Motivation

Jack code (example)

```
class Main {
  static int x;

  function void main() {
    // Inputs and multiplies two numbers
    var int a, b, x;
    let a = Keyboard.readInt("Enter a number");
    let b = Keyboard.readInt("Enter a number");
    let x = mult(a,b);
    return;
  }
}

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

Our ultimate goal:

Translate high-level programs into executable code.

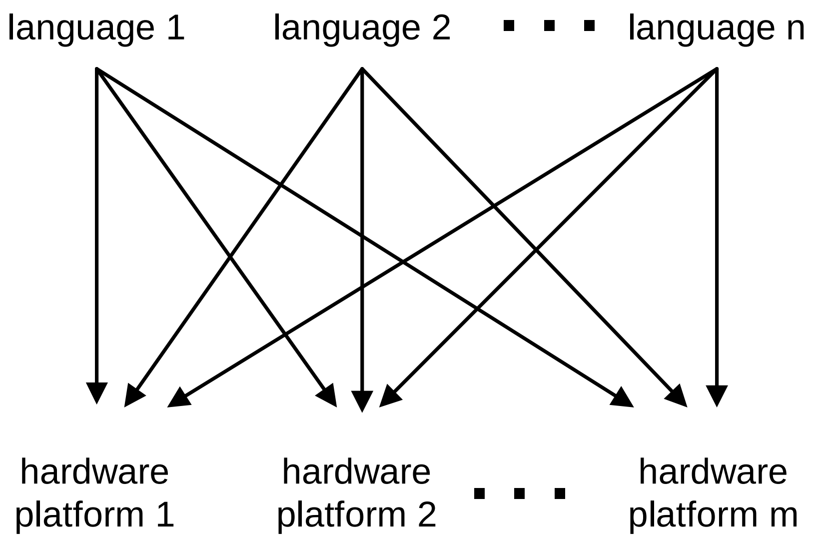
Compiler

Hack 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
...
```

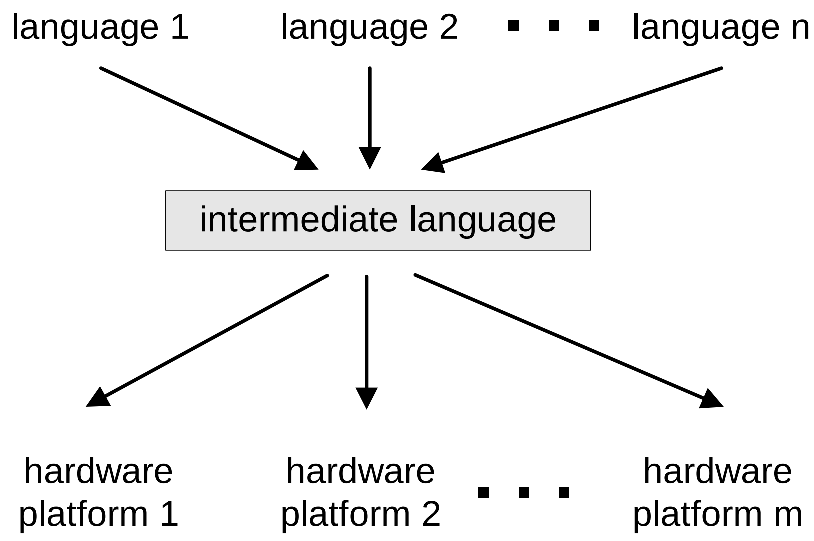
Compilation models

direct compilation:



requires $n \cdot m$ translators

2-tier compilation:

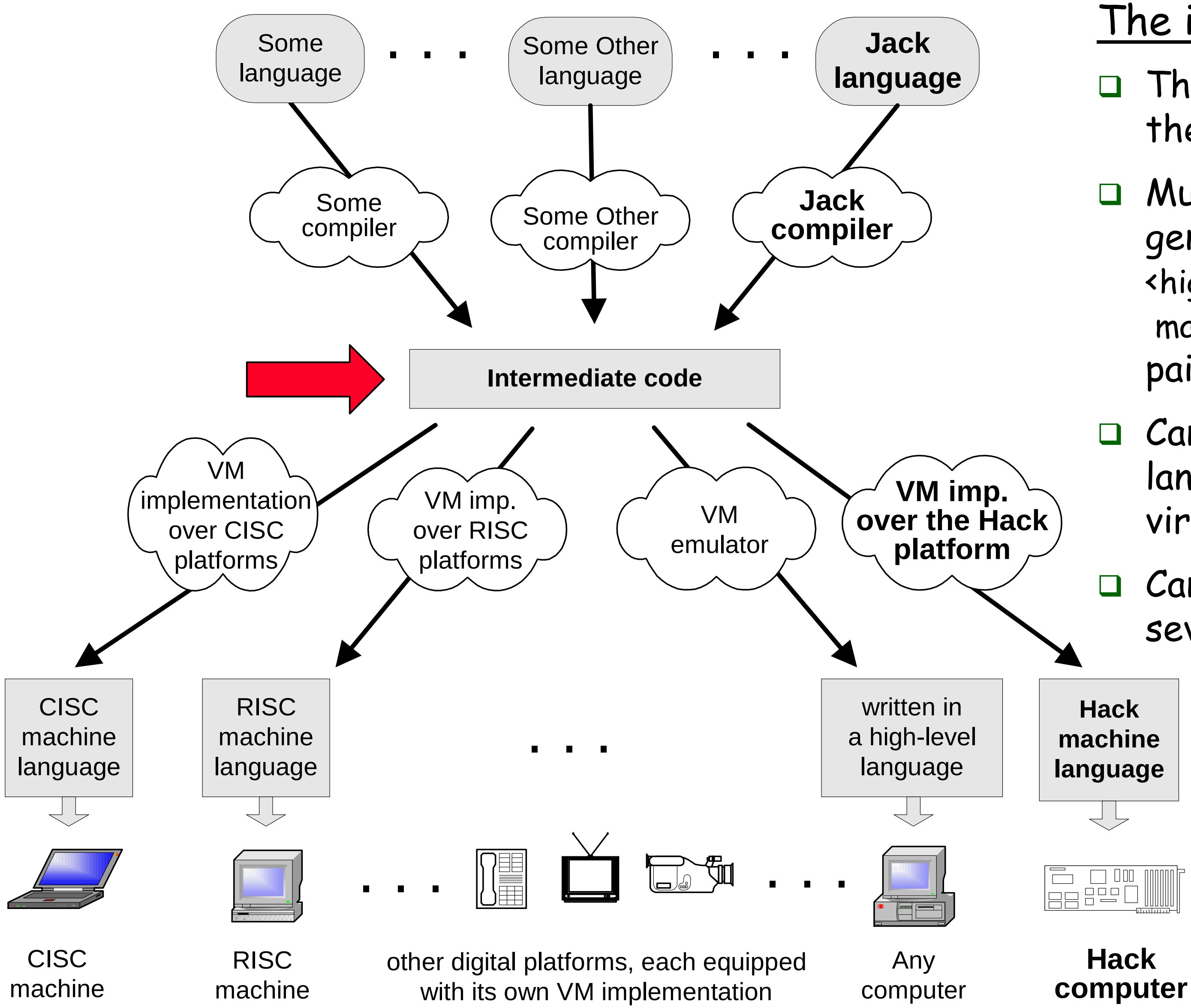


requires $n + m$ translators

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.

The big picture

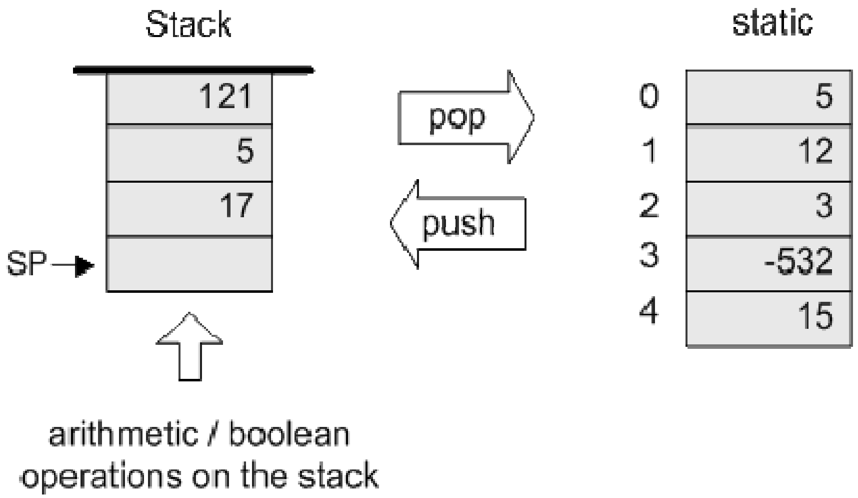


The intermediate code:

- The interface between the 2 compilation stages
- Must be sufficiently general to support many <high-level language, machine-language> pairs
- Can be modeled as the language of an abstract virtual machine (VM)
- Can be implemented in several different ways.

Our VM model is *stack-oriented*

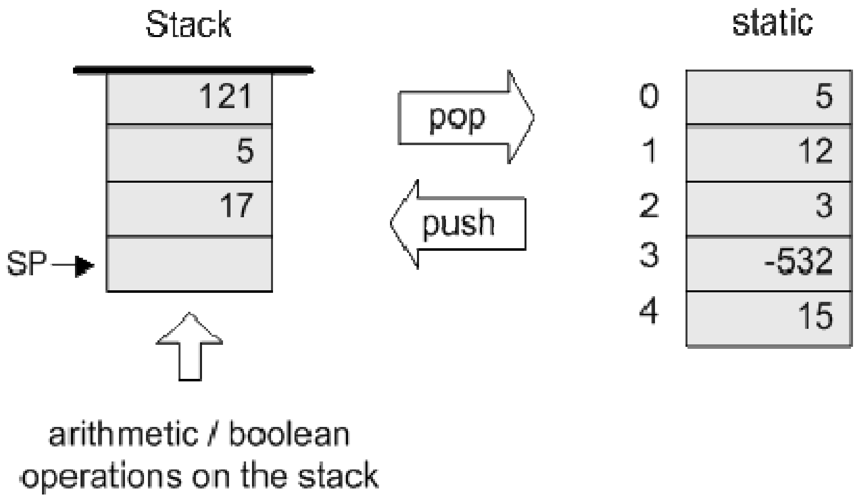
- All operations are done on a stack
- Data is saved in several separate *memory segments*
- All the memory segments behave the same
- One of the memory segments m is called `static`, and we will use it (as an arbitrary example) in the following examples:



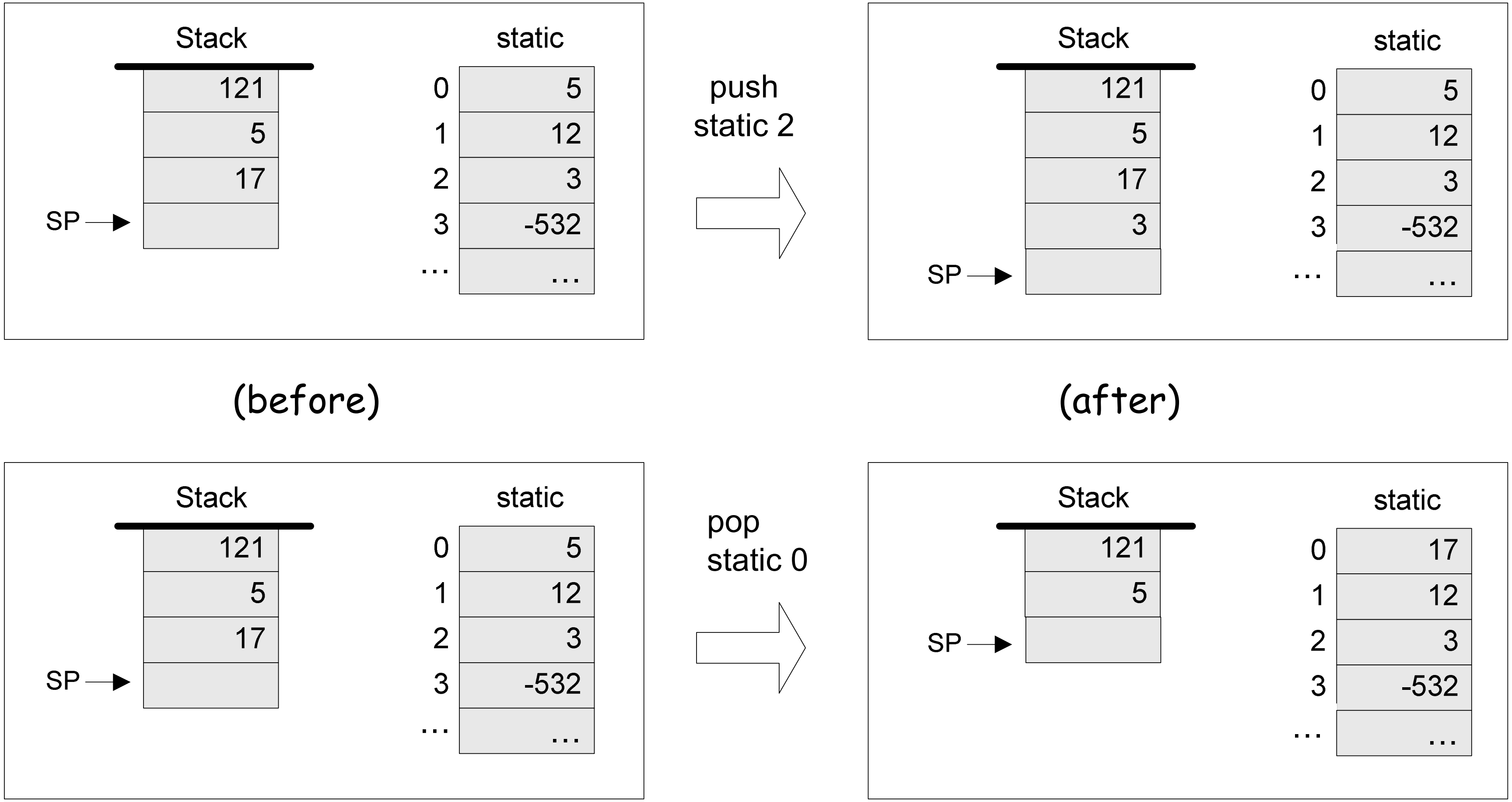
Data types

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 (`0` and `-1`, standing for `true` and `false`)
- a pointer (memory address)



Memory access operations



The stack:

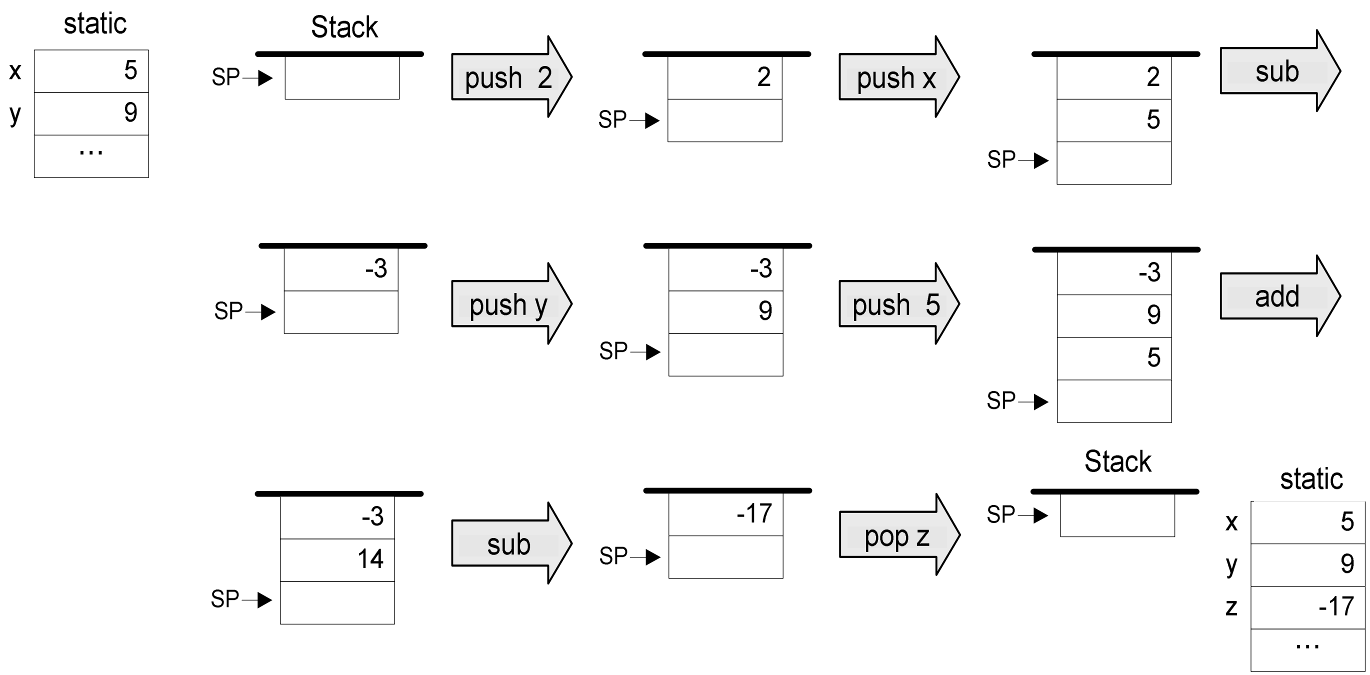
- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.

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)

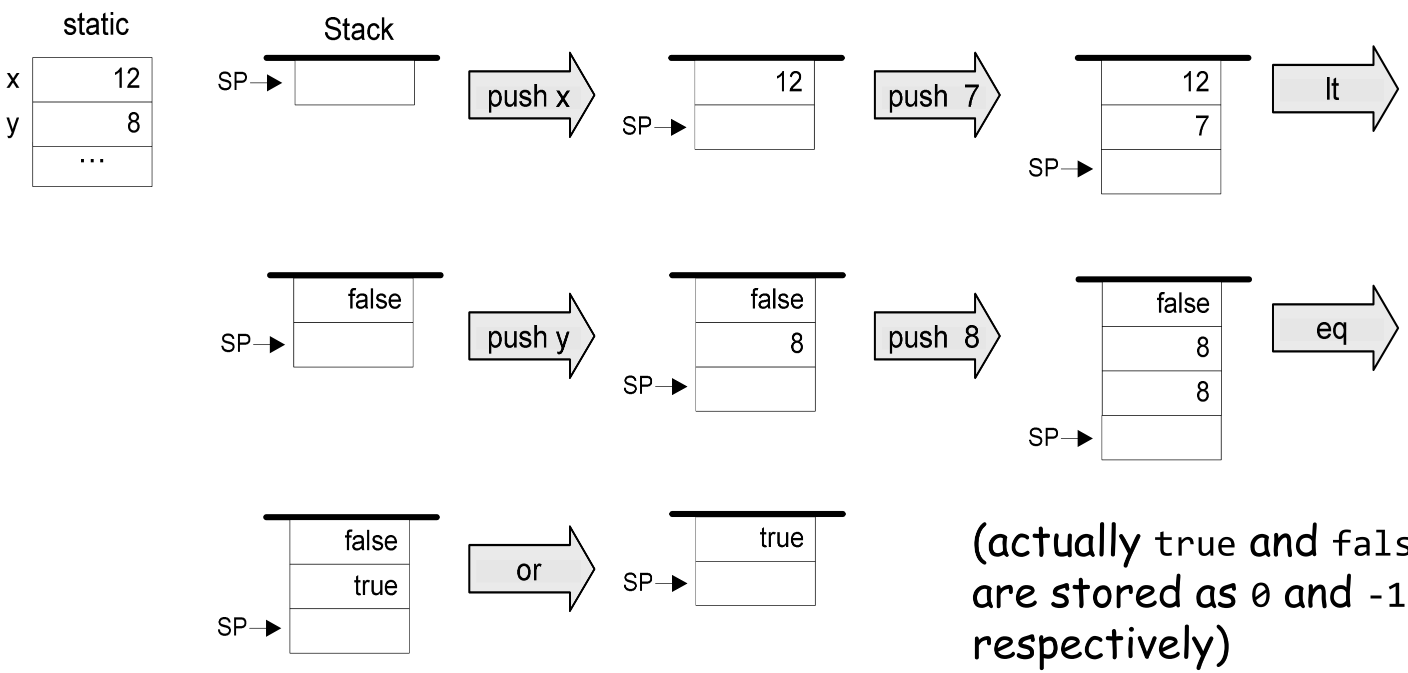


Evaluation of Boolean expressions

VM code (example)

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

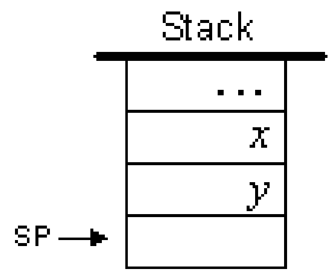
(suppose that
x refers to `static 0`, and
y refers to `static 1`)



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



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 memory 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 different roles of the eight memory segments will become relevant when we'll talk about the compiler

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

So, we'll now see an example of a VM program

The example includes three new VM commands:

- `function` *functionSymbol* // function declaration
- `label` *labelSymbol* // label declaration
- `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:

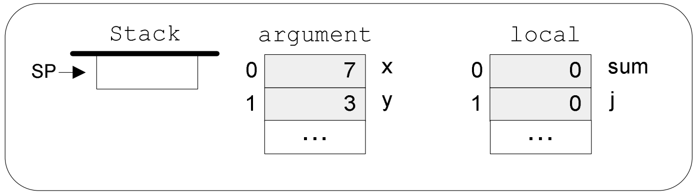
```
push x
push n
gt
if-goto loop          // Note that x, n, and the truth value were removed from the stack.
```

VM programming (example)

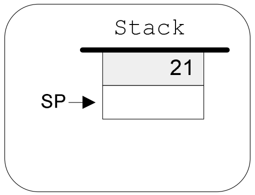
High-level code

```
function mult (x,y) {
  int result, j;
  result = 0;
  j = y;
  while ~(j = 0) {
    result = result + x;
    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
```

VM programming: multiple functions

Compilation:

- A Jack application is a set of 1 or more class files (just like `.java` files).
- When we apply the Jack compiler to these files, the compiler creates a set of 1 or more `.vm` files (just like `.class` files). Each method in the Jack app is translated into a VM function written in the VM language
- Thus, a VM file consists of one or more VM functions.

Execution:

- At any given point of time, only one VM function is executing (the "current function"), while 0 or more functions are waiting for it to terminate (the functions up the "calling hierarchy")
- For example, a `main` function starts running; at some point we may reach the command `call factorial`, at which point the `factorial` function starts running; then we may reach the command `call mult`, at which point the `mult` function starts running, while both `main` and `factorial` are waiting for it to terminate

`The stack:` a global data structure, used to save and restore the resources (memory segments) of all the VM functions up the calling hierarchy (e.g. `main` and `factorial`). The tip of this stack if the working stack of the current function (e.g. `mult`).

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?

- 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 platfrom
- How to do it? see project 8.

Subroutines

```
// 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

"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
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 popping 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).

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

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

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*.

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).

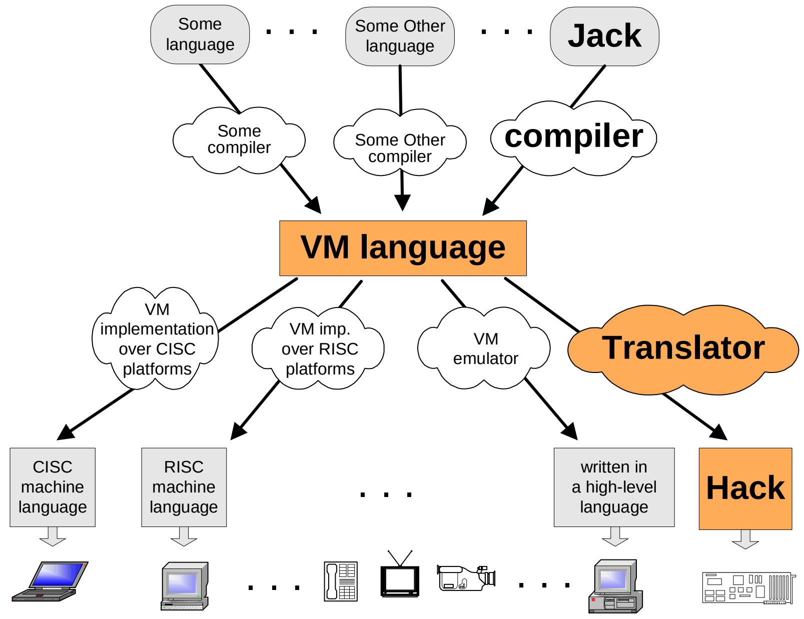
Q: How should we make all this work "like magic"?

A: We'll use the stack cleverly.

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
- 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.