User Manual

PLP Dev Team
ECEN
Oklahoma State University, OK

January 16, 2012

Contents

1 Introduction

This document serves as the hardware and software developers guide for the Progressive Learning Platform (PLP) System on a Chip. The PLP board is a unique learning platform designed to be simple, open, and of course, useful for education.

2 Getting PLP

Always make sure to download the latest version of PLP. This manual reflects the latest version.

PLP is available for download in the downloads section. The archive contains the following directory structure:

1. hw - Hardware images for the CPU
2. sw - Software tools, example programs, and software library
   (a) PLPTool - PLP Software tools, see next section
   (b) libplp - PLP software library
   (c) examples - Software examples

3 Software Tools (PLPTool)

PLPTool is a software suite for PLP that includes an assembler, a simulator, and a board programmer interface.

3.1 Running PLPTool

PLPTool requires a Java Runtime Environment that complies with at least Java 2 Platform SE 5 (1.5) specifications. PLPTool is shipped with RXTX library for serial communication in Windows. If you use Linux, you will have to install the library provided by the distribution that you use. Serial communication is used to download programs to the board.

PLPTool is located in `sw/PLPTool` in the PLP download archive. You need to uncompress this folder to run the software. If you use Windows, you need to run the batch file `PLPToolWin32.bat` if you use the 32-bit version of Windows, or `PLPToolWin64.bat` if you use the 64-bit version. PLPTool will give a warning if it can not detect the RXTX library, and the user will not be able to program the board or use the serial terminal.

3.1.1 Command Line Options

Launching PLPTool with no command line arguments will launch the GUI. Launching PLPTool with a `.plp` file as the only argument will launch the GUI and open that project.

```
java -jar PLPTool.jar <.plp file to open>
```

or in Windows, where XX is the CPU type (32- or 64-bit):

```
PLPToolWinXX.bat <.plp file to open>
```

Running PLPTool with the command line argument below will list the source files that are in the project. It will also display the index of said source files that is used to export, remove, or set a source file as top level source.

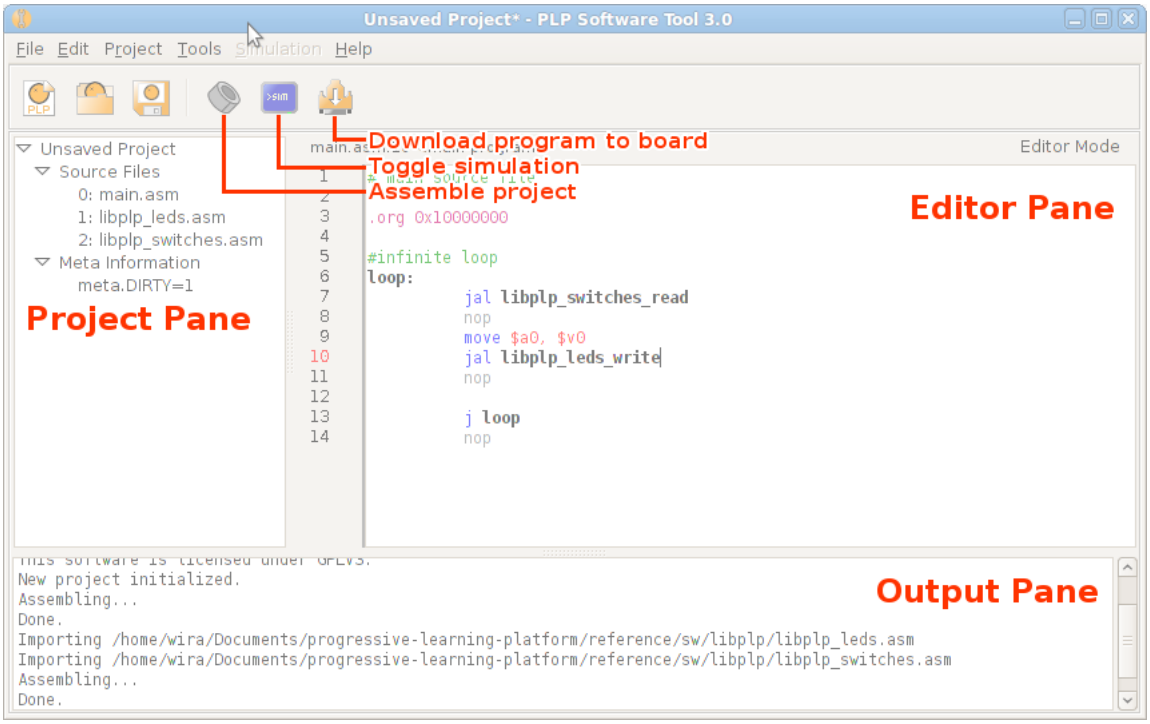
```
java -jar PLPTool.jar -plp <plpfile>
```

The `-plp <plpfile>` command can also take additional arguments that can be used to manipulate the project file without launching the GUI:

Command Line Option	Description
-c ¡asm 1¿ ¡asm 2¿ ...	Creates ¡plpfile¿ and imports ¡asm 1¿, ¡asm 2¿, ... to the project
-p ¡port¿	Programs ¡plpfile¿ to serial port ¡port¿
-a	Performs an assembly of the source files inside ¡plpfile¿
-i ¡asm 1¿ ¡asm 2¿ ...	Imports ¡asm 1¿, ¡asm 2¿, ... into ¡plpfile¿ project file
-d ¡directory¿	Imports all files in ¡directory¿ to the ¡plpfile¿ project file
-e ¡index¿ ¡file¿	Exports the source file with the index ¡index¿ as ¡file¿
-r ¡index¿	Removes the source file with the index ¡index¿
-s ¡index¿	Set the source file with the index ¡index¿ as the main program
-m ¡index¿ ¡new index¿	Set ¡new index¿ for the source file with the index ¡index¿

3.2 Graphical User Interface

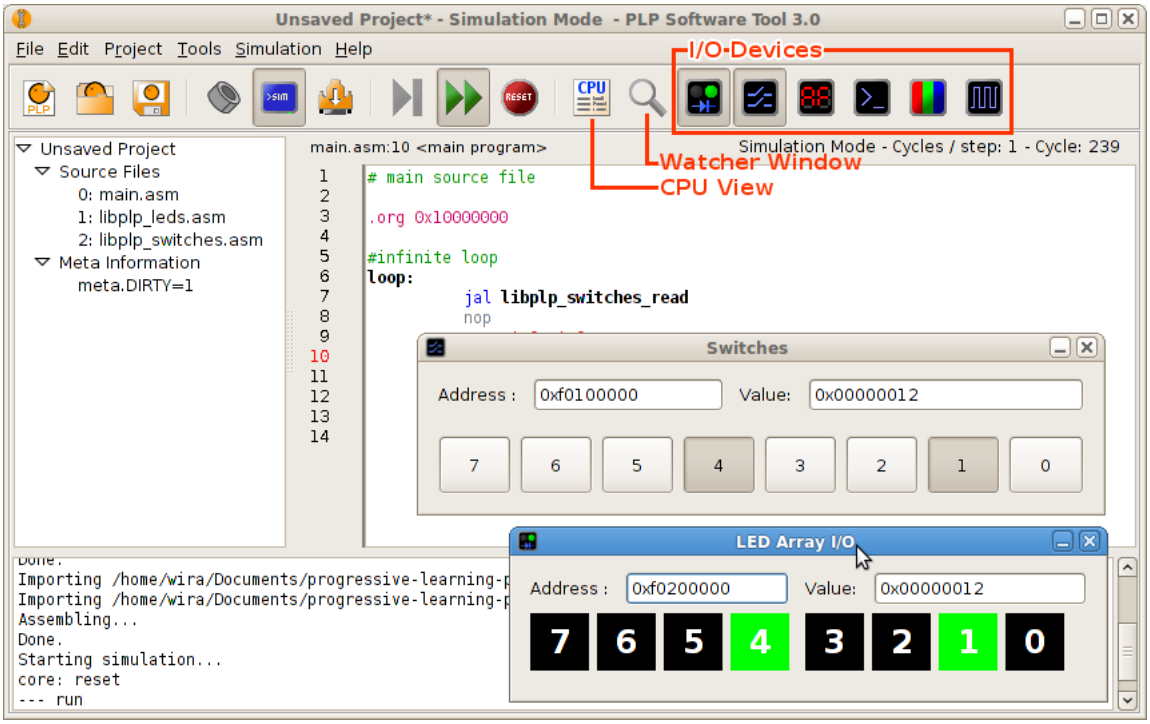
PLPTool starts in the development environment view, displaying the current open file, files in the project, and a status window. From this view, you can import or create new assembly files, assemble the project, launch the simulator, and program the PLP board.



The **Project Pane** contains all the source files in the project. The **Editor Pane** displays the contents of the currently open source file. The **Output Pane** displays informative, warning, and error messages.

3.3 Cycle Accurate Simulator

PLPTool includes a cycle-accurate simulator of the system. This simulator can be accessed from the command-line, or through the graphical user interface.



The simulation mode adds additional controls to the main window. The first three are the step, run, and reset buttons. Step (F5) will advance the simulation by one cycle, the run toggle button (F7) will continuously run the simulation, and the reset button (F9) will return the CPU to the reset state. The CPU view button will display the CPU window where users can view and modify register file contents, see the disassembly listing, and access the debug console for advanced interaction with the simulation.

3.3.1 Setting a Breakpoint

Breakpoint can be set by double-clicking the line number column. This can only applies in lines where there is actually an instruction present. Double-click on an existing breakpoint to clear it, or use `Tools -> Clear Breakpoints` to clear ALL breakpoints.

3.3.2 Command-Line Mode

To run the command-line simulator:

```
java -jar PLPTool.jar -s <plp file to simulate>
```

3.4 Programmer

You can program the PLP board by selecting `Project->Program PLP Board` in the development view. The programmer dialog allows you to select the communications port.

Additionally, you can program the PLP board from the command line with:

```
java -jar PLPTool.jar -plp <plp project> -p <communications port>
```

4 Instruction Set and Assembly Language

This section describes all supported instructions and pseudo-instructions by the PLP system. It also gives examples on how to use each instruction in a program and notes on limitations.

4.1 Syntax

4.1.1 Instructions

Instructions are written in `<opcode> <destination>,<operands>` format, which differ slightly for each type of instruction.

R-type instructions have an opcode and three arguments (either registers or shift amount). For example, to add register `$s0` and `$s1`, and store it in register `$t0`, the instruction would be:

```
add $t0, $s0, $s1
```

Shift instructions use the last argument for the shift amount:

```
sll $t0, $s0, 5  #shift s0 left 5 bits and store into t0
```

I-type instructions have two register arguments and an immediate field. For example, to perform a logical OR on `$s0` with the value 0xfeed and store the result in `$t0`, the instruction would be:

```
ori $t0, $s0, 0xfeed
```

You may also specify the immediate field in decimal by leaving off the leading '0x'.

Branch instructions require the immediate field to be a label name:

```
beq $s0, $s1, loop #if s0 == s1, branch to label "loop"
```

J-type instructions have only one argument, and must be a label:

```
jal my_function #jump and link to label "my_function"
```

4.1.2 Pseudo-ops

PLPTool supports a number of pseudo instructions designed to ease programming and make the assembly more human readable. A list of these instructions, as well as what they map to in the core instruction set is given later in this document.

4.1.3 Memory Organization (.org)

In order to resolve branch and jump targets, the programmer must inform the assembler **before** any instructions, labels, or includes, where the program starts in memory. The address must be word aligned (multiple of 4).

```
For example, to begin the program at address 0x10000000 (RAM):
```

4.1.4 Labels

Label support allows the programmer to use branch and jump instructions. Labels are appended with a colon.

For example, to create a label "main":

```
<instructions>
main:
<instructions>
```

4.1.5 Comments

Comments may appear anywhere in the program code, including on label, instruction, and directive lines. Comments are prefixed with a `'#'` character, and all text after the comment character until the end of the line is ignored by the assembler.

```
#a comment on my own line!
add $s0, $s0, $s1 #another comment!
```

4.1.6 Data and String Allocation

There are three ways to allocate space for data with PLPTool: allocating (and optionally initializing) a single word, allocating space in terms of number of words, and by allocating a string.

Allocate and Initialize Word The `.word` directive allocates a single word with or without an initial value. This is especially useful after a label for easy access.

For example, to allocate a variable, initialized to the value 4:

```
my_variable:
.word 4

...

li $t0, my_variable #get a pointer to my variable
lw $t1, 0($t0)      #t1 has my_variable now
```

Empty Space Allocation PLPTool supports allocating space by taking the number of words to allocate, as opposed to a single word with the `.word` directive. This is accomplished using the `.space` directive. For example, to allocate a variable with length of 2 words:

```
long_variable:
.space 2

...

li $t0, long_variable #get a pointer to the variable
lw $t1, 0($t0)        #get first word
lw $t2, 4($t0)        #get second word
```

String Allocation PLPTool also supports two types of string allocation `.ascii, and .asciiz. .ascii` allocates a packed array of characters without a trailing null terminator, which indicates the end of a string. `.asciiz` allocates a packed array of characters with a trailing null terminator.

```
#a string of characters
my_string:
.ascii "a bunch of characters" #no null terminator here!
#a null terminated string of characters
my_string_null:
.asciiz "a bunch of characters" #null terminated! use me for string operations
```

PLPTool also supports escaping newline characters with `\n`.

4.1.7 Notes on the Assembler

`.org` must be the first non-comment statement in the program.

It is possible to load a pointer to a label using the load immediate pseudo instruction (li).

4.2 Ops

4.2.1 R-type Arithmetic and Logical Instructions

4.2.2 R-type Shift Instructions

4.2.3 R-type Jump Register Instructions

4.2.4 I-type Branch Instructions

4.2.5 I-type Arithmetic and Logical Instructions

4.2.6 I-type Load Upper Immediate Instruction

4.2.7 I-type Load and Store Word Instructions

4.2.8 J-type Instructions

4.3 Pseudo-ops

4.4 Notes on Register Usage

5 Hardware Description

5.1 Memory Map

5.2 ROM

The ROM module is a non-volatile, read-only memory that stores the fload bootloader, which is used with PLPTool to load programs over the serial port. The PLP board starts at address 0x00000000 on power-up and board reset, causing the bootloader to run.

5.3 RAM

The RAM module is a volatile, random access memory that stores all downloaded program code and data. Generally, the programmer will place their program in RAM, beginning at the beginning of RAM using the .org statement (.org 0x10000000). Additionally, the stack is generally initialized at the top of RAM (`$sp = 0x10ffffc`).

5.4 UART

The UART module is a UART, running at 57600 baud, with 8 data bits, 1 stop bit, and no parity. The UART module is connected to the serial port on the Nexys 2 development kit.

The UART module is designed to send or receive a single byte at a time, and can only store one byte in the receive buffer. This means that you must read received data (by polling) before the next byte is available.

There are four registers that the UART module uses:

0xf0000000	command register
0xf0000004	status register
0xf0000008	receive buffer
0xf000000c	send buffer

The command register always reads 0. Writing the value 0x01 will initiate a send operation using the lowest byte in the send buffer. Writing a 0x02 will clear the ready flag in the status register.

The status register uses only the bottom two bits, with the remaining bits always reading as 0. status0? is the clear to send bit, which is set after a successful transfer of the data in the send buffer, indicating that another transfer may be made. The cts bit is 0 during a transfer, and the data in the send buffer should not be modified. status1? is the ready bit, which is set when a new byte has been successfully received. The ready bit can be cleared by writing 0x02 to the command register.

The receive buffer holds the last received byte. On a successful receive, the ready bit will be set, allowing the programmer to poll the status register for incoming data. When the ready bit is not set, the receive buffer is considered invalid.

The send buffer holds the byte that will be sent or is ready to send. During a send operation (cts is 0), the data in the transmit buffer must not be modified. Only update the send buffer when cts is set.

The UART supports interrupts, and will trigger an interrupt whenever data is available in the receive buffer. **The user must complete a receive and clear the status bit in the UART before clearing the interrupt status bit for the UART.**

5.5 Switches

The switches module is a read-only register that always holds the current value of the switch positions. There are 8 switches on the Nexys2 development board, which are mapped to the lowest byte of the switches module register. Writing to this location has no effect.

5.6 LEDs

The LEDs module is a read-write register that stores the value of the on-board LEDs. There are 8 LEDs, mapped to the lowest byte of the LED register. Reading this value returns the current LED state, writing will update the LEDs.

5.7 GPIO

5.8 VGA

5.9 PLPID

The PLPID module contains two registers that describe the board identity and frequency. Writing to either register has no effect.

0xf0500000 - PLPID (0xdeadbeef for this version) 0xf0500004 - Board frequency (50MHz, 0x2faf080, for the reference design)

The CPUID module is useful for dynamically calculating wait time in a busy-wait loop. For example, if you wanted to wait .5 seconds, you could read the board frequency, shift right by 1 bit, and call the `libplp_wait` function.

5.10 Timer

The timer module is a single 32-bit counter that increments by one every clock cycle. It can be written to at any time. At overflow, the timer will continue counting. The timer module is useful for waiting a specific amount of time with high resolution (20ns on the reference design).

The timer module supports interrupts, and will trigger an interrupt when the timer overflows. The user can configure a specific timed interrupt by presetting the timer value to N cycles before the overflow condition.