系統程式-期末報告

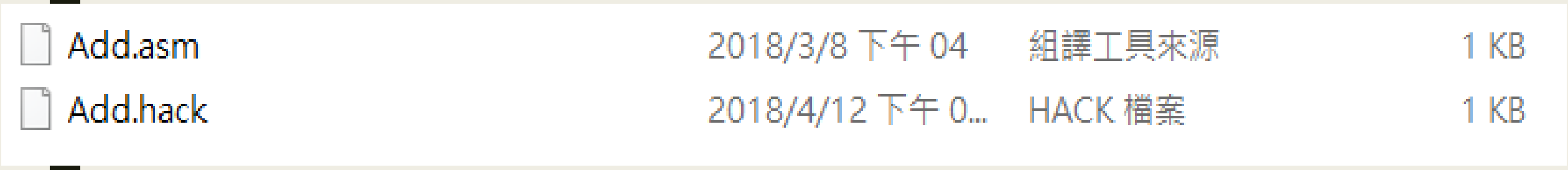
資工三 周明萱
資工二 何羽倩

組譯器

組譯器用來翻譯組合語言的程式，並將其轉譯成可執行的機械碼

-輸入為組合語言的原始程式(.asm)

-輸出則變成可執行的機械碼(.hack)



Predefined Symbols Any Hack assembly program is allowed to use the following predefined symbols.

Label	*RAM address*	*(hexa)*
SP	0	0x0000
LCL	1	0x0001
ARG	2	0x0002
THIS	3	0x0003
THAT	4	0x0004
R0-R15	0-15	0x0000-f
SCREEN	16384	0x4000
KBD	24576	0x6000

組合語言就是包含了L指令、A指令與C指令
我們為了做出組譯器，
第一步就是將此表能成功地將組合語言轉成機械碼

The *C*-instruction revisited

dest = comp; jump

	comp								*dest*			*jump*		
binary:	1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

(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

The A-instruction

`@value` `// A ← value`

Where *value* is either a number or a symbol referring to some number.

Used for:

- 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]
```

將表格轉換為JS的字典，這邊就是程式碼與表格的對照

```
var ctable = {
    "0": 0b0101010,
    "1": 0b0111111,
    "-1": 0b0111010,
    "D": 0b0001100,
    "A": 0b0110000,
    "M": 0b1110000,
    "!D": 0b0001101,
    "!A": 0b0110001,
    "!M": 0b1110001,
    "-D": 0b0001111,
    "-A": 0b0110011,
    "-M": 0b1110011,
    "D+1": 0b0011111,
    "A+1": 0b0110111,
    "M+1": 0b1110111,
    "D-1": 0b0001110,
    "A-1": 0b0110010,
    "M-1": 0b1110010,
    "D+A": 0b0000010,
    "D+M": 0b1000010,
    "D-A": 0b0010011,
    "D-M": 0b1010011,
    "A-D": 0b0000111,
    "M-D": 0b1000111,
    "D&A": 0b0000000,
    "D&M": 0b1000000,
    "D|A": 0b0010101,
    "D|M": 0b1010101
}
```

The C-instruction revisited

dest = comp; jump

	1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3
binary:				*comp*							*dest*			*jump*		

(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


```
var dtable = {
    "": 0b000,
    "M": 0b001,
    "D": 0b010,
    "MD": 0b011,
    "A": 0b100,
    "AM": 0b101,
    "AD": 0b110,
    "AMD": 0b111
}
```

```
var jtable = {
    "": 0b000,
    "JGT": 0b001,
    "JEQ": 0b010,
    "JGE": 0b011,
    "JLT": 0b100,
    "JNE": 0b101,
    "JLE": 0b110,
    "JMP": 0b111
}
```

L指令

- 符號表大致上預先會有圖中所示的東西
- 注意：這裡雖然使用十進位作紀錄，但是你可以使用任何方式紀錄它
- L指令的()中的符號會被記錄在這裡跟它們標記的位址一起
- 後面的A指令也會用到這個表

Predefined Symbols Any Hack assembly program is all ing predefined symbols.

Label	*RAM address*	*(hexa)*
SP	0	0x0000
LCL	1	0x0001
ARG	2	0x0002
THIS	3	0x0003
THAT	4	0x0004
R0-R15	0-15	0x0000-f
SCREEN	16384	0x4000
KBD	24576	0x6000

```
var symTable = {
    "R0": 0,
    "R1": 1,
    "R2": 2,
    "R3": 3,
    "R4": 4,
    "R5": 5,
    "R6": 6,
    "R7": 7,
    "R8": 8,
    "R9": 9,
    "R10": 10,
    "R11": 11,
    "R12": 12,
    "R13": 13,
    "R14": 14,
    "R15": 15,
    "SP": 0,
    "LCL": 1,
    "ARG": 2,
    "THIS": 3,
    "THAT": 4,
    "KBD": 24576,
    "SCREEN": 16384
};
```

A指令

- 有@就是A指令
- 如果是數字直接轉16bit的二進位輸出最多到0111 1111 1111 1111
- 如果不是數字就查符號表，若已存在輸出看你的符號表需不需要轉二進位，不需要就直接輸出
- 如果不存在在表中建立資料，注意從16(0000 0000 0001 0000) 開始 0到15是R的保留紀錄到符號表之後記得還是要輸出
- 簡單來說在不是數字的情況下A指令做的就是 查表→輸出 或是 查表→不存在→紀錄在符號表中並輸出(誰先誰後沒有差別)
- 記得輸出的時候要左補零到16bit

C指令

- 在第二階段查表輸出階段 由於 L指令在第一階段作完了，所以程式碼可以簡單地寫成不是A指令就是C指令
- 要注意這個前提建立在，你已經把註解跟L指令從要輸入到第二階段的資料中拿掉了
- C指令的標準型式為dest = comp ; jump
  當dest不存在 “=”可以省略，當jump不存在 ”;”可以省略
  你可能會看見dest = comp或是comp ; jump 其實全部一起出現的機率比較少，但還是會出現，所以要作(全出現還比較好作就是了)
- 機器碼型式為binary:{111a,c1c2c3c4,c5c6d1d2,d3j1j2j3}

如果用中文比喻

組譯器就是電腦要看懂人一句話所需要的工具，但電腦由於看不懂一個句子

因此需要一個工具來做拆解、查表、轉譯、合併及輸出，工具當中會有各種自己建立字典對應的詞或位置

而一個句子中如同 ”今天的天氣是@晴天”，首先是拆解-"今天” 、“的”、”天氣”、”是” 、”@晴天”

查表後發現首先是”天氣”是L指令也就是Label指令，電腦本身原本就會的東西可以直接做輸出

而”@晴天”則屬於address指令利用程式碼去做轉換位址的動作

最後則是Control/Compute指令每個C指令都可以拆分為三個部分，通常會出現兩個PART(部分) dest = comp; jump

像是”天氣”兩個字，天跟氣就有在字典中對應的位置，在查TABLE作轉譯後即可輸出

最後再重新組成電腦看得懂的機械碼就是組譯器的主要過程

組譯器主要執行兩個步驟

再來是如果用執行過程來說，可以分成兩大部分

首先是PASS1計算位址，判斷各個機械碼在程式中的位址

接著是PASS2主程式依據L,A,C三種指令轉為機械碼

補充：正規表達式 用於分解字串
通常用”//”來包住正規表達式的主體運行碼
工具中用於全域來拆解輸入的字句

實機測試