C TO PLP COMPILER

VIMARSH DEO, Arizona State University

Using this compiler, we can produce a low level assembly code for a high level C program. This low level code can be executed on PLP Tool also known as Progressive learning Platform. This gives students a better understanding of how the low level code appears for a high level language such as C.

CCS Concepts: • **Advanced Computer Architecture → Computer Architecture→ Compiler Design**;

General Terms: Design, Low Level Language, Assembly Language, Compiler

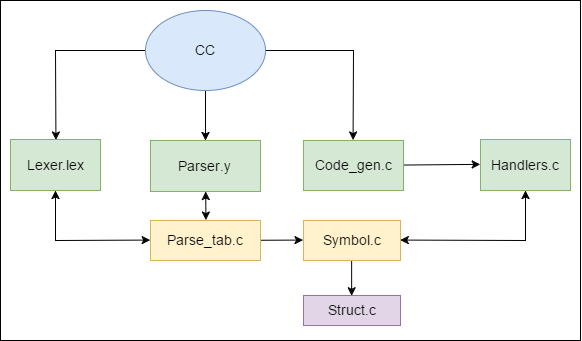
Additional Key Words and Phrases: PLP, Progressive Learning Platform

1 INTRODUCTION

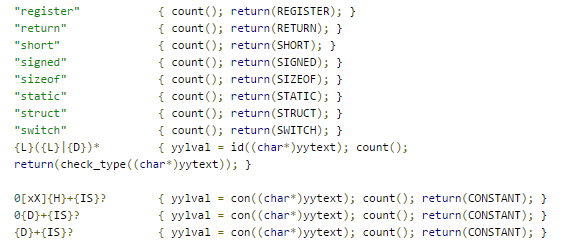
The Progressive Learning Platform (PLP) utilized on the PLP Board provides a unique learning platform designed to be simple, open, and useful for education[1]. PLPTool is the software suite for the Progressive Learning Platform that incorporates an assembler, a board simulator, and a board programming interface. This project is built on the idea of students able to see PLP code of their C Program so that they can learn how for each and every line of high level C language, hundreds of lines of low level assembly language generates. Initially a master test plan was created to highlight the functionalities having been implemented and yet to implement by the compiler. Most commonly used features of C language like print, scan, division, modulus, error message handling etc were handled and hence a compiler powerful enough to generate PLP code for those features was created.

This report states how to install and execute this compiler, Later on, there are two sample C programs along with their PLP code generated from the compiler and screenshots of its output on PLPTool when the code is executed. In the next sections, there is discussion on some limitations and design decision of the compiler in the section of results and discussion. Last but not the least, the report contains conclusion and acknowledgement along with references and appendix for referring to all the functionalities supported.

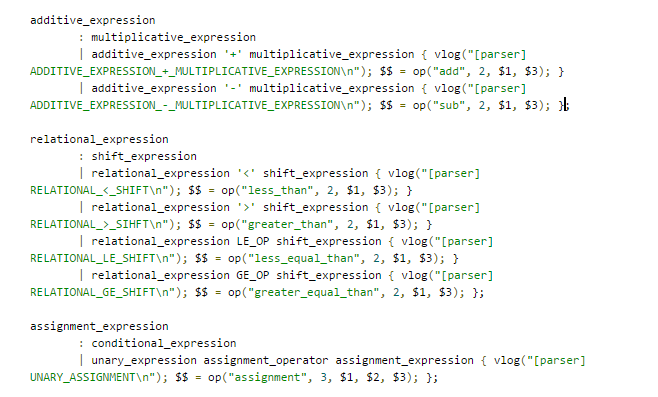
2 OVERVIEW



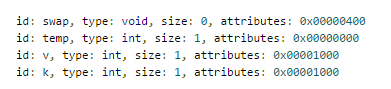
In the above figure, cc denotes the compiler. Initially, lexical analysis is done on the C code. Job of lexer (aka lexical analyzer, scanner and tokenizer) is to divide input into tokens, and discard whitespace and comments. Token is smallest unit useful for parsing. This starts the front end. We use flex which consumes a lexical specification (lexer.lex) and generates C source code(lex.yy.c) for lexical analyzer. In ordinary usage, lex generates a file that can be linked in with the rest of the compiler/interpreter toolchain. To communicate with downstream phases, the C code that lex generates exports the procedure yylex i.e. when a downstream phase needs the next token, it calls yylex(). Below is an example of patterns and their corresponding token retuned by the lexical analyzer (lexer.lex).



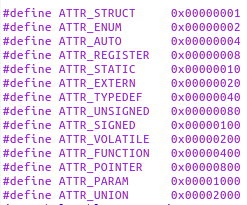
Next Bison parser checks whether the generated tokens form a meaningful expression. This makes use of a context-free grammar that defines algorithmic procedures for components. Parsing a sentence would involve dividing it into words and phrases and identifying the type of each component. It implements LALR(1) parsing and also installs symbol table from symbol.c. Below is an example showing the context free grammar used by parser for parsing (parser.y)



A symbol table is an abstract data type that is used to hold names of variables, constants, and functions. Along with the name, attributes such as data type, data address, data size, initial value, and scope are stored. For example when a = 2 is read by parser.y, parser matches it with declarations and creates a new node with children matching all the input parameters. Below is an example of a symbol table generated by our compiler. It gives information about the function swap, its local variable temp and its arguments, v and k.



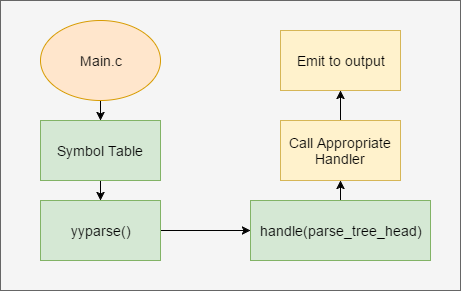
The below table shows the meaning of each attribute used in the symbol table.



Below is an example of a parse tree generated by our compiler. We can see the various components of the parse tree. We need to handle all these components properly in our backend code to generate corresponding PLP code.



After lexing, parsing, and symbol table creation, finally the backend computations code\_gen is called from main.c. When the appropriate id of the node (as seen in the parse table) matches with the possible ids in the handler, corresponding emit function is called from handlers. For handling new functionalities or making changes in the way it was previously handled, we need to make changes in these backend files.



3 EXECUTING THE COMPILER

3.1 Installation

C to PLP Compiler ##Installation Guide [2]

1. Installing Dependencies: Building PLPC requires GNU flex, GNU bison, gcc, and libarchive (git has been included because it is required to clone this repository). The following has been tested on Ubuntu 14.04 LTS (64-bit) using PLPTool 5.2 (link included in instructions), but should work on any Debian Linux distribution:

sudo apt-get install git flex bison gcc libarchive-dev

2. Cloning C to PLP Compiler Repository: The following commands will clone a copy of this repository into your home directory. The remainder of these instructions assume it is cloned to your home directory.

cd ~

git clone https://github.com/Progressive-Learning-Platform/CLP.git

3. Installing PLPTool: You will also need to have a copy of the PLPTool JAR file. The simplest way to get this on a Debian machine is to install the PLPTool Debian package (.deb), which can be found [here](https://github.com/Progressive-Learning-Platform/progressive-learning-platform/releases). For the remainder of this guide it will be assumed that you installed PLPTool using this method.

4. Setting Environment Variables: The top level Makefile, located in the CLP directory, builds and tests the compiler. In order for this process to complete successfully two environment variables need to be set: the bin directory needs to be added to $PATH and $plptool needs to be set to the command used to run PLPTool. The simplest permanent solution to modify .profile, which can be done using the following command (this uses nano, but use whatever text editor you are most comfortable with):

nano ~/.profile

Add the following to the end of the file:

# add plptool environment variable

plptool="java -jar /usr/lib/plptool5/PLPToolStatic.jar"

export plptool

# add PLP compiler bin to PATH

export PATH=$PATH:$HOME/CLP/bin

Use Ctrl+x to exit nano, type y to modify the file, and hit enter to save with the same file name. You must log out (or restart) in order for these changes to take effect.

5. Building and Testing Compiler Now your system should be ready to build and test the compiler. After logging back, in navigate to the CLP directory:

cd ~/CLP

The compiler will be built and the automated test will be run using the following command:

make

3.2 Running the Compiler

Create a file containing a valid C program. For example I create a file called example.c and place it in /bin directory inside /CLP

Now to compile the C program, use ./plpc command

./plpc example.c

It will generate a tar file plpc.plp. To untar the file, use tar command

tar –xvf plpc.plp

It will generate three files, namely entrypoint.asm, plpc.plp.asm and plp.metafile

The files entrypoint.asm, plpc.plp.asm comprise of the entire PLP code for the C program. Entrypoint.asm is the driver file which calls the main function inside plpc.plp.asm file.

The contents of entrypoint.asm file remains the same irrespective of the C program

.org 0x10000000

\_entrypoint:

# enable the stack

li $sp, 0x10fffffc

# call main

call main

# if we get here just halt

\_halt:

j \_halt

nop

4 SAMPLE PROGRAMS/FEATURES

4.1 Bubble Sort

In this section, a bubble sort program is implemented inside which new keywords such as plp\_scanint, plp\_printstring and plp\_printint are used. plp\_scanint[A.4.5] is used to read numbers in array. plp\_printstring[A.4.10] is used to print string on UART and plp\_printint[A.4.8] is used to print integers on UART of the PLPTool. For more information on these features, the appendix can be referred.

C Code:

void main()

{

int array[10], n, c, d, swap;

char a[1];

plp\_printstring("Enter number of elements");

plp\_scanint(n);

a="";

plp\_printstring("Enter integers");

for (c = 0; c < n; c++){

plp\_scanint(array[c]);

plp\_printint(array[c]);

plp\_printstring(a);

}

for (c = 0 ; c < ( n - 1 ); c++)

{

for (d = 0 ; d < n - c - 1; d++)

{

if (array[d] > array[d+1])

{

swap       = array[d];

array[d]   = array[d+1];

array[d+1] = swap;

}

}

}

plp\_printstring("Sorted list in ascending order:");

for ( c = 0 ; c < n ; c++ ){

plp\_printint(array[c]);

plp\_printstring(a);

}

}

Output:

# LINE 5:  plp\_printstring("Enter number of elements");

#

move $v1, $ra

li $0, 0

li $t0, PLPCC\_000008

jal PLPCC\_000001

nop

PLPCC\_000007:

ori $a0, $zero, 0x000d #newline

jal PLPCC\_000002

nop

ori $a0, $zero, 0x000a  #linefeed

jal PLPCC\_000002

nop

j PLPCC\_000000

nop

PLPCC\_000002:

lui $t0, 0xf000 #uart base address

PLPCC\_000003:

lw $t1, 4($t0) #get the uart status

andi $t1, $t1, 0x01 #mask for the cts bit

beq $t1, $zero, PLPCC\_000003

nop

sw $a0, 12($t0) #write the data to the output buffer

sw $t1, 0($t0) #send the data!

jr $31

nop

PLPCC\_000001: #we have a pointer to the string in a0, just loop and increment until we see a null

move $t9, $31 #save the return address

move $t8, $t0 #save the argument

PLPCC\_000004:

lw $a0, 0($t8) #first 1-4 characters

beq $a0, $zero, PLPCC\_000005

nop

PLPCC\_000006:

jal PLPCC\_000002 #write this byte

nop

j PLPCC\_000004

addiu $t8, $t8, 4 #increment for the next word

PLPCC\_000005:

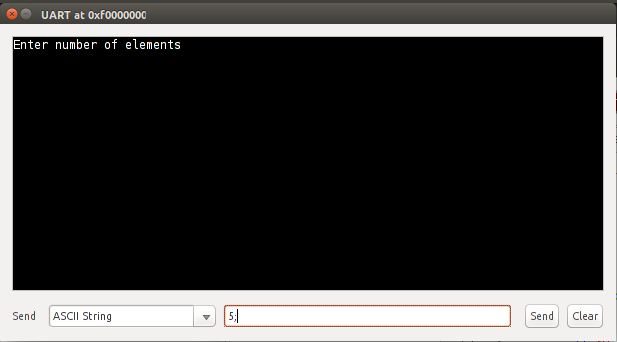
jr $t9 #go home

nop

PLPCC\_000000:

move $ra, $v1

The above block of code got generated for plp\_printstring instruction. The above PLP instructions go on reading the string from the label PLPCC\_000008, and goes on printing it word by word. The below screenshot represents the same.



# LINE 6:  plp\_scanint(n);

#

move $t3, $sp

li $s0, 0xF0000000 # UART

li $s2, 0b10 # Status register bit mask

li $t5, 10 # Used to convert decimal to binary

li $t6, 48 # Used to convert ASCII to decimal

li $t7, 59 # Used to store semi colon

li $t8, 32 # space character

li $t0, 0 # value of current number

PLPCC\_000010:

jal PLPCC\_000011

nop

# Check for end of input

bne $v0, $t7, PLPCC\_000012

nop

push $t0

move $v0, $0

j PLPCC\_000009

nop

PLPCC\_000012:

# Check for space indicating end of number

bne $v0, $t8, PLPCC\_000013

nop

#code for initializing variable

push $t0

move $t0, $0 #clear input

j PLPCC\_000010

nop

# Assume value is a number, convert to binary

PLPCC\_000013:

mullo $t0, $t0, $t5 #multiply current number by 10

subu $v0, $v0, $t6 #convert UART character from ascii to decimal

addu $t0, $t0, $v0

j PLPCC\_000010

#Description: Reads UART and returns it in $v0

# Uses s2, s4, v0

PLPCC\_000011:

PLPCC\_000014:

lw $s4, 4($s0) #load status register

and $s4, $s4, $s2 #mask for ready bit

bne $s4, $s2, PLPCC\_000011

nop

lw $v0, 8($s0) #load from recieve buffer

sw $s2, 0($s0) #command register: clear status

jr $ra

nop

PLPCC\_000009:

pop $t1

addiu $t0, $t3, 16 # \*n

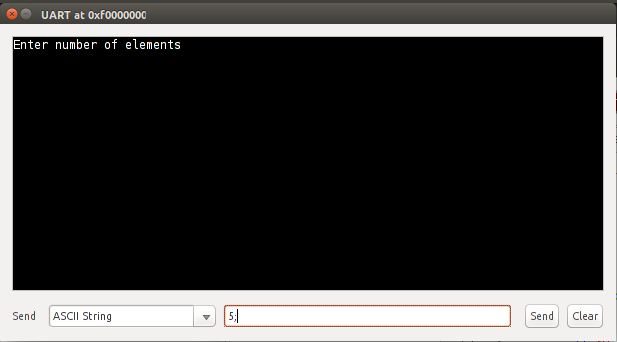
push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

The above piece of code performs the scan function on variable n. It keeps on reading consecutive numbers until a semi colon is encountered and pushes it onto stack. Later it pops the value out of stack and stores it in the memory location of variable n. I entered 5 with semicolon and pressed send as shown in figure below.



PLPCC\_000028:

pop $t1

addiu $t0, $sp, 20 # \*array

push $t0

lw $t0, 16($sp) # c

pop $t2

sll $t0, $t0, 2

addu $t0, $t0, $t2

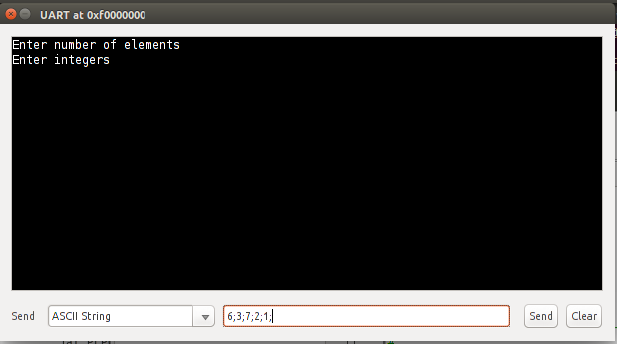
push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

The above code shows how to store values in array after scanning. We know that a scan operation pushes the values onto stack. Here , we pop the value from stack, then calculate the exact address of the array where we want to store the number and then perform store operation of the read value to that memory address. Below is the screenshot which shows entering of numbers in the array. We need to make sure that we put a semicolon after every digit since we have just one argument in scan instruction. For multiple arguments, we put space between numbers before putting semi colon.



#

# LINE 12:  plp\_printint(array[c]);

#

move $v1, $ra

li $0, 0

li $t6, 0

addiu $t0, $sp, 20 # \*array

push $t0

lw $t0, 16($sp) # c

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

slt $t3, $t0, $0

beq $t3, $0, PLPCC\_000046

nop

li $t6, 1 #negative number flag

li $t1, -1

mullo $t0, $t0, $t1

PLPCC\_000046:

li $t1, 10

li $t5, 0 #counter

jal PLPCC\_000035

nop

j PLPCC\_000034

nop

PLPCC\_000035:

li $t4, 0

li $v0, 0 # return value

li $t3, 0 # align shift count

PLPCC\_000036:

beq $t1, $t0, PLPCC\_000037

nop

slt $t2, $t1, $t0 #t1 < t0

beq $t2, $0, PLPCC\_000037

nop

sll $t1, $t1, 1

addiu $t3, $t3, 1

j PLPCC\_000036

nop

PLPCC\_000037:

slt $t2, $t1, $t0 #t0 > t1

beq $t2, $0, PLPCC\_000038

nop

subu $t0, $t0, $t1

PLPCC\_000038:

bne $t1, $t0, PLPCC\_000039

nop

addiu $t2, $t2, 1

subu $t0, $t0, $t1

PLPCC\_000039:

nop

sll $v0, $v0, 1 #shift result left

addu $v0, $v0, $t2

beq $t4, $t3, PLPCC\_000040

nop

srl $t1, $t1, 1

addiu $t4, $t4, 1

j PLPCC\_000037

nop

PLPCC\_000040:

addiu $t0, $t0, 48 #adding ascii value

push $t0

move $t0, $v0

addiu $t5, $t5 , 1

li $t1, 10

slt $t3, $t0, $t1

beq $t3, $zero, PLPCC\_000035

nop

PLPCC\_000041:

addiu $t0, $t0, 48

push $t0

addiu $t5, $t5 , 1

beq $t6 ,$0, PLPCC\_000042

nop

li $t0, 45

push $t0

addiu $t5, $t5 , 1

PLPCC\_000042:

lui $t0, 0xf000 #uart base address

PLPCC\_000043:

lw  $t1, 4($t0) #get the uart status

andi $t1, $t1, 0x01 #mask for the cts bit

beq $t1, $zero, PLPCC\_000043

nop

PLPCC\_000044:

pop $a0

sw  $a0, 12($t0) #write the data to the output buffer

sw  $t1, 0($t0) #send the data!

addiu $t5, $t5, -1 #decrement counter

bne $t5, $zero, PLPCC\_000044

nop

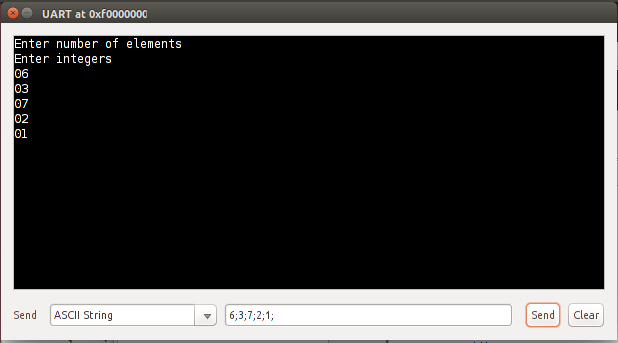
jr $ra

nop

PLPCC\_000034:

move $ra, $v1

The above PLP code shows the handling of the keyword plp\_printint to print the scanned inputs on UART. It goes on reading each number from memory, converts it from number to ascii, and keeps printing the number on UART. The figure below represents the same.



# LINE 16:  for (c = 0 ; c < ( n - 1 ); c++)

#

addiu $t0, $sp, 12 # \*c

push $t0

li $t0, 0

pop $t1

sw $t0, 0($t1)

#

# LINE 27:  }

#

PLPCC\_000055:

lw $t0, 12($sp) # c

push $t0

lw $t0, 20($sp) # n

push $t0

li $t0, 1

pop $t1

subu $t0, $t1, $t0

pop $t1

slt $t0, $t1, $t0

beq $t0, $zero, PLPCC\_000056

nop

#

# LINE 18:  for (d = 0 ; d < n - c - 1; d++)

#

addiu $t0, $sp, 8 # \*d

push $t0

li $t0, 0

pop $t1

sw $t0, 0($t1)

#

# LINE 26:  }

#

PLPCC\_000058:

lw $t0, 8($sp) # d

push $t0

lw $t0, 20($sp) # n

push $t0

lw $t0, 20($sp) # c

pop $t1

subu $t0, $t1, $t0

push $t0

li $t0, 1

pop $t1

subu $t0, $t1, $t0

pop $t1

slt $t0, $t1, $t0

beq $t0, $zero, PLPCC\_000059

nop

#

# LINE 20:  if (array[d] > array[d+1])

#

addiu $t0, $sp, 20 # \*array

push $t0

lw $t0, 12($sp) # d

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

push $t0

addiu $t0, $sp, 24 # \*array

push $t0

lw $t0, 16($sp) # d

push $t0

li $t0, 1

pop $t1

addu $t0, $t0, $t1

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

pop $t1

slt $t0, $t0, $t1

beq $t0, $zero, PLPCC\_000061

nop

#

# LINE 22:  swap = array[d];

#

addiu $t0, $sp, 4 # \*swap

push $t0

addiu $t0, $sp, 24 # \*array

push $t0

lw $t0, 16($sp) # d

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

pop $t1

sw $t0, 0($t1)

#

# LINE 23:  array[d] = array[d+1];

#

addiu $t0, $sp, 20 # \*array

push $t0

lw $t0, 12($sp) # d

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

push $t0

addiu $t0, $sp, 24 # \*array

push $t0

lw $t0, 16($sp) # d

push $t0

li $t0, 1

pop $t1

addu $t0, $t0, $t1

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

pop $t1

sw $t0, 0($t1)

#

# LINE 24:  array[d+1] = swap;

#

addiu $t0, $sp, 20 # \*array

push $t0

lw $t0, 12($sp) # d

push $t0

li $t0, 1

pop $t1

addu $t0, $t0, $t1

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

push $t0

lw $t0, 8($sp) # swap

pop $t1

sw $t0, 0($t1)

PLPCC\_000061:

PLPCC\_000060:

addiu $t0, $sp, 8 # \*d

lw $t1, 0($t0)

addiu $t2, $t1, 1

sw $t2, 0($t0)

move $t0, $t1

j PLPCC\_000058

nop

PLPCC\_000059:

PLPCC\_000057:

addiu $t0, $sp, 12 # \*c

lw $t1, 0($t0)

addiu $t2, $t1, 1

sw $t2, 0($t0)

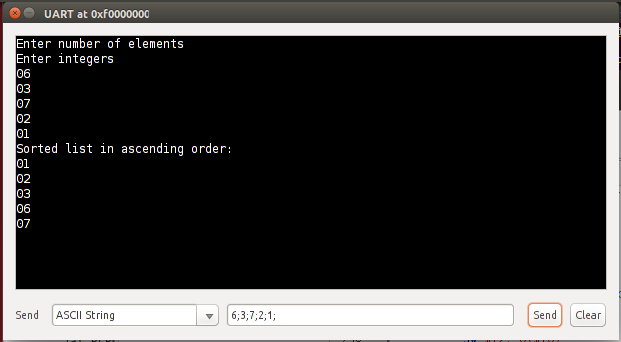
move $t0, $t1

j PLPCC\_000055

nop

PLPCC\_000056:

The above PLP Code shows the entire bubble sort logic to swap the numbers based on their values. It implements two for loops as well. The PLP code for the loops and sort logic is shown above. Here, we load the value of array at index d and index d+1, and compare both. After comparison, if array[d] is greater than array[d+1], we allocate memory for another temporary variable temp and use it to swap the value between these two array indices. This is how sorting is performed and a sorted array is displayed on screen as shown below.



4.2 String Manipulation Program

In this program, we show the usage of do while loop, switch case, modulus operator, strcpy, strlen, strcmp, strcat, plp\_scanchar, plp\_printstring and plp\_printint. This program keeps on reading characters from UART till condition is met and based on the character read from UART, it performs some string operations. For more information on these features, the appendix can be referred.

C Code

void main(){

char ch, a[20], b[10], c[20], d[10], e[10], newline[1] ;

int counter, length, res;

a = "CtoPLP";

b = "enter character";

c = "Switch Case and Do While Ended";

e = " compiler";

newline = "";

counter = 1;

length = 0;

do{

plp\_printstring(b);

plp\_scanchar(ch);

switch (ch){

case 'a': res = strcmp(a,b);

plp\_printint(res);

plp\_printstring(newline);

counter++;

break;

case 'b': length = strlen(a);

plp\_printint(length);

plp\_printstring(newline);

counter++;

break;

case 'c': strcpy(d,"Used strcpy");

plp\_printstring(d);

counter++;

break;

case 'd': strcat(a,e);

plp\_printstring(a);

counter++;

break;

default: counter++;

break;

}

}while(counter%5 != 0);

plp\_printstring(c);

}

Output:

PLPCC\_000021:

PLPCC\_000007:

li $t0, 5

move $t1, $t0

lw $t0, 300($sp) # counter

PLPCC\_000097:

li $v0, 0 # return value

li $t3, 0 # align shift count

li $t4, 0 # loop counter

PLPCC\_000098:

beq $t1, $t0, PLPCC\_000099

nop

slt $t2, $t1, $t0 #t1 < t0

beq $t2, $0, PLPCC\_000099

nop

sll $t1, $t1, 1

addiu $t3, $t3, 1

j PLPCC\_000098

nop

PLPCC\_000099:

slt $t2, $t1, $t0 #t0 > t1

beq $t2, $0, PLPCC\_000100

nop

subu $t0, $t0, $t1

PLPCC\_000100:

bne $t1, $t0, PLPCC\_000101

nop

addiu $t2, $t2, 1

subu $t0, $t0, $t1

PLPCC\_000101:

nop

sll $v0, $v0, 1 #shift result left

addu $v0, $v0, $t2

beq $t4, $t3, PLPCC\_000102

nop

srl $t1, $t1, 1

addiu $t4, $t4, 1

j PLPCC\_000099

nop

PLPCC\_000102:

push $t0

li $t0, 0

pop $t1

move $t2, $t0

move $t0, $zero

beq $t1, $t2, PLPCC\_000096

nop

ori $t0, $zero, 1

PLPCC\_000096:

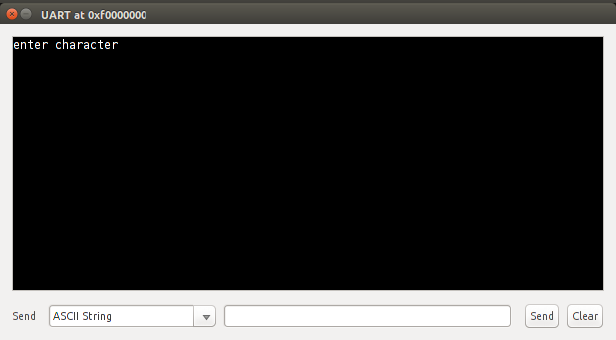
beq $t0, $zero, PLPCC\_000006

nop

j PLPCC\_000005

nop

Initially, when we execute the plp code on tool, it initiates a do while loop where it checks the value of a variable called counter by using modulus function. Inside the loop, it enters a switch case where we are asked to enter a character. The above PLP code shows the condition mentioned in do while loop. It performs the modulus function on counter variable and compares with the zero.



#

# LINE 15:  switch (ch){

#

lw $t0, 288($sp) # ch

#

# LINE 16:  case 'a': res = strcmp(a,b);

#

push $t0

li $t0, 'a'

pop $t1

move $t2, $t0

move $t0, $t1

bne $t1, $t2, PLPCC\_000022

nop

addiu $t0, $sp, 292 # \*res

push $t0

lw $t0, 172($sp) # b

move $t1, $t0

lw $t0, 212($sp) # a

PLPCC\_000023:

lw $t2, 0($t0) #read from string1

lw $t3, 0($t1) #read from string2

bne $t2, $t3, PLPCC\_000024

nop

beq $t2, $0, PLPCC\_000024

nop

beq $t3, $0, PLPCC\_000024

nop

addiu $t0, $t0, 4

addiu $t1, $t1, 4

j PLPCC\_000023

nop

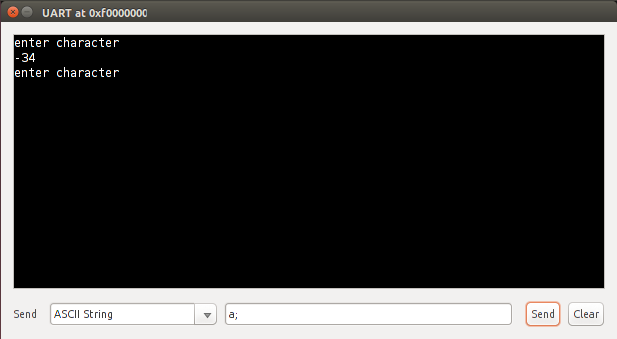
PLPCC\_000024:

subu $t0, $t2, $t3

pop $t1

sw $t0, 0($t1)

The above PLP code shows how the character is read in switch case. When we enter ‘a’ with semicolon, which marks the end of input, it does the first operation in switch case which is strcmp operation as seen in C code and prints the result, -34, on screen and increments the counter. As we are in a do while loop, we again enter the switch case where it asks to enter another character.



# LINE 21:  case 'b': length = strlen(a);

#

push $t0

li $t0, 'b'

pop $t1

move $t2, $t0

move $t0, $t1

bne $t1, $t2, PLPCC\_000046

nop

addiu $t0, $sp, 296 # \*length

push $t0

lw $t0, 212($sp) # a

li $t1, 0 #counter

PLPCC\_000047:

lw $t2, 0($t0)

beq $t2, $0, PLPCC\_000048

nop

addiu $t0, $t0, 4 #increment memory

addiu $t1, $t1, 1 #increment counter

j PLPCC\_000047

nop

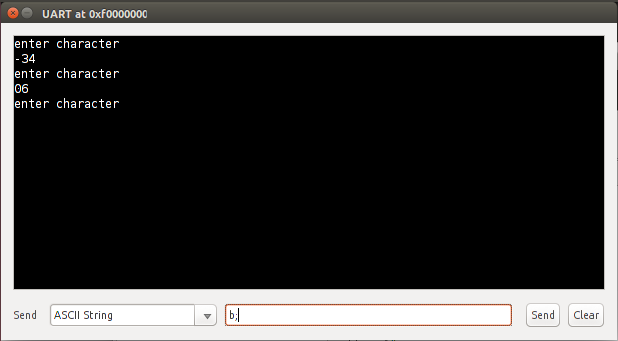
PLPCC\_000048:

move $t0, $t1

pop $t1

sw $t0, 0($t1)

The above PLP code shows the working of strlen when we enter ‘b’. It calculates the length of the string using strlen and prints it on the screen which is 6. Counter gets incremented and it asks us to enter another character.



#

# LINE 26:  case 'c': strcpy(d,"Used strcpy");

#

push $t0

li $t0, 'c'

pop $t1

move $t2, $t0

move $t0, $t1

bne $t1, $t2, PLPCC\_000070

nop

li $t0, PLPCC\_000073

move $t1, $t0

addiu $t0, $sp, 48 # \*d

move $t2, $t0

li $t0, PLPCC\_000074

sw $t0, 0($t2)

PLPCC\_000071:

lw $t2, 0($t1)

beq $t2, $0, PLPCC\_000072

nop

sw $t2, 0($t0)

addiu $t0, $t0, 4

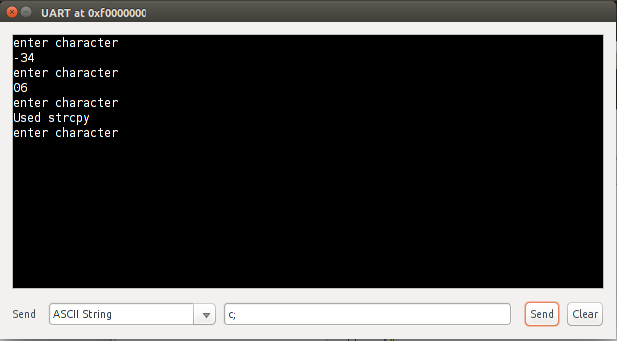
addiu $t1, $t1, 4

j PLPCC\_000071

nop

PLPCC\_000072:

sw $t2, 0($t0)

The above PLP code shows the working of strcpy when we enter ‘c’. Inside switch case, it uses strcpy to copy a string to a new variable and print it on the screen, which is “Used strcpy”. Counter gets incremented and it asks us to enter another character.

#

# LINE 30:  case 'd': strcat(a,e);

#

push $t0

li $t0, 'd'

pop $t1

move $t2, $t0

move $t0, $t1

bne $t1, $t2, PLPCC\_000083

nop

lw $t0, 8($sp) # e

move $t1, $t0

lw $t0, 208($sp) # a

PLPCC\_000084:

lw $t2, 0($t0)

beq $t2, $0, PLPCC\_000085

nop

addiu $t0, $t0, 4

j PLPCC\_000084

nop

PLPCC\_000085:

lw $t2, 0($t1)

beq $t2, $0, PLPCC\_000086

nop

sw $t2, 0($t0)

addiu $t0, $t0, 4

addiu $t1, $t1, 4

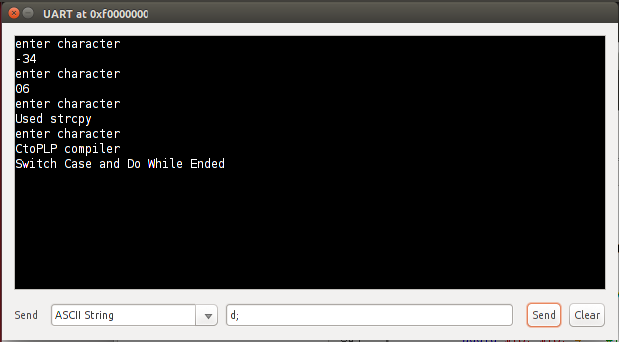
j PLPCC\_000085

nop

PLPCC\_000086:

sw $t2, 0($t0)

The above PLP code shows the working of strcat when we enter‘d’. Inside switch case, it uses strcat function to merge two strings and print it on the screen, which is “CtoPLP Compiler”. Counter gets incremented and come out of the while loop. After the loop ends, it prints another message on the screen which is “Switch Case and Do While ended”



5 RESULTS AND DISCUSSION

Although most of the implementations in this compiler are the same as in standard C, but there are some design decisions we made for this compiler to make the assembly code produced, PLP compatible. The print and scan library functions of C cannot be implemented on PLP Tool. PLPtool cannot take input from keyboard and display on monitor. It can take input and display output on UART only. Hence we defined our own keywords, plp\_scanchar, plp\_scanint, plp\_scanstring, plp\_printchar, plp\_printint and plp\_printstring. The keywords getchar, putchar, gets and puts read and display on UART. Also, we cannot read a semi colon from UART using plp\_scanchar, plp\_scanstring or gets since semicolon represents the end of input on UART. For some keywords, we can have one or more arguments. The syntax and usage of these functions can be found in the Appendix. Also, we are supporting datatypes of int, char, struct, union and void. For rest of the datatypes such as float and double which are supported in C, our compiler will throw an error message stating that the corresponding datatype is not supported. Also, we have stored the value of return address and stack pointer in specific registers at the start of each function to make sure we return to the correct place the function is being called from.

6 CONCLUSIONS

This report introduced CtoPLP compiler installation and execution details. Also, it has mentioned two sample programs in C, their corresponding PLP code and the screenshots of the output as produced on PLPTool. These programs have covered most of the looping and decision making instructions, mathematical instructions, string functions and library functions implemented in this compiler. This report also contains appendix for further details on the functions implemented in the compiler. Using this compiler, students will be able to easily generate PLP code for their C program. They will have a better understanding of low level assembly code and how their usage.

ACKNOWLEDGMENTS

I would like to express my gratitude to my professor Dr. Sohum Sohoni whose mentoring has been the guiding light in my project. He used to conduct regular meetings to help me understand the requirements of this project and would discuss any roadblocks for implementing the same, using which I could deliver the project in the estimated time.

REFERENCES

|  |  |
| --- | --- |
| [1] | http://progressive-learning-platform.github.io/plptool.html |
| [2] | https://github.com/Progressive-Learning-Platform/CLP |

APPENDIX:

A.1 Looping and Decision Making Instructions

A.1.1 If Condition

If statement can be used for decision making inside a program.

C Code:

void main( )

{

int x,y,z;

x=15;

y=13;

if (x > y )

{

z = x;

}

}

PLP Code:

#

# LINE 9: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -12

#

# LINE 3:  x=15;

#

addiu $t0, $sp, 12 # \*x

push $t0

li $t0, 15

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  y=13;

#

addiu $t0, $sp, 8 # \*y

push $t0

li $t0, 13

pop $t1

sw $t0, 0($t1)

#

# LINE 5:  if (x > y )

#

lw $t0, 12($sp) # x

push $t0

lw $t0, 12($sp) # y

pop $t1

slt $t0, $t0, $t1

beq $t0, $zero, PLPCC\_000000

nop

#

# LINE 7:  z = x;

#

addiu $t0, $sp, 4 # \*z

push $t0

lw $t0, 16($sp) # x

pop $t1

sw $t0, 0($t1)

PLPCC\_000000:

addiu $sp, $sp, 16 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.1.2 Continue Keyword

Continue keyword can be used to skip instructions in that iteration and jump to next iteration of the loop.

C Code:

void main(){

int i,sum;

int a[5] = {1,2,3,4,5};

sum = 0;

for(i = 0;i<5;i++){

if(a[i]>3){

       continue;

}

sum += a[i];

}

}

PLP Code:

#

# LINE 11: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -108

#

# LINE 2:  int a[5] = {1,2,3,4,5};

#

li $t0, 1

sw $t0, 12($sp)

li $t0, 2

sw $t0, 16($sp)

li $t0, 3

sw $t0, 20($sp)

li $t0, 4

sw $t0, 24($sp)

li $t0, 5

sw $t0, 28($sp)

#

# LINE 3:  sum = 0;

#

addiu $t0, $sp, 4 # \*sum

push $t0

li $t0, 0

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  for(i = 0;i<5;i++){

#

addiu $t0, $sp, 8 # \*i

push $t0

li $t0, 0

pop $t1

sw $t0, 0($t1)

#

# LINE 9:  }

#

PLPCC\_000000:

lw $t0, 8($sp) # i

push $t0

li $t0, 5

pop $t1

slt $t0, $t1, $t0

beq $t0, $zero, PLPCC\_000001

nop

#

# LINE 5:                 if(a[i]>3){

#

addiu $t0, $sp, 12 # \*a

push $t0

lw $t0, 12($sp) # i

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

push $t0

li $t0, 3

pop $t1

slt $t0, $t0, $t1

#

# LINE 8:  sum += a[i];

#

beq $t0, $zero, PLPCC\_000003

nop

#

# LINE 6:  continue;

#

j PLPCC\_000002

nop

PLPCC\_000003:

addiu $t0, $sp, 4 # \*sum

push $t0

addiu $t0, $sp, 16 # \*a

push $t0

lw $t0, 16($sp) # i

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

pop $t1

lw $t2, 0($t1)

addu $t0, $t0, $t2

sw $t0, 0($t1)

PLPCC\_000002:

addiu $t0, $sp, 8 # \*i

lw $t1, 0($t0)

addiu $t2, $t1, 1

sw $t2, 0($t0)

move $t0, $t1

j PLPCC\_000000

nop

PLPCC\_000001:

addiu $sp, $sp, 112 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.1.3 Break Keyword

Break can be used to come out of the innermost loop or switch case inside a program.

C Code:

void main(){

int sum,i;

int a[5] = {1,2,3,4,5};

sum = 0;

for(i = 0;i<5;i++){

if(a[i]>3){

       break;

}

sum += a[i];

}

}

PLP Code:

#

# LINE 11: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -108

#

# LINE 2:  int a[5] = {1,2,3,4,5};

#

li $t0, 1

sw $t0, 12($sp)

li $t0, 2

sw $t0, 16($sp)

li $t0, 3

sw $t0, 20($sp)

li $t0, 4

sw $t0, 24($sp)

li $t0, 5

sw $t0, 28($sp)

#

# LINE 3:  sum = 0;

#

addiu $t0, $sp, 8 # \*sum

push $t0

li $t0, 0

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  for(i = 0;i<5;i++){

#

addiu $t0, $sp, 4 # \*i

push $t0

li $t0, 0

pop $t1

sw $t0, 0($t1)

#

# LINE 9:  }

#

PLPCC\_000000:

lw $t0, 4($sp) # i

push $t0

li $t0, 5

pop $t1

slt $t0, $t1, $t0

beq $t0, $zero, PLPCC\_000001

nop

#

# LINE 5: if(a[i]>3){

#

addiu $t0, $sp, 12 # \*a

push $t0

lw $t0, 8($sp) # i

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

push $t0

li $t0, 3

pop $t1

slt $t0, $t0, $t1

#

# LINE 8:  sum += a[i];

#

beq $t0, $zero, PLPCC\_000003

nop

#

# LINE 6:  break;

#

j PLPCC\_000001

nop

PLPCC\_000003:

addiu $t0, $sp, 8 # \*sum

push $t0

addiu $t0, $sp, 16 # \*a

push $t0

lw $t0, 12($sp) # i

pop $t1

sll $t0, $t0, 2

addu $t0, $t0, $t1

lw $t0, 0($t0)

pop $t1

lw $t2, 0($t1)

addu $t0, $t0, $t2

sw $t0, 0($t1)

PLPCC\_000002:

addiu $t0, $sp, 4 # \*i

lw $t1, 0($t0)

addiu $t2, $t1, 1

sw $t2, 0($t0)

move $t0, $t1

j PLPCC\_000000

nop

PLPCC\_000001:

addiu $sp, $sp, 112 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.1.4 Switch Case

Switch case is used for decision making inside a program.

C Code:

void main( )

{

int a,b,c,choice;

a= 20, b= 10, choice = 1;

switch(choice)

{

case 1:

c=a+b;

break;

case 2:

c=a-b;

break;

default:

c=0;

}

}

PLP Code:

#

# LINE 17: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -16

#

# LINE 3:  a= 20, b= 10, choice = 1;

#

addiu $t0, $sp, 16 # \*a

push $t0

li $t0, 20

pop $t1

sw $t0, 0($t1)

addiu $t0, $sp, 12 # \*b

push $t0

li $t0, 10

pop $t1

sw $t0, 0($t1)

addiu $t0, $sp, 4 # \*choice

push $t0

li $t0, 1

pop $t1

sw $t0, 0($t1)

#

# LINE 5:  switch(choice)

#

lw $t0, 4($sp) # choice

#

# LINE 8:  c=a+b;

#

push $t0

#

# LINE 7:  case 1:

#

li $t0, 1

pop $t1

move $t2, $t0

move $t0, $t1

bne $t1, $t2, PLPCC\_000001

nop

addiu $t0, $sp, 8 # \*c

push $t0

lw $t0, 20($sp) # a

push $t0

lw $t0, 20($sp) # b

pop $t1

addu $t0, $t0, $t1

pop $t1

sw $t0, 0($t1)

#

# LINE 9:  break;

#

j PLPCC\_000000

nop

PLPCC\_000001:

#

# LINE 11:  c=a-b;

#

push $t0

#

# LINE 10:  case 2:

#

li $t0, 2

pop $t1

move $t2, $t0

move $t0, $t1

bne $t1, $t2, PLPCC\_000002

nop

addiu $t0, $sp, 8 # \*c

push $t0

lw $t0, 20($sp) # a

push $t0

lw $t0, 20($sp) # b

pop $t1

subu $t0, $t1, $t0

pop $t1

sw $t0, 0($t1)

#

# LINE 12:  break;

#

j PLPCC\_000000

nop

PLPCC\_000002:

#

# LINE 14:  c=0;

#

PLPCC\_000003:

addiu $t0, $sp, 8 # \*c

push $t0

li $t0, 0

pop $t1

sw $t0, 0($t1)

#

# LINE 15:  }

#

PLPCC\_000000:

#

addiu $sp, $sp, 20 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.1.5 Do While Loop

Do while is used for iteration inside a program.

C Code:

void main()

{

int a,i;

a=5;

i=1;

do

{

a= a\*i;

i++;

}while(i <= 5);

}

PLP Code:

#

# LINE 10: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -8

#

# LINE 3:  a=5;

#

addiu $t0, $sp, 8 # \*a

push $t0

li $t0, 5

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  i=1;

#

addiu $t0, $sp, 4 # \*i

push $t0

li $t0, 1

pop $t1

sw $t0, 0($t1)

#

# LINE 9:  }while(i <= 5);

#

PLPCC\_000000:

#

# LINE 7:  a= a\*i;

#

addiu $t0, $sp, 8 # \*a

push $t0

lw $t0, 12($sp) # a

push $t0

lw $t0, 12($sp) # i

pop $t1

mullo $t0, $t0, $t1

pop $t1

sw $t0, 0($t1)

#

# LINE 8:  i++;

#

addiu $t0, $sp, 4 # \*i

lw $t1, 0($t0)

addiu $t2, $t1, 1

sw $t2, 0($t0)

move $t0, $t1

PLPCC\_000002:

lw $t0, 4($sp) # i

push $t0

li $t0, 5

pop $t1

slt $t0, $t0, $t1

nor $t0, $t0, $t0 # $t0 = !$t0

andi $t0, $t0, 1 # Mask LSB

beq $t0, $zero, PLPCC\_000001

nop

j PLPCC\_000000

nop

PLPCC\_000001:

addiu $sp, $sp, 12 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.1.6. Goto Statement

Goto is used to jump to any defined label in a program.

C Code:

void main(){

int i;

i=0;

label1:

if(i!=5){

i++;

goto label1;

}

}

PLP Code:

#

# LINE 9: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -4

#

# LINE 2:  i=0;

#

addiu $t0, $sp, 4 # \*i

push $t0

li $t0, 0

pop $t1

sw $t0, 0($t1)

label1:

#

# LINE 4:  if(i!=5){

#

lw $t0, 4($sp) # i

push $t0

li $t0, 5

pop $t1

move $t2, $t0

move $t0, $zero

beq $t1, $t2, PLPCC\_000001

nop

ori $t0, $zero, 1

PLPCC\_000001:

beq $t0, $zero, PLPCC\_000000

nop

#

# LINE 5:  i++;

#

addiu $t0, $sp, 4 # \*i

lw $t1, 0($t0)

addiu $t2, $t1, 1

sw $t2, 0($t0)

move $t0, $t1

#

# LINE 6:  goto label1;

#

j label1

nop

PLPCC\_000000:

addiu $sp, $sp, 8 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.2 Mathematics Instructions

A.2.1 Unary Plus

Unary Plus is used in the same way as used in C.

C Code:

void main(){

int a, b;

a= -10;

b = +a;

}

PLP Code:

#

# LINE 4: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -8

#

# LINE 2:  a= -10;

#

addiu $t0, $sp, 8 # \*a

push $t0

li $t0, 10

li $t1, -1

mullo $t0, $t0, $t1

pop $t1

sw $t0, 0($t1)

#

# LINE 3:  b = +a;

#

addiu $t0, $sp, 4 # \*b

push $t0

lw $t0, 12($sp) # a

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 12 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.2.2 Unary Minus

Unary Minus is used to negate an integer inside a program.

C Code:

void main(){

int a = 10;

a= -a;

}

PLP Code:

#

# LINE 3: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -4

#

# LINE 1:  int a = 10;

#

li $t0, 10

sw $t0, 4($sp)

#

# LINE 2:  a= -a;

#

addiu $t0, $sp, 4 # \*a

push $t0

lw $t0, 8($sp) # a

li $t1, -1

mullo $t0, $t0, $t1

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 8 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.2.3 Division

Division operator is used for division between two operands inside a program.

C Code:

void main(){

int a,b,c;

a= 10;

b = 2;

c = a/b;

}

PLP Code:

#

# LINE 5: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -12

#

# LINE 2:  a= 10;

#

addiu $t0, $sp, 12 # \*a

push $t0

li $t0, 10

pop $t1

sw $t0, 0($t1)

#

# LINE 3:  b = 2;

#

addiu $t0, $sp, 8 # \*b

push $t0

li $t0, 2

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  c = a/b;

#

addiu $t0, $sp, 4 # \*c

push $t0

lw $t0, 12($sp) # b

move $t1, $t0

lw $t0, 16($sp) # a

PLPCC\_000000:

li $v0, 0 # return value

li $t3, 0 # align shift count

li $t4, 0 # loop counter

PLPCC\_000001:

beq $t1, $t0, PLPCC\_000002

nop

slt $t2, $t1, $t0 #t1 < t0

beq $t2, $0, PLPCC\_000002

nop

sll $t1, $t1, 1

addiu $t3, $t3, 1

j PLPCC\_000001

nop

PLPCC\_000002:

slt $t2, $t1, $t0 #t0 > t1

beq $t2, $0, PLPCC\_000003

nop

subu $t0, $t0, $t1

PLPCC\_000003:

bne $t1, $t0, PLPCC\_000004

nop

addiu $t2, $t2, 1

subu $t0, $t0, $t1

PLPCC\_000004:

nop

sll $v0, $v0, 1 #shift result left

addu $v0, $v0, $t2

beq $t4, $t3, PLPCC\_000005

nop

srl $t1, $t1, 1

addiu $t4, $t4, 1

j PLPCC\_000002

nop

PLPCC\_000005:

move $t0, $v0

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 16 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.2.4 Modulus

Modulus operator is used to calculate remainder inside a program.

C Code:

void main(){

int a,b,c;

a= 10;

b = 3;

c = a%b;

}

PLP Code:

#

# LINE 5: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -12

#

# LINE 2:  a= 10;

#

addiu $t0, $sp, 12 # \*a

push $t0

li $t0, 10

pop $t1

sw $t0, 0($t1)

#

# LINE 3:  b = 3;

#

addiu $t0, $sp, 8 # \*b

push $t0

li $t0, 3

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  c = a%b;

#

addiu $t0, $sp, 4 # \*c

push $t0

lw $t0, 12($sp) # b

move $t1, $t0

lw $t0, 16($sp) # a

PLPCC\_000000:

li $v0, 0 # return value

li $t3, 0 # align shift count

li $t4, 0 # loop counter

PLPCC\_000001:

beq $t1, $t0, PLPCC\_000002

nop

slt $t2, $t1, $t0 #t1 < t0

beq $t2, $0, PLPCC\_000002

nop

sll $t1, $t1, 1

addiu $t3, $t3, 1

j PLPCC\_000001

nop

PLPCC\_000002:

slt $t2, $t1, $t0 #t0 > t1

beq $t2, $0, PLPCC\_000003

nop

subu $t0, $t0, $t1

PLPCC\_000003:

bne $t1, $t0, PLPCC\_000004

nop

addiu $t2, $t2, 1

subu $t0, $t0, $t1

PLPCC\_000004:

nop

sll $v0, $v0, 1 #shift result left

addu $v0, $v0, $t2

beq $t4, $t3, PLPCC\_000005

nop

srl $t1, $t1, 1

addiu $t4, $t4, 1

j PLPCC\_000002

nop

PLPCC\_000005:

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 16 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.2.5 Division Assignment

Division Assignment is used to perform division by a number and initialize that number with the result.

C Code:

void main(){

int a,b;

a= 10;

b = 3;

a /= b;

}

PLP Code:

#

# LINE 5: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -8

#

# LINE 2:  a= 10;

#

addiu $t0, $sp, 8 # \*a

push $t0

li $t0, 10

pop $t1

sw $t0, 0($t1)

#

# LINE 3:  b = 3;

#

addiu $t0, $sp, 4 # \*b

push $t0

li $t0, 3

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  a /= b;

#

addiu $t0, $sp, 8 # \*a

push $t0

lw $t0, 8($sp) # b

pop $t1

lw $t2, 0($t1)

push $t1

move $t1, $t0

move $t0, $t2

PLPCC\_000000:

li $v0, 0 # return value

li $t3, 0 # align shift count

li $t4, 0 # loop counter

PLPCC\_000001:

beq $t1, $t0, PLPCC\_000002

nop

slt $t2, $t1, $t0 #t1 < t0

beq $t2, $0, PLPCC\_000002

nop

sll $t1, $t1, 1

addiu $t3, $t3, 1

j PLPCC\_000001

nop

PLPCC\_000002:

slt $t2, $t1, $t0 #t0 > t1

beq $t2, $0, PLPCC\_000003

nop

subu $t0, $t0, $t1

PLPCC\_000003:

bne $t1, $t0, PLPCC\_000004

nop

addiu $t2, $t2, 1

subu $t0, $t0, $t1

PLPCC\_000004:

nop

sll $v0, $v0, 1 #shift result left

addu $v0, $v0, $t2

beq $t4, $t3, PLPCC\_000005

nop

srl $t1, $t1, 1

addiu $t4, $t4, 1

j PLPCC\_000002

nop

PLPCC\_000005:

move $t0, $v0

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 12 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.2.6 Modulus Assignment

Modulus Assignment is used to perform division by a number and initialize the number with the remainder.

C Code:

void main(){

int a,b;

a= 10;

b = 3;

a %= b;

}

PLP Code:

#

# LINE 5: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -8

#

# LINE 2:  a= 10;

#

addiu $t0, $sp, 8 # \*a

push $t0

li $t0, 10

pop $t1

sw $t0, 0($t1)

#

# LINE 3:  b = 3;

#

addiu $t0, $sp, 4 # \*b

push $t0

li $t0, 3

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  a %= b;

#

addiu $t0, $sp, 8 # \*a

push $t0

lw $t0, 8($sp) # b

pop $t1

lw $t2, 0($t1)

push $t1

move $t1, $t0

move $t0, $t2

PLPCC\_000000:

li $v0, 0 # return value

li $t3, 0 # align shift count

li $t4, 0 # loop counter

PLPCC\_000001:

beq $t1, $t0, PLPCC\_000002

nop

slt $t2, $t1, $t0 #t1 < t0

beq $t2, $0, PLPCC\_000002

nop

sll $t1, $t1, 1

addiu $t3, $t3, 1

j PLPCC\_000001

nop

PLPCC\_000002:

slt $t2, $t1, $t0 #t0 > t1

beq $t2, $0, PLPCC\_000003

nop

subu $t0, $t0, $t1

PLPCC\_000003:

bne $t1, $t0, PLPCC\_000004

nop

addiu $t2, $t2, 1

subu $t0, $t0, $t1

PLPCC\_000004:

nop

sll $v0, $v0, 1 #shift result left

addu $v0, $v0, $t2

beq $t4, $t3, PLPCC\_000005

nop

srl $t1, $t1, 1

addiu $t4, $t4, 1

j PLPCC\_000002

nop

PLPCC\_000005:

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 12 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.3 String Functions

A.3.1 Strcpy

Strcpy is used to copy from one string to another string.

C Code:

void main()

{

char a[20],b[20];

a="PLP";

strcpy(b,a);

}

PLP Code:

#

# LINE 6: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -160

#

# LINE 3:  a="PLP";

#

addiu $t0, $sp, 84 # \*a

push $t0

li $t0, PLPCC\_000000

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  strcpy(b,a);

#

lw $t0, 84($sp) # a

move $t1, $t0

addiu $t0, $sp, 4 # \*b

move $t2, $t0

li $t0, PLPCC\_000003

sw $t0, 0($t2)

PLPCC\_000001:

lw $t2, 0($t1)

beq $t2, $0, PLPCC\_000002

nop

sw $t2, 0($t0)

addiu $t0, $t0, 4

addiu $t1, $t1, 4

j PLPCC\_000001

nop

PLPCC\_000002:

sw $t2, 0($t0)

addiu $sp, $sp, 164 # Epilogue

move $sp, $s7

move $ra, $a3

return

PLPCC\_000000:

.asciiw "PLP"

.space 1

PLPCC\_000003:

.space 20

A.3.2 Strlen

Strlen is used to calculate the length of a string.

C Code:

void main ()

{

char str[50];

int len;

str= "This is C to plp testing";

len = strlen(str);

return;

}

PLP Code:

#

# LINE 9: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -204

#

# LINE 5:  str= "This is C to plp testing";

#

addiu $t0, $sp, 4 # \*str

push $t0

li $t0, PLPCC\_000000

pop $t1

sw $t0, 0($t1)

#

# LINE 7:  len = strlen(str);

#

addiu $t0, $sp, 204 # \*len

push $t0

lw $t0, 8($sp) # str

li $t1, 0 #counter

PLPCC\_000001:

lw $t2, 0($t0)

beq $t2, $0, PLPCC\_000002

nop

addiu $t0, $t0, 4 #increment memory

addiu $t1, $t1, 1 #increment counter

j PLPCC\_000001

nop

PLPCC\_000002:

move $t0, $t1

pop $t1

sw $t0, 0($t1)

#

# LINE 8:  return;

#

move $v0, $t0

addiu $sp, $sp, 208 # Epilogue

move $sp, $s7

move $ra, $a3

return

PLPCC\_000000:

.asciiw "This is C to plp testing"

.space 1

A.3.3 Strcmp

Strcmp is used to compare two strings and returns the difference of both.

C Code:

void main()

{

int c;

char a[20],b[20];

a="PLP";

b = "PLP";

c = strcmp(b,a);

}

PLP Code:

#

# LINE 8: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -164

#

# LINE 4:  a="PLP";

#

addiu $t0, $sp, 88 # \*a

push $t0

li $t0, PLPCC\_000000

pop $t1

sw $t0, 0($t1)

#

# LINE 5:  b = "PLP";

#

addiu $t0, $sp, 8 # \*b

push $t0

li $t0, PLPCC\_000001

pop $t1

sw $t0, 0($t1)

#

# LINE 6:  c = strcmp(b,a);

#

addiu $t0, $sp, 4 # \*c

push $t0

lw $t0, 92($sp) # a

move $t1, $t0

lw $t0, 12($sp) # b

PLPCC\_000002:

lw $t2, 0($t0) #read from string1

lw $t3, 0($t1) #read from string2

bne $t2, $t3, PLPCC\_000003

nop

beq $t2, $0, PLPCC\_000003

nop

beq $t3, $0, PLPCC\_000003

nop

addiu $t0, $t0, 4

addiu $t1, $t1, 4

j PLPCC\_000002

nop

PLPCC\_000003:

subu $t0, $t2, $t3

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 168 # Epilogue

move $sp, $s7

move $ra, $a3

return

PLPCC\_000000:

.asciiw "PLP"

.space 1

PLPCC\_000001:

.asciiw "PLP"

.space 1

A.3.4 Strcat

Strcat is used to concatenate two strings and store the result in the first string.

C Code:

void main()

{

int c;

char a[20],b[20];

a="PLP";

b = "Cto";

strcat(b,a);

}

PLP Code:

#

# LINE 8: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -164

#

# LINE 4:  a="PLP";

#

addiu $t0, $sp, 88 # \*a

push $t0

li $t0, PLPCC\_000000

pop $t1

sw $t0, 0($t1)

#

# LINE 5:  b = "Cto";

#

addiu $t0, $sp, 8 # \*b

push $t0

li $t0, PLPCC\_000001

pop $t1

sw $t0, 0($t1)

#

# LINE 6:  strcat(b,a);

#

lw $t0, 88($sp) # a

move $t1, $t0

lw $t0, 8($sp) # b

PLPCC\_000002:

lw $t2, 0($t0)

beq $t2, $0, PLPCC\_000003

nop

addiu $t0, $t0, 4

j PLPCC\_000002

nop

PLPCC\_000003:

lw $t2, 0($t1)

beq $t2, $0, PLPCC\_000004

nop

sw $t2, 0($t0)

addiu $t0, $t0, 4

addiu $t1, $t1, 4

j PLPCC\_000003

nop

PLPCC\_000004:

sw $t2, 0($t0)

addiu $sp, $sp, 168 # Epilogue

move $sp, $s7

move $ra, $a3

return

PLPCC\_000000:

.asciiw "PLP"

.space 1

PLPCC\_000001:

.asciiw "Cto"

.space 1

A.4. Library Functions

A.4.1 Getchar

Getchar is used read a character inside a variable from UART.

C Code:

void main()

{

char c;

c = getchar();

}

PLP Code:

#

# LINE 5: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -4

#

# LINE 3:  c = getchar();

#

addiu $t0, $sp, 4 # \*c

push $t0

li $s0, 0xF0000000 # UART

li $s2, 0b10 # Status register bit mask

li $t0, 0 # value of current number

PLPCC\_000000:

PLPCC\_000001:

lw $s4, 4($s0) #load status register

and $s4, $s4, $s2 #mask for ready bit

bne $s4, $s2, PLPCC\_000000

nop

lw $t0, 8($s0) #load from recieve buffer

sw $s2, 0($s0) #command register: clear status

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 8 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.4.2 Putchar

Putchar is used display the value of a character variable on UART.

C Code:

int main ()

{

char ch;

for(ch = 'A' ; ch <= 'Z' ; ch++)

{

putchar(ch);

}

return(0);

}

PLP Code:

#

# LINE 10: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -4

#

# LINE 4:  for(ch = 'A' ; ch <= 'Z' ; ch++)

#

addiu $t0, $sp, 4 # \*ch

push $t0

li $t0, 'A'

pop $t1

sw $t0, 0($t1)

#

# LINE 7:  }

#

PLPCC\_000000:

lw $t0, 4($sp) # ch

push $t0

li $t0, 'Z'

pop $t1

slt $t0, $t0, $t1

nor $t0, $t0, $t0 # $t0 = !$t0

andi $t0, $t0, 1 # Mask LSB

beq $t0, $zero, PLPCC\_000001

nop

#

# LINE 6:  putchar(ch);

#

lw $t0, 4($sp) # ch

move $a0, $t0

PLPCC\_000003:

lui $t0, 0xf000 #uart base address

PLPCC\_000004:

lw  $t1, 4($t0) #get the uart status

andi $t1, $t1, 0x01 #mask for the cts bit

beq $t1, $zero, PLPCC\_000004

nop

PLPCC\_000005:

sw  $a0, 12($t0) #write the data to the output buffer

sw  $t1, 0($t0) #send the data!

move $t0, $a0

PLPCC\_000002:

addiu $t0, $sp, 4 # \*ch

lw $t1, 0($t0)

addiu $t2, $t1, 1

sw $t2, 0($t0)

move $t0, $t1

j PLPCC\_000000

nop

PLPCC\_000001:

#

# LINE 9:  return(0);

#

li $t0, 0

move $v0, $t0

addiu $sp, $sp, 8 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.4.3 Gets

Gets is used to read a string inside a variable from UART. But we need to put a semicolon at the end of our string on UART before pressing send, since semi colon marks the end of input.

C Code:

int main()

{

char str[50];

gets(str);

return(0);

}

PLP Code:

#

# LINE 5: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -200

#

# LINE 3:  gets(str);

#

move $t3, $sp

li $s0, 0xF0000000 # UART

li $s2, 0b10 # Status register bit mask

li $t7, 59 # Used to store semi colon

li $t0, 0 # value of current number

li $a0, 0x100f0000 # Memory location for storing string

PLPCC\_000004:

li $s1, 0

lw $s3, 0($a0)

bne $s1, $s3, PLPCC\_000005

nop

li $s1, 0x100f0000

beq $a0, $s1, PLPCC\_000006

nop

addiu $a0, $a0, 4

j PLPCC\_000006

nop

PLPCC\_000005:

addiu $a0, $a0, 4

j PLPCC\_000004

nop

PLPCC\_000006:

push $a0

PLPCC\_000000:

jal PLPCC\_000001

nop

# Check for end of input

bne $v0, $t7, PLPCC\_000002

nop

move $t1, $a0

j PLPCC\_000007

nop

PLPCC\_000002:

sw $v0, 0($a0)

addiu $a0, $a0, 4

j PLPCC\_000000

nop

#Description: Reads UART and returns it in $v0

# Uses s2, s4, v0

PLPCC\_000001:

PLPCC\_000003:

lw $s4, 4($s0) #load status register

and $s4, $s4, $s2 #mask for ready bit

bne $s4, $s2, PLPCC\_000001

nop

lw $v0, 8($s0) #load from recieve buffer

sw $s2, 0($s0) #command register: clear status

jr $ra

nop

PLPCC\_000007:

addiu $t0, $t3, 4 # \*str

push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  return(0);

#

li $t0, 0

move $v0, $t0

addiu $sp, $sp, 204 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.4.4 Puts

Puts is used to display a string on UART.

C Code:

int main()

{

char str[50];

str = "CtoPLP";

puts(str);

return(0);

}

PLP Code:

#

# LINE 6: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -200

#

# LINE 3:  str = "CtoPLP";

#

addiu $t0, $sp, 4 # \*str

push $t0

li $t0, PLPCC\_000000

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  puts(str);

#

lw $t0, 4($sp) # str

li $t6, 1

jal PLPCC\_000002

nop

PLPCC\_000008:

ori $a0, $zero, 0x000d #newline

jal PLPCC\_000003

nop

ori $a0, $zero, 0x000a  #linefeed

jal PLPCC\_000003

nop

j PLPCC\_000001

nop

PLPCC\_000003:

lui $t0, 0xf000 #uart base address

PLPCC\_000004:

lw  $t1, 4($t0) #get the uart status

andi $t1, $t1, 0x01 #mask for the cts bit

beq $t1, $zero, PLPCC\_000004

nop

sw  $a0, 12($t0) #write the data to the output buffer

sw  $t1, 0($t0) #send the data!

jr $31

nop

PLPCC\_000002: #we have a pointer to the string in a0, just loop and increment until we see a null

move $t9, $31 #save the return address

move $t8, $t0 #save the argument

PLPCC\_000005:

lw $a0, 0($t8) #first 1-4 characters

beq $a0, $zero, PLPCC\_000006

nop

PLPCC\_000007:

jal PLPCC\_000003 #write this byte

nop

j PLPCC\_000005

addiu $t8, $t8, 4 #increment for the next word

PLPCC\_000006:

jr $t9 #go home

nop

PLPCC\_000001:

move $t0, $t6

#

# LINE 5:  return(0);

#

li $t0, 0

move $v0, $t0

addiu $sp, $sp, 204 # Epilogue

move $sp, $s7

move $ra, $a3

return

PLPCC\_000000:

.asciiw "CtoPLP"

.space 1

A.4.5 plp\_scanint

This keyword is used to read integers from UART.

Syntax: plp\_scanint (varname)

Where varname is of type int.

Example: plp\_scanint (a);

We can also have more than one arguments in plp\_scanint, provided both the variables are of type int. A semicolon marks the end of input while entering from UART.

When we have to enter more than one number on UART due to multiple arguments, we need to separate the numbers using space character on UART and end the input with semicolon before pressing send on UART. Each number is read in a separate variable given as an argument inside plp\_scanint.

Example: int a, b;

plp\_scanint (a, b);

C Code:

void main()

{

int a,i;

plp\_scanint(i,a);

}

PLP Code:

#

# LINE 4: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -8

#

# LINE 3:  plp\_scanint(i,a);

#

move $t3, $sp

li $s0, 0xF0000000 # UART

li $s2, 0b10 # Status register bit mask

li $t5, 10 # Used to convert decimal to binary

li $t6, 48 # Used to convert ASCII to decimal

li $t7, 59 # Used to store semi colon

li $t8, 32 # space character

li $t0, 0 # value of current number

PLPCC\_000001:

jal PLPCC\_000002

nop

# Check for end of input

bne $v0, $t7, PLPCC\_000003

nop

push $t0

move $v0, $0

j PLPCC\_000000

nop

PLPCC\_000003:

# Check for space indicating end of number

bne $v0, $t8, PLPCC\_000004

nop

#code for initializing variable

push $t0

move $t0, $0 #clear input

j PLPCC\_000001

nop

# Assume value is a number, convert to binary

PLPCC\_000004:

mullo $t0, $t0, $t5 #multiply current number by 10

subu $v0, $v0, $t6 #convert UART character from ascii to decimal

addu $t0, $t0, $v0

j PLPCC\_000001

#Description: Reads UART and returns it in $v0

# Uses s2, s4, v0

PLPCC\_000002:

PLPCC\_000005:

lw $s4, 4($s0) #load status register

and $s4, $s4, $s2 #mask for ready bit

bne $s4, $s2, PLPCC\_000002

nop

lw $v0, 8($s0) #load from recieve buffer

sw $s2, 0($s0) #command register: clear status

jr $ra

nop

PLPCC\_000000:

pop $t1

addiu $t0, $t3, 8 # \*a

push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

pop $t1

addiu $t0, $t3, 4 # \*i

push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 12 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.4.6 plp\_scanchar

This keyword is used to read characters from UART.

Syntax: plp\_ scanchar (varname)

Where varname is of type char.

Example: plp\_ scanchar (a);

We can also have more than one arguments in plp\_ scanchar, provided both the variables are of type char. A semicolon marks the end of input while entering from UART.

When we have to enter more than one characters on UART due to multiple arguments, we need to enter all the characters one after another and end the input with semicolon before pressing send on UART. Each character is read in a separate variable given as an argument inside plp\_ scanint.

Example: char a, b;

plp\_ scanchar (a, b);

C Code:

void main()

{

char a,i;

plp\_scanchar(i,a);

}

PLP Code:

#

# LINE 4: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -8

#

# LINE 3:  plp\_scanchar(i,a);

#

move $t3, $sp

li $s0, 0xF0000000 # UART

li $s2, 0b10 # Status register bit mask

li $t5, 10 # Used to convert decimal to binary

li $t6, 48 # Used to convert ASCII to decimal

li $t7, 59 # Used to store semi colon

li $t8, 32 # space character

li $t0, 0 # value of current number

PLPCC\_000001:

jal PLPCC\_000002

nop

# Check for end of input

bne $v0, $t7, PLPCC\_000003

nop

move $v0, $0

j PLPCC\_000000

nop

PLPCC\_000003:

push $v0

j PLPCC\_000001

nop

#Description: Reads UART and returns it in $v0

# Uses s2, s4, v0

PLPCC\_000002:

PLPCC\_000004:

lw $s4, 4($s0) #load status register

and $s4, $s4, $s2 #mask for ready bit

bne $s4, $s2, PLPCC\_000002

nop

lw $v0, 8($s0) #load from recieve buffer

sw $s2, 0($s0) #command register: clear status

jr $ra

nop

PLPCC\_000000:

pop $t1

addiu $t0, $t3, 8 # \*a

push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

pop $t1

addiu $t0, $t3, 4 # \*i

push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 12 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.4.7 plp\_scanstring

This keyword is used to read strings on UART.

Syntax: plp\_ scanstring (varname)

Where varname is an array of type char.

Example: plp\_ scanstring (a);

We can also have more than one arguments in plp\_ scanstring, provided both the variables are array of type char.

A semicolon marks the end of each string while entering from UART. When we enter more than one string on UART due to multiple arguments given in plp\_ scanstring , we end each string with a semi colon before pressing send on UART.

Example: char a[20], b[20];

plp\_ scanstring (a, b);

C Code:

void main()

{

char a[20],b[20];

plp\_scanstring(a,b);

}

PLP Code:

#

# LINE 4: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -160

#

# LINE 3:  plp\_scanstring(a,b);

#

move $t3, $sp

li $s0, 0xF0000000 # UART

li $s2, 0b10 # Status register bit mask

li $t5, 10 # Used to convert decimal to binary

li $t6, 48 # Used to convert ASCII to decimal

li $t7, 59 # Used to store semi colon

li $t8, 32 # space character

li $t0, 0 # value of current number

li $t4, 2         # number of arguments

li $a0, 0x100f0000 # Memory location for storing string

PLPCC\_000005:

li $s1, 0

lw $s3, 0($a0)

bne $s1, $s3, PLPCC\_000006

nop

li $s1, 0x100f0000

beq $a0, $s1, PLPCC\_000007

nop

addiu $a0, $a0, 4

j PLPCC\_000007

nop

PLPCC\_000006:

addiu $a0, $a0, 4

j PLPCC\_000005

nop

PLPCC\_000007:

push $a0

PLPCC\_000001:

jal PLPCC\_000002

nop

# Check for end of input

bne $v0, $t7, PLPCC\_000003

nop

addiu $t4, $t4, -1

addiu $a0, $a0, 4

beq $t4, $0, PLPCC\_000000

nop

push $a0

j PLPCC\_000001

nop

PLPCC\_000003:

sw $v0, 0($a0)

addiu $a0, $a0, 4

j PLPCC\_000001

nop

#Description: Reads UART and returns it in $v0

# Uses s2, s4, v0

PLPCC\_000002:

PLPCC\_000004:

lw $s4, 4($s0) #load status register

and $s4, $s4, $s2 #mask for ready bit

bne $s4, $s2, PLPCC\_000002

nop

lw $v0, 8($s0) #load from recieve buffer

sw $s2, 0($s0) #command register: clear status

jr $ra

nop

PLPCC\_000000:

pop $t1

addiu $t0, $t3, 4 # \*b

push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

pop $t1

addiu $t0, $t3, 84 # \*a

push $t0

move $t0, $t1

pop $t1

sw $t0, 0($t1)

addiu $sp, $sp, 164 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.4.8 plp\_printint

This keyword is used to print integers on UART.

Syntax: plp\_ printint (varname)

Where varname is a variable of type int.

Example: plp\_ printint (a);

We can also have more than one arguments in plp\_ printint, provided both the variables are of type int.

Example: int a, b;

plp\_ printint (a, b);

C Code:

void main()

{

int i;

for(i=1;i<=5;i++){

plp\_printint(i);

}

}

PLP Code:

#

# LINE 6: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -4

#

# LINE 3:  for(i=1;i<=5;i++){

#

addiu $t0, $sp, 4 # \*i

push $t0

li $t0, 1

pop $t1

sw $t0, 0($t1)

#

# LINE 5:  }

#

PLPCC\_000000:

lw $t0, 4($sp) # i

push $t0

li $t0, 5

pop $t1

slt $t0, $t0, $t1

nor $t0, $t0, $t0 # $t0 = !$t0

andi $t0, $t0, 1 # Mask LSB

beq $t0, $zero, PLPCC\_000001

nop

#

# LINE 4:  plp\_printint(i);

#

move $v1, $ra

li $0, 0

li $t6, 0

lw $t0, 4($sp) # i

slt $t3, $t0, $0

beq $t3, $0, PLPCC\_000015

nop

li $t6, 1 #negative number flag

li $t1, -1

mullo $t0, $t0, $t1

PLPCC\_000015:

li $t1, 10

li $t5, 0 #counter

jal PLPCC\_000004

nop

j PLPCC\_000003

nop

PLPCC\_000004:

li $t4, 0

li $v0, 0 # return value

li $t3, 0 # align shift count

PLPCC\_000005:

beq $t1, $t0, PLPCC\_000006

nop

slt $t2, $t1, $t0 #t1 < t0

beq $t2, $0, PLPCC\_000006

nop

sll $t1, $t1, 1

addiu $t3, $t3, 1

j PLPCC\_000005

nop

PLPCC\_000006:

slt $t2, $t1, $t0 #t0 > t1

beq $t2, $0, PLPCC\_000007

nop

subu $t0, $t0, $t1

PLPCC\_000007:

bne $t1, $t0, PLPCC\_000008

nop

addiu $t2, $t2, 1

subu $t0, $t0, $t1

PLPCC\_000008:

nop

sll $v0, $v0, 1 #shift result left

addu $v0, $v0, $t2

beq $t4, $t3, PLPCC\_000009

nop

srl $t1, $t1, 1

addiu $t4, $t4, 1

j PLPCC\_000006

nop

PLPCC\_000009:

addiu $t0, $t0, 48 #adding ascii value

push $t0

move $t0, $v0

addiu $t5, $t5 , 1

li $t1, 10

slt $t3, $t0, $t1

beq $t3, $zero, PLPCC\_000004

nop

PLPCC\_000010:

addiu $t0, $t0, 48

push $t0

addiu $t5, $t5 , 1

beq $t6 ,$0, PLPCC\_000011

nop

li $t0, 45

push $t0

addiu $t5, $t5 , 1

PLPCC\_000011:

lui $t0, 0xf000 #uart base address

PLPCC\_000012:

lw  $t1, 4($t0) #get the uart status

andi $t1, $t1, 0x01 #mask for the cts bit

beq $t1, $zero, PLPCC\_000012

nop

PLPCC\_000013:

pop $a0

sw  $a0, 12($t0) #write the data to the output buffer

sw  $t1, 0($t0) #send the data!

addiu $t5, $t5, -1 #decrement counter

bne $t5, $zero, PLPCC\_000013

nop

jr $ra

nop

PLPCC\_000003:

move $ra, $v1

PLPCC\_000002:

addiu $t0, $sp, 4 # \*i

lw $t1, 0($t0)

addiu $t2, $t1, 1

sw $t2, 0($t0)

move $t0, $t1

j PLPCC\_000000

nop

PLPCC\_000001:

addiu $sp, $sp, 8 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.4.9 plp\_printchar

This keyword is used to print characters on UART.

Syntax: plp\_ printchar (varname)

Where varname is a variable of type char.

Example: plp\_ printchar (ch);

We can also have more than one arguments in plp\_ printchar, provided both the variables are of type char.

Example: char a, b;

plp\_ printchar (a, b);

C Code:

void main()

{

char a,b;

a='a';

b='b';

plp\_printchar(a,b);

}

PLP Code:

#

# LINE 6: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -8

#

# LINE 3:  a='a';

#

addiu $t0, $sp, 8 # \*a

push $t0

li $t0, 'a'

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  b='b';

#

addiu $t0, $sp, 4 # \*b

push $t0

li $t0, 'b'

pop $t1

sw $t0, 0($t1)

#

# LINE 5:  plp\_printchar(a,b);

#

lw $t0, 8($sp) # a

move $a0, $t0

jal PLPCC\_000001

nop

lw $t0, 4($sp) # b

move $a0, $t0

jal PLPCC\_000001

nop

j PLPCC\_000000

nop

PLPCC\_000001:

lui $t0, 0xf000 #uart base address

PLPCC\_000002:

lw  $t1, 4($t0) #get the uart status

andi $t1, $t1, 0x01 #mask for the cts bit

beq $t1, $zero, PLPCC\_000002

nop

PLPCC\_000003:

sw  $a0, 12($t0) #write the data to the output buffer

sw  $t1, 0($t0) #send the data!

jr $ra

nop

PLPCC\_000000:

addiu $sp, $sp, 12 # Epilogue

move $sp, $s7

move $ra, $a3

return

A.4.10 plp\_printstring

This keyword is used to print strings on UART.

Syntax: plp\_printstring(varname)

Where varname is an array of type char.

Example: plp\_printstring(arr);

We can also have more than one arguments in plp\_printstring, provided both are character arrays or as we may call it, strings.

Example: char a[20], b[20]

plp\_printstring(a, b);

Also, we can directly write a string in double quotes without storing in a variable and provide it as an argument to plp\_printstring

Example: plp\_printstring(“C to PLP”);

After printing on UART, plp\_printstring inserts a new line on UART.

C Code:

void main()

{

char a[20],b[20];

a="PLP";

b="toC";

plp\_printstring(a,b);

}

PLP Code:

#

# LINE 6: }

#

main:

move $a3, $ra

move $s7, $sp

addiu $sp, $sp, -160

#

# LINE 3:  a="PLP";

#

addiu $t0, $sp, 84 # \*a

push $t0

li $t0, PLPCC\_000000

pop $t1

sw $t0, 0($t1)

#

# LINE 4:  b="toC";

#

addiu $t0, $sp, 4 # \*b

push $t0

li $t0, PLPCC\_000001

pop $t1

sw $t0, 0($t1)

#

# LINE 5:  plp\_printstring(a,b);

#

move $v1, $ra

li $0, 0

lw $t0, 84($sp) # a

jal PLPCC\_000003

nop

lw $t0, 4($sp) # b

jal PLPCC\_000003

nop

PLPCC\_000009:

ori $a0, $zero, 0x000d #newline

jal PLPCC\_000004

nop

ori $a0, $zero, 0x000a  #linefeed

jal PLPCC\_000004

nop

j PLPCC\_000002

nop

PLPCC\_000004:

lui $t0, 0xf000 #uart base address

PLPCC\_000005:

lw  $t1, 4($t0) #get the uart status

andi $t1, $t1, 0x01 #mask for the cts bit

beq $t1, $zero, PLPCC\_000005

nop

sw  $a0, 12($t0) #write the data to the output buffer

sw  $t1, 0($t0) #send the data!

jr $31

nop

PLPCC\_000003: #we have a pointer to the string in a0, just loop and increment until we see a null

move $t9, $31 #save the return address

move $t8, $t0 #save the argument

PLPCC\_000006:

lw $a0, 0($t8) #first 1-4 characters

beq $a0, $zero, PLPCC\_000007

nop

PLPCC\_000008:

jal PLPCC\_000004 #write this byte

nop

j PLPCC\_000006

addiu $t8, $t8, 4 #increment for the next word

PLPCC\_000007:

jr $t9 #go home

nop

PLPCC\_000002:

move $ra, $v1

addiu $sp, $sp, 164 # Epilogue

move $sp, $s7

move $ra, $a3

return

PLPCC\_000000:

.asciiw "PLP"

.space 1

PLPCC\_000001:

.asciiw "toC"

.space 1