Josh A. Bosley
BLang Overview, and
Documentation

BL
V.0

BL (Bosley's Language) is a pseudo-assembly language developed to simulate the step-by-step instruction nature of assembly programming. The purpose of the BL project is to provide an outlet for putting a multitude of programming concepts into practice.

The BLang Engine:

Overview:

The engine is modeled after a 'system-on-a-chip.' That is to say, it is logically structured after a computing system that has multiple hardware components engineered to operate together on a specific chip. The engine contains operational memory, a central processing unit, and registers for computation. While each piece of hardware is not logically self-contained, their functionality is emulated via BL functions.

Registers:

The BLang engine has 60 registers for use by the programmer. r0 -> r59 These registers are for use by the programmer, and are never used for calculations, or operations of the engine. That means that the programmer never has to worry about which registers might be overwritten by an operation.

Memory:

The memory module is globally accessible, and contains no information regarding the running of the current program. This means that program instructions cannot be corrupted by a user's memory alteration operations. The memory module is expandable, and will grow to the size of any positive index called. For instance: When a program initiates, the memory size is 0. If the engine detects the use of a memory spot larger than its current size, it will grow to the called size prior to storing the data to the called index. The memory module's max size is dependent upon the host machine.

Instruction Loading:

The instruction to be operated is loaded onto the engine via the syntax parser. When a line is read in from the .basm file, it is broken-up by instruction type and is added to the engine differently depending on the instruction. All instructions must be within a function, and all programs must contain at least one function named 'main,' which will serve as the entry point.

System cycle:

Each function when parsed is turned into a function-frame structure, and is added to the engines function memory, hereafter referred to as frame-memory. Upon initialization the engine will locate the entry point function 'main' within its frame-memory, and add it to the frame-stack. Once the function is fully loaded into the stack, the instructions within the frame will begin to be executed sequentially. In the event of a jump, or branch, the engine's instruction counter (IC) will move to the label within the current frame (CF) and continue sequential execution. In the event of a call instruction the function being called is found within frame-memory and added to the frame-stack. The engine will then begin executing that function's instructions until complete, and pop that frame of the frame-stack. The engine will then return to the caller function at the index immediately after the call instruction.

Program Example:

```
<moot
        p “Function moot!\n”        // Indicate we are in function moot
        p m10                       // Print item from mem at index 10
        p “\n”                      // Print \n character
>
<main
        mov r1 $1                   // Store 1 in register 1
        mov r0 $5                   // Store 5 in register 0
        @loop                       // Create label to jump, or branch to later
                p “Loop  : “        // Print a string
                p r1                // Print the contents of register 1
                p “\n”              // Print \n character
                add r1 r1 $1        // Increment r1 by 1
                bne r0 r1 @loop     // If (r0 != r1): GOTO @loop;

        mov m10 r1                  // First mem call, so mem is expanded to
                                    // size of 11 (0-10) and r1 moved to m10
        call moot                   // Function call to moot
        p “All done.”               // Print to signal completion
>
```

Program Output:

```
Loop :1
Loop :2
Loop :3
Loop :4
Loop :5
Function moot!
5
All done.
```

Instruction Manual Key:

Each instruction requires from 1 – 3 parameters depending on the instruction operation. Each parameter is referred to as a 'slot' and labeled sequentially as: s1 s2 s3. Each slot can be a float, integer, register, memory index, label, function name, or string depending on the instruction. The following will be used to list indicate what is accepted:

Integer:	I	**Memory Index**:	M
Float:	F	**Register Index**:	R
String	S	**Label**	L
Unused	U	**Function Name**	FN

Instruction Slot Example:

ble **r0 r0 @top** = instruction **s1 s2 s3**

To make instruction manual easy to read, it has been moved to the next page

INSTRUCTION	ACCEPTED	DESCRIPTION
add	S1: R M S2: R M I F S S3: R M I F S	Add s2 and s3, store in s1
ble	S1: R M I S2: R M I S3: L	Branch to label if s1, is less than or equal to s2
bne	S1: R M I S2: R M I S3: L	Branch to label if s1, and s2 are not equal
bgt	S1: R M I S2: R M I S3: L	Branch to label if s1 is greater than s2
blt	S1: R M I S2: R M I S3: L	Branch to label if s1 is less than s2
beq	S1: R M I S2: R M I S3: L	Branch to label if s1 and s2 are equal
bge	S1: R M I S2: R M I S3: L	Branch to label if s1 is greater than / equal to s2
call	S1: FN S2: UN S3: UN	Call function s1
div	S1: R M S2: R M I F S3: R M I F	Divide s2 by s3, store in s1
jmp	S1: L S2: UN S3: UN	Jump to label s1
mov	S1: R M S2: R M I F S3: UN	Put s2 into s1. Doesn't delete s2

mult	S1: R M S2: R M I F S3: R M I F	Multiply s2 and s3, store in s1
p	S1: S R M I F S2: UN S3: UN	Print s1
sub	S1: R M S2: R M I F S3: R M I F	Sub s2 by s3, store in s1
r	S1 : R M S2: UN S3: UN	Read value from user

File Output Instructions (Different than other instructions):

f	S1 : R M S2 : (a || w) S3 : R M	Filename/location Output mode (a / w) Data source

Example File Output :

```
<main
        p “Please enter a filename to write to : “
        r r1
        p “\n”
        p “What would you like to write to file ? “
        r r2
        p “\n”
        f r1 w r2    // Write r2 out to r1 in w mode (Erases pre-existing file)
>
```