# Micro C Language & Jasmin Instructions

**Note:** This is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.
The information contained within this document is strictly for reference. If you wish to apply ideas contained in this document, you are taking full responsibility for your actions.

# Contents

# The Jasmin Assembly Language

## 1 Assembly Statements

A Jasmin assembly language program consists of statements, one per line. A statement can be an instruction (and any operands) or a directive (and any operands). There can also be a statement label on a separate line.

### 1.1 Instructions

A Jasmin instruction statement consists of an operation mnemonic (a reserved Jasmin word that represents a single operation code), or *opcode*, in the class file. Many instructions have no operands (any needed values are at the top of the operand stack), and other instruction statements include one, two, or three explicit operands after the operation mnemonic. An example of an instruction that has no explicit operands but requires two implicit operand values on the operand stack:

```
IADD
```

The integer add instruction pops off two integer values at the top of the operand stack, adds them, and pushes the integer sum onto the operand stack. The following store instruction pops off the floating-point value at the top of the stack and stores it into the local variable whose value is in slot 5 of the local variables array. It requires the explicit operand 5.

```
FSTORE 5
```

The instruction

```
getstatic MCClass/d I
```

has two explicit operands separated by a blank. It pushes the integer value of the static field *d* of class *MCClass* onto the operand stack. It does not require any operand values on the operand stack. The second explicit operand, `I`, is the *type descriptor* for a scalar integer.
The last example above illustrates a couple of Jasmin features. The operand mnemonic is not case sensitive, so you can use `GETSTATIC` or `getstatic`. Whereas Java uses the period . to separate components of a name, as in *MCClass.d*, Jasmin uses the slash /.

### 1.2 Directives

Directive statements provide information to the Jasmin assembler that may affect the contents of the class file. A directive always begins with a period immediately followed by the reserved directive word. For example, the `.method` directive starts a Jasmin method, and the `.end` directive ends it. Directive statements can also include various keywords and operands.
For example, the main method of a Java program always has the form

```
public static void main(String args[])
```

The equivalent Jasmin directive is

```
.method public static main([Ljava/lang/String;)V
```

where `public` and `static` are keywords and `main([Ljava/lang/String;)V` is the signature of the main method and its return type. A method's signature consists of the method's name and an encoding of its formal parameter types. As explained further below, `[Ljava/lang/String;` is the type descriptor for ärray of `String`s. The `V` at the end means the method returns `void` (nothing).

### 1.3 Statement Labels

A Jasmin statement label appears on a separate line and consists of an identifier followed by a colon. An instruction statement preceded by a statement label can be the target of a branch instruction, such as `GOTO`. An example of a label is:

```
Label0:
```

### 1.4 Type Descriptors

Many Jasmin instructions have explicit operands that specify data types. These instructions use Jasmin type descriptors. The type descriptors for the scalar types are:

| Java scalar type | Jasmin type descriptor |
|---|---|
| int | I |
| float | F |
| boolean | Z |
| char | C |

The type descriptor `V` represents `void`.
The type descriptor of a class consists of the upper-case letter L followed immediately by the fully qualified class name (using / instead of .) followed immediately by a semicolon ;.
Some examples:

| Java class | Jasmin type descriptor |
|---|---|
| java.lang.String | Ljava/lang/String; |
| java.util.HashMap | Ljava/util/HashMap; |
| MC | LMC; |

A type descriptor for an array type is prefaced by a left bracket `[` for each array dimension. Examples:

| Java class | Jasmin type descriptor |
|---|---|
| java.lang.String[] | [Ljava/lang/String; |
| int[][][] | [[[I |
| MC[][] | [[LMC; |

Another example: A call to the static `format()` method of the Java `String` class has two parameters, the format string and an array of object values, and it returns a string value.

```
String.format("The square root of %4d is %8.4f", n, root)
```

This compiles to the Jasmin instruction

```
invokestatic java/lang/String/format(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
```

This instruction requires that the address of the string constant `"The square root of %4d is %8.4f"` and the address of the array containing the values of `n` and `root` be at the top of the operand stack. The instruction pops off the string and array addresses to be used as parameters of the call, and the call pushes the string value result onto the operand stack.

## 2 Program Structure

The compiler will compile a MC program into a Jasmin class. The main body becomes the main method, and each global variable becomes a static field of the class. Each function declaration becomes a static method of the class.

### 2.1 Programs and Fields

The MC compiler will compile the MC program and generate code similar to what the Java compiler generates when compiling the Java class. The generated object code includes the `<init>` method which is the default class constructor. Each function declaration is compiled into a method of the class, and each global variable is compiled into a static field of the class.

### 2.2 Methods

The MC compiler will compile each MC function declaration as the Java compiler would compile a static method of a class. Jasmin methods have local variables, such as formal parameter and declared variable. Local variable values are kept in slots of the local variables array in the method's stack frame. The `.var` directive indicates the slot number, name, and data type of each local variable. This directive is optional but is useful for debugging.

### 2.3 Sizes of the Local Variables Array and the Operand Stack

The MC compiler must keep track of the maximum sizes of the local variables array and of the operand stack as it compiles each function. At the end of compiling a MC function declaration as a Jasmin method, the MC compiler must generate `.limit locals` and `.limit stack` directives in the Jasmin assembly object code. For example:

```
.limit locals 2
.limit stack 7
```

## 3 Emitting Instructions

The list below represents almost all the Jasmin operation mnemonics that the MC compiler can emit as it generates instruction statements.
Jasmin has other instructions that the compiler won't emit, including instructions that work with char, long and double data types.

```
Load constant
ICONST_0, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5, ICONST_M1,
FCONST_0, FCONST_1, FCONST_2, ACONST_NULL,
BIPUSH, SIPUSH, LDC

Load value or address
ILOAD_0, ILOAD_1, ILOAD_2, ILOAD_3,
FLOAD_0, FLOAD_1, FLOAD_2, FLOAD_3,
ALOAD_0, ALOAD_1, ALOAD_2, ALOAD_3,
ILOAD, FLOAD, ALOAD,
GETSTATIC

Store value or address
ISTORE_0, ISTORE_1, ISTORE_2, ISTORE_3,
FSTORE_0, FSTORE_1, FSTORE_2, FSTORE_3,
ASTORE_0, ASTORE_1, ASTORE_2, ASTORE_3,
ISTORE, FSTORE, ASTORE,
PUTSTATIC

Arithmetic and logical
IADD, FADD, ISUB, FSUB, IMUL, FMUL,
IDIV, FDIV, IREM, INEG, FNEG, IINC

Type conversion
I2F

Compare and branch
IFEQ, IFNE, IFLT, IFLE, IFGT, IFGE,
IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT, IF_ICMPLE, IF_ICMPGT, IF_ICMPGE,
FCMPL, FCMPG, GOTO

Arrays
NEWARRAY, ANEWARRAY,
IALOAD, FALOAD, BALOAD, AALOAD,
IASTORE, FASTORE, BASTORE, AASTORE
```

```
Call and return
INVOKESTATIC, INVOKESPECIAL,
RETURN, IRETURN, FRETURN, ARETURN

Operand stack
POP, SWAP, DUP
```

# 4 Load and Store Instructions

Jasmin has instructions to load (push a value onto the operand stack) an integer, floating-point, or string constant. It has instructions to load a value from a local variable, field, or array element. It has instructions to store (pop off a value from the operand stack to save it) a value into a local variable, field, or array element.

## 4.1 Load Constant

The basic instruction to load a constant is `LDC`, which has one explicit operand. Examples:

```
ldc 123
ldc 3.14
ldc "This is a Micro C program."
```

Note that when a string constant is loaded, as in the preceding example, the address of the string constant is pushed onto the operand stack.
Jasmin has several "shortcut" instructions that load small integer values: `ICONST_0, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5` and `ICONST_M1` load the values $0, 1, 2, 3, 4, 5$ and $-1$, respectively. Each instruction has no explicit operand and is shorter (and likely faster) than the equivalent `LDC` instruction. Similarly, the instructions `FCONST_0, FCONST_1, FCONST_2` load the small floating-point values $0, 1$, and 2, respectively. The instruction `BIPUSH` loads an integer value from $-128$ up to and including 127, and the instruction `SIPUSH` loads an integer value from $-32,768$ up to and including $32,767$. Each has a single explicit operand. Examples:

```
bipush 14
sipush 1024
```

The instruction `ACONST_NULL` has no explicit operand and loads the *null* value.

## 4.2 Load Local Value

A local value of a Jasmin method is kept in the local variables array of the method's stack frame. The basic instructions to load values are `ILOAD, FLOAD, and ALOAD` to load an integer, floating-point, and address value, respectively. Each has one explicit operand, the index number of the value's slot in the local variables array of the active stack frame. The `ILOAD` instruction also loads a boolean value. Examples:

```
iload 7
fload 12
aload 4
```

Note that in each instruction above, the operand is the slot number of the value to load from the local variables array. There are shortcut instructions to load integer, floating-point, and address values from slots $0, 1, 2$, and 3: `ILOAD_0, ILOAD_1, ILOAD_2, ILOAD_3, FLOAD_0, FLOAD_1, FLOAD_2, FLOAD_3, ALOAD_0, ALOAD_1, ALOAD_2` and `ALOAD_3`. Since the slot number is "built into" the instruction mnemonic, each instruction has no explicit operands.

## 4.3 Load Variable

Jasmin instruction `GETSTATIC` loads the value of a static field of a class. (Recall that the MC compiler will generate a static field for each global variable.) It requires two explicit operands: the fully qualified name of the field (with a slash /as name separator) and the type descriptor of the field. Examples:

```
getstatic MCClass/a I
getstatic java/lang/System/out Ljava/io/PrintWriter;
```

### 4.4 Load Array Element

Jasmin instructions `IALOAD, FALOAD, BALOAD` and `AALOAD` load an integer, floating-point, boolean, character, and address value, respectively, from an array element. Each has no explicit operands, but requires two operand values on top of the operand stack, the address of the array and the integer value of the element index.

### 4.5 Store Local Value

Jasmin's `ISTORE, FSTORE` and `ASTORE` instructions pop an integer, floating-point, and address value, respectively, off the operand stack and store it into a slot of the local variables array of the active stack frame. Each instruction has one explicit operand, the slot number, and of course each one also expects a value of the appropriate type on top of the operand stack. The `ISTORE` instruction also stores a boolean or character value. Examples:

```
istore 3
fstore 12
astore 7
```

As with the load instructions, there are the shortcut instructions to store integer, floating-point, and address values into slots $0, 1, 2$, and 3: `ISTORE_0, ISTORE_1, ISTORE_2, ISTORE_3, FSTORE_0, FSTORE_1, FSTORE_2, FSTORE_3, ASTORE_0, ASTORE_1, ASTORE_2` and `ASTORE_3`. Each one has no explicit operands but expects a value of the appropriate type on top of the operand stack.

### 4.6 Store Field

Jasmin instructions `PUTSTATIC` store a value from the operand stack into a static field. It has two explicit operands: the fully qualified field name and the type descriptor. Examples:

```
putstatic MCClass/check Z
putstatic MCClass/buffer [Ljava/lang/String;
putstatic MCClass/arrayint [I
```

### 4.7 Store Array Element

Jasmin instructions `IASTORE, FASTORE, BASTORE` and `AASTORE` store an integer, floating-point, boolean, and address value, respectively, into an array element. Each has no explicit operands, but requires three operand values on top of the operand stack: the address of the array, the integer value of the element index, and a value of the appropriate type to store.

## 5 Data Manipulation Instructions

Jasmin has numerous instructions that manipulate data values that are on top of the operand stack.

### 5.1 Pop, Swap, and Duplicate

The `POP` instruction pops off and discards the value at the top of the operand stack. The `SWAP` instruction exchanges the top two values of the operand stack. The `DUP` instruction makes a copy of the value at the top of the operand stack and then pushes the copy onto the stack.

### 5.2 Arithmetic

Jasmin instruction `IADD` has no explicit operands but requires two integer values at the top of the operand stack. It pops off the two values, adds them, and pushes their sum onto the stack. `IMUL` is similar except that it multiplies the values and pushes their product. `ISUB` subtracts the topmost value on the operand stack from the value beneath it and pushes their difference. `IDIV` performs integer division by dividing the topmost value on the operand stack into the value beneath it and pushes the truncated integer quotient. `IREM` performs integer division but pushes the remainder instead.
Corresponding instructions for floating-point values are `FADD, FSUB, FMUL, FDIV` and `FREM`. Instruction `FDIV` performs a floating-point division to generate a floating-point quotient.
Instructions `INEG` and `FNEG` negate the integer or floating-point value, respectively, at the top of the operand stack. They have no explicit operands.

Instruction `IINC` increments the integer value of a local variable. It has two explicit operands, the first is the slot number of the local variable, and the second is the integer constant value from -128 up to and including 127, which is the amount to increment the local value.

### 5.3 Logical

Jasmin instructions `IAND` and `IOR` pop off the top two integer values from the operand stack, perform a logical AND, or OR operation on the values, respectively, and pushes the result onto the operand stack. They have no explicit operands. Note that MC uses the short-circuit evaluation for logical expressions.

### 5.4 Convert

The Jasmin instruction `I2F` converts the integer value at the top of the operand stack to floating-point.

### 5.5 Create Objects and Arrays

Jasmin's instruction `NEWARRAY` creates an array of scalar values. It has one explicit operand, the scalar type int, float, or boolean, and it expects the integer value on top of the operand stack which is the count of elements to create for the array. It creates the array and pushes the array's address onto the operand stack. Examples:

```
newarray int
newarray float
```

Instruction `ANEWARRAY` is similar, except that it creates an array of object references (addresses). It requires the count of elements on top of the operand stack, and its one explicit operand is the fully qualified name of a class. For example:

```
anewarray java/lang/String
```

## 6 Control Instructions

The Jasmin control instructions perform comparisons and branches, as well as function calls and returns.

### 6.1 Unconditional Branch

The Jasmin `GOTO` instruction does an unconditional branch to a labeled statement. Its one explicit operand is the statement label (without the colon). For example:

```
goto Label0
```

A `GOTO` instruction can branch only to another statement in the same Jasmin method.

### 6.2 Compare and Branch

The Jasmin instructions `IFEQ, IFNE, IFLT, IFLE, IFGT` and `IFGE` each pops off the integer value at the top of the operand stack, compares the value to zero, and branches if the value is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to zero, respectively. Each instruction has a statement label as its one explicit operand. Examples:

```
ifeq Label0
iflt Label5
```

A common Jasmin programming convention is integer value 0 represents the boolean value $false$ and 1 represents $true$.
The instructions `IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT, IF_ICMPLE, IF_ICMPGT` and `IF_ICMPGE` pops two integer values off the operand stack and compares them. It branches if the first value is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to the second value (the one that was at the very top of the stack), respectively. Each instruction has a statement label as its one explicit operand. For example:

```
if_icmpgt Label42
```

The instructions FCMPG and FCMPL pops two floating-point values off the operand stack and compares them. If the first value is equal to the second value, it pushes 0 onto the stack. If the first value is less than the second value, it pushes $-1$. If the first value is greater than the second value, it pushes 1. Otherwise, at least one of the first value or the second is $NaN$. The FCMPG instruction pushes the integer value 1 onto the operand stack and the FCMPL instruction pushes the integer value $-1$ onto the stack. They have no explicit operands.

## 6.3 Call and Return

Jasmin instructions that the MC compiler will emit to call a method are INVOKESTATIC and INVOKESPECIAL. Each instruction pushes a new stack frame onto the Java runtime stack for the called method.
Instruction INVOKESTATIC calls a static method of a Jasmin class. It has one explicit operand, the method signature (fully qualified name and formal parameter type descriptors) of the Jasmin method concatenated with the return type descriptor. If the method requires any actual parameter values, they are required to be on top of the operand stack, pushed in the correct order. Unless the called method has no return value (i.e., it returns void), the instruction pushes the return value of the method onto the caller's operand stack. Examples:

```
invokestatic io/getInt()I
invokestatic io/putStringLn(Ljava/lang/String;)V
invokestatic io/putFloat(F)V
```

The MC compiler will emit the INVOKESPECIAL only to call constructor methods. Jasmin constructor methods are always named <init> and they never return values. For example:

```
invokespecial java/lang/Object/<init>()V
```

Jasmin has several instructions that return from a method call: RETURN, IRETURN, FRETURN and ARETURN. Instruction RETURN returns to the caller when the called method does not have a return value. Instructions IRETURN, FRETURN and ARETURN each pops an integer, floating-point, or address value, respectively, from the operand stack as the return value, and then it returns from a method call. The calling instruction then pushes the return value onto the caller's operand stack. Each of the return instructions pops off the called method's stack frame from the Java runtime stack.

# Code Generation

## 1 Compiling Program

**Program header**

```
.source <file_name>
.class public <program_name>
.super java/lang/Object
```

Code for generating fields for global variables

**Class constructor**

```
.method public <init>()V
.var 0 is this L<program_name>; from Label0 to Label1
Label0:
        <code for setting default values for global variables>
        aload_0
        invokespecial java/lang/Object/<init>()V
Label1:
        return
.limit stack 1
.limit locals 1
.end method
```

Code for methods for function declarations

Code for method for the main function declarations

### 1.1 Program Header

The program header consists of the `.source` directive, the `.class` directive and the `.super` directive. The `.source` directive specifies the source of assembly. The `.class` directive includes the keyword public followed by the MC program name in lower case. The `.super` directive specifies the superclass, which for program class is always `java/lang/Object`.

### 1.2 Global Variables

For each MC global variable declared, the compiler generates a static field of the program class. For example,

```
int a,x[3];
float b,y[5];
boolean c,z[2];
string d,t[4];
```

compiles to

```
.field static a I
.field static x [I
.field static b F
.field static y [F
.field static c Z
.field static z [Z
.field static d Ljava/lang/String;
.field static t [Ljava/lang/String;
```

The `.field` directive specifies each field of the program class. It uses the static keyword followed by the field name and the type specification.

## 1.3 Class Constructor

The MC compiler always generates the same default constructor for the program class. All Jasmin constructors are named `<init>`. When the constructor executes, slot 0 of the local variables array contains the address of the class (i.e., `this` in Java). Then the instructions for initializing the global variable's values are executed (you can also put these instructions in the main method instead). Finally, the `ALOAD_0` instruction pushes this address onto the operand stack and the `INVOKESPECIAL` instruction passes it to the constructor of `java/lang/Object`.

## 1.4 Main Function

**Main method header**

```
.method public static main([Ljava/lang/String;)V
```

Code for generating directives for local variables

Code for setting default values for global variables
(you can also put these instructions in the `<init>` method instead)

Code for setting default values for local variables

Code for statements

**Main method epilogue**

```
        return
.limit stack m
.limit locals n
.end method
```

# 2 Compiling Function Declarations

**Method header**

```
.method public static <function_name><return_type_descriptor>
```

Code for generating directives for parameters and local variables

Code for setting default values for local variables

Code for statements

Code for return statement (just in case void function does not return)

**Method epilogue**

```
.limit stack m
.limit locals n
.end method
```

In Jasmin method for a declared MC function, the values of the method's formal parameters and local variables are stored in slots of the local variables array. Slot 0 is for the first parameter, and the following parameters and local variables are assigned successive slots. The `.var` directive specifies the slot number and data type for each parameter and local variable. For example, the statements

```
float test(int a, float b, boolean c[]){
string x,y[3];
...
```

compile to

```
.method public static test(IF[Z)F
.var 0 is a I from Label0 to Label1
.var 1 is b F from Label0 to Label1
.var 2 is c [Z from Label0 to Label1
.var 3 is x Ljava/lang/String; from Label0 to Label1
.var 4 is y [Ljava/lang/String; from Label0 to Label1
...
```

# 3 Compiling Statements

# 4 Compiling Expressions

# 5 Compiling Identifiers, Array's Elements and Literals