COMSM1302 電腦架構概論

Week 10 作業:A Hack VM Translator（Phase 2）完整詳解

`function` · `return` · `call` · 資料夾編譯與開機碼

完整實作指南 (含三條 sprintf 的參考組語與四個測試的走讀)

2026 年 7 月

本作業的任務：完成上週開工的 Hack VM 翻譯器（函式呼叫三兄弟 `function` ／ `return` ／ `call`，加上整個資料夾的編譯與開機碼），並用提供的四組測試腳本驗證。前置條件：第九週的 `StackTest.vm` 必須能通過（第九週其餘部分沒完成沒關係）。**需要的軟體：**nand2tetris 的 VM 模擬器與 CPU 模擬器（需要 Java Runtime Environment）。

目錄

1　開工前：新骨架的兩個變更

題目

本週建議直接用新骨架，而不是沿用上週的程式碼。若堅持搬舊碼，要注意哪兩件事？為什麼？

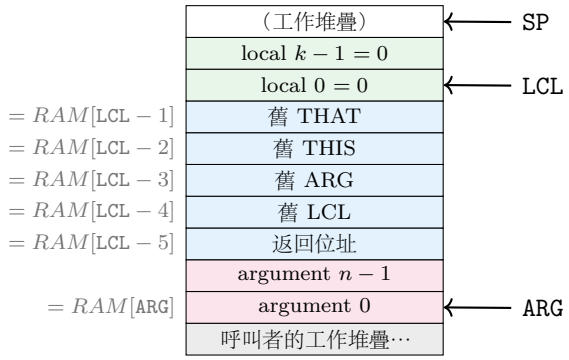
解答

1. **兩個名字被改掉了**：`THIS` 這個 enum 值改名 `KW_THIS`、`TokenType` 這個 struct 改名 `HackTokenType`——因為它們與 Windows 上資料夾掃描程式碼需要引入的系統標頭**名稱衝突**。不改名，`list_vm_files`（第 5 節要用的檔案掃描函式）無法編譯。這是 C 的經典痛點：沒有命名空間，所有識別字共享一個全域池，跟系統標頭撞名只能自己讓路。

2. **`parse_file` 裡把 SP 初始化成 256 的程式碼被註解掉了**——因為本週部分測試腳本會**預先佈置一個非空的堆疊**來模擬「之前已經發生過函式呼叫」（Test 1、Test 2 都是這樣）。翻譯器若強行把 SP 蓋回 256，測試環境就被破壞了。SP 的初始化從此屬於**開機碼**（bootstrap）——第 5 節資料夾編譯時才加回來，位置完全不同。

2　呼叫框架回顧（寫程式前的地圖）

三條指令生成的組語都圍繞同一個資料結構——**呼叫框架（call frame）**。題目明說：「你應假設框架在堆疊上的存放順序**與講義完全一致**，測試腳本才會通過。」先把這張地圖釘在牆上：



三條指令對這張圖的責任分工：

VM 指令	職責
`call f n`	推返回位址與四個舊暫存器（**恰好照這個順序**）→ $ARG = SP-5-n$ → $LCL = SP$ → 跳去 `f`
`function f k`	放函式入口標籤 → $SP = LCL + k$ 並把 local $0..k-1$ 清零
`return`	救出返回位址（$RAM[LCL-5]$）→ 返回值寫入 $RAM[ARG]$ → $SP = ARG+1$ → 由 $LCL-1$ 往下恢復 THAT ／ THIS ／ ARG ／ LCL → 跳返回位址

跟上週一樣，每個函式只需填一條 `sprintf`。骨架另外提供兩個標籤工具：`get_next_label_name(abc, xyz)` 為檔名 `abc` 生成一個**沒用過**的組語標籤（給 `call` 的返回

位址用）；新的 `get_function_label(abc, xyz)` 為函式名 `abc` 生成**標準形式**的函式入口標籤（給 `function` 與 `call` 共用——兩邊必須產生同一個名字，跳躍才接得上）。

3　parse_function 與 parse_return（Test 1）

3.1　parse_function

題目

補完 `parse_function` 的 `sprintf: function f k` 應輸出什麼組語?

解答

兩件事：放入口標籤；騰出 k 格 local 並**清零**。進入函式時 `call` 已把 LCL 設成堆疊頂，所以 local 區從 LCL 開始、SP 要推進到 $LCL + k$：

```
(Foo.bar)          // get_function_label("Foo.bar", ...) 的結果
    @LCL           // SP = LCL（從 local 0 開始鋪）
    D=M
    @SP
    M=D
    @k             // D = k（sprintf 用 %d 填入）
    D=A
(Foo.bar$ZERO)     // 迴圈：還有 local 沒清零嗎?
    @Foo.bar$DONE
    D;JEQ          // D == 0 -> 鋪完了
    @SP
    A=M
    M=0            // RAM[SP] = 0
    @SP
    M=M+1          // SP++
    D=D-1
    @Foo.bar$ZERO
    0;JMP
(Foo.bar$DONE)
```

兩個迴圈標籤用 `get_next_label_name` 生成（保證不重複）。**為什麼用執行期迴圈而不是展開 k 次 `push constant 0`?** 都可以——展開版每個 local 約 5 條指令、迴圈版固定約 13 條；$k \le 2$ 時展開比較短，k 大時迴圈比較短。用迴圈的另一個理由很現實：題目要求**一條 `sprintf`**，迴圈版的格式字串是固定形狀（三個標籤與一個 `%d`），展開版則需要 C 端再套一層迴圈。

為什麼 local 必須清零? 這是 VM 規格：高階語言假設區域變數初始值為 0。忘了清零，SimpleFunction 會算錯（它開頭就 `push local 0`）。

3.2　parse_return

題目

補完 parse_return 的 sprintf。

解答

照第 2 節地圖按部就班，**順序是規格**（理由在後）：

```
// 1. R13 = RAM[LCL-5]（先救返回位址！）
    @5
    D=A
    @LCL
    A=M-D           // A = LCL-5
    D=M
    @R13
    M=D
// 2. RAM[ARG] = 返回值（堆疊頂）
    @SP
    AM=M-1          // SP--, A = SP
    D=M             // D = 返回值
    @ARG
    A=M
    M=D
// 3. SP = ARG + 1（呼叫者視角：n 個引數換成 1 個返回值）
    @ARG
    D=M+1
    @SP
    M=D
// 4. 由 LCL-1 往下恢復四個暫存器（LCL 本身當游標用）
    @LCL
    AM=M-1          // LCL--, A = 原LCL-1
    D=M
    @THAT
    M=D             // THAT = RAM[原LCL-1]
    @LCL
    AM=M-1
    D=M
    @THIS
    M=D             // THIS = RAM[原LCL-2]
    @LCL
    AM=M-1
    D=M
    @ARG
    M=D             // ARG = RAM[原LCL-3]
    @LCL
```

```
    A=M-1
    D=M
    @LCL
    M=D             // LCL = RAM[原LCL-4]（游標功成身退）
// 5. 跳返回位址
    @R13
    A=M
    0;JMP
```

三個「為什麼是這個順序」：

1. **返回位址必須第一個救**（存進翻譯器保留的暫存器 `R13`，即 RAM[13]）。致命陷阱：當 $n = 0$ 時 $ARG = SP - 5 - 0$ 恰好指向 **返回位址那一格**！第 2 步「返回值寫入 $RAM[ARG]$」會把它**覆寫**。不先備份，無引數的函式一律跳到外太空——NestedCall 與 FibonacciElement（`call Sys.init 0`、`call Main.fibonacci 1` 的遞迴底層）就是抓這個 bug 的。
2. $SP = ARG + 1$ **要在恢復 ARG 之前**——用的是**被呼叫者**框架裡的 ARG；恢復之後 ARG 就變成呼叫者的了。
3. **恢復順序由高往低**（THAT → THIS → ARG → LCL）：讓 `AM=M-1` 把 LCL 自身當**遞減游標**用，省掉一個暫存器；LCL 最後一個恢復，游標剛好功成身退。（另一常見寫法是先 `R14 = LCL` 再用 R14 遊走，多幾條指令但語意更直白——兩者都對。）

3.3　Test 1：SimpleFunction 走讀

題目

用 `SimpleFunction.vm`（測試資料 Test 1）驗證。VM 檔裡**沒有** `call`——測試腳本直接把堆疊佈置成「`call SimpleFunction.test 2`（返回位址 9）剛發生完」的樣子。輸出的 `.asm` 必須跟測試腳本、`.cmp` 檔放同一個資料夾。

解答

測試腳本佈置的世界（這正是第 1 節說「不能在 `parse_file` 裡重設 SP」的原因）：

暫存器	值	RAM	值
SP	317	RAM[310]	1234（argument 0）
LCL	317	RAM[311]	37（argument 1）
ARG	310	RAM[312]	9（返回位址）
THIS	3000	RAM[313]	305(舊 LCL)
THAT	4000	RAM[314]	300(舊 ARG)
		RAM[315]	3010(舊 THIS)
		RAM[316]	4010(舊 THAT)

`SimpleFunction.test` 的本體與手算（`function SimpleFunction.test 2` 先把 RAM[317]、RAM[318] 清零、SP → 319）：

VM 指令	堆疊頂變化	說明
`push local 0`	0	local 0 = 0
`push local 1`	0, 0	
`add`	0	0 + 0
`not`	−1	¬0 = `0xFFFF`
`push argument 0`	−1, 1234	
`add`	1233	
`push argument 1`	1233, 37	
`sub`	**1196**	返回值
`return`	—	見下

`return` 逐步：R13 = $RAM[317-5] = RAM[312] = 9$；返回值 1196 寫入 $RAM[ARG] = RAM[310]$（覆掉 argument 0）；$SP = 310+1 = 311$；恢復 THAT = 4010、THIS = 3010、ARG = 300、LCL = 305；跳位址 9。

`.cmp` 期望的最終狀態——與手算完全一致：

$$RAM[0..4] = 311,\ 305,\ 300,\ 3010,\ 4010; \qquad RAM[310] = 1196$$

除錯建議（題目強烈推薦的流程）：先把 `SimpleFunction.vm` 載入 **VM 模擬器**配同一個測試腳本單步走，看每條 VM 指令對堆疊做了什麼；再對照你生成的組語在 **CPU 模擬器**裡的行為——第一個分歧點就是 bug。

4　parse_call（Test 2）

題目

補完 `parse_call` 的 `sprintf`，用 `NestedCall.vm`（Test 2）驗證。注意：要編譯的檔案叫 `Sys.vm`，但輸出必須叫 `NestedCall.asm` 測試腳本才找得到。

解答

`call f n` 的組語——把第 2 節的框架**推**出來（返回標籤 `RET` 由 `get_next_label_name` 生成，函式標籤由 `get_function_label` 生成）：

```
// 1. 推返回位址（用標籤的 ROM 位址）
    @RET
    D=A
    @SP
    A=M
    M=D
    @SP
    M=M+1
// 2. 依序推 LCL、ARG、THIS、THAT（四段同形）
    @LCL
```

```
    D=M
    @SP
    A=M
    M=D
    @SP
    M=M+1
    //（ARG、THIS、THAT 同上，各 6 條）
// 3. ARG = SP - 5 - n（n 由 sprintf 以 %d 填入 n+5 亦可）
    @SP
    D=M
    @5
    D=D-A
    @n
    D=D-A
    @ARG
    M=D
// 4. LCL = SP（新函式的 local 區從這裡開始）
    @SP
    D=M
    @LCL
    M=D
// 5. 跳去函式本體
    @Foo.bar        // get_function_label 的結果
    0;JMP
// 6. 放返回標籤——函式 return 後從這裡繼續
(RET)
```

兩個容易忽略的點：

1. $ARG = SP - 5 - n$ **是「推完框架之後」算的**：引數是呼叫**前**就推上堆疊的（`push` 序列），推完 5 格框架後，第一個引數在 $SP - 5 - n$。順序不能換：先算 ARG 再推框架的話公式就變了。

2. **返回標籤每次呼叫都要新的**（`get_next_label_name`）：同一個函式可能被呼叫一百次，每個呼叫點的返回位址都不同。函式入口標籤則相反——必須每次都一樣（`get_function_label` 是純函數式的命名規則，不帶計數器），`call` 與 `function` 兩邊才會生成同一個名字。

Test 2 (NestedCall) 在測什麼： `Sys.init` → `Sys.main` → `Sys.add12` 的**巢狀**呼叫，專抓 SimpleFunction 抓不到的 bug：框架推入順序錯、THIS/THAT 沒有正確保存／恢復、`call`/`return` 推入與彈出的字數不對等（堆疊「歪斜」，症狀是 RAM[0] 錯）。同樣由測試腳本佈置假框架，沒有外部 `call Sys.init`。

期望值： $RAM[0..6] = 261, 261, 256, 4000, 5000, 135, 246$。

題目的重要提醒：你的框架裡的**返回位址**數值**不會**跟 VM 模擬器一致，也不必一致——它取決

於你的翻譯器生成了幾條組語指令，測試腳本刻意不比對這一格。除錯時可搭配測試資料附的 `.html` 檔（NestedCallStack.html 畫出了各時間點的期望堆疊）。

5　資料夾編譯與開機碼（Test 3、Test 4）

5.1　compile_folder 導讀

題目

讀懂骨架的 `compile_folder`（`is_folder` 與 `list_vm_files` 是平台相關的黑魔法，題目明說不用看懂），然後在它開頭加上：初始化 SP = 256、模擬一次對 `Sys.init` 的呼叫。

解答

為什麼掃資料夾「出乎意料地難」？ C 標準函式庫**沒有**列目錄的可攜 API——POSIX 用 `opendir/readdir`，Windows 用 `FindFirstFile/FindNextFile`，兩套完全不同（這正是那兩個識別字撞名改名的來源）。骨架把這坨平台細節包掉，`compile_folder` 的骨幹是：

```
// 概念流程（骨架原始碼的形狀）
compile_folder(folder, output):
    寫入開機碼              // <-- 你要補的就是這裡
    for 每個 folder 裡的 .vm 檔:
        lex_file(...)      // 與單檔模式相同
        parse_file(...)    // 與單檔模式相同
```

開機碼（bootstrap）——你要加的組語，恰好是兩件事：

```
// (1) SP = 256（上週從 parse_file 拔掉的那行，回歸正確位置）
    @256
    D=A
    @SP
    M=D
// (2) call Sys.init 0 —— 直接重用 parse_call 的生成邏輯!
//     推返回標籤、推 LCL/ARG/THIS/THAT、
//     ARG = SP-5、LCL = SP、跳 Sys.init、放返回標籤
```

實作技巧：(2) 不必重寫——在 C 端直接呼叫你的 `parse_call` 生成函式（傳入函式名 `Sys.init`、$n = 0$）即可，一行搞定且永遠與 `call` 的實作同步。

為什麼是「模擬完整呼叫」而不是單純 `goto Sys.init`？兩個理由：(1) 測試腳本的期望值假設完整框架——開機推了 5 格，Sys.init 的堆疊從 **261** 開始，FibonacciElement 期望 $RAM[261] = 3$ 正是這個佈局；(2) `Sys.init` 以 `function Sys.init k` 開場，它假設 LCL 已被 `call` 設好——沒有框架，local 清零會寫到未定義的位置。題目也保證：`Sys.init` **不含 `return`**（結尾是無窮迴圈），所以那個永遠不會被用到的返回標籤無傷大雅。

5.2　Test 3 與 Test 4

題目

用 `FibonacciElement`（Test 3）與 `StaticsTest`（Test 4）驗證——這兩個測試都要把**整個資料夾**當參數傳給翻譯器，且測試腳本**不會**再幫你佈置框架。

解答

Test 3：FibonacciElement——函式機制的總驗收。資料夾含兩個檔：

- `Main.vm`：`Main.fibonacci`，**遞迴**計算費氏數列第 n 項（$fib(n) = fib(n-1) + fib(n-2)$），對翻譯器而言遞迴不需要任何特殊處理——這正是呼叫框架設計的勝利：每層呼叫的框架結構相同、位置由暫存器攜帶，第幾層都一樣）；
- `Sys.vm`：`Sys.init` 呼叫 `Main.fibonacci 4` 後進入無窮迴圈。

輸出必須叫 `FibonacciElement.asm`（資料夾同名）。**沒有開機碼這個測試必掛**——它假設翻譯器自己初始化執行環境。期望值：

$$RAM[0] = 262, \qquad RAM[261] = 3 \quad (fib(4) = 3)$$

（開機框架佔 256–260，返回值落在 261，SP 停在 262。）

Test 4：StaticsTest——檢驗**上週**的 `static` 實作在多檔下是否正確，本週不用寫新程式碼。資料夾含 `Class1.vm`、`Class2.vm`、`Sys.vm`：兩個 Class 各有 `set/get` 函式，分別對**自己的** static 0、static 1 存取；`Sys.init` 先 `Class1.set(6,8)`、`Class2.set(23,15)`，再取 `Class1.get()` $= 6 - 8 = -2$、`Class2.get()` $= 23 - 15 = 8$。期望值：

$$RAM[0] = 263, \qquad RAM[261] = -2, \qquad RAM[262] = 8$$

通過的前提：上週翻譯 `static i` 時用了「**檔名.i**」形式的組語符號（如 `@Class1.0`、`@Class2.0`）——組譯器自然把不同檔案的 static 配到不同 RAM 格。若當初用了全域共用的命名（例如一律 `@static.0`），Class2 的 `set` 會覆寫 Class1 的資料，兩個 `get` 會算出 $8 - 15$ 之類的錯值——「應該直接通過或直接失敗」，題目這句話就是這個意思。

進階討論

你剛完成的是一個真正的編譯器後端。把視野拉遠：（1）**這套呼叫協定不是玩具**——x86-64 的 System V ABI 做的是同一件事，只是快取友善版：返回位址照樣推堆疊（`call` 指令內建），但前六個整數引數走暫存器（rdi、rsi、rdx、rcx、r8、r9）而非堆疊，callee-saved 暫存器選擇性保存，框架指標 rbp 在優化編譯下甚至可以省略。Hack VM 的「全部進堆疊」是這個家族最樸素的成員。（2）**開機碼無處不在**——你的 C 程式在 `main` 之前也有一段 `crt0`（C runtime zero）：設定堆疊、清 BSS 段、準備 argc/argv 再 `call main`——與你剛寫的 `SP=256; call Sys.init 0` 精神完全相同。（3）**每檔一個 static 命名空間**就是連結器（linker）符號解析的雛形：C 的 `static` 全域變數有 internal linkage，效果與 `Class1.0` 如出一轍。

A 附錄：速查表

A.1 三條指令的完整規格

指令	生成組語的行為
`call f n`	推返回標籤位址 → 推 LCL、ARG、THIS、THAT → $ARG = SP - 5 - n$ → $LCL = SP$ → 跳 `(f)` → 放返回標籤
`function f k`	放標籤 `(f)`（僅由函式名導出，`get_function_label`）→ $SP = LCL + k$、local $0..k-1$ 清零
`return`	$R13 = RAM[LCL-5]$ → $RAM[ARG]$ = 返回值 → $SP = ARG+1$ → THAT、THIS、ARG、LCL ← $RAM[LCL-1..LCL-4]$ → 跳 $R13$

框架內相對 `LCL` 的位置：$LCL-1$ 舊 THAT、$LCL-2$ 舊 THIS、$LCL-3$ 舊 ARG、$LCL-4$ 舊 LCL、$LCL-5$ 返回位址；引數在 ARG、$ARG+1$、⋯；local 在（新）LCL 起。

A.2 四個測試總表

測試	輸入	前置佈置	期望值（`.cmp`）
1 SimpleFunction	單檔	腳本佈置假框架	$RAM[0..4] = 311, 305, 300, 3010, 4010$; $RAM[310] = 1196$
2 NestedCall	`Sys.vm` → `NestedCall.asm`	腳本佈置假框架	$RAM[0..6] = 261, 261, 256, 4000, 5000, 135, 246$
3 FibonacciElement	整資料夾	**無**（靠開機碼）	$RAM[0] = 262$; $RAM[261] = 3$
4 StaticsTest	整資料夾	**無**（靠開機碼）	$RAM[0] = 263$; $RAM[261] = -2$; $RAM[262] = 8$

A.3 常見錯誤型錄

症狀	病因
無引數函式 return 後跳到亂七八糟的位址	返回位址沒先存 R13 就寫 $RAM[ARG]$（$n=0$ 時兩者同一格!）
Test 1 過、Test 2 的 RAM[0] 錯（堆疊歪斜）	`call` 推的字數與 `return` 收的字數不對等（少推／多推框架欄位）
Test 2 的 RAM[3] ／ RAM[4] 錯	THIS ／ THAT 保存或恢復順序錯（框架順序必須與講義一致）
Test 3 必掛、前兩個都過	沒有開機碼，或開機碼沒模擬完整 `call`（只 `goto Sys.init`，堆疊基底差 5 格）
遞迴（fibonacci）算錯但單層呼叫對	返回標籤重複（沒用 `get_next_label_name` 生成唯一標籤）
Test 4 掛、前三個都過	上週的 `static` 沒用「檔名.i」命名，多檔共用了同一組 RAM 格
local 沒清零、SimpleFunction 算出非 1196	`function` 只挪 SP 沒鋪零
框架返回位址與 VM 模擬器不同而恐慌	正常！位址取決於你生成的指令數，測試腳本不比對這一格

A.4 Hack RAM 配置備忘

位址	用途	位址	用途
0–4	SP、LCL、ARG、THIS、THAT	16–255	static
5–12	temp 段（R5–R12）	256–2047	堆疊
13–15	翻譯器保留（R13–R15）	2048–16383	堆積

A.5 參考資料

- COMSM1302 第十週講義與影片（函式呼叫機制、呼叫框架、開機流程），University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 8（Virtual Machine II: Program Control）——四組測試腳本的出處.
- nand2tetris Project 8（nand2tetris.org/project08）與 NestedCall 測試說明（NestedCallStack.html）.
- System V AMD64 ABI（x86-64 呼叫慣例對照）.