

# bombs-must-detonate: Virtual Machine Implementation

Mason Smith

June 11, 2008

## Abstract

bombs-must-detonate (BMD) is a multiplayer Bomberman-like game in which human players program AIs to compete with each other. bombs-must-detonate was written by Brian Go, Dane Rukavina, and Mason Smith for CS136c. This document details the usage and implementation of the bombs-must-detonate virtual machine.

## 1 Getting bombs-must-detonate

BMD can be downloaded at <http://bmd.masonium.com/bmd-src.tar.gz>. You can also check out the code directly from its Subversion repository. Use the following command to check out the project:

```
svn checkout http://bombs-must-detonate.googlecode.com/svn/trunk/ bombs-must-detonate-rea
```

## 2 Installation and Usage

The VM is contained in the folder appropriately labelled `vm`. Compiling the VM requires the Boost C++ libraries, which can be downloaded at <http://www.boost.org/users/download/>. To compile the virtual machine, simply use `make all` in the `vm` folder. Compiling yields three components. the executables `vm-test` and `vm-run`, and the static library `libbmdvm.a`.

### 2.1 vm-test

`vm-test` runs the unit tests to ensure that the virtual machine is operating properly. The tests are located in the subfolder called `tests`. Tests fail when they cannot be loaded, when they throw runtime exceptions, or when they fail execution signals (see the `assert` execution signals 3.5.2).

### 2.2 vm-run

`vm-run` is used to run a single file. It takes the assembly file to run as input. It optionally takes a function name (defined in the assembly file) as well as any arguments. Only primitive 3.1 arguments can be passed.

## 3 Architecture

The BMD virtual machine was inspired by the O’Caml virtual machine and thus shares many features. The machine is stack-based, with an accumulator. It natively recognizes a variety of types. Special care was taken to make the machine particularly safe with respect to types and scoping. Each machine also has a number of global fields, which must be allocated statically.

### 3.1 Types

The machine is typed, recognizing the following types:

- integers
- floating-point decimals
- strings
- lists
- tagged blocks
- null (uninitialized value)

Integers, floating points, and strings are primitive values. When manipulated on the stack, these values are copied, rather than referenced. Lists and blocks are manipulated by reference rather than by value.

Lists are cons pairs, where the cdr must be a list. There is a special value of a list called *nil*, which represents the empty list. Lists are immutable, in that their cars and cdrs cannot be set after creation of the list.

Blocks are fixed-length arrays, which can be tagged as structs or arrays. Most operations do not distinguish between blocks that have different tags, though there are some exceptions.

Null values are initialized. Blocks that are not created with initializer values have null in their elements. Operations that expect specific types typically throw an error specifically stating when they receive an uninitialized value.

### 3.2 Stack Manipulation

Stack manipulation is again similar to that of O’caml. One can load values to the accumulator from the stack and to specific locations in the stack from the accumulator. Manipulation between stack elements is much more limited, though there are a few special instructions which alter the stack only (for instance, the rev instruction). Return values from functions are normally placed on the stack.

One interesting safety feature of BMD VM is the special treatment of the stack frame. Like the O’caml VM, function calls are set up by pushing the return address onto the stack. This creates a new stack frame record. However, normal pop operations cannot pop this return address. So, one cannot destroy the stack by merely trying to step out of the stack without returning properly.

### 3.3 Remote Procedure Calls

Two virtual machines can be linked together and call each other's functions. In addition, the virtual machine itself provides functions (as opposed to opcodes) that an assembly program can call. Calls of either type are referred to as remote procedure calls. Using these functions is similar to a normal function call, as far as setting up the stack goes. However, RPCs return two values. The first value is whether or not the remote call succeeded. A remote can fail for any number of reasons:

- The virtual machine does not recognize the specified function, and the linked virtual machine does not have the function (or there is no linked virtual machine).
- The number of arguments pushed onto the stack was insufficient.
- On a linked call, the call threw a runtime error.

The actual return value of the function is pushed onto the top of the stack.

### 3.4 Instructions

A full listing of the instructions is available at BMD's google code wiki.

### 3.5 Execution Signals

Execution signals are special signals to the virtual machine that perform various actions. They are denoted by a hyphen (-) in the assembly code.

#### 3.5.1 -function

The `-function` signals indicates the beginning of a function.

```
-function,my_add,1
__my_add_0:
acc 0
pop
add
return
```

The signal takes two parameters. The first is the name of the function, and the second is the number of arguments that the function takes. The function name is not required to be the same as the label starting the function, and the virtual machine can call functions based on label name or function signal name. Thus, `call my_add` and `call __my_add_0` do the same thing. Local calls are *not* checked for the number of arguments. This information is used internally to keep track of the function call stack.

### 3.5.2 `-assert`

The `-assert` signal is mostly used for testing. It can be used to assert that certain conditions are met when control passes through the signal. The `-assert` signal has multiple forms, demonstrated below.

```
const 2
push
const 3
add
-assert,acc,5
push
const 0
lt
jz after
-assert,does_not_reach
const "failure"
after:
const "awesome"
...
```

The `-assert,acc` signal checks that the accumulator has the specified value. The suggested value can be any primitive, but not reference types. The `-assert,does_not_reach` throws an error if control passes through it. This is useful for ensuring that control goes in a certain direction.

Another `-assert` signal is `-assert,error`. This signal throws an error if the execution of the bytecode does not throw an error. This is mainly used for testing, to confirm (for instance) that certain instructions are typesafe.

### 3.5.3 `-allocfields`

This signal tells the virtual machine the number of global fields to allocate. If the signal is encountered multiple times in a file, the last one is used. `-allocfields` is interpreted at load-time, so the number of fields is fixed once program execution begins.

### 3.5.4 `-init`

This signal specifies a function to be called immediately after the file has finished loading.

## 4 Implementation

The virtual machine was implemented in 3900 lines of C++, with about 500 additional lines of tests in BMD assembly code. C++ was the language known best by Dane and I collectively, and the game engine and the virtual machine needed to be fairly well-integrated. Although using C++ had many quirks, one surprising non-issue was that of memory management. By extensive use of the `boost::shared_ptr<T>` wrapper and STL container classes, memory issues never became a major problem in the code.

One peculiar aspect of my code is the extensive use of macros. Normally I frown on C macros, but C++ lacks the full blown code-generation capabilities that, say Lisp has, and they were required to avoid code duplication that would probably double the size of the code as it currently stands. In any case, I tried to limit my use of the macros

If you want to start viewing the code, the best places start are `value.cpp/hpp`, `instr.cpp/hpp`, and `vm.cpp/hpp`.

## 4.1 Runtime Values

Values were implemented using the `boost::variant` template class. This class essentially allowed values to be defined in a style more reminiscent of O’Caml’s variant types, albeit much more cumbersome. In general, there are two methods of operating on a `boost::variant` derivative. The primary method used in the VM implementation was the visitor pattern. A class inheriting from `boost::static_visitor<T>` defines the `()` operator on each relevant class. For instance, one would define a `to_string` method as such:

```
class to_string_visitor : boost::static_visitor<string>
{
public:
    string operator(int i) { return string(itoa(i)); }
    string operator(double d) { return string(ftoa(f)); }
    string operator(ListRef lr)
    {
        return boost::apply_visitor(to_string_visitor(), lr->get_car()) +
            to_string_visitor()(lr->get_cdr());
    }
    ...
};
```

The above code also demonstrates using a visitor, via a call to `boost::apply_visitor`. This method is again similar to O’Caml’s pattern matching, albeit much more verbose and less transparent.

The second method is using the `boost::get<T>` method, which returns a pointer to the appropriate value iff the variable stored is actually of the appropriate type. This is less recommended and in general less useful, since defining any method on all varieties of values would require very tedious case analysis.

## 4.2 Actors

A BMD virtual machine is not actually created directly. (As a matter of fact, it can’t be created directly, as it has a private constructor.) Instead, an Actor is created, which owns its virtual machine. The actor is responsible for holding all of the information related to a specific agent, such as the instruction list, the global fields, and the list of available functions. Actors can contain a link to another actor, in which case the virtual machine exports remote calls to the linked virtual machine / actor.

An alternative architecture was previously considered, in which all actors operate under one global virtual machine. RPCs to linked actors would be

simulated on the same stack, but each actor would appear to be running on its own virtual machine. This architecture has some useful benefits; for instance, keeping track of the call stack across actors is much simpler. (This issue is not actually fully solved in my current implementation. The call stack is traced within an actor, but calls that the other actor makes are not fully recorded.) However, in the separate-VM structure, one has to worry less about stack corruption between actors.

### 4.3 File Loading

A significant part of file loading was case analysis on the instruction type. A couple of macros were used to make the code shorter, but it essentially looks as follows:

```
if (instr_name == "acc")
    return new acc_instr();
if (instr_name == "return")
    return new return_instr();
if (instr_name == "const")
    return new const_instr(Value::parse_value(arg1));
...
```

Note that this sort of case analysis is necessary in many languages, not just object-oriented ones. One notable exception is Java. One can use reflection to automatically generate an instance of an object given only the name of the object in string representation. Since C++ doesn't have reflection, however, this wasn't an option.

All of the relevant parsing code is located in `actor_parse.cpp`.

### 4.4 Remote Procedure Calls

When an RPC instruction is encountered, the VM is inspected to see if it contains that function. VM functions are implemented as normal C++ functions, which are registered to the VM. This allows a user of the BMD VM library to add their own functions to the VM to be available to the programs.

When the function name is not found in the VM, the linked actor is checked for the function. (The list of functions the actor has access to in its instructions is generated by the `-function` execution signals.3.5.1). The VM has a method called `VM::call_function`, which sets up a call stack using supplied arguments and runs the code.

One disadvantage to this approach is that counting the number of execute instructions becomes more difficult, as transferring control to the linked VM is treated as the execution of only one instruction. (Again, this is one of those problems that I didn't quite solve.)