## The VM language

**Goal:** Complete the specification and implementation of the VM model and language

### Arithmetic / Boolean commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add x, y to VM's memory</td>
</tr>
<tr>
<td>sub</td>
<td>subtract y from x, store result in x</td>
</tr>
<tr>
<td>neg</td>
<td>negate x</td>
</tr>
<tr>
<td>eq</td>
<td>compare x and y, set 1 if equal, 0 otherwise</td>
</tr>
<tr>
<td>gt</td>
<td>compare x and y, set 1 if x &gt; y, 0 otherwise</td>
</tr>
<tr>
<td>lt</td>
<td>compare x and y, set 1 if x &lt; y, 0 otherwise</td>
</tr>
<tr>
<td>and</td>
<td>AND x and y, store result in x</td>
</tr>
<tr>
<td>or</td>
<td>OR x and y, store result in x</td>
</tr>
<tr>
<td>not</td>
<td>invert x</td>
</tr>
</tbody>
</table>

### Program flow commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>(declaration)</td>
</tr>
<tr>
<td>goto</td>
<td>(label)</td>
</tr>
<tr>
<td>if-goto</td>
<td>(label)</td>
</tr>
</tbody>
</table>

### Memory access commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop x</td>
<td>(pop into x, which is a variable or a constant)</td>
</tr>
<tr>
<td>push y</td>
<td>(y being a variable or a constant)</td>
</tr>
</tbody>
</table>

### Function calling commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>(declaration)</td>
</tr>
<tr>
<td>call</td>
<td>(a function)</td>
</tr>
<tr>
<td>return</td>
<td>(from a function)</td>
</tr>
</tbody>
</table>

**Method:** (a) specify the abstraction (model's constructs and commands)  
(b) propose how to implement it over the Hack platform.

---

### The compilation challenge

#### Source code (high-level language)

```java
class Main {
    static int x;
    public static void main() {
        int a, b, c;
        a = Keyboard.readInt("Enter a number");
        b = Keyboard.readInt("Enter a number");
        c = Keyboard.readInt("Enter a number");
        int x = solve(a, b, c);
    }

    public static int solve(int a, int b, int c) {
        int x = 0;
        if (a == 0) {
            x = -c / b;
        } else {
            x = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
        }
        return x;
    }
}
```

**Target code**

```
0000000000010000
1110111100000000
1101010100000000
1111110000000000
0000000000000000
1111001000000000
0000000000000000
1111001110000000
0000000000000000
1111000100000000
0000000000000000
1111000010000000
0000000000000000
1110111100000000
0000000000000000
1111110000000000
0000000000000000
1111001000000000
0000000000000000
1111001110000000
0000000000000000
1111000100000000
0000000000000000
1111000010000000
0000000000000000
1111111100000000
0000000000000000
1110111100000000
0000000000000000
1111110000000000
0000000000000000
```

**Our ultimate goal:** Translate high-level programs into executable code.

[Compiler]
The compilation challenge / two-tier setting

We’ll develop the compiler later in the course.
We now turn to describe how to complete the implementation of the VM language.
That is -- how to translate each VM command into assembly commands that perform the desired semantics.

Typical compiler’s source code input:

The compilation challenge

How to translate such high-level code into machine language?

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.

Typical compiler’s source code input:

VM translator

 Program flow logic (branching)
Boolean expressions
Function call and return logic
Arithmetic expressions

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

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

Label declaration
Unconditional jump to the VM command following the label
VM command following the label

How to translate these 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.
Flow of control

### Pseudo code

```plaintext
if (cond)
    statement1
else
    statement2
```

### VM code

```plaintext
~cond
if-goto elseLabel
statement1
goto contLabel
label elseLabel
statement2
label contLabel
```

Flow of control

### Pseudo code

```plaintext
while (cond)
    statement...
```

### VM code

```plaintext
~(cond)
if-goto exitLabel
statement
goto contLabel
label exitLabel
...
```

Lecture plan

- **Arithmetic / Boolean commands**
  - add
  - sub
  - neg
  - eq
  - gt
  - lt
  - and
  - or
  - not

- **Memory access commands**
  - pop x (pop into x, which is a variable)
  - push y (y being a variable or a constant)

- **Program flow commands**
  - label (declaration)
  - goto (label)
  - if-goto (label)

- **Function calling commands**
  - function (declaration)
  - call (a function)
  - return (from a function)

Subroutines

- **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

```plaintext
// Compute x = (-b + sqrt(b^2 - 4*a*c)) / 2*a
if (~(a = 0))
    x = (-b + sqrt(b^2 - 4*a*c)) / (2 * a)
else
    x = -c / b
```
Subroutines in the VM language

**Calling code, aka "caller" (example)**

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

**Called code, aka "callee" (example)**

```plaintext
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
  label loop
  push argument 0
  push constant 0
  eqif
    goto end // if arg0==0, jump to end
  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
```

### VM subroutine call-and-return commands

- `call`: invokes a subroutine
- `return`: returns from a subroutine

---

**Subroutines in the VM language**

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 callee is responsible for removing the argument(s) from the stack, and for popping onto the stack the result of its execution.

---

**What behind subroutines**

The following scenario happens:

- The caller pushes the necessary arguments and call callee
- The state of the caller is saved
- The space of callee's local variables is allocated
- The callee executes what it is supposed to do
- The callee removes all arguments and pushes the result to the stack
- The space of the callee is recycled
- The caller's state is reinstalled
- Jump back to where is called

---

**Stack as the facility for subroutines**

<table>
<thead>
<tr>
<th>code</th>
<th>flow</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>function a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|        ...
| function b |
|    call c |
|    call d |
|        ...
| function c |
|    call d |
|        ...
| function d |
|        ...
| start a  |
|        start b  |
|        start c  |
|        start d  |
|        end d  |
|        end e  |
|        start b  |
|        start c  |
|        start d  |
|        end b  |
|        end c  |
|        start a  |
Q: Why this particular syntax?
A: Because it simplifies the VM implementation (later).

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

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 omitted
  label end
  push local 0 // push result
  return

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.

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)
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 \texttt{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 implementation of the VM's stack on the host Hack RAM

<table>
<thead>
<tr>
<th>ARD</th>
<th>Global stack:</th>
<th>Working stack:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The entire RAM area dedicated for holding the stack</td>
<td>The stack that the current function sees</td>
</tr>
<tr>
<td>frames of all the functions up the calling chain</td>
<td>arguments pushed by the caller for the current function</td>
<td></td>
</tr>
<tr>
<td>argument 0</td>
<td>saved stack of the calling function</td>
<td></td>
</tr>
<tr>
<td>argument 1</td>
<td>saved the arguments of the calling function just after the current function returns</td>
<td></td>
</tr>
<tr>
<td>\ldots</td>
<td>saved return address</td>
<td></td>
</tr>
<tr>
<td>\ldots</td>
<td>saved LCL</td>
<td></td>
</tr>
<tr>
<td>saved ARGs</td>
<td>saved ARS</td>
<td></td>
</tr>
<tr>
<td>saved THIS</td>
<td>saved THAT</td>
<td></td>
</tr>
<tr>
<td>saved THAT</td>
<td>local 0</td>
<td></td>
</tr>
<tr>
<td>local 0</td>
<td>\ldots</td>
<td></td>
</tr>
<tr>
<td>local ( \text{iVar}+1 )</td>
<td>working stack of the current function</td>
<td></td>
</tr>
</tbody>
</table>

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

- 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

```
// 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 returnAddress
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
```

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

```
// 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

```
// 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 returnAddress // 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.
Example: factorial

High-level code

```c
function fact (n) {
    int result, j;
    result = 1;
    j = 1;
    while ((j=j+1) <= n) {
        result = result * j;
    }
    return result;
}
```

Pseudo code

```c
... loop:
    if ((j=j+1) > n) goto end
    result = result * j
    goto loop
end:
...
```

VM code (first approx.)

```c
function fact(n) {
    push 0
    pop result
    push 1
    pop j
    label loop
    push 1
    push j
    add
    pop j
    push n
    if-goto end
    push result
    mult
    pop result
    goto loop
    label end
    push local 0
    return
}
```

Example: factorial

High-level code

```c
function fact(n) {
    int r;
    if (n=1) {
        r = n * fact(n-1);
    } else {
        r = 1;
    }
    return r;
}
```

VM code (first approx.)

```c
function fact(n) {
    push n
    push 1
    eq
    if-goto else
    push n
    push 1
    sub
    fact
    push n
    mult
    pop r
    goto cont
    label else
    push 1
    pop r
    label cont
    push r
    return
```
Example: factorial

High-level code

```c
int fact(int n) {
    if (n != 1) {
        return n * fact(n-1);
    } else {
        return 1;
    }
}
```

VM code (first approx.)

```c
function fact(n) {
    push argument 0
    push constant 1
    eq
    if-goto else
    push argument 0
    push constant 1
    sub
    call fact
    push argument 0
    call mult
    goto cont
    label else
    push 1
    label cont
    return
}
```

Calling stack for fact(4)

High-level code

```c
function fact(n) {
    int r;
    if (n != 1) {
        r = n * fact(n-1);
    } else {
        r = 1;
    }
    return r;
}
```

stack

```
```


slide 34

slide 35

slide 36
Calling stack for fact(4)

High-level code

```
function fact(n) {
    int r;
    if (n!=1)
        r = n * fact(n-1);
    else
        r = 1;
    return r;
}
```

stack

```
frame fact(4)
frame mult(4,6)
```

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
  ```

Perspective

Benefits of the VM approach

- Code transportability: compiling for different platforms requires replacing only the VM implementation.
- Language inter-operability: code of multiple languages can be shared using the same VM.
- Common software libraries.
- Code mobility: Internet, cloud.

Benefits of managed code:

- Security
- Array bounds, index checking, ...
- Add-on code
- Etc.

VM Cons

- Performance.

Perspective

Some virtues of the modularity implied by the VM approach to program translation:

- Improvements in the VM implementation are shared by all compilers above it.
- Every new digital device with a VM implementation gains immediate access to an existing software base.
- New programming languages can be implemented easily using simple compilers.

Benefits of managed code:

- Security
- Array bounds, index checking, ...
- Add-on code
- Etc.

VM Cons

- Performance.