COMSM1302 電腦架構概論

第七週完整教材：ISA 與微架構

Hack 指令集架構 · Hack 微架構 · 比較架構學 · 管線化與快取

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

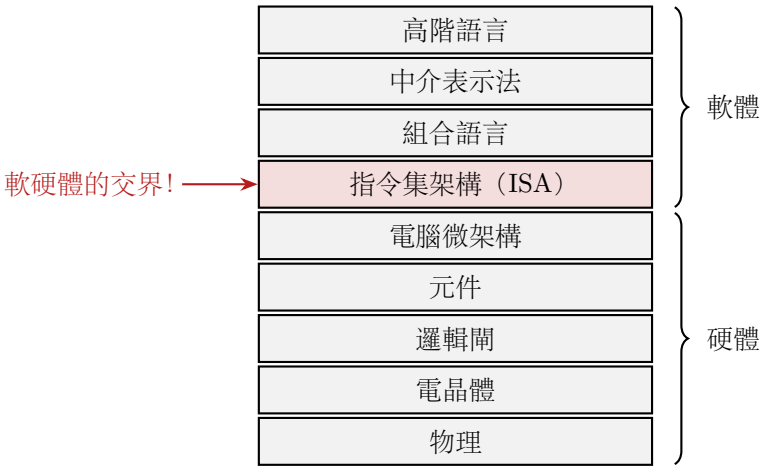
2026 年 7 月

本教材的使用方式：本講義整合了第七週四份投影片（7-1 The Hack instruction set architecture、7-2 The Hack microarchitecture、7-3 Comparative architecture、7-4 Microarchitecture optimisation: Pipelining and caching）的全部內容，並補充了 RISC vs CISC 的現代論戰、資料前遞（forwarding）、分支預測等延伸知識。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。本週把前六週的所有線索**收攏成一台完整的電腦**：機器碼格式（ISA）、CPU 的內部接線（微架構）、Hack 與真實世界架構的對照，以及現代 CPU 讓程式跑得飛快的兩大絕招——管線化與快取。

目錄

1　Hack 指令集架構（The Hack ISA）

1.1　什麼是 ISA?



定義

微架構（microarchitecture）是電腦在硬體上的**實體設計**——電路圖與 PCB 佈局。**指令集架構（instruction set architecture, ISA）**則是電腦**對機器碼指令做出反應的方式**——一份行為規格。

類比：當你依照規格實作了一個 C 函式（例如「`mult(x,y)` 應回傳 $x \times y$」），使用者**不需要記得你怎麼寫的**就能呼叫它；你也可以改寫實作，只要仍回傳 $x \times y$ 就不會引入 bug。同樣地，**微架構「實作」ISA**：寫組語時不需要知道 ISA 怎麼被實作，而且你的程式碼在**任何**符合該 ISA 的硬體上都能正確運作。例如 AMD 與 Intel 的現代 CPU 微架構天差地別，卻普遍都實作 x86-64 這一個 ISA。

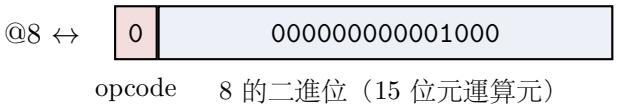
ISA 的性質	微架構的性質
字長（word length）	時脈速度
機器碼指令	能源效率
暫存器與記憶體	ALU 電路設計
I/O 記憶體映射	I/O 與 CPU 的實體連接
執行模型（如擷取—執行週期）	對未定義行為的反應（如 `A=D;JMP`）

講 Hack 組語時其實已涵蓋幾乎整個 Hack ISA——只缺**機器碼指令格式**（本節）。實驗課也已做完大部分 Hack 微架構——剩下的見第 2 章。（本週作業：在 Logisim 裡蓋出 Hack CPU!）

1.2　A-指令

A-指令（address instruction）

opcode（操作碼）說明這是哪一種指令；**operand（運算元）**是它的引數。A-指令的 opcode 是 **0**，後接**一個 15 位元運算元**，作用是把運算元複製進 A（注意這同時使 M 變成 $RAM[A]$）。

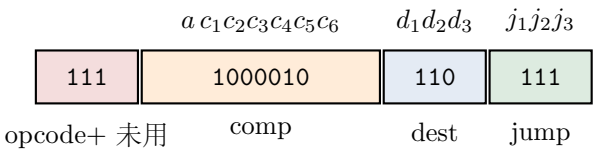


這正是第五週「@ 只能載 15 位元」的**硬體根源**：16 位元的指令扣掉 1 位元 opcode，只剩 15 位元放數值——沒有空間再多了。

1.3　C-指令

C-指令（compute instruction）

opcode 是 **1**，接著兩個**未使用位元**（慣例設 1），再接三個運算元：**comp** 指定做哪個運算、**dest** 指定結果存到哪裡、**jump** 指定是否把 PC 更新為 A。



以組語 `MD=A+D;JMP` 為例：comp 對應 `A+D`、dest 對應 `MD=`、jump 對應 `;JMP`。

1.3.1 comp 欄位（$a\,c_1c_2c_3c_4c_5c_6$）

$a=0$	$a=1$	c_1	c_2	c_3	c_4	c_5	c_6
`0`		1	0	1	0	1	0
`1`		1	1	1	1	1	1
`-1`		1	1	1	0	1	0
`D`		0	0	1	1	0	0
`A`	`M`	1	1	0	0	0	0
`!D`		0	0	1	1	0	1
`!A`	`!M`	1	1	0	0	0	1
`-D`		0	0	1	1	1	1
`-A`	`-M`	1	1	0	0	1	1
`D+1`		0	1	1	1	1	1
`A+1`	`M+1`	1	1	0	1	1	1
`D-1`		0	0	1	1	1	0
`A-1`	`M-1`	1	1	0	0	1	0
`D+A`	`D+M`	0	0	0	0	1	0
`D-A`	`D-M`	0	1	0	0	1	1
`A-D`	`M-D`	0	0	0	1	1	1
`D&A`	`D&M`	0	0	0	0	0	0
`D|A`	`D|M`	0	1	0	1	0	1

注意 a **位元的作用是在** A **與** M **之間二選一**作為輸入——同一組 c 位元，$a=0$ 用 A、$a=1$ 用 M。

1.3.2 dest 欄位（$d_1d_2d_3$）

目的地	d_1	d_2	d_3
（不存）	0	0	0
`M=`	0	0	1
`D=`	0	1	0
`DM=`	0	1	1
`A=`	1	0	0
`AM=`	1	0	1
`AD=`	1	1	0
`ADM=`	1	1	1

d_1 高 $\Rightarrow$ 結果存入 A；$d_2 \Rightarrow D$；$d_3 \Rightarrow M$。三位全低 = 不存。**一個位元管一個暫存器**——這就是為什麼多重指定（如 `AMD=`）在硬體上是「免費」的。

1.3.3　jump 欄位（$j_1j_2j_3$）

條件	j_1	j_2	j_3
（不跳）	0	0	0
`;JGT`	0	0	1
`;JEQ`	0	1	0
`;JGE`	0	1	1
`;JLT`	1	0	0
`;JNE`	1	0	1
`;JLE`	1	1	0
`;JMP`	1	1	1

jump 的語意

CPU 跳躍（把 A 存入 PC）當且僅當：j_1 高**且**運算結果為**負**；或 j_2 高**且**運算結果為**零**；或 j_3 高**且**運算結果為**正**。三個位元就是 <0、$=0$、>0 三個旗標的開關——所以七種條件恰好是 2^3-1 種非空組合，`JMP` = 三個全開。

1.4　兩個解碼範例

機器碼 → 組語

例一： `1110101010000111`。拆解：opcode `111` | comp `0101010` | dest `000` | jump `111`。查表：comp `0101010` = `0`；dest `000` = 不存；jump `111` = `;JMP`。**組語：** `0;JMP`——第五週每支程式結尾的無窮迴圈！

例二： `1111000010101000`。拆解：`111` | comp `1000010` | dest `101` | jump `000`。查表：$a=1$ 且 $c=000010$ = `D+M`；dest `101` = `AM=`；jump `000` = 不跳。**組語：** `AM=D+M`。

回去看實驗課做的 ALU 設計，比較 comp 的行為與 ALU 輸出 `out` ——你看得出 ALU 如何實作 C-指令嗎？（第 2 章揭曉。）**這些表都不用背**——考試會提供副本，當參考資料用就好！

本章重點

- ISA = 行為規格（軟硬體之間的合約）；微架構 = 實體實作。同一 ISA 可有多種微架構（AMD vs Intel 之於 x86-64）。
- A-指令：`0` + 15 位元運算元 → 複製進 A；這就是 `@` 的 15 位元限制。
- C-指令：`111` + $a\,c_1..c_6$（comp）+ $d_1d_2d_3$（dest）+ $j_1j_2j_3$（jump）。
- a 選 A 或 M；d 三位各管一個暫存器；j 三位對應負／零／正三旗標。
- 查表解碼／編碼機器碼是本週核心技能——表格不用背。

2　Hack 微架構（The Hack Microarchitecture）

2.1　任務：在 Logisim 蓋一台 Hack 電腦

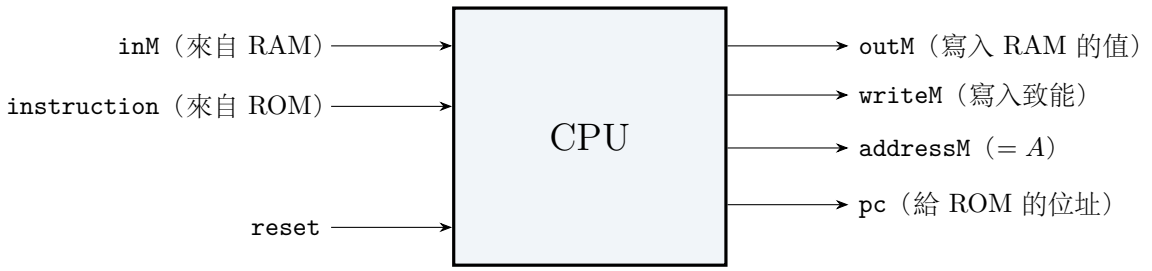
本週主要作業是在 Logisim 裡蓋出 Hack 電腦。從下週起課程將完全從硬體轉向軟體（把高階語言翻譯成組語的過程）。所需元件你都已做過，骨架檔也載入了模型版本。建議用 Logisim **內建**的 ROM、RAM 與暫存器（而非自己設計的版本），方便在模擬時檢視與編輯內容、實際測試 CPU。重點放在電腦本身——記憶體映射 I/O 接在哪裡不難想像，但不列入考試。

2.2　元件：記憶體

ROM	**RAM**
輸入：`address`；輸出：`out` 64KB、15 位元位址空間、16 位元字 `out` $= ROM[$`address`$]$，**無時脈** 存放要執行的**程式**	輸入：`address`、`in`、`load`；輸出：`out` 64KB、15 位元位址空間、16 位元字 `out` $= RAM[$`address`$]$，**無時脈** 存放**資料**；每個時脈 tick，若 `load` $= 1$ 則 $RAM[$`address`$] \leftarrow$ `in`

正常的 Hack 應是 32KB RAM、14 位元位址空間；這裡用 64KB 是因為 Logisim 不易模擬螢幕，乾脆把像素存在 $RAM[$`0x4000`$]$–$RAM[$`0x5FFF`$]$ ——用**真的記憶體**取代記憶體映射。

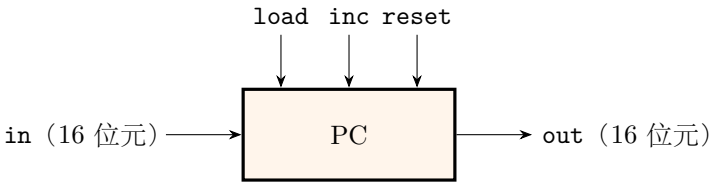
2.3　元件：CPU 的黑盒子規格



CPU 必須遵循擷取—執行週期。**每個時脈 tick 之後**：

- CPU 應把輸入 `instruction`（即 $ROM[$`pc`$]$）當機器碼執行；
- `inM` 應是 M 的值（從 RAM 載入）；
- `pc` 應是程式計數器的值；
- `addressM` 應是 A 的值；
- 若 CPU 要寫 RAM：`writeM` $= 1$ 且 `outM` 為寫入值；否則 `writeM` $= 0$；
- 若 `reset` 高，PC 歸零（A、D 不必歸零）。

2.4　CPU 元件：程式計數器（PC）



`load`、`inc`、`reset` **同時最多一個為高**。每個時脈週期：若 `reset` $= 1$，`out` $\leftarrow 0$；若 `inc` $= 1$，`out` $\leftarrow$ `out` $+ 1$；若 `load` $= 1$，`out` $\leftarrow$ `in`。（普通暫存器 = 拿掉 `inc` 與 `reset` 的同一元件——這正是第四週的計數器模式！）

在 Hack CPU 中的用法：**有跳躍發生** $\rightarrow$ `load`（載入 A）；**無跳躍** $\rightarrow$ `inc`（下一條指令）；**重開機** $\rightarrow$ `reset`。

2.5　CPU 元件：ALU

ALU **無時脈**（純組合邏輯），依控制位 zx, nx, zy, ny, f, no 從 x、y 算出 `out`：

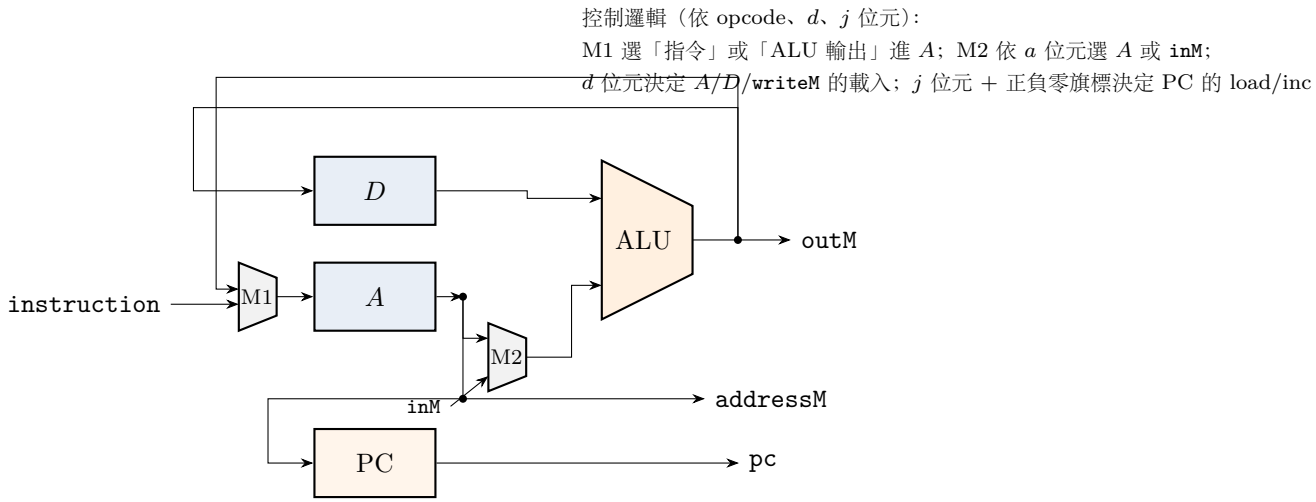
zx	nx	zy	ny	f	no	`out`
1	0	1	0	1	0	`0`
1	1	1	1	1	1	`1`
1	1	1	0	1	0	`-1`
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	$!x$
1	1	0	0	0	1	$!y$
0	0	1	1	1	1	$-x$
1	1	0	0	1	1	$-y$
0	1	1	1	1	1	$x+1$
1	1	0	1	1	1	$y+1$
0	0	1	1	1	0	$x-1$
1	1	0	0	1	0	$y-1$
0	0	0	0	1	0	$x+y$
0	1	0	0	1	1	$x-y$
0	0	0	1	1	1	$y-x$
0	0	0	0	0	0	$x \,\&\, y$
0	1	0	1	0	1	$x \mid y$

關鍵對照

把這張表與第 1 章的 comp 表並排看：**這些控制位恰好就是 C-指令的** $c_1c_2c_3c_4c_5c_6$**！**把 x 接 D、y 接（a 位元選出的）A 或 M，再把指令的 6 個 c 位元直接接上 ALU 的 6 個控制位——comp 欄位就「免解碼」地實作完成了。這不是巧合，是 Hack **ISA 與微架構協同設計**的範例。

要評估 C-指令的 jump 部分，還需要一個子電路判斷 `out` 是**正、負、還是零**（例如：符號位元給「負」、16 位元 NOR 給「零」、兩者皆非為「正」），再與 $j_1 j_2 j_3$ 做 AND-OR 組合出「是否跳躍」訊號。

2.6　把 CPU 接起來



資料流總結（Nisan & Schocken 的參考實作，**只是可能的實作之一**）：

1. **A 暫存器的輸入 MUX（M1）**：A-指令時選 `instruction`（載入常數）；C-指令且 $d_1 = 1$ 時選 ALU 輸出。
2. **ALU 的 y 輸入 MUX（M2）**：依 a 位元在 A 與 `inM` 之間選擇；x 輸入固定接 D。
3. **6 個 c 位元直接接 ALU 控制位**；ALU 輸出分送 `outM`、D、A（各由 d 位元 gating）。
4. **PC**：跳躍條件成立 → `load`（吃 A）；否則 `inc`；`reset` 輸入直通。
5. `writeM` $= d_3 \wedge (\text{opcode} = 1)$；`addressM` 恆為 A。

本章重點

- Logisim 版用 64KB RAM 直存像素，取代記憶體映射（非考試範圍）。
- ROM/RAM 讀取無時脈；RAM 寫入在時脈 tick 且 `load`=1。
- CPU 黑盒子：入 `instruction/inM/reset`，出 `outM/writeM/addressM/pc`。
- PC 元件 = 有 load/inc/reset 的暫存器；跳躍 → load、否則 inc。
- ALU 控制位 ≡ comp 的 $c_1..c_6$；a 位元選 A/M；jump 需要正／負／零判斷子電路。

3　比較架構學（Comparative Architecture）

3.1　看到新 ISA 該注意什麼?

新 ISA 檢查清單

- **字長**（「64 位元 CPU」= 64 位元字長）;
- **位址空間**大小;
- **指令長度**（通常是字長的倍數，**可能是變長的**!）;
- **設計哲學**: Harvard vs von Neumann、RISC vs CISC;
- **暫存器**（特殊用途 vs 一般用途）;
- **定址模式**;
- 可用的算術、邏輯、分支運算;
- **硬體中斷**;
- **堆疊**（本課程稍後）。

語法備註: 多數組語把**運算元放前面**，與 Hack 不同。例如 MIPS 中，Hack 的 `A=D+M` 會寫成 `add A,D,M`。這純屬外觀——一行組語仍對應一條機器碼指令。Hack 的特殊語法是因為它**只有兩種指令**!

3.2　Harvard vs von Neumann

Harvard 架構

指令與資料分開存（Hack!）

von Neumann 架構

同一記憶體（多數現代 ISA）

Hack 從 ROM 讀指令、把資料存 RAM——兩個**分開的記憶庫**，是 **Harvard 架構**的例子（主要優點: **硬體較簡單**）。多數現代 ISA 改用 **von Neumann 架構**: 指令與資料**存在同一個 RAM**。這使程式可以被當作資料載入、修改、甚至自我修改——作業系統載入執行檔的能力正建立在此。

3.3　RISC vs CISC

精簡指令集（RISC）	**複雜指令集（CISC）**
微架構簡單	微架構複雜
組語用**許多簡單指令**	組語用**少量複雜指令**
記憶體使用較不緊湊	記憶體使用較緊湊
定長指令	**變長**指令
指令大多各花 1 週期	指令花任意多週期
ISA 只有「標準」功能	ISA 有「額外」功能（如執行模式）

RISC 與 CISC 不是絕對或嚴謹的定義，而是**光譜的兩端**。現代 CPU 上 CISC 通常跑得比較快；RISC 仍廣泛用於低功耗、低成本的應用（如嵌入式硬體）。歷史上優勢曾多次易手——2000 年時 RISC 連高速應用都佔優。

延伸補充：現代觀點——「ISA 之戰」已淡化。現代高效能 CPU（無論 x86 還是 ARM）都在前端把指令 **解碼成內部的微運算（micro-ops）** 再執行，並用 micro-op 快取降低解碼成本；研究（如 Blem et al. 的 ARM vs x86 實測）顯示能效與效能主要取決於**微架構與製程**，ISA 屬於 RISC 或 CISC 本身影響甚小。新興的開放 ISA **RISC-V** 因無授權費、模組化簡潔而快速崛起。課程表格描述的是**經典設計哲學**的對比，仍是理解歷史與嵌入式現況的好框架。

3.4　暫存器：特殊用途 vs 一般用途

Hack 中：PC 是程式計數器、M 是記憶體暫存器、A 是控制 M 的位址暫存器。這些都是**特殊用途暫存器（special-purpose registers）**：在硬體中有特定角色、能參與的 ALU 運算受限，且通常是 ISA 特有的——別的 ISA 沒有 A、M 的等價物。幾乎所有 ISA 都有 PC，但許多把它稱為**指令暫存器（IR）**。

D 則相反，是**一般用途暫存器（general-purpose register）**：可在 ALU 運算中扮演任何角色。多數架構有 **32 個以上**的一般用途暫存器，它們行為全都相同——每一個都是 D 的直接類比。

3.5　定址模式（Addressing Modes）

定義

定址模式定義 ISA 如何把指令的**運算元**對應到**資料**。

Hack 具備最常見的三種：

模式	意義	例子
立即定址（immediate）	運算元**就是資料**	`@511`：把 511 當**數字** 511。ARM7：`LDR R0, #0xDEADBEEF` 把值載入 R0
直接定址（direct）	運算元是資料的**位置**	`D=A+D`：comp 運算元指名 A、D 兩個暫存器、dest 指名存入 D。ARM7：`LDR R0, 0xDEADBEEF` 把**該位址的值**載入 R0
間接定址（indirect）	運算元是**指向資料的指標**的位置	`M=D+1`：dest 的 `M` 表示存到「A 中存的位址」。ARM7：`LDR R0, [R1]` 讀 R1、再讀該位址的記憶體

CISC ISA 常有更多定址模式以提升效率。例如 ARM7 支援**索引間接定址 (indexed indirect)**：`LDR R0, [R1, #0xBEEF]` 讀 R1 中的位址、加上 `0xBEEF`、再把該位址的值存入 R0。這對**陣列**超有用——若 R1 是 C 陣列 `arr` 的位址，一條指令就完成 `R0 = arr[0xBEEF]`！

3.6　常見 ISA 巡禮

- **x64**（又名 x86-64、AMD64）：多數現代 64 位元桌機與筆電。（Intel 曾自研 64 位元的 IA-64 ／ Itanium，2019 年停產後也改用 x64。）非常偏 CISC。跑它的 CPU 通常又快又高效，但昂貴、耗電、需要大量散熱。
- **ARM**：面向可攜裝置的 64 ／ 32 位元 ISA 家族，比 x64 更偏 RISC。晶片組功耗與散熱需求低，但速度與效率較遜。幾乎所有現代手機平板都用 ARM；現代 Mac 也用 Apple 自研的 ARM 變體。背後的公司 Arm Holdings 就在**布里斯托**！
- **MIPS**：簡單的 RISC ISA 家族，用於嵌入式應用。例如 PIC 晶片單價 £3–£10、跑 15–120MHz，低階款的功耗跟一顆 LED 差不多。任天堂 N64 也跑 MIPS 變體——《超級瑪利歐 64》裡的兔子就因此取名 MIPS！

3.7　進階功能：硬體中斷（Hardware Interrupts）

回想 Hack 的輸入靠**輪詢（polling）**：讀 RAM[0x6000] 得到當下按著的鍵。**問題：輪詢很糟**！浪費大量時脈週期，而且（時脈較低時）可能**漏掉輸入**。

中斷（interrupt）

所有現代 ISA 都用**中斷**處理輸入：

1. CPU 有一支以上專用的**中斷訊號腳位**（硬體層級，例如某個鍵被按下）；
2. 收到中斷訊號時，CPU **停下手邊工作、保存當前 PC**，立刻分支到處理該中斷的程式碼；
3. 處理完後，CPU **恢復原本的 PC**，從中斷點繼續執行。

中斷也用於其他用途，例如**存取受保護記憶體**的嘗試——這就是指標亂掉時你會收到 segfault 的原因。

本章重點

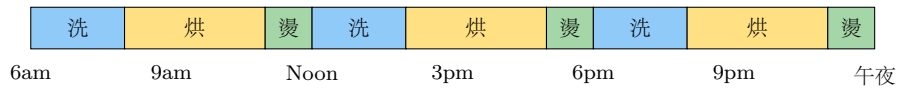
- 檢查清單：字長／位址空間／指令長度／設計哲學／暫存器／定址模式／運算／中斷／堆疊。
- Harvard（Hack：ROM+RAM 分開、硬體簡單）vs von Neumann（同一記憶體、現代主流）。
- RISC vs CISC 是光譜；現代觀點：micro-ops 讓 ISA 差異淡化，效能看微架構。
- 特殊用途（PC/A/M）vs 一般用途（D；多數 ISA 有 32+ 個）。
- 定址模式：立即（資料）、直接（資料的位置）、間接（指標的位置）；CISC 另有索引間接等。
- x64 = 桌機 CISC；ARM = 行動裝置（Bristol!）；MIPS = 嵌入式與 N64。
- 中斷取代輪詢：保存 PC → 跳處理程式 → 恢復 PC；segfault 也是中斷。

4　微架構最佳化：管線化與快取（Pipelining and Caching）

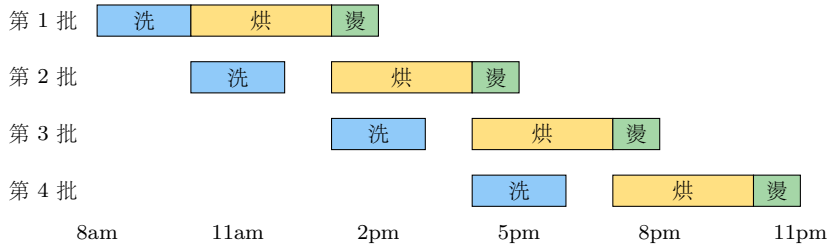
4.1　洗衣服的啟示：管線化

假設你積了好幾批衣服要洗。一批衣服：洗衣機 2 小時、烘衣機 3 小時、燙摺 1 小時。怎麼安排?

壞做法——一批做完才開始下一批（每批 6 小時，3 批 18 小時）：



好做法——第一批進烘衣機時，第二批就進洗衣機：



（課程投影片還展示了對齊到「3 小時一格」的版本——即把「換手時間點」定在固定間隔，如同時脈週期。）

管線化（pipelining）

在可平行處，讓各元件不要閒置，就能大幅提高**吞吐量（throughput）**——這就是管線化。注意：每一批衣服仍然要 6 小時（**延遲**不變），但單位時間洗完的批數大增（**吞吐量**提升）。

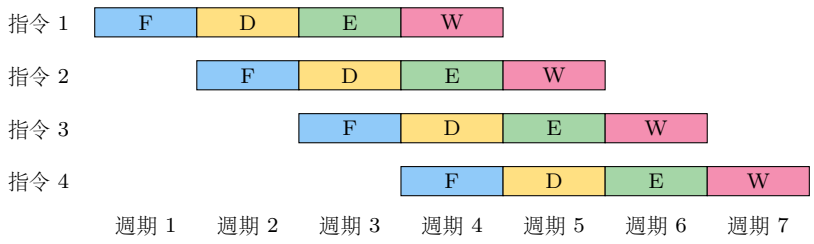
4.2　管線化的理想

提高 CPU 時脈的主要障礙是**傳播延遲（propagation delay）**：訊號穿過閘、到達穩定值所需的時間（第三週!）。

對 Hack 的 C-指令，可把擷取—執行週期分成**四個階段**（A-指令會跳過其中一些）：

Fetch（擷取）	從 $ROM[PC]$ 取下一條指令
Decode（解碼）	把 ALU 的輸入設好
Execute（執行）	讓 ALU 的輸出穩定
Writeback（寫回）	把值寫入 RAM 與暫存器、更新 PC

每個階段用**不同的硬體**，所以時脈只要配合 **最慢一個階段**的傳播延遲，而不是整個擷取—執行週期的總延遲!



穩定狀態下**每個週期完成一條指令**——即使每條指令要走四個階段。現代 CPU 的管線更複雜（常超過 20 級），但原理相同，這四個階段仍是很好的基本框架。

4.3　管線化的現實：危障與停滯

會出什麼問題？三種**危障（hazards）**：

- **資料危障（data hazard）**：某指令需要**前一條尚未算完的指令**修改的資料。例：`D=D+1` 後接 `A=A+D`——第二條需要第一條算出的新 D。
- **條件危障（conditional ／ control hazard）**：發生在**條件分支**。下一條該進管線的指令取決於分支是否發生。例：`M;JEQ` 之後，下一條指令可能在 $ROM[PC+1]$ 也可能在 $ROM[A]$。
- **結構危障（structural hazard）**：兩條指令需要**同一個實體資源**（例如同時用到 ALU 的同一部分）。（替 Hack 做管線不會遇到這種。）

任何危障都可能導致**停滯（stall，又稱 bubble）**：管線的其餘部分停住，直到問題指令完成。避免與緩解停滯是現代 CPU 設計中複雜的一環。

延伸補充：現代 CPU 怎麼對付危障。

- **資料前遞（forwarding ／ bypassing）**：把 Execute 階段的結果**直接繞線**給下一條指令的輸入，不等寫回暫存器——多數資料危障因此不必停滯。
- **分支預測（branch prediction）**：硬體記錄分支歷史來**猜**會不會跳，先照猜測繼續抓指令（**推測執行，speculative execution**）；猜錯就**沖掉（flush）**管線重來。現代預測器準確率常超過 95%。
- **亂序執行（out-of-order execution）**：讓不相依的後續指令先執行，填補停滯的空檔。

4.4　多核心呢？

定義

CPU 核心（core）是能獨立執行機器碼指令的自足電路。Hack CPU 整台就是一個核心。現代桌機 CPU 常有 4–16 核心；現代 GPU 有**數千**個。

有了多核心，上述所有問題都**更嚴重**（核心之間還要搶記憶體、搶快取、保持一致性）。多核心的效益多半來自**作業系統把核心分給不同應用程式**，或**開發者明確撰寫平行程式碼**——不是「自動變快」的魔法。

4.5　記憶體快取（Memory Caching）

AMD Zen 4 微架構（Ryzen 7000 系列）最高時脈 5.7GHz，即每秒 5.7×10^9 週期——每週期約 **0.175ns**。從主記憶體取一個值要多久（**延遲，latency**）？約 **73.3ns = 418 個週期**！我們絕不想每個擷取—執行週期都做一次。

為什麼這麼慢？

- 16GB+ 的位址空間意味著更複雜的電子電路 → 更長的傳播延遲；
- 16GB+ 的 SRAM 貴到不現實，所以主記憶體用 **DRAM**（第四週！）；
- **實體距離**！電流以光速（3×10^8 m/s）行進，走一公尺要 3.33ns——記憶體插槽離 CPU 幾十公分，一來一回就是好幾 ns。

現代 CPU 用**更小、更快、更近**的**記憶體快取**解決：L1、L2、L3 三層。以 Zen 4 為例（約略值）：

資料位置	容量	存取週期數	存取時間
暫存器	1.8KB ／核心	1	0.175ns
L1 快取	64KB ／核心	4	0.7ns
L2 快取	1MB ／核心	14	2.45ns
L3 快取	4MB ／核心	50	8.75ns
DDR5 RAM	16–64GB	418	73ns
快速 SSD	0.5–4TB	140,000	25µs
快速 HDD	0.5–4TB	28,500,000	5ms

（暫存器容量是指 224 個整數暫存器；另外還有數百個其他暫存器。）

運作方式：指令**預先**載入 L1 快取；遇到條件分支時盡可能「**往前看**」、把兩種可能都載入。其餘空間給**最常用的資料**（例如常用變數的記憶體）：先塞 L1、再 L2、再 L3、最後才是主記憶體。把主記憶體存取（**快取未中，cache miss**）的次數壓到最低，是 CPU 設計**與**程式碼最佳化的重要課題。

4.6　架構知識的應用一：謂詞化（Predication）

理解管線化告訴我們：if 與迴圈需要**分支**，分支製造條件危障，因此**不成比例地慢**。但把邏輯運算式求值成 0 或 1 **不會**產生條件危障——可能產生資料危障，但整體仍快。例如：

```
if (x <= 50) {
    y += 10;
} else {
    y = 3;
}
```

⇒

```
y = (y+10)*(x <= 50) + 3*(x > 50);
```

（C 的比較運算子回傳 0 或 1：條件成立的那一項留下、另一項乘 0 消失——**用算術取代分支**。）這叫**謂詞化（predication）**或 **無分支程式設計（branchless programming）**。

與內嵌組語一樣：**只對已確認是效能瓶頸的程式碼**這麼做。否則不值得換來 bug、耗時與可讀性損失。（現代編譯器也很聰明，你自作聰明可能**弊大於利**！）

4.7　架構知識的應用二：迴圈展開（Loop Unrolling）

定義

迴圈展開：把簡單迴圈的多次迭代**手動寫開**，以減少分支次數。

```
// 把 from 的前 count 個元素
// 複製到 to
int broke_array_copy(
    int from[], int to[],
    int count)
{
    for (int i = 0;
         i < count; i++)
        to[i] = from[i];
}
```

⇒

```
int woke_array_copy(
    int from[], int to[],
    int count)
{
    while (count >= 8) {
        *to++ = *from++;  // x8
        // ...同一行重複八次...
        count -= 8;
    }
    while (count > 0) {
        *to++ = *from++;
        count -= 1;
    }
}
```

設 `count` $= 500$。左邊要用 `count` $+ 1 = 501$ 次條件分支；右邊只要 $(1 + \texttt{count}/8) + (1 + \texttt{count}\%8) = 68$ 次。（順帶也省掉 500 次 `i` 的加法。）

嚇人的例子：Duff's device。把迴圈展開與 C 的 switch fall-through 合體：

```
int bespoke_array_copy(int from[], int to[], int count)
{
    register int n = (count + 7) / 8;
    switch (count % 8)
    {
    case 0: do {  *to++ = *from++;
    case 7:       *to++ = *from++;
```

```
    case 6:       *to++ = *from++;
    case 5:       *to++ = *from++;
    case 4:       *to++ = *from++;
    case 3:       *to++ = *from++;
    case 2:       *to++ = *from++;
    case 1:       *to++ = *from++;
            } while (--n > 0);
    }
}
```

switch 先跳到正確的「餘數」位置補齊零頭，然後 `do-while` 一圈複製八個。創作者 Tom Duff 自評：「很多人（甚至 Brian Kernighan? ）說 C 最糟的特性是 switch 的 case 標籤不會自動 break。這段程式碼算是這場辯論的某種論據，但我不確定是正方還是反方。」

本章重點

- 管線化：讓各階段硬體不閒置 → 吞吐量大增（延遲不變）；時脈只需配合最慢階段。
- Hack 四階段：Fetch ／ Decode ／ Execute ／ Writeback。
- 危障：資料（要用還沒算完的值）、條件（分支去向未知）、結構（搶資源）；都可能造成停滯。現代解法：前遞、分支預測 + 推測執行、亂序執行。
- 多核心不是自動變快：靠 OS 分工或顯式平行程式。
- 記憶體層級：暫存器 → L1 → L2 → L3 → RAM（418 週期！）→ SSD → HDD；減少 cache miss 是設計與最佳化的核心。
- 應用：謂詞化（用算術消分支）、迴圈展開（減少分支次數）——都只該用在確認過的瓶頸上。

5 綜合練習題（附詳解）

練習 1：機器碼解碼

把下列機器碼翻譯成組語：(a) `0000000000101010`；(b) `1110110000010000`；(c) `1111110111100001`。

解答

(a) 第一位是 **0** ⇒ A-指令；運算元 $= 101010_2 = 42$ ⇒ **`@42`**。

(b) `111` | comp `0110000` | dest `010` | jump `000`。$a = 0$、$c = 110000$ 查表 = `A`；dest `010` = `D=`；不跳 ⇒ **`D=A`**。

(c) `111` | comp `1110111` | dest `100` | jump `001`。$a = 1$、$c = 110111$ 查表 = `M+1`；dest `100` = `A=`；jump `001` = `;JGT` ⇒ **`A=M+1;JGT`**。

練習 2：組語編碼

把下列組語翻譯成 16 位元機器碼：(a) `@SCREEN`；(b) `MD=D-1`；(c) `D;JLE`。

解答

(a) `SCREEN` $= 16384 = 100000000000000_2$（15 位元）⇒ **`0100000000000000`**。

(b) comp `D-1`：$a = 0$、$c = 001110$；dest `MD` $= d = 011$；不跳 $j = 000$ ⇒ `111 0001110 011 000` = **`1110001110011000`**。

(c) comp `D`：$a = 0$、$c = 001100$；dest 無 $=$ `000`；`JLE` $= j = 110$ ⇒ `111 0001100 000 110` = **`1110001100000110`**。

練習 3：語法限制的硬體根源

用指令格式解釋：(a) 為什麼沒有 `D=17` 這種指令？(b) 為什麼多重指定（`AMD=`）「免費」？(c) 為什麼恰好有 7 種跳躍條件？

解答

(a) C-指令的 comp 欄位只有 $a + 6$ 個位元，只能編碼表中固定的 28 種運算——沒有任何空間塞一個任意常數。常數只能經由 A-指令（有 15 位元運算元）進入 CPU。

(b) dest 是**三個獨立位元**，d_1、d_2、d_3 各自 gating A、D、M 的載入訊號。寫一個或寫三個目的地用的是**同一條指令、同一個週期**。

(c) jump 是三個位元，各對應「結果 < 0」「$= 0$」「> 0」三個旗標；$2^3 = 8$ 種組合扣掉全 0（不跳）恰好 7 種。`JMP` $=$ `111`（任何結果都滿足三者之一）。

練習 4：ISA 還是微架構？

下列各項屬於 ISA 性質還是微架構性質？(a) 時脈 5.7GHz；(b) 16 位元字長；(c) `A=D;JMP` 的行為；(d) ALU 用進位前瞻加法器；(e) `KBD` 映射到 `0x6000`；(f) 指令 `D=D+M` 的存在。

解答

(a) **微架構**（速度是實作特性）。(b) **ISA**（字長是規格）。(c) **微架構**——這是**未定義行為**，ISA 沒規定，各實作可以不同（正是表中的例子）。(d) **微架構**（電路設計選擇；換成漣波進位加法器不影響任何程式）。(e) **ISA**（I/O 記憶體映射是規格的一部分）。(f) **ISA**（機器碼指令集是規格核心）。

練習 5：PC 元件追蹤

PC 初值 20。連續五個時脈的輸入為：(1) inc=1；(2) load=1, in=100；(3) inc=1；(4) reset=1；(5) inc=1。寫出每個時脈後的輸出。

解答

(1) $20+1=\mathbf{21}$；(2) 載入 in $\Rightarrow \mathbf{100}$；(3) $100+1=\mathbf{101}$；(4) 歸零 $\Rightarrow \mathbf{0}$；(5) $0+1=\mathbf{1}$。對應到 CPU：(2) 是跳躍發生（A 進 PC）、(4) 是使用者按下 reset、其餘是正常前進。

練習 6：CPU 接線理解

(a) 執行 A-指令時，A 暫存器前的 MUX（M1）選什麼？ALU 在做什麼？(b) `writeM` 什麼時候是 1？(c) 為什麼 `addressM` 直接接 A、不經過任何邏輯？

解答

(a) M1 選 `instruction`（15 位元運算元左補 0），載入 A。ALU **照常運算**但輸出被忽略——d 位元全部視為 0、不寫任何暫存器、`writeM` $=0$、不跳。
(b) `writeM` $=1$ 當且僅當這是 **C-指令**（opcode $=1$）**且** $d_3=1$（dest 含 `M`）。兩個條件 AND 起來——漏掉 opcode 檢查的話，A-指令的位元圖樣可能誤觸寫入！
(c) ISA 規定讀寫 RAM 的位址**永遠**是 A 的值（$M \equiv RAM[A]$）。不需要選擇、不需要運算，直接拉線即可——最簡單的正確實作。

練習 7：Harvard vs von Neumann

(a) Hack 是哪種架構？根據是什麼？(b) von Neumann 的主要優點是什麼？舉一個 Harvard 做不到的事。(c) Harvard 的主要優點？

解答

(a) **Harvard**：指令存 ROM、資料存 RAM，**兩個分開的記憶庫**、各自的匯流排。
(b) 指令與資料同放 RAM，所以**程式可以被當資料處理**：作業系統能從磁碟載入程式到記憶體再執行（Hack 做不到——程式燒在 ROM 裡）；編譯器能產出新程式立刻執行；（極端情況）程式能自我修改。
(c) **硬體較簡單**：不用仲裁「這個週期要取指令還是取資料」，兩者可同時進行、不互搶頻寬（也因此避開一類結構危障）。

練習 8：定址模式判別

判斷下列各指令（或其欄位）使用哪種定址模式：(a) `@100` 中的 `100`；(b) `D=M` 中的 `M`；(c) `D=A` 中的 `A`；(d) ARM7 `LDR R0, [R1, #8]`。

解答

(a) **立即定址**——100 就是要載入的資料本身。

(b) **間接定址**——`M` 表示「A 中存的位址上的資料」：運算元透過 A 這個**指標**才找到資料。

(c) **直接定址**——運算元指名資料所在的**位置**（暫存器 A），直接取用其內容。

(d) **索引間接定址**——先讀 R1 得基底位址、加上偏移量 8、再到該位址取資料。一條指令完成 `arr[2]`（int 為 4 位元組時）。

練習 9：管線加速比

某 CPU 四階段延遲為：Fetch 2ns、Decode 1ns、Execute 3ns、Writeback 2ns。(a) 不用管線：時脈週期多長？執行 100 條指令要多久？(b) 用管線：時脈週期多長？100 條指令（無停滯）要多久？(c) 加速比是多少？理想上限是多少？

解答

(a) 不用管線時，一個週期得走完全部四階段：$2+1+3+2=8$ns。100 條 $\times$ 8ns $=$ **800ns**。

(b) 管線時脈配合**最慢階段**：3ns。第一條指令要 4 個週期填滿管線，之後每週期完成一條：$(4+99)\times 3 = 309$ns $\approx$ **309ns**。

(c) 加速比 $= 800/309 \approx$ **2.6**。理想上限是**階段數 4**，但只有當四個階段延遲**完全均等**（各 2ns、時脈 2ns）且無停滯時才達到：$800/(103\times 2)\approx 3.9$。不均等的階段（Execute 3ns）拖慢了整條管線——這就是為什麼實際 CPU 設計要盡量把管線切得平均。

練習 10：危障辨識

指出下列 Hack 指令序列中每一處危障的類型：

```
@100      // (1)
D=M       // (2)
D=D+1     // (3)
@LOOP     // (4)
D;JNE     // (5)
M=0       // (6)
```

解答

- (1)→(2)：**資料危障**——(2) 的 `M` 依賴 (1) 寫入的新 A（$M = RAM[A]$）。
- (2)→(3)：**資料危障**——(3) 需要 (2) 算出的新 D。
- (4)→(5)：**資料危障**——(5) 的跳躍目標是 (4) 剛載入的 A。

- (5)→(6)：**條件危障**——(5) 是條件分支，在它於 Execute 階段解出結果之前，管線不知道下一條該抓 (6) 還是 `LOOP` 處的指令。

（資料前遞可消除多數資料危障的停滯；條件危障要靠分支預測。）

練習 11：快取數量級

用第 4 章的 Zen 4 表格：(a) 一段程式每次迭代要 1 次 L1 命中；改寫後每次迭代變成 1 次 RAM 存取。慢了幾倍？(b) 為什麼「把常一起用的資料放在相鄰位址」（空間局部性）能加速？(c) 從 SSD 讀資料比從 RAM 讀慢幾倍？

解答

(a) L1 = 4 週期、RAM = 418 週期 ⇒ 約 $418/4 \approx$ **105 倍**。資料放哪裡對效能的影響可以遠大於演算法層面的微調！

(b) 快取以**快取行（cache line**，通常 64 位元組）為單位搬運：存取一個位址時，**整行鄰近資料**一起進快取。相鄰位址的後續存取因此直接命中 L1——這就是循序掃陣列遠快於隨機跳躍（或鏈結串列）的原因。

(c) $140{,}000/418 \approx$ **335 倍**（25µs vs 73ns）。所以「把資料留在記憶體」與「別讓工作集超過 RAM」是效能的鐵律。

練習 12：謂詞化與迴圈展開

(a) 把下列 C 程式謂詞化（消除分支）：

```
if (a > b) { max = a; } else { max = b; }
```

(b) `count` = 100 時，8 路展開的複製迴圈（第 4 章的 `woke_array_copy`）用幾次條件分支？原始版呢？(c) 為什麼這兩招都不該隨手亂用？

解答

(a)

```
max = a*(a > b) + b*(a <= b);
```

（比較運算回傳 0/1，恰好一項存活。）

(b) 展開版：$(1 + 100/8) + (1 + 100\%8) = (1 + 12) + (1 + 4) =$ **18** 次；原始版：$100 + 1 =$ **101** 次。約 5.6 倍的分支縮減。

(c) 因為**可讀性、維護性下降且容易寫錯**（展開版的邊界處理、謂詞化的運算子優先序都是 bug 溫床）；而且**現代編譯器通常會自動**做這些最佳化（`-O2`/`-O3` 會展開迴圈、產生條件搬移指令），手寫版本反而可能妨礙編譯器，弊大於利。只在 profiler 證實的瓶頸上使用。

A 附錄：速查表

A.1 C-指令編碼總表

格式： `111` $a\,c_1c_2c_3c_4c_5c_6\,d_1d_2d_3\,j_1j_2j_3$

a=0	a=1	c_1	c_2	c_3	c_4	c_5	c_6
`0`		1	0	1	0	1	0
`1`		1	1	1	1	1	1
`-1`		1	1	1	0	1	0
`D`		0	0	1	1	0	0
`A`	`M`	1	1	0	0	0	0
`!D`		0	0	1	1	0	1
`!A`	`!M`	1	1	0	0	0	1
`-D`		0	0	1	1	1	1
`-A`	`-M`	1	1	0	0	1	1
`D+1`		0	1	1	1	1	1
`A+1`	`M+1`	1	1	0	1	1	1
`D-1`		0	0	1	1	1	0
`A-1`	`M-1`	1	1	0	0	1	0
`D+A`	`D+M`	0	0	0	0	1	0
`D-A`	`D-M`	0	1	0	0	1	1
`A-D`	`M-D`	0	0	0	1	1	1
`D&A`	`D&M`	0	0	0	0	0	0
`D|A`	`D|M`	0	1	0	1	0	1

dest	d_1	d_2	d_3
—	0	0	0
`M=`	0	0	1
`D=`	0	1	0
`DM=`	0	1	1
`A=`	1	0	0
`AM=`	1	0	1
`AD=`	1	1	0
`ADM=`	1	1	1

jump	j_1	j_2	j_3
—	0	0	0
`JGT`	0	0	1
`JEQ`	0	1	0
`JGE`	0	1	1
`JLT`	1	0	0
`JNE`	1	0	1
`JLE`	1	1	0
`JMP`	1	1	1

j_1：結果 <0 則跳；j_2：$=0$ 則跳；j_3：>0 則跳。$c_1..c_6$ 恰為 ALU 的 zx, nx, zy, ny, f, no（$x = D$、$y = a?M : A$）。

A.2 ISA 快速比較

	Hack	x64	ARM	MIPS
哲學	教學用	極偏 CISC	偏 RISC	RISC
指令長度	固定 16 位元	變長（1–15 位元組）	固定（A64：32 位元）	固定 32 位元
記憶體架構	Harvard	von Neumann	von Neumann	von Neumann
一般用途暫存器	1（D）	16	31	32
典型用途	課堂	桌機／伺服器	手機／ Mac	嵌入式／舊主機

A.3 記憶體層級（Zen 4 約略值）

層級	容量	週期	時間
暫存器	1.8KB ／核心	1	0.175ns
L1	64KB ／核心	4	0.7ns
L2	1MB ／核心	14	2.45ns
L3	4MB ／核心	50	8.75ns
DDR5	16–64GB	418	73ns
SSD	0.5–4TB	1.4×10^5	25µs
HDD	0.5–4TB	2.85×10^7	5ms

A.4 名詞中英對照

英文	中文	英文	中文
instruction set architecture	指令集架構	pipelining	管線化
microarchitecture	微架構	throughput / latency	吞吐量／延遲
opcode / operand	操作碼／運算元	hazard	危障
Harvard architecture	哈佛架構	stall / bubble	停滯／氣泡
von Neumann architecture	馮諾伊曼架構	forwarding	資料前遞
RISC / CISC	精簡／複雜指令集	branch prediction	分支預測
general-purpose register	一般用途暫存器	speculative execution	推測執行
special-purpose register	特殊用途暫存器	core	核心
addressing mode	定址模式	cache / cache miss	快取／快取未中
immediate / direct / indirect	立即／直接／間接	predication	謂詞化
hardware interrupt	硬體中斷	loop unrolling	迴圈展開
polling	輪詢	Duff's device	達夫裝置

A.5 參考資料

- John Lapinskas, *The Hack ISA ／ The Hack microarchitecture ／ Comparative architecture ／ Microarchitecture optimisation*（7-1 至 7-4），COMSM1302, University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 4–5（機器語言與電腦架構）.
- Blem, Menon & Sankaralingam, *ISA Wars: Understanding the Relevance of ISA being RISC or CISC…*, ACM TOCS, 2015（RISC/CISC 實測研究）.
- Chips and Cheese, *ARM or x86? ISA Doesn't Matter*（micro-ops 與現代觀點）.
- Hennessy & Patterson, *Computer Architecture: A Quantitative Approach*（管線化、危障、快取的標準教科書）.
- Tom Duff, *Duff's device*（1983 年的 netnews 貼文）.