15-122: Principles of Imperative Computation

Recitation Week 12 — Tom Cortina

C0VM: Review

Consider the following program written in C0:

```
int exp(int b, int e)
//@requires e >= 0;
{
  if (e== 0)
    return 1;
  else
    return b * exp(b, e-1);
}
int main() {
  int x = 5;
  int y = 2;
  int z = exp(x, y);
  return z;
}
```

If we compile this to bytecode using the `-b` flag, we get the following (abbreviated) file:

```
C0 C0 FF EE       # magic number
00 09             # version 4, arch = 1 (64 bits)

00 00             # int pool count

00 00             # string pool total size

00 02             # function count
# function_pool

#<main>
00 00             # number of arguments = 0
00 03             # number of local variables = 3
00 14             # code length = 20 bytes
10 05    # bipush 5        # 5
36 00    # vstore 0        # x = 5;
10 02    # bipush 2        # 2
36 01    # vstore 1        # y = 2;
15 00    # vload 0         # x
15 01    # vload 1         # y
B8 00 01 # invokestatic 1  # exp(x, y)
36 02    # vstore 2        # z = exp(x, y);
15 02    # vload 2         # z
B0       # return          #
```

```
#<exp>
00 02             # number of arguments = 2
00 02             # number of local variables = 2
00 1E             # code length = 30 bytes
15 01    # vload 1         # e
10 00    # bipush 0        # 0
9F 00 06 # if_cmpeq +6     # if (e == 0) goto <00:then>
A7 00 09 # goto +9         # goto <01:else>
# <00:then>
10 01    # bipush 1        # 1
B0       # return          #
A7 00 11 # goto +17        # goto <02:endif>
# <01:else>
15 00    # vload 0         # b
15 00    # vload 0         # b
15 01    # vload 1         # e
10 01    # bipush 1        # 1
64       # isub            # (e - 1)
B8 00 01 # invokestatic 1  # exp(b, (e - 1))
68       # imul            # (b * exp(b, (e - 1)))
B0       # return          #
# <02:endif>

00 00             # native count
```

The C0VM has five variables that keep track of the state of the program as it is executing:

`S` - the operand stack

`P` - an array holding the bytes of the current function being executed ("the program")

`pc` - index into array `P` indicating the location of the next instruction opcode to execute

`V` - array of local variables (including any arguments first)

`callStack` - a stack of frames holding the 4 elements above for functions "on hold" as the current function executes

Let's trace how the C0VM simulates the program. Initially, the function `main` is loaded as the current "program" and we have:

```
S = []
P = [10, 05, 36, 00, 10, 02, 36, 01, 15, 00, 15, 01, B8, 00, 01, 36, 02, 15, 02, B0]
pc = 0
V = [-, -, -]
callStack = []
```

Note that `S` and `callStack` are empty, `P` contains the bytes for the main function (SHOWN IN HEX), `pc` indicates that the next instruction to execute is at index 0 in the "program", and `V` has space for three local variables (x, y, z).

C0VM goes to P[0] (= P[pc]) and executes `10 05 # bipush 5` to yield:

```
S = [5],  P = [10, 05, 36, 00, ..., 15, 02, B0],  pc = 2,  V = [-, -, -],
callStack = []
```

Next, C0VM goes to P[2] and executes `36 00 # vstore 0`, setting x (i.e. V[0]) to 5:

```
S = [],  P = [10, 05, 36, 00, ..., 15, 02, B0],  pc = 4,  V = [5, -, -],
callStack = []
```

C0VM executes `10 02 # bipush 2` to yield:

```
S = [2],  P = [10, 05, 36, 00, ..., 15, 02, B0],  pc = 6,  V = [5, -, -],
callStack = []
```

C0VM executes `36 01 # vstore 1`, setting y (i.e. V[1]) to 2:

```
S = [],  P = [10, 05, 36, 00, ..., 15, 02, B0],  pc = 8,  V = [5, 2, -],
callStack = []
```

C0VM executes `15 00 # vload 0` to yield:

```
S = [5],  P = [10, 05, 36, 00, ..., 15, 02, B0],  pc = 10,  V = [5, 2, -],
callStack = []
```

C0VM executes `15 01 # vload 1` to yield:

```
S = [5 2], P = [10, 05, 36, 00, ..., 15, 02, B0],  pc = 12,  V = [5, 2, -],
callStack = []
```

(Note that stack S is shown from bottom to top, reading left to right.)

Now, C0VM executes `B8 00 01 # invokestatic 1`. When this happens, the current state of S, P, pc and V are stored as a *frame* on `callStack`. Then these variables are reset for the new function with an empty operand stack, an array with the bytes of the `exp` function (i.e. function #1), a program counter reset to 0, and an array of local variables initialized using the top 2 values from the old operand stack popped off into the V array (since `exp` requires 2 arguments, note the order). Once this instruction is completed, we have:

```
S = []
P = [15, 01, 10, 00, 9F, 00, 06, A7, 00, 09, 10, 01, B0, A7, 00, 11, 15, 00, 15, 00,
     15, 01, 10, 01, 64, B8, 00, 01, 68, B0]
pc = 0
V = [5, 2]
callStack = [{S,P,pc,V}] = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

Note that when C0VM goes to execute the next instruction, the program counter has been reset back to 0 and is indexing the first byte of the new function that has been loaded into the P array.

C0VM executes `15 01 # vload 1` to yield:

```
S = [2],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 2,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

C0VM executes `10 00 # bipush 0` to yield:

```
S = [2 0], P = [15, 01, 10, 00, ..., 68, B0],  pc = 4,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

C0VM executes `9F 00 06 # if_cmpeq +6`. This pops off the top two values from the operand stack and compares them to see if they're equal. If so, then it adds 6 to the program counter to "jump" to another location in the program. Otherwise, it adds 3 to the program counter (since the instruction has 3 bytes) to go on to the next instruction as usual. In this case, the two values are not equal, so 3 is added to the program counter:

```
S = [],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 7,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

C0VM executes `A7 00 09 # goto +9`. This always adds the given amount (+9) to the program counter to jump to the "else" part of the "program":

```
S = [],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 16,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

C0VM executes `15 00 # vload 0` to yield:

```
S = [5],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 18,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

C0VM executes `15 00 # vload 0` to yield:

```
S = [5 5],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 20,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

C0VM executes `15 01 # vload 1` to yield:

```
S = [5 5 2],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 22,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

C0VM executes `10 01 # bipush 1` to yield:

```
S = [5 5 2 1],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 24,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

C0VM executes `64 # isub` to yield:

```
S = [5 5 1],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 25,  V = [5, 2],
callStack = [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

Now, C0VM executes `B8 00 01 # invokestatic 1`. This is a call to the same function (recursion!) but the action is the same as before. The current state of S, P, pc and V are stored in a frame and these variables are reset for the "new" function, which happens to be the same function again. Note how the top two values on the *old* operand stack are popped off to set the local variables for the new V array.

```
S = []
P = [15, 01, 10, 00, 9F, 00, 06, A7, 00, 09, 10, 01, B0, A7, 00, 11, 15, 00, 15, 00,
     15, 01, 10, 01, 64, B8, 00, 01, 68, B0]
pc = 0
V = [5, 1]
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `15 01 # vload 1` to yield:

```
S = [1],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 2,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `10 00 # bipush 0` to yield:

```
S = [1 0],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 4,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `9F 00 06 # if_cmpeq +6`. In this case, the two values are not equal, so 3 is added to the program counter:

```
S = [],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 7,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `A7 00 09 # goto +9`:

```
S = [],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 16,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `15 00 # vload 0` to yield:

```
S = [5], P = [15, 01, 10, 00, ..., 68, B0],  pc = 18,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `15 00 # vload 0` to yield:

```
S = [5 5], P = [15, 01, 10, 00, ..., 68, B0],  pc = 20,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `15 01 # vload 1` to yield:

```
S = [5 5 1],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 22, V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `10 01 # bipush 1` to yield:

```
S = [5 5 1 1],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 24,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

C0VM executes `64 # isub` to yield:

```
S = [5 5 0],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 25,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

Now, C0VM executes `B8 00 01 # invokestatic 1` :

```
S = []
P = [15, 01, 10, 00, 9F, 00, 06, A7, 00, 09, 10, 01, B0, A7, 00, 11, 15, 00, 15, 00,
     15, 01, 10, 01, 64, B8, 00, 01, 68, B0]
pc = 0
V = [5, 0]
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 1] } ]
```

C0VM executes `15 01 # vload 1` to yield:

```
S = [0],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 2,  V = [5, 0],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 1] } ]
```

C0VM executes `10 00 # bipush 0` to yield:

```
S = [0 0],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 4,  V = [5, 0],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 1] } ]
```

C0VM executes `9F 00 06 # if_cmpeq +6`. In this case, the two values *are* equal, so 6 is added to the program counter:

```
S = [],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 10,  V = [5, 0],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 1] } ]
```

C0VM executes `10 01 # bipush 1` to yield:

```
S = [1],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 12,  V = [5, 0],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 1] } ]
```

Now, the C0VM executes `B0 # return`. Note that the return value for this function was already pushed on top of the operand stack immediately before the return is executed using the `bipush` operation. In this case, the return value is temporarily popped off of S, then the top frame on the call stack (shown at the end of the sequence of frames above) is used to restore the previous values of S, P, pc, and V, and then the return value is pushed back on the restored operand stack:

```
S = [5 1],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 28,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

Note that the program counter is restored back to 28, where we left off when we performed the `invokestatic` operation.

Next, the C0VM executes `68 # imul`:

```
S = [5],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 29,  V = [5, 1],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] }
               { [5], [15, 01, 10, 00, ..., 68, B0], 28, [5, 2] } ]
```

The C0VM executes `B0 # return` again. As before, the return value is temporarily popped off of S, then the top frame on the call stack (shown at the end of the sequence of frames above) is used to restore the previous values of S, P, pc, and V, and then the return value is pushed back on the restored operand stack:

```
S = [5 5],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 28,  V = [5, 2],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

Next, the C0VM executes `68 # imul`:

```
S = [25],  P = [15, 01, 10, 00, ..., 68, B0],  pc = 29,  V = [5, 2],
callStack =  [ { [], [10, 05, 36, 00, ..., 15, 02, B0], 15, [5, 2, -] } ]
```

The C0VM executes `B0 # return` again. Note that in this case, we are returning back to main now:

```
S = [25],  P = [10, 05, 36, 00, ..., 15, 02, B0], pc = 15, V = [5, 2, -],
callStack =  []
```

The C0VM executes `36 02 # vstore 2`, setting z (i.e. V[2]) to 25 $= 5^2$:

```
S = [], P = [10, 05, 36, 00, ..., 15, 02, B0],  pc = 17,  V = [5, 2, 25],
callStack =  []
```

The C0VM executes `36 02 # vload 2` (to prepare for the return from main):

```
S = [25],  P = [10, 05, 36, 00, ..., 15, 02, B0],  pc = 19,  V = [5, 2, 25],
callStack =  []
```

The C0VM executes `B0 # return`. In this case, since the call stack is empty, this must be a return from main, so the program ends and the top of the stack (25) is returned as the final answer.

Trace through this carefully and you will see how each of the C0VM operations in this example works, and you will also see how function calls (and recursive calls) work at the system level!

C0VM: New Things (Class Examples 5, 6, and 7)

Example 5 shows how fields of a struct are allocated and accessed in bytecode. Allocation is done with the `new` opcode which requires the number of bytes required for the struct. Access is done using the `aaddf` opcode. First, the address of the start of the struct is pushed on the operand stack by the previous opcode. Then the `aaddf` opcode adds the given byte offset to this address to yield the address of the specific field of the struct. Finally, the `imload` opcode takes the address off the operand stack and replaces it with the value at that address (i.e. the contents of the field of the struct).

Example 6 shows how arrays are allocated and accessed in bytecode. Allocation is done with the `newarray` opcode which requires the size of the element for the array. The number of elements required should be on top of the operand stack prior to running `newarray`. Access is performed using the `aadds` opcode. Before this opcode is executed, the address of the start of the array and the index required must be pushed on the operand stack by prior opcodes (see example). Then the `aadds` opcode pops these off, computes the address of the desired element and pushes this address back on the operand stack. Again, the `imload` opcode can be used to get the data at the address given on top of the operand stack to complete an array access (e.g. see the access for `A[99]`). Figure out how `imstore` works.

Native functions and the string pool are illustrated in Example 7. This is left for you to investigate!