We acknowledge and pay our respects to the Kaurna people, the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

THE UNIVERSITY of ADELAIDE

Computer Systems

Lecture 07: Virtual Machine and The Stack II

Review: The whole system

Compiler

Programming Language

Chapters 10-11

Translator

Virtual Machine

Chapters 7-8

Assembler

Assembly Language

Chapters 6

System Architecture

Machine Language

Chapters 4-5

Gate Logic

Hardware Platform

Chapters 1-3

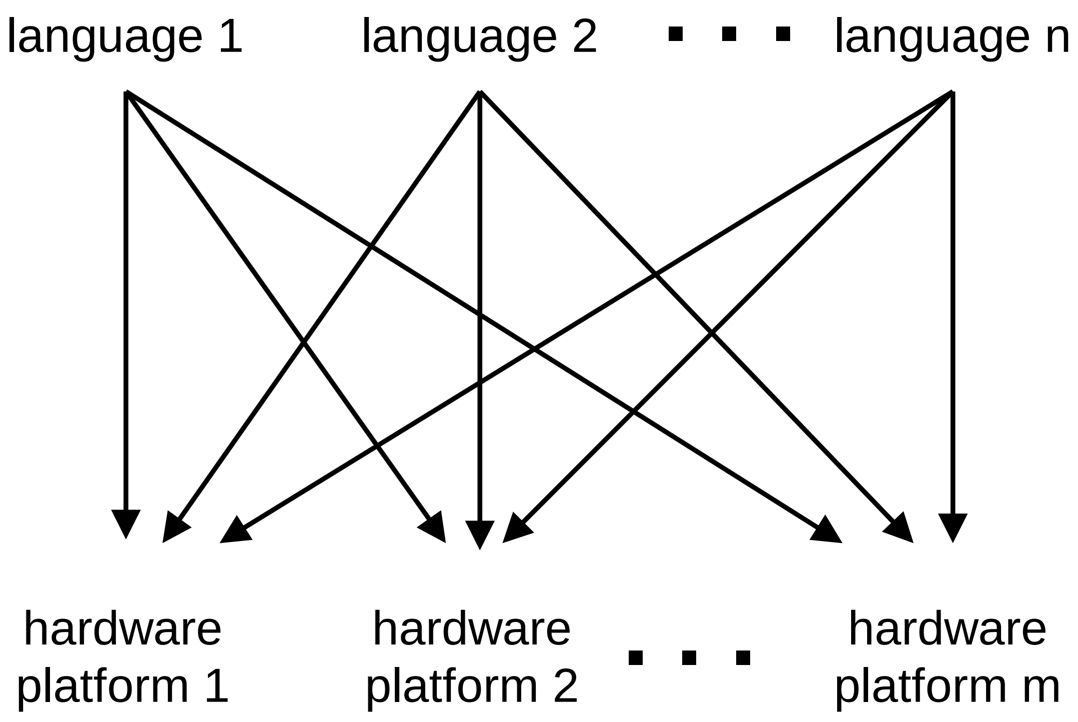
Electronics

Chips and Logic Gates

The Virtual Machine

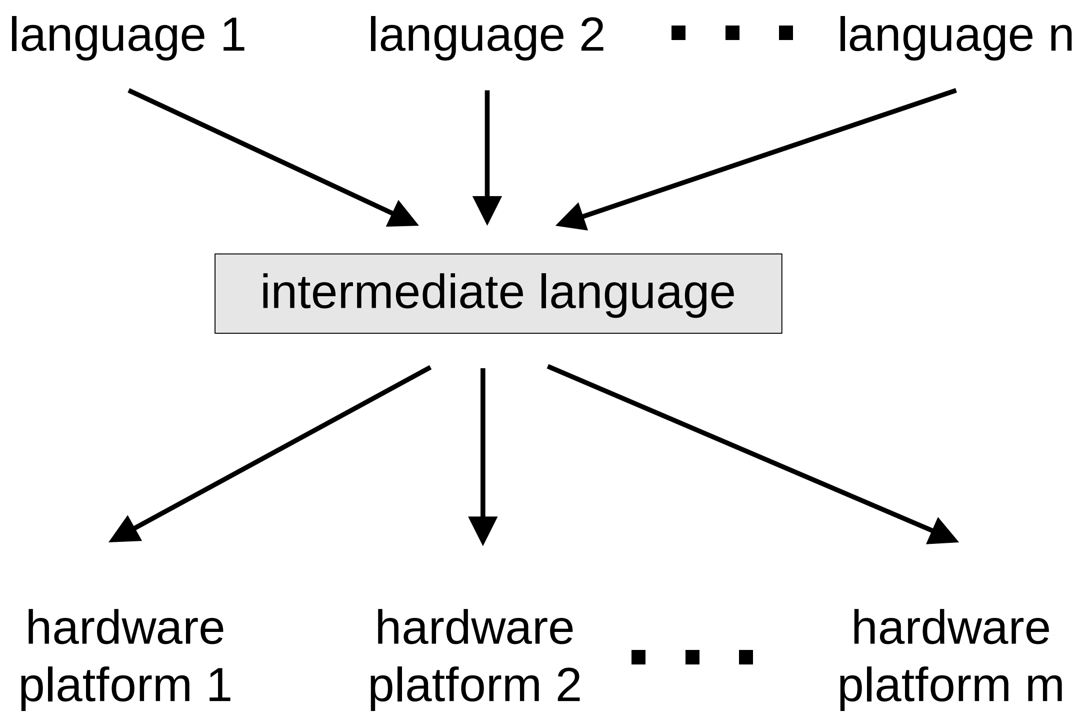
Compilation models

direct compilation:



requires $n . m$ translators

2-tier compilation:



requires $n + m$ translators

Two-tier compilation:

- First compilation stage: depends only on the details of the source language
- Second compilation stage: depends only on the details of the target language.

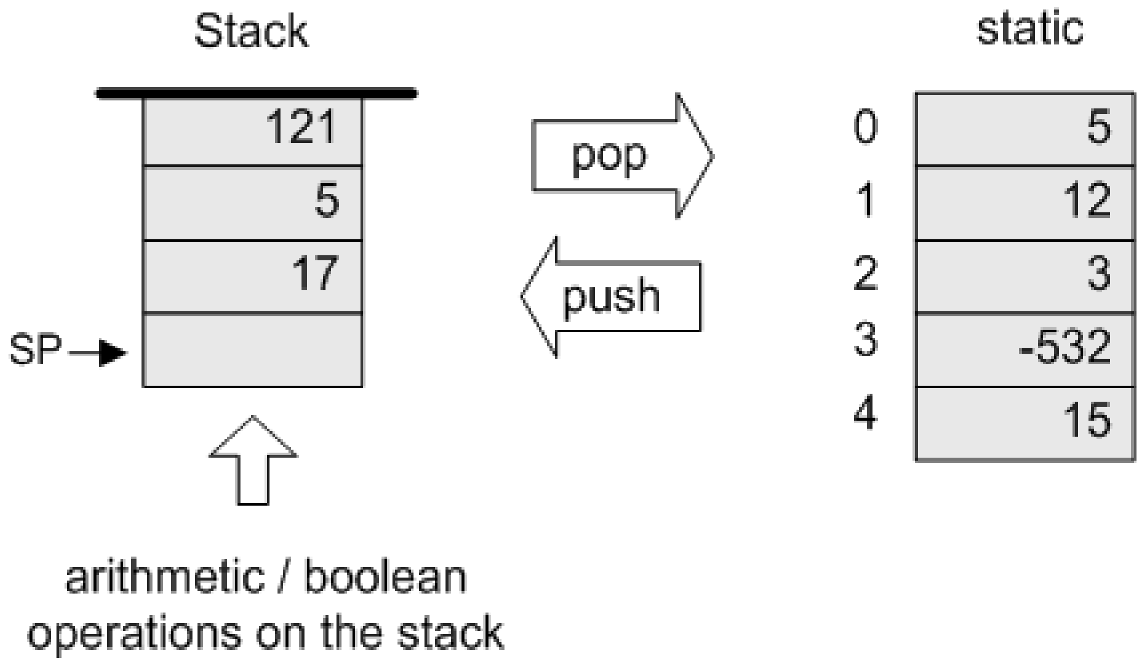
Our VM model is *stack-oriented*

All operations are done on a stack

Data is saved in several separate *memory segments*

All the memory segments behave the same

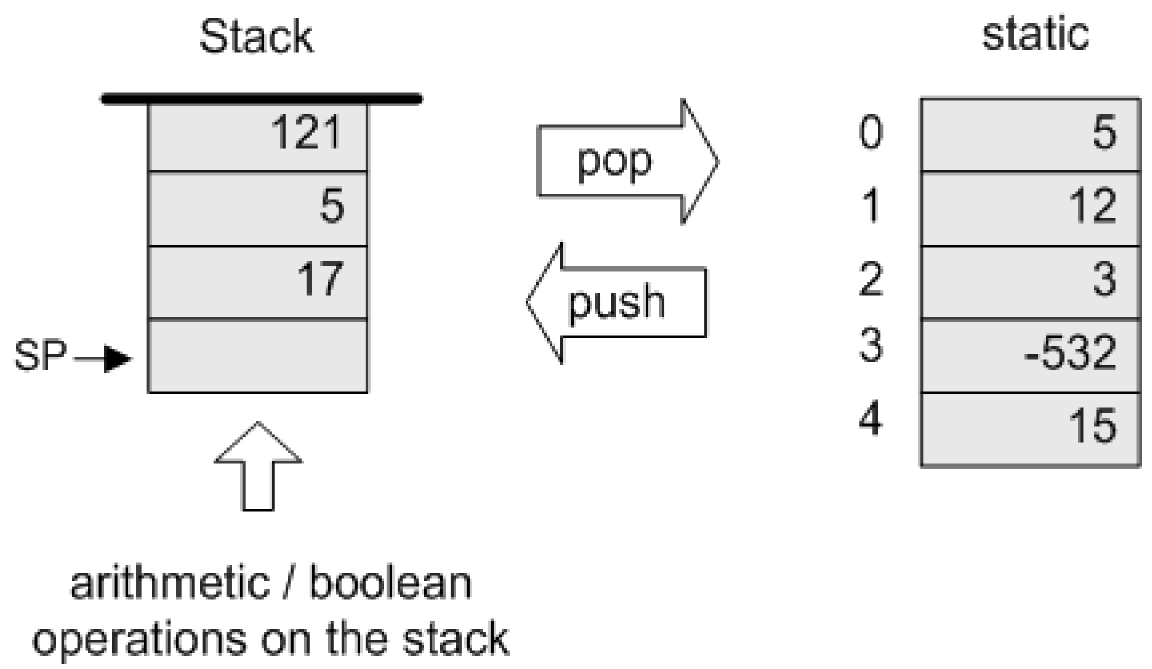
One of the memory segments m is called `static`, and we will use it (as an arbitrary example) in the following examples:



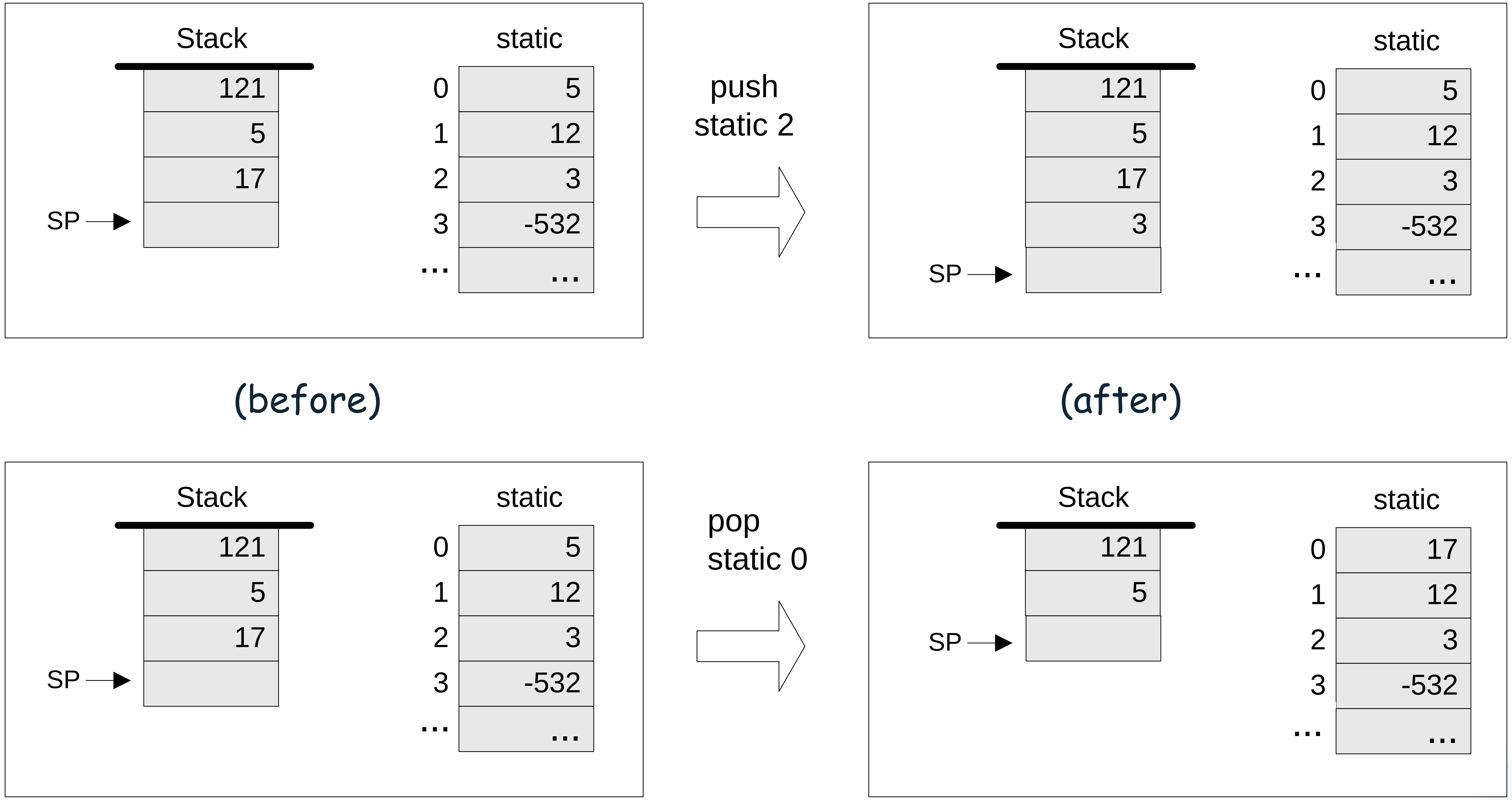
Data types

Our VM model features a single 16-bit data type that can be used as:

- an integer value (16-bit 2’s complement: -32768, ... , 32767)
- a Boolean value (-1 and 0, standing for true and false)
- a pointer (memory address)



Memory access operations



The stack:

- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.

THE UNIVERSITY of ADELAIDE

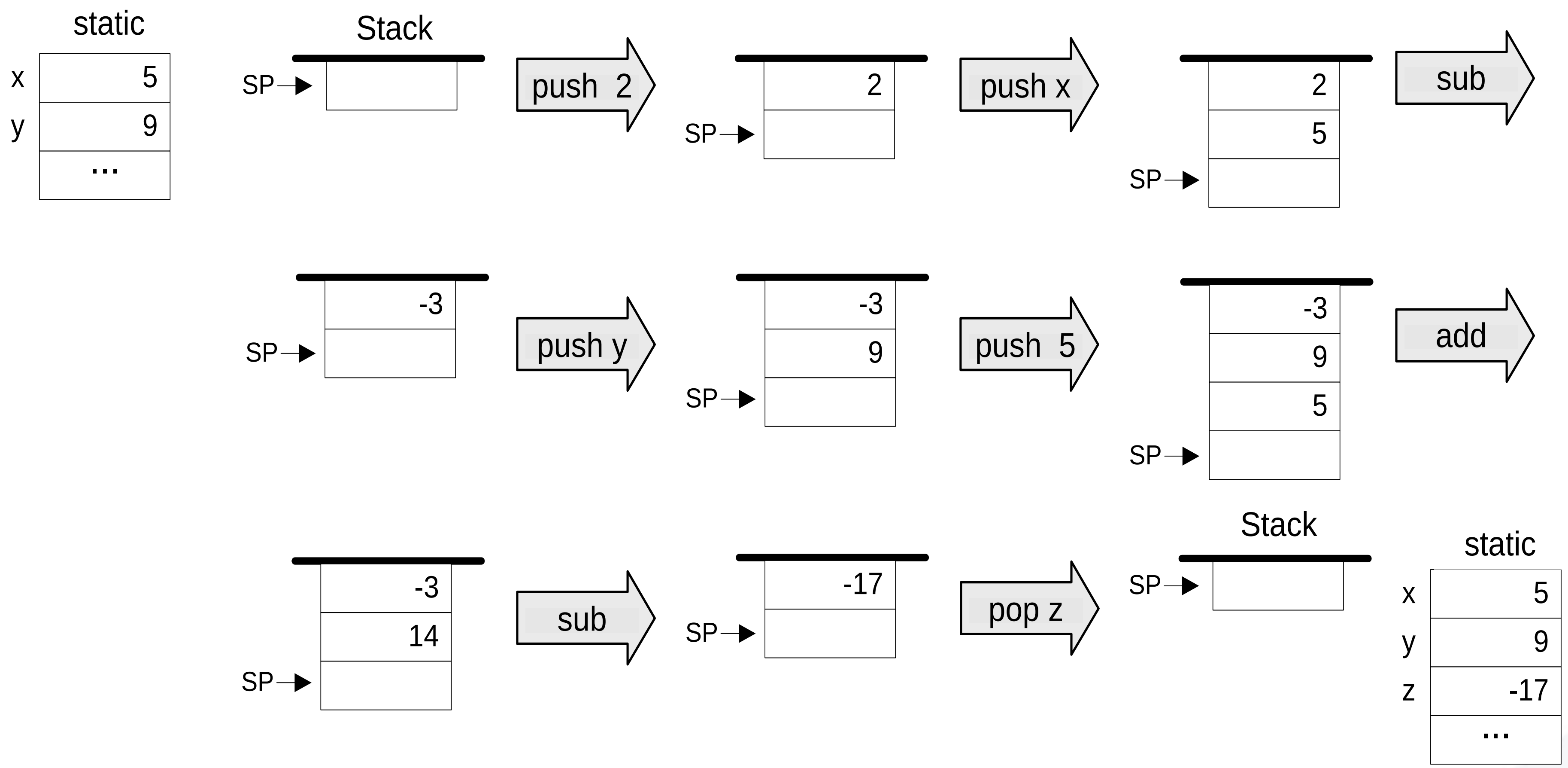
Expressions

Evaluation of arithmetic expressions

VM code (example)

```
// z=(2-x)-(y+5)
push 2
push x
sub
push y
push 5
add
sub
pop z
```

(suppose that
x refers to static 0,
y refers to static 1, and
z refers to static 2)

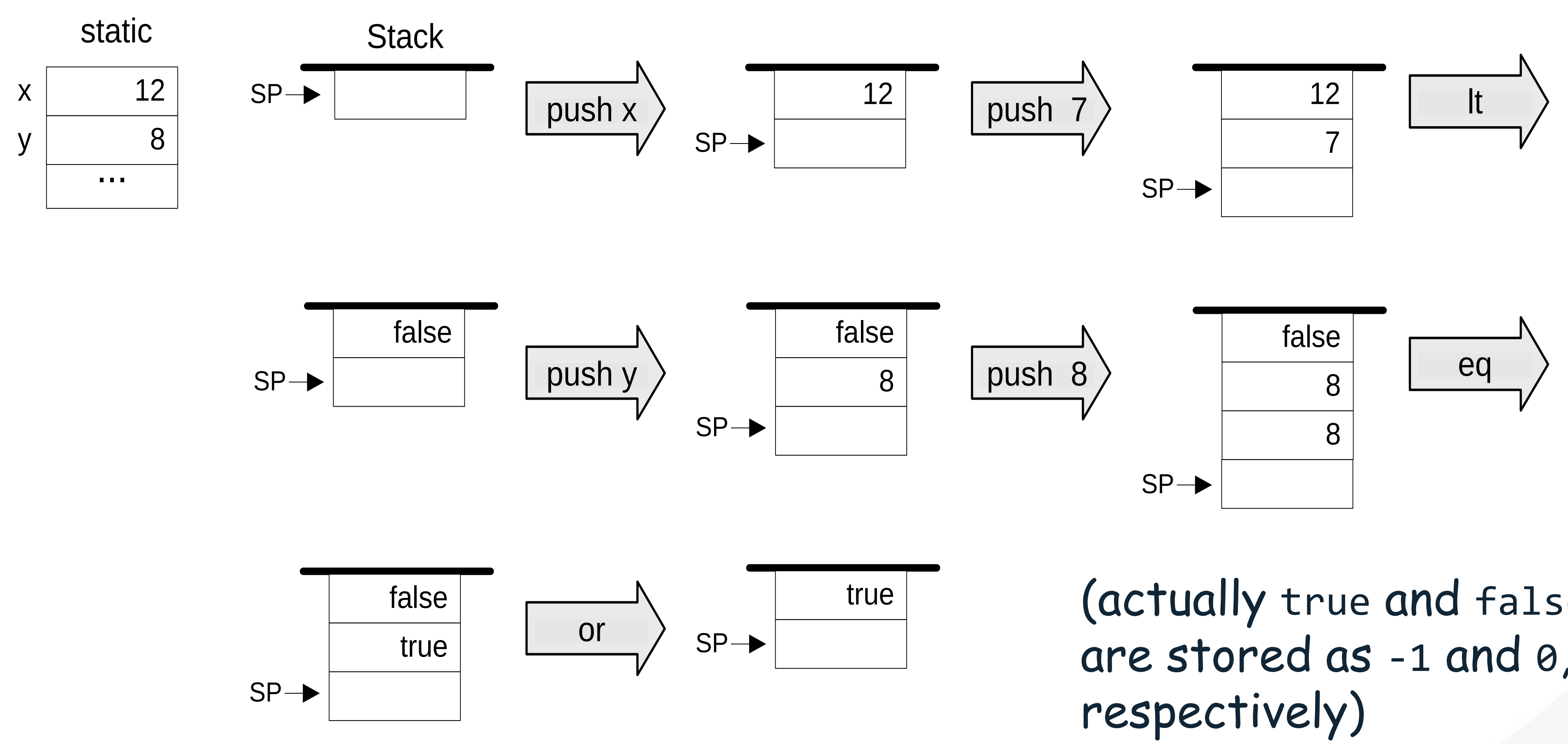


Evaluation of Boolean expressions

VM code (example)

```
// (x<7) or (y=8)
push x
push 7
lt
push y
push 8
eq
or
```

(suppose that
`x` refers to `static 0`, and
`y` refers to `static 1`)

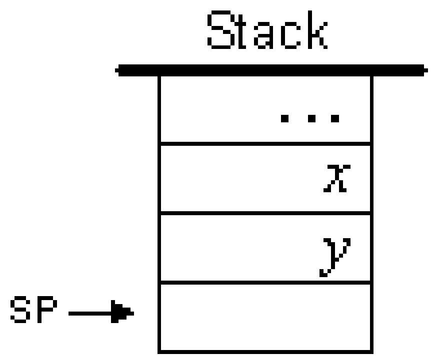


(actually `true` and `false` are stored as `-1` and `0`, respectively)

Arithmetic and Boolean commands in the VM language

(wrap-up)

Command	**Return value** (after popping the operand/s)	Comment
`add`	$x + y$	Integer addition (2's complement)
`sub`	$x - y$	Integer subtraction (2's complement)
`neg`	$-y$	Arithmetic negation (2's complement)
`eq`	true if $x = y$ and false otherwise	Equality
`gt`	true if $x > y$ and false otherwise	Greater than
`lt`	true if $x < y$ and false otherwise	Less than
`and`	x And y	Bit-wise
`or`	x Or y	Bit-wise
`not`	Not y	Bit-wise



Expressions to Hack VM Language

`not (a or b)`

```
push static 0
push static 1
or
not
```

`d + c + b + a`

```
push static 3
push static 2
add
push static 1
add
push static 0
add
```

THE UNIVERSITY of ADELAIDE

Expressions to Hack VM Language

(4 + a) * (c - 9)

```
push constant 4
push static 0
add
push static 1
push constant 9
sub
call Math.multiply 2
```

true and false

```
push constant 0
not
push constant 0
and
```

THE UNIVERSITY
of ADELAIDE

Virtual Machine Memory Structure

THE UNIVERSITY of ADELAIDE

The VM's Memory segments

A VM program is designed to provide an interim abstraction of a program written in some high-level language

Modern OO high-level languages normally feature the following variable kinds:

Class level:

- Static variables (class-level variables)
- Private variables (aka "object variables" / "fields" / "properties")

Method level:

- Local variables
- Argument variables

When translated into the VM language,

The static, private, local and argument variables are mapped by the compiler on the four memory segments
`static`, `this`, `local`, `argument`

In addition, there are four additional memory segments, whose role will be presented later:
`that`, `constant`, `pointer`, `temp`.

Memory segments and access commands

The VM abstraction includes 8 separate memory segments named:
static, this, local, argument, that, constant, pointer, temp

As far as VM programming commands go, all memory segments look and behave the same

To access a particular segment entry, use the following generic syntax:

<u>Memory access VM commands</u>:

- pop *memorySegment* *index*
- push *memorySegment* *index*

Where *memorySegment* is `static`, `this`, `local`, `argument`, `that`, `constant`, `pointer`, or `temp`

And *index* is a non-negative integer

<u>Notes:</u>

(In all our code examples thus far, *memorySegment* was `static`)

The roles of the eight memory segments will become relevant when we talk about compiling

At the VM abstraction level, all memory segments are treated the same way.

VM programming

VM programs are normally written by *compilers*, not by humans

However, compilers are written by humans ...

In order to write or optimize a compiler, it helps to first understand the spirit of the compiler's target language – the VM language

The example VM program includes four new VM commands:

- `function` *functionSymbol int* // function declaration
- `label` *labelSymbol* // label declaration
- `goto` *labelSymbol* // jump to execute the command after *labelSymbol*
- `if-goto` *labelSymbol* // pop x
  // if x=true, jump to execute the command after *labelSymbol*
  // else proceed to execute the next command in the program

For example, to effect `if (x > n) goto loop`, we can use the following VM commands:

```
push x
push n
gt
if-goto loop          // Note that x, n, and the truth value were removed from the stack.
```

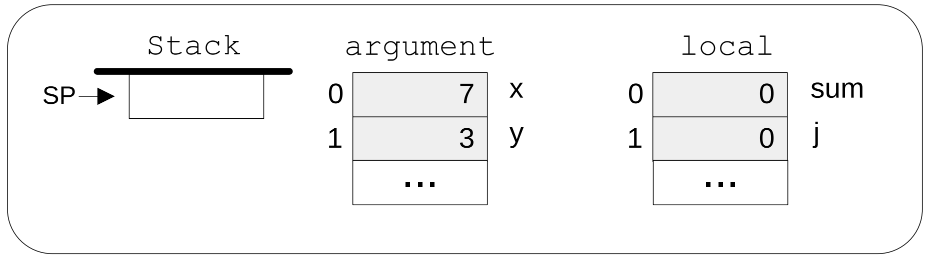
THE UNIVERSITY *of* ADELAIDE

VM programming (example)

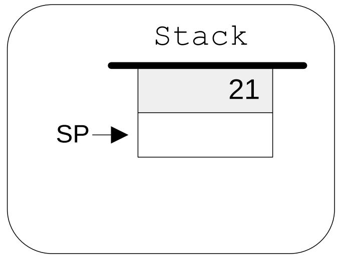
High-level code

```
function int mult(x,y)
{
  var int result, j;
  let result = 0;
  let j = y;
  while ~(j = 0)
  {
    let result = result + x;
    let j = j - 1;
  }
  return result;
}
```

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



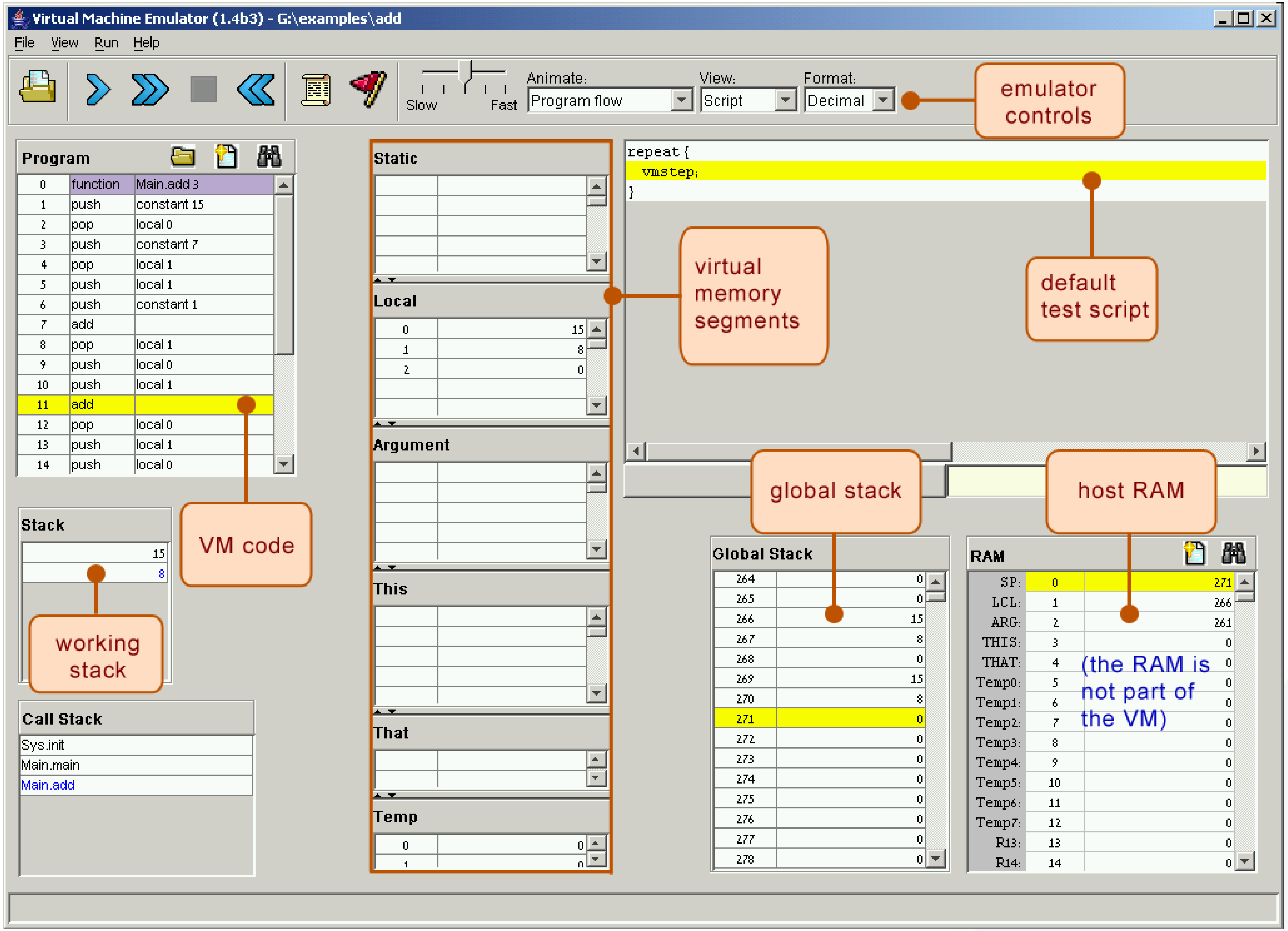
VM code (first approx.)

```
function mult(x,y)
   push 0
   pop result
   push y
   pop j
label loop
   push j
   push 0
   eq
   if-goto end
   push result
   push x
   add
   pop result
   push j
   push 1
   sub
   pop j
   goto loop
label end
   push result
   return
```

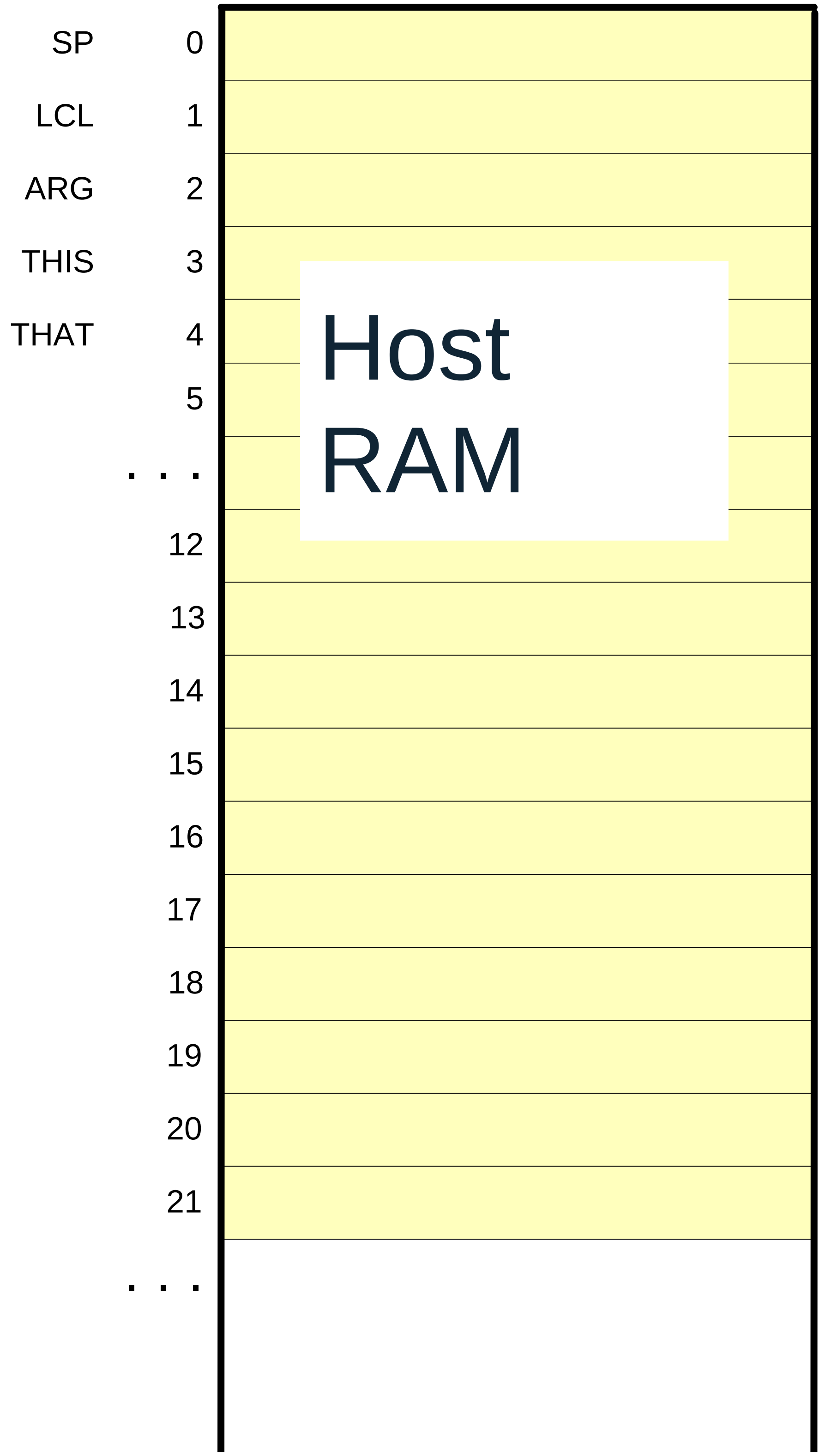
VM code

```
function mult 2
  push   constant 0
  pop    local 0
  push   argument 1
  pop    local 1
label    loop
  push   local 1
  push   constant 0
  eq
  if-goto end
  push   local 0
  push   argument 0
  add
  pop    local 0
  push   local 1
  push   constant 1
  sub
  pop    local 1
  goto   loop
label    end
  push   local 0
  return
```

Software implementation: Hack VM emulator



VM implementation on the Hack platform



<u>The stack:</u> a global data structure, used to save and restore the resources of all the VM functions up the calling hierarchy.

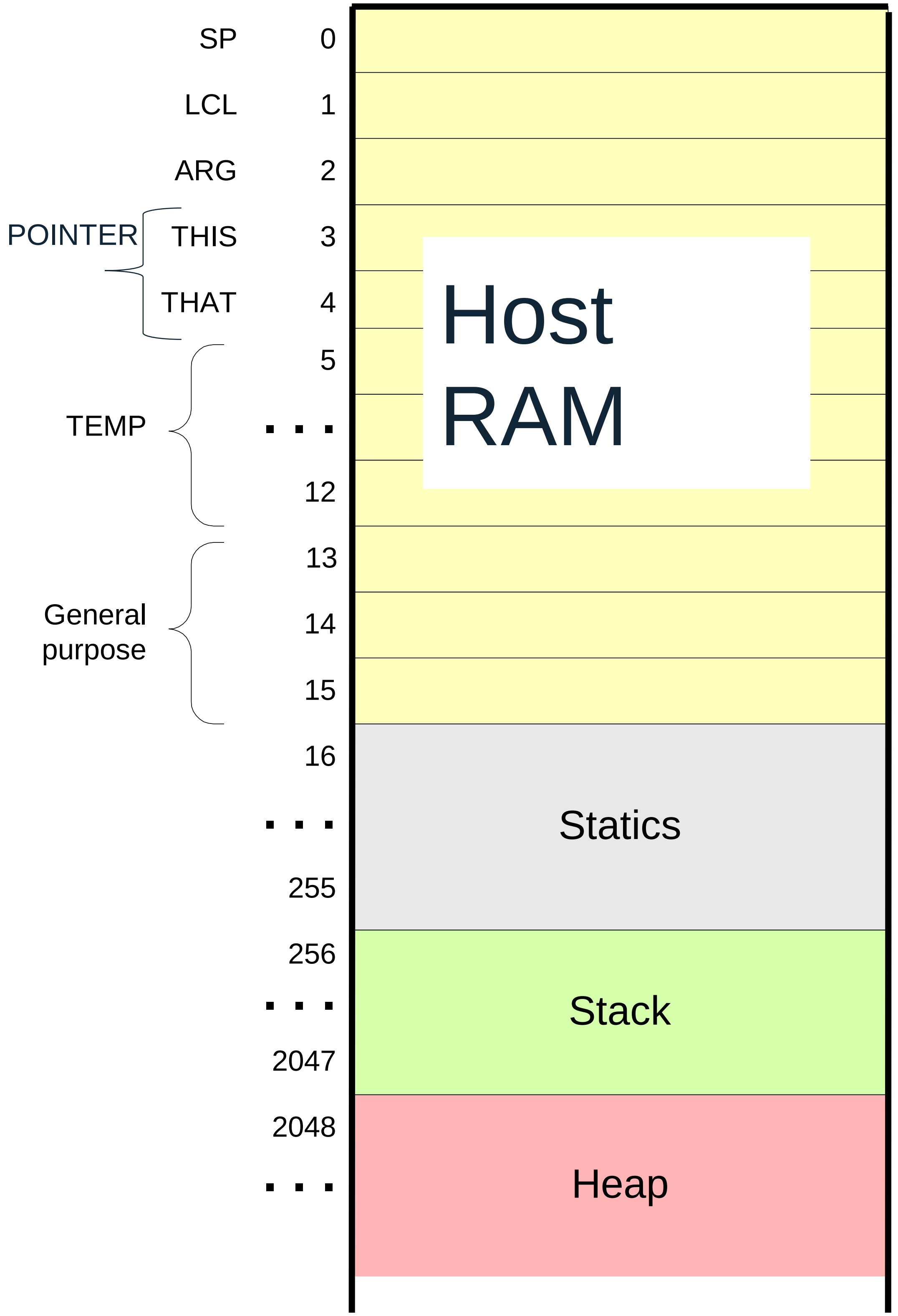
The tip of this stack is the working stack of the current function

<u>static, constant, temp, pointer</u>:
Global memory segments, all functions see the same four segments

<u>local,argument,this,that</u>:
these segments are local at the function level; each function sees its own, private copy of each one of these four segments

<u>The challenge:</u>
represent all these logical constructs on the same single physical address space -- the host RAM.

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

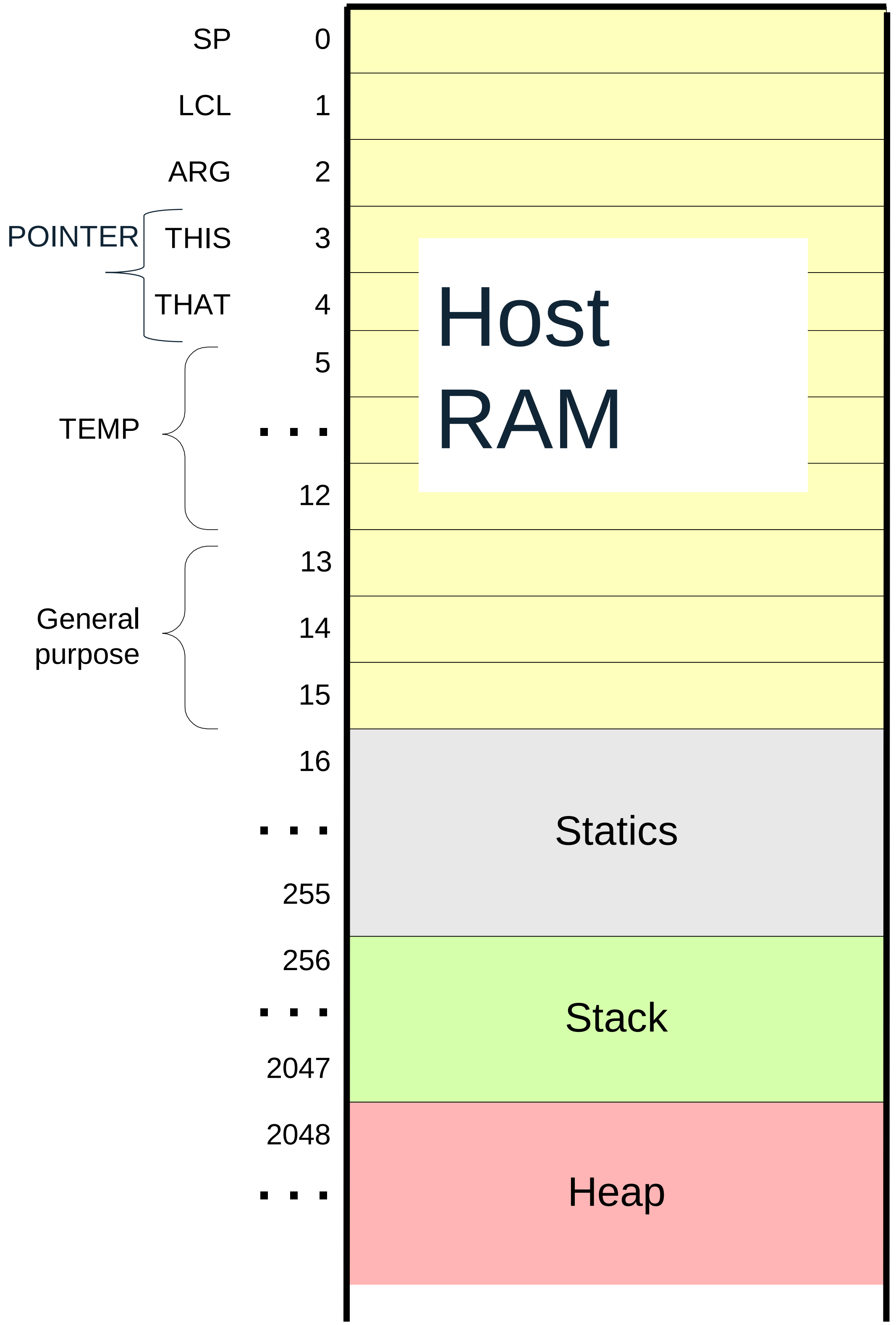
static: mapped on RAM[16 ... 255];
each segment reference static *i* appearing in a VM file named f is compiled to the assembly language symbol f.i (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

local,argument,this,that: these method-level segments are mapped somewhere from address 256 onward, on the "stack" or the "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 virtual segment:
access to constant *i* is implemented by supplying the constant *i.*

pointer: RAM[3..4] to change THIS and THAT.

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.

THE UNIVERSITY of ADELAIDE

VM Translator Parsing

- Memory locations R13, R14, R15 can be used as temporary variables if required
- `push constant 1`

```
@SP
AM=M+1
A=A-1
M=1
```

- `pop static 7 (in a VM file named Bob.vm)`

```
@SP
AM=M-1
D=M
@Bob.7
M=D
```

VM Translator Parsing

- `push constant 5`

```
@5
D=A
@SP
AM=M+1
A=A-1
M=D
```

- `add`

```
@SP
AM=M-1
D=M
A=A-1
M=D+M
```

Program Flow Control

Branches & Loops

Program flow commands in the VM language

In the VM language, the program flow abstraction is delivered using three commands:

```
label c       // label declaration

goto c        // unconditional jump to the
              // VM command following the label c

if-goto c     // pops the topmost stack element;
              // if it's not zero, jumps to the
              // VM command following the label c
```

VM code example:

```
function mult 1
  push constant 0
  pop local 0
label loop
  push argument 0
  push constant 0
  eq
  if-goto end
  push argument 0
  push constant 1
  sub
  pop argument 0
  push argument 1
  push local 0
  add
  pop local 0
  goto loop
label end
  push local 0
  return
```

How to translate these three abstractions into assembly?

- Simple: label declarations and goto directives can be effected directly by assembly commands
- More to the point: given any one of these three VM commands, the VM Translator must emit one or more assembly commands that effects the same semantics on the Hack platform
- How to do it? see project 8.

Program Flow Translation

Translating a loop into Hack Virtual Machine code

```
// assume i and sum have been initialised to 0
while ( i < 10 ) {
    let sum = sum + i;
    let i = i + 1;
}
```

Virtual Machine code

```
label LOOP
push local 0
push constant 10
lt
not
if-goto LOOPEND
push local 1
push local 0
add
pop local 1
push local 0
push constant 1
add
pop local 0
goto LOOP
label LOOPEND
```

Program Flow Translation

Translating `if-goto labelX` into Hack Assembler

assume it is inside a function named Example.func

```
@SP
AM=M-1
D=M
@Example.func$labelX
D;JNE
```

THE UNIVERSITY *of* ADELAIDE

Program Flow Control

Functions

Subroutines

```
// Compute x = (-b + sqrt(b^2 -4*a*c)) / 2*a
if (~(a = 0))
{
   let x = (- b + Math.sqrt(b * b – 4 * a * c)) / (2 * a);
} else
{
   let x = - c / b ;
}
```

<u>Subroutines = a major programming artifact</u>

- Basic idea: the given language can be extended at will by user-defined commands (aka *subroutines* / *functions* / *methods* ...)

- Important: the language's primitive commands and the user-defined commands have the same look-and-feel

- This transparent extensibility is the most important abstraction delivered by high-level programming languages

- The challenge: implement this abstraction, i.e. allow the program control to flow effortlessly from one subroutine to another

"A well-designed system consists of a collection of black box modules, each executing its effect like magic" (Steven Pinker, *How The Mind Works*)

Subroutines in the VM language

Calling code (example)

```
...
// computes (7 + 2) * 3 - 5
push constant 7
push constant 2
add
push constant 3
call mult 2
push constant 5
sub
...
```

VM subroutine call-and-return commands

Called code, aka "callee" (example)

```
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  push argument 0
  push constant 0
  eq
  if-goto end // if arg0 == 0, jump to end
  push argument 0
  push 1
  sub
  pop argument 0  // arg0--
  push argument 1
  push local 0
  add
  pop local 0  // result += arg1
  goto loop
label end
  push local 0  // push result
  return
```

The invocation of the VM's primitive commands and subroutines follow exactly the same rules:

- The caller pushes the necessary argument(s) and calls the command / function for its effect
- The called command / function is responsible for removing the argument(s) from the stack, and for pushing onto the stack the result of its execution.

Function commands in the VM language

```
function g nVars    // here starts a function called g,
                    // which has nVars local variables

call g nArgs        // invoke function g for its effect;
                    // nArgs arguments have already been pushed onto the stack

return              // terminate execution and return control to the caller
```

Q: Why this particular syntax?

A: Because it simplifies the VM implementation (later).

Program Flow Translation

Translating a recursive function into Hack VM code.

```
function int recfib(int x)
{
    var int fib ;

    let fib = 1 ;
    if ( x > 1 )
    {
           let fib = recfib(x - 1) + recfib(x - 2) ;
    }

    return fib ;
}
```

Program Flow Translation

Translating a recursive function into Hack VM code.

Function:

```
function X.recfib 1

push constant 1
pop local 0

push argument 0
push constant 1
gt
not
if-goto IFEND
```

```
push argument 0
push constant 1
sub
call X.recfib 1
push argument 0
push constant 2
sub
call X.recfib 1
add
pop local 0

IFEND
push local 0
return
```

Function call-and-return conventions

Calling function

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

called function aka "callee" (example)

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

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:
  - Any 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.

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

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 but is empty
- 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`.

Stack Simulation

Assume that there is a function Really.useful that takes two integer parameters and but does not return a result

Write the Hack Virtual Machine code to call Really.useful and pass it the parameters 7 and 12.

```
push constant 7
push constant 12
call Really.useful 2
pop temp 0
```

The function-call-and-return: VM Challenges

- Every time a function is called, it has its own arguments
- Every time a function is called, it has its own local variables with their own scope
  - How do we separate these from other scopes?
- Functions can be nested/recursive
  - How do we make sure each call's arguments/variables are separate?
- Functions can be called from and return to different locations
  - How to we track where to resume to after a function returns?

THE UNIVERSITY of ADELAIDE

The function-call-and-return: VM View

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

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

- Save the return address within *f* 's code:
the address of the command just after the call
- Save the virtual segments of *f*
- Set the local and argument segment pointers of *g*
- Transfer control to *g*

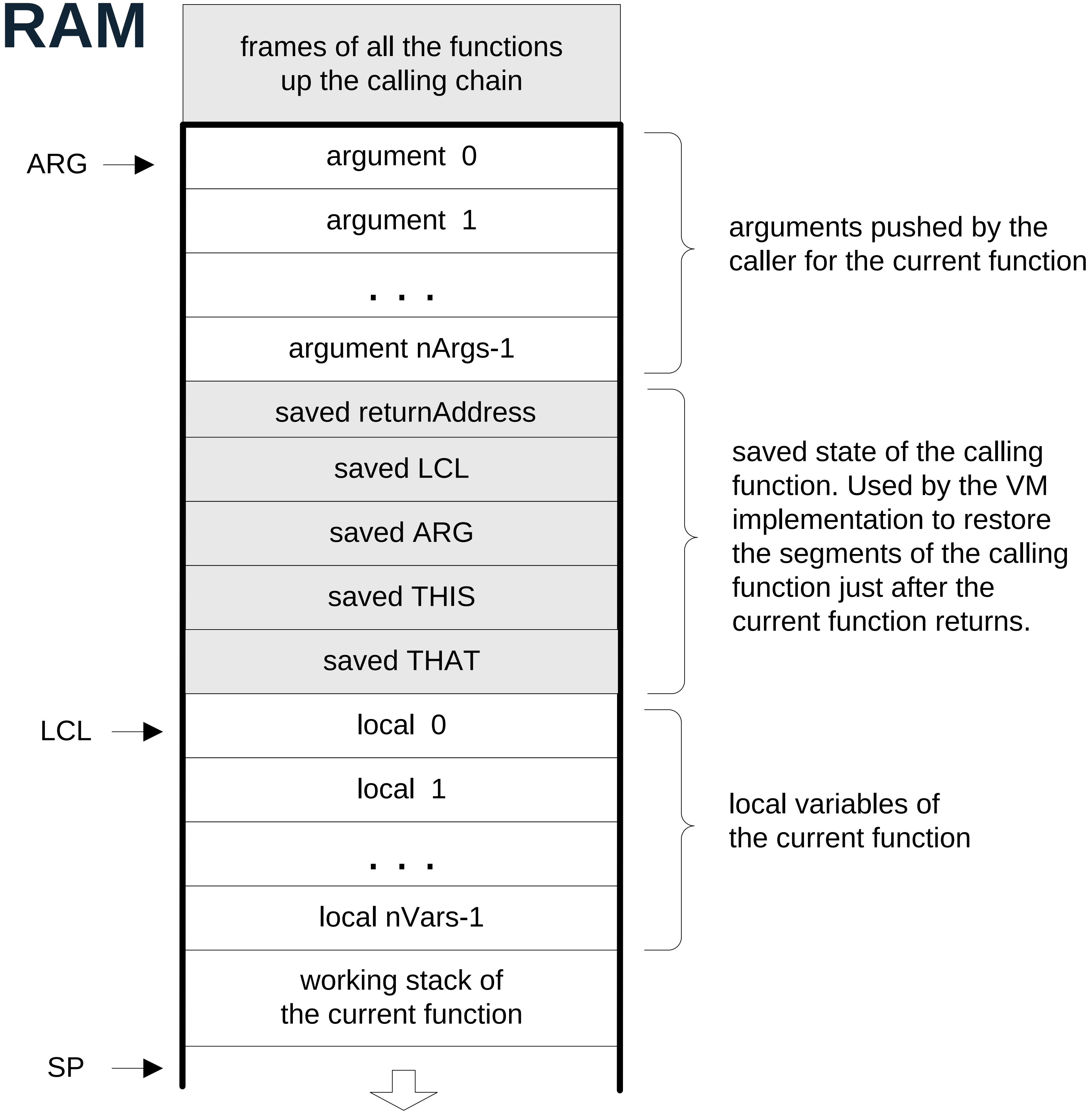
When control is transferred to *g*, the VM implementation must:

- Allocate, and initialize to *0*, as many local variables as needed by *g*

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

- Replace *g* 's arguments and other data on the stack with *g* 's result
- Restore the virtual segments of *f*
- Transfer control back to *f* (jump to the saved return address).

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



<u>Global stack:</u>

the entire RAM area dedicated for holding the stack

<u>Working stack:</u>

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

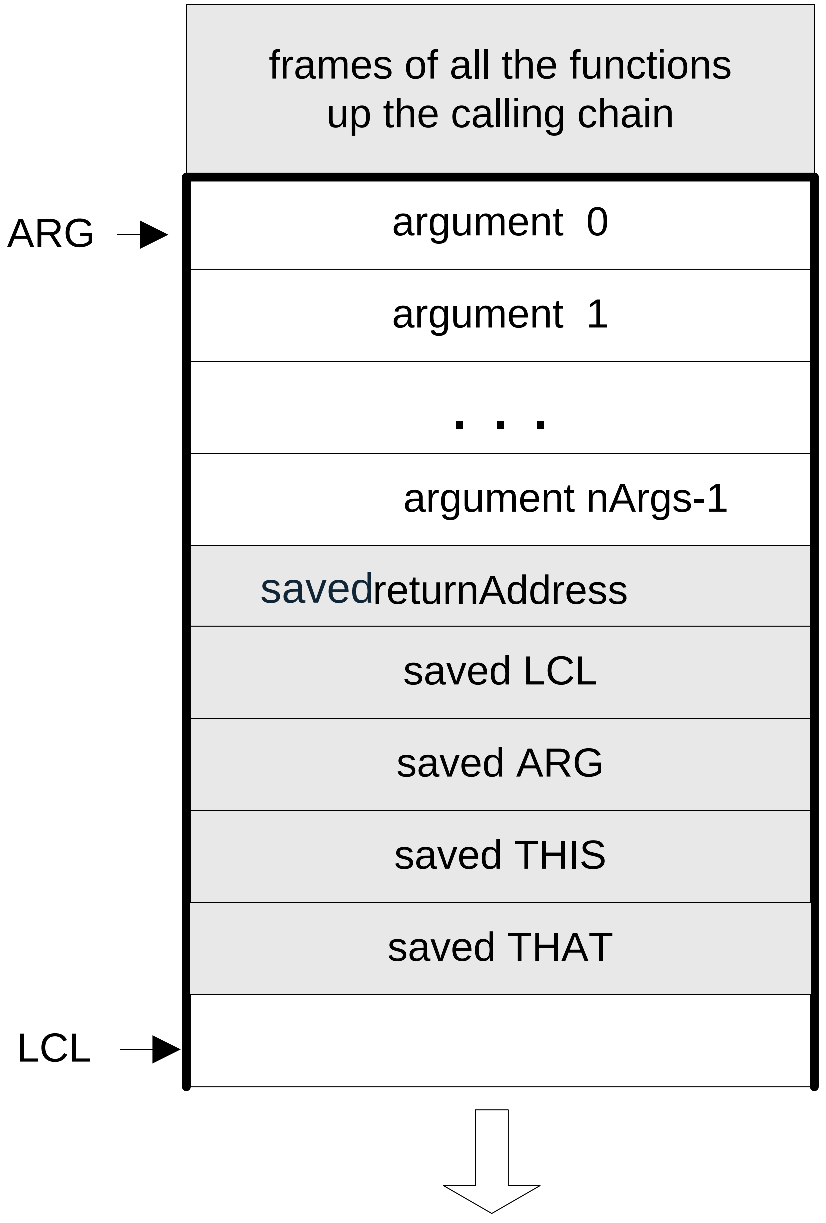
The current function sees only the working stack, and has access only to its memory segments

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 at the command call g nArgs.
  // we assume that nArgs arguments have been pushed
  // onto the stack. What do we do next?
  // We generate a unique label, for example retAddr01;
  // Next, we effect the following logic:
  push retAddr01     // 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 ARG for g
  LCL = SP           // repositions LCL for g
  goto g             // transfers control to g
(retAddr01)          // the generated label
```



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

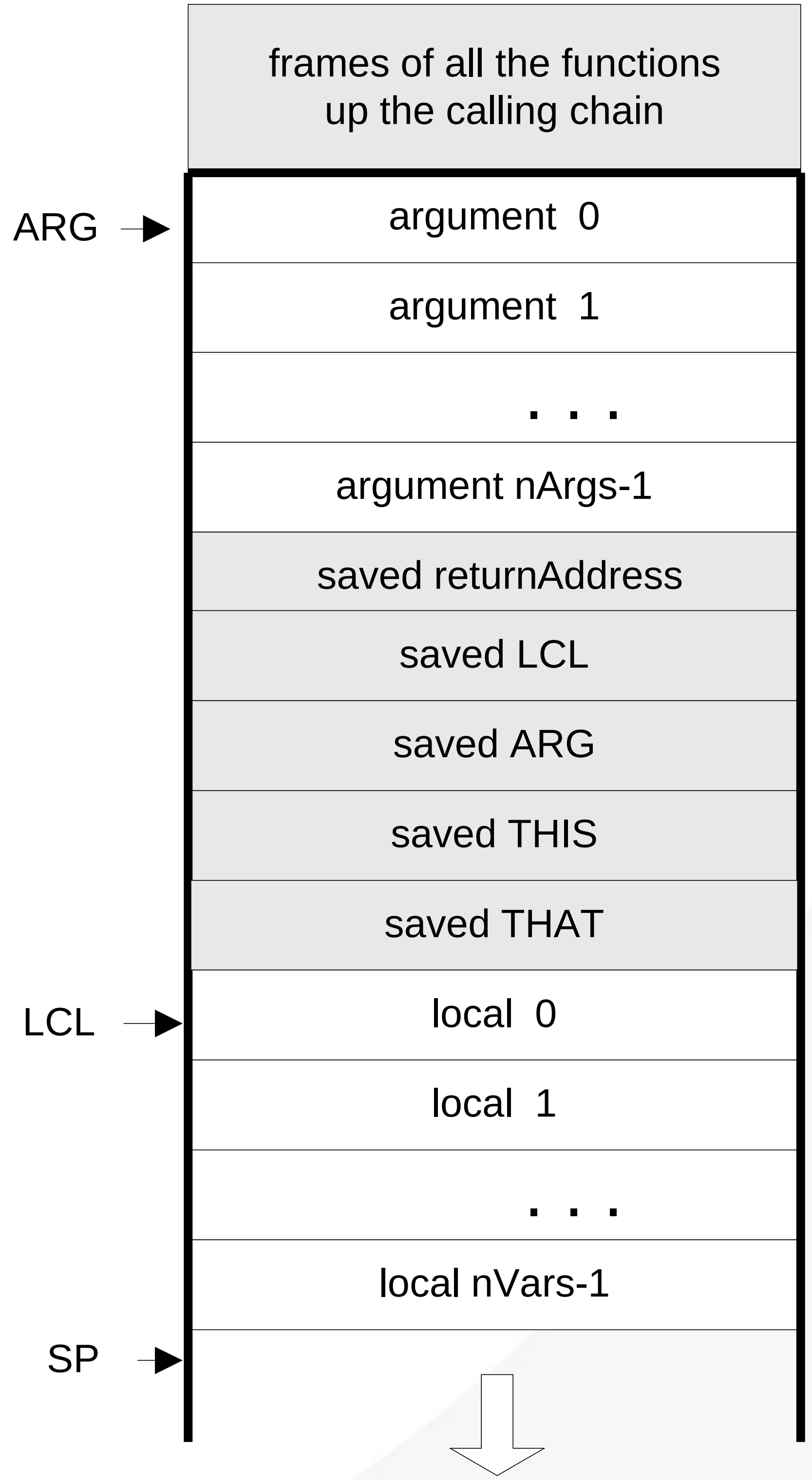
__Implementation:__ **If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.**

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 constant 0
```

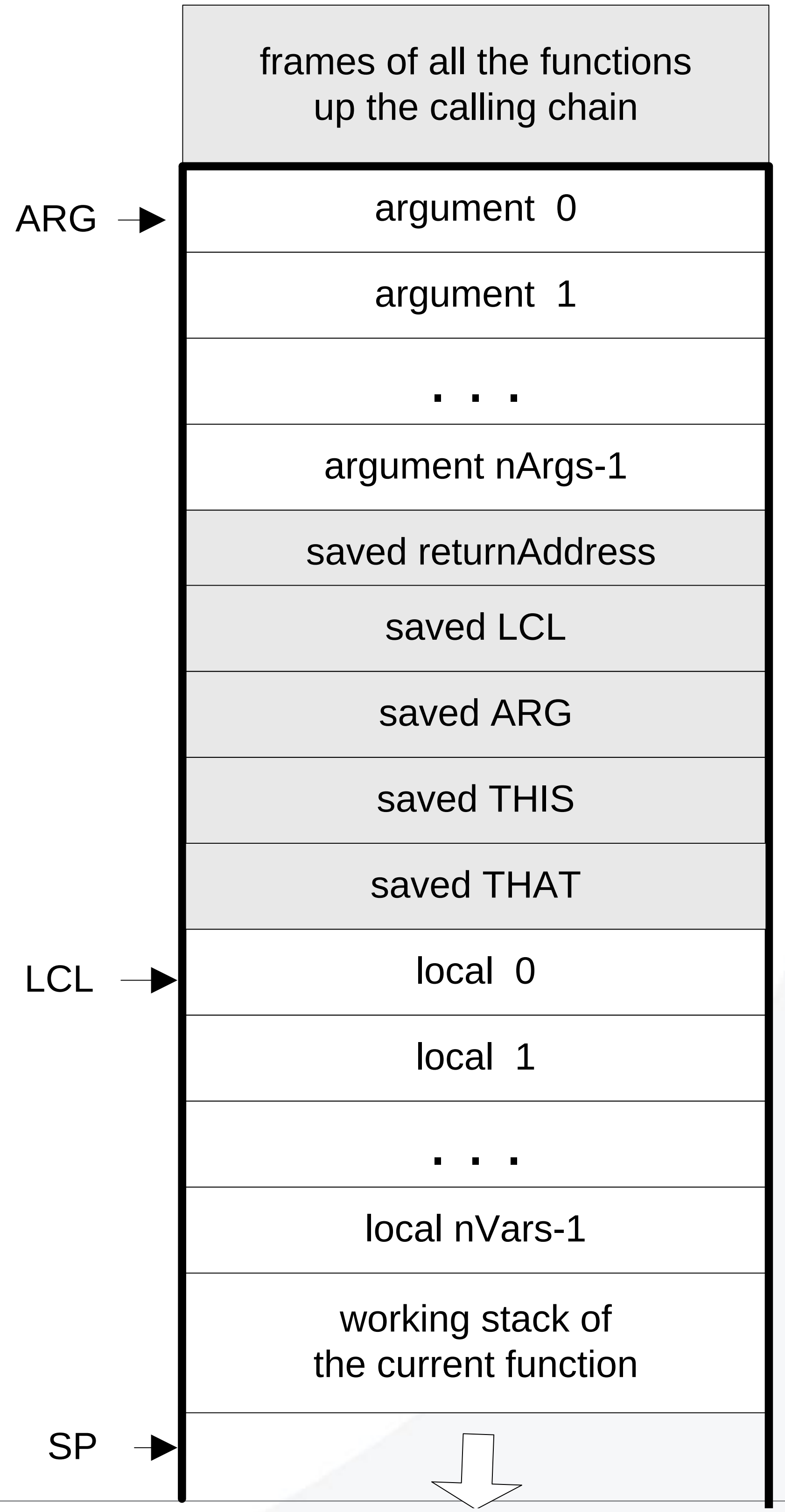


<u>Implementation:</u> If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Implementing the `return` command

```
return
```

```
// In the course of implementing the code of g,
// we arrive at 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 saved returnAddress
```



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

THE UNIVERSITY
of ADELAIDE