Final

不saveR7很可能导致跳不出来

Let's say that code A calls subroutine B.
A is the caller. B is the callee.

Either Caller- or Callee-Saved

(Remember: code A calls subroutine B.)

If a register might be changed by B,
- code A is responsible for
- copying the bits elsewhere before calling B.
- Such a register is caller-saved.

R7 is always caller-saved.

If subroutine B guarantees
- that a register does not change,
- that register is callee-saved.
- Note: B can still use the register, but must save and restore the original contents to do so.

Any output register is also caller-saved.

```
LOOP      LDI R0,KBSR
          BRzp LOOP  轮询等待KBSR最高位置1可以读取
          LDI R0 KBDR  从KBDR加载数据到R0
KBSR .FILL 0xFE00
KBDR.FILL 0xFE02
```

display：processor写入DDR，置状态位DSR为0，display展示输出，并把状态位设置为1，processor等待状态位变1以后写下一个字符。示例如下：

```
LOOP      LDI R0 DSR
          BRzp LOOP轮询等待DSR最高位为1
```

表示可以显示

```
          STI R_, DDR
DSR       .FILL 0xFE04
DDR       .FILL 0xFE06
```

```
      .ORIG x3000
      LD  R0,NUM5
      STI R0,DDR
      HALT
NUM5  .FILL x35      ; ASCII digit '5'
DDR   .FILL xFE06
      .END
```

Answer: does not wait for display to be ready

将一个task分成不同的composition：
sequential decomposition：按顺序ABCD
conditional decomposition：有判断
iterative decomposition：迭代（计算1 to n）

lc3的一些缺点考察：
1.内存小

2. **(4 points)** A friend wants to add a 640×480-pixel monochrome (two-color) graphics adapter to his LC-3-based computer. Using **NO MORE THAN 25 WORDS**, including any necessary calculations, explain how to accomplish this goal, or why the goal is impossible.

(640 × 480 pixels × 1 bit/pixel) / 16 bits/memory location = 19,200 memory locations

LC-3 has only 512 (xFE00 to xFFFF) usable for memory-mapped I/O, so …
- **(1) Cannot map individual pixels without changes to design [as students know it], but**
- **(2) Can change board design (hardware for I/O) to expand memory-mapped I/O region, or**
- **(3) Can use one or two ports with address / data I/O model [not something students have seen, but an acceptable answer].**

2.标签跳转距离有限

```
; SEQUENCE 1                      ; SEQUENCE 2
        LD    R1,LABEL                    LD    R0,OTHER
        BRnzp LATER                       LDR   R1,R0,#0
                                          BRnzp LATER
                                  OTHER   .FILL LABEL
```

In some cases, sequence 1 may fail, while sequence 2 continues to work.
USING 30 WORDS OR FEWER, EXPLAIN WHY.

LABEL may be too far away for a 9-bit offset (LD). Sequence 2 works for any memory location.

类型	存储大小	值范围
char	1 字节	-128 到 127 或 0 到 255
unsigned char	1 字节	0 到 255
signed char	1 字节	-128 到 127
int	2 或 4 字节	-32,768 到 32,767 或 -2,147,483,648
unsigned int	2 或 4 字节	0 到 65,535 或 0 到 4,294,967,295
short	2 字节	-32,768 到 32,767
unsigned short	2 字节	0 到 65,535
long	4 字节	-2,147,483,648 到 2,147,483,647
unsigned long	4 字节	0 到 4,294,967,295

lc3是16位操作系统，一个地址16bit

Format Specifier	Interpretation
%c	store one ASCII character (as char)
%d	convert decimal integer to int
%f	convert decimal real number to float
%lf	convert decimal real number to double

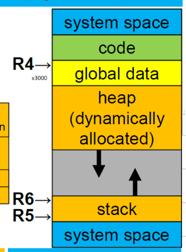
Format Specifier	Interpretation
%u	convert decimal integer to unsigned int
%x or %X	convert hexadecimal integer to unsigned int

With LC-3, code starts at x3000 by convention.

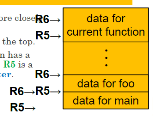
And program data is mapped into memory after the code.

With LC-3, we have
- trap vector table at x0000 to x00FF,
- interrupt/exception table at x0100 to x01FF, and
- OS code (trap subroutines).

The starting location depends on the size of the program (and data). The heap grows downward into unused memory.



Let's look more closely at the stack. R6 points to the top. Each function has a stack frame. R5 is a frame pointer.



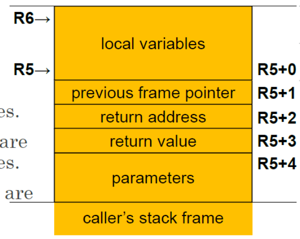
R6指向栈顶部
R5栈帧指针

With LC-3, the base can be xFE00, allowing the stack to grow into unused memory.

With LC-3, we have memory-mapped I/O at xFE00 to xFFFF.

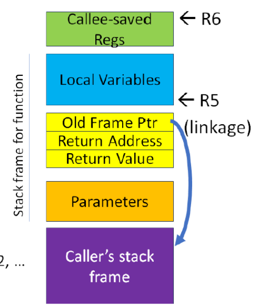
在system space中有512个

R6 points to top of stack.
R5 points to bottom of local variables.
R5+0, -1, … are local variables.
R5+4, +5, … are parameters.



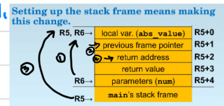
Stack Frames in LC-3

- Function needs to create space for local variables and linkage
- Set R5 to point to bottom of local variables – this is called the frame pointer
- Parameters start at +4, then +5, …
- Local variables start at +0, then -1, -2, …

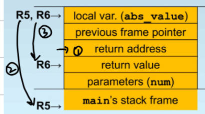



```
LDR R0,R4,#0   load 全局变量
push{ ADD R6,R6,#-1  从上一个stack步入当前
      STR R0,R6,#0
JSR FIND_ABS
pop{ LDR R0,R6,#0   调用后R6指向return value
     ADD R6,R6,#2   重新指向上一个stack
STR R0,R4,#0
```

Setting up the stack frame means making this change.

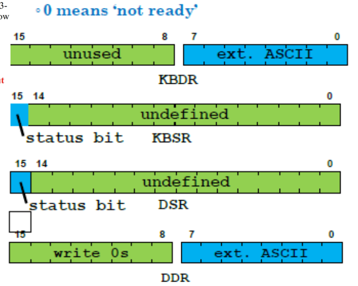



```
FIND_ABS
①ADD R6,R6,#-4
②STR R5,R6,#1   存R6用R6
③ADD R5,R6,#0
④STR R7,R5,#2   存R7用R5
```




```
①LDR R7,R5,#2   回收R7用R5
②LDR R5,R5,#1
ADD R6,R6,#3 → pop
RET
```

- 1 means 'ready'
- 0 means 'not ready'



c转lc3经典

```
typedef struct double_list_t double_list_t;
struct double_list_t {
    double_list_t* prev;  // previous element of list
    double_list_t* next;  // next element of list
};
```

+0	prev
+1	next

1. **(10 points)** Implement the list insertion code shown below as an LC-3 assembly subroutine. The diagram to the right of the code shows the stack on entry to your subroutine.
   - Do NOT set up a stack frame.
   - Use **NO MORE THAN SEVEN INSTRUCTIONS** (not counting RET, provided for you).
   - Your code may **change only R0, R1, and R2**.
   - **Do not change R6**—the subroutine returns `void`.
   - *Hint: if you put the right values into the three registers, you need only one instruction per line of C code.*

```
void
dl_insert (double_list_t* head, double_list_t* elt)
{
    elt->next = head->next;
    elt->prev = head;
    head->next->prev = elt;
    head->next = elt;
}
```

R6 -> | head | elt |

```
DL_INSERT   LDR R0,R6,#0     ; head
            LDR R1,R6,#1     ; elt
            LDR R2,R0,#1     ; head->next
            STR R2,R1,#1
            STR R0,R1,#0
            STR R1,R2,#0
            STR R1,R0,#1
            RET
```

2. **(10 points)** Implement the code shown below to find the first element of a list as an LC-3 assembly subroutine. The diagram to the right of the code shows the stack on entry to your subroutine.
   - Do NOT set up a stack frame.
   - Use **NO MORE THAN TEN INSTRUCTIONS** (not counting RET, provided for you).
   - Your code may **change only R0, R1, R2, R3, and R6**.
   - **Be sure to push the return value on top of the stack.**

```
void*
dl_first (double_list_t* head)
{
    return (head == head->next ? NULL : head->next);
}
```

R6 -> | head |

```
DL_FIRST    LDR R0,R6,#0     ; head
            LDR R1,R0,#1     ; head->next
            NOT R1,R1
            ADD R1,R1,#1
            ADD R1,R1,R0
            BRz EQUAL        ; 0 is NULL
            LDR R1,R0,#1
EQUAL       ADD R6,R6,#-1
            STR R1,R6,#0

            RET
```

```
typedef struct dl_t dl_t;
struct dl_t {
    dl_t* prev; // previous element in the list
    dl_t* next; // next element in the list
};
```

+0	prev
+1	next

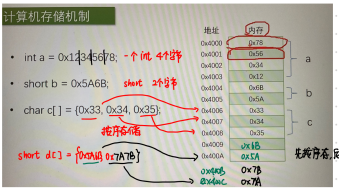
1. **(10 points)** Implement the function `dl_length` shown below as an LC-3 assembly subroutine. The diagram to the right of the code shows the stack on entry to your subroutine.
   - Your code may **change only R0, R1, R2 and R3**.
   - Do **NOT** set up a stack frame. The local variable `count` can be kept in a register of your choice (R0-R3).
   - **USE 15 OR FEWER INSTRUCTIONS** (not counting `RET`, provided for you).
   - **Push the return value onto stack** before returning.

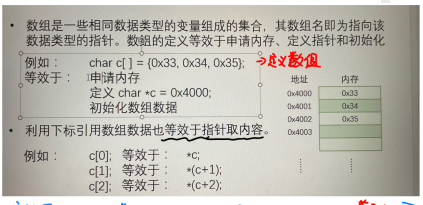
R6 → | head |

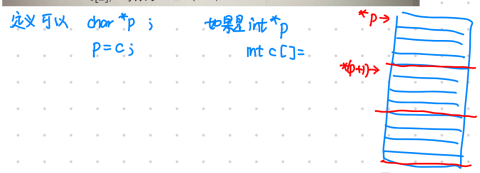
```
int16_t dl_length (dl_t* head) {
    int16_t count = 0;
    for (dl_t* elt = head->next; elt != head; elt = elt->next) {
        ++count;
    }
    return count;
}
```

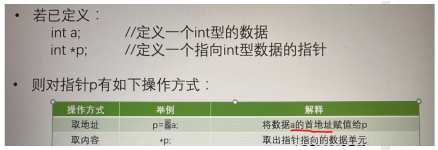
```
DL_LENGTH   AND   R0,R0,#0    ; count
            LDR   R1,R6,#0    ; head
            LDR   R2,R1,#1    ; elt
            NOT   R1,R1       ; R1 <- -head
            ADD   R1,R1,#1
LOOP        ADD   R3,R1,R2    ; R3 <- elt - head
            BRz   DONE
            ADD   R0,R0,#1    ; count++
            LDR   R2,R2,#1    ; elt = elt->next
            BRnzp LOOP
DONE        ADD   R6,R6,#-1   ; return count
            STR   R0,R6,#0
```

指针









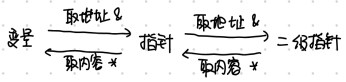
指针变量存某个数据单元首地址.
指针指向这个数据单元. 1个字节 8个位.

数据类型		指向该数据类型的指针	
(unsigned) char	1字节	(unsigned) char *	x字节
(unsigned) short	2字节	(unsigned) short *	x字节
(unsigned) int	4字节	(unsigned) int *	x字节
(unsigned) long	4字节	(unsigned) long *	x字节
float	4字节	float *	x字节
double	8字节	double *	x字节

- 16位系统：x=2，32位系统：x=4，64位系统：x=8

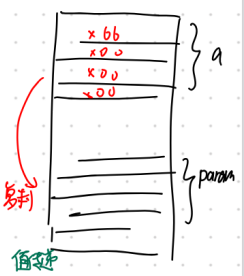
注：如果写 int *p,p1; 表示只有p是指针. p1是int类型.

```
printf("%d", sizeof(p))
```
输出结果显示8 说明 int* 占8字节



值传递和地址传递关于const

```
#include <stdio.h>

void fun(int param)
{
    param=0x88;
    printf("%x\n",param);
}

int main(void)
{
    int a=0x66;

    fun(a);
    printf("%x\n",a);

    return 0;
}
```

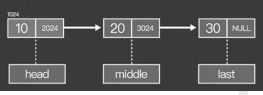


∴ 打出 0x88
0x66 a 不会变

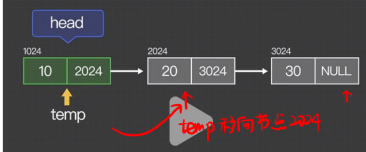
```
#include <stdio.h>
int FindMax(int *array)
{

}

int main(void)
{
    int a[]={1,2,3,5,4,3};
    int Max;

    Max=FindMax(a);

    printf("Max=%d\n",Max);

    return 0;
}
```

若这里声明 const int* array 只能读不能改
子函数无法赋值给 array
a 数组名本来就是首地址


```
int data;
struct LNode * next;
};

struct LNode *head, *middle, *last;

head   = (LNode *)malloc(sizeof(struct LNode));
middle = (LNode *)malloc(sizeof(struct LNode));
last   = (LNode *)malloc(sizeof(struct LNode));

head->data = 10;
middle->data = 20;
last->data = 30;

head->next = middle;
middle->next = last;
last->next = NULL;
```

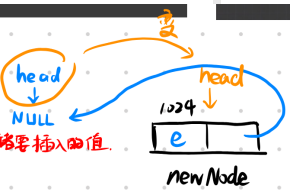



```
struct LNode *temp = head;
while (temp != NULL) {
  printf("%d ", temp->data);
  temp = temp->next;
}
```
创建临时指针 temp. 指头节点. 把址分配给他



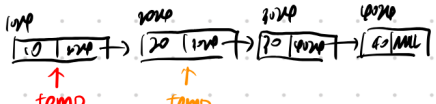
链表头插法

```
struct LNode *head = NULL;

void insertHead (int e) {
  struct LNode *newNode = (LNode *)malloc(sizeof(struct LNode));
  newNode->data = e;
  newNode->next = head;
  head = newNode;
}
```
指针 head 目前链表为空链表. 其头指针也为 NULL
① 创建链表节点并存储要插入的值.
② 赋值.
③ 让新节点的 next 指向当前 head
④ 让 head 指向新节点



插入顺序 10. 20. 30. 链表顺序 30. 20. 10.

搜索:

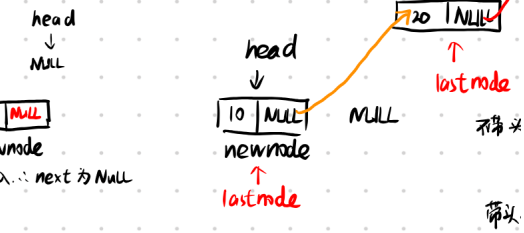



```
struct LNode* searchNode (struct LNode *head, int key) {
  struct LNode *temp = head;
  while (temp != NULL) {
    if (temp->data == key) {
      return temp;
    }
    temp = temp->next;
  }
  return NULL;
}
```
temp 数据域与 key 对比
temp 右移

要搜索的值记为 key
若 key=30

链表尾插法:

```
struct LNode *head;

void inserLast (int e) {
  struct LNode *newNode;
  newNode = (LNode *)malloc(sizeof(struct LNode));
  newNode->data = e;
  newNode->next = NULL;

  if (head == NULL) {
    head = newNode;
  } else {
    struct LNode *lastNode = head;
    while (lastNode->next != NULL) {
      lastNode = lastNode->next;
    }
    lastNode->next = newNode;
  }
```
创新节点
head为null, 让新节点成为头节点
判断链表是否为空后作为
不为NULL 说明链表不为空, 有其他节点 先找链表表尾, 通过表尾插入新节点

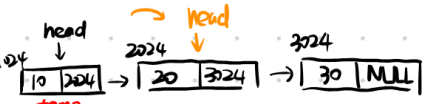


尾插入∴ next 为 NULL

插入顺序 10.20.30 ; 链表顺序为 10.20.30

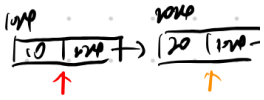
链表删除节点

```
bool deleteNode (struct LNode **head, int key) {
  struct LNode *temp;
  if ((*head)->data == key) {
    temp = *head;
    *head = (*head)->next;
    free(temp);
    return true;
  } else {
    struct LNode *pre = *head;
    while (pre->next != NULL) {
      if (pre->next->data == key) {
        temp = pre->next;
        pre->next = pre->next->next;
        free(temp);
        return true;
      } else {
        pre = pre->next;
      }
    }
    return false;
  }
```
head 后移
删除 temp. 删 head 初始位.
pre
找到待删节点或最后一个节点为止.
跳过待删节点



key: 待删节点
先使用临时变量 temp. 保持 head 初始位置.

搜索:




```
struct LNode* searchNode (struct
  struct LNode *temp = head;
  while (temp != NULL) {
    if (temp->data == key) {
      return temp;
    }
    temp = temp->next;
```