The MOC-V Programming Language

A Language for Memory Coherency and Ordering Validation in x86 Processors

Benjamin Panning
bjp2122

Table of Contents

1 Language Summary 3
2 Data Types, Variables, and Operators 3
2.1 Data Types 3
2.2 Variable Names and Declarations 3
2.3 Arrays 4
2.4 Arithmetic Operators 4
3 Expressions and Program Structure Constructs 5
3.1 Comments 5
3.2 Expressions 5
3.3 Statements 5
3.4 Conditional Constructs 6
3.5 Looping Constructs 6
3.6 Functions 6
3.6.1 Variables as Input Parameters 7
3.6.2 Arrays as Input Parameters 7
3.6.3 Return Values from Functions 7
4 Program Structure and Scoping 7
4.1 Statement Blocks 7
4.2 Function Blocks 7
4.3 Random Code Blocks 8
4.4 Random Code 8
4.5 Additional Scoping Rules 9
5 Automatic Data Consistency Checking 9
5.1 Defining a Memory Region to be Tested 9
5.2 Simulating Data Modification 9

1 Language Summary

The increase in parallelism within CPU architecture has made validation of memory coherency increasingly more difficult. The best way to ensure coherency and memory ordering is to randomize memory access patterns and perform a check to determine whether data corruption has occurred. Typical ways to do this include generating random sequences of instructions and comparing the resulting memory map to a simulated memory map.

Another approach would be to generate a sequence of instructions that follow a directed algorithm, with specific areas for randomization. This should allow a programmer to write an algorithm that is stressful on the memory hierarchy, while still being able to specify a degree of randomness to increase the likelihood of catching memory coherency and ordering issues.

This language is called MOC-V, and is described in detail within this document.

2 Data Types, Variables, and Operators

2.1 Data Types

MOC-V is concerned only with a limited number of data types. This language is concerned only with writing patterns of data to memory, and therefore does not implement any form of floating point numbers. The data types incorporated in this language are as follows:

void	The undefined type
char	One byte data type representing an ASCII character
ord1	One byte data type used in numerical calculations and data access
ord2	Two byte data type used in numerical calculations and data access
ord4	Four byte data type used in numerical calculations and data access

2.2 Variable Names and Declarations

Variable names within MOC-V are a series of consecutive non-white space characters that start with a letter, including an underscore, and contain one or more letters or numbers. Variables may not take the same name as any key-words defined within the language.

Variables must be declared before they can be used. Also, variables may not be declared more than once within the same scope. For instance, if a variable is declared within the global score, that variable may not be re-declared within the global scope or re-declared in any narrower scope contained within the global scope.

A variable declaration consists of a data type followed by a variable name, and an optional initialization. The format of this is as follows:

```
data_type variable_id = initial_value
```

So for instance, to declare a four byte variable named “x” and initialize it to the value 0x1234, the user would follow the form:

```
ord4 x = 0x1234 ;
```

2.3 Arrays

Arrays can be implemented to store collections of data types. The most common uses are for storing arrays of characters which can be interpreted as text and creating regions of memory to be tested. Arrays have the property of being safe, in that if they are indexed beyond their range they wrap-around and begin indexing from the beginning of the array. Arrays are defined as follows:

```
array array_id[size_in_ord4] = initialization_opt
```

The first thing to notice is that an array has no data type associated with it. Arrays are always declared with a size given in bytes, and are always accessed on a byte boundary – regardless of the data type being associated with them. This is to allow unaligned accesses when using ord2 and ord4 data types. Arrays are accessed as follows:

```
array_id[byte_offset]data_type
```

2.4 Arithmetic Operators

Both logical and bitwise arithmetic operations are supported in MOC-V. The set of valid arithmetic operators within MOC-V are as follows:

+	Addition
-	Subtraction
*	Multiplication
/	Integer division
%	Remainder division
^	Bitwise Exclusive-Or
|	Bitwise Or
&	Bitwise And
=	Assignment

There are also a number of valid comparators within MOC-V. These are operators which return one if the comparison is true or zero is the comparison is false. These operators are as follows:

==	Equal
!=	Not Equal

>	Greater than
<	Less than
>=	Greater than or Equal
<=	Less than or Equal

Finally, there are operators which are used to index into arrays and break expressions into explicit expression grouping – thereby guaranteeing an ordering of operations. These operators are as follows:

( )	Expression Grouping
[ ]	Array Indexing

All operators are left associative except for the assignment operator which is right associative. The precedence of operators is as follows:

Highest Precedence:	( )
	* / %
	+ -
	^ | &
	== != > < >= <=
Lowest Precedence:	=

3 Expressions and Program Structure Constructs

3.1 Comments

Comments are indicated either using an enclosing /* */ structure or using a single line double slash, //. MOC-V also supports nesting enclosed comments. For instance, the comment:

```
/*  This is /* a valid */ comment */
```

3.2 Expressions

Expressions are any constant, variable, or series of constants, variables, and operators that result in a value which can be assigned to a variable or used for comparison.

3.3 Statements

Statements consist of combinations of expressions that identify a single programmatic action. Statements must be ended with a semicolon, which represents a sequencing between individual statements. This

sequencing between statements guarantees that one statement will be completed before the next statement is executed.

3.4 Conditional Constructs

MOC-V supports conditional constructs that allow if-then-else structures to be created. These constructs are as follows:

if(expr) stmt	If-statement with no concluding else-statement
if(expr) stmt else stmt	If-statement including else-statement

3.5 Looping Constructs

Looping constructs are supported both in the form of while-loops and for-loops. These may be constructed as follows:

while(expr) stmt	While statement that executes until expr evaluates to 0
for(expr ; expr ; expr) stmt	For statement with pre-executed expression, evaluated expression, and in loop executed expression

It should be noted that both the for and the while constructs require the expressions listed – they are not optional. In the case of the while loop, the expression is intended to be a comparison or calculation that evaluates to non-zero when the loop is to continue and zero when the loop is to terminate.

For the case of the for-loop, the expressions are intended to be an initialization, a calculation to determine if the loop terminates or continues, and an expression to be executed following each loop, respectively.

3.6 Functions

Functions are used within MOC-V to break programs into modular units that contain similar functionality. Functions are defined by declaring a type associated with the return value, which can be void to represent no return, followed by an identifier and a parameter list. The parameter list may contain void, but may not be empty. This is defined as follows:

```
data_type function_identifier( opt_param_list )
```

Where opt_param_list is a list of input parameters in the form of data_type followed by identifiers. Functions must be declared before being used, however, the declaration of a function can be the same as its definition. In the case where a function must be declared before it is defined, the syntax is as follows:

```
data_type function_id( opt_decl_list );
```

The difference between the function call and the function declaration is that the declaration must include the data types of each input parameter associated with the function. The form of this is a data type followed by an identifier. The identifier specified within the function declaration is then treated as a variable local to that function block – following the same scoping rules as a local variable declared within a function block.

3.6.1 Variables as Input Parameters

When a function takes a data type as input that is not an array, that variable is always passed by value. There is no concept of passing values by reference within MOC-V, which means that any variable passed to a function is guaranteed not to be modified or corrupted by that function.

3.6.2 Arrays as Input Parameters

When a function takes an array as an input parameter, that array is always passed as a reference to the data array. This is necessary to limit the size of the stack when passing large arrays. Arrays that are declared locally are still defined on the stack, and when passed as parameters the reference is made to that declared data on the stack. These references are not stack-pointer or base-pointer relative, and are an offset from the base of the stack segment to guarantee that references made to arrays on the stack are maintained regardless of how the stack-pointer is modified.

3.6.3 Return Values from Functions

Functions may return any data type including void, but may not return arrays. Only a single instance of any returnable data type may be returned from any function.

For any function that needs to return an array, it must be done using the standard array references to either a locally defined array in the calling function or to a globally defined array.

4 Program Structure and Scoping

4.1 Statement Blocks

Program structure within MOC-V is defined through statement blocks, function blocks, and random code blocks. Statement blocks consist of a set of statements between two brackets, as follows:

{ stmt_list }

The stmt_list in this format is a single statement or a list of statements. White space within this form is neglected.

Scoping rules within MOC-V maintain a variable only within the block in which the variable is defined. A variable must also be declared before it can be used. Variables declared outside of the scope of any function block are considered global variables, and are visible within any block contained within the program

4.2 Function Blocks

Function blocks include the first statement block following a function declaration. Function blocks may not be defined within other function blocks, and must also be declared before they are used within program

order. All functions are declared at the global level, and are therefore visible to all other code segments following the functions declaration.

4.3 Random Code Blocks

MOC-V allows users to define random code blocks. These are code blocks which, when randomly selected for inclusion within the current compilation, are treated much like a normal function block – except with the restriction that it cannot accept a list of parameters or return any data value.

Random code blocks are declared using the following syntax:

```
rcode rcode_block_id;
```

The term rcode is a keyword which defines a random code block. A random code block is defined in the same manner that a function block is defined – which includes the rcode keyword followed by an identifier and then a statement block.

```
rcode rcode_block_id { stmt_list }
```

4.4 Random Code

One of the key features of MOC-V is that it allows the programmer to specify an amount of randomness within a program. This randomness is defined using random code blocks and the random code sequences are generated at compile time based on an input seed value.

Random code sequences are defined using the block declarations previously specified. The random code blocks act much like a function block, except that they are not guaranteed to run, and because they do not accept nor return values, cannot be integrated within the core functionality of a MOC-V program.

Once a random code block is defined, it may be used within the program structure as follows:

```
[ (weight)rcode_block_id ]
```

Where weight is a value that determines the odds that a given random code block is inserted at compile time. This weight must be an integer value between 0 and 100, where a value of 0 indicates that the random code block will never be inserted, and a value of 100 means that the code block will be inserted at every compilation.

The selection of random code blocks for insertion within the program can also be specified as selection from a list of random code blocks. This can be done using the following syntax:

```
[ (weight1)rcode_block_id1 (weight2)rcode_block_id2 … ]
```

Using this syntax, at compile time the compiler will select one or none of the listed random code blocks for insertion within the program. The weights specified within this syntax must sum to a value less than 100, and determination of which random code block is selected is based upon the weight of each code block using the same rules specified for a single random code block.

This form of selecting random code blocks may also be used within the random code block definitions themselves – allowing for programmers to randomize the random code blocks themselves.

It should be noted that random code blocks may reference other random code blocks, however it is not recommended to attempt recursion or cycles within random code blocks – as these lead to infinite loops within program structure.

4.5 Additional Scoping Rules

In addition to the block level scoping rules, it is also required that all variables are declared before they are used within each block. There is a single namespace within MOC-V that includes variable names, function names, and random code block names. Identical names, even when referring to objects of different types or usage, will be considered conflicting.

Statement blocks always use open scoping rules. This includes function blocks as well as random code blocks. This open scoping means that each block of code inherits the symbol tables of its calling blocks, including every symbol table in the chain all the way to the global scope.

Finally, scoping in MOC-V is static, meaning that the relationship between names and the variables they represent is determined entirely by the text of the program, and not at run-time.

5 Automatic Data Consistency Checking

MOC-V is designed to allow programmers to automatically generate data consistency checks for modifications made to regions of test memory. There is no benefit to being able to generate directed random data access patterns if there is no way to check whether the data remained consistent during program execution. For this reason, MOC-V simulates the modifications made to each test region of memory and allows the user to automatically perform a check on those regions to determine if the memory access and modification was coherent.

5.1 Defining a Memory Region to be Tested

Consistency checking is only performed on memory that is globally defined and marked globally as memory to be checked for consistency. A region of memory that can be tested must be a globally defined array. This array must also be marked to indicate that it is being tested for consistency. The syntax for marking a region of memory as a region being tested is as follows:

```
#test_region        array_identifier
```

This syntax must be used in conjunction with a global declaration of an array with a matching identifier name.

5.2 Simulating Data Modification

The modifications to the data array are tested both in random and non-random sections of code. This means that the user can modify data using only random code blocks, or using none at all, and still the coherency check will be accurate.

The data modification is simulated entirely during compile time – the final resulting memory arrays will be known before the program is executed. The compiler simulates all operations performed on each memory address being tested and constructs a resulting memory image from that simulation.

The memory image is placed in the same data segment that the tested arrays are located in. If the user wishes to compare this memory image to the original data arrays, the user may access the array using the name of the array being tested prefixed with the word "mocv_". For instance, if the user defines an array for test as follows:

```
#test_region        my_data_array
array               my_data_array[100]
```

Then the final memory image for what that data array should be after the program has completed modifying it can be accessed using the identifier:

```
mocv_my_data_array
```

A typical way to check for consistency using this mechanism would be to do a comparison over the entire data array with its corresponding test array. This could be done as follows:

```
for( i = 0; i < array_size; i = i + 1 )
{
        if( my_data_array[ i ] != mocv_my_data_array[ i ] )
        {
                /*  Flag an error  */
        }
}
```