**系統程式-課程筆記**

**˙**指令

|  |  |
| --- | --- |
| gcc sum.c –o sum | 用gcc編譯器去編譯sum.c程式，輸出一個執行檔sum |
| ./sum | 執行sum |

˙把codeblocks編譯器加進vs code：

1. 找到資料夾......codeblocks\MinGW\bin並複製路徑
2. 去控制台，選擇系統及安全性->系統->進階系統設定
3. 環境變數，選擇系統變數的path->新增並貼上路徑->重開vs code

˙hash雜湊：把一個字串用固定的方式轉成一種數字

ex:

usigned int h = 37

h = h\*147 + \*p; //\*p是ASCII碼

hash( )=37 hash(h) = 5543 //5543=37\*147+104(h的ASCII)

hash(he) = 814922 = 5543\*147+101 //e是101的ASCII

***04-map***

**(main.c)循序搜尋**

mapNew(&jMap, 17);

呼叫mapNew : 建立一個大小為17的表格，變數名為jmap

jMap.table = jList; : 把jList塞進去

jMap.top = 8; : 有8個元素

mapLookup(&jMap, "JLE"); : 在jMap裡尋找”JLE”結構

**(map.c)**

map->size = size; : 在main.c裡設為17，東西不能塞超過17個

mapFind : 在map陣列裡找

***生成語法***

N = cat | dog |:or的意思

S = N Y

句子 = 名詞 動詞

V = run | eat

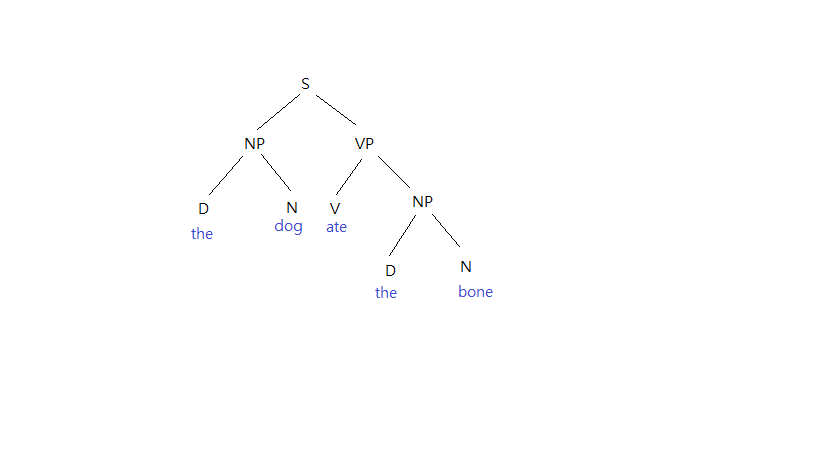
N->dog , V->run產生dog run

˙BNF語法

S = NP VP 句子 = 名詞子句 + 動詞子句

NP = D N 名詞子句 = 定詞 + 名詞

VP = V NP 動詞子句 = 動詞 + 名詞子句

語法樹：

E = F( [ + - ] )\*

E:運算式 F:Factor ( )\*:可出現0次以上

F = Number | ‘ ( ‘ E ‘ ) ‘

//E只能寫0~9 ex:’ 3 + 5 ‘

***運算式編譯器 (exp0)***

**ex:**

E = F( [ + - ] )\* //假設輸入’ 3 + 5 ‘

F = Number | ‘ ( ‘ E ‘ ) ‘

parse: 3 + 5

**結果：**

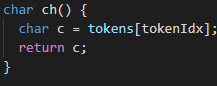
t0 = 3 //t0是一個暫存器

t1 = 5 //t1存5

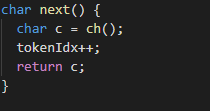
t2 = t0 + t1 //t2 = t0 + t1 = 3 + 5，最後產生的東西放在t2

**部分程式碼註解：**

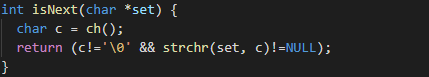
// 取得目前字元

******

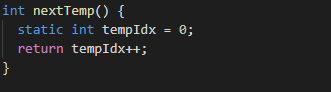
// 取得目前字元，同時進到下一格



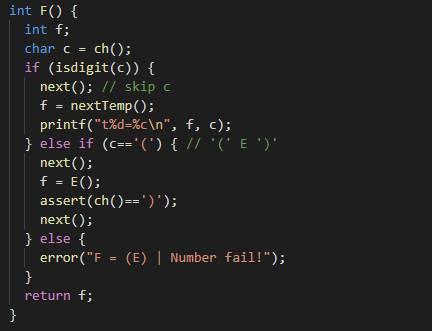
// ex: isNext("+-") 用來判斷下一個字元是不是 + 或 -



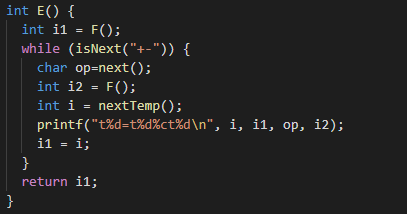
// 產生下一個臨時變數的代號， ex: 3 代表 t3。

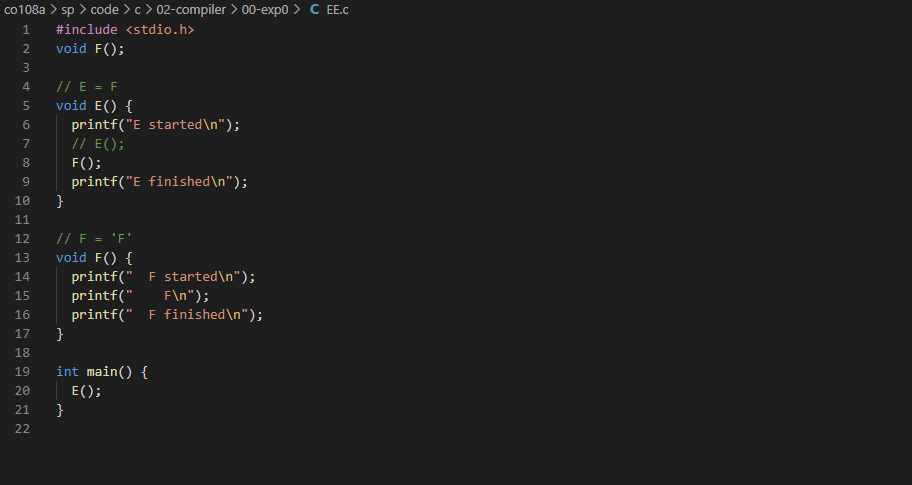


// F = Number | '(' E ')'



// E = F ([+-] F)\*



**輸入指令：gcc exp0.c –o exp0 : 編譯出一個執行檔exp0.exe**

主程式呼叫一個E

E印出E started

E呼叫了F，此F就是下面的F，但F比後面定義，但在前面就呼叫了，所以前面要加一個void F ( )

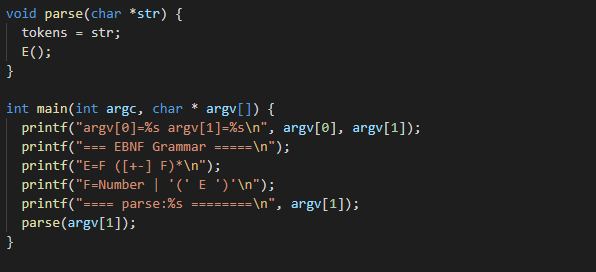
**結果：**

E started

F started

F finished

E finished

parse(argv[1]); //把argv[1]傳進parse函數

arg : 參數 argc : 參數個數 argv : 參數變數，陣列

argv[1] : 第一個參數

./lexer 第0個參數 sum.c第一個參數

˙若輸入./exp0 ‘ x + 5 – y ‘就會編譯中間碼

#t0 = x , @x (變數), D = M / @5 (數字), D =A , #t1 = 5

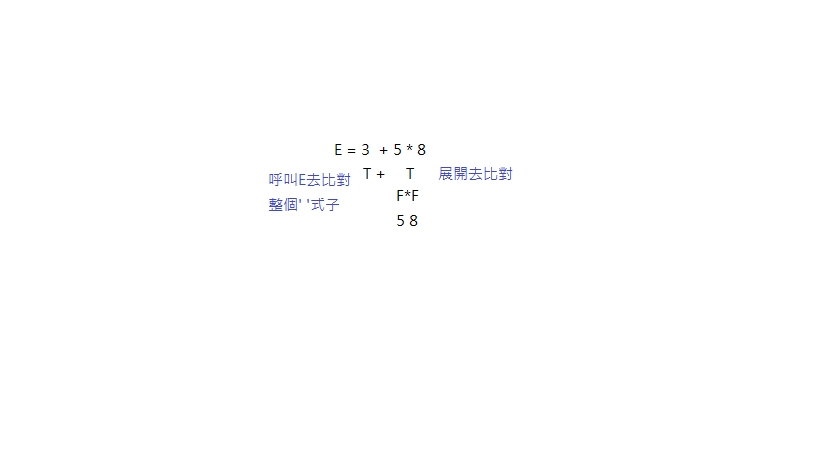
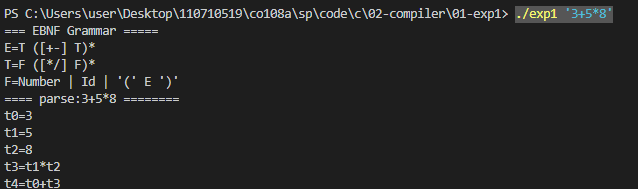
***exp1***

 printf("E=T ([+-] T)\*\n");

越下層的運算，優先序越高

  printf("T=F ([\*/] F)\*\n");

**執行./exp1 '3+5\*8'**

**結果：**

***Compiler.c語法***

PROG = STMTS 一個程式就是一堆陳述

BLOCK = { STMTS } 一個區塊就是有{ }中間加一堆陳述

STMTS = STMT\* STMT是一堆0次以上的STMT

STMT = WHILE / BLOCK / ASSIGN

一個陳述有可能是while迴圈/BLOCK區塊/ASSIGN指定

WHILE = while ( E ) STMT

一個while迴圈是用while開頭，再接(運算式)，最後陳述

ASSIGN = id = ‘ = ‘ E 一個指定是一個變數名稱，他等於一個運算式

E = F ( OP E )\*

F = ( E ) | Number | ID

***05-compiler-run***

執行指令：

minge32-make ：編譯原始碼的建置工具

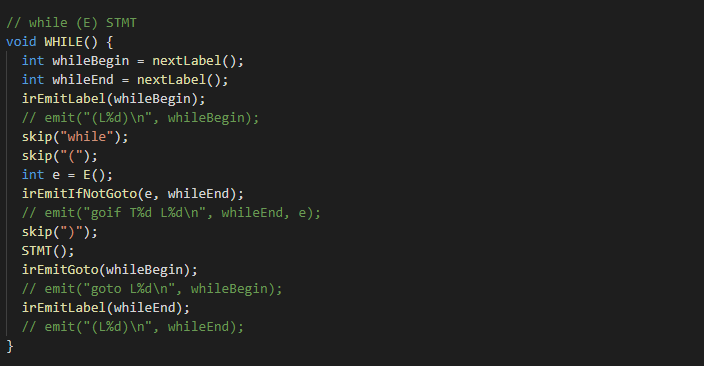
./complier test/sum.c –ir –run

只打 –ir 代表會輸出中間碼但不執行

若加上 –run 代表除了輸出中間碼，還會執行

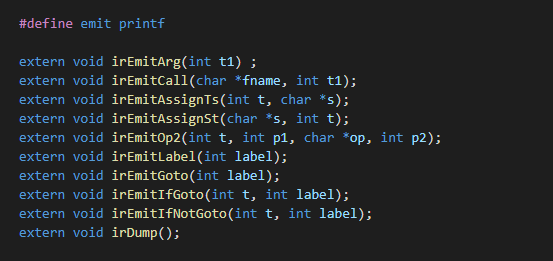
irvm.c裡的trace = printf 印出指令 = emit

\*xxxx.c檔裡面放程式碼 xxxx.h檔裡面放標頭，放define的東西



irEmitLabel：

本來是直接emit，變成直接呼叫irEmitLabel。目的是讓它格式統一，可以存在陣列裡。

typedef struct {

ir.h裡面定義的東西

  IrType type;

  int t, t1, t2, label;

  char \*s, \*op;

} IR;

t,t1,t2:臨時參數 lable:標記代號

op:做什麼動作。Ex:加法 \*s:目標參數

整個結構會儲存成一個陣列，放在ir[ ];裡面

void irEmitAssignTs(int t, char \*s) {

  irNew((IR) {.type=IrAssignTs, .op="t=s", .t=t, .s=s});

}

IrAssignTs：t = sum 之類的 t:取得代號 s:取得名字

void irDump() {

  printf("=======irDump()==========\n");

  for (int i=0; i<irTop; i++) {

    printf("%02d: ", i);

    irPrint(&ir[i]);

  }

}

irDump:把中間碼全部印出來

irvm.c:用來執行中間碼的程式

 if (eq(op, "s=t")) { //如果是s=t這樣的指令

    int \*vp = varLookup(p->s);

    \*vp = t[p->t]; //先把t取出來，再塞到新的變數

    trace("%s = t%d (%d)\n", p->s, p->t, \*vp); //印出來

  }

檔案間的關係圖：

main.c => lexer.c

          compiler.c

          ir.c       => irvm.c

main.c會呼叫compiler.c / lexer.c / ir.c

ir.c裡面會呼叫irvm.c 其他的都是呼叫對應的.h檔

***asmVm***

執行指令：

mingw32-make

./asm ../test/Add

測試範例：Add.asm

讓R0 = 2 + 3

// Computes R0 = 2 + 3

@2

D=A

@3

D=D+A

@0

M=D //讓第0個記憶體塞入2 + 3

執行結果：

00: @2 0000000000000010 0002

把hack的指令，每一個都轉成機器碼。2進位(1110…000)；16進位(ec10)

把組合語言編成機器碼的方式，就稱為組譯器

01: D=A 1110110000010000 ec10

02: @3 0000000000000011 0003

03: D=D+A 1110000010010000 e090

04: @0 0000000000000000 0000

05: M=D 1110001100001000 e308

第一階段目的：編出每一個符號的位置

void pass1(string inFile) { //輸入是一個檔案名稱

  printf("============= PASS1 ================\n");

  char line[100]="";

  FILE \*fp = fopen(inFile, "r"); //開檔案

  int address = 0;

  while (fgets(line, sizeof(line), fp)) { //開完之後就一行一行讀

    char \*code = parse(line); //如果那行是註解(如果不是就編碼)

    if (strlen(code)==0) continue; //就在這裡做continue

    printf("%02d:%s\n", address, code);

    if (code[0] == '(') {

      char label[100];

      sscanf(code, "(%[^)])", label);

      symAdd(&symMap, label, address); // 記住符號位址，給 pass2 編碼時使用

    } else {

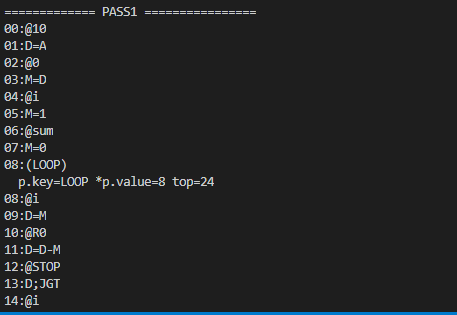
      address ++;

    }

  }

  fclose(fp);

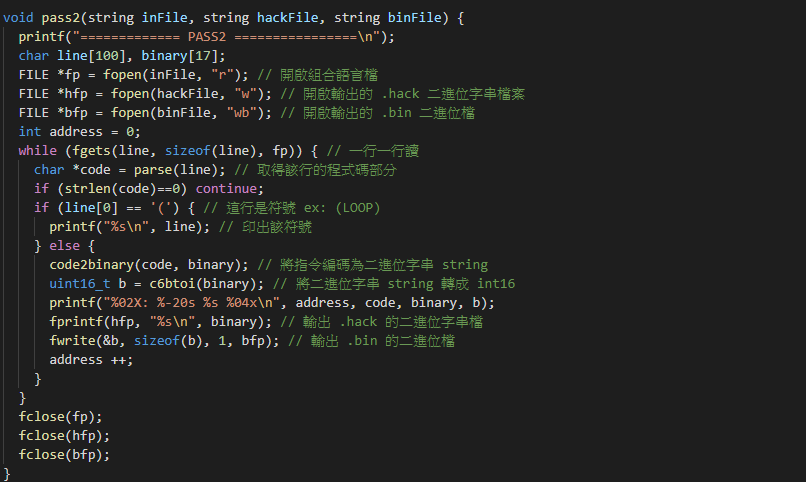
}



碰到標記(LOOP)就記住位置

p.key=LOOP是8

接下來每個指令都一直加1

第二階段：處理文字指令轉成二進位(真正編碼的動作)

因為PASS2有輸出，所以會開一個16進位的輸出檔，然後再開一個二進位的輸出檔

再一行一行讀，如果是標記就印出來，因為第一階段已經記住標記的位置

***vm.c***

執行指令：

mingw32-make

./vm ../test/Add.bin

int imTop = 0;

int16\_t im[32768], m[65536];

im : 指令記憶體，在hack cpu這台電腦裡面有兩個記憶體

指令放在指令記憶體，指令記憶體最大道32768

資料放在資料記憶體，資料記憶體基本上也是32768就夠了，但為了容易擴充，設為65536

// run: ./vm <file.bin>

int main(int argc, char \*argv[]) {

  char \*binFileName = argv[1];

  FILE \*binFile = fopen(binFileName, "rb");

  imTop = fread(im, sizeof(uint16\_t), 32768, binFile);

  fclose(binFile);

  run(im, m);

}

先把指定的檔案打開讀進來，binFileName輸入檔就是bin檔

argv[1]:取得第一個參數

im:打開之後讀進來，讀到指令記憶體裡面

fclose(binFile); :讀完就關閉檔案

run(im, m); : 把指令記憶體的程式從0開始跑

\*run做的事情：模擬機器的執行過程

int run(uint16\_t \*im, int16\_t \*m) {

  int16\_t D = 0, A = 0, PC = 0;

  uint16\_t I = 0;

  uint16\_t a, c, d, j;

在hackcpu裡有一個程式計數器pc、主要儲存位置的A暫存器、儲存資料的D暫存器。

PC = 0 :從第0個指令開始執行

\*指令執行到超過程式大小的時候會跳出一個無窮迴圈while(1)

while (1) {

    int16\_t aluOut = 0, AM = 0;

    if (PC >= imTop) {

      debug("exit program !\n");

      break;

    }

只有虛擬機才知道超過程式的範圍！

I = im[PC]; //指令記憶體目前的指令把它取出來放到I裡面

debug("PC=%04X I=%04X", PC, I); //印出現在的PC跟I是多少(影片中的debug寫的是printf)

PC ++; //取完之後PC +1

 if ((I & 0x8000) == 0) { // A 指令 檢查A指令的第一碼是不是0

A = I; } //如果指令的第一碼是0，就直接把A設成0

 else { // 如果指令的第一碼是1，就是C 指令

      a = (I & 0x1000) >> 12;

      c = (I & 0x0FC0) >>  6;

      d = (I & 0x0038) >>  3;

      j = (I & 0x0007) >>  0;

a = (I & 0x1000) >> 12 ：

取出A欄位，往右移12個，就會把它移到最右邊從1開始的地方，以此類推

***vm的指令欄位提取***

**格式：111 a c1..c6 d1..d3 j1..j3**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | c | d | j |
|  | 111a | 1234 | 5612 | 3123 |
| 0x1000 | 0001 | 0000 | 0000 | 000a |
| 0x0FC0 | 0000 | 1111 | 1100 | 0000 |
| 0x0038 | 0000 | 0000 | 0011 | 1000 |
| 0x0007 | 0000 | 0000 | 0000 | 0111 |

變成：12 == 0000 0000 0000 000a

6 == 0000 0000 00 c1 .. c6

3 == 0000 0000 0000 0 d1 .. d3

0 == 0000 0000 0000 0 j1 .. j3

AM = (a == 0) ? A : m[A];

      switch (c) { // 處理 c1..6, 計算 aluOut

        case 0x2A: aluOut = 0;      break; // "0",   "101010"

.

.

.

 case 0x15: aluOut = D | AM; break; // "D|AM","010101"

        default: assert(0);

      }

if (d & BIT(2)) A = aluOut;

if (d & BIT(1)) D = aluOut;

if (d & BIT(0)) m[A] = aluOut;

如果d的第2位元是1，就要寫入A

如果d的第1位元是1，就要寫入D

如果d的第0位元是1，就要寫入M

透過這樣可以正確的寫入到暫存器或記憶體

\*處理跳躍指令，看alu的輸出

j欄位是0，不管怎樣都不跳

其他就要看alu的out是大於0還是小於0決定

switch (j) {

        case 0x0: break;                          //

        case 0x1: if (aluOut >  0) PC = A; break; // JGT

.

.

.

case 0x6: if (aluOut <= 0) PC = A; break; // JLE

        case 0x7: PC = A; break;                  // JMP

      }

如果j欄位是0x7=111，不管怎樣都要跳(把PC設成A)

debug(" A=%04X D=%04X m[A]=%04X", A, D, m[A]);

把A,D,M印出來觀察

***03asmVm/gcc/01-add(配合main.c)***

執行指令：

gcc -fverbose-asm -S add.c -o add.s：

-fverbose-asm代表要在產生的組合語言裡面產生詳細的格式

-S是要產生組合語言的意思

所以它會把add.c轉換成組合語言add.s

gcc main.c add.c -o add

./add

gcc -S add.c -o add.s

結果：add(5, 8)=13

**add.s**

.text

.globl \_add

.def \_add; .scl 2; .type 32; .endef

\_add: //函數名稱

pushl %ebp #

此三行在做堆疊的動作

movl %esp, %ebp #,

subl $16, %esp #,

movl 8(%ebp), %eax # a, tmp89 //#後面是註解，意思是8(%ebp)=a，取ebp + 8 的記憶體內容，所以ebp + 8 = a

movl %eax, -4(%ebp) # tmp89, t // -4(%ebp)=t，取ebp – 4的記憶體內容，ebp – 4 = t

movl 12(%ebp), %eax # b, tmp90 //%eax暫存器，ebp + 12 = b

movl %eax, -8(%ebp) # tmp90, x //ebp – 8 = x

movl 8(%ebp), %edx # a, tmp91

movl 12(%ebp), %eax # b, tmp92

addl %edx, %eax # tmp91, D.1490

leave

ret

.ident "GCC: (tdm-1) 5.1.0"

**add.c：**把a和b相加回傳

int add(int a, int b) {

  int t = a, x=b; //臨時變數

  return a+b;

}

***fib.c***

執行指令：

gcc -fverbose-asm -S fib.c -o fib.s

gcc -c fib.c -o fib.o //只想編譯不想連結

gcc main.c fib.c -o fib //編譯並連結

./fib

    .file    "fib.c"

    .text

    .globl    \_fib

    .def    \_fib;    .scl    2;    .type    32;    .endef

\_fib:

    pushl    %ebp     #                 # 前置堆疊框架處理

    movl    %esp, %ebp     #,

    pushl    %ebx     #

    subl    $20, %esp     #,

    cmpl    $1, 8(%ebp)     #, n        # if n <=1

    jg    L2     #,

    movl    $1, %eax     #, D.1493      #    eax= 1  ... return

    jmp    L3     #

L2:

    movl    8(%ebp), %eax     # n, tmp93 # eax = n

    subl    $1, %eax     #, D.1493       # eax = eax - 1

    movl    %eax, (%esp)     # D.1493,   # 推入參數 n-1

    call    \_fib     #                   # 呼叫 fib(n-1)

    movl    %eax, %ebx     #, D.1493     # 取得傳回值放入 ebx

    movl    8(%ebp), %eax     # n, tmp94 # eax = n

    subl    $2, %eax     #, D.1493       # eax = eax - 2

    movl    %eax, (%esp)     # D.1493,   # 推入參數 n-2

    call    \_fib     #                   # 呼叫 fib(n-2)

    addl    %ebx, %eax     # D.1493, D.1493 # eax = fib(n-1)+fib(n-2)

L3:

    addl    $20, %esp     #,              # 堆疊後置段，恢復 ebp, esp 的值

    popl    %ebx     #

    popl    %ebp     #

    ret                                   # return

    .ident    "GCC: (tdm-1) 5.1.0"

```

***inline.c(內嵌組合語言)***

執行：

gcc inline.c -o inline

./line

結果：

sum = 30

c語言可以把組合語言直接嵌在c裡面，讓他們去做溝通

int main(int argc, char \*\*argv)

{

    int32\_t var1=10, var2=20, sum = 0;

    asm volatile ("addl %%ebx,%%eax;" //要代表一個%的話要用兩個%

                 : "=a" (sum)              /\* output: sum = EAX \*/

                 : "a" (var1), "b" (var2)  /\* inputs: EAX = var1, EBX = var2 \*/

把var1傳給a暫存器，就是eax，把var2傳給b暫存器，就是ebx。然後把ebx跟eax加起來放在eax裡面

    );

    printf("sum = %d\n", sum);

    return 0;

}