Project 7.

Parser

- Handles the parsing of a single .vm file
- Reads a VM command, parses the command into its lexical components, and provides convenient access to these components
- Ignores all white space and comments

Routine	Arguments	Returns	Function
commandType	—	C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL	Returns a constant representing the type of the current command. C_ARITHMETIC is returned for all the arithmetic/logical commands.
arg1	—	string	Returns the first argument of the current command. In the case of C_ARITHMETIC, the command itself (add, sub, etc.) is returned. Should not be called if the current command is C_RETURN.
arg2	—	int	Returns the second argument of the current command. Should be called only if the current command is C_PUSH, C_POP, C_FUNCTION, or C_CALL.

divides into type of commands

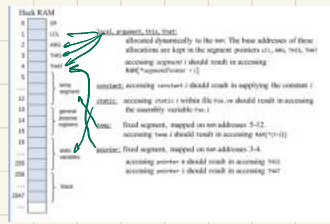
CodeWriter

Generates assembly code from the parsed VM command:

Routine	Arguments	Returns	Function
Constructor	Output file / stream	—	Opens the output file / stream and gets ready to write into it.
writeArithmetic	command (string)	—	Writes to the output file the assembly code that implements the given arithmetic command.
WritePushPop	command (C_PUSH or C_POP), segment (string), index (int)	—	Writes to the output file the assembly code that implements the given command, where command is either C_PUSH or C_POP.
Close	—	—	Closes the output file.

▿ Recommended way to do it

Generally accepted mapping:



VM translator needs to understand these symbols

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.

write Arithmetic

Arithmetic / Logical commands:

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

push const 1
push const 2
add/sub/and/or — pop x, pop y, push x+y

```
SP--
y=*SP
SP--
x=*SP
*SP=x+y
SP++
```

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

push const 1
neg — pop x, !x, push x

```
SP--
y=*SP
y=!y
*SP=y
SP++
```

```
@SP
M=M-1
@SP
A=M
M=!M
@SP //SP++
M=M+1
```

push const 1
push const 2
eq. — pop x1, pop x2, y=x1==x2, push y

```
SP--
y=*SP
SP--
x=*SP
@LOOP_TRUE
x-y;JEQ
*SP=0
@AFTER
0;JMP
(LOOP_TRUE)
*SP=-1
(AFTER)
SP++
```

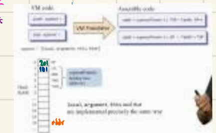
eq → JEQ
lt → JLT
gt → JGT

```
@SP
M=M-1
@SP
A=M
D=M
@SP
M=M-1
@SP
A=M
D=M-D
@LOOP_TRUE
D;JEQ
@SP
A=M
M=0
@AFTER
0;JMP
(LOOP_TRUE)
@SP
A=M
M=-1
(AFTER)
@SP //SP++
M=M+1
```

Write push/pop:

local, argument, this, that

- ▿ Normal Java method has local variables, argument variables, member var of current object, other data variables
  - to translate these semantics, need separate memory segments. VM translator will map those
- ▿ Abstractly, implement in similar way
- push segment/arg 1, pop local 2 — dont care where segments are in RAM



Eg: push local 3

Logic: addr = LCL + arg 2 (101, 3)

push: *SP = *addr, SP++

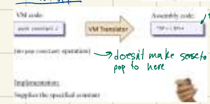
pop: addr = LCL + arg 2, SP--, *addr = *SP

```
@R15
M=D
@SP
M=M-1
@SP
A=M
D=M
@R15
A=M
M=D
```

```
@LCL //D=addr (101)
D=M
@arg2 (3)
D=D+A //D=104
A=D //D=RAM[LCL+arg]
D=M
@SP //SP=D
A=M
M=D
@SP //SP++
M=M+1
```

Eg: Set RAM[5]=RAM[3]
```
@5
D=M
@3
M=D
```

Constant

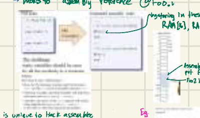


→ doesn't make sense to pop to here

truly virtual → No actual memory segment
- simply assigns const. i to Stack pointer

static

- ▿ if compiler encounters static variables (class level)
  - → gets dedicated segment
  - → Challenge - these variables need to be seen by all methods of program
    - → required to store in global space - all code can see
    - → works to assembly reference "@Foo.i"



by storing in these variables, will map to RAM[16], RAM[17], ..., RAM[255]

Now can be Accessed "globally"

→ in order of declaration

→ static 5 can be alone, static 2 → ordering of index values not same

Note: this is unique to Hack assembler, other tricks implement static differently

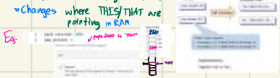
temp

- ▿ Sometimes compiler needs to use temporary storage variables
- ▿ Our VM has 8 places for this (fixed)
- ▿ Mapped to base address 5-12

pointer

- ▿ segment purpose used by compiler (makes sense later)
  - → if compiler translates method, needs to remember base address of this/that segment
  - → ONLY needs 2 segments, 0/1
- ▿ Changes where THIS/THAT are pointing in RAM

Eg. — pops 2000 to "THAT"



→ pushes to this or that

push: *SP = THIS/THAT, SP++

```
@THIS/THAT
D=M
@SP
A=M
M=D
@SP
M=M+1
```

pop: SP--, THIS/THAT = *SP

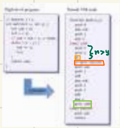
```
@SP
M=M-1
@SP
A=M
D=M
@THIS/THAT
M=D
```

push pointer 0 → *SP = RAM[3]
vs.
push this 0 → *SP = RAM[RAM[3]]

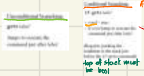
Project 8

Branching

Eg. Computing Multiplication w/ Looping Addition



Compiler adds labels for start/end of while loop



f n>y, goto end

→ CHANGES Stack
- by popping off top value to evaluate if goto or not

top of stack must be bool

similar to (labels), @LABEL 0;JMP for uncond. D;JLE, etc.

label LABEL	(LABEL)
goto LABEL	@LABEL 0;JMP
if-goto LABEL	pop cond Jump to label

SP--
x = *SP

@SP
M=M-1
@SP
A=M
D=M

x=-1 is true
x=0 is false

@LABEL
x;JNE

@LABEL
D;JNE

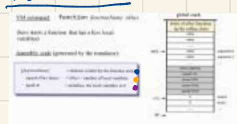
if NOT 0, must be true

CodeWriter

Additional functionality:

Routine	Arguments	Returns	Function
setFileName	fileName (string)	—	Informs the codeWriter that the translation of a new VM file has started (called by the main program of the VM translator).
writeInit	—	—	Writes the assembly instructions that effect the bootstrap code that initializes the VM. This code must be placed at the beginning of the generated *.asm file.
writeLabel	✓ label (string)	—	Writes assembly code that effects the label command.
writeGoto	✓ label (string)	—	Writes assembly code that effects the goto command.
writeIf	✓ label (string)	—	Writes assembly code that effects the if-goto command.
writeFunction	✓ functionName(string) numVars (int)	—	Writes assembly code that effects the function command.
writeCall	functionName(string) numArgs (int)	—	Writes assembly code that effects the call command.
writeReturn	—	—	Writes assembly code that effects the return command.

Function



- All we have to do is set up local by pushing 0s
- Now can proceed to rest of fxn

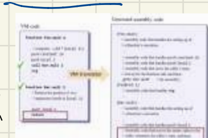
(LABEL)

push0:

```
@0
D=A
@SP
A=M
M=D
@SP
M=M+1
```

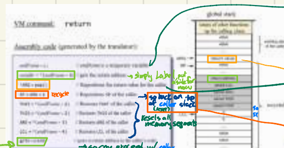
repeat nVars times

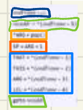
Return



return label that was generated w/ call

As per contract, when return is called, top of stack is return value





```
@LCL
D=M
@R15
M=D
@5
D=D-A
A=D
D=M
@R14
M=D
```

```
@SP
M=M-1
@SP
A=M
D=M
@ARG
A=M
M=D
```

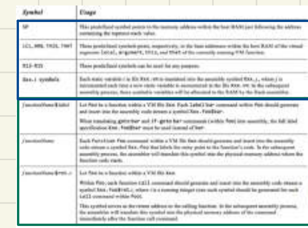
```
@ARG
D=M+1
@SP
M=D
```

```
@R15
D=M
@1/2/3/4
D=D-A
A=D
D=M
@THAT/THIS
M=D
etc.
```

```
@R14
A=M
0;JMP
```

Need to implement:

→ 8.7
- Mapping

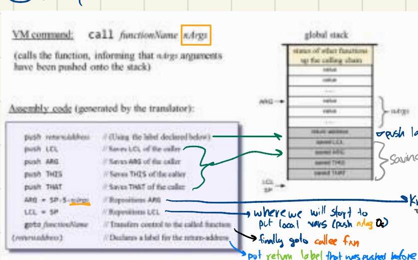


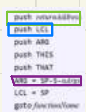
Project 7 (done)

Project 8

- Call

Call





```
@{fxnName}$ret.
D=A
@SP
A=M
M=D
@SP
M=M+1
```

```
@LCL /ARG/THIS/THAT
D=M
@SP
A=M
M=D
@SP
M=M+1
```

```
@nARGs
D=D-A
@SP
D=D-M
@ARG
M=D
```

```
@5
D=A
```

Remaining:

Fib Element

Statics test