COMSM1302 電腦架構概論
第九週完整教材：Hack 虛擬機

虛擬機與中介表示法 · 堆疊機算術與邏輯 · 分支與記憶體 · VM 翻譯器

依據 John Lapinskas（University of Bristol）課程講義編寫並延伸

2026 年 7 月

本教材的使用方式：本講義整合了第九週四份投影片（9-1 Virtual machines and intermediate representations、9-2 The Hack VM I: Structure, arithmetic and logic、9-3 The Hack VM II: Branching and memory、9-4 Implementing the Hack VM translator）的全部內容，並補充了堆疊機 vs 暫存器機、JVM ／ WebAssembly、VM 指令翻譯成組語的具體範例等延伸知識。每章結尾附有「本章重點」整理；第 5 章為綜合練習題，附完整詳解。上週我們寫了組譯器（組語 → 機器碼）；本週往抽象階梯再爬一層：學習 Hack 的**中介表示法** Hack VM，並理解如何寫一個 **VM 翻譯器**把 VM 程式碼編譯成組語。

目錄

1　虛擬機與中介表示法（VMs and IRs）

1.1　下一個目標

回想多數高階語言會先編譯到**中介表示法（intermediate representation, IR）**，再進一步編譯到組語。接下來兩週我們要學 Hack 的 IR——**Hack VM**——並寫一個 **VM 翻譯器（VM translator）**把 Hack VM 程式碼編譯成組語。

1.2　為什麼要用 IR?

IR 讓編譯工作乾淨地切成三段：

階段	工作	相依性
前端（front end）	高階語言 → IR（含最佳化）	**不**依賴架構、依賴語言
中端（middle end）	IR 內部的最佳化	**不**依賴架構、**不**依賴語言
後端（back end）	IR → 組語（含最佳化）	依賴架構、**不**依賴語言

許多語言的編譯器可以（也確實）共用同一個 IR：基於既有 IR 做新編譯器時**只需要寫前端**；ISA ／微架構更新時**只需要改後端**——這些工作（大致上）只要做一次，而不是每個編譯器各做一次。

1.3　真實世界的 IR

- **LLVM**：從零設計的通用開源 IR，力求適用於盡可能多的語言。多數新編譯器都做成 LLVM 的前端。
- **GIMPLE**：GCC（最重要）的 IR。GCC 在 C 與 C++ 效能上與 LLVM 系編譯器不相上下，且對老舊架構的支援明顯更強。
- **JVM**：Java 的 IR。Java **完全不用編譯器後端**，而是跑一個 VM 直接執行 IR——保證了可攜性，但犧牲效能。

1.4　虛擬機（Virtual Machines）

回想 Hack 中，組語、ISA 與微架構的設計**緊緊綁在一起**：決定支援哪些指令時，得在三者間妥協——寫組語時什麼有用、什麼塞得進 16 位元的字、什麼在微架構容易實作。

虛擬機（virtual machine, VM）

在 IR 中，我們**切斷這些連結**：把 IR 程式碼想成跑在一台 **虛擬機**上——一台**不以實體硬體存在**、而是由實際存在的電腦**模擬**出來的電腦。惱人的是，「VM」一詞既指這台「電腦」在實體硬體上跑的 **具體實例**，也指這台「電腦」的**設計**——你可能同時說「在 Windows 上同時跑兩個 Hack VM」和「Hack VM 的指令集」。

1.4.1　另一種虛擬機

本課程關心的是 **IR 的 VM**（多半永遠不會做成真硬體）。但**真實架構的 VM** 也很常用，用途例如：執行不受信任的軟體（隔離真正的作業系統與硬體）；多台實體機器跑同步的虛擬伺服器副本以便**故障轉移**；一台實體超級電腦跑多個 VM 做分散式高效能運算（AWS、BC4）；從外部分析運行中的作業系統（找安全漏洞）；跑其他架構的原生軟體（如行動裝置開發測試）。（投影片還挖苦了某遊戲引擎的按安裝收費政策——VM 可以自動大量安裝來灌爆帳單，這正是該政策荒謬之處。）在編譯器情境**以外**聽到「虛擬機」，多半指的是這種！

1.5　Hack VM 的目標與非目標

目標

- 實作**函式呼叫**！（下週）
- 正式的**編譯期記憶體配置**！（下週）
- **多檔案編譯**以支援函式庫。（下週）
- 算術／邏輯運算式的**乾淨程式碼**，如 $(x \,\&\, y) \mid (x \,\&\, z) + 5 > y + y - 42$。
- 比只有 D 暫存器**更多的工作空間**。
- 比組語**更精簡的語法**。
- 記憶體位址與實體記憶體**脫鉤**。

其中許多目標來自同一個源頭：把 Hack VM 建立在**堆疊**之上。

非目標（要到課程最後用高階語言 Jack 才實現）：人類友善的程式碼（變數名、for/while 迴圈——VM 程式碼本來就該由編譯器產生）；運算式的直接寫法；具型別的變數（字串、陣列）；抽象資料型別（C 的 struct）；執行期記憶體配置（C 的 `malloc`）。

本章重點

- IR 把編譯切成前端（語言相依）／中端（皆不相依）／後端（架構相依）；新語言只寫前端、新架構只改後端。
- LLVM（通用）、GIMPLE（GCC）、JVM（直接執行 IR，可攜但慢）。
- VM = 被模擬的電腦；一詞兩義（設計 vs 執行實例）。
- Hack VM 是堆疊機：函式呼叫、記憶體配置、運算式編譯等目標都源自堆疊。

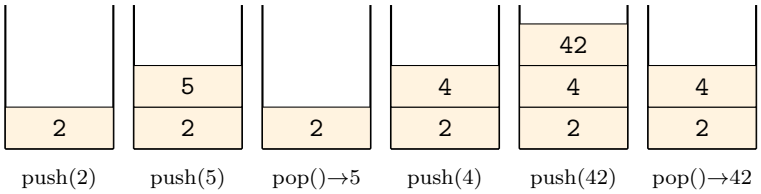
2　Hack VM I：結構、算術與邏輯

2.1　堆疊機與堆疊複習

堆疊機（stack machine）

Hack VM 是**堆疊機**的例子：由一個**堆疊**取代暫存器進行算術／邏輯運算，記憶體**只**用於儲存。堆疊支援的操作（COMSM1201）：`create()` 建立新堆疊；`push(x)` 把 x 加到堆疊**頂**；`pop()` 移除**最近加入**的資料並回傳。堆疊是 **LIFO**：「後進先出（Last In, First Out）」。

追蹤 `push(2)`、`push(5)`、`pop()`、`push(4)`、`push(42)`、`pop()`：



（JVM 也是堆疊機，所以這樣做有充分的現實理由！）

延伸補充：堆疊機 vs 暫存器機。堆疊機的優點：程式碼**更小**（運算元隱含在順序中，不用編碼暫存器編號）、編譯器**容易產生程式碼**（見 2.6 的 RPN）、驗證簡單。缺點：同一計算需要**更多條指令**（值要先搬上堆疊），直譯執行時較慢——研究顯示暫存器式 VM 平均少執行約 47% 的指令。現實中兩者並存：JVM 與 **WebAssembly** 是堆疊機（Wasm 另配有類似暫存器的 locals）、Lua 與 Android 曾用的 Dalvik 是暫存器機；高效能執行環境常在內部把堆疊碼轉成暫存器形式再跑。

2.2　虛擬記憶體

在組語中我們用**實體記憶體**——每個位址就是送往實體晶片上實體閂鎖的邏輯訊號（ROM 或 RAM）。IR 應該**可攜**，所以改用**虛擬記憶體**：行為像實體記憶體（每個位址存一個字），但**不管**每個位址實際存放在哪裡。（ISA 相依的）VM 翻譯器會在編譯時把虛擬位址映射到實體位址。

延伸（投影片註腳）：現代電腦中虛擬記憶體還有第二個更重要的角色：每個行程的組語程式碼都假設自己是唯一在跑的行程、能存取任何位址——這其實就是虛擬記憶體。作業系統用專用機器碼指令維護**分頁表（page table）**把每個行程的虛擬位址空間映射回實體記憶體；甚至可能不映射到 RAM——少用的資料會被換出到**分頁檔（page file）**。

Hack VM 有 **8 個**獨立的虛擬記憶體庫，VM 翻譯器會把它們映射到底層 RAM 的不同**段（segment，連續區塊）**。本章先看兩個：

- `local`：區域變數的通用儲存空間；
- `constant`：在每個 15 位元位址 i 存放常數 i。這塊「記憶體」**唯讀**，不對應任何實體 ROM 或 RAM ——它是把常數帶上堆疊的語法手段。

2.3　語法：push 與 pop

記憶體存取語法

堆疊只有一個，所以不必 `create()`。

- `push [memory] [address]`：把該虛擬記憶體位址的值**推上**堆疊；
- `pop [memory] [address]`：彈出堆疊頂的值並**存入**該位址。（單獨的 `pop` **不是**合法語法。）

push 與 pop 是**唯一**的記憶體管理手段。例如把 `local 4` 複製到 `local 250`：`push local 4` 然後 `pop local 250`。

投影片的完整追蹤範例（`local 33..37` 初值 451、1138、0、0、13）：

指令	堆疊（左 = 底）	`local` 的變化
`push local 34`	1138	—
`push local 36`	1138, 0	—
`push local 34`	1138, 0, 1138	—
`push constant 255`	1138, 0, 1138, 255	—
`pop local 35`	1138, 0, 1138	`local 35` ← 255
`pop local 33`	1138, 0	`local 33` ← 1138

2.4　堆疊運算

所有算術運算都**經由堆疊**進行。例如 `add`（無引數）：彈出堆疊頂**兩個**值、相加、把結果推回堆疊。其他運算的做法與語法完全相同。**不能**直接對 `local` 裡的兩個值相加——必須先推上堆疊。

邏輯比較同理：`eq` 彈出兩個值、判斷是否相等、把結果推回。

速查表（考試會提供）

指令	彈出	計算	說明
`add`	2 個值	$x + y$	整數加法
`sub`	2 個值	$x - y$	整數減法
`neg`	1 個值	$-y$	算術取負
`and`	2 個值	$x \,\&\, y$	逐位元 AND
`or`	2 個值	$x \mid y$	逐位元 OR
`not`	1 個值	$\neg y$	逐位元 NOT
`eq`	2 個值	$x == y$	測試相等
`gt`	2 個值	$x > y$	測試大於
`lt`	2 個值	$x < y$	測試小於

- 彈兩個值的運算中，y 是**先彈出**的值、x 是第二個。例：`push constant 3`、`push constant 1`、`sub` 結束時堆疊頂是 2 而**不是** -2。

- 所有算術用 2 的補數，例如 $-x = \neg x + 1$。
- 所有邏輯把 **true 寫成 0xFFFF（65535）、false 寫成 0x0000**——這使得**逐位元運算同時可當邏輯運算用**！

2.5　完整範例：複合邏輯測試

設 `local 3` = 64、`local 5` = 256。要測試 (`local 3` > 42) **and** (`local 3` < `local 5` − 100)：分別測試兩個條件、再 `and` 起來（記住 `and` ／ `or` ／ `not` 既是逐位元也是邏輯運算！）。

```
push local 3       // 64
push constant 42
gt                 // 64>42: true
push local 3       // 64
push local 5       // 256
push constant 100
sub                // 256-100=156
lt                 // 64<156: true
and                // true
```

逐步的堆疊內容（左 = 底）：

指令後	堆疊
`push local 3`	64
`push constant 42`	64, 42
`gt`	65535
`push local 3`	65535, 64
`push local 5`	65535, 64, 256
`push constant 100`	65535, 64, 256, 100
`sub`	65535, 64, 156
`lt`	65535, 65535
`and`	65535

注意「先算 `local 5` − 100 留在堆疊上、再 `lt`」的模式：**子運算式的結果自然地留在堆疊頂**，等著被外層運算取用——這正是堆疊機讓運算式編譯變乾淨的原因。

延伸補充：逆波蘭記法（RPN）。本週 workshop 會示範如何用樹與文法把一般算術運算式轉成「堆疊順序」，即**逆波蘭記法（Reverse Polish Notation）**（非考試範圍）。核心想法：對運算式的語法樹做**後序走訪**（先左子樹、再右子樹、最後根節點），輸出的指令序列就是 VM 碼。例如 $(3+4)\times 5$ 的後序是 3 4 + 5 ×，對應 `push 3`、`push 4`、`add`、`push 5`、`mult`。第 8 章的解析器輸出 CST，本章的堆疊機吃 RPN——編譯器前端與後端就此接上。

本章重點

- Hack VM 是堆疊機：運算全走堆疊、記憶體只做儲存；LIFO。
- 8 個虛擬記憶體段；`constant` 唯讀、不佔實體記憶體。
- `push/pop [memory] [address]` 是唯一的記憶體操作；單獨 `pop` 不合法。
- 二元運算：y = 先彈出、x = 後彈出，算 $x \circ y$。
- true = `0xFFFF`、false = `0x0000` ⇒ 位元運算兼作邏輯運算。
- 運算式編譯 = 語法樹的後序走訪（RPN）。

3 Hack VM II：分支與記憶體

3.1 標籤與 goto

Hack VM 怎麼處理迴圈與條件？跟組語一樣——用跳躍。現在改叫 **goto**，因為它們不再只對應單一一條機器碼指令。

分支語法

- `label LABEL_NAME`：在程式碼該處宣告標籤；
- `goto LABEL_NAME`：從程式任何地方跳到該標籤（嚴格說 goto 與 label 必須在**同一個函式**內——見下週）；
- `if-goto LABEL_NAME`：**彈出**堆疊頂，若結果**非零**（即不是 false）則執行 goto。

`if-goto` 的用法對應組語的 `D;JNE`，差別是：比較對象是**堆疊頂**而不是 D；且語言內建了真正的邏輯運算子 `gt`、`eq`、`lt`、`and`、`or`、`not` 來取代各種跳躍條件——不再需要七種 jump，一種 `if-goto` 搭配比較運算就夠。

3.2 pointer 與 this：執行期定址

用 `local` 只能存取**執行前就知道**的位址，這會造成問題：假設記憶體裡存了一個陣列，想取它的第 `local 0` 個元素——但 `push local (local 0)` **不是**合法 VM 碼！

pointer 與 this

兩個特殊記憶體段。`this` 到實體 RAM 的映射 **不在編譯期固定**，而是**執行期**決定。保證：

$$\texttt{this } i \text{ 映射到 } RAM[(\texttt{pointer 0}) + i] \quad \text{對所有} i.$$

改寫 `pointer 0`，就**整段搬動**了 `this` 的落點——這是 VM 層的「指標解參考」。

用 this 實作陣列

仍需事先決定陣列放在哪段實體記憶體（跟組語一樣）。假設決定放在 $RAM[\texttt{0x0800}]$–$RAM[\texttt{0x08FF}]$，要取第 `local 0` 個元素（從 0 數起）：

```
push constant 2048  // 0x0800
push local 0
add                 // 堆疊：(local 0) + 0x0800
pop pointer 0       // this i 現在映射到 (local 0)+0x0800+i
push this 0         // 陣列第 (local 0) 個元素上堆疊
```

當然，若 `local 0` ≥ 256 就出界了！高階語言將自動處理這種記憶體配置，但現在得手動來。（投影片原話：Life is suffering.）

3.3 I/O 與 that

I/O 可以用同一招存取螢幕 $RAM[\texttt{0x4000–0x5FFF}]$ 與鍵盤 $RAM[\texttt{0x6000}]$……對吧？可以，但惱人的是**不能用 this**。之後會看到 this 在編譯抽象資料型別（如 C 的 struct）時有特殊角色，所以不能映射到任意記憶體段（第 11 週就懂了）。目前只需知道：

this vs that

- this 只能存取 $RAM[\texttt{0x0800–0x3FFF}]$；
- 這範圍之外（例如螢幕、鍵盤）必須用 that。that 的行為幾乎與 this 相同，僅有兩點不同：映射基準是 pointer 1（而非 pointer 0）；可存取**任何**實體 RAM 位址。

3.4 八個虛擬記憶體段

已見過 local、constant、this、that、pointer。剩下三個：

- argument：函式呼叫開始時**持有該次呼叫的引數**，不可寫入。（詳見下週。）
- static：內容**跨函式呼叫持續存在**（之後用於高階語言的 static 與全域變數）。
- temp：行為與 local 完全相同但小得多，作為高階語言編譯器編譯**單一指令**時的「工作空間」，不用打擾 local 的內容。

（對照：local 在函式呼叫開始時是**空的**，函式返回時內容被**丟棄**。）

投影片同時把 fill.asm（按鍵時全螢幕變白、否則全黑）改寫成 fill.vm 做對照——用 that 掃描螢幕記憶體、用 if-goto 做迴圈，行數遠少於組語版。

本章重點

- label ／ goto ／ if-goto（彈堆疊、非零則跳）取代組語的七種 jump。
- this $\quad i \mapsto RAM[\texttt{pointer 0} + i]$：執行期定址，陣列的基礎；pop pointer 0 = 搬動整段。
- this 限 0x0800–0x3FFF；螢幕鍵盤等其他位址用 that（基準 pointer 1）。
- 八段：local（呼叫時清空）、argument （持引數、唯讀）、static（跨呼叫持續）、temp（小型工作區）、constant（唯讀常數）、this ／ that ／ pointer。

4　實作 Hack VM 翻譯器

4.1　Hack VM 的權杖

類別	內容
關鍵字	`push`、`pop`；`add`、`sub`、`neg`、`and`、`or`、`not`、`eq`、`gt`、`lt`；`local`、`constant`、`this`、`that`、`pointer`、`argument`、`static`、`temp`；`label`、`goto`、`if-goto`；`function`、`call`、`return`（下週！）
整數字面值	十進位 0…32767
識別字	無空白、非關鍵字、字母開頭
換行	

沒有新東西——你已經知道詞法器怎麼運作，所以這部分作業已經幫你寫好了。組譯器在詞法階段建了標籤符號表；這裡**不需要**（除非想做好錯誤處理，而我們不想）。為什麼？因為 VM 的 `label X` 可以**直接翻譯**成組語的 `(X)`、`goto X` 翻成 `@X` + `0;JMP` ——把「解析標籤位址」的工作**下放給組譯器**就好，VM 翻譯器根本不必知道標籤在哪。

4.2　Hack VM 的文法

Hack VM 的文法有很多種寫法，以下是其中一種：

Hack VM 文法

⟨*instruction*⟩ ::= (‘`push`’, ⟨*data*⟩ | ‘`pop`’, ⟨*data*⟩
| ‘`add`’ | ‘`sub`’ | ‘`neg`’ | ‘`and`’ | ‘`or`’ | ‘`not`’ | ‘`eq`’ | ‘`gt`’ | ‘`lt`’
| ‘`label`’, identifier | ‘`goto`’, identifier | ‘`if-goto`’, identifier
| ‘`function`’, identifier, integerLiteral
| ‘`call`’, identifier, integerLiteral | ‘`return`’), newline

⟨*data*⟩ ::= (‘`local`’ | ‘`constant`’ | ‘`this`’ | ‘`that`’ | ‘`pointer`’
| ‘`argument`’ | ‘`static`’ | ‘`temp`’), integerLiteral

這其實比 Hack 組語**簡單得多**——它是 **LL(1)** 文法：每個 ⟨*instruction*⟩ 怎麼解析，看**第一個權杖**就知道。（對照：組語因為 `DM=M+D` 需要 LL(2)。）**難的部分其實是把 VM 指令翻譯成組語！**

4.3　配置記憶體：基底位址與位移

把虛擬記憶體映射到實體記憶體（**配置記憶體**）的一般做法與 `this`/`that` 相同。假設要把 `local` 映射到實體記憶體：選一個 RAM 位址（例如 300），然後 `local 0` $\mapsto$ $RAM[300]$、`local 1` $\mapsto$ $RAM[301]$、……、`local` $k \mapsto RAM[300+k]$。

基底位址與位移

300 稱為**基底位址 (base address)**、`local` 後面的數字稱為**位移 (offset)**：每個虛擬位址映射到**基底位址＋位移**。例如 `pointer 0` 就是 `this` 的基底位址、`pointer 1` 是 `that` 的基底位址。

4.4 硬體限制與段長度

Hack VM 的八個段各有 64KB 記憶體（32,768 個 16 位元字），堆疊還可以無限高；但 Hack CPU 總共只支援 64KB——**總得有取捨**。（這不是 Hack VM 特有的：多數計算模型都假設無限儲存——圖靈機有無限長的紙帶、C 程式對 `malloc` 總量沒有硬上限。）

解法：給每個段一個**長度**，禁止存取超過長度的區域。例如給 `local` 長度 5，就只允許 `local 0` 到 `local 4`；「`local 5` 的位置」就能拿去當（例如）`argument` 的基底位址。

現代系統嚴格執行這件事：行程存取配置段以外的位址時，硬體產生**中斷**——通常導致 **segmentation fault** 崩潰（若該段是堆疊，錯誤就叫 **stack overflow**）。這需要 Hack 沒有的硬體支援，軟體模擬又太慢，所以我們只是**手動追蹤**段長度。

C 的記憶體配置對照

C 的 `long long` 是 64 位元整數，在 64 位元電腦上是一個字。

- `long long myArray[128];` 配置 128 個字的段，把**基底位址**存進 `long long *` 變數 `myArray`。
- `myArray[50]` 回傳位址 `myArray` +50 的值——第 51 個元素。**陣列索引就是基底＋位移!**
- `myArray[128]` 試圖存取段外（超出一個字）⇒ segmentation fault。
- 其他型別按資料大小折算：`short *myArray = malloc(64*sizeof(short));` 配置 16 個字、每字塞四個 16 位元 short；`myArray[50]` 取位址 `myArray`+12 處的第三個 short。現代 ISA 能高效操作這種佈局。（CLion 等 IDE 有記憶體與反組譯視圖，可以自己寫測試碼觀察!）

4.5 Hack 的記憶體配置慣例

這些細節是**我們這個 Hack VM 實作**特有的（不是 Hack VM 本身的一部分，但 Nand2tetris 的 VM 模擬器假設它們）。組語那些還沒解釋的神祕關鍵字（除了 `SP`）全都是**存放段基底位址的變數**：

- `local` 的基底位址存在 $RAM[1]$ = `LCL`；
- `argument` 的基底位址存在 $RAM[2]$ = `ARG`；
- `pointer` 固定配置在基底 3、長度 2：`pointer 0`（= `this` 的基底）就是 $RAM[3]$ = `THIS`；`pointer 1`（= `that` 的基底）就是 $RAM[4]$ = `THAT`；

- `temp` 固定配置在基底 5、長度 8；
- `static` 固定配置在基底 16、長度 240。編譯 `Foo.vm` 時，`static 5` 應映射到組語變數 `Foo.5`（解釋見下週！）；
- `constant` 不出現在實體記憶體。

目前假設 $RAM[1\text{–}4]$ 在執行開始時已被初始化成合理的值（提供的測試腳本會做這件事！）。

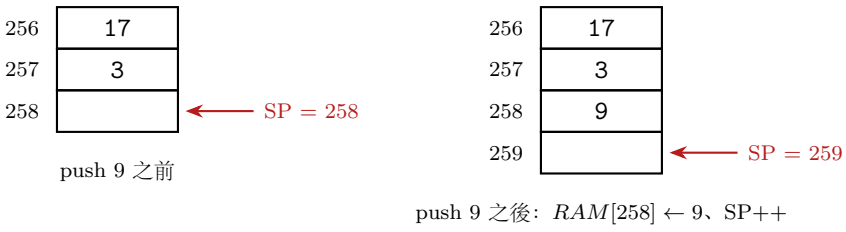
4.6　實作堆疊

堆疊也配置固定段：**基底 256、長度 1792**。**堆疊頂再往上一格**的位址存在 $RAM[0] = \texttt{SP}$（Stack Pointer）。

push 與 pop 的機制

- **push** x：把 x 寫入 $RAM[SP]$，然後 `SP++`；
- **pop 到** $RAM[i]$：先 `SP--`，再把 $RAM[SP]$ 複製到 $RAM[i]$。

注意 pop 時**不必清零** $RAM[SP]$——整段都保留給堆疊，下次用到時自然會被覆寫。



push 9 之前　　push 9 之後：$RAM[258] \leftarrow 9$、SP++

嚴格說 `local` 與 `argument` 也會存成堆疊的 **子段**——但這週先不用擔心！

4.7　延伸補充：VM 指令的組語翻譯範例

投影片說「難的是翻譯本身」但把細節留給作業。以下是三個代表性翻譯（標準 Nand2tetris 做法），展示每條 VM 指令如何展開成一段組語：

翻譯範例

(1) `push constant 7`——常數上堆疊：

```
@7
D=A       // D = 7
@SP
A=M       // A = SP（堆疊頂上一格）
M=D       // RAM[SP] = 7
@SP
M=M+1     // SP++
```

(2) `pop local 2`——彈出存入 `local 2`（用 `R13` 暫存目標位址）：

```
@LCL
D=M       // D = local 的基底位址
@2
D=D+A     // D = 基底 + 2
@R13
M=D       // R13 = 目標位址
@SP
M=M-1     // SP--
A=M
D=M       // D = 彈出的值
@R13
A=M
M=D       // RAM[基底+2] = 值
```

(3) `add`——彈兩個、加、推回（實作上直接原地改寫，少一次 push）：

```
@SP
M=M-1     // SP--
A=M
D=M       // D = y（先彈出）
@SP
M=M-1     // SP--
A=M
M=D+M     // RAM[SP] = x + y（結果原地放回）
@SP
M=M+1     // SP++
```

一行 VM 碼展開成 7–13 行組語——這就是 IR 「更精簡的語法」目標的具體體現，也解釋了為什麼 `goto` 不再對應單一機器碼指令。

4.8 Hack VM 記憶體總覽

記憶體映射總表（考試會提供）

關鍵字	位址	用途
SP	0	[堆疊頂值的位址] +1
LCL	1	存 local 段的基底位址
ARG	2	存 argument 段的基底位址
THIS	3	pointer 0（this 段的基底位址）
THAT	4	pointer 1（that 段的基底位址）
R5–R12	5–12	temp 段（最大 8）
R13–R15	13–15	VM 翻譯器的暫時變數（需要時用）
—	16–255	static 段（最大 240）
—	256–2047	保留給堆疊，含 local 與 argument 段（合計最大 1792）
—	2048–16383	「堆積（heap）」記憶體，可配置給 this 或 that 段
SCREEN	16384–24575	記憶體映射的螢幕輸出
KBD	24576	記憶體映射的鍵盤輸入

第五週的「神祕關鍵字」至此全部解密：SP、LCL、ARG、THIS、THAT 是 VM 實作的基礎設施，R0–R15 中 5–12 給 temp、13–15 給翻譯器自用。

本章重點

- VM 翻譯器不需要標籤符號表：標籤直接翻成組語標籤，位址解析交給組譯器。
- Hack VM 文法是 LL(1)（第一個權杖決定一切）；難點在**翻譯**不在解析。
- 虛擬位址 = 基底位址 + 位移；段有長度、越界即 segfault（堆疊段則是 stack overflow）。
- 慣例：SP=0、LCL=1、ARG=2、THIS=3、THAT=4、temp=5–12、翻譯器暫存 =13–15、static=16–255、堆疊 =256–2047、堆積 =2048–16383。
- push：寫 $RAM[SP]$ 再 SP++；pop：SP-- 再讀 $RAM[SP]$；不必清零。

5　綜合練習題（附詳解）

練習 1：堆疊追蹤

初始堆疊為空，`local 0` = 10、`local 1` = 20。逐步寫出每條指令後的堆疊與 `local` 內容：

```
push local 0
push local 1
push local 0
pop local 1
add
pop local 0
```

解答

（堆疊左 = 底）

指令後	堆疊	`local`
`push local 0`	10	0:10, 1:20
`push local 1`	10, 20	0:10, 1:20
`push local 0`	10, 20, 10	0:10, 1:20
`pop local 1`	10, 20	0:10, 1:**10**
`add`	30	0:10, 1:10
`pop local 0`	（空）	0:**30**, 1:10

最終：`local 0` = 30、`local 1` = 10。

練習 2：運算元順序

(a) `push constant 3`、`push constant 1`、`sub` 之後堆疊頂是什麼？(b) 要計算 5 − `local 2`，指令順序怎麼寫？(c) `neg` 與 `not` 對值 3 各得到什麼（16 位元）？

解答

(a) $y = 1$（先彈出）、$x = 3$，`sub` 算 $x - y = \mathbf{2}$（不是 −2）。口訣：**堆疊順序 = 運算式由左至右的順序**。

(b) 先推被減數、再推減數：`push constant 5`、`push local 2`、`sub`。

(c) `neg`：−3 的 2 的補數 = `0xFFFD` = 65533。`not`：¬3 = ¬`0x0003` = `0xFFFC` = 65532。驗證關係 $-x = \neg x + 1$：65532 + 1 = 65533。✓

練習 3：運算式編譯

寫出計算 (`local 0` + `local 1`) & (`local 0` − 7) 並把結果存入 `local 2` 的 VM 碼。

解答

後序走訪：先算左子式、再算右子式、最後套運算子。

```
push local 0
push local 1
add             // local 0 + local 1
push local 0
push constant 7
sub             // local 0 - 7
and             // 兩者逐位元 AND
pop local 2
```

注意兩個子運算式的結果依序疊在堆疊上，`and` 恰好取到它們。

練習 4：true 與 false 的表示

(a) 為什麼 Hack VM 用 `0xFFFF` 表示 true 而不是 1？(b) 設堆疊頂是 `gt` 的結果，接著執行 `not`。這在邏輯上等價於什麼比較？(c) 若某語言用 1 當 true，`and` 還能兼作邏輯運算嗎？

解答

(a) 因為 `0xFFFF` 是**全 1**：這使逐位元運算 **同時就是**邏輯運算——`0xFFFF` & `0xFFFF` = `0xFFFF`（true and true = true）、`0xFFFF` & `0x0000` = `0x0000`，不需要另設一套邏輯指令。（另一個觀點：`0xFFFF` 是 2 的補數的 $-1 = \neg 0$，所以 `not false = true` 也在位元層面自動成立。）

(b) $\neg(x > y) \equiv x \leq y$——這就是 VM 只需要 `gt`/`eq`/`lt` 三種比較的原因：其餘三種（$\geq$、$\neq$、$\leq$）各補一個 `not` 就有了。

(c) 不行。位元運算上 $1 \,\&\, 2 = 0$，但「true and true」應為 true——用 1 當 true 時位元 AND 與邏輯 AND 不一致（C 因此需要 `&` 與 `&&` 兩種運算子！）。

練習 5：RPN 轉換

把運算式 $((a + b) - c) > (d \mid e)$ 轉成 VM 碼，其中 a–e 分別是 `local 0`–`local 4`。

解答

語法樹後序走訪：$a\ b\ +\ c\ -\ d\ e\ \mid\ >$。

```
push local 0    // a
push local 1    // b
add
push local 2    // c
sub             // (a+b)-c
push local 3    // d
push local 4    // e
or              // d|e
gt              // ((a+b)-c) > (d|e)
```

檢查運算元順序：gt 彈出 $y = d|e$（後推的）、$x = (a + b) - c$（先推的），計算 $x > y$。✓

練習 6：迴圈翻譯

把下列 C 程式翻成 Hack VM（i 用 local 0、sum 用 local 1）：

```
sum = 0;
for (i = 10; i > 0; i--) { sum += i; }
```

解答

```
push constant 0
pop local 1         // sum = 0
push constant 10
pop local 0         // i = 10
label LOOP
push local 0
push constant 0
gt                  // i > 0 ?
not                 // 取反：i <= 0 ?
if-goto END         // 條件不成立就離開
push local 1
push local 0
add
pop local 1         // sum += i
push local 0
push constant 1
sub
pop local 0         // i--
goto LOOP
label END
```

另一種寫法是把測試放在迴圈尾（if-goto LOOP 直接用 i > 0 的結果，不需 not），少一次跳躍：兩種都正確。注意 if-goto **彈出**比較結果——堆疊在每輪迴圈開始時保持乾淨。

練習 7：陣列存取

陣列存放在 $RAM[0x0900]$ 起。寫 VM 碼完成 arr[local 2] = arr[local 2] + 1（假設不出界）。

解答

```
push constant 2304   // 0x0900
push local 2
add
pop pointer 0        // this 0 -> arr[local 2]
```

```
push this 0
push constant 1
add
pop this 0            // 寫回同一位置
```

關鍵：`pop pointer 0` 之後 `this 0` 就**指著**目標元素，讀（`push this 0`）與寫（`pop this 0`）都不必重算位址。`0x0900` = 2304 在 `this` 允許的 `0x0800`–`0x3FFF` 範圍內。✓

練習 8：this vs that

(a) 為什麼把螢幕塗黑的程式必須用 `that` 而不能用 `this`？(b) 寫 VM 碼把螢幕的第一個字（$RAM[16384]$）設為全 1（-1）。

解答

(a) 螢幕在 RAM[`0x4000`–`0x5FFF`]（16384–24575），超出 `this` 允許的 `0x0800`–`0x3FFF`。`this` 被保留給之後編譯抽象資料型別（struct）用，映射範圍受限；`that`（基準 `pointer 1`）可指向任何 RAM 位址。

(b)

```
push constant 16384
pop pointer 1         // that 0 -> RAM[16384]
push constant 0
not                   // 0xFFFF (-1, 全 1)
pop that 0            // 螢幕第一個字 = 全黑
```

（`push constant` 只能推 0–32767，所以「-1」用 `push constant 0` + `not` 或 `push constant 1` + `neg` 製造。）

練習 9：基底與位移

某次執行中 $RAM[1] = 317$、$RAM[2] = 310$、$RAM[3] = 3000$、$RAM[4] = 16384$。求下列虛擬位址對應的實體位址：(a) `local 4`；(b) `argument 1`；(c) `this 12`；(d) `that 100`；(e) `temp 3`；(f) `pointer 1`。

解答

(a) `LCL` + 4 = 317 + 4 = **321**。(b) `ARG` + 1 = 310 + 1 = **311**。(c) `THIS` + 12 = 3000 + 12 = **3012**。(d) `THAT` + 100 = 16384 + 100 = **16484**（螢幕記憶體內——正在畫圖！）。(e) `temp` 基底**固定**為 5：5 + 3 = **8**（不查 RAM——這是編譯期常數）。(f) `pointer` 基底固定為 3：3 + 1 = **4**（也就是 `THAT` 本身）。注意 (e)(f) 與 (a)–(d) 的本質差異：固定段的位址在**編譯期**就知道，`local` 等段要在**執行期**讀基底變數。

練習 10：記憶體映射

(a) 編譯 `Foo.vm` 時 `static 5` 映射到哪裡？為什麼用組語變數而不是固定位址？(b) `R13`–`R15` 的用途是什麼？在練習哪一題的翻譯範例中出現過？(c) 堆疊段最多能長到哪？再長會發生什麼事？

解答

(a) 映射到組語變數 `Foo.5`——由**組譯器**從 16 起自動配置 RAM（恰好落在 static 段 16–255 內）。用變數而非固定位址的好處：**多檔案編譯**時 `Foo.vm` 的 `static 5` 與 `Bar.vm` 的 `static 5` 自動得到 **不同**位址（`Foo.5` vs `Bar.5`），不會互相踩踏（詳見下週）。

(b) VM 翻譯器的**暫時變數**。`pop local 2` 的翻譯範例用了 `R13` 暫存目標位址——因為計算「基底 + 位移」和「彈出值」都需要經過 D，得有地方寄放其中一個。

(c) 堆疊段是 256–2047（長度 1792，還要容納 `local` 與 `argument` 子段）。超過就是 **stack overflow**——現代系統靠硬體中斷抓到它，Hack 沒有這種硬體支援，所以只能靠手動追蹤（或默默覆寫 heap 資料，產生詭異 bug）。

練習 11：SP 機制

初始 $SP = 258$、$RAM[256] = 17$、$RAM[257] = 3$。逐步寫出執行 `push constant 9`、`add`、`pop local 0`（設 `LCL` $= 300$）後 SP 與相關 RAM 的值。

解答

push constant 9：$RAM[258] \leftarrow 9$、$SP = 259$。堆疊（256 起）：17, 3, 9。

add：$SP = 258$，$D \leftarrow RAM[258] = 9$（$y$）；$SP = 257$，$RAM[257] \leftarrow 3 + 9 = 12$；$SP = 258$。堆疊：17, 12。（$RAM[258]$ 仍殘留 9，但已在堆疊外——不必清零，下次 push 會覆寫。）

pop local 0：$SP = 257$, $RAM[300] \leftarrow RAM[257] = 12$。堆疊：17。最終 $SP = 257$、$RAM[300] = 12$、$RAM[257]$ 殘留 12（無害）。

練習 12：端到端翻譯

把 VM 指令 `push temp 3` 翻譯成 Hack 組語。提示：`temp` 是**固定段**。

解答

`temp` 基底固定為 5，所以 `temp 3` 就是 $RAM[8]$——位址是**編譯期常數**，不必像 `local` 那樣執行期讀 `LCL`：

```
@8
D=M      // D = RAM[8]（temp 3 的值）
@SP
A=M
M=D      // RAM[SP] = D
@SP
M=M+1    // SP++
```

對照 `push local 3` 的版本開頭得是 `@LCL / D=M / @3 / A=D+A / D=M`（執行期算位址）——固定段少了兩條指令。這種「編譯期能算就不留到執行期」正是後端最佳化的雛形。

A 附錄：速查表

A.1 Hack VM 指令總表

類別	指令	語意
記憶體	`push [seg] [i]`	把 `seg i` 的值推上堆疊
	`pop [seg] [i]`	彈出堆疊頂、存入 `seg i`
算術	`add` ／ `sub` ／ `neg`	$x+y$ ／ $x-y$ ／ $-y$
邏輯	`and` ／ `or` ／ `not`	逐位元 & ／ | ／ ¬（兼作邏輯運算）
比較	`eq` ／ `gt` ／ `lt`	$x==y$ ／ $x>y$ ／ $x<y$，結果 `0xFFFF`/`0x0000`
分支	`label X`	宣告標籤
	`goto X`	無條件跳躍
	`if-goto X`	彈出堆疊頂，非零則跳
函式	`function` ／ `call` ／ `return`	下週！

二元運算：y = 先彈出、x = 後彈出，計算 $x \circ y$。

A.2 八個虛擬記憶體段

段	基底	長度	性質
`constant`	—	—	唯讀；`constant` i 的值就是 i
`local`	$RAM[1]$=`LCL`	動態	區域變數；呼叫時清空
`argument`	$RAM[2]$=`ARG`	動態	函式引數；唯讀
`this`	$RAM[3]$=`THIS`	動態	限 `0x0800`–`0x3FFF`；供 struct 用
`that`	$RAM[4]$=`THAT`	動態	可指任何 RAM 位址
`pointer`	3（固定）	2	`pointer 0`=THIS、`pointer 1`=THAT
`temp`	5（固定）	8	編譯器工作空間
`static`	16（固定）	240	跨呼叫持續；`Foo.vm` 的 `static` i → 組語變數 `Foo.`i

A.3 記憶體映射總表

位址	關鍵字	用途
0	`SP`	堆疊頂上一格的位址
1	`LCL`	`local` 基底
2	`ARG`	`argument` 基底
3	`THIS`	`this` 基底（`pointer 0`）
4	`THAT`	`that` 基底（`pointer 1`）
5–12	`R5`–`R12`	`temp` 段
13–15	`R13`–`R15`	VM 翻譯器暫存
16–255	—	`static` 段
256–2047	—	堆疊（含 `local`/`argument`）
2048–16383	—	堆積（給 `this`/`that`）
16384–24575	`SCREEN`	螢幕
24576	`KBD`	鍵盤

A.4 名詞中英對照

英文	中文	英文	中文
intermediate representation	中介表示法	base address	基底位址
virtual machine	虛擬機	offset	位移
VM translator	VM 翻譯器	segment	段
front / middle / back end	前端／中端／後端	segmentation fault	記憶體區段錯誤
stack machine	堆疊機	stack overflow	堆疊溢位
LIFO	後進先出	stack pointer (SP)	堆疊指標
virtual memory	虛擬記憶體	heap	堆積
page table / page file	分頁表／分頁檔	Reverse Polish Notation	逆波蘭記法
memory allocation	記憶體配置	postorder traversal	後序走訪

A.5 參考資料

- John Lapinskas, *Virtual machines and intermediate representations ／ The Hack VM I & II ／ Implementing the Hack VM translator*（9-1 至 9-4），COMSM1302, University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 7–8（Virtual Machine I & II）.
- LLVM Project, *LLVM Language Reference Manual*（真實世界 IR 的規格範例）.
- Shi, Casey, Ertl & Gregg, *Virtual Machine Showdown: Stack versus Registers*, ACM TOPLAS, 2008（堆疊機 vs 暫存器機的實測研究）.
- WebAssembly Design Rationale（現代堆疊機 IR 的設計取捨）.