

# Function Calls in Real and Virtual Machines

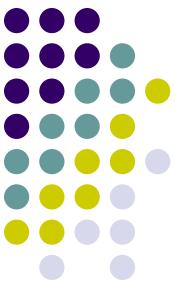


- Function calls in real and virtual machines tend to be more complicated than interpreting function calls in high-level languages due to the fact that there is typically no notion of scope.
- We begin by studying function calls in our bytecode machine and then look at function calls on the Intel chip.

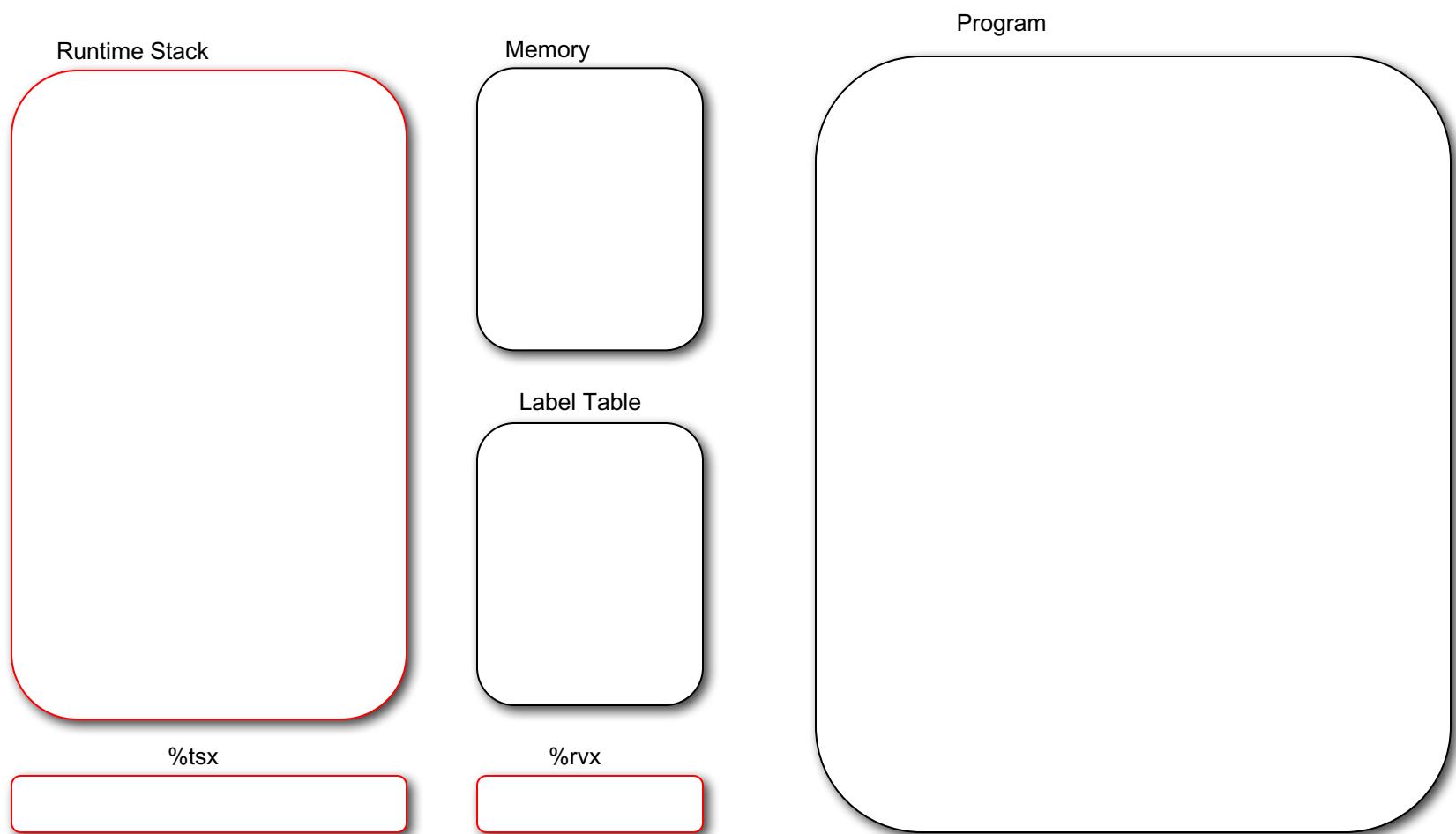


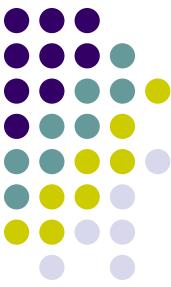
# Exp2Bytecode

- In order to facilitate function calls on our bytecode machine we add the following:
  - A runtime stack and a ‘top of stack’ register (%tsx)
  - A ‘return value’ register (%rvx)
  - A set of instructions that manipulate the runtime stack
    - pushv -- push a value on the stack
    - popv -- pop a value off the stack
    - pushf -- push a stack frame
    - popf -- pop a stack frame
- Instructions for calling and returning from functions
  - call -- jumps to function
  - return -- continues execution at the instruction after the call instruction



# Virtual Machine Design





# Exp2Bytecode

- We also introduce the idea of *indirect addressing* in order to access stack locations
- We let `%tsx` be the top of stack register, in order access the second to top element we write: `%tsx[-1]`
  - We can read the value, e.g.
    - store a `%tsx[-1]`
  - We can write a value to that location, e.g.
    - store `%tsx[-1] 3`



# Exp2Bytecode

```
prog : instr_list

instr_list : labeled_instr instr_list
| empty

labeled_instr : label_def instr

label_def : NAME '::'
| empty

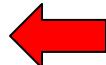
instr : PRINT opt_string exp ';'
| INPUT opt_string storable ';'
| STORE storable exp ';'
| JUMPT exp label ';'
| JUMPF exp label ';'
| JUMP label ';'
| CALL label ';'
| RETURN ';'
| PUSHV exp ';'
| POPV opt_storable ';'
| PUSHF size ';'
| POPF size ';'
| STOP ';'
| NOOP ';'

opt_string : STRING
| empty

opt_storable : storable
| empty

size : exp

label : NAME
```



```
exp : '+' exp exp
| '-' exp exp
| '-' exp
| '*' exp exp
| '/' exp exp
| EQ exp exp
| LE exp exp
| '!' exp
| '(' exp ')'
| storable
| NUMBER

storable : var
| RVX
| TSX opt_offset

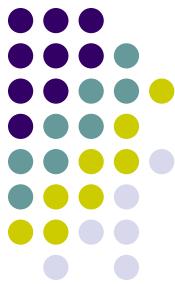
opt_offset : '[' offset ']'
| empty

offset : exp

var : NAME
```

exp2bytecode\_gram.py

# Exp2Bytecode – Abstract Machine



- Exp2bytecode introduces ‘storable’ in order to deal with different classes of memory.

```
prog : instr_list

instr_list : labeled_instr instr_list
| empty

labeled_instr : label_def instr

label_def : NAME ':'
| empty

instr : PRINT opt_string exp ';'
| INPUT opt_string storable ';'
| STORE storable exp ';'
| JUMPT exp label ';'
| JUMPF exp label ';'
| JUMP label ';'
| CALL label ';'
| RETURN ';'
| PUSHV exp ';'
| POPV opt_storable ';'
| PUSHF size ';'
| POPF size ';'
| STOP ';'
| NOOP ';'

opt_string : STRING
| empty

opt_storable : storable
| empty

size : exp

label : NAME
```

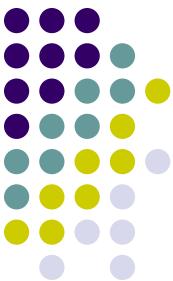
```
exp : '+' exp exp
| '-' exp exp
| '-' exp
| '*' exp exp
| '/' exp exp
| EQ exp exp
| LE exp exp
| '!' exp
| '(' exp ')'
| storable
| NUMBER

storable : var
| RVX
| TSX opt_offset

opt_offset : '[' offset ']'
| empty

offset : exp

var : NAME
```



# Exp2Bytecode

```
# Lexer for Exp2bytecode

from ply import lex

reserved = {
    'print' : 'PRINT',
    'input' : 'INPUT',
    'store' : 'STORE',
    'jumpT' : 'JUMPT',
    'jumpF' : 'JUMPF',
    'jump' : 'JUMP',
    'call' : 'CALL',
    'return': 'RETURN',
    'pushv' : 'PUSHV',
    'popv' : 'POPV',
    'pushf' : 'PUSHF',
    'popf' : 'POPF',
    'stop' : 'STOP',
    'noop' : 'NOOP'
}

literals = ['!', ':', ';', '+', '-', '*', '/', '(', ')', '[', ']']

tokens = ['NAME', 'NUMBER', 'EQ', 'LE', 'RVX', 'TSX', 'STRING'] +\
    list(reserved.values())

t_EQ = '=='
t_LE = '<='
t_RVX = '%rvx'
t_TSX = '%tsx'
t_ignore = ' \t'
```

exp2bytecode\_lexer.py

```
def t_NAME(t):
    r'[a-zA-Z][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'NAME')      # Check for reserved words
    return t

def t_NUMBER(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t

def t_STRING(t):
    r'".*"'
    t.value = t.value[1:-1] # strip the quotes
    return t

def t_NEWLINE(t):
    r'\n'
    pass

def t_COMMENT(t):
    r'#./*'
    pass

def t_error(t):
    print("Illegal character %s" % t.value[0])
    t.lexer.skip(1)

# build the lexer
lexer = lex.lex(debug=0)
```



# Calling Convention

- In order to execute function calls the calling function and the called function have to agree upon how parameters are passed and return values are returned -- this is called the *calling convention*.
- Our calling convention is fairly straight forward:
  - Actual arguments are passed by pushing them on the stack in *reverse* order
  - Return values are returned in the %rvx register
  - Each function is responsible for maintaining the integrity of the runtime stack, that is, if it pushed something on the stack then it has to also remove it (this is not true for all calling conventions!)



# Example Program

- The first program we consider is written in Cuppa3 as follows:

```
declare add(a,b) {  
    return a+b  
}  
  
declare x = 3;  
declare y = 2;  
put add(x,y);
```



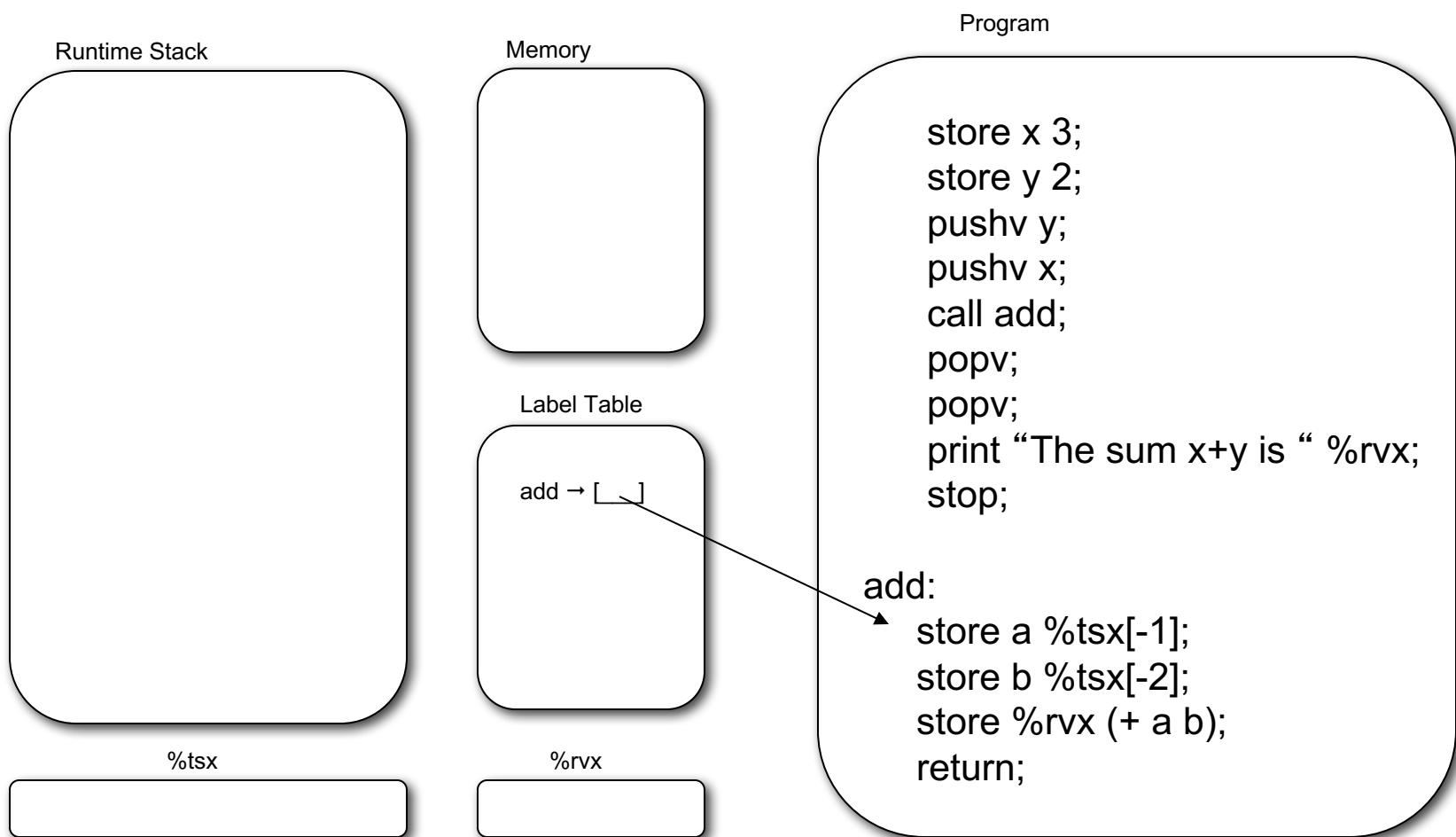
# Example Program

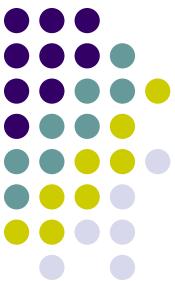
```
declare add(a,b) {  
    return a+b  
}  
  
declare x = 3;  
declare y = 2;  
put add(x,y);
```

```
store x 3;  
store y 2;  
pushv y;          # push second argument onto stack  
pushv x;          # push first argument onto stack  
call add;         # push current address onto stack and jump to function  
popv;             # pop first argument  
popv;             # pop second argument  
print "The sum x+y is " %rvx;  
stop;  
  
add:  
    store a %tsx[-1];      # get value of first argument  
    store b %tsx[-2];      # get value of second argument  
    store %rvx (+ a b);   # store the sum in the 'return value register'  
    return;                # pop return address off stack and return to that address
```

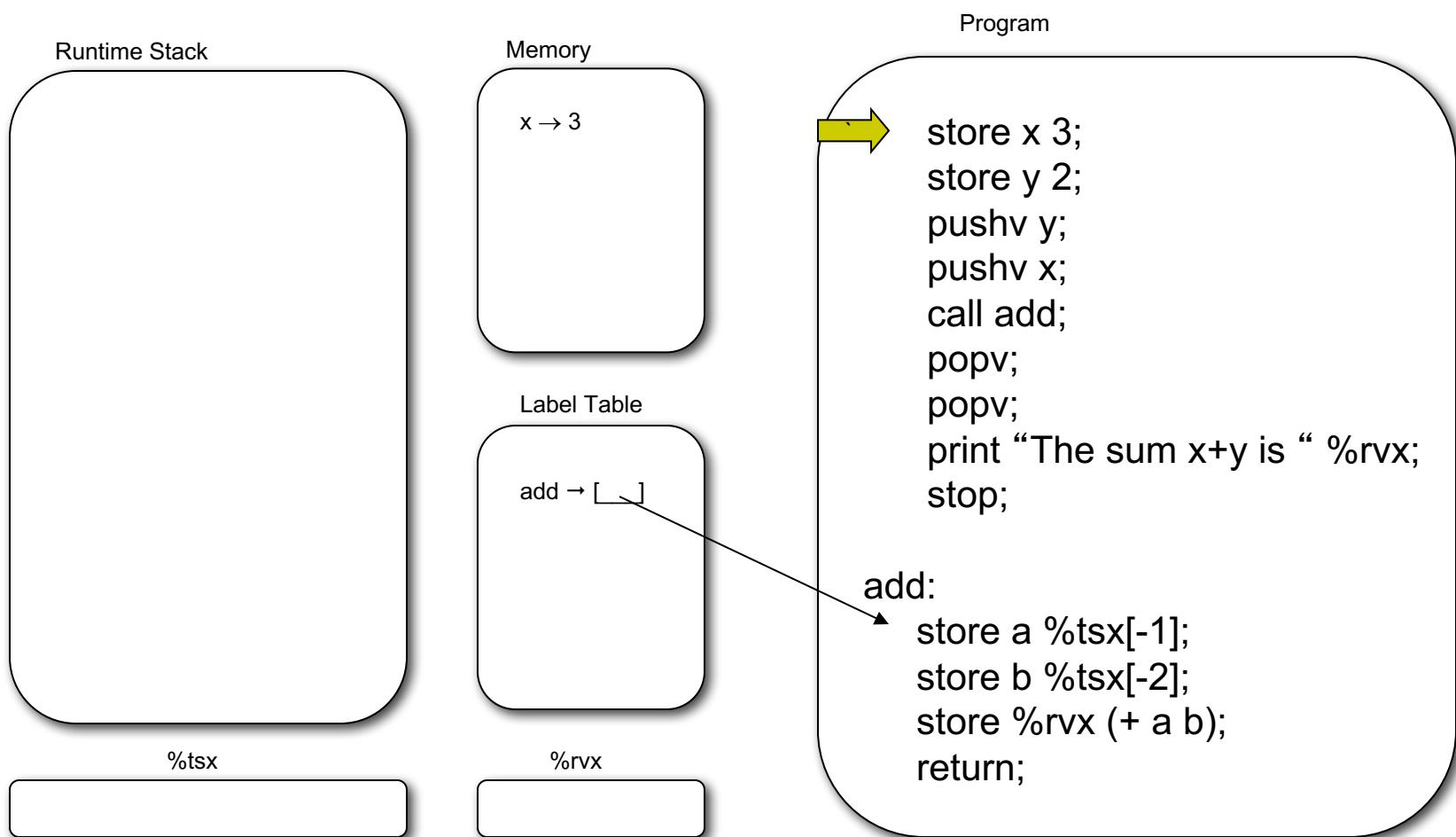


# Virtual Machine Design



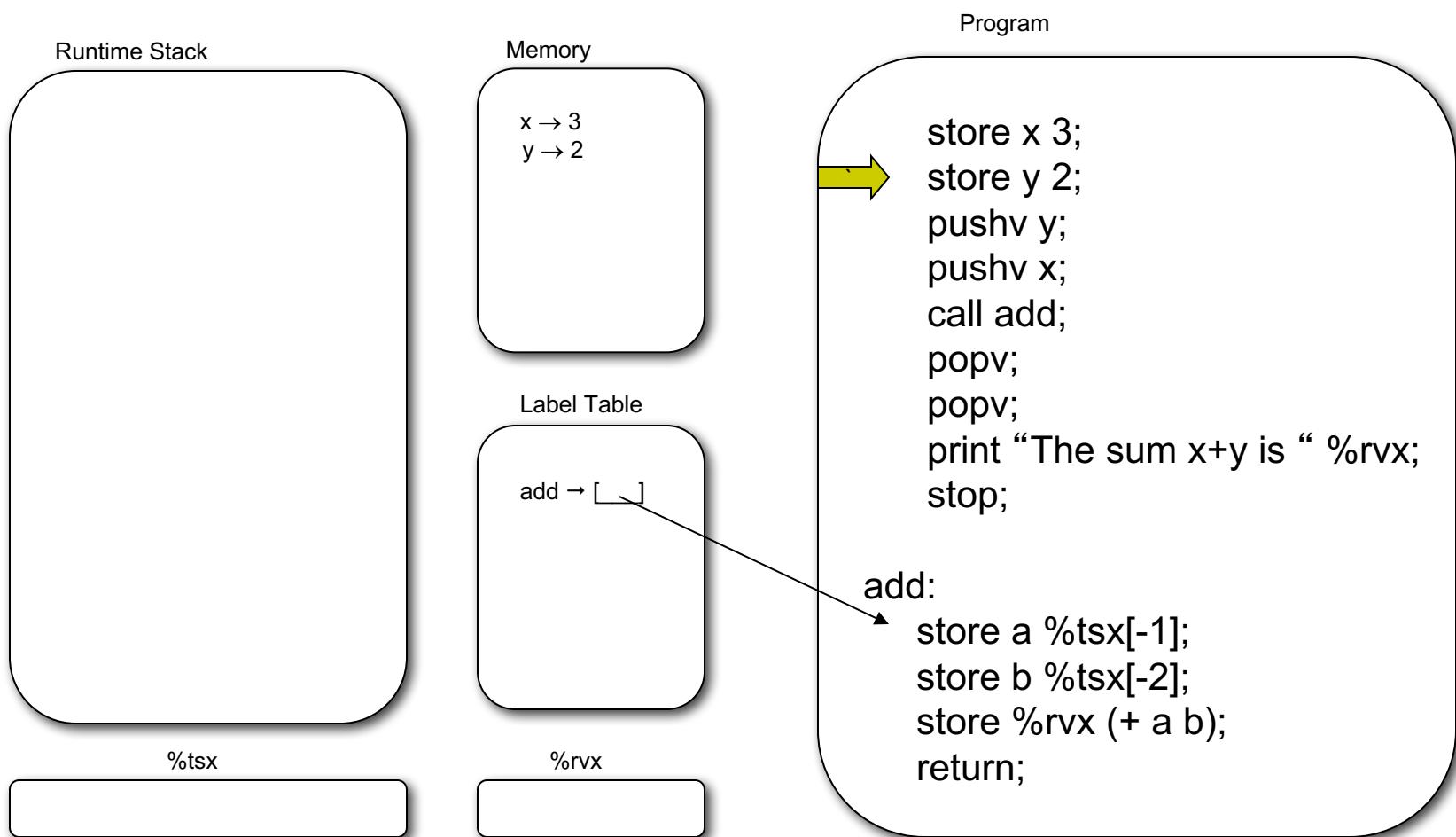


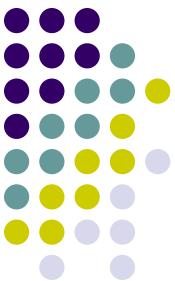
# Virtual Machine Design



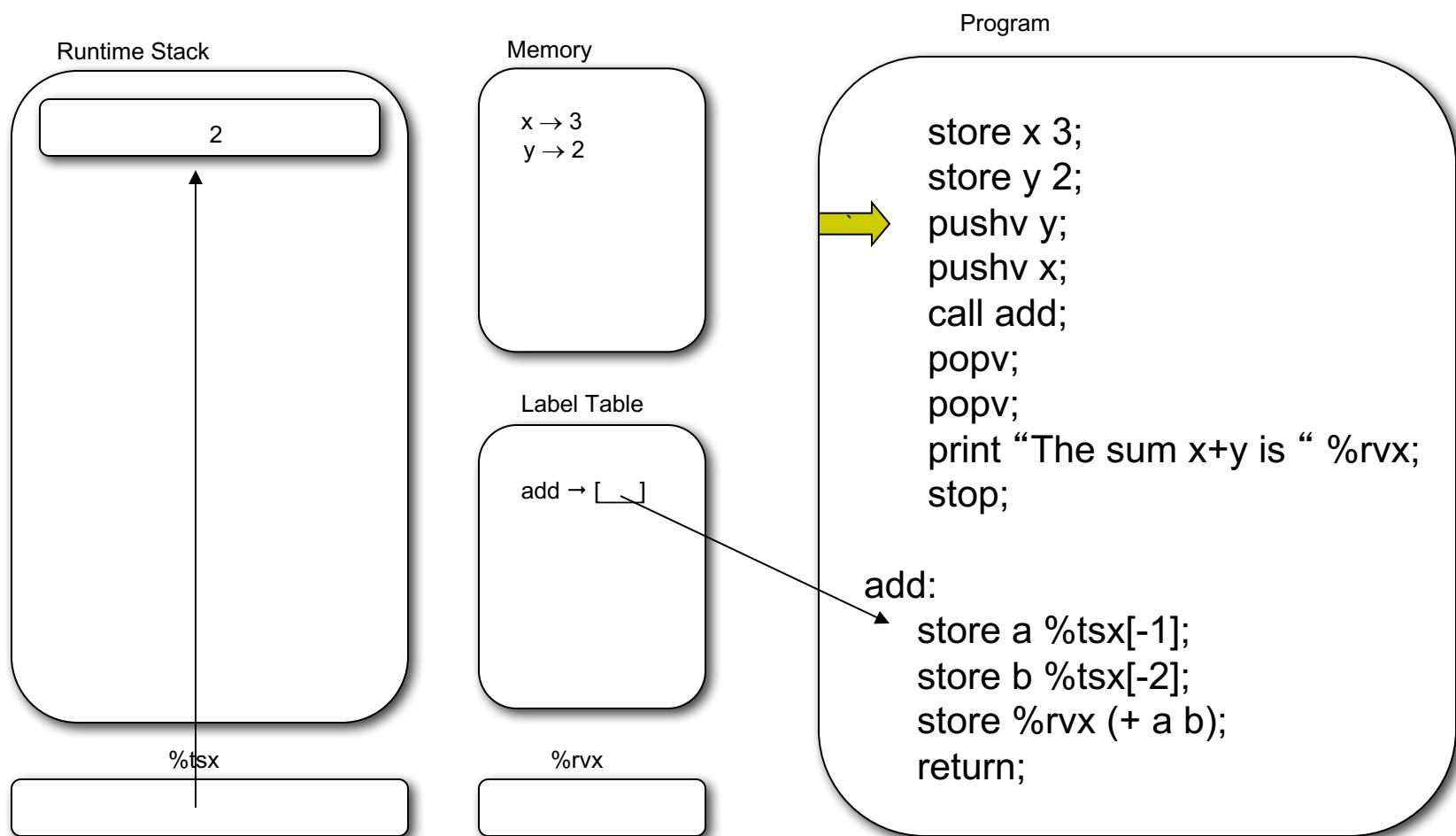


# Virtual Machine Design



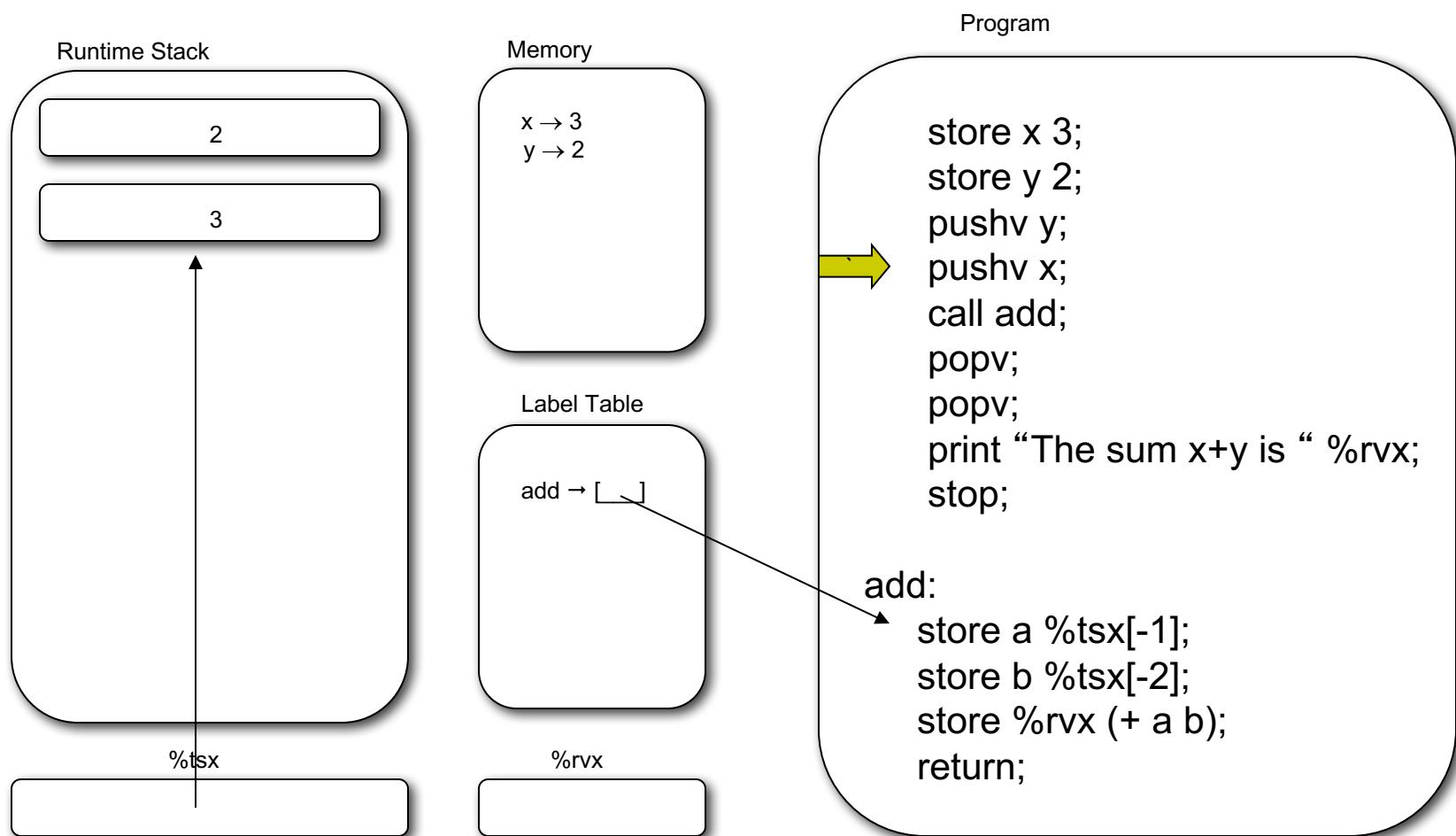


# Virtual Machine Design



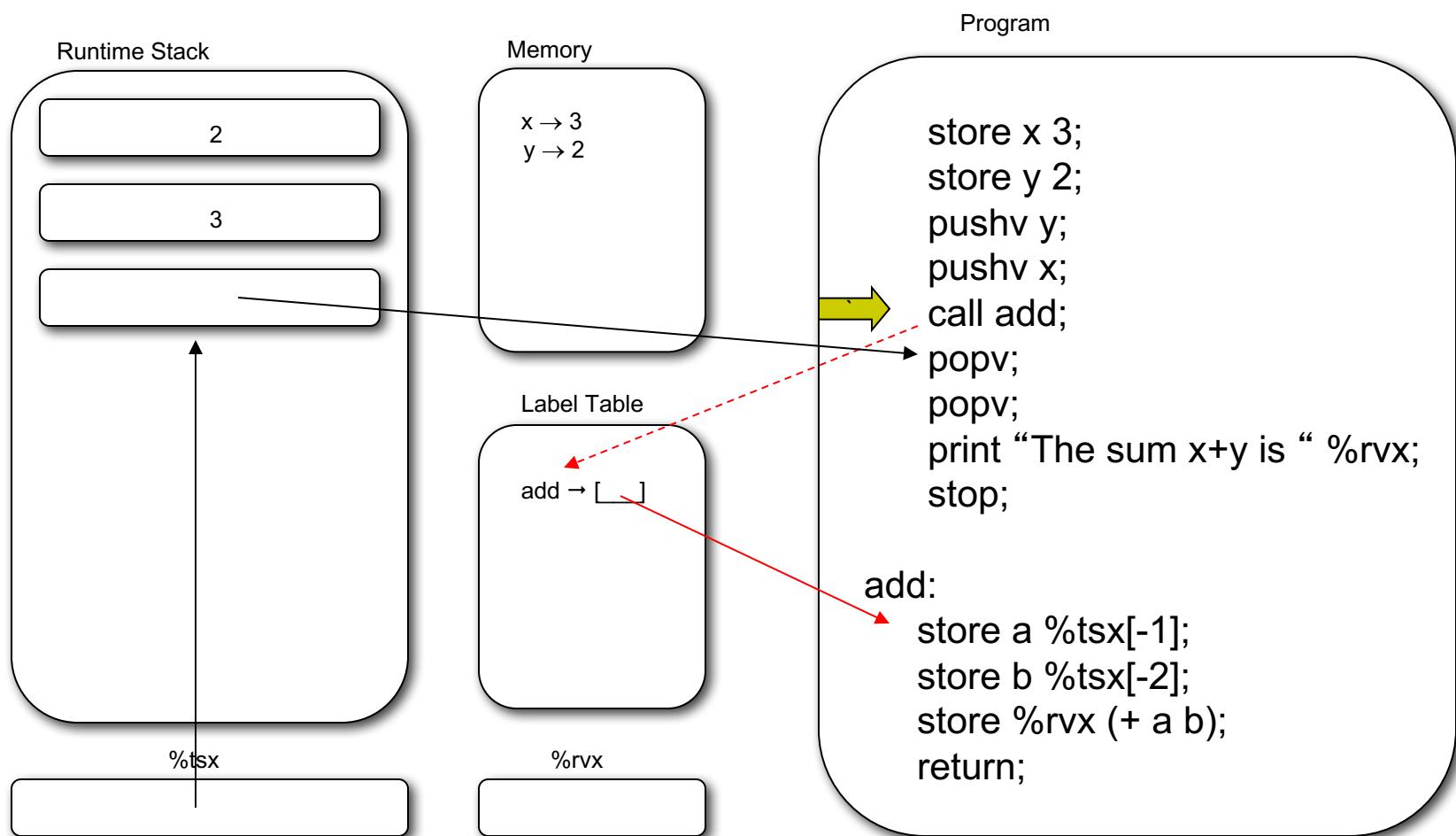


# Virtual Machine Design



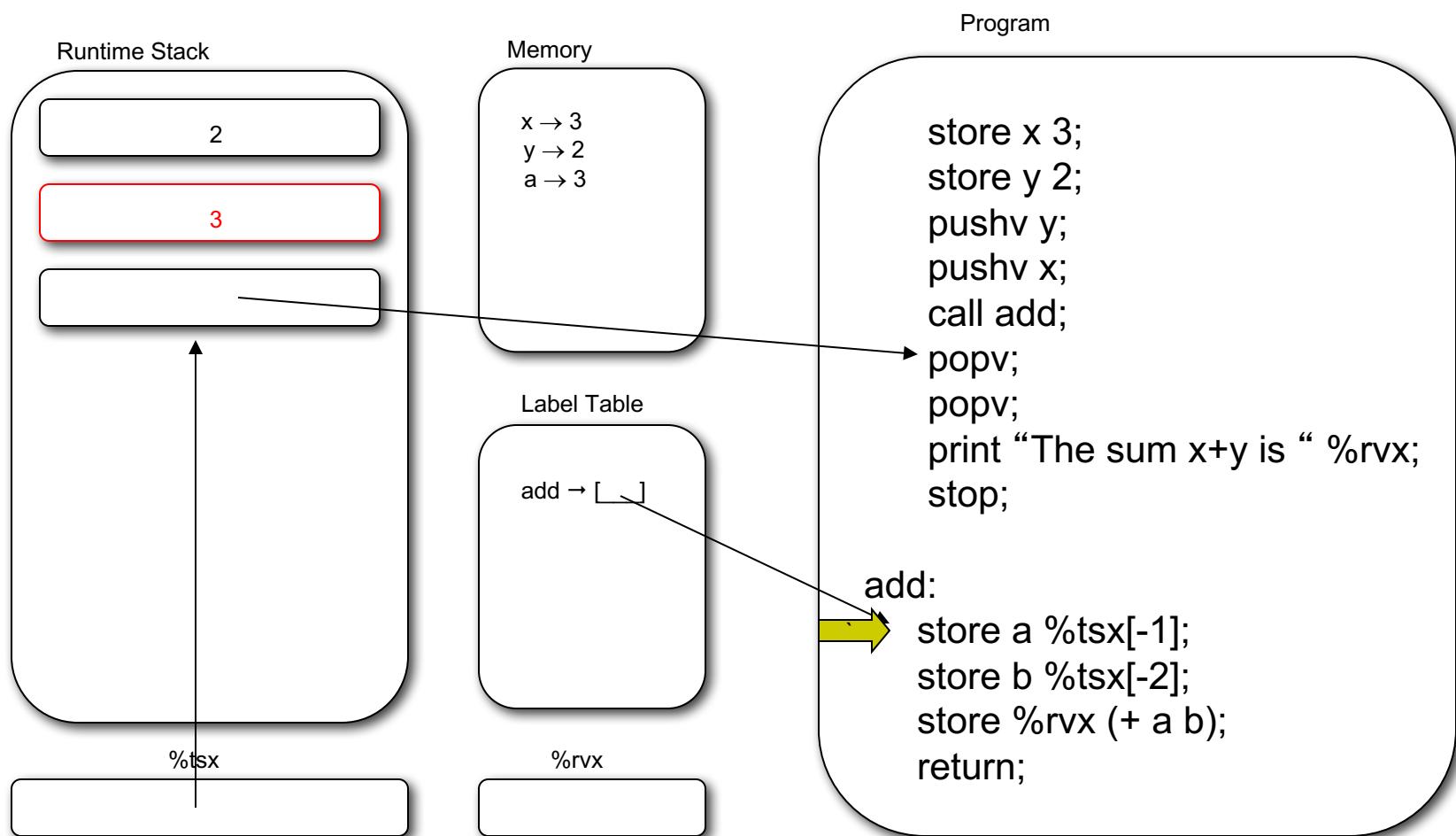


# Virtual Machine Design



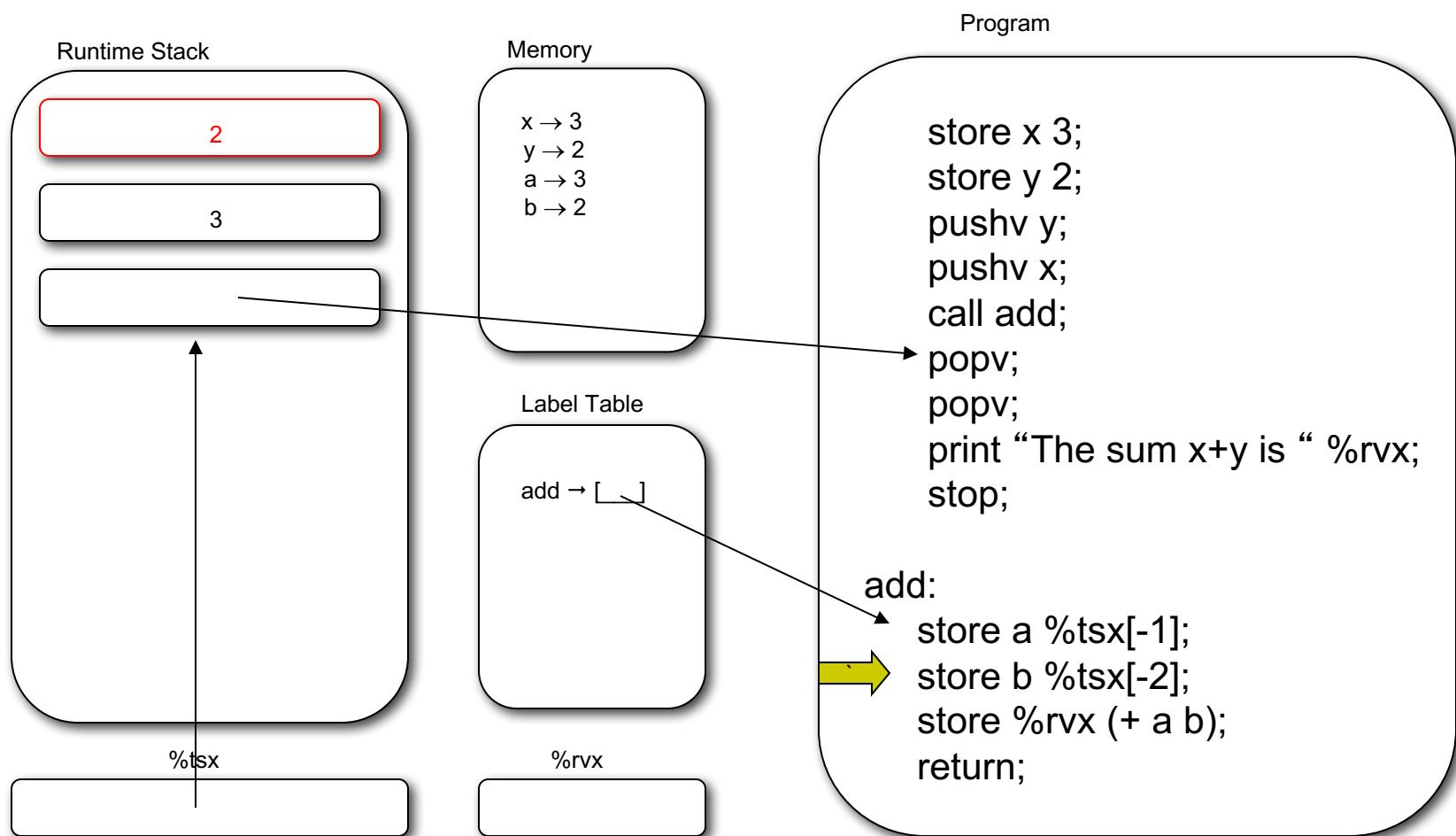


# Virtual Machine Design



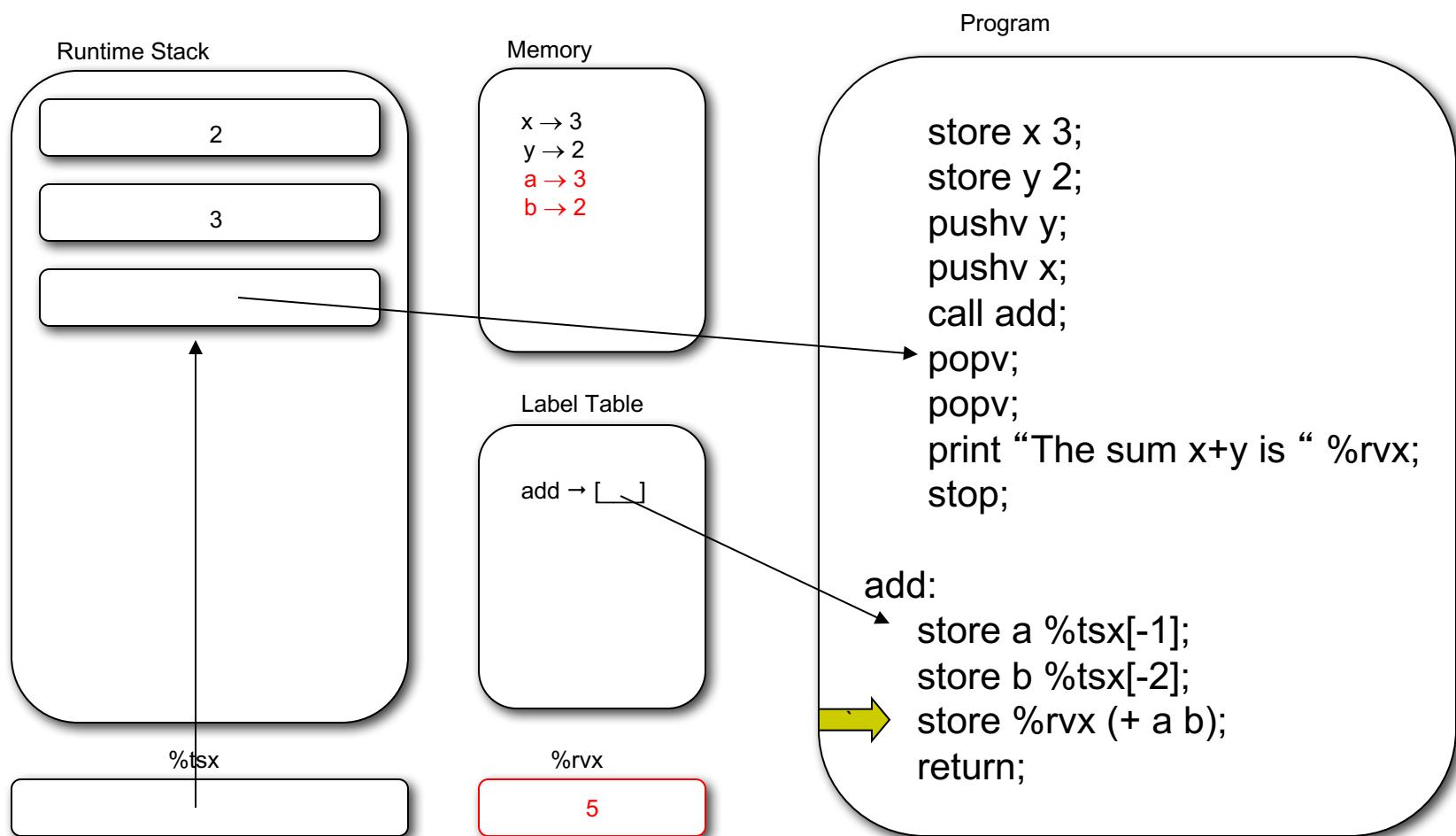


# Virtual Machine Design



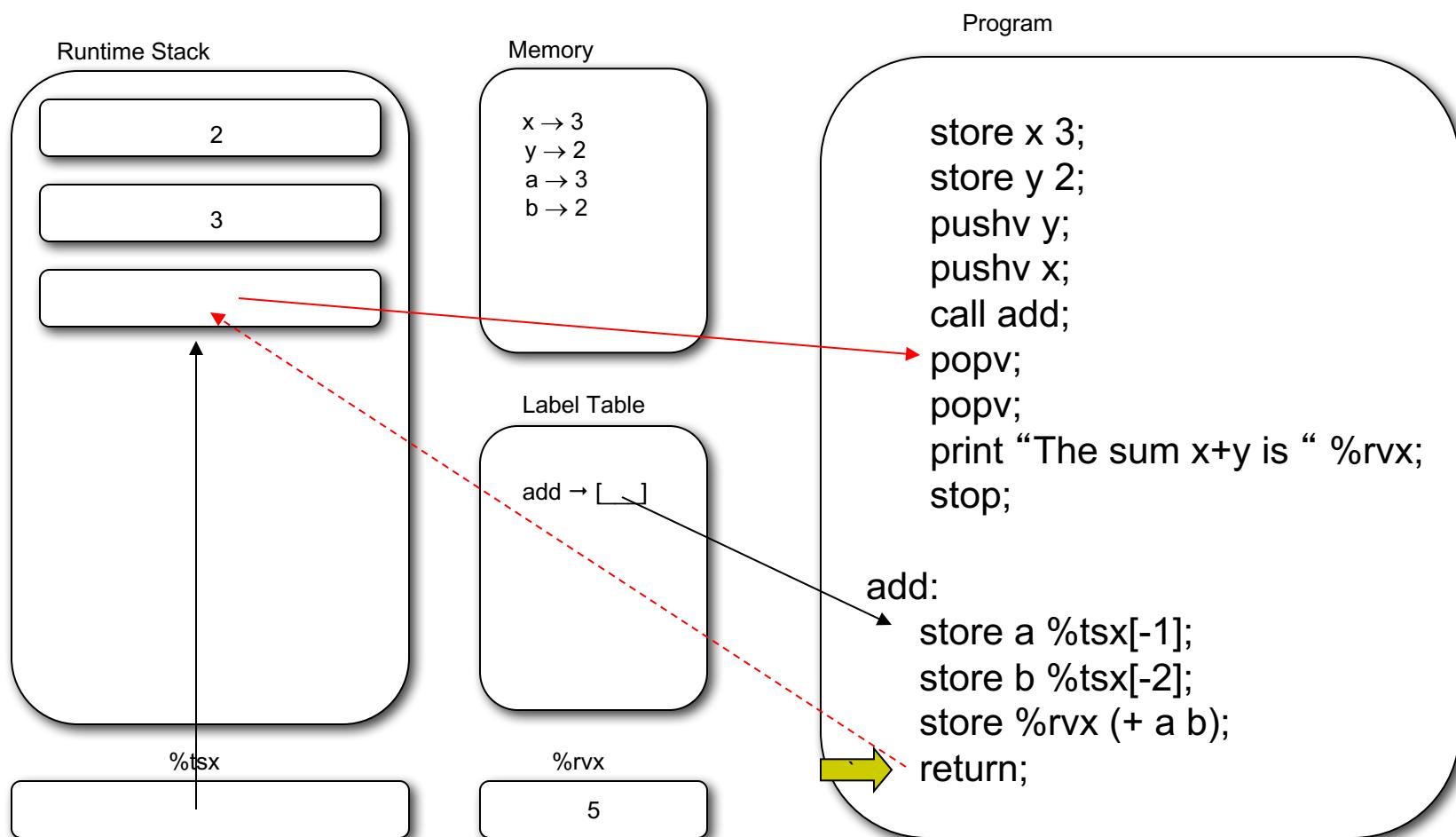


# Virtual Machine Design



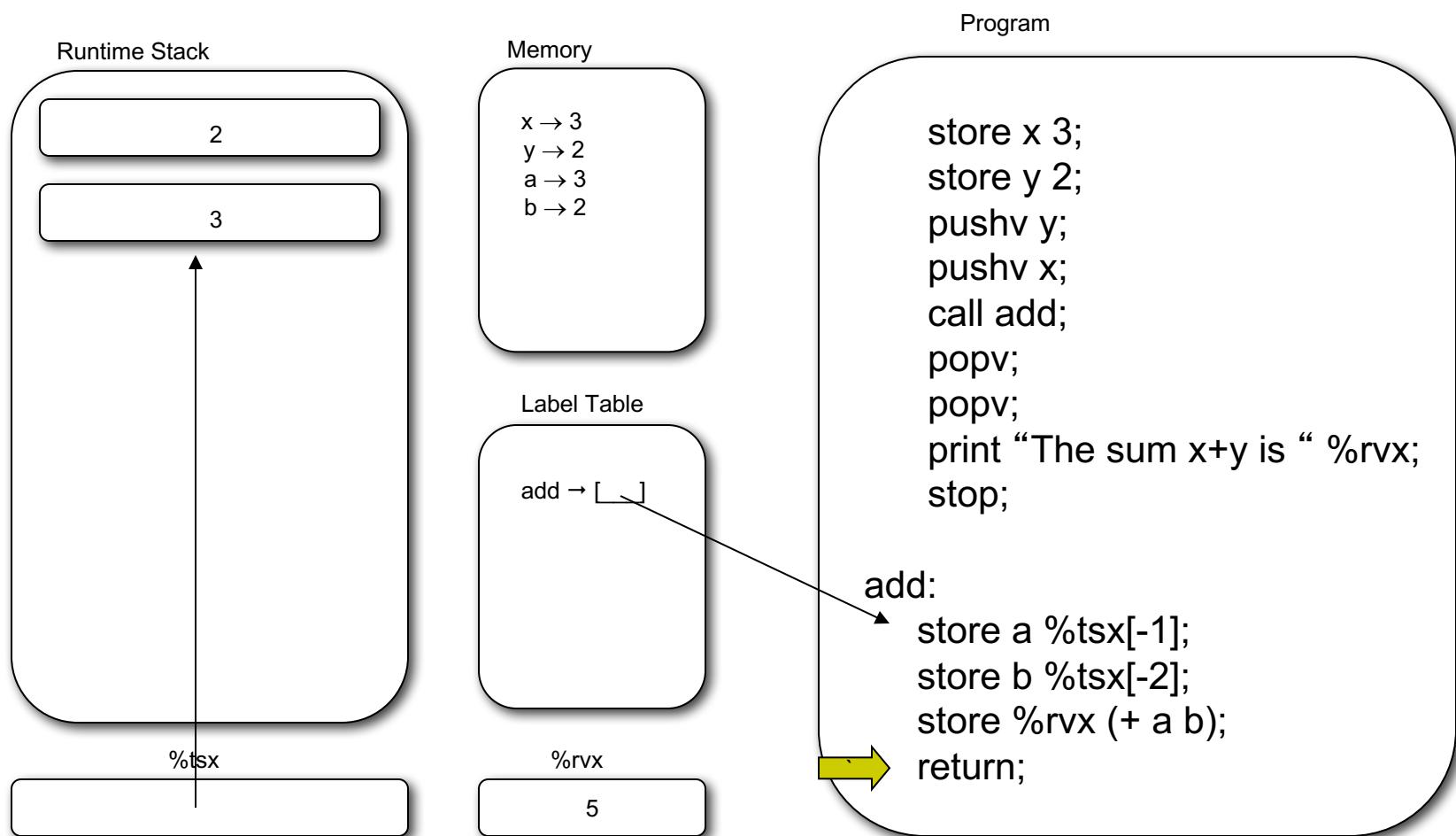


# Virtual Machine Design



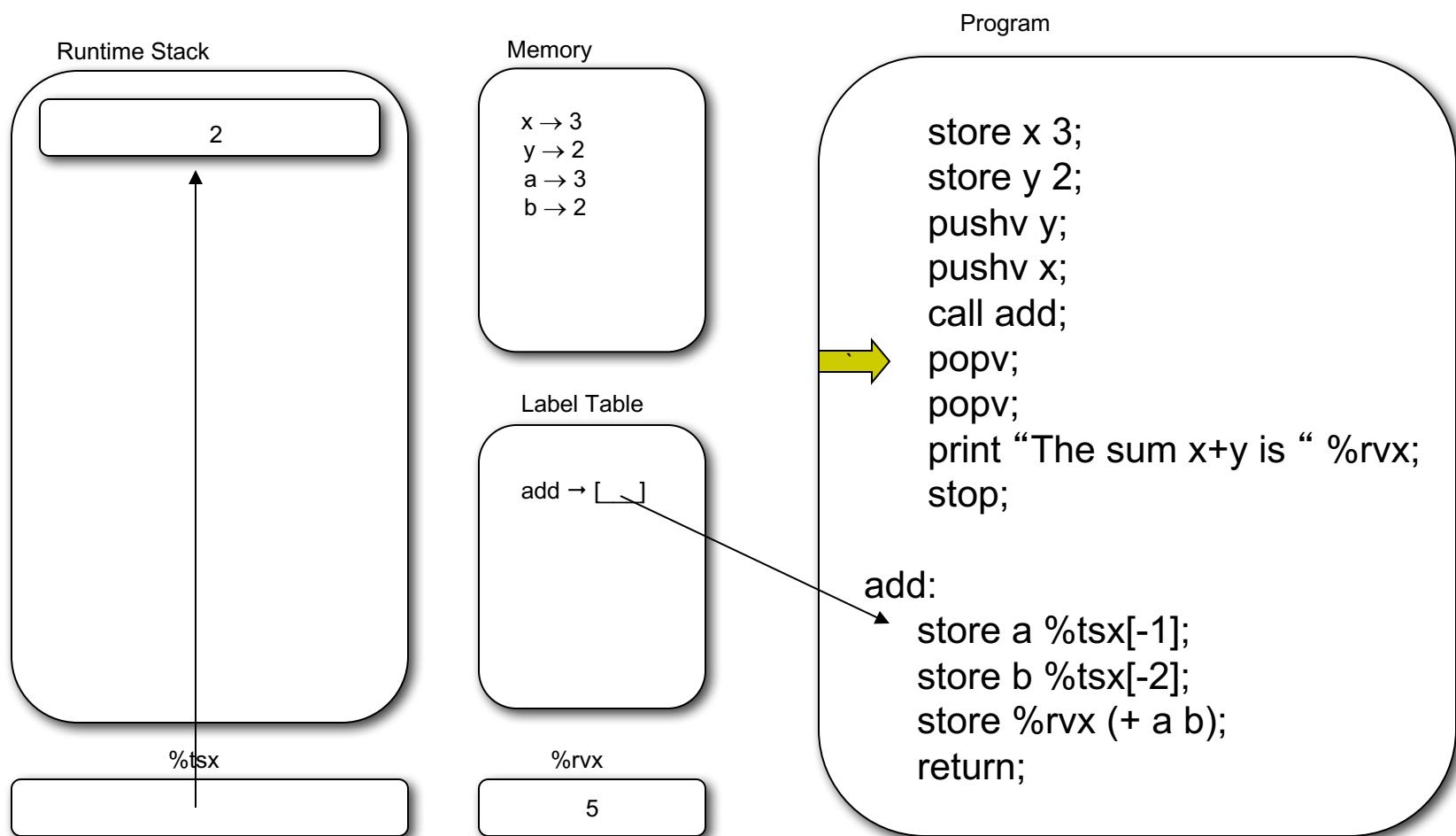


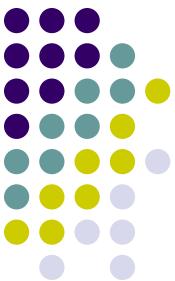
# Virtual Machine Design



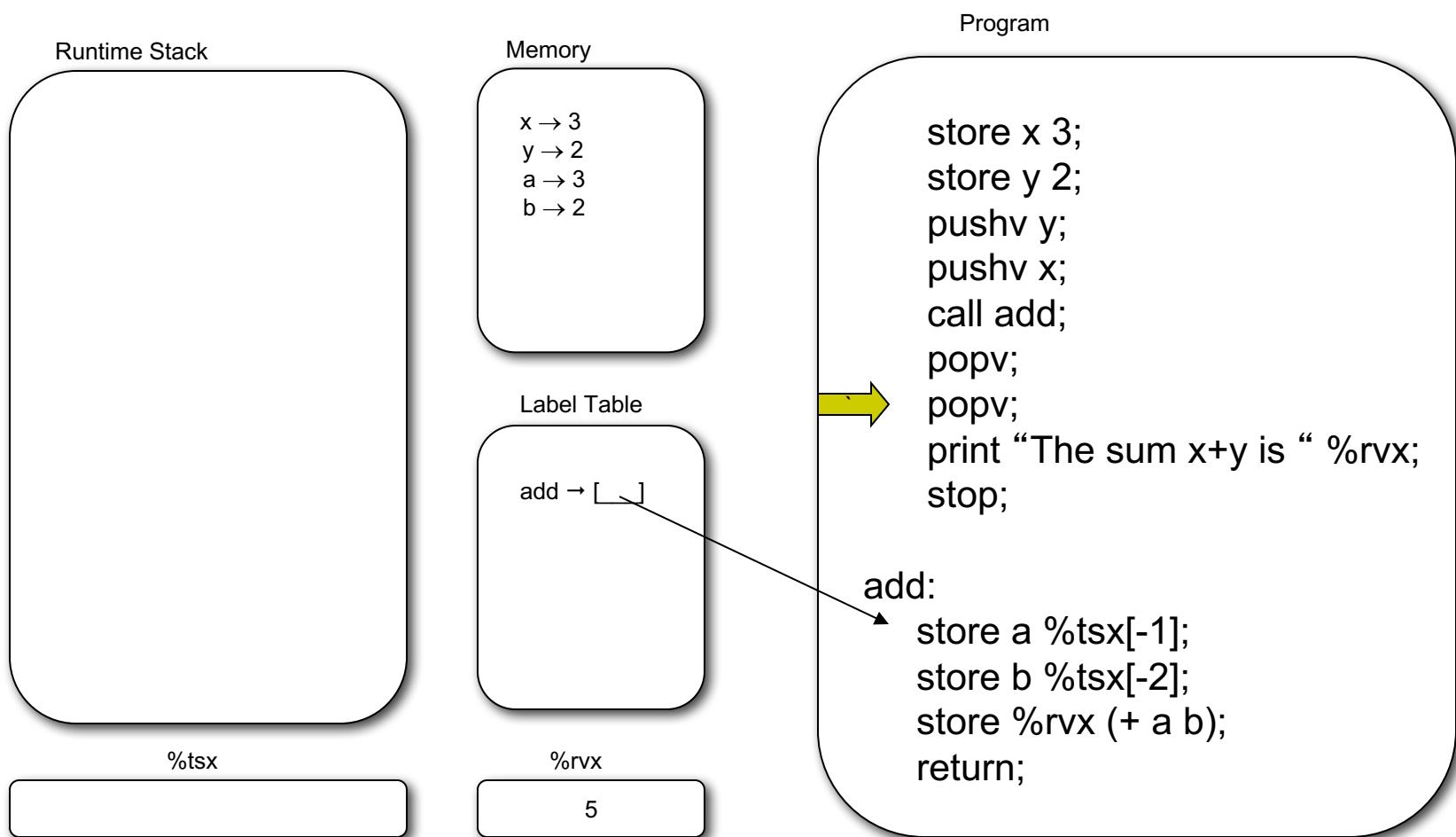


# Virtual Machine Design





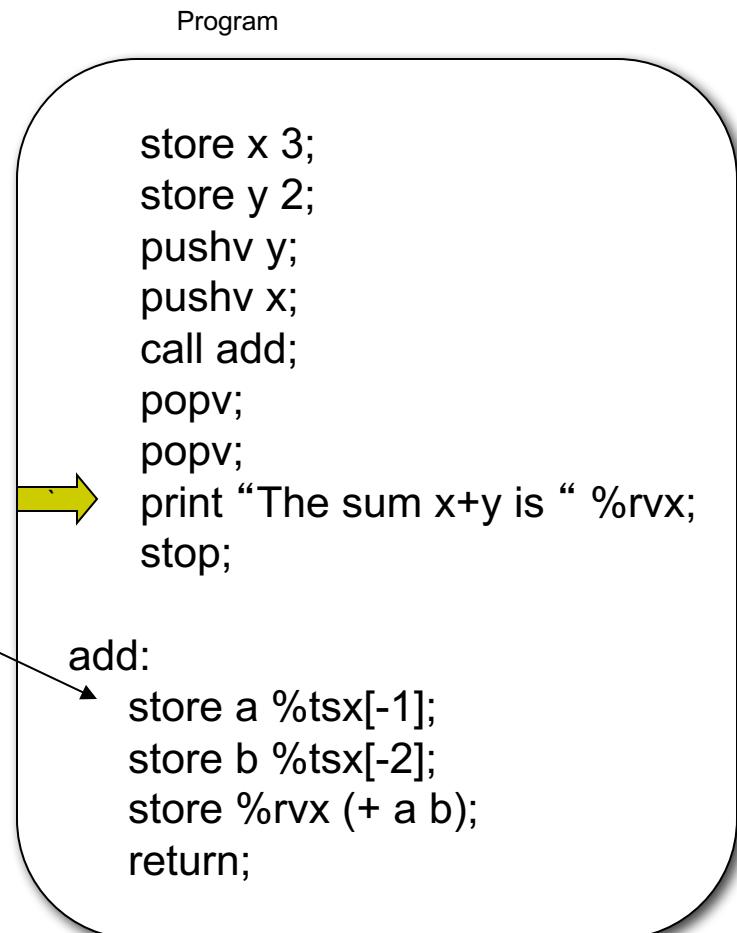
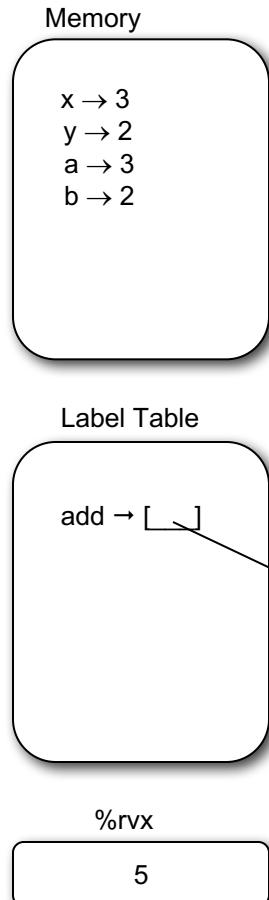
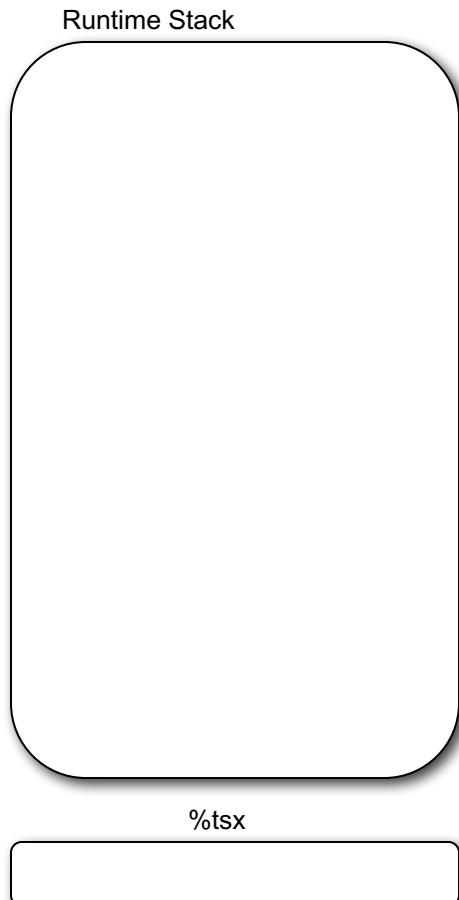
# Virtual Machine Design

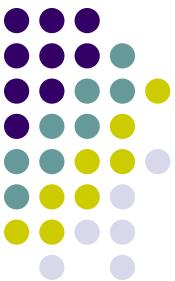




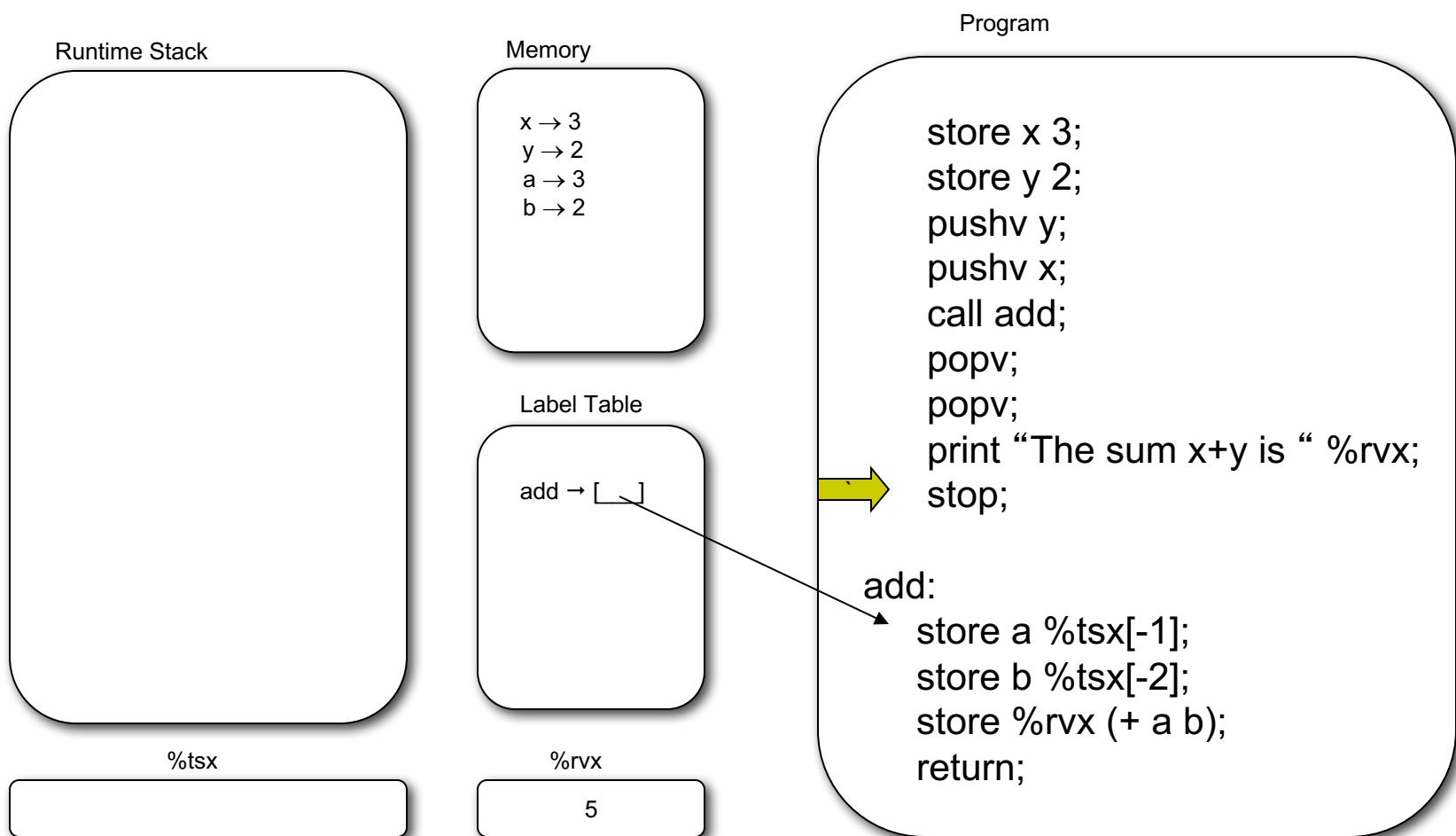
# Virtual Machine Design

The sum of x+y is 5



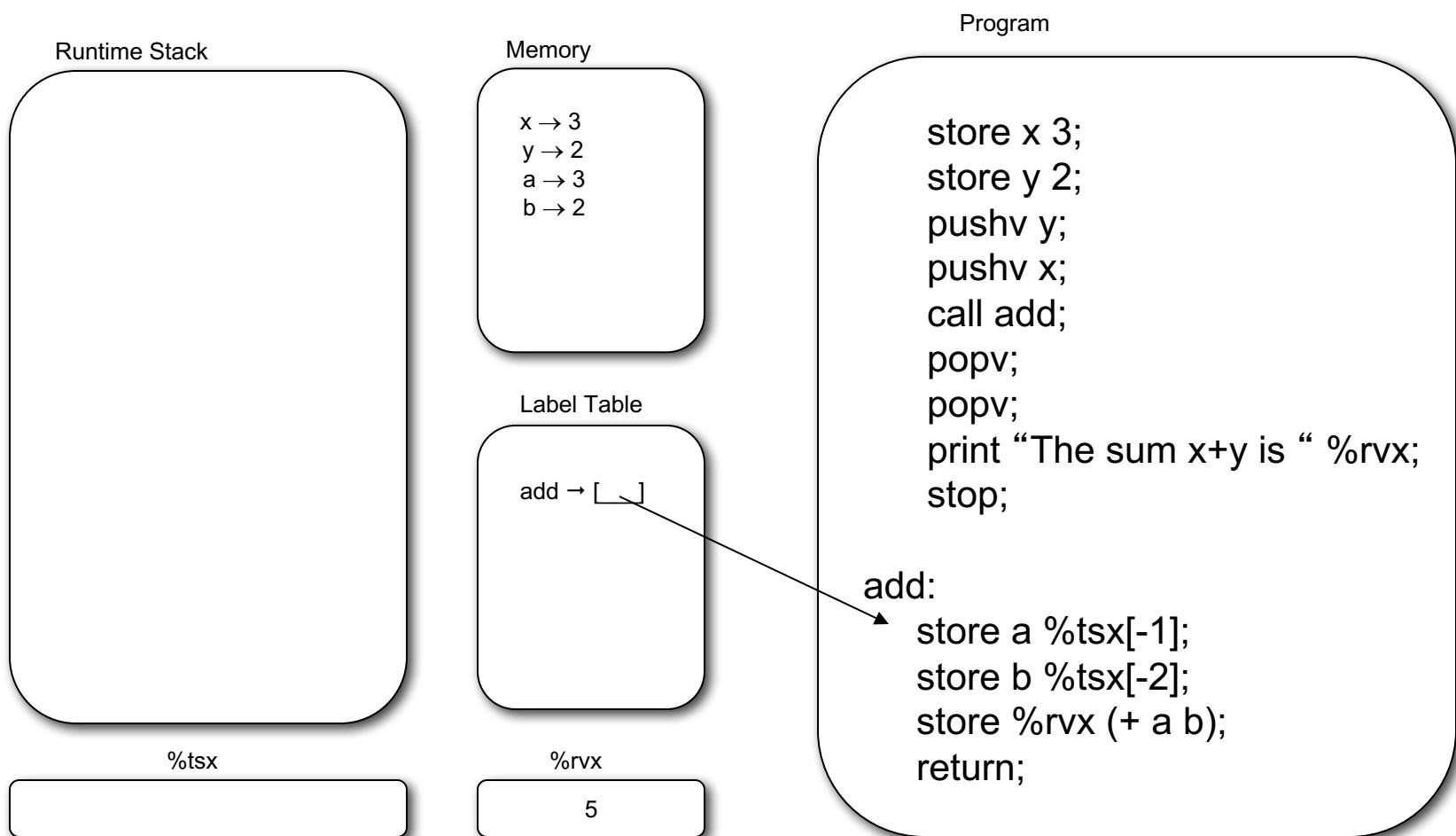


# Virtual Machine Design





# Virtual Machine Design



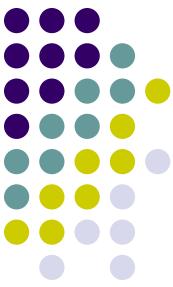


# Example Program

- Here is another way to write the same program taking advantage of indirect addressing.

```
declare add(a,b) {  
    return a+b  
}  
  
declare x = 3;  
declare y = 2;  
put add(x,y);
```

```
store x 3;  
store y 2;  
pushv y;  
pushv x;  
call add;  
popv;  
popv;  
print "The sum x+y is " %rvx;  
stop;  
  
add:  
    store %rvx (+ %tsx[-1] %tsx[-2]);  
    return;
```



# Recursive Program

- Recursive programs are tricky because function local variables, if one is not careful, are overwritten by recursive calls to the same function.
- Solution: store function local variables in a *stack frame* with one stack frame for each function invocation.



# Recursive Programs

- As an example we will study the bytecode that implements the following Cuppa3 program:

```
// print the sequence 0 1 2
declare seq(x) {
    if (1<=x)
        seq(x-1)
        put(x)
}
seq(2)
```



# Recursive Programs

- Our first attempt:

```
// print the sequence 0 1 2
declare seq(x) {
    if (1<=x)
        seq(x-1)
    put(x)
}

seq(2)
```

```
# print the sequence 0 1 2
pushv 2;
call seq;
popv;
stop;

seq:
    store x %tsx[-1];
    jumpF x L1;
    pushv (- x 1);
    call seq;
    popv;
L1:
    print x;
    return;
```

Note: this program does **NOT** work because x is a global variable which will get overwritten by each invocation of seq.



# Recursive Programs

- We rewrite the program using *stack frames* to store function local variables.

```
// print the sequence 0 1 2
declare seq(x) {
    if (1<=x)
        seq(x-1)
    put(x)
}
seq(2)
```

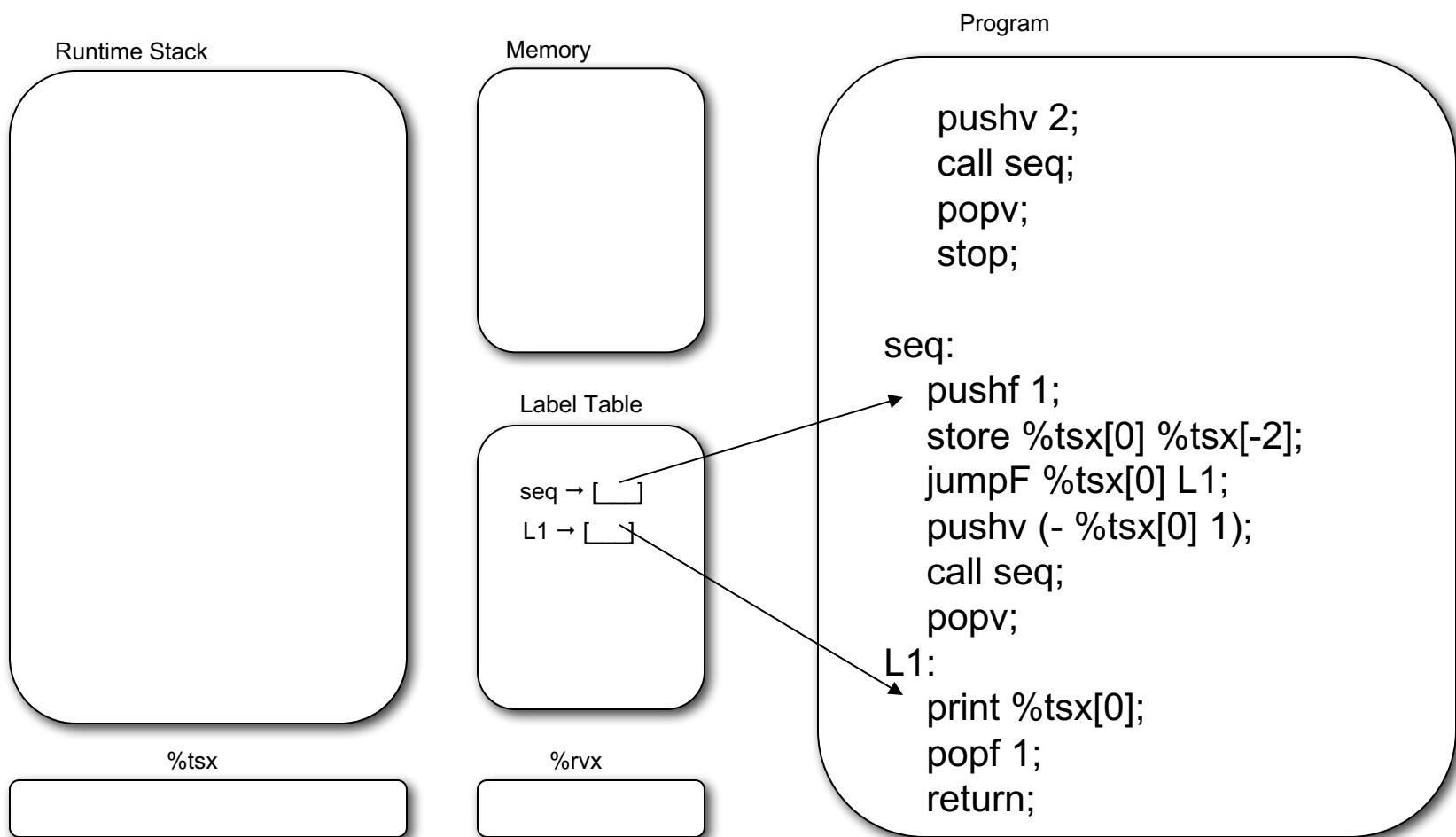
```
# print the sequence 0 1 2
pushv 2;
call seq;
popv;
stop;

seq:
    pushf 1;          # push a frame of size 1 - variable x
    store %tsx[0] %tsx[-2]; # initialize x with the actual argument
    jumpF %tsx[0] L1;      # test x
    pushv (- %tsx[0] 1);
    call seq;
    popv;
L1:
    print %tsx[0];
    popf 1;          # remove the frame from the stack
    return;
```

Note: This program now works, local variables have their own location on the stack per function invocation.

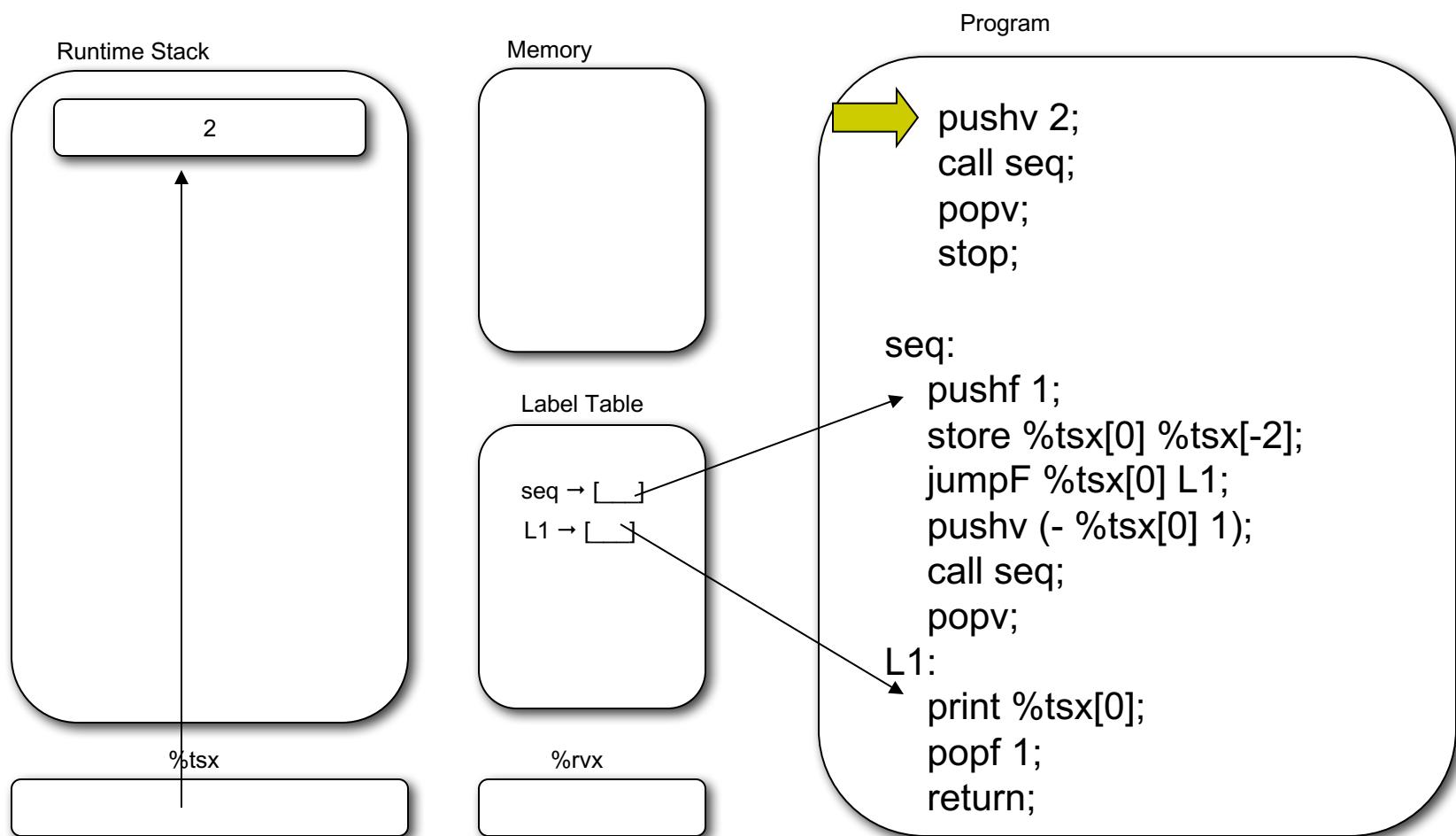


# Virtual Machine Design



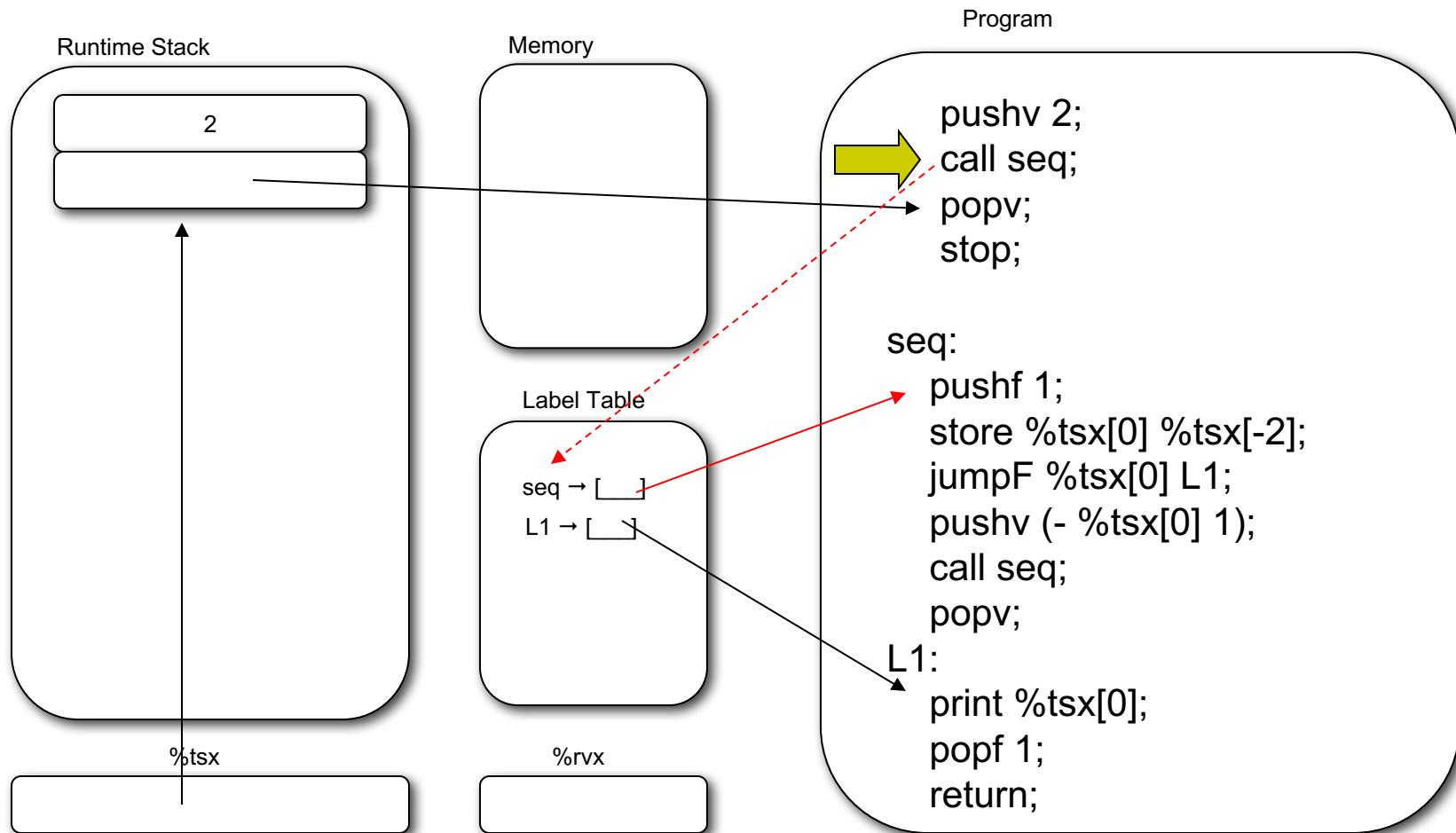


# Virtual Machine Design



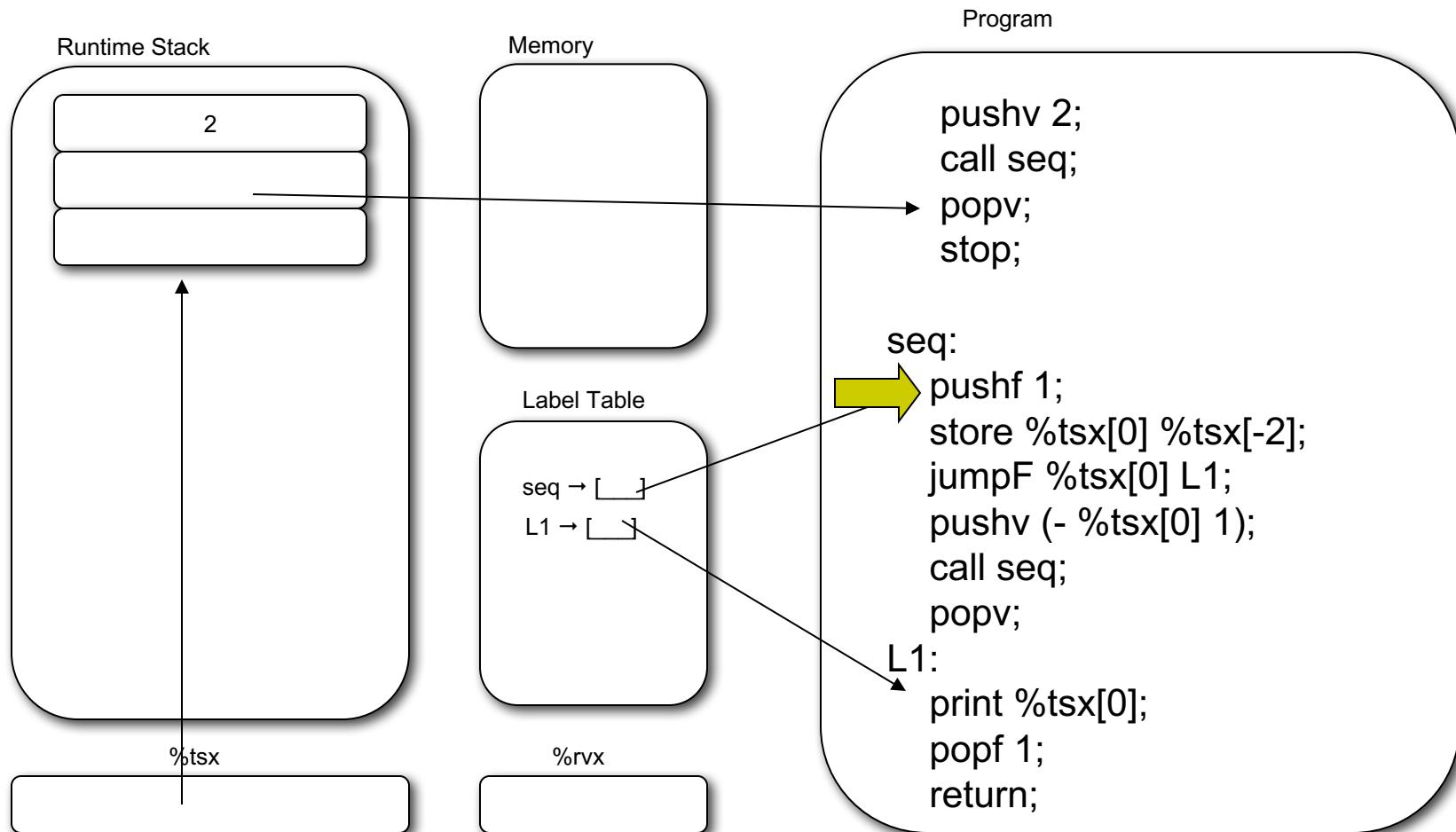


# Virtual Machine Design



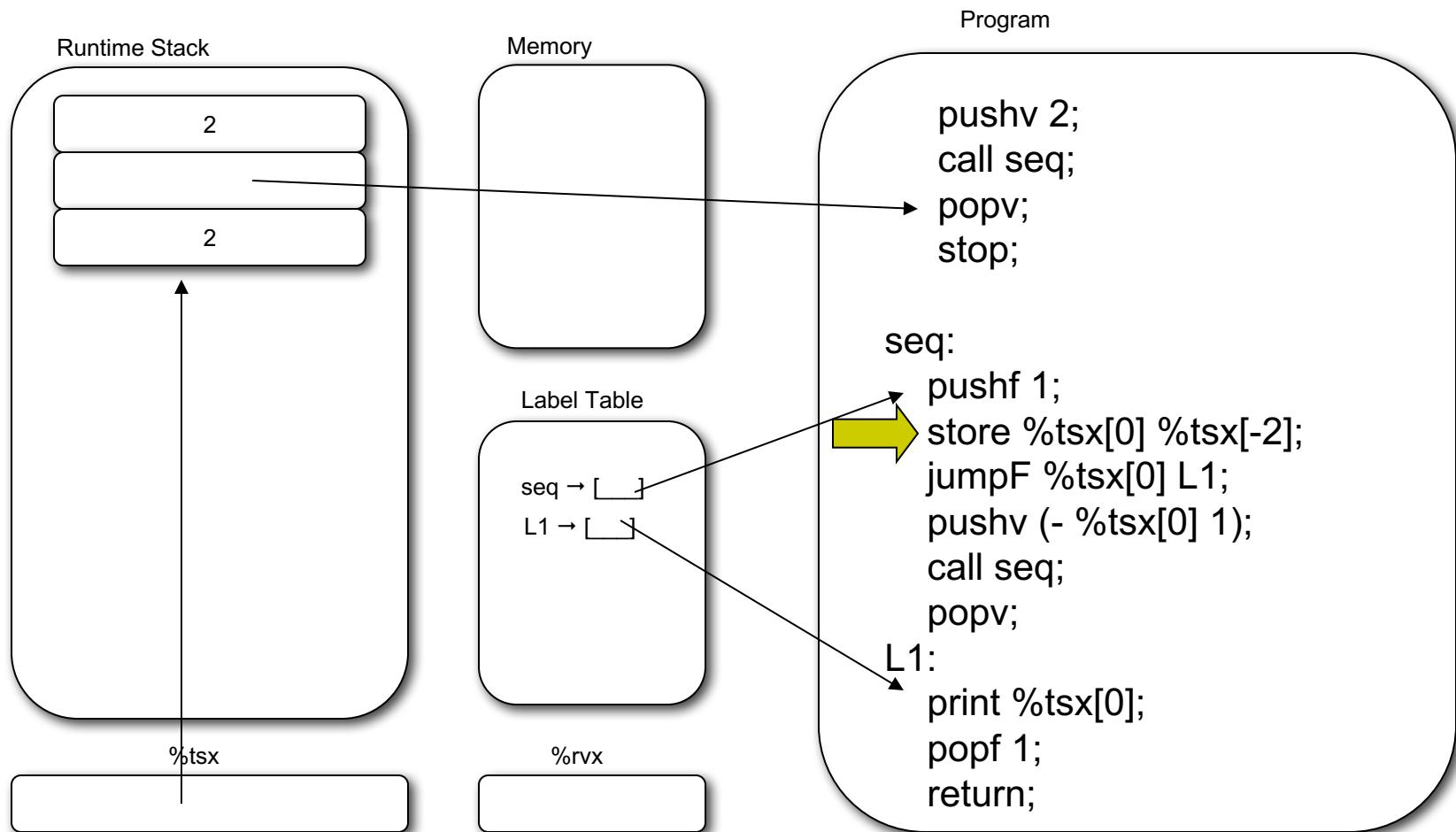


# Virtual Machine Design



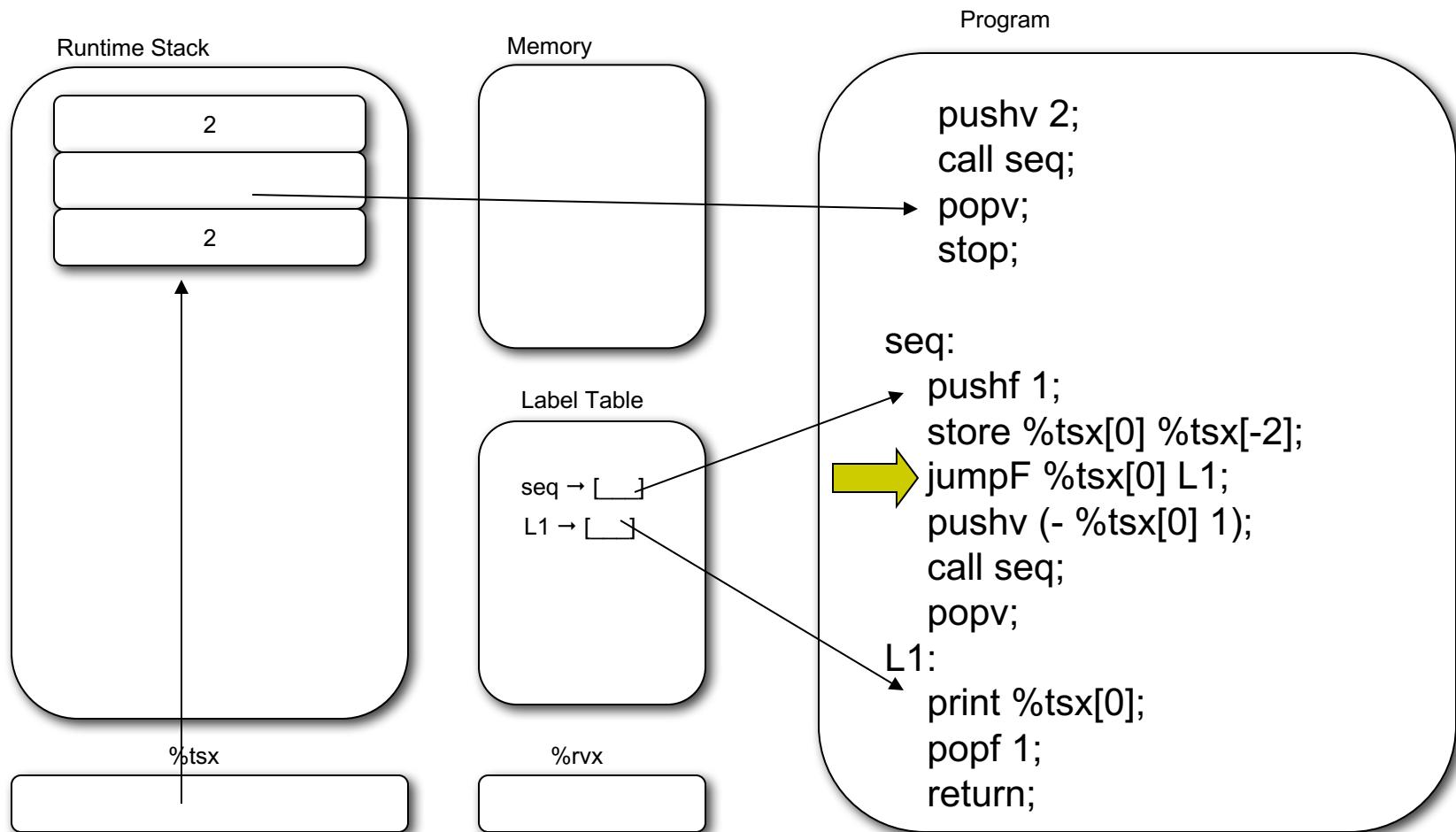


# Virtual Machine Design



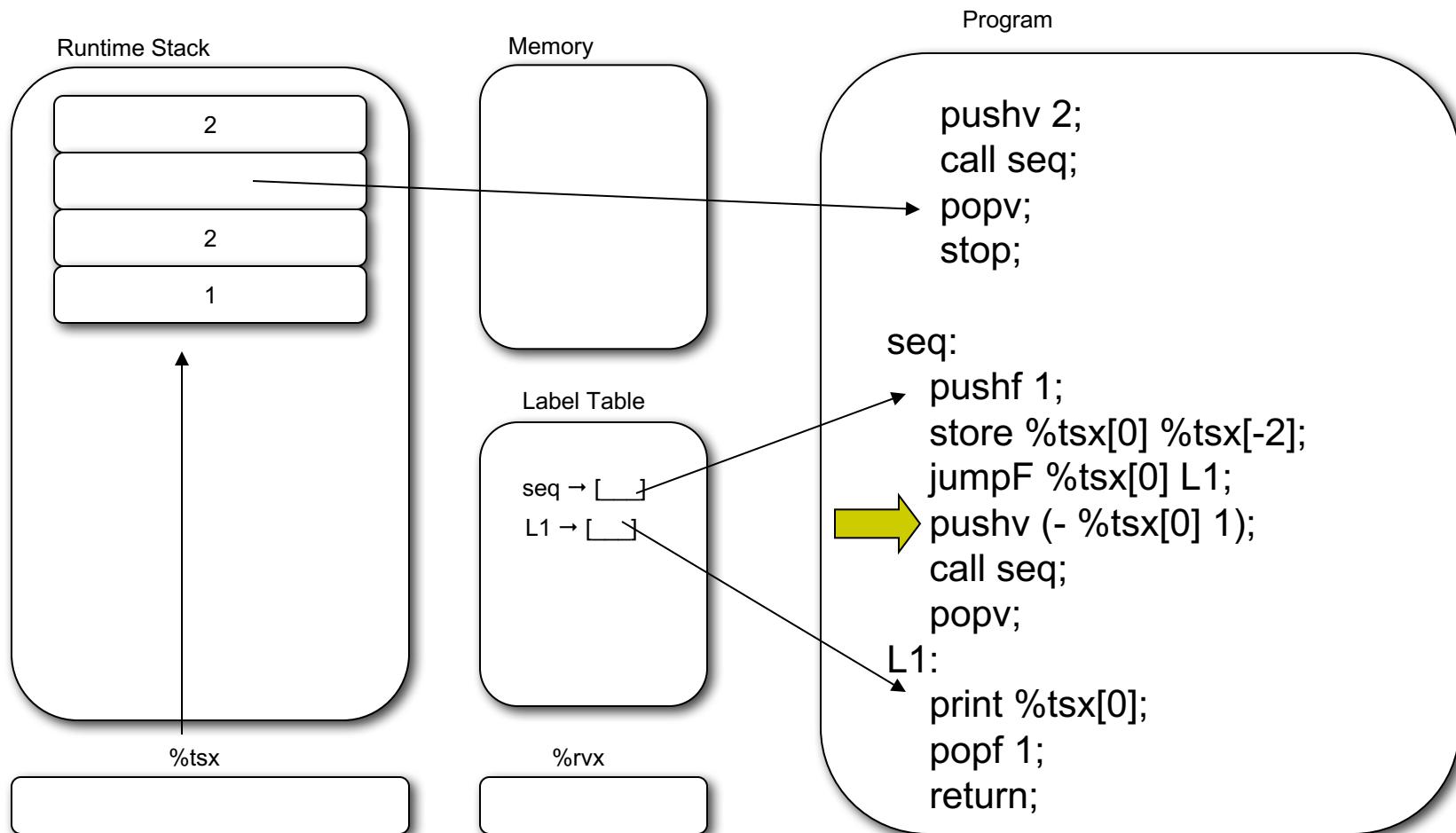


# Virtual Machine Design



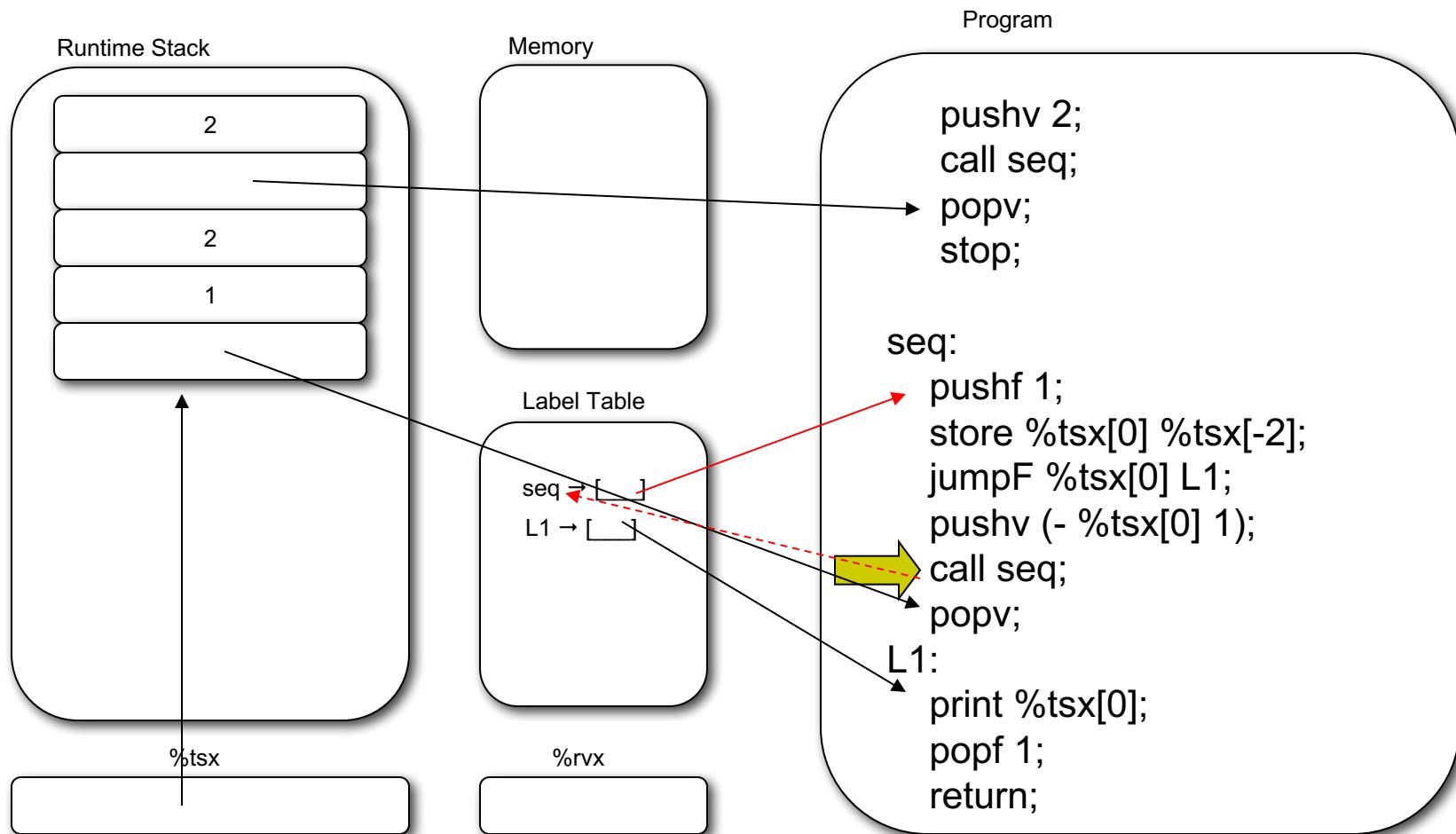


# Virtual Machine Design



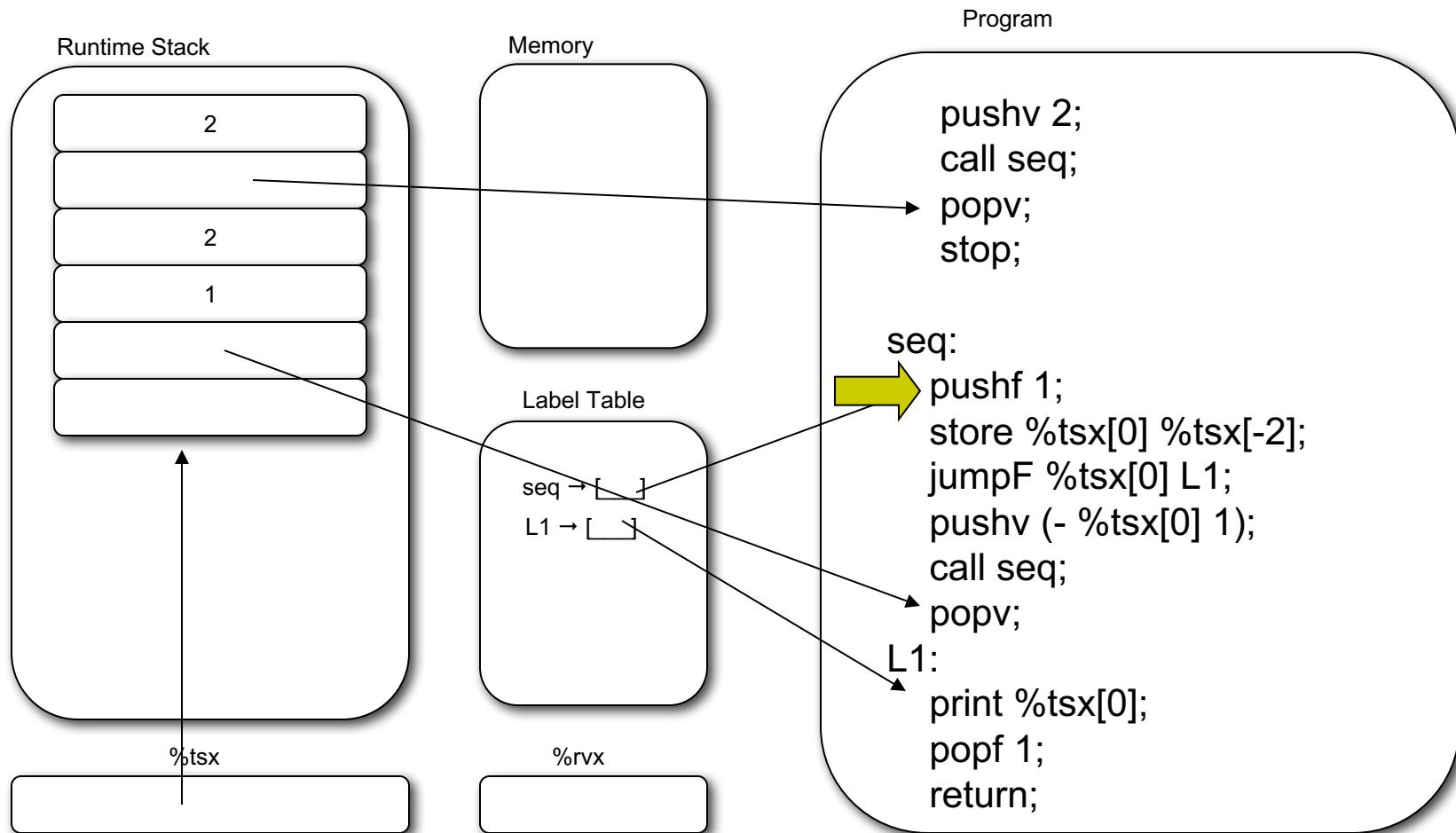


# Virtual Machine Design



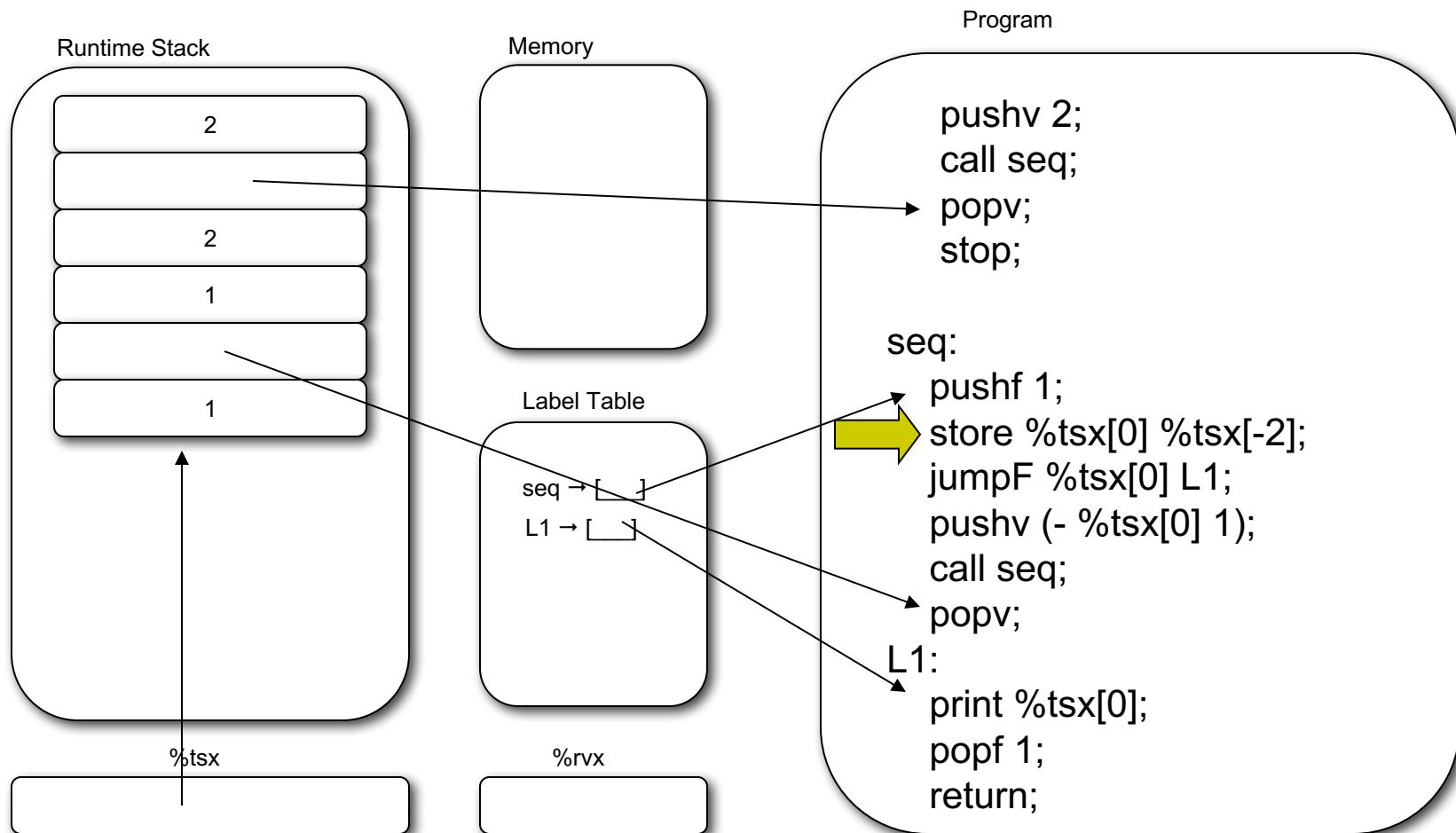


# Virtual Machine Design



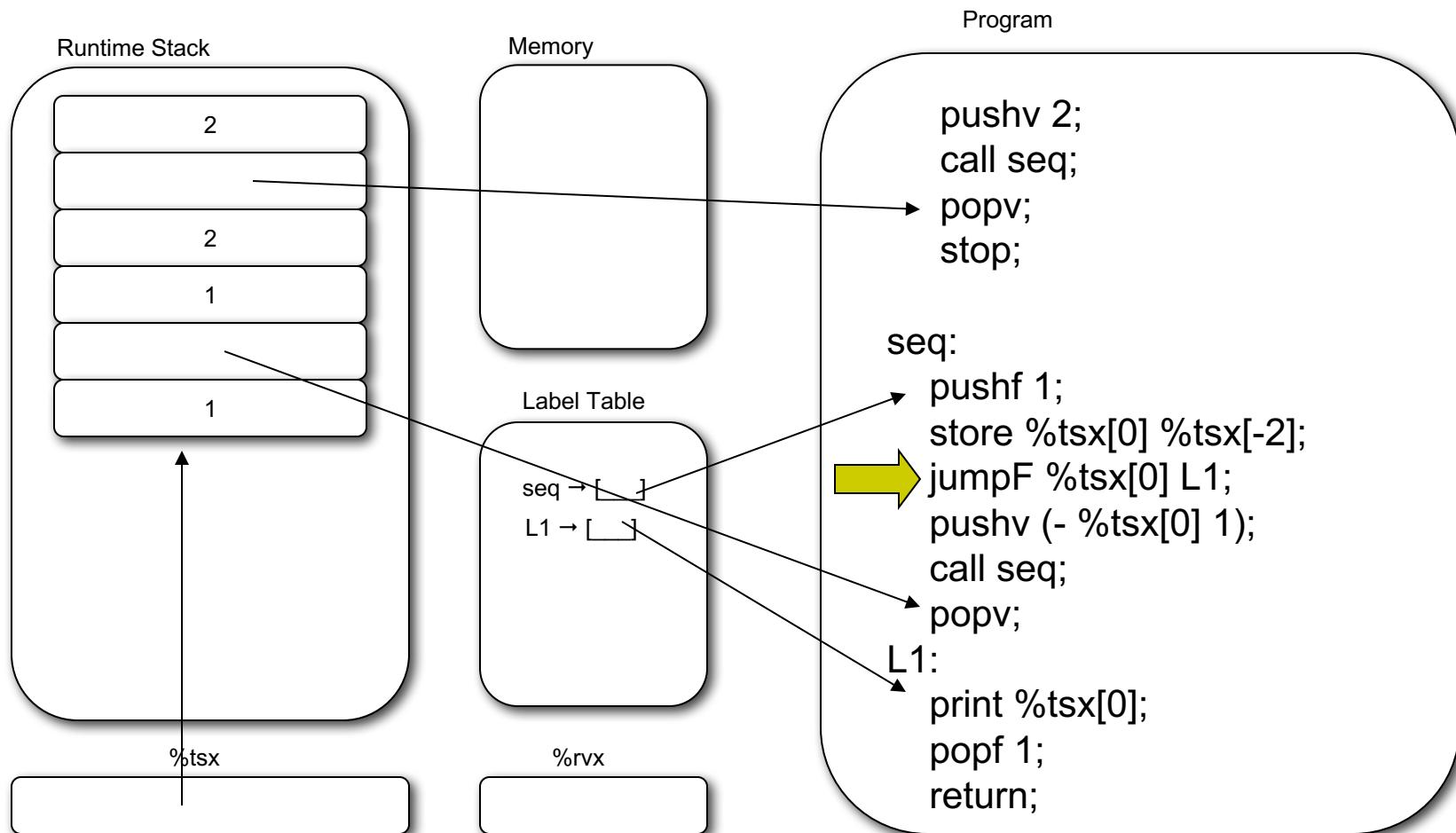


# Virtual Machine Design



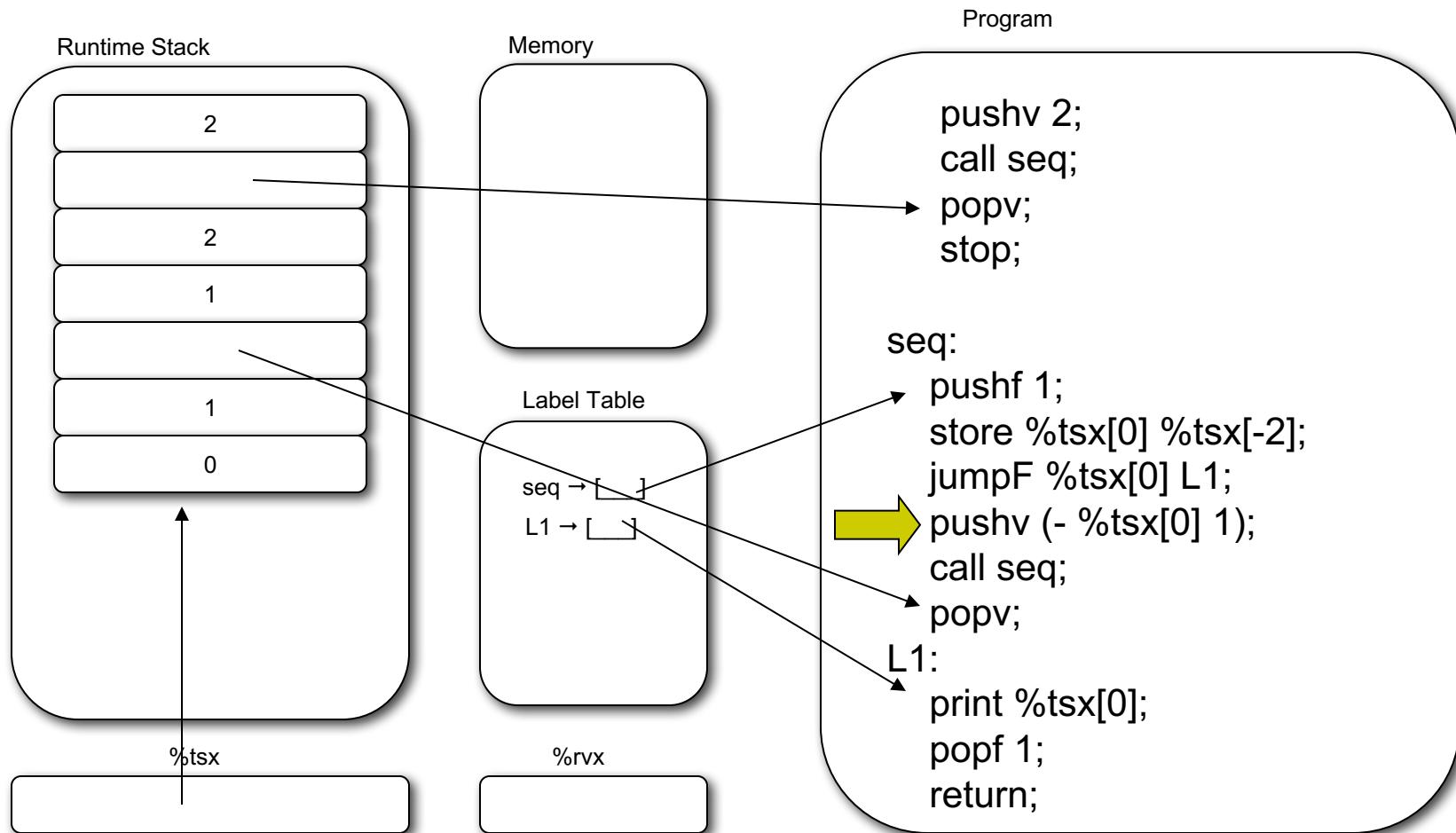


# Virtual Machine Design



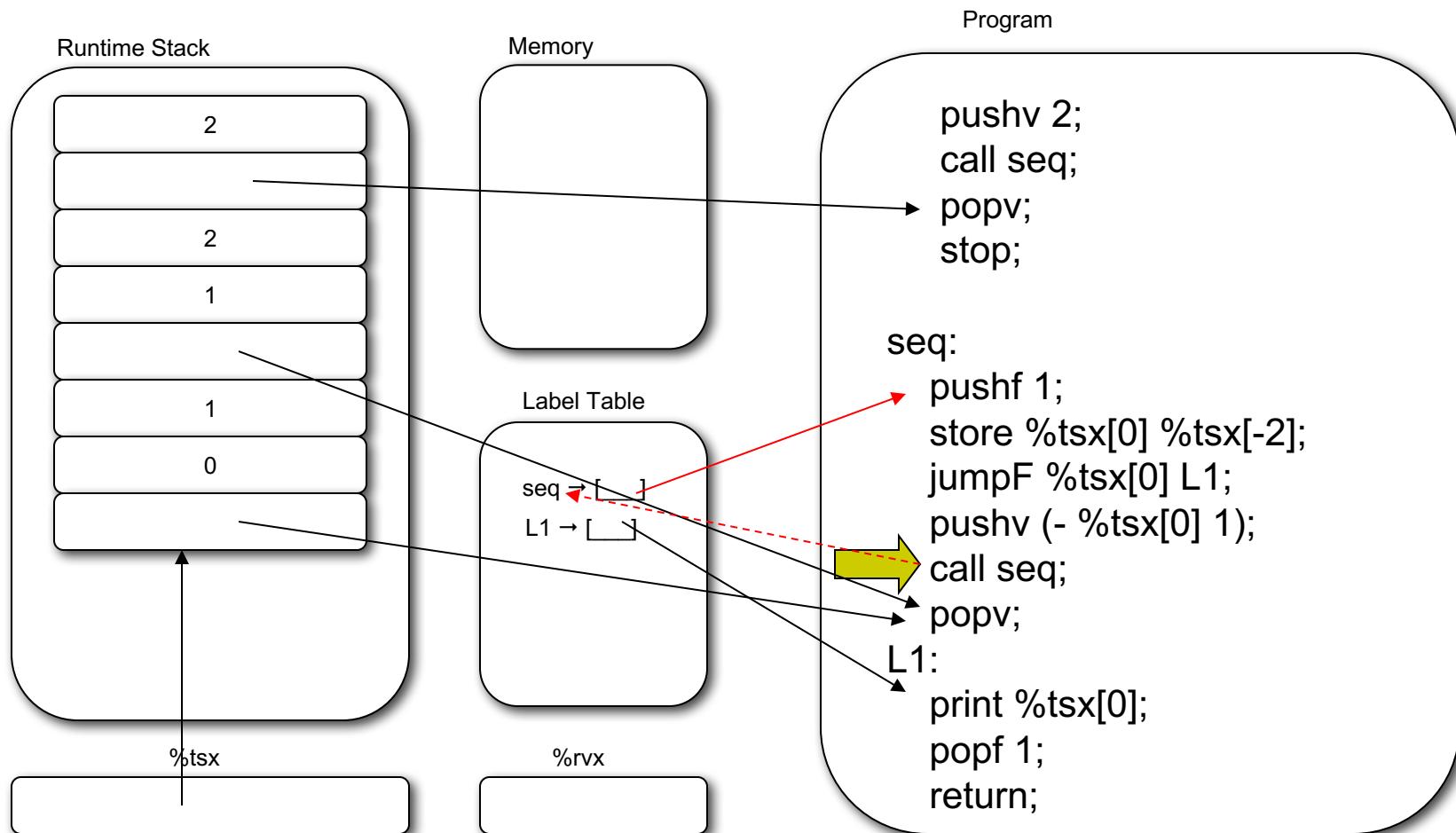


# Virtual Machine Design



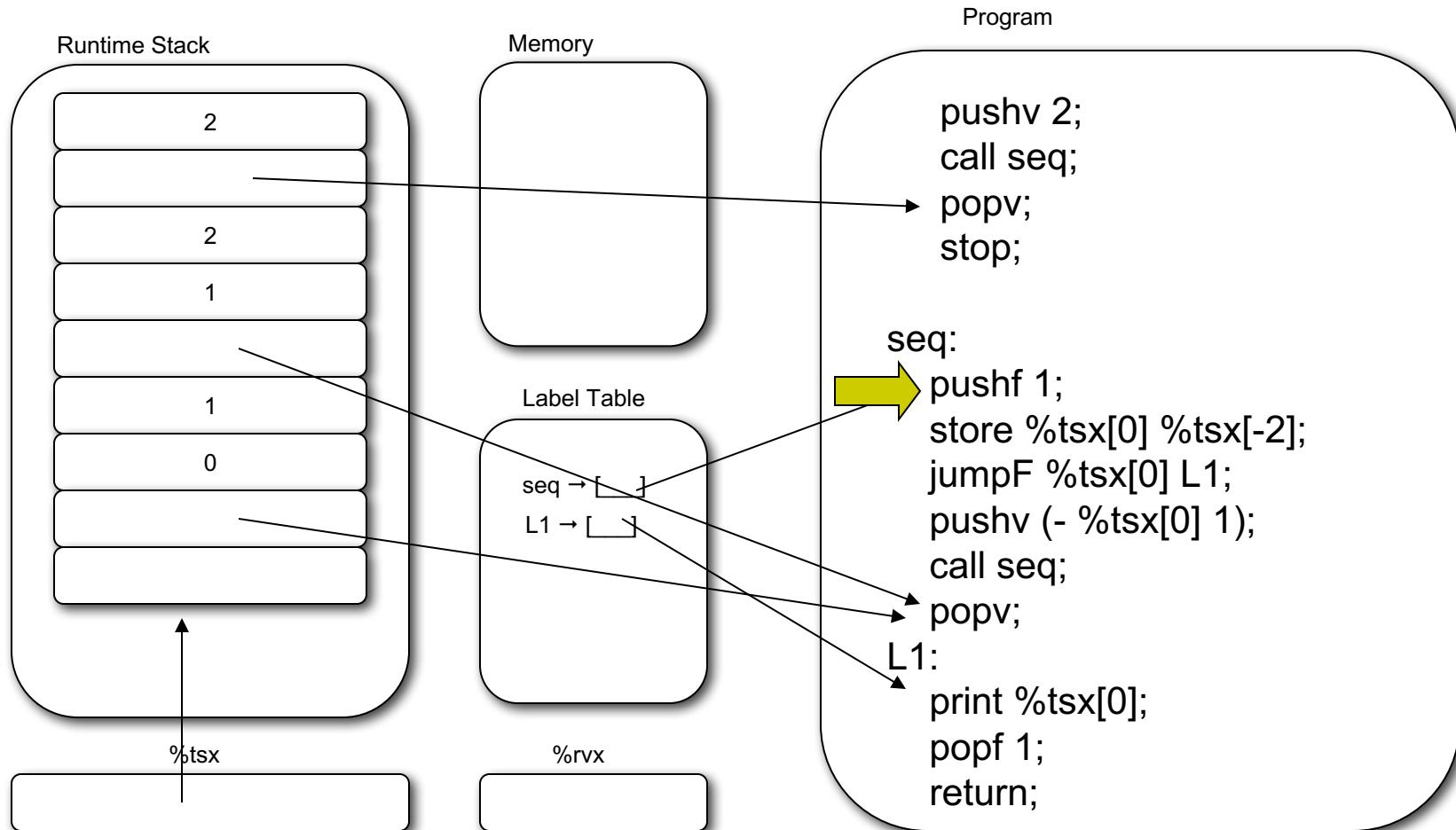


# Virtual Machine Design



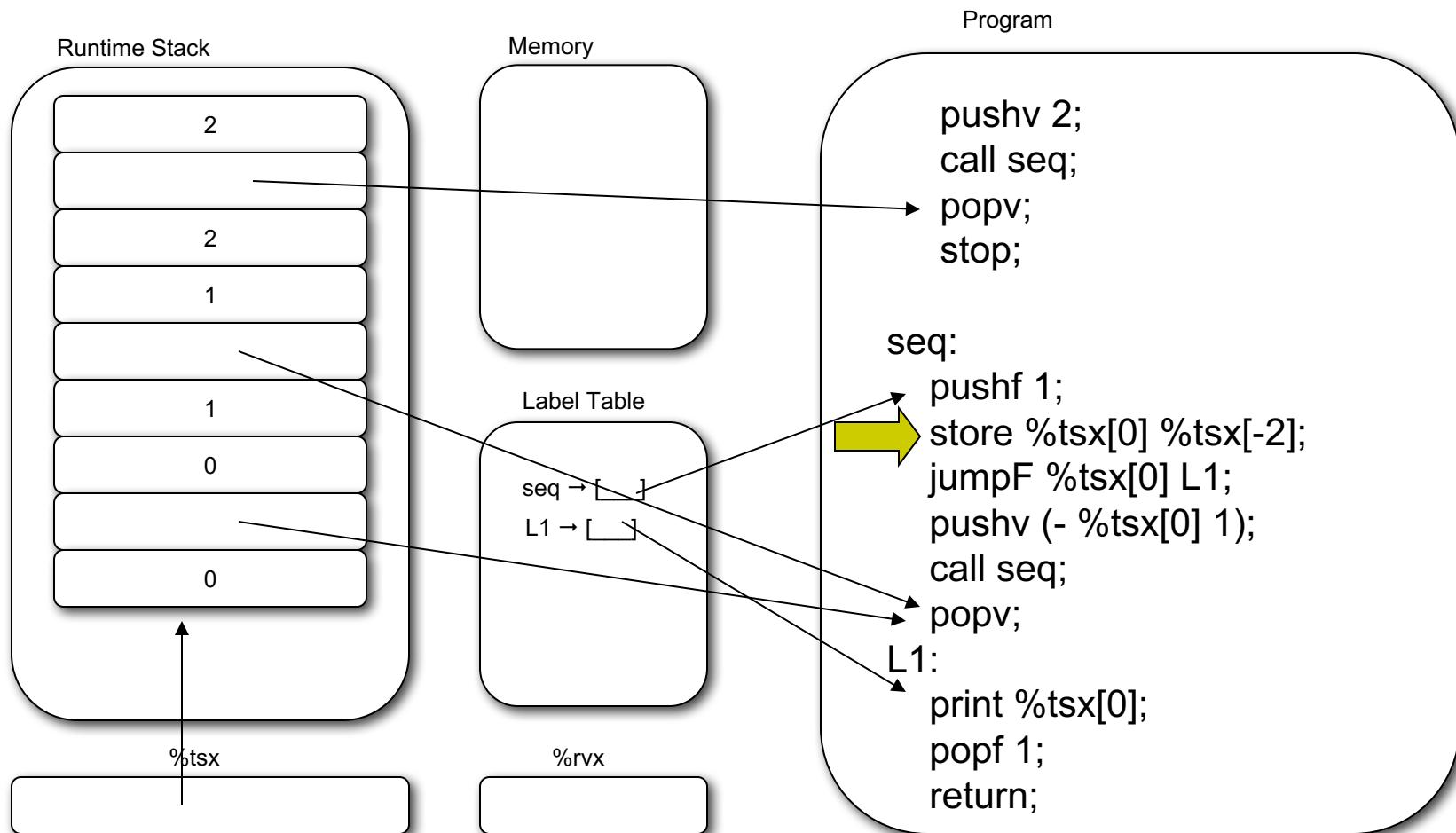


# Virtual Machine Design



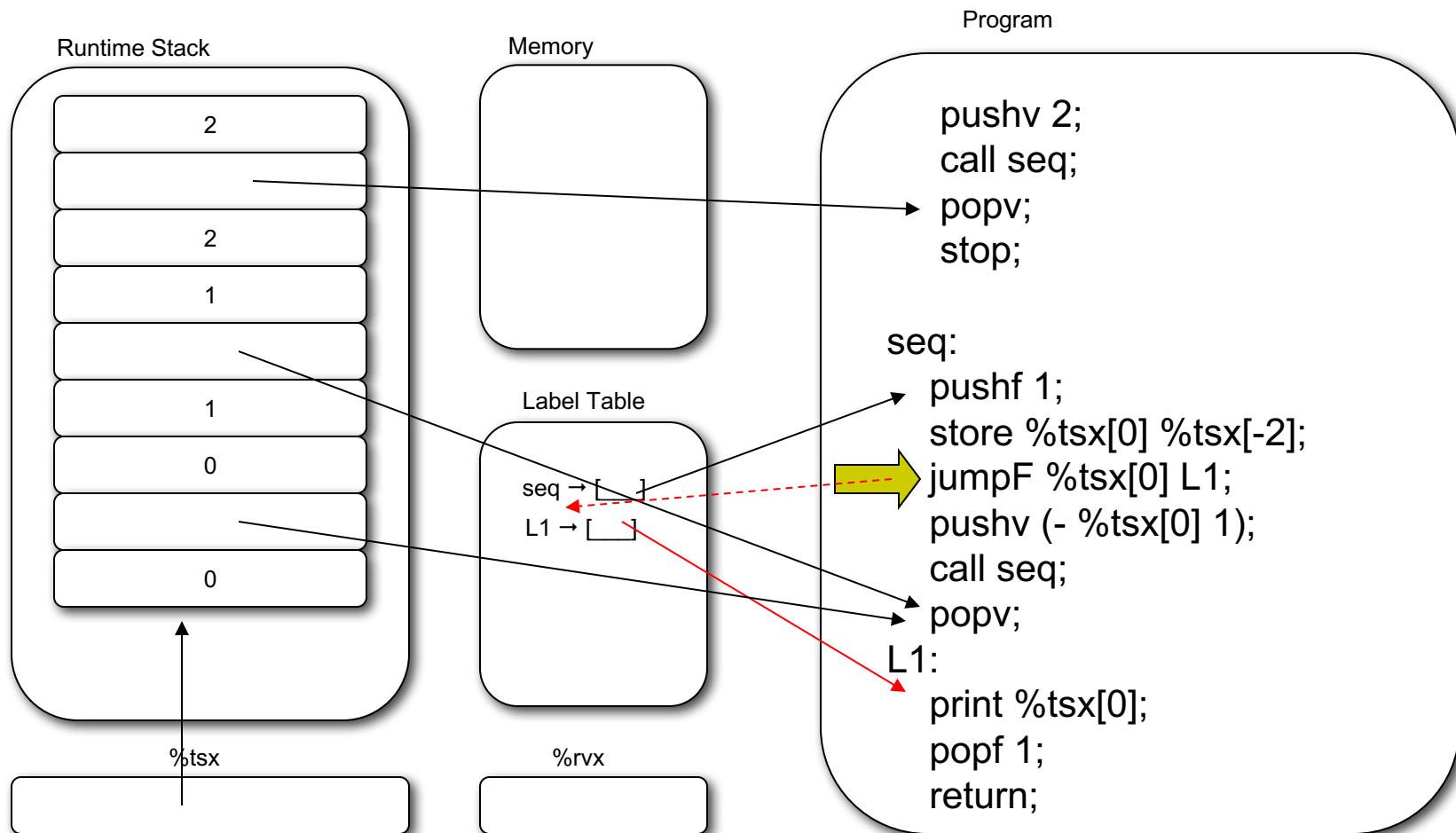


# Virtual Machine Design



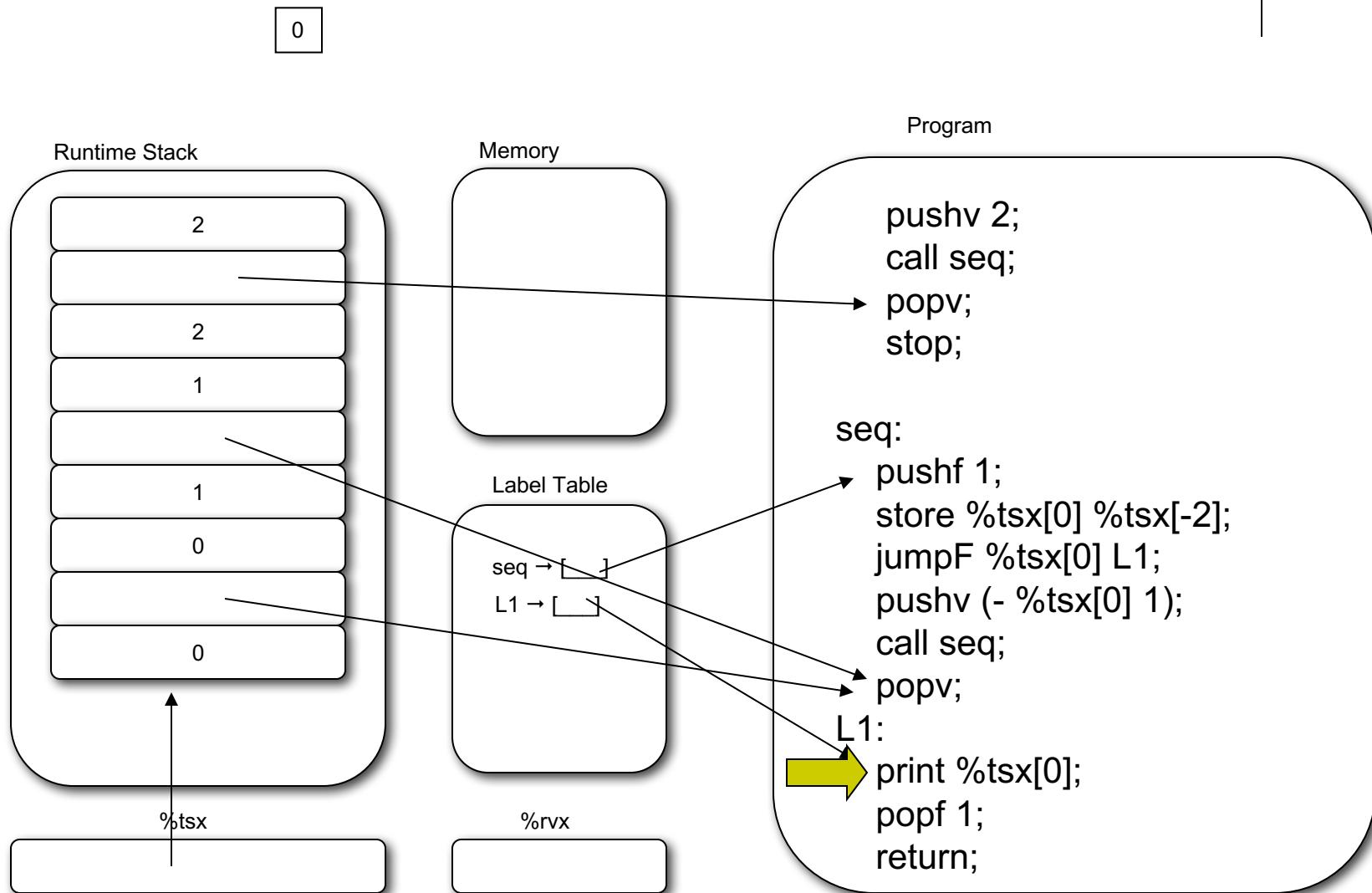


# Virtual Machine Design



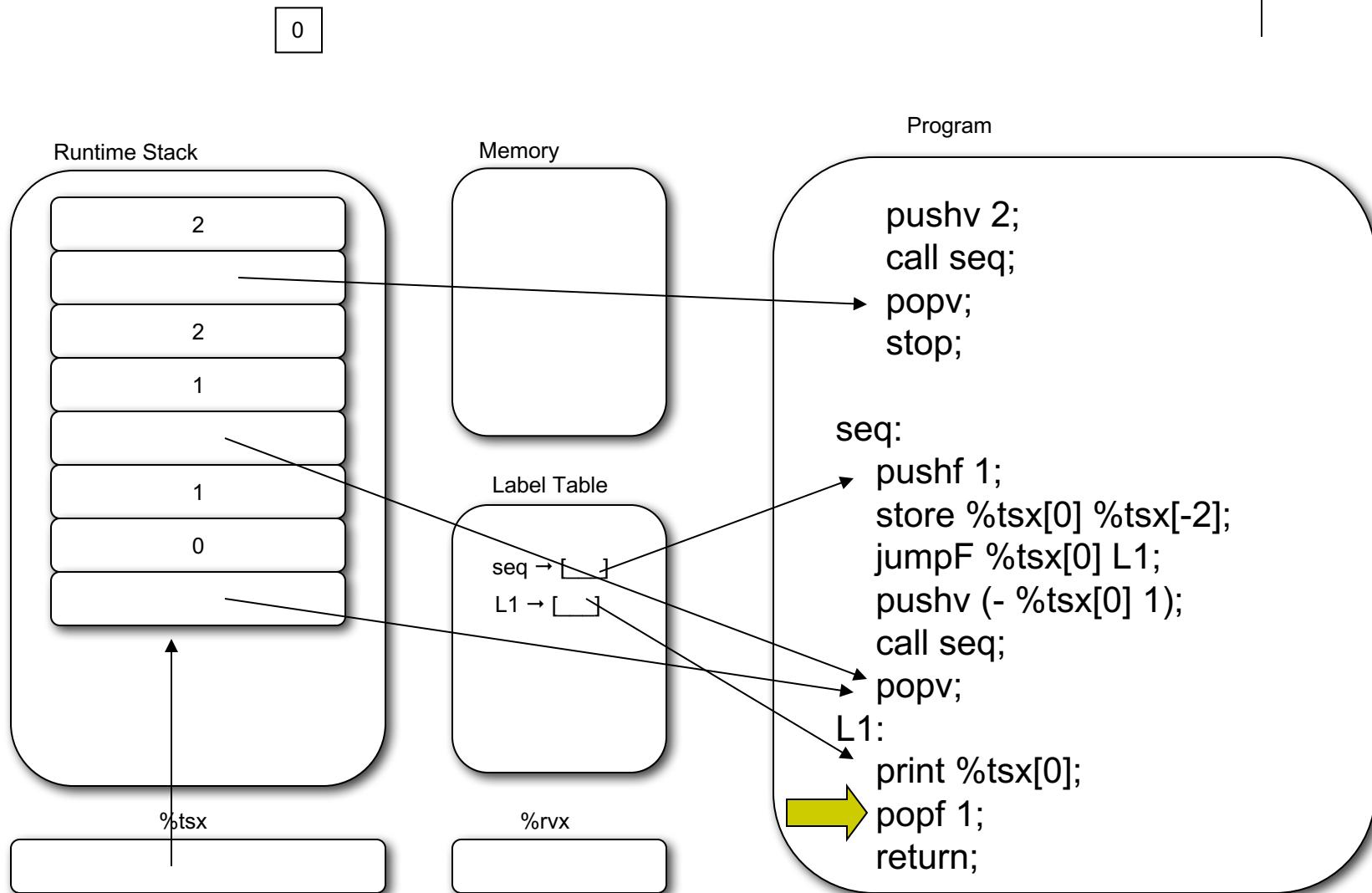


# Virtual Machine Design



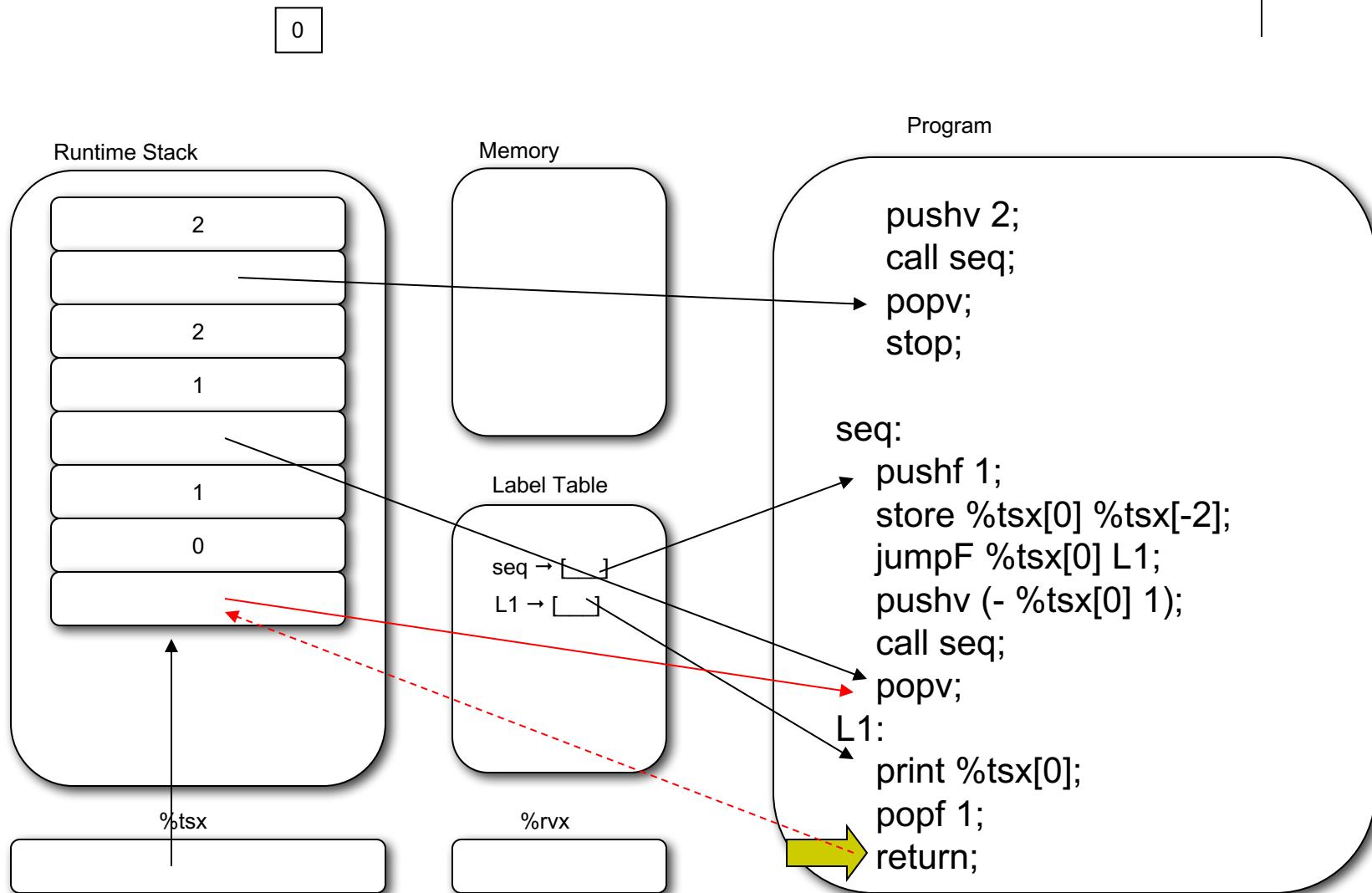


# Virtual Machine Design



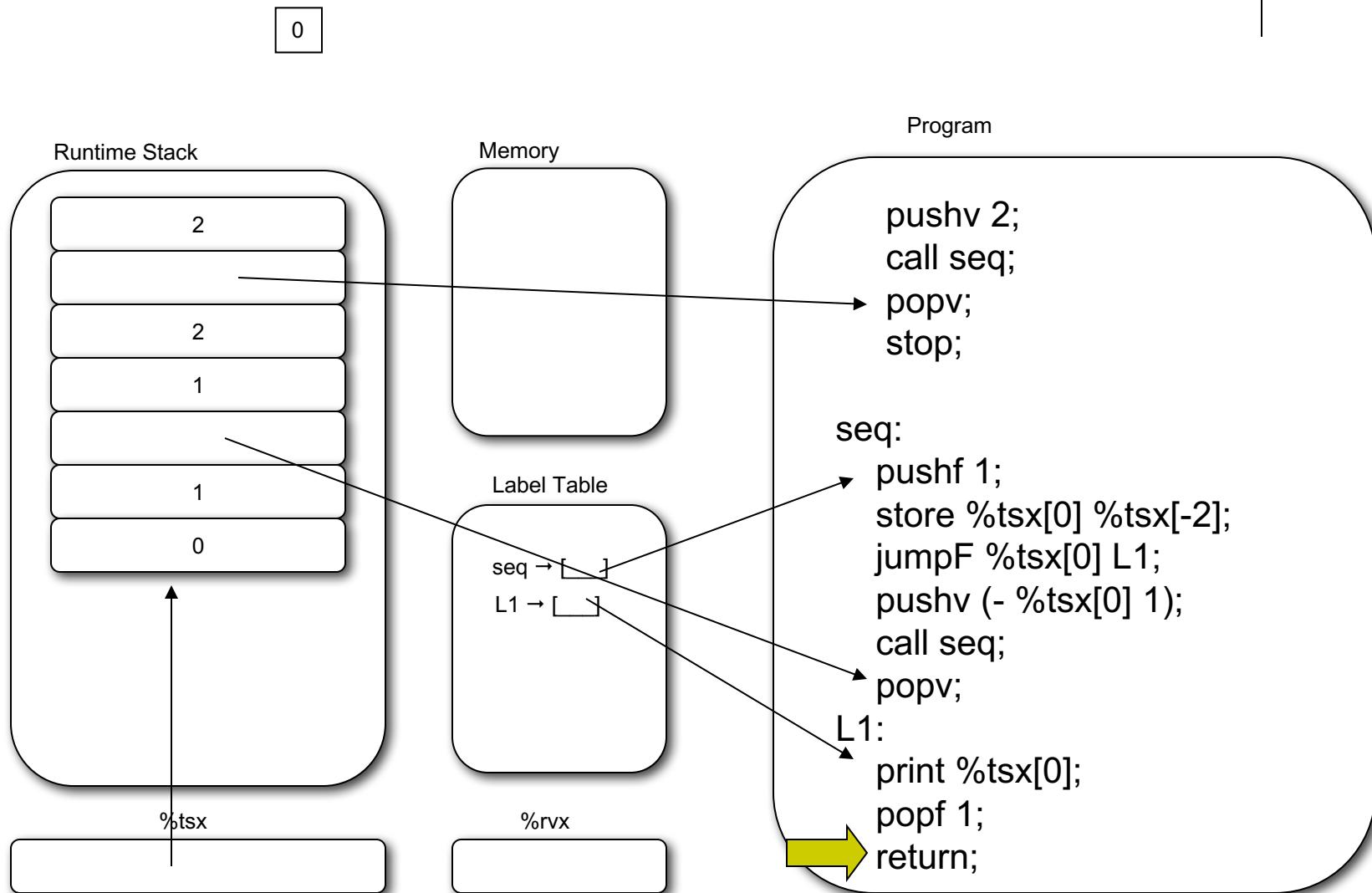


# Virtual Machine Design



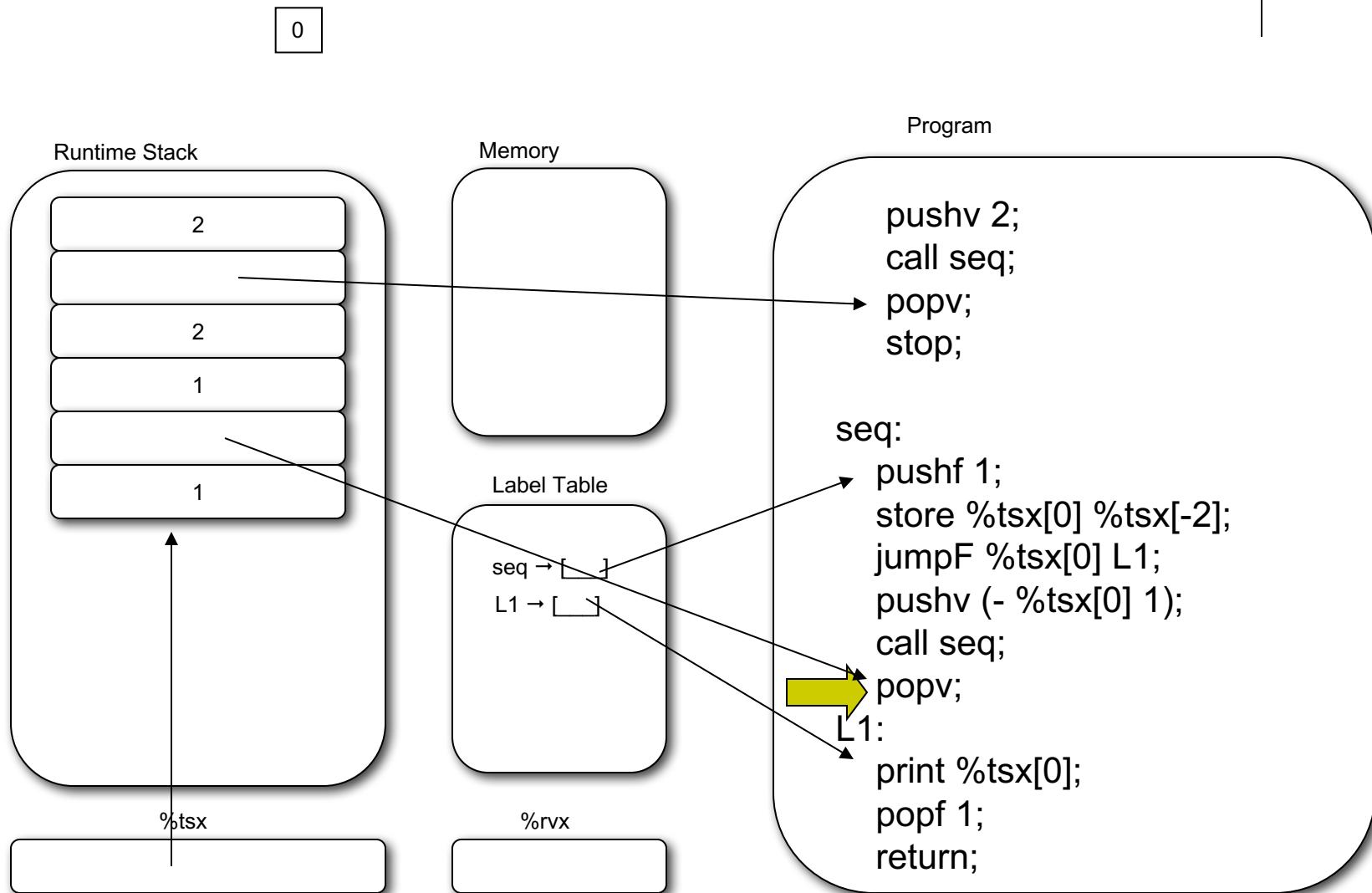


# Virtual Machine Design





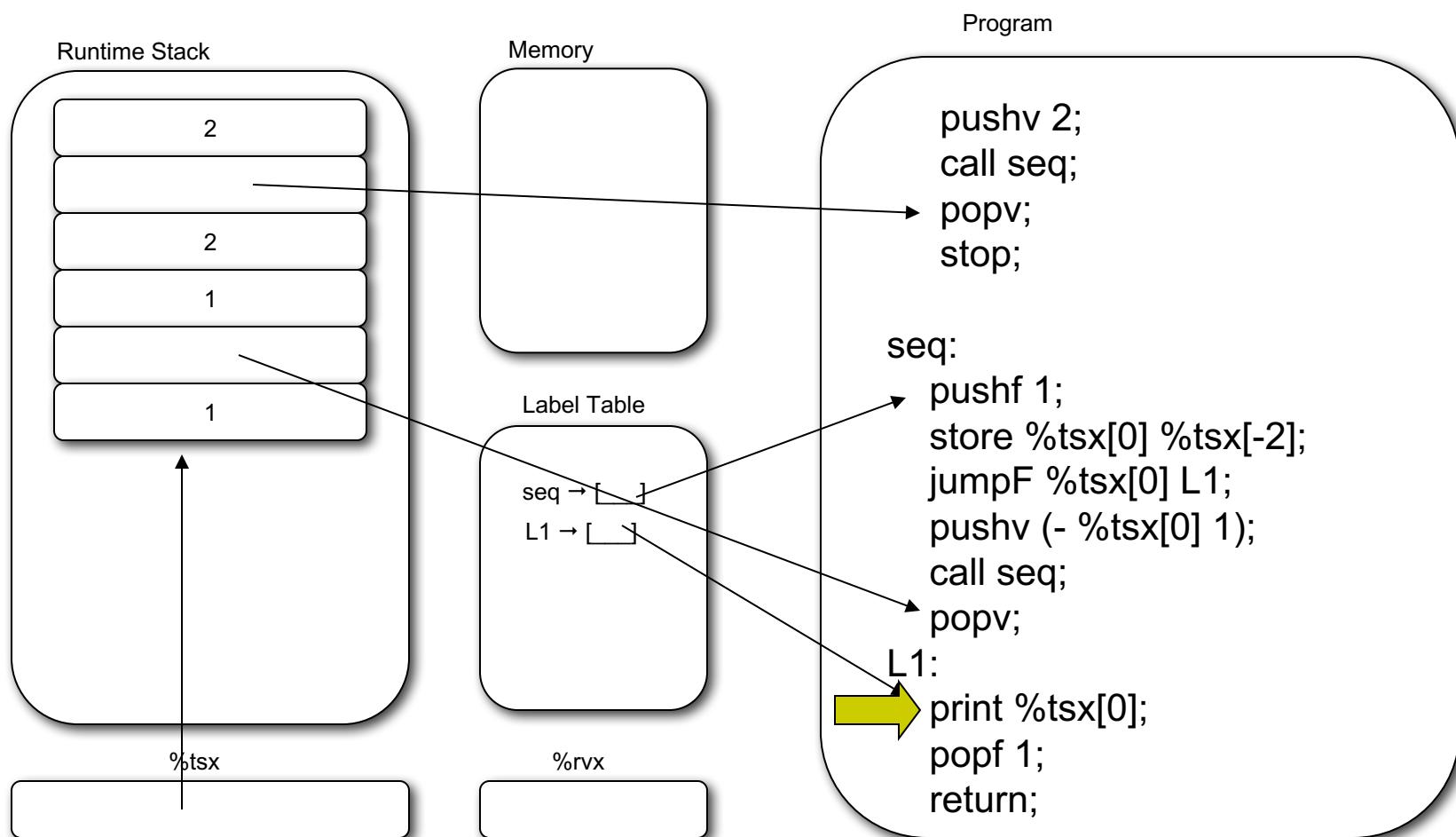
# Virtual Machine Design





# Virtual Machine Design

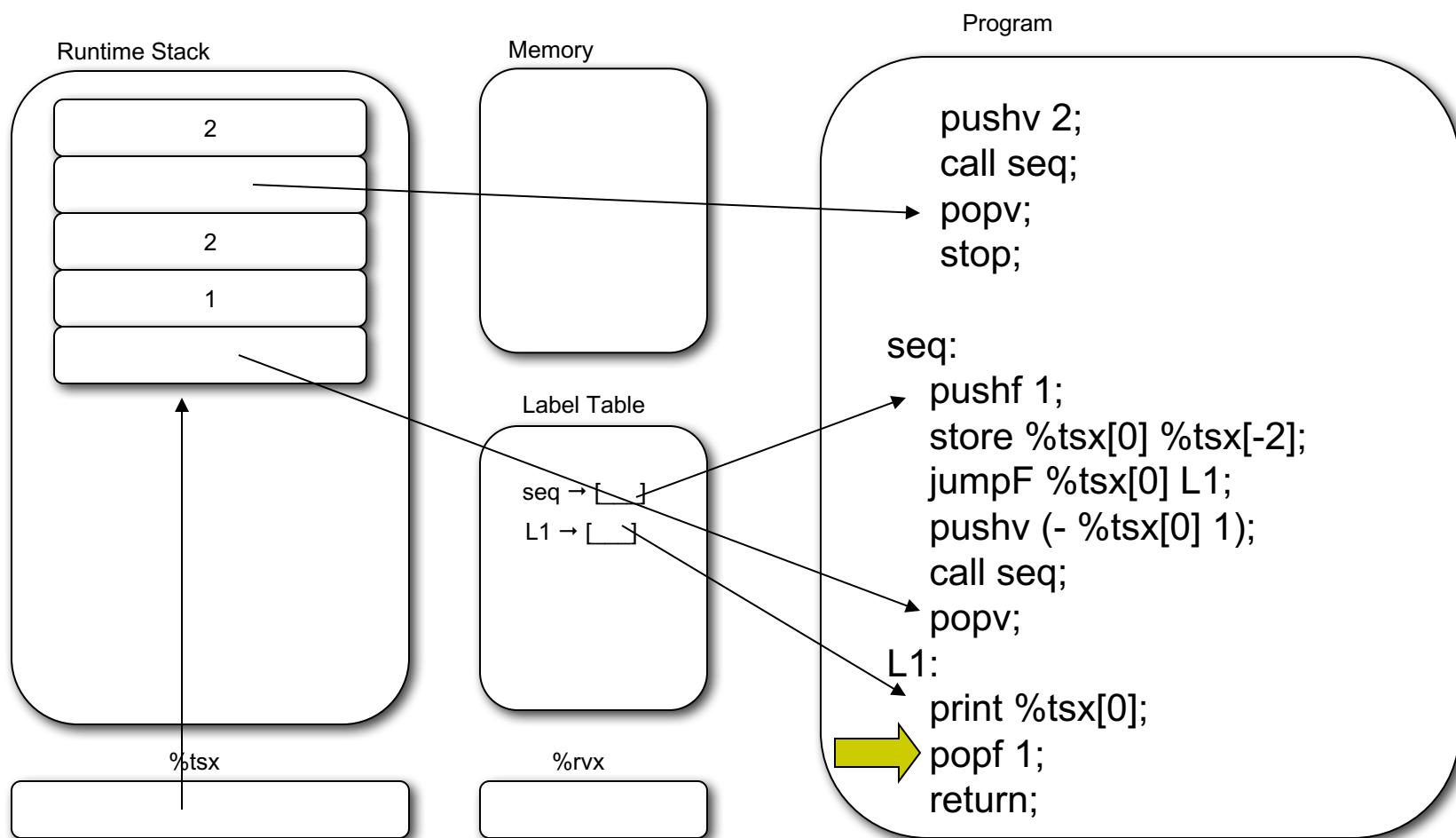
0 1





# Virtual Machine Design

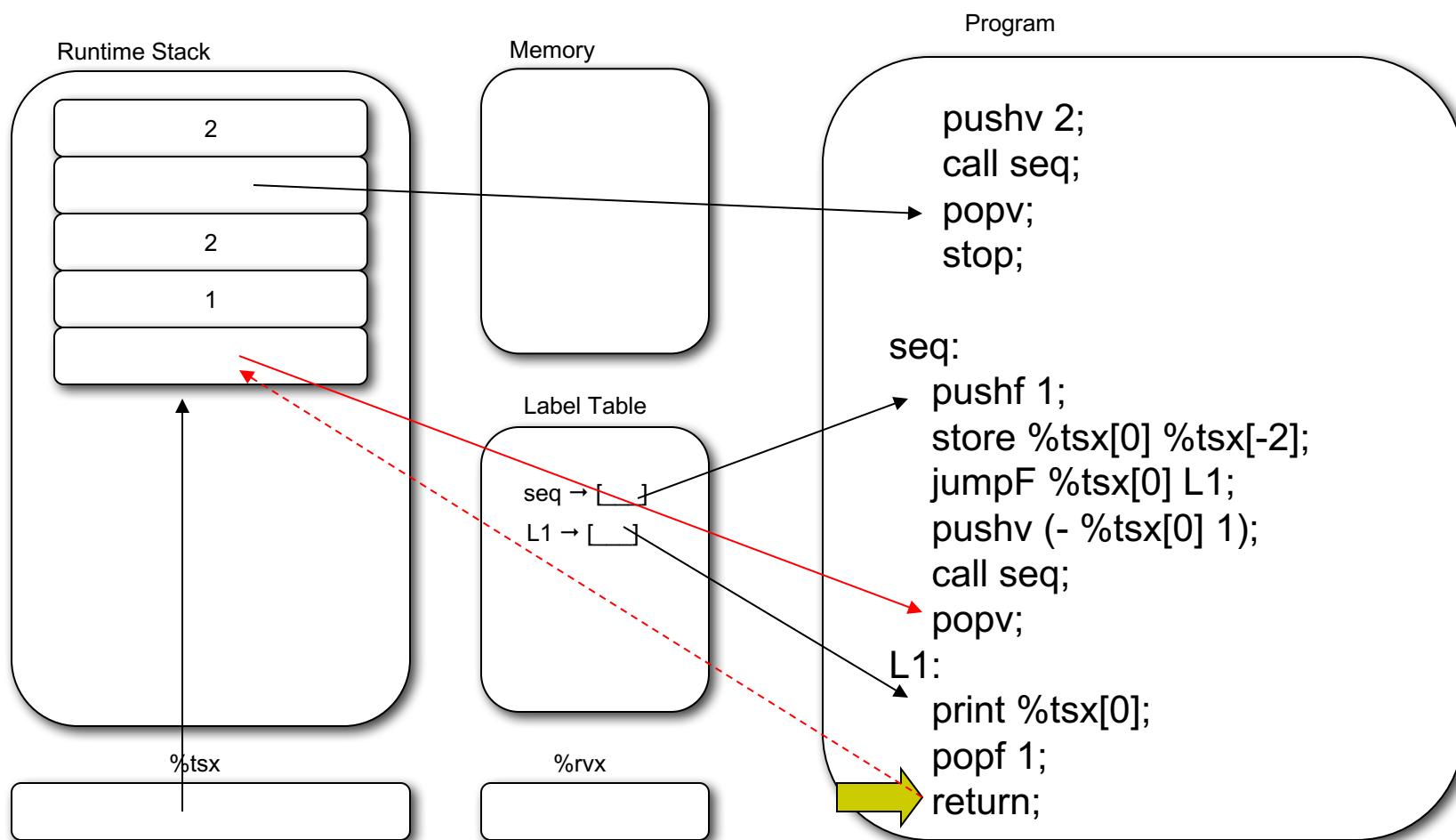
0 1





# Virtual Machine Design

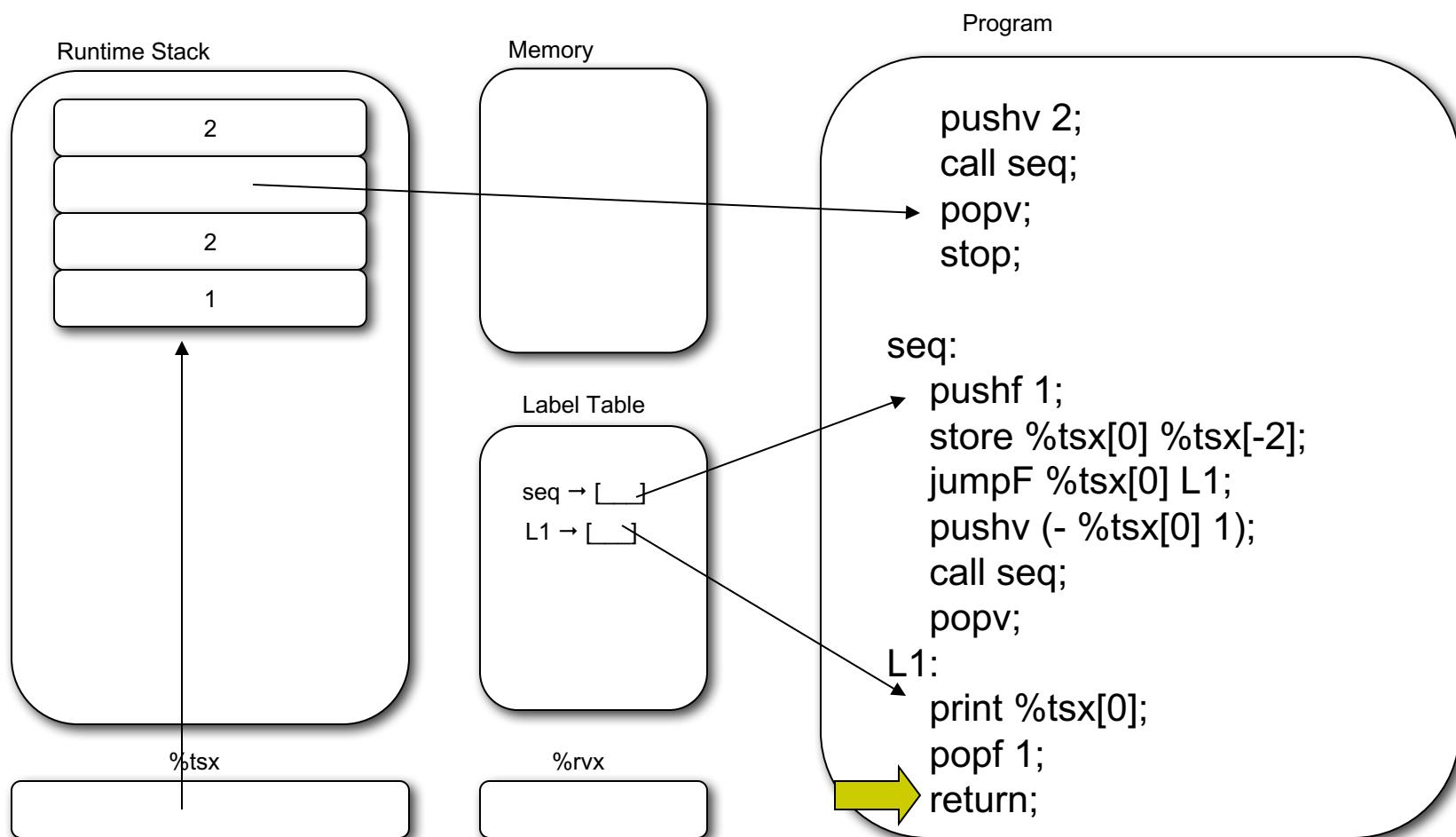
0 1





# Virtual Machine Design

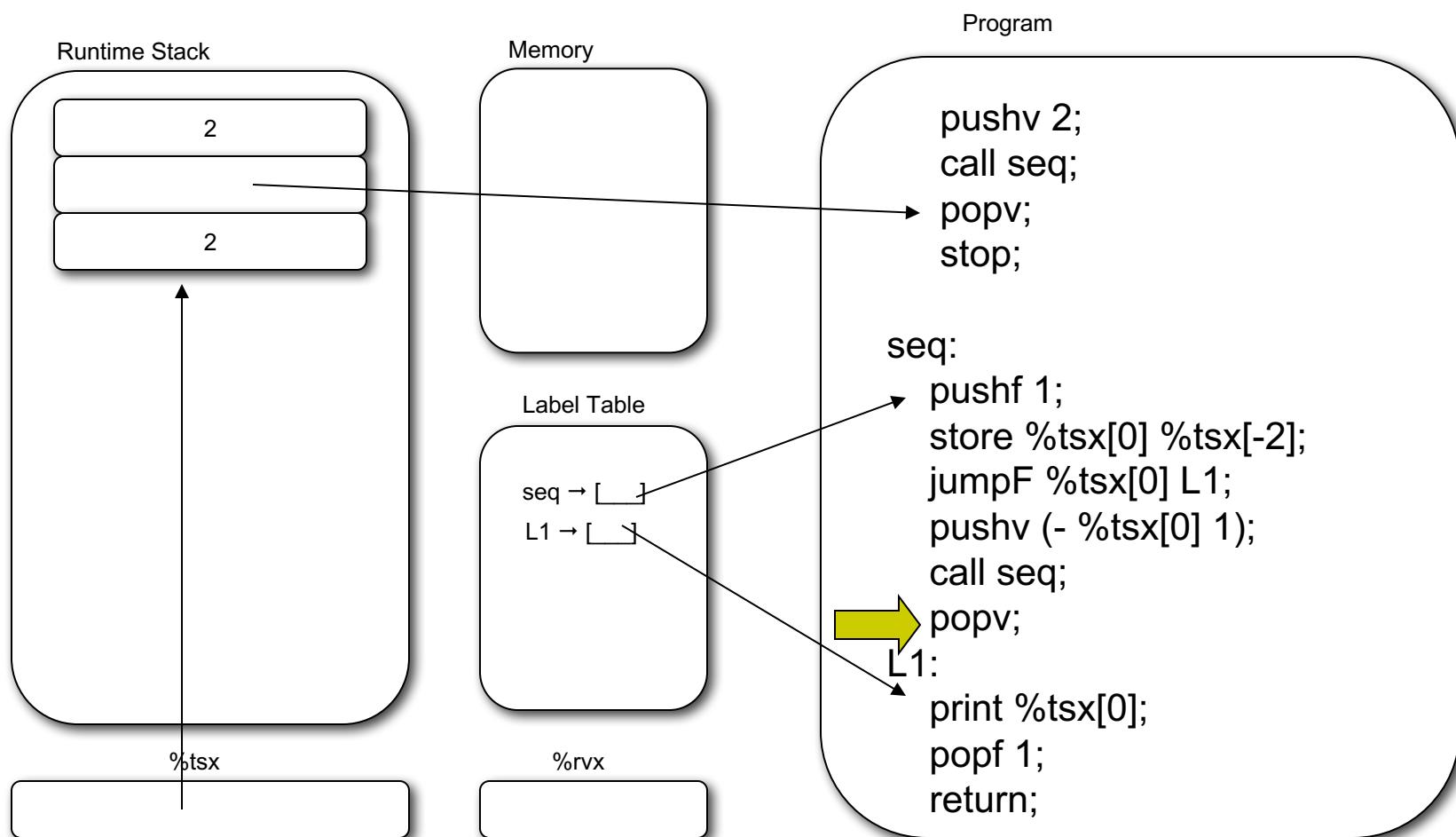
0 1





# Virtual Machine Design

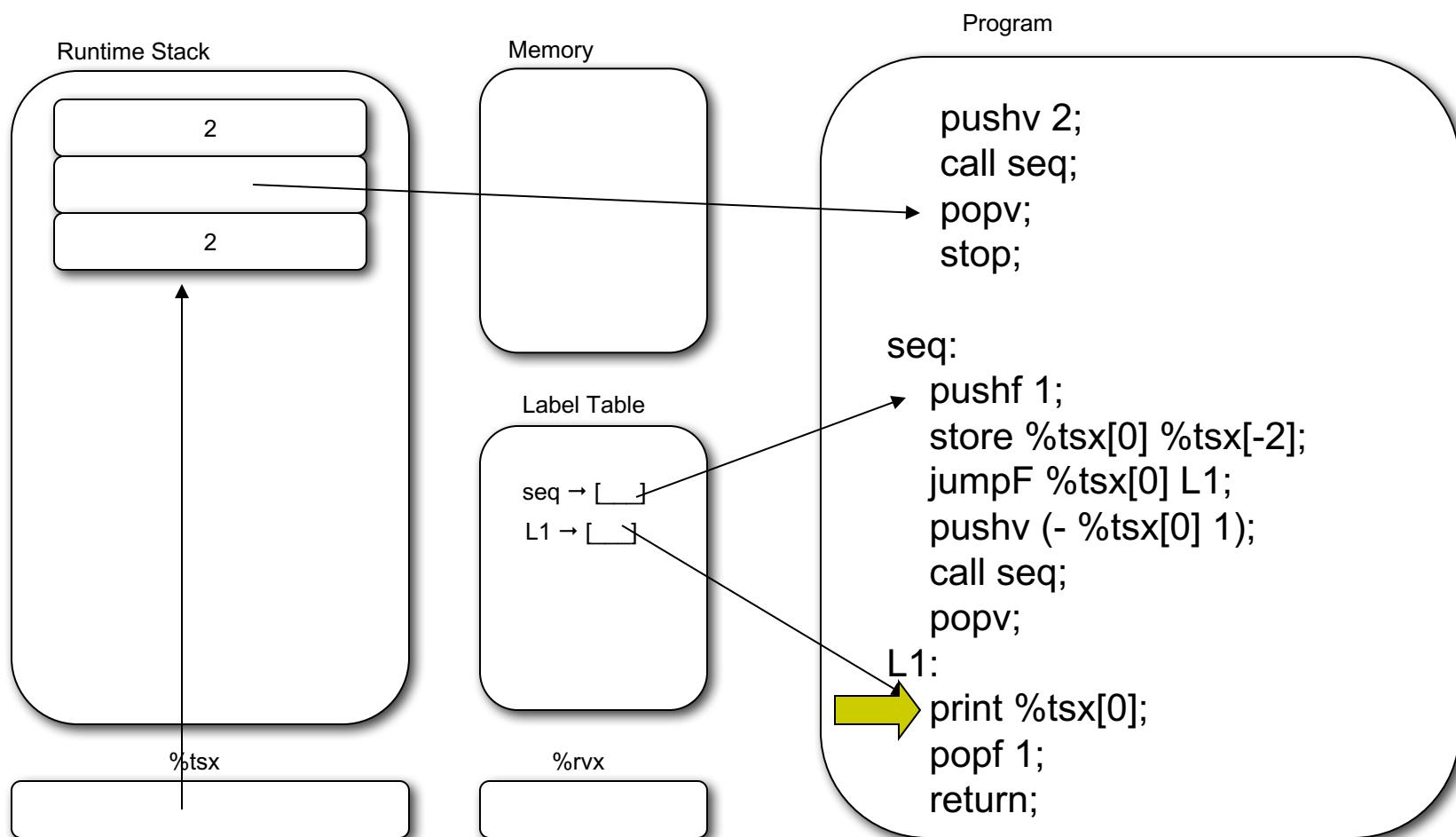
0 1





# Virtual Machine Design

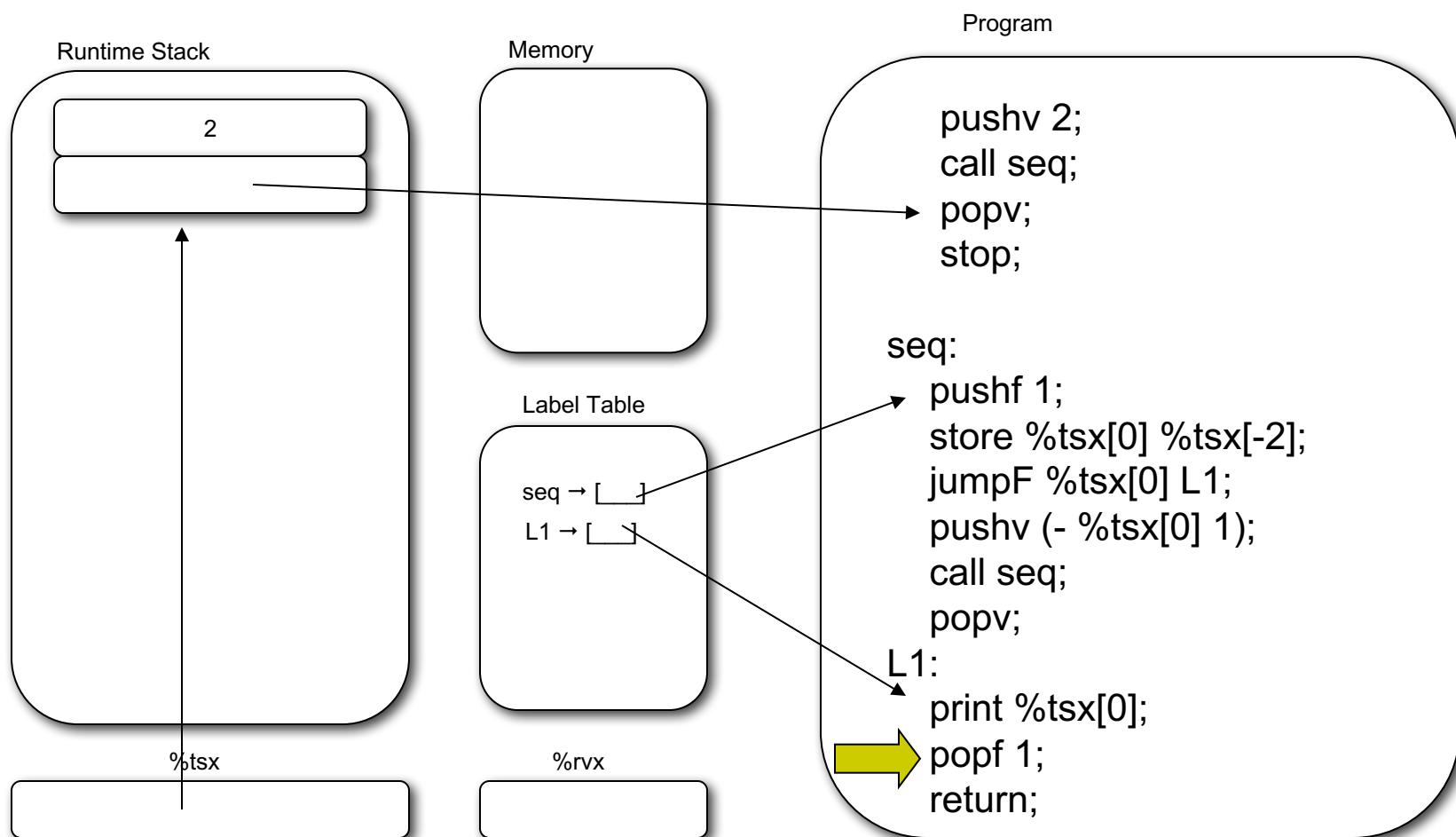
0 1 2

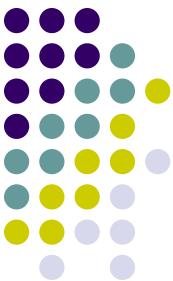




# Virtual Machine Design

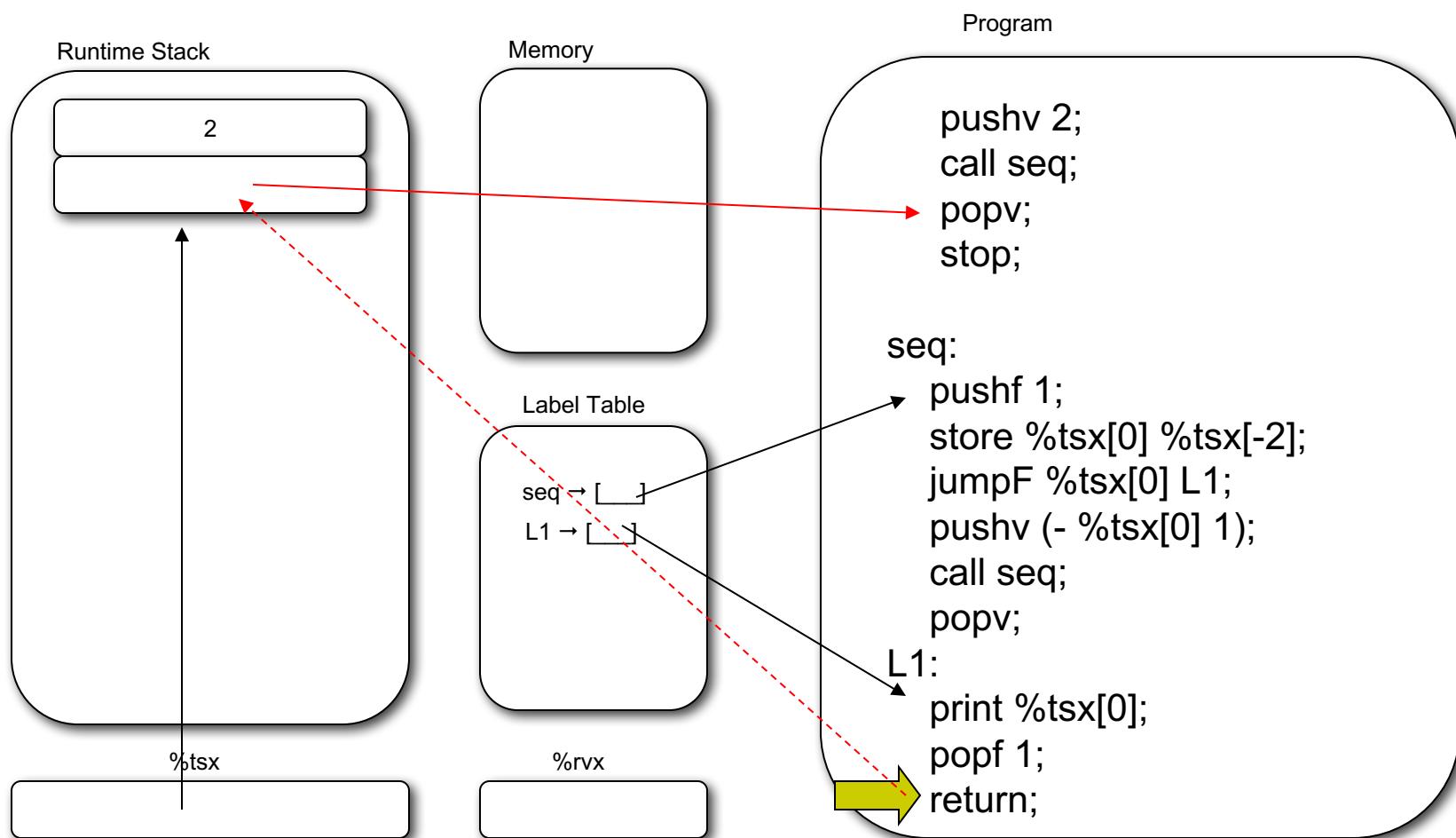
0 1 2





# Virtual Machine Design

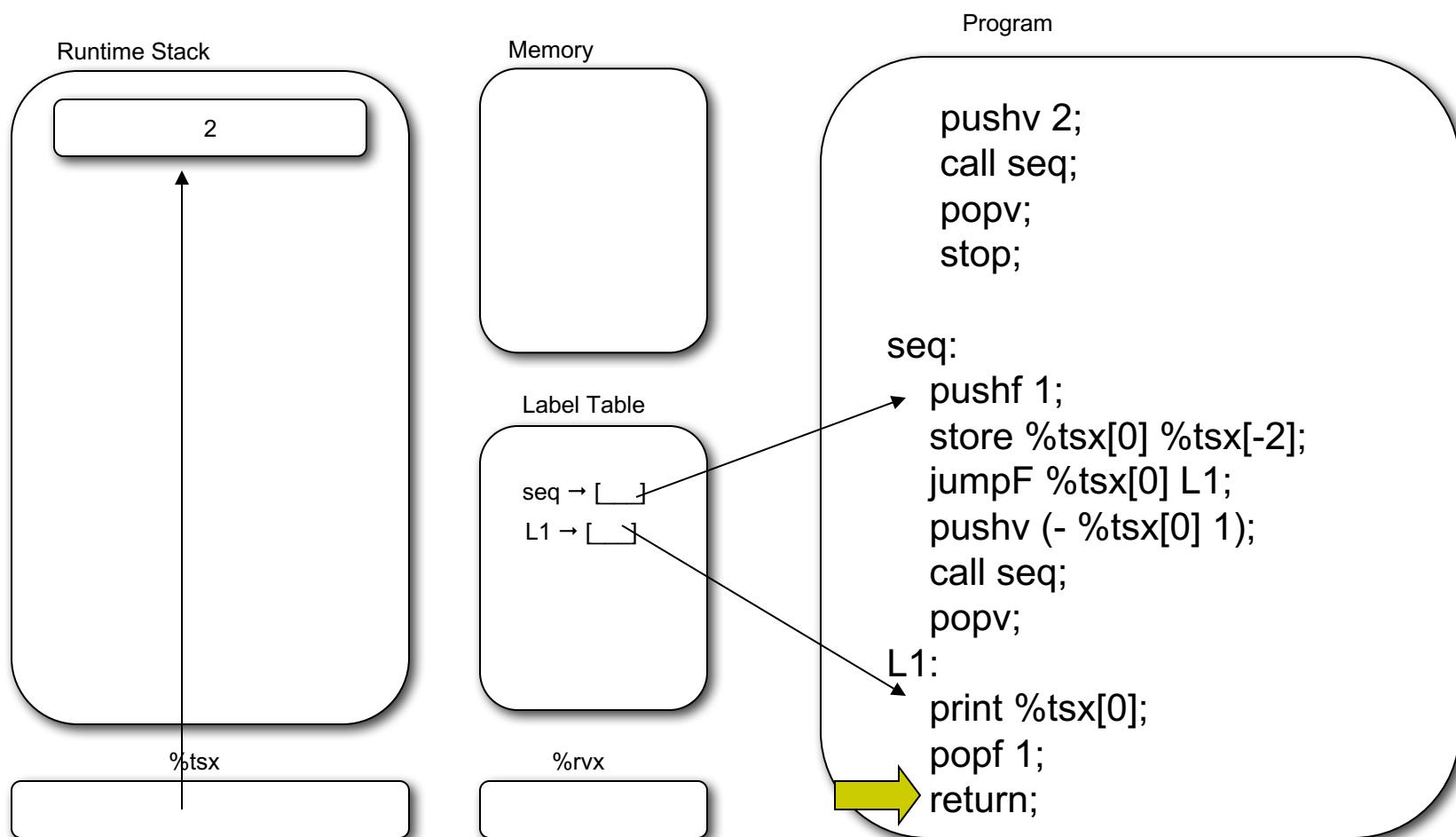
0 1 2





# Virtual Machine Design

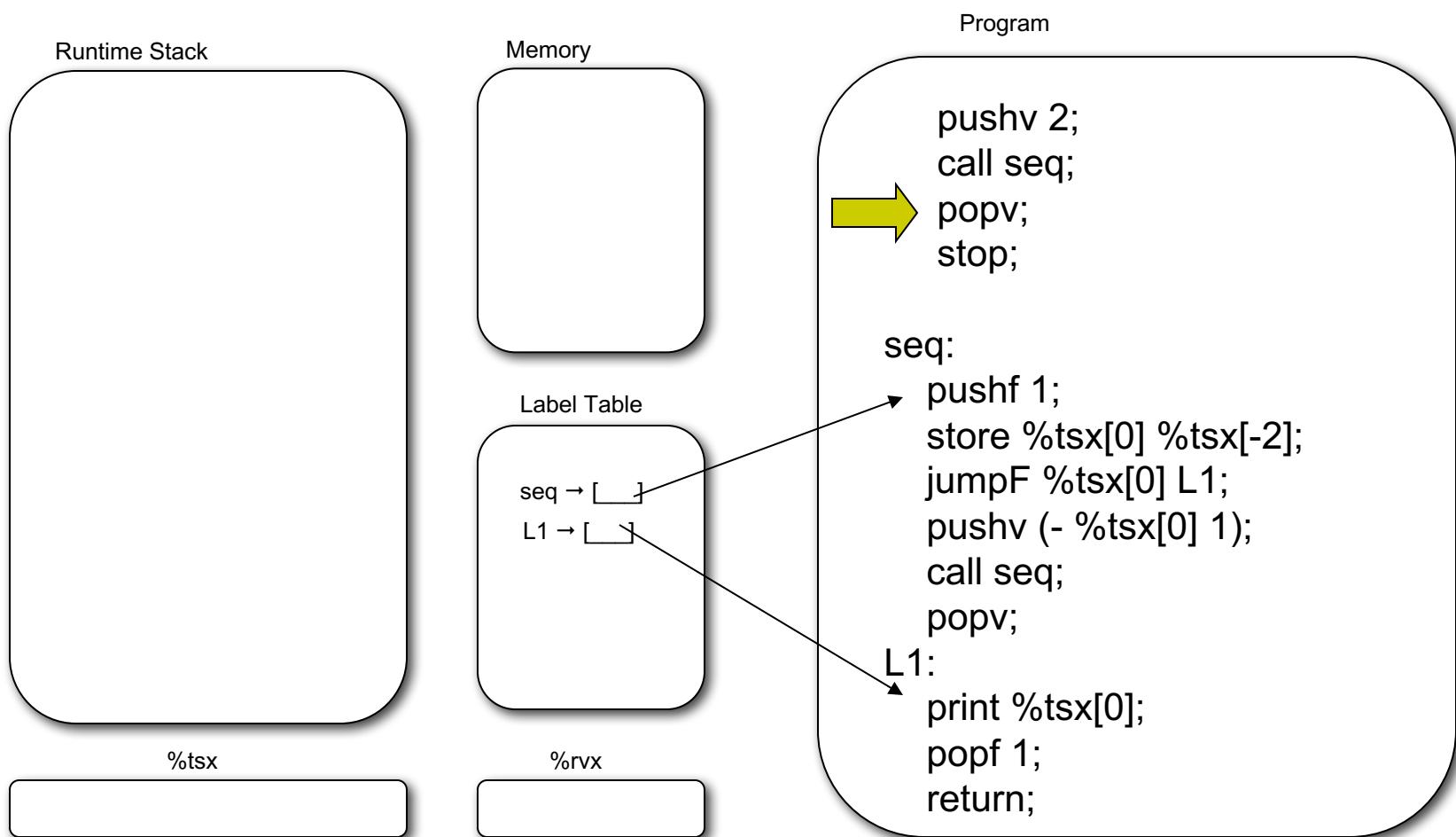
0 1 2





# Virtual Machine Design

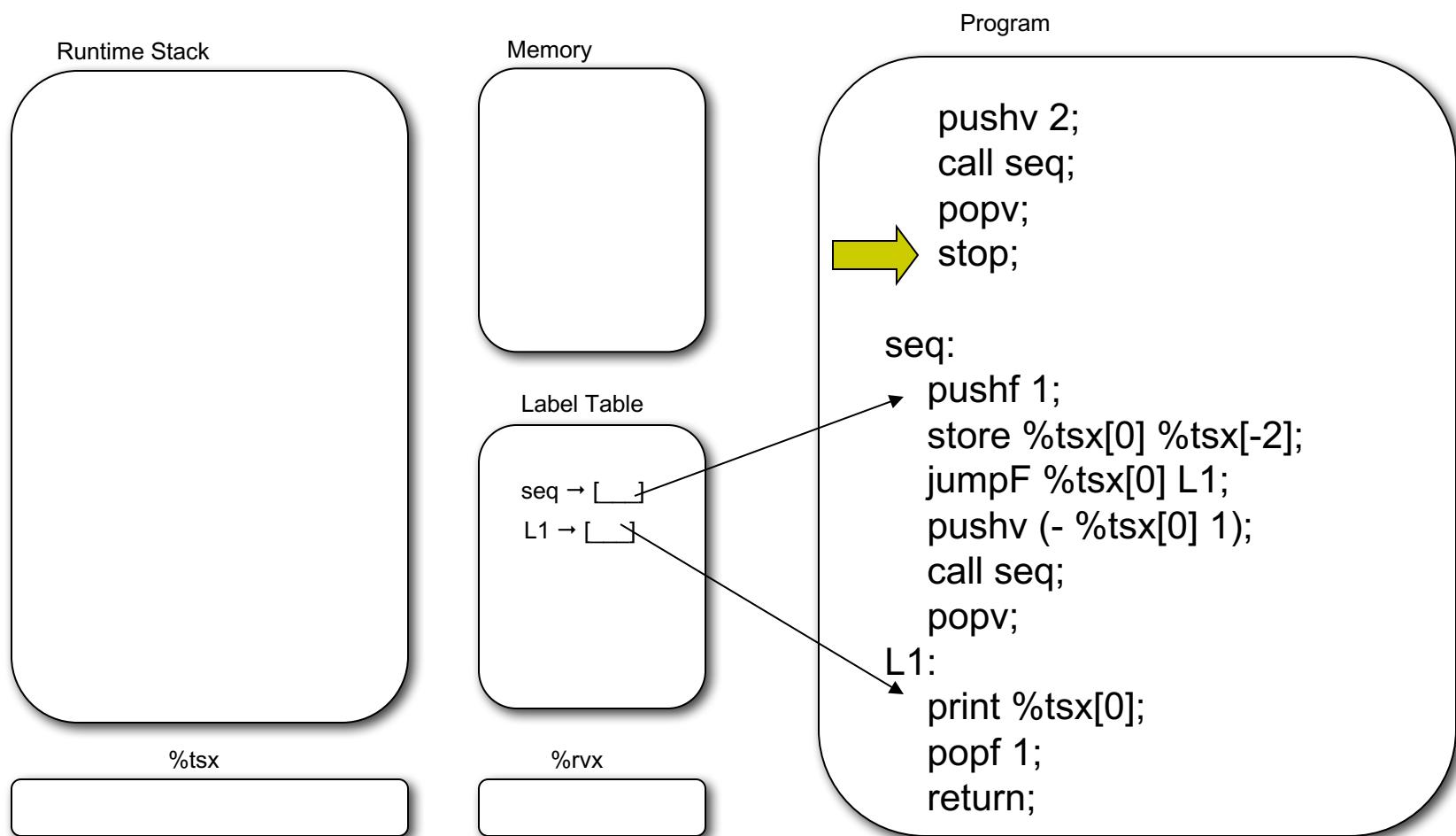
0 1 2





# Virtual Machine Design

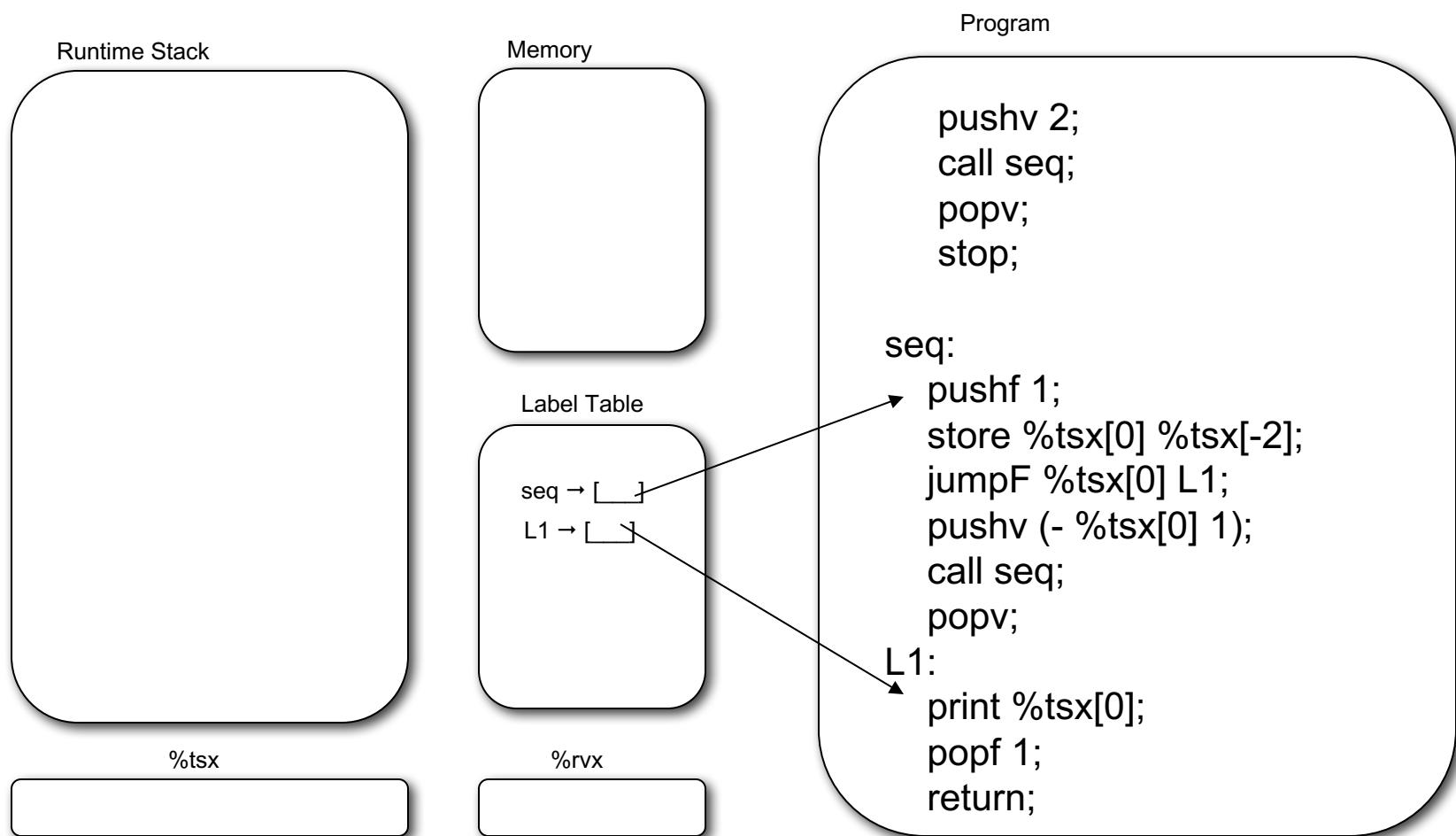
0 1 2





# Virtual Machine Design

0 1 2



# Exp2Bytecode – A Machine

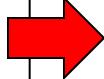
```
# define and initialize the structures of our abstract machine

class State:

    def __init__(self):
        self.initialize()

    def initialize(self):
        self.program = []
        self.symbol_table = dict()
        self.label_table = dict()
        self.runtime_stack = []
        self.instr_ix = 0
        self.rvx = None # return value register

state = State()
```

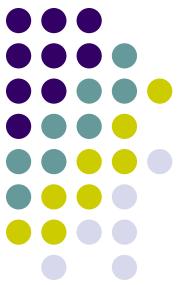


```
def p_labeled_instr(p):
    ...
    labeled_instr : label_def instr
    ...

    # if label exists record it in the label table
    if p[1]:
        state.label_table[p[1]] = state.instr_ix
    # append instr to program
    state.program.append(p[2])
    state.instr_ix += 1
```

```
def p_instr(p):
    ...
    instr : PRINT opt_string exp ';'
    | INPUT opt_string NAME ';'
    | STORE storable exp ';'
    | JUMPT exp label ';'
    | JUMPF exp label ';'
    | JUMP label ';'
    | CALL label ';'
    | RETURN ';'
    | PUSHV exp ';'
    | POPV opt_storable ';'
    | PUSHF size ';'
    | POPF size ';'
    | STOP ';'
    | NOOP ';'
    ...
    # for each instr assemble the appropriate tuple
    if p[1] == 'print':
        p[0] = ('print', p[2], p[3])
    elif p[1] == 'input':
        p[0] = ('input', p[2], p[3])
    elif p[1] == 'store':
        p[0] = ('store', p[2], p[3])
    elif p[1] == 'jumpT':
        p[0] = ('jumpt', p[2], p[3])
    elif p[1] == 'jumpF':
        p[0] = ('jumpf', p[2], p[3])
    elif p[1] == 'jump':
        p[0] = ('jump', p[2])
    elif p[1] == 'call':
        p[0] = ('call', p[2])
    elif p[1] == 'return':
        p[0] = ('return',)
    elif p[1] == 'pushv':
        p[0] = ('pushv', p[2])
    elif p[1] == 'popv':
        p[0] = ('popv', p[2])
    elif p[1] == 'pushf':
        p[0] = ('pushf', p[2])
    elif p[1] == 'popf':
        p[0] = ('popf', p[2]) ...
```

# Exp2Bytecode – Abstract Machine



- As in exp1bytecode – the interpreter is just one big loop that interprets the instructions on the program list.

```
def interp_program():
    'execute abstract bytecode machine'

    # We cannot use the list iterator here because we
    # need to be able to interpret jump instructions

    # start at the first instruction in program
    state.instr_ix = 0

    # keep interpreting until we run out of instructions
    # or we hit a 'stop'
    while True:
        if state.instr_ix == len(state.program):
            # no more instructions
            break
        else:
            # get instruction from program
            instr = state.program[state.instr_ix]

            # instruction format: (type, [arg1, arg2, ...])
            type = instr[0]
```

Interpret instructions here...

# Exp2Bytecode – Abstract Machine



```
elif type == 'store':
    # STORE storables exp
    storables = instr[1] # storables itself is a tuple (type, children...)
    val = eval_exp_tree(instr[2])
    do_storable(storables, val) ←
    state.instr_ix += 1

elif type == 'call':
    # call label
    # push the current instruction pointer
    state.runtime_stack.append(state.instr_ix) ←
    # get the target address from the label table
    label = instr[1]
    state.instr_ix = state.label_table.get(label, None)

    # print("jumping to instruction {}".format(state.instr_ix))

elif type == 'return':
    # return
    # pop the return address off the stack and jump to it
    state.instr_ix = state.runtime_stack.pop() ←
    state.instr_ix += 1
```

```
elif type == 'pushv':
    # pushv exp
    exp_tree = instr[1]
    val = eval_exp_tree(exp_tree)
    # push value onto stack
    state.runtime_stack.append(val)

    state.instr_ix += 1

elif type == 'popv':
    # popv opt_storable
    storables = instr[1]
    val = state.runtime_stack.pop()

    if storables:
        do_storable(storables, val)

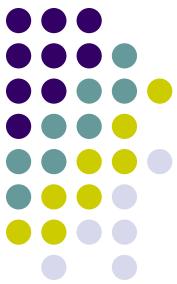
    state.instr_ix += 1
```

```
elif type == 'pushf':
    # pushf size_exp
    size_val = eval_exp_tree(instr[1])

    # pushing a stack frame onto the stack
    # zeroing out each stack location in the frame
    for i in range(size_val):
        state.runtime_stack.append(0)

    state.instr_ix += 1
```

# Exp2Bytecode – Abstract Machine



```
def do_storable(storable, val):
    # store the value in the appropriate storables

    tsx = get_tsx()

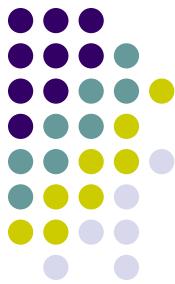
    if storable[0] == 'id':
        # ('id', name)
        name = storable[1]
        state.symbol_table[name] = val

    elif storable[0] == '%rvx':
        # ('%rvx',)
        state.rvx = val

    elif storable[0] == '%tsx':
        # ('%tsx', opt_offset_exp)
        if storable[1]:
            offset = eval_exp_tree(storable[1])
            state.runtime_stack[tsx+offset] = val
        else:
            state.runtime_stack[tsx] = val

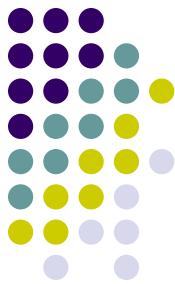
    else:
        raise ValueError("Unknown storables {}".format(storable[0]))
```

# Function Calls on Real Machines



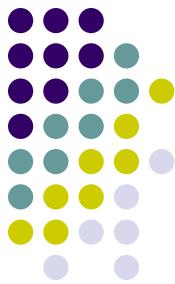
- Function calls on real machines work similar to the function calls on our bytecode virtual machine.
- The main difference is the level of abstraction of the instruction set.
- We show examples of programs compiled on an Intel Mac.

# Function Calls on Real Machines



- Functions on real machines are broken into three parts:
  - Function prologue - set up the stack environment in which the function executes
  - Function code - the code in the body of the function
  - Function epilogue - clean up the stack.
- On the Intel chip the function prologue looks something like this:
  - `pushl %ebp // push the basepointer/framepointer on the stack`
  - `movl %esp, %ebp // make the current stack pointer the current frame pointer`
  - `subl $n, %esp // make a new frame of size n`

# Function Calls on Real Machines



- The function epilogue usually looks like this:
  - `movl %ebp, %esp`
  - `popl %ebp`
  - `ret`
- The [x86](#) processor contains a built-in instruction which performs part of the epilogue. The following code is equivalent to the above code:
  - `leave`
  - `ret`



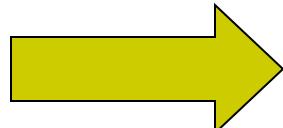
# The Add Program

Compiling a C program on an Intel Mac.

call.c

```
int add(int a, int b)
{
    return a+b;
}

int main() {
    int x = add(3,2);
}
```



call.s

```
.text
.globl _add
_add:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    leave
    ret

.globl _main
_main:
    pushl %ebp
    movl %esp, %ebp
    subl $40, %esp
    movl $2, 4(%esp)
    movl $3, (%esp)
    call _add
    movl %eax, -12(%ebp)
    leave
    ret

.subsections_via_symbols
```

```
gcc -S -fno-asynchronous-unwind-tables -fno-dwarf2-cfi-asm call.c
```



# The Add Program

```
.text
.globl _add
_add:
    pushl %ebp          // save base ptr onto stack
    movl %esp, %ebp     // move stack ptr to base ptr
    subl $8, %esp        // make a new frame on stack
    movl 12(%ebp), %eax // get argument a at offset 12 from base ptr, store in register %eax
    addl 8(%ebp), %eax  // get argument b at offset 8 from base pts, add to %eax
    leave               // remove frame
    ret                 // return from call

.globl _main
_main:
    pushl %ebp          // save base ptr onto stack
    movl %esp, %ebp     // move stack ptr to base ptr
    subl $40, %esp       // make a new frame on stack
    movl $2, 4(%esp)     // move constant 2 to TOS offset 4
    movl $3, (%esp)      // move constant 3 to TOS
    call _add            // call function add
    movl %eax, -12(%ebp) // retrieve result from eax register and store it at the ‘location’ for x.
    leave               // clean up the stack
    ret                 // return to OS

.subsections_via_symbols
```

See [http://en.wikipedia.org/wiki/Function\\_prologue](http://en.wikipedia.org/wiki/Function_prologue) and <http://en.wikipedia.org/wiki/X86>

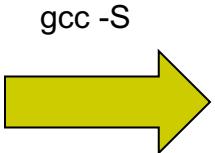


# The Recursive Seq Program

seq.c

```
void seq(int x) {
    if (1<=x)
        seq(x-1);
    print(x);
}

int main() {
    seq(2);
}
```



gcc -S

seq.s

```
.text
.globl _seq
_seq:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    cmpl $0, 8(%ebp)
    jle L2
    movl 8(%ebp), %eax
    decl %eax
    movl %eax, (%esp)
    call _seq

L2:
    movl 8(%ebp), %eax
    movl %eax, (%esp)
    call L_print$stub
    leave
    ret

.globl _main
_main:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    movl $2, (%esp)
    call _seq
    leave
    ret
```

We have more to say about this when we look at compiling programs for actual machines.