EP1200 Introduction to Computing Systems Engineering

Virtual Machine II

The compilation challenge

Source code (high-level language)

```
class Main {
  static int x;

  function void main() {
    // Inputs three numbers
    var int a, b, c;
    let a = Keyboard.readInt("Enter a number");
    let b = Keyboard.readInt("Enter a number");
    let c = Keyboard.readInt("Enter a number");
    let x = solve(a,b,c);
    return;
  }
}

  // Solves a quadratic equation (sort of)
  function int solve(int a, int b, int c) {
     var int x;
     if (~(a = 0))
        x=(-b+sqrt(b*b-4*a*c))/(2*a);
     else
        x=-c/b;
     return x;
  }
}
```

Our ultimate goal:

Translate high-level programs into executable code.

Compiler

Target code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
```

The compilation challenge / two-tier setting

Jack source code

```
if (~(a = 0))
   x = (-b+sqrt(b*b–4*a*c))/(2*a)
else
   x = -c/b
```

Compiler

VM (pseudo) code

```
   push a
   push 0
   eq
   if-goto elseLabel
   push b
   neg
   push b
   push b
   call mult
   push 4
   push a
   call mult
   push c
   call mult
   sub
   call sqrt
   add
   push 2
   push a
   call mult
   call div
   pop x
   goto contLable
elseLabel:
   push c
   neg
   push b
   call div
   pop x
contLable:
```

VM translator

Machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010010
1110001100000001
...
```

- We'll develop the Jack-VM compiler later
- Focus now:
  - complete the definition of the VM language
  - translate each VM command into assembly commands that perform the desired semantics

The compilation challenge

Typical compiler's source code input:

```
// Computes x = (-b + sqrt(b^2 -4*a*c)) / 2*a

if (~(a = 0))
   x = (-b + sqrt(b * b – 4 * a * c)) / (2 * a)
else
   x = - c / b
```

program flow logic (branching)	Boolean expressions	function call and return logic	arithmetic expressions
(this lecture)	(previous lecture)	**(this lecture)**	(previous lecture)

<u>How to translate such high-level code into assembly?</u>

- In a two-tier compilation model, the overall translation challenge is broken between a *front-end* compilation stage and a subsequent *back-end* translation stage
- In our Hack-Jack platform, all the above sub-tasks (handling arithmetic / Boolean expressions and program flow / function calling commands) are done by the back-end, i.e. by the VM translator.

Program flow commands in the VM language

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 1
  sub
  pop argument 0
  push argument 1
  push local 0
  add
  pop local 0
  goto loop
label end
  push local 0
  return
```

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
```

How to translate these three abstractions into assembly?

- Label declaration
  - Can be translated directly to assembly commands
- Goto and Conditional goto commands
  - Combination of assembly commands that effect the same semantics (change to stack and program execution)

Subroutines (Functions or Methods)

```
// Compute x = (-b + sqrt(b^2 -4*a*c)) / 2*a
if (~(a = 0))
   x = (-b + sqrt(b * b – 4 * a * c)) / (2 * a)
else
   x = - c / b
```

Subroutines = a major programming artifact

- 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 between one subroutine to the other

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:

- Caller
  - Pushes the necessary argument(s)
  - Calls the callee
- Callee
  - Removes the argument(s) from the stack,
  - Pushes results onto the stack

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

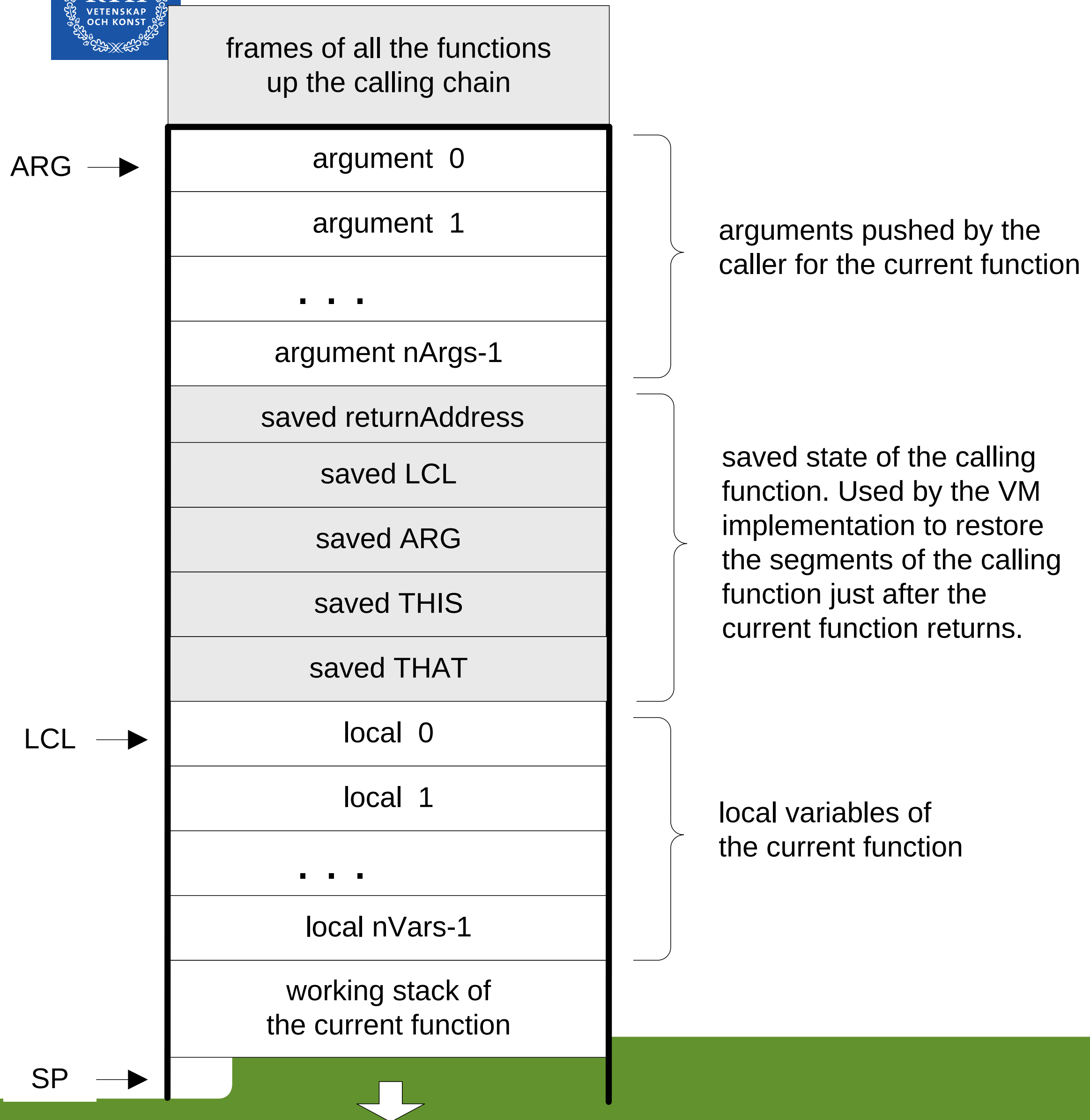
Black = black box magic, delivered by the VM implementation

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 variables are 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 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

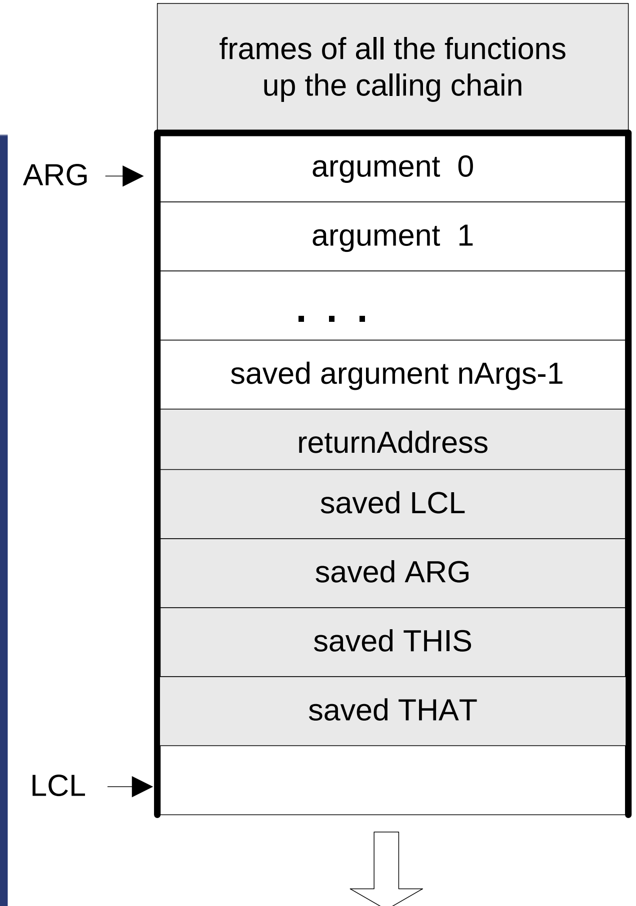
The current function sees only the working stack, and has access only to its memory segments

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 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)

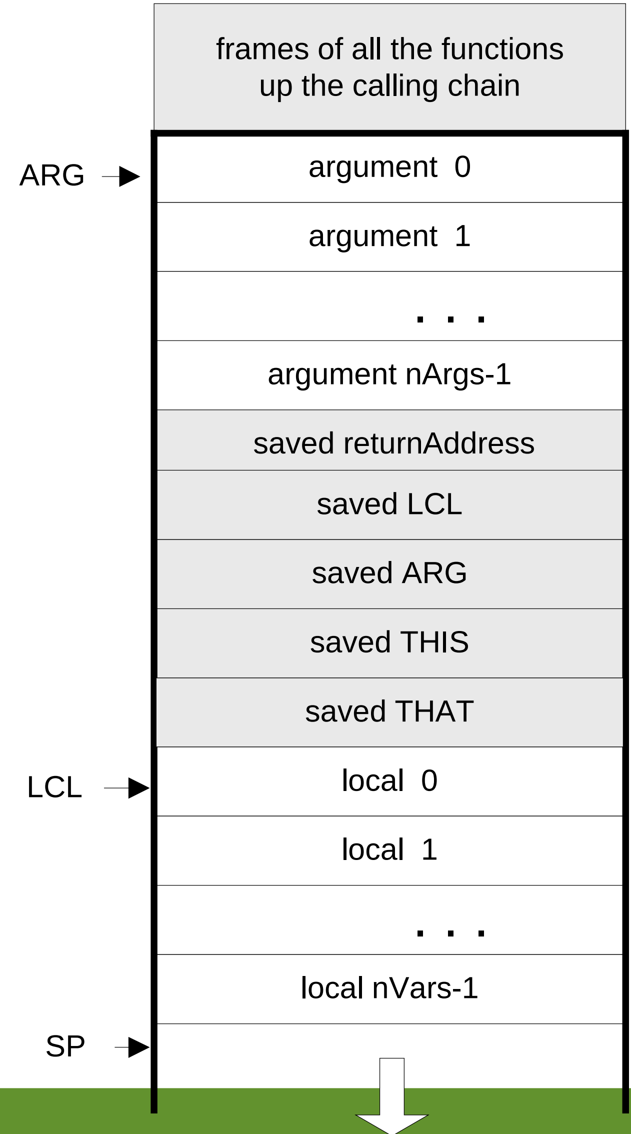
<u>Implementation:</u> If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Implementing the `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
```

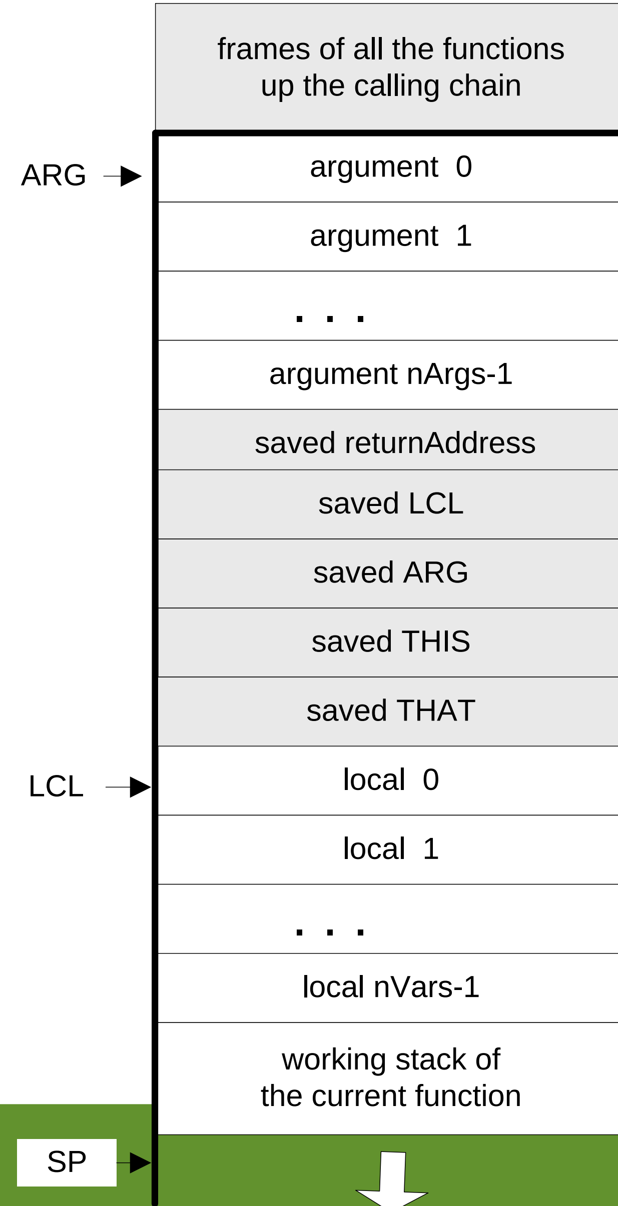


Implementation: If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

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
```



<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.

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
```

Proposed API

CodeWriter: Translates VM commands into Hack assembly code. The routines listed here should be added to the `CodeWriter` module API given in chapter 7.

Routine	Arguments	Returns	Function
`writeInit`	--	--	Writes the assembly code that effects the VM initialization, also called *bootstrap code*. This code must be placed at the beginning of the output file.
`writeLabel`	`label` (string)	--	Writes the assembly code that is the translation of the `label` command.
`writeGoto`	`label` (string)	--	Writes the assembly code that is the translation of the `goto` command.
`writeIf`	`label` (string)	--	Writes the assembly code that is the translation of the `if-goto` command.
`writeCall`	`functionName` (string) `numArgs` (int)	--	Writes the assembly code that is the translation of the `call` command.
`writeReturn`	--	--	Writes the assembly code that is the translation of the `return` command.
`writeFunction`	`functionName` (string) `numLocals` (int)	--	Writes the assembly code that is the trans. of the given `function` command.

Project

Read Chapter 8 of the book

Do Project 7 from the course web page

- Implement the remaining parts of the VM to assembly compiler by extending the code you wrote for Project 6
- Submit your solution by 8am on 27 April, 2017