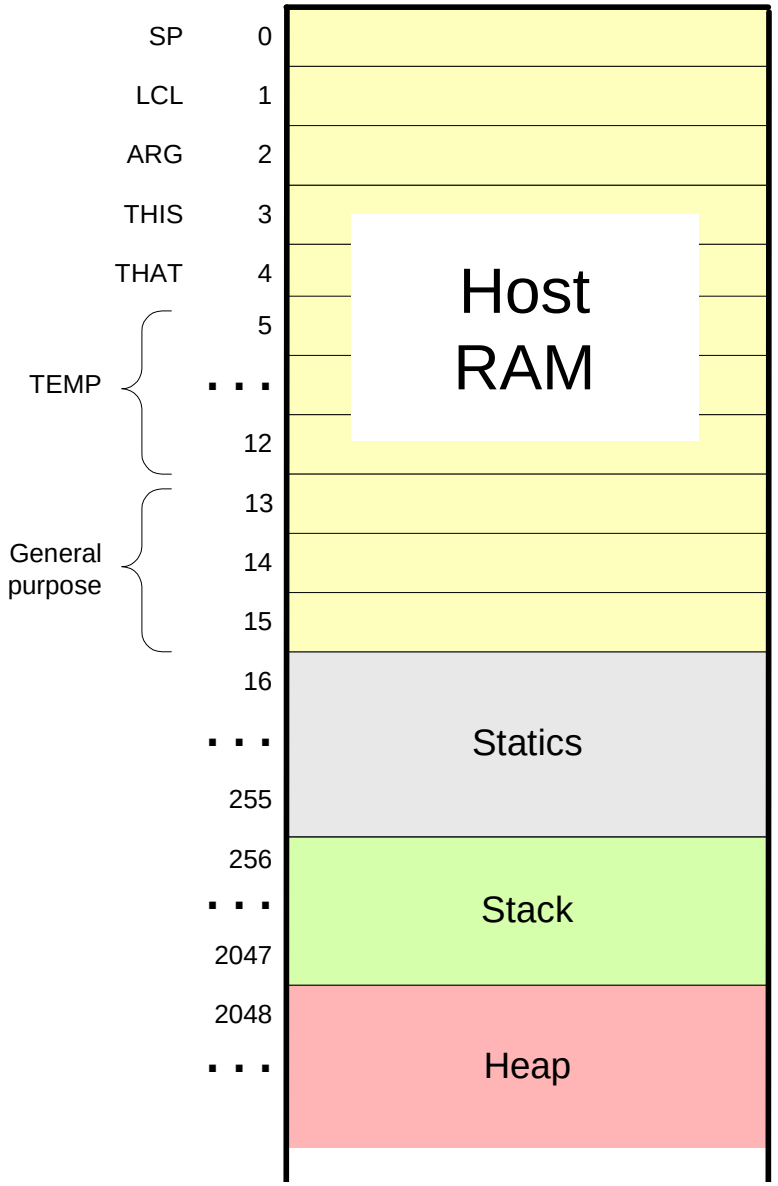
VM implementation on the Hack platform



Basic idea: the mapping of the stack and the global segments on the RAM is easy (fixed); the mapping of the function-level segments is dynamic, using pointers

The stack: mapped on RAM[256 ... 2047];
The stack pointer is kept in RAM address SP

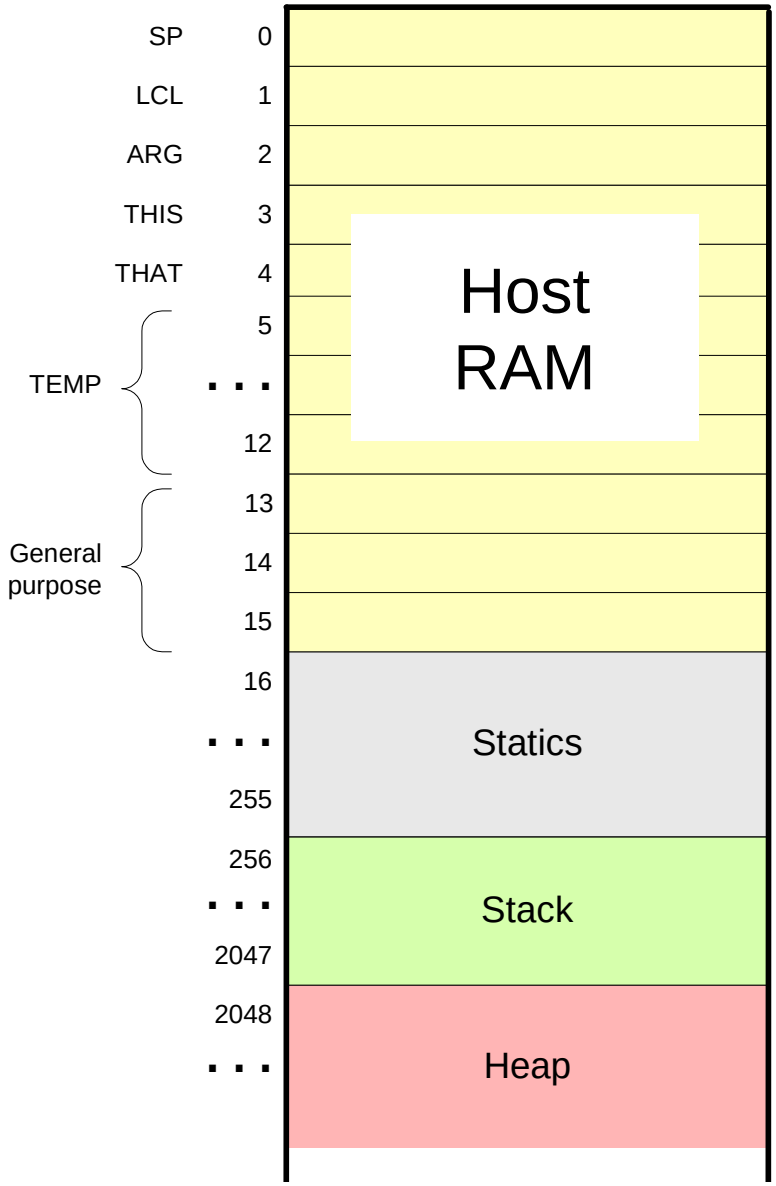
static: mapped on RAM[16 ... 255];
each segment reference static *i* appearing in a VM file named f is compiled to the assembly language symbol f.i (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

local,argument,this,that: these method-level segments are mapped somewhere from address 2048 onward, in an area called "heap". The base addresses of these segments are kept in RAM addresses LCL, ARG, THIS, and THAT. Access to the *i*-th entry of any of these segments is implemented by accessing RAM[segmentBase + *i*]

constant: a truly a virtual segment:
access to constant *i* is implemented by supplying the constant *i*.

pointer: discussed later.

VM implementation on the Hack platform



Practice exercises

Now that we know how the memory segments are mapped on the host RAM, we can write Hack commands that realize the various VM commands. for example, let us write the Hack code that implements the following VM commands:

- `push constant 1`
- `pop static 7` (suppose it appears in a VM file named f)
- `push constant 5`
- `add`
- `pop local 2`
- `eq`

Tips:

1. The implementation of any one of these VM commands requires several Hack assembly commands involving pointer arithmetic (using commands like `A=M`)
2. If you run out of registers (you have only two ...), you may use `R13`, `R14`, and `R15`.

Function call-and-return conventions

Calling function

```
function demo 3
  ...
  push constant 7
  push constant 2
  add
  push constant 3
  call mult
  ...
```

called function aka "callee" (example)

```
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  ...          // rest of code ommitted
label end
  push local 0  // push result
  return
```

Although not obvious in this example, every VM function has a private set of 5 memory segments (`local`, `argument`, `this`, `that`, `pointer`)

These resources exist as long as the function is running.

Call-and-return programming convention

- The caller must push the necessary argument(s), call the callee, and wait for it to return
- Before the callee terminates (returns), it must push a return value
- At the point of return, the callee's resources are recycled, the caller's state is re-instated, execution continues from the command just after the `call`
- **Caller's net effect:** the arguments were replaced by the return value (just like with primitive commands)

Behind the scene

- Recycling and re-instating subroutine resources and states is a major headache
- Some agent (either the VM or the compiler) should manage it behind the scene "like magic"
- In our implementation, the magic is VM / stack-based, and is considered a great CS gem.

The function-call-and-return protocol

```
function g nVars
call g nArgs
return
```

The caller's view:

- Before calling a function *g*, I must push onto the stack as many arguments as needed by *g*
- Next, I invoke the function using the command `call g nArgs`
- After *g* returns:
  - The arguments that I pushed before the call have disappeared from the stack, and a return value (that always exists) appears at the top of the stack
  - All my memory segments (`local`, `argument`, `this`, `that`, `pointer`) are the same as before the call.

Blue = VM function writer's responsibility

Black = black box magic, delivered by the VM implementation

Thus, the VM implementation writer must worry about the "black operations" only.

The callee's (*g* 's) view:

- When I start executing, my `argument` segment has been initialized with actual argument values passed by the caller
- My `local` variables segment has been allocated and initialized to zero
- The `static` segment that I see has been set to the `static` segment of the VM file to which I belong, and the working stack that I see is empty
- Before exiting, I must push a value onto the stack and then use the command `return`.

The function-call-and-return protocol: the VM implementation view

When function *f* calls function *g*, the VM implementation must:

- Save the return address within *f* 's code:
  the address of the command just after the `call`
- Save the virtual segments of *f*
- Allocate, and initialize to 0, as many local variables as needed by *g*
- Set the `local` and `argument` segment pointers of *g*
- Transfer control to *g*.

```
function g nVars
call g nArgs
return
```

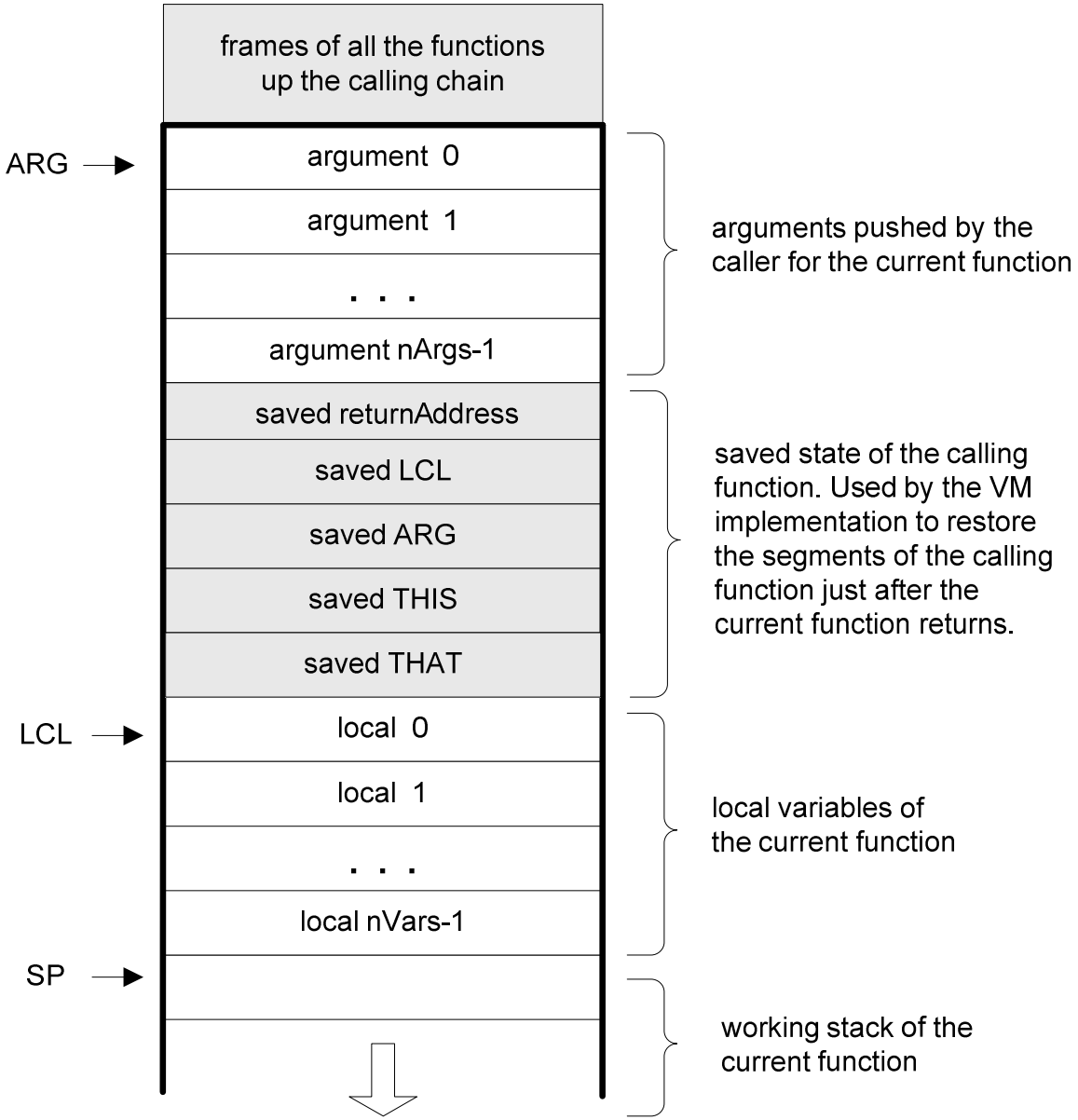
When *g* terminates and control should return to *f*, the VM implementation must:

- Clear *g* 's arguments and other junk from the stack
- Restore the virtual segments of *f*
- Transfer control back to *f*
  (jump to the saved return address).

Q: How should we make all this work "like magic"?

A: We'll use the stack cleverly.

The implementation of the VM's stack on the host Hack RAM



Global stack:
the entire RAM area dedicated to hold the stack

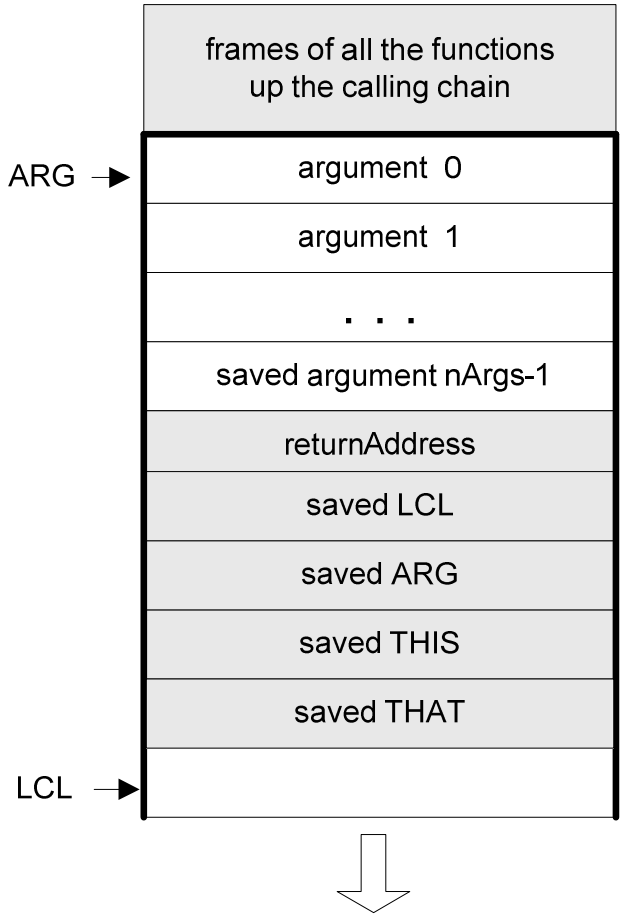
Working stack:
from SP onwards: the stack that the current function sees

- At any point of time, only one function (the *current function*) is executing; other functions may be waiting up the calling chain
- Shaded areas: irrelevant to the current function
- The current function sees only the working stack, as well as its virtual memory segments
- The rest of the stack holds the frozen states of all the functions up the calling hierarchy.

Implementing the `call g nArgs` command

`call g nArgs`

```
  // In the course of implementing the code of f
  // (the caller), we arrive to the command call g nArgs.
  // we assume that nArgs arguments have been pushed
  // onto the stack. What do we do next?
  // We generate a symbol, let's call it returnAddress;
  // Next, we effect the following logic:
  push returnAddress // saves the return address
  push LCL           // saves the LCL of f
  push ARG           // saves the ARG of f
  push THIS          // saves the THIS of f
  push THAT          // saves the THAT of f
  ARG = SP-nArgs-5   // repositions SP for g
  LCL = SP           // repositions LCL for g
  goto g             // transfers control to g
returnAddress:       // the generated symbol
```



None of this code is executed yet ... At this point we are just *generating code* (or simulating the VM code on some platform)

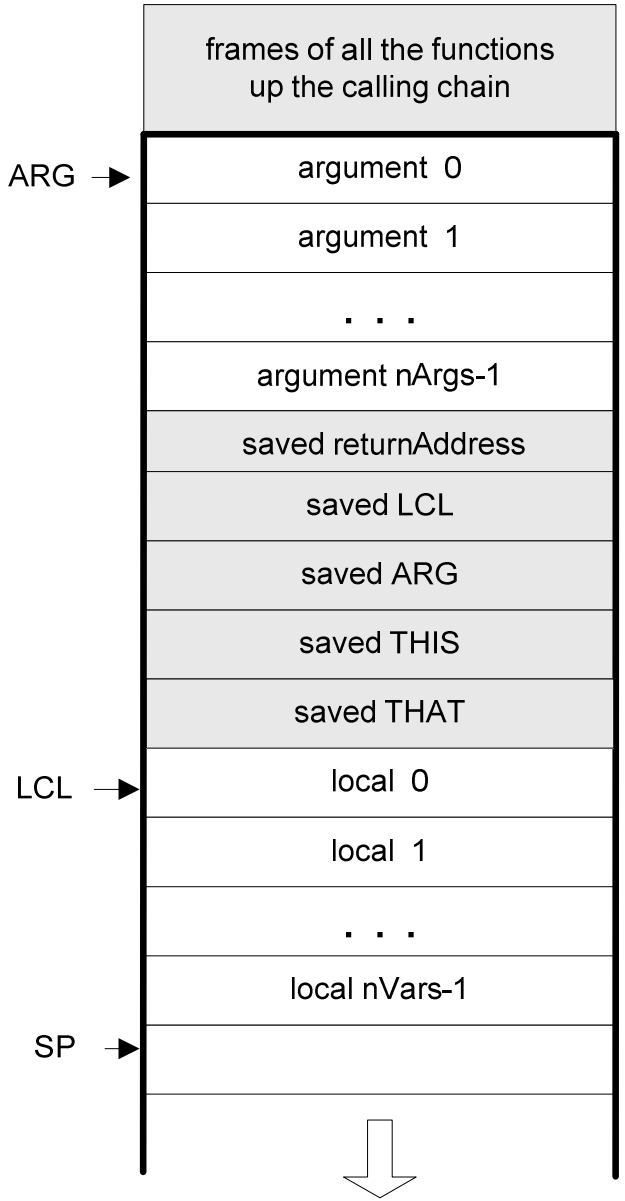
Implementation: If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Implementing the `function g nVars` command

`function g nVars`

```
// to implement the command function g nVars,
// we effect the following logic:

g:
  repeat nVars times:
  push 0
```

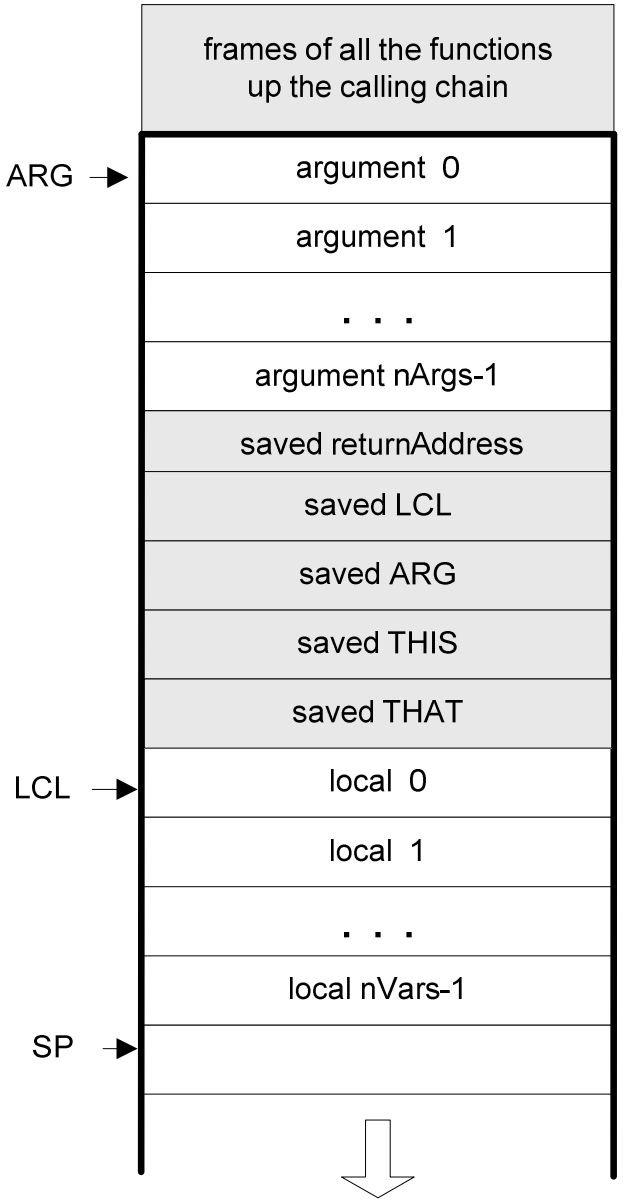


Implementation: If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Implementing the `return` command

return

```
// In the course of implementing the code of g,
// we arrive to the command return.
// We assume that a return value has been pushed
// onto the stack.
// We effect the following logic:
  frame = LCL           // frame is a temp. variable
  retAddr = *(frame-5) // retAddr is a temp. variable
  *ARG = pop            // repositions the return value
                        // for the caller
  SP=ARG+1              // restores the caller's SP
  THAT = *(frame-1)     // restores the caller's THAT
  THIS = *(frame-2)     // restores the caller's THIS
  ARG = *(frame-3)      // restores the caller's ARG
  LCL = *(frame-4)      // restores the caller's LCL
  goto retAddr          // goto returnAddress
```



Implementation: If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Bootstrapping

A high-level jack *program* (aka *application*) is a set of class files.
By a Jack convention, one class must be called `Main`, and this class must have at least one function, called `main`.

The contract: when we tell the computer to execute a Jack program,
the function `Main.main` starts running

Implementation:

- After the program is compiled, each class file is translated into a `.vm` file
- The operating system is also implemented as a set of `.vm` files (aka "libraries") that co-exist alongside the program's `.vm` files
- One of the OS libraries, called `Sys.vm`, includes a method called `init`.
  The `Sys.init` function starts with some OS initialization code (we'll deal with this later, when we discuss the OS), then it does `call Main.main`
- Thus, to bootstrap, the VM implementation has to effect (e.g. in assembly), the following operations:

```
SP = 256        // initialize the stack pointer to 0x0100
call Sys.init   // call the function that calls Main.main
```

VM implementation over the Hack platform

- Extends the VM implementation described in the last lecture (chapter 7)
- The result: a single assembly program file with lots of agreed-upon symbols:

Symbol	*Usage*
`SP, LCL, ARG, THIS, THAT`	These predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments `local`, `argument`, `this`, and `that`.
`R13 - R15`	These predefined symbols can be used for any purpose.
`Xxx.j`	Each static variable `j` in a VM file `Xxx.vm` is translated into the assembly symbol `Xxx.j`. In the subsequent assembly process, these symbolic variables will be allocated RAM space by the Hack assembler.
`functionName$label`	Each `label b` command in a VM function `f` should generate a globally unique symbol "`f$b`" where "`f`" is the function name and "`b`" is the label symbol within the VM function's code. When translating `goto b` and `if-goto b` VM commands into the target language, the full label specification "`f$b`" must be used instead of "`b`".
`(FunctionName)`	Each VM function f should generates a symbol "`f`" that refers to its entry point in the instruction memory of the target computer.
return-address	Each VM function call should generate and insert into the translated code a unique symbol that serves as a return address, namely the memory location (in the target platform's memory) of the command following the function call.