COMSM1302 電腦架構概論

Week 5 作業：Hack 組合語言練習（第二部分）完整詳解

邊緣觸發輸詢 · 移位與旋轉 · Collatz 猜想 · 快速乘法 · Rogue

逐題解析（含完整可執行程式碼）

2026 年 7 月

本作業的任務：

1. 學習正確的輸入輪詢（input polling）；
2. 學習左移／右移，用它們研究 Collatz 猜想並實作更快的乘法演算法；
3. 為一個遊戲實作基本的控制方案。

這是第一部分之後的**進階練習卷**——題目刻意出得比一週的合理份量多，做不完不用擔心；但每一題都在鍛鍊真實世界裡常用的組語技巧。

目錄

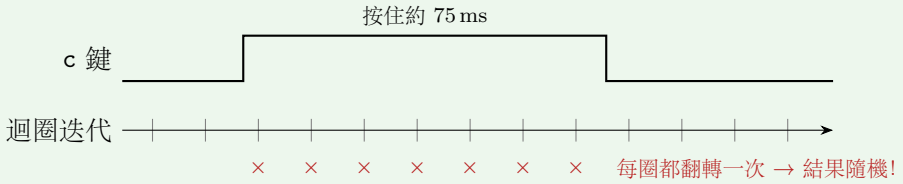
1　輸入處理的缺失環節（checker3.asm）

題目

把第一部分的 `checker2.asm`（按**住** `c` 切換棋盤）改成 `checker3.asm`：使用者**每按一次** `c`，兩種圖樣就切換一次。

解答

為什麼天真的寫法會壞掉？最自然的想法：沿用無窮繪圖迴圈，加一個變數 `pattern`，「`c` 有被按著就翻轉 `pattern`」。問題出在**速度的鴻溝**：繪圖迴圈每秒跑上百圈，而一次按鍵平均持續約 75 ms——大約 1/13 秒。一次按鍵期間迴圈跑了幾十圈，`pattern` 就被翻轉了幾十次，最後停在哪個值**形同擲硬幣**。



對症下藥：位準觸發 vs. 邊緣觸發。這正是課程第一部分（第三、四週）講過的對比：輪詢按鍵時，我們常常不想在「鍵**正被**按著」（**位準**）時觸發，而想在「自上次檢查以來鍵 **被按下了**」（**邊緣**）時觸發——一圈輪詢迴圈就類比一個時脈週期。解法是第四週升緣偵測器的**軟體版**：用變數 `prev` 記住「上一圈 `c` 是否按著」，只有 *現在按著 ∧ ¬剛才按著*（升緣!）才翻轉：

```
// checker3.asm -- 每「按一次」c 就切換棋盤
// pattern = 目前圖樣的起始字組（0x5555 或 0xAAAA）
// prev    = 上一圈 c 是否按著（軟體版升緣偵測）
    @21845
    D=A
    @pattern
    M=D             // pattern = 0x5555（左上黑）
    @prev
    M=0
(MAIN)
// 升緣偵測：只有「現在按著且剛才沒按」才翻轉
    @KBD
    D=M
    @99
    D=D-A
    @NOTC
    D;JNE           // 現在沒按 c
    @prev
    D=M
    @DRAW
    D;JNE           // 按著但上一圈也按著 → 不翻轉
    @prev
```

```
    M=1             // 記住「已按下」
    @pattern
    M=!M            // 翻轉圖樣（升緣！）
    @DRAW
    0;JMP
(NOTC)
    @prev
    M=0             // 放開了 → 重新武裝
(DRAW)
// 以下與 checker2 相同：畫整幅棋盤
    @pattern
    D=M
    @val
    M=D
    @SCREEN
    D=A
    @addr
    M=D
    @row
    M=0
(ROWLOOP)
    @k
    M=0
(WORDLOOP)
    @val
    D=M
    @addr
    A=M
    M=D
    @addr
    M=M+1
    @k
    MD=M+1
    @32
    D=D-A
    @WORDLOOP
    D;JLT
    @val
    M=!M
    @row
    MD=M+1
    @256
    D=D-A
    @ROWLOOP
```

```
    D;JLT
    @MAIN
    0;JMP
```

對照第四週： `prev` 就是 Mealy 版 one-shot 的那顆 D 正反器（$S_{new} = in$），翻轉條件 $in \wedge \neg S$ 就是 $rise$。硬體裡用正反器擋住位準，軟體裡用一個變數——同一個思想，兩層抽象。

2　旋轉與移位（Rotation and Shifting）

四種操作速覽（範例取自題目）：

操作	`1111111001111111` 變成	掉出的位	補進的位
左移（shift）	`1111110011111110`	最左掉出	右補 0
左旋（rotate）	`1111110011111111`	——	右補**舊最左位**
右旋	`1111111100111111`	——	左補**舊最右位**
右移（邏輯）	`0111111100111111`	最右掉出	左補 0

為什麼左移 = 乘 2? 位值原理：第 i 位的權重是 2^i，全體左移一格後每個位的權重都翻倍，所以整個數翻倍（超出 16 位的部分自然丟棄，即 mod 2^{16}）。這也是 Hack 裡實作移位的鑰匙：**Hack 沒有移位指令，但 $x + x$ 就是左移一位**。右移則區分**邏輯**右移（左補 0）與**算術**右移（讀作有號數除以 2、左補符號位）——對正數兩者相同。

2.1　leftshift.asm

題目

寫 `leftshift.asm`：`RAM[2]` ← `RAM[0]` ≪ `RAM[1]`（`RAM[1]` 非負）。想想 `RAM[1]` ≥ 16 時如何優化。

解答

核心：左移一次 = 自加一次，重複 `RAM[1]` 次。**優化**：移 16 位以上時每一位都被推出字組外，結果必為 0——直接寫 0，免跑迴圈（`RAM[1]` 可能大到 32767，不優化會空轉三萬圈）。

```
// leftshift.asm -- RAM[2] = RAM[0] << RAM[1]
    @R0
    D=M
    @R2
    M=D             // 結果從 RAM[0] 出發
    @R1
    D=M
    @n
    M=D
// 優化：n >= 16 → 全部位都移出 → 結果 0
```

```
    @16
    D=D-A
    @ZERO
    D;JGE
(LOOP)
    @n
    D=M
    @END
    D;JLE          // 移完了
    @R2
    D=M
    M=D+M          // R2 = 2 * R2（左移一位！）
    @n
    M=M-1
    @LOOP
    0;JMP
(ZERO)
    @R2
    M=0
(END)
    @END
    0;JMP
```

注意 `M=M+M` 不是合法的 comp——必須先 `D=M` 再 `M=D+M`。測試：`0x0001`$\ll 4 = 16$；`0xFFFF`$\ll 1 =$ `0xFFFE` ✓（負數照樣正確：自加對位圖案一視同仁）。

2.2　leftrotate.asm

題目

寫 `leftrotate.asm`：對 `RAM[0]` 左旋 `RAM[1]` 次存入 `RAM[2]`（`RAM[1]` 非負，不需優化 ≥ 16）。

解答

左旋一位 = 左移一位，**再把掉出去的舊最高位從最低位補回來**。判斷舊最高位：MSB 是符號位——$x < 0 \Leftrightarrow$ MSB $= 1$！而左移後的 LSB 必為 0，所以「補 1」用 $+1$ 就行：

$$\mathrm{rotl}(x) = \begin{cases} x + x, & x \geq 0 \\ x + x + 1, & x < 0 \end{cases}$$

```
// leftrotate.asm -- RAM[2] = RAM[0] 左旋 RAM[1] 位
    @R0
    D=M
    @R2
    M=D
    @R1
```

```
    D=M
    @n
    M=D
(LOOP)
    @n
    D=M
    @END
    D;JLE
    @R2
    D=M             // D = 舊值（符號位 = 舊 MSB）
    @NEG
    D;JLT
    @R2
    M=D+M           // MSB=0: 純加倍
    @DEC
    0;JMP
(NEG)
    @R2
    M=D+M           // MSB=1: 加倍……
    @R2
    M=M+1           // ……再把舊 MSB 補進 LSB
(DEC)
    @n
    M=M-1
    @LOOP
    0;JMP
(END)
    @END
    0;JMP
```

驗證題目範例：`1010111000011011` 左旋一位 → 舊 MSB $= 1$，加倍得 `0101110000110110`，$+1$ 得 `0101110000110111` ✓。

2.3　rightrotate.asm

題目

寫 `rightrotate.asm`：右旋 `RAM[1]` 次（`RAM[1]` 至多 15）。提示：右旋可以用左旋漂亮地表達——只需對左旋程式做一個小改動。

解答

手環比喻看穿一切：旋轉 16 次回到原點，所以

$$\text{右旋} k \text{ 位} \;=\; \text{左旋}(16-k) \text{ 位}.$$

對左旋程式的「小改動」就一行：迴圈計數器從 `RAM[1]` 改成 $16 - \texttt{RAM[1]}$。

```
// rightrotate.asm -- 右旋 k 位 = 左旋 16-k 位
    @R0
    D=M
    @R2
    M=D
    @16
    D=A
    @R1
    D=D-M           // D = 16 - k
    @n
    M=D
// 以下與 leftrotate 的迴圈一字不差
(LOOP)
    @n
    D=M
    @END
    D;JLE
    @R2
    D=M
    @NEG
    D;JLT
    @R2
    M=D+M
    @DEC
    0;JMP
(NEG)
    @R2
    M=D+M
    @R2
    M=M+1
(DEC)
    @n
    M=M-1
    @LOOP
    0;JMP
(END)
    @END
    0;JMP
```

邊界檢查：$k = 0 \to$ 左旋 16 位 = 原值 ✓（多跑 16 圈但結果正確）；$k = 15 \to$ 左旋 1 位 ✓。
驗證範例：`1111111001111111` 右旋一位：舊 LSB = 1 補到 MSB → `1111111100111111` ✓。

2.4　rightshift.asm

題目

寫 `rightshift.asm`：邏輯右移 `RAM[1]` 次（`RAM[1]` 至多 15）。提示：最簡單的做法從**右旋**開始！

解答

右旋之後，把「繞回來」的位擦掉就是右移。右旋 k 位後，原本的低 k 位繞到了字組頂端——邏輯右移要求那裡是 0，所以拿遮罩把**高 k 位清零**：

$$x \gg k \;=\; \mathrm{rotr}(x,k) \wedge \left(2^{\,16-k}-1\right)$$

（$2^{16-k}-1$ = 低 $16-k$ 位全 1。）妙處：旋轉圈數與遮罩的冪次**都是** $16-k$——在同一個迴圈裡順便把 $p = 2^{16-k}$ 累乘出來，最後 $mask = p-1$。$k=0$ 時 p 自加 16 次溢位成 0，$mask = -1 =$ `0xFFFF`，恰好是「不遮」✓。

```
// rightshift.asm -- 邏輯右移 = 右旋 + 清高位
    @R0
    D=M
    @R2
    M=D
    @16
    D=A
    @R1
    D=D-M           // D = 16 - k
    @n
    M=D
    @p
    M=1             // p 將累積成 2^(16-k)
(LOOP)
    @n
    D=M
    @DONE
    D;JLE
    @p
    D=M
    M=D+M           // p 翻倍（與旋轉同步）
    @R2
    D=M
    @NEG
    D;JLT
    @R2
    M=D+M
    @DEC
    0;JMP
(NEG)
```

```
    @R2
    M=D+M
    @R2
    M=M+1
(DEC)
    @n
    M=M-1
    @LOOP
    0;JMP
(DONE)
    @p
    D=M-1           // mask = p - 1 = 低 16-k 位全 1
    @R2
    M=D&M           // 擦掉繞回來的高 k 位
(END)
    @END
    0;JMP
```

驗證範例：1111111001111111 右移 1：右旋 1 得 1111111100111111，遮罩 $2^{15}-1$ 清掉 MSB → 0111111100111111 ✓。

行業註腳（題目原話）：左右移是最常用的位元運算之一，硬體實作又便宜，所以幾乎所有 CPU 都有移位指令——Hack 沒有，純粹是為了讓 ALU 保持簡單。

3　Collatz 猜想（collatz.asm）

題目

迭代規則：正整數 x，奇數 $\to 3x+1$、偶數 $\to x/2$；例如 $5 \to 16 \to 8 \to 4 \to 2 \to 1$。Collatz 猜想：不管從哪個正整數出發，最終都會回到 1（重要的未解數學問題）。寫 `collatz.asm`：從 `RAM[0]` 開始跑到 1，把途經的每個數依序寫入 `RAM[32]` 起的記憶體（`RAM[0..31]` 留作變數）。例：`RAM[0]`$= 5 \Rightarrow$ `RAM[32..37]` $= 5, 16, 8, 4, 2, 1$。提示：除以 2 用右移比天真做法快得多。

解答

把數學規則翻成 Hack 的三塊拼圖：

- **奇偶判斷**：$x \,\&\, 1$（第一部分的遮罩!）；
- $3x+1$：沒有乘法指令，但 $3x+1 = x+x+x+1$——A 停在 `@x` 上連做三次記憶體運算即可；
- $x/2$：x 是**正偶數**，右移一位 = 右旋一位 = **左旋 15 位**（LSB 是 0，不會有位繞進 MSB，結果保持正數 ✓）。這就是提示說的「右移比天真做法快」：天真除法（迴圈減 2 計數）是 $O(x)$——$x = 30000$ 時要一萬五千圈；左旋 15 位固定 15 圈，快上千倍。

```
// collatz.asm -- 從 RAM[0] 迭代到 1,
//                軌跡寫入 RAM[32], RAM[33], ...
    @R0
    D=M
    @x
    M=D
    @32
    D=A
    @ptr
    M=D             // 輸出指標
(STORE)
// RAM[ptr++] = x（間接定址）
    @x
    D=M
    @ptr
    A=M
    M=D
    @ptr
    M=M+1
// x == 1 → 結束
    @x
    D=M-1
    @END
    D;JEQ
// 奇偶: x & 1
    @x
    D=M
    @1
    D=D&A
    @EVEN
    D;JEQ
// 奇數: x = 3x + 1（A 停在 x 上連續運算）
    @x
    D=M
    M=D+M           // 2x
    M=D+M           // 3x
    M=M+1           // 3x + 1
    @STORE
    0;JMP
(EVEN)
// 偶數: x >>= 1, 用左旋 15 位實作
    @15
    D=A
    @r
```

```
    M=D
(ROT)
    @x
    D=M
    @RNEG
    D;JLT
    @x
    M=D+M           // MSB=0: 加倍
    @RNEXT
    0;JMP
(RNEG)
    @x
    M=D+M           // MSB=1: 加倍後補 LSB
    @x
    M=M+1
(RNEXT)
    @r
    MD=M-1
    @ROT
    D;JGT
    @STORE
    0;JMP
(END)
    @END
    0;JMP
```

追蹤 RAM[0]= 5： 5（奇）→ 16 → 8 → 4 → 2 → 1，依序寫入 RAM[32..37] = 5, 16, 8, 4, 2, 1 ✓。**溢位提醒：** Hack 只有 16 位元有號數（最大 32767）。$3x+1$ 一步就可能把 $x > 10922$ 的奇數推出範圍——測試時挑軌跡峰值不超過 32767 的輸入（例如 5、6、7、27 的峰值 9232 安全）。題目說「這類數值驗證是早期電腦最常見的應用之一」——你正在重演 1950 年代數學家用真空管電腦做的事，只是他們的字組更寬一點。

4　快速乘法（Fast Multiplication）

題目

第一部分的簡單乘法把 m 位數乘 n 位數要 $O(2^m)$ 時間（重複加法的圈數 = RAM[0] 的**數值**）。能改進到 $O(mn)$ 嗎？提示：採用你小學學過的「直式乘法」，位元運算會有幫助！

解答

二進位直式乘法（shift-and-add）。十進位直式乘法：把被乘數依序乘上每一位數字、錯位相加。二進位更簡單——每位數字只有 0 或 1，「乘上一位」變成「**要或不要**把（錯位後的）被乘數加進結果」：

$$a \times b \;=\; \sum_{i=0}^{15} b_i \cdot (a \ll i)$$

掃描 b 的每一位（遮罩自加，第一部分 popcount 的老朋友），同步把 a 的「錯位副本」翻倍；該位是 1 就把副本加進結果：

```
// fastmult.asm -- R2 = R0 * R1, 固定 16 圈
    @R2
    M=0
    @R1
    D=M
    @sh
    M=D             // sh = R1 的錯位副本（每圈翻倍）
    @mask
    M=1             // 掃描 R0 的位
(LOOP)
// if (R0 & mask) R2 += sh
    @mask
    D=M
    @R0
    D=D&M
    @SKIP
    D;JEQ
    @sh
    D=M
    @R2
    M=D+M
(SKIP)
// sh 翻倍（錯一位）
    @sh
    D=M
    M=D+M
// mask 翻倍；溢位成 0 = 16 位掃完
    @mask
    D=M
    MD=D+M
    @LOOP
    D;JNE
(END)
    @END
    0;JMP
```

複雜度對比：

	第一部分 `mult.asm`	本題 `fastmult.asm`
迴圈圈數	`R0` 的數值（最多 2^{15}）	**固定 16**
漸進時間	$O(2^m)$	$O(mn)$（16 圈 × $O(m)$ 位加法）
300×300 實測	300 圈	16 圈

附贈的正確性：這個演算法對**負數**也對——2's complement 乘法 mod 2^{16} 的位圖案與無號乘法相同（第二週的「加法器不分正負」在乘法的延伸）。硬體乘法器（例如 Hack 沒有的那顆）本質上就是把這 16 圈展開成 16 層加法器陣列——你剛寫的程式就是硬體乘法器的軟體轉世。

5　Rogue（rogue.asm）

題目

用 Hack 組語寫出 Rogue 的遊戲場 `rogue.asm`：23 列 × 64 欄的格子，每格寬 8 像素、高 11 像素（剩下 3 條像素列塗黑）。開機時 `@` 畫在左上格；按方向鍵時整個 `@` 往該方向移動一格，除非會超出格線。提示：ROM 位址可以像一般值一樣存進變數再跳過去——用變數 `bookmark` 存「回來的地方」，就能實作簡陋的「函式呼叫」。

解答

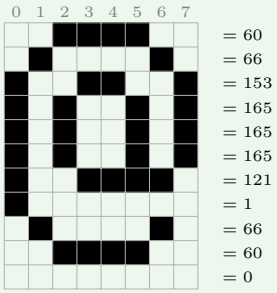
第一步：把幾何算清楚。

- 橫向：64 欄 × 8 px = 512 ✓ 整好一個螢幕寬。一個字組 16 px = **兩格**：偶數欄住低位元組（bit 0–7）、奇數欄住高位元組（bit 8–15）——LSB 在左的規則決定了這個分配；
- 縱向：23 列 × 11 px = 253，剩 $256 - 253 = 3$ 條像素列（253–255）塗黑；
- 格 (r, c) 的第 0 條像素列住在

$$addr = 16384 + 32 \times (11r) + \lfloor c/2 \rfloor,$$

  之後每條像素列 +32，共寫 11 次。

第二步：`@` 的點陣圖。 8 寬 × 11 高，**最左像素對應 bit 0**（每列的值 $= \sum_{\text{黑像素}c} 2^c$）：



（若想完全複製講義圖 2 的字形，照同樣方法逐列轉換即可——方法不變，只是 11 個常數不同。）

第三步：架構與 bookmark「函式」。 畫格子的程式碼要用兩次（擦舊、畫新），抄兩份太蠢——用題目教的技巧把它包成「函式」`PUTCELL`：呼叫前把**返回點的 ROM 位址**（標籤當值用!）存進 `bookmark`，函式結尾 `@bookmark / A=M / 0;JMP` 跳回去。

變數	用途
crow, ccol	@ 目前的格座標
nrow, ncol	試圖移往的格座標
drow, dcol	方向鍵解出的位移
erase	PUTCELL 模式：0 畫圖、1 擦除
bookmark	「函式」的返回位址
RAM[40..50]	@ 字形的 11 列點陣

方向鍵的掃描碼（Hack 字元集）：左 = 130、上 = 131、右 = 132、下 = 133。「每按一次動一格」用**等待放開**實現（比 checker3 的 prev 更簡單的邊緣觸發法）。

完整程式：

```
// rogue.asm -- 23x64 格 Rogue 遊戲場，方向鍵移動 @
// ============ 初始化：字形表 RAM[40..50] ============
    @60
    D=A
    @40
    M=D        // ..####..
    @66
    D=A
    @41
    M=D        // .#....#.
    @153
    D=A
    @42
    M=D        // #..##..#
    @165
    D=A
    @43
    M=D        // #.#..#.#
    @165
    D=A
    @44
    M=D
    @165
    D=A
    @45
    M=D
    @121
    D=A
    @46
    M=D        // #..####.
    @1
    D=A
```

```
    @47
    M=D         // #.......
    @66
    D=A
    @48
    M=D         // .#....#.
    @60
    D=A
    @49
    M=D         // ..####..
    @50
    M=0         // ........
// ============ 清空螢幕；底部 3 條像素列塗黑 ============
    @SCREEN
    D=A
    @addr
    M=D
(CLR)                   // RAM[16384..24479] = 0（253 列）
    @addr
    D=M
    @24480
    D=D-A
    @BLK
    D;JGE
    @addr
    A=M
    M=0
    @addr
    M=M+1
    @CLR
    0;JMP
(BLK)                   // RAM[24480..24575] = -1（3 條黑列）
    @addr
    D=M
    @24576
    D=D-A
    @CINIT
    D;JGE
    @addr
    A=M
    M=-1
    @addr
    M=M+1
    @BLK
```

```
    0;JMP
(CINIT)
// ============ @ 放到左上格 (0,0) 並畫出 ============
    @crow
    M=0
    @ccol
    M=0
    @erase
    M=0
    @ROINIT
    D=A             // 標籤當值: 返回位址存 bookmark
    @bookmark
    M=D
    @PUTCELL
    0;JMP
(ROINIT)
// ============ 主迴圈: 輪詢方向鍵 ============
(MAIN)
    @KBD
    D=M
    @MAIN
    D;JEQ           // 沒按鍵
    @key
    M=D
    @130
    D=D-A
    @LEFT
    D;JEQ
    @key
    D=M
    @131
    D=D-A
    @UPK
    D;JEQ
    @key
    D=M
    @132
    D=D-A
    @RIGHT
    D;JEQ
    @key
    D=M
    @133
    D=D-A
```

```
    @DOWN
    D;JEQ
(WAITR)                 // 非方向鍵（或移動完）：等放開
    @KBD
    D=M
    @WAITR
    D;JNE
    @MAIN
    0;JMP
(LEFT)
    @drow
    M=0
    @dcol
    M=-1
    @TRYMOVE
    0;JMP
(UPK)
    @drow
    M=-1
    @dcol
    M=0
    @TRYMOVE
    0;JMP
(RIGHT)
    @drow
    M=0
    @dcol
    M=1
    @TRYMOVE
    0;JMP
(DOWN)
    @drow
    M=1
    @dcol
    M=0
(TRYMOVE)
// 邊界檢查：0 <= nrow <= 22, 0 <= ncol <= 63
    @crow
    D=M
    @drow
    D=D+M
    @nrow
    M=D
    @WAITR
```

```
    D;JLT
    @22
    D=D-A
    @WAITR
    D;JGT
    @ccol
    D=M
    @dcol
    D=D+M
    @ncol
    M=D
    @WAITR
    D;JLT
    @63
    D=D-A
    @WAITR
    D;JGT
// 擦掉舊格（呼叫 PUTCELL，erase=1）
    @erase
    M=1
    @RETE
    D=A
    @bookmark
    M=D
    @PUTCELL
    0;JMP
(RETE)
// 座標更新、畫新格（erase=0）
    @nrow
    D=M
    @crow
    M=D
    @ncol
    D=M
    @ccol
    M=D
    @erase
    M=0
    @RETD
    D=A
    @bookmark
    M=D
    @PUTCELL
    0;JMP
```

```
(RETD)
    @WAITR
    0;JMP          // 等放開，一次按鍵只動一格
// ============ 「函式」PUTCELL ============
// 在 (crow,ccol) 畫 @（erase=0）或擦成白（erase=1）
// 結束跳回 bookmark
(PUTCELL)
// addr = SCREEN + 352*crow + ccol/2
    @SCREEN
    D=A
    @addr
    M=D
    @crow
    D=M
    @i
    M=D
(MUL352)               // 加法代替乘法：至多 22 圈
    @i
    D=M
    @MULDONE
    D;JLE
    @352
    D=A
    @addr
    M=D+M
    @i
    M=M-1
    @MUL352
    0;JMP
(MULDONE)
    @ccol
    D=M
    @c
    M=D
(HALVE)                // addr += ccol/2; odd = ccol%2
    @c
    D=M
    @2
    D=D-A
    @HDONE
    D;JLT
    @c
    M=D
    @addr
```

```
    M=M+1
    @HALVE
    0;JMP
(HDONE)
    @c
    D=M
    @odd
    M=D             // 0 = 低位元組, 1 = 高位元組
// 逐列寫 11 列
    @40
    D=A
    @gp
    M=D             // 字形指標
    @11
    D=A
    @i
    M=D
(PROW)
// v = erase ? 0 : glyph[gp]
    @gp
    A=M
    D=M
    @v
    M=D
    @erase
    D=M
    @VOK
    D;JEQ
    @v
    M=0
(VOK)
// 奇數欄: v <<= 8; 遮罩留低位元組（255）
// 偶數欄: v 不動; 遮罩留高位元組（!255）
    @odd
    D=M
    @ODDCOL
    D;JNE
    @255
    D=!A            // D = 0xFF00: 保留高 8 位
    @WMERGE
    0;JMP
(ODDCOL)
    @8
    D=A
```

```
    @j
    M=D
(SH8)                   // v <<= 8（8 次自加）
    @v
    D=M
    M=D+M
    @j
    MD=M-1
    @SH8
    D;JGT
    @255
    D=A             // D = 0x00FF：保留低 8 位
(WMERGE)
// RAM[addr] = (RAM[addr] & 遮罩) | v
    @addr
    A=M
    D=D&M           // D = 要保留的另半格
    @v
    D=D|M
    @addr
    A=M
    M=D
// 下一條像素列
    @32
    D=A
    @addr
    M=D+M
    @gp
    M=M+1
    @i
    MD=M-1
    @PROW
    D;JGT
// 返回呼叫點
    @bookmark
    A=M
    0;JMP
```

設計重點回顧：

- **讀—改—寫（read-modify-write）**：一個字組住兩格，寫自己那半格前必須先讀出整字組、用遮罩保住鄰居的半格再 OR 回去——不然每次移動都會把隔壁格擦掉；
- **乘法都用加法迴圈**：$352 \times crow$ 至多 22 圈、$ccol/2$ 至多 32 圈——輸入域小，簡單勝過聰明（想快可用第 4 節的移位乘法）；

- **`bookmark` 的極限**（題目的伏筆）：這種「函式呼叫」只有**一層**——若 `PUTCELL` 裡再「呼叫」別的函式，第二次寫 `bookmark` 就把第一個返回位址 **蓋掉**了。解法是把返回位址**疊**起來——這正是第九、十週堆疊（stack）與呼叫框架的開場白。

測試流程：No animation 模式執行 → 左上角出現 `@`、底部 3 條黑線 → 點鍵盤按方向鍵：每按一次動一格、到邊界停住、移動時不留殘影 ✓。

進階討論

從這裡到真的 Rogue 還缺什麼？（1）**字元庫**：把 11 位元組的字形表擴充成整套 ASCII——這就是 Jack OS（第 11 週）裡 `Output` 類別做的事，它的字形表寫法跟你的 `RAM[40..50]` 一模一樣；（2）**真正的函式呼叫**：怪物 AI、地圖生成都需要巢狀與遞迴呼叫——bookmark 不夠用，需要堆疊（第 9 週）；（3）**亂數**：地城生成需要偽隨機數（線性同餘產生器用乘加即可，你已經會寫乘法了）。1980 年的原版 Rogue 跑在 PDP-11 上——字組同樣 16 位元，跟你手上的 Hack 機器並沒有差多遠。

A 附錄：速查表

A.1 本作業的程式

程式	功能	核心技巧
`checker3.asm`	每按一次切換圖樣	軟體升緣偵測（`prev`）
`leftshift.asm`	$x \ll k$	自加 = 左移；$k \geq 16$ 直接 0
`leftrotate.asm`	左旋 k	$x < 0$ 判舊 MSB、+1 補位
`rightrotate.asm`	右旋 k	= 左旋 $16 - k$
`rightshift.asm`	$x \gg k$（邏輯）	右旋 + 遮罩 $2^{16-k} - 1$
`collatz.asm`	Collatz 軌跡	$3x$+1 連加、除 2 = 左旋 15
`fastmult.asm`	$O(mn)$ 乘法	shift-and-add
`rogue.asm`	遊戲場 + 方向鍵	讀改寫半字組、bookmark 呼叫

A.2 移位／旋轉的恆等式

恆等式	用途
$x + x = x \ll 1$	Hack 唯一的移位原語
$\mathrm{rotl}(x) = 2x + [x < 0]$	左旋一位
$\mathrm{rotr}(x, k) = \mathrm{rotl}(x, 16 - k)$	右旋歸約
$x \gg k = \mathrm{rotr}(x, k) \wedge (2^{16-k}{-}1)$	邏輯右移
$x/2 = x \gg 1$（$x \geq 0$ 偶數）	Collatz 的快速除法
$a \times b = \sum_i b_i (a \ll i)$	shift-and-add 乘法

A.3 關鍵常數與掃描碼

常數	意義
方向鍵 130 ／ 131 ／ 132 ／ 133	左／上／右／下
`c` 鍵 = 99	小寫字母用 ASCII
$352 = 32 \times 11$	Rogue 一格列的字組跨距
$24480 = 16384 + 32 \times 253$	底部黑帶起點
255 ／ `!255`	低／高位元組遮罩
字形表 `RAM[40..50]`	60, 66, 153, 165, 165, 165, 121, 1, 66, 60, 0

A.4 參考資料

- COMSM1302 第五週講義與作業卷第一部分（遮罩、間接定址、螢幕鍵盤映射），University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），App. 5（Hack 字元集：方向鍵 130–133）與 Ch. 12（Output 類別的字形表）.
- Collatz conjecture——重要的未解問題；Randall Munroe, xkcd #710（題目引用的漫畫）.

- 移位運算與 shift-and-add 乘法：任何計算機組織教科書的乘法器章節（Harris & Harris Ch. 5）.