COMSM1302 電腦架構概論
第五週完整教材：Hack 組合語言

擷取—執行週期 · Hack 架構 · 組合語言基礎 · 跳躍與流程控制 · 記憶體映射 I/O

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

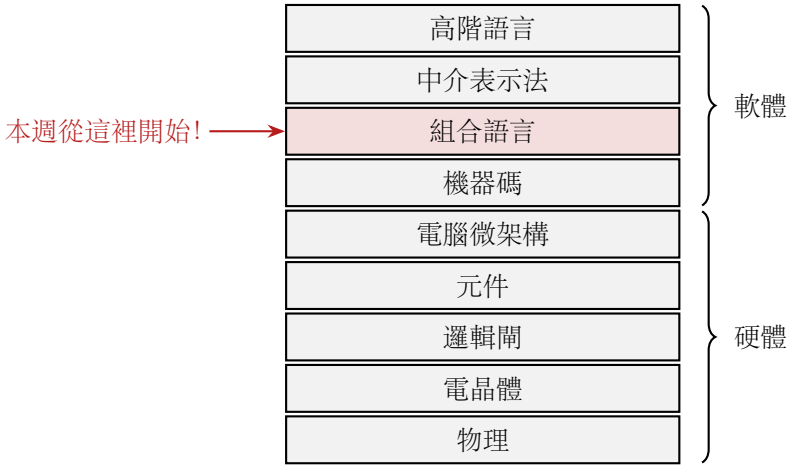
2026 年 7 月

本教材的使用方式：本講義整合了第五週四份投影片（5-1 The fetch-execute cycle、5-2 Hack assembly I: The basics、5-3 Hack assembly II: Loops and conditionals、5-4 Hack assembly III: Input and output）的全部內容，並補充了 nand2tetris 的機器碼格式、鍵盤代碼表、圖靈機與 Church–Turing 論題等延伸知識。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。本週是課程的**轉捩點**：前四週在抽象階梯的硬體層蓋積木（閘、加法器、暫存器、計數器），本週開始**往軟體層爬**——學習第一個「語言」：Hack 組合語言，並理解 CPU 如何一拍一拍地執行程式。

目錄

1　擷取—執行週期與 Hack 架構（The Fetch-Execute Cycle）

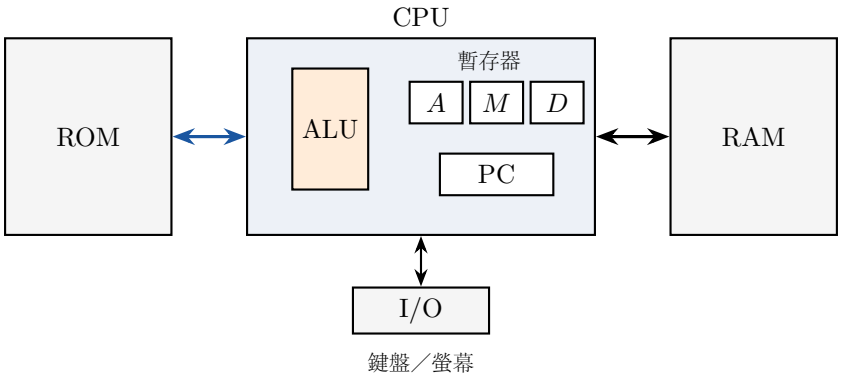
1.1　課程要往哪裡去



課程前半部專注在**硬體**：你已在實驗課做出 Hack CPU 的各個元件（暫存器、PC、ALU 等）。要理解 Hack 架構，我們必須開始思考**軟體**。組合語言、機器碼與架構三者**緊密綁在一起**，所以我們先學 Hack 組合語言，再往下鑽。之後的路線圖：

1. 組裝出完整的 Hack CPU（下週!）；
2. 用 C 寫一個 Hack **組譯器（assembler）**；
3. 朝 Hack 的高階語言「Minijack」邁進——用 C 寫 **VM 翻譯器**，再寫一個（非常簡單的）**編譯器**。

1.2　Hack 架構總覽



擷取—執行週期（fetch-execute cycle）

每個時脈週期，Hack **中央處理器（CPU）**從**唯讀記憶體（ROM）**讀取（**擷取，fetch**）一條 16 位元二進位**指令**，然後**執行（execute）**這條指令——執行過程可能讀寫鍵盤、螢幕或 32KB 的**隨機存取記憶體（RAM）**。這個循環是**所有 CPU 共通**的心跳。（不是所有 CPU 都從 ROM 擷取指令——本課程稍後會談。）

CPU 內部：

- 最複雜的部分是**算術邏輯單元（ALU）**，處理 +、−、& 等算術與布林運算（第二週做過！）；
- 四個暫存器：A、M、D 與**程式計數器（PC）**——PC 存放**下一條要擷取的指令位址**（第四週的計數器！）。

為什麼需要暫存器？因為每個時脈只能從 RAM 讀**一個字（word）**。中間結果若都得寫回 RAM，速度會慢得無法接受——所以把最常用的值留在 CPU 內部。

1.3　看一個程式跑起來

以下程式計算 $RAM[0] + RAM[1] + 17$ 並把結果存入 $RAM[2]$（$RAM[i]$ 表示 RAM 位址 i 的 16 位元字）。假設初始 $RAM[0] = 10$、$RAM[1] = 42$。先別管每條指令怎麼編碼——**專注在擷取—執行週期**上：

ROM 位址	指令（組語）	A	M	D	PC′	效果
0	`@0`	0	10	–	1	$A \leftarrow 0$；M 自動 $= RAM[0] = 10$
1	`D=M`	0	10	10	2	把 $RAM[0]$ 讀進 D
2	`@1`	1	42	10	3	$A \leftarrow 1$；$M = RAM[1] = 42$
3	`D=D+M`	1	42	52	4	$D \leftarrow 10 + 42 = 52$
4	`@17`	17	0	52	5	直接把常數 17 載入 A
5	`D=D+A`	17	0	69	6	$D \leftarrow 52 + 17 = 69$
6	`@2`	2	0	69	7	$A \leftarrow 2$，準備寫入位址 2
7	`M=D`	2	69	69	8	$RAM[2] \leftarrow 69$ ✓
8	`@8`	8	0	69	9	$A \leftarrow 8$（自己的位址）
9	`0;JMP`	8	0	69	8	無條件跳回位址 8：無窮迴圈

幾個關鍵觀察：

- M **永遠是** $RAM[A]$：改 A 就等於「換一格記憶體來看」。讀寫 RAM 都透過 M，所以**先載入位址到 A、再操作 M** 是 Hack 的基本節奏。
- PC 每條指令後**自動加一**——除非被跳躍指令改寫。**操縱 PC 就是實作迴圈與條件的方法**。
- 程式最後進入**無窮迴圈**。所有 Hack 程式都這樣收尾——因為**沒有「halt」指令**！若不圈住，PC 會繼續前進、執行 ROM 裡的垃圾內容。

本章重點

- 本週從硬體跨入軟體：組語 → 組譯器 → VM → 編譯器的路線圖。
- Hack = ROM（程式）+ CPU（ALU、$A/M/D$/PC）+ RAM（資料）+ 鍵盤／螢幕 I/O。
- 擷取—執行週期：每拍從 ROM[PC] 擷取一條 16 位元指令並執行。
- $M \equiv RAM[A]$；暫存器存在的理由：每拍只能讀一個 RAM 字。

- 沒有 halt——用無窮迴圈收尾；操縱 PC = 流程控制。

2　Hack 組合語言 I：基礎（The Basics）

2.1　一點歷史：從 EDSAC 到個人電腦

1949 年，劍橋的 **EDSAC** 成為最早投入常規使用的 **儲存程式電腦**之一（與曼徹斯特 Mark 1 有先後之爭）。而即使在那時，人們就已經**不想直接寫機器碼**了——他們改用**助憶符（mnemonics）**：一行文字對應一條指令。這就是第一個**組合語言（assembly language）**，1950 年代普及開來。把組語翻譯成機器碼的程式叫**組譯器（assembler）**。

那打孔卡呢？ 到 1970 年代都還有人用打孔卡輸入機器碼——但不是因為組語還沒發明！當全公司只有一台大型主機時，用整批打孔卡載入程式或資料，比讓大家排隊進機房一個個打字快多了。之後打孔卡讓位給**笨終端機（dumb terminal）**（本身無運算能力、透過網路連上主機），再於 1980–90 年代讓位給**個人電腦**。

2.2　為什麼學 Hack 組語？

由於**一行組語對應一條指令**，世界上沒有「唯一的組合語言」——不同架構（x86-64、i386、ARM、MIPS…）各有各的組語。我們教 Hack 組語不是因為它本身直接有用，而是它能**幫你打好基礎，之後學任何架構的組語都容易**。

最佳化過的組語常比編譯出來的程式碼更快。多數高效能語言（C、C++、Rust）都允許**內嵌組語**（如 gcc 的 `asm`）。

警告：組語**不可攜、難讀、難維護**！只有當你**已經確認**（例如用 profiler）某段程式是嚴重的效能瓶頸時，才考慮用組語「最佳化」。一般而言，**過早最佳化是壞主意**。

2.3　Hack 的暫存器

Hack CPU 有暫存器 A、M、D、PC，各存一個 16 位元字：

暫存器	意義與用途
A	**位址暫存器（address register）**。唯一能**直接載入常數**的暫存器。
M	**記憶體暫存器（memory register）**。讀 = $RAM[A]$、寫 = 更新 $RAM[A]$。
D	**資料暫存器（data register）**。一般用途儲存；改它**不會**影響 M。
PC	**程式計數器**。下一條指令永遠是 $ROM[PC]$；寫它 = 在程式裡跳躍。

M 的本質：把 A 的值**當指標並解參考（dereference）**——就像 C 語言的 `*p`！更長期或更大量的儲存，用 RAM。（寫 PC 的方式與其他暫存器不同——見第 3 章。）

2.4　暫存器操作

`@[數字]` 把數字載入 A：例如 `@42` 使 $A = 42$。其餘操作都是「**目的地 = 運算**」的形式：

常數	例	一元運算	例	二元運算	例
指定 0	`M=0`	恆等	`D=A`	加法	`D=A+D`
指定 1	`D=1`	取負	`A=-D`	減法	`M=M-D`
指定 -1	`A=-1`	位元 NOT	`D=!D`	位元 AND	`D=D&A`
		遞增	`A=A+1`	位元 OR	`A=D|M`
		遞減	`M=M-1`		

別死背這張表——需要時回來查就好！

合法性規則（重要！）

- 所有二**元運算**必須介於**兩個不同的暫存器**之間，且**至少一個是** D。所以 `D=A+M`（沒有 D 參與運算）與 `M=D+D`（同一暫存器兩次）都**不合法**。✗
- 更複雜的算式如 `A=D+M+A`、`D=17` 也不合法。✗
- **多重指定**合法：左邊寫多個暫存器，如 `MD=D-M` 把 $D-M$ 同時存入 M 與 D。語法規定 A、M、D 必須**按此順序**出現——`AMD=D+M` 合法 ✓、`DAM=D+M` 不合法 ✗。

2.5　第一個程式：add.asm

把 $RAM[0]+RAM[1]+17$ 存入 $RAM[2]$（即第 1 章追蹤過的程式）：

```
@0
D=M      // D <- RAM[0]
@1
D=D+M    // D <- D + RAM[1]
@17
D=D+A    // D <- D + 17
@2
M=D      // RAM[2] <- D
```

注意「`@` 位址 → 操作 `M`」的成對節奏——這是 Hack 組語的典型型態。（所有 Hack 程式都應以無窮迴圈收尾——見第 3 章。）

2.6　組譯器的小恩惠：註解與關鍵字

組語勝過機器碼的另一好處：**註解**。組譯器會跳過以 `//` 開頭的行、空行與行首空白；行中註解（如 `A=D // 註解`）也可以。

組譯器還會把 `@R0` 換成 `@0`、`@R1` 換成 `@1`、……直到 `@R15` 換成 `@15`。

虛擬暫存器（virtual registers）

RAM 位址 0 到 15 有時稱為**虛擬暫存器**。硬體上它們與其他位址無異，但我們**約定**用它們存放輸入輸出等。寫 `@R5` 而非 `@5`，是向自己表明：「我在乎的是**位址** 5（要用 M 讀 $RAM[5]$）」，而不是**數值** 5（例如要加到 D 上）。

還有其他這類關鍵字，（很）之後才會用到：

$$\texttt{SP} \mapsto 0, \quad \texttt{LCL} \mapsto 1, \quad \texttt{ARG} \mapsto 2, \quad \texttt{THIS} \mapsto 3, \quad \texttt{THAT} \mapsto 4,$$

$$\texttt{SCREEN} \mapsto 16384, \qquad \texttt{KBD} \mapsto 24576.$$

用關鍵字改寫 add.asm，可讀性更好：

```
// D <- RAM[0]
@R0
D=M
// D <- D + RAM[1]
@R1
D=D+M
// D <- D + 17
@17
D=D+A
// RAM[2] <- D
@R2
M=D
```

2.7　較大的恩惠：變數

組譯器也支援 `@` 敘述裡的**字母變數**。第一次寫 `@var` 時，組譯器把字串 `var` 綁定到一個**唯一的 RAM 位址**（從 **16** 開始分配），之後所有 `@var` 都替換成該位址。變數**區分大小寫**。

例：依序使用變數 `foo`、`Foo`、`FOO`，則 `@foo` 變成 `@16`、`@Foo` 變成 `@17`、`@FOO` 變成 `@18`。

盡量用變數取代裸位址——可讀性大增。但小心：**這不是 C 的變數**！沒有型別，而且**唯一的記憶體配置機制**就是上面說的「從 16 號開始編號」。
什麼時候不適合用變數？ 例如輸出是**兩個各一百個數字的列表**——你不會想宣告兩百個變數名。更糟的是**長度在執行期才決定**的列表——變數在組譯期就得定案，根本無法對應。這種情況要直接操作位址（指標式寫法，見練習）。

本章重點

- 1949 年 EDSAC 時代就有組語；組譯器把助憶符翻成機器碼。
- 每個架構有自己的組語；學 Hack 是為了打基礎；別過早最佳化。
- A = 位址（唯一可載常數）；M = $RAM[A]$（指標解參考）；D = 資料；PC = 下一條指令位址。
- 二元運算：兩個不同暫存器、至少一個 D；多重指定按 A、M、D 順序。
- `R0`–`R15`、`SCREEN`、`KBD` 等關鍵字；變數從位址 16 起自動分配、區分大小寫。

3　Hack 組合語言 II：迴圈與條件（Loops and Conditionals）

3.1　來自過去的恐懼：goto

到目前為止我們能算 $(RAM[3] + RAM[4])$ & $RAM[5]$ 之類的簡單算式，但還不算「真正的電腦」——只是**接了時鐘的計算機**。我們需要迴圈與條件，但 Hack 沒有 `if` 或 `while`。我們先聊點……更古老、更黑暗、更飢餓的東西：**goto**。

在 C 裡，`goto` 讓你從程式**任何地方**「跳」到指定的標籤：

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n\n");
    goto skip;
    printf("Gotos can skip lines of code.\n\n");
skip:
    return 0;
}
```

執行第 5 行的 `goto skip` 時，程式跳過第 6 行的列印，直接從第 8 行 `skip:` 之後繼續，回傳 0。

3.2　用 goto 做出迴圈與條件

C 的一切流程控制都能用 goto + 單行 if 表達！例如：

```c
int main() {
    int i = 0;
    while (i < 10) {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

⇒

```c
int main() {
    int i = 0;
loop:
    if (i >= 10)
        goto endloop;
    printf("%d ", i);
    i++;
    goto loop;
endloop:
    return 0;
}
```

注意右邊的模式：**條件反轉**（`i < 10` 變成 `i >= 10` 跳出）、迴圈尾端**無條件跳回開頭**。這正是等一下 Hack 組語的寫法。

3.3　Go To Statement Considered Harmful

那為什麼寫程式要用 `while` 和 `else` 而不用 goto?

goto **完全不受限制**（在函式內）：你可以在一萬行的函式裡，從 20 個不同地方跳到同一個標籤。看到迴圈或 if，你**確切知道**控制流會怎麼走；看到標籤，程式**可能從任何地方跳過來**。

結構化程式設計（structured programming）

用 `if`、`while` 與**函式呼叫**控制程式流程的做法，稱為**結構化程式設計**，於 1960 年代興起。在那之前這些構造都不存在，一切流程控制都靠 goto。如今它是整個軟體工程學科的基礎。（本節標題來自 Edsger W. Dijkstra 1968 年的著名文章 *Go To Statement Considered Harmful*——該文開創了「結構化程式設計」一詞與運動。）

C 裡仍有極少數情況 goto 算合理（例如多層迴圈的錯誤處理出口），但除非你很清楚自己在做什麼，最好避開。

可怕的真相是：在組語層面，goto 加上單行 if **通常是唯一的流程控制**——編譯器把你寫的每個 `while` 都翻譯成跳躍。結構化程式設計是**高階語言給你的抽象**，硬體裡不存在。

3.4　Hack 的跳躍（jumps）

我們把組語層的 goto 稱為**跳躍（jump）**或**分支（branch）**，以強調每個都對應**單一一條機器碼指令**。

跳躍語法

任何**不以 `@` 開頭**的指令，後面都可以接分號與七種跳躍助憶符之一，讀作：「若〔指令的運算結果〕滿足〔條件〕，goto A **中存的 ROM 位址**」。

- 例：`M=A+D;JGT` 把 $A+D$ 存入 M，然後**若** $A+D>0$ 跳到 A 中的位址。
- 可省略左邊的指定：`A+D;JGT` 也是合法組語——不存值，只算 $A+D$ 並據以決定是否跳。

助憶符	全名	跳躍條件
`JMP`	JuMP	永遠跳
`JGT`	Jump if Greater Than	〔結果〕>0
`JEQ`	Jump if EQual	〔結果〕$=0$
`JLT`	Jump if Less Than	〔結果〕<0
`JGE`	Jump if Greater than or Equal	〔結果〕≥ 0
`JNE`	Jump if Not Equal	〔結果〕$\neq 0$
`JLE`	Jump if Less than or Equal	〔結果〕≤ 0

一樣不用背——用時查表。記住：**所有跳躍都跳到 A 存的位址**。

警告：PC 與 A 是**同時**在下一個時脈開始時更新的！像 `A=A+D;JMP` 這種「一邊改 A 一邊要跳去 A」的指令是**未定義行為**——你不知道會跳到舊 A 還是新 A。

3.5　組譯器的幫助：標籤（labels）

每一（非註解）行組語就是一行機器碼。所以要無條件跳到組語第 100 行（存於 $ROM[99]$），得寫 `@99` 加 `0;JMP`。（習慣上用 `0;JMP` 表示「不附帶運算的跳躍」——這是慣例，不是規定。）

問題：這樣維護起來是災難。若要刪掉第 1–99 行中的任何一行，或插入一行呢？**整個檔案的跳躍都要改**！

解法：跟 C 一樣，組譯器提供**標籤**。形如 `(Label)` 的一行**不對應任何機器碼**；若下一行會落在 ROM 位置 100，組譯器就把所有 `@Label` 替換成 `@100`。

3.6　完整範例：Sum.asm

`Sum.asm` 把 $0+1+\cdots+RAM[0]$ 的總和輸出到 $RAM[1]$：

$RAM[0]$（輸入）	$RAM[1]$（輸出）
0	0
1	$0+1=1$
2	$0+1+2=3$
3	$0+1+2+3=6$
⋮	⋮

```
// sum = 0, i = 1
@sum
M=0
@i
M=1
(LOOP)
    // if i - RAM[0] > 0, goto END
    @i
    D=M
    @R0
    D=D-M
    @END
    D;JGT
    // sum = sum + i
    @i
    D=M
    @sum
    M=M+D
    // i = i + 1
    @i
    M=M+1
    // goto LOOP
    @LOOP
```

```
    0;JMP
(END)
    // 慣例的收尾無窮迴圈
    @END
    0;JMP
```

逐段解說：

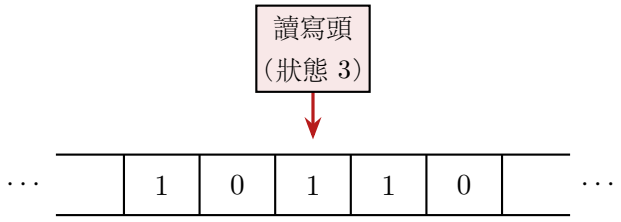
1. **初始化**：變數 `sum`（組譯器分配位址 16）設 0、`i`（位址 17）設 1。
2. **迴圈條件**：算 $D = i - RAM[0]$；若 $D > 0$（即 $i > RAM[0]$）跳去 `END`。注意這正是 goto 版 while 的**條件反轉**模式。
3. **迴圈主體**：$sum \mathrel{+}= i$、$i \mathrel{+}= 1$，然後無條件跳回 `LOOP`。
4. **收尾**：`(END)` 處自我跳躍的無窮迴圈。最後把 `sum` 搬到 $RAM[1]$ 的步驟可插在 `(END)` 之前——完整版見練習 6 的變形。

（實際課程影片中的版本把結果直接累加在 $RAM[1]$，兩種寫法等價；上面的版本多用了變數，展示變數與標籤如何合作。）

3.7　什麼才算「真正的」電腦？

圖靈機（Turing machine）　（此定義本身不列入考試）

一條**雙向無限**的紙帶分成許多格，每格存二進位值；加上一個**讀寫頭**與**有限多個狀態**。每一步，依據當前格子的值與內部狀態，讀寫頭寫下 1 或 0、向左或向右移一格，然後機器改變狀態或**停機（halt）**。例如：「狀態 3、讀到 1 → 寫 0、右移、進入狀態 3」。



為什麼要在乎？ 因為 **Church–Turing 論題**說：只要一個計算問題**可解**，就存在一台精心設計的圖靈機能解它——把輸入寫上紙帶、跑到停機、讀紙帶得輸出。

圖靈完備（Turing-complete）

若一個計算模型能**模擬任何圖靈機**，我們就說它是**圖靈完備**的——它就是一台「真正的電腦」。有了跳躍（條件分支）與可讀寫的記憶體，Hack（在記憶體無限的理想化下）就是圖靈完備的。

注意：不是所有計算問題都可解！ 判斷「任意一台圖靈機是否終會停機」是**不可能**的——這就是著名的**停機問題（Halting Problem）**。

本章重點

- 組語層的流程控制只有 goto + 單行 if；高階語言的 while/if 是編譯器提供的抽象（結構化程式設計，Dijkstra 1968）。
- Hack 跳躍：`dest=comp;jump`，七種條件（JMP/JGT/JEQ/JLT/JGE/JNE/JLE），一律跳到 A 中位址。
- `A=A+D;JMP` 未定義；先設 A、再跳。
- `(Label)` 不佔機器碼；組譯器把 `@Label` 換成位址——可維護性的救星。
- while 迴圈的組語模式：條件反轉跳出 + 尾端無條件跳回。
- 圖靈機、Church–Turing 論題、圖靈完備、停機問題。

4　Hack 組合語言 III：輸入與輸出（Input and Output）

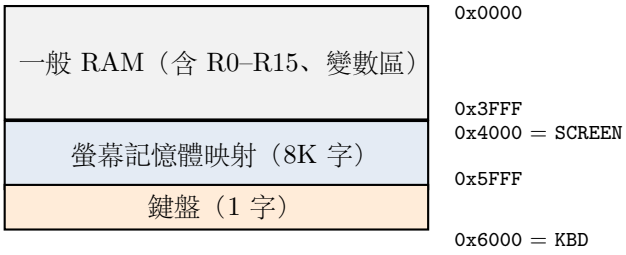
4.1　記憶體映射 I/O

Hack 唯一的輸入週邊是**鍵盤**、唯一的輸出週邊是**螢幕**。

記憶體映射 I/O（memory-mapped I/O）

所有輸入輸出都**映射到記憶體位址**：硬體層面上，CPU **寫螢幕的方式與寫記憶體相同、讀鍵盤的方式與讀記憶體相同**。組語因此寫起來非常簡單！

Hack 有 32KB 實體記憶體、以 16 位元字為單位，所以位址需要 $2^{18}/2^4 = 2^{14}$ 位元表示：位址空間 `0x0000` 到 `0x3FFF`。整張記憶體地圖：



- 寫入 `0x4000`–`0x5FFF` 的任何值都會**顯示在螢幕上**；
- 若鍵盤有鍵被按住，其代碼會出現在 `0x6000`。

4.2　鍵盤輸入

關鍵字 `KBD` 映射到 `0x6000`（= 24576），方式跟 `R1` 映射到 `0x0001` 一樣。例如按住「d」時，`@KBD` 會把 24576 載入 A、此時 $M = 100$（「d」的代碼）。**沒有按鍵時，`0x6000` 內容為 0**。

常用按鍵代碼（節錄自 Nisan & Schocken 附錄 5）：

按鍵	代碼	按鍵	代碼	按鍵	代碼
space	32	0–9	48–57	newline（Enter）	128
A–Z	65–90	a–z	97–122	backspace	129
← ↑ → ↓	130–133	esc	140	F1–F12	141–152

警告：無法偵測**同時按下多個鍵**（一次只有一個代碼）。現代鍵盤其實也常有這問題（key rollover 限制）！

4.3　螢幕輸出

Hack 的解析度是 512×256：256 列、每列 512 個像素。像素依「**書寫順序**」編號——由左至右、由上至下。

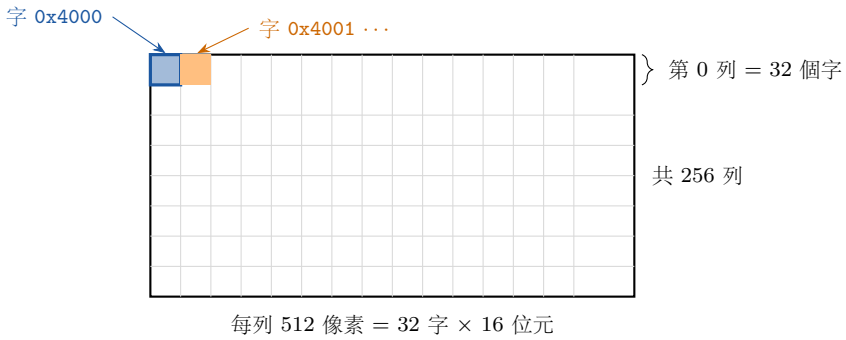
像素 ↔ 記憶體的對應

第 i 個像素顯示黑色，若且唯若從位址 `0x4000` 起算、由 lsb 數來第 i 個位元為 1（0 = 白色）。所以 `0x4000`–`0x5FFF` 的**每個字控制 16 個像素**！

等價地，第 r 列、第 c 行（都從 0 起算）的像素，由位址

$$\texttt{0x4000} + 32r + (c \,/\, 16) \qquad \text{(整數除法)}$$

中**由右數來第** $(c \bmod 16)$ **個位元**控制。



還有第三種寫法——直接把位址的**二進位結構**拆開：

$$\underbrace{010}_{\text{固定前綴}}\ \underbrace{rrrrrrrr}_{r\text{ 的 8 位元}}\ \underbrace{ccccc}_{c\text{ 的前 5 位元}}$$

為什麼三種寫法等價？`0x4000` $= 010\,00000000\,00000_2$；$32r$ 恰好把 r 左移 5 位；$c/16$（整數除法）就是 c 的 9 位元中**丟掉最低 4 位**後剩下的前 5 位。把三段接起來，正是上面的位元佈局。（$c \bmod 16 = c$ 的最低 4 位元 = 字內的位元編號。）

4.4　陷阱與技巧

警告：`@` 指令只能載入最多 15 位元的值！（機器碼裡 A-指令的第一個位元固定是 0——見附錄。）想把前 16 個像素塗黑，也就是把 `0xFFFF` 寫進 `0x4000`：`@65535` 是**行不通**的。替代方案：

- `@0` 之後 `A=!A`（0 全反 = `0xFFFF`）；
- 或者更簡單：直接 `A=-1`（2 的補數的 −1 就是全 1）。

實用技巧：

- CPU 模擬器可以切換以**十六進位或二進位**顯示暫存器與記憶體（比十進位好對照位元圖樣）。
- 關鍵字 `SCREEN` 映射到 `0x4000`：`@SCREEN` 加 `M=1` 就把第一個像素塗黑。
- 螢幕記憶體**可讀也可寫**！若左上 16 個像素全黑，`@SCREEN` 加 `D=M` 會把 `0xFFFF` 存入 D。

4.5　完整範例：Fill.asm

`Fill.asm`：把整個螢幕塗黑；但只要有任何鍵被按住，改塗白。無限輪詢（poll）鍵盤：

```
(POLL)
    // 依鍵盤狀態決定顏色：預設黑 (-1)
    @KBD
    D=M
    @BLACK
    D;JEQ      // 沒有按鍵 (KBD=0) -> 塗黑
    @colour    // 有按鍵 -> 塗白 (0)
    M=0
    @FILL
    0;JMP
(BLACK)
    @colour
    M=-1       // 全 1 = 16 個黑像素
(FILL)
    // i 從螢幕最後一個字往回塗到第一個字
    @8191
    D=A
    @i
    M=D
(FILLLOOP)
    // addr = SCREEN + i
    @i
    D=M
    @SCREEN
    D=D+A
    @addr
    M=D
    // RAM[addr] = colour
    @colour
    D=M
    @addr
    A=M        // 把 A 設成 addr 的內容：指標!
    M=D
    // i = i - 1; 若 i >= 0 繼續塗
    @i
    MD=M-1
    @FILLLOOP
    D;JGE
    // 塗完整個螢幕，回去重新讀鍵盤
    @POLL
    0;JMP
```

值得注意的三件事：

1. **輪詢迴圈**：程式永不停止，每塗完一輪就回 `POLL` 重讀 `KBD`——這是沒有中斷（interrupt）機制時的標準做法。
2. **指標寫法**：`A=M` 把「變數 `addr` 存的位址」載入 A，下一行的 `M` 就是**那個位址**的內容——等同 C 的 `*addr = colour`。這正是第 2 章說「變數行不通」時（大量、動態位置）的替代方案。
3. 螢幕共 8192 個字（`0x4000`–`0x5FFF`），所以 `i` 從 8191 遞減到 0。一次寫一個字 = 一次塗 16 個像素。

本章重點

- 記憶體映射 I/O：讀鍵盤、寫螢幕與讀寫 RAM 的指令完全相同。
- 位址地圖：RAM `0x0000`–`0x3FFF`；螢幕 `0x4000`–`0x5FFF`（= `SCREEN`）；鍵盤 `0x6000`（= `KBD`，無鍵時為 0）。
- 螢幕 512×256、書寫順序編號；像素 $(r, c) \to$ 位址 $\texttt{0x4000} + 32r + c/16$、位元 $c \bmod 16$；一個字管 16 個像素。
- `@` 只能載 15 位元非負值；要 `0xFFFF` 用 `A=-1`。
- `A=M` = 指標解參考；輪詢 = 無中斷機制下的 I/O 標準模式。

5　綜合練習題（附詳解）

練習 1：擷取—執行追蹤

初始 $RAM[0]=7$、$RAM[1]=3$，其餘為 0。逐條追蹤下列程式，寫出每條指令後的 A、D 與相關 RAM 內容：

```
@0
D=M
@1
D=D-M
@3
M=D
```

解答

ROM	指令	A	D	備註
0	`@0`	0	0	$M = RAM[0] = 7$
1	`D=M`	0	7	$D \leftarrow 7$
2	`@1`	1	7	$M = RAM[1] = 3$
3	`D=D-M`	1	4	$D \leftarrow 7 - 3 = 4$
4	`@3`	3	4	$M = RAM[3]$
5	`M=D`	3	4	$RAM[3] \leftarrow 4$

程式效果：$RAM[3] = RAM[0] - RAM[1] = 4$。（嚴格來說程式還該加上無窮迴圈收尾。）

練習 2：哪些指令合法？

判斷下列每條指令是否為合法的 Hack 組語，不合法者說明原因：`D=D+M`、`D=A+M`、`M=D+D`、`AMD=D|M`、`DAM=D+A`、`D=17`、`A=A+1`、`M=-M`、`@-5`、`D=M;JGT`。

解答

- `D=D+M` ✓（二元運算含 D、兩個不同暫存器）。
- `D=A+M` ✗——二元運算的**運算元**必須至少一個是 D；$A+M$ 沒有 D 參與。
- `M=D+D` ✗——二元運算必須介於**兩個不同**暫存器。
- `AMD=D|M` ✓（多重指定，A、M、D 順序正確）。
- `DAM=D+A` ✗——多重指定必須按 A、M、D 的順序。
- `D=17` ✗——常數只能是 0、1、-1；載入 17 要用 `@17` 再 `D=A`。
- `A=A+1` ✓（一元遞增）。
- `M=-M` ✓（一元取負）。

- `@-5` ✗——`@` 只接受**非負**的 15 位元數（或符號）；負數要用運算做出來（如 `@5`、`D=-A`）。
- `D=M;JGT` ✓——合法：把 M 存入 D，若 $M > 0$ 則跳到 A 中位址。（注意：這裡 A 同時是跳躍目標與 M 的位址——實務上要小心安排。）

練習 3：直線程式

寫出計算 $RAM[2] = RAM[0] - RAM[1] + 5$ 的 Hack 組語（含收尾）。

解答

```
@R0
D=M       // D = RAM[0]
@R1
D=D-M     // D = RAM[0] - RAM[1]
@5
D=D+A     // D = D + 5
@R2
M=D       // RAM[2] = D
(END)
@END
0;JMP
```

模式與 add.asm 完全相同：常數也得先進 A 才能參與運算。

練習 4：變數的位址分配

一支程式依序首次使用變數 `count`、`Count`、`tmp`，之後再次出現 `@count`。問四次 `@` 各被替換成什麼？若程式同時也用了 `@R7` 與標籤 `(LOOP)`（位於 ROM 位置 12），它們又如何處理？

解答

變數從位址 16 依**首次出現順序**分配、區分大小寫：`@count` → `@16`、`@Count` → `@17`（大小寫不同即不同變數）、`@tmp` → `@18`；再次出現的 `@count` 仍是 `@16`（同一變數同一位址）。
`@R7` 是**預定義符號**，直接替換成 `@7`（不佔變數區）。`(LOOP)` 是**標籤**：本身不產生機器碼，所有 `@LOOP` 替換成 `@12`（ROM 位址，而非 RAM）。注意三種符號的本質不同：變數 → RAM 資料位址、標籤 → ROM 指令位址、預定義符號 → 固定數值。

練習 5：if ／ else

寫出組語實作：若 $RAM[0] > 0$ 則 $RAM[1] = 1$，否則 $RAM[1] = 0$。

解答

```
@R0
D=M
@POS
D;JGT      // RAM[0] > 0 -> 跳去 POS
// else 分支
@R1
M=0
@END
0;JMP      // 跳過 then 分支!
(POS)
@R1
M=1
(END)
@END
0;JMP
```

兩個常見錯誤：(1) 忘了 else 分支結尾的 `0;JMP`，導致執行完 else 又「掉進」then 分支；(2) 條件方向弄反。對照 C 的 goto 版 if ／ else 可幫助檢查。

練習 6：迴圈——乘法

Hack 沒有乘法指令！用重複加法寫出 $RAM[2] = RAM[0] \times RAM[1]$ （假設 $RAM[1] \geq 0$）。

解答

```
// RAM[2] = 0; i = RAM[1]
@R2
M=0
@R1
D=M
@i
M=D
(LOOP)
// if i == 0, goto END
@i
D=M
@END
D;JEQ
// RAM[2] += RAM[0]
@R0
D=M
@R2
M=M+D
// i--
```

```
@i
M=M-1
@LOOP
0;JMP
(END)
@END
0;JMP
```

迴圈骨架同 Sum.asm：條件反轉跳出（$i = 0$ 時結束）、主體累加、計數器遞減、無條件跳回。執行 $RAM[1]$ 次加法，每次把 $RAM[0]$ 加進 $RAM[2]$。

練習 7：為什麼 `A=A+D;JMP` 未定義？

解釋此指令的問題，並給出安全的等價寫法（跳到 $A + D$）。

解答

跳躍的語意是「跳到 A **中的位址**」，但這條指令**同時**把 A 改成 $A + D$。PC 與 A 都在**下一個時脈開始時**更新，所以硬體無法保證 PC 拿到的是舊 A 還是新 A——未定義行為。

安全寫法：**先**把目標算好放進 A，**再**用另一條指令跳：

```
A=A+D      // 先更新 A（這條不跳）
0;JMP      // 再無條件跳到（新的）A
```

第二條指令執行時 A 已穩定，跳躍目標明確。

練習 8：跳躍條件判讀

設 $D = -3$。下列指令各會不會跳？`D;JGT`、`D;JLE`、`D;JNE`、`D;JEQ`、`D+1;JLT`、`0;JMP`。

解答

判斷依據都是〔運算結果〕與 0 的比較：

- `D;JGT`：$-3 > 0$？否 → **不跳**。
- `D;JLE`：$-3 \le 0$？是 → **跳**。
- `D;JNE`：$-3 \neq 0$？是 → **跳**。
- `D;JEQ`：$-3 = 0$？否 → **不跳**。
- `D+1;JLT`：$-3 + 1 = -2 < 0$？是 → **跳**（注意條件看的是**運算結果** $D + 1$，不是 D）。
- `0;JMP`：無條件 → **跳**。

練習 9：螢幕位址計算

(a) 求第 $r = 100$ 列、第 $c = 200$ 行像素對應的 RAM 位址與位元編號。(b) 若要把該像素**塗黑但不動同字的其他 15 個像素**，該怎麼做（文字說明即可）？

解答

(a) 位址 $=$ `0x4000` $+ 32 \times 100 + 200/16 = 16384 + 3200 + 12 = 19596 =$ `0x4C8C`。位元編號 $= 200 \bmod 16 = 8$（由 lsb 起算第 8 位）。

(b) 用**位元 OR**：先造出只有第 8 位是 1 的遮罩 $2^8 = 256$，再 `RAM[19596] |= 256`：

```
@256
D=A         // D = 遮罩 0000000100000000
@19596
M=D|M       // 只把第 8 位設 1，其他位不變
```

若直接 `M=D` 會把同字其他 15 個像素全塗白！（塗白單一像素則用 AND 與反遮罩。）

練習 10：15 位元的限制

(a) 為什麼 `@65535` 無法把 `0xFFFF` 載入 A？(b) 給出兩種把 `0xFFFF` 放進 RAM[`SCREEN`] 的寫法。(c) 想載入 40000（超過 15 位元範圍的正數）該怎麼辦？

解答

(a) A-指令的機器碼第一位固定是 0（區別於 C-指令），只剩 **15 位元**放數值——範圍 0 到 32767。65535 超出範圍。

(b) 方法一：

```
@SCREEN
M=-1        // 2 的補數 -1 = 全 1 = 0xFFFF
```

方法二：

```
@0
D=!A        // D = NOT 0 = 0xFFFF
@SCREEN
M=D
```

(c) 40000 $=$ `0x9C40` 的最高位是 1，無法直接 `@`。用運算組出來，例如 $40000 = 20000 \times 2 = 20000 + 20000$：

```
@20000
D=A
@20000
D=D+A       // D = 40000（以 2 的補數位元圖樣存在）
```

（在 16 位元 2 的補數下它「等於」-25536，但位元圖樣正確——當位址或位元圖樣使用時完全沒問題。）

練習 11：等待按鍵

寫一段組語：**忙等（busy-wait）**直到使用者按下任意鍵，然後把按鍵代碼存入 $RAM[0]$。再改成：等待使用者按下「空白鍵」（代碼 32）才繼續。

解答

任意鍵版本：

```
(WAIT)
@KBD
D=M
@WAIT
D;JEQ       // KBD == 0（沒按鍵）就繼續等
@R0
M=D         // 存下按鍵代碼
```

空白鍵版本——把「是否為 0」的測試換成「是否等於 32」：

```
(WAIT)
@KBD
D=M
@32
D=D-A       // D = KBD - 32
@WAIT
D;JNE       // 不是空白鍵（差值非 0）就繼續等
```

技巧：Hack 沒有「比較相等」指令，**相減後與 0 比較**就是標準做法（JEQ ／ JNE）。

練習 12：觀念題——圖靈完備

(a) 敘述 Church–Turing 論題與「圖靈完備」的意義。(b) Hack 少了跳躍指令還會是圖靈完備的嗎？為什麼？(c) 停機問題告訴我們什麼？(d) 嚴格來說，真實的 Hack 電腦是圖靈完備的嗎？

解答

(a) Church–Turing 論題：凡是**可解**的計算問題，都存在圖靈機能解。因此若一個計算模型能**模擬任何圖靈機**（稱為**圖靈完備**），它就能算「真正的電腦」。

(b) **不會**。沒有（條件）跳躍就無法依資料改變控制流——程式只能從頭直直跑到尾，做的事相當於固定的組合邏輯（「接了時鐘的計算機」）。條件分支 + 可讀寫記憶體是關鍵成分。

(c) **不是所有問題都可解**：不存在演算法能判定任意圖靈機（程式）是否終將停機。這是「可計算性」的根本上限，與硬體多快無關。

(d) 嚴格來說**不是**：圖靈機有**無限**紙帶，Hack 只有 32KB RAM（任何真實電腦記憶體都有限）。慣例上我們說「在記憶體理想化為無限的前提下」它是圖靈完備的；真實機器都只是圖靈機的**有限近似**。

A 附錄：速查表

A.1 Hack 組語速查

語法	意義
`@n`（$0 \le n \le 32767$）	$A \leftarrow n$（A-指令，15 位元）
`@` 符號	$A \leftarrow$ 符號對應的位址（變數／標籤／預定義）
`dest=comp`	計算 comp、存入 dest
`comp;jump`	計算 comp、依條件跳到 $ROM[A]$
`dest=comp;jump`	以上兩者兼有
`(Label)`	標籤宣告（不佔機器碼）
`// ...`	註解

dest（按 A、M、D 順序）：`M`、`D`、`MD`、`A`、`AM`、`AD`、`AMD`。

comp 可用運算：`0`、`1`、`-1`；`D`、`A`、`M`；`!D`、`!A`、`!M`；`-D`、`-A`、`-M`；`D+1`、`A+1`、`M+1`；`D-1`、`A-1`、`M-1`；`D+A`、`D+M`；`D-A`、`D-M`；`A-D`、`M-D`；`D&A`、`D&M`；`D|A`、`D|M`。

jump：`JGT`（>0）、`JEQ`（$=0$）、`JGE`（≥ 0）、`JLT`（<0）、`JNE`（$\ne 0$）、`JLE`（≤ 0）、`JMP`（永遠）。

A.2 延伸補充：機器碼格式（下週預告）

- **A-指令**：`0vvvvvvvvvvvvvvv`——第一位 0，後 15 位是數值。
- **C-指令**：`111a cccccc ddd jjj`——固定前綴 `111`；`a` 位選擇 ALU 第二運算元用 A 還是 M；6 個 `c` 位是 ALU 控制位（第二週的 zx/nx/zy/ny/f/no!）；3 個 `d` 位指定目的地 $A/D/M$；3 個 `j` 位編碼跳躍條件。

這解釋了語法限制的**硬體根源**：comp 表就是 ALU 控制位能編出的函數集；dest 恰好 3 位對應 3 個暫存器；jump 恰好 3 位對應 $<, =, >$ 三個旗標的組合。

A.3 預定義符號

符號	位址	符號	位址
`R0`–`R15`	0–15	`THIS`	3
`SP`	0	`THAT`	4
`LCL`	1	`SCREEN`	16384（`0x4000`）
`ARG`	2	`KBD`	24576（`0x6000`）

A.4 記憶體地圖

位址範圍	內容	備註
`0x0000`–`0x000F`	虛擬暫存器 R0–R15	慣例用途
`0x0010`–	變數區	組譯器從 16 起分配
–`0x3FFF`	一般 RAM	共 16K 字
`0x4000`–`0x5FFF`	螢幕映射	8K 字 $= 512 \times 256$ 像素
`0x6000`	鍵盤	無鍵時為 0

A.5 名詞中英對照

英文	中文	英文	中文
fetch-execute cycle	擷取—執行週期	label	標籤
instruction	指令	jump / branch	跳躍／分支
program counter (PC)	程式計數器	structured programming	結構化程式設計
assembler	組譯器	memory-mapped I/O	記憶體映射輸入輸出
assembly language	組合語言	polling	輪詢
mnemonic	助憶符	busy-wait	忙等
machine code	機器碼	Turing machine	圖靈機
address register	位址暫存器	Church–Turing thesis	Church–Turing 論題
dereference	解參考	Turing-complete	圖靈完備
virtual registers	虛擬暫存器	Halting Problem	停機問題
variable	變數	stored-program computer	儲存程式電腦

A.6 參考資料

- John Lapinskas, *The fetch-execute cycle ／ Hack assembly I–III*（5-1 至 5-4），COMSM1302, University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 4（Machine Language）與附錄 5（鍵盤代碼）.
- Edsger W. Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM, 1968.
- Wikipedia：*Hack computer*、*EDSAC*、*Turing machine*、*Church–Turing thesis*、*Halting problem*.
- Randall Munroe, xkcd #292 *goto*（迅猛龍漫畫）.