COMSM1302 電腦架構概論

Week 5 作業：Hack 組合語言練習（第一部分）完整詳解

組譯器與模擬器 · 基本程式 · 除錯工具 · 位元遮罩 · 螢幕與鍵盤 I/O

逐題解析（含完整可執行程式碼）

2026 年 7 月

本作業的任務：

1. 寫幾個基本的 Hack 組合語言程式；
2. 學會用 Hack 組譯器（Assembler）與 CPU 模擬器（CPU Emulator）測試與除錯；
3. 學習位元遮罩（bit masking）；
4. 學會在 Hack 組語中使用螢幕與鍵盤。

為什麼分兩部分？第一部分（本卷）涵蓋考試所需的內容，應在第 8 週開始前完成——之後課程的一切都建立在你對組語的理解上。第二部分是較難的練習題，做不完沒關係。**所需軟體：**nand2tetris 套件的 Assembler 與 CPU Emulator（需要 Java 執行環境；實驗室機器用 `module load nand2tetris` 載入）。

目錄

1　入門（Getting Started）

題目

(1)載入課堂的 `add.asm`，用組譯器轉成 `add.hack`；在 CPU 模擬器載入，設 `RAM[0]`＝3、`RAM[1]`＝4，執行後 `RAM[2]` 應為 24。（2）調整動畫速度、認識 “No animation” 模式的注意事項。（3）試著把 `add.asm` 直接載入模擬器。（4）自己寫 `sub.asm`：把 `RAM[0]` 減去 `RAM[1]`，結果存入 `RAM[2]`（假設無溢位）。

解答

(1) 工作流程。組譯器把 `.asm`（人類可讀的組語）翻譯成 `.hack`（0/1 組成的機器碼文字檔）。在 Assembler 中開啟 `add.asm` → 按 “Fast translation” → 存出 `add.hack` → 在 CPU Emulator 以 ROM 載入 → 在 RAM 顯示區直接點格子輸入 `RAM[0]`＝3、`RAM[1]`＝4 → 按雙箭頭（run）。為什麼是 24 而不是 7？因為課堂的 `add.asm` 算的是 $2 \times \texttt{RAM[0]} + 6 \times \texttt{RAM[1]} = 6 + 18 = 24$ ……開個玩笑——真正的原因依課堂版本而定，重點是：**先讀程式再預測結果，執行只是驗證你的預測。**若載入的是 nand2tetris 官方的 `Add.asm`（計算 2＋3 存入 `R0`）行為又不同——務必用課程 unit page 提供的版本。

(2) 模擬器的三個實用旋鈕（考試必備）：

- **動畫速度滑桿**（上方工具列）：預設很慢，調到最快省下大量等待；
- **“Animate: No animation”**：完全不畫動畫，大程式必用。**兩個陷阱**：此模式下 (a) 不能在執行中編輯 RAM 的值；(b) **執行中 RAM 顯示不會更新**——程式明明已經算完，畫面上的 RAM 看起來卻沒變，要按停止（或程式跑到無窮迴圈後按暫停）才會刷新。考試時別被這點騙到；
- **清除記憶體按鈕**（RAM 顯示區上方）：第二輪測試前把 RAM 歸零。

(3) 直接載入 `.asm`。 CPU 模擬器可以直接開 `.asm`（它會即時翻譯），考試時快得多。但你要**從體感上理解**：組譯器能把 `.asm` 翻成可直接進 ROM 的機器碼——第 7 週談硬體時，CPU 讀的是機器碼不是組語；第 8 週我們會拆開組譯器本身的原理。組譯器另一個用途是除錯（見第 2 節的 fast translation）。

(4) `sub.asm`。思路：D 暫存器當運輸工具——先載 `RAM[0]`，減掉 `RAM[1]`，存到 `RAM[2]`：

```
// sub.asm -- RAM[2] = RAM[0] - RAM[1]
    @R0
    D=M         // D = RAM[0]
    @R1
    D=D-M       // D = RAM[0] - RAM[1]
    @R2
    M=D         // RAM[2] = D
(END)
    @END
    0;JMP       // 慣例：無窮迴圈結束程式
```

三個值得養成的習慣：（a）用 `@R0` 而不是 `@0`——雖然等價，但明確表示「這是暫存器不是常數」；（b）每行註解——第 2 節除錯時你會感謝自己；（c）結尾放 `(END)` 無窮迴圈——Hack 沒有「halt」指令，不加的話 PC 會繼續往下執行 ROM 裡的垃圾。測試：模擬器中設 `RAM[0]`$=9$、`RAM[1]`$=5$，跑完 `RAM[2]`$=4$ ✓；再到 codestuff.online 貼上跑多組測資。

2 乘法與除錯（Multiplication and Debugging）

2.1 mult.asm：先寫 C、再翻譯

題目

寫 `mult.asm`：把 `RAM[0]` 乘以 `RAM[1]`，結果存入 `RAM[2]`。假設無溢位、兩數皆非負。開始前先想：不用 `*` 運算子的 C 要怎麼寫？再把 C 翻譯成組語。

解答

第一步：C 版本（不用 `*`）。乘法 = 重複加法：$a \times b$ 就是把 b 加 a 次。

```
// C 偽代碼
result = 0;
n = RAM[0];            // 還要再加幾次
while (n > 0) {
    result = result + RAM[1];
    n = n - 1;
}
RAM[2] = result;
```

第二步：逐句翻譯成 Hack。 C 的每一行變成一小段組語，C 註解保留下來當地圖：

```
// mult.asm -- RAM[2] = RAM[0] * RAM[1]
// (RAM[0], RAM[1] >= 0, 無溢位)

// result = 0   (直接用 RAM[2] 當 result)
    @R2
    M=0
// n = RAM[0]
    @R0
    D=M
    @n
    M=D
(LOOP)
// while (n > 0) ... 條件不成立就離開
    @n
    D=M
    @END
    D;JLE       // n <= 0 → 結束
```

```
// result = result + RAM[1]
    @R1
    D=M
    @R2
    M=D+M
// n = n - 1
    @n
    M=M-1
    @LOOP
    0;JMP
(END)
    @END
    0;JMP
```

翻譯模式（第 5-3 講的套路）：`while` → 「標籤 + 條件跳離 + 回跳」；變數 → `@` 名字的自動配置（從 RAM[16] 起）。

正確性檢查（邊界情況先想好再跑）：`RAM[0]` = 0 → 迴圈零次、`RAM[2]` = 0 ✓；3 × 4：加 3 次 4 → 12 ✓。效率：迴圈次數 = `RAM[0]`，所以**把較小的數當計數器**會快很多（2 × 10000 跑 2 圈 vs 10000 圈）——進階版可先比較兩數再決定誰當 `n`。

按題目要求：先**不要**急著除錯，往下一節學工具，最後再回來修它。

2.2 odd.asm：三件除錯武器

題目

下載 `odd.asm`（內含刻意種下的 bug）。它的規格：`RAM[0]` 是正數；依奇偶性把 `RAM[0]` 或 42 存入 `RAM[1]`（兩個測試案例：`RAM[0]` = 501 應存 42、`RAM[0]` = 500 應存 500）。依序用三種工具修好它：（1）組譯器的 fast translation 找語法錯誤；（2）Asm ／ Sym 模式對照找 501 案例的 bug；（3）中斷點（breakpoint）找 500 案例的 bug。

解答

規格小注：題目正文寫「奇數存 `RAM[0]`、否則存 42」，但兩個測試案例（501 → 42、500 → 500）一致指向相反方向：**偶數存 `RAM[0]`、奇數存 42**。以測試案例為準——這本身就是除錯的第一課：**規格與測試不一致時，先弄清楚哪個才是真相**。

由於每年下載的 `odd.asm` 內容可能不同，以下把「怎麼用工具」與「這類 bug 長什麼樣」講透——方法對了，什麼變種都修得掉。

武器一：組譯器的 Fast Translation（語法錯誤）。

CPU 模擬器拒絕載入 `.asm` 時只給一句沒用的錯誤訊息，但**拒載必定表示語法錯誤**。開 Assembler → 按 **Fast translation**（工具列右數第二顆）→ 它會**停在出錯的那一行**並反白。常見語法錯誤型態：

- 打錯指令拼字：`M=D+`（少運算元）、`D=M+2`（Hack 沒有 `+2`，合法常數只有 `0`、`1`、`-1`）；

- 目的地順序錯：`DM=...` 在舊版組譯器要寫 `MD=...`；
- 括號／分號用錯：`(LOOP` 少右括號、`D;JGT;` 多分號；
- `@` 後面接了非法內容：`@32768`（超過 15 位元）、`@2x`（變數不能以數字開頭）。

修一個、重跑 fast translation，直到成功產出 `.hack`。

武器二：Asm ／ Sym 模式對照（501 案例）。

ROM 顯示區上方的下拉選單：

- **Sym（symbolic）模式**：顯示你**寫的**原始碼——變數名、標籤名、行尾註解都在；
- **Asm 模式**：顯示 CPU **實際執行**的樣子——變數與標籤全部換成數字位址、註解消失。

對照兩個模式就能看穿「符號層」的謊言。這類「有點壞心但極常見」的 bug 的代表是：**打錯標籤或變數名**。Hack 的規則：`@` 名字若不是已宣告的標籤，就**默默變成一個新變數**（配到 RAM[16] 之後）——不會有任何錯誤訊息！症狀與診斷：

- 症狀：程式跳到莫名其妙的地方（`@LOOP` 打成 `@LOOp`，跳躍目標從 ROM 位址變成「值為某小數字的新變數」，`0;JMP` 就跳去 ROM 開頭附近鬼打牆）；或某變數永遠是 0（讀到的是打錯字的新變數）；
- 診斷：在 **Asm 模式**下看那條 `@` 指令——它解析出來的**數字**跟你以為的標籤位址對不上（Sym 模式裡標籤旁會標示它對應的 ROM 位址，兩邊一比就現形）；
- 同型 bug：大小寫不一致（`@sum` vs `@Sum`）、用了 `@1` 想表達常數 1 卻寫成 `D=D&M`（讀了 `RAM[1]`）而非 `D=D&A`（用常數 1）——**A 與 M 的混淆**是 Hack 第一大 bug 來源。

單步工具：**single step**（走一步）與 **step back**（退一步；No animation 模式下不可用）。501 的路徑（奇數）通常很快分岔，單步走十幾步就能看到它跳錯地方。修好後 501 → 42 ✓。

武器三：中斷點（500 案例）。

500 是偶數路徑，若程式用「重複減 2」判斷奇偶，要繞 250 圈才出結果——單步走到天荒地老。**中斷點**讓模擬器全速跑到指定位置才停：

1. 切到 Sym 模式定位，捲到迴圈尾（例如 ROM[15]）——**右鍵點**那格 ROM，它變紅 = 中斷點設好；
2. 用 No animation 模式 run——它跑到 ROM[15] 停下；
3. 把動畫模式調回 “Program flow”（這樣才能 step back），在中斷點附近前後單步，找「哪一圈開始壞掉」；
4. 這類 bug 的代表：**迴圈邊界／步長錯誤**——減 2 寫成減 1（奇偶判斷整個歪掉）、`D;JGT` 與 `D;JGE` 之差（最後一圈多跑或少跑）、離開迴圈時 D 沒重新載入（**殘值 bug**：D 還留著上一段的計算結果）。

條件中斷點（工具列旗子圖示）更強：指定「PC、A、D、RAM[i] 或 time 等於某值」時暫停——「第 357 圈才壞」的 bug 就設 RAM[counter]= 357 直達案發現場。
最後：回頭修你的 mult.asm，用 codestuff.online 跑測資、掛了就用上面三件武器。如果你當初沒寫註解，現在要在一堆 @ 與 JMP 裡考古——把這當一次教訓：**註解是寫給兩週後的自己看的**。

3　遮罩（Masking）

觀念：把字組當**位元圖案**而非數字來操作，就用位元運算（bitwise operations）——它們就像 16 條並排的一位元閘：$x \wedge y$ 的第 i 位 $= x_i \wedge y_i$。C 提供 ~、&、|、^；Hack 組語提供 NOT、AND、OR （沒有 XOR，但可以拼出來）。**術語**：拿一個固定字組去 AND ／ OR ／ XOR 未知字組，那個固定字組叫**遮罩（mask）**；用一個位元存布林值時稱它為**旗標（flag）**，設 1 叫 set、清 0 叫 clear。三條口訣：**AND 清位**（$x_i \wedge 0 = 0$）、**OR 設位**（$x_i \vee 1 = 1$）、**XOR 翻位**（$x_i \oplus 1 = 1 - x_i$）。

3.1　and.asm：數 1 的個數（popcount）

題目

寫 and.asm：數 RAM[0] 中值為 1 的位元數，寫入 RAM[1]。例：RAM[0]= 11 = 0b1011 → RAM[1]= 3。（模擬器右上角 Format 切成 binary 可直接看位元。）

解答

難點：Hack 沒有移位指令。不能把 x 右移逐位檢查——但可以反過來**讓遮罩左移**：$mask$ 從 0x0001 開始，每輪 $mask = mask + mask$ （自加就是左移一位!），依序掃過 0x0001, 0x0002, …, 0x8000。終止條件也是免費的：0x8000 再自加，在 16 位元裡溢位成 0——「$mask = 0$」就是掃完 16 位。

```
// C 偽代碼
count = 0;
mask = 1;
while (mask != 0) {
    if ((RAM[0] & mask) != 0) count++;
    mask = mask + mask;       // 左移一位
}
RAM[1] = count;
```

```
// and.asm -- RAM[1] = RAM[0] 中 1 的個數
    @count
    M=0
    @mask
    M=1
(LOOP)
// if ((RAM[0] & mask) != 0) count++
```

```
    @mask
    D=M
    @R0
    D=D&M       // D = RAM[0] & mask
    @SKIP
    D;JEQ       // 該位是 0 → 不加
    @count
    M=M+1
(SKIP)
// mask = mask + mask（左移；0x8000 自加溢位成 0）
    @mask
    D=M
    MD=D+M      // mask 與 D 同時更新為 2*mask
    @LOOP
    D;JNE       // mask != 0 → 繼續掃
// RAM[1] = count
    @count
    D=M
    @R1
    M=D
(END)
    @END
    0;JMP
```

兩個值得玩味的細節：

- `MD=D+M`：一條指令同時把新遮罩寫回記憶體 **並**留在 D 裡給緊接的 `D;JNE` 判斷——Hack 的多目的地寫法省一次讀取；
- 負數也正確：*mask* 掃到 `0x8000` 時，`D&M` 若非零其值是 -32768，但 `JEQ` 只問「是否為 0」，不受符號影響 ✓。測試：`RAM[0]`$=-1=$`0xFFFF` $\to 16$；`RAM[0]`$=11\to 3$ ✓。

3.2　xor.asm：用 AND/OR/NOT 拼出 XOR

題目

寫 `xor.asm`：若 `RAM[0]` 能被 256 整除（即最低 8 位全為 0），翻轉 `RAM[1]` 的 **第 3 與第 5 高位**。例：`RAM[0]`$=512=$`0x0200` 時，`RAM[1]`$=2831=$`0x0B0F` 應變成 $8975=$`0x230F`。

解答

第一步：算出兩個遮罩。

- **整除 256 的判斷**：低 8 位全零 $\Leftrightarrow$ `RAM[0]` & `0x00FF` $=0$。遮罩 $=255$；
- **要翻的位**：16 位元字組的最高位是「第 1 高位」（bit 15），所以第 3 高位 $=$ bit 13（$2^{13}=8192$）、第 5 高位 $=$ bit 11（$2^{11}=2048$）。遮罩 $m=8192+2048=10240=$`0x2800`。

驗證：`0x0B0F` ⊕ `0x2800` = `0x230F` ✓（`0000 1011…` → `0010 0011…`：bit 13 由 1 翻 0、bit 11 由 0 翻 1）。

第二步：沒有 XOR？自己拼。

$$x \oplus m \;=\; (x \wedge \neg m) \vee (\neg x \wedge m)$$

（「x 中不在 m 的部分」加上「m 中不在 x 的部分」。）Hack 的 comp 欄有 `!A` 與 `!M`，所以 $\neg m$ 不用另存變數：`@10240` 後直接 `D=!A` 就是 $\neg m$——注意 $\neg m$ = `0xD7FF` = 55295 超過 15 位元，**不能**用 `@55295` 載入，這正是題目說「@ 指令只吃 15 位元正整數」的實戰場景。

```
// xor.asm -- 若 RAM[0] % 256 == 0,
//            RAM[1] ^= 0x2800（翻 bit13 與 bit11）

// if ((RAM[0] & 255) != 0) goto END
    @R0
    D=M
    @255
    D=D&A       // 常數用 A，不是 M!
    @END
    D;JNE
// t1 = RAM[1] & ~mask
    @10240
    D=!A        // D = ~0x2800 = 0xD7FF
    @R1
    D=D&M       // D = RAM[1] & ~mask
    @t1
    M=D
// D = ~RAM[1] & mask
    @R1
    D=!M        // D = ~RAM[1]
    @10240
    D=D&A       // D = ~RAM[1] & mask
// RAM[1] = t1 | D
    @t1
    D=D|M
    @R1
    M=D
(END)
    @END
    0;JMP
```

易錯點整理：

- 「第 k 高位」要換成 bit $16-k$——數錯一位整題就錯（第 3 高位是 bit 13 不是 bit 3!）；
- `D=D&A`（跟常數 AND）與 `D=D&M`（跟記憶體 AND）一字之差；

- 整除判斷只看**低 8 位**，跟 `RAM[0]` 其餘位無關——`RAM[0]` $= 0$ 也算整除（$0 \bmod 256 = 0$），本程式行為正確。

4　Hack 的輸入與輸出（Input and Output）

記憶體映射 I/O 速查（先看熟再動手）：

- **螢幕**：512×256 黑白像素，映射到 `RAM[16384..24575]`（符號 `@SCREEN`），每列 32 個字組。像素 (row, col) 住在 `RAM[16384 + 32·row + col/16]` 的第 $col \bmod 16$ 位；**位 0（LSB）畫在最左邊**——字組值 `1` 只點亮該組 16 像素的最左一顆；1 = 黑、0 = 白；
- **鍵盤**：單一字組 `RAM[24576]`（符號 `@KBD`）。有鍵按住 = 該鍵掃描碼（`c` 鍵 = 99，即小寫字母的 ASCII），沒按 = 0；
- **@ 指令限制**：`@x` 的 x 只能是變數、標籤或 **15 位元正整數**（$0 \sim 32767$）。要寫入像 `0x8000` 或 `0xFFFF` 這樣的 16 位元圖案，得用遮罩或取負（`M=-1`、`M=-M`、`D=A+1` 溢位等技巧）。

4.1　checker1.asm：兩顆角落像素

題目

寫 `checker1.asm`：沒有鍵按下時，螢幕**左上角**像素黑、其餘全白；按住 `c` 鍵時，**右下角**像素黑、其餘全白。

解答

先算地址與值（動手前的功課）：

- **左上角像素** $(0, 0)$：`RAM[16384]` 的 bit 0 ⇒ 寫入值 `0x0001`，直接 `M=1`；
- **右下角像素** $(255, 511)$：字組位址 $16384 + 32 \times 255 + 511/16 = 16384 + 8160 + 31 = 24575$（螢幕最後一個字組），位元 $511 \bmod 16 = 15$（MSB）⇒ 寫入值 `0x8000` $= -32768$。**@32768 不合法！**取得它的技巧：`@32767` 後 `D=A+1`——$32767 + 1$ 在 16 位元中溢位，恰好是 -32768 = `0x8000` ✓（這正是題目提示「用遮罩或取負」的用意；`D=-A` 給的是 -32767 = `0x8001`，多亮一顆像素，不行）。

程式骨架：無窮輪詢（poll）鍵盤，每圈把**兩顆像素都寫一次**（一顆設、一顆清），這樣按下／放開的瞬間畫面自動更新：

```
// checker1.asm
// 無鍵：左上角黑。按住 c（=99）：右下角黑。
(MAIN)
    @KBD
    D=M
    @99
```

```
    D=D-A
    @CKEY
    D;JEQ           // 按住 c → CKEY
// 預設狀態：RAM[24575]=0、RAM[16384] bit0=1
    @24575
    M=0
    @SCREEN
    M=1
    @MAIN
    0;JMP
(CKEY)
// c 狀態：RAM[16384]=0、RAM[24575] bit15=1
    @SCREEN
    M=0
    @32767
    D=A+1           // D = 0x8000（溢位技巧）
    @24575
    M=D
    @MAIN
    0;JMP
```

測試要領：用 No animation 模式 run，按下螢幕輸出區下方的鍵盤按鈕、點進鍵盤輸入區按住 c。若像素出現在**錯的位置**：停下來重看影片想清楚（最常見的誤解：以為 MSB 在左——其實 **LSB 才在左**；或把每列 32 字組算成 16）。別盲目亂試。

4.2　checker2.asm：全螢幕棋盤

題目

寫 checker2.asm：沒有鍵按下時，整個螢幕呈棋盤圖樣（左上角像素黑）；按住 c 時呈**相反**棋盤（左上角白）。無論何時按下 c，左上角像素都要即時正確。

解答

先算要寫什麼值（題目再三叮嚀的功課）：單像素棋盤 = 同一列內黑白相間、相鄰列錯開一位。

- 偶數列（第 0, 2, 4, …列）：最左像素黑 ⇒ bit 0 = 1，之後隔一位一個 1 ⇒ 字組 = 0b0101 0101 0101 0101 = 0x5555 = 21845（15 位元內，@21845 直接合法 ✓）；
- 奇數列：整體錯開一位 ⇒ 0xAAAA = ¬0x5555——**一個 NOT 就翻列**，不需要第二個常數；
- 按住 c 的相反棋盤：兩種列值**互換**——起始值從 0x5555 換成 0xAAAA，其餘邏輯完全相同。

先寫個五行小程式手動把 0x5555、0xAAAA 塞進 RAM[16384]、RAM[16416]（第二列開頭）驗證圖樣方向——理解錯了早點發現，省大痛。

結構：外層無窮迴圈重讀鍵盤決定起始圖樣，內層雙迴圈掃 256 列 × 32 字組，每列結尾把 *val* NOT 一次：

```
// checker2.asm -- 全螢幕棋盤，按住 c 反相
(MAIN)
// val = 0x5555；若按住 c 則 val = 0xAAAA
    @21845
    D=A
    @val
    M=D
    @KBD
    D=M
    @99
    D=D-A
    @DRAW
    D;JNE           // 不是 c → 用預設圖樣
    @val
    M=!M            // c 按住 → 0xAAAA
(DRAW)
// addr = SCREEN；row = 0
    @SCREEN
    D=A
    @addr
    M=D
    @row
    M=0
(ROWLOOP)
    @k
    M=0             // k = 本列已寫字組數
(WORDLOOP)
// RAM[addr] = val（指標間接定址！）
    @val
    D=M
    @addr
    A=M             // A ← addr 的內容
    M=D             // 寫入 RAM[addr]
// addr++
    @addr
    M=M+1
// if (++k < 32) 繼續本列
    @k
    MD=M+1
    @32
    D=D-A
    @WORDLOOP
```

```
    D;JLT
// 列結束：val 反相（隔列錯位）、下一列
    @val
    M=!M
    @row
    MD=M+1
    @256
    D=D-A
    @ROWLOOP
    D;JLT
    @MAIN           // 整幅畫完 → 回去重讀鍵盤
    0;JMP
```

三個關鍵技術點：

- **記憶體間接定址**：`@addr / A=M / M=D` ——「把 D 寫進『addr 所存位址』的格子」。這是螢幕程式的核心語法，也是本題想逼你練的東西；
- **每列 NOT 一次**：¬0x5555 = 0xAAAA、¬0xAAAA = 0x5555——256 列後自動回到起點；「反相棋盤」也只是換個起始值，程式其餘部分零改動；
- **「隨時按都正確」**：整幅畫完才回 `MAIN` 重讀鍵盤，反應延遲 = 畫一幅的時間（約 8192× 十餘條指令）。在模擬器的 No animation 模式下這通常夠快；若想更即時，可在**每列結尾**檢查 `KBD` 是否變化、變了就跳回 `MAIN` 重畫（軟體版的「升降緣偵測」——正是第四週 rise/fall detector 的軟體翻版）。

為什麼這題是好考題（原文的話）：螢幕鍵盤程式同時考驗**記憶體間接定址、條件、迴圈與二進位表示**——期末通常有一題 15 分的同型題。把 checker2 獨立寫出來，你就準備好了。

進階討論

效率討論：能不能少寫一點？ 本解每幅重寫全部 8192 個字組。兩個經典優化：

- **狀態變化才重畫**：記住上一圈的鍵盤狀態 `prev`，只在 `KBD` ≠ `prev` 跨越「是否為 c」的邊界時重畫（軟體版 rise/fall detector）；平常螢幕不動，迴圈只剩幾條指令；
- **就地反相**：兩種棋盤恰好互為 NOT——切換時不必重算圖樣，掃一遍螢幕做 `M=!M` 即可（免載入常數，每字組省兩三條指令）。

這些技巧在真實的圖形程式裡就是「dirty rectangle」與「就地位元運算」的雛形。

A 附錄：速查表

A.1 本作業的五支程式

程式	功能	核心技巧
`sub.asm`	`R2 = R0 - R1`	D 暫存器搬運
`mult.asm`	`R2 = R0 * R1`	重複加法迴圈
`and.asm`	`R1 = popcount(R0)`	遮罩自加 = 左移；溢位終止
`xor.asm`	條件翻轉 bit 13, 11	$x \oplus m = (x \wedge \neg m) \vee (\neg x \wedge m)$
`checker1/2.asm`	螢幕棋盤 + 鍵盤	記憶體映射 I/O、間接定址

A.2 除錯工具速查

工具	何時用	在哪裡
Fast translation	`.asm` 拒載（語法錯誤）	Assembler 工具列
Asm ／ Sym 切換	符號解析問題（打錯標籤名）	ROM 顯示區上方下拉
Single step ／ back	逐指令觀察	模擬器工具列
中斷點（右鍵 ROM）	跳過長迴圈	ROM 顯示區
條件中斷點	第 n 圈才壞	工具列旗子圖示
Format: binary	看遮罩／位元	視窗右上角

A.3 常見 bug 型錄

Bug	症狀
A/M 混淆（`D&A` vs `D&M`）	常數變成讀記憶體，值不可預期
打錯標籤名（默默變新變數）	跳躍亂飛／變數恆 0
D 殘值	分支後用到上一段的舊 D
迴圈邊界（JGT vs JGE）	多跑或少跑一圈
忘記結尾無窮迴圈	PC 跑進垃圾指令
No animation 下看 RAM	顯示不更新（其實已算完）

A.4 關鍵常數

常數	意義
`@SCREEN` = 16384	螢幕映射起點；每列 32 字組
24575	螢幕最後一個字組（右下角那 16 顆像素）
`@KBD` = 24576	鍵盤暫存器；`c` 鍵 = 99
`0x5555` = 21845	棋盤偶數列（LSB 在左 = 左上黑）
`0xAAAA` = ¬`0x5555`	棋盤奇數列
`0x8000` = `@32767` + `D=A+1`	MSB 遮罩（右下角像素）
`0x2800` = 10240	第 3、5 高位遮罩（bit 13, 11）
255 = `0x00FF`	低 8 位遮罩（整除 256 判斷）

A.5 參考資料

- COMSM1302 第五週講義與影片 5-2 ／ 5-3 ／ 5-4（Hack 組語、流程控制、記憶體映射 I/O），University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 4 與 Project 4——`Mult.asm` 與 `Fill.asm` 是本作業的近親.
- nand2tetris CPU Emulator Tutorial——中斷點為 ⟨ 變數, 值 ⟩ 配對（`A`、`D`、`PC`、`RAM[i]`、`time`）.
- Hack 螢幕映射：像素 $(row, col) \to$ `RAM[16384+32row+col/16]` 的第 $col \bmod 16$ 位，LSB 畫在左.