

15-122: Principles of Imperative Computation, Fall 2013

Homework 9 Programming: The C0VM

10 40
15 01
14 05 D2
CF
B0

Checkpoint: Monday, November 25, 2013 by 22:00

Due: Thursday, December 5, 2013 by 22:00

In this assignment you will implement a virtual machine for C0, the C0VM. It has been influenced by the Java Virtual Machine (JVM) and the LLVM, a low-level virtual machine for compiler backends. We kept its definition much simpler than the JVM, following the design of C0. Bytecode verification, one of the cornerstones of the JVM design, fell victim to this simplification; in this way the machine bears a closer resemblance to the LLVM. It is a fully functional design and should be able to execute arbitrary C0 code.

The purpose of this assignment is to give you practice in writing C programs in the kind of application where C is indeed often used in practice. C is appropriate here because a virtual machine has to perform some low-level data and memory manipulation that is difficult to make simultaneously efficient and safe in a higher-level language. It should also help you gain a deeper understanding how C0 (and, by extension, C) programs are executed.

The code handout for this assignment is at

<http://www.cs.cmu.edu/~rjsimmon/15122-f13/hw/15122-prog9.tgz>

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a 15 handin limit for the checkpoint, and a 20 handin limit for the full assignment. Additional handins will incur a 1-point penalty per handin. *The checkpoint is for Task 1 only. This is far, far less than half the assignment; if you have only finished the checkpoint by the checkpoint deadline you should consider yourself behind schedule.*

About this writeup

The C0VM is defined in stages, and we have test programs which exercise only part of the specification. We strongly recommend that you construct your implementation following these stages and debug and test each stage (on your own *and* with the autograder) before moving on to the next. Each part has its own challenges, but each part should be relatively small and self-contained.

This document describes the structure of the C0VM first, then the instruction set (bytecodes) for the C0VM, and then the file format for a C0 program in bytecode form. After this, the document will describe the tasks you need to perform, step by step. Read this document very carefully as you prepare to do your work.

1 The Structure of the C0VM

Compiled code to be executed by the C0 virtual machine is represented in a byte code format, typically stored in a file ending in extension `.bc0` which we call the *bytecode file*. This file contains numerical and string constants as well as byte code for the functions defined in the C0 source. The precise form of this file is specified in Section 3.

1.1 Types

C0 has so-called *small types* `int`, `bool`, `char`, `string`, `t[]`, and `t*`. Values of these types can be passed to or from functions and held in variables. We think of the small types as constituting two classes: the *primitive types* `int`, `bool`, and `char` and the *reference types* `string`, `t[]` and `t*`. All primitive types are conflated in the C0VM and denoted by `w32`, indicating a 32 bit word. Values of primitive types are denoted by x , i , or n . We also conflate all reference types and write `*`. Values of reference type are denoted by a for *address*. An address uses either 32 bits or 64 bits. On the Linux machines you use, it will be 64 bits.

In addition C0 has *large types* `struct s` which cannot be passed directly, but must be stored in memory. When the C0VM executes a program, it stores values of large type on *its* heap and refers to them by their address. Calculations of how to access struct fields are performed statically by the compiler, which computes proper offsets for each access. In addition, arrays (referenced by values of type `t[]`), memory cells (referenced by values of type `t*`) and strings (referenced by values of type `string`) are stored on the heap.

1.2 Runtime Data Areas

The C0VM defines several data areas that are used during the execution of a program.

1.2.1 The Program Counter

In Clac, the program instructions were strings, and they were stored in a queue of strings. In the C0VM, instructions are stored in an array of (unsigned) bytes (`ubyte *P`). The program counter `pc` holds the address of the program instruction currently being executed. Unless a non-local transfer of control occurs (`goto`, conditional branch, function call or return) it is incremented by the number of bytes in the current instruction before the next instruction is fetched and interpreted.

1.2.2 The Call Stack

The C0VM has a call stack consisting of *frames*, each one containing local variables, a local operand stack, and a return address. The call stack grows when a function is called and shrinks when a function returns, deallocating the frame during the return.

1.2.3 The Operand Stack

The C0VM is a *stack machine*, similar in design to Clac and the JVM. This means arithmetic operations and other instructions pop their operands from an operand stack and push

their result back onto the operand stack. This is in contrast with *register machines* where instructions such as arithmetic operations act on a finite set of registers.

1.2.4 The Heap

At runtime, the C0 heap contains C0 strings, C0 arrays (`t[]`) and C0 cells (`t*`). C0 arrays have to store size information so that dynamic array bounds checks can be performed. You will probably want to use the C heap to implement the C0 heap: that is, you will allocate strings, arrays, and cells directly using calls to `xmalloc` and `xcalloc` as defined earlier in this course.

Since C0 is designed as a garbage-collected language, you will not be able to free space allocated on behalf of C0 unless you are willing to implement (or use) a garbage collector. We do not view this as a memory leak of the C0VM implementation. On the other hand, temporary data structures required for the C0VM's own operation should be properly freed.

1.2.5 Constant Pools

Numerical constants requiring more than 8 bits and all string constants occurring in the program will be allocated in constant pools, called the *integer pool* and the *string pool*. They never change during program execution.

1.2.6 Function Pools

Functions, either C0 functions defined in a source file or library functions, are kept in pools called the *function pool* and the *native pool*, respectively. Functions in the function pool are stored with their bytecode instructions, while functions in the native pool store an index into a table of C function pointers that the C0VM implementation can dereference and invoke.

1.3 Frames

A *frame* stores data and partial results during the execution of a function body. It holds a return address, which should be the next instruction in the calling function, an array of local variables denoted by $V[0], \dots, V[num_vars - 1]$, and an operand stack for computing expression values. At any point during the execution there is a *current frame* as well as a *calling frame*, where the latter is restored when a function returns to its caller.

Since values of small type may occupy 4 bytes or 8 bytes, the data structures implementing frames, including the array of local variables and the operand stack, allocate 8 bytes for each data value, whether of primitive type or reference type. From the C perspective, things are simplest if these are of type `void*`, but to clarify intent, we create a type alias `c0_value`, defined to be `void*`, for arbitrary C0 values. This strategy requires us to be able to cast ints to pointers and vice versa without loss of information. The C99 standard leaves this behavior implementation-defined, but the Andrew version of `gcc` implements such casts. We define an inline function `VAL(x)` to cast an integer x to a `c0_value`, and a converse `INT(p)` to cast a pointer p to a 32-bit signed integer (an `int32_t`). For the latter to work, p must have been created with a `VAL(x)` cast, otherwise the result is unpredictable except for `NULL` (which returns 0). So we always have `INT(VAL(x)) == x` for any integer x .

The main function we have provided (see the file `c0vm_main.c`) performs some simple tests to verify that casts between `void*` and `int` do not lose information. If any of these tests fail, C0VM will abort with an appropriate error message.

1.4 Runtime Errors

In order to fully capture the behavior of C0 programs, you must correctly issue errors for things like dereferencing NULL, indexing into an array outside of its bounds, and dividing by zero. Check the C0 Language Reference for details on what kinds of errors you must handle, and then use the following provided functions (defined in `c0vm_abort.h` and `c0vm_abort.c`) to issue appropriate error messages:

```
void c0_user_error(char *err);           // for calls to error() from C0
void c0_assertion_failure(char *err);   // for failed assertions from C0
void c0_memory_error(char *err);        // for memory-related errors
void c0_arith_error(char *err);         // for arithmetic-related errors
```

For unexpected situations that arise while executing bytecode, situations which could indicate a bug in your VM, you may use the standard C library functions `abort` or `assert` to abort your program. See Section 4.3 for more details on this distinction.

2 Instruction Set

We group the instructions by type, in order of increasing complexity from the implementation point of view. We recommend implementing them in order and aborting with an appropriate message when an unimplemented instruction is encountered.

2.1 Stack Manipulation

There are three instructions for direct stack manipulation without regard to types.

```
0x59 dup           S, v -> S, v, v
0x57 pop           S, v -> S
0x5F swap          S, v1, v2 -> S, v2, v1
```

2.2 Arithmetic Instructions

Arithmetic operations in C0 are defined using modular arithmetic based on a two's complement signed representation. This does not match your implementation language (C) very well, where the result of *signed* arithmetic overflow is undefined. There are two solutions to this problem: first, because *unsigned* arithmetic overflow is defined to be modular arithmetic, you could cast `int32_t` values to `uint32_t`, perform unsigned arithmetic, then cast back. This should not be required for in your implementations, however: we will compile your code with the `-fwrapv` command-line argument that forces `gcc` to treat integer arithmetic to be defined as signed twos-complement arithmetic. (Remember, however, that this is not part of the C standard.)

Casting signed integers to unsigned integers and/or using the `-fwrapv` command-line argument does not do everything that is needed to ensure C0 compliance, but it goes a long way. We recommend a careful reading of the arithmetic operations in the C0 Reference, as well as the file `signed-casting.c` which demos some of C's casting behavior.

For this implementation strategy to be correct, it is important to verify that our C environment does indeed use a two's complement representation and that the C type of `int` has 32 bits. The provided `main` function (see the file `c0vm_main.c`) performs these checks before starting the abstract machine and aborts the execution if necessary.

In the instruction table below (and for subsequent tables), we use `w32` for the type of primitive values and `*` for the type of reference values. Each line has an opcode in hex notation, followed by the operation mnemonic, followed by the effect of the operation, first on the stack, then any other effect.

```
0x60 iadd      S, x:w32, y:w32 -> S, x+y:w32
0x7E iand      S, x:w32, y:w32 -> S, x&y:w32
0x6C idiv      S, x:w32, y:w32 -> S, x/y:w32
0x68 imul      S, x:w32, y:w32 -> S, x*y:w32
0x80 ior       S, x:w32, y:w32 -> S, x|y:w32
0x70 irem      S, x:w32, y:w32 -> S, x%y:w32
0x78 ishl      S, x:w32, y:w32 -> S, x<<y:w32
0x7A ishr      S, x:w32, y:w32 -> S, x>>y:w32
0x64 isub      S, x:w32, y:w32 -> S, x-y:w32
0x82 ixor      S, x:w32, y:w32 -> S, x^y:w32
```

`idiv`, `irem`, `ishr`, and `ishl` can raise exceptions (use the provided function `c0_arith_error` to generate a message). Please refer to the C0 language specification for important details.

We have omitted negation `-x`, which the compiler can simulate with `0-x`, and bitwise negation `~x`, which the compiler can simulate with `x^(-1)`.

2.3 Local Variables

We can move data generically between local variables and the stack, because all primitive types can fit into the size of a reference type (`c0_value`, which is defined as `void*`, is 8 bytes on 64-bit systems). The instruction operand `<i>` is one byte following the opcode `0x15` or `0x36` in the instruction stream. Because this is the only way to access a local variable, each function can have at most 256 local variables, which includes the function arguments.

```
0x15 vload <i>    S -> S, v          (v = V[i])
0x36 vstore <i>   S, v -> S          (V[i] = v)
```

2.4 Constants

We can push constants onto the operand stack. There are three different forms: (1) a constant `null` which is directly coded into the instruction, (2) a small *signed* (byte-sized) constant `` which is an instruction operand and must be sign-extended to 32 bits, and (3)

a constant stored in the constant pool. For the latter, we distinguish constants of primitive type from those of reference type because they are stored in different pools.

The two constant loading instructions `ildc` and `aldc` take two unsigned bytes as operands, which must be combined into an unsigned integer index for the appropriate constant pool. The integer pool stores the constants directly, and the index given to `ildc` is an index into the integer pool. The string pool is one large array of character strings, each terminated by `'\0'`. The index given to `aldc` indicates the position of the first character; its address is therefore of type `char*` in C and understood by C as a string.

```
0x01 aconst_null   S -> S, null:*
0x10 bipush <b>    S -> S, x:w32      (x = (w32)b, sign extended)
0x13 ildc <c1,c2>  S -> S, x:w32      (x = int_pool[(c1<<8)|c2])
0x14 aldc <c1,c2>  S -> S, a:*        (a = &string_pool[(c1<<8)|c2])
```

2.5 Control Flow

Each instruction implicitly increments the program counter by the number of bytes making up the instruction. Control flow instructions change this by jumping to another instruction under certain conditions. The addressing is relative to the address of the branch instruction. The offset is a *signed* 16 bit integer that is given as a two-byte operand to the instruction. It must be signed so we can branch backwards in the program. Note that `if_cmpeq` and `if_cmpne` can be used to compare either integers or pointers for equality or inequality, whereas the other comparisons only make sense on integers. The `nop` “no-op” instruction has no effect.

```
0x00 nop          S -> S

0x9F if_cmpeq <o1,o2> S, v1, v2 -> S      (pc = pc+(o1<<8|o2) if v1 == v2)
0xA0 if_cmpne <o1,o2> S, v1, v2 -> S      (pc = pc+(o1<<8|o2) if v1 != v2)
0xA1 if_icmplt <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x < y)
0xA2 if_icmpge <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x >= y)
0xA3 if_icmpgt <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x > y)
0xA4 if_icmple <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x <= y)

0xA7 goto <o1,o2>   S -> S                (pc = pc+(o1<<8|o2))

0xBF athrow        S, a:* -> S            (c0_user_error(a))
0xCF assert        S, x:w32, a:* -> S      (c0_assertion_failure(a) if x == 0)
```

2.6 Function Calls and Returns

Function calls come in two forms: invoking a C0 function defined in the same bytecode file and invoking a library function defined in C. In either case, generic arguments `v1` through `vn` are passed on the operand stack and consumed. The C0VM implementation must guarantee that the result `v` is pushed onto the stack when the function returns. For functions returning

`void`, a dummy value is pushed onto the operand stack to provide a uniform interface to functions.

Function information is stored in the function pool or native pool, both of which are addressed by the instruction operand consisting of two bytes, which must be reconstituted into an *unsigned* 16 bit quantity indexing into the appropriate pool.

```
0xB8 invokestatic <c1,c2> S, v1, v2, ..., vn -> S, v
                    (function_pool[c1<<8|c2] = g, g(v1,...,vn) = v)
0xB0 return      ., v -> .   (return v to caller)
```

When invoking a C0 function (instruction `invokestatic`) we have to preserve the program counter `pc` as a return address, the current local variable array `V` and the current operand stack `S`. This is the information in a *frame* which is pushed onto a global call stack. Then we set the `pc` to the beginning of the code for the called function `g`, allocate a new array of local variables, and initialize it with the function arguments from the old operand stack. We also create a new empty operand stack for use in the called function.

When processing a `return` instruction we restore the `pc`, the local variable array `V` and the operand stack `S` from the last frame on the call stack. We also need to arrange that the return value is popped from the current operand stack and pushed onto the operand stack of the frame we return to. Some temporary data structures may need to be deallocated at this point.

The `main` function always has index 0 in the function pool and takes 0 arguments. After reading the file, setting up appropriate data structures, etc., your C0VM implementation should start executing byte code at the beginning of this function and at the end print the final value.

In this assignment, you should use two instances of the abstract data type of stacks, one holding frames (the call stack) the other holding operands (the operand stack).¹ Each of them is used just using pushes and pops, as the stack interface dictates. Remember, your stacks or arrays will have to share data of different types, namely primitive types and reference types.

A reasonable mapping to C implementation types would be `int32_t` for primitive types and `void*` for reference types, implementing respectively the types `w32` and `*` in the description of the instructions. This allows you to store all C0VM values of small type as type `c0_value` (i.e., `void *`) and cast them to the appropriate type for the operations you need to perform on them and to cast the result back when finished.

¹There are a number of other plausible implementation paths for the two stacks. One possibility is inspired by the system call stack for languages such as C. In this strategy there is one global array storing return addresses, local variables, and (in the case of the C0VM) also an area to use as the operand stack. This implementation would not use our abstract datatype of stacks, but implement the stack as an array with a so-called stack pointer which is the current top of the stack. Of course, it is not really treated exactly like an abstract stack, because the access to local variables violates the pure stack discipline. Another possibility is to have a single operand stack, shared throughout the execution, rather than local ones for each frame. This is possible because the C0VM specification requires that when a function returns, its operand stack must be empty.

2.7 Native Function Calls

Native function calls have the same form as C0 function calls, but the two-byte instruction argument indexes into the native pool, rather than the function pool.

```
0xB7 invokenative <c1,c2> S, v1, v2, ..., vn -> S, v
```

The value `native_pool[c1<<8|c2]` is an index i into a separate runtime structure, the `native_function_table`. From there you retrieve the address of a function g which has type

```
c0_value (*g)(c0_value*);
```

The type `c0_value*` should be read as an array of pointers of type `c0_value`, which indicates generic data. It also returns a value of generic type. In order to call this function you have to construct an array of length n and store arguments `v1` through `vn` at indices 0 through $n - 1$, and then invoke the function g on this array. The result has to be pushed back onto the operand stack.

Native function calls do not therefore involve explicitly managed stack frames. Of course, your abstract machine implementation is using the system stack, so when you call the library function, the library function also uses the system stack, rather than any stack managed explicitly by your virtual machine.

The mapping between native library functions and their indices into the native function table is given as a series of `NATIVE_*` macros in the file `c0vm_c0ffi.h`.

2.8 Memory Allocation, Load, and Store

Besides function calls, the trickiest aspect of the C0VM implementation is the management of the C0 runtime heap. There are two basic options. One is to allocate on the C runtime heap (that is available to your C0VM implementation as it runs) one very large array and perform C0 allocations inside this array. The type of references would then be `int`, values denoting array indices. Alternatively, you can satisfy each C0 allocation request separately by allocating a sufficient amount of space on the C runtime heap, using C pointers to implement C0VM references. The latter option is advantageous for the purpose of calling native functions because the C runtime heap is shared between the running C0VM implementation and the C0 bytecode program that it executes. In either case, the implementation of allocation must take care to initialize all memory requested by C0 to all zeros, as required by the semantics.

```
0xBB new <s>          S -> S, a:*      (*a is now allocated, size <s>)
```

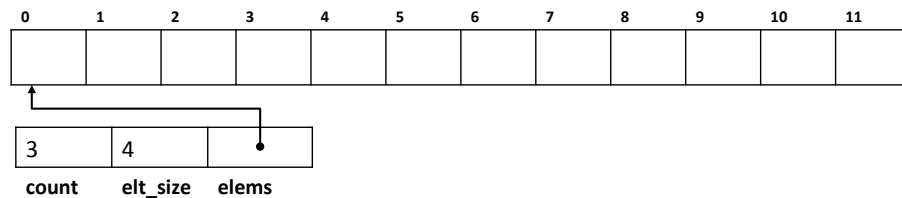
The `new <s>` instruction allocates memory for holding data of size s , and returns the address of that memory. Here s is an unsigned byte, expressing the memory size in bytes. The data size is computed statically by the C0 compiler. For example, the C0 expression `alloc(int)` would translate to `new 4`, while `alloc(struct b)` would translate to `new n`, where n is the size of a `struct b` in memory in bytes, which is always known at compile time.


```

0xBC newarray <s>  S, n:w32 -> S, a:*      (a[0..n) now allocated)
0xBE arraylength  S, a:* -> S, n:w32      (n = \length(a))

```

The `newarray <s>` instruction allocates memory for an array, each of whose elements has size s . The number n of elements to allocate is passed on the stack since it cannot in general be known at compile time. C0 arrays are represented as pointers to a struct `c0_array_header` that has three fields. The first, `count`, is the number of elements in the array, and is determined at runtime (this is `n` in the instruction above). The second, `elt_size`, represent the number of bytes in a single element; the compiler is able to determine this at compile time based on the type (this is `s` in the instruction above).



The third field is `elems`, a void pointer to the beginning of the memory allocated for the actual array, which contains $n*s$ bytes. The length of an array, stored in the `count` field, can be retrieved with the `arraylength` C0VM instruction. Every C0 array you encounter in the VM will be a pointer to one of these structs.²

The picture above is an array of three objects with `elt_size` of 4, perhaps a C0 integer. The third integer (array offset 2 in the C0 array) is stored in the bytes labeled 8-11. Reading from or writing to this integer requires two steps:

1. First, we need to compute the actual memory address of the integer. If we cast the void pointer in the `elems` field to a `char` pointer named `arr`, then the address where we should look up the third integer is `&arr[2*4]` - the address 8 bytes into the block of allocated memory (array offset 2×4 bytes per array element; recall that a `char` is defined to be one byte).
2. Now that we have the address of the integer, we can cast our pointer to an integer pointer: reading from or writing to this address will read or write exactly the 4 bytes we need once the address has been cast to `int*`.

In the C0VM, these two steps are split into different VM instructions for address arithmetic and loading from or storing to a computed address. Address arithmetic comes in two forms:

```

0x62 aaddf <f>      S, a:* -> S, (a+f):*  (a != NULL; f field offset)
0x63 aadds          S, a:*, i:w32 -> S, (a->elems+s*i):*
                                (a != NULL, 0 <= i < \length(a))

```

The `aadds` instruction performs computes the address of an array element. The operand a on the stack must be the address of an array, and the operand i must be a valid index for this array. The C0VM must issue an error message and abort if i is not a valid index, which can

²It is possible for a C0 array to be NULL; the VM should treat NULL like an array with length 0.

be determined from the stored array length; use the provided function `c0_memory_error` to issue this error. We then use the element size s stored with the array to compute the address of the i th element.

A simpler process happens when you access a field of a struct with `aaddf`. The compiler computes an offset f (an unsigned one-byte quantity), and you must add f bytes to the pointer a you have been given. The address a may be `NULL`, and if this is the case, then you must raise a memory error by calling the provided function `c0_memory_error`.

Note that one `aadds` instruction is necessary for every array access, even if we access the element at index 0. When accessing a field of a struct, we may not need to perform any offsets the address of the field we're accessing is the same as the address of the struct.

Once we have calculated an address that holds a value of small type (either a primitive type or a reference type), we can load it from memory or store something at the given location. For primitive types, we use the `imload` instruction and `imstore` instructions, for reference types the `amload` and `amstore` instructions.

```
0x2E imload   S, a:* -> S, x:w32   (x = *a, a != NULL, load 4 bytes)
0x2F amload   S, a:* -> S, b:*     (b = *a, a != NULL, load address)
0x4E imstore  S, a:*, x:w32 -> S   (*a = x, a != NULL, store 4 bytes)
0x4F amstore  S, a:*, b:* -> S     (*a = b, a != NULL, store address)
```

For character arrays, we also need to be able to load an individual byte and cast it as a value of C0VM type `w32`, zero-extending it. Zero-extension works here because the original C0 type of such a value must have been `char` whose range is limited to 7 bits. This is done by the `cmload` instruction; `cmstore` performs the opposite, masking the given value of type `w32` to 7 bits.

```
0x34 cmload   S, a:* -> S, x:w32   (x = (w32)(*a), a != NULL, load 1 byte)
0x55 cmstore  S, a:*, x:w32 -> S   (*a = x & 0x7f, a != NULL, store 1 byte)
```

As for the address arithmetic instructions you must check that the address a of a load or store instruction is non-`NULL` and raise a memory error if a is `NULL`.

3 Bytecode File Format

The bytecode file, usually with extension `.bc0`, is produced by the `cc0` compiler when invoked with the `-b` or `--bytecode` flag. In order to allow you to easily read bytecode, and also write your own bytecode, the binary file is coded in hexadecimal form, where two-digit bytes are separated by whitespace. In addition, the file may contain comments starting with `#` and extending to the end of the line.

We describe the format as pseudo-structs, where we use the types described below. For multi-byte types, each byte is given separately by two hexadecimal digits, with the most significant byte first.

```
u4 - 4 byte unsigned integer
u2 - 2 byte unsigned integer
u1 - 1 byte unsigned integer
```

i4 - 4 byte signed (two's complement) integer
 fi - struct function_info, defined below
 ni - struct native_info, defined below

The size of some arrays is variable, depending on earlier fields. These are only arrays conceptually, of course. In the file, all the information is just stored as sequences of bytes separated by whitespace.

```

struct bc0_file {
    u4 magic;                # magic number, always 0xc0c0ffee
    u2 version+arch;        # version number and architecture
    u2 int_count;           # number of integer constants
    i4 int_pool[int_count]; # integer constants
    u2 string_count;        # number of characters in string pool
    u1 string_pool[string_count]; # adjacent '\0'-terminated strings
    u2 function_count;      # number of functions
    fi function_pool[function_count]; # function info
    u2 native_count;        # number of native (library) functions
    ni native_pool[native_count]; # native function info
};

struct function_info {
    u2 num_args;            # number of arguments, V[0..num_args)
    u2 num_vars;           # number of variables, V[0..num_vars)
    u2 code_length;        # number of bytes of bytecode
    u1 code[code_length];  # bytecode
};

struct native_info {
    u2 num_args;            # number of arguments, V[0..num_args)
    u2 function_table_index; # index into table of library functions
};

```

We are providing code that reads bytecode files and marshals the information into similar internal C structures.

4 Programming Tasks and Coding Advice

There are many complexities in implementing a virtual machine, especially one that is rich enough so it can execute all of C0! Fortunately, some of the complexities (such as parsing the bytecode file) are taken care of by code we are providing, but others remain. You will complete the code in `c0vm.c`.

The following are suggested strategies to help you work effectively throughout this project.

4.1 Testing

We have provided a few test cases in `tests/`, but it is more effective to write your own. Write a small file, say `test.c0` and compile it with `cc0 -b test.c0`, which will create a bytecode file `test.bc0`. Then run it with `./c0vm test.bc0`. Compare your answers with the ones you get with `cc0 -o mytest mytest.c0` and `./mytest`. The script `testvm` automates this process:

```
% ./testvm tests/iadd.c0
Files iadd.c0.c0out and iadd.c0.bc0out are identical
```

4.2 Incremental Implementation

Implement a subset of the instruction set and test your C0VM implementation on code that only uses the subset. Generate some test cases using `cc0 -b` from simple C0 sources, or use some of the supplied examples that use limited instructions. You should recognize instructions that are valid but not in your subset and give a “*not yet implemented*” message and returning rather than aborting in the same way as for other errors. Test one stage thoroughly before moving on. After extending the machine, first make sure the old, simple examples still run correctly, a process called *regression testing*. The stages follow our discussion of the instruction set:

Task 1 (15 pts) [2.1,2.2] Initialize the following variables correctly in the `execute` function in the `c0vm.c` file: `S`, `P`, `pc`.

Add code to handle arithmetic instructions, plus `bipush`, `swap`, and `return`. (The `return` instruction need only simulate returning from `main` for now, see `PROG_HINTS.txt` in the code handout.) C0 programs with only a `main` function returning an expression made of small constants can be used to test these capabilities, e.g.,

```
int main() {
    return 15 * ((1<<10) - 24) + 122;
}
```

Task 2 (15 pts) [2.3,2.4] Add code to deal with local variables and constants; you’ll need to initialize the variable `V` to something better than `NULL`. C0 source files containing straight-line code using variables and large constants can be used to test these capabilities, e.g.,:

```
int main() {
    int x = 15122;
    int y = x * x;
    return y;
}
```

Task 3 (20 pts) [2.5] Add code to handle `goto` and conditionals (e.g., `if_icmpge`). Now you should be able to execute loops, as in

```
int main () {
    int i; int sum = 0;
    for (i = 15; i <= 122; i++)
        sum += i;
    return sum;
}
```

Task 4 (15 pts) [2.6,2.7] Add function calls (`invokestatic`, `invokenative`); you'll probably want to initialize the variable `callStack` to something better than `NULL` and use `callStack` to manage the call stack in some form. You will also need to revisit `return` (see `PROG_HINTS.txt` in the code handout). You may want to focus on ordinary C0 function calls (`invokestatic`, `return`) before moving on to native function calls (`invokenative`).

Now your `main` function can call auxiliary functions, such as the ubiquitous recursive factorial function, and library functions that print output:

```
int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n-1);
}
int main () {
    printint(factorial(15));
    println(" is the factorial of 15");
    return 0;
}
```

Task 5 (10 pts) [2.8] Add the C0 heap, where arrays and structs are allocated. After this, you should be able to run arbitrary C0 code, including your peg solitaire solver, your images assignment, and your string buffers.

4.3 Assertions

Ideally, we would establish invariants of the bytecode that we read from a file to make sure no runtime memory or type error occurs. In the JVM this is referred to as *bytecode verification*. Unfortunately, the current bytecode format does not provide enough information to do this. Even if it did, it would be a major project in itself. So you have to fall back on dynamic checks. These checks come in two categories:

1. The usual checks on the runtime structure of your own code, verifying that pointers are not null, etc.
2. Checks that the C0 bytecode you have read in behaves properly.

Some of the checks in the second category are mandated:

- (a) The C0 program must not dereference the C0 null pointer or perform pointer arithmetic on it.

- (b) The C0 program must not access memory outside the bounds of a C0 array.
- (c) The C0 program must not perform illegal integer division (division by 0, or the min int divided by -1).
- (d) The C0 program must not shift left or right by a number < 0 or ≥ 32 .

If you encounter these runtime errors, you should produce error messages using the provided functions

- `void c0_memory_error(char *err)` (for memory-related errors)
- `void c0_arith_error(char *err)` (for division or shift-related errors)

By calling these functions, which are declared in `c0vm_abort.h`, you make it clear that this is a runtime error in the bytecode you are executing and not a bug in your machine.

The first category of checks should in principle be redundant. For example, the `cc0` compiler should never produce a bytecode file that jumps to an invalid address. Nevertheless, bytecode written by hand or a bug in the `cc0` compiler or your VM could lead to such issues. Since C does not guarantee detection of such incorrect jumps or accesses, your code should do that using appropriate `assert` statements, or `ASSERT`, `REQUIRES`, and `ENSURES`. Then, if the bytecode itself or your virtual machine implementation has a bug, it will be discovered as soon as unexpected incorrect behavior occurs. The macro annotations are recommended so that there is no undue overhead for correct code when your machine has been debugged.

4.4 The Proliferation of Types

One common issue in writing virtual machines and similar (almost) self-referential code is the proliferation of different types at different levels, sometimes with the same name. It will be important for you to understand the various level of types involved, whether they are signed or unsigned, and what they refer to. In addition you will have to cast between types for various operations.

The following table might help. The C type is only recommended, not required.

C0 type	C0VM type	C type (recommended)
<code>int</code>	<code>w32</code>	<code>int32_t</code>
<code>bool</code>	<code>w32</code>	<code>int32_t</code>
<code>char</code>	<code>w32</code>	<code>int32_t</code>
<code>t[]</code>	<code>*</code>	<code>void*</code>
<code>t*</code>	<code>*</code>	<code>void*</code>
<code>struct s</code>	(none)	(none)
<code>function</code>	function pool index	<code>struct function_info</code>
<code>library fn</code>	native pool index	<code>struct native_info</code>

Sometimes you will have to cast the C type that represent the C0VM types to a form where they are appropriate for the operation to be performed on them. An example of this are the arithmetic operations, which are reliably performed on unsigned ints so we have to cast to and from these types.

4.5 Manage Your Time Well

Remember that this homework is worth **75 points**, and the last 25 points are *much* more difficult than the first 25 points. You should plan on working on this for an hour or two every day, so you can ask for help early on if you need it. **Don't wait until the last few days!** Post general questions on Piazza (e.g., questions about the C0VM specification, wording of tasks, requirements for handin, etc.).

A C0VM Instruction Reference

What follows is a reference for the C0VM bytecode, which is also given as part of the handout (`c0vm-ref.txt`). Every line that describes an operation has the following format:

```
0xYZ omne          S -> S' (other effect)
```

where `0xYZ` is the opcode in hex, `omne` is the operation mnemonic, and the remaining text describes the effect of the operation. The description includes the effect on the stack (e.g. transform stack `S` into stack `S'`) and any other effects (e.g. modify the program counter).

`S` = operand stack

`V` = local variable array, `V[0..num_vars)`

Instruction operands:

`<i>` = local variable index (unsigned)

`` = byte (signed)

`<s>` = element size in bytes (unsigned)

`<f>` = field offset in struct in bytes (unsigned)

`<c>` = `<c1,c2>` = pool index = `(c1<<8|c2)` (unsigned)

`<o>` = `<o1,o2>` = pc offset = `(o1<<8|o2)` (signed)

Stack operands:

`a : *` = address ("reference")

`x, i, n` : `w32` = 32 bit word representing an int, bool, or char ("primitive")

`v` = arbitrary value (`v:*` or `v:w32`)

Stack operations

```
0x59 dup          S, v -> S, v, v
```

```
0x57 pop          S, v -> S
```

```
0x5F swap         S, v1, v2 -> S, v2, v1
```

Arithmetic

```

0x60 iadd      S, x:w32, y:w32 -> S, x+y:w32
0x7E iand      S, x:w32, y:w32 -> S, x&y:w32
0x6C idiv      S, x:w32, y:w32 -> S, x/y:w32
0x68 imul      S, x:w32, y:w32 -> S, x*y:w32
0x80 ior       S, x:w32, y:w32 -> S, x|y:w32
0x70 irem      S, x:w32, y:w32 -> S, x%y:w32
0x78 ishl      S, x:w32, y:w32 -> S, x<<y:w32
0x7A ishr      S, x:w32, y:w32 -> S, x>>y:w32
0x64 isub      S, x:w32, y:w32 -> S, x-y:w32
0x82 ixor      S, x:w32, y:w32 -> S, x^y:w32

```

Local Variables

```

0x15 vload <i>    S -> S, v      v = V[i]
0x36 vstore <i>  S, v -> S      V[i] = v

```

Constants

```

0x01 aconst_null S -> S, null:*
0x10 bipush <b>  S -> S, x:w32      (x = (w32)b, sign extended)
0x13 ldc <c1,c2> S -> S, x:w32      (x = int_pool[(c1<<8|c2)])
0x14 ldc <c1,c2> S -> S, a:*        (a = &string_pool[(c1<<8|c2)])

```

Control Flow

```

0x00 nop          S -> S
0x9F if_cmpeq <o1,o2> S, v1, v2 -> S      (pc = pc+(o1<<8|o2) if v1 == v2)
0xA0 if_cmpne <o1,o2> S, v1, v2 -> S      (pc = pc+(o1<<8|o2) if v1 != v2)
0xA1 if_icmplt <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x < y)
0xA2 if_icmpge <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x >= y)
0xA3 if_icmpgt <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x > y)
0xA4 if_icmple <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x <= y)
0xA7 goto <o1,o2>   S -> S              (pc = pc+(o1<<8|o2))
0xBF athrow        S, a:* -> S          (c0_user_error(a))
0xCF assert        S, x:w32, a:* -> S    (c0_assertion_failure(a) if x == 0)

```

Functions

```

0xB8 invokestatic <c1,c2> S, v1, v2, ..., vn -> S, v
                          (function_pool[c1<<8|c2] => g, g(v1,...,vn) = v)
0xB0 return      ., v -> .      (return v to caller)
0xB7 invokenative <c1,c2> S, v1, v2, ..., vn -> S, v
                          (native_pool[c1<<8|c2] => g, g(v1,...,vn) = v)

```


Memory

```
0xBB new <s>      S -> S, a:*          (*a is now allocated, size <s>)
0xBC newarray <s> S, n:w32 -> S, a:*   (a[0..n) now allocated)
0xBE arraylength S, a:* -> S, n:w32   (n = \length(a))

0x62 aaddf <f>   S, a:* -> S, (a+f):*
                                     (a != NULL; f field offset in bytes)
0x63 aadds       S, a:*, i:w32 -> S, (a->elems+s*i):*
                                     (a != NULL, 0 <= i < \length(a))

0x2E imload     S, a:* -> S, x:w32   (x = *a, a != NULL, load 4 bytes)
0x2F amload     S, a:* -> S, b:*     (b = *a, a != NULL, load address)
0x4E imstore    S, a:*, x:w32 -> S   (*a = x, a != NULL, store 4 bytes)
0x4F amstore    S, a:*, b:* -> S     (*a = b, a != NULL, store address)

0x34 cmload     S, a:* -> S, x:w32   (x = (w32)(*a), a != NULL, load 1 byte)
0x55 cmstore    S, a:*, x:w32 -> S   (*a = x & 0x7f, a != NULL, store 1 byte)
```