PlasMa

aka Phil's lights and switches Machine

Mini-Mainframe

Simulator Project

Instruction Set Manual

Toy-A

Phil Tipping

www.philizound.co.uk

Table of Contents

1. Introduction

This describes the Toy-A 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 is a variant of the ‘Toy’ computer used as a teaching aid at Princeton university, USA. This has 16 functions and 16 registers, all represented as 4-bit nibbles within a 16-bit instruction, making it very easy to hand-assemble.

The university uses variants of the 16 functions to demonstrate different programming techniques, and PlasMa supports two of these, which I've named Toy-A and Toy-B. Toy-A is the most common variant, and was used in a hardware unit by Penn Engineering[1].

Princeton’s terminology is used where possible so examples and tutorials on the internet can be followed. Existing programs should run with little or no change; see Operational notes for exceptions.

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, USA, and is described in their book 'Computer Science'. More details at:

https://introcs.cs.princeton.edu/java/home and https://introcs.cs.princeton.edu/java/60machine

Coursera course site: https://www.coursera.org/learn/cs-algorithms-theory-machines

Thanks also to Adrian Rawson for suggestions and support.

3. Documentation

These following documents are downloadable from the philizound.co.uk website or by contacting me directly (email address is at the bottom of the website page).

- PlasMa Machine Manual
- PlasMa Instruction Set - Toy-A
- PlasMa Instruction Set - Toy-B
- PlasMa Instruction Set - Advanced
- PlasMaSim Simulator Manual
- Plasm Assembler Manual

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 (r0 is read-only and contains zero).

5. I/O overview

Basic I/O is achieved by accessing the highest memory location $FF.

Writing the contents of a register to this address (using 'store' or 'store indirect') displays the value in hex and decimal on the built-in display screen.

1 Albeit with far fewer lights and switches!

Reading from this address into a register (using 'load' or 'load indirect') halts the program and allows user input from the built-in hex keypad.

6. Instructions overview

Instructions are encoded such that one nibble represents the opcode, giving 16 possible functions, and the other nibbles represent either a register (there are 16 registers) or an 8-bit value which can be a memory address or literal number.

The left-hand (ms) nibble is the opcode for all instructions, and the others are encoded in (mostly) two ways, called format 1 and format 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.

Example 1: $72F5 decodes as opcode 7, dest reg 2, literal/address value $F5.
This translates to **load address** where R[d] ← addr, so the value $F5 is written to reg 2.

Example 2: $C587 decodes as opcode C, reg 5, memory address $87.
This translates to **branch zero** where if (R[d] == 0) pc ← addr, so the program jumps to address $87 if the contents of reg 5 is zero.

Example 3: $E400 decodes as opcode E, reg 4 (the last two nibbles are unused).
This translates to **jump register** where pc ← R[d], so the program jumps to the address held in reg 4.

More details and other encodings are shown in the Instruction table - Toy-A.

7. Operational notes

PlasMa's microcode 1 should be backwards compatible with existing programs for this Toy variant. If your program does not run, it may be due to my incorrect assumptions, 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.

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.

8. Instruction table - Toy-A

Plasm assembler terminology:

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

Code	**Description** (nibbles marked '-' are ignored)	**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`
`1ds0`	If **s** or **t** are zero, this behaves like a 'copy reg' instruction as reg 0 is always zero; Plasm offers a 'cp' mnemonic for convenience.	`cp rd rs`
`1000`	If **d** is zero, this behaves like a 'no-operation' instruction as reg 0 is read-only; Plasm offers a 'nop' mnemonic for convenience.	`nop`
`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`	`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.	`and rd rs rt`
`4dst`	`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`
`5dst`	`left shift: R[d] ← R[s] << R[t]` Shift the contents of reg s to the left by the number held in reg t, and store the result in reg d.	`shl rd rs rt`
`6dst`	`right shift: R[d] ← R[s] >> R[t] signed` Shift the contents of reg s to the `right` by the number held in reg t, and store the result in reg d. The sign bit is propagated.	`shr rd rs rt`
`7daa`	`load address: R[d] ← addr` Load the specified 8-bit number into reg d. The ms 8 bits of reg d are cleared to zero. The number can be an address or a literal value, but it cannot be greater than 255. To load a larger number, use the 'load' instruction.	`lda rd aa`
`8daa`	`load: R[d] ← mem[addr]` Load the 16-bit contents of memory at the specified address into reg d.	`ld rd aa`
`9daa`	`store: mem[addr] ← R[d]` Store the contents of reg d to memory at the specified address.	`st aa rd`
`Ad-t`	`load indirect: R[d] ← mem[R[t]]` Load the 16-bit contents of memory at the address held in reg t into reg d.	`ldi rd rt`

Code	**Description** (nibbles marked '-' are ignored)	**Plasm syntax**
`Bd-t`	`store indirect: mem[R[t]] ← R[d]` Store the contents of reg d into memory at the address held in reg t.	`sti rt rd`
`Cdaa`	`branch zero: if (R[d] == 0) pc ← addr` Branch/jump to the specified address if the contents of reg d are zero.	`jz rd aa`
`C0aa`	If **d** is zero, this behaves like an unconditional jump as reg 0 is always zero; Plasm offers a 'jmp' mnemonic for convenience.	`jmp aa`
`Ddaa`	`branch positive: if (R[d] > 0) pc ← addr` Branch/jump to the specified address if the contents of reg d are positive, i.e. if the signed value is +1 or greater.	`jp rd aa`
`Ed--`	`jump register: pc ← R[d]` Branch/jump to the memory address held in reg d. This can be used to continue processing after a 'jump and link' instruction.	`jrg rd`
`Fdaa`	`jump and link: R[d] ← pc + 1`[2]`; pc ← addr` Save the address of the next instruction in reg d, then branch/jump to the specified address. The 'jump register' instruction can be used to return to that 'next' instruction.	`jlk rd aa`

2 There is no '+1' in the Princeton documents, but when the machine is stopped with PC/IR showing this instruction, the '+1' makes more sense (to me!). Either way, the functionality is the same; the address stored in R[d] is that of the *next* instruction.