COMSM1302 電腦架構概論
2024 年課堂測驗一、二完整詳解

In-Class Test 1（2024/10/23）· In-Class Test 2（2024/12/12）
理論與實作全部 32 題逐題解析

University of Bristol —Overview of Computer Architecture

2026 年 7 月

試卷背景。COMSM1302 每學期有兩次課堂測驗（in-class test），各佔 100 分、限時 2 小時、各含 16 題：

- **Test 1**（第 6 週，2024/10/23）：涵蓋第 1–4 週——布林代數、卡諾圖、邏輯閘、時序邏輯（閂鎖與正反器）、暫存器與記憶體、有限狀態機、CMOS 電晶體。實作部分在 Logisim（考試版 Logiexam）中完成電路。
- **Test 2**（第 11 週，2024/12/12）：涵蓋第 5–10 週——Hack 組合語言、ISA 與微架構、編譯器（lexing ／ parsing ／ EBNF）、Hack VM 與堆疊機、函式呼叫、堆積記憶體配置。實作部分要寫 `.asm` 檔，可用 Hack assembler 與 CPU 模擬器測試，但**不可**使用 VM 模擬器。

理論部分在 Blackboard 上自動閱卷（約 50 分）、實作部分繳交檔案（約 50 分）。實作題「只看行為給分」：只要行為正確、只用允許的元件，就拿滿分；整齊度、作法不列入評分（除非題目特別說明）。

目錄

第一部分　Test 1（2024/10/23）——數位邏輯與時序電路

1　Test 1 實作部分（Practical Section，50 分）

實作規則。每題在提供的 Logisim 骨架檔（skeleton）內對應的子電路中作答；輸入／輸出腳位已放好。只能使用每題列出的元件與「Wiring」資料夾內的東西（常數、分線器等）；可以自建子電路，但子電路內部也只能用該題允許的元件。

1.1　實作第 1 題（10 分）：實作布林表達式

題目

建立一個電路實作布林表達式：

$$\neg\big((\neg A \vee B) \ \vee \ \neg(A \wedge \neg C) \ \vee \ \neg(A \vee \neg A)\big)$$

允許元件：2 輸入 AND、2 輸入 OR、NOT 閘。

解答

第一步：先化簡再接線。整條式子是 $\neg(X \vee Y \vee Z)$ 的形式，其中

$$X = \neg A \vee B, \qquad Y = \neg(A \wedge \neg C), \qquad Z = \neg(A \vee \neg A).$$

觀察 Z：$A \vee \neg A = 1$（互補律——恆真），所以 $Z = \neg 1 = 0$。Z 對 OR 沒有貢獻（同一律），整式變成 $\neg(X \vee Y)$。

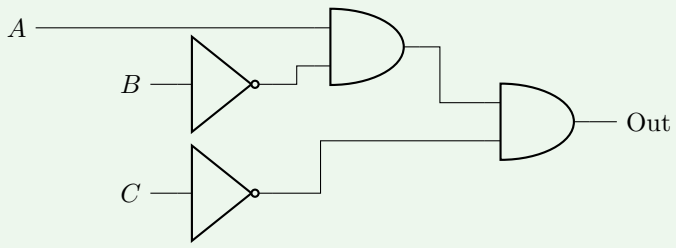
套用 De Morgan：

$$\neg(X \vee Y) = \neg X \wedge \neg Y = \neg(\neg A \vee B) \ \wedge \ (A \wedge \neg C) = (A \wedge \neg B) \ \wedge \ (A \wedge \neg C).$$

用冪等律去掉重複的 A：

$$\boxed{\text{Out} = A \wedge \neg B \wedge \neg C}$$

電路（2 個 NOT、2 個 AND）：



真值表驗證（8 列全檢查）：Out 只在 A=1, B=0, C=0 時為 1。代回原式：$X = \neg 1 \vee 0 = 0$、$Y = \neg(1 \wedge 1) = 0$、$Z = 0$，$\neg(0 \vee 0 \vee 0) = 1$ ✓。再抽查一列，例如 A=1, B=0, C=1：$Y = \neg(1 \wedge 0) = 1 \Rightarrow \text{Out} = \neg(\cdots \vee 1) = 0$ ✓。

陷阱與延伸

- **保險策略**：如果不放心化簡，可以「照式子直譯」接線——題目允許的三種閘足以直譯整條式子（約 9 個閘）。評分只看行為，兩種做法都是滿分；但化簡後的電路接線少、出錯機率低，時間壓力下反而更安全。化簡完**務必**用真值表抽查幾列，確認沒有代數失誤。
- **考點**：$A \vee \neg A = 1$（恆真式）與 De Morgan 定律 $\neg(x \vee y) = \neg x \wedge \neg y$ 是第 1 週的核心內容。出題者故意塞一個恆真子式，測試你能不能看出它可以整段刪掉。

1.2　實作第 2 題（10 分）：依真值表建立電路（至多 6 個閘）

題目

建立一個實作下列真值表的電路，**至多使用 6 個邏輯閘**。用超過 6 個閘的正確電路只拿部分分數（依閘數遞減）。

A	B	C	D	Out
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

允許元件：任何 2 輸入以下的邏輯閘。

解答

關鍵觀察：Out 與 A 無關！ 把 A=0 的前 8 列與 A=1 的後 8 列並排比較：兩組對 (B, C, D) 的輸出完全相同 $(1, 0, 1, 0, 1, 1, 0, 0)$。於是這其實是三變數問題 $f(B, C, D)$。

三變數卡諾圖（列 = B，行 = CD 格雷碼順序）：

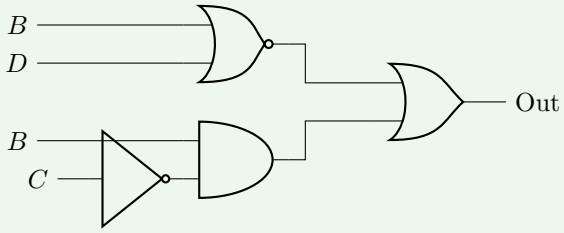
	CD=00	01	11	10
B=0	1	0	0	1
B=1	1	1	0	0

- 綠色群：B=0 列上 D=0 的兩格（CD=00 與 10，左右迴繞相鄰）⇒ $\neg B \wedge \neg D$；
- 橘色群：B=1 列上 C=0 的兩格（CD=00 與 01）⇒ $B \wedge \neg C$。

$$\text{Out} = (\neg B \wedge \neg D) \vee (B \wedge \neg C)$$

只用 4 個閘的實作（NOR 把「兩個 NOT ＋一個 AND」壓成一顆——題目允許任何 2 輸入閘）：

$$\neg B \wedge \neg D = \neg(B \vee D) = B \downarrow D \quad \text{(De Morgan)}$$



閘數：NOR + NOT + AND + OR = 4 ≤ 6 ✓。（如果只想用 AND ／ OR ／ NOT：$\neg B$、$\neg D$、$\neg C$、兩個 AND、一個 OR，正好 6 個，也在限制內。）

驗證（抽查全部 8 種 (B, C, D)）：000→1✓、001→0✓、010→1✓、011→0✓、100→1✓、101→1✓、110→0✓、111→0✓。A 腳位留空不接即可（skeleton 已有的輸入腳位不需要全部用上）。

陷阱與延伸

解題流程建議：16 列的真值表先別急著開四變數卡諾圖——掃一眼「有沒有變數其實沒作用」（上下半相同 ⇒ 與 A 無關；奇偶列相同 ⇒ 與 D 無關……）。少一個變數，卡諾圖從 16 格變 8 格，分群、接線都省一半。這是出題者安排好的「捷徑」，也是 6 個閘限制能輕鬆達成的原因。

1.3　實作第 3 題（15 分）：8 字 × 16 位元 RAM

題目

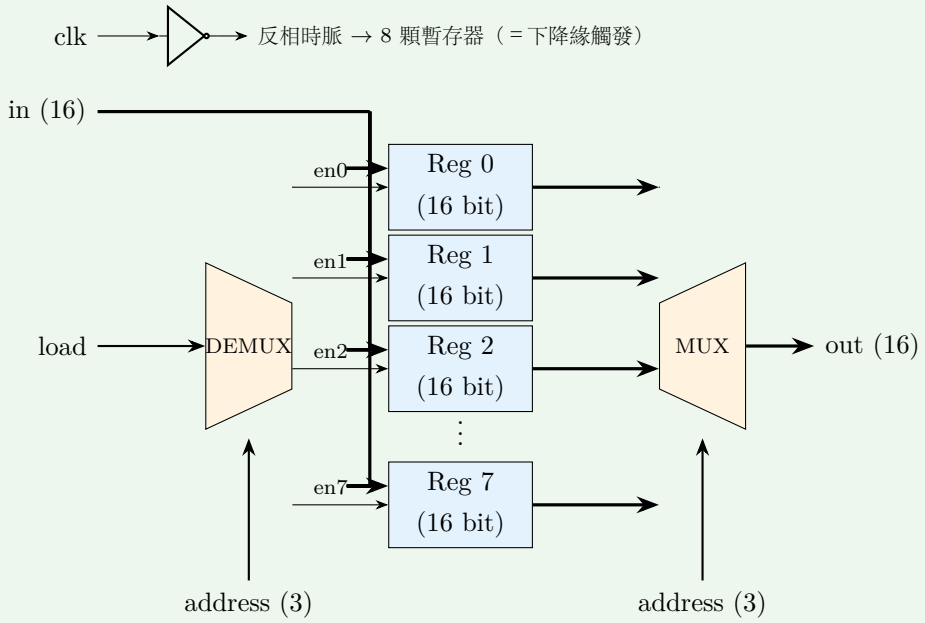
建立一個 8 字（word）RAM 電路，字長 2 位元組（16 位元）。電路內 *out* 必須輸出 *address* 所指的儲存值，且 *address* 一改變，*out* 就要立即更新（非同步讀取）。在**時脈下降緣**，若 *load* 為高，把 *address* 所指的儲存值設為 *in*。

允許元件：任何 2 輸入以下邏輯閘、暫存器（registers）、多工器（multiplexers）、解多工器（demultiplexers）。

解答

整體結構： 8 個字 ⇒ 位址 3 位元（$2^3 = 8$）；字長 16 位元 ⇒ 每個字用一顆 16 位元暫存器。RAM = 「暫存器陣列 + 寫入選路 + 讀取選路」，這正是第 3 週「從暫存器到 RAM」的標準構造：

- **寫入路徑（DEMUX）：** *in*（16 位元）同時接到**全部** 8 顆暫存器的 D 輸入；*load* 訊號經過一個 1-to-8 解多工器，由 *address* 選擇送往哪一顆暫存器的 enable（load）腳。這樣每個時脈緣最多只有被選中的那顆暫存器會吃進新值。
- **讀取路徑（MUX）：** 8 顆暫存器的輸出接到一個 8-to-1 多工器（select = *address*），輸出即 *out*。多工器是**純組合邏輯**，所以位址一變、*out* 立即跟著變——正好滿足「不等時脈就更新」的要求 ✓。
- **下降緣觸發：** 兩種作法擇一——(1) 在 Logisim 中把每顆暫存器的 Trigger 屬性改成 Falling Edge；或 (2) 保持上升緣觸發，但把時脈先過一個 NOT 閘再接進暫存器（時脈反相後，原本的下降緣變成上升緣）。兩者行為等價，元件都在允許清單內。



Logisim 實作細節：

- 暫存器 Data Bits 設 16；DEMUX ／ MUX 的 Select Bits 設 3（Logisim 內建的多工器可以直接設定，不必自己用 2-to-1 疊出 8-to-1——但用 7 顆 2-to-1 疊成樹也可以）。
- 暫存器的 enable 腳（Logisim 暫存器上的 en）接 DEMUX 的對應輸出；DEMUX 沒被選中的輸出為 0，所以其他暫存器保持原值。
- **為什麼讀取不受時脈影響？** 讀取路徑上完全沒有時序元件——暫存器的 Q 端隨時可讀，MUX 又是組合邏輯，所以 *address* 改變後經過閘延遲 *out* 就更新。「更新儲存值」才需要等下降緣。

陷阱與延伸

常見錯誤：(1) 把 *load* 直接接到所有暫存器——會 8 個字同時被寫入；*load* 必須經過 DEMUX「解」到正確的那一顆。(2) 忘了處理下降緣——Logisim 暫存器預設上升緣觸發，照預設接線行為就差半個週期。(3) 把 *in* 也接 DEMUX——不需要：資料匯流排可以廣播給所有暫存器，真正決定「誰吃進去」的是 enable 訊號。這種「資料廣播、控制選路」的模式正是真實 RAM（wordline 選列、bitline 送資料）的縮影。

1.4　實作第 4 題（15 分）：可程式脈寬產生器

題目

建立一個電路：2 位元輸入 *in*、1 位元輸出 *out*。從某個時脈上升緣開始，*out* 要維持高電位 *in* 個完整時脈週期，然後轉為低電位 1 個完整週期。若 *out* 為高時 *in* 改變，*out* 必須在下一個上升緣轉低、並保持低 1 個完整週期。

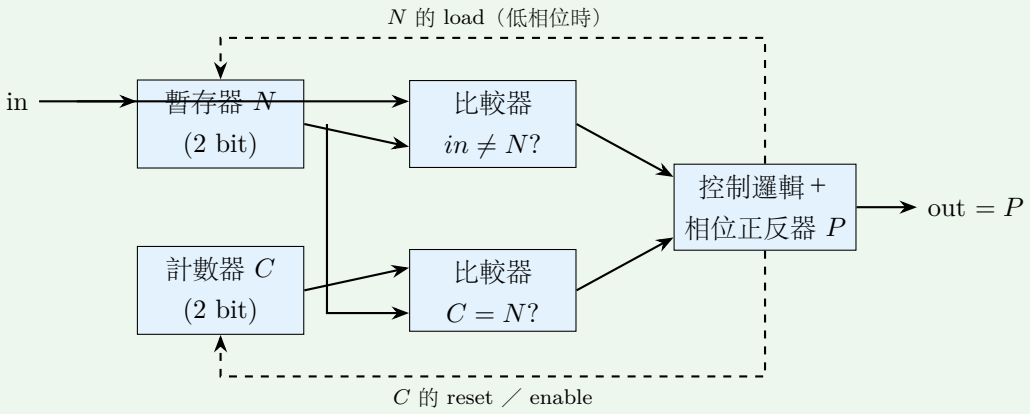
允許元件：全部元件皆可使用。

解答

解讀規格。這是一個週期性的「高 n 拍、低 1 拍」脈寬產生器（n = 開始輸出時取樣到的 *in*）：

- 進入「高相位」時把 *in* 拍照存起來（記作 N）——之後比較用的是這張快照；
- 高相位維持 N 個完整週期後，轉入「低相位」正好 1 個週期，然後重新取樣 *in*、開始下一輪；
- **中止規則**：高相位期間若目前的 $in \neq N$，下一個上升緣立即轉入低相位（也是 1 個完整週期），再照常重新開始；
- $N = 0$ 的邊界情況：「高 0 拍」= 根本不拉高，*out* 持續為低，每拍重新取樣直到 $in \neq 0$。

資料路徑＋狀態機設計（全部元件可用，所以放心用 Logisim 內建的 Register、Counter、Comparator）：



每個上升緣的轉移規則（P = 相位正反器，$P = 1$ 表示高相位，*out* 直接輸出 P，是 Moore 式輸出）：

目前 P	條件	動作（下一狀態）
1（高）	$in \neq N$（in 變了）	$P \leftarrow 0$（中止）
1（高）	$C = N$（拍數滿）	$P \leftarrow 0$
1（高）	其他	$C \leftarrow C+1$，P 維持 1
0（低）	$in \neq 0$	$N \leftarrow in$、$C \leftarrow 1$、$P \leftarrow 1$
0（低）	$in = 0$	維持 $P = 0$（持續重新取樣）

為什麼計數要「進高相位時設 C=1、C=N 時離開」? 進入高相位的那個上升緣本身就開始了第 1 個高電位週期，所以 C 記「目前正在進行第幾拍」；當 $C = N$ 的那拍結束（下一個上升緣）正好滿 N 個完整週期，此時轉低 ✓。低相位不需要計數器——它固定只有 1 拍，下一個上升緣一定離開（除非 $in = 0$ 持續為低）。

元件層面： $in \neq N$ 用兩顆 XOR（逐位元比較）+ OR；$C = N$ 用兩顆 XNOR + AND；P 用一顆 D 正反器，D 端接上表的組合邏輯；C 用 2 位元計數器（load = 進高相位、值 01；enable = P∧ 未離開）。由於「全部元件皆可」，也可以直接用 Logisim 的 Counter 與 Comparator 元件，接線更少。

陷阱與延伸

規格陷阱： (1)「in 個**完整**週期」——輸出必須在上升緣切換，不能用組合邏輯讓它中途變化；這就是為什麼 out 要從正反器（Moore 輸出）出來，而不是直接由比較器出來（會有毛刺 glitch）。(2) 中止後「保持低 1 個完整週期」——中止與正常結束走**同一個**低相位路徑，所以不需要額外狀態。(3) in 是 2 位元，N 最大 3：計數器 2 位元就夠，不會溢位。(4) 記得把 in 的快照存進暫存器再比較——若直接拿活的 in 當目標拍數，「in 中途改變」和「拍數改變」就分不開了。

2　Test 1 理論部分（Theory Section，50 分）

2.1　理論第 1 題（4 分）：補完真值表

題目

補完下列布林表達式的真值表：$A \wedge B$、$\neg(A \vee B)$、$A \oplus B$、$A \to B$（每格填 0 或 1）。

解答

A	B	$A \wedge B$	$\neg(A \vee B)$	$A \oplus B$	$A \to B$
0	0	0	**1**	0	**1**
0	1	0	0	**1**	**1**
1	0	0	0	**1**	0
1	1	**1**	0	0	**1**

逐欄理由：

- $A \wedge B$（AND）：兩者皆 1 才是 1——只有最後一列。
- $\neg(A \vee B)$（NOR）：OR 只有 00 時為 0，取反後只有 00 列是 1。
- $A \oplus B$（XOR）：「恰好一個為 1」——01、10 兩列。
- $A \to B$（蘊涵）：唯一為 0 的情況是「前提真、結論假」（A=1, B=0）；A=0 時整條蘊涵**空虛地為真**（vacuously true）——這是最常寫錯的一欄。

2.2　理論第 2 題（4 分）：NAND 電路配對

題目

把四個 NAND 電路配對到等價的布林表達式（⟨a⟩–⟨d⟩ 各是“A AND B”、“A NOR B”、“A OR B”、“A XOR B”之一）：

a) NAND(A, A) 與 NAND(B, B) 的輸出進入第三個 NAND（共 3 閘）；

b) 5 閘的對稱網路：$\neg A$ 與 B 進一個 NAND、$\neg B$ 與 A 進一個 NAND，兩者輸出進最後的 NAND；

c) 與 a) 相同的 3 閘後面再加一個 NAND(x, x)（共 4 閘）；

d) NAND(A, B) 後接 NAND(x, x)（共 2 閘）。

解答

先記住兩個基本事實：NAND$(x, x) = \neg x$（NAND 當反相器），以及 De Morgan：$\neg(\neg A \wedge \neg B) = A \vee B$。

電路	逐步化簡	答案
a)	$\mathrm{NAND}(\neg A, \neg B) = \neg(\neg A \wedge \neg B) = A \vee B$	**A OR B**
b)	$\mathrm{NAND}\big(\underbrace{\neg(\neg A \wedge B)}_{A \vee \neg B}, \underbrace{\neg(A \wedge \neg B)}_{\neg A \vee B}\big) = \neg\big((A \vee \neg B)(\neg A \vee B)\big)$	**A XOR B**
c)	$\neg(A \vee B)$（把 a) 的 OR 再反相）	**A NOR B**
d)	$\neg\big(\mathrm{NAND}(A, B)\big) = \neg\neg(A \wedge B) = A \wedge B$	**A AND B**

b) 的補充推導：$(A \vee \neg B) \wedge (\neg A \vee B)$ 恰好是「A 與 B 相等」（XNOR）——代 $A{=}B$ 得 1、$A{\neq}B$ 得 0；最外層 NAND 再取反，得到 XOR ✓。也可以直接列真值表：$A{=}0, B{=}1$ 時內層兩個 NAND 輸出 $(0 \Rightarrow) 1 \cdot \neg(0 \wedge 1){=}1 \cdots\cdots$最後輸出 1；$A{=}B$ 時輸出 0。

記憶法（第 1 週核心）：NAND 是**功能完備**的——$\neg x = \mathrm{NAND}(x, x)$（1 閘）、AND = NAND + 反相（2 閘）、OR = 兩個反相 + NAND（3 閘）、NOR = OR + 反相（4 閘）、XOR = 4–5 閘。看閘數就能先猜個大概，再驗證。

2.3　理論第 3 題（3 分）：Hack ALU 恆等式

題目

根據 Hack ALU 規格，下列布林表達式等價於什麼？（$x + y$ 表示 16 位元二補數加法、$\neg$ 表示逐位元取反）

a) $\neg(x + \neg y)$

b) $(\neg 0 + 0)$

c) $\neg(x + \neg 0)$

解答

關鍵恆等式（二補數）：對 16 位元字 a，逐位元取反等於 $\neg a = -a - 1$（因為 $a + \neg a = 1111\ldots1_2 = -1$）。把它機械地代入即可：

a) $\neg y = -y - 1 \Rightarrow x + \neg y = x - y - 1$；再取反：$\neg(x - y - 1) = -(x - y - 1) - 1 = \boxed{y - x}$。（這正是 ALU 表中 $y - x$ 那一列的實作原理！）

b) $\neg 0 = -0 - 1 = -1$，所以 $\neg 0 + 0 = \boxed{-1}$（常數 -1，即 `0xFFFF`）。

c) $\neg 0 = -1 \Rightarrow x + \neg 0 = x - 1$；取反：$\neg(x - 1) = -(x - 1) - 1 = \boxed{-x}$（取負，ALU 表中的 $-x$）。

背景：Hack ALU 只有一個加法器與 AND，所有「減法、取負、常數 ± 1」都是用「先取反、再加、再取反」這類恆等式拼出來的；考卷附的 reference sheet 有 ALU 行為表，這題測驗你是否理解表格背後的代數而不只是背表。

2.4　理論第 4 題（4 分）：雙卡諾圖合併

題目

兩張 2 變數卡諾圖（分別是 $C{=}0$ 與 $C{=}1$ 的切片）合起來描述 A, B, C 的函數：

$C{=}0$	$B{=}0$	$B{=}1$
$A{=}0$	0	1
$A{=}1$	1	1

$C{=}1$	$B{=}0$	$B{=}1$
$A{=}0$	0	1
$A{=}1$	0	1

下列哪個布林表達式正確代表這兩張表定義的輸出？

a) $(A \wedge B \wedge C) \vee (B \wedge \neg C)$

b) $\neg(A \vee B) \wedge B$

c) $(A \wedge B) \vee (B \wedge \neg C)$

d) $B \vee (A \wedge \neg C)$

e) 以上不止一個

f) 以上皆非

解答

先列出所有為 1 的格子（記作 (A, B, C)）：$C{=}0$ 片：$(0,1,0)$、$(1,0,0)$、$(1,1,0)$；$C{=}1$ 片：$(0,1,1)$、$(1,1,1)$。

觀察分群：

- $B{=}1$ 的四格**全部**為 1（兩張表的 $B{=}1$ 整欄）⇒ 蘊涵項 B；
- 剩下唯一的 1 是 $(1,0,0)$，由 $A \wedge \neg C$ 覆蓋（它同時覆蓋 $(1,1,0)$，無妨——重疊不影響 OR）。

$$f = B \vee (A \wedge \neg C) \quad \Rightarrow \quad \boxed{\text{答案：d)}}$$

逐一排除其他選項（找一格反例即可）：

選項	測試格 (A, B, C)	選項值	正確值
a) $(A{\wedge}B{\wedge}C) \vee (B{\wedge}\neg C)$	$(0,1,1)$	$0 \vee 0 = 0$	1 ✗
b) $\neg(A{\vee}B) \wedge B$	恆為 0（$B{=}1$ 時 $\neg(A{\vee}B){=}0$）	0	— ✗
c) $(A{\wedge}B) \vee (B{\wedge}\neg C)$	$(0,1,1)$	$0 \vee 0 = 0$	1 ✗

只有 d) 正確，所以 e)「不止一個」也不成立。

方法提示：多張卡諾圖切片 = 一個三變數函數。選擇題不必自己化簡到最簡式——「把每個選項對 8 格逐一驗證、找反例排除」通常更快也更保險；特別注意 c) 與 d) 只差在 $(0,1,1)$ 這一格。

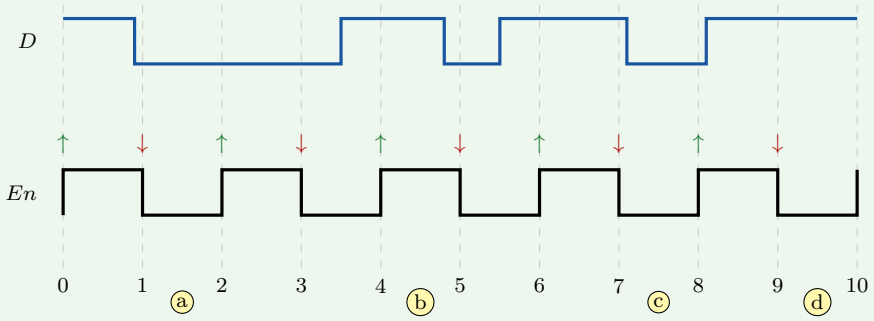
2.5　理論第 5 題（4 分）：D 正反器——上升緣 vs 下降緣

題目

時序圖給出 D 正反器的兩個輸入：資料 D 與時脈 En。問在標示的四個時間點 a=1.5、b=4.5、c=7.5、d=9.5，若正反器改為**下降緣觸發**，Q 輸出與**上升緣觸發**時相同（same）還是不同（different）？

解答

把波形整理成事件表。En 是週期 2 的方波：上升緣在 $t = 0, 2, 4, 6, 8$；下降緣在 $t = 1, 3, 5, 7, 9$。D 的變化（從圖上讀出）：起始為 1，在 $t \approx 0.9$ 落下、3.5 升起、4.8 落下、5.5 升起、7.1 落下、8.1 升起後保持到底。



在每個時脈緣取樣 D（正反器的 Q ＝最近一次觸發緣取到的 D；圖上 D 的變化都刻意錯開時脈緣，唯一要小心的是 t=7：D 在 7 之後一點點才落下，所以下降緣 7 取到的是 1）：

上升緣 t	0	2	4	6	8
取到的 D	1	0	1	1	0
下降緣 t	1	3	5	7	9
取到的 D	0	0	0	1	1

四個時間點的比較（Q ＝「最近一個觸發緣」的取樣值）：

時點	最近上升緣	Q_{rise}	最近下降緣	Q_{fall}	答案
$a = 1.5$	t=0	1	t=1	0	**different**
$b = 4.5$	t=4	1	t=3	0	**different**
$c = 7.5$	t=6	1	t=7	1	**same**
$d = 9.5$	t=8	0	t=9	1	**different**

$$\boxed{a = \text{different},\quad b = \text{different},\quad c = \text{same},\quad d = \text{different}}$$

陷阱與延伸

解題套路（第 3 週）：這類題目不要用「腦內模擬」，一律先列出**所有上升緣與下降緣的取樣值**兩行表，再逐點比對——五個緣 × 兩種觸發共 10 個值，一分鐘內填完，之後每個時間點就是查表。特別小心 D 的變化「剛好在時脈緣前後一點點」的地方：變化在緣**之前** ⇒ 取到新值；在緣**之後** ⇒ 取到舊值（本題 t=0.9 與 t=7.1 就是兩個這樣的陷阱）。

2.6　理論第 6 題（5 分）：時序邏輯是非題

題目

判斷下列敘述是否為真：

a) 主動高（active-high）R-S 閂鎖在 R=1、S=0 時 Q 會被設為 1。

b) 依課堂內容，NAND 閘可以由兩個並聯的 n 型電晶體與兩個串聯的 p 型電晶體組成。

c) 當資料存取速度是最重要的記憶體需求時，正反器比 SRAM 和 DRAM 更適合。

d) 一個在時脈上升緣開始輸出高電位的 one-shot，會在下一個**下降**緣把輸出變回低電位。

e) SRAM 中交叉耦合反相器內儲存的值必須定期刷新（refresh）以防資料流失。

解答

	答案	理由
a)	**False**	R = Reset、S = Set。R=1, S=0 是「重設」：$Q \to 0$，不是 1。（設成 1 需要 S=1, R=0。）
b)	**False**	恰好**說反了**：CMOS NAND 是 **n 型串聯**（下拉網路：A 且 B 才拉低）+ **p 型並聯**（上拉網路：A 或 B 為 0 就拉高）。題目的講法其實是 NOR 的結構。
c)	**True**	存取速度階層：正反器（暫存器）> SRAM > DRAM。正反器最貴、最耗電晶體，但讀寫最快——這正是 CPU 暫存器用正反器的原因。
d)	**False**	課堂上的 one-shot 輸出**一個完整時脈週期**的脈衝——高到**下一個上升緣**為止，不是半個週期後的下降緣。
e)	**False**	需要刷新的是 **DRAM**（電容漏電）。SRAM 的交叉耦合反相器只要供電就會互相「撐住」對方的值，不需刷新。

2.7　理論第 7 題（4 分）：最短關鍵路徑

題目

元件傳播延遲：NOT = 5 ps、AND = 10 ps、OR = 15 ps、導線可忽略。下列哪個電路的**關鍵路徑最短**？

1) OR(A, B) 與 OR$(A, \neg B)$ 進入 AND；
2) $\neg A$ 與 B 進入 AND，輸出再經 NOT；
3) AND$(A, \neg B)$ 的輸出與 B 進入第二個 AND；
4) OR$(A, \neg B)$ 的輸出與 B 進入 AND；
5) 以上不止一個。

解答

關鍵路徑＝輸入到輸出「最慢」的一條路。逐一把每條路徑的延遲加總：

電路	各路徑延遲（ps）	關鍵路徑
1)	A：OR + AND $= 25$；　B（經 NOT）：$5+15+10=30$	30 ps
2)	A：NOT + AND + NOT $= 5+10+5=$ **20**；　B：$10+5=15$	**20 ps**
3)	B（經 NOT）：$5+10+10=25$；　A：$10+10=20$	25 ps
4)	B（經 NOT）：$5+15+10=30$；　A：$15+10=25$	30 ps

答案：電路 2)（關鍵路徑$5+10+5=20$ ps）

直覺：OR 是最貴的閘（15 ps）——電路 2 是唯一**完全不用 OR** 的選項，它的三層閘全是便宜的 NOT ／ AND，總和反而比別人「兩層」還短。關鍵路徑比的是**深度加權延遲**，不是閘的數量或層數。

2.8　理論第 8 題（6 分）：十六進位加法

題目

計算下列十六進位加法，答案用 2 位十六進位（大寫字母），若算術溢位則捨棄進位位元：a) `0x9E` + `0x35`；　b) `0xDD` + `0xB1`。

解答

a) 逐位計算（base 16 直式）：

`E` + `5` $= 14 + 5 = 19 = 16 + 3$ ⇒ 寫`3`、進1；　　`9` + `3` + `1` $= 13 =$ `D`.

$$\boxed{\texttt{0x9E} + \texttt{0x35} = \texttt{0xD3}}$$

驗算（十進位）：$158 + 53 = 211 = 13 \times 16 + 3$ ✓，未超過 8 位元範圍（≤ 255），無溢位。

b) 逐位計算：

$$\texttt{D}+1=13+1=14=\texttt{E};\qquad \texttt{D}+\texttt{B}=13+11=24=16+8\ \Rightarrow\ \text{寫}\texttt{8}\text{、進}1.$$

總和是 `0x18E`（$221+177=398>255$，溢位）——依題意**捨棄進位**，保留低 2 位：

$$\boxed{\texttt{0xDD}+\texttt{0xB1}=\texttt{0x8E}}$$

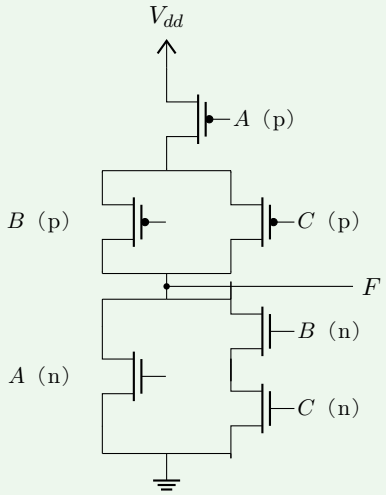
2.9　理論第 9 題（5 分）：CMOS 電晶體電路

題目

下圖的電晶體電路實作了哪個邏輯函數 F？（電路：V_{dd} 下方一顆閘極接 A 的 p 型電晶體，其下接兩顆**並聯**的 p 型（閘極 B、C）到輸出 F；F 到地之間是一顆閘極 A 的 n 型**並聯**「閘極 B、C 的兩顆 n 型**串聯**」。）選項：$F=\neg A\wedge(\neg B\vee\neg C)$、$F=\neg A\vee(\neg B\wedge\neg C)$、$F=(A\wedge B)\vee(A\wedge C)$、$F=A\vee(B\wedge C)$、$F=\neg(A\wedge B)\vee\neg(A\wedge C)$、不止一個、皆非。

解答

CMOS 判讀口訣（第 4 週）： p 型在**閘極 $=0$ 時導通**、拉**高**（連 V_{dd}）；n 型在**閘極 $=1$ 時導通**、拉**低**（連 GND）。**串聯＝ AND（兩顆都導通才通）、並聯＝ OR**。



讀上拉網路（何時 $F=1$）： $V_{dd}\to F$ 的通路＝「A 的 p」**串聯**「B 的 p **並聯** C 的 p」。p 型在 0 導通，所以通路導通條件：

$$(A=0)\wedge\big((B=0)\vee(C=0)\big)\quad\Longrightarrow\quad F=\neg A\wedge(\neg B\vee\neg C).$$

用下拉網路交叉驗證（何時 $F=0$）：「A 的 n」**並聯**「B、C 的 n **串聯**」，n 型在 1 導通：$A\vee(B\wedge C)$。取反：$\neg(A\vee(B\wedge C))=\neg A\wedge(\neg B\vee\neg C)$ ——與上拉一致（互補網路）✓。

$$\boxed{F=\neg A\wedge(\neg B\vee\neg C)\ \text{（第一個選項）}}$$

這其實就是 OR-AND-INVERT：$F=\neg\big(A\vee(B\wedge C)\big)$。

2.10　理論第 10 題（6 分）：浮點數表示

題目

某 16 位元浮點格式：4 位元指數（偏移 bias = 7）、11 位元尾數（另有 1 個符號位元）。求十進位數 $-355/64$ 的 sign、exponent、mantissa 各自儲存的二進位值（含前導 0、不含空格）。

解答

第一步：把 355/64 寫成二進位。

$$355 = 256 + 64 + 32 + 2 + 1 = 101100011_2, \qquad 64 = 2^6,$$

所以 $355/64 = 101100011_2/2^6 = 101.100011_2$（把小數點左移 6 位）。

第二步：正規化（調整成 $1.xxx \times 2^E$）：

$$101.100011_2 = 1.01100011_2 \times 2^2.$$

第三步：填三個欄位。

- **sign**：數字是負的 $\Rightarrow \boxed{1}$。
- **exponent**：實際指數 2 加偏移 7：$2 + 7 = 9 = \boxed{1001_2}$（4 位元）。
- **mantissa**：正規化後小數點後的位元 01100011，右邊補 0 湊滿 11 位：$\boxed{01100011000}$。（前導的 1 是**隱含位元**，不儲存。）

$$\text{sign} = 1, \qquad \text{exponent} = 1001, \qquad \text{mantissa} = 01100011000$$

驗算：$1.01100011_2 = 1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{128} + \frac{1}{256} = \frac{355}{256}$；乘 2^2 得 $\frac{355}{64} = 5.546875$，加上負號 ✓。

陷阱與延伸

常見失分點：(1) 忘記隱含前導 1，把尾數寫成 10110001100；(2) 指數忘了加 bias（寫成 0010）；(3) 355/64 的除以 2^6 其實就是「小數點左移 6 位」——不需要真的做十進位小數乘 2 迭代（雖然也能得到同樣結果，考卷上手算 0.546875×2 五、六次容易出錯）。

2.11　理論第 11 題（5 分）：FSM 狀態圖選擇

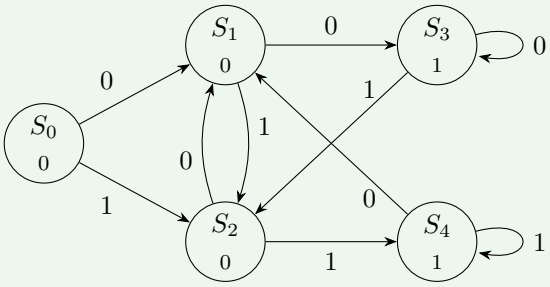
題目

一個 FSM 每次讀入一個位元（0 或 1）。若**連續**收到相同的輸入值 2 次以上，輸出為 1，否則輸出 0。六個候選狀態圖中哪個正確代表這個系統？（前兩個是 3 狀態的 Mealy 圖——邊上標輸入/輸出；後四個是 Moore 圖——圈內標輸出，其中兩個 4 狀態、兩個 5 狀態。）

解答

正確答案：第 5 個（左下角的 5 狀態 Moore 圖）。

這個系統需要記住兩件事：**上一個輸入是什麼**（0 或 1），以及**目前的連續長度是否已達 2**。最自然的 Moore 解需要 5 個狀態：



- S_0：初始（還沒讀任何東西），輸出 0；
- S_1 ／ S_2：剛看到**一個** 0 ／一個 1，輸出 0；
- S_3 ／ S_4：連續 ≥ 2 個 0 ／ ≥ 2 個 1，輸出 1，同值輸入時**自迴圈**留在原地；
- 換值時（$S_3 \xrightarrow{1} S_2$、$S_4 \xrightarrow{0} S_1$）回到「連續 1 個」的對應狀態——**不是**回 S_0！新的輸入本身就是新連續串的第一個。

其餘五個選項錯在哪（都用測試序列快速擊倒）：

選項	判定	反例
1（Mealy）	✗	邊 $S_1 \xrightarrow{1/0} S_0$ 把「剛讀到的 1」忘掉了。輸入 $0,1,1$：輸出 $0,0,0$，正確應為 $0,0,\mathbf{1}$。
2（Mealy）	✗	沒有任何自迴圈；$S_1 \xrightarrow{0/1} S_2$ 之後把「連續 0」記到「上一個是 1」的狀態去。輸入 $0,0,0$：輸出 $0,1,0$，正確應為 $0,1,\mathbf{1}$。
3（4 狀態 Moore）	✗	只有一個輸出 1 的狀態 S_3，且 $S_3 \xrightarrow{0} S_1$：連續第 3 個 0 輸出變回 0。輸入 $0,0,0$：輸出 $0,1,0$ ✗。
4（4 狀態 Moore）	✗	S_3 的兩條出邊（0 與 1）都回 S_0，連續串一到 3 就斷。輸入 $0,0,0$：輸出 $0,1,0$ ✗。
6（5 狀態 Moore）	✗	換值時回 S_0（$S_1 \xrightarrow{1} S_0$、$S_3 \xrightarrow{1} S_0$ 等）而不是對應的「連續 1 個」狀態。輸入 $0,1,1$：輸出 $0,0,0$ ✗。

由於只有第 5 個正確，「以上不止一個」也不成立。

答案：第 5 個狀態圖（5 狀態、換值走$S_3 \to S_2$，$S_4 \to S_1$ 的 Moore 機）

陷阱與延伸

通用檢驗法：對每個候選圖跑兩條測試序列——$0,0,0$（期望輸出 $0,1,1$）與 $0,1,1$（期望 $0,0,1$）。這兩條就足以殺掉本題全部五個錯誤選項；它們分別針對兩種典型設計錯誤：「連續串超過 2 之後斷掉」與「換值時忘記新輸入」。

2.12　理論第 12 題（0 分）：意見回饋題

題目

測驗中（理論或實作）是否有你不確定題意、或認為題目可能有錯的地方？若有請說明你的詮釋。本題不計分，若答案是「沒有」請留白。

解答

這是一道**安全閥**題：讓考生在自動閱卷的環境下有機會記錄「我對某題的另一種合理解讀」，閱卷者可據此人工調整。策略建議：只有在**真的**存在歧義時才寫（並具體指出題號、兩種解讀、你採用哪種與理由）；不要拿來寫求情或碎念——不影響分數，但清晰的歧義說明在爭議時可能救回分數。

第二部分　Test 2（2024/12/12）——組合語言、編譯器與 Hack VM

3　Test 2 理論部分（Section 1 — Theory，約 50 分）

3.1　理論第 1 題（2 分）：行動裝置的 ISA

題目

大多數現代手機、平板與現代 Mac 使用哪種指令集架構？選項：x64、MIPS、ARM、IA-64、Blast processing、以上皆非。

解答

ARM

ARM 是授權式的 RISC 架構：Apple（A 系列、M 系列）、Qualcomm Snapdragon、Samsung Exynos 等行動晶片都是 ARM 指令集的實作。x64（x86-64）主宰桌機／伺服器；MIPS 已边缘化（嵌入式）；IA-64（Itanium）已停產；「Blast processing」是 90 年代 Sega 的行銷用語——干擾項。考點：**ISA（指令集）與微架構（實作）分離**——同一 ARM ISA 有無數不同廠商的微架構實作。

3.2　理論第 2 題（3 分）：C 指令位元翻轉

題目

在 Hack 機器碼中，把 C 指令的**第 4 個最高有效位元**從 0 翻成 1（例如 `0xEA87` 改成 `0xFA87`）會有什麼效果？

解答

C 指令的位元布局（16 位元，由高到低）：

$$\underbrace{111}_{\text{bits 15–13}}\ \underbrace{a}_{\text{bit 12}}\ \underbrace{c_1c_2c_3c_4c_5c_6}_{\text{comp}}\ \underbrace{d_1d_2d_3}_{\text{dest}}\ \underbrace{j_1j_2j_3}_{\text{jump}}$$

第 4 個最高有效位元就是 **bit 12**——a **位元**。a 決定 ALU 的 y 輸入來源：a=0 用暫存器 `A`、a=1 用 `M`（即 `RAM[A]`）。驗證：`0xEA87` $= 1110\,1010\,1000\,0111_2$，bit 12 從 0 變 1 後為 `0xFA87` ✓。

它會把運算中所有的 A 換成 M。

其他選項為何錯：a 位元的意義**不依賴**其他位元（凡 comp 欄用到 y 輸入者一律改抓 M）；「把結果存到 M」是 dest 欄 d_3（bit 3）的事；「減 1」「清零輸入」是 comp 欄的事。

3.3　理論第 3 題（3 分）：lexing 的意義

題目

下列哪項最能描述「lexing（詞法分析）」的意義?

解答

把字串轉換成一串 token（詞元）的過程。

編譯器管線（第 8 週）：**lexing**（字元流 → token 流）→ **parsing**（token 流 → 語法樹 CST，即第一個干擾選項描述的「分析 token 序列的語法」）→ **語意分析**（建符號表——第三個干擾選項）→ 產碼。「把一種語言的檔案轉成另一種語言」是整個 **編譯／翻譯**的定義，不是 lexing 單一階段。

3.4　理論第 4 題（3 分）：VM 的用途

題目

下列哪一項**不是** VM（虛擬機）的用途?

a) 從外部分析執行中的作業系統以找出安全漏洞。

b) 在某架構上執行原生於另一架構的軟體。

c) 讓編譯器跑得更快。

d) 規範一種用來編譯高階語言的中間表示法。

e) 安全地執行不受信任的軟體，不讓它接觸底層 OS 與硬體。

解答

c) 讓編譯器跑得更快。

逐項對照第 9 週「VM 的種類與用途」：a) VM 自省（introspection）／安全研究 ✓ 是真實用途；b) 跨架構模擬（如 Rosetta、QEMU）✓；d) **Hack VM 本身**就是「用 VM 定義 IR」的例子 ✓；e) 沙箱（sandboxing）✓。c) 恰恰相反——多一層 VM 的**目標碼執行**通常更慢，且 VM 與「編譯器本身的執行速度」無關；以 VM 為目標可能讓編譯器**更容易寫**（可移植性），但不是「跑得更快」。

3.5　理論第 5 題（5 分）：編譯器是非題

題目

判斷真偽：

a) LLVM 是一個常用的 parser。

b) 實務上程式設計師很少自己手寫 parser，多半用軟體自動產生。

c) 符號表是編譯器使用的資料結構，不會出現在編譯器產生的程式碼中。

d) Cross-compilation（交叉編譯）是替新語言或新架構寫第一個編譯器的方法：先用組合語言寫一個原型，再用原型去編譯一個功能更完整的高階語言版編譯器。

e) 系統的指令集架構（ISA）規範系統對指令 **如何回應（行為）**，而非系統在硬體上如何實作。

解答

	答案	理由
a)	**False**	LLVM 是**編譯器基礎設施**（IR ＋最佳化＋後端產碼框架），不是 parser。parser 是各語言前端（如 Clang）的一部分。
b)	**True**	課堂立場：寫 parser 又煩又容易錯，實務上普遍使用 **parser generator**（yacc ／ bison、ANTLR 等）從文法自動產生。（真實世界確有大型編譯器手寫遞迴下降 parser，但依課堂敘述本題為真。）
c)	**True**	符號表是**編譯期**的工具——identifier → 位址／段的對照；產出的機器碼裡只剩下數字位址，符號表本身不隨附。
d)	**False**	題目描述的是 **bootstrapping**（自舉）。Cross-compilation 是「在 A 架構的機器上編譯出給 B 架構執行的程式」。兩個詞刻意互換，經典陷阱。
e)	**True**	這正是 ISA 的定義：對程式設計師可見的 **行為契約**（指令、暫存器、記憶體模型）；怎麼實作（管線、快取、電晶體）是微架構的事。

3.6　理論第 6 題（5 分）：編譯到 Hack VM 是非題

題目

從高階語言編譯到 Hack VM 時，判斷真偽：

a) `static` 段應該用來存全域變數。

b) `that` 段應該用來存指向目前物件的指標。

c) 編譯函式呼叫時，產生的 Hack VM 碼會用 `pointer`、`this` 和／或 `that` 明確地改變 `LCL` 與 `ARG`。

d) 即使編譯器很有效率，一行設定變數為某運算式結果的高階敘述，仍可能編譯成上千行 Hack VM 碼。

e) 你產生的 Hack VM 碼中傳給函式的引數，永遠與你所編譯的高階語言程式碼中傳入的引數相同。

解答

	答案	理由
a)	**True**	`static` 段跨函式呼叫持續存在、以檔案為單位共享——正是全域（類別層級）變數的家。
b)	**False**	「目前物件」的指標放在 `pointer 0`（THIS），透過 `this` 段存取欄位。`that`（`pointer 1`）約定用於**陣列**存取。
c)	**False**	`pointer` 段只有 0（THIS）與 1（THAT）兩格，**碰不到** `LCL` 與 `ARG`。呼叫時保存／設定 LCL、ARG 是 **VM 翻譯器**產生的組合語言在做，不是 VM 碼層級的操作。
d)	**True**	例如 `let s = "` 很長的字串`"`——每個字元都要 `push constant` + `call String.appendChar 2`；幾百字元的字串常值就是上千行 VM 碼。巨大的運算式同理。
e)	**False**	**方法（method）呼叫**會多推一個隱藏引數——物件本身（`this`）：高階寫 `obj.f(x)`（1 個引數），VM 碼是 `push obj; push x; call C.f 2`（2 個）。

3.7　理論第 7 題（5 分）：VM 堆疊運算

題目

下列 VM 操作序列執行完後，堆疊頂端的十進位值是多少?

```
push constant 5
push constant 27
push constant 2
add
pop local 0
push constant 7
sub
neg
push local 0
or
```

解答

逐行追蹤（堆疊由左到右 = 由底到頂）：

指令	堆疊	說明
`push constant 5`	[5]	
`push constant 27`	[5, 27]	
`push constant 2`	[5, 27, 2]	
`add`	[5, 29]	彈 y=2、x=27，推 $27+2$
`pop local 0`	[5]	`local 0` $\leftarrow 29$
`push constant 7`	[5, 7]	
`sub`	[−2]	$x-y=5-7$（順序！先彈的是 y）
`neg`	[2]	單元運算：$-(-2)$
`push local 0`	[2, 29]	把剛存的 29 推回來
`or`	[31]	**逐位元** OR

最後一步是本題的核心：`or` 是**位元運算**——

$$2 = 00010_2, \qquad 29 = 11101_2, \qquad 00010 \vee 11101 = 11111_2 = 31.$$

$$\boxed{\text{堆疊頂端} = 31}$$

兩個經典陷阱：(1) `sub` 的運算元順序是「第二個彈出的減第一個彈出的」($x-y$)，寫成 $7-5=2$ 就錯了；(2) `or` ／ `and` 在 Hack VM 是逐位元運算（true 用 `0xFFFF` 表示才讓位元運算兼作邏輯運算），$2 \vee 29$ 絕不是「非零就取 1」。

3.8　理論第 8 題（4 分）：效能敘述單選

題目

下列哪**一個**敘述為真?

a) 管線化（pipelining）只在多 CPU 的電腦上才有意義。

b) 使用 predication（謂詞化），常常可以把用 if 的程式碼改寫成不需要任何分支、跳躍或 goto 的形式。

c) 現代電腦夠快，永遠不必擔心從記憶體取資料的時間。

d) 迴圈展開（loop unrolling）能提升效率的唯一原因是減少控制危障（control hazards）、讓管線化更有效。

e) 在效能問題實際出現之前，先用迴圈展開之類的效能技巧通常是個好主意。

解答

b)（predication 可去除分支）為真。

Predication 把「if 條件成立才做」改成「兩邊都算、用條件遮罩選結果」（例如 `cmov` 條件搬移），消除跳躍、避免分支預測失誤——正確 ✓。

其餘為何錯：a) 管線化發生在**單一** CPU 內部（取指／解碼／執行重疊），與 CPU 數量無關。c) 恰相反——記憶體牆（memory wall）是現代效能的主要瓶頸，快取層級就是為此存在。d)「唯一」錯：迴圈展開也減少**迴圈自身的負擔**（計數、比較、跳回），並創造指令級平行與更多排程空間。e)「過早最佳化是萬惡之源」——先量測、有問題再最佳化；預先手動展開讓程式難讀且常常沒效果。

3.9　理論第 9 題（10 分）：VM 記憶體追蹤

題目

RAM 初始化為：`RAM[0]`＝ 290、`RAM[1]`＝ 270、`RAM[2]`＝ 280、`RAM[3]`＝ 3000、`RAM[4]`＝ 3005、`RAM[5]`＝ 3002、……、`RAM[2999]`＝ 3000、`RAM[3000]`＝ 3003、`RAM[3001]`＝ 3002、`RAM[3002]`＝ 3007、`RAM[3003]`＝ 3001、`RAM[3004]`＝ 3009、`RAM[3005]`＝ 3008、`RAM[3006]`＝ 2999、`RAM[3007]`＝ 3004、`RAM[3008]`＝ 3006、`RAM[3009]`＝ 3004。執行下列 Hack VM 碼：

```
push this 0
push that 2
push constant 3002
pop pointer 0
pop pointer 1
push this 3
push that 2
```

填入執行結束後 `RAM[0]`、`RAM[3]`、`RAM[290]`、`RAM[291]`、`RAM[292]` 的值（資訊不足則填 UNKNOWN）。

解答

先解碼固定位址（Hack VM 記憶體映射）： `RAM[0]` = SP（堆疊指標，指向堆疊頂**上方**一格）、`RAM[3]` = THIS（`this` 段基底）、`RAM[4]` = THAT（`that` 段基底）。初始：SP = 290、THIS = 3000、THAT = 3005。

逐行執行（`push`：寫 `RAM[SP]` 再 SP++；`pop`：SP– 再讀 `RAM[SP]`）：

指令	效果	SP
`push this 0`	推 `RAM[THIS+0]` = `RAM[3000]` = 3003 → `RAM[290]` = 3003	291
`push that 2`	推 `RAM[THAT+2]` = `RAM[3007]` = 3004 → `RAM[291]` = 3004	292
`push constant 3002`	`RAM[292]` = 3002	293
`pop pointer 0`	彈出 3002 → THIS（`RAM[3]`）= 3002	292
`pop pointer 1`	彈出 3004 → THAT（`RAM[4]`）= 3004	291
`push this 3`	**用新的 THIS**：推 `RAM[3002+3]` = `RAM[3005]` = 3008 → `RAM[291]` = 3008（覆蓋掉 3004）	292
`push that 2`	**用新的 THAT**：推 `RAM[3004+2]` = `RAM[3006]` = 2999 → `RAM[292]` = 2999（覆蓋掉 3002）	293

最終答案：

`RAM[0]` = 293	`RAM[3]` = 3002	`RAM[290]` = 3003
`RAM[291]` = 3008	`RAM[292]` = 2999	

考點三連發：（1）`pointer 0/1` 就是 `RAM[3]` ／ `RAM[4]` ——`pop pointer` 會**重定基底**，影響之後所有 `this` ／ `that` 存取；（2）`pop` 之後格子裡的舊值**不清除**，但隨後的 `push` 會覆蓋；（3）SP 最後停在 293（兩彈兩推抵銷後淨推 3 格）。

3.10　理論第 10 題（5 分）：EBNF 文法

題目

給定「摸貓噪音」的 EBNF 文法：

$$\langle \text{spicy} \rangle ::= \text{‘OUCH! ’}$$

$$\langle \text{relaxed} \rangle ::= \big(\text{‘mew ’ } \{\text{‘prrrr ’}\} \;\big|\; \text{‘prrrr ’ ‘miaow ’}\big)\ [\langle \text{spicy} \rangle]$$

$$\langle \text{catnoise} \rangle ::= \{\langle \text{relaxed} \rangle\}\ \langle \text{spicy} \rangle$$

下列哪個**不是**合法的 ⟨catnoise⟩？

a) mew prrrr prrrr prrrr OUCH!

b) prrrr miaow OUCH! OUCH!

c) OUCH!

d) mew OUCH!

e) prrrr prrrr OUCH!

f) mew mew mew prrrr prrrr OUCH!

解答

先讀懂記號：$\{x\}$ = 重複 0 次以上；$[x]$ = 出現 0 或 1 次；| = 擇一。整個 catnoise = 「若干個 relaxed，最後必以一個 spicy（OUCH!）收尾」。relaxed 的開頭只有兩種：`mew`（後面接任意多個 `prrrr`），或恰好 `prrrr miaow`。

逐項檢查：

	合法?	剖析
a)	✓	relaxed = `mew prrrr prrrr prrrr`（無 spicy 尾），再接結尾 spicy。
b)	✓	relaxed = `prrrr miaow` + 選用的 spicy `OUCH!`，再接結尾 `OUCH!`。
c)	✓	0 個 relaxed + 結尾 spicy。
d)	✓	relaxed = `mew`（0 個 prrrr、無 spicy），再接結尾 `OUCH!`。
e)	✗ **不合法**	`prrrr` 開頭的 relaxed **必須**接 `miaow`；`prrrr prrrr` 無法由任何規則導出（`prrrr` 的重複只能出現在 `mew` 之後）。
f)	✓	三個 relaxed：`mew`、`mew`、`mew prrrr prrrr`，再接結尾 `OUCH!`。

答案：e) prrrr prrrr OUCH!

（因此「不止一個」與「皆非」都不成立。）

3.11　理論第 11 題（5 分）：first-fit 記憶體配置

題目

考慮課堂上「嘗試 2」（attempt 2）：first-fit、**不含合併（coalescence）**的配置演算法。記憶體開頭如下（節錄）：

```
RAM[0x800]=0x0804  RAM[0x801]=0x0001  RAM[0x802]=0x080E  RAM[0x803]=0x0003
RAM[0x804]=0x0802  RAM[0x805]=0x000F  RAM[0x806]=0x0810  RAM[0x807]=0x0005
RAM[0x808]=0x0814  RAM[0x809]=0x00FF  RAM[0x80A]=0x0900  RAM[0x80B]=0x0003
RAM[0x80C]=0x0820  RAM[0x80D]=0x0005  RAM[0x80E]=0x0808  RAM[0x80F]=0x0001
RAM[0x810]=0x0812  RAM[0x811]=0x000A  RAM[0x812]=0x0816  RAM[0x813]=0x0003
RAM[0x814]=0xFFFF
```

若呼叫配置函式、引數為 4，回傳值是多少？（16 位元大寫十六進位、含 `0x` 前綴；若無合適空

閒段，函式回傳 `0xFFFF`；若資訊不足以判斷，填 UNKNOWN。）

解答

回憶嘗試 2 的資料結構（第 10 週）：

- 每個段的**前一個字**（`base−1`）存 **可用大小**；`alloc` 回傳可用區基底 `base`（跳過放大小的那格）；
- 空閒段串成**單向連結串列**：`RAM[base]` 存下一個空閒段的 `base`；`RAM[0x800]` 是串列開頭指標，`0xFFFF` 表示串列結尾；
- `alloc(size)`：沿串列 first-fit 找第一個大小 $\geq$ `size` 的段，移出串列、必要時在尾端切割，回傳其 `base`。

沿著串列走（head = `RAM[0x800]` = `0x804`）：

空閒段 base	大小（`RAM[base-1]`）	next（`RAM[base]`）	夠放 4 嗎?
`0x804`	`RAM[0x803]` = 3	`0x802`	$3 < 4$ ✗
`0x802`	`RAM[0x801]` = 1	`0x80E`	$1 < 4$ ✗
`0x80E`	`RAM[0x80D]` = 5	`0x808`	$5 \geq 4$ ✓ **第一个合適**
`0x808`	`RAM[0x807]` = 5	`0x814`	（不再檢查——first-fit）
`0x814`	`RAM[0x813]` = 3	`0xFFFF`（結尾）	

first-fit 在 `0x80E`（大小 5）停下：把它移出串列、把大小改成 4、在尾端切出剩餘部分，然後**回傳可用區基底**：

$$\boxed{\texttt{0x080E}}$$

整段記憶體的完整圖像（驗證讀法正確——各段首尾相接、不重疊）：

範圍	base（大小）	狀態
`0x801`–`0x802`	`0x802`（1）	空閒
`0x803`–`0x806`	`0x804`（3）	空閒
`0x807`–`0x80C`	`0x808`（5）	空閒
`0x80D`–`0x812`	`0x80E`（5）	空閒 ← **配置這段**
`0x813`–`0x816`	`0x814`（3）	空閒

段內其餘的值（如 `RAM[0x805]`=`0x000F`、`RAM[0x80A]`=`0x0900`）是先前使用留下的 **殘值（垃圾）**——不在串列走訪路徑上就沒有意義，這是本題最主要的干擾設計。另外「一堆相鄰的小空閒段沒被合併」正是嘗試 2「無 coalescence」的特徵，也提示你用對了演算法版本。

3.12　理論第 12 題（0 分）：意見回饋題

題目

與 Test 1 第 12 題相同：如對題意有疑義請說明，否則留白（不計分）。

解答

同 Test 1：只在真正存在歧義時，具體寫出題號、兩種解讀與你採用的詮釋即可。

4　Test 2 實作部分（Section 2 — Practical，50 分）

實作規則。每題交一個 `.asm` 檔（`Q1.asm`–`Q4.asm`）。可以（也鼓勵）用 Hack assembler 與 CPU 模擬器測試；**不可**使用 VM 模擬器。行為正確即滿分；建議附上簡短註解說明思路，部分分數依此判給。

4.1　實作第 1 題（10 分）：三分支條件

題目

程式讀取 `RAM[7]`，然後：若 `RAM[7]` $= 2$，設 `RAM[3]` $= 5$；否則若 `RAM[7]` < 2，設 `RAM[3]` $=$ `RAM[7]` $+25$；否則設 `RAM[3]` $=$ `RAM[7]` $\times 4$。範例：$2 \to 5$、$-5 \to 20$、$42 \to 168$。可假設 `RAM[7]` 介於 -1000 與 1000（不會溢位）。

解答

思路：算 $D =$ `RAM[7]` $- 2$ 一次，用兩個條件跳躍把三個分支分開（$D = 0$、$D < 0$、其餘）。乘 4 沒有乘法指令——用**兩次自加**（$x{+}x{+}x{+}x = ((x{\cdot}2){\cdot}2)$）；Hack ALU 沒有 `D+D`，所以借 `RAM[3]` 當工作格，用 `M=D+M` 完成翻倍。

```
// Q1.asm  -- RAM[3] = f(RAM[7])
    @7
    D=M          // D = RAM[7]
    @2
    D=D-A        // D = RAM[7] - 2
    @EQCASE
    D;JEQ        // RAM[7] == 2
    @LTCASE
    D;JLT        // RAM[7] <  2
// --- 否則: RAM[3] = RAM[7] * 4 ---
    @7
    D=M          // D = x
    @3
    M=D          // RAM[3] = x
    M=D+M        // RAM[3] = 2x   (D=x, M=x)
    D=M          // D = 2x
    M=D+M        // RAM[3] = 4x   (D=2x, M=2x)
    @END
    0;JMP
(EQCASE)         // RAM[3] = 5
    @5
    D=A
    @3
    M=D
    @END
```

```
    0;JMP
(LTCASE)        // RAM[3] = RAM[7] + 25
    @7
    D=M
    @25
    D=D+A
    @3
    M=D
(END)
    @END
    0;JMP       // 慣例：無窮迴圈結束程式
```

測試三個官方案例： x=2：走 EQCASE → 5 ✓；x=−5：$-5-2=-7<0$ 走 LTCASE → $-5+25=20$ ✓；x=42：$42-2=40>0$ 走乘 4 → 168 ✓。

陷阱與延伸

常見錯誤： (1) `@2; D=D-A` 寫成 `D=A-D`（方向反了，條件跳躍全部顛倒）；(2) 寫 `D=D+D`——**不是合法的 Hack comp**！comp 欄只有 `D+A` ／ `D+M` 等組合，翻倍必須經過 A 或 M；(3) 忘了結尾無窮迴圈——CPU 會繼續執行雜訊指令、可能改壞 `RAM[3]`。

4.2　實作第 2 題（15 分）：畫黑色三角形

題目

在螢幕上畫出黑色三角形：直角在**左下方向**延伸——第 1 列 1 個像素、第 2 列 2 個像素……第 256 列 256 個像素（左端對齊螢幕左緣；螢幕高恰為 256 列）。**程式碼至多 200 行**，否則只拿極少分數。

解答

螢幕記憶體複習（第 5 週）： 螢幕從 `SCREEN`（16384）開始，每列 512 像素 = 32 個 16 位元字；第 r 列第一個字在 `SCREEN` $+\,32r$。一個字的**最低位元（LSB）是最左邊的像素**，位元值 1 = 黑。

核心觀察： 第 r 列（0 起算）要塗 $r+1$ 個像素 = `full` 個全黑字（`0xFFFF` $=-1$）加一個「低 k 位為 1」的部分字（遮罩）。逐列走時：

$$\text{遮罩} m \;\to\; 2m+1 \quad \text{(多塗一個像素)},$$

當 m 湊滿 16 位元（$m=-1$）時：該列的「部分字」本身就是全黑字；下一列起 `full` 加 1、m 重設為 1。這樣完全不需要乘除法或位址計算技巧。

```
// Q2.asm -- 對角三角形：第 r 列塗 r+1 個像素
    @SCREEN
    D=A
    @addr       // addr = 目前列的起始位址
    M=D
    @full       // full = 本列全黑字數
```

```
    M=0
    @mask       // mask = 本列部分字（低 k 位為 1）
    M=1
    @row
    M=0
(ROWLOOP)
    @addr
    D=M
    @a          // a = 本列游標
    M=D
    @full
    D=M
    @count      // count = 還要寫幾個全黑字
    M=D
(FULLLOOP)      // 寫 full 個 0xFFFF
    @count
    D=M
    @PARTIAL
    D;JEQ
    @a
    A=M
    M=-1        // 全黑字
    @a
    M=M+1
    @count
    M=M-1
    @FULLLOOP
    0;JMP
(PARTIAL)       // 寫部分字（遮罩）
    @mask
    D=M
    @a
    A=M
    M=D
    @32         // addr += 32（下一列）
    D=A
    @addr
    M=D+M
    @mask       // mask 湊滿 16 位了嗎？（mask == -1）
    D=M
    D=D+1
    @INCFULL
    D;JEQ
    @mask       // mask = mask*2 + 1（多一個像素）
```

```
    D=M
    M=D+M
    M=M+1
    @NEXTROW
    0;JMP
(INCFULL)       // 進位：多一個全黑字、遮罩重設
    @full
    M=M+1
    @mask
    M=1
(NEXTROW)
    @row
    MD=M+1
    @256
    D=D-A
    @ROWLOOP
    D;JLT       // row < 256 繼續
(END)
    @END
    0;JMP
```

行數：約 60 行 ≪ 200 ✓。
正確性抽查：第 0 列：`full`= 0、`mask`= 1 → 1 個像素 ✓；第 14 列：`mask`= `0x7FFF`（15 個）✓；第 15 列：`mask`= `0xFFFF`（16 個 = 整字）✓，寫完觸發 INCFULL；第 16 列：`full`= 1、`mask`= 1 → 17 個 ✓；第 255 列：`full`= 15、`mask`= `0xFFFF` → 256 個 ✓。

陷阱與延伸

為什麼不能「每列逐像素畫」? 逐像素要對每個像素做「字位址 + 位元遮罩」計算，內圈邏輯長、而且 200 行限制逼你抽象化。「全黑字 + 成長遮罩」把每列工作降為 $O(\text{full})$ 次整字寫入，且遮罩演化只是 `M=D+M; M=M+1`。**陷阱**：(1) LSB 在左——遮罩要從低位往高位長，寫成 `0x8000` 往右長就左右顛倒；(2) `mask*2+1` 同樣不能寫 `D+D`（不合法 comp）；(3) 檢查 `mask == -1` 要在「寫入之後、更新之前」做，順序錯會少一個像素或多一列全黑。

4.3　實作第 3 題（15 分）：手工翻譯 VM 片段

題目

把下列 Hack VM 片段**直接翻譯**成 Hack 組合語言：

```
function Hello.world 4
push temp 0
pop local 2
```

假設目前的 RAM 內容是 Hack VM 的狀態、下一行要執行的就是 `function Hello.world 4`；採用標準記憶體映射，且**堆疊指標已更新到全域堆疊目前的頂端**。函式開頭的組語標籤請

用 (call$Hello.world)。只檢查 RAM 的最終狀態。

解答

語意回顧（第 10 週）：

- `function f k`：宣告函式進入點，並把 k 個區域變數**初始化為 0**。呼叫方的 `call` 已把 LCL 設為目前 SP，所以「配置並清零 local」= 把 0 推上堆疊 k 次；
- `push temp 0`：temp 段固定在 R5–R12，`temp 0` 就是 `RAM[5]`；
- `pop local 2`：彈出堆疊頂，寫入 `RAM[LCL+2]`——恰好是剛清零的第 3 個區域變數。

```
// Q3.asm -- function Hello.world 4 / push temp 0 / pop local 2
(call$Hello.world)
// ---- function Hello.world 4: 推 4 個 0（清零 local 0..3）----
    @SP
    A=M
    M=0
    @SP
    M=M+1
    @SP
    A=M
    M=0
    @SP
    M=M+1
    @SP
    A=M
    M=0
    @SP
    M=M+1
    @SP
    A=M
    M=0
    @SP
    M=M+1
// ---- push temp 0: temp 0 = RAM[5] ----
    @5
    D=M
    @SP
    A=M
    M=D
    @SP
    M=M+1
// ---- pop local 2: RAM[LCL+2] = 彈出值 ----
    @LCL        // 先算目的位址存到 R13
    D=M
```

```
    @2
    D=D+A
    @R13
    M=D
    @SP
    AM=M-1      // SP-- 且 A=新 SP
    D=M         // D = 彈出的值（= RAM[5] 的內容）
    @R13
    A=M
    M=D
(END)
    @END
    0;JMP
```

淨效果（只看 RAM 最終狀態，正是評分方式）：設進入時 SP $= s$、LCL $= s$（call 的約定）：`RAM[`s`]`–`RAM[`s`+3]` 清為 0，接著 `RAM[`s`+2]`（= local 2）被覆寫為 `RAM[5]` 的值；`push` ／ `pop` 抵銷後 SP $= s + 4$。

陷阱與延伸

取分要點：(1) 為什麼開頭直接推 0 就行？——題目說 SP 已在全域堆疊頂端，而 `call` 慣例保證函式進入時 `LCL` = SP，locals 就「長」在堆疊頂上；(2) `pop local 2` 的標準模式是「位址先存 `R13`」——`R13`–`R15` 保留給翻譯器當暫存正是這用途。偏好短碼者可用 `@LCL; A=M+1; A=A+1; M=D` 的指標步進法，行為相同；(3) 標籤寫成題目指定的 `(call$Hello.world)`——`$` 在 Hack 組語的符號中是合法字元，這是該課 VM 翻譯器替函式進入點取名的慣例；(4) 這段程式不含 `return`，所以**不能**動 ARG ／ THIS ／ THAT，也不要自作主張加返回序列。

4.4　實作第 4 題（10 分）：鍵盤十六進位輸入

題目

程式從鍵盤讀入一個 `0x0`–`0x7FFF` 的十六進位數（無 `0x` 前綴、A–F 大寫、以 Enter（newLine）結尾），轉成二進位存入 `RAM[500]`；每個 newLine 之後繼續讀下一個數存 `RAM[501]`，依此類推。例如輸入 ‘4’ ‘A’ ‘7’ Enter ‘9’ Enter Enter ‘5’ ‘A’ ‘C’ ‘E’ Enter，則 `RAM[500]`=`0x4A7`、`RAM[501]`=`0x9`、`RAM[502]`=`0x0`、`RAM[503]`=`0x5ACE`。鍵盤狀態每次變化之間至少間隔 500 個 CPU 週期（但不固定）。除 0–9、A–F、Enter、無按鍵外不需處理其他輸入。

解答

分析。三個子問題：

1. **按鍵邊緣偵測**：`KBD`（`RAM[24576]`）反映「目前按著的鍵」，按 500 週期會被讀到上千次——必須記住上一次的值 `prev`，只在 `KBD` ≠ `prev` 時處理（狀態**變化**），且值為 0（放開）時只更新 `prev` 不動作。
2. **字元 → 數位**：Hack 鍵盤碼：‘0’–‘9’ = 48–57、‘A’–‘F’ = 65–70、newLine = 128。數字：

$d = k - 48$；字母：$d = k - 55$（'A'$= 65 \to 10$）。判斷用 $k < 58$ 分流即可（合法輸入下唯一的分界）。

3. **累積**：每收到一個數位，$v \leftarrow 16v + d$ （十六進位的「往左推一位」）。$\times 16$ = 翻倍 4 次，用 `M=D+M` 模式實現。

```
// Q4.asm -- 鍵盤十六進位讀入 RAM[500..]
    @500
    D=A
    @ptr        // ptr = 下一個結果的存放位址
    M=D
    @val        // val = 累積中的數值
    M=0
    @prev       // prev = 上次看到的 KBD 值
    M=0
(LOOP)
    @KBD
    D=M
    @prev
    D=D-M       // KBD 變了嗎?
    @LOOP
    D;JEQ       // 沒變: 繼續輪詢
    @KBD
    D=M
    @prev
    M=D         // prev = 新值 (D 仍為 k)
    @LOOP
    D;JEQ       // k == 0: 只是放開按鍵
    @128
    D=D-A
    @STORE
    D;JEQ       // k == 128: Enter
// ---- 字元轉數位 ----
    @prev
    D=M
    @58
    D=D-A       // k - 58 >= 0 ?
    @LETTER
    D;JGE
    @prev       // 數字: d = k - 48
    D=M
    @48
    D=D-A
    @ACC
    0;JMP
```

```
(LETTER)        // 字母: d = k - 55
    @prev
    D=M
    @55
    D=D-A
(ACC)           // val = val*16 + d
    @dig
    M=D
    @val
    D=M
    M=D+M       // val = 2v
    D=M
    M=D+M       // 4v
    D=M
    M=D+M       // 8v
    D=M
    M=D+M       // 16v
    @dig
    D=M
    @val
    M=D+M       // 16v + d
    @LOOP
    0;JMP
(STORE)         // Enter: 存檔、歸零、前進
    @val
    D=M
    @ptr
    A=M
    M=D         // RAM[ptr] = val
    @ptr
    M=M+1
    @val
    M=0
    @LOOP
    0;JMP
```

用官方例子驗證： '4'（52）→ d=4, v=4；'A'（65）→ d=10, $v = 64 + 10 = 74 =$ `0x4A`；'7' → $v = 1184 + 7 =$ `0x4A7`；Enter → `RAM[500]` = `0x4A7` ✓。'9' Enter → `RAM[501]` = 9 ✓；緊接的 Enter（中間無數位）→ `RAM[502]` = 0 ✓；'5ACE' Enter → `RAM[503]` = `0x5ACE` ✓。

陷阱與延伸

失分點解剖： (1) **沒做邊緣偵測**——按一次 '4' 被累積幾百次，結果完全錯誤；「500 週期間隔」的提示就是在暗示輪詢絕對來得及，但必須偵測**變化**而不是**電位**；(2) 忘了「放開按鍵」也是一次狀態變化（KBD 變 0），若不跳過會把 0 當字元處理；(3) 連續兩個 Enter 代表「空數字」= 0

——val 歸零後直接再存一次即可，本演算法自然涵蓋；(4) 'A' 的碼是 65 不是 10——轉換偏移 55 記錯就全錯；(5) 順序陷阱：必須**先**更新 `prev` 再分支，否則 Enter 處理完回到 LOOP 又會觸發一次。

5 兩份試卷總覽與備考建議

5.1 考點分布

試卷	題目	考點（對應週次）
Test 1	實作 1	布林化簡、De Morgan、恆真式（W1）
	實作 2	真值表 → 卡諾圖 → 最簡電路（W1）
	實作 3	暫存器陣列 + MUX/DEMUX 構成 RAM（W3）
	實作 4	FSM + 計數器 + 比較器的資料路徑設計（W3–4）
	理論 1–4	真值表、NAND 等價、ALU 恆等式、卡諾圖（W1–2）
	理論 5–7	D 正反器觸發緣、時序是非題、關鍵路徑（W3–4）
	理論 8–10	十六進位加法、CMOS 網路、浮點數（W2、W4）
	理論 11	Moore ／ Mealy 狀態圖判讀（W4）
Test 2	理論 1–2	ISA vs 微架構、C 指令編碼（W7）
	理論 3–5	lexing、VM 用途、編譯器術語（W8–9）
	理論 6–7	VM 段語意、堆疊運算追蹤（W9）
	理論 8	管線、predication、記憶體牆（W7）
	理論 9	`pointer/this/that` 重定基底追蹤（W9）
	理論 10–11	EBNF 判合法、first-fit 配置（W8、W10）
	實作 1–4	Hack 組語：分支、繪圖、VM 翻譯、鍵盤 I/O（W5–10）

5.2 可通用的解題套路

1. **選擇題用反例，不用化簡。**狀態圖、布林等價、EBNF 這類「哪個正確／哪個不合法」的題目，設計 2–3 條短測試序列逐一擊倒選項，比推導最簡式快且不易錯（見 T1 理論 4、11；T2 理論 10）。
2. **時序題先列「取樣表」。**把每個時脈緣取到的 D 一次列完，再回答任何時間點的問題（T1 理論 5）。

3. **VM 追蹤題逐行畫堆疊**。標明 SP 與被覆蓋的殘值；`pop pointer` 之後的段存取一律用**新**基底（T2 理論 7、9）。
4. **Hack 組語三大慣例**：結尾無窮迴圈；`D+D` 不存在（翻倍走 `M=D+M`）；`R13`–`R15` 是 pop 目的位址的合法暫存區。
5. **鍵盤／輪詢題必問自己**：我偵測的是「電位」還是「變化」?（T2 實作 4 與第 5 週 lab 的 keypress 完全同型。）
6. **配置器題先畫段圖**。把 base、大小、串列指標畫成表格再走 first-fit，殘值垃圾就騙不到你（T2 理論 11）。