

## C SCI 316 (Kong): TinyJ Assignment 3

This assignment is to be submitted *no later than* Friday, May 21.

**IMPORTANT:** Even though this due date is after the 1:40 section's final exam, *your final exam may possibly include questions that test your ability to write code that needs to be written to complete this assignment.* Also, if euclid unexpectedly goes down after 6 p.m. on the due date, the deadline will *not* be extended. Try to submit before noon that day, and on an earlier day if possible.

See p. 5 for information about the *late* submission deadline for Lisp and TinyJ assignments.

The assignment is to complete the TJasn program so it will correctly *execute* the TinyJ virtual machine (VM) instructions that it generates. For each VM instruction, there will be a corresponding file `*instr.java` in your `TJasn/virtualMachine` directory. Every such file has an `execute()` method. In `STOPinstr.java`, `ADDTOPTRinstr.java`, `HEAPALLOCinstr.java`, `NOPorDISCARDVALUEinstr.java`, and `READINTinstr.java`, that method is written for you. In the other 29 `*instr.java` files, the body of the `execute()` method contains a comment of the form `/* ???????? */` *which you need to replace with code that executes the corresponding VM instruction:* Your code must make appropriate changes to the expression evaluation stack [`CodeInterpreter.EXPRSTACK[]`], data memory [`TJ.data[]`], and VM registers [`CodeInterpreter.ESP`, `.PC`, `.FP`, `.ASP`, etc.], and must produce correct output (if any). See the example at the beginning of the "How to Do the Assignment" section below.

### Some Information About the Implementation of the TinyJ Virtual Machine

In `CodeInterpreter.java`, the variables `PC`, `IR`, `ESP`, `FP`, and `ASP` (see lines 14 – 18) represent VM registers; the array `EXPRSTACK[]` (line 19) represents the expression evaluation stack.

`PC` contains the code memory address of the next instruction to be fetched for execution. (When that instruction is fetched, it is put into `IR` for execution.)

`ESP` stores a count of the number of items that are currently on the expression evaluation stack—if `ESP > 0`, then `EXPRSTACK[ESP - 1]` is the topmost element of the stack.

`FP` is a pointer to the data memory location at offset 0 in the currently executing method activation's stackframe. (Thus `FP+k` will be a pointer to the location at offset  $k$  in that stackframe.)

`ASP` is a pointer to the *first unused* location in the stack-dynamically allocated part of data memory—i.e., `ASP` points to the first location above the currently allocated locations in that part of data memory. `ASP` must therefore be increased/decreased by  $k$  when  $k$  stackframe locations are allocated/deallocated.

In `TJ.java`, the `ArrayList` `generatedCode` (line 30) and the array `data[]` (line 26) represent code and data memory, respectively. `TJ.generatedCode.get(a)` returns the instruction stored in code memory at address  $a$ . `TJ.data[a]` represents the data memory location whose address is  $a$ .

The VM's fetch-execute loop is in the `interpretCode()` method of `CodeInterpreter.java` (line 48); this is called by `start()` (line 247). Its most important lines are:

```
75  IR = TJ.generatedCode.get(oldPC = PC++);    and    77  IR.execute();
```

Line 75 *fetches* the instruction stored (in code memory) at the address in `PC` and increments `PC`.<sup>†</sup> Line 77 *executes* the instruction that has just been fetched.

<sup>†</sup>The address of the instruction that is fetched is stored into `oldPC`—so when that instruction is being executed, and immediately afterwards, `oldPC` will contain the address of the instruction. When a debugging stop occurs—see the "How to Debug If You are Stumped" section below—the VM uses the address in `oldPC` to determine which instruction has just been executed. `PC - 1` will usually be equal to `oldPC`, but will sometimes be different because `PC` may be changed by execution of **JUMP**, **JUMPONFALSE**, **CALLSTATMETHOD**, or **RETURN**.

## Installation of DEMO Instruction Classes

Installation of TinyJ Assignment 2 should have installed the instruction classes from my solution (the "DEMO instruction classes") into your TJsolclasses/TJasn/virtualMachine directory.

**Before you work on this assignment, use the following commands to copy these DEMO instruction classes into your TJasn/virtualMachine directory:**

```
cp TJsolclasses/TJasn/virtualMachine/*instr.class TJasn/virtualMachine
[on euclid and venus, working directory = your home directory]
copy /y TJsolclasses\TJasn\virtualMachine\*instr.class TJasn\virtualMachine
[in a cmd.exe window on a PC, working directory = c:\316java]
```

Then, since the DEMO classes have correct execute() methods, you should be able to *execute* TinyJ programs (e.g., the 16 CS316ex\*.java files) as follows:

```
java -cp . TJasn.TJ CS316exk.java k.out
```

As an example, you can enter: `java -cp . TJasn.TJ CS316ex9.java 9.out`

You will see: Using the following DEMO instruction classes:

```
PUSHSTATADDR PUSHNUM SAVETOADDR INITSTKFRM LOADFROMADDR PASSPARAM CALLSTATMETHOD ADD
CHANGESIGN WRITEINT WRITELNOP WRITESTRING PUSHLOCADDR EQ JUMPONFALSE JUMP SUB RETURN
```

This says that your program is using my DEMO versions of these instruction classes, rather than your own versions of the classes. The program will ask:

Want debugging stop or post-execution dump? (y/n)

Respond by entering **n** and the program will begin to execute the CS316ex9 program. (The output should be the same as the output you get when you compile CS316ex9.java using `javac CS316ex9.java` and then enter `java CS316ex9` to execute the generated byte codes.)

## How to Do the Assignment

This assignment assumes you know what each TinyJ VM instruction should do when it is executed—if you are not clear as to what certain instructions should do when executed, refer back to the “Effects of Executing Each TinyJ Virtual Machine Instruction” pages at:

<https://euclid.cs.qc.cuny.edu/316/Memory-allocation-VM-instruction-set-and-hints-for-asn-2.pdf>

As an example of how to complete an execute() method, consider the execute() method of ADDInstr.java. As Java (unlike C++) is guaranteed to evaluate expressions from left to right, the `/* ???????? */` in ADDInstr.java can be replaced with the following statement:

```
EXPRSTACK[--ESP-1] += EXPRSTACK[ESP];
```

After making this change, use the command

```
javac -cp . TJasn/virtualMachine/ADDInstr.java
```

to recompile ADDInstr.java and produce a file ADDInstr.class which will replace the DEMO version of ADDInstr.class you copied into TJasn/virtualMachine. The new version of ADDInstr.class can then be tested by executing a TinyJ program whose virtual machine code uses the ADD instruction. For example, you could enter:

```
java -cp . TJasn.TJ CS316ex9.java 9.out
```

When you do this you should find that ADD is no longer on the list of DEMO instruction classes reported by TJasn.TJ! As before, when you are asked `Want debugging stop or post-execution dump? (y/n)` you should enter **n**. [The `javac` and `java` commands above assume that your working directory is your home directory on venus or euclid, and `c:\316java` on a PC.]

I suggest you work on the other instruction classes in the following order. First, work on MUL, SUB, DIV, MOD, and CHANGESIGN, which are used by, e.g., CS316ex0.java. Next, work on NOT, AND, OR, EQ, NE, LT, GT, LE, GE, PUSHNUM, WRITELNOP, WRITEINT, WRITESTRING, JUMP, JUMPONFALSE, PUSHSTATADDR, LOADFROMADDR, and SAVETOADDR, which are used by, e.g., CS316ex3.java. Finally, work on PUSHLOCADDR, PASSPARAM, CALLSTATMETHOD, INITSTKFRM, and RETURN, all of which are used by, e.g., CS316ex9.java.

Each time you complete the execute() method of a file xxxxinstr.java, it might well be a good idea to immediately compile that .java file and then (if there are no compilation errors)

immediately test your `execute()` method by using `TJasn.TJ` to execute a `CS316exk.java` program whose virtual machine code uses the instruction in question. If `TJasn.TJ` fails to execute that `CS316exk` program correctly, then your new `execute()` method is bugged and you need to fix it. [If you want to work on another instruction's `execute()` method first, then copy my DEMO version of `xxxinstr.class` back from your `TJsolclasses/TJasn/virtualMachine` directory into your `TJasn/virtualMachine` directory, so it replaces your bugged version.] If the `CS316exk` program is executed correctly, save a backup copy of the file that contains your new `execute()` method—e.g., put the copy on **venus** if you are working on **euclid**.

## How to Debug If You are Stumped

Suppose that you have just written the `execute()` method of `LOADFROMADDRinstr.java` (and that any other `execute()` methods you may have written earlier all work). After compiling `LOADFROMADDRinstr.java` [`javac -cp . TJasn/virtualMachine/LOADFROMADDRinstr.java`], if your program `TJasn.TJ` executes some TinyJ input file [say, `CS316ex3.java`] incorrectly, try the following debugging method:

Run `TJasn.TJ` on that same input file again, **but set the TinyJ virtual machine to stop as soon as it has executed a LOADFROMADDR instruction.** Do this as follows:

```
java -cp . TJasn.TJ CS316ex3.java 3a.out
Want debugging stop or post-execution dump? (y/n)  Y
Enter MINIMUM no. of instructions to execute before debugging stop.
(Enter -1 to get a post-execution dump but no debugging stop.):  0
Stop after executing what instruction? (e.g., PUSHNUM)
(Enter * to stop after executing just 0 instructions.):  LOADFROMADDR
```

Next, **copy my DEMO version** of `LOADFROMADDRinstr.class` back from your `TJsolclasses/TJasn/virtualMachine` directory into your `TJasn/virtualMachine` directory, and then repeat the above with the same debugging stop settings. **But this time use a different output file name**—`3b.out` instead of `3a.out`, say. After these two runs, the file `3a.out` will contain dumps of the contents of the virtual machine's registers and memory locations ***immediately before and immediately after*** execution of **your** version of `LOADFROMADDRinstr`'s `execute()`, whereas `3b.out` will contain the same information for **my** version of `LOADFROMADDRinstr`'s `execute()`. Use `diff -c` or `fc.exe /n` to compare `3a.out` and `3b.out`. Their dumps ***before*** execution of the **LOADFROMADDR** instruction should be identical. (If they differ, one of the `execute()` methods you wrote earlier is bugged!) But the dumps ***after*** execution of **LOADFROMADDR** may well be different, in which case you should be able to see what your version of `LOADFROMADDRinstr`'s `execute()` did wrong.

In the event that the dumps ***after*** execution of **LOADFROMADDR** are also identical, make a note of the *total number of instructions executed before execution was stopped*—this number will be shown near the bottom of the "after" dumps. If this number is, say, **27**, then repeat the above process, ***but this time set the virtual machine to execute  $27+1 = 28$  instructions***, and to then stop after it executes any **LOADFROMADDR** instruction:

```
Want debugging stop or post-execution dump? (y/n)  Y
Enter MINIMUM no. of instructions to execute before debugging stop.
(Enter -1 to get a post-execution dump but no debugging stop.):  28
Stop after executing what instruction? (e.g., PUSHNUM)
(Enter * to stop after executing just 28 instructions.):  LOADFROMADDR
```

## Use of CodeInterpreter.POINTERTAG

In the TinyJ virtual machine, a pointer to *data memory* address *a* is represented by the integer `a + CodeInterpreter.POINTERTAG`, where `CodeInterpreter.POINTERTAG` is a constant equal to `0x7FFF0000 = 01111111 11111111 00000000 00000000`. [This depends on the fact that the machine's data memory addresses are small enough to be stored in the lower two bytes of an int.]

When executing **PUSHSTATADDR** *a*, remember to push the *pointer* corresponding to address *a*. In other words, remember to push  $a + \text{POINTERTAG}$ .

When executing **LOADFROMADDR**, remember that the topmost stack item is a *pointer*: If *p* is on top of the stack, then the data memory address to load from is  $p - \text{POINTERTAG}$ .

When executing **SAVETOADDR**, remember that the second item on the stack is a *pointer*: If *p* is the second item on the stack, then the data memory address to write to is  $p - \text{POINTERTAG}$ .

When executing **CALLSTATMETHOD**, **INITSTKFRM**, **PASSPARAM**, and **RETURN**, remember that ASP is a pointer and refers to the data memory address  $\text{ASP} - \text{POINTERTAG}$ .

When executing **PUSHLOCADDR** *a*, remember that FP *already contains a pointer*, so you should *not* add POINTERTAG to FP+*a*. You also should *not* add POINTERTAG to FP before storing it during execution of **INITSTKFRM**, and should *not* subtract POINTERTAG when retrieving FP from the stackframe during execution of **RETURN**. Similarly, since FP and ASP both store pointers, do *not* add or subtract POINTERTAG when copying from ASP into FP or vice versa during execution of **INITSTKFRM** and **RETURN**.

When executing **WRITESTRING** *a b*, remember that *a* and *b* are ordinary data memory addresses, not pointers: Do *not* subtract POINTERTAG from *a* and *b*.

POINTERTAG is *not used* with code memory addresses: Never subtract POINTERTAG from PC, and never add POINTERTAG to a code memory address.

## How to Test Your Solution

When you have completed all the incomplete `execute()` methods and recompiled each of the modified `*instr.java` files, test your program on different source files (which should at least include the 16 `CS316exk.java` files) as follows:

```
java -cp . TJasn.TJ TinyJ-source-file-name output-file-name
```

For example: `java -cp . TJasn.TJ CS316ex12.java 12.out`

Your working directory should be your *home directory* on *venus* or *euclid*, and `c:\316java` on a PC.

When it asks you `Want debugging stop or post-execution dump? (y/n)` you should enter **n**.

The runtime behavior should always be the same as when the same source file is compiled using `javac` and the resulting `.class` file is executed by a Java virtual machine.

Also check that, for each `CS316exk.java` source file, a post-execution dump produced after execution of 1000 instructions is always the same for your program as for my solution. Recall that you can run *my solution* as follows:

```
java -cp TJsolclasses:. TJasn.TJ CS316exk.java k.sol [on euclid or venus]
java -cp "TJsolclasses;." TJasn.TJ CS316exk.java k.sol [on a PC]
```

## How to Submit Your Solution

This assignment is to be submitted *no later than Friday, May 21*. [Reminder: Even though this due date is after the 1:40 section's final exam, *your final exam may possibly include questions that test your ability to write code that needs to be written to complete this assignment*. Also, if *euclid* unexpectedly goes down after 6 p.m. on the due date, the deadline will *not* be extended. Try to submit before noon that day, and on an earlier day if possible.]

See the next page for information regarding the *late* submission deadline for Lisp and TinyJ assignments.

This assignment counts **2%** towards your grade if the grade is computed using rule A. You may work with up to two other students. As stated on p. 3 of the first-day announcements document, when two or three students work together *each student must submit separately*.

If you have been working on *euclid*, submit by following just the four steps 1, 2, 4, and 5 below; *you must NOT do step 3!!* If you have been working on *venus* or your PC (*NOT euclid!*), do all of 1 – 5 below.

These instructions assume your working directory on **venus** / **euclid** is your home directory; the instructions relating to a PC assume you are working in a **cmd** window with `c:\316java` as your working directory.

1. Remove any incomplete **\*instr.java** files you have **NOT** successfully completed from the TJasn/virtualMachine directory of *the machine you have been working on*. Then, at the beginning of each **\*instr.java** you have successfully completed, add a comment that gives your name and the names of the students you worked with (if any). As usual, you may work with up to two other students, but see the remarks about this on p. 3 of the first-day announcements document.
2. Enter the following command on **euclid**:  

```
cp TJsolclasses/TJasn/virtualMachine/*instr.class TJasn/virtualMachine
```
3. **This step does not apply if you have been working on euclid: You must not enter the rm and scp commands given below if you have been working on euclid!** If, and only if, you have been working on **venus** or your own computer (**NOT euclid!**), enter the following command on **euclid**:  

```
rm TJasn/virtualMachine/*instr.java
```

Then logout from **euclid** and enter the following command at the shell prompt or in a **cmd** window on the machine that contains the **\*instr.java** files you wish to submit; this command assumes your working directory is your home directory on **venus** or `c:\316java` on your PC:  

```
scp TJasn/virtualMachine/*instr.java xxxxx316@euclid.cs.qc.cuny.edu:TJasn/virtualMachine
```

Here **xxxxx316** means your **euclid** username, and the `:` after `.edu` is needed.
4. On **euclid**, recompile all the **\*instr.java** files you have completed, as follows:  

```
euclid> javac TJasn/virtualMachine/*instr.java
```

If any **\*instr.java** file cannot be compiled on **euclid**, you will receive no credit for this assignment. (In view of step 1 above, this should not happen.)
5. **Test your program on euclid.** (See the section "How to Test Your Solution" above.)

**IMPORTANT: Do NOT open any of your submitted **\*instr.java** files in an editor on euclid after the due date, unless you are resubmitting a corrected version of your solution as a *late* submission. Also do not execute `mv`, `chmod`, or `touch` with your submitted file as an argument after the due date. (It is OK to view a submitted file using the `less` file viewer after the due date.)**


As stated on page 3 of the first-day announcements document, you are required to keep a backup copy of each submitted file on **venus**, and another copy elsewhere. You can enter the following command on **euclid** to put copies of your submitted files on **venus**:

```
scp TJasn/virtualMachine/*instr.java your venus username@149.4.211.180:TJasn
```

(This puts the **\*instr.java** files in the TJasn directory on **venus** rather than in TJasn/virtualMachine.)

To email copies of your submitted files to yourself, first enter the following command on **euclid**:

```
alpine -attachlist $(find TJasn -name *instr.java -newer TJasn) $USER
```

When **alpine** starts up, look at the list of attached files to make sure it includes all of your submitted files. Then type  to send the email.

### **Late Submission Deadline for Lisp and TinyJ Assignments**

While I *expect* **euclid** to stay up after 6 p.m. on Friday, May 21, there is no guarantee that it will stay up. If **euclid** stays up, then late submissions of Lisp and TinyJ assignments (including corrected submissions) will be accepted until noon on Tuesday, May 25; but this late submission deadline will not be extended if **euclid** goes down at any time after 6 p.m. on May 21. Note that, if **euclid** goes down at any time after 6 p.m. on May 21, then it might well not be brought back up until after noon on May 25, in which case students would not be able to submit any assignments they were planning to submit when **euclid** went down! To avoid the risk of not being able to submit, make any late / corrected submissions of Lisp and TinyJ assignments no later than 6 p.m. on Friday, May 21.



## Hints for TinyJ Assignment 3

Before you work on the assignment, read page 1 above. When writing the `execute()` method for an instruction, if you are not clear as to what that instruction should do when it is executed, refer back to the “Effects of Executing Each TinyJ Virtual Machine Instruction” pages at:

<https://euclid.cs.qc.cuny.edu/316/Memory-allocation-VM-instruction-set-and-hints-for-asn-2.pdf>

### Remarks on the `execute()` Methods You Have to Write

#### 1. Instructions That Do Not Deal With Pointers to Data Memory Locations and Have No Operands

##### ADD, SUB, DIV, MUL, MOD, AND, OR

As explained on p. 2, **ADD**'s `execute()` method can be implemented as follows:

```
EXPRSTACK[--ESP-1] += EXPRSTACK[ESP];
```

The `execute()` methods for **SUB**, **DIV**, **MUL**, **MOD**, **AND**, and **OR** can be written analogously. Use 1 and 0 to represent **true** and **false**; then the `execute()` methods for **AND** and **OR** can use `&=` and `|=` where **ADD**'s `execute()` uses `+=`.

##### LE, GE, LT, GT, EQ, NE

One simple way to write the `execute()` methods for these instructions is to use conditional expressions. For example, **LE**'s `execute()` method could be implemented as follows:

```
EXPRSTACK[--ESP-1] = (EXPRSTACK[ESP-1] <= EXPRSTACK[ESP]) ? 1 : 0;
```

##### CHANGESIGN, NOT

These instructions should change `EXPRSTACK[ESP-1]` without changing `ESP`. (Note: `*= -1` is a concise way to negate an **int** variable; `^= 1` is a concise way to change its value from 1 to 0 or vice versa.)

##### Writelnop, Writeint

These instructions should print to `System.out`: **Writelnop** should print a newline, and **Writeint** should print an integer that is popped off `EXPRSTACK`. (Note that `EXPRSTACK[--ESP]` represents a value that is popped off `EXPRSTACK`.)

#### 2. Instructions That Do Not Deal With Pointers to Data Memory Locations but Have Operands

The operand of any `OneOperandInstruction` (such as `PUSHNUMinstr` or `JUMPinstr`) is in its `operand` field (which it inherits from `OneOperandInstruction`). The two operands of a `WRITESTRINGinstr` are in its `firstOperand` and `secondOperand` fields (which it inherits from `TwoOperandInstruction`).

Recall from p. 1 that `PC` represents the VM's program counter: It stores the *code memory* address of the location from which the next instruction will be fetched for execution.

**PUSHNUM 17** should push its operand onto `EXPRSTACK` (i.e., put 17 into `EXPRSTACK[ESP++]`).

**JUMP 17** should put its operand 17 into `PC`, as `PC` represents the program counter.

**JUMPONFALSE 17** should look at a value that is popped off `EXPRSTACK` (i.e., look at the value `EXPRSTACK[--ESP]`) and put its operand 17 into `PC` just if the popped value is 0.

**WRITESTRING 7 17** should print (to `System.out`) the characters that are stored in the data memory locations with addresses  $\geq$  its `firstOperand` 7 but  $\leq$  its `secondOperand` 17. These locations are represented by:

```
TJ.data[7], TJ.data[8], ..., TJ.data[16], TJ.data[17]
```

You must perform a `(char)` cast on the value in each location before printing it.

### 3. Instructions That Deal With Pointers to Data Memory Locations

Before you write the `execute()` methods for these instructions, you should read the section “Use of `CodeInterpreter.POINTERTAG`” on pp. 3 – 4.

Also, recall from p. 1 that *FP stores a pointer to the data memory location at offset 0 in the currently executing method activation's stackframe*. (Thus  $FP + k$  will be a pointer to the location at offset  $k$  in that stackframe.)

**PUSHSTATADDR 7** should push  $7 + \text{POINTERTAG}$  (i.e., push a pointer to the data memory location whose address is 7) onto `EXPRSTACK`; in other words, it should put the pointer  $7 + \text{POINTERTAG}$  into `EXPRSTACK[ESP++]`.

**PUSHLOCADDR 7** should push  $FP + 7$  (i.e., push a pointer to the data memory location at offset 7 in the currently executing method activation's stackframe) onto `EXPRSTACK`; in other words, it should put the pointer  $FP + 7$  into `EXPRSTACK[ESP++]`.

**LOADFROMADDR** If  $p$  is the pointer in `EXPRSTACK[ESP-1]`, execution of this instruction should copy the value in the data memory location to which  $p$  points (i.e., copy the value `TJ.data[p-POINTERTAG]`) into `EXPRSTACK[ESP-1]`; the copied value should overwrite the pointer  $p$ .

**SAVETOADDR** If  $v$  is the value in `EXPRSTACK[ESP-1]` and  $p$  is the pointer in `EXPRSTACK[ESP-2]`, then this instruction should store  $v$  into the data memory location to which  $p$  points (i.e., it should store  $v$  into `TJ.data[p-POINTERTAG]`), and should also decrease `ESP` by 2 to pop  $v$  and  $p$  off `EXPRSTACK`.

### 4. Instructions Associated with Calls of Static Methods and Return from Called Methods

Recall from p. 1 that *ASP stores a pointer to the first unused location in the stack-dynamically allocated part of data memory*—i.e., *ASP* points to the *first location above* the currently allocated locations in that part of data memory.

So *ASP* must be increased / decreased by  $k$  when  $k$  stackframe locations are allocated / deallocated.

We will write **S-PUSH**  $y$  for `TJ.data[ASP++ - POINTERTAG] = y;`

We will write **S-POP**  $y$  for `y = TJ.data[--ASP - POINTERTAG];`

**51: PASSPARAM** should **S-PUSH** a value that is popped off `EXPRSTACK`  
—i.e., it should **S-PUSH** `EXPRSTACK[--ESP]`

**52: CALLSTATMETHOD 671** should **S-PUSH** `PC` [saves return addr (here, **53**) into new frame]  
and then Set `PC` to 671 [transfers control to the called method's code]

**671: INITSTKFRM 7** should **S-PUSH** `FP` [saves caller's `FP` at offset 0 in the new frame]  
and then Set `FP` to `ASP-1` [makes `FP` point to offset 0 in the new frame]  
and then Increase `ASP` by 7 [allocates space for callee's local variables]

**713: RETURN 4** should Set `ASP` to `FP+1` [deallocates space used by callee's variables]  
and then **S-POP** `FP` [restores caller's `FP`]  
and then **S-POP** `PC` [puts the saved return address into `PC`]  
and then Decrease `ASP` by 4 [deallocates space used by formal parameters]