HW4 1-9 & HW 5

HW4 T1

Recall the machine busy example from Example 2.11 in Section 2.6.7. Assuming the BUSYNESS bit vector is stored in R2, we can use the LC-3 instruction 0101 011 010 1 00001 (AND R3, R2, #1) to determine whether machine 0 is busy or not. If the result of this instruction is 0, then machine 0 is busy.

- Write an LC-3 instruction that determines whether machine 2 is busy.
- Write an LC-3 instruction that determines whether both machines 2 and 3 are busy.
- Write an LC-3 instruction that determines whether all of the machines are busy.
- Can you write an LC-3 instruction that determines whether machine 6 is busy? Is there a problem here?

BUSYNESS bit vector: 8bits, each bit represent 1 machine

- AND R3 R2 #4   (00100)
- AND R3 R2 #12 (01100)
- AND R3 R2 #31 (11111)
- LC-3立即数最大5位，无法表示100000。若希望实现则需要先将mask加载到某个Register中，再做AND

HW4 T2

Suppose the following LC-3 program is loaded into memory starting at location x30FF.

If the program is executed, what is the value in R2 at the end of execution?

Address	Value
x30FF	1110 0010 0000 0001
x3100	0110 0100 0100 0010
x3101	1111 0000 0010 0101
x3102	0001 0100 0100 0001
x3103	0001 0100 1000 0010

上面的机器码对应的为

```
    LEA R1, #1
    LDR R2, R1, #2
    TRAP x25
    x1441
    x1482
```

对应的执行过程为 x3100 + 1 -> R1，[R1 + 2] -> R2，得到最后的结果为x1482

HW4 T3

The LC-3 ISA contains the instruction `LDR DR, BaseR, offset`. After the instruction is decoded, the following operations (called microinstructions) are carried out to complete the processing of the LDR instruction:

```
MAR <- BaseR + SEXT(offset6) ; set up the memory address

MDR <- Memory[MAR] ; read mem at BaseR + offset

DR <- MDR ; load DR
```

Suppose that the architect of the LC-3 wanted to include an instruction `MOVE DR, SR` that would copy the memory location with address given by SR and store it into the memory location whose address is in DR.

1. The MOVE instruction is not really necessary since it can be accomplished with a sequence of existing LC-3 instructions. What sequence of existing LC-3 instructions implements (also called "emulates") `MOVE R0,R1`? (You may assume that no other registers store important values.)
2. If the MOVE instruction were added to the LC-3 ISA, what sequence of microinstructions, following the decode operation, would emulate `MOVE DR,SR`?

HW4 T3

MOVE的功能为从SR获得地址A，从地址A读取内容存到DR内部存储的地址所对应的地址空间

用以下两条指令即可实现MOVE

```
LDR R2, R1, #0
STR R2, R0, #0
```

解析为微指令则对应

```
; 读取到MDR
MAR <- SR
MDR <- Memory[MAR]
; 从MDR存储到内存
MAR <- DR
Memory[MAR] <- MDR
```

HW4 T4

The LC-3 does not have an opcode for XOR, so we're required to write instructions to implement the XOR operation by ourselves. Assume that the reserved opcode 1101 is implemented as OR instruction, which shares the same format as AND instruction.

The following instructions will store the value of (R1 XOR R2) to R3 (XOR R3, R1, R2). Fill in the two missing instructions to complete the program. You are only allowed to use the registers R1, R2, R3, and R4.

Address	**Instruction**
x3000	1001 100 001 111111
x3001	
x3002	1001 011 010 111111
x3003	
x3004	1101 011 011 000 100

HW4 T4

题目中已知的三条指令分别为

$$
\begin{aligned}
p \oplus q &= (p \wedge \neg q) \vee (\neg p \wedge q) = p\bar{q} + \bar{p}q \\
&= (p \vee q) \wedge (\neg p \vee \neg q) = (p+q)(\bar{p}+\bar{q}) \\
&= (p \vee q) \wedge \neg(p \wedge q) = (p+q)(\overline{pq})
\end{aligned}
$$

```
NOT R4，R1
* UNKNOWN
NOT R3，R2
* UNKNOWN
OR  R3，R4，R3
```

为了表示XOR R3，R1，R2 考虑XOR的几种用OR AND NOT的表示，按照如下逻辑从最后一条反向推测

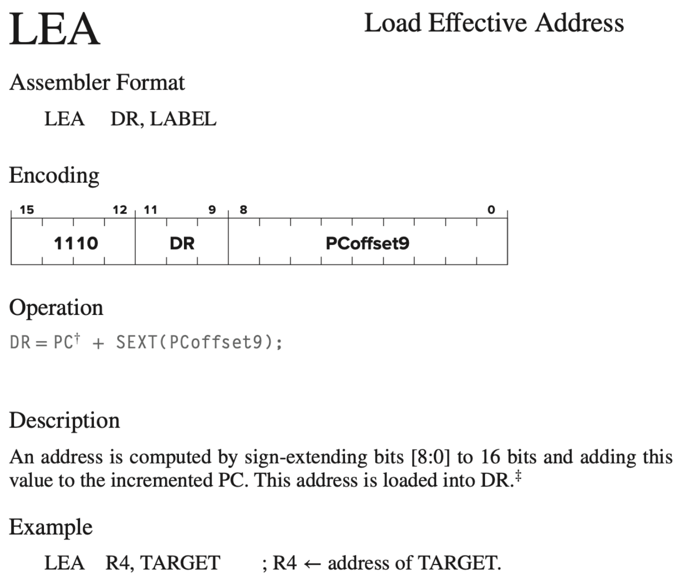
```
R3 <- R4 OR R3
R4 <- ( (NOT Ra) AND Rb )  R3 <- ( Ra AND (NOT Rb) )
R3 <- NOT R2 => b == 2
R4 <- NOT R1 => a == 1
```

由此可以推测得到，第一条未知语句为AND R4,R4,R2 (R4 = NOT R1)，第二条为AND R3,R3,R1 (R3 = NOT R2)

HW4 T5

List five addressing modes in LC3. Given instructions ADD, NOT, LEA,LDR and JMP, categorize them into operate instructions, data movement instructions, or control instructions. For each instruction mentioned above, list addressing modes that can be used.

Instruction	Type	Addr mode(s)
ADD	operate	register, immediate
NOT	operate	register
LEA	**data movement**	**immediate**
LDR	data movement	Base+offset
JMP	control	register



LEA Load Effective Address

Assembler Format

LEA DR, LABEL

Encoding

15–12	11–9	8–0
1110	DR	PCoffset9

Operation

```
DR = PC† + SEXT(PCoffset9);
```

Description

An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. This address is loaded into DR.‡

Example

LEA R4, TARGET ; R4 ← address of TARGET.

HW4 T6

1. Write a single LC3 assembly instruction that copies the content of R5 to R4.
2. Write a single LC3 assembly instruction that clears the content of R3. (i.e. R3 = 0)
3. **Write 3 LC3 assembly instructions that does R1=R6-R7.**
   1. **You are ONLY allowed to change the value of R1.**
   2. **You may assume that the initial value of R1 is 0.**
4. Write 3 LC3 assembly instructions that multiply the value at label DATA by 2. (Mem[DATA] = Mem[DATA] * 2)
   1. You are ONLY allowed to change the value of R1.
   2. You don't need to restore or clear the value of the register you used.
   3. No need to consider overflow.
5. **Set condition codes based on the value of R1 using only one LC-3 instruction.**
   1. **You are not allowed to change any value in the registers.**

HW4 T6

1. 略
2. 目标为清零R3，最简单的做法就是与0
3. 注意题目，**只允许修改R1的值**
   `NOT R1, R7, ADD R1, R1, #1, ADD R1, R1, R6`
4. **这道题也只允许修改R1的值**
   `LD R1, DATA, ADD R1, R1, R1, ST R1, DATA`
5. **注意本题不允许修改值**但是又希望根据R1的值设置条件位，由于条件位根据指令结果设置，故可以考虑对R1做不修改R1的计算实现，例如加0或者与-1（补码全1）
   `ADD R1, R1, #0 or AND R1, R1, #-1`

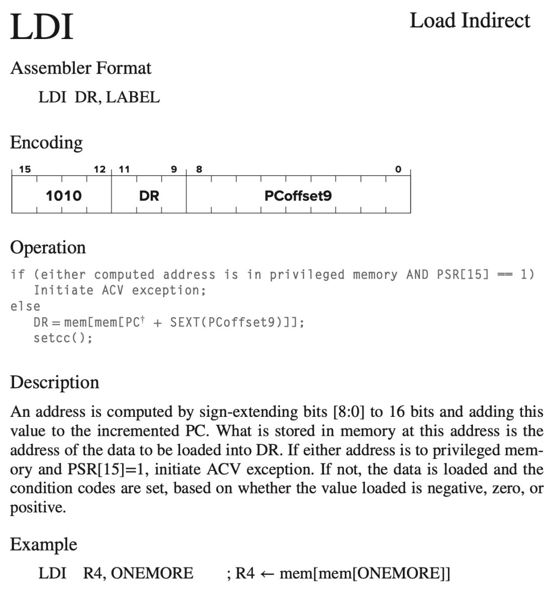
HW4 T7

If the current PC points to the address of an JMP instruction, how many memory accesses are required for the LC-3 to process that instruction? What about ADD and LDI instructions?

JMP: **只涉及一次取指**

ADD: **只涉及一次取指**，随后的计算均在寄存器中完成

LDI: 取指需要一次，随后根据LDI的功能完成两次访存操作，**共计三次**



LDI   Load Indirect

Assembler Format

    LDI DR, LABEL

Encoding

15–12	11–9	8–0
1010	DR	PCoffset9

Operation

```
if (either computed address is in privileged memory AND PSR[15] == 1)
   Initiate ACV exception;
else
   DR = mem[mem[PC† + SEXT(PCoffset9)]];
   setcc();
```

Description

An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. What is stored in memory at this address is the address of the data to be loaded into DR. If either address is to privileged memory and PSR[15]=1, initiate ACV exception. If not, the data is loaded and the condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

    LDI   R4, ONEMORE   ; R4 ← mem[mem[ONEMORE]]

HW4 T8

The content in PC is x3010. The content of the following memory unit is as follows:

1. After the execution of the following code, What is the value stored in R6?

2. Can you use one LEA instruction to do the same task as the three instructions above do? (Only consider loading value into R6.)

Address	Value
x304E	x70A4
x304F	x70A3
x3050	x70A2
x70A2	x70A4
x70A3	x70A3
x70A4	x70A2

Address	Value
x3010	1110 0110 0011 1110
x3011	0110 1000 1100 0001
x3012	0110 1111 0000 0001
x3013	0110 1101 1111 1111

1. 本题中指令的具体行为

先使用 LEA 指令，偏移量为 x3E，计算得到地址 0x304F 放入 R3.

再使用 LDR 指令，偏移量为 x01，从 R3 中读取地址计算，得到地址 0x3050，读取内存将 0x70A2 放入 R4.

再使用 LDR 指令，偏移量为 x01，从 R4 中读取地址计算，得到地址 0x70A3，读取内存将 0x70A3 放入 R7.

再使用 LDR 指令，偏移量为 11 1111，从R7中读取地址计算，得到地址 0x70A2，读取内存将 0x70A4 放入 R6.

2. 若使用 LEA，启动时PC为x3010，需要的偏移量为(x70A4 - (x3010 + x1)) = x4093，而LEA的偏移量为9位，最大能够表示0xFF，无法计算得到目标的x70A4

HW4 T9

After the execution of the following code, the value stored in R0 is 12. Please speculate what the value stored in R5 is like.

表格中的机器码可以被转换为下面的伪代码

```
r0 = r0 & 0;
r7 = r7 & 0;
r6 = r0 + 1;
do {
  r6 = r6 + r6;
  r4 = r5 & r6;
  if (r4 != 0) {
    r0 += 3;
  }
  r7 = r7 + 2;
  r1 = r7 - 14;
} while(r1 < 0);
r7 = r7 & 0;
```

Address	Value
x3000	0101 0000 0010 0000
x3001	0101 1111 1110 0000
x3002	0001 1100 0010 0001
x3003	0001 1101 1000 0110
x3004	0101 1001 0100 0110
x3005	0000 0100 0000 0001
x3006	0001 0000 0010 0011
x3007	0001 1111 1110 0010
x3008	0001 0011 1111 0010
x3009	0000 1001 1111 1001
x300A	0101 1111 1110 0000

HW5 T1 T2

略

HW5 T3

The following program has an error in it. What is the error? How would you fix it?

在本题中A和B处存储的值将会被作为指令执行

xDEAD -> 0b1101 111010101101，OPCode为1101对应保留指令出错，
修正方式为将A和B移动到HALT和.END之间即可，此时A和B便能被正确地
作为LABEL使用

```
    .ORIG x3000
A   .FILL xDEAD
B   .FILL xBEEF
    LD R0, A
    ST R0, B
    HALT
    .END
```

HW5 T4

Suppose you write two separate assembly language modules that you expect to be combined by the linker. Each module uses the label AGAIN, and neither module contains the pseudo-op .EXTERNAL AGAIN. Is there a problem using the label AGAIN in both modules? Why or why not?

当没有.EXTERNAL的时候，程序只会在自己的符号表中查找AGAIN的地址，并不会有冲突。

HW5 T5

Your friend has just written a simple program intended to calculate complements, which is as follows: However, it does not seem to be reliable for some reason...

Questions:

- What's the 2's complement of xF001 in hex?
- Will the program store the complement to DATA?
- What will happen afterwards? Why?

```
      .ORIG x3000
      ; Simple program that should calculate
      ; complement of DATA and store the result back
      LD R2, DATA
      NOT R2, R2
      ADD R2, R2, #1
      ST R2, DATA
DATA .FILL xF001
      .END
```

xF001 -> 0b1111 0000 0000 0001 -反-> 0b0000 1111 1111 1110 -补-> 0b 0000 1111 1111 1111 -> 0xFFFF

按照程序的过程，四行代码分别为从DATA加载到R2，对R2内的值取反，然后加一，最后存回DATA

接下来被存到DATA位置的0x0FFF被解析为 `BRnzp #-1`，无限循环

HW5 T6

What's the difference between pseudo-ops .FILL, .BLKW and .STRINGZ in LC3?

本题为概念题

.FILL 向当前地址存放一个16位数据

.BLKW 预留数个空间

.STRINGZ 预留字符串长度 + 1个空间，+1对应字符串结尾符号

HW6 T7

It is often useful to find the midpoint between two values. For this problem, assume A and B are both even numbers, and A is less than B. For example, if A = 2 and B = 8, the midpoint is 5. The following program finds the midpoint of two even numbers A and B by continually incrementing the smaller number and decrementing the larger number. You can assume that A and B have been loaded with values before this program starts execution.

Your job: Insert the missing instructions.

```
     .ORIG x3000
     LD R0, A
     LD R1, B
X    _______________ (a)
     _______________ (b)
     ADD R2, R2, R1
     _______________ (c)
     ADD R1, R1, #-1
     _______________ (d)
     BRnzp X
DONE ST R1,C
     TRAP x25
A    .BLKW 1
B    .BLKW 1
C    .BLKW 1
     .END
```

分析:

首先能确定A被存储在R0中，B被存储在R1中

已知的几行代码分别为将R2和R1的值相加放入R2中；R1自减1；无条件跳转到X处。DONE对应的是将R1存储到C，即可以推测最后的结果将会被存储在R1中。

根据上面的信息可以知道c及以上的部分为判断，故d对应更新R0的值为了判断二者相等，需要对二者相减，故a和b将R0中的值转换为负数补码并与R1相加，在c处判断是否跳转，判断是否相等使用z条件码。

HW6 T8

We all know that we can achieve left-shift by adding the number to itself. For example, ADD R0, R0, R0 will left-shift R0 by 1 bit. However, **right-shift** is not that easy. Complete the following LC3 program so that it will **right-shift R0 by 1 bit**. Note that some comments have been deleted.

分析：

根据不完整的程序可以判断程序可能的右移实现为使用一个循环计算右移

观察主循环，BRp L显然是判断循环是否结束，初始化区中明示了循环变量存储在R2中，d执行后结果为0或负则跳出循环，则能够知道d的目的是R2的自减，对应ADD R2, R2, #-1，故可以将程序转换为下面这样的伪代码

```
R1 = 0;
R2 = 15;
R3 = ( a );
R4 = 1;
R5 = 0;

for (; R2 > 0; R2 --) {
  R5 = R3 & R0;
  if ( R5 ( b ) ){
  } else {
    R1 += R4;
  }
  R3 ( c )
  R4 <<= 1;
}
```

```
 .ORIG x3000
    ; Suppose R0 is already loaded with the target number
    ; Initialize
    AND R1, R1, #0      ; Result
    ADD R2, R1, #15     ; Loop var i
    ADD R3, R1, #__ (a) ; 1 << (**DELETED**)
    ADD R4, R1, #1      ; 1 << (15 - i)
    AND R5, R5, #0      ; Temp result
    ; Main Loop
L   AND R5, R3, R0      ; Test bit
    BR___ (b) N         ; **DELETED**
    ADD R1, R1, R4      ; Add to result
N   ADD R3, __, __ (c)  ; **DELETED**
    ADD R4, R4, R4      ; L-shift R4
    ADD __, __, __ (d)  ; **DELETED**
    BRp L
    ; End
    HALT
    .END
```

HW6 T9

The following operations are performed on a stack:

- What dose the stack contain after the PUSH H ?
- At which point does the stack contain the most element?

```
PUSH A
PUSH B
POP
PUSH C
POP
PUSH D
PUSH E
PUSH F
POP
PUSH G
POP
POP
POP
PUSH H
```

Without removing the element left in the stack from the previous operations, we change this stack to a queue
(the front of queue is the top of stack), and perform

- What does the stack contain now?

```
ENQUEUE I
DEQUEUE
ENQUEUE J
ENQUEUE K
DEQUEUE
ENQUEUE L
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
ENQUEUE M
DEQUEUE
```

HW5 T10

Write a function that implements another stack function, PEEK. PEEK returns the value of the top element of the stack **without removing the element from the stack**. The return value is stored in R0, so you don't need to save R0. PEEK should also do underflow error checking: if an underflow occurs, you should output the string "Stack underflow error" and halt. (Suppose the pointer of top of the stack is in R6, and the stack can only take up the memory space from x3FFF to x3FF0)

实现PEEK实际上只需要注意下溢出检查即可，答案里面的做法是使用当前栈顶指针与x4000比较，答案的判断没有考虑栈顶指针地址小于x4000的情况。

发生下溢出错误时打印字符串后即可直接HALT，未出错则可以直接使用LDR从R6保存地址指向的内存空间取值即可。

答案里面考虑了对R1和R2的恢复，但是只要实现了核心的检查和取值就算作正确。