abCore16 C-like SSL: User Guide

Version 1.0 (Reflecting implementation up to arrays)

1. Introduction

Welcome to the abCore16 C-like Simple Source Language (SSL). This language is designed to be compiled into Simple Assembly Language (SAL) for the abCore16, a 16-bit microprocessor simulated in Python. It provides a higher-level, more readable way to write programs compared to raw assembly, incorporating familiar C-like syntax for variables, expressions, control flow, and functions.

This guide will walk you through the features of the language, its syntax, and how to write programs for the abCore16 system.

2. Getting Started: File Extension

C-like SSL source files should use the .ssl extension (e.g., my_program.ssl). The toolchain's main.py script will automatically recognize this extension and use the PLY-based compiler.

3. Basic Program Structure

A program in C-like SSL consists of a series of top-level declarations. These can be:

- **Global Variable Declarations:** Declaring variables that are accessible from anywhere in the program.
- **Function Definitions:** Defining reusable blocks of code.
- **Simple Global Statements (Limited):** While possible, it's highly recommended to place all executable logic within functions, particularly a main function.
    - Allowed global statements (outside functions): assignments, print statements, expression statements (like a function call), and empty statements.
    - **Not allowed globally:** if, while, for statements. These must be inside a function.

Recommended Structure (with a main function):

```
// Global variable declarations (optional)
var global_var1;
var global_var2 = 100;

// Function definitions (optional, can be before or after main)
```

```
func helper_function(param1) {
  print param1 + 5;
  return;
}

// Main function - program execution typically starts here
func main() {
  // Local variable declarations
  var local_x = 10;
  var local_y;

  // Statements: assignments, expressions, control flow, function calls
  local_y = local_x * 2;
  helper_function(local_y);
  print local_y;

  if (local_x < local_y) {
    print 1; // True
  } else {
    print 0; // False
  }

  var i;
  for (i = 0; i < 3; i = i + 1) {
    print i;
  }
}
// Other function definitions (optional)
```

The compiler toolchain will automatically look for a function named main and generate code to call it first if no other global executable statements are present.

4. Comments

Single-line comments start with // and extend to the end of the line.

```
// This is a single-line comment
var x = 10; // This comment is after a statement
```

Multi-line block comments (like /* ... */) are **not** currently supported.

5. Data Types and Variables

- **Data Type:** The language currently supports a single primary data type: a 16-bit signed integer. All variables and expression results are treated as such. Values range from -32768 to 32767. Unsigned operations are not explicitly distinguished at the SSL level but may occur at the SAL/machine code level for certain operations (e.g., LOADM from memory).
- **Variable Declaration:** Variables must be declared using the var keyword before they are used (though the compiler currently has a fallback to treat undeclared variables on first use in an assignment or as an operand as implicit globals, this is **not recommended** and may change).
    - Declarations can be global (outside any function) or local (inside a function block).
    - Local variables declared with var inside a function are scoped to that entire function.
    - Variables declared in a for loop initializer (for (var i = 0; ...)) are also scoped to the containing function. True block-level scoping (where a variable is only live within its nearest enclosing {...}) is not yet fully implemented for all var declarations.

```
var global_variable;     // Global, uninitialized (defaults to 0 in simulator memory)
var initialized_global = 123;

func my_function() {
   var local_a;        // Local to my_function, uninitialized
   var local_b = 45;    // Local to my_function, initialized
   local_a = 10;
  print local_a + local_b;
}
```

- **Identifiers (Variable and Function Names):**

- Must start with a letter (a-z, A-Z) or an underscore (_).
- Can be followed by letters, underscores, or digits (0-9).
- Are case-sensitive (e.g., myVar is different from myvar).
- Cannot be the same as reserved keywords.

6. Operators and Expressions

Expressions combine values, variables, and operators to produce a new value.

- **Literals:**
  - **Numbers:** Decimal integers (e.g., 10, 0, 123). Hexadecimal literals (e.g., 0xFF, 0x1A) are *not* directly supported in SSL expressions but can be used in the original SSL syntax that compiles to LOAD reg, #value. For C-like SSL, use decimal numbers.
- **Arithmetic Operators:**
  - + (addition)
  - - (subtraction, also unary minus)
  - * (multiplication)
  - Division and Modulo are not currently supported.
- **Assignment Operator:**
  - = (assignment)
  - Example: x = y + 10;
  - Assignments can be used as expressions within for loop clauses (e.g., for (i = 0; ...; i = i + 1)).
- **Comparison Operators (evaluate to 1 for true, 0 for false):**
  - == (equal to)
  - != (not equal to)
  - < (less than)
  - > (greater than)
  - <= (less than or equal to)

    - >= (greater than or equal to)

- **Logical Operators (evaluate to 1 for true, 0 for false):**
    - && (logical AND - short-circuiting)
    - || (logical OR - short-circuiting)
    - ! (logical NOT - unary)
- **Parentheses:**
    - () for grouping expressions and controlling order of operations.
    - Example: result = (a + b) * c;
- **Operator Precedence (from highest to lowest, with associativity):**

1. ! (logical NOT), - (unary minus) - Right associative
2. * (multiplication) - Left associative
3. + (addition), - (subtraction) - Left associative
4. <, >, <=, >= (comparison) - Non-associative
5. ==, != (equality) - Non-associative
6. && (logical AND) - Left associative
7. || (logical OR) - Left associative
8. = (assignment) - Right associative (currently only as a statement or in for clauses, not general expression assignment)

7. Statements

Statements are the building blocks of execution. Most statements must end with a semicolon (;).

- **Empty Statement:**
    - Just a semicolon. Does nothing.
    - ;
- **Expression Statement:**
    - An expression followed by a semicolon. The expression is evaluated for its side effects.

    - x + 1; (result is discarded, but if x was my_func(), my_func would be called)
    - my_function_call(10);
- **Assignment Statement:**
    - Assigns the value of an expression to a variable.
    - variable = expression;
    - x = 10;
    - y = x + z * 2;
- **Variable Declaration Statement:**
    - var identifier;
    - var identifier = expression;
- **Print Statement:**
    - Outputs the value of an expression to the simulator's default output.
    - print expression;
    - print 123;
    - print my_variable + 5;
- **Block Statement (Compound Statement):**
    - A sequence of zero or more statements enclosed in curly braces {}.
    - Does not require a semicolon after the closing brace.
    - Used as the body for if, else, while, for, and functions.

```
{
   var temp = x;
   x = y;
  y = temp;
  print x;
}
```

8. Control Flow

- **if-else Statement:**

```
if (condition_expression) {
  // statements if condition is true (1)
}
if (condition_expression) {
  // statements if condition is true (1)
} else {
  // statements if condition is false (0)
}
```

  - The condition_expression is evaluated. If its value is non-zero (true), the first block is executed.
  - If an else clause is present and the condition is zero (false), the else block is executed.
  - Braces {} are required for the statement blocks, even for single statements.

- **while Statement:**

```
while (condition_expression) {
  // statements to repeat as long as condition is true
}
```

  - The condition_expression is evaluated *before* each iteration.
  - If non-zero (true), the block of statements is executed.
  - This process repeats until the condition becomes zero (false).
  - Braces {} are required for the loop body.

- **for Statement:**

```
for (initializer_clause; condition_clause; update_clause) {
// statements to repeat
}
```

- **initializer_clause**: Executed once before the loop begins. Can be:
  - A variable declaration and initialization: var i = 0
  - An assignment: i = 0 (if i was declared earlier)
  - Any other expression (evaluated for side effects).
  - Empty.
- **condition_clause**: An expression evaluated *before* each iteration. If non-zero (true), the loop body executes. If zero (false), the loop terminates. If empty, it's considered always true (infinite loop unless broken by other means).
- **update_clause**: An expression executed *after* each iteration of the loop body, before the condition is re-evaluated.
- **Body**: A block statement {...}.
- **Execution Flow:**

1. initializer_clause runs.
2. condition_clause is evaluated.
3. If false, loop terminates.
4. If true, body executes.
5. update_clause runs.
6. Go back to step 2.

- Example:

```
var sum = 0;
for (var i = 1; i <= 10; i = i + 1) {
  sum = sum + i;
}
print sum; // Prints 55
```

9. Functions

Functions are defined blocks of code that can be called by name.

- **Definition:**

```
func function_name(parameter1, parameter2, ...) {
  // statements (function body)
  // optional: return expression;
  // optional: return; (for functions not returning a value)
}
```

  - func keyword starts a function definition.
  - function_name is an identifier.
  - Parameters are identifiers separated by commas. Currently, all parameters are passed by value and are treated as 16-bit integers.
  - The function body is a block statement {...}.

- **Calling Functions:**

```
function_name(argument1, argument2, ...); // As a statement
my_variable = function_name(arg1, arg2) + 10; // As part of an expression
```

  - Arguments are expressions.
  - The number of arguments in a call must match the number of parameters in the function definition.

- **return Statement:**
  - return expression;: Exits the current function and returns the value of expression to the caller. The returned value is placed in register R0 by convention.
  - return;: Exits the current function without returning a specific value (like a void function).
  - If a function reaches the end of its body without encountering a return statement, it will implicitly return (the value in R0 will be whatever it was).

- **Example:**

```
func add(a, b) {
  return a + b;
```

```
}

func main() {
  var result;
  result = add(10, 25); // result will be 35
  print result;
}
```

10. Scope of Variables

- **Global Variables:** Declared outside any function with var. Accessible from any part of the program after their declaration.
- **Function Parameters:** Treated as local variables within the function, initialized with the values of the arguments passed during the function call.
- **Local Variables:**
    - Declared inside a function (or block) with var.
    - **Current Implementation Detail:** Variables declared with var inside a function (including those in for loop initializers like for (var i=0;...)) are currently scoped to the *entire function*. They are allocated on the function's stack frame when the function is called.
    - True C-style block scoping (where a variable declared in {...} or a for header is only visible within that specific block/loop) is a planned future enhancement. For now, avoid re-declaring a variable name with var within the same function, even in different blocks or loops, as it will refer to the same function-level stack slot.

11. Standard Library / Built-in Functions

Currently, there are no extensive built-in library functions other than print (which compiles to an OUT instruction).

12. Compilation and Execution

1. Write your C-like SSL code in a file with a .ssl extension.
2. Use main.py from the abCore16 toolchain to process it:

```
python main.py your_program.ssl
```

3. This will:
    - Compile your .ssl file to an intermediate Simple Assembly Language (SAL) file (.sal).
    - Assemble the SAL file into a binary machine code file (.bin).
    - (Optionally) Disassemble the binary back to SAL for verification (_disassembled.sal).
    - Run the binary file on the abCore16 simulator.
    - Output from print statements will appear on the console, prefixed with SIM MMIO OUTPUT.

13. Example Program (Factorial)

```
// factorial.ssl
// Calculates factorial of a number

func factorial(n) {
  var result;
  if (n <= 1) {
    result = 1;
  } else {
    result = n * factorial(n - 1); // Recursive call
  }
  return result;
}


func main() {
  var num = 5; // Calculate factorial of 5
  var fact_result;

  fact_result = factorial(num);
  print num;     // Output: 5
  print fact_result; // Output: 120 (5! = 120)
}
```

14. Current Limitations and Future Work

- Single 16-bit integer data type.
- No arrays, structs, or pointers.
- No division or modulo operators.
- Limited block scoping for variables (currently function-scoped for var declarations within functions).
- No break or continue for loops yet.
- Minimal standard library.

This guide provides a snapshot of the language. As the abCore16 project evolves, new features and refinements will be added.