit) is written to ring light 0, bit 1 to ring light 1, and so on. The ms bit of R[d] (bit 15) is ignored. If the value in **s** indicates a status, the corresponding bits in R[d] are written to it. Bit 0 (the ls bit) is written to status light 0, bit 1 to status light 1, and so on. Bits 4 to 15 in R[d] are ignored.
ms char	ls char	8	**TTY Write Char Literal** tty display ← ds This is the same as TTY Write Char except that the byte is the 8-bit value in the **d** and **s** nibbles of the instruction instead of the contents of rv[12].
dest reg	0	9	**PTR Get Status** R[d] ← status Read the current status value into R[d]. 0=OFF 1=RDY 2=BUSY 3=EOT (end of tape) 4=DATA_RDY 5=ERR_DATA (data value unexpected) 5=ERR_READ (media) 6=ERR_PARITY (only used by the 7-bit+parity tape reader)

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

d	s	t (hex)	description
0	1	9	**PTR Request Read** 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 Data 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.
dest reg	2	9	**PTR Get Data** R[d] ← data If the reader status is DATA_RDY, transfer the data value from the device's read buffer to R[d] and set the status to RDY, otherwise do nothing. The data corresponds to the most recent PTR Request Read call. In all cases, the function returns immediately.
dest reg	0	A	**PTP Get Status** R[d] ← status Read the current status of the paper-tape punch into R[d]. 0=OFF 1=RDY 2=BUSY 3=EOT (end of tape) 4=ERR_WRITE (media)

d	s	t (hex)	description
srce reg	1	A	**PTP Request Write** data ← R[d] If the punch status is RDY, write the ls-byte in R[d] 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.
dest reg	0	B	**LP Get Status** R[d] ← status Read the current printer status value into R[d]. If the printer is ready to accept commands, set R[d]=0 (RDY), otherwise set R[d]=1 (BUSY)[13].
srce reg	1	B	**LP Write Data** Printer ← R[d] If the printer status is RDY, print the 16-bit value in hex and decimal format followed by a newline, otherwise the function does nothing. The function does not return until the whole line has been sent to the printer; this may impact your program's performance. If the printer becomes busy during the transfer, the program will retry for a brief period. If the printer remains busy, the function will timeout and the remaining data discarded.

13 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	**s**	**t (hex)**	**description**
srce reg	2	B	**LP Write Char** Printer ← R[d] If the printer status is RDY, send the least significant byte in R[d] as an ASCII character to the printer, otherwise the function does nothing. The value $00 is interpreted as a new line (linefeed); see footnote 10. You can also use explicit linefeed, carriage-return or other printer controls as required. See also LP Write Char Literal.
ms char	ls char	C	**LP Write Char Literal** Printer ← ds This is the same as LP Write Char except that the byte is the 8-bit value in the **d** and **s** nibbles of the instruction instead of the contents of R[d].