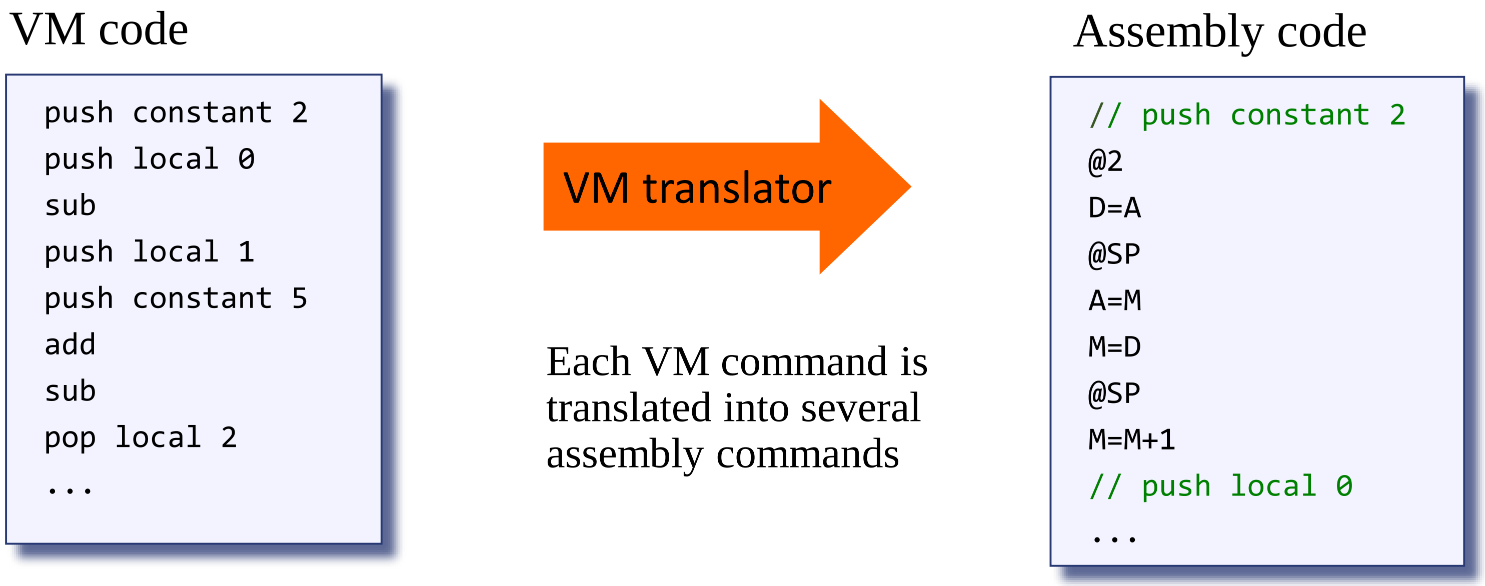
VM translator



VM code

```
push constant 2
push local 0
sub
push local 1
push constant 5
add
sub
pop local 2
...
```

Each VM command is translated into several assembly commands

Assembly code

```
// push constant 2
@2
D=A
@SP
A=M
M=D
@SP
M=M+1
// push local 0
...
```

In order to write a VM translator, we must be familiar with:

- the source language
- the target language
- the VM mapping on the target platform.

Source: VM language

Arithmetic / Logical commands

```
add
sub
neg
eq
gt
lt
and
or
not
```

Memory access commands

`pop` *segment i*

`push` *segment i*

Branching commands

`label` *label*

`goto` *label*

`if-goto` *label*

Function commands

`function` *functionName nVars*

`call` *functionName nArgs*

`return`

Target: symbolic Hack code

<u>A instruction:</u>

```
@value
```

where *value* is either a non-negative decimal constant or a symbol referring to such a constant

Semantics:

- sets the `A` register to *value*;
- makes `M` the `RAM` location whose address is *value*.
  (`M` stands for `RAM[A]`)

<u>C instruction:</u>

```
dest = comp ; jump
```

(*dest* and *jump* are optional)

where:

comp =

```
0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A,
             M,     !M,     -M,      M+1,      M-1, D+M, D-M, M-D, D&M, D|M
```

dest =

```
null, M, D, MD, A, AM, AD, AMD
```

(`M` stands for `RAM[A]`)

jump =

```
null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP
```

Semantics:

- computes the value of *comp* and stores the result in *dest*;
- if (*comp jump* `0`) is true, jumps to execute the instruction in `ROM[A]`.

Standard VM mapping on the Hack platform

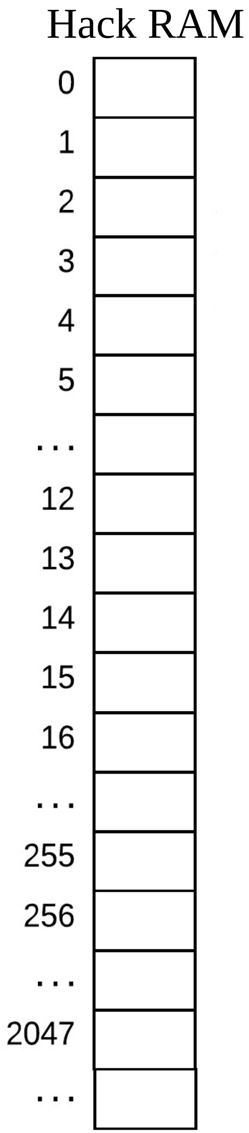
VM mapping decisions:

- How to map the VM’s data structures using the host hardware platform
- How to express the VM’s commands using the host machine language

Standard mapping:

- Specifies how to do the mapping in an agreed-upon way
- Benefits:
    - ❑ Compatibility with other software systems
    - ❑ Standard testing.

Standard VM mapping on the Hack platform



Hack RAM

0

1

2

3

4

5

...

12

13

14

15

16

...

255

256

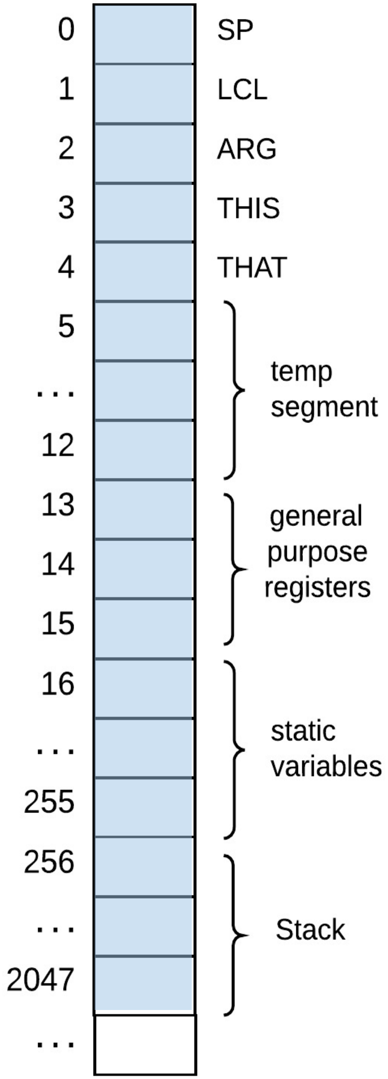
...

2047

...

Standard VM mapping on the Hack platform

Hack RAM



local, argument, this, that:

allocated dynamically to the RAM. The base addresses of these allocations are kept in the segment pointers LCL, ARG, THIS, THAT

accessing *segment i* should result in accessing RAM[**segmentPointer* + *i*]

constant: accessing constant *i* should result in supplying the constant *i*

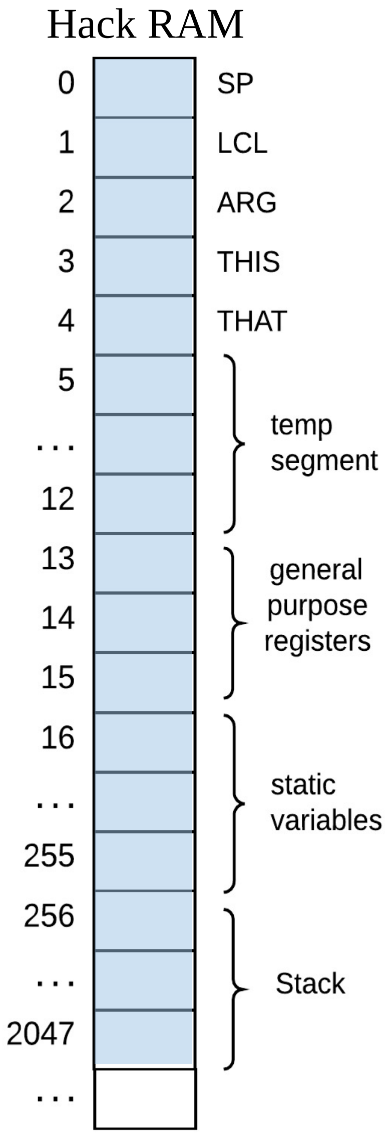
static: accessing static *i* within file Foo.vm should result in accessing the assembly variable Foo.*i*

temp: fixed segment, mapped on RAM addresses 5-12.
accessing temp *i* should result in accessing RAM[5+*i*]

pointer: fixed segment, mapped on RAM addresses 3-4.

accessing pointer 0 should result in accessing THIS
accessing pointer 1 should result in accessing THAT

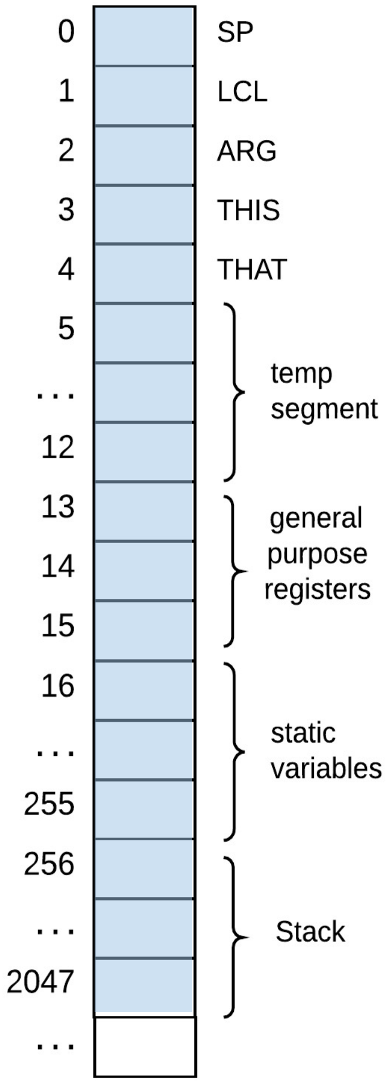
Standard VM mapping on the Hack platform



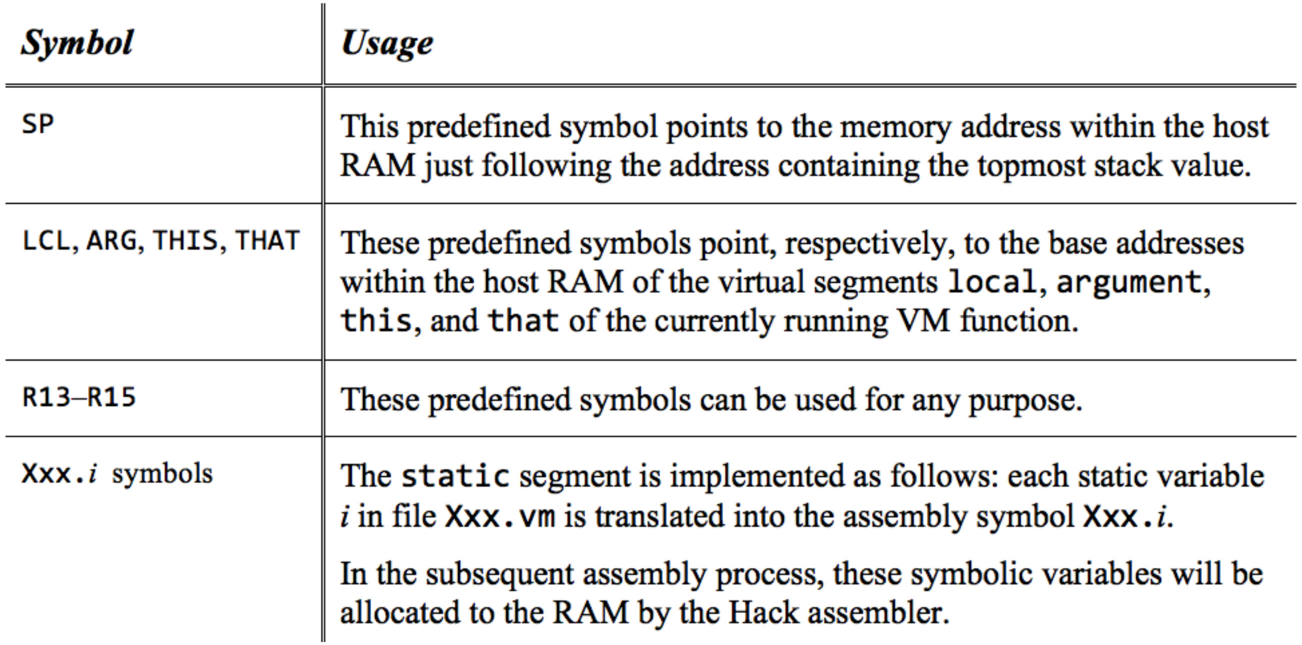
In order to realize this mapping, the VM translator should use some special variables / symbols:

Standard VM mapping on the Hack platform

Hack RAM



In order to realize this mapping, the VM translator should use some special variables / symbols:

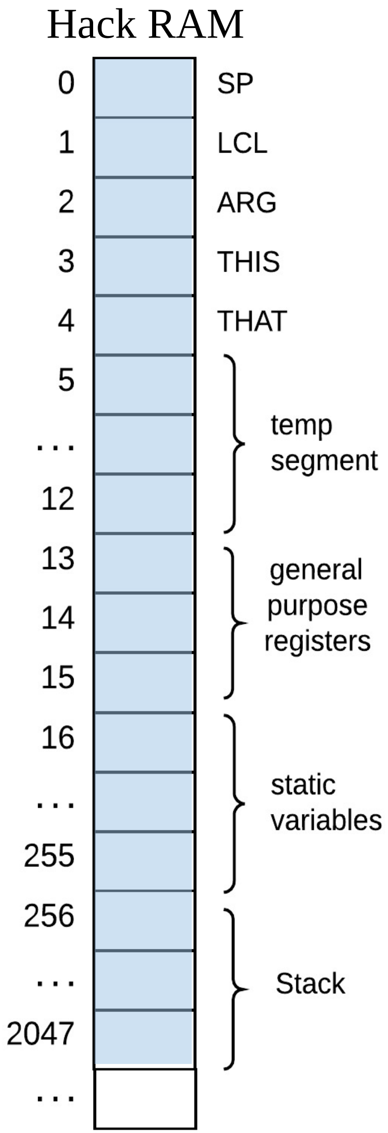


Symbol	*Usage*
SP	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
LCL, ARG, THIS, THAT	These predefined symbols point, respectively, to the base addresses within the host RAM of the virtual segments local, argument, this, and that of the currently running VM function.
R13–R15	These predefined symbols can be used for any purpose.
Xxx.*i* symbols	The static segment is implemented as follows: each static variable *i* in file Xxx.vm is translated into the assembly symbol Xxx.*i*. In the subsequent assembly process, these symbolic variables will be allocated to the RAM by the Hack assembler.

Implementation note:

The standard mapping will be extended in project 8,
when we’ll complete the VM translator’s implementation.

Standard VM mapping on the Hack platform



local, argument, this, that:
allocated dynamically to the RAM. The base addresses of these allocations are kept in the segment pointers LCL, ARG, THIS, THAT

accessing *segment i* should result in accessing RAM[**segmentPointer* + *i*]

constant: accessing constant *i* should result in supplying the constant *i*

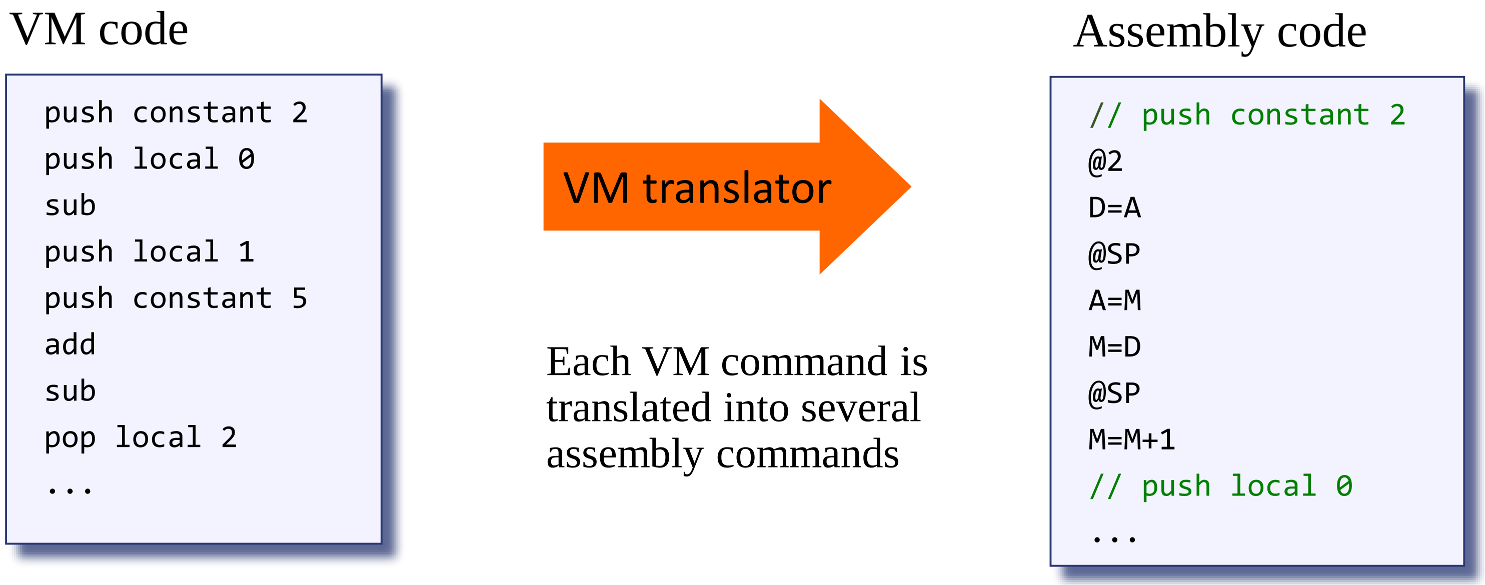
static: accessing static *i* within file Foo.vm should result in accessing the assembly variable Foo.*i*

temp: fixed segment, mapped on RAM addresses 5-12.
accessing temp *i* should result in accessing RAM[5+*i*]

pointer: fixed segment, mapped on RAM addresses 3-4.

accessing pointer 0 should result in accessing THIS
accessing pointer 1 should result in accessing THAT

VM translator



VM code

```
push constant 2
push local 0
sub
push local 1
push constant 5
add
sub
pop local 2
...
```

Assembly code

```
// push constant 2
@2
D=A
@SP
A=M
M=D
@SP
M=M+1
// push local 0
...
```

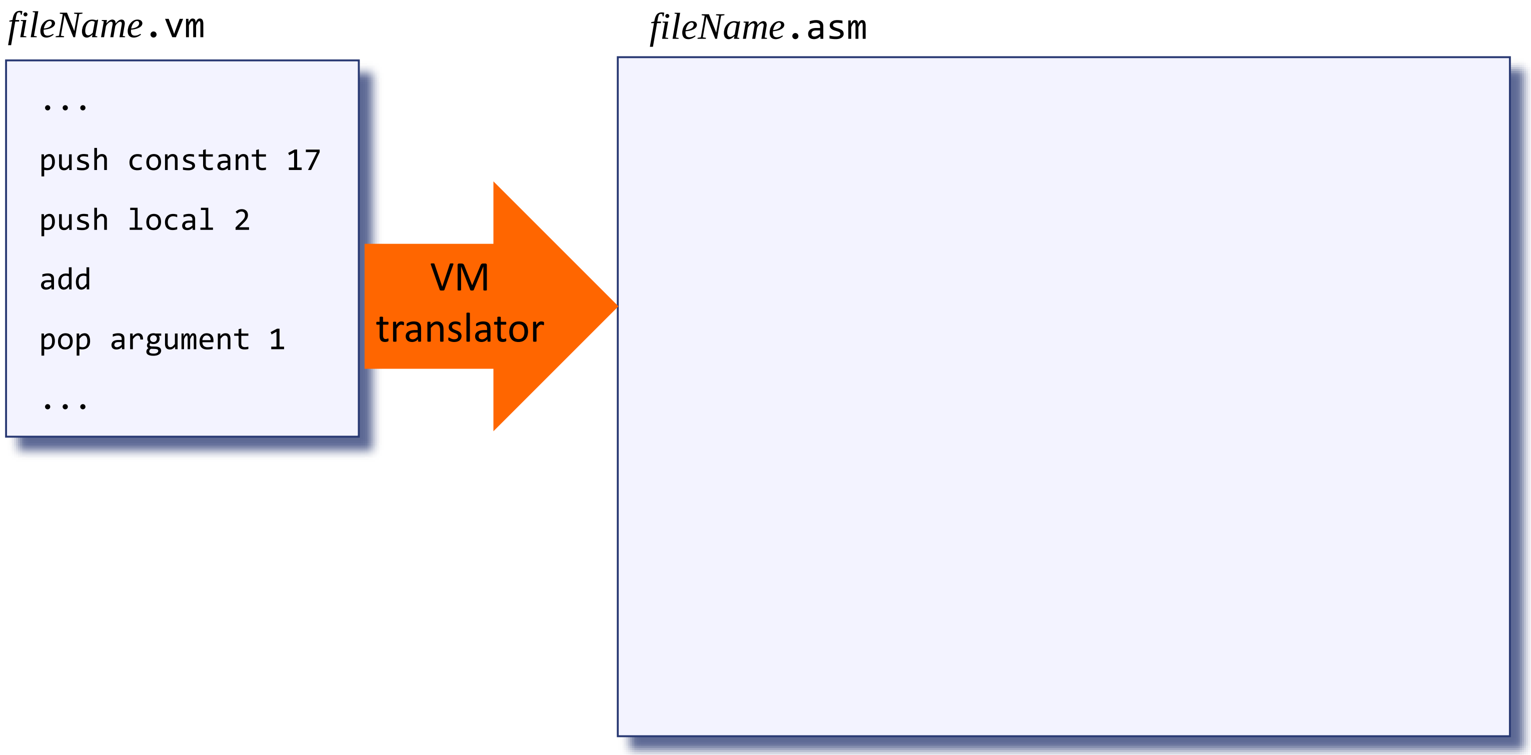
Each VM command is translated into several assembly commands

In order to write a VM translator, we must be familiar with:

- the source language
- the target language
- the VM mapping on the target platform.

Project 7

Objective: build a basic VM translator that handles a subset of the VM language: stack arithmetic and memory access (push/pop) commands



fileName.vm

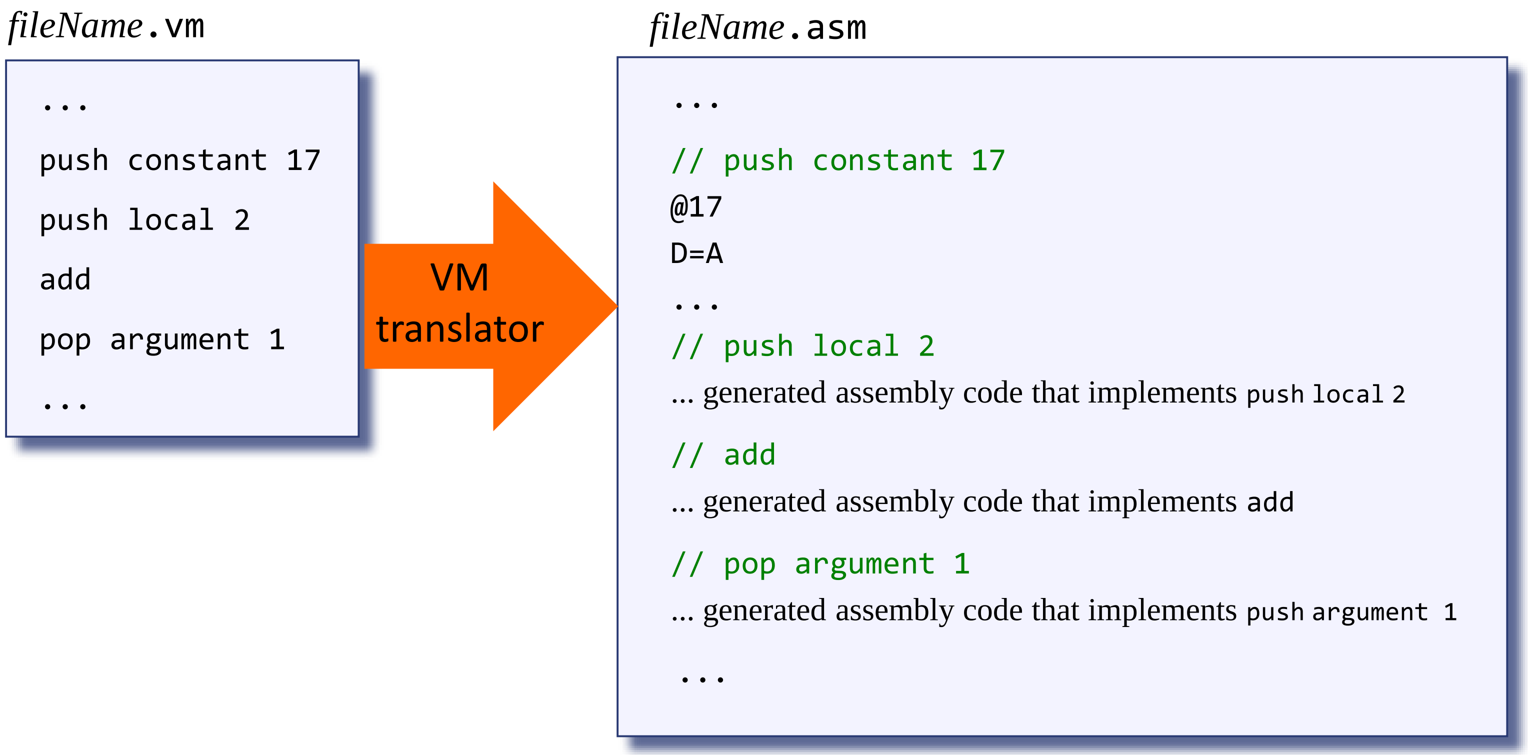
```
...
push constant 17
push local 2
add
pop argument 1
...
```

VM translator

fileName.asm

Project 7

Objective: build a basic VM translator that handles a subset of the VM language: stack arithmetic and memory access (push/pop) commands



fileName.vm

```
...
push constant 17
push local 2
add
pop argument 1
...
```

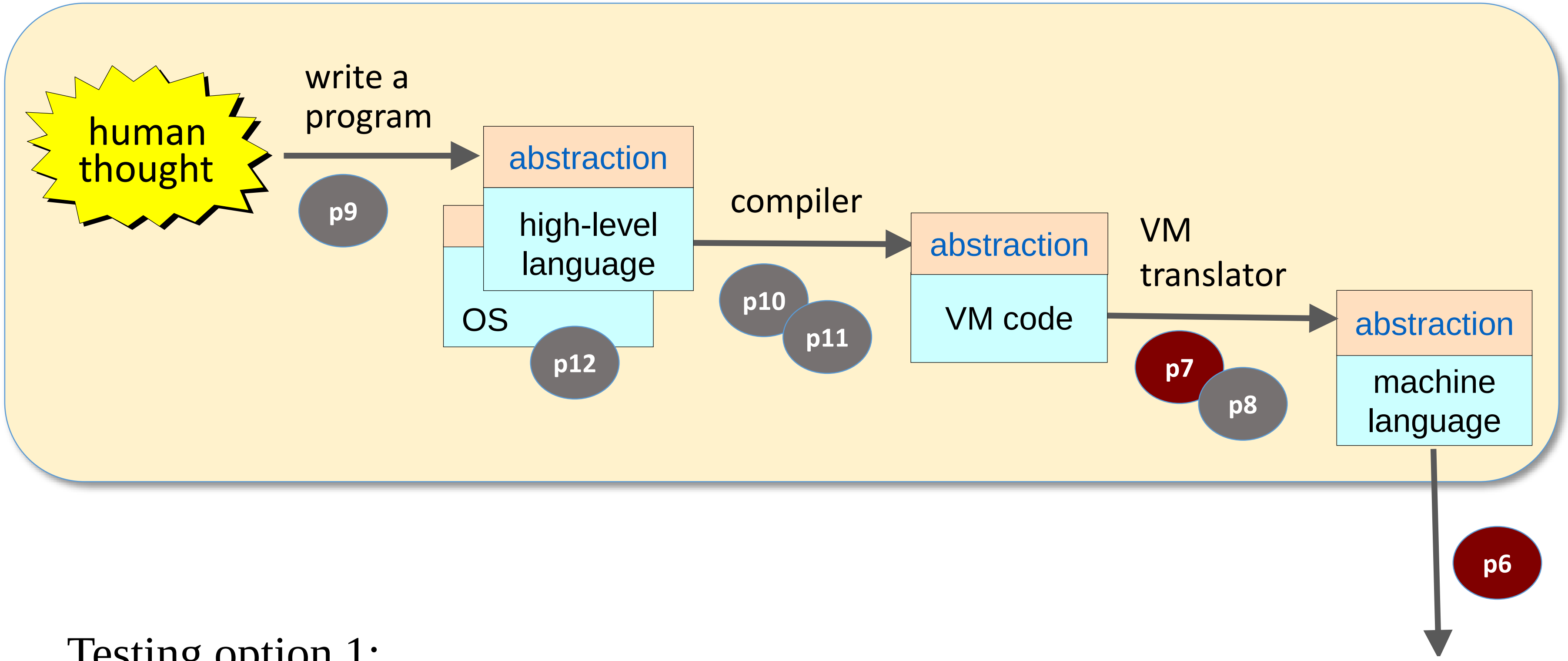
fileName.asm

```
...
// push constant 17
@17
D=A
...
// push local 2
... generated assembly code that implements push local 2
// add
... generated assembly code that implements add
// pop argument 1
... generated assembly code that implements push argument 1
...
```

To test the translation:

Run the generated code on the target platform

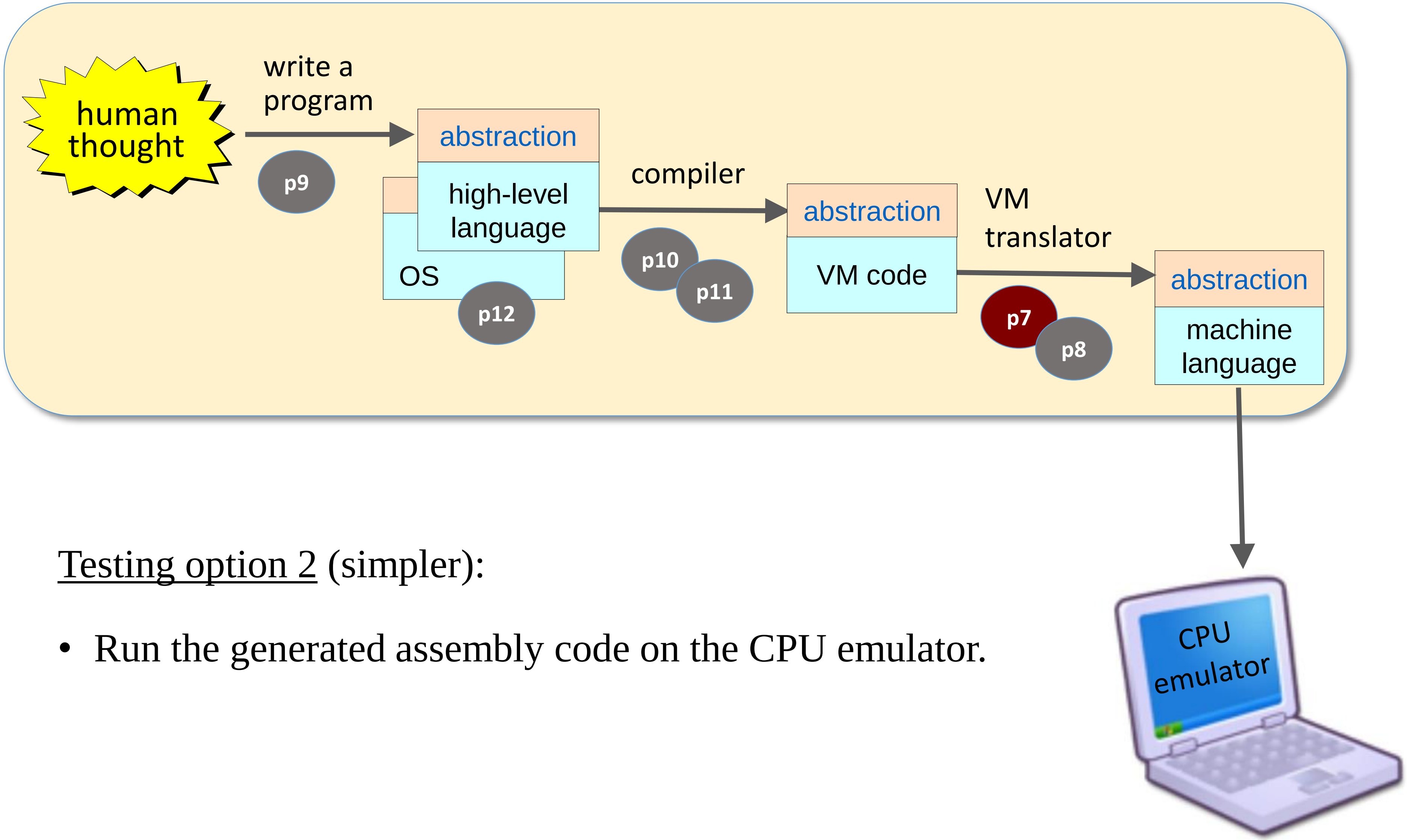
Project 7: testing



Testing option 1:

- Translate the generated assembly code into machine language,
- Run the binary code of the Hack computer

Project 7: testing



Testing option 2 (simpler):

- Run the generated assembly code on the CPU emulator.

Development Plan

Objective: build a basic VM translator that handles the VM language *stack arithmetic* and *memory access* (push/pop) commands

Contract

- Write a VM-to-Hack translator, conforming to the *Standard VM-on-Hack Mapping*
- Use your VM translator to translate and test the supplied .vm programs, yielding corresponding .asm programs

Test programs

- SimpleAdd
- StackTest
- BasicTest
  - BasicTest.vm
  - BasicTest.tst
  - BasicTest.cmp
  - BasicTestVME.tst
- PointerTest
- StaticTest

BasicTest.vm (example)

```
...
push constant 510
pop temp 6
push local 0
push that 5
add
push argumen
sub
...
```

BasicTest.asm

```
...
// push constant 510
@510
D=A
...
```

Development Plan

Objective: build a basic VM translator that handles the VM language
stack arithmetic and *memory access* (`push`/`pop`) commands

Contract

- Write a VM-to-Hack translator, conforming to the *Standard VM-on-Hack Mapping*
- Use your VM translator to translate and test the supplied `.vm` programs, yielding corresponding `.asm` programs
- When executed on the supplied CPU emulator, the generated `.asm` programs should deliver the same results mandated by the supplied test scripts and compare files.

Test programs

- ❑ `SimpleAdd`
- ❑ `StackTest`
- ❑ `BasicTest`
  - ❑ `BasicTest.vm`
  - ❑ `BasicTest.tst`
  - ❑ `BasicTest.cmp`
  - ❑ `BasicTestVME.tst`
- ❑ `PointerTest`
- ❑ `StaticTest`

`BasicTest.vm` (example)

```
...
push constant 510
pop temp 6
push local 0
push that 5
add
push argumen
sub
...
```

`BasicTest.asm`

```
...
// push constant 510
@510
D=A
...
```

Development Plan

Objective: build a basic VM translator that handles the VM language *stack arithmetic* and *memory access* (push/pop) commands

For each test *xxx*.vm program:

0. (optional) load *xxx*VME.tst into the VM emulator; run the test script and inspect the program’s operation
1. use your translator to translate *xxx*.vm; The result will be a file named *xxx*.asm
2. inspect the generated code; If there’s a problem, fix your translator and go to stage 1
3. Load *xxx*.tst into the CPU emulator
4. Run the test script, inspect the results
5. If there’s a problem, fix your translator and go to stage 1.

Test programs

- ❑ SimpleAdd
- ❑ StackTest
- ❑ BasicTest
  - ❑ BasicTest.vm
  - ❑ BasicTest.tst
  - ❑ BasicTest.cmp
  - ❑ BasicTestVME.tst
- ❑ PointerTest
- ❑ StaticTest

BasicTest.vm (example)

```
...
push constant 510
pop temp 6
push local 0
push that 5
add
push argumen
sub
...
```

BasicTest.asm

```
...
// push constant 510
@510
D=A
...
```

Tools and resources

Objective: build a basic VM translator that handles the VM language
stack arithmetic and *memory access* (`push`/`pop`) commands

Tools and resources:

- Test programs and compare files: `nand2tetris/projects/07`
- Experimenting with the test VM programs: the supplied *VM emulator*
- Translating the test VM programs into assembly: your *VM translator*
- Testing the resulting assembly code: the supplied *CPU emulator*
- Programming language for implementing your VM translator: Java, Python, ...
- Tutorials: VM emulator, CPU emulator (`nand2tetris` web site)
- Reference: chapter 7 in *The Elements of Computing Systems*