程式人

用 20 分鐘

向 nand2tetris 學會設計處理器

陳鍾誠

2016 年 1 月 6 日

在大學的時候

- 我念資訊科學系
  - 雖然沒修《計算機結構》
  - 但是有修《數位邏輯》

而且

- 還自己看了《計算機結構》的書

後來

- 上了資工研究所
  - 碩士班時《計算機結構》是必修
  - 博士班時《計算機結構》還必考

所以

- 算來我整整念了三次《計算機結構》

但是

- 我真的讀懂了嗎？

如果讀懂了

- 我應該有能力設計自己的
  - CPU 處理器
  - 還有整台電腦了
- 不是嗎？

後來

- 我到金門技術學院教書
- 還教計算機結構

我很心虛

- 因為，自己沒設計過電腦
- 教甚麼計算機結構呢？

難道

- 上課照著課本念，就算教完了嗎？

所以

- 我決定自己設計處理器
- 自己設計一整台電腦

當然

- 我可以選擇用《麵包版》
  - 插出一整台電腦
- 但是這太困難了
  - 萬一哪條線差錯了，或者接觸不良
  - 那我就掛了

而且

- 要那麼多的邏輯閘，我要買多少 74xxx 的晶片才夠呢？

還好

- 現在有 FPGA 板，可以讓我們用《硬體描述語言》寫程式
- 寫完就可以燒進去變成一台電腦

這樣

- 我就不用擔心
  - 接錯線
  - 接觸不良
  - 邏輯閘不夠
- 等等問題了

所以

- 我開始學習用 VHDL，Verilog 寫硬體程式

經過一小段嘗試後

- 我覺得 VHDL 的語法有點囉嗦，所以就選擇了寫起來簡單的 Verilog

然後

- 從基本 and, or, not 閘開始
- 一路建構出
  - 半加器、全加器、 32 位元加法器

接著

- 還用 and, or, not 閘
  - 建構《正反器、暫存器》等等記憶元件
- 雖然 Verilog 可以直接宣告整塊記憶體
  - 但是我覺得還是從頭開始比較扎實

有了前面的32位元加法器

- 我就可以設計
  - 《算術邏輯單元》 ALU

接著問題來了

- 我要怎麼設計出
  - 一顆處理器 CPU
  - 還有整台電腦呢？

我有點卡住了

- 雖然我唸過三次《計算機結構》
- 對理論也算是很熟悉了

但是

- 我就是設計不出來
- 設計出來了也不能正常運作

還好

- 我會寫程式，知道
  - 當程式寫不出來的時候
  - 就上網找範例吧！

經過搜尋之後

- 我找到一顆由華盛頓大學 Richard 老師設計的處理器
- 然後把它看懂，而且測試過了

有了這個經驗之後

- 我終於設計出了自己的處理器 CPU0

後來上計算機結構時

- 我就拿自己設計的 Verilog 版 CPU0 當範例

結果發現

- 很多學生覺得太難
- 很多學生學不會

當然

- 這有部分是學生不夠用功
- 也有部分是老師教得不好

但是

- 我已經盡力簡化 Verilog 程式了

於是

- 我想或許是 CPU0 本身還是太難
- 所以我又設計了一顆更簡單的 MCU0

然後

- 又用來教學生
- 不過還是有很多學生沒學會

上課時

- 有些學生沒在聽
- 有些聽了不會卻沒有問
- 有聽又有問的同學通常都會了
  - 不過也沒剩下幾個了

上個學期

- 我去 coursera 修了 nand2tetris 這門課
- 我覺得這門課太棒了！

後來我決定

- 就用 nand2tetris 線上課程當作是計算機結構的主軸

讓學生們透過

- 實作 nand2tetris 的習題來學習計算機結構

這樣

- 就不會學完之後，還不知道怎麼設計處理器和電腦了。

如果

- 我們連一台簡單的電腦都設計不出來
- 就拼命看教科書裡那種有一大堆 pipeline 管線結構的複雜電腦
- 然後計算要加多少快取才能得到最好的效能

那我們到底

- 是在學些甚麼呢？

我一直覺得

- 《計算機結構》聖經版的
  - 白算盤教科書
- 是寫給在 Intel 或 ARM 裡面工作
  - 已經設計過十顆處理器的人看的

而不是

- 寫給《念大學資工系的學生》看的
- 所以
  - 我不太喜歡《白算盤》
  - 也不喜歡《紅算盤和綠算盤》

當我看到

- Nand2tetris 這門課時

我就被這門課

- 深深的吸引了！

因為

- Nand2tetris 這門課
- 既不教你《白算盤、紅算盤》
- 也不教你《管線與快取》

而是直接教你

- 從 nand 閘開始，建構出基本元件
- 然後從全加器、ALU、暫存器

  一路向上建構出 CPU 與整台電腦

雖然 nand2tetris 課程中

- 那顆 CPU 的指令長得很奇怪
  - 這是我對 nand2tetris 課程的唯一抱怨
- 但是電路設計卻簡潔有力

更棒的是

- 這些習題都是老師精心設計
- 讓你可以一步一步，像爬樓梯一般的學習，盡可能讓你每一步都踩得很踏實

你只要跟著習題

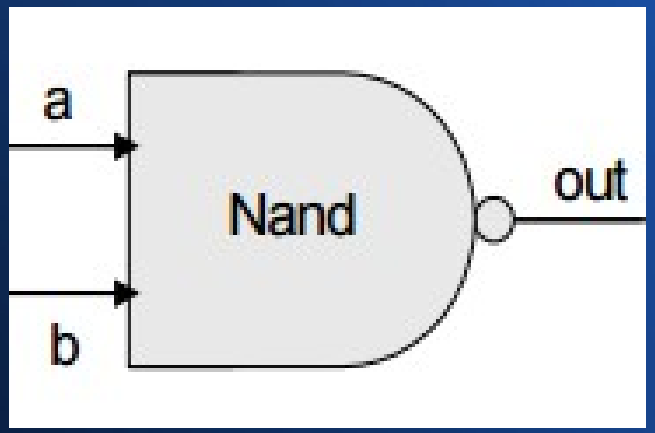
- 一題一題做上來，就能學會設計一顆處理器的方法
- 建構出一台完整的電腦了。

現在

- 就讓我們開始向 nand2tetris 學習如何設計處理器吧！

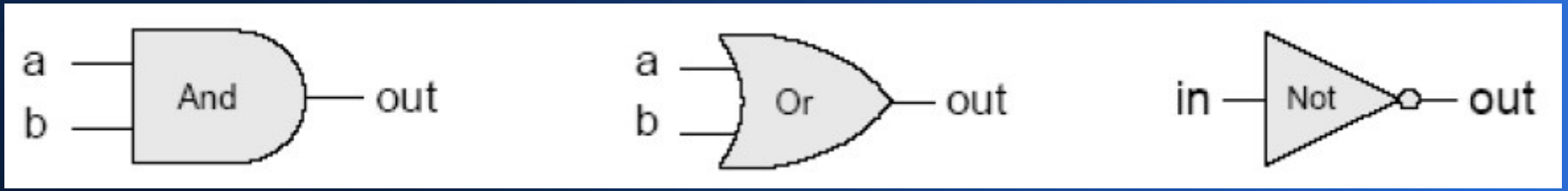
首先、我們所擁有的

- 就只是最基本的 nand 閘而已



您可以想想

- 如何用 nand 閘，建構出
  - and, or, not



- 等基本邏輯閘

您只要會布林代數

- 就可以導出下列算式

```
Not(a)   = Nand(a,a)

true  = Not(false)

And(a,b)  = Not(Nand(a,b))

Or(a,b)  = Not(And(Not(a),Not(b)))

Xor(a,b)  = Or(And(a,Not(b)),And(Not(a),b))
```

George Boole

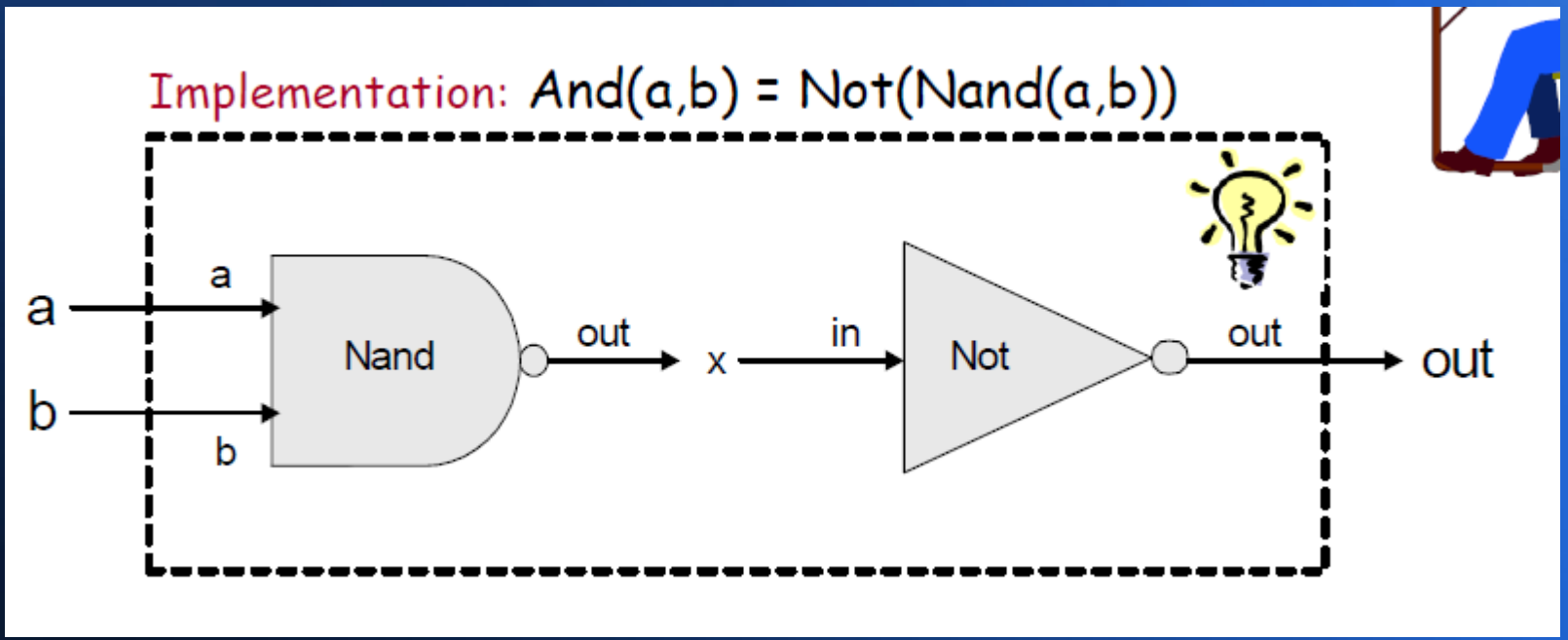
("A Calculu

然後畫出對應的電路

- 像是 not(a) = nand(a, a)
- 你就只要把 nand 的兩條輸入線接再一起，就做完了。

要做 and 閘

- 只要用 nand+not 接起來就完成了



（nand 是唯一的基本元件，而 not 剛剛建構過了）

要做用 nand 做出 or 閘
會稍微麻煩一點

- 但也只要套這個公式就行了

```
Or(a,b) = Not(And(Not(a),Not(b)))
```

不過、要通過 nand2tetris 的課程考驗

- 你不能只會畫圖，還要會寫 HDL 程式

And.hdl

```
CHIP And
{    IN  a, b;
     OUT out;
     Nand(a = a,
          b = b,
          out = x);
     Not(in = x, out = out)
}
```

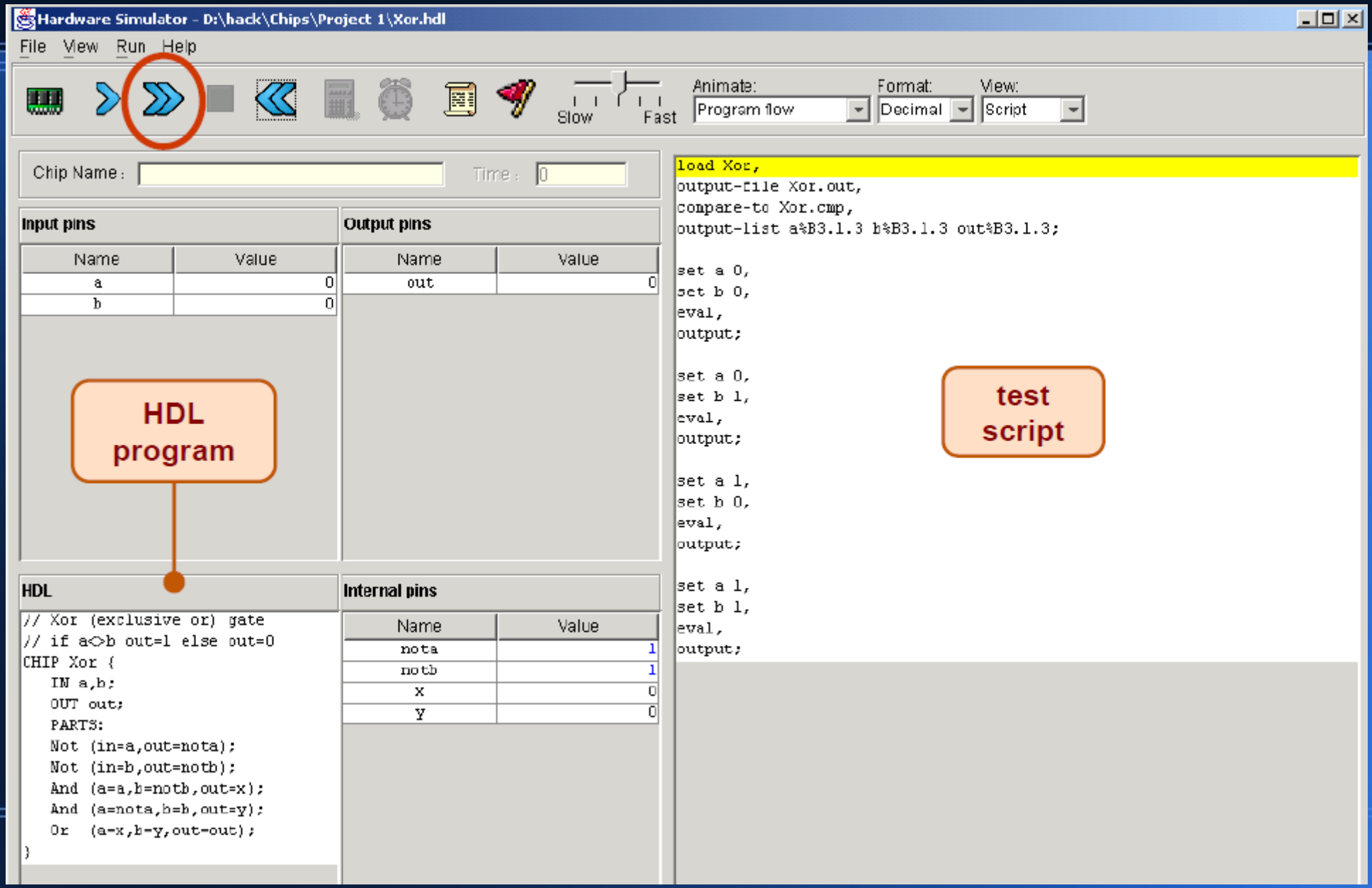
而且、老師都已經給了框架

- 您只要把內容填上就好了！

And.hdl

```
CHIP And
{    IN  a, b;
     OUT out;
     // implementation missing
}
```

不過、寫好之後記得要測試



老師們很用心的

- 找了一堆程式人員，專門為這門課開發了
  - HDL 模擬軟體
  - 虛擬機軟體
  - 以及完整的測試案例

您只要跟著習題的腳步

- 一題一題寫好並測試就行了

每一個章節

- 都會有一些習題，等著你去完成

像是第一章就有 16 題

Project 1: Elementary Logic Gates

Chips

Chip (HDL)	Description	Test Script	Compare File
Nand	Nand gate (primitive)		
Not	Not gate	Not.tst	Not.cmp
And	And gate	And.tst	And.cmp
Or	Or gate	Or.tst	Or.cmp
Xor	Xor gate	Xor.tst	Xor.cmp
Mux	Mux gate	Mux.tst	Mux.cmp
DMux	DMux gate	DMux.tst	DMux.cmp
Not16	16-bit Not	Not16.tst	Not16.cmp
And16	16-bit And	And16.tst	And16.cmp
Or16	16-bit Or	Or16.tst	Or16.cmp
Mux16	16-bit multiplexor	Mux16.tst	Mux16.cmp
Or8Way	Or(in0,in1,...,in7)	Or8Way.tst	Or8Way.cmp
Mux4Way16	16-bit/4-way mux	Mux4Way16.tst	Mux4Way16.cmp
Mux8Way16	16-bit/8-way mux	Mux8Way16.tst	Mux8Way16.cmp
DMux4Way	4-way demultiplexor	DMux4Way.tst	DMux4Way.cmp
DMux8Way	8-way demultiplexor	DMux8Way.tst	DMux8Way.cmp

用 nand 建構基本邏輯閘

多工器與解多工器

更大更多輸入的邏輯閘

更大更多輸入的
多工器與解多工器

而且老師都準備好了

- 溫馨的小提醒

Project 1 tips

- Read the Introduction + Chapter 1 of the book
- Download the book's software suite
- Go through the hardware simulator tutorial
- Do Project 0 (optional)
- You're in business.

等著你來

- ~~自投羅網！~~
- ~~自生自滅！~~
- 自主學習！

相信您一定可以

- 學會如何《設計處理器》的

要寫出第一章的習題

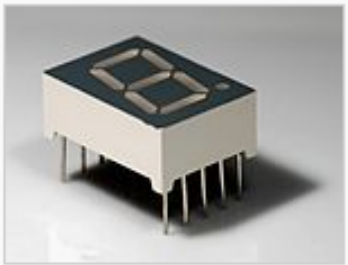
- 首先要學會
  - 真值表
  - 卡諾圖
  - 數位電路

如果您學過數位邏輯

您一定知道七段顯示器中每一根亮棒的電路應該怎麼設計吧？

七段顯示器（英語：Seven-segment display）為常用顯示數字的電子元件。因為藉由七個發光二極體以不同組合來顯示數字，所以稱為「七劃管」、「七段數碼管」、「七段顯示器」，由於所有燈管全亮時所表示的是「8」，所以又稱「8字管」、「8字顯示器」。

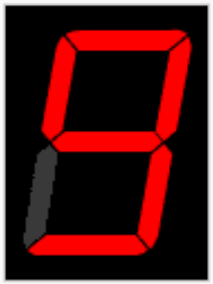
有些七段顯示器還會在右下角附加一個表示小數點的燈管，因此也稱八段管。



以發光二極體作顯示組件的七段顯示器，由本體下的十支接腳控制發光的部份。

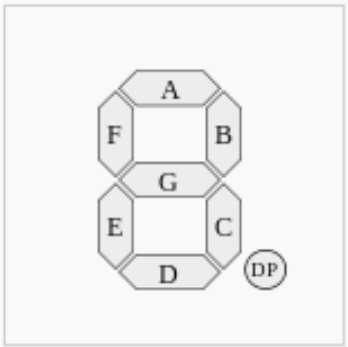
構造 [編輯]

一般的七段顯示器擁有八個發光二極體用以顯示十進位0至9的數字，也可以顯示英文字母，包括十六進位中的英文A至F（b，d為小寫，其他為大寫）。現時大部份的七段顯示器會以斜體顯示。



HP-16C計算機的七段顯示器能夠顯示2，8，10，16進位的數字

七段顯示器由四個直向、三個橫向及上右下角一點的發光二極體組成，由以上向條發光體組合出不同的數字。除七段顯示器外，還有十四及十六劃的顯示器，但現時已被點陣顯示器（英語：Dot-matrix）所取代。七段顯示器分為**共陽極**及**共陰極**，共陽極的七段顯示器的正極（或陽極）為八個發光二極體的共有正極，其他接點為獨立發光二極體的負極（或陰極），使用者只需把正極接電，不同的負極接地就能控制七段顯示器顯示不同的數字。共陰極的七段顯示器與共陽極的只是接駁方法相反而已。



以上為七段顯示器各劃的排序，用家可根據排序接連控制器。

控制 [編輯]

共陽極與共陰極的七段顯示器已可以特定的集成電路控制，只要向集成電路輸入4-bit的二進位數字訊號就能控制七段顯示器顯示。

您只要

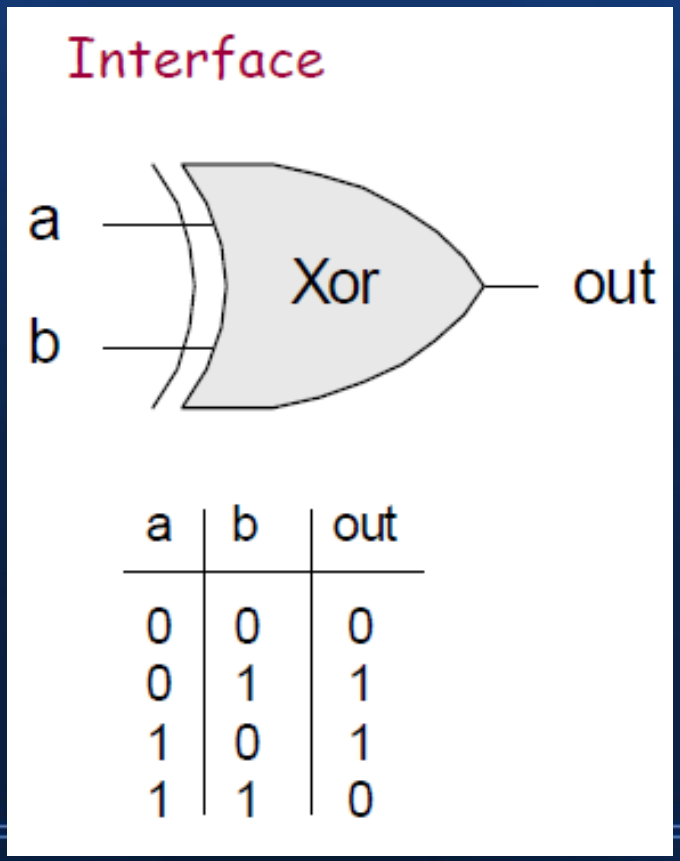
- 會列出真值表
- 會填寫卡諾圖並框出區塊
- 然後根據區塊寫出邏輯式
- 接著畫出數位電路圖就可以了

如果您忘記了

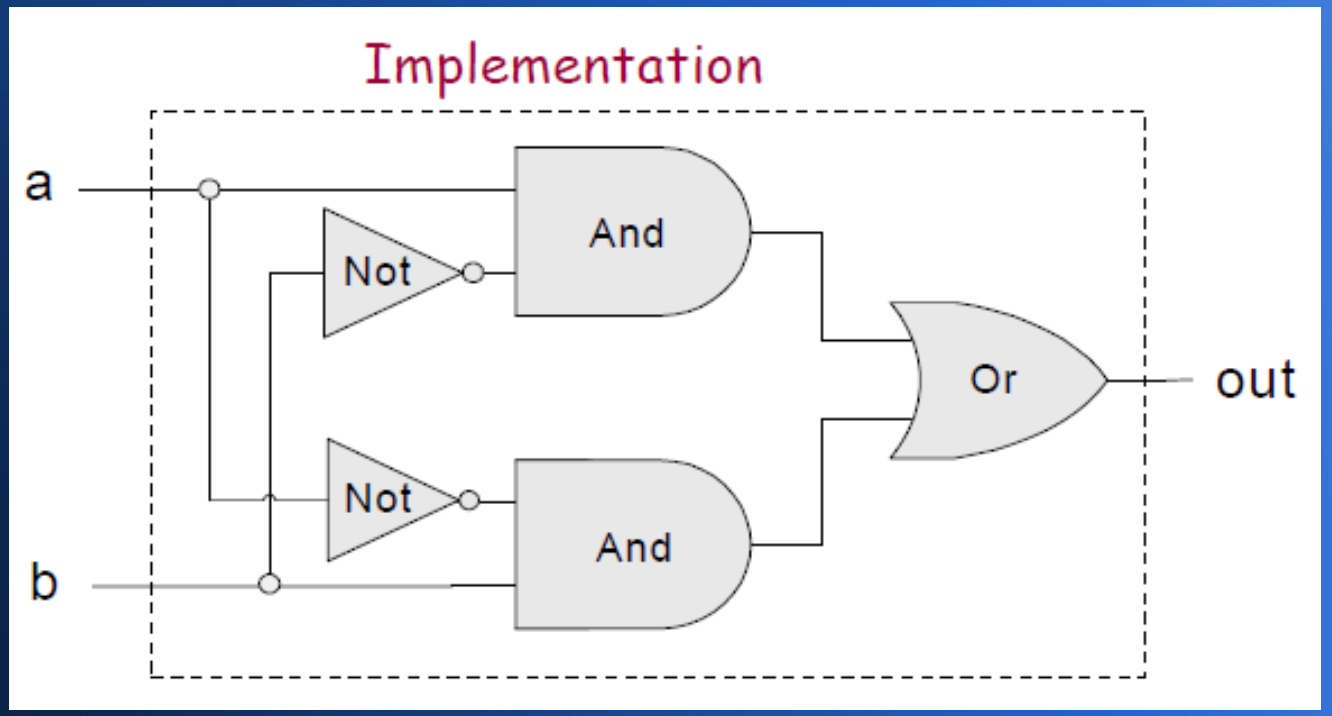
- 現在就隨便選一根七段顯示器的亮棒，重新開始
  - 列真值表
  - 畫卡諾圖
  - 寫邏輯式
  - 畫數位電路
- 這樣您就有足夠的基礎完成這一章的習題了！

當您用 nand 建構出 and, or, not 之後

- 可以繼續建構 xor



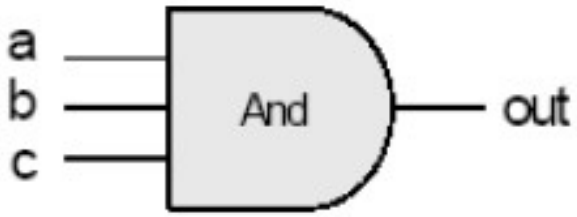
a	b	out
0	0	0
0	1	1
1	0	1
1	1	0



接著繼續向上建構出更大的元件

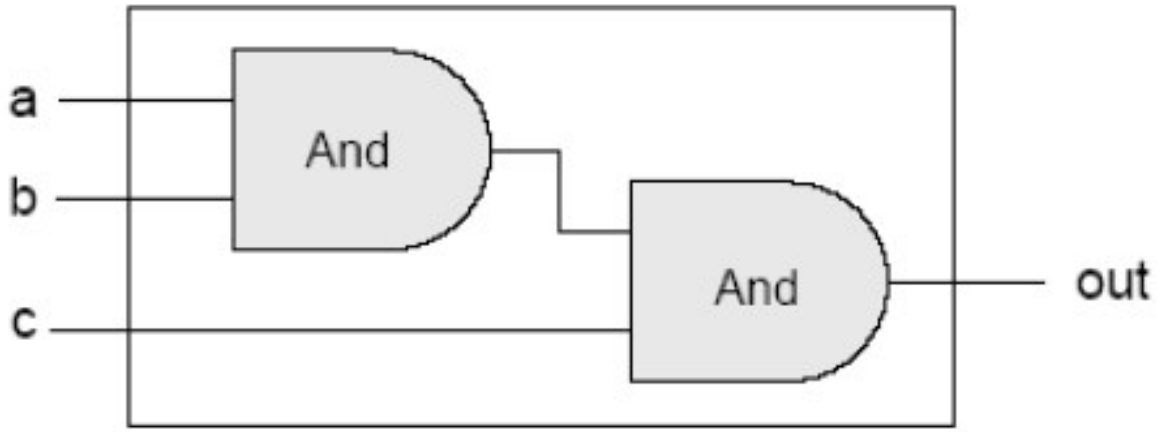
Composite gates:

Gate interface




```
If a=b=c=1 then out=1
else out=0
```

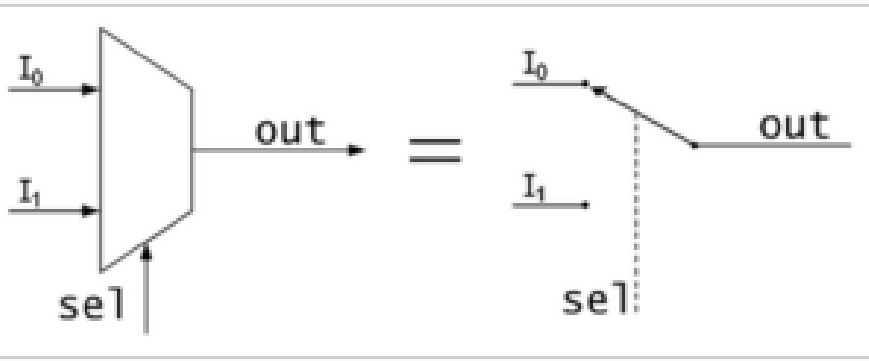
Gate implementation



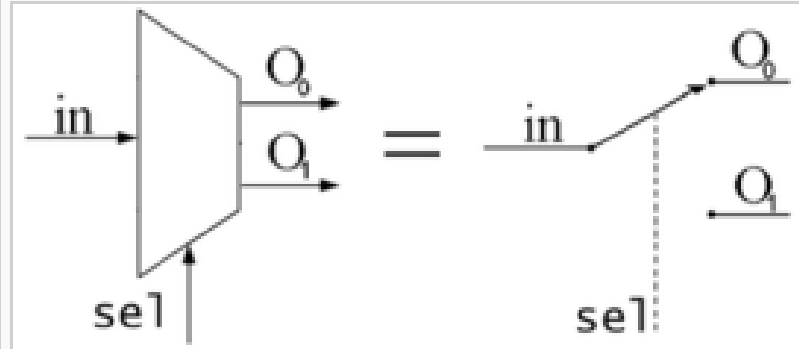
還有控制電路中很重要的

- 多工器： MUX
- 解碼器： DMUX

注意：這邊是示意圖，
真正電路必須用 AND, OR, NOT 去建構出來

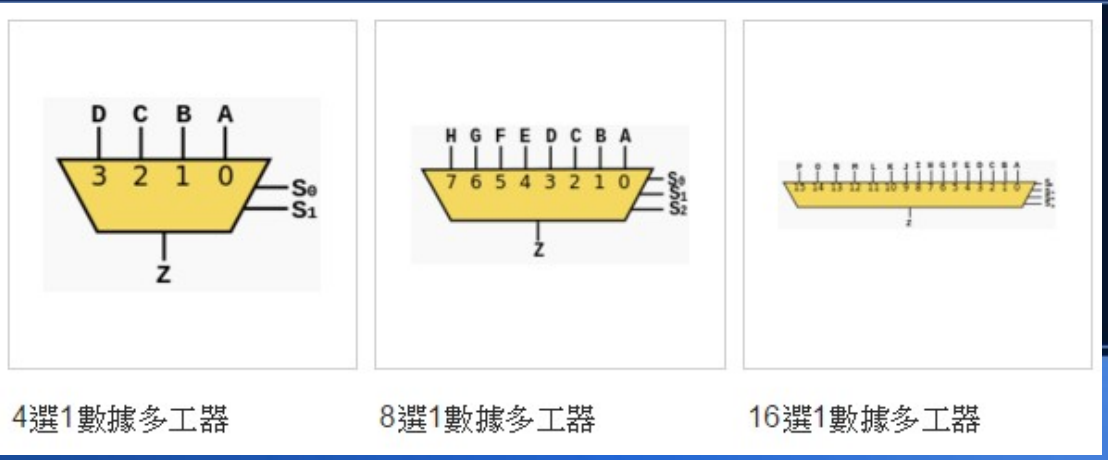


2選1數據多工器的結構簡圖，其功能類似一個雙擲的開關。



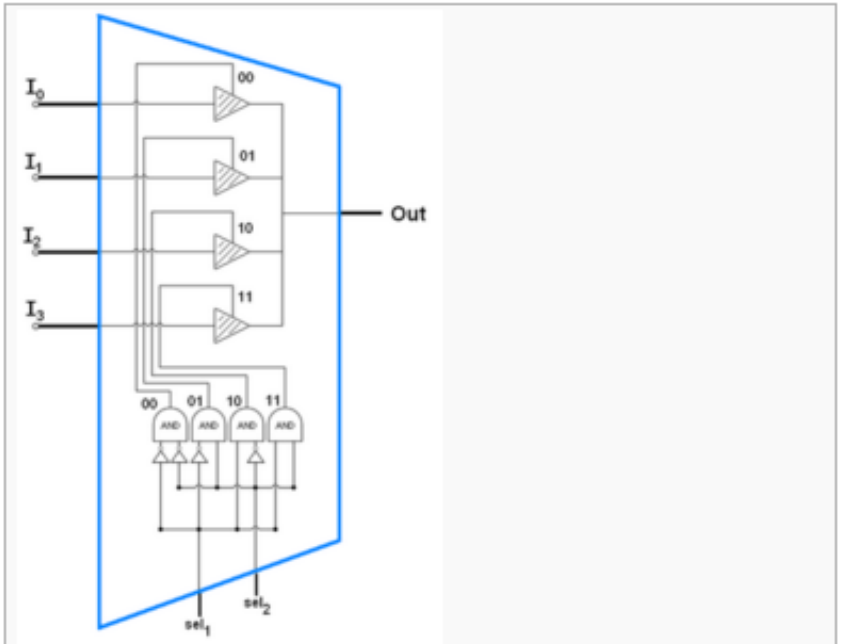
1線—2線數據分配器。像數據多工器一樣，它可以等同於一個控制開關。

然後是更大的多工器與解碼器



4選1數據多工器　8選1數據多工器　16選1數據多工器

- 四輸入多工器 :Mux4way16
- 四輸入解碼器
  - DMux4way, DMux4way16
- 八輸入多工器：
  - Mux8way, Mux8way16
- 八輸出解碼器： DMux8way



兩種實現方式：

- 由一個解碼器、幾個及閘、一個或閘實現
- 由幾個三態門、幾個及閘（及閘充當解碼器）

注意：I_n輸入引腳的下標表示選擇端所表示的二進位數的各位

4選1數據多工器的布林函數

$$F = (A \cdot \overline{S_0} \cdot \overline{S_1}) + (B \cdot \overline{S_0} \cdot S_1) + (C \cdot S_0 \cdot \overline{S_1}) + (D \cdot S_0 \cdot S_1)$$

這樣就完成了第一章的習題

接著在第二章

- 我們要學習《運算電路》的設計

所謂的運算電路

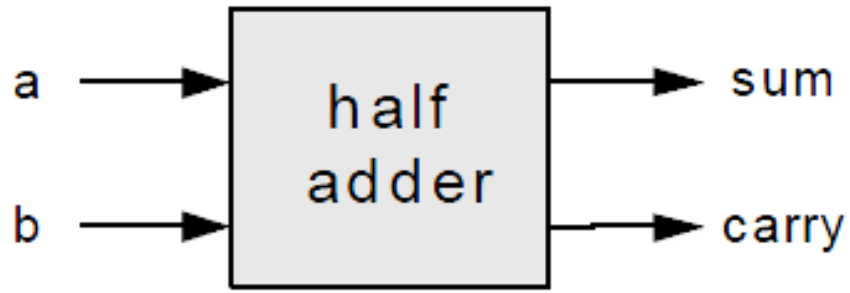
- 主要就是
  - 《加法器》
  - 《減法器》
  - 算術邏輯單元 ALU

而這一切

- 必須從一位元的加法電路開始
- 包含《半加器》和《全加器》

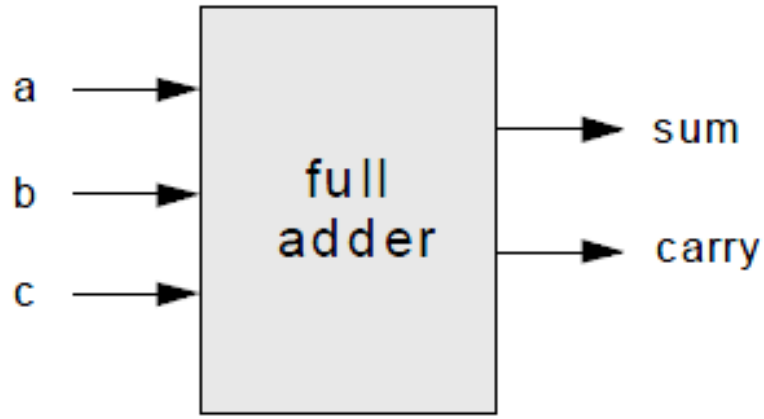
半加器

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



全加器

a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



只要學過數位邏輯中的

- 《真值表》和《卡諾圖》
- 您應該可以輕易設計出進位 C 的電路
- 總和 S 電路可以用 XOR 完成

我發現很多學生

- 雖然修過《數位邏輯》這門課
- 但學期一過就全部還給老師了！

而且、他們常常不知道

- 《數位邏輯》就是用來設計電腦硬體的核心學問
- 也不知道《數位邏輯》與《計算機結構》課程的關聯

雖然我常常提醒他們

- 但是學東西似乎就是要經過
  1. 見山是山
  2. 見山不是山
  3. 見山又是山
- 等三個階段

當一個人還在

- 第一階段的《見山是山》狀態時
- 你告訴他《山不是山》
  - 他只會認為你是白癡而已！

當他進入第二階段

- 《見山不是山》的狀態時
- 你告訴他《見山又是山》
  - 他會認為你道行不夠
  - 給你一個鄙視的眼神

等他到達第三階段

- 發現《見山又是山》的時候
- 這時你已經不需要教他了
- 因為他已經完全學會了

這三個層次

- 就是當老師最難以突破的障礙了

教學生《計算機結構》

- 也是如此！

記得有位在台科大念研究所的學生

- 在臉書上告訴我
  - 他修我的《計算機結構》都聽不懂
  - 為何我們不教《白算盤》那本書
- 我真的很想問他
  - 那你整個學期怎麼都沒提出來
  - 直到畢業都沒有告訴我這件事
  - 等到你去台科大念碩士了才告訴我呢？

我想、這也不能怪他

- 因為他很可能不知道
  - 《計算機結構》到底是甚麼？
- 只知道、研究所常常會考
  - 《白算盤》那一本書！

畢竟

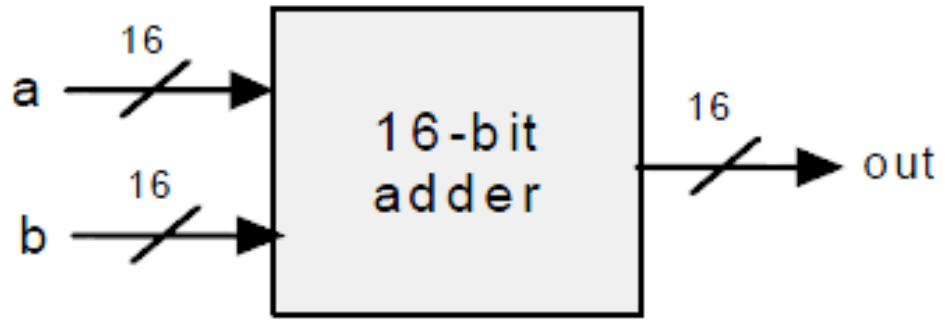
- 我們從小就像《王語嫣》那樣
- 每天背書背書背書，只是為了讓~~《表哥》~~《爸媽》開心而已！

抱歉，離題太遠了！

當你設計出

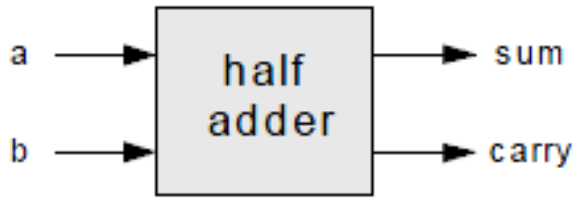
- 半加器、全加器之後
- 就可以把一堆全加器串起來

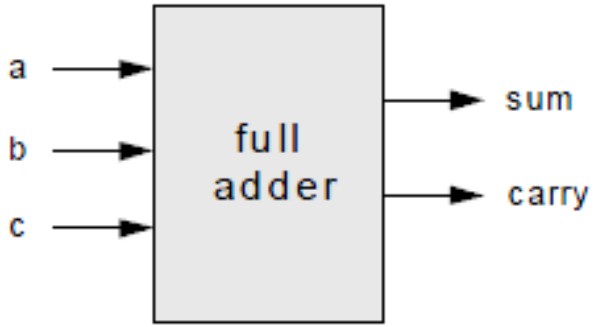
設計出 16 位元加法器

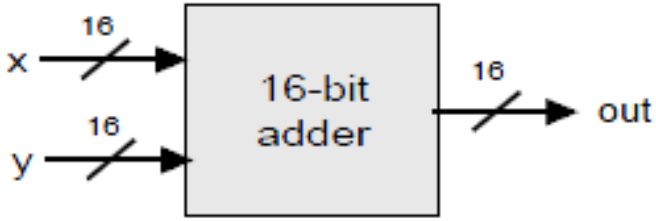


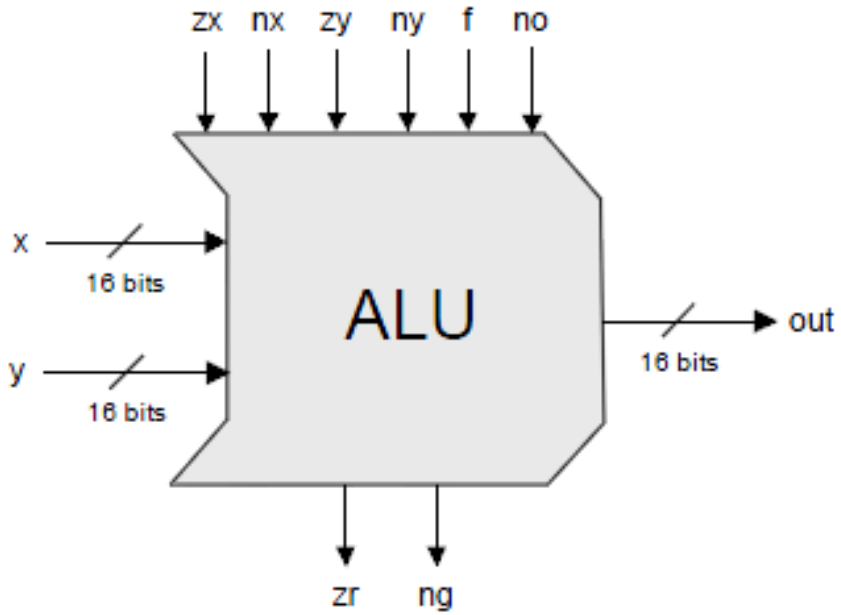
…	1	0	1	1	a
					+
…	0	0	1	0	b
…	1	1	0	1	out

然後就可以更上一層樓，設計出 ALU 了









out(x, y, control bits) =

x+y, x-y, y-x,

0, 1, -1,

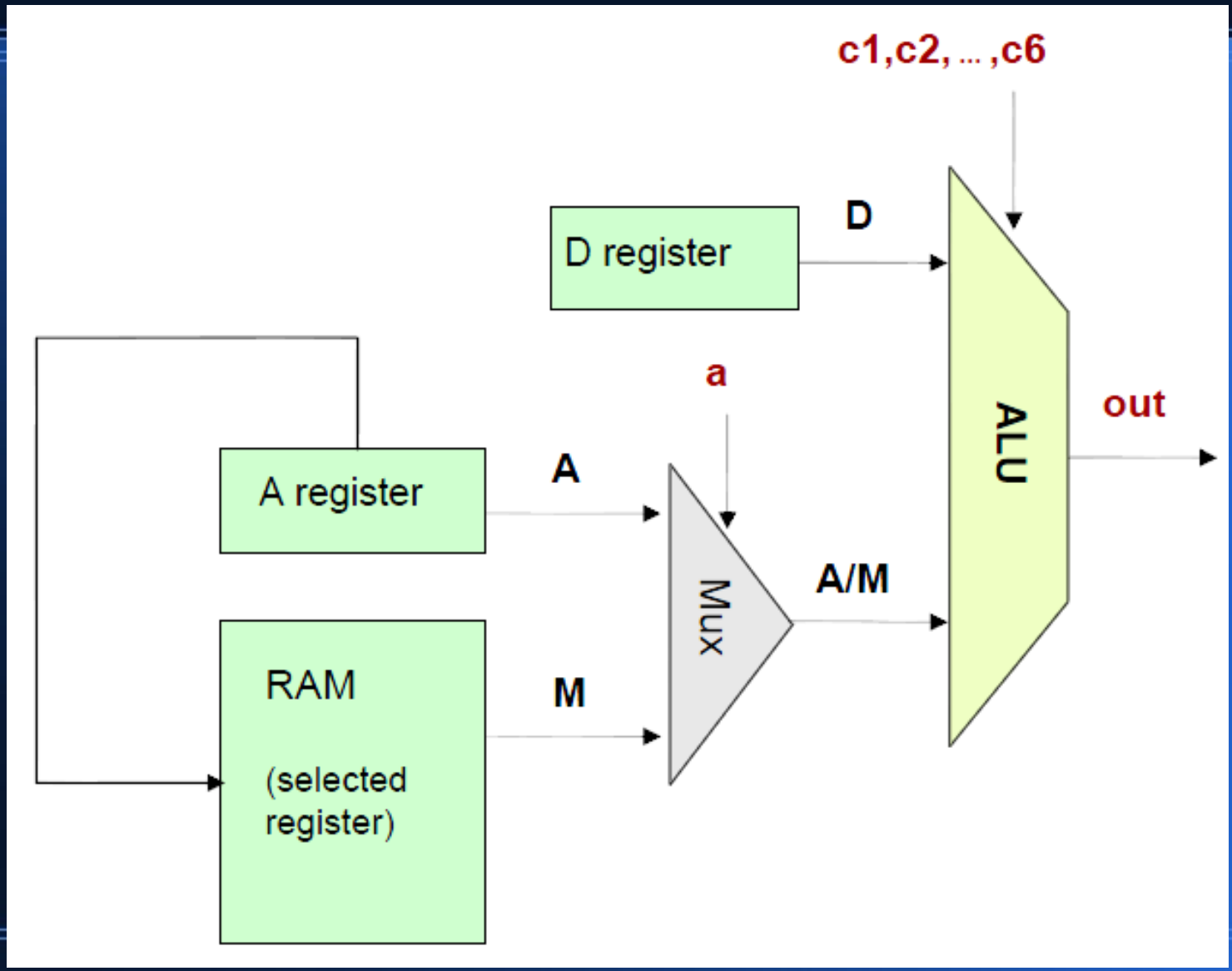
x, y, -x, -y,

x!, y!,

x+1, y+1, x-1, y-1,

x&y, x|y

接著再繼續向上提升，設計出 CPU

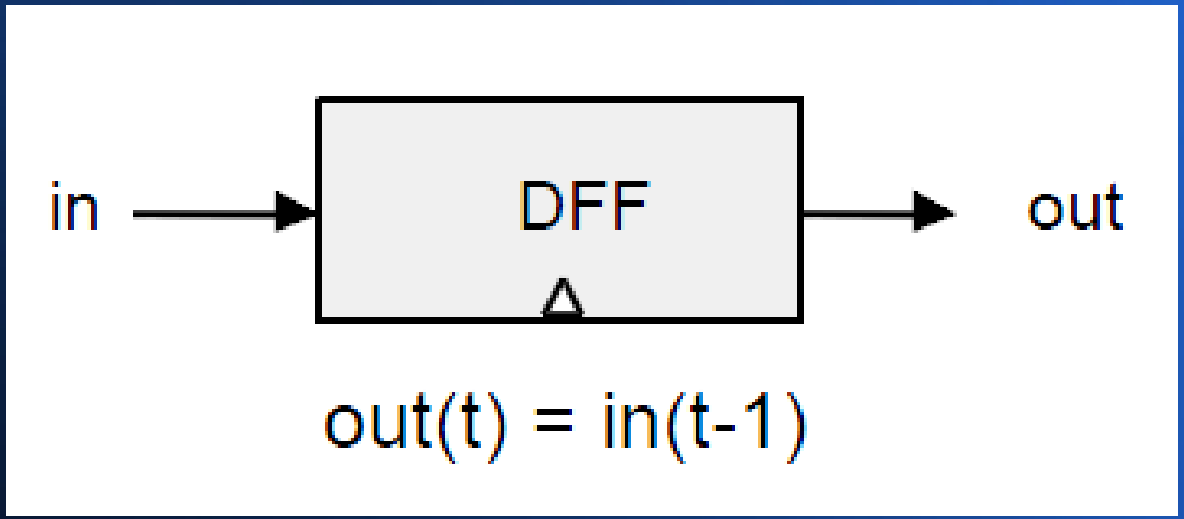


但是、請先不要太急

- 因為從 ALU 到 CPU 之間
- 還缺了一個重要的元素
- 那就是《記憶單元》
- 特別是《暫存器》

要做出暫存器

- 必須先做出一位元的 D 型正反器



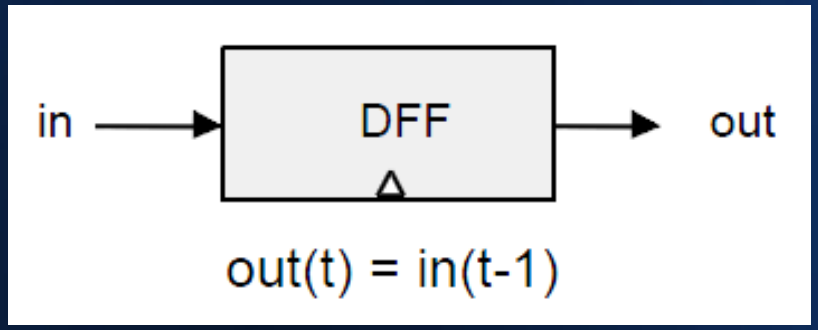
- 而且最好是邊緣觸發型的！

至於怎麼做出邊緣觸發 D 型正反器呢？

- 關於這件事請參考《數位邏輯》課本！
  - 還記得 SR 正反器，JK 正反器
  - 還有《主從式正反器》嗎？
    - 那就是邊緣觸發正反器了
  - 不過也可以直接加上脈衝偵測電路
    - 這樣就不需要用主從架構了

在 nand2tetris 這門課中

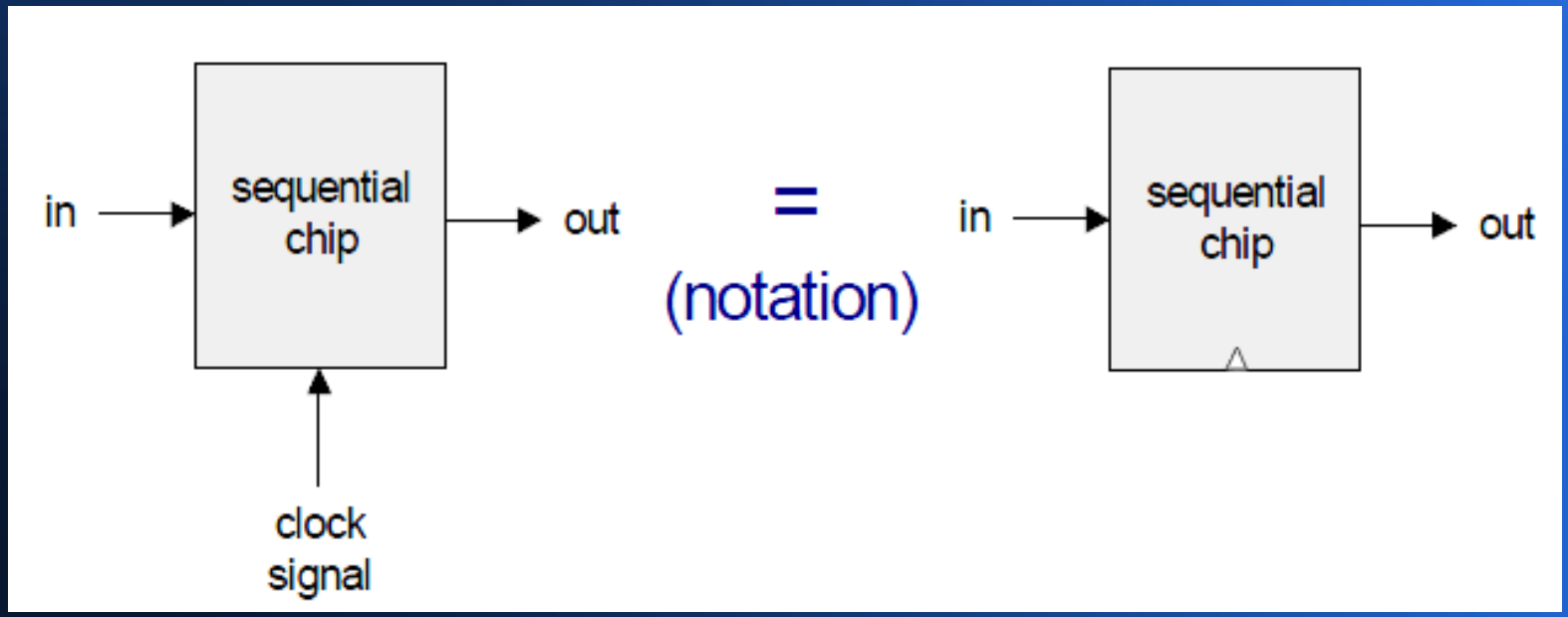
- 老師很好心的給了我們
  - DFF 這種 D 型邊緣觸發正反器
  - 讓我們可以跳過這一段的實作



不過大家最好還是翻翻數位邏輯課本，
才不會有那種不踏實的感覺。

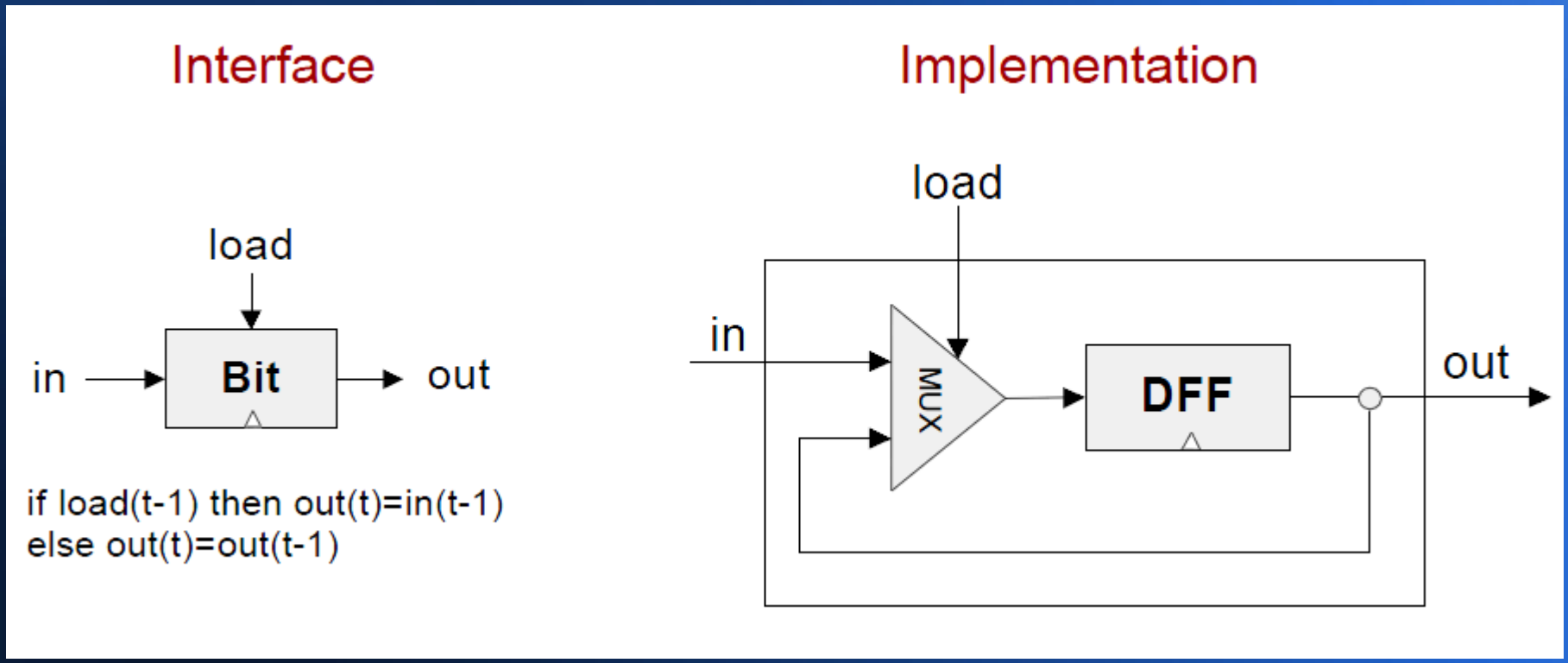
還有請記得，《邊緣觸發》的元件

- 其實都隱含了時脈 clock 的概念，只是 clock 線都改用了一個小三角形代替而已。



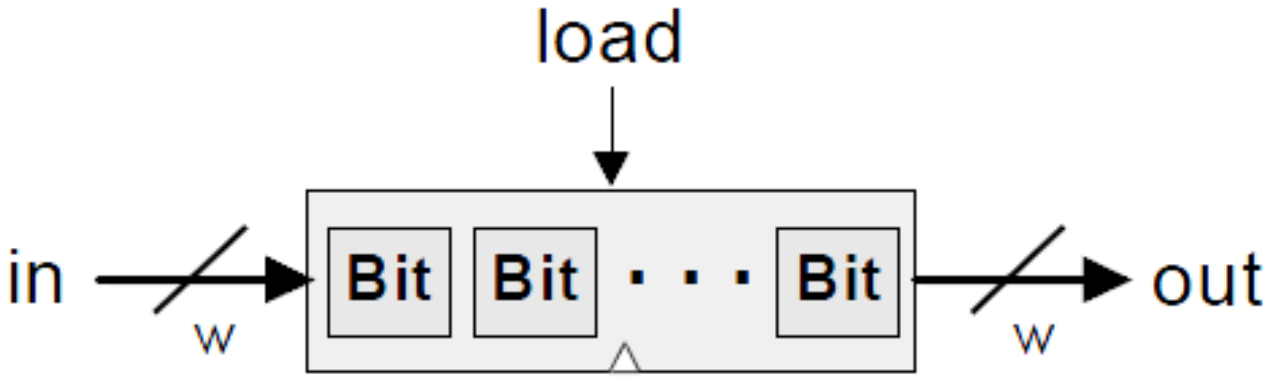
有了D型正反器(DFF)

- 就可以做出一位元存儲器



然後就可以做出寬度為 w 位元的存儲器

- 也就是暫存器了



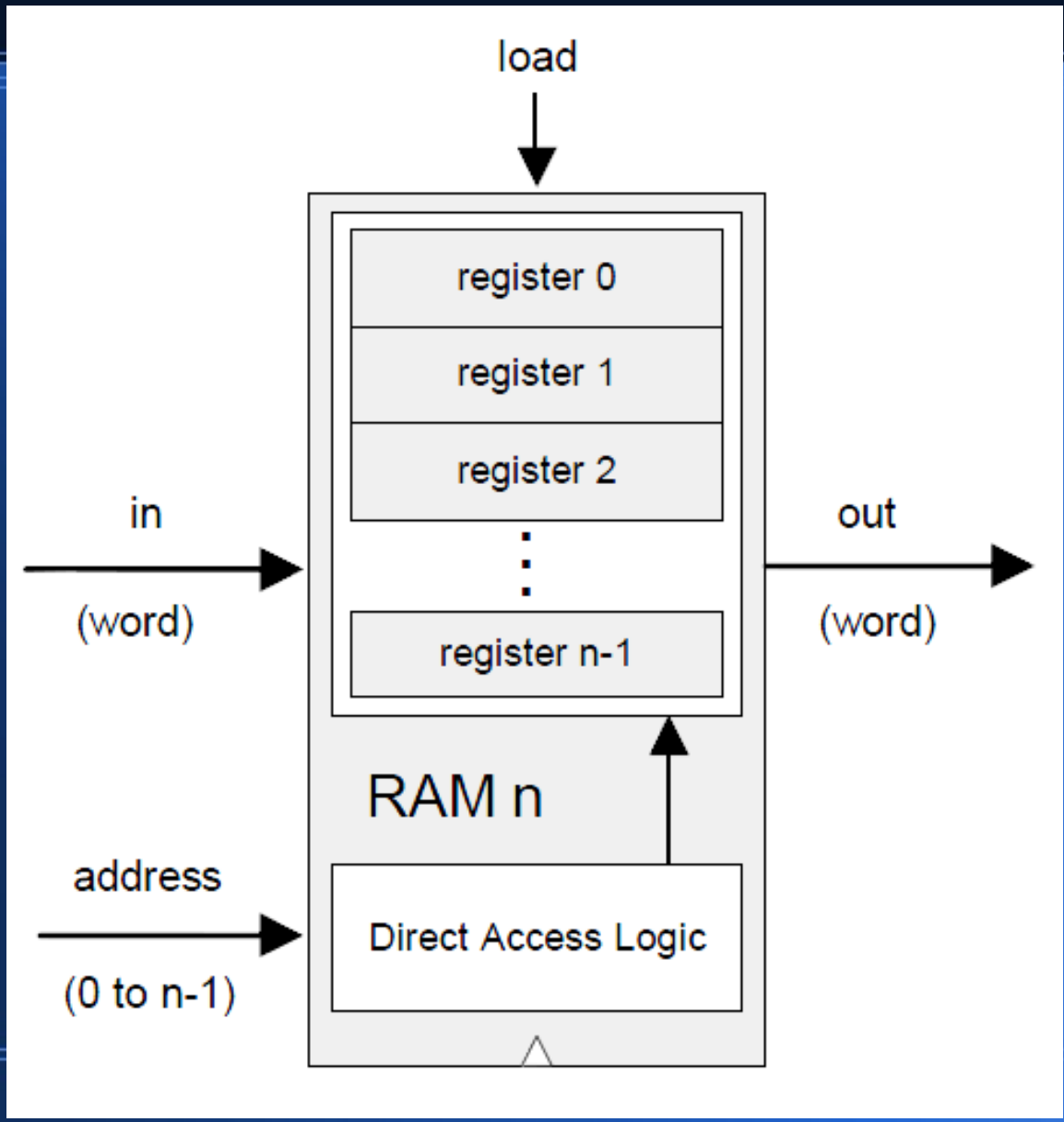
if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

w-bit register

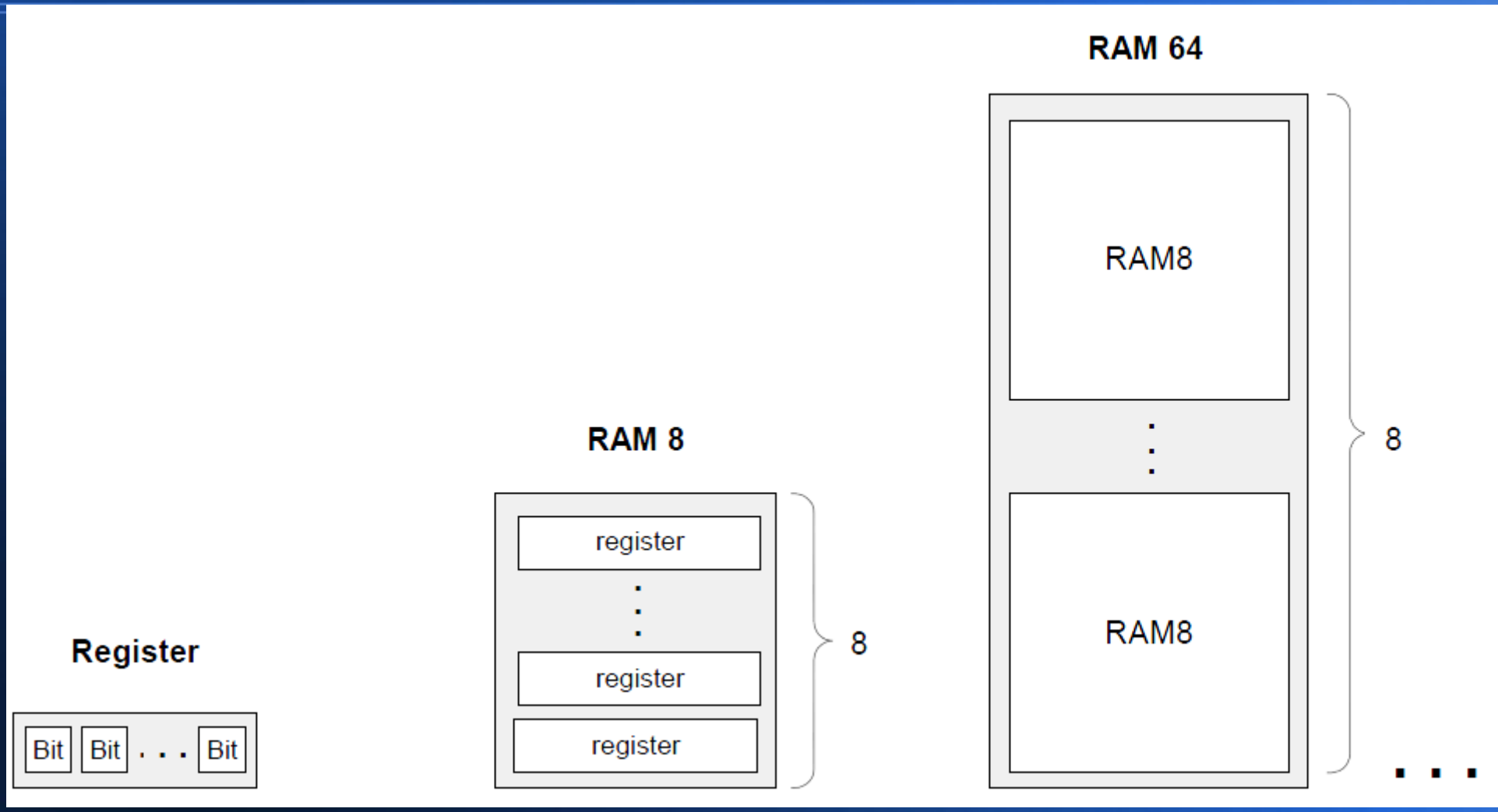
接著只要將很多暫存器集合起來

- 加上控制線路

  就可以做出記憶體



我們可以從 8 字組、64 字組一路上升



直到做出 16K 記憶體為止

- 這樣就足夠本課程使用了
- 因為 HackCPU 位址線只有 15 條，最大定址空間為 32K

到目前為止

- 我們已經有了 ALU 、暫存器、記憶體、還有多工器解碼器等元件
- 距離設計 CPU 已經是萬事俱備，只欠東風了！

問題是、東風到底在哪裡呢？

對於設計 CPU 而言

- 那個東風就是《指令集》
  - 英文是 Instruction Set
- 有了指令集，我們才能開始設計處理器

但是、CPU和指令集

- 是一個《雞生蛋、蛋生雞》的問題
  - 沒有 CPU ，哪來的指令集
  - 沒有指令集，又怎麼設計 CPU 呢？

所以、CPU和指令集要一起設計

但是對於一個新手而言

- 這又怎麼做得到呢？

還好

- 我們有老師！
- 老師已經設計出了 CPU 和指令集
- 我們只要看懂指令集後再來設計 CPU 就行了。

為了看懂指令集

- 我們必須學習組合語言
- 而且是 HackCPU 的組合語言

但是、HackCPU 的組合語言有點怪

- 或者說非常怪，超級奇怪…

舉例而言、一般的組合語言可能長這樣

```
// In what follows R1,R2,R3 are registers, PC is program counter,
// and addr is some value.

ADD R1,R2,R3      // R1 ← R2 + R3

ADDI R1,R2,addr   // R1 ← R2 + addr

AND R1,R1,R2      // R1 ← R1 and R2 (bit-wise)

JMP addr          // PC ← addr

JEQ R1,R2,addr    // IF R1 == R2 THEN PC ← addr ELSE PC++

LOAD R1, addr     // R1 ← RAM[addr]

STORE R1, addr    // RAM[addr] ← R1

NOP               // Do nothing

// Etc. - some 50-300 command variants
```

但是 HackCPU 的組合語言長這樣

```
// Adds 1+...+100.
      @i      // i refers to some RAM location
      M=1     // i=1
      @sum    // sum refers to some RAM location
      M=0     // sum=0
(LOOP)
      @i
      D=M     // D = i
      @100
      D=D-A   // D = i - 100
      @END
      D;JGT   // If (i-100) > 0 goto END
      @i
      D=M     // D = i
      @sum
      M=D+M   // sum += i
      @i
      M=M+1   // i++
      @LOOP
      0;JMP   // Got LOOP
 (END)
      @END
      0;JMP   // Infinite loop
```

為何長得這麼奇怪

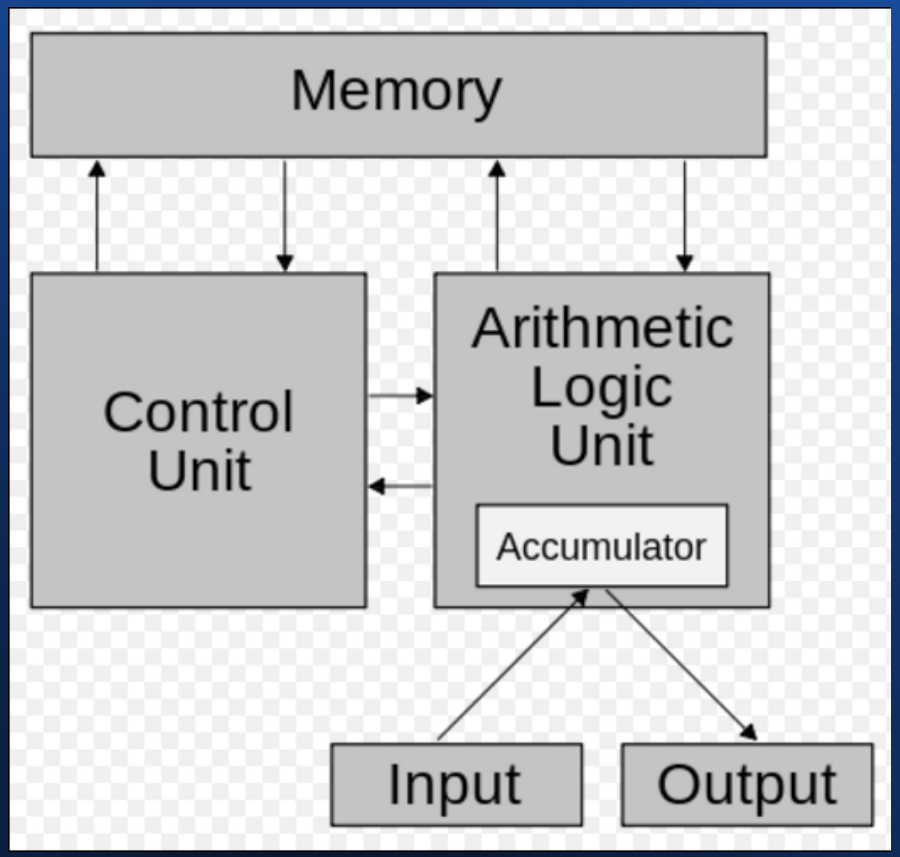
- 這當然是有原因的！

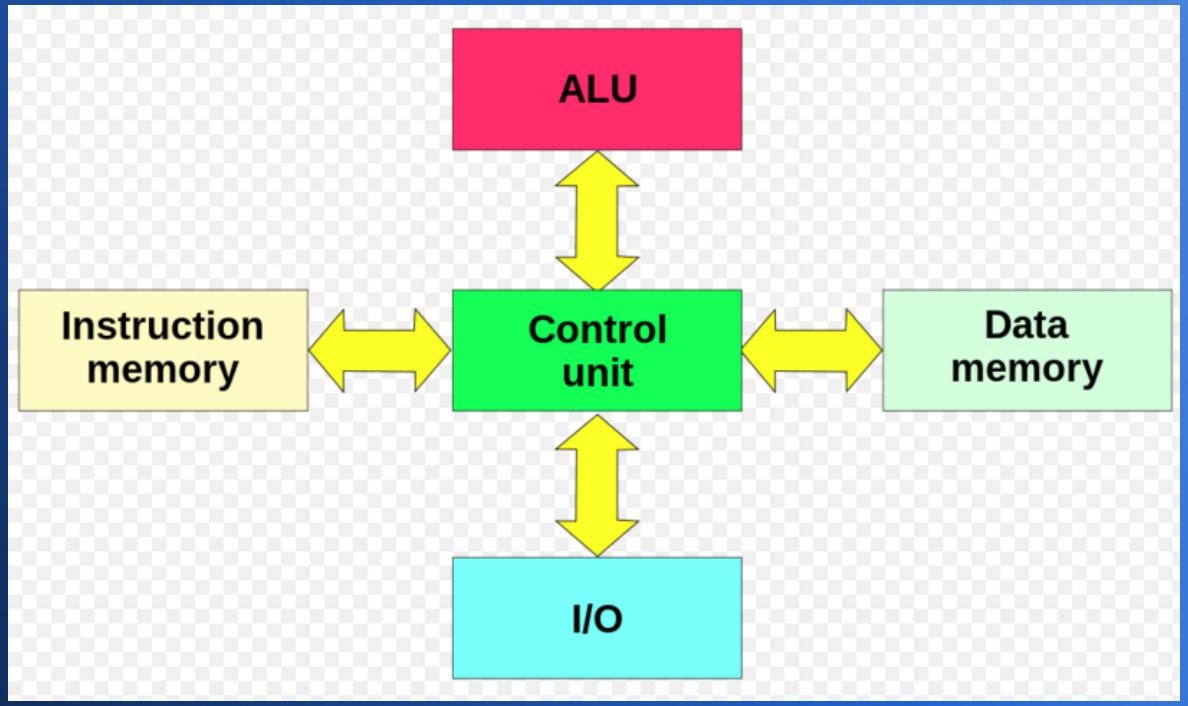
HackCPU 採用的是《哈佛架構》

- 而不是《馮紐曼架構》
- 該架構將《指令》與《資料》分別放在兩個不同的記憶體當中。
- 這樣就可以同時存取指令和資料，而不會互相衝突了。

馮紐曼架構 V.S. 哈佛架構

圖片來自





《指令記憶體》和《資料記憶體》分開，可同時存取

(a) 馮紐曼架構

(b) 哈佛架構

其實自從管線架構盛行之後

- 《哈佛架構》就變得很重要
- 因為管線處理器需要同時存取《指令》和《資料》記憶體
- 於是哈佛架構開始越來越普遍！

不過 HackCPU 採用哈佛架構

- 主要是為了讓 CPU 設計簡單的原因
- 而不是為了用管線增快執行速度
  - 但是勉強來說，HackCPU 可以說是一顆有兩階管線的處理器。
  - （問題是一般管線架構至少 3 階以上，典型的是 5 階，ARM 現在都做到 13 階了）

讓我們暫時忘記

- 那些煩人的管線技術
- 先專注在 HackCPU 的指令集與處理器上

HackCPU 的指令

- 通常要兩個一組，成對的看

Used for:	Coding example:
Entering a constant value (A = value)	`@17 // A = 17` `D = A // D = 17`
Selecting a RAM location (register = RAM[A])	`@17 // A = 17` `D = M // D = RAM[17]`

Selecting a ROM location (PC = A)	Later
	`@17 // A = 17`
	`JMP // fetch the instruction`
	`// stored in ROM[17]`

舉例而言

- 前面那個有 @ 的指令，是用來定址的，稱為 A 型指令
- 後面那個指令，是用來計算的，稱為 C 型指令

```
@17     // A = 17
D = A   // D = 17
```

```
@17     // A = 17
D = M   // D = RAM[17]
```

- 後面 C 型指令中的 A, M=M[A] 都會受前面的 A 型指令影響

同樣的，跳躍指令也會受 A 型指令影響

- 因為會跳到 A 型指令所指定的位址

Later

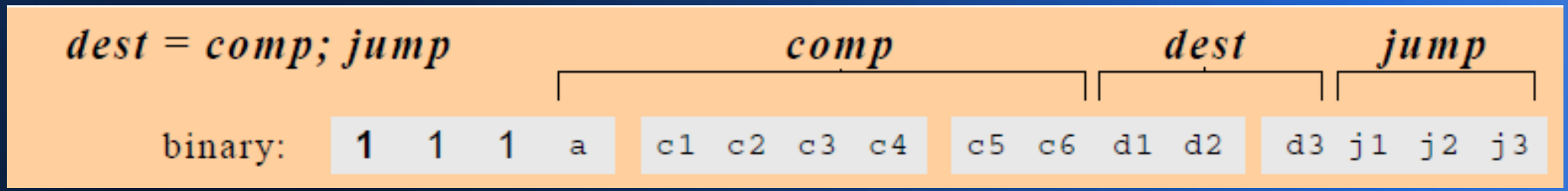
```
@17     // A = 17
JMP     // fetch the instruction
        // stored in ROM[17]
```

有了這個概念後，你應該就能看懂下列程式了

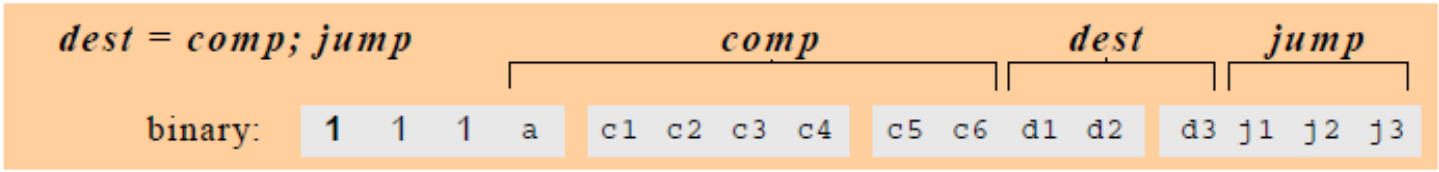
```
// Adds 1+...+100.
      @i      // i refers to some RAM location
      M=1     // i=1
      @sum    // sum refers to some RAM location
      M=0     // sum=0
(LOOP)
      @i
      D=M     // D = i
      @100
      D=D-A   // D = i - 100
      @END
      D;JGT   // If (i-100) > 0 goto END
      @i
      D=M     // D = i
      @sum
      M=D+M   // sum += i
      @i
      M=M+1   // i++
      @LOOP
      0;JMP   // Got LOOP
 (END)
      @END
      0;JMP   // Infinite loop
```

這些指令對應的機器碼格式如下

- A 型： 0 + address[14..0]
- C 型： 111 + comp + dest + jump



C 型指令的編碼表如下

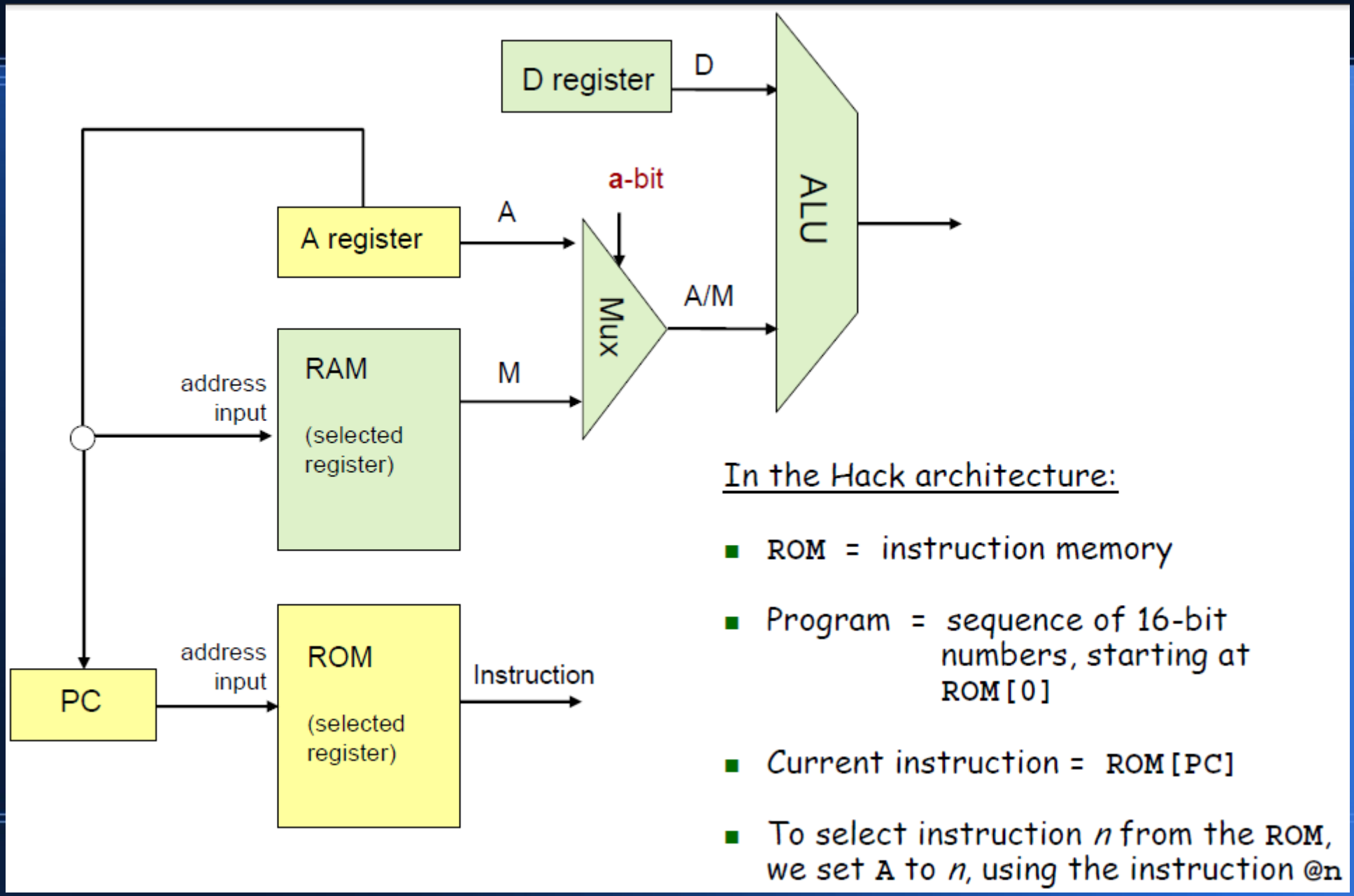


(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

j1 (*out* < 0)	j2 (*out* = 0)	j3 (*out* > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If *out* > 0 jump
0	1	0	JEQ	If *out* = 0 jump
0	1	1	JGE	If *out* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* ≠ 0 jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

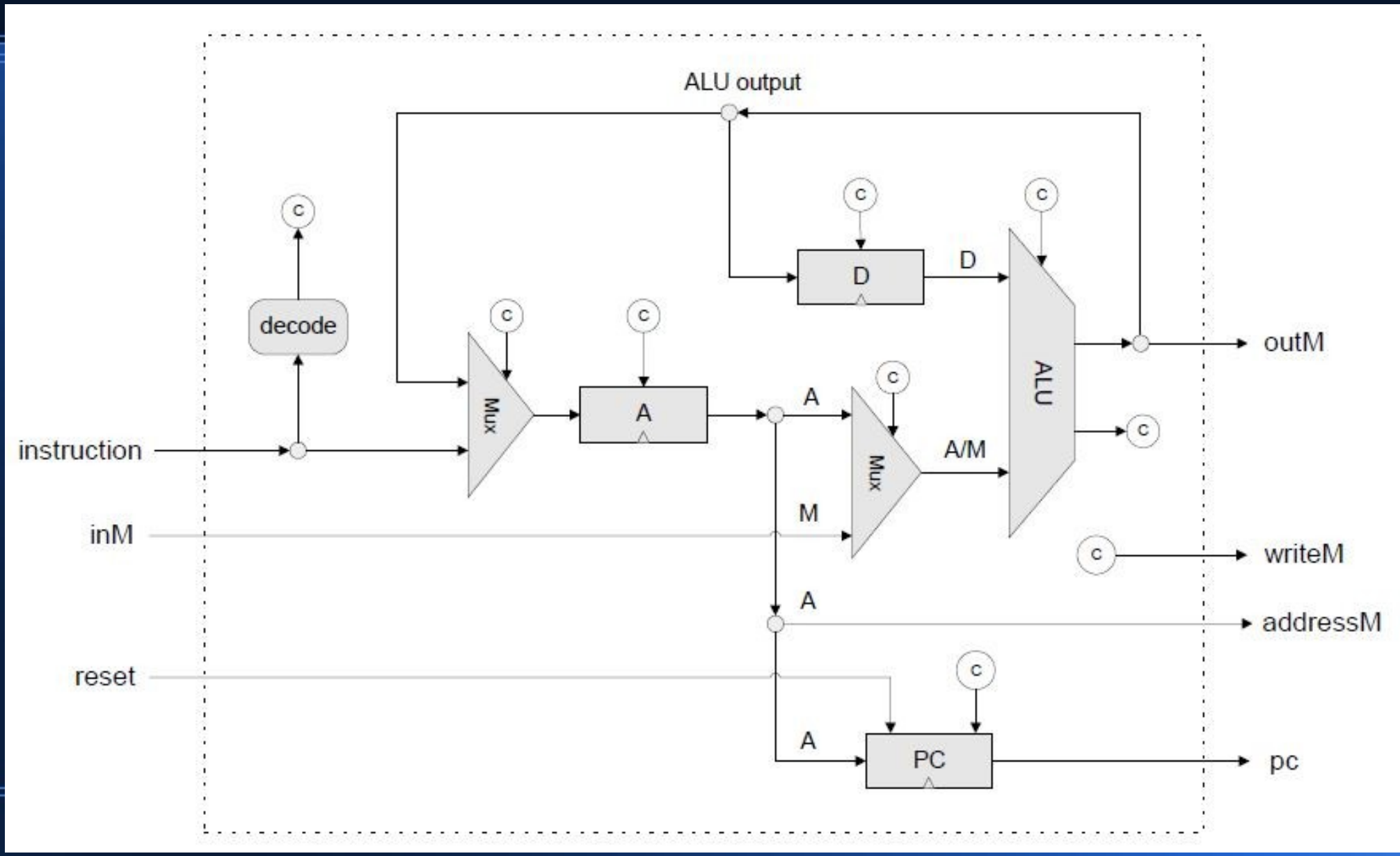
然後、我們就可以開始研究 HackCPU 的架構了



In the Hack architecture:

- ROM = instruction memory
- Program = sequence of 16-bit numbers, starting at ROM[0]
- Current instruction = ROM[PC]
- To select instruction *n* from the ROM, we set A to *n*, using the instruction @n

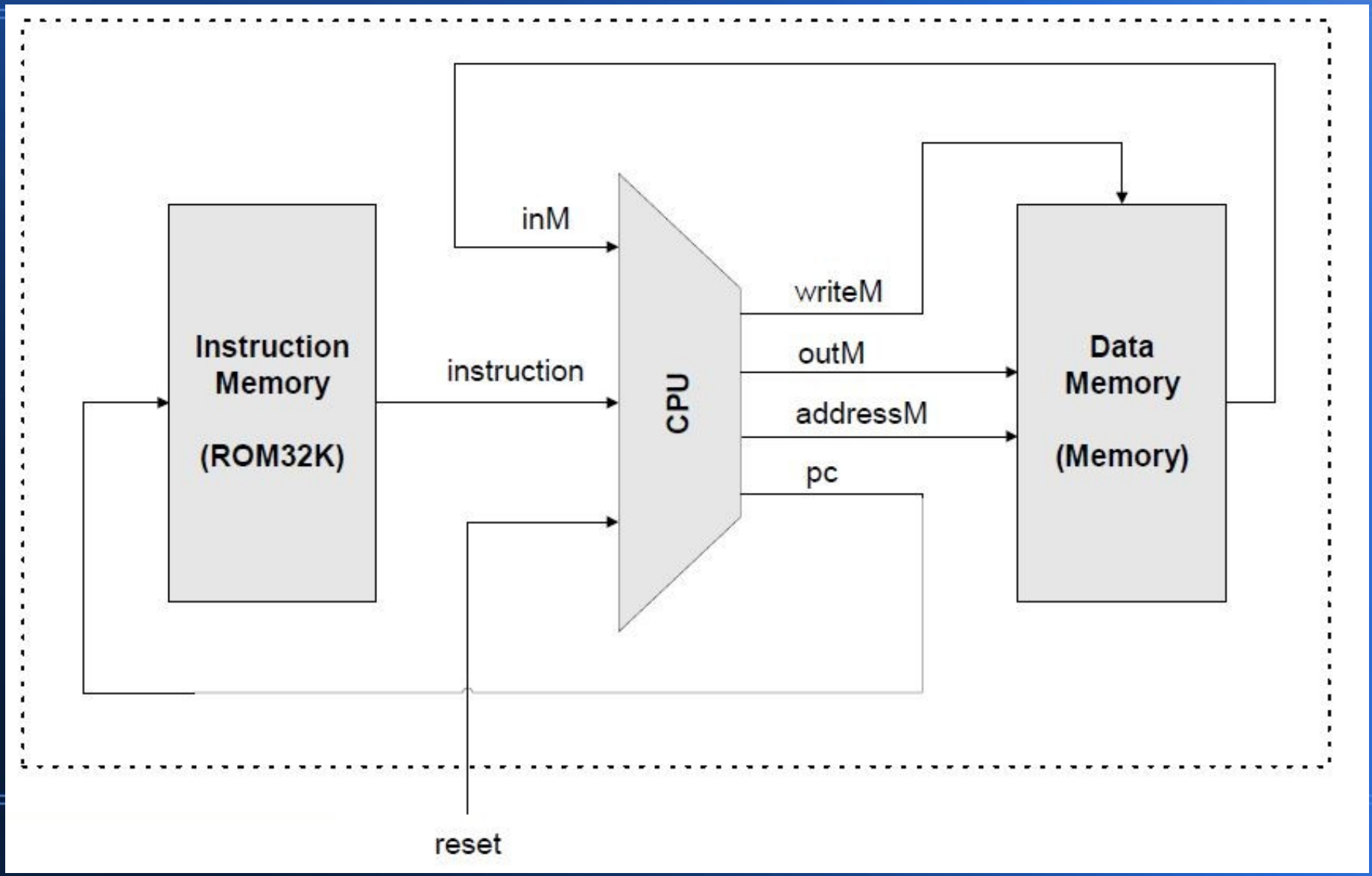
HackCPU 的詳細架構如下



在上圖中

- 有個 decode 解碼單元，還有很多 ⓒ 符號的控制線路
- 這些都是你要在習題中去設計的

把 CPU 和記憶體整合起來，就是一台完整的電腦了



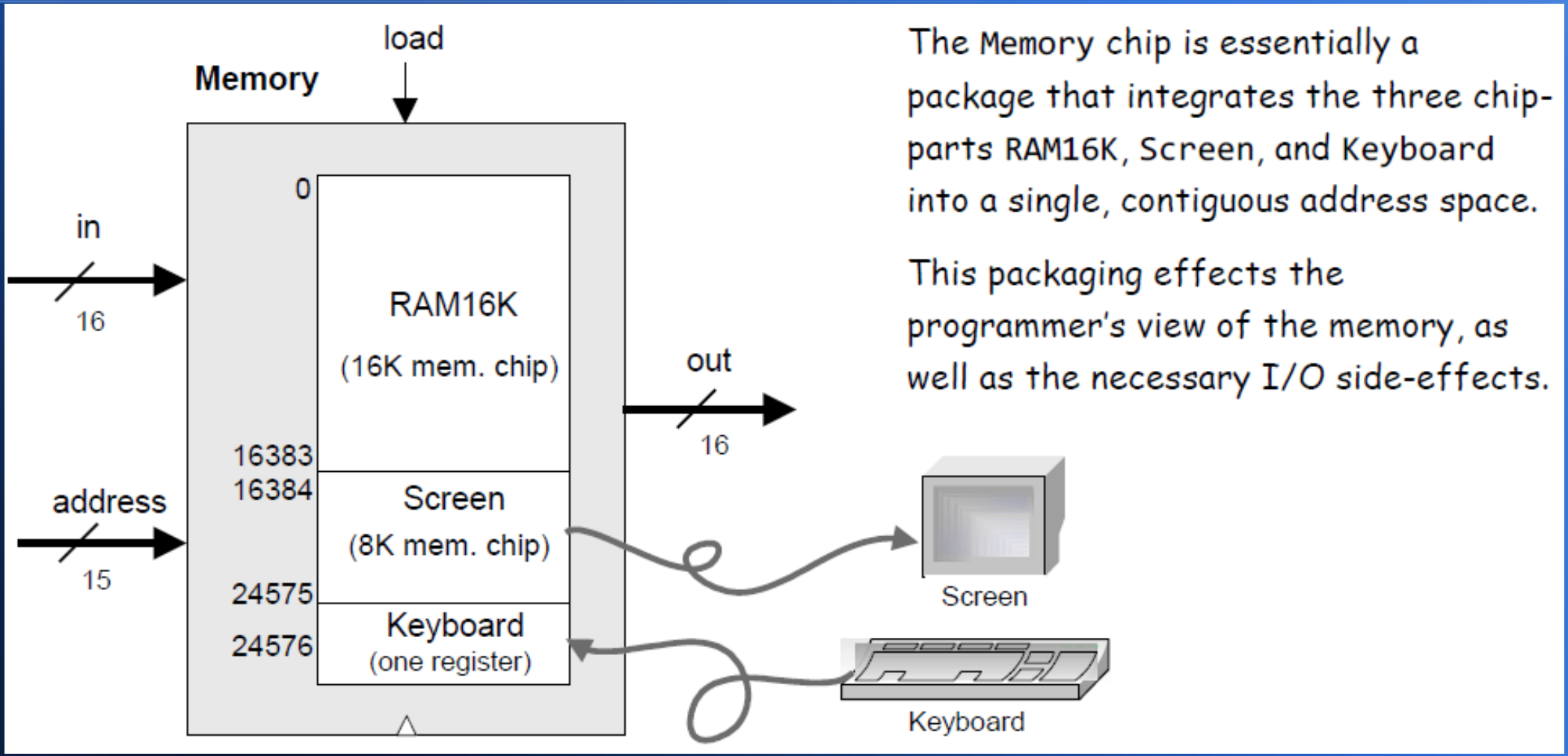
這時、一定會有人問說

- 那這台電腦該怎麼做輸出入呢？

這個問題其實不難

- 上述的 Hack Computer
  - 採用記憶體映射輸出入
  - 包含一個鍵盤和黑白螢幕

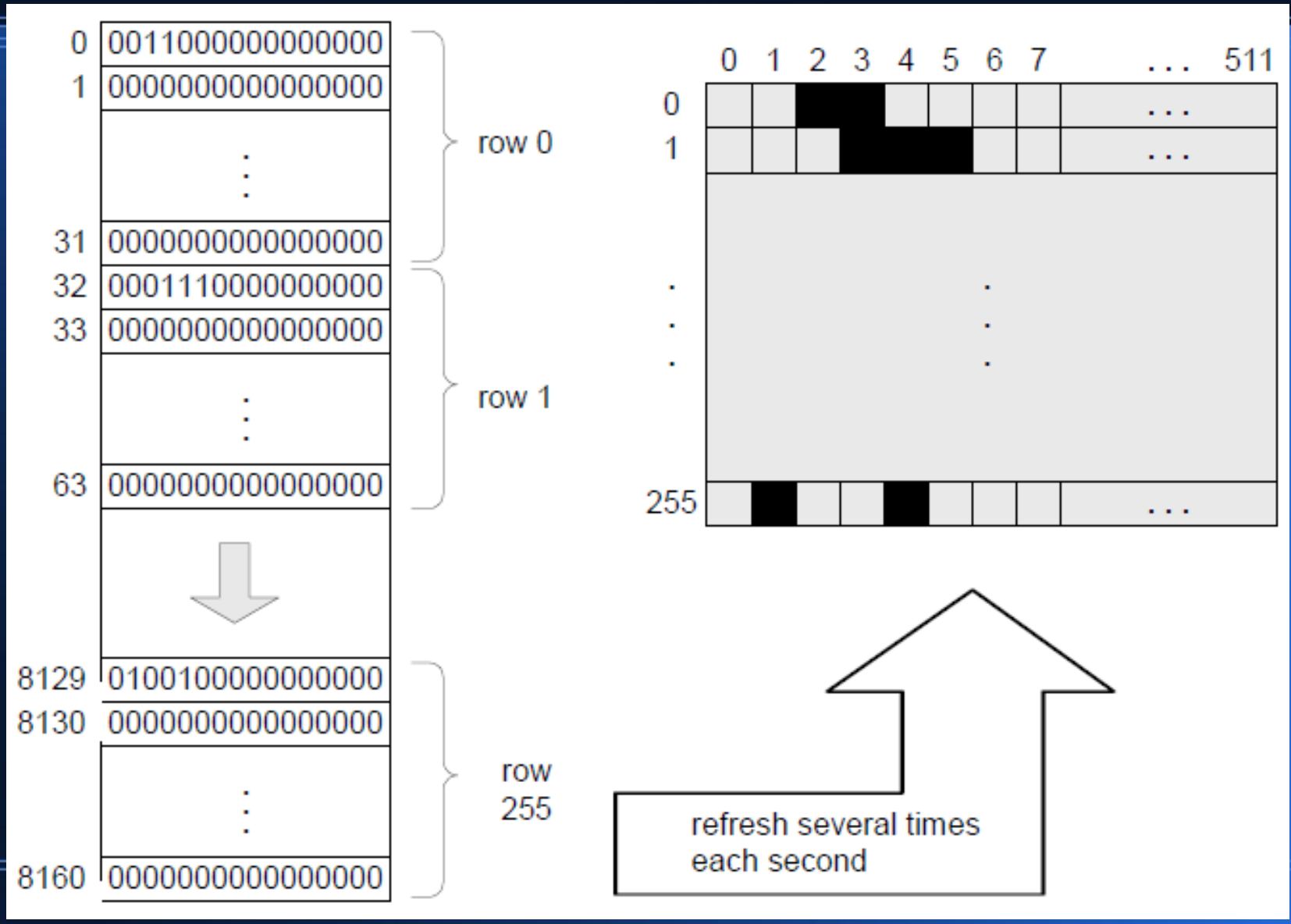
其記憶體映射配置如下



The Memory chip is essentially a package that integrates the three chip-parts RAM16K, Screen, and Keyboard into a single, contiguous address space.

This packaging effects the programmer's view of the memory, as well as the necessary I/O side-effects.

更詳細的螢幕映射方式如下

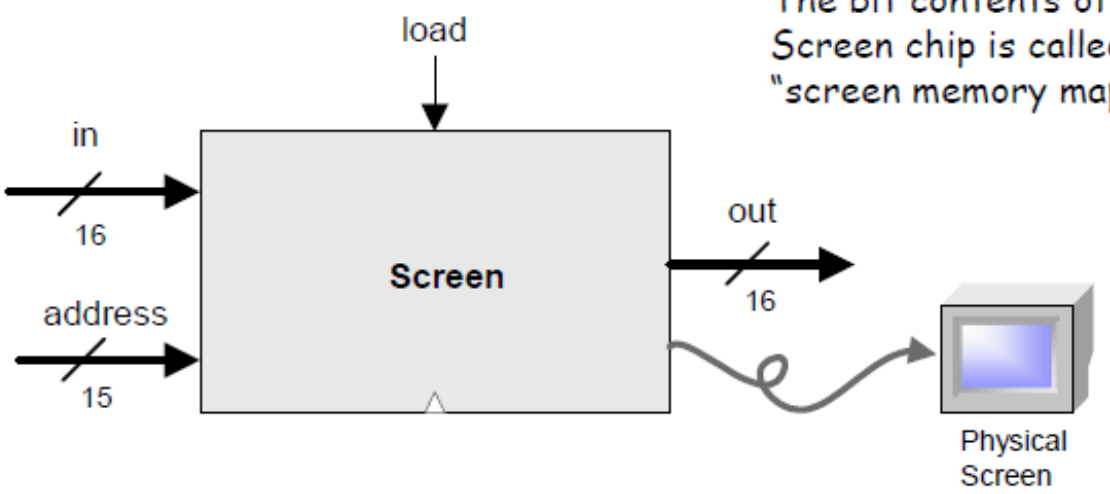


於是你只要讀取或寫入記憶體

- 就可以和輸出入裝置進行溝通了

而且、老師們還很溫馨的請人寫了模擬器

讓你的組合語言程式可以轉換成機器碼後執行，還有模擬螢幕和鍵盤喔！



The bit contents of the Screen chip is called the "screen memory map"

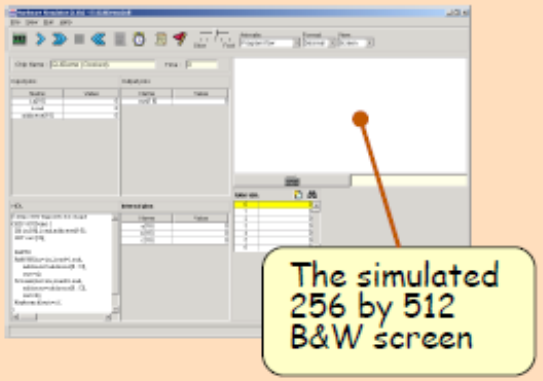
The Screen chip has a basic RAM chip functionality:

- read logic: `out = Screen[address]`
- write logic: `if load then Screen[address] = in`

Side effect:

Continuously refreshes a 256 by 512 black-and-white screen device

Simulated screen:



When loaded into the hardware simulator, the built-in `Screen.hdl` chip opens up a screen window; the simulator then refreshes this window from the screen memory map several times each second.

這樣

- 你就可以測試自己寫的組合語言是否正確了
- 真的是太溫馨了阿！

當然、習題裏一定會有

- 讓你可以測試輸出入是否正確的題目
- 那個習題就是：
  - 請你寫出一個組合語言程式，當偵測到鍵盤被按下時，就讓螢幕反白

寫完第 4 章的這些組合語言習題之後

- 老師才讓我們開始設計 CPU 和電腦

這樣

- 是不是太溫馨了呢？

非常歡迎

- 大家一起加入 nand2tetris 的行列
- 開始學習如何設計自己的電腦！

我已經幫大家開好 Facebook 社團了

- 您可以從社團裡得到更多的相關資訊。
- 社團網址如下：
  - https://www.facebook.com/groups/nand2tetris/

現在、就加入社團和我們一起學習

- 如何設計一台電腦的硬體
- 還有所有軟體，包含：
  - 組譯器、虛擬機、編譯器
  - 作業系統

nand2tetris

等你喔！